UPPSALA
UNIVERSITET

IT 12 013
Examensarbete 15 hp
Maj 2012

A Peer-To-Peer based chat system

Tommy Mattsson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

A Peer-To-Peer chat system

Tommy Mattsson

Chat systems has been around since even before the modern Internet came into existence. Today there are a variety of chat systems of varying complexity. Most of these chat systems require that a user creates an account to be used with the provided service. There has been a lot of research when it comes to networks, network communication and the use of Peer-To-Peer software has become increasingly popular. This report describes one possible way to implement a chat system using a Peer-To-Peer model instead of a client/server model without the need to sign up for the specified service to be able to use it.

Handledare: Justin Pearson
Ämnesgranskare: Karl Marklund
Examinator: Olle Gällmo
IT 12 013
Tryckt av: Reprocentralen ITC

Contents

1 Introduction **5**
1.1 Background . . . 5
1.2 Problem definition . . . 5
1.3 Motivation . . . 6
1.4 Related work . . . 6

2 Peer-To-Peer **7**
2.1 Gnutella . . . 8
2.2 Bit-torrent . . . 9

3 Tools and protocols **10**
3.1 Erlang . . . 10
3.1.1 Concurrency . . . 10
3.1.2 Message passing . . . 10
3.1.3 Robustness . . . 10
3.1.4 Network communication . . . 11
3.1.5 Distributed Erlang . . . 11
3.1.6 Libraries . . . 12
3.1.7 Operating system support . . . 12
3.2 External tools and libraries . . . 12
3.2.1 Simplicity . . . 12
3.2.2 Makefile . . . 12
3.2.3 Considered libraries . . . 13
3.3 Protocols . . . 13

4 Implementation **14**
4.1 Assumptions . . . 14
4.2 Asynchronous message passing . . . 14
4.3 Application structure . . . 15
4.3.1 Overview . . . 15
4.3.2 Contact list file . . . 17
4.3.3 Online users with unchanged locations . . . 17
4.3.3.1 Duplicate sockets between nodes . . . 17
4.3.4 Online users with changed location . . . 18
4.3.4.1 Network efficiency . . . 21
4.3.5 Backup location . . . 21

4.4 Operating system support . . . 22

5 Possible improvements/future work 23

5.1 Unimplemented parts . . . 23

5.1.1 Record last known location . . . 23

5.1.1.1 dets . . . 23

5.1.2 Registering some nodes that is not alone . . . 23

5.2 Bugs . . . 24

5.2.1 Socket negotiation . . . 24

5.2.2 Random crashes when a node disconnects . . . 24

5.2.3 Backup location . . . 24

5.3 Application structure . . . 25

5.3.1 Overall structure . . . 25

5.3.2 Distributed Erlang . . . 25

5.4 Data caching . . . 25

5.5 Request message time-out . . . 26

5.6 Backup location . . . 27

5.6.1 Possibility to add additional backup locations . . . 27

5.6.2 Node registration negotiation . . . 27

5.7 Security . . . 28

Appendices 29

A Installation guide 29

Bibliography 30

Chapter 1

Introduction

In this report techniques for developing a Peer-To-Peer chat system is studied and an analysis is performed of difficulties and problems for developing such a system. For this report a Peer-To-Peer chat system application was developed, and its design is described in section 4. The design of this application uses one of several different possible Peer-To-Peer solutions and a few of the existing solutions for Peer-To-Peer communication is described in section 2.

1.1 Background

When one thinks of a chat system one will most likely think of one of the more common chat systems like MSN Messenger, Skype or Google Talk. All of these require the user to create an account and sign in to be able to use their service. A lot of people uses different chat systems to talk with other people. This require them to keep track of many different accounts to be able to talk to everyone they want to talk to. What if there existed a system that require no previously created account, no login and uses no central server to enable users to communicate with their friends? The application described in this report is an attempt to address this question.

1.2 Problem definition

Creating a simple chat system using a client/server model is very easy, and can be done in less than a day if the programmer is used to network programming. All that is needed is a server application which can supply the location of a specific user to a client requesting that user and a client application that will connect to a given location for a user.

A chat system with a Peer-To-Peer model is a lot more difficult to create. The main issue with such an application is the fact that the location for anyone is simply not known for sure. There are several techniques for Peer-To-Peer communication, which is described in section 2.

1.3 Motivation

The idea was to develop a simple to use, free chat system with translation capability of sent and received message. This was to be achieved by developing a chat system that used a Peer-To-Peer communication method in order to be server-less and use some free translation service. The translation part was originally supposed to be performed by Google Translate. Unfortunately the Google Translate API was taken down as a free service and is now only available as a paid service [6]. Because of this the translation part of the chat system was abandoned.

1.4 Related work

When it comes to Peer-To-Peer systems there are numerous services each with different implementations. This report describes briefly how two of these methods work, namely Gnutella and Bit-torrent. Bit-torrent is today one of the most commonly used and widely known Peer-To-Peer protocols and it is actually responsible for around $43 - 70\%$ of all network communication as of February 2009 [10].

When it comes to chat systems using Peer-To-Peer communications the number of applications is not as many. A few systems known for sure that use a Peer-To-Peer communication method is Skype[11] and μChat which is an application developed for use with μTorrent [13]. μChat is actually the closest thing in existence (to the current knowledge of the author) to the application developed for this report. The main difference is that μChat has a chat room which this application does not have. μChat also have an initial base to stand on (in the form of *trackers*, which is further described in section 2.2) which this application of course does not have since it is a brand new application.

Chapter 2

Peer-To-Peer

Peer-To-Peer communication mechanisms are used in order to get away from the drawbacks a client/server model suffers from. The following points are a few of the drawbacks that the classic client/server model suffer from:

- It has a single point of failure. Essentially this means that if the server goes down the service is not able to function properly if it is even able to function at all.
- The server need to serve a possibly high amount of work because all requests in the system is directed to the server, so the load of the server is high.

