LogicCrowd: a Declarative Programming Platform for Mobile Crowdsourcing

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*— We present an attempt to engage social media networks, bringing the crowdsourcing model into mobile environments. We introduce *LogicCrowd*, a declarative programming paradigm for mobile crowdsourcing applications, developed as an extension of Prolog. *LogicCrowd* aims at filling the gap between traditional machine computation, which operates upon its database, and social media, which is capable of interacting with real people. 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; social media network

I. INTRODUCTION

In recent years, the emergence of the crowdsourcing paradigm has dramatically brought change in the landscape of solving complex problems. Often, tasks such as translation, audio transcription, rating products, photo tagging, and so on, seem to be difficult to do by only a machine. However, integrating human intelligence with machine computation is expected to provide a great promise for increasing capabilities to complete complex tasks.

Crowdsourcing is simply known as the power of crowd. The term was originally coined in an article by Jeff Howe [1], and refers to the idea of outsourcing some kind of task to a larger group of people in the form of an open call. The success in the implementation of the idea has been evidenced in many business platforms, such as Amazon's Mechanical Turk (MTurk) [1], iStockPhoto [2], and Crowdflower [3]. Such platforms support crowdsourced execution of Microtasks or Human Intelligence Tasks (HITs), in which people do simple jobs requiring little or no domain expertise via the Internet, and get paid on a per-job basis when it is completed. Nowadays, the use of mobile phones is widespread in both urban and rural areas. It is common that people spend a large amount of time commuting or waiting for various events. Most of them are engaged in a variety of pleasurable activities, like playing with their mobile phones. Hence, mixing the mobile platform and the crowdsourcing model is expected to potentially offer vast resources for computation.

In our work, we explore a crowdsourcing platform designed for mobile users. There are two main justifications for the exploration. First, crowdsourcing is still a relatively new research area in mobile computing and has not yet fully penetrated the mobile workforce. Second, mobile phones not only offer great functionalities including multi-sensing capabilities (e.g., geo-location, movement or audio and visual sensors) but also provide efficient ways to collect data and such capabilities can extend existing Web-based crowdsourcing applications.

The aim of this paper is to propose a declarative programming paradigm for leveraging the knowledge of people through crowdsourcing. *LogicCrowd* is proposed as an innovative approach that integrates logic programming into crowdsourcing middleware in order to provide a declarative programming platform for mobile apps that can use crowdsourcing. The use of logic programming is aimed at providing 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 this paper.

The rest of this paper starts with the introduction of motivating examples of *LogicCrowd* in Section II, and then, briefly sketches the syntax and semantics of *LogicCrowd*, focusing on an extension of the Prolog meta-interpreter in Section III. The architecture of our mobile crowdsourcing platform is explained in Section IV. The working of the platform is illustrated in Section V. Related work is given in Section VI. Section VII concludes with future work.

II. MOTIVATING EXAMPLES

LogicCrowd, a declarative crowdsourcing platform, is built on top of existing social networking infrastructure for bringing the crowdsourcing model into the mobile context. Here is a list of tasks/programs that we require that *LogicCrowd* should be able to support:

- Given datasets including sensing data or interesting items, provide a recommendation to the user based on those datasets using the crowd to determine decisions.

[1] http://www.mturk.com
[2] http://www.istockphoto.com
[3] http://www.crowdflower.com

- Given a list of products, rank them by "quality" by asking for experts or users who have used the products.
- Ask the crowd's opinion on issues; identify/express sentiments such as like or dislike, and other feedback.

Due to the fact that *LogicCrowd* is programmed based on Prolog, which is a popular logic programming language [2], there is opportunity for the applications to include reasoning, above and beyond the exemplified tasks. Other logic programming languages can be employed but we start with Prolog due to its relatively simple semantics and popularity. Since the mobile users/developers can fix/decide on facts and rules (users' independence), the application becomes configurable and reprogrammable. In the present paper, we will provide some examples in the light of the three types of tasks mentioned previously, together with presenting a relevant prototype.

III. PROGRAMMING IN LOGICCROWD

Our working hypothesis is that, the programs, which are executed by both the machines and the crowd, can work well when people behave rationally. *LogicCrowd*'s three extensions to Prolog are as follows: (1) it allows some predicates which are open, i.e. 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 have a number of capabilities relating to working with the crowd via social network(s); and (3) it provides a choice between synchronous and asynchronous calls for crowd execution of queries, making computation more flexible when engaging the crowd.

