Chapter 3
The uHelp Application

Nardine Osman, Bruno Rosell, Andrew Koster, Marco Schorlemmer, Carles Sierra, and Jordi Sabater-Mir

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. uHelp provides a platform for building a community of helpful people and supports them in finding help for day-to-day tasks. uHelp uses electronic institutions to coordinate interactions between individuals, and it relies on a trio of techniques — semantic similarity, a trust model, and a flooding algorithm — to help efficiently find the most trusted volunteers for a given task request. This chapter provides an overview of the uHelp application, describes its underlying electronic institution, and presents a brief introduction to the integrated technologies that allow uHelp find most suitable volunteers efficiently.

Nardine Osman
IIIA-CSIC, Barcelona, e-mail: nardine@iiia.csic.es
Bruno Rosell
IIIA-CSIC, Barcelona, e-mail: rosell@iiia.csic.es
Andrew Koster
IIIA-CSIC, Barcelona, e-mail: andrew@iiia.csic.es
Marco Schorlemmer
IIIA-CSIC, Barcelona, e-mail: marco@iiia.csic.es
Carles Sierra
IIIA-CSIC, Barcelona, e-mail: sierra@iiia.csic.es
Jordi Sabater-Mir
IIIA-CSIC, Barcelona, e-mail: jsabater@iiia.csic.es
3.1 Objectives

In this chapter, we present uHelp, 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, thus supporting the balancing of societal needs, which contributes to community well-being. For instance, the community could be a group of neighbours, friends or close family who turn to each other for help with performing day-to-day tasks. As a prototype scenario, we started by focusing on the community of parents who need help with dropping off their children at school, picking them up from school, or babysitting their children. According to [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.

In uHelp, picking up children is one of the main tasks. However, we have also expanded the domain and designed a number of other tasks, including a generic task, to allow people to ask for help with their day-to-day activities. Additionally, we 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 organising carpools.

The uHelp application essentially provides a platform to build and support a community to which members can turn to find help with day-to-day tasks [9, 8]. The uHelp application 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, such as Facebook, or constructed automatically by accessing the address book of people’s telephones when they join uHelp). The social network is used specifically to allow users to ask for help with picking up, or taking care of their children. Three key technologies are then used to allow one to find help efficiently: a flooding algorithm, a computational trust model, and a semantic similarity model.

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 flooding algorithm relies on the community member’s trust evaluations of one another to decide whether to forward the request or not, as requests are only forwarded to trusted volunteers. Trust evaluations are calculated automatically by the computational trust model based on requesters’ evaluations of the past performance of the volunteer, with respect to the specific task requested. This is made possible as uHelp allows the requester to evaluate a volunteer’s performance after they complete there tasks. 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. The computed semantic similarity measures essentially help the trust model use evaluations of a volunteer performing tasks in the past to estimate his performance at a similar future task.
The remainder of this chapter is divided as follows. Section 3.2 presents the implementation of the uHelp app; Section 3.3 presents the electronic institution specification underlying the uHelp app; and Section 3.4 describes the integrated technologies that help finding volunteers efficiently.

3.2 Illustration

In this section we present the user interface of the uHelp application. The implementation was carried out using the Ionic framework,\(^1\) which allows the development of the application as a web application that would automatically be translated as well into iOS and Android applications. This allowed us to build a cross-platform application. We also made use of the Apache Cordova platform,\(^2\) for developing plugins that map device hardware to plugins in javascript, which could then be used on the Ionic framework.

In order to use uHelp it is necessary for the user to configure her data upon downloading the application. She must first either login via Facebook or register with the system and import her connections from her list of contacts, to create her social network. Once this is done, the user can start to use uHelp to request help, and be contacted by others in the community who need help in return.

The uHelp application’s usability is designed around four main views: 1) the “Help” view, where users may request help; 2) the “Requests” view, where users can track and manage requests for help (issued by them or others); 3) the “Community” view, where users can track and manage their community, or social network; and 4) the “Settings” view, where users can change their user-specific settings, as illustrated shortly. In what follows, we go over each of the uHelp application views in a bit more detail.

