COMP/ELEC 429/556 Project 3: Intra-Domain Routing Protocols for Bisco GSR9999

Assigned: Thu, 12 March

Due: 11:55pm, Thu, 9 April

1 Introduction

You have just joined Bisco Systems, a networking equipment company, as a Design Engineer. Your first project is to design and implement intradomain routing protocols for Bisco's upcoming product, the GSR9999 router.

In particular, you are required to design and implement a variant of the distance vector routing protocol (DV). Senior engineers (those enrolled in COMP/ELEC 556) are also required to design and implement a variant of the link-state routing protocol (LS). Junior engineers (those enrolled in COMP/ELEC 429) are encouraged to design and implement LS for extra credit. The functional specifications of the DV and LS protocols are provided (see Section 4 and 5) and your implementation *must* follow these specifications. You are also given the GSR9999's system interfaces (see Section 2) which you must use to implement the routing protocols.

You will implement your designs in a GSR9999 simulator. The simulator essentially provides the GSR9999 system interfaces as described in Section 2. The simulator is written in C++, so you should use C++ for your implementation. You may use any C++ library. Details on the simulator is given in Section 7.

Note that the routing protocols you are going to implement and the network model in this project are simplified. The DV and LS protocols have only a limited set of features. Every node in the network is assumed to be a router and is identified by a unique ID. No hierarchical addressing is used. Each node has a number of ports, which may or may not be connected to a neighbor router. Figure 1 illustrates a simple network example.

The source code of the simulator and some input/output files for testing purpose are available from `http://www.clear.rice.edu/comp429/projects/project3/project3.tar`. 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

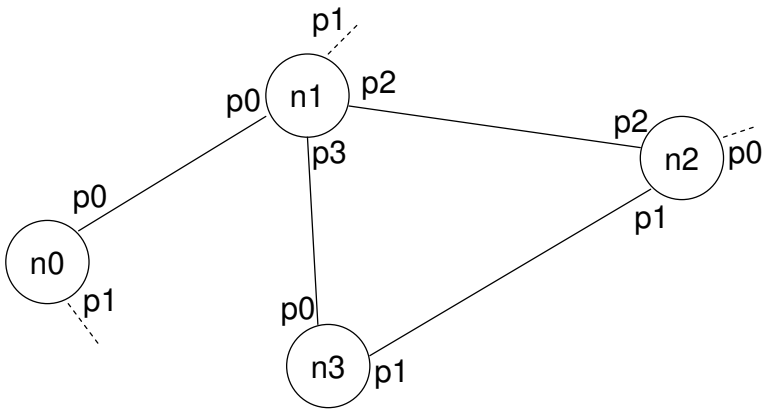

Figure 1: Simple network example.

`RoutingProtocolImpl.[h,cc]` files to implement your routing software, and use the constants already defined in `global.h`. 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 Section 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 also compile with no warning on a CLEAR machine based on the given Makefile.

Grading: Here is how various aspects of your project will contribute to your total grade:

- Conformance to specifications (35 points for DV, 7 points for LS). Correctness of message formats, message contents, message timing, etc.
- Behavioral correctness (35 points for DV, 7 points for LS). Correctness of DV/LS update handling, link status detection, route computation, and packet forwarding, etc.
- Robustness (10 points for DV, 6 points for LS). Able to handle topologies of varying sizes, lack of memory leakage/hogging, simulator crash, etc. To check for memory leakage, if you run the simulator, then run the tool `top`, you should be able to see the memory usage of your simulator process. If the simulator's memory usage keeps increasing, then you almost surely have a memory leak or have implemented something wrong.

- Testing strategy (15 points). You should include a `test.txt` file with your submission to describe your testing strategy (both at the component level and at the system level) and provide test cases that you have developed to convince yourself that your routing protocols are working properly.
- Style (5 points). Lack of compilation warnings on CLEAR based on the given Makefile, appropriate division of functions, appropriate organization of code into files, documentation/commenting of source code, appropriate use of data structures, etc.

