CS 1652: Project 3

Overview

In this project, you will implement **link state** and **distance vector routing protocols** in the context of a simple *overlay network*. An overlay network is a logical network that runs on top of an underlying physical network. Overlay networks can be useful for applications that need services that are not well supported on the native internet, since they allow implementing custom routing and packet recovery protocols[1]. In this project, your "routers" will be normal hosts that communicate with each other using UDP. Clients will be able to connect to your routers (overlay nodes) and use them to send data to each other.

The overlay client program is already implemented for you. The client-management parts of the overlay node are also implemented for you. Your job is to implement the routing and forwarding logic within the overlay node.

Environment

The overlay network can be run as a real overlay network on GENI, where each overlay node is run on a GENI VM in a different site with a different public IP address. It can also be run in emulation in a single host (GENI or local VM) where each overlay node runs on the same machine and nodes are distinguished based on the ports they receive messages on.

The overlay *topology* (set of nodes and logical edges) is specified in a configuration file. For initial testing, we are providing the following **simple_overlay.conf** file, that specifies a simple topology with four nodes running on the same machine:

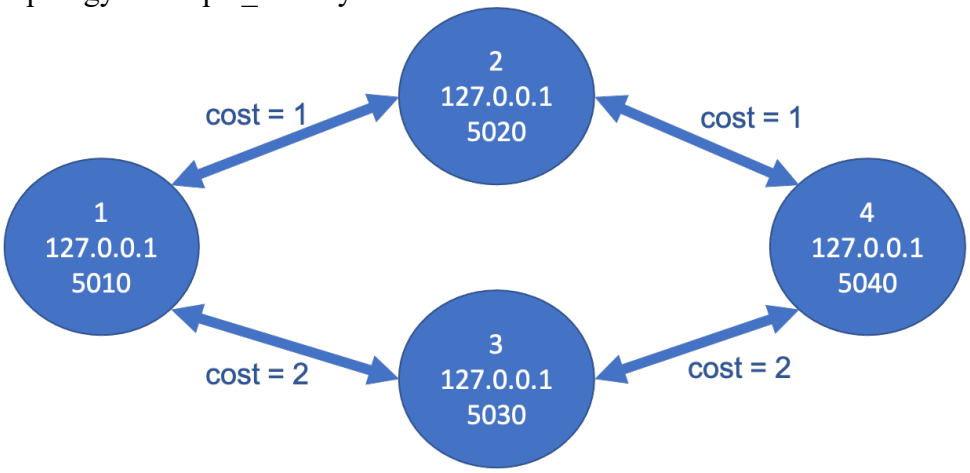
```
1 127.0.0.1   5010
2 127.0.0.1   5020
3 127.0.0.1   5030
4 127.0.0.1   5040

1 2 1
1 3 2
2 1 1
2 4 1
3 1 2
3 4 2
4 2 1
4 3 2
```

The first part of the configuration file specifies the list of nodes, where each line has the form:
`node_ID  IP_addr  port`

The second part of the configuration file specifies the list of (logical) edges, where each line has the form:
`source_node_ID  destination_node_ID  cost`

[1] For this project, we are mainly using an overlay setting to simplify implementation (you can write normal socket programs instead of needing to program routers). But, if you're interested in the overlay concept, you can find more information here: https://www.cnds.jhu.edu/papers/structured_overlays_icdcs2017_vision.pdf

Therefore, the topology in simple_overlay.conf looks like this:



An overlay node is run with the command:
`./bin/overlay_node <ID> <conf_filename> <routing_mode>`

`ID` refers to the node's ID (e.g. a number from 1-4 in our example)
`conf_filename` refers to the name of the configuration file (e.g. simple_overlay.conf in our example)
`routing_mode` should be either LS for link state routing or DV for distance vector routing.

When starting up an overlay network, you should run one instance of the overlay_node program for each node in the topology. For example, to run the topology above with link state routing, you would run (each command in a different terminal window):

```
./bin/overlay_node 1 simple_overlay.conf LS
./bin/overlay_node 2 simple_overlay.conf LS
./bin/overlay_node 3 simple_overlay.conf LS
./bin/overlay_node 4 simple_overlay.conf LS
```

Once the overlay network is running, overlay *clients* can connect to the network to send messages to each other. The overlay client has the following interface:

```
./bin/overlay_client <overlay_node_IP> <overlay_node_port>
                     <client_data_port> <destination_overlay_node_ID>
                     <destination_client_port>
```

`overlay_node_IP` refers to the IP address of the overlay node that this client wants to connect to
`overlay_node_port` refers to the port used by the overlay node that this client wants to connect to
`client_data_port` is a port chosen by the client that it will use to send data to the overlay node. This port is used to distinguish different clients that connect to the same overlay node.
`destination_overlay_node_ID` refers to the ID of the overlay node that the target destination client is connected to
`destination_client__port` refers to the `client_data_port` that the target destination client is using

For example, if we want to run one client that connects to overlay node 1 (with client port 7777) and communicates with another client connected to overlay node 4 (with client port 8888), we could run:
First client: `./bin/overlay_client 127.0.0.1 5010 7777 4 8888`
Second client: `./bin/overlay_client 127.0.0.1 5040 8888 1 7777`

Note that the client data ports 7777 and 8888 are arbitrary, you can use whatever ports you want, as long as they don't conflict with ports that are already in use.

