Clemson Relay Chat Network
CPSC 3600 - Network Systems
Fall 2021

1 Project Introduction

In this project you will be creating a distributed network based on a heavily modified version of the Internet Relay Chat protocol. Building on your previous knowledge, this project will introduce 1) how to use selectors to support simultaneous, non-blocking processing of multiple sockets, and 2) how to implement a distributed network capable of maintaining a shared state across all machines in the network.

2 Project File Structure

The template for the project contains several files and folders. A brief explanation of each file is given below. You will be responsible for completing the CRCServer.py file.

ChatServer.py This file specifies the behavior of the chat servers. Several functions have been stubbed out that are used by the test manager. More detail is provided about each of these functions in comments in ChatServer.py.

ChatClient.py This file specifies the behavior of the chat clients. I've provided you will a fully implemented ChatClient. Feel free to examine the code, but you do not need to modify it in any way.

NOTE: The client is implemented using threads rather than selectors. Do not implement your server using threads. You must used selectors in your server for this project.

ChatMessageParser.py This file will be used to parse messages transmitted by your users and servers. Use the static function *parse_messages()* in the MessageParser class to convert a byte string into a list containing one or more messages. Different message types are represented using different classes. Take a look at these classes for more information about how to use them.

The message protocols are defined in section 4. You will see class properties inside of each Message class corresponding to the parameters defined in the message protocol. Each Message Class also defines 1) the variable_message_length of a given message (based on how long the content actually is), 2) the bytes that were unpacked to create this instance of a message, and 3) a static function that can be used to create the appropriate byte array for a given message type. The code snippet below gives you an example of how to use these properties and functions.

```
# There are six important message classes:
# * MessageParser
# * ServerRegistrationMessage
# * ClientRegistrationMessage
# * StatusUpdateMessage
# * ClientChatMessage
# * ClientQuitMessage

# Assume that one of more messages has been read into the msg_bytes variable prior
# to this code snippet. The static function parse_messages in the MessageParser class
# can be used to convert the byte representation of a message into the corresponding class.
# MessageParser.parse_messages() returns a list containing each of the messages present
# in the provided bytes
messages = MessageParser.parse_messages(msg_bytes)

# Use the static bytes() function present in each message class to create an appropriate
# byte representation for a given type of message. Each bytes message accepts the specific
# parameters required for a given type of message.
server_reg_msg = ServerRegistrationMessage.bytes(msg_source_id, last_hop_id,
                                                 server_name, server_info)
client_reg_msg = ClientRegistrationMessage.bytes(source_id, last_hop_id, client_name,
                                                 client_info)
chat_msg = ClientChatMessage.bytes(source_id, destination_id, content)
status_msg = StatusUpdateMessage.bytes(source_id, destination_id, message_code, content)
client_quit_msg = ClientQuitMessage(source_id, content)
```

ChatTestManager.py This file is what you will use to test your program. I've provided a series of test cases (see the TestCases/ folder) that evaluate different aspects of your code. The ChatTestManager will run these test cases and automatically launch the different servers and users required for each test.

The different tests are specified in a dictionary at the end of ChatTestManager.py. I recommend that you work on one test at a time. You can comment out all other tests to avoid running tests that you have

confirmed are working, or that you know will not work yet. This will save you time while testing. You should not edit any of this code other than commenting/uncommenting the tests you want to run.

Testers/ This folder contains some helper classes to run the different tests used by ChatTestManager.py. You should not edit any of this code.

TestCases/ This folder contains the specific test cases. Each file specifies what servers and users should be launched, what user commands should be sent (i.e. send a message, quit the network, etc), and what the expected state of the network is at the completion of the test. You should not edit any these files.

3 Reading and Writing to Multiple Sockets Simultaneously

As we've discussed in class, many socket operations are blocking operations. This means that, upon calling these methods, all other code execution in your program will halt until this method is able to return a value. Examples of blocking operations include send(), recv(), connect(), and accept(). Recv() and accept() are the most obvious blocking calls you'll make on a socket. In both cases, these functions cannot return anything until another socket attempts to communicate with the it. Send() is technically a blocking call since it is possible that a message cannot be sent due to the receiver's buffer being full; however, send() will not typically block when called. That said, since it can block, we must treat it as a blocking call. Connect() will also block if a server is not ready to accept a new connection.