Working in groups: You may work in groups of up to three students. If you have trouble finding partners, please contact the instructor.

Submission: Put your source code files, the `test.txt` file, test case files, and a README file that specifies the names of the students in your group and anything else you want the TAs to know about your project under a single directory such as `project3/`. Submit a tar archive file of this directory (e.g. created by running `tar cf project3.tar project3`) to Canvas by 11:55pm on the due date.

2 Bisco GSR9999 System Interfaces

You can assume that the GSR9999 operating system has only a single thread of execution for all running software. That is, a function call to your routing protocol software will be executed without interruption by the system. Everything happens sequentially on a router. This assumption will greatly reduce the complexity of any code you write. An implication of this is that any pending alarm will be set off only after a function call is completed and control has been returned to the operating system. You can assume your functions will take negligible amount of time to execute, so the impact on the timing of scheduled alarm is negligible.

2.1 Initialization

On bootup, the system will initialize your routing software by calling your init() function with the following information:

- Number of ports
- Your router ID
- Routing protocol used (P_DV or P_LS)

Note that ports on each router are numerically identified and can range from 0 to $2^{16} - 2$. $2^{16} - 1$ or 0xffff is treated as a special value as explained below. Router ID can range from 0 to $2^{16} - 1$.

2.2 System Call Interfaces

The following system calls are provided by the GSR9999 system.

- set_alarm(this, duration, *d) - Set an alarm. You specify an amount of time that the system should wait before setting off an alarm. When an alarm is set off, your handle_alarm() function will be invoked (see Section 2.3). d is a pointer to a piece of arbitrary data that is associated with the alarm, it is passed to your handle_alarm() function so that you can use d to figure out what the alarm is meant for. "this" is a reference to the router itself. Multiple alarms can be set.

  When setting an alarm using set_alarm(), you can pass a piece of memory "data" to the system. This data memory should not be freed or modified by your code until the corresponding alarm has been delivered via "handle_alarm()", and the control over the data memory has been returned to you.

- send(p, *pkt, s) - Send a packet pkt of size s bytes out on the port number p. send() in the GSR9999 always completely send the entire packet of s bytes as a single packet. Note that this behavior is unlike the Linux socket send().

  On sending a packet, the memory storing the packet is "owned" by the underlying system, so you should not free or modify a packet's memory after passing it to send(). This also implies that packet memory must be dynamically allocated. On the other hand, on receiving a packet, once your recv() function is called, the packet memory is owned by your routing software, so it is your routing software's duty to free packet memory after a receive if the packet memory is no longer needed.

- time() - Return the current time of the system in milliseconds since bootup.

2.3 Handler Interfaces You Must Implement

- init(num_ports, router_id, protocol_type) - This function is called to pass configuration parameters to your routing protocol, including the total number of ports on this router, the ID of this router, and the protocol type (P_DV or P_LS) to be used. Suppose num_ports is 5, then the ports are numerically identified by IDs 0, 1, 2, 3, 4.

- handle_alarm(*d) - When an alarm is set off, the system will call your handle_alarm() function. Your function should inspect the associated data pointer d to determine the correct course of action.

- recv(p, *pkt, s) - When a packet pkt of size s arrives via port number p, your recv() function is called. Your function should inspect the content of pkt to determine the correct course of action. recv() in the GSR9999 always receive a whole packet as sent by the origin. Note that this behavior is unlike the Linux socket recv().

  When a DATA packet is originating at a router (the DATA packet will be created by the simulator's `xmit` event), it will be received by your RoutingProtocolImpl's recv() function with a special incoming port number of 0xffff. This indicates that the DATA packet is originating locally rather than received from a neighbor.

  When a DATA packet has been received by its destination, the packet memory should be freed.

