

I. Documentation

Environment and Overview

Program 4 involved implementing a “UdpRelay” class where each instantiation of the class is part of a single node (machine) on a network. Each UdpRelay receives messages sent via UDP broadcast locally, then sends the message to any UdpRelay nodes connected to it via TCP connection, and the UdpRelay nodes receiving that TCP message in-turn broadcast the message via UDP to other machines on their node.

The program was designed and tested for **Linux** and built using Linux system commands (socket, send, recv, close) in <unistd.h>, as well as the pthreads library. It is C++98 compliant, and therefore C++11 and C++14 compliant also.

Technique to Implement UdpRelay Class

In order to handle several operations that need to be running concurrently, we use the **pthread**s library to spin up multiple threads for **task parallelization**. UdpRelay also utilizes the Socket and UdpMulticast classes written by Dr. Fukuda in order to establish TCP sockets between UdpRelay nodes (Socket), and to send and receive local UDP broadcasts (UdpMulticast).

Program Structure + Implementation

As mentioned, we use several threads to achieve task parallelization. Our implementation details for these threads are as follows:

- **Command Thread:** A single thread spun up in UdpRelay’s constructor, it persists for the duration of the UdpRelay object and takes user commands of:
add remoteGroupIP:portNumber | Connects via TCP to a remote UdpRelay node by calling addRemoteIP()
delete remoteGroupIP | Deletes a TCP connection to remote group by calling terminateRemoteCxn()
show | Shows current TCP connections to UdpRelay nodes by group name (uw1-320-XX) with socket number being used for that connection
help | Shows all user commands available with a brief description
quit | Terminates all TCP connections, cancels or joins all threads and terminates the program.
- **relayIn Thread:** A single thread spun up in the constructor which persists for the duration of the UdpRelay object. It accepts local UDP broadcast messages and sends via TCP those messages to all connected UdpRelay nodes by calling tcpMultiCastToRemoteGroups().
- **Accept Thread:** A single thread spun up in the UdpRelay constructor, it persists the duration of the UdpRelay object. It accepts requests to connect to another UdpRelay node via TCP, and spins up a relayOut thread.

- **relayOut Thread:** Persists the length of a single TCP connection between two UdpRelay nodes. There are as many of these as there are active TCP connections to that node. Receives TCP message from another UdpRelay node, then broadcasts the message via UDP using `sendLocalMessage()`.

The pthreads library requires that thread function called in `pthread_create()` are either static class methods, or stand-alone functions. Due to this requirement, all thread functions are static methods of UdpRelay class which receive the current UdpRelay object as parameter. Most of the thread functions also call an actual class method, to simplify operations on the UdpRelay object.

Our program implementation that may differ from others uses two **maps** to hold connections:

- **map<string,int> tcpCxnns:** Maps all TCP connections to UdpRelay nodes with the remote IP (string) or remote name (string) mapped to the socket number (int). This is done so messages can be sent to each remote group through the appropriate socket.
- **map<int,thread_t> outThreads:** Maps all active sockets to other UdpRelay nodes to the relayOut thread ID that is handling that socket. This allows a call of `pthread_join()` to terminate each relayOut thread after it finishes execution.

Additionally, we use a **queue<thread_t>** to hold the thread IDs of relayOut threads that have terminated. When this queue reaches a size of 5, or the program is terminated, a function calls `pthread_join()` on all the IDs, closing them all out.

We also use a **semaphore** `sem_wait()` call(<semaphore.h>) to halt the initial program thread in the constructor after the command, relayIn and accept threads are spun up. When a user types “quit” into the command thread all TCP connections are closed using `terminateAllTcpConnections()`, a `sem_post()` call is issued returning control to the initial thread and all created threads are cancelled or joined before the program exits.

Error checking is done via several means in the program:

1. There is error checking around sockets being opened (both TCP and UDP) to make sure the socket is established correctly. If the socket fails, the user is informed and the program awaits the next command via user thread.
2. In the command thread, there is a “help” command to show what commands are available and what parameters they take. If the user provides an incorrect command, they are informed and the program awaits the next command.
3. If the user passes remote IP numbers or port numbers that are not correct to either the “add” or “delete” connection commands, the user is informed, and the program awaits the next command.
4. Per specifications, if an incoming TCP connection already exists on a UdpRelay node, if the user attempts to add another TCP connection to the same node, the first connection is closed and deleted. This is done to prevent a race condition that would produce duplicate messages.

