Towards Declarative Programming for Mobile Crowdsourcing: P2P Aspects

Jurairat Phuttharak and Seng W. Loke
Department of Computer Science and Computer Engineering
La Trobe University, VIC 3086, Australia
e-mail: jphuttharak@students.latrobe.edu.au, s.loke@latrobe.edu.au

Abstract— **Peer-to-Peer technologies have been widely used in networks which manage vast amount of data daily. The proliferation of mobile devices strongly motivates mobile peer-to-peer network (M-P2P) applications, with benefits from network effects. We argue that logic programming for crowdsourcing can be useful in peer-to-peer computing for querying and multicasting tasks shared over peer networks. We introduce a declarative crowdsourcing platform for mobile applications, which combines conventional machine computation and the power of the crowd in social networking, particularly in M-P2P networks. This paper discusses a simple extension of Prolog, which we call *LogicCrowd*, focusing on enabling goal evaluation over peers in mobile peer networks. Additionally, we demonstrate that logic programming for crowdsourcing can be useful in peer-to-peer computing for querying and P2P style of task sharing over short-range networks. In this paper, we illustrate the potential of our approach via programming idioms, a prototype implementation and scenarios.**

Keywords*— *declarative programming language; mobile application; mobile crowdsourcing; peer-to-peer network

I. Introduction

Mobile Peer-to-Peer (M-P2P) networks are an increasingly popular kind of network adapted to mobile computing devices with wireless interfaces. From the physical layer's aspect, mobile peers interact with each other in a peer-to-peer (P2P) fashion. The major characteristic of M-P2P networks is that no centralized server is required, meaning that each peer acquires and provides resources and services to other peers by direct exchange. Moreover, M-P2P networks can be regarded as an ad-hoc network supporting multi-hop routing. M-P2P technologies such as Bluetooth or Wi-Fi Direct are increasingly interesting in mobile applications today. Some M-P2P applications have been designed for providing location-based services. For example, suppose George wants to find a nice restaurant within 1 km of his current location. He could obtain this information by means of M-P2P interactions. Mobile users could check in, share nearby information (photos, news, or deals) and opinions with friends in a M-P2P manner. Indeed, it is also possible for mobile users to discover friends nearby a café, parks, or to arrange meet-ups and conferences via M-P2P collaborations. Notably, some research utilize M-P2P networks contributing toward the crowdsourcing concept which refers to a distributed problem solving model where solutions are obtained by throwing it out to a relatively large, possibly undefined group of people for monetary or ethical benefits through an open call [1]. There have been explorations of crowdsourcing for the mobile environment [2-6]. Crowdsourcing via smartphones has been reviewed in [7], providing a taxonomy of mobile crowdsourcing in terms of new applications and similar services based on crowd-generated data. Txtagle [2], MobileWorks [3], UbiAsk [4], CrowdITS [5] and Smart Mob [6] are crowdsourcing applications enhanced with a range of different sensors such as camera, GPS, communication signals, accelerometer and so on. Moreover, CrowdDB [8,9] proposed extensions, to establish a query language and processing techniques for crowdsourcing in order to integrate human input for processing queries that a normal database system cannot answer.

The aim of our research is to investigate a declarative programming paradigm integrated with crowdsourcing for mobile environments. 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. 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 M-P2P networks. We first introduced the key concept of logic programming in *LogicCrowd* and its architecture in [10], but this paper will expand on and focus on its M-P2P aspects. Furthermore, the study of energy consumption characteristics for our *LogicCrowd* prototype had been previously explored in [11] which showed the relationship between energy consumption and two different types of crowd executions with different kinds of network connections. In [11], an energy-consumption crowdsourcing model for the Android platform was developed and we also implemented the notion of *computing with an energy budget* via an extension of the *LogicCrowd* meta-interpreter. In this paper, we focus on applying the declarative programming paradigm for P2P-style crowdsourcing.

The rest of this paper is organized as follows. Section II gives a brief overview of the overall *LogicCrowd* platform. Section III introduces the M-P2P aspects of *LogicCrowd*. The working of the platform is illustrated in Section IV and Section V concludes with future work.