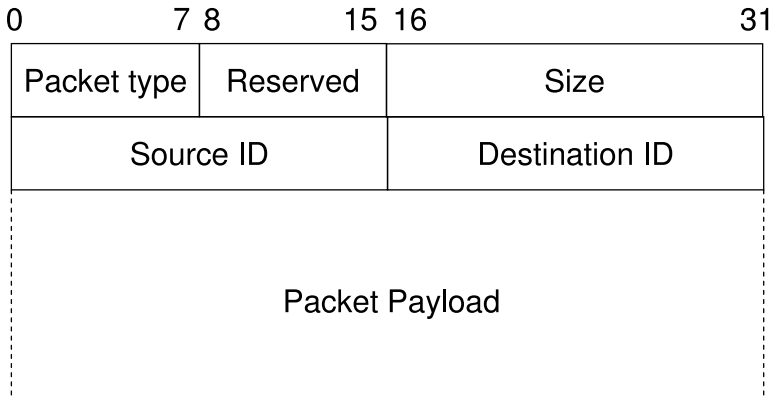

Figure 2: General packet format.

2.4 General Packet Format

Figure 2 illustrates the general packet format that you must use. Network byte order is Big Endian. So you must use byte order conversion functions (e.g. htons() and ntohs()) to transmit all numeric values with the proper byte ordering. Packet type is an 8 bit number corresponding to the 5 packet types defined below. The Reserved section is unused. Size is the size of the entire packet in number of bytes. You may assume that the content needed to be carried by a packet never exceeds $2^{16} - 1$ bytes. Source ID is the node that generated the packet. Destination ID is the node that the packet is destined to.

2.5 Packet Type

There are five packet types in the system:

- DATA - Regular data packet that needs to be forwarded by a router.
- PING - A packet sent only to an immediate neighbor.
- PONG - A packet sent immediately in response to a PING packet to a neighbor. This allows you to detect the existence of a neighbor as well as measure the round-trip delay (i.e. link cost).
- DV - A distance vector routing update packet.
- LS - A link-state routing update packet.

3 Neighbor Status Detection

~~You will use the PING and PONG packet types to periodically check the status of a port and discover whether a neighbor exists and if so the neighbor's ID and the round-trip delay to the neighbor. The checks should be performed on every port once every 10 seconds. PING and PONG packets have a 32bit payload, which stores the time (as returned by `time()`) when the PING packet was sent. A PING packet's destination ID is unused; a router uses the corresponding PONG packet's source ID to discover the ID of its current neighbor.~~

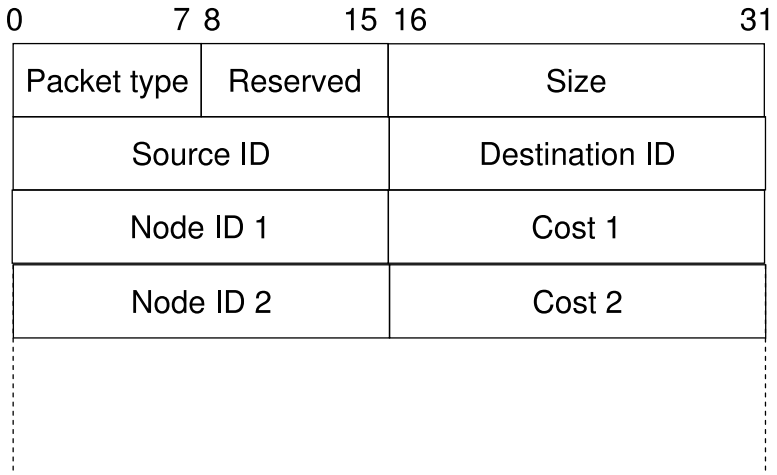

Figure 3: DV update packet format.