Dr. Fukuda’s **UdpMulticast.cpp** is **not working as-intended** also, because it uses `INADDR_ANY` to set a destination address for a server socket. This allows a BroadcastClient to send a UDP message to the UdpRelay only on its remote group IP, or randomly to multiple or all UdpRelay objects on different remote group IPs. This creates duplicate messages because multiple UdpRelay objects are receiving the same UDP message simultaneously.

II. Execution Output: Here is the correct **7-node, 3-remote group output** specified in the Program 4 grading rubric:

```
Terminal
dph267@uw1-320-05:~/CSS503/Program4$ ./UdpRelay 237.255.255.255:24879
UdpRelay: booted up at 237.255.255.255:24879
% Registered: uw1-320-09
UdpRelay: received 18 bytes from uw1-320-09 = Hello!
UdpRelay: broadcast buf[22] to 237.255.255.255:24879
```

237 UdpRelay

```
Terminal
dph267@uw1-320-08:~/CSS503/Program4/java$ java BroadcastServer 238.255.255.255:24879
uw1-320-05.uwb.edu: Hello!
uw1-320-05.uwb.edu: Ola!
```

238 BroadcastServer

```
Terminal
dph267@uw1-320-06:~/CSS503/Program4/java$ java BroadcastServer 237.255.255.255:24879
uw1-320-05.uwb.edu: Hello!
```

237 BroadcastServer

```
Terminal
dph267@uw1-320-09:~/CSS503/Program4$ ./UdpRelay 239.255.255.255:24879
UdpRelay: booted up at 239.255.255.255:24879
% add uw1-320-07:24879
Registered: uw1-320-07
Added: uw1-320-07:6
% add uw1-320-05:24879
Registered: uw1-320-05
Added: uw1-320-05:7
% UdpRelay: relay Hello! to remoteGroup[uw1-320-05]
UdpRelay: relay Hello! to remoteGroup[uw1-320-07]
delete uw1-320-05
UdpRelay: deleted uw1-320-05
% UdpRelay: relay Ola! to remoteGroup[uw1-320-07]
delete uw1-320-07
UdpRelay: deleted uw1-320-07
%
```

239 UdpRelay

```
Terminal
dph267@uw1-320-07:~/CSS503/Program4$ ./UdpRelay 238.255.255.255:24879
UdpRelay: booted up at 238.255.255.255:24879
% Registered: uw1-320-09
UdpRelay: received 18 bytes from uw1-320-09 = Hello!
UdpRelay: broadcast buf[22] to 238.255.255.255:24879
UdpRelay: received 16 bytes from uw1-320-09 = Ola!
UdpRelay: broadcast buf[20] to 238.255.255.255:24879
```

