Distance Vector Routing
Project 2 - Due Friday, November 17
CSE 461, Autumn 2000

1 Overview

Project 2 requires your group to implement a simple distance vector routing protocol. Your implementation of the Fishnet Routing Protocol (FRP) will send and receive routing update messages describing network reachability according to the protocol description below.

There are six pass/fail tests which your implementation will need to pass. These tests will be used to determine your grade for the project, equally. Each test will demonstrate whether your implementation establishes reachability between two nodes in a short time.

You will be expected to implement the bodies of the stubs contained in `frp.c`. Modify *only* this file, as it will be the only one turned in. If you need additional structures or functions, declare them in `frp.c`.

This document describes the protocol and interfaces for relevant parts of the network framework. A small glossary is included at the end.

2 Steps

These steps are included as a guide. For example, although broken into separate steps, you may choose to implement fragmentation at the same time as periodic updates, rather than afterward. The list of functions to modify is not an exhaustive list: you may have to modify others we do not mention here.

1. Download the distribution from the course web page
2. Un-tar the distribution, using `tar xvzf`.
3. `cd` to the distribution directory
4. Run `./configure`
5. Run `make`
6. If you have problems at this point, look over the README for required software, and if that doesn't help, mail the instructor.
7. Run `./reference basic.tst`
8. Run `./reference` again for each of the other tests (`.tst` files), each should show a graphical animation, then print `success` to the console. If you don't see the animation, and are logged in from a windows machine, use ReflectionX. The animation will not work particularly well over a modem - there are ways to disable it described in Section 7.1.1.

9. Read the rest of this document, consulting `fishnet.h`, and the unimplemented stubs in `frp.c`.

10. Fill in `struct frp_packet`, using the existing `fishnet_header` structure.

11. Create a structure for each advertisement.

12. Make periodic updates work, filling in `periodic_update()` , `frp_input()`, `frp_output()`, and `traverse_tree()`

13. Make triggered updates work, filling in `triggered_update()`, and modifying `route_timeout()` and `frp_input()` (to generate triggered updates when information changes).

14. Implement split-horizon by modifying `triggered_update()` and `periodic_update()`.

15. Make fragmentation work by modifying `traverse_tree()` and `periodic_update()`.

16. Implement checksum calculation, consulting the book's implementation as necessary.

17. Have `frp_input()` verify the checksum.

18. Verify that your program passes all tests. `make tests` was designed to run all tests, one after the other. `make noxtests` will run the tests, without the graphical display.

3 Fishnet Simulator Overview

Relevant system structures are defined in `fishnet.h`. Each `node` has `routes`, making up a forwarding table in radix trie form, and `ifaces`, the interfaces used for input and output.

Routes are stored as a tree. It is not important that you be comfortable with the ordering of this tree, only that `on` and `off` are the child pointers, since you will traverse this tree to generate routing updates.

Interfaces have a local address, `addr`, a Maximum Transfer Unit, `mtu`, and a reference to the node, `local_node`, the interface belongs to.

4 Protocol Operation

This section describes the basic operation of the protocol you will be implementing. It is very similar to the RIP protocol described in RFC1058, and most recently RFC2453. These documents can be used as a reference when further explanation of protocol processing is required.

4.1 Output Processing

This section describes the generation of updates by a node. Updates fall into two categories: the periodic distribution of the entire routing state of a node, and triggered updates which send an update message concerning the change of an individual route.

4.1.1 Periodic Updates

Each node in the network sends out periodic updates that encode all the routes in the routing table, except those omitted by split horizon as described below.

Periodic updates should encode the entire state of the routing table as seen by that node. There is one independent timer running for each interface of the node.

You should fill in the stub:

```
static void periodic_update(iface i)
```

so that it properly constructs a full periodic update and sends it out the output interface. As already included in the stub, the periodic update function should schedule itself to send a new periodic advertisement after the interval has expired. The initial timeout is scheduled by the `start_frp()` function in `frp.c`.

To traverse the entire routing table, you should fill in the `traverse_tree()` stub that has been left in the distribution. The radix trie is used to store the routes in a manner which allows the longest match to be determined easily. Each node in the radix trie may have a right and left child. `traverse_tree()` simply walks through all the routing table entries for a given node, stopping at each, and recursing on the left and right elements. If it is called with a null route entry at any point it exits that part of the traversal. You should enhance `traverse_tree()` to carry enough state to encode the route entries into the update packet.

4.1.2 Triggered Updates

Triggered updates are sent in response to a metric change of an existing route. They should be sent when the metric of the existing best route changes, when the output interface changes, and when a route expires due to timeout (see below).

