LogicCrowd: Integrating Logic Programming and Crowdsourcing for Mobile and Pervasive Computing Platforms

Submitted by
Jurairat Phuttharak
B.Sc.(Comp), M.Sc.(IT)

A thesis submitted in total fulfilment
of the requirements for the degree
of Doctor of Philosophy

School of Engineering & Mathematic Sciences
College of Science, Health and Engineering

La Trobe University
Bundoora, Victoria 3086
Australia

October 2015

Statement of Authorship

Except where reference is made in the text of the thesis, this thesis contains no material published elsewhere or extracted in whole or in part from a thesis submitted for the award of any other degree or diploma.

No other person's work has been used without due acknowledgment in the main text of the thesis.

This thesis has not been submitted for the award of any degree or diploma in any other tertiary institution.

I declare that the research in this thesis is my own original work during my PhD candidature under the supervision of members of advisory panel, i.e. Associate Professor Dr Seng Wai Loke (main supervisor) and Associate Professor Dr Richard Lai (co-supervisor), except where otherwise acknowledged in the text.

Melbourne, 26/10/2015

Jurairat Phuttharak

Acknowledgements

I wish to express my sincerest gratitude to my supervisors, Associate Professor Dr Seng Wai Loke and Associate Professor Dr Richard Lai, for all their hard work and excellent support throughout my doctoral studies. Thanks to Dr Loke for his knowledge, enthusiasm and kindness and for helping me develop ideas and providing critical feedback. Thanks to Dr Lai for his continuous support and for taking care of me, in my personal life specifically when my mother passed away and went through my grieving period. Without their determined support, unwavering commitment, and steadfast encouragement, I would not have reached this stage.

Special thanks to La Trobe University and the College of Science, Health and Engineering for providing me with a strong foundation of skills, beneficial to my profession as a teacher and researcher. I am most grateful for the full funding I received from the Royal Thai Government and Prince of Songkla University to pursue this PhD research. Special thanks to my colleagues and good friends on the Trang campus who encouraged me in my study and shared my workload while I took this long leave. Without this support, I would never have been able to strengthen my professional development.

I would like to thank my colleagues at La Trobe University and my Thai friends; even though they did not directly help me with the thesis, the time we spent together helped me find balance in life. Thanks to the anonymous reviewers for giving feedback on my publications to help me become more confident in continuing the work.

Finally, my heartfelt thanks go to my father for his love and patience. My years in Australia meant I was unable to take good care of him during his suffering from heart disease. His true understanding has made it possible for me to undertake this work and complete it. Thanks to my sister for her advice and support by looking after my dad in Thailand. Without her support, this thesis would not have been a success.

Last but not least, my gratitude goes to my best friends Suthep and Jiraporn. Thanks to Suthep for listening, offering me advice and supporting me through this entire process of my study and my life. Thanks to Jiraporn for proofreading all my work, and for her inspiration, and love. Her encouragement was the biggest energy for me to continue working.

Abstract

Tasks requiring computations have been traditionally solved by machines alone. Programmers have to provide a formal problem description and an algorithm to the machine so that it can compute a solution. In order to increase the capabilities of machine-based computational systems, more sophisticated algorithms can be used. However, we cannot solely depend on machines for certain tasks and queries, especially those that need critical thinking, analytical skills, human judgment, and an understanding of physical world context. Processing such tasks using machines alone could result in poor answers. This might be partly due to inadequate information stored in the database or the complexity of the problem. In such cases, despite the use of sophisticated algorithms, the sole use of machines cannot guarantee the quality of task completion. To address this issue, human integration with machine processing has been argued to be an option. With multiple skills, particularly in creative thinking, visual processing, planning, and analysis, humans, when integrated with machine computation for solving complicated task-related problems, are expected to produce higher levels of accuracy and efficiency. The emergence of the crowdsourcing paradigm has brought a dramatic change in the landscape of solving complex problems. Crowdsourcing is an approach that involves human intelligence to solve complex problems that may be rather difficult for computers to solve alone.

At the same time, there has been an increasing need for easy-to-use programming languages. The declarative programming paradigm assumes programs to be descriptive in nature, focusing on the what rather than the how. Declarative programming languages and the declarative paradigm in general have received an increasing amount of widespread attention in the last decade. Due to the advantages of a high-level declarative manner of programming and the growing interest in crowdsourcing applications, the declarative approach to the development of crowdsourcing systems has tremendous potential.

In this thesis, the tasks are designed to be completed by the crowd through mobile technology. This design is based on the evidence that mobile devices have become the most accessible tool and allow immediate communication between people nearby and across the globe through Internet connections. Also, since smart mobile devices are enriched with a set of embedded sensors and context

data, they can support working-while-mobile, location-based services and peer-to-peer (P2P) communication. In this light, it may be argued that the use of mobile technology benefits crowdsourcing platforms.

The focus of this research is on the development of a framework to construct mobile crowdsourcing applications by integrating the logic programming paradigm with crowd-based human processing. *LogicCrowd* is introduced as an innovative approach which combines conventional logic-based machine computation and the power of the crowd. It is built on top of existing social networking infrastructure, online crowdsourcing market platforms and peer-to-peer networks.

The contribution of this thesis is two-fold. First, we propose a systematic approach to developing mobile crowdsourcing applications via the proposed *LogicCrowd* framework. By extending the capabilities of Prolog, this approach enables a logic program to involve human input and capabilities in problem-solving and query-answering, rather than working only with a closed set of facts and rules in a local database. The approach also provides a novel extended unification scheme within the logic-programming paradigm. By applying the declarative programming paradigm over peer-to-peer networks, a *LogicCrowd* program is able to evaluate queries over multiple hops in mobile ad-hoc networks.

Second, findings from a study involving extensive testing with a prototype system contribute insights into the use of mobile *LogicCrowd*-sourcing for real world applications. In this study, we demonstrate the versatility and usefulness of the system by introducing prototype applications based on our framework. The findings illustrate how the approach facilitates mobile users to leverage combinations of human intelligence and knowledge-based reasoning in order to solve complex problems. Furthermore, we analyse the energy characteristics of *LogicCrowd* programs to understand factors that influence energy consumption across resource-limited mobile crowdsourcing platforms. The results indicate how mobile users can reduce the energy consumption of the mobile device when dealing with *LogicCrowd* queries.

Contributions and Thesis Outcomes

Conference Papers

1. Phuttharak, J. and Loke, S. (2013). Logiccrowd: A declarative programming platform for mobile crowdsourcing. In *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM 2013*, pages 1323-1330, Melbourne, Victoria, Australia.

2. Phuttharak, J. and Loke, S. (2014a). Declarative programming for mobile crowd- sourcing: Energy considerations and applications. In Stojmenovic, I., Cheng, Z., and Guo, S., editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services, volume 131 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 237-249. Springer International Publishing.

3. Phuttharak, J. and Loke, S. W. (2014b). Towards declarative programming for mobile crowdsourcing: P2P aspects. In *Proceedings of the 2014 IEEE 15th International Conference on Mobile Data Management - Volume 02*, MDM '14, pages 61-66, Washington, DC, USA. IEEE Computer Society.

Presentations

1. Presented a paper *LogicCrowd: A Declarative Programming Platform for Mobile Crowdsourcing*, at the 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-13), Melbourne, Australia, Jul. 16-18, 2013.

2. Presented a paper *Declarative Programming for Mobile Crowdsourcing: a LogicCrowd Approach*, at the 2nd La Trobe Computer Science and Computer Engineering Research Workshop (LaTrobeCSCERW2013), Melbourne, Australia, Nov. 4, 2013.

3. Presented a paper *Declarative Programming for Mobile Crowdsourcing: Energy Considerations and Applications*, at the 10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous), Japan, Dec. 2-4, 2013.

4. Presented a paper *Towards Declarative Programming for Mobile Crowdsourcing: P2P Aspects*, at the 1st International Workshop on Mobile Collaborative Crowdsourcing and Sensing (M-CROS) in conjunction with the 15th IEEE International Conference on Mobile Data Management, Brisbane, Australia, Jul. 14-18, 2014.

Submitted Journal Papers

1. J. Phuttharak and S. W. Loke, "Mobile Crowdsourcing in Peer-to-Peer Opportunistic Networks: Energy Usage and Response Analysis," Journal of Network and Computer Applications, 2015. (Under review, the number of pages in submitted paper: 32 pages)

2. J. Phuttharak and S. W. Loke, "LogicCrowd: Combining Logic Programming and Crowdsourcing for Crowd-Powered Knowledge-Based Mobile Applications," Journal of Expert Systems With Applications, 2015. (Under 2^{nd} review, the number of pages in submitted paper: 43 pages)

Project Website

1. Publication papers:
   https://sites.google.com/site/jurairatbb/publications

2. LogicCrowd - Source code:
   https://sites.google.com/site/jurairatbb/codes

3. LogicCrowd - Experimental data:
   https://github.com/jurairat/LogicCrowd

4. LogicCrowd - NetLogo Simulation:
   https://github.com/jurairat/LogicCrowd-Netlogo

5. LogicCrowd - Metainterpreter:
   https://github.com/jurairat/metaLogicCrowd

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

We are now gradually moving towards the era of the Internet of Things. People and things can be connected anytime, anyplace, with anything and anyone, ideally using any path/network and any service [Perera et al., 2014]. With an increase in the number of connected people and devices/objects, a deluge of valuable data has been generated, requiring intelligence and more sophisticated mechanisms to create a broad range of applications to explore business opportunities. In recent times, crowdsourcing has emerged to facilitate data processing and problem solving. The ultimate goal of crowdsourcing is to utilize human processing power to solve problems that computers cannot (yet) solve. Due to the potential of Internet computing which allows seamless interaction between devices and people via technologies such as Wi-Fi, Bluetooth and RFID, the ability to process crowdsourcing data becomes more varied than before, especially in terms of opportunities to have immediate interaction with and thus contributions from a crowd. There have been many ways explored for crowds to help data processing. One possible way is based on the idea that data can be generated or collected directly by human beings, either by making queries or requests. The other way is to generate data through sensors or users' devices without intervention of users.

Crowdsourcing is increasing in popularity as a form of distributed problem solving enabled by digital technologies. The "crowd" is invited to contribute to tasks/projects, this contribution being in the form of knowledge or intelligence. Crowdsourcing is a flexible model that is applicable to a wide range of human activities, from the production of consumer goods and media content to matters of science and the environment. For example, Youtube.com and Wikipedia.org can be viewed as immensely popular applications of crowdsourcing to create social

information systems. In the case of earthquakes [Zook et al., 2010], for instance, individuals can obtain immediate access to various data available through the Cloud to gain relevant knowledge that helps them more accurately estimate the impact of ground movement and prepare strategies for emergency evacuation. Moreover, Threadless.com, one example of crowdsourcing, has been used in consumer goods and media. At Threadless, the members of the online community are allowed to cooperate with the company by creating, inspecting, improving and selecting products before any considerable investment is made.

Despite strong evidence of previous successes, such as Mechanical Turk[1], and CrowdFlower[2], the promising advantages of crowdsourcing are far from being fully realised along social, quality control, privacy, trust, and technical dimensions [Brabham, 2013; Zambonelli, 2011]. During the past decade, rather than being limited only to Web-based collaboration, computing systems are becoming more intimately embedded in physical and social contexts, promising truly ubiquitous computing. Crowds become more engaged through mobile devices in capturing, sharing and validating the sheer amount of data, resulting in more dynamic and prompt data. Such crowdsourced data using pervasive sensing devices are expected to boost the ability to identify and fulfill user and government needs, such as the social need for physical maintenance of the city (e.g., reporting broken lamps or a hole in the street). Clearly, people of a community can cooperate in contributing to these monitoring capabilities through useful data in the form of pictures and sensor data. For example, they can send a geo-tagged picture of a pothole from their smart phones. Or people can exploit the sensing capabilities of these devices by entering textual information on pictures, for example, to signal the fact that there is an oil spill on the road as this cannot be adequately captured by a camera.

However, program design with crowdsourcing software is profoundly different from traditional computer-based systems. There are many factors that distinguish crowdsourcing software from traditional software [Law and Ahn, 2011; Lease and Alonso, 2014; Quinn and Bederson, 2011]. First, it needs new programming metaphors and infrastructure that support the design, implementation and execution of crowdsourcing. There are also key issues concerning languages for expressing crowdsourcing behavior and computations. Second, since such systems largely depend on humans, the process might take longer than before in order to find responsive crowds and collect the completed tasks from them. Therefore,

[1]https://www.mturk.com/mturk/welcome
[2]http://www.crowdflower.com/

strategies to recruit and motivate crowds (to participate in generating data) are needed for the development of crowdsourcing systems. Finally, the process of accumulating crowd feedback needs verification. Developing effective algorithms that can leverage decision-making, average the guess of a group of people and avoid systemic bias in answering, is a substantial challenge. Hence, in designing the algorithms, developers have to maintain a balance in the management of speed, money, and reliability.

1.1 Thesis Aims and Problem Statement

Before discussing the key ideas of the current thesis, we introduce a sample of questions and tasks which can give insight into the issues that the thesis aims to address. Some of the questions are:

"Which Thai restaurants nearby would you recommend for Pad Thai?"
"What is the meaning of 'buzzing' in this sentence?"
"Which photo has the most beautiful scenery?"
"Is this art decoration better than the other?"
"What did they say in this audio clip?"
"Where is the cat in this photo?"
"Please help revise this."
"Is this animal cute?"
"Is this video a funny one?"
"Which nearby cafés have a short queue?"
"Which is the best way to get a taxi here?"
"Choose only the photos that have flowers."
"Find the name of the 'CEO' of this company."
"Where can I buy a pen for less than $3 here?"
"Where can I most likely find available car park?"
"What is the most scenic way to get from Johor to Selangor?"
"How do I tell her that I am looking for a tool to cut finger nails (saying this in Japanese)?"
"Will this crowd please get me footage of this game from different perspectives in this stadium?"

To ask questions and perform tasks similar to the ones mentioned above, most researchers exploit the potential of artificial intelligence (AI) in relation to image

processing, voice recognition, data mining techniques, natural language understanding and so on, which means sophisticated algorithms are required. Clearly, solving these types of tasks using a machine only might be difficult to achieve. Instead, asking humans might be a better solution. More specifically, in addition to the exploitation of machines, humans with multiple and creative skills in visual processing, planning, and task analysis are expected to be able to do complicated tasks with higher levels of accuracy and efficiency.

Moreover, it seems that some queries cannot be simply answered with accurate results by database systems. This might happen when information stored in the database is insufficient. For example, we attempt to find the names of CEOs in a company's database using the query `"SELECT name FROM employee WHERE position = CEO"`. The result will be `"null"` or `"no answer"` if the `employee` table instance in the database at that time does not contain a record for "CEO" or contain a record for "Chief Executive Officer," rather than using its shortened form (CEO). This problem might be due to the use of words that differ from the ones recorded in the database during the data entry.

Another example is when the query is not in the knowledge base or is out of scope of the database. Consider the following facts in Prolog which defines the parent relationship:

```
parent(charles,james).
parent(sophia,elizabeth).
parent(elizabeth,james).
```

In this case, the facts describe parent relationships. The `"parent"` relationship holds in relation to family members; for example, `charles` is a parent of `james` and `sophia` is a parent of `elizabeth`. Now consider the two queries below:

```
?-parent(elizabeth,james).
?-parent(elizabeth,george).
```

For the first query `(elizabeth,james)` the result from Prolog's evaluation is "true" when the relationship indicating that `elizabeth` is the parent of `james` is already stored in the knowledge base. For the second query `(elizabeth,george)`, the result from the evaluation will be `"false"` when there is no data on the relationship of these two people is found in the knowledge base. This problem emerges since the search ability is restricted to the local knowledge base only. In the light of these examples, it can be argued that the degree of the flexibility in the use of resources available for searching is indicative of the success in finding results.

This thesis identifies a gap in the current research. That is, there is a need for

further research on exploiting the potential of humans to solve complex problems through the use of crowdsourcing concepts but there are two key issues (i) how can complex crowdsourcing behaviors be expressed, represented and coded up? (ii) how can crowdsourcing behaviors be integrated with computational behavior programmatically? This thesis will explore the high-level abstraction of logic-based declarative programming as a solution towards these issues. We propose and investigate *LogicCrowd*, a concept combining logic programming and crowdsourcing towards a solution. The *LogicCrowd* framework (based on the popular logic programming language Prolog) has been built to provide a proof-of-concept implementation. Moreover, we consider issues in enabling *LogicCrowd* for mobile users, the majority of computer users in the future.

The problematic issues exemplified above are addressed through the following research questions (RQs):

RQ1. How can the declarative programming paradigm be combined with the crowdsourcing concept to solve complex problems? This RQ will be more focused on how logic programming is able to express queries to the crowd.

RQ2. What are the extensions to the Prolog meta-interpreter to implement *LogicCrowd*? In what ways can traditional unification in Prolog be extended with crowdsourcing behaviors, and how can *LogicCrowd* programs be extended for peer-to-peer mobile networks?

RQ3. What types of architectural approaches can be used to build the *LogicCrowd* framework? How can mobile computing notions in the framework enhance the system's capability and broaden the range of applications?

RQ4. In what way can *LogicCrowd* programs be written to best benefit mobile users? How do we address issues in using *LogicCrowd* for mobile users?

RQ5. What are the energy characteristics of *LogicCrowd* programs when running on energy-limited mobile devices?

1.2 Thesis Scope

The focus of this research is on the development of a framework for constructing mobile crowdsourcing applications by applying the logic programming paradigm. Taking a complementary view to the imperative programming paradigm [Little

et al., 2010b] which allows programmers to specify the steps required in order to solve a particular task, *LogicCrowd* introduces the concept of declarative programming to crowdsourcing. The use of logic programming is aimed at providing expressive power, declarative semantics, a higher level of abstraction, and allowing query and manipulation of knowledge and reasoning.

In addition, *LogicCrowd* has been designed by combining conventional logic-based machine computation and the power of the crowd obtained from (1) social networks, (2) existing crowdsourcing market platforms and (3) mobile P2P networks, in order to solve problems that machines might find difficult. Moreover, *LogicCrowd* can operate on a mobile platform, enabling mobile crowdsourcing behaviors. It can also be useful in P2P computing for querying and multicasting tasks shared over peer networks.

In this research, the logic programming language we exploit is Prolog. Prolog is used due to its relatively simple semantics and popularity. However, other logic programming languages can also be employed. With *LogicCrowd* applications being configurable and reprogrammable, mobile users or developers are able to fix/decide on facts and rules. Therefore, *LogicCrowd* programmers are assumed to have a basic background on Prolog, though syntactic sugar and user interface (UI) forms can be used for non-technical users.

1.3 Our Approach

As mentioned, our view of declarative programming aims to incorporate crowdsourcing behaviors in the mobile environment using *LogicCrowd* programs. *LogicCrowd* makes use of the features of Prolog, which provides a mechanism for recursively extracting the sets of data values implicit in the facts and rules of a program. Furthermore, *LogicCrowd* exploits the ability of crowdsourcing to solve problems that are complicated and thus cannot be simply solved by a machine alone. We also investigated a decentralized crowdsourcing approach in our *LogicCrowd* framework; each peer/node is able to process and distribute its information without relying on any centralized authority through mechanisms that are based on its own connections to other peers.

1.3.1 Declarative Programming

The declarative programming paradigm assumes programs to be descriptive in nature. The declarative paradigm allows the programmer to focus on what is to be solved, which usually depends on the context of the language [Lloyd, 1994]. A declarative programming language is a realization of the declarative paradigm. It refers to a program that is a theory written in some suitable logical and structural form, allowing a reader of the program to obtain a precise meaning for it.

Prolog [Sterling and Shapiro, 1994] is a logic programming language, which is well-known for its expressive power, ease of use, and its facilitating of rapid development. *LogicCrowd* builds on the strengths of Prolog. In addition, Prolog is well-suited for writing a meta-interpreter, (which is an interpreter for the language written in the language itself). Prolog programs can be naturally represented as Prolog terms and are easily inspected and manipulated using built-in mechanisms.

1.3.2 Mobile Crowdsourcing

Fuchs-Kittowski and Faust [2014] define "mobile crowdsourcing" as a situation when "a group of people voluntarily collects and shares data using widely available mobile devices." They further explain that this data is processed and provided via a data-sharing infrastructure to third parties who are interested in integrating this data. Typically, a mobile crowdsourcing system consists of a platform residing on the cloud and mobile smartphone. Based on Chatzimilioudis et al.'s study [2012], there are two architectures in mobile crowdsourcing: (1) centralized and (2) decentralized. In the centralized method, the data which are generated and collected from the crowd are transferred to a server where the answer can be used, whereas all computations and communications are performed locally on smartphones in the decentralized approach. In light of the flexibility that mobile crowdsourcing offers and the opportunity to make use of sensors available in mobile devices, we develop the prototype on a mobile platform.

1.3.3 Decentralized Computing

Mobile environments are distributed and heterogeneous. That is, they allow devices and systems to be integrated and embedded together with computing and communication systems through wireless transmission. Moreover, computing has

mobilized itself beyond the desktop PC. By having *LogicCrowd* in mobile environments, we find that decentralized computing can enhance our framework. Decentralized computing is where all computations and communications are performed by peers in an appropriate manner. This approach is known as peer-to-peer computing, in which each node/peer of the system is responsible for contributing to the global community. This type of computing can be located at different places with the geographic location being often relevant to the computational process itself.

1.3.4 Research Methodology

We use the experimental research method in this work. More specifically, we develop a prototype to evaluate our framework. In this development, both real and synthetic data are used to evaluate the performance of our approach. We also evaluate the performance of our system by using empirical testing in a lab and simulations in order to construct basic scalability experiments. We build a prototype system to illustrate the feasibility of the concept. We also build various applications that include Crowd Voting and Recommendation, Crowd-Powered Comparison, and Crowd Parking and Boundary Finding, all of which can be applied in many situations. These will show the development efficiencies, versatility, and robustness of our system for building mobile crowdsourcing applications.

1.4 Thesis Contributions

The thesis investigates a declarative crowdsourcing platform for mobile applications using the *LogicCrowd* framework. *LogicCrowd* utilizes a declarative style of programming for crowdsourcing, especially in the mobile environment. It is built on top of existing social networking infrastructure, online crowdsourcing market platforms and P2P networks as well as data in logic programs. The framework allows mobile users to create rules via an event-driven approach.

The key distinguishing feature for the *LogicCrowd* framework is to add crowdsourcing behavior to the logic programming paradigm in order to leverage human knowledge to solve problems, which might be difficult for a computer alone. More specifically, the key contributions of this thesis are:

- We propose a systematic approach for developing mobile crowdsourcing applications via the introduction of the *LogicCrowd* framework.

- We propose the *LogicCrowd* language that extends the capabilities of Prolog in three dimensions. First, it allows predicates to query the crowd, rather than a closed set of facts. Second, specific operators encoded by extending the basic Prolog meta-interpreter are designed for managing interactions in a concise and succinct manner with the crowd. Third, it provides two methods, synchronous and asynchronous calls for the flexible crowd execution of queries. Building on these extensions, *LogicCrowd* also has an extended unification scheme in the logic programming paradigm, essentially enabling queries to access the crowd in an on-demand fashion for the purposes of unifying arguments in predicates when traditional unification might not work.
- We investigate and showed how to apply mobile peer crowdsourcing networks. This extension enables crowd query evaluations to take place over multiple hops in mobile P2P networks.
- We demonstrate the system's versatility and usefulness via prototype applications based on our framework. The evaluation results, including the pilot study and deployment, show that our system is feasible and can work effectively in the real world.
- We examine how *LogicCrowd* programs influence power consumption in different conditions in order to understand the impact on battery life when implementing and running crowdsourcing applications using *LogicCrowd*.

1.5 Thesis Outline

The thesis is organized as follows.

Chapter 2 reviews background knowledge on the crowdsourcing paradigm based on a survey of significant projects. We outline a range of frameworks, techniques, key aspects and challenges for developing crowdsourcing applications. This section also defines a taxonomy based on the key issues in programming with crowdsourcing.

Chapter 3 introduces a declarative programming paradigm integrated with crowdsourcing for mobile environments. We describe the concept and the relevant key ideas of logic programming and the *LogicCrowd* concept. We detail the *LogicCrowd* formalism, comprising the notion of crowdsourcing behaviors and simple extensions of Prolog in *LogicCrowd*. A novel unification approach, called crowd unification, which is embedded in an extension of Prolog, is introduced in

this chapter. Furthermore, we outline how we apply the declarative programming paradigm to mobile peer crowdsourcing networks.

Chapter 4 details the *LogicCrowd* framework, which is built based on the *LogicCrowd* concepts. The *LogicCrowd* framework integrates logic programming with crowdsourcing platforms to provide a declarative programming platform for mobile applications. The key components of the *LogicCrowd* framework, including Prolog engine, Mediator, Knowledge Base and Crowdsourcing APIs, are designed to combine conventional machine computation and the power of the crowd.

Chapter 5 describes the applications that have been built using the *LogicCrowd* framework. We illustrate scenarios using these applications that can be applied for the purposes of increasing convenience for users. These scenarios are derived and expanded from the motivating scenario in Section 3.1. Based on these implementations, the notion of *LogicCrowd* is validated and demonstrated.

Chapter 6 presents the performance evaluation of the *LogicCrowd* prototype. This chapter examines how running a *LogicCrowd* program influences power consumption on a mobile device under different conditions, in order to find the most efficient way to implement *LogicCrowd* crowdsourcing applications. We investigate the energy characteristics of our *LogicCrowd* prototype on actual mobile devices with two extensions of Prolog: (1) basic crowd predicates and (2) crowd unification.

Chapter 7 explores empirical testing in a lab and simulations to study the distribution of crowd tasks for mobile crowdsourcing in an opportunistic network and to describe the basic scalability experiments that were conducted. We create a mobile crowdsourcing simulation model based on an understanding of the basic operations of the *LogicCrowd* prototype as implemented on real devices. The purpose of the simulation is to analyze the relationship between the parameters of mobile crowdsourcing and the number of response messages from the crowd, and to study the energy consumption characteristics of mobile crowdsourcing, especially in a large-scale ad-hoc network.

Chapter 8 summarizes findings and key contributions of this thesis. We also concluded by suggesting directions for future research.

Chapter 2

Background and Related Work

In this chapter, we present a review of a range of frameworks, techniques, key aspects, and challenges for developing crowdsourcing applications. The chapter starts with a brief overview of the crowdsourcing paradigm, based on a survey of significant projects. Then, Section 2.1 gives an overview of the crowdsourcing concept and related topics. This section also defines a taxonomy based on the key issues in programming with crowdsourcing. Section 2.2 details the most common tools and approaches to support programmers in implementing complex crowdsourcing applications. Section 2.3 discusses the characteristics of crowdsourcing workflows and algorithms. Section 2.4 reviews applications of crowdsourcing indifferent aspects. Section 2.5 discusses existing frameworks and architectures of crowdsourcing systems. A proposed general crowdsourcing architecture is also presented in this section. Section 2.6 reviews the mobility dimension that has become a key aspect of today's crowdsourcing paradigm. Section 2.7 discusses the current gaps and our plan for addressing our research questions detailed in Chapter 1. Finally, Section 2.8 summarizes the chapter by highlighting the gaps in the literature which we address in this thesis.

2.1 An Overview of Crowdsourcing

In this section, we focus on existing definitions and ideas of crowdsourcing in order to clarify its meaning, and on the concepts underlying crowdsourcing. In addition to this, crowdsourcing is compared with other similar concepts. Although many studies have focused on increasing the performance of machine-based computational systems and using sophisticated algorithms and computing architectures to solve complex problems, there are a large number of tasks that currently cannot be

accurately and efficiently performed by machines [Quinn and Bederson, 2011]. Examples of such tasks are related to the areas of sentiment analysis, natural language understanding, image recognition and creativity. These kinds of tasks are better suited to humans who are innately good at creativity, visual processing, planning, and analysis tasks. Hence, humans can perform them easily with high accuracy and efficiency. Recently, there has emerged a new computing approach that employs human abilities to perform the tasks combined with machine computation. This new mode of human involvement with machine is called *"crowdsourcing"*.

The term crowdsourcing was originally coined by Jeff Howe, a contributing editor at Wired magazine [Howe, 2006]. He defined it as the "act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call".

From his definition, three key prerequisites that strengthen the benefits of crowdsourcing are emphasized. First, the crowdsourcing task must be completed by a large group. Second, the ability for a requester to access a large group of people working towards a solution is required. Third, the requester needs to engage these people in a crowdsourcing task via an open call. This engagement is mainly achieved by the use of social web technologies [Hetmank, 2014]. From the emergence of crowdsourcing concepts, there are several successful crowdsourcing platforms in the market, such as Amazon's Mechanical Turk (MTurk) being probably the most well known, which have further fostered business and development. These platforms mostly appear on the Web where they allow requesters to post tasks in order to hire workers to perform these tasks for a small reward. Consequently, a number of crowdsourcing systems have emerged to harness human intelligence to tackle complex problems.

There have been various other explanations proposed in the related literature to characterize crowdsourcing. For instance, Estellés-Arolas and González-Ladrón-de-Guevara [2012] analyzed the existing definitions of crowdsourcing and created an integrated definition that considers several aspects related to the crowd, the initiator, and the underlying process. They claim that crowdsourcing is a type of participative online activity in which an individual, an institution, a non-profit organization, or company proposes to a group of individuals of varying knowledge, heterogeneity, and number, via a flexible open call, the voluntary undertaking of a task. The undertaking of the task, of variable complexity and modularity, and in which the crowd should participate bringing their work, money, knowledge and/or experience, always entails mutual benefit. The user will receive the satisfaction

of a given type of need, be it economic, social recognition, self-esteem, or the development of individual skills, while the crowdsourcer will obtain and utilize to their advantage that which the user has brought to the venture, whose form will depend on the type of activity undertaken.

Even though this definition seems to be complicated, it is able to reduce misunderstandings and gives some clarification. In software engineering contexts, crowdsourcing is typically used to refer to a system [Doan et al., 2011] as follows: "a crowdsourcing system enlists a crowd of humans to help solve a problem defined by the system owners". The idea behind crowdsourcing which is widely discussed and accepted by researchers in the literature is that the crowd plays a significant role in the concept of crowdsourcing and thus, crowdsourcing is conceptualized based on the underpinnings related to the crowd. It should be noted that the term "crowdsourcing" has been ambiguously used alongside such concepts as the wisdom of the crowd, collective intelligence, human computation, open innovation and social computing due to their similar meanings (which will be discussed later). Quinn and Bederson [Quinn and Bederson, 2011] studied the human computation paradigm. They review its definition and presented a classification system for human computation systems that highlights the distinctions and similarities among various projects. Figure 2.1 shows the intersection of related topics with human computation.

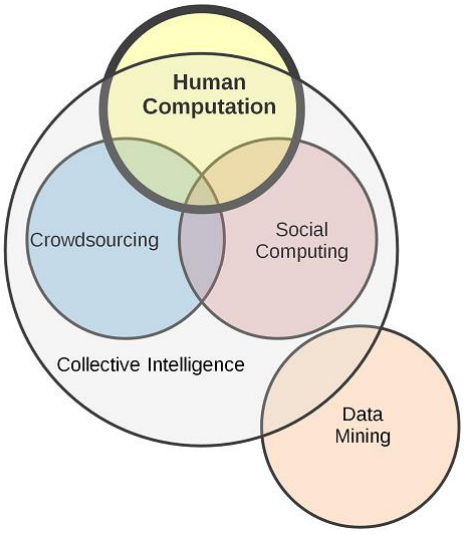


Figure 2.1: The intersection between relevant topics and human computation (adapted from [Quinn and Bederson, 2011])

In some studies [Brabham, 2008], the notion of crowdsourcing has been explained through the notion of crowd wisdom. The wisdom of the crowd was termed by Surowiecki [2005] who claims that aggregating solutions of a group of people

is often more accurate than those of individuals or any particular members of a group. Surowiecki also states that web technology is a suitable tool for aggregating millions of disparate, independent ideas in the way of markets and intelligent voting systems, reporting that there are four key conditions for crowd wisdom: cognitive diversity, independence, decentralization and aggregation. Under these conditions, heterogeneous individual ideas in the crowd are not averaged, but aggregated into final solutions. The principles of statistics predict this will be true as long as the guesses are all made independently, and the group does not have a systemic bias in relation to the answer.

Although the "wisdom of crowds" contributes to explaining many aspects of crowdsourcing, other aspects should be considered to reveal the potential of crowdsourcing. In his recent study, Brabham [2013] broadens the discussion on crowdsourcing by adding the idea of collective intelligence which focuses on an ability to coordinate between individuals that allows crowdsourcing to find solutions in a coordinated way. Collective intelligence is not a new phenomenon. It has received increasing interest from both academic and practical areas for several years. In recent years, Malone et al. [2011] define the term "collective intelligence" as "groups of individuals doing things collectively that seem intelligent." This concept is different from the wisdom of crowds which is based on the independence of individuals whereas the idea of collective intelligence focuses on the collaboration between individuals. There is an additional distinction between collective intelligence and crowdsourcing in terms of applications. Crowdsourcing is the idea that can be applied to both intelligence and simple tasks and these tasks are performed either in a competitive or collaborative way by each of the individuals. In contrast, collective intelligence applies only when the process is performed by a group of participants, rather than individuals [Little, 2011].

One important term that is used broadly and is related to the crowdsourcing concept is human computation. There is some overlap between human computation and crowdsourcing. In [Quinn and Bederson, 2011], the authors give a comprehensive overview of human computation in general and a classification of its different aspects. They argue that the modern usage of the term human computation was inspired by von Ahn's dissertation [Von Ahn, 2005]. Von Ahn defines human computation as a paradigm for utilizing human processing power to solve problems that computers cannot yet solve. By taking into consideration the definitions proposed in a series of other papers [Quinn and Bederson, 2011], the

characteristics of human computation can be concluded as follows: 1) the problems fit a general paradigm of computation and sometimes the problems can be solved by computers, and 2) the human participates directly in the computation system. Therefore, the highlight of human computation and crowdsourcing is the vital role of humans performing tasks. Noticeably, one major difference between human computation and crowdsourcing is that human computation replaces computers with humans whereas crowdsourcing replaces traditional in-house human workers with members of the public [Quinn and Bederson, 2011].

Open innovation is often discussed among the topics relating to crowdsourcing. This concept is derived from the business strategy and innovation literature. It refers to the conscious attempt by firms to incorporate ideas initiated from outside the firms into innovation processes within the firms, or to send internally initiated ideas outside of the firms for commercial purposes [Seltzer and Mahmoudi, 2013]. Marjanovicet et al. [2012] proposed that both crowdsourcing and open innovation should be classified into the same paradigm, where organizations harvest knowledge and expertise from the outside, as opposite to closed innovation. There are two important aspects that distinguish between crowdsourcing and open innovation. First, open innovation only addresses innovation processes, whereas the crowdsourcing concept covers various types of tasks. Second, the firms participate with their customers or other companies in the open innovation paradigm, while the organization depends on the public or members of the crowd in crowdsourcing activities.

Another term situated at the intersection of crowdsourcing and human computation is social computing. It concerns both harnessing human intelligence for computational tasks and the design of computational systems that support social behavior and interactions. Therefore, social computing refers to the applications of various social media tools including blogs, wikis, social bookmarking, peer-to-peer networks, open source communities, photo and video sharing communities, and online business networks. It can be further defined as "applications and services that facilitate collective action and social interaction online" in which a rich amount of multimedia information and aggregate knowledge is exchanged and developed [Parameswaran and Whinston, 2007]. Quinn and Bederson [2011] point out the key distinction between human computation and social computing, that is, social computing accommodates relatively natural human behavior that is mediated by technology, whereas participation in human computation is under the primary control of the human computation system.

The post-PC era sees a shift in the computing platform from desktop to mobile computing. Along with the unique multi-sensing capabilities of modern mobile devices, the smartphone can eventually unfold the potential of crowdsourcing. Smartphones offer a great platform for extending either web-based or distributed crowdsourcing applications to a larger contributing crowd and makes contributing easier and omnipresent. Moreover, smartphone users are able to provide large amounts of opportunistic or participatory data that can contribute to complex and novel problem solving. Fuchs-Kittowski and Faust [2014] define "mobile crowdsourcing" as when "a group of people voluntarily collects and shares data using widely available mobile devices where this data is processed and provided via a data sharing infrastructure to third parties interested in integrating and remixing this data." According to the key characteristics of mobile crowdsourcing as mentioned in [Chatzimilioudis et al., 2012], there are two aspects of the crowd's contributions including participatory and opportunistic aspects. Participatory crowdsourcing refers to a data collection approach performed by opt-in users while the input for opportunistic crowdsourcing is the data generated from sensors and computations performed by the crowd's devices automatically. Thus, a variety of possible mobile crowdsourcing applications have emerged in different fields. Based on an extensive literature review in [Fuchs-Kittowski and Faust, 2014], a categorization of existing applications of mobile crowdsourcing systems has been described. These applications can be categorized as either people-centric or environment-centric. However, the specific mobile crowdsourcing-based applications differ with regard to a number of various dimensions. Based on differences at the system level, these applications are grouped according to criteria such as type of sensor, data capturing method, admission criteria of a participant, and degree of participation.

As mentioned, crowd computing is a powerful approach to addressing problems that are hard for computers but trivial for humans to solve. Crowdsourcing platforms in the market such as MTurk, microWorkers, and CrowdFlower allow developers to programmatically post tasks that are completed by human workers who receive rewards or payment in return. Recently, several innovative crowdsourcing applications have emerged by integrating human computing into systems. For instance, the VizWiz[1] and Be My Eyes[2] apps help blind people who wish to take a picture send a question to volunteer helpers from around the world and then receive an answer back from the crowd in seconds. However, implementing

[1]http://www.vizwiz.org/
[2]http://www.bemyeyes.org/

crowdsourcing programs is still challenging. First, program design with human computation is profoundly different from traditional computer-based systems. It needs powerful new programming metaphors and infrastructures that support the design, implementation, and execution of human computing. Second, since the systems rely on humans, the process might take longer than before in order to find responding workers and collect the completed tasks. Therefore, recruitment and motivation techniques under a limited budget are an essential component of developing crowdsourcing systems. Third, the process of accumulating the crowd's feedback needs to involve verification. Developing effective algorithms to leverage decision making, average the guess of a group of people, even to avoid a systemic bias about the answer, is a substantial challenge. Hence, developers have to manage tradeoffs between speed, money, and reliability in designing their algorithms.

In view of these challenges, we have explored the existing crowdsourcing literature in terms of programming-based systems which integrate the ability of programs to delegate work or computation to humans. We identified five main dimensions in crowd programming, and these inform our taxonomy. Our criteria for defining the taxonomy are based on key issues in programming with crowdsourcing, and how they have been tackled in existing work. We also expand each dimension to highlight the unique set of challenges in terms of programming/implementation needs in order to be considered during the development and evaluation of such collaborative systems. These key aspects are summarized in Figure 2.2, and discussed in the following sections.

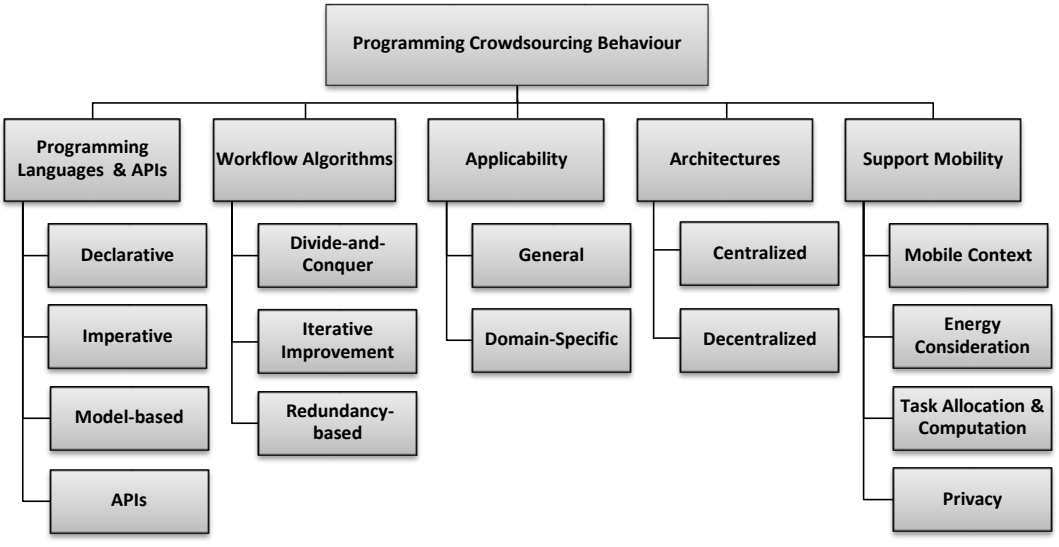


Figure 2.2: A Taxonomy of Issues in Programming CrowdSourcing Behaviour

2.2 Programming Languages and APIs

In recent years, several tools to support programmers to implement complex crowdsourcing applications with micro-tasks have been proposed. They provide a library of functions, or are a simple extension of a query language, that can be called within programs written in a variety of programming languages. After exploring existing work, we discuss the set of programming tools dealing with the crowdsourcing concept from four viewpoints: 1) declarative programming languages, 2) imperative programming languages, 3) model-based languages, and 4) APIs.

2.2.1 Declarative Programming Languages

The declarative programming paradigm assumes programs to be descriptive in nature. Most classical programming paradigms put emphasis on *how* a problem should be solved. For example, the imperative, object-oriented and functional models all let the programmer specify the steps required in order to solve a particular task. In contrast, the declarative paradigm lets the programmer focus on *what* is to be solved which usually depends on the context of the language [Lloyd, 1994]. Consequently, a declarative programming language is the basic property of the declarative paradigm. It refers to a program that is a theory written in some suitable logical and structural form leading a reader of the program to immediately get a precise meaning for it. Beginning at a high level of abstraction, the programmer can concentrate on stating what is to be computed, not necessarily how it is to be computed [Mottola, 2005].

One of the most famous and widely used programming languages with a declarative nature is Prolog [Sterling and Shapiro, 1994], also known as a logic programming language. Consider the problem of the factorial mathematical function shown below.

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{else} \end{cases}$$

This problem can be written in Prolog using the declarative paradigm as displayed in listing 2.1. When writing in a declarative language, the programmer needs to concentrate only on the logic (what), rather than the control (how). In particular, a Prolog program consists of facts and rules which serve to define relations on sets of values. The imperative component of Prolog is its execution engine based on unification and resolution, mechanisms for recursively extracting sets of data values implicit in the facts and rules of a program. We will briefly introduce

the basic Prolog in this section.

```
factorial(0,1).
factorial(N,F):-
        N > 0,
        N1 is N-1,
        factorial(N1,F1),
        F is N*F1.
```

Listing 2.1: Prolog implementation of the factorial function (declarative)

A fact consists of an identifier followed by an n-tuple of constants. A relation identifier is referred to as a predicate. When a tuple of values is in a relation we say the tuple satisfies the predicate. Thus, a fact states that a certain tuple of values satisfies a predicate unconditionally. On the other hand, a rule defines the relation in a more general sense. It gives conditions under which tuples satisfy a predicate.

As a simple example, consider the following Prolog program which defines the grandparent relationship.

```
parent(john,jack).
parent(john,jane).
parent(jane,jeff).
grandparent(A,C):-parent(A,B),parent(B,C).
```

As shown in the example above, the program consists of three facts and one rule. The facts describe the relation parents: `john` is a parent of `jack` and `jane`, whereas `jane` is a parent of `jeff`.

The last line of this program constitutes a rule which defines a new predicate `grandparent`. It consists of a head and a body which are separated by "`:-`" read as "if". The left side of the rule (head) looks like a fact, except that the parameters of the fact are capitalized indicating they are variables rather than constants. The right side of the rule (body) consists of two terms separated by a comma which is read as "AND". The rule describes the relation of grandparents that is interpreted as `A` is a grandparent of `C` if `A` is a parent of `B` and `B` is a parent of `C`.

Now, consider the following goal:

```
?-grandparent(john,jeff).
```

Typically, such a query/goal can be typed into a Prolog command shell similar to a UNIX command shell after the Prolog prompt "`?-`". The Prolog interpreter reads a program and then allows the user to compute a query based on the definitions (facts and rules). A query "asks" the system if there are any data values which satisfy a particular predicate. The above goal is asking: is `john` a grandparent of `jeff`? By searching through the facts and rules, this goal will be evaluated

locally to be "true".

One of the key processes in the execution of Prolog programs is a matching operation called "unification". When Prolog is attempting to satisfy a goal, clauses are selected if their head unifies with the goal. Prolog's pattern matching is based on the concept of a substitution. For example, Prolog matches `woman(X)` with `woman(jane)`, thereby instantiating the variable `X` to `jane`. Therefore, two terms match, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal.

To understand deeper properties of logic programs, we can look at the logical and operational meaning of programs. Operational semantics are a category of formal programming language semantics in which certain desired properties of programs and the constructs of programming languages are verified by constructing proofs from logical statements about their execution and procedures, rather than by attaching mathematical meaning to their terms [Plotkin, 1981]. We describe an operational semantic for the language of pure Prolog in the style of structural operational semantics [Plotkin, 1981]. First, we define the relation $P \vdash_{\theta} G$ to hold when the goal G succeeded with Prolog program P with the result θ (the substitution). The rules for pure Prolog are in the format $[Label]\frac{premise}{conclusion}$ where the conclusion holds whenever the premises hold, as follows.

$$[true] \quad \frac{}{P \vdash_{\epsilon} true}$$

$$[atom] \quad \frac{P \vdash_{\theta} H{:-}G \wedge \theta{=}mgu(A,H) \wedge \ P \vdash_{\gamma} G\theta}{P \vdash_{\theta\gamma} A}$$

$$[conjunction] \quad \frac{P \vdash_{\theta} G_1 \wedge \ P \vdash_{\gamma} G_2\theta}{P \vdash_{\theta\gamma} G_1{,}G_2}$$

The rule $[true]$ contains no premise called an axiom. It means this rule is always true independent of the premise and ϵ denotes the empty substitution. For the rule $[atom]$, it refers to the evaluation of a goal in the standard Prolog in the form $H :- G_1, G_2, ..., G_n$ where G_i is a subgoal. The key idea of the evaluation of the goal in Prolog is based on the concept of a substitution. The θ is the result of standard substitution in Prolog. We can compute the most general unifier (mgu) denoted by $\theta = mgu(term1, term2)$.

The rule $[conjunction]$ represents the operation of the conjunctive several goals. In Prolog, we can conjoin several goal and pose the conjunction as a query. It also allows us to write the rule's bodies that contain conjunctions. A conjoined query/rule $G_1, G_2, ..., G_n$, posed with respect to a program P, is a logical consequence of P if and only if the conjunction of the G_i is a logical consequence of P.

In joining goals to form a collection of goals the common "," denoted the logical conjunction "AND".

Declarative programming languages and the declarative paradigm in general have been attracting more widespread attention in the last decade. There is an increasing need for easy-to-use programming languages [Lloyd, 1984]. SQL[3] and XML[4] are examples of declarative languages addressing specific needs which are very common nowadays. SQL is a well-established language and is the de-facto standard for database management and querying, while XML usage is expanding though often more for marketing reasons than for real practical needs, and is being used in various areas related to data management.

Recently, crowdsourcing services have been widely used for complex human-reliant data processing tasks, such as identifying items on images or extracting opinions, emotions and sentiments in text [Doan et al., 2011]. These systems mostly provide a programming layer by utilizing the development process of writing task-based systems and optimizing their performance building on top of crowdsourcing platform markets such as MTurk, CrowdFlower and so on. Due to the advantage of a high-level declarative manner of programming and the growing interest in crowdsourcing applications, the declarative approach to the development of crowdsourcing systems has emerged. Existing work using the declarative abstraction is discussed in two groups: SQL-like languages and rule-based languages.

2.2.1.1 SQL-like Languages

SQL-like refers to the fact that the syntax of the query language is designed to be similar to SQL syntax but it is used in specific domains. This SQL-like declarative approach to the crowdsourcing paradigm has received much interest in database research. Several studies have extended classical query processing languages. For instance, Franklin et al. [2011] offer an SQL interface to pose complex queries in crowdsourcing called CrowdDB. It exploits the iterator-based query processing paradigm to integrate crowd functionality into a DBMS. CrowdDB extends a traditional query engine with crowdsourced operators that take input from humans, and can generate and submit task requests to a micro-task crowdsourcing platform. Marcus et al. [2011] present Qurk, an adaptive query system which allows crowd-powered processing for relational databases using an SQL-based query language. In Qurk, lightweight user-defined functions are defined by users/developers

[3]http://en.wikipedia.org/wiki/SQL
[4]http://www.w3schools.com/xml/

in order to describe their workflow at a high level and to automatically optimize workflow and tuning parameters. Parameswaran et al. [2012] present Deco, a database system for declarative crowdsourcing. They extend the internal schema with functional dependencies including fetch and resolution rules for crowdsourcing tuples and resolving conflicts. Simperl et al. [2015] propose work contributing extensions of the SPARQL query language and the Linked Data service technology with crowdsourcing functionality.

2.2.1.2 Rule-based Languages

Rule-based languages relate to several areas in databases and logic-based AI. They incorporate practical human knowledge called facts and rules for example, in the form of if-then rules [Parameswaran and Whinston, 2007] . In crowdsourcing systems, the high-level control flow of algorithms using logic programming has been explored in many studies. Recently, Parameswaran and Polyzotis [2011] describe the design of a declarative language involving human-computable functions, standard relational operators, as well as algorithmic computation using a Datalog-like formalism. Their work addresses the challenges involved in optimizing queries posed in their model, especially the trade-offs between uncertainty, time allocated and monetary costs.

Later, Morishima et al. [2012] present an executable rule-based language for data-centric human computations and crowdsourced processing named Cylog/Crowd4U. CyLog, deriving from a Datalog-like language developed by the FusionCOMP project, refers to a database abstraction that handles complex data-centric human/machine computations. There are three extensions dealing with human/machine computation: 1) allowing predicates to be open to crowd input, 2) a built-in reward system at the language level to give well-defined semantics, and 3) allowing one to dynamically add/delete the rules in a program and make higher-order crowdsourcing possible.

Using the logic programming paradigm, Lü and Fogarty [2012] present CrowdLogic which is a general-purpose language allowing the developer to declaratively specify the high-level control flow of algorithms for executing human computation via imperative programming (e.g., Java). This study enables the interpreter to optimize an algorithm for trade-offs between the costs of traditional computation and the costs of human tasks.

Gonnokami et al. [2013] propose the Condition-Task-Store (CTS) abstraction which is a declarative approach to implementing complex data-centric crowdsourcing with micro-tasks. The work shows that the proposed framework is able to naturally extend the task template adopted by many crowdsourcing platforms to define microtasks, to allow a declarative description of crowdsourcing systems and to have large expressive power. A task template is introduced as the core component of CTS. The template forms a set of CTS rules to define the microtasks and can be used in many kinds of crowdsourcing platforms, such as MTurk and CrowdForge. This framework also introduces a novel criterion to measure the expressive power of programming languages for crowdsourcing by focusing on the class of games. The class represents the size of the mechanism design space, i.e., a set of possible mechanisms that can be implemented to exploit the wisdom of the crowd. The results showed that the proposed framework is not only Turing complete, it can also support a wide range of game structures.

The above work and our work uses logic programming languages. In general, logic languages provide a suite of built-in functions to support programming. These languages are easy to use and provide rapid development support for implementation and maintenance. Another advantage of using logic programming languages is the potential of using backtracking to generate answers. The backtracking method is a mechanism manifested in a form of searching. When there are multiple clauses defining a relation, it is possible that either some of the clauses defining the relation are not applicable in a particular instance or that there are multiple solutions. To cooperate with the crowdsourcing paradigm, however, there are key issues that need to be considered during the designing phase in logic programming systems.

One important issue is to allow some predicates to be open, i.e. to be answered by the crowd; rather than a closed set of facts. These predicates enable users to send a query to the crowd in order to solve complex problems. Also, the open predicates have to be designed with more flexibility. For instance, each open predicate consists of inputs for sending the task's conditions to the crowd and outputs for accumulating the crowd's results. Moreover, one of the most important issues facing crowdsourcing is how to manage the answers returned by the crowd. As there may be a delay for the crowd to provide answers via any crowdsourcing platform, the logic programming systems have to be designed to tackle this problem in relation to this possible delay.

2.2.2 Imperative Programming Languages

Imperative programming is a programming paradigm that describes computation in terms of statements that change a program state.[5] This is the same way as the imperative sentences in natural languages which express commands to take action. Therefore, imperative programs define sequences of commands for the computer to perform. The characteristics of programming languages based on the imperative paradigm consist of 1) the procedure which is a sequence of statements, 2) the modification of the sequential flow of execution such as conditional or looping statements, and 3) the role of variables to serve as abstractions of hardware memory cells [Sebesta, 2012, p. 7-31].

Many crowdsourcing systems [Doan et al., 2011] have been proposed in the last couple of years, including crowd programming approaches which rely on imperative programming models. These programmatic methods are often offered as programmable access to specifying the interaction with crowdsourcing services.

Turkit [Little et al., 2010b] and Turkalytics [Heymann and Garcia-Molina, 2011] are implemented using scripts extending Javascript and provide an API wrapper enabling communication with the MTurk platform. In Turkit [Little et al., 2010b], additional functions have been written to support common subroutines and help make writing human computation algorithms easier. A Turkit script uses the crash-and-rerun programming model in order to guarantee that long running processes can be seamlessly resumed after a crash. Turkalytics [Heymann and Garcia-Molina, 2011] has programming libraries designed to enable code reuse and data sharing.

Jabberwocky [Ahmad et al., 2011] is procedurally programmed and compiles into a functional programming framework inspired by MapReduce [Kittur et al., 2011]. This framework is designed to enable cross-platform programming for human computation. The component code stack enables functions on arbitrary communication graphs between humans and machines, as well as provides parameterized functions for standard human activities.

Barowy et al. [2012] introduce Automan which is a set of function calls implemented as an embedded domain-specific language in the Scala programing language. It focuses on the automated management of the quality of answers and budget management. The authors claim that the model can be used to construct arbitrarily complex computations.

[5] http://en.wikipedia.org/wiki/Imperative_programming

2.2.3 Model-based Languages

Model-based languages refer to any language used to express information or knowledge or systems in a structure that is defined by a consistent set of rules.[6] Work has been proposed as modelling languages to support the development of complex crowdsourcing systems. CrowdLang [Minder and Bernstein, 2011] is a language for describing crowdsourcing systems in terms of basic operators, data items, and control flow constructs. It has been introduced as a concept for a general-purpose framework for interweaving human and machine computation within complex problem solving.

The proposed framework consists of three major components. First, the CrowdLang Library simplifies a design of new human computation systems and supports the seamless reuse of existing interaction patterns by providing an extensible programming library. Second, the CrowdLang Engine addresses the technical challenges of executing human computation algorithms by managing crowd latency, debugging human computation code, and re-executing human computation after exceptions. Third, the CrowdLang Integrator integrates different execution platforms, such as micro-task markets and games with a purpose. The framework also supports operators for task decomposition and group decision processes. The proposed operators, such as the divide-and-conquer method, aggregation, multiply, merging or reducing for vertical task decomposition, the horizontal control flow of the algorithm, as well as for the aggregation of results. The authors claim that this framework is able to evaluate different design parameters such as the influence of reward functions, Q&A mechanisms, and different planning approaches to quality, price and throughput.

2.2.4 APIs

An Application Programming Interface, or API (in short), is a set of routines, protocols, and tools for building software applications.[7] In other words, an API refers to an abstraction specifying how one software program can request services from another software program [Matthew C. Wagner and Morozova, 2013]. An API has a set of idioms that define the proper way for developers to request services from their applications. The purpose of the API is to facilitate interaction between these programs.

[6] http://en.wikipedia.org/wiki/Modeling_language
[7] http://en.wikipedia.org/wiki/Application_programming_interface

In crowdsourcing platforms, APIs are used to link platforms to other platforms or applications, mainly with the purpose of giving developers access and building applications around their data. It means that the platforms are able to make the applications reach outside the boundaries of their users, connecting with other platform users, generating a larger volume and diversity of contractors and applications. Today, most crowdsourcing platforms in the marketplace intend to be picked up and used by outside developers. They offer APIs that let developers build applications on their platform. Many crowdsourcing APIs, both newbie and popular APIs, are published on the API's website.[8]

In platforms like MTurk[9], ClickWork[10] or MicroWorker[11] which make requests to humans to do simple tasks for a little amount of money, their APIs typically allow developers to submit tasks to their services, approve completed tasks, and incorporate the answers into developers' software applications. On the developers' side, the application acts as the remote procedure caller sending the request to the service and receiving the results. Moreover, work has been done to customize crowdsourcing platforms in the market. To help people structure the tasks and improve the quality of results, APIs are built on top of these platforms. For example, as mentioned earlier, Turkit [Little et al., 2010b] is a tool providing a programming API to run iterative tasks on MTurk called the crash-and-rerun programming model. In this model, a program can be executed many times, without repeating costly work. It allows programmers to write algorithmic tasks as simple straight-line imperative programs in order to connect to MTurk.

Matera et al. [2014] investigated more user-friendly services by introducing a system built as user-friendly software running on top of Amazon Web Services (AWS). They implement a crowdsourcing task manager that allows task requesters to deploy such tasks with just a few clicks and share the task easily with their workers. The crowdsourcing platforms are not only based on labor-paid crowdsourcing but also, social media platforms like Twitter[12], Facebook[13], YouTube[14] and Flickr[15] have also played a critical role in generating crowdsourced data. For example, after the earthquake in Haiti and Indonesia or even the tsunami in Japan,

[8] http://www.programmableweb.com/category/crowdsourcing/apis?category=20163
[9] https://www.mturk.com/
[10] http://www.clickworker.com/
[11] https://microworkers.com/
[12] https://twitter.com/
[13] https://www.facebook.com/
[14] https://www.youtube.com/
[15] https://www.flickr.com/

people published numerous texts and photos about their experiences via social media sites. Donations, volunteers and tools to help started arriving within an hour, exemplifying one benefit of the powerful propagation capability of social media sites. Developers can make use of social media as augmenting tools to support collaborative development. Their social media APIs allow application to use the social connections and profile information in order to propagate tasks or activities among friends or public. With APIs like Facebook, developers are able to use social context in their crowdsourcing applications by utilizing profile, friend, page group, photo, and event data.

