

SEMANTIC SOCIAL NETWORKING VIA THE EXTENSIBLE
MESSAGING AND PRESENCE PROTOCOL (XMPP)

by

Geoffrey Foster

Supervised by Dr. Tony White
School of Computer Science

Honours Project (COMP 4905)
Submitted in partial fulfillment of the
requirements for the degree of
Bachelor of Computer Science, Honours

at

Carleton University
Ottawa, Ontario
April 2007

© Copyright by Geoffrey Foster, 2007

Abstract

Social networking services are quite useful and have been steadily gaining in popularity, but each service typically exhibits the same problems. These are, the lack of actual real-time communication and socializing and also the data pertaining to the social network being stored in a closed manner. The following paper is an experiment in constructing social networks on top of an existing instant messaging network and utilizing open technologies. By using an instant messaging network for the backbone of the social network users are easily able to communicate in real time by merely using the instant messaging network and thus solving the first problem. The second problem is overcome by using open technologies, specifically RDF, as a data storage mechanism.

This paper begins with an introduction to the problems with existing social networking websites and instant messaging technologies and then proceeds to provide the required background information about the technologies that are used. This is followed by an explanation of the messages that are passed between clients during the construction and maintenance of the social networks which is then followed by the conclusions and a brief analysis of the increase in the number of messages.

Acknowledgements

I would like to thank Dr. Tony White for supervising this project. I would also like to thank my friends and family for their support. Also, a thanks goes out to all those on my own Jabber buddy list who put up with me constantly signing on and off while I was doing testing on my client implementation. Finally, I would like to thank the creators of the Smack XMPP library as well as the creators of the Jena RDF library who made my life easier by providing a good set of tools to use.

Table of Contents

Abstract	ii
Acknowledgements	iii
List of Figures	vi
List of Algorithms	vii
Chapter 1 Introduction	1
Chapter 2 Background	3
2.1 Semantic Web	3
2.1.1 RDF	3
2.2 Instant Messaging	4
2.2.1 Extensible Messaging and Presence Protocol	4
Chapter 3 Protocols	7
3.1 Introduction	7
3.2 Network Generation	8
3.3 Network Management	14
3.3.1 Adding a Buddy	15
3.3.2 Removing a Buddy	16
3.3.3 Pushing Data	21
3.3.4 Determining Degrees of Separation	22
Chapter 4 Results	24
4.1 Analysis of Additional Messages	24
4.1.1 Network Creation	24
4.1.2 Adding/Removing Buddies or Performing a knows-push	25
4.2 Implementation and Testing	26

Chapter 5	Future Work	27
Appendix A	Compact Disc Contents	28
Bibliography	29

List of Figures

Figure 3.1	Initial Network Creation	9
Figure 3.2	Steps involved in a knows-query	11
Figure 3.3	Receiving knows-query results	13
Figure 3.4	Roster changes propagating through network	14
Figure 3.5	Adding a buddy	17
Figure 3.4	Removing a buddy	21
Figure 3.5	Steps involved in a knows-query	22

List of Algorithms

1	Sign-in	8
2	ProcessKnowsResults(G, P_p, jid, K)	12
3	AddRosterEntry(G, P_p, P_a)	16
4	RemoveRosterEntry(G, P_p, P_r)	19
5	FoafCleanup(G, P_p)	19
6	BreadthFirstSearch(G, P_p)[Cormen et al., 2004]	23

Chapter 1

Introduction

A social network is a graph in which the nodes represent people or organizations and the links between the nodes represent some form of social relationship between them. Typically the link will represent the idea that the two nodes (people) know each other or are friends with one another.

The analysis of social networks is used in many different fields of study such as sociology and anthropology and as a result many different ideas have resulted from these studies. Of particular note is the Small World Phenomenon[Milgram, 1967] which suggested that two random United States citizens were, on average, connected through a chain of six people.

Traditionally a social network has referred to social connections in the real world but over the course of the past decade social networks have made the transition onto the internet. Beginning in 1995 with Classmates.com many other sites have emerged for the purposes of social networking. Some sites attempt to build social networks around a niche area while others attempt to build their social networks to encompass as many people as possible. Some sites try to build their networks with a very social and relaxed atmosphere while others try to provide a more professional or business-like atmosphere. Some of the more popular social networking websites at present are Facebook[Mark Zuckerberg, 2007], MySpace[MySpace, 2007] and LinkedIn[LinkedIn Corporation, 2007]. Facebook was originally created to mimic the social atmosphere of higher-educational facilities while MySpace tried to mimic the independent music and party scene. LinkedIn on the other hand is designed to connect people based upon business and professional relationships.

There are many benefits of social networking websites. Sites like Facebook have often allowed users to reconnect with friends from their past, while sites like MySpace have allowed people to discover new independent musical artists. LinkedIn allows job seekers to be matched up with job offerings through the people they know. On top of

this many people also find the usage of social networking websites to be, in general, fun.

Although social networking websites have many benefits, there are also some noticeable problems with them. Users will often sign up for multiple social networking websites because of friends being scattered among the different services and the users desire to be apart of each of their friends social network. This leads to many scattered and incomplete social networks as well as plenty of duplicated data. The duplicated data stems from the ability for users to enter information about themselves that is displayed in a profile. Typically, in more social and non-professional oriented social networking services, this information includes information such as favorite music, movies, foods, television shows and so on. This data is then usually stored in a manner known only to the developers of the social networking website which makes it very non-portable. This in itself does not seem like such a bad thing, but when a user starts to use multiple social networking websites they will often have to duplicate the same data. When the user then chooses to update their profile information they are then required to update it on each of the websites instead of just once.

In addition to this, the majority of these social networking services offer no real means of actual socializing. Many of the sites provide features for leaving some form of message but do not offer any sort of real-time communication.

This paper, as well as the associated implementation, is an experiment in attempting to solve both of these problems by combining social networking and instant messaging together and constructing social networks on top of an existing instant messaging network. The utilization of an instant messaging network solves the problem of real-time communication. In order to solve the problem of data-duplication a standard data storage mechanism, RDF, will be utilized which makes it easily transferrable to anywhere that supports it.

Chapter 2

Background

2.1 Semantic Web

The Semantic Web is a term used to describe a philosophy, a set of design principles, working groups, and various technologies. Part of the idea behind the Semantic Web is to express web content and information in a manner that is easily understood and interpreted by computer software[Herman, 2007].

2.1.1 RDF

RDF (Resource Description Framework) is a technology used within the Semantic Web community to represent statements about resources in the form of subject-predicate-object expressions called triples. An example of such a statement is “the sky”–“has the colour”–“blue”. In this case the subject is “the sky”, the predicate “has the colour” and the object is “blue”.

Friend of a Friend (FOAF)

The Friend of a Friend (FOAF) project was originally created by Libby Miller and Dan Brickley and aims to create a machine-readable modeling of user profiles and social networks. It uses RDF to represent relationships between people as well as attributes about them, such as name, gender, interests, and so on. Storing social network information in an RDF document is very beneficial. It provides a standardized way of storing the information thus making it very portable. It also allows queries to be run locally on information that is cached in the document resulting in reduced latency when asking a question of the social network. An RDF document is also easily extended. For example, relationships between people, typically defined by the knows predicate in a FOAF document can be extended to provide more specific information about the relationship. The information conveyed could be genealogical

specifying parent, child and sibling relationships or it could convey other relationships such as marriage or co-workers.