238 UdpRelay

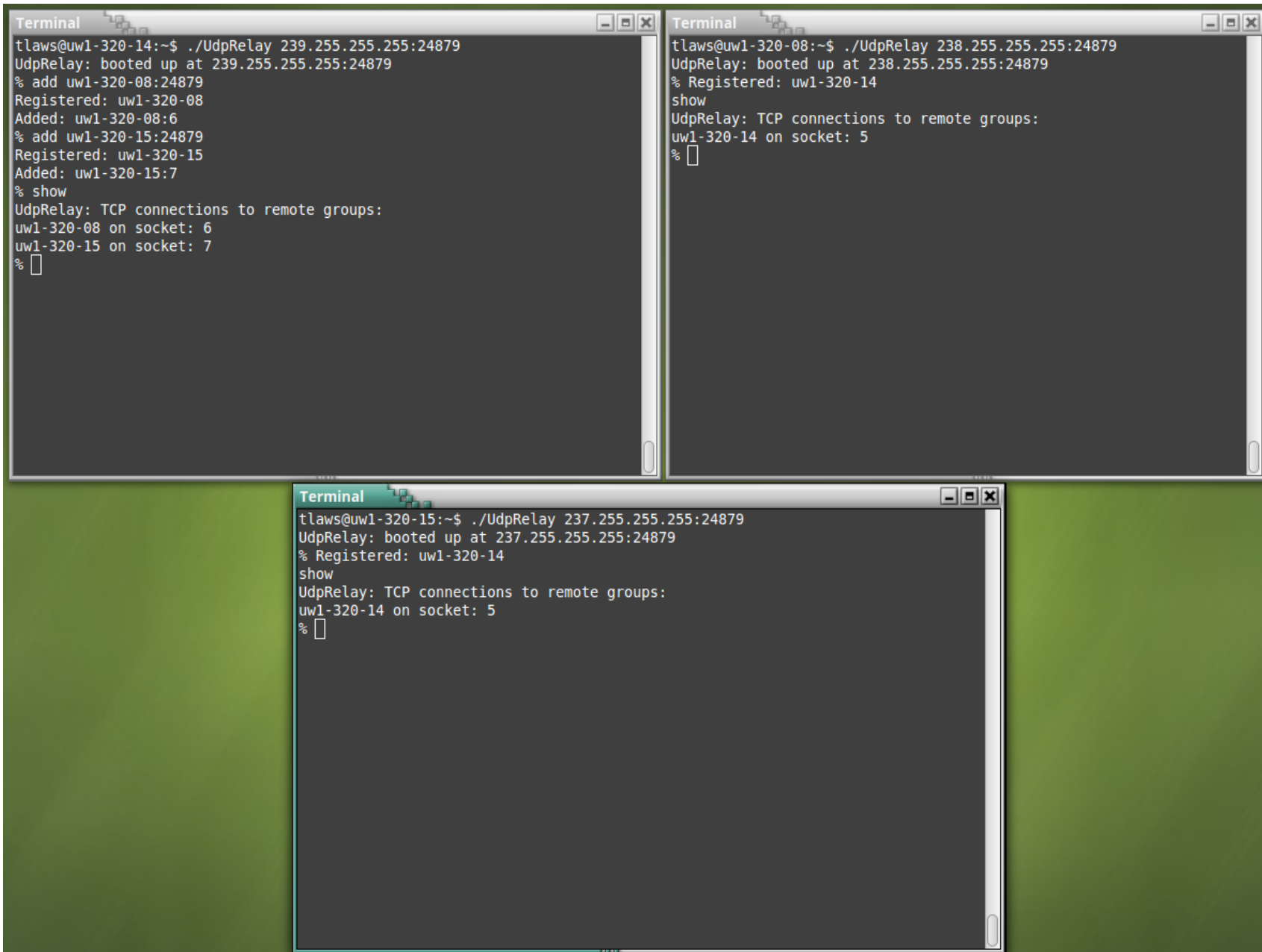
```
Terminal
dph267@uw1-320-10:~/CSS503/Program4/java$ java BroadcastServer 239.255.255.255:24879
uw1-320-05.uwb.edu: Hello!
uw1-320-05.uwb.edu: Ola!
uw1-320-05.uwb.edu: Adieu!
```

239 BroadcastServer

```
Terminal
dph267@uw1-320-05:~/CSS503/Program4$ cd java
dph267@uw1-320-05:~/CSS503/Program4/java$ java BroadcastClient 239.255.255.255:24879 Hello!
dph267@uw1-320-05:~/CSS503/Program4/java$ java BroadcastClient 239.255.255.255:24879 Ola!
dph267@uw1-320-05:~/CSS503/Program4/java$ java BroadcastClient 239.255.255.255:24879 Adieu!
```

239 BroadcastClient

Execution Output – show + add commands: Two remote groups are connected using the “add” command to the UdpRelay node at 239 (top left). When “show” is called on all three nodes, it shows each connection properly.



