Introduction

This document discusses the DuckChat instant messaging application, both how it works and how to use it. DuckChat is a UDP styled chat program where client may connect to one or more servers and chat with one another. This application was designed and built for a networking course. I have since then improved and added more functionality to the application.

There are two versions of the application, stored in the two directories *duckchat* and *duckchat_v2*. The first version uses just a single server; multiple clients may connect and communicate only on the single server. The second version is an extension of the first; multiple servers can now be run in parallel to reduce individual server load. Multiple clients can connect to different servers and chat with one another across the different servers. More on how they work is discussed later.

Make Instructions

The following instructions apply to both DuckChat project directories.

To compile the whole project, type 'make all' to compile both the server and client executables. You may also type 'make client' and 'make server' to separately compile the client and server executables, respectively. To clean the project, type 'make clean'; this removes all object and executable files from the directory, 'cleaning' the directory.

If you want to build a TAR archive of the project, type 'make tarfile'. The entire project will be compressed and packed into a tar archive with the .tgz extension. You can then unpack the archive by executing the command:

```
tar -xvzf <file.tgz>
```

where <*file.tgz*>[1] is the archive file. To list all the targets, type 'make help'.

Client

The following instructions apply to both DuckChat project directories.

Executing the Client

Before you use any clients, make sure that you have at least one server running to connect to.

The usage for the client is the following:

```
./client server_socket server_port username
```

where *server_socket* is the hostname that the server is running on, *server_port* is the port number the server is listening on, and *username* is the desired username. The hostname must be valid and a server must be listening on that hostname. The server port must be in the range

[1] To unpack the archive, run the command `tar -xvzf <file.tgz>`.

[0, 65535][2]. The username specified must be within a certain length and must not already be used by another client.

For example, if we want to connect a client to the host 127.0.0.1 on port 4000, and we want the username user1, we run the following:

```
./client 127.0.0.1 4000 user1
```

If the client successfully connects, and the username is unique, you should see the following:

```
[Client]: Establishing connection...
---------------  DuckChat  ---------------
Connected to: 127.0.0.1:4000
Logged in as user1
Type '/help' for help, '/exit' to exit.

>
```

Using the Client

Once you're client is connected, you may now chat with other connected users. To send a message to the other users, simply type a message and press enter. All other users listening on the channel will receive the message. When the message begins with a forward slash (/), it is interpreted as a special command. The commands supported are listed below:

• /join <channel> – Subscribes you to the channel <*channel*>, creating it of it doesn't exist.

• /leave <channel> – Unsubscribes you from the channel <*channel*>.

• /list – Lists all the available channels on the server.

• /who <channel> – Lists all the users currently listening on the channel <*channel*>.

• /switch <channel> – Switches your listening channel to <channel>, only if you are already subscribed.

• /subscribed – Lists all the channels you are currently subscribed to.

• /whoami – Displays your username.

• /whereami – Displays the address of the server you are connected to.

• /clear – Clears the terminal screen.

• /help – Displays all the possible special commands and their function.

• /exit – Exits the client application.

[2] Ports 0-1024 are reserved on most systems for privileged services, therefore are mostly not used/accessible.

You may be subscribed to multiple channels at once, but you may only send messages to one channel at a time, your 'active' channel. The most recently joined channel is your active channel, and you may switch to another with the '/switch' command. When you leave a channel, you will no longer receive any messages from other users on that channel.

When you receive a message, it will appear in the following format:

```
[channel][username]: message
```

The username is the client who sent the message, and the channel is the channel they sent the message on.

Troubleshooting

The following error messages may appear while attempting to use or launch the client. A diagnosis and solution is offered for each.

```
[Client]: Path to domain socket length exceeds the length allowed.
```

The hostname you specified is too long; a number in parenthesis will follow this message that specifies the maximum length the hostname can be. Make sure the hostname is shorter than this length.

```
[Client]: Failed to locate the host.
```

The client failed to locate the host; either due to a connection error the hostname is invalid. Check your network status and settings, and make sure the hostname is valid.

```
[Client]: Server socket must be in the range [0, 65535].
```

The socket you specified is out of bounds. Make sure the socket specified is in the required randge.

```
[Client]: Server timed out.
```

The client failed to connect to the server, either it was refused or took too long. Make sure that the hostname and port number specified match with the server's address.

```
[Client]: The specified username is already in use.
```

The username is already occupied by another client. You will need to use a different username.

