CMPUT 379 - Assignment #2 (10%)

Modelling a Software Defined Network using FIFOs (first draft)

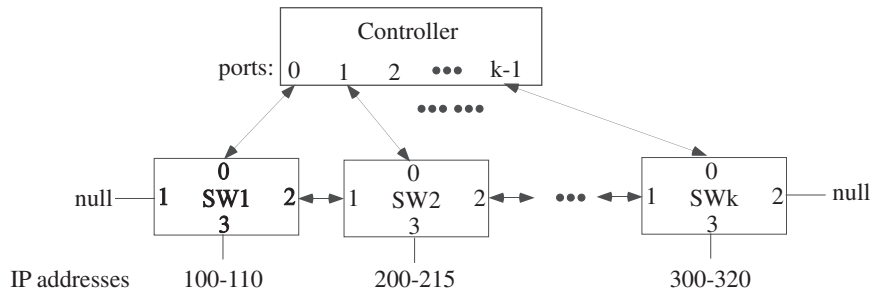
Due: Thursday, November 1, 2018, 09:00 PM
(electronic submission)

Objectives

This programming assignment is intended to give you experience in developing peer-to-peer programs that utilize signals for examining the progress of the running processes, FIFOs for communication, and I/O multiplexing for nonblocking I/O.

Problem Context

A software defined network (SDN) is a computer network that utilizes a new networking architecture where a controller device controls the flow of traffic packets among a number of packet switches. This new architecture is in contrast with the traditional networking architecture utilizing routers that are independent devices that communicate with each other to route their carried traffic. Throughout the assignment, we will deal with linear SDN topologies, as illustrated in the figure below.



In the figure, we have

- A controller with $k \geq 1$ bidirectional ports, and k attached switches
- Each switch has 4 bidirectional ports connected as follows: port 0 links the switch to the controller, port 1 (port 2) links the switch to its left (respectively, right) neighbouring switch (if any exists), and port 3 carries traffic packets between the switch and a number of hosts whose network layer addresses (IP addresses) fall in some range. For example, in the figure `sw1` is connected to hosts with IP addresses in the range $[100 - 110]$. The specified IP address ranges of any two different switches are disjoint.

In SDN, each switch is a simple device that stores a flow table. Each entry in the table includes

- A set of packet header field values to which an incoming packet will be matched
- A set of actions to be taken when a packet matches a flow table entry

- A set of counters that are updated as packets are matched to flow table entries. These actions might be to forward the packet to a given output switch port, or to drop the packet.
- A priority field to disambiguate table entries that may each match the header fields of an incoming packet.

Thus, each entry in a flow table defines a *pattern-action* rule. The pattern is defined by the rule's header field values associated with different protocols at different layers in the protocol stack (e.g., the application layer, the transport layer, the network layer, and the link layer). The action can include forwarding the packet to an output port, or dropping the packet.

Initially (when a switch is rebooted), the flow table is empty. Subsequently, when a packet arrives to the switch, the switch extracts the relevant header fields and compares their values with each entry in the flow table looking for the highest priority rule that matches the extracted fields. If no matching rule exists, the switch queries the controller to get a rule to apply and store in its flow table.

The `a2sdn` Program

In this assignment, you are asked to write a C/C++ program, called `a2sdn`, that implements the transactions performed by a simple linear SDN. The program can be invoked as a controller using `"a2sdn cont nSwitch"` where `cont` is a reserved word, and `nSwitch` specifies the number of switches in the network (at most `MAX_NSW= 7` switches). The program can also be invoked as a switch using `"a2sdn swi trafficFile [null|swj] [null|swk] IPlow-IPhigh"`. In this form, the program simulates switch `swi` by processing traffic read from file `trafficFile`. Port 1 and port 2 of `swi` are connected to switches `swj` and `swk`, respectively. Either, or both, of these two switches may be `null`. Switch `swi` handles traffic from hosts in the IP range `[IPlow-IPhigh]`. Each IP address is $\leq$ `MAXIP` $(= 1000)$.

Data transmissions among the switches and the controller use FIFOs. Each FIFO is named `fifo-x-y` where $\texttt{x} \neq \texttt{y}$, and $\texttt{x} = 0$ (or, $\texttt{y} = 0$) for the controller, and $\texttt{x}, \texttt{y} \in [1, \texttt{MAX\_NSW}]$ for a switch. Thus, e.g., `sw2` sends data to the controller on `fifo-2-0`.

The Flow Table

In the assignment, each entry in a flow table stores the following information:

```
[srcIP_lo, srcIP_hi, destIP_lo, destIP_hi, actionType,
actionVal, pri, pktCount]
```

where

- `srcIP` and `destIP` are packet header fields that are part of each traffic packet. `(srcIP_lo, srcIP_hi)` and `(destIP_lo, destIP_hi)` specify ranges of IP addresses in the rule. A packet matches the pattern in a flow table if $\texttt{s}srcIP \in [srcIP\_lo, srcIP\_hi]$ and $\texttt{d}destIP \in [destIP\_lo, destIP\_hi]$.
- `actionType` can either be *forward* or *drop* a matched packet. If the `actionType` is to forward a packet then `actionVal` specifies the switch port to which the packet should be forwarded. Else, if `actionType` is to drop the packet then `actionVal` is not used.

