Declarative Programming for Mobile Crowdsourcing: Energy Considerations and Applications

Jurairat Phuttharak and Seng W. Loke

Department of Computer Science & Computer Engineering,
La Trobe Univerity, VIC, 3086, Australia
jphuttharak@students.latrobe.edu.au, s.loke@latrobe.edu.au

Abstract. This paper introduces *LogicCrowd*, a declarative programming platform for mobile crowdsourcing applications (using social media networks and peer-to-peer networks), developed as an extension of Prolog. We present a study of energy consumption characteristics for our *LogicCrowd* prototype. Based on the measurements, we develop an energy-crowdsourcing consumption model for *LogicCrowd* on the Android platform and also extend the *LogicCrowd* meta-interpreter for computing with an energy budget corresponding to a certain battery lifetime.

Keywords: declarative programming language; mobile application; mobile crowdsourcing; peer-to-peer; mobile energy consumption model

1 Introduction

Crowdsourcing is simply known as the power of the crowd [1]. There are successful crowdsourcing services in the marketplace today, including Amazon's Mechanical Turk[1], MicroWorker[2] and MicroTask[3] that has become more interesting. Those products offer a framework to access the crowd which enables the employers to submit individually designed tasks. To take advantage of the widespread mobile's opportunities for crowdsourcing, we attempt to engage social media networks and bring the crowdsourcing model into mobile environments. The contribution of this paper is an innovative approach to highlight the significance and advantages of using declarative programming language for leveraging the knowledge of people through mobile crowdsourcing contexts, and to study the energy implications of this.

We designed *LogicCrowd* (first briefly introduced in [2], and will be extended here), a declarative crowdsourcing platform for mobile applications, which combines conventional machine computation and the power of the crowd in social media networks and peer-to-peer networks. With the advantages of expressive power (ease of programming and compact code) and declarative semantics (ease of program transformation and transparent parallelism), logic programming has benefits for our framework. We integrate logic programming into a crowdsourcing mobile middleware in order to provide a declarative programming platform for mobile apps that can use crowdsourcing. In addition, given that energy is a crucial resource on

[1] https://www.mturk.com/ [2] http://microworkers.com/ [3] http://www.microtask.com/

mobiles, energy consumed when using Wi-Fi and Bluetooth communication technologies in Android phones for crowdsourcing is experimentally measured. In this paper, we investigate relationships among energy consumption, two different types of crowd execution (Synchronous and Asynchronous) and different kinds of aforementioned network connections. We also explore the relationships between energy consumption and the waiting period of crowdsourced queries; based on those measurements, we develop an energy-crowdsourcing consumption model for the Android platform and also implement the notion of *computing with an energy budget* via an extension of the *LogicCrowd* meta-interpreter.

In the following, Section 2 introduces the concept of the *LogicCrowd* platform, its architecture and several example *LogicCrowd* applications. Section 3 evaluates the prototype and proposes an energy consumption model for mobile crowdsourcing. Section 4 reviews related work, and Section 5 concludes with future work.

2 *LogicCrowd* and its Applications

LogicCrowd, at this stage, is designed for users or mobile developers who have a basic background on Prolog, though syntactic sugar and UI forms can also be used.

Crowd Predicates. Queries to underlying crowds are abstracted as predicates. 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 the inputs or conditions when asking the crowd represented by parameters. 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 is augmented by crowdsourcing. We have clauses of the form: $\mathcal{A}$:- $\mathcal{G}$, where $\mathcal{G}$ is defined by $\mathcal{G} ::= \mathcal{A} \mid \mathcal{D} \mid (\mathcal{G}, \mathcal{G})$, $\mathcal{A}$ is an atomic goal and $\mathcal{D}$ is a crowd predicate; e.g., this crowd predicate represents a person/user's query about a place, sent to the crowd:

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

The question and picture predicates are in the crowd's conditions acting as parameters for queries to the open crowd. In our work, we define the relevant crowd's 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.

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[7] which could be easily extended to accommodate calls to tuProlog 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),
        (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).
```

[7] http://apice.unibo.it/xwiki/bin/view/Tuprolog/

```
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) inserted 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 that 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.

