Data Networks Project 3: Implementing Intra-Domain Routing Protocols

Assigned: Thursday, 28 June 2007

Due: 11:59pm, Friday, 20 July 2007

1 Assignment

In this project, we ask you to implement the Distance Vector (DV) and the Link State (LS) routing protocols for the Bisco GSR9999 router. Your routing software will be tested in a *simulator* that simulates a network of Bisco GSR9999 routers. Your implementation of these routing protocols **must** follow the specifications provided in the Project 2 assignment **as well as** the additional instructions provided in this document.

The source code of the simulator and some input/output files for testing purpose are available from the course web page.
To extract the individual files from the archive, run `tar xf project3.tar`.

The expected simulator output for 3 trivial scenarios for the DV protocol is provided (see `simpletest*`). You may want to begin by implementing the DV protocol first. No expected simulator output for the LS protocol is provided.

The majority of the provided source code is for the simulator and you do not need to study this code. To complete this project, you only need to understand how to use the simulator to run tests, how to invoke the GSR9999 system interfaces defined in `Node.h`, how to extend the `RoutingProtocolImpl.[h,cc]` files to implement your routing software, and use the constants already defined in `global.h`. (Throughout this document, we refer to the C++ source code. Corresponding classes with the same names can be found in the Java version of the simulator. Obviously, there are no header files in Java. Also, a README file in the Java distribution provides some hints.)

In summary, the code files you need to look at are:

- `Node.h` for the GSR9999 system interfaces
- `global.h` for the defined constants that you **must** use
- `RoutingProtocolImpl.[h,cc]` where you should begin your implementation

The RoutingProtocolImpl C++ class in the provided code is the abstract interface for your routing protocol software. It defines the set of interfaces that your routing protocol software must handle (i.e. init(), handle_alarm(), recv(), just like the specification in Project 2). You will implement these interfaces as well as the functions of the DV and LS protocols.

Your implementation code must be contained in the files RoutingProtocolImpl.h, RoutingProtocolImpl.cc, and additional files of your own creation. Other than RoutingProtocolImpl.[h,cc] and the Makefile, you are **not** allowed to modify any other parts of the provided code. In fact, during grading, any changes that you make to the other provided code files will be completely ignored. Your implementation must compile without warnings on a CIP pool machine.

Grading: Your project will be graded based on the performance of your submission on a set of test scripts. We will make most of these scripts available to you well in advance of the submission deadline, so you can test your submissions on these scripts. Some of the tests will run your submission in configurations where it needs to interoperate with a different implementation. To test these configurations, we encourage you to swap your implementations with other groups to test them for interoperability.

Working in groups: In most cases, you will be working with your partner from Project 2. If you need to change partners, please contact the course staff.

Submission: Please pack your submission into a tar archive and mail it to datanets-projects@mpi-sws.mpg.de by 11:59pm on the due date. In your email, specify the names and ids of the students in your group. Your tar archive must contain a Makefile that compiles your implementation automatically, as described below. Please do not add to or change the command line arguments of the simulator, or the format of the configuration file.

2 Simulator Overview

After you compile the code using the provided Makefile, you will have the executable of the simulator, with your RoutingProtocolImpl implementation included in the executable. When you add additional files, you will need to update the Makefile accordingly to include them.

The simulator executable can be run as follows: `Simulator config_file DV|LS`. The `config_file` specifies the topology of the simulated network and certain network events, it is explained in Section 2.1. The `DV|LS` switch is used to indicate whether the simulator should use the DV protocol or the LS protocol.

2.1 Configuration File

See the configuration file `simpletest3` included in the code as an example. A configuration file contains 3 sections. The first section is `[nodes]`, which simply lists the IDs of the routers/nodes in the network. The second is the `[links]` section, where each link in the network is specified. For example, `(1,2) delay 0.010 prob 0.0` means that a link exists between routers 1 and 2, the delay is 0.010 second (i.e. 10ms), and the packet loss probability is 0.0 (i.e. no packet loss). Using `[nodes]` and `[links]` declarations, you can create arbitrary network topologies.

The final section is `[events]`, which specifies what should happen to the network in the middle of a simulation. There are 4 kinds of valid events in the configuration file.

- xmit. `0.01 xmit (2,4)` means that at time 0.01 second, a DATA packet is generated at router ID 2 destined for router ID 4.
- linkdying. `450.00 linkdying (3,4)` means that at time 450 seconds, the link between router ID 3 and 4 will fail. No packet can be delivered over a link that has failed.

- linkcomingup. `750.00 linkcomingup (3,4)` means that at time 750 seconds, the link between router ID 3 and 4 will be healed. Once a link is back up, delivery of packets resumes as normal.

- end. `1000.00 end` means that the simulation should terminate at time 1000 seconds. If this event is not specified, the simulator runs indefinitely.

You should familiarize yourself with the provided configuration files `simpletest{1,2,3}` and try to create some configuration files of your own. The supplied configuration files are only meant to get you started. We will provide other configurations files covering more test cases later. However, you are strongly encouraged to create additional configuration files of your own to help test your routing protocols.

2.2 Simulator Output

The simulator prints outputs to the screen. These outputs provide details of your routing software's operation to help you determine whether your software is running correctly. For example, look at `simpletest1.out`, which is the output generated by the simulator running an implementation of DV on the input file `simpletest1`. Look at lines after `Step 2: Simulator beginning to run...`. Each line in the output starts with the time of an event, and a description of the event. For example, from looking at `simpletest1.out`, we can see that the DV implementation is generating PONG packets immediately in response to PING packets (which is the desired behavior). Also, by tracing the PING packets, we can also see that PING packets are correctly being sent once every 10 seconds. Below is a list of all the output events and their explanations (see `simpletest*.out` for examples).