Modify

```
static void triggered_update(iface i,route r)
```

to send an update immediately to the output interface `i`, containing only information about the route `r`. Please use the `route_update_event()` function from `frp.c` to call `triggered_update()`, as it also updates the display.

Ideally, triggered updates would be batched together to stop storms of updates from occurring during topology changes. You will not be expected to batch triggered updates.

Both (`periodic_update()` and `triggered_update()`) functions should use the stub:

```
static void frp_output(iface i,frp_packet p)
```

to send packets, formatted as described in Section 5, into the network. This stub must be augmented to perform whatever output processing is common to both updates, and then call:

```
    node_packet_output(interface i, unsigned char *buffer,
                       unsigned int length, address destination)
```

to send the packet across that interface.

4.1.3 Split Horizon

Your implementation must properly implement split horizon. It is not necessary to implement poison reverse as all the networks in the project are point-to-point.

Simple split horizon means that you should not send out advertisements on the interface which is the current output interface for that particular route. This can be implemented using `is_iface()` (Section 6.2) to test the output interface of a route.

4.2 Input Processing

When an update is received, each route in the update is compared against the current state of the node's routing table. The current routing table is examined using the `find_route()` function.

In the implementation, input is handled by the

```
void frp_input(iface i,unsigned char *buffer,unsigned int length)
```

function, which is present an a stub in `frp.c`. This function should decode the update, and should iterate over the advertisements, and for each:

- If no matching route entry exists in the local routing table, create one using `add_route()`. Set the new route's metric using the assignment operator. Send a triggered update using `route_update_event()`.
- If the metric contained in the advertisement is less than that of the existing route, change the output interface and the metric to the advertised route. A triggered update should be sent to all other peers with the new metric.
- If the advertisement came from the output interface of the current best route, accept the new metric and send a triggered update. If the new metric is infinity, remove the route.
- Otherwise, ignore the advertisement.

4.3 Route Timeouts

At any time, a routing peer or intermediate link might fail. There is no explicit information given to the routing protocol that this has happened, and so a mechanism must exist to remove stale routing information.

We will associate a timer with each route being maintained by a node. This timer should be reset under the following conditions:

- during the initial insertion of the route
- an advertisement concerning that route has been received from the existing output interface for that route (see `is_iface()`, Section 6.2)
- an advertisement from a different interface which has a lower metric than the current route

If the timer expires, the route can be removed from the local node. Send a triggered update with an infinite metric when a route times out, then delete the route.

The timer can be reset using the function `set_route_timer(route r)` function.

4.4 Route Injection

Since the project environment only implements point-to-point links, all routes used during your tests will be host routes (`/32`), and are the addresses assigned to the interfaces of the switches. These routes are created as part of the configuration process and will be in the routing table when you start running. The metric for interface routes local to a node is `0`. Each node also has a routing table entry for the broadcast address defined by convention to be `255.255.255.255`. You may choose to filter these routes although they won't affect the correctness of the protocol.

5 Packet Encoding

The assignment includes the encoding and decoding of update packets into the byte representation sent across the network. We suggest you complete this portion of the project first since it provides functionality which the other components require.

Common practice in C implementations of protocols is to perform such encoding functions by using C structures as a template. This implementation method has a number of problems, but it is simple and a common technique, so we recommend that you use it here.

FRP packets are encoded as the payload of a generic Fishnet packet. They include a small number of fixed fields followed by a variable length set of advertisements. The number of advertisements in the update is variable. It is encoded explicitly by a field in the fixed header, and implicitly in the length of the overall network packet.

```
+-----------------+
|   Fishnet       |
|   header        |
|                 |
+ --------------- +
| FRP fixed header |
+ --------------- +
|                 |
|    adv1         |
+ ----------------+
|   ....          |
```

The Fishnet network level header looks like:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     TTL       |    protocol   |            length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- The protocol number should be set to that assigned to FRP: `3`, or `proto_frp`.
- The source address in the network header should be set to that of the output interface `i->addr`.
- The destination address should be sent to `0xffffffff` which is the Fishnet broadcast address. This is used to reach the other end of the link anonymously, since the FRP protocol entity doesn't know the address of the remote peer.
- The TTL should be set to `1` to prevent inadvertent forwarding of FRP packets.
- The length field in the network header should reflect the total length of the datagram, including the Fishnet header, the FRP header, and the advertisements.

The FRP header includes three fields:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     version   |    entries    |      checksum                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- The version field should be set to 1
- The total number of advertisements included in this update should be placed in the single byte field `entries`
- The checksum should be calculated over the entire body of the packet, starting at the beginning of the FRP header, and including all of the updates. The checksum does not include the Fishnet header.