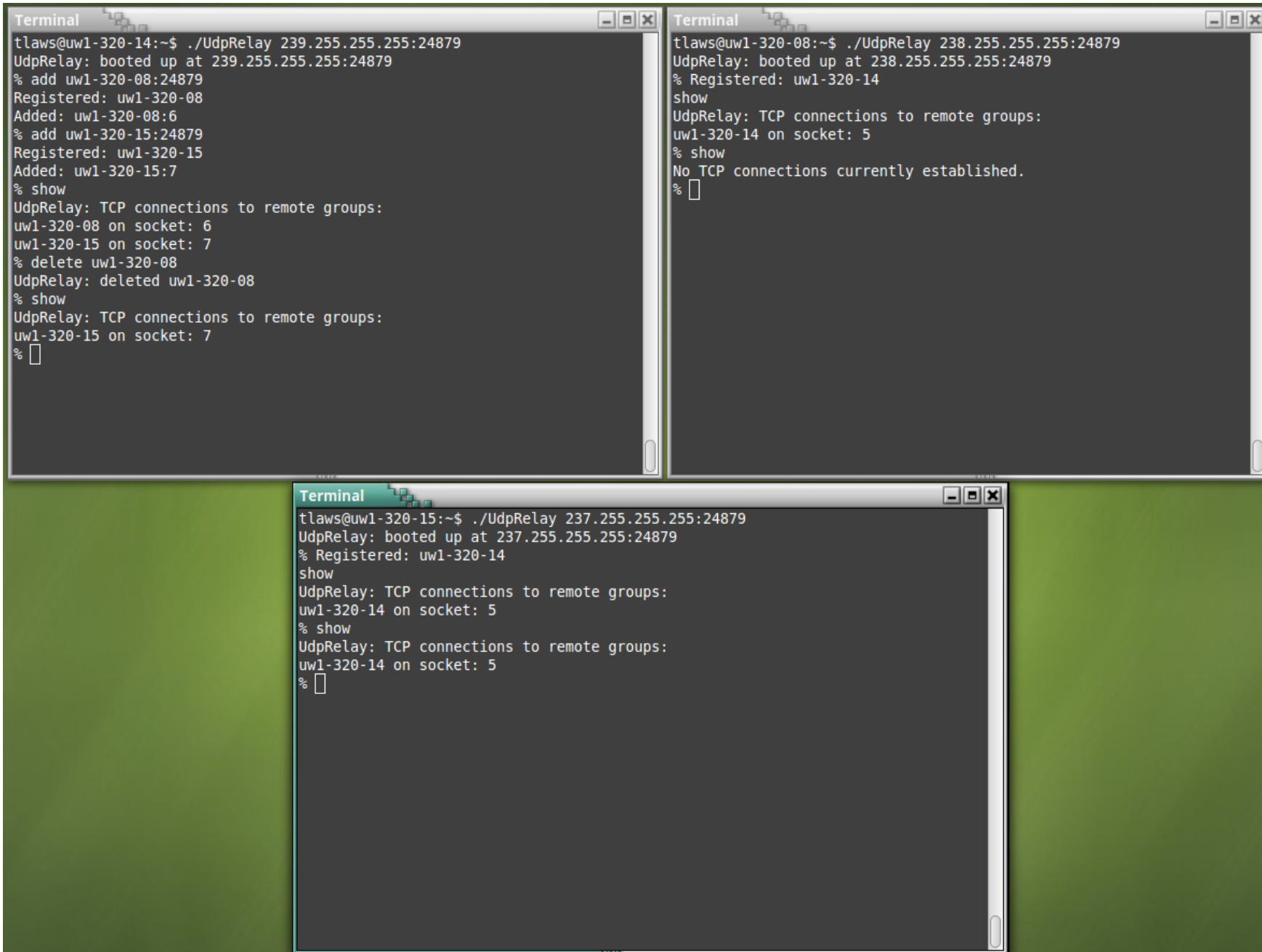
The image displays three terminal windows arranged in a 2x2 grid, with the bottom-right position empty. Each window shows the execution of the UdpRelay program on a different host. The top-left window (uw1-320-14) shows the addition of two remote groups (uw1-320-08 and uw1-320-15) and the resulting TCP connections. The top-right window (uw1-320-08) shows the registration of the local group (uw1-320-14) and the resulting TCP connection. The bottom-left window (uw1-320-15) shows the registration of the local group (uw1-320-14) and the resulting TCP connection. All windows show the 'show' command output, which lists the TCP connections to remote groups and the local group on a specific socket.

```
Terminal
tlaws@uw1-320-14:~$ ./UdpRelay 239.255.255.255:24879
UdpRelay: booted up at 239.255.255.255:24879
% add uw1-320-08:24879
Registered: uw1-320-08
Added: uw1-320-08:6
% add uw1-320-15:24879
Registered: uw1-320-15
Added: uw1-320-15:7
% show
UdpRelay: TCP connections to remote groups:
uw1-320-08 on socket: 6
uw1-320-15 on socket: 7
%

Terminal
tlaws@uw1-320-08:~$ ./UdpRelay 238.255.255.255:24879
UdpRelay: booted up at 238.255.255.255:24879
% Registered: uw1-320-14
show
UdpRelay: TCP connections to remote groups:
uw1-320-14 on socket: 5
%

Terminal
tlaws@uw1-320-15:~$ ./UdpRelay 237.255.255.255:24879
UdpRelay: booted up at 237.255.255.255:24879
% Registered: uw1-320-14
show
UdpRelay: TCP connections to remote groups:
uw1-320-14 on socket: 5
%
```

