Taneli Hukkinen, 220628
Teemu Kekkonen, 295310
Janne Jaanila, 225733

S-38.3610 Network Programming

Software Project

Documentation
mChat

1. Overview

The goal of the project is to implement a distributed group chat system. It consists of interconnected servers and clients. Clients will have to connect to one of the servers. Both the client and the server parts of the system will be implemented. The clients may join chat groups called rooms and discuss with the other clients in the same room. The users in a room can be directly connected to any one of the servers. Therefore, the messages have to be relayed across the network to all servers.

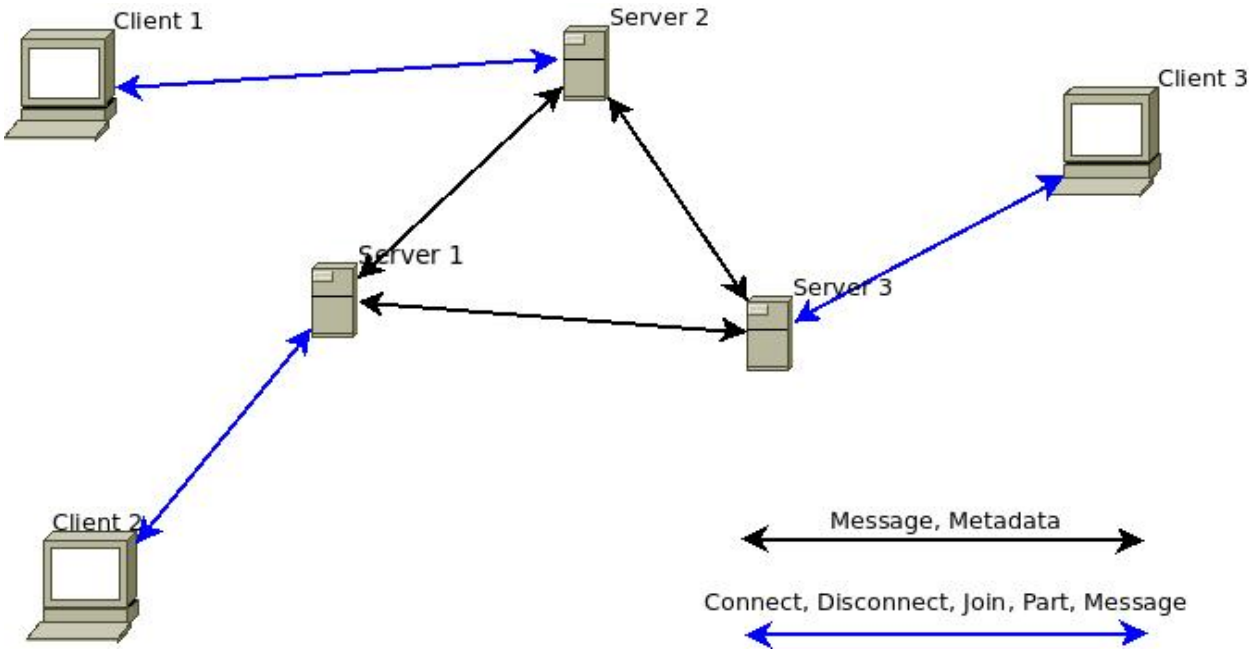

Figure 1. System overview.

2. Requirements description

The software is divided into server and client components. The server maintains connections to other servers and relays messages of clients. In our model, all servers are connected to each other, thus forming a fully connected network. The advantage of this is simple implementation and message relaying. Other servers will be preconfigured to the servers. Disadvantages include a large number of connections that is N*(N-1) where N is the number of servers. Another way could be using a spanning tree of the network like in Internet Relay Chat protocol [1], which would decrease the number of needed connections. However, we decided to use our method for simplicity. The server keeps track of which rooms the directly connected users belong to. Only the messages related to rooms occupied by that particular client are forwarded to it.

The client part consists of a chat user interface and a section that communicates with one server. The user interface will be a simple command line interface as the focus is in the network programming. The client is able to perform following actions: connect to a server, disconnect from server, send message, join room and leave room. Connect and disconnect will create a TCP connection between client and server. Join and leave room will send the server connected to the client an order to perform desired action. The server will keep track of rooms which the client belongs to. Send message will send a message to the server which will then forward it to other servers and clients. Long messages, which are over the maximum of 1024 bytes, are divided into multiple messages.

In addition, a tester component will be implemented for testing the robustness of the servers. The methods that will be used include testing large load, network failures, and unexpected communications such as a malicious or a buggy client.

The intended operation platform for the server software will be Linux and the implementation language will be Python. Compared to a low-level language like C, Python should allow easier cross-platform development and faster development time [2].

3. Instructions

Python 3.2.3 or higher is required to run the server and the client (this is the version on Aalto servers, March 20th, 2015).

Client

Run the client with terminal commands:

```
cd src/client
python3 client.py
```

Client commands:

```
connect <IP> <Port>
disconnect
join <room_name>
part <room_name>
quit
msg <room_name> <message>
nick <new_nick>
help
```

Server

The server (and multiple servers) can easily be run locally. Currently, in Phase 2, testing over the Internet requires editing some code files so that is not covered in this document.

Run the first server:

```
cd src/server
python3 main_without_daemon.py
```

The first server now listens to clients on port 6061.

