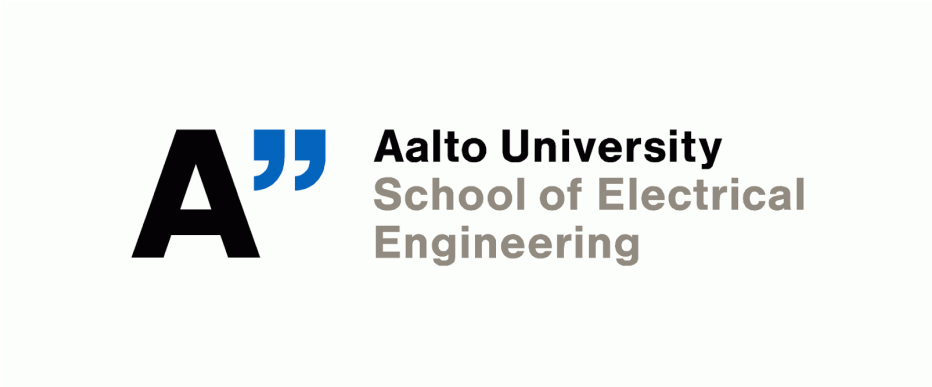


S-38.3610 Network Programming

Spring 2015

Phase 2 Report

20.03.2015

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

Ending Mar 20th: initial implementation must exist

- There must be an implementation that compiles and "roughly" works, even if there are known defects and missing features
- **Deliverable: 1) document sections 1, 2, 3, 4, 6, 8 and 9 must exist. On testing, an initial plan is provided, but test results are not yet needed. 2) There is an initial implementation in git repository.**
- Advisor will meet the group after this phase on agreed time
- Everyone reviews one other project and delivers feedback by Apr 3th

1. Overview and overall architecture

Overview

FindMe is a name resolution application, in which one can query the information of a node (user) having only a piece of his/her information. For example, a query can be about knowing the complete information about a node just by sending its name. A query not necessarily has to be the name of the node, instead an email address can be used, and also location or street address information can be sent as part of the query to know rest of the information.

Github repository

https://github.com/antenehd/FINDME

Architecture

FindMe can be divided into two separate applications:

1) **Client application:**

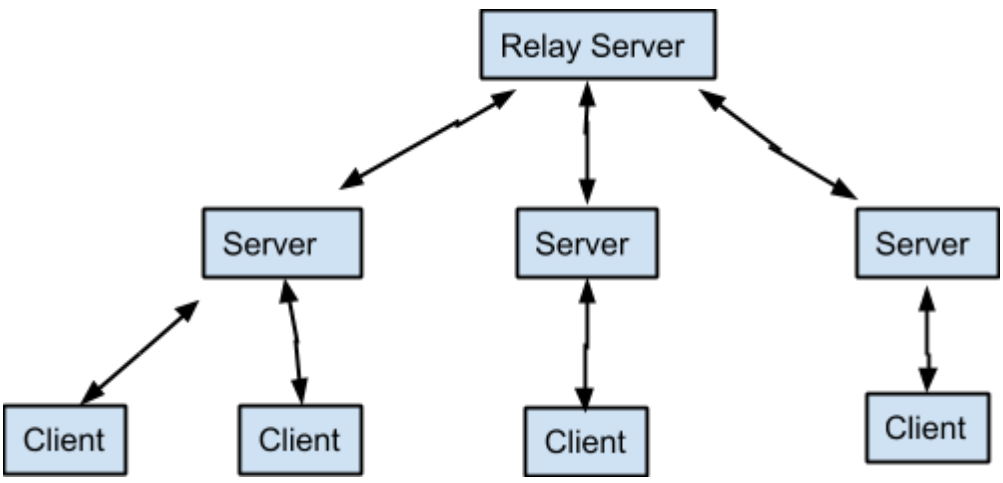
Client construct the query based on the input of the user, send the query to the server and waits for a reply, once the reply is received, the client displays the information to the user.