Execution Output – show + delete commands: Following the adding of both nodes on the previous page, one of the nodes is deleted from the 239 UdpRelay (top left). “show” is then called on all 3 UdpRelay nodes, and output is correct showing that a connection still exists between 239 and 237, but the connection from 239 to 238 has been properly terminated after the “delete” call.



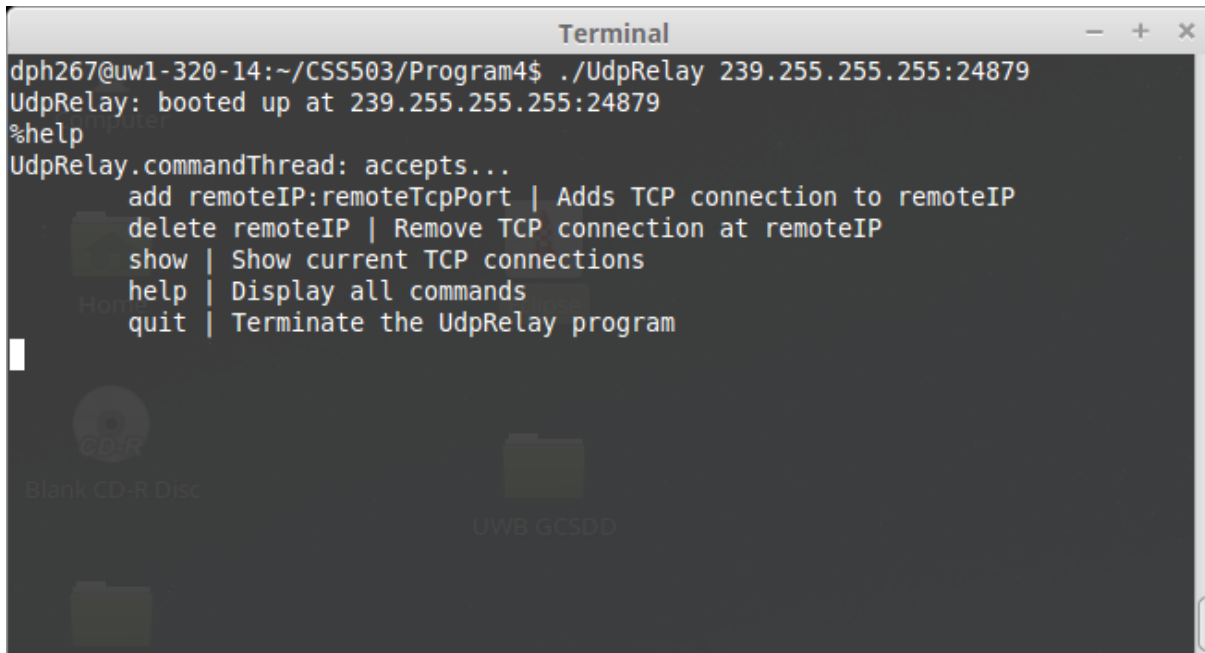
The image displays three terminal windows, each showing the execution of the UdpRelay program on a different node. The top-left window is on node uw1-320-14, the top-right on uw1-320-08, and the bottom window on uw1-320-15. Each window shows the process of adding nodes, deleting a node, and then displaying the current state of TCP connections.

```
Terminal
tllaws@uw1-320-14:~$ ./UdpRelay 239.255.255.255:24879
UdpRelay: booted up at 239.255.255.255:24879
% add uw1-320-08:24879
Registered: uw1-320-08
Added: uw1-320-08:6
% add uw1-320-15:24879
Registered: uw1-320-15
Added: uw1-320-15:7
% show
UdpRelay: TCP connections to remote groups:
uw1-320-08 on socket: 6
uw1-320-15 on socket: 7
% delete uw1-320-08
UdpRelay: deleted uw1-320-08
% show
UdpRelay: TCP connections to remote groups:
uw1-320-15 on socket: 7
%

Terminal
tllaws@uw1-320-08:~$ ./UdpRelay 238.255.255.255:24879
UdpRelay: booted up at 238.255.255.255:24879
% Registered: uw1-320-14
show
UdpRelay: TCP connections to remote groups:
uw1-320-14 on socket: 5
% show
No TCP connections currently established.
%

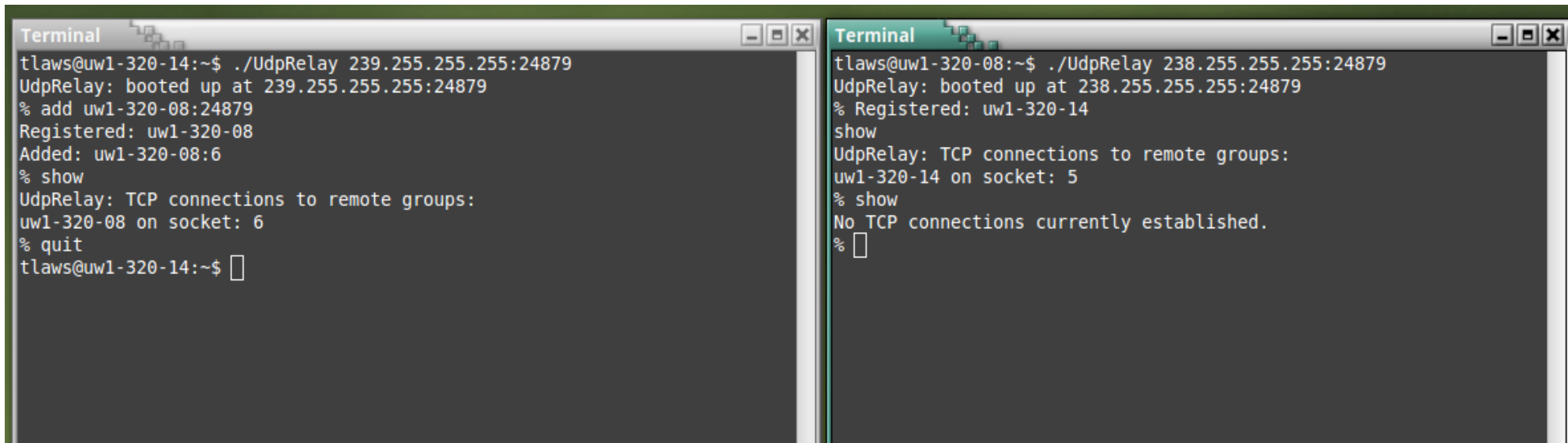
Terminal
tllaws@uw1-320-15:~$ ./UdpRelay 237.255.255.255:24879
UdpRelay: booted up at 237.255.255.255:24879
% Registered: uw1-320-14
show
UdpRelay: TCP connections to remote groups:
uw1-320-14 on socket: 5
% show
UdpRelay: TCP connections to remote groups:
uw1-320-14 on socket: 5
%
```

