Progress in IS

Wei Li
Michael N. Huhns
Wei-Tek Tsai
Wenjun Wu *Editors*

Crowdsourcing

Cloud-Based Software Development

Springer

Progress in IS

More information about this series at http://www.springer.com/series/10440

Wei Li · Michael N. Huhns
Wei-Tek Tsai · Wenjun Wu
Editors

Crowdsourcing

Cloud-Based Software Development

Springer

Editors
Wei Li
State Key Laboratory of Software Development Environment, School of Computer Science and Engineering
Beihang University
Beijing
China

Michael N. Huhns
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC
USA

Wei-Tek Tsai
Department of Computer Science and Engineering, School of Computing, Informatics and Decision Systems Engineering
Arizona State University
Tempe, AZ
USA

Wenjun Wu
School of Computer Science and Engineering
Beihang University
Beijing
China

ISSN 2196-8705 ISSN 2196-8713 (electronic)
Progress in IS
ISBN 978-3-662-47010-7 ISBN 978-3-662-47011-4 (eBook)
DOI 10.1007/978-3-662-47011-4

Library of Congress Control Number: 2015938742

Springer Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer-Verlag GmbH Berlin Heidelberg is part of Springer Science+Business Media (www.springer.com)

Foreword

The last few years have seen a sea change in how computer science proceeds as a profession. This change is motivated not only by advances in technology and innovations in business models, but also by the emergence of new attitudes toward technology, business, and work. In part, this has to do with lowered barriers to entry into the profession; in part, with a new generation of software developers coming of age, many of whom are not formally trained as computer scientists; in part, with businesses pushing toward open innovation; and in part, with the continuing internationalization of the economy and contributions by active participants from virtually anywhere.

Regardless of the underlying causes, we are now seeing a new pattern of software development gaining prominence. In this pattern, a programming project is broken down into pieces of work; each piece is performed by one or more people; the results come together; and the project is assembled. This, the idea of *software crowdsourcing*, is the theme of this book.

A difference between software crowdsourcing and traditional crowdsourcing is the presence of significant structure in the problems being solved, as well as in the teams drawn from the "crowd" that solve these problems.

A difference between software crowdsourcing and traditional software contracting is that each step of the above-mentioned pattern now may be achieved in a proactive and team-oriented manner. For example, the selection of a problem to solve, its breakdown into subproblems mapped to concrete tasks, the assignment of tasks to different teams, the testing of their solutions or products, and the assembly of the products into a solution to the original problem are all tasks that the crowd would accomplish, potentially with limited or no control from a central party.

The chapters in this book address the numerous challenges to be overcome in advancing the vision of software crowdsourcing and helping make it a dominant approach in software development. In skimming a draft of this book, I see chapters that take up major challenges.

- "Crowdsourcing for Large-Scale Software Development" and "The Five Stages of Open Source Volunteering" provide a historical perspective and tutorial

material on the key concepts and best practices of software crowdsourcing, such as they are established today.

- Some of these challenges apply to Internet applications in general, but are made more acute in the case of software development. In this category I would include the selection of workers, trust, and reputation, and broadly the nature of work. "Worker-centric Design for Software Crowdsourcing: Towards Cloud Careers" introduces these concepts.
- Some challenges pertain to new models for collaboration, including interaction protocols, human participation, incentives, and team work as are required for this setting. "Bootstrapping the Next Generation of Social Machines"–"An Evolutionary and Automated Virtual Team Making Approach for Crowdsourcing Platforms" address these challenges through approaches that introduce a broad range of concepts.
- Some challenges pertain to motivating humans to participate actively and positively. "Collaborative Majority Vote: Improving Result Quality in Crowdsourcing Marketplaces" describes a voting method as a basis for crowd-based quality assurance and "Towards a Game Theoretical Model for Software Crowdsourcing Processes" an incentive mechanism for encouraging high quality.
- Some challenges apply to software development in general. Here, I would include "TRUSTIE: A Software Development Platform for Crowdsourcing", which describes a software environment that realizes some of the concepts introduced in "Crowdsourcing for Large-Scale Software Development", "The Five Stages of Open Source Volunteering", and "Worker-centric Design for Software Crowdsourcing: Towards Cloud Careers".
- Some challenges apply to maintaining communities of participants as sustainable ecosystems. In this category I would include "Social Clouds: Crowdsourcing Cloud Infrastructure" and its notion of a Social Cloud.
- Some challenges become clearer when one attempts to pull together various technical ideas into deployed software systems. "Recommending Web Services Using Crowdsourced Testing Data" shows how to predict the quality of service offered by a web service instance based on crowdsourced performance data. "A Cloud-based Infrastructure for Crowdsourcing Data from Mobile Devices" illustrates the practical challenges in connection with a crowd-sensing application.

The book is a timely contribution to computer science that is at once both practical and scholarly. I applaud and congratulate the authors and editors on a job well done. Enjoy!

Munindar P. Singh
North Carolina State University
Raleigh, NC, USA

Contents

Overview

Summary of the Book

This book, *Cloud-Based Software Crowdsourcing*, brings together research efforts on many areas such as software engineering, service oriented computing, social networking and cloud computing, which are driving and shaping an emerging research field C software crowdsourcing. In the chapters of this book, you will find the perspectives of pioneering researchers on the fundamental principles, software architecture, development process, and a cloud-based architecture to support distributed software crowdsourcing.

Crowdsourcing software development or software crowdsourcing is an emerging software engineering approach. Software development has been outsourced for a long time, but the use of a cloud to outsource software development to a crowd of developers is new. All software development tasks can be crowdsourced, including requirements, design, coding, testing, evolution, and documentation. Software crowdsourcing practices blur the distinction between end users and developers, and follow the co-creation principle, i.e., a regular end-user becomes a co-designer, co-developer, and co-maintainer. This is a paradigm shift from conventional industrial software development, with developers distinct from users, to a crowdsourcing-based peer-production software development in which many users can participate. A cloud provides a scalable platform with sufficient resources, including computing power and software databases, for a large crowd of developers. With the increasingly powerful cloud software tools, it significantly reduces the amount of manual labor needed in setting up software production environments, thus empowering peer developers to perform software crowdsourcing tasks efficiently in design, coding, and testing. By taking advantage of the elastic resource provision and infrastructure, software crowdsourcing organizers can swiftly orchestrate distributed and large-scale development over highly dynamic communities.

Preliminary crowdsourcing practices and platforms including Apples App Store, TopCoder demonstrate this advantage of crowdsourcing in terms of software ecosystem expansion and product quality improvement. Recently, multiple seminars and workshops have been held to start theoretical and empirical studies on software

crowdsourcing. Many open questions need to be explored: What are the tenets for the crowdsourcing development of socio-technical ecosystems? What are the unique characteristics of the crowdsourcing method that distinguishes it from other classic software development methods? What can one align the software architecture with crowdsourcing organization of software ecosystems? How can one govern the social structure of the socio-technical ecosystem to manage the community, regulate the development activities, and balance the potential conflicts between project budget and time constraint?

This book summarizes the state-of-art research in the emerging field and introduces the important research topics including fundamental principles, theoretical frameworks and best practices of applications and systems. The book is a collection of papers written by pioneers and researchers who attended the Dagstuhl seminar in 2013. As this book is contributed by multiple authors with different perspectives, each paper will have its own unique views and notations, but collectively, these papers provide a comprehensive look of this new and exciting field.

Book Organization

The book is divided into three parts: Part I ("Crowdsourcing for Large-Scale Software Development"–"Worker-centric Design for Software Crowdsourcing: Towards Cloud Careers") describes the basic concepts and notation in software crowdsourcing. Part II ("Bootstrapping the Next Generation of Social Machines"–"Towards a Game Theoretical Model for Software Crowdsourcing Processes") covers the theoretical frameworks and models on software crowdsourcing from different perspectives. Part III ("TRUSTIE: A Software Development Platform for Crowdsourcing"–"A Cloud-based Infrastructure for Crowdsourcing Data from Mobile Devices") presents the software crowdsourcing platforms and technologies.

Part I: Software Crowdsourcing Concepts and Design Issues

"Crowdsourcing for Large-Scale Software Development" defines the notation and principles of software crowdsourcing and introduces a 4-level maturity model for assessing software crowdsourcing ecosystems in terms of platform architecture, community scale, organization fabric and development approaches. "The Five Stages of Open Source Volunteering" reviews best practices of the open source projects, and identifies a five-stage process for volunteering and recruitment that can significantly increase the chances for a successful open source project.

Software crowdsourcing projects largely depend upon the global labor forces consisting of temporary crowd workers. But it will be problematic if we expect software crowdsourcing to become a sustainable industry where workers only need to maintain reduced levels of commitment with their commissioners. "Worker-

centric Design for Software Crowdsourcing: Towards Cloud Careers" analyzes the relevant issues including: trust and reputation development between workers, team selection and building for specific tasks, and contextualization of software crowdsourcing projects as a motivating factor for workers.

Part II: Software Crowdsourcing Models and Architectures

"Bootstrapping the Next Generation of Social Machines"–"Towards a Game Theoretical Model for Software Crowdsourcing Processes presents multiple approaches to modeling and analyzing software crowdsourcing systems and processes.

Both "Bootstrapping the Next Generation of Social Machines" and "Multi-Agent System Approach for Modeling and Supporting Software Crowdsourcing" attempt to create multi-agent models of software crowdsourcing systems but with different viewpoints. "Bootstrapping the Next Generation of Social Machines" develops the model on the basis of a conceptual notion called social machine, to describe human behavior and interaction protocols. These social machines emerging from human community and computing resources are governed by both computational and collective social process. "Multi-Agent System Approach for Modeling and Supporting Software Crowdsourcing" proposes an agent-based analytic framework to model the organization and process of software crowdsourcing. Based on the model, the authors have developed a service-based multi-agent system platform called AutoService to simulate software crowdsourcing process and validate the theoretical framework. Adaptive and programmable incentive mechanism is essential for software crowdsourcing systems to support processing of complex and inter-dependent development tasks. "Supporting Multilevel Incentive Mechanisms in Crowdsourcing Systems: an Artifact-centric View" presents a novel, artifact-centric approach for modeling and deploying incentives in software crowdsourcing environments. The proposed framework augments the Artifact's lifecycle model with incentive mechanisms to facilitate team formation, task orchestration, run-time management of data flow and dependencies, collaboration and coordination patterns.

The last three chapters focus on modeling and optimization of software crowdsourcing processes. "An Evolutionary and Automated Virtual Team Making Approach for Crowdsourcing Platforms" proposes a systematic and automated approach to optimize the assignment of crowd workers to a crowdsourcing task. By considering the major constraints in software crowdsourcing processes, the authors formulate this optimization problem with the framework of search-based software engineering. Given the dynamic nature of crowd workers participating in software crowdsourcing processes, it is essential to design effective quality-control management to ensure the quality of crowd submissions. "Collaborative Majority Vote: Improving Result Quality in Crowdsourcing Marketplaces" introduces a collaboration mechanism to extend majority vote, one of the most widely used quality-assurance methods, enabling workers to interact and communicate during task

executions. "Towards a Game Theoretical Model for Software Crowdsourcing Processes" introduces a conceptual framework of software crowdsourcing process and develops a game theoretical model of peer software production to describe competitive nature of software crowdsourcing. The analysis of the model indicates that prize-only awarding mechanism in software crowdsourcing can only motivate dominant developers with superior skills to other crowd developers.

Part III: Software Crowdsourcing Systems

"TRUSTIE: A Software Development Platform for Crowdsourcing"–"A Cloud-based Infrastructure for Crowdsourcing Data from Mobile Devices" describes software architectures and platforms to support Cloud-based software crowdsourcing and demonstrates examples of practices.

"TRUSTIE: A Software Development Platform for Crowdsourcing" and "Recommending Web Services Using Crowdsourced Testing Data" present new ideas about software crowdsourcing platforms from two different aspects. "TRUSTIE: A Software Development Platform for Crowdsourcing" introduces a software development environment named by Trustworthy Software Tools and Integration Environment (Trustie), which supports a community oriented and trustworthy software development framework. This framework integrates many features including crowd collaboration, resource sharing, run-time monitoring and trustworthiness analysis. "Social Clouds: Crowdsourcing Cloud Infrastructure" proposes a novel approach of an infrastructure crowdsourcing model, termed as Social Cloud, to facilitate a user-contributed cloud fabric to host software crowdsourcing activities on which software development services and systems can be hosted.

The other chapters of this part are mainly about the applications of software crowdsourcing. "Recommending Web Services Using Crowdsourced Testing Data" focuses on improving the QoS prediction of service platforms using crowdsourced testing data. "A Cloud-based Infrastructure for Crowdsourcing Data from Mobile Devices" focuses on crowd sensing applications that often involve massive number of smartphone users for collecting large scale of data. The author presents a multi-cloud based crowd-sensing platform named by APISENSE, supporting participatory sensing experiments in the wild.

Wei Li
Michael N. Huhns
Wei-Tek Tsai
Wenjun Wu

Part I
Software Crowdsourcing Concepts and Design Issues

Crowdsourcing for Large-Scale Software Development

Wei Li, Wei-Tek Tsai and Wenjun Wu

Abstract Large scale software systems with increasing complexity, variability and uncertainty, brings about grand challenges for traditional software engineering. Recently, crowdsourcing practices in the domain of software development such as Apple App Store and TopCoder have exhibited a promising and viable solution to the issues. The use of a crowd for developing software is predicted to take its place alongside established methodologies, such as agile, global software development, service-oriented computing, and the traditional waterfall. In this chapter, we propose a conceptual framework for the emerging crowdsourcing development methodology. We define the fundamental principles, software architecture, development process, and maturity model of the methodology for crowd workforce motivation, coordination and governance.

1 Introduction

Crowdsourcing has captured the attention of the world recently [1]. Numerous tasks or designs conventionally carried out by professionals are now being crowdsourced to the general public who may not know each other to perform in a collaborative manner. Specifically, crowdsourcing has been utilized for identifying chemical structure, designing mining infrastructure, estimating mining resources, medical drug development, logo design, and even software design and development. Recently, crowdsourcing has been adopted in the mobile phone market, where both Apple's App Store [2] and Google's Android market open up their programming APIs and allow designers

W. Li · W. Wu (✉)
School of Computer Science and Engineering, Beihang University, Beijing, China
e-mail: wwj@nlsde.buaa.edu.cn

W. Li
e-mail: liwei@nlsde.buaa.edu.cn

W.-T. Tsai
School of Computing Informatics and Decision Systems Engineering,
Arizona State University, Tempe, AZ, USA
e-mail: wtsai@asu.edu

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_1

to upload their creative applications and make these software products available for consumers to download and purchase. Such an innovative approach resulted in an exponential growth in the number of applications hosted in both smartphone stores. The proliferation of this practice indicates a paradigm shift from traditional software factory to crowdsourcing based peer-production mode, where large numbers of regular end-users are empowered as co-creators or co-designers, and their creative energy is coordinated to participant in large projects without a traditional organization. As an emerging paradigm of software development, how can we generalize the practice of software crowdsourcing and define an effective software engineering methodology for the construction of real complex and large-scale software systems? Software development is commonly considered as one of the most challenging and creative activities. As one software problem is solved by a new solution, another new software problem is subsequently created by the solution. Thus, the software engineering history has a long list of techniques, processes, and tools in the last 50 years, yet the field is still seeking for new solutions and new technologies each year as it encountered new problems. Many traditional engineering techniques such as modeling, simulation, prototyping, testing and inspection were invented to address the importance of the engineering aspects of software development. Furthermore many new techniques such as model checking, automated code generation, design techniques, have been proposed for large-scale software development. The complex nature of software engineering determines that software crowdsourcing has many unique features and issues different from general crowdsourcing. Specifically, software crowdsourcing needs to support.

- The rigorous engineering discipline of software development, such as rigid syntax and semantics of programming languages, modeling languages, and documentation or process standards.
- The creativity aspects of software requirement analysis, design, testing, and evolution. The issue is to stimulate and preserve creativity in these software development tasks through collective intelligence?
- The psychology issues of crowdsourcing such as competition, incentive, recognition, sharing, collaboration, and learning. The psychology must be competitive while at the same time friendly, socialable, learning, and personal fulfillment for participants, requesters and administrators.
- The financial aspects of all parties including requesters, crowdsourcing platforms, and participants.
- Quality aspects including objective qualities such as functional correctness, performance, security, reliability, maintainability, safety, and subjective qualities such as usability.

In this chapter, we intend to thoroughly discuss and explore fundamental principles of software crowdsourcing to facilitate designing, development and maintenance of a variety of large-scale software systems. The rest of the chapter is organized as follows: Sect. 2 defines the notion of "software crowdsourcing" and compares it with other mainstream software development methods including agile software development, outsourcing and open source. Section 3 describes

the principles of software crowdsourcing including co-innovation, competitive development and offense-defense based quality assurance. Section 4 presents a Cloud-based software crowdsourcing architecture for crowd workforce coordination and governance. Section 5 introduces a four-level maturity model for the assessment software crowdsourcing process from the aspects of system's scale, project time spans, developers and software platforms.

2 Overview of Crowdsourcing Based Software Development

Modern large and complex software systems [3] demonstrate the following prominent characteristics: highly decentralization of systems with heterogeneous and varying elements, inherently conflicting, unknowable, and diverse requirements as well as continuous evolution and deployment. All these features totally contradict the most common assumptions made by the mainstream software engineering methodologies. As software development is highly iterative, software development processes often emphasize on delivery of intermediate documents such as requirement documents, design documents, test plans, test case documents. Especially, in plan-driven software development methods such as Waterfall model, Spiral model, and model-driven process, it is assumed that a group of software architects can make accurate analysis of stakeholders' requirements and define a consistent and systematic blueprint for software architecture and detailed development plan to implement every functional requirement. Moreover, all the software development activities can be well coordinated and achieve steady progress towards the predefined milestones that lead to fully tested releases of the working software systems. None of these methodologies can support agile and adaptive software development of large-scale software systems in response to constant changes in user requirements and dynamic networked environments.

To address these problems, software development processes have evolved from the traditional plan-driven process to recent agile methods [4], distributed software development [5], open-source approach [6], and community-driven software development. These modern software methods propose three major elements to transform software engineering methodologies.

(1) Adaptive Software Construction and Evolution: Software developers must be able to constantly improve and update software systems to meet the dynamic of user requirements.
(2) Community and Social Intelligence: To conquer the complexity of large-scale software systems, software projects have to tap into collective intelligence and manpower of developer communities to have an on-demand workforce with diversified skills and talents.

(3) Open and Distributed Process for Software Innovation: An open and distributed software process is vital for facilitating community-based software development and innovation. Community members need to be well coordinated to solve challenging problems in software projects.

Crowdsourcing is an online, distributed problem solving and production model that leverages the collective intelligence of online communities for specific management goals. Apparently, it is promising to apply the principle of crowdsourcing in the domain of software development and incorporates all the above ingredients to enable a new software paradigm—software crowdsourcing.

2.1 What's Software Crowdsourcing?

Software crowdsourcing is an open call for participation in any task of software development, including documentation, design, coding and testing. These tasks are normally conducted by either members of a software enterprise or people contracted by the enterprise. But in software crowdsourcing, all the tasks can be assigned to anyone in the general public. In this way, software crowdsourcing not only capture high-level requirements and quality attributes in terms of community interests, but also facilitate end-users to create their own customized software services. Therefore, software crowdsourcing is essentially a community-based software development paradigm.

Software crowdsourcing adopts an adaptive, open and distributed development process. Rich engagement with both user and developer communities enables software crowdsourcing practitioners to quickly respond to fresh feedbacks and demands from their users and make frequent and in-time improvements on their software. Internet-based software platform with built-in online community and market functionalities facilitates coordinate millions of distributed community members in the construction and evolution of large-scale software systems. Well-known software crowdsourcing platforms including Apple's App Store, TopCoder [7] demonstrate the success of software crowdsourcing processes in terms of software ecosystem expansion and product quality improvement.

(1) Online Software Marketplace for Software Crowdsourcing
Apple's App Store is essentially an online IOS application marketplace where IOS vendors can directly deliver their products and creative designs to smartphone end-users. This mobile software market introduced the micro-payment and ranking mechanism to enable designers to gain profits from application downloads. The high ranking and download rate of a vendor's iPhone application brings in considerable profit for him. This market-oriented incentive motivates vendors to contribute innovative designs to the marketplace and to attract more downloads from users of the App Store. Also, by posting comments and reviews on the IOS applications available in the marketplace, customers can explicitly present their requirements to application designers. There are competitive relationships among vendors working on the same

category of IOS applications. They have to quickly respond to the feedbacks and increasing demands from their customers and improve their software to secure their advantage to other competitors and retain their customers. All these mechanisms of the App Store promote the exponential growth in the IOS ecosystem. Within less than 4 years, Apple's App Store has become a huge mobile application ecosystem where up to 150,000 active publishers have created over 700,000 IOS applications.

Despite the official IOS developer program provided by Apple, many IOS application designers still need convenient and transparent channels to seek funding and programmers for developing novel application ideas, collecting more user reviews and testing applications before releasing them on the marketplace. Therefore, many community-based, collaborative platforms have been emerging as smart-phone applications incubators for the App Store. For instance, AppStori (www.appstori.com) introduces a crowd funding approach to build an online community for developing promising ideas about new iPhone applications. It provides a "preview" window for iPhone application enthusiasts to choose their favorite ideas, support and actively engage in the promising projects. Designers are encouraged to propose project concepts to the review committee who evaluates the novelty of the project proposals and grant them into the formal AppStori projects. After their proposals gets approval, each AppStori project team estimates development effort including its deadline and budget, and reaches out to crowd for funding support through the AppStori platform.

(2) Contests and Tournaments for Software Crowdsourcing

Another crowdsourcing example—TopCoder, creates a software contest model in which all the software development tasks are organized as contests and the developer of the best solution wins the top prize. Since 2001, TopCoder has established a commercial online platform to support its ecosystem and gather a virtual global workforce with more than 250,000 registered members and 50,000 active participants. Given such a global workforce, some large research institutes are taking advantage of the TopCoder platform for their software crowdsourcing projects. For instance, NASA, teamed up with Harvard University, has established NASA Tournament Lab (NTL), to encourage competitions among talented young programmers for the most creative algorithms needed by NASA researchers [8].

TopCoder's idea of using contests to reach a board audience of developers with various backgrounds and expertise has a long tradition in history. For decades, design contests featuring competition between participants competing for the best idea. TopCoder's crowdsourcing platform allows developers to competitively disclose their creative software solutions and artifacts to the corporations to win the contest prizes. Towards the same programming challenges, TopCoder members can interact, discuss and share their insights and experiences with like-minded peers on the Web forum, and even build social networks to establish a sense of community. These patterns of simultaneously cooperating and competing behavior resemble the concept of co-opetition between firms, defined as a portmanteau of cooperation and competition ties among units in the same social structure.

2.2 Comparison Between Software Crowdsourcing and Other Software Development Methods

As a new software development method, software crowdsourcing inherits many important ideas from other modern software development methods including agile software development, outsourcing and open source. Essentially, software crowdsourcing is a distributed and open software development paradigm with the basis of online labor market and developer community. As shown in Table 1, the major similarities and differences between software crowdsourcing and other methods are summarized in the dimensions of distribution, coordination, openness and development incentive.

(1) Agile Software Development Versus Software Crowdsourcing
Agile software development was proposed to address the weakness of plan-driven approaches with the introduction of iterative and incremental development. With adaptive planning, evolutionary development, and an iterative coding method, agile approach can significantly accelerate the delivery of new software features in response to the fast change in user requirement. Its organization emphasizes co-located face-to-face interaction, in which software engineers and a customer representative can have constant communication and discussion through the whole lifecycle of software development. Apparently, pure agile processes are best suited for small-sized or medium-sized projects, and attempts to scale them up for large and complex projects, usually result in re-adoption of more traditional practices [9]. Currently, most research efforts on how to take a hybrid approach of plan-driven and agile method focus on intra-organization scenario, where agile style development is embraced at the team level and the traditional upfront planning as well as maturity framework is adopted in the enterprise level. Even with the hybrid approach, all the practices of agile methods are often limited within the boundary of companies and organization. In contrast, crowdsourcing methods supports large-scale, distributed online collaboration for market-driven massive software production, overcoming the

Table 1 Comparison between software crowdsourcing and other methods

	Agile software development	Global software development and outsourcing	Open source software development	Software crowdsourcing
Distribution	Co-located	Globally distributed	Globally distributed	Globally distributed
Coordination	Face-to-face meeting	Tightly coupled collaboration with online tools	Loosely coupled and self-control collaboration via social network	Loosely coupled coopetition via social network
Openness	Not necessary	Not necessary	Open source code to community	Open process to community
Incentive	Permanent employment	Contract-based employment	Volunteers	Online labor market

limitation of both plan-driven process and agile approach. The community-oriented governance structure of crowdsourcing enables us to efficiently harness the intellectual resources for the development of large-scale software systems. More importantly, driven by the reward and diverse user requirements, community members can be self-organized into many agile teams that can quickly accomplish development tasks and deliver high quality products.

(2) Global Software Process Versus Software Crowdsourcing

With the wider adoption of outsourcing in global software companies, their software engineers distributed across the world have to collaborate on software development tasks via online environments. Such a distributed software process brings about many research issues on software engineering such as knowledge sharing, collaborative working and process management. Despite the distributed nature of software crowdsourcing, it is completely different from distributed software development, especially outsourcing. Software projects based on outsourcing depends upon pre-arranged contracts between the companies and their consulting partners. Distributed software process involves a stable and consistent collaboration among the developers located in different places. But software crowdsourcing targets the labor force of developer communities via open solicitation, thus the collaboration in software crowdsourcing process is more loosely coupled, temporary and task-driven than distributed software development, especially outsourcing. By engaging with open communities, it can leverage creativity from more diversified talented developers beyond the boundaries of enterprises. Moreover, software crowdsourcing distinguishes from distributed software development by introducing the competitive mechanism into the organization of software development. Collaboration is no longer the only way to orchestrating development activities. Instead, individual developers can compete for prizes based on their performance in the same development tasks.

(3) Open Source Versus Software Crowdsourcing

There is high similarity between software crowdsourcing and open source. Both methods emphasize the openness of software development but from different aspects. Open source method values the openness of source code among software development communities and encourages contribution from community members through intrinsic and altruism incentives such as reputation, opportunity to learn program skills, and willingness to address user needs. In contrast, software crowdsourcing features the openness of software process, distributing all kinds of development tasks to communities. Software crowdsourcing adopts explicit incentives, especially monetary award such as contest prize, to stimulate participation of community members. Therefore, software crowdsourcing can be regarded as an extension of open source, which generalizes the practices of peer production via bartering, collaboration and competition. But it doesn't necessarily distribute end-products and source-material to the general public without any cost. Instead, it emphasizes on the community-driven software development on the basis of open software platforms, online labor market and financial rewarding mechanism.

3 Principles of Software Crowdsourcing for Software Ecosystem

Practices of software crowdsourcing may take different strategies to achieve different project goals such as broadening participation, seeking novel solutions, identifying talents and acquiring high-quality software products. But they have to obey the same principles that are vital to the growth and prosperity of a vibrant software ecosystem. This section presents three common principles of software crowdsourcing including Co-Innovation, Competitive Software Development and Offense-Defense based Quality Assurance.

3.1 Co-Innovation and Peer-Production

Crowdsourcing software development emphasizes the fundamental principle of Co-Creation, Co-Design and Co-Development, which encourages wide engagement of community members in the construction of large-scale system ecosystems. This principle enables every community member including end-users, professional programmers and domain experts with active roles in participation of software development, which goes beyond the boundary of the conventional software development practices.

(1) Peer-Production and Egalitarian Design Process
The meaning of value and the process of value creation in modern software innovation are rapidly shifting from a software product and developer-centric view to personalized consumer experiences. Informed, networked and active consumers are increasingly co-creating value with the professional developers. Peer-production of software crowdsourcing follows an egalitarian design process, in which large numbers of regular end-users are empowered as co-creators or co-designers to engage with all sorts of innovative activities relevant to software production, such as easy idea validation, collaboration on requirements, comments on usability, and even programming and testing. With intuitive and easy-to-use programming tools, regular end users can not only rank software applications by comparing their performance and quality, but also generate lightweight applications and customize specific features to meet their diverse purposes and demands.

Peer-production fosters and accelerates the emergence of an open platform software ecosystem by broadening crowd participation and connecting platform vendors, software developers and customer communities. The success of a large-scale software crowdsourcing project essentially depends upon the scale and activity of such a developer community. Previous studies on open source ecosystems [10, 11] reveal that most open source projects in the major open source platforms such as Sourceforge and Github only have limited number of active members. Thus, it is critical for a software crowdsourcing project to take all the viable methods including financial support, intellectual resource of development and learning to recruit as many participants as possible.

In a software ecosystem, peer-production can drive Co-Innovation through its online software marketplace and labor platform. The online software marketplace enables large scale distribution for promoting technological adoption and integration of software innovation. Novel software products and services are delivered instantly to regular consumers so that they could do test-and-drive of the new products and generate instant feedbacks for developers to perform full-fledged system analysis and real-time refinement.

(2) Creativity and Diversity in Software Crowdsourcing Communities
Crowdsourcing needs a diverse community of crowd workers in distributed locations with different background to ensure a diversity of opinions are expressed to encourage creativity, and to avoid any biased of individual leaderships within a specific community. Especially when the major goal of a software crowdsourcing project is to explore novel and brilliant ideas for future research and development, designers should be encouraged to propose a wide range of concepts and models for a public solicitation.

This reflects one of creativity principles low threshold, high ceiling and wide walls [12]. Software crowdsourcing often sets up a low threshold to encourage more people to take a variety of software development tasks to broaden the community. It also adopts high ceiling strategy to screen qualified programmers for complex and challenging projects. According to the principle of wide wall, platforms and tools of software crowdsourcing should present enough freedom for participants to explore new dimensions and seek innovative solutions.

3.2 Competitive Software Development

Co-opetition relationship among its participants plays a vital role in efficient organization of peer-production in software crowdsourcing. Software crowdsourcing contests encourage competition between participants competing for the best software solution, thus facilitating the delivery of high quality software product with reduced development costs. These contests with relevant coordination mechanisms, contest rules, and prize structures actually generate monetary incentives for participation and influence dynamic behavior of these participants. On the other hand, software crowdsourcing platforms also provide collaboration and community functionalities such as user profile, online forum, web chat, group voting and wiki. These collaboration services enable community members to vote on favorite ideas or software solutions, discuss various topics, share of insights with other like-minded peers, receive assistance from others and generally derive benefits through collaborative innovating activities.

(1) Competition in Software Crowdsourcing

The process of software crowdsourcing often exhibits strong competitive characteristics, which resembles the procedure of paper selection in a high-profile conference. The specification of a software crowdsourcing task including the technical requirement, time constraint, and rewards for the work, is publically announced to the community as an open call for the best solution via competition. As a massive response to this call, community members, who are interested in this task, can start to take on the task and submit their results to a review board of the community, who is in charge of evaluating the quality of their pieces and make the final decision for the final winners. TopCoder adopts such a competitive development process that breaks down the basic software development steps in the Waterfall paradigm, such as conceptualization, requirement analysis, architecture design, component production, module assembly, software testing, into a series of online contests.

Figure 1 illustrates the TopCoder process model where all the software components in the software system need to be developed separately through online contests and then assembled up for further testing and integration. These TopCoder contests involve the following three phases:

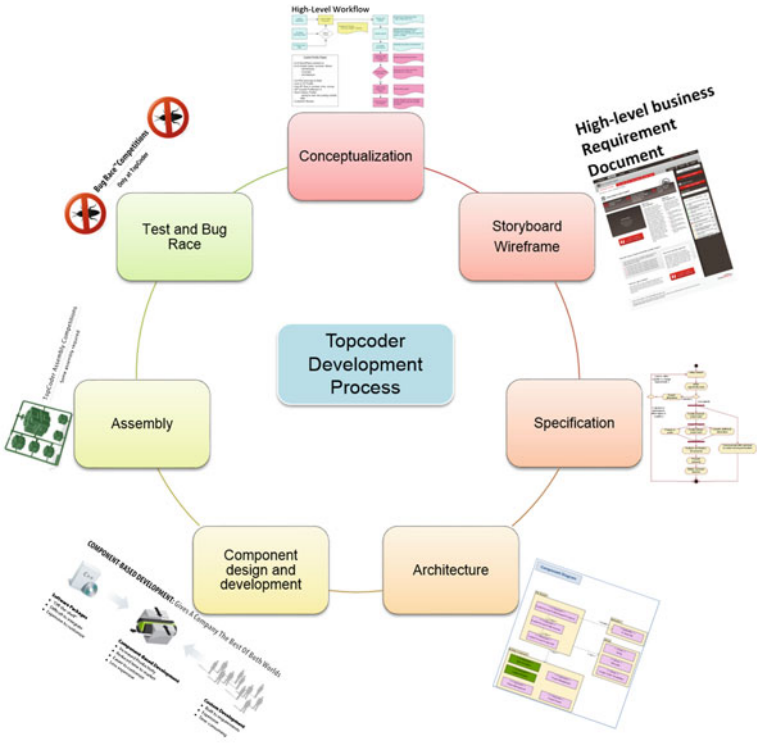


Fig. 1 TopCoder competitive development process

- Task Solicitation:
  A TopCoder project manager defines the requirement in natural language and publishes the specification of a task on the platform to call for the contribution of any potential developers in the community. Sometimes he can specify the qualification criteria of the task to allow only the developers with adequate skills and expertise to register.
- Solution Bidding:
  Developers with the right knowledge base and skill sets for the task register the TopCoder contest and start to work on the task based on their understanding of the requirement. If they finish the task within the time constraint, they can submit their solutions to the review team of the task as bidding for the winner prize.
- Evaluation and Rewarding:
  The review team evaluates all the submissions and determines two winners out of the best submissions. Only the submissions with the best quality can possibly win the prize eventually. And the selected solution and review suggestions can be used as intermediate materials in the next step of the project. Apparently, such a competitive process aims at seeking the best solution with the highest quality from multiple outputs for the same development tasks. In addition to the purpose of quality screening, it also manages to reduce the cost of labor work by only paying the winners among all the participants. However, this cost reduction has negative impact on non-winning teams since their efforts and contributions are not going to get any monetary payoff. Therefore, weak teams may lose their motivations to undertake the task if they have few chance to outperform their competitors. To address this issue, non-monetary rewards must be present in the contest as supplementary incentives for the participants. For instance, a contest run by a famous IT company can have many contestants who are willing to participate to improve their reputation ranking even if the company may offer relatively low winner prize. The non-financial incentive has to function in the context of collaborative community where members value reputation, learning opportunity, career development and altruism.

(2) Collaboration in Software Crowdsourcing

As collaboration and coordination are the essential elements in any distributed software process, all the software crowdsourcing platforms support more or less collaboration among their participates. For instance, many online IOS communities such as AppStori (www.appstori.com) and iBetaTest (www.iBetaTest.com) provide collaborative software crowdsourcing venues to bridge iPhone application designers, IOS developers, beta-testers, funding donors and supportive customers together. Therefore, small teams or individuals on iPhone application development can seek external funding, obtain help to run beta-testing on early versions of their apps, and collect preliminary customer feedbacks before releasing their products onto the

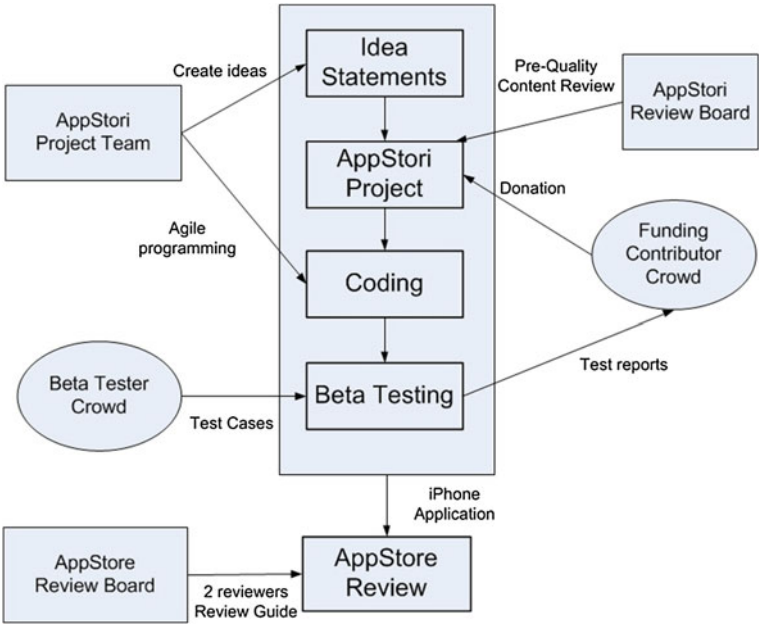


Fig. 2 AppStori crowdsourcing development processes

very competitive App Store. Figure 2 illustrates the major players in the platform of AppStori:

- Crowdfunding and Stakeholders:
  Similar to Kickstarter.com in the financial market, any AppStori member can post a novel idea of IOS application onto the AppStori, specifying the project goals and deadlines, and ask for funding. After the AppStori Review Board approves the project, any people who love the idea of the project can become a stakeholder by funding the project. Although a stakeholder cannot withdraw its fund before the end of the project, the person can check the progress of the project and request the project leader to finish the project within a specified time frame. Otherwise, the funding will not be transferred to the developer team at the end.
- Transparency and Agile Development:
  AppStori encourages 100 % transparency between the developer team and the community. Every team member must provide detailed personal profile to present their background and role in the project. And the project status must be updated in time to display the progress to the community. Anyone who is willing to be part of the team can contribute ideas to guide the projects development process, provide social support for the project via social media promotion, and even become beta testers for quality assurance. The beta testers can evaluate early versions of the application under test and give direct feedbacks and bug reports to the developers.

- Stimulating Creativity in Mobile Application Development:
  AppStori encourages exploration diverse paths and embracing rich styles to create innovative mobile applications. Web 2.0 communications make it easier for donors, developers, and testers to exchange ideas, post comments and figure out brilliant ideas. AppStoris open and creative environment enables numerous cycles of trial-and-error towards the successful design until it can be ready for publication on App Store.
- Knowledge sharing and Learning:
  AppStori is a great place for people to learn new applications, technologies, and trend in mobile applications, and mobile consumers and enthusiasts can connect directly with mobile developers and entrepreneurs for direct feedbacks. These feedbacks are expensive to obtain in the past.

3.3 Offense-Defense Based Quality Assurance

Obtaining quality software is a common but challenging goal for software crowdsourcing projects. Software quality has many dimensions including reliability, performance, security, safety, maintainability, controllability, and usability. As the breadth and diversity among crowdsourcing participants, workers and researchers is immense, it is critical to find an effective way to rigorously evaluate crowd submissions and assure the quality of the final outcome.

Based on the observation of success practices of crowdsourcing software tests such as TopCoder and uTest (www.utest.com), we find that the Offense-Defense based quality assurance is the most fundamental way to eliminate the potential deficits in the documents, models and codes submitted by crowd. Multiple groups from the community undertake different responsibilities of software development tasks and intensively cooperate with each other to find problems in their counterparts work and reduce the bugs in their own work. Such cross-validation activities can be characterized as offense (finding bugs in other peoples work) and defense (reducing bug in peoples own work). Table 2 gives the definition of offense and defense.

Table 2 Offense and defense definition

Offense activities	Defense activities
Evaluate the inputs including any input documents, prototypes, interviews and relevant materials that will be used in performing the tasks	Evaluate the outputs including any deliverables such as documents and software
Goal: maximize the number of faults in the input documents, and provide feedback to those who prepared the inputs	Goal: minimize the number of bugs that will be found by other teams or people (crowd)

- Offense: To carry out the tasks, each team needs to understand its requirements, and examine the validity of the inputs to determine if they are feasible, correct, consistent and complete. This can be done by inspecting, reviewing, simulating, model checking, verifying the contents of the inputs. This kind of process often reveals mistakes, inconsistency, incompleteness, complex user interactions, invalid assumptions, and other issues, which can be useful feedbacks to those who prepared the input documents. Because any mistakes in the input documents may cause significant problems in the current tasks. Thus, the goal is to maximize the fault detection rate of the input documents.
- Defense: Once requirements are understood, the team needs to prepare its output. However, the team realizes that their outputs will be cross examined by other teams carefully, and the team may lose its creditability if its outputs are of low quality. Thus, the team needs to spend significant time to check and verify its deliverables to minimize both the probability of bugs and the damage of potential bugs.

The Offense-Defense quality assurance can be exemplified by Harvard-TopCoder collaborative project [13], to facilitate Harvard biomedical scientists to work with the TopCoder community on developing computational algorithms and software tools for exploring highly uncertain innovation problems in the biomedical domain, which are mostly related to large-scale biological data analysis. The process of this software development project essentially involves three parties with their responsibilities (Table 3):

Table 3 Offense and defense analysis of Harvard-TopCoder teams

	Main tasks	Offense	Defense
Catalysts	Develop problem statements with Researchers for the project and specify final acceptance criteria, secure funding for crowdsourcing	Need to ensure the problem is feasible, thus go over with the Research team to maximize the probability of identifying problems in requirements	Need to ensure that the Researcher understand the problem and well decompose the problem to minimize the ambiguity in problem statements
Researchers (domain expert)	Decompose the problem, develop the high-level design for the crowd to develop components or algorithms, develop test cases for acceptance testing, and evaluate the algorithm/code	Working with the catalysts to develop quality specification, make sure that the problem is feasible to maximize the probability of identifying problems in requirements	The specification/test cases developed must be of high quality, and thus they need to review and inspect the specification carefully, and answer any inquiries from the crowd

(continued)

Table 3 (continued)

	Main tasks	Offense	Defense
Crowd (programmers)	Develop algorithms or components based on specification supplied	The crowd will review the specification carefully to ensure complete understanding, and identify any problems in the specification, and develop test cases to test other algorithms submitted by other contestants	The crowd will evaluate its algorithm or components carefully using test cases developed to minimize the possibility of bugs in the code

- **Researchers**: They are domain experts working with Catalysts to finalize the submission to the crowd including problem statement, they also prepare test data, and finally score algorithms submitted by the crowd.
- **Catalysts**: They prepare the needs and funding for Researchers and the crowd.
- **Crowd**: They are TopCoder members with programming skill, looking at the problems issued by Researchers, participating in the competition by coming up with new algorithms. And if they win, they will get rewards from Catalysts.

4 Software Crowdsourcing Architecture and Models

Because of diversity and distance among the participants of software crowdsourcing projects, it is essential to have a Cloud-based software platform to support large-scale open and distributed development processes in software crowdsourcing. Imagine a large crowd of developers located in different time zones try to work together on the same software project. A cloud provides them with a scalable platform with sufficient resources, including computing power and software databases. With the emerging cloud software tools such as DevOps (a portmanteau of development and operations) [14] and large-scale software mining, it significantly reduces the amount of manual labor needed in setting up software production environments and empowers peer developers to perform software crowdsourcing tasks efficiently in design, coding, and testing.

4.1 Software Crowdsourcing Architecture

Despite different needs and strategies adopted by software crowdsourcing processes, they actually share much commonality in term of platform support. As an online labor market where crowd workforce can autonomously choose a variety of software development tasks requested by project initiators, a software crowdsourcing platform needs to effectively facilitate synergy between two clouds C human cloud and

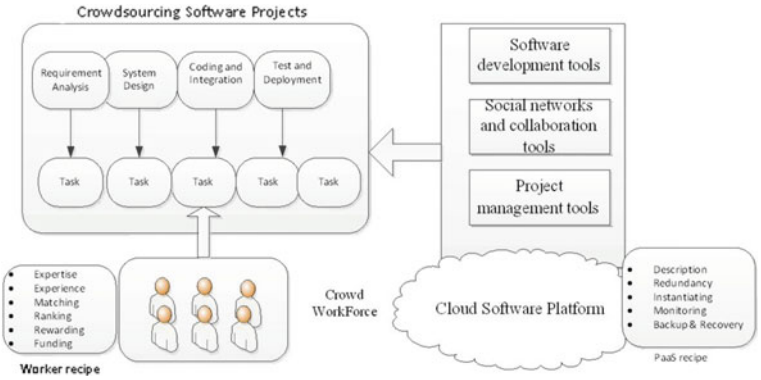


Fig. 3 Reference architecture for cloud-based software crowdsourcing

machine cloud. Many core services pertaining to the labor management and project governance must be incorporated into the platform including expertise ranking, team formatting, task matching, and rewarding as well as crowd funding. Moreover, each individual should be able to easily initialize a virtual workspace with design and coding software tools customized for specific tasks in a crowdsourcing project. All these common elements are encapsulated in reference architecture of cloud-based software crowdsourcing system, shown in Fig. 3.

There are three major groups of software tools in a cloud-based software crowdsourcing:

- **Software Development Tools**:
  In any software crowdsourcing projects, crowd workforce needs modeling tools, simulation tools, programming language tools such as compilers and intelligent editors, design notations, and testing tools. An IDE for crowdsourcing can integrate these tools for requirements, design, coding, compilers, debuggers, performance analysis, testing, and maintenance. For example, cloud software configuration tools such as Chef (www.opscode.com/chef/) and Puppet (puppetlabs.com) allow community members to establish their own virtualized development environment. Map-Reduce based log management tools support large-scale system log administration and analysis, and enable community members to resolve software problems and enhance system reliability using log messages.
- **Social Network and Collaboration Tools**:
  Facebook, twitters, wikis, blogs and other Web-based collaboration tools allow participants to communicate for sharing and collaboration. For example, Facebook profiles can be enhanced to facilitate the formation of a virtual crowdsourcing team even if the participants do not previously know each other. A collaborative blackboard-based platform can be used where participants can see a common area and suggest ideas to improve the solutions posted there. Novel financial tools for cloud payment and virtual credit management can support project funding supply and monetary exchange between project initiators and crowd workforce.

- **Project Management Tools**:
  Crowdsourcing project management should support project cost estimation, development planning, decision making, bug tracking, and software repository maintenance, all specialized for the context of the dynamic developer community. In addition to these regular functions, it needs to incorporate crowdsourcing specific services such as ranking, reputation, and award systems for both products and participants. For example, TopCoder introduces a sophisticated ranking scheme, similar to sport tournaments, to rank the skills of community members in software development. Community members often make up their mind to participate in a specific contest if they know the ranking of the participants already enrolled.

Through DevOps technologies, Cloud platforms allow users to describe their own software stacks in form of recipes and cookbooks. With the properly specified PaaS recipes, software crowdsourcing project managers can establish a customized cloud software environment to facilitate software crowdsourcing process:

(1) A manager utilizes software networking and collaboration tools to design rewarding mechanism to motivate crowd workers and project management tools to coordinate development tasks among crowd workforce by ranking expertise of each individual and matching their skills with the different levels of tasks.

(2) Towards a specific software project, a manager sets up a virtual system platform with all the necessary software development gears to assist crowd workers with their crowdsourcing tasks.

(3) Both cloud platform and workflow can work in an elastic way to make cost-efficient resource utilization. Depending on the value of software product and scale of software development, a project manager can specify the appropriate budget to attract as many talent developers as possible and provision computing resources to sustain the development activities.

4.2 Software Crowdsourcing Models

Software crowdsourcing can be characterized in terms of the crowd size, software scale and complexity, development processes, and competition or collaboration rules. Formal models for designing and modeling software crowdsourcing can have the following foundations:

- **Game theory**: The nature of contests in competition-based crowdsourcing can be analyzed via game theory. For example, one can determine the reputation reward value based on the number of participants and reward price, because often participants are willing to compete to gain reputation rather than receiving the reward price using Nash Equilibrium [15].
- **Economic models**: Economic competition models provide strategies and incentives for a crowd to participate, and reward structuring rules for organizers to maximize their return from crowdsourcing. The recent development of contest theory

[16] introduced new mathematical tools, such as all-pay auction, to describe the synergy among individual efforts, competition prize structure, and product quality.
- **Optimization theory**: Due to the competitive and dynamic nature of software crowdsourcing processes, it is challenging to coordinate unstable virtual teams, optimize the partition and allocation of development tasks, and balance costs, time, and quality. Thus, a search-based software engineering approach [17] can be applied in software crowdsourcing to address the optimization problems.

5 Maturity Model for Software Crowdsourcing

We introduce a maturity model as the basis for comparing and assessing software crowdsourcing process. We define four levels of software crowdsourcing organization. These levels of crowdsourcing projects are characterized by the size of development teams, the scale of software systems under development, the project duration and platforms support for software development processes. The definition of the four-level maturity model is summarized in Table 4.

- **Level 1**: A project of this level aims at developing a small-size software system with well-defined modules. It often takes a limited amount of time span (less than few months) for a single person to finish such a project. Currently, most software crowdsourcing projects such as AppStori, TopCoder, and uTest, can be categorized into this level. In these projects, both coders and software products are ranked by their development performance and product quality. To facilitate the collaboration between project representatives and community participants, crowdsourcing platforms provide communication tools such as wiki, blogs, and comments as well as software development tools such as an IDE, testing, compilers, simulation, modeling, and program analysis.
- **Level 2**: A project of this level aims at developing medium-size software system with well-defined specifications. It often takes a medium time span (several months to less than 1 year) for teams of people (<10) to finish such a project. And it adopts adaptive development processes with intelligent feedback in a common cloud platform where people can freely share thoughts. At this level, a crowdsourcing platform supports an adaptive development process that allows concurrent development processes with feedback from fellow participants; intelligent analysis of coders, software products, and comments; multi-phase software testing and evaluation; Big Data analytics, automated wrapping of software services into SaaS (Software-as-a-Service), annotations with terms from an ontology, cross references to DBpedia, and Wikipedia; automated analysis and classification of software services; ontology annotation and reasoning such as linking those services with compatible input/output.
- **Level 3**: A project of this level aims at building large systems with clearly specified requirements. It often needs teams of people (<100 and >10) to work on such a project across a long time span (<2 years). The increase in scale and complexity of

Table 4 Offense and defense analysis of Harvard-TopCoder teams

	Developers	Software	Time span	Platform support
Level 1	Individuals or small number of small teams	Small applications or software modules	Less than 2 months	Developer ranking; social networks; software repository of components; communication among participants
Level 2	Teams of people	Well-defined systems	Several months to less than 1 year	Level 1 tools + automated code analysis; wrapping of software services into SaaS; service publication and discovery
Level 3	Teams of people	Well-defined large systems	Long time span (less than 2 years)	Level 2 + automated cross verification, automated requirement; design, service matching; automated regression analysis
Level 4	Multinational teams of developers	Large and adaptive systems	Long time span or perpetual evolving software	Level 3 tools + domain-oriented ontology, reasoning and annotation; automated cross verification and test generation processes; automated configuration of platform

the systems brings challenges on the aspect of quality assurance. To ensure the high quality of software products submitted by many community participants, it relies upon automated cross verification and cross comparison among contributions. A crowdsourcing platform at this level contains automated matching of requirements to existing components including matching of specification, services, and tests; and automated regression testing.

- **Level 4**: A project of this level involves multinational collaboration of large and adaptive software ecosystems. The development process of such a complex and large-scale project often takes years to accomplish. More intelligent software development tools are necessary to conquer the complexity and uncertainty through the long evolutionary lifecycle of the system. A crowdsourcing platform at this

level may contain domain-oriented crowdsourcing with ontology, reasoning, and annotation; automated cross verification and test generation processes; automated configuration of crowdsourcing platform; and may restructure the platform as SaaS with tenant customization.

6 Conclusion

Cloud-based software crowdsourcing is a new approach for low-cost rapid software development, and the existence of large ecosystems has shown that this approach is viable. This chapter gives the definition of software crowdsourcing and thoroughly discusses both the fundamental principle and conceptual models of this emerging methodology.

As an open call for participation in any task of software development, software crowdsourcing can be organized in different ways such as online software market,on-demand labor resource and collaborative communities. Essentially, software crowdsourcing is a distributed and open software development paradigm that inherits many important ideas from other modern software development methods including agile software development, outsourcing and open source. But the distinguishing features of software crowdsourcing including loosely coupled coordination, open process and market-driven incentive mechanism, make it possible to tackle with the challenges of large-scale software development.

In this chapter, we summarize three common principles of software crowdsourcing including Co-Innovation, Competitive Software Development and Offense-Defense based Quality Assurance. To follow the principles and support large-scale software development with crowd, a software crowdsourcing platform needs to adopt a Cloud-based software architecture and incorporate core services for effectively facilitating synergy between two clouds C human cloud and machine cloud. Furthermore, we introduce a four-level maturity model as the basis for comparing and assessing software crowdsourcing process. These levels of crowdsourcing projects are characterized by the size of development teams, the scale of software systems under development, the project duration and platforms support for software development processes.

Software crowdsourcing is a ripe and promising area for research including modeling, analysis, simulation, experimentation, and support environment development. There are many future research directions including theoretical models, optimization methods, infrastructure support features, and social issues. Theoretical tools such as game theory and economic models should be used to describe synergy among individual efforts, incentive mechanisms, product quality and project constraints. Formalized software crowdsourcing process models and optimization methods need to be developed to fulfill the optimal goals of software crowdsourcing projects. Current software crowdsourcing platforms have to incorporate more customizable features to enable crowdsourcing project managers to design and adopt specific ways of crowdsourcing organization in terms of incentive scheme, quality assurance mechanism and

community governance. Lastly, more investigations need to be conduct on welfare of crowdsourcing workers such as their health care, career development path, and working ethnics.

References

1. Doan, A., Ramakrishnan, R., Halevy, A.Y.: Crowdsourcing systems on the world-wide web. Commun. ACM **54**(4), 86–96 (2011)
2. App Store: http://www.apple.com/iphone/from-the-app-store/ (2014). Accessed on 18 Aug 2014
3. Northrop, L., Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., et al.: {Ultra-Large-Scale Systems}-the software challenge of the future (2006)
4. Martin, R.C.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall PTR, Upper Saddle River (2003)
5. Herbsleb, J.D., Moitra, D.: Global software development. IEEE Softw. **18**(2), 16–20 (2001)
6. Raymond, E.: The cathedral and the bazaar. Knowl. Technol. Policy **12**(3), 23–49 (1999)
7. Lakhani, K., Garvin, D.A., Lonstein, E.: Topcoder (a): developing software through crowdsourcing. Harvard Business School General Management Unit Case (610–032) (2010)
8. Andy, L., Raugh, A., Erickson, K., Grayzeck, E.J., Knopf, W., Lydon, M., Lakhani, K., Crusan, J., Morgan, T.H.: The NASA tournament laboratory ("NTL"): improving data access at PDS while spreading joy and engaging students through 16 micro-contests. In: AAS/Division for Planetary Sciences Meeting Abstracts, vol. 44 (2012)
9. Barlow, J.B., Giboney, J.S., Keith, M.J., Wilson, D.W., Schuetzler, R.M., Lowry, P.B., Vance, A.: Overview and guidance on agile development in large organizations. Commun. Assoc. Inf. Syst. **29**(1), 2 (2011)
10. Xu, J., Christley, S., Madey, G.: Application of social network analysis to the study of open source software (2006)
11. Madey, G., Freeh, V., Tynan, R.: Modeling the free/open source software community: a quantitative investigation, In: Free/Open Source Software, Development, pp. 203–221 (2005)
12. Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B., Pausch, R., Selker, T., Eisenberg, M.: Design principles for tools to support creative thinking (2005)
13. Lakhani, K.R., Boudreau, K.J., Loh, P.R., Backstrom, L., Baldwin, C., Lonstein, E., Lydon, M., MacCormack, A., Arnaout, R.A., Guinan, E.C.: Prize-based contests can provide solutions to computational biology problems. Nat. Biotechnol. **31**(2), 108–111 (2013)
14. Spinellis, D.: Don't install software by hand. IEEE Softw. **29**(4), 86–87 (2012)
15. Wu, W., Tsai, W.T., Li, W.: An evaluation framework for software crowdsourcing. Front. Comput. Sci. **7**(5), 694–709 (2013)
16. Corchón, L.C.: The theory of contests: a survey. Rev. Econ. Des. **11**(2), 69–100 (2007)
17. Harman, M.: The current state and future of search based software engineering. In: 2007 Future of Software Engineering, pp. 342–357. IEEE Computer Society (2007)

The Five Stages of Open Source Volunteering

Dirk Riehle

Abstract Today's software systems build on open source software. Thus, we need to understand how to successfully create, nurture, and mature the software development communities of these open source projects. In this article, we review and discuss best practices of the open source volunteering and recruitment process that successful project leaders are using to lead their projects to success. We combine the perspective of the volunteer, looking at a project, with the perspective of a project leader, looking to find additional volunteers for the project. We identify a five-stage process consisting of a connecting, understanding, engaging, performing, and leading stage. The underlying best practices, when applied, significantly increase the chance of an open source project being successful.

1 Introduction

Open source software has become an important part of the Internet and today's enterprises. There is little software left that does not at least include some open source components. Thus, understanding how open source projects work and how to utilize them is a critical capability of software product companies who wish to crowd-source some of their development work.

Open source software projects can be split into community open source and commercial open source software projects [1, 2]. Community open source software is software that is owned by a community, typically by way of distributed copyright ownership or by ownership through a non-profit foundation. Commercial open source software development is curated by a single company, which maintains the ownership of all relevant intellectual property. According to Mickos, commercial open source rarely receives and incorporates code contributions from their user communities [3]; however, community open source does.

D. Riehle (✉)
Computer Science Department, Friedrich-Alexander-Universität Erlangen-Nüremberg, Erlangen, Germany
e-mail: dirk@riehle.org; dirk.riehle@fau.de

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_2

Community open source software projects rely on volunteer work to a (significant) extent. Projects which are more mature and relied upon by companies may gain commercial support, which reduces reliance on volunteers [4]. Commercial support can take the form of direct financial contributions, which allows foundations to acquire employees. Also, companies may assign their own employees to contribute to the project. Leaders of new and small community open source projects cannot expect commercial support, and must learn how to recruit and retain volunteers if they want their project to grow.

In this article we review best practices of open source community management, specifically, how to find, keep, and grow volunteers. The article is based on a literature review and observation of existing open source projects. We identify five stages of the open source volunteering process which we call the connecting, understanding, engaging, performing, and leading stages. For each stage, we discuss the best practices of the actors of that stage. In addition, we discuss the underlying guiding principles that we found to be common to all the practices.

Thus, this article makes the following contributions:

- It defines guiding principles underlying the volunteering process;
- It presents a five-stage model of the open source volunteering process;
- It collects and catalogs best practices applicable to each stage.

The article is structured as follows. Section 2 presents the guiding principles, Sect. 3 introduces the five-stage process and discusses its properties. Section 4 walks through the stages in detail, discussing best practices and supporting tools. Related work as relevant to the different sections is discussed in place. Section 5 concludes the article.

2 Guiding Principles of Open Source Projects

In reviewing the literature (as referenced in place) and working with open source communities as well as from prior work we identified the following three guiding principles project leaders need to understand for an effective recruiting process:

- Recruiting is Investment [5, 6]
- Open Communication [5, 6]
- Open Collaboration [5, 7]

We discuss these principles in turn.

2.1 Recruiting is Investment

According to Fogel, every interaction with a user is a chance to recruit a new volunteer [6]. At the same time, according to Fitzpatrick and Collins-Sussman, the scarcest

resource that a project has is attention and focus [8]. In combination, this leads to the primary guiding principle of open source volunteer recruiting:

- **Recruiting volunteers is a project investment**

Recruiting volunteers is obviously necessary for a project to grow. However, the time spent on recruiting takes attention and focus away from actual software development. Spending time on recruiting should therefore be viewed as an investment to be made wisely.

Investments may or may not work out. The time spent on a potential volunteer may or may not be wasted. Thus, if time is spent on recruiting, it should be spent well, and it should be spent in a form commensurate with the likelihood of success, in this case, of finding a new volunteer. The best practices of Sect. 4 all embed this principle.

2.2 Open Communication

Open source projects follow a particular style of communication which helps support a distributed volunteer community. Fogel, for example, argues that communication styles portray project members (who may never have met in person) to each other [6]. He argues for general principles (all communication should be public) and very specific principles (no conversations in the bug tracker). Using this and other sources, we derived the following four fundamental maxims of open communication that characterize open source projects and the way project members communicate with each other and potential volunteers:

- **Public**. All communication should be public and not take place behind closed doors; any private side-communication is discouraged.
- **Written**. All communication should be in written form; if this is not possible, any relevant communication should be transcribed or summarized in writing.
- **Complete**. Communication should be comprehensive and to the extent possible, complete. Assumptions are made explicit and key conclusions are summarized.
- **Archived**. All communication should get archived for search and later public review. Thus, previous conversations are available for posterity.

Taken together, these maxims create transparency and discipline communication, leading to more effective distributed collaboration. Although not all communication within a project will embody all four maxims, a project which is motivated to be transparent and grow its community will make the according effort, for example, by transcribing or summarizing non-email forms of communication in order to provide a public archive.

Public communication ensures that all members of the community have the opportunity to participate, which creates buy-in and trust. Written communication enables asynchronous, distributed work. People who are less fluent in the language used for communication also benefit from having additional time to absorb the meaning [9],

which makes the project accessible to a wider audience. Complete communication reduces opportunities for misunderstanding and ensures that the community shares a common understanding of objectives. Archiving increases transparency by ensuring that decisions can be understood in context.

Many of these concepts reinforce one another. Writing enables archiving, as current search technology is text-based. The need to derive meaning from archives encourages more complete communication. Archives are more comprehensive and comprehensible if all conversation is public.

2.3 Open Collaboration

No two open source projects follow the same software development process. However, in prior work and by way of project reviews, we identified three underlying fundamental components of open source collaboration [7]. These three maxims of open collaboration are:

- **Egalitarian**. Everyone may join a project, no principled or artificial status-based barriers to participation exist.
- **Meritocratic**. Decisions are made based on the merits of the arguments, and status is determined by the merits of a person's contributions.
- **Self-organizing**. Processes adapt to people rather than people to processes.

Related work frequently subsumes the first two maxims under the single concept of meritocracy. We find it helpful to distinguish between them: Openness in the context of egalitarianism means that all people have the opportunity to participate, whereas openness in the context of meritocracy ensures that all work is evaluated on the basis of its intrinsic value.

These concepts are in stark contrast to traditional work inside companies. Projects are not egalitarian: Employees are assigned to work on them and cannot choose to work on other projects. Decisions are not necessarily made on the basis of the merit of the arguments, but are ultimately the choice of the person with the greatest power. Finally, processes in a large company are typically defined by a central department and employees are expected to adapt their work processes to the company environment.

Open source projects are different: It is recognized that any potential volunteer could become a valuable resource. Thus, an effective project process must be open to accepting volunteers (egalitarianism), must recognize quality regardless of the source (meritocracy), and allow processes to develop according to the needs of the community (self-organizing). The five stage process of open source volunteering described in the following Section is based on the three guiding principles of the open source volunteering process: recruitment is investment, open communication, and open collaboration.

3 The Five-Stage Volunteering Process

In [10], Behlendorf illustrates a typical example of how a developer might join a project, rise through the ranks, and become a project leader. The developer

1. needs to solve a problem,
2. searches the web for appropriate software,
3. finds a matching project,
4. checks out the project,
5. gives the project a try and is happy,
6. finds a bug and reports it,
7. makes a first contribution,
8. engages in a conversation,
9. keeps contributing,
10. receives a vote of trust, and
11. ultimately leads the project.

By correlating this 11-step process with Fogel's work [6] and by aligning it with the Onion model of roles in open source software development [11], we were led to a simpler and denser five stage model of the open source volunteering process than the one proposed by Behlendorf. This model is shown in Fig. 1.

An innovation of this model is the addition of the two complementary views of volunteer and project (leader), which lead to complementary but mutually supporting best practices and activities. The stages are defined in the following way:

- **Stage 1: Connecting**. In this stage, a potential volunteer stumbles over a project by lucky chance or, after searching for something like it, finds the project through a search engine. The project needs to prepare for this to happen, which requires marketing itself through appropriate channels and at appropriate portals.
- **Stage 2: Understanding**. In this stage, once a potential volunteer is looking at a project's website, the website needs to draw him or her in. Using a variety of best practices, the project helps the visitor quickly understand what the project is about and whether it should be of interest to them.

#	Stage	Volunteer View	Volunteer Role Name	Project View
1	Connecting	Find project	Seeker	Market project
2	Understanding	Understand project	Visitor (Reader)	Explain project
3	Engaging	Engage with project	User	Engage with user
4	Performing	Work within project	Contributor	Work with contributor
5	Leading	Lead project	Leader	Enable career

Fig. 1 A five-stage model of the open source volunteering process

- **Stage 3: Engaging**. In this stage, a potential volunteer is inspired to engage with the project, for instance by installing the software or joining a mailing list. The project strives to welcome users to the community and direct them toward the next stage by providing information of simple ways to volunteer.
- **Stage 4: Performing**. In this stage, a volunteer contributes to the project. The project community needs to be receptive to initial efforts by reacting quickly to contributions and creating conversations to improve quality. The project needs to guide users towards becoming regular contributors.
- **Stage 5: Leading**. In this stage, the volunteer accepts responsibility for the direction of the project or community. The project must have a mechanism for identifying potential leaders and making decisions on their promotion to leader status as well as for communicating this clearly.

Not all volunteers pass through all stages, but stages can only be taken one after another. Each subsequent stage will be reached by fewer volunteers. The best practices described in the next section support each stage. For a project, they help increase volunteer commitment. When recruitment is viewed as an investment, best practices are aligned with promoting long-term involvement. For a volunteer, best practices advise on how to achieve goals, from finding a project that fulfills a need to gaining recognition within a project. We now describe each stage in detail, along with selected best practices from each perspective and tools to support them.

4 Best Practices and Supporting Tools

A best practice "is a broadly-accepted, typically informally-defined, method for achieving a particular goal that is considered superior to most other known methods" (author's adaptation of the Wikipedia entry on "best practice" [12]). Thus, a best practice is a method reflecting the state-of-the-art as applicable in a particular context.

The following best practices have been derived from the respective references, in particular [6, 8, 10, 13–15]. They have been grouped according to the phases described in Sect. 3 and divided into the two perspectives described there: the volunteer's view and the project leader's view. They are based on the application of the three principles of open source volunteering described earlier in Sect. 2. Due to the large number of sometimes mundane best practices, not all are discussed in detail.

The principle which informs all best practices from the project view is that of recruitment as an investment. In a project, time is the scarcest resource, and recruitment takes time. Increasing the long-term return on time invested [6]—or encouraging volunteers to move to each successive phase—is therefore the objective of the project's leadership. Open communication and open collaboration are also reflected in the best practices; they are the underlying tenet that make open source projects work.

A volunteer is not a passive subject to be recruited, but an individual with objectives in mind. At each phase, a volunteer wants to ensure maximum value for the investment, which is where best practices come into play. The volunteer who is prepared will achieve better results than one who fails to consider the project's needs.

Volunteer (Seeker) View	Project View
Stumbling	**Active Outreach**
• Stumble upon project • Follow word-of-mouth	• Choose a good name • Define relevant channels • Use channels consistently • Announce visibly • Be matter of fact
Searching	**Passive Inflow**

• Use general search engine
• Search on open source portal
	• Register on all portals
	• Support search engines
	• Support lucky chance
	  ◦ Use portal features
	  ◦ Provide findable summaries
	  ◦ Work towards portal metrics

Fig. 2 Best practices of Stage 1, the Connecting stage

4.1 Stage 1: Connecting

Figure 2 displays all best practices of Stage 1, the Connecting stage. Volunteers can be separated into two categories, those that stumble onto the project by luck, and those that search for a solution to a problem they have. Project best practices can be split into active outreach being performed and passive inflow that needs to be prepared for. Two terms stick out among the project best practices, channel and portal:

- An open source project channel is a communication channel for a project to reach potentially interested parties, in particular volunteers. Examples of such channels are:
  - Social media channels like Facebook or Twitter
  - Targeted communication channels like Slashdot or Hacker News
  - Specific open source conferences like OSCON or ApacheCon
- An open source project portal is a portal website dedicated to open source projects. Examples of such websites are:
  - Project hosting sites like SourceForge or Github
  - Meta-sites like Freshmeat or Open Hub

Channels are mostly used for active outreach and when the project has a story to tell, for example, the initial release. Portals are used for passive inflow where searchers can find them when they are seeking a solution.

A project should choose a good name that is easy to remember and ideally indicative of the project's purpose. As an alternative to descriptive names, wholly artificial

names may serve the project equally well. Any communication then should stick to that name and use it consistently. The relevant channels and portals (see above) need to utilized repeatedly, consistently and predictably. Any communication should be matter-of-fact rather than hyperbole-projects are trying to create a long-term reputation, not a short spike of attention followed by disappointment over the hyperbole. The most common form of communication is the announcement of new releases of the software, followed by announcements over major developments in the project community or sponsorship.

4.2 Stage 2: Understanding

In the second stage, the emphasis is on communicating the project's purpose to a volunteer who wants to quickly learn if the project answers his or her need. Figure 3 lists relevant best practices.

Various pieces of information need to be easily accessible, both in terms of finding and understanding the information. A first step is to have a clear mission statement that spells out the project's purpose and does so in a highly visible place, for example, the front page of the project's website on a software forge. Examples and screen-shots should be easily accessible to make it straightforward for visitors to assess what the software does in practical and tangible terms (short of downloading and installing the software, which would be the next step). Words are only so good-examples and screen-shots sometimes communicate more clearly.

Many visitors will also want to know about related project information like software licenses or (assumed) quality of the software (by way of development status). Thus, a project should display prominently which open source license it is using, what state of development it is currently in, and what future expected developments are, including upcoming releases and key new features and functionalities. The

Volunteer (Visitor) View	Project View
Reading Up	**Explain Project**
• Read up on project	• Have clear mission statement
	• Provide examples and screen-shots
	• State license and terms
	• Provide simple downloads
	• Show current and future releases
	• Show development status
	• Provide user documentation
	• State whether volunteers are welcome
	• State rules of engagement

Fig. 3 Best practices of Stage 2, the Understanding stage

visitor, who wants to try the software, may need user documentation, which should therefore be provided.

Visitors have questions or may want to become volunteers, and hence a project is well advised to spell whether volunteers are welcome and what the project rules are so that someone considering to participate will know what they are getting into.

4.3 Stage 3: Engaging

In the engagement stage, the emphasis is on facilitating communication between the project and the volunteer. Figure 4 lists relevant best practices.

It needs to be clear (and clearly displayed) how current project members can be reached. At this stage, the project may only be perceived as an anonymous entity with no particular face. Potential volunteers need starting points, for example, forums or mailing lists where they can ask questions.

A first response should be welcoming of a new potential volunteer, and any possible rudeness, whether incidental or deliberate, needs to be stopped immediately. It is paramount that any project member redirects any privately posed questions to a public forum and avoids answering questions in private; this would be a highly inefficient use of their time. Visitors need to understand that they consume time and hence should do their homework or should be guided to do their homework before asking. Doing one's homework implies reading existing materials to avoid redundant questions. Also, visitors have to learn to ask in public so that everyone can

Volunteer (User) View	Project View
All Volunteers	**Towards all Volunteers**
● Do your homework before asking	● Show how to reach project
◦ Search archives	● Welcome to community
◦ Read documentation	● Stop any rudeness
● Communicate prudently	● Avoid private discussions
◦ Be matter of fact	● Accept initial redundancy
◦ Don't jump to conclusions	● Provide simple tasks
	● Provide incremental tasks
	● Call out lurkers from the shadows
Software Developers	**Towards Software Developers**
● Respect project practices	● Provide tool access
● Respect project culture	● Remove arbitrary tool obstacles
● Accept guidance	● Show requirements list
	● Provide developer guidelines
	● Provide developer documentation

Fig. 4 Best practices of Stage 3, the Engaging stage

learn from their considerations and questions. Communication, both on the visitor and the project side, should be matter of fact and content focused, trying to help solve the problem or question at hand.

For more advanced visitors, or users of the software, it should be possible to learn about simple tasks that the project would benefit from. The project should spell out such tasks, even if writing them down may cost nearly as much time as performing them, because simple tasks provide a mechanism to engage volunteers. Similarly, there should be incremental tasks to be picked up, which will allow volunteers to work with existing developers rather than alone. Incremental tasks also introduce volunteers to existing technical aspects of the project.

A lot of things can go wrong when setting up a project for engaging potential volunteers, and appropriate attention needs to be paid so that tools and project artifacts like task and requirements lists are accessible, and that developers can find appropriate guidelines and documentation.

Underlying all these project best practices is the guiding principle of making it as easy as possible for a volunteer to make a first contribution. Getting to that first contribution is the single most important hurdle a project has to overcome. Thus, many of the best practices work hand-in-hand to make that first contribution happen.

4.4 Stage 4: Performing

The performance stage is when the volunteer contributes to the project. It can be subdivided into a first contribution and later more regular contributions. Figure 5 lists appropriate best practices for project leaders.

A volunteer's first contribution is like dipping a toe into the water. Depending on how the experience feels, the volunteer may not come back. Thus, it is important to ensure that this first contribution becomes a positive experience. For one, a contribution should be well received and reacted to. Nothing is worse than no reaction, for example, by letting a patch sit idle. The appropriate reaction to a patch is to turn it into a conversation, not only to say thanks, but also to encourage further contributions by pointing the volunteer to related issues. In all but the most simplest patches or contributions, the volunteer may have to be guided to reworking the contribution, for example, to ensure compliance with the project's programming guidelines. Code review of a patch submission is a general best practice, but also shows the volunteer that their contribution is being taken serious, even if it leads to a request to fix a problem with the submission. Finally, after a successful contribution, it is critical to pay credit to who credit is due and list the volunteer as a contributor to the project.

Volunteers who have become regular contributors may then be willing to pick up other tasks outside their original interests. Still, project leaders should track and play to volunteer interests when asking them for help, for example, to work on a particular feature. Volunteers, who have bought into the project are frequently willing to pick up work that they originally did not join the project for. This includes unloved tasks like project documentation and is not restricted to technical tasks alone.

Volunteer (Contributor) View	Project View
General Practices	**General Practices**
• Contribute	• Be component-oriented • Work from features • No discussions in bug tracker • Decide using consensus • Vote as a last resort
First Contributions	**Towards First Contributions**
• Contribute	• React speedily, don't sit on patches • Turn contributions into conversations • Practice conspicuous code review • Track contributions, provide credit • Praise plentifully, criticize specifically • Prevent territoriality
Regular Contributions	**Towards Regular Contributions**
• Contribute	• Track interests, assign accordingly • Distinguish inquiry from assignment • Share technical and managerial tasks • Follow-up on delegated assignments • Document practices and traditions • Archive practice descriptions

Fig. 5 Best practices of Stage 4, the Performing stage

A project leader who asked a contributor to perform some work and received a commitment needs to fulfill a managerial role now. For example, if the contributor is not providing the promised feature, the project leader may have to inquire about progress, nudging the contributor along (and making mental notes as to whether this was a good request that matched the volunteers interests). Sometimes, a project may run into difficult people. "Difficult" or even "poisonous" people, according to Fitzpatrick and Collins-Sussman, may waste a projects time or split and even ruin a project [8]. Best practices to prepare for the problem are to

1. build a healthy community and
2. document all decisions.