Run more servers:

```
python3 main_for_testing_additional_servers.py
```

An additional local server is now running and connected to the first server. This command can be run as many times as wanted to start more servers.
Server can be run in daemon mode with main.py:

4. Communication protocol

The communication protocol of mChat system works on top of TCP. Thus, communication is reliable and this protocol does not need separate acknowledgement messages.

Protocol messages have one or more fields separated by spaces and they end in newline. The protocol message fields may not contain spaces or newlines. An exception to this rule is the <message> field of a MSG protocol message. This field may contain spaces but not newlines.

All protocol message fields are UTF-8 encoded text. The maximum length of a protocol message is 1024 bytes. If byte number 1024 of a protocol message is not a newline, the server ignores the newline.

Server-to-Client communication

The server only sends MSG messages to clients. This happens when the client is subscribed to a channel and another client sends message to that channel. The starting tag MSG is followed by a <nick> field that tells the nickname of the sender and a <room> field that tells the name of the channel. The last field of the MSG message is the content of the message. This field may contain spaces.

Client-to-Server communication

A client may send MSG, PART or JOIN messages to the connected server. JOIN and PART are used to tell the server that the client wants to leave or join a chat room. These messages have the target room in the second field.

The client may also send a MSG message to the server. The server only accepts a MSG from a client if the client has first joined the room that the client wants to send a message to. The server forwards a valid MSG to all clients connected to it that are subscribed to that room. The server also forwards the MSG to all other servers.

Server-to-Server communication

A server has two listening sockets for clients and other servers separately.

When a server receives a valid MSG from a client, it forwards it to all other servers. If a server receives an MSG from another server, it forwards it to clients that are subscribed to the room described in <room> field of the MSG message.

The servers distribute information about other servers. For this purpose, message types ALL_ADDRS and MY_ADDR are used. When a new server connects to an existing server, the existing server sends an ALL_ADDRS message to the new server. This message contains addresses and ports of all the servers that are already connected to the existing server. The new server responds to ALL_ADDRS with a MY_ADDR message. MY_ADDR tells the existing server the address and port of the new server. When the existing server receives the MY_ADDR, it accepts the new server as a part of the network. When the new server receives an ALL_ADDRS, it may start connecting to all the other servers that are listed in the ALL_ADDRS message.

Keep-alive messages

The protocol has keepalive messages that are used for removing dead connections. The server sends a HEART message to clients and other servers regularly and expects a BLEED response. If the response is not received within a predefined time, the connection is closed and removed.

The server will also respond to HEART messages from clients with a BLEED response. This means that clients may implement keep-alive messages to check that the server is alive.

List of all message types:

MSG <nick> <room> <message>\n
PART <room>\n
JOIN <room>\n

HEART\n
BLEED\n
MY_ADDR <ip> <listen_port>\n
ALL_ADDRS <ip1> <listen_port1> <ip2> <listen_port2>...\n

Note: The <listen_port> fields in MY_ADDR and ALL_ADDRS messages contain the listen port that the server uses to listen to incoming server connections. It is NOT the same port that listens to incoming client connections.

6. Quality assurance

Tests will be done with Python scripts. The scripts will launch clients and servers with proper settings and use their internal methods to test them. The test module is called test.py.

The main focus in testing is in the stability of the server. Some load testing is also provided.

Common tests

- Compatibility with Python 3.2.3. This system has to run on Aalto machines.
- Ipv4/Ipv6 functionality.

Server

Stability tests
- Too long protocol message fields.
- Missing protocol message fields/ending newline.
- Unallowed actions by client.
    - Joining a room it already belongs to.
    - Parting from room it does not belong to.
    - Sending a message to room it does not belong to.
- Heartbleed functionality.
- Disconnecting server(s) within a network.

High load tests
- One server, multiple clients.
- Multiple servers, multiple clients.

Client

Stability tests
- Unallowed actions by user.

- Heartbleed functionality.
- Disconnecting server.

High load tests
- Maybe some message receiving test.

8. Distribution of work

First phase

The initial meeting was during the topic giving session. Merely the project name and the potential programming languages (C or C++, Python) were decided during that meeting.

We had another project meeting on 12th of February 2015 with all the group members. In that session, we planned the overall software functionality and drafted server network figures. Additionally, different message types and their data were decided at least tentatively.

The first draft of the project plan was written to Google Drive singly. No meeting was arranged but we have also discussed in IRC. The development repository is in GitHub: https://github.com/jjaanila/mChat

Janne wrote most of the Python client in 23.2.2015 and finished it 26.2. It should now be fully operational. Teemu wrote the base of the document.

Second phase

After the first advisor meeting, Python was chosen as the implementation language for both the server and the client.

We had active discussion in IRC throughout the second phase. We also had occasional get-togethers where we coded together and planned the project further.

Janne added heartbleed support to the client.

Taneli designed the server discovery protocol.

Taneli wrote most of the server implementation. Janne did some parts and helped with testing. Teemu helped in discovering and fixing bugs and design flaws.

Teemu started working on testing code.

All of us participated in writing the document.

9. References

[1] Internet Relay Chat, RFC 1459 - https://tools.ietf.org/html/rfc1459
[2] Ousterhout, J.K., "Scripting: higher level programming for the 21st Century," *Computer* , vol.31, no.3, pp.23,30, Mar 1998