Peer-To-Peer is often abbreviated as *P2P* and this abbreviation is used throughout the rest of this report. P2P communication is interesting and potentially effective mainly because it takes away the single point of failure drawback that the client/server application suffers from and it gives an application a distributed communication instead. This enables *peers* to communicate directly with each other instead of having a server in the middle which all communication passes through. This makes it possible for a properly implemented P2P application to tolerate loss of multiple nodes and each node will possibly perform a balanced amount of work (in relation to the total amount of work for all nodes).

Compare this with a client/server application that has a single central server, meaning that if the server goes offline it is not possible to use the application at all. The central server will have a high load of work to be performed since it will need to serve possibly all requests, while in a P2P application each node will serve a portion of the total amount of requests since the work is distributed on each node.

The central idea of a P2P system is to not use a centralized server for serving requests. Some P2P systems is cheating a bit in some sense by using a server to get the P2P communication started. For instance the Bit-torrent protocol use this approach, which is described in section 2.2.

2.1 Gnutella

Gnutella is an early P2P communication mechanism. A Gnutella node needs to know the location of at least one other Gnutella node in order to be able to work at all. Therefore a Gnutella node is initialised with a set of working nodes. A Gnutella node is connected to its working set and each connected node has its own working set of Gnutella nodes. No Gnutella node needs to have the same working set as another Gnutella node. This way of connecting creates a mesh of connected nodes, see figure 2.1.

The protocol uses flooding to find desired data among connected nodes. A node *A* wants to get some data from the cloud so A makes a request for this data from all its connected nodes and each of these nodes will make requests to all of their connected nodes. In order to not letting the flooding actually flood the network forever each request is given a *Time To Live* (TTL) which is an integer number that is decremented at each node it reaches. A request is not further replicated from a node when the TTL for a request reaches zero. When data is found inside the Gnutella network, the data is sent directly to the requesting node. For a graphical illustration of the data propagation to the requesting node, see figure 2.2

This mechanism of connected nodes gives a node the possibility to be connected to just hundreds of nodes but actually be able to reach data from millions of nodes in the Gnutella cloud.[7]

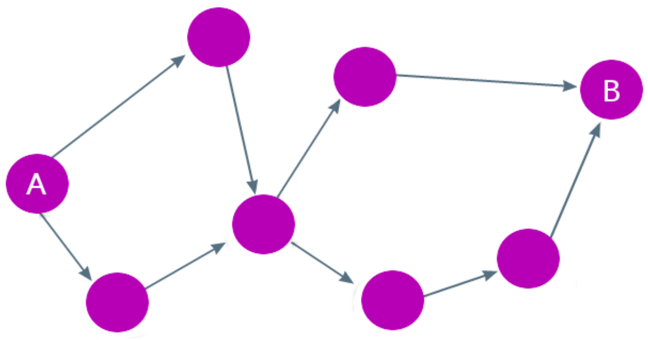


Figure 2.1: An illustration of a Gnutella mesh

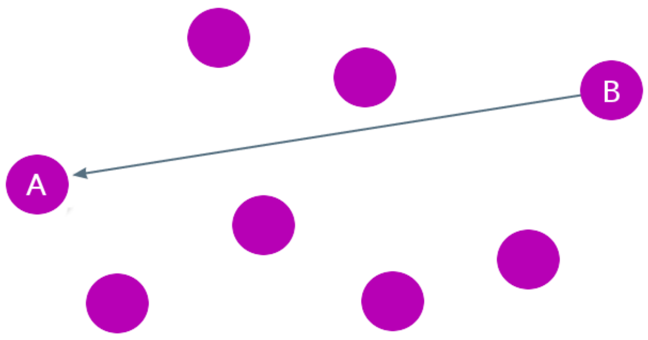


Figure 2.2: An illustration of found data inside a Gnutella mesh

2.2 Bit-torrent

Bit-torrent is for the most part used for file-sharing, so file-sharing will be used as an example when describing Bit-torrent. The protocol uses a central server (called a *tracker*) that keeps tracks of all connected peers that have a file, see figure 2.3 for an illustration. It is possible to have different trackers at different servers, which connects a node to different clouds of peers. Each file is divided up to a number of chunks and a peer can have all (called a *seeder peer*) or just some of the chunks (called a *download peer*). The basic idea is that even though a peer do not have the entire file, the peer can still help out in the cloud by serving the chunks of the file it does have. See figure 2.4.

So to summarize, the trackers keep tracks of peers that have have a file (or parts of a file) and a down-loader requests a list of these peers (called *swarm list*). The down-loader gets the list of these peers and starts connecting to them. After connecting with some or all of the peers in the received swarm list the down-loader starts making requests for parts of the file (*chunks*). The down-loader will receive chunks from other peers and will share these chunks with other peers that is interested in those chunks. So even though a peer have not received an entire file it can still help the Bit-torrent network by sharing the chunks it do have access to.[8]

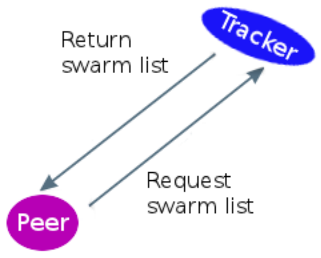


Figure 2.3: An illustration of the the swarm list request from the Tracker

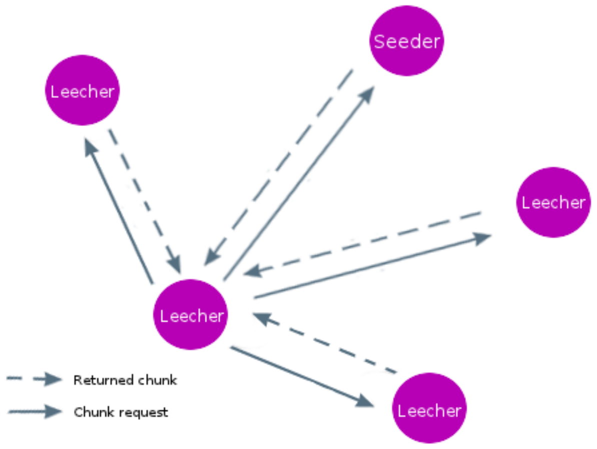


Figure 2.4: An illustration of requesting and sharing chunks of data with other nodes

Chapter 3

Tools and protocols

3.1 Erlang

The implementation of the server-less chat was made in Erlang [1]. The following points are the main reasons to why Erlang was the programming language of choice for developing this application.