1. **Asking for help.** Asking for help is done by pressing the “Help!” button (see Figures 1(a) and 1(c)). However, the user must first choose the action that she requires help with from the list tasks. The elements of this list, and the options given, correspond to elements in the predefined ontology. We note that different actions will require different parameters. To keep the application user friendly, once an action is chosen from the list (Figure 1(b)), the appropriate fields will appear, where the user can fill in the details of the requested action (Figure 1(c)). For example, if the action is “Care for a relative”, then the relative field appears, along with the pickup and drop-off location and time, but if the action is “Get me something”, then the object field appears, along with the pickup and drop-off location and time.

Only after filling in the required parameters can the user proceed to pressing the “Help!” button. This button then triggers a flooding algorithm that spreads the help request in order to find volunteers, as described in Section 3.4.1. When a

---

1. https://ionicframework.com
2. https://cordova.apache.org
Fig. 3.1 Various screens from the “Help” view

(a) The general task  (b) List of predefined tasks  (c) “Care for a relative” task

Fig. 3.2 The “Requests” view  Fig. 3.3 The “Community” view
Table 3.1 The state of a requester’s task and its representation

<table>
<thead>
<tr>
<th>State</th>
<th>Colour</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>looking for volunteers</td>
<td>yellow</td>
<td>Waiting...</td>
</tr>
<tr>
<td>pending assignment 1</td>
<td>green</td>
<td>Choose volunteer!</td>
</tr>
<tr>
<td>pending assignment 2</td>
<td>red</td>
<td>Choose volunteer!</td>
</tr>
<tr>
<td>assigned</td>
<td>yellow</td>
<td>Help on its way</td>
</tr>
<tr>
<td>completed</td>
<td>green</td>
<td>Please, rate!</td>
</tr>
<tr>
<td>rated</td>
<td>grey</td>
<td>Rated</td>
</tr>
<tr>
<td>cancelled</td>
<td>grey</td>
<td></td>
</tr>
<tr>
<td>expired</td>
<td>grey</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2 The state of a requestee’s task and its representation

<table>
<thead>
<tr>
<th>State</th>
<th>Colour</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unanswered</td>
<td>green</td>
<td>Can you help?</td>
</tr>
</tbody>
</table>
| declined     | grey   | I cannot help :(
| accepted     | yellow | Waiting to be selected ... |
| selected     | red    | Do it!            |
| not selected | grey   | Help no longer needed, thanks! |
| completed    | grey   | Completed         |
| cancelled    | grey   |                   |
| expired      | grey   |                   |

request is received by another user, she receives a notification and is asked to respond.

2. **Tracking requests.** Users can see requests for help (sent by or to them) along with their state in the “Requests” view. In Figure 3.2, green text bubbles (to the right) represent one’s own requests, while grey ones (to the left) represent requests received. Again, the presentation of each item in the list is dependant on the semantics of the action. In the case of the “Care for a relative” action, the name of the child and the pickup location will be displayed. Additionally, the “state” parameter is presented to help the user track the progress of requests, such as ‘waiting for volunteers’, ‘help on its way’, or ‘completed’. The state for the requestee is different from that of the requester. Tables 3.1 and 3.2 present the different states for each, along with how is this state visualised to the user. The visualisation is executed by assigning a colour along with a piece of text describing that state (see Figure 3.2). We use colours to categorise states into four categories: (1) those that need to be acted upon, which get the colour green; (2) those that need to be acted upon immediately (for example, when ones needs to urgently select a volunteer as the deadline to select volunteers is approaching), which get the colour red; (3) those that require no action at the moment, as others are expected to be acting upon those tasks, which get the colour yellow; and (4) those that are considered closed, which get the colour grey.

The requester’s actions that are permitted in this view are: 1) cancelling requests, where the cancel button should be active from the moment the request
is made until a predefined amount of time preceding the task’s deadline; 2) chat with volunteers, where the requester can have private chats and even calls with the volunteers if needed; 3) select volunteer, where the requester can select her volunteer from the list of available volunteers; and 4) rating requests, where the rating bar should only be activated if the task has been completed or the deadline has passed.

