u-Help: supporting helpful communities with information technology*

Andrew Koster[1], Jordi Madrenas[1], Nardine Osman[1], Marco Schorlemmer[1], Jordi Sabater-Mir[1], Carles Sierra[1], Angela Fabregues[1,2], Dave de Jonge[1,2], Josep Puyol-Gruart[1,2], and Pere Garcia-Calvés[1]

[1] IIIA-CSIC
Bellaterra, Spain
[2] Universitat Autònoma de Barcelona
Bellaterra, Spain

Abstract. When people need help with day-to-day tasks they turn to family, friends or neighbours to help them out. Finding someone to help out can be a stressful waste of time. Despite an increasingly networked world, technology falls short in supporting such daily irritations. **u-Help** provides a platform for building a community of helpful people and supports them in finding help for day-to-day tasks. It relies on a trio of techniques that allow a requester and volunteer to find one another easily, and build up a community around such provision of services. First, we use an ontology to distinguish between the various tasks that **u-Help** allows people to provide. Second, a computational trust model is used to aggregate feedback from community members and allows people to discover who are good or bad at performing the various tasks. Last, a flooding algorithm, similar to the popular Gnutella algorithm, quickly disseminates requests for help through the community. This paper describes these three techniques in detail. This work is implemented as an iPhone application that we also describe in this paper.

1 Introduction

The web has been evolving into a social space enabling individuals to be pulled opportunistically into peer communities to achieve both personal and group goals. This has been possible due to the widespread adoption of software applications that support and facilitate basic group interaction and collaboration such as file sharing, instant messaging, blogging, or social networking. These sorts of applications depend more on social conventions that arise within the community that uses them than on the particular features offered in the first place.

But, despite the success of certain well-designed social software platforms such as Flickr, MySpace, LinkedIn, Facebook, or Twitter, current social applications do not allow a community to form around a specific need. Particularly, users who are looking for either new, partially formed, or already standing communities whose interactions are much more domain specific and specialised, need software tools and platforms that support the formation of and the adjustment to the evolving community.

* AT2012, 15-16 October 2012, Dubrovnik, Croatia.

An example of such a community, is that of a group of neighbours, friends or close family who turn to each other for help with performing day-to-day tasks. According to Beech et al. [1] being late to pick up the children is a major stress factor for working parents. From one of the interviews they conducted, they quote a mother: "the worst time is the afternoons, and trying to finish off work to leave on time to collect my son from the nursery." A follow-up study [12] indicates that one of the most severe problems that working parents encounter is coping with unexpected scheduling issues. We use this community as a prototype scenario, but emphasise that there are many other communities in which similar technologies can be used. For instance, sport schools where people try to find practice partners, or a group of commuters organizing carpools. The core characteristic is not so much the activity, but the supporting of a community, with its own set of rules and expectations.

In this paper we present **u-Help**, a software application that provides a fully distributed platform for building and maintaining a local community of people helping each other with their day-to-day tasks. The aim of **u-Help** is twofold. The first is straightforward: we aim to provide an application that finds a trustworthy member of the community who will help with the required task. The second aim is longer term: as the community evolves, it may have different rules and requirements. As such, maybe the tasks to be performed change, the way in which members' actions are evaluated change. The aim is to provide the tools to change the **u-Help** application together with the users. In this paper we focus exclusively on the first aim: we describe how the **u-Help** application supports a helpful community.

u-Help starts from the pre-existing social relationships between users. We thus assume the existence of a social network (identities and connections may be imported from various other social networking applications). However, the social network is used specifically to allow users to ask for help with picking up, or taking care of their children. To allow such a request to propagate quickly through the network, we use a flooding algorithm. After a volunteer has been found and performs the task, the **u-Help** application allows the requester to evaluate that volunteer's performance. Such evaluations are aggregated in a computational trust model. Finally, we rely on semantic techniques to describe the tasks formally and define a similarity measure between them, in order to use evaluations of a volunteer performing one task to estimate his performance at another task.

2 The u-Help Application

The **u-Help** application provides a platform to build and support a community to which members can turn to find help with day-to-day tasks. In this section we describe the three key technologies used in developing **u-Help**. Underlying these technologies is the existing social network data. The users of **u-Help** are represented as nodes in a graph and social relationships between the users are the edges of the graph. Such a representation could be imported from one of a number of social networking applications, such as Facebook, or constructed by automatically by accessing the address book of people's telephones when they join **u-Help**.