The checksum algorithm is as follows:

1. Set the checksum field to zero
2. Walk through the packet 16 bits at a time, adding each 16 bit word to a cumulative sum modulo 2^{16}. There should be an integral number of words.
3. Invert the bits of the checksum using the C unary bit inversion (˜) operator.

You may use the checksum algorithm on page 95 of your textbook.
Following the FRP header, an update contains one or more routing advertisements

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         destination                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      prefix   |    metric     |         reserved              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Each advertisement contains a 32 bit destination, an 8 bit field for the prefix length, and an 8 bit field for the metric. Each advertisement also contains a 16 bit pad to maintain word alignment of the entries. Although word alignment isn't strictly necessary, many protocols choose to keep fields word aligned to make implementation easier.

5.1 Byte Ordering

Fishnet uses a different byte ordering than some hosts, including x86. Most network protocols such as IP are specified in big-endian ordering for multibyte fields, where the most significant byte of the word lies in the first address. Little-endian ordering, as used on x86, places the least significant byte first.

```
unsigned int htonl(unsigned int source)
unsigned int ntohl(unsigned int source)
```

Convert a 32 bit word from host and network order and conversely. Both of these functions do the right thing regardless of the byte ordering of the target architecture.

```
unsigned short htons(unsigned short source)
unsigned short ntohs(unsigned short source)
```

Convert a 16 bit word from host and network order and conversely.

5.2 Fragmentation

Each link in the network has a limit on the size of an individual frame. The network protocol itself enforces a limit of 64 kilobytes per packet, but the link MTU is usually much smaller. The MTU for a given interface `i` is stored in `i->mtu`.

The representation of all of the information contained in a periodic routing update might exceed this limit on the output link. The routing protocol implementation will need to cut up (fragment) the updates to fit in the output link MTU. This means determining when no more advertisements can be placed in the packet, sending the current packet, and beginning to encode a new one.

Although some of the packets of an entire set of updates may be lost, the route processing is tolerant of small numbers of failures, and processes the entries independently.

6 Implementation Interfaces

This section describes functions we have implemented on your behalf. Your implementation will be simpler if you use these interfaces. Some function prototypes may have changed since this document was written, so consult `fishnet.h`.

6.1 Timers

You will be required to have some minimal interaction with the timer system. You will only need to use the following two calls to manipulate timers.

The timer system works by having you register a callback. The callback consists of a function pointer and an opaque pointer argument to pass to the function. This function will be called with this single argument approximately when the interval has expired.

Each timeout expires only once, or never if it is explicitly removed.

`timer register_timer(when interval, void (*function)(void *), void *argument)`

register a callback function to be invoked after some `interval` period. This period is expressed in terms of ticks. HZ is a macro you can used to convert between seconds and ticks which contains the number of ticks per second.

This project is scheduled in real time rather than simulated time. A HZ timer will expire in approximately one real second. This can have strange implications when you use the debugger, since all the timers will fire if you continue after a pause.

`void remove_timer(timer t)`

will remove a timer before it has expired. The callback function will never be called.

6.2 Route Table

You will also be required to interact with the route table.

`route add_route(node n,address to, unsigned int to_prefix, void (*f)(), void *a)`

`add_route()` takes a destination address, a prefix length, and callback function to handle the output for packets following that route.

To add a route on node `n` with output interface `i` use:

```
add_route(i->n,destination,prefix,node_packet_output,i);
```

`void set_iface(route r, iface i)`

Sets the route `r` to point out the destination interface `i`.

`boolean is_iface(route r, iface i)`

Checks if some interface `i` is the output interface of a route `r`. Returns TRUE or FALSE.

`route find_route(node n, address destination,int prefix)`

Performs an exact search to find the routing node associated with the same destination or prefix. It returns `0` if no such route exists in the table.

`route resolve_route(node n, address destination)`

Find the longest match route entry for the given `destination`, and return it or `0` if there was no match.

`void remove_route(node n, address destination, unsigned int prefix)`

Remove the route from the routing table which matches the `destination` and the `prefix`, if it exists.

6.3 Tracing

You may find it helpful to examine network traffic in the course of debugging your implementation. To trace all packets on an interface add:

`node:`*node*`:interface:`*interface*`:trace everything`

into your test file, where *node* represents the node name, for example, `a`, and *interface* represents the node's interface, for example, `a->b`.

Short descriptions of each packet passing that interface will then be printed on the console as they occur. You should limit the number of interfaces you run tracing on at once since the output can be large.