2.2.5 Summary

In this subsection, we classified the programming language support for crowdsourcing development support into four groups 1) declarative programming languages, 2) imperative programming languages, 3) model-based languages, and 4) APIs. These programming languages are considered as very necessary for the development of crowdsourcing systems, and the most appropriate language must be carefully selected. As we can see, each type of programming language has its own distinctive functions, depending on what kind of work it does. For example, declarative programming languages, which include SQL-like and Prolog, are well-known for their expressive power, ease of use, and rapid development. By contrast, imperative programming languages, such as Java, emphasize the sequential flow of execution, and it is, thus, appropriate for work which requires both conditional and looping statements and has complicated algorithms. Finally, we illustrate several tools and platforms that provide the services for building crowdsourcing applications known as APIs.

2.3 Crowdsourcing Workflow Algorithms

In traditional computation, all processes exploit computers to tackle problems and algorithms have been designed to automatically solve these tasks. Many sophisticated algorithms, such as for object recognition, sorting quantifiable objects, translating text from one language to others, removing spelling and correcting grammar mistakes, have been developed to automatically address specific types of issues. Despite their availability, these algorithms are still far from achieving

human-level abilities [Zhao and Zhu, 2014]. Recently, there has been an emergence of the crowdsourcing computation concept to address such issues difficult for machines. Crowdsourcing algorithms are based on the ground concepts of the study of algorithms in computer science, such as operation and control structures and programming paradigms [Law and Ahn, 2011].

For crowdsourcing computation, the developer writes procedures that coordinate the efforts of a crowd on a large scale in order to tackle complex tasks. A crowdsourcing algorithm refers to a set of instructions designed to organize humans to carry out the process of computation. These algorithms can include the processes of partitioning complex tasks into smaller subtasks, assigning tasks to a crowd, collecting results, and cleansing and aggregating the answers provided by humans. Moreover, the term crowdsourcing algorithm also refers to the logic that performs a specific function of interest e.g., finding majority voting or filtering the items. The aim of this section is not to provide a detailed review of all crowdsourcing algorithms; rather we refer the reader to others [Quinn and Bederson, 2009, 2011; Yuen et al., 2009, 2011], but to discuss a representative sample that relates to the work in this thesis.

Workflow designs of crowdsourcing refers to the process of planning and executing the complex tasks that can be accurately solved by a pool of crowd workers. Such workflows may decompose larger tasks into smaller subtasks, and later compose subtask solutions into an overall work product. Several research projects explore crowdsourcing workflow designs. In this section, we review some significant work that inspired and motivated the research in this thesis. In the literature, they serve as basic building blocks for crowdsourcing algorithms, which can be generally classified into three main patterns as follows.

2.3.1 Divide-and-Conquer Pattern

Solving complex large problems is not trivial for an individual crowd worker. Thus, the process of decomposing jobs into multiple tasks, which requires multiple workers, needs careful attention in workflow design in crowdsourcing. Some researchers have applied a divide-and-conquer algorithmic principle to solve complex tasks. For example, Zhang et al. [2011] address the topic by investigating the cooperation between algorithmic concepts and human potential for enabling the crowd to effectively play a role in an organized problem-solving process.

They illustrate their work through an example and identify three subareas of their study: 1) applying divide-and-conquer as an algorithmic paradigm to design the task's pattern, 2) taking advantage of the capabilities of people to guide the control flow of an algorithm, and 3) using a crowd as a general problem solver and engaging the crowd in the process of constructing an overall plan for the problem-solving process and for executing the plan.

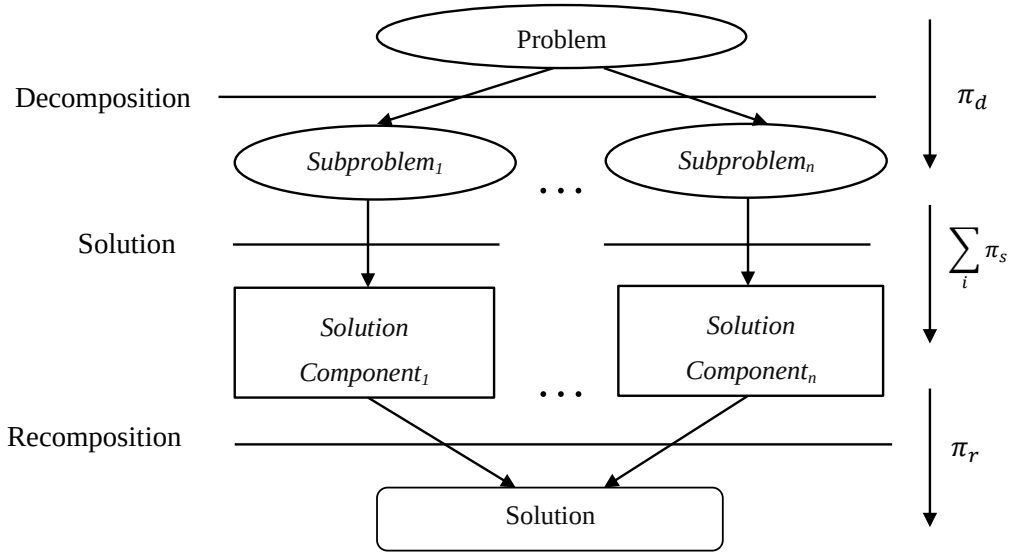


Figure 2.3: Reasoning about the structure of divide-and-conquer algorithms through considering costs of decomposition, subproblem solution, and recomposition (adapted from [Horvitz, 1987])

Figure 2.3 provides a structural view of divide-and-conquer algorithms for distributed human computation consisting of decomposing, solving and recomposing [Horvitz, 1987]. It refers to the process of decomposing a problem into sub-problems and composing solutions of sub-problems into a solution.

Recent work by Negri et al. [2011] and Aoki and Morishima [2013], for example, applied the divide-and-conquer approach to various crowdsourcing applications. According to Negri et al. [2011], a textual entailment corpora acquisition has been solved. The complex tasks are separated into a pipeline of simpler subtasks which are accessible to a large crowd of non-experts, and the quality control mechanisms are integrated into each stage of the whole process. Later, the application of the divide-and-conquer principle to data enumeration in the microtask-based framework is presented [Aoki and Morishima, 2013].

The method allows workers to join the process of gradually generating smaller tasks for enumerating data items in the divide-and-conquer fashion; therefore, the programmer does not need to provide many microtasks in advance.

In text editing tasks, crowdsourced human contributions are directly integrated into the user interface [Bernstein et al., 2010; Kittur et al., 2011]. In [Bernstein et al., 2010], the authors established a workflow pattern for proofreading and editing text into three stages called Find-Fix-Verify. Rather than asking a single crowd worker to read and edit an entire paragraph, which might result in poor quality work, this procedure recruits a group of workers to find candidate areas for improvement, then revise a set of candidate improvements, and finally filters out incorrect candidates. The Find-Fix-Verify process divides a task in a manner that maintains accuracy and reliability. By applying MapReduce, a programming framework based on divide-and-conquer, Kittur et al. [2011] introduced a system called CrowdForge and constructed human computation algorithms for writing simple articles and making product comparisons. Kulkarni et al. [2012] also applied the divide-and-conquer approach to crowdsourcing in a more general setting and presented an experimental evaluation. They found it difficult for the crowd to effectively divide the problem without appropriate guidance.

2.3.2 Iterative Improvement Pattern

The basic idea of iterative improvement has emerged from a situation that an individual working on a task independently lacks sufficient capabilities to complete the task accurately. For instance, when the requester asked workers to edit text in open-ended tasks like Wikipedia[16], they found that roughly 30% of the results are poor because the workers put in minimal effort to complete these tasks and did not submit valuable work [Bernstein et al., 2010]. Thus, the repeating control flow with crowd workers is able to improve the quality of tasks [Little et al., 2010a]. The iterative improvement workflow is a sequence of creation tasks where the result of each task feeds into the next one, as shown in Figure 2.4

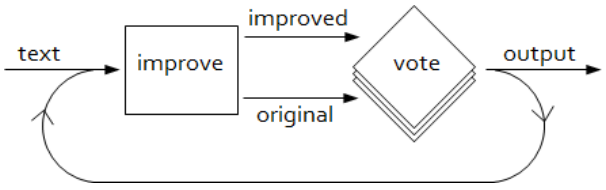


Figure 2.4: Flowchart for the iterative text improvement task (adapted from [Little et al., 2009])

The iterative improvement design pattern was first described by Little et al. [2010b]. They introduced TurKit as a toolkit for deploying iterative tasks to

[16]https://www.wikipedia.org/

Mechanical Turk. Initial findings state that iterative workflows improve the quality of results for refining tasks, such as writing or collective brainstorming at using multiple workers to build on and improve upon an existing task. Using the crash-and-rerun model, it allows the developer to focus on imperative ordinary function calls and provides an abstraction over the specific features and synchronization issues of Mechanical Turk. The model addresses the issues related to high-cost and high-latency steps involving humans. It was found to be cheap to rerun an entire program up to the point where it crashes, as long as it runs locally whereas the results must be stored in a database in order to be accessible in future reruns for remote and costly operations.

With the iterative workflow having several issues to be decided, such as how many votes it takes to determine the consensus opinion or how many times to repeat the improvement loop, there has been lately an increasing interest in the optimization of iterative crowdsourced workflow [Law and Ahn, 2011]. From an active learning perspective, a machine leaning model is able to determine which work products may still be improved and then assign workers most likely to make such improvements [Kittur et al., 2013]. For example, Dai et al. [2010; 2011] proposes TURKONTROL, a theoretical model based on a Partially Observable Markov Decision Process (POMDPs) model to optimize an iterative improvement workflow. Its model automatically specifies which operation to execute next based on the probability that an average worker can improve upon the current solution. The findings show that higher quality outputs in a simulated environment had been obtained. In the meantime, the work in [Lin et al., 2012] and [Petuchowski and Lease, 2014] optimized an iterative improvement algorithm so that the error, computation time and memory usage can be reduced.

Bernstein et al. [2010] introduced a find-fix-verify workflow pattern to improve the quality of proofreading and document shortening. The iterative workflow is used in their work (as well as divide-and-conquer as mentioned earlier). Find-Fix-Verify splits complex crowd intelligence tasks into a series of generation and review stages that utilize independent agreement and voting to produce reliable results. From this pattern, the task was broken down into three separate workflow stages. The first stage, known as the Find stage, asks workers to highlight the areas of a text that could be shortened. In the Fix stage, workers are asked to edit and shorten the highlighted areas without changing the meaning of the text. Finally, the workers are asked at the Verify stage to flag edits that change the meaning of the text. This process prevents errant crowd workers from

contributing too much, too little, or introducing errors into the document. Moreover, there are several kinds of crowd tasks in addition to text editing jobs to which the iterative design workflow can be applied; for instance, counting calories on the food plate [Noronha et al., 2011], creating taxonomies to organize information [Chilton et al., 2013; Karampinas and Triantafillou, 2012] speech-to-text transcription [Chen et al., 2011] and ground-truth generation in the medical area [Foncubierta Rodríguez and Müller, 2012]. The human-computation algorithms have been developed for these applications. These algorithms partition the tasks into a group of sub-tasks and then each subtask is sent through an iterative structure that allows the workers to iteratively refine the tasks to potentially achieve high accuracy in the responses.

2.3.3 Redundancy-based Quality Control Pattern

One of the major challenges in developing crowdsourcing systems is quality control to ensure the quality of crowdsourcing results, since there is no guarantee that all workers have sufficient abilities or motivations to complete the offered tasks at a satisfactory level of quality. One way to approach this problem is to introduce redundancy and find ways to filter noisy inputs. Consider the following example: a requester intends to describe photos by tagging them with meaningful and descriptive keywords in order to categorize these photos in a library. Each photo is assigned to a worker with the proper skills necessary to complete it. However, by asking a sufficient number of workers to perform the same task independently, we are able to gain the most common responses as the solution and expect a high quality of the "correct" (i.e., majority) answer.

In recent years, there has been an increasing interest in developing quality control mechanisms for crowdsourcing that aim to make sufficient use of redundancy to guarantee high quality results. Much work identifies the "correct" answer from multiple workers' responses by using majority voting. This technique simply chooses the answer on which the majority of workers agree. For example, Barowy et al. [2012] optimize the majority voting approach by introducing an algorithm to estimate the confidence level of the responses that would be acceptable by the requester. In doing this, the system is able to automatically schedule enough human jobs to reach the desired quality while staying under budget. Several other studies have used more sophisticated statistical aggregation techniques to compute the aggregated result of each task separately [Matsui et al., 2013]. According to a

comprehensive empirical study in [Quoc Viet Hung et al., 2013], the authors design a generic and extensible benchmarking framework to assist in the evaluation of different aggregation techniques so that they are useful for subsequent studies in the crowdsourcing area.

Although choosing the best answer from majority voting or a set of answers directly is a quality assurance mechanism, there is no guarantee that this mechanism is the best choice. Several recent studies have begun to examine the presumed quality of workers as judged by their past work. Liu et al. [2012b] studied the improvement using a quality control mechanism relying on workers' past performances. They introduced a verification model which is employed to replace the voting strategy. Intuitively, the system is more likely to accept the answers provided by the worker with a good accuracy. Given the probability distribution of workers' performances, they apply the Bayesian theorem to estimate the accuracy of each result. With the different reliabilities and diverse backgrounds of crowd workers, a large volume of work has been proposed to deal with the uncertainty and diversity of the workers' reliabilities. These methods often have a form of weighted majority voting where the answers of the majority of the workers are selected. Von Ahn et al. [2008] and Liu et al. [2013] show that the workers' reliabilities can be estimated using gold standard questions with known answers. Jin et al. [2003] and Raykar et al. [2010] applied a statistical method and an Expectation Maximization (EM) algorithm to infer the answers of the tasks and the reliability of workers. This algorithm has been used in classification problems where the training data is annotated by a low-cost noisy labeler. This EM approach has been applied to a more complicated probabilistic model for image labeling proposed in [Welinder et al., 2010; Whitehill et al., 2009]. The EM algorithm has been used in the machine learning community for several decades so that there is a lot of work and reviews on this topic [Dawid et al., 1979; McLachlan and Krishnan, 2008]. The EM algorithm provides maximum likelihood estimates for hidden model parameters, based on a sequence of steps that converge to a locally optimal solution.

2.3.4 Summary

In this subsection, we study crowdsourcing algorithms that coordinate the efforts of a crowd to tackle complex tasks. The essential workflow designs of crowdsourcing have been categorized into three patterns, namely the divide-and-conquer, iterative improvement, and redundancy-based quality control patterns.

Each pattern refers to the process of planning and executing the complex tasks that can be accurately solved by a pool of crowd workers. The divide-and-conquer workflow relates to a decomposition of a problem into sub-problems and the composition of solutions of the sub-problems into a solution whereas the iterative improvement pattern focuses on running recursive tasks to improve the quality of the solutions. For the redundancy-based quality control pattern, as its name suggests, this approach makes sufficient use of redundancy to guarantee high quality and reliable results.

2.4 Crowdsourcing Applicability

Crowdsourcing can be utilized for different applications for both scientific and business purposes. In several previous studies, the applications of crowdsourcing have been extensively reviewed. In one of the earliest articles introducing crowdsourcing, Howe [2006] discusses the concept through many applications in solving real business issues, for example, iStockphoto for image sharing, InnoCentive for problem solving and MTurk for micro-tasks. The applications are not limited only to business problems, but crowdsourcing is also applied to scientific research and engineering fields, such as volunteered geographic information [Goodchild and Glennon, 2010], the cultural heritage domain [Oomen and Aroyo, 2011], and the public health ecosystem [Swan, 2012].

Different literature surveys have proposed typologies of crowdsourcing applications from different perspectives. In an early study, Howe [2006] defines four basic categories of crowdsourcing applications: crowd wisdom or collective intelligence, crowd creation or user-generated content, crowd voting, and crowdfunding. He states that crowdsourcing is a complex phenomenon and often involves a combination of these categories, thus it can be hard to classify. Later, many researchers attempted to explore the applications of crowdsourcing based on different purposes and aspects. Of these, Whitla [2009], focusing on marketing-related areas, classifies crowdsourcing applications into three categories: product development, advertising and promotion, and market research. Brabham [2012] identifies crowdsourcing applications based on four different functions for addressing the problem solving process, namely knowledge discovery and management, broadcast search, peer-vetted creative production, and distributed human intelligence tasking. Yuen et al. [2011] studied crowdsourcing systems and grouped crowdsourcing applications into four categories: voting systems, information sharing systems, games and

creative systems. Zhao and Zhu [2014] point out that context and function are important characteristics to categorize crowdsourcing applications. The authors extracted functions of 126 crowdsourcing applications, and grouped them into four categories: design and development, test and evaluation, idea and consultant, and others. Contexts are divided into two: business and non-business. A business context consists of for-profit organizations, while non-business includes non-profit organizations and institutions.

Here, we explore crowdsourcing applications from the programming perspective within the software development process. Malone and Laubacher [2010] studied the building blocks of crowdsourcing systems. As they mentioned, a crowdsourcing system needs to address a number of fundamental challenges including deciding on the type of contributions that users can make, recruiting and retaining users, combining contributions to solve the problem, and evaluating users and their contributions. Similarly, David Geiger [2011] design four basic questions that are important for building a crowdsourcing application: "What is being done?", "Who is doing it?", "Why are they doing it?", and "How is it being done?". From this point, we group the applications of crowdsourcing into two categories: general purpose and domain-specific purpose. Table 2.1 summarizes the characteristics of crowdsourcing applications according to the proposed categorization.

First, crowdsourcing applications with a general purpose have been designed to support a wide variety of tasks which do not require special expertise from the contributors and are not targeted to any user group in particular. For example, MTurk, the popular crowdsourcing platform, has been designed for general purpose usage, where anyone can post tasks and offers the practitioner a way to quickly access a large user pool, collect data, and compensate users with micropayments. Other crowdsourcing applications, such as Deco [Parameswaran et al., 2012] and Turkit [Little et al., 2009] are also developed with general objectives. As proposed, the programming languages are designed to be general to create a variety of tasks and then to be integrated with an existing crowdsourcing market. Second, crowdsourcing applications developed with domain-specific purposes are designed to contribute to particular situations, such as tasks performed in a particular domain or a participator requiring domain-specific knowledge or skills. For example, developing or testing open source software through crowdsourcing requires expertise in particular programming languages and platforms [Misra et al., 2014] According to the classification of crowdsourcing applications as general or domain-specific purpose, we consider four dimensions as follows.

Table 2.1: An summary of the characteristics of crowdsourcing applications distinguished as general purpose or domain-specific purpose

	Crowdsourcing Applications	General Purpose	Domain-Specific Purpose
Dimensions	**Task (What is being done?)**	▪ Simple tasks e.g., image tagging, translation of simple text, ranking ▪ Require low involvement to complete the task ▪ Use generic skills to solve the problems	▪ Complex problems e.g., software development task, writing academic paper ▪ Invest considerable effort to complete the task ▪ Require expertise or specific skills for problem solving
	Contributor (Who is doing it?)	▪ Moderate education and training ▪ Non expert	▪ Substantial domain understanding ▪ Special expertise
	Motivation (Why are they doing it?)	▪ Intrinsic motivation such as payment, financial reward, social obligation	▪ Intrinsic motivation based on satisfaction of specific participant such as passion, enjoyment, community identification, personal achievement
	Evaluation Mechanism (How is it done?)	▪ Majority rating or decision ▪ Integration/ aggregation ▪ Assign a task to multiple users and select the result that was most commonly returned	▪ Various data processing techniques to combine the results e.g., data mining, machine learning algorithms ▪ Individual contribution representing the solution in specific domain and select the best one by other groups of people
	Examples	▪ Market Platform e.g., Mechanical Turk, MicroWork, iStockPhoto ▪ Programming Platform e.g., Turkit, Automan, Turkomatic, Deco, CrowdSearcher	▪ Software development e.g., uTest, mob4hire.com, TopCoder.com, CrowdSpirit ▪ Game e.g., ESP game, reCAPTCHA ▪ Community e.g., Ushahidi-Haiti ▪ Creativity and design e.g., 99designs, Atizo, InnoCentive, Threadless, IdeaStrom

• ***Task.*** In crowdsourcing, the task refers to a human intelligence required task that could be either simple or complex. Schenk and Guittard [2011] classify crowd tasks as either simple or complex. A simple task, such as data collection, the translation of simple texts or even image marking, can be carried out easily and cheaply, whereas a complex task in the context of innovation projects or problem solving requires complex processes and much more effort.

The task in a general purpose system tends to be a simple task that requires relatively low involvement from the worker to complete, with no specific knowledge skill required to solve problems. In a typical example, such as crowd rating of certain products, workers need to give scores or opinions, and maybe combine and then evaluate results. In contrast, tasks with a domain-specific purpose relate to a task in a specific domain or it tends to be a complex task which has high structural complexity and intellectual demands. In a creation system such as

uTest[17] which is a crowdsourcing software testing service, professional testers are required. Domain specific systems tend to be complex and address intellectually demanding problems that require contributors to invest considerable effort.

• ***Contributor.*** Geiger et al. [2011] state that the nature of the contributors in a crowdsourcing system correlates strongly with the characteristics of the tasks performed. Consequently, the role and nature of its crowd contributors can differ substantially. Rouse [2010] noted that simple tasks can be accomplished with moderate education and training whereas sophisticated tasks need substantial domain understanding and highly developed business acumen. Steinfeld et al. [2013] categorized public participation as either general purpose or domain-specific systems. General purpose systems do not require special expertise from contributors and are not targeted to any user group in particular, while domain-specific systems are designed for user groups that have a special purpose.

Crowdsourcing applications can be decomposed according to the capability of the required contributors which depends on the complexity and skills involved in the tasks. Therefore, we distinguish the crowd participants as general and specific purpose as well. A crowdsourcing application may require a non-expert or a moderate education for participants to complete the tasks. Also, a crowdsourcing application with a domain-specific purpose often requires expertise and substantial knowledge in a particular domain to do the tasks. For example, an image-tagging task or ranking products does not require much specialized expertise to perform, which means almost anyone can do the task. On the other hand, crowdsourcing applications, such as uTest.com, mob4hire.com[18], TopCoder.com[19], CrowdSpirit[20] and PeoplePerHour[21] involve development and testing, so this crowdsourcing requires specific abilities, specializations, or skills on the part of participants.

• ***Motivation.*** Motivation and incentive are critical factors to attract participation in crowdsourcing applications. The motivation is the recruitment strategy which requires users to contribute to crowd tasks in crowdsourcing systems. Doan et al. [2011] discuss crowdsourcing systems on the Web from a variety of perspectives. They introduce the nature of collaboration on crowd contribution in two aspects: *explicit*, allowing contributors to build artifacts that are beneficial to the whole community, and *implicit*, permitting contributors to solve a problem as a

[17]http://www.utest.com/
[18]http://www.mob4hire.com/
[19]http://www.topcoder.com/
[20]http://www.crowdspirit.com
[21]http://www.peopleperhour.com

side-effect of something else they are doing. They also consider recruitment strategies in five major aspects: authority, pay for users, asking for volunteers, making contributions as a requirement to use a different service, and piggybacking on established systems.

Kaufmann et al. [2011] explore the worker's motivation in crowdsourcing. According to their study, the motivating factors are categorized into intrinsic (e.g., enjoyment and community motivation) and extrinsic motivation (e.g., immediate payoffs, delayed payoffs, social motivation). Intrinsic motivation exists if an individual's act on the activity is driven by internal factors, e.g., acting just for fun or enjoyment. With extrinsic motivation, the activity is just an external instrument for achieving a certain desired outcome, e.g., acting for money or to avoid sanctions. They found that the intrinsic motivation factors seem to dominate the extrinsic ones, especially the different facets of enjoyment-based motivation like task autonomy and skill variety.

Related to our classification, the motivation aspect can be examined by dividing it into general and domain-specific purposes for crowdsourcing applications. Similar to the study of Doan et al. and Kaufmann et al., the motivation dimension of a crowdsourcing application with a general purpose relates to activities including payment, financial reward, and social obligation which impact upon contributors to participate in the tasks. On the other hand, the motivation with a domain-specific purpose in crowdsourcing applications might be expected to be based on the satisfaction associated with the activity itself such as passion, enjoyment, community identification, or personal achievement. An explicit example for the general purpose is the financial reward for micro-tasks, such as MTurk, iStockPhoto[22], reCAPTCHA[23], 99designs[24] or Netflix Prize[25]. For domain-specific purposes, examples are ESP game[26], Google's AdWords[27], and Ushahidi-Haiti[28].

• ***Evaluation Mechanism.*** Evaluation plays a vital role in providing feedback to the requester in order to increase quality as well as in selecting the best result from a large set of crowd solutions.

Several studies attempt to control effective feedback and produce better results. Dow et al. [2012] investigate evaluation mechanisms of a feedback system

[22]http://www.istockphoto.com
[23]http://www.google.com/recaptcha/intro/index.html
[24]http://99designs.com
[25]http://www.netflixprize.com
[26]http://www.esp-games.com
[27]https://www.google.com/adwords
[28]http://www.ushahidi.com

for crowdsourced work. The authors introduced the Shepherd system to examine the effects of different kinds of assessment in crowds. The system supports three feedback modes for the workers: no assessment, self-assessment, and external assessment. In the no-assessment condition, participants move forward directly from one review to the next, and they do not have an opportunity to modify their reviews. For the self-assessment mode, participants have the opportunity to edit their own reviews. In the external assessment condition, an expert reads, judges each participant's review and then returns the assessment to participants to optionally edits his/her review before submitting the results. They concluded that both external and self-assessment led to a significantly higher quality of work than no feedback.

After the crowd has contributed to the tasks, all answers are collected and combined with a technique called aggregation of contributions. Answer aggregation is one of the most important challenges of crowdsourcing. Schenk and Guittard [2011] noted the fundamental distinctions in aggregation processes for crowdsourcing, including integrative and selective methods. Integrative crowdsourcing creates value by pooling potentially large quantities of complementary input whereas selective crowdsourcing generates value by asking the crowd to provide a set of options. In addition, the mechanisms for utilizing the collective intelligence of the crowd for evaluation purposes are broadly explored. Hung et al. [2013] evaluate many aggregation techniques and then classify them based on their computing model into two main categories: non-iterative and iterative aggregation techniques. The non-iterative approach, such as majority decision [Kuncheva et al., 2003], Honeypot [Lee et al., 2010] and ELICE [Khattak and Salleb-Aouissi, 2011], uses heuristics to compute a single aggregated value of each task separately. These techniques differ in the preprocessing step and the probability of computation. The preprocessing process is not required in the majority decision scheme whereas Honeypot requires the filter method for filtering spam answers. ELICE focused on worker expertise and question difficulty. Iterative aggregation crowdsourcing performs a series of iterations which consists of a sequence of computational rounds by updating the object probabilities and repeating until convergence. The widely used approaches in this category include Expectation Maximization [Ipeirotis et al., 2010], Supervised Learning from Multiple Experts [Raykar et al., 2009] and Iterative Learning [Karger et al., 2011].

Based on our categories, crowdsourcing applications with general purpose evaluation mechanisms use a basic approach for gathering contributions from the

crowd, such as majority rating/decision and integration aggregation. Along the general purpose dimension, our work focuses on assigning a task to multiple users who submit their individual results to crowdsourcing systems and then select the result that was most commonly returned. Another typical example is majority voting for the product of interest in MTurk; this does not only focus on a loose combination of user contributions with little effort it but also emphasizes a tight combination with substantial effort, e.g., merging code in software development in uTest, and no real combination at all, e.g., merely listing textual reviews in Wikipedia. Conversely, the domain-specific purpose of crowdsourcing applications in this dimension describes individual contributions that represent the solution to a specific problem and then selects the best one. This is mainly used in contests, e.g.,99designs, Atizo[29], InnoCentive[30], Netflix Prize, IdeaStrom[31] or Threadless[32]. Furthermore, various data processing techniques, such as data mining, machine learning and other sophisticated algorithms (e.g., Expectation Maximization) are applied to these categories in order to pre-process, select and combine results that are often noisy and contain redundant data.

2.4.1 Summary

In several previous studies, crowdsourcing has been widely applied for both scientific and business purposes. We have explored crowdsourcing applications and categorized them into two main groups, including general and domain-specific purpose. With general purpose, crowdsourcing applications have been designed to support a wide range of tasks which do not require special expertise. In contrast, crowdsourcing applications developed for the domain-specific purposes are designed to contribute to particular situations such as tasks performed in a particular domain or a participant requiring the domain-specific knowledge or skills.

2.5 Crowdsourcing Architectures

In the past decade, there has been active research on crowdsourcing [Doan et al., 2011; Geiger et al., 2011; Pedersen et al., 2013; Quinn and Bederson, 2011; Yuen et al., 2009]. In the existing literature, crowdsourcing has been explored from many

[29]https://www.atizo.com
[30]http://www.innocentive.com
[31]http://www.ideastorm.com
[32]https://www.threadless.com

different perspectives, especially in technical domain independent perspectives and business domain-specific perspectives [Luz et al., 2014].

From a technical domain independent perspective, crowdsourcing research has focused on methods, frameworks, components and technologies for solving problems which merge human intelligence into machine computation. For example, Doan et al. [2011] review many crowdsourcing platforms on the Web and then classify them based on their applications. Both explicit and implicit crowdsourcing systems have been introduced and they also discuss the essential mechanisms for building crowdsourcing systems based upon a list of challenging questions; e.g., "how to recruit users?", "How to combine user contributions to solve the target problem?", "How to evaluate users and their contributions?". Pedersen et al. [2013] describe the conceptual model of crowdsourcing in the Information System (IS) aspect based on the traditional Input-Process-Output model. The model consists of six components involved in the crowdsourcing phenomenon that includes problem, process, people, governance, technology, and outcome. These components can bring a greater understanding of crowdsourcing from the IS aspect and provide benefits to future crowdsourcing researchers.

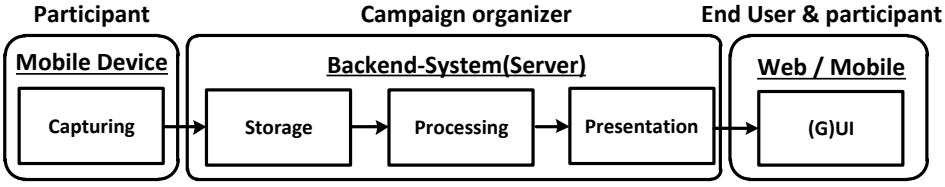


Figure 2.5: General Architecture for Crowdsourcing, from [Fuchs-Kittowski and Faust, 2014]

In the literature, most crowdsourcing applications have been built on Web technologies, allowing online workers to complete the task via the web [Doan et al., 2011]. These main components are generally organized as client-server architectures. According to Fuchs-Kittowski and Faust [2014], the general crowdsourcing architecture is composed of participants/clients and backend-systems/servers as show in Figure 2.5. On the client side, participants can be workers or requesters via web-based and mobile devices. The client functions as a data capturing device and provides the user interface. In contrast, the backend-system is for data storage, processing, and visualization. Likewise, Zhao and Zhu [2014] emphasize functional components regarding a transformation process. It is a process or collection of processes that transform inputs into outputs. There are three categories of components: assigners who initiate and manage the task, providers who respond

to the task and attempt to submit their solutions as feedback, and an mediation platform which links assigners and providers and serves as a crowdsourcing enabler. In addition, Estrin et al. [2010] and Khorashadi et al. [2013] present common architectural components with a particular focus on data capturing and processing.

Hetmank [2013] presents the typical components and functions that may be implemented in crowdsourcing systems with a special focus on campaign management. The author derives four components: user management, task management, contribution management, and workflow management. The crowdsourcing theoretical framework presented by Ponciano [2014] pays special attention to analyzing strategies for designing and managing distributed applications through crowdsourcing platforms. Their framework is designed to assist the analysis of the diverse aspects related to crowdsourcing applications. It is divided into three dimensions: QoS requirements which are requesters' effectiveness measures, design and management strategies related to how platforms manage application execution, and human aspects which are worker characteristics.

Several architectures of applications emphasize task management e.g., aggregating the results obtained from different crowd workers, and the distribution of data capturing tasks to participants. Luz et al. [2014] study crowdsourcing platforms with a focus on solving micro-tasks and complex tasks. They group the crowd's tasks into several subtasks such as partition task, aggregation task, qualification task, and grading task. By proposing a task-oriented crowdsourcing system, such tasks have relationships among the workers, the requesters, and the system itself. They attempt to generalize the task flow process as well as analyze the conceptual model through several existing crowdsourcing platforms. Difallah et al. [2013] propose a the framework driven by crowd tasks. The tasks are performed by the most suitable worker. Based on push technology, workers and tasks are automatically matched using an underlying categorization structure that exploits entities extracted from the task descriptions as well as the worker's profiles based on information available on social networks.

Crowdsourcing systems have not only been implemented on desktops or workstations, they also have been exploited using mobile devices. In recent years, numerous mobile crowdsourcing applications have emerged and have shown potential for business and society [Chatzimilioudis et al., 2012]. Fuchs-Kittowski and Faust

[2014] reviewed the related conceptual work in the domain of mobile crowdsourcing systems and proposed a general architecture for mobile crowdsourcing applications. The proposed architecture is divided into two parts: 1) the backend system or server which plans and monitors the targeted data collection effort as well as recruits and cooperates with well-suited participants, and 2) the mobile device or mobile client which contributes to the geo-crowdsourcing campaign by capturing and sharing geospatial data using their own mobile device. There are a number of general architectures proposed on mobile crowdsourcing applications like Medusa [Ra et al., 2012], Vita [Hu et al., 2013], MoCoMapps [Hupfer et al., 2012], PRISM [Das et al., 2010], and AnonySense [Cornelius et al., 2008]. These frameworks not only enhance the general architectures to solve cost-efficient development issues but also focus on generality, security, scalability, and privacy.

From the analysis of existing crowdsourcing applications and architectures, we will present a classification scheme and a general architecture with the typical roles, components and functionalities of crowdsourcing systems from the perspective of programmers and practitioners. The main goal of this section is to gain a better understanding of the typical functionalities and design aspects during the development and evaluation of crowdsourcing systems. The architectures of crowdsourcing applications are divided into two major aspects of crowdsourcing computing: 1) centralized, and 2) decentralized crowdsourcing architectures, as described below.

2.5.1 Centralized Crowdsourcing Application Architecture

This architecture is the client-server model; that is, the server component provides services, functions and resources to one or many clients that initiate requests for such services. In this model, all computing is done at a central location/server and computing resources reside at the primary data center. The clients or terminals only send requests to the center and then receive the results from their server. From the analysis of existing crowdsourcing applications and architectures, we organize this kind of computing in a centralized crowdsourcing model. The following is a generic architecture of a centralized crowdsourcing solution (see Figure 2.6).

The three typical roles of this architecture are: the three roles of 1) ***crowdsourcing manager*** that initiates and operates the crowd's tasks including coordinating with well-suited participants; 2) ***contributor*** that participates in the tasks and receives payment or rewards after evaluation by the requester, and is involved in the data capturing process, where such data could be the input of

users, services or sensors especially sensing data on mobile devices; and 3) ***requester/end user*** that accesses a GUI on web-based or mobile technologies and processes the data captured by the contributors according to requests, interests and needs, e.g., integrating, analyzing and remixing this data.

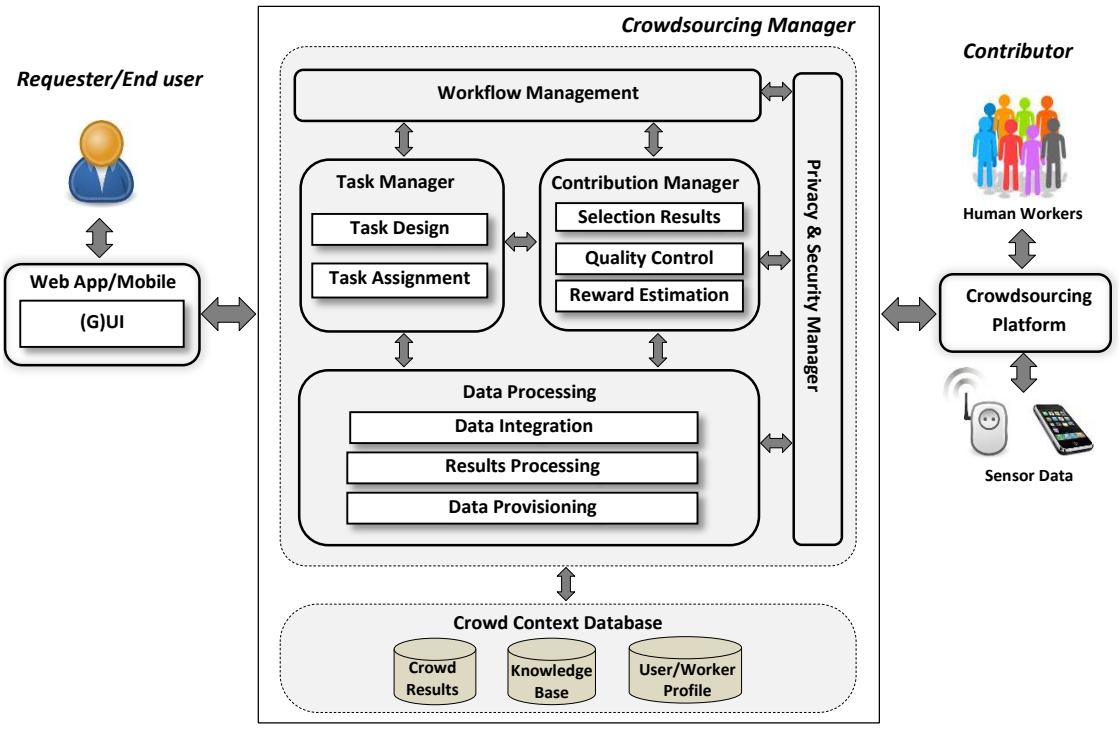


Figure 2.6: Architectural components and roles of centralized crowdsourcing applications

The functional tasks of the crowdsourcing manager component are:

Task Manager – This component handles the incoming submissions of tasks and their distribution to the crowd that will solve it. On task design, this sub-component dynamically creates as many tasks as required from the requester via a GUI. We realize that the quality of the results highly depends on this phase; thus, the key aspects of tasks such as, types of tasks, the process of executing tasks, and the corresponding instructions and constraints, must be determined before assigning tasks to the crowd. Apart from this, other components, such as keywords, candidate answers, maximum accepted answers, reward or incentive scheme, submission time and latency should be carefully defined. In the task assignment component, the crowdsourcing application should be concerned with assigning the right task to the right workers. Therefore, the task has been routed based on the task specification and the worker profile. Hetmank [2013] mentions

that the important conditions when assigning the task to the crowd are the sufficient worker's knowledge and skills to complete the task and an appropriate period of time when the worker can and is willing to work.

Contribution Manager – This component manages the process of selecting the solution from the crowd, the quality assurance as well as computing an appropriate reward for the crowd. To select the best of the feedback from a large group of people, several methods such as majority voting, control group [Hirth et al., 2013] or expert decision making have been used. Furthermore, various data processing techniques, such as data mining or machine learning algorithms, may be applied to select and combine results that are often overwhelming and contain redundant data. However, the quality control process is one of the key aspects to ensure qualified feedback from the workers. According to [Dow et al., 2012], the authors classify the evaluation mechanism into three levels including no assessment, self-assessment and external assessment. They also found that online workers produce better results when they self-assess or receive external feedback. Moreover, this component deals with the monetary reward given to the crowd in the case of paid crowdsourcing platforms. In this method, the system obtains a monetary reward by evaluating the quality of workers. The approaches such as majority agreement [Barowy et al., 2012], and using a set of known answers to check for errors and to identify workers who make many mistakes [Difallah et al., 2013; Kazai, 2011], have been used in this component.

Data Processing – The processing data component involves preparing captured data, storage, processing stored data and then arranging data for presentation. First of all, the data integration component extracts and transforms captured data from a large crowd into an internal data structure. Examples of data integration processes are situations where each vote is counted and stored in a conventional DB, and that audio files are analyzed by extracting words for speech recognition or matching with original sounds. The results processing component operates on the stored raw data to extract features of interest and obtain insight into the observed phenomenon. It also uses several approaches for processing such as data processing for numeric and statistics modelling, image processing and sophisticated machine learning algorithms. The last component relates to preparing data for presentation or for transferring to other systems. The presented data can be visualizations of the raw data or information to end users. Furthermore, the results are usually presented through web-based applications or on a mobile device.

Workflow Management – This component is for workflow design for complex tasks with requirements and constraints. The workflow can be decomposing larger tasks into smaller subtasks, and recomposing subtasks into the best solution as described earlier. To gain optimal results, a workflow coordinates among the inputs and the outputs of independent human or machine functions [Hetmank, 2013]. Many sophisticated workflow algorithms such as iterations for running recursive tasks are required to improve the quality of results.

Privacy and Security Manager – This component is concerned with participant privacy and rights, data security, as well as access control and authentication protection. Geiger et al. [2011] propose four levels of accessibility of peer contribution to the task; with four classes: none, view, assess, and modify. Each reflects the degree of privacy that a crowdsourcing process enables. This component protections the participants' right when a crowdsourcing platform makes use of participants' data without their consent [Schmidt, 2010]. The participants are able to use the privacy control provided by crowdsourcing systems in order to either remove data or prevent their personal information being used.

Crowd Context Database – The data repositories such as crowd results, end-user/participant profile and generic/specific knowledge base are kept in a crowdsourcing database. The crowd results repository is a temporary database recording the results of relevant crowd tasks. It also includes some additional data that may be derived based on the raw data that is for analysis and output display. The knowledge base repository stores long-term data which integrates either the conventional data relevant to the tasks or inferred data which is generated based on crowd input.

The knowledge base enables the inference process which is useful for doing probability issues from the past tasks or even learning the trustworthiness of workers [Amsterdamer et al., 2015]. Last, the user/worker profile repository contains details about the request and each of the crowd workers. It is useful for the management of workers, routing suitable tasks, rewarding and task assignment.

2.5.2 Decentralized Crowdsourcing Application Architecture

In contrast, in decentralized platforms, all computation and communication is performed locally in each peer in an appropriate manner. This framework is known

as distributed computing in which each node/peer of the system is equally responsible for contributing to the global result and could be located at different places with that the geographic location being relevant to the computational process itself [Duckham, 2013]. Thus, each node is able to process and distribute its information without relying on any centralized authority through mechanisms based on its interactions with the environment. In crowdsourcing systems, it generally can be considered as a distributed process of problem solving through a flexible group of human contributors. In decentralized methods, the systems propagate the request/task among many contributors especially on mobile devices, where all computing and communication is operated on locals. Then, these devices are permitted to fully manipulate and distribute the request to others via communication channels [Chatzimilioudis et al., 2012].

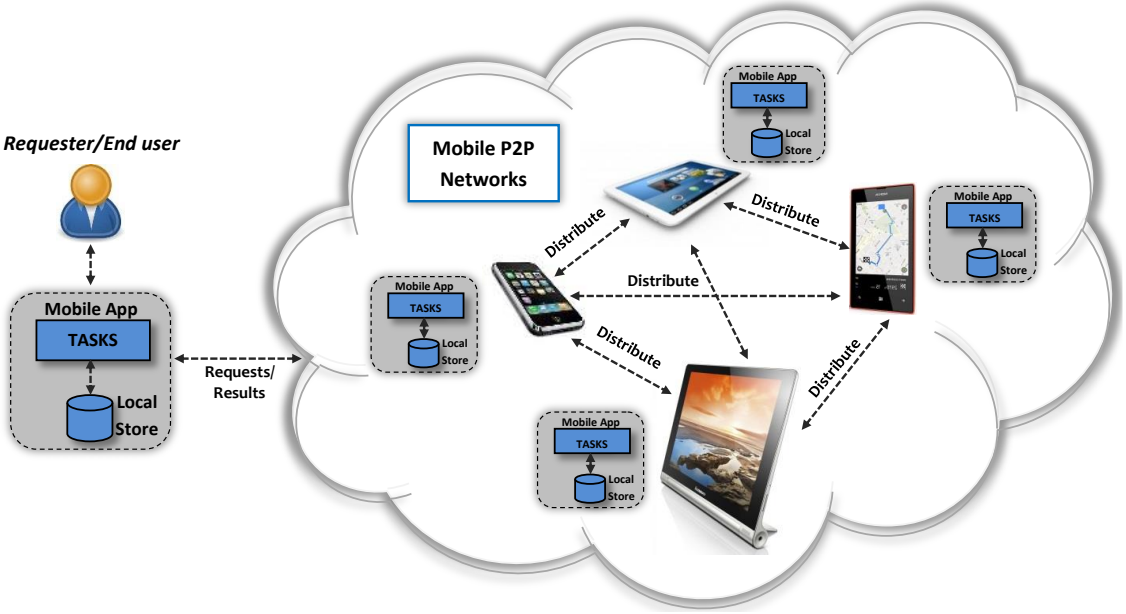


Figure 2.7: Architecture of decentralized crowdsourcing applications

There is a little work proposing completely decentralized crowdsourcing approaches from reviews such as [Chatzimilioudis et al., 2012]. Konstantinidis et al. [2011] propose the framework called SmartOpt for searching objects (e.g., image, video, etc.) which are captured by the user in a mobile social community. The main contribution of SmartOpt is to use location data made available by the crowd to optimize the search process with peer-to-peer systems. This approach is able to minimize energy consumption during searching, reduce the query response time when conducting the search and also maximize the recall rate of the user query. Constantinides et al. [2012] extends the sensing capability of smartphones by allowing them to identify their geographically nearest neighbouring nodes in

real-time called CrowdCast. This framework is beneficial to the crowdsourcing paradigm since it provides full access to mobile workforces and adds the temporal dimension to location data in order to exploit trajectory-related information.

After reviewing the decentralized approaches in crowdsourcing systems, our proposed generic architecture of decentralized crowdsourcing applications is shown in Figure 2.7. The mobile or other stationary devices are the resource providers/-workers, and performs, similarly to a mobile peer-to-peer network. Each peer is able to communicate and exchange data with each other in the local vicinity and also can be asked to contribute to identifying mobility patterns or the popularity of a given trajectory. Moreover, they are involved explicitly or implicitly (e.g., by allowing the capturing of sensor data by their mobile devices in the background) in the crowdsourcing process. The captured data is stored and processed in the local database on the mobile device.

2.5.3 Summary

In conclusion, the main goal of this section is to gain a better understanding of the typical functionalities and the design-related aspects during the development and construction of crowdsourcing applications. We have classified crowdsourcing applications into two categories: 1) centralized crowdsourcing application architecture 2) decentralized crowdsourcing application architecture. In the centralized crowdsourcing architecture, all computing is done at a central location or server and computing resources reside at the primary data center. In the decentralized approach, all computing is distributed to each node equally and each node is able to process and propagate their information without relying on a server.

2.6 Support for Mobility

Today's mobile devices, especially smartphones do not only serve as key computing and communication devices, they also come with a rich set of embedded sensors, such as an accelerometer, GPS, gyroscope, digital compass, camera, and microphone. The data obtained from these sensors can also be leveraged. In this regard, the use of smartphone enriches the possibilities in crowdsourcing and it acts as an enabler for large-scale sensing applications.

Studies on mobile phone sensing capabilities, its current state, and its key challenges are given in [Chatzimilioudis et al., 2012; Ganti et al., 2011; Lane et al.,

2010]. Lane et al. [2010] distinguish the multiple scales at which mobile phone sensing is currently being studied by the research community. The three distinct scales are personal, group, and community sensing. Personal sensing applications are designed for a single individual by obtaining data for one's own interest such as the monitoring of movement patterns (e.g., running, walking) of an individual for personal healthcare reasons. Group sensing applications share a common goal, concern, or interest of the people who know and trust each other which creates different privacy and ground truth characteristics. Community sensing involves the participation of a large group of people that cannot easily be measured by a single individual. For example, intelligent transportation systems may require traffic congestion monitoring and air pollution level monitoring. In this study, the authors also discuss two sensing paradigms in a design space relating to the characteristics of the user actively involved in the sensing system - opportunistic and participatory. Participatory sensing requires the users' active involvement in the data collection activity such as taking a picture or reporting a road closure, whereas opportunistic sensing is a fully-automated approach to data collection with minimal or no user involvement such as continuous location sampling in which the action of the user is not explicit.

Ganti et al. [2011] also study mobile sensing applications. They suggest the term mobile crowdsensing to refer to a wide spectrum of community sensing paradigms. Currently, the research on mobile crowdsensing is more likely to move increasingly towards mobile crowdsourcing, that is, using a data collection paradigm that leverages the vast mobile sensor networks; therefore, these terms have been used in the same sense. In their survey, Ganti et al. [2011] focused on mobile crowdsensing at the community scale. The existing crowdsensing applications are classified into three categories: environmental, infrastructure, and social. These categories are based on the type of phenomenon being measured or mapped. Furthermore, they note that mobile crowdsensing applications have unique characteristics that differentiate them from traditional sensor networks such as multimodality sensing and device mobility. Such unique capabilities lead to important issues to be considered in the development of applications. For example, millions of mobile devices carried by people calls for the need for data reuse across different applications. Hence, the authors argue that it is critical to identify common data needs and to support the reuse of sensor data across applications in order to efficiently manage multiple concurrent and reusable applications. In

addition, the dynamic conditions of mobile devices such as variations in their energy levels and communication channels, device owners' preferences and human involvement are important factors in mobile crowdsensing development.

Chatzimilioudis et al. [2012] studied the emerging field of mobile crowdsourcing and presented a mobile crowdsourcing taxonomy. By considering two key features of smartphones, mobile sensor and localization ability, a mobile crowdsourcing classification is developed. The authors claim that a large number of crowdsourcing applications are location-aware, which means such applications can further benefit from adding the spatio-temporal dimension to location data in order to exploit trajectory-related information. The authors also address important issues of smartphones such as intermittent connectivity, energy, and data transfer rates that should be considered during the design phase of mobile crowdsourcing applications. Two architectures – centralized and decentralized mobile crowdsourcing – are mentioned in their work. In the centralized method, the data generated and collected from the crowd are transferred to a server where the answer is operated on, whereas all computations and communications are performed locally on smartphones in a decentralized approach. They argue that the decentralized model performs poorly in terms of energy consumption if it invokes expensive computation tasks on all participants. Based on their study, a comparison between three different localizations (GPS-only, WiFi-only, and 3G-only) was undertaken in three aspects: energy, accuracy, and monetary. GPS-only consumes the highest energy compared to the others, while the least energy is used by WiFi-only. Although 3G-only is found to produce the highest accuracy of data transmission, it requires expenditure on a data package. WiFi-only, however, saves the cost of using data.

2.6.1 Key Challenges in Mobile Crowdsourcing

We review the literature on applications of the mobile crowdsourcing paradigm. There are some key challenges that need to be considered and discussed in any framework in the context of mobile crowdsourcing systems.

2.6.1.1 Inferring Mobile Context

As mobile devices become very common in our daily lives, these devices might store important user context data. Also, users always carry their mobile devices

with them. Thus, user/device context becomes a key element, enabling mobile applications to be user-centric and adjusted to user requirements. Mobile context-aware computing involves the environmental awareness of the system, especially in view of the mobile devices' inherent mobility [Chen and Kotz, 2000]. Examples of context include a geographical mode (e.g., whether the user is on a car, bus, train, or on foot within a circular area), a temporal mode (e.g., in given dates, during given hours), a kinetic mode of humans (e.g., walking, standing, jogging, running), a user profile (e.g., age, gender), a social mode (e.g., in a meeting, on a phone call, watching movie), or occurrence of certain situations (e.g., crowded in the protest, potholes on the road) [Musumba and Nyongesa, 2013]. In the reviewed literature [Afridi, 2011; Alt et al., 2010; Tamilin et al., 2012], context is an important issue and needs to be addressed in mobile crowdsourcing systems and its applications. Afridi [2011] states that context must be precisely specified in the task in order to deliver and execute the right tasks to the right people in the right circumstances.

Several mobile crowdsourcing applications are location-aware crowdsourcing platforms that share and solve tasks in either the requester or the worker's location. It is context dependent and dynamic (time and location) and may have a number of relevant conditions. Tamilin et al. [2012] discuss context-aware mobile crowdsourcing in a way that the context should maximize conditions for user participation by presenting only tasks relevant to the user, with minimal user intervention and minimizing the consumption of resources of mobile devices, especially the battery. According to [Wang et al., 2013], context-aware applications are not used merely by individual users, but a group of users is often involved. The authors propose a context-aware organization model for mobile collaboration which manages the group in mobile environments with solutions including a special weighted majority voting algorithm. Based on their approximate strategy, they are able to represent a way to make decisions and recommendations in collaborative mobile environments. Similarly, Ganti et al. [2011] discuss the current practice of context analytics, which can lead to an explosion of analytics when many crowd sensing applications coexist. This means that they may access the same sensor or involve similar computation in their inference.

Microsoft research has a project exploring the use of spatial context in crowdsourcing, called *spatial crowdsourcing*[33]. They study how to get people to do simple task as specific locations. gMission[34] is also another such platform which

[33]http://research.microsoft.com/en-us/projects/spatialcrowdsourcing/
[34]http://www.gmissionhkust.com/

features with a collection of techniques including geographic sensing, worker detection, and task recommendation to address the needs to get information related to geographic location.

2.6.1.2 Energy Consideration

The energy consumption of mobile devices, such as smartphones, has increasingly become a concern by various sectors, ranging from smartphone manufacturers, mobile developers, to end users. Although battery capacity has been increasing in the past few years, the battery life of mobile devices is not catching up proportionally for a large spectrum of current applications [Bornholt et al., 2012]. Prior work [Balasubramanian et al., 2009; Gupta and Mohapatra, 2007; Xiao et al., 2008] has studied the energy consumption characteristics of mobile network technologies that are in widespread use today. Balasubramanian et al. [2009] conducted a measurement study to quantify the energy consumed by data transferred across 3G, GSM, and WiFi. They found that energy consumption is intimately related to the characteristics of the workload and not just the total transfer size. 3G and GSM incur high tail energy overhead because of lingering in high power states after completing a transfer. However, the transmission energy consumed by WiFi is significantly smaller than both 3G and GSM, especially for large transfer sizes. Gupta et al. [2007] presented a measurement study of the energy consumption of VoIP applications over WiFi-based mobile phones. The authors found that intelligent scanning strategies and aggressive use of the Power Save Mode (PSM) in WiFi can reduce power consumption for VoIP applications. Xiao et al. [2008] measured the energy consumption for Youtube-like video streaming applications in mobile phones using both WiFi and 3G. Their focus is on the energy utilization of various storage strategies and application-level strategies such as delayed-playback and playback after download.

Although the primary usage of mobile phones is reserved for users' regular activities such as making calls and Internet access, today's smartphone is no longer only a communication device. By engaging complex sensing capabilities such as WiFi, Bluetooth, GPS, audio, and video on mobile devices, many new classes of mobile applications have emerged. However, the embedded sensors in mobile devices are major sources of power consumption. Hence, these applications are still confronting limited battery capacities.

There has been a fair amount of work investigating energy optimization on mobile sensing applications in recent years. Wang et al. [2009] designed a framework for using energy efficiently in mobile sensing systems. They use a mobile context-awareness environment to extract more meaningful characteristics of users and surroundings in real time. The hierarchical sensor management strategy has been proposed to recognize user states as well as to detect state transitions. By powering only a minimum set of sensors and using appropriate sensor duty cycles, the system significantly improves device battery life. Shih et al. [2002] explored an event-driven power-saving method to reduce system energy consumption. In their work, the idle power that is the energy consumed in a standby mode is focused. They claim that the device will be powered on only when there is an incoming or outgoing call or when the user needs to use devices for other purposes. Viredaz et al. [2003] conducted surveys on effective methods for saving energy on mobile devices in terms of improving the design and cooperation of system hardware, software as well as multiple sensing sources.

Mobile crowdsourcing applications typically employ mobile sensors to form interactive and participatory sensor networks in order to enable public and professional users to gather, analyze, or even share local knowledge. However, participation in these systems can easily expose mobile users to a significant drain on already limited mobile battery resources. Energy is consumed in all aspects of applications ranging from sensing, processing to data transmission. The processing of mobile crowdsourcing also includes routing workers, propagating tasks among crowd workers or even pulling feedback from crowd responses via mobile devices and integrating to obtain real-time answers. In [Zhuang et al., 2010], the authors present an adaptive scheme for obtaining a phone location by switching from accurate but energy-expensive GPS probing to energy-efficient but less accurate WiFi/cellular localization. Lane et al. [2013] propose a system for collecting mobile sensor data from smartphones called Piggyback Crowdsensing. To perform energy-efficient crowdsourcing of mobile sensor data, data collection, computation and uploading should be occurred as a background process or during everyday smartphone user operations, such as placing calls or using applications. This is because during that time the energy overhead of user participation is low and the phone no longer needs to be woken from an idle sleep state.

2.6.1.3 Task Allocation and Computation

In mobile crowdsourcing, task allocation aims to allocate a specific set of outsourced tasks to a set of mobile users who can potentially finish these tasks more accurately and efficiently. For example, if the task is to translate the Japanese language to English, the mobile users who know Japanese or live in Japan might be preferred to be recruited for the task. Some studies have investigated task allocation. Karger et al. [2014] propose a budget-optimal task allocation algorithm to effectively assign the tasks to appropriate workers. They provide a non-adaptive task allocation scheme and an inference algorithm based on low-rank matrix approximations and belief propagation. Ho et al [2013] explore the task assignment problem by applying online primal-dual techniques. They also propose a near-optimal adaptive assignment algorithm. The result shows that adaptively assigning workers to tasks can lead to more accurate prediction and lower cost when the available workers are diverse. Reddy et al. [2010] claim geographic and temporal availabilities of mobile users are factors that strongly impact task delay and should be considered in worker selection.

With the dynamic conditions of the set of mobile devices, the local analytics performing certain primitive processing of the raw data on the device are needed. The results are initially computed before shipping the processed data back to the server for further processing and consumption. For instance, in inferring human activity applications [Lu et al., 2009a; Miluzzo et al., 2008], local analytics incorporated into phone classifiers perform complex data analysis such as feature extraction, decision tree classifications and data stream mining before transmitting it to the server. Ganti et al. [2011] summarize that there are two benefits for motivating localized computation. First, the overall processing performance consumes less energy and bandwidth than transmitting the raw data. Second, it is able to minimize the amount of processing in the server. Sometimes, the delay in the transmission of raw sensor data on intermittently connected channels or postponing feedback from workers is more time consuming than shipping processed data.