It is necessary then to detect the problem: Difficult people typically don't show respect, miss social cues, are overly emotional, and make sweeping claims not based on any data. Best practices to handle the problem are to

1. not engage them,
2. ignore them if possible,
3. remove them from the project if necessary.

General engineering management advice applies as well. The system software architecture needs to match its social structure, which typically implies a well-componentized structure so that developers can work independently of each other and in a distributed fashion. The requirements and open tasks in contrast should be feature-oriented, as completing a feature is a major motivation for a developer because it provides meaning to the work being performed.

Unlike in traditional (non-open) contexts, however, the principles of open communication and open collaboration need to be maintained. This requires appropriate consensus-oriented and merit-based discussion as to decisions to be taken. Voting to make a decision is a last resort to resolve a conflict and should be used rarely.

4.5 Stage 5: Leading

In the final phase, the volunteer becomes a project leader and takes responsibility for the best practices of the project view for the earlier phases. Figure 6 shows some of the best practices at this level.

At this stage, a project leader is basically a manager, but without the power found inside traditional organizations. He or she has to rely on the power of persuasion and goodwill that contributors have developed towards the project. With increasing commercialization and paid-for participation in open source, some of these challenges become less serious, and developers may need less intrinsic motivation. Still it remains good practice to personally motivate developers through their work beyond the possible salary that an employer may be paying for their open source work.

The power of leadership rests on setting a good example by taking responsibility and acting accordingly, by praising other people's work and acknowledging their contributions. Additional actions may be necessary to help contributors outside the project, for example, if they are performing open source work on company-time

Volunteer View	Project View
General Practices	
• Take responsibility	
• Praise other contributors	
• Acknowledge contributions	
• Support others towards their manager	
• Provide tokens of appreciation	
• Define community career	
• Define roles and positions	
• Decide promotion privately	
• Announce promotion clearly	

Fig. 6 Best practices of Stage 5, the Leading stage

without the employer having a particular interest in the project. Then, the open source project leader may have to help motivate why the developer's work ultimately benefits his or her employer.

The open source project itself needs management in that contributors find the work formally acknowledged in the form of traditional credits. Contributors are also having a form of open source career. Taking steps in this career, most notably from contributor to committer, may be touchy subjects, and are one of the few discussions that the existing project leaders may have to decide privately and not in the public eye. This is justified, because such a discussion is typically more about the social aspects of working with the to-be-promoted person rather than his or her technical capabilities. In case of a positive decision, the promotion needs to be announced publicly and documented accordingly so that everyone in the project knows.

5 Conclusion

This article first identified the three guiding principles of open source projects. Volunteers are the lifeblood of an open source project, and an effective recruiting process considers all three principles.

First, recruiting is an investment: time and effort are invested in order to yield long-term results. Second, open communication facilitates the volunteer process by creating transparency. Third, open collaboration opens the recruitment process to any potential volunteer and allows them to contribute to their fullest extent.

A model of five phases of engagement is presented. This model looks at the different levels of volunteer commitment from both the perspective of the volunteer and of the project.

The three volunteering principles are used to advance a number of best practices which are tied to the five stages of engagement. At each phase—connecting, understanding, engaging, performing and leading—there are objectives and best practices for both the volunteer and the project, as represented by its leadership. The best practices are derived from existing literature and observation.

Acknowledgments I would like to thank Ann Barcomb and the anonymous reviewers for helpful comments that improved the paper.

References

1. Riehle, D.: The economic motivation of open source software: stakeholder perspectives. Computer **40**(4), 25–32 (2007)
2. Riehle, D.: The economic case for open source foundations. Computer **43**(1), 86–90 (2010)
3. Mickos, M.: Open for business: building successful commerce around open source, (2010)
4. Riehle, D. Riemer, P. Kolassa, C. Schmidt, M.: Paid vs. volunteer work in open source. In: 47th Hawaii international conference on system sciences (HICSS), pp.3286–3295, January 2014

5. Riehle, D.: The open source knowledge sharing and volunteering process, (2011)
6. Fogel, K.: Producing Open Source Software. O'Reilly, Farnham (2005)
7. Riehle, D., Ellenberger, J., Menahem, T., Mikhailovski, B., Natchetoi, Y., Naveh, B., Odenwald, T.: Open collaboration within corporations using software forges. IEEE Softw. **26**(2), 52–58 (2009)
8. Fitzpatrick, B., Collins-Sussman, B.: How open source projects survive poisonous people, (2008)
9. Carmel, E., Tija, P.: Offshoring Information Technology. Sourcing and Outsourcing to a Global Workforce. Cambridge University Press, Cambridge (2006)
10. Behlendorf, B.: How to contribute to open source projects, (2011)
11. Crowston, K., Howison, J.: The social structure of free and open source software development. First Monday **10**(2) (2005). First Monday, Special Issue # 2: Open Source—3 October 2005 The social structure of free and open source software development (originally published in Volume 10, Number 2, February 2005)
12. Wikipedia. Definition of Best Practice
13. Bacon, J.: The Art of the Community. O'Reilly, Farnham (2012)
14. Delacretaz, B.. Open source collaboration tools are good for you, (2009)
15. Gabriel, R., Goldman, R.: Innovation Happens Elsewhere. Elsevier (2005)

Worker-Centric Design for Software Crowdsourcing: Towards Cloud Careers

Dave Murray-Rust, Ognjen Scekic and Donghui Lin

Abstract Crowdsourcing is emerging as a compelling technique for the cost-effective creation of software, with tools such as ODesk and TopCoder supporting large scale distributed development. From the point of view of the commissioners of software, there are many advantages to crowdsourcing work—as well as cost, it can be a more scalable process, as there is the possibility of selecting from a large pool of expertise. From the point of view of workers, there is a different set of benefits, including choice of when and how to work, providing a means to build a portfolio, and a lower level of commitment to any particular employer. The crowdsourcing of software development—in common with some other activities such as design—represents an alternative to existing mechanisms that require skilled workers. However, if crowdsourcing were to replace traditional employment for a significant proportion of software developers, the reduced levels of commitment between workers and commissioners could prove problematic for workers over time. In this paper, explore three areas of interest: (i) trust and reputation development; (ii) team selection and team building; (iii) contextualisation of the work carried out. By drawing together work in these areas from the point of view of *workers* rather than *commissioners*, we highlight some of the incipient issues with the growth of crowdsourced labour. We also explore ways in which crowdsourcing of software development—and other skilled practices—differers from microtasking.

D. Murray-Rust (✉)
CISA, School of Informatics, University of Edinburgh, Edinburgh, UK
e-mail: d.murray-rust@ed.ac.uk
URL: http://www.cisa.inf.ed.ac.uk

O. Scekic
Distributed Systems Group, Vienna University of Technology, Vienna, Austria
e-mail: oscekic@dsg.tuwien.ac.at
URL: http://dsg.tuwien.ac.at

D. Lin
Department of Social Informatics, Kyoto University, Kyoto, Japan
e-mail: lindh@i.kyoto-u.ac.jp
URL: http://www.i.kyoto-u.ac.jp

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_3

1 Introduction

Crowdsourcing is emerging as a powerful tool for carrying out many different tasks, and commissioning work of different kinds. Organisations accrue many benefits from crowdsourcing work, typically including: cost, quality, network effects, lower commitment, greater pool of expertise, scalability and on-demand labour [8, 10, 22].

Most analyses of crowdsourcing take the point of view of the commissioners of work rather than the workers themselves: how is it possible to get work done better, more cheaply, more robustly or faster. When considered as an "outsider" technology, this need to prove value to commissioners of work is completely understandable. However, crowdsourcing is no longer a niche activity, however. In 2009, it was estimated that cloudworkers had been paid up to $2Bn over the preceding decade [9]; the number of participants has grown by over 100 % per year, and there are now over 6 million cloudworkers worldwide.

Mechanical Turk is seeing a shift from casual, spare time work carried out by Americans, to Indian workers who derive essential income from the work that they do [21]. This has led to some analysis of the ethics of "professional crowdsourcing", where the monetary rewards have a significant effect on the people carrying out the work. Silberman et al. [25] discuss several problems faced by Mechanical Turk workers (Turkers), such as employers who don't pay, or reject work; conning naïve users into downloading malware or participating in scams; and poorly defined or structured tasks. Bederson and Quinn [2] discuss wage-based issues, and call for hourly rates for crowdworkers, or at least an expected hourly rate to be published, along with clear quality metrics which stop employers being able to arbitrarily reject work after it has been done. On the positive side, Horton [12] finds some evidence that online employers are seen as more trustworthy than local employers. Of particular interest is Felstiner's discussion of the crowdsourcing industry in the context of labour laws [8].

Software crowdsourcing is markedly different to "Turking" and other similar microtasking activities, for a number of reasons. Frei [9] divides crowd labour into *micro tasks, macro tasks, small projects* and *complex projects*. While much of the crowdsourcing industry focusses on microtasks, software creation tends to fall into the small- or complex-project brackets, requiring workers to bring in existing skills, and some degree of coordination or direct worker contact.

The software crowdsourcing industry is arguably older than micro-tasking: RentACoder, Guru, LiveOps and Elance all began before Mechanical Turk was introduced, and TopCoder and oDesk appeared prior to the explosion of crowd labour platforms (2006 onwards [9, p. 4]). As such, software crowdsourcing can be seen as a natural evolution of a freelancer-based industry: it is very common for developers to work on a short term basis, being brought in for particular projects without expecting a continuing relationship with the client. The crowdsourcing aspect is largely a technological addition to simplify existing practices.

As well as commercial crowdsourcing, there is a grand tradition of free crowdsourcing exemplified by the Free and Libre Open Source Software (FLOSS)

movement, which has created many high profile, high quality complex pieces of software (e.g. the Linux kernel). A side effect of this is the profusion of tools for carrying out distributed programming tasks, such as chat applications, distributed version control systems (DVCSs), bug and issue trackers, unit test frameworks, continuous deployment systems etc. This means that the software community as a whole has greater literacy with the techniques and practices that allow and support distributed working than many other areas.

Just as the entry requirements for creating software are higher than those for microtasking, the monetary incentives are greater too: as of 2010, on average, a Turker earns $1.25/h, which is less than the minimum wage in India. In contrast, an average developer on oDesk earns $15/h—double the US minimum wage, and higher than designers ($10/h) or technical writers ($8/h) [8].

On the topic of payment, a concern for software creators is the cannibalisation of their market; the average yearly wage for a software developer in the US is $92 k [27], approximately four times the oDesk average, and which also includes benefits and a sense of job security; this economic advantage is one of the chief motivations for firms to crowdsource work (although there are others, such as innovation and competitiveness [10]). An early article about crowdsourcing [13] examines the disruptive effect that crowdsourced stock photo sites (e.g. iStockphoto.com) had on the professional stock photography market. The article also hints that a large part of professional photographer's ability to charge for their work is dependent on access to professional quality equipment (although this is a position professional photographers may disagree with). Software development is different here: low-end computers can be used to create high quality code, and programmer selection is more likely to be carried out on the basis of a laundry list of technologies mastered than computational hardware owned.

There is the question of whether this commoditisation of the low end really disrupts the lives of working professionals. In the design world, companies such as 99designs produce give access to design at lower prices than were previously available; however, it is an open question whether this represents lost sales for high end design houses, or simply the opening up of the market to a new audience. Software development already has a highly diverse ecosystem, with pricey, local "boutique" consultancies pitted against low cost, low quality outsourcing houses. Software crowdsourcing has the potential to impact on both of these communities, as (a) the price is low enough to be competitive with the cheaper providers and (b) the quality can be high enough to make some mid- to high-level providers worried.

Software development is typically seen as a long term career, with potential for high earnings and a transition into management. Most jobs would include benefits, job security and would provide necessary equipment. This can be contrasted with crowdworking which is fraught with asymmetric power relationships and poor conditions:

> Depending on a firm's quality standards, crowdsourcing can be astoundingly cheap. Crowd workers receive low wages, no benefits, no job security, and have not much prospect at present of organizing to change these conditions. Employers do not need to provide facilities and support for a workforce, nor do they need to pay overhead fees to an outside contractor. [8]

In this paper, we set out our opinions around the question of what it would take to make software crowdsourcing a *sustainable* industry. This means being able to attract intelligent, motivated individuals, who can make enough money to satisfy themselves. Essentially, we ask the question "What would we want from a crowdsourcing marketplace", or, more eloquently:

> Can we foresee a future crowd workplace in which we would want our children to participate? [16]

2 Themes of Interest

As noted previously, there are many issues faced by most crowdworkers, whether performing relatively unskilled micro tasks or larger complex projects. However, there are some issues that are particular to the situation where existing professional activities are replaced with crowdsourcing. While microtasks create a new labour market, there is already a large population of people who expect to be able to construct a career around software development. Here we attempt to tease out some of these expectations and issues, and highlight where current and emerging technologies and systems can help to support these workers.

These themes are by no means comprehensive; rather they represent a solid backbone around which to start building career ladders for cloud software developers, and combatting the atomisation of workers and information asymmetry which are endemic in the cloud.

2.1 Trust and Reputation as Prerequisite for "Cloud Careers"

Arguably, co-workers are one of the most important factors contributing to a pleasant and productive working environment . In traditional companies workers usually cannot directly select their co-workers. However, since the nature of the employment relationship is a long-lasting one, it gives them time to get to know their colleagues and forge working relationships. The management will actively monitor these relationships in order to achieve a more harmonic, and thus more productive or creative environment.

In crowdsourcing environments, the relationship of workers with the platform and co-workers are irregular and short-lived. This leaves no time to get to know and other workers. Crowdsourced teams are often unique, both time- and composition-wise. Co-workers are often hidden behind digital profiles, creating an atmosphere of distrust and discomfort. Furthermore, such settings provide and ideal environment for attempting fraudulent activities, such as multitasking, rent-seeking or tragedy of the commons style exploitation [19].

Hence, managers and workers must be supported in the task of assembling teams; while there are many different approaches to this, almost all of them rely directly or indirectly on some sort of *trust* or *reputation* metrics.

Trust and reputation are two terms often used incorrectly and interchangeably, as varying definitions for both terms exist, and are used in different contexts by different authors. In this paper, we use a loose and operative description which we feel is in general agreement with the majority of the crowdsourcing community.

Trust is a concept denoting one's *personal* expectation of someone else. Reputation is an aggregated, *communal* expectation of an individual.[1] Trust influences reputation, and vice versa:

- $T(a, b)$—denotes the level of trust which a has for b;
- $R_c(x)$—denotes an aggregate measure of worker x's trustworthiness within community c.

In case of crowdsourcing and other socio-technical systems, this means that a worker (Alice) can trust a co-worker (Bob), if her personal feeling or past experience supports the trust. However, Alice's high opinion of Bob may not be shared, and he could have a low reputation within the community. If the low reputation is simply a result of having few collaborations, then over time as Bob works with people, the community view will change and his reputation will increase.

This small example demonstrates two well-known problems: (a) bootstrapping of trust/reputation; and (b) the dilemma of choosing trust over reputation.

The trust-reputation dilemma is reflected in the fact that although reputation reflects an aggregated community view, depending on the particular collaboration pattern in a team, it may be better to favour trust over reputation as a metric. However, this depends on the confidence level of the trust metric, and a trade-off is usually required.

Trust bootstrapping is related to the fact that the trust emerges only after several interactions between the same two subjects have taken place. Reputation bootstrapping is related to the fact that a subject's good reputation can be established only after the majority of community members (or its most influential members) have interacted with the subject.

In crowdsourcing environments, it is often not realistic to expect enough interactions to build up valid trust and reputation metrics. Teams are formed and dispersed, people join and leave the community, and multiple crowdsourcing platforms exists without pervasive identities.

Hence, a number of techniques and systems have been developed which aid the building and management of trust and reputation [20]—see [15] for a survey, and [6] for recent applications to open collaborative systems. Such systems help workers

[1] For a comprehensive and detailed discussion on different aspect and definitions of trust and reputation, the reader is referred to [26].

estimate their initial trust values for other workers based on evaluations of established authorities or majority votes. The intention is that the assessed trust values will exhibit a selective effect, encouraging workers to engage in interactions that will subsequently allow more precise personal trust assessment. As the accuracy of the trust values improves, so does the accuracy of the reputation metrics.

Trust and reputation systems can be highly context-specific [14, 18] and multi-faceted (see Fig. 1 for an example). Each worker is valued differently by each prior collaborator, in each context where they have worked. To form a full evaluation, the opinions of all of those collaborators should be taken into account. However, experiences are highly context dependant, and (for example) a worker's natural behaviours may align more closely with the norms of one context, leading to a higher perception of their quality in that context than others.

This context-dependency is a barrier to trust and reputation metrics being transferable between different platforms/projects. We use the term *reputation transfer* to denote any set of commonly-agreed and shared metrics, methods and data allowing a unified view of a worker's trust and reputation over different platforms.

Reputation transfer is one of the crucial requirements for emerging crowdsourcing systems and one of the important research questions that still needs to be addressed. Solving this problem would allow workers to maintain the reputation across different platforms and avoid platform lock-ins, thus allowing for a more stable future career as platforms come and go. The bootstrapping problem would be greatly reduced, as when a new platform starts up, an initial set of data is available. Overall, it would make the crowdsourced labour more attractive for skilled workers and complex tasks by allowing the workers to move their careers entirely to a competitive and fair crowdsourcing environment.

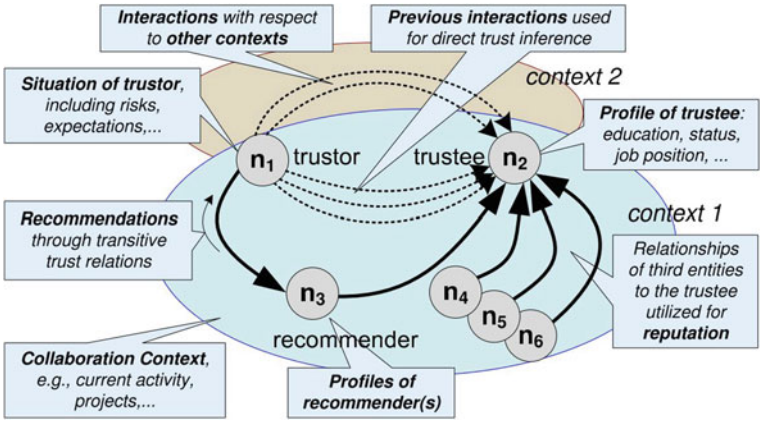


Fig. 1 Factors influencing trust in socio-technical systems (reproduced from [26]). When evaluating a *trustee* n_2, the *trustor* n_1 must take into account: bilateral interactions in this context; bilateral interactions in other contexts; relationships with others in this context; evaluations of recommenders (and the profile of the recommender)

However, solving the problem of reputation transfer is far from trivial. A general solution requires re-interpreting each worker's past context-dependent performance relative to the current context. The current context may have emerged since the design of the original performance tracking system, so there may be types of data which are unavailable. Additionally, as metrics change over time, behaviour changes to match them, so it becomes unfair to judge past performance on the standards of today.

The proliferation of different metrics and trust models indicate that agreeing on a uniform, context-independent trust and reputation model is practically unfeasible. A new approach and some out-of-the-box thinking will be needed take to address this problem.

2.2 Team Selection

In the crowdsourcing environment, large complex projects require cooperation between a number of crowd workers. As well as the issues of trust and reputation discussed previously, the composition of the team can have an effect on performance, and also the satisfaction of the workers who constitute the teams.

Kittur et al. propose that crowdsourcing labour markets can be regarded as a loosely coupled distributed computing system in with each crowd worker is analogous to a processor [16]. Just as it is important to organize distributed processors for various tasks, team formation and selection for crowdsourced software development is an important issue. Two major areas which should should be dealt with here are the possibility of self-organization of crowd workers, and the manner in which crowd workers are matched to tasks.

2.2.1 Self-Organization of Crowd Workers

Existing crowdsourcing platforms do not have much support for coordination and interaction among crowd workers. In some platforms like Amazon Mechanical Turk, tasks are separated in an atomic manner so that crowd workers do not need to collaborate with each other. Other platforms support complex projects with offline collaboration among crowd workers under the guidance of the work requester. However, creative work like software development requires a large degree of knowledge integration, coordinated effort and interaction among workers.

Self-organization is a process of formation of global order and coordination based on local interaction among individuals/components, which has been previously discussed in natural sciences, distributed computing environments, multi-agent systems, and so on. To deal with organization issues, Crowston et al. have previously studied self-organization of teams in open source software development [4], and illustrate the effectiveness of self-assignment of tasks based on the experiences in several projects. In software crowdsourcing, characteristics such as autonomy, decentralized control, emergence, and adaptation should be considered with respect to the organisation

of crowd workers. Methodologies for self-organization in multi-agent systems (e.g. [23]) can be applied to software crowdsourcing, and to some extent a crowd working platform can be conceptualised as a multi-agent system, where reorganisation happens "bottom-up", with no explicit central control; or, under an internal central control or planning by the work requester.

2.2.2 Task Matching for Crowd Workers

The task allocation problem has been discussed for decades in artificial intelligence and distributed computing circles. In crowdsourcing environments, recent researches focus on how tasks can be decomposed to allow modelling and execution as workflows with iterative tasks for the purpose of quality assurance [5]. However, in creative, complex crowdworking, the matching of tasks to workers is equally important. Two main factors should be considered: the skills possessed by crowd-workers, and the incentives needed to motivate them.

Chilton et al. investigate the task search behaviours of crowd workers, and find that workers tend to gravitate towards the newest tasks on offer due to user interface constraints of existing platforms [3]. Therefore, it is important that crowdsourcing platforms for creative complex work support task matching mechanisms to make full use of the skills of crowd workers. Anagnostopoulos et al. propose an optimization solution for team formation in social networks to deal with following requirements [1], which is also necessary in software crowdsourcing:

1. all skills required by the task should be satisfied;
2. communication overhead within the team should be small;
3. workload of tasks should be fairly balanced among people.

Another important factor in task matching is incentive of crowd workers, including both financial incentive and social incentives [24]. Therefore, tasks, skills, and incentives should be appropriately modelled when developing mechanisms for task matching in software crowdsourcing.

Additionally, the *Social Compute Unit* [7] provides a framework for creating teams of people and associated computing resources whose skills and incentives are matched to solving particular problems.

It will become increasingly necessary to provide mechanisms by which software crowdworkers can collaborate with people they know and trust; where they can organise themselves effectively as situations and contexts evolve; and where they are able to utilise—and improve—their skills on a variety of non-monotonous tasks.

2.3 Contextualisation

The context and purpose of software development can be a large motivating factor for workers; Bederson and Quinn [2] call for reduced anonymity on both sides, and

provision of task content. Similarly, Zittrain [28] discusses how decontextualized tasks remove the ability of workers to understand the moral valence of their labour, and decide whether the task they are carrying out is morally acceptable to them. Examples included range from spammers attempting to break Captchas to governments outsourcing recognition of persons of interest in photographs. A worker identifying people in CCTV photos or writing reviews of restaurants they have never visited might have a feeling that they are complicit in something ethically questionable:

> Do not do any HITs that involve: filling in CAPTCHAs; secret shopping; test our web page; … If you feel in your gut it's not on the level, IT'S NOT. Why? Because they are scams… spamgirl on TurkerNation [25]

Harris [11] presents a taxonomy of ways in which people can be hired to carry out morally ambiguous activities, and while software development is less prone to some of these issues, the strong emphasis on modular design in software makes it harder to divine the purpose of any particular code unit; a worker creating general computational infrastructure may well not give any clue about the intended purpose of the system.

When discussing general collective intelligence situations, Malone [17] describes the reward for taking part as being based on "Money, Love or Glory". The complement of this (leaving aside the pecuniary aspects) is that one should be engaged in a task that one does not hate, and is not ashamed of. Additionally, Malone suggests that commissioners of collective intelligence should engage with the design questions: What is being done? Who is doing it? Why are they doing it? How is it being done? These questions can be reversed to create a list of questions which crowdworkers should be able to ask, both for their own peace of mind and as a way for commissioning entities to engage with the Love and Glory motivations:

- *"What is the overall project?"* At a basic level, it is important to worker to know what the project it—are they building a recommendation system, or a face recognition system, or a social network? This allows the worker to understand their immediate moral or ethical stance with respect to the work, as well as building commitment and fostering pride in work done.
- *"Who is commissioning the work?"* What does the worker feel about the organisation that is asking for the work to be done? By avoiding anonymity, commissioners have the possibility to build loyalty within their atomized, cloud-based workforce.
- *"Why they are commissioning it?"* Beyond the simple specification of what the system to be built is, there is the question of what is it to be used for, and what are the overall goals and intentions of the commissioner. A worker might have different feelings about creating a data integration tool dependant on whether it was going to give people more useful information in their social networks, or be used by the government to catch criminals. Workers should be able to understand whether the goals of the project align with or conflict with their moral and ethical proclivities? Again, this feeds into developing a sense of pride in the work—beyond technical achievement in creating the software artefact, what is its effect on the world at large going to be?

- *"How is it being done?"* In the context of crowdsourced software development becoming a viable, sustainable career choice, the mechanics of the commissioning process are something with which ethically conscious workers will need to engage. As noted previously, the mechanism of outsourcing work to crowds can have a huge effect on the viability of the profession—if programming were to be carried out through competitions where many teams create solutions, but only one team gets paid, that would change the dynamics of the market. When thinking about longer timescales, workers may want to be selective about what kinds of system they engage with.

3 Discussion

The list of points we have raised here is far from complete. There is similar work in the literature, although much of it is aimed at crowd-work in general, and this tends to have a slant towards the Turking, micro-task end of the spectrum, and hence addresses a different set of communities and issues. Some good overviews are: [16], for describing several ways in which cloud work could be improved, and in particular highlighting the need for creating career ladders, through addressing questions of *motivation, job design, reputation* and *hierarchy*. Bederson and Quinn [2] discuss wage issues for cloud workers, but also provide a set of guidelines for improving the system as a whole, both in economic terms—disclosing pay, long-term feedback, price tasks based on time, grievance processes etc.—an non-economically, by providing task *context* and reducing *anonymity*; Harris deals with a related issue, looking at the ease with which nefarious employers can contract out illegal, unethical or just unsavoury activities [11].

Some of the key issues missing from this treatment are:

- We have discussed the need for trust and reputation between crowdworkers; there is also the need for accountability for commissioners of work. Requesters on Mechanical Turk currently are not bound by reputation systems. Silberman et al. [25] have been building systems for workers to track and comment on the qualities of the requesters, so that workers get a fairer deal.
- Traditional workers have the benefit of many organisational structures that support them. Labour laws ensure a safe and healthy environment; employers are tasked with managing the physical space that they inhabit in working hours; they will meet other people in they workplace; advocacy groups and unions may exist to represent the needs of workers. For a crowd working career, something providing some of the properties of these structures would need to be created.

A question which comes to mind given that the discussion is generally concerned with improving labour practices is: *Why are we interested in people who are typically relatively well off, rather than Turkers earning minimum wage?*. Arguably, most people who can participate in software crowdsourcing are quite well off, in that they have been able to become educated, computer literate, and highly skilled. Hence,

they are in a relatively strong position when it comes to protecting their rights and careers. However, it is exactly these qualities which make the software crowdsourcing industry worth engaging with: it is an articulate and visible industry, and as such, can lead the way in changing employment practices. There are battles which need to be fought, and entitlements which need to be won, and hopefully fighting the easier battles first can serve as a blueprint for other industries in the future.

Finally, beyond talking about this, what could we do to ensure that these things happen? Is it primarily computational infrastructure? Or are there social organisations that would need to be put in place? In general, in the area of socio-technical systems, it is necessary to program the people as well as the machines; there is a confluence of human behaviour change and computational support needed to create a sustainable, profitable, long-term and above all *humane* marketplace for crowdsourced software.

References

1. Anagnostopoulos, A., Becchetti, L., Castillo, C., Gionis, A., Leonardi, S.: Online team formation in social networks. In: Proceedings of the 21st International Conference on World Wide Web, pp. 839–848. ACM (2012)
2. Bederson, B.B., Quinn, A.J.: Web workers unite! addressing challenges of online laborers. In: Proceedings of the 2011 Annual Conference Extended Abstracts on Human Factors in Computing Systems—CHI EA'11, pp. 97–106. ACM Press, New York (2011). doi:10.1145/1979742.1979606
3. Chilton, L.B., Horton, J.J., Miller, R.C., Azenkot, S.: Task search in a human computation market. In: Proceedings of the ACM SIGKDD Workshop on Human Computation, pp. 1–9. ACM (2010)
4. Crowston, K., Li, Q., Wei, K., Eseryel, U.Y., Howison, J.: Self-organization of teams for free/libre open source software development. Inf. Softw. Technol. **49**(6), 564–575 (2007). http://linkinghub.elsevier.com/retrieve/pii/S0950584907000080
5. Dai, P., Weld, D.S.: Others: decision-theoretic control of crowd-sourced workflows. In: Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)
6. De Alfaro, L., Kulshreshtha, A., Pye, I., Adler, B.T.: Reputation systems for open collaboration. Commun. ACM **54**(8), 81–87 (2011)
7. Dustdar, S., Bhattacharya, K.: The social compute unit. IEEE Internet Comput. **15**(3), 64–69 (2011). doi:10.1109/MIC.2011.68
8. Felstiner, A.: Working the crowd : employment and labor law in the crowdsourcing industry (2010)
9. Frei, B.: Paid Crowdsourcing. Technical Report. www.smartsheet.com. http://www.smartsheet.com/paid-crowdsourcing-current-state-and-progress (2009)
10. Gassenheimer, J.B., Siguaw, J.A., Hunter, G.L.: Exploring motivations and the capacity for business crowdsourcing. AMS Rev. 1–12 (2013). doi:10.1007/s13162-013-0055-8
11. Harris, C.G.: Dirty deeds done dirt cheap. In: 2011 IEEE International Conference on Privacy, Security, Risk and Trust, pp. 1314–1317 (2011)
12. Horton, J.J.: The condition of the turking class: are online employers fair and honest? Econ. Lett. **111**(1), 10–12 (2011). http://www.sciencedirect.com/science/article/pii/S0165176510004398
13. Howe, J.: The rise of crowdsourcing. Wired Mag. **14**(6), 1–4 (2006)
14. Josang, A., Ismail, R., Boyd, C.: A survey of trust and reputation systems for online service provision. Decis. Support Syst. **43**(2), 618–644 (2007). doi:10.1016/j.dss.2005.05.019. http://linkinghub.elsevier.com/retrieve/pii/S0167923605000849

15. Jøsang, A., Ismail, R., Boyd, C.: A survey of trust and reputation systems for online service provision. Decis. Support Syst. **43**(2), 618–644 (2007)
16. Kittur, A., Nickerson, J.V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M., Horton, J.: The future of crowd work. In: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, pp. 1301–1318. ACM (2013)
17. Malone, T.W., Laubacher, R., Dellarocas, C.: The collective intelligence genome. MIT Sloan Manag. Rev. **51**(3), 21–31 (2010). http://raptor1.bizlab.mtsu.edu/S-Drive/KJIH/2010StudyAbroad/JournalArticles/TheCollectiveIntelligenceGenome.pdf
18. Marti, S., Garcia-Molina, H.: Taxonomy of trust: categorizing P2P reputation systems. Comput. Netw. (April 2005), 1–20 (2006). http://www.sciencedirect.com/science/article/pii/S138912860500215X
19. Prendergast, C.: The provision of incentives in firms. J. Econ. Lit. **37**(1), 7–63 (1999). http://www.jstor.org/stable/2564725
20. Resnick, P., Kuwabara, K., Zeckhauser, R., Friedman, E.: Reputation systems. Commun. ACM **43**(12), 45–48 (2000)
21. Ross, J., Irani, L., Silberman, M., Zaldivar, A., Tomlinson, B.: Who are the crowdworkers? Shifting demographics in mechanical turk. In: CHI'10 Extended Abstracts on Human Factors in Computing Systems, pp. 2863–2872 (2010). http://dl.acm.org/citation.cfm?id=1753873
22. Schenk, E., Guittard, C.: Towards a characterization of crowdsourcing practices. J. Innov. Econ. Manag. **1**(7), 93–107 (2011). http://www.cairn.info/revue-journal-of-innovation-economics-2011-1-page-93.htm
23. Serugendo, G.D.M.: Self-organisation and emergence in multi-agent systems. Knowl. Eng. Rev. **20**(2), 165–189 (2005)
24. Shaw, A.D., Horton, J.J., Chen, D.L.: Designing incentives for inexpert human raters. In: Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work, pp. 275–284. ACM (2011)
25. Silberman, M.S., Irani, L., Ross, J.: Ethics and tactics of professional crowdwork. XRDS: Crossroads ACM Mag. Stud. **17**(2), 39 (2010). doi:10.1145/1869086.1869100
26. Skopik, F.: Dynamic trust in mixed service-oriented systems. Ph.D. Thesis (2010). http://hydra.infosys.tuwien.ac.at/Staff/sd/papers/Diss.F.Skopik.pdf
27. Software developer salary in United States. http://www.indeed.com/salary/q-Software-Developer-l-United-States.html. Accessed 21 Nov 2013
28. Zittrain, J.: Ubiquitous human computing. Philos. Trans. Ser. A Math. Phys. Eng. Sci. **366**(1881), 3813–3821 (2008). doi:10.1098/rsta.2008.0116

Part II
Software Crowdsourcing Models and Architectures

Bootstrapping the Next Generation of Social Machines

Dave Murray-Rust and Dave Robertson

Abstract The term "social machines" denotes a class of systems where humans and machines interact so that computational infrastructure supports human creativity. Flagship examples such as Wikipedia and Ushahidi demonstrate how computational coordination can enhance information sharing and aggregation, while the Zooniverse family of projects show how social machines can produce scientific knowledge. These socio-technical systems cannot easily be analysed in purely computational or purely sociological terms, and they cannot be reduced to Turing machines. Social machines are used in the creation of software, from software crowdsourcing projects such as TopCoder and oDesk, to distributed development platforms such at GitHub and Bitbucket. Hence, social machines are increasingly used to create the software infrastructure for new social machine. However, social machine development is a more complex process than software development, as the community must be "programmed" as well as the machines. This leads to development in the context evolving and unknown requirements, and having to deal with more sociological concepts than formal systems designers usually work with. We hence model the process using two coupled social machines: the *target* social machine, with whatever purposes the creators envisions, and the *development* social machine which is used to create it. As an example, oDesk can form part of a development social machine which might be used to create a target social machine, e.g. "the next Facebook". In this chapter, we describe a formalism for social machines, consisting of i) a community of humans and their "social software" interacting with ii) a collection of computational resources and their associated state, protocols and ability to analyse data and make inferences. We draw on the ideas of 'desire lines' and 'play-in' to argue that top down design of social machines is impossible, that we hence need to leverage computational support in creating complex systems in an iterative, dynamic and emergent manner, and that our formalism provides a possible blueprint for how to do this.

D. Murray-Rust (✉) · D. Robertson
CISA, School of Informatics, University of Edinburgh, Edinburgh, Scotland, UK
e-mail: d.murray-rust@ed.ac.uk

D. Robertson
e-mail: dr@inf.ed.ac.uk

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_4

1 Introduction

With the rise of the web, and social networking, there has been a shift from thinking about systems which are programmed and then used to systems where users engage with their computational environment and each other for a variety of purposes, from frivolous banter on Twitter to crisis management in Ushahidi. In many cases, *interaction* in its many forms is now the central tenet of system design.

The term "social machines", introduced by Berners-Lee [3, 9] describes a broad class of systems "...in which the people do the creative work and the machine does the administration". Increasingly, the line between algorithms and human behaviour is blurring, so that in order to understand and develop current and future systems, an understanding of distributed complexity must be combined with insight into incentives and motivations—systems are no longer reducible to Turing machines. Flagship projects such as Wikipedia, Ushahidi and Zooniverse demonstrate the power of integrating human and machine intelligence to both create and make sense of knowledge.

As a result of this change in thinking, and the growing complexity in interconnectedness of systems, the practices used to develop architectures and infrastructures must change. On the one hand, it is necessary to include a greater understanding of the end user community and its behaviour throughout the development process. It is increasingly difficult to write requirement specifications ahead of time, as they must evolve in response to evolving user populations and cultures, and the scope has to expand to include human behaviours and motivations as well as software functionality. On the other hand, development is becoming more distributed, with larger, more diffuse teams, increasingly dependent on computational coordination to glue them together. Essentially, we need to use social machines to create future social machines.

A wide spectrum of activities is carried out through integration of human and machine intelligence, or "collective intelligence" (CI). Social machines occupy a part of that space, where interaction and flexibility are key [27]. Another area in this space, overlapping with social machines is crowdsourcing, where an evolving group of participants is incentivised to carry out a common task. The utility of crowdsourcing for simple tasks has been well demonstrated; additionally, the crowdsourcing of complex, creative tasks has seen a dramatic growth in recent years. 99 designs is a notable example, providing a platform for running design competitions.[1] In a similar vein, TopCoder[2] decomposes "the entire digital lifecycle" into small, self-contained tasks, and completes these by running competitions. In contrast, oDesk offers a more traditional labour marked model, where jobs are posted, freelancers apply, and the commissioner of the work selects someone to carry out the work.[3]

[1] http://99designs.co.uk/how-it-works.

[2] http://www.topcoder.com/whatiseoi/.

[3] https://www.odesk.com/info/howitworks/client/.

These software crowdsourcing systems are social machines for software development—machine intelligence is used to coordinate human activity. However, the emphasis is on human management to decompose and split up tasks, rather than engagement with the end user community. Even systems designed to perform simple tasks rely on a community of people, as well as computational infrastructure to tie them together. A key question hence becomes how to manage and influence the community to achieve the desired results [11]. Essentially, one must think in terms of programming social computers, and carry out tasks such as maintaining groups and convincing people to take on roles as well as simply initiating computations [25]; this prompts a move towards attempting to understand the science behind the magic of crowd based systems [13] and how the communication between people is as important as their individual or aggregate attributes [20]. To create a map of the CI space, Malone et al. used analysis of over 250 CI systems to construct a CI genome, defining the current solution space for the four major components of such systems: "What is being done? Who is doing it? Why are they doing it? And, How?" [12].

In this chapter, we are concerned with techniques that can aid in the development of the next generation of complex web based systems—social machines. This means understanding (i) how the *development* social machines can be set up to create software infrastructure and (ii) how the *target* social machines can be analysed and monitored and brought into the development process.

2 Social Software Versus Machine Software

In the social machines discussed, part of the operation of the system is defined by the affordances and pathways of the technical infrastructure—the machine software—and part is defined by the social and cultural behaviour of the users—the social software. To get a sense of the distinctions:

- When using a distributed version control system (DVCS), the machine software provides a set of technical capabilities around creating new versions of source code, annotating these with comments and structuring them using repositories, branches and tags. It is up to any community to then formulate their social software to define when things should be committed and pushed or pulled, which repository (if any) is the master, who can commit, how pull-requests should be dealt with etc.
- Wikipedia provides quite basic machine software for collaboratively developing articles. Editing and version histories are provided, as are some tools such as locking pages from excessive editing. However, they have extensive—and well documented—social software to decide which articles can be created, which edits are appropriate, who can arbitrate on decisions etc.
- The Polymath blog collaboratively solves research level mathematical problems [14]; the only machine software they use is a standard blogging platform which supports Latex equations, and a wiki. The social software built on top of this is generally around the process of carrying out the work, with some additional ideas such as extracting finished outcomes from the blog comments to the wiki summary.

From the point of view of a system designer, a natural question to ask is what determines whether functionality should be carried out socially or mechanically. Arguably, there is spectrum on which any operational feature can be placed, from un-elucidated social norms, through general agreement to formal specification of varying degrees and finally mechanical implementation. In general, following the ideal in the vision of social machines given above, mechanisation of uninteresting processes is desirable. Hence, we cast the question as "what prevents mechanisation of processes encoded as social software", and offer several reasons:

- *There is no mechanical solution.* Deciding whether an edit on Wikipedia represents a neutral point of view computationally is currently not possible, even though it has been strongly formalised in the community.[4]
- *The desired functionality cannot be adequately formalised.*
- *The desired functionality is not known.* In many crowdsourcing systems, the desired macro-scale behaviour is well known, but the mechanisms by which to create it are not known.
- *There are many different use cases.* In DVCS systems, there are many different ways in which the end users need to use the software to fit in with their own organisational structures (or lack of them). In this case, it is appropriate to provide a generic set of functionality, and add structure through social software.

In many cases, there is a tension between the overall purpose of the system and constructing and maintaining the social fabric which allows it to function. For instance, Wikipedia editors tend to be very strict when analysing changes made, and are quick to revert unacceptable edits in the hope of maintaining high quality articles. However, this can also have the effect of disincentivising newcomers, and reducing the available pool of editors [7]. Caring for the social fabric of a crowdsourcing platform can be a complex process—Galaxy Zoo analyses the strength of at least 13 different motivations for participation, across a diverse user population [21], while an even larger set of motivations has been explored for the Zooniverse platform [22]. As a result, the Zooniverse developers need to be able to experiment with both small and large changes to the system; often, the changes which are most important to maintaining the community are tangential to the main activity being carried out [1].

Taken together, these trends point to a world where software must increasingly be developed with a fuzzy and changing user community; dynamic, unspecifiable requirements to which the outcome is highly sensitive; and by an anonymous group of geographically diverse creators. We have a situation where pre-existing development methodologies struggle, but we are dealing with developers and users who are used to working with distributed, computationally mediated systems. Social machines provide a useful abstraction here, to look at both the development and execution of these systems.

[4] Wiki editors will use comments such as "violates WP:NPOV" to flag offending edits.

3 Social Machines for Social Software Development

We are concerned with using social machines to generate new social machines; a *development* social machine is used to create the socio-technical infrastructure of a *target* social machine. For example, a system like oDesk (the development social machine) could be used to manage the development of a Zooniverse style crowdsourcing system (the target social machine).

We will look at this in terms of constructing "social artifacts" to integrate with communities of engaged participants. Artifacts have been used in the multi-agent and computer supported collaborative work (CSCW) communities as a means of enhancing the coordination of groups of agents, and providing automation which the more goal directed actors in the system can leverage [19]. We use the term "social artifacts" to differentiate those which are explicitly designed to be used by groups of people, by providing the mechanical infrastructure of a social machine. There is also strong overlap between our term social artifact and the electronic institutions described in [5].

Figure 1 gives a pictorial representation of a pair of social machines; the upper "creation" social machine is used to construct and maintain the lower "target" social

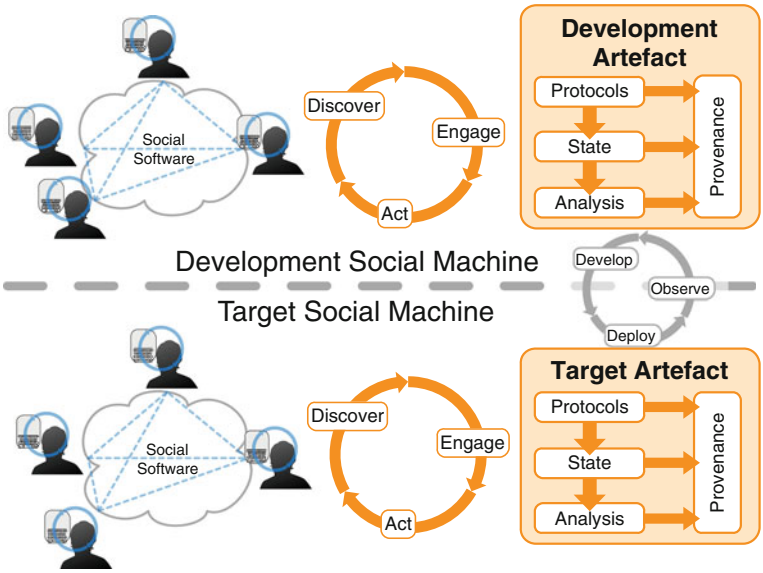


Fig. 1 The development of social software can be seen as an interaction between two social machines. The top artifact coordinates the development of software, providing tools for the organisation of developers. The lower artifact is the software produced, which coordinates the activity of a population of people around some common task. The coordinator can observe the state changes of both of these, and make use of inference to understand the actions of both communities. The bottom picture shows some pathways through this representation described in the following examples

machine. Each machine is composed of a community of participants and a social artifact, which carries out the mechanical side of the social machine. The artefact takes over mechanical parts of tasks such as calculation, bookkeeping, message passing, data analysis, digital asset management etc. allowing the humans to focus on the creative side of their activities.

The community of people (on the left hand side of each machine) is defined at two levels. Individuals have individual qualities—goals, skills, preferences etc. which are heterogeneous across the population. The community as a whole has some level of "social software"—the conventions and practices which the people engage in. This social software may be explicit or implict; it may have different levels of uptake among different groups; people may disagree on what exactly it should be; but in most successful communities there are some norms and behaviours which emerge and are important to the dynamics of that community.

In lieu of a formal specification, we will assume that artifacts are characterised by:

- Some **internal state**; this is likely to be a combination of desired output and bookkeeping. In a social network it would include a graph of people and their relations, messages sent and received and user profiles. In a software development situation it includes all of the source code produced, team assignments or other organisational structure and any personal profiles. Generally, the state of any particular interaction will be maintained by the artefact, removing overhead from the participants.
- **Protocols** to execute, which define the behaviour of the artefact. In a social network, these would specify what happens when one user sends a message to another, or searches for "friends called X who live in Y", or un-friends someone. In a development environment, protocols would include best practices around committing to source repositories and high level development workflows (e.g. waterfall model, Agile development). Specifying protocols allows the system to know and explain:
  - ways to initiate allowed operations,
  - the effects of those operations,
  - the actors involved in any given operation.

For social artefacts, we include two further desirable features. These are not hard requirements, but part of making systems transparent and trusted by their human participants:

- **analysis**, to aggregate and describe user behaviour and state changes in a way which is useful both to machines and people. This helps the system to explain itself to the participants, which can form part of an incentive mechanism, provide feedback for more effective use and development, or simply be transparent about what is happening at any given point in time.
- **provenance**—support for annotating the current state with where data came from, how it has been used, and by whom. This is part of a system's ability to explain why the current state has been reached, and provides a foundation on which to build accountability and trust within the system.

These qualities can be seen towards the right of Fig. 1: protocols (triggered through some form of interaction) modify the internal state of the artifact. These state changes are then analysed, and provenance is maintained at every step.

It is important in this discussion that the protocols embodied in artifacts are first class objects. This means that the interactions carried out using the artifact can be sufficiently formally described that they are separate from the code of the artifact itself. The artifact provides an execution environment for an interaction protocol, by providing an interpreter along with any necessary storage of state and communication mechanisms. Making protocols first class objects give many advantages—see [16] for an overview—but key here is that it allows many operations to be carried out on them: they can be examined for a security analysis; modified based on a richer interaction model; analysed as components of successful social machines; discovered by and shared with other systems which would like to re-use successful protocols and so on. Extracting out interactions supports artefacts in being self modifying—since interaction protocols are first class objects, the artefact can contain protocols for modifying it's own protocols—a vital ability for systems in a dynamic and changing environment.

In order for humans to interact with these artifacts, to make a coupling between the user community and the social artifact, some steps are necessary:

- A user must **discover** an appropriate artifact to interact with, according to the user's desires. For developers, this might be selection of TopCoder or oDesk as a means to acquire currency; for the general public, this might be selection of Galaxy Zoo as a place to contribute to scientific knowledge.
- The user must then **engage** with the artifact, through understanding enough of its functionality to be able to select and commit to a role in of the protocols. This might be signing up as a freelancer on oDesk, or getting an account on Zooniverse.
- The user can then **act**, mediated through the artifact, by enacting the chosen role, possibly in concert with other users. This could be bidding for, winning and then completing an oDesk task, or classifying a stream of photographs on Galaxy Zoo.

There are, of course, shortcuts to this: once an artifact has been discovered, actors can re-engage without having to rediscover it; similarly, many actions might be carried out when performing a single role (or repeating a role indefinitely). However, all of these steps are necessary for interacting with any new social artifact. These steps are shown in the centre of Fig. 1, as the coupling mechanism between the community of participants on the left and the social artifacts on the right.

In this chapter, we use the Lightweight Coordination Calculus (LCC) [23, 24] to describe the protocols contained in these social artifacts. LCC has been used for general coordination, as well as to synthesise argumentation protocols [15], carry out knowledge sharing in peer-to-peer architectures [28] and to add computational support to interactions on Twitter [17].

LCC was designed to capture the idea of an institution: to allow the specification of interactions separately from the agents involved in them. It attempts to avoid

over-specification in terms of either ontologies or performatives, to provide just enough coordination for agents to collaborate effectively. As well as being relatively concise, as a declarative specification, LCC has the useful property of being both normative and executable—as well as describing what *should* happen, with a suitable interpreter it can be run directly.

The main concepts defined within LCC are:

- The roles that actors take, along with any necessary parameters
- Messages sent from one actor to another, specified as both content and the role of the sender/receiver
- Sequencing and choice between blocks of actions
- The flow of variables through interactions
- Other useful computational elements such as mathematics, filtering, solving for variables (which can include general code execution), building and unpacking lists etc.

However, the particular language used is not important, and other similar languages exist, e.g. [26], and translations have been made from LCC into BPEL4WS and the ISLANDER electronic institution specification system [24].

Figure 2 gives a quick sketch of how these ideas are represented syntactically in LCC.

We will use these formulations to explore the question of how a social, artifact-centric view of software development can lead to a more dynamic development process, including transitions from social to machine software.

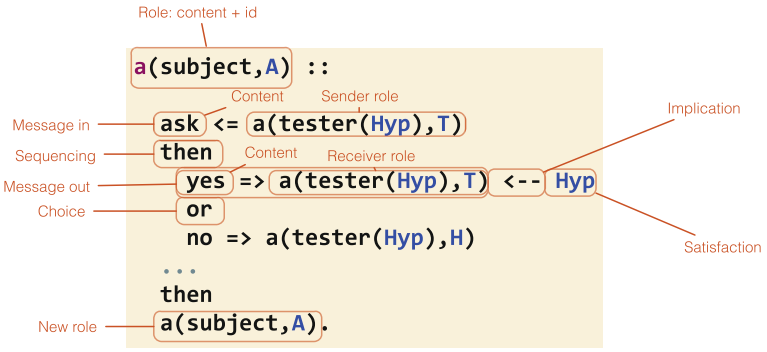


Fig. 2 Annotated example fragment of LCC. This is from a distributed hypothesis testing protocol, where one agent wants to get the opinion of many of it's friends about a hypothesis. This fragment denotes the behaviour required of an agent to be a test subject. An agent `A` can be a subject if it waits for a message from a tester (`T`) with a hypothesis `Hyp` in. The tester should then see if the hypothesis holds true for it—i.e. attempt to satisfy it. If it does, it should send a message saying `yes` to the tester, if not, one saying `no`. After the message has been sent, it should then take on the role of being a subject again, ready for the next hypothesis

4 Illustrative Scenarios

Here we describe two existing collective intelligence platforms, and provide example scenarios showing how an artifact centric view of their development would work, and what benefits it would bring. These are hypothetical descriptions rather than implemented systems—thought experiments to illustrate how a system along the lines suggested here could be used.

4.1 Learning Development Practices

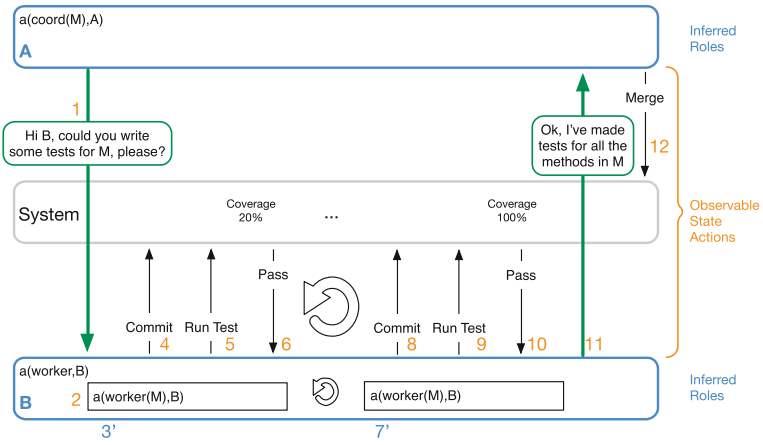
The Collabode editor supports real-time collaborative software development [6], allowing multiple users to work on the same collection of files (project) simultaneously. The biggest gains were seen in "microtasking", where one main user was responsible for structuring the code at a high level, and then parcelling out development and testing of individual methods or components. The Collabode authors suggest that it was not possible to design how this microtasking should be organised ahead of time; looking at the synchronisation between a task organiser and a test developer, they say:

> Between these two extremes, we are continuing to prototype new synchronization strategies. The key idea is to use signals for broken code compilation errors, failing test cases to determine whether a given contributor's changes are ready for sharing or not, and if so, to share them automatically.

A typical test development procedure might run as follows:

1. Adam asks Bob to write some tests for a module
2. Bob writes a test for the first method
3. Bob runs the test and it passes
4. At this point, the tests all pass, but the code coverage of public API methods is minimal
5. Bob then creates tests for the rest of the methods
6. Bob finds that several of the tests fail (but now top level coverage is 100 %)
7. Bob messages Adam to say he's completed the tests, and several of them fail
8. At this point, Adam merges in Bob's changes, to be able to modify the main code to pass the tests

Typically, this would be carried out some kind social software—agreements and coordination between Adam and Bob. However, as projects become large and distributed, this kind of social software becomes harder to maintain. In a fully distributed, crowdsourcing system, developers may well not know each other, common languages (both computational and natural) are not a given, and generally, increasing amounts of coordination need to be carried out mechanically.




```
Init  a(coord(M),A) ::                                          0
   1  create_tests(M) => a(worker,Tester) <- select(M,Tester)
      then
      merge(Tester) <- complete(M) <= a(worker,Tester).11

Init  a(worker,B) ::                1
   2  a(worker(M),B) <- create_tests(M) <= a(coord(M), A )
      then
  11  complete(M) => a(worker(M),A).

   2  a(worker(M),B) ::
      (
 3',7'  create_test(Method,T) <- not(tested(M,Method)) then
 4,8    commit_test(Method,T)                          then
 5,9    check_test(Method,T,R)                         then
6,10    a(worker(M),B)
      )  or null .X
```

Fig. 3 Observed sequence of development actions (*top*) and potential LSC program to be inferred. The collaborators start as a general worker and a coordinator of tasks for a particular software module M (Init). A decides to ask B to write some tests (0), and sends a message through the coordination software to that effect (1). B accedes (2), and goes round a cycle of creating code locally (3′), then checking it in and running it. When there are no more methods left to test, B sends A a message to this effect (11) who then merges in the changes (12). Events and processes marked in *blue* are not formally defined, while, those in *orange* represent changes of state for the social artifact. At point (X), since no value can be found to satisfy not(tested(M,Method)), there are no untested methods left, so the role is exited by the *null* instruction, and B returns to being a general worker

An alternative approach is to define this as a formal workflow; The bottom part of Fig. 3 gives a possible representation of this interaction using the LCC language. Three roles are specified:

- a coordinator (coord) responsible for code within a particular module (M);
- a general worker who can be assigned to different tasks;
- a worker who is specifically working on creating tests for module M;

Within this formulation, three broad classes of action can be seen:

- *actor/actor communication*, where the two developers message each other to coordinate actions and exchange information or requests;
- *state modification*, where an actor triggers changes in the state of the development artifact, for example committing code;
- *external input* or *computation*; here agent A must decide which of the workers to put to work on module M; agent B needs to actually create the test code before submitting it.

This immediately brings up two intertwined questions: is it the only way to request tests? and where might this formalisation come from?

Firstly, it is clearly not the only way that this kind of work could be parcelled out: Bob might refuse to create the tests; Bob might only be able to write some of the tests, at which point Chloe would have to take over; Adam might merge in Bob's code before all of the tests were written; Eva might be working on the module at the same time, so in between Bob running out of methods to test and Adam carrying out a merge, a collection of new methods are added. This is a case where, especially in the case of developing a new *kind* of collaboration environment, the range of desirable possibilities is unlikely to be clear ahead of time.

Secondly, while the protocol could be hand coded, the system would be far more flexible if the protocols could be inferred, by analysing natural human interaction and generalising this to create an abstracted formal representation. To give an indication of plausibility, the top part of Fig. 3 shows the observable interactions and state changes triggered by the two developers separated from their internal state and ex-system activities. The things that need to be inferred are:

- The interaction is bounded by two messages and a trailing commit—and here, the closing commit might or might not be part of the interaction; also, it would be possible to ask the developers to indicate when interactions begin and end. This leads to the main `coord` and `worker` roles, as well as the two messages that bind the coordination together.
- There is a variable M representing the module to be examined—it is present in the initial message, and all of B's actions relate to code in this module. This gives us the unbound `M`s that are threaded through the code.
- That B takes on a repetitive role related to writing code, committing it and running tests—this would be built up over multiple interactions, and a real specification would allow for more variation here. This leads to defining the `worker(M)` role, for working on a particular module. The detail of the internal state representation would affect the scope of possible inference here: could the system somehow divine that B was creating tests for untested methods one by one, and create the `not(tested(M,Method))` clause, or just that B was writing some code and checking it in?
- B stops and sends a message once complete API coverage is achieved—in a software development environment, that is likely to be a key, visible metric, which

would naturally be foregrounded in any inference routine. Depending on the inference, this might be a `not(tested(M,Method))` clause failing to satisfy as above, or it could be `coverage(M) < 1.0` or similar.
- At the start of the interaction, A selected B to create the tests. It may be impossible to infer how this selection came to be. However, it is clear that *some* selection needed to be done, since another person was contacted, and that since the variable `M` is involved, that should be part of the selection, leading to the signature `select(M,Tester)`.[5]

Given all the components in this list, it is not much of a leap to the formal specification at the bottom of Fig. 3—indeed all the components except the initial roles and the selection procedure have been covered. So now the selection procedure is left as the part where human input is required—when initiating this protocol, A would be asked, somehow "Choose a worker you would like to create tests for you". The computational infrastructure would then take care of all of the bookkeeping, up until it could say "Right, all the methods are complete, why don't you merge in the tests."

This is a purposefully simple example, which is necessary to explain in detail how all of the linkages would be constructed. Also, we have focussed on arguing that the inference is completely automatable. In a real world, we would hope that more complex tasks could be automated, and that ways to manage tasks could be created, leading to high level workflows like Agile processes or even traditional software development practises, depending on the community and their needs. At the same time, we would expect a greater degree of human-computer interaction, in deciding what should be extracted, refining example workflows and so on.

4.2 Mechanisms for Carrying Out Crowdwork

The Zooniverse project[6] is designed to build on the success of Galaxy Zoo[7] in convincing humans to carry out microtasks for the advancement of science. Where galaxy zoo focussed on a single application area (detection of galaxies), the Zooniverse commoditises this approach, and allows for the creation of many domain specific experiments. Part of the goal of Zooniverse is to understand the dynamics of crowdsourcing communities, so they need to be able to experiment with different approaches to initiating, building and maintaining communities.

Snapshot Serengeti[8] is a typical Zooniverse project, where users are asked to look at automatically captured photographs of the savannah and identify the number and species of animals along with their activity. Once introductory training has been

[5] LCC is Prolog inspired; in imperative programming terms, the signature would be more akin to `Tester = select(M)`.

[6] http://www.zooniverse.org/.

[7] http://www.galaxyzoo.org/.

[8] http://www.snapshotserengeti.org.

completed, users carry out the microtasks of classifying images according to the number and activity of animals in them.

On the surface, this is a simple setup: after going through the introduction process, each user repeatedly carries out tasks. However, this is where programming the social software becomes deeply important—creating the community and incentives which draw people back to a highly repetitive task. Taking an artifactual view, each atomic task is represented by a protocol describing inputs and outputs; these atomic tasks are then sequenced together into larger units that describe the movement through the universe of possible activities. The manner in which these protocols are arranged can have a huge influence on the people using the system: do they provide incentives to carry out work? of what kind? is the system gamified and reputation based? is there variation in the tasks to be done? is communication encouraged? and how? are newcomers gently guided in? is there a transition arc from newbie to old hand?

Figure 4 illustrates how these tasks might be sequenced if we were to implement it using social artifacts. We start the Discover-Act-Engage cycle here by assuming that users have already found the website, so the first task is *engagement*, where users

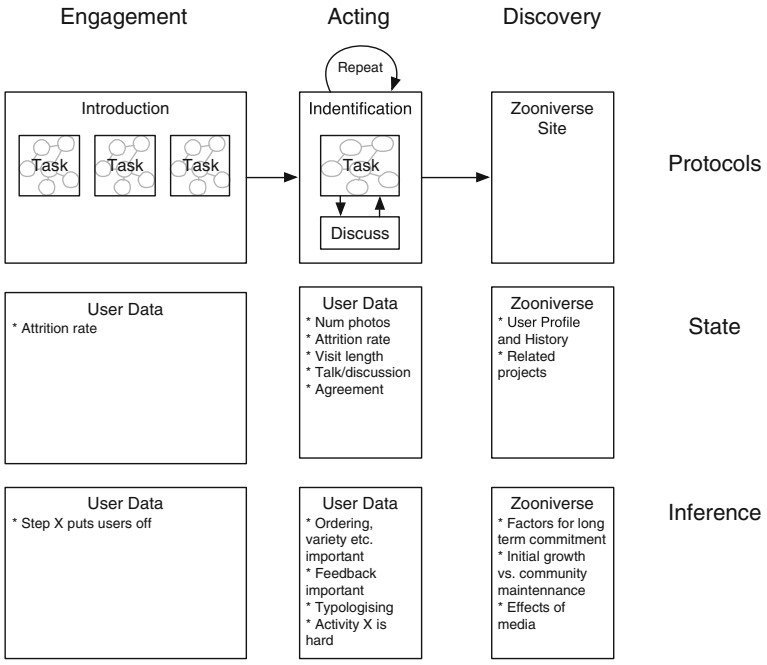


Fig. 4 Overview of the operation of Snapshot Serengeti, along with hypothesised states and inferences which could be recorded. When a new user joins the site, they first carry out a series of introductory tasks to familiarise themselves with the user interface and the domain of classification. They then start identifying animals, with a constant option to look at the discussion page for each photograph. There is also always the possibility to join general discussion groups, and explore the wider Zooniverse community

are gently led into the system via an interactive tutorial. Next, they progress onto actually carrying out the task at hand, and *act* by tagging photos; while doing this, they have the opportunity to break out into discussion of a particular photo. When they have had enough tagging, they might explore the discussion section of the site, or the wider Zooniverse, leading them to *discover* more machines and communities to engage with.

As well as the protocols which are enacted, the social artefact keeps track of state. In addition to the direct outcomes (i.e. tagged photos), social machines of this type are beginning to be instrumented, to allow for analysis of their users: how long do they carry out tasks for? Are there particular points where they drop out of the system? The logical next step from observing user behaviour is to influence it and evaluate the changes; and this is a particular point where computational infrastructure can have a huge impact. Many large companies, notably Google are basing development on computationally mediated large scale experiments such as A/B testing, where users are randomly selected to receive control or test versions of a page. In order to carry out well grounded experiments, it becomes necessary to have formalised protocols to manage the deployment of nested test conditions across a changing population [29].

The bottom of Fig. 5 shows how this kind of workflow could be built based on a social machine style analysis of the combined development and target systems. User behaviour on the existing site is tracked and analysed, and the target system can then carry out some inference to determine e.g. where most people are lost on signup? what are the most problematic tasks? are there discrete user groups with different requirements? The results of this inference are then visible to the human developers and the automatic systems which form part of their development environment. Working with the protocols and computational intelligence embedded in their development artifact, the developers can then identify the most important points to address, and devise a computational experiment to test it. The code modifications are then made—as state changes to the development environment—and then deployed according to the computational experiment which has been designed. The experimental protocol then observes the effects on the community, and feeds that back into the development process.

This is just one pathway through the coupled social machines. Other pathways could include user consultation, or be developer led, or be purely automated. The example presented is a single illustration of the kinds of development process which can be represented and modelled by taking a coupled social machines approach.

5 Discussion

Throughout this chapter, we have suggested that enabling transitions between social and machine software is a desirable course of action. Social software can arise dynamically and change in response to the evolution of the community; what then is the benefit of formalising this? We suggest several advantages:

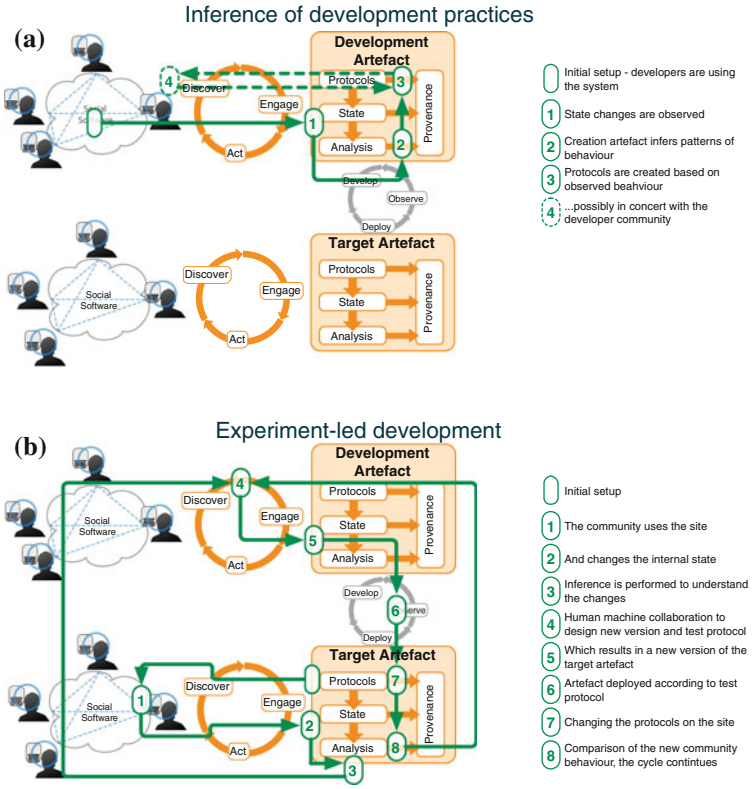


Fig. 5 Two possible paths through connected social machines for dynamic development. **a** When learning development practices, existing development is observed, and the behavioural patterns behind it are inferred. This inference is then converted into a formal protocol which can be re-used and shared as desired. **b** For Zooniverse style systems: Based on user behaviour, patterns of behaviour can be inferred, which are analysed by both human developers and the machine protocols which they are using. This results in a testing plan that allows for instrumented deployment of the revised social protocols to analyse the effects of the modifications

- *Re-use*: by codifying behaviours deemed useful, they can be shared with others. This avoids multiple groups having to recreate the same ideas, and gives communities who just want to get things done a toolkit of tested techniques to work with. This is similar to the way that knowledge sharing protocols were shared in the OpenKnowledge project [28]. If a particular technique for committing source code, or losing less users in introductory stages, or integrating the results of computational analysis is extracted, it can then be shared with others facing similar issues; in the case of Zooniverse, future site developers would have a library of design patterns to draw on.
- *Evaluation*: if social workflows can be formalised, it becomes possible to track when they occur, who is part of them, what the outcomes are and so on. This

supports a quantitative analysis of what works for particular communities. If this analysis is done over well defined, widely used protocols, then the results will have greater impact. When *provenance* information is included, questions can be asked post-hoc about which communities they worked for, what kinds of people use particular techniques, or avoid certain ways of working.

- *Objectivity and explanation*: providing a canonical model of behaviour provides an object for examination—does the community agree on the process? Does everyone consider it fair? If the model can be shown to new users, it may shorten their learning curve when attempting to synchronise with an established community with complex rules, and hence lower the barrier to entry.
- *Automation and guidance*: defining complex procedures formally allows for the mechanical parts to be carried out automatically—getting the machines to do the bookkeeping. At the same time, having a representation of the state of any given process, and the possible next moves, allows for machine support of human decision making, offering up possible courses of action which would advance the interaction.

A central idea here is that natural human behaviour can be codified into formally executable specifications. In this case, it makes sense to start with social software—one can provide open ended tools, and allow users to explore what they would like to do with them. This bears a similarity to end user programming [10]: an open system is created, which the developers can configure as they see fit. A closer relationship is to the *desire lines* used by designers to understand how people want their creations to be. In "The Oregon Experiment" [2], Alexander grapples with an analogous problem to that of creating computational architecture for communities—that of creating physical architecture for communities. In particular, he states that while a masterplan can create a *totality*, it cannot create a *whole* [2, pp. 10]—that imposing a complete order on human activities leads to chaos. His answer is to invoke organic growth; for example, to leave empty, open spaces and observe the routes that people take through them, before codifying these "pathways of desire" into paved paths. This principle has been applied to computer human interaction many times—see [18] for some examples; the author examines desire lines in everyday life, and relates them to the operation of computer software. Of particular interest is the idea that it is the *traces* of behaviour which constitute the desire line: the physical manifestation in trampled grass, mis-filled forms or smashed cars. Part of the artifactual view of computing is the capability and necessity of maintaining history, of instrumenting interaction so that exactly these traces can be recorded, analysed and generalised. However, given the size, complexity, detail and technicality of traces in social computational systems, few humans are able to recognise and react to the desire lines encoded therein; it is hence essential to bring computational machinery to bear on the problem of extracting useful patterns from human behaviour.

Another viewpoint in this area is Harel's concept of *play-in* [8]; here, initially dumb interfaces are programmed by enacting short scenes, and then explaining to the device what *should* have happened:

> Hey phone, whenever I press this key and hold it down for at least a half-second, you should switch on—meaning that your display should light up and show the cellular provider, my name, and the time.

From multiple interactions such as this, the complete behaviour of the system can be defined, in Harel's case by creating a sequence chart [4]. Unfortunately this style of programming has not become widespread, possibly due to the large number of examples necessary to build up complete patterns of behaviour.

The immediate question is then what differentiates our social machine, artifact oriented view of software development from Harel's *play-in* such that we feel confident arguing its practicality? And more generally, why do we feel that the problem of inferring useful, desired patterns of behaviour is soluble in this area.

Firstly, what does it mean for an inferred pattern to be useful? This can take many forms: noticing a common pattern in messages and commits could allow a particular coordination workflow to be generalised, which allows for automation and sharing; noticing a common pattern of social interaction could lead to developing mechanisms that support and encourage it, or, if it is deemed to have a negative effect, mechanisms that suppress or warn about it. In many of these cases, patterns need not be exact so long as working with them improves the system: perhaps the workflow for committing was not *optimal*, or exactly what was in the developer's minds, but it can still be *useful*, so long as it can be understood and it aids coordination or one of the other benefits listed above.

Secondly, we are not starting with a blank slate—we start with a system which already *does something*, and we look for incremental improvements. This makes the inference task much easier—people already have the tools to create the behaviour they desire, so they do not have to explain it to the system; rather, it is up to the system to attempt to improve their experience, whether it is streamlining their development experience, or incentivising them to continue working for Zooniverse.

Next, since we are working with live systems, changes happen in realtime and their effects can be monitored; just as Google's rigorous A/B testing drives their interface design, we can analyse our emerging protocols and ask if their presence is helpful or detrimental? Are there similar protocols elsewhere which we could draw on? Does this modification help or hinder? Does adding in gamification draw people in or drive them away? And which people? There is also no requirement that any analysis and extraction needs to be entirely automagical—these are fundamentally social machines, so it makes perfect sense to involve the user community in what happens: they can vote on alternative proposals, suggest modifications or co-create their own protocols with machine support.

Finally, the nature of the communities makes a difference. In our examples here, we have two interconnected social machines. The human population of one machine is constituted by developers, who are technically skilled and motivated to spend time working with the infrastructure. They are hence in a position to understand its workings and be able to aid in the artifact modification process. The other machine has a more general population, but typically much larger. This gives a far bigger pool of behaviour from which to draw potential interaction patterns, and a lot of test subject for assessing their effectiveness.

6 Conclusion

In this chapter, we have set out a viewpoint, a vision for a particular way of conceptualising the creation of software artifacts, where coupled social machines provide points of coordination for diverse communities of developers and participants, and use computational intelligence to refine natural human behaviour into repeatable, shareable, automatable, analysable protocols. This model will not apply to all software development; it is an attempt to capture some of the trends which are setting the direction of future software development. In particular, we argue that when creating large scale social machines, where diverse communities of users are electronically connected and supported to form social machines, it is necessary to have the kind of computational support that we outline here.

Acknowledgments This work is supported under SOCIAM: The Theory and Practice of Social Machines, a programme funded by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant number EP/J017728/1, and a collaboration between the Universities of Edinburgh, Oxford, and Southampton.

