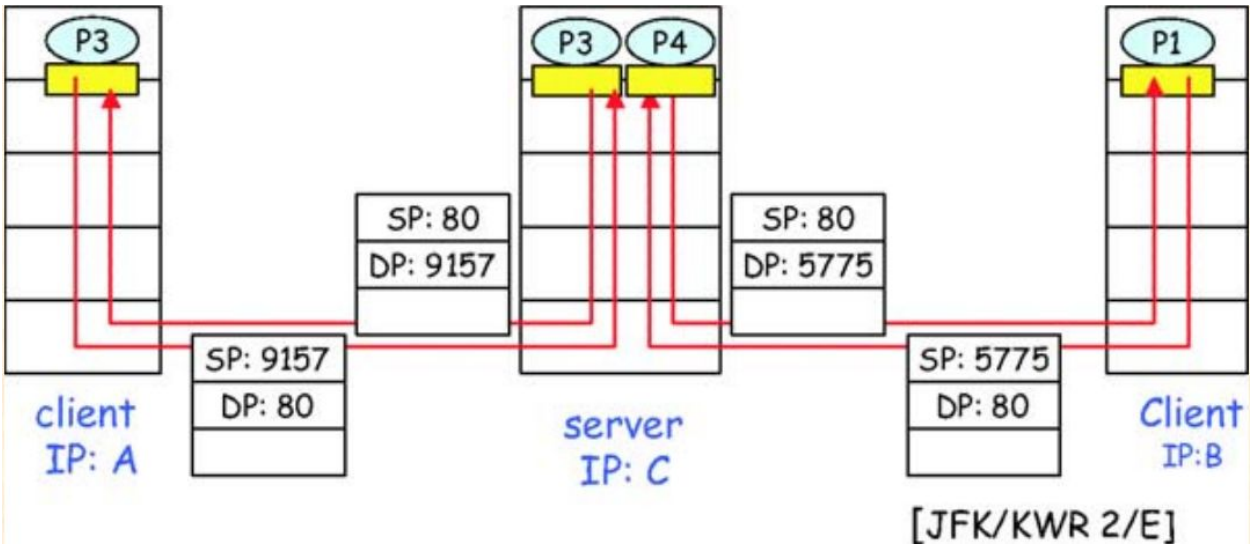
CMPUT 391 Assignment 3
Design Document

Software Defined Network (SDN) using FIFOs and Sockets



James Hryniw 1431912
Date: 2018-11-21

Objectives

The main objectives of the assignment were to introduce the concepts of inter-process communication with TCP sockets. The challenges of I/O multiplexing, FIFO communication between switches and message serialization were solved in a previous implementation of a linear software defined network (Assignment 2).

Design Overview

Overall, the design behind my implementation of the controller and switch in *a3sdn* was driven primarily by the concept of encapsulation. I wanted the controller and switch (nodes) only to handle accepting, processing and relaying messages whereas other classes could handle the specifics of managing communication protocols, message serialization/deserialization and I/O multiplexing. This decision was extremely useful, few code changes were required between this version and an all-FIFO version of the application.

Beyond changing switch-controller communication to a TCP protocol, *a3sdn* also adds several new features to the application. In the paragraphs below, I will highlight and explain the main design decisions behind implementing the following features:

- TCP socket communication between the controller and switches
- Simulating traffic delays in switches
- Gracefully handling socket connection loss

Socket communication is abstracted by new Port subclasses FifoPort and SocketPort. By default, a SocketPort tries to create a client TCP socket. Switches use the server IP and port to startup create a connection. If any error occurs during the connection process, the program exits cleanly. Therefore, switches must be started **after** the controller.

In the controller, a listening socket is created when the controller starts, and any accepted connections create a new SocketPort for a particular switch. One design issue is that when a connection is first established, we don't know which switch connected because it hasn't yet sent an OPEN packet. The controller handles this by first storing the connection in a list of "unknown ports". When an OPEN packet is received, only then is a switch assigned to a particular port based on it's switch id.

To simulate traffic delays, the switch will stop reading or processing traffic for *time* milliseconds if it reads a line in the traffic file of the form *swi delay time* where *i* is the switch id. The program implements the delay using an internal state machine and the system clock. A switch has three mode; PRE_ACK (before ACK is received), DELAY and NORMAL. In NORMAL mode, the switch processes traffic and commands as usual. Once a switch reads a delay traffic line, it will compute the system time at which to exit the DELAY state and then switch to DELAY mode. In DELAY mode, the program checks

if the delay time has expired using the system clock (in which case it goes back to NORMAL mode) and reads *list* and *exit* commands **but does not process traffic**.