The volunteer’s actions that are permitted in this view are: 1) accept, where the volunteer accepts to volunteer for some task; 2) decline, where the volunteer declines from volunteering for some task; 3) cancel, where a volunteer can cancel after accepting, and the cancel button should be active from the moment the task is accepted until a predefined amount of time preceding the task’s deadline; 4) chat with requester, where the volunteer can have a private chat or even call with the requester if needed; 5) complete task, where the volunteer can mark its task as completed before the deadline has passed.

3. Community. In the “Community” view (Figure 3.3), a user can see the list of uHelp community members in its social network and can view (or even manually adjust, if needed) the trust they hold on each of those members.

4. Settings. The final view for the user is the “Settings” view. This is where the user specifies all the information needed by the uHelp platform. The main menu (Figure 4(a)) gives a good overview of what can be changed. For example, the user can edit her profile, adding information required for the various tasks. In this case, we show how information about her children and the locations where

![Fig. 3.4 Various screens from the “Settings” view](image-url)
they need to be picked up or brought to may be edited. Additionally the user should be able to select its privacy setting, for instance, whether requesters can contact a volunteer by phone or not (Figure 4(b)).

3.3 The EI Specification

In this section we show the underlying electronic institution (EI) specification [2] of the uHelp application.

3.3.1 Dialogical Framework

In the EI model of uHelp, we can distinguish between three main roles:

- User, which is the main agent representing the human user, and it is the agent that may ask for help, or receive and propagate help requests;
- Requester, which is the role that the agent representing the human user takes when dealing with a specific request that has been requested by that human user; and
- Requestee, which is the role that the agent representing the human user takes when dealing with a specific request that has been requested from that human user.

Basically, each human user agent will be represented by the User agent. However, every time a request for help is made, the agent will “stay and go” to another scene where it plays the role of Requester or Requestee to deal with that request. Additionally, we have one software agent:

- TrustAgent, which is the agent that updates one’s trust on another and has access to the database, which keeps track of users’ ratings on each other along with the help requests that users receive (as required by the flooding algorithm, see Line 4 of Algorithm 1).

3.3.2 Performative Structure

The workflow of the uHelp application can be divided in three stages: asking for help, flooding a help request, and dealing with help requests. These three stages can naturally be represented in an EI-specification as three separate scenes, “Help” scene, “Flooding” scene, and “Requests” scene. The first scene will have exactly one instance, where everyone is allowed to ask for help. However, the second scene will have as many instances as help requests issued. All user agents can be in more than one scene at the same time (and even in more than one instance of the second
Fig. 3.5 The performative structure of the uHelp application

and third scenes, as more than one help request may be flooded at a time). The user agents can even play different roles in different scenes. For instance, the agent may be a Requester in one scene and a Requestee in another. The idea is that every human user will be represented by a User agent in the “Help” scene (asking for help). Then every time a help request is issued in that scene, a new instance of the “Flooding” scene is created for that request, which everyone moves to by performing a “stay and go”. In the “Flooding” scene, the user agents (with the help of the trust agents) will propagate a help request as illustrated by the flooding algorithm of Section 3.4.1. However, the first time a request is made, the requester creates a new instance of the “Requests” scene for this request. And then, every user agent that receives this request in the “Flooding” scene will move to the “Requests” scene instance by performing a “stay and go”. In this new scene, the agent issuing the request will play the role Requester and the agents receiving that request will play the role Requestee. The performative structure of the institution is presented by Figure 3.5. Next, we present the three scenes in more detail.
1. Help scene: In this scene, the users (represented by their User agents) can issue help requests when needed. In addition to sending those messages, upon sending a help request, everyone will perform a stay and go to a new instance of the Flooding scene that deals with propagating this request. The protocol of the Help scene is straightforward and it is presented by Figure 3.6.