Mobile devices are connected only intermittently when they opportunistically contact each other, which is known as Delay Tolerant Networks (DTNs). DTNs use a store-carry-forward paradigm to allow communication when a path through the network is not reliable due to frequent disconnections. Due to the dynamic nature of moving hosts which may join and leave from the platform at any time, a mobile network topology is likely to change very often. In addition, communication range

might be limited when a mobile user goes outside of a given location, causing unavailability of data (tasks or feedback from crowd) in his mobile device at that location. A routing protocol is needed when the data needs to be transmitted between two nodes. With this issue, the development of crowdsourcing-related mobile applications needs to be considered. In [Moreira and Mendes, 2011], there is an informative survey on opportunistic routing for delay tolerant networks. They classify the routing techniques of opportunistic networks and also evaluate the performance of each method. Recently, Socievole et al. [2013] used social information extracted from multiple social networks to improve message delivery in opportunistic networks. The multiple social networks are based on social metrics which exploit social information extracted from different network layers in which a node forwards packets using a routing metric that combines three measures: node centrality, tie strength and a tie predictor. Chaintreau et al. [2008] used Online Social Networks (OSN) to take advantage of node mobility in an opportunistic manner. According to their proposed model, it shows that the induced topology supports a well-decentralized routing scheme (i.e. greedy routing) and a spatial gossip mechanism when nodes maintain connections with other nodes that they have met in the past.

2.6.1.4 Preserving User Privacy

To be deployed and accepted widely for mobile crowdsourcing, the privacy issue must be considered by both service providers/owner and mobile users/workers. The attractive features and pertinent services of a smartphone enable new kinds of security and privacy intrusions. For example, the research reported in [Young and Quan-Haase, 2009] states that 82% of active Facebook users disclosed personal information such as their birth date, cell phone number, personal address, political and sexual orientation, and partner's name. This vulnerability permits legitimate applications to gather sensitive personal information without the users' full awareness. Moreover, the recording of intimate discussions, taking photographs of private scenes, or tracing users' paths and monitoring the locations they visited are possible intrusions into their privacy. In mobile crowdsourcing, the crowd's tasks could expose the personal interests of the owner and the objective of tasks to the public. Likewise, the feedback from users that is commonly tagged with spatio-temporal information reveals abundant personal information of mobile users, such as location, personal activities, and social relationships. As a result, maintaining the privacy of users is considered as crucial in mobile crowdsourcing.

In general, the user's information is able to be protected from intruders by using cryptography for data transmitting and processing. Liu et al. [2012a] propose an approach to preserve privacy in mobile collaborative learning. The approach allows mobile users to perturb the training data with a distinct perturbation matrix. This scheme then regresses these mathematical relationships between training samples to preserve the accuracy of classification. Chon et al. [2012] proposed a 24-hour time lapse during which users can manually review and delete any data which they deem too sensitive to share. Users also have the ability to block the transmission of data in advance when anticipating activities of a sensitive nature. Anonymity, as an effective solution for privacy preservation, has also been adopted to preserve mobile users' privacy in mobile crowdsensing [Ren et al., 2015]. Kapadia et al. [2008] propose the AnonySense architecture as a means of protecting user privacy when reporting context sensitive information, as it offers protection along multiple layers without manual intervention from the user. It allows using anonymous nodes for the delivery of tasks and the submission of reports. Ren et al. [2015] suggest that anonymous techniques should be carefully developed for information transfer in local-based mobile crowdsourcing since local servers are generally deployed for commercial purposes and are not trusted by mobile users.

2.6.2 Summary

We reviewed the literature on the applications of mobile crowdsourcing and identified the unique characteristics of crowdsourcing for supporting mobility. The key challenges that include inferring mobile context, energy consideration, task allocation and computation, and preserving user privacy have been reviewed in this subsection.

2.7 Discussion

We reviewed the literature which focuses on the key issues in programming with crowdsourcing. In this thesis, crowdsourcing is regarded as human based computation, its main function being to solve complex problems. The focus is on the combined work between a machine and human that brings about seamless data processing. However, how can complex crowdsourcing behaviors be expressed, represented and coded up? And how can crowdsourcing behaviors be integrated

with computational behavior programmatically? We will answer these questions in Chapter 3.

We have found that logic programming has been predominantly useful in research on rapid prototype development. This is because of its ease of use and its well-defined relatively simple semantics. In related work, although the notion of logic programming is mentioned, the implementation and the use of general logic programming in the crowdsourcing paradigm have not been adequately explored. There is a lack of a concept of a general logic programming language integrated into the crowdsourcing paradigm. In addition, how logic programming is able to express queries to the crowd has not been adequately addressed. These gaps are summarized in Research Questions 1 and 2 (in Chapter 1), which will be answered in Chapter 3.

There are several types of architectural approaches used to build crowdsourcing applications as mentioned in the literature. However, how to formulate and implement a framework in which a declarative programming platform is integrated with crowdsourcing in mobile environments have not been sufficiently examined. This issue is addressed in Research Question 3 and it is presented in Chapter 4.

When we have a tool to represent the queries posed for the crowd and to combine conventional machine computation and the power of the crowd, how can we best use this tool to benefit mobile users? This question is discussed in Chapter 5.

Energy consumption behavior is important for resource-limited mobile devices. In recent years, although there has been several studies investigating energy optimization on mobile sensing applications, there has been inadequate exploration of the energy characteristics of mobile crowdsourcing applications, especially for energy-limited mobile devices. This issue is discussed in Chapters 6 and 7.

2.8 Summary of Chapter

The emergence of the crowdsourcing paradigm has brought a dramatic change in the landscape of solving complex problems. Crowdsourcing refers to a powerful approach to address problems and finding relevant solutions that are hard for computers to do but trivial for humans. Integrating human intelligence with machine computation can increase capabilities to complete complex tasks in a way where humans with their abilities are integrated with machine computation.

We have given an extensive survey of current crowdsourcing research in this chapter. Highlighting the programming platforms for crowdsourcing, we have presented a taxonomy of the issues found in this area and several dimensions in which these issues have been tackled, focusing on programming languages and APIs, workflow algorithms, applicability, architectures and the support for mobility.

Recently, crowdsourcing has steadily moved across many disciplines in both scientific and industrial sectors. It has developed in new contexts such as new business ideas and solutions to social problems and consequently, there are new products and services being launched every day that are leveraging the power of the crowd to find solutions to problems. As we noted, there will be a general trend toward the increasing use of crowdsourcing platforms that use mobile technology. Mobile devices increasingly bridge the digital gap in the community because more people access the Internet via mobile devices than via laptops or desktop computers. Most mobile devices are also enriched with a set of embedded sensors and context data. In the near future, crowdsourcing applications will be developed to allow rich mobile contributions as well as mobile ad-hoc contributions, so that crowdsourced data available as services will be of new forms that could be provided by users or generated automatically by mobile sensors.

In the next chapter, we propose a declarative programming paradigm integrated with crowdsourcing for mobile environments. This chapter will discuss a simple extension of Prolog, which we call *LogicCrowd* that automatically leverages human knowledge through the crowdsourcing paradigm. We also describe the concept and the relevant key ideas of logic programming and the *LogicCrowd* framework.

Chapter 3

A Declarative Programming Platform for Mobile Crowdsourcing

As discussed in Chapter 2, we reviewed a range of concepts, frameworks, techniques, key aspects, and challenges of the crowdsourcing paradigm based on a survey of significant research. In this chapter, we introduce a declarative programming paradigm integrated with crowdsourcing for mobile environments. The use of logic programming is aimed at providing expressive and declarative semantics, a higher level of abstraction, and allowing query and manipulation of knowledge and reasoning. We designed *LogicCrowd*, a declarative crowdsourcing platform for mobile applications, which combines conventional logic-based machine computation and the power of the crowd from social networking, via centralized or mobile peer-to-peer networks.

This chapter is organized as follows. Section 3.1 introduces the concept of declarative programming for crowdsourcing. Section 3.2 provides an overview of the relevant key ideas of logic programming and *LogicCrowd*. Section 3.3 details the *LogicCrowd* formalism, comprising the notion of crowdsourcing programs and extensions of Prolog including the basic meta-interpreter for *LogicCrowd*. Adding to basic *LogicCrowd*, two extensions are then described: crowd unification and peer-to-peer querying. Section 3.4 introduces the concept of crowd unification. Section 3.5 details the fundamentals of mobile peer-to-peer networks in the crowdsourcing paradigm. We summarize the chapter in Section 3.6.

3.1 Declarative Programming

As explained in Chapter 2, the declarative programming paradigm refers to a style of programming that describes what is to be computed, not necessarily how it is to be computed. Thus, declarative programming languages are written in a logical form enabling a reader of the program to grasp what is being solved.

Each goal/query is evaluated locally in a closed set of knowledge base. If Prolog is able to match this goal in its knowledge base, the goal will be satisfied. On the other hand, if Prolog can not do so, the evaluation of the goal will be false or non-existent. This issue is caused by the limited set of facts and rules in the local database. It also notes that this knowledge base is extremely literal. They expect the data has been properly cleaned and validated before entry and do not naively tolerate inconsistencies in the data or queries.

Moreover, one of the key features of logic-based programming is the unification of the terms in the program. It represents the mechanism of pattern matching over logic values. In order to unify two terms, if they are atoms, they must have the same term/structure or if they contain variables, these variables must be uniformly instantiated with terms in such a way that the resulting terms are equal [Sterling and Shapiro, 1994].

Unfortunately, this matching scheme is often not adequate for matching certain structures in complex queries as in the following example. If the `parent/2` predicate contained photos rather than names as follows:

```
parent(<photo1.jpg>,<photo2.jpg>).
parent(<photo3.jpg>,<photo4.jpg>).

grandparent(A,C):-parent(A,B),parent(B,C).
```

It has a query:

```
?-grandparent(<photoA.jpg>,<photoB.jpg>).
```

When running the above query, the Prolog interpreter responds to the query about the facts and the rules represented in its database. In this case, the system must determine if the sub goals `parent(<photoA.jpg>,Y)` and `parent(X,<photoB.jpg>)` unify, for some `X` and `Y`. Therefore, if `X` and `Y` are exactly the same photo, it can prove that this query is true. However, it is generally hard for the query engine to "unify" the two images `X` and `Y`, even if they are images of the same person. As a result, logic unification needs a more sophisticated scheme that can overcome this constraint; that is, human processing is involved. This is one of the justifications for the need to have people to provide answers, especially ones accessible to the public. A person can say that `<photoA.jpg>` and `<photoB.jpg>`

are of the same person, and so they could, in principle, unify, while a machine would have difficulties comparing two photos, in general.

Recently, there has been substantial interest in harnessing crowdsourcing for solving problems that are impossible or too expensive to answer correctly using computers. The emergence of the crowdsourcing paradigm has dramatically brought a change in the landscape of solving complex problems. Crowdsourcing refers to a powerful approach to addressing problems and finding relevant solutions that are hard for computers to do but trivial for humans. To solve such queries in a database, human input is required to provide information that is false or non-existent in a local database to perform computationally difficult functions such as matching, ranking or aggregating results based on "fuzzy" criteria. Remarkably, crowdsourcing allows programmers to incorporate human computation as a building block in algorithms that cannot be fully automated by computers themselves, such as text analysis and verification or image comparison and recognition.

There has been significant work which applies the crowdsourcing paradigm to address such issues. As mentioned in Chapter 2, proposed systems such as Deco [Parameswaran et al., 2012], CrowdDB [Franklin et al., 2011], TurkDB [Parameswaran and Polyzotis, 2011] and Qurk [Marcus et al., 2011] apply a declarative approach to designing small extensions to SQL so that humans can incorporate the process of SQL queries. These studies have mostly implemented crowdsourcing on top of Amazon's Mechanical Turk (MTurk). Moreover, proposed systems such as Turkit [Little et al., 2010b] and HProc [Heymann and Garcia-Molina, 2011], Jabberwocky [Ahmad et al., 2011] are procedural programming libraries designed to optimize worker productivity and tasks, which enable programmers to interface with MTurk.

However, our work proposes a declarative programming paradigm for leveraging the knowledge of people through crowdsourcing. *LogicCrowd* is introduced as an innovative approach that integrates logic programming into crowdsourcing middleware in order to provide a declarative programming platform for applications. In *LogicCrowd*, we propose an alternative declarative style of programming for crowdsourcing which leverages on Prolog and is more expressive than simple crowd SQL queries. The use of logic programming is aimed at providing a higher level of abstraction and specialized language features, and also allows the querying and manipulation of knowledge and reasoning. It also provides ease of programming and maintenance. This approach comprises an interpreted Prolog-based declarative language (including facts and rules) and crowdsourcing middleware;

both of which will be presented in the next section.

In our framework, *LogicCrowd* is a Turning-complete full logic-programming approach that allows programmers to create their rules via an event-driven approach. *LogicCrowd* is able to interface not only with the existing crowdsourcing market platform (MTurk) but is also open to social media networks (Facebook) and peer-to-peer networks to use as crowdsourced data in logic programs.

Moreover, the use of smartphones today enriches the possibilities in crowdsourcing and it acts as an enabler for large-scale sensing applications. Some works including Txtagle [Eagle, 2009], MobileWorks [Narula et al., 2011], UbiAsk [Liu et al., 2011], CrowdITS [Ali et al., 2012] and Smart Mob [Rheingold, 2002] have developed crowdsourcing applications enhanced with a range of different sensors such as cameras, GPS, communication signals, accelerometer and so on. However, in our platform, we have developed a novel mobile application that enables declarative programming with mobile crowdsourcing through the use of social networks and peer-to-peer networks. Thus, sensor data such GPS locations can also be incorporated to scope the queries.

LogicCrowd, a declarative crowdsourcing platform, is built on top of existing social networking infrastructure and online crowdsourcing market platforms to bring the crowdsourcing model into the mobile context. An indicative list of tasks/programs which *LogicCrowd* can support is as follows:

- Given datasets including sensing data or interesting items, provide a recommendation to the users based on these datasets using the crowd to determine decisions.
- Given a list of products, rank them by "quality" by asking for experts or users who have used the products.
- Given two media objects, unify them by asking users whether these objects are the same or not.
- Ask the crowd's opinion on issues; identify/express sentiments such as like or dislike, and other feedback.
- Find a destination point or boundary area such as finding parking lots, looking for a good seat in a big football stadium, escaping from disastrous situations like a tsunami or bush fire as fast as possible.

We provide some examples of the tasks listed above, together with a presentation of the relevant prototype in Chapter 4. Due to the fact that *LogicCrowd* is programmed based on Prolog, which is a popular logic programming language, there are opportunities for applications that include reasoning, above and beyond

the example tasks. Other logic programming languages can be employed, but we start with Prolog due to its relatively simple semantics and popularity. Since mobile users/developers can fix/decide on facts and rules (users' independence), the application becomes configurable and reprogrammable. In the next section, we provide a brief outline of the *LogicCrowd* formalism, comprising the notion of crowdsourcing programs and an extension of Prolog. At this stage, *LogicCrowd* is designed for users or mobile developers who have a basic background in Prolog, though syntactic sugar and UI forms can be used.

3.2 *LogicCrowd* Overview

The key idea of logic programming is to use a computer to draw conclusions from declarative descriptions. Such descriptions are called the logic programs which consist of a set of sentences in a logical form, expressing axioms (facts) and a goal statement. The rules of inference are applied to determine whether the axioms are sufficient to ensure the truth of the goal statement. The execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms [Sterling and Shapiro, 1994]. In other words, the programmer specifies the relationships among the data values and then poses queries to the execution environment in order to examine whether certain relationships hold. Hence, a logic program defines a base of knowledge from which implicit knowledge can be derived through explicit facts and rules.

Returning to our earlier example in Prolog, the following sentences are declarative sentences of natural language that describe facts and rules.

```
Tom is John's child.                                                    (i)
Ann is Tom's child.                                                    (ii)
John is Mark's child.                                                 (iii)
Alice is John's child.                                                 (iv)
The grandchild of a person is a child of a child of this person..      (v)
```

These sentences are able to be formalized directly into the syntax of Prolog as below. With sentences (i)-(iv), the syntax atomic formulas describing facts are introduced in (1)-(3). The final sentence (v) is the rule of inference which can be formalized into the syntax of Prolog in (5).

```
child(tom,john).                                                        (1)
child(ann,tom).                                                         (2)
child(john,mark).                                                       (3)
child(alice,john).                                                      (4)
grandchild(X,Y):- child(X,Z), child(Z,Y).                               (5)
```

Such clauses refer to individuals in some situations and to relations between those individuals. In Prolog, more precisely, the symbols for denoting individuals (e.g., `tom`, `john`, and `ann`) will be called constants whereas the symbol for denoting relations (e.g., `child` and `grandchild`) are called predicate symbols. Also, the symbols for denoting variables (e.g., `X`, `Y`, and `Z`) are strings of characters beginning with an upper-case letter or an underscore. In (5), the rule is written in a form of clauses and read declaratively as logical implications "if `child(X,Z)` and `child(Z,Y)` are true then `grandchild(X,Y)`". The symbol ":-" is to be read as "if" and it looks like the leftward-pointing arrow "⇐" that is sometimes used in ordinary logic. The comma that separates the formulas `child(X,Z)` and `child(Z,Y)` is to be read as "and". The `grandchild(X,Y)` is called the head of the rule and `child(X,Z),child(Z,Y)` is called the body. Hence, facts are rules that have no body.

In *LogicCrowd*, the notions of logic programming and the crowdsourcing paradigm are combined in order to cope with the limitations emerging from traditional machine computation which operates upon its a closed database of a set of rules and facts. *LogicCrowd* makes use of the potential of Prolog which is able to provide a mechanism that recursively extracts the sets of data values implicit in the facts and rules of a program. Additionally, *LogicCrowd* exploits the ability of crowdsourcing to solve problems that are complicated and thus cannot be simply fixed with by the machine itself. *LogicCrowd*, a declarative crowdsourcing platform, has been designed to be able to work with crowdsourced data of various platforms including working through crowdsourcing market platforms (i.e., MTurk), social media network (i.e., Facebook), or even P2P networks (via Bluetooth and Wi-Fi direct). *LogicCrowd* extends the capabilities of Prolog in three dimensions as follows:

1) It allows some predicates which open queries to the public, i.e., a possibility to be answered by the crowd; rather than a closed set of facts.
2) New operators encoded by extending the basic Prolog meta-interpreter have been specifically designed to increase the capabilities of working with the crowd via various platforms: crowdsourcing market platforms, social networks, and P2P networks, as mentioned.
3) It provides a choice of executing queries, either synchronously or asynchronously, resulting in greater flexibility of the computation process conducted when engaging the crowd.

LogicCrowd also has an extended unification scheme in the logic programming paradigm, essentially enabling queries to access the crowd in an on-demand fashion

for the purpose of unifying arguments in predicates when traditional unification might not work, e.g., in comparing media objects. The notion of unification is at the heart of the computation model of logic programs, and implements pattern matching. Pairwise matching with the sequence pattern process establishes equivalence between logic terms. For example, the terms `anna` and `anna` unify because they are the same atom. Similarly, the terms `woman(anna)` and `woman(anna)` unify because they are the same complex term. In contrast, the terms `woman(anna)` and `woman(taylor)` do not unify since they are logically different. Consider a query which asks to unify two photos `photo('a.jpg')` and `photo('b.jpg')`. With a matching in unification, the logic program would say that these terms cannot be unified as they are not the same term. In fact, `'a.jpg'` and `'b.jpg'` could be the same photo, but if this fact is not represented in a logic program explicitly, normal unification in Prolog would treat them as different since the names of the files are different. But we introduce *crowd unification* as a convenient mechanism to harness the power of people in order to solve such unification problems. In crowd unification, we would ask the crowd if the two photos are the same, and if the crowd say so, then the two photos would unify. This means we adopt a broader view of unification beyond the traditional view, which we term crowd-unification, as explained in Section 3.4.

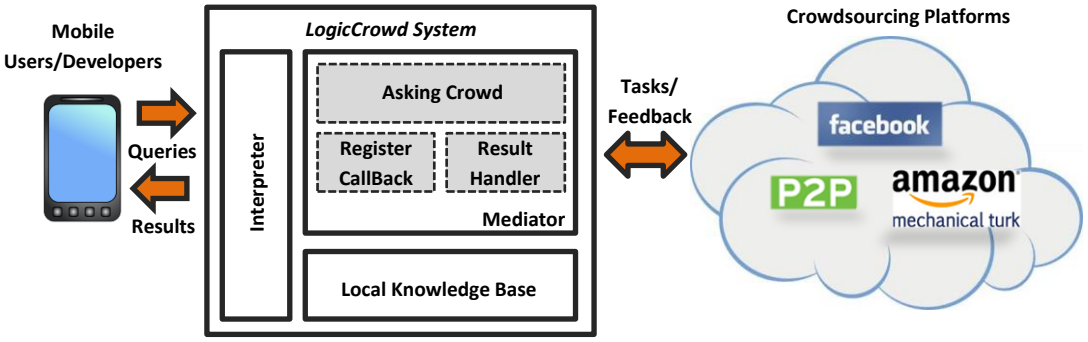


Figure 3.1: An overview of the functions of the *LogicCrowd* system

Figure 3.1 presents a basic overview of the functions of the *LogicCrowd* system in executing a query/goal. A query/goal will be evaluated by a *LogicCrowd* Interpreter using the local knowledge base. When the query/goal is required to access or connect to the crowd, this query is sent to a mediator that communicates between the *LogicCrowd* interpreter and crowdsourcing platforms. In this sense, the mediator works on transforming the query into a format understandable to crowdsourcing platforms. Additionally, the mediator acts in the role of a handler to deal with the return of feedback from the crowd. The *LogicCrowd* architecture

is described in more detail in Chapter 4.

LogicCrowd is designed for users or developers who have a basic background in Prolog through syntactic sugar and UI forms. The users/developers are able to exploit *LogicCrowd* by passing the queries to the public to solve complex problems. To make the *LogicCrowd* system more flexible, users/developers can write a small program in the Prolog style to evaluate complex goals. For a better understanding of how *LogicCrowd* works, an example of a program which has been loaded in the *LogicCrowd* system is shown below.

```
?-phone-number?(Number)#[question('What is Mike's phone number?'),
                askto([facebook]), expiry('0,30,0')].
```

The example above shows a query sent to the crowd. This query is to ask friends on Facebook for Mike's phone number. The question and options predicates are in the crowd's conditions acting as parameters for sending the query to the open crowd. In our work, we define the relevant crowd's conditions based on the common questions: what to ask, whom to answer, what the location is (i.e., providing a scope), and when to receive responses. The next program contains a rule using a crowd predicate in a *LogicCrowd* program to recommend well-known nearby restaurants.

```
recommend(Restaurant):-
          thai(R),
          nice?(Restaurant)#[asktype('choice'),
                question('Which restaurant do you recommend?'),
                options(R), askto([mturk,facebook]),
                locatedin('Bangkok'), expiry('2,0,0')].
```

Declaratively, the rule says recommend a restaurant if (denoted by ":-") the restaurant is an option returned from the crowd (using MTurk and Facebook), is Thai and is located in Bangkok. Procedurally, this rule can be interpreted as a Prolog rule in which the user decides on the goal to search for Thai restaurants. The first sub-goal, represented by `thai(R)`, will start by searching for Thai restaurants in the existing (local on-mobile) knowledge base via the process of machine computation. The result of the search will be taken as input to the crowd query. The input will next be sent to the crowd in the second sub-goal in order to find out which Thai restaurants in a list would be recommended by the crowd. In the crowd condition, the parameter pertaining to the question ('Which restaurant do you recommend?'), options, target addressees, and their current locations, as well as the specific expiry time for returning the feedback, are made explicit to the crowd.

It is apparent that *LogicCrowd* empowers people by combining human input

with machine computation for processing queries that cannot be simply answered by a normal knowledge base system. By allowing users/developer to query the crowd about facts and rules, *LogicCrowd* is considered as a flexible tool which can be applied for use in many real situations. These applications become configurable and reprogrammable. Apart from this, *LogicCrowd* operates on the mobile platform and works through the mobility-support system. It can also be useful in peer-to-peer computing for querying and multicasting tasks shared over peer networks (as Section 3.4 explains).

3.3 *LogicCrowd* in Detail

This section details the *LogicCrowd* formalism, comprising the notion of the crowd-sourcing program and its design. It has three important extensions to Prolog. First, it allows predicates to query the public, rather than a closed set of facts. Second, specific operators encoded by extending the basic Prolog meta-interpreter are designed for querying the crowd. For the second extension, we give the details in Chapter 4, Section 4.2. Third, it provides two methods including synchronous and asynchronous calls for flexible crowd execution of queries.

3.3.1 *LogicCrowd* Programs

A *LogicCrowd* program allows predicates to query the public (crowd) rather than a closed set of facts. Queries to underlying crowds are abstracted as predicates, which we term *crowd predicates* of the form:

```
<crowd_KW>?(<crowd_answer>)#[<crowd_conditions>].
```

The request for a task for crowd computing is identified by a crowd keyword (`crowd_KW`). The `crowd_answer` is an output from the crowd for each task represented by a variable and `crowd_conditions` are inputs or conditions when asking the crowd. The crowd predicate has its own operators "`?`" and "`#`" referring to crowd identity and crowd conditions, respectively. For example, the following shows a query to the crowd in order to ask for a nice Thai restaurant; where users choose from three options:

```
nice?(Answer)#[question('Which restaurant do you recommend?'),
      options(['Thai Sontaya','Baan Thai','Le Bangkok'])].
```

The question and options predicates are in the crowd conditions acting as parameters for sending queries to the open crowd. In our work, we define the relevant crowd conditions based on common questions: what to ask, whom to answer, what

the location is (i.e., providing scope); and when to receive the response. Below is another example using a crowd predicate in a *LogicCrowd* program to represent a person/user's query about a place, sent to the crowd, where the question is to ask where (or which place) is `a.jpg` a photo of.

```
place?(Answer)#[asktype('photo'),
               question('Where is it?'),
               picture('a.jpg')].
```

The question and picture predicates are in the crowd conditions acting as parameters for sending queries to the open crowd. In our work, we define the relevant crowd conditions based on common question templates: what to ask, whom to answer, what the location is (i.e., providing spatial scope), and when to receive the response. Below is a rule using a crowd predicate in a *LogicCrowd* program to ask about incomplete data.

```
thisPlaceIs(Answer):-
          place?(Answer)#[asktype('photo'),
               question('Where is it?'),picture('a.jpg'),
               askto([facebook,bluetooth]),expiry('0,30,0')].
```

This rule can be interpreted as a Prolog rule in which the user decides on the goal to ask for the name of a place given a photo. The crowd predicate is represented by `place` (crowd keyword) and provides the variable `Answer` to return the feedback from the crowd. The variable `Answer` will be instantiated with the answer from the crowd. For example, if `a.jpg` is a photo of the statue of Liberty, `Answer` could be instantiated with value "New York", i.e. Answer = "New York" which is an answer substitution. In the crowd conditions, the parameters pertaining to the question type, the question (`'Where is it?'`), the link to the picture(s), target addressees as well as the specific expiry time for returning the results are made explicit. In contrast to our previous work, here, we develop more options to ask the crowd: 1) sending the queries via a social media network, i.e. Facebook, and 2) sending queries to a peer-to-peer network, i.e. via Bluetooth. Apart from recommending a place, we can have rules for ranking the commercial products to serve business demands as follows:

```
bestHandbag(Bestbag):-
          brand(Handbag),
          besthandbag?(Popbag)#[asktype('choice'),
               question('What is the most popular handbag?'),
               options(Handbag),askto([facebook]),
               expiry('1,0,0')],
          quicksort(Popbag,'@>',Bestbag).
```

The rule again shows the application of both machine and human computation in one single task. It starts with `brand(Handbag)` searching for brands (of the

handbags) in the knowledge base. Once the search succeeds and the relevant result comes up, such a result will be directly sent to the crowd, which will then reply to the question ('`What is the most popular handbag?`'). In this case, an expiry time is set for an hour. The system will return the feedback from the crowd according to the time set by the users. After this, the system will rank the handbags (derived from crowd) in order of their popularity.

3.3.2 An Operational Semantics for *LogicCrowd* Programs

There are various approaches to giving meanings to programming languages. An operational semantics is one of the methods for specifying the semantics of programming languages. Operational semantics is defined by specifying the behavior of programs during execution [Plotkin, 1981]. We outline an operational semantics for the language of pure Prolog augmented by an oracle representing the crowd. Such a language consists of clauses of the form:

$$\mathcal{A} : - \mathcal{G}$$

where $\mathcal{G}$ is defined by

$$\mathcal{G} ::= \mathcal{A} \mid \mathcal{D} \mid (\mathcal{G}, \mathcal{G})$$

where $\mathcal{A}$ is an atomic goal and $\mathcal{D}$ is a crowd predicate

We describe the *LogicCrowd* operational semantics in the style of structural operational semantics. Structural operational semantics represents computation by means of deductive systems that run the abstract machine as a system of logical inferences [Plotkin, 1981]. In the structural operational semantics, the semantic descriptions are given by inference rules that consist of 1) a set of premises, and 2) a conclusion. The general form of an inference rules has constructs as follows: (1) a set of premises (e.g., $premise_i$ by i is an integer from 1 to n) is listed above a horizontal line; and (2) the conclusion is below a series of premises.

$$\frac{premise_1 \ premise_2 \ ... \ premise_n}{conclusion}$$

To precisely describe the operational behaviour of *LogicCrowd*, we first define the following relation ω, $P \vdash_{\theta,\pi} G$ to hold when the goal G succeeded with Prolog program P with the results θ (the substitutions) and π (the crowd support or crowd responses) in the world ω which is the set of answers from responders to crowd queries.

The rules for the *LogicCrowd* program are as follows in the format $[Label]\frac{premise}{conclusion}$ where the conclusion holds whenever the premises hold. These rules extend those given for pure Prolog, described in Chapter 2. We assume that

the crowd goals in the [*atom*] rule might be used in the head of the rule via the crowd operator as mentioned in Section 3.3.1.

$$[true] \quad \frac{}{\omega,P \vdash_{\epsilon,\emptyset} true}$$

$$[atom] \quad \frac{\omega,\, P \vdash_{\epsilon,\emptyset} H{:-}G \wedge \theta{=}mgu_{\pi_1}(A,H) \wedge \omega,\, P \vdash_{\gamma,\pi_2} G\theta}{\omega,\, P \vdash_{\theta\gamma,\pi_1\cup\pi_2} A}$$

$$[conjunction] \quad \frac{\omega,\, P \vdash_{\theta,\pi_1} G_1 \wedge \omega,\, P \vdash_{\gamma,\pi_2} G_2\theta}{\omega,P \vdash_{\theta\gamma,\pi_1\cup\pi_2} G_1,G_2}$$

From the above rules, the formal operational description of the standard Prolog is represented in the form $H :- G_1, G_2, ..., G_n$, where G_i is a subgoal in program P and "," is the AND operator. However, we extend the standard evaluation to involve the crowd or the world (denoted by ω) to which some queries/goals have been propagated in order to solve certain goals. After evaluation by the crowd, the result (in the form of crowd support denoted by π) is returned together with θ, which is the result of standard substitution in Prolog. The rule [*true*] is an axiom where the number of premises is zero denoting that the goal "true" is always true independent of the premise. Note that ϵ denotes the empty substitution and $\emptyset$ the empty set (zero crowd support).

In the second rule, the rule [*atom*] refers to the evaluation of a goal in Prolog. The key process in the execution of the Prolog program is a matching operation. When Prolog is attempting to satisfy a goal, clauses are selected if their head unifies with the goal. Prolog's pattern matching is based on the concept of a substitution. Thus, a matching operation produces a substitution and the substitution of the logical variable by the atom is known as a unifier. Based on this rule, we can compute the most general unifier (*mgu*) that comprises the substitutions θ and crowd support π, denoted by $\theta = mgu_{\pi}(term1, term2)$.

In the rule [*atom*], we can write another version if we want to constrain the crowd goal to be only used in the body of the rule via the crowd predicate. For simplicity without loss of generality, this is the version we use as shown below.

$$[atom] \quad \frac{\omega,\, P \vdash_{\epsilon,\emptyset} H{:-}G \wedge \theta{=}mgu(A,H) \wedge \omega,\, P \vdash_{\gamma,\pi} G\theta}{\omega,\, P \vdash_{\theta\gamma,\pi} A}$$

The rule [*conjunction*] represents the operation of the conjunctive goals. In Prolog, we are allowed to write rule bodies that contain conjunctions. Notably, in case there are conjunctive rules in the body, the head of the rule is considered true if and only if all terms/rules in the body are also true. Hence, the results collected from crowd (π) would be returned as combining the element of crowd

support $\pi_i ... \pi_n$ together with θ and γ, which are the result of standard substitution in Prolog.

Moreover, we define the wide range of crowd support/response in the formal description languages. As mentioned, the crowd support is a set of possible answers that are received from the crowd. The next section describes a range of operators based on counting responses from the crowd.

3.3.2.1 Crowd Response Counting Operators

We define a range of operators that are variants of the crowd predicate. Viewing the world ω as a function that returns a set of answers to queries, for a query q, the crowd support π_q for q (we write π in cases where q is clear from the context) can be defined as a subset of crowd answers/responses to query q denoted by

$$\pi_q = \omega(q) \subseteq (\{q\} \times U \times V)$$

where U denotes a set of users to which queries are posed (i.e., the crowd members) and V denotes the set of possible answers.

Crowd responses can be obtained for arbitrary queries where the set of possible answers can be domain specific, and instantiated in variables and used within *LogicCrowd* programs. We can define a variety of crowd answers extracted from crowd responses as follows.

$$[collect] \qquad \frac{\omega(q){=}\pi \wedge l = [v|(q,u,v) \in \pi]}{\omega,\, P \vdash_{\{L/l\},\pi} crowd\_ans(q,L)}$$

$$[most\_frequent] \quad \frac{\omega(q){=}\pi \wedge l = [v|(v,c) \in f(\pi) \wedge c = maxCnt(\pi)]}{\omega,\, P \vdash_{\{L/l\},\pi} crowd\_ans(q,L)}$$

$$[determine] \qquad \frac{\omega(q){=}\pi \wedge l = [v|(v,c) \in f(\pi) \wedge c \geq q.N\,]}{\omega,\, P \vdash_{\{L/l\},\pi} crowd\_ans(q,L)}$$

$$[topK] \qquad \frac{\omega(q){=}\pi \;\wedge\; l{=}topK(q.K,sortbyCnt(f(\pi)))}{\omega, P \vdash_{\{L/l\},\pi} crowd\_ans(q,L)}$$

Note that the first rule [*collect*] represents all possible values of the crowd's answers in response to query q. The answers with duplicate values eliminated are contained in the list l, as the result, which instantiates the variable L, returning the answer substitution $\{L/l\}$ with crowd support π. To find the crowd answers in the second rule $[most\_frequent]$, it must be the answer most frequently chosen by or most popular among the crowd. $f(\pi)$ denotes a set of pairs (v, c) where $v \in V$ and c is a non-negative integer denoting the number of times the value v has been returned as a response in π. The function $maxCnt(\pi)$ computes the highest count of answers, i.e. the answer receiving the top votes in the set of π. The next rule

[*determine*] accepts the list of crowd responses with counts c greater than or equal to $q.N$, a positive number defined as part of query q ($q.N$ denotes N is part of or a property of q). The last rule [*topK*] selects the crowd responses from function $topK(q.K, l')$. In this list l', all answers are ranked from the highest to the lowest number of crowd votes/responses by the function $sortbyCnt(f(\pi))$ with $q.K$ being a positive number defined in the query. So this operator returns the *topK* answers by vote.

3.3.2.2 Crowd Computing Cost

After describing the formal operational semantics especially the crowd support (π) for general crowd querying, it is clear that crowdsourcing can be integrated systematically into logic programs in order to solve complex problems which are very difficult to do by machine alone. However, existing crowdsourcing systems might suffer from low user participation rate, response time and data quality. As a consequence, motivational incentives should be considered in order to potentially attract user attention and encourage high quality data from crowdsourcing participants. It is possible to model incentives in the operational description, for example, as follows (where $C(q)$ denotes a crowd goal with a query q):

$$[true] \quad \frac{}{\omega, P \vdash_{\epsilon,\emptyset,0} true}$$

$$[atom] \quad \frac{\omega,\, P \vdash_{\epsilon,\emptyset,0} H{:-}G \wedge \theta{=}mgu(A,H) \wedge \omega,\, P \vdash_{\gamma,\pi,c} G\theta}{\omega,\, P \vdash_{\theta\gamma,\pi,c} A}$$

$$[conjunction] \quad \frac{\omega,\, P \vdash_{\theta,\pi_1,c_1} G_1 \wedge \omega,\, P \vdash_{\gamma,\pi_2,c_2} G_2\theta}{\omega, P \vdash_{\theta\gamma,\pi_1\cup\pi_2,c_1+c_2} G_1, G_2}$$

$$[crowd] \quad \frac{\omega(q){=}\pi, \wedge\, (\theta{=}comp()) \wedge c = |\pi| \times reward\_rate}{\omega, P \vdash_{\theta,\gamma,\pi,c} C(q)}$$

The fourth rule is a simple general meta-rule representing the rules above for crowd query/answering. The function comp() is simply a placeholder function that instantiates θ with the query answer in the case of crowd query/answering as shown in the rules above. Note that the first three rules are the formal operational description of the standard Prolog evaluation that has been extended to carry accumulated costs.

In the above [*crowd*] rule, c denotes the total cost of financial rewards for each query/task. The cost is returned as the result from the evaluation of the crowd query. For the [*crowd*] rule, it represents the total cost c that is spent for query q. Also, the total cost of a crowd query is computed from the total number of responses from the crowd multiplied by a reward_rate as set by the

query issuer. Note that the cost returned is the cost of a successful execution - costs incurred in backtracking from failed branches in the evaluation are not counted in the above rules - these rules can be modified to update a store keeping track of all costs incurred even in failed branches of the evaluation. If crowd responses are simulated, the rule can be executed breadth-first-search wise as a means to search for the cheapest goal evaluation path or executed depth-first-search wise to estimate the cost of a goal evaluation. The operational rules above are implemented in Prolog as a meta-interpreter for *LogicCrowd*, as described in Chapter 4.

3.3.3 *LogicCrowd* Execution Models

One of the most important issues about crowdsourcing is how to manage the answers returned by the crowd. As there may be a delay for the crowd to provide answers via social media networking (such as Facebook), *LogicCrowd* has been designed to tackle this delay. First, the `registercallback` predicate is designed for registering and handling the returned results from the crowd by using Mediator. This predicate has been explained more detail in Section 4.2.1, Chapter 4. Second, two methods are used to execute the rules: synchronous and asynchronous executions.

In the synchronous operation, we implement *LogicCrowd* according to the standard Prolog program execution model, which runs sequentially without any parallel extensions. When *LogicCrowd* is executing a crowd predicate, the process or the evaluation will be suspended until the system receives the returned feedback from the crowd. We support this mechanism via a small extension to the crowd predicate as the following form:

```
<crowd_KW>?(<crowd_answer>)#[syn,crowd_conditions].
```

The query was issued in the synchronous mode when we put the atom "syn" in the crowd's conditions.

For example, the list of restaurants in a particular area will be recommended through this goal comprising of two sub-goals that will be executed sequentially. After the first sub-goal querying restaurants in the local database, the result will be passed to the second sub-goal (crowd predicate). After receiving the list of restaurants, the crowd predicate will deliver this list and all conditions to ask for suggestions from the crowd. The system will then wait for a while before returning the results from the crowd.

```
recommend:-
          findall(X,(thai(X),melbourne(X)),R),
          nice?(_)#[syn,
                question('Which restaurant do you recommend?'),
                asktype('choice'),options(R),askto([friend]),
                expiry('2,0,0')].
```

Another more sophisticated example below further illustrates the process of synchronous execution. We add three more sub-goals that extend the above example. The aim of the main goal is to show the location of the best restaurant, that is the one chosen as the most popular restaurant by the crowd. The `selectOne/2` predicate will not be invoked until the results from the crowd are returned, i.e. the crowd predicate suspends until the results come back or until expiry (in the synchronous mode); in this case, it may take up to two hours so that results are obtained as many as possible till expiry time (set at two hours). After that, `getLocation/2` and `show/1` predicates will be executed respectively.

```
recommendOne:-
          findall(X,(thai(X),melbourne(X)),R),
          nice?(Restaurant)#[syn,
                question('Which restaurant do you recommend?'),
                asktype('choice'),options(R),askto([friend]),
                expiry('2,0,0')],
          selectOne(Restaurant,BestOne),
          getLocation(BestOne,Loc),show(Loc).
```

In contrast, the asynchronous operation exploits the multi-threading capability available since *LogicCrowd* is built on top of tuProlog which integrates seamlessly with Java/Android. As a result, a new thread is created for each such asynchronous crowd predicate evaluation, to run independently. For the asynchronous mode, we have a crowd predicate of the form:

```
<crowd_KW>?(<crowd_answer>)#[asyn,crowd_conditions].
```

Asynchronous execution takes place when we specify the atom "asyn" in the crowd's conditions. In contrast to synchronous processing, asynchronous operation permits other processes to continue before its execution has finished. In our case, when the asynchronous method is used, the evaluation of the crowd predicate is put into the background. A new thread is then created and the next sub-goal can be executed without blocking the previous crowd predicate.

The rule below illustrates an asynchronous version of the recommendation of well-known restaurants. This example, extending the above synchronous example, appends `doSomethingElse/2` predicate at the end of the goal. This predicate is executed immediately without waiting for the previous crowd predicate to complete, that is, regardless of whether the completed results have been returned by

the crowd. However, the developers need to provide a rule that will be invoked when results come back from the crowd, namely, `handle_crowd_answer/3`, in order to receive the results and do further processing with the crowd's answer. A predicate of this exact name must be used since this will be recognized by the system. As found in the example above, `handle_crowd_answer/3` is written to show the location of the best restaurant which receives the top score from the crowd's votes for the query.

```
recommend:-
          findall(X,(thai(X),melbourne(X)),R),
          nice?(_)#[asyn,
                question('Which restaurant do you recommend?'),
                asktype('choice'),options(R),
                askto([friend]),expiry('2,0,0')],
          doSomethingElse(A,B).

handle_crowd_answer(nice,Restaurant,Location):-
          getLocation(BestOne,Location).
```

3.4 Crowd Unification

This section provides an outline of crowd unification comprising the definition of crowd unification, its operational semantics and syntax, unifying rules and an extension with the crowd unification algorithm in Prolog. We first give a brief overview of traditional unification in a logic programming paradigm.

3.4.1 Definition of Unification

As mentioned earlier, unification is the kernel of deduction processes used in a theorem prover and a logic programming language, e.g., in Prolog [Sterling and Shapiro, 1994]. Unification is, informally, generalized matching. Two terms unify, if they are equal or if they contain variables that can be instantiated in such a way that the resulting terms are equal. In other words, the process of substitution is applied to unify terms and formulae - unification is a process of determining and applying a certain substitution to a set of expressions (terms or formulae) in order to make them identical. Let E_1 and E_2 be certain expressions. To unify E_1 and E_2 we must find a substitution σ which transforms the two terms into a single term. σ unifies E_1 and E_2 if and only if $E_1\sigma = E_2\sigma$. A substitution σ satisfying the above condition is called a unifier (or a unifying substitution) for expressions E_1 and E_2. In addition, a unifier for expressions E_1 and E_2 is a *most general unifier (mgu,*for short) if it is the most general substitution which unifies E_1 and

E_2. Formally, it can be defined that a substitution σ is the most general unifier for a certain set of expressions if and only if, for any other unifier θ of this set of expressions, there exists a substitution γ, such that $\theta = \sigma\gamma$.

3.4.2 The Idea of Crowd Unification

Crowd unification is an attempt to engage crowd computing in a pattern-matching scheme of the logic programming paradigm. Crowd unification refers to a distributed pattern-matching problem in which the queries/tasks are propagated beyond the boundaries of a local database through public networks in order to find a "unification" of two objects. Classical unification only provides substitution and a pairwise matching technique with a sequence pattern process which is inappropriate for complex queries such as unifying two images or sound or even media files. Such complex queries need more complicated abstractions for "unifying" them. However, this is a simple job if it is done by a human. Ultimately, integrating human intelligence with machine computation is expected to provide great promise for increasing capabilities to complete complex queries in logic programming.

Before we discuss the crowd unification operator and syntax, one essential feature that should be considered is a particular data type for doing crowd unification (in our case we focus on Prolog data types). Typically, there is only one type of data in Prolog called the *term*. There are four kinds of terms: atoms, numbers, variables, and complex terms (or structures). Atoms and numbers are lumped together under the category constants, and constants and variables together make up the simple terms of Prolog. In recent years, there have been wide varieties of media data types such as image, sound, and video, which are stored in database or knowledge base systems. In our case, we categorize these kinds of data in one type named *media*, under an extended notion of *constants*. In Prolog, media objects are collected in knowledge bases and are referred to via their stored path or filename enclosed in single quotes. Hence, for comparing media objects, the traditional pattern-matching scheme in unification, which is an act of checking a perceived sequence of strings for the constituents of their terms, is inadequate.

There is a built-in Prolog predicate `=/2` (recall that the `/2` at the end is to indicate that this predicate takes two arguments) with regard to unification. This predicate can be written in infix or prefix form. For instance, `jane = jane` is written as `=(jane,jane)` in standard Prolog form. In our research, we add a new built-in Prolog predicate "`*=/2`" which we call crowd unification to unify two terms by asking for crowd support. The operator "`*=`" is encoded by extending

the basic Prolog meta-interpreter which is described in Section 4.2.2, Chapter 4. For example, if we pose the query `'a.jpg' *= 'b.jpg'`, our extended Prolog will compare the images and respond with `'yes'` or `'no'` depending on the answer obtained via crowd support. The notion of crowd support is explained in more detail in the next sub-section.

More formally, crowd unification is defined as follows. Let E_1 and E_2 be expressions. To unify E_1 and E_2 with the crowd, we must find a substitution σ which transforms the two terms into equivalent terms (equivalence in a sense described below) and find a crowd support π which justifies that the terms are (crowd-)unifiable. We say that E_1 and E_2 unifies with crowd support, denoted by $E_1\sigma \star = E_2\sigma$, if and only if E_1 and E_2 unify according to the crowd unification rules below.

There are standard unification rules [Sterling and Shapiro, 1994] which determine the conditions for unifying a pair of terms. We have adjusted these rules to make them more suitable for the conditions of crowd unification. The crowd unification rules for logic programming are stated as follows.

- If term1 and term2 are constants, then term1 and term2 unify if and only if the crowd say they are the same. While this can be applied for arbitrary constants, a typical use case for this as illustrated earlier is that the two constants term1 and term2 are both referring to media type objects and the crowd says that the object depicted in both are the same (e.g., photos or video of the same person). Note that for consistency in a run of the program, we assume that the results are stored after asking the crowd in each run, so that the crowd is asked only once whether two terms are the same, i.e., the crowd cannot say that the two terms are the same in one goal and different in another goal. Also, for consistency, we have the following rules (which can be coded in our meta-interpreter):
  - *reflexivity*: term1 $\star$= term1
  - *symmetry*: term1 $\star$= term2 if and only if term2 $\star$= term1
  - *transitivity*: if term1 $\star$= term2 and term2 $\star$= term3 then term1 $\star$= term3

  With symmetry, if on asking, the crowd tells us that term1 unifies with term2, then we do not need to ask the crowd if we need to know whether term2 will unify with term1, rather, we assume so. With transitivity, if on asking, the crowd says that term1 unifies with term2 and also that term2

unifies with term3, then we do not need to ask the crowd if we need to know whether term1 unifies with term3, rather, we assume so.

- If term1 is a variable and term2 is any type of term, then term1 and term2 unify, and term1 is instantiated to term2. Similarly, if term2 is a variable and term1 is any type of term, then term1 and term2 unify, and term2 is instantiated to term1.
- If term1 and term2 are complex terms, then they unify if and only if:
  - they have the same functor and arity,
  - all their corresponding arguments unify, and
  - the variable instantiations are compatible.
- Two terms unify if and only if it follows the previous three clauses that they unify.

In the first clause, it tells us when two constants unify. Unifying two constants (e.g., each constant denoting a media object) is the key point of extension in crowd unification where crowd opinion is integrated to achieve the equivalence of the terms. The second clause tells us when two terms, one of which is a variable, unify. Just as importantly, this clause also says what instantiations we have to perform to make the two terms the same. The third clause tells us when two complex terms unify. Its first three clauses mirror perfectly the recursive structure of terms. Finally, the fourth clause is also important. It says that the first three clauses tell us all we need to know about the unification of two terms. If two terms cannot be shown to unify using clauses one to three, then they do not unify.

3.4.3 An Operational Semantics for Crowd Unification

We defined the operational semantics for *LogicCrowd* program in Section 3.3.2, and we extend the formal description semantics for crowd unification in this section. The following shows the [$crowd\_unification$] rule which refers to the world ω as a function returning a set of answers to queries, for a query q.

$$[crowd\_unification]\ \frac{\omega(?{=}(x,y)){=}\pi}{\omega, P \vdash_{\epsilon,\pi} (x\ \star{=}\ y)}$$

From this rule, π_q can be defined as a subset of crowd responses to query q denoted by $\pi_q = \omega(q) \subseteq (\{q\} \times U \times V)$. To unify two terms in the crowd unification scheme, π is the set of crowd answers which support their unification, e.g., for $x\star{=}y$, q is `?=(x,y)`, i.e., asking the question "Are term `x` and term `y` referencing the same?" So, for example, suppose the world only has two users, i.e., $U = \{u_1, u_2\}$, and both says x and y are the same, i.e., we have

$\pi_q = \{(q, u_1, yes), (q, u_2, yes)\}$. However, suppose $U = \{u_1, u_2, u_3\}$, and only two say x and y are the same, i.e., we have $\pi_q = \{(q, u_1, yes), (q, u_2, yes), (q, u_3, no)\}$, then we can still unify x and y if we accept a majority decision.

To ensure the quality of crowdsourcing results, we have exploited the workflow design that makes sufficient use of redundancy to guarantee quality results. We define a range of operators that are variants of the `*=/2` operator. The value of π, representing the level of crowd support that the two terms (x and y) unify, can be categorized into four different types. We give the crowd support (π) rules for crowd unification procedure more formally as follows:

$$[all]\ \frac{\omega(?{=}(x,y)){=}\pi\ \wedge\ \forall(q,u,v) \in \pi,\ v{=}yes}{\omega,\ P\ \vdash_{\epsilon,\pi}\ (x\ \ \star{=}_{all}\ \ y)}$$

$$[majority]\ \frac{\omega(?{=}(x,y)){=}\pi \wedge |\{u|(q,u,v) \in \pi \wedge v{=}yes\}| \geq \frac{|\pi|}{2}}{\omega,\ P\ \vdash_{\epsilon,\pi}\ (x\ \ \star{=}_{major}\ \ y)}$$

$$[at\ least\ \mathrm{D}]\ \frac{\omega(?{=}(x,y)){=}\pi \wedge |\{u|(q,u,v) \in \pi \wedge v{=}yes\}|{\geq}D}{\omega,\ P\ \vdash_{\epsilon,\pi}\ (x\ \ \star{=}_{D}\ \ y)}$$

$$[trust]\ \frac{\omega(?{=}(x,y)){=}\pi \wedge A{\subseteq}U \wedge |\{u|(q,u,v){\in}\pi\ {\wedge}v{=}yes\}|{\geq}\frac{|A|}{2}}{\omega,\ P\ \vdash_{\epsilon,\pi}\ (x\ \ \star{=}_{trust(A)}\ \ y)}$$

In all these rules, q is `?=(x,y)` and `x` and `y` are constants to be compared via humans (e.g., `x` and `y` denote photos or videos). These rules present the crowd support (π) in different types in which they agree that the two terms unify. Note that in the [*all*] rule, all crowd responses to the query assert that `x` and `y` are unifiable. The rule [*majority*] states that the largest group of people is the one who accepts that the two terms are the same. The third rule [*at least D*] states that to unify two terms `x` and `y`, it is enough that at least D persons of the crowd responses say "yes." Finally, to unify two terms with the last rule [*trust*], the majority of persons in a trusted (presumably, authoritative) group $A \subseteq U$ must say so.

3.4.4 Crowd Unification Algorithm

In this section, we describe an algorithm implementing crowd unification, which is derived directly from the operational semantics above. The algorithm finds the solution to complex queries that cannot be done by the typical machine unification scheme. However, we exploit the typical unification algorithm [Robinson, 1971] for basic substitution in order to assign values to variables in its arguments before matching these two terms with the crowd. The process of crowd unification occurs using the operator "`*=`" to send out the query to peers or to public social networks.

To avoid endlessly waiting for the crowd's responses, a waiting period or an expiry time is set. After expiry, the matching results from the crowd are collected and processed. There are several conditions to say that a pair of terms is equivalent as mentioned in the previous sub-section, for instance, two terms unify when the majority of crowd responses say both of them are the same. The condition of crowd support must be determined by the query issuer before the query is distributed to the crowd. The algorithm in Listing 3.1 illustrates the crowd unification process to identify equivalence of pairs of terms.

```
Function: CrowdUnification
Input:    Two terms T1 and T2 to be unified by a crowd, trusted
          group A, crowd support type ST, expiry time τ.
Output:   True or False
Algorithm:
    μ=0, S=0, S_A=0,
    while (τ is not reached)
    do
        R_μ = ask query(=?(T1,T2)) of Peer_μ
        μ  = μ + 1;
        if R_μ then S = S + 1
        if R_μ ∧ (Peer_μ ∈ A) then S_A = S_A + 1

    case ST
        all:        if μ  = S then return true
                    else  return false
        majority:   if S ≥ μ/2 then return true
                    else return false
        atleastD:   if S ≥ D then return true
                    else  return false
        trust:      if S_A ≥ |A|/2  then return true
                    else return false
```

Listing 3.1: The crowd unification algorithm

In the algorithm, μ denotes the total number of crowd responses to the query and S denotes the number of people who said that two terms are equivalent (S_A denotes those in a set of trusted people A who said the two terms are equivalent) whereas R_μ is the answer from $Peer_\mu$ (a member of the crowd) which is true if the two terms are the same, or false, otherwise. Note that the input for the algorithm includes two terms (T_1 *and* T_2), crowd support type (ST), trusted group A (if applicable), and expiry time (τ). T_1 *and* T_2 can be constant or variable. If they contain variables, these variables have to be bound to specific terms via the standard unification procedure [Robinson, 1971] before asking the crowd to unify. The output of the algorithm will return true if they unify, or return false if the terms do not unify. The algorithm starts by checking the expiry time or waiting period; the algorithm keeps sending the query and waiting for answers from the

crowd/peers until expiry. There are four cases of crowd support in order to identify two term equivalence, as mentioned in Section 3.4.3.

3.5 *LogicCrowd* in P2P Networks

Crowdsourcing refers to a distributed problem-solving model in which the problems/tasks are propagated beyond the local database through public networks. Classical crowdsourcing approaches tend to be centralized, where a server collects and computes answers generated by the crowd. Centralized methods are currently utilized by social networks such as Google+, Twitter and Facebook and so on. With the rapid growth of smartphone technologies over the past few years, a decentralized method of crowdsourcing has emerged. Mobile users can easily interact with each other in a Mobile Peer-to-Peer (M-P2P) fashion which can be regarded as an ad-hoc network supporting multi-hop routing, content forwarding, and distributed decentralized processing. In a M-P2P network, each peer is fully autonomous with regard to its respective resources. For crowdsourcing in mobile P2P ad-hoc networks, we should consider the key issues as follows.

- *Expressive query language and distributing content among peers*: M-P2P networks are mainly used for data sharing, and typically support a simple query facility. Modelling crowdsourcing in peer networks obviously needs mechanisms including protocols and a query language for propagating tasks and performing searches over distributed resources and devices.
- *Resource constraints of mobile devices*: M-P2P networks typically have resource constraints in terms of the battery power of the nodes. In the crowdsourcing approach, each mobile node can act as both a server and a client. A node acting as a client wishes to access a required task at a given frequency whereas the node acting as a server wishes to carefully broadcast tasks to mobile nodes with its limited battery level. Managing energy consumption is key.
- *Manipulating crowd answers over M-P2P networks*: After propagating crowd tasks among mobile peers, the crowdsourcing technique has to provide a set of routing operators in order to distribute computation and aggregate results for the query-originating peer. Routing algorithms and backtracking search might be applied to this problem.

In the following, we discuss the three main issues listed above for P2P-style crowdsourcing in mobile environments.

3.5.1 Extending Prolog for Mobile P2P Querying

As mentioned previously, *LogicCrowd* allows predicates to query the crowd rather than a closed set of facts in a local database. Also, *LogicCrowd* in a M-P2P network queries among peers via mobile communication technologies such as Wi-Fi and Bluetooth. Each peer can identify itself with a unique identifier called a peer identifier which can be an IP address, a MAC address and a port number. Hence, peers can send queries to each other using this identifier. While distributing queries among peers, *LogicCrowd* sends a query to discover other reachable running *LogicCrowd* programs, and then sends queries to the peers whom it discovers. When receiving the query, a peer answers the query and replies to the requestor. Moreover, the peer is able to pass tasks on to his/her friends and this process might continue with subsequent peers. In our process, we define time-to-live and power-to-live values to limit the lifetime of tasks so that the action of forwarding tasks to other peers can be stopped. We explore an extension of Prolog with new constructs that enable query evaluations to take place over multiple hops in M-P2P networks. The meta-interpreter with M-P2P feature in *LogicCrowd* given in Section 4.2.3, Chapter 4.

3.5.1.1 Decentralized-control P2P crowdsourcing Model

In this model, we assume that each peer is able to process and distribute their tasks without any control from the task-originating peer. Note that we use queries or task and query inter-changeably. Peers can convert a task or query it receives into another task when passing it on and the replies will be sent back along the same path as queries. In our case, a Bluetooth connection is used as a protocol for passing the tasks to peers. To propagate the tasks among peers, we introduce a new kind of crowd goal called *global task* written with a prefix "⋆" such as `⋆task`. This goal is evaluated by being propagated across the peer network. We assume that each peer on an M-P2P network has the means to receive, process, and if necessary, forward such goals to its peers. As an example, consider the following logic program which defines the task/query of asking the crowd about well-known Thai restaurants.

```
recommend(Restaurant):-thai(R),⋆nice?(Restaurant)
                       #[syn,asktype('choice'),
                       question('Which restaurant do you recommend?'),
                       options(R),askto([facebook,bluetooth]),
                       expiry('0,30,0')].
```

The rule sends the query for restaurant voting via Wi-Fi and Bluetooth technologies. The last goal `*nice?(Restaurant)#[syn,asktype('choice'),question ('Which restaurant do you recommend?'),options(R),askto([facebook,blu etooth]),expire('0,30,0')]` is evaluated as follows: the goal is first sent to the user's friend list (both on Facebook and via Bluetooth connection) and then the user's peers are permitted to forward this goal again to their friends. Typically, such a goal without the "$\star$" operator is evaluated only in the user's friend list as mentioned in `askto` of crowd conditions. However, when the goal invokes the global task rule, the execution is not just only in the local friend's list. The global task will be propagated to the friends of the peers to ask for their responses. Within a waiting period specified in crowd conditions, the peer waits for and collects the answers to the global goal.

