Networks and Distributed Systems	Project 1: Simple Bridge
CS3700 Spring 2015	September 21, 2014

This project is due at 11:59:59pm on February 2, 2015 and is worth 12% of your grade. You must complete it with a partner. You may only complete it alone or in a group of three if you have the instructor's explicit permission to do so for this project.

Note that there is a milestone deadline for this project, at 11:59:59pm on February 2, 2015. More details are in the Milestone section below.

1 Description

You will implement a simple bridge that is able to establish a spanning tree and forward frames between its various ports. Since running your bridge on Northeastern's network would likely impact real traffic, we will provide you with a simulation environment that emulates LANs and end hosts. Your bridges must construct a spanning tree (and disable any ports that are not part of the tree), must forward frames between these ports, learn the locations (ports) of hosts, and handle both bridge and port failures (e.g., by automatically reconfiguring the spanning tree).

Your bridges will be tested for both correctness and performance. Part of your grade will come from the overhead your network has (i.e., lower overhead incurs a higher score) and the fraction of packets that are successfully delivered between end hosts. Your results will be compared to your classmates' via a leaderboard.

2 Requirements

To simplify the project, instead of using real packet formats, we will be sending our data across the wire in JSON (many languages have utilities to encode and decode JSON, and you are welcome to use these libraries). Your bridge program must meet the following requirements:

- Form a spanning tree in order to prevent packet loops
- Handle the failure of bridges, the failure of bridge ports, and the introduction of new bridges and LANs over time
- Learn the locations of end hosts
- Deliver end host packets to the destination
- Handle the mobility of end hosts between LANs
- The send and receiver must print out specified debugging messages to STDOUT

You should implement a simplified version of the standard bridge spanning tree protocol that we discussed in class. Note that more sophisticated and properly tuned algorithms (i.e., those which perform better) will be given higher credit. For example, some desired properties include (but are not limited to):

- Fast convergence: Require little time to form a spanning tree.
- Low overhead: Reduce packet flooding when possible.

Regardless, correctness matters most; performance is a secondary concern. We will test your code and measure these two performance metrics; better performance will result in higher credit. We will test your code by introducing a variety of different errors and failures; you should handle these errors gracefully, recover, and *never* crash.

3 Your program

For this project, you will submit one program: a program `3700bridge` that implements a bridge. You may use any language of your choice, and we will give you basic starter code for a few different languages. You may *not* use any bridging libraries in your project; you must implement all of the bridge logic yourself.

If you use C or any other compiled language, your executable should be named `3700bridge`. If you use an interpreted language, you *must* write a brief Bash shell script, named `3700bridge`, that conforms to the input syntax above and then launches your program with whichever incantations are necessary. For example, if you write your solution in Java, your Bash script might resemble

```
#!/usr/bin/perl -w
$args = join(' ', @ARGV);
print `java -jar 3700bridge.jar $args`;
```

or, if you use python, your program should look like

```
#!/usr/bin/python
import foo from bar
...
```

and should be marked executable.

3.1 Starter code

Very basic starter code for the assignment in perl is available in `/course/cs3700sp15/code/project1`. Provided is a simple implementation of a bridge that simply connects to the LANs and broadcasts a "Hello world!" message twice every second. You may use this code as a basis for your project, but it is *strongly* recommended that you do not unless you are very comfortable with perl. To get started, you should copy down this directory into your own local directory (i.e., `cp -r /course/cs3700sp15/code/project1 ~/`).

3.2 Requirements

The command line syntax for your bridge is given below. The bridge program takes command line arguments representing (a) the ID of the bridge, and (b) the LANs that it should connect to. The bridge must be connected to at least one LAN. The syntax for launching your bridge is therefore:

```
./3700bridge <id> <LAN> [<LAN> [<LAN> ...]]
```

`id` (Required) The id of the bridge. For simplicity, all bridge ids are four-digit hexadecimal numbers (e.g., `0aa1` or `f29a`).