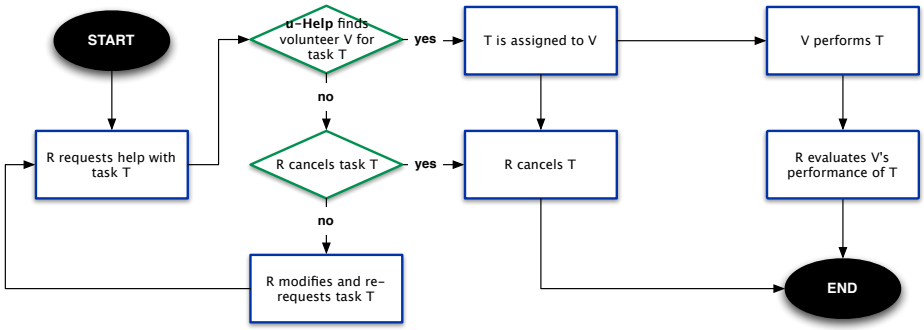


Fig. 1: Diagram of the general workflow for finding help with the **u-Help** application.

When a member of the community needs help for performing a certain task, the *flooding algorithm* propagates this request, starting with that member's direct neighbours in the graph and flooding the request out from there until a satisfactory volunteer for the task is found. The workflow of this process is drawn in the diagram Figure 1. In order to find a volunteer, the **u-Help** application has a number of components, outlined in Figure 2. As seen in this diagram, the flooding algorithm relies on the community member's trust evaluations of one another to decide whether to forward the request or not. Such trust evaluations are calculated automatically by the *computational trust model* based on requesters' evaluations of the performance of the volunteer, in the specific task requested. The trust model, in turn, relies on the *semantics of the task* to find similar tasks, when little to no feedback is available for a task.

We describe these three technologies below. In Section 2.2 we describe how users' ratings of specific tasks can be aggregated to obtain a trust evaluations and in Section 2.3 we describe the flooding algorithm and we start, in the next section, with a full description of the tasks for which **u-Help** can be used and the semantics of these tasks.

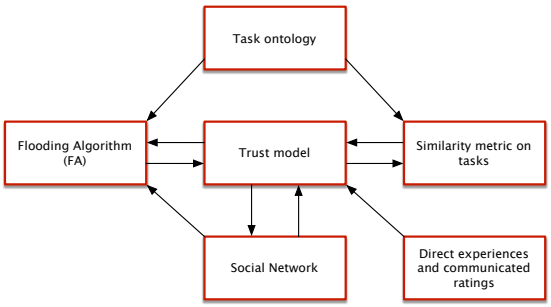


Fig. 2: Abstract overview of the **u-Help** application's components

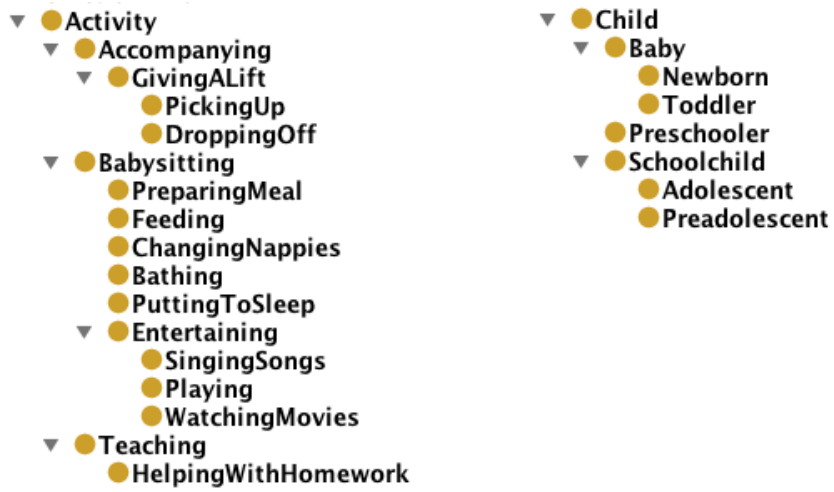
2.1 Tasks

We shall assume that the tasks managed by the u-Help community consist of the *activity* the task is about (babysitting, picking up, cooking, etc.) and the *child*

to whom the activity is applied (baby, toddler, preschooler, etc.). Consequently we need to handle two separate hierarchies.

On one hand, the different kinds of activities are usually organized in a meronomy[1], specifying which subactivities may be part of a given activity (e.g., that changing nappies is a subactivity of babysitting). Here we take inspiration from the design decisions suggested by the Process Specification Language (PSL) [7]. The activity meronomy is given in Figure 3a.

On the other hand, the different kinds of children are more naturally organized in a taxonomy, specifying subclasses of children (e.g., that a toddler is a baby). To build the child class hierarchy, we have been looking at WordNet [5] for inspiration. WordNet has situated two of the different senses of *child* in two different fragments of its taxonomy, but for u-Help we decided to merge these two in one taxonomy, as we are treating *child* both as an offspring and a juvenile person. The child taxonomy is drawn in Figure 3b.



