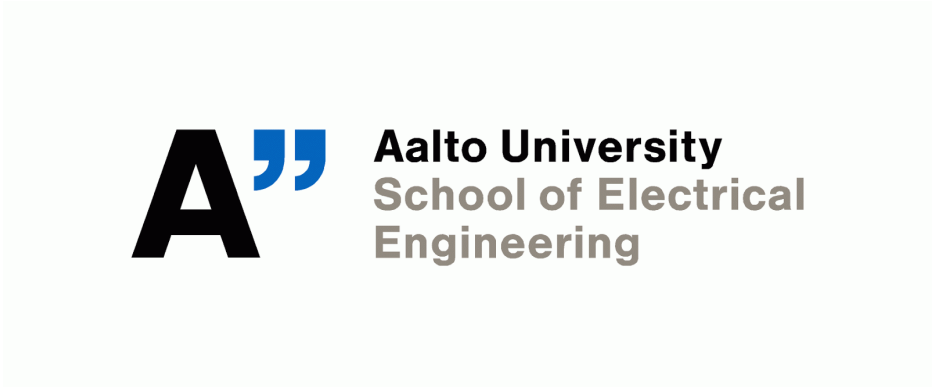



# S-38.3610 Network Programming

# Spring 2015

# Final Report

10.04.2015

Group 3: FINDME

Raghavendra M S, raghavendra.mudugoduseetarama@aalto.fi
Anteneh Adem, anteneh.adem@aalto.fi
Raul Morquecho, raul.morquecho.martinez@aalto.fi
Jaume Benseny, jaume.benseny@aalto.fi

**Table of Content**

# 1. Overview and overall architecture

## Overview

FindMe is **location resolution application for mobile devices** that keeps updated information about the position of all moving FindMe clients as well as complementary information about them. Any device can query information about another device by providing requested device ID. At every communication, FindMe clients share their actual location and therefore FindMe servers have updated information about clients position. FindMe architecture and protocol are designed to be simple and light in order to be run in low-power devices thus addressing requirements from Internet of Things.

## Architecture

FindMe can be divided into three separate applications:

1) **Client application:**

Client construct and send queries based on the available information it has from the other users. It sends the query to the server and waits for a reply. Once the reply is received, the client displays the information to the user.

2) **Server application:**

Server is responsible for three main functionalities. First, it resolves received queries from clients about other client´s position or information. Second, it forwards update messages from clients to the Master Server. And third it updates their local information about clients according to updates shared by the Master Server. The server is divided broadly into three modules.

a. Module 1. Data processing module.
b. Module 2. UDP module.

3) **Master server application:**

Master server application is responsible for relaying update messages between servers so that servers across the network could synchronize their database to keep consistent data. It is also responsible for providing server id for new servers.

To avoid delay and enhance performance, the Master Server only forwards client's updates (forwarded by servers) to all servers except the one that sends the update and it doesn't analyse update data. As possible additional feature it could have maintained information consistency, by having its own database, of client updates by checking time stamps and deleting conflicting messages. Another desired functionality that could not be implemented was to support new servers that appear to the network with outdated client information so that they could ask for the whole database information.

### Advantages of the Master Server architecture

The Master server acts like a hub. It concentrates all client's petitions towards one point providing the following advantages:

- Avoidance of mesh networking there by decreasing configuration at the servers and enabling servers to join the network dynamically.
- Good balance between distributed information system and centralized information system

In case update message analysis could have been implemented, Master server could have act as switch instead of a hub (time was not enough). It could have potentially benefited the system by:

- Centralizing update decision-making
- Simplified information coherence across servers

### Disadvantages of Master Server architecture

However, due to this architecture the whole network operation relays on the presence and operativeness of the Master Server. Taking as example the DNS service, this problems could be solved by the introduction of the following features:

- Server coronation
- Multiple Master servers towards federated infrastructure

The server coronation occurs when Master server has not responded to server's requests for some time. At this point, servers broadcast to all known servers their unique ID. The new Master Server will the one with the lowest server ID. Unfortunately this functionality was not developed.

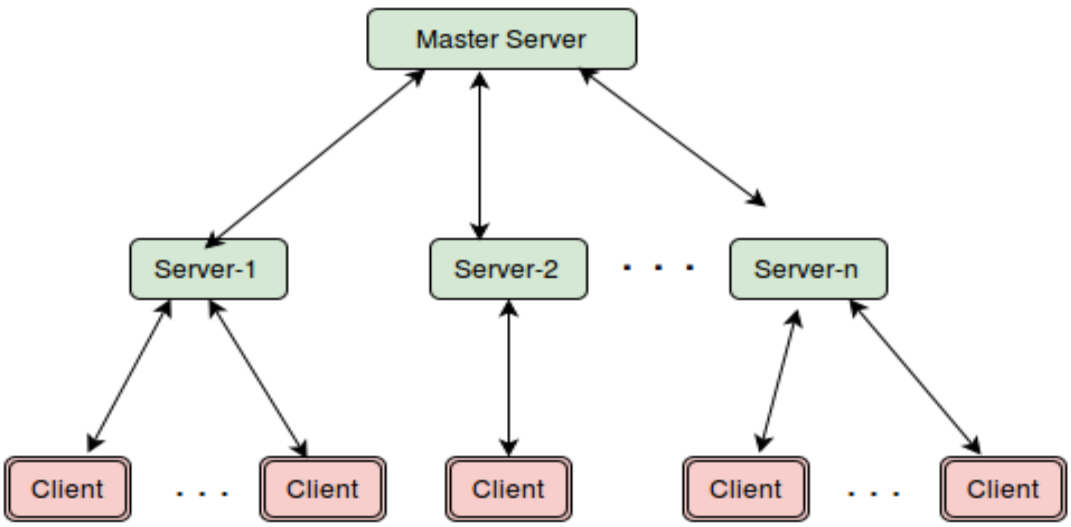



Fig 1. Overall architecture

# 2. Requirements description

**NOTE**

Main implemented functions, intended operation platform and programming language are described in Chapter 5 Implementation.

## General requirements

Major requirements to be implemented as part of FindMe:

- To handle name resolution queries from clients
- To handle information update messages from clients
- Data synchronization between the servers through Master server
- Uniquely identify clients

## Detailed requirements

The following tables describe in detail the specific requirements of each of the applications as well as their respective modules.

| **Client requirements** |
| --- |
| Features |
| ● Query servers about other clients information (location)<br>● Receives replays from the server and prints it to the screen<br>● Send the location information of the user to the server each time the user opens the application for sending a query and at the same time that the user requests a name resolution.<br>● This module is also responsible for interacting with the user; a dialog through a terminal will enable the user to send a query in order to retrieve the desired information.<br>● NOT IMPLEMENTED - Ask about client location within a time frame. |

**Server requirements**

- The servers store about clients:
  - Name
  - Email Address
  - GPS Location
  - Postal Address
- Servers synchronize information to keep updated data about clients
- Uniquely identifying the clients
- Handle resolution queries from clients
- NOT IMPLEMENTED - They must be resilient to one server failure

**Module 1. Data processing module**

Features

- Generate client IDs
- Receiving queries from clients and encode response
- Receiving messages from server and change particular entry
- Tell maser server its up and asks for a server id if the server is a new server
- Whenever a server starts it should contact its relay server so that it could get database updates
- If the server is a new server it gets a server id when it contacts its relay server
- send synch message to server containing the server's id, if the server does not have an id send one byte set to zeros.
- receive ack from relay server
- prepare a socket for sending and receiving database updates to the relay server