2.2 Instant Messaging

The concept of instant messaging (IM) essentially began in the late 1980s with the creation of Internet Relay Chat (IRC). IRC allowed users to join chat-rooms and send messages but lacked the concept of a buddy list, that is, a list of friends whom they knew. Then, in the second half of the 1990s, Mirabilis created ICQ. ICQ is an instant messaging client and communication protocol. The client consisted of a buddy list and the ability for users to send messages to each other. Soon after the creation of ICQ America Online (AOL) created their own instant messaging platform called AIM and around the same time they also acquired ICQ. The popularity of instant messaging continued to increase and other companies, such as Yahoo and Microsoft, created their own IM clients, Yahoo! Messenger and MSN Messenger. Each of these instant messaging services created their own proprietary communication protocols which prevented the different clients from communicating with one another. This resulted in the same negative situation that social networking sites have, which is the requirement of users to have, and use, multiple accounts on each of the networks in order to be able to chat with all of their friends.

In 1999-2000, the Jabber protocol was created as an attempt to solve this problem. It was created to be an open and extensible messaging network based around XML. From 2002 to 2004 the Jabber protocol was formally standardized and IETF formalized resulting in RFC 3920[Saint-Andre, 2006a] and RFC 3921[Saint-Andre, 2006b]. Since then Jabber has become officially known as XMPP which is short for eXtensible Messaging and Presence Protocol.

2.2.1 Extensible Messaging and Presence Protocol

Extensible Messaging and Presence Protocol (XMPP) is an open XML stream based protocol for real-time communication. There are a number of key components of XMPP that are utilized by this project. They are: Jabber Identifiers, the Roster, IQ Stanzas and XMPP Extension Protocols.

Jabber Identifiers

A Jabber Identifier (JID) is a unique identifier. The most common usage, and the one used throughout this project, is the representation of an instant messaging user. The JID is structured much like an e-mail address and consists of a user name and a domain. Optionally, a resource may be attached to it that typically defines the client that is connected. The structure of the JID that will be used throughout this paper and project is that of either *user@host* or *user@host/resource* where the latter is referred to as a JID with a resource or a “full” JID.

Roster

A roster is a users contact list (also known as a buddy list). It contains any number of roster entries where each roster entry is identified by a unique JID. The roster is stored on the server that a client utilizes for XMPP communication.

IQ Stanzas

An IQ stanza, which is short for info/query, is an XMPP message type that is used for making a request and receiving a response back from an entity. The idea behind an IQ message is to send a message to either the server or another client and receive a response. Listing 2.1 shows the basic structure of an IQ message.

```
<iq id='UID'
  to='user@domain/resource'
  type='(get | set | result | error) '>
  :
</iq>
```

Listing 2.1: Structure of an IQ message

An important thing to note is that when an IQ is to be directed from a client to another client the “to” field must specify a full JID containing a resource otherwise the server will process the IQ.

The following rules apply to all IQ messages[Saint-Andre, 2006a]:

1. The ‘id’ attribute is REQUIRED for IQ stanzas.

2. The ‘type’ attribute is REQUIRED for IQ stanzas. The value MUST be one of the following:
 - get – The stanza is a request for information or requirements.
 - set – The stanza provides required data, sets new values, or replaces existing values.
 - result – The stanza is a response to a successful get or set request.
 - error – An error has occurred regarding processing or delivery of a previously-sent get or set.
3. An entity that receives an IQ request of type “get” or “set” MUST reply with an IQ response of type “result” or “error” (the response MUST preserve the ‘id’ attribute of the request).
4. An entity that receives a stanza of type “result” or “error” MUST NOT respond to the stanza by sending a further IQ response of type “result” or “error”; however, as shown above, the requesting entity MAY send another request (e.g., an IQ of type “set” in order to provide required information discovered through a get/result pair).
5. An IQ stanza of type “get” or “set” MUST contain one and only one child element that specifies the semantics of the particular request or response.
6. An IQ stanza of type “result” MUST include zero or one child elements.
7. An IQ stanza of type “error” SHOULD include the child element contained in the associated “get” or “set” and MUST include an <error/> child.

XMPP Extension Protocols

Service Discovery XMPP is a very open format and as such there are many enhancements to it. An XMPP Extension Protocol (XEP) is an enhancement to XMPP. The process of formalizing these enhancements is undertaken by the XMPP Standards Foundation (XSF) (<http://www.xmpp.org/extensions/>).

An XEP of particular note is the Service Discovery Extension[Hildebrand et al., 2007]. This particular extension allows an XMPP client to discover information about the capabilities of other clients and what XMPP extensions they support.

Chapter 3

Protocols

3.1 Introduction

Instant messaging (IM) networks are very dynamic. Users are constantly adding and removing buddies to their contact lists. Users are also signing onto and off of the instant messaging network at varying rates to meet their needs. Network stability also plays a role in how a user uses an instant messaging application and network.

Some users stay signed into their IM network for as long as they can and only sign off due to reasons beyond their control (power outages, network failures, etc.), while other users only sign into the network periodically for brief periods of time.

The above problems provide a difficult challenge when attempting to construct a peer-to-peer based social network on top of the existing server/client based IM network, but one that is also solvable.

In order to solve these problems a communication protocol has been created for Semantic Social Instant Messaging. There are two important aspects of the protocol. The first is the initial creation of a users social network, and the second is the maintenance of the social network as users add and remove people to their roster.

The following statements are used throughout the next section and the meaning of them is important.

P_i is a Person object described using the FOAF namespace

P_i **foaf:knows** P_j This represents the statement that P_i knows P_j where $P_i \neq P_j$ and P_i, P_j are Person objects described using the FOAF namespace in an RDF document.

u_i represents a Jabber user

jid_i represents the Jabber Identifier of u_i

$R(u_i)$ is the roster of u_i and consists of a set $\{u_j | u_i \neq u_j, u_i \in R(u_j) \text{ and } u_j \in R(u_i)\}$

It is also important to note that throughout this section it is assumed that all users involved in the social network have clients which support the social networking features. In order to determine if a client does support Semantic Social Instant Messaging, service discovery is performed as outlined in 2.2.1.

3.2 Network Generation

The process of constructing a users social network begins when they initially sign onto an XMPP server using their Jabber/XMPP account. Assuming that the network has not already been constructed then the following describes the process that is followed in order to create the social network. If the network has already been constructed then the generation phase is skipped and the client goes directly into network maintenance mode (see 3.3). Algorithm 1 and Figure 3.1 demonstrate the steps followed in the initial creation of a users social network.

Algorithm 1: Sign-in

```

begin
  if model exists then
    foreach roster entry  $\in \{\text{roster}\}$  where  $|\text{roster entry friends}|=0$  do
      if roster entry online then
        SendKnowsRequest(roster entry);
  end

```

Assuming that the user, u_0 , is initially creating their social network then a new RDF model is created. A new Person from the FOAF namespace, P_0 , is created to represent the user u_0 . The jabberID property of the newly created Person is added and set to the JID of u_0 , jid_0 . Because the new RDF model is being created to represent the user u_0 , the PrimaryTopic value of the model, which specifies the particular person that this document is a description of, is set to be P_0 . This phase of social network generation does not require the user, u_0 , to be logged in because no network communication is required.