In our earlier projects we were only working with a single socket at a time. In this case, blocking operations don't typically cause major problems. However, blocking operations become a major problem as soon as a program needs to monitor several sockets in parallel. Consider the following situation: a machine is monitoring two sockets, A and B. The machine calls A.recv() and blocks. Even if B sends() data to the machine before A does, this data will not be read until after the machine receives a message from A, since the call to A.recv() has blocked. Even worse, assume that A is waiting for the machine to forward it information from B. Since our machine has blocked on A.recv(), it will never read the information from B and cannot forward it to A. Our code has now hung and cannot continue.

There are many different ways we can handle this situation. For this project we will use a *selector* to address it. A selector can be subscribed to multiple IO sources that involve blocking calls (like a socket). Prior to attempting to perform a blocking call on an IO source, your code can query the selector to determine which registered IO sources are ready to perform that operation (i.e. will not block if that operation is performed). This makes it easy to avoid calling a function that will cause your code to block.

Selectors in python can be created as shown in the below code segment. *Note: The selectors module is* ***not*** *available in Python 2.*

```
import selectors
my_selector = selectors.DefaultSelector()
```

When creating a new socket, your code will need to register the socket with your selector. This includes both server and client sockets. When registering a socket with a selector, you can specify whether you want to perform read operations, write operations, or both on the socket. You will typically only perform read operations on server sockets (accept is a read function). Client sockets may be read from and/or written to depending on the nature of your code. You can also associate an object with the socket during registration. This is a useful place to store information about the socket, such as a write buffer or information about the machine on the other side of the connection. The below code snippet shows how to register a socket with a selector.

```
#Assume that we have already created a socket and stored it in my_sock
my_sock.setblocking(False)

# The events field can be used to specify that we only care about reading, writing, or that we care
# about both reading and writing to this socket. Examples of only reading and only writing are
# shown below:
# events = selectors.EVENT_READ
# events = selectors.EVENT_WRITE
events = selectors.EVENT_READ | selectors.EVENT_WRITE

# We can associate an object with the socket by passing it to the selector in the optional data
# parameter. In this case we'll pass in a dictionary initialized with two values: an ID and a
# write_buffer string. This dictionary will be returned with the socket whenever we call select()
# and learn that this socket is ready to be read or written to.
my_sock_data = {}
my_sock_data['id'] = 73
my_sock_data['write_buffer'] = b''      # Initialize the write buffer to an empty byte array

my_selector.register(my_sock, events, my_sock_data)
```

You can call select() on your selector to get a list of IO sources that are ready to have a blocking operation performed. Each IO source will also be returned with a mask indicating what type of event is ready to be

performed (i.e. read event, write event, or both) and with the data associated with the IO source upon registering it with the selector. The below code snippet shows how to call select() and iterate over the returned IO sources.

```
1  # Note: select() is itself a blocking call. It will block until at least one IO object
2  #       has a blocking operation ready to be performed. You should include a timeout
3  #       value to prevent select() from permanently blocking
4  ready_IO = my_selector.select(timeout=1)                # Get a list of the ready IO devices
5
6  for io_device, event_mask in ready_IO:
7      my_sock = io_device.fileobj                         # Get a reference to the socket
8      my_sock_data = io_device.data                       # Get the data associated with the socket
9
10     # Check if a read operation can be performed
11     if event_mask & selectors.EVENT_READ:
12         msg = my_sock.recv(1024)                        # Read data from the socket
13         handle_msg(my_sock_data, msg)                   # Pass the received msg and my_sock_data to
14                                                         # a function for processing
15
16     # Check if a write operation can be performed
17     if event_mask & selectors.EVENT_WRITE:
18         if len(my_sock_data['write_buffer']) > 0:       # Check to make sure we have data to write
19             my_sock.send(my_sock_data['write_buffer'])  # Send the contents of write_buffer
20             my_sock_data['write_buffer'] = b''          # Clear the contents of write_buffer
```

You must never call send, recv, accept, or any other blocking call except for while iterating over the sockets returned by a call to select(). In the above example, it would be safe to call these functions within the for loop and within handle_msg, as this is called within the for loop.