2. Flooding scene: In this scene, first the trust agent provides details about one’s trust on its neighbours, and the help request is then propagated (or not) accordingly. This repeats until propagation terminates, where the conditions for
termination are discussed in further detail in Section 3.4.1. However, we note that when an instance of this scene is first created, the user agent issuing the help request automatically performs a stay and go to a new instance of the Requests scene, to manage with this request. Then, every time a new user agent receives this help request in the Flooding scene, and regardless of whether it propagates it or not, that user agent will also perform a stay and go to the corresponding instance of the Requests scene. The protocol of the Flooding scene is relatively straightforward and it is presented by Figure 3.7.

3. Requests scene: In this scene the Requestees can accept or decline, and they are allowed to change their mind, up until a certain deadline passes for collecting responses, which will move the scene to the next state (the “Pending Assignment 2” state of Figure 3.8). As long as there is at least one accept, and the deadline for assigning volunteers has not passed yet, the Requester will be able to select a volunteer (moving the state to “Assigned 1”, Figure 3.8). At this point the Requestee either declares the task as done, or the task is assumed to be completed after the deadline passes (moving the state to “Completed”, Figure 3.8). After this, the Requester is expected to rate the volunteer’s performance, which triggers the trust agent to update the database accordingly. We note however, that both the Requester and Requestee may cancel the request, or volunteering, respectively, at any point in time (which moves the state to “Cancelled by Requester” or “Cancelled by Requestee”, respectively – Figure 3.8). Additionally, if no Requestees volunteer in time, or the Requester does not select a volunteer in time, then the task will be marked as expired (the “Expired” state, Figure 3.8). The protocol of this scene is presented by Figure 3.8.

3.4 The Integrated Technologies

3.4.1 Flooding Algorithm

The flooding algorithm is the core computational process in the uHelp 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 for a specific task, the flooding algorithm sends this request to that person’s trusted 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 time passed since the initial request was made. In fact, in Figure 1(a), one can see how
the user controls the flooding algorithm by deciding the required trust level for a given task, along with the friendship level, which represents the number of hops in a graph (friends stand for 1 hop, friends of friends stand for 2 hops, etc.).

The reason for this is that we not only want to find someone willing to volunteer for a task, but the person must also be trustworthy when performing this task. The way we calculate trust between two people is described in Section 3.4.3. To calculate trust along a path, we assume that trust satisfies the triangular norm inequality, that is for all $\alpha, \beta, \gamma$ nodes in a network $\text{Trust}(\alpha, \gamma) \leq T(\text{Trust}(\alpha, \beta), \text{Trust}(\beta, \gamma))$ for some T-norm function $T$. Thus, trust is monotonically decreasing along any path. We use multiplication for the experiments, for instance we get $\text{Trust}(\alpha, \gamma) = \text{Trust}(\alpha, \beta) \cdot \text{Trust}(\beta, \gamma)$. Though more optimistic functions could be used, like $\min$.

The flooding algorithm stops propagating the request when the cumulative trustworthiness of a node falls below a certain threshold $\tau$. Algorithm 1 gives the pseudocode for the algorithm that performs this flooding.

Every node in the network runs Algorithm 1. It works in a straightforward manner: every time a node’s user needs help she calls the $\text{FLOOD}$ function with the following arguments:

- the task she needs help with (which is obtained from the form the user fills in before pressing help),
- the minimum level of trust $\tau$ required on the person to execute the task (also obtained from the form the user fills in before pressing help button, see the “Trust” field in Figure 1(a)),
- the maximum number of hops (or friendship level) that the flooding algorithm is allowed (also obtained from the form the user fills in before pressing help button, see the “Friendship” field in Figure 1(a)),
- the deadline for someone to accept or reject (which is obtained from the task’s deadline that the user specifies, see for instance the Drop off date of Figure 1(c). We note that this deadline precedes the task’s deadline by a predefined fixed amount of time. For instance, if the deadline for the task is tonight at 6pm, then the deadline to accept might be today at 3pm, which is 3 hours before the task’s deadline.)