One nice advantage of using TCP sockets is that it is possible to tell when socket has lost connection to the other end. The *recv(...)* function returns an integer in three possible ranges -- which triggers the following responses when attempting to read a packet.

Recv Return	**Cause**	**Response**
> 0	Successfully read a packet	Decode and return the packet
= 0	The peer socket closed gracefully.	Return a CLOSE packet
< 0	An error occurred while reading the socket.	If errno is **EAGAIN** or **EINT,** return a null pointer, otherwise print the error and return a CLOSE packet

When a network node (controller or switch) receives a nullptr packet, they will not process a packet but keep the connection alive. This is because EAGAIN and EINT are not fatal errors. Instead, they indicate that the socket is not yet ready to read from (in non-blocking socket I/O) or a signal interrupted the system call, respectively.

Conversely, when a node receives a CLOSE packet this explicitly tells the node that it is time to close the connection. In the controller, we print that we lost connection to the switch and close the port but the process continues to run as normal.

Note: An extra output line will be printed when a switch / controller gets a CLOSE packet.

Project Status

Both the controller and switches are functionally complete. In a perfect system, the switches would try to reconnect to the server if the initial connection didn't work or they lose connection to the server, but this wasn't part of the specification.

In order to meet the assignment specification, no major issues were encountered.

Testing and Results

The robust object-oriented design enabled an incremental implementation of the specification. Testing focused primarily on the new parts of the application rather than the features already implemented.

- Bi-directional socket communication

- The CLOSE packet (closing nodes gracefully)
- Handling delays in the traffic file
- Parsing server address and port from the input

Since the project started with all the basic functionality in place, each feature was developed then tested incrementally, in the same order as above. To make this possible, the server address and port were hardcoded into the program until the command line functionality was added. Otherwise, the implementation each feature was relatively dependency-free.

In the early stages, testing was conducted using the same data files as used in Assignment 2. Since the project was based upon a working version, it was a simple matter of matching the results to those previously obtained -- with the underlying communication layer changed.

Only near the end, when delay functionality was added, that I relied on the data files with delays provided on eClass. For example, using the new data file t2.dat (*sw1 delay 5000* line added):

Terminal 1:
~/.../linear-sdn> a3sdn cont 2 9912

Received (src= sw1, dest= cont) [OPEN]
(port0= cont, port1= null, port2= sw2, port3= 100-110)

... (more Received/Transmitted logs)

cmd[controller]: list

Switch information:
[sw1] port1= -1, port2= 2, port3= 100-110
[sw2] port1= 1, port2= -1, port3= 200-210

Packet Stats:
Received: OPEN:2, QUERY:4
Transmitted: ACK:2, ADD:4

Terminal 2:
~/.../linear-sdn> a3sdn sw1 t2.dat null sw2 100-110 127.0.0.1 9912

Transmitted (src= sw1, dest= cont) [OPEN]
(port0= cont, port1= null, port2= sw2, port3= 100-110)
Received (src= cont, dest= sw1) [ACK]

** Entering a delay period of 5000 msec

... (more Received/Transmitted logs)

cmd[sw1]: list

Flow table:
[0] (src= 0-1000, destIP= 100-110, action= FORWARD:3, pri= 4, pktCount= 5)
[1] (src= 0-1000, destIP= 200-210, action= FORWARD:2, pri= 4, pktCount= 1)
[2] (src= 0-1000, destIP= 300-300, action= DROP:0, pri= 4, pktCount= 1)

Packet Stats:
Received: ADMIT:5, ACK:1, ADDRULE:2, RELAYIN:2
Transmitted: OPEN:1, QUERY:2, RELAYOUT:1

Terminal 3:
~/.../linear-sdn> a3sdn sw2 t2.dat sw1 null 200-210 127.0.0.1 9912

Transmitted (src= sw2, dest= cont) [OPEN]
    (port0= cont, port1= sw1, port2= null, port3= 200-210)

... (more Received/Transmitted logs)

cmd[sw2]: list

Flow table:
[0] (src= 0-1000, destIP= 200-210, action= FORWARD:3, pri= 4, pktCount= 1)
[1] (src= 0-1000, destIP= 300-300, action= DROP:0, pri= 4, pktCount= 1)
[2] (src= 0-1000, destIP= 100-110, action= FORWARD:1, pri= 4, pktCount= 2)

Packet Stats:
Received: ADMIT:3, ACK:1, ADDRULE:2, RELAYIN:1
Transmitted: OPEN:1, QUERY:2, RELAYOUT:2

As a final test, this series of commands was repeated to a non-local IP address to prove that the system does in fact use TCP sockets. The results were the same.

Acknowledgements

The programs in this assignment were developed without aid from anyone. Some minor pieces of code were modified from internet sources. Where applicable, they are cited in the code using comments.