3.1.1 Concurrency

Erlang is a language quite suitable for concurrency. The Erlang language only uses processes, instead of a mixture of processes and threads like for instance the C programming language, to achieve concurrency. These processes are very lightweight, in fact they are called "lightweight processes" [1], which makes it possible to create a large number [2] of Erlang processes without lowering system performance drastically.

3.1.2 Message passing

The way processes is implemented in Erlang has the side-effect that no data is shared between processes, i.e. there is no shared memory between processes. To be able to share data, Erlang processes need to use message passing. By using message passing there is no need to use locks to avoid data races.[1]

3.1.3 Robustness

Erlang is a very robust programming language in the sense that it provides advanced failure handling mechanism. For a good implementation of a system it can be possible to achieve a system uptime of 99.999% [12]. For a network application robustness can be very important and Erlang is one of the better choices when it comes to robustness.

[1]Erlang, `www.erlang.org`

3.1.4 Network communication

Erlang has good support for network programming and handles network connections better than most programming languages. A comparison between the web-servers Apache[2] and YAWS [3] (a web-server written entirely in Erlang) shows that YAWS is by far more effective than Apache [5].

Erlang has a library for dealing with TCP connections and another library for dealing with UDP connections which both contain useful and simple to use functions for handling network connections. Both TCP and UDP connections can be configured to be *active*. In Erlang this means that a process that owns an active socket receives messages in the exact same way that message passing is performed between processes. Message passing between process on the same machine is performed with the following structure:

```
receive
    atom_message ->
        perform_some_action;
    {several, parameters, message} ->
        perform_some_action
end.
```

When a socket is active, messages is not received by using the `gen_tcp:recv` function. Instead messages coming in on a socket is automatically directed to the process that owns that socket. The structure for detecting the messages looks like this:

```
receive
    {tcp, Socket, Msg} ->
        perform_some_action
end.
```

As can be seen, the structure looks exactly the same. This makes it possible to use the same receive structure for both receiving messages over a TCP socket and messages coming from other processes on the same machine. This mechanism of active connections can be very powerful when designing a chat system, and is used extensively throughout the application described in this report. More details is given in section 4.2.

3.1.5 Distributed Erlang

Besides being lightweight, robust and being able to handle a large number of network connections, Erlang actually has built in support for developing distributed applications [3]. It is for instance possible to give names to nodes and use those names along with some built in functions to provide communication between nodes, this could make developing distributed Erlang applications much easier.

The Distributed Erlang functionality was actually not discovered until very late during application development so it has not been used for this application. Further discussion about Distributed Erlang will take place in section 5.

[2]Apache, `http://httpd.apache.org/`

[3]YAWS, `http://yaws.hyber.org`

3.1.6 Libraries

One of the minor reasons for choosing Erlang, but still a contributing factor is that Erlang has a wide range of built in libraries. This makes developing applications in Erlang a little easier since one do not need to reimplement commonly used functions. An example is the library `lists`. Here is a short list of some of the functions that the lists library implements:

- `lists:all` - Returns a boolean depending on a given predicate and if that is true or not for all elements in a list.
- `lists:delete` - Deletes a given element from a list.
- `lists:map` - Returns a new list where each element is taken from an older list and a given function has been applied to each element in the old list.
- `lists:nth` - Returns the nth element in a list.

3.1.7 Operating system support

Possible wide-spread distribution of the application was desirable, so a very important requirement was that the application should work with and perform well on a range of systems. Erlang is able to run on Windows, Linux/UNIX, Mac and even Android [4], so it is absolutely possible to use an application written in Erlang on a wide range of systems.

3.2 External tools and libraries

In this section an explanation is given to why certain tools and libraries was used for development and the structure of the application.

3.2.1 Simplicity

No mandatory external tools or libraries is used for running the application. This choice was done in order to keep the application simple and to be able to spread the application as wide as possible without worrying about if a system has support for a certain library or tool.

3.2.2 Makefile

The use of a Makefile [5] is not needed but in order to simplify compiling and running of the application a Makefile was constructed and used throughout development. In addition to the Makefile, an Emakefile was used. Emakefile is a Make utility for Erlang which uses Erlang syntax and is used by a regular Makefile to provide an easier way to compile Erlang code according to your preferences and complexity. It is of course possible to compile and run the application without using the Makefile, but it is overall easier to use the Makefile to compile rather than to compile each file on its own.

[4]Erlang Embedded, `http://www.erlang-embedded.com/`

[5]GNU Make, `http://www.gnu.org/software/make/`

3.2.3 Considered libraries

At the very beginning during development of the application described in this report the following library was considered to be part in the application structure.

Erlang XMPP

The external library EXMPP (Erlang XMPP)[6] was initially considered for aiding the clients to pass messages to each other efficiently. It was at first rejected because it simply did not work on the systems available for development, which turned out to be installation issues and nothing more. A later try to get it to work was successful but at that time it was rejected because it would have caused to much re-structuring of the already produced code which there simply was not enough time to do.

3.3 Protocols

EXMPP was considered but rejected as the protocol for use to deliver messages between two clients. Of course not only chat messages need to be passed between nodes, which will be described in more detail in section 4. So to keep things simple and in order to provide reliable transfer of all messages the protocol used for network communication is TCP. Erlang has a library called gen_tcp which implements a number of important functions for TCP communication.

[6]EXMPP, `https://support.process-one.net/doc/display/EXMPP/exmpp+home`

Chapter 4

Implementation

4.1 Assumptions

The only really big assumption this application makes is that there is only one user with a specific user-name inside the system and that user actually is the correct user. In order to be able to focus on the P2P part this assumption is made because that it is a valid assumption for a P2P system with a *secure channel*. The notion of a secure channel guarantees that the user actually is the user it claims to be and that the communication is secure. The application described in this report has no implemented security since that is not part of the P2P problem. Security is of course important in such a system but it will not be discussed here because it is outside the scope of the problem.

4.2 Asynchronous message passing

First there is need to mention that all communication between any two or more nodes is completely asynchronous. All messages is guaranteed to arrive since the application uses TCP sockets but no message being sent needs to be acknowledged on a conceptual P2P communication level, so no assumptions are made on the order of arriving messages. This means that any type of message can come before any other type of message. There is a few exceptions to this rule, which is described when appropriate.