`LAN` (Required) The unique name of the LAN(s) the bridge should connect to. The LANs are named using ASCII strings; the names themselves are not

You should configure your bridges to periodically broadcast BPDUs on all points. You should broadcast BPDUs no more frequently than once every 500 ms. Using those BPDUs, you should constantly be listening for new roots, new bridges, etc, and should make decisions about which ports are active and inactive upon receiving each BPDU. Additionally, you should "timeout" BPDUs after 750 ms. To aid in grading and debugging, your bridge program should print out messages about the spanning tree calculation to STDOUT. When starting up, your bridge should print out

```
Bridge <id> starting up
```

where `<id>` is the ID of the bridge. When your bridge selects a new root, it should print out

```
New root: <id>/<root>
```

where `<id>` is the ID of the local bridge and `<root>` is the ID of the new root. When your bridge changes its root port, it should print out

```
Root port: <id>/<port_id>
```

where `<port_id>` is the port number (0-indexed). Finally, when your bridge decides that a port is the designed port for a LAN, it should print out:

```
Designated port: <id>/<port_id>
```

and when your bridge decides that a port should be disabled, it should print out:

```
Disabled port: <id>/<port_id>
```

Additionally, your bridge should build up a forwarding table as discussed in class. Your should "timeout" forwarding table entries 5 s after receiving the last message from that address. Additionally, when any of your bridge's ports changes state (designated, root, etc), you should flush your forwarding table. When forwarding data packets, your bridge program should print out the following messages to STDOUT. When your bridge receives a message on an *active* port (i.e., not disabled), it should print out

```
Received message <id> on port <port_id> from <source> to <dest>
```

where `<id>` is the unique identifier of the data message, `<port_id>` is the port number on the bridge that the message was received on, and `<source>` and `<dest>` are the source and destination of the message. Once your bridge makes a forwarding decision about the message, it should print out one of three messages:

```
Forwarding message <id> to port <port_id>
```

or

```
Broadcasting message <id> to all ports
```

or

```
Not forwarding message <id>
```

Thus, every message your bridge receives should have one of the above three lines printed out. This will help you to debug why your bridges are misrouting messages (if this should ever occur).

You should develop your client program on the CCIS Linux machines, as these have the necessary compiler and library support. You are welcome to use your own Linux/OS X/Windows machines, but you are responsible for getting your code working, and your code *must* work when graded on the CCIS Linux machines. If you do not have a CCIS account, you should get one ASAP in order to complete the project.

3.3 Packet format

In order to simplify the development and debugging of this project, we use JSON (JavaScript Objection Notation) to format all messages sent on the wire. Most common programming languages have built-in support for encode and decoding JSON messages, and you should use these when sending and receiving messages (i.e., you do not have to create or parse JSON messages yourself). The format of all messages is

```
{"source":"<source>", "dest":"<destination>", "type":"<type>", "message":{<message>}}
```

where `<source>` and `<destination>` are either bridge or end host addresses. Recall that all addresses are four-byte hexadecimal numbers (e.g., 98a2), and a special broadcast address `ffff` indicates the packet should be received by all hosts and bridges. Additionally, `<message>` should be the JSON-encoded message itself, and `type` is either `bpdu` for BPDUs or `data` for end-host data packets. For example, a BPDU that you send from bridge 02a1 might look like

```
{"source":"92b4", "dest":"ffff", "type": "bpdu",
  "message":{"id":"92b4", "root":"02a1", "cost":3, "port":2}}
```

All data packets will include a unique `id` field that you should use refer to that message. For example, a data message from host 28aa to 97bf might look like

```
{"source":"28aa", "dest":"97bf", "type": "data", "message":{"id": 17}}
```

3.4 Connecting to the LANs

We will be using UNIX domain sockets to emulate the LANs, with one domain socket per LAN that your bridge is connected to. You do not need to be intimately familiar with how these work, but they essentially give you a socket-like device that you can read and write from. Whenever you write to it, all other end hosts and bridges on that LAN will receive your message. You should constantly be reading from it to make sure you receive all messages (they will be buffered if you don't read immediately).

One thing to note is that we will be using *abstract domain sockets*, which means that you should put a `\0` byte before the LAN name. So, if you were trying to connect to the LAN named

LAN#amislove#1, the name that you would actually connect to is \0LAN#amislove#1. We will be using the SOCK_SEQPACKET socket type, which basically provides a reliable message-oriented stream.

