

Fishnet Assignment 1: Distance Vector Routing

Due: May 13, 2002.

In this assignment, you will work in teams of one to four (try to form your own group; if you can't find one, you can either work alone, or contact John-Paul and he'll group you with other unassigned people) to implement a Distance Vector Routing protocol. We provide an example program, `random.c`, that forwards echo request and echo reply packets across the Fishnet network using random forwarding (that is, each fishnet node sends a packet in a random direction until it eventually reaches its destination). Your job is to extend this capability to use routing tables instead of random forwarding and to automatically create and maintain these routing tables. The goal of this assignment is for you to understand distance vector routing in a significant way.

1 Introduction to the Fishnet

Fishnet is a class-sized network that we will build over the remainder of the quarter. You will build a Fishnet node progressively, working in a group, over a series of assignments. Once we are done, nodes from each team will be able to communicate with one another in a shared network.

1.1 Development Environment

We will be developing under Solaris (Sun's flavor of Unix) and writing C code. It is assumed that you will use a window system (such as X windows) to allow multiple processes to be visible at once. Each node of the Fishnet network will run as a separate process on a Unix workstation (consequently, a single workstation can support multiple nodes by spawning multiple processes). The compiler we will use is `gcc`, and the debugger we will use is `gdb`. You may use other tools, but we won't support them and we will grade your assignments using `gcc` and `gdb` (i.e. it's your problem if your software doesn't compile using `gcc`, not ours). You can use any editor you prefer, such as `emacs`.

1.2 What is a Fishnet?

The term "Fishnet" is somewhat overloaded, and before going further we want to clarify what it means. First, it is the code that makes up all of the assignments. This is the *Fishnet development environment*. Second, it is a running network of many nodes and a single fishhead (a key component to be described next). This is what we most commonly mean by a *Fishnet*. Note, however, that there are many potential Fishnets. You will all start your own, private, Fishnet to develop and test the code required for this assignment. Each of you will run your own fishhead, and nodes of your network will not interact with nodes of other students' networks. Finally, there is The Fishnet, with a capital "T". This is the one public, shared Fishnet (a running network) we will create over the quarter that everyone's node is able to join and participate in a collective class network. This is accomplished simply by pointing your code at a fishhead that we run.

1.3 Fishnet Components

The Fishnet project code, like everything else you need, is available from the class directory (~cs123b). The package we provide you with contains the following key components:

fishhead

fishhead is a program that manages Fishnet nodes. A network can contain many nodes, but only one fishhead. The main function of the fishhead is to tell individual nodes who their neighbors are. It is important to understand that the fishhead manages the topology and decides who is connected to whom, not you in your programs. (For the curious, the nodes of networks you operate are run as separate processes that communicate with each other using a UDP overlay.) When you develop a Fishnet node and run it, one of the first things it will do is to join a Fishnet network by contacting the fishhead. This means that before you start one of your Fishnet nodes there must be a fishhead process running. You only need to start a fishhead for your network once, even though nodes of the network can come and go.

libfish.a

The fish library implements all of the Fishnet functionality that you will need for your assignments. When you send a message using the `fish_sendpacket()` function, for example, `libfish.a` is called to do the work of sending the packet. `libfish.a` also prints a large (but controllable) amount of debugging information to the console (`stderr`) to help you understand what is going on with your program. The library source code is available in the `fishsrc` directory so that you can see how the Fishnet really works and to help with your debugging, but you **MUST NOT** change this code at all so that you remain compatible with other people's nodes.

fish.h

This is the header file that you should include in your C program to gain access to the functionality implemented in the fish library. It contains the dozen or so Fishnet API functions that you can call, as well as the structures that define packet formats, and other Fishnet constants. You should read the comments in this file, as it is the definitive Fishnet API documentation rather than a separate manual.

2. What You Need To Write

You will write a C program in a single file called `hw1.c` that does the following:

- Takes three whitespace separated command line arguments: first, the hostname on which you have started a fishhead running; second, the port number on which you started the fishhead; and last the address you want for your node in the Fishnet, which is simply a small number. This is already implemented in the example.
- Joins the Fishnet described by the command line arguments with the specified address. You will need to use the `fish_joinnetwork()` call, and you will need to have started your own fishhead before running your program. Remember, you choose the address of your node, but the fishhead decides what other nodes you

are connected to, and your neighbors change as other nodes come and go. This is also already implemented in the example.