**Module 2. UDP Module**

Features

- Manages network configuration
- Opens UDP socket and listens for petitions from clients and servers
- Fills predefined structure with received petitions
- Creates message queue to communicate with module 1
- Monitors status of message queue and UDP sockets
- NOT IMPLEMENTED - Replay to clients/servers about position correctly updated

| **Intermodule communication** |
| --- |
| ● Message queue is created to manage petitions from Module 2 and Module 1.<br>● Message queue enables FIFO, prioritization of messages as well as multi-thread compatibility in case of future requirements. |

| **Master server requirements** |
| --- |
| Features |
| ● General objective: Synchronizing database between servers<br>● Receives database updates from other servers<br>● send database updates to all other servers except the one that send the database update<br>● Does not have its own database to store client information, it only relays updates between servers.<br>● Assigns unique server id for new servers<br>● It could only serve a configurable number of servers, above this number it rejects JOIN requests by sending JOIN$00000 message. Software limited maximum number of servers is 100 ipv4 servers and 100 ipv6 servers.<br>● It support both IPv4 and IPv6 messages and it runs on UNIX OS. |

# 3. Instructions

**NOTE:**

Code can be found at:
https://github.com/antenehd/FINDME

### How to build the Client

Once you have downloaded the source code file from the github, open a terminal, go to the directory where the source code is stored, and run the following command:

user@machine:path$ gcc -std=c99 -Wall -pedantic -g -O0 client_v_x_x.c -o output_name

Where the compilation input starts after the $ sign, <client_v_x_x.c> is the correct name of the source code file and <output_name> is the desired name you want to give to your program.

Note: For this compilation to work, the build must be done in a Linux OS with GCC installed.

### How to build the Server

Inside FINDME/server folder execute command

make clean; make

### How to use the Client

The client application is pretty simple to use, the only thing the user needs to do is running the application on the terminal, for that:

1) Open a terminal
2) Go to the folder where you have compiled and stored your program
3) Run the program with the following command:

   ./output_name

   Where <output_name> is the name you have previously assigned to your program during compilation.

4) Follow the instructions on the screen

### How to use the Server

Make the relevant changes inside Findme.conf.

Note: The SERVER_ID inside Findme.conf should be 00000 initially.

Execute by typing the command

./Nameserver

### How to build the Master Server

To build: where '>' is the command line prompt after navigating in to ‘masterserver’ directory

>make

### How to use the Master Server

First configure "servid.conf" and "address.conf" files.
File "serv.conf" should be configured only once when the master server starts for the first time after that it should not be changed since it contains the maximum server id given by this master server. At first it should be configure with the first server the master server will assign.After this the program will increment this value and save the incremented value whenever it provide new server id.To configure the first server id, save the following in "serv.conf" file:

servid=<firstservid>

where firstservid is replaced by the real first server id and it should always be 5 characters long,if first server id is not 5 character long preceed it with zeros to make it 5 characters long.
File "address.conf" contains the local ip address and port number this master server listen for incoming messages. if this is not configured this values should be give in terminal as argument when the program is started.To configure the address value save the following in "address.conf" file:
port=<portnumber> ipv4=<ipv4address> ipv6=<ipv6address> maxsrvs=<maxnumberofservs>

where <portnumber>, <ipv4address>, <ipv6address> and <maxnumberofservs> is replaced with the real value.

**To run this program:**

>./mserver

check if the server has started properly by listening on the addresses and the port given:

>sudo netstate -ulp

**To clean:**

>make clean

# 4. Communication protocol

FINDME communication is implemented using UDP protocol because its nature is connectionless and asynchronous.

The implemented communication units are:

- **Client to server communication**
  - NEW client. When a client is run for the first time, it will send a message with 0 as ID
    - 0$QUERY\n
  - QUERY message. When the client requests a query, it will send the key work QUERY on the field following the ID and then the type of query (NAME or EMAIL) followed by the word the user provides to resolve, after all that, the current location of the client requesting the query is specified

        - 01$QUERY$NAME$Name_to_resolve$LOCATION$Rue de la baguette 23, Paris 43029, France$LOCATION$48d 51m 12s North, 2d 20m 55s East$TIMESTAMP$1
    - UPDATE message. When a client sends an update, the string sent looks very similar to the QUERY update, the difference is the key work in the second position, which specifies that it's an update
        - 01$UPDATE$NAME$Raul$EMAIL$raul@aalto.fi$ADDRESS$Pyramid street 77, Cairo 92039, Egypt$LOCATION$30d 3m 1s North, 31d 14m 1s East$TIMESTAMP$1

- **Server to server communication**
    - At start new server to master server, JOIN request:
        - JOIN$00000
    - At start old server(server with server id=SRVID) to master server, JOIN request:
        - JOIN$SRVID
    - Master server to new server, JOIN response:
        - If JOIN request is accepted, master assigns server id=SRVID for new server:
            - JOIN$SRVID
        - If JOIN request is declined:
            - JOIN$00000
    - Master server to old server with server id SRVID, JOIN response:
        - If JOIN request is accepted:
            - JOIN$SRVID
        - if JOIN request is declined:
            - JOIN$00000

    - Server UPDATE message to other servers(first it is sent to master server and from there it will be relayed to other servers). SRVID is the the server id of the originating server. MSG is whatever message it want to send to all other servers.
        - UPDATE$SRVID$MSG
    - When server leaves network, server to master server, DISJOIN request:
        - DISJOIN$SRVID
    - Master server to server, DISJOIN response:
        - DISJOIN$SRVID

Each of the following tables summarizes the exchange of messages required to achieve high level functionalities.

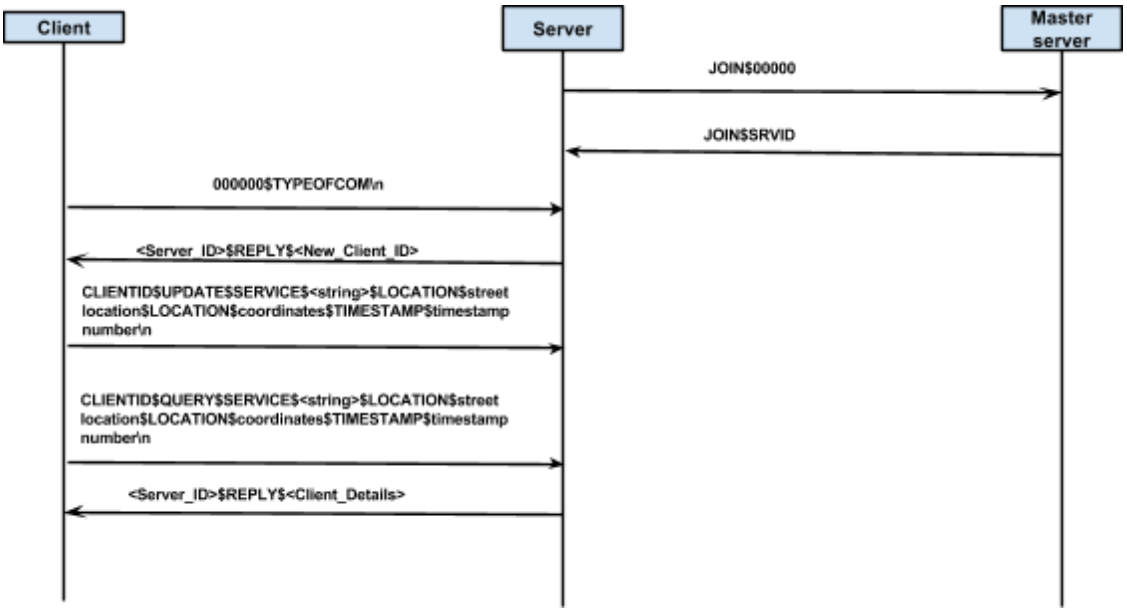
### Client to server communication

| NEW CLIENT |
| --- |
| ● When connecting for the first time, it sends a zero as an ID to indicate that is a new client, once it is assigned an ID by the server; it uses it to identify itself on future queries.<br>● CLIENTID$TYPEOFCOM\n<br>● example: 000000000$QUERY\n |