- Event_Xmit_Pkt_On_Link. `time = 0 Event_Xmit_Pkt_On_Link (1,2) packet type is PING` means that at time 0 second, a packet of type PING is transmitted on link (1,2) in the direction from router ID 1 to router ID 2.

- Event_Recv_Pkt_On_Node. `time = 0.01 Event_Recv_Pkt_On_Node 2 packet type is PING` means that at time 0.01 second, a PING packet is receieved by router ID 2.

- Event_Alarm. `time = 1 Event_Alarm on node 2` means that at time 1 second, a previously scheduled alarm is triggered at router ID 2. This alarm in `simpletest1.out` actually corresponds to the checking of state freshness every second.

- Event_Xmit_Data_Pkt. `time = 1 Event_Xmit_Data_Pkt source node 2 destination node 1 packet type is DATA` means that at time 1 second, a DATA packet is originating from router ID 2 destined for router ID 1. By following the events associated with the DATA packet, you can see whether the network route taken by the DATA packet is correct.

- Event_Link_Die. `time = 450 Event_Link_Die (3,4)` means that at time 450 seconds, the link (3,4) fails.

- Event_Link_Come_Up. `time = 750 Event_Link_Come_Up (3,4)` means that at time 750 seconds, the link (3,4) is healed.

You should take a look at `simpletest1.out` and convince yourself that the outputs represent the correct behavior of a DV implementation running on the `simpletest1` network topology. In testing your implementation, you should also look at the outputs to verify correctness.

2.3 Accessing GSR9999 System Interfaces

Take a look at `RoutingProtocolImpl.h`. When your `RoutingProtocolImpl` instance is constructed, a `Node *n` pointer is passed in by the caller. The `Node *n` pointer is stored in the instance variable `sys` (see `RoutingProtocolImpl.cc`.) Subsequently, your RoutingProtocolImpl functions can access the GSR9999 System Interfaces via the `sys` pointer. For example, from within any `RoutingProtocolImpl` function, to get the current time, you call:

```
sys->time().
```

To send a packet, you call:

```
sys->send(port, packet, size).
```

To set an alarm, you call:

```
sys->set_alarm(this, duration, d).
```

See `Node.h` for additional explanation of the GSR9999 system interfaces.

Our simulator is not a complete computing environment, so you will still need to use the basic system calls provided by UNIX to do this project. For example, you will most likely need to use `malloc()`, `calloc()`, `htons()`, `ntohs()`, etc.

Also, please refer to the project 2 assignment document for additional information about the GSR9999 system interfaces.

3 Testing

As mentioned earlier, in the code distribution, we have included some basic test cases and the corresponding output from a DV implementation to help you start with testing your own code. The files are `simpletest*`. The `.desc` file describes the test case. The configuration file (without extension) specifies the network topology and special simulator events such as `xmit`. The `.out` file contains the simulator output of a DV implementation. Your DV implementation should produce output very similar (if not entirely identical) to that of the `.out` files. If you have reasons to believe the `.out` files are wrong, please let the course staff know.

IMPORTANT The 3 provided test configurations are extremely trivial and are not meant to stress test the correctness of your system. In fact, even if your DV implementation produces outputs exactly identical to the `.out` files provided does not mean that your DV implementation is correct. To properly test your implementation, you are strongly encouraged to develop code to test individual parts of your implementation (e.g. the Dijkstra's algorithm can be tested in isolation), as well as complex topologies with link failures to test the overall correctness of your implementation. We will also provide more test cases that help you test your software.

4 C++ Issues

Following are a few notes about the implementation of the C++ version of the simulator. If you are a C++ expert, you may ignore the following discussion.

First of all, you need to be aware that your `RoutingProtocolImpl` class (i.e. your routing protocol software) will be instantiated into multiple copies by the simulator. Each copy will be responsible for managing the routing of one router in the simulated network. What this means is that you must not define or use any global variables in your routing protocol implementation. This is because if you have global variables, these variables will be shared, accessed, and modified by all copies of `RoutingProtocolImpl`, which leads to incorrect results.

Instead, define your non-local variables inside the `RoutingProtocolImpl` class in `RoutingProtocolImpl.h`. One example is the `Node *sys` variable that is already in the `RoutingProtocolImple.h` file. By declaring your non-local variables inside the class, a copy of these variables will be made when the `RoutingProtocolImpl` object is created. Inside your `RoutingProtocolImpl` functions (e.g. `recv()`), you can use these class variables *as if* they were global variables in C. For example, you can access `sys` in `RoutingProtocolImpl::RoutingProtocolImpl()` even though it isn't a local variable of the function.

Similarly, you should declare additional functions you define to implement your routing protocol features inside the `RoutingProtocolImpl` class in `RoutingProtocolImpl.h` (some examples of function declarations are already there).

To implement a routing protocol function declared in the RoutingProtocolImpl class, say, hypothetically you have declared a function called `forward_packet()`, you need to use the syntax: `return_type RoutingProtocolImpl::forward_packet(...)` {} in the code file. This is to indicate that the `forward_packet()` function belongs to the intended class. `RoutingProtocolImpl.cc` has some examples.

You can then use regular C syntax to implement the body of your function. You can declare local variables as usual, and access class variables (e.g. `sys`) as if they were C global variables. Finally, as mentioned before, to access the GSR9999 system interfaces, you need to use `sys->` to access the interfaces (i.e. `set_alarm()`, `send()`, `time()`) provided by `sys`.

`RoutingProtocolImpl::RoutingProtocolImpl(Node *n)` {} is a special constructor function, `RoutingProtocolImpl::~RoutingProtocolImpl()` {} is a special deconstructor function. You do not need to modify these functions to implement your routing protocols.

Here is a URL with potentially useful information about C++:

http://www.cplusplus.com/doc/tutorial/