However, there is an important drawback in this approach: peers may suffer from a high level of redundancy where the same task may be received or retransmitted by a peer multiple times. To alleviate this problem, we introduce a crowd algorithm where each peer records the tasks sent to his/her friends. In this case, the task's ID has been acquired and compared with the peer's known task IDs, and only a task with ID not found in the database is forwarded. The algorithm in Listing 3.2 shows the implementation in our *LogicCrowd*'s Mediator.

```
Define T is a set of taskID in receiver's DB
 @ Initial state (sender)   //In the originating node, its user creates a task
     create taskID          //Generate new task's identifier
     broadcast(task, taskID)  //Distribute a task to crowd with taskID
 @ Listening state (receiver)    //On receiving side
     receiving(task, taskID)
     if (taskID ∉ T) then        //If the received taskID is not in local DB
        set T = T ∪ {taskID}     //Store taskID
        broadcast(task, taskID) //Distribute a task to crowd with taskID
     end
```

Listing 3.2: The decentralized crowdsourcing algorithm for task distribution to avoid redundancies

Here, at an initial state, a sender will create a task, generate the task's ID, and then distribute the task and ID to the public. When a receiver approaches the task, the task's ID is checked in the local database: if it does not exist in the database, then the task's ID will be stored in its DB. Next, the receiver will continue broadcasting the task and ID to its friends.

3.5.1.2 Centralized-control P2P Crowdsourcing Model

In contrast to decentralized P2P crowdsourcing where each peer determines the relaying of tasks/queries, in this model, the origin peer (task's owner) is able to fully control the distribution of tasks to peers. By obtaining the identifier of the friends of its friend, the original peer can directly query the friends of friends, i.e. instead of friends passing the queries on, the origin peer gets its friends' friends and passes the query on itself (provided they are in network range). For this, we implement a crowd predicate formulated with the following construct: `PeerID*Task`. Similar to LogicPeer [Loke, 2006], we add the prefix of crowd task goal with the peer identifier and "*" in order to send a task to a specific peer. A peer asks for the friends of its friend via a goal as follows, where `PeerID` is the identifier of a peer and FL is a list of friend identifiers for the friends of the peer: `PeerID*friends(FL).`

```
eval(PeerID,Task):-
        PeerID*Task,        %send Task to the peer
        PeerID*friends(FL),%get access to friends' identifiers
        member(Friend,FL), %select peer's friend from friends list
        eval(Friend,Task). %send Task to peer's friend
```

Listing 3.3: A Prolog program supporting the P2P crowdsourcing model

As an example in Listing 3.3, the Prolog program supports the centralized-control P2P crowdsourcing model by allowing the task's owner or original peer to access the friends list of friends recursively. In doing so, the original peer can handle the list of peer's friends and find a peer which satisfies a given task. The task is evaluated against the peers in a peer network in a depth first search manner starting from the local peer until time-to-live (τ) and power-to-live (ξ) values limit the lifetime of task propagation. In the first rule of the predicate `eval/2`, the task is firstly evaluated against the given peer (`PeerID`). Sometimes, we can use the form: `self*Task` which means the task will be sent to the friends of the original peer. In the second rule, the peer `PeerID` is queried for its list of friends, and then one of the friends is selected via `member/2` and the goal is recursively evaluated against the selected friend. The goal `PeerID*friends(FL)` will fail if either the peer `PeerID` cannot be contacted or its friends cannot be obtained, in which case, Prolog backtracking on the `member/2` goal will result in another friend being chosen from the friend list. The evaluation completes when the result from the crowd has been returned to the origin peer due to expiry as in the crowd conditions.

3.5.2 Peer Propagation Control Mechanism

To reduce or even avoid endlessly propagating tasks among peers in a peer networks, power-to-live (ξ) and time-to-live (τ) have been defined. ξ restricts the maximum amount of energy to be used by each peer when propagating tasks through peer networks. The limited resources of mobile devices especially battery capacity can be used to address the problem of overuse of resources when propagating tasks in peer networks. The ξ value is set by an energy budget corresponding to a current energy level. ξ can be calculated by the following equation:

$$\xi = \beta(\%) \times E_{current} \tag{3.1}$$

where the energy budget β (%) is a user's policy of energy usage allowed for*LogicCrowd* programs to distribute tasks; $E_{current}$ denotes the current energy level based on the current battery power remaining. When a peer intends to broad/multi-cast a task, it must first estimate the average energy usage ($E_{transfer}$) (e.g., via preprogrammed benchmarks). To allow distributing tasks among peer networks, we use the condition given as follows:

$$E_{transfer} \leq \xi \tag{3.2}$$

According to this relation, the task is allowed to be forwarded only when the energy estimated for that task, i.e. $E_{transfer}$, is less than or equal to ξ. For example, assume that the mobile user specified an energy budget of 25% of the current phone's battery level and the current battery power $E_{current}$ is 4,440 mWh. When propagating a task to peers, the energy consumed is estimated to be 674.1 mWh, which is less than the energy budget (1,110 mWh); as a result, the system then continues to forward this task. In contrast, if the estimated energy of forwarding the task is greater than the energy budget, the process of forwarding will be stopped in order to maintain the energy levels.

Another possible control is the τ value which limits the lifetime of the task (or query) and each peer keeps track of the tasks which it has forwarded. Every task/-goal is tagged with a τ value which is modified (with time taken so far subtracted from it) across queries. In our case, the τ value, which is the expiry/waiting period in *LogicCrowd*'s conditions, is reduced after the peer forwards the task to another. The following equation is used to calculate a τ value for tagging queries when forwarding to peers:

$$\tau = T_{source} – T_{forwarder} - \mu \tag{3.3}$$

According to the relation above, T_{source} refers to the time when the task's

originator or owner is expecting answers to be returned by the other peers, while $T_{forwarder}$ defines the time when the sender/forwarder issues the query to other peers. μ relates to the time for message queuing, setting up or running the system. The μ value could be a default value preset in the *LogicCrowd* program or a system-defined value. A minimum value of τ should be greater than or equal to the μ of each device. τ-tagged queries can solve or ameliorate the unbounded problem of distributing tasks on peer networks. Moreover, both mechanisms can work cooperatively by setting the priority of ξ to be higher than that of τ. For example, in the case where we want to forward a task to other peers, the system will check whether the value of ξ is sufficient for such forwarding. Normally, when ξ is found to be sufficient, the forwarding process will run. However, if we lack sufficient τ or ξ, the task will not be forwarded. The peer propagation mechanism will be exploited in P2P opportunistic networks in Chapter 7.

3.5.3 Manipulating Crowd Answers

After waiting for a while (in our case, an expiry/waiting period in crowd conditions is set for a crowd query), the answers have been gleaned from the peer network. The replies go back along the same path that the query took in reaching the peer. It is still possible, and if not, e.g., when peers move out of ranges, answers are simple dropped. Two methods to deal with peer responses are as follows.

3.5.3.1 Black-box Crowd Answering

This method relates to the decentralized-control P2P crowdsourcing model as mentioned in Section 3.5.1.1 in which every peer processes queries and distributes tasks independently, without the original peer seeing this. Each peer individually propagates the tasks to its friends in order to ask them for their answers. Within a specified waiting period, the local peer waits for and collects answers from the origin peer, and finally the replies are sent back along the same route as the query. A peer is not able to see through all paths of peer networks. One advantage of this method is to reduce the workload of each peer assembling the crowd's answers. In contrast, a disadvantage is also witnessed: the origin peer cannot take control of subsequent peers in order to distribute his/her tasks. Consequently, it might be difficult to determine the quality and reliability of the answers received. For reliability, we also introduce two operators to simplify controllable peer answers.

Firstly, a goal of the form: **[Number]*Task** evaluates in such a way that the Task must be broadcasted to peers and the Number denotes the minimum required number of responding peers to the task. The *LogicCrowd* program will process and collect the answers from the crowd until the total number of responding peers reaches such a minimum (unless time expired). For example, a task of the form `[10]*niceRestaurant` means that the task originator needs at least 10 people to respond to this query. On obtaining 10 responses (whatever the answers), the results are returned without waiting until the expiry time.

Secondly, a goal of the form: **<Number>*Task** evaluates in such the way that the Task must be broadcasted to peers and the Number now denotes the minimum frequency of an answer that will be accepted by the origin. In dealing with answers, these results from peers will be collected and calculated. If an answer is found where the number of peers with this answer reaches the number required, the system will automatically return the responses – there is no need for a delay until the expiry period imposed in the original crowd conditions. For example, with a task of the form `<5>*niceRes....option[a, b, c]`, as soon as any one answer, say 'a', is chosen by at least 5 peers, the result 'a' will be returned without further waiting. The two operators control the return of peer responses and offer a more flexible approach. Meanwhile, the original peer can have some control over the extent to which tasks are propagated.

3.5.3.2 Transparent Crowd Answering

This method has been designed for the centralized-control P2P crowdsourcing model as mentioned in Section 3.5.1.2 in which the original peer fully controls the distribution of the task to peers. By acquiring peers' IDs, the task can be propagated directly to the peer. Furthermore, the peer can return the list of his/her friends to the original peer in order to choose appropriate friends to forward the tasks. Obviously, this relies on the original peer being able to gather, in a transparent way, the identifiers of the friends of its friends (their friends, etc). The advantage of this method is that the origin peer is able to control the peer network by selecting the peers to whom he/she would like to broadcast. Using this method, the results are likely to be promising and are regarded as potentially more reliable and higher in validity. However, one possible drawback is that since an original peer is the only one who processes all the answers sent by peers, the load is much heavier on the original peer, and also the friends of friends must be

within range of the original peer. Hence, such transparent crowd answering is not always possible.

3.6 Summary

We have proposed a declarative style of programming for crowdsourcing, especially in the mobile environment. This paper discusses a simple extension of Prolog, which we call *LogicCrowd* that automatically leverages human knowledge through the crowdsourcing paradigm. In our framework, *LogicCrowd* allows programmers/users to create their rules via an event-driven approach and interfaces with various platforms: crowdsourcing market platforms, social networks, and P2P networks.

We have also introduced a novel unification approach, called crowd unification which is embedded in an extension of Prolog. The unification of the terms is the key feature of logic-based programming which represents the mechanism of pattern matching over logic terms. Furthermore, we have expanded our application of the declarative programming paradigm over mobile peer networks. This extension enables query evaluations to take place over multiple hops in mobile P2P networks.

Note that waiting for certain threshold results, limiting the wait (e.g., via τ or ξ), and asynchronous querying are important features in a dynamic mobile environment where peers who are within connection range (e.g., Bluetooth or Wi-Fi Direct range) can vary, as nodes discover or disappear and nodes are discovered or reappeared.

In summary, this chapter provided an overview of our declarative programming platform for mobile crowdsourcing. By presenting *LogicCrowd* which serves as a tool to open knowledge base querying to the public and to integrate the crowdsourcing model into the mobile environment, research questions are addressed and solutions provided. In the next chapter, we illustrate the potential of our approach via programming idioms, a prototype implementation and several scenarios.

Chapter 4

Implementation of the *LogicCrowd* Framework

We discussed the main issues of *LogicCrowd* and its formalism for representing crowdsourcing programs as an extension of Prolog in Chapter 3. In this chapter, we detail the *LogicCrowd* framework, which is built based on the *LogicCrowd* concepts. The *LogicCrowd* framework integrates logic programming with crowdsourcing platforms in order to provide a declarative programming platform for mobile applications. The key components of the *LogicCrowd* framework, including Prolog, Mediator, Knowledge Base and Crowdsourcing APIs, are designed to combine conventional machine computation and the power of the crowd.

The chapter is organized as follows. Section 4.1 introduces the architecture of the *LogicCrowd* framework which is a declarative programming platform integrated with crowdsourcing for mobile environments. Section 4.2 describes the Interpreter component of the *LogicCrowd* framework and gives details on the extension of the pure Prolog meta-interpreter with crowd unification and M-P2P processing. Section 4.3 introduces the Mediator component and explains our custom-built Prolog libraries. In addition, we describe the operations of the crowdsourcing APIs used by *LogicCrowd* in this section. We summarize the chapter in Section 4.4.

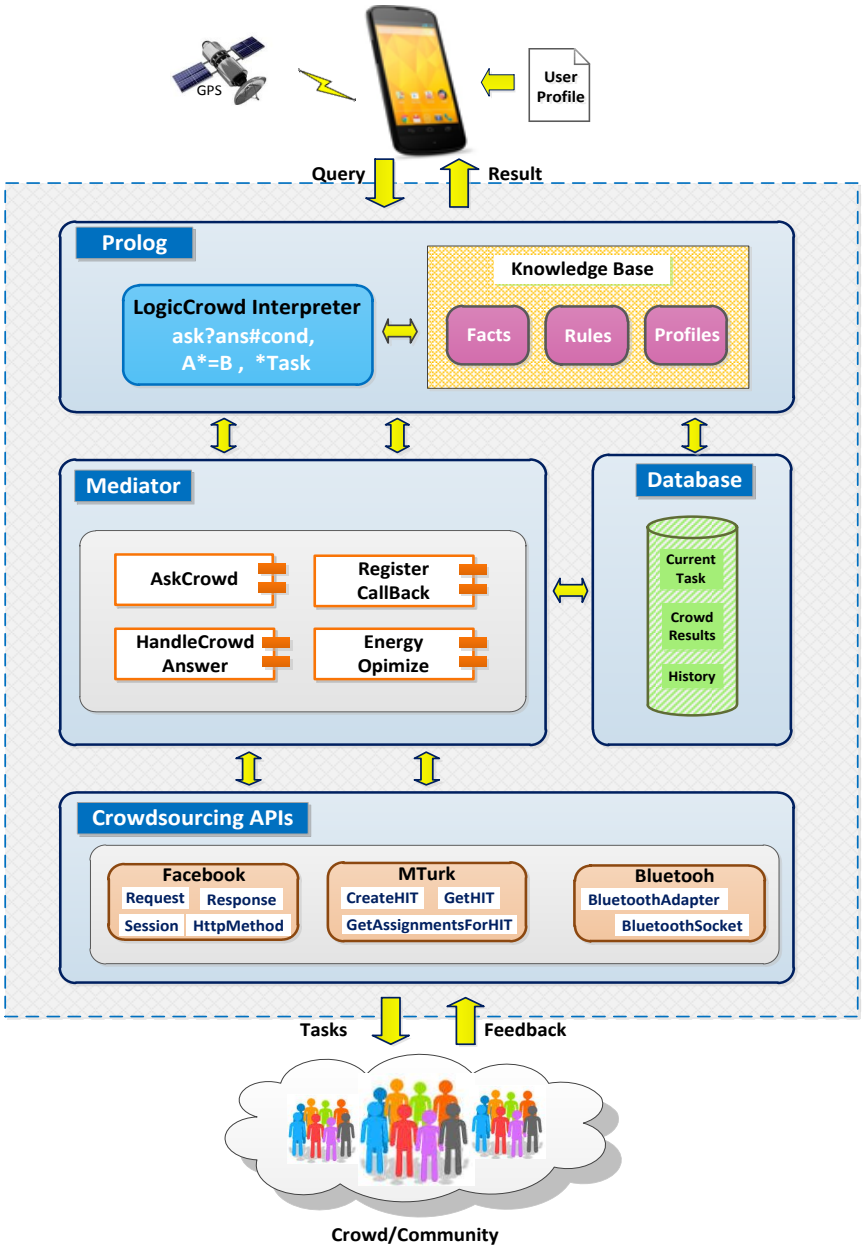


Figure 4.1: An overview of the *LogicCrowd* Framework

4.1 Conceptual Framework

We introduce the *LogicCrowd* framework, which utilizes the logic programming paradigm to allow querying and manipulation of knowledge and reasoning, including querying the crowd, in order to solve complex problems via mobile technologies. The main contribution of the *LogicCrowd* framework is providing an innovative approach that integrates logic programming with crowdsourcing middleware in order to provide a declarative programming platform for mobile applications.

We have built a prototype implementation of the *LogicCrowd* framework by using the Android platform. The prototype integrates tuProlog with Android via our own custom-built Java programs. The corresponding high-level architecture of the *LogicCrowd* framework is described in Figure 4.1 The *LogicCrowd* framework contains four main components: Prolog, Mediator, Knowledge Base and Crowdsourcing APIs.

The execution of *LogicCrowd* programs is initiated by the mobile user posing a single goal, called a query, through the system's user interface form, which is similar to a Prolog command shell. Currently, *LogicCrowd* is designed for users/developers who have a basic background in Prolog. The users are able to write rules or small programs in the Prolog style in order to make the *LogicCrowd* application more flexible for evaluating complex goals. After users make a query to the system, the query is sent to the Prolog component for evaluation. The interpreter is a program that evaluates programs by performing yes/no computation. It interprets input including a rule/program and a query, and answers yes if the query is a logical consequence of the program and no otherwise [Sterling and Shapiro, 1994]. The basic computation models of logic programs such as unification, backtracking, substitution, handling of conjunctions and so on have been used in this component.

The *LogicCrowd* Interpreter is an extension of the interpreter of pure Prolog which adds extra features (e.g., to evaluate the crowd predicate by connecting to the crowd). The extension of the basic Prolog meta-interpreter is described in Chapter 3 as well as in Section 4.2 which presents the extension of meta-interpreter of the crowd unification and P2P network. In this component, the collection of rules, programs and facts of the system are stored in the local database or knowledge bases. The user profile (e.g., personal information, his/her friends' lists in social media) and mobile context data (e.g., GPS coordinates, peers discovery via Bluetooth) are also collected in the knowledge base. These data are used when evaluating the query or goal.

The second component of the *LogicCrowd* system is called a Mediator which is created by our own custom-built Java program in the prototype implementation of the *LogicCrowd* system. As the names suggests, the Mediator contributes to the communication between the *LogicCrowd* Interpreter and crowdsourcing platforms. This enables the logic program to contact the public, rather than working only with a closed set of facts in a local database. The functionalities of the Mediator are described as follows.

- *Executing crowd queries* – After a crowd predicate is evaluated by the *LogicCrowd* Interpreter, the crowd's parameters (e.g., question, crowd's conditions, etc.) are delivered to the Mediator. The Mediator prepares and transforms these conditions into an acceptable form for crowdsourcing services. The request to create the crowd's task is then propagated to the crowd via crowdsourcing APIs.
- *Registering and callback the issued queries* – As mentioned, the crowd predicates with conditions are executed via the Mediator which sends these conditions to the crowdsourcing platforms. The query is registered as in an index. When the results are returned, the system automatically calls back the registered function associated with the query and then continuously executes the next sub-goal of the *LogicCrowd* program. This execution technique which is either Synchronous or Asynchronous has been described in Chapter 3. In Section 4.3, the detailed process flow diagrams when implementing these executions are presented.
- *Handling the crowd results* – When executing a crowd task, there may be a time lag as a result of waiting for the return of answers from the crowd. The Mediator will manage these answers either by displaying them to the mobile user interface or by returning results by instantiating Prolog variables.
- *Optimizing energy consumption* – We modify the *LogicCrowd* Interpreter to use the energy estimations during run-time to monitor application workload and adapt its behavior dynamically to save energy. The estimated power models per crowd goal or rule for different network connections with a particular execution method have been proposed (see Chapter 6). The Mediator estimates the energy usage of a crowd predicate with an energy budget corresponding to a certain battery lifetime. If the estimated energy of the crowd predicate/query is greater than the energy budget, the query is skipped and the system stops the process in order to maintain the battery usage within budget.

In the database component, the data repositories include task information, crowd results, and query history. The task repository keeps the task data such as ID, question, and crowd conditions. It is a temporary database that records the current task which is propagated to the crowd. In addition, it facilitates the ability to refer to the results that are returned by the crowd. Meanwhile, the crowd result repository records the feedback of relevant crowd tasks. It also includes raw data collected from several sources for analysis and display. The query history

database keeps the historical data of task queries and crowd results. The query history database collects and records the past queries and answers initially posed by the crowd. This database is considered as useful in case the duplicated query is posted in the system; that is, we can save time by making use of its answer which has been already recorded in the database, functioning as a cache.

The last component is the Crowdsourcing APIs which is an important part of crowdsourcing queries. In our work, there are three different crowdsourcing platforms that *LogicCrowd* is able to access. These platforms include Facebook, MTurk and P2P via Bluetooth and Wi-Fi. *LogicCrowd* exploits the Facebook API to post the task to mobile users' walls. Also, *LogicCrowd* utilizes MTurk API to create the task and send the request to workers via MTurk's website. We use the Android Bluetooth API to let the system wirelessly connect to other Bluetooth devices, enabling point-to-point and multipoint communication features.

4.2 *LogicCrowd* Framework: Meta-Interpreter

As mentioned in Chapter 3, *LogicCrowd* has three important extensions to Prolog: (1) predicates to query the crowd (2) specific operators encoded by extending the basic Prolog meta-interpreter designed to cooperate with the crowd, and (3) two methods of the logic execution model. In this section, we describe the extended Prolog meta-interpreter for the *LogicCrowd* framework.

4.2.1 The Basic Meta-Interpreter for *LogicCrowd*

We introduced the crowd predicate and its operators containing the crowd identity and crowd conditions in Chapter 3, Section 3.3.1. The crowd predicate is encoded by extending the pure Prolog meta-interpreter. In general, a meta-interpreter for a language is an interpreter for the language written in the language itself. It gives access to the computation process of the language and enables the building of an integrated programming environment [Sterling and Shapiro, 1994]. As we already noted, Prolog is a language suitable for writing meta-interpreters and, in particular for writing meta-programs which are the programs that use other programs as data. The basic meta-interpreter of pure Prolog is usually called vanilla (see Listing 4.1).

The predicate `solve` is used for a meta-interpreter of pure Prolog. Three clauses of the program above are written on another level of abstraction which

is usually called the clause reduction level. It makes use of the built-in meta-predicate `clause(Head,Body)` which, assuming `Head` is instantiated, finds a clause matching the head `Head` and then instantiates `Body` to the body of that clause. The clause reduction level mentioned above represents only one of many possible levels of abstraction on the computation of a meta-interpreter. When analyzing the structure of meta-interpreters, one can identify various levels of abstraction.

```
solve(true).
solve((A,B)):-solve(A), solve(B).
solve(A):- clause(A,Body),solve(Body).
```

Listing 4.1: A meta-interpreter for pure Prolog

The following meta-interpreter has been used in our *LogicCrowd* system. In this version, the cut (!) and negation (not or \+) are used to control backtracking which is able to reduce the search space of Prolog computations. In this version, it is able to evaluate the built-in predicate of pure Prolog by using the `builtin/1` predicate.

```
solve(true):-!.
solve(not(A)):- !,\+solve(A).
solve((A)):- builtin(A),!, A.
solve((A, Body)) :- !,solve(A), solve(Body).
solve((A)):- clause(A,Body),solve(Body).
```

Listing 4.2: A meta-interpreter for pure Prolog with cut and negation

The *LogicCrowd* program is an extension of the pure Prolog meta-interpreter. By introducing crowd predicates as mentioned in Section 3.3.1, it is able to query the crowd rather than a closed set of local databases. The parts of the *LogicCrowd* meta-interpreter considered to be a foundation of the programs are shown in Listing 4.3. The complete meta-interpreter for *LogicCrowd* can be seen in Appendix A. Listing 4.3 shows that the *LogicCrowd* meta-interpreter is presented in a simplified form as an extension of pure Prolog. In this thesis, we use tuProlog[1] which is able to integrate seamlessly with Java/Android.

[1]http://tuprolog.apice.unibo.it/

```
:- op(70,xfy,'?').
:- op(75,xfy,'#').
%% as a pure Prolog extension
solve(true):-!.
solve(not(A)):- !,\+solve(A).
solve((A)):- builtin(A),!, A.
solve((A, Body)) :- !,solve(A), solve(Body).
solve((A)):- clause(A,Body),solve(Body).
%% LogicCrowd extension
solve(Askcrowd?Result#Condition):- !, solvecond(Condition),
        (asyn,!,asynproc(Askcrowd,Result); synproc(Askcrowd,Result)).
synproc(Askcrowd,Result):-
        checkcond(TypeQuestion,Question,Namemsg,Link,Description,Picture,Options,
                Askto,Group,Locatedin,Expiry,EndTime,WPTime,Forward,ForwardQID,
                ST,AtLeast),
        askcrowd(syn,Askcrowd,TypeQuestion,Question,Options,Namemsg,Link,
                Description,Picture,Askto,Group,Locatedin,Expiry,EndTime,WPTime,
                Forward,ForwardQID,ST,AtLeast,QuestionID).
        registercallbacksyn(QuestionID,Question,Askto,TypeQuestion,Expiry,Forward,
                Result).

asynproc(Askcrowd,Result):-
        checkcond(TypeQuestion,Question,Namemsg,Link,Description,Picture,Options,
                Askto,Group,Locatedin,Expiry,EndTime,WPTime,Forward,ForwardQID,
                ST,AtLeast),
        askcrowd(syn,Askcrowd,TypeQuestion,Question,Options,Namemsg,Link,
                Description,Picture,Askto,Group,Locatedin,Expiry,EndTime,WPTime,
                Forward,ForwardQID,ST,AtLeast,QuestionID).
        registercallbackasyn(Askcrowd,QuestionID,Question,Askto,TypeQuestion,
                Expiry,Forward,ST,AtLeast,Result).
solvecond([]):- !.
solvecond(Condition):-Condition=..[_H|[Head,Body]],asserta(Head),solvecond(Body).
checkcond(TypeQuestion,Question,Namemsg,Link,Description,Picture,Options,Askto,
         Group,Locatedin,Expiry,EndTime,WPTime,Forward,ForwardQID,ST,AtLeast):-
                (asktype(A),!, TypeQuestion = A; set(TypeQuestion)),
                (question(B),!, Question = B; set(Question)),
                (askto(C),!,Askto = C; set(Askto)),
                (link(D),!, Link = D; set(Link)),
                (description(E),!, Description = E; set(Description)),
                (picture(F),!, Picture = F; set(Picture)),
                (options(G),!, Options =G; set(Options)),
                ...
                (expiry(K),!,Expiry = K; set(Expiry)).
set(X):- X = 'null'.
```

Listing 4.3: The *LogicCrowd* meta-interpreter – an extension of pure Prolog

As mentioned in Chapter 3, the crowd predicate has its own operators "`?`" and "`#`" which can be embedded into the Prolog program as a distinguished predicate referring to the crowd identity and crowd conditions. In Prolog, users are able to define new operators through the `op/3` predicate which is presented in the form

below:

```
:- op(Precedence, Type, Name).
```

Precedence is a number between 0 and 1200. The higher the number, the greater the precedence will be. Type is an atom specifying the type and associativity of the operator. In our case, this atom is `xfy` that is an infix operator; the `f` represents the operator, and the `x` and `y` represent the arguments. In our case, crowd operators have been defined as follows.

```
:- op(70,xfy,'?').
:- op(75,xfy,'#').
```

From the above rule, the `solve/1` predicate represents a meta-interpreter for pure Prolog extended to evaluate goals with the crowd operators. This rule delegates the evaluation of such goals to `solvecond/1, asynproc/2, and synproc/2` predicates.

```
solve(Askcrowd?Result#Condition):-
              !, solvecond(Condition),
              (asyn,!,asynproc(Askcrowd,Result);
               synproc(Askcrowd,Result)).
```

The `solvecond/1` predicate represents a meta-interpreter for the crowd conditions. It asserts the new fact (the crowd conditions) which will be inserted at the beginning of the knowledge base.

```
solvecond([]):- !.
solvecond(Condition):- Condition =..[_H|[Head,Body]],
                       asserta(Head),
                       solvecond(Body).
```

The `checkcond/17` predicate binds the values of the crowd conditions to actual variables, and sets "`null`" to variables in the case that the fact is not in the knowledge base.

```
checkcond(TypeQuestion,Question,Namemsg,Link,Description,Picture,
         Options,Askto,Group,Locatedin,Expiry,EndTime,WPTime,Forward,
         ForwardQID,ST,AtLeast):-
              (asktype(A),!, TypeQuestion = A; set(TypeQuestion)),
              (question(B),!, Question = B; set(Question)),
              (askto(C),!,Askto = C; set(Askto)),,
              (link(D),!, Link = D; set(Link)),
              (description(E),!, Description = E; set(Description)),
              (picture(F),!, Picture = F; set(Picture)),
              (options(G),!, Options =G; set(Options)),
              ...
              (expiry(K),!,Expiry = K; set(Expiry)).
set(X):- X = 'null'.
```

In the next sub-goal, `(asyn,!,asynproc(Askcrowd,Result);synproc(Ask-`

`crowd,Result))` expresses exactly the `If-Then-Else` control construct in Prolog. This means that if "`asyn`" then `asynproc/2` is evaluated, otherwise `synproc/2` has been solved. The `asynproc/2` rule calls `askcrowd/20` and `registercallbackasyn/10` predicates to execute the task in asynchronous mode. In contrast, the `synproc/2` predicate calls `askcrowd/20` and `registercallbacksyn/7` to evaluate the task by the synchronous execution mode. The `askcrowd/20` predicate connects to the crowd. The `askcrowd/20` passes the crowd conditions to a process outside of the main tuProlog thread; in our case, the particular social media network (Facebook), mobile peer networks (Bluetooth/WiFi Direct), and online crowdsourcing market platforms (MTurk). The query is issued to the crowd (i.e. friends on Facebook) via the Mediator which is running on the Android platforms.

The `registercallbackasyn/10` and `registercallbacksyn/7` predicates, where the query is executed in asynchronous and synchronous mode respectively, are also bound to the crowd via Mediator. It functions (1) to register tasks (which have been sent to the crowd) and (2) to return the results. After posting the task to the crowd (i.e., Facebook), this predicate will register the task and return all results from the crowd back to the main program after the expiration time. These goals are basic functions of the *LogicCrowd* meta-interpreter. We also have extensions for crowd unification and M-P2P networks which will be explained in more detail in the following section.

4.2.2 Extending the Meta-interpreter to Include Crowd Unification

Our *LogicCrowd* meta-interpreter has been first introduced in the previous section with details on the basic operators. In this section, we introduce new operators that invoke crowd unification by extending the pure Prolog meta-interpreter. We have added crowd unification and crowd response counting operators to the *LogicCrowd* meta-interpreter. The crowd unification operators "*=" are embedded into the Prolog program as a distinguished predicate. The excerpt for extending crowd unification from the *LogicCrowd* meta-interpreter is given in Listing 4.4

In Listing 4.4, we define the operators for crowd unification (*=) in either prefix or infix form with the same precedence. The crowd operator definition for "*=" is as follows:

```
:- op(35,fx,'*=').
:- op(35,xfy,'*=').
```

```
:- op(35,fx,'*=').
:- op(35,xfy,'*=').
...
%% crowd unification extension
solve((*=(A,B))):-!,solve(A),solve(B),crowdUni(A,B,all).
solve((A*=B)):- !,solve(A),solve(B),crowdUni(A,B,all).
solve((*=(A,B)@ST)):- !,solve(A),solve(B),crowdUni(A,B,ST).
solve((A*=B)@ST):- !,solve(A),solve(B),crowdUni(A,B,ST).
solve((A*=B)@ST*Peer#Time):- !,solve(A),solve(B),crowdUni(A,B,ST,Peer,Time).
crowdUni(Term1,Term2,ST):-
        setLink(Term1,Term2,Path),setCond(Path,ST),asynproc(unify,Result).
crowdUni(Term1,Term2,ST,Peer,Time):-
        setLink(Term1,Term2,Path),
        setCond(Path,ST,Peer,Time),asynproc(unify,Result).
setLink(Term1,Term2,Path):-
        Term1 =.. [Head1|Path1], Term2 =.. [Head2|Path2],
        length(Path1, Length1),length(Path2, Length2),
        (Length1 = 0,!,append(Path1,Head1,Link1); Path1 =.. [_H1|[Link1,T1]]),
        (Length2 = 0,!,append(Path2,Head2,Link2); Path2 =.. [_H1|[Link2,T2]]),
        Path = [Link1,Link2].
setCond(Path,ST):-
        asserta(asktype('choice')),asserta(question('Are they the same?')),
        asserta(options(['Yes','No'])),asserta(picture(Path)),
        asserta(askto([bluetooth,mturk,facebook])),
        asserta(expiry('0,20,0')), asserta(supporttype(ST)).
setCond(Path,ST,Peer,Time):-
        asserta(asktype('choice')),asserta(question('Are they the same?')),
        asserta(options(['Yes','No'])),asserta(picture(Path)),asserta(expiry
        (Time)),(Peer = 'mix',!,asserta(askto([bluetooth,facebook,mturk]));
        mturk(Peer)),asserta(supporttype(ST)).
mturk(Peer) :- (Peer = 'mturk',!,asserta(askto([mturk]));facebook(Peer)).
facebook(Peer):-(Peer = 'facebook',!,asserta(askto([facebook]));bluetooth(Peer)).
bluetooth(Peer):-(Peer = 'bluetooth',!,asserta(askto([bluetooth]))).
```

Listing 4.4: An excerpt of the *LogicCrowd* meta-interpreter extended with crowd unification

In crowd unification predicates, there are five different styles available for querying. The following examples show the crowd unification query in various styles, explained further in Section 5.2, Chapter 5.

```
?- photoA.jpg *= photoB.jpg.
?- *=(photoA.jpg,photoB.jpg).
?- photoA.jpg *= photoB.jpg@major.
?- *=(photoA.jpg,photoB.jpg)@major.
?- photoA.jpg*= photoB.jpg@major*mturk#"0,30,0".
```

As shown in the meta-interpreter above, the `solve/1` predicate represents the meta-interpreter for pure Prolog extended to evaluate goals with the crowd unification operators. To be more flexible to evaluate the crowd unification goal, we create two different crowd unification predicates: `crowdUni/3` and `crowdUni/5`. However, both rules delegate the evaluation of the crowd-powered goals to `setLink/3`,

`setCond/3` and `asynproc/2` predicates. The `crowdUni/3` rule contains three parameters: `Term1, Term2` (the objects to ask the crowd to unify) and `ST` which refers to a type of crowd support to unify the two terms. The `crowdUni/5` rule involves five arguments: `Term1, Term2, ST, Peer,` and `Time`. `Peer` relates to the crowdsourcing platforms and `Time` is an expiry time for the crowd to return the feedback.

The `setLink/3` predicate evaluates the terms (`Term1` and `Term2`) which are two objects (e.g., photos) for unifying and then the path of such objects is determined. In the next sub-goal, the `setCond/2` predicate represents processing of the crowd unification conditions. It asserts the new fact (the important conditions for crowd unification) which is inserted at the beginning of the knowledge base. The `setCond/4` predicate has the same predicate name as `setCond/2` but there are four arguments. The aim `setCond/4` is similar to `setCond/2`, but the `Peer` variable for the crowdsourcing platforms (e.g, Facebook, MTurk, and Bluetooth) and the `Time` variable which contains the waiting period for the results to be returned by the crowd (e.g., waiting 30 minutes) have been included for the evaluation process.

Crowd unification exploits `asynproc/2` (as described in Section 4.2.1) to connect with the crowd. The `asynproc/2` predicate calls the `askcrowd/20` and `registercallbackasyn/10` predicates. The first `askcrowd/20` connects to the crowd by passing the crowd conditions to the Mediator (as describe in Section 4.3.1). Then, the task is created and propagated to the external crowd or to other peer devices via the available API crowdsourcing platforms. Next, the goal `registercallbackasyn/10` is called and is executed via the Mediator. It registers tasks that have been sent to the crowd and manages the called results.

4.2.3 Extending the Meta-Interpreter for Mobile P2P Processing

As mentioned previously, *LogicCrowd* in a M-P2P network is able to query peers through mobile connection technologies such as Bluetooth. The process starts when *LogicCrowd* sends a query to discover other peers who are running *LogicCrowd* programs within a connection range. When receiving the query, a peer answers and replies to the requester. The peer is also able to pass the query to his/her friends and this process might continue with subsequent peers. Two propagation models have been proposed, ie., the decentralized and centralized control P2P crowdsourcing models as discussed in Section 3.5, Chapter 3. In this section,

we present an extension of Prolog that allows query evaluations over multiple hops in M-P2P networks.

In the decentralized-control P2P crowdsourcing model, the task is distributed from a peer to other peers without any control from the originating source (i.e., a requester). The "global task" written with a prefix "*" such as `*task` has been introduced to propagate the task among peers. This means that each peer on a M-P2P network has the means to receive, process, and if necessary, forward such goals to their peers. The global goal operator "*" is embedded into the Prolog program as a distinguished predicate. The excerpt for extending P2P to *LogicCrowd* meta-interpreter is given in Listing 4.5.

```
:- op(65,fx,'*').
...
solve(*Askcrowd?Result#Condition):- !,
      solvecond(Condition),asserta(forward(true)),
     (asyn,!,asynproc(Askcrowd,Result); synproc(Askcrowd,Result)).
 ...
```

Listing 4.5: The rule for P2P processing added to the *LogicCrowd* meta-interpreter

In Listing 4.5, the operator notation for the peer task "*" is defined in the prefix form with precedence value 65. The `solve/1` predicate represents the meta-interpreter for pure Prolog extended to evaluate goals with the global peer task operators. This rule delegates the evaluation of crowd-powered goals to `solvecond/1, asserta/1, asynproc/2,` and `synproc/2` predicates. As mentioned in Section 3.3.2, the `solvecond/1` predicate asserts the new fact (crowd conditions) which will be inserted at the beginning of the knowledge base. The `asserta/1` is a built-in predicate that asserts a fact "`forward(true)`" in the database. By doing this, the term is asserted into the crowd condition as the first fact or clause of the corresponding predicate. After finding this condition, the Mediator further processes it by allowing peers to forward the task to others.

In the next sub-goal, `(asyn,!,asynproc(Askcrowd,Result);synproc(Askcrowd,Result))` expresses exactly the If-Then-Else control construct in Prolog. This means that if "`asyn`" then `asynproc/2` is evaluated, otherwise `synproc/2` is solved. As mentioned in Section 4.2.1, the `asynproc/2` rule calls `askcrowd/20` and `registercallbackasyn/10` predicates to execute the task in the asynchronous mode. In contrast, the `synproc/2` predicate calls `askcrowd/20` and `registercallbacksyn/7` to evaluate the task by the synchronous execution mode.

In contrast, centralized P2P crowdsourcing is able to fully control the distribution of tasks to peers. The original peer can query friends of friends by running

a logic program in itself or in remote peers by obtaining the identifier of the friends of its friend. The implementation of a central control peer processing predicate is in the form: `PeerID*Task`. We also add the prefix of the crowd task goal operator with the peer identifier in order to send a task to a specific peer. The rule for extending centralized-control P2P added to the *LogicCrowd* meta-interpreter is given in Listing 4.6.

```
:- op(72,xfy,'*').
...
solve(PeerID*Askcrowd?Result#Condition):- !,
      solvecond(Condition), solvepeer(PeerID),
      (asyn,!,asynproc(Askcrowd,Result); synproc(Askcrowd,Result)).
solvepeer(PeerID):- friend(PeerID), asserta(askto(PeerID)).
...
```

Listing 4.6: An excerpt of the *LogicCrowd* meta-interpreter extended with P2P functionality

The global task "*" operator has been defined in infix form with a lower precedence than the decentralized control operator. The `solve/1` predicate represents the extension of the Prolog meta-interpreter which evaluates goals with the centralized control model. This rule delegates the evaluation of global task goals to `solvecond/1`,`solvepeer/1`,`asynproc/2`, and `synproc/2`. These rules (mentioned in the basic extension of the *LogicCrowd* meta-interpreter) have been exploited in this method. However, the rule `solvepeer/1` has been added to send the task to a specific peer. This rule executes the `friend/1` predicate which returns "true" when the PeerID is in the user's friend list. Moreover, the built-in predicate `asserta/1` when executed, adds a new fact "`askto(PeerID)`" to the crowd conditions. Then the mediator is able to distribute the task to specific peers (`PeerID`).

4.3 *LogicCrowd* Framework: Mediator and Crowdsourcing APIs

The Mediator is an important component that bridges the gap between logic programming and crowdsourcing platforms. The prototype implementation of *LogicCrowd* is programmed based on Prolog by using tuProlog integrated with the Android platform. tuProlog is able to fully access Java resources such as objects, classes, methods, etc. Moreover, tuProlog allows the development of new tuProlog libraries, enabling us to write additional functions making it possible for Prolog to communicate with the crowdsourcing platform. In this section, we detail how the crowdsourcing APIs have been used and implemented in the Mediator of

the *LogicCrowd* framework, also explaining how the Mediator communicates with crowdsourcing APIs

4.3.1 Mediator – The custom-built Prolog libraries

The Mediator in the *LogicCrowd* framework has been implemented by extending the tuProlog class library in Java. The extended library defines our new predicates in the form of methods. Listing 4.7 gives an excerpt of the *LogicCrowd* extended library, defining the predicate (`askcrowd/20`). The Java method `askcrowd_20` implements the predicate `askcrowd/20`. It receives the *crowd keyword* and *crowd conditions* from the Prolog program as input terms. These inputs are then processed into a format that is appropriate for communicating with different crowdsourcing platforms. Finally, when the feedback from the crowd is returned, the results are reported through the output variable (`outresult`).

```
import alice.tuprolog.*;
public class LogicCrowdLibrary extends Library {
// predicate askcrowd/20
public boolean askcrowd_20(Term exec,Term crowdkeyword,Term typequestion,
              Term question, Term options,Term namemsg,
              Term link,Term description,Term picture,Term askto,
              Term group,Term locatedin,Term expiry,Term endtime,
              Term wptime,Term forward,Term forwardqid,
              Term supporttype,Term atleast, Var questionID) {
      String QID = new String();
      QID =createID();
      setCrowdInfo(exec,crowdkeyword,typequestion,question,options,
          namemsg,link,description,picture,askto,group,loctedin,
          expiry,endtime,wptime,forward,forwardqid,supporttype,atleast);
      for (int i = 0; i < askto.length; i++) {
      switch (askto.valueOf(AskTo)){
              case bluetooth:
                   private LogicPeerService PeerService = null;
                   PeerService.startprocess(QID,question,expiry);
              case facebook:
                   Thread threadfacebook = new Thread(){
                   public void run() {
                      logicfacebook.PostQuestion(params,pathpost);}};
                   threadfacebook.start();
              case mturk:
                   Thread threadmturk = new Thread(){
                   public void run() {
                      uploadmturk.PosttoMturk(QID,question,expiry;}};
                   threadmturk.start(); }
      }
      return questionID.unify(getEngine(),new Struct(QID)); }
}
```

Listing 4.7: An excerpt of the *LogicCrowd* extended library in tuProlog

The newly added predicates, `askcrowd/20`, `registercallbacksyn/7`, and `registercallbackasyn/10` are implemented as Java methods and added to the

Prolog library. To execute such methods, this library is loaded into the main program which is running on the Android platform, as shown in Listing 4.8. The program creates the Prolog engine, loads `LogicCrowdLibrary`, defines the theory that contains the extended Prolog meta-interpreter code and user-defined facts, and loads the theory into the engine. For each goal/query read from the standard input of the Android device, the `solve` method is invoked; if multiple solutions exist, the `solveNext` makes it possible to explore alternatives.

The Mediator is not limited to the two classes (i.e., LogicCrowdLibrary and LogicMediator) that communicate with Prolog and Android, as mentioned earlier. There are other important classes. For example, DBhandler deals with the database about the task sent to the crowd. Another is ResultHandler which gathers answers, processes the data and reports the results. The Mediator plays an important role in controlling the execution of both synchronous and asynchronous operations. The process and the implementation of these two operations are presented in the next section.

```
import alice.tuprolog.*;
import alice.tuprolog.lib.*;
public class LogicMediator extends  Fragment {
   private Button.OnClickListener execOnClickListener = new
   OnClickListener(){
      Prolog engine = new Prolog();
      meta = LoadFile("LogicCrowd.pl");
      engine.loadLibrary("LogicCrowdLibrary");
      Theory theory = new Theory(meta);
      engine.setTheory(theory);
      engine.addTheory(userfact);
      goal = queryText;
      SolveInfo info = engine.solve(goal);
      while   (info.isSuccess()){
              Solution =   Solution + solveInfoToString(info);
              if (engine.hasOpenAlternatives()){
              info=engine.solveNext();
              } else break;
      }
      displayResults(solution);}
}
```

Listing 4.8: An excerpt of the main program of the *LogicCrowd* Mediator

4.3.2 Synchronous/Asynchronous Executions

We have described the *LogicCrowd* execution models that include synchronous and asynchronous operation modes in Section 3.3.3, Chapter 3. The synchronous mode refers to the standard Prolog program execution which is running sequentially without any parallel extensions. Thus, to execute a crowd predicate, the evaluation of the next sub-goal is suspended until the system receives the returned results

from the crowd for the current sub-goal. In contrast, the asynchronous operation exploits the multi-threading capability available since *LogicCrowd* is built on top of tuProlog which integrates seamlessly with Java/Android. A new thread is created for each such asynchronous crowd predicate evaluation in order to run independently.

These execution models have been implemented by extending the tuProlog library with new predicates. Listing 4.9 is an excerpt of the library for the synchronous mode, defined as the predicate (`registercallbacksyn/7`). This function handles the returned answers from the crowd and then returns the solution to the main Prolog program. To detail how *LogicCrowd* execution works, we present the models as a sequence diagram. Figure 4.2 shows the sequence diagram in the synchronous mode. Assume that the mobile user sets up goals (in rules, or a *LogicCrowd* program) which can query either the local facts database (i.e., conventional machine query) or the crowd. If the execution starts on a conventional machine query, it interacts synchronously with the knowledge base and returns the solutions to the main goal and/or continues to the next sub-goal(s). As mentioned, the crowd predicates with conditions are executed via the Mediator using the "`askcrowd`" method to send these conditions to the crowdsourcing platforms, and the task is registered by the "`registercallbacksyn`" method. During this step, there may be a time lag between when feedback is sent by crowd and when it is returned to the system. The answer from the crowd will be managed through the "`resulthandler`" method which displays the mobile user's results or returns the results by instantiating Prolog variables.

```
...
public boolean registercallbacksyn_7(Term questionID,Term question,
                              Term askto,Term typeQuestion,
                              Term expiry,Term forward, Var result){
      String sumResult = "";
      ResultHandler crowdResult = new ResultHandler(context,mHandler);
      final long waiting;
      getTermtoString(questionID,question,askto,
                      typeQuestion,expiry,forward);
      waiting = convertTime(expiry);
      try {Thread.sleep(waiting);}
      catch (InterruptedException error)
            {error.printStackTrace();}
      crowdResult.getResult(QID,Askto,QuestionType,Forward,Expiry);
      sumResult = crowdResult.summaryResult(QID,Question,Askto);
      return result.unify(getEngine(),new Struct(sumResult);
}
...
```

Listing 4.9: An excerpt of the registercallbacksyn method in the *LogicCrowd* extended library

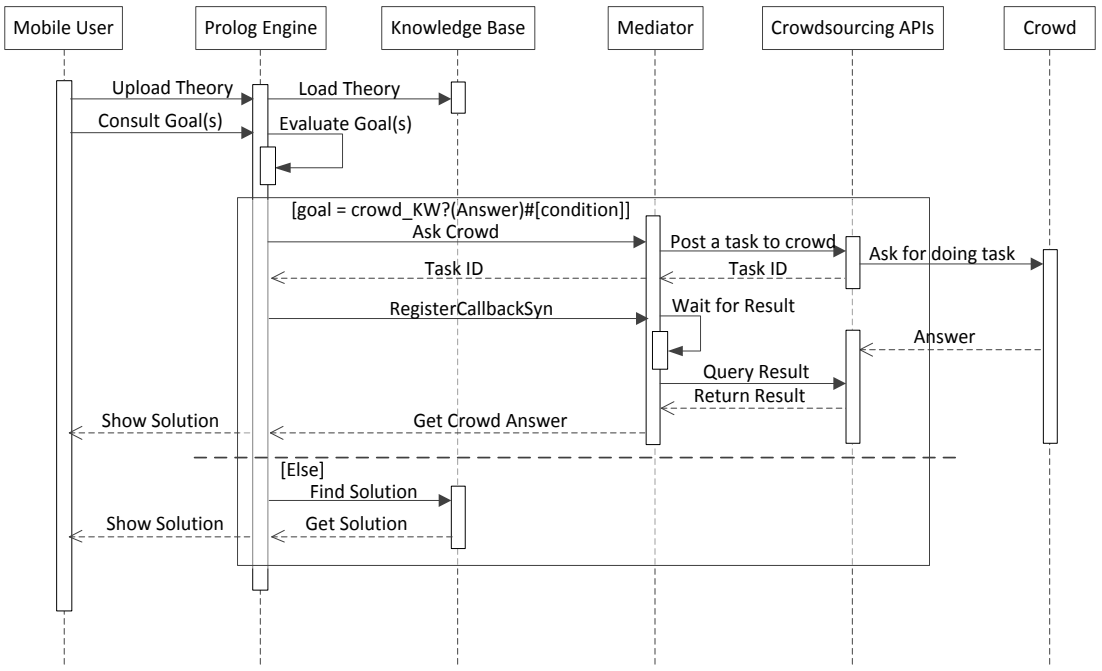


Figure 4.2: A sequence diagram for the synchronous mode

The implementation of the asynchronous execution mode is illustrated in Listing 4.10. The `registercallbackasyn_10` method is written as part of the extended library of tuProlog. A new thread is created for each such asynchronous crowd predicate evaluation to run independently.

```
...
public boolean registercallbackasyn_10(Term argument,Term questionID,
      Term question,Term askto,Term typeQuestion,Term expiry,
      Term forward,Term supportType,Term atLeastValue, Var result) {
   String sumResults = "";
   ResultHandler crowdResult = new ResultHandler(context,mHandler);
   final long waiting;
   getTermtoString(questionID,question,askto,typeQuestion,expiry,
                   forward,supportType,atLeastValue);
   waiting = convertTime(expiry);
   Thread newThread = new Thread(){
         @Override
          public void run() {
             try {
                   Thread.sleep(waiting);}
             catch (InterruptedException error){
                    error.printStackTrace();}
       crowdResult.getResult(QID,Askto,QuestionType,Forward,Expiry);
       sumResult = crowdResult.summaryResult(QID,Question,Askto,ST,
                   AtLeastValue);
       handleCrowdAnswer(QID,sumbresult);
       }};
       threadAsyn.start();
       return result.unify(getEngine(),new Struct("true");
}
...
```

Listing 4.10: An excerpt of the registercallbackasyn method in the *LogicCrowd* extended library

The sequence diagram for the asynchronous operation mode is illustrated in Figure 4.3. The request for tasks can be run in parallel by creating a new thread containing an instance of the Prolog engine; hence, each such goal is executed without delay and without waiting for the results from the crowd. As shown Figure 4.3, the process is similar to that in the synchronous mode, but a difference occurs after calling the "`registercallbackasyn`" method. A background thread has been created to wait for the returned results from the crowd, so that the original thread continues to execute, i.e., in the meantime, if there are further sub-goals, they are executed simultaneously without being blocked by an incomplete processing of a crowd predicate. After the results from the crowd have been returned, a corresponding `handle_crowd_answer/3` rule is then executed to process the crowd's results or simply display the results to the user.

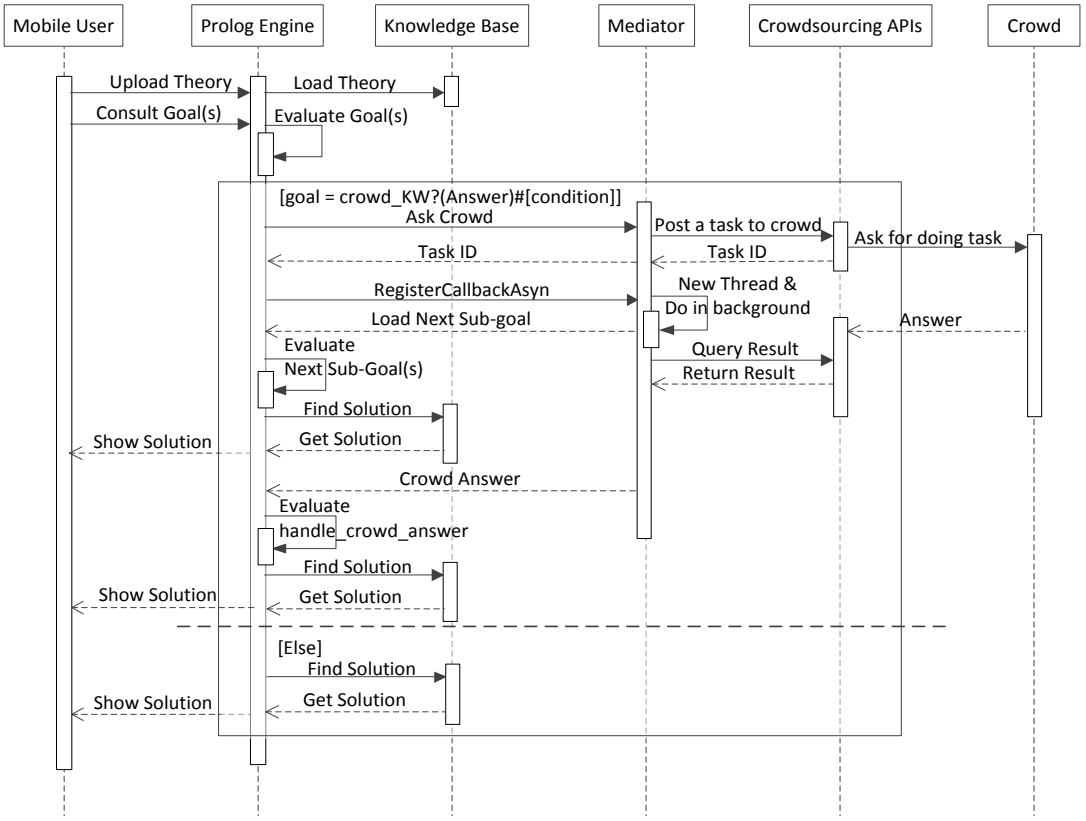


Figure 4.3: A sequence diagram for the asynchronous mode

4.3.3 Crowdsourcing APIs

To communicate with the crowd, the available crowdsourcing APIs platforms are deployed. In our prototype, the *LogicCrowd* program interfaces with three types

of crowd systems, social media networks (Facebook), mobile peer networks (Bluetooth/WiFi Direct), and online crowdsourcing market platforms (MTurk). Using "`askto`" in crowd conditions, *LogicCrowd* enables the Mediator to contact to such crowdsourcing platforms via their crowdsourcing API. In the following sections, the operations of crowdsourcing APIs, employed in *LogicCrowd*, are explained in more detail.

Facebook

Facebook is an online social network service where its members can share messages and status updates with online friends. Developer can integrate Facebook functions into the mobile environment through Facebook's mobile API. The Facebook API is designed to facilitate access to data provided by a user and the user's friends. It also provides a SDK in Android and the Graph API that allows applications to read and write Facebook social graphs.

By integrating with Facebook through the Facebook Android SDK, *LogicCrowd* is able to access the user account information and post items (e.g., texts, photos, tasks) to the user's wall. Listing 4.11 illustrates the *LogicCrowd* code that exploits API methods to post a task and obtain the feedback (or answer) from the crowd. In *LogicCrowd*, before sharing the task on Facebook, authentication is done and permission have been granted by the user. The `LoginButton` class is deployed to access the token and enable the user to log in and out. The Facebook SDK includes an `AsyncTask` class that processes the parameters sent as a part of the URI on a UI Thread. Thus, as soon as the user posts the task to Facebook as a message in the URI the Graph API, the thread in the background creates a connection, checks the user session, posts the task on the user's wall and finally gives the returned feedback when the response has been posted by the user.

```
...
public Response PostQuestion(final Bundle params, final String path){
...
   AsyncTask<Void, Void, Response> task = new AsyncTask<Void, Void,
   Response>() {
      @Override
      protected Response doInBackground(Void... voids) {
         Request request = new Request(Session.getActiveSession(),
                           path,params, HttpMethod.POST);
         return request.executeAndWait();}
  };
  task.execute();
...
```

Listing 4.11: An excerpt of the PostQuestion method in Facebook in *LogicCrowd*

Amazon Mechanical Turk (MTurk) API

MTurk is a crowdsourcing Internet marketplace that enables the usage of human inputs to complete tasks, especially those that cannot be completed by a computer. This type of task is called a micro-tasks. Using MTurk, users/business people can post a wide variety of tasks in a global, diverse and scalable worker community. Workers also have the opportunity to choose the tasks from a large number of tasks of varying nature.

Although the MTurk API has not been designed to support the Android platform, *LogicCrowd* can access the MTurk Web service by exploiting the MTurk PHP library. By creating our own Web server, requests from a *LogicCrowd* application can be sent to the MTurk Web service through the MTurk PHP API. Listing 4.12 shows an excerpt of the *LogicCrowd* program that passes parameters (a task) from the Android platform to the Web server. In this listing, the class "`PostToMturk`" containing the method "`setParams`" is created to set up the connection and send HTTP requests to our Web server. The method "`setParams`" will add the crowd's parameters to create HIT (Human Intelligence Tasks) in MTurk and call the "`makeHttpRequest`" method to post these parameters to the Web server.

```
...
 public class PostToMturk  {
      JSONParser jsonParser = new JSONParser();
       public static Context mCtx;
       public static Handler mHandler;
       public PosttoMturk(Context ctx,Handler handler) {
                 this.mCtx = ctx;
                 mHandler = handler;
 }
 public void setParams(String Accesskey,String Secretkey,String
                      QuestionID,String Question, String Time,
                      String Link) {
      List<NameValuePair> params = new ArrayList<NameValuePair>();
      params.add(new BasicNameValuePair("accesskey",Accesskey));
      params.add(new BasicNameValuePair("secretkey", Secretkey));
      params.add(new BasicNameValuePair("question", Question));
      params.add(new BasicNameValuePair("timetomturk", Time));
      params.add(new BasicNameValuePair("link", Link));
      JSONObject json = jsonParser.makeHttpRequest(url_create_product,
                        "POST",params);}
}
...
```

Listing 4.12: An excerpt of the PostToMturk class in *LogicCrowd*

The MTurk PHP API contains a set of libraries and tools designed to set up and query crowdsourcing tasks with the MTurk Web service. This API allows the Web service to be invoked using its SOAP interface or its REST interface. In our case, *LogicCrowd* uses the REST interface where the response appears in the form

of an XML document that conforms to a specific schema. Based on the operation being invoked, it may be necessary to set certain parameters. The following are the operations from the MTurk API used by *LogicCrowd*.