(tables continue in the following page for the sake of readability)

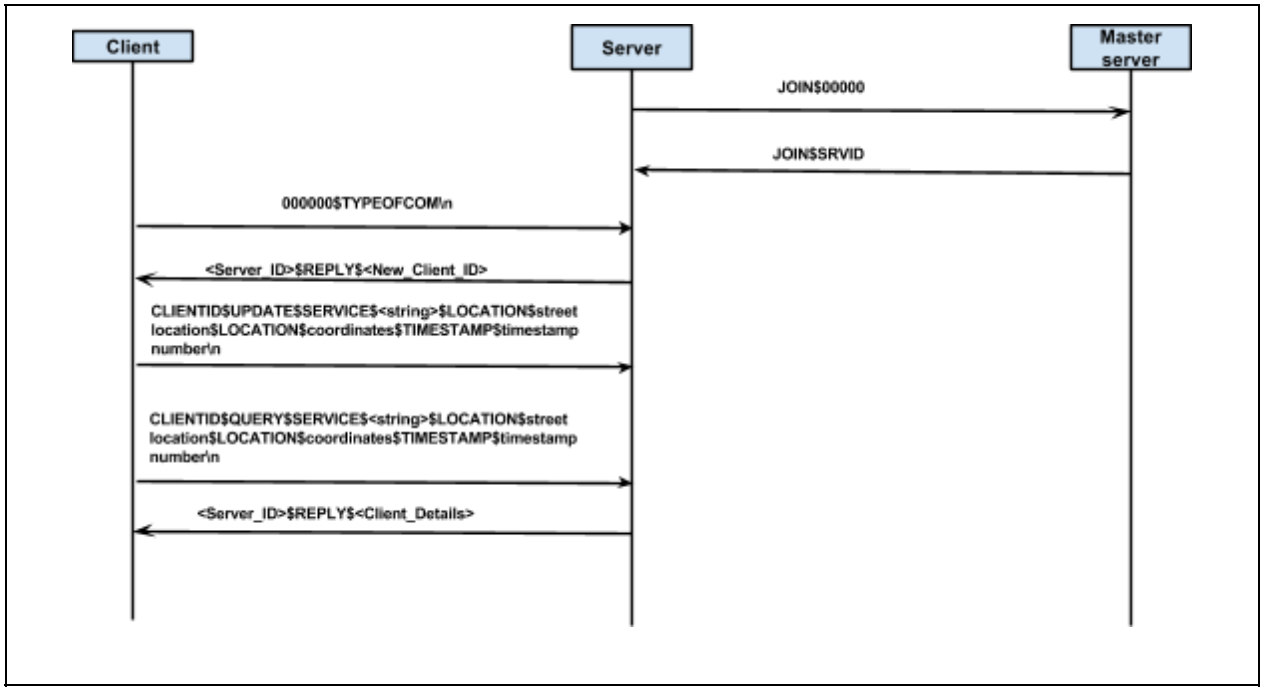
| CLIENT QUERY |
| --- |
| ● The query message sent by the client is composed by 10 fields plus a terminating end-of-line to recognize when the string is terminated<br>● The first field is the client ID assigned previously by the server<br>● The second field is the type of request, it could be QUERY or UPDATE<br>● The third field indicates what information the string contains<br>● The fourth field is the information to be resolved itself<br>● The fifth field is always the key word LOCATION, indicating what comes next<br>● The sixth field is the current location of the client requesting the query |

- The seventh field is always the key word LOCATION, indicating what comes next
- The eight field is the location in coordinates
- The ninth field is the key word TIMESTAMP indicating what the next field is
- The tenth location is the timestamp itself
- CLIENTID$TYPEOFCOM$SERVICE$<string>$LOCATION$street location$LOCATION$coordinates$TIMESTAMP$timestamp number\n
- example: 01$QUERY$NAME$Name_to_resolve$LOCATION$Rue de la baguette 23, Paris 43029, France$LOCATION$48d 51m 12s North, 2d 20m 55s East$TIMESTAMP$1




## UPDATE PETITION

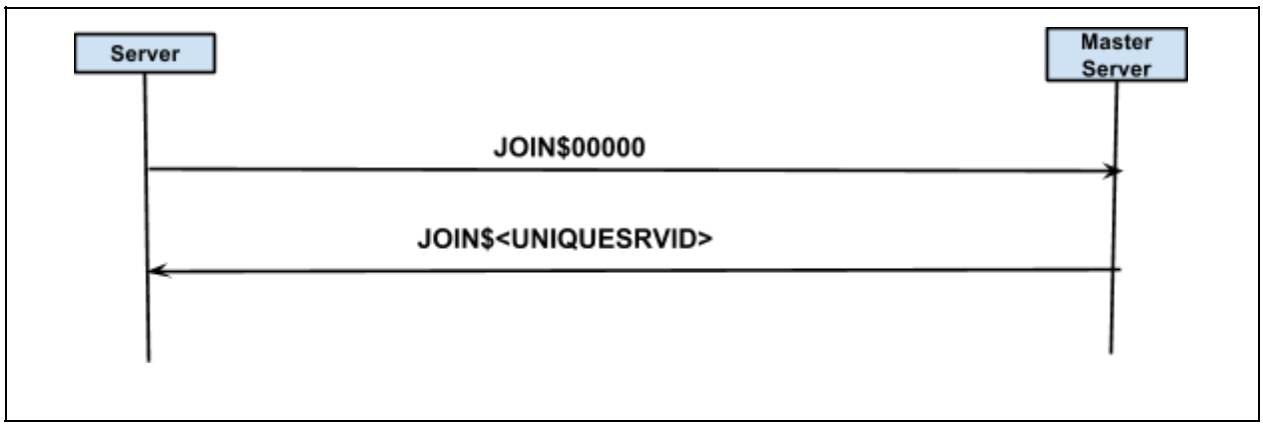
- The update message is used when a client wants to update its own information on the system, for example, if its email address has changed.
    - CLIENTID$UPDATE$SERVICE$<string>$LOCATION$street location$LOCATION$coordinates$TIMESTAMP$timestamp number\n
    - example: 01$UPDATE$NAME$Raul$EMAIL$raul@aalto.fi$ADDRESS$Pyramid street 77, Cairo 92039, Egypt$LOCATION$30d 3m 1s North, 31d 14m 1s East$TIMESTAMP$1



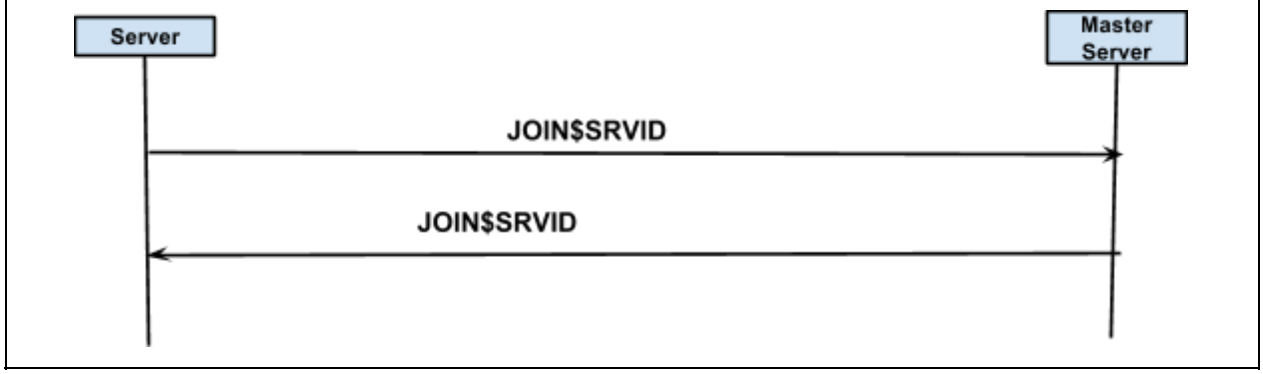

### Server to server communication

Communication among servers is implemented by UDP protocol. Constant notification of updates is shared through the network using the relay server.