Server 1.0

This section covers the first version of the server, located in the *duckchat* directory.

Executing the Server

The usage for the server is the following:

```
./server domain_name port_num
```

where *domain_name* is the host address for the server to bind to, and *port_num* is the port number the server will listen on. The host address may be 'localhost', the IP hostname of the machine that the server is running on, or the IP address of an interface connected to the machine.

Just like the client, the arguments must be valid; the hostname must be valid and the port number must be within the valid range.

For example, if we want to run the server on localhost listening on port 4000, we run the following:

```
./server localhost 4000
```

or

```
./server localhost 127.0.0.1 4000
```

If the server successfully binds itself to the requested address, you should see the following:

```
[02/01/2018 11:51:02] Duckchat server launched addressed at 127.0.0.1:4000
```

Note that the message has a timestamp, and likely won't be the same as shown in this example.

Using the Server

The server runs on its own and requires no user input or attention. The server outputs debugging and logging messages, displaying what all the connected clients are doing. This includes all messages sent by all clients, all *list* and *who* requests made, and all channels created, joined, left, and removed. A sample output of what the server would display is shown below:

```
[02/01/2018 11:51:02] Duckchat server launched addressed at 127.0.0.1:4000
[02/01/2018 11:54:09] user1 logged in from 127.0.0.1:38842
[02/01/2018 11:54:09] user1 joined the channel Common
[02/01/2018 11:54:24] user2 logged in from 127.0.0.1:55076
[02/01/2018 11:54:24] user2 joined the channel Common
[02/01/2018 11:54:30] [Common][user1]: "Hi"
[02/01/2018 11:54:37] [Common][user2]: "Hello"
[02/01/2018 11:54:40] user1 created the channel room
[02/01/2018 11:54:40] user1 joined the channel room
[02/01/2018 11:54:52] user2 listed available channels on server
[02/01/2018 11:54:58] user2 listed all users on channel room
[02/01/2018 11:55:09] user1 kept alive
[02/01/2018 11:55:12] user2 logged out
[02/01/2018 11:55:16] user1 logged out
[02/01/2018 11:55:16] Removed the empty channel room
```

As you may notice, the server implicitly handles empty channels by removing them once no more users are listening on the channel. The server also features soft-state tracking of each connected user. This means that the server will forcefully log out any inactive users that have

not sent a packet with a certain time period. The clients will send a packet regularly to keep them logged into the server (to prevent logged in clients from being logged out if the client hasn't sent a message in a while). This is to prevent from any crashed clients from permanently remaining in the server's database(s).

To stop the server from running, simply press CTRL+C or send a SIGINT to the running process. To do this, run the following command:

```
$ ps -a
PID   TTY                TIME  CMD
1351  pts/0          00:00:00  server
1382  pts/2          00:00:00  ps
$ kill -SIGINT 1351
```

You should then see the message displayed by the server:

```
[02/01/2018 12:11:34] Duckchat server terminated
```

Note that the PID shown in this example will not likely be the same on your system. It should be the PID corresponding with the command *server*.

Troubleshooting

The following error messages may appear when trying to run the server. A diagnosis and solution are offered for each.

```
[Server]: Failed to locate the host.
```

The domain name specified is not valid or doesn't exist. Make sure you specify the right hostname and no typos are present.

```
[Server]: Failed to assign the requested address.
```

The specified hostname and/or port number are already occupied. You must specify a different hostname and/or port number.

```
[Server]: Server socket must be in the range [0, 65536].
```

The server socket specified is not valid. You must specify a socket within the given range.

Server 2.0

This section covers the second version of the server, located in the *duckchat_v2* directory.