In section 3.1.4 active sockets was mentioned. This construct makes it possible to mix receiving messages from a network communication with messages between processes on the same machine. This active socket mechanism is the functionality used throughout the entire application to provide asynchronous communication both between nodes on different computers but also between processes on the same machine.

4.3 Application structure

4.3.1 Overview

The application is divided into the following major modules:

main.erl

The application is started from the module main.erl (from here on called *Main*). After start-up Main will create a listener process (*Listener*), a connection handling process (*Router*) and a main user interface process (*MainUI*). The functionality of Main consists of the following parts:

- Notifying Router and MainUI when a contact goes offline.
- Notifying MainUI when a contact comes online.
- Start a client process (*Client*) when a contact is found to be online.
- Make sure that a connected contact actually is a desired contact and close the connection if the contact is not desired.
- Shutting down a Client when a contact goes offline.

main_UI.erl

MainUI is a very simple user interface. It is possible to view online contacts and start a chat with an online contact. To start a chat it is possible to double click on the name of the contact or by selecting the contact and click on a button just below the online contacts list. See figure 4.1 for an illustration of the user interface. To quit the application there are two ways of doing it (actually three if shutting down the Erlang shell is taken into account):

- Clicking on a button with the text *Quit*.
- Clicking on the *X* button of the MainUI window.

router.erl

The most complex file in the application. Only a brief overview of the functionality of this module is given here, a more detailed description is given in the section 4.3.4. Router is the module which takes care of all P2P functionality. It tries to connect to users at their last known location and if that fails it will try to find a user with a P2P communication method.

listener.erl

Listener does one thing and one thing only, accept incoming TCP connections (*sockets*) and give control of these to Router.

client.erl

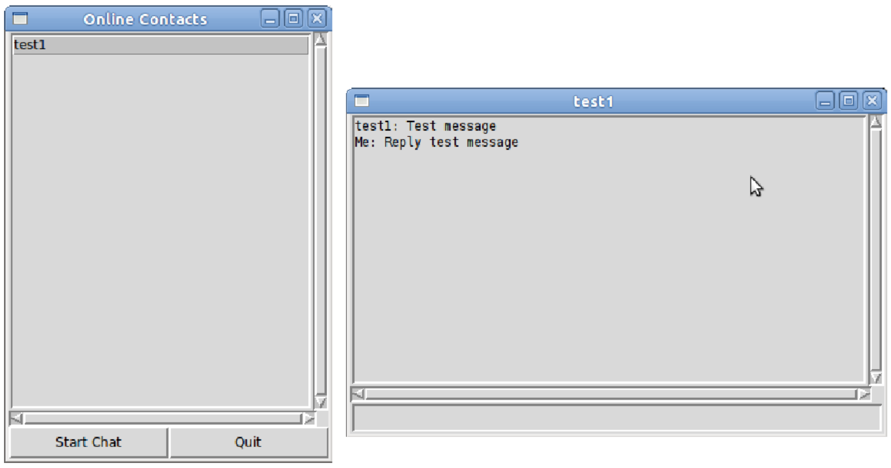
Client is fairly simple, it keeps track of who it is connected to and sockets it can send messages on, each client owns the socket it can send messages on. So for each contact that comes online a Client is started to handle messages sent to and from that contact. Client also starts a ClientUI.

client_UI.erl

A simple user interface for a Client process. It provides a box for printing messages and a box for viewing sent and received messages. See figure 4.1 for an illustration of the user interface. A ClientUI is started in two cases:

- The user wants to chat with someone and double clicks on a contact in MainUI.
- The Client receives a chat message from the other end as is supposed to display the message.

There are also a few support modules in the completed application in order to keep the code more simple and make the logic of the application more clear. Detailed set-up and execution instructions are described in appendix A. Here follows a description of the conceptual structure of the two major parts of the application and an additional part that is needed. See figure 4.2 for an illustration of the application structure.



```
test1: Test message
Me: Reply test message
```

Figure 4.1: The application user interface. MainUI to the left and ClientUI to the right.

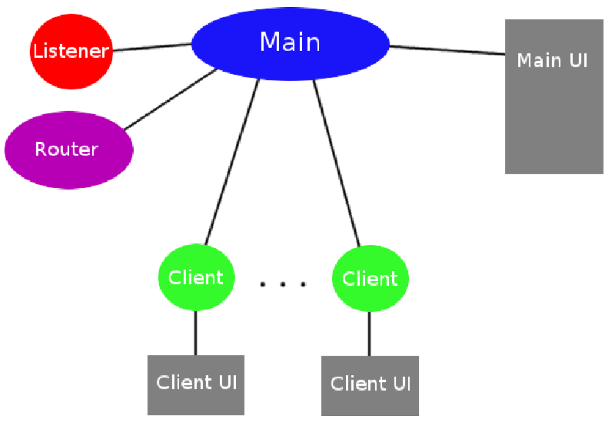


Figure 4.2: An illustration of the application structure

4.3.2 Contact list file

A user of a chat system wants to connect to a number of previously added contacts. These contacts resides inside a file with a file-name of the following form `contact_list_$USER_NAME.contacts` where `$USER_NAME` is the chosen user-name the user want to use.

On a Linux/UNIX system this file is located inside `/home/$USER/` where `$USER` is the user-name of the logged in user. The contact list file uses Erlang syntax to simplify reading from it. Erlang has file-read functions that handles a file written in Erlang syntax.

As writing to files can be complicated and possibly prone to errors, there is no support for adding contacts through the user interface. The only way to add a new contact is to manually write the name of a user and add an IP address for that contact (can be any valid IP address if the IP is not known) into the contact file. The application also needs to be restarted in order for the application to see the new contact. This choice was made in order to be able to focus on parts related to the P2P functionality.

4.3.3 Online users with unchanged locations

Router is the module taking care of connecting users and finding users with unknown location through P2P communication. The very first thing Router does after starting up is to try and connect to all contacts that was read by Main from the contact file and passed to Router to search for. Router tries to connect to each IP address associated with each contact. If a connection is successful an exchange of identification is performed to establish if the user on the other end of the connection is a desired user.