Synchronous vs Asynchronous Execution. One of the most important issues about crowdsourcing is how to manage the answer(s) returned by the crowd. Since a delay for the crowd to provide 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 execution. 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. We support this mechanism via a small extension to the crowd predicate as shown in the following form: `<crowd_KW>?<(crowd_answer)>#[syn,crowd_condi-tions]`.The query is issued in the synchronous mode when we have the atom "syn" in the crowd conditions. In the asynchronous operation, the multi-threading capability available is exploited since *LogicCrowd* is built on top of tuProlog which integrates seamlessly with Java/Android. 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 synchronous processing, asynchronous operation would permit other goals to continue before answers from the crowd return; when the asynchronous method occurs, 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.

Design and Implementation. We have built a prototype implementation of *LogicCrowd* integrating tuProlog with Android via our own custom-built Java program called a Mediator. 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); otherwise, the crowd is queried. The architecture of *LogicCrowd* is detailed in [2].

***LogicCrowd*'s Example Applications.** We illustrate how *LogicCrowd* can be applied for the purpose of conveniences. The first scenario is the application of *LogicCrowd* to ask for the meanings or clarifications of pictures. That is, one takes pictures of the unknown things (i.e., any symbols or events) and then sends them to the crowd via *LogicCrowd* (i.e., your friends or people of the area) to find out what they exactly mean. The rule for this help can be written as below:

```
askcrowd:- picture?(_)#[asyn, asktype("photo"),question("Please translate to English?"),
           picture("instruction.jpg")askto([bluetooth]), expiry("0,30,0")].
handle_crowd_answer(picture,Instruction):- show_instruction(Instruction).
```

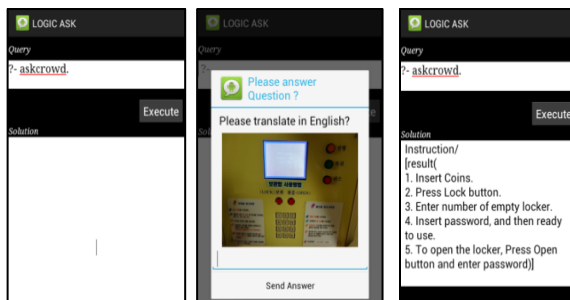
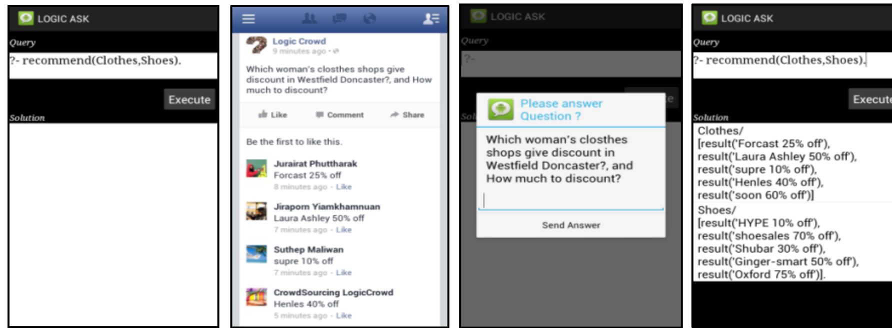


Fig. 1. Sending the requests to Facebook and Bluetooth and showing the results: (left screen) Execute Goal, (middle screen) Ask friends via Bluetooth, (right screen) Display Result.

The `asktocrowd/1` aims to ask the question by sending the photo to peers via Bluetooth connections and the crowd predicate identified using the crowd keyword "`picture`" is called as the open predicate to peers. The crowd query is in asynchronous mode with expiry in 30 minutes. With asynchronous operation, if there are goals after the crowd predicate, these goals can execute without waiting for the results from the crowd. 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 translating the instruction to English. On expiry, the result is returned to the query originator in *LogicCrowd*, where the `handle_crowd_answer/2` predicate would be automatically executed, which, in this scenario, is programmed to display the results, as shown in Figure 1(c). Instead of only Bluetooth, Facebook could also be used. In the second scenario, a *LogicCrowd* program can be applied to implement a recommendation system for a couple shopping in a big mall. In the rule below, there are two crowd predicates, the first one with crowd keyword "clothesShop" asks the crowd to recommend woman's clothes shops and the second with crowd keyword "shoeShop" asking about shoe shops, both in the Westfield Doncaster shopping center, by sending the query to friends via Facebook and Bluetooth and waiting up to 10 minutes.

```
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")].
```



(a) Execute Goal (b) Ask Question via Facebook (c)Ask Question via Bluetooth (d) Display Results

Fig. 2. Sending the requests to Facebook and to devices via Bluetooth and showing the results.