(a) Meronomy of activities in **u-Help** (b) Taxonomy of children in **u-Help**

Fig. 3: Formal description of the tasks, to perform some activity with a child.

A task is a pairing of an activity with a child, for instance ($GivingALift$, $Schoolchild$) or ($Playing, Toddler$). If the activity specified is not a leaf of the meronomy, the requester expects the volunteer to perform all the subactivities involved. For instance, if the activity is *GivingALift*, the volunteer is expected to perform both *PickingUp* and *DroppingOff*. However, if the child is not a leaf of the taxonomy it is simply not as detailed a specification: the schoolchild is either a preadolescent or an adolescent, but clearly not both.

This gives the requester the freedom to specify a task at a more, or less, detailed level, which can affect how the flooding algorithm propagates the request. Specifically, the specification of the task has a direct effect on the trust model, because any evaluation of a volunteer's performance is linked to the task he

[1] A meronomy is a type of hierarchy that deals with part-whole relationships, in contrast to a taxonomy whose categorisation is based on discrete sets.

performed. When evaluating a target for the performance of another task, the *similarity* between the new task and previously performed tasks is an important factor to take into account for estimating his performance. The similarity is computed separately for the activity and child types and combined in the trust model. The "similarity" between activities is considered purely from a trust perspective. We use the OpinioNet algorithm [11] to propagate evaluations of other activities through the meronomy. The similarity over the child taxonomy is considered from a more semantic perspective and we use the similarity measure proposed by Li et al. [10] to calculate this similarity.

In OpinioNet, opinions may be assigned by users to nodes of a structural graph, and the OpinioNet algorithm gives a method for propagating these opinions throughout the graph [11]. In **u-Help** the opinions are satisfaction ratings of how a user performed at an activity and the structural graph is the meronomy of those activities. We argue that a user's performance at one activity should influence his trustworthiness for performing similar activities (nearby nodes in the meronomy). We make use of the OpinioNet algorithm for the propagation of trust in graphs based on the part-of relation.

The basic idea of the OpinioNet algorithm is that if a node in the graph does not receive a direct evaluation, then its evaluation may be deduced from its children nodes' evaluations. This is because the parent node is structurally composed of its children nodes. Hence, the evaluations on children nodes must necessarily influence the deduced evaluation on a parent node. OpinioNet refers to the direct evaluation on a node or an evaluation of it that is deduced from evaluations of the parts that compose it as the 'intrinsic evaluation' of that node.

Additionally, an 'extrinsic evaluation' is an evaluation that is propagated down from parent nodes to children. In the absence of information about the node itself, or the parts that compose it, information may be inherited from what one belongs to. In other words, in the absence of information about intrinsic evaluation, the evaluation of that node is calculated based on evaluations of its parents' nodes.

As an example of how the OpinioNet algorithm propagates evaluations through the meronomy, consider that if the target performed well at the PreparingMeal activity, this will affect the evaluation of that target for the activity of BabySitting. This is an 'intrinsic evaluation'. An 'extrinsic evaluation' would be to say that performing well at BabySitting carries over to a good evaluation for the subactivity of ChangingNappies if there is no direct evaluation of the target performing the ChangingNappies activity.

However, we also argue that many of the activities change, dependent on the type of child. For instance, preparing a meal for a baby is very different to preparing a meal for an adolescent. We therefore need to take the similarity between children into account as well. For this, we rely on the similarity measure proposed by Li et al. [10], which takes three aspects of taxonomies into account:

- the distance l between two classes in the taxonomy: the closer, the more similar the classes are;

– the depth h in the taxonomy of most specific subsumer: the deeper in the taxonomy the subsumer is, the more similar the classes are;
– the local semantic density d of instances of these classes: the greater the information content of the subsumer, the more similar the classes are.

These three measures are combined in a set of equations that calculates the similarity between two classes in a taxonomy: in our case the similarity between two types of children.

In the trust model of the next section we combine the two similarity measures for tasks. Let t_1 and t_2 be tasks, then $sim(t_1, t_2)$ is the similarity (based on the taxonomy) between the children with who an activity is performed, while $prop(e, t_1, t_2)$ propagates an evaluation e of the activity performed in t_1 to the activity in t_2.