Execution Output – help command: Here is a screenshot of the “help” command working correctly:



```
Terminal
dph267@uw1-320-14:~/CS5503/Program4$ ./UdpRelay 239.255.255.255:24879
UdpRelay: booted up at 239.255.255.255:24879
%help
UdpRelay.commandThread: accepts...
  add remoteIP:remoteTcpPort | Adds TCP connection to remoteIP
  delete remoteIP | Remove TCP connection at remoteIP
  show | Show current TCP connections
  help | Display all commands
  quit | Terminate the UdpRelay program
```

Execution Output – quit command: Here is a screenshot of the quit command. Notice a TCP connection is established between the two remote groups, and “show” is called on both UdpRelay objects to show that the connection exists. Then, “quit” is entered into the UdpRelay node at left, closing connections and terminating the program. When “show” is called again on the still-running UdpRelay, it lists all connections properly terminated:



```
Terminal
tlaws@uw1-320-14:~$ ./UdpRelay 239.255.255.255:24879
UdpRelay: booted up at 239.255.255.255:24879
% add uw1-320-08:24879
Registered: uw1-320-08
Added: uw1-320-08:6
% show
UdpRelay: TCP connections to remote groups:
uw1-320-08 on socket: 6
% quit
tlaws@uw1-320-14:~$

Terminal
tlaws@uw1-320-08:~$ ./UdpRelay 238.255.255.255:24879
UdpRelay: booted up at 238.255.255.255:24879
% Registered: uw1-320-14
show
UdpRelay: TCP connections to remote groups:
uw1-320-14 on socket: 5
% show
No TCP connections currently established.
%
```


