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.

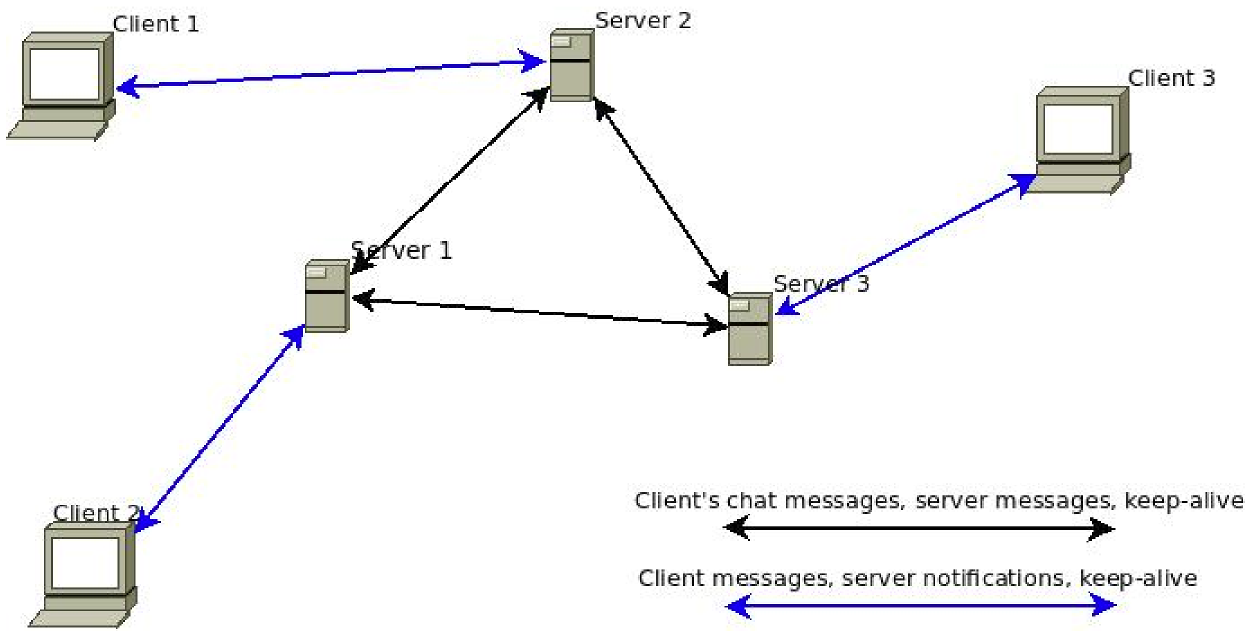
Client 1
Server 2
Client 3
Server 1
Server 3
Client 2
Client's chat messages, server messages, keep-alive
Client messages, server notifications, keep-alive

Figure 1. System overview.

Figure 1. above shows the network topology. Clients interact with a server with client specific messages. E.g. with a JOIN message a client can join a room or with PART it can leave it. Most of this traffic happens between the client and the server the client is directly connected to. Only chat messages (message type: MSG) will be broadcasted to other servers. Server notifications are forwarded to the client from the server network.

Between the servers there are, in addition to client chat messages, messages used in communication between servers and server notifications (they are included in server messages) which are forwarded to clients.

All the links are covered with a heartbeat like keep-alive functionality called Heartbleed.

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. New servers can be dynamically added to the server network. 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 UTF-8 characters, are divided into multiple messages.

As additional features, we will have private rooms, which can be joined with the room name and the room's password. Private rooms are implemented in client only and the server has no idea on which room is private and which is not. The server will also notify other users if a client joins or leaves a room they are in.

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 and client 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, April 10th, 2015).

Client

Run the client with terminal commands:

```
cd src/client
python3 client.py [-d]
```

-d Disables all prints for cleaner testing. Should not be needed in normal use.

Client commands:

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

The commands are quite self-explanatory except for “part”. It is IRC slang and means leaving.

Server

The server (and multiple servers) can be run with following instructions.

The format of a server start command is:

```
python3 main.py start \
<DNS or IP> <Listen port for clients> <Listen port for servers> \
[Existing server’s IP] [Existing server’s listen port for servers]
```

E.g. Run the first server:

```
cd src/server
python3 main.py start localhost 6061 6062
```

The first server now listens to local clients on port 6061. As you can see, existing server’s address can be left out if there is no existing network to connect to. This way we start a new server network.

E.g. Run another server:

```
python3 main.py start localhost 6063 6064 localhost 6062
```

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 (just remember to alter the ports).

The server will be started as a daemon by default. You can change this behavior by giving main.py a flag --non-daemon.

A server can be stopped with command

```
python3 main.py stop <DNS or IP> <Listen port for clients>
```

E.g. This would stop our first server

```
python3 main.py stop localhost 6061
```

Running a server which is connectable from internet requires binding the server to computer's network address, not localhost.

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 characters. If at the latest character number 1024 of a protocol message is not a newline, the server ignores the received data.

List of all message types:

MSG <nick> <room> <message>\n
SYSTEM <room> <message>\n
PART <room>\n
JOIN <room>\n
NICK <new_nick>\n
HEART\n
BLEED\n
MY_ADDR <ip> <listen_port>\n
ALL_ADDRS <ip_1> <listen_port_1> <ip_2> <listen_port_2> … <ip_n> <listen_port_n>\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.

Server-to-Client communication

The server only sends MSG and SYSTEM messages to clients. MSG is forwarded when the client is subscribed to a room and another client sends message to that room. 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 room. The last field of the MSG message is the content of the message. This field may contain spaces.