2.2 Trust Model for u-Help

Every member of the community maintains his own trust evaluations of other members of the community. These trust evaluations are task-dependent, with which we mean that a user's evaluation of another may change, depending on the task for which he is evaluated. This trust evaluation is crucial in the functioning of the flooding algorithm, which only forwards a request for help to a neighbour if the trust level in that neighbour is sufficiently high. This particular use of trust places a number of requirements on how we treat trust evaluations:

– We assume a trust evaluation is a numerical evaluation of an agent between zero and one, where zero is the equivalent of completely untrustworthy, one is the equivalent of fully trustworthy and anything in between is a degree of trustworthiness. An alternative interpretation of this is the (estimated) probability that the target of the evaluation will perform the task in a satisfactory manner.
– We assume that these evaluations have a transitive property: if an agent forwards the request to one of his own neighbours, the trustworthiness of that neighbour in the requested task to the original requester depends both on the trustworthiness of the first neighbour, as well as that person's trust evaluation of the new target. We assume trust is monotonically decreasing along a single path. This can be modeled using a T-norm and we use multiplication to model this. a's trust in c along the path $a \rightarrow b \rightarrow c$ is $trust(b, c) \cdot trust(a, b)$, where $trust(a, b)$ is the function that returns a's trust evaluation of b. Note that $trust$ is not strictly transitive and we cannot say anything about $trust(a, c)$ based on this property: we only use this for propagating trust through the network.
– We assume that if a node receives the same request two times that its trustworthiness can increase with regards to that request. This increase in trust can be modeled by a T-conorm. The requests reaching a node along two different paths are not independent events, however, because the paths taken may be largely similar and only differ in one or two nodes. We thus cannot use probabilistic disjunction, and instead choose to use the maximum of the trustworthiness along both paths.

These assumptions are not uncommon in the trust literature. For instance, TidalTrust [6] is a trust model that propagates trust evaluations through a social network relying on similar properties, although a more sophisticated algorithm is used to compute the trustworthiness of a node. For a more in-depth discussion on transitivity of trust and considerations that should be taken into account when propagating trust through a network, see [8,4]. Specifically we are assuming, in this case, that if agent a trusts agent b to perform a task, a also trusts b to delegate that task. Additionally, if c was previously unknown, then a's trust in b can somehow be seen as an upper bound of the trust a has in c for performing this task. Consider the case of picking up a child from school: if we trust b to pick up our child, and b trusts c, we will definitely not put more trust in c than in b (and in reality, probably considerably less). However, on b's recommendation, we may be willing to give c a try (especially when nobody else is available). Additionally, if c comes recommended by multiple people that we trust, then we are more inclined to trust c and accept him to perform the task.

The flooding algorithm calls the **Trust** function, which is a call to the trust model. Because **u-Help** is a distributed application and every member of the community has his own **u-Help** application, the trust evaluations are different from user to user. Every **u-Help** application comes equipped with the same trust model, but the evaluations it uses to compute evaluations are different. When the flooding algorithm calls the trust model, it calculates the trustworthiness of a target with regards to a specific task. The two main problems, inherent in **u-Help**, are that:

1. Trust evaluations are task-specific and the target may not ever have performed the required task before.
2. There may be factors outside of the **u-Help** platform that cause people to gain, or lose trust, in one another.

For the former, we take the similarity between tasks into account and allow trust to carry over, to a certain extent, to similar tasks. As explained in Section 2.1, this similarity is measured along a number of dimensions, in order to get an accurate measurement of the conceptual difference between performing one task and another. In this way, evaluations of past experiences with one user for a similar task can be used to evaluate that user for a task he has never performed.

Additionally, some tasks may be more sensitive than others and require more trustworthy agents to perform them. Therefore we propose that users may specify a minimum trust level for each task. This allows users to, for example, specify that to pick up his children, the agent must be trusted with a value of at least 0.8, while for fetching a coffee from the cafe a trust level of 0.4 is sufficient.

Evaluating direct experiences. When a task is performed, the requester can evaluate its performance by giving a numerical rating between zero and one. A rating of zero means that the task was performed in a very unsatisfactory manner and an evaluation of one means the task was performed perfectly. When a trust evaluation is needed for a new task, these ratings are aggregated together. Let t be the new task, v be a volunteer and S be the set of past satisfaction ratings

obtained thus far from direct experiences with v. Additionally, let $value, task$ and $time$ be functions such that, for a rating s, $value(s)$ is the numerical evaluation, $task(s)$ the task and $time(s)$ the time at which the task took place.

First we select those ratings that are semantically near enough to be considered. In other words $S_t = \{s \in S | sim(t, task(s)) > \delta\}$, with δ a threshold for the distance between tasks. The direct trust is computed as follows:

$$Trust(v,t) = \frac{default_v + \sum_{s \in S_t} sim(t, task(s)) \cdot prop(value(s), task(s), t) \cdot \lambda^{now - time(s)}}{1 + \sum_{s \in S_t} sim(t, task(s)) \cdot \lambda^{now - time(s)}}$$

Where λ is a parameter for discounting outcomes of past tasks over time, $default_v$ is the default trust value of volunteer v (as specified by the user) and now is the current time.