The overlay client program reads text from stdin and attempts to send it to the specified target destination client through the overlay. So, once your clients are running, you can type in the terminal at each client to try to send messages between them.

But, sending messages between node 1 and 4 won't work yet! For this to work, you need to implement a routing protocol…

As a simple test to check your setup, you can run two clients that are both connected to the same overlay node. In this case, there is no need for a routing protocol, since the overlay node can just deliver the message locally. For example, to run two clients connected to overlay node 1:
First client: `./bin/overlay_client 127.0.0.1 5010 7777 1 8888`
Second client: `./bin/overlay_client 127.0.0.1 5010 8888 1 7777`

In this case, you should be able to type in each client's terminal window and see that they get each other's messages.

Routing and Forwarding

Your job in this project is to implement the overlay node's routing and forwarding logic.

Recall from lecture that *forwarding* refers to processing an incoming data packet, determining the next hop to send it to (via forwarding table lookup), and sending it on that next hop.

Routing involves exchanging control messages to learn about the network topology, and computing forwarding tables based on that information. As part of the routing protocol, a node must also monitor its own links and send updates if their status changes (e.g. if the link fails).

The specific routing protocol the overlay node uses should be determined by the *route_mode* option specified on the command line. If this option is LS, the overlay node should use a link state algorithm. If this option is DV, the overlay node should use a distance vector algorithm.

Link Monitoring: For both the link state and distance vector algorithms, each overlay node must run a link monitoring protocol to detect failure (or recovery) of its local links. To do this, each overlay node should send periodic *heartbeat* messages on each of its local links (e.g. once per second). Upon receiving a heartbeat message, a neighbor should reply with a *heartbeat_echo* message to indicate that it is up and reachable. Note that we are using UDP, so it is possible for any individual heartbeat message to be lost without necessarily indicating a link failure. However, if a node does not receive any heartbeat_echo on a link for 10 seconds, it should declare the link to be failed (and send a routing update that reflects that).

Route Computation: Upon receiving a routing update (link state advertisement or distance vector update), the overlay node should run its route computation algorithm, update its forwarding table (if needed), and send further updates as needed (i.e. flood the link state update, or send further distance vector updates, according to the algorithm).

Implementation Details

On startup, each overlay node creates 2 UDP sockets: a *control* socket and a *data* socket.
The control socket is to be used for sending messages related to the routing protocol, while the data socket is used for forwarding data.

The data socket is bound to the port specified in the configuration file, and the control socket is bound to that port + 1 (note that this means that if you are running multiple overlay nodes on the same machine, you need to make sure their data ports are separated by at least 2 ports. The simple_overlay.conf file handles this -- it assigns ports that are spaced by 10 ports for simplicity).

The overlay node uses an *event handling system* to handle socket and timeout events (this system is borrowed from the Spread Toolkit: www.spread.org). The function **E_attach_fd()** is used register a socket (file descriptor) and callback function with the event system, such that the callback function will get triggered when data arrives on that socket.
The function **E_queue()** is used to schedule a timeout event, such that the specified callback function will be triggered after a specified delay.

On startup, the overlay node registers the callback function handle_overlay_ctrl() on its control socket and registers the callback function handle_overlay_data() on its data socket.

The **handle_overlay_ctrl()** function is where you **should implement your link monitoring and routing logic**. Upon receiving a control message (heartbeat, heartbeat_echo, link state advertisement, distance vector), you should check the type of the message and respond appropriately:

- Upon heartbeat: send heartbeat_echo to sender
- Upon heartbeat_echo: if link was previously dead, send update indicating that it is up; push forward timer for declaring link dead
- Upon link state advertisement: if there is a change in the link weight, re-run route computation and update forwarding table; flood to neighbors
- Upon distance vector update: if there is a change, re-run route computation and update forwarding table; if my distance vector changed, send to neighbors

A basic implementation that checks the packet type and calls an appropriate function based on the type is provided. You will need to implement the logic for each of these packet handling functions (handle_heartbeat, handle_heartbeat_echo, handle_lsa, handle_dv).