References

1. http://blog.galaxyzoo.org/2013/05/28/updates-to-talk/
2. Alexander, C.: The Oregon Experiment. Oxford University Press, Oxford (1975)
3. Berners-Lee, T., Fischetti, M., By-Dertouzos, M.F.: Weaving the Web: the original design and ultimate destiny of the World Wide Web by its inventor. http://dl.acm.org/citation.cfm?id=556560 (2000)
4. Damm, W., Harel, D.: LSCs: breathing life into message sequence charts. Fom. Method. Syst. Des. **19**(1), 45–80 (2001)
5. D'Inverno, M., Luck, M., Noriega, P., Rodriguez-Aguilar, J.A., Sierra, C.: Communicating open systems. Artif. Intell. **186** 38–94 (2012). doi:10.1016/j.artint.2012.03.004. http://linkinghub.elsevier.com/retrieve/pii/S0004370212000252
6. Goldman, M., Little, G., Miller, R.: Collabode: collaborative coding in the browser. In: CHASE11, Waikiki, Honolulu, pp. 65–68 (2011)
7. Halfaker, A., Kittur, A., Riedl, J.: Don't bite the newbies (2011). doi:10.1145/2038558.2038585. http://dl.acm.org/citation.cfm?id=2038558.2038585
8. Harel, D.: Can programming be liberated, period? Computer **41**(1), 28–37 (2008). doi:10.1109/MC.2008.10. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4445599. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4445599
9. Hendler, J., Berners-Lee, T.: From the semantic web to social machines: a research challenge for AI on the World Wide Web. Artif. Intell. (2010). http://www.sciencedirect.com/science/article/pii/S0004370209001404
10. Ko, A.J., Myers, B., Rosson, M.B., Rothermel, G., Shaw, M., Wiedenbeck, S., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H.: The state of the art in end-user software engineering. ACM Comput. Sur. **43**(3), 1–44 (2011). doi:10.1145/1922649.1922658. http://portal.acm.org/citation.cfm?doid=1922649.1922658
11. Lepri, B., Salah, A., Pianesi, F., Pentland, A.: Human behavior understanding for inducing behavioral change: social and theoretical aspects. Communications in computer and information science, Constr. Ambient Intell. 252–263. (2012). http://link.springer.com/chapter/10.1007/978-3-642-31479-7_44

12. Malone, T.W., Laubacher, R., Dellarocas, C.: Harnessing Crowds: Mapping the Genome of Collective Intelligence, MIT Sloan (2009)
13. Malone, T.W., Laubacher, R., Dellarocas, C.: The Collective Intelligence Genome. In: MIT Sloan Manage. Rev. **51**(3), 21–31 (2010)
14. Martin, U., Pease, A.: Mathematical practice, crowdsourcing, and social machines. In: Carette, J., Aspinall, D., Sojka, P., Lange, C., Windsteiger, W. (eds.) Intelligent Computer Mathematics: MKM, Calculemus, DML, and Systems and Projects 2013, pp. 98–119. Springer (2013). http://arxiv.org/abs/1305.0900
15. McGinnis, J., Robertson, D., Walton, C.: Protocol synthesis with dialogue structure theory. In: Simon Parsons, Nicolas Maudet, Pavlos Moraitis, Iyad Rahwan (eds.) Argumentation in Multi-Agent Systems, pp. 199–216. Springer (2006). http://link.springer.com/chapter/10.1007/11794578_13
16. Miller, T., McGinnis, J.: Amongst first-class protocols. In: Alexander Artikis, Gregory M. P. O'Hare, Kostas Stathis, George Vouros (eds.) Engineering Societies in the Agents World VIII, pp. 208–223 (2008). http://link.springer.com/chapter/10.1007/978-3-540-87654-0_11
17. Murray-Rust, D., Robertson, D.: LSCitter: building social machines by augmenting existing social networks with interaction models. In: SOCM at WWW, Seoul (2014)
18. Myhill, C.: Commercial success by looking for desire lines. Computer Human Interaction, pp. 293–304. Springer, Berlin (2004)
19. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Auton. Agents Multi-Agent Syst. (2008). http://link.springer.com/article/10.1007/s10458-008-9053-x
20. Pentland, A.: The new science of building great teams. Harvard Bus. Rev. **90**(4), 60–69 (2012). http://www.citeulike.org/group/15592/article/10606943
21. Raddick, M.J., Bracey, G., Gay, P.L., Lintott, C.J., Cardamone, C., Murray, P., Schawinski, K., Szalay, A.S., Vandenberg, J.: Galaxy Zoo: Motivations of Citizen Scientists. Astron. Educ. Rev. **12**(1) 010–106. American Astronomical Society (2013)
22. Reed, J., Raddick, M.J., Lardner, A., Carney, K.: An exploratory factor analysis of motivations for participating in zooniverse, a collection of virtual citizen science projects. In: 2013 46th Hawaii International Conference on System Sciences, pp. 610–619. IEEE (2013). doi:10.1109/HICSS.2013.85. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6479908
23. Robertson, D.: A lightweight coordination calculus for agent systems. Declarative Agent Languages and Technologies II, pp. 183–197. Springer, Berlin (2005). doi:10.1007/11493402_11
24. Robertson, D.: Lightweight coordination calculus for agent systems: retrospective and prospective. Declarative Agent Languages and Technologies 2011, LNAI 7169, pp. 84–89. Springer, Berlin (2012). http://www.springerlink.com/index/R563456340176230.pdf
25. Robertson, D., Giunchiglia, F.: Programming the social computer. Philos. Trans. R. Soc. Ser. A. Phy. Sci. Eng. **371**(1987) (2013). http://rsta.royalsocietypublishing.org/content/371/1987/20120379.short
26. Schall, D., Satzger, B., Psaier, H.: Crowdsourcing tasks to social networks in BPEL4People. World Wide Web (2012). doi:10.1007/s11280-012-0180-6. http://link.springer.com/10.1007/s11280-012-0180-6
27. Shadbolt, N., Smith, D., Simperl, E., Van Kleek, M., Yang, Y., Hall, W.: Towards a classification framework for social machines. In: SOCM2013: The Theory and Practice of Social Machines. Rio de Janeiro, Brazil (2013)
28. Siebes, R., Dupplaw, D., Kotoulas, S., Pinninck, A.P.D., Harmelen, F.V., Robertson, D.: The openknowledge system : an interaction-centered approach to knowledge sharing. On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, pp. 381–390. Springer, Berlin (2007)
29. Tang, D., Agarwal, A., O'Brien, D., Meyer, M.: Overlapping experiment infrastructure: more, better, faster experimentation. In: Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 17–26 (2010). http://dl.acm.org/citation.cfm?id=1835810

Multi-Agent System Approach for Modeling and Supporting Software Crowdsourcing

Xinjun Mao, Fu Hou and Wei Wu

Abstract The advent and successful practices of software crowdsourcing needs to investigate its in-depth essence and seek effective technologies to support its activities and satisfy increasing requirements. We highlight crowdsourcing participants consist of a multi-agent system and software crowdsourcing is a multi-agent problem-solving process. This paper discusses the characteristics and challenges of software crowdsourcing in contrast to traditional software development, and present a general analysis framework based on multi-agent system to examine the organization and behaviours of software crowdsourcing. Several software crowdsourcing models performed on typical platforms like *Topcode*, *uTest* are established and their organization and coordination are discussed. We have developed a service-based multi-agent system platform called *AutoService* that provides some fundamental capabilities like autonomy, monitoring, flexible interaction and organization, and can serve as an infrastructure to support software crowdsourcing models and tackle its challenges. A software crowdsourcing prototype is developed and some scenarios are exemplified to illustrate our approach.

1 Introduction

In the past almost 50 years software engineering has obtained enormous and continuous advances to solve software crisis, which can be divided into multiple phases [1]. In each phase, the progresses of software engineering result from some

X. Mao (✉)
Department of Computer Science and Technology, College of Computer Science, National University of Defense Technology, Changsha, Hunan, China
e-mail: mao.xinjun@gmail.com; xjmao@nudt.edu.cn

F. Hou · W. Wu
College of Computer Science, National University of Defense Technology, Changsha, Hunan, China
e-mail: fu.houmail@gmail.com

W. Wu
e-mail: wuei08@gmail.com

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_5

major sources of changes and concerns. We can also witness that a large number of software systems have been successfully developed. However, software crisis still exists with software taking longer and costing more to develop, and not working very well when eventually delivered. The speed of software development can not keep up with the increase of software systems that are expected by increasing potential stakeholders. Moreover, the advances in hardware device and Internet technology along with the vast amounts of potentially available data make software crisis much worse [2].

In the literature of software engineering, researchers and practitioners in both academic and industry take great efforts to seek effective approaches to tackling software crisis. Various software development models, methodologies, languages and platforms have been proposed like OOSE, agile methodology, service-oriented approach, etc. Some of them are mutually contradictory such as heavyweight methodologies (like waterfall development model) and lightweight methodologies (like agile methodology), which represents some changes of the thinking on the essences of software development issues and solutions.

Traditionally software projects are developed under specific organizations (e.g., IT Company). They are organized as structured teams, each member of which is assigned definite roles and development tasks. Developers are stable and pre-arranged, and they normally cooperate with each other by direct interactions like weekly meetings or face-to-face discussions to complete software development activities.

Recently, software crowdsourcing is emerging as a new software development method and has attracted great attention from both practitioners in industry and researchers in academic [3, 9, 10]. In such method, requestors of software development can publish its development requirements in the Internet. Individuals that intend to devote to the software development can submit their proposals. Requestors then decide who and which proposals are accepted according to their provided solution, their experiences and past achievements, etc. The individuals that participate in the software development are called as crowdsourcers. They can collaborate with each other based on web 2.0 tools like Blog, Twitter, etc., and submit the resulting artefacts like program, documents, or test cases. Requestors evaluate crowdsourcers artifacts and pay their rewards. It is regarded as a novel business models that have been successfully applied in Apple Store [4].

In contrast with traditional development method, software crowdsourcing attempts to utilize massive talents in the Internet and therefore decrease development costs, seek potentially considerable spectrum of returned solutions and adopt competition within the crowd to improve software quality [5, 19]. It is based on the Internet and the emerging Web 2.0 technologies to facilitate the connectivity and collaboration of networked people crowdsourcers distributed in different sites. The power of software crowdsourcing is that it can aggregate and utilize thousands of human beings and their talents (e.g., application knowledge, program expertise) into software development process, and therefore can solve the software development problems that traditional software engineering is otherwise difficult or unsolvable (e.g., to find various developers and complete software development in a relative short period) [9].

Although it is still in an early stage, software crowdsourcing have gain successes in several domains (e.g., mobile applications) and many companies realized its potential business value and launched campaigns [12]. Apple Company established App Store in 2008 that attracted millions of crowdsourcers. More than 775,000 apps have been made available by crowdsourcing approach [4]. Several software crowdsourcing platforms like *Topcoder*, *uTest*, etc., have been developed to aggregate millions of skilled software engineers (e.g., programmers in *C++* or *Java*, or software testers) to support various crowdsourcing purposes (e.g., applications, algorithms, performances and datasets [8]) and corresponding development activities [6, 7, 11, 13, 14, 18]. Moreover, various software crowdsourcing methods have been proposed and practiced to satisfy different requirements and problem-solving ways [15–17].

Against the background, it is necessary to investigate the in-depth essence and challenges of software crowdsourcing and seek effective technologies to support its activities and satisfy increasing requirements. The remaining sections are organized as follows. Section 2 discusses the characteristics and potential challenges of software crowdsourcing in contrast to traditional software development. Section 3 present a general analysis framework based on multi-agent system to examine the organization and behaviours of software crowdsourcing, with which several software crowdsourcing models have been presented. Section 4 introduces a service-based multi-agent system platform called *AutoService* that provides some fundamental capabilities to serve as infrastructure to support software crowdsourcing models. A crowdsourcing prototype is developed and some scenarios are exemplified to illustrate our approach. Last, some conclusions and future researches are discussed in Sect. 5.

2 Characteristics and Challenges of Software Crowdsourcing

In contrast with traditional software development, software crowdsourcing has the following characteristics (see Table 1). First, the developers in software crowdsourcing are individuals in the Internet. Normally, they have expertise in specific domain (e.g., mobile application) and skills to perform software development activities, e.g., finding bugs in the software system, constructing a program satisfying the requirements. Different from the developers in traditional software development, in which the roles and tasks of developers are pre-defined and their development behaviors are constrained by the organization norms and project constraints, crowdsourcers in software crowdsourcing are more autonomous in their behaviours. Second, the development organizations in software crowdsourcing are formed dynamically in term of the collaboration among requestors, organizers and developers. They actually are virtual organizations consisting of various participants in the Internet and evolving during the software crowdsourcing process (see Fig. 1). Third, in traditional software development, the interactions between developers are tightly related with the software development details (e.g., requirement, design, testing) and are normally performed

Table 1 Comparison between software crowdsourcing and traditional software development

	Traditional Software Development	Software Crowdsourcing
Developer	Coming from development organization	Coming from Individuals in the Internet
Development Organization	Real organization well-structured beforehand, Relatively stable	Virtual organization formed dynamically, Evolving
Interaction	Face-to-face	By Internet and Web 2.0
Collaboration	Cooperation	Competition, cooperation, negotiation, and hybrid

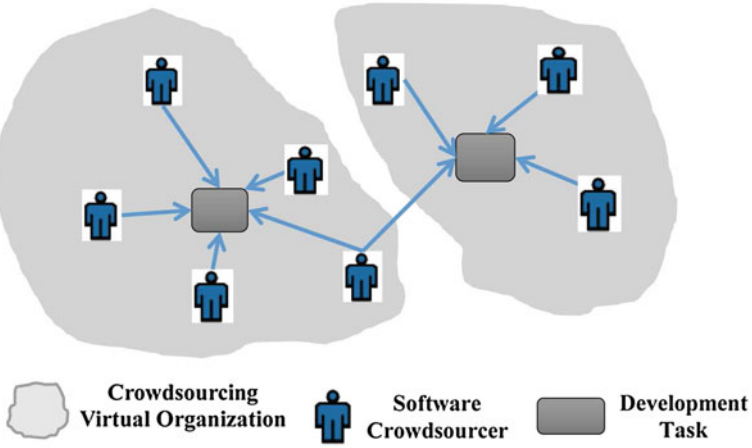


Fig. 1 Illustration of Software Crowdsourcings

by face-to-face meetings or discussions. However, in software crowdsourcing, interactions between participants in software crowdsourcing are performed either to form virtual development organizations or to exchange development details. For example, requestors publish their requirements and the individuals submit their proposal. These interactions are performed in term of the Internet and depending on Web 2.0 technologies. Therefore, specific platforms are necessary to support the interactions among the participants. Lastly, the developers in traditional software development form a team and cooperate with each other to complete the development tasks. However, the collaborations between participants in software crowdsourcing are various, including competition, cooperation and negotiation. For example, individuals compete with each other for some software crowdsourcing task, requestors negotiate with crowdsourcers about their rewards.

There are numerous skilled software engineering talents distributed in the Internet. The purpose of software crowdsourcing is to aggregate and utilize these talents for

the software development. However, the diversity, autonomy, dynamic of individual challenges the software crowdsourcing in several aspects.

- **Find expected objects** Each individual in the Internet has their competences and skills in some aspects like domain knowledge, architecture design, programming in specific language or software testing. The requirements that requestors publish for crowdsourcing need various talents. Therefore, on the one hand it is important for requestor to find the qualified and competent individuals to participate the software crowdsourcing. One popular way at present is to publish the requirements in some crowdsourcing platform like *TopCoder* or *uTest*. Such approach is a passive way to find the crowdsourcers. On the other hand, individuals in Internet need to find the requirements that they are interested in and are qualified. At present individuals find the appropriate requirements by browsing the software crowdsourcing websites. Such way can not provide time-fashion services for crowdsourcers to get the requirements information.
- **Support effective crowdsourcing activities and collaboration** For complex development requirements, software crowdsourcing needs multiple participants that have various skills and play different roles. For example, *Topcoder* supports requestors to publish requirements, Copilots to organize software development and platform, and various crowdsourcer involving in the software development. In addition to traditional development activities, these participants should perform a number of crowdsourcing activities like publishing, proposing, evaluation and rewarding, etc. There are various collaboration among the participants, which depending the software development requirements, the problem-solving way, the roles that they play. Therefore, effective mechanisms and approaches for crowdsourcing activities and collaboration should be provided to organize the participants in software crowdsourcing and satisfy the development and collaborations requirements.
- **Tackle autonomy and dynamic issues** One important characteristics of software crowdsourcing is the dynamic. The requirements for crowdsourcing, individuals involving in crowdsourcing, the artifacts and their quality, and the collaborations among crowdsourcers dynamically evolves. Moreover, the individuals involving in software crowdsourcing are autonomous in their behaviors, for example joining, leaving software crowdsourcing and making their efforts on crowdsourcing. The dynamic results in high risks of software crowdsourcing and need to effectively manage the dynamic and autonomy. For example, it is necessary to perceive and adapt to the dynamic.

3 Multi-Agent System Models of Software Crowdsourcing

In essence, software crowdsourcing is a distributed problem-solving, in which the participating entities are autonomous individuals distributed over the Internet, and the problem-solving is the process to complete the software development tasks

by performing various software development activities and their collaborations. Therefore, crowdsourcing participants consist of a multi-agent system. This section presents a multi-agent system analysis framework for software crowdsourcing, with which a number of software crowdsourcing models are established to examine their common properties.

3.1 Multi-Agent System Analysis Framework for Software Crowdsourcing

The individuals involving into the software crowdsourcing can be abstracted as autonomous agents performing various development and collaboration behaviors. Typically, there are multiple agents playing various roles and completing different tasks in the process of software crowdsourcing. For example, the individual who has development requirements (e.g., designing algorithms, finding bugs in the program, or programming) to be performed is known as "*requestor*", persons that are responsible for organizing and managing software crowdsourcing are called as "*organizer*" or "*Copilots*" in *TopCoder*, these who want to sign up to perform work are described as "*worker*" or "*crowdsourcer*". The software development requirements may have an associated payment and a completion time. Workers can browse or query requirements along with the payment and time information before deciding whether to work on them.

Software crowdsourcing process is actually a distributed problem-solving of multi-agent system. Here, the problem corresponds to software development requirements, problem-solving means completing software development tasks satisfying the requirements, e.g., constructing a programming or finding bugs in the program. Typically, the process of software crowdsourcing can be described as follows. Requestor identifies and publishes software development requirements together with the corresponding incentive policy like payment and constraints like competition rule and completing time. Organizers are often necessary when the requirements are complex. Some crowdsourcing platforms like *Topcoder* provide "*Copilots*" to decompose the development tasks into a number of modular that are easy to be tackled. These development tasks and requirements are then to be released online in some platforms like *Topcoder* or *uTest* to a crowd of outsiders. The individuals that intend to participate in the software development can undertake the tasks in a sole or collaborative way (e.g., based on Web 2.0 tools in the Internet). Upon completion, the individuals should submit their software artifacts (e.g., documents that describe the designed algorithms, program codes that implement the requirement, testing cases that can find bugs) to the software crowdsourcing platform. The requestors or the organizers that are on behalf of requestors will then assesses the quality of the submitted software artifacts and provide rewards.

There are several platforms supporting the above process of software crowdsourcing like *TopCoder*, *GetCoder*, *uTest*, etc. However, as the diversity of software

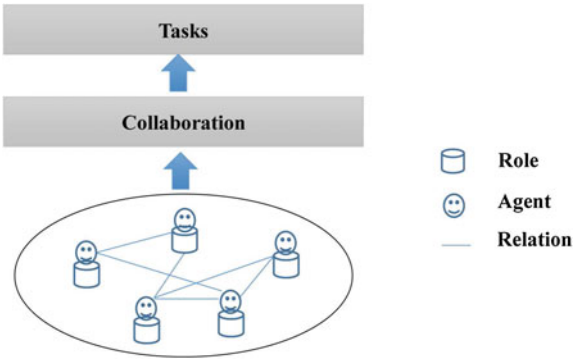


Fig. 2 Multi-agent analysis framework of software crowdsourcing

development requirements and corresponding tasks, the involving individuals, their roles and behaviours, problem-solving ways are various and different. In order to define various software crowdsourcing models, clarify their differences and further investigate the expected technologies, it is necessary to create an analysis framework for software crowdsourcing.

Our proposed framework is based on multi-agent system approach to modeling the individuals in the software crowdsourcing and their organization and behaviors (see Fig. 2). Formally, the analysis framework is defined as 5-tuple = *<Tasks, Agents, Roles, Collaboration, Constraints>*, where

- *Tasks*: the set of software development tasks that should be completed in the software crowdsourcing. Some software crowdsourcing has single task, e.g., finding bugs in the program, designing algorithm satisfying the requirements. Others may have multiple and interrelated tasks, e.g., defining the requirements models, providing architecture designs, constructing the program modular, etc.
- *Agents*: the set of agents playing various roles to participate in the software crowdsourcing. Each agent has its responsibilities depending on the roles that it plays. Typically, a software crowdsourcing consists of at least one requestor agent and a number of crowdsourcing agents, i.e. crowdsourcers.
- *Roles*: the set of roles involving in the software crowdsourcing. Some software crowdsourcing only involves with requestor and crowdsourcer to release requirements and perform development tasks. Others with complex development tasks may need organizer to manage requirements and crowdsourcing process, and third-party to assess the quality of the submitted software artifacts.
- *Collaboration*: the structured interactions sequence that defines how agents in software crowdsourcing collaborate with each other to complete the development tasks. Some protocol of software crowdsourcing is simple only requiring limited interactions between requestors and crowdsourcers. Others may consist of a number of interaction sequences to deal with the issues in the software crowdsourcing process.

- *Constraints*: the conditions that should be satisfied and the resources that the software crowdsourcing provides. For example, software crowdsourcing typically has time restriction and normally provides incentives (like rewards) to encourage crowdsourcing.

3.2 Multi-Agent System Models of Software Crowdsourcing

There are kinds of software crowdsourcing provided by existing software crowdsourcing platforms to satisfy various demands on software crowdsourcing. To model these software crowdsourcing manners is helpful to examine their fundamental elements and characteristics, and find which model is suitable for satisfying specific crowdsourcing requirements. Moreover, it can also help us to identify the differences of various software crowdsourcing models, and seek effective technologies to support these software crowdsourcing models.

From the viewpoint of requestors, the crowdsourcing can be performed into two ways: direct and indirect. In the direct way, requestor directly to interact with crowdsourcers to obtain individual talents and complete development tasks, e.g. obtaining and assessing the software artifacts. Generally, this kind of crowdsourcing is suitable for simple development tasks. In the indirect way, requestors interact with some proxy like organizers or Copilots who are responsible for managing development tasks and collaborating with crowdsourcers. This kind of crowdsourcing is more suitable for complex development tasks that are difficult for requestors to manage. From the viewpoint of crowdsourcers, they can participate in the crowdsourcing either in a competition or independence way. In the competition way, crowdsourcers compete with each other to satisfy the development requirements and win the rewards. Typically, if the requestor only needs limited solutions for their problems, such manner can be used. For example, requestors seek the best algorithms to satisfy the requirements, then only parts of crowdsourcers can win. In the independence way, there is no any relationship between crowdsourcers. They perform their development activities in an independent way and anyone who can satisfy the development requirements can win (see Table 2). In the following we will describe these models based on the analysis framework proposed in Sect. 3.1.

- **Direct and competing model of software crowdsourcing**
  Let DC = $<Tasks_{dc}, Agents_{dc}, Roles_{dc}, Collaboration_{dc}, Constraints_{dc}>$ represents direct and competing model of software crowdsourcing, where
  - $Tasks_{dc}$: the development tasks of crowdsourcing are simple and easy to be managed by requestors, e.g., to construct a program or design an algorithm to satisfy the requirements. Moreover, the tasks are required to be completed in a competing way.
  - $Agents_{dc}$: The agents in the models either play requestor role or crowdsourcer role. There is competing relationship among agents playing crowdsourcer role.
  - $Roles_{dc}$: there are only requestor and crowdsourcer roles in the model.

Table 2 Models of software crowdsourcing

	Competing	Independent
Direct	• Requestors directly interact with crowdsourcers to complete crowdsourcing, e.g., releasing development tasks, obtaining and assessing software artifacts	• Requestors directly interact with crowdsourcers to complete crowdsourcing, e.g., releasing development tasks, obtaining and assessing the software artifacts
	• Crowdsourcers compete with each other to complete software development tasks and win the rewards	• Crowdsourcers are independent with each other to participate in the crowdsourcing
Indirect	• Organizer or Copilots act as the proxy of requestors and are responsible for managing development tasks for requestors	• Organizer or Copilots act as the proxy of requestors and are responsible for managing development tasks for requestors
	• Requestors indirectly interact with crowdsourcers to complete crowdsourcing	• Requestors indirectly interact with crowdsourcers to complete crowdsourcing
	• Crowdsourcers compete with each other to complete software development task and win the rewards	• Crowdsourcers are independent with each other to participate in the crowdsourcing

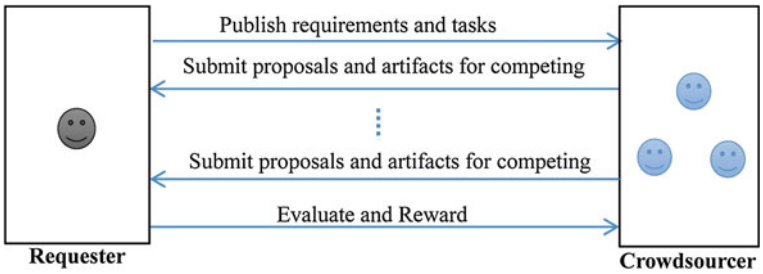


Fig. 3 Collaboration in direct and competing model of software crowdsourcing

– $Collaboration_{dc}$: the collaboration occurs between requestor agents and crowdsourcer agents. Typically, it is performed in a way described in Fig. 3.
– $Constraints_{dc}$: there are incentive policies for crowdsourcers to participate in crowdsourcing. The crowdsourcers are required to compete with each other to win the rewards.

- **Indirect and competing model of software crowdsourcing**
  Let IC = $<Tasks_{ic}, Agents_{ic}, Roles_{ic}, Collaboration_{ic}, Constraints_{ic}>$ represents indirect and competing model of software crowdsourcing, where

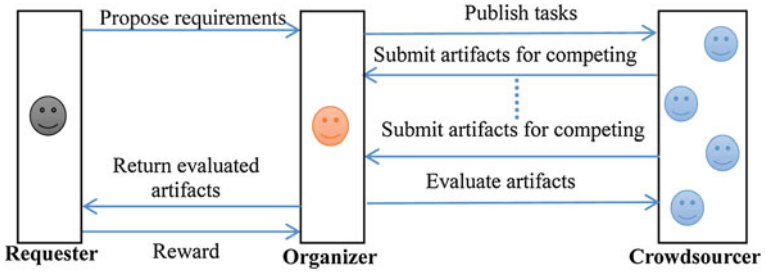


Fig. 4 Collaboration in indirect and competing model of software crowdsourcing

- $Tasks_{ic}$: the development tasks of software crowdsourcing are relatively complex and difficult to be managed by requestors. For example, the tasks should be decomposed into small ones for crowdsourcing, e.g., to analyze, design and construct a software system satisfying the requirements. Moreover, the tasks are required to be completed in a competing way.
- $Agents_{ic}$: There are several kinds of agents in the models, e.g., requestor, crowdsourcer, organizer, etc.
- $Roles_{ic}$: In addition to the requestor and crowdsourcer roles, organizer and other roles are introduced in the model to manage development tasks, assess the quality of the submitted proposals and software artifacts.
- $Collaboration_{ic}$: the collaboration occurs between requestors, organizers and crowdsourcers, etc. Typically, organizer is the proxy of requestors and the collaboration among them is performed in a way described in Fig. 4.
- $Constraints_{ic}$: there are incentive policies for crowdsourcers to participate in crowdsourcing. The crowdsourcers are required to compete with each other to win the rewards.

- **Direct and independent model of software crowdsourcing**
  Let DN = $<Tasks_{dn}, Agents_{dn}, Roles_{dn}, Collaboration_{dn}, Constraints_{dn}>$ represents direct and independent model of software crowdsourcing, where
  - $Tasks_{dn}$: the development tasks of crowdsourcing are simple and easy to be managed by requestors, e.g., to test and find bugs in a program. Moreover, the tasks need multiple and various results that are not conflicting with each other and can be completed by crowdsourcers in an independent and non-competing way.
  - $Agents_{dn}$: The agents in the models either play requestor role or play crowdsourcer role.
  - $Roles_{dn}$: there are only requestor and crowdsourcer roles in the model.
  - $Collaboration_{dn}$: the collaboration occurs between requestors and crowdsourcers. Typically, it is performed in a way described in Fig. 5.
  - $Constraints_{dn}$: there are incentive policies for crowdsourcers to participate in software crowdsourcing.

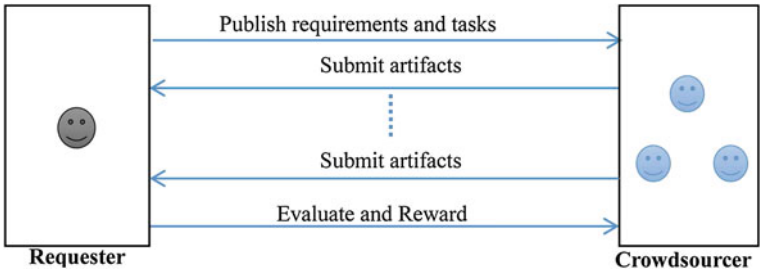


Fig. 5 Collaboration in direct and independent model of software crowdsourcing

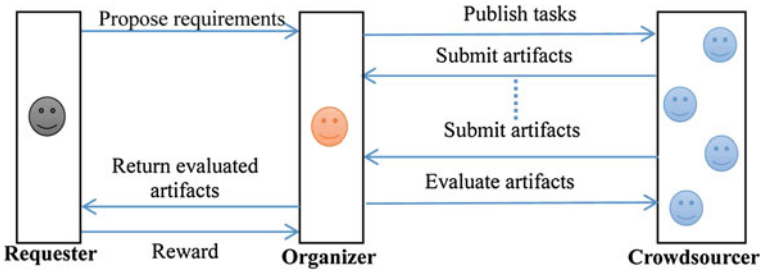


Fig. 6 Collaboration in indirect and independent model of software crowdsourcing

- **Indirect and independent model of software crowdsourcing**
  Let IN = $<Tasks_{in}, Agents_{in}, Roles_{in}, Collaboration_{in}, Constraints_{in}>$ represents indirect and independent model of software crowdsourcing, where
  - $Tasks_{in}$: the development tasks of software crowdsourcing are relatively complex and difficult to be managed by requestors. For example, the tasks should be decomposed into small ones for crowdsourcing. Moreover, the tasks need multiple and various results that are not conflicting and can be completed by crowdsourcers in an independent and non-competing way.
  - $Agents_{in}$: There are several kinds of agents in the models, e.g., requestor, crowdsourcer, organizer, etc.
  - $Roles_{in}$: In addition to the requestor and crowdsourcer roles, organizer and other roles are introduced in the model to manage development tasks, assess the quality of the submitted proposals and software artifacts.
  - $Collaboration_{in}$: the collaboration occurs between requestors, organizers and crowdsourcers, etc. Typically, organizer is the proxy of requestors and the collaboration among them is performed in a way described in Fig. 6.
  - $Constraints_{in}$: there are incentive policies for crowdsourcers to participate in crowdsourcing.

Existing software crowdsourcing platforms like *Topcoder* support the direct and competing model of software crowdsourcing, indirect and competing model of software crowdsourcing. Platforms like *uTest*, *Bugfinders* support the direct and independent model of software crowdsourcing.

4 MAS Approach for Supporting Software Crowdsourcing

Doubtlessly software crowdsourcing involves with multiple agents playing various roles and interacting with each other to complete software development tasks. It can be modeled as multi-agent problem-solving and can borrow multi-agent system technologies to support software crowdsourcing.

4.1 Software Crowdsourcing based on Multi-Agent Technology

Multi-agent system provides a number of technologies that can be used in supporting software crowdsourcing such as software architecture for autonomous behaviors, communication languages, collaboration models, etc. Each individual in software crowdsourcing such as requestor, crowdsourcer, organizer, etc., can be provided with software agent that acts as personal assistant and supports their crowdsourcing activities like publishing requirements, evaluating the artifacts, submitting software artifacts or proposals, etc. All of software agents constitute a multi-agent virtual organization in which they collaborate with each other to complete software crowdsourcing (see Fig. 7).

- **Software agents as personal assistant** Software agents for individuals in software crowdsourcing are responsible for providing fundamental services of software crowdsourcing (e.g., publish software crowdsourcing task, submit proposal,

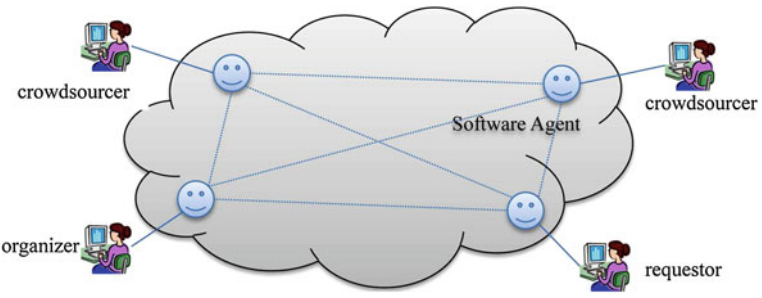


Fig. 7 A view of software crowdsourcing based on multi-agent system technologies

evaluate the quality, pay the rewards), managing various resources for individuals such as talents and skills, requirements and tasks, acquaintance, history of software crowdsourcing, etc. They also provide autonomous and pro-active behaviours that help individuals to participate in the software crowdsourcing when concerned events occur. For example, they can classify the published tasks in the software crowdsourcing platform and rank them according to the previous engagements, interests and skills.

- **Support various and flexible collaboration** Various collaborations are needed in the process of software crowdsourcing, which depends on the crowdsourcing tasks and organization. For example, the crowdsourcer can negotiate with the requestors about the rewards and payments, multiple crowdsourcers can cooperate with each other and form a temporary team to participate in the software crowdsourcing by self-organization. These collaboration models can be encapsulated into the software agents and their organization.
- **Manage crowdsourcing organization** In the process of software crowdsourcing, individuals constitute a multi-agent organization. As the autonomy of crowdsourcers and the dynamic of software crowdsourcing, the organization should be effectively managed in order to satisfy the development requirements. Various basic functionalities and services should be provided to form, adjust, evolve and dismiss software crowdsourcing organization. For example, when the requirements are published, a multi-agent organization for the crowdsourcing tasks should be established. Such organization should be maintained in the whole process of software crowdsourcing, e.g., joining a crowdsourcer, dismiss a crowdsourcer.
- **Response to Changes and Events** Typically software crowdsourcing are performed in the Internet, in which various events related with software crowdsourcing may occur. For example, a new crowdsourcing task is published and a crowdsourcer with specific skill like programming with JavaBeans is registered. Software agent can provide sensor to perceive various events and changes in order to know what happens and response in a time-fashion way. They can also provide self-adaptation capabilities to adapt to the changes and support software crowdsourcing in a better way, e.g., acquiring required talents of skills, experiences, reputations and domain knowledge that crowdsourcing task requires.

4.2 AutoService Supporting Platform and Case Study

We have developed a service-based multi-agent system platform called *AutoService* that provides some fundamental capabilities like autonomy, monitoring, flexible interaction and organization, and can serve as infrastructure to support software crowdsourcing models and tackle its challenges.

In *AutoService*, services deployed in Internet are managed by software agents, which means that any accesses to services firstly should be processed by the agents, whether and how to provide a service depends on the autonomous decisions of software agents that manage the services. Applications or software agent can access

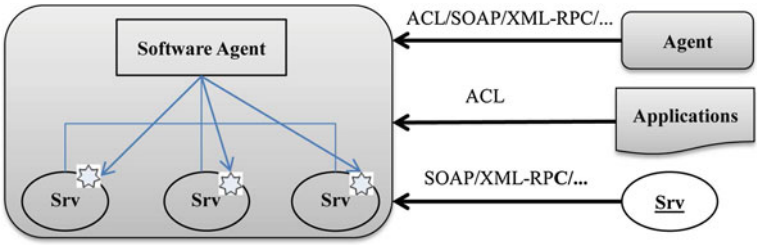


Fig. 8 Model of Autonomous Service and their access manners

services by SOA protocol or agent communication language (see Fig. 8). A software agent can encapsulate one or more services and each service is managed by at least one software agent. Therefore, each individual like crowdsourcer, requestor or organizer, etc., can be equipped with a software agent that provides a number of elementary services to support their crowdsourcing activities. The autonomous behaviors based on the crowdsourcing services enable individual to participate in software crowdsourcing in a better way. For example, crowdsourcing can actively join a crowdsourcing organization when perceiving some new tasks to be published.

The services and the agent have their status that can be monitored in *AutoService* and the monitored information is effectively managed and can be flexibly accessed by software agent or services that intend to obtain. Autonomous service registry in *AutoService* supports the registration of services, agents and their statuses. Moreover, in addition to query the registration and status information it also provides subscribe/publish functions to actively obtain concerned information in a time-fashion way. Therefore, the software agents for crowdsourcer, requestor or organizer, etc., can be monitored as for its availability, online, the involved crowdsourcing tasks, etc. They can also know about the new published crowdsourcing tasks, the status of involving crowdsourcer, the progress of the development tasks, etc. These functionalities enable individuals in software crowdsourcing to participate in software crowdsourcing in a better way and effectively deal with dynamic based on the *AutoService* infrastructure, i.e., knowing what happens.

The agent and service in *AutoService* can interact with each other in term of two ways. One is the low-level SOA protocols, with which services in the platform can be accessed. For example, requestor in the software crowdsourcing can access the services published by crowdsourcer agent to get its ongoing tasks. The other is the high-level interactions and collaboration protocols specified in agent communication language. Such interactions and encapsulated protocols support various collaboration models (e.g., ones defined in Sect. 3.2).

Based on the development and running support of autonomous services, we have developed a software prototype to simulate software crowdsourcing and validate our proposed approach (see Fig. 9). Just described as above, several kinds of agents that play various roles like crowdsourcer, requestor, organizer, etc., and encapsulate services related with the roles have been implemented and deployed in the

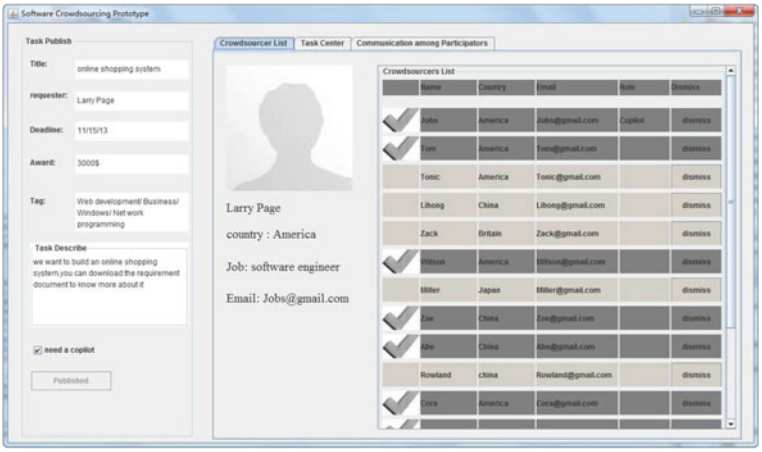


Fig. 9 Model of Autonomous Service and their access manners

AutoService platform. These agents together with the systems agents like registration center, task center, etc., can collaborate with each other to publish/subscribe tasks, bid and negotiation, call-for-proposals, etc. Requestor agent can publish crowdsourcing tasks to the individuals that have the competences or skills, therefore improve the effectiveness. The crowdsourcer can obtain the newly published tasks that are suitable for their skills. They also can query the crowdsourcers that have registered in the systems, find the crowdsourcers with same skills in order to know the adversary, obtain their status.

Comparing with existing software crowdsourcing platforms, e.g. *Topcoder* and *uTest*, our system has the following characteristics. First, if a task is released, our platform can send message to the crowdsourcers who have the relevant expertises, which will make the crowdsourcers more possible to complete in the task. Second, our platform can monitor the process of the implementation of the published tasks. For example, we can know how much the crowdsourcers have done, and whether the crowdsourcer still works on the task. Last, the crowdsourcers who join in the task can contact with each other. We can see the communication information among the crowdsourcers by our platform.

5 Conclusion

The progresses of Internet technologies and social computing (e.g. Web 2.0) provide novel approaches to tackling the issues of software development. Software crowdsourcing is an emerging area of software engineering that allows massive autonomous crowds across the world to participate in the software development in term of supporting platforms in Internet. It represents a new method to organize and

manage software project development, in which developers come from crowds rather than specialized software companies, interactions among developers occur in the Internet virtual space rather than traditional meetings or face-to-face discussions, crowdsourcers typically compete with each other to complete software development tasks and win the rewards. In the past years, software crowdsourcing has gained great successes in industry and several platforms like *Topcoder*, *uTest* have been developed in support of software crowdsourcing.

The successful practices of software crowdsourcing inevitably need us to investigate its essence behind various software crowdsourcing platforms and find its potential challenges to software technologies. Multi-agent system that consists of multiple interacting autonomous agents to solve specific common problem can act as an abstract approach to modeling the constituents of software crowdsourcing and an enabling technology to support software crowdsourcing process. The individuals in the software crowdsourcing can be modeled as autonomous agents that play various roles and take corresponding behaviours to perform software development activities. They constitute a virtual software development organization and collaborate with each other to complete crowdsourcing tasks. We present a general analysis framework based on multi-agent system to examine the constituents of software crowdsourcing organization, their roles and behaviours, especially collaborations. From the viewpoints of crowdsourcing requestors and crowdsourcers respectively, we identify four different software crowdsourcing models, including direct and competing model of software crowdsourcing, indirect and competing model of software crowdsourcing, direct and independent model of software crowdsourcing, indirect and independent model of software crowdsourcing. Especially, we detail the characteristics of each model and their collaboration process.

The autonomy of individuals in software crowdsourcing and their complex collaborations, together with dynamic and massive talents in Internet mean that technologies should be provided to support software crowdsourcing, promote the management of crowdsourcing virtual organization, and enrich the utilization of various crowdsourcing resources. Multi-agent system provides such capabilities as autonomous and pro-active behaviours of software agents, perceive of situated environment, various collaboration among agents, which can act as an alternative technology to support software crowdsourcing. We have developed a service-based multi-agent system platform called *AutoService* that provides some fundamental capabilities like autonomy, monitoring, flexible interaction and organization, and can serve as infrastructure to support software crowdsourcing models and tackle its challenges. A crowdsourcing prototype is developed and some scenarios are exemplified to illustrate our approach.

Acknowledgments The authors gratefully acknowledge the financial support from Natural Science Foundation of China under granted number under Grant No. 61070034 and 61379051; Program for New Century Excellent Talents in University NCET-10-0898; and Open Fund SKLSDE-2012KF-0X from State Key Laboratory of Software Development Environment.

References

1. Boehm, B.: A view of 20th and 21st century software engineering. In: Proceedings of ICSE, pp. 12–29 (2006)
2. Fitzgerald, B.: Software Crisis 2.0. Software technology. IEEE Comput. **45**(4), 89–91 (2012)
3. Zhao, Y., Zhu, Q.: Evaluation on crowdsourcing research: current status and future direction. Inf. Syst. Front., 1–18 (2012)
4. Bergvall-Kreborna, B. Howcroft, D.: The Apple business model: crowdsourcing mobile applications. Accounting Forum, Elsevier (2013)
5. Schall, D., Psaier, H., Treiber, M., Skopik, F.: Engineering service-oriented crowdsourcing for enterprise environments (2010)
6. Vukovi, M.: Crowdsourcing for enterprises. In: Proceeding of Congress on Services, pp. 686–682 (2009)
7. Fried, D.: Crowdsourcing in the software development industry. Nexus of Entrepreneurship and Technology Initiative Fall (2010)
8. Yuen, M.-C., King, I., Leung, K.-S.: A survey of crowdsourcing systems. In: Proceedings of IEEE International Conference on Social Computing, pp. 766–773 (2011)
9. Wenjun, W.U., W-T. Tsai. Wei L.I.: An evaluation framework for software crowdsourcing. Front. Comput. Sci. **7**(5), 694–709 (2013)
10. Pan, Y., Blevis, E.: A survey of crowdsourcing as a means of collaboration and the implications of crowdsourcing for interaction design. In: Proceedings of International Conference on Collaboration Technologies and Systems, pp. 397–403 (2011)
11. Hetmank, L.: Components and functions of crowdsourcing systems: a systematic literature review. In: Proceedings of 11th International Conference on Wirtschafts Informatik, pp. 55–39 (2013)
12. Olson, D.-L., Rosacker, K.: Crowdsourcing and open source software participation. Service Business, pp. 1–13 (2012)
13. Geiger, D.: Crowdsourcing information systems: a systems theory perspective. In: 22nd Australasian Conference on Information Systems (2011)
14. Alonso, O.: Perspectives on infrastructure for crowdsourcing. In: Proceedings of International Workshop on Crowdsourcing for Search and Data Mining (CSDM 2011), pp. 7–10 (2011)
15. Davis, J.G.: From crowdsourcing to crowdservicing. IEEE Internet Comput. **15**(3), 92–94 (2011)
16. Iren, D., Bilgen, S.: Methodology for managing crowdsourcing in organizational projects. http://www3.informatik.uni-wuerzburg.de/events/summerschool2012/proceedings/Iren.pdf (2011)
17. Saxton, G.D., Oh, O., Kishore, R.: Rules of crowdsourcing: models, issues, and systems of control. Inf. Syst. Manag. **30**(1), 2–20 (2013)
18. Storey, M.A., Treude, C., van Deursen, A., Cheng, L.-T.: The impact of social media onsoftware engineering practices and tools. In: Proceedings of FoSER (2010), pp. 359–363 (2010)
19. Baziliana, M., Rice, A., et al.: Open source software and crowdsourcing for energy analysis. Energy Policy, **49**(5): 149–153 (2012)

Supporting Multilevel Incentive Mechanisms in Crowdsourcing Systems: An Artifact-Centric View

Ognjen Scekic, Hong-Linh Truong and Schahram Dustdar

Abstract Crowdsourcing systems of the future (e.g., Social Compute Units—SCUs, collective adaptive systems) need to support complex collaborative processes, such as software development. This presupposes deploying ad-hoc assembled teams of human and machine services that actively collaborate and communicate among each other, exchanging different artifacts and jointly processing them. Major challenges in such environments (e.g., team formation, adaptability, runtime management of data-flow and collaboration patterns) can be somewhat alleviated by delegating the responsibility and the know-how needed for these duties to the participating crowd members, while indirectly controlling and stimulating them through appropriate incentive mechanisms. Existing process-centric collaboration modeling approaches (e.g., workflows) are incapable of encoding such incentive mechanisms. Therefore, in this paper we analyze different interaction aspects that incentive mechanisms cover and formulate them as requirements for future systems to support. We then propose an artifact-centric approach for modeling incentives in rich crowdsourcing environments that meets these requirements.

1 Introduction

Many previous works on crowdsourcing seem to assume that *crowd* is an unlimited pool of adequate human workforce and typically focus on problems such as: locating the most appropriate candidates for performing the tasks, or comparing different payment schemes. However, while the assumption of the practically unlimited crowd may hold true in case of simple, independent tasks, the practice shows that the current

O. Scekic (✉) · H.-L. Truong · S. Dustdar
Distributed Systems Group, Vienna University of Technology, Vienna, Austria
e-mail: oscekic@dsg.tuwien.ac.at

H.-L. Truong
e-mail: truong@dsg.tuwien.ac.at

S. Dustdar
e-mail: dustdar@dsg.tuwien.ac.at

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_6

crowdsourcing models (both from the technical and the from business perspective) fail to attract and retain workers capable of performing complex/modular, interdependent tasks [23, 27].

One of the reasons is that in the case of (highly-)skilled individuals, the crowd is, in fact, a quite limited resource pool that an ever increasing number of crowdsourcing efforts are trying to tap into. This means that the individuals need to be motivated through diverse, elaborate incentive and rewarding strategies to join a particular crowdsourcing effort and to provide their professional services at an expected level.

The other reason is that the existing crowdsourcing platforms do not offer flexible, human-like collaboration platforms to the crowd workers. Rather, the tasks are either assigned to the human workers by the system executing the workflow, or the humans bid for tasks on micro-task platforms. In both cases, the managing system dictates the orchestration, treating the crowd workers as machine computing elements, which are requested to respect the prescribed orchestration and various functional and quality constraints without being able to influence them.[1]

This situation contradicts the very reason why humans are included into computations in the first place—to do better what computers are not good in doing—i.e., to bring in creativity, flexibility in unforeseen situations, but most importantly, ability to quickly perform complex tasks by establishing ad-hoc collaborations and adapting them when needed.

1.1 Motivation

To make humans first-class citizens, workers must be given more influence on selecting their collaboration partners, coordination patterns and communication channels. Hard constraints and worker commitment protocols should be loosened to make the systems more attractive for human workers. The price to pay for this is a degree of outcome uncertainty that must be reckoned with. We can either embrace it as an inherent property of these *socio-technical/crowdsourcing systems* (like we do in most everyday life situations) or try to blindly follow the conventional paradigms and seek to detect and/or correct those uncertainties. Embracing uncertainty, however, does not mean promoting it, but rather implies usage of different passive measures for reducing it to an acceptable level. This can be achieved through *incentive mechanisms* motivating workers to self-organize and self-correct. To this end, in this paper we investigate the necessary requirements for defining and enacting such incentive mechanisms in the novel types of crowdsourcing systems [12].

To the best of our knowledge, incentives in crowdsourcing have so far been only considered at the granularity level (scope) of a business process (Sect. 4.2). As a business process typically contains a flow of different activities on multiple artifacts, workers can exhibit different behaviors depending on which activity they perform,

[1] See [5] for an overview of task distribution and coordination models.

on which artifact and with which co-workers. This means that the existing incentive models are successfully applicable only in a limited number of cases, where business processes are simple and dominated by a single activity. This is exactly the case with today's commercial crowdsourcing platforms that incentivize business processes that typically require a single worker to process and return an artifact (e.g., describe a bug, submit a design, tag a photo, translate a text). As these processes are simple, the incentives need only to focus on the core activity, and to promote wanted behaviors, like diligence and quality.

However, performing complex tasks, such as software development, with crowdsourced, ad-hoc teams involves many activities, workers and interactions that are not predictable in advance. Developers may come and go; their performance may vary; they may be using different tools to communicate, coordinate and produce code, tests and documentation; they may be used to different development methodologies and team organizations. It is not realistic to expect that a team formed out of such diverse individuals will adhere to a prescribed execution plan. In fact, designing a work process with so many unknowns will probably result in an inefficient workflow at runtime [3]. And without a valid workflow, it is impossible to design appropriate incentive mechanisms either.

Similar problems have previously been investigated by large traditional companies trying to impose uniform work processes across geographically distributed workforce. They discovered that different internal teams would agree more easily on a common set of artifacts to use in interactions rather than on the common activity flow [16]. This resulted in the birth of *artifact-centric workflows* [10]. The principal idea is to focus on data (artifacts), rather than on processes, and to leave the actors more freedom to self-organize, while controlling them indirectly through artifacts augmented with a formal lifecycle model (see Sect. 4.1 for more information).

We believe that, if extended with incentive mechanisms, the artifact-centric approach can be successfully used to describe and guide complex crowdsourcing processes. Augmenting artifact models with incentive mechanisms creates entities that self-motivate people to process and control them. By attracting workers to work at them, the artifacts push their way through the lifecycle. In addition, if we encode incentive application rules at the artifact-granularity level, we can express much finer conditions. This opens up an array of new possibilities for motivating humans to work in crowdsourcing platforms.

1.2 Contributions and Article Structure

In this paper we propose applying artifact-centric approach to designing incentives for socio-techincal systems. We argue that this approach may be better suited than the traditional process-oriented approach, covering better the different possible aspects of human behavior and business needs in complex collabrative environments. We then identify the concrete requirements for designing such incentive systems.

The rest of the paper is structured as follows: Sect. 2 introduces some fundamental notions that are used in the rest of the paper. The rest of Sect. 2 presents a motivating example, and uses it to highlight important aspects when designing incentive mechanism models for crowdsourced, artifact-centric workflows. In Sect. 3 we further analyze these aspects and identify important requirements for novel crowdsourcing systems supporting artifact-centric incentive mechanisms. In Sect. 4 we present a short review of related work on incentives in crowdsourcing and traditional artifact-centric workflows. Section 5 presents the summary and concludes the paper.

2 Artifact-Centric Incentives

We begin by defining some important terms as used throughout this paper:

Definition 1 (*Incentive*) Any scheme employed by the system to stimulate (motivate) increased level of certain worker activities (e.g., productivity, speed, quality of work, number of participants) or to discourage certain activities (e.g., drop-out rate), before the actual execution of those activities.

Definition 2 (*Reward*) Any kind of recompense for worthy services rendered or retribution for wrongdoing exerted upon workers after the completion of the activity.

Definition 3 (*Incentive Mechanism*) A clearly delimited incentive rule targeting a specific dysfunctional behavior.

An incentive mechanism consists of the following three components [23]:

(1) **Evaluation Method**—used to assess the quality of worker's performance from different aspects. Provides input for making a decision whether to apply a reward/sanction.
(2) **Incentive Logic**—represents the business logic behind the incentive mechanism used to interpret evaluation results and decide on application of rewarding actions.
(3) **Rewarding Action**—represents the concrete measure taken against individuals or teams to influence one particular future behavior.

Definition 4 (*Business Artifact*) First-class entity of a business process encapsulating all the information necessary for its processing throughout the entire execution of the business process. The notion of artifact includes not only the 'raw' data that is produced or processed during the business process, but also the metadata describing the lifecycle, relationships with other artifacts and context-dependent information.[2]

The artifacts are identified and described by domain experts. They can correspond to the actual (physical or digital) entities used by the participants in a business process

[2] Adapted from [10].

(such as invoices, bills, source code files, commitment history), or be abstract entities that facilitate the process execution management.

Apart from the obvious purpose of capturing (intermediate) business process goals, each artifact is also supposed to capture the information for evaluating if and how well the goals have been achieved. For example, in addition to the description of the problem and associated fix code, a software bug report artifact may contain the history of actions taken, allowing to draw conclusions on the quality and speed of the work performed on the artifact.

In order to control the evolution of the artifact, each artifact must contain a *lifecycle (model)*.

Definition 5 (*Artifact Lifecycle Model*) The lifecycle model describes the crucial, business-relevant states in which the artifact can be found, as well as rules and constraints governing who, how and when can process the artifact.[3]

The lifecycle model is often formally encoded as a finite state-machine, although other models can be used. It is used to monitor and control the progressing of the artifact through the business process. While the business process owner cannot influence how exactly the process is executed, it can ensure that different artifacts fulfill certain properties at certain times, and with respect to other artifacts, by encoding these expectations and constraints into the lifecycle model. This way, the artifact encapsulates enough information to be able to move through the workflow on its own. Artifact states are used also to monitor the execution of the entire process. At any point during the runtime, the state of the business process is represented by the union of the current states of all the artifacts belonging to the process.

In general case, a single artifact may be changed through different tasks at different times or concurrently. In order to ensure consistency of artifacts' states, these changes need to be performed through transactions.

2.1 Applying Artifact-Centric Incentives in Crowdsourcing Environments

We propose applying the artifact-centric paradigm for defining incentive mechanisms for complex, crowdsourced business processes. Existing incentive mechanisms focus only on the behavior of individuals and teams [23]. The approach we propose here instead focuses on multiple aspects of human participation in business processes at fine-grained levels. It incorporates the existing personal incentive mechanisms and includes them into the novel incentive model.

The novel artifact-centric incentive model should be viewed as an integral part of the artifact itself. This means that the artifact becomes self-sufficient in human-based workflows, in the sense that the artifact itself attracts and motivates workers

[3] Adapted from [18].

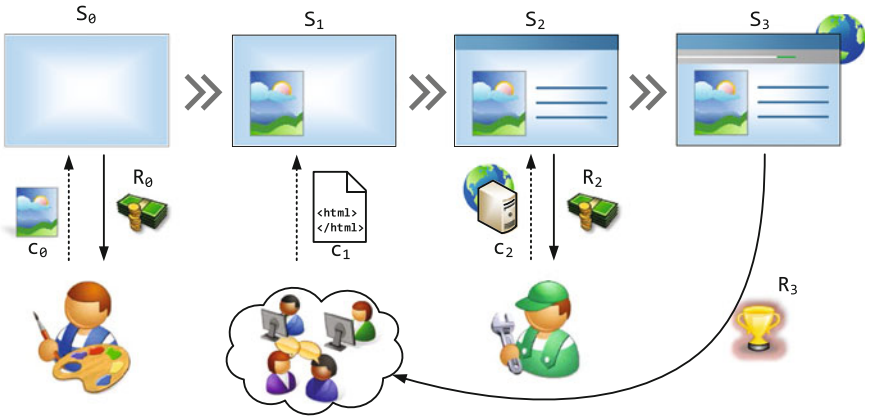


Fig. 1 Artifact-centric representation of a simple software development process. S_i—artifact states. R_i—per-state rewards (incentives). c_i—worker contributions to the artifact's data model

to perform targeted actions and to work through the states of the artifact's lifecycle, effectively performing an artifact-driven orchestration.

To help us illustrate the idea better, let us introduce a simple motivating example employing the concept of artifact-centric incentives that we will use in the rest of the paper for identifying and analyzing various requirements for building such systems:

2.1.1 Motivating Example

Consider a service that crowdsources building of a simple web page for customers (Fig. 1):

A customer submits an informal description of web page requirements (Product Requirement Document—PRD). In order to build the web page, a professional is required to discuss the requirements with the customer in detail and to produce an artifact containing functional specification at the technical level (Functional Specification Document—FSD), which must be approved by the customer. Once the FSD is available, a designer can produce the graphics (GR), and a web developer incorporate the graphics with the programming code to produce the html artifact embedding the graphics (HTML). A tester then uploads the web page, tests it against the FSD, producing a final report (FR), which must be finally approved by the customer.

To keep the use case simple, let us assume that the FSD contains just three lifecycle states—IN_PROGRESS, CUSTOMER_APPROVED and DEVELOPER_APPROVED. Upon submitting the PRD, the new FSD is created and put into IN_PROGRESS state. An incentive associated with this state is offered, e.g., either a FCFS strategy with monetary reward increasing over time, or a reverse auction, as specified by the customer. The customer also specifies other constraints, such as time constraints for setting the artifact into CUSTOMER_APPROVED state, and the minimal quality metrics of workers (reputation, expertise).

The FSD artifact is then offered in the crowdsourcing market. The system that manages the market does not pick out the workers, but rather limits itself to advertising the task (artifact) to potentially interested candidates—those who are available and fulfill the quality requirements.

Once a worker (requirements engineer) applies and commits to working on the artifact, an activity is created for him, as in [30]. Although the creation of a functional specification document usually requires many activities, iterations, document changes and interactions with the customer, the system will not enforce any particular workflow on the worker, but will rather let him organize it completely to his will. The customer's approval will ultimately allow the FSD artifact to transition into the CUSTOMER_APPROVED state.

The FSD now contains some precise graphical requirements and guidelines, out of which a new artifact GR is created. The GR will be used for attracting graphical designers, instructing them, rewarding them, and collecting the produced graphical elements for the web page. It is in the form of a web page, stating the requirements, and promising the reward. A potential incentive strategy here is the tournament reward, where the best designs are awarded, based on the subjective evaluation of the customer [23].

The web developer is chosen similarly to the requirements engineer. The HTML artifact also contains 4 states: AWAIT_FSD_APPROVED, IN_PROGRESS, CUSTOMER_APPROVED, TESTER_APPROVED. Once the developer commits to producing the HTML artifact, he finds the artifact initialized into the AWAIT_FSD_APPROVED state. In order to push the HTML artifact into the IN_PROGRESS state, the developer is required to check the FSD first. If the functional specification is clearly written, and allows him to proceed with writing the code based on it, he sets the FSR into the DEVELOPER_APPROVED state, automatically triggering the transition of the HTML artifact into the IN_PROGRESS state. If the FSD still needs to be improved, the developer resets the FSD into the IN_PROGRESS state, requiring the requirements engineer to work more on it. The remainder of the use case is easy to infer.

2.2 Discussion

Let us first explore how the artifact-centric approach influences modeling of incentives. When the actual monetary reward will be paid to the requirements engineer can depend on many different conditions, and it is exactly the expressive richness of these conditions that makes the artifact-centric incentives so powerful. For example, we may want to specify a much higher reward if the FSD gets developer-approved in the first n iterations. Or, we may want to allow an unlimited number of iterations between the developer and the requirements engineer, but tie the reward amount to a time constraint. Or, most commonly, combine the two incentive mechanisms to promote both speed and excellence.

If the customer expects the FSD to be a big document, the requirements engineer may be incentivized to find and coordinate a small team of helpers that will help speed up the process. The customer can control the number of team members by limiting the number of *roles* that can work on a particular artifact. Different team-incentive mechanisms and reward sharing strategies can be used here—see [23].

The actual payments can be performed after certain state transitions, or only after all the artifacts reach their final states. Furthermore, a deferred team bonus may be promised to all the participants to promote good cooperation between different actors in the process.

As explained in Sect. 4.2, each incentive scheme is vulnerable to the elaborate forms of dysfunctional behavior. In our case, this can be a particular problem, as workers are mostly expected to apply/bid for processing the artifacts themselves, allowing them to coordinate and use different strategies to fool the system. This is why it is very important to foresee and handle these situations. Different methods are deployed to fight this kind of behavior:

- Combination of incentives. If we can foresee the negative application effects of a single incentive mechanism, then we can also construct new incentive mechanism to discourage this kind of behavior.
- Commitment protocols. Offering different commitment protocols restricts workers from maliciously obtaining the benefits on account of other collaborators [7].
- Semi-active worker selection and randomization. This presupposes initially choosing the suitable (reputable) workers, and allowing only them to bid for tasks. Additionally, a non-best bid may be randomly chosen to discourage fixing of prices.
- Sealed-bid auctions. They prevent bidders from seeing the offered prices of others.

This short discussion demonstrates why incentive mechanisms need to be specified at various finer-grained levels, rather than at the business process level only. In fact, we can identify the following dimensions/levels for which incentive mechanisms should be definable:

- **State-dependent incentive mechanisms**. Mechanisms associated with a state of the artifact. The state can be represented not only by a "real" state in the life-cycle model, but also by a combination of values of different metrics, such as: current quality, the number of past contributors, current price, urgency, accuracy, importance, etc.
- **Temporal incentive mechanisms**. Mechanisms conditioning the rewarding action with temporal constraints, e.g., reward may increase as a deadline approaches.
- **Artifact-interdependent incentive mechanisms**. Mechanisms allowing users to specify other artifacts to be processed together/dependently/independently (or in different patterns) with this artifact; or, make the reward payment dependent on the outcome/state of another artifact. These incentives would stimulate the crowd to self-organize and loosely follow the data and control flow we envisaged.

- **Personal incentive mechanisms** on:
  - *Individual level*. Mechanisms targeting individuals, or intended to attract specific types of workers (e.g., experienced, efficient, creative, reputable)
  - *Team level*. Mechanisms designed to promote team efforts on the artifact, e.g., by promising team-based rewards.

3 Requirement Analysis

In Sect. 3.1 we will further analyze these abstraction dimensions, and formulate requirements for designing a novel incentive mechanism model to cover them. This model is meant to extend the conventional lifecycle model of the business artifacts. In Sect. 3.2 we will then introduce and analyze a set of crucial requirements for building and managing sustainable crowdsourcing careers over longer periods of time and different employment providers.

3.1 Requirements for Artifact's Incentive Model

3.1.1 State-Dependent Incentive Mechanisms

Finite state machines are the most commonly used formalism to describe the lifecycle model of an artifact due to their expressiveness, comprehensible semantics and tool support. Consequently, it comes natural to use artifact states in conditions for applying incentive mechanisms. However, since most artifacts are documents intended to be processed by humans, their lifecycle models need to be kept reasonably simple. This means that we often lack the fine granularity needed for expressing an incentive condition.

This is why we propose that, apart from the artifact's regular lifecycle states visible to humans and used for guiding the business process—*hard states*, a set of machine-processable *soft states* for regulating incentives also be specified. Soft states could be defined as sub- or super-states of existing states that contain no entry or exit transitions, but are 'entered' whenever the lifecycle model is in the associated hard state and the *entry predicate* for the soft state holds true.

Entry predicates would allow us to specify various metric thresholds as conditions for applying an incentive mechanism. In our motivation example, this would allow us to introduce a metric *transCnt* that would keep count of the number of transitions between CUSTOMER_APPROVED and IN_PROGRESS hard-states of the FSD artifact. A super-softstate DISAGREEMENT, comprising both CUSTOMER_APPROVED and IN_PROGRESS can be defined with the entry predicate: $transCnt > 5$ (Fig. 2). Detecting that there is a disagreement on the functional specification between the requirements engineer and the developer is an important fact to

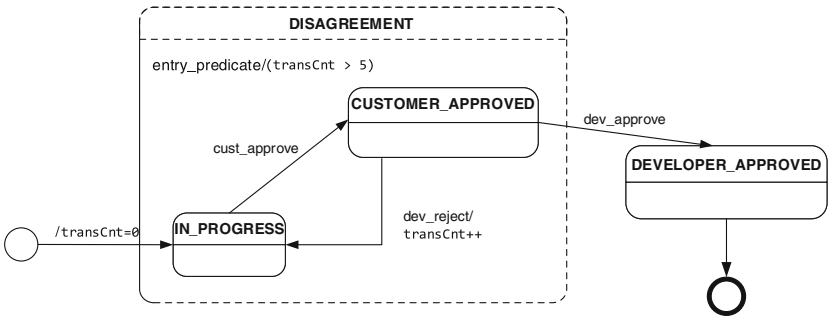


Fig. 2 State-dependent incentives. Soft states are outlined with the *dashed line*

consider when deciding which incentive mechanisms to apply. In our case, entering the DISAGREEMENT state could be used as signal for applying an incentive mechanism that will help resolve the issue, e.g., by promising a penalty if the agreement is not met in a specified time, or by discontinuing the engagement of the workers.

Of course, incentive conditions could be specified just as predicates for the purpose of constructing incentive mechanisms, i.e., without introducing the notion of soft states. However, conceptualizing the conditions as states and associating them with hard states forces incentive designers to use the artifact-centric paradigm, reducing the number of possible conditions and making them addressable entities in the model. Also, in order to exhibit effect, certain incentive mechanisms must be presented in advance to the workers. In these situations it is helpful to have a limited number of incentive conditions associated with artifact states, making the incentives transparent and understandable to workers.

However, the main advantage of this approach is separation of concerns; while an artifact's hard states can be standardized for use throughout different companies, company-specific soft states can be defined to support specific incentives and applied to existing artifact lifecycle models without affecting their primary usage.

3.1.2 Temporal Incentive Mechanisms

Including temporal dimension in the artifact's lifecycle model is essential, as incentive mechanisms exhibit their effects only if promised in advance and applied upon an action is completed. Furthermore, it is essential to be able to encode proper/expected ordering of events leading to a reward or punishment, or to detect activities taking too much time. Therefore, the time management must include both time-interval semantics, as well as the event ordering.

A way to meet these requirements would be incorporating the time model and the operators of the Linear Temporal Logic (LTL). LTL operates on a simple, discrete, linear time model, isomorphic to the set of natural numbers $\mathbb{N}$. The time moments (*ticks*) are therefore counted from the agreed 'beginning of time' onwards. The events

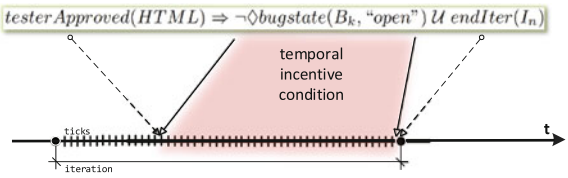


Fig. 3 A temporal incentive condition encoded in LTL

happen at ticks. Events and states are represented by logical propositions that can treated with a set of temporal and standard logical operators.

While any platform-specific implementation of incentive mechanisms must include time queries in some way, to the best of our knowledge, there are no known systematic approaches to modeling temporal logic operators in the domain of incentive management. In the area of BPM, on the other hand, we have seen successfull attempts of including LTL into process models [19].

We propose introducing declarative LTL constructs for incentive mechanisms *on the artifact level* by applying similar principles as in [19]. The LTL constructs can be used to express temporal propositions for various incentive conditions. For instance, looking back at our example, we can specify that *after* the HTML artifact gets into the TESTER_APPROVED state, a BUG_REPORT artifact should *never* be approved for a missing feature. Of course, in real systems, 'never' will have a limited duration, after which the whole proposition should expire, e.g., after an iteration's end (Fig. 3).

Another beneficial notion we suggest be introduced into the artifact lifecycle model is the notion of *iterations*. Iterations are time intervals with just-in-time initialization and finalization. They can be used for representing work phases meaningful to humans that are inherently unstable, such as sick leave, working hours or project phase. This means that we could define an iteration for the purpose of describing that phase. However, when designing an incentive mechanism, we may not know exactly when the iteration would start, nor when it would end. Therefore, we would express the incentive conditions by using iterations, rather than ticks, and leave it to the underlying system to signal the iteration's starting and ending times and handle the incentive execution. The iteration abstraction can be expressed in LTL, but we suggest using it along with the standard LTL operators for simplifying the time management as it corresponds better to the organization of human work.

Including declarative LTL constructs on the artifact level adds a new dimension of expressiveness to the incentive mechanisms. In addition, the constructs can be used for runtime monitoring of crowdsourcing platforms for specifying temporal invariants.

3.1.3 Artifact-Interdependent Incentive Mechanisms

In complex collaborative efforts, such as software development processes, the lifecycle of a single artifact cannot be considered independently of the states of other artifacts in the business process. Therefore, it becomes imperative to formally capture these dependencies in the lifecycle model.

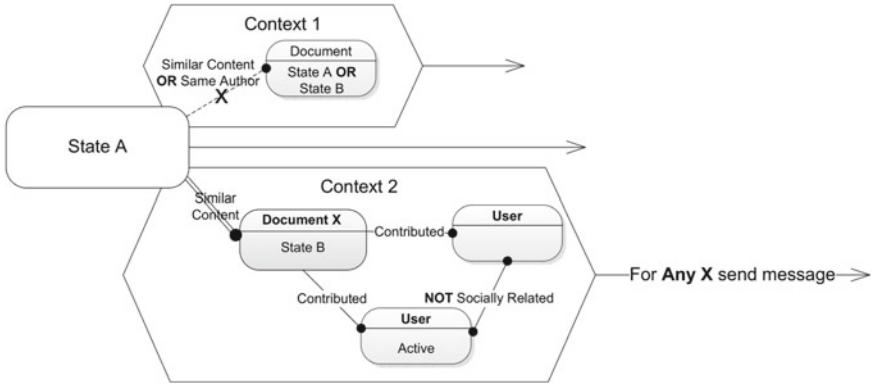


Fig. 4 An example of artifact interdependency contexts (from [14])

We propose formalizing the dependencies among different artifacts so that they can be used to express different incentive conditions. These conditions can then be integrated into the incentive model used to augment the lifecycle model of the artifact. To the best of our knowledge, the concept of relating the lifecycle states of different artifacts to express incentive conditions has never been formally proposed before.

The paper [14] presents one possible formalism that could be adapted for such a purpose. It enriches the conventional artifact lifecycle model by introducing the notion of 'state contexts' and 'context-aware state transitions'. The contexts are defined graphically. Relationships between artifact and role entities in context definition offer the expressiveness of the first-order predicate calculus. This allows us to express the necessary artifact interdependencies. Figure 4 (borrowed from [14]) displays an example of the graphical notation. For more information, the reader is referred to the original paper.

For example, we could use this notation to express that the FSR needs to be moved into the DEVELOPER_APPROVED state first in order for the HTML artifact to move into the IN_PROGRESS state. But we can also use the same notation to, for example, prevent a reward being paid if there is at least one bug report in the unresolved state. The benefit of using a graphical notation that includes universal and existential quantifiers is that it makes it easy for humans to specify and reuse this type of conditions.

3.1.4 Personal Incentive Mechanisms

As each personality is different, a single incentive can never work the same way on every person. The conventional approach when designing the incentives for a particular crowd effort is to select those that suite best an average worker in the targeted group. However, unless this group is large enough this approach may not perform well. Indeed, in limited efforts, just a few participants with a particular

interest or affinity for that task may contribute much more than the rest of the crowd [21]. Therefore, for assembling small-scale teams focused on specific tasks/artifacts it is important to identify and attract such individuals. One way of achieving this is through personalized incentives.

For example, if we want to attract a promising, young software engineer to our team, then we cannot expect him to be able to solve certain tasks as fast as an engineer already experienced in that area. That is why we may be willing to value and reward his effort levels rather than his speed. We may also tolerate certain errors (e.g., failed code reviews, reopened bugs) and not penalize him, hoping to improve his engineering skills for future collaborations. On the other hand, employing an experienced senior engineer implies paying him more, but also evaluating his performance on speed and quality metrics.

Therefore, the artifact's incentive model should offer different "incentive packages" (Fig. 5) that consider different metrics and promise different rewards appealing to different groups of workers. Incentive packages are a way of including existing research on modeling personal incentives (see [23]) into the new artifact-centric

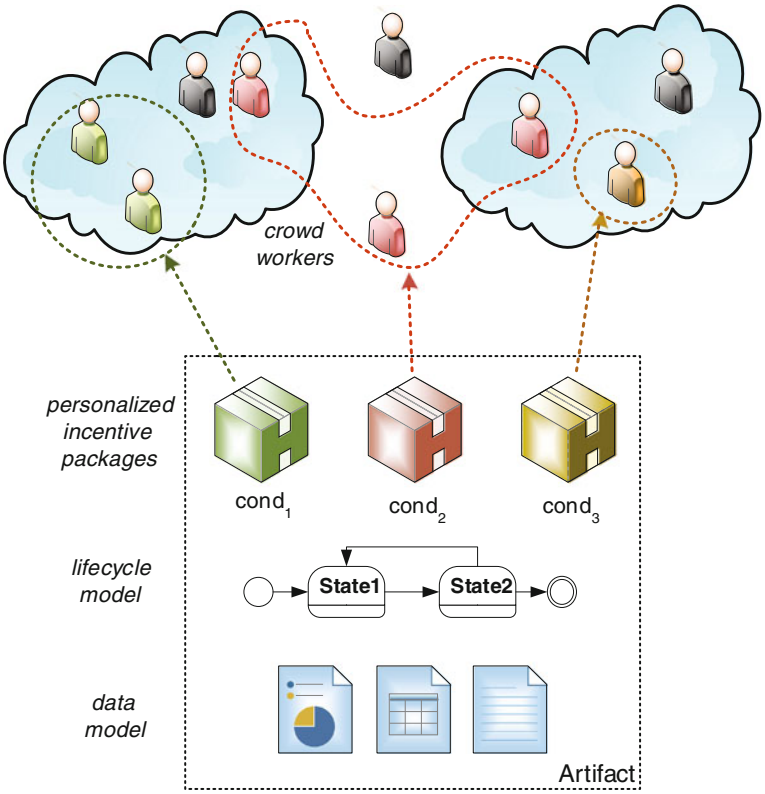


Fig. 5 Personalized incentives help attract (groups of) workers with specific properties

paradigm. It should be possible to enable/disable incentive packages as needed, e.g., when enough workers apply for one incentive package, or when the reward money runs out. Also, it should be possible to specify inter-package enabling conditions—e.g., requiring a number of workers (non-)applying for another package first (or at the same time). For instance, an incentive package targeting a team lead should be enabled only if the package meant to attract developers managed to attract a sufficient number of appropriate candidates.

Incentive packages could in special cases target particular individuals rather than groups. In this case, we can rely on the particular worker's behavioral history to infer (by machine learning) potentially interesting activities, tasks and collaborators to the worker. If the artifact's lifecycle model foresees a potentially favorable set of conditions that could attract this particular worker, then a tailored incentive can be offered to attract him to work on the artifact. Multiple individuals could be targeted in parallel, but each worker could only claim the reward of his personal incentive package.

An interesting example where this approach could be successfully used are the so-called *structural incentives*, i.e., incentives that motivate people by promising to establish certain social or professional collaboration relationships/patterns between workers. For example, young professionals may find the possibility to collaborate with renowned experts to be more attractive in a short term than a higher salary because of the prestige associated with it. Similarly, the possibility to collaborate with known and trusted collaborators from the past [25] may be the determining factor in choosing to work on one artifact over another.

