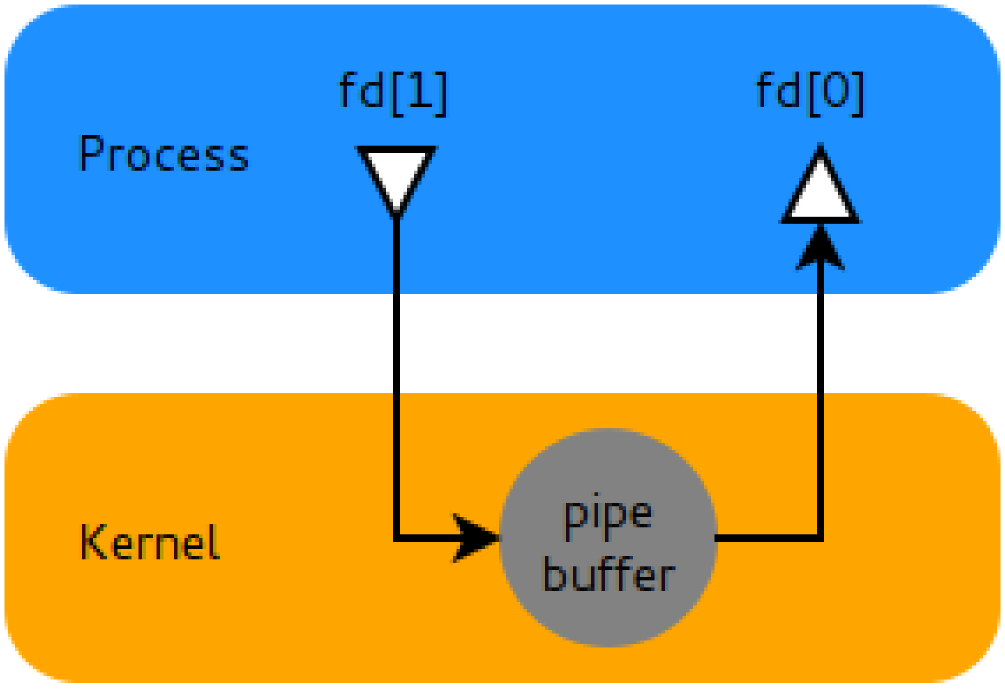
CMPUT 391 Assignment 2
Design Document

Software Defined Network (SDN) using FIFOs



James Hryniw 1431912
Date: 2018-10-30

Objectives

The main objectives of the assignment were to introduce the concepts of inter-process communication using FIFOs, and creating an asynchronous client/server architecture using I/O multiplexing. Serializing, deserializing and handling messages were also an important challenge in creating the network that matched the specifications.

Design Overview

Overall, the design behind my implementation of the controller and switch in *a2sdn* 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 FIFOs, message serialization/deserialization and I/O multiplexing. In later paragraphs, I will explain in more detail how I decided to tackle each of these subcomponents. The highlights of the design are:

- An extensible object-oriented design
- The controller and switches are completely asynchronous
- The packets know how to pack and unpack themselves using streams
- Packets are queued if they cannot be immediately handled

My implementation uses concepts of object oriented programming extensively. High-level controller and switch functionality is shared through the abstract NetworkNode parent, which primarily handles port management. I decided to poll stdin separately from the FIFOs so that all file descriptors in each poll could be handled uniformly. Since both the controller and switch read the same commands, I provide a simple list() template method to specialize its implementation in each of the subclasses.

For I/O multiplexing, I use a non-blocking poll for all nodes. As a design choice, the controller does not wait for the switches to open in order to run. I did this in order to match the results posted in the assignment.

Packet serialization and deserialization was handled by a separate Packet class hierarchy. A basic packet contains a type and header, which handles most cases and the subclasses OpenPacket and AddPacket contain additional payload information (switch and flow rule information, respectively). A simple decoding interface initializes a read stream, reads the type and then delegates the construction of the object to a packet constructor taking an input stream. The encoding interface does the same with output streams. Theoretically this architecture could be made to work even if the packet isn’t written in one chunk.

Finally, when a switch receives a packet that it cannot handle it queues that packet (waiting for an ADD response) while still processing packets. I prevent requerying by storing a map of already made queries.

Other notable design choices:

- I consider FORWARD and DELIVER as two separate actions internally since it made the implementation easier (only FORWARD packets require a relay). However, both are printed as FORWARD in the output so they are indistinguishable externally.

Project Status

Both the controller and switches are functionally complete. I hit one blocking issue during implementation where the destructor of my Port class was being called by accident due to c++ copy semantics. This caused my FIFOs to close right after opening them, giving each new FIFO the same file descriptor (which confused me for a while). Once discovered, it was quickly solved by wrapping them as std::shared_ptrs in the NetworkNode port vector.

Testing and Results

The robust object-oriented design enabled an incremental implementation of the specification. The program was developed and tested in small iterations:

- Input parsing
- The list and exit command
- Bi-directional FIFO communication
- The OPEN and ACK packets
- …

In the early stages, testing the controller and switch was primarily a manual process. Significant effort was put into input domain testing for the command line arguments and list and exit commands. When testing basic I/O functionality, I would often manually pipe input messages into a FIFO to simulate a message from another node:

```
~/.../linear-sdn> echo ‘encoded message’ > fifo-x-y
```

Once most of the functionality was in place, I relied on the data files provided in the specification and on eClass. In each test, I opened the controller then each switch sequentially. Comparing the output of the list command was used for verification, with the received and transmitted packet stats being the most useful criteria. For example, using the data file t2.dat:

Terminal 1:

```
~/.../linear-sdn> a2sdn cont 2

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> a2sdn sw1 t2.dat null sw2 100-110

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

Terminal 3:

```
~/.../linear-sdn> a2sdn sw2 t2.dat sw1 null 200-210

Flow table:
[0] (src= 0-1000, destIP= 200-210, action= FORWARD:3, pri= 4, pktCount= 0)
[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:0
Transmitted: OPEN:1, QUERY:2, RELAYOUT:2
```

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.