UPPSALA
UNIVERSITET

IT 11 065
Examensarbete 30 hp
Augusti 2011

Context aware file sharing protocol designed for mobile devices

Niclas Axelsson

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

Context aware file sharing protocol designed for mobile devices

Niclas Axelsson

Future applications for embedded devices are expected to share some common functionality. The ability to create mobile mesh networks is one of them and will enable application de- velopers to create ground breaking innovative applications which has not been possible yet. Having an Erlang application which provides just that functionality could foster innovative application development and convince developers to use Erlang for their work.

Handledare: Jan Nyström
Ämnesgranskare: Mikael Pettersson
Examinator: Anders Jansson
IT 11 065
Tryckt av: Reprocentralen ITC

Contents

Acknowledgements

First of all I want to thank my supervisor, Henry Nystrom, for giving me valuable tips and input. I also want to thank Christoffer Ferm for bouncing ideas about network connectivity with me. Fabian Bergstrom, Fredrik Andersson, Gustav Simonsson and Henrik Nordh deserves my gratitude for making the travel to and from Stockholm more interesting. Last I want to thank my parents and my brother for supporting me during the years at the university and during the time it took to write this thesis.

1 Introduction

The development of file sharing protocols have progressed rapidly during the past years as internet have become faster. A couple of examples of protocols is bittorrent[10], direct connect, gnutella[5] and many more. Most of these protocols are working in a peer-to-peer fashion, but they all needs a central server, or "superpeers"[5], that handles routing[5], search indexes[10] etc.

More and more people are using an advanced mobile phone and according to BBC News there is over 5 billion mobile phones worldwide and this number is growing rapidly[4].

This protocol was initially thought to be run inside the embedded systems of a car, to transfer collected data (Milage, fuel consumption etc) to a *gathering station*.

2 Socius

2.1 Introduction

Socius is the name of the application, it is latin and means sharing. The concept behind *Socius* is quite simple. It operates in a delay-tolerant network[6] and is meant to be run on mobile devices that utilizes connectivity through ad-hoc network. Socius is an application that uses a *subscribe/publish* kind of mechanism to deliver data. To publish data, the user needs to give *tags* to the files he/she wants to share. When a file is associated with a tag, the system recognizes it as shared. These tags consists of strings with some descriptive name. Say for example that a user wants to share a photo of a sunset, he or she can then give the picture tags like ”sunset” or ”photo”. Except from the tags, the files are also indexed by a hash. This is to prevent from duplicate data on the same node. To subscribe to files the user needs to insert some *interests* into the system. These interests are later used when requesting files from nodes. When there is interests (> 1) Socius will broadcast a request-message (See 2.4.2) with all the interests. When a remote node receives a request, it looks into the internal database and searches for files that matches the interests provided in the request-message. If one or more matches is found, it sends back a response message containing the hashes of the files matched. All of this is happening without the need for a central server.

This kind of protocol could be very useful in environments where connectivity to the internet is limited, i.e. the subway. Say that a subway traveller, lets call him *Bob*, wants to listen to a specific band or song, but could not do so because of the limited connectivity. If one of the other subways travellers had Socius installed and also had published data with the same tags that *Bob* where interested in. In that case Bob could get these files within some time[1].

2.2 Communication

One of the interesting things with Socius is how it handles files. Both internally, but also how it finds files that is of interest of the user. This is managed by *Interests* and *Tags*. How they work and what they are used for will be described in the following sections.

2.2.1 Publish data with Tags

When a user starts the application for the first time, *Socius* looks in its sharing path which is specified in the configuration file (See 4), and starts indexing the files there. For each file it will create an internal record with meta data information, e.g. *filename*, *file-hash* and *file*

[1] Depends on the size of the file and how good the transmit rate is

path. After the indexing is done, the user can associate a file with one or more *tags*. A tag is a string defining the content of the file. These tags are used as a index and is used in the search process. A file can only be accessed by other users if it have one or more tags.

2.2.2 Data subscription

As mentioned in the introduction, we subscribe to data by adding one or more *interests* to Socius. An *interest* is, just like *tags* a string describing a subject that the user is interested in. Socius will then continuously broadcast *request messages*, containing the interests given, to nodes nearby. To see which interests that is currently active one can give the following command to socius: `taghandler:get_interests().`.

2.2.3 File discovery

The file discovery is passive in the sense that the user does not have to make an active search in order to request files. When there is one or more *interests* defined, a *request message* is broadcasted to the neighboring nodes. This message is broadcasted continuously. When a node receives such a message it extracts the *interests* and compares them to the *tags* contained in the database. If a match is found, a message containing the matching files are constructed and sent to the node where the search message originated from.

2.3 Application structure

The application contsists of four (4) main modules. All of these have a *supervisor***??** module, except for the *communication worker*-module who is in charge for keeping the belonging process alive. A short presentation of each of these main modules follows.