II. An Overview of LogicCrowd

This section provides a brief outline of the *LogicCrowd* formalism, comprising the notion of the crowdsourcing programs and an extension of Prolog. At this stage, *LogicCrowd* is designed for users or mobile developers who have a basic background on Prolog, through syntactic sugar and UI forms can also be used.

A. *LogicCrowd - Logic Crowdsourcing Program*

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. An operational semantics for the language of pure Prolog can be 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. For example, the following shows a query to the crowd in order to ask for a nice Thai restaurant:

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

The question and options predicates are in the crowd's conditions acting as parameters for sending queries to the open crowd. In our work, we define the relevant crowd's 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 a rule using a crowd predicate in a *LogicCrowd* program for recommending well-known restaurants.

```
recommend(Restaurant):-thai(R), nice?(Restaurant)#[asktype("choice"),
          question("Which restaurant do you recommend?"),
          options(R), 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 search for Thai restaurants. The first sub-goal, represented by `thai(R)`, will start with 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 an input to the crowd query. The input will be next sent to the crowd in the second sub-goal in order to find out which restaurants in the list of Thai restaurants would be recommended by the crowd. In the crowd condition, the parameter pertaining to the question ("Which restaurant do you recommend?"), options (choices), target addressees as well as the specific expiry time for returning the results are made explicit. We provide several options to ask the crowd: 1) sending the queries via a social network, i.e. Facebook, and 2) sending queries to a peer-to-peer network, i.e. via Bluetooth. Other potions include connecting to Amazon's Mechanical Turk or other crowd platforms. LogicCrowd, hence, can provide a uniform programmatic interface to multiple crowdsourcing platforms, if required.

B. *LogicCrowd Goals*

In the previous sub-section (Section II A), the crowd operators "?" and "#" can be embedded into Prolog programs as a distinguished predicate, referring to crowd identity and crowd conditions. These operators encoded by extending the basic Prolog meta-interpreter (implemented on tuProlog for Android) have been specifically designed to have a number of capabilities relating to working with the crowd via social network(s) and peer-to-peer networks. The *LogicCrowd* meta-interpreter is given as follows.

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

solve(Askcrowd?Result#Condition):- !,solvecond(Condition),
    (asyn,!,asynproc(Askcrowd,Result);synproc(Askcrowd,Result)),
     doretraction.
synproc(Askcrowd,Result):
    checkcond(TypeQuestion,Question,Picture,Options,Askto,Group,
             Locatedin,Expiry),
    askcrowd(Askcrowd,TypeQuestion,Question,Options,Picture,Askto,
             Group,Locatedin, Expiry,QuestionID),
    registercallbacksyn(QuestionID,Question,Askto,TypeQuestion,
             Expiry,Result).
asynproc(Askcrowd,Result):-
    checkcond(TypeQuestion,Question,Picture,Options,Askto,Group,
            Locatedin,Expiry),
    askcrowd(Askcrowd,TypeQuestion,Question,Options,Picture,Askto,
            Group,Locatedin,Expiry,QuestionID),
    registercallbackasyn(Askcrowd,QuestionID,Question,Askto,
            TypeQuestion,Expiry,Result).
doretraction :-(asyn,!,retract(asyn);retract(syn)).
solvecond([]):- !.
solvecond(Condition):-Condition =..[_H|[Head,Body]],asserta(Head),
                      solvecond(Body).
checkcond(TypeQuestion,Question,Picture,Options,Askto,Group,Locatedin
          ,Expiry):-(asktype(A),!,TypeQuestion=A; set(TypeQuestion)),
          (question(B),!, Question = B; set(Question)),
          (picture(F),!, Picture = F; set(Picture)),
          (options(G),!, Options =G; set(Options)),
          (askto(H),!,Askto = H; set(Askto)),
          (group(I),!,Group = I; set(Group)),
          (locatedin(J),!,Locatedin = J; set(Locatedin)),
          (expiry(K),!,Expiry = K; set(Expiry)).
set(X):- X = 'null'.
```