Executing the Server(s)

The server can now take multiple arguments. The first two arguments are the hostname and port number for the server to bind to (just like the previous version), and any additional arguments

specified after are the hostnames and port numbers of any additional servers that are adjacently connected. The usage is shown below:

```
./server domain_name port_num [domain_name port_num] ...
```

A script is also provided in the project directory, called *start_servers.sh*. You may open the file, uncomment your desired topology, and then execute the script. This is recommended over running the servers individually.

If we want to run the servers that build the following topology

4000 --------- 4001 --------- 4002

we would need to run the following:

```
./server localhost 4000 localhost 4001
./server localhost 4001 localhost 4000 localhost 4002
./server localhost 4002 localhost 4001
```

The server on port 4000 is told to connect to the server on port 4001. Likewise, 4001 is connected to 4000 and 4002, and 4002 is connected to 4001.

Another example; we want to build the topology:

4000 ---------- 4001

| |

4002 ---------- 4003

we run the servers as shown:

```
./server localhost 4000 localhost 4001 localhost 4002
./server localhost 4001 localhost 4000 localhost 4003
./server localhost 4002 localhost 4000 localhost 4003
./server localhost 4003 localhost 4001 localhost 4002
```

Using the Server(s)

Like the previous version, no input is needed; the server(s) run on its own.

An example output of the servers is shown below:

```
127.0.0.1:4001 Duckchat server launched
127.0.0.1:4000 Duckchat server launched
127.0.0.1:4000 127.0.0.1:50738 recv Request VERIFY user1
127.0.0.1:4001 127.0.0.1:4000 recv S2S VERIFY user1
127.0.0.1:4001 127.0.0.1:50738 send VERIFICATION user1
```

```
127.0.0.1:4000 127.0.0.1:4001 send S2S VERIFY user1
127.0.0.1:4000 127.0.0.1:50738 recv Request LOGIN user1
127.0.0.1:4000 127.0.0.1:50738 recv Request JOIN user1 Common
127.0.0.1:4001 127.0.0.1:4000 recv S2S JOIN Common
127.0.0.1:4000 127.0.0.1:4001 send S2S JOIN Common
127.0.0.1:4000 127.0.0.1:50738 recv Request SAY user1 Common "hello?"
127.0.0.1:4001 127.0.0.1:4000 recv S2S SAY user1 Common "hello?"
127.0.0.1:4001 127.0.0.1:4000 send S2S LEAVE Common
127.0.0.1:4000 127.0.0.1:4001 send S2S SAY user1 Common "hello?"
127.0.0.1:4000 127.0.0.1:4001 recv S2S LEAVE Common
127.0.0.1:4000 127.0.0.1:50738 recv Request LIST user1
127.0.0.1:4001 127.0.0.1:4000 recv S2S LIST
127.0.0.1:4001 127.0.0.1:50738 send LIST REPLY
127.0.0.1:4000 127.0.0.1:4001 send S2S LIST
127.0.0.1:4000 127.0.0.1:50738 recv Request WHO user1 Common
127.0.0.1:4001 127.0.0.1:4000 recv S2S WHO Common
127.0.0.1:4001 127.0.0.1:50738 send WHO REPLY Common
127.0.0.1:4000 127.0.0.1:4001 send S2S WHO Common
```

The logging and debugging messages are formatted differently than the previous version. Since the multiple servers are typically run on one terminal screen, each log message has two addresses followed by the log message itself. The left address is the server sending and receiving the message, and the right address is the server the messages are sent and received from. In this example, we have a simple, 2-server topology where user1 logs in, joins the channel Common, sends a message, lists the channels and users on Common.