For example, by analyzing the code repository logs we could determine that developer A often collaborated with developers B and C on the same .java files, and that they were often reviewing each other's code submissions. Based on the code snippets they were submitting, we can infer their common expertise, e.g., which databases or libraries they used [26]. This gives us reason to believe that the same three persons collaborating on a new project within their area of expertise are probably going to be productive. For this reason, we may want to incentivize them to join our effort, and put out three individual incentive packages targeting them. The incentive for developer A could contain the condition that at least one of the other two developers would also have to accept working on the artifact. The packages for developers B and C would be similar.

It is probable that the developers A, B and C would more likely join an effort with known collaborators. Therefore, the application of multiple personalized incentives can also exhibit a significant group effect, while transferring the organizational and motivational burden onto workers themselves, since they would be persuading each other much more efficiently than an automated system could do.

While personal incentives can achieve powerful motivating effects, their expressiveness and limitations fully depend on the adopted underlying model of personal incentives. However, rather than discussing the properties and limitations of the different existing personal incentive models, here we limit ourselves to suggesting how the existing models can be integrated into the encompassing artifact-centric incentive model.

3.2 Requirements for Sustainable Crowdsourcing Careers

One of the biggest problems when dealing with incentives in crowdsourcing in general (and especially with personal incentives) is selecting and defining metrics to accurately describe current worker contributions and appropriately interpret past performance in the current context. Solving this problem would in theory allow different employers to track and update the performance history of the crowd workers in a uniform way, and allow the workers to use the reputation records with different employers very much like CVs and recommendation letters are used today. We call this (temporal and locational) *transfer of reputation*. The notion of transferable reputation is one of the key enabling conditions for successful application of personal incentives.

Unfortunately, defining a comprehensive set of metrics to cover so many aspects of human work that would allow us to build a uniform record of one's working history is impossible. Even though, for certain highly-specific domains, it may be possible to define referent metrics ontologies, in majority of real-life applications this is not a viable solution, nor one that will likely get embraced by the employers. Furthermore, a metric's relevance may change with time.

This is why we suggest not to predefine specific metrics to be kept, but rather keep a public history of worker's performance and employ a *reputation service* for *just-in-time metric assessment* as a cost-effective alternative to the development of dedicated metrics. In this way, the ad-hoc invoked service would map a worker's performance records spanning a specified time period into a set of given, context-specific metrics of interest to the current employer.

Different reputation service providers could offer different QoS at different prices, according to the needs of the employer (Fig. 6). For example, for performing a simple programming task, the employer may require "someone with basic programming skills". This means that the reputation service needs to return a metric indicating whether a candidate has done programming before. A software web service that will check the candidate's activity metrics on sites such as StackOverflow, or recommendations on sites such as LinkedIn can be employed here to return/calculate a rough estimate of the worker's reputation. However, the service will produce results immediately, and will cost little. On the other hand, if the employer needs "an Informix database security expert" in his team, then the employer may want to use a human-based service (HBS) [24] employing *subjective evaluations* [23] from other software developers who would be asked to review the candidate's personal work history or even his code from open-source projects. BetterQoS, though, would probably imply longer invocation times and higher price. Invocation results should be appended to the existing worker's history, and serve as another piece of data valuable for future evaluations, especially for monitoring the development of worker's skills and working attitude.

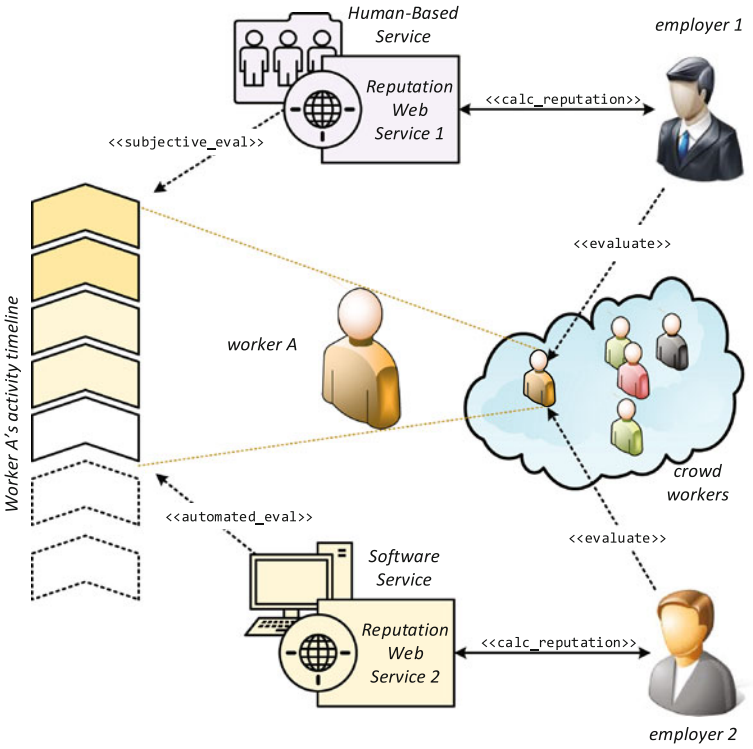


Fig. 6 Reputation is evaluated from worker's public history records and interpreted upon request through reputation service

This approach would allow employer to keep using any internal labor metrics he wants, while allowing transfer of reputation through shared activity history whose meaning is mapped to particular metrics via the reputation service.

3.3 Requirements Summary

In previous sections we explored the different aspects we find worth of including in a future incentive model for socio-technical/crowdsourcing systems. We presented suggestions in form of requirements, providing simple, but illustrative examples as justification, and discussing potential issues and benefits. Table 1 presents a high-level summary of this requirement analysis. Although non-exhaustive, it provides a useful overview of the different levels at which incentives need to be addressed, opening up space for more focused research.

Table 1 Summary of requirements for supporting artifact-centric incentive mechanisms in crowdsourcing environments

	Application level	Motivation	Proposed requirements
Incentive model level	Artifact state	Defining per-state incentive mechanisms	Incentive conditions as soft states
	Time	Expressing temporal incentive conditions	Time-interval semantics and event ordering
		Scheduling of deferred rewarding actions	Iterations
		Evaluating/mining past work	
	Inter-artifact	Allowing the states of other artifacts influence incentives offered for processing this artifact	Formalism for expressing artifact inter-dependencies
	Worker(s)	Attracting particular individuals to work on artifact	Personalized incentive packages
			Inter-package dependencies
		Transferring organizational effort to humans	Structural incentives
Inter-organizational level	Spanning multiple crowdsourcing employers	Enabling transfer of worker reputation between employers	Public record of worker performance
		Creating favorable conditions for sustainable "careers in the cloud"	Ad-hoc, (human-) service-based interpretation of the metrics in current context

4 Related Work

4.1 Artifact-Centric Business Process Modeling

Artifact-centric BPM, also known as 'document-centric' or 'data-driven' BPM has attracted a lot of research attention in the past. Here we will review only a small selection of fundamental papers that enable the reader to understand the background and motivation for our approach.

One of the landmark ideas of the artifact-centric paradigm is that it is possible to design workflow systems without explicit control flow, where the actual execution is governed by the artifacts themselves, also serving as input as outputs. The paper

[30] presets a prototype implementation of a document-driven workflow system, demonstrating the feasibility of this approach. In [18] the authors informally describe the business artifact concept and its lifecycle models, while [1] introduces a formal model and operational semantics. Authors of [8] analyze the problem of verification of artifact behavior in operational models. The paper [16] presents a methodology and patterns for building up real business operational models using artifacts. Finally, [10] presents a comprehensive survey of the fundamental research on artifact-centric BPM.

For an overview of more recent developments in the area, the reader is referred to the following publications: [3, 4, 6, 11, 28, 29].

4.2 Incentives and Rewarding

Related work in the area of incentives originates mostly from economics, game theory, organizational science and psychology. The principal economic theory treating incentives today is the *Agency Theory* [2, 13]. Incentives are defined as the principal mechanism for aligning interests of business owners and workers. As a single incentive always targets a specific behavior and induces unwanted responses from workers [13], multiple incentives are usually combined to counteract the dysfunctional behavior and produce wanted results. Opportunities for dysfunctional behavior increase with the complexity of labor, and so does the need to use and combine multiple incentives. The paper [20] presents a comprehensive review and comparison of different incentive strategies in traditional businesses.

The number of computer science papers treating these topics is limited. Incentives are discussed usually within particular, application-specific contexts, like peer-to-peer networks, agent-based systems and human-labor platforms (e.g., Amazon Mechanical Turk), rather than being considered at a general level. In [22] the aim is to introduce appropriate incentives to maximize peer-to-peer content sharing. In [31] the authors seek to maximize the extent of social network by motivating people to invite others to visit more content. In [15] the authors try to determine quality of crowdsourced work when a task is done iteratively compared to when it is done in parallel. In [17] the authors investigate how different monetary rewards influence the productivity of mTurkers. In [21] the authors compare the effects of lottery incentive and competitive rankings in a collaborative mapping environment. In [9] the authors analyze two commonly used approaches to detect cheating and properly validate submitted tasks on popular crowdsourcing platforms.

An overview of typical incentives and rewarding practices in crowdsourcing systems can be found in [23, 27]. A common conclusion is that incentives used in today's social computing platforms are mostly limited to simple piece-rates that may be suited for simple task processing, but are inappropriate for the more advanced collaborative efforts such as software development. However, both studies suggest that, depending

on the environment, there exist appropriate types of incentives that combined together should succeed in motivating and rewarding workers taking part in more complex or intellectually more challenging labor activities.

5 Conclusion

We believe that, in order to support collaborative processes of increased complexity, the crowdsourcing platforms will need to leverage human-based services to tackle important challenges such as team formation, adaptability and runtime management of collaboration processes. However, introducing humans into the loop requires specific methods for attracting, motivating and controlling humans. We suggested this could be done with a combination of artifact-centric workflows and rich incentive mechanisms. We then analyzed different aspects that an incentive mechanism model for such systems should cover, and suggested integrating it into the artifact lifecycle model to create encapsulated units that can be offered to the crowd for processing. The novel artifact model would allow the crowd workers to independently drive the processing in the envisioned direction and tackle the aforementioned challenges. The result of our analysis is a set of requirements that the future systems should support, ultimately providing a working environment that would promote fairness, worker's reputation transfer and ultimately, a fundamental step towards building a framework for managing "careers in the cloud".

References

1. Bhattacharya, K., Gerede, C., Hull, R., Liu, R., Su, J.: Business Process Management, pp. 288–304. Springer, Berlin (2007). doi:10.1007/978-3-540-75183-0_21
2. Bloom, M., Milkovich, G.: The relationship between risk, incentive pay, and organizational performance. Acad. Manag. J. **41**(3), 283–297 (1998)
3. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. IEEE Data Eng. Bull. **32**(3), 3–9 (2009)
4. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. Inf. Syst. **38**(4), 561–584 (2013). doi:10.1016/j.is.2012.09.002
5. Dustdar, S., Truong, H.L.: Virtualizing software and humans for elastic processes in multiple clouds—a service management perspective. Int. J. Next-Gener. Comput. **3**(2), 109–126 (2012)
6. Fritz, C., Hull, R., Su, J.: Automatic construction of simple artifact-based business processes. In: International Conference on Database Theory (ICDT '09), p. 225 (2009). doi:10.1145/1514894.1514922
7. Gal, Y., Grosz, B., Kraus, S., Pfeffer, A., Shieber, S.: Agent decision-making in open mixed networks. Artif. Intell. **174**(18), 1460–1480 (2010). doi:10.1016/j.artint.2010.09.002
8. Gerede, C., Su, J.: Specification and verification of artifact behaviors in business process models. In: International Conference on Service-Oriented Computing (ICSOC 2007), pp. 181–192 (2007)
9. Hirth, M., Hossfeld, T., Tran-Gia, P.: Analyzing costs and accuracy of validation mechanisms for crowdsourcing platforms. Math. Comput. Model. (2012). doi:10.1016/j.mcm.2012.01.006

10. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: On the Move to Meaningful Internet Systems (OTM), pp. 1152–1163 (2008)
11. Hull, R.: Towards flexible service interoperation using business artifacts. In:15th IEEE International Conference on Enterprise Distributed Object Computing (EDOC), pp. 20–21. (2011). doi:10.1109/EDOC.2011.27
12. Kaganer, E., Carmel, E., Hirschheim, R., Olsen, T.: Managing the human cloud. MIT Sloan Manag. Rev. **54**(2), 22–32 (2013)
13. Laffont, J.J., Martimort, D.: The Theory of Incentives. Princeton University Press, New Jersey (2002)
14. Liptchinsky, V., Khazankin, R.: A novel approach to modeling context-aware and social collaboration processes. In: 24th International Conference on Advanced Information Systems Engineering (CAiSE'12). Springer, Gdansk (2012). doi:10.1007/978-3-642-31095-9_37
15. Little, G., Chilton, L.B., Goldman, M., Miller, R.C.: Exploring iterative and parallel human computation processes. In: Proceedings of the ACM SIGKDD Workshop on Human Computation, HCOMP'10, pp. 68–76. ACM, New York(2010). doi:10.1145/1837885.1837907
16. Liu, R., Bhattacharya, K., Wu, F.: Modeling business contexture and behavior using business artifacts. In: J. Krogstie, A. Opdahl, G. Sindre (eds.) Advanced Information Systems Engineering. Lecture Notes in Computer Science, vol. 4495, pp. 324–339. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72988-4_23
17. Mason, W., Watts, D.J.: Financial incentives and the performance of crowds. In: Proceedings of the ACM SIGKDD Workshop on Human Computation (HCOMP'09), vol. 11, pp. 77–85. ACM, Paris(2009). doi:10.1145/1600150.1600175
18. Nandi, P., Kumaran, S.: Adaptive business objects—a new component model for business integration. In: Proceeding of the 7th International Conference on Enterprise Information Systems (ICEIS'07), pp. 179–188. Miami(2005)
19. Pesic, M., Schonenberg, H., van der Aalst, W.M.: DECLARE: full support for loosely-structured processes. In: 11th IEEE International Conference on Enterprise Distributed Object Computing (EDOC'07), pp. 287–287. IEEE (2007). doi:10.1109/EDOC.2007.14
20. Prendergast, C.: The provision of incentives in firms. J. Econ. Lit. **37(1)**, 7–63 (1999). http://www.jstor.org/stable/2564725
21. Ramchurn, S., Huynh, T., Venanzi, M., Shi, B.: Collabmap: crowdsourcing maps for emergency planning. In: Proceedings of the ACM Web Science, Paris, France (2013). http://eprints.soton.ac.uk/350677/
22. Sato, K., Hashimoto, R., Yoshino, M., Shinkuma, R., Takahashi, T.: Incentive mechanism considering variety of user cost in P2P content sharing. In: Global Telecommunications Conference (IEEE GLOBECOM '08), pp. 1–5. IEEE (2008). doi:10.1109/GLOCOM.2008.ECP.426
23. Scekic, O., Truong, H.L., Dustdar, S.: Incentives and rewarding in social computing. Commun. ACM **56**(6), 72 (2013). doi:10.1145/2461256.2461275
24. Schall, D., Dustdar, S., Blake, M.B.: Programming human and software-based web services. Computer **43**(7), 82–85 (2010). doi:10.1109/MC.2010.205
25. Schall, D., Skopik, F., Psaier, H., Dustdar, S.: Bridging socially-enhanced virtual communities. In: Proceedings of the ACM SAC 2011 (2011)
26. Teyton, C., Falleri, J.R., Blanc, X.: Mining library migration graphs. In: 19th Conference on Reverse Engineering, pp. 289–298. IEEE (2012). doi:10.1109/WCRE.2012.38
27. Tokarchuk, O., Cuel, R., Zamarian, M.: Analyzing crowd labor and designing incentives for humans in the loop. IEEE Internet Computing, pp. 45–51. (2012). doi:10.1109/MIC.2012.66
28. Vaculin, R., Heath, T., Hull, R.: Data-centric web services based on business artifacts. In:19th International Conference on Web Services (ICWS'12) (1), 42–49 (2012). doi:10.1109/ICWS.2012.101
29. Vaculin, R., Hull, R., Heath, T., Cochran, C., Nigam, A., Sukaviriya, P.: Declarative business artifact centric modeling of decision and knowledge intensive business processes. In: 15th International Conference on Enterprise Distributed Object Computing (EDOC'11), pp. 151–160. IEEE (2011). doi:10.1109/EDOC.2011.36

30. Wang, J., Kumar, A.: A framework for document-driven workflow systems. In: W.M. van der Aalst, B. Benatallah, F. Casati, F. Curbera (eds.) Proceedings of International Conference on Business Process Management (BPM'05). Lecture Notes in Computer Science, vol. 3649, pp. 285–301. Springer (2005). doi:10.1007/11538394_19
31. Yogo, K., Shinkuma, R., Takahashi, T., Konishi, T., Itaya, S., Doi, S., Yamada, K.: Differentiated Incentive Rewarding for Social Networking Services pp. 169–172 (2010). doi:10.1109/SAINT.2010.65

An Evolutionary and Automated Virtual Team Making Approach for Crowdsourcing Platforms

Tao Yue, Shaukat Ali and Shuai Wang

Abstract Crowdsourcing has demonstrated its capability of supporting various software development activities including development and testing as it can be seen by several successful crowdsourcing platforms such as TopCoder and uTest. However, to crowd source large-scale and complex software development and testing tasks, there are several optimization challenges to be addressed such as division of tasks, searching a set of registrants, and assignment of tasks to registrants.Since in crowdsourcing a task can be assigned to registrants geographically distributed with various backgrounds, the quality of final task deliverables is a key issue. As the first step to improve the quality, we propose a systematic and automated approach to optimize the assignment of registrants in a crowdsourcing platform to a crowdsourcing task. The objective is to find the best fit of a group of registrants to the defined task. A few examples of factors forming the optimization problem include budget defined by the task submitter and pay expectation from a registrant, skills required by a task, skills of a registrant, task delivering deadline, and availability of a registrant. We first collected a set of commonly seen factors that have impact on the perfect matching between tasks submitted and a virtual team that consists of a selected set of registrants. We then formulated the optimization objective as a fitness function‖the heuristics used by search algorithms (e.g., Genetic Algorithms) to find an optimal solution. We empirically evaluated a set of well-known search algorithms in software engineering, along with the proposed fitness function, to identify the best solution for our optimization problem. Results of our experiments are very positive in terms of solving optimization problems in a crowdsourcing context.

T. Yue (✉) · S. Ali · S. Wang
Certus Software V&V Center, Simula Research Laboratory, Oslo, Norway
e-mail: tao@simula.no

S. Ali
e-mail: shaukat@simula.no

S. Wang
e-mail: shuai@simula.no

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_7

1 Introduction

Crowdsourcing software engineering is gaining more and more attention these days as increasing number of companies start looking for an innovative way to develop software and conducting other software engineering activities such as testing. The main reason is that the cost can be significantly reduced. Moreover some crowdsourcing platforms such as Topcoder[1] and UTest[2] have shown their success and a large number of registrants of these platforms form a large virtual pool for performing tasks virtually. However, to compare with traditional software development practices, crowdsourcing software engineering is still at its early stage, which leaves a lot of space for research. Especially large-scale software engineering on crowdsourcing platforms are facing a lot of challenges, one of which is how to decompose, schedule and integrate tasks such that the overall quality and productivity can be maintained as they are developed in a traditional software development environment.

Towards supporting large-scale, crowdsourcing software engineering, in this chapter, we propose a search-based approach, along with a series of experiments to demonstrate how search-based software engineering can be applied to address optimization problems in crowdsourcing software engineering. In this chapter, we particularly focus on assisting platform managers to form a virtual team on a crowdsourcing platform for a submitted task such that the overall quality and productivity of performing this task can be ensured to a certain extent. This is an optimization problem as there are some constraints to find such a virtual team. For example, the cost to hiring the team members of the virtual team should be within the budget, the task should be completed within certain duration, and the task should match the background of the virtual team members.

The core of Search Based Software Engineering (SBSE) are search algorithms (e.g., Genetic Algorithms mimicking natural selection process) that can efficiently find optimal solutions to the problems that have large complex search spaces. Typical examples of such problems in software engineering include: optimal allocation of requirements, optimal architecture design, test case generation, and test optimization. According to the comprehensive review of Harman et al. [11], SBSE has been extensively investigated to address various software engineering problems spanning from requirements, testing to reengineering of a typical software development lifecycle. Particularly for requirements, SBSE has been applied for various optimization problems such as requirements selection [6], prioritization [7], and assignment [10, 14], with different objectives such as maximizing customers'/stakeholders' satisfaction, maximizing benefits/value and minimizing cost. For testing SBSE has been applied to successfully address test generation and test optimization problems [1, 11]. To the best of our knowledge, SBSE has never been applied to address optimization issues

[1] http://www.topcoder.com/.

[2] http://www.utest.com/.

existing in crowdsourcing. In this paper, we mainly present a pilot study we recently conducted to demonstrate that SBSE can also be applied for addressing optimization issues in crowdsourcing.

The rest of paper is organized as follows. Section 2 provides the overview of our approach. Section 3 presents a conceptual model that formalizes key elements of our approach. In Sect. 4, our search-based crowdsourcing methodology and results of the experiment we conducted to evaluate the fit-ness function, the key element of our search-based crowdsourcing methodology. Section 5 discusses the threats to validity of our experiments and we conclude the paper and discuss the future work in Sect. 6.

2 Overview

In this section, we provide an overview of the approach we propose in this paper and its extensions for future, as shown in Fig. 1. We classify stakeholders that are relevant to a crowdsourcing platform into four groups: Task Submitter, Crowd, Platform Manager and Virtual Team. A task submitter is a person who submits a task through the crowdsourcing platform and looks for a virtual team (from the crowd) to complete the task. A platform manager (employee of the crowdsourcing platform) is assigned to manage the task (including assisting the formation of the virtual team and negotiation between the task submitter and the crowd) on the behalf of the crowdsourcing platform. The virtual team is formed by selecting a group of registrants of the crowdsourcing platform who expressed their willing to complete the task via the crowdsourcing platform.

Our objective is to propose a solution, integrated as part of the services provided by the crowdsourcing platform, to assist the platform manage to form a virtual team according to the requirements from the task submitter (provided as part of the task description) at the same time satisfying the expectation of the virtual team members. In other words, such a solution aims to find a match between the task submitter side and the virtual team members. Doing so, we believe, will indirectly improve the quality and productivity of software development activities via a crowdsourcing platform.

In Fig. 1, we highlight the key features, properties and technologies to apply of our solution. It is important to notice that such a solution is Generic in the sense that it is not specific to a particular type of task that a crowdsourcing platform can provide such as testing. Therefore, the solution can be widely applied in any crowdsourcing platform, as far as we can see.

The first key component of our solution is to provide a set of specification methodologies for task submitters to specify tasks (including budget, duration, task content, requirements) and for an individual to define her/his profile (e.g., experience, skills). Such specification methodologies ideally should be easy to use for end users (in our context, task submitters, crowd and platform manager). Submitted tasks and crowd profiles specified using these methodologies could then be automatically collected,

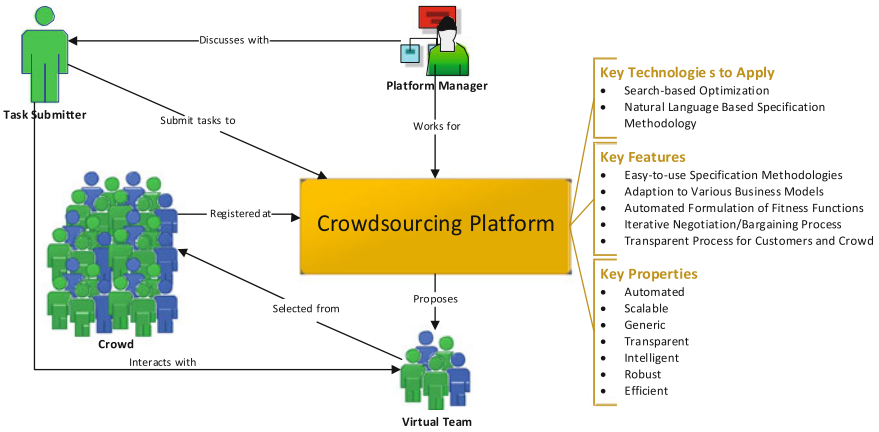


Fig. 1 Overview

analyzed to serve other components of the platform such as the automated formation of fitness functions.

After all these information are collected by the crowdsourcing platform, our solution is then ready to take a task submitted by a task submitter and automatically propose a solution, which is a virtual team selected from the crowd. The platform manager can then coordinate the virtual team and the task submitter to complete the task. We rely on search-based optimization methodologies to automatically identify a virtual team that is optimal in the sense that in the scope of the crowd, the profiles of the team members of the virtual team fit the requirements of the task best. It is worth noting that search algorithms work together with carefully designed fitness functions, which are used to guide search algorithms towards the direction of finding an optimal solution. Such methodologies are *Automated*, *Scalable* and *Efficient*. Details of the search-based methodologies will be provided in Sect. 4.1. In case that all the required information to derive a fitness function for search algorithm can be extracted automatically from the task specifications and crowd profile specifications, the derivation of a fitness function can be *Automated*.

It is important to notice that we are not aiming to replace platform managers. Instead, we aim to assist platform managers to better conduct their jobs. For example, the process of identifying a virtual team from the crowd to perform a task, if it is required to be manually done by a platform manager, she/he has to go through the registrants' profiles, their bids and try to, mostly based on their experience, to form a virtual team that can complete the task submitted and satisfies the constraints such as budget, time schedule and required expertise. One can instantly understand that if the task is complex enough to require a virtual team with more than 10 members, different expertise, and tight schedule, which is often the case for supporting large-scale crowdsourcing software development, manually forming such a virtual team satisfying all these constraints is simply unmanageable. Therefore, this inspires us to pro-pose an automated, scalable, intelligent, robust and efficient solution to assist

platform managers. In addition, our solution can be easily customized for catering needs of various crowdsourcing platforms executing different business models.

Ideally a crowdsourcing platform should be able to provide an effective mechanism to support the negotiation between a task submitter and the crowd in terms of price, schedule, etc. The common practice is that a platform manager plays the role to coordinate the negotiation or bargaining process without any intelligent support from the platform, which might lead to low productivity and therefore less customer satisfaction. We however propose a simulation-based, intelligent negotiation/bargaining process. Based on our search-based methodologies, we can instantly find a solution that satisfies a set of constraints that are defined based on the results of a round of negotiation. A new negotiation implies updating this set of constraints and therefore triggers the execution of our search-based optimization methodologies to find another solution satisfying the updated set of constraints.

Depending on the business model adopted by a crowdsourcing platform, it might be useful to make the formation of virtual teams, the negotiation process transparent to task submitters and the crowd as well. Doing so might somehow lead to a healthier (virtual) working environment on the crowdsourcing platform and therefore indirectly contributing to a higher quality and productivity of the development process via the crowdsourcing platform.

3 Conceptual Model

In this section, we formally specify concepts that are related to the method-ologies we propose as a conceptual model in UML class diagram (Fig. 2). Each concept is presented as a class and the relationships among concepts are captured as UML associations or generalizations. From the figure, one can notice that we can classify stakeholders into three types: `Registrant`, `Submitter`, and `Platform-ProjectManager`.

`Registrant` registered her/himself on the crowdsourcing platform and specified her/his profile accordingly. Such a `Profile` should contain a list of information that is required to evaluate an individual based on his/her programming language skills, natural language skills and rank at the platform, experience, etc. It is important to notice that in our conceptual model, we capture the classifications of `expertise`, `programmingLanguageSkill` and `naturalLanguage-Skill` of a registrant's profile as two enumerations: `ExpertiseType`, `ProgrammingLanguageType` and `NaturalLanguageType`. Such enumerations/classifications can be easily extended for different purposes. `rankAtPlatform` is an attribute of `Profile` that represents a rank of a registrant maintained by the platform. A platform provides a mechanism to rank a registrant according to her/his performance, which is usually evaluated by task submitters, and other registrants who worked with this particular registrant. `Experience` of a registrant is calculated based on the types of tasks that the registrant has completed via the platform. `averagePaymentPer-Task` is derived from the information

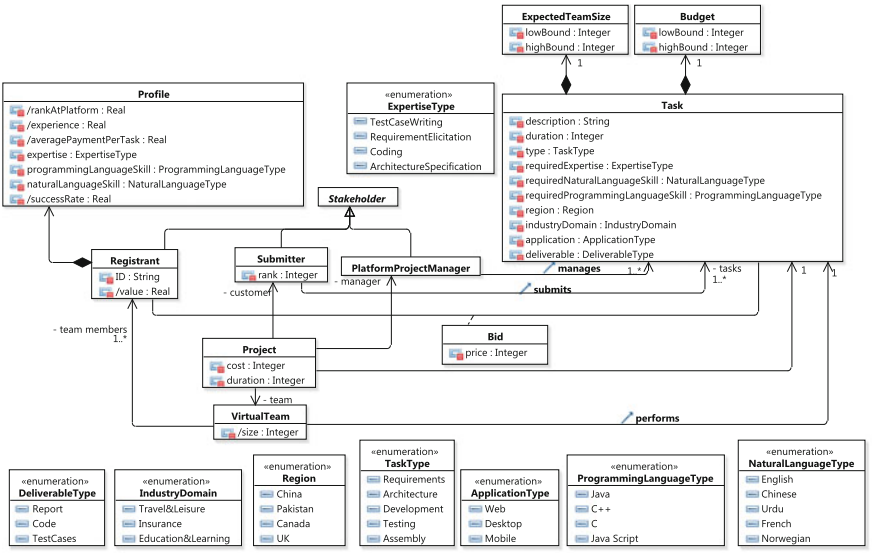


Fig. 2 General conceptual model

maintained by the platform as it has records all the payments that have been done for the registrant. `successRate` is another factor that should be accounted for when evaluating a registrant. It represents the rate of the successful task delivered via the platform.

Another important concept is `Task`, which captures the task `Description`, `Duration`, etc. We defined an enumeration `TaskType` to classify different types of tasks including `Requirements`, `Architecture`, `Development`, `Testing` and `Assembly`, which can be easily extended when needed. When a task submitter submits a task, as part of the specification of the task, she/he has to also define `requiredExpertise`, `requiredNaturalLanguageSkill`, `requiredProgrammingLanguageSkill`, which are classified using the enumerations (`ExpertiseType`, `ProgrammingLanguageType` and `NaturalLanguageType`) also referenced by three attributes of a registrant's profile. Therefore, it is easy to understand that ideally an optimal solution should match required expertise and skills of a task and profiles of the virtual team members. Besides these six attributes of class Task, we also capture `Region`, `IndustryDomain`, `ApplicationType` and `DeliverableType`, which are useful information needed to form a virtual team to finish a task. Two other important pieces of information that are associated to a task are `ExpectedTeamSize` and `Budget`, which are provided by a task submitter while submitting a task, which are constraints in terms of forming a virtual team. The size of a formed virtual team should be within the range of the expected team size of a submitted task and the budget should be sufficient to pay the virtual team members.

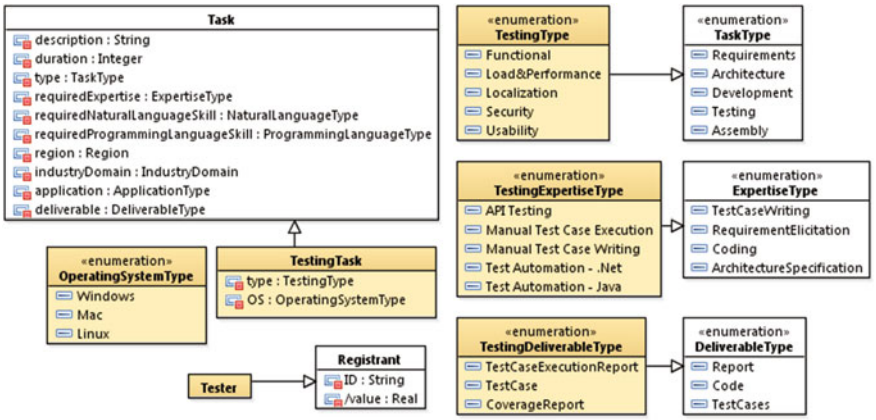


Fig. 3 Conceptual model for testing

After a virtual team is formed, a project is then created in the platform. Such a project should define the real cost to complete a task and time required to finish the task. Such information is used by a fitness function used by search-based algorithms to find an optimal solution. `Bid` plays an important role in most of existing crowdsourcing platforms such as TopCoder by linking a registrant to a specific task. Bids submitted by registrants for a task are used to check whether the cost to form a virtual team (i.e., the sum of the bids submitted by the virtual team members) is within the budget range of the task submitter.

Note that the conceptual model presented in Fig. 2 is generic and therefore not targeting to any specific type of tasks. However, since it is generic, we show an example (Fig. 2) how it can be extended for conducting a specific task. In the context of Fig. 2, our general conceptual model is extended for capturing concepts for conducting testing tasks. For example, crowdsourcing platform UTest is a platform focusing on testing tasks only. From Fig. 3, one can notice that for testing, four new enumerations are defined to extend four enumerations defined in Fig. 2. By doing so, we can extend the generic platform into a specialized one by introducing more specialized information such as `TestingType`. Besides introducing additional enumerations, we also define two next concepts (i.e., `TestingTask` and `Tester`) to extend generic concepts Task and Registrant.

4 Search-Based Crowdsourcing Methodologies

According to the comprehensive review of Harman et al. [11], Search Based Software Engineering (SBSE) has been extensively investigated to address various software engineering problems spanning from requirements, testing to reengineering of a typical software development lifecycle. Particularly for requirements, SBSE has been applied for various optimization problems such as requirements selection

[6], prioritization [7], and assignment [10, 14], with different objectives such as maximizing customers'/stakeholders' satisfaction, maximizing benefits/value and minimizing cost. For testing SBSE has been applied to successfully address test generation and test optimization problems [1, 11]. To the best of our knowledge, SBSE has never been applied to address optimization issues existing in crowdsourcing. In this section, we mainly present a pilot study we recently conducted to demonstrate that SBSE can also be applied for addressing optimization issues in crowdsourcing and it is promising in a future to integrate such as an optimization methodology as part of crowdsourcing platforms.

The main challenge in proposing a SBSE solution is to propose and assess a fitness function for the intended optimization problem. In the rest of the section, we propose the fitness function for our crowdsourcing problem and evaluate the fitness function in conjunction with the following search algorithms, i.e., Genetic Algorithms (GAs), (1+1) Evolutionary Algorithm (EA), Alternating Variable Method (AVM). Random Search (RS) was used as the baseline to evaluate the performance of these algorithms.

4.1 Description on Selected Search Algorithms

The most common search algorithms that have been employed for SBSE are evolutionary algorithms, simulated annealing, hill climbing (HC), ant colony optimization, and particle swarm optimization [8]. Among these algorithms, HC is a simpler, local search algorithm. The SBSE techniques using more complex, global search algorithms are often compared with the techniques based on HC and random search to determine whether their complexity is warranted to address a specific SBSE problem. The use of the more complex search algorithm may only be justified if it performs significantly better than, for instance, random search.

To use a search algorithm, a fitness function needs to be defined. The fitness function should be able to evaluate the quality of a candidate solution (i.e., an element in the search space). The fitness function is problem dependent, and proper care needs to be taken for developing adequate fitness functions. The fitness function will be used to guide the search toward fitter solutions.

Below, we provide a brief description of the search algorithms that we used in the pilot study and we will investigate more algorithms in the future. AVM was selected as a representative of local search algorithms. GA was selected since it is the most commonly used global search algorithm in SBSE [1]. (1+1) EA is simpler than GAs, but in previous software testing work we found that it can be more effective in some cases (e.g., see [3]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [1].

4.1.1 Genetic Algorithms

Genetic Algorithms (GAs) are the most well-known [1] and are inspired by the Darwinian evolution theory. A population of individuals (i.e., candidate solutions) is evolved through a series of generations, where reproducing individuals evolve through crossover and mutation operators. GAs are the most commonly used algorithms and hence we do not provide further details; however an interested reader may consult the following reference for more details [13].

4.1.2 (1+1) Evolutionary Algorithm

(1+1) Evolutionary Algorithm (EA) [9] is simpler than GAs. In (1+1) EA, popu-lation size is one, i.e., we have only one individual in the population and the individual is represented as a bit string. As opposed to GAs, we do not use the crossover operator but only rely on a bitwise mutation operator for exploring the search space. To produce an offspring, this operator independently flips each bit in the bit string with a probability (p) based on the length of the string. If the fitness of the child is better than that of the parent (bit string of the child before mutation), the child is retained for the next generation.

4.1.3 Alternating Variable Method

Alternating Variable Method (AVM) is a local search algorithm first introduced by Korel [12]. The algorithm works in the following way: Suppose we have a set of variables $\{v_1, v_2, v_n\}$, we then try to maximize fitness of v_1, while keeping the rest of the variables constant, which are generated randomly. The search is stopped if a solution is found; otherwise, if the solution is not found, but we found a minimum with respect to v_1, we switch to the second variable. Now, we fix v_1 at the found minimum value and try to minimize v_2, while keeping the rest of the variables constant. The search continues in this way, until we find a solution or we have explored all the variables.

4.2 Problem Representation and Fitness Function

Our goal is to form a virtual team to complete a task within the budget specified for a task by a submitter by accounting for an optimal matching between required expertise, skills, and other relevant information with the team members' experience, expertise, skills and bids. The ultimate objective is to form a virtual team that should work like in a real world in the sense that the real values of the virtual team should be appreciated (reflected as payment received for the task) and deliver high quality deliverables in a productive way. We believe, by doing so, the overall quality and

productivity of software development via crowdsourcing platforms would be improved. Moreover, we also expect this philosophy of forming a virtual team would be very useful especially in the context of practicing large-scale software development via crowdsourcing, as conducting large-scale software development tasks is not anymore one person task. Teamwork in crowd should be taken into account for managing and conducting this kind of tasks. The problem is more complicated if we account for scheduling and dependencies among sub-tasks that are required to be completed by more than one persons. To this end, a scalable, systematic task scheduling, virtual team formation is a very important issue to tackle. In this paper, we make a first step towards this direction.

Suppose we have a Crowd C with a set of m registrants $C = \{r_1, r_2, \ldots, r_m\}$. A task submitter submits a task by defining the budget range: $Budget_{min}$ and $Budget_{max}$ and team size range n: $TeamSize_{min}$ and $TeamSize_{max}$.

$$TeamSize_{min} \leq n \leq TeamSize_{max} \tag{1}$$

A solution would be a virtual team of registrants who bided for the task: $V = \{r_1, r_2, r_n\}$, where $n \leq m$. Registrant i has a property defining his/her value $Value_i$, which is calculated based on the four factors: $SuccessfulRating(0-1)$, $CustomerRating(0-1)$, $Experience(0-1)$, and $PaymentHistory$ (average payment/task in the past).

$$Value_i = \frac{SuccessfulRating + CustomerRating + Experience + PaymentHistory}{4} \tag{2}$$

Notice that all these four values are normalized between 0 and 1. In the above formula, we take average of all these four values and the resultant $value_i$ will be again between 0 and 1.

Each registrant i provides $RBid_i$ to complete the task. To form an optimal virtual team, the solution must satisfy the following requirements: (1) Budget and team size requirements; (2) A solution must provide a bid values for all registrants as much as closer to their requested bids ($f_{bidGap}(n)$); (3) Each registrant in a virtual team must be assigned a bid which is fair according to his/her experience, ratings, and payment history ($f_{similarity}(n)$).

For optimization problem, our optimization parameter is Bid_i corresponding to registrant i. For the first requirement, for each registrant in a virtual team, a search algorithm finds a bid value for each registrant ($RBid_i \leq Bid_i \leq RBid_i$) such that:

$$Budget_{min} \leq \Sigma_{i=1}^{n} Bid_i \leq Budget_{max} \tag{3}$$

For the second requirement, we calculate $f_{bidGap}(n)$, whose formula is shown below, where we try to make the Bid_i as close as possible to $RBid_i$ for each registrant.

$$f_{bidGap}(n) = \frac{\Sigma_{i=1}^{n} nor(|RBid_i - Bid_i|)}{n} \tag{4}$$

In the formula below, $nor()$ is a normalization function, which is calculated as $nor(x) = x/(x+1)$. We adopted this normalization function from the literature and has proven to be more robust than other normalization functions in the context of search-based software engineering [2, 3].

For the third requirement, we calculate $f_{similarity}(n)$, which is calculated by the following formula:

$$f_{similarity}(n) = \frac{\Sigma_{i=1}^{n} nor(|Value_i * Budget_{max} - Bid_i|)}{n} \tag{5}$$

Based on the above requirements, our fitness function can be formulated as below:

$$f_{Fitness}(n) = \frac{(f_{similarity}(n) + xor(f_{bmax}(n), f_{bmin}(n)) + f_{bidGap}(n))}{3} \tag{6}$$

where f_{bmax} and f_{bmin} are defined as follows:

$$f_{bmax}(n) = \begin{cases} 0, & \Sigma_{i=1}^{n} Bid_i - Budget_{max} \leq 0 \\ nor(\Sigma_{i=1}^{n} Bid_i - Budget_{max}), & \Sigma_{i=1}^{n} Bid_i - Budget_{max} > 0 \end{cases} \tag{7}$$

$$f_{bmin}(n) = \begin{cases} 0, & Budget_{max} - \Sigma_{i=1}^{n} Bid_i \leq 0 \\ nor(Budget_{max} - \Sigma_{i=1}^{n} Bid_i), & Budget_{max} - \Sigma_{i=1}^{n} Bid_i > 0 \end{cases} \tag{8}$$

4.3 Empirical Evaluation

This section discusses the experiment design, execution, and analysis of the evaluation of the fitness function with the four search algorithms for addressing our optimization problem.

4.3.1 Experiment Design

The objective of our experiments is to evaluate proposed fitness function in conjunction with the selected search algorithms in terms of solving our optimization problem: Find a virtual team (v) of size n from crowd C of size m registrants, such that v meets all budget and team size requirements of a project, each registrant must obtain a bid that is closer to what was requested, and each registrant must obtain a bid value that matches his/her ratings, experience, and payment history.

4.3.2 Research Questions

In these experiments, we address the following research question:

RQ1: Are the search algorithms effective to solve our optimization problem, to compare with RS?

RQ2: Among AVM, (1+1) EA and GA, which one fares best in solving our optimization problem?

4.3.3 Selection Criteria of Search Algorithms and Parameter Settings

In our experiments, we compared four search algorithms: AVM, GA, (1+1) EA, and RS (Sect. 4.1). AVM was selected as a representative of local search algorithms. GA was selected since it is the most commonly used global search algorithm in search-based software engineering [1]. We selected steady state GA with a population size of 100 and a crossover rate of 0.75, with a 1.5 bias for rank selection. We used a standard one-point crossover, and mutation of a variable is done with the standard probability 1/n, where n is the number of variables. Different settings would lead to different performance of a search algorithm, but standard settings usually perform well [5]. (1+1) EA is simpler than GAs, but in previous software testing work we found that it can be more effective in some cases (e.g., [3]). We used RS as the comparison baseline to assess the difficulty of the addressed problem [1].

4.3.4 Artificial Problems Design

In addition, to empirically evaluate whether the fitness function defined in Sect. 4.2 really address our optimization problem, we created artificial problems inspired from famous crowdsourcing platforms such as TopCode and uTest. Topcoder has 480,000 software developers, algorithmists, and digital designers, whereas Utest has 60,000 testers. Keeping this information, we created a crowd C of size 60,000 for our pilot study. For each bidder in the crowd, we assigned random values for the four parameters: $SuccessfulRating$, $CustomerRating$, $Experience$, and $PaymentHistory$. Each value ranges from 0 to 1.

After populating the crowd, we created projects with various characteristics. In total, we created 6000 projects. The budget for each project ranged from 100USD C 10000USD with the increment of 100USD. Since each project can have a minimum and maximum budget value, the minimum value was set to 10 % less of the given project budget and the maximum value was set to 10 % more of the given project budget. For example, if the given project is 100USD, then the minimum budget would be 90 and maximum budget would be 110. For each project, we set a series of number of team sizes, which are as: 2–4, 5–7, 8–10, 11–13, 14–16.

For each project, we set the number of bidders into the following three classes:

- Low (20, 50, 80, 100)
- Medium (200, 300, 400, 500)
- High (1000, 2000, 3000, 4000)

For each bidder, we randomly generate a value for $RBid$ from 0 to $\frac{Budget_{max}}{teamSize_{max}}$ of a project and to make $RBid$ fair based on the four parameters, i.e., $SuccessfulRating$, $CustomerRating$, $Experience$, and $PaymentHistory$, we modified the generated $RBid$ as follows:

$$\frac{SuccessfulRating + CustomerRating + Experience + PaymentHistory}{4} * \frac{Budget_{max}}{teamSize_{max}} \tag{9}$$

Moreover, we restricted search algorithms to generate a bid value ranging from $0.5RBid$–$1.5RBid$. The purpose was to avoid generating unrealistic bids values.

4.3.5 Statistical Tests

To compare the obtained results of the four search algorithms, the Kruskal-Wallis test, the Wilcoxon signed-rank test and the Vargha and Delaney statistics are used, based on the guidelines for reporting statistical tests for randomized algorithms presented in [3, 8].

To check if there are significant differences across the four algorithms, we first performed the KruskalCWallis test. Obtained p-value indicates whether there is significant difference among the four algorithms. However this test does not tell us which algorithm is significantly different with which algorithm. Therefore, we further performed the Wilcoxon signed-rank test to calculate a p-value for deciding whether there is a significant difference between a pair of search algorithms. We chose the significance level of 0.05, which means there is a significant difference if a p-value is less than 0.05.

As investigated in [3], it is not sufficient to interpret results only using p-values. To better interpret the results, the statistical test results must be interpreted in conjunction with an effect size measure, which helps determining practical significance of the results. We used the Vargha and Delaney statistics ($\widehat{A}_{12}$) to calculate the effect size measure, which is selected based on the guidelines proposed in [3]. In our context, given the fitness function FS ($f_{Fitness}(n)$), $\widehat{A}_{12}$ is used to compare the probability of yielding highest fitness value (low FS value) for two algorithms A and B. If $\widehat{A}_{12}$ is equal to 0.5, the two algorithms are equivalent. If $\widehat{A}_{12}$ is more than 0.5, it means the first algorithm A has higher chances of obtaining higher fitness value than B.

4.3.6 Experiment Execution

For each of the 100 artificial problems, we ran experiments 100 times for each of the four search algorithms for each problem. We let all the four algorithms run up to 2000 generations for each problem and collected final fitness value calculated in the 2000th generation. We used a PC with Intel Core Duo CPU 2.20 GHz with 4 GB of RAM, running Linux Ubuntu operating system for the execution of experiment.

4.3.7 Results and Analysis

To answer our research questions, we compared the three search algorithms with RS based on mean fitness values achieved after 2000 generations for each algorithm and each of the 100 problems. Recall that each problem was repeated for 100 times to account for random variation.

Table 1 provides the Vargha and Delaney statistics. The column $A > B$ means the number of problems out of 100 for which an algorithm A has higher chances of obtaining higher fitness value than B, $A < B$ means vice versa, and $A = B$ means the number of problems for which there were no differences between A and B as $\widehat{A}_{12}$ equals to 0.5.

Table 2 summarizes results of the Wilcoxon signed-rank test for RQ1 and RQ2. The column $A > B$ means the number of problems out of 100 for which an algorithm A was significantly better than B, $A < B$ means vice versa, and $A = B$ means the number of problems for which there were no significant differences between A and B based on p-values calculated by the Wilcoxon test.

Results for RQ1

To answer RQ1, we compared each search algorithm with RS, based on the mean fitness values of 100 runs obtained for each problem. Results for RQ1 are shown in the first three rows of Tables 1 and 2.

AVM versus RS: As we can see in Table 1, AVM performed better than RS for 4045 problems but for 1889 problems the results were statistically significant (Table 2).

Table 1 Results for the Vargha and Delaney $\widehat{A}_{12}$ statistics

	Pair of Algorithms (A vs. B)	$A > B$	$A < B$	$A = B$
RQ1	AVM versus RS	4045	1953	2
	(1+1)EA versus RS	2847	3149	4
	GA versus RS	2883	3115	2
RQ2	AVM vs (1+1)EA	4104	1892	4
	AVM versus GA	4115	1880	5
	(1+1)EA versus GA	2873	3127	0

Table 2 Results for the Wilcoxon signed-rank test at significance level of 0.05-artificial problems

	Pair of algorithms (A vs. B)	$A > B$	$A < B$	$A = B$
RQ1	AVM versus RS	1889	28	4083
	(1+1)EA versus RS	143	186	5671
	GA versus RS	151	150	5699
RQ2	AVM vs (1+1)EA	1931	14	4055
	AVM versus GA	1881	19	4100
	(1+1)EA versus GA	156	158	5686

RS performed better for 1953 problems as shown in the first row of Table 1, and there were no significant differences for 4083 problems.

(1+1) EA versus RS: (1+1) EA performed better than RS for 2847 problems (Table 1), 143 problems out of which were significantly better than RS (Table 2). There were no significant differences for 5671 problems (Table 2).

GA versus RS: In case of GA, it performed better than RS for 2883 problems (Table 1). Out of 2883, for 151 problems GA was significantly better than RS (Table 2). For 5699 problems there were no significant differences (Table 2).

Concluding Remarks: Based on the above results, we can answer RQ1 as follows: AVM is significantly better than RS for finding an optimal solution for our problem. For other two algorithms (GA and (1+1) EA), we didn't observe significant differences than RS.

Results for RQ2

The results to answer RQ2 are presented in the last three rows of Tables 1 and 2. These results are also based on the mean fitness values obtained for each problem for each algorithm after running the problem 100 times.

AVM versus (1+1) EA: AVM performed better than (1+1) EA for 4104 problems (Table 1), but for 1931 problems it was significantly better than (1+1) EA (Table 2). AVM performed worse than (1+1) EA for 1892 problems (Table 2) and in 14 out of these 1892 problems (1+1) EA was significantly better than AVM (Table 2). There were no significant differences between the algorithms for 4055 problems as shown in Table 2.

AVM versus GA: AVM performed better than GA for 4115 problems (Table 1) and out of these 4115 problems AVM performed significantly better than GA for 1881 problems (Table 2). AVM performed significantly worse than AVM for 19 problems (Table 2).

(1+1) EA versus GA: Regarding the (1+1) EA versus GA pair, as we can see from Table 2 that (1+1) EA was significantly better than GA for 156 problems, whereas GA was significantly better than (1+1) EA for 158 problems. For the rest of the problems, there were no significant differences.

Concluding Remarks: Based on the above results, we can answer RQ2 as follows: AVM is the best algorithm in terms of finding an optimal solution in our context and

the rest of the algorithms performed worse than AVM and there were no significant difference between the performance of the three algorithms.

Discussion
In this section, we provide an overall discussion based on the results of the experiments.

We observed from the results that AVM is significantly better than RS in finding an optimal solution (RQ1) and for the rest of the algorithms there are no significant differences than RS. Among all the studied algorithms, AVM is significantly better than the rest of the algorithms (RQ2).

The performance of algorithms can be argued based on their working. AVM works is a local search algorithm. If the fitness function provides a clear gradient towards the global optima, then AVM will quickly converge to one of them, which might be the case for our current context. On the other hand, (1+1) EA puts more focus on the exploration of the search landscape. When there is a clear gradient toward global optima, (1+1) EA is still able to reach those optima in reasonable time, but will spend some time in exploring other areas of the search space. This latter property becomes essential in difficult landscapes where there are many local optima. In these cases, AVM gets stuck and has to restart from other points in the search landscape. On the other hand, (1+1) EA, thanks to its mutation operator, has always a non-zero probability of escaping from local optima. Similar is the case for GA, which tries to explore (mutation operator) and exploit (crossover) the search space and hence may require more generations. By increasing the number of generations, we expect that the performance of GA and (1+1) EA can be improved.

Based on the above results, we can conclude that in our current context AVM has the ability to solve a wide range of problems. However, more experiments are needed in the future to thoroughly evaluate our fitness function with the real data from crowdsourcing platforms.

5 Threats to Validity

To reduce construct validity threats, we chose an effectiveness measure called fitness value, which is comparable across all four search algorithms (AVM, (1+1) EA, GA and RS. Furthermore, we used the same stopping criterion for all algorithms, i.e., number of generations. This criterion is a comparable measure of efficiency across all the algorithms.

The most probable conclusion validity threat in experiments involving randomized algorithms is due to random variations. To address it, we repeated experiments 100 times to reduce the possibility that the results were obtained by chance. Furthermore, we performed the Wilcoxon test to compare the algorithms mean fitness values of 100 runs and determine the statistical significance of the results. We chose this test since it is appropriate for the continuous data [4], thus matching our situation. To determine the practical significance of the results obtained, we measured the effect

size using the $\widehat{A}_{12}$ values, which is recommended to be used in conjunction with the Wilcoxon test to better interpret the results [3].

A possible threat to internal validity is that we have experimented with only one configuration setting for the GA parameters. However, these settings are in accordance with the common guidelines in the literature and our previous experience on testing problems. Parameter tuning can improve the performance of GAs, although default parameters often provide reasonable results [5].

One common external validity threat in the software engineering experiments is about generalization of results. To deal with this, we conducted an empirical evaluation of our proposed fitness function using 6000 artificial problems of varying complexity.

6 Conclusion and Future Work

To compare with traditional software engineering development, crowdsourcing software engineering, especially for developing large-scale software, is still far away from being mature. In this paper, we propose a search-based approach to make a very first step toward this direction by providing an automated, scalable and intelligent solution to assist platform managers to find an optimal solution when forming a virtual team for a submitted task via a crowdsourcing platform. We conducted a pilot study and results show that AVM is a promising search algorithm, together with the defined fitness function, can efficiently find an optimal solution for our problems. In the future, we plan to conduct more experiments based on real data that can be collected from existing crowdsourcing platforms such as Topcoder and UTest. We also plan to provide an integrated solution starting from specifying tasks and profiles of registrants, automatically collecting data for search, until providing feedback to end users such as registrants, platform managers and submitters in a transparent manner. By doing so, we hope, in certain extent, we can improve the quality and productivityranking, repuation and reward system of the current practice of performing soft-ware engineering tasks via crowdsourcing platforms.

References

1. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. **36**(6), 742–762 (2010)
2. Arcuri, A.: It does matter how you normalise the branch distance in search based software testing. In: 2010 Third International Conference on Software Testing, Verification and Validation (ICST), pp. 205–214. IEEE (2010)
3. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Softw. Test. Verif. and Reliab. **23**(2), 119–147 (2013)

4. Arcuri, A., Briand, L.: A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 1–10. IEEE (2011)
5. Arcuri, A., Fraser, G.: Search based software engineering. On Parameter Tuning in Search Based Software Engineering, pp. 33–47. Springer, New York (2011)
6. Bagnall, A.J., Rayward-Smith, V.J., Whittley, I.M.: The next release problem. Info. Softw. Technol. **43**(14), 883–890 (2001)
7. Baker, P., Harman, M., Steinhofel, K., Skaliotis, A.: Search based approaches to component selection and prioritization for the next release problem. In: 22nd IEEE International Conference on Software Maintenance. ICSM'06. pp. 176–185. IEEE (2006)
8. Burke, E.K., Kendall, G.: Search Methodologies. Springer, New York (2005)
9. Droste, S., Jansen, T., Wegener, I.: On the analysis of the (1+ 1) evolutionary algorithm. Theor. Comput. Sci. **276**(1), 51–81 (2002)
10. Finkelstein, A., Harman, M., Mansouri, S.A., Ren, J., Zhang, Y.: A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making. Requir. Eng. **14**(4), 231–245 (2009)
11. Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: a comprehensive analysis and review of trends techniques and applications. Technical Report Department of Computer Science, Kings College London, TR-09-03 (2009)
12. Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. **16**(8), 870–879 (1990)
13. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004)
14. Yue, T., Ali, S.: Applying search algorithms for optimizing stakeholders familiarity and balancing workload in requirements assignment. In: Proceedings of the 2014 Conference on Genetic and Evolutionary Computation, pp. 1295–1302. ACM, New York (2014)

Collaborative Majority Vote: Improving Result Quality in Crowdsourcing Marketplaces

Dennis Nordheimer, Khrystyna Nordheimer, Martin Schader and Axel Korthaus

Abstract Crowdsourcing markets, such as Amazon's Mechanical Turk, are designed for easy distribution of micro-tasks to an on-demand scalable workforce. Improving the quality of the submitted results is still one of the main challenges for quality control management in these markets. Although beneficial effects of synchronous collaboration on the quality of work are well-established in other domains, interaction and collaboration mechanisms are not yet supported by most crowdsourcing platforms, and thus, not considered as a means of ensuring high-quality processing of tasks. In this paper, we address this challenge and present a new method that extends majority vote, one of the most widely used quality assurance mechanisms, enabling workers to interact and communicate during task execution. We illustrate how to apply this method to the basic scenarios of task execution and present the enabling technology for the proposed real-time collaborative extension. We summarize its positive impacts on the quality of results and discuss its limitations.

1 Introduction

Over the past few years, "Crowdsourcing" has evolved from a side issue to a successful business model in the modern online world. The basic idea behind crowdsourcing lies in outsourcing tasks to a large and undefined group of workers over the Internet

D. Nordheimer · K. Nordheimer (✉) · M. Schader
Chair in Information Systems III, University of Mannheim, Schloss,
68131 Mannheim, Germany
e-mail: nordheimer@uni-mannheim.de

D. Nordheimer
e-mail: nordheimer@wifo.uni-mannheim.de

M. Schader
e-mail: martin.schader@uni-mannheim.de

A. Korthaus
College of Business, Victoria University International,
PO Box 14428, Melbourne, VIC 8001, Australia
e-mail: axel.korthaus@vu.edu.au

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_8

in the form of an open call [2]. Today, crowdsourcing is widely used in areas like open innovation [1], competition markets [11], or collaborative knowledge creation. By combining capabilities of humans and computers, it provides a powerful way to solve complex problems that neither of them can solve alone [17].

In our paper, we focus on commercial crowdsourcing platforms like Amazon's Mechanical Turk (MTurk) that are designed for solving micro-tasks, such as data categorization, image labeling, or product classification. These small tasks are usually posted by requesters to a scalable workforce and can be done by any worker who is allowed to perform the work. Because of the open nature of the crowdsourcing platforms and the limited control over the workforce, the main challenge for quality control management is to guarantee reliable results.

Until now, several quality assurance approaches have been developed for specific needs of crowdsourcing markets. One of the most widely used mechanisms is majority voting [20, 21]. This method aggregates multiple results provided by different workers on the same task in order to derive a single correct result or the result with the highest probability of correctness [3, 6]. Obviously, majority vote is suited only for quality control of a few task types that have a limited answer space and one objectively correct solution. Another drawback is that, if the result is not of the desired quality, the system will cause higher costs by assigning the same task to other workers. However, the main weakness of majority vote lies in the fact that good workers can be discouraged since correctly solved tasks can be voted down by false results. By analyzing how workers on crowdsourcing platforms perceive different quality assurance mechanisms, Schulze et al. found out that majority vote is perceived as being unfair and should not be the criterion to decide whether workers are paid or not [19].

Another commonly used quality assurance mechanism is peer review, also known as validating review. This approach leverages a reviewer or a group of reviewers verifying the submitted results [5]. In contrast to majority vote, this method can be applied for quality control of non-deterministic tasks where different answers can be considered valid, e.g., language translation or content creation. However, employing reviewers who are usually more experienced workers causes more costs and requires an additional effort in quality management. Since, again, false reviews can vote down correct answers, peer review can demotivate good workers to perform such tasks [19].

There are several other approaches which are common practice for quality management on crowdsourcing marketplaces: Qualification tests or qualification restrictions can be applied to limit the worker pool to work on certain task types [4]. Gold standard data sets (tasks with known answers) can be mixed into the stream of regular tasks to establish the overall quality of results submitted by a worker [16]. Improving peer reviews are used not only to provide a rating whether a result should be accepted or rejected, but also to improve it [5].

To the best of our knowledge, all quality assurance mechanisms for micro-tasks do not consider the collaborative aspects of working processes as a means for improving worker quality [8, 9]. Statistical quality control techniques (like dynamic majority vote) even explicitly assume that workers must work independently and do not collaborate with each other [7]. On the other hand, collaboration has for a long

time been known to be more beneficial than non-collaborative work because of providing stronger socialization, better worker satisfaction and motivation, as well as higher quality of produced results [8, 13, 14]. It also can assist to overcome different viewpoints and skills of workers with different backgrounds as well as to reduce understanding and interpretation issues regarding task description or task execution. Finally, newer software technologies afford a new means toward the introduction and the research of collaborative environments on crowdsourcing marketplaces.

Considering the above-mentioned challenges of quality control techniques and the positive effects of collaboration on workers' result quality, we propose to extend the conventional quality assurance methods by providing collaborative capabilities to working processes in crowdsourcing marketplaces. To simplify the solution and to limit the scenarios where collaboration could be used, for this paper, we only focus on majority vote as a method for quality control of two main types of tasks: deterministic tasks (e.g., data categorization or image labeling) and non-deterministic tasks (e.g., content creation or language translation). As a first step in exploring the influence of worker collaboration and interaction on the quality of the submitted results, we want to answer the following research questions:

- How the traditional processing of tasks can be enhanced by collaborative capabilities?
- What are the application scenarios for solving micro-tasks in a collaborative environment?

To answer these questions, we describe a new method for quality management on crowdsourcing marketplaces, called *Collaborative Majority Vote (CMV)*, which extends the traditional majority vote by enabling a certain number of workers to simultaneously work as a group on the same task.

2 Related Work

With the emergence of crowdsourcing and its recognition as a successful business model, a large research agenda has emerged in quality management focusing on finding the best suited methods to ensure high quality results. Depending on specific task types, recent efforts have taken several directions including manual verification of results, use of redundancy to aggregate correct answers, or identification of spammers and their exclusion from further task execution. However, existing quality assurance approaches do not consider real-time collaboration between workers as a means of improving the quality of results. Since the collaborative aspects of task execution and their implications on overall quality have not yet been sufficiently studied in the context of crowdsourcing marketplaces, we first analyze the main advantages and limitations of collaboration that have already been established in other domains and, then, project them into the context of quality control in crowdsourcing markets.

In the area of collaborative writing, McCarthy et al. found out that the synchronous collaboration on a document produces more natural responses than asynchronous

collaborative work [14]. Synchronous collaboration also helps to resolve misunderstandings and differing assumptions between individuals with regard to specific task settings. Comparing synchronous and asynchronous collaborative writing, Lowry et al. stated higher quality of produced results at greater efficiency [13]. The authors also reported that collaborative work provides stronger socialization, communication, and higher worker satisfaction.

Examining exploratory web search as a collaborative activity, Morris discovered that providing explicit support for collaboration leads to better coverage of the relevant information space, higher worker confidence in the completeness and correctness of search results, and increases the productivity of the search process [15].

Although collaboration during work execution is the norm rather than the exception and its positive impacts on the quality of results are well-established, interactions between workers in most of the common crowdsourcing platforms are not supported and collaboration mechanisms are usually absent [8]. As a result, only sporadic research is dedicated to the analysis of collaboration aspects in the crowdsourcing area. For measuring the crowdsourcing performance of MTurk, Kosinski et al. assume the independency of task execution and point out that their results may not hold by taking collaborative aspects into consideration and that further research in this direction is needed [9].

Little et al. explored parallel and iterative human computation processes on MTurk and compared the quality of the received results [12]. While in the first case, the workers performed their work independently of each other, in the second case, the work was iteratively based on the others' work outcomes. The experiment showed that simply applying iterative processes to work execution already increases the overall quality of results. In order to analyze potential benefits of collaboration, Kittur conducted an experiment for the collaborative translation of a poem, in which workers had the possibility to interact during task execution [8]. As enabling technology the author used an open-source platform Etherpad that allows users to synchronously edit the text, marks a worker's contribution in a different color, and supports real-time communication among workers. To evaluate the quality of the received results, 16 bilingual workers on MTurk were asked to rate the crowdsourced translation by comparing it to the original poem translation. As result, 14 of 16 raters preferred the crowdsourced version to the original translation. Regarding the perception of the collaborative process, workers found it enjoyable and rewarding to interact, communicate with each other, and monitor contributions of participating workers. The experiment shows that letting workers interact and collaborate while solving the crowdsourced tasks can significantly improve the quality of results, the overall performance, and their motivation.

In summary, the above-mentioned studies give good reasons and motivation for analyzing the applicability of collaboration to solving micro-tasks and studying its impacts on the quality of results.

3 Conceptual Model

This section describes the CMV approach, which is the main contribution of our paper. The method enhances the traditional majority voting procedure with collaborative functionality in order to improve the quality of results without causing any additional effort in quality management. After the assumptions related to task types, worker pool, and platform settings are clarified in Sect. 3.1, the concept of CMV is established by Sect. 3.2. Subsequently, the applicability of CMV for solving deterministic and non-deterministic tasks is described in Sects. 3.3 and 3.4.

3.1 Assumptions

For the basic scenarios in our model, we consider commercial crowdsourcing marketplaces like MTurk that are designed for solving micro-tasks such as product classification, data validation, or picture annotation. On these platforms, requesters (e.g., businesses or developers) outsource tasks to a scalable group of workers. They publish so-called Human Intelligence Tasks, denoted by $HITs = \{h_1, h_2, \ldots, h_m\}$, a set of m similar tasks of the same task type. Each $h \in HITs$ can be flexibly selected and solved by any worker from the worker pool $W = \{w_1, w_2, \ldots, w_n\}$ that is allowed to perform the work. The results are then submitted back to the platform in return for a small compensation per completed h. In case of deterministic tasks, only one result is assumed to be correct. The answer set can be predefined and provided to the workers or not. In contrast, for non-deterministic tasks multiple answers can be considered valid. At this point, note that in our approach we are going to relax the restriction of applicability of majority vote only to deterministic task types.

3.2 Collaborative Majority Vote

The traditional majority vote consists of three steps: assignment of the same task to multiple workers, solving the task, and deduction of a single result on the basis of the individual results submitted by the workers. Obviously, in the second step the workers work independently on their solution for the given task and submit this back to the platform. In the third step, the system alone is responsible for deriving valid results. With regard to these aspects, our method implies some significant modifications.

Compared to traditional majority vote, the process flow in our model is covered by four main phases according to the desired activities. In the first phase, a predefined number of workers from the worker pool W is assigned to the same task $h \in HITs$. This number of workers is often called redundancy level; it has a direct influence on the quality of results to be derived in the last phase. For simplicity, we set the redundancy level to 3 and do not consider any impacts of its changes on the result's

quality in our model. During the assignment process, various factors can be taken into account, e.g., mixing experienced and non-experienced workers or filtering workers according to additional task restrictions that could not be applied in qualification tests. The assignment and handling of available HITs is performed by the crowdsourcing platform.

In the second phase, the individual answers of assigned workers, e.g., w_1, w_2, w_3, are recorded. We denote this set of individual answers as $\mathbf{a}^*$. In our case with three workers, this set is equal to $\mathbf{a}^* = \{a^*_{w_1}, a^*_{w_2}, a^*_{w_3}\}$. Up to this point, the process is similar to the traditional procedure, and if applying the conventional majority vote at this stage, a single result can be simply derived by comparing the answers $a^*_{w_1}$, $a^*_{w_2}$ and $a^*_{w_3}$.

In CMV, we introduce an additional decision phase where the workers have the possibility to inspect the answers of other workers in the group, to communicate and to discuss in real-time, and to change or confirm their initial submissions. During this phase, each worker can observe the decision making process of the others. This approach requires lightweight software technologies that allow synchronization of browser contents between workers in real-time, and especially, with regard to co-scrolling and co-form filling. Furthermore, workers can collaborate by using the provided real-time chat that can be seen as a means to resolve misunderstandings, to adduce evidences, to share own skills, to supervise and educate new workers, or to clarify specific settings of a task. The enabling technology for this real-time collaborative extension and its technical realization is described in Sect. 4. Once collaboration was terminated, workers have to confirm or change their initial results, and thus, a new set of answers can be recorded. We denote this set of collaboration-based answers as $\mathbf{a}^{**}$, i.e., in our case $\mathbf{a}^{**} = \{a^{**}_{w_1}, a^{**}_{w_2}, a^{**}_{w_3}\}$. Now, a single result needs to be aggregated on the basis of $\mathbf{a}^{**}$ in some common way; then, the task can be considered solved.

According to the above-mentioned description of the process flow in CMV, we mark its four phases as follows: *Assignment*, *Individual Decision*, *Collaborative Decision*, and *Result Determination*. The overall design phase of CMV differs depending on the task types, as well as on the number of possible valid responses and is described below in Sects. 3.3 and 3.4.

3.3 Deterministic Tasks

Scenario I A simple scenario using CMV for quality control of deterministic tasks is shown in Fig. 1. As described above, the process starts with the assignment of a certain number of workers (here: $w_{k-1} \in W, w_k \in W, w_{k+1} \in W$) to an available task (here: $h_x \in HITs$) and can then successively be applied to each task in $HITs = \{h_1, h_2, \ldots, h_x, \ldots, h_m\}$. Afterwards, the workers in the group work on the task independently of each other and individually submit their primary decisions (here: $\mathbf{a}^* = \left\{a^*_{w_{k-1}}, a^*_{w_k}, a^*_{w_{k+1}}\right\}$) back to the system. As soon as all answers are received,

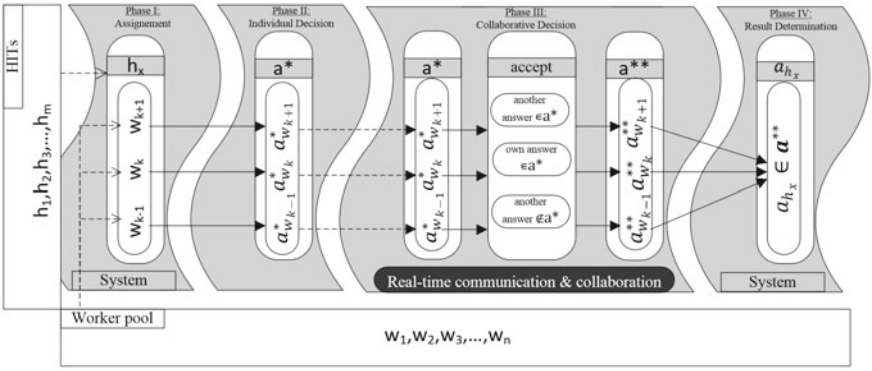


Fig. 1 Process flow of CMV for deterministic tasks

the workers are put in collaborative decision mode where they can see $\mathbf{a}^*$, the answers of all other workers. In the ideal case, where all of delivered answers are equal, i.e., $a^*_{w_{k-1}} = a^*_{w_k} = a^*_{w_{k+1}}$, workers obviously do not need to collaborate at all. Thus, they can quickly finish the task by confirming their initial answers, so that $\mathbf{a}^* = \mathbf{a}^{**}$. This step can be automatically executed by the system as well. In the last phase, the system derives a single result a_{h_x} for the task h_x by applying the traditional majority vote to the answer set $\mathbf{a}^{**}$.

Scenario II The real-time collaboration between workers during solving the same task in a group opens up methodically new perspectives for quality assurance in crowdsourcing marketplaces. Up to now, the traditional majority vote, in which a single result is derived based on the comparison or aggregation of multiple results, is exclusively performed by the system. This is because the workers, who provide these results, are not related to each other. In contrast, the CMV method offers the workers the opportunity to participate in this procedure. To facilitate this, the possibilities to complete the task in the collaborative mode have to be tightened by an additional restriction: After completing the collaboration, all replies must be identical, i.e., according to the notions in Scenario I, the collaboration-based answer set $\mathbf{a}^{**}$ must consist of only one item with $a^{**}_{w_{k-1}} = a^{**}_{w_k} = a^{**}_{w_{k+1}}$. In other words, the workers have to agree on the same answer and accept it as their own. Afterwards, the given task can be regarded as solved, and therefore, the system does not need to perform any additional computations in a last phase. At this point, it is important to mention that this scenario may lead to a more lively collaboration, and accordingly, result in a better impact on the quality of the submitted results. In case of disagreement with the majority, a worker can interrupt the task execution and a new worker can be assigned the group.

3.4 Non-deterministic Tasks

Scenario III The application of the CMV approach to non-deterministic task types differs from the previously described scenarios in the design of the collaborative mode as well as in the result set that can be considered valid. Figure 2 illustrates the overall process flow of the CMV for quality control of non-deterministic task types.

Similar to the steps in Scenario I, after assigning the task $h_x \in HITs$ to a group of workers and submitting the set of the primary replies workers have the possibility to inspect the answers of other workers, to discuss their decisions, and to change or confirm the initial submissions. Additionally, they now have the possibility of rating answers accepted by other workers in the group, and thus, of stating whether opposing results are valid or not. Identifying a reply as valid, the workers accept this answer as their own so that it can be added to their answer sets. After providing ratings, which are invisible to other workers, the set of collaboration-based answers $\mathbf{a}^{**} = \left\{\mathbf{a}^{**}_{\mathbf{w_{k-1}}}, \mathbf{a}^{**}_{\mathbf{w_k}}, \mathbf{a}^{**}_{\mathbf{w_{k+1}}}\right\}$ can be recorded and submitted to the system for further calculations in order to derive the final set of valid answers for the task h_x, i.e., to determine $\mathbf{a}_{\mathbf{h_x}} \subseteq \mathbf{a}^{**}$. In contrast to Scenario I, where the collaboration-based replies in the set $\mathbf{a}^{**}$ are single items, $\mathbf{a}^{**}_{\mathbf{w_{k-1}}}, \mathbf{a}^{**}_{\mathbf{w_k}}, \mathbf{a}^{**}_{\mathbf{w_{k+1}}}$ are subsets of answers, and therefore, marked in bold. This is also the case for $\mathbf{a}_{\mathbf{h_x}}$.

The next simple example explains the rating procedure in CMV for non-deterministic task types: Assigned to the same task $h_1 \in HITs$, the workers w_1, w_2, w_3 have already taken their collaboration-based decisions so that $\mathbf{a}^* = \mathbf{a}^{**} = \left\{\mathbf{a}^{**}_{\mathbf{w_1}}, \mathbf{a}^{**}_{\mathbf{w_2}}, \mathbf{a}^{**}_{\mathbf{w_3}}\right\}$ where $\mathbf{a}^{**}_{\mathbf{w_1}} = \{a_1\}$, $\mathbf{a}^{**}_{\mathbf{w_2}} = \{a_2\}$, and $\mathbf{a}^{**}_{\mathbf{w_3}} = \{a_3\}$, i.e., all workers have confirmed their answer. Based on implications of the communication, the worker w_1 states that a_2 can also be considered valid. This results in $\mathbf{a}^{**}_{\mathbf{w_1}} = \{a_1, a_2\}$. In contrast, worker w_2 identifies the other results as false. The worker w_3 considers a_1 to be correct so that $\mathbf{a}^{**}_{\mathbf{w_3}} = \{a_1, a_3\}$. Table 1 summarizes these rating results. In the next step, the system aggregates the set of valid results by calculating the best rated answers, and identifies that $\mathbf{a}_{\mathbf{h_1}} = \{a_1, a_2\}$.