2.3.1 File handler

The *file handler* module handles all contact with the actual data. The current implementation works with the underlaying file system, but it is designed to be easialy exchangeable to other types of storage, e.g SQL-database, key-value store etc. The files contained in the *file handler* is indexed by a *checksum*, a value that is unique for each file. By using *checksums* as indexes, it is very easy to detect duplicated files[2].
A file is represented by four fields; *checksum*, *filename*, *filepath* and *tags* (See 2.2). Since we use the underlaying file system the *file handler* does not need to have the actual content in the database.

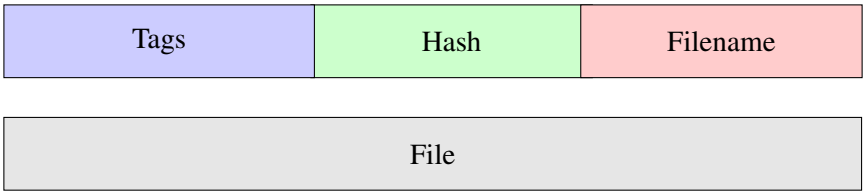

Figure 2.1: Internal representation of a file

Each file can be seen as two parts, one with the information that is contained in the internal database of *file handler* and one containing the actual file.

[2]The *checksum* is based on the content of the file which makes it easy to detect files with the same content

2.3.2 Tag handler

This module handles *tags* and *interests*. It is the *tag handler* that keeps the association database that contains the relations between tags and files. When a file gets associated with a tag, that file is regarded as *shared*.

2.3.3 Communication server

In order to handle incoming request and to broadcast we have the *Communication server*. It listens for request- and response messages and for each new node that sends a message to Socius, the *communication server* spawns a *worker* process.

It does also keep track of how many connections Socius handles right now, and enforces the restrictions given in the configuration file. If a message is received, the server looks if there is any network ports free, and spawns a *communication worker* and assigns it a network port number. When a worker later dies, the *communication server* catches the exit signal and frees the port so it can be used by another worker.

2.3.4 Communication worker

The *communication server* is listening for incoming connection, and when it receives such a *communication worker* is spawned and given the socket which the connection resides in. Each connection is assigned a unique network port number which is later freed when the connection closes. This construct, with one worker process per connection, makes usage of SMP/multicore cpus possible. Since the protocol used is built upon UDP, the worker needs to manually check for correct acks and timeouts. There is options in the configuration files that sets how long a timout is, in milliseconds, and how many timeouts a worker can take before it terminates. As mentioned, the protocol is build upon UDP and we must therefore handle timeouts and ACK manually. This was some problem since the total time spent on waiting for messages was high (See. **??**). A solution to this problem could be to use *sliding window*, meaning that we can have multiple packets in transfer at the time.

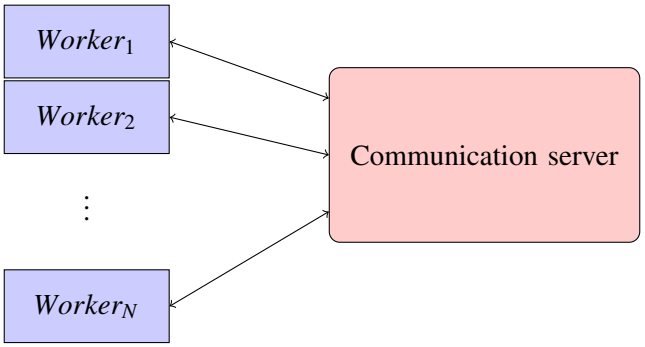

Figure 2.2: The communication server spawns a new worker for each new connection

2.4 The protocol

Since this protocol is meant to be run in a mobile devices in decentralized network. I looked at both *proactive* and *reactive* ad-hoc protocols and decided that *OLSR* was the best fit for

this project. It is a proactive protocol and consumes a little more power than a reactive ditto[7] (Such as *AODV*).

2.4.1 Using Socius in an Ad-hoc network

If we look at an ad-hoc network with several hops between the sender and receiver, one can see that this protocol works quite well. The broadcast is spread over the network and clients can answer the sender to initiate a file transfer. This is not a feature in Socius itself, but in the layers underneath.

2.4.2 Specification

The protocol used in *Socius* is based on seven (7) different types of packets and is build upon UDP.

Broadcast

Identifier	Tags
1	List of strings

Consist of two fields; *Identifier*, Which in this case should be set to 1, and *tags*. Tags is a list of strings. This packet is broadcasted in intervals to nodes nearby.

File response

Identifier	Md5 sums
2	List of md5 sums

This packet should be sent as a response of a *broadcast*-message if the client got files with some of the tags specified in the *broadcast*-message.

Filerequest

Identifier	Md5 sums
3	List of md5 sums

When the client have received a *File response* it looks in the lists of md5 sums and compares it to the local file store, and sorts out the duplicates. If the resulting list is not empty, it is inserted into a *Filerequest*-packet which is then sent to the sender.

Filemetadata

Identifier	Filename	Hash	Tags
4	String	Binary	List of strings

Every file is preceded by a *Filemetadata*-packet. This packet contains information about the file, soon to be transferred. When a user receives this message, it should send back an *Ack* where the *segnum*-field is zero (0).