In order for the servers to reduce individual load, channel subtrees are formed inside the topology. When users join a channel and send messages across the network, a channel subtree is formed among the servers. Only servers who have connected clients on the channel are sent messages. For example, suppose we have the following topology:

```
4000                 4004
  |                    |
4001 ------ 4003 ------ 4005
  |                    |
4002                 4006
```

Then, suppose two clients connect; user1 connects at 4000, user2 connects at 4003. Now suppose user1 sends a message on the same channel as user2, then you should see the following output:

```
127.0.0.1:4000 127.0.0.1:35936 recv Request SAY user1 Common "hello"
127.0.0.1:4001 127.0.0.1:4000 recv S2S SAY user1 Common "hello"
127.0.0.1:4001 127.0.0.1:4002 send S2S SAY user1 Common "hello"
127.0.0.1:4001 127.0.0.1:4003 send S2S SAY user1 Common "hello"
127.0.0.1:4002 127.0.0.1:4001 recv S2S SAY user1 Common "hello"
127.0.0.1:4002 127.0.0.1:4001 send S2S LEAVE Common
127.0.0.1:4001 127.0.0.1:4002 recv S2S LEAVE Common
127.0.0.1:4003 127.0.0.1:4001 recv S2S SAY user1 Common "hello"
127.0.0.1:4003 127.0.0.1:4005 send S2S SAY user1 Common "hello"
127.0.0.1:4005 127.0.0.1:4003 recv S2S SAY user1 Common "hello"
127.0.0.1:4005 127.0.0.1:4004 send S2S SAY user1 Common "hello"
127.0.0.1:4004 127.0.0.1:4005 recv S2S SAY user1 Common "hello"
127.0.0.1:4004 127.0.0.1:4005 send S2S LEAVE Common
127.0.0.1:4005 127.0.0.1:4006 send S2S SAY user1 Common "hello"
127.0.0.1:4006 127.0.0.1:4005 recv S2S SAY user1 Common "hello"
127.0.0.1:4006 127.0.0.1:4005 send S2S LEAVE Common
127.0.0.1:4005 127.0.0.1:4004 recv S2S LEAVE Common
127.0.0.1:4005 127.0.0.1:4006 recv S2S LEAVE Common
127.0.0.1:4005 127.0.0.1:4003 send S2S LEAVE Common
127.0.0.1:4003 127.0.0.1:4005 recv S2S LEAVE Common
127.0.0.1:4000 127.0.0.1:4001 send S2S SAY user1 Common "hello"
127.0.0.1:4003 127.0.0.1:35637 recv Request SAY user2 Common "hi"
127.0.0.1:4001 127.0.0.1:4003 recv S2S SAY user2 Common "hi"
127.0.0.1:4001 127.0.0.1:4000 send S2S SAY user2 Common "hi"
127.0.0.1:4003 127.0.0.1:4001 send S2S SAY user2 Common "hi"
127.0.0.1:4000 127.0.0.1:4001 recv S2S SAY user2 Common "hi"
```

Here, we see the message gets broadcasted to server 4001, which then broadcasts it to 4002 and 4003. However, since 4002 has no listening clients, it replies with a leave request. The same goes for servers 4004, 4005, and 4006. As a result, the following subtree for that channel is then formed:

4000

|

4001 -------- 4003

Every so often, the servers will broadcast join requests for all the channels to all its neighbors, refreshing the entire tree. The channel subtree will then be rebuilt after the next message that gets sent. This is to guard against network failures.

The servers also have soft-state tracking features for other servers. If a server crashes, all its neighbors will detect this from not receiving any packets, and will remove the server from the topology. For example, suppose we have the following topology:

4000 ------- 4001 -------- 4002

| | |

4003 ------- 4004 -------- 4005

Now suppose server 4001 crashes for some reason. After detecting the crashed neighbor, the topology will update itself to the following:

4000 4002

| |

4003 ------- 4004 -------- 4005

The clients can then continue to communicate with one another.

A few things to note from crashed servers. If you have a linear topology and a middle server crashes, clients in-between the crashed server will no longer be able to communicate. The period between the server crashing and the topology update will prevent clients form logging on and sending *who* or *list* requests, but may still communicate between unaffected servers. After the update, everything should be back to normal. If you want to restart the crashed server, you will need to restart the entire topology.

Troubleshooting

All error messages that may appear from running the server(s) are the same as the previous version. Refer to the troubleshooting section under *Server 1.0*.