- Waits to get a line of input from the keyboard of the form "send <nnn> <message>", then sends an echo request packet across the Fishnet to the destination, from where an echo reply packet is sent back to and received at the original sender. This is already implemented in the example, as is the basic code for forwarding and receiving.
- Maintains a routing table which is an array of MAXADVERTISEMENTS routing table entries. You must define the structure of a routing table entry. It should have several types of information: a destination address, the best distance to that address, the preferred neighbor, and the age of the information.
- Performs a periodic update every 10 seconds. (Use the timer interface described in fish.h.) As part of the periodic update, call:
`fish_debug(FISH_DEBUG_ROUTING, "Periodic update\n")` to print a debugging message. Then age the information in the routing table and send routing updates to all neighbors, as described below.
- To age the routing table, you should remove any entries that have not been refreshed (as described further down) for the past three consecutive timer intervals.
- Then, use the `routing_packet` structure defined in fish.h to send a routing update to all neighbors. The value of the packet header's protocol field should be set to `FISH_PROTOCOL_ROUTING`; the destination should be `ALL_NEIGHBORS`. The packet should include one advertisement for the address of the local node at a distance metric of zero, plus one advertisement with address and distance for each entry in the routing table. (These entries have been learned from neighboring nodes as described below.) Note that the size of a routing packet is `PACKET_HEADER_SIZE + num_adv*sizeof(struct route_advertisement)`.
- Each node receives routing update messages from neighboring nodes and uses them to update and refresh the contents of the routing table as follows. Routing advertisements carried in the routing message are processed in turn as described below, where each advertisement is for a destination D at distance metric C learned from preferred neighbor N.
- Every time a routing table entry is added, changed, or refreshed, its age should be set to 0.
- The distance you record in the routing table should always be one more than the distance advertised by your neighbor.
- When you receive a route for a new destination at a cost less than INFINITY, add it to the routing table.
- When you receive an advertisement for a route that is cheaper than the route currently in the routing table, replace the existing route with the new one.
- When you receive an advertisement for a route from the preferred neighbor and its cost has not changed, refresh the route by setting its age back to 0. If its cost has changed, update the routing table entry to reflect that change. If the cost of the route is now INFINITY, remove the route from the routing table.

- Print the following messages using `fish_debug` when changing the contents of the routing table. The new route (or the route being deleted) is to destination `D` via preferred neighbor `N` at cost `C`.
- When adding a route to a new destination, print "Route add to `D` via `N` cost `C`".
- When changing a route to a known destination, print "Route change to `D` via `N` cost `C`".
- When refreshing a route, print "Route refresh to `D` via `N` cost `C`".
- When removing a route, print "Route remove to `D` via `N` cost `C`".
- Forward any packets received from neighboring nodes that are destined for other nodes, decrementing the TTL as in the example program. Instead of sending the packet to a random neighbor, you should send it to the preferred neighbor listed for the destination in the routing table. If there is no route for the destination, print "`D unreachable`", where `D` is the destination.
- Your program must perform all tasks in any order and run until you give the command "`exit`", when `libfish` will end the program.

3. Step-by-Step Development and Test Instructions

You can go about building your program in any fashion you prefer. Here is a suggested set of steps to develop the required functionality.

1. Download the Fishnet package from the course directory and copy it into your home directory. The easiest way to do this is:


```
cp ~cs123b/fishnet.tar .
```
2. Unpack the sources using `tar`, as follows:


```
tar xvf fishnet.tar .
```

As a general note, you can use the `man` command to get a help page for any Unix command or function we ask you to use that you don't understand.
3. Read `fish.h`. The comments in this file tell you what the Fishnet API calls are, what they do, what arguments they take and return, and so forth. You won't find this information anywhere else.
4. Start a process to manage a Fishnet by running the `fishhead` program that is inside the `fishnet` directory. Type "`./fishhead --help`" to find out what command line arguments it accepts. You will need to give it a argument to indicate which port it should listen on for messages. Pick a number between 1024 and 32K. If the number you choose is the same as someone else's `fishhead` will fail to start. The `fishhead` program runs indefinitely to keep the Fishnet it is running up. You should leave it running and create another window in which to do your development. When you're done, stop it by typing Control-C. Note: the `fishsrc` directory contains the fishnet source code so that you may see it and use it for debugging, but you must not change it.
5. Test the `fishhead` by running the sample program, `random`, in a separate window. You will need to specify the machine and port number used by your `fishhead`, along with a domain (use any value here) and a number to specify the address of