~~To determine port status using PING/PONG, you must use an embedded timestamp in the PING/PONG messages and use the following procedures. When a router generates a PING packet, it must store the current time in the PING message payload, then sends the PING message to a neighbor with the correct source ID (the destination ID is unused in the PING packet). When the neighbor router receives the PING message, it must update the received message's type to PONG, copy the source ID to the destination ID, update the source ID to its own, then send the resulting PONG message (with the original timestamp still in the payload) immediately back to the neighbor. When the PONG message is received, the timestamp in the message is compared to the current time to compute the RTT. This is in fact how the `ping` tool measures RTT.~~

Since PING messages are generated every 10 seconds, a port's status is refreshed by PONG messages approximately once every 10 seconds. A link should be declared dead when the status has not been refreshed for 15 seconds. The port status should be properly changed within 1 second of the expiration. That is, if you implement a 1-second periodic check on all the state's freshness, that's sufficient.

4 Specifications of the DV Protocol

4.1 DV Routing Update Packet Format

See Figure 3. You must construct your DV routing update messages according to this format.

4.2 Behavioral Specifications

You must implement the following features:

- You must implement the poison reverse variant of distance vector protocol.
- The cost to reach a node is the round-trip delay (in ms) to reach that node.
- DV periodic updates are sent every 30 seconds.

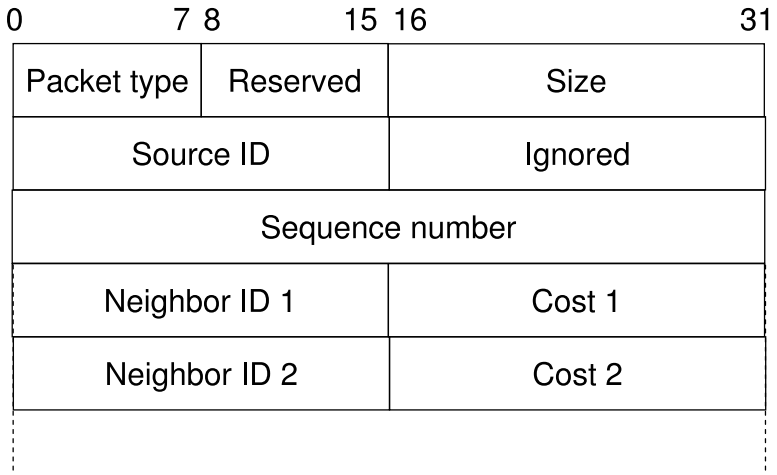

Figure 4: LS update packet format.

- DV entry that is not refreshed within 45 seconds are timed out and removed. You should remove the expired state within 1 second of the expiration time. That is, if you implement a 1-second periodic check on all the state's freshness, that's sufficient.
- DV triggered updates are sent as soon as a local DV change occurs.
- Your DV protocol should update the forwarding table to always reflect the current known best paths.
- For simplicity, your DV implementation should only record the best route among all neighbors for a destination, and not record second best route, third best route etc. That is, when the best route fails, the only way to discover an alternative route is from getting a new DV update message containing an alternate route.

5 Specifications of the LS Protocol

5.1 LS Routing Update Packet Format

See Figure 4. You must construct your LS routing update messages according to this format.

5.2 Behavioral Specifications

You must implement the following features:

- You must implement flooding of LS update packets.
- You must use sequence number to correctly implement flooding of LS update.
- You do not need to implement reliable flooding. That is, you do not need to send acknowledgement or retransmission for LS update packets.

- The link cost is the round-trip delay (in ms) of a link.
- LS periodic updates are sent every 30 seconds.
- LS entry that is not refreshed within 45 seconds are timed out. You should remove the expired state within 1 second of the expiration time. That is, if you implement a 1-second periodic check on all the state's freshness, that's sufficient.
- LS triggered updates are sent as soon as a neighbor status change is detected.
- The Dijkstra's algorithm must be used to compute the correct shortest paths based on the most current link-state knowledge.
- Your LS protocol should update the forwarding table to always reflect the most current computed shortest paths.