Exactly how to connect to a UNIX domain socket depends on your programming language. For example, if you were using perl to complete the project, your code for connecting would look like:

```
use IO::Socket::UNIX;

my $lan = IO::Socket::UNIX->new(
   Type => SOCK_SEQPACKET,
   Peer => "\0<lan>"
);
```

where <lan> is the name of the LAN that is passed in on the command line. You can then read and write from the $lan variable. In python, your code would look like

```
from socket import socket, SOCK_SEQPACKET, AF_UNIX

s = socket (AF_UNIX, SOCK_SEQPACKET)
s.connect ('\0<lan>')
```

with similar results.

I would encourage you to write your code in an event-driven style using `select()` or `poll()` on all of the LANs that your bridge is connected to. This will keep your code single-threaded and will make debugging your code significantly easier. Alternatively, you can implement your bridge in a threaded model (with one thread attached to each LAN), but expect it to be significantly more difficult to debug.

4 Testing your code

In order for you to test your code in our network simulator, we have included a perl script that will create the emulated LANs, run your bridge program and connect it to these LANs, start and stop your bridges in a configurable way, and create and record end host traffic. This script is included in the starter code, and you can run it by executing

```
./run <config-file>
```

where `config-file` is the configuration file that describes the network you would like to implement.

4.1 Config file format

The configuration file that you specify describes the LANs, the bridges, and the connections between these. It also contains information about when bridge come up and down, and the end host traffic that should be generated. The file is formatted in JSON and has the following elements

`lifetime` (Required) The number of seconds the simulation should run for.

`bridges` (Required) An array of bridge elements (described below). At least one bridge must be specified. Each bridge element is an associative array that has the following properties:

- `id` (Required) The ID of the bridge, a string.
- `lans` (Required) An array of the LAN IDs that the bridge is connected to. All LANs are identified by a non-negative number.
- `start` (Optional) The start time (in seconds) when the bridge should be turned on. If not specified, the bridge is started at the beginning of the simulation.
- `stop` (Optional) The stop time (in seconds) when the bridge should be turned off. If not specified, the bridge is stopped at the end of the simulation.

`hosts` (Required) The number of hosts to generate (these are randomly attached to LANs).

`traffic` (Required) The number of end host packets to randomly generate (these are sent with randomly selected sources and destinations).

`wait` (Optional) The number of seconds to wait before sending any data traffic (default of 2 seconds).

`seed` (Optional) The random seed to choose. If not specified, a random value is chosen. Setting this value will allow for a reproducible set of hosts and traffic.

For example, a simple network with two LANs connected by a single bridge would be:

```
{
   "lifetime": 30,
   "bridges": [{"id": "A", "lans": [1, 2]}],
   "hosts": 10,
   "traffic": 1000
}
```

and a more complex network may be

```
{
   "lifetime": 30,
   "bridges": [{"id": "A", "lans": [1, 3]},
               {"id": "B", "lans": [2, 3], "stop": 7},
               {"id": "C", "lans": [1, 2], "start": 5, "stop": 9},
               {"id": "D", "lans": [2, 4]},
               {"id": "E", "lans": [2, 4, 5, 6]}]
   "hosts": 100,
   "traffic": 10000
}
```

4.2 `run` output

The output of the `run` script includes timestamps and all logging information from your bridges and the emulated end hosts. Note that all data traffic will be delayed for 2 seconds at the beginning of the simulation to allow your bridges to form an initial spanning tree. At the end, the output also includes some statistics about the your bridges' performance:

```
bash$ ./run config.json
...
[ 14.9990    Host ed10] Sent message 776 to 41c1
[ 15.0001  Bridge 92ba] Received message 776 on port 0 from ed10 to 41c1
Simulation finished.
Total packets sent: 6730
Total data packets sent: 2000
Total data packets received: 1984
Total data packets dropped: 16  (message ids 52, 70, 181, 320, 517, 571, 634, 776, 900, 1111,
Total data packets duplicated: 17  (message ids 311, 433, 541, 630, 632, 658, 717, 804, 998,
Data packet delivery ratio: 0.992000
```

each of the fields is self-explanatory. Ideally, you would like all messages to be delivered (a delivery ratio of 1.0), the number of packets dropped and duplicated (a message can cause two packets if it the network is being re-configured). Additionally, you want the number of total packets sent to be low as well (this includes BPDUs, which are overhead).