This formula calculates the direct trust as a weighted average of the ratings, after being propagated over the activity meronomy using the OpinioNet algorithm, with the similarity between child types and discount factor over time as weights. The default value loses importance the more (recent and similar) ratings there are available. When there are few, or no direct experiences the direct trust evaluation is near the default trust value, but as more ratings get taken into account, the influence of the default value diminishes.

Communicated experiences. When a volunteer is not a direct neighbour of the requester, the volunteer was found by the flooding algorithm along a path of trusted delegators. For community building it is important to disseminate information about the performance of the members of the community quickly [2], in order to ensure good performers are kept in the loop, while people who perform badly at certain tasks are not asked to perform those tasks again.

Specifically, the direct neighbours of the volunteer must be told about the volunteer's performance, because they are, ultimately, the ones who recommend that volunteer to other members of the community. All evaluations of a task are therefore sent to all neighbours of the volunteer. If there is a central infrastructure, this is easy to accomplish, but because the **u-Help** platform is fully distributed, we cannot assume the structure of the social network is known to the requester. The **u-Help** application thus requires the nodes to always forward messages reliably. That way the requester's **u-Help** client can send the message to the volunteer's client, which is guaranteed to forward this message to all its neighbours. In this sense the **u-Help** client should be seen as a governor agent [3]: it mediates the participation of the user, but is not fully under that user's control. We do not consider security issues, such as falsifying messages, but these could be dealt with by using encryption.

The receiving client deals with received outcomes as if they are from its own user and simply adds them to the set of outcomes it considers when it evaluates trust. This ignores some of the issues of trust, such as that an outcome is a subjective evaluation, taking the user's own criteria into account and may not be immediately applicable for other users. To deal with this, we propose to extend the **u-Help** trust model in the future with a trust alignment module [9], but for the moment we assume other users' evaluations of the performance of a task are directly applicable.

2.3 Flooding Algorithm

The flooding algorithm is the core computational process in the **u-Help** platform. It ensures requests for help are disseminated through the community. As stated earlier, we build upon a social network representation of the community: this is represented by some graph, in which the members of the community are nodes, and the edges represent friendship relations between two people. When someone wants to request help with a specific task, the flooding algorithm sends this request to that person's neighbours in the graph (i.e. its friends) and from there it continues to flood through the network. This is similar to a number of other algorithms designed for rapidly disseminating a message through a graph, most prominently the Gnutella algorithm for P2P file sharing. The main difference between existing approaches and the flooding algorithm discussed here is that the decision to stop forwarding the request is made primarily based on trust, rather than on other things, like the number of hops the request has made through the network or the time passed since the initial request was made.

The reason for this is that we not only want to find someone willing to volunteer for a task, but he must also be trustworthy when performing this task. The way we calculate trust is described in the previous section. With the trust evaluation, the flooding algorithm can then propagate a request through the network. For calculating trust along a path, we recall that we assume trust is transitive, and thus we can use a T-norm to ensure trust is monotonically decreasing along a path. As stated in the previous section, we use multiplication. The flooding algorithm stops propagating the request when the cumulative trustworthiness of a node falls below a certain threshold τ. Algorithm 1 gives the pseudocode for the algorithm that performs this flooding.

The algorithm works in a straightforward manner: every time a friend's request is received (in Algorithm 1 this is simply a call to the **FloodRequest** function on the corresponding node), the algorithm propagates the request to neighbouring nodes only if the request is either a new request ($task \notin MyTasks$), or the trust level of the path requesting this is sufficiently higher than the trust level of the previous path that led to the same request (where the decision of whether a trust level is 'sufficiently higher' is determined by the minimum acceptable change in trust level σ).

When a user wants to request help for task t, the software calls **FloodRequest**$(t, 1, \emptyset)$. The algorithm floods the request through the network. The **Confirm** method sends a message back to the original requester when the human user associated with the node in question accepts a request (**HumanUserAccepts**). The **u-Help** platform then filters out possible bad matches by using the agent's own trust model to check for volunteers below the threshold τ (the minimal accepted trust level), and a choice is made out of all eligible volunteers. There are a number of options for performing this choice. In general this is up to the user, but he may delegate this to **u-Help**, that could automatically select the most trustworthy volunteer, which is outside the scope of this paper.

Concerning efficiency, we note that the flooding algorithm is not affected by loops in the graph as flooding is stopped when the node itself is found in the re-

quest path (due to the condtion on line 2 of Algorithm 1). In the worst case, with $\sigma = 0$, for a graph G, the number of messages is $\sum_{p \in loop\_free\_paths(G)} length(p)$, which can be exponential in the number of nodes.