File transfer

Identifier	File number	Segment number	Content
5	Positive integer	Positive integer	Binary

This packet contains the actual data. It also have fields indicating the *File number* and *Segment number*. Both of these fields are used when the receiver generates the *Ack*-packet.

Ack

Identifier	File number	Segment number
6	Positive integer	Positive integer

When a client receives a *filemetadata-/file transfer* packet an *Ack* packet is generated. If a *filemetadata* packet was received, then the *segment number* should be 0.

Goodbye

Identifier	Status
7	Integer

When all files are sent on the senders side, he sends a *goodbye*-message to indicate that the connection is about to stop. The *status* field is not yet implemented.

2.4.3 Sequence diagram

An example of a connection can be showed by this sequence diagram;

In this diagram we have two nodes, *Client1* and *Client2*. The first client, *Client1*, have some entries in its *interest*-list and because of that does Socius broadcast this list.

Broadcasting interest-list Broadcast message is received by *Client2* who makes a lookup in the tag database to see if there is any matches. If there was a match the hashes of these files is included in a new packet, a *file list* packet, and sends it as a response to the broadcast message.

File list *Client1* checks the hashes provided in the *file list* packet against its file database. For each hash, the client tries to match it against the internal file database. If a match is found, that hash will be removed from the list. When this is done, and the resulting list contains more than 1 element, the client will create a *file request* packet with all the hashes it wants to download.

Filerequest list The *file request* packet contains a list with hashes and when *Client2* receives this he will extract the first hash and get the file which is associated with it. A *meta data* packet is created, which contains information as *filenum* and *seqnum* (See 2.4.2). These fields is used when the other node later *ACK*s them. This packet are then sent to *Client1*. When the remote client receives such a packet, it creates a file and opens a handle to it.

ACK This is used as a reset to received packets. It contains two fields; *File number* and *Segment number*. If the packets which is going to be ACKed is of type *Filemetadata*, the *segment number* field must be zero (0).
File response The actual data is transferred by the *file response* packet. When a client receives this it writes the data to disk and sends an *ACK* to indicate that it received the packet.

Goodbye When all the files are sent, the sender tells the remote node this by sending a *Goodbye* packet. When the remote client receives this packet it response with another *goodbye* packet and terminates the connection. This packet contains one field, *status*, which is intended to indicate if something went wrong. This feature is however not yet implemented.