This introduces some additional complexity. You will commonly want to send a message outside of this loop (i.e. a user types a message in the terminal and hits enter). Instead of sending the message immediately, we must instead store the information to be sent in a write buffer. We can then examine this write buffer after calling select() and determining that a write operation can be performed. This is where the data registered with our socket becomes extremely useful. We can store the write buffer inside of this object and then refer back to it when it is time to write. This is what is modeled in the above code snippet. If the handle_msg function determines that a response to the message is needed, this response can be appended to my_sock_data['write_buffer'], which will then be read on lines 18-20 in the above example.

Note: select() will almost always indicate that a write operation can be performed. As such, make sure you check that the write buffer contains information to be sent before attempting to send it, or else you will constantly send an empty message across the socket. You also want to remember to clear the write buffer after sending data, otherwise the same messages will be sent repeatedly.

Finally, when you are ready to close a socket you will also want to unregister it from the selector. You should also close the selector once you no longer need it. The below code snippet shows how to do this.

```
1  # Unregister a socket from the selector
2  my_selector.unregister(my_sock)
3
4  # Close the selector once it is no longer needed
5  my_selector.close()
```

4 Parsing the CRC Protocol

Clemson Relay Chat supports five unique message formats: 1) server registration, 2) client registration, 3) client chat messages, 4) status updates, and 5) client quit messages. All messages are encoded to byte arrays prior to transmission. Each message consists of three parts: 1) a byte encoding the message type, 2) a fixed-length header specified for each message type, and 3) a variable length payload encoding human-readable information (i.e. a server's name, the content of a chat message, etc).

Due to how messages are sent and received, it is possible to receive multiple messages from a single recv call to a socket. The CRC Protocol does not use message delimiters to separate individual messages. Instead, each message specifies its own length. As such, the MessageParser.parse_messages() function always checks to see if more than one message is contained in the provided bytes.

On joining the network, each machine (including both servers and users) selects a random 32-bit identifier. This will be used to determine who sent messages and who should receive messages. Each message includes the ID of the original source (i.e. the machine that first created this message). Messages may also contain a last-hop ID (i.e. the ID of the adjacent machine that forwarded the message now being parsed) and a destination ID (i.e. the ID of the machine who should receive this message).

The individual message formats are shown below. The first field of each message encodes the message type as a single byte. The specific value expected for each message type is given for each message type. The size of each field is also reported (i.e. byte, half, int, bool, etc).

4.1 Server Registration Messages

- Message Type (byte = 0x00)
- Source ID (int) - the unique identifier associated with the machine that originated this message
- Last Hop ID (int) - the unique identifier associated with the machine that last forwarded this message. **Note: there is one exception to this rule. When a server first joins the network, it sets the Last Hop ID to 0. This allows the first server that receives the message to know that this is a new adjacent server.**
- Server Name Length (byte) - the length of a variable length field storing the new server's name
- Server Info Length (half) - the length of a variable length field with information about the new server
- Server Name String (variable length string, ASCII encoding) - the new server's name
- Server Info String (variable length string, ASCII encoding) - information about the new server

4.2 Client Registration Messages

- Message Type (byte = 0x80)
- Source ID (int) - the unique identifier associated with the machine that originated this message
- Last Hop ID (int) - the unique identifier associated with the machine that last forwarded this message. **Note: there is one exception to this rule. When a client first joins the network, it sets the Last Hop ID to 0. This allows the first server that receives the message to know that this is a new adjacent client.**
- Client Name Length (byte) - the length of a variable length field storing the new user's name
- Client Info Length (half) - the length of a variable length field with information about the new user
- Client Name String (variable length string, ASCII encoding) - the new user's name
- Client Info String (variable length string, ASCII encoding) - information about the new user

4.3 Client Chat Messages

- Message Type (byte = 0x81)
- Source ID (int) - the unique identifier associated with the machine that originated this message
- Destination ID (int) - the unique identifier associated with the machine targeted by this message
- Message Length (int) - the length of a variable length field with the message
- Message String (variable length string, ASCII encoding) - the content of the message