After executing the crowd sub-goal shown in Figure 2(a), the question then appears on both Facebook over the Internet, and friends' devices connected via Bluetooth, as illustrated in Figures 2(b) and 2(c). A while later, several friends' answer the question by mentioning shop(s) which offer discount. Within 10 minutes, i.e. the expiry time, the result is returned back to *LogicCrowd* in the originating device. The system then returns the results to the main goal. Figure 2(d) displays the result consisting of a list of the woman's clothes shops and a list of the shoes shops with discount details. We can extend the first scenario to be more flexible by using the asynchronous operation mode in one program, as shown below. The `recommend/1` rule is a goal to ask the crowd for recommendations on the shops in a particular shopping mall.

```
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 friends might answer the request by suggesting shop(s) that offer discount. Within 10 minutes, the result is returned to *LogicCrowd*. Then, the `handle_crowd_answer/4` predicate would be automatically executed in the first sub-goal and is programmed to first select woman's clothes shops with offers of more than 15% discount and then to 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 the shoes shops with offers of more than 30%. Figure 3(left – asyn. mode) and 3(right – syn. mode) display the final results consisting of a list of woman's clothes and shoes shop – note that the display can, of course, be pretty-formatted for the user.

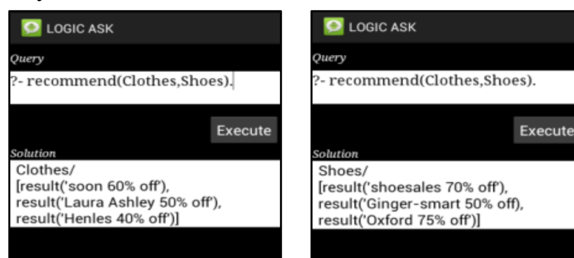


Fig. 3. The complex scenario - sending the requests to crowd and showing the results.

3 Energy Considerations in *LogicCrowd*

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 approach can be divided into four phases.

The first phase: setup. We first designed experiments to obtain two sets of measurements. In the first experiment, the aim was to compare the total power consumption for asking crowd in different network connections: WI-FI, Bluetooth and Mix (use 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 below. These rules show six main kinds of crowd predicates used in the measurements: 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 at 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 study 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*.

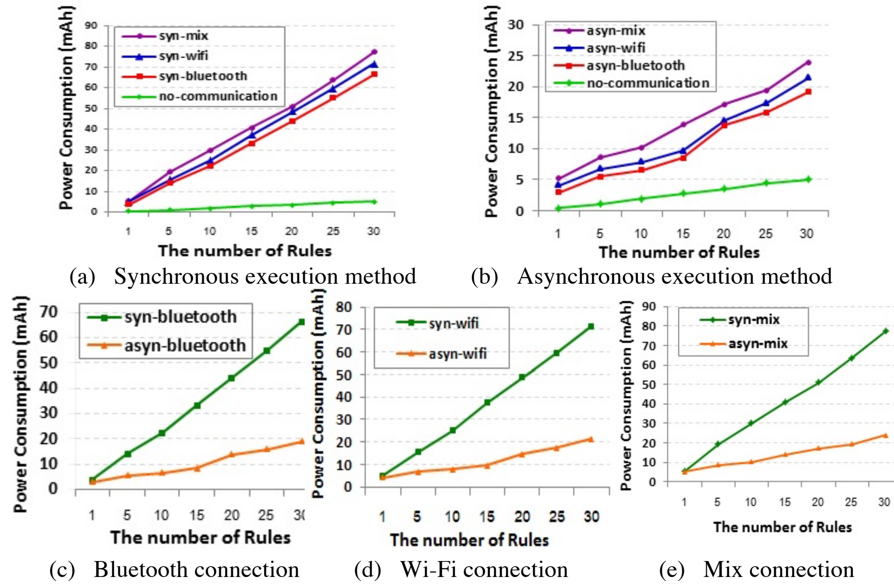
```
syn_bt:- findall(B,(thai(B),melbourne(B)),A),nice?(Ans)#[syn,asktype('choice'),question
     ('Which restaurant do you recommend?'),options(A),askto([bluetooth]),expiry('0,0,60')].
syn_wf:- findall(B,(thai(B),melbourne(B)),A),nice?(Ans)#[syn,asktype('choice'),question
     ('Which restaurant do you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

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 had been applied in this case. The expiry times in the crowd predicate had been changed by increasing in 5 minute intervals up to 20 minutes. This experiment leads up to the follow-up third phase of the study where we could 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 Android operating system version 4.1.2, Jelly Bean. A Power tool called Little Eye[8] was utilized for capturing the battery level and also for monitoring the power consumption of the *LogicCrowd* program.