Algorithm 1: The Flooding Algorithm

Input: $\tau : [0, 1]$, minimum trust level
Input: $\sigma : [0, 1]$, minimum change in trust to re-flood the network
Input: $me : Node$, the identifier of the current node
Input: $friends : 2^{Node}$, set of neighouring nodes
Data: $MyTasks := \emptyset$, a list of tasks that have been requested
Data: $OldTrust := \emptyset \rightarrow \emptyset$, a map with the current trustworthiness for every task requested

```
1  FloodRequest(task, trustlevel, path): begin
2      if me ∉ path then
3          if task ∉ MyTasks then
4              MyTasks := MyTasks ∪ task
5              OldTrust(task) := trustlevel
6              Propagate(task, trustlevel, path ⊕ me)
7          end
8          else if trustlevel − OldTrust(task) > σ then
9              OldTrust(task) := trustlevel
10             Propagate(task, trustlevel, path ⊕ me)
11         end
12     end
13 end
14
15 Propagate(task, trustlevel, path): begin
16     if HumanUserAccepts(task) then
17         Confirm(task, path)
18     end
19     foreach Node n ∈ friends do
20         trust := Trust(n, task) · trustlevel
21         if trust > τ then
22             n → FloodRequest(task, trust, path)
23         end
24     end
25 end
```

Finally, we note that, if needed, the algorithm could be changed to put a limit on the number of hops by taking a maximum number of hops into account, in addition to considering the trust level. This can be achieved by adding a parameter n in Algorithm 1: the maximum number of allowed hops, and changing the **Propagate** method to no longer propagate if the number of hops in a request message is equal to n. This would make our algorithm more similar to the Gnutella flooding algorithm, but it would still take trust into account.

Because **u-Help** is designed to run on mobile devices, it is important to take breaks in connectivity into account. In the implemented version of the algorithm, there is the option to use a time limit, after which it is assumed that any node that has not answered was unreachable. This allows the algorithm to deal with

users whose cellphones are switched off, or unreachable for other reasons. Additionally, at a lower level, the messaging protocol could detect failures to receive and resend the message when the receiving node is back online.

3 Implementation and deployment

In this section we describe the main implementation details of the **u-Help** application. The implementation was designed for iOS, and is useable on the iPod Touch, the iPhone, and the iPad. Other possible platforms, such as Android, were taken into account, but the widespread use of iOS gave the breakthrough.

In order to benefit from **u-Help** it is necessary for the user to configure her data. She must set up (or import) her connections in the social network, and have all settings and data entered correctly. Once this is done, the user can start to use **u-Help** to request help, and be contacted by others in the community who need help in return.

The **u-Help** application's usability is designed around four main views:

- **The "Help" view:** This is where users may request help.
- **The "Requests" view:** This is the place for users to track, and manage, their requests for help.
- **The "Duties" view:** This is the place for users to track, and manage, their duties, or the tasks that they have volunteered to do.
- **The "Settings" view:** This view allows the user to change all user-specific settings, as well as add, edit and remove information in the **u-Help** application. For instance, the names of the user's children and the locations they may need to be picked up from, or brought to. Additionally the user may set a number of parameters, such as the minimum acceptable trust level for each task, the maximum number of hops in the flooding algorithm and other parameters discussed in Section 2.

In what follows, we go over each of the **u-Help** application views in detail.

3.1 Asking for help

Asking for help is done by pressing the "Find Help!" button (see Figure 4). However, the user must first choose the action that she requires help with from the list of predefined tasks. This list, and the options given, correspond to the task ontology in Section 2.1.

In some cases, actions may require additional parameters for the user to fill in. For example, in the case of picking up a child, the user will need to specify: (1) which child is being picked up, (2) the location where the child needs to be picked up from, and (3) on what day and time should the child be picked up. However, different actions will require different parameters. Hence, to keep the application user friendly, once an action is chosen from the list, various additional fields may appear beneath the action list, where the user can either fill in the details of the requested action or choose the details from a predefined list (such as the case of the "pick up my child" action, where the children and the possible locations would be specified in the settings).

Fig. 4: The "Help" view

Fig. 5: "Requests" view

Only after selecting the requirements, can the user proceed to pressing the "Find Help!" button. This button then triggers the flooding algorithm, as described in Section 2.3. When a request is received by another user, she receives a notification and is asked to respond. The answer is used on line 16 of the flooding algorithm (Algorithm 1), where a confirmation is sent back to the requester if the user accepts the task.

3.2 Tracking requests

Users can see their current requests for help and their status in the "Requests" view (see Figure 5).