The **handle_overlay_data()** function is called when a data packet arrives. There are two possibilities for an arriving packet:

1. It is destined for a client connected to this overlay node. In this case, the overlay node will call deliver_locally() to send to its connected client. This is already implemented for you.
2. It is destined for a client connected to a different overlay node. In this case, the overlay node calls the function **forward_data()**. **You need to implement this.** In this function, the node should look up the destination overlay node id (dst_id field in the data_pkt_hdr) in its forwarding table and then send on the next hop for that destination. For testing purposes, the overlay node should also **stamp its ID in the "path" field of the packet header and increment the path_len in the header** before forwarding it. This will allow you to see the path that packets are taking through the network (the overlay_client prints the path for each message it receives).
    a. Note: you may find it simplest to use the Node_List initialized from the configuration file as your forwarding table.

Of course, to start the whole process, the overlay needs to send its initial link state / distance vector information on startup, and schedule periodic events (e.g. heartbeats, periodic link state / distance vector updates). This logic should be implemented in the functions **init_link_state()** and **init_distance_vector()**, which are called on startup (based on the routing mode).

Note that packet types are already partially defined for you in **packets.h** you should review these and extend the control packet types as indicated in the comments.
Important Note: Sending structs across the network like this is NOT 100% safe. As you saw in project 2, byte order can vary across architectures, so protocols typically convert all fields to network byte order. For this project you can ignore this and assume all your hosts use the same architecture.

It is ok to refer to existing implementations of Dijkstra's algorithm to implement your link state route computation, but you must cite any external sources you use (in both your README and in source code comments). Note that your textbook provides pseudocode for both distance vector and link state routing.

We provide basic Node_List and Edge_List data structures to store the initial representation of the graph read from the configuration file. You should feel free to translate these to other data structures if it simplifies your route computation algorithms.

Note: Both the overlay_node.c and overlay_client.c include a large number of debugging print statements that may be helpful in understanding the program flow and debugging your work. To turn these on, you can edit the source files to change #define PRINT_DEBUG 0 to #define PRINT_DEBUG 1 When this option is set, all lines with Alarm(DEBUG, …) will be printed. Otherwise only Alarm(PRINT, …) and Alarm(EXIT, …) lines will be printed. Note that Alarm(EXIT, …) also exits the program, so this should only be used for fatal errors. The Alarm() function comes from the same libspread-util library as the event handling system.

Testing

Before submission, you should ensure that you can:

1. Send messages between two overlay clients connected to the same overlay node (this should already work in the code you are given, but make sure nothing broke)
2. Send messages between two overlay clients connected to different overlay nodes. Make sure this works (i.e. the messages are received and printed correctly):
    a. Using Link State routing
    b. Using Distance Vector routing
    c. If the two overlay nodes are neighbors (have direct link between them)
    d. If the two overlay nodes are not neighbors
    e. With the simple 4-node overlay topology
    f. With more complex overlay topologies
3. Check that the paths being used are correct (i.e. are actually shortest paths), based on the path information stamped in the packet header and printed in the overlay client.
4. Check that you can re-route after failures:
    a. Kill (ctrl+c) an overlay node on the active path between your clients. The neighbors of that node should detect this, and send link updates. The routing should converge to a new shortest path and you should be able to send messages between the clients.
5. Check that you can re-route after node/link recovery:
    a. Bring the node you killed back up. It should begin responding to heartbeats and sending routing updates again, causing the protocols to re-route and include it again.

Extra Credit: Dynamic Link monitoring

For extra credit, you can extend your heartbeat protocol to not only monitor whether or not a link is up, but what its roundtrip latency is. Then, instead of using the default link costs from the configuration file, you can use the roundtrip latency values as the link costs to find shortest paths in terms of latency. To test this, you should use GENI nodes located in different sites so that there will actually be different propagation delays on different links. Include the overlay configuration file you used for testing and description of your approach and testing (including which site you used for each node) in a README that you submit with your project. You can also look into the netem utility for artificially increasing the latency between node (https://openwrt.org/docs/guide-user/network/traffic-shaping/sch_netem). As an extra challenge, consider extending your monitoring protocol to calculate the loss rate on the link and implement a link cost metric that penalizes lossy links.

Extra Credit: Bug Bounties

Similar to the Petnet stack, the overlay infrastructure for this project was developed this semester, in a limited amount of time. You can submit bug reports (identifying the source and proposing a fix) and/or improvements for extra credit. The number of points will depend on the severity/complexity of the bug/improvement.

Logistics

- You should ideally work in teams of exactly two students. Working alone is also permitted. Please let me know if you plan to change teams from project 2.
- You are encouraged to use version control to collaborate with your partner on the same codebase. Using Github for this purpose is encouraged, but your repositories must be PRIVATE and shared only with your partner
- Your code must be written in C and must compile with the default versions of gcc shipped with Ubuntu LTS 18. We will expect that running "make" will generate the executables `overlay_node`, and `overlay_client` (note that you should not need to modify the overlay_client) according to our Makefile.
- You should submit a .zip or .tar file that includes everything needed to compile and run your program (including the files we provided). We will expect to be able to unzip/untar your file, run make, and then run your programs.
- Assignments will be submitted in Canvas