The second phase: measurements. Figure 4(a)-(e) shows 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 has been shown in a way that all graphs display similar trends - a linear increase, which is more scalable than an exponential increase.

---

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



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

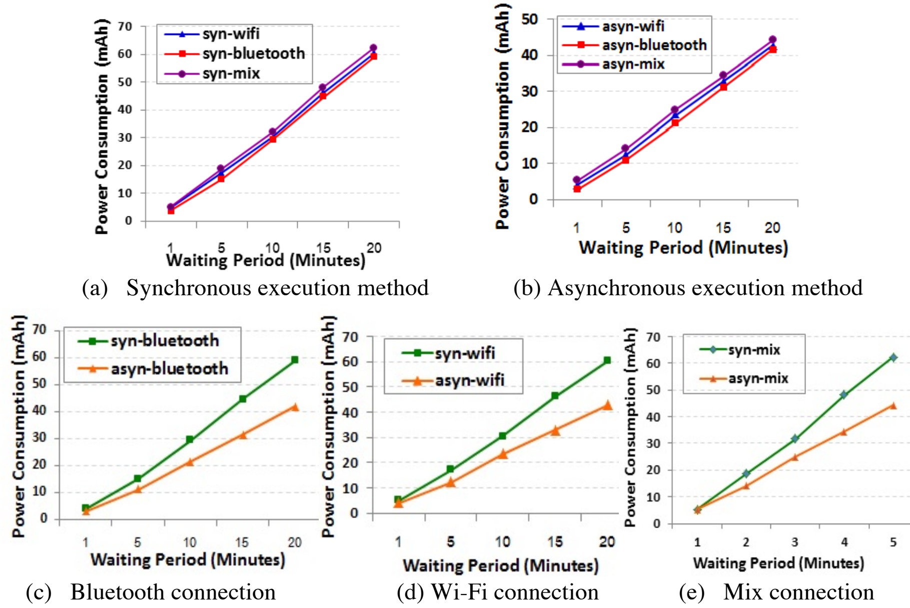
(c) Bluetooth connection (d) Wi-Fi connection (e) Mix connection

Fig. 4. 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 power consumption (milliamps per hour).

As shown in Figure 4(a), asking the crowd by using Mix connection with Synchronous mode consumes the most energy, accounting for 1.11 times of the total power consumed for Wi-Fi, 1.27 times for Bluetooth, 15 times that of "no-communication". For each rule using a crowd predicate call with Mix connection, the average power consumption is spent differently: 4.872 mAh (milliamps per hour) on display (keeping the *LogicCrowd* front screen up), 0.378 mAh on CPU, and 0.018 mAh 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 with the Synchronous method consumes around 1.14 times marginally more power than crowd asking using the Bluetooth connection with the same method, and spends about 13.46 times significantly more energy than using the rules. In each rule with this Wi-Fi mode, the average power consumed for display, CPU, and Wi-Fi varies: 4.674 mAh, 0.265 mAh, and 0.018 mAh respectively. Also, crowd asking using Bluetooth with synchronous execution consumes around 11.93 times of the total energy compared to rules without communication with the crowd - with no crowd goals, 3.396 mAh of average power per rule is used for display and 0.281 mAh for CPU. In compassion with no-communication mode, we can see that Wi-Fi, Bluetooth, and Mix all consume much higher energy. This is because with the use of the crowd predicate in crowd-communicative modes, it takes time to wait for crowd's responses. The more rules used, the longer the waiting period takes, and the higher the energy consumed. Most energy has been found to be mostly used for display, accounted for 70-90% of the total spent on the foreground when asking the crowd (an optimization this suggests is to wait for the crowd answers in the background). Figure 4(b) shows the differences in power consumption when using Wi-Fi, Bluetooth and Mix