- `CreateHIT`: The `CreateHIT` operation enables the creation of a HIT based on a set of request parameters (i.e., `HITTypeId, Question, MaxAssignments`). Once a HIT is successfully created, a response from the worker in the form of a HIT element is received. Among other attributes, the HIT element contains the HITId which is used to uniquely identify the HIT for future processing.
- `GETHIT`: *LogicCrowd* uses the `GetHIT` operation to retrieve the details of the specified HIT.
- `GetAssignmentsForHIT`: The `GetAssignmentsForHIT` operation returns all completed assignments for a HIT. It is used to access the results of the HIT. The operation also returns the `GetAssignmentsForHITResult` element which includes the set of assignments of the HIT. Each assignment element includes the `AssignmentId`, the corresponding `WorkerId` and the submitted answers.

Bluetooth API

As mentioned, *LogicCrowd* has been designed to work through M-P2P networks so that peers can send queries to each other via Bluetooth connections. In the prototype, we carry out *LogicCrowd* M-P2P processing via the Bluetooth API on the Android platform. The Bluetooth API allows applications to wirelessly connect to other Bluetooth devices based on unique MACs. In general, the Bluetooth API consists of the four major tasks to communicate using Bluetooth, namely setting up Bluetooth, finding devices, connecting devices, and transferring data between devices. The above features are provided as "Classic Bluetooth". In our implementation, the *LogicCrowd* program works with the Classic Bluetooth version, though Bluetooth Low Energy (BLE) can also be used. The following are the key processes of the *LogicCrowd* program that exploit the Bluetooth APIs to distribute crowd tasks:

- ***Setting up Bluetooth:*** The *LogicCrowd* program exploites the `class BluetoothAdapter` that is available in the `android.bluetooth` package. This class is an entry point for all Bluetooth interaction. By this, all the Bluetooth devices can be discovered. Then, the Bluetooth devices are initialized according to their MAC address and a `BluetoothServerSocket` is finally created to receive responses from the surrounding Bluetooth devices. .

- ***Discovering Peers:*** After setting up the Bluetooth adapter, *LogicCrowd* works by finding remote Bluetooth devices either through device discovery or by querying the list of paired (bonded) devices. The *LogicCrowd* program calls the `startDiscovery` method to start discovering devices. By registering a `BroadcastReceiver` object for the ACTION_FOUND Intent, *LogicCrowd* filters this action to receive information on each device discovered. Then, the name of each device is shown using an ArrayAdapter.
- ***Establishing connection between peers:*** To create a connection between *LogicCrowd* peers, the implementation of both server and client mechanisms is required. Once a peer opens a server socket, the other peer must initiate the connection. The peer who acts as a server holds an open `BluetoothServerSocket` object by calling the `listenUsingRfcommWithServiceRecode(String,UUID)` in order to listen for incoming connection requests. The other peer plays the role of the client and opens an RFCOMM channel to the server. The server socket listens for an incoming connection request from the clients. When one request is accepted, a `BluetoothSocket` object is created.
- ***Distributing crowd's task:*** When two peers have been connected, they are ready to distribute the task. The task is transmitted through `getInputStream` and `getOutputStream` methods by reading and writing data to the stream. The transmission is performed using the `read(byte[])` and `write(byte[])` functions.

4.4 Summary

In summary, this chapter has presented the implementation of the notion of *LogicCrowd* via the *LogicCrowd* framework for mobile applications. A key idea is to utilize the logic programming paradigm to allow the querying and manipulation of knowledge and reasoning, including posing queries to the crowd, in order to solve complex problems through mobile technologies. The *LogicCrowd* framework provides a systematic approach for developing crowdsourcing mobile applications using the logic programming paradigm. Moreover, the *LogicCrowd* framework can bridge the gap between logic programming and crowdsourcing platforms.

In the next chapter, we describe more example applications built using the *LogicCrowd* framework from concept to their implementation. We also illustrate how *LogicCrowd* can be applied for the purpose of increasing convenience of users.

Therefore, useful knowledge as well as helpful observations concerning the *LogicCrowd* concept are derived via these applications.

Chapter 5

Applications using *LogicCrowd*

In the previous chapters, we proposed the *LogicCrowd* concept and described the *LogicCrowd* framework for building applications. In this chapter, we introduce applications that have been built using the *LogicCrowd* framework. In addition, the scenarios of these applications are derived from and add to the motivating scenario in Section 3.1.

Moreover, via these applications, we aim to derive useful insight as well as make beneficial observations concerning the *LogicCrowd* concept. All of the applications described in this chapter have been published in conference proceedings and presented in workshops [Phuttharak and Loke, 2013, 2014a,b].

The chapter is organized as follows. Section 5.1 deals with the implementation of the Crowd Voting and Recommendation application which is the initial idea for the *LogicCrowd* framework. Section 5.2 introduces the Crowd-Powered Comparisons application, which illustrates the crowd unification paradigm in the *LogicCrowd* framework. Section 5.3 presents the AskingCrowd Parking and the Boundary Finding applications which demonstrate *LogicCrowd* in mobile peer-to-peer networks. We summarize the chapter in Section 5.4

5.1 Crowd Voting and Recommendation Systems

We create a small program called *crowd voting and recommendations* which represents a way to make decisions and recommendations in collaborative mobile environments. The details of the work have been described in [Phuttharak and Loke, 2013, 2014a]. The general idea of this example is to clarify the concept of *LogicCrowd* and to illustrate services that can be implemented in *LogicCrowd*.

5.1.1 Scenarios for Crowd Voting and Recommendation

Scenario 1: *LogicCrowd* can be applied in our daily life such as in making decisions. People make hundreds of decisions every day. In some cases, making decisions is straightforward and it simply happens without much effort being needed. However, when situations are more complicated and need group or expert opinion, the voting technique might be a solution for making better decisions. Crowd voting leverages the community's judgment to organize, filter and stack-rank content such as newspaper articles, products, music and movies. It is one of the most popular forms of crowdsourcing, which generates the highest levels of active participation or engagement.

This scenario invoking mobile crowdsourcing relates to ranking products. This example can be applied in retail, such as retail in a research marketing field that determines how to market products. The application aims to rank popular handbags by aggregating the power of human and machine computations. This scenario deploys *LogicCrowd* for crowd voting to decide which handbags are the most popular; a user is thus able to gauge others' perspectives for products before making investment-related decisions about a new purchase.

Scenario 2: We exploit *LogicCrowd* as a tool to predict the rating or preferences that the crowd would give to an item, called a crowd recommendation system. Recommendations are key for information retrieval and content discovery in today's information-rich environment, to effectively filter out the pieces of information that are considered most appropriate for the users.

In the second scenario, a *LogicCrowd* program is developed to implement a recommendation system for shopping in a big mall. Consider a couple shopping in a large mall. The couple is looking for a lady's dress and sports shoes at reasonable prices. They do not want to spend much time looking for them and want to find

the ones currently on sale in any shop(s) in that the mall. In this case, an application programmed in *LogicCrowd* can be used to serve their needs. With support mobility, *LogicCrowd* allows users to connect with surrounding mobile devices via a Bluetooth connection during the period they are shopping in that area. Hence, the couple can simply send the nearby crowd their request for recommendations based on their specific requirements.

5.1.2 Demonstration

In *LogicCrowd*, we implemented the applications based on the two case scenarios above. The first scenario is the application of *LogicCrowd* to ask the crowd to rank products by using a voting technique. The program below shows the rule/goal written in the Prolog style.

The rule aims to rank popular handbags according to the number of votes received from user's friends.

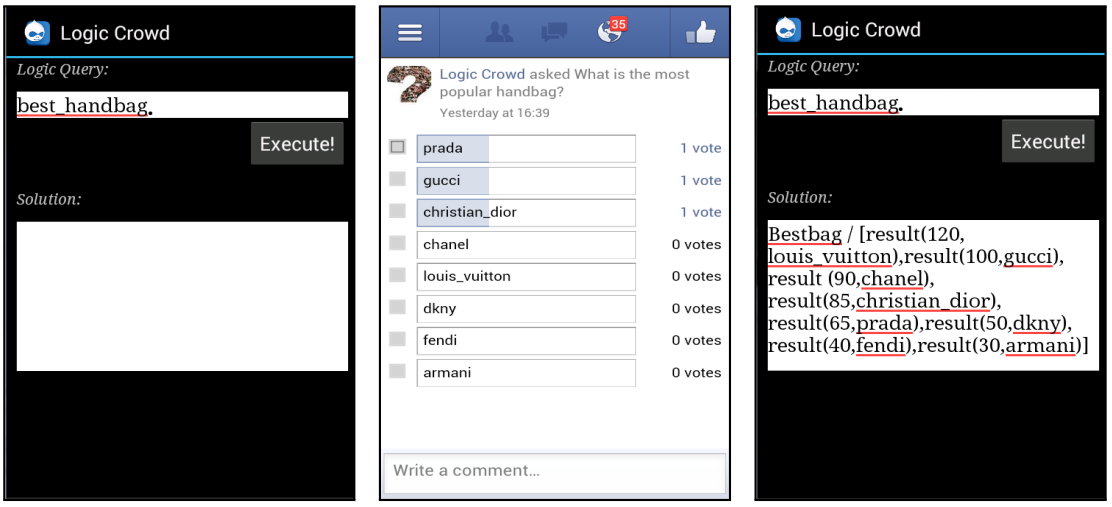
```
best_handbag:-
          findall(X, brand(X), Handbag),
          besthandbag?(_)#[asyn, asktype("choice"),
               question("What is the most popular handbag?"),
               options(Handbag),
               askto([facebook]),
               expiry("5,0,0")].
handle_crowd_answer(besthandbag,CrowdResult,Bestbag):-
          quicksort(CrowdResult,"@>",Bestbag).
```

The rule `best_handbag/0` consists of two sub-goals. In the first sub-goal, `findall/3` predicate is a built-in predicate that collects all the solutions to a query and puts them in a single list. Thus, all brands of handbags, collected from the local knowledge base, are put into the list in the "`Handbag`" variable. Next the second sub-goal shows the crowd predicate to query the crowd. This means that the users would like to create and propagate the task to the crowd. In the crowd predicate, the crowd conditions are explained as follows:

- operation: asyn (asynchronous version).
- question: Which is the most popular handbag?
- options: as in the list contained in the Handbag Variable
- ask to: Facebook.
- expiry: 5 hours.

After executing the crowd sub-goal shown in Figure 5.1(a), the question with these conditions then appears on Facebook as illustrated in Figure 5.1(b). With the asynchronous operation, if there are goals after the crowd predicate, these goals can execute without the delay of waiting for the results from the crowd. After a while, several Facebook friends are anticipated to answer the request by

voting for the product(s) they have used or prefer. Within five hours, on expiry, the results are returned to *LogicCrowd*. Then, the `handle_crowd_answer/3` predicate is automatically executed. In this scenario, `handle_crowd_answer/3` is programmed to sort the list of handbags in order, from the highest to the lowest voting scores. Figure 5.1(c) displays the final results consisting of a list of products and scores of the handbag brands as voted by the crowd. Note that the display can be pretty-formatted depending on the user.



(a) Execute Goal **(b)** Post Question to Crowd **(c)** Display Results

Figure 5.1: Sending a handbag brand ranking request to Facebook and the results

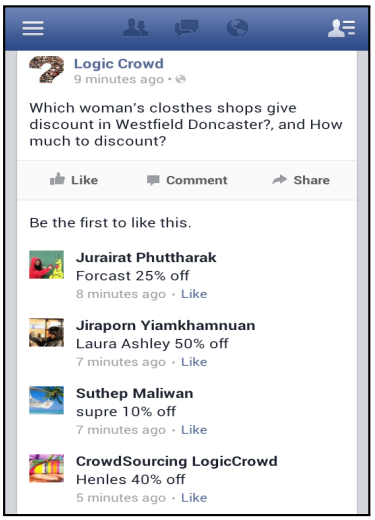
In the second scenario, a *LogicCrowd* program is applied to implement a recommendation system for a couple shopping in a large mall. By setting the rule as shown below, the couple can simply send the crowd their request for the recommendation with their specific requirements.

```
recommend(Clothes,Shoes):-
        clothesShop?(Clothes)#[syn, asktype("message"),
            question("Which woman's closthes shops give discount
                    in Westfield Doncaster?, and How much to
                    discount?"),
            askto([facebook,bluetooth]),
            expiry("0,10,0")],
        shoeShop?(Shoes)#[syn, asktype("message"),
            question("Which shoes shops give discount in
                    Westfield Doncaster?,and How much to
                    discount?"),
            askto([facebook,bluetooth]),
            expiry("0,10,0")].
```

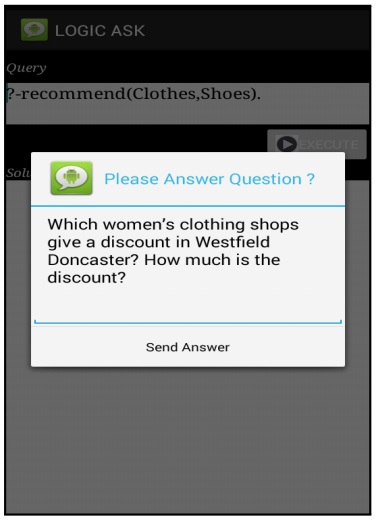
The aim of the goal is to request recommendations for the shops on the basis of the restricted conditions that are: 1) women's clothing shops and 2) shoe shops. In this rule, there are two crowd predicates: the first one with crowd keyword

"`clothesShop`" asks the crowd to recommend women's clothing shops and the second with crowd keyword "`shoeShop`" asks for recommendations on shoe shops, with both shops being located in the Westfield Doncaster shopping center, by sending the query to friends via Facebook and Bluetooth and waiting for ten minutes for the returned results. Both crowd predicates were executed in the synchronous mode so that the second predicate call is suspended until the first predicate call has completed.

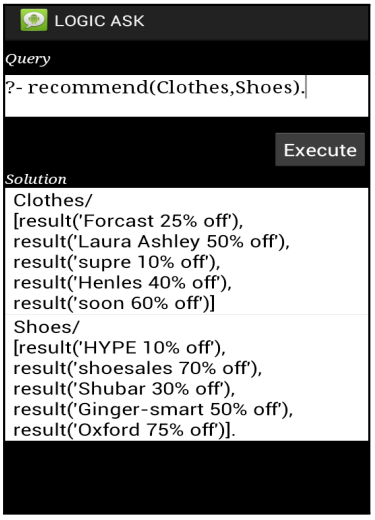
(a) Execute Goal



(b) Ask Question via Facebook



(c) Ask Question via Bluetooth



(d) Display Result

Figure 5.2: Sending the requests to Facebook and to devices via Bluetooth and showing the results

After executing the crowd sub-goal shown in Figure 5.2(a), the question with the above conditions then appear on both Facebook over the Internet, and friends' devices connected via Bluetooth, as illustrated in Figures 5.2(b) and 5.2(c). A while later, several friends are anticipated to answer the questions by mentioning

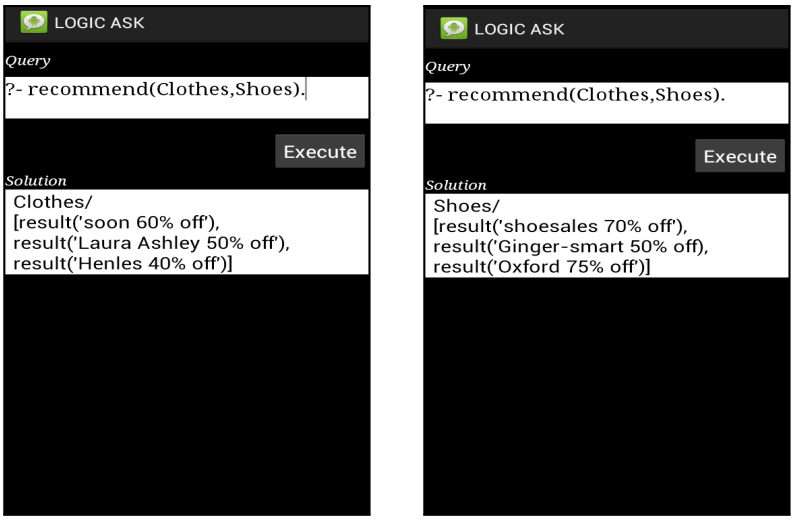
shop(s) which offer discounts. Within 10 minutes, i.e. the expiry time, the result is returned to *LogicCrowd* in the originating device. The system then returns the results to the main goal. Figure 5.2(d) displays the result consisting of a list of the women's clothing shops and a list of the shoe shops with discount details.

We can extend the first scenario to be more flexible by using the asynchronous operation mode, as shown below. Moreover, it illustrates how *LogicCrowd* works in both synchronous and asynchronous mode in one program. The `recommend/1` rule is a goal to ask the crowd for recommendations on the shops in a particular shopping mall. With the asynchronous execution mode in the first sub-goal, the next sub-goal can be executed without the delay of waiting for the results from the first one. Two questions were posted on Facebook and were sent to nearby friends via Bluetooth.

```
recommend(Clothes,Shoes):-
          clothesShop?(_)#[asyn,asktype("message"),
               question("Which woman's closthes shops give discount
                        in Westfield Doncaster?, and How much to
                        discount?"),
               askto([facebook,bluetooth]),
               expiry("0,10,0")],
          shoeShop?(ShoeList,Sdiscount)#[syn, asktype("message"),
               question("Which shoes shops give discount in
                        Westfield Doncaster?,and How much to
                        discount?"),
               askto([facebook,bluetooth]),
               expiry("0,10,0")],
          select_shop(Shoes,ShoeList,Sdiscount),
          Sdiscount > 50%.
handle_crowd_answer(clothesShop,ClothesList,Cdiscount,Clothes):-
          select_shop(List,ClothesList,Cdisconunt),
          Cdiscount > 30%,
          quicksort(Clothes,"@>",List).
```

After a while, several Facebook friends and friends on Bluetooth might answer the request by suggesting shop(s) that offer a discount. Within ten minutes, on expiry, the result is returned to *LogicCrowd*.

Then, the `handle_crowd_answer/4` predicate is automatically executed in the first sub-goal. In this scenario, `handle_crowd_answer/4` is programmed to first select women's clothing shops with offers of more than 15% discount and then sort the list of these shops in order from the highest to lowest discount. The result from the second sub-goal was passed to the next sub-goal `(select_shop/3)` that chooses shoe shops with discount offers of more than 30%. Figure 5.3(a) and 5.3(b) display the final results consisting of a list of women's clothing and shoe shops. Note that the display can, of course, be pretty-formatted depending on the user.



(a) Display 1^{st} result (Asynchronous mode)

(b) Display 2^{nd} result (Synchronous Mode)

Figure 5.3: The complex scenario - sending the requests to the crowd and showing the results

5.2 Crowd-Powered Comparisons

Comparing data is difficult or impossible to encode in computer algorithms. For example, if given photos, it is very simple for a person to tell whether the pictures are the same or not. The Crowd-Powered Comparisons (CPC) application implements the crowd unification paradigm in the *LogicCrowd* framework. As mentioned in Section 3.4, *LogicCrowd* has an extended unification scheme in logic programming for the purpose of unifying arguments in predicates when traditional unification might not work. The motivation of CPC employs the power of people to solve such unification problems. Additionally, mixing the mobile platform and the crowdsourcing model can potentially offer vast resources for computations and be used to build expert systems that involve the crowd in helping to answer queries.

5.2.1 Usage Scenario

The CPC application exploits the crowd unification scheme that has been designed to process complex queries with humans. In Section 3.4.3, we have introduced the syntax of crowd unification which can be used seamlessly with Prolog and is written in Prolog rules. Moreover, the CPC deploys the *LogicCrowd* program to interface with crowdsourced data such as Facebook, MTurk and P2P networks.

To illustrate, the crowd unification in *LogicCrowd*, the following example shows a simple crowd unification query that asks the crowd to unify two pictures.

```
?-"photoA.jpg"*= "photoB.jpg".
```

The evaluation process of crowd unification is presented as follows. The *LogicCrowd* meta-interpreter interprets the operator "`*=`" which denotes the unification of the terms by asking the crowd. Also, this predicate can be written in prefix form: `*=("photoA.jpg","photoB.jpg")`. The `*=/2` predicates are executed via the Mediator using the "`askcrowd`" method to send two pictures to the specified social network and P2P network (via Bluetooth) surrounding the user. Next, the "`registercallback`" method registers the query and returns the results. After posting the query to the crowd (i.e., Facebook via WiFi/4G, and/or via Bluetooth), the answers from the crowd are managed through the "`resulthandler`" method in order to display answers to the mobile user or to return the answers in instantiated Prolog variables after the default expiration time. From the above example, the crowd unification query can be written in a more flexible manner by requesting the level of crowd support represented in Section 3.4.3. There are four cases to specify pairs of term equivalence, namely 1) all, 2) majority, 3) at least, and 4) trusted. The following examples show the crowd unification query with the required level of crowd support requested in the Prolog style.

```
"photoA.jpg"*= "photoB.jpg"@all.                    (1)
"photoA.jpg"*= "photoB.jpg"@major.                  (2)
"photoA.jpg"*= "photoB.jpg"@at(10).                 (3)
"photoA.jpg"*= "photoB.jpg"@trust.                  (4)
```

As mentioned in Section 3.4.3, these queries have been written in the form `X *= Y@Type or *=(X,Y)@Type` where "`Type`" denotes one of the four different degrees of crowd support. This technique is a workflow design for quality control mechanisms. It aims to ask a sufficient number of workers to perform the same task independently to determine the quality and reliability of the answers received. In the first query with operator `@all, photos A` and `B` unify if and only if all crowd responses to the query say that the two photos are unifiable (or the same). This operator can be omitted, and the form `x*=y` defaults to the `@all` operator. The second example query with `@major` operator illustrates that two photos unify if the majority of crowd responses say so. The third query with the `@at` operator shows the requirement for crowd unification that at least 10 of the crowd responses say that two terms are the same. In the last query, the operator `@trust` represents asking the (predefined) list of trusted persons (who could be in the list of friends in Facebook or using paired devices via Bluetooth). When using this operator,

it means that the answers from the trusted group have a higher priority than the answers from outsiders of the group. Hence, the two images will unify if the majority of the trusted group say they are equivalent, even though outsiders of the group do not agree.

As mentioned, the crowd unification query can be written in Prolog rules. The following example shows the simple rule/goal `grandparents/2` which aims to define someone to be someone's grandparents.

```
grandparents(A,C):- parents(A,X), parents(Y,C), X*=Y@major.
```

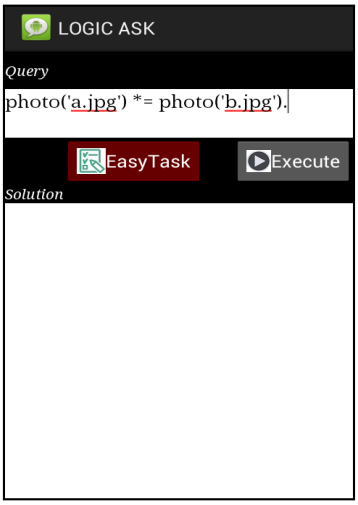
This rule will succeed if A is a parent of X, Y is a parent of C, and X and Y are unified by a crowd majority. The variables X and Y are available to be instantiated, e.g., `X = "photoX.jpg"`. The type of data is "image" which is collected in a knowledge base that functions as a stored path or filename. The photos X and Y will be propagated to the crowd in order to seek unification about them. If the results from the crowd are true, then the goal has been achieved. However, the last sub-goal `*=/2` predicate represents the type of crowd support "`@major`" stating that two photos unify if and only if the majority of people who accept the query say that X and Y are photos of the same person.

Interfacing with multiple crowdsourcing platforms is one of the essential features of a *LogicCrowd* program that, for instance, integrates with Facebook, a popular free social network, and connects to mobile devices via Bluetooth in the vicinity. *LogicCrowd* has been enhanced by adding the capability to access a popular crowdsourcing marketplace, Amazon's Mechanical Turk, also called MTurk. By exploiting the strengths of *LogicCrowd* for rule-based declarative programming, crowd unification can be used to propagate tasks widely to reach reliable and willing crowd sources who might return quality results.

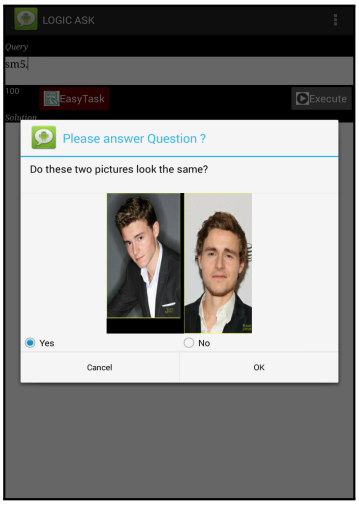
5.2.2 Demonstration

We demonstrate the CPC application within three different case scenarios. Figure 5.4(a) shows a simple crowd unification query in the CPC application. After executing the crowd unification goal, the question then appears on peer mobiles through Bluetooth connections on the query originator's Facebook wall (to reach his/her Facebook friends), and also on MTurk as a HIT (Human Intelligence Task), as illustrated in Figures 5.4(b), 5.4(c) and 5.4(d) respectively. After a while, peers or workers answer the request by comparing the two photos. Within 30 minutes (a

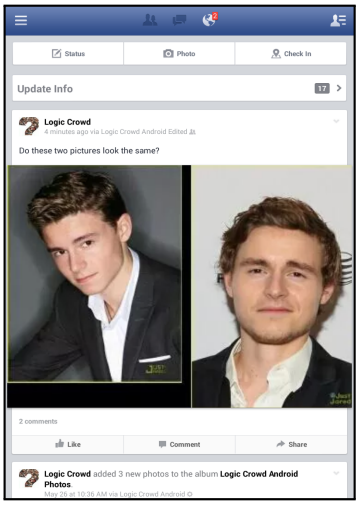
time set in the query), the result is returned to the query originator in *LogicCrowd*. This scenario is programmed to display the results, as shown in Figure 5.4(e).



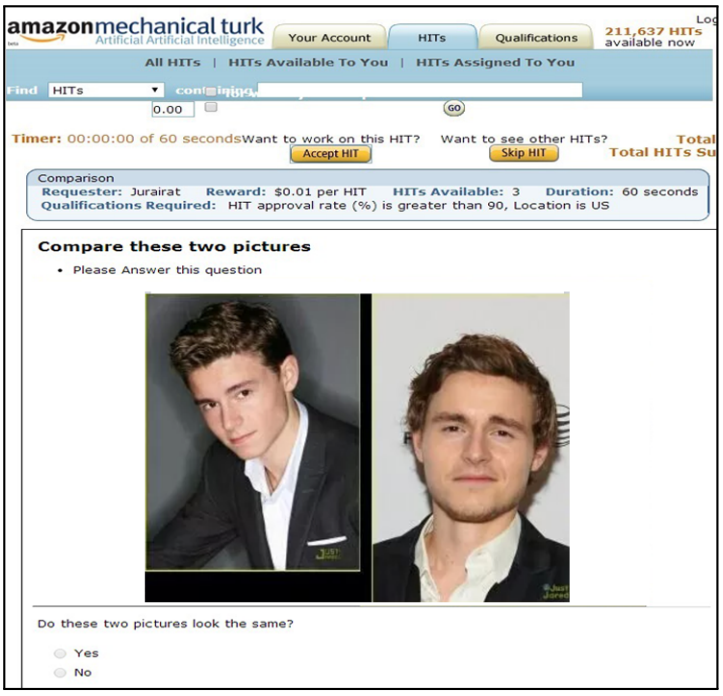
(a) Execute Goal



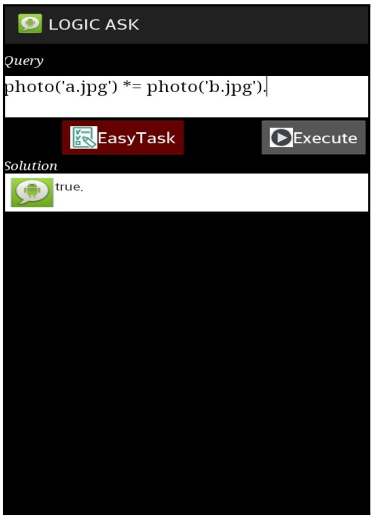
(b) Send via Bluetooth



(c) Post to Facebook



(d) Post to MTurk



(e) Displaying Results

Figure 5.4: Distributing a task to the crowd to unify the photos

Crowd unification can be used with the crowd predicate of *LogicCrowd* program in order to effectively solve complex problems. The following scenario illustrates the use of both crowd unification and crowd predicates in the same program. The rule aims to translate the paragraph or text from non-English to English. In order to increase the reliability of the returned results, crowd unification has been deployed to ask the crowd to unify the original text and the translated text again.

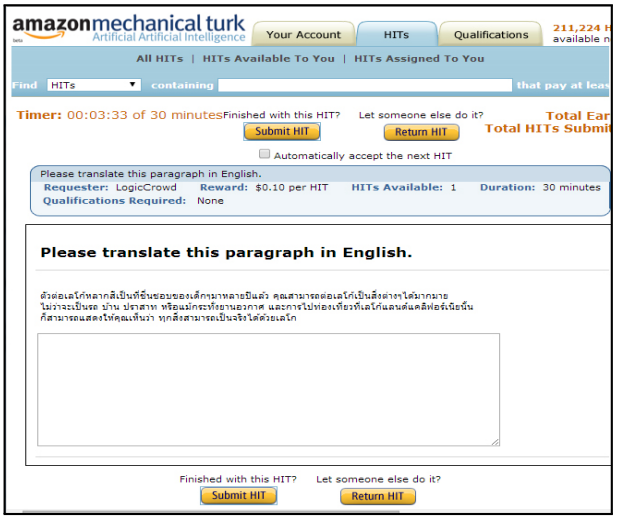
We have applied a divide-and-conquer and iterative workflow pattern to solve this task. The task has been split into two subtasks, first to translate from non-English languages to the English language and second, to verify the texts. Then the solutions to these subtasks are integrated into one solution. The iterative workflow has been used on this task to improve the quality of results. The Translate-and-Verify workflow pattern has been introduced to divide a complex task into a series of generation and review stages. Rather than asking a single crowd worker to translate and proofread the text, the rule below is designed to recruit a group of workers to filter incorrect results and thus produce reliable answers.

```
translate(Text):-
      toEng?(Text)#[syn,asktype("message"),
            question("Please translate this paragraph in English."),
            file(Para),
            askto([facebook,bluetooth]),
            expiry("2,0,0")],
      *=(Para,Text)@at(3).
```

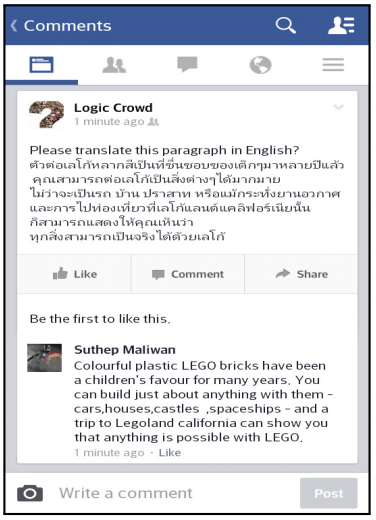
In the first sub-goal of `translate/1` rule, the crowd predicate identified using the crowd keyword "`toEng`" is called to ask the crowd (via Facebook and MTurk) for translation. The question and text paragraph (in a non-English language) are distributed to the crowd in a synchronous mode in the maximum waiting time of 2 hours, as shown in Figures 5.5(a) and 5.5(b). With synchronous operation, if there are any goals after the crowd predicate, these goals will be blocked until the results are returned from the crowd. After a while, friends from Facebook or workers from MTurk answer the request by translating the paragraph into English. On time expiry, the result is returned and passed to the next sub-goal. The `*=/2` predicate then attempts to unify the original text and translate the text by distributing both the original text and the translated version of the text to the crowd, as illustrated in Figure 5.5(c). With the type of crowd support "`@at(3)`", the result will be true when at least three people agree that both texts convey the same meaning. The agreement of a group, rather than one person, is expected to increase the reliability of the answer.

Furthermore, crowd unification is able to be applied to a wide variety of applications. *LogicCrowd* programs, being implemented on a mobile device, integrate seamlessly with the extensive functionalities of the mobile phone, e.g., location-awareness. Due to the fact that *LogicCrowd* is programmed based on Prolog, there is an opportunity for the applications to include reasoning above and beyond the aforementioned example tasks. In the next scenario, we create a recommendation application by using the crowd unification scheme and the idea of context of the

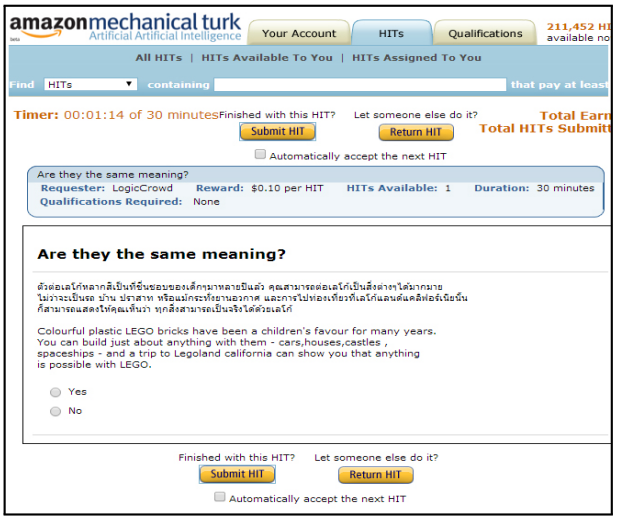
mobile environment.



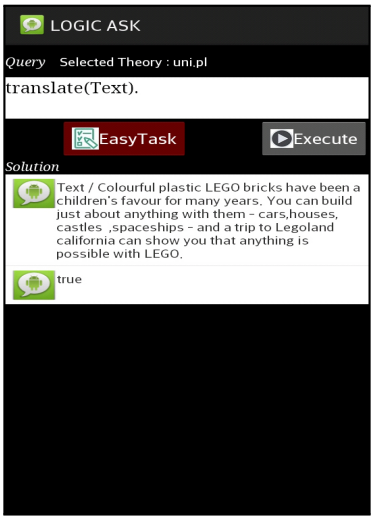
(a) Post HITs to MTurk (1^{st} sub-goal)



(b) Post Task to Facebook



(c) Post HITs to MTurk (2^{nd} sub-goal)



(d) Displaying Results

Figure 5.5: Translating and verifying a translation

Today, many people use social networks such as Facebook or Twitter, and enjoy sharing with friends the places where they are located by checking in, or they may update their status by showing or posting photos of activities they are doing on their own walls. In light of these benefits, we have designed what we call a photo-based activity recommendation application which can be used by a user to obtain recommendations from the crowd about interesting activities by making use of photos. The following program aims to suggest activities surrounding a user's location that are popular and not too expensive and to show recommendations about them on the map.

```
recomAct:- getLoc(Loc),
           cost(Photo1,Cost,Loc),
           Cost<=200,
           popactivity(Photo2,Loc),
           Photo1*=Photo2@major,
           showMap(Photo1,Loc).
```

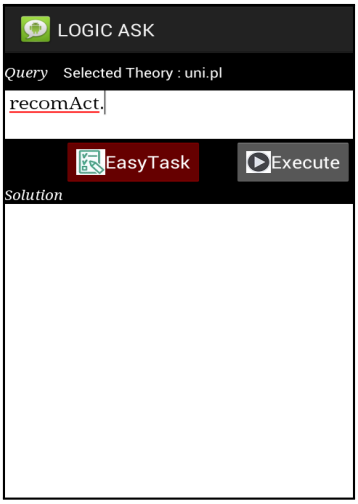
The `recomAct/1` rules consist of 4 sub-goals including `cost/3`, `popactivity/2`, `*=/2`, and `showMap/2` predicates. The `cost/3` predicate will find a photo and the cost of an activity which is located in the mobile user's location (stored in the `Loc` variable). In this case, the cost is conditioned to be less than and equal to 200 dollars, and hence, only activities that fit this condition will be selected. In the next sub-goal, the `popactivity/2` aims to show photos of famous activities in the `Loc` area by searching from a local database (assumed pre-installed). A popular activity can be recorded in the predicate `popactivity/2` in the knowledge base before use by calling `activity/2` shown in the following rule.

```
activity(P,Loc):-
          pop?(P)#[syn,asktype("upload"),
                question("Upload a pic of popular activity
                          around there."),file(P),
                askto([facebook,bluetooth]),
                locatedin(Loc),
                expiry("1,0,0"),
                determine(5)],
                assertAct(P,Loc).
assertAct([],Location):-!.
assertAct(Photo,Location):-
          Photo=..[_H|[Pic,Body]],
          assert(popactivity(Pic,Location)),
          assertAct(Body,Location).
```

In the first sub-goal of the `activity/2` rule, the crowd predicate identified using the crowd keyword "`pop`" is called as the open predicate to peers. It aims to find a list of popular activities (P) in the specific location (`Loc`). The query asks the crowd to post a photo of activities which have been regarded as popular/well-known in the particular `Loc` area via Facebook friends and Bluetooth connections to people nearby within a range, and waits for one hour for the results to be returned as shown in Figure 5.6(b) and 5.6(c). After expiry, the photos and their location will be inserted into the local database by calling `assertAct/2` rule in predicate `popactivity/2`.

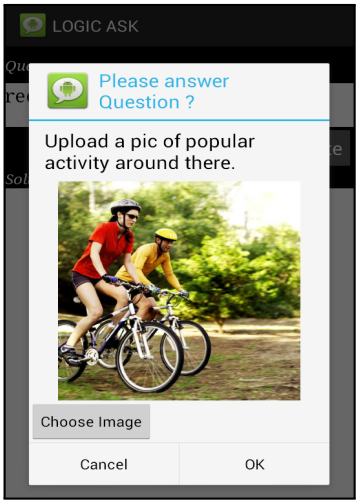
However, this query returns results before the expiry when the number of crowd responses reaches 5 because this query is defined with the crowd response counting operator "`determine(5)`". In the `recomAct/1` rule, there is a crowd

unification predicate (`*=/2`) which aims to send the photos to the crowd in order to match the activities identified via photos. In this case, we send these photos to MTurk where the workers are able to unify the two photos within 30 minutes (by a default setting) as shown in Figure 5.6(d). With the type of crowd support "`@major`", the result will be true when the majority of people agree that both photos describe the same activity. After a while, the results are gathered and sent to the `showMap/2` predicate, which displays the suggested activities at the user's location, as illustrated in Figure 5.6(e) (Prolog backtracking or `findAll` can be used to retrieve more than one activity, if available).



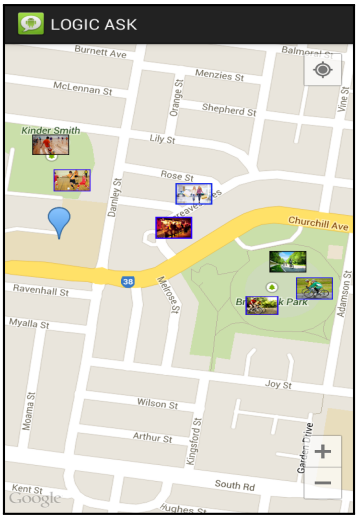
(a) Execute Goal

(b) Post activities to Facebook



(c) Send activity via Bluetooth

(d) Post to MTurk



(e) Show results on a map

Figure 5.6: A photo-based crowd-powered activity recommendation app

5.3 AskCrowd Parking and Boundary Finding

We introduce two applications, named AskCrowd Parking and Boundary Finding. These two applications are used to demonstrate *LogicCrowd* in M-P2P (mobile peer-to-peer) networks. The idea of AskCrowd Parking is to deploy the power of the crowd in the mobile environment for the purpose of solving or alleviating the parking problem. The boundary Finding application aims to find the boundary of effective zones using GPS coordinates and Google Maps API in mobile devices in order to create a destination point or boundary area. The Boundary Finding is published in [Phuttharak and Loke, 2014b].

5.3.1 AskCrowd Parking Application

In big cities or busy areas such as hospitals or universities, searching for vacant parking spots is common. In [Shoup, 2005], the study shows that over 8% of the traffic in a congested area is caused by vehicles cruising in search of available parking spots when driving to work or into a congested city. Some cities try to mitigate this issue by deploying smart parking systems that use information and communication technologies such as sensors to collect and distribute real-time data about parking vacancies and provide routes so that drivers are able to find parking spots more quickly. Although the use of these parking systems provides benefits, the huge cost caused by the initial investment and maintenance inhibits the widespread adoption of these systems in most other cities.

The idea of the AskCrowd Parking application is to deploy the properties of crowdsourcing in the context of smart parking. It uses the information collected through the crowd to find parking spots and provide guidance. This design is able to lower the cost of installing and maintaining a dedicated infrastructure. More specifically, we implement the system using *LogicCrowd* in M-P2P network framework which can be useful in peer-to-peer computing for querying and multicasting tasks shared over peer networks via mobile communication technologies such as Wi-Fi Direct, Bluetooth, and LTE Direct.

There has been increasing interest in the use of mobile crowdsourcing for parking systems. Many mobile applications including Streetline[1], VoicePack[2], and PocketParker [Nandugudi et al., 2014] have been recently released by seeking help

[1]http://www.streetline.com/
[2]http://www.voicepark.org/

from the general public to locate available parking spaces. The systems in [Chinrungrueng et al., 2007] and SPARK [Lu et al., 2009b] employ wireless sensors and VANET (Vehicular Ad hoc NETwork) devices respectively to collect and disseminate information about parking availability in order to help drivers find vacant parking spaces. The limitation in these systems, however, appears to be that they do not provide any information about a parking space being taken. Parking availability information, by itself, is not particularly useful since empty spots in crowded areas are quickly consumed.

CrowdPark [Yan et al., 2011] achieves parking reservation by crowdsourcing information about when parking resources will be available, and using this availability information to help other users find parking spots. The combination approaches including incentive designs, sensor data processing techniques, and navigation-based tools have been used to address the parking reservation issues. However, reservation-based solutions might make drivers' operation more complicated and could collapse if only a few drivers participate. Recently, the research in [Mathur et al., 2010] using distance sensors attached to taxis to obtain parking availability information has provided promising results, even though this approach is more expensive and less scalable than using mobile phones. Crowd-based Smart Parking [Chen et al., 2012] offers higher agility, lower cost and larger coverage because it employs a large number of mobile phone users. However, these systems require a high initial investment and maintenance cost. Our approach considers a distributed problem-solving model to coordinate crowd participants in order to make mobile crowdsourcing a cheap but effective solution to find parking spots.

5.3.1.1 Scenario

To attain a better understanding of AskCrowd Parking, we describe a scenario which explains the process of seeking a parking spot using *LogicCrowd*. Jason and his family are going shopping at the Queen Victoria market over the weekend. When shopping at this market at this time, they cannot avoid the problem of finding a parking sport, as parking is usually scarce. In addition to this, finding a parking spot in such a large area which is divided into many zones also adds difficulty to the task. Jason has previously experienced this problem, driving for over an hour around the area to find a parking spot, which is a waste of fuel.

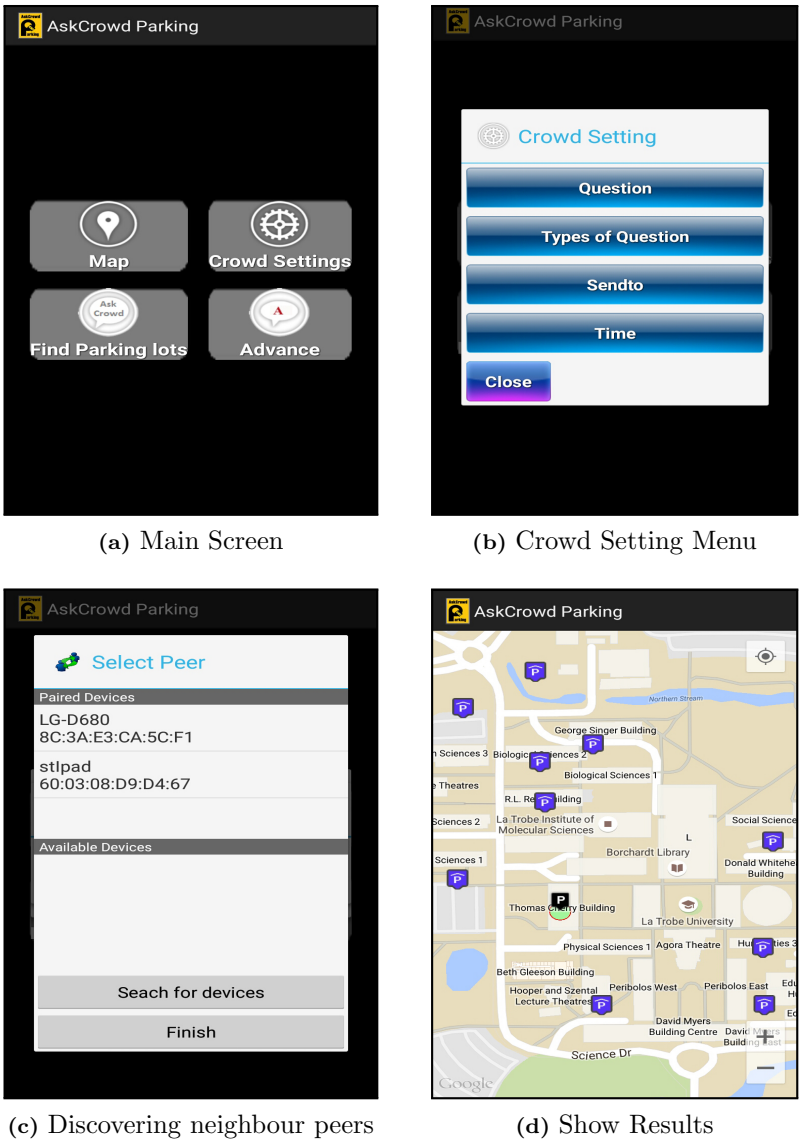
The problematic situation improved when Jason decided to exploit the AskCrowd Parking application to search a parking spot. He sends his request

for a vacant parking spot to other peers' mobile devices in the vicinity via Bluetooth and Wi-Fi Direct. Since these mobile devices or peers are allowed to forward his request to others, there is a possibility that Jason's request reaches a larger number of peers around this particular market area. After a while, he receives results that show the position of the available parking spot's GPS coordinates via Google Maps. He can then drive directly to the particular parking spot, saving time and money on fuel, or he can at least improve his chances of finding car park by knowing roughly which part of the large carpark to go to.

5.3.1.2 Implementation

We created a prototype called AskCrowd Parking that utilizes the *LogicCrowd* framework. The prototype is able to run in two different modes, the centralized and decentralized crowdsourcing models, as discussed in Section 3.5.1. The centralized crowdsourcing mode enables the origin peer (the task's owner) to fully control the distribution of tasks to his/her peers, whereas the decentralized crowdsourcing mode allows each peer to distribute their tasks to friends without any control from the originating peer.

Figure 5.7(a) shows the screenshots for the AskCrowd Parking application in the scenario. Before the user sends the query to find an available parking space, he/she must assign the query's conditions such as the question, expiry time for the question, execution operations these being either synchronous or asynchronous, and mode type (either centralized or decentralized crowdsourcing modes). These conditions are defined in the menu crowd settings, illustrated in Figure 5.7(b). After setting these conditions, the question is ready to be posed to the crowd to find an available parking spot. In Figure 5.7(c), the query has been executed and the peers/devices have been discovered and shown on the screen. Then the results are processed and the position is shown on Google Maps, as displayed in Figure 5.7(d).

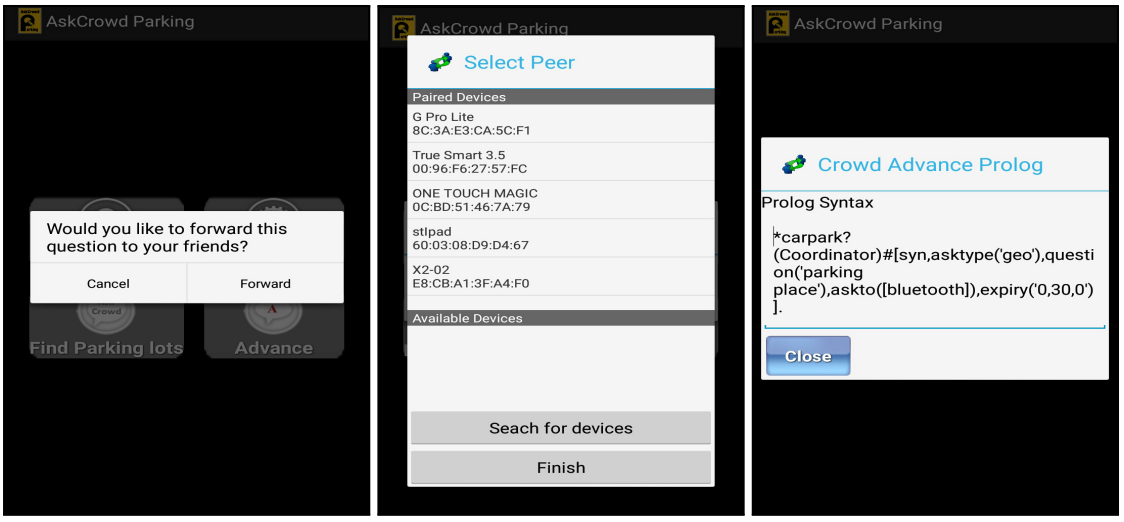


(a) Main Screen **(b)** Crowd Setting Menu

(c) Discovering neighbour peers **(d)** Show Results

Figure 5.7: AskingCrowd Parking application screenshots

Figure 5.8(a) shows how the decentralized crowdsourcing mode works. By selecting the forwarding mode in a crowd setting menu, the decentralized mode is enabled. The origin peer executes the query, allowing its friends to forward it to others, as displayed in Figure 5.8(b). After expiry, the results are collected and returned to the origin peer by showing the particular position on the map as shown in Figure 5.7(d). In the scenario, Jason can simply search for parking spots by sending his request via Bluetooth. The AskingCrowd Parking application eventually returns him the directions of the available parking spot to Jason. In addition to this benefit, a tech-savvy user is able to manually code up their own request/query by writing in Prolog syntax, as shown in Figure 5.8(c).



(a) Forwarding Menu **(b)** Distributing task among peers **(c)** Writing own rule

Figure 5.8: AskingCrowd Parking application screenshots – Decentralized crowdsourcing mode

5.3.2 Boundary Finding Application

Over the last decade, thousands of people have suffered and died from serious disease outbreaks such as Ebola and SARs. Recently, Middle East Respiratory syndrome (MERs) has been another fatal disease causing acute and serious illness in human beings. Furthermore, recent natural disasters, such as the earthquake in Nepal or the devastating heat wave in Pakistan have been reported as causes of great suffering to people' lives. In the face of such natural disastrous situations as tsunamis and bush fires, evacuating people from dangerous sites as fast as possible seems to be the only way for their survival. Identifying or zoning an area which is considered to be a dangerous area, is necessary. The ability to identify a zone boundary is also important in our daily life. For example, identifying a school zone can be useful to remind motorists to slow down when passing this area in order to ensure the safety of students. Finding the boundary of effective zones can thus be regarded as useful but challenging.

In light of the above, we are highly aware of the importance of finding the boundary and thus aim to design the application prototype for it by deploying *LogicCrowd* and seeking help from people. The idea of the Boundary Finding application exploits decentralized and distributed computing [Duckham, 2013] with geographic information, leading to highly efficient localized solutions, not only for network functions, but for all kinds of computations and tasks. In this application,

dealing with spatial computing, the decentralized approach is a tool for efficient and reliable problem solving in distributed computational environments such as mobile sensor networks, delay-tolerant networks, and mobile ad hoc networks. To find the boundary of the area, an algorithm is deployed. In our case, the convex hull concept is used to compute the boundary (for more details see Section 5.3.2.2).

5.3.2.1 Problem and Scenario

To better understand zone boundary finding, we illustrate the use case scenario for the Boundary Finding application utilizing a *LogicCrowd* program. Suppose Jim wants to attend a music concert held in a football stadium. Due to the popularity of the singer, an audience of more than 1,000 people, including Jim, attend this concert. However, Jim needs to leave the concert while it is still in progress. Leaving the concert surrounded by such a huge crowd causes difficulties for him. Using the Boundary Finding application, Jim can finally find the nearest exit.

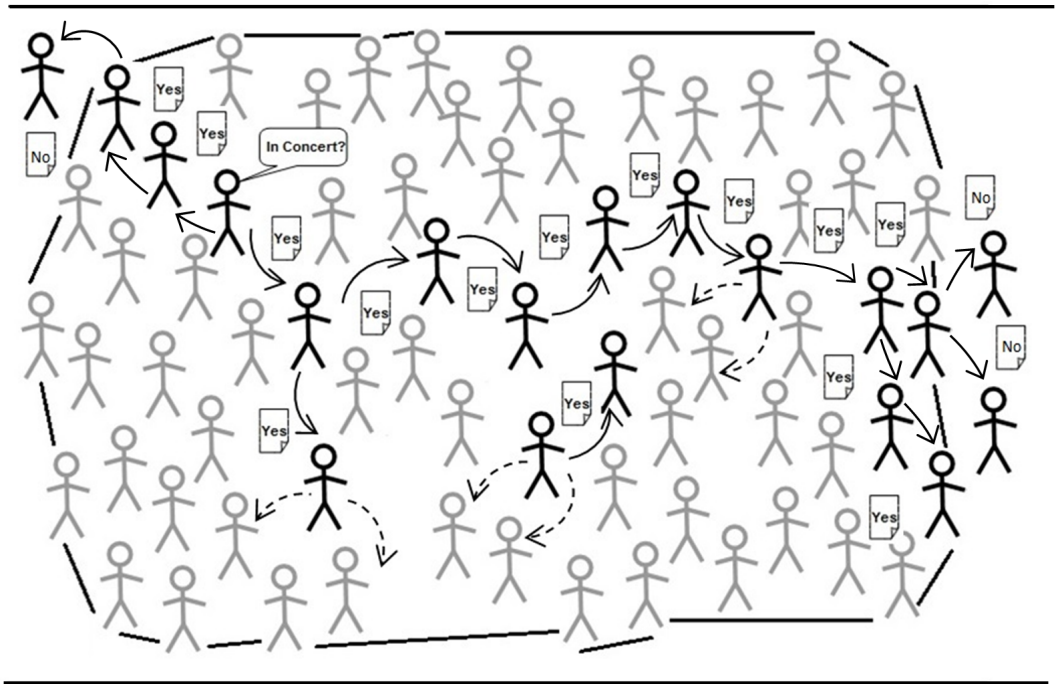


Figure 5.9: An illustration of a Boundary Finding scenario

Jim could start by sending the query to neighbours via a Bluetooth or Wi-Fi Direct connection. He can then pass this query to randomly selected neighbours, ask them the question "Are you at the concert?" and also let them pass this question to their neighbours. The query then continues to be passed to the neighbours. The expiry time for this query is identified by Jim. When the time is over, the

results will be returned to Jim with the same route as it has been forwarded. Finally, the application computes the results, which then displays the boundary and current locations of responders on the map, as shown in Figure 5.9. Boundaries will be where responders start to say "No" to the question.

5.3.2.2 Boundary Algorithm

As previously mentioned, this application aims to search for the boundary of the particular event or activity by asking the crowd through mobile ad hoc networks. The system returns the boundary of some location, either the event's location or a reference object's location. The result is shown in a polyline boundary which is a multipoint that contains the collection of the start and end points of each line. In our case, we use the coordinates in the map as the points.

Figure 5.10 shows the boundary that computes a spatial boundary based on the locations of collected events of types $E_1 ... E_n$, where all events have their own coordinate location. To find this boundary, the computation of a planar convex hull [Berg et al., 2008] in computational geometry is applied. The convex hull stands for the boundary of the minimal convex set containing a given non-empty finite set of points in the plane. Unless the points are collinear, the convex hull is a simple closed polygonal chain.

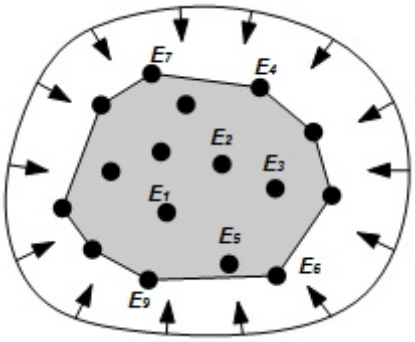


Figure 5.10: The example of convex hulls

```
Algorithm: Quickhull
Function QUICKHULL(a,b,S)
   if S = ∅ then return ()
   else
       c = index of a point that is furthest (max distance) from ab.
       Let A be the set containing points strictly right of (a, c)
       Let B be the set containing points strictly right of (c, b)
       return {QUICKHULL(A, a, c) ∪ (c) ∪ QUICKHULL(B, c, b)}
```

Listing 5.1: Quickhull algorithm

Figure 5.10 also illustrates a simple convex hull. From our case scenario in Section 5.3.2.1, the boundary is all locations/areas in which people exist in the

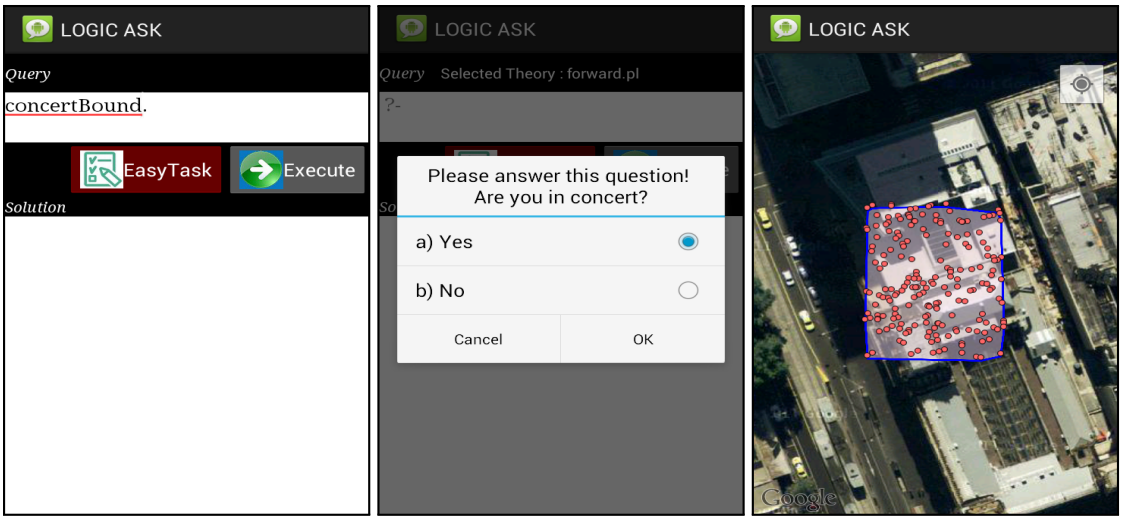
concert. In this case, all people, whose locations are within this boundary, receive the query from the source that asks whether they are in the concert or not. If they say "yes", the coordinate is returned to the origin. Then the boundary is computed. In particular, we apply the Quickhull algorithm [O'Rourke, 1998] to compute the boundary of this application. Listing 5.1 shows its pseudo code.