Again, the presentation of each item in the list is dependant on the semantics of the action. In the case of the "pick up my child" action, the name of the child, the location to be picked up from, and the date and time the child needs to be picked up is presented. In addition to those, the "status" parameter is also presented to help the user track the progress of its requests.

The status can have one of the following values, which are self-explanatory:

- Looking for a volunteer
- No volunteers found
- Assigned to *VolunteerName*
- Failed by *VolunteerName*
- Completed by *VolunteerName*
- Cancelled by *VolunteerName*
- Cancelled by *RequesterName*

The change in the status should sometimes trigger alert messages, informing the user of important changes. For instance: if the status changes to "No volunteers found", then a message should be sent to the user, notifying him that the **u-Help** platform failed to find a volunteer. Other important changes are also accompanied by appropriate messages.

The only two actions that are permitted in this view are:

– **Cancelling requests.** The cancel button is active from the moment the request is made and until half an hour before the task's deadline. A part of the future work is to allow different deadlines to be specified for different tasks. Additionally, deadlines could also be calculated dynamically based, for instance, on the location of the volunteer. We note that when pressed, the cancel button will change the status to "Cancelled by *RequesterName*" and delete the item from the list of requests. Furthermore, if a volunteer was assigned to this request, the the item will also be deleted from the volunteer's list of duties and an alert message is sent to the volunteer.
– **Rating requests.** The rating bar is only activated if the status is "Completed by *VolunteerName*" or "Failed by *VolunteerName*". Note that the **u-Help** platform designates a task as "failed" if the task's deadline has passed, but the volunteer has not confirmed that the task has been completed. If the volunteer marks it as completed, the requester can still give him an unsatisfactory rating. Similarly if **u-Help** thinks the volunteer failed to complete the task due to the deadline passing, but the user knows the task was completed adequately, the volunteer can be given a satisfactory rating. The completion mechanic thus exists mainly as a way of providing information to the user and does not affect the rating of the task.

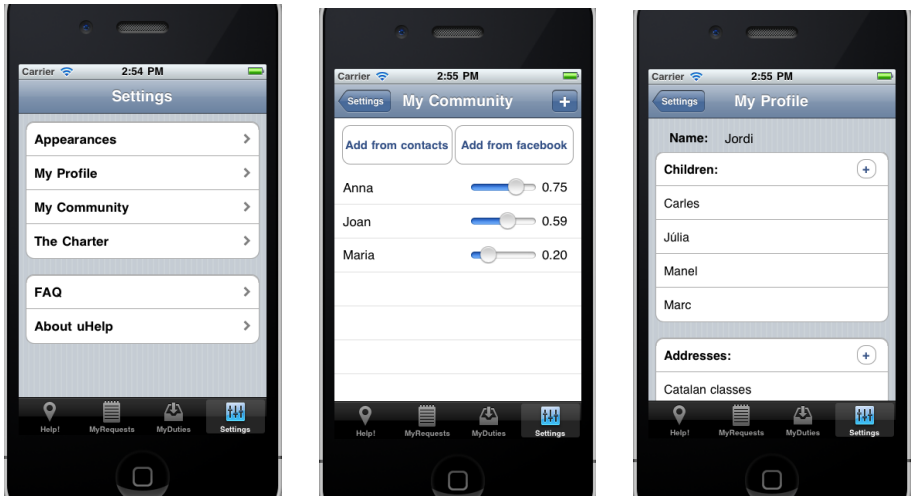
3.3 Tracking duties

In the "Duties" view, a user can see the list of tasks she has volunteered for. Moreover, she can cancel them, or flag them as completed. The interface is very similar to that of the "Requests" view, but with the difference that to complete the task, the volunteer may need extra information, which can be accessed from this view.

3.4 Settings

The final view for the user is the "Settings" view. This is where the user specifies all the information needed by the **u-Help** platform. The main menu (see Figure 6a) gives a good overview of what can be changed. The most important aspects to the user are that information about the social network can be imported and changed here (see Figure 6b), and the user can edit her profile, adding information required for the various tasks (see Figure 6c). For instance, information about her children, activities where they need to be accompanied, locations where they need to be picked up or brought to, etc.

The Charter gives information about community-specific settings, such as how long before a task the requester (and volunteer) can cancel and what tasks are supported by this **u-Help** community. We aim to extend the charter. One direction is to add norms to the charter, such as what the minimum requirements for "completing a task in a satisfactory manner" are. Another direction that can be explored simultaneously is to allow users to negotiate about changes to the charter. These changes are needed in order to support a growing and changing community.



(a) Main menu of the "Settings" view

(b) Add neighbours or change the trust level

(c) Add children, locations, etc.