6 Design Issues

Your design must address the technical issues described below. **Note that this is certainly not meant to be an exhaustive list of issues**. These issues are provided to help stimulate and organize your thoughts.

Common System Design Issues - Below are design issues that are common to both DV and LS routing protocols.

- ~~Router ID - At bootup time, your routing software is initialized with a unique 16-bit identifier of the router. This ID does not change over time. So your design needs to remember this system configuration.~~
- Routing Protocol Selection - At bootup time, your routing software is initialized with the routing protocol that should be used in the system (either P_DV or P_LS). Once initialized, this configuration does not change over time. So your design needs to remember this system configuration.
- Port Status Data Structure - ~~At bootup time, your routing software is initialized with the total number of ports on the router. The total number of ports does not change over time. A port *may* be connected to a neighbor router separated by a non-zero, positive round-trip delay, or may be unconnected. A neighbor router is uniquely identified by its 16-bit identifier, the round-trip delay to a neighbor router is used as the link cost in your routing protocols. Therefore, you need a suitable data structure for maintaining such port status information.~~
- Port Status Monitoring - In the GSR9999 router, port status information (i.e. neighbor router ID and link round-trip delay, if connected) must be detected dynamically by software. You need a port status detection algorithm that attempts to test port status once every 10 seconds and updates the port status data structure accordingly. You need to use the PING and PONG packet types and the time() and set_alarm() system calls specified in Section 2 to periodically learn the neighbor router ID and measure the round-trip delay. ~~You need to decide (1) how to generate PING packets periodically, (2) how to construct PING packets,~~ (3) how PING packets are processed when received, (4) how PONG packets are processed when received.

- Forwarding Table Data Structure - You need a suitable data structure for maintaining the forwarding table. Note that at bootup time, a router in the network does not know exactly how many routers are in the network. The number of routers in the network can also change dynamically. Your data structure must support insertion and deletion of forwarding table entries. This forwarding table structure should be generic such that it can be used by your packet forwarding function no matter which routing protocols (DV or LS) is being used.

- Packet Forwarding - You need to decide how to handle the forwarding of regular DATA packets.

DV Protocol Design Issues - See Section 4 for the specification of the protocol you need to implement. Below are a set of issues you must address.

- Distance Vector Data Structure - You need a suitable data structure for maintaining the distance vector at a router. Note that an entry in the distance vector must be refreshed periodically or else it must be removed after a timeout. Your data structure should support dynamic insertion and deletion.

- Distance Vector Entry Freshness Check - You need to decide how you will implement a periodic check of the freshness of the distance vector entries.

- Direct Neighbors Maintenance - You need to decide how to insert, delete, and refresh direct neighbors of a router in the distance vector structure.

- DV Announcement - You need to decide how to send DV routing update packets.

- DV Routing Update Packet Construction - You need to decide how a DV routing update packet can be constructed. Your protocol should implement the poison reverse DV variant.

  Link cost (i.e. round-trip time) should be represented in unit of milliseconds in routing update messages as an `unsigned short`.

  The DV update packet should contain as few entries as possible. That is, if a destination D is not reachable from router N, then router N's DV update packets should not include an entry for D. As a result, the size of the DV updates is minimized, and the withdrawal of a route is implicit. In other words, you should not have (ID, INFINITY_COST) pairs in your DV update packets except when it's for implementing poison reverse.

- DV Packet processing - You need to decide how to process DV routing update packets and update the distance vector data structure.

  For simplicity, your DV implementation should only record the best route among all neighbors for a destination, and not record second best route, third best route etc. That is, when the best route fails, the only way to discover an alternative route is from getting a new DV update message containing an alternate route.

- Forwarding Table Maintenance - You need to decide how to update the forwarding table based on information in the distance vector data structure.