The other parameters of the $\text{FLOOD}$ function are automatically set accordingly.

- $\text{messagetype} = \text{HELP}$, this is because the flooding algorithm is not only used to ask for help, but also to cancel previous requests. For instance, when the user cancels a request before selecting a volunteer, then all those that have been asked for help should be informed that the request is now cancelled. In this case, we run the same flooding algorithm, but with $\text{messagetype} = \text{CANCELLED}$. Also, when a user selects a volunteer, then all those that have been asked for help (except the selected volunteer) should be informed that their help is no longer needed. In this case, we again run the same flooding algorithm, but with $\text{messagetype} = \text{NOTNEEDED}$.
- $\text{pathtrust} = 1$, this is the initial trust. Every time a message propagates to its neighbouring nodes, trust is multiplied by the trust on that new node. We start
Algorithm 1 Flooding Algorithm. We note by $n \rightarrow m$ the request to execute method $m$ at node $n$. Methods are defined as functions and are non-blocking.

Require: $me : \text{Node}$ \hspace{1cm} \triangleright My identifier
Require: friends : $\mathcal{N}_\text{Node}$ \hspace{1cm} \triangleright Set of neighbouring nodes
Require: Trust : Node $\times$ Task $\rightarrow [0,1] \cup \perp$ \hspace{1cm} \triangleright Trust on friends for given tasks. $\perp$ for unknown people
Require: OldPathTrust : Task $\rightarrow [0,1]$ \hspace{1cm} \triangleright Previous trust received from a path for given tasks. Initially it is -1.
Require: $T : [0,1] \times [0,1] \rightarrow [0,1]$ \hspace{1cm} \triangleright A T-norm function, e.g. min, $\cdot$.
Require: $\sigma : [0,1]$ \hspace{1cm} \triangleright Minimum increase in trust to re-flood the network
Require: ReceivedRequests : $\mathcal{N}_\text{Task}$ \hspace{1cm} \triangleright The set of received requests.

1: function PROPAGATE(task, messagetype, $\tau$, pathtrust, path, deadline, hops)
2: \hspace{1cm} if $me \notin$ path and $\text{Now}() < \text{deadline}$ then
3: \hspace{2cm} if messagetype $==$ HELP then
4: \hspace{3cm} if path $\notin$ me.ReceivedRequests then
5: \hspace{4cm} OldPathTrust(task) := pathtrust;
6: \hspace{4cm} me $\rightarrow$ FLOOD(task, messagetype, $\tau$, pathtrust, path $\cup$ me, deadline, hops, false);
7: \hspace{3cm} else if pathtrust $-$ OldPathTrust(task) $>$ $\sigma$ then
8: \hspace{4cm} OldPathTrust(task) := pathtrust;
9: \hspace{4cm} me $\rightarrow$ FLOOD(task, messagetype, $\tau$, pathtrust, path $\cup$ me, deadline, hops, true);
10: \hspace{2cm} end if
11: \hspace{2cm} else
12: \hspace{3cm} me $\rightarrow$ FLOOD(task, messagetype, $\tau$, pathtrust, path $\cup$ me, deadline, hops, false);
13: \hspace{2cm} end if
14: \hspace{2cm} end if
15: end function

16: function FLOOD(task, messagetype, $\tau$, pathtrust, path, deadline, hops, asked?)
17: \hspace{1cm} if $\neg$ asked? then
18: \hspace{2cm} if messagetype $==$ HELP then
19: \hspace{3cm} me $\rightarrow$ Msg.Help(task,deadline);
20: \hspace{2cm} else if messagetype $==$ NOTNEEDED then
21: \hspace{3cm} me $\rightarrow$ Msg.NotNeeded(task);
22: \hspace{2cm} else if messagetype $==$ CANCELLED then
23: \hspace{3cm} me $\rightarrow$ Msg.Canceled(task);
24: \hspace{2cm} end if
25: \hspace{2cm} end if
26: \hspace{1cm} for all $n \in$ friends do
27: \hspace{2cm} NewPathTrust := $T$(Trust($n$,task), pathtrust);
28: \hspace{2cm} if NewPathTrust $\geq$ $\tau$ and length(path) $-$ 1 $<$ hops then
29: \hspace{3cm} $n \rightarrow$ PROPAGATE(task, messagetype, $\tau$, NewPathTrust, path, deadline,hops);
30: \hspace{2cm} end if
31: \hspace{1cm} end for
32: end function
with the value 1. That is, if the requester $r$ propagates the request to its neighbour $n$, then $r$'s trust on $n$ is initially multiplied by 1.