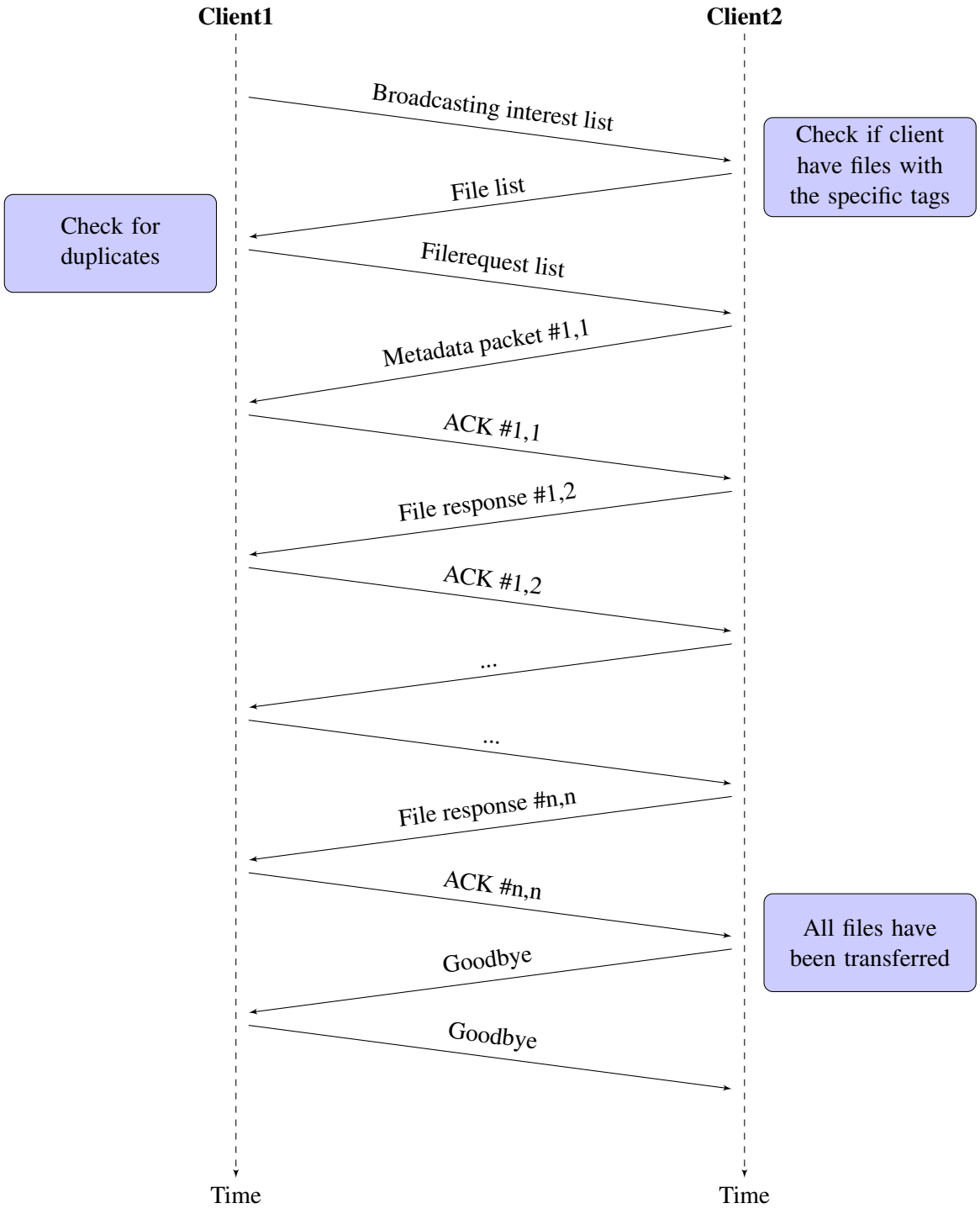

Figure 2.3: Sequence diagram of the Socius protocol

2.5 Tests and analyze

The test was performed using one Apple Macbook Pro 15", Apple Mini, Both running Mac OS X Snow leopard, and one HTC Desire with Android 2.2. All of the devices were connected to a D-Link DIR-655 Wlan router using WiFi (801.11b). The tests was executed about ten times, and then an average value was calculated from the result.

2.5.1 Throughput

The throuput was tested by measure the time it took from that the initial packet arrived (The broadcast-message) and to the file transfer to complete. The file sent was a 6.5Mb data file.

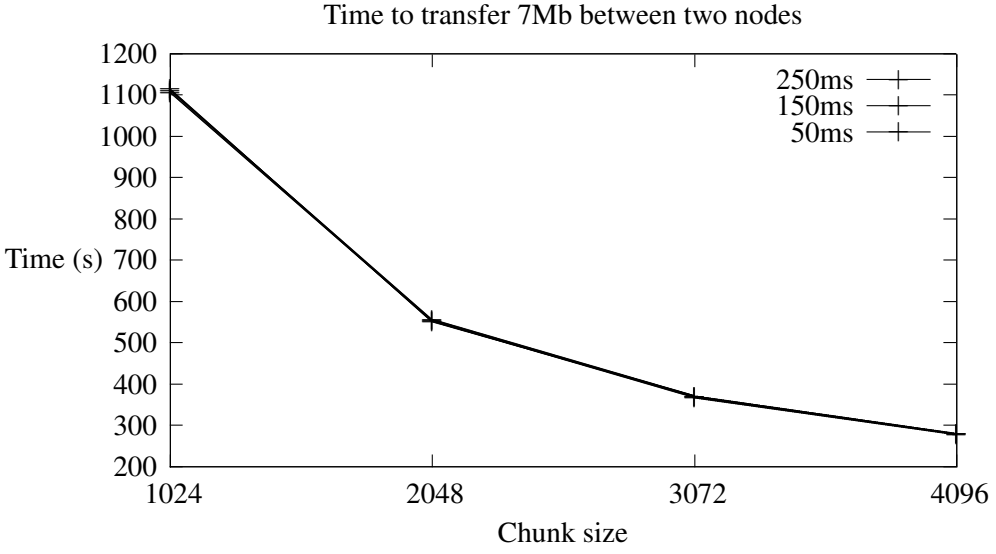

Figure 2.4: Throughput between a mac mini and HTC desire with different timeouts

The graph in figure 2.4 shows the time it took for the file transfer to complete using different sizes of the chunks and a value of 250ms for the timeout[3].
As you can see, the throughput does not change dramatically between different values of timeout.

These graph tells us that the different value of *timeout* does not affect the time of the transfer. The next step would then be to analyze how many packets was resent during the different values of *timeout*. One of the most interesting fact seen in both graphs can be found at the x-axis. When the *chunk size* decreases, the throuput grows. The most reasonable explanation for this is that we have a problem with the way we send packets. Right now we only allow one packet in transfer at the time and this could be one of the reasons why the protocol have a low throughput. If we had used TCP instead the limitation with one packet in transfer would be solved and and features within TCP such as *sliding window* and *fast recovery/retransmit* would most likeley increase the throughput [11].

Another possible reason why we get increased throughput with higher values of *chunk_size* is the latency in the filehandler. In the current implementation, each received data-packet is written to disc directly.