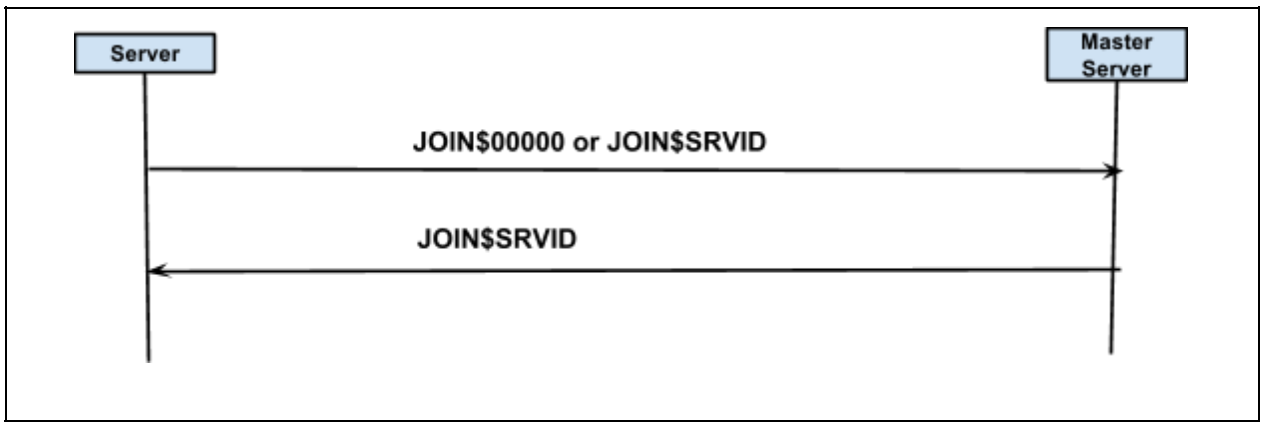
| NEW SERVER JOIN TO MASTER SERVER |
| --- |
| ● JOIN$00000 |




| OLD SERVER JOIN TO MASTER SERVER |
| --- |
| • JOIN$SRVID |




| MASTER SERVER TO SERVER (response to JOIN) |
| --- |
| • JOIN$SRVID , SRVID is assigned by master server if the JOIN request was from new server. |


Server
Master Server
JOIN$00000 or JOIN$SRVID
JOIN$SRVID

# 5. Implementation description

The FindMe applications and libraries have been fully developed by the FindMe members during the Network programming course only in C language. Applications comply to the UNIX standard and have been tested in the nwprog* hosts.

The following block diagram depicts the main modules and functions enabling the function of the different applications.

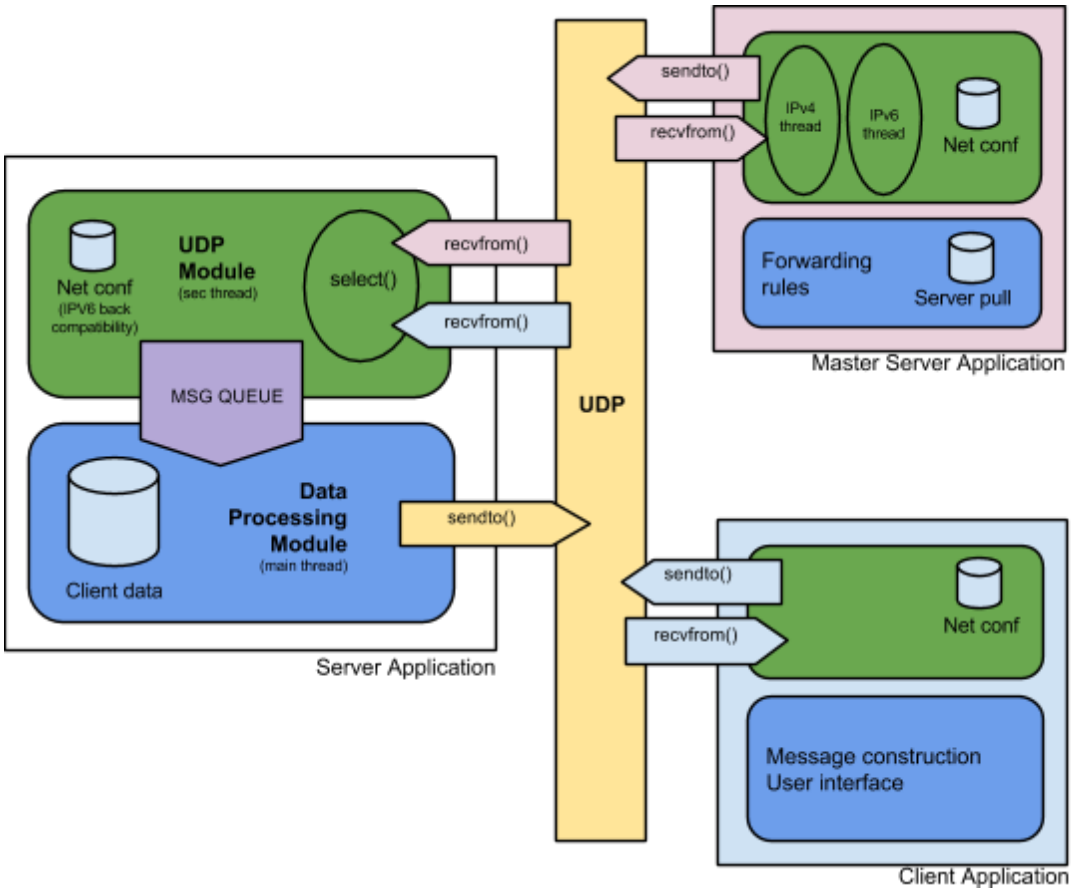



Fig 2. FINDMe Block Diagram

The following tables describe the particular details that are more relevant to the FindMe application implementation:

| **Client implementation** |
| --- |
| General description |
| ● Due to the fact that the client does not need to be connected to multiple parties, it does not make use of multi-threads/processes. It is implemented in a single threaded/processes program, in which it runs linearly, and iterate with the help of a loop in order to take different requests from the user.<br>● The client is stand alone, it does not need to share code, memory or libraries with any other modules, due to the fact that the client runs in a separate computer, the only contact between the client and the rest of the implementation on this project is through the network sockets |
| Main functions |
| *int main();*<br><br>This is the main function, and most of the logic happens inside of this function. First, it creates an array of structures that hold the locations used by the user, it then picks a random location to use as current location, then it initialized the structure with the IP and port to send messages to. After that it checks if the file with ID exists in the current folder, if not, it sends the initial message to the server asking for an ID, once it has an ID it creates a file and saves the ID in that file, next it asks the user the name and email if it's the first time it uses this program, and it store that information into files. Once that is done, it builds a string containing the current location to send to the server as initial update. When all this initial protocol is done, it then offers to the user a few options, the user choses the more convenient and continues to input what the program asks from him. These options are inside of a loop, for this reason, the program keeps asking the user to input the next task to perform until the user choses to exit the program.<br><br>*int create_list(struct info *list, int elements, char *nombre, char *correo);*<br><br>This function is meant to create an array of structures that hold information about different locations. Unfortunately there was not enough time to integrate the program to a GPS or another device capable of sending real time locations, so the alternative was just to create a |

bunch of random locations and have them in an array for the program to chose one of them randomly when sending an update to the server.