- **path** = \{me\}, this adds the requester as the first person in the path of people this message is being sent to. This ensures that the flooding algorithm does not later on ask the requester for help for his own request. (see Line 2 of Algorithm 1)
- **asked?** = yes, this is the parameter that decides whether the user should be informed of the new message or not, and it is needed to make sure that if one's uhelp app propagates a message more than once, the human user of this app is not informed more than once. We note here that we do allow a node to propagate a message more than once, if received from different paths. Re-flooding is discussed and motivated shortly.

When a node runs the FLOOD function, it first checks whether that user should be informed of any new help requests (or that help is no longer needed, or cancelled, in the other cases of using the flooding algorithm). Only if the user has not been already informed (i.e., if ¬asked?), then the appropriate pop-up message is triggered. After that, the node goes on to propagate that message to its neighbours, if the conditions are fulfilled. Conditions are considered fulfilled when the trust on that neighbour is within the threshold $\tau$, and the number of hops within the acceptable limit ($\text{hops}$). Recall that when adding a new node to the path, trust is updated by multiplying the the trust on the node propagating the request by the trust the propagating node has on the propagated node (Line 27 of Algorithm 1).

The message is propagated by having the trusted neighbouring nodes executing the PROPAGATE function. With the exception of excluding the asked? parameter, the function has the same parameters as the FLOOD function, with the pathtrust updated with the new trust value.

When a neighbouring node executes a PROPAGATE function, it first verifies that there is no loop with the flooding by checking that the node is not already included in the path and that we are not beyond the deadline. Then, the node first checks whether it has been asked for help. If it has not, then the FLOOD function is called, with the asked? parameter set to false and the node appending itself to the path. If it has been asked earlier for help for this specific task, then it is allowed to re-flood the network if the task has been requested previously but the path trust on the node is now sufficiently higher than before (by a difference larger than $\sigma$). In this case, the FLOOD function is called, with the asked? parameter set to true and the node appending itself to the path. This last point is very important as the depth by which a request percolates the network depends on the cumulated level of trust. There may be several paths connecting any two nodes, and thus the trust between them should be considered as the maximum of the cumulated trust over the paths. If we reach a node with a certain level of trust we will flood the network with that value. However, if later on a higher value is found we need to re-flood the network as this high value may make nodes previously not reached by the non-increasing effect of the T-norm be reachable now.

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 request path (due to the condition on Line 2 of Algorithm 1). Setting parameter $\sigma$
is very important for the efficiency of the algorithm, as if it is set very low the number of messages can become exponential in the number of nodes for particular topologies. In the worst case, with $\sigma = 0$, for a graph $G$, the number of messages is $\sum_{p \in \text{loop-free paths}(G)} \text{length}(p)$, which can be exponential in the number of nodes. An adequate value has to be set experimentally.

Finally, we note that the algorithm limits the number of hops in addition to considering the trust level. This is achieved by comparing the length of the path with the $\text{hops}$ parameter representing the maximum number of hops in the FLOOD method. This limitation in the number of hops makes our algorithm somehow similar to the Gnutella flooding algorithm, although in our case we also limit the flooding by taking trust into account.

### 3.4.2 Semantic Similarity

The tasks managed by the uHelp community consist of the activity the task is about (care for a relative, which could include babysitting, picking up, cooking, etc.), and sometimes, the relevant relative to whom the activity is applied (we focus on children, and hence we may have baby, toddler, preschooler, etc.). Consequently we need to handle two separate hierarchies.