[3] When a packet are sent, the client waits for some time to receive an ACK from the other side, and this time is referred to here as a *timeout*

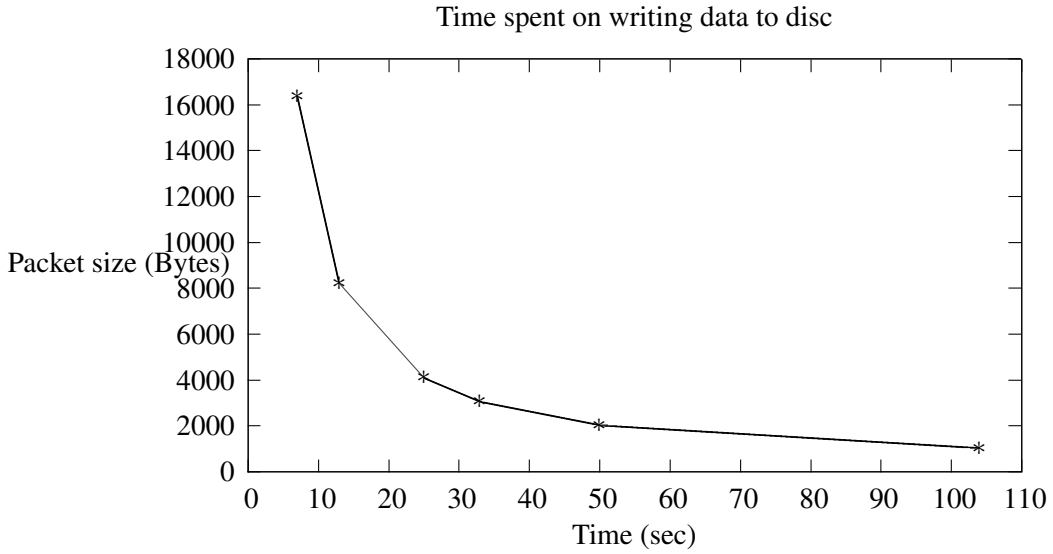

Figure 2.5: Benchmark of the filehandler-module

As you can see in the graph, there is some overhead when writing small packets to disc. One solution would be to implement a local buffer in the filehandler. When the buffer is full it writes it to disc and clean itself up. The benchmark is implemented as a anonymous function:

```
fun(ChunkSize, Packets) ->
  {_, S, _} = erlang:now(),
  [ filehandler:append_file("testfile", <<X:ChunkSize>>)
          || X <- lists:seq(1, Packets)],
  {_, S1, _} = erlang:now(),
  S1-S,
end.
```

2.5.2 Resent ACKs

When a faulty ACK is received it resent. A faulty ACK is when the *segnum* or *filenum* fields contains a value which was not expected. The experiment setup is the same as in the throughput (See. 2.5)

This graph shows a curve that starts quite high, 1231 ACK-packets where resent when chunk size was 1024 bytes, and decreased quite fast when the size increased. We know that the file sent was 6.5Mb, and if we used a chunk size of 1024 the total number of chunks would then be about 6500. According to these numbers, an average of 24.2% of the packets will be resent. Looking at the case where we achieved the best throughput, with a chunk size of 4096bytes, the resent packages is about 17%. This is probably because we increased the chunk size with a multiple of 4 and the total number of packets sent would then decrease with a fourth.

If we instead looks at the following graph in which we set the timeout to 150ms:
If you compare this graph with Fig 2.6 you can see that there is a lot more resent packets here. We decreased the timeout and hence the amount of resent packets increased. If we go back and look at the the result collected at the *Throughput* experiment (See. 2.5) we can see that the timeout does not affect the total throughput. This is quite interesting since we