The next phase requires the user to be logged in. The user, u_0 , downloads their

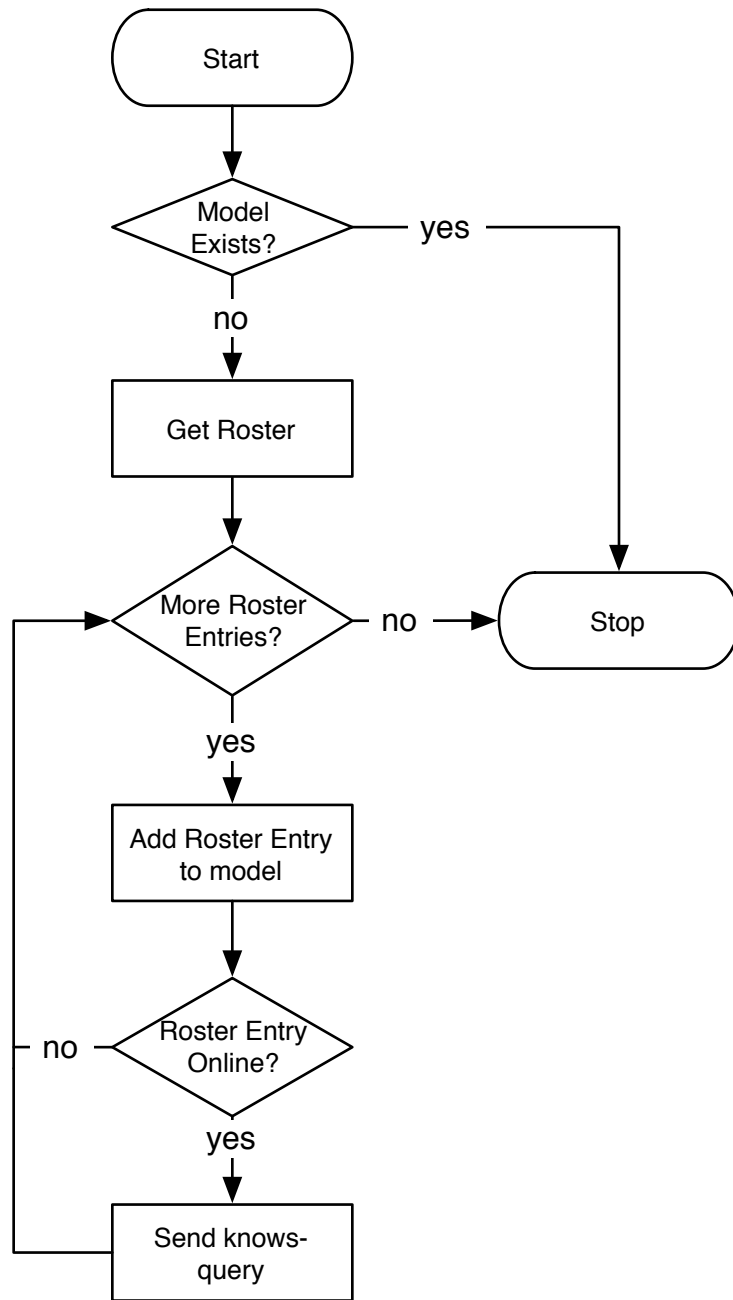


Figure 3.1: Flowchart showing the steps followed when initially constructing the social network upon first sign-on

roster, $R(u_0)$, from the server. The roster is the set $\{r_i | r_i \in R(u_0)\}$ where each r_i is a roster entry. For each of the entries, r_i , a new Person, P_i , in the FOAF namespace is created and added to the RDF model. Like before, the jabberID property of P_i is set to be the JID of r_i . Now, because $r_i \in R(u_0)$ then the knows property from the FOAF namespace can be set between P_0 and P_i . This results in a “knows” connection being formed between P_0 and P_i . This stage is demonstrated in Figure 3.2a

From $R(u_0)$ all of the entries that are currently online are selected to contain a new subset, $R(u_0)_{online} \subseteq R(u_0)$ (Figure 3.2b). Iterating over each $r_i \in R(u_0)_{online}$ an IQ message is sent from r_0 to r_i that is a request for a list of people who they know (Figure 3.2c). The structure of this message is outlined in Listing 3.1.

```
<iq id='UID' to='user@domain/resource' type='get'>
  <knows-query>
    <jabberId>JID</jabberId>
    :
    <jabberId>JID</jabberId>
  </knows-query>
</iq>
```

Listing 3.1: Querying for a list of knows

The important things to note here are that the “to” field of the IQ stanza is set to a full JID that contains the resource since we want the client to handle the request and not the server. Also, the “type” is a get because we are sending an information request. The next thing to note is that the single sub-element of the IQ message is a <knows-query> and it may contain any number of sub-elements of type <jabberId>. Each jabberId stanza contains a bare (sans resource) JID for which we are requesting information about. Initially the knows-query will contain only a single sub-element that is the JID of the person the request is being sent to.

Upon receiving a knows-query, the client will respond with an IQ result (or error if something went wrong) whose structure is outlined in Listing 3.2.

```
<iq id='UID' to='user@domain/resource' type='result'>
  <knows-results>
    <person jabberId='user@domain'>
      <knows>user@domain</knows>
    </person>
  </knows-results>
</iq>
```

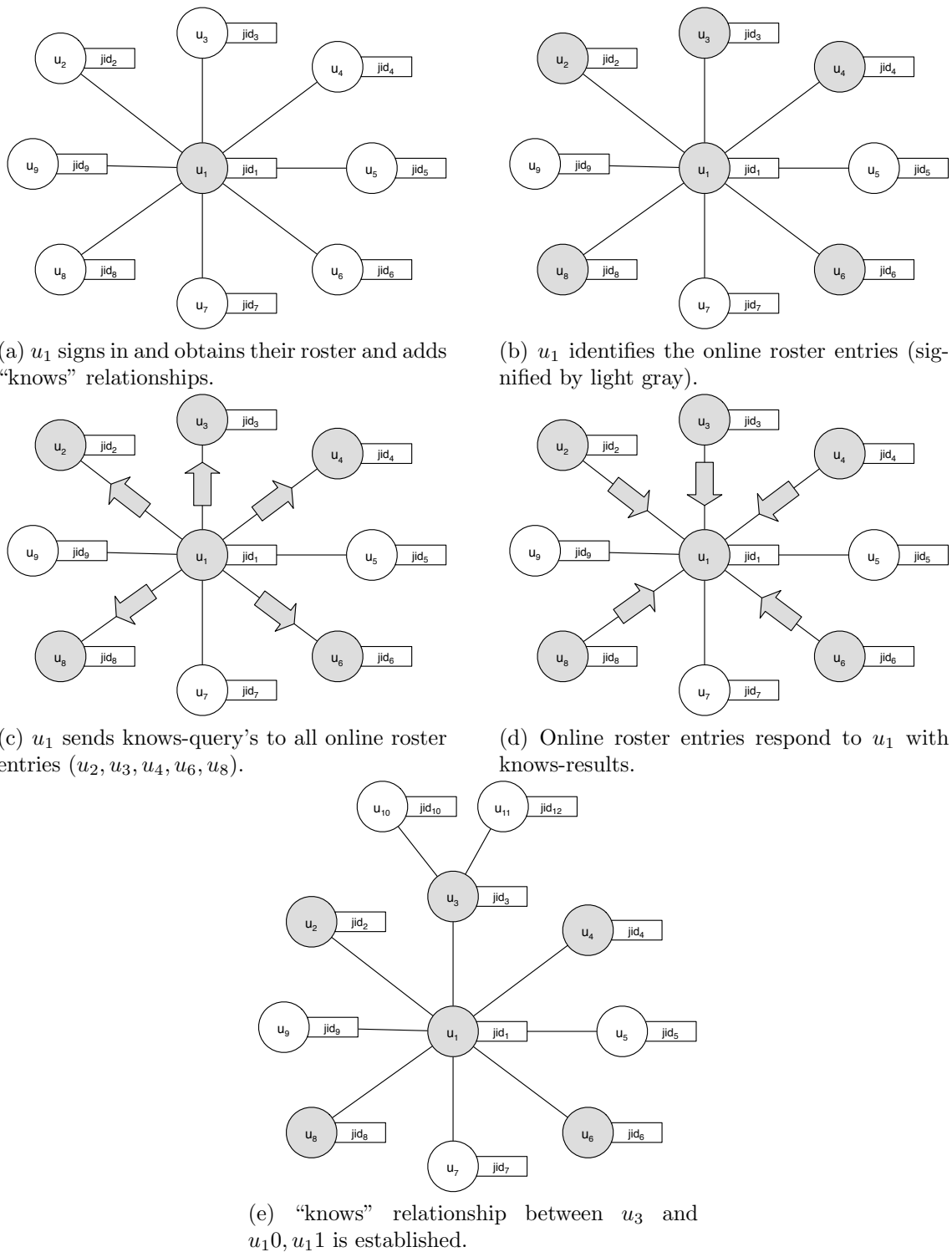


Figure 3.2: Demonstrating the process that occurs when a user, u_1 , begins to create their social network

```

    :
  </person>
  :
  <person jabberId='user@domain '>
    <knows>user@domain</knows>
    :
  </person>
</knows-results>
</iq>

```

Listing 3.2: Results of a knows-query

The “id” will be set to the UID that was contained in the original knows-query message and the “to” field will be set to the sender of the original knows-query. Next, for each `<jabberId>` that was contained within the knows-query we will be returning a `<person jabberId=“user@domain”>` where the `jabberId` attribute of the person stanza will match up to one of the `jabberId` elements in the knows-query. Contained within the person element will be a list of `<knows>` elements where each knows element is the JID of someone that the person knows. This packet is then sent back to the original requester.