5.3.2.3 Demonstration

In the scenario above, the problems could be solved by using help from the crowd to provide the best answer to the particular query. In the Boundary Finding application, one can send a query via Bluetooth or Wi-Fi Direct technologies to the crowd. We exploit GPS coordinates and Google Maps APIs in mobile devices to create a destination point or boundary area. Returning to our scenario, a program finds the boundary of a particular area. The program can be written as the following rule.

```
concertBound:-
          *inConcert?(Co)#[syn,asktype("geo"),
                question("Are you in concert?"),
                option(["yes","no"]),
                askto([bluetooth]),
                expiry("1,0,0")],
          bound(Co,Bound),
          currentLoc(CurrentLoc),
          showMap(CurrentLoc, Bound).
```

The `concertBound/0` aims to estimate the boundary of the concert by sending a question to the crowd via Bluetooth connections and the crowd predicate, identified using the crowd keyword "`inConcert`", is called as the open predicate to peers. Also, the operator "*" added before the crowd keyword enables peers to distribute the question to people attending the concert. We create new crowd conditions: `asktype("geo")` is a type of question/task which could return the coordinates (latitudes and longitude) of peers at a current position. After executing the crowd sub-goal, as shown in Figure 5.11(a), the question with these conditions then appears on friends' devices that are connected via Bluetooth in Figure 5.11(b). Some time later, each peer propagates this question among his/her friends. Within one hour, set as the expiry time, the result is returned to the query origin via *LogicCrowd*, where `bound/2` is programmed to draw the boundary using the coordinates and `currentLoc/1` uses GPS to obtain the current location. Finally, `showMap/2` will display the boundary and current locations of responders who say "Yes" on Google Maps, as shown in Figure 5.11(c).



(a) Execute Goal (b) Ask Question to Crowd (c) Display Result

Figure 5.11: Distributing geo task among peers via Bluetooth to find a boundary

5.4 Summary

We have presented the Crowd Voting and Recommendation application. We have illustrated the concept of *LogicCrowd* and provided services that can be used in *LogicCrowd*. Note that *LogicCrowd* can be applied to situations on a daily basis, such as making general decisions and doing research in a marketing field. In addition, we have demonstrated the crowd unification paradigm and have described a Crowd-Powered Comparisons application which employs the power of the people to solve unification problems that might be difficult to solve using the traditional approach.

Furthermore, in this study we have introduced two applications: AskCrowd Parking and Boundary Finding applications. These applications are used to demonstrate the *LogicCrowd* in mobile peer-to-peer networks. AskCrowd Parking employs the power of the crowd in the mobile environment aiming to solve or ameliorate the parking problem as an example. Boundary Finding aims for finding the boundary of effective zones using GPS coordinates and the Google Maps API in mobile devices to determine a destination point or boundary area.

In summary, we have explained in detail the four applications built using the *LogicCrowd* framework from concept to their implementation. We have illustrated how *LogicCrowd* can be applied for conveniences. This involves the querying and manipulation of crowdsourcing over social media networks and peer-to-peer

networks. We believe that the *logic* programming approach can go further than imperative approaches, as it enables rule-based reasoning with resource descriptions and more expressive queries. In the next chapter, we evaluate the proposed *LogicCrowd* framework by measuring the energy consumption characteristics of our *LogicCrowd* prototype. The findings show the relationship between *LogicCrowd* and energy consumption on various platforms and compare the magnitude of the coefficients in *LogicCrowd* factors influencing the energy consumption across crowdsourcing platforms. These results are useful for mobile users and developers using our *LogicCrowd* applications.

Chapter 6

Performance Evaluation of the *LogicCrowd* Framework

In the previous five chapters, we have described *LogicCrowd* from its concept to implementation, as well as the usage and applications of the *LogicCrowd* framework. In this chapter, we discuss the performance evaluation of our *LogicCrowd* prototype. More specifically, this chapter examines how a *LogicCrowd* program can influence energy consumption on the mobile devices under different criteria of operation in order to study practical issues in *LogicCrowd* applications. This chapter have been published in conference proceedings [Phuttharak and Loke, 2014a].

The chapter is organized as follows. Section 6.1 presents our evaluation methods and gives an introduction as to how to evaluate *LogicCrowd*'s performance. Section 6.2 investigates the energy consumption characteristics of basic crowd predicate in the *LogicCrowd* prototype. This study of the evaluation measurement is divided into four phases: setup, do measurements, determine energy consumption models, and work out extensions to the *LogicCrowd* meta-interpreter. This section also provides a model for managing energy consumption in the *LogicCrowd* program execution.

Section 6.3 explores the energy characteristics of crowd unification based on our *LogicCrowd* prototype. The study addresses energy consumption behavior of crowd unification when using various crowdsourcing platforms including MTurk, Facebook and Bluetooth via actual mobile devices. We conclude the chapter in Section 6.4.

6.1 *LogicCrowd* Framework Evaluation

In this section, we evaluate the *LogicCrowd* framework based on its energy consumption because energy efficiency is an important aspect in the mobile environment [Barroso and Holzle, 2007; Dinh et al., 2013; Miettinen and Nurminen, 2010]. The energy consumption of mobile devices, such as smartphones, has increasingly become a concern from various sectors, ranging from smartphone manufacturers, mobile developers, to end-users. Although battery capacity has improved with CPU and memory increases over the years, the battery life of mobile devices is not catching up proportionally for a large spectrum of current applications.

Mobile crowdsourcing is an area of rapid innovation and will become increasingly prevalent in society. Many mobile applications integrate rich media and location tracking features, including exploiting multi-mobile sensors such as accelerometers, gyroscopes, GPS, cameras and microphones, which task the deployed mobile devices to form interactive and participatory sensor networks. This enables public and professional users to gather, analyze and share local knowledge. Hence, the energy consumption of mobile crowdsourcing is an important issue to be considered.

In the evaluation methodology, we aim to estimate the energy consumption behavior of the *LogicCrowd* prototype implemented on three different extensions of Prolog, as mentioned in Chapter 3, including (1) basic crowd predicate, (2) crowd unification, and (3) *LogicCrowd* in P2P networks. The proposed methodology describes two aspects: evaluation on actual mobile devices and evaluation in simulation environments.

First, the evaluation on actual mobile devices contains the testing setup used to collect energy consumption measures of all three extensions for different criteria, namely (1) execution modes of crowdsourcing, (2) length of wait time, (3) the number of crowd queries, (4) the size of transmitted contents, and (5) types of network connections.

Second, the evaluation in simulations is to study the distribution of crowd tasks for mobile crowdsourcing in an opportunistic network and to construct basic scalability experiments. The purpose of the simulation evaluation results presented in Chapter 7 is two-fold: (a) to analyze the relationship between the parameters of mobile crowdsourcing and the number of responses from the crowd, and (b) to study the energy consumption characteristics of mobile crowdsourcing, especially in a large scale ad-hoc network.

In this chapter, we investigate the energy characteristics of the *LogicCrowd* prototype on actual mobile devices in the first two extensions as mentioned above: (1) basic crowd predicate (see in Section 6.2) and (2) crowd unification (see in Section 6.3).

6.2 Evaluation of *LogicCrowd* – Energy Considerations for the Basic Crowd Predicate

Mobile phones have limited energy resources, especially battery capacity. In this section, we present a study of the energy consumption characteristics of our *LogicCrowd* prototype and provide a model for managing energy consumption in *LogicCrowd* program execution. Our evaluation process was done in four phases, as follows.

6.2.1 The First Phase: Setup

We designed our experiments to obtain two sets of measurements. In the first experiment, the aim was to compare the total energy consumption when asking the crowd using different network connections (e.g., Wi-Fi, Bluetooth and Mix (using both Bluetooth and Wi-Fi in the same crowd goal) and execution modes (Synchronous and Asynchronous). We created simple programs using the crowd predicate to send a query to the crowd as shown in Listing 6.1.

```
syn_bt:- findall(B,(thai(B),melbourne(B)),A),nice?(Ans)#[syn,asktype('choice'),
     question('Which restaurant do you recommend?'),options(A),askto([bluetooth]),
     expiry('0,0,60')].
syn_wf:- findall(B,(thai(B),melbourne(B)),A),nice?(Ans)#[syn,asktype('choice'),
     question('Which restaurant do you recommend?'),options(A),askto([facebook]),
     expiry('0,0,60')].
syn_mix:- findall(B,(thai(B),melbourne(B)),A),nice?(Ans)#[syn,asktype('choice'),
     question('Which restaurant do you recommend?'),options(A),askto([facebook,
     bluetooth]),expiry('0,0,60')].
asyn_bt:- findall(B,(thai(B),melbourne(B)),A),nice?(_)#[asyn,asktype('choice'),
     question('Which restaurant do you recommend?'),options(A),askto([bluetooth]),
     expiry('0,0,60')].
asyn_wf:- findall(B,(thai(B),melbourne(B)),A),nice?(_)#[asyn,asktype('choice'),
     question('Which restaurant do you recommend?'),options(A),askto([facebook]),
     expiry('0,0,60')].
asyn_mix:- findall(B,(thai(B),melbourne(B)),A),nice?(_)#[asyn,asktype('choice'),
     question('Which restaurant do you recommend?'),options(A),askto([facebook,
     bluetooth]),expiry('0,0,60')].
```

Listing 6.1: The simple rules using the crowd predicate for the experiment

These rules show six different kinds of crowd predicates used in the measurements, namely Synchronous–Bluetooth, Synchronous–WiFi, Synchronous–Mix, Asynchronous–Bluetooth, Asynchronous–WiFi, and Asynchronous–Mix. In each rule, the expiry time in the crowd condition is set to 60 seconds. In this experiment, we increased the number of rules (and hence, the number of crowd goals) by 5 each time, going up to 30 rules in order to investigate the hypothesis that *if the number of rules increase, the power consumption will constantly increase in a simple linear form under the different connections and executions.*

The second experiment was conducted to determine whether *if there is an increase in waiting period (expiry time) for answers to a query sent to the crowd, the energy consumption will increase steadily in a simple linear curve.* The rules from the first experiment were applied in this case. The expiry times in the crowd predicate were changed by increasing by 5 minute intervals up to 20 minutes. This experiment leads to the third phase of the study where we estimate the power consumption per query of *LogicCrowd* programs, under different communication connections and execution methods. All tests in these two experiments were performed on Nexus S running the Android operating system version 4.1.2, Jelly Bean, powered by 3.7 volts Li-ion battery. A power tool called Little Eye[1] was utilized to capture the battery level of the Android smartphone and also to monitor the power consumption of the *LogicCrowd* program.

6.2.2 The Second Phase: Measurements

Figure 6.1(a)-(e) shows the total energy consumption (for display, CPU and networking) as the number of rules (i.e., correspondingly the number of crowd predicate calls) varies when using different network connections (Bluetooth, Wi-Fi and Mix) and different execution modes (Synchronous and Asynchronous), compared with the baseline "no-communication" run of similar rules but without crowd predicates, i.e., the baseline.

According to the results, the hypothesis of the first experiment is shown to hold as all graphs display similar trends that is, a linear increase, which is more scalable than an exponential increase. As shown in Figure 6.1(a), asking the crowd by using Mix connection with the Synchronous mode consumes the most energy, accounting for 1.11 times of the total power consumed for Wi-Fi, 1.27 times for Bluetooth, and 15 times that of "no-communication". For a rule using a crowd

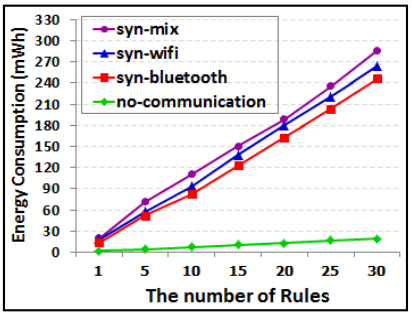
[1] http://www.littleeye.co/

predicate call with Mix connection for the waiting period of 1 minute, the power consumption is spent differently: 18.026 mWh (milliwatts per hour) on display (keeping the *LogicCrowd* front screen up), 1.399 mWh on CPU, and 0.067 mWh on Wi-Fi – display clearly dominates but if the application runs in the background CPU and connectivity will be important.

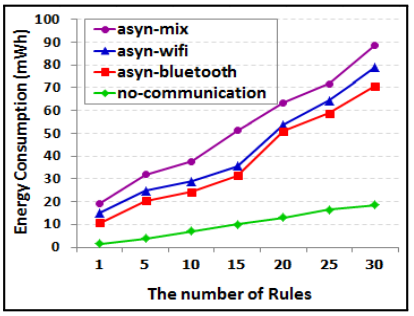
Crowd asking using the Wi-Fi connection in the Synchronous mode consumes marginally around 1.14 times more power than crowd asking using the Bluetooth connection with the same method, and consumes about 13.46 times significantly more energy than using rules without communication with the crowd. In a rule with this Wi-Fi mode and 1 minute waiting period, the power consumed for display, CPU, and Wi-Fi varies: 17.294 mWh, 0.981 mWh, and 0.067 mWh respectively. Also, crowd asking using Bluetooth with Synchronous execution consumes around 11.93 times the total energy compared to rules without communication with the crowd - with no crowd goals, 12.565 mWh of power for a rule is used for display and 1.040 mWh for CPU. In comparison with the no-communication mode, we can see that Wi-Fi, Bluetooth, and Mix all consume much more energy. This is because with the use of the crowd predicate in crowd-communicative modes, it takes time to wait for the crowd's responses. The more rules used, the longer the waiting period took, and the higher the energy consumed. Most of the energy has been found to be mostly used for display, accounting for 70-90% of the total energy spent on the foreground when asking the crowd (an optimization this suggests is to wait for the crowd answers in background mode).

Figure 6.1(b) shows the differences in power consumption when using Wi-Fi, Bluetooth and Mix connections in the Asynchronous mode, where the overall result is found similar to that of the Synchronous mode. That is, the total power consumed when using Mix connection is around 1.245 times, and 1.463 times slightly higher than the power consumed when using the Wi-Fi and Bluetooth connection with the same method. It consumed 6.360 times more energy than the amount used with the no crowd communication rules. Here, for each rule with Mix connection, 17.301 mWh of the mean power is spent on display, 1.809 mWh on CPU, and 0.067 mWh on Wi-Fi, while the total power consumed when using the Wi-Fi connection is around 1.17 times higher than the power consumed when using the Bluetooth connection under the same method. However, the amount of energy spent using Wi-Fi is about 5.09 times higher than the energy for no crowd communication rules. The mean energy spent per rule on display, CPU, and Wi-Fi is 13.838 mWh, 1.243 mWh, and 0.056 mWh respectively. In addition, using

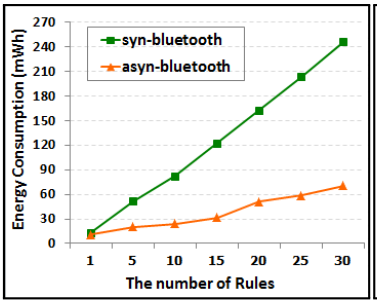
Bluetooth in the Asynchronous mode consumes around 4.26 times more energy than "no-communication" with the crowd.



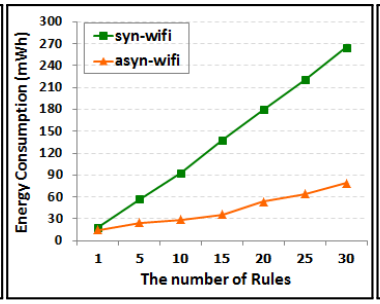
(a) Synchronous execution method



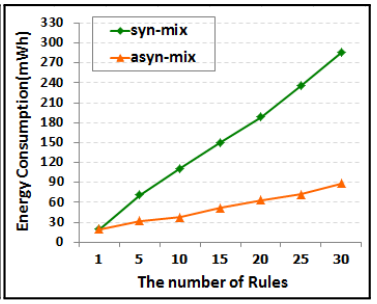
(b) Asynchronous execution method



(c) Bluetooth connection



(d) Wi-Fi connection



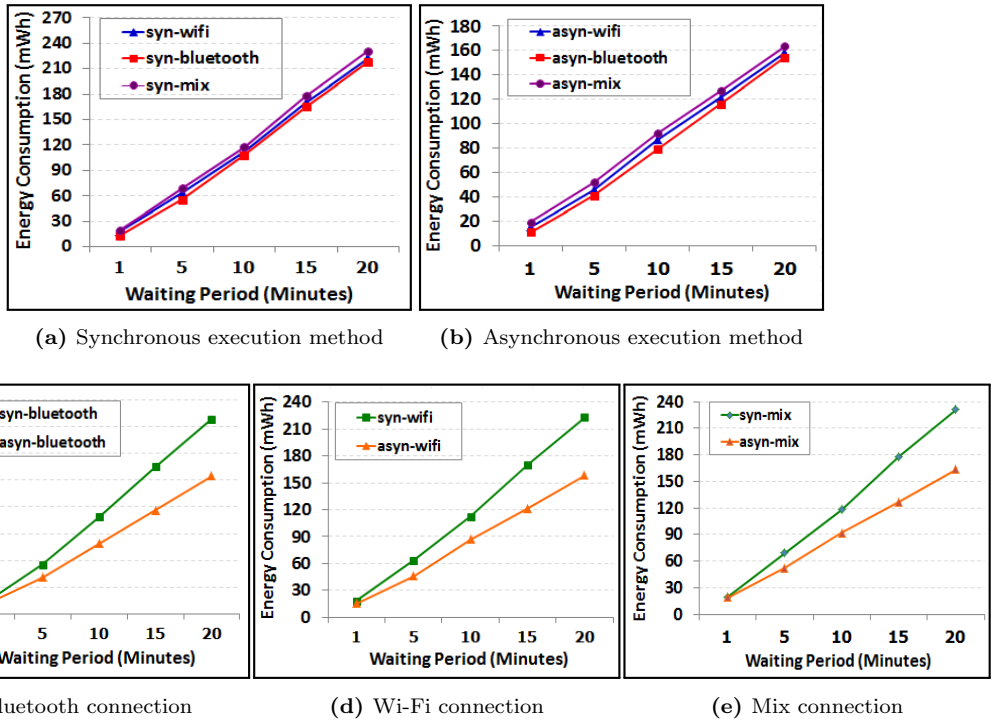
(e) Mix connection

Figure 6.1: The relationship between energy consumption and the number of rules/queries in different types of execution methods and communication technologies, x-axis is the number of rules/queries; y-axis is energy consumption (milliwatts per hour)

Figure 6.1(c)-(e) presents the energy consumption of different operation modes (Synchronous and Asynchronous) when the rules have been increased at equivalent intervals (every 5 rules). The result shows that the Synchronous mode consumes significantly more energy than execution in the Asynchronous mode. Also, 20 executed rules in the Synchronous-Bluetooth mode consume 163.17 mWh whereas the Asynchronous-Bluetooth mode with the same size of rules consumes only 51.06 mWh. In this regard, we can say that the average energy consumed in the Synchronous mode with Bluetooth, Wi-Fi, and Mix connection is three times higher than the average energy used in the Asynchronous mode. The findings of the above energy analysis cases is expected to contribute to the strategic guidance for selecting the most appropriate energy-related conditions to support the intelligent use of asking the crowd in a *LogicCrowd* program.

Figure 6.2 shows the relationship between energy consumption and the approximate waiting period of returned feedback from the crowd. According to the results, the hypothesis of the second experiment has been proven in that all the graphs display similar trends and the regression line slopes upwards. The longer

the waiting period, the more energy consumed. In Figure 6.2(a), the comparison of energy consumption among Wi-Fi, Bluetooth and Mix connection in the Synchronous mode is demonstrated. The graph shows that the energy of a rule using Mix connection consumes slightly more energy than that using Wi-Fi and Bluetooth connections. In addition, the energy consumption of a rule using Wi-Fi is approximately 1.11 times higher than that using a Bluetooth connection. A similar trend is shown in Figure 6.2(b) in which the average power is 1.14 times more likely to be consumed by Wi-Fi than by Bluetooth connection in the Asynchronous operation.



(a) Synchronous execution method **(b)** Asynchronous execution method

(c) Bluetooth connection **(d)** Wi-Fi connection **(e)** Mix connection

Figure 6.2: The relationship between energy consumption and waiting period/expiry for different types of execution methods and communication technologies, x-axis is waiting period (minutes); y-axis is energy consumption (milliwatts per hour)

From the experiments, we can see that whether the method is Synchronous or Asynchronous, the total rate of approximate power spent is not much different. Compared with Wi-Fi and Bluetooth, Mix connection consumes the most energy, whereas Bluetooth consumes the least. As in the other experiments on number of rules, it should also be noted that lots of power is spent on display, accounting for around 60-90% of the total energy. Further, the rate of energy consumption is congruent with the total amount of waiting time. Simply said, the longer we wait for crowd responses, the more power is consumed. However, Figure 6.2(c), 6.2(d)

and 6.2(e) shows the energy consumption difference between Synchronous and Asynchronous execution. With an increasing waiting period in steps of 5 minutes, the energy consumption per rule of Synchronous version grows reasonably around 1.32, 1.36 and 1.29 times more than the energy consumption per rule of Asynchronous operation among Wi-Fi, Bluetooth and Mix communication connections respectively.

According to the second experiment, we designed a simple linear energy consumption models for each type of execution modes and communication technologies. Those models will be explained in the following section.

6.2.3 The Third Phase: Energy Consumption Models

Based on the measurements above, we constructed a simple (linear) energy consumption model. Table 6.1 shows energy consumption functions per rule with respect to the waiting period when using Bluetooth, Wi-Fi and Mix communication with different execution methods. Regarding the second experiment, these formulas have been designed in order to predict the power usage of a *LogicCrowd* program given the waiting period/expiry conditions in its crowd predicates. This means that these functions can estimate the power (denoted by y) which is consumed when using a particular communication technology with a particular execution method for a waiting period of T minutes.

Table 6.1: Energy consumption functions per rule with respect to the waiting period when using Bluetooth, Wi-Fi and Mix communication technologies using different execution methods

Connections	***Synchronous Execution***	***Asynchronous Execution***
Wi-Fi	$y = 10.749T + 7.955$	$y = 7.541T + 8.588$
Bluetooth	$y = 10.811T + 1.787$	$y = 7.548T + 3.164$
Mix	$y = 11.067T + 10.168$	$y = 7.574T + 13.512$

Functions, as mentioned above, can predict the energy usage for a rule with one crowd predicate. The functions can be obtained for any Android device via a set of benchmark measurements as we have done above on our test device. To predict the energy usage for a group of crowd predicates, we created the applicable formulas as shown in Table 6.2. Synchronous functions from Table 6.1 can only be applied to these formulas. Because they are working sequentially without any parallel extensions, the energy usage of rules could be then estimated one by one. On the other hand, the Asynchronous execution method runs in parallel without being blocked by incomplete processing of a crowd predicate. Hence, a function

for Asynchronous method has been developed as shown in Table 6.2. This function can be used to calculate the power of battery (i.e. y) which is consumed when using N_{WF} rules of Wi-Fi, N_{BT} rules of Bluetooth and N_{MIX} rules of Mix connections for a waiting period of T_{MAX} minutes.

In order to possibly verify our energy consumption model, we made additional measurements and compared the obtained results with the results calculated with the energy consumption functions from Table 6.1 and Table 6.2. For instance, if we execute the rule with the expiry condition of 30 minutes by using Wi-Fi with Synchronous mode, according to Table 6.1 and the following function y = $10.749T + 7.955$, for $T = 30$, we consume 330.425 mWh of the battery. Measurements in a real world environment showed that the rule consumed around 328.638 mWh of energy, which is similar to the energy consumption calculated by using energy consumption functions from Table 6.1.

Table 6.2: Energy consumption functions with respect to a group of crowd predicates

Execution Modes	***Connections***	***Functions***
Synchronous	*Wi-Fi*	$y = \sum_{i=1}^{N}(10.749T_i + 7.955)$
	Bluetooth	$y = \sum_{i=1}^{N}(10.811T_i + 1.787)$
	Mix	$y = \sum_{i=1}^{N}(11.067T_i + 10.168)$
Asynchronous	$y = 1.987N_{BT} + 2.079N_{WF} + 2.790N_{MIX} + 7.437T_{MAX} - 1.650$	

Another verification example is that if we execute the rules which have 1 rule for Wi-Fi, 2 rules for Bluetooth and 4 rules for Mix connection with Asynchronous method and 5 minutes of maximum waiting period, according to Table 6.2 and the following function $y = 1.987N_{BT} + 2.079N_{WF} + 2.790N_{MIX} + 7.43T_{MAX} - 1.650$, for $N_{BT} = 2, N_{WF} = 1, N_{MIX} = 4, T_{MAX} = 5$, the power consumed is 52.713 mWh. Real measurements showed the rule consumed around 53.187 mWh; which is not much different from the energy consumption estimated from the functions in Table 6.2. We have shown that the energy consumption can be appropriated linearly, roughly speaking.

6.2.4 The Fourth Phase: Extensions to the *LogicCrowd* Meta-Interpreter

The proposed models in the 3^{rd} phase have been deployed here to manage energy consumption of *LogicCrowd* programs. In this phase, we present the algorithm implemented in *LogicCrowd* by computing the energy budget corresponding to a

certain battery lifetime. We modify the *LogicCrowd*'s meta-interpreter to use the energy estimations during run-time to monitor application workload and adapt its behavior dynamically to save energy.

The power per crowd goal/rule for each network connection with a particular execution method is estimated via Table 6.1. Managing the energy usage of *LogicCrowd* program can be defined as follows. $E_{crowd(i)}(t_i)$ denotes the energy consumption of a crowd predicate with waiting period of time t_i, where i can be one of six different aspects of querying the crowd (as mentioned in Table 6.1), where $i \in \{syn-bt, syn-wf, syn-mix, asyn-bt, asyn-wf, asyn-mix\}$. $E_{current}$ denotes the current energy level based on the current battery power remaining. Suppose that the energy budget $\beta(\%)$ is a user's policy of energy usage allowed for *LogicCrowd* programs, i.e., a percentage of current battery level $E_{current}$. To manage energy consumption of *LogicCrowd* applications and enhance battery performance, we use the condition given as follows: $E_{crowd(i)}(t_i) \leq \beta(\%) \times E_{current}$.

According to the relation above, the crowd predicate goal is allowed to proceed only when the energy estimated for that crowd predicate, i.e. $E_{crowd(i)}$ at t_i minutes, is less than or equal to the amount of energy budgeted, i.e., $\beta E_{current}$. For example, assume that the mobile user specified an energy budget of 25% of the current phone's battery level and the current remaining battery energy $E_{current}$ is 4,440 mWh. Note that remaining battery can be measured in other units or via the built-in level API (as in Android), but mWh is used as an example. When evaluating a rule with a crowd predicate under the Synchronous execution using Mix network connection with a one hour waiting time, the energy consumed is estimated to be 674.14 mWh, which is less than the energy budget (1,110 mWh). As a result the system then continues to process this rule, and the crowd goal is allowed to proceed. In contrast, if the estimated energy of this rule is greater than the energy budget, the rule will be skipped and the system will stop the process in order to maintain the battery energy. This is a way to manage the energy spent on the application. This algorithm and the modified rules in the meta-interpreter are as shown in Listing 6.2.

In Listing 6.2, let the process start with the goal G consisting of $g_1, g_2, g_i \ldots, g_n$ where g_i is a sub-goal of G in Prolog. For every g_i, which is a crowd predicate, its energy use is with particular conditions (execution connection mode) and then compared with the energy budget (i.e. current energy level). If the energy usage is not greater than and equal to the budget, the system will continue and execute this

rule; otherwise, the system will terminate and show the message "not enough energy". With this algorithm, we applied to *LogicCrowd*'s meta-interpreter as shown in Listing 6.3. We extended predicate named `enough_energy/3` to `asynproc/2` and `synproc/2` in *LogicCrowd* meta-interpreter shown in Appendix A. This predicate will be executed via the Mediator (custom-built Java program) the "enough_energy" method.

Define $G:-g_1,g_2,g_i,...,g_n$ where g_i is sub-goal in Prolog.
    β where the energy budget (%)
for each g_i do
  if g_i is a crowd predicate **then**
    check for crowd's conditions
    $j \leftarrow$ (execution mode and connection type)
    where $j \in \{syn-bt, syn-wf, syn-mx, asyn-bt,$
        $asyn-wf, asyn-mx\}$
    $t_j \leftarrow$ expiry
    compute energy consumption = $E_{crowd(j)}(t_j)$
    get current energy level = $E_{current}$
  if $E_{crowd(j)}(t_j) \leq (\beta \times E_{current})$ **then** evaluate g_i
  else message "Not enough energy."
  break

Listing 6.2: The energy budget control algorithm

```
...
synproc(Askcrowd,Result):-
       checkcond(TypeQuestion,...,Expiry),
       enough_energy(syn,Askto,Expiry),
       askcrowd(Askcrowd,...,QuestionID),
registercallbacksyn(QuestionID,..., Result).

asynproc(Askcrowd,Result):-
       checkcond(TypeQuestion,...,Expiry),
       enough_energy(asyn,Askto,Expiry),
       askcrowd(Askcrowd,...,QuestionID),
registercallbackasyn(Askcrowd,...,Result).
...
```

Listing 6.3: The modified rules in the LogicCrowd meta-interpreter

6.3 Evaluation of *LogicCrowd* – Energy Considerations in Crowd Unification

In this section, we investigate the energy characteristics of crowd unification based on our *LogicCrowd* prototype. The objective of this study is to present a quantitative study of crowd unification's energy consumption when using various platforms (namely, MTurk, Facebook and Bluetooth) to crowdsource answers from a mobile device. Our energy measurements are carried out on a LG Pro Lite mobile phone,

as a representative device. We note that actual energy consumption will vary from device to device but we argue that our results provide an insight into the feasibility of our approach, from an energy consumption standpoint. Additionally, we aim to explore the relationship between energy consumption and the different criteria of crowd unification including (1) length of wait time, (2) the number of crowd queries, and (3) the sizes of transmitted contents.

In this section, we measure the energy consumption of mobile applications via changes in battery levels. Although battery level changes are coarse-grained, it can be easily collected through a user-level application on mobile devices. Also, the findings can benefit mobile users when they try to estimate the battery consumption before making queries to the crowd using different platforms. The experiments and energy analysis are discussed as follows.

6.3.1 Experimental Setup

To compare the battery level decrease when using crowd unification in four different kinds of crowdsourcing platforms: MTurk, Facebook, Bluetooth, and Mix (combining MTurk, Facebook and Bluetooth in the same query), we create a simple crowd unification query/rule of the form: `photo('a.jpg')*= photo('b.jpg').` For each instance of the query issued, we design different conditions relating to three factors which impact on battery changes such as waiting time, the number of queries, and the data sizes. For example, a query is sent to MTurk which has a waiting period of 5 minutes and a photo size of 100 Kilobytes.

In this experiment, the query was configured by setting the waiting period starting from 5 minutes onwards and running in 5 minute intervals up to 15 minutes. The photo sizes tested were 100, 500, and 1000 Kilobytes. We also explored increasing the number of repeated queries by 5 each time, going up to 15 queries. The set of tests is for investigating the relationship between energy consumption of crowd unification and those three factors. A total of 144 test cases was defined in *LogicCrowd* programs running on the Android operating system. All tests in this experiment were performed on a LG Pro Lite D680 phone running Android v4.1.2 Jelly Bean, Bluetooth v3.0, RAM 1GB, powered by a Li-ion 3140 mAh battery, with WiFi connections to the Internet. Each test ran in foreground mode with display and brightness set at 69% - the idea is that this is a typical use-case scenario.

6.3.2 Experimental Results and Analysis

Figure 6.3 shows the percentage of battery level change (indicative of energy consumption) with varying waiting periods (the interval time of returned feedback from the crowd) when using four different types of peers, i.e., MTurk, Facebook, Bluetooth and Mix, for crowd-sourcing results. Each graph displays battery level changes in three different data sizes (i.e., 100, 500, 100 KB) and varying number of queries (i.e., 1, 5, 10, 15 crowd unification queries).

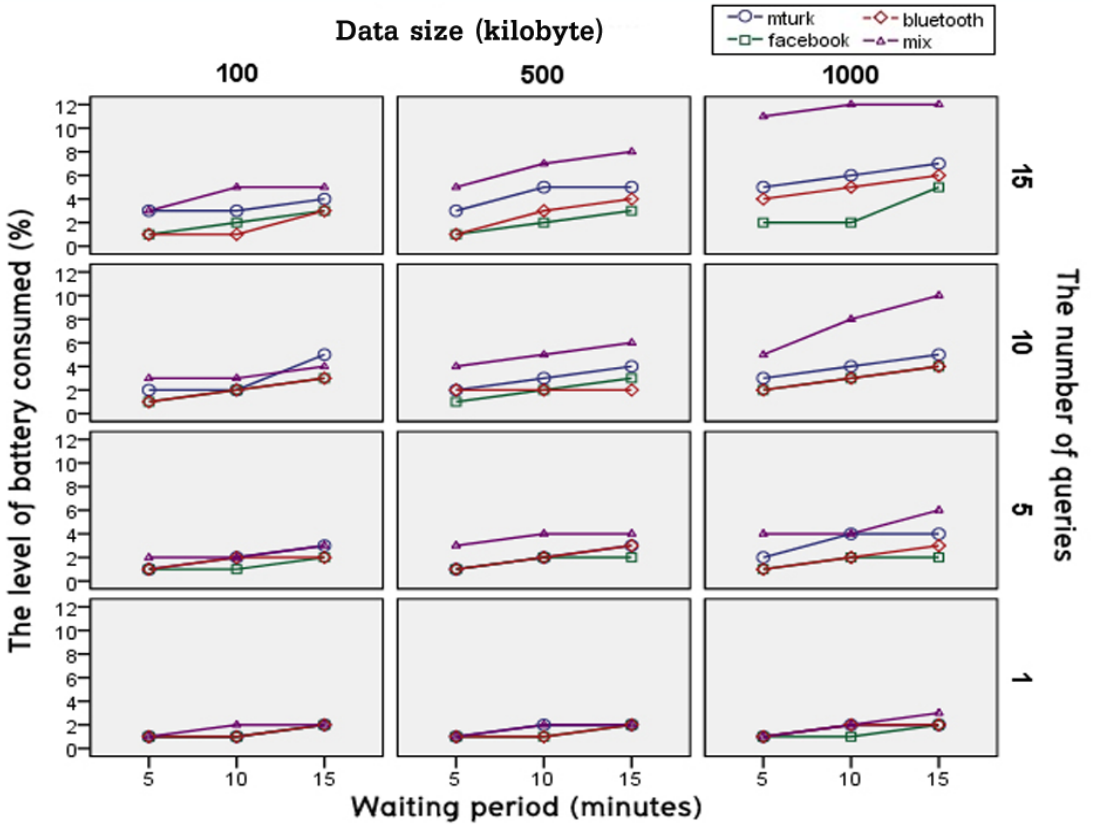


Figure 6.3: The relationships between the level of battery change and waiting period for different data sizes and varying number of queries

Figure 6.3 reports the similarities of energy consumption when a query was sent to the crowd for different data sizes, that is, there is only a slight increase in the consumption of battery when waiting period moved up from 5 minutes to 15 minutes. Moreover, the percentage of change in the battery level per query is not much different across the various crowdsourcing platforms, MTurk, Facebook, Bluetooth and Mix. For example, asking the crowd via MTurk by sending a photo with the size of 100 KB and waiting for feedback from the crowd for 5 minutes, the battery level changed by only 1% which equals the battery level change spent by asking the crowd via Facebook, Bluetooth or Mix. With the photo size of 100

KB and a 15-minutes waiting period, the average of the battery level change was about 2% for all four crowd modes. However, for the waiting period of 5 minutes, the change was only 1% even when the data size was changed to 500 KB or to 1000 KB among all peers.

Consider the results as presented through these graphs when the number of queries increases, it is apparent that with more queries, more energy is consumed as we expect. The change in the battery level of MTurk is an example. When the query was sent via the MTurk platform with the data size of 100 KB and a waiting period of 5 minutes, the battery level change by 1%, 1%, 2%, and 3% corresponding to the increasing number of queries of 1, 5, 10, and 15, respectively. Moreover, when the waiting period was moved to 15 minutes with the same data size of 100 KB and the queries were sent to MTurk, there was a change in the battery consumption by 2%, 3%, 4%, and 5% with 1, 5, 10, and 15 queries, respectively. With the same conditions, the percentages of battery changes of 2%, 2%, 3%, and 3% were found when querying using Facebook and Bluetooth while for Mix querying, slightly more energy was used, i.e., a change of 2%, 3%, 4%, and 5% for the number of queries of 1, 5, 10, and 15, respectively.

It is evident that the number of queries and the data size impact upon battery consumption. When the number of queries is higher and the data size is bigger, the level of battery consumed in each crowd-sourced data can vary significantly. The platform of Mix queries consumes the most energy while querying MTurk uses more energy than querying via Facebook or Bluetooth. Interestingly, asking the crowd via Facebook consumes the least energy. That is, in the case of 15 queries, with a size of 500 KB and a 15-minute waiting period, the battery consumption used by in Mix, MTurk, Bluetooth and Facebook is 8%, 5%, 4%, and 3%, respectively. With a data size of 1 MB using the same criteria above, the energy consumed by in Mix, MTurk, Bluetooth and Facebook is 12%, 7%, 6%, and 5%, respectively

In Figure 6.3, all graphs display similar trends, that is, roughly linear relationships. As the aim of our study is to investigate the relationships between battery consumption and the different predictor variables of crowd unification, this work studies *the percentage of change in battery consumed as affected by (1) the waiting period, (2) the number of crowd queries, and (3) the data sizes of the transmitted contents.*

To investigate these relationships, we perform multiple linear regression analysis with SPSS, Version 21. We report on the results using multiple regression

analysis, including standardized coefficients (Beta values), t-statistics, and p values (significance level). The Beta value can be a measure of how strongly the predictor variables influence the level of energy consumption. The analysis was conducted separately across the four different groups of peer platforms (MTurk, Facebook, Bluetooth, and Mix).

The findings show that the three factors that are the waiting period of time, the number of crowd queries, and the data size of the transmission significantly determine the extent of battery life to be consumed ($p \leq 0.001$ at 0.05 level of significance). This means that when the waiting period is longer and the number of queries and the data size increases, more battery will be consumed. From Table 6.3, the Beta value report that each of the four modes (three platforms) affects the change of battery level differently. In MTurk, the beta value of the number of queries reached the highest (Beta = 0.721), suggesting that it made the highest impact on battery consumption, as compared to the Beta values of the size and the waiting period, these being 0.360 and 0.465, respectively.

Table 6.3: The impact of the factors on the energy consumption across four crowdsourcing modes (three platforms)

Peers	**Factors**	**Beta**	**t**	**Sig**
MTurk	1) Waiting period	0.465	7.170	0.000
	2) Data size	0.360	5.552	0.000
	3) No. of crowd queries	0.721	11.112	0.000
Facebook	1) Waiting period	0.690	7.781	0.000
	2) Data size	0.261	2.936	0.000
	3) No. of crowd queries	0.451	5.084	0.000
Bluetooth	1) Waiting period	0.530	5.574	0.000
	2) Data size	0.424	4.467	0.000
	3) No. of crowd queries	0.501	5.269	0.000
Mix	1) Waiting period	0.252	3.458	0.002
	2) Data size	0.497	6.808	0.000
	3) No. of crowd queries	0.721	9.877	0.000

For the Facebook platform, the waiting period had the highest impact on energy consumption compared to the other variables. However, for querying via

Bluetooth connections, the Beta values of the four factors were not much different, suggesting that they have similar levels of impact on the battery consumption. Finally, in Mix querying, the factor that has the greatest impact on energy levels is the number of queries with Beta value at 0.721. The results mean that, for example, (i) when using MTurk, the number of queries impact more on energy consumption due to the interactions involved, (ii) when using Facebook, the waiting period is more important, and (iii) when using Bluetooth, the waiting period and number of queries are of higher impact on energy consumption.

As most of the above queries involve waiting for crowd responses, we also conducted experiments to compare the energy consumption between running in the foreground (with the *LogicCrowd* interface displayed) and in the background (display is off while waiting). By sending a query to MTurk with a data size of 100 KB and varying the waiting period as 1, 2 and 3 hours, our results show that running the program in the background can reduce energy use in a given run by as much as 98-99% compared with running it in the foreground; i.e., the average power consumed is mostly spent on the display (around 90% on average of total battery usage during a run). Hence, a suggestion to save power while running the program (and waiting for responses) is to adjust the brightness to low light or even turn off the display while waiting. In addition, users can include suitable conditions in the crowd operators or choose a suitable operator in order to reduce the energy usage of queries by specifying when the number of crowd responses reaches the value that is set (e.g., use 'majority' if 'all' is really not required).

6.3.3 Discussion

In light of the results of the experiments, we conclude that the more the three factors increase in magnitude, the more energy is consumed. It is clear that each crowdsourcing platform has an effect on the energy usage of mobile phones in slightly different ways. Certainly, Mix queries consume the most energy of the four platforms (as we expect) whereas the query via Facebook used the least energy followed by Bluetooth and MTurk, respectively. By setting appropriate and suitable crowd conditions, crowd operators, and crowdsourcing platforms, users are able to reduce the energy consumption of the mobile when dealing with crowd querying. Moreover, to save energy, ensuring the program waits in the background for crowd responses is a practical and useful method. The overall conclusion from the experiments is that our approach is feasible on the user's own personal mobile platform for a range of methods used to reach the crowd, and that even a large

number of queries can be supported with relatively small impact on the battery level.

6.4 Summary

Energy consumption behavior is important for resource-limited mobile devices. We evaluated the *LogicCrowd* framework by conducting an energy analysis of *LogicCrowd* programs. In this chapter, we investigated the energy characteristics of the *LogicCrowd* prototype on actual mobile devices with two extensions of Prolog: 1) basic crowd predicate and 2) crowd unification. By using the findings to strategically select the most appropriate energy-related conditions, crowd operators, and crowdsourcing platforms, mobile users are able to reduce the energy consumption of their mobile when dealing with crowd querying. While our results are based on particular devices, our results are indicative of what we can expect in future better battery devices.

In the next chapter, the empirical testing in a lab and simulations will be explored in order to investigate the distribution of crowd tasks for mobile crowdsourcing in an opportunistic network and to construct basic scalability experiments.

Chapter 7

LogicCrowd in Peer-to-Peer Opportunistic Networks

In the previous chapter, we discussed the performance evaluation of the *LogicCrowd* prototype in terms of energy consumption. The energy characteristics of the *LogicCrowd* prototype with the first two extensions of Prolog including the basic crowd predicate and crowd unification have been explored in Chapter 6. In this chapter, we evaluate the performance of the *LogicCrowd* prototype in P2P networks. We conduct basic scalability experiments in order to study a task/query propagation strategy devised to support mobile crowdsourcing in intermittently connected peer-to-peer opportunistic networks. This chapter focuses only on P2P propagation and the collection of tasks/queries and their responses and also proposes an energy consumption model to estimate the energy used by both an owner (source node) and workers (mediator or terminal node).

This chapter is organized as follows. Section 7.1 gives a brief overview of P2P opportunistic networks. Section 7.2 introduces our basic network model and assumptions. Section 7.3 details the crowdsourcing task/query propagation strategy in intermittently connected P2P opportunistic networks. Section 7.4 discusses analytical results on peer responses and energy consumption, but for limited idealized well-structured networks. This is then compared to extensive simulation and evaluation results given in Section 7.5 to study response rates and energy usage behaviors in other types of randomly generated networks. Finally, Section 7.6 concludes the chapter.

7.1 An Overview of P2P Opportunistic Networks

Recently, crowdsourcing has emerged as a novel and transformative platform that engages individuals, groups, and communities in the act of collecting, analyzing, and disseminating environmental, social and other information for which spatiotemporal features are relevant [To et al., 2014]. Moreover, mobile computing and wireless networks have become an important part of our modern life. For example, in emergency situations such as finding a missing child or pet, the most effective action is to inform people nearby where the child or pet was lost and ask for nearby help for the search. In this case, real time ad hoc connections among nearby smartphone users are exploited for task distribution and a trajectory map may be generated for locating or predicting the child's or pet's location, or to map areas seen by volunteers. In another scenario, suppose Mary wants to meet up with her friends in a nice not-too-crowded restaurant. She can search for such a restaurant by deploying mobile peer-to-peer networks to ask people who are in nearby cafes or restaurants. With the popularity of mobile social networking and participatory sensing, mobile crowdsourcing has the potential to help tackle new problems in relation to real-time data collection and analysis, and coordination among a large number of participants.

Mobile devices are connected only intermittently when they opportunistically contact each other, yielding a concept known as Delay Tolerant Networks (DTNs)[Fall, 2003]. DTNs use a store-carry-forward paradigm to allow communication when a path through the network is not reliable due to frequent disconnections. A node receiving a packet from one of its contacts can buffer the message, carry it while moving, and then forward it to the encountered nodes for delivery. A network that routes packets using the store-carry-forward approach is also called an opportunistic network, because the nodes forward messages when an opportunity arises during an encounter or contact. With the unreliability and dynamism of mobile ad hoc or opportunistic networks, there are key issues in the development of crowdsourcing-related mobile applications that need to be considered. Due to the dynamic nature of moving hosts which may join or leave the platform at any time, the network topology is likely to change very often, and network partitioning can occur frequently. Moreover, communication range might be limited when a mobile user goes outside of a given location, causing unavailability of data (tasks or feedback from crowd) from that device at that location. A routing protocol is needed when the data needs to be transmitted between the two nodes.

Furthermore, traditional methods of data management in crowdsourcing generally consider only centralized or client-server communication while mobile ad-hoc network communication involves multi-hop transfers and decentralized processing. The issue related to resource constraints (e.g., energy) in mobile devices is also important. A node disseminates tasks among crowd workers through mobile peers in its range, obtains responses from such mobile devices, and integrates the responses to obtain real-time answers. The effectiveness of such processes could be limited by the lack of energy resources in mobile devices, hence there is a question of whether such peer-to-peer querying is feasible from the energy usage perspective. Moreover, it is not easy to estimate the response rates from peers using such an approach. To our knowledge, earlier work has not adequately studied or tackled peer-to-peer crowdsourcing in mobile environments with intermittently-connected opportunistic networks from the energy and response rate viewpoints.

Indeed, the energy consumption of mobile devices, such as smartphones, has increasingly become a concern from various sectors, ranging from smartphone manufacturers, mobile developers, to end users. Although battery capacity has been increasing in the past few years, the battery life of mobile devices is not catching up proportionally for a large spectrum of current applications. Mobile crowdsourcing applications utilise many mobile sensors such as an accelerometer, digital compass, gyroscope, GPS, microphone, and camera to form participatory sensor networks in order to enable public and professional users to gather, analyze and share local knowledge. However, participation in these systems can easily expose mobile users to a significant drain on already limited mobile battery resources. Particularly in opportunistic communication, mobile users/nodes with high social connectivity may quickly deplete their energy resources. Hence, the energy characteristics of mobile crowdsourcing in opportunistic networks is an important issue to be investigated.

The next section starts by introducing our basic network model and assumptions. While this chapter examines our *LogicCrowd* peer-to-peer prototype, we believe that our insights are generally applicable to other similar systems.

7.2 Basic Network Model

The fundamental dynamic network topologies investigated in our framework are based on spatial random node distributions and presumed short-range wireless connectivity between the nodes. We assume a network composed of N nodes.

Nodes are assumed to be randomly placed in an area A, interconnected by wireless links and have the same capabilities to store messages in a buffer of maximum size α. We can define a constant node density $\rho = N/A$, i.e. the expected number of nodes per unit area.

In mobile crowdsourcing in our case, a requester/originator propagates tasks/ queries to peers/workers whom it discovers. When receiving the queries, a worker answers the queries and replies to the requester. Moreover, the peer is able to forward tasks to others. This process might continue with subsequent peers. We assume that a user, and therefore a node, may act as both a requester and a worker in this network model. During the peer-to-peer propagation process, we define time-to-live and power-to-live values to limit the lifetime of tasks so that the action of forwarding tasks to other peers can be stopped. Responses are returned along the path tasks/queries are propagated. To model the wireless transmission between the nodes, a radio link model is assumed in which each node has a certain transmission range r and uses omnidirectional antennas. As illustrated in Figure 7.1, two nodes are able to communicate directly via a wireless link, if they are within range of each other. Only bidirectional links are considered.

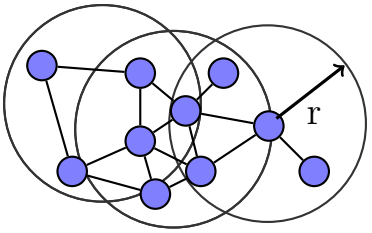


Figure 7.1: Modeling the topology of an ad-hoc network

We provide some basic definitions from graph theory and define the nomenclature used in this thesis. With the above network model, we represent an ad hoc network at each time instant as an undirected graph G. A simple graph G is a pair $G = (V, E)$ where V is a set of n nodes, called the vertices, and a set of m node pairs called the edges of G. The set of nodes, denoted by $V = \{1, ..., n\}$, represents the network enabled ad hoc devices; and the set of edges, denoted by E, represents the bi-directional wireless communication links (note that the link needs to be maintained for transfer of a task/query and a response). Hence, the size of a graph is the number of vertices of that graph. In graph theory, the degree of a node of a graph is the number of edges connected to the node. The degree of a node v, denoted as $d(v)$, is the number of neighbours of node v, i.e., its number of links. A node of degree $d = 0$ is isolated. The number $d_{min}(G) = min\{d(v)|\ v \in V\}$ is the minimum degree of G, the number $d_{max}(G) = max\{d(v)|\ v \in V\}$ is the maximum

degree. The average node degree of G is denoted as $d_{avg}(G) = \frac{1}{n}\sum_{v=1}^{n} d(v)$.

7.3 Task Propagation Strategy in Mobile Crowdsourcing

In this section, we describe the main characteristics of our propagation strategy. Mobile crowdsourcing refers to a distributed problem-solving model in which problems/tasks are propagated beyond the local database through public networks, especially mobile ad-hoc or opportunistic networks. Mobile users can easily interact with each other in a mobile network fashion which can be regarded as an ad-hoc network supporting multi-hop routing, content forwarding, and distributed decentralized processing. In our model, we assume that each peer is able to process and distribute their tasks without any control from the query/task-originating peer. Peers can convert a task/query it receives into another task when passing it on and the replies will be sent back along the same path (i.e., the reverse) as queries.

In our approach, the distributed task in mobile crowdsourcing is based on the opportunistic routing approach named Epidemic [Vahdat et al., 2000]. In the approach, messages to nodes, called carriers, are distributed within connected ranges of ad hoc networks. In this method, the messages are replicated at a highly increasing rate through a connected network. Epidemic then relies upon carriers coming into contact with another node (carrier) within range of the wireless network through node mobility. At this point, through such transitive transmission of data, messages have a high probability of eventually reaching their destinations. However, in our case, we do not have fixed destinations but aim to use such an epidemic approach to merely propagate tasks/queries to peers and then to collect results. It is possible that a peer might lose connectivity before sending its response; hence, we assume enough stability in the network connection so that a peer receiving a task/query from another peer can perform the task and then send the response back to the peer from which it obtained the task/query - this is an *optimistic* assumption. In general, the number of responses would be less than or equal to the number of peers receiving the tasks.

In our idealised model, forwarding from one peer to another is considered with a set of N nodes within transmission range r in a circular area πr^2 (with node density around each peer being $\rho = \frac{N}{\pi r^2}$). By pairing or multicasting, the task then is propagated and forwarded to other nodes within the range through

wireless interfaces such as Bluetooth, ZigBee and Wi-Fi Direct. However, one of the important issues in a forwarding strategy is that the nodes may suffer from a high level of redundancy with the same task being received or retransmitted by each node multiple times. To alleviate this problem, the task ID is acquired and recorded when each node receives the task sent to others. During the forwarding process, the task ID received from the source node is compared with the history of task IDs in the peer and then only a task with ID not being found in the table can be forwarded. In practice, the history might be a limited sized cache.

Moreover, to reduce or even avoid endless loops when propagating the task among modes in wireless ad hoc networks, Time-to-Live (τ) and Power-to-Live (ξ), which are defined in Section 3.5.2, are utilized in this chapter. τ is the value that limits the lifetime of the task propagation. Every task is tagged with a τ value which is modified (with time taken so far subtracted from it) across queries.

In our case, the τ value, which is the expiry/waiting period of the task, is reduced after the node forwards the task to another node. In the current node, the following equation is used to calculate a τ value for tagging the task before forwarding the task to another peer:

$$\tau = T_{source} - T_{forwarder} - \mu$$

where T_{source} refers to the end time (or time of expiry) from the task originator (who initiated the task and is waiting for answers sent back from other nodes in the network), while $T_{forwarder}$ defines the start time of the current node which is about to forward the task to another peer. μ relates to the time for transmitting packets for each device, including network set-up time or time for related computations in data transfer. The μ value could be limited by a default value preset in a user's device or a user-defined value. A minimum value of τ should be greater than or equal to μ of each device. (Note that this way of calculating τ provides the base case or largest possible value for τ. If time for returning answers α is to be considered, we would have $\tau = T_{source} - T_{forwarder} - \mu - \alpha$ i.e., we set $\alpha = 0$ in this simulation.) Figure 7.2 illustrates how to calculate the τ-tagged task of a node before forwarding the task to a neighbour.

The above τ-tagged tasks can solve or ameliorate the unbounded problem of distributing tasks in wireless ad hoc networks. Another possible propagation control parameter is ξ which we define as the maximum value of energy to be used. It is determined by each node that propagates tasks through the network. Such a parameter provides a means to take into account the limited resources of mobile

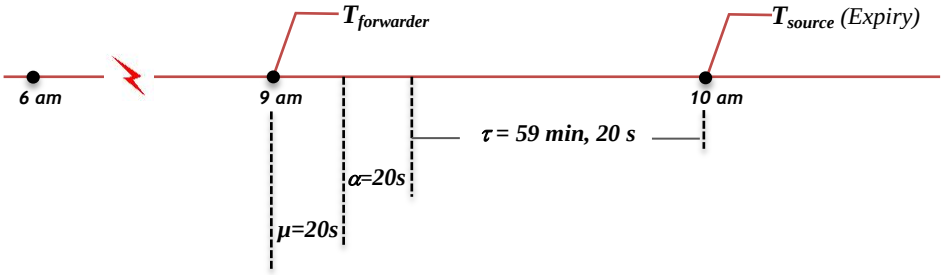


Figure 7.2: The diagram of calculating the τ value when propagating the task among nodes in wireless networks

devices, especially battery capacity, when propagating tasks. The ξ value is set by an energy budget (set by the system or user) corresponding to a current energy level. ξ can be calculated by the equation (3.1) in Section 3.5.2. Moreover, the task is allowed to be forwarded among peer networks by using the equation (3.2), mentioned in Section 3.5.2.

Moreover, both mechanisms can be managed to work cooperatively by setting the priority of ξ to be higher than that of τ. For example, in the case where we want to forward a task to other nodes, our strategy will check whether the value of ξ is sufficient for such forwarding. Normally, when ξ is found sufficient, the forwarding will be performed. However, if we lack sufficient τ or ξ, the task will not be forwarded. A node records queries it has received (each query is assumed to have a unique ID) and does not answer the same query twice.

We can also calculate the number of returned messages from the crowd. We assume that the number of nodes in the network with no isolated node is N nodes. Each node can compute the number of response messages throughout the τ interval, i.e., the number of response messages received by node i during τ is represented as follows:

$$R_i = \sum_{j=1}^{N} a(i, j, \tau)$$

$$\text{where } a(i, j, \tau) = \begin{cases} 1, \text{ if } j \text{ encounters } i\text{'s query and } j \text{ responses with an answer during } \tau \\ 0, \text{ otherwise} \end{cases}$$

7.4 Peer Responses and Energy Consumption in an Idealised Well-Structured Network

In the previous section, a method was presented to propagate the crowd tasks/-queries in mobile opportunistic networks. Moreover, we can find the total number of returned messages from the crowd during a run. However, we find that the number of responses or answers from peers can also be estimated by using task propagation conditions such as τ, the packet size of the crowd task, and network types. In this section, we concentrate on the prediction of the number of responses in the network using an idealised well-structured network, and propose an energy consumption model to estimate the energy used by both an owner (source node) and workers (including mediator or terminal node types).

7.4.1 Upper Bound Estimation for Peer Responses in Crowdsourcing Opportunistic Networks

We consider forecasting the number of response messages assuming that the type of network is a directed n-ary tree network. The n-ary tree network is a rooted tree in which each node has no more than n children (degree). In our case, the degree is the number of neighbour nodes who can receive a crowd task from the task's owner within a communication range (e.g., Bluetooth or Wi-Fi Direct range). To predict the number of returned messages, we categorize the task propagation of crowdsourcing in opportunistic networks into two aspects as follows.