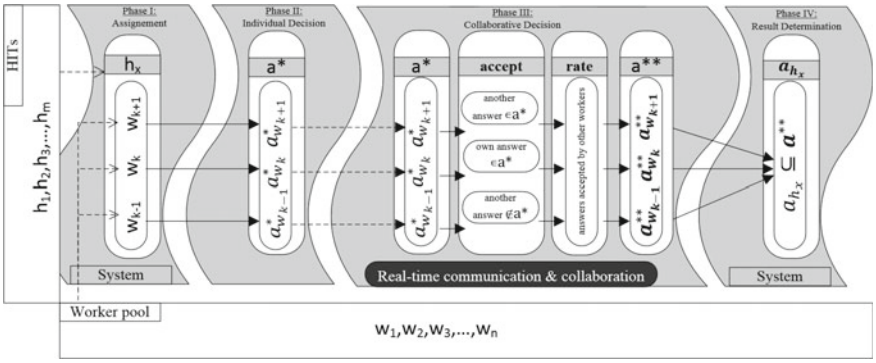


Fig. 2 Process flow of CMV for non-deterministic tasks

Table 1 Exemplary application of the rating process in CMV for non-deterministic tasks

	a_1	a_2	a_3
w_1	**valid**	valid	false
w_2	false	**valid**	false
w_3	valid	false	**valid**

Obviously, by combining the collaboration, the real-time communication and the rating procedure, the CMV method exhibits the main advantage of peer review, namely, the manual verification of the quality of the submitted results [3, 5]. Furthermore, this kind of quality control can be used not only to evaluate the results for a given task, but also to detect and to label spammers on the fly by analyzing a certain set of completed tasks and the corresponding ratings provided by other workers. Identifying and blocking low-performing workers requires more complex methods, and usually imposes additional efforts in quality management [18, 24].

4 Discussion

Benefits The main weakness of traditional majority vote lies in the fact that correct results may be rejected if they disagree with the wrong decision of the majority. According to the process flow of CMV for deterministic tasks, the next example demonstrates how our method can counteract this deficit (see Fig. 3): Resolving the same task $h_{10} \in \mathit{HITs}$ independently of each other, the workers w_1, w_2 and w_3 produce the following results: $a^*_{w_1} = a_1, a^*_{w_2} = a_3$ and $a^*_{w_3} = a_1$. Assume that only the answer a_3 is correct. Applying majority vote, the system will calculate the false result a_1 for h_{10} and correspondingly downgrade the failure rate of worker w_2. In contrast, CMV allows for the worker w_2 to communicate and to affect the answers of the others (e.g., by providing links to answer sources) so that worker w_1 changes the decision to a_3. In this case, majority vote will result in the correct answer a_3. Such willingness of more experienced workers to correct errors of other participants was observed in the experiment conducted by Kulkarni et al. [10].

The objective of the CMV method is to transfer the benefits of synchronous collaboration, which have been already well-established in other domains to the area of quality assurance in crowdsourcing marketplaces and to open up methodically new perspectives for ensuring high-quality processing of tasks. In general, providing mechanisms for real-time collaborations enables outsourcing a wider range of task types that are currently out of scope of micro-task crowdsourcing.

Enabling Technology In order to implement and evaluate our approach, we need a suitable software technology for supporting real-time collaboration and communication between workers. Special attention is hereby placed on co-scrolling and co-form filling. After a comparison of existing tools we decided on the novel lightweight

collaboration solution, which enables two or more workers to connect and to work together in a single web application at the same time [22, 23]. They can collaboratively click, type, navigate, and follow actions of other participants in real-time. In contrast to other similar solutions, it requires no downloads or plug-in installations, provides far better scaling and allows for the collaboration of 100 or more users in a single session without experiencing significant time-lags.

Restrictions Because of the newness of the proposed CMV approach, its limitations have yet to be studied depending on task types and scenarios of processing. With regard to Scenario I in Sect. 3.3, low-performing workers and spammers can continuously change their primary decisions aiming to reduce their failure rates. To avoid this, suitable countermeasures are needed, e.g., such as the rating procedure described in Sect. 3.4. In the case where workers have to agree on the same answer (see Scenario II in Sect. 3.4), the collaboration may lead to an increase in abortion rates during the process of task execution.

Propositions Based on the literature review and analysis we project the identified positive effects of collaborative work into the proposed CMV approach and derive the following tentative propositions:

P1 Knowing that the own contributions can be inspected by others prompts crowd workers perform their work more efficiently and enables better spammer identification, thus improving the result quality.
P2 Real-time communication reduces problems regarding task understanding and combines different worker skills and perspectives, thus improving the result quality.
P3 Group work increases workers' motivation and satisfaction, thus improving the perception of the quality assurance mechanisms.

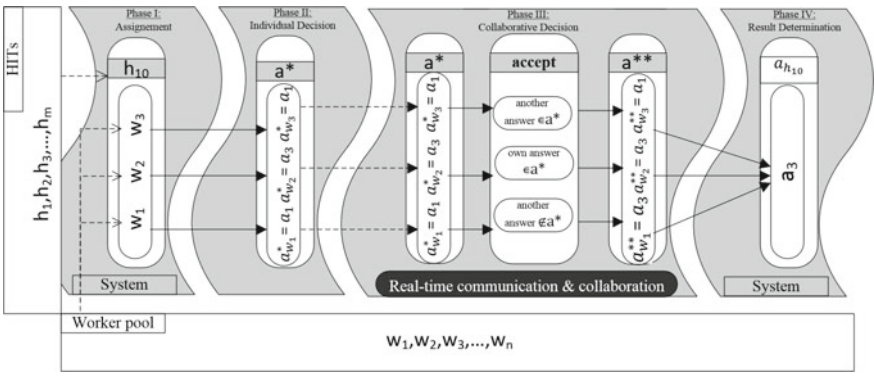


Fig. 3 Counteracting the wrong decision of majority

5 Conclusion and Further Work

In this paper we proposed a new method for quality management on crowdsourcing market-places, which extends the traditional majority vote by real-time collaboration capabilities and aims at improving result quality of submitted micro-tasks. First, we presented the conceptual model by describing the general process flow in CMV and assumptions made. Second, we illustrated three application scenarios with appropriate examples for deterministic and non-deterministic task types. At the end, we discussed possible restrictions of our approach and derived three propositions. To confirm or deny these propositions, to find out further limitations, as well as to identify possible improvements in the collaboration phase, in our ongoing research we are working on testing and evaluating the proposed CMV approach. During the ongoing experiment we aim to answer the following research questions:

- What effect does the collaboration have on the quality of work results?
- What types of micro-tasks can be solved more effectively in a collaborative environment?

As soon as evaluation results are received, they will be published as a follow-up to this research paper.

References

1. Chesbrough, H.W.: Open Innovation: The New Imperative for Creating and Profiting from Technology. Harvard Business School Press Books, Boston (2003)
2. Howe, J.: Crowdsourcing: Why the Power of the Crowd Is Driving the Future of Business. Crown Publishing Group, New York (2008)
3. Ipeirotis, P.G., Provost, F., Wang, J.: Quality management on Amazon Mechanical Turk. In: Proceedings of the ACM SIGKDD Workshop on Human Computation (2010). doi:10.1145/1837885.1837906
4. Kazai, G.: An exploration of the influence that task parameters have on the performance of crowds. In: Proceedings of the CrowdConf (2010)
5. Kern, R., Bauer, C., Thies, H., Satzger, G.: Validating results of human-based electronic services leveraging multiple reviewers. In: Proceedings of the 16th Americas Conference on Information Systems (2010)
6. Kern, R., Thies, H., Satzger, G.: Efficient quality management of human-based electronic services leveraging group decision making. In: Proceedings of the 19th European Conference on Information Systems (2011)
7. Kern, R., Thies, H., Zirpins, C., Satzger, G.: Dynamic and goal-based quality management for human-based electronic services. Int. J. Coop. Inf. Syst. (2012). doi:10.1142/S0218843012400011
8. Kittur, A.: Crowdsourcing, collaboration and creativity. XRDS: crossroads, the ACM magazine for students (2010). doi:10.1145/1869086.1869096
9. Kosinski, M., Bachrach, Y., Kasneci, G., Van-Gael, J., Graepel, T.: Crowd IQ: measuring the intelligence of crowdsourcing platforms. In: Proceedings of the 3rd Annual ACM Web Science Conference (2012). doi:10.1145/2380718.2380739
10. Kulkarni, A.P., Can, M., Hartmann, B.: Turkomatic: automatic recursive task and workflow design for mechanical turk. In: Proceedings of ACM CHI Conference on Human Factors in Computing Systems (2011)

11. Leimeister, J.M., Huber, M., Bretschneider, U., Krcmar, H.: Leveraging crowdsourcing: activation-supporting components for IT-based ideas competition. J. Manag. Inf. Syst. (2009). doi:10.2753/MIS0742-1222260108
12. Little, G., Chilton, L.B., Goldman, M., Miller R.C.: Exploring iterative and parallel human computation processes. In: Proceedings of the ACM SIGKDD Workshop on Human Computation (2010). doi:10.1145/1837885.1837907
13. Lowry, P.B., Albrecht, C.C., Lee, J.D., Nunamaker, J.F.: Users' experiences in collaborative writing using collaboratus, an internet-based collaborative work. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (2002). doi:10.1109/HICSS.2002.993879
14. McCarthy, J., Miles, V., Monk, A.: An experimental study of common ground in text-based communication. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (1991). doi:10.1145/108844.108890
15. Morris, M.R.: Interfaces for collaborative exploratory web search: motivations and directions for multi-user design. In: Proceedings of ACM SIGCHI Conference on Human Factors in Computing Systems, Workshop on Exploratory Search and HCI: Designing and Evaluating Interfaces to Support Exploratory Search Interaction, pp. 9–12 (2007)
16. Oleson, D., Sorokin, A., Laughlin, G., Hester, V., Le, J., Biewald, L.: Programmatic gold: targeted and scalable quality assurance in crowdsourcing. In: The 3rd Human Computation Workshop (2011)
17. Quinn, A.J., Bederson, B.B.: Human computation: a survey and taxonomy of a growing field. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (2011). doi:10.1145/1978942.1979148
18. Raykar, V.C., Yu, S.: Eliminating spammers and ranking annotators for crowdsourced labeling tasks. J. Mach. Learn. Res. **13**, 491–518 (2012)
19. Schulze, T., Nordheimer, D., Schader, M.: Worker perception of quality assurance mechanisms in crowdsourcing and human computation markets. In: Proceedings of the 19th Americas Conference on Information Systems (2013)
20. Snow, R., O'Connor, B., Jurafsky, D., Ng, A.Y.: Cheap and fast—but is it good? Evaluating non-expert annotations for natural language tasks. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (2008)
21. Sorokin, A., Forsyth, D.: Utility data annotation with Amazon Mechanical Turk. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (2008). doi:10.1109/cvprw.2008.4562953
22. Thum, C.: Enabling lightweight real-time collaboration. In: Becker, C., Gaul, W., Heinzl, A., Schader, M., Veit, D. (eds.) Informationstechnologie und Oekonomie, Band 41, Peter-Lang-Publisher, Bern, Dissertation (2012)
23. Thum, C., Schwind, M.: Synchronite—a service for real-time lightweight collaboration. In: Proceedings of the 2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (2010). doi:10.1109/3PGCIC.2010.36
24. Vuurens, J., de Vries, A.P., Eickhoff, C.: How much spam can you take? An analysis of crowdsourcing results to increase accuracy. In: Proceedings of ACM SIGIR Workshop on Crowdsourcing for Information Retrieval (2011)

Towards a Game Theoretical Model for Software Crowdsourcing Processes

Wenjun Wu, Wei-Tek Tsai, Zhenghui Hu and Yuchuan Wu

Abstract Recently software crowdsourcing has become an emerging development paradigm in software ecosystems. This paper first introduces a software crowdsourcing framework in the context of software ecosystems. The framework includes a game-theoretical model for peer software production to describe the competitive nature of software crowdsourcing. The analysis of this model indicates that if the only reward is the prize, only superior developers will participate in the software crowdsourcing. This explains the phenomenon that while software crowdsourcing is open for anyone to compete, but only few will engage in competition. This is validated by a large historical data collected from a popular software crowdsourcing website over a 10-years period. Further, we perform a case study on a NASA software crowdsourcing project to take a closer examination at how community developers participate in different types of tasks through the software crowdsourcing process.

1 Introduction

With the increase in software system complexity, many companies open up their platforms to embrace the collaborative effort from online developers to develop software. Software crowdsourcing is a promising approach to allow organizations to outsource software development tasks to a virtual and on-demand workforce. By tapping into the collective intelligence of participants, organizations aim to acquire

W. Wu (✉) · Z. Hu · Y. Wu
School of Computer Science and Engineering, Beihang University, Beijing, China
e-mail: wwj@nlsde.buaa.edu.cn

W.-T. Tsai
School of Computing Informatics and Decision Systems Engineering,
Arizona State University, Tempe, AZ, USA
e-mail: wtsai@asu.edu

Z. Hu
e-mail: waiaml@nlsde.buaa.edu.cn

Y. Wu
e-mail: wuyuchuan@nlsde.buaa.edu.cn

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_9

the high quality software with a reduced cost while stimulating the community to focus on certain skills or knowledge in various software ecosystems.

Currently, major software crowdsourcing platforms including Apples App Store, TopCoder [1], and uTest [2], demonstrate the advantage of crowdsourcing in terms of software ecosystem expansion and product quality improvement. Apples App Store is an online IOS application market [3, 4], where developers can directly deliver their creative designs and products to smartphone customers. These developers are motivated to contribute innovative designs for both reputation and payment by the micro-payment mechanism of the App Store. Around the App Store, there are many community-based, collaborative platforms for the smart-phone applications incubators. For example, AppStori [5] introduces a crowd funding approach to build an online community for developing promising ideas about new iPhone applications. Another software crowdsourcing example C TopCoder, creates a software contest model where programming tasks are posted as contests and the developer of the best solution wins the top prize. Following this model, TopCoder has established an online platform to support its ecosystem and gathered a virtual global workforce with more than 250,000 registered members and nearly 50,000 active participants. All these TopCoder members compete against each other in software development tasks such as requirement analysis, algorithm design, coding, and testing.

The way of organizing software development in all these practices of software crowdsourcing is changing from traditional software factory or distributed development teams to decentralized, peer-production based ecosystems of software developers. Firstly, software crowdsourcing open up its development process to online community. Its openness doesnt refer to free access to source code like open source development. Instead, it denotes an OPEN call for participation in any tasks of software development, including documentation, design, coding and testing. Secondly, it depends upon community workforce in its decentralized development process. Software development tasks are normally conducted by either internal staffs within a software enterprise or people from contracting firms. For example, software companies can adopt global software development [6] where multiple teams of professionals work collaboratively in remote locations. But in software crowdsourcing, all the tasks can be assigned to anyone from general public. Lastly, software crowdsourcing introduces explicit incentives such as financial rewards to motivate community developers. Open source projects usually rely upon volunteers with reputations to accomplish tasks [7]. Thus, it greatly extends the concept of open source community into the notion of market-driven software ecosystem.

Although software crowdsourcing indicates a new trend in software development, fundamental principles behind software crowdsourcing are still being developed. There is a need to study software crowdsourcing processes in a market-driven software ecosystem. Majority of publications on distributed software processes [8–11] focus on facilitation of collaboration and information sharing among software engineering teams or individual programmers. They did not propose frameworks to model software crowdsourcing particularly those competitive software crowdsourcing processes such as those practised by TopCoder. Interestingly, scholars from the economics and management science have studied the mechanism of

crowdsourcing systems, such as pricing and bidding strategies as well as rewarding rules [12–14]. But their approaches are limited in the scope of classic auction methodology and are not directly related to software crowdsourcing process, i.e., maximizing the software quality and creativity via crowdsourcing.

This paper examines a variety of issues in software crowdsourcing including quality, costs, diversity of solutions, and competition scenarios. The success of a software crowdsourcing process must be able to broaden participation, ensure quality of solution, encourage diversity of solutions, identify potential talents and maximize learning for both active participants and passive observers. This paper proposes a new software process model to describe the competitive nature of software crowdsourcing process. The rest of the paper is organized as follows: Sect. 2 presents the framework of software crowdsourcing for software ecosystem. Section 3 develops a theoretical model of peer software production in the process of software processing based on contest theory. A case study is conducted to examine a NASA software crowdsourcing project in Sect. 4. Section 5 gives the conclusion.

2 Crowdsourcing for Software Ecosystem

A software ecosystem is a networked community of organizations, where they share a common interest in a central software technology. Apples App Store is an example of market-driven software ecosystem, where developers, vendors and end-users exchange their values through IOS based smartphone applications. Crowdsourcing can play as an essential social instrument to build a vibrant software ecosystem and promote its growth. This section describes the major components in a software ecosystem from the perspective for social-technical theory and presents a software crowdsourcing process model for the development of a software ecosystem.

2.1 Socio-Technical Ecosystem

According to the framework defined in [15], a social-technical software ecosystem has two major aspects: technical platform and stakeholder community built around the platform. Figure 1 illustrates such a social-technical ecosystem toward cloud applications.

The bottom layers in the common cloud architecture: IaaS and PaaS specify the standard API and reference implementation of a cloud-based platform. As SaaS (Software-as-a-Service) is often about domain-specific application-level software, it can be regarded as a collection of applications that can be further derived and composed into more full-fledge application systems. Every application, no matter whether atomic or composite, can be developed and contributed by community users and developers.

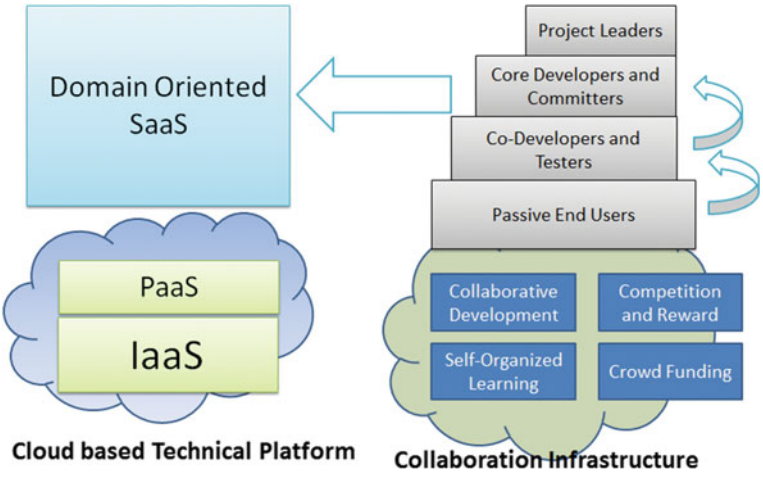


Fig. 1 Cloud based social-technical ecosystem

The collaboration infrastructure is a hierarchical community with passive end-users, a set of internal and external developers as well as domain experts. Passive end-users are interested in using applications provided by SaaS without making any comments on the quality of the application. More active users will get involved in the SaaS development by posting their feedbacks and testing the available application services. Some of them can fix the minor issues and contribute patches to the software repository. On the top levels of this hierarchy, there are project leaders and core developers who are in charge of the major tasks in software design and maintenance.

The governance regime in this ecosystem should be the major stakeholders who can make strategic decisions and utilize coordination mechanisms to motivate people to join in the community and make contributions. Common coordination mechanisms include collaborative development, competition reward, crowd funding, self-organized learning for knowledge transfer. Software crowdsourcing is an effective governance paradigm when the project leaders need to open up its development process and make open calls to the community for undertaking project tasks. It enables the leading stakeholders in the ecosystem to seek innovative design ideas, identify talents, increase participation from the crowd, improve software quality and reduce development cost. The complexity and time-spanning of projects in the software ecosystem determines the process of software crowdsourcing processes.

2.2 Software Crowdsourcing Process Model

A software crowdsourcing process is to motivate community developers to participant in peer production tasks specified by stakeholders in a software ecosystem and deliver the quality software products to the software ecosystem. Figure 2 illustrates the basic

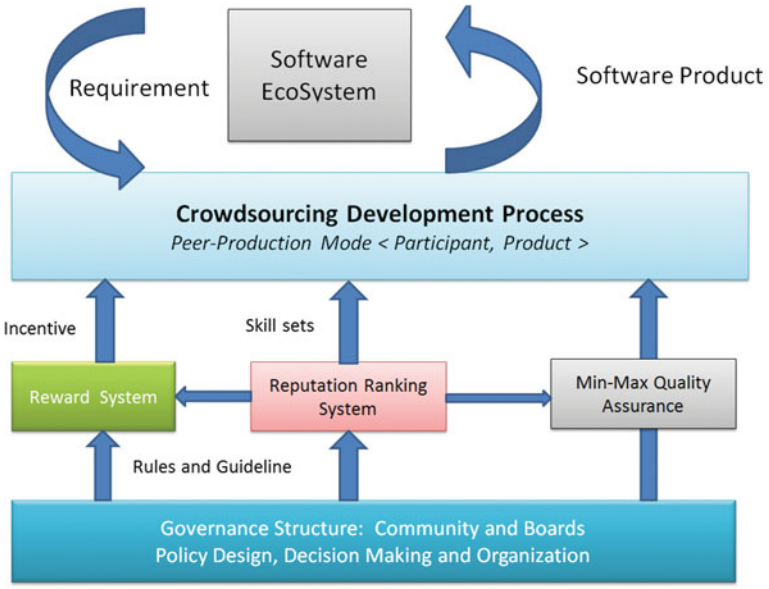


Fig. 2 Software crowdsourcing for developing software ecosystems

management structure of software crowdsourcing process, which consists of the reward system, reputation ranking system, and quality assurance system as well as governance regime.

The governance board is the top-level management body in charge of policy design, strategic decision making and architectural planning as well as regulation. The board has three essential tools to cultivate a healthy and vibrant community of the software ecosystem: (1) a reward mechanism; (2) a reputation mechanism; and (3) a quality assurance mechanism. The reward mechanism provides the incentive for crowds in the community to participant in software development. The reputation ranking mechanism classifies the skill levels of each individual member in the community and defines the foundations for the reward system. The quality assurance system evaluates the quality of products contributed by crowds and selects the best quality one to integrate it into the software ecosystem. Based on this human infrastructure, project managers can adopt the peer production mode to innovate their conventional software development life cycles and achieve multiple goals in the development of software systems, including quality software, rapid acquisition, talents identification, cost reduction, ideas creation and broadening participation. After the priority of the specific goals is set, appropriate cooperation and competition mechanism of the project must be devised to steer the process towards these goals.

(1) Reward System
Crowdsourcing needs an effective rewarding system to attract the broader engagement from the developer community, which is critical to maintain the community diversity and vibrancy of a software ecosystem. For an open-source community, its rewarding system often deeply relies upon its hacker cultures where people have strong willingness to improve their coding skills, or to enjoy the challenging programming tasks, or to achieve and enhance their reputation within the community through voluntary contributions [16]. Although these social rewards mechanisms are still effective in crowdsourcing-based software development, it is not enough to drive loosely-coupled individuals to efficiently accomplish complex tasks. Financial incentives provide powerful motivation to boost performance of crowd workforce [17]. According to various economic theories, rational workers will choose to improve their performance in response to a scheme that rewards such improvements with financial gain. Performance-based pay schemes can increase both the quantity of work and the quality of product.

(2) Reputation System
A reputation system gathers, distributes, and aggregates feedbacks about the behavior and historical performance of both requestors and workers, and employs a ranking scheme to generate scores for them. Such a rank scheme usually is developed on the basis of user comments, third-party reviews, and competition performance evaluation [18]. In general, the major components in a reputation mechanism often include: evaluation metrics for the task performed, feedback collectors, and reputation engine.

Evaluation metrics: Reputation rating can be either objective or subjective. Quantity measurements of an individuals performance, such as the number of successful submissions, total rewards and download counts, can demonstrate software design skills. Other metrics such as third-party rating and user comments can present subjective opinions on the quality of specific submissions. Not only the workers and their work can be evaluated, the credibility of requestors can also be evaluated including their history of delivering compensation in time.

Feedback collectors: A feedback collector can collect the reputation-related information and build up a detailed profile for each person in this community to determine their skillset.

Reputation engine: A reputation engine can compute the value of the users reputation ratings from their profiles provider by the feedback collector and the latest performance in the crowdsourcing. Rating scheme for crowdsourcing systems is still in its early stage, the contest rating has been a research problem for a long time. For example, the well-known Elo rating model [19] for calculating skill levels of chess players based on the statistics has been widely used in a various contests ranging from chess challenges to baseball games. Many improvements have been done based on the Elo rating model to enhance the accuracy of the rating algorithm and handle multiple player game situations by incorporating more factors such as players volatility and Bayesian inference framework [20, 21].

(3) Min-Max Quality Assurance
Given the inherent diversity among the community in a software ecosystem, it is unlikely that the quality of crowd submissions can stay stable for various projects. Thus, software crowdsourcing process needs quality assurance mechanism to review and evaluate the crowd submissions. Min-Max model is introduced in [22] to describe the quality assurance mechanism in a software process, where one party tries to minimize an objective function, yet the other party tries to maximize the same objective function as though both parties compete with each other in a game. For example, a specification team needs to produce quality specifications for the coding team to develop the code; the specification team will minimize the potential errors in his specification document, while the coding team will identify as many bugs as possible in the specification document before coding.

The Min-Max quality assurance mechanism involves two basic operations C offense and defense:

Offense: each team needs to understand its requirements, and examine the validity of the inputs to determine if they are feasible, correct, consistent and complete. This can be done by inspecting, reviewing, simulating, model checking, verifying the contents of the inputs. Because any mistakes in the input document may cause significant problems in the current tasks. Thus, the goal is to maximize the fault detection rate of the input documents.

Defense: Once requirements are understood, the team needs to prepare its output. However, the team realizes that their outputs will be cross examined by other teams carefully, and the team may lose its creditability if its outputs are of low quality. Thus, the team needs to spend significant time to check and verify its deliverables to minimize the probability of bugs and to minimize the damage of potential bugs.

There are two ways to implement quality assurance for software crowdsourcing: central review and peer review. In the case of central review, the community governance body set up a specialized team for undertaking offense duty in testing and reviewing crowd submissions. It works well when the scale of crowd participation is limited. When it comes to large scale of software crowdsourcing, it becomes infeasible to review massive number of submissions only through a central offense team. In such a scenario, peer review needs to be adopted to scale up the process, which means that crowd developers needs to perform both offense and defense operations to eliminate unqualified submissions. For example, algorithm contests in TopCoder have a special challenge phase where participants can eliminate their opponents by finding bugs in their code.

(4) Crowdsourcing Governance
The governance of software crowdsourcing is a set of structures, processes and policies by which the software development within the community is directed and controlled to maximize the community value. A common approach is to use an open technical platform and with an open architecture, the governance board can continually align synergistic relationships of people, knowledge, and resources that promote harmonious software growth in ever changing requirements.

The governing board may include experts in the application domain and in software engineering. The board is responsible for creating outlines for the project and making strategic decisions to steer the progress as well as mitigating major risks during development. The appropriate policies and rules in both rewarding and reputation mechanism need to be designed and optimized by the board to expand the scale of community network and maintain the skill levels among developers. Moreover, while the board should set policies and development guidelines to efficiently regulate the practice of the community to create rules accepted by the community, community members can also request the board for new or modifications of existing rules.

The board also leads a working group who follows the policies set by the board to coordinate individuals in the community for the project development. The members in this working group can be elected from the crowdsourcing community. Their major responsibilities include: maintaining the crowdsourcing platform, leading the efforts of software crowdsourcing activities, evaluating the outcomes from crowdsourcing tasks and integrating crowd contributions into the software ecosystem.

3 Peer Productions and Contest Theory Model

This section presents a contest theory model to describe the peer production process in software crowdsourcing, and the model can be used to analyse various factors including the effort to finish the competing tasks, the award prize and skill level of competing players. All the major software artifacts, such as requirement specification, design scheme, software components, can be developed via peer production through software crowdsourcing.

Each crowdsourcing task involves N workers who independently produce M solutions following the specification of the task. Normally the number of the solutions is smaller than the number of the worker ($M \leq N$), because not all the workers can successfully finish the task within the specified time constraint. Afterward, these M solutions are examined and evaluated to select P products that completely satisfy the requirement of the task. Note that there are different ways to handle the candidate solutions to create the final product. For example, P solutions are of the same functions and only the best will be chosen as the final product. In other cases, P solutions are merely partitions of the entire software product and need to be merged together. The peer production model can be formally defined as follows:

Definition 3.1

- W *worker*, $W = \{w_1, w_2, ..., w_N\}$
- T *Task*, $T = t_1, t_2, ..., t_M$
- B *Bid*, $B_{w,t}$ *is the solution for the task* $t \in T$, *delivered by the worker* $w \in W$
- C *Cost*, $C_{w,t}$ *incurred by the worker* $w \in W$ *for the task* t

- S *Skill level of workers*, $S_w = S_{w,t}$, *represents the expertise of each worker in the task t*
- A *Activity*, $A = [A_1, A_2, ..., A_H]$, *which define a sequence of activities in the process of software crowdsourcing development*.

Each A_i is a seven-element tuple, $(W_i, t_i, B_i, U_i, X, Y, Z)$, among which $W_i \subseteq W$ denotes all the participants of the activity, $t_i \in T$ represents the task of activity, and B_i represents the solutions submitted by all the participants in this activity.

For each worker $w_k \in W_i$, he creates a solution B_{w_k,t_i} for the task t_i. All the bids submitted by the workers are defined as $B_i = \{B_{w,t_i} \mid w_1, ..., w_M \in W_i\}$. The rest element in the tuple U, X, Y, Z represents utility function, quality function, ranking function and reward function respectively.

(1) Utility function, U_i, $B^M \rightarrow V$, describes the common value that the activity organizer can collect from all the bids $B_{w,t} \in B_i$ submitted by the activity participants. Depending upon the nature of development tasks, the utility function can be devised in different forms. When the organizer manager wants to merge the valid bids into a single product, he can set an aggregation function: $U_t = \sum V(B_{w,t})$, where V is the value mapping for each bid. When the organizer manager needs to select the best bid as the final solution, he can set an maximization function: $U_t = Max(\{V(B_{w,t})\})$.
(2) The Quality Function, $X : B \rightarrow Q$, $QualityQ = [0, 1]$, $X(B_{w_k,t_i})$, describes the evaluation criteria for judging the quality of submissions.
(3) The Ranking Function, $Y : B \times Q \times S \rightarrow S, S'_{w_k} = Y(B_{w_k,t_i}, X(P_{w_k,t_i}), S_{w_k})$, specifies the mechanism to calculate the skill set of each participant based on their performance in both the current activity and the past ones.
(4) The Reward Function, $Z : B \times Q \times S \rightarrow R$, R is the set of all possible rewards, where R is the reward for the participants. Normally, only part of the workers who outperform others in the activity can be awarded in the form of prize. And the majority workers can benefit from other forms of reward such as certificates, credits and ranking scores.

Contest theory [23] introduces a theoretic framework to model competitive scenarios, where agents compete for scarce resources. In these situations agents can influence the outcome of the process by actions. A contest theory to analyze the factors among participation and prize structure of the peer-production model.

3.1 Contest Theory Model for Competitive Peer Production

Each activity A_i defined by the peer-production model should be regarded as a contest in the contest-theory framework. Assume there are a finite set of workers W who get involved in this activity. And the quality of the solution submitted by a worker w_i determines his probability of obtaining the prize. One can define a contest success function relating the quality of a workers solution to the probabilities that they obtain the prize.

Let $p_i = p_i(Q_1, Q_2, ..., Q_i, ..., Q_n)$ be the probability that worker w_i wins the crowdsourcing contest when his product quality is Q_i according to the quality function defined above. In addition, we need to define a cost function $C_i(Q_i)$ representing the cost incurred by the workers effort and a prize function $V_i(Q_1, ..., Q_n)$ representing the prize awarded to the worker. Based on these three functions, every worker can calculate his possible payoff by considering all the solutions presented by the participants:

The expected payoff function of worker i denoted by $\pi_i()$, is

$$\pi_i(Q_1, ..., Q_n) = p_i(Q_1, ..., Q_n) \times V_i(Q_1, ..., Q_n) - C_i(Q_i) \tag{3.1}$$

Contest theory models the competitive peer production as an N-player game where each worker makes his action according to the payoff function (3.1). When a workers payoff expectance is higher than zero, he will certainly spend quality time and effort on the task. Otherwise, he will lose the incentive to make contribution to the task. To simplify the analysis of this N-player game, one can assume the contest success function takes the form in (3.2):

$$p_i = \frac{\phi(Q_i)}{\Sigma_{j=1}^{n}\phi(Q_i)} \ if \ \Sigma_{j=1}^{n}\phi(Q_i) > 0, \quad p_i = \frac{1}{n} \ if \ \Sigma_{j=1}^{n}\phi(Q_i) \tag{3.2}$$

In most cases of crowdsourcing contest, we can confidently assume that the prize awarded to the winners is constant, that is $V_i(Q_1, ..., Q_n) = V$.

Define $x_i = \phi(Q_i)$, and assume $C_i(Q_i) = C_i(\phi^{-1}(x_i)) = d_i x_i$, then (3.1) can be transformed into

$$\pi_i = \frac{x_i}{\Sigma_{j=1}^{n}} V - d_i x_i \tag{3.3}$$

Proposition 3.1 There is a unique Nash equilibrium. The players can be divided into active group and inactive group.

(1) Active Group with m workers $m \leq n$, $\forall w_i \ i = 1, ... m$, $\Sigma_1^m d_j > d_i(m-1)$
(2) Inactive Group with $n-m$ workers: $\forall w_i \ i = m+1, ... n$, $\Sigma_1^m d_j \leq d_i(m-1)$

Actually, d is the ratio of a workers effort against his chance to win the contest. The proof of this proposition can be found in the survey paper of contest theory [24].

To illustrate this competition mechanism, we can calculate the active group and inactive group in a peer-production contest with ten workers. And we assume the Contest Success Function is $\phi(Q_i) = Q_i$, the d values of each player are listed in (3.4).

$$d_i = \sigma \times i, \quad i = 1, 2, ..., 10 \tag{3.4}$$

σ is the gap between the consecutive elements in the sequence. If we replace d_i of the inequality in Proposition 3.1 with the (3.4), it is easy to infer the following inequality:

$$\frac{\sigma m(m+1)}{2} > \sigma m(m-1) \tag{3.5}$$

Thus, we can have $m < 3$, which means only two workers will actively make contributions to the task. Clearly, when competitors have heterogeneous abilities, they will behave quite differently in a software crowdsourcing contest. We can examine the worker behavior in a more general scenario. Assume d follow a probability distribution. Since d depends upon a players skill. The higher his skill is, the lower effort he needs to take to win the contest. When the skill levels among the players are uniformly distributed or normally distributed, we can run a simple test to calculate the division point M in both cases. Suppose the uniform distribution is $d \sim \cup(0, 1)$ and the normal distribution is $d \sim \cup(0.5, 1)$. With the number of participant N enumerated from 10 to 200, we can calculate the M-value according to Proposition 3.1. Figure 3 demonstrates the results of both cases.

Interestingly, the M-value seems not be affected by the participant number and completely determined by the random variable Ds distribution. When it is a uniform distribution, M stays between 2 and 3, with the average value of 2.6. When it is a normal distribution, M nearly remains within the range from between 3 and 4.5. The partition of the players demonstrates the evidence of so-called superstar effect in

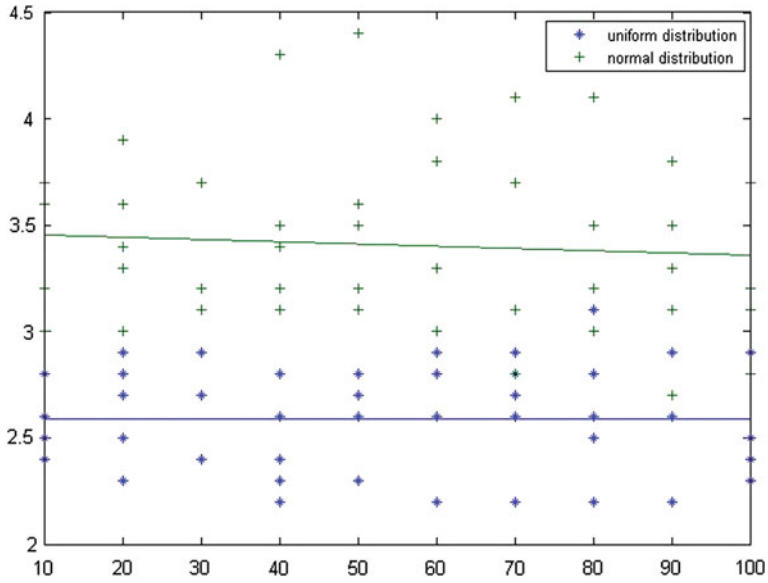


Fig. 3 M-value under different D values (uniform versus normal distribution)

many competitive sporting environments where regular players have little incentive to exert effort at the presence of super players.

3.2 TopCoder Participation and Prize

The theoretical results of the contest model can be validated by historical contest data collected from the TopCoder website. The data included 1734 design contests and 910 development contests from 4/2004 to 2/2012. Figures 4, 5 and 6 demonstrates the number of registrations, submissions, and valid solutions under the influence of prize reward.

Apparently, the average number of submissions in TopCoder development contests is much lower than the number of active coders available. The average registration numbers for TopCoder design and development competitions are about 13 and 25 respectively. And out of these registrations, only 2 design submissions and 5 development submissions have been delivered. This phenomenon confirms the theoretical analysis in Sect. 3.1, where only 2 or 3 workers with superior skills will be active. Once a potential participant senses that two strong contenders already signed up for a competition, they will opt out from the competition.

The other important observation about the data is that prize value has little influence on the motivation of community workers. Contrary to the common belief, higher prize value will not attract more participation to compete. This may be caused by the difficulty of those tasks with a high price. According to the statistics of TopCoder, the average completion duration of a TopCoder task is about two weeks. So the tight time constraints of those challenging tasks can render too much risk for most workers to take.

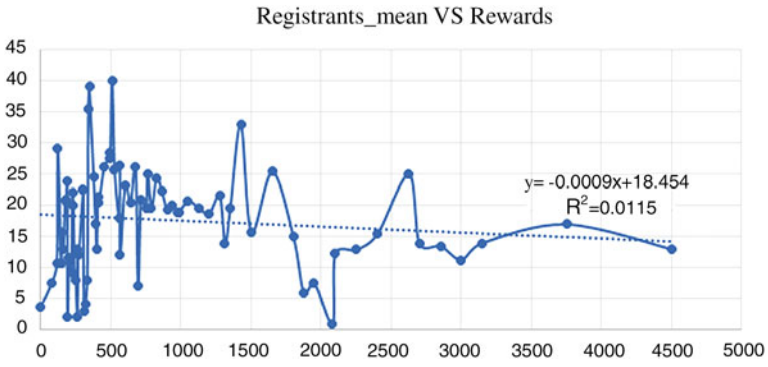


Fig. 4 Registration-reward curve in development contests of TopCoder

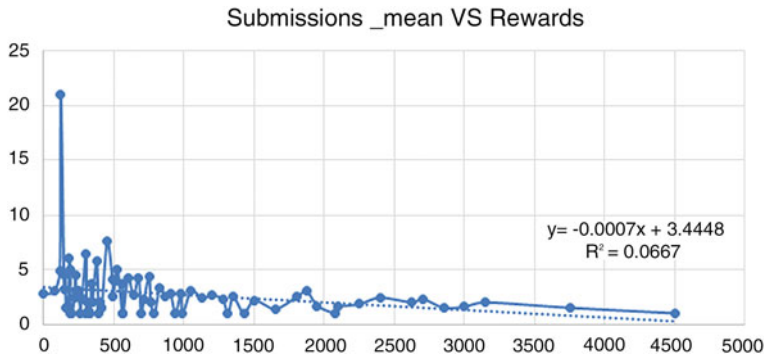


Fig. 5 Submission-reward curve in development contests of TopCoder

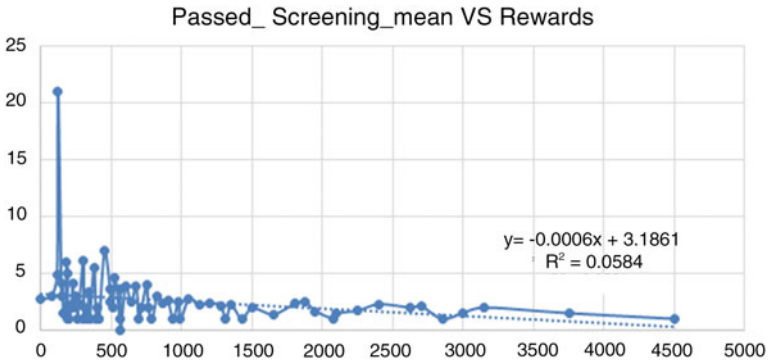


Fig. 6 Effective-reward curve in development contests of TopCoder

4 Case Study

This section presents a case study based on the research from MIT team [24]. This software crowdsourcing project is about building a Web collaboratory named by Zero Robotics where high school students can develop their robotic program on SPHERES, a mini satellite device inside the International Space Station.

The software architecture of this collaboratory is a cloud-based science gateway with a Web portal and a backend computation engine. The Web portal provides a programming interface for students to write their codes and simulate the control of the SPHERES device. And the distributed computation engine that can allocate Amazon EC2 virtual machines on demand is responsible for user code compilation, runtime linkage and execution. These two major components of Zero Robotics were developed in different ways: the computation engine was implemented by in-house programmers in the team, while the portal was crowdsourced to the TopCoder community.

According to the design document of the project, the portal adopts Liferay Portal, JBoss and MySQL, all of them are popular open-source Java Web application frameworks. The MIT team aimed at utilizing the collective intelligence of thousands of TopCoder programmers with Java expertise to complete this project. It designed a development plan that divided the project into the eight phases including conceptualization, requirement specification, UI prototype, architecture and component design, component implementation, testing and deployment. The plan consists of 54 design and development contests and 163 bug race contests from April 2011 to December 2011. Figure 7 displays this software crowdsourcing process.

(1) Conceptualization: Only one conceptualization contest was held to produce the documents including system overview, high-level workflow and use-case diagrams pertaining to the Zero Robotics portal. Only two submissions were delivered from 19 registrants, with the total prize of $2,040.

(2) Requirement Specification: In specification contests, competitors collaborate with the winners of previous contests and TopCoder clients in the project forums to finalize the Application Requirement Specification for the new system. The average cost of the three contests is $2,210.

(3) Architecture and Component Design: The purpose of architecture and component contests is to seek optimal design for the overall framework and its major components including user registration, system administration, simulation editor and scoring engine based on the software requirements produced in the specification contests. Totally 141 TopCoder members registered the 13 contests and delivered 17 submissions with the average cost of $2,367.

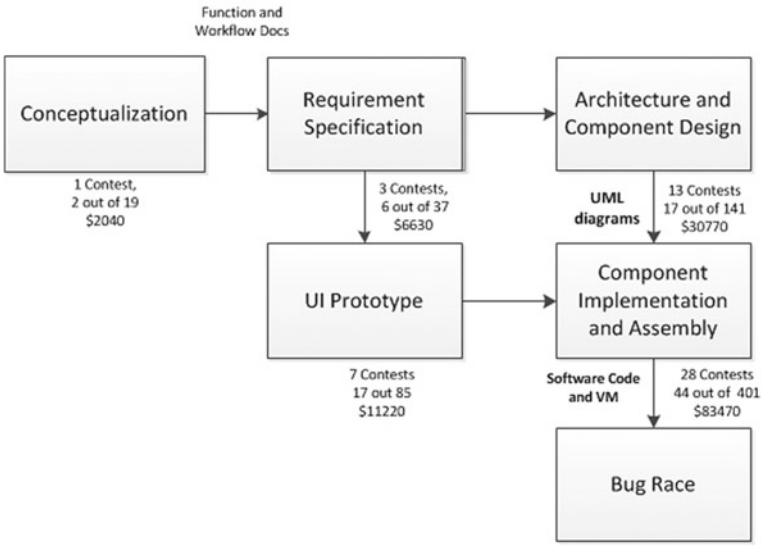


Fig. 7 Zero robotics software development processes

(4) UI Prototype: In addition to the design contests for software architecture and components, the Zero Robotics project also organized seven contests to develop innovative ideas and design scheme for the user interface via fast UI prototyping tools. Interestingly, the project only spent $1,602 for 17 quality solutions out of 85 submissions among all the contests, whose cost was obviously lower than any other types of the contests. The reason for the high production efficiency in the UI prototype contests can be well explained by the Proposition 3.1. Because a UI design activity usually needs less technical skills than other software development tasks such as architecture design and coding, it tends to attract more active members with the belief of high winner probability.

(5) Component Implementation: The Component Implementation and Assembly has the largest number of contests—28 contests for developing Alliance Portlet, Tournament Management Portlet, User Profile Portlet and other IDE components. And it is the most expensive phase during which the MIT team spent $2981 for each contest. Obviously, the component implementation contests often demands high programming skills and require significant efforts to win the prizes, thus suppressing the motivation of registrants without dominant skill levels. In fact, only 44 qualified solutions were produced out of 401 registrations through the component contests.

In summary, all the Zero Robotics contests cumulatively received 149 full submissions out of 700 registrations, and 57 prizes for these contests were awarded. There have been a total of 239 unique participants in the 54 contests. We can visualize the competitive relationship among these participants in Fig. 8. Each node represents a participant and a link between two nodes indicates that the two participants have engaged in the same contest.

This competitive graph contains a weakly connected component, indicating that the participants in the Zero Robotics projects are interested on all kinds of contest announcements. For instance, registrants in the assembly contests may also register the UI prototype contests. Furthermore, the participants in the same type of the contests exhibit similarity because their connections display the same modularity class in the graph. The characteristics of the graph demonstrate that the project successfully captured the attention of the community.

We can further examine the relationship between the contests and the contributing participants by creating a bipartite graph with two separate node sets: contest nodes and participant nodes. In the graph shown by Fig. 9, the large vertices represent the contests and the small vertices represent the participants who delivered a submission in the contests. We adopt a color scheme to distinguish different types of contests and participants. There are seven types of contest nodes, each of which are designed by a color. There are also two types of participant nodes: the nodes with the red color refer to the contest winners and the other nodes with the light color refer to those who regularly deliver submissions but fail to win.

Apparently, the bipartite graph doesnt have the same rich connectedness as the graph in Fig. 8. Because we removed all the participant nodes from the graph, who only register a contest but fail to deliver submissions, this graph contains much less nodes than the one in Fig. 8. Moreover, the whole graph is clearly partitioned into

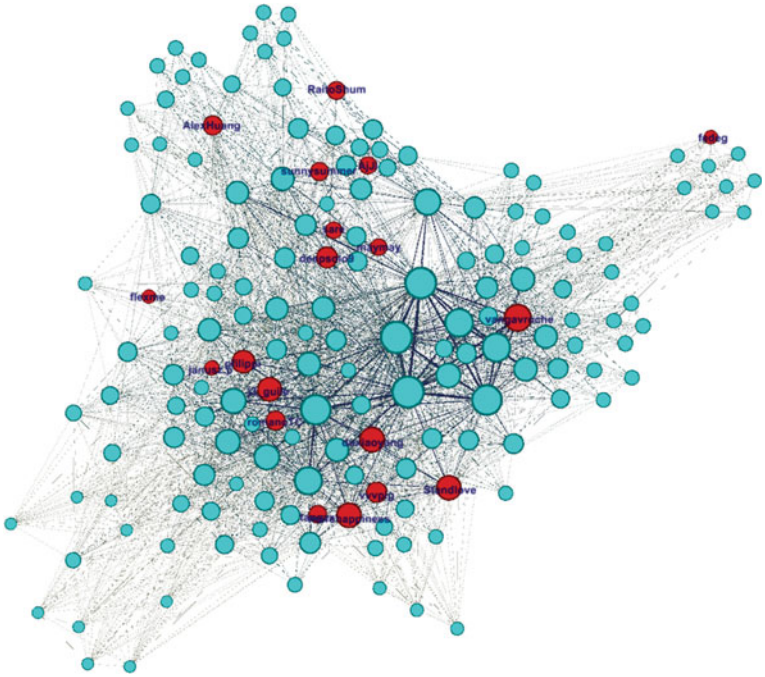


Fig. 8 A competition network consisting of all the registrants

multiple communities according to the types of the contests. Within each community, dominate players seem to win the prizes in the most contests. It can be explained by the fact that normally an active TopCoder member only possesses skills for a specific type of software contests and dedicates his efforts to those contests. For example, when a player is good at developing requirement specifications, and has gained valuable experience through the previous requirement contests in the project, it is reasonable for the person to stay focused on the requirement contests, instead of component development contests.

The case study confirms the contest theory model defined in Sect. 3. A disproportionately large number of people registered for these contests but cannot submit their work to complete the contests regularly. It appears that they gauged their probability of winning by the discussion forum content; and a small subset of the participants ultimately followed through to submit a solution. Furthermore, the emergence of players with superior skills in certain contests will discourage other participants from engaging in the competition.

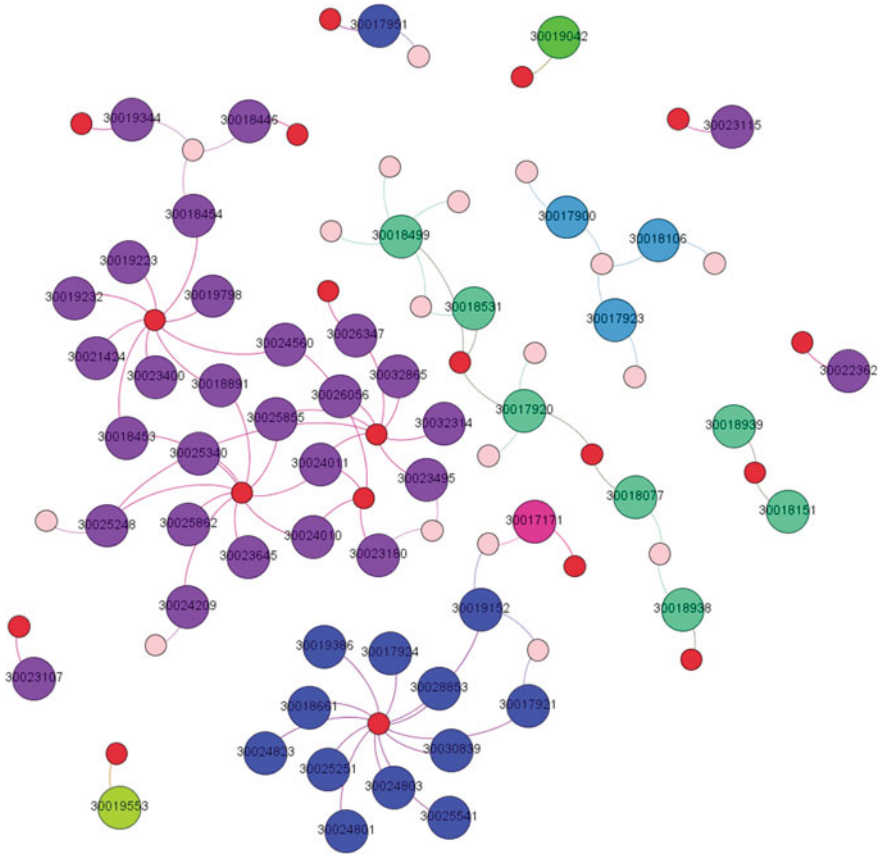


Fig. 9 A Bipartite graph consisting of contests and contributors

5 Conclusion

As the vital social instrument to a software ecosystem, software crowdsourcing can foster community formation and leverage collective intelligence for quality software development. This paper analyzes software crowdsourcing processes, and examines its key characteristics. Specifically, it proposes a game-theoretical model to analyze the relationship between crowd incentive, development skills and software quality in competitive peer production. The analysis of the model indicates that the prize-only awarding mechanism can motivate developers with superior skills to participate. This result is validated through historical data collected from the TopCoder website. Further, we perform a case study on a NASA Zero Robotics collaborator project that adopted the TopCoder platform for its portal development. By visualizing the competitive interaction between community participants and development tasks via

bipartite graph, we found that each type of design and coding tasks retains a handful of contributors who kept winning the contests with qualified solutions.

There are still many open questions to be explored in future work:

(1) Currently our model has only been applied in the TopCoder process with the prize awarding mechanism. It does not cover the cases when software crowdsourcing process adopts other non-monetary and hybrid incentives as the award mechanisms.

(2) One of the major concerns on software process research is the optimization of the project management with the constraint of both project budget and time schedule. It is important to study the properties of Nash equilibrium in our model and design a game theory based process allocation algorithm to fulfill the optimal goal of crowd development.

We plan to investigate more case studies and gather more data from other crowdsourcing platforms to further validate the framework and its related mathematical models. In addition, we will develop software tools to facilitate software architects to design software crowdsourcing processes, estimate project costs and mitigate potential risk incurred by crowdsourcing.

Acknowledgments This work was supported by National High-Tech R&D Program of China (Grant No. 2013AA01A210) and the State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2013ZX-03).

References

1. Lakhani, K., Garvin, D.A., Eric, L.: Topcoder (a): developing software through crowdsourcing (15 January 2010). Harvard Business School General Management Unit Case No. 610-032 (2010)
2. uTest, https://www.utest.com/ (2013). Accessed 12 Jan 2013
3. Bosch, J.: From software product lines to software ecosystems. In: Proceedings of the 13th International Software Product Line Conference, SPLC'09, pp. 111–119. Carnegie Mellon University, Pittsburgh (2009)
4. Jansen, S., Finkelstein, A., Brinkkemper, S.: A sense of community:a research agenda for software ecosystems. In: Presented at the 31st International Conference on Software Engineering—Companion Volume, ICSE-Companion 2009, pp. 187–190. IEEE (2009)
5. AppStori: http://appstori.com/ (2012). Accessed 10 Jun 2012
6. Herbsleb, J.D., Moitra, D.: Global software development. Softw. IEEE **18**(2), 16–20 (2001)
7. Hars, A., Ou, S.: Working for free? Motivations of participating in open source projects. In: Proceedings of the 34th Annual Hawaii International Conference on System Sciences, January 2001, 9 pp. (2001)
8. Crowston, K., Wei, K., Howison, J., Wiggins, A.: Free/libre open-source software development: what we know and what we do not know. ACM Comput. Surv. **44**(2):7:1–7:35 (2008)
9. Šmite, D., Wohlin, C., Gorschek, T., Feldt, R.: Empirical evidence in global software engineering: a systematic review. Empir. Softw. Eng. **15**(1):91–118 (2010)
10. Ramasubbu, N., Cataldo, M., Balan, R.K., Herbsleb, J.D.: Configuring global software teams: a multi-company analysis of project productivity, quality, and profits. In: Proceedings of the 33rd International Conference on Software Engineering, ICSE'11, pp. 261–270. ACM, New York (2011)

11. Scacchi, Walt, Feller, J., Fitzgerald, B., Hissam, S., Lakhani, K.: Understanding free/open source software development processes. Softw. Process: Improv. Pract. **11**(2), 95–105 (2006)
12. Archak, N.: Money, glory and cheap talk: analyzing strategic behavior of contestants in simultaneous crowdsourcing contests on www.topcoder.com. In: Proceedings of the 19th International Conference on World Wide Web, WWW'10, pp. 21–30. ACM, New York (2010)
13. Bacon, D.F., Chen, Y., Parkes, D., Rao, M.: A market-based approach to software evolution. In: Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications, OOPSLA'09, pp. 973–980. ACM, New York (2009)
14. DiPalantino, D., Vojnovic, M.: Crowdsourcing and all-pay auctions. In: Proceedings of the 10th ACM Conference on Electronic Commerce, EC'09, pp. 119–128. ACM, New York (2009)
15. Towne, W.B., Herbsleb, J., MllerBirn, C.: The vista ecosystem: current statusand future directions. Technical report, Institute for Software Research, Pittsburgh, PA (2010)
16. Ye, Y., Kishida, K.: Toward an understanding of the motivation open source software developers. In: Proceedings of the 25th International Conference on Software Engineering, ICSE'03, pp. 419–429. IEEE Computer Society, Washington (2003)
17. Mason, W., Watts, D.J.: Financial incentives and the "performance of crowds". In: Proceedings of the ACM SIGKDD Workshop on Human Computation, HCOMP'09, pp. 77–85. ACM, New York (2009)
18. Bled Electronic Commerce, Jsang, A., Ismail, R. The beta reputation system. In: Proceedings of the 15th Bled Electronic Commerce Conference (2002)
19. Elo, A.: The Rating of Chessplayers, Past and Present. Arco Publishing, New York (1978)
20. Herbrich, R., Graepel, T.: Trueskilltm: a bayesian skill rating system. Technical report (2006)
21. TopCoder Inc., Algorithm competition rating system. Technical report (2008)
22. Wenjun, W., Tsai, W.-T., Li, W.: An evaluation framework for software crowdsourcing. Front. Comput. Sci. **7**(5), 694–709 (2013)
23. Corchn, L.C.: The theory of contests: a survey. Rev. Econ. Des. **11**(2):69–100 (2007)
24. Nag, S., Heffan, I., Saenz-Otero, A., Lydon, M.: Spheres zero robotics software development: lessons on crowdsourcing and collaborative competition. In: 2012 IEEE Aerospace Conference, March, 2012, pp. 1–17 (2012)

Part III
Software Crowdsourcing Systems

TRUSTIE: A Software Development Platform for Crowdsourcing

Huaimin Wang, Gang Yin, Xiang Li and Xiao Li

Abstract Software development is either creation activities that rely on developers creativity and talents, or manufacturing activities that follow the engineering processes. Engineering processes need to include creation activities to address tasks such as requirement elicitation and bug finding. On the other hand, by exploiting the crowd wisdom, open-source development has been demonstrated to be a suitable environment for software creation. However, it also has several limitations, such as guaranteeing the progress and quality of production process. This paper introduces a software development platform and ecosystem that combines the strengths of the two models. First, we propose the Trustworthy Software Model as a basis to support such a hybrid development ecosystem. The core of this model contains a novel lifecycle model, an evidence model and an evolution model. Second, based on the model, we propose the Trustworthy Software Development and Evolution Service Model. It integrates *crowd collaboration*, *resource sharing*, *runtime monitoring*, and *trustworthiness analysis* into an unified framework. Based on this integrated model, we designed and implemented TRUSTIE, which distinguishes itself from other software crowdsourcing platforms by providing the software collaborative development service and the resource sharing service with the general support of trustworthiness-analysis tools. TRUSTIE enables crowd-oriented collaboration among internal development teams and the external crowds by combining the software creation and software manufacturing in one ecosystem.

H. Wang · G. Yin (✉) · X. Li · X. Li
National Laboratory for Parallel and Distributed Processing, School of Computer, National University of Defense Technology, Changsha 410073, China
e-mail: jack_nudt@163.com

H. Wang
e-mail: whm_w@163.com

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_10

1 Introduction

Software development is an intellectual activity [1]. During the early phases of the software development, most of works are creative activities where people work together to analyze requirements and design software. Once the initial specification or design is available, automated algorithms are available to perform analysis for producing quality code, such as completeness and consistency checkers, automated code generators, and test case generators. The later processes, often rigorously, may be considered as a software manufacturing process. In spite of significant progress in software technology, many steps of software development processes are still manual. For example, requirement elicitation and bug removal [2] are mostly creative tasks in which automation plays a very limited role. Encouraging and facilitating creative activities are very important.

1.1 Lessons from Open-Source Software Development

Recently, Open-Source Software (OSS) has significantly changed our understanding of software development. Since the 1980s, OSS has continued to grow in both quality and quantity, and has become a source of software for numerous organizations. OSS development is different from traditional software development in several aspects: teams are decentralized; resources are rapidly shared; new versions are frequently released; and online communities of developers have always been formed. These characteristics enable people to create software in a distributed and collaborative manner. For example, OSS websites like *Github*, *Google Code* and *Sourceforge* make it possible for anyone to create and manage OSS projects at any time. Besides, OSS projects are open to all the developers. For example, users from all over the world, regardless of their prior training or experience, can engage in design discussion, contribute their code, and engage in testing through bug reporting. Thus, software development is greatly facilitated through this openness and massive crowd participation. Software development will profit greatly from an effective ecosystem empowered by *crowd wisdom*.

1.2 Crowd Wisdom

In this paper, "crowd" means "an undefined large group of people" [3]. For example, in Wikipedia, there are more than 19 million registered user accounts,[1] who have edited more than 30 million pages.[2] Their accuracy was found to be similar to the Encyclopedia Britannica [4]. Linus Torvalds, creator of the Linux Open Source

[1] http://en.wikipedia.org/wiki/Wikipedia:Wikipedians.

[2] https://en.wikipedia.org/wiki/Special:Statistics.

Operating System, said that "the most exciting developments for Linux will happen in user space, not kernel space" [5], in which "the user space" is the environment where a large number of people contribute their code. The Mozilla OSS project, which produces the famous Firefox browser, has gathered a crowd of over 1,800 people as acknowledged contributors.[3]

Software creation activities are now becoming an active arena for crowd wisdom. The success of this transformation is evidenced by the above-mentioned and other successful OSS projects. The insight nature of this success can be explained by the "wisdom-of-crowds" effect in cognition, coordination or cooperation problems [6]. The aggregated performance of a crowd will often outperform any single elite or small team of elites. Software creation tasks, such as eliciting requirement, negotiating the design of modules, finding and fixing bugs, are indeed cognition, coordination or cooperation problems. In traditional software engineering, these innovative tasks are assigned to dedicated teams, and are performed under a central control. However, as reported in [7], innovation and knowledge are essentially distributed and can hardly be aggregated by using centralized models. In OSS projects, software creation is outsourced to an open crowd, where massive, diversified, and non-professional contributions converge to the diffusion of innovation, resulting in the rise of wisdom-of-crowds revolution in software development.

1.3 Ecosystem Incorporates Engineering and Crowd Wisdom

By exploiting crowd wisdom, OSS development can alleviate the problems encountered in software creation which are hard to tackle by engineering methods. However, crowd wisdom method is not intended for all scenarios. The majority of commercial or industrial software systems are still developed through traditional engineering methods, though with the addition of agile elements recently. This is due to the reason that the engineering methods and tools have central control over requirements management, progress scheduling and quality assurance. Crowd wisdom is essentially elusive and unpredictable. Without central control, OSS development can hardly guarantee anything ahead of time, which is intolerable for most commercial products.

For the above reason, we do not advocate that the crowd wisdom method should replace the engineering method. Instead, we propose that these two paradigms should be combined, so that traditional software production can benefit from crowd wisdom. The end of this reasoning coincides with the business strategies of big companies, such as IBM, who embraces open source to benefit its software business [8]. However, we take a different perspective as development platform designers in understanding this end.

Our approach is to establish a software ecosystem that incorporates engineering methods and crowd wisdom. In the ecosystem, software creation activities are

[3] http://www.mozilla.org/credits/.

well-supported by exploiting crowd wisdom; meanwhile software manufacture is well-supported through implements of engineering methods. While crowd wisdom methods stress more in respecting the creativity of each individual [9], an important goal of engineering is the quality or trustworthiness of the software system. A key challenge in bridging these two types of development methods is to ensure the quality or trustworthiness of a software system up to an industrial standard and meanwhile respecting the creativity of each member of the crowd.

In this paper, based on how software systems are actually evolved in crowd-based development practices, we adopt a *crowd-based approach*, by proposing a Trustworthy Software Model (TSM) for quality assurance of the new ecosystem with a platform. We propose a new Software Development and Evolution Service Model (SDESM) that offers *crowd collaboration, resource sharing*, *runtime monitoring* and *trustworthiness analysis* as four basic services. Based on the TSM and SDESM, we implemented TRUSTIE (**Trust**worthy software tools and **I**ntegration **E**nvironment), a software development platform.

This paper is organized as follows: Sect. 2 describes the TSM; Sect. 3 proposes the SDESM. Section 4 introduces TRUSTIE including its architecture and application practices; Sect. 5 covers related work; The last section concludes this paper.

2 Trustworthy Software Model (TSM)

In the crowd-based software development paradigm, the meaning of software trustworthiness is fundamentally different from that of the traditional software. It is the foundation for designing a software development ecosystem built upon crowd wisdom.

2.1 Life Cycle Model

In the traditional lifecycle model, software consists of program and documentation, and it has two phases: development phase and application phase [10], as shown in Fig. 1a. After completing the development, a software system enters the application phase through a distribution or releasing process. Any user feedbacks are returned to the development team for software update for the next release.

In crowd-based development, huge amounts of software data can be generated at different phases of software life cycle. These data are accumulated, and shared in various developer communities, which not only reflect software functionality, but also can be used to analyze and measure various software properties. Take OSS as an example. The software data are spread mainly in collaborative development communities (such as *SourceForge* and *GitHub*) and knowledge sharing communities (such as *StackOverflow*). The former consists of software repositories that can be used to analyze the quality of codes and software development processes, while the latter

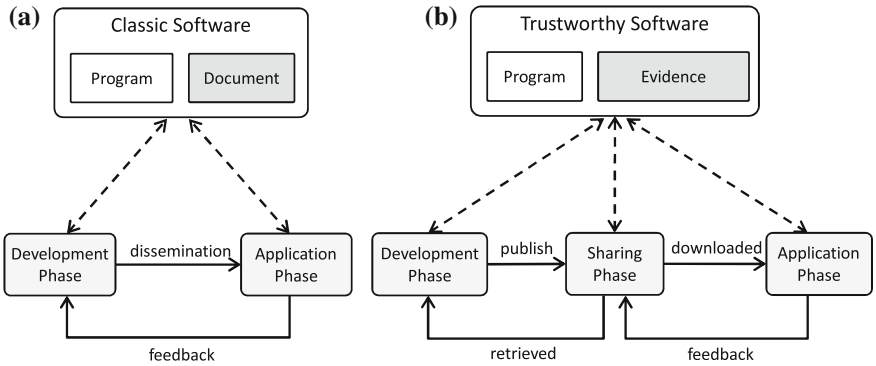


Fig. 1 Lifecycle models of **a** classic and **b** Trustworthy Software

can be used as the textual feedbacks from the crowds (such as comments, Q&As). The corresponding life cycle model is shown in Fig. 1b.

In this new model, the contents of *software* are code and evidences, and specifically, evidences as specific data that provide useful facts about various software properties. A software life cycle is now extended into three interwoven phases: development, sharing and application. The sharing phase is important for the collection and exploitation of crowd wisdom. When early versions of software (e.g., alpha versions) enter the sharing phase through releases, they can be downloaded, tested, analyzed, and assessed by the public. All user-generated data are then fed back to the development team, so the developers can identify defects and possible extension points.

Based on this model, a software entity has different forms in the three phases: software project, software resource and software instance. A *software project* contains a set of software artifacts and development data generated by developers with a roadmap. A *software resource* contains a set of software programs and evidences published by its provider. A *software instance* contains a set of running programs, status and application data of an online software system. The separation of the three forms of software entities will make it clear how to support software development in different software lifecycles.

2.2 Trustworthy Evidence Model

Software trustworthiness evidences are structured or unstructured data that can directly or indirectly reflect trustworthiness attributes. Here, attributes include not only *objective* quality attributes like correctness, reliability, performance, safety, security [11], but also *subjective* attributes such as user evaluations. In some cases, meta-attributes calculated by comparing different quality attributes can also be used as evidence. In different phases of a software life cycle, different trustworthiness

evidences can be generated. The trustworthiness evidence model mainly contains three types of software evidences:

Development Evidences are evidences produced in the development phase, including measurements and descriptions that reflect various attributes of a software artifact, development process and teams. Examples are source code quality, bug fix time, and measurements on the other development activities.

Sharing Evidences are evidence produced in the sharing phase, including community-based measurements like number of downloads and followers, measurements of project activities, and ranking results. Community evidences are data generated from online sharing platforms. Thus, they are indirect attributes of software systems. These reflect the social attributes of a software system.

Application Evidences are evidences produced in application activities, including measurements and assessments given by users that reflect either the quality (availability, reliability, and security) or the functional and performance features (usability and maintainability) of a software system.

Different kinds of software entities contain specified sets of software evidences. A software project generally contains all development evidences of a software system. Software resources usually include all the sharing evidences of a software system, relevant parts of the development evidences and application evidences. A software instance contains all application evidences of an online software system.

2.3 Software Evolution Model

Software evolution is a continuous process of modifications to meet application requirements. It is an essential way towards producing quality software systems in crowd-based development. During the software evolution process, developers modify and upgrade the software system based on existing trustworthiness evidences. Evolution activities will also produce new trustworthiness evidences. Based on the new software lifecycle model mentioned in previous subsection, evolution activities fall into the following three categories: version evolution, resource evolution, and runtime evolution. Each corresponds to the development, sharing and application phase respectively. Given a specific software system, a version evolution can generate multiple instances of resource evolution that in turn generates multiple instances of runtime evolution as shown in Fig. 2a.

Version Evolution is the continuous source code evolution of a software system during the development phase. This includes the design, coding, testing and release activities carried out in the face of changing requirements, and different development processes might be involved. Version evolution activities produce development evidences and it has a dependency on trustworthiness evidences produced by resource evolution and runtime evolution activities.

Resource Evolution is the change of software trustworthiness attributes directly or indirectly caused by continuous updates of software trustworthiness evidences.

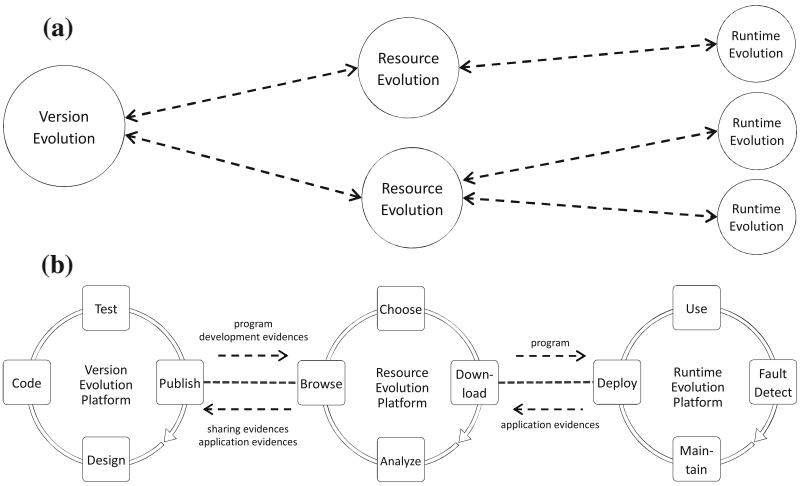


Fig. 2 Three evolution models: **a** relations between three kinds of evolution. **b** The internal actions in software evolution loops

The basic steps of resource evolution include the update of programs, evidences and the recalculation of trustworthiness attributes.

Runtime Evolution is the change of software runtime activities including software update, application deployment, system maintenance, error handling and so on. Systems of different types or different scales evolve differently in their application phase. Runtime evolution provides trustworthiness evidences generated in software systems' application phase to resource evolution activities.

There are complicated mutual impacts and restrictions between different evolution activities. Malfunction of any type of evolution activity can cause serious negative or harmful effect to the development of the software system. For example, the more runtime evolution instances, the more trustworthiness evidences they can provide to resource evolution instances and the quicker version evolution activities such as bug fixing, hence the faster the maintenance and improvement process. Version evolution is the original driving force of other evolution activities; malfunctioned version evolution activities like poor project management might lead to software runtime failure and sometimes force teams to start new version evolution activities. Besides, resource evolution should focus on the aggregation, sharing and analysis of software evidences. Suitable resource evolution mechanisms should be designed to attract participations of software vendors and users.

Under specific circumstances, software vendors might directly offer their software system as online services, which to some extend brings the above three types of evolution activities together. This integration is meaningful to the improvement of the overall efficiency and quality of crowd-based software evolution activities.

3 Trustworthy Software Development and Evolution Service Model

The TSDESM (Trustworthy Software Development and Evolution Service Model) is an Internet-based software evolution model. It supports software evolution activities in the development, sharing and application phases, and the formation, gathering, sharing and utilization of trustworthiness evidences. It provides an evolution-based approach for crowd-based software development paradigm. As shown in Fig. 3, this service architecture centers on *trustworthiness analysis*, offers *crowd collaboration, resource sharing* and *runtime monitoring* as basic services. Specifically, the crowd collaboration service supports crowd creation and manufacture, and provides mechanisms for the integration or transformation between the two. The resource sharing service provides mechanisms, like entity sharing and evidence sharing, to achieve rapid prototype distribution and application feedback. The runtime monitoring service provides services of data gathering, aggregation and analysis service during the runtime. The trustworthiness analysis service is responsible for measuring and analyzing the data generated in the crowd collaboration service, runtime monitoring service and the resource sharing service. Meanwhile, it provides comprehensive analysis mechanisms for various crowd-based development tasks.

The basics of this architecture are the massive amount of trustworthiness evidences generated in different software evolution processes. The four services not only output different types of trustworthiness evidences, but also establish and optimize some of their own functionalities by utilizing trustworthiness evidences, as depicted in Fig. 3. Specifically, collaboration development activities are supported by the crowd collaboration service. The development process data they produce are the main source of development evidences, including version repositories, code commit logs, and

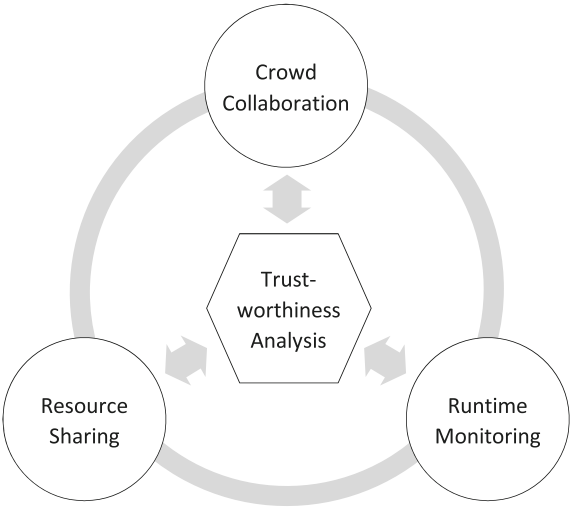


Fig. 3 Trustworthy software development and evolution environment architecture

issue tracking repositories. The resource sharing service continuously aggregates and accumulates trustworthiness evidences from the crowd, including test reports, use case description and comments. The runtime monitoring service outputs behavior data of software instances, such as running log, which is important for evaluating and improving the runtime trustworthiness. The trustworthiness analysis service is responsible for measuring, analyzing and evaluating various evidences, providing developers and users with important statistical results and analysis tools.