When the original requester receives a knows-results message in reply to a knows-query it will process it as described in the flowchart in Figure 3.3 and outlined in Procedure 2: ProcessKnowsResults.

Procedure `ProcessKnowsResults(G, P_p, jid, K)`

Input: G , social network graph

Input: P_p , primary person

Input: jid , JabberID of person

Input: $K = \{j | j \text{ is a jabber ID, } j \text{ knows } jid\}$

$G = G + P_{jid}$

foreach $j \in K$ **do**

if $P_j \notin G$ **then**
 $G = G + P_j$

SendKnowsRequest(jid, K)

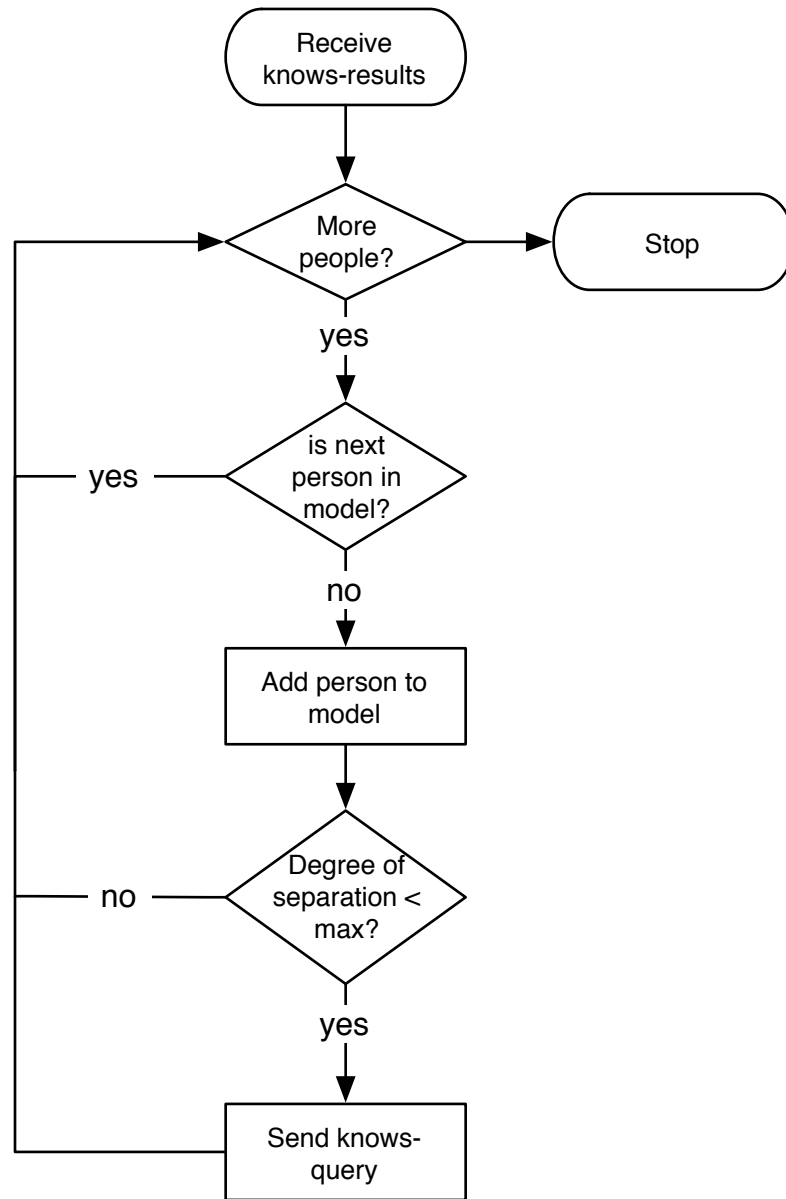


Figure 3.3: Flowchart showing the steps followed when a knows-results IQ is received

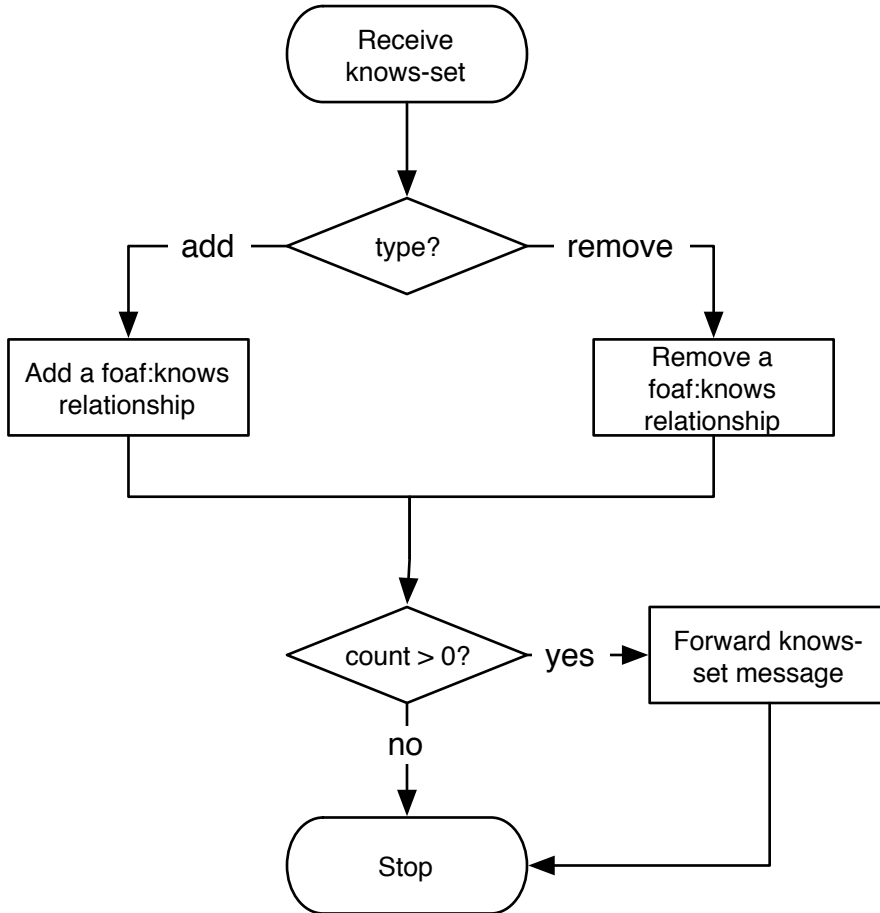


Figure 3.4: Flowchart showing the steps followed when a knows-set IQ is received containing either a buddy added or buddy removed

3.3 Network Management

Once the social network has initially been constructed it must also be managed. The management of the network involves maintenance on the graph that is a representation of a users social network. Since the network is constructed in a peer-to-peer manner this maintenance relies on a users buddies sending information to them when events that affect the social network occur. A users buddies rely upon them to do the same. This peer-to-peer nature means that each user participating in the social network also acts as a message relayer. When a user receives an important piece of information they pass the information along to all the members of their roster. Client applications implementing Semantic Social Instant Messaging should adhere to a few restrictions in order to prevent flooding the network with an excessive amount of

traffic.