A. LogicCrowd – Logic Crowdsourcing Programs

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. For example, the following shows a query to the crowd in order to ask for a nice restaurant:

```
delicious?(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 the 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), delicious?(Restaurant)#
                    [asktype("choice",
                    question("Which restaurant do you recommend?"),
                    options(R), askto("closed_friend"),
                    locatedin("Bangkok"), expiry("2,0,0")].
```

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 prefer?"), options, target addressees and their current locations, as well as the specific expiry time for returning the feedback are made explicit. Apart from recommending a place, we can have rules for ranking the commercial products to serve business demands as follows:

```
best_handbag(Bestbag):- brand(Handbag), besthandbag?(Popbag)#
                    [asktype("choice"),
                     question("What is the most popular handbag?"),
                     options(Handbag), askto("public"), expiry("@")],
                     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, the symbol "`@`" will be used to allow the non-expiration period of time for that question. The system will return feedback from the crowd according to the time set by the users. After that, the system will rank the handbags (derived from crowd) in order of their popularity.

B. Extending Meta-Interpreter in LogicCrowd

The *LogicCrowd* meta-interpreter is presented in a simplified form as an extension of pure Prolog (in practice, we used tuProlog) which could be easily extended to accommodate calls to tuProlog [3] via built-in libraries. 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),
   checkcond(TypeQuestion,Question,Options,Askto,Locatedin,Expiry),
   askcrowd(Askcrowd,TypeQuestion,Question,Options,Askto,Locatedin,
            Expiry,QuestionID),
   registercallback(QuestionID,TypeQuestion,Expiry,Result).

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

checkcond(TypeQuestion,Question,Options,Askto,Locatedin,Expiry):-
         (asktype(A),!, TypeQuestion = A; set(TypeQuestion)),
         (question(B),!, Question = B; set(Question)),
         (options(C),!, Options =C; set(Options)),
         (askto(D),!,Askto = D; set(Askto)),
         (locatedin(E),!,Locatedin = E; set(Locatedin)),
         (expiry(F),!,Expiry = F; set(Expiry)).

set(X):- X = 'null'.
```

In the previous sub-section (Section III A), the *crowd operators* "?" and "#" can be embedded into Prolog programs as a distinguished predicate, referring to crowd identity and crowd conditions. 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 evaluation of such goals to `solvecond/1`, `checkcond/6`, `askcrowd/8` and `registercallback/4` predicates:

```
solve(Askcrowd?Result#Condition):- !,
   solvecond(Condition),
   checkcond(TypeQuestion,Question,Options,Askto,Locatedin,Expiry),
   askcrowd(Askcrowd,TypeQuestion,Question,Options,Askto,Locatedin,
            Expiry,QuestionID),
   registercallback(QuestionID,TypeQuestion,Expiry,Result).
```

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

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

As, the `checkcond/6` predicate will bind the values of the crowd conditions to actual variables, and it will set "null" to variables in case that the fact is not in the knowledge base.

```
checkcond(TypeQuestion,Question,Options,Askto,Locatedin,Expiry):-
        (asktype(A),!, TypeQuestion = A; set(TypeQuestion)),
        (question(B),!, Question = B; set(Question)),
        (options(C),!, Options =C; set(Options)),
        (askto(D),!,Askto = D; set(Askto)),
        (locatedin(E),!,Locatedin = E; set(Locatedin)),
        (expiry(F),!,Expiry = F; set(Expiry)).

set(X):- X = 'null'.
```

Regarding the `askcrowd/8` predicate, it functions to connect to the crowd. The `askcrowd/8` will pass the crowd conditions to a process outside of the main tuProlog thread; in our case, the particular social media network we currently use is Facebook[4], though other social networking platforms can be integrated. The query is issued to the crowd (i.e., friends on Facebook) via the Mediator which is running on the Android platform (this will be explained further in (Section IV) The `registercallback/4` predicate is 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 called results. After posting the task on the crowd (i.e., Facebook), this predicate will register the task and return all results from the crowd back to main program after the expiration time.