Besides, the four services are not isolated. They provide services to each other through open interfaces. For example, through the interfaces provided by the resource sharing service, the crowd collaboration service can recommend useful software components and services to programmers to facilitate agile development. The resource sharing service can also call the interfaces of the trustworthiness analysis service to get the trustworthiness evidences of a certain software resource. The crowd collaboration service uses these interfaces to evaluate on-going development activities in the code quality and development efficiency. And the runtime monitoring service can publish logs of critical system faults to resource sharing service, which in turn publishes these log data to crowd for possible solution.

3.1 Crowd Collaboration Service

Software is the virtualization of real world objects and the incarnation of knowledge and wisdom. All software development activities are essentially a kind of knowledge-intensive collaboration activities [12, 13], be it software manufacture or software creation. However, these two types of collaborations are different in many aspects. Collaborations in software manufacture activities are conducted by a closed team of developers, while collaborations in software creation often involve the external crowds. This difference entails different mechanisms and tools for development, interaction and rewarding systems.

The goal of the crowd collaboration service is to support the integration or transformation of software creation and software manufacture. As an indispensable procedure in software creation activity, it means to make adaptation of selected works of creation and integrate them into products of manufacture. Online communication and sharing tools like BBS, blog, wiki, micro-blog, which can support collaborations of a large crowd, are important for software creation activities. These tools can help disseminate creative ideas and works in a short period of time, attracting more potential users and contributors. Meanwhile, tools used in software manufacture are mainly aimed at improving the level of development automation and better process management. These include tools of desktop development, version management, process management, bug management and so on.

The crowd collaboration service should provide the development tools of both types and the mechanisms for their integration. These mechanisms can resolve their inter-dependencies and conflicts. Besides, these mechanisms should be flexible and adaptable, especially for projects which adopt engineering management on core

members and crowd management on peripheral contributors. Here we propose three reference models.

Community Model is the mechanism that supports communication and sharing among a group of individuals with similar interests. Community tools include BBS, blog and wiki and so on. The community model integrates community tools with software manufacture tools, builds development communities around development processes or software artifacts. For example, developers and users can create a community for sharing and communicate development activities based on a certain software issue. The community mechanism supports and encourages individual developers to be creative and in turn inspires crowd wisdom. See Fig. 4a.

Social Network Model is the mechanism for maintaining social relationships among users. Basic elements of a social network are relations like "follow" , "friend" and "group". Users broadcast their dynamics to other relevant users through the social network. Often, there are complicated social relations among software developers, such as code committer network [14] and bug report network [15]. Social network tools are mainly responsible for maintaining social relationships in software development activities, achieving mutual awareness and seamless interaction. As depicted in Fig. 4a.

Process Organization Model is the mechanism for organizing and reusing efficient collaboration processes and tools for software development activities with a relatively stable work flow. Software production line is a new network-based software development environment which is integrative, extensible and collaborative [16]. Following a given development procedure, this new environment can organize and customize software development elements related to developers, tools and artifacts.

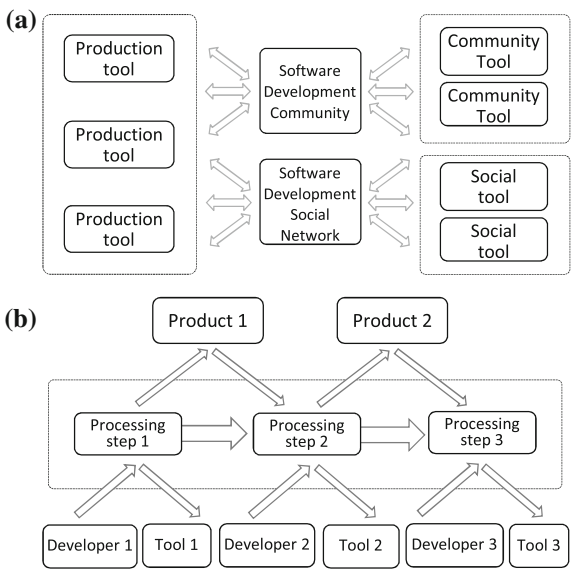


Fig. 4 Network-based crowd collaboration service: **a** the two integration models for software creation and production. **b** The tool integration model for development progress organization

In this way, it can easily customize a dedicated software development environment for a particular team of developers. As depicted in Fig. 4b.

3.2 Resource Sharing Service

During the evolution of a software project, developers and users create various artifacts, tools and data. These resources are valuable to reuse and reference in later development. Resource Sharing is an essential utility for both software manufacture and software creation. For software manufacture activities, sharing of artifacts, processes and information within a certain closed-team project should be supported and encouraged in the platform. On the other hand, to attract a larger crowd to participate in software creation activities, the platform should also encourage and remind core developers to share tutorials and example codes. Oftentimes, resources and knowledge shared in crowd development are not well-structured documentations and goes beyond the boundary of any project or team. This implies that the platform should have free sharing utility for the crowd to upload, edit, mark, comment and vote for or against various kinds of resources.

The recourse sharing service is the major platform for software resource evolution. It can provide software program and evidence sharing utility to different groups of developers. The challenge here is to continually collect and store massive amounts of software resources. Two mechanisms are introduced in this service.

Program Sharing Mechanism supports publishing, accessing and updating of software components, software services and other types of software programs. For software services, the software entity data also include the interface descriptions and URL addresses of each service instance. For open source software, software entity data often include source code, compilation or installation scripts and the address of the source code repository. The software program sharing mechanism can accelerate software system's distribution and dissemination. In other words, it accelerates a software instance's transformation from version evolution to runtime evolution, speeding up the exposure of software bugs and other issues.

Evidence Sharing Mechanism supports publish, access and update of software trustworthiness evidences generated in the development, sharing and application phases. For software services, their evidences also include service instances' real-time availability status and operation status. For open source software systems, the evidences include bug repositories, mailing lists, open source licenses, sponsors' information and the activeness of development. The software evidence sharing mechanisms can speed up software evidences' dissemination and update; shorten the response time on user feedbacks.

Currently, software resources are widely dispersed over various online software resource sites. Take open source software resource sties as an example. They include software community sites (e.g., *Linux*, *Apache* and *Eclipse*), project hosting sites (e.g., *SourceForge* and Github), software directories (e.g., *Softpedia.net* and *Ohloh*) and programming Q&A websites (e.g., *StackOverflow* and *AskUbuntu*). These sites

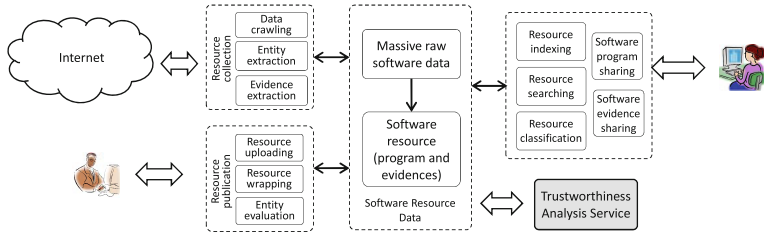


Fig. 5 Core mechanisms of the software resource sharing service

contain huge amounts of publicly accessible software programs and various types of trustworthiness evidences. These data are of different formats and their organizations are different from one site to another, so it is challenging to discover quality software resources (Fig. 5).

There are several techniques that can be used to collect massive quality software resources. These include:

Resource Publication allows users to register and submit various software entities and their trustworthy evidence to the resource sharing platform. Different types of software resource data are organized and wrapped by a unified structure, e.g., the RAS mechanism [17].

Resource Collection crawls and preprocesses software resources from various online software resource platforms. It is an important automatic technique for establishing the large-scale software resources sharing service.

Resource Organization and Mining supports effective classification and retrieval of massive software resources, and mines the data for software evaluation and analysis based on trustworthiness evidences.

3.3 Runtime Monitoring Service

The runtime monitoring service is an infrastructure for achieving trustworthy runtime evolution in the application phase. The idea is to monitor the behavior and status of software instances, and provide raw or filtered runtime log data for trustworthy analysis service. By providing such information, the runtime monitoring service can support fault localization and dynamic modification, and eventually support trustworthy running of software systems.

The runtime monitoring service can be implemented in three modules as shown in Fig. 6.

Monitoring Development Tool transfers normal software into monitoring-enabled software. A general transformation approach is to insert monitoring probes into the targeted software system. Specifically, the monitoring-enabled transformation process contains several sequential phases, including monitoring requirement description, monitoring probes generation, monitoring probes insertion and

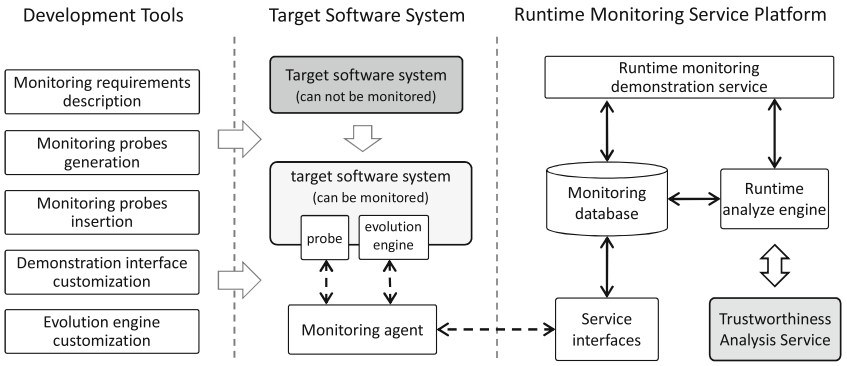


Fig. 6 The model of software runtime monitoring service

software interface customization. In addition, the monitoring development tool can also support dynamic adjustment by deploying evolution engine into a software system. However, the implementation of evolution engine needs API support of targeted software system.

Targeted Software System contains three general components. The monitoring probes send system runtime status to a monitoring agent which deliveries these information into the runtime monitoring service platform. For software systems that change frequently, the targeted software system can deploy an evolution engine to implement dynamic adjustment. In details, the evolution engine can either execute evolution commands sent from the monitoring agent, or automatically adjust system based on local monitoring data.

Runtime Monitoring Service Platform is a system-level remote service that supports monitoring of several targeted software systems. It contains APIs of monitoring service, runtime analysis engine and runtime monitoring demonstration service and so on. The monitoring database is responsible for storing raw monitoring data and recognized faulty event data. For very large complex software system, the monitoring database also needs mechanism to process stream data and the capacity of massive data storage. The runtime analysis engine conducts data mining on monitoring data through calling the APIs of trustworthy analysis service, to evaluate system running status, diagnosis of system faults, and update monitoring database. The monitoring agent can send monitoring data, acquire fault information or issue adjustment commands through accessing the APIs of monitoring service.

The runtime monitoring service can be regarded as a new service provided to software system running in the network. By doing so, this not only simplifies application logic of targeted software system, but also unleashes the advantage of data mining to improve the effectiveness of monitoring.

3.4 Trustworthiness Analysis Service

Trustworthiness analysis measures and evaluates various development behaviors, software artifacts and components by mining massive software evidence data. Resources and knowledge generated by software creation activities are neither well-managed nor well-organized, thus posing challenges for effective data mining. For example, in crowd wisdom methods, most requirements are hidden in comments and discussions informally created by the crowd. Thus, data from crowd have to be preprocessed and analyzed to be properly reused and referenced in software manufacture activities. Trustworthiness analysis service is the key to disclose and refine them, making the transformation from creation to manufacture possible. Besides, for both engineering methods and crowd wisdom methods, analysis is also crucial for understanding development status and evaluating development problems. Finally, results of analysis often act as useful references for optimizing future development. For example, the analysis and monitoring of user feedback has become the norm to evaluate existing software features and extract new requirements [18].

In the proposed software model, trustworthiness analysis service is the fundamental mechanism that makes trustworthy evolution during the development and sharing phase possible. It measures and assesses development behaviors, software artifacts and software products by analyzing and mining massive software evidence data, thus leading software evolution activities towards the desired direction. The trustworthiness analysis service contains three core mechanisms, i.e., development data analysis (for development evidences), runtime data analysis (for trustworthiness evidences) and resource trustworthiness rating (for runtime evidences).

Development Data Analysis consists of mechanisms which analyze software development data (including process data and work-in-process) to measure and assess software development activities and ultimately help improve project development efficiency and software product quality. The core model of these mechanisms is given in the left triangle of Fig. 7. The task of development evidence extraction is to extract relevant development data from the software project environment. This process in turn imposes new requirements on the reorganization or adjustment of software project process data. The development data analysis mechanism is to analyze

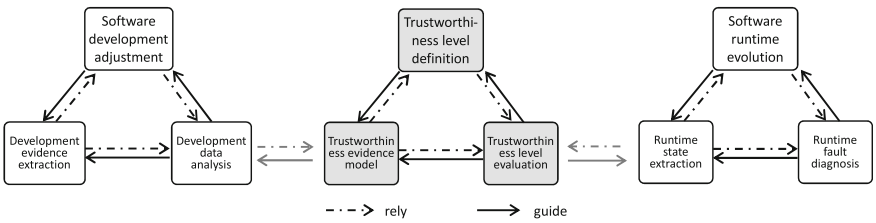


Fig. 7 The core model of trustworthy software comprehensive analysis (from *left* to *right*): **a** development data analysis model of software projects. **b** Trustworthiness evaluation model of software resources. **c** Runtime data analysis model of software instances

and measure the extracted data. This helps assess development status and identify software problems, providing important statistics for development optimization.

Resource Trustworthiness Evaluation gives the estimated rating of the target software resource for the given trustworthiness attribute, by analyzing and evaluating the evidences generated in the three phases of software life cycle. The core model of these mechanisms is depicted in the central triangle Fig. 7. The rating of software trustworthiness is based on the trustworthiness rating requirement model, which can be established according to users' expectation. Trustworthiness evidence model is the set of pertinent evidences determined by the definition and assessment process of trustworthiness rating. It is the basis of software trustworthiness rating. Trustworthiness rating assessment is the method and mechanism which rates software entities' trustworthiness, which is often domain specific. An example of the trustworthiness rating from TRUSTIE can be seen in Sect. 4.3.

Runtime Data Analysis analyzes and mines the application evidences generated in software runtime to evaluate system running status and diagnose faulty. Bo doing so, we can dynamically adjust software system to achieve trustworthy running and evolution of software. The core model of these mechanisms is given in the right triangle of Fig. 7. The runtime state extraction component extracts and analyzes important evidences related to the analyzed targets, including application independent system-level evidences and application dependent logic-level evidences. The runtime fault diagnosis component mainly analyzes and localizes evidences, and diagnoses software runtime faults. The recognized faults can be taken as runtime evolution evidences, and be published into resource sharing platform as application evidences.

For the construction of any specific trustworthy software development environment, the software project data analysis model and resource trustworthiness rating model in Fig. 7 are widely applicable. Software development data can be intermediate software artifacts like source code files or executable software modules. They can also be procedure logs of a specific project task like development logs, issue lists and mailing lists. The software trustworthiness rating model can explicitly give a software system's trustworthiness rating and its definition. It may also give an unsupervised ranking requirement according to some trustworthiness attribute. The measurements and descriptions produced in the software project data analysis process are important evidences of the software development phase.

4 TRUSTIE: Software Production and Evaluation with Crowdsourcing

Based on the trustworthy software development service model, we designed and implemented TRUSTIE (**Trust**worthy software tools and **I**ntegration **E**nvironment), a platform for software production and evaluation with crowdsourcing (www.trustie.net).

The goal of TRUSTIE is to help internal development teams and external crowd developers to improve the quality and the productivity of software. To achieve this goal, TRRUSTIE uses software crowdsourcing to bridge the external crowd wisdom and internal engineering by using various contributions from the crowds, which are tasks that can be performed distributedly beyond the internal development team for the software projects. In TRUSTIE, any software development tasks can be outsourced in an implied manner. Even the evaluations of the submitted contributions are possible to be outsourced. The organizers of the outsourcing mechanism are mainly the internal teams of the software projects (Fig. 8).

TRUSTIE employs five kinds of technologies, achieving the key mechanisms of bridging the external crowd wisdom and internal engineering process, supporting central control, decentralized contract, and three application models. The platform has developed about 60 software tools covering software development activities including requirement engineering, design, packaging and maintenance. We have designed the system of software collaborative development analysis and the system of software product-line framework. Based on the former system, we developed four product-line systems for automatic software production. We also achieve the integration of the collaborative development core service with those systems. The technologies and platform of TRUSTIE have been used in China in various software industries and research institutions.

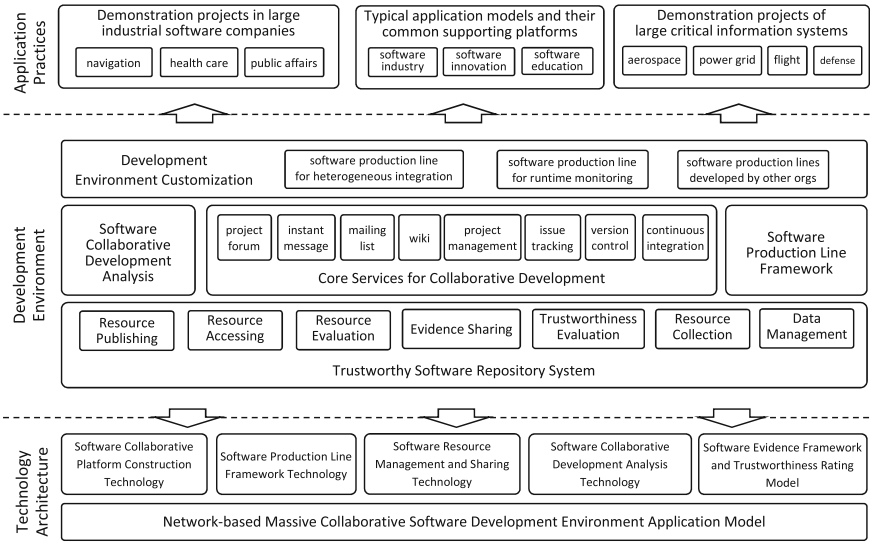


Fig. 8 The development environment, technology architecture and application practices of TRUSTIE

4.1 Software Crowdsourcing Model and Process in TRUSTIE

Based on the theoretical framework proposed in [19], the software crowdsourcing model in TUSTIE can be categorized as a non-competitive model. In TRUSTIE, either individuals or development teams can participate in a *collaborative* manner to create software. All the participants contribute code or resources throughout the entire crowdsourcing process for better connection of internal software teams and external Crowds. This is quite different from the competitive model (such as crowdsourcing platforms in *TopCoder* and *AppStori*), where people participate in a *competitive* manner to create software. Only selected or highly talented developers (or teams) can survive in the crowdsourcing process and become the only contributors or be rewarded with funding.

As a non-competitive model, the process of software crowdsourcing in TRUSTIE is to ease the collaboration burden and maximize the throughput of development outcome. This is supported by the collaborative development service and resource sharing service (explained later in Sects. 4.2 and 4.3) in TRUSTIE in an implied style. Generally, TRUSTIE includes three crowdsourcing processes: (1) Development outsourcing: TRUSTIE outsources the task pieces of software creation and production to the crowds by using its network-based crowd collaboration service. Currently, TRUSTIE is considering outsource the codes review tasks [20]. (2) Testing outsourcing: TRUSTIE enables the collection and integration of public bug reports and comments by software resource sharing service. (3) Maintenance outsourcing: TRUSTIE enables the collection of runtime status of monitored systems for quick runtime evolution. This may enable the outsourcing of system maintenance tasks by recruiting skilled system administrators. Currently, these processes are mainly driven by the interests and consumption requirements of the crowds.

The collaborative development service and resource sharing service are the keys to support the non-competitive crowdsourcing model in TRUSITE. In the next two subsections, we describe the two services in details.

4.2 Trustworthy Software Collaborative Development Service

The trustworthy software collaborative development service supports the crowdsourcing process with software creation tools and software manufacturing tools. These tools are exposed to both internal development teams and external crowd with configurable options. Their integration is achieved through the community model and social network model. Based on this, we build the collaborative development analysis system that can analyze and measure development behavior, and the software product-line framework system that supports the organization of collaboration process and the customization of the development environment. Specifically, it consists of the following services:

Social Collaborative Development Service: This service provides software creation tools like project forums, collaborative editing tools, mailing lists, and instant communication. It also provides manufacture tools like project management, configuration management, bug management and continuous integration tools. It combines both kinds of tools into a unified development environment [21]. Besides, this service supports the sharing of technical knowledge, and achieves developers' mutual awareness, and interaction through mechanisms applied in the community model and the social network model.

Collaborative Development Analysis Service: this service has provided a platform to fetch and store massive software development data. It has also integrated various tools for measuring development behaviors into the configuration management tool set. Behind this service is a comprehensive assessment technology which concerns the evidences of software products and evidences of development teams simultaneously [22]. Besides, it integrates a technology that analyzes a programmer's development capability and the traces of his or her technical improvement [23].

Software Product-Line Construction Service: specifically, this service organizes and customizes developers, software tools and software artifacts involved in a specific software development process based on some given development steps. In this way, the service helps establish suitable software production lines and a dedicated development environment for the developer team [24].

As of June 2013, TRUSTIE supported 600 software projects and 700 competition projects. Besides, by using a set of evidence standards, it has extracted the development evidences of quality OSS projects from online OSS communities.

4.3 Trustworthy Software Resource Sharing Service

In TRUSTIE, the trustworthy software resource sharing service supports the crowdsourcing process by allowing users, either internal development teams or external crowd, to publish, retrieve, and evaluate software resources. It can transfer the traditional *closed static* software component storage model into the *open dynamic* software resource sharing model [25]. Combined with the software rating and evaluation model [26], this mechanism integrates trustworthiness evidence framework into the software resource information architecture [27]. It seamlessly integrates central, static resource storage with open, dynamic resource aggregation technology.

Resource Collecting and Organization: this service collects massive software resources by using the resource publishing and resource retrieval technology. Based on the RAS technology standard, it achieves evidences' packaging and utilization throughout the whole software life cycle. In order to improve the effectiveness of resource management, it classifies massive software resources from multiple sites using text mining and tag mining technology [28] and feature analysis technology [29]. Both extracted meta data and mined knowledge data is stored and indexed in cloud-based persistent storage including relational and non-relational databases. Up to June 2012, TRUSTIE has already published 170 self-developed software

resources and more than 61,000 software resources collected from other online software libraries and open source communities.

Resource Trustworthiness Evaluation: this service provides automatic assessment mechanisms and manual assessment mechanisms, e.g., the assessment mechanism for service trustworthy evolution [30, 31]. Though software trustworthiness rating depends on specific application domain, we believe there can be a trustworthiness rating reference model which captures universal software attributes. This reference model can be customized for any specific application domain.

The trustworthiness rating model employed by TRUSTIE's trustworthy software resource repository system is a trustworthiness rating reference method whose rating dimensions are from user expectation, application validation and the methodology of evaluation [26], see Table 1. For example, in some critical areas (like aerospace), even

Table 1 Software trustworthiness rating model

Level	Name	Content
0	Unknown	The lowest trustworthiness level. It means no software trustworthiness evidence is found. It cannot determine whether the target software system satisfies users' expectations of the trustworthiness attributes of the same type of software systems
1	Available	The software entity can be accessed, and can function as described by the software provider. It implies that the target software system satisfies users' basic expectations over the functional attributes of the same type of software systems
2	Verified	On the basis of the Available level, software provider publishes a declaration of the set of software trustworthiness attributes according to some documented standards. This declaration can be verified through domain-specific software assessment mechanisms. It indicates that the target software system satisfies users' general expectations over the trustworthiness attributes of the same type of software systems, and these user-expected trustworthiness attributes are verified
3	Applied	On the basis of the Verified level, software systems have been applied in related domains and have verifiable cases of successful application. It implies that the software system satisfies users' general expectations over the trustworthiness attributes of the same type of software systems, and these user-expected trustworthiness attributes have been verified by real application
4	Assessed	On the basis of the Applied level, trustworthiness of an Assessed software system should pass the assessment conducted by authoritative software trustworthiness rating agencies according to specific documented trustworthiness rating standards. This indicates the software system satisfies relatively higher user expectations over the trustworthiness attributes of the same type of software systems, and these user-expected trustworthiness attributes are confirmed by authorities
5	Proved	The highest trustworthiness level. It means on the basis of the Assessed level, the user-submitted software trustworthiness attributes can be strictly proved

the first applicable version of the target software system is required to reach a high trustworthiness level (e.g., level 4). The prerequisite of reaching level 3 and above is the evidence of successful case application of the software system. This requires the trustworthiness rating standards of these areas to be customized according to domain-specific descriptions. For example, aerospace software users can specify the definition of "successful cases of application" as "successful experimentation under simulated environment".

To ensure the reliability of trustworthy evaluation, the data assessed by TRUSTIE platform are aggregated automatically from the software tools in TRUSTIE, such as issue tracking tools, version control tools, resource sharing tools and communication tools, without any intervention.

5 Related Work

Throughout the development of software technology, software development technology and its supporting environment have always played important roles in driving software technological innovation and industrial development. The rise of crowd-based open source development has brought new opportunities. The vendors of software development environments have shifted their attention from providing local development support to facilitating globally distributed crowd development.

In software development methodology and development environment architecture, researchers have studied the changes in software development technology. Yang and colleagues [13] realized the profound impact of the Internet on software systems and software development activities. They systematically proposed the Internetware model and a set of architecture-centered software technologies and development methods, which pave the way for establishing the conceptual model, the evolution process and the supporting environment of trustworthy software systems. Based on Sourceforge and other similar open source project hosting platforms, Booch and colleague [32] have described the definitions and basic characteristics of software collaborative development environments. Using OSS development and community-based service systems as prototypes, Kazman and colleague [33] have proposed the Metropolis model which facilitates crowdsourcing software development. The Metropolis model adopts a hierarchical software development architecture, where participants are divided into the core, the peripheral and the mass. It addresses principles like openness, mash-ups, unknown requirements, continuous evolution, unstable resources and emergent behaviors. Through numerous case studies, Tapscott and colleague [34] have indicated that software development and more and more other business models have begun to adopt the ideas and mechanisms of mass collaboration, including openness, peering, sharing, and acting globally. Herbsleb [2] has proposed the concept of Global Software Engineering. He discussed what new challenges global software development imply in aspects including software architecture design, requirement elicitation, development environments and tools. Possible future research directions have also been identified and illustrated.

For the crowdsourcing model, Howe [3] has illustrated its origin, present status and future with several real world examples. He listed the open source movement, the development of collaboration tools and the rise of vibrant community as the keys to the rise of crowdsourcing, which is meaningful in establishing the crowd-based software development ecosystem.

For collaborative development tools and technologies, recent years have witnessed numerous researches on the analysis of distributed, social development technologies and practices. Mockus et al. [35] have analyzed the software development data of the Apache Server open source project, which has a major impact in software engineering research and has become the pioneer work in the field of Mining Software Repositories. Crowston and Howison [36] have examined 120 OSS projects hosted on Sourceforge.com. They have identified different patterns of developer interactions and their relation with the team size. Sarma and colleague [37]have proposed a browser named Tesseract for visualizing 'socio-technical' dependencies in development activities, which aims at increasing mutual awareness among developers of distributed teams. Dabbish et al. [38] analyzed Github.com, a project hosting and social development website, and how transparency plays an important role in developer collaboration in Github. Posnett et al. [39] analyzed the focus and ownership relations between software developers and artifacts in distributed development. They propose a unified view of measuring focus and ownership by drawing the predator-prey food web from ecology to model the developer-artifact contribution network. They found through empirical studies that the number of defects is related to the distribution of developer focus. Bird et al. [40] analyzed the development process of Windows Vista. Specifically, they compared the post-release failures of components that are developed by collocated teams with those developed in a distributed manner. The difference is found to be insignificant. More recent researches on collaborative development analysis tend to focus on empirical study and aim at making constructive suggestions and possible improvements on existing collaboration tools.

In OSS, resource sharing, mining, and trustworthiness evaluation become important issues. SourceForge, Github and other project hosting sites have accumulated a huge number of projects and data. In 2009, Mockus and colleague [41] have conducted a preliminary census of OSS repositories. Their results indicate that more than 120,000 and 130,000 projects were then hosted on SourceForge.com and Github.com, respectively. In 2013, these two metrics are reported to be 470,000 and 4,000,000 respectively [42]. With the wide application of OSS, researchers began to focus on the measurement of OSS trustworthiness. In its essence, software trustworthiness is the natural extension of the notion of software quality in the Internet era [13]. How to scientifically assess software quality has always been one of the most challenging issues of software engineering research [43]. After 40 years of development, Software Metric has become an important software engineering research direction concerning the problems of software quality. Quality assessment technologies have become specialized and standardized. Numerous impactful software quality models were then proposed [44, 45]. Based on these models, researchers and practitioners have further designed quality models that take community factors into consideration. These include the Navica [46], OpenBRR [47] and SQO-OSS [48] models.

For example, the OpenBRR (Open Business Readiness Rating) model is a mature OSS quality assessment model which aims to rate software projects and the code of the entire OSS community in a standard and open fashion and eventually facilitates the evaluation and application of OSS. Its assessment categories include Functionality, Usability, Quality, Security, Performance, Scalability, Architecture, Support, Documentation, Adoption, Community and Professionalism. Its assessment process involves ranking the importance of categories or metrics, processing the data, and translating the data into the Business Readiness Rating. For the moment, assessment and utilization of online OSS resources are still a hot topic for SE researchers.

In the industry, there is a major trend of the integration of software development environment with online collaboration tools. CollabNet[4] is one of the software vendors who intentionally integrate OSS development methods into software development environments. It has published the TeamForge platform which integrates configuration management, continuous integration, issue tracking, project management, lab management and collaboration tools into a Web app life cycle management platform. In this way, it supports distributed collaborative development and high quality software delivery. The Visual Studio Integrated Development Environment[5] is an enterprise IDE for desktop development environment. Recently it has added TFS (Team Foundation Server) that supports team collaboration mechanisms like version control, iteration tracking and the task panel. IBM Rational Jazz[6] is an open and transparent collaborative development platform for commercial use. The team collaboration, requirements composition and quality management tools of Rational Jazz can support the development of trustworthy software products. Besides, IBM has once encouraged the use of the IIOSB (IBM's Internal Open Source Bazaar) [8] in its commercial software development environment, which we believe is an important attempt to integrate software creation and manufacture.

There are research efforts on approaches to harnessing crowd wisdom for software development. Some emphasizes the importance of open, decentralized management. Bird and colleague [49] found that the development of Firefox project is distributed both geographically and organizationally. According to interviews with the creators of Linux, Perl and Apache [50], letting the crowd takes over is an indispensable step for the success of OSS projects. Project owners are thus encouraged to set up mechanisms and generate utilities for a larger crowd to participate easily, rather than act against this openness. However, decentralization comes at a cost. Compared to traditional centralized, co-located projects, this globally distributed OSS development model must face the challenge of incompatibilities and the risk of lack of awareness [2]. To harness crowd wisdom, software projects should be equipped with tools and practices that meet increasing coordination needs. The importance of accommodating diversities and conflicts has also been addressed in OSS practices and researches. A typical OSS project uses issue tracking systems to manage bug reports and feature requests. These bug reports cover various aspects of the target software project, and

[4] http://www.collab.net.

[5] http://www.visualstudio.com.

[6] http://www-01.ibm.com/software/rational/jazz/.

some of them are conflicting with each other. However, there are no dictators who make arbitrary decisions to cast aside any of these bugs. Instead, on which advice to take is totally for the whole community to decide. Those not taken are also kept in the project memory, and may have the opportunity to get re-opened [51]. Similar mechanisms can also be seen in the way developers manage their code contribution. For many successful Git-based OSS projects like the Android project and projects on Github, contributors do not have to always follow the central code depot. They can independently code on their own branch of the project, and merge the code back as a patch whenever they want [52]. To accommodate diversity, project managers are recommended to set up mechanisms to foster a culture that encourages different opinions. Besides, communication tools are needed to resolve conflicts and build consensus. For example, the *Stack Overflow* uses a voting mechanism to identify high quality posts.

Recently, crowdsourcing software development or software crowdsourcing was coined to identify an emerging area of software engineering [53]. It is described to be an open call for participation in any task of software development, including documentation, design, coding and testing. These tasks are normally conducted by either members of a software enterprise or people contracted by the enterprise. But in software crowdsourcing, all the tasks can be assigned to anyone in the general public. Many software engineering researchers have studied the concept models, processes and common architecture of software crowdsourcing. Wu et al. [54] has studied two famous software crowdsourcing platforms, *TopCoder* and *AppStori*. By mining the *TopCoder* data, authors found that the number of participants and hours spent on competition are surprisingly smaller than expected. Clear problem definition, transparency, diversity have been pointed out as the key lessons learned from current software crowdsourcing. For both software crowdsourcing platforms, the Min-Max nature among participants has been found to be a key design element. In another paper [19], the authors proposed a novel evaluation framework for software crowdsourcing projects. In the framework, the Min-Max relationship is used as a major aspect in evaluating the competitions of crowdsourcing projects. In a previous *Dagstuhl* Seminar [55], researchers from different domains have spent collective effort exploiting and validating the new idea of *Cloud-based Software Crowdsourcing*, where the software crowdsourcing processes and models are achieved with computer cloud support. Possible common architecture for *Cloud-based Software Crowdsourcing* has been identified. Important issues related to concept models, processes and design patterns have also been addressed. As discussed in the study of Tsai et al. [56], software crowdsourcing has enabled the synergy architecture between a cloud of machines and a cloud of humans. In such architecture, crowdsourcing models including game theory, optimization theory and so on would be well supported by cloud-based tools.

6 Conclusion

This paper proposes an ecosystem framework to deeply integrate the traditional engineering methodology and the crowd-based development process. Based on this framework, we develop the TRUSTIE, a non-competitive software crowdsourcing platform. It supports crowd collaboration, resource sharing, runtime monitoring, and trustworthiness analysis for trustworthy software evolution. TRUSTIE has been used successfully in a number of software companies in China since 2008. Our future work includes improving the evidence management and analysis capability of TRUSTIE through infrastructure upgrade, improving the collaborative development service and resource sharing service, and exploring the possibility of integrating competitive crowdsourcing models, such as creative works competition.

Acknowledgments This research is supported by the National High Technology Research and Development Program of China (Grant No. 2012AA011200), and National Natural Science Foundation of China (Grant No. 61432020 and 61472430). Our gratitude goes to all members of the TRUSTIE project, for their hard work and contribution, and also to the experts from the information technology domain of the National 863 Plan, for their continuous support and guidance.

References

1. DeMarco, T., Lister, T.: Peopleware-Productive Projects and Teams. Dorset House Publishing Co., New York (1987)
2. Herbsleb, J.D.: Global software engineering: the future of socio-technical coordination. In: 2007 Future of Software Engineering, pp. 188–198. IEEE Computer Society (2007)
3. Howe, J.: Crowdsourcing: Why the Power of the Crowd is Driving the Future of Business. Random House, New York (2008)
4. Giles, J.: Internet encyclopaedias go head to head. Nature **438**(7070), 900–901 (2005)
5. Torvalds, L.: The linux edge. Commun. ACM **42**(4), 38–39 (1999)
6. Surowiecki, J.: The Wisdom of crowds: why the many are smarter than the few and how collective wisdom shapes business. Economies, Societies and Nations (2004)
7. Lakhani, K.R., Panetta, J.A.: The principles of distributed innovation. Innovations **2**(3), 97–112 (2007)
8. Capek, P.G., Frank, S.P., Gerdt, S., Shields, D.: A history of IBM's open-source involvement and strategy. IBM SYST. J. **44**(2), 249–257 (2005)
9. Raymond, E.: The cathedral and the bazaar. Knowl. Technol. Policy **12**(3), 23–49 (1999)
10. Xu, J.: System Programming Language. China Science Press, Beijing (1987)
11. Hasselbring, W., Reussner, R.: Toward trustworthy software systems. Computer **39**(4), 91–92 (2006)
12. Robillard, P.N.: The role of knowledge in software development. Commun. ACM **42**(1), 87–92 (1999)
13. Yang, F., Lü, J., Mei, H.: Technical framework for internetware: an architecture centric approach. Sci. China Ser. F: Inf. Sci. **51**(6), 610–622 (2008)
14. Huang, S.K.: Mining version histories to verify the learning process of legitimate peripheral participants. ACM SIGSOFT Softw. Eng. Notes **30**(4), 1–5 (2005)
15. Zanetti, M.S., Scholtes, I., Tessone, C.J., Schweitzer, F.: Categorizing bugs with social networks: a case study on four open source software communities. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 1032–1041. IEEE Press (2013)

16. Dou, W., Wei, G.W., Wei, J.C.: Collaborative software development environment and its construction method. J. Front. Comput. Sci. Technol. **5**(7), 624–632 (2011)
17. TrustieTeam: Trustie software resource management specification, trustie-srmc v2.0 (2011)
18. O'reilly, T.: What is web 2.0: design patterns and business models for the next generation of software. Commun. Strateg. (65) (2007)
19. Wu, W., Tsai, W.T., Li, W.: An evaluation framework for software crowdsourcing. Front. Comput. Sci. **7**(5), 694–709 (2013)
20. Yu, Y., Wang, H., Yin, G., Ling, C.: Reviewer recommender of pull-requests in GitHub. In: 2014 30th IEEE International Conference on International Conference on Software Maintenance and Evolution (ICSME 2014 TOOLS), pages to appear. IEEE (2014)
21. TrustieTeam: Trustie collaborative development environment reference specification, trustie-forge v2.0 (2011)
22. Lin, Y., Huai-Min, W., Gang, Y., Dian-Xi, S., Xiang, L.: Mining and analyzing behavioral characteristic of developers in open source software. Chin. J. Comput. **10**, 1909–1918 (2010)
23. Zhou, M., Mockus, A.: Developer fluency: achieving true mastery in software projects. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 137–146. ACM (2010)
24. TrustieTeam: Trustie software production line integration specification, trustie-spl v3.1 (2011)
25. Zhao, J., Xie, B., Wang, Y., Xu, Y.: TSRR: A software resource repository for trustworthiness resource management and reuse. In: SEKE, pp. 752–756 (2010)
26. TrustieTeam: Trustie software trustworthiness classification specification, trustie-stc v2.0 (2011)
27. Cai, S., Zou, Y., Shao, L., Xie, B., Shao, W.: Framework supporting software assets evaluation on trustworthiness. J. Softw. China **21**(2), 359–372 (2010)
28. Wang, T., Wang, H., Yin, G., Ling, C.X., Li, X., Zou, P.: Mining software profile across multiple repositories for hierarchical categorization. In: 2013 29th IEEE International Conference on Software Maintenance (ICSM), pp. 240–249. IEEE (2013)
29. Yu, Y., Wang, H., Yin, G., Liu, B.: Mining and recommending software features across multiple web repositories. In: Proceedings of the 5th Asia-Pacific Symposium on Internetware, p. 9. ACM (2013)
30. Shao, L., Zhao, J., Xie, T., Zhang, L., Xie, B., Mei, H.: User-perceived service availability: a metric and an estimation approach. In: IEEE International Conference on Web Services, ICWS 2009, pp. 647–654. IEEE (2009)
31. Zeng, J., Sun, H.L., Liu, X.D., Deng, T., Huai, J.P.: Dynamic evolution mechanism for trustworthy software based on service composition. J. Softw. **21**(2), 261–276 (2010)
32. Booch, G., Brown, A.W.: Collaborative development environments. Adv. Comput. **59**, 1–27 (2003)
33. Kazman, R., Chen, H.M.: The metropolis model a new logic for development of crowdsourced systems. Commun. ACM **52**(7), 76–84 (2009)
34. Tapscott, D., Williams, A.D.: Wikinomics: How Mass Collaboration Changes Everything. Penguin, New York (2008)
35. Mockus, A., Fielding, R.T., Herbsleb, J.: A case study of open source software development: the apache server. In: Proceedings of the 22nd International Conference on Software Engineering, pp. 263–272. ACM (2000)
36. Crowston, K., Howison, J.: The social structure of free and open source software development. First Monday **10**(2) (2005)
37. Sarma, A., Maccherone, L., Wagstrom, P., Herbsleb, J.: Tesseract: interactive visual exploration of socio-technical relationships in software development. In: IEEE 31st International Conference on Software Engineering, ICSE 2009, pp. 23–33. IEEE (2009)
38. Dabbish, L., Stuart, C., Tsay, J., Herbsleb, J.: Social coding in GitHub: transparency and collaboration in an open software repository. In: Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work, pp. 1277–1286. ACM (2012)
39. Posnett, D., D'Souza, R., Devanbu, P., Filkov, V.: Dual ecological measures of focus in software development. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 452–461. IEEE (2013)

40. Bird, C., Nagappan, N., Devanbu, P., Gall, H., Murphy, B.: Does distributed development affect software quality?: An empirical case study of windows vista. Commun. ACM **52**(8), 85–93 (2009)
41. Mockus, A.: Amassing and indexing a large sample of version control systems: towards the census of public source code history. In: 6th IEEE International Working Conference on Mining Software Repositories, MSR'09, pp. 11–20. IEEE (2009)
42. Begel, A., Bosch, J., Storey, M.A.: Social networking meets software development: perspectives from github, msdn, stack exchange, and topcoder. Softw. IEEE **30**(1), 52–66 (2013)
43. Liu, K., Shan, Z., Wang, J., He, J., Zhang, Z., Qin, Y.: Overview on major research plan of trustworthy software. Bull. Natl. Nat. Sci. Found. China **22**(3), 145–151 (2008)
44. Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., Merrit, M.J.: Characteristics of Software Quality, vol. 1. North-Holland Publishing Company, Amsterdam (1978)
45. McCall, J.A., Richards, P.K., Walters, G.F.: Factors in Software Quality. General Electric National Technical Information Service, Berlin (1977)
46. Golden, B.: Succeeding with Open Source. Addison-Wesley Professional, Boston (2005)
47. Wasserman, A., Pal, M., Chan, C.: The business readiness rating model: an evaluation framework for open source. In: Proceedings of the EFOSS Workshop, Como, Italy (2006)
48. Russo, B., Damiani, E., Hissam, S., Lundell, B., Succi, G.: Open Source Development, Communities and Quality. Springer, Berlin (2008)
49. Bird, C., Nagappan, N.: Who? Where? What? examining distributed development in two large open source projects. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), pp. 237–246. IEEE (2012)
50. Barr, J.: The paradox of free/open source project management (2005). http://archive09.linux.com/feature/42466. Accessed 6 May 2014
51. Aranda, J., Venolia, G.: The secret life of bugs: going past the errors and omissions in software repositories. In: Proceedings of the 31st International Conference on Software Engineering, pp. 298–308. IEEE Computer Society (2009)
52. Anonymous: Gerrit code review—a quick introduction, version 2.10-rc0-199-g60bca74 (2014). https://gerrit-review.googlesource.com/Documentation/intro-quick.html
53. Anonymous: Crowdsourcing software development, from Wikipedia (2014). http://en.wikipedia.org/wiki/Crowdsourcing_software_development
54. Wu, W., Tsai, W.T., Li, W.: Creative software crowdsourcing: from components and algorithm development to project concept formations. Int. J. Creat. Comput. **1**(1), 57–91 (2013)
55. Huhns, M.N., Li, W., Tsai, W.T.: Cloud-based software crowdsourcing (dagstuhl seminar 13362). Dagstuhl Rep. **3**(9) (2013)
56. Tsai, W.T., Wu, W., Huhns, M.N.: Cloud-based software crowdsourcing. IEEE Internet Comput. **18**(3), 78–83 (2014). http://doi.ieeecomputersociety.org/10.1109/MIC.2014.46

Social Clouds: Crowdsourcing Cloud Infrastructure

Kyle Chard and Simon Caton

Abstract Software crowdsourcing is becoming an increasingly viable model for creating production software addressing every aspect of the software development lifecycle. However, as software development processes become yet more complex requiring dedicated systems for development, testing, and deployment, software crowdsourcing projects must also acquire considerable infrastructure in order to facilitate development. We propose the use of an infrastructure crowdsourcing model, termed a Social Cloud, to facilitate a user-contributed cloud fabric on which software development services and systems can be hosted. Social Clouds are motivated by the needs of individuals or groups for specific resources or capabilities that can be made available by connected peers. Social Clouds leverage lessons learned through volunteer computing and crowdsourcing projects such as the willingness of individuals to make their resources available and offer their expertise altruistically for "good causes" or in exchange for other resources or payment. In this chapter we present the Social Cloud model and describe how it can be used to crowdsource software infrastructure.

1 Introduction

Software crowdsourcing [33, 44] is a new approach to software engineering in which individual tasks of the software development lifecycle such as coding, deployment, documentation and testing are outsourced to a potentially unknown group of individuals from the general public. Building upon the wide-spread adoption of crowdsourcing, which focuses on unskilled or semi-skilled participants collectively achieving a given goal, software crowdsourcing further extends this model by requiring sophisticated coordination of expert users who together–knowingly or

K. Chard (✉)
Computation Institute, University of Chicago and Argonne National Laboratory, Lemont, USA
e-mail: chard@uchicago.edu

S. Caton
Karlsruhe Service Research Institute, Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: simon.caton@kit.edu

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_11

unknowingly–participate in an orchestrated workflow. This model moves away from traditional software development approaches towards a decentralized, peer-based model in which open calls for participation are used to accomplish tasks.

Software crowdsourcing shows great promise as a model for developing production software without requiring dedicated teams of developers. Software crowdsourcing projects cover a wide array of efforts ranging from contribution to AppStores (e.g. Apple AppStore or Google Play) through to individual fine grained contests in tools such as TopCoder.[1] In TopCoder users set challenges (e.g., programming, design or testing) along with a prize, deadline and requirements for completing the challenge. Other users can then discover and accept challenges. Upon completion a winning contribution is selected and the winner is rewarded. In this case, participants are motivated by monetary rewards; however in other crowdsourcing scenarios they may also be motivated by other factors such as altruism or standing in the community.

Like any software development model, software crowdsourcing requires significant computing infrastructure for all stages of the software development lifecycle. For instance, storage resources are required for code repositories, databases, test datasets, and documentation. Compute resources are required for testing, continuous integration, and hosting of services. These requirements are in stark contrast to the infrastructure requirements of other crowdsourcing models which typically require only minimal infrastructure to track tasks and results across many thousands of users. Existing crowdsourcing approaches will not scale to the sophisticated requirements of software crowdsourcing. In a crowdsourced environment questions arise over who should operate and maintain such infrastructure. Due to the collaborative nature of most software development processes it is infeasible to operate all infrastructure on individuals' machines, rather, collaborative infrastructure is required so that all users can access these systems.

Recent work has explored the use of cloud based approaches to support the requirements of software development activities. In fact, systems such as TopCoder offer the ability to provision a cloud virtual machine (VM) for a given challenge. Cloud based systems allow access by all participants and enable use of virtualized resources. This, for example, allows participants to create sophisticated infrastructure with the same ease at which they could do it locally. However, it does not solve the problem of how these cloud resources are procured. And due to the predominant pay-as-you go model of cloud usage it does not solve the problem of how these resources are paid for.

In this chapter we present the use of a Social Cloud [9] as a model for providing such shared infrastructure via an infrastructure crowdsourcing-like model. Originally developed as a resource sharing framework for sharing resources between connected individuals in a social network we suggest here that such approaches can also be applied to form a crowdsourced resource fabric for software development. The Social Cloud model is built upon the premise of virtualized resource contribution from semi-anonymous (crowd) and socially connected users. Virtualization techniques enable contributed resources to be used securely (from both the consumer's and provider's

[1] http://www.topcoder.com/.

perspectives) and to also provide an intuitive and sandboxed environment on which to construct services. We present an overview of a social storage cloud, social content delivery network (S-CDN) and social compute cloud to be used as the basis for supplying shared infrastructure for software crowdsourcing and describe the use of a currency-based, social network-based and matching-based model for allocating resources in such environments.

2 Social Clouds

A Social Cloud is "*a resource and service sharing framework utilizing relationships established between members of a social network.*" [9]. It is a dynamic environment through which cloud-like provisioning scenarios can be established based upon the implicit levels of trust represented between individuals in a social network. Where a social network describes relationships between individuals and social networks exist in, and can be extracted from, any number of multi-user system, including for example, crowdsourcing systems.

In the remainder of this section we present an overview of Social Clouds and describe the crowdsourcing calls that can be employed. We then present implementations of a Social Storage Cloud, Social Content Delivery Network (CDN), and Social Compute Cloud. This will enable us to describe a general Social Cloud model for crowdsourcing Cloud infrastructures (in Sect. 4.1). In each of these settings, we have also investigated different mechanisms for managing exchange using credit-based, social network-based and preference-based models, each of which can be viewed as a proxy for handling different types of crowdsourcing calls.

2.1 Motivation and Overview

The vision of a Social Cloud is motivated by the need of individuals or groups to access resources they are not in possession of, but that could be made available by connected peers. Later, we describe a Social Storage Cloud, a Social Content Delivery Network and a Social Compute Cloud; however additional resource types (such as software, capabilities, software licenses, etc.) could also be shared. In each case a Social Cloud provides a platform for sharing resources within a social network. Using this approach, users can download and install a middleware, leverage their personal social network, and provide resources to, or consume resources from, their connections.

The basis for using existing online social networks is that the explicit act of adding a "friend" or deriving an association between individuals implies that a user has some degree of knowledge of the individual being added. Such connectivity between individuals can be used to infer that a trust relationship exists between them. Similar relationships can be extracted between software crowdsourcing participants

due to the reliance on coordinated and collaborative development practices. In both situations there may be varying degrees of trust between participants, for instance family members have more trust in one another than acquaintances do. Likewise, close collaborators in a software development process may have more trust in one another than in members they do not know. The social network model provides a way to encode this information as basic social relationships and to augment the social graph with additional social or collaborative constructs such as groups (e.g. friend or project lists), previous interactions, or social discourse.

Another way to think about a Social Cloud is to consider that social groups are analogous to dynamic Virtual Organizations (VOs) [15]. Groups, like VOs, have policies that define the intent of the group, the membership of the group and sharing policies for the group. Clearly, in this model Social Clouds are not mutually exclusive, that is, users may be simultaneously members of multiple Social Clouds. Whereas a VO is often associated with a particular application or activity, and is often disbanded once this activity completes, a group is longer lasting and may be used in the context of multiple applications or activities. While Social Clouds may be constructed based on Groups or VOs, we take the latter view, and use the formation of social groups to support multiple activities. In addition, different sharing policies or market metaphors can be defined depending on the group, for instance a user may be more likely to share resources openly with close collaborators without requiring a high degree of reciprocation, however the same might not be true for friends or more distant collaborators.

Resources in a Social Cloud may represent a physical or virtual entity (or capability) of limited availability. A resource could therefore encompass people, skills, information, computing capacity, or software licenses—hence, a resource provides a particular capability that is of use to other members of a group or community. Resources shared in a Social Cloud are by definition heterogeneous and potentially complementary, for example one user may share storage in exchange for access to compute. Or in the case of software crowdsourcing, a user may contribute compute resources to a particular group of users associated with a particular project while using storage resources associated with that same project.

In order to manage exchange, a Social Cloud requires mechanisms by which resource sharing can be controlled and regulated. Like crowdsourcing applications, in which contributors are rewarded for their contributions in different ways, a Social Cloud must also support different models for reward. Crowdsourcing applications often make use of monetary rewards for contribution (e.g., Amazon Mechanical Turk) or leverage reputation-based rewards and altruism (e.g., Wikipedia). Similarly, crowd workers have different motivations for participating (see: [43]), and we argue that many of these are also present in a Social Cloud context. In a Social Cloud we use the notion of a social marketplace as a model for facilitating and regulating exchange between individuals. A marketplace is an economic system that provides a clear mechanism for determining matches between requests and contributions. Importantly, a marketplace need not require monetary exchange, rather non-monetary protocols such as reciprocation, preference matching, and social graph based protocols can be used to determine appropriate allocations. If we compare

possible non-monetary incentives for participating in a Social Cloud (see: [21]) to the motivation of participating in crowdsourcing platforms as defined in [43], there is a resounding overlap.

2.2 Crowdsourcing Calls

As Social Clouds are a form of *infrastructure crowdsourcing* they can be constructed with various calls depending on the purpose, and intent of the call. By leveraging [40]'s definition of call types, we refer to the following types of call for a Social Cloud:

- **Open Hierarchical** a call for resources to construct a personal Social Cloud. Here the call initiator may specify policies that define which resources are accepted for use. Where this could include specific relationship types, interaction histories or competencies etc. The important thing to note in this case, however, is that the call itself, is open; implying that anyone can offer to participate, but their participation may be subjected to user-specific policies.
- **Open Flat** a call for platform resources in the management of a Social Cloud. In this call, a Social Cloud platform asks for computational resources to facilitate its basic functionality. Resources can be contributed by any member of the community. We refer to this type of platform as a co-operative platform see: [22].
- **Closed Hierarchical** a call for resources from a specific social group. Here a call is only visible to a specific (sub)set of a user's friends and/or collaborators. Final selection, as with the open hierarchical call, may still be subject to user-specific policies.
- **Closed Flat** a call for platform resources from a specific (sub)community. Here a user or set of users define the social boundaries of a Social Cloud, for instance friends of friends, or a given social group or circle. Anyone within this community may provide resources for the Social Cloud platform in a similar manner to the Open Flat call.

From these call types, a given Social Cloud may enable contributions from a tight group of participants or more widely across a social network (e.g., friends of friends). Similarly, Social Clouds face the same difficulties as crowdsourcing applications with respect to quality, however, unlike typical crowdsourcing applications simple approaches such as task redundancy are not applicable. For this reason, we rely on interpersonal trust as a model for establishing reputation and predicting quality. Software crowdsourcing approaches, given their requirement for expert user contributions and complex tasks, may also leverage such approaches for establishing contribution quality.

2.3 Social Storage Cloud

In [9, 10] we present a Social Storage Cloud designed to enable users to share elastic storage resources. The general architecture of a Social Storage Cloud is shown in Fig. 1. We implement a Social Storage Cloud as a service-based Facebook application. Where a social network (Facebook) provides user and group management as well as the medium to interact with the Social Cloud infrastructure through an embedded web interface (Facebook application) which in turn exposes the storage service interfaces directly. To participate in a Social Storage Cloud users must deploy and host a simple storage service on their resources. Consumers can then interact directly with a specific storage service when allocated via a social/market protocol. The social marketplace is responsible for facilitating sharing and includes components for service registration and discovery, implementing and abstracting a chosen market protocol, managing and monitoring provisions, and regulating the economy. In this case we implement two economic markets: a posted price and a reverse auction. Both markets operate independently and are designed to work simultaneously.

Storage services are implemented as Web Services Resource Framework [12] (WSRF) services and provide an interface for users to access virtualized storage. Contributors must install this service on their local resources and register the service with the Social Storage Cloud application to participate in the market. This service exposes a set of file manipulation operations to users and maps their actions to operations on the local file system. Users create new storage instances by first requesting an allocation from a Social Storage Cloud and then passing the resulting service level agreement (SLA) (formed as the result of allocation) to a specific storage service, this creates a mapping between a user, agreement, and a storage instance. Instances are identified by a user and agreement allowing individual users to have multiple storage

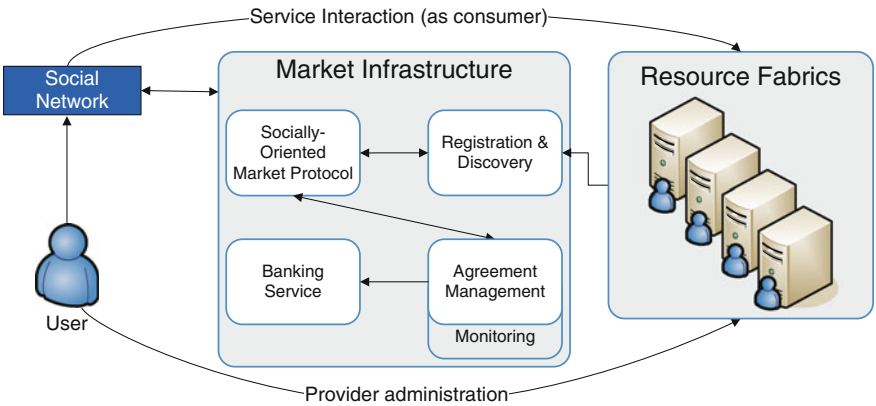


Fig. 1 Social Storage Cloud Architecture. Users register shared services, their friends are then able to provision and use these resources through a Facebook application. Allocation is conducted by an underlying market infrastructure

instances in the same storage service. The storage service creates a representative WSRF resource and an associated working directory for each instance to sandbox storage access. The resource keeps track of service levels as outlined in the agreement such as the data storage limit. Additionally the service has interfaces to list storage contents, retrieve the amount of storage used/available, upload, download, preview and delete files. These interfaces are all made available via the integrated Facebook application.

The two market mechanisms (posted price, and reverse auction) operate similarly, allowing consumers to select and pay for storage resources hosted by their friends. In a posted price market users select storage from a list of friends' service offers. In the reverse auction (tender) market, consumers outline specific storage requirements and pass this description to the Social Cloud infrastructure; providers then bid to host the storage. Both mechanisms result in the establishment of an SLA between users. The SLA is redeemed through the appropriate storage service to create a storage instance. An example summary user interface is shown in Fig. 2. This summary view shows user allocations both on others' resources as well as others' allocations on contributed resources.

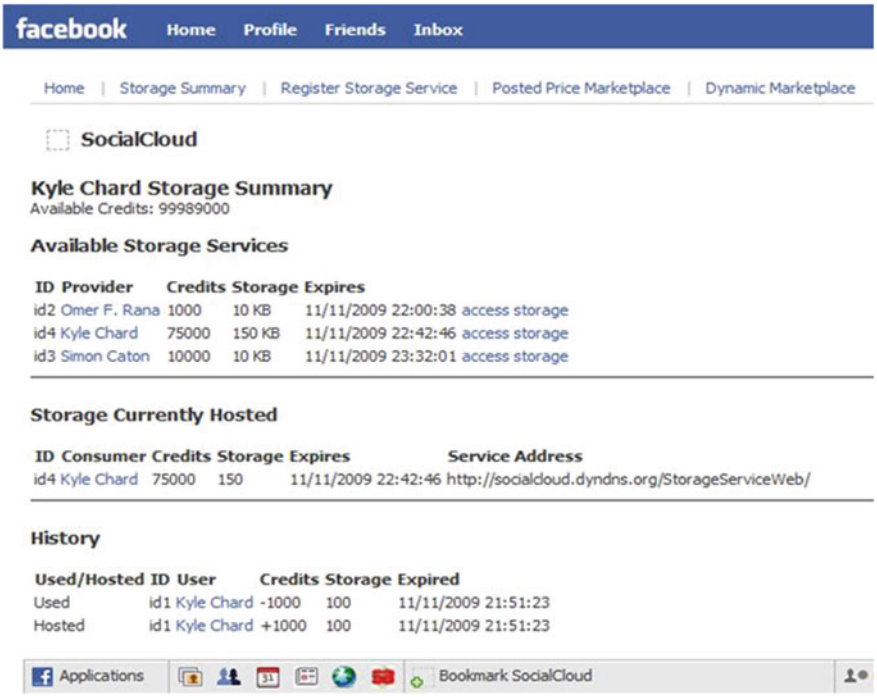


Fig. 2 Social Storage Cloud user summary interface. This interface displays available storage services in a user's network. It also lists other users consuming storage resources as well as historical reservations

The Social Storage Cloud is the simplest model with respect to issuing a call, and it uses the most common approach for regulating exchange–a monetary model. Calls in this case are closed. Only established social relationships (or friend in Facebook terminology) are permitted. The call is also hierarchical in that requesting users set the rules for the exchange through the use of an institutionalized (economic) mechanism, which also performs the allocation process. In [9, 10], we explore the use of a credit-based system that rewards users for contributing resources and charges users for consuming resources. The use of a credit model requires the implementation of services to manage the exchange of credits securely. We use a Banking service to manage users' credit balances and all agreements a user is participating (or has participated) in. Credits are exchanged between users when an agreement is made, prior to the service being used. The two concurrent economic markets, posted price and reverse auctions, are designed to model different forms of exchange: a posted price index, and a reverse Vickrey auction. Following an allocation, communication between consumer and provider is through the establishment of an SLA (represented using WS-Agreement [3]).

In a **posted price** market a user can advertise a particular service in their Social Cloud describing the capabilities of the service and defining the price for using this service. Other users can then select an advertised service and define specific requirements (storage amount, duration, availability, and penalties) of the provision. This approach is analogous to the process followed by crowdsourcing applications such as Amazon Mechanical Turk where requesters post tasks and advertise a stated price for accomplishing the task. The Social Storage Cloud uses a simple index service to store offers and provides an interface for other users to then discover and select these offers. When a user selects a service offer they also specify their required service levels, a SLA is created defining the requirements of the provision such as duration and storage amount. Before using the service, the generated SLA must be passed to the appropriate storage service to create an instance. The storage service determines if it will accept the agreement based on local policy and current resource capacity. Having instantiated storage the agreement is passed to the Banking service to exchange credits. A copy of the agreement is stored as a form of receipt.

In a **reverse auction** (tender) market, a requesting user can specify their storage requirements and then submit an auction request to the Social Storage Cloud. The user's friends can then bid to provide the requested storage. We rely on auction mechanisms provided by the DRIVE meta-scheduler [8]. In particular, we use a reverse Vickrey auction protocol as it has the dominant bidding strategy of truth telling, i.e., a user's best bidding strategy is to truthfully bid in accordance to their (private) preferences, making the Vickrey auction more socially centric. It also means that "anti-social" behavior such as counter speculation is fruitless. In a reverse auction providers compete (bid) for the right to host a specific task. The DRIVE auctioneer uses the list of friends to locate a group of suitable storage services based on user specified requirements; these are termed the bidders in the auction. Each bidder then computes a bid based on the requirements expressed by the consumer. The storage services include a DRIVE-enabled interface that is able to compute simple bids based on the amount of storage requested. The auctioneer determines the auction winner

and creates an SLA between the consumer and the winning bidder. As in the posted price mechanism, the agreement is sent to the specified service for instantiation and the bank for credit transfer. In this model the consumer is charged the price of the second lowest bid, as the Vickrey auction is a second-price mechanism.

When considering this market approach for crowdsourcing there are clear advantages and disadvantages: The use of a virtual currency provides a tangible framework for users to visualize their contribution and consumption rates in a way they can relate to: if they run out of credits, they cannot consume resources; users can manage their credits independently with respect to their personal supply and demand; any mechanism or means of executing a call can be designed to facilitate exchange and achieve specific design intentions; and, the design of the mechanisms have obvious parallels to classic crowdsourcing call structures. Despite these advantages, however, there is an obvious challenge: the need to manage and maintain economic stability, i.e., inflation/deflation over time and the dynamics of context: users, like workers can come and go which can aggravate inflation/deflation. These challenges cannot be understated.

2.4 Social Content Delivery Network

In [11, 32] we present the notion of a Social Content Delivery Network (CDN) which builds upon the idea of a Social Storage Cloud to deliver scalable and efficient distribution of shared data over user contributed resources. The Social CDN is designed to enable data-based scientific exchanges via relationships expressed in social or community networks. Figure 3 illustrates the Social CDN model. Here a user has produced a data artifact for a collaborative project (e.g. the results of a scientific experiment, a new software component, bundle or library, or new data set for analysis), which the user wishes to share with their collaborators. In the Social Storage Cloud setting presented previously this would constitute a backup action being performed by the user using the storage resources of a their social peers, similar to that of a Dropbox storage action. In a Social CDN, however, the backup action is secondary to the action of sharing and distributing data amongst collaborators.

The Social CDN model builds upon a network of *Data Followers*, analogous to to Twitter followers, users that follow data status updates and data posts of a given user. Note that users do not automatically reciprocally follow their data followers. Therefore, like circles in Google+, a follower-followee connection is not considered bilateral. However, the relationship between these users must be bilaterally authorized for our assumptions on pre-existent trust to hold. In order to share data, a user appends an artifact (via a social network application) to a status message or Tweet. This action tells the Social CDN which dataset should be shared, in which data follower circle, and invokes the Social CDN's data transport and management algorithms to execute the sharing action. Likewise, when users access data status messages they may also be published to enable social interactions around data usage.

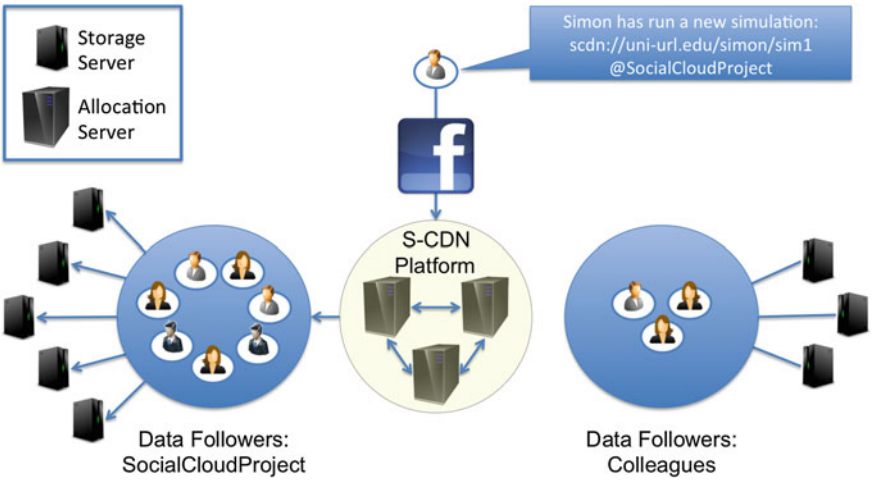


Fig. 3 Social CDN: usage and overview. Data followers of a user receive data updates when the user shares a dataset with the Social CDN. Allocation servers determine the most efficient placement and transfer of the dataset

As in the the Social Storage Cloud setting, users contribute storage resources to the Social CDN, i.e., Social Cloud platform. This enables the Social CDN to use storage resources located on the edge devices owned (and therefore contributed) by members of data follower circles, including the user sharing the data. These servers are used to share and access data, and also as temporary hosts for others' data within the network. Users who provide resources to the Social CDN can also have the data encapsulated by a data post pushed to their resources in the form of replicas by the Social CDN architecture.

By building on the notion of a Social Storage Cloud, storage resources contributed to the Social CDN are used by the central Social CDN application to distribute data across replica nodes. In this sense, the system constructs a distributed data warehouse across resources networked in accordance to the social network of their owners. However, rather than using proprietary storage services the Social CDN builds upon Globus [14], a provider of high performance data transfer and sharing. Globus uses'endpoints' installed on users' resources to enable transfer and sharing. The Social CDN application uses the Globus APIs to alter sharing properties on an endpoint to enable replicas to be stored and accessed. It then uses these same APIs to transfer data between a replica and the requester's endpoint. The Social CDN web application is implemented as a Django application, it uses a social network adapter to connect to Facebook for authentication and access to the user's social graph, and a local database to store and manage users and allocations.

In this setting, the main crowdsourcing-like call is in the construction of the Social CDN infrastructure—an open flat call in that anyone can take part and provide resources to the Social CDN. How these resources are then used, is another matter,

as we can assume that data is not shared universally. Therefore, there are a myriad of ways in which to move data around the Social CDN. Unlike the Social Storage Cloud, the incentives to take part in and contribute to a Social CDN are more closely tied with personal and collective benefit. For this reason, we have not implemented a market-like setting as we expect the Social CDN to be autonomically [28] self-managed over time based on how interactions and data shares emerge over time.

Instead, we leverage the social basis of a Social Cloud to use network structure and network analysis algorithms to select appropriate replica locations on which to distribute data. This approach attempts to place replicas of each dataset on selected available endpoints of the dataset's owner's friends (followers), where the replication factor (the number of nodes selected for replication) is a configurable system parameter. The approach attempts to predict usage based on relationships between users, for example by placing replicas on friends with the strongest connection to the data contributor.

In the Social CDN we use social network analysis algorithms to rank potential replica locations, for example determining important, well connected individuals in the network. We use graph theory metrics such as centrality, clustering coefficient, and node betweenness to determine nodes that are important within a network. In each case the network analysis algorithms identify a ranked list of nodes (social network members) that are used to place replicas. To address availability constraints we construct a graph that has edges between nodes if the availability of two nodes overlaps, and a "distance" weighting is assigned to each edge that describes the transfer characteristics of the connection. When selecting replicas, we choose a subset of nodes that cover the entire graph with the lowest-cost edges. In [11] we show that using such approaches can improve the accessibility of datasets by preemptively allocating replicas to nodes that are located "near" to potential requesters.

In a crowdsourcing context the use of social network based allocation strategies is appropriate as it presupposes quality driven through matching individuals that know one another, or have worked together in the past. For example, such approaches could be used to select users (and their resources) that are central in a network and therefore have strong bonds with other members of the network (or project). The use of a social network-based allocation approach upon the premises and observation of previous as well as existing collaborative actions suits the establishment of Social Clouds in scenarios like software crowdsourcing. Such contexts are inherently network based, and even if workers do not know each other in the physical world, they may have digital (working) relationships in (semi)anonymous networks that can be seen as a forms of pre-existent trust. Thus fulfilling the basic premises of a Social Cloud.

In the Social CDN, we do not adopt an SLA-based approach. The reason is quite simple: replicas and their resources are assumed to be transient and, naturally, also replicated. Similarly, there is no way of determining for how long a replica should be available, or how long the follower-followee relationship will be required or maintained. Furthermore, the Social CDN is responsible for the placement of replicas and their migration (as appropriate) around the network, therefore it would also be cumbersome to create and sign SLAs in this setting. This means a Social

CDN using the allocation method presented is constructed on a best-effort basis, and inherently reliant on the collaborative impetus of its users.

2.5 Social Compute Cloud

In [5] we present a Social Compute Cloud designed to enable access to elastic compute capabilities provided through a cloud fabric constructed over resources contributed by users. This model allows users to contribute virtualized compute resources via a lightweight agent running on their resources. Consumers can then lease these virtualized resources and write applications using a restricted programming language.

The general architecture of the Social Compute Cloud is shown in Fig. 4. The Social Compute Cloud is built upon Seattle [4], an open source Peer-to-Peer (P2P) computing platform. The Social Compute Cloud extends Seattle to use Facebook APIs to authenticate users and to associate and access a user's social graph. It uses Seattle's virtualized resource fabric to enable consumers to offer their resources to the cloud by hosting sandboxed lightweight virtual machines on which consumers can execute applications, potentially in parallel, on their computing resources. Consumers can access these virtualized resources (via the secure virtual machine interfaces) to execute arbitrary programs written in a Python-based language. While the concept of a Social Compute Cloud could be applied to any type of virtualization environment we use lightweight programming (application level) virtualization as this considerably reduces overhead and the burden on providers; in [42] we explore the use of a more heavyweight virtualization environment based on Xen, however the time to create and contextualize VMs was shown to be considerable.

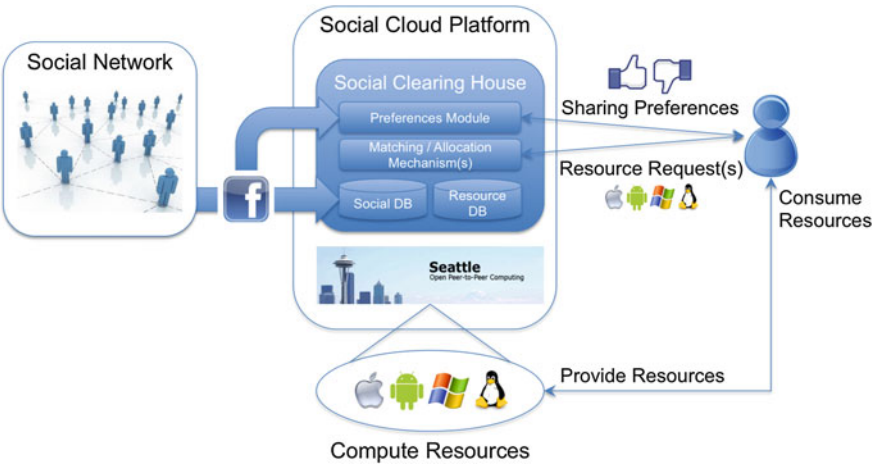


Fig. 4 Social Compute Cloud Architecture

We chose to base the Social Compute Cloud on Seattle due to its lightweight virtualization middleware and its extensible clearing house model which we extend to enable social allocation. Building upon Seattle we leverage the same base implementation for account creation and registration, resource contribution infrastructure, and resource acquisition mechanisms. We have extended and deployed a new social clearing house that leverages social information derived from users' Facebook profiles and relationships along with a range of different preference matching allocation protocols. We have implemented a service that enables users to define sharing preferences (e.g., a ranked order of other users) and new interfaces in Seattle that allow users to view and manage these preferences.

Figure 5 shows a user interface which presents the resources being used by a particular user as well as the users that are using this particular user's resources. This interface, extended from Seattle, provides a model to renew and cancel existing reservations.

The Social Compute Cloud poses one main type of call: a two-way closed hierarchical call for resources. Here, users specify preferences with whom they are willing to consume resources from and provide resources to. These preferences, as will be discussed in more detail later, are not binary, i.e., either provide or not provide, but also rank users against one another. In a similar manner to the previous two Social Cloud settings, an open/closed flat call for platform resources, could also be envisaged. Unlike the Social Storage Cloud, we move away from a credit model, but

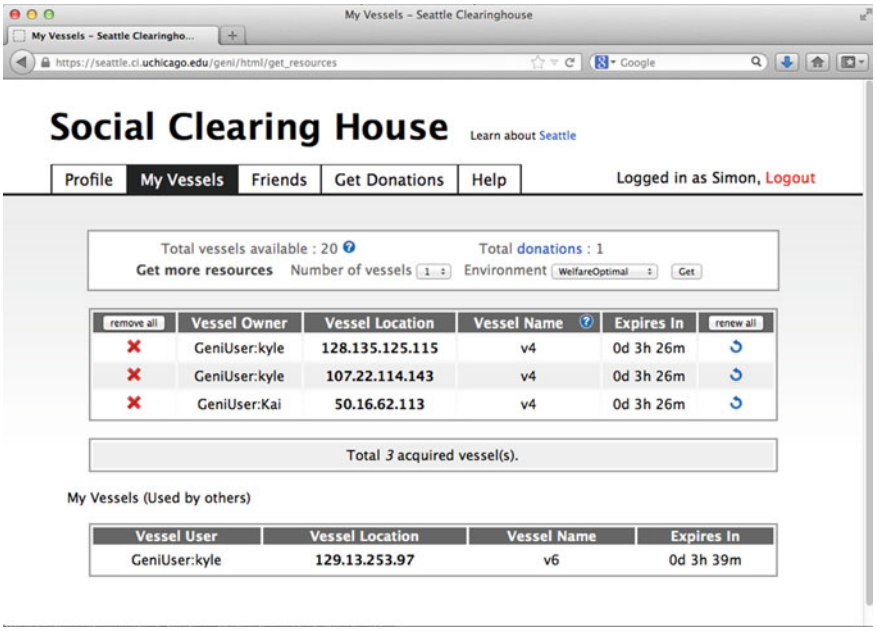


Fig. 5 Social Compute Cloud user summary interface showing current allocations for an individual user

retain user involvement in call management. Similarly, to the Social CDN, we place a heavier reliance on social network constructs, but instead of using social network analysis methods to determine "good" allocations, we involve user choice in the form of sharing preferences, and allocate (or match) based upon these preferences.