III. Discussion

Limitations and Potential Improvements/Extensions

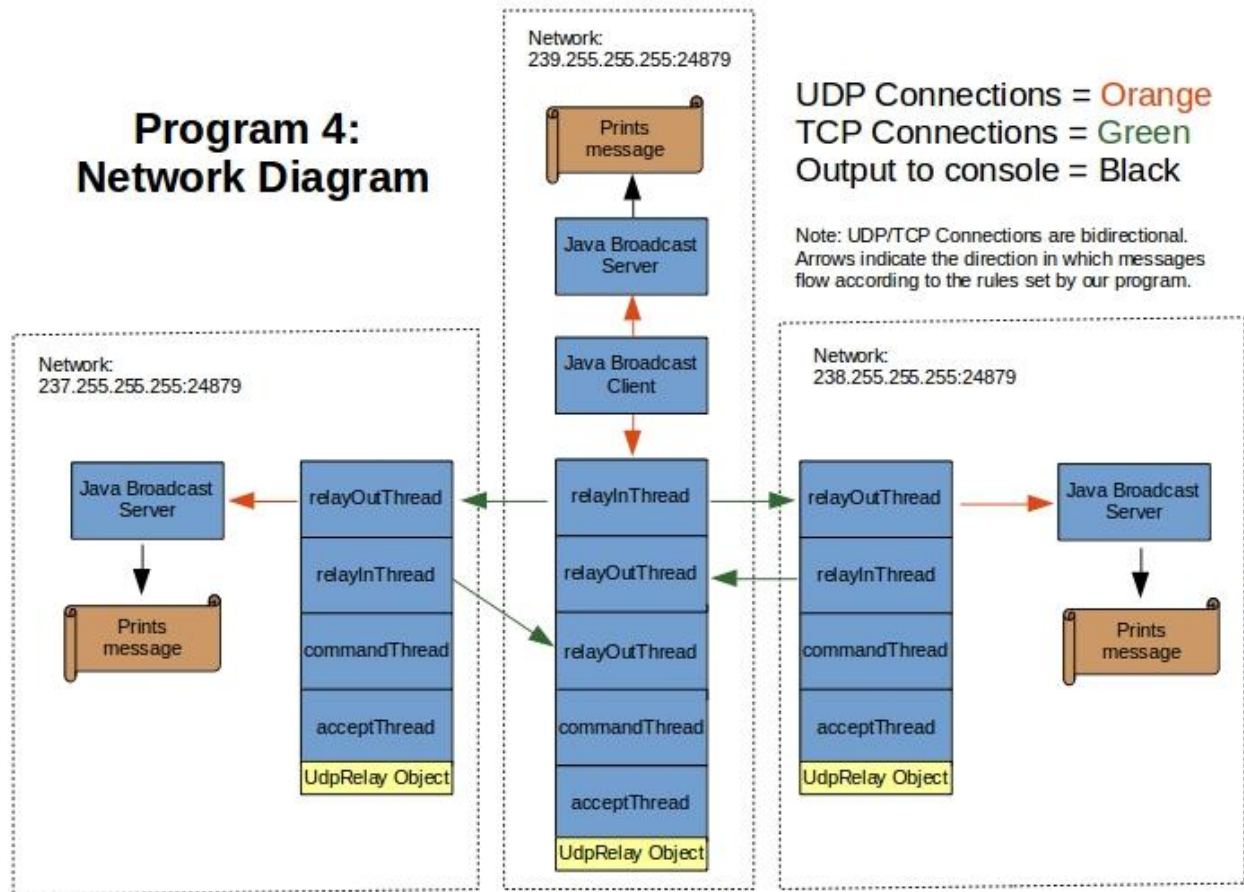
- The program specification states that if an incoming TCP connection exists from a specific UdpRelay, if a connection request comes in from the same UdpRelay, the first connection is deleted. However, the program, by design, creates potential race conditions that can produce duplicate messages despite the IP addresses being contained in the message packet. Consider a situation where node 1 is connected to nodes 2 and 3. Node 2 is also connected to node 3. Node 1 performs a TCP broadcast of a message to 2 and 3, then node 2 also sends the message to node 3. Node 3 would receive the same message twice.
- There is effectively no documentation for the Socket and UdpMulticast classes, making them somewhat difficult to use. While we know what each method is supposed to accomplish, the lack of commenting about how the Socket and UdpMulticast objects are used makes it easy to create binding errors, or let locally-declared objects fall out of scope, closing their sockets.
- There are numerous memory leaks created in Socket + UdpMulticast classes, though some of it is cleaned up with destructors.
- Due to the need to keep the command thread, relayIn thread and accept thread active for the duration of the UdpRelay's existence, their thread functions loop infinitely and thus need to be cancelled. Since they are cancelled, it's possible for a thread to be in the midst of an operation when it's terminated. However, due to the nature of the program only accepting user inputs, we deemed this acceptable, since a user would have to be typing simultaneously on two machines in order to create this condition. The relayOut threads are joined properly after terminating themselves when a socket is closed.
- As per the specification, the program is hard-coded to take 5-digit port numbers and only IPv4 addresses of 12 total digits. It will accept no IPv6 IPs, or IP addresses with less than 12 digits (ex: 10.0.0.12) as the remote group number when booting up the UdpRelay.