C. Synchronous/Asynchronous Execution

One of the most important issues about crowdsourcing is how to manage the answer(s) returned by the crowd. Because there may be a delay for the crowd to provide the answers via social media networking (such as Facebook), *LogicCrowd* has been designed to tackle this delay. First, the `registercallback/4` predicate is designed for registering and handling the returned results from the crowd by using Mediator as noted in the previous sub-section. 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 is running 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 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(Restaurant):- findall(X,(thai(X),melbourne(X)),R),
                        delicious?(Restaurant)#[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 is returned, i.e. the crowd predicate suspends until results come back or until expiry (in the synchronous mode); in this case, it may take about two hours. After that, `getLocation/2` and `show/1` predicates will be executed respectively.

```
recommendOne:- findall(X,(thai(X),melbourne(X)),R),
               delicious?(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 asynchronous mode, 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's conditions. In contrast to the synchronous operation, asynchronous processing enables methods to return immediately without blocking on the calling thread. It means that the crowd predicate is evaluated in the background, not blocking. As a result, the next sub-goal can be executed. The rule below illustrates an

[4] http://www.facebook.com

asynchronous version of the recommendation example above.

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

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

This example appends `doSomethingElse/2` predicate at the end of the goal. This predicate is executed without waiting for results to come back from the crowd. In the asynchronous operation, the programmer needs 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 exactly this 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 gets the top score from crowd's vote for the query `delicious`.

IV. LogicCrowd Architecture

LogicCrowd is a declarative programming paradigm for the mobile platform which is designed to combine conventional machine computation and the power of the crowd in social media networks. Currently, we have built an initial 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. The corresponding high-level architecture of the prototype system is described in Figure 1.

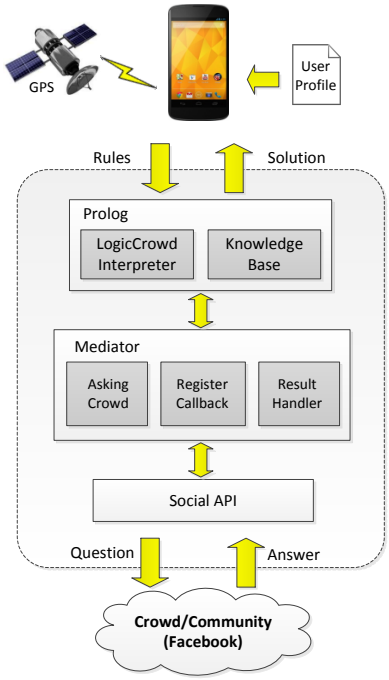


Figure 1. A LogicCrowd prototype architecture.

Our prototype implementation comprises three main components: (i) a Prolog system (i.e., tuProlog, running on Android), which contains the *LogicCrowd* interpreter and a knowledge base of the user profile and relevant information; (ii) Social API which is a software protocol to interact with the social media network; and (iii) a mediator between Prolog and Social API which is to (1) execute crowd queries, (2) register the issued tasks/queries, and (3) handle the results from the crowd.