On one hand, the different kinds of activities are usually organised in a meronomy, specifying which sub-activities may be part of a given activity (e.g., that changing nappies is a sub-activity of babysitting). Here we take inspiration from the design decisions suggested by the Process Specification Language (PSL) [6]. An example of a meronomy of activities and sub-activities for the uHelp community is given in Figure 3.9.

![Fig. 3.9 Meronomy of activities](image)

On the other hand, the different kinds of children are more naturally organised in a taxonomy, specifying subclasses of children (e.g., that a toddler is a child). Figure 3.10 shows an example of a taxonomy of children for the uHelp community.

---

3 A meronomy is a type of hierarchy that deals with part-of relationships, in contrast to a taxonomy whose structure is based on is-a relationships.
that reflects a hypernym-hyponym relation, which is similar to the one defined in WordNet [4].

![Fig. 3.10 Taxonomy of children](image)

For our specific case study, a task is a pairing of a “care for” activity with a “child”, for instance (giving a lift, preschooler) or (playing, toddler). If the activity specified is not a leaf of the meronomy, the requester expects the volunteer to be capable of performing all the subactivities involved. For instance, if the activity is giving a lift, the volunteer is expected to perform both picking up and dropping off. However, in a taxonomy this is treated differently: if the child is not a leaf of the taxonomy it is simply not as detailed a specification: a 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 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.

Consequently, task similarity has to reflect this influence on the quality of task performance. For this reason, activities and their meronomical structure, and types of children and their taxonomical structure, affect the trust evaluations of specific tasks in different ways.

We use OpinioNet [11] to propagate evaluations of other activities through the meronomy. 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. In uHelp, the opinions are satisfaction ratings of how a user performed an activity and the structural graph is the meronomy of those activities. We argue that a user’s performance of one activity should influence his trustwor-
thinness 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 evaluations, 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 preparing meal 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 changing nappies if there is no direct evaluation of the target performing the changing nappies 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.

The similarity over the child taxonomy is considered from a more semantic perspective and we use the measure proposed by Li et al. [10], which combines well-known edge-based and node-based techniques, and correlates well with similarity assessments as done by humans. In particular, it takes three aspects of taxonomies into account:

- the distance $l$ between two tasks $t_1$ and $t_2$ in the taxonomy: the closer, the more similar the tasks are;
- the depth $h$ in the taxonomy of their most specific subsumer $\text{sub}(t_1, t_2)$: the deeper in the taxonomy the subsumer is, the more similar the tasks are;
- the local semantic density $d$ of instances of these tasks: the greater the information content of the subsumer, $d = -\log p(\text{sub}(t_1, t_2))$, the more similar the tasks are.

Consequently, Li et al. define the first measure to be anti-monotonic in the range $[0, 1]$ and the other measures to be monotonic in the same range, where 0 represents complete dissimilarity, and 1 represents complete similarity. The contribution of each of these measures is taken independently of each other:

$$\text{sim}(t_1, t_2) = e^{-\alpha l}\cdot \tanh \beta h \cdot \tanh \lambda d$$
Constants $\alpha$, $\beta$ and $\lambda$ are positive real numbers that determine the relative influence of each of the three measures on the final similarity. They provide us with suitable adjustment points for the overall semantic similarity to fit well and evolve with the actual usage of terms by a concrete community.

### 3.4.3 Trust Model

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. The transitivity of trust is not uncommon in the trust literature. For instance, TidalTrust [5] is a trust model that propagates trust evaluations through a social network, 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, we refer the interested reader to [7, 3].

The trust mechanism implemented in uHelp makes use of the evaluations that users generate after a task is completed by a volunteer.