Potential improvements/extensions:

- Socket and UdpMulticast classes could be improved with more detailed documentation and more comprehensive destructors to eliminate memory leaks.
- Though outside the scope of work and requiring a moderate amount of new code, the program could be made to recognize IPv6 addresses or IPv4 addresses of less than 12 digits. This would create a UdpRelay class that is far more flexible, and more robust to newer technology, than the currently specified version.
- Eliminating the design flaw that allows for duplicate message through race conditions would require large overhead. One solution would be if the UdpRelays were capable of communicating with each other to automatically ensure no cyclical network connections could be created, then duplicate packets could be prevented entirely.
- Socket and UdpMulticast classes use a variety of deprecated or obsolete functions (bzero() & itoa(), for example) which could be replaced with more highly optimized, safer and standardized C++ functions that accomplish the same task (such as memset() & to_string()).
- A Boost library testing framework combined with a script to run specific code on 7 UW machines could make testing of the code much, much simpler than having to log into 7 UW lab computers separately, and execute commands on each.

Discussion of Step 3 Execution Output: 7-Node, 3-Remote Group

Here is a diagram depicting the activity of each thread for the sample output detailed in the Program 4 specification, which involves 3 UdpRelay nodes acting as 3 separate remote groups, with a single Java BroadcastClient. The process originates with the BroadcastClient on 239.255.255.255 (center node) sending a UDP message to the UdpRelay on 239.255.255.255.



Each UdpRelay object operates on a separate “node” which separates them from UDP broadcasts from a java BroadcastClient that occur on a single node. The UdpRelay object in the 239 node creates the correct output by pushing the UDP message received from the BroadcastClient on the 239.255.255.255 network across to the UdpRelay objects on the 238.255.255.255 and 237.255.255.255 nodes, which in turn send out their own UDP broadcast that is picked up by the local java BroadcastServers. Both 237 and 238 UdpRelays send the message back to the 239 UdpRelay via TCP, but as it’s a duplicate packet, 239 discards the message.

Thus, all 3 UdpRelay objects and all 3 java BroadcastServers appear to be operating on the same network group: 239.255.255.255, where the message originates. This achieves the goal of the program as a UDP broadcast simulation across multiple machines.

In the execution output, all the messages have the correct length in bytes also (18 & 16).

After the 239 UdpRelay deletes the connection to the 237 UdpRelay group, only the 238 UdpRelay node receives the “Ola!” message. After all connections are deleted, only the local 239 BroadcastServer receives the “Adieu!” message.

When executing Dr. Fukuda’s actual code from his version of UdpRelay.cpp and .h, it occasionally runs into segmentation faults and falls prey to, what I can only surmise, are UW lab network inconsistencies. Our program does not have these issues, though rarely, the first message sent out by a java BroadcastClient is not received, but all subsequent messages are. This, too, could be the result of UW network issues, or due to the unreliability inherent to UDP packets, which arrive in no particular order.