The execution process of *LogicCrowd* is as follows: 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 the 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 will be executed via the Mediator using the "askcrowd" method to send those conditions to the social media platform, and the task will be registered by the "registercallback" method. During this step, there may be a time lag due to waiting for the return of feedback from the crowd. The answer from the crowd will be managed through the "resulthandler" method in order to display to the mobile user results, or to return the results by instantiating Prolog variables. Figure 2 illustrates the sequence diagram of LogicCrowd in the synchronous operation.

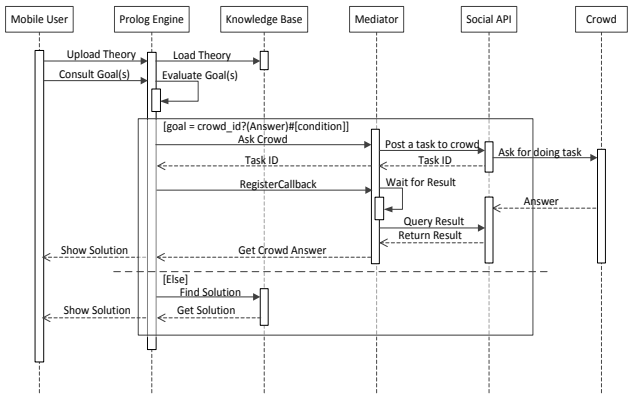


Figure 2. A sequence diagram for synchronous mode.

The asynchronous operation mode is illustrated in Figure 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 in waiting for the results from crowd. In Figure 3, the process is similar to that in the synchronous mode, but the difference happens after calling the "registercallback" 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 will be executed simultaneously without being blocked by an incomplete processing of a crowd predicate. After the return of the results from the crowd, a corresponding `handle_crowd_answer/3` rule will then be executed

for doing further processing with the crowd's result or simply displaying the results to the user.

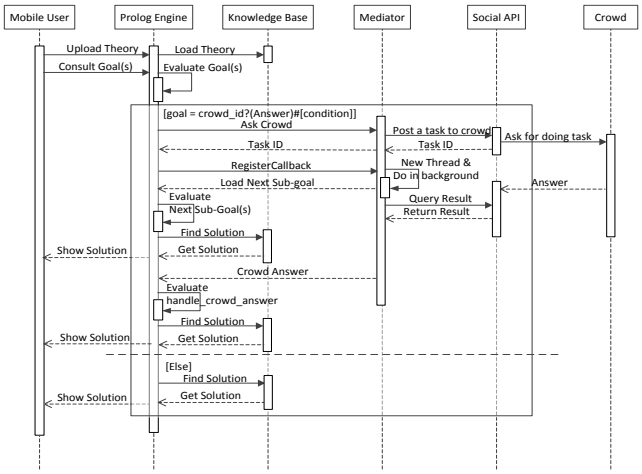
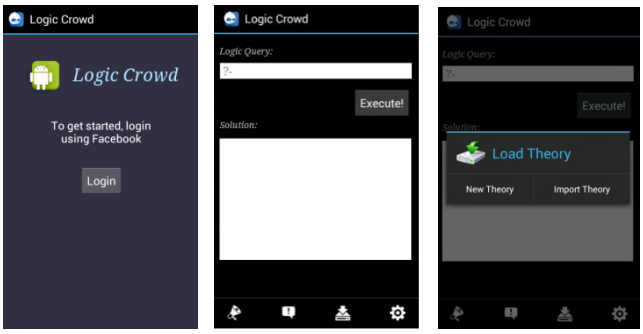


Figure 3. A sequence diagram for asynchronous mode.

V. PROTOTYPE IMPLEMENTATION AND SCENARIOS

In this section, our implementation is illustrated via scenarios. As mentioned, we extended a meta-interpreter of pure Prolog and created our own Java built-in library called the Mediator between Prolog and social networks. In our case, tuProlog, which is an open-source and a light-weight Prolog framework, has been deployed, as previously stated, since it can be used with Android. Meanwhile, Facebook has been selected for a data-centric crowd computation in our application. API-based solutions for accessing social media platforms are also used. In the *LogicCrowd* framework, the features of the Facebook Graph API have been used (e.g., to create a new question and then to obtain several comments and answers to the question). Figure 4 shows screenshots of the *LogicCrowd* application. To start the prototype, the user must log on to *LogicCrowd* via his/her Facebook account as shown in Figure 4(a). The *LogicCrowd*'s home screen display is shown in Figure 4(b), that is waiting for a user query/command and the result would be shown in the textbox "Solution". To execute the query/command, the relevant theories must be first loaded into the database theory, done by the user creating a new theory or by importing an existing theory as illustrated in Figure 4(c).

We illustrate the application of our prototype in a real world scenario derived from our motivating example. Consider a mobile user who plans to go out for dinner with family after work. This person has specific requirements concerning the location and type of the restaurant. Here, we assume that the user's particular location and a list of restaurants in any area would be obtained and stored in the local on-mobile knowledge base via GPS and other third party applications, though in practice, Internet queries to on-line services can be also be used to obtain such information. The user's profile and relevant information are held in Prolog form is shown in listing 1.



(a) Login To *LogicCrowd* (b) HomeScreen (c) Load Theory

Figure 4. Screenshots of *LogicCrowd* application.