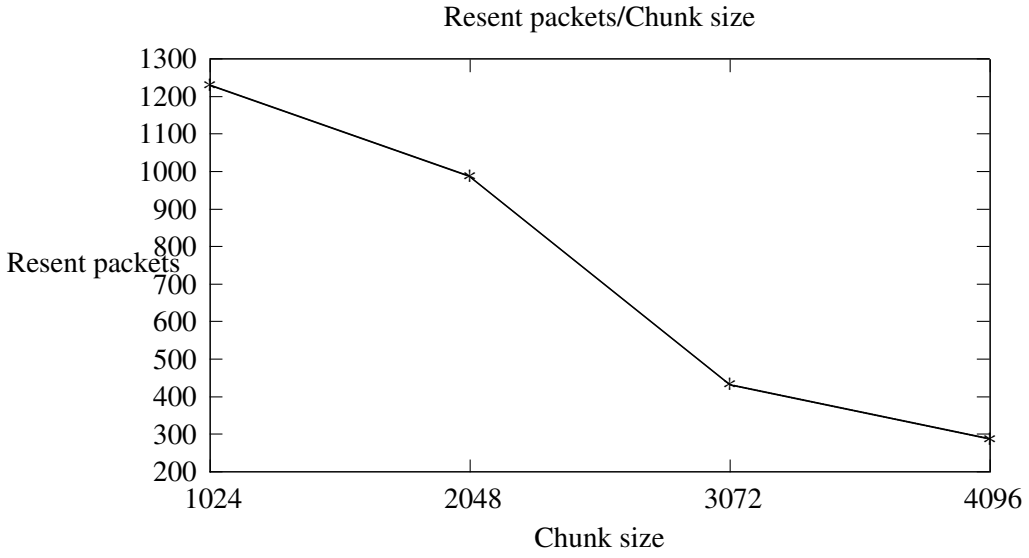

Figure 2.6: Resent packets with 250ms timeout

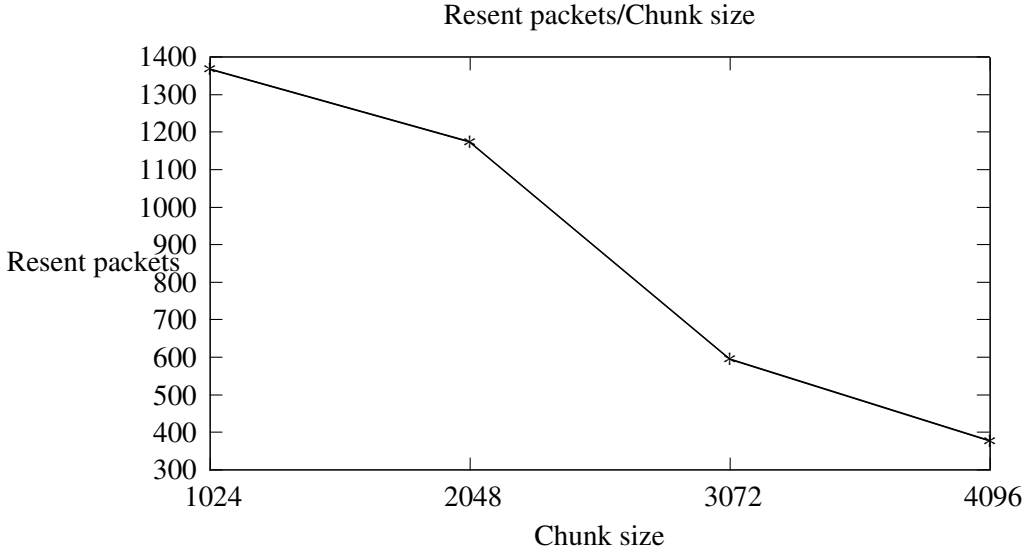

Figure 2.7: Resent packets with 150ms timeout

want to optimize for power consumption and can therefore have a reasonable high value on the timeout, and still get the same throughput as with lower values. This means that we do not have to use the network as much as we do with a low value and can, because of this, decrease the energy consumption usage.

2.6 Conclusion

The logic of *Socius* works, but there is still a lot of work before it can be used outside the academic world. From the beginning it was meant to be run inside of cars to gather and transfer data to special "gathering stations", but after some changes to the protocol we saw that it could be used in a more general situations. In a populated are such as urban transportations and buses an application, such as Socius, could greatly benefit the travelers. It

uses UDP, something that had a quite negative impact on the throughput. During testing (See 2.4.3) we saw different values on the *timeout* did not change the throughput, but rather confirmed that our protocol have reached it's limit when it comes to throuput which at best was around 20kb/sec. One of the problems with the Socius protocol is that it only allow one packet in transfer at the time. If a packet is lost, then the entire queue of packets are delayed until the
Socius consists of four modules and two of these, file- and taghandler, can be extended so the behaviour changes. It is, for example, possible to extend the file handler so it uses a *key-value* store instead of the file store.

2.7 Future Work

Socius is a prototype platform and there is still many things that could be improved.

Limit the numbers of tags in messages

As it is now, the length of the lists, containing tags, do not have any limitation. This means that a user who have a very large number of interest-tags in the database will be broadcasting packets of a large size. One solution to this problem would be to implement some kind of fair scheduler that extracts a fixed number of tags for use in messages. The scheduler would have to be a fair scheduling, so that starvation of tags would be prevented.

Limit the amount of hashes in the file response packet

If a client have a large number of files with the same tag and that tag gets requested, then the *file response* packet will become very large. Since this protocol is designed for mobile devices, with limited connectivity, there is not a very large group that will be able to transfer more than maybe $1-5$ files.

Use TCP instead of UDP

UDP was choosen at first cause it was easy to do prototyping with it. The performance, however, was not so good. This may be a result of that there is only one single packet in the air at the time. Meaning that we just send one packet at the time, and for each data-packet we need to ACK it. The throughput with UDP increases when we are using a larger chunk size of the packets (See figure **??**), something that is related to the problem with just having one packet in transmission at the time. If we would use TCP instead, we would have things like *sliding window* and *fast retransmit/recovery*. These things would most probably increase the throughput.

Prevent from getting the same file from two separate senders

It is possible that a client receives the same file from two separate senders at the same time. This can be prevented if the hash, provided in the *filemetadata* packet, is inserted to the file store on receive.

Save tags on exit

When Socius is halted, all the tags stored in the *tag handler* dissapears. The database used in *tag handler* is memory based, and since we do not save the data manually, it is removed

when the server shuts down. Fixing this should be fairly simple by adding a save-database call in the *tag handlers terminate* function.

Able to resume files

If a file is in transfer between two nodes, and the connection is disrupted, the receiving node will just close the connection and end up with a non-complete file. There is two problems associated with this issue; The first one is that if a non-complete file will occupy the file system until the user manually removes it. The other problem is that if the connection is disrupted, the connection resets. This causes a lot of network- and time overhead in a disruptive network. If a mechanism for resuming unfinished would be presented, both the disk usage and throughput would increase.