- `pri` specifies a rule priority value (0 is the highest, `MINPRI= 4` is the lowest).

  Note: In the assignment, the use of the `pri` field is not tested. However, it is an important field that plays a role when the controller issues overlapping rules.

We assume that the flow table in each switch is large enough to hold rules for processing at most 100 arriving traffic packet headers to the switch.

Note: For simplicity, only packet headers are routed in the network. In real life, entire packets (with their header and data fields) are routed in a network.

Traffic File Format

Each switch reads its arriving traffic packets from a common `trafficFile`. The file has a number of lines formatted as follows:

- A line can be empty
- A line that starts with `'#'` is a comment line
- A line of the form `"swi srcIP destIP"` specifies that a packet with the specified source and destination IP addresses has reached port 3 of `swi`. You may assume that the `srcIP` address lies within the range handled by the switch. Only `swi` processes this packet header; other switches ignore the line.

More details on processing `trafficFile` are given below.

Packet Types

Communication in the network uses messages stored in formatted packets. Each packet has a type, and carries a message (except ACK packets). Your program should support at least the following packet types.

- `OPEN` and `ACK`: When a switch starts, it sends an `OPEN` packet to the controller. The carried message contains the switch number, the numbers of its neighbouring switches (if any), and the range of IP addresses served by the switch. Upon receiving an `OPEN` packet, the controller updates its stored information about the switch, and replies with a packet of type `ACK` (no carried message).
- `QUERY` and `ADD`: When processing an incoming packet header (the header may be read from the traffic file, or relayed to the switch by one of its neighbours), if a switch does not find a matching rule in the flow table, the switch sends a `QUERY` packet to the controller. The controller replies with a rule stored in a packet of type `ADD`. The switch then stores and applies the received rule.
- `RELAY`: A switch may forward a received packet header to a neighbour (as instructed by a matching rule in the flow table). This information is passed to the neighbour in a `RELAY` packet.

The Controller Loop

When `a2sdn` is invoked to simulate a controller, the program uses I/O multiplexing (e.g., `select()` or `poll()`) to handle I/O from the keyboard, and the attached switches in a nonblocking manner. Each iteration of the main loop of the controller performs the following steps:

1. Poll the keyboard for a user command. The user can issue one of the following commands.
   - **list**: The program writes the stored information about the attached switches that have opened connection with the controller. As well, for each transmitted or received packet type, the program writes an aggregate count of handled packets of this type.
   - **exit**: The program writes the above information and exits.
2. Poll the incoming FIFOs from the attached switches. The controller handles each incoming packet, as described in the Packet Types section.

In addition, upon receiving signal `USER1`, the controller displays the information specified by the `list` command.

The Switch Loop

When `a2sdn` is invoked to simulate a switch, the program installs an initial rule in its flow table:

```
[srcIP_lo= 0, srcIP_hi= MAXIP, destIP_lo= IPlow, destIP_hi=
IPhigh, actionType= FORWARD, actionVal= 3, pri= MINPRI,
pktCount= 0]
```

The rule matches an arriving packet header with any possible `srcIP` $\leq$ `MAXIP` value, and a `destIP` $\in$ `[IPlow-IPhigh]`. Recall that the range `[IPlow-IPhigh]` is specified on the command line as addresses handled by the switch.

The program then uses I/O multiplexing to handle I/O from the keyboard, the controller, and the attached switches in a nonblocking manner. Each iteration of the main loop performs the following steps:

1. Read and process a single line from the traffic line (if the EOF has not been reached yet). The switch ignores empty lines, comment lines, and lines specifying other handling switches. A packet header is considered *admitted* if the line specifies the current switch.

   Note: After reading all lines in the traffic file, the program continues to monitor and process keyboard commands, and the incoming packets from neighbouring devices.
2. Poll the keyboard for a user command. The user can issue one of the following commands.
   - **list**: The program writes all entries in the flow table, and for each transmitted or received packet type, the program writes an aggregate count of handled packets of this type.
   - **exit**: The program writes the above information and exits.

3. Poll the incoming FIFOs from the controller and the attached switches. The switch handles each incoming packet, as described in the Packet Types section.

In addition, upon receiving signal `USER1`, the switch displays the information specified by the `list` command.