4.3 Testing script

Additionally, we have included a basic test script that runs your code under a variety of different network configurations and also checks your code's compatibility with the grading script. If your code fails in the test script we provide, you can be assured that it will fare poorly when run under the grading script. To run the test script, simply type

```
amislove@gorf:~/project1$ ./test
Basic (no failures, no new bridges) tests (PDR = 1.0)
  One bridge, one LAN                                    [PASS]
  One bridge, two LANs                                   [PASS]
  One bridge, three LANs                                 [PASS]
  Two bridges, one LAN                                   [PASS]
  Two bridges, two LANs                                  [PASS]
```

This will run your code on a number of configurations, and will output whether your program performs sufficiently. If you wish to run one of the tests manually, you can do so with

```
amislove@gorf:~/project1$ ./run basic-4.conf
```

4.4 Performance testing

As mentioned in class, 15% of your grade on this project will come from performance. Your project will be graded against the submissions of your peers. To help you know how you're doing, the testing script will also run a series of performance tests at the end; for each test that you successfully complete, it will report your time elapsed and bytes sent to a common data base. For example, you might see

```
Performance tests
  Network 1                                          [PASS]
    99.1% packets delivered, 3.0% overhead
```

This indicates that you successfully delivered 99.1% of all end-host packets and had an overhead of 3%. This score will be reported to the common database.

In order to see how your project ranks, you can run

```
amislove@gorf:~$ /course/cs3700sp15/bin/project1/printstats
----- TEST: Eight bridges, eight LANs -----
Least overhead:
1: amislove                                    200 packets
2: foo                                         220 packets

Highest delivery ratio:
1: foo                                         1.00000
2: amislove                                    0.950000
```

which will print out the rank of each group for each performance test, divided into the number of packets sent and the delivery ratio. In this particular example, `amislove`'s project has lower overhead but delivers fewer of the packets. Obviously, you would ideally have both more packets delivered and fewer packets sent.

5 Grading

The grading in this project will consist of

60% Program correctness

15% Performance

15% Style and documentation

10% Milestone functionality

6 Submitting your project

6.1 Registering your team

You and your partner should first register as a team by running the `/course/cs3700sp15/bin/register` script. You should pick out a team name (no spaces or non-alphanumeric characters, please) and run

```
/course/cs3700sp15/bin/register project1 <teamname>
```

This will either report back success or will give you an error message. If you have trouble registering, please contact the course staff.

You must register your team by 11:59:59pm on January 21, 2015.

6.2 Milestone

In order to ensure that you are making sufficient progress, you will have an interim milestone deadline. For the milestone, your code must pass the Basic (no failures, no new bridges) tests in the testing script. In other words, your code must correctly deliver all packets when the set of bridges is static and does not change over the course of the simulation.

You should submit your milestone by running the `/course/cs3700sp15/bin/turnin` script. Specifically, you should create a `project3` directory, and place all of your code in it. Then, run

```
/course/cs3700sp15/bin/turnin project1-milestone <dir>
```

Where `<dir>` is the name of the directory with your submission. The script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit! You should receive an email confirmation of your submission.

You must submit your milestone by 11:59:59pm on February 2, 2015. No slip days can be used on the milestone.

6.3 Final submission

For the final submission, you should submit your (thoroughly documented) code along with a plain-text (no Word or PDF) README file. In this file, you should describe your high-level approach, the challenges you faced, a list of properties/features of your design that you think is good, and an overview of how you tested your code. You should also describe the basic transport protocol that you implemented, and why you made that choice.

You should submit your project by running the `/course/cs3700sp15/bin/turnin` script. Specifically, you should create a `project1` directory, and place all of your code and README files in it. Then, run

```
/course/cs3700sp15/bin/turnin project1 <dir>
```

Where `<dir>` is the name of the directory with your submission. Again, the script will print out every file that you are submitting, make sure that it prints out all of the files you wish to submit! You should receive an email confirmation of your submission.

You must submit your project by 11:59:59pm on February 9, 2015.

7 Advice

A few pointers that you may find useful while working on this project:

- Start by getting your code working on simple network configurations. Slowly introduce more complex networks. Get in the habit of using the random seed so that you can reproduce errors when they occur (to aid debugging).
- Check the Piazza forum for question and clarifications. You should post project-specific questions there first, before emailing the course staff.
- Finally, get started early and come to the instructor's office hours and TA lab hours. You are welcome to come to the lab and work, and ask the TA any questions you may have.