```
thai('Le Bangkok').  italian('La Porchetta').  halal('Nandos').
thai('Baan Thai').   china('china bar').  indian('Red Pepper').
thai('Narai').       italian('Lupino').   indian('Bonfire Cafe').
thai('ThaiSontaya'). western('Basil').    western('Summer Hill').
```

```
melbourne('Le Bangkok').      melbourne('Basil').
melbourne('Baan Thai').       ivanhoe('Lupino').
melbourne('Thai sontaya').    preston('Red Pepper').
melbourne('china_bar').       preston('Bonfire cafe').
reservior('La Porchetta').    bundoora('Narai').
bundoora('Nandos').           kingsbury('Summer hill').
```

Listing 1. The knowledge base in Prolog format.

The rule below shows the rule/goal, say, programmed by the user.

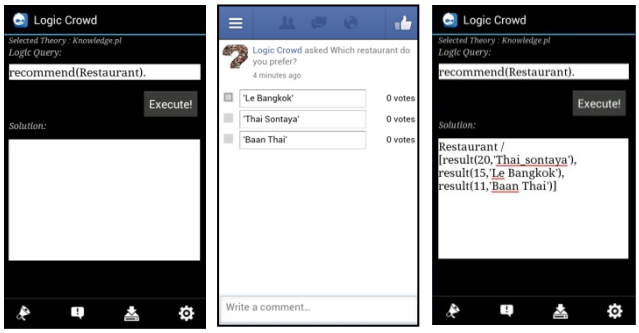
```
recommend(Restaurant):- findall(X,(thai(X),melbourne(X)),R),
                        delicious?(Restaurant)#[syn,
                        question("Which restaurant do you recommend?"),
                        asktype("choice"), options(R), askto("friend"),
                        expiry("3,0,0")].
```

The aim of the goal is to request a recommendation of restaurants on the basis of the restricted conditions that only 'Thai restaurants in Melbourne' are included. In this rule, a crowd predicate is one sub-goal of the main goal. It means that the user would like to ask the crowd by sending the query with conditions as follows:

- Operation: syn (synchronous version).
- Question: Which restaurant do you recommend?
- Type of Question: Multiple choice.
- Options:
  1) Le Bangkok
  2) Thai Sontaya
  3) Baan Thai
- Ask to: Friend.
- Expiry: In 3 hours.

After executing the crowd sub-goal, the question with the above conditions then appears on Facebook as illustrated in Figures 5(a) and 5(b). A while later, several Facebook friends answer the question by voting the restaurant(s) they have found preferable. Within three hours, i.e. the expiry time, the result is returned back to *LogicCrowd*. The system then returns the results to the main goal. Figure 5(c) displays the result consisting of a list of the restaurants and the scores of the restaurants as voted by the crowd.

In this example running on synchronous mode, the crowd predicate is executed sequentially without any parallel extensions. The process will be suspended until the system receives the answers from the crowd. The results show that "Thai Sontaya" has 20 votes, "Le Bangkok" has 15 votes and "Baan Thai" has 13 votes.



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

Figure 5. Sending a restaurant selection request to Facebook and showing the results.

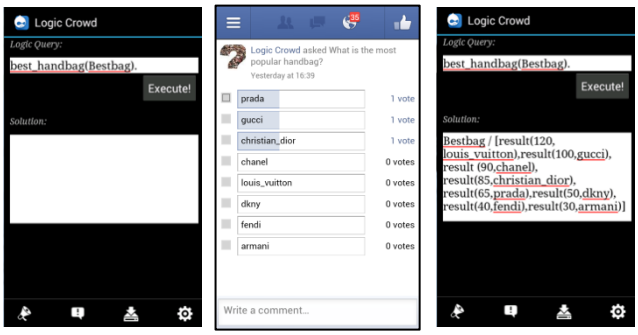
The second scenario invoking mobile crowdsourcing is about ranking products. This example can be applied in retail, e.g., in the research marketing field that determines how to market products. The following rule aims to rank popular handbags by aggregating the power of human and machine computation.

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

We give the second scenario in asynchronous mode. First, all brands of the handbag, collected from inside a knowledge base, are put into the list, in the 'Handbag' variable. Then, the crowd predicate is called as the open predicate to the public; it means that the user would like to create the task and send that to the crowd. The conditions in the query would be explained as follows:

- Operation: asyn (asynchronous version).
- Question: Which is the most popular handbag?
- Type of Question: Multiple choices.
- Options: as in the list contained in the Handbag Variable
- Ask to: Public.
- Expiry: 5 hours.

After executing the crowd sub-goal shown in Figure 6(a), the question with these conditions then appears on Facebook as illustrated in Figure 6(b). With asynchronous operation, if there are goals after the crowd predicate, these goals can execute without delay in waiting for the results from the crowd. After a while, several Facebook friends are supposed to answer the request by voting for the product(s) they have used or found preferable. Within five hours, on expiry, the result is returned to *LogicCrowd*. Then, the `handle_crowd_answer/3` predicate would be automatically executed. In this scenario, `handle_crowd_answer/3` is programmed to sort the list of handbags in order by the highest to lowest voting score. Figure 6(c) displays the final results consisting of a list of the products and the scores of the handbag brands as voted by the crowd – note that the display can, of course, be pretty-formatted depending on the user.



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

Figure 6. Sending a handbag brand ranking request to Facebook and showing the results.

Finally, we show a more complex *LogicCrowd* application. It shows how *LogicCrowd* works in both synchronous and asynchronous mode put into one program. The following rule aims to find the top ranked attractive-tourist places to visit in Bangkok, Thailand, and to find the name of the place via its photo.