If it turns out that it is not a desired user the connection is closed. In the case that the user on the other end is a desired user there are two sub-cases:

- It is the expected user.
- It is another desired user.

Both cases is treated in the same way because the application does not assume that the user on the other end is the correct user, so regardless which user it is (as long it is a desired user) the application just replace the user-name associated with that connection and notifies the MainUI that this contact is now online and able to receive messages.

After Router has tried connecting to all users and is done with a possible exchange of user-names it will start to try and find missing users with a P2P communication method, which is described in section 4.3.4. It is possible for a node to be unable to connect to any desired users causing that node to be alone. A solution to this problem was developed for this application and is described in section 4.3.5.

4.3.3.1 Duplicate sockets between nodes

Because all messages is sent asynchronously the situation can arise that two nodes tries to find each other at their last known location independently. The timing can possibly play out so that each node is allowed a socket to the other node before either of them has shared its ID with the other. This will cause two (or possibly more, but no more than two should appear from what I have been able to deduce logically) sockets to exist between two nodes. This is a waste of resources so the Client will start a negotiation in order to close all but one socket. For this purpose the following mechanism was developed to be used as a socket closing negotiation:

1. A node *A* senses that there is more than one active socket for a contact.
2. A then randomly chooses one of the sockets to be closed. The Client marks this socket as *closed* and sends a `CLOSE` message on that socket.
3. The other node, *B*, also perform step 1 and 2, possibly at the same time.
4. If a `CLOSE` message is received on a socket marked as closed, the receiver of the `CLOSE` message simply closes the socket, because this means that the two nodes agreed on the same socket to close.

It does not matter if it is A or B that receives the `CLOSE` message first. Both A and B will check if it is okay to close the socket and close it if it is okay. The node that did not close the socket will get an error message saying that the socket is closed and will take action accordingly.

4.3.4 Online users with changed location

In order for a node to find users that has moved to another location this application uses a flood mechanism similar to the one used in the Gnutella protocol described in section 2.1. Router sends out messages on the form `P2P::FIND::ID::$ID::TTL::$TTL`, where `$ID` is the user-name of a desired contact and `$TTL` is the *Time To Live* integer number described in section 2.1. This type of message is sent out to all currently online contacts. See (4) in figure 4.3 for an illustration of the propagation of this type of message. The prefix `P2P` is used for all messages concerning the P2P mechanism for finding missing contacts. TTL is set to the integer number of 6. The number 6 was chosen based on the "Six Degrees of Separation" theory [9]. It is an unproven theory but should overall be an okay choice since there is some kind of thought behind this number.

If a user is found, a message is sent back to the nodes requesting that user-name. The message is then propagated back the way it came from in order to deliver information about the location of the user to the nodes requesting that information. See (7) in figure 4.3 for an illustration. All nodes that propagated the message saves information about the location of the found user in order to speed up future searches for this user from other nodes. The form of this message is `P2P::FOUND::ID::$ID::LOC::$IP` where `$ID` is the id of the requested user and `$IP` is the IP address for that user. If a user is found but it is known that this contact is offline, a message on the form `P2P::OFFLINE::ID::$ID` is sent back in the same manner. If a user is not found, a message on the form `P2P::NOT_FOUND::ID::$ID` is sent back in the same manner as for the other two messages described above. An implicit assumption to these messages is that a `P2P::FIND::ID` message was received at some earlier time to make a request for that id.

If the flooding mechanism finds the information for a request, the information is sent back to all nodes requesting it. These nodes will then try to establish a connection with the node on that location. See (8) in figure 4.3 for an illustration. It is possible for a node to go offline (or be disconnected) which will cause the sent back information to be incorrect. If this happens the functionality described in section 4.3.3 will take care of the connection failure that will occur and just ignore that it failed and continue to search for that contact.

There is a possibility that a situation arises where a node has received the desired information for a contact but there is still `FOUND` or `NOT_FOUND` messages being propagated between other nodes. For an illustration of such a scenario see (9) of figure 4.3. Some optimizations for situations like this is described in section 4.3.4.1.

In order to keep the logic clear for which nodes are unknown, known and desired contacts several different lists is kept inside Router. The following lists for users exists:

Contacts
: All desired users for this node. A status field is kept for each user signifying if the status for that user is online, offline or undefined (that a user has not yet been found either online or offline).

Known
: Users with a known location but not inside the set of desired users (users inside the list `Contacts`) for this node.

Unknown
: Incoming connections where the user-name is yet to be determined. So in this list there will exist information about connections for which no user-name has yet been specified and information about connections where the user-name need to be confirmed. This is because the application does not assume that a successful connection for a specific user-name actually means that the correct desired user is in that location.

Look For
: Users that other nodes want with a currently unknown location.

Contacts Look For
: Users that this node want with a currently unknown location.