2) **Server application:**

Server is responsible for resolving the received queries from clients. The server is divided broadly into three modules.

a. Module 1. Data processing module
b. Module 2. Server interactive module
c. Module 3. UDP module

Overall architecture



2. Requirements description

Requirements description (done in first phase, can be modified later)

- what functions are implemented, how do they behave
- how is the functionality distributed between different components
- how is the software used
- what is the intended operation platform and programming language. If there are library dependencies, etc. mention them here (confirm these with advisor -- software must run on nwprog* hosts)

General requirements

Major requirements to be implemented as part of FindMe:

- Uniquely identify clients
- Handling name resolution query from clients
- Handling an information update message from clients
- Data synchronization between the servers

Server requirements and associated modules

Server behaviour / required features:

- 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
- They must be resilient to one server failure
- OPTIONAL FEATURE - Store queries that cannot be processed yet

Module 1. Data processing module

Module is responsible for

- Generate client IDs
- Receiving queries from clients and encode response
- Receiving messages from server and change particular entry

Inter module communication requirements

- Communications with Module 3 and Module 2 to manage data update petitions.

Module 2. Server interactive module

Server intercation module handle intercations between servers and it contains two submodule, Server synch module and Relay server module.

Server synch module:

- General objective: 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

Relay server module:

- General objective: Synchronizing database between servers
- Sends database updates to all other servers
- Receives database updates from other servers
- Does not have its own database, it only relays database updates between server
- Assign server id for new servers

Network communication requirements

- Periodic communication with relay server.
- Relay server relay database updates for

Inter module communication

- There is no intermodule interaction for Relay server module.
- Server synch module creates socket for sending and receiving database updates to and from the relay server for Module1 and Module 3 respectively.

Relay server requirments:

- receive synch messages from servers
- check the synch message if it contains servers id
    - if server id exists send back an ack message containing the same server id
    - if server id is 0 assign new id and send back an ack message containing the new server id
- receive database updates from other servers
- send database updates to all other servers except the one that send the database update

Server synch module requirements:

- 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 3. UDP Module

Module is responsible for:

- 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
- OPTIONAL - Replay to clients/servers about position correctly updated

Network communication requirements

- Asynchronous communication with clients and servers

Inter module communication

- Communications with Module 1 to manage data update petitions and server updates

Client requirements

Client behaviour / required features:

- Query servers about other clients
- Receives replays from the server and prints it to the screen
- 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.
- 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.
- OPTIONAL FEATURE - Ask about client location within a time frame

Communication requirements

- Asynchronous communication containing small data fields

3. Instructions

Instructions

how to build

Server

TBD - Will be ready for meeting with Arseny week of April 23th.

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_1_2.c -o output_name

Where the compilation input starts after the $ sign, 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 use

Server

TBD - Will be ready for meeting with Arseny week of April 23th.

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

must work in Aalto login servers, or in course test servers

(To be updated before March 20th)

4. Communication protocol

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

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.
- CLIENTID$TYPEOFCOM$SERVICE$<string>$LOCATIONUPDATE\n
- example: 000000000$QUERY

QUERY MSG

- The query message sent by the client is composed by 5 fields plus a terminating end-of-line to recognize when the string is terminated
- The first field is the client ID assigned previously by the server
- The second field is the type of request, it could be QUERY or UPDATE
- The third field indicates what information the string contains
- The fourth field is the information itself
- The fifth update is always the current location of the client
- CLIENTID$TYPEOFCOM$SERVICE$<string>$LOCATIONUPDATE\n
- example: 000001$QUERY$NAME$raghu$LOCATION$10.19.19\n

UPDATE MSG

- 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$TYPEOFCOM$SERVICE$<string>$LOCATIONUPDATE\n
- example: 10000001$UPDATE$NAME$brad$EMAIL$brad@gmail.com$ADDRESS$Espoo$LOCATION$192.168.0.1$TIMESTAMP$12345\n

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 SYNC TO RELAY SERVER

- SYNC=0x00

OLD SERVER SYNC TO RELAY SERVER

- SRVID (one byte)
- eg. SRVID=10 ---> SYNC=0x0A

RELAY SERVER TO SERVER (response to SYNC)

- ACK=SRVID (one byte), SRVID could be new if SYNC was zero otherwise SRVID=SYNC

5. Implementation description

Implementation description

client

server

(IT DOESN'T NEED TO BE DELIVERED UNTIL THE FINAL REPORT)

Module 1. Data processing module

Architecture:

- 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.