From the above rules, the `solve/1` predicate is a meta-interpreter for pure Prolog extended to evaluate goals with the crowd operators (`?` and `#`). This rule delegates evaluation of such goals to `solvecond/1`, `asynproc/2`, `synproc/2`, `doretraction/0` predicates. The `solvecond/1` predicate represents a meta-interpreter for the crowd conditions. It will assert the new fact (the crowd conditions) into the knowledge base. The `synproc/2` and `asynproc/2` predicates distinguish between synchronous and asynchronous executions (these will be explained further in the next sub-section). Both rules contain calls to the `checkcond/8` predicate and `askcrowd/10` predicate. The first is to bind the values of the crowd conditions to actual variables and set "null" to variables in case the fact is not in the knowledge base and the latter is to connect to the crowd by passing the crowd conditions to a process outside of the main tuProlog thread. The goal `registercallbacksyn/6` is called when the execution is synchronous and its function is to register tasks (which have been sent to the crowd) and to manage returned results. It should be noted that, in terms of the general concepts, `registercallbackasyn/7` is rather similar to `registercallbacksyn/6`, but it can solely operate in the asynchronous mode.

C. *Synchronous and Asynchronous Executions*

One of the most important issues about crowdsourcing is how to manage the answer(s) returned by the crowd. Since a delay from the crowd in providing answers via social media networks and peer-to-peer networks is expected. *LogicCrowd* has been designed to tackle this issue. We developed two different methods to evaluate the rules: synchronous and asynchronous executions.

In the synchronous operation, we implement *LogicCrowd* according to the standard Prolog program execution model. The model is running sequentially without any parallel extensions - when *LogicCrowd* is executing a crowd predicate, the evaluation will be suspended until the system receives the answers from the crowd. The query is issued in the synchronous mode when we have the atom "`syn`" in the crowd conditions. We support this mechanism via a small extension to the crowd predicate as shown in the following form below.

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

In the asynchronous operation, the multi-threading capability available is exploited since *LogicCrowd.* As such, a new thread is created for each such asynchronous crowd predicate evaluation, to run independently. We have a crowd predicate of the form:

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

The asynchronous execution takes place when we specify the atom "asyn" in the crowd conditions. In contrast to the synchronous processing, asynchronous operation would permit other goals to continue before answers from the crowd return. In our case, when the asynchronous method is used, the evaluation of the crowd predicate will be in a newly created background thread, and the next sub-goal can be executed without blocking the crowd predicate that is waiting for results.

III. Crowdsourcing in Mobile 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 with 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. To model 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 term of 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.

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

A. Extending Prolog for mobile peer-to-peer quering

As we 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 the queries among each other using that 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 value to limit the lifetime of tasks so that the action of forwarding tasks to other peers can been stopped. We explore an extension of Prolog in with new constructs that enable query evaluations to take place over multiple hops in M-P2P networks.

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 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 as queries. In our case, Bluetooth connection has been 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 a M-P2P network has the means to receive, to process, and if necessary, to 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 restaurants 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,bluetooth]),expire("0,30,0")]` will be 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 "`*`" operator will be 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 flaw 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 following algorithm shows the implementation in our *LogicCrowd*'s Mediator.

```
Define $T$ :- $t_1, t_{2,}...,t_i$  where $T$ is a set of taskID in receiver's DB
@ Initial state (sender)        // In original, mobile's 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 $\notin$ $T$) then// Check if the received taskID is not in local DB
            set $T = T \cup$ {taskID      // Store taskID
            broadcast(task,taskID)// Distribute a task to crowd with taskID
      end
```

In the crowd algorithm, 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.

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 query friends of friends by running a logic program in itself or remote peers, i.e. instead of friends passing the queries on, the origin peer gets its friends' friends and passes the query on itself (providing they are in network range). For this, we implement a crowd predicate formulated with the following construct: **`PeerID*Task`**. 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 as follows 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)`**. As an example, the following Prolog program supports the centralized-control P2P crowdsourcing model by allowing the task's owner/original peer to access the friends list of friends recursively. In doing so, the original 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 (TTL) and power-to-live (PTL) 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 in the form: **`self*Task`** that 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. Evaluation completes when the result from crowd has been returned to the origin peer due to expiry as in the crowd conditions.