Preference matching algorithms allow consumers and producers to specify their preferences for interacting with one another. This type of matching is successfully applied in a variety of cases, including the admission of students to colleges, and prospective pupils to schools. In the Social Compute Cloud we capture the supply and demand of individual users through a social clearing house. As we use a centralized implementation we can derive the complete supply and demand in the market and therefore match preferences between all participants in a given Social Cloud. This of course is only after users define their preferences towards other users, that is, their willingness to share resources with, and consume resources from, other users in the Social Cloud.

To determine the "best" matches between users, given the stated requirements, we use several different matching algorithms. The chosen algorithms differ with respect to their ability to satisfy different market objectives and performance. Commonly used market objectives include finding solutions to the matching problem which are stable (i.e., no matched user has an incentive to deviate from the solution) or optimizing the total welfare of the users, the fairness between the two sides of the market, or the computation time to find a solution. The choice of particular market objectives in turn affects which allocation and matching strategies can be considered. This can range from direct negotiation to a centralized instance that computes this matching; and both monetary and non-monetary mechanisms can be applied. Our approach considers non-monetary allocation mechanisms based on user preferences. Depending on the specific market objective, several algorithms exist that compute a solution to the matching problem, e.g. computing a particularly fair solution or one with a high user welfare.

Two-sided preference-based matching is much studied in the economic literature, and as such algorithms in this domain can be applied in many other settings. For this reason we developed a standalone matching service that enables clients to request matches using existing protocols. This service is used by the Social Compute Cloud (via the social clearing house) to compute matches for given scenarios. The matching service includes three algorithms from the literature, and a fourth of our own implementation. These algorithms have been selected to address limitations when preferences are incomplete or specified with indifference, that is when not all pairwise combinations have preferences associated or when users associate the same preference to many other users. Briefly, the matching algorithms offered in the Social Compute Cloud are:

- **Deferred-Acceptance (DA)** [16]: is the best known algorithm for two-sided matching and has the advantages of having a short runtime and at the same time always yielding a stable solution. However, it cannot provide guarantees about welfare, and yields a particularly unfair solution where one side gets the best stable solution and the other side gets the worst stable solution.

- **Welfare-Optimal (WO)** [26]: is a common matching algorithm that yields stable solutions with the best welfare for certain preference structures. The approach uses structures of the set of stable solutions and applies graph-based algorithms to select the best matches.
- **Shift** [24]: is designed to find stable solutions with consideration for welfare and fairness when indifference or incomplete lists are present. While DA and WO can still be used in such settings, they can no longer guarantee to find the globally best solution. In such settings, Shift can find a stable match, with the maximum number of matched pairs for certain special cases. However, these scenarios are in general hard to approximate, and consequently the standard algorithms are not able to provide non-trivial quality bounds with respect to their objectives. Finding the optimal solution for the matching problem with respect to the most common metrics: welfare or fairness, is NP-hard [24].
- **Genetic Algorithm with Threshold Accepting (GATA)** [23]: is a heuristic-based algorithm that yields superior solutions compared to the other algorithms. The GA starts with randomly created (but stable) solutions and uses the standard mutation and crossover operators to increase the quality of the solutions. GATA then uses this solution as input for the TA algorithm, an effective local search heuristic that applies and accepts small changes within a certain threshold of the current solution performance.

Each of these algorithms have their specific performance merits and we have studied their performance with respect to a Social Cloud in [5, 23]. The key findings, however, suggest that the GATA or similar GATA-like approaches perform well in larger problem sizes, with more complex preference structures and with stochastic supply and demand, rather than a batch allocation mode in which two-sided matching algorithms are often applied. This means for the purposes of facilitating crowdsourcing calls, that users can be more involved in terms of how the call clears.

Like the Social CDN approach, there is currently no handling of SLAs in the Social Compute Cloud. Rather, each compute node is reserved for a predetermined period of time, if during this time a node goes offline the resource consumer will be notified and as a form of "enforcement" may consequently update their sharing preferences. Given the social context of a Social Cloud, it is foreseeable that any issue of this sort be first discussed in order to find either a resolution or cause for the error. This social process is important so as to not damage the real world relationship underpinning the exchange. It is also similar to the feedback and discourse methods often used in crowdsourcing platforms (like Amazon's Mechanical Turk) when employers are not satisfied with the quality of a worker's results.

3 Quality Management, Trust, and Agreements

Having established the means to architect a Social Cloud, and allocate resources in various settings, the question of (collaboration) quality arises with respect to the actual infrastructure provided. This, for the moment, is even irrespective of how this

infrastructure is used. Instead, we refer here to quality of infrastructure instances that are sourced from the community.

The crowdsourcing literature proposes many different means of assessing and defining (worker) quality, reliability and trustworthiness with respect to (task) solutions. Where examples include: redundant scheduling, gold standard questions, qualification tests, peer review and employer acceptance rate. Whilst not all of these approaches are relevant or meaningful in crowdsourcing computational infrastructures, it is still important to maintain a notion of quality that supports collaborative processes. In terms of a computational infrastructure, there are several aspects that can be used to denote and measure quality both quantitatively (e.g. availability, error rate, mean error recovery time, etc.) and qualitatively (non-functional parameters like owner's technical competence, trustworthiness, responsiveness to crowd sourcing calls, etc.).

Without delving into quality properties, the crowdsourcing literature does, in general, differentiate between different methods of assessing quality. Where up-front task design and post-hoc result analysis are the two main methods of controlling work quality [30]. Up-front task design typically involves methods of pre-selection: a means of ensuring a minimum ex-ante quality level of contributions [17] so that an employer mitigates the risk of poor quality solutions through pre-selection tests or processes. Typically, these are in the form of qualification tests, or thresholds for worker attributes. Post-hoc result analysis, however, allows a worker to perform a task before validating the result in some way. Here typical examples are the gold standard (micro)task[2] [37], the redundant scheduling of tasks to multiple workers to provide a basis for solution comparison and worker reliability [29], TopCoder-like competitions, and peer review (for example as in Wikipedia). In fact, we see similar approaches to these used in volunteer computing settings where compute nodes are inherently unreliable e.g. [2, 7].

Despite the level of research into quantitative methods of deriving and defining the quality and reliability of results in the crowdsourcing literature, it is hard to avoid the more subjective issues surrounding the perception of quality with respect to the qualifications and/or competence of a worker. We observed in [13] that although crowdsourcing platforms use several mechanisms to assess worker reliability and capabilities these methods can seldom be applied to identify actual worker abilities or competencies. Instead, they reveal only whether the worker is likely to posses the necessary abilities to perform a specific (micro) task and/or if they will do so diligently. If we consider the notion of a Social Cloud, where computational resources are crowdsourced within a specific (social) community, we can see that the "standard" means of assessing quality may not be sufficient. In [13], we attempted to disentangle quality and competence; where the latter is potentially influential on ex-ante expectations of quality, and thus in decisions related to resource allocations as well as requests.

An alternative method to assessing quality was proposed by [25]: to infer a level of trust in the worker via the accuracy of their solutions. For the purposes of a Social

[2]Tasks where the solution is known ex-ante to test worker accuracy.

Cloud, we can augment this approach by redefining accuracy as either perceived quality ex-ante (in a qualitative sense) or observed quality ex-post using predefined measures for quality of service (in a quantitative sense). This would capture the two methods of assessing quality mentioned above. We can also augment [25]'s inference of trust through the assumption of pre-existent inter-personal trust between members of a Social Cloud. In assessing quality in this manner, we are highlighting two artifacts of a Social Cloud: trust and some form of provisioning agreement.

To avoid a lengthy discussion on trust, we refer to [6] where we defined trust for the context of a Social Cloud as follows: "Trust is a positive expectation or assumption on future outcomes that results from proven contextualized personal interaction-histories corresponding to conventional relationship types and can be leveraged by formal and informal rules and conventions within a Social Cloud to facilitate as well as influence the scope of collaborative exchange". Two of the most relevant aspects in this definition from a crowdsourcing perspective are "interaction-histories" and "conventional relationship types". Where the former overlaps with ex-post measures of prior performance (potentially) in various collaborative contexts, the latter is somewhat abrasive in the context of crowdsourcing. We tend to view workers as a part of a large anonymous human crowd of workers. In a Social Cloud this is not the case (nor may it be the case in software crowdsourcing projects), and consequently, we can view trust at a more subjective and personal level, and (at least) assume that the existence of trust will be positively correlated to quality in the general sense.

However, the ability of users to rely on trust alone is dependent on the type of relationship [41]. This differentiated view of trust means that in some circumstances some form of collaborative agreement that clearly defines measures of quality is needed. In [9, 10] we explored the implementation details of formal agreements or SLAs (Service Level Agreements). However, this in retrospect was an over-engineered approach. Reference [41] observed that in the relationship contexts (close) friend and family, agreements are not perceived necessary by users. However, in relationship contexts such as acquaintances and colleagues (the arguably more likely relationship types in crowdsourcing contexts) some formalization of an agreement as well as some form of explicit incentive to contribute is necessary. This does not, however, imply the formal representation of a collaborative action using a standard like WS-Agreement [3], but rather a leaner representation capturing: the minimum details of the exchange (actors, and definition of instance); basic measures of quality (e.g. availability); implications of failure; and (when applicable) a definition of reward be it tangible, e.g., monetary payment, or intangible, e.g. reputation points.

4 A Social Cloud for Software Crowdsourcing

In this section we present a general Social Cloud architecture and describe how the principles of Social Cloud Computing can be leveraged in software crowdsourcing applications. As software development processes are inherently collaborative, they

involve groups of contributors who are in some cases unknown and in others based on strong social ties between one another; they require sophisticated resources and software to develop, test, deploy, integrate, and perform other common software lifecycle processes; and they rely on contributions from various people to achieve these goals. Thus, there are two areas in which Social Cloud principles can be applied to software crowdsourcing: (1) as a model for supporting software crowdsourcing infrastructure requirements, and (2) as a means of using social network analysis to derive competency and quality.

4.1 General Social Cloud Architecture

Based on our previous experience, we now present a unified architecture that supports Social Storage, Social CDN and Social Compute Clouds. Like any Cloud model, and as discussed above, a platform is required to coordinate and facilitate the basic functionality of a Social Cloud (user management, resource allocation, etc.). The resources to support this platform, can either be provided by a third party, or crowdsourced from the Social Cloud user base, as a form of co-operative platform [22]. Figure 6 shows the high level architecture for a Social Cloud and its key components, which are as follows:

A **Social Marketplace** is an institutionalized microeconomic system that defines how supply is allocated to demand. In other words, it is responsible for the facilitation,

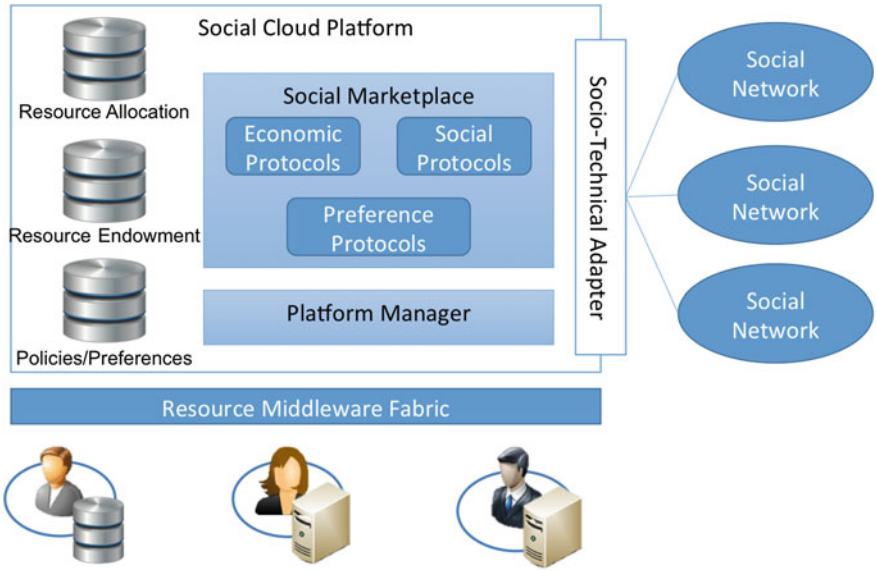


Fig. 6 General Social Cloud Architecture

and clearing of calls. A social marketplace captures the following: the protocols used for distributed resource allocation, the rules of exchange, i.e. who can take part and with whom may they exchange, and the formalization of one or more allocation mechanisms. A social marketplace is therefore the central point in the system where all information concerning users, their sharing policies, and their resource supply and demand is kept. For this reason, the social marketplace requires data stores: to capture the participants, the social graph of its users, as well as their sharing policies; and a resource manager to keep track of resource reservations, availability, and allocations.

A **Platform Manager** administrates the basic functionality of the Social Cloud. The platform manager is a (semi-)autonomic co-operative system managing its resources either through the creation of calls for platform resources, or syphoning off parts of contributed computational resources. It is responsible for ensuring that the Social Cloud is responsive and available. Such approaches may also be applied more widely as a means of supporting other crowdsourcing platforms.

A **socio-technical adapter** provides access to the necessary aspects of users' social networks, and acts as a means of authentication. The socio-technical adapter could leverage any source for social information such as an existing social network platform or a software crowdsourcing application. Once a user's social network has been acquired via the socio-technical adapter, the social marketplace requires the preferences and policies of the user to facilitate resource allocation.

Data stores record state for the Social Cloud such as users, social graphs, resource endowments, policies and preferences, and current and historical allocations. These data stores are used by the social marketplace to influence allocation and by participants to manage their interactions with the social cloud.

A **resource middleware fabric** provides the basic resource fabrics, resource virtualization and sandboxing mechanisms for provisioning and consuming resources. In the examples above the middleware includes mechanisms to access the storage and compute resources of participants. It defines the protocols needed for users and resources to join and leave the system. The middleware is also responsible for ensuring secure sharing of resources by implementing interfaces and sandboxed environments that protect both providers and consumers.

Resources are the technical endowment of users that are provided to, and consumed from, the Social Cloud. These resources could include personal computers, servers or clusters and specifically the storage and compute capabilities that these resources make available.

4.2 Crowdsourcing Infrastructure for Software Crowdsourcing

Figure 7 shows how the unified Social Cloud architecture can be used to crowdsource infrastructure required to facilitate a software development project. In this case, various users–some with existing relationships between one another and some

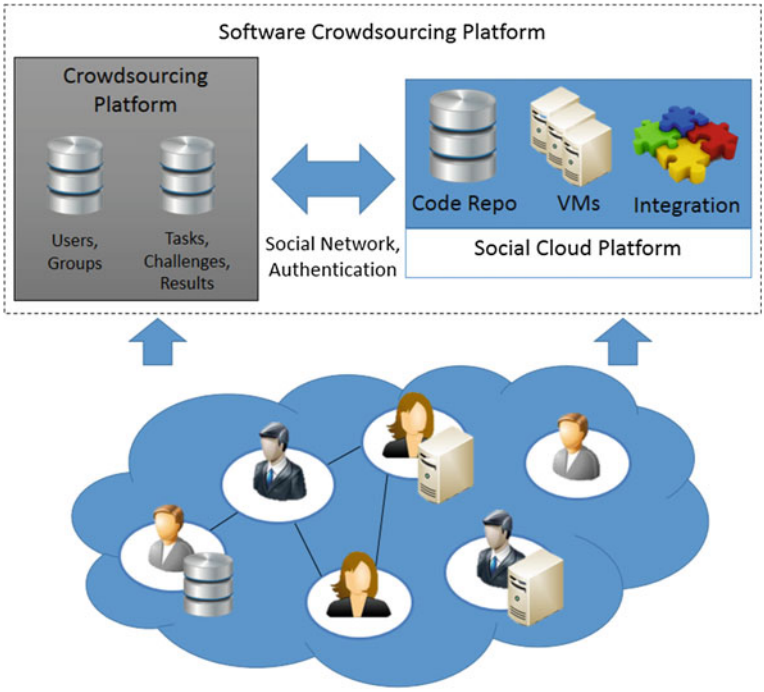


Fig. 7 A software crowdsourcing model that leverages a Social Cloud contributed by participants to facilitate operation of important software development services

without–are participating in a software crowdsourcing project. Not only are these users contributing their software engineering capabilities to the software crowdsourcing project but some are also contributing resources via an associated Social Cloud that allows the hosting and execution of various important software development services for the project.

The Social Cloud leverages the social network derived from the software crowdsourcing platform or potentially from other social networks that exist between users. We expect that in most cases the crowdsourcing platform will provide the authentication and authorization model as well as providing user and group information to the Social Cloud. The Social Cloud uses this information (e.g., groups) to enable user contributed resources and capabilities to be shared with other participants. By establishing a network (based on groups) between users we can also enable other techniques to improve collaboration including, for example, the ability to provide communication between members and to facilitate rapid feedback. It may also provide mechanisms to more easily discover capable participants and motivate contribution [27].

When selecting a software project for participation, participants will be able to optionally contribute resources to the project. Users will be presented a Social Cloud

interface that enables the selection of local resources, and by using existing storage and compute services they will be able to make their resources accessible to the Social Cloud. At this stage agreement on the market model must be made (e.g., how are users rewarded for their contribution) at the same time as reaching agreement regarding their participation in the project (e.g., rewards associated with their software engineering contributions). Policies will need to be developed regarding who can access a contributed resource and for what reason. For example, perhaps resources will be contributed for short term leases to support testing; or alternatively they might be contributed for longer periods to support project wide services such as code repositories.

Other project members can then access and use resources contributed to the Social Cloud through published interfaces and adhering to the market protocols selected. Depending on the contribution and market model, they may be able to consume resources at will by virtue of being a member of the project or they may require some form of matching between their requirements and others capabilities. Resource allocation will follow the general Social Cloud model in which virtualized and sandboxed resources are offered and used by other participants.

There are several different market approaches that can be applied to a software crowdsourcing Social Cloud: (1) The market may be tied to the model employed by the crowdsourcing application, or (2) the market may be independent from the crowdsourcing application and operate only within the context of the Social Cloud.

In the first model, the Social Cloud will use the credits (virtual or real) that are associated with the software crowdsourcing platform. In this case, contributors will associate a credit value with their resource contributions using an economic model. For instance, a contributor may charge $1 credit per day for using their compute resources. Similarly, consumers (or the project) will be charged credits when accessing or using these resources. If resources are used for the project itself, these credits may be taken from the overall reward associated with the project. We expect that such capabilities are present in most crowdsourcing applications and that we can leverage these workflows via published APIs or other means.

In the second model, the Social Cloud will apply an independent economic model such as a credit model, social network model, or preference matching model. The difference between an independent credit model and the credit model tied to the crowdsourcing application is that all credits and exchanges will be managed entirely by the Social Cloud. The social network model will allow project participants to share resources with a group, enabling, for example specification of restrictions based on relationships between individuals. Such restrictions may include allowing access to only close friends or collaborators who have worked together on a previous project, or who share a connection in an external social network. The preference model provides a more regulated mechanism for members to control their contributions and use of resources. In this model users will be able to contribute resources and optionally define preferences for which members can use these resources. Consumers, when requesting resources, may define preferences for which members they use resources from.

4.3 Establishing Trust and Competency via Social Networks

Online social networks such as Facebook, Google+ and Twitter provide a model in which relationships between individuals are encoded digitally and can be accessed programmatically to develop socially-aware applications. Even implicit social networks formed via user accounts, groups, publications and online actions can be extracted and used to establish linkages between individuals. The premise of a Social Cloud builds upon these relationships to infer a level of trust between individuals. This trust can be leveraged to encourage higher levels of quality of service in sharing settings.

As discussed in Sect. 3 such networks provide a powerful model for inferring capabilities and competencies. In traditional crowdsourcing models such requirements are less prevalent as tasks are often oriented around unskilled activities. However, in software crowdsourcing projects it is important to establish levels of proficiency. This is increasingly valuable as it is often difficult to determine the accuracy of self-reported competency, and equally challenging to construct suitable measures to capture, in general, transferable notions of competence for crowdsourcing [13]. While reputation measures provide a method to address these requirements, they require bootstrapping and also reliability that the reputation model cannot be subverted. We believe that the ability to leverage trusted connections between individuals following a Social Cloud model is of particular value in the general crowdsourcing domain for these reasons.

To do this, we first need to define the frames of reference when discussing trust for a Social Cloud or in settings where social structures are used in crowdsourcing. We identified in [6] three frames of reference for trust: (1) trust as an intrinsic (subjective) attribute of an inter-personal social relation, i.e. trust as a basic foundation of social actions; (2) trust in the competence of an individual to be able to deliver a given resource or capability, i.e. a belief in the self-awareness and personal evaluation of competence; and (3) trust in an individual to deliver, i.e. keep their promises, adhere to any (informal) agreements etc. While it is easy to conceive the first and third frame of reference being captured by the social network structure and any associated SLA, soft agreement or "gentleman's agreement" respectively, it is however, difficult to interpret the second frame of reference without a basic definition of competence.

In [13] we provide an overarching summary of competence in crowdsourcing scenarios. Where competence refers to an inflected action with respect to "practical knowledge" and is characterized by "being able to", "wanting to", "being allowed to", and "being obliged to" do something [39]. In crowdsourcing literature the term of competence is often misused to refer to a capability or some form of (domain) authority [13].

We argue that the premises of a Social Cloud could provide a basic framework for (in)formally observing competence as a consequence of network effects: the social ascription of competence by peers. For the ascription of competence a "social" element comes into play as a reciprocal situational ascription of appropriateness to specific actions by the actor themselves and another person or persons (e.g. a

collaborator) who are participating in the relevant situation [13]. Consider, for example, a system similar to LinkedIn endorsements. While such a system would be particularly apt in a software crowdsourcing model issues surround the interpretation of reputation. Social networks, however, provide a mechanism to apply transitive inference, that is, to traverse the graph of endorsements based on previous interactions with a particular entity or project. Such approaches can be used to further enhance the reliability of competence measures.

Coming back to the original frames of reference the approaches put forth by Social Clouds are equally applicable to a software crowdsourcing environment. The implicit and explicit use of social networks provide models to: (1) establish trust between individuals and projects; (2) provide the ability to infer competence based on previous interactions and endorsements; and (3) utilize inherent social incentives and disincentives associated with participation and delivery.

5 Related Work

Until now there has been little research into the implementation of software crowdsourcing applications. Most examples such as TopCoder, uTest and user-contributed application stores are commercially developed and focus on specific aspects of the software development lifecycle. As yet, these approaches offer only limited infrastructure capabilities for their projects, for example TopCoder offers cloud resources on specific projects.

There is however, much literature relating to the exchange or sharing of resources using social fabrics. For example, Intel's "progress thru processors"[3] Facebook application enables contribute of excess compute power to individually selected scientific projects. Users are not rewarded for their contribution as such, however they can view and publish statistics of their contributions. Upon joining the application users may post information to their news feed, or inform friends of the application. The progress thru processors application relies on a generic resource layer constructed by deploying a BOINC [2] application on the users machine.

McMahon and Milenkovic [34] proposed Social Volunteer Computing, an extension of traditional Volunteer Computing, where consumers of resources have underlying social relationships with providers. This approach is similar to the nature of a Social Cloud, but it does not consider the actual sharing of resources, as there is no notion of bilateral exchange.

Pezzi [38] proposes a Social Cloud as a means of cultivating collective intelligence and facilitating the development of self-organizing, resilient communities. In this vision the social network and its services are provided by network nodes owned by members of the network rather than by centralized servers owned by the social network. Pezzi's work is in its infancy and has no architectural details or implementation.

[3] http://www.facebook.com/progressthruprocessors.

Ali et al. [1] present the application of our Social Cloud model to enable users in developing countries to share access to virtual machines through platforms like Amazon EC2. In effect they subdivide existing allocations to amortize instance cost over a wider group of users. Using a cloud bartering model (similar to our previous virtual credit model), the system enables resource sharing using social networks without the exchange of money and relying on a notion of trust to avoid free riding. Like our approach, they use a virtual container (LXC) to provide virtualization within the existing virtual machine instance.

Mohaisen et al. [35] present an extension to our definition of a Social Cloud. The authors investigate how a Social Compute Cloud could be designed, and propose extensions to several well known scheduling mechanisms for task assignments. Their approach considers resource endowment and physical network structure as core factors in the allocation problem.

Gracia-Tinedo et al. [18–20] propose a Friend-to-Friend Cloud storage solution, i.e. dropbox via a social network: F2Box. They analyze and discuss how to retain a reliable service whilst using the best effort provisioning of storage resources from friends. They identify that a pure friend-to-friend system cannot compare in terms of quality of service with traditional storage services. Therefore, they propose a hybrid approach where reliability and availability can be improved using services like Amazon's S3. This approach provides a valuable consideration in the realization of a Social Cloud, but is not necessarily transferable to our setting.

Wu et al. [45, 46] describe a lightweight framework to enable developers to create domain specific collaborative science gateways. They use social network APIs (OpenID, OAuth and OpenSocial) and virtualized cloud resources to facilitate collaboration as well as fine grained and dynamic user controlled resource sharing using social network-based group authorization. These same authorization models are then used to facilitate execution of computational bioinformatics jobs on customized virtual machines shared between users.

Kuada and Olesen [31] propose opportunistic cloud computing services (OCCS): a social network approach for the provisioning and management of enterprise cloud resources. Their idea is to provide a governing platform for enterprise level social networking platforms consisting of interoperable Cloud management tools for the platform's resources, which are provided by the enterprises themselves. The authors, present the challenges and opportunities of an OCCS platform, but there is no indication that they have yet built an OCCS. Similarly, Diaspora,[4] and My3 [36] apply similar concepts to host online social networks on resource provided by their users.

There have also been several publications on economic models for a Social Cloud. Zhang et al. [47] and we [21] discuss different types of incentives users face during their participation in a Social Cloud, and describe the challenges of providing the right incentives to motivate participation. While in another study [22], we investigated how the infrastructure of a Social Cloud can be co-operatively provided by the participating members, and present an economic model that takes individual incentives and resource availability into account.

[4] https://joindiaspora.com/.

6 Conclusion

As software crowdsourcing becomes an increasingly viable alternative to dedicated software development teams the infrastructure required to support such dynamic collaborations will continue to increase. Already, software crowdsourcing projects are turning to cloud resources as a model for providing resources on which tasks can be completed. However, we argue that these approaches will not scale and may become costly as projects become larger and more common. In this chapter we have described an alternative approach in which the very same crowdsourcing principles are applied to acquire infrastructure resources. Through the use of an infrastructure crowdsourcing model users can assemble a virtual infrastructure on which software development processes can be performed.

Social Clouds are a well studied approach for facilitating the exchange of infrastructure resources using various economic and non-economic protocols. They provide a model for exchanging heterogeneous resources between individuals connected via a social graph. Where the social graph provides the ability to derive relationships and therefore infer pre-existent trust between users, which in turn can be used to increase quality and trustworthiness with respect to shared infrastructure. Social Clouds provide the necessary mechanisms to ensure resources are used securely and to manage allocation across a pool of participants. In previous work, we have developed three Social Clouds, focused on storage, content delivery, and compute. We have also explored the use of different allocation protocols based on credit, social network, and preference models. These Social Clouds and their associated allocation models provide a generic basis on which other services can be developed.

The use of a Social Cloud model as a basis for, or a companion to, a software crowdsourcing system will enable individual projects to leverage not only the skills contributed by participants but also their infrastructure resources. This integration will allow the deployment and operation of important software development services hosted collectively by the project's members. The use of a infrastructure crowdsourcing approach is perhaps the most appropriate model for provisioning infrastructure given that the philosophies behind crowdsourcing software and infrastructure are the same. Finally, the same social network analysis algorithms used in a Social Cloud to infer trust and competency may also provide value in a software crowdsourcing model.

References

1. Ali, Z., Rasool, R.U., Bloodsworth, P.: Social networking for sharing cloud resources. In: 2012 Second International Conference on Cloud and Green Computing (CGC), pp. 160–166 (2012)
2. Anderson, D.P.: Boinc: a system for public-resource computing and storage. In: 5th IEEE/ACM International Workshop on Grid Computing, pp. 4–10 (2004)
3. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Nakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web services agreement specification (WS-agreement). In: Open Grid Forum, vol. 128 (2007)

4. Cappos, J., Beschastnikh, I., Krishnamurthy, A., Anderson T.: Seattle: a platform for educational cloud computing. In: The 40th Technical Symposium of the ACM Special Interest Group for Computer Science Education (SIGCSE'09), Chattanooga, TN USA (2009)
5. Caton, S., Haas, C., Chard, K., Bubendorfer, K., Rana, O.: A social compute cloud: allocating and sharing infrastructure resources via social networks (2014)
6. Caton, S., Dukat, C., Grenz, T., Haas, C., Pfadenhauer, M., Weinhardt, C.: Foundations of trust: contextualising trust in social clouds. In: 2012 Second International Conference on Cloud and Green Computing (CGC), pp. 424–429. IEEE (2012)
7. Caton, S., Rana, O.: Towards autonomic management for cloud services based upon volunteered resources. Concurr. Comput.: Pract. Exp. **23** (2011). Special Issue on Autonomic Cloud Computing: Technologies, Services, and Applications
8. Chard, K., Bubendorfer, K.: Using secure auctions to build a distributed meta-scheduler for the grid. In: Buyya, R., Bubendorfer, K. (eds.) Market Oriented Grid and Utility Computing. Wiley Series on Parallel and Distributed Computing, pp. 569–588. Wiley, New York (2009)
9. Chard, K., Bubendorfer, K., Caton, S., Rana, O.: Social cloud computing: a vision for socially motivated resource sharing. IEEE Trans. Serv. Comput. **99**(PrePrints), 1 (2012)
10. Chard, K., Caton, S., Rana, O., Bubendorfer, K.: Social cloud: cloud computing in social networks. In: 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD), pp. 99–106 (2010)
11. Chard, K., Caton, S., Rana, O., Katz, D.S.: A social content delivery network for scientific cooperation: vision, design, and architecture. In: The Third International Workshop on Data Intensive Computing in the Clouds (DataCloud 2012) (2012)
12. Czajkowski, K., Ferguson, D.F., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W.: The WS-resource framework. Technical report, Globus. http://www.globus.org/wsrf/specs/ws-wsrf.pdf (2004). Accessed Dec 2010
13. Dukat, C., Caton, S.: Towards the competence of crowdsourcees: literature-based considerations on the problem of assessing crowdsourcees' qualities. In: International Workshop on Crowdwork and Human Computation at the IEEE Third International Conference on Cloud and Green Computing (CGC), pp. 536–540. IEEE (2013)
14. Foster, I.: Globus online: accelerating and democratizing science through cloud-based services. IEEE Internet Comput. **15**(3), 70–73 (2011)
15. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the grid: enabling scalable virtual organizations. Int. J. High Perform. Comput. Appl. **15**, 200–222 (2001)
16. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. Am. Math. Mon. **69**, 9–15 (1962)
17. Geiger, D., Seedorf, S., Schulze, T., Nickerson, R.C., Schader, M.: Managing the crowd: towards a taxonomy of crowdsourcing processes. In: AMCIS (2011)
18. Gracia-Tinedo, R., Sanchez-Artigas, M., Garcia-Lopez, P.: Analysis of data availability in F2F storage systems: when correlations matter. In: 2012 IEEE 12th International Conference on Peer-to-Peer Computing (P2P), pp. 225–236. IEEE (2012)
19. Gracia-Tinedo, R., Sánchez-Artigas, M., Garcia-Lopez, P.: F2box: cloudifying F2F storage systems with high availability correlation. In: 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 123–130. IEEE (2012)
20. Gracia-Tinedo, R., Sánchez-Artigas, M., Moreno-Martinez, A., Garcia-Lopez, P.: Friendbox: a hybrid F2F personal storage application. In: 2012 IEEE 5th International Conference on Cloud Computing (CLOUD), pp. 131–138. IEEE (2012)
21. Haas, C., Caton, S., Weinhardt, C.: Engineering incentives in social clouds. In: Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2011), pp. 572–575 (2011)
22. Haas, C., Caton, S., Chard, K., Weinhardt, C.: Co-operative infrastructures: an economic model for providing infrastructures for social cloud computing. In: Proceedings of the Forty-Sixth Annual Hawaii International Conference on System Sciences (HICSS), Grand Wailea, Maui, USA (2013)

23. Haas, C., Kimbrough, S., Caton, S., Weinhardt, C.: Preference-based resource allocation: using heuristics to solve two-sided matching problems with indifferences. In: 10th International Conference on Economics of Grids, Clouds, Systems, and Services (Under Review) (2013)
24. Halldórsson, M.M., Iwama, K., Miyazaki, S., Yanagisawa, H.: Improved approximation results for the stable marriage problem. ACM Trans. Algorithms (TALG) **3**(3), 30 (2007)
25. Ipeirotis, P.G., Provost, F., Wang, J.: Quality management on Amazon Mechanical Turk. In: Proceedings of the ACM SIGKDD Workshop on Human Computation, pp. 64–67. ACM (2010)
26. Irving, R.W., Leather, P., Gusfield, D.: An efficient algorithm for the optimal stable marriage. J. ACM **34**(3), 532–543 (1987)
27. John, K., Bubendorfer, K., Chard, K.: A social cloud for public eResearch. In: Proceedings of the 7th IEEE International Conference on eScience. Stockholm, Sweden (2011)
28. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
29. Kern, R., Zirpins, C., Agarwal, S.: Managing quality of human-based eservices. Service-Oriented Computing-ICSOC 2008 Workshops, pp. 304–309. Springer, New York (2009)
30. Kittur, A., Nickerson, J.V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M., Horton, J.: The future of crowd work. In: Proceedings of the 2013 Conference on Computer Supported Cooperative Work, pp. 1301–1318. ACM (2013)
31. Kuada, E., Olesen, H.: A social network approach to provisioning and management of cloud computing services for enterprises. In: The Second International Conference on Cloud Computing, GRIDs, and Virtualization, CLOUD COMPUTING 2011, pp. 98–104 (2011)
32. Kugler, K., Chard, K., Caton, S., Rana, O., Katz, D.S.: Constructing a social content delivery network for escience. In: 2013 IEEE 9th International Conference on eScience (eScience), pp. 350–356 (2013)
33. Lonstein, E., Lakhani, K., Garvin, D.: Topcoder (a): developing software through crowdsourcing. Technical report, Harvard Business School General Management Unit Case (2010)
34. McMahon, A., Milenkovic, V.: Social volunteer computing. J. Syst. Cybern. Inf. (JSCI) **9**(4), 34–38 (2011)
35. Mohaisen, A., Tran, H., Chandra, A., Kim, Y.: Socialcloud: using social networks for building distributed computing services. arXiv:1112.2254 (2011)
36. Narendula, R., Papaioannou, T.G., Aberer, K.: My3: a highly-available P2P-based online social network. In: 2011 IEEE International Conference on Peer-to-Peer Computing (P2P), pp. 166–167. IEEE (2011)
37. Oleson, D., Sorokin, A., Laughlin, G.P., Hester, V., Le, J., Biewald, L.: Programmatic gold: targeted and scalable quality assurance in crowdsourcing. Hum. Comput. **11**, 11 (2011)
38. Pezzi, R.: Information technology tools for a transition economy, September 2009
39. Pfadenhauer, M.: Competence-more than just a buzzword and a provocative term? Modeling and Measuring Competencies in Higher Education, pp. 81–90. Springer, New York (2013)
40. Pisano, G.P., Verganti, R.: Which kind of collaboration is right for you. Harv. Bus. Rev. **86**(12), 78–86 (2008)
41. Thal, R.: Representing agreements in social clouds. Master's Thesis, Karlsruhe Institute of Technology (2013)
42. Thaufeeg, A.M., Bubendorfer, K., Chard, K.: Collaborative eResearch in a social cloud. In: 2011 IEEE 7th International Conference on E-Science (e-Science), pp. 224–231 (2011)
43. Tokarchuk, O., Cuel, R., Zamarian, M.: Analyzing crowd labor and designing incentives for humans in the loop. IEEE Internet Comput. **16**(5), 45–51 (2012)
44. Wu, W., Tsai, W.-T., Li, W.: An evaluation framework for software crowdsourcing. Front. Comput. Sci. **7**(5), 694–709 (2013)
45. Wu, W., Zhang, H., Li, Z.: Open social based collaborative science gateways. In: 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp, 554–559. IEEE (2011)
46. Wu, W., Zhang, H., Li, Z., Mao, Y.: Creating a cloud-based life science gateway. In: 2011 IEEE 7th International Conference on E-Science (e-Science), pp. 55–61. IEEE (2011)
47. Zhang, Y., van der Schaar, M.: Incentive provision and job allocation in social cloud systems. To appear in IEEE J. Sel. Areas Commun. (2013)

Recommending Web Services Using Crowdsourced Testing Data

Hailong Sun, Wancai Zhang, Minzhi Yan and Xudong Liu

Abstract With the rapid growth of Web Services in the past decade, the issue of QoS-aware Web service recommendation is becoming more and more critical. Web service QoS is highly relevant to the corresponding invocation context like invocation time and location. Therefore, it is of paramount importance to collect the QoS data with different invocation context. We have crawled over 30,000 Web services distributed across Internet. In this work, we propose to use crowdsourcing to collect the required QoS data. This is achieved through two approaches. On the one hand, we deploy a generic Web service invocation client to 343 Planet-Lab nodes and these nodes serve as simulated users distributing worldwide. The Web service invocation client is scheduled to invoke target Web services from time to time. On the other hand, we design and develop a mobile crowdsourced Web service tesing framework on Android platform, with which a user can easily invoke selected Web services. With the above two approaches, the observed service invocation data, e.g. response time, will be collected in this way. Then we design a Temporal QoS-Aware Web Service Recommendation Framework to predict missing QoS value under various temporal context. Further, we formalize this problem as a generalized tensor factorization model and propose a Non-negative Tensor Factorization (NTF) algorithm which is able to deal with the triadic relations of user-service-time model. Extensive experiments are conducted based on collected Crowdsourced testing data. The comprehensive experimental analysis shows that our approach achieves better prediction accuracy than other approaches.

H. Sun (✉) · W. Zhang · M. Yan · X. Liu
School of Computer Science and Engineering, Beihang University, Beijing, China
e-mail: sunhl@act.buaa.edu.cn

W. Zhang
e-mail: zhangwc@act.buaa.edu.cn

M. Yan
e-mail: yanmz@act.buaa.edu.cn

X. Liu
e-mail: liuxd@act.buaa.edu.cn

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_12

1 Introduction

Service oriented computing [9] promises to enable efficient software development through composing reusable services with standard interfaces. And Web services [17] are the most important category of software services, which can be accessed with standard protocols like SOAP (Simple Object Access Protocol) and HTTP. In most cases, Web services are provided by third-party providers and users invoke services with no need for knowing the details of service implementation and runtime environment.

In Web service area, service discovery, which is known as a hot research issue, aims at finding a best suitable service instance according to user requirements in terms of both functionality and non-functionality. There has been a large body of work [6, 8, 12] focusing on addressing discovery of services meeting users' functional requirements, and many approaches [5, 20] based on information retrieval methods have been proposed to deal with the functional matchmaking between user requirements and candidate services.

Typical non-functionality requirements include response time, reliability, cost, availability, throughput etc. Usually non-functional attributes are called QoS (Quality of Services) attributes. In recent years, much attention has been drawn on recommending Web services to users by analyzing Web service invocation records of users. In this regards, model-based methods and collaborative filtering based methods [3, 7, 22] are utilized to achieve this goal. Service recommendation mainly considers services' non-functional properties. The emergence of multiple Web services with the same functionality offers a set of alternatives for users to choose from based on their QoS. With the growing number of functionally equivalent services on the web, it is quite important to recommend services by considering their non-functional QoS properties. The high QoS of Web services can help service users to reduce re-engineering cost and to produce more effective service-oriented systems. There are mainly three reasons that can explain why services with the same functionality can have different non-functional attributes. First, there can be multiple versions of implementation for the same service functionality, and each implementation can produce different non-functional performance due to adoption of different architecture and algorithms. Second, even for the same service implementation, there can be multiple service instances deployed onto various runtime environments that can greatly influence the delivered QoS attributes. Third, from the users' perspective, invocation time, physical location and client configuration will also leads to different non-functional experience. It is necessary to take all these issues into consideration to design a good service discovery solution. Intuitively, we can greatly improve recommendation effectiveness if we have enough service invocation data with invocation contexts similar to that of the target user. Thus it is highly desirable to collect a large amount of invocation data from users, which cannot be done without large-scale user participation.

Moreover, crowdsourcing [4], as a fast growing field, aims at leveraging an undefined set of people to build a distributed problem solving environment for certain

task processing. Crowdsourcing is especially effective in solving problems that are difficult for computers, e.g. image labeling, natural language processing, and object recognition. Many successful applications like *Wikipedia*, *reCaptcha*, *Google Image Labeler* have been built with crowdsourcing. In essence, the core concept of crowdsourcing is the active participation of users, in which users are not only application consumers, but also application system contributors.

In this work, we propose to leverage the crowdsoucing approach to collect as much as possible service QoS data. With this data, we hope to design effective recommendation algorithms to recommend appropriate Web services for users in specific time and location context. The data collecting is realized with the following two efforts: on the one hand, we simulate service users with computing nodes in Planetlab[1] testbed and the selected nodes are highly distributed in geographical location; on the other hand, we develop an Android App that can be used by Android mobile device users to invoke Web services. With the obtained service invocation data, we propose a temporal QoS-Aware Web service prediction framework, which employs an novel Collaborative Filtering (CF) algorithm for Web service recommendation. The CF method can predict the missing QoS value of Web services by employing their historical QoS values. So we should be able to predict the missing QoS value of Web services from distribution location users at different invocation time to optimize the Web services recommendation.

This chapter is organized as follows. In Sect. 2, we present the design and implementation of a crowdsourced testing framework for collecting Web service QoS data. Section 3 describe a tensor-based model and corresponding collaborative filtering algorithms for recommending services to users. In Sect. 4, we perform an extensive set of experiments with the obtained service QoS data to evaluate the recommendation algorithms. Finally Sect. 6 concludes this work.

2 QoS Data Collection with Crowdsourced Testing

Since the Web service QoS experienced by users are closely related to the invocation context like client configuration, location and invocation time, we hope to obtain as much invocation context data and corresponding observed QoS data as possible. According to the above analysis, crowdsourcing provides an effective means to achieve this goal. Considering people are more and more reliant on mobile devices, we design iTest, a mobile crowdsourcing based testing framework that can facilitate the invocation of Web services from Mobile devices. With iTest, volunteers can help test certain Web services whenever and wherever they go. However, crowdsourcing applications usually face a challenge to design an effective incentive mechanism so as to attract people to participate. To complement the weak points of mobile crowdsourcing approach, we simulate a large scale of users with cloud nodes to test the targeted Web services.

[1] https://www.planet-lab.org/

2.1 Real Web Service Crawling

In this work, we are concerned with real Web services functioning over Internet. Therefore we must collect the information of the target Web services, which is done using Web crawling technique. Actually, this is part of work in our Service4All[2] [16], a PaaS (Platform as a Service) platform facilitating the development of service oriented software. ServiceXchange is a subsystem of Service4All, which is responsible for collecting services on the web, analyzing the relationships of them, and continuously monitoring their responses for fetching QoS information. Developers can search services by keywords, names, tags and even potential service relations with the QoS requirements.

Now there are about 30,000 Web services collected in ServiceXchange. Developers can find their required services and then subscribe and reuse them to build complex service-oriented applications. Although ServiceXchange has been integrated with Service4All, it can still work as a stand-alone platform that support service discovery for external users. In this work, we are mainly concerned with the testing of the collected 30,000+ Web services in ServiceXchange.

2.2 Simulated Crowd Testing with Cloud

Although crowdsourcing is very helpful in obtaining the required data in this work, there are still a few challenging issues to deal with to build an effective crowdsourced service testing solution. As there are a lot of distributed computing nodes available over Internet, we can approximate crowdsourcing by simulating various invoking context in terms of invocation time and location with those real physical resources. In this section, we give an overview of WS-TaaS [18], a Web service load testing platform using Planetlab.

2.2.1 Requirements of WS-TaaS

To design a cloud-based Web Service load testing environment, we need to analyze its requirements and take advantage of the strong points of cloud testing. Taking above aspects into account, we conclude that the following features should be guaranteed when building WS-TaaS: transparency, elasticity, geographical distribution, massive concurrency and sufficient bandwidth.

- *Transparency.* The transparency in WS-TaaS is divided into two aspects. (1) Hardware Transparency: testers have no need to know exactly where the test nodes are deployed. (2) Middleware Transparency: when the hardware environment is ready, the testing middlewares should be prepared automatically without tester involvement.

[2] http://www.service4all.org.cn

- *Elasticity.* All the test capabilities should scale up and down automatically commensurate with the test demand. In WS-TaaS, the required resources of every test task should be estimated in advance to provision more or withdraw the extra ones.
- *Geographical Distribution*. To simulate the real runtime scenario of a web service, WS-TaaS is required to provide geographically distributed test nodes to simulate multiple users from different locations all over the world.
- *Massive Concurrency and Sufficient Bandwidth*. As in Web Service load testing process the load span can be very wide, so WS-TaaS have to support massive concurrent load testing. Meanwhile, the bandwidth need to be sufficient accordingly.

2.2.2 Conceptual Architecture of WS-TaaS

In a cloud-based load testing environment for Web Service, we argue that these four components are needed: Test Task Receiver & Monitor (TTRM), Test Task Manager (TTM), Middleware Manager and TestRunner. Figure 1 shows the conceptual architecture of a cloud-based Web Service load testing system, including the four main components above, which we explain as follows:

- *Test Task Receiver & Monitor*. TTRM is in charge of supplying testers with friendly guide to input test configuration information and submitting test tasks. The testing process can also be monitored here.
- *Test Task Manager*. TTM manages the queue of test tasks and dispatchs them to test nodes in the light of testers' intention, and then gathers and merges the test results.
- *TestRunner*. TestRunners are deployed on all test nodes and play the role of web service invoker. They can also analyse the validity of web service invocation results.
- *Middleware Manager*. Middleware Manager manages all the TestRunners and provide available TestRunners for TTM with elasticity.

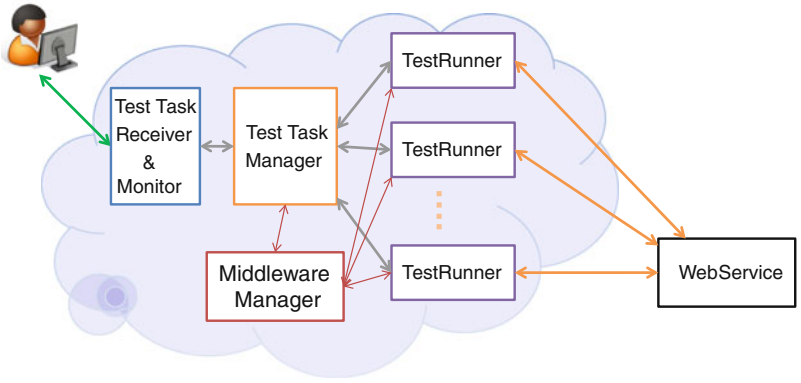


Fig. 1 Conceptual architecture of a web service cloud Testing System

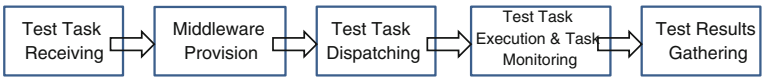


Fig. 2 Workflow of WS-TaaS

2.2.3 Testing Workflow of WS-TaaS

The workflow of WS-TaaS is divided into five steps, as shown in Fig. 2.

(1) Firstly, TTRM receives configuration of a test task submitted by the tester and transmit it to TTM. (2) Then TTM selects test nodes from available nodes in terms of their performance, and asks Middleware Manager to provision TestRunners on the test nodes selected. (Please see Sect. 4.1 for details). (3) After that, TTM decides the numbers of concurrent request of each node selected and divides the test task into subtasks with specific scheduling and dispatching strategies (The details of the strategies are shown in Sect. 4.2), and then dispatches them to the test nodes selected. (4) Then, after dispatching the test task, TTM notifies all the participanting TestRunners to start invoking the target service simultaneously. During test task execution, TTRM periodically sends query requests to TTM at a fixed time interval to obtain the execution state. (5) In the meantime, TTM also periodically queries the test results from all the involved TestRunners. On receiving a query response, the intermediate test results will be displayed to users. The query and gathering process will be continued until the test task finishes.

2.2.4 Test Mode of WS-TaaS

For different kinds of test needs, WS-TaaS provides three test modes as follows.

- *Static Test.* Static test is provided to test the performance of a web service under user-specified load. In such a test, the test task is just deployed and executed once.
- *Step Test.* In a step test, the tester need to input the start number, step size and the end number of concurrent requests. Then the test task is deployed and executes step by step with different numbers of concurrent requests, which increase by the step size from the start number till it meets the end one. Step test can tell the tester how the web service would offer usability in a specific load span.
- *Maximal Test.* Maximal test is used to determine the load limit of a web service. Like a step test, the start number and step size is needed, but the end number is not needed. So the strategy for judging whether the current load is the load limit is required. At present, we briefly define judgement principle as follows: If more than 10 % of the concurrent requests are failed (the invoking result is inconsistent

with the expected result or the invoking process timed out) under a load, then we define this failure and take this load amount as the load limit of the target web service. This determination is made by Test Task Manager with the calculation of failure percentage. Once the load limit is determined, this test task will be stopped and the final report is generated for the tester.

2.3 iTest: Testing Web Services with Mobile Crowdsourcing

Considering that Web service invocation is highly related with invocation context like time and location, we choose to leverage mobile phones to collect the Service QoS data since the use of mobile phone is largely diversified in terms of geo-location, time and client environments. Invoking a Web service usually involves a lot of programming work to construct, send, receive and parse SOAP messages, therefore we must hide the programming complexity so as to let users easily fulfill the Web service testing task without knowing too much technical details.

As mentioned in Sect. 2.1, we have collected over 30,000 Web services. The WSDL files and corresponding information about these services are stored in a repository, i.e. WSR in Fig. 3. WSR maintains all the services needed to be tested. On the client side, we have developed, iTest, an Android App based on PhoneGap framework that provides support for developing mobile apps with standard Web technologies.

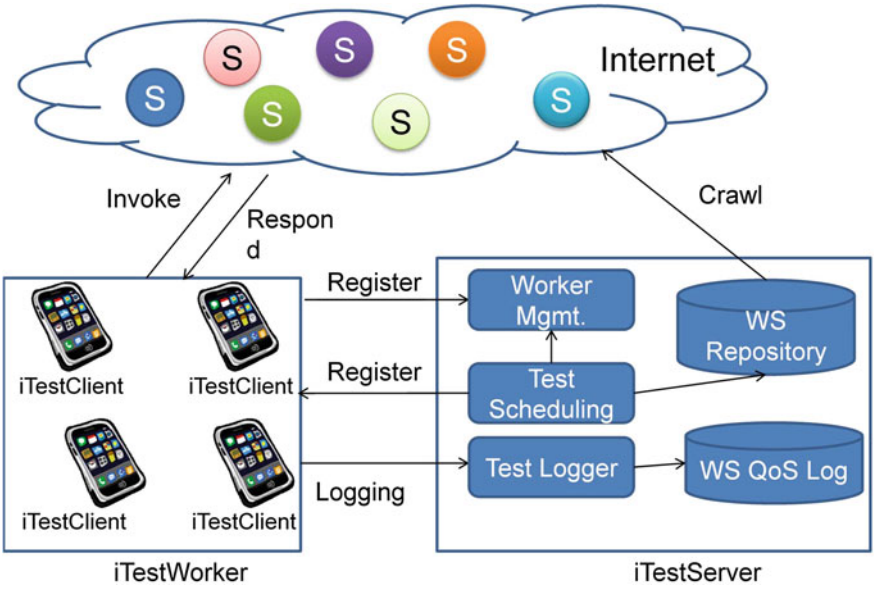


Fig. 3 iTest: testing web services with mobile crowdsourcing

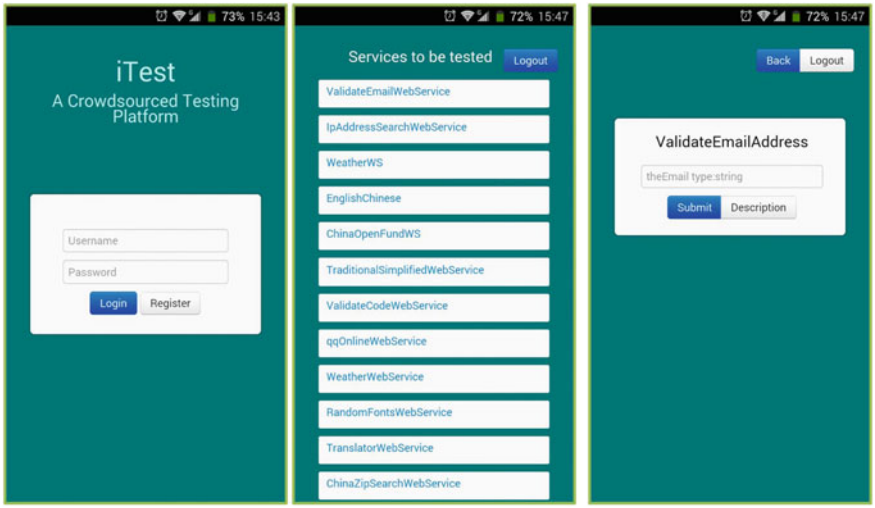


Fig. 4 Screenshots of iTestClient

Currently, we only implement iTestClient on Android platform, as shown in Fig. 4. Once iTestClient is installed on an Android mobile phone, the mobile phone will be registered with the server side as a Web service testing volunteer called iTestWorker. The iTestServer is responsible for managing all the registered volunteers, scheduling testing tasks and maintaining QoS data. When an iTestWorker goes online, it will notify iTestServer and the latter will respond with a list of Web services to be tested. Then the user can select a Web service to test. iTestClient fetches the WSDL file of the selected Web service through HTTP protocol and it then generate an GUI interface for users to fill in with the necessary parameter values. Once the user click the "invoke" button, a SOAP request for the target Web service will be sent out and iTestClient will wait for the response from the target Web service synchronously. During this process, iTestClient logs the GPS location, time to invoke the service and time to get the response from the service. All the logged information will be sent to iTestServer.

3 Temproal QoS-Aware Web Service Recommendation with Crowdsourced Testing Data

Service QoS properties are not the same as what service providers have declared since practical QoS data is influenced by user contexts. To obtain accurate Web service QoS value for a certain service user, basing on the QoS dataset obtained through crowdsourced testing described in Sect. 2 we propose a temporal QoS-aware Web service recommendation framework [19] to make prediction of missing QoS

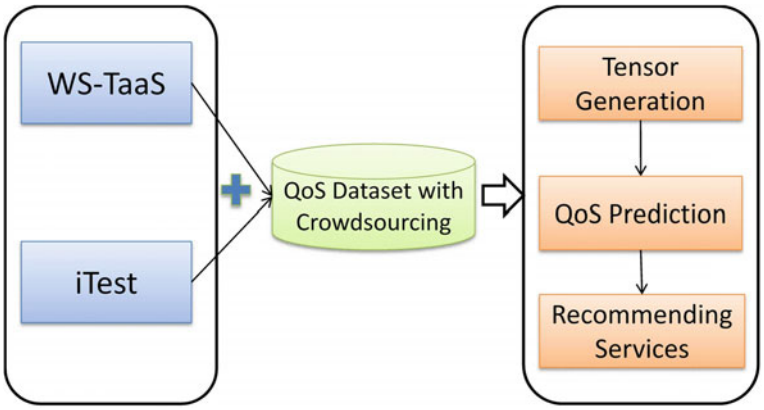


Fig. 5 Temporal QoS-Aware Web Service Recommendation Framework

values. As shown in Fig. 5, our QoS prediction framework collects Web services QoS information from different service users. A Web service user can obtain the service QoS value prediction through our prediction framework, if the service QoS information contributions of the user surpass the threshold. The more service QoS information contributions, the higher QoS value prediction accuracy can be achieved. After collecting a large number of QoS information, we filter some inferior QoS information for the training data and employ the prediction engine to generate the predictor model for predicting the missing QoS value. Due to the space limitation, we mainly introduce the prediction algorithm principle in this paper.

3.1 Problem Formulation

The research problem studied in this work is stated as follows: *Given a Web service QoS dataset of temporal information with user-service interactions, recommend to each user under a given temporal context an optimal services list.* To illustrate these concepts, the following example is given.

A Toy Example: Consider the instance of recommending services to users in specific temporal context which is assigned to service invocation time in this paper. Then the $\langle user, service, time \rangle$ triplets have the following attributes:

- *User: the set of all service users to whom Web services are recommended; it is defined as UserID.*
- *Service: the set of all the Web services that can be recommended; it is defined as ServiceID.*
- *Time: the Web service invocation time when the user invoke the service; it is defined as TimeID.*

Then the service QoS value assigned to a service invocation from a user also depends on where and when the service was invoked. For instance, a specific service is recommended to users in different locations, significantly depending on when they are planning to invoke it.

Each QoS value is described by three dimensionality according to *userID*, *serviceID* and *timeID*. Thus the QoS value is represented as points in the three-dimensional space, with the coordinates of each point corresponding to the index of the triplet ⟨*userID*, *serviceID*, *timeID*⟩. A straightforward method to capture the three-dimensional interactions among the triplet ⟨*user*, *service*, *time*⟩ is to model these relations as a tensor. The QoS value of Web service invocations from J services by I users at K time intervals are denoted as a tensor $\mathcal{Y} \in \mathbb{R}^{I \times J \times K}$, i.e., a three-dimensional tensor, with $I \times J \times K$ entries which are denoted as $\mathcal{Y}_{ijk}$: (1) $\mathcal{Y}_{ijk} = Rating$ indicates the missing QoS value that the service j has been invoked by user i under the context type k, and *Rating* is this service QoS value; (2) $\mathcal{Y}_{ijk} = 0$ indicates that the service has not been invoked. The real-world Service QoS value dataset is very sparse, even though the density of the dataset collected by our system is only 30 %.

To obtain the missing QoS value in the *user-service-time* tensor, the Web service QoS observed by other service users can be employed for predicting the Web service for the current user. Once these initial Web service QoS value is obtained, our recommendation system will try to estimate the Web service QoS value which has not been obtained for the $\langle user, service, time \rangle$ triplets by using the QoS value function T:

$$UserID \times ServiceID \times TimeID \rightarrow Rating$$

where *UserID*, *ServiceID* and *TimeID* are the index of users, services and time periods, respectively and $Rating$ is the QoS value corresponding to the three-dimensional index.

As we can see from this example and other cases, an algorithm is needed to estimate the QoS value function T. In this paper, CP decomposition model is used to reconstruct the temporal three-dimensional *user-service-time* tensor. As mentioned in Sect. 3, the main idea behind CP decomposition model is to find a set of low-rank tensors to approximate the original tensor. Our approach is designed as a two-phase process. Firstly, the temporal QoS value tensor composed of the observed QoS value is constructed. Then we propose a non-negative tensor factorization approach to predict the missing QoS value in the tensor.

3.2 Construct QoS Value Tensor

When a service user invokes a Web service, the QoS properties performance will be collected by our recommendation system. After running a period of time, the recommender accumulates a collection of Web service QoS property data, which can be represented by a set of quadruplets ⟨*UserID*, *ServicID*, *TimeID*, *Rating*⟩

(or $\langle u, s, t, r \rangle$ for short). Using the QoS value data, a temporal three-dimensional tensor $\mathcal{Y} \in \mathbb{R}^{I \times J \times K}$ can be constructed, where I, J, K are the number of users, services and time periods, respectively. Each entry of tensor represents the QoS value of $\langle u, s \rangle$ pair at time period k.

The three-dimensional Temporal Tensor Construct algorithm is given in Algorithm 1: the input is a set of Web service QoS value, and the output is the constructed temporal tensor $\mathcal{Y} \in \mathbb{R}^{I \times J \times K}$. Each frontal slice in tensor $\mathcal{Y}$ corresponds to a $\langle u, s \rangle$ pair QoS value matrix for each time interval.

Algorithm 1: Temporal Tensor Construct

Input: a set of quadruplets $\langle u, s, t, r \rangle$ for Web service QoS value dataset.

Output: a temporal tensor $\mathcal{Y} \in \mathbb{R}^{I \times J \times K}$.

1: load all quadruplets $\langle u, s, t, r \rangle$ of the Web service Qos value,
2: use the set of $\langle u, s, 1, r \rangle$ to construct a *user, service* matrix $\mathbf{U}^{(1)}$ that takes all I users as the rows and all J services as the columns in the time of period 1,
3: the element of the matrix $\mathbf{U}^{(1)}$ is the r of the quadruplet $\langle u, s, t, r \rangle$ according to the corresponding $\langle u, s, 1 \rangle$ triplet,
4: construct all the matrices $\mathbf{U}^{(1)}, \mathbf{U}^{(2)}, \cdots, \mathbf{U}^{(K)}$ for K time periods,
5: an augmented matrix $\mathbf{U}$ can be built by horizontally concatenating all matrices as shown in Fig. 6 (a) denoted as $\mathbf{Y}_{(1)}$,
6: Construct tensor $\mathcal{Y} \in \mathbb{R}^{I \times J \times K}$ as shown in Fig. 6b, each slice of tensor is one matrix of $\mathbf{Y}_{(1)}$.
7: **Return**: $\mathcal{Y} \in \mathbb{R}^{I \times J \times K}$

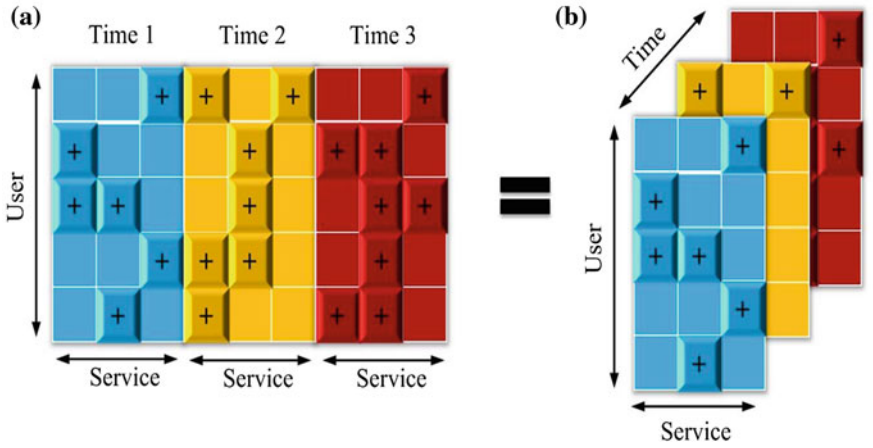


Fig. 6 Slices of time-specific matrices with users and services are transformed into a temporal tensor

3.3 Non-negative CP Decomposition

In the real-world, the Web service QoS value is always non-negative, so the temporal QoS value tensor is presented as an non-negative three-way tensor $\mathcal{Y} \in \mathbb{R}_+^{I \times J \times K}$, and decomposed components are a set of matrices: $\mathbf{U} \in \mathbb{R}_+^{I \times R_\mathcal{Y}}$, $\mathbf{S} \in \mathbb{R}_+^{I \times R_\mathcal{Y}}$ and $\mathbf{T} \in \mathbb{R}_+^{I \times R_\mathcal{Y}}$, here and elsewhere, $\mathbb{R}_+$ denotes the non-negative orthant with appropriate dimensions. As presented in the previous section, our goal is to find a set of factor matrices as the to approximate the tensor, whose rank is the number of the components. Adding the nonnegativity restriction to the CP decomposition model, we can get a non-negative CP decomposition model (NNCP). Our three-dimensional NNCP decomposition model is given by:

$$\mathcal{Y} = \sum_{r=1}^{R_\mathcal{Y}} \mathbf{u}_r \circ \mathbf{s}_r \circ \mathbf{t}_r + \mathcal{E}, \tag{1}$$

where the vectors $\mathbf{u}_r, \mathbf{s}_r, \mathbf{t}_r$ are restricted to have only non-negative elements and the tensor $\mathcal{E} \in \mathbb{R}_+^{I \times J \times K}$ is errors or noise depended on the application.

The QoS value tensor should be reconstructed for predicting all missing QoS values. A new fitting algorithm which approximates the tensor with non-negative value should be designed. Firstly, we define a cost function to quantify the quality of approximation, which can be constructed using some measure of distance between two non-negative tensors $\mathcal{Y}$ and $\hat{\mathcal{Y}}$. One useful measure is simply the square of the Euclidean distance between $\mathcal{Y}$ and $\hat{\mathcal{Y}}$,

$$\|\mathcal{Y} - \hat{\mathcal{Y}}\|_F^2 = \sum_{ijk} (\mathcal{Y}_{ijk} - \hat{\mathcal{Y}}_{ijk})^2, \tag{2}$$

where $\mathcal{Y}_{ijk}$ is the Web service QoS value of jth service from ith user at k-time, $\hat{\mathcal{Y}}_{ijk}$ is the approximation value, the lower bound is zero, and clearly vanishes if and only if $\mathcal{Y} = \hat{\mathcal{Y}}$. Then, we consider the formulations of NNCP as a optimal problem:

$$\min_{\mathbf{u}_r, \mathbf{s}_r, \mathbf{t}_r} \frac{1}{2} \|\mathcal{Y}_{ijk} - \sum_{r=1}^{R_\mathcal{Y}} \mathbf{u}_r \circ \mathbf{s}_r \circ \mathbf{t}_r\|_F^2,$$
$$s.t. \quad \mathbf{u}_r, \mathbf{s}_r, \mathbf{t}_r \geqslant 0. \tag{3}$$

We use multiplicative updating algorithms [13] for factor matrices $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{T}$ to approximate the non-negative tensor. Then we are easy to obtain the partial derivative of the objective Eq. (3):

$$\frac{\partial f}{\partial \mathbf{u}_l^{(i)}} = \sum_{r=1}^{R_\mathcal{Y}} \mathbf{u}_l^{(i)} (\mathbf{s}_r \cdot \mathbf{s}_l)(\mathbf{t}_r \cdot \mathbf{t}_l) - \sum_{j,k} \mathcal{Y}_{ijk} \mathbf{s}_l^{(j)} \mathbf{t}_l^{(k)} \tag{4}$$

where $\mathbf{u}_l^{(i)}$ is the l-th column and ith row element of factor matrix $\mathbf{U}$, $\mathbf{s}_r$ is the rth vector of factor matrix $\mathbf{S}$, $l \in R_{\mathcal{Y}}$, $\cdot$ denotes the inner product and for more details see [15]. Then we can obtain the following update rule by using a multiplicative update rule:

$$\mathbf{u}_l^{(i+1)} \leftarrow \frac{\mathbf{u}_l^{(i)} \sum_{j,k} \mathcal{Y}_{ijk} \mathbf{s}_l^{(j)} \mathbf{t}_l^{(k)}}{\sum_{r=1}^{R_{\mathcal{Y}}} \mathbf{u}_r^{(i)} (\mathbf{s}_r \cdot \mathbf{s}_l)(\mathbf{t}_r \cdot \mathbf{t}_l)}, \tag{5}$$

the updating rules for the rest of factor matrices can be easily derived in the same way, $\mathbf{s}_l^{(n)}$ and $\mathbf{t}_l^{(n)}$ are shown as follows:

$$\mathbf{s}_l^{(j+1)} \leftarrow \frac{\mathbf{s}_l^{(j)} \sum_{i,k} \mathcal{Y}_{ijk} \mathbf{u}_l^{(i)} \mathbf{t}_l^{(k)}}{\sum_{r=1}^{R_{\mathcal{Y}}} \mathbf{s}_r^{(j)} (\mathbf{u}_r \cdot \mathbf{u}_l)(\mathbf{t}_r \cdot \mathbf{t}_l)}; \tag{6}$$

$$\mathbf{t}_l^{(k+1)} \leftarrow \frac{\mathbf{t}_l^{(k)} \sum_{i,j} \mathcal{Y}_{ikl} \mathbf{u}_l^{(i)} \mathbf{s}_l^{(j)}}{\sum_{r=1}^{R_{\mathcal{Y}}} \mathbf{t}_r^{(k)} (\mathbf{u}_r \cdot \mathbf{u}_l)(\mathbf{s}_r \cdot \mathbf{s}_l)}, \tag{7}$$

where the vectors $\mathbf{u}_r, \mathbf{s}_r, \mathbf{t}_r$ are composed of non-negative value when they are initialized. So far we have described the details of NNCP algorithm for predicting the missing QoS value with non-negative value. In summary, Algorithm 2 gives the whole factorization scheme for NNCP. In each iteration of our algorithm, the new value of $\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{T}}$ is calculated by multiplying the current value by a factor depended on the quality of the approximation in Eq. (1). The quality of the approximation improves monotonically with the application of these multiplicative update rules. The convergence proof of the multiplicative rule was introduced by Lee and Seung [13].

Given the latent factor matrices $\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{T}}$, the prediction QoS value of Web service j from service user i at time k is given by:

$$\mathcal{Y}_{ijk} \approx \sum_{r=1}^{R_{\mathcal{Y}}} \mathbf{u}_r^{(i)} \mathbf{s}_r^{(j)} \mathbf{t}_r^{(k)}. \tag{8}$$

Notice that increasing the number $R_{\mathcal{Y}}$ of components allows us to represent more and more factor structures of the Web service QoS value. However, as the number of components increases, we go from under-fitting to over-fitting these structures, i.e., we face the usual tradeoff between approximating complex structures and over-fitting them.

4 Experiments

In this section, we introduce the experiment dataset, our evaluation metrics, and the experiment results. We use the QoS prediction accuracy to measure prediction quality, and address the following questions: (1) How do the tensor density and

Algorithm 2: Non-negative CP decomposition algorithm

Input: the tensor $\mathcal{Y} \in \mathbb{R}_{+}^{I \times J \times K}$, the rank R of tensor $\mathcal{Y}$.

Output: three non-negative factor matrices $\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{T}}$.

Procedure $[\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{T}}]$ = NNCP($\mathcal{Y}$, R)
Initialize: $\mathbf{U} \in \mathbb{R}_{+}^{I \times R}$, $\mathbf{S} \in \mathbb{R}_{+}^{J \times R}$, and $\mathbf{T} \in \mathbb{R}_{+}^{K \times R}$ by small non-negative value.
Repeat
for $l = 1, \cdots, I$ **do**
  $\hat{\mathbf{U}} \leftarrow Eq.(5)$
end for
for $l = 1, \cdots, J$ **do**
  $\hat{\mathbf{S}} \leftarrow Eq.(6)$
end for
for $l = 1, \cdots, K$ **do**
  $\hat{\mathbf{T}} \leftarrow Eq.(7)$
end for
Until convergence or maximum iterations exhausted.
Return: $\hat{\mathbf{U}}, \hat{\mathbf{S}}, \hat{\mathbf{T}}$
EndProcedure

factor matrices dimensionality influence prediction accuracy? The factor matrices dimensionality determines how many the latent factors which have direct influence on prediction accuracy. (2) How does our approach compare with other CF methods?

We implement the algorithms described in Sect. 3 with Matlab. For constructing the temporal QoS value tensor and solving the non-negative CP decomposition, we use the Matlab Tensor Toolbox [1]. The experiments are conducted on a Dell PowerEdge T620 machine with 2 Intel Xeon 2.00 GHz processors and 16 GB RAM, running Window Server 2008.

4.1 Dataset from Simulated Crowdsourced Testing

To evaluate the effectiveness of our recommendation methods, we use WS-TaaS, the simulated crowdsourcing approach, to test the QoS of realworld Web services. We use more than 600 distributed slices of Planet-Lab nodes and we select the slices which have successfully invoked at least 50 Web services so that there are enough observations to be split in various proportions of training and testing set for our evaluation. Finally, 343 slices were selected as the Web service users, and 5,817 publicly available real-world Web services are monitored by each slice continuously. The other of the more than 10,000 initially collected Web services are excluded in this experiment due to: (1) authentication required; (2) refused by the provider (e.g., the Web service is hosted by a private golf club); (3) permanent invocation failure (e.g., the Web service is shutdown). In this experiment, each 343 Planet-Lab slices invokes all the Web services continuously. The QoS data has been continuously collected from July 26. In this experiment, we only use a small dataset consisting of

Table 1 Statistics of web service QoS Value

Statistics	Response-Time	Throughput
Scale	0–200 s	0–1000 kbps
Mean	0.6840	7.2445
Num. of service users	343	343
Num. of web services	5817	5817
Num. of time periods	32	32

these Web services QoS performances of 4 days from July 26 to 29 of 2013 in 32 time intervals lasting for 3 h.

We collect Web service invocation records from all the slices, and represent one observation in the dataset as a quadruplet $\langle u, s, t, r \rangle$. The dataset contains more than 19 million quadruplets, 343 users, 5,817 services and 32 time periods. Finally, we obtain two $343 \times 5817 \times 32$ *user-service-time* tensors. One tensor contains response time value, and the other one contains throughput value. Response time is defined as the persistent time between a service user sending a request and receiving the corresponding response, while throughput is defined as the average rate of successful message size per second. The statistics of Web service QoS performance dataset are summarized in Table 1. The distributions of response time and throughput are shown in Fig. 7. In Table 1, the means of response-time is 0.6840 seconds and throughput is 7.2445 kbps. In Fig. 7a shows that more than 95 % of the response time elements are smaller than 1.6 s, and Fig. 7b shows that more than 99 % of the throughput elements are smaller than 100 kbps. In this paper we only study the response-time and throughput, our NNCP method can be used to predicting any other QoS value directly without modifications. The value of the element in the three-dimensional tensor is the corresponding QoS value, when predicting value of a certain QoS value (e.g., popularity, availability, failure probability, etc.).