2.8 Related work

A very good example on how a content aware system can look like is *Haggle*. It is aimed for mobile devices and a lot of research have been put down on Haggle[9]. I looked on Haggle in the early phase of this project, and was greatly inspired of it. Socius is not meant to be a replacement for Haggle, but rather a light weight version of it. Another paper that inspired me in this project were *Media Sharing based on Colocation Prediction in Urban Transport*[8]. They presents an approach to content sharing over urban transports. Their software tries to determine which source is the best, of the available peers, based on variables such as travel time, connection strength etc.

3 Erlang for the Android platform

3.1 Installing Erlang on Android

To install the finished port to an Android device was a little complicated in the beginning. One had to install a *ftp* server on the device, use a ftp client from a pc and then transfer Erlang to the device. So instead of having to do all this steps I created an installation program. The installation program is meant to be really simple for the user. It consists of a button which, when pressed, downloads a *TGZ*[1]-archive from an external source[2] and extracts it to the data-directory of the installation program, in this case */data/data/org.burbas.erlang/files/*. After all this is done it sets the file permissions on the executable files. Due to some constraints in the file permissions, the installation can not change any other permissions than for itself. That means that no other program can access Erlang and its files (See first section in 3.3).

3.2 Distribution

In the erlang manual one can read "For each host where a distributed Erlang node is running there should also be an EPMD running."**??**. So in order to get the distribution to work the *epmd* daemon must be started. In my first attempt to port Erlang i tried to start *epmd* through erlang by providing the *erl*-script with the *-name*[1] flag. The erlang manual states that "*The Erlang Port Mapper Daemon epmd is automatically started at every host where an Erlang node is started. It is responsible for mapping the symbolic node names to machine addresses. See epmd(1).*"[1]. However, when running on android this caused a crash and a coredump, which was caused when erlang itself tried to start *epmd*.

3.3 Interface between Androids GUI and Erlang

The erlang team provides a interface module for Java, *JInterface*. "*The Jinterface package provides a set of tools for communication with Erlang processes. It can also be used for communication with other Java processes using the same package, as well as C processes using the Erl_Interface library.*" [2]. JInterface requires the erlang port daemon (See 3.1) in order to work. I have created a small Java example that runs on android and is able to communicate with Erlang.

[1] Unix *.TAR* file archive compressed with Gnu Zip (*.GZIP*) compression; the TAR file archives multiple files into one file and the Gzip compression reduces the archive file size.

[2] `http://erlang.burbas.se/arm_erlang_R14B.tgz`

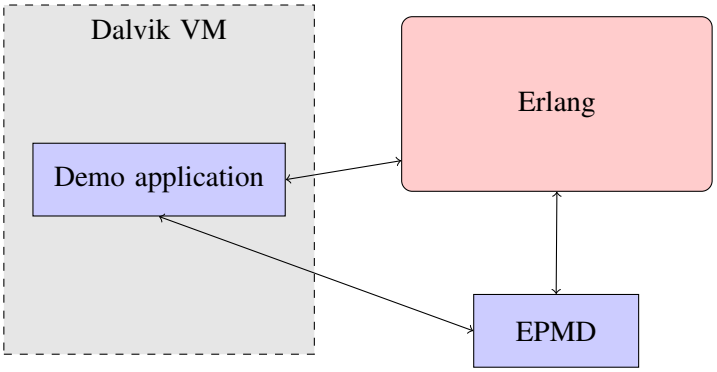

Figure 3.1: JInterface interaction

When the demo application starts it brings up two background processes; one that starts *epmd* and the other one starts the Erlang VM. After this is done, the demo application creates a new OtpNode, provided by JInterface, object by calling

```
OtpNode erlang_node = new OtpNode(name_of_node, magic_cookie);
```

The OtpNode object act as an Erlang node. It is very important that the *magic cookie* is the same on all nodes that will participate. In order to send and receive messages one need to create a *mailbox*. The mailbox acts as an erlang process, and can send and receive packets from other nodes.

3.4 Problems

3.4.1 Need for a ”rooted” device in order to get EShell access

One problem is that in order to get access to *EShell* from a third party terminal editor, the device needs to be *rooted* . This is because Erlang is residing inside the internal memory and the files are belonging to the installation problem. Other users will not be able to read, write or execute the Erlang files due to file permissions. These permissions can not be changed on a *non-rooted* device. So non of the Erlang files can be reached outside the installation program. One solution to this would be to include a terminal editor with the installation program. It is not a good solution, but it is hard to get access without rooting the device.

3.5 Conclusion and related works

Android is an operating system for small devices, especially mobile phones, and its popularity is rapidly growing. Erlang provides distribution and fault-tolerance and can be of great interest for the Android community. Android uses java as their main language, and with JInterface, Erlang and Java can communicate with eachother and perhaps combine their strength. CouchDB[3] is a key-value store with support for syncing. They have been working on an Erlang port and have a prototype of their database running on Android.