For instance, every time someone volunteers to care for a child, the adult responsible of the child generates an evaluation. An evaluation $E$ is a tuple with the form

$$(\text{Requester}, \text{Volunteer}, \text{ActivityType}, \text{ChildType}, \text{Value})$$

where $\text{Requester}$ is the person that was asking for help, $\text{Volunteer}$ is the ID of the person that took care of the child, $\text{ActivityType} \in$ Meronomy of activities, $\text{ChildType} \in$ leafs in Children’s taxonomy, and $\text{Value} \in [0, 1]$ with 0 meaning complete failure and 1 complete satisfaction. We refer to the elements in the tuple using superscripts. For instance, to refer to the ChildType value we use the form $E^{\text{ChildType}}$.

If the activity includes other subactivities, the user (the requester of the task) can decide to evaluate the general activity only, or to perform a detailed evaluation of each one of the leaf activities. For instance, if the activity was “giving a lift”, the user can evaluate just “giving a lift” or he/she can evaluate each one of the single subactivities (“picking up”, “dropping off”). The ChildType is always a leaf in the Children’s taxonomy.

Some examples of evaluations:

- $(\text{Ann123}, \text{John243}, \text{“entertaining”}, \text{“toddler”}, 0.7)$

---

Note that although in this section we talk about caring for children, the same approach is followed for other tasks, where $\text{ActivityType}$ depends on the task, and the $\text{ChildType}$ can be replaced either by $\text{Ob jectType}$ (for instance when someone asks others to donate to them, lend them, or get them something) or $\text{LocationType}$ (for instance when someone asks others to go with them somewhere).
We calculate the trust of an individual as a function of the trust that the individual has regarding the type of activity (see meronomy of activities) and the object of the activity (see children’s taxonomy).

\[
Trust(R, V, T, Ch, t) = \alpha * Trust_{object}(R, V, Ch, t) + (1 - \alpha) * Trust_{activity}(R, V, T, t)
\]

where \( R \) is the requester ID, \( V \) the volunteer ID, \( T \) the activity, \( Ch \) the type of child, \( t \) the time and \( \alpha \) the weight given to each type of trust. We discuss each type of trust next.

\textbf{Trust}_{object}(R, V, Ch, t). \textit{We use a similarity threshold to filter out the evaluations associated to types of children too distant from the target child in the Children’s taxonomy. The similarity value for the Children’s taxonomy is calculated using the following formula:}

\[
simVal(R, Ch_i, Ch_j) = \begin{cases} 
\text{sim}(Ch_i, Ch_j) & \text{if sim}(Ch_i, Ch_j) > \text{simTh} \\
0 & \text{otherwise}
\end{cases}
\]

where \( \text{sim}(Ch_i, Ch_j) \in [0, 1] \) is the semantic similarity between types of children \( Ch_i \) and \( Ch_j \) as already defined in the previous section and \( \text{simTh} \in [0, 1] \) is the similarity threshold defined by the community. Below that threshold, the similarity between two types of children is considered too low to use the associated experience.

The trust that requester \( R \) gives to volunteer \( V \) associated to a particular object \( Ch \) at time \( t \) is calculated using the formula:

\[
Trust_{object}(R, V, Ch, t) = \frac{\sum_{E_i \in \text{Evaluations}(t)} (\text{simVal}(R, Ch, E_i^{ChildType}) \cdot E_i^{Value})}{\sum_{E_i \in \text{Evaluations}(t)} \text{simVal}(R, Ch, E_i^{ChildType})}
\]

where \( \text{Evaluations}(t) \) is the subset of evaluations in the time window \([t - TWin, t]\), where \( TWin \) is a predefined parameter.

\textbf{Trust}_{activity}(R, V, T, t). \textit{To evaluate the trust of an individual regarding the type of activity that she/he is requested to perform, we make use of the meronomy of activities and the OpinioNet algorithm [11]. OpinioNet allows us to consider not only the direct evaluations of an activity but the evaluations associated to subactivities and superactivities. For instance, if there are evaluations of a volunteer associated to “entertaining” a child, these evaluations can be used to say something about the capacity of the volunteer to “singing songs”, “playing” and “watching movies”. Similarly, if there is an evaluation associated to “dropping off” a child it can be used to say something about the capacity of the volunteer of “accompanying” a child. For the details on how OpinioNet works, we refer the interested reader to [11].}
References