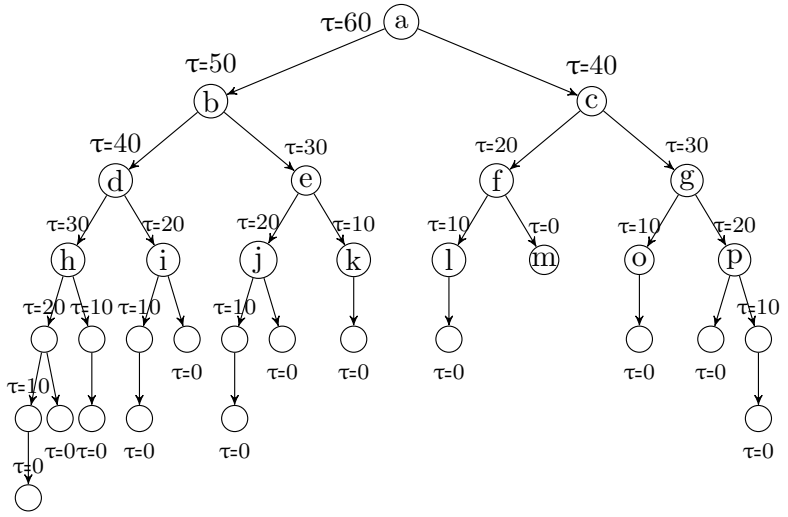
7.4.1.1 Point-to-Point Communication

Point-to-Point connection refers to communication between two nodes or end-points. Based on this connection, a node can only communicate or transmit information to one node at a time. In a mobile ad hoc network, the standard wireless link technology, such as Bluetooth, can also support point-to-point communication which follows the master-slave relationship. In Bluetooth, when two devices wish to connect each other, one of them will be a master and the other will become its slave. When such a master and slave pair is established, then the connection is formed. Such communication is applied to our task propagation study by sending tasks along path connections within transmission ranges of each device. We assume that the n-ary tree network has been deployed to be a graph model to estimate the total number of responses by peers.

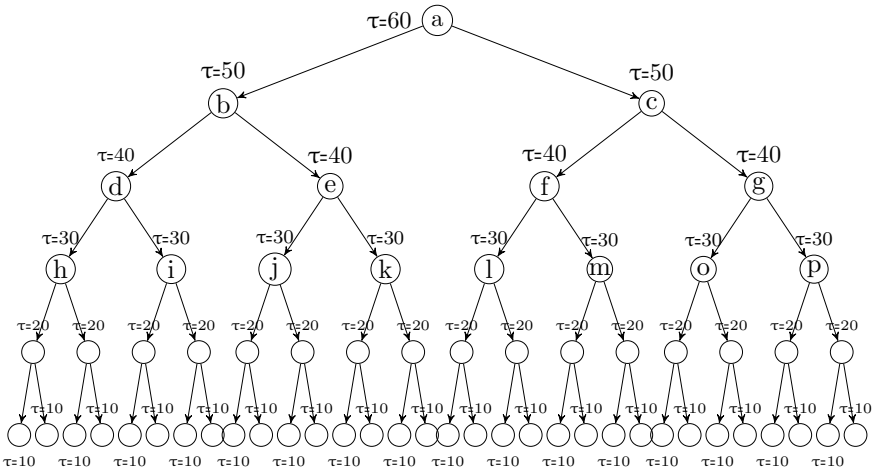
Figure 7.3(a) illustrates the propagation of crowd tasks on the n-ary graph tree network where the degree is 2, which means each node has at most two children. The crowd task is distributed among peers across the network by using a value to limit the lifetime of task propagation so that the action of forwarding tasks to other peers can been stopped. The crowd task is tagged with a τ value, the expiry/waiting period of the task, which is modified with the time taken so far (subtracted by the μ value throughout the network) as shown earlier. In Figure 7.3(a), for instance, τ in the original node is set at 60 seconds to wait for the returned answers from the crowd, and μ, which is the buffer (transmission) time for each device, has been defaulted at 10 seconds for every transmission. At time=0, when the original node 'a' sends the task to its neighbours ('b' and 'c') where 'b' had been previously selected, the connection will be established, and then the crowd task forwarded to 'b' will be tagged with τ=60 at node 'a'. After passing on the task (taking 10 seconds), the connection between node 'a' and node 'b' can be released and reestablished. Next, at time=10, node 'b' is permitted to forward that task to its neighbours using a new connection that has been created to node 'd' and the task is forwarded from 'b' to 'd' with the tag τ value set at node 'b', i.e. 50 seconds (i.e., = 60-10).

Concurrently, node 'a', after releasing the connection to 'b', at time=10, is able to pair with another neighbour (node 'c'), and pass the task to node 'c' taking 10 seconds to do so. So, at time=20, when node 'c' distributes the task to others, the τ value of node 'c' is set to 40 (=60-10-10) seconds for messages coming out from node 'c'. In the process of returning answers, the τ-tag value of nodes 'b' and 'c', i.e., the neighbours of node 'a', could not be the same value since it is connected by a point-to-point method, where connections and transmissions happen one at a time, i.e., when node 'b' and 'c' want to return answers to the original node (node 'a'), the connection could not be simultaneously established. For example, node 'c' must collect and return all peer answers to node 'a' after waiting up to 40 seconds (i.e., the τ-tagged value of node 'c') (or strictly speaking, waiting up to 30=40-10 seconds to allow 10 seconds to send results back but we simplify this in the rest of the discussion which follows, without affecting the overall general conclusions on the effect of wait times on number of peer responses). Note that the connection will be discarded on expiry. After node 'b' waits for at most 50 seconds (τ-tagged value of node 'b'), the new connection (between node 'a' and 'b') will be established to return all responses to node 'a'. Each node in the opportunistic network is able to recursively spread crowd tasks among peers until

$\tau \leq \mu$; note that we should consider also the time to transmit messages back (and use $\tau \leq \mu + \alpha$) but we make the simplifying assumption adequate to obtain upper bounds in the number of responses.



(a) Distribution of τ values with point-to-point connections (one at a time transfer)



(b) Distribution of τ values with point-to-multipoint connections (concurrent transfer)

Figure 7.3: τ value distributions in two types of network connections

In this propagation method, we assume that probabilities of each traversed node for responding and forwarding the task are γ= 1 and δ=1 respectively. This means that each peer in the network attempts to answer the received task and also forward it to their peers. Therefore, the upper bound of the number of returned messages could be predicted by using the τ-tagged value in each node. From Figure 7.3(a), the τ of each node with the same value is counted, as shown in Table 7.1. In Table 7.1, each *count* value represents the total number of nodes with the same τ value in the tree in Figure 7.3(a). The sum of the *count* values

excluding the *count* value at the original (node a, τ=60) (the shaded numbers in the table), hence, is the total number of returned messages. For this example, the upper bound on the number of returned messages is 32.

Table 7.1: Counts of nodes with the same τ values

N	0	1	2	3	4	5	6	Total
τ	60	50	40	30	20	10	0	
Count	1	1	2	3	5	8	13	32

It can be seen that the counts form numbers of the (in this case degree (d =)2-step) Fibonacci sequence:

$$F_n = F_{n-1} + F_{n-2}, n \geq 1, F_0 = 1 \text{ and } F_1 = 1 \tag{7.1}$$

In general, with a tree of nodes with degree d, we can estimate the upper bound M on the number of response messages from the crowd, which is the number of nodes who see the crowd task message (assuming that a node who sees the crowd task messages responses with a probably of 1.0), as the sum of the first $\left\lfloor \frac{\tau}{\mu} \right\rfloor$ numbers in the d-step Fibonacci sequence (excluding the originator $F_0^{(d)}$):

$$M = \sum_{n=1}^{h} F_n^{(d)} \tag{7.2}$$

where d is an integer, $h = \left\lfloor \frac{\tau}{\mu} \right\rfloor$, and

$$F_n^{(d)} = F_{n-1}^{(d)} + F_{n-2}^{(d)} + ... + F_{n-d}^{(d)} \tag{7.3}$$

for $n > 1$, and $F_0^{(d)} = 1$, $F_1^{(d)} = 1$ and $F_k^{(d)} = 0$ for $k < 0$. To estimate the upper bound of returned messages from the crowd, we can use this formula which has an exact solution.[1] In a network of higher degrees, the τ-tag value is calculated in a way similar to the calculation for degree two.

Also, for $\tau = j \cdot \mu$ for some integer j, an increase of μ in τ means M increases by $F_{j+1}^{(d)}$. Another increase of μ in τ results in M increasing by a further $F_{j+2}^{(d)}$. But note that $lim_{j\to\infty} \frac{F_{j+2}^{(d)}}{F_{j+1}^{(d)}}$ is the d-anacci constant, which approaches 2, in networks with high fan-out, or a very large degree d, i.e., each increase of μ in the wait time results in at most double the previous increase in the number of responses. For $d = 2$, i.e., (dropping the superscripts) for the Fibonacci sequence, $lim_{j\to\infty} \frac{F_{j+2}}{F_{j+1}}$ tends to ϕ, the golden ratio, and since $\sum_{n=0}^{h} F_n = F_{h+2} - 1$ (i.e., the sum of the first $h + 1$ (counting the first from $n = 0$) Fibonacci numbers is the $(h + 2)th$

[1] http://mathworld.wolfram.com/Fibonaccin-StepNumber.html

Fibonacci number minus 1), in the ideal case in a binary tree network, an increase in μ in the wait time can result in a fractional increase in the (upper bound in the) number of responses of at most $lim_{j\to\infty}\frac{F_{j+2}}{\sum_{n=0}^{j+1}F_n} = lim_{j\to\infty}\frac{F_{j+2}}{F_{j+3}-1} = \phi^{-1}$, i.e., $\approx 62\%$, since F_{j+2} is the increase, and $\sum_{n=0}^{j+1}F_n$ is the number of response where $\tau = j \cdot \mu$.

7.4.1.2 Point-to-Multipoint (Multicast) Communication

In contrast with point-to-point, the point-to-multipoint connection is established through one-to-many concurrent connections. In mobile ad hoc networks, the standard wireless communication technology such as Wi-Fi Direct, LTE Direct also supports these kinds of connections. Wi-Fi Direct[2] uses P2P Groups, which are functionally equivalent to traditional Wi-Fi infrastructure networks. The device implementing access-point like functionality in the P2P Group is referred to as the P2P Group Owner (P2P GO), and devices acting as clients are known as P2P Clients. When two P2P devices discover each other, they negotiate their roles (P2P Client and P2P GO) to establish a P2P Group. Once the P2P Group is established, other P2P Clients can join the group as in a traditional Wi-Fi network.

Figure 7.3(b) shows the distribution of crowd tasks using point-to-multipoint communication in a binary tree network. Note that, in general, we would be dealing with d-Cayley trees.[3] At the root of the tree (source node), the crowd task is distributed among peers across the network and the wait time clock is started here. In Figure 7.3(b), for instance, τ in the original node is initially set at 60 seconds to wait for the returned answers from the crowd, and μ has been defaulted to 10 seconds for every transmission between nodes. When the original node 'a' distributes, simultaneously, the task to its neighbours ('b' and 'c'), both nodes 'b' and 'c' are assumed to receive the task with τ=60 tagged at node 'a'. After nodes 'b' and 'c' receive the task, they attempt to forward it to other children nodes. When the task is forwarded to nodes 'd', 'e', 'f' and 'g', the task is tagged with $\tau = 50$ at nodes 'b' and 'c'. With point-to-multipoint connection, the τ-tag value of nodes 'b' and 'c' can be the same due to concurrency. When nodes 'b' and 'c' collect all their peer responses, the connections among nodes 'a', 'b' and 'c' have been established and then nodes 'b' and 'c' will return concurrently all answers to node 'a' after waiting at most 50 seconds. Each node in the opportunistic network

[2]http://www.wi-fi.org/discover-wi-fi/wi-fi-direct/
[3]http://mathworld.wolfram.com/CayleyTree.html

is able to do the same and recursively passes on the crowd task among peers until $\mu \geq \tau$.

With the same assumption as before, that is, each peer who receives the task/query will answer itself as well as forward it to others, the upper bound M on the number of returned messages in this case can be given as below, where we assume that the network is a complete n-ary tree which has degree d and height h of the tree:

$$M = \frac{d(d^h - 1)}{d - 1} \tag{7.4}$$

To see this, at level 0 there is $d^0 = 1$ node. The next level has d^1 nodes, and so on, with d^l nodes at level l. Hence, the total number of nodes will be $\sum_{i=0}^{h} d^i = \frac{d^{h+1}-1}{d-1}$. However, the root node (source) has to be subtracted since it cannot answer itself. Therefore, $M = \frac{d^{h+1}-1}{d-1} - 1 = \frac{d(d^h-1)}{d-1}$.

Suppose $h = \left\lfloor \frac{\tau}{\mu} \right\rfloor$ and $\tau = j \cdot \mu$ for some integer j. Now if the wait time τ is increased by μ, we will get d^j more responses, i.e. an increase of a factor of $1 + d^j/(\frac{d(d^{j-1}-1)}{d-1}) = 1 + \frac{d^{j-1}}{d^{j-1}-1} \cdot (d-1)$, which tends to d for very large j. For $d = 2$, say with $j = 5$, we have an increase in the (upper bound in the) number of responses of a factor of 2.0667, i.e., doubling the total number of responses. For $d > 2$, an increase in wait time of μ could increase the total number of responses by a factor of d.

In summary, in an idealised network such as the above, (in both point and multipoint case) it always pays to wait a further μ time units to allow for a potentially sizeable increase in the number of responses. However, this increase may not happen in real networks (and the increase in wait time is only possible if the user agrees to it and if the extra energy consumed is bearable). This is because in reality there could many nodes without links to any nodes, that is, the out degree of many nodes could be zero instead of d. Then one should consider networks with a degree distribution, that is, the degree of each node is given by a probability distribution, $P(d)$ gives the probability that a node has degree d. Instead of the above, we then replace d with the expected value of d, given by $E[P(d)]$. For example, suppose only 5% of the nodes connect to $d = 3$ other nodes (and the rest not connected to any) so that $E[P(d)] = 0.15$. The above analysis then suggests that the number of responses might only increase by a factor 0.15 for an increase in wait time of μ. Also, in general, if the probability of a node receiving a crowd task responding with an answer (regardless of whether

it forwards the crowd task on or not) is γ, then the number of responses might only increase by a factor $\gamma \cdot 0.15$ for an increase in wait time of μ. In fact, it can be non-exponential, as our experiments later show that an increase in wait time results in only linear increases in the number of responses in random networks.

For scale-free networks where $P(d) = d^{-x}$ for some constant x, there are nodes which are effectively hubs with many links and some nodes with very poor connectivity, hence a strategy then might be to forward crowd tasks mainly to such hub nodes - however, to determine such hub nodes will require metadata and other work (not within the scope of this thesis).

7.4.2 Energy Consumption Models of Mobile Crowdsourcing in Opportunistic Networks

In this section, we study the energy consumption characteristics of mobile crowdsourcing in opportunistic networks. The energy usage of requesters and workers in mobile crowdsourcing is described. Also, the factors that affect the drain on mobile battery life in ad-hoc networks are examined. Finally, models to simulate the energy consumption of mobile crowdsourcing in opportunistic networks are proposed.

There are three different types of users in our mobile crowdsourcing on opportunistic networks.

- *Source*: Source acts as a requester who can create and distribute the task, and also collect completed answers from workers. When initiating a crowd task, the source will attach task details called crowd conditions which are parameters for distributing tasks to workers. Crowd conditions include Time-to-Live (τ), Power-to-Live (ξ), any media files path related to the task, an authorized friends' list (which is the list of friends who are permitted to answer the task), and, if applicable, even the condition in which the source allows workers to forward the task to others. In distributing tasks, an encountered node in the connection range or a friend on the list is selected and then the task is sent to him/her. After a waiting period (τ), each worker gathers the completed answers and sends them back to the source.
- *Mediator*: Mediator acts as a worker and a requester. In general, the encountered node/mediator has three key functions: to receive the task from another node, to perform that task and, based on crowd conditions, to forward that task with the original conditions (and modified parameters such

as an updated τ) to other workers. During the answering process, each mediator returns feedback or answers along the same path which they have been traversed. The task will be disseminated among other nodes in the network depending on the permissions for forwarding the task from the original source and the willingness of nodes to forward it to others. The task will be active on the network until timeout (τ).

- *Terminal*: Terminal acts as a worker who only returns the answer to the source without forwarding that task to others.

In mobile crowdsourcing especially on opportunistic networks, nodes communicate with each other via multi-hop and decentralized processing. Hence, a node is able to involve many intermediate relay nodes (mediators) in order to forward the task between them. The effectiveness of such processes could be limited by a lack of energy resources on mobile devices. Here, we study the following factors in mobile crowdsourcing in opportunistic networks which influence energy usage in mobile nodes.

1) Waiting Period - Waiting period is an important factor affecting energy consumption in mobile crowdsourcing. A waiting period or an expiry time is set as part of the crowd conditions for a crowd task. This time is used to determine how long a node should wait to collect answers back from peers. The waiting period or τ value is decremented and used to tag the crowd task of each node during the propagating process as described earlier. Determining the appropriate waiting period on crowd tasks is not only to give adequate time to obtain crowd responses but also to improve energy efficiency for mobile nodes in such opportunistic networks.

2) Task Size - In general, crowd tasks can contain multimedia files such as messages, images, voice-recording or video. There is a large volume of such tasks that contain media files. In opportunistic networks, the packet size and data transfer rate affect energy consumed in each mobile node during the task propagation process with either multi-hop or single-hop connections.

3) Network Connection - As mentioned in our work, there are two idealised topologies for network communication 1) Point-to-Point Communication, and 2) Point-to-Multipoint Communication. The communication between nodes might have different types of network connectivity and also have different bandwidth. Hence, the total number of messages or responses returned to the source node can vary with each initiated task, and such conditions have significant impact on

energy usage for each node in mobile crowdsourcing, especially in opportunistic networks.

4) Node Types - We categorised nodes as being in three different roles: source, mediator and terminal. They are also classified into three messaging types: (1) sender (2) receiver, and (3) forwarder. Therefore, node roles and types are factors to be considered in the energy consumed by each node.

5) Degree of node - The degree refers to the number of neighbour nodes who can receive crowd tasks from the source within the communication range. This factor is related to network density. The higher the node density the more returned messages can be received from the crowd. However, the relationship between node density and energy consumption is an interesting issue to be addressed in ad-hoc networks.

6) Background/Foreground Display Modes - The brightness of the user display mode on mobile device has been known to be one of the major battery consumers. With mobile crowdsourcing, waiting for crowd results is, hence, a key issue that impacts energy waste in nodes. Choosing an appropriate mode is thus significant to save the battery life for the mobile device.

These six aforementioned factors influence the energy usage of mobile crowdsourcing in opportunistic networks.

The overall energy consumption model of a node consists of two main parts: the first is the contribution from energy usage during the run-time process ($E_{runtime}$) and the second is the contribution from energy spent during the idle-time process ($E_{idletime}$). The run-time process refers to the processes of the node when its sends, forwards, or receives the crowd tasks (i.e., the data packets) among peers whereas the idle-time process relates to the time spent on waiting for crowd responses or results. Thus, the first component can be derived by multiplying the overall transmission time, denoted by $T_{runtime}$, and the power consumption rate, denoted by $P_{runtime}$ of battery during the run-time process. The function $T_{runtime}$ is largely dominated by communication and can be computed by multiplying the network transmission rate (rt), the transferred packet size (p), and the number of neighbour nodes (m). The second component is derived by multiplying the idle time ($T_{idletime}$) and the power consumption rate ($P_{idletime}$) of battery during the idle-time process. The function $T_{idletime}$ is defined by subtracting $T_{runtime}$ from the waiting period (τ) (which is specified in crowd conditions for the crowd task

as seen earlier).

$$E = E_{runtime} + E_{idletime} \tag{7.5}$$

$$E_{runtime} = T_{runtime} \times P_{runtime} \tag{7.6}$$

$$T_{runtime} = rt \times p \times m \tag{7.7}$$

$$E_{idletime} = T_{idletime} \times P_{idletime} \tag{7.8}$$

$$T_{idletime} = \tau - T_{runtime} \tag{7.9}$$

For instance, suppose a node wants to distribute crowd tasks among peers on a point-to-point communication network with a transfer rate of 2.1 Mbps (rt = 1/2100 seconds per KB). The size of such a task is 3000 KB (p=*3000*) and there are 5 neighbour nodes (m=*5*) connecting to it. The waiting period in the crowd condition is set at 20 minutes. Also, suppose that there are power consumption rates on the runtime process ($P_{runtime}$) of 0.003% (of battery life) per second and the power usage rate on an idle-time process ($P_{idletime}$) of 0.002% per second. Substituting *rt, p*, and m in equation (7.7), we have $T_{runtime}$ = 7.14 seconds and we can calculate $T_{idletime}$ = 1192.86 seconds. Substituting $T_{runtime}$ and $T_{idletime}$ in equation (7.6) and equation (7.8) respectively, we find that the total percentage of battery usage of the source node calculated by equation (7.5) is 2.41%.

7.5 Simulation and Evaluation Results

In the previous section, ideal networks are described for the purpose of comparison. In this section, we explore empirical testing in a lab and simulations to study the distribution of crowd tasks for mobile crowdsourcing in an opportunistic network and to construct basic scalability experiments. We create a mobile crowdsourcing simulation model based on the basic operations of the *LogicCrowd* program [Phuttharak and Loke, 2013, 2014b] which is implemented on real devices. But to explore a range of different aspects of scalability in the ad-hoc network, a simulation model is required. The mobile crowdsourcing simulator was developed in NetLogo[4], a multiagent modeling language that provides a programmable environment for simulating natural and social phenomena. In this work, we use NetLogo to simulate the crowd network as the opportunistic network and perform experiments on two different aspects: crowd task propagation in such networks, and energy consumption. The purpose of the simulation results presented in this

[4]https://ccl.northwestern.edu/netlogo/

section is two-fold: (a) to analyze the relationship between the parameters of mobile crowdsourcing and the number of response messages from the crowd, and (b) to study the energy consumption characteristics of mobile crowdsourcing, especially in a large scale ad-hoc network.

7.5.1 Simulation with NetLogo and Scenarios

Figure 7.4 is the simulator interface that shows how crowd task propagation operates in an opportunistic network. Nodes are linked with nearby neighbour nodes in communication distances and then they are connected to the large network. In the model, we specify the parameters, namely types of nodes, waiting period, task size, the number of neighbour nodes, the probability of a peer responding (which we assume uniform, for simplicity), display modes, and energy budget allocated for crowd tasking.

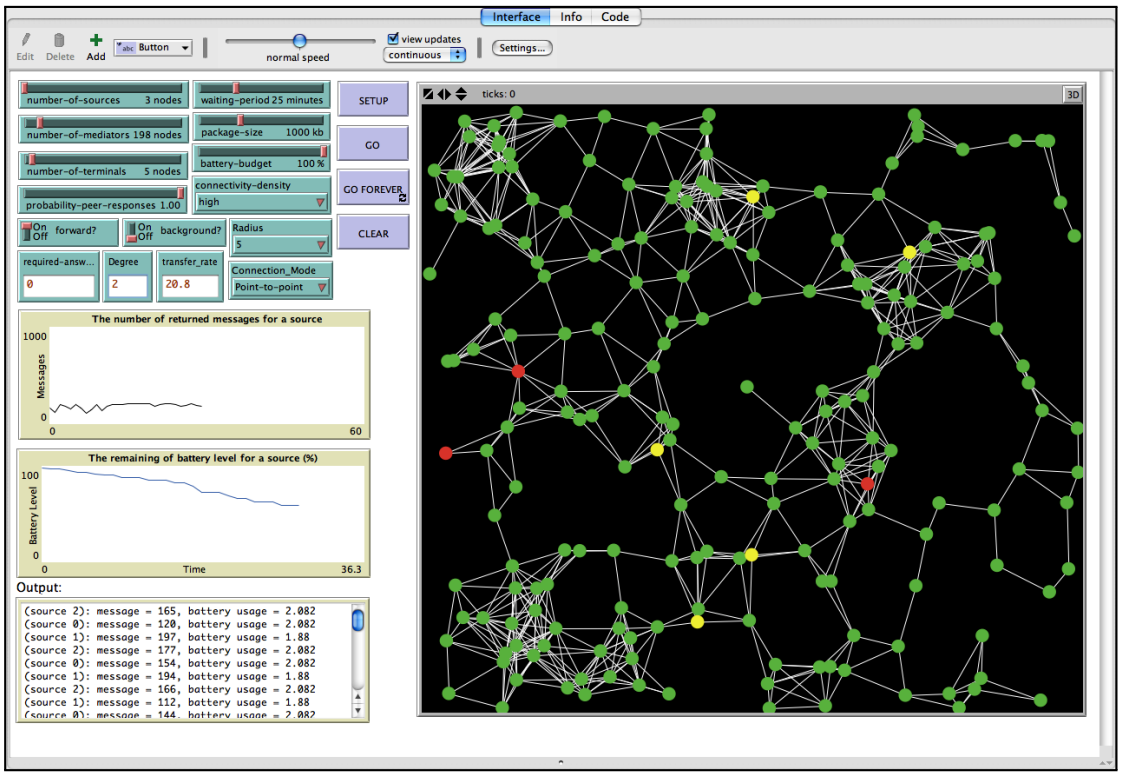


Figure 7.4: The mobile crowdsourcing simulator interface

The simulation model assigns different values for the parameters of mobile crowdsourcing in order to study their impact on the sending and receiving of crowd tasks and on the battery usage of the nodes within a simulation execution. All details on parameter settings can be found in Table 7.2. We fixed the area size over which the network spans at 300 x 300 square meters (the ratio is 1 meter = 1 patch in NetLogo). By creating nodes randomly positioned within

this area, we generated 30,000 nodes. These 30,000 nodes were connected to each other with communication distances of 10 meters. When any nodes are within a communication range, they are then assumed to be connected up to the given degree setting.

For example, if the degree was set to 7, this means that each node was connected randomly to neighbours within his/her range of 7 nodes or less. In this experiment, we determine the degree of each node that was up to 10 and there were no isolated nodes. We assumed the probability of each node for responding and forwarding the task was equal to 1, in that every node in the network who received the crowd task was willing to reply to the origin and to forward such task to his/her neighbours. During measurement, the waiting period in the crowd task condition is varied by increasing in 10 minute intervals from 10 minutes up to 60 minutes. Also, the packet size started at 100 KB and was increased in 500 KB intervals up to 3000 KB. Two types of network connections are explored during the experiments including (1) point-to-point and point-to-multipoint, and both display models (2) background (display not visible to the user) and foreground (display visible to the user). We ran our experiments with random graphs as illustrated in Figure B.1.

Table 7.2: The parameters of the different scenarios

Parameters	Values for Scenarios
Area size	300x300 m^2
Communication Distances	10 meters
Number of nodes	30,000 nodes
Probability of peers forwarding	1.0
Probability of peers responding	1.0
Waiting period	10-60 minutes
Packet size	100-3000 KB
Energy budget	100%
Network connection	Point-to-point/Point-to-Multipoint
Degree of node (uniform)	1-10
Display mode	Foreground/Background

7.5.2 Peer Response Analysis

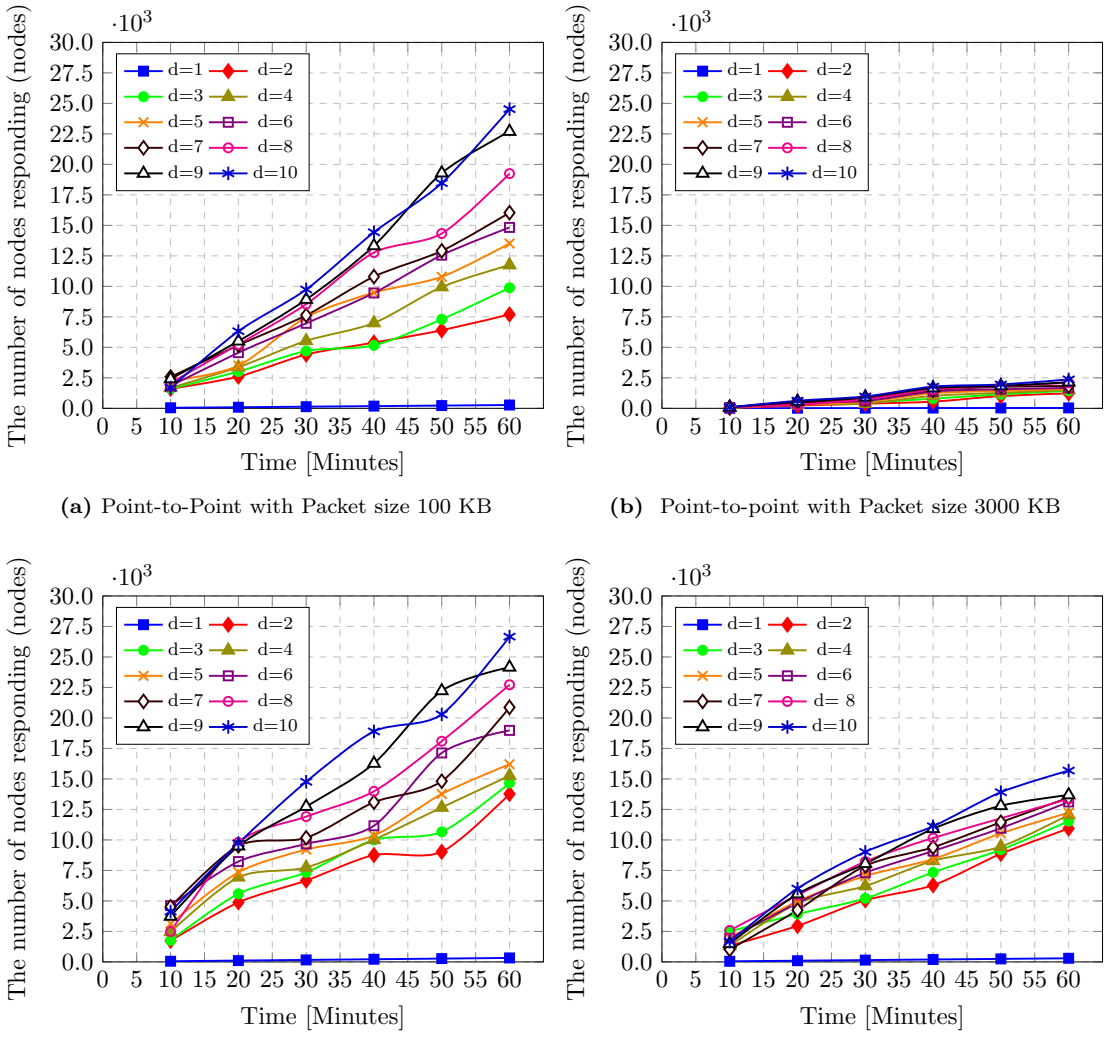
Figure 7.5 shows the number of nodes responding with varying waiting periods on the origin, compared among different degrees of nodes (1-10), varying numbers of packet sizes and two different types of network connections (point-to-point and point-to-multipoint). Figure 7.5(a) presents the results for waiting periods from 10 minutes up to 60 minutes. It is apparent that the longer the waiting period,

the higher the number of nodes responding, which we expect. Note that at the waiting period of 10 minutes with nodes of degree 2 and packet sizes of 100 KB, the number of nodes responding was 1607 nodes, and when the waiting period was moved to 20, 30, 40, 50 and 60 minutes with the same conditions, there were 2601, 4411, 5388, 6400 and 7700 nodes of peers responding to the source node, respectively. Note that for the random graph we used for this experiment, the increase in responses grows linearly with wait time on the origin, in contrast to the optimistic settings in the idealised graphs we saw earlier.

The graph displays an explicit increase in the number of peers returning the task answers when the degrees of node rose from 1 to 10. For example, asking the crowd via the point-to-point network connections for the waiting period of 20 minutes, and packet size of 100 KB, but with varying node degrees of 1, 3, 5, 7, and 9, the number of peers responding was 90, 3012, 3497, 5176, and 5506 nodes, respectively. It must be noted that we are using a very high density scenario - lower density scenarios will result in lower response rates. However, the results rely more on the node degrees (the actual connectivity of the nodes). For example, a lower density of nodes but still similar connectivity (within range of networks) will give similar results. An effect on high density might be on lowering bandwidth of wireless connections but these effects are out of scope of this study.

In contrast, the results illustrated in Figure 7.5(b) clearly show that packet size was increased markedly to 3000 KB from 100 KB, the larger the packet sizes, the smaller the number of nodes return answers for the same waiting periods since more time is required for transmissions. For example, in the case of packet size 100 KB, a 40-minutes waiting period, and node degrees of 4, 6, 8 and 10, the count of peers responding were 6999, 9460, 12772, and 14433, respectively. When measuring with packet sizes of 1000 KB with the same criteria above, the total number of nodes responding was 2693, 3035, 3590, and 4011, respectively.

However, one of the factors that impacts on the number of responses from peers is the type of network communication. Figures 7.5(c) and 7.5(d) show the number of nodes responding using the point-to-multipoint connection with varying degrees of nodes and different packet sizes. For example, the number of returned responses from peers at the degree of 6 with the task size of 100 KB and waiting for 30 minutes in the one-to-one connection was 6968 nodes, as shown in Figure 7.5(a). When each node was connected with the point-to-multipoint connection with the same conditions as above, the number of peers responding was about 9671 nodes, as shown in Figure 7.5(c). It was considerably more than the number of returned



(a) Point-to-Point with Packet size 100 KB

(b) Point-to-point with Packet size 3000 KB

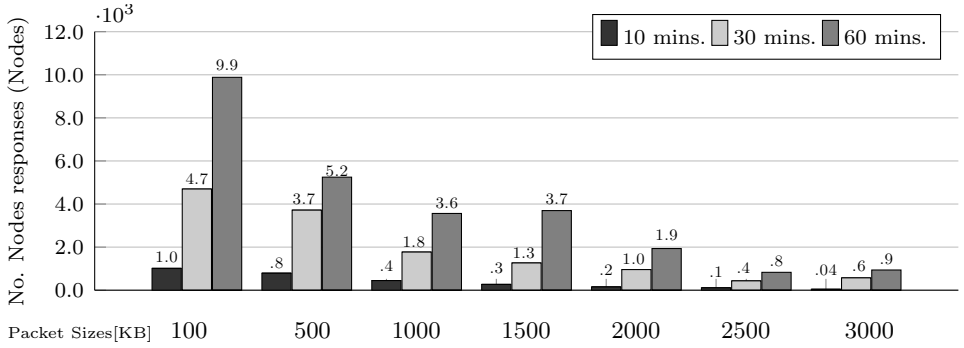
(c) Point-to-Multipoint with Packet size 100 KB

(d) Point-to-Multipoint with Packet size 3000 KB

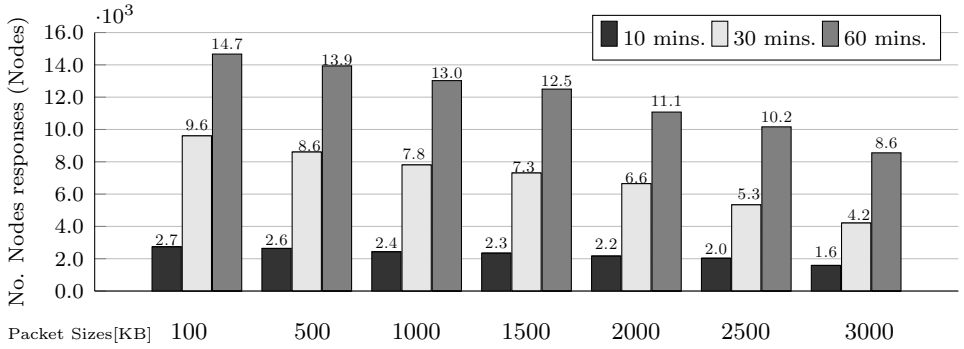
Figure 7.5: The relationships between the number of nodes responding and waiting period for different data packet sizes, degrees of nodes and network connections (d = degree of nodes)

responses from peers with the point-to-point connection by around 1.4 times. The point-to-multipoint connection shares a similar trend to the point-to-point connection. The higher the degrees of nodes and the waiting period, the more nodes returned answers in a simple linear increase, again, as expected. However, when the packet size was increasing, the number of nodes responding goes down.

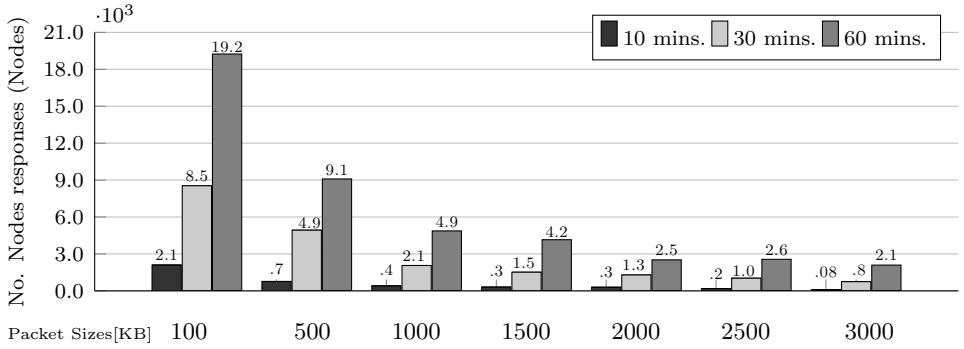
To show other dimensions of the results, Figure 7.6 displays the number of nodes returning responses with varying packet sizes for different waiting periods, network connections and degrees of nodes. In the figure, the packet sizes had been increased from 100 KB to 3000 KB in equivalent intervals (500 KB). Each group of packet sizes was compared among three different waiting periods (i.e., 10, 30,



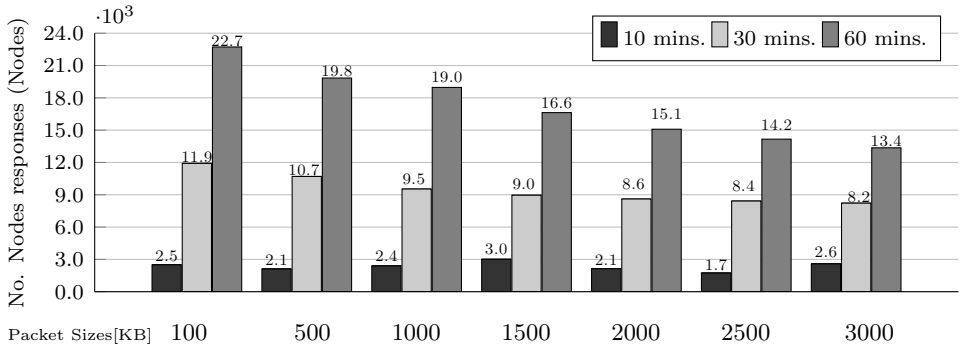
(a) The number of nodes responding with point-to-point, with degree 3



(b) The number of nodes responding with point-to-multipoint with degree 3



(c) The number of nodes responding with point-to-point with degree 8



(d) The number of nodes responding with point-to-multipoint with degree 8

Figure 7.6: The relationships between the number of nodes responding and packet sizes for different degrees of nodes and network connections

60 minutes) and two types of connections. The results were similar to Figure 7.5. As examples, the results of degrees of 3 nodes and 8 nodes are shown in Figure 7.6(a),7.6(b) and Figure 7.6(c),7.6(d) respectively. For example in Figure 7.6, with the nodes of degree 3 at the waiting period of 60 minutes and point-to-point connection, when the packet sized was 100, 500, 1000, 1500, 2000, 2500 and 3000 KB, there were 4697, 3717, 1771, 1265, 957, 573, and 434 nodes of peers responding to the source node, respectively. While each node was connected with the point-to-multipoint connection with the same conditions as above, the number of peers responding were 9604, 8604, 7806, 6647, 5336, 4210 shown in the Figure 5(b).

We can observe the relationship between packet size and the number of peer responses, that is, the larger packet size of tasks was propagated, the fewer the answers returned. However, if we want to increase the number of responses from peers, we have to expand waiting time or nodes must connect with a high degree of neighbours whether the type of connection is point-to-point or point-to-multipoint. Consider that the number of returned messages was around 1000 messages in the experiment. From the findings at a packet size of 1000 KB with nodes of degree 2 and connected by point-to-point, the original node had to wait for 30 minutes to reach 1000 returned messages from peers, whereas, the source node had to wait longer up to 40 minutes to get 1000 messages when a packet size of task was 2000 KB with node degrees of 2 and point-to-point connection. For the largest packet size (3000 KB) to be distributed with the same conditions, the origin node had to wait for 50 minutes in order to achieve 1000 returned answers.

Moreover, we noticed that if the degree of nodes rises to 10 with the same packet size, it needs a shorter waiting time to reach the same number of peers responding. For example, in nodes of degree 10, at a packet size of 1000 KB using point-to-point connection, the source spent just only 20 minutes to receive 1000 returned answers, while the origin node waited for 30 minutes to get 1000 results from peers at a packet size of 2000 KB, with node degrees of 2 and connecting with point-to-point. For the biggest size of a packet at 3000 KB, the source spent for 40 minutes to wait for the returned messages from nodes with the same conditions as above.

7.5.3 Energy Consumption Analysis

In our energy consumption study, we measured the power consumption of mobile applications via changes in battery levels. Although battery level changes are

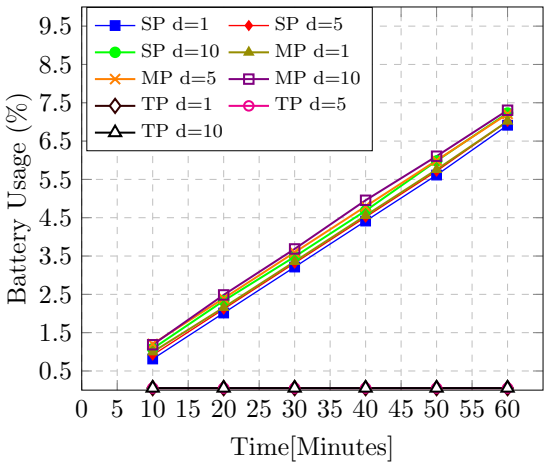
coarse-grained, it can be easily collected through a user-level application on mobile devices. Also, we aim to have findings that are understandable to, and can benefit mobile users when they try to estimate the battery consumption before making queries to the crowd using different platforms and methods.

The energy consumption models (equations 7.5-7.9) proposed in Section 7.4.2 have been used for estimating the energy consumption for each node in the experiments. And the experimental set up is the same environment and network structure as previous section (Section 7.5.2). All details on parameter settings for these experiments can be found in Table 7.2. The experiments and energy analysis are discussed as follows.

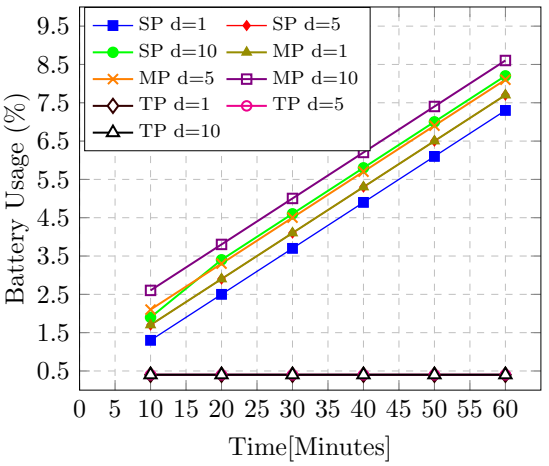
Figure 7.7 shows the percentages of battery level changes with varying waiting periods (10 to 60 minutes), compared among three types of nodes with different node degrees (i.e., 1, 5, 10), varying data packet sizes (100, 3000 KB), and two types of network connection with foreground display mode. In Figure 7.7(a), the percentage of changes in battery level of the source and the mediator was not much different across the various waiting periods and degrees of node with data packet of 100 MB. For example, the energy consumed by the source node at the degrees of 1, 5, 10 nodes at waiting period of 60 minutes was about 7.213%, 7.266%, 7.332% whereas the power usage of the mediator in the same condition was about 7.266%, 7.319%, 7.385%. Note that the terminal nodes consumed the least energy around 0.053 % at 1,5,10 node degree and 60 minute-waiting. According to these results, during the process of task propagation, the mediator seems to consume more battery than the source node while the terminal nodes consume the least. Moreover, the power consumption of the source and the mediator constantly increased in a simple linear form when the waiting period moved from 10 up to 60 minutes. It is apparent that with the longer waiting period, the more energy is consumed. However, the energy used by the terminal node stayed steady despite an increase in the waiting period from 10 to 60 minutes. The reason is that when a terminal node processes the task, it just only receives the task and returns the answer to the source without forwarding that task to others. The terminal node does not have to wait for time expiry to collect the answers from neighbours.

Figure 7.7(b) reports the similarities of energy consumption when a task was sent to the crowd for different data packet sizes (here, 3000 KB). There is a slight difference in battery consumption of the source and mediator when we increase the number of packet sizes. For example, the battery usage of the source node at node degrees of 1, 5, 10 at the waiting period of 60 minutes was about 7.3%, 7.701%,

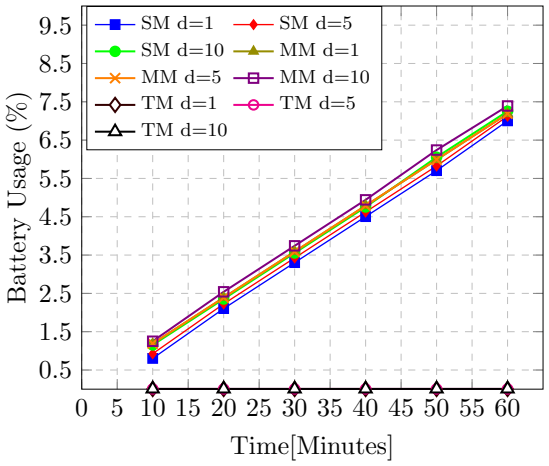
8.202% whereas the energy usage of the mediator node in the same condition was 7.701%, 8.102%, 8.603%, respectively.



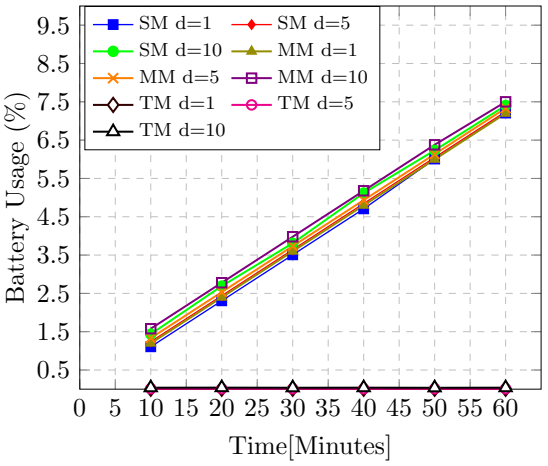
(a) Point-to-Point with Packet size 100 KB



(b) Point-to-point with Packet size 3000 KB



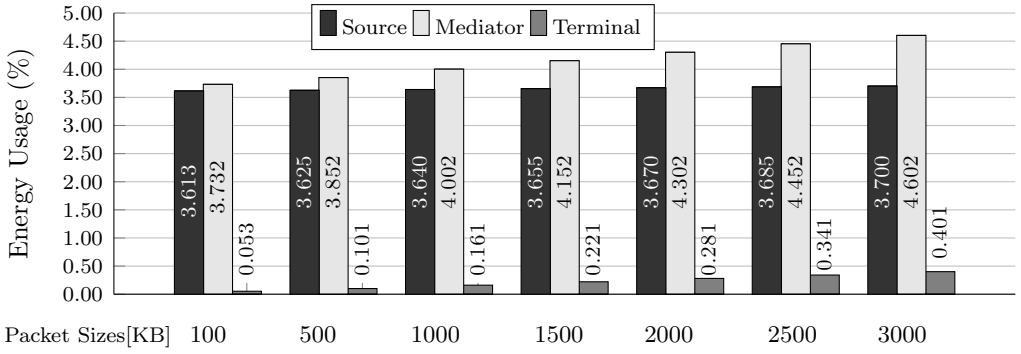
(c) Point-to-Multipoint with Packet size 100 KB



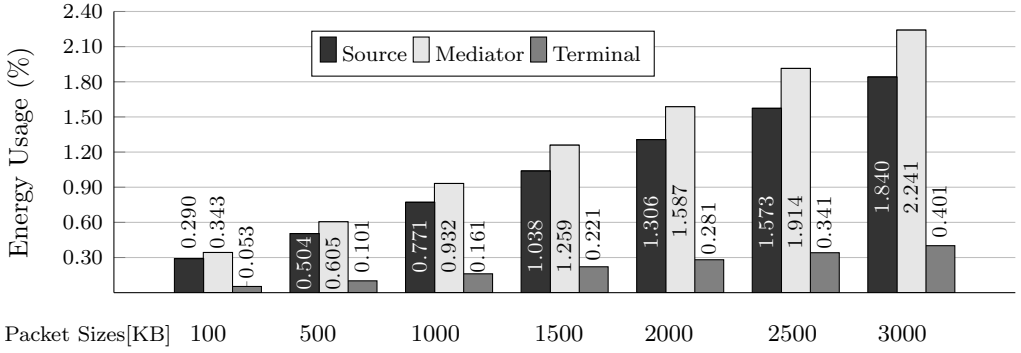
(d) Point-to-Multipoint with Packet size 3000 KB

Figure 7.7: The relationships between the energy usage of a mobile and waiting period for different node types, data packet sizes, degrees of nodes and network connections (SP = the source node with point-to-point connection, MP = the mediator node with point-to-point connection, TP = the terminal node with point-to-point connection, SM = the source node with point-to-multipoint connection, MM = the mediator node with point-to-multipoint connection and TM = the terminal node with point-to-multipoint connection)

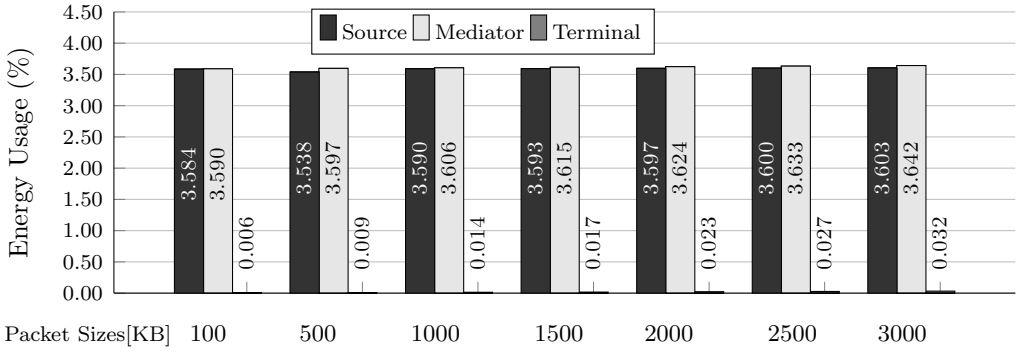
The terminal node consumed energy of around 0.401% with the same condition as mentioned above. Figure 7.7(c) and 7.7(d) show the percentage of changes in battery levels with the point-to-multipoint connection. It clearly shows similar trends to those in the point-to-point communication shown in Figure 7.7(a) and 7.7(b). However, they consumed less energy than when the nodes are connected using the point-to-point method. For example, the energy consumed by the source



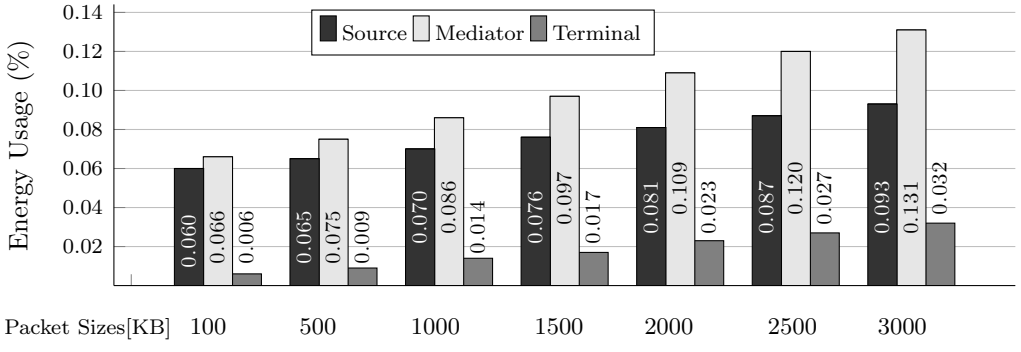
(a) Point-to-point connection with foreground mode



(b) Point-to-point connection with background mode



(c) Point-to-multipoint connection with foreground mode



(d) Point-to-multipoint connection with background mode

Figure 7.8: The relationships between the energy usage of three kinds of nodes and packet sizes using a degree of 6 and waiting period of 30 minutes

node at degrees of 1,5,10 at the waiting period of 60 minutes is 7.203%, 7.225%, 7.253% for packet sizes of 100 KB and 7.206%, 7.243%, 7.403% for packet size of 3000 KB. While the energy spent by the mediator is about 7.188%, 7.19%, 7.191 % for packet size of 100 KB and 7.197%, 7.233%, 7.278% for packet size of 3000 KB with node degrees of 1,5,10. Finally, the terminal node spends the least energy, of around 0.005% for packet size of 100 KB and 0.011% for packet size of 3 MB with the same conditions.

Figure 7.8 compares the percentages of changes in battery levels used among the source, mediator, and terminal with the varying number of packet sizes (100 – 3000 KB). In the Figure 7.8(a) and 7.8(b), the energy consumption of nodes between the background and foreground modes are presented respectively with point-to-point connection at a degree of 6 nodes and 30 minute-waiting. On the other hand, Figure 7.8(c) and 7.8(d) presents the power consumption of nodes between the background and foreground setting display modes with point-to-multipoint connection at the degree of 6 for 30 minutes respectively.

As can be seen, the energy used by the source, mediator and terminal nodes slightly increases when the packet sizes rise from 100 to 3000KB. For example, in Figure 7.8(a), the energy consumption of the source at packet size from 100 to 3000 KB with the foreground mode was 3.613%, 3.625%, 3.640%, 3.655%, 3.670%, 3.685%, and 3.700% respectively. Additionally, the energy consumption of the source and mediator with their foreground setting was considerably higher than the power consumption of such nodes with the background mode. For example, in Figure 7.8(a) and 7.8(b), the energy consumption of the source and mediator at packet size of 500 KB with their foreground mode was 3.625%, and 3.852%, while for the background mode, they were 0.504%, and 0.605%. However, the battery levels of terminal nodes, either foreground or background, were equal because they only consume time when they respond to the source node without forwarding the task to others.

Note that the mediator spends the most energy, compared with all types of nodes of either any display setting (foreground, background) or communication type (point-to-point, point-to-multipoint). The reason for this is that the mediator drains the energy during the answering and forwarding task process. Furthermore, it clearly shows that the power consumption of nodes with point-to-point consumed significantly higher energy than the energy consumption of nodes with point-to-multipoint connection. For example, in Figure 7(b), the battery usage of

the source, mediator and terminal nodes at 1500 KB with the point-to-point connection was 1.038%,1.259%, and 0.221%, while the battery level of such nodes at the same conditions but connection with point-to-multipoint was 0.076%, 0.097%, and 0.017%. We note that the energy usage of mediation is adequately low. Playing the role of mediator is, hence, not prohibitive in such networks.

7.5.4 Discussion

We next summarize our key findings regarding the number of peer responses and energy consumption. There are four main factors that impact on the number of responses from peers, namely degree of nodes, task size, waiting period and network connection. The results show that the higher the node degree and the waiting period, the more answers we can expect. In contrast, when the packet size increases, the number of nodes responding reduces. For the factor of network connection, it clearly shows that the number of nodes responding with point-to-multipoint communication is much more than the number of peers returning answers with point-to-point connections. Moreover, the probability of the node responding to the task and the probability of the node being willing to forward the task to others are also crucial factors that influence the number of responses. Each peer is allowed to copy and send the task to others. As a result, the task is able to propagate beyond the communication range of the original node. In this regard, if each peer is willing to forward the task to its friends, there is a higher tendency to obtain more answers. However, this ability to forward the task imposes limitations. In our case, each peer/mediator will return answers along the same path which they have traversed during the answering process. Hence, if the peer moves out of connection range before the answers are returned, these answers are not able to be sent to the origin.

When we compare the total number of returned messages in real experiments with a computing model using idealized well-structured networks in Section 7.4, we find that the results with any conditions from measurements are not higher than the upper bound of the results predicted by using idealized models. For example, the number of peers responding calculated by a prediction model is 1595 when the degree of 2 is set with the task size of 1000 KB and waiting time of 10 minutes in a point-to-point connection. However in the experiment, each node was connected with the same conditions as above, but the number of peers responding is 315.

From the experiments, it was found that the total number of responses seldom reaches the idealized results even if the waiting period or degree of nodes increased. We found that there are some factors that have impacted on the number of responses during measurement. Firstly, based on a task propagation strategy, we used a task ID to eliminate a high level of redundancy with the same task being received or retransmitted by each node multiple times. As a result, the number of peers responding decreased much more than expected. After any two nodes were connected, the task ID received from the source is compared with a history of task IDs in the destination node and only a task with ID not being found in DB can be propagated. In contrast, the task with ID being found in the table of the destination is not forwarded, then the connection is discarded and the source searches for other neighbour nodes in order to propagate again. By doing so, the time in waiting period (τ) is lost during the processes of making connections and verifying task IDs.

Secondly, density is one of the important factors that affects the total number of peers responding. The higher the node density, the more connections there are. Thus, the task is able to be propagated to more neighbours who are connected to the source within the communication range. In our experiments, the density of the testing area is about 0.33 $node/m^2$ (there are 30,000 nodes within the area size at 300x300 m^2). With the large size of network experimental area and low density, the total number of returned messages might not reach the upper bound of the prediction model results. Moreover, a node with high fan-out (a node connects with high degree of neighbours) is able to impact on the total number of returned messages. When a node attempts to forward a task to its neighbours, if it routes by choosing a neighbour with high fan-out, an opportunity to receive returned messages tends to be higher than by selecting a peer who has lower fan-out.

Furthermore, the node acting as a terminal who only returns the answer to the source without forwarding that task to others is also a factor impacting on the number of peers responding. Regarding our experiments, the terminal nodes were created, randomly positioned within the area by defaulting at 10% of all nodes. We estimate that there were 3,000 terminal nodes scattered on the network. If a source randomly encounters these nodes, it implies that the lifetime of task propagation has been limited. Hence, the more terminal nodes there are in the network, the less the number of peers responding we receive. Another factor influencing the total number of responses is battery level and pre-specified energy budgets of each node. The received task is allowed to be propagated to others whenever the

node has enough energy by $E_{transfer} \leq \xi$, given the details in Section 3. In our *LogicCrowd* program, the ξ value is set by an energy budget corresponding to a current energy level. The energy budget is a user's policy of energy usage allowed for the programs to distribute tasks. Thus, the higher the percentage of energy budget assigned, the greater the opportunity for forwarding messages, bringing an increase in the number of returned messages from peers. In our experiment, the energy budget was set at 100% so it allows a maximum level of current energy to execute/forward a task.