The first restriction is that a receiver of an update does not send the update back to the initial sender. This is done to prevent an infinite looping of messages between two users and because sending the information back to someone who just sent it to you is entirely useless.

The second restriction is that each update has a time to live (TTL) value associated with it. This is a positive integer value that gets decremented by one every time it is forwarded. When someone receives a message that has a TTL of zero, they must not forward the message any further. It is up to the initiator of the message to determine the initial TTL value but a reasonable initial value is 6 which is derived from the small world phenomenon theory[Milgram, 1967].

The important events that occur in a social network are when two people add or remove each other from their rosters and also when a user receives a knows-result in response to a knows-query.

3.3.1 Adding a Buddy

When a user, u_i , adds a buddy, u_j , to their roster u_i must send out a notification to all $u \in R(u_i) - \{u_j\}$ so that their view of the social network is updated accordingly. This is done by sending an IQ message structured like Listing 3.3. In this IQ message the jabberId attribute in the `<person jabberId='user@domain'>` stanza is set to be the JID of the user u_i and the text of the `<added>` field is set to be the JID of the added user u_j . Procedure 3: AddRosterEntry outlines the simple set of steps that is followed when adding a user.

Figure 3.5 demonstrates how, when two users add each other to their rosters, the network is flooded (to an extent) informing the users within both u_i 's and u_j 's social networks of the change so that each user within the social network updates their view of the social network to reflect the change.

```
<iq id='UID' to='user@domain/resource' type='set'>
  <knows-changed ttl='6'>
    <person jabberId='user@domain'>
      <added>user@domain</added>
    </person>
```

```
</knows-changed>
</iq>
```

Listing 3.3: Sending a knows-changed when a user has been added to another users roster

An empty result IQ should be sent back

```
<iq id='UID' to='user@domain/resource' type='result' />
```

Listing 3.4: Results

Procedure AddRosterEntry(G, P_p, P_a)

Input: G , the FOAF graph

Input: P_p , the primary person

Input: P_a , the person to add

begin

$G \leftarrow G + \{P_a\};$

foreach *vertex* $u \in G - \{P_a\}$ **do**

 SendRosterAddition(P_u, P_a);

end

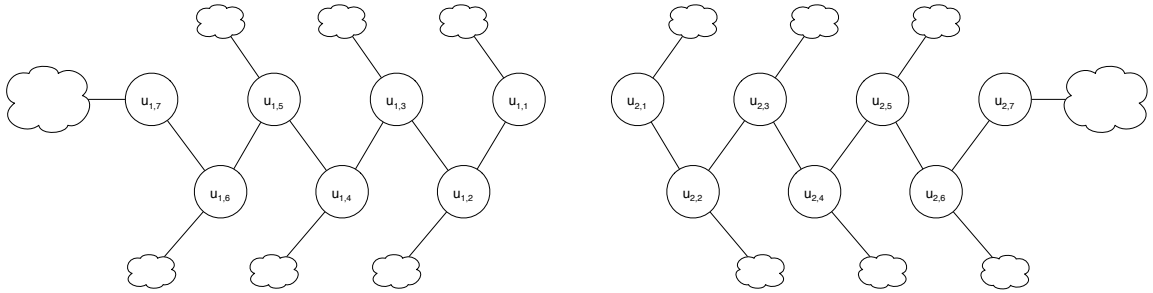
3.3.2 Removing a Buddy

Likewise, whenever we remove (delete) a buddy from our network we must also inform our buddies of this update so that it can propagate throughout the network.

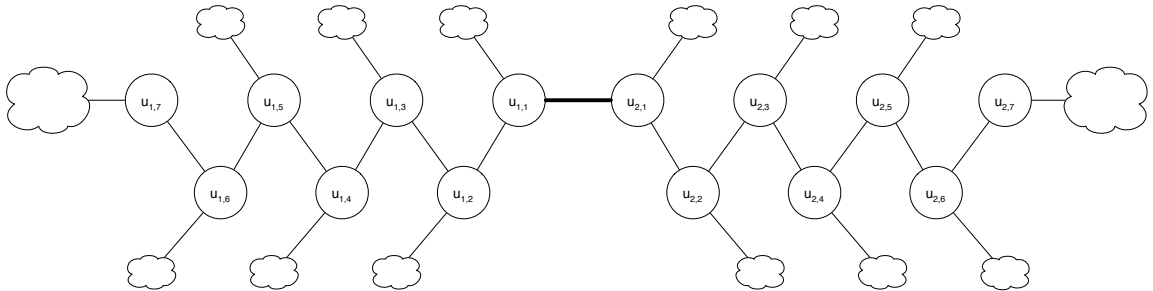
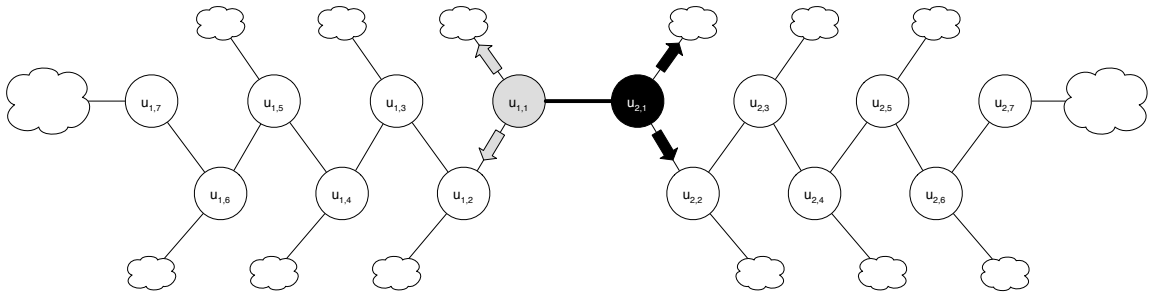
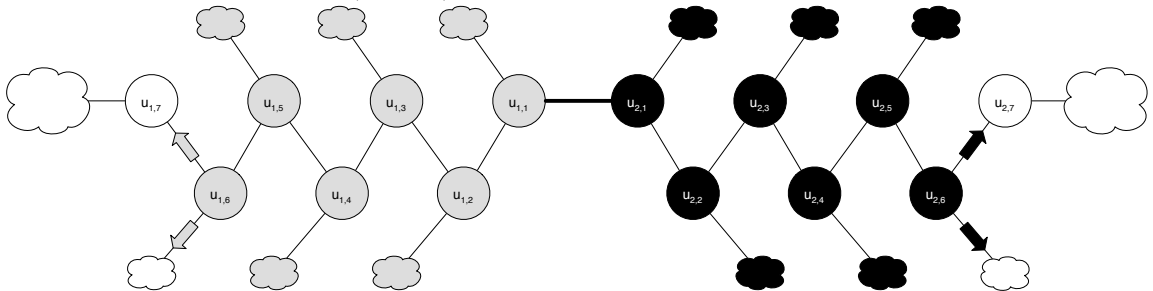
```
<iq id='UID' to='user@domain/resource' type='set'>
  <knows-changed ttl='6'>
    <person jabberId='user@domain'>
      <removed>user@domain</removed>
    </person>
  </knows-changed>
</iq>
```

Listing 3.5: Sending a knows-changed when a user has been removed from a users roster

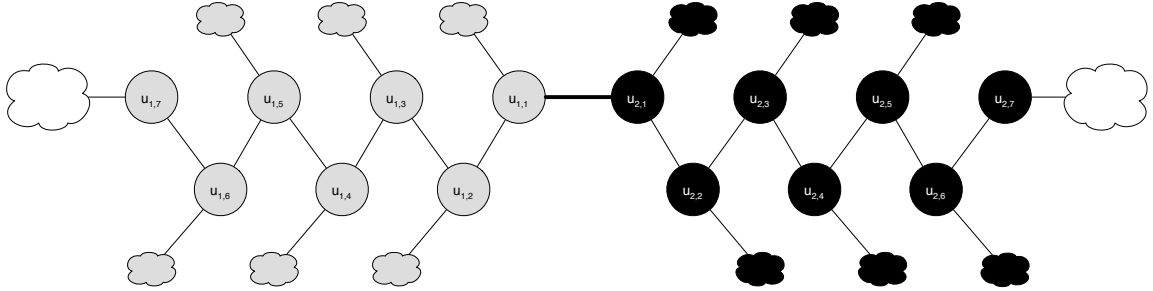
Figure 3.5: Adding a buddy



(a) Two disconnected social networks.