```
asktocrowd(Name):-
          findall(Place,thailand(Place,bangkok),Attractive_places),
          placetogo?(_)#[asyn,asktype('choice'), question('Where is
          a must-visit place in Bangkok for holiday?'),
          options(Attractive_places),askto('public'),
          expiry(24,0,0)].
handle_crowd_answer(placetogo,CrowdResult,_):-
          quicksort(CrowdResult,'@>',PlaceList),
          selectTop(1,PlaceList,MostPopPlace),
          photo(MostPopPlace,jpgfile), % get its photo
          place?(Name)#[syn,asktype('photo'),
          question('Where is the place?'),
          picture(jpgfile), askto('friend'), expiry(3,0,0)].
```

The `asktocrowd/1` rule has a goal to ask the crowd for recommendations on the attractive places to visit, i.e. the crowd predicate identified using the crowd keyword "`placetogo`" is called as the open predicate to the social media network. Places get votes from the crowd, and the results are placed in a list. The conditions in the query would be explained as follows:

- Operation: asyn (asynchronous version).
- Question: Where is a must-visit place in Bangkok for holiday?
- Type of Question: Multiple choices.
- Options: as in the list contained in the `Attractive_places` variable
- Ask to: Public.
- Expiry: 24 hours.

Asynchronous execution is used in this sub-goal. The `handle_crowd_ answer/3` predicate is automatically executed by the system after the results are returned. In this case, the top place in the list from the crowd results (instantiated in `CrowdResult`) is selected within the `handle_crowd_answer/3` predicate. Then, the photo (or

its filename) for this place is retrieved via the `photo/2` predicate (assuming a database). The next sub-goal aims to ask the crowd for obtaining the name of the place in the posted photo, using the synchronous mode. The crowd predicate with crowd keyword "`place`" is called as the open predicate to the public. The conditions in the query would be explained as follows.

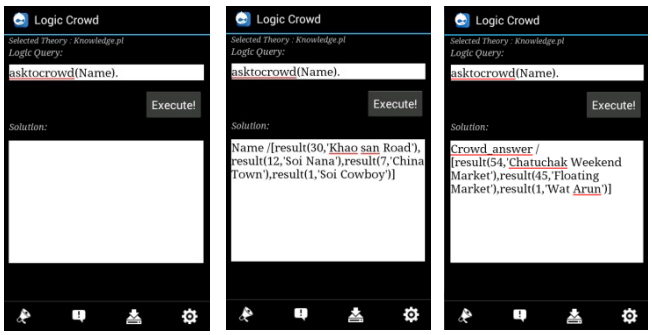
- Operation: syn (synchronous version).
- Question: What is the name of the place in this picture?
- Type of Question: Photo.
- Photo: a.jpg. // suppose this is `jpgfile`
- Ask to: Friend.
- Expiry: 3 hours.

With synchronous operation, the process would be blocked until the crowd's feedback has been returned to *LogicCrowd* (in this sub-goal taking up to 3 hours).

In another variant of the example, consider two goals, one to get the top three places and another to get the name of a place given a photo (e.g., 'a.jpg').