4.2 Evaluation Measurements

Given a quadruplet $\langle u, s, t, r \rangle$ as $T = \langle u, s, t, r \rangle$, we evaluate the prediction quality of our method in comparison with other collaborative filtering methods using *Mean Absolute Error* (MAE) and *Root Mean Squared Error* (RMSE) [23]. MAE is defined as:

$$MAE = \frac{1}{|T|} \sum_{i,j,k} \left| \mathcal{Y}_{ijk} - \hat{\mathcal{Y}}_{ijk} \right| \tag{9}$$

where $\mathcal{Y}_{ijk}$ denotes actual QoS value of Web service j observed by user i at time period k, $\hat{\mathcal{Y}}_{ijk}$ represents the predicted QoS value of service j for user i at time period k, and $|T|$ is the number of predicted value. The MAE is the average absolute

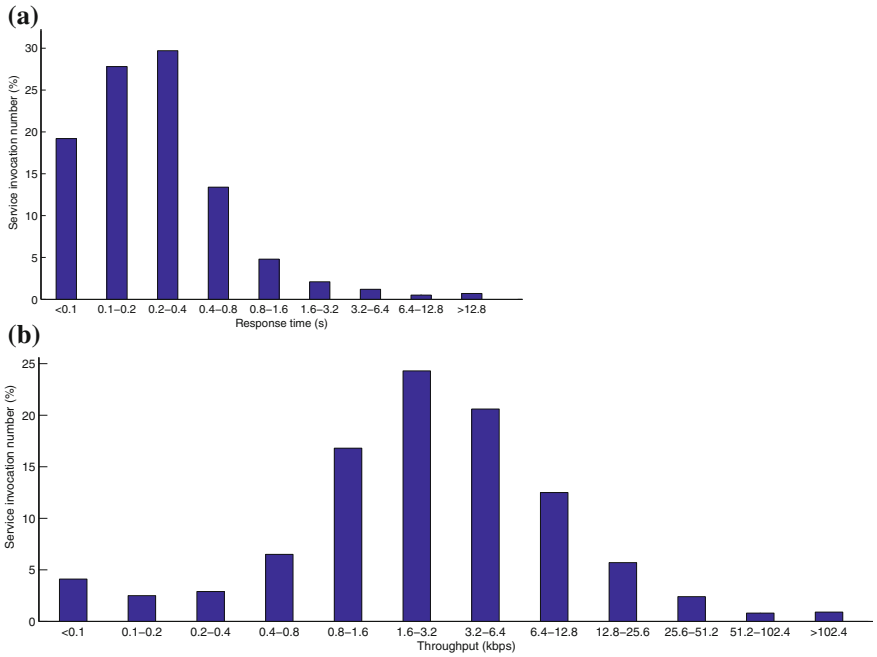


Fig. 7 Web services QoS distribution

deviation of predictions to the ground truth data, and places the equal weight on each individual difference. RMSE is defined as:

$$RMSE = \sqrt{\frac{1}{|T|}\sum_{i,j,k}\left(\mathcal{Y}_{ijk} - \hat{\mathcal{Y}}_{ijk}\right)^2} \tag{10}$$

where smaller MAE (or RMSE) indicates better prediction accuracy. Since the errors are squared before they are averaged, the RMSE gives extra weight to relatively large errors.

4.3 Baseline Algorithms

For comparison purpose, we investigate whether our approach can be captured by the following 6 baseline algorithms, according to the prediction performance measured on the dataset. The baselines involved in this comparative experiment are listed below:

- *UMean*: This method uses the mean QoS value of all Web services QoS value from a service user who has invoked these services to predict the missing QoS value which this service user has not invoked.
- *IMean*: This method uses the mean QoS value of each Web service QoS value from all service users who have invoked this service to predict the missing QoS value which other service users have not invoked this service.
- *UPCC* (User-based collaborative filtering method using Pearson Correlation Coefficient): This method is a classical CF algorithm that involves similar user behavior to make prediction [14].
- *IPCC* (Item-based collaborative filtering method using Pearson Correlation Coefficient): This method is widely used in e-commerce scenarios [11].
- *WSRec*: This method is a hybrid collaborative algorithm that combines both UPCC and IPCC approaches, and employs both the similar users and similar Web services for the QoS value prediction [21].
- *RSVD*: SVD (Singular Value Decomposition) is proposed by [2] in Collaborative Filtering area, and used to exploit the '*latent structure*' of the original data. In this paper, we use the regularized SVD method proposed in [10].

In this part, the above six baseline methods are compared with our NNCP approach given the same training and testing cases. Since the baseline algorithms cannot be directly applied to context-aware prediction problem, we employ a special formulation for making the comparison with our NNCP approach. We consider the three-dimensional *user-service-time* tensor as a set of *user-service* matrix slices in terms of time interval. Firstly, we compress the tensor into a *user-service* matrix. Each element of this matrix is the average of the specific $\langle user, service \rangle$ pair during all the time intervals. For each slice of the tensor, the baseline algorithms are applied for predicting the missing QoS value. Secondly, we compute the MAE and RMSE of these baselines, and make the comparison with our NNCP method.

In the real-world, the dataset is usually very sparse since a service user usually only invokes a very small number of Web services. We randomly remove QoS value to sparse the dataset, and obtain the sparser dataset with different density from 5 % to 25 %, ascending by 5 % each time. For example, dataset density 5 % means that we randomly leave 5 % of the dataset for training and the other value becomes testing set. The parameter settings of our NNCP method is that latent features dimensionality set as 20. The comparison result of this experiment are presented in Tables 2 and 3, and the detailed investigations of parameter settings will be provided in the following subsections.

From Tables 2 and 3, we can observe that our NNCP approach significantly improves the prediction accuracy, and obtains smaller MAE and RMSE value consistently for both response-time and throughput with different matrix densities. The MAE and RMSE value of throughput are much larger than those of response-time, because the range of throughput is 0-1000 kbps, while the range of response-time is only 0-20 s. With the increase of dataset density from 5 % to 25 %, the MAE and RMSE value of our NNCP method becomes smaller, because denser dataset provides more information for the missing QoS value prediction. Our NNCP method achieves

Table 2 Web service QoS performance comparison: MAE (mean average error)

WS QoS Property	Method	MAE				
		5 %	10 %	15 %	20 %	25 %
Response time	UMean	0.8156	0.7247	0.7161	0.6758	0.6361
	IMean	0.5708	0.4919	0.4988	0.4158	0.4083
	IPCC	0.6861	0.7972	0.5146	0.6014	0.4073
	UPCC	0.5965	0.6627	0.6625	0.6014	0.5435
	WSRec	0.5135	0.5252	0.5268	0.3947	0.3717
	RSVD	0.9162	0.8375	0.8168	0.8088	0.7800
	NNCP	0.4838	0.3589	0.3254	0.3178	0.3148
Throughput	UMean	8.3696	8.4262	8.0827	7.7713	7.7113
	IMean	6.7947	7.0433	6.4606	5.7356	5.2033
	IPCC	8.2521	8.6508	8.1413	8.8179	8.3416
	UPCC	8.0533	7.7259	7.1103	7.3437	7.0486
	WSRec	6.3139	6.2608	5.9656	5.9222	4.7879
	RSVD	9.6429	8.9885	7.5998	5.6261	5.1030
	NNCP	6.0007	5.4889	4.9859	4.5001	4.0385

Table 3 Web service QoS performance comparison: RMSE (Root mean square error)

WS QoS Property	**Method**	**RMSE**				
		5 %	10 %	15 %	20 %	25 %
Response time	UMean	2.3807	1.9589	1.9937	1.6229	1.4217
	IMean	2.3344	2.0264	2.4146	2.0878	1.7216
	IPCC	3.8511	3.8336	3.3770	2.5129	1.9188
	UPCC	2.3424	1.8843	1.9331	1.5129	1.2671
	WSRec	2.1838	2.0207	2.1533	1.7144	1.2975
	RSVD	6.6970	5.2284	3.8099	4.9581	3.6419
	NNCP	1.1470	1.0685	1.0502	1.0434	1.0399
Throughput	UMean	32.7424	35.3732	32.8413	44.4918	40.9749
	IMean	33.5447	34.5250	25.6687	22.7903	19.3721
	IPCC	41.4411	40.9693	37.4096	48.9877	42.6471
	UPCC	31.8687	32.9089	29.6238	29.2614	25.1004
	WSRec	23.0171	24.6223	22.4384	22.3709	17.9580
	RSVD	23.5928	25.4172	20.3695	19.7478	19.9420
	NNCP	10.8098	10.1738	9.57085	8.98722	8.43047

better performance than the baselines. But some factors of disharmony in Tables 2 and 3 are that the MAE and RMSE of baselines are not decreasing with the increase of dataset density in the strict sense. The fluctuation is cause by that prediction value of the baselines is only in one layer, and the value of testing set intersperse among 32 layers, which increase the uncertainty of prediction.

4.4 Impact of Dataset Sparseness

In this section, we investigate the impact of data sparseness on the prediction accuracy as shown in Fig. 8. We vary the density of the training matrix from 5 to 25 % with a step of 5 %. Figure 8a, b are the MAE and RMSE results of response-time. Figure 8c, d are the MAE and RMSE results of throughput. Figure 8 shows that: (1) With the increase of the training density, the performance of our method enhances indicating that better prediction is achieved with more QoS data. (2) Our NNCP method outperforms baselines consistently. The reason of this phenomenon is that baselines only utilize the two-dimensional static relations of *user-service* model without considering the

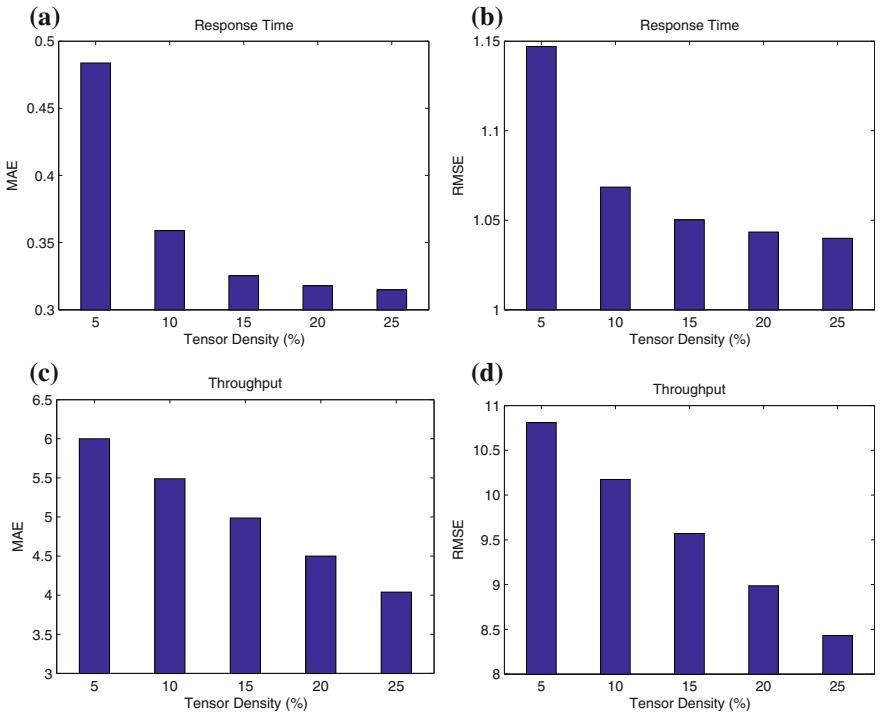


Fig. 8 Impact of tensor density (Dimensionality = 20)

more useful triadic relations of both the user and the service with the temporal information in the *user-service-time* model.

4.5 Impact of Dimensionality

The parameter dimensionality determines how many latent factors involve to tensor factorization. In this section, we investigate the impact of the dimensionality. We set the tensor density as 25 %, and vary the value of dimensionality from 1 to 20 with a step value of one.

Figure 9a, b show the MAE and RMSE results of response-time, and Fig. 9c, d show the MAE and RMSE results of throughput. Figure 9 shows that with the increase of latent factor number from 1 to 20, the value of MAE and RMSE keeps a declining trend. These observed results coincide with the intuition that relative larger number of latent factor produce smaller error ratio. But, more factors will require longer computation time and storage space. Moreover, when the dimensionality exceeds a certain threshold, it may cause the over-fitting problem, which will degrade the prediction accuracy.

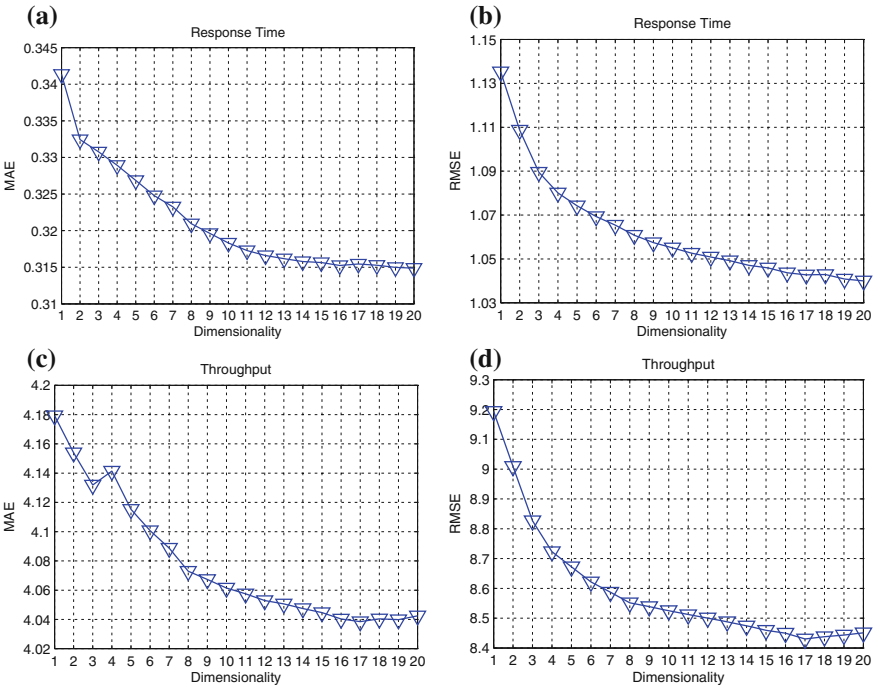


Fig. 9 Impact of factor matrices dimensionality (Tensor Density = 25 %)

5 Discussion

In this work, we design a tensor factorization based approach to recommending services to a user through leveraging crowdsourced testing data. The basic method is collaborative filtering, which can make prediction for a user by analyzing the similar users' behaviors. In service oriented computing, services are provided by various providers and it is impossible to obtain the attributes of a service by traditional program analysis or white-box testing. One practical and effective method to measure the QoS of a service should be black-box testing. The more user data is collected, the more probably we can find the users similar to the current user. In our scenario, "similar" means similar users can get similar QoS values when they invoke similar services. Crowdsourcing can help expand the scale of the potential users to participate in service testing.

iTest is such a crowdsourcing framework to support mobile users to help test services. Although many crowdsourcing applications rely on people's volunteering contributions, appropriate incentive mechanisms can attract people's participation and improve the quality of crowdsourcing tasks. Currently iTest totally depends on users' volunteering participation and we are still working on designing an effective incentive mechanism. The data we have obtained from testing with Planetlab nodes does simulate the variety of crowdsourced data. However, this method has the following limitations. First, crowdsourced data usually contains noise data while our simulated approach does not consider this. For example, even two users at the same location and time invoke the same service, however they may experience obviously different response time. A user's phone undergoing a high overload due to the concurrent running of several resource-demanding applications need more time to process the service invocation results. Therefore, filtering the abnormal crowdsourced testing data is of great importance to the recommendation algorithm. Second, our simulation approach does not simulate the variety of mobile clients in terms of browsers, operating systems, other software and hardware capabilities. In practice, client compatibility testing is a critical step to ensure the provided Web service can be run on various client environments. In light of this, detailed user contexts should be considered in computing the similarity of users.

6 Conclusions

Service discovery is one of the most important research topics in service oriented computing. In recent years, as the number of Web services has been continuously growing, recommending functionally equivalent Web services with different QoS properties is gaining momentum. Since the observed QoS data of users is dependent not only on services themselves, but also on invocation contexts like location and time, we need to collect as much as possible users' invocation data. To this end, we propose to leverage crowdsourcing to collect Web service QoS data. On the one hand,

we design a simulated crowdsourcing method by using Planetlab nodes to serve as workers. On the other hand, we design a mobile crowdsourcing framework iTest to test Web services with Android phones. In Web service recommendation, matrix factorization is one of the most popular approaches to CF. But the two-dimension model is not powerful to tackle the triadic relations of temporal QoS value. We extend the MF model to three dimensions through the use of tensor and employ the non-negative tensor factorization approach to advance the QoS-aware Web service recommendation performance in considering of temporal information.With a dataset obtained through our simulated crowdsourcing approach, we conduct an extensive set of experiments to evaluate our tensor factorization based methods. In the experimental results, a higher accuracy of QoS value prediction is obtained with the three-dimensional *user-service-time* model when comparing our method with other standard CF methods, which proves that crowdsourced testing can help improve recommendation results.

Future work can leads to two directions. First, our iTest framework is limited to a small scale of use due to lack of appropriate incentives. This is why we only use the dataset from simulated crowdsourcing appraoch. We will study an effective incentive mechanisms for mobile crowdsourced testing. Second, our recommendation approach only considers and models the relations between QoS value and the triplet $\langle user, service, time \rangle$. But in other cases, service users in different geographic locations at the same time may observe different QoS performance of the same Web service. Besides the temporal contextual information, more contextual information that influences the client-side QoS performance (e.g., the workload of the service servers, network conditions of the users, etc.) should be considered to improve the prediction accuracy. In our future work, we will continue to explore more user context information in the design of recommendation methods.

References

1. Bader B.W., Kolda T.G., et al. (2012) Matlab tensor toolbox version 2.5. http://www.sandia.gov/tgkolda/TensorToolbox/
2. Billsus, D., Pazzani, M.J.: Learning collaborative information filters. In:ICML, vol. 98, pp. 46–54 (1998)
3. Chen, X., Zheng, Z., Liu, X., Huang, Z., Sun, H.: Personalized Qos-aware web service recommendation and visualization. IEEE Trans. Serv. Comput. **6**(1), 35–47 (2013)
4. Doan, A., Ramakrishnan, R., Halevy, A.Y.: Crowdsourcing systems on the world-wide web. Commun. ACM **54**(4), 86–96 (2011)
5. Dong X., Halevy A.Y., Madhavan J., Nemes E., Zhang J.: Simlarity search for web services. In: VLDB, pp. 372–383 (2004)
6. He, Q., Yan, J., Yang, Y., Kowalczyk, R., Jin, H.: A decentralized service discovery approach on peer-to-peer networks. IEEE Trans. Serv. Comput, **6**(1), 64–75 (2013)
7. Li C., Zhang R., Huai J., Guo X., Sun H.: A probabilistic approach for web service discovery. In: IEEE SCC, pp. 49–56 (2013)
8. Paliwal, A.V., Shafiq, B., Vaidya, J., Xiong, H., Adam, N.R.: Semantics-based automated service discovery. IEEE Trans. Serv. Comput. **5**(2), 260–275 (2012)

9. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: state of the art and research challenges. IEEE Comput. **40**(11), 38–45 (2007)
10. Paterek, A.: Improving regularized singular value decomposition for collaborative filtering. In: Proceedings of KDD cup. workshop, vol. 2007 pp. 5–8 (2007)
11. Sarwar B., Karypis G., Konstan J., Riedl J.: Item-based collaborative filtering recommendation algorithms. In: Proceedings of the 10th International Conference on World Wide Web, pp. 285–295. ACM (2001)
12. Segev, A., Toch, E.: Context-based matching and ranking of web services for composition. IEEE Trans. Serv. Comput. **2**(3), 210–222 (2009)
13. Seung, D., Lee, L.: Algorithms for non-negative matrix factorization. Adv. Neural.Inf. Process. Syst. **13**, 556–562 (2001)
14. Shao L., Zhang J., Wei Y., Zhao J., Xie B., Mei H.: Personalized Qos prediction for web services via collaborative filtering. In: IEEE International Conference on IEEE Web Services, ICWS 2007, pp. 439–446 (2007)
15. Shashua A., Hazan T.: Non-negative tensor factorization with applications to statistics and computer vision. In: Proceedings of the 22nd International Conference on Machine learning, pp. 792–799.ACM (2005)
16. Sun H., Wang X., Yan M., Tang Y., Liu X.: Towards a scalable paaS for service oriented software. In: ICPADS, pp. 522–527 (2013)
17. W3C Web services activity. http://www.w3.org/2002/ws/ (2002)
18. Yan M., Sun H., Wang X., Liu X.: WS-TaaS: a testing as a service platform for web service load testing. In: ICPADS, pp. 456–463 (2012)
19. Zhang W., Sun H., Liu X., Guo X.: Temporal Qos-aware web service recommendation via non-negative tensor factorization. In: WWW, pp. 585–596 (2014)
20. Zheng, G., Bouguettaya, A.: Service mining on the web. IEEE Trans. Serv. Comput. **2**(1), 65–78 (2009)
21. Zheng Z., Ma H., Lyu M.R., King I.: WSRec: a collaborative filtering based web service recommender system. In: IEEE International Conference on IEEE Web Services ICWS 2009, pp. 437–444 (2009)
22. Zheng, Z., Ma, H., Lyu, M.R., King, I.: Qos-aware web service recommendation by collaborative filtering. IEEE Trans. Serv. Comput. **4**(2), 140–152 (2011)
23. Zheng Z., Ma H., Lyu M., King I.: Collaborative web service Qos prediction via neighborhood integrated matrix factorization (2012)

A Cloud-Based Infrastructure for Crowdsourcing Data from Mobile Devices

Nicolas Haderer, Fawaz Paraiso, Christophe Ribeiro, Philippe Merle, Romain Rouvoy and Lionel Seinturier

Abstract In the vast galaxy of crowdsourcing activities, crowd-sensing consists in using users' cellphones for collecting large sets of data. In this chapter, we present the *APISENSE* distributed crowd-sensing platform. In particular, *APISENSE* provides a participative environment to easily deploy sensing experiments in the wild. Beyond the scientific contributions of this platform, the technical originality of *APISENSE* lies in its Cloud orientation, which is built on top of the soCloud distributed multi-cloud platform, and the remote deployment of scripts within the mobile devices of the participants. We validate this solution by reporting on various crowd-sensing experiments we deployed using Android smartphones and comparing our solution to existing crowd-sensing platforms.

1 Introduction

The wisdom of the crowd is extensively used in many activities of our ever more connected world. Crowdsourcing denotes the practice of obtaining needed services, ideas, or contents by soliciting contributions from a large group of people, and

N. Haderer · F. Paraiso · C. Ribeiro · P. Merle · R. Rouvoy · L. Seinturier (✉)
University Lille 1 - Inria, Villeneuve d'Ascq, France
e-mail: Lionel.Seinturier@univ-lille1.fr

N. Haderer
e-mail: Nicolas.Haderer@inria.fr

F. Paraiso
e-mail: Fawaz.Paraiso@inria.fr

C. Ribeiro
e-mail: Christophe.Ribeiro@inria.fr

P. Merle
e-mail: Philippe.Merle@inria.fr

R. Rouvoy
e-mail: Romain.Rouvoy@inria.fr

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4_13

especially from an online community.[1] As users are more than ever going mobile, the research activities have also taken the path of mitigating crowdsourcing with mobile computing to give birth to the domain of crowd-sensing, which is the enlistment of large numbers of ordinary people in gathering data.[2] Using cellphones to collect user activity traces is also reported in the literature either as *participatory sensing* [5], which requires explicit user actions to share sensors' data, or as *opportunistic sensing* where the mobile sensing application collects and shares data without user involvement. These approaches have been largely used in multiple research studies including traffic and road monitoring [2], social networking [17] or environmental monitoring [18]. However, developing a sensing application to collect a specific dataset over a given population is not trivial. Indeed, a participatory and opportunistic sensing application needs to cope with a set of key challenges [7, 14], including energy limitation, privacy concern and needs to provide incentive mechanisms in order to attract participants.

In addition, crowd-sensing applications must also face the challenge of collecting large sets of data emitted by numerous clients potentially distributed all over the world. We believe that cloud computing infrastructures are solutions of choice for this. Indeed, cloud computing emerged as a way for "*enabling convenient, on-demand network access to a shared pool of configurable computing resources* (e.g., *networks, servers, storage, applications, and services*) *that can be rapidly provisioned and released with minimal management effort or service provider interaction*" [16].

In this chapter, we present the *APISENSE* environment for crowdsourcing data from mobile devices. Interestingly, the backend of *APISENSE* is built on top of the soCloud distributed multi-cloud platform to scale the gathering of crowd-contributed datasets. We therefore start by introducing soCloud in Sect. 2, and next, we present the architecture of *APISENSE* in Sect. 3. Section 4 reports on some experiments that have been conducted with *APISENSE*. Section 5 discusses some empirical validations. Section 6 compares our approach with existing solutions. Section 7 concludes this chapter.

2 soCloud Overview

In this section we present an overview of soCloud. After having discussed some background elements (Sect. 2.1), we present the main features of soCloud (Sect. 2.2), the way applications can be developed with soCloud (Sect. 2.3), and we report on the integration of soCloud with existing IaaS and PaaS cloud environments (Sect. 2.4).

[1] https://en.wikipedia.org/wiki/Crowdsourcing.

[2] http://researcher.watson.ibm.com/researcher/view_project.php?id=3011.

2.1 Service Component Architecture

soCloud is based on the Service Component Architecture (SCA) standard [19]. SCA is a set of OASIS specifications for building distributed applications and systems using Service-Oriented Architecture (SOA) principles [9]. SCA promotes a vision of Service-Oriented Computing (SOC) where services are independent of implementation languages (e.g., Java, Spring, BPEL, C++, COBOL, C), networked service access technologies (e.g., Web Services, JMS), interface definition languages (e.g., WSDL, Java) and non-functional properties. SCA thus provides a framework that can accommodate many different forms of SOC systems.

soCloud is implemented with FraSCAti [22] that is an open source framework for deploying and executing SCA-based applications. FraSCAti provides a component-based approach to support the heterogeneous composition of various interface definition languages (WSDL, Java), implementation technologies (Java, Spring, EJB, BPEL, OSGI, Jython, Jruby, Xquery, Groovy, Velocity, FScript, Beanshell), and binding technologies (Web Services, REST, HTTP, JSON-RPC, UPnP, JavaRMI, JMS, JGroups). Moreover, FraSCAti introduces reflective capabilities to the SCA programming model, and allows dynamic introspection and reconfiguration via a specialization of the Fractal component model [4]. These features open new perspectives for bringing agility to SOA and for the runtime management of SCA applications. FraSCAti is the execution environment of both the soCloud multi-cloud PaaS and soCloud SaaS applications deployed on the top of this multi-cloud PaaS.

2.2 Main Features of soCloud

soCloud is a service-oriented component-based PaaS for managing portability, elasticity, provisioning, and high availability across multiple clouds. soCloud is a distributed PaaS, that provides a model for building distributed SaaS applications based on an extension of the OASIS SCA standard.

Multi-cloud portability

The different layers of a cloud environment (IaaS, PaaS, SaaS) provide dedicated services. Although their granularity and complexity vary, we believe that a principled definition of these services is needed to promote the interoperability and federation between heterogeneous cloud environments [20]. Hence, our multi-cloud infrastructure uses SCA both for the definition of the services provided by the PaaS layer and for the services of SaaS applications that run on top of this PaaS. Therefore, soCloud uses SCA as an open service model to provide portability.

Multi-cloud provisioning

soCloud provides a consistent methodology based on the SCA standard that describes how SaaS applications are modelled. soCloud provides services based on FraSCAti to

deploy and allows runtime management of SaaS applications and hardware resources. This consistent methodology offers flexibility and choice for developers to provision and deliver SaaS applications and hardware resources across multiple clouds. soCloud provides a multi-cloud service enabling policy-based provisioning across multiple cloud providers.

Multi-cloud elasticity

The management of elasticity across multiple clouds is complex and appears to be approaching the limits of what is done with managing elasticity in a single cloud. In fact, systems become more interconnected and diverse, then latency and outages can occur at any time. The soCloud architecture focuses on the interactions among components, leaving such issues to be dealt with at runtime. Particularly, any automated set of actions aimed to modify the structure, behaviour, or performance of SaaS applications deployed with the soCloud platform while it continues operating. A remaining option is autonomic computing [12]. soCloud elasticity is managed with the MAPE-K (**M**onitor, **A**nalyse, **P**lan, **E**xecute, **K**nowledge) reference model [12] for autonomic control loop.

Multi-cloud high availability

soCloud provides high availability in two ways [21]. Firstly, by using a multi-cloud load balancer service to switch from one application to another when failure occurs. Secondly, the soCloud architecture uses a replication strategy. Especially, soCloud uses redundancy at all levels to ensure that no single component failure in a cloud provider impacts the overall system availability.

2.3 soCloud Applications

Application specification

soCloud applications are built by using the SCA model. As illustrated in Fig. 1, the basic SCA building blocks are software components [25], which provide services, require references and expose properties. The references and services are connected by wires. SCA specifies a hierarchical component model, which means that components can be implemented either by primitive language entities or by subcomponents. In the latter case the components are called composites. Any provided services or required references contained within a composite can be exposed by the composite itself by means of promotion links. To support service-oriented interactions via different communication protocols, SCA provides the notion of binding. For SCA references, a binding describes the access mechanism used to invoke a remote service. In the case of services, a binding describes the access mechanism that clients use to invoke the service.

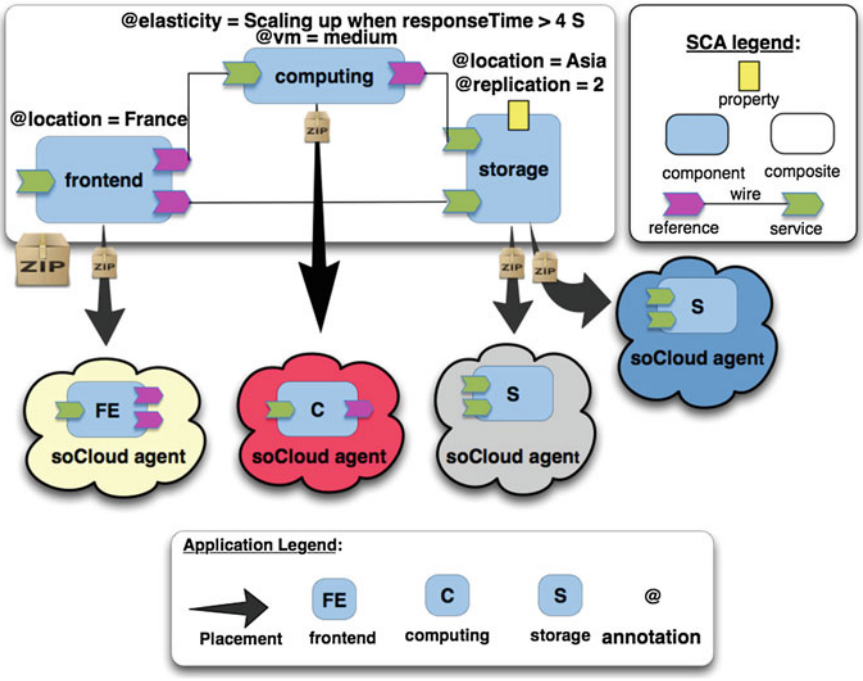


Fig. 1 An annotated soCloud application

We describe how SCA can be used to package SaaS applications. The first requirement is that the package must describe and contain all artifacts needed for the application. The second requirement is that provisioning constraints and elasticity rules must be described in the package. The SCA assembly model specification describes how SCA and non-SCA artifacts (such as code files) are packaged. The central unit of deployment in SCA is a contribution. A contribution is a package that contains implementations, interfaces and other artifacts necessary to run components. The SCA packaging format is based on ZIP files, however other packaging formats are explicitly allowed. As shown in Fig. 1, a three-tier application is packaged as a ZIP file (SCA contribution) and its architecture is described. This application is composed of three components. One component represents the frontend-tier of the application. The second one represents the computing-tier of the application. The last one represents the storage-tier of the application. Each component of this three-tier application is packaged as an SCA contribution.

Annotations

Some cloud-based applications require a more detailed description of their deployment (see Fig. 1). The deployment and monitoring of soCloud applications are bound by a description of the overall software system architecture and the requirements of the underlying components, that we refer to as the *application manifest*. In particular,

the *application manifest* should be platform-independent. Basically, the *application manifest* consists of describing what components the application is composed, with functional and non-functional requirements for deployment. In fact, the application can be composed of multiple components (see Fig. 1). The application manifest could define elasticity rules on each component (e.g., increase/decrease the number of instances of the component). Commonly, scale up or down, is translated to a **condition-action** statement that reasons on performance indicators of the deployed component. In order to fulfill the requirements for the soCloud application descriptor, we propose to annotate the SCA components with the four following annotations:

1. **placement constraint** (@*location*) maps the components of a soCloud application to available physical hosts within a geographical datacenter in a multi-cloud environment,
2. **computing constraint** (@*vm*) provides necessary computing resources defined for components of a soCloud application in a multi-cloud environment,
3. **replication** (@*replication*) specifies the number of instances of the component that must be deployed in a multi-cloud environment,
4. **elasticity rule** (@*elasticity*) defines a specific elasticity rule that should be applied to the component deployed in a multi-cloud environment.

For example, let us consider the three-tier web application described in Fig. 1, which consists of three components: Frontend, computing, and storage. The annotation @*location=France* on the frontend component indicates to deploy this component on a cloud provider located in France. Next, the annotations @*vm=medium* on the computing component specifies the computing resources required by this component and can be deployed on any cloud provider. The annotation @*elasticity=Scaling up when responseTime* $>$ 4 S defines an elasticity rule on the computing component, which scales up when the response time is superior to 4 s. Finally, the annotations @*location=Asia* and @*replication* $= 2$ on the storage component indicate to deploy this component on two different cloud providers located in Asia. soCloud automates the deployment of this three-tier application in a multiple cloud environment by respecting the given annotations.

2.4 Integration with Existing IaaS/PaaS

soCloud has been tested and can be deployed on ten target cloud environments that are publicly accessible on the Internet: Windows Azure,[3] DELL KACE,[4] Amazon EC2,[5]

[3] https://www.windowsazure.com.

[4] https://www.kace.com.

[5] http://aws.amazon.com/ec2.

Eucalyptus private clouds, CloudBees,[6] OpenShift,[7] dotCloud,[8] Jelastic,[9] Heroku,[10] and Appfog.[11] The first four cloud environments are IaaS platforms. In these cases, a PaaS stack, that is composed of a Linux or Windows distribution, a Java virtual machine (JVM), and FraSCAti, is deployed on top of these IaaS. The last six cloud environments are PaaS platforms composed a Linux distribution, a JVM, and a Web application container (e.g., Eclipse Jetty or Apache Tomcat). In these cases, soCloud including FraSCAti is deployed as a WAR file on top of them. Then *APISENSE* is deployed as a soCloud application, and is managed by soCloud autonomously.

3 The *APISENSE* Distributed Crowd-Sensing Environment

The main objective of *APISENSE* is to provide to scientists a crowd-sensing environment, which is open, easily extensible and configurable in order to be reused in various contexts. The *APISENSE* environment distinguishes between two roles. The first role, called *scientist*, can be a researcher who wants to define and deploy an experiment over a large population of mobile users. The environment therefore provides a set of services allowing her to describe experimental requirements in a scripting language, deploying experiment scripts over a subset of participants and connect other services to the environment in order to extract and reuse dataset collected in other contexts (e.g., visualization, analysis, replay). The second role is played by mobile phone users, identified as a *participant*. The *APISENSE* environment provides a mobile application allowing to download experiments, execute them in a dedicated sandbox and automatically upload the collected datasets on the *APISENSE* server.

Concerning the first role, *scientist*, we distinguish two tasks: (i) experiment design, storage, and deployment, and (ii) data gathering. The first task consists in creating the experiment script and making it available to *participants*. This task is performed by the so-called experiment store. The second task consists in gathering data that is sent by participants' mobile phones. This is performed by the data gathering nodes.

The remainder of this section presents the experiment store (Sect. 3.1), the client-side infrastructure that is used by *participants* (Sect. 3.2), and the data gathering nodes (Sect. 3.3).

[6] http://www.cloudbees.com.
[7] https://openshift.redhat.com.
[8] https://www.dotcloud.com.
[9] http://jelastic.com.
[10] http://www.heroku.com.
[11] http://www.appfog.com.

3.1 Experiment Store Infrastructure

Figure 2 provides an overview of the APISENSE experiment store infrastructure. This infrastructure is hosted on soCloud. The *Scientist Frontend* and *Participant Frontend* components are the endpoints for the two categories of users involved in the environment. Both components define all the services that can be remotely invoked by scientists and participants.

Crowd-sensing Library. To reduce the *learning curve*, we decided to adopt standard scripting languages in order to ease the description of experiments by scientists. We therefore propose the *APISENSE* crowd-sensing library as an extension of the JavaScript, CoffeeScript, and Python languages, which provides an efficient mean to describe an experiment without any specific knowledge of mobile device programming technologies (e.g., Android SDK). The choice of these host languages was mainly motivated by their native support for JSON (*JavaScript Object Notation*), which is a lightweight data-interchange format reducing the communication overhead.

The *APISENSE* crowd-sensing library adopts a reactive programming model based on the enlistment of handlers, which are triggered upon the occurrence of specific events (cf. Sect. 4). In addition to that, the API of *APISENSE* defines a set of sensing functions, which can be used to retrieve specific contextual data from sensors. The library supports a wide range of features to build dataset from built-in sensors proposed by smartphone technologies, such as GPS, compas, accelerometers,

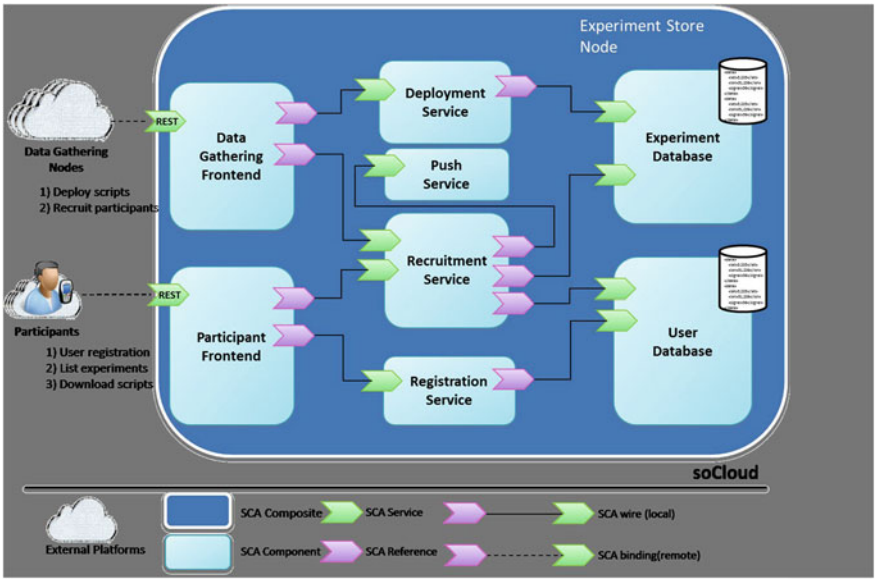


Fig. 2 Architecture of the *APISENSE* experiment store infrastructure

bluetooth, phone call, application status (installed, running) in the context of *opportunistic* crowd-sensing, but also to specify participatory sensing experiments (e.g., surveys).

Privacy Filters. In addition to a crowd-sensing script, the scientist can configure some privacy filters to limit the volume of collected data and enforce the privacy of the participants. In particular, *APISENSE* currently supports two types of filters. **Area filter** allows the scientist to specify a geographic area where the data requires to be collected. For example, this area can be the place where the scientist is interested in collecting a GSM signal (e.g., campus area). This filter guarantees the participants that no data is collected and sent to the *APISENSE* server outside of this area. **Period filter** allows the scientist to define a time period during which the experiment should be active and collect data. For example, this period can be specified as the working hours in order to automatically discard data collected during the night, while the participant is expected to be at home.

By combining these filters, the scientist preserves the privacy of participants, reduces the volume of collected data, and improves the energy efficiency of the mobile application (cf. Sect. 5).

Deployment Model. Once an experiment is defined with the crowd-sensing library, the scientist can publish it into the *Experiment Store* component in order to make it available to participants. Once published, two deployment strategies can be considered for deploying experiments. The former, called the pull-based approach, is a proactive deployment strategy where participants download the list of experiments from the remote server. The latter, known as the push-based approach, propagates the experiments list updates to the mobiles devices of participants. In the case of *APISENSE*, the push-based strategy would induce a communication and energy overhead and, in order to leave the choice to participants to select the experiments they are interested in, we adopted the pull-based approach as a deployment strategy. Therefore, when the mobile device of a participant connects to the *Experiment Store*, it sends its characteristics (including hardware, current location, sensor available and sensors that participants want to share) and receives the list of experiments that are compatible with the profile of the participant. The scientists can therefore configure the *Experiment Store* to limit the visibility of their experiments according the characteristics of participants. In order to reduce the privacy risk, the device characteristics sent by the participants are not stored by the infrastructure and scientists cannot access to this information.

Additionally, the *Experiment Store* component is also used to update the behavior of the experiment once deployed in the wild. When an opportunistic connection is established between the mobile device and the *APISENSE* server, the version of the experiment deployed in the mobile device is compared to the latest version published in the server. The installed crowd-sensing experiment is automatically updated with the latest version of the experiment without imposing participants to re-download manually the experiment. In order to avoid any versioning problem, each dataset uploaded automatically includes the version of the experiment used to build the

dataset. Thus, scientists can configure the *Experiment Store* component in order to keep or discard datasets collected by older versions of the experiment.

3.2 Client-Side Infrastructure

Figure 3 depicts the *APISENSE* mobile application's architecture. Building on the top of Android SDK, this architecture is mainly composed of four main parts allowing (*i*) to interpret experiment scripts (*Facades*, *Scripting engine*) (*ii*) to establish connection with the remote server infrastructure (*Network Manager*), (*iii*) to control the privacy parameters of the user (*Privacy Manager*), and (*iv*) to control power saving strategies (*Battery Manager*).

Scripting Engine. The *Scripting Engine* is based on the JSR 223 specification, and integrates the scripting languages that are supported by this specification. The *Sensor Facades* bridge the Android SDK with the *Scripting Engine*. This *Scripting Engine* covers three roles: A *security role* to prevent malicious calls of critical code for the mobile device, an *efficiency role* by including a cache mechanism to limit system calls and preserve the battery, and an *accessibility role* to leverage the development of crowd-sensing experiments, as illustrated in Sect. 4.

Battery Manager. Although the latest generation of smartphones provides very powerful computing capabilities, the major obstacle to enable continuous sensing applications is related to their energy restrictions. This component monitors the current battery level and suspends the *Scripting Engine* component when the battery level goes below a specific threshold (20 % by default) in order to stop all running experiments. This threshold can be configured by the participant to decide the critical level of battery she wants to preserve to keep using her smartphone.

Privacy Manager. In order to cope with the ethical issues related to crowd-sensing activities, the *APISENSE* mobile application allows *participants* to adjust their

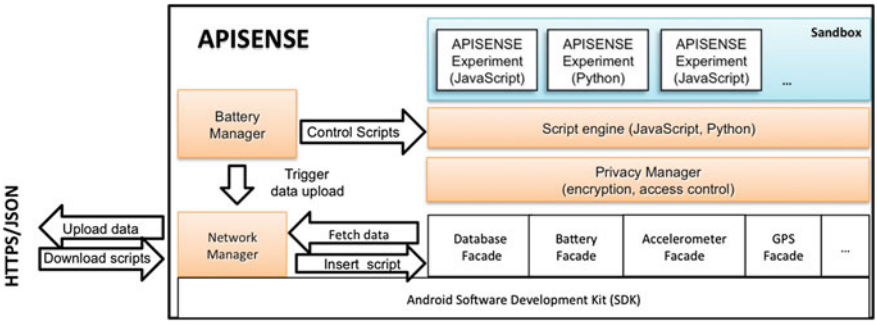


Fig. 3 Architecture of the *APISENSE* mobile application

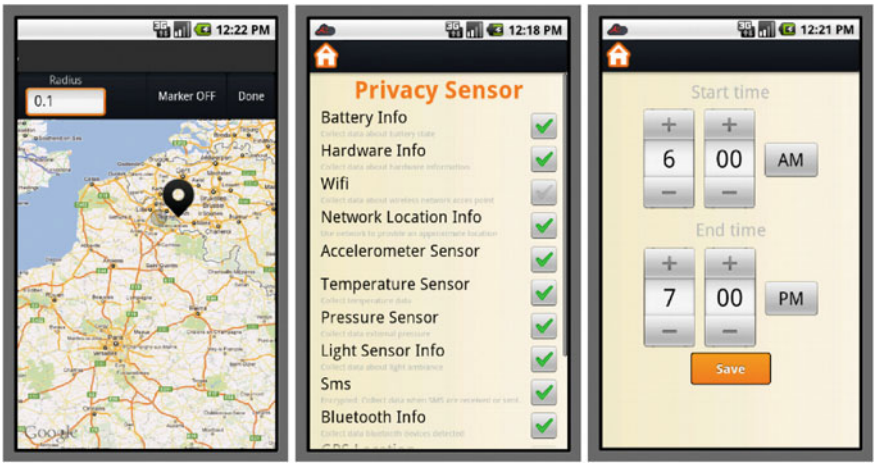


Fig. 4 Participant privacy preferences

privacy preferences in order to constrain the conditions under which the experiments can collect data. As depicted in Fig. 4, three categories of privacy rules are currently defined. Rules related to *location* and *time* specify geographical zone and time interval conditions under which experiments are authorized to collect data, respectively. All the privacy rules defined by the participant are interpreted by the *Privacy Manager* component, which suspends the *Scripting Engine* component when one of these rules is triggered. The last category of privacy rules refers to *authorization rules*, which prevent sensors activation or access to raw sensor data if the participant does not want to share this information. Additionally, a built-in component uses cryptography hashing to prevent experiments to collect sensitive raw data, such as phone numbers, SMS text, or address book.

The *APISENSE* mobile application is based on the Android operating system for the following reasons. First, the Android operating system is popular and largely adopted by the population, unit sales for Android OS smartphones were ranked first among all smartphone OS handsets sold worldwide in the third quarter of 2013 with a market share of 80 % according to Gartner.[12] Secondly, Android is an open environment supporting all the requirements for continuous sensing applications (e.g., multitasking, background processing and ability to develop an application with continuous access to all the sensors). Yet, as a matter of future work, we can think of porting the *APISENSE* mobile application on other systems, such as iOS 7 and Windows Phone.

A participant willing to be involved in one or more crowd-sensing experiments proposed by scientists can download the *APISENSE* mobile application by flashing the QR code published on the *APISENSE* website,[13] install it, and quickly

[12] http://www.gartner.com/newsroom/id/2623415.

[13] http://www.apisense.net.

create an account on the remote server infrastructure. Once registered, the HTTP communications between the mobile device of the participant and the remote server infrastructure are authenticated and encrypted in order to reduce potential privacy leaks. From there, the participant can connect to the *Experiment Store*, download and execute one or several crowd-sensing experiments proposed by scientists. The datasets that are collected by the experiments are sent to gathering nodes.

3.3 Data Gathering Nodes

When collected by participants' mobile phones, data are sent to so-called data gathering nodes. The data gathering nodes are the part of the server-side infrastructure where mobile phones upload the pieces of data that are sensed. This is the second task that is assigned to the *scientist* role. This task is decoupled from the first one that consists in deploying experiment scripts. The rationale is that the experiment store is an element which is rather central and well-known both by scientists to register experiments and by participants to retrieve experiments. Although as for other online stores, such as application stores, the experiment store can be replicated, we do not expect to have a lot of different copies of public experiment stores. Note that this may not be case, whenever organizations would want to set up private stores. However, the requirements are different when collecting data. It may be the case that, for confidentiality reasons, for legal reasons, or for questions of data size, each scientist may want to deploy her own gathering infrastructure to have some better guarantees on the data that is being collected.

Figure 5 presents the architecture of a data gathering node. Both scientists and participants can access the service interface of a node. The main entry point is for participants' smartphone to upload sensed data. Yet a service interface is also provided to scientists to enable managing the experiment and querying the dataset.

The uploading of data can be performed continuously, or delayed, for example to enable scenarios where the sensing is performed while the phone is offline. In addition, data can be sent as soon as they are produced, or in bulk mode, to optimize network connectivity and/or to save power. In fact, the policy for uploading data is fully under the control of the *scientist*. She can program in the experiment script any policy she needs to put into practice.

As for the experiment store, the data gathering nodes are hosted on the soCloud infrastructure. We benefit from the elasticity property of soCloud to scale up and down the resources that are assigned to each gathering node. This is especially important for experiments that generate large amounts of data.

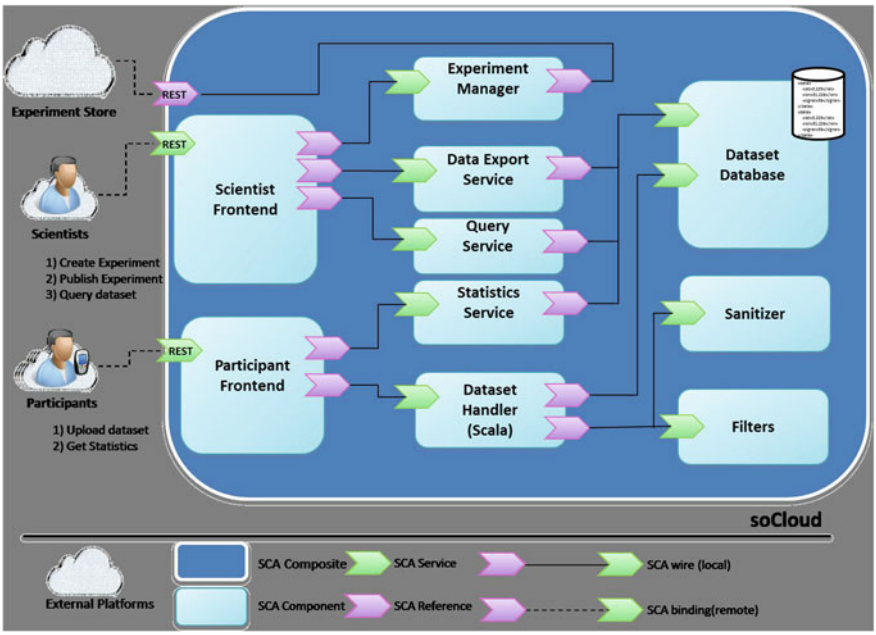


Fig. 5 Architecture of a *APISENSE* data gathering node

4 Crowd-Sensing Experiments

This section reports on four experiments that have been deployed in the wild using our platform. These examples demonstrate the variety of crowd-sensing experimentations that are covered by the *APISENSE* infrastructure.

4.1 Revealing Users' Identity from Mobility Traces

This first experiment aimed at identifying the potential privacy leaks related to the sporadic disclosure of user's locations. To support this experiment, we developed an *APISENSE* script, which reports every hour the location of a participant, known as Alice. Listing 1 describes the Python script we used to realize this experiment. This script subscribes to the periodic scheduler provided by the `time` facade in order to trigger the associated lambda function every hour. This function dumps a timestamped longitude/latitude position of Alice, which is automatically forwarded to the server.

Listing 1 Identifying GeoPrivacy Leaks (Python).

```
time.schedule({'period': '1h'},
  lambda t: trace.add({ 'time': t.timestamp,
                        'lon': gps.longitude(), 'lat': gps.latitude() }))
```

While this periodic report can be considered as anonymous since no participant's identifier is included in the dataset, this study has shown that the identity of Alice can be semi-automatically inferred from her mobility traces. To do so, we built a mobility model from the dataset we collected in order to identify clusters of Alice's locations as her *points of interest* (POI). By analyzing the size of the clusters and their relative timestamps, we can guess that the largest POI in the night relates to the house of Alice. Invoking a geocoding service with the center of this POI provides us a postal address, which can be used as an input to the yellow pages in order to retrieve a list of candidate names. In parallel, we can identify the places associated to the other POIs by using the Foursquare API, which provides a list of potential places where Alice is used to go. From there, we evaluate the results of search queries made on Google by combining candidate names and places, and we rank the names based on the number of pertinent results obtained for each name. This heuristic has demonstrated that the identity of a large population of participants can be easily revealed by sporadically monitoring their locations [13].

4.2 Building WiFi/GSM Signal Open Data Maps

This second experiment illustrates the benefits of using *APISENSE* to automatically build two open data maps from datasets collected in the wild. Listing 2 is a JavaScript script, which is triggered whenever the location of a participant changes by a distance of 10 m in a period of 5 min. When these conditions are met, the script builds a trace, which contains the location of the participant, and attaches WiFi and GSM networks characteristics.

Listing 2 Building an Open Data Map (JavaScript).

```
trace.setHeader('gsm_operator', gsm.operator());
location.onLocationChanged({ period: '5min',
  distance: '10m' }, function(loc) {
    return trace.add({
      time: loc.timestamp,
      lat: loc.latitude, lon: loc.longitude,
      wifi: { network_id: wifi.bssid(),
              signal_strength: wifi.rssi() },
      gsm: { cell_id: gsm.cellId(),
             signal_strength: gsm.dbm() } });
});
```

From the dataset, collected by three participants over one week, we build an QuadTree geospatial index to identify the minimum bounding rectangles that contain at least a given number of signal measures. These rectangles are then automatically colored based on the median signal value observed in this rectangle (cf. Fig. 6). This

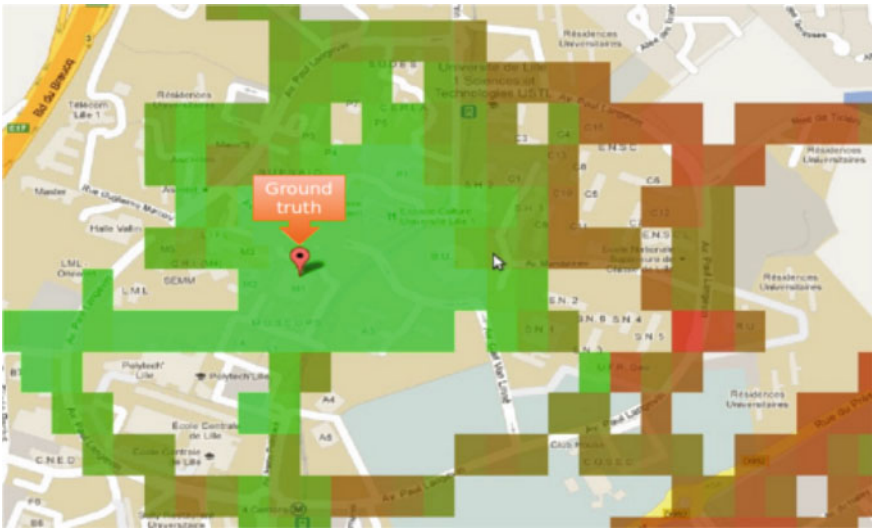


Fig. 6 GSM open data map

map has been assessed by comparing it with a ground truth map locating the GSM antennas and WiFi routers.[14]

4.3 Detecting Exceptions Raised by User Applications

The third experiment highlights that *APISENSE* does not impose to collect geolocated dataset and can also be used to build a realistic dataset focusing on the exceptions that are raised by the participants' applications. To build such a dataset, Listing 3 describes a CoffeeScript script that uses the Android logging system (`logCat`) and subscribes to error logs (`'*:E'`). Whenever, the reported log refers to an exception, the script builds a new trace that contains the details of the log and retrieves the name of the application reporting this exception.

Listing 3 Catching Mobile Applications' Exceptions (CoffeeScript).

```
logcat.onLog {filter: '*:E'},
  (log) -> if log.message contains 'Exception'
    trace.save
      message: log.message,
      time: log.timestamp,
      application: apps.process(log.pid).applicationName,
      topTask: apps.topTask().applicationName
```

Once deployed in the wild, the exceptions reported by the participants can be used to build a taxonomy of exceptions raised by mobile applications. Figure 7 depicts

[14] http://www.cartoradio.fr.

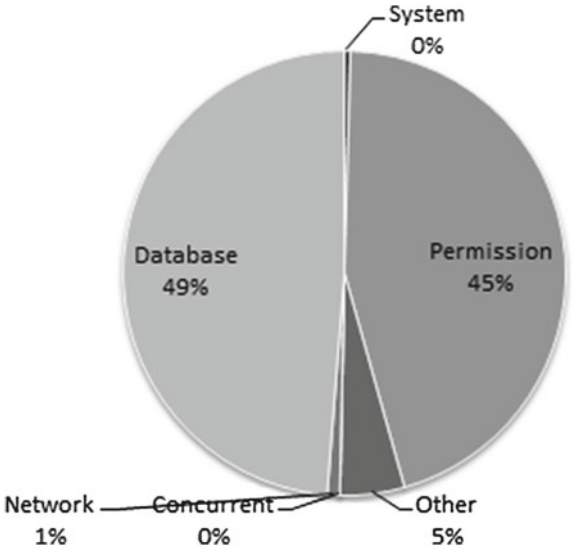


Fig. 7 Type of exceptions raised by mobile applications

the result of this experiment as a dataset collected by three participants over one month. In particular, one can observe that a large majority of errors reported by the participant's applications are related to permission or database accesses, which can usually be fixed by checking that the application is granted an access prior to any invocation of a sensor or the database. This experiment is a preliminary step in order to better identify bugs raised by applications once they are deployed in the wild as we believe that the diversity of mobile devices and operating conditions makes difficult the application of traditional in vitro testing techniques.

4.4 Experimenting Machine Learning Models

The fourth experiment does not only collect user-contributed datasets, but also deals with the empirical validation of models on a population of participants. In this scenario, the scientist wanted to assess the machine learning model she defined for detecting the activity of the users: *standing*, *walking*, *sitting*, *running*, or *jumping*. To assess this model, she deployed a script that integrates two phases: An exploration phase and an exploitation one. To set up this experiment, we extended the scripting library by integrating a popular machine learning tool [15] and adding a new facade to use its features from scripts. The script (cf. Listing 4) therefore starts with an exploration phase in order to learn a specific user model. During this phase, *APISENSE* generates some dialogs to interact with the participant and ask her to repeat some specific movements. The script automatically switches to the next movement when the model has recorded enough raw data from the accelerometer to provide an

accurate estimation. Once the model is considered as complete, the script dynamically replaces the timer handler to switch into the exploration phase. The dataset collected by the server-side infrastructure of *APISENSE* contains the model statistics observed for each participant contributing to the experiment.

Listing 4 Assessing Machine Learning Models (JavaScript).

```
var classes = ["walk","jog","stand", "sit", "up", "down"];
var current = 0; var buffer = new Array();
var model = weka.newModel(["avrX","avrY",...], classes);
var filter = "|(dx>"+delta+")(dy>"+delta+")(dz>"+delta+")";

var input = accelerometer.onChange(filter,
    function(acc) { buffer.push(acc) });

var learn = time.schedule({ period: '5s' }, function(t) {
  if (model.learn(classes[current]) >= threshold) {
    current++;
  }
  if (current < classes.length) { // Learning phase
    input.suspend();
    var output = dialog.display({ message: "Select movement", spinner: classes });
    model.record(attributes(buffer), output);
    sleep('2s');
    buffer = new Array();
    input.resume();
  } else { // Exploitation phase
    dialog.display({message: "Learning phase completed"});
    learn.cancel();
    model.setClassifier(weka.NAIVE_BAYES);
    time.schedule({ period: '5s' }, function(t) {
      trace.add({
        position: model.evaluate(attributes(buffer)),
           stats: model.statistics() });
      buffer = new Array();
} } });
```

Table 1 reports on the collected statistics of this experiment and shows that the prediction model developed by the scientist matches quite accurately the targeted classes.

Table 1 Representative confusion matrix

Predicted class							Acc (%)
	Walk	Jog	Stand	Sit	Up	Down	
Walk	**66**	0	4	0	0	0	94.3
Jog	0	**21**	0	0	0	0	100
Stand	4	0	**40**	0	0	0	90.9
Sit	0	0	2	**83**	0	0	97.6
Up stair	0	0	0	0	**22**	0	100
Down stair	0	0	0	0	0	**11**	100

5 Empirical Validations

Evaluating the Programming Model. In this section, we compare the *APISENSE* crowd-sensing library to two state-of-the-art approaches: AnonySense [24] and Pogo [3]. We use the *RogueFinder* case study, which has been introduced by AnonySense and recently evaluated by Pogo. *RogueFinder* is an application that detects and reports WiFi access points. Listings 5 and 6 therefore report on the implementation of this case study in AnonySense and Pogo, as decribed in the literature, while Listing 7 describes the implementation of this case study in *APISENSE*.

Listing 5 Implementing *RogueFinder* in AnonySense.

```
(Task 25043) (Expires 1196728453)
(Accept (= @carrier 'professor'))
(Report (location SSIDs) (Every 1 Minute)
(In location
  (Polygon (Point 1 1) (Point 2 2)
  (Point 3 0))))
```

Listing 6 Implementing *RogueFinder* in Pogo (JavaScript).

```
function start() {
  var polygon = [{x:1, y:1}, {x:2, y:2}, {x:3, y:0}];
  var subscription = subscribe('wifi-scan', function(msg){
    publish(msg, 'filtered-scans');
  }, { interval: 60 * 1000 });
  subscription.release();
  subscribe('location', function(msg) {
    if (locationInPolygon(msg, polygon))
      subscription.renew();
    else
      subscription.release();
  });
}
```

Listing 7 Implementing *RogueFinder* in *APISENSE* (CoffeeScript).

```
time.schedule { period: '1min' },
  (t) -> trace.add { location: wifi.bssid() }
```

One can observe that *APISENSE* provides a more concise notation to describe crowd-sensing experiments than the state-of-the-art approaches. This concision is partly due to the fact that *APISENSE* encourages the separation of concerns by externalizing the management of time and space filters in the configuration of the experiment. A direct impact of this property is that the execution of *APISENSE* scripts better preserves the battery of the mobile device compared to Pogo, as it does not keep triggering the script when the user leaves the assigned polygon. Nonetheless, this statement is only based on an observation of Pogo as the library is not made freely available to confirm this empirically.

Evaluating the Energy Consumption. In this section, we compare the energy consumption of *APISENSE* to a native Android application and another state-of-the-art crowd-sensing solution: FUNF [1]. FUNF provides an Android toolkit to build custom

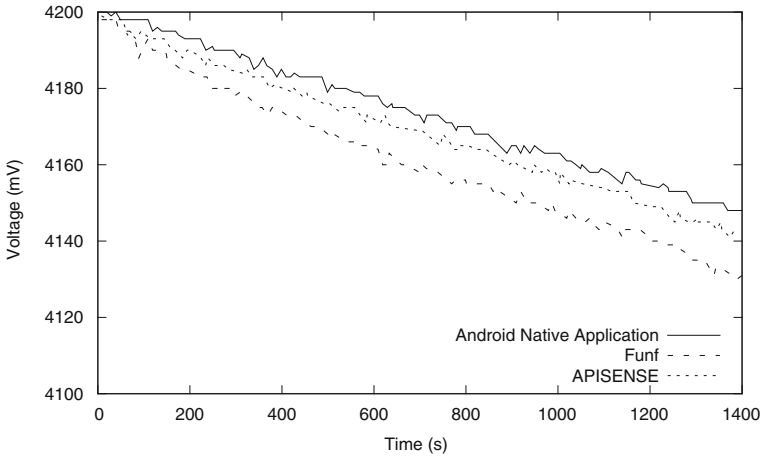


Fig. 8 Energy consumptions of Android, *APISENSE*, and FUNF

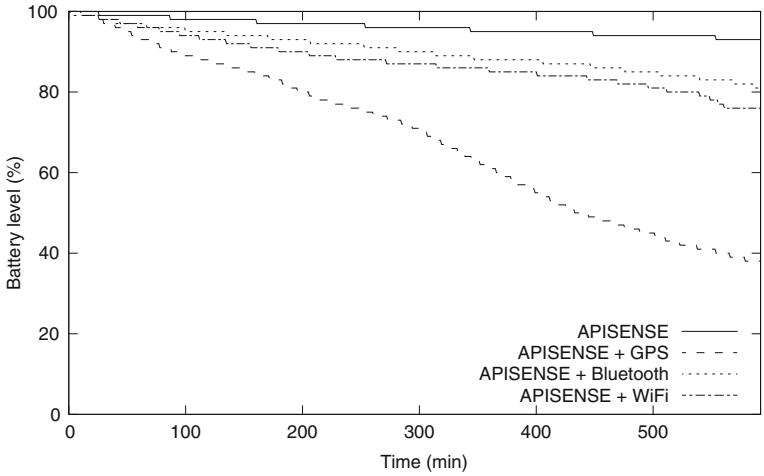


Fig. 9 Impact of *APISENSE* on the battery lifespan

crowd-sensing applications "à la carte". For each technology, we developed a sensing application that collects the battery level every 10 min. Figure 8 reports on this battery level, and consequently on the energy consumption of these applications. The measured voltage reflects the battery level.

Compared to the baseline, which corresponds to the native Android application, one can observe that the overhead induced by our solution is lower than the one imposed by the FUNF toolkit. This efficiency can be explained by the various optimizations included in our crowd-sensing library. Although more energyvorous than a native application, our solution does not require advanced skills of the Android

development framework and covers the deployment and reporting phases on behalf of the developer.

As the energy consumption strongly depends on (*i*) the nature of the experiment, (*ii*) the types of sensors accessed, and (*iii*) the volume of produced data, we conducted a second experiment in order to quantify the impact of sensors (see Fig. 9). For this experiment, we developed three scripts, which we deployed separately. The first script, labelled APISENSE + Bluetooth, triggers a Bluetooth scan every minute and collects both the battery level as well as the resulting Bluetooth scan. The second script, APISENSE + GPS, records every minute the current location collected from the GPS sensor, while the third script, APISENSE + WiFi, collects a WiFi scan every minute. These experiments demonstrate that, even when stressing sensors, it is still possible to collect data during a working day without charging the mobile phone (40 % of battery left after 10 h of pulling the GPS sensor).

6 Related Work

A limited number of data collection tools are freely available on the market. SYSTEMSENS [10], a system based on Android, focuses on collecting usage context (e.g., CPU, memory, network info, battery) of smartphones in order to better understand the battery consumption of installed applications. Similarly, LIVELABS [23] is a tool to measure wireless networks in the field with the principal objective to generate a complete network coverage map in order to help client to select network interface or network operators to identify blind spots in the network. However, all these tools are closed solutions, designed for collecting specific datasets and cannot be reused in unforeseen contexts in contrast to *APISENSE*. Furthermore, these projects are typical experiments deployed on mobile devices, without providing any privacy guarantee.

FUNF [1] is an Android toolkit focusing on the development of sensing applications. FUNF in a box is a service provided by FUNF to easily build a dedicated sensing application from a web interface, while data is periodically published via the Dropbox service. As demonstrated in Sect. 5, the current version of FUNF does not provide any support for saving energy nor preserving user privacy. Furthermore, the current solution does not support the dynamic re-deployment of experiments once deployed in the wild.

More interestingly, MYEXPERIENCE [11] is a system proposed for Windows mobile smartphones, tackling the *learning curve* issue by providing a lightweight configuration language based on XML in order to control the features of the application without writing C# code. MYEXPERIENCE collects data using a *participatory* approach—i.e., by interacting with users when a specific event occurs (e.g., asking to report on the quality of the conversation after a phone call ends). However, MYEXPERIENCE does not consider several critical issues, such as maintaining the privacy of participants or the strategic deployment of experiments. Even if an experiment can be modified in the wild, each experiment still requires a physical access to the

mobile device in order to be installed, thus making it difficult to be applied on a large population of participants.

In the literature, several deployment strategies of crowd-sensing applications have been studied. For example, AnonySense [24] uses—as *APISENSE*—a pull-based approach where mobile nodes periodically download all sensing experiments available on the server. A crowd-sensing experiment is written in a domain-specific language and defines when a mobile node should sense and under which conditions the report should be submitted to the server. However, AnonySense does not provide any mechanism to filter the mobile nodes able to download sensing experiments, thus introducing a communication overhead if the node does not match the experiment requirements.

On the contrary, PRISM [8] and Pogo [3] adopt a push-based approach to deploy sensing experiments over mobile nodes. PRISM is a mobile platform, running on Microsoft Windows Mobile 5.0, and supporting the execution of generic *binary code* in a secure way to develop real-time participatory sensing applications. To support real-time sensing, PRISM server needs to keep track of each mobile node and the report they periodically send (e.g., current location, battery left) before selecting the appropriate mobile phones to push application binaries. Pogo proposes a middleware for building crowd-sensing applications and using the XMPP protocol to disseminate the datasets. Nonetheless, Pogo does not implement any client-side optimizations to save the mobile device battery (e.g., area and period filters) as it systematically forwards the collected data to the server.

SensorSafe [6] is another participatory platform, which allows users to share data with privacy guaranties. As our platform, SensorSafe provides fine-grained temporal and location access control mechanisms to keep the control of data collected by sensors on mobile phones. However, participants have to define their privacy rules from a web interface while in *APISENSE* these rules are defined directly from the mobile phone.

7 Conclusion

While it has been generally acknowledged as a keystone for the mobile computing community, the development of crowd-sensing platforms remains a sensitive and critical task, which requires to take into account a variety of requirements covering both technical and ethical issues. To address these challenges, we report in this chapter on the design and the implementation of the *APISENSE* distributed platform. This platform distinguishes between two roles: *Scientists* requiring a sustainable environment to deploy sensing experiments and *participants* using their own mobile device to contribute to scientific experiments. On the server-side, *APISENSE* is built on the principles of Cloud computing and the soCloud distributed service-oriented component-based PaaS, which manages portability, provisioning, elasticity, and high availability of cloud applications across multiple clouds. Then *APISENSE* offers to scientists a modular service-oriented architecture, which can be customized upon

their requirements. On the client-side, the *APISENSE* platform provides a mobile application allowing to download experiments, executing them in a dedicated sandbox and uploading datasets to the *APISENSE* server. Based on the principle of *only collect what you need*, the *APISENSE* platform delivers an efficient yet flexible solution to ease the retrieval of realistic datasets.

Acknowledgments This work is partially funded by the ANR (French National Research Agency) ARPEGE SocEDA project and the EU FP7 PaaSage project.

References

1. Aharony, N., Pan, W., Ip, C., Khayal, I., Pentland, A.: Social fMRI: investigating and shaping social mechanisms in the real world. Pervasive Mob. Comput. **7**(6), 643–659 (2011)
2. Biagioni, J., Gerlich, T., Merrifield, T., Eriksson, J.: EasyTracker: automatic transit tracking, mapping, and arrival time prediction using smartphones. In: 9th International Conference on Embedded Networked Sensor Systems, pp. 68–81. ACM (2011). doi:10.1145/2070942.2070950
3. Brouwers, N., Woehrle, M., Stern, R., Kalech, M., Feldman, A., Provan, G., Malazi, H., Zamanifar, K., Khalili, A., Dulman, S., et al.: Pogo, a middleware for mobile phone sensing. In: 13th International Middleware Conference, pp. 106–113. Springer (2012)
4. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The fractal component model and its support in Java: experiences with auto-adaptive and reconfigurable systems. Softw.: Pract. Exp. (SPE) **36**(11–12), 1257–1284 (2006)
5. Burke, J., Estrin, D., Hansen, M., Parker, A., Ramanathan, N., Reddy, S., Srivastava, M.B.: Participatory sensing. In: Workshop on World-Sensor-Web (WSW'06): Mobile Device Centric Sensor Networks and Applications, pp. 117–134 (2006)
6. Choi, H., Chakraborty, S., Greenblatt, M., Charbiwala, Z., Srivastava, M.: SensorSafe: managing health-related sensory information with fine-grained privacy controls. Technical report, TR-UCLA-NESL-201009-01 (2010)
7. Cuff, D., Hansen, M., Kang, J.: Urban sensing: out of the woods. Commun. ACM **51**(3), 24–33 (2008)
8. Das, T., Mohan, P., Padmanabhan, V., Ramjee, R., Sharma, A.: Prism: platform for remote sensing using smartphones. In: 8th International Conference on Mobile Systems, Applications, and Services, pp. 63–76. ACM (2010)
9. Erl, T.: SOA: Principles of Service Design, vol. 1. Prentice Hall, Upper Saddle River (2008)
10. Falaki, H., Mahajan, R., Estrin, D.: SystemSens: a tool for monitoring usage in smartphone research deployments. In: 6th International Workshop on MobiArch, pp. 25–30. ACM (2011)
11. Froehlich, J., Chen, M., Consolvo, S., Harrison, B., Landay, J.: MyExperience: a system for in situ tracing and capturing of user feedback on mobile phones. In: 5th International Conference on Mobile Systems, Applications, and Services, pp. 57–70. ACM (2007)
12. Kephart, J.: An architectural blueprint for autonomic computing. IBM White paper (2006)
13. Killijian, M.O., Roy, M., Trédan, G.: Beyond Francisco cabs: building a *-lity mining dataset. In: Workshop on the Analysis of Mobile Phone Networks (2010)
14. Lane, N., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., Campbell, A.: A survey of mobile phone sensing. IEEE Commun. Mag. **48**(9), 140–150 (2010)
15. Liu, P., Chen, Y., Tang, W., Yue, Q.: Mobile WEKA as data mining tool on android. Adv. Electr. Eng. Autom. **139**, 75–80 (2012)
16. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology. http://www.nist.gov/itl/cloud/upload/cloud-def-v15.pdf (2009)

17. Miluzzo, E., Lane, N., Lu, H., Campbell, A.: Research in the App store era: experiences from the CenceMe App deployment on the iPhone. In: 1st International Work. Research in the Large: Using App Stores, Markets, and Other Wide Distribution Channels in UbiComp Research (2010)
18. Mun, M., Reddy, S., Shilton, K., Yau, N., Burke, J., Estrin, D., Hansen, M., Howard, E., West, R., Boda, P.: PEIR, the personal environmental impact report, as a platform for participatory sensing systems research. In: 7th International Conference on Mobile Systems, Applications, and Services, pp. 55–68. ACM (2009)
19. OASIS: Reference Model for Service Oriented Architecture 1.0. http://oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf (2006)
20. Paraiso, F., Haderer, N., Merle, P., Rouvoy, R., Seinturier, L.: A federated multi-cloud PaaS infrastructure. In: 5th IEEE International Conference on Cloud Computing, pp. 392–399. United States (2012). doi:10.1109/CLOUD.2012.79
21. Paraiso, F., Merle, P., Seinturier, L.: Managing elasticity across multiple cloud providers. In: 1st International Workshop on Multi-Cloud Applications and Federated Clouds. Prague, Czech, Republic (2013). http://hal.inria.fr/hal-00790455
22. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Softw.: Pract. Exp. (SPE) **42**(5), 559–583 (2012)
23. Shepard, C., Rahmati, A., Tossell, C., Zhong, L., Kortum, P.: LiveLab: measuring wireless networks and smartphone users in the field. ACM SIGMETRICS Perform. Eval. Rev. **38**(3), 15–20 (2011)
24. Shin, M., Cornelius, C., Peebles, D., Kapadia, A., Kotz, D., Triandopoulos, N.: AnonySense: a system for anonymous opportunistic sensing. Pervasive Mob. Comput. **7**(1), 16–30 (2011)
25. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York (1998)

Index

W. Li et al. (eds.), *Crowdsourcing*, Progress in IS,
DOI 10.1007/978-3-662-47011-4