*int build_location(char *locstr, struct info extract, int num_cl, int indic, int option, char *query, char *query2);*

This function is meant to create the strings that the client will be sending to the server. The arguments of the function take different parameters that depend of what kind of string will be build (request ID, Query, Update), and then, according to the arguments provided, the function choses internally what string needs to build and return it by modifying the “locstr” parameter.

*int separate_function(char *to_sepa, char **save, int how_many);*

This function is used when the client receives an answer to a query, with this function, the string received from a server can be broken down into separate fields so then it can be printed to the screen for the user to read the requested information.

**Server implementation**

- The server’s main thread initializes signal management, log file, reads configurations files and creates hashtable to store client’s information.
- It creates a second thread that initializes all network configuration, sockets, message queue and select.

**Module 1. Data processing module**

- Data information is stored in memory when servers are up
- Communication with other modules is carried out with message queues
- In order to identify a client uniquely across all the servers, the server will generate an identifier for the client. Every server has a range of numbers and server can assign a number from that range to a client. This helps to keep the identifier unique across the servers. Once a client gets assigned by a number the same number will be used every time the client tries to contact the server.

| Main functions |
| --- |
| **1) void HandleServerReceivedMsg(stRcvdMsg * , int8*,uint64);**<br>This function will handle all the message sent from the peer servers.<br>**2) void HandleClientReceivedMsg(stRcvdMsg * , int8*,uint64);**<br>This function will handle all the message sent from the clients.<br>**3) uint32 FillRecord(stRecord * pstActRec,int8 * pi8MsgPtr**);<br>Function to change the contents inside the record. |

| **Module 2. UDP Module (and Message queue)** |
| --- |
| ● 2 sockets are created one for receiving datagrams from the master server and a second one to get messages for the clients. Management of the sockets is performed by select().<br>● IPV6 sockaddr structure is created with IP and port introduced by the user from terminal. In case user introduces IPv4 address mask is applied to allow compatibility.<br>● IPv6 datagram socket is binded and placed within a infinite loop listening for datagrams. Number of bytes to be read is controlled by MSG_MAX_LEN which is shared as global to the whole program.<br>● Because we work with threads and they share memory space, we exchange addresses through the message queue. This way we improve performance of the system avoiding to copy a variable structure for each message.<br>● Message queue is created by using the UNIX standard. Limit of messages in the message queue is controlled by constant MAX_MSG = 100 and size of messages in the message queue is defined by constant mqstat.mq_msgsize. Moreover, it's created with MQ_MODE in a particular way where Module 2 can only write and Module 1 can only read.<br>● Errors regarding message queue are controlled and errno values are checked for:<br>○ ERNO: EINTR: MQ was interrupted by a signal<br>○ ERNO: EMSGSIZE: MSG length is longer than MQ_MAX_LENGH.<br>● Messages within the queue have equal priority = 1.<br>● API defined by the UNIX standard is used to put and get messages from the message queue. |
| Main functions |
| **1) void * ProcessThreadStart();** |

A thread start function and also responsible to receiving message from both client and receiver and putting it into message queue.

**2) void RequestServID();**

Function to get initial serverId from the master server.

## Master server implementation

- The master server runs as daemon
- UDP is used for communication with other servers.
- Two UDP sockets are created as global variables and one is bind to IPv4 and the other one to IPv6 address but with same port number.
- Master server is limited to serving configurable maximum number of servers and it can not be more than 100 (software bounded).
- It gets the ip address and port number to listen on and maximum number of servers from terminal. If it is not available from terminal it checks for it in configuration file.
- If there is error in parameters given in terminal the master server will not start
- If no parameters are given in terminal, it looks in to the configuration file and if there is error the master server will not start.
- For the master server to start it needs port number, IPv4 or/and IPv6 address, maximum number of servers and starting server id.
- Starting server id is the first server id the master server assigns to a new server. It should be configured only once and after that it will be incremented and saved every time the master server assigns a server id for a new server.
- To enhance performance of the master server, Ipv4 and Ipv6 messages are handled using separate threads.
- If the master server accepts JOIN request from new servers(server with no server id) it assigns a unique address and saves the address and server id of the new server in different global static arrays. A total of four such arrays, two of them store addresses and server ids for ipv4 servers and the other two for ipv6 servers. The master server uses the addresses stored in the address storing array to forward UPDATE messages and it uses the server id stored in server-id storing array to search for a specific server's array index in address storing array so as not to send back UPDATE messages or to delete it. It is the same for old servers except that it will not assign server id. If the request to JOIN is declined the address will not be saved.

- When the master server receives DISJOIN reques It parses the server id from the DISJOIN request and looks into the static array that saves server ids for this server id, if it is found it deletes it from the array that saves server ids and the array that saves server addresses and sends DISJOIN response. If it is not found it simply sends DISJOIN response.

- When the master server receives UPDATE message it parses the server id from the message and use this server id to exclude the sender of the UPDATE when forwarding the UPDATE message to all the servers that successfully JOIN this master server.

- When the main thread receives SIGPIPE or SIGTERM or SIGHUP signal it sends SIGUSR1 signal to the threads that have already started. When Ipv4 and ipv6 handling threads receive SIGUSR1 signal they will exit. Also when ipv4 or ipv6 message handling thread receive SIGPIPE it sends SIGUSR1 signal to itself and to the other thread(does include the main thread). The main thread always exists if there is no thread it is waiting for to finish.

Main functions

main(): the main function reads the configuration files, setup sockets for ipv6 and ipv4 communication and creates two threads. One thread for receiving and processing ipv4 messages and the other thread for receiving and processing ipv6 messages.

ipv4Msgs(): Receive ipv4 messages hands over the messages to handleMsg(,,,,) function.

ipv6Msgs(): Receive ipv6 messages hands over the messages to handleMsg(,,,,) function.

handleMsg(,,,,): Process messages and sends back proper response.

# 6. Quality assurance

Testing has been performed to each of the modules and applications during its development as described in the following tables. Moreover, testing code can be found in each of the folders containing their source starting with the name of the application and the functionality tested as test_server_udp.c for code that tested udp features of the server.

| **Client test scenarios** |
| --- |
| ● ***Test Case 1***<br><br>*Client runs for the first time*<br><br>Step 1) Run the client in a terminal with the command "./program", where <program> is the name of the program you assigned in the compilation.<br><br>Step 2) During the first time the client runs, after the client has send a message to the server with zero as ID, it will receive an ID to be assigned to that client, once this has happened, the client will ask the user to input the name for this client, write the name and press enter.<br><br>Step 3) After the name has been assigned, the client will ask the user to input the email associated to this client, write the email you want to assigned and press enter.<br><br>Step 4) Now the client has all data assigned and saved into the hard disk, now the menu is displayed to the client, select option 4 to exit the program<br><br>● **Test Case 2**<br><br>*Client runs for the second or any subsequent time*<br><br>Step 1) *Run the client in a terminal with the command "./program", where <program> is the name of the program you assigned in the compilation.*<br><br>*Step 2) Client checks whether it has assigned an ID, since this is not the first time it runs, then it retrieves the ID, name and email from hard disk and send the first mandatory update to the server, chose number 4 to exit the program.* |

- **Test case 3**

*Client requests to resolve a name*

Step 1) *Run the client in a terminal with the command "./program", where <program> is the name of the program you assigned in the compilation.*

*Step 2) After all the initial procedure proper to a client which is running for a second or subsequent time, the user menu will show up, select number 1 to provide the name you want to resolve and press enter.*

Step 3) The program will prompt you to enter the name you want to resolve, write it and press enter.