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

B. Peers propagation control mechanism

To reduce or even avoid endless loops for propagating the task among peer networks, Power-to-Live (PTL) and Time-to-Live (TTL) have been defined. PTL is a maximum value of energy which is determined by each peer in order to be able to propagate tasks through peer networks. The limited resources of mobile devices especially battery capacity can solve the infinite loop problem when propagating the tasks in peer networks. The PTL value is set by an energy budget corresponding to a current energy level. PTL can be calculated by the following equation: PTL = β (%) $\times$ $E_{current}$. 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 ($E_{broadcast}$) (e.g., via preprogrammed benchmarks). To allow distributing tasks among peer networks, we use the condition given as follows: $E_{broadcast} \leq$ PTL. According to this relation, the task is allowed to forward only when the energy estimated for that task, i.e. $E_{broadcast}$, is less than or equal to PTL. 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 1200 mAh. When propagating a task to peers, the energy consumed is estimated to be 182.2 mAh, which is less than the energy budget (300 mAh); 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 TTL value which limits the lifetime of the task and each peer keeps track of the tasks, which it forwarded. Every task/goal is tagged with a TTL value which is modified (with time taken so far subtracted from it) across queries. In our case, the TTL 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 TTL value for tagging queries when forwarding to peers:

$$\text{TTL} = \text{EndTime}_{\textit{orginal}} - \text{StartTime}_{\textit{forwarder}} - \text{BufferTime}$$

According to the relation above, $\text{EndTime}_{orginal}$ refers to the time of the original/task's owner waiting for answers back from the other peers, while $\text{StartTime}_{forwarder}$ defines the start time of the sender/forwarder who issued the query to other peers. BufferTime relates to the time for queuing, waiting, setting up or running the system. The BufferTime value could be a default value preset in the *LogicCrowd* program or a user-defined value. A minimum value of TTL should be greater than or equal to the BufferTime of each device. TTL-tagged queries can solve or ameliorate the unbounded problem of distributing tasks on peer networks. Moreover, both mechanisms can be managed to work cooperatively by setting the priority of PTL to be higher than that of TTL. For example, in the case where we want to forward a task to other peers, the system will check whether the value of PPL is sufficient for such forwarding. Normally, when PPL is found sufficient, the forwarding process will run. However, if we lack sufficient TTL or PTL, the task will not be fowarded.

C. 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. We divided the methods to deal with peer responses into two abstract models as follows.

1) Black-box crowd answering

This method relates to the decentralized-control P2P crowdsourcing model as mentioned in Section III (A) in which every peer processes queries and distributes tasks independently, without the origin 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 programming abstraction 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 crowd responses with that answer for an answer to be valid. The *LogicCrowd* program will process and collect the responses from the crowd until the total number of responding peers reaches such a minimum. Also, the system can send query results back to origin peers regardless of the expiry of a wait which has been set by the crowd's conditions. Secondly, a goal of the form: **`<Number>*Task`** evaluates in such the way that the `Task` must be broadcasted to peers and the `Number` denotes the majority of peers' answers. In dealing with answers, those results from peers will be collected and calculated. If an answer is found where the number of peers with this answer is the majority requirement set by the origin, the system will automatically return those responses - there is no need for a delay until the expiry period imposed in the original crowd conditions. The two abstract mechanisms could control the return of peer responses and offer a more flexible approach. Meanwhile, the origin can have some control of the extent to which tasks are supposed to be sent to peers.

2) Transparent crowd answering

This method has been designed for the centralized-control P2P crowdsourcing model as mentioned in Section III(A) in which the origin 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 origin in order to choose appropriate friends to forward the tasks. Obviously, this relies on the origin peer being able to gather, in a transparent way, the identifiers of the friends of its friends (an their friends, etc). The advantage of this method is that the origin peer is able to control the peer network by selecting the peers who he/she would like to broadcast to. By this method, the results are likely to be promising - regarded as potentially more reliable and higher in validity. However, one possible drawback is that since an origin peer is the only one who processes all answers sent by peers, the load is much heavier on the origin peer.