```
asktocrowd(Name):-
         findall(Place,thailand(Place,bangkok),Attractive_places),
         placetogo?(_)#[asyn,asktype('choice'), question('Where is
         a must-visit place in Bangkok for holiday?'),
         options(Attractive_places),askto('public'),
         expiry(24,0,0)],
         place?(Name)#[syn,asktype('photo'),
         question('Where is the the place?'),
         picture('a.jpg'), askto('friend'), expiry(3,0,0)].
handle_crowd_answer(placetogo,CrowdResult,PopPlace):-
         quicksort(CrowdResult,'@>',PlaceList),
         selectTop(3,PlaceList,PopPlaces).
```

In Figure 7(a), executing the main goal is shown. We can see from this scenario that the results in the second (syn) sub-goal will be received prior to the first (asyn) sub-goal. The result is shown in Figure 7(b). After 21 hours, the results of the request in the first sub-goal is returned showing the top three attractive places, as in Figure 7(c).



(a) Execute Goal (b) Result from 2nd sub-goal (c) Result from 1st sub-goal

Figure 7. Sending a place ranking request and asking for comments to Facebook and showing the results.

VI. Related Work

In recent years, there has been increasing interest, especially in the data processing and database community, in using crowdsourcing as a part of database query processing [4-8]. CrowdDB [9,10], TurkDB [11,12,13], Deco [14] proposed extensions, in different aspects, to established query language and processing techniques in order to integrate human input for processing queries that a normal database system cannot answer. A common approach in such studies is to design small extensions to SQL so that the crowd can participate in the process of SQL queries. Meanwhile, Crowd4U [15] leveraged a declarative platform for database abstraction by extending CyLog to issue open queries to the crowd. By these approaches, a declarative paradigm has been employed to leverage crowdsourced data. In *LogicCrowd*, we, however, propose an alternative declarative style of programming for crowdsourcing which leverages on Prolog and is more expressive than simple crowd SQL queries.

There has been prior work that has implemented crowdsourcing on Amazon's Mechanical Turk (MTurk). Turkit[16] and HProc[17], Jabberwocky[18] are a procedural programming libraries designed to optimize worker productivity and tasks which enable programmers to interface with MTurk. In our framework, *LogicCrowd* is a logic-programming approach that allows programmers to create their rules via an event-driven approach and interface with social media networks to use crowdsourced data from within logic programs.

Recently, there have been explorations of crowdsourcing for the mobile environment [19-24]. Crowdsourcing via smartphones has been reviewed in [25], providing taxonomy of mobile crowdsourcing in terms of new applications and similar services based on crowd-generated data. Txtagle [26], MobileWorks [27], UbiAsk [28], CrowdITS [29] and Smart Mob [30] are crowdsourcing applications enhanced with a range of different sensors such as camera, GPS, communication signals, accelerometer and so on. However, in our platform, we develop a novel mobile application that enables declarative programming with mobile crowdsourcing through the use of social media network – sensor data can also be incorporated such as GPS locations in queries, to scope queries.

VII. Conclusion and Future Work

We have presented *LogicCrowd*, a logic-programming language for declarative mobile crowdsourcing. LogicCrowd offers a practical and principled approach for opening query evaluation to involve the crowd, and so, integrating crowd solutions with conventional machine computation. LogicCrowd basically extends the meta-interpreter of pure Prolog via Java built-ins. Due to these extensions; mobile users are able to program their own goals/rules in order to connect to (1) the crowd and (2) to available context resources such as mobile sensors, all in one language.

In our framework, we exploit social platforms for *LogicCrowd* to provide seamless access to broad sets of people, enabling the crowd to eventually receive good responses. Either with individual/public relationships or short/long–term relationships with crowds, the social media network might motivate people to provide relevant responses and at the same time improve the quality of results provided by them. Experiments with LogicCrowd prototype on social media (Facebook) demonstrated that logic-based programming can achieve effective crowdsourcing.

In future work, we will continue to create a larger variety of human crowd tasks/queries, such as comparing between pictures and answering incomplete data, and to use other social networks (Google+, etc). Also, we will attempt to extend *LogicCrowd* to query sensors with WiFi or Bluetooth interfaces that allow mobile-to-mobile connections in particular areas, enabling a mobile version of *LogicPeer* [31].

ACKNOWLEDGMENT