(b) $u_{1,1}$ and $u_{2,1}$ add each other to their rosters.(c) $u_{1,1}$ sends a knows-changed to $R(u_{1,1}) - u_{2,1}$ informing them of the addition and likewise $u_{2,1}$ sends a knows-changed to $R(u_{2,1}) - u_{1,1}$.

(d) The knows-changed messages propagate through the network.



(e) When the message reaches $u_{i,7}, i = \{1, 2\}$ the TTL value has been reached and no more messages are sent.

An empty result IQ should be sent back

```
<iq id='UID' to='user@domain/resource' type='result' />
```

Listing 3.6: Results

Removing a buddy from the network is slightly more complicated than adding a buddy. This is because a social network graph can have more than one path to a particular person and removing a node may or may not result in other nodes being removed. Other nodes may be removed if the only path from the primary person to them exists through the person that is being removed from the graph.

There are multiple ways of doing this, but for our purposes we do the following as outlined in Algorithm 4:

Given the FOAF graph that describes the primary person, P_p 's social network and a person to remove from the graph, P_r , remove P_r . This could potentially result in a disjoint graph because some set of people P_i, \dots, P_j that are friends with P_r and whom the only connection P_p had with was through P_r . Because there no longer is a connection to that set they are no longer in our social network and should be removed. In order to determine who to remove the process is simple, perform a breadth first search (Algorithm 6) on the social network graph using P_p as the source. As a result there is a distance and a parent associated with each person in the graph. Iterating over each person in the graph (excluding P_p) the distance and parent are examined. If the distance is ∞ or the parent is **nil** then there is no path to this person from P_p anymore so they are removed as well (see Procedure 4: RemoveRosterEntry). Figure 3.4 illustrates the process of two users removing each other from their rosters.

Procedure RemoveRosterEntry(G, P_p, P_r)

Input: G , the FOAF graph

Input: P_p , the primary person

Input: P_r , the Person to remove

begin

$G \leftarrow G - \{P_r\};$

 FoafCleanup(G, P_p);

foreach $u \in R(u_p)$ **do**

 SendRosterRemoval(P_u, P_r);

end

Procedure FoafCleanup(G, P_p)

Input: G , the FOAF graph

Input: P_p , the primary person

begin

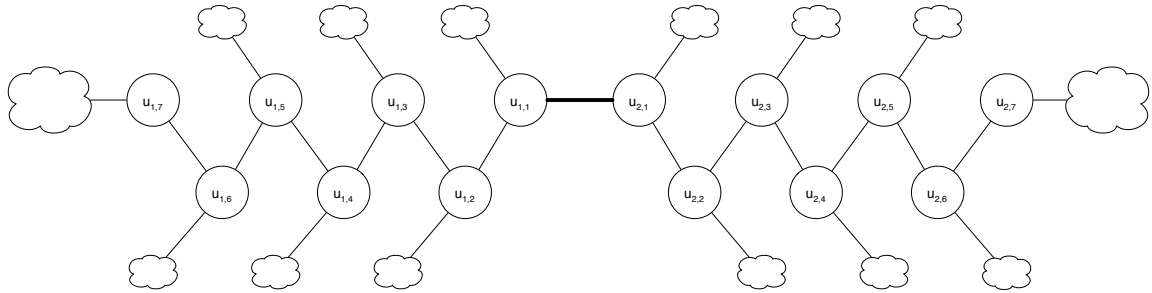
 BreadthFirstSearch(G, P_p);

foreach *vertex* $u \in G - \{P_p\}$ **do**

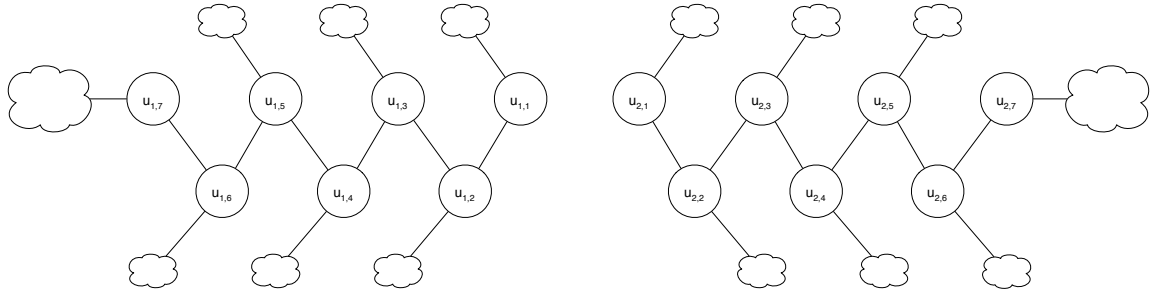
if $d[u] = \infty \parallel \pi[u] = \mathit{nil}$ **then**

$G \leftarrow G - \{u\};$

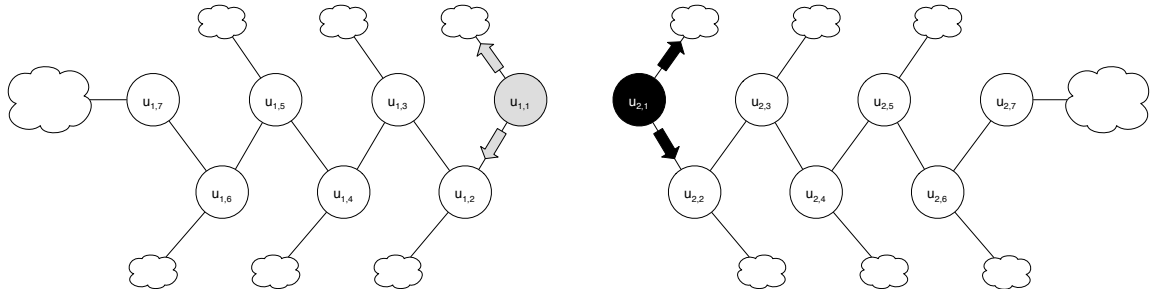
end



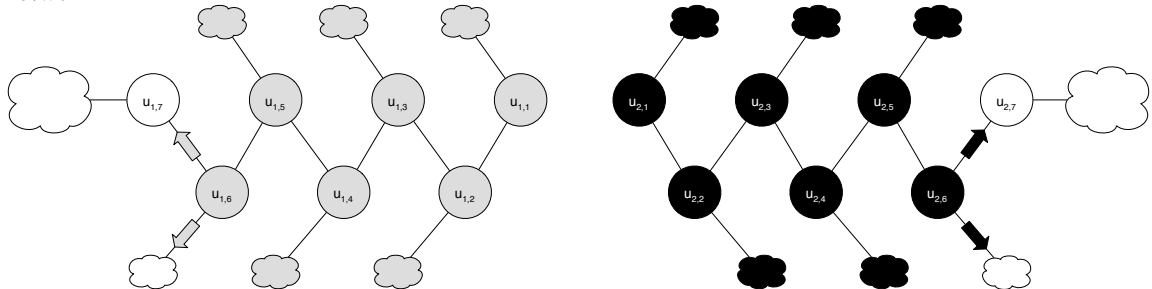
(a) Begin with a social network where $u_{1,1}$ and $u_{2,1}$ are about to remove each other from their rosters



(b) $u_{1,1}$ and $u_{2,1}$ remove each other from their rosters. $u_{1,1} \notin R(u_{2,1})$ and $u_{2,1} \notin R(u_{1,1})$



(c) $u_{1,1}$ begins to flood their social network with the update and $u_{2,1}$ does the same with their social network



(d) The buddy removal flood propagates through the network.