Module 2. Server interactive module

Architecture:

- Each server is connected to a relay server. (star topology)

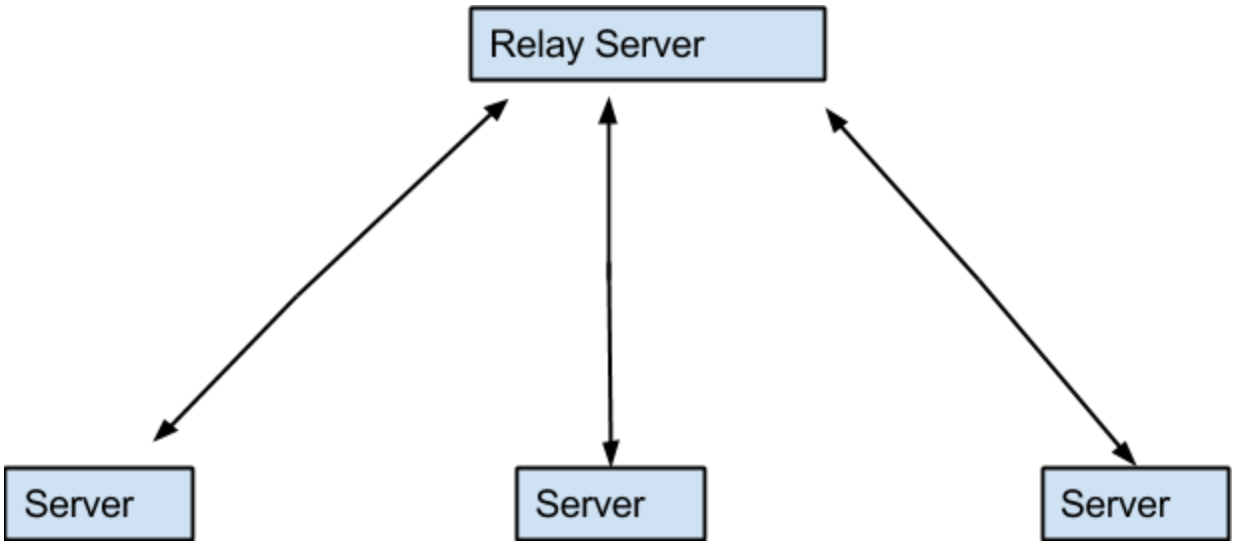


figure: sample architecture for three servers

Implementation:

- UDP is used for connection with relay server
- Servers will be configured to know all peer servers and synch operation should be restricted to only these configured peers.
- When a server starts it contacts the relay server so that it could send and receive database updates.
- Relay server creates two sockets, one for receiving SYNC and sending ACK and another one for relaying database updates (receiving and sending database updates)
- Relay server saves the remote address contained in SYNC datagram to send ACK and database updates to the server afterwards.
- Relay server assigns new server ids sequentially so the maximum server id that is already assigned will be saved in a file.
- server gets the relay servers ip address and its own server id,if it is an old server, from a file
- server creates socket to send ACK to relay server and afterwards uses same socket to send and receive database updates.
- new server saves its server id in a file
- servers get unique server id from relay server

Module 3. Client interactive module

- 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.

Module 4 - Client

What kind of server architecture better addresses the client requirements?

- The server must support sending and receiving data over UDP protocol and the data will be sent/received in string format, as long as the interface is available with this characteristics, the rest of the server architecture doesn't concern the client.

With what other modules the client shares code, features or libraries?

- 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.

Time line execution / flow diagram
(to be updated later)

Is there any module the client needs to validate its logic?

- Yes, it needs the server to be working properly in order to validate a successful working estate, this can be tested on the last stage of the project, however, the server is not expecting to be working properly during the first two thirds of the project, because of that, a simple test server will be created to validate client behaviour, this test server does not have any logic or any complicated structure, it just receives strings through the network socket and prints them to the screen.
- In the final integration, and before the testing part of this project is when the client needs the server to be properly working in order to be validated.

6. Quality assurance

Quality assurance

describe tester, and the tests that have been made

Testing:

- Develop tests along with main code.
    - Every developed will be responsible for the testing of its own code.
- “Tester” software component
    - **We need to define someone responsible for the Tester software component.**
    - Standalone executable that executes a thorough set of tests that cover all software functionality
    - Network tests: send protocol messages in automated sequence
    - Could also include unit tests directly on functions
    - Test also high load, erroneous protocol messages, network disconnections (may need to be simulated)

(To be updated for March 20)

Tested scenarios:

Module 1. Data processing module

1. New client requesting an ID.
2. client sending a query for a not existing record.
3. Request a new ID, Update the content and query for same record.
4. Handle update record message from server.
5. Handling SIGINT
6. New record to a from a server.
7. New record message for an existing record from a server.
8. Memory leak test for all the above mentioned tests.