connections with Asynchronous mode, where the overall result is found similar to that of Synchronous operation. 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 Wi-Fi and Bluetooth connection with the same method. Its energy spent 6.360 times more than the amount spent using the no crowd communication rules. Here, for each rule with Mix connection, 4.676 mAh of the mean power is spent on display, 0.489 mAh on CPU, and 0.018 mAh on Wi-Fi. While, the total power consumed when using Wi-Fi connection is around 1.17 times higher than the power consumed when using Bluetooth connection under the same method. Still, the amount of energy Wi-Fi spent 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 3.740 mAh, 0.336 mAh, and 0.015 mAh respectively. In addition, using Bluetooth with Asynchronous execution consumes around 4.26 times more energy than "no-communication" with the crowd. Figure 4(c)-(e) presents the energy consumption of different operation modes (Synchronous and Asynchronous) when the rules have been increased with equivalent interval (every 5 rules). The result shows that Synchronous execution remarkably consumes more energy than execution with Asynchronous mode. Also, 20 executed rules with Synchronous-Bluetooth mode consume 44.1 mAh whereas the Asynchronous-Bluetooth mode with the same size of rules spends only 13.8 mAh. In this regard, we can say that the average of energy consumed in Synchronous execution with Bluetooth, Wi-Fi, and Mix connection is three times higher than the average of the 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 crowd in the *LogicCrowd* program. Figure 5 shows the relationship between the 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 shown in that all graphs display similar trends, the regression line slopes upwards. The longer the waiting period, the more energy consumed. In Figure 5(a), the comparison of energy consumption among Wi-Fi, Bluetooth and Mix connection in 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; the energy consumption of a rule using Wi-Fi is approximately 1.11 times slightly higher than that using Bluetooth connection. The similar trend is shown in Figure 5(b) in which the average power is 1.14 times more likely to be consumed by Wi-Fi than by Bluetooth connection' with Asynchronous operation. 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 spends the most energy, whereas Bluetooth consumes the least. 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 5(c), 5(d) and 5(e) shows the energy consumption between Synchronous and Asynchronous execution. With increasing waiting period in every 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 simple linear energy consumption models for each type of execution modes and communication technologies. Those models will be explained next.



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

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

Fig. 5. The relationship between energy consumption and waiting period/expiry in different types of execution methods and communication technologies, x-axis is waiting period (Minutes); y-axis is power consumption (milliamps per hour).

The third phase: energy consumption models. Based on the measurements above, we constructed a simple (linear) energy consumption model. Table 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. It means that these functions can estimate the power (denoted by $\boldsymbol{y}$), which is consumed when using a particular communication technology with a particular execution method for a waiting period of T minutes.

Table 1. Energy consumption functions per rule with respect to the waiting period when using Bluetooth, Wi-Fi and Mix communication in different execution methods.

Connections	*Synchronous Execution*	*Asynchronous Execution*
Wi-Fi	$y = 2.905\ T + 2.150$	$y = 2.038\ T + 2.321$
Bluetooth	$y = 2.922\ T + 0.483$	$y = 2.040\ T + 0.855$
Mix	$y = 2.991\ T + 2.748$	$y = 2.047\ T + 3.652$

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 2. Synchronous functions from Table 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, Asynchronous execution method is running 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 2. This function can be used to calculate the power of battery (i.e. $\boldsymbol{y}$) which is consumed when using $\mathbf{N}_{\mathrm{WF}}$ rules of Wi-Fi, $\mathbf{N}_{\mathrm{BT}}$ rules of Bluetooth and $\mathbf{N}_{\mathrm{MIX}}$ rules of Mix connections for a waiting period of $\mathbf{T}_{\mathrm{MAX}}$ 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 1 and Table 2. For instance, if we execute the rule which the expiry crowd condition is 30 minutes by using Wi-Fi with Synchronous mode, according to Table 1 and the following function $y = 2.905\ T + 2.150$, for T = 30, we spend 89.3 mAh of the battery. Measurements in a real world environment showed that the rule consumed around 88.821 mAh of energy, which is similar to the energy consumption calculated by using the functions from Table 1.

Table 2. Energy consumption functions with respect to a group of crowd predicates

Execution Modes	*Connections*	*Functions*
Synchronous	*Wi-Fi*	$\boldsymbol{y = \sum_{i=1}^{N}(2.905\,T_i + 2.150)}$
	Bluetooth	$\boldsymbol{y = \sum_{i=1}^{N}(2.922\,T_i + 0.483)}$
	Mix	$\boldsymbol{y = \sum_{i=1}^{N}(2.991\,T_i + 2.748)}$
Asynchronous		$\boldsymbol{y = 0.537\ N_{BT} + 0.562\ N_{WF} + 0.754\ N_{MIX} + 2.01\ T_{MAX} - 0.446}$

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 2 and the following function $y = 0.537\ N_{BT} + 0.562\ N_{WF} + 0.754\ N_{MIX} + 2.01\ T_{MAX} - 0.446$, for $N_{BT} = 1$, $N_{WF} = 2$, $N_{MIX} = 4$, $T_{MAX} = 5$, the power consumed is 14.256 mAh. Real measurements showed the rule consumed around 14.375 mAh; which is not much different from the energy consumption estimated from the functions in Table 2.