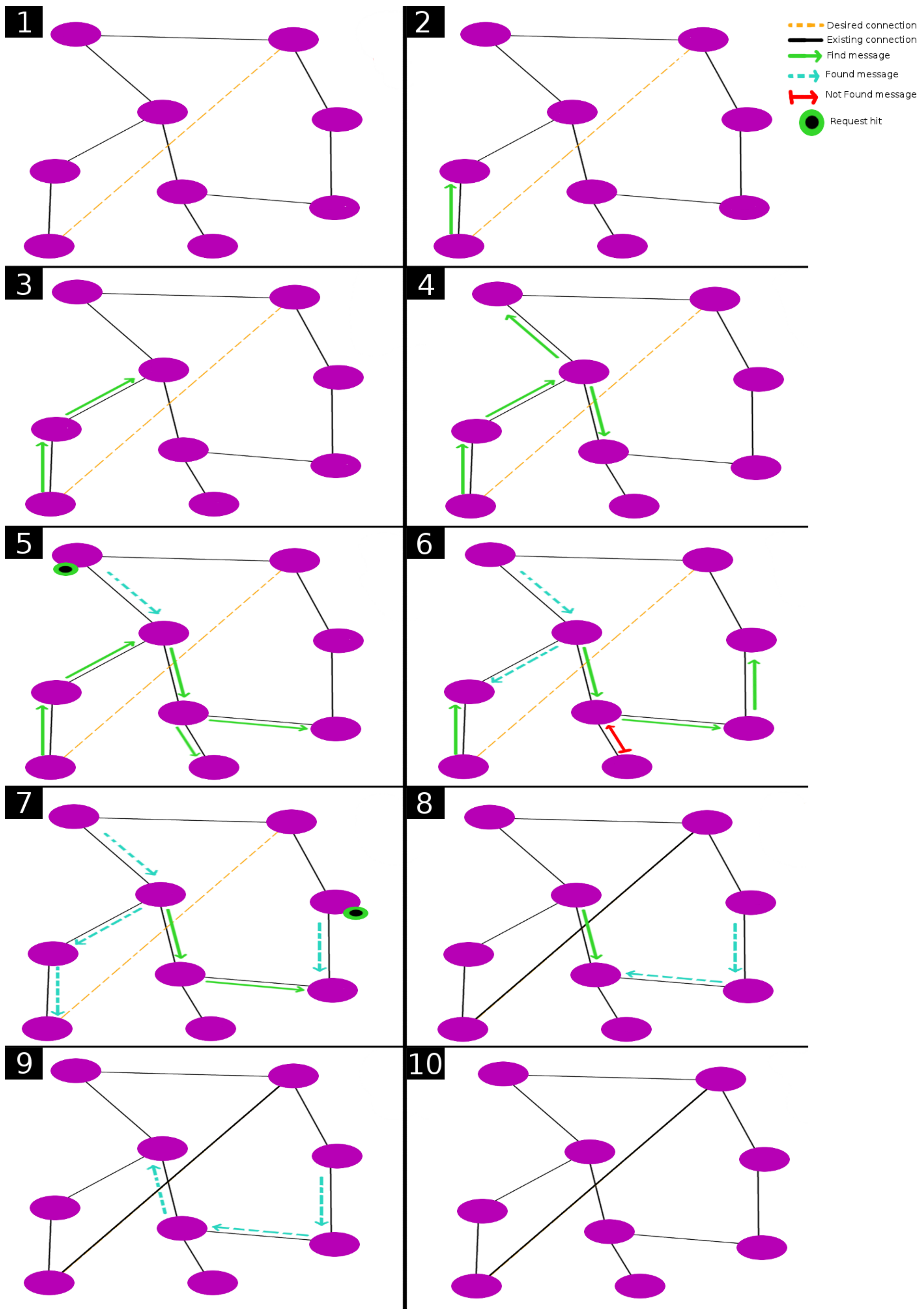


Figure 4.3: An illustration of the implemented flood mechanism

4.3.4.1 Network efficiency

In order to be network efficient, a mechanism is needed to not send too many unnecessary messages. To that end the following mechanism was developed for this application. Two lists is kept for each user-name in the lists `Known`, `Look For` and `Contacts Look For`. The lists kept for each user-name are the following:

I Request From
: A list with user-names of your connected contacts that requests has been sent out to concerning a specific user.

Request From Me
: A list with user-names of your connected contacts that is interested in a specific user.

When Router receives a `P2P::FIND` message and the requested id is not known on this node (and $TTL > 0$), Router will propagate new requests to all its online contacts, but to be effective Router will only send a request to a connected node if that user is not inside the list `I Request From`, this is used to reduce the number of duplicate messages being sent requesting a specific user.

During the P2P find mechanism Router will receive these messages `P2P::FOUND::ID`, `P2P::OFFLINE::ID`. These messages are all propagated to the users residing inside the list `Request From Me`. These users are the ones interested in a specific id and should be notified about the status of that id. The list for that user is set to empty after all nodes interested in that users has been notified. An important thing to notice here is that even though a response message was sent to the users requesting a specific users, duplicate messages coming from other nodes will not be passed forward. This is because the `Request From Me` list will be empty so there will be no one to send a response message to. This is illustrated in (9) and (10) in figure 4.3.

The last message yet to be described is the `P2P::NOT_FOUND::ID` message. This message will not set the `Request From Me` list to empty, it will only remove the user-name of the node supplying the message from the `I Request From` list.

These two lists is not kept for entries inside the `Unknown` list since those entries is for unconfirmed users. Which means that the user-name for an entry is either wrong or simply does not exist because it has yet to be received.

4.3.5 Backup location

It is possible for a node to not find any of its desired contacts causing that node to be alone. In this case there is one of two possible things to do:

1. Wait until someone finds you.
2. Connect to an undesired node and communicate with that node in order to try and find some desired contact.

In this application the second possibility was chosen in order to solve this problem without actually ignoring it, which is the case with choice 1.

A backup location is a node where users is able to register that they are alone and get another registered node from to start finding its desired users. A backup location has the exact same logic as any other node with one small difference, a backup node is not supposed to try and connect to a backup node, so a small flag is kept inside the application in order to signify if the node is a regular node or a backup node. In the current version only one backup location exists, which is hard-coded into the application, but further extension of this concept is discussed in section 5.6. Negotiation with a backup node requires special types of messages in order to distinguish that a node wants to get a node registered at a backup location. The currently implemented and used messages are:

`P2P::REG::ID::$ID`
: A message signifying that a user wants to register at a backup location so that an alone node can get some other node to start communicating with. This message is both a request to register and a request to get a registered node. This is because the logic of actually registering nodes that is not alone is unimplemented. A possible way to implement this is described in section 5.6.

`P2P::NODE::ID::$ID::LOC::$IP`
: This type of message is passed from a backup location to a requesting node to signify that this node was picked among the backup locations known nodes.

4.4 Operating system support

The intended support for the application was supposed to be most modern desktop operating systems such as Linux, Mac and Windows and possibly mobile operating systems like Android. Due to lack of time, researching how to find and use a common location for the contact file on a Windows was not performed, therefore the ensured support for the application is only Linux and UNIX systems using Erlang version R15B. Due to lack of knowledge and access to a Mac machine, support for Mac is not tested.

Chapter 5

Possible improvements/future work

5.1 Unimplemented parts

The parts described in this section are things that has been considered conceptually but not developed as part of the application.