Example 1. In this example, we open two terminal windows on the same lab workstation. On the first window, we invoke `a2sdn` as a controller. On the second window, we invoke `a2sdn` as a switch, as described in the traffic file below:

```
# Traffic file for a2sdn
#     a2sdn cont 1
#     a2sdn sw1 t1.dat null null 100-110

sw1  100  102
sw2  200  300
sw3  200  300
sw1  100  103
sw1  100  104
```

- A list command to the controller produces the following output:

```
Switch information:
[sw1] port1= -1, port2= -1, port3= 100-110

Packet Stats:
       Received:    OPEN:1, QUERY:0
       Transmitted: ACK:1, ADD:0
```

  Here, `port= -1` refers to the null connection of a port.

- A list command to switch `sw1` produces the following output:

```
Flow table:
[0] (srcIP= 0-1000, destIP= 100-110, action= DELIVER:3, pri= 4, pktCount= 3)

Packet Stats:
       Received:    ADMIT:3, ACK:1, ADDRULE:0, RELAYIN:0
       Transmitted: OPEN:1, QUERY:0, RELAYOUT:0
```

  Here, the switch did not query the controller since all admitted packets are handled by the initial rule installed by the switch.

Example 2. In this example, we open three terminal windows on the same lab workstation. On the first window, we invoke `a2sdn` as a controller. On the second and third windows, we invoke `a2sdn` as switches `sw1` and `sw2`, respectively, as described in the traffic file below:

```
# Traffic file for a2sdn
#     a2sdn cont 2
#     a2sdn sw1 t2.dat null sw2 100-110
#     a2sdn sw2 t2.dat sw1 null 200-210
```

```
sw1  100  102
sw1  100  200
sw1  100  300
sw2  200  300
sw3  200  300
sw2  200  100
sw2  200  100
sw1  100  103
sw1  100  104
```

- A list command to the controller produces the following output:

```
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
```

- A list command to switch `sw1` produces the following output:

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

Note that rule [1] in the flow table drops a packet with `dstIP= 200` instead of forwarding the packet to `sw2`. This can happen if `sw1` queries the controller before the start of `sw2`. The controller then instructs `sw1` to drop the packet.

- A list command to switch `sw2` produces the following output:

```
Flow table:
[0] (srcIP= 0-1000, destIP= 200-210, action= DELIVER:3, pri= 4, pktCount= 0)
[1] (srcIP= 0-1000, destIP= 300-300, action= DROP:0, pri= 4, pktCount= 1)
[2] (srcIP= 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
```

More Details

1. This is an individual assignment. Do not work in groups.

2. Only standard include files and libraries provided when you compile the program using `gcc` or `g++` should be used.

3. Although many details about this assignment are given in this description, there are many other design decisions that are left for you to make. In such cases, you should make reasonable design decisions that do not contradict what we have said and do not significantly change the purpose of the assignment. Document such design decisions in your source code, and discuss them in your report. Of course, you may ask questions about this assignment (e.g., in the Discussion Forum) and we may choose to provide more information or provide some clarification. However, the basic requirements of this assignment will not change.

4. When developing and testing your program, **make sure you clean up all processes before you logout of a workstation.** Marks will be deducted for processes left on workstations.

Deliverables

1. All programs should compile and run on the lab machines (e.g., ug[00 to 34].cs.ualberta.ca) using only standard libraries (e.g., standard I/O library, math, and pthread libraries are allowed).

2. Make sure your programs compile and run in a fresh directory.

3. Your work (including a Makefile) should be combined into a single tar archive 'submit.tar'.

   (a) Executing 'make' should produce the `a2sdn` executable file.

   (b) Executing 'make clean' should remove unneeded files produced in compilation.

   (c) Executing 'make tar' should produce the 'submit.tar' archive.

   (d) Your code should include suitable internal documentation of the key functions. If you use code from the textbooks, or code posted on eclass, acknowledge the use of the code in the internal documentation. Make sure to place such acknowledgments in close proximity of the code used.

4. Typeset a project report (e.g., one to three pages either in HTML or PDF) with the following (minimal set of) sections:

   – **Objectives:** state the project objectives and value from your point of view (which may be different from the one mentioned above)

   – **Design Overview:** highlight in point-form the important features of your design

   – **Project Status:** describe the status of your project; mention difficulties encountered in the implementation

   – **Testing and Results:** comment on how you tested your implementation

- **Acknowledgments:** acknowledge sources of assistance

5. Upload your tar archive using the **Assignment #2 submission/feedback** links on the course's web page. Late submission is available for 24 hours for a penalty of 10% of the points assigned to the phase.

6. It is strongly suggested that you **submit early and submit often**. Only your **last successful submission** will be used for grading.

Marking

Roughly speaking, the breakdown of marks is as follows:

20% : successful compilation of reasonably complete program that is: modular, logically organized, easy to read and understand, includes error checking after important function calls, and acknowledges code used from the textbooks or the posted lab material

05% : ease of managing the project using the makefile

65% : correctness of implementing the controller and the switch modules and displaying the required information using keyboard commands and signals.

10% : quality of the information provided in the project report