Regarding the energy consumption analysis, there are five key factors that influence the level of battery consumption. These factors are waiting period, display modes, node types, packet sizes, and degree of nodes. To sum up, the higher the waiting period, brightness, packet sizes and degree of node, the more energy consumed. Moreover, nodes with a foreground display mode spend a considerably higher amount of energy than those with the background mode. Notice that waiting periods and display modes are crucial factors that have been proved to largely affect mobile energy usage, as compared to other factors. To reduce battery usage during task distribution, the user might consider placing the waiting period with short intervals and use the background mode. But users have different needs. For example, one user requires at least 20 returned messages from a crowded area. Hence, to save battery, the user is able to default the waiting period to a maximum of 5 minutes by sending the task with small packet sizes (e.g., 100 KB) based on the assumption that each node is willing to forward the task to others. We also found that acting as a mediator (relaying queries on answers) is not prohibitive, and could work in reality (with an appropriate energy budget set). This means that networks with active peer-to-peer crowdsourcing could be feasible, with many nodes opting to act as mediators while using the network mediators.

7.6 Summary

For the end-user who wants to issue crowdsourcing tasks/queries via mobile peer networks, a question is how long should I wait? It is generally difficult to determine suitable wait times and parameters (i.e., the crowd task conditions) to use in order meet the user's own requirements, such as to obtain adequate numbers of responses while keeping within pre-specified energy budgets. This chapter takes a step towards addressing these issues, showing that particular trends and observations can be made to help the user gauge such crowd task conditions. In

particular, the crowdsourcing method has been examined in order to study issues in peer responses and energy consumption for crowdsourcing using mobile opportunistic networks. We propose models to predict the upper bound on the number of responses in the network and also give an energy consumption model to estimate the energy used by both owners, mediators, and workers. The simulation studied the distribution of crowd tasks of mobile crowdsourcing in random networks with limited communication ranges and explored the factors of mobile crowdsourcing in opportunistic networks that have an impact on crowd task propagation and energy usage for each node.

However, there is still abundant work to be done in order to obtain more accurate and precise estimates for the end-user, given the knowledge of the current mobile peer network surrounding the user, including how to estimate contextual factors that can be useful for predicting the results of queries or responses for tasking, e.g., people densities nearby which can influence what wait times to use if a user wants a certain number of responses, as well as how to properly categorise the range of mobile crowdsourcing tasks according to required energy budgets and responsiveness.

Chapter 8

Conclusion and Future Work

8.1 Summary and Contributions

This thesis has described the concept of a declarative programming paradigm integrated with crowdsourcing behaviors for the mobile environment. The *LogicCrowd* framework has been introduced to fill the gap between traditional machine computation and the power of the crowd. In this section, we summarize the findings in the thesis. This section also presents a consideration of insight and implications for the use of crowdsourcing in the mobile environment.

Chapter 2 provided background knowledge on the crowdsourcing paradigm and key issues for developing its applications.

Chapter 3 presented the declarative programming paradigm integrated with crowdsourcing for the mobile environment in particular. The concept and the relevant key ideas of logic programming have been defined. This is followed by detailing the *LogicCrowd* concept and the simple extensions of Prolog. The simple extensions of Prolog in *LogicCrowd* allow queries to be opened to the crowd in order to solve more complex problems. The crowd predicate is implemented by extending the pure Prolog meta-interpreter with capabilities for working with the crowd through various platforms that include (1) crowdsourcing market platforms, (2) social networks, and (3) P2P networks. In addition, the ability of *LogicCrowd* to provide choice for executing queries of both synchronous and asynchronous calls makes the computation process more flexible when engaging the crowd.

Chapter 4 presented the architecture of the *LogicCrowd* framework, which is a realization of the declarative crowdsourcing concept via mobile technologies. This *LogicCrowd* framework contributes a proof-of-concept for an innovative approach

that integrates logic programming with crowdsourcing middleware in order to provide a declarative programming platform for mobile crowdsourcing applications. In addition, the four core components of the *LogicCrowd* framework, (1) Prolog, (2) Mediator, (3) Knowledge Base and (4) Crowdsourcing APIs, are designed to combine conventional machine computation and the power of the crowd.

Chapter 5 detailed a number of prototype applications built on the *LogicCrowd* framework. These applications are derived and expanded from the motivating scenario in Section 3.1. The applications were designed to utilize and show different uses of the *LogicCrowd* framework. The applications demonstrated and validated the notion of *LogicCrowd*.

Chapter 6 presented the performance evaluation of the *LogicCrowd* prototype. The energy characteristics of the *LogicCrowd* prototype on real mobile devices were investigated in two extensions of Prolog: (1) basic crowd predicate and (2) crowd unification. Findings from these experiments revealed that our proposed approach was feasible on the user's own personal mobile platform for a range of methods used to reach the crowd, and that even a large number of queries could be supported with relatively small impact on the battery level.

Chapter 7 presented the basic scalability experiments. This presentation aimed to examine a task/query propagation strategy. The experiments focused on supporting mobile crowdsourcing in intermittently connected peer-to-peer opportunistic networks. The main contribution of this chapter focuses only on P2P propagation and the collection of tasks/queries and their responses. The key result from these experiments includes an energy analysis to estimate the energy used by both an owner (source node) and workers (mediator or terminal node).

In summary, the key contributions of this thesis are as follows.

- We developed the *LogicCrowd*, which facilitates users/developers to connect to the crowd rather than a closed set of facts in a database so that complex problems can be solved. *LogicCrowd* is built on top of tuProlog, which integrates seamlessly with Java/Android.
- We extended the capabilities of Prolog in the *LogicCrowd* programs. By extending the basic Prolog meta-interpreter, new operators are added as built-in predicates. This enables query evaluations to seamlessly extend to the crowd, going outside the local database. In addition, two flexible methods are provided to execute the rules: synchronous and asynchronous executions.

- We proposed crowd unification as a mechanism for involving humans in solving comparison problems, essentially enabling queries to access the crowd in an on-demand fashion. The purpose of crowd unification is to extend the pattern-matching notion control in logic programming beyond the boundaries of traditional term comparision and a local database through public networks.
- We also extended Prolog for mobile P2P querying. *LogicCrowd* enables queries that exploit connections among peers via mobile communication technologies such as Wi-Fi Direct and Bluetooth. Each peer is able to process and distribute his/her tasks with or without the control of the original peer.
- We demonstrated the system's versatility and usefulness by introducing prototype applications based on our framework using the declarative programming language integrated with crowdsourcing behaviors. The findings showed that our system is feasible and can work effectively in the real world.
- We evaluated the *LogicCrowd* framework by studying the power performance of the *LogicCrowd* prototype. The results from this evaluation showed that *LogicCrowd* programs influence the power consumption that occurs in different conditions. It appeared that the users were able to reduce and manage the energy consumption of their *LogicCrowd* queries by selecting the most appropriate energy-related conditions, such as waiting period, size of transmitted content and crowdsourcing platforms. These findings are a step towards helping users with practical usage of *LogicCrowd* programs and helping developers understand the energy implications of mobile *LogicCrowd*.

In conclusion, this thesis has presented the concept of crowdsourcing behaviors incorporated within the declarative programming paradigm. The use of logic programming is aimed at providing expressive power, declarative semantics, a higher level of abstraction, and allowing the query and manipulation of knowledge and reasoning. The key distinguishing feature of the *LogicCrowd* framework is a design which combines conventional logic-based machine computation with the power of the crowd to solve problems that might be difficult for a machine alone. In our case, the crowdsourcing platforms that the *LogicCrowd* system interfaces with are (1) social networks, (2) existing crowdsourcing market platforms, and (3) mobile P2P networks. Moreover, *LogicCrowd* can operate on the mobile platform and works through the mobility-support systems. It can also be useful in P2P computing for querying and multicasting tasks that are shared over peer networks.

During the process of designing, implementing and developing the framework and its applications the five research questions from Chapter 1 are answered. We can envision a future when crowdsourcing is a normal part of the logic programming paradigm. The systems or applications we build can be more flexible and more powerful by using human intelligence integrated with conventional machine computation.

Overall, mobile crowdsourcing is very challenging. We are building future applications with current knowledge and technologies. There are still many challenges in mobile crowdsourcing, incorporating crowd power with logic programming and building mobile crowdsourcing applications. However, we believe that, with our proposed approach, mobile crowdsourcing applications will provide a higher level of abstraction with specialized language features, and will also allow querying and manipulation of knowledge and reasoning, interleaved with crowd input.

8.2 Future Work

We are at the dawn of the crowdsourcing paradigm and this thesis has obviously not addressed all relevant issues. We will now discuss some challenges and address the limitations of this study. This section concludes by suggesting directions for future research.

8.2.1 Incentive Mechanisms

Adequate user participation is one of the most critical factors that can be used to determine whether crowdsourcing applications achieve good service quality [Yang et al., 2012; Yuen et al., 2011]. Thus, incentive mechanisms are necessary to attract more user participation in a crowdsourcing task. In our prototype, most of users' contributions are based on voluntary participation. Even though our system has interfaced with MTurk, which is a popular crowdsourcing Internet marketplace, it has still neglected some critical properties of incentive mechanisms.

In the future, we will add features or qualities that are expected to motivate users to participate in *LogicCrowd* applications. By extending the Prolog meta-interpreter in the *LogicCrowd* program, we are able to build the incentive mechanism into our system. The requesters could determine the reward offered to the workers, especially monetary incentive structures and they could control

the total payment to the workers that can maximize the benefits for both the requesters and workers. Moreover, we take advantage of the pervasive smartphones such as GPS or WiFi/3G/4G interfaces, to scale up sensitive data collection and analysis in the background to improve participation levels. A *LogicCrowd* program can be analysed for its minimal cost to achieve a particular level of crowd support for its results.

8.2.2 Improving the Task Propagation Strategy

We proposed the task propagation strategy in Chapter 7. In this current thesis, we focused on the propagation and collection of tasks over M-P2P networks. We have defined the time-to-live (τ) and power-to-live (ξ) values to limit the lifetime of tasks so that the action of forwarding tasks to other peers can be stopped. To stop distributing tasks over the M-P2P network, the task propagation algorithm that we proposed emphasizes only two cases: using the expiry of waiting time (τ) and the amount of remaining battery level (ξ).

However, there are not only two reasons in which the requester might want to stop the forwarding tasks; for example, first, the requester may be satisfied with the total number of responses/answers returned before expiry, and second, the requester needs to immediately cancel forwarding the task. In the case of decentralized-control P2P model, our system does not support this type of request yet. The model in relation to the task propagation, can be further developed to make this possible. Results from this further development can enhance its ability for task distribution and optimize the total energy it consumes.

In addition to this, to improve the quality of results, particularly in M-P2P networks, the strategy of task propagation can be improved. This can be done by improving its potential in routing to the right group of people, using the task history of the requester. Historic data can be used to record the paths that are traversed or people who are trusted and often responding with quality contributions are reordered for future reference. Additionally, the use of this data in combination with the location-based features of smartphones can allow the accuracy of tasks to be distributed to the right places and occasions.

8.2.3 Integrating with other Frameworks

The current prototype implementation starts with Prolog due to its relatively simple semantics and popularity. However, our *LogicCrowd* concept can be applied to

other declarative programming languages; e.g., Answer Set Programming (ASP). For future study, we will employ ASP to develop crowdsourcing applications; in particular, we will use ASP for building systems for crowd judgment and reasoning. In addition, our prototype is built on top of tuProlog, which integrates seamlessly into the Android platform. The next version of our prototype will be available for extending either Web-based applications or iOS platforms in order to enable crowdsourcing applications for a larger contributing crowd and make contributing easier and ubiquitous.

Appendix A

The *LogicCrowd* Meta-Interpreter

```
:- op(35,fx,'*=').
:- op(35,xfy,'*=').
:- op(65,fx,'*').
:- op(70,xfy,'?').
:- op(72,xfy,'*').
:- op(75,xfy,'#').

%% as a pure Prolog extension
solve(true):-!.
solve(not(A)):- !,\+solve(A).
solve((A)):- builtin(A),!, A.
solve((A, Body)) :- !,solve(A), solve(Body).
solve((A)):- clause(A,Body), solve(Body).

%% LogicCrowd extension
solve(Askcrowd?Result#Condition):- !,
        solvecond(Condition),
        (asyn,!,asynproc(Askcrowd,Result);
                  synproc(Askcrowd,Result)).

%% crowd unification extension
solve((*=(A,B))):-!,
        solve(A), solve(B), crowdUni(A,B,all).
solve((A*=B)):- !,
        solve(A), solve(B), crowdUni(A,B,all).
solve((*=(A,B)@ST)):- !,
        solve(A), solve(B), crowdUni(A,B,ST).
solve((A*=B)@ST):- !,
        solve(A), solve(B), crowdUni(A,B,ST).
solve((A*=B)@ST*Peer#Time):- !,
        solve(A), solve(B),
        crowdUni(A,B,ST,Peer,Time).

```

```
crowdUni(Term1,Term2,ST):-
        setLink(Term1,Term2,Path),
        setCond(Path,ST),
        asynproc(unify,Result).
crowdUni(Term1,Term2,ST,Peer,Time):-
        setLink(Term1,Term2,Path),
        setCond(Path,ST,Peer,Time),
        asynproc(unify,Result).

setLink(Term1,Term2,Path):-
        Term1 =.. [Head1|Path1],
        Term2 =.. [Head2|Path2],
        length(Path1, Length1),
        length(Path2, Length2),
        (Length1 = 0,!,append(Path1,Head1,Link1);
                        Path1 =.. [_H1|[Link1,T1]]),
        (Length2 = 0,!,append(Path2,Head2,Link2);
                        Path2 =.. [_H1|[Link2,T2]]),
        Path = [Link1,Link2].

setCond(Path,ST):-
        asserta(asktype('choice')),
        asserta(question('Are they the same?')),
        asserta(options(['Yes','No'])),
        asserta(picture(Path)),
        asserta(askto([bluetooth,mturk,facebook])),
        asserta(expiry('0,20,0')),
        asserta(supporttype(ST)).

setCond(Path,ST,Peer,Time):-
        asserta(asktype('choice')),
        asserta(question('Are they the same?')),
        asserta(options(['Yes','No'])),
        asserta(picture(Path)),
        asserta(expiry(Time)),
        (Peer = 'mix',!,asserta(askto([bluetooth,facebook,mturk]));
                        mturk(Peer)),
        asserta(supporttype(ST)).

mturk(Peer):-
        (Peer = 'mturk',!,asserta(askto([mturk]));facebook(Peer)).
facebook(Peer):-
        (Peer = 'facebook',!,asserta(askto([facebook]));bluetooth(Peer)).
bluetooth(Peer):-
        (Peer = 'bluetooth',!,asserta(askto([bluetooth]))).

%% P2P processing
solve(*Askcrowd?Result#Condition):- !,
        solvecond(Condition),asserta(forward(true)),
        (asyn,!,asynproc(Askcrowd,Result); synproc(Askcrowd,Result)).

%% Centralized-control P2P
solve(PeerID*Askcrowd?Result#Condition):- !,
```

```
87          solvecond(Condition),
88          solvepeer(PeerID),
89          (asyn,!,asynproc(Askcrowd,Result); synproc(Askcrowd,Result)),
90
91  solvepeer(PeerID):-
92          friend(PeerID), asserta(askto(PeerID)).
93
94  synproc(Askcrowd,Result):-
95          checkcond(TypeQuestion,Question,Nammemsg,Link,Description,Picture,
96                  Options,Askto,Group,Locatedin,Expiry,EndTime,WPTime,Forward,
97                  ForwardQID,ST,AtLeast),
98          askcrowd(syn,Askcrowd,TypeQuestion,Question,Options,Nammemsg,Link,
99                  Description,Picture,Askto,Group,Locatedin,Expiry,EndTime,
100                 WPTime,Forward,ForwardQID,ST,AtLeast,QuestionID).
101         registercallbacksyn(QuestionID,Question,Askto,TypeQuestion,Expiry,
102                 Forward,Result).
103
104 asynproc(Askcrowd,Result):-
105         checkcond(TypeQuestion,Question,Nammemsg,Link,Description,Picture,
106                 Options,Askto,Group,Locatedin,Expiry,EndTime,WPTime,Forward,
107                 ForwardQID,ST,AtLeast),
108         askcrowd(asyn,Askcrowd,TypeQuestion,Question,Options,Nammemsg,Link,
109                 Description,Picture,Askto,Group,Locatedin,Expiry,EndTime,
110                 WPTime,Forward,ForwardQID,ST,AtLeast,QuestionID).
111         registercallbackasyn(Askcrowd,QuestionID,Question,Askto,TypeQuestion,
112                 Expiry,Forward,ST,AtLeast,Result).
113
114 solvecond([]):- !.
115 solvecond(Condition):-
116         Condition=..[_H|[Head,Body]],
117         asserta(Head),
118         solvecond(Body).
119
120 checkcond(TypeQuestion,Question,Askto,Link,Description,Picture,Options,
121         Nammemsg,Group,Locatedin,Expiry,EndTime,WPTime,Forward,ForwardQID,
122         ST,AtLeast):-
123                 (asktype(A),!, TypeQuestion = A; set(TypeQuestion)),
124                 (question(B),!, Question = B; set(Question)),
125                 (askto(C),!,Askto = C; set(Askto)),
126                 (link(D),!, Link = D; set(Link)),
127                 (description(E),!,Description= E; set(Description)),
128                 (picture(F),!, Picture = F; set(Picture)),
129                 (options(G),!, Options =G; set(Options)),
130                 (nammemsg(H),!, Nammemsg = H; set(Nammemsg)),
131                 (group(I),!,Group = I; set(Group)),
132                 (locatedin(J),!,Locatedin = J; set(Locatedin)),
133                 (expiry(K),!,Expiry = K; set(Expiry)),
134                 (endtime(L),!,EndTime = L; set(EndTime)),
135                 (wptime(M),!,WPTime = M; set(WPTime)),
136                 (forward(N),!,Forward = N; set(Forward)),
137                 (forwardqid(O),!,ForwardQID = O; set(ForwardQID)).
138
139 set(X):- X = 'null'.
```

```
140
141 setsupport(AtLeast,ST):-
142         set(AtLeast),(supporttype(P),!,ST = P; set(ST)).
```

Appendix B

Simulation of Mobile Crowdsourcing

This appendix presents the use of the LogicCrowd simulation model implemented in NetLogo. The model can be downloaded at https://github.com/jurairat/LogicCrowd-Netlogo

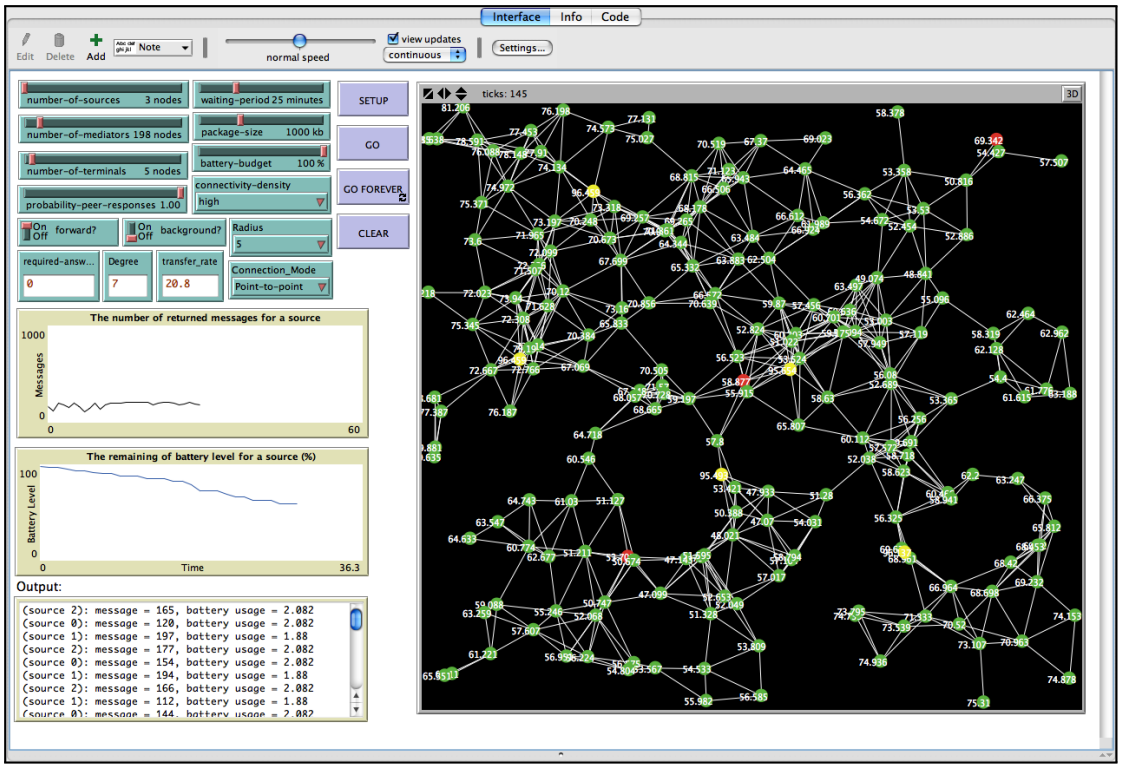


Figure B.1: The mobile crowdsourcing simulator interface

B.1 What is it?

In our work, the mobile crowdsourcing simulator has been developed in NetLogo which is a multi-agent modeling language that provides a programmable environment for simulating natural and social phenomena. This model demonstrates the crowd task propagation through a mobile opportunistic network.

B.2 How it works?

Each node represents a mobile device connected with nearby neighbors in communication distances given by the RADIUS chooser. Then the nodes are linked together in the large network. Each time step (tick), a source node (colored red) attempts to distribute the task to neighbors within its connection range. The neighbors that might be the mediator (colored green) or the terminal (colored yellow) will respond with a probability given by the PROBABILITY-PEER-RESPONSES slider. The probability represents the probability of a peer responding to the task for the whole network.

Before the task is spread through the network, the crowd's conditions are given by the WAITING-PERIOD slider, ENERGY-BUDGET slider, DATA-SIZE slider and CONNECTION-MODE chooser. Moreover, the FORWARD and BACKGROUND switchers have been given. The FORWARD switcher is an option for turning on or off in order to forward the task to other nodes whereas the BACKGROUND switcher is to switch the display mode (running programs in background or foreground) in the nodes (i.e., the mobile device) for estimating the energy during propagation.

During propagation of the message, a mediator node is able to send the message to the source and also forward queries or tasks to its neighbors whereas a terminal node is able to only return an answer and cannot forward that task to others. The total number of returned messages to a source and the battery level of a source has been plotted and displayed in the graphs in Figure B.1.

B.3 How to use it?

Using the sliders, choose the NUMBER-OF-SOURCES, NUMBER-OF-MEDIATORS and NUMBER-OF-TERMINALS. The network is created based on the

peer-to-peer network communication distance given by RADIUS. A node is randomly chosen and connected to the nearest node within the setting distance. This process is repeated until all nodes are connected and the links of the nodes is based on the CONNECTIVITY-DENSITY chooser.

Then press SETUP to create the network. Press GO to run the model. The model will stop running once all sources have completely propagated the tasks and received the returned messages from neighbor nodes. Press GO FOREVER to keep running the model over and over again, until the expiry or not enough battery level of each device to propagate task or the user can press the button again to stop it.

The PROBABILITY-PEER-RESPONSES, WAITING-PERIOD, DATA-SIZE, ENERGY-BUDGET, CONNECTION-MODE, FORWARD, and BACK-GROUND (discussed in "How it Works" above) can be adjusted before pressing GO, or while the model is running. The RETURNED MESSAGES plot shows the total number of peer responses to a source in each round. The ENERGY CONSUMPTION plot shows the battery level of a source when propagating the task in each round.

Bibliography

Afridi, A. H. (2011). Crowdsourcing in mobile: A three stage context based process. In *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, DASC '11, pages 242–245, Washington, DC, USA. IEEE Computer Society.

Ahmad, S., Battle, A., Malkani, Z., and Kamvar, S. (2011). The jabberwocky programming environment for structured social computing. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 53–64, New York, NY, USA. ACM.

Ali, K., Al-Yaseen, D., Ejaz, A., Javed, T., and Hassanein, H. (2012). CrowdITS: Crowdsourcing in intelligent transportation systems. In *Proceedings of Wireless Communications and Networking Conference (WCNC), 2012 IEEE*, pages 3307–3311, Paris, France.

Alt, F., Shirazi, A. S., Schmidt, A., Kramer, U., and Nawaz, Z. (2010). Location-based crowdsourcing: Extending crowdsourcing to the real world. In *Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries*, NordiCHI '10, pages 13–22, New York, NY, USA. ACM.

Amsterdamer, Y., Davidson, S. B., Kukliansky, A., Milo, T., Novgorodov, S., and Somech, A. (2015). Managing general and individual knowledge in crowd mining applications. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research,Online Proceedings*, Asilomar, California, USA.

Aoki, H. and Morishima, A. (2013). A divide-and-conquer approach for crowdsourced data enumeration. In Jatowt, A., Lim, E.-P., Ding, Y., Miura, A., Tezuka, T., Dias, G., Tanaka, K., Flanagin, A., and Dai, B., editors, *Social Informatics*, volume 8238 of *Lecture Notes in Computer Science*, pages 60–74. Springer International Publishing.

Balasubramanian, N., Balasubramanian, A., and Venkataramani, A. (2009). Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 280–293, New York, NY, USA. ACM.

Barowy, D. W., Curtsinger, C., Berger, E. D., and McGregor, A. (2012). AutoMan: A platform for integrating human-based and digital computation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 639–654, New York, NY, USA. ACM.

Barroso, L. and Holzle, U. (2007). The case for energy-proportional computing. *Computer*, 40(12):33–37.

Berg, M. d., Cheong, O., Kreveld, M. v., and Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition.

Bernstein, M. S., Little, G., Miller, R. C., Hartmann, B., Ackerman, M. S., Karger, D. R., Crowell, D., and Panovich, K. (2010). Soylent: A word processor with a crowd inside. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 313–322, New York, NY, USA. ACM.

Bornholt, J., Mytkowicz, T., and McKinley, K. S. (2012). The model is not enough: Understanding energy consumption in mobile devices. *Power (watts)*, 1(2):3.

Brabham, D. C. (2008). Crowdsourcing as a model for problem solving an introduction and cases. *Convergence: the international journal of research into new media technologies*, 14(1):75–90.

Brabham, D. C. (2012). Crowdsourcing: A model for leveraging online communities. In *The Participatory Cultures Handbook*, pages 120–129. Routledge.

Brabham, D. C. (2013). *Crowdsourcing*. The MIT Press essential knowledge series. The MIT Press, Cambridge.

Chaintreau, A., Fraigniaud, P., and Lebhar, E. (2008). Opportunistic spatial gossip over mobile social networks. In *Proceedings of the First Workshop on Online Social Networks*, WOSN '08, pages 73–78, New York, NY, USA. ACM.

Chatzimilioudis, G., Konstantinidis, A., Laoudias, C., and Zeinalipour-Yazti, D. (2012). Crowdsourcing with smartphones. *Internet Computing, IEEE*, 16(5):36–44.

Chen, G. and Kotz, D. (2000). A survey of context-aware mobile computing research. Technical report, Hanover, NH, USA.

Chen, X., Santos-Neto, E., and Ripeanu, M. (2012). Crowd-based smart parking: A case study for mobile crowdsourcing. In *Mobile Wireless Middleware, Operating Systems, and Applications - 5th International Conference, Mobilware 2012, Berlin, Germany*, pages 16–30.

Chen, Y., Liem, B., and Zhang, H. (2011). An iterative dual pathway structure for speech-to-text transcription. In *Proceedings of the 3rd Human Computation Workshop (HCOMP 2011)*, San Francisco, CA.

Chilton, L. B., Little, G., Edge, D., Weld, D. S., and Landay, J. A. (2013). Cascade: Crowdsourcing taxonomy creation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1999–2008, New York, NY, USA. ACM.

Chinrungrueng, J., Sunantachaikul, U., and Triamlumlerd, S. (2007). Smart parking: An application of optical wireless sensor network. In *Proceedings of the 2007 International Symposium on Applications and the Internet Workshops*, SAINT-W '07, pages 66–66, Washington, DC, USA. IEEE Computer Society.

Chon, Y., Lane, N. D., Li, F., Cha, H., and Zhao, F. (2012). Automatically characterizing places with opportunistic crowdsensing using smartphones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 481–490, New York, NY, USA. ACM.

Constantinides, M., Constantinou, G., Panteli, A., Phokas, T., Chatzimilioudis, G., and Zeinalipour-Yazti, D. (2012). Proximity interactions with crowdcast. In *The 11th Hellenic Data Management Symposium*, HDMS'12, Chania, Greece.

Cornelius, C., Kapadia, A., Kotz, D., Peebles, D., Shin, M., and Triandopoulos, N. (2008). Anonysense: Privacy-aware people-centric sensing. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 211–224, New York, NY, USA. ACM.

Dai, P., Daniel, M., and Weld, S. (2010). Decision-theoretic control of crowd-sourced workflows. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*, pages 1168–1174, Atlanta, Georgia, USA.

Dai, P., Mausam, and Weld, D. S. (2011). Artificial intelligence for artificial artificial intelligence. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence, AAAI'11*, pages 1153–1159, San Francisco, CA, USA.

Das, T., Mohan, P., Padmanabhan, V. N., Ramjee, R., and Sharma, A. (2010). PRISM: Platform for remote sensing using smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 63–76, New York, NY, USA. ACM.

Dawid, P., Skene, A. M., Dawidt, A. P., and Skene, A. M. (1979). Maximum likelihood estimation of observer error-rates using the em algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):20–28.

Difallah, D. E., Demartini, G., and Cudré-Mauroux, P. (2013). Pick-A-Crowd: Tell me what you like, and i'll tell you what to do. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 367–374, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee.

Dinh, H. T., Lee, C., Niyato, D., and Wang, P. (2013). A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing*, 13(18):1587–1611.

Doan, A., Ramakrishnan, R., and Halevy, A. Y. (2011). Crowdsourcing systems on the world-wide web. *Commun. ACM*, 54(4):86–96.

Dow, S., Kulkarni, A., Klemmer, S., and Hartmann, B. (2012). Shepherding the crowd yields better work. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1013–1022, New York, NY, USA. ACM.

Duckham, M. (2013). *Decentralized Spatial Computing - Foundations of Geosensor Networks*. Springer, Berlin.

Eagle, N. (2009). Txteagle: Mobile crowdsourcing. In *Proceedings of the 3rd International Conference on Internationalization, Design and Global Development:*

Held As Part of HCI International 2009, IDGD '09, pages 447–456, Berlin, Heidelberg. Springer-Verlag.

Ell, B., Vr, D., and Simperl, E. (2015). Spartiqulation: Verbalizing sparql queries. In Simperl, E., Norton, B., Mladenic, D., Della Valle, E., Fundulaki, I., Passant, A., and Troncy, R., editors, *The Semantic Web: ESWC 2012 Satellite Events*, volume 7540 of *Lecture Notes in Computer Science*, pages 117–131. Springer Berlin Heidelberg.

Estellés-Arolas, E. and González-Ladrón-De-Guevara, F. (2012). Towards an integrated crowdsourcing definition. *Journal of Information Science*, 38(2):189–200.

Estrin, D. (2010). Participatory sensing: applications and architecture [internet predictions]. *Internet Computing, IEEE*, 14(1):12–42.

Fall, K. (2003). A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '03, pages 27–34, New York, NY, USA. ACM.

Foncubierta Rodríguez, A. and Müller, H. (2012). Ground Truth Generation in Medical Imaging: A crowdsourcing-based iterative approach. In *Proceedings of the ACM Multimedia 2012 Workshop on Crowdsourcing for Multimedia*, CrowdMM '12, pages 9–14, New York, NY, USA. ACM.

Franklin, M. J., Kossmann, D., Kraska, T., Ramesh, S., and Xin, R. (2011). CrowdDB: answering queries with crowdsourcing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 61–72, New York, NY, USA. ACM.

Fuchs-Kittowski, F. and Faust, D. (2014). Architecture of mobile crowdsourcing systems. In Baloian, N., Burstein, F., Ogata, H., Santoro, F., and Zurita, G., editors, *Collaboration and Technology*, volume 8658 of *Lecture Notes in Computer Science*, pages 121–136. Springer International Publishing.

Ganti, R., Ye, F., and Lei, H. (2011). Mobile crowdsensing: current state and future challenges. *Communications Magazine, IEEE*, 49(11):32–39.

Geiger, D., Seedorf, S., Schulze, T., Nickerson, R., and Schader, M. (2011). Managing the crowd: Towards a taxonomy of crowdsourcing processes. In *Proceedings*

of the 17th Americas Conference on Information Systems, Detroit, Michigan, USA.

Gonnokami, K., Morishima, A., and Kitagawa, H. (2013). Condition-task-store: A declarative abstraction for microtask-based complex crowd-sourcing. In Cheng, R., Sarma, A. D., Maniu, S., and Senellart, P., editors, *Proceedings of the First VLDB Workshop on Databases and Crowdsourcing, DBCrowd 2013*, volume 1025 of *CEUR Workshop Proceedings*, pages 20–25, Riva del Garda, Trento, Italy. CEUR-WS.org.

Goodchild, M. F. and Glennon, J. A. (2010). Crowdsourcing geographic information for disaster response: a research frontier. *International Journal of Digital Earth*, 3(3):231–241.

Gupta, A. and Mohapatra, P. (2007). Energy consumption and conservation in wifi based phones: A measurement-based study. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON '07. 4th Annual IEEE Communications Society Conference on*, pages 122–131, San Diego, California, USA.

Hetmank, L. (2013). Components and functions of crowdsourcing systems–a systematic literature review. In *Proceedings of the 11th International Conference on Wirtschaftsinformatik*, pages 55–69, University Leipzig, Germany.

Hetmank, L. (2014). A synopsis of enterprise crowdsourcing literature. Twenty Second European Conference on Information Systems, Tel Aviv, available at http://ecis2014.eu/E-poster/files/0771-file1.pdf, retrieved on February 2015.

Heymann, P. and Garcia-Molina, H. (2011). Turkalytics: Analytics for human computation. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 477–486, New York, NY, USA. ACM.

Hirth, M., Hoßfeld, T., and Tran-Gia, P. (2013). Analyzing costs and accuracy of validation mechanisms for crowdsourcing platforms. *Mathematical and Computer Modelling*, 57(11–12):2918 – 2932. Information System Security and Performance Modeling and Simulation for Future Mobile Networks.

Ho, C.-J., Jabbari, S., and Vaughan, J. W. (2013). Adaptive task assignment for crowdsourced classification. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 534–542, Atlanta, USA.

Horvitz, E. J. (1987). Problem-solving design: Reasoning about computational value, tradeoffs, and resources. In *Proceedings of the Second Annual NASA Research Forum*, pages 26–43, NASA Ames Research Center, Palo Alto, California,USA.

Howe, J. (2006). The rise of crowdsourcing. *Wired magazine*, 14(6):1–4.

Hu, X., Chu, T., Chan, H., and Leung, V. (2013). Vita: A crowdsensing-oriented mobile cyber-physical system. *Emerging Topics in Computing, IEEE Transactions on*, 1(1):148–165.

Hupfer, S., Muller, M., Levy, S., Gruen, D., Sempere, A., Ross, S., and Priedhorsky, R. (2012). MoCoMapps: Mobile collaborative map-based applications. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work Companion*, CSCW '12, pages 43–44, New York, NY, USA. ACM.

Ipeirotis, P. G., Provost, F., and Wang, J. (2010). Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 64–67, New York, NY, USA. ACM.

Jin, R. and Ghahramani, Z. (2003). Learning with multiple labels. In *Advances in neural information processing systems*, pages 921–928. MIT Press, Cambridge, MA, USA.

Kapadia, A., Triandopoulos, N., Cornelius, C., Peebles, D., and Kotz, D. (2008). Anonysense: Opportunistic and privacy-preserving context collection. In *Proceedings of the 6th International Conference on Pervasive Computing*, Pervasive '08, pages 280–297, Berlin, Heidelberg. Springer-Verlag.

Karampinas, D. and Triantafillou, P. (2012). Crowdsourcing taxonomies. In *Proceedings of the 9th International Conference on The Semantic Web: Research and Applications*, ESWC'12, pages 545–559, Berlin, Heidelberg. Springer-Verlag.

Karger, D. R., Oh, S., and Shah, D. (2011). Iterative learning for reliable crowdsourcing systems. In *Advances in Neural Information Processing Systems 24*, pages 1953–1961. Curran Associates, Inc.

Karger, D. R., Oh, S., and Shah, D. (2014). Budget-optimal task allocation for reliable crowdsourcing systems. *Operations Research*, 62(1):1–24.

Kaufmann, N., Schulze, T., and Veit, D. (2011). More than fun and money. worker motivation in crowdsourcing-a study on mechanical turk. In *Proceedings of the 17th Americas Conference on Information Systems*, pages 1–11, Detroit, Michigan, USA.

Kazai, G. (2011). In search of quality in crowdsourcing for search engine evaluation. In Clough, P., Foley, C., Gurrin, C., Jones, G., Kraaij, W., Lee, H., and Mudoch, V., editors, *Advances in Information Retrieval*, volume 6611 of *Lecture Notes in Computer Science*, pages 165–176. Springer Berlin Heidelberg.

Khattak, F. K. and Salleb-Aouissi, A. (2011). Quality control of crowd labeling through expert evaluation. In *Proceedings of the NIPS 2nd Workshop on Computational Social Science and the Wisdom of Crowds (NIPS 2011)*, pages 1–5, Sierra Nevada, Spain.

Khorashadi, B., Das, S. M., and Gupta, R. (2013). Flexible architecture for location based crowdsourcing of contextual data. QUALCOMM Incorporated, Silicon Valley Patent Group, June 25: US08472980.

Kittur, A., Nickerson, J. V., Bernstein, M., Gerber, E., Shaw, A., Zimmerman, J., Lease, M., and Horton, J. (2013). The future of crowd work. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*, CSCW '13, pages 1301–1318, New York, NY, USA. ACM.

Kittur, A., Smus, B., Khamkar, S., and Kraut, R. E. (2011). CrowdForge: Crowdsourcing complex work. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 43–52, New York, NY, USA. ACM.

Konstantinidis, A., Zeinalipour-Yazti, D., Andreou, P., and Samaras, G. (2011). Multi-objective query optimization in smartphone social networks. In *Proceedings of the 2011 IEEE 12th International Conference on Mobile Data Management - Volume 01*, MDM '11, pages 27–32, Washington, DC, USA. IEEE Computer Society.

Kulkarni, A., Can, M., and Hartmann, B. (2012). Collaboratively crowdsourcing workflows with turkomatic. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1003–1012, New York, NY, USA. ACM.

Kuncheva, L., Whitaker, C., Shipp, C., and Duin, R. (2003). Limits on the majority vote accuracy in classifier fusion. *Pattern Analysis and Applications*, 6(1):22–31.

Lane, N., Miluzzo, E., Lu, H., Peebles, D., Choudhury, T., and Campbell, A. (2010). A survey of mobile phone sensing. *Communications Magazine, IEEE*, 48(9):140–150.

Lane, N. D., Chon, Y., Zhou, L., Zhang, Y., Li, F., Kim, D., Ding, G., Zhao, F., and Cha, H. (2013). Piggyback crowdsensing (pcs): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 7:1–7:14, New York, NY, USA. ACM.

Law, E. and Ahn, L. v. (2011). Human computation. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 5(3):1–121.

Lease, M. and Alonso, O. (2014). Crowdsourcing and human computation, introduction. *Encyclopedia of Social Network Analysis and Mining (ESNAM)*, pages 304–315.

Lee, K., Caverlee, J., and Webb, S. (2010). The social honeypot project: Protecting online communities from spammers. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1139–1140, New York, NY, USA. ACM.

Lin, C. H., Daniel, M., and Weld, S. (2012). Dynamically switching between synergistic workflows for crowdsourcing. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence, AAAI'12*, pages 132–133, Toronto, Ontario, Canada.

Little, G., Chilton, L. B., Goldman, M., and Miller, R. C. (2009). TurKit: tools for iterative tasks on mechanical turk. In *Proceedings of the ACM SIGKDD workshop on human computation*, pages 29–30, New York, NY, USA. ACM.

Little, G., Chilton, L. B., Goldman, M., and Miller, R. C. (2010a). Exploring iterative and parallel human computation processes. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 68–76, New York, NY, USA. ACM.

Little, G., Chilton, L. B., Goldman, M., and Miller, R. C. (2010b). TurKit: Human computation algorithms on mechanical turk. In *Proceedings of the 23Nd Annual*

ACM Symposium on User Interface Software and Technology, UIST '10, pages 57–66, New York, NY, USA. ACM.

Little, G. D. G. (2011). *Programming with human computation.* PhD thesis, Massachusetts Institute of Technology.

Liu, B., Jiang, Y., Sha, F., and Govindan, R. (2012a). Cloud-enabled privacy-preserving collaborative learning for mobile sensing. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, SenSys '12, pages 57–70, New York, NY, USA. ACM.

Liu, Q., Ihler, A. T., and Steyvers, M. (2013). Scoring workers in crowdsourcing: How many control questions are enough? In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 26*, pages 1914–1922. Curran Associates, Inc.

Liu, X., Lu, M., Ooi, B. C., Shen, Y., Wu, S., and Zhang, M. (2012b). CDAS: A crowdsourcing data analytics system. *Proc. VLDB Endow.*, 5(10):1040–1051.

Liu, Y., Vili Lehdonvirta, T., and Nakajima, T. (2011). Engaging social medias: Case mobile crowdsourcing. *SoME'11*, pages 1–4.

Lloyd, J. W. (1984). *Foundations of Logic Programming.* Springer-Verlag New York, Inc., New York, NY, USA.

Lloyd, J. W. (1994). Practical advantages of declarative programming. *Joint Conference on Declarative Programming, GULP-PRODE*, 94:94.

Loke, S. W. (2006). Declarative programming of integrated peer-to-peer and web based systems: the case of prolog. *Journal of Systems and Software*, 79(4):523 – 536.

Lü, H. and Fogarty, J. (2012). Crowd-Logic: Implementing and optimizing human computation algorithms using logic programming. Technical report.

Lu, H., Pan, W., Lane, N. D., Choudhury, T., and Campbell, A. T. (2009a). Soundsense: Scalable sound sensing for people-centric applications on mobile phones. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 165–178, New York, NY, USA. ACM.

Lu, R., Lin, X., Zhu, H., and Shen, X. (2009b). Spark: A new vanet-based smart parking scheme for large parking lots. In *Proceedings of INFOCOM 2009, IEEE*, pages 1413–1421, Rio de Janeiro, Brazil.

Luz, N., Silva, N., and Novais, P. (2014). A survey of task-oriented crowdsourcing. *Artificial Intelligence Review*, pages 1–27.

Malone, T. W., Laubacher, R., and Dellarocas, C. (2010). The collective intelligence genome. *IEEE Engineering Management Review*, 38(3):38.

Malone, T. W., Laubacher, R., and Dellarocas, C. (2011). Harnessing Crowds: Mapping the genome of collective intelligence. *MIT Sloan Research Paper*, (4732-09).

Marcus, A., Wu, E., Karger, D. R., Madden, S., and Miller, R. C. (2011). Demonstration of qurk: a query processor for humanoperators. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1315–1318, Athens, Greece. ACM.

Marjanovic, S., Fry, C., and Chataway, J. (2012). Crowdsourcing based business models: In search of evidence for innovation 2.0. *Science and Public Policy*, 39(3):318–332.

Matera, T., Jakes, J., Cheng, M., and Belongie, S. (2014). A user friendly crowdsourcing task manager. In *Workshop on Computer Vision and Human Computation*, Columbus, OH.

Mathur, S., Jin, T., Kasturirangan, N., Chandrasekaran, J., Xue, W., Gruteser, M., and Trappe, W. (2010). Parknet: Drive-by sensing of road-side parking statistics. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 123–136, New York, NY, USA. ACM.

Matsui, T., Baba, Y., Kamishima, T., and Kashima, H. (2013). Crowdsourcing quality control for item ordering tasks. In *Proceedings of The First AAAI Conference on Human Computation and Crowdsourcing*, Palm Springs, California,USA.

Matthew C. Wagner, S. V. E. and Morozova, Y. (2013). Application programming interfaces (APIs) a primer and discussion of oracle america v. google. Technical report, the American Intellectual Property Law Association.

McLachlan, G. and Krishnan, T. (2008). *The EM algorithm and extensions*. Wiley series in probability and statistics. Wiley, Hoboken, NJ, 2. ed edition.

Miettinen, A. P. and Nurminen, J. K. (2010). Energy efficiency of mobile clients in cloud computing. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 4–4, Berkeley, CA, USA. USENIX Association.

Miluzzo, E., Lane, N. D., Fodor, K., Peterson, R., Lu, H., Musolesi, M., Eisenman, S. B., Zheng, X., and Campbell, A. T. (2008). Sensing meets mobile social networks: The design, implementation and evaluation of the cenceme application. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 337–350, New York, NY, USA. ACM.

Minder, P. and Bernstein, A. (2011). CrowdLang first steps towards programmable human computers for general computation. In *Proceedings of the 3rd Human Computation Workshop*, AAAI Workshops, pages 103–108, San Francisco, California, USA. AAAI Press.

Misra, A., Gooze, A., Watkins, K., Asad, M., and Le Dantec, C. A. (2014). Crowdsourcing and its application to transportation data collection and management. *Transportation Research Record: Journal of the Transportation Research Board*, 2414(1):1–8.

Moreira, W. and Mendes, P. (2011). Survey on opportunistic routing for delay/disruption tolerant networks. Technical Report SITI-TR-11-02, SITI, University Lusófona.

Morishima, A., Shinagawa, N., Mitsuishi, T., Aoki, H., and Fukusumi, S. (2012). CyLog/Crowd4U: A declarative platform for complex data-centric crowdsourcing. *Proc. VLDB Endow.*, 5(12):1918–1921.

Mottola, A. (2005). *Design and implementation of a declarative programming language in a reactive environment*. PhD thesis, Università degli Studi di Roma.

Musumba, G. W. and Nyongesa, H. O. (2013). Context awareness in mobile computing: A review. *International Journal of Machine Learning and Applications*, 2(1).

Nandugudi, A., Ki, T., Nuessle, C., and Challen, G. (2014). Pocketparker: Pocketsourcing parking lot availability. In *Proceedings of the 2014 ACM International*

Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '14, pages 963–973, New York, NY, USA. ACM.

Narula, P., Gutheim, P., Rolnitzky, D., Kulkarni, A., and Hartmann, B. (2011). Mobileworks: A mobile crowdsourcing platform for workers at the bottom of the pyramid. *Human Computation*, 11:11.

Negri, M., Bentivogli, L., Mehdad, Y., Giampiccolo, D., and Marchetti, A. (2011). Divide and Conquer: Crowdsourcing the creation of cross-lingual textual entailment corpora. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, pages 670–679, Stroudsburg, PA, USA. Association for Computational Linguistics.

Noronha, J., Hysen, E., Zhang, H., and Gajos, K. Z. (2011). PlateMate: Crowdsourcing nutrition analysis from food photographs. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 1–12, New York, NY, USA. ACM.

Oomen, J. and Aroyo, L. (2011). Crowdsourcing in the cultural heritage domain: Opportunities and challenges. In *Proceedings of the 5th International Conference on Communities and Technologies*, pages 138–149, New York, NY, USA. ACM.

O'Rourke, J. (1998). *Computational Geometry in C*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press.

Parameswaran, A. G., Park, H., Garcia-Molina, H., Polyzotis, N., and Widom, J. (2012). Deco: declarative crowdsourcing. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1203–1212, Maui, HI, USA. ACM.

Parameswaran, A. G. and Polyzotis, N. (2011). Answering Queries using Humans, Algorithms and Databases. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research,Online Proceedings*, pages 160–166, Asilomar, CA, USA.

Parameswaran, M. and Whinston, A. B. (2007). Social computing: An overview. *Communications of the Association for Information Systems*, 19(1):37.

Pedersen, J., Kocsis, D., Tripathi, A., Tarrell, A., Weerakoon, A., Tahmasbi, N., Xiong, J., Deng, W., Oh, O., and De Vreede, G.-J. (2013). Conceptual foundations of crowdsourcing: A review of IS research. In *Proceedings of the*

46th Annual Hawaii International Conference on System Sciences, pages 579–588, Maui, HI,USA. IEEE.

Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2014). Context aware computing for the internet of things: A survey. *Communications Surveys Tutorials, IEEE*, 16(1):414–454.

Petuchowski, E. and Lease, M. (2014). TurKPF: Turkontrol as a particle filter. *CoRR*, abs/1404.5078.

Phuttharak, J. and Loke, S. (2013). Logiccrowd: A declarative programming platform for mobile crowdsourcing. In *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM 2013*, pages 1323–1330, Melbourne, Victoria, Australia.

Phuttharak, J. and Loke, S. (2014a). Declarative programming for mobile crowdsourcing: Energy considerations and applications. In Stojmenovic, I., Cheng, Z., and Guo, S., editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, volume 131 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 237–249. Springer International Publishing.

Phuttharak, J. and Loke, S. W. (2014b). Towards declarative programming for mobile crowdsourcing: P2p aspects. In *Proceedings of the 2014 IEEE 15th International Conference on Mobile Data Management - Volume 02*, MDM '14, pages 61–66, Washington, DC, USA. IEEE Computer Society.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarbus University.

Ponciano, L., Brasileiro, F., Andrade, N., and Sampaio, L. (2014). Considering human aspects on strategies for designing and managing distributed human computation. *Journal of Internet Services and Applications*, 5(1).

Quinn, A. J. and Bederson, B. B. (2009). A taxonomy of distributed human computation. Technical Report HCIL-2009-23, University of Maryland, College Park.

Quinn, A. J. and Bederson, B. B. (2011). Human Computation: A survey and taxonomy of a growing field. In *Proceedings of the SIGCHI Conference on*

Human Factors in Computing Systems, CHI '11, pages 1403–1412, New York, NY, USA. ACM.

Quoc Viet Hung, N., Tam, N., Tran, L., and Aberer, K. (2013). An evaluation of aggregation techniques in crowdsourcing. In Lin, X., Manolopoulos, Y., Srivastava, D., and Huang, G., editors, *Web Information Systems Engineering – WISE 2013*, volume 8181 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg.

Ra, M.-R., Liu, B., La Porta, T. F., and Govindan, R. (2012). Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 337–350, New York, NY, USA. ACM.

Raykar, V. C., Yu, S., Zhao, L. H., Jerebko, A., Florin, C., Valadez, G. H., Bogoni, L., and Moy, L. (2009). Supervised learning from multiple experts: Whom to trust when everyone lies a bit. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 889–896, New York, NY, USA. ACM.

Raykar, V. C., Yu, S., Zhao, L. H., Valadez, G. H., Florin, C., Bogoni, L., and Moy, L. (2010). Learning from crowds. *The Journal of Machine Learning Research*, 11:1297–1322.

Reddy, S., Estrin, D., and Srivastava, M. (2010). Recruitment framework for participatory sensing data collections. In Floréen, P., Krüger, A., and Spasojevic, M., editors, *Pervasive Computing*, volume 6030 of *Lecture Notes in Computer Science*, pages 138–155. Springer Berlin Heidelberg.

Ren, J., Zhang, Y., Zhang, K., and Shen, X. (2015). Exploiting mobile crowdsourcing for pervasive cloud services: challenges and solutions. *Communications Magazine, IEEE*, 53(3):98–105.

Rheingold, H. (2002). *Smart Mobs: The Next Social Revolution*. Perseus Publishing.

Robinson, J. A. (1971). Computational logic: The unification computation. *Machine intelligence*, 6:63–72.

Rouse, A. C. (2010). A preliminary taxonomy of crowdsourcing. In *Proceeding of Australasian Conference on Information Systems (ACIS 2010)*, number 76.

Schenk, E. and Guittard, C. (2011). Towards a characterization of crowdsourcing practices. *Journal of Innovation Economics & Management*, 7(1):93–107.

Schmidt, L. (2010). Crowdsourcing for human subjects research. In *Proceedings of CrowdConf*, San Francisco, CA, USA.

Sebesta, R. W. (2012). *Concepts of Programming Languages*. Addison-Wesley Publishing Company, USA, 10th edition.

Seltzer, E. and Mahmoudi, D. (2013). Citizen participation, open innovation, and crowdsourcing challenges and opportunities for planning. *Journal of Planning Literature*, 28(1):3–18.

Shih, E., Bahl, P., and Sinclair, M. J. (2002). Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 160–171, New York, NY, USA. ACM.

Shoup, D. C. (2005). *The high cost of free parking*. American Planning Association, Chicago.

Socievole, A., Yoneki, E., De Rango, F., and Crowcroft, J. (2013). Opportunistic message routing using multi-layer social networks. In *Proceedings of the 2Nd ACM Workshop on High Performance Mobile Opportunistic Systems*, HPMOSys '13, pages 39–46, New York, NY, USA. ACM.

Steinfeld, A., Tomasic, A., and Zimmerman, J. (2013). Bringing Customers Back into Transportation: Citizen-driven transit service innovation via social computing. In Susan Bregman, K. E. W., editor, *Best Practices for Transportation Agency Use of Social Media*, pages 164–175. CRC Press, Boca Raton, Florida, USA.

Sterling, L. and Shapiro, E. (1994). *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA.

Surowiecki, J. (2005). *The Wisdom of Crowds*. Knopf Doubleday Publishing Group.

Swan, M. (2012). Crowdsourced health research studies: an important emerging complement to clinical trials in the public health research ecosystem. *Journal of medical Internet research*, 14(2).

Tamilin, A., Carreras, I., Ssebaggala, E., Opira, A., and Conci, N. (2012). Context-aware mobile crowdsourcing. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 717–720, New York, NY, USA. ACM.

To, H., Ghinita, G., and Shahabi, C. (2014). A framework for protecting worker location privacy in spatial crowdsourcing. *Proc. VLDB Endow.*, 7(10):919–930.

Vahdat, A., Becker, D., et al. (2000). Epidemic routing for partially connected ad hoc networks. Technical report, Duke University.

Viredaz, M. A., Brakmo, L. S., and Hamburgen, W. R. (2003). Energy management on handheld devices. *Queue*, 1(7):44–52.

Von Ahn, L. (2005). *Human Computation*. PhD thesis, Pittsburgh, PA, USA.

Von Ahn, L., Maurer, B., McMillen, C., Abraham, D., and Blum, M. (2008). reCAPTCHA: Human-based character recognition via web security measures. *Science*, 321(5895):1465–1468.

Wang, W., Gu, J., Yang, J., and Chen, P. (2013). A group based context-aware strategy for mobile collaborative applications. In Zhang, W., editor, *Advanced Technology in Teaching*, volume 163 of *Advances in Intelligent and Soft Computing*, pages 541–549. Springer Berlin Heidelberg.

Wang, Y., Lin, J., Annavaram, M., Jacobson, Q. A., Hong, J., Krishnamachari, B., and Sadeh, N. (2009). A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services*, MobiSys '09, pages 179–192, New York, NY, USA. ACM.

Welinder, P., Branson, S., Perona, P., and Belongie, S. J. (2010). The multidimensional wisdom of crowds. In Lafferty, J., Williams, C., Shawe-Taylor, J., Zemel, R., and Culotta, A., editors, *Advances in Neural Information Processing Systems 23*, pages 2424–2432. Curran Associates, Inc.

Whitehill, J., Ruvolo, P., Wu, T., Bergsma, J., and Movellan, J. R. (2009). Whose vote should count more: Optimal integration of labels from labelers of unknown expertise. In Bengio, Y., Schuurmans, D., Lafferty, J. D., Williams, C. K. I., and Culotta, A., editors, *Advances in Neural Information Processing Systems 22:*

23rd Annual Conference on Neural Information Processing Systems 2009. Proceedings of a meeting held 7-10 December 2009, Vancouver, British Columbia, Canada., pages 2035–2043. Curran Associates, Inc.

Whitla, P. (2009). Crowdsourcing and its application in marketing activities. *Contemporary Management Research*, 5(1):15–28.

Xiao, Y., Kalyanaraman, R. S., and Yla-Jaaski, A. (2008). Energy consumption of mobile youtube: Quantitative measurement and analysis. In *Proceedings of the 2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, NGMAST '08, pages 61–69, Washington, DC, USA. IEEE Computer Society.

Yan, T., Hoh, B., Ganesan, D., Tracton, K., Iwuchukwu, T., and Lee, J.-S. (2011). Crowdpark: A crowdsourcing-based parking reservation system for mobile phones. *University of Massachusetts at Amherst Tech. Report*.

Yang, D., Xue, G., Fang, X., and Tang, J. (2012). Crowdsourcing to smartphones: Incentive mechanism design for mobile phone sensing. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 173–184, New York, NY, USA. ACM.

Young, A. L. and Quan-Haase, A. (2009). Information revelation and internet privacy concerns on social network sites: A case study of facebook. In *Proceedings of the Fourth International Conference on Communities and Technologies*, pages 265–274, New York, NY, USA. ACM.

Yuen, M.-C., Chen, L.-J., and King, I. (2009). A survey of human computation systems. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 04*, CSE '09, pages 723–728, Washington, DC, USA. IEEE Computer Society.

Yuen, M.-C., King, I., and Leung, K.-S. (2011). A survey of crowdsourcing systems. In *Proceedings of 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust (Passat) and 2011 IEEE Third International Conference on Social Computing (Socialcom) (2011)*, pages 766–773, MIT Media Lab, Boston, USA.

Zambonelli, F. (2011). Pervasive urban crowdsourcing: Visions and challenges. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, pages 578–583.

Zhang, H., Horvitz, E., Miller, R. C., and Parkes, D. C. (2011). Crowdsourcing general computation. In *Proceedings of the 2011 ACM Conference on Human Factors in Computing Systems*, Vancouver, British Columbia. New York, NY. Association for Computing Machinery.

Zhao, Y. and Zhu, Q. (2014). Evaluation on Crowdsourcing Research: Current status and future direction. *Information Systems Frontiers*, 16(3):417–434.

Zhuang, Z., Kim, K.-H., and Singh, J. P. (2010). Improving energy efficiency of location sensing on smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 315–330, New York, NY, USA. ACM.

Zook, M., Graham, M., Shelton, T., and Gorman, S. (2010). Volunteered geographic information and crowdsourcing disaster relief: A case study of the haitian earthquake. *World Medical & Health Policy*, 2(2):7–33.