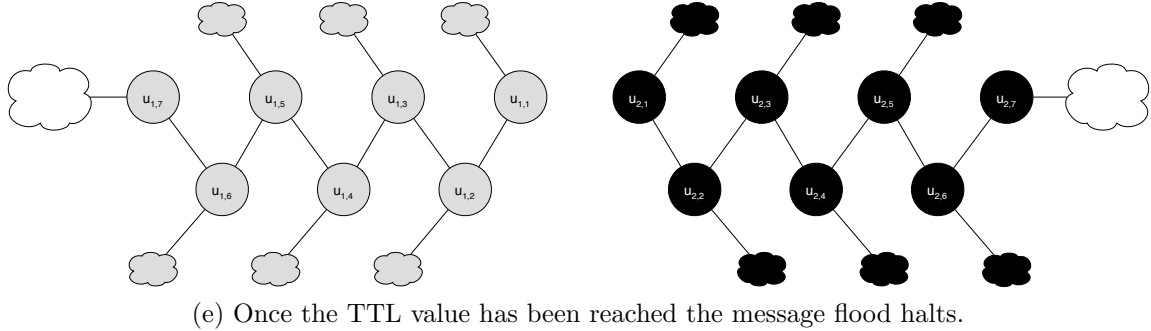


Figure 3.4: Removing a buddy

3.3.3 Pushing Data

Whenever a knows-query is sent from a user, u_i , to a user, u_j , they should respond to it with a knows-results. At this point u_i 's social network may change because new data is received. If this is the case then it is possible that other users that are connected to u_i will also need to update their data. But, because some user, u_k who's connection to u_j exists only through u_i there is no way for u_k to know of the update. That is why u_i is relied upon to “push” the new data out to these users. This is done by using a knows-push type message as shown in Listing 3.7. Figure 3.5 illustrates the process of a user sending a knows-request, receiving a knows-result and then pushing the results out to the rest of their social network.

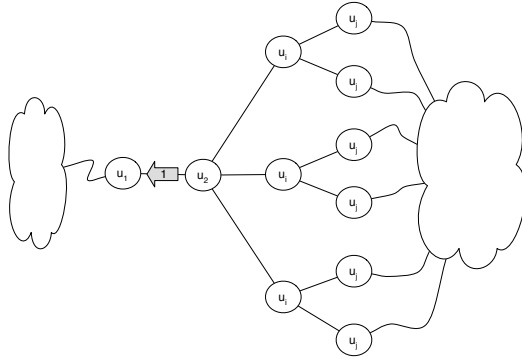
```

<iq id='UID' to='user@domain/resource' type='set'>
  <knows-push ttl='6'>
    <person jabberId='user@domain'>
      <knows>user@domain</knows>
      :
      <knows>user@domain</knows>
    </person>
  </knows-push>
</iq>

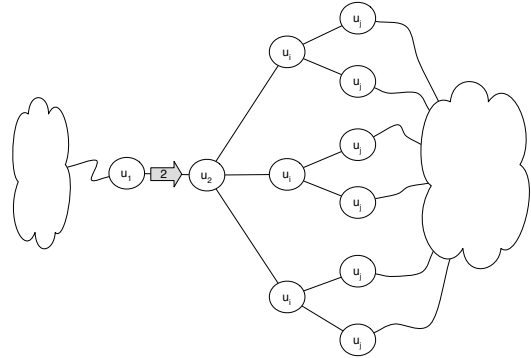
```

Listing 3.7: Sending a knows-push

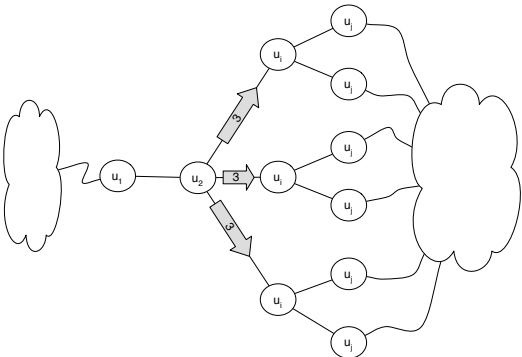
Figure 3.5: Demonstrating the process that occurs when a user, u_2 , sends a knows-query to another user, u_1 , and the resulting data flooding.



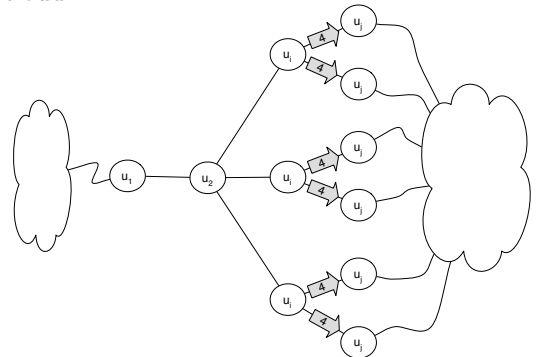
(a) Initial knows-query is sent from u_2 to u_1



(b) u_1 replies to u_2 with a knows-results made up of u_1 's social network designated by the left cloud



(c) u_2 receives results, processes and forwards a knows-push to each $u_i \in R(u_2) - \{u_1\}$



(d) Each u_i forwards the knows-push to each $u_j \in R(u_i) - \{u_i\}$ who in turn forwards it along where it reaches the right cloud and continues being forward until a TTL has been exceeded

3.3.4 Determining Degrees of Separation

When constructing the social network we are actually just constructing a graph where each vertex of the graph represents a person and an edge between two vertices represents the statement that the two people know each other. This allows us to use all existing graph algorithms on the social network. In particular, when trying to determine the degrees of separation between two people we can simply do a breadth first search. This also results in the shortest path from one person to another.

Doing a breadth first search offers two benefits, first, it allows us to determine the minimum degrees of separation between two people in the social network, and secondly, it gives us the chain of people through which the connection is made.

Procedure BreadthFirstSearch(G, P_p) [Cormen et al., 2004]

Input: G , the FOAF graph

Input: P_p , the primary person

 DEQUEUE **begin**

 foreach *vertex* $u \in V[G] - \{P_p\}$ **do**

 $colour[u] \leftarrow WHITE$;

 $d[u] \leftarrow \infty$;

 $\pi[u] \leftarrow \mathbf{nil}$;

 $colour[P_p] \leftarrow GRAY$;

 $d[P_p] \leftarrow 0$;

 $\pi[P_p] \leftarrow \mathbf{nil}$;

 $Q \leftarrow \emptyset$;

 ENQUEUE(Q, P_p);

 while $Q \neq \emptyset$ **do**

 $u \leftarrow \text{DEQUEUE}(Q)$;

 foreach $v \in Adj[u]$ **do**

 if $colour[v] = WHITE$ **then**

 $colour[v] \leftarrow GRAY$;

 $d[v] \leftarrow d[u] + 1$;

 $\pi[v] \leftarrow u$;

 ENQUEUE(Q, v);

 $colour[u] \leftarrow BLACK$;

end

Chapter 4

Results

The original hypothesis about the ability to use an instant messaging network in order to construct and maintain social networks has been verified through both an implementation and the techniques discussed throughout this paper. Social networks were successfully created and maintained via XMPP. The resulting social network was successfully stored as an RDF document using the FOAF ontology. Steps were also taken to help minimize the number of additional messages that are propagated through the instant messaging network. A brief analysis of the additional messages follows in section 4.1. Due to time constraints a method for searching the social networks has not yet been constructed. Queries can be performed on local RDF data but the ability to send a search throughout the network is currently not available.