7 Test Descriptions

Each test is driven by a series of timers. Since the tests evaluate protocols running in real time, and require distributed convergence after topology changes, they may take some time to complete. Each test will run for a predefined interval (located at the end of the test file), and print success or failure on the console window.

Note that `interop.tst` uses `reference` or `reference.exe` from the distribution to verify the interoperability of your protocol. This executable can also be used to evaluate the test cases and watch the behavior of the reference code (for example `reference basic.tst`).

7.1 Graphical Output

Each test should bring up a window that displays the network topology as nodes and edges.

Each node contains a label consisting of the node identifier, and possibly a number in parenthesis. This number is the metric associated with the best route being used by this node to get to the destination.

red edge a link which has been taken down

grey edge a running link not used by any node to get to the destination

blue edge a running link used by some node to get to the destination. Each blue edge will have at least one arrow showing the direction it is being used.

arrow This edge is being used along that direction. A edge involved in a two-node loop will have two arrows.

7.1.1 Turning off the Display

At some point during your debugging it may be more useful not to have the animated display window pop up during the run. Removing all lines beginning with `topology` from the test configuration file will stop the window from being created. Since the simulator runs in real time, disabling the display may affect the behavior of your program. We will verify your implementation with the display disabled using `make noxtests`.

7.2 basic.tst

`basic.tst` consists of a network of threes nodes. The test verifies that full connectivity is established among the three nodes.

The drawing (on the graphical display) shows reachability to `c`.

`basic.tst` exits after 5 seconds.

7.3 trigger.tst

`trigger.tst` uses the same topology as `basic.tst`, but after some interval has expired, the link between `a` and `b` is removed. `b` should timeout its route to `a`, and send a triggered update to `c`.

The destination node in the drawing is `a`.

`trigger.tst` exits after 16 seconds.

7.4 mtu.tst

`mtu.tst` involves a larger topology, and a dynamic route change along a downed link. After adequate time has passed to allow the routing protocol to re-converge, a ping packet is sent between `g` and `b`.

One of the links in `mtu.tst` (the link between `c` and `g`) has a small MTU, only large enough to fit updates containing only single advertisements.

`mtu.tst` exits after 16 seconds.

7.5 infinity.tst

`infinity.tst` involves a large fairly richly connected topology. The display shows reachability to the node `i` in the lower right hand corner. lower Connectivity will be tested from `a` to `i`, then `i` will be disconnected from the cloud. After 17 seconds another probe will be sent which insures that routes from `i` have been flushed. Connectivity is the reestablished and tested again.

`infinity.tst` takes 35 seconds to run.

7.6 circle.tst

`circle.tst` tests the ability to resolve the longer path (the outer circle) after the shortest path has failed. It tests connectivity once before and once after the link failure, between nodes `b` and `d`.

It takes 21 seconds to run.

7.7 interop.tst

Test `interop.tst` is qualitatively different from the other tests. It tests the ability of your FRP implementation to exchange routes with the reference implementation, which is shipped as part of your distribution. If you are not using Windows or Linux, stop here and send me mail about how to get access to a reference implementation.

The test simply waits a suitable amount of time (10 seconds) for the network including both implementations to converge, and verifies reachability by sending a ping message across it.

The destination being shown by the drawing is in the reference implementation, and is thus not on-screen. You should however see arrows pointing towards this off-screen destination once it has converged, with metrics larger than 3.

This tests generally fails when your checksum algorithm, or byte ordering is not correct.

8 Glossary

adjacency The peer-to-peer protocol relationship. Usually involves an explicit session between peers.

advertisement Protocol information which contains reachability information for a destination network from the perspective of the source node. An advertisement has a destination, a prefix, and a metric.

converge the process of obtaining a stable state where all nodes have the shortest paths after some change in topology

infinity A finite upper bound on the metric space. No routes are advertised with metrics greater than infinity. A route advertised with an infinite metric is considered to be unreachable from that node.

inject To originate a route.

MTU *maximum transmission unit* The maximum number of bytes in a frame which can be sent on a particular link.

metric An integer measure of the relative goodness of a route, often measured in number of forwarding hops. Paths with smaller metrics are more preferable.

peer The remote protocol entity with which you are exchanging routing information. More generally any remote protocol entity with which a node might be having a conversation.

point-to-point A link type in which there are only two interfaces.

periodic update An update which is sent at regular intervals including all of the routes known by the source. one or zero peers.

triggered update An update which is sent in direct response to a route change (asynchronous), usually including only the route in question.

update A single network message consisting of some fixed information and one or more advertisements.