5.1.1 Record last known location

A very important part for this application to be efficient between closing and starting the application the last known location of each contact needs to be written down to a file as soon as a location is known. The application is able to find a location of a user that has changed location, but it does not record it at the moment of writing this report. This is because writing to files can be difficult and prone to errors. Reading from a file using Erlang was easy, but during development writing to a file has not been touched.

5.1.1.1 dets

Erlang contains a functionality called "dets", which is a "Disk Based Term Storage" [4]. It provides a way to store records on disk with a fast access time. This functionality could possibly be used in order to avoid dealing with file writing within the application but still be able to record the last known location of a node that went offline or was disconnected.

This is completely untested functionality. However from the description it seems that it could make the application both more clear and perform overall better since saving of the last known location for a contact is stored on disk and kept there.

5.1.2 Registering some nodes that is not alone

In order to avoid confusion for people used to dealing with Erlang, the use of the word 'register' in this section (and every other section as well) does not refer to the internal Erlang mechanism for registering a process or a node. Instead 'register' is used as a way to describe that an instance of the application communicates with a backup location in order to say that the application instance has no online contacts.

The idea was actually to have some nodes registered at backup locations in order to keep some nodes from each cluster of nodes registered in order to be able to connect clusters of nodes. The thought behind this is that the situation can possibly arise that a number of contacts is connected in a mesh like in figure 2.1. But that there can be several such meshes where nodes in one mesh can never connect to nodes in the other mesh, because there is no connection from any node in one mesh to any other node in the other mesh.

This functionality is unimplemented because an additional small assumption is made:

- The backup locations will most likely never be alone anyway because the backup locations is just a regular chat system that wants to connect to its own contacts.

One possible way is to register all regular nodes at backup locations. This will however cause backup locations to be subjected to higher data load since a lot more requests will be made to the backup locations and force them to store information about a larger number of nodes. A possible solution to this is described in section 5.6.2.

5.2 Bugs

This section describes found bugs in the application.

5.2.1 Socket negotiation

The situation of multiple sockets between two nodes described in section 4.3.3 works partly. The Clients is able to sense that there is multiple sockets between two nodes and is able to choose one of them to close and send a `CLOSE` message to the other node. Something however is wrong when receiving, interpreting or acting upon a `CLOSE` message because a node dies after receiving such a message.

5.2.2 Random crashes when a node disconnects

Most of the time it is possible for two nodes to connect to each other and have one of the nodes go offline and online and they will connect to each other successfully each time. Every now and then however the timing plays out so that if a node goes offline, some or all nodes connected to the node going offline will also die. The cause for this is a bug in updating the internal records for the application, but the exact location of the bug is not known.

5.2.3 Backup location

The communication with a backup location currently does not work. A backup location is able to be started as a backup location and is able to work as a regular node, but if a node tries to connect to a backup location it fails. When two nodes knows each other and one of them is a backup location the communication works without any problem. No crashes has been detected when trying to connect to a backup location, but no successful communication in this capacity was recorded when testing.

5.3 Application structure

In this section a description is given for possible ways to improve the application structure.

5.3.1 Overall structure

The chosen structure was picked in order to give each client control of all sent and received messages to enable Router to focus its computation on the P2P finding mechanism. In retrospect this may not have been the best idea. Right now the application keeps information about contacts both inside Main, Router and Client which makes the synchronisation of the information between the processes a bit more complicated than it need to be. A possible alternative structure could be to keep Main, MainUI, Listener, Client, ClientUI and move the functionality of Router inside Main and let Main keep control of all sockets.

This scheme will put more strain to Main and in order to do as little work with messages not concerning the P2P find mechanism, all chat messages will just immediately be passed to the Client that is interested in messages from a specific socket.

5.3.2 Distributed Erlang

The use of the Distributed Erlang functionality would likely cause the application to be less prone to error and possibly make the code for the application easier to implement and clearer to understand for people other than the developer of the application. This is of course only conjecture since it was not used during development of the application. From the looks of the Distributed Erlang functionality it would likely produce a less complicated application by using this functionality.

5.4 Data caching

In the current version of the application caching of users and their locations is done at each node that receives some information about another user regardless if the information concerns a desired user or not. This was developed in order to speed up future searches for that user from other nodes. There is one problem with this scheme however which is that the user a node have information about may go offline and if that happens the data that was cached is no longer valid.

One way to solve this is to periodically connect to the all nodes inside the `Known` list and verify that the users are still there. There exist three possible scenarios:

- The user is still there. In this case everything is fine and the node keeps its cached data.
- The desired user is not there, but another user is there. The data is invalidated because it is no longer accurate.
- A connection attempt was unsuccessful, so there is nobody on the other end. The data is invalidated in this case as well.

5.5 Request message time-out

A node finds contacts that has moved by sending `FIND` request messages (see section 4.3.4). The best-case scenario is that a request is answered with a `FOUND` response. The worst-case scenario however is that a request is never answered at all because the `FIND` message propagation is aborted at some point because nodes goes offline. This case is not taken care of in the application. The following scheme is proposed as a possible way to keep track of requests that are never answered:

1. For each desired contact yet to be found where a request has been sent out, keep a time-stamp of when the request was sent out.

2. Define a time-out interval.

3. When responses and requests comes to a node, check the time-stamp for a desired contact and see if $currentTime - timeStamp \geq time - out$. If the subtraction on the left hand exceeds the time-out value on the right side, new requests need to be sent out for that contact.

4. If no requests or responses arrives at all from any node, a "real" time-out is needed to be able to verify if there is any request messages that has timed out.

Keeping a time-stamp in Erlang is easy, there is a function called `now` that calculates the current time. A time-out value can be kept in an include file with a time value in milliseconds. The reason for having it in milliseconds is because of a specific erlang structure which will be described next.