4.1 Analysis of Additional Messages

4.1.1 Network Creation

Let $R(x)$ be the roster of user x then $|R(x)|$ is the size of the roster of the user x . Let $u_1, u_2, u_3 \dots u_n$ be people existing in a social network that starts at u_1 , consists of n people and the degrees of separation between any two users u_i, u_j is $\leq d$ where d is the maximum depth of the social network. Since there are n people within the social network $|R(x)| < n$ (since $R(x)$ will not contain the user x). Also, because of the way in which XMPP works, for every sent IQ message a result is sent back for a total of two messages. Thus, under the ideal circumstances where all users except x where x is some u_i , have already constructed their social networks and x has just begun the process, the total number of messages that x will generate during the construction phase is $2 \cdot d \cdot |R(x)|$, but we know that $|R(x)| < n$ and therefore the total number of messages is at most $2 \cdot d \cdot (n - 1)$ which is $\mathcal{O}(n)$.

In less ideal circumstances each user will have at most a 1-degree of separation

social network to begin with. The user u_0 will send a request some other user $u_i \in R(u_0)$ and u_i will respond with their 1-degree of separation social network. Then, when u_0 makes a request for information about u_j , where $u_j \in R(u_i)$, u_i responds with an empty set because u_i has not yet received further information about u_j .

Now, whenever u_j receives an update to their social network they will send a knows-push message that contains the newly received information.

4.1.2 Adding/Removing Buddies or Performing a knows-push

Suppose there are two separate and distinct social networks S_1 and S_2 where the user $u_1 \in S_1$ and $u_2 \in S_2$ and $|S_1| = n$ and $|S_2| = m$. Now suppose that u_1 and u_2 have added each other to their rosters and have thus become friends. A new social network is created from this where $S = S_1 \cup S_2$ and $|S| = (m + n)$. At this point u_1 will send a buddy-added message to all members of its roster excluding the new member u_2 and u_2 will do the same but also exclude u_1 . Now, each member of u_1 's roster receives the message for a total of $2 \cdot |R(u_1)|$ messages having been sent. Each $u_i \in R(u_1)$ acknowledges to u_1 the receipt of the message and also forwards the message along to each member of $|R(u_i)|$ and this process continues d times where d is the maximum distance from u_1 that the message should travel (a TTL value is attached to the message and decremented with each hop).

If S_1 is a complete graph then $2((n - 1)(n - 2) + (n - 1))$ messages will be sent/received, resulting in a total of $\mathcal{O}(n^2)$ messages. This assumes each client has no knowledge of the network, but, assuming that each u_i knows about the structure of the social network, which should typically be the case (as in most clients do although some may not have a full view yet) then a client can make a smart decision and notice that u_1 knows both u_i and u_j and the originator of the message is u_1 , therefore u_i will not need to forward the message to u_j (and vice versa) since they should both receive the message directly from u_1 . A complete graph where everyone knows the structure of the network is the best case and produces the fewest messages $\Omega(n)$

Since both buddy removal messages and knows-push messages occur in the exact same manner as adding a buddy the total number of messages generated by them is equivalent.

4.2 Implementation and Testing

The protocol, as described in Chapter 3, has been implemented in Java using Smack, an Open-Source XMPP library. As a result an XMPP client with a minimal feature set has been implemented to test the validity of the protocols. Through various testing and tweaking the methods for building and maintaining a social network were refined. Testing typically consisted of starting many clients with different rosters and performing add or remove buddy operations and verifying that a clients view of the social network was accurate.

Chapter 5

Future Work

The following section outlines future work that can be done on this project and the benefits that the enhancements may provide.

Implementing the features directly into the server would decrease the amount of message traffic generated because the social network would no longer be built using a peer-to-peer architecture. It would also allow additional web services to be built on top of it.

Utilizing the PubSub XEP[Saint-Andre and Smith, 2006] storage of social network data could be offloaded to the server and give the ability for the friends of a user to subscribe to their social network and automatically be informed of updates to the network by the server when updates occur.

There are practical usages to social networks beyond the fun that they can provide users. One such usage is in trust networks. Incoming instant messages could be examined based upon sender and if the sender is not contained within the social network then the assumption that the incoming message is SPIM (messaging SPAM) could be made and then actions, such as ignoring the message or adding the sender of the message to a privacy list, could be taken.

Appendix A

Compact Disc Contents

Provided with this paper is a CD containing the source code for an experimental implementation of the protocols discussed within this paper. The Java SDK 5.0 or higher is required to compile and the JRE 5.0+ is required in order to execute it.

In order to build the application it must be built with javac with all required additional packages in the Classpath.

In order to run the application the `net.gdashoff.semsocim.Main` class needs to be executed. It takes two optional command line arguments, `-debug` and `-path`. The `-debug` flag puts the client in debug mode and displays a message view window that allows the user to see all incoming and outgoing XML messages. The `-path` argument takes a filesystem path of where it should save configuration data as well as the generated RDF FOAF documents.

Source code is optionally available via SVN from <http://svn.g-Off.net/SemSocIm>.

Bibliography

- [Brickley and Miller, 2006] Brickley, D. and Miller, L. (2006). FOAF Vocabulary Specification. <http://xmlns.com/foaf/0.1/>.
- [Classmates Online, Inc., 2007] Classmates Online, Inc. (1995-2007). Classmates. <http://classmates.com>.
- [Cormen et al., 2004] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2004). *Introduction to Algorithms*, chapter 22.2 Breadth-first Search, pages 531–532. MIT Press, 2nd edition.
- [Herman, 2007] Herman, I. (2007). W3C Semantic Web Frequently Asked Questions. <http://www.w3.org/2001/sw/SW-FAQ>.
- [Herman et al., 2007] Herman, I., Swick, R., and Brickley, D. (2007). Resource Description Framework (RDF). <http://www.w3.org/RDF/>.
- [Hildebrand et al., 2007] Hildebrand, J., Millard, P., Eatmon, R., and Saint-Andre, P. (2007). XEP-0030: Service Discovery. <http://www.xmpp.org/extensions/xep-0030.html>.
- [Iannella, 2001] Iannella, R. (2001). Representing vCard Objects in RDF/XML. <http://www.w3.org/TR/vcard-rdf>.
- [LinkedIn Corporation, 2007] LinkedIn Corporation (2003-2007). LinkedIn. <http://www.linkedin.com/>.
- [Manola and Miller, 2004] Manola, F. and Miller, E. (2004). RDF Primer. <http://www.w3.org/TR/rdf-primer/>.
- [Mark Zuckerberg, 2007] Mark Zuckerberg (2007). Facebook. <http://www.facebook.com>.
- [Milgram, 1967] Milgram, S. (1967). The Small World Problem. *Psychology Today*, pages 60–67.
- [MySpace, 2007] MySpace (2003-2007). MySpace. <http://myspace.com/>.
- [Saint-Andre, 2005] Saint-Andre, P. (2005). XEP-0144: Roster Item Exchange. <http://www.xmpp.org/extensions/xep-0144.html>.
- [Saint-Andre, 2006a] Saint-Andre, P. (2006a). Extensible Messaging and Presence Protocol (XMPP): Core. <http://www.ietf.org/internet-drafts/draft-saintandre-rfc3920bis-00.txt>.

[Saint-Andre, 2006b] Saint-Andre, P. (2006b). Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. <http://www.ietf.org/internet-drafts/draft-saintandre-rfc3921bis-00.txt>.

[Saint-Andre and Smith, 2006] Saint-Andre, P. and Smith, K. (2006). Personal Eventing via Pubsub. <http://www.xmpp.org/extensions/xep-0163.html>.