Step 4) Now the request has been sent to the server, the client shows the sentence "The information you requested is:" and in the following line the information received from the server.

Step 5) The user menu is shown again, chose number 4 to exit the program.

- **Test case 4**

*Client queries the server in order to resolve an email.*

Step 1) Run the client in a terminal with the command "./program", where <program> is the name of the program you assigned in the compilation.

Step 2) After all the initial procedure proper to a client which is running for a second or subsequent time, the user menu will show up, select the number 2 to get the information of a user by providing its email.

Step 3) The program will prompt the user to provide the email he wants to resolves, write the user of your concern and press enter.

Step 4) The program will now display the sentence "The information you requested is:" followed by the data received from the server. After that the user menu will be displayed again, select option 4 to exit the program.

- **Test Case 5**

*Client requests to update its own information*

Step 1) *Run the client in a terminal with the command "./program", where <program> is the name of the program you assigned in the compilation.*

Step 2) After all the initial procedure proper to a client which is running for a second or subsequent time, the user menu will show up, select option 3 to make an update of your own information.

Step 3) The program will ask the user to input the name and email to be updated in the format <name:email>, write it and press enter.

Step 4) The update is sent to the server and the data is updated.

Step 5) The user menu is shown again, select option 4 to exit the program.

| **Server test scenarios** |
|---|
| **Module 1. Data processing module** |
| 1. New client requesting an ID.<br>2. client sending a query for a not existing record.<br>3. Request a new ID, Update the content and query for same record.<br>4. Handle update record message from server.<br>5. Handling SIGINT<br>6. New record to a from a server.<br>7. New record message for an existing record from a server.<br>8. erroneous protocol messages.<br>9. Memory leak test for all the above mentioned tests with **Valgrind tool.** |
| **Module 2. UDP Module** |
| 1. IPV4 datagrams are correctly received<br>2. IPV6 datagrams are correctly received |

3. Datagrams with length longer than MAX_MSG_LEN are sent but discarded.
4. IPV4 and IPV6 are received simultaneously for the same socket
5. Structure is filled with information from datagrams both IPv4 and IPv6
6. Datagrams are sent in sequence mixing IPv4 and IPv6 using nwp servers.
7. Datagrams are sent in large quantities to the server to test heavy load response.
8. Memory leak test for all the above mentioned tests. (valgrind ./server)
9. CODE: /server/src/test_server_udp.c

**Intermodule communication**

1. Message queue content is always limited to memory address length because we work with pointers (threads share memory space)
2. Structure is pushed into message queue
3. Structure is correctly pulled from message queue
4. Address pulled from message queue allows sento()
5. Memory leak test for all the above mentioned tests with valgrind
6. CODE: /server/src/test_server_udp.c

**Master server test scenarios**

1. Master server starts properly
2. Master server receives JOIN message
3. Master server receives DISJOIN message
4. Master server receives UPDATE message
5. Master server receives unknown message
6. Master server receives out of limit datagram
7. Check memory for memory error and memory leak
8. Datagrams are sent in sequence mixing IPv4 and IPv6 using nwp servers.
9. Datagrams are sent in large quantities to the server to test heavy load response.
10. Erroneous protocol messages.
11. Memory leak test with **valgrind**

**Test 1 :** Master Server starts properly

**1.1 :** Master server gets necessary parameters for start up from terminal(>)

>./mserver -port 5002 -ipv4 127.0.0.1 -ipv6 ::1 -mns 10

Check on another terminal using netstat if it is listening on the above addresses

>netstat -lunp

udp 0 0 127.0.0.1:5002 0.0.0.0:* 14860/mserver

udp6 0 0 ::1:5002 :::* 14860/mserver

Result from netstat shows it has properly parsed the the parameters and started

**1.2** Master server gets necessary parameters to start from configuration file.

In configuration file “address.conf” :

ipv4=127.0.0.1 ipv6=::1 port=5002 maxsrvs=4

In configuration file “serverId=00046” :

serverId=00046

start master server:

>./mserver

Check on another terminal using netstat if it is listening on the configured addresses

>netstat -lunp

udp 0 0 127.0.0.1:5002 0.0.0.0:* 14860/mserver

udp6 0 0 ::1:5002 :::* 14860/mserver

Result from netstat shows it has properly started by reading configuration files

Note: The remaining test are done by builing and running test.c which is found in test folder of

master folder.

To build test: > gcc -o test.o test.c

To run test(before running test start the master server to listen on “127.0.0.1” and “::1” on port 5003):>./test.o

**Test 2 :** Master server receives JOIN message

**2.1 :** Master Server receives JOIN ipv6 and ipv4 messages as new server (receives JOIN$00000)

Result: master server responds by sending JOIN messages containing new server id(send JOIN$00001 for ipv4 and JOIN$00002 for ipv6) for both ipv4 and ipv6 JOIN requests.(PASSED)

**2.2 :** Server sends JOIN ipv4 and ipv6 messages as an old server(having server id SRVID=00001 ipv4 and JOIN$00002 ipv6)

Result: master server responds by sending JOIN messages containing the same server id as the request (send JOIN$00001 for ipv4 and JOIN$00002 for ipv6) for both ipv4 and ipv6 JOIN requests.(PASSED)

Note: The server id could be different from the above server ids.

**Test 3 :** Master server receives DISJOIN message

Master server received ipv4 and ipv6 DISJOIN messages (DISJOIN$00001 for ipv4 and DISJOIN$00002 for ipv6)

Result: master server sent back DISJOIN message(DISJOIN$00001 for ipv4 and DISJOIN$00002 for ipv6).(PASSED)

Note: whether the server who send the DISJOIN messaged is joined or not when the DISJOIN message was received the master server alway sends back a DISJOIN message containing the server id in the request DISJOIN message.

**Test 4 :** Master server receives UPDATE messages

Master server receives one UPDATE message from ipv4 server with id 00001 (UPDATE$00001$MSG4..) another one from ipv6 server with id 00002 (UPDATE$00002$MSG6..)

Result: Master server forwards the UPDATE message from ipv4 server to ipv6 server and the UPDATE message from ipv6 server to ipv4 server.(PASSED)

Note: Servers should successfully JOIN the master server to receive UPDATE.

**Test 5 :** Master server receives unknow message

Master server receives unknow message (DSJOIN&00001) from ipv4 server and unknow message (DSJOIN$00001) from ipv6 server.

Result: Master server does not send any response to unknow messages(PASSED)

**Test 6 :** Master server receives out of limit datagram (max limit is 1048 bytes)

Master server receives out of limit datagram (more than 1048 bytes) from both ipv4 and ipv6 servers

Result: Master server does not send any response (PASSED)

**Test 7 :** Check memory for memory error and memory leak

Start the master server using valgrind:> valgrind ./mserver

Do all the previous tests.