To be able to handle time-outs of request messages even when there is no incoming request or response messages some time-out mechanism is needed. In Erlang the message passing `receive` constructs described in section 3.1.4 has an `after` construct which can be used for this purpose. The structure looks like this:

```
receive
    {regular, message} ->
        perform_some_action;
    {tcp, Socket, Msg} ->
        perform_some_action
after
    300000 -> % 300000 milliseconds = 5 minutes
        perform_some_action
end.
```

If no message is received within the specified time-out the code in that block is executed. The `after` construct uses a value specified in milliseconds[1]. This functionality can be used for the loop for Router to overcome the problem when no response or request messages is received. When the loop time-outs there is no need to do the calculation described in step 3. The node already know that no message has arrived within the specified time-out so a new request can immediately be sent out for all contacts yet to be found.

[1]The time-out value in the code above is just an example, no research has been performed on what a good time-out value should be.

5.6 Backup location

In this section possible improvements for the backup location functionality is described.

5.6.1 Possibility to add additional backup locations

The current layout contains only one single backup node. This is not in style with the P2P philosophy since it is a single point of failure. Since all nodes operate in the exact same way, it is possible for any node to become a backup location. In order for this scheme to work a node need to broadcast to all its contacts that it is in fact a backup location. One way to do this could be to attach a flag in each message signifying that it is a backup location so that connected contacts can record the location for that node so it can connect to it if needed.

In order for this scheme to be efficient the information about a backup node need to be spread to other nodes besides the nodes that the backup location is already connected too. How to do this efficiently has not been figured out. Two possible ideas was thought of however:

- Periodically broadcast all known backup locations to all currently connected nodes and pass these messages on (also with the same TTL mentioned before to keep messages from replicate forever).
- It is probable that a backup location will have a bunch of nodes inside its `Known` list so it could periodically try and connect to all of those nodes and immediately send a message like `BACKUP::ID::$ID::LOC::$IP`.

The above methods are just conjecture and has not been tested or even been closely examined to determine if either of the ideas is in fact viable.

5.6.2 Node registration negotiation

In order to keep down the number of registered nodes at backup locations, some mechanism of negotiating which nodes is going to be registered at a backup location is needed. One idea was to have special *backup node registration* negotiation messages floating around in order to remedy this problem, but this will cause the network to be filled with a lot of unnecessary traffic which is not desirable. Another idea was to attach the number of online contacts for a node together with all messages being sent and to only keep nodes with the highest amount of online contacts registered at backup locations.

The second idea will not suffer from unnecessary traffic since it will just use the already generated find mechanisms traffic to do extra work. The hard part that was never completely figured out for this application was how to actually determine which nodes should be registered at a backup location. Contacts can come online and go offline often which in turn will cause the number of online contacts for a node to shift often. This will cause a lot of traffic to backup locations which also is not desirable.

5.7 Security

The current version of the application provides no security. A possible security mechanism with RSA keys is one way to solve this (and probably one of the more secure ways) to obtain security for the application. To determine that a user is the user it claims to be, the public key of the user could be saved when first adding that user and perform a handshaking mechanism after getting the id of that user. Something like this is proposed as a possible handshaking mechanism:

1. Node A sends a random message encrypted with the stored public key of node B.
2. Node A saves the message that was sent.
3. Node B decrypts this message and sends a message back using the public key of A.
4. Node A decrypts the message from B and checks if it matches the stored message from point 2.
5. If the match is okay then node B is the user it claims to be. If the match is not okay, then the user is not the desired user.

It is a mechanism that potentially can provide security for the application. But it is unknown to the writer if this mechanism is enough to provide authenticity and confidentiality for the communication between nodes.

Appendix A

Installation guide

In order to execute this application the following is needed:

- Erlang version R15B.
- A system supporting the location `/home/$USER/`.

On a Linux/UNIX system it is possible to use a supplied Makefile to start the application. With the supplied Makefile there is two ways of starting the application:

Regular node
Performed with the following make command from the command line:
`make start ARG1=$USER` where `$USER` is the user-name the user want to use.

Backup node
Performed with the following make command from the command line:
`make start ARG1=$USER ARG2=backup`, where `$USER` is the user-name the user want to use.

It is of course possible to execute the application without the Makefile. To start the application without the Makefile, the following step by step procedure is needed to be performed:

1. Compile all the .erl files. There is two ways of doing this.
   (a) Use the erlang command line compiler with the following command:
   `erlc $FILE` where `$FILE` is the .erl file to be compiled with a relative or absolute path to the file.
   (b) Start an Erlang shell and use the following built in command:
   `c($FILE).` where `$FILE` is the `.erl` file to be compiled with a relative or absolute path to the file.
2. Start an erlang shell or use a previously started shell.
3. Type in the following command: `main:start([$ID, $TYPE]).` where `$ID` is the desired user-name and `$TYPE` is either the Erlang atom `regular` or `backup` to specify if the application should be treated as a backup or a regular node.

Bibliography

[1] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in Erlang*. Prentice Hall, second edition, 1996.

[2] Ericsson. Performance Measurements of Threads in Java and Processes in Erlang. `http://www.sics.se/~joe/ericsson/du98024.html`, April 2012.

[3] Erlang. Distributed erlang. `http://www.erlang.org/doc/reference_manual/distributed.html`.

[4] Erlang. dets. `http://www.erlang.org/doc/man/dets.html`, April 2012.

[5] Ali Ghodsi and Joe Armstrong. Apache vs. Yaws. `http://www.sics.se/~joe/apachevsyaws.html`, April 2012.

[6] Google. Google Translate API FAQ. `https://developers.google.com/translate/v2/faq`, April 2012.

[7] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[8] James F. Kurose and Keith W. Ross. *Computer Networking a Top-Down approach*. Pearson Addison-Wesley, fifth edition, 2010.

[9] David H. Roper. Six degrees of separation. *Our Daily Bread*, February 2012. `http://odb.org/2012/02/04/six-degrees-of-separation`.

[10] Hendrik Schulze and Klaus Mochalski. Internet study 2008/2009. `http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf`.

[11] Skype. About Skype. `https://support.skype.com/en-us/faq/FA10983/What-are-P2P-communications`, April 2012.

[12] Mike Williams. Why did we create erlang? ACM SIGPLAN workshop on Erlang, August 2003. `http://www.erlang.se/workshop/2003/paper/mikesrationale.pdf`.

[13] Patrick Williams. myChat: We just need each other. `http://blog.bittorrent.com/2011/06/30/uchat-we-just-need-each-other`, April 2012.