4.4 Status Messages

- Message Type (byte = 0x81)
- Source ID (int) - the unique identifier associated with the machine that originated this message
- Destination ID (int) - the unique identifier associated with the machine targeted by this message
- Status ID (byte)
- Message Length (int) - the length of a variable length field with the message
- Message String (variable length string, ASCII encoding) - the content of the message

4.5 Client Quit Messages

- Message Type (byte = 0x82)
- Source ID (int) - the unique identifier associated with the machine that originated this message
- Message Length (int) - the length of a variable length field with the quit message
- Message String (variable length string, ASCII encoding) - the content of the quit message

5 Implementing a Distributed Client-Server Network

We saw an example of something **similar to** our distributed network when we considered self-learning switches. Please note, self-learning switches are not distributed networks in the same sense as this project is, however the update process by which they learn about the network has parallels with the type of distributed network we'll be implementing.

One of the chief challenge in implementing a distributed network is maintaining state across all machines on the network. When a new server comes online, or when a new user connects to the network, we need to immediately communicate this to all other machines. This differs from self-learning switches, where each switch learns about the state of the whole network over time as messages move through it.

A second challenge in implementing a distributed server network is ensuring that each machine knows how to efficiently communicate with every other machine (i.e. if Server 37 wants to send a message to Client 123, what adjacent machine is this message forwarded to). For this project, we will accomplish this by structuring the network as a spanning tree, similar to self-learning switches. Thus, when a server receives a message from a given machine over a specific link, the server learns that in order to communicate with that given machine it should send messages to that specific link (i.e. if the message from Client 123 is given to Server 37 by Server 42, Server 37 now knows that it should send messages to Server 42 when it wants to talk to Client 123).

To address these challenges, your Chat Server will need to do the following:

Keep track of the state of the network: This includes storing information about every other machine on the network and which of these machines are adjacent to the server (an adjacent machine is one who can be communicated with directly via a socket). Your server will also need to store the information required to communicate with each machine on the network. Since our network is structured as a spanning tree, your server needs to simply store a "first link" value for each machine. The first link is the adjacent machine that is the first node on the path to a specific machine through the spanning tree.

Communicate updates to the state of the network to other machines: When your server receives a state update message (i.e. a server had connected to the network, a user has quit, etc) it needs to forward this to all adjacent nodes unaware of this information. Given that our network is structured as a spanning tree, this means sending the message to all adjacent nodes except for the node that we received this message from. When a machine first connects to the network, it knows nothing about the current state of the network. In the event that a server receives a message directly from the new machine, the server must also communicate all existing state information to the new machine. It is important that remote servers NOT do this, as this would clog the network with unnecessary messages. *Note: clients only keep track of what other clients are on the network, which means they should not receive server state updates.*

6 Testing Your Project

Run the included CRCTestManager.py to test your code. The end of this file includes a dictionary containing references to different test cases, which are specified in the TestCases/ folder. I recommend commenting out all but the tests you are currently working on to save time.

As this project involves actual sockets, it takes a real amount of time for messages to travel between hosts (even without actually crossing the Internet). As such, the test cases are configured to introduce a small amount of delay in between each action on the network. In my testing, the default delays are long enough to allow all messages to reach their destination. However, if your computer is slower, or if you run all of the tests simultaneously, you may have tests fail because a message did not have time to reach its destination, not because your code is wrong. As such, I recommend running no more than a few tests at a time. This is especially true for the test cases involving eleven servers. If you need to increase the amount of time between actions, you may edit the test cases to increase the time specified after each WAIT command.

The test cases on Gradescope are the same as the test cases you run locally, except that the WAIT times are increased substantially to ensure that all messages have time to arrive at their destination prior to a test completing. This means that it will take Gradescope a long time to grade your submissions. As such, I recommend testing locally and only submitting to Gradescope when you have made major progress.

7 Submitting Your Project

Please submit ChatServer.py to the assignment page on Gradescope. Do not include any other files in your submission.