Use netstat to know the process id(pid) of the MS(5003 is the listening port number):

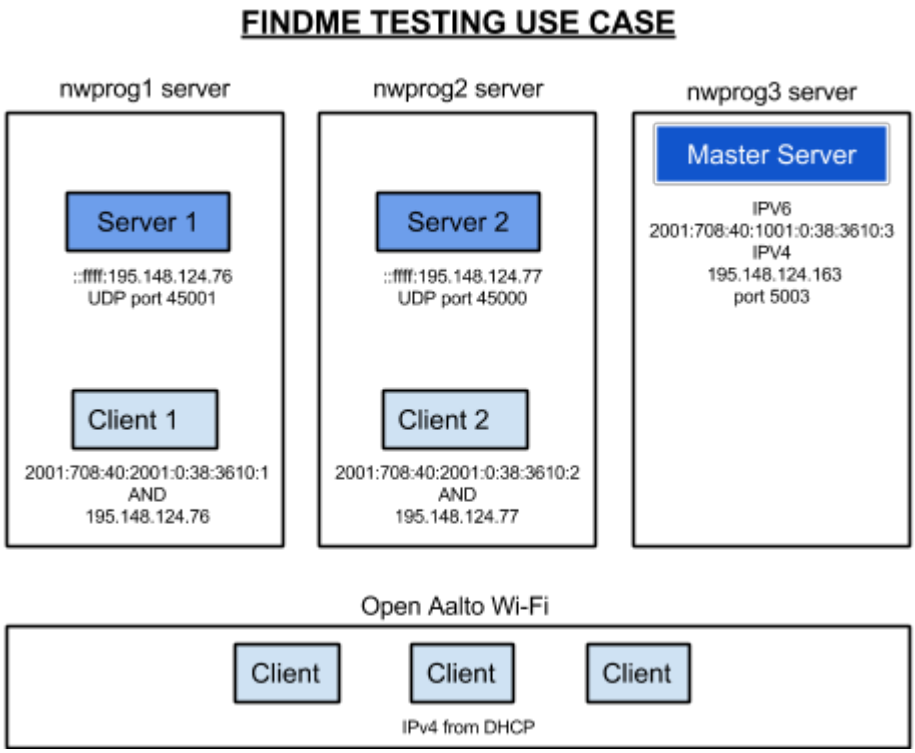
:>sudo netstat -ulp | grep 5003

Kill the MS(master server) process:> kill <mserverpid>

Result : Valgrind result shows no memory error and 1306 bytes still reachable memory which is not considered a memory leak.

A tester application that integrates all tests for all different applications was not developed due to time limitation.

The following figure describes one test scenario including all applications in order to validate the global performance of the FINDME network. This scenario can be tested by using executables and configuration files to be found in git /demo/.




-

This scenario validates the hierarchical architecture of the FINDME network compatible with both IPv4 and IPv6. The service works with clients both in and outside the netlab.hut.fi network.

# 7. Known defects and other shortcomings

| **Client defect and shortcomings** |
| --- |
| ● The application doesn't work as a daemon.<br>● If we input spaces in the name or email to be resolved, the application only takes the word before the first space and ignores whatever is after that space.<br>● Extra tests are needed in order to verify that the application can work with different inputs. Because the application has only been tested with valid inputs, and the only invalid input in the tests was to provide the symbol $ which is used as a delimiter in order to filter to request the user to provide a valid input not containing that symbol, however, other invalid input should be provided to discover programming defects/bugs.<br>● There is a need to test the program to exit in several different situations, because during the verification, only proper exits have been performed, however, there might be memory loss of unfreed pointers in some alternative program exits. |

| **Server defect and shortcomings** |
| --- |
| ● The application doesn't work as a daemon. |
| **Module 1. Data processing module** |
| The transport protocol used is UDP and we do not have any reliability mechanism to cope with out of sequence delivery and packet loss, hence<br>1) Timers can be implemented.<br>2) Retransmission mechanism can be added. |
| **Module 2. UDP module** |
| ● UDP module is executed as a thread therefore in case of heavy load of the server more threads could be instantiated.<br>● Message queue would allow this improvement with no additional synchronization except maybe for mutex in receivefrom(). |

**Intermodule communication**

- Prioritization within the message queue could be implemented to provide preference to server update messages over client messages.
- Message queue can be modified in order to exchange the value of variables in case multi process implementation was required.

**Master defects and shortcomings**

- Since no security measure is implemented the master server can not differentiate between a legitimate server and a malicious one. So anyone who knows the ip address and port number the master server is running on and who knows the messages structure can do among other things the following:
    - Make the master server run out of unique server ids by continuously sending JOIN message as a new server and DISJOIN message.
    - Poison and crowd the database of other servers, that have already joined to the master server, by sending too many bad UPDATE messages to the master server
- Only a single master server is implemented so if it is down the whole network will be down.

# 8. Distribution of work

| General roles | |
|---|---|
| Project Manager | Jaume Benseny |
| Overall integration and testing coordinator | Raghavendra M |
| Documentation coordinator | Jaume Benseny |

| Design and implementation roles | |
|---|---|
| Server application - Module 1 Data processing<br>Server application - Module integrations | Raghavendra M |
| Server application - Module 2 UPD | Jaume Benseny |
| Master server application | Anteneh Adem |
| Client application | Raul Morquecho |

| Project Phases | |
|---|---|
| **Project Phase 1** | |
| Original plan | Design of FINDME basic functionalities, division of tasks and responsibilities. |
| Group Meeting 1<br>12.02.2015<br>Maarit | - Everybody participated<br>- Definition of basic functionalities (location resolution network), applications (server and client), modules implementing such functionalities (server is divided in data processing module, client communication and server communication) and who develops what (Raul client, Raghu data processing, Jaume client communication, Anteneh server communication).<br>- Agreement of communication protocols (client communication is UDP and server communication is TCP)<br>- Check Chapter 10 for all details about the meeting. |
| Group Meeting 2<br>23.02.2015<br>Maarit | - Everybody participated<br>- Discussion of basic architecture and possible implementation of the different modules. Inter-module communication discussion and synchronization among applications.<br>- Details definition of messages to be exchanged between applications. |

| | |
|---|---|
| Results of Phase 1: Arseny meeting 27.02.2015 | - Everybody participated<br>- Arseny approves idea behind the FINDME network. Recommendations are received in terms of general architecture of the application.<br>- First version of the client is shown.<br>- Phase 1 finished with clear idea of architecture, modules to be implemented and requirements. |
| **Project Phase 2** | |
| Original plan | - week 1 (2-8) – Each component develops basic features of its module and tests them.<br>- week 2 (9-15) – Integration of code and general test<br>- week 3 (16-20) – Fixing of general test small improvements. |
| Group Meeting 3 13.03.2015 Maarit | - Raghu, Anteneh and Jaume participated<br>- Everybody presents code developed until today. Raghu presents the data structures that modules are going to use to share information through messages queues. Information about client is stored in a hash table that stays in memory.<br>- Change in the architecture is made: We create a new application called Master Server that will act as a hub and centralize message forwarding. All communication will be UDP. Anteneh takes responsibility for the Master Server instead of the server to server communication. |
| Group Meeting 4 20.03.2015 Maarit | - Module to module meetings have taken place to address specific integration issues.<br>- Module 1 and Module 3. Message queue will exchange pointers instead of data because threads share memory. This will be faster.<br>- Merging will be done by Raghu. Jaume will coordinate with Raul to move udp testing forward. |
| Arseny meeting 2 27.03.2015 | - Everybody participated<br>- Phase 2 has finished and the project is still in the process of integration of Module 1, 2 and 3.<br>- Module 1, 3 and client implement basic functionalities which have also been tested.<br>- Integration of Module 1 and 3 is planned for weekend 21th of April in order to present demo to Arseny the following week. Testing manager role is still pending to be assigned.<br>- Arseny assesses acceptable advance in code and documentation. He likes the new Master server role. |
| **Project Phase 3** | |
| Original plan | - week 1 (23-27) – Continue with integration of modules and parallel testing. |

| | |
|---|---|
| | - week 2 (30-3) – General testing and development of test software.<br>- week 3 (6-10) – Code refinement and documentation elaboration. |
| Group Meeting 5<br>01.04.2015<br>Maarit | - Everybody participated<br>- During this meeting some exchange of messages between client and server was performed. Testing between server and master server advanced.<br>- Updates on documentation were also discussed as new packet diagrams for communication protocol chapter, block diagram for implementation chapter and tables to be filled by all participants.<br>- Strategy for the global test is discussed as well as possible demo scenarios. |

For extended details about discussion during meetings please check Chaper 10 - “Log of work and meetings”.