SYSTEM message is sent after some specific event happens. Currently, these messages are sent only if a client joins a room, leaves a room or changes its nick. The message is sent to all clients who have subscribed to that room (also the newcomer if JOIN is the event). SYSTEM message has three fields: SYSTEM <room_name> <message>\n. Message field contains the actual message which will show up to clients.

Client-to-Server communication

A client may send 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. Joining a private room does not affect the communication level in any way for ordinary JOIN message is used for that.

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.

Client has a possibility to change it's nickname with NICK message. Changing nick is possible also by sending a new MSG with a different nick. This will trigger the nick change on the server side. NICK message has only one extra field, besides the NICK tag, the new nickname.

Server-to-Server communication

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

When a server receives a valid MSG from a client, it forwards it to all other servers. If a server receives a 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 keep-alive 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. This is implemented in our client application.

5. Implementation description

Common

Python uses unicode encoding which means that the maximum size of a character is four bytes. Because our maximum message length is 1024 character we needed 4096 bytes long buffers. It is not used fully almost ever in western countries but we have the support now anyway.

Client

The client software is implemented with Python 3.2.3. Most of the functionality is in client.py file but the UI and some helper classes have been extracted to own files. Timers.py is used for connection related timers and rooms.py describes different types of rooms. Fancyui.py is a very simple command line UI.

The network programming part of client was quite straight forward TCP networking. It supports domain names and IPv6. It has a proper data sending and receiving methods. It can poll the server with HEART messages and disconnects if the server is lost and it answers the server to its polls. All the basic actions of mChat system are implemented. It all happens in the same thread except the sending of periodical HEART message. It uses a “timed thread” which is invoked every n:th second. One interesting design choice was to implement private rooms as client-side only. The client will calculate a SHA-1 hash out of given room name and password which is then used as room name when communicating with the server. The user of client can still see the original room name.

Client does not log anything as it was not required.

Server

The server fully implements our specification. It is a single-threaded, non-blocking server by design. It uses the select() call in a while loop to check which clients and servers and which of the two listen sockets can be read data from. Daemonization is implemented by subclassing the Python 3 version of the public domain Daemon() class, available in [3].

The core of the server logic lies in __start_server() method of MChatServer() class. Run() is a wrapper method for __start_server() that handles keyboard interrupts, SIGTERMs and any possible unhandled exceptions of __start_server().

Logging

Server logs events to two separate files <hostname>_<port>_server.log and <hostname>_<port>_server.log.1. When the first file reaches size of 5MB, it is renamed as the latter one and a new <hostname>_<port>_server.log is created. When this new file is full, the .1 file is replaced with it and a new log file is created. Therefore there is always 5-10 MB of log available (once the first 5 MB have been logged).
Logged events:

- Server starts.
- Server shuts down.
- New client connects.
- Client disconnects.
- New server connects.
- Server disconnects.
- A client tried to connect but the server is full.
- A server tried to connect but the server is full.
- Connecting failures.
- An uncaught exception occurs.

6. Quality assurance

Tests were planned to be done with Python scripts. The scripts will launch clients and servers with proper settings and use their internal methods to test them. The main focus in testing is in the stability of the server. Some load testing was also planned.

Networking between multiple computers was tested manually by setting up servers on several interconnected machines. The servers were started with main.py. Each server except the first one was given the IP address and server port of another server. The servers correctly found the other servers by exchanging the information of the servers they already know.

Planned common test cases

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

Planned server test cases

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.

Planned client test cases

Stability tests

- Unallowed actions by user.
- Heartbleed functionality.
- Disconnecting server.

High load tests

- Maybe some message receiving test.

Executed tests

- Compatibility with Python 3.2.3.
- Ipv4/Ipv6 functionality.
- Missing protocol message fields/ending newline.
- Sending only newlines from client.

Some of the tests (like illegal join/part) have been tested offline during development but network tests have not been done.

7. Known defects and other shortcomings

Communication Protocol

It is unnecessary to send client's nickname in the MSG message, since we have a separate NICK message and the server knows the nickname of each client connected to it. On the other hand, an alternative to the SYSTEM messages could be versions of JOIN and PART for both server and client.

The protocol should have some way for the server to answer to failed JOIN, PART or other attempts. The current specifications does not have any NACK type message available.

Client

In the client, sometimes the system call "clear" fails and the whole print history is shown twice on screen. This has happened especially with the nick command. This bug seems to be hard to replicate. The UI usually repairs itself after the next command.

The client has also another more severe bug: if the user is writing a message and another message is received, the written message disappears. This could be fixed with using some more sophisticated way of UI implementation like curses library. However, the UI was not the subject of this project so it was kept pretty simple.

Server

We could have used epoll instead of select to check sockets for incoming data. It would have been a faster way to do the same thing.

The server fails to be entirely non-blocking. It uses select() to find the sockets that can be read, but it doesn't use non-blocking sockets. Therefore the server may block, even though that is a very rare case.

8. Distribution of work

Janne Jaanila:

- client code
- signal handling

Taneli Hukkinen:

- server code

Teemu Kekkonen

- testing
- DNS support

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.

Third phase

Janne added new features to the client and added some minor functionality to the server. He also participated writing the document.

Taneli added support for system messages in the server and made the server store clients' nicknames. He added support for NICK messages to the server. Taneli fixed all remaining server related bugs that were open in our GitHub issue tracking. Taneli also took part in writing the final document.

Teemu did mainly manual testing and wrote the document a little.

9. References

[1] Oikarinen, J., Internet Relay Chat, RFC 1459, May 1993 - https://tools.ietf.org/html/rfc1459 (opened 10.04.2015)
[2] Ousterhout, J.K., "Scripting: higher level programming for the 21st Century," *Computer* , vol.31, no.3, pp.23,30, Mar 1998
[3] Marechal, Sander, A simple unix/linux daemon in Python, Feb 2007 - http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/ (opened 10.04.2015)