- DV update packets are generated periodically every 30 seconds. Triggered updates should be considered as additional updates separate from the periodic updates. That is, triggered updates should not affect the schedule of the normal 30 second periodic updates. For example, suppose periodic updates are sent at time 0, 30, 60, 90, 120... etc. Even if a triggered update occurs at time 42, the regular updates should still occur at time 60, 90, 120, etc and not be interrupted.

LS Protocol Design Issues - See Section 5 for the specification of the protocol you need to implement. Below are a set of issues you must address.

- Link-state data structure - You need a suitable data structure for storing link-state information collected from other routers in the network. Note that link-state information must be periodically refreshed or else it is removed after a timeout. Your data structure should support dynamic insertion and deletion. This link-state information will also be used by the Dijkstra's algorithm for computing shortest paths.
- Link-state Entry Freshness Check - You need to decide how you will implement a periodic check of the freshness of the link-state entries.
- LS Announcement - You need to decide how to generate LS routing update packets.
- LS routing Update Packet Construction - You need to decide how to construct a LS routing update packet.

  Link cost (i.e. round-trip time) should be represented in unit of milliseconds in routing update messages as an `unsigned short`.

  The LS update packet should also be as small as possible. You should never have (ID, INFINITY_COST) pairs in your LS update packets.
- LS Packet processing - You need to decide how to process LS routing update packets and update the link-state data structure.
- Forwarding Table Maintenance - You need to decide how to update the forwarding table based on information in the link-state data structure and the Dijkstra's shortest path algorithm.
- LS update packets are generated periodically every 30 seconds. Triggered updates should be considered as additional updates separate from the periodic updates. That is, triggered updates should not affect the schedule of the normal 30 second periodic updates. For example, suppose periodic updates are sent at time 0, 30, 60, 90, 120... etc. Even if a triggered update occurs at time 42, the regular updates should still occur at time 60, 90, 120, etc and not be interrupted.

7 The Simulator

Get the simulator at `http://www.clear.rice.edu/comp429/projects/project3/project3.tar`. After you compile the code using the provided `Makefile`, you will have the executable `Simulator`, with the `RoutingProtocolImpl` object 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 7.1. The `DV|LS` switch is used to indicate whether the simulator should use the DV protocol or the LS protocol.

7.1 Configuration File

See the configuration files `simpletest{1,2,3}` include in the code as examples. 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 5 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.
- changedelay. `850.00 changedelay (3,4)0.080` means that at time 850 seconds, the link between router ID 3 and 4 will have a new delay of 0.08 second.
- end. `1000.00 end` means that the simulation should terminate at time 1000 seconds. If this event is not specified, the simulator can run 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. You will need to create additional configuration files of your own to help test your routing protocols. In particular, none of the providing configuration files exercises the `changedelay` event.

7.2 Simulator Output

The simulator prints outputs to the screen. You may redirect such output to a file by using the > shell operator. These outputs provide details of the running of your routing software 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.

- 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 received 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.
- Event_Change_Delay. `time = 850 Event_Change_Delay (3,4)` means that at time 850 seconds, the delay on link (3,4) changed.

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.

7.3 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 Linux to do this project. For example, you will most likely need to use `malloc()`, `calloc()`, `htons()`, `ntohs()`, etc.

8 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 instructor know.

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.

You should include a `test.txt` file with your project submission to explain your testing strategy and the test cases that you have developed.

9 C++ Issues

The simulator is written in C++. You may use any C++ library. If you do not know anything about C++, do not panic. The C++ knowledge you need to complete this project is extremely minimal. You can do most of the coding as if you are programming in C as long as you follow a few syntactic rules.

If you are already a C++ expert, you may ignore the following discussion and use any C++ features you want.

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 simulator. What this means is that you must not define and use any global variables in your routing protocol implementation. This is because if you have global variables, the global variables will be shared, accessed, and modified by all copies of `RoutingProtocolImpl`, leading to terrible outcomes.

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 `RoutingProtocolImpl.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/