# 9. References

Pointers to related literature that the group has used as source of information for the work (e.g., on distributed algorithms, etc.)

1) Hash tables - Wikipedia

2) http://www.math.ucla.edu/~wittman/10b.1.10w/Lectures/Lec23.pdf

3) UNIX Application Programming Lecture slides (Risto Järvinen)

# 10. Log of work and meetings

**First meeting 12/02/2015 Maarit**

**Attendance :** We all participated

Data exchange

- Name
- GPS Location
- Postal Address

Server behaviour:

- The servers store location of devices as GPS position.
- We start with 2 servers receiving information from clients.
- Servers synchronize information to keep updates copies of locations
- OPTIONAL - Store queries that can not be processed yet
- Module 1 - Database manager - Raghu
- Module 2 - Tcp manager to keep sync among servers - Anteneh
- Module 3 - Udp manager to accept and replay queries from clients - Jaume

Client behaviour:

- It tries to connect to a pool of addresses and chooses one randomly.
- Informs about present location (automatically)
- Send query
- Receives informs and prints
- OPTIONAL - Ask about client location within a time frame
- Module 4 - Client (takes care of locations and data) - Raul

Protocols:

- UDP between client - server
- TCP between servers
- Client have identifier no security requirements

**Second meeting 23/02/2015 Maarit**

**Attendance :** We all participated

**Questions to be addressed**

We all need to come with our homework done, at least answers to this questions:

- What tasks is my module responsible for?
- What communications my module is involved with? Is acting as server or client?
- What kind of server architecture better addresses my requirements? Do you have firewall / NAT problems?
- With what other modules I share program? Do we share features? Do we share libraries? What interfaces do you plan to use? Pipes? Shared memory?
- Do I need to define a state machine? A timeline execution?
- Is there any module you need to validate your logic?

**Overall architecture**

Server side:

- Thread 1 - Module 1 - Database manager
- Thread 2 - Module 2 -

**Module 1 - Database manager - Raghu**

Module is responsible for

- Generate client IDs
- Receiving querys from clients and encode response
- Receiving msgs from server and change particular entry

With what other modules I share program? Do we share features? Do we share libraries?

- How do we encode messages in the query?
- Communications with Module 3 and Module 2 are message queues.

**Module 2 - Tcp manager to keep sync among servers**

Module is responsible for

- General objective: Synchronizing database between servers
- Send 1 x n notification of database updates

**Module 3 - Udp manager to accept and replay queries from clients**

Module is responsible for:

- UDP communication in the server side
    - Open socket server and listen to clients
    - Replay to clients about other client position
    - Replay to clients about position correcly updated
- UDP communication in the client side
    - Inform server about its new position
    - Request position of other client

What kind of server architecture better addresses my requirements?

- We are talking about a UDP server receiving requests from multiple clients.
- The number of clients can be large but the traffic exchanged is small
- No need for threads. Select with 1 thread can be enough. Max parent + 1 pre-fork child.
- We assume work in IPV6 network with no firewalls.

With what other modules I share program? Do we share features? Do we share libraries?

- I share with both client and server programs
- Client: I have to know when to update my position
- Client: I have to inform about server replays

- Server: Inform about client new position
- Server: Inform about client request
- Communication : message queue.

Do I need to define a state machine? A timeline execution?

- Yes. State machine to coordinate with other modules.
- Timeline execution only for my UDP communication.

Is there any module you need to validate your logic?

- Client: Raul
- Server: Database manager

**Module 4 - Client (takes care of locations and data)**

**CLIENT TO SERVER COMMUNICATION**

NEW CLIENT

- CLIENTID$TYPEOFCOM$SERVICE$<string>$LOCATIONUPDATE\n
- example: 0$\n

QUERY MSG

- CLIENTID$TYPEOFCOM$SERVICE$<string>$LOCATIONUPDATE\n
- example: 000001$QUERY$NAME$raghu$10.19.19\n
- example: 000001$EMAIL$rahgu@aalto.fi$2890182\n

UPDATE MSG

- CLIENTID$TYPEOFCOM$SERVICE$<string>$LOCATIONUPDATE\n
- example: 000001$UPDATE$NAME$raghu$10.19.19\n

**Third meeting 13/03/2015 Maarit**

**Attendance :** Raghu, Anteneh and Jaume

Objective : Present code that everyone has developed until today.

Raghu presents the data structures that modules are going to use to share information through messages queus. Information about client is stored in a hastable that stays in memory.

Master server role:

- Servers only send client's updates to master server
- Master server is responsible for sending updates to all other servers
- OPTIONAL - All servers shall send keep alive msg to master so master can update them properly.
- Master provides ids to all servers
- Master server keeps list of running servers and their status
- OPTIONAL - Servers fight to become master

Decisions:

- **Modules will work as independent threads and exchange data through ONE SINGLE message queue.**
- Server and client modules send recieved information using msg structure
- **Msg strucure is made of client or server address and exchanged message and CLIENT OR SERVER UDP MSG.**
- Database thread will send replays straight to clients and servers using information exchanged in the data strucure.
- **Server to server is UDP connection.**
- At start-up, server will read IP addresses from all servers from a file and try to establish connections
- Server module establishes tcp sockets with a other servers, creates select and shares with database module.

**SERVER TO SERVER (MASTER) MESSAGES**

NEW SERVER

- SERVERID$
- example: 00000001$\n

SERVER BACK ONLINE

- SERVERID$TYPEOFCOM$
- example: 000001$NEW$000001$NAME$raghu$10.19.19\n

**TASKS**

Raghu

- Read from both mq with select and test current core.
- Initialize mutex inside database module
- Sending code to servers and clients
- Share logging code

Anteneh

- At start-up, server will read IP addresses from config file try to send hello word “message connections” with the master.
- Initial SERVER ID is something like 000000001 and shares with master server. In return it gets new ID and stores back to file and stores as global variable.
- Server ID should be stores on disk so it’s premanent
- Develop unit test

Jaume

- 2 UDP socket that listens to different ports and fills msg queue
- Select reads from them and puts to the msgq
- Develop unit test

Further improvements

- How to sync new servers with full database
- Send ACK to clients and maybe check for integrity with checksum

Integration date : **Integration meeting on Friday 20th in Maarit.**

**Forth meeting 20/03/2015 Maarit**

**Attendance :** Module to module meetings have taken place to address specific integration issues.

Objective : Accelerate integration.

Module 1 and Module 3.

Message queu will exchange pointers instead of data because threads share memory. This will be faster.

Merging will be done by Raghu. Jaume will coordinate with Raul to move udp testing forward.

**Fifth meeting 01/04/2015 Maarit**

**Attendance :** All

Objective : Testing of applications.

We have 3 applications communicating: client, server and master server. During this meeting correct exchange of messages between client and server was performed as well as proper forwarding by the master server.

Updates on documentation were also discusssed as new packet diagrams for communication protocol chapter, block diagram for implementation chapter and tables to be filled by all participants.

Strategy for the gobal test is discussed as well as possible demo scenarios.