[3] `http://www.couchone.com`

3.6 Future work

EPMD

Erlang uses a name server to address all of the nodes in a distributed setup. When a erlang process wants to send a message to another node, it makes a request to the epmd with the name of the node, on the form *Name@Node*, and will get the IP-address and port number in return. When one starts erlang with the flags *-name*, epmd should be started automatically. When doing this on Android, Erlang crashes and gives a coredump. To go around this one can start the epmd daemon manually with the *-d* flag and then execute erlang with the *-name* flag.

Benchmark and optimization

Fredrik Andersson and *Fabian Bergstrom* wrote a master thesis with the subject *Development of an Erlang System Adapted to Embedded Devices*. They decreased erlangs disk footprint by 60% by stripping the beam files[3]. Some of their work can be applied to the Android port.

4 Appendix A: Socius User guide

4.1 Configuration file

The configuration file is located in the *priv* directory of the Socius root. It is named *socius.conf* and contains a number of options. These options are formatted as a erlang list with tuples. Each tuple contains of an identifier and a value.
socius.network.listening_interfaces, String
Here is the network interfaces which will listen for request specified. Multiple interfaces are separated with comma.

socius.network.max_connections, Positive Integer
The max amount of connections we allow. Each active connection occupies a port.

socius.network.packet.size, Positive Integer N where $N = 2^M$
Specifies the size of packets sent. The size must be to the power of 2 (1024, 2048 and so on).

socius.network.broadcast.interval, Positive Integer
How often (In milliseconds) we broadcasts requests.

socius.network.timeout_msec, Positive Integer
When a packet is sent, Socius waits for a response for some time. If no response have been received during the specified time, the packet will be resent. This option specifies how long this time is (In milliseconds).

socius.network.max_timeouts, Integer
If a connection is very lossy Socius can drop the connection if a specified number of timeouts have occured. If this value is -1 the number of timeouts is ignored and no connection is dropped due to it.

socius.path.sharing, String
This contains a path to where all files that are shared is located. Downloaded files will also be saved here.

4.2 Getting started

To start Socius, just enter its *ebin* directory and start erlang. When the Erlang VM is started you can run `application:start(socius)`. This operation should return ”ok” if the application was started correctly. If not you might have an error in the configuration file. Now when the application is up and running, you can see which files that currently is occupying the system. To do so you can use the *filehandler:get_file_list()*-function.

```
2> filehandler:get_file_list().
[{file_information,<<82,137,223,115,125,245,115,38,252,
                     221,34,89,122,251,31,172>>,
                   "testing.txt",
                   "/Users/sample/socius/sharing/testing.txt",
                   undefined}]
```

This function returns a list of tuples. Each of these tuples corresponds to a file and contains five fields. The second field is the most important here.

4.2.1 Sharing

The definition of a file is that it should have, at least, one file associated to it. So to share a file the user only needs to associate a tag to it.

4.2.2 Subscribe for files

To subscribe for data the user only needs to add an entry in the interest-database. This is done by calling the `taghandler:add_interest(Interest)`-function. The interest can be of any erlang term. To see which interests that is in the database just run `taghandler:get_interests()`. To remove an interest use `taghandler:remove_interest(Interest)`.

Bibliography

[1] Ericsson AB. Erlang - distribution protocol (manual). Available at *http://www.erlang.org/doc/apps/erts/erl_dist_protocol.html*.

[2] Ericsson AB. Erlang - the jinterface package (manual). Available at *http://www.erlang.org/doc/apps/jinterface/jinterface_users_guide.html*.

[3] Fredrik Andersson and Fabian Bergstrom. Development of an erlang system adopted to embedded devices. Master's thesis, Uppsala University, 2011.

[4] BBC. Over 5 billion mobile phone connections worldwide, 2010. Available at *http://www.bbc.co.uk/news/10569081*.

[5] Clip2. The annotated gnutella protocol specification v0.4. Available at *http://rfc-gnutella.sourceforge.net/developer/stable/index.html*.

[6] Kevin Fall. A delay-tolerant network architecture for challenged. *ACM SIGCOMM*, 2003.

[7] Kevin Fall. A delay-tolerant network architecture for challenged internets. SIGCOMM 03, 2003.

[8] Aleksandr Huhtonen. Comparing aodv and olsr routing protocols. *HUT T-110.551 Seminar on Internet working*, 2004.

[9] Liam McNamara, Cecilia Mascolo, and Licia Capra. Media sharing based on colocation prediction in urban transport. MobiCom 08, 2008.

[10] James Scott, Pan Hui, Jon Crowcroft, and Christophe Diot. Haggle: a networking architecture designed around mobile users, 2006.

[11] TheoryOrg. Bittorrent protocol specification v1.0. Available at *http://wiki.theory.org/BitTorrentSpecification*.

[12] George Xylomenos and George C. Polyzos. Tcp and udp performance over a wireless lan. In *TCP and UDP Performance over a Wireless LAN*, pages 439–446, 1999.