IV. Prototype Implementation and Scenarios

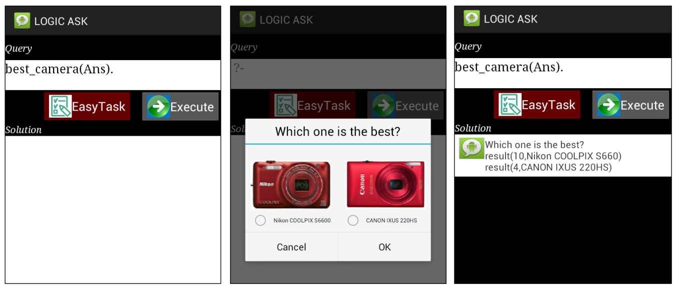
In this section, our implementation is illustrated via scenarios. Currently, we have built a prototype implementation of *LogicCrowd* by designing and executing it on the Android platform. The prototype integrates *tuProlog* with Android via our own custom-built Java program called a Mediator which is the middleware between Prolog and our M-P2P protocols. We believe that the logic programming approach can go further than imperative approaches, as it would enable rule-based reasoning with resource descriptions and more expressive queries. Below, we describe examples in greater detail and point out several applications that relates to utilization of M-P2P network in real world scenarios.

A. Subjective 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 anyone whether those pictures are the same or not. For *LogicCrowd*, we develop declarative crowd query operators which are lightweight extensions of Prolog for comparing items/tasks. The first scenario is the application of *LogicCrowd* to compare photos. For example, the photos of two models of cameras are sent to the crowd via *LogicCrowd* (i.e. people in a particular area) to ask for their comparative suggestions about which model would be regarded as better to buy. The query for this is:

```
best_camera(Ans):-camera(X),camera(Y),photo(X,A),photo(Y,B),
      <10>*best?(Ans)#[syn,asktype('compare'),question('
       Which one is the best?'),options([X,Y]), file([A,B]),
        askto([bluetooth]),expiry('0,30,0')].
```

The sub-goal camera/1 succeeds if X and Y are currently instantiated to a camera while sub-goal photo/2 succeeds if A is X's photo and B is Y's photo. In this rule, a crowd predicate is one sub-goal of the main goal. The crowd predicate identified using the crowd keyword "best" is called as the open predicate to peers. The crowd query has been sent via Bluetooth connection in a synchronous mode within expiry of 30 minutes. Here, *LogicCrowd* has two new crowd conditions built in: `asktype('compare')` is a type of task and `file([A,B])` is a list of related multimedia items which will be sent to the crowd. In this crowd predicate, the extra condition <10>* is added to enable peers to forward the task under the condition that, if any answer (in this case either camera as being "better") has been sent by at least 10 people, that answer that has 10 people's support can be returned to the system autonomatically with no need to wait for the expiry time. After executing the crowd sub-goal, the question with the above conditions then appears on peer mobiles as illustrated in Figures 1(a) and 1(b). After a while, peers are supposed to answer the request by comparing camera's features. Within 30 minutes,,the result is returned to the query origin in *LogicCrowd*. This scenario is programmed to display the results, as shown in Figure 1(c).



```
best_camera(Ans).
```

```
?-
```

Which one is the best?

```
Which one is the best?
result(10,Nikon COOLPIX S660)
result(4,CANON IXUS 220HS)
```

(a) Execute Goal (b) Send Task to Crowd (c) Display Result

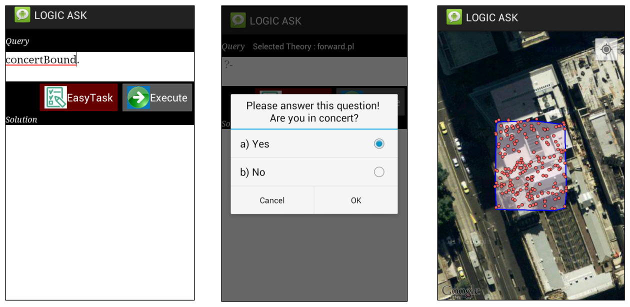
Fig. 1. Broadcasting comparison task via Bluetooth and showing the results.