Module 2. Server interactive module

1. New server joins the network

2. Old server joins the network
3. Relay server accepts more than one SYNC containing same server id
4. Server sends database updates
5. Relay server receives out of limit datagram
6. Server does not get back an ACK
7. Server receives multiple ACKs
8. Memory leak test

Module 3. UDP module

1. IPV4 datagrams are correctly received
2. IPV6 datagrams are correctly received
3. Datagrams with length longer that MAX_MSG_LEN can't be aggregated at this point. Function check_datagram to be elaborated.
4. IPV4 and IPV6 are received simultaneously for the same socket
5. Structure is filled with information from datagrams both IPv4 and IPv6
    a. Raghu wants to change content to pointers, to be tested
    b. After this change, mq content is always limited to memory address length
6. Messages that are too long to be put in the mq are discarded
7. Structure is pushed into message queue
    a. **Adjustment is required to solve msg loss.**
8. Structure is correctly pulled from message queue
9. Memory leak test for all the above mentioned tests. (valgrind ./server)
10. TBD:
    a. Multiple parallel clients/servers overflowing -> Priority to mq msgs? Multiple threads to collect data?

Module 4. Client

For the client module, there will be several stages for testing, the tests will consist of 2 different methods, defect testing and verification testing.

It is worth to notice that for this module, only functional tests will be performed, that is due to the nature of the client, which doesn't need to receive a huge amount of messages, and for that, performance testing will be left out, since the client is meant to only exchange few messages with the server.

- **Verification testing**

For verification testing, the tests will be performed with correct values corresponding to the expected inputs of the object or component being tested. No incorrect inputs will be provided due to the fact that the purpose of this testing is to verify that the software is suitable for its purpose.

- **Defect testing**

During this part of the testing, and by the contrary of verification testing, the objects and components being tested will be input with erroneous values, this is done in order to find defects in the software, and to see how it handles different kinds of inputs, so we would verify that it can receive erroneous values and handle them appropriately.

- **Object testing**

During object testing, the smaller isolated portions of the client module will be tested separately. The smallest isolated portions of client software are, for instance, functions, which receive parameters from the main program, process information internally and then handle back the modified data.

- **Component testing**

Component testing is, in our case, the entire client application, it is classified as a component due to the fact that it belongs to a bigger system, which is the whole FINDME DNS-type system. Once defined that the whole client application is a component, we can then do component testing on it with a simple test server, which only accepts and returns messages to this client component.

When this 2 stages have passed, then the client component will be tested together with the servers in the system testing phase.

7. Known defects and other shortcomings

Known defects and other shortcomings

describe also what would you do more or differently, if you had more time

(To be updated)

8. Distribution of work

Distribution of work

> How was the work distributed, who did what, how much time was used by each group member. Describe each phase separately.
>
> Document when project meetings were arrange

Implementation roles:

Server Side:

- Module 1 . Data processing module - Raghu
- Module 2. Server interactive module - Anteneh
- Module 3. UDP Module - Jaume Benseny

Client Side:

- Complete client implementation - Raul

Work plan for phase 2:

- week 1 (2-8) – Each component develops basic features of its module and tests them.
- week 2 (9-15) – Integration of code and general test
- week 3 (16-20) – Fixing of general test small improvements.

Results after phase 2: Phase 2 has finished and the project is still in the process of integration of Module 1, 2 and 3. Module 1, 3 and client implement basic functionalities which have also been tested. 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.

Work plan for phase 3:

- week 1 (23-27) – Continue with integration of modules and parallel testing.
- week 2 (30-3) – General testing and development of test software.
- week 3 (6-10) – Code refinement and documentation elaboration.
    - **ending Apr 10th**: Deadline for final implementation and documentation. There will be no extensions on deadline. The final demo of the projects is arranged on the same day.

Documentation:

Person responsible for the documentation is the project Manager Jaume Benseny. He is also in charge of keeping meeting notes and logs. The plan is to fill the present document as the project moves forward.

Project meeting logs

Please check the attached document called “Log of work and meetings.pdf” for extra information.

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)