The first author is sponsored by the Royal Thai Government, through the Ministry of Science and Technology, Thailand.

REFERENCES

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

[2] L. Sterling, E. Shapiro, and M. Eytan, The art of Prolog vol. 94: Wiley Online Library, 1986.

[3] E. Denti, "tuProlog Manual," Sep. 2012; http://amsacta.unibo.it/3451/1/ tuprolog-guide.pdf.

[4] J. Wang, T. Kraska, M. J. Franklin, and J. Feng, "CrowdER: crowdsourcing entity resolution," Proceedings of the VLDB Endowment, vol. 5, pp. 1483-1494, 2012.

[5] E. Simperl, B. Norton, and D. Vrandecic, "Crowdsourcing Tasks in Open Query Answering," in 2012 AAAI Spring Symposium Series, 2012.

[6] X. Liu, M. Lu, B. C. Ooi, Y. Shen, S. Wu, and M. Zhang, "CDAS: a crowdsourcing data analytics system," Proc. VLDB Endow., vol. 5, pp. 1040-1051, 2012.

[7] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom, "Crowdscreen: Algorithms for filtering data with humans," in Proceedings of the 2012 international conference on Management of Data, 2012, pp. 361-372.

[8] A. Parameswaran and N. Polyzotis, "Answering Queries using Humans, Algorithms and Databases," presented at the Conference on Inovative Data Systems Research (CIDR 2011), Asilomar, 2011.

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

[10] 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.

[11] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller, "Crowdsourced databases: Query processing with people," in 5th Biennial Conference on Innovative Data Systems Research (CIDR '11) Asilomar, California, USA, 2011.

[12] A. Marcus, E. Wu, D. R. Karger, S. Madden, and R. C. Miller, "Demonstration of qurk: a query processor for human operators," presented at the Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11), 2011.

[13] A. Marcus, E. Wu, D. Karger, S. Madden, and R. Miller, "Human-powered sorts and joins," Proceedings of the VLDB Endowment, vol. 5, pp. 13-24, 2011.

[14] A. G. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom, "Deco: declarative crowdsourcing," presented at the Proceedings of the 21st ACM international conference on Information and knowledge management, Maui, Hawaii, USA, 2012.

[15] A. Morishima, N. Shinagawa, T. Mitsuishi, H. Aoki, and S. Fukusumi, "CyLog/Crowd4U: a declarative platform for complex data-centric crowdsourcing," Proc. VLDB Endow., vol. 5, pp. 1918-1921, 2012.

[16] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller, "Turkit: human computation algorithms on mechanical turk," in Proceedings of the 23nd annual ACM symposium on User interface software and technology, 2010, pp. 57-66.

[17] P. Heymann and H. Garcia-Molina, "Turkalytics: analytics for human computation," presented at the Proceedings of the 20th international conference on World wide web, Hyderabad, India, 2011.

[18] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar, "The jabberwocky programming environment for structured social computing," presented at the Proceedings of the 24th annual ACM symposium on User interface software and technology, Santa Barbara, California, USA, 2011.

[19] T. Yan, M. Marzilli, R. Holmes, D. Ganesan, and M. Corner, "mCrowd: a platform for mobile crowdsourcing," in Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, 2009, pp. 347-348.

[20] M. Stevens and E. D’Hondt, "Crowdsourcing of pollution data using smartphones," in 1st Ubiquitous Crowdsourcing Workshop at UbiComp, 2010.

[21] Y. Luon, C. Aperjis, and B. A. Huberman, "Rankr: A Mobile System for Crowdsourcing Opinions," Mobile Computing, Applications, and Services, pp. 20-31, 2012.

[22] A. Gupta, W. Thies, E. Cutrell, and R. Balakrishnan, "mClerk: enabling mobile crowdsourcing in developing regions," in Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems, 2012, pp. 1843-1852.

[23] M. Demirbas, M. A. Bayir, C. G. Akcora, Y. S. Yilmaz, and H. Ferhatosmanoglu, "Crowd-sourced sensing and collaboration using twitter," in World of Wireless Mobile and Multimedia Networks (WoWMoM), 2010 IEEE International Symposium on a, 2010, pp. 1-9.

[24] F. Alt, A. S. Shirazi, A. Schmidt, U. Kramer, and Z. Nawaz, "Location-based crowdsourcing: extending crowdsourcing to the real world," in Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries, 2010, pp. 13-22.

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

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

[27] P. Narula, P. Gutheim, D. Rolnitzky, A. Kulkarni, and B. Hartmann, "MobileWorks: A Mobile Crowdsourcing Platform for Workers at the Bottom of the Pyramid," Association for the Advancement of Artificial Intelligence, 2011.

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

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

[30] H. Rheingold. “Smart Mobs.” Perseus Publishing, 2003.

[31] S. W. Loke, "Declarative programming of integrated peer-to-peer and Web based systems: the case of Prolog," Journal of Systems and Software, vol. 79, pp. 523-536, 2006.