The fourth phase: extensions to the *LogicCrowd* metainterpreter. The proposed models in the 3rd 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 estimated power per crowd goal/rule for each network connection with a particular execution method is estimated via Table 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 is six different aspects of querying the crowd (as mentioned in Table 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* β (%) is a user's policy of energy usage allowed for the *LogicCrowd* programs, i.e., a percentage of $E_{current}$. To manage energy consumption of *LogicCrowd*

applications and enhance battery performance, we use the condition given as follows: $E_{crowd(i)}(t_i) \leq \beta(\%) \times E_{current}$. Then, 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 battery power $E_{current}$ is 1200 mAh. When evaluating a rule with a crowd predicate under Synchronous execution using Mix network connection with a one hour waiting time, 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 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**, thereby** manag**ing** the energy spent on the application. This algorithm and the modified rules in the meta-interpreter are as shown below.

Define G :- $g_1, g_2, g_i, \ldots, g_n$ 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

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

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

4 Related Work

CrowdDB [3] and Deco [4] 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. Crowd4U [5] leveraged a declarative platform for database abstraction by extending CyLog to issue open queries to the crowd. 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 and interfaces with social media and peer-to-peer networks in order to use crowdsourced data within logic programs. Balasubramanian *et al.* [6] measured energy consumption when using GSM, 3G, and Wi-Fi, showing that 3G and WiFi have high tail energy use at the end of data transfer. Xiao *et al.* [7] investigated energy consumed in mobile applications for video streaming, reporting that Wi-Fi used energy more efficiently than 3G. The work on VoIP applications over Wi-Fi based mobile phones [8] showed that using a power saving mode in Wi-Fi accompanied with intelligent scanning techniques for networks can reduce consumed energy. Our focus is, however, on the development of novel energy-efficient algorithms for managing energy consumption in *LogicCrowd* applications that use Wi-Fi and Bluetooth.

5 Conclusion

We have presented *LogicCrowd*, a logic-programming language for declarative mobile crowdsourcing, providing a practical and principled approach for query evaluation to involve the crowd. We also conducted an energy analysis of *LogicCrowd* programs, using the findings for strategically selecting the most appropriate energy-related conditions for execution, and to design a simple energy consumption model for *LogicCrowd* on Android phones. We showed how to manage the energy consumption of *LogicCrowd* programs by implementing the notion of computing with an energy budget via an extension of the *LogicCrowd* meta-interpreter.

References

[1] Howe, J.: The rise of crowdsourcing. Wired magazine 14, 1--4 (2006)

[2] Phuttharak, J., Loke, S.W.: LogicCrowd: a Declarative Programming Platform for Mobile Crowdsourcing. Proc. of The 12th IEEE International Conference on Ubiquitous Computing and Communications (IUCC-2013). IEEE, Melbourne, Australia (2013)

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

[4] Parameswaran, A.G., Park, H., Garcia-Molina, H., Polyzotis, N., Widom, J.: Deco: declarative crowdsourcing. Proc. of the 21st ACM international conference on Information and knowledge management, pp. 1203--1212. ACM, Hawaii, USA (2012)

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

[6] Balasubramanian, N., Balasubramanian, A., Venkataramani, A.: Energy consumption in mobile phones: a measurement study and implications for network applications. Proc. of the 9th ACM SIGCOMM conference on Internet measurement, pp. 280--293. (2009)

[7] Yu, X., Kalyanaraman, R.S., Yla-Jaaski, A.: Energy Consumption of Mobile YouTube: Quantitative Measurement and Analysis. Proc. of the 2nd International Conf. on Next Generation Mobile Applications, Services and Technologies, pp. 61--69. (2008)

[8] Gupta, A., Mohapatra, P.: Energy Consumption and Conservation in WiFi Based Phones: A Measurement-Based Study. Proc. of the 4th Annual IEEE Comms. Society Conf. on Sensor, Mesh and Ad Hoc Communications and Networks, pp. 122--131. (2007)