B. *Destination and Boundary Finding Application*

Problems such as finding parking lots, looking for gift shop special offers and even finding disabled toilets or lifts in a particular area involve location finding. Apparently, it is difficult for anyone to be able to find the geo-position accurately and appropriately which could best serve individual users' requirements. This problem is unavoidable since most data required is 'real-time' and thus needs to be updated all the time. Furthermore, in a situation like looking for a good seat in a big football stadium, we might find it difficult to find the best seating options located in the zone of the same team supporters. Also, in the face of natural disastrous situations, like a tsunami and bushfire, it is crucial to escape from the site as fast as possible. As such, finding the boundary of effective zones can be most challenging. These problems could be solved by seeking help from the crowd to provide the best answer. In *LogicCrowd*, one can send a query via Bluetooth or Wi-Fi Direct technologies to the crowd. We exploit GPS coordinates and Google map API in mobile devices to create a destination point or boundary area. Here is an example to show how the *LogicCrowd* program can be applied. In this scenario, a *LogicCrowd* program finds the boundary of a particular area. Suppose Jim is in a huge music concert. He wants to leave the concert for some reason. With a number of people around him, he needs to find out which way is nearest to the exit gate. The program can be written as the rule below.

```
concertBound:-*inConcert?(Coordinates)#[syn,asktype('geo'),
        question(Are you in concert?),option(['yes','no']),
        askto([bluetooth]),expiry('1,0,0'),
        bound(Coordinates,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 among people in a 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 shown in Figure 2(a), the question with these conditions then appears on the friends' devices connected via Bluetooth in Figure 2(b). A while later, each peer propagates this question among his/her friends. Within 1 hour, on expiry, the result is returned to the query origin via *LogicCrowd*, where `bound/2` is programmed for drawing the boundary using the coordinates and `currentLoc/1` uses GPS to get the current location and finally `showMap/2` will display the boundary and current locations of responders on Google Map, as in Figure 2(c).



```
concertBound.
```

```
?-
```

Please answer this question!
Are you in concert?

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

Fig. 2. Distributing geo task among peers via Bluetooth to find the boundary

V. CONCLUSION

We have presented an approach towards integrative use of crowdsourcing and mobile peer-to-peer networks within a declarative programming paradigm. We have also argued that logic programming for crowdsourcing can be useful in peer-to-peer computing for querying and broadcasting tasks shared over peer networks. In *LogicCrowd*, Prolog has been used as a specification notation, as a rapid prototyping language, and for convenient coding of P2P queries.

REFERENCES

[1] J. Howe, "The rise of crowdsourcing," Wired magazine, vol. 14, pp. 1-4, 2006.

[2] N. Eagle, "txteagle: Mobile Crowdsourcing," in Internationalization, Design and Global Development. vol. 5623, N. Aykin, Ed., ed: Springer Berlin Heidelberg, 2009, pp. 447-456.

[3] P. Narula, P. Gutheim, D. Rolnitzky, A. Kulkarni, and B. Hartmann, "MobileWorks: A Mobile Crowdsourcing Platform for Workers at the Bottom of the Pyramid," *Human Computation,volume WS-11-11 of AAAI Workshops, AAAI, 2011*.

[4] Y. Liu, V. Lehdonvirta, T. Alexandrova, M. Liu, and T. Nakajima, "Engaging Social Medias: Case Mobile Crowdsourcing," SoME'11, 2011.

[5] K. Ali, D. Al-Yaseen, A. Ejaz, T. Javed, and H. S. Hassanein, "Crowdits: Crowdsourcing in intelligent transportation systems," Proc. of the Wireless Communications and Networking Conference (WCNC), 2012, pp. 3307-3311.

[6] H. Rheingold. "Smart Mobs." Perseus Publishing, 2003.

[7] G. Chatzimilioudis, A. Konstantinidis, C. Laoudias, and D. Zeinalipour-Yazti, "Crowdsourcing with Smartphones," Internet Computing, IEEE, vol. 16, pp. 36-44, 2012.

[8] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin, "CrowdDB: answering queries with crowdsourcing," SIGMOD Conference, 2011, pp. 61-72.

[9] A. Feng, M. Franklin, D. Kossmann, T. Kraska, S. Madden, S. Ramesh, et al., "Crowddb: Query processing with the vldb crowd," Proceedings of the VLDB Endowment, vol. 4, 2011.

[10] J. Phuttharak and S. W. Loke, "LogicCrowd: A Declarative Programming Platform for Mobile Crowdsourcing," Proc. of the Int. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom), 2013, pp. 1323-1330.

[11] J. Phuttharak and S. Loke, " Declarative Programming for Mobile Crowdsourcing: Energy Considerations and Applications," Proc. of the 10th Intern. Conf. on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous-2013), Tokyo, Japan, 2013.