Fig. 6: Various screens in the "Settings" view

4 Discussion and Conclusions

In this paper we have introduced the **u-Help** application, mainly in its capacity as a platform for finding help within a community. However, as we said in the introduction, there is a second aim behind its development, to study how a community evolves and the methods we presented need to be augmented or changed over time. The idea is to implement an adaptable community charter, with which the community can decide on such changes together. To study this, we aim to roll out the application in a small, controlled, community. In preparation for this, we are currently running agent-based simulations of such a community, using the **u-Help** application. Initial results show that the flooding algorithm and trust model ensure that untrustworthy members of the community are quickly ignored for help requests.

A specific extension that we intend to make in the **u-Help** application is the design of a charter in which the community's norms are described. For the tasks described in this paper, such norms could be that someone picking up children may never be late. A competing, more relaxed, norm could be that the volunteer may never be more than five minutes late. The community needs some way of proposing, and deciding upon such competing norms. Additionally, there must be some way of measuring norm compliance and identifying norms which need revising: if volunteers are given mostly satisfactory ratings despite not complying with a norm, the community may need to rethink that norm.

With this move to a more normative description of when a task is performed in a satisfactory manner, the incorporation of communicated ratings into the trust model can also be extended. We briefly mentioned that such ratings are subjective. If users are asked to give a rating of the volunteer, but additionally tick any norms that weren't complied with, this information can be used in the

communication to decide on the importance of the rating: different users give different importance to the various norms of the community.

However, such future work aside, the **u-Help** application as it is now, allows users to ask for help and find a trusted volunteer within their community. This is a first step towards building applications that explicitly support community formation around specific activities.

Acknowledgements

This work is supported by the Generalitat de Catalunya grant 2009-SGR-1434, the CBIT project (TIN2010-16306), the Agreement Technologies project (CONSOLIDER CSD2007-0022, INGENIO 2010), and the ACE ERA-Net project.

Author Contributions

The task ontology and similarity measures were conceived by MS and NO. The trust model was designed by JSM and AK, and the flooding algorithm by CS and NO. The implementation for iOS was made by JM. AK wrote the initial versions of this paper with contributions from AF, MS and NO; and prepared the final version. All authors read and approved the final manuscript.

References

1. Beech, S., Geelhoed, E., Murphy, R., Parker, J., Sellen, A., Shaw, K.: The lifestyles of working parents: Implications and opportunities for new technologies. Tech. Rep. HPL-2003-88(R.1), HP Laboratories (2004)
2. Conte, R., Paolucci, M.: Reputation in Artificial Societies: Social beliefs for social order. Kluwer Academic Publishers (2002)
3. Esteva, M., Rosell, B., Rodriguez-Aguilar, J.A., Arcos, J.L.: Ameli: An agent-based middleware for electronic institutions. In: AAMAS'04, pp. 236–243. ACM, New York, USA (2004)
4. Falcone, R., Castelfranchi, C.: Trust and transitivity: a complex deceptive relationship. In: TRUST@AAMAS'10. pp. 43–53. IFAAMAS, Toronto, Canada (2010)
5. Fellbaum, C. (ed.): WordNet: An Electronic Lexical Database. MIT Press (1998)
6. Golbeck, J.: Combining provenance with trust in social networks for semantic web content filtering. In: Moreau, L., Foster, I. (eds.) Provenance and Annotation of Data (IPAW 2006), LNCS, vol. 4145, pp. 101–108. Springer (2006)
7. Grüninger, M., Menzel, C.: The Process Specification Language (PSL): Theory and applications. AI Magazine 34(3), 63–74 (2003)
8. Jøsang, A., Gray, E., Kinateder, M.: Simplification and analysis of transitive trust networks. Web Intelligence and Agent Systems 4, 139–161 (2006)
9. Koster, A., Sabater-Mir, J., Schorlemmer, M.: Trust alignment: a sine qua non of open multi-agent systems. In: Meersman, R., Dillon, T., Herrero, P. (eds.) CoopIS'11. LNCS, vol. 7044, pp. 182–191. Springer, Hersonissos, Greece (2011)
10. Li, Y., Bandar, Z.A., McLean, D.: An approach for measuring semantic similarity between words using multiple information sources. IEEE Transactions on Knowledge and Data Engineering 15(4), 871–881 (2003)
11. Osman, N., Sierra, C., Sabater-Mir, J.: Propagation of opinions in structural graphs. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) ECAI'10. Frontiers in AI and Applications, vol. 215, pp. 595–600. IOS Press, Lisbon, Portugal (2010)
12. Sellen, A., Hyams, J., Eardley, R.: The everyday problems of working parents: Implications for new technologies. Tech. Rep. HPL-2004-37, HP Laboratories (2004)