- the newly added node. Observe the messages that appear in the fishhead window. Now repeat the process by executing another instance of `random` in another window using a different address. You can now send messages between the two nodes using the “send” command. Make sure you are comfortable with sending messages between two or more nodes in this manner. You can observe the topology of your fishnet as each node is added by opening a web browser and directing it to <http://machinename.ucsd.edu:port>, where `machinename` is the machine the fishhead is running on and `port` is the port you specified when starting the fishhead.
6. Now start developing your program. It should be entirely contained in the file called `hw1.c` in the top level directory created when you unpacked Fishnet. At the top of this file you MUST include a comment including the full names and e-mail addresses of EVERYONE in your group. We have supplied a Makefile so that you can type “`make hw1`” to compile your program. This will invoke the compiler, `gcc`, with the right command line arguments. As you write your program, note that our solution is roughly 400 lines of code, of which more than 150 are included in the example program. Consequently if your solution is significantly more than this then you are doing something the hard way and should ask for help. You may want to start `hw1`, by starting with the example program as follows:
 - a. Copy `random.c` to the new file `hw1.c` in the same top level directory created when you unpacked Fishnet.
 - b. Add a comment to the top of the file including the full names and e-mail addresses of everyone in your group.
 - c. Compile this program by typing “`make hw1.c`”, execute it and have it connect to your fishhead to make sure that it behaves like `random`.
 7. Leave the program performing random forwarding and add the code for the periodic routing timer. Look at `fish.h` to see the timer API calls. When the timer fires, send a routing update message with one advertisement (for the node itself) to all neighbors and print the required messages. Test this by running both a two and three node network and seeing that routing updates are exchanged.
 8. Define the structure for a routing table entry and add the routing table. Change the random forwarding routine to forward using the routing table and print out the forwarding related messages. Test this so far by running a two node network. You should not be able to send any non-routing messages (they should all be dropped because there is nothing in the routing table).
 9. Write the code to add new advertisements in routing updates to the routing table and to send routing updates that encode the information in the table. Test this with a two and three node network. You should see routes added at each node for the other nodes, and forwarding should have begun to work.
 10. Write the code to handle the remaining routing update cases. Test your program on a two node network. Routes should be added, and refreshed too.
 11. Write the code to age entries in the routing table and expire them when they get old. Test this with a two node network by letting routing stabilize and killing one node. The other node should eventually expire its route to the killed node.

12. Comment your program if you haven't already. Good comments don't belabor the obvious (e.g., "calling the main loop" near `fish_main()`). Rather, good comments tell us how you have arranged your code and assumptions you have made, as well as anything non-obvious.
13. At this stage you have a complete program and should do the turnin cases.
14. *For Fun.* (Do this after you have saved the turnin output.) Speed up route convergence by adding triggered updates. Whenever a received update changes your routing table, then immediately send updates to your neighbors. Try this on the turnin test case to see the difference.
15. *For Fun.* (Do this after you have saved the turnin output.) Speed up route convergence by adding the split horizon with poison reverse heuristic. When sending an advertisement to a neighbor, note which advertisements were learned from that neighbor in the first place. Send these routes with a metric of INFINITY. Try this on the turnin test case to see the difference.

4. What to Turn In and Our Grading Philosophy

To prepare for turnin, you need to gather the output of your program running the test cases below as well as answer two discussion questions.

1. Run a three node network, with each node A, B, and C running in a separate window. Wait until the routing tables have stabilized and send one packet from A to B. Then kill node B (with a Control-C) and wait for the routing tables to stabilize. Try sending from A to B and observe the result. Now restart node B, wait for the routing to stabilize, and try sending again. Capture the entire output of the three sessions using, for example, `script`. To deal with the typing input while your program is producing output, you may want to cut and paste the "send" command.
2. Question: What happens after the node B fails and why? Describe the general progression of your output for nodes A and C in no more than three sentences.
3. Run a three node chain network by starting the fishhead with the "--topology" argument and then starting nodes A, B, and C in that order. Node B should be in the middle of the chain. Wait until the routing tables have stabilized. Then kill node C and wait for the routing tables to stabilize. Capture the entire output of the three sessions using, for example, `script`.
4. Question: What happens after the node C fails and why? Describe the general progression of your output for nodes A and B in no more than three sentences.
5. Question: The periodic rules we have specified age routes and then send updates. Instead we could send updates and then age routes. Which scheme is better and why? Hint: consider the first test case (with a triangle topology). You may want to try changing your code to see the effect.

You should turn in one copy per team of electronic and paper material as follows:

1. Join the class fishnet at `ieng9.ucsd.edu:7777` using your program and send a message to node 1, the bulletin board node.

2. One C file called `hw1.c` containing the source code of your solution. Again, remember that you must include a comment at the top of this file listing the full names and e-mail addresses of everyone in your group. Submit this electronically using the `turnin` program, as follows:

```
turnin -c cs123b hw1.c.
```
3. One stapled paper writeup with your names and e-mail addresses that contains:
 - a. A printout of the output from the test cases described above.
 - b. A printout of the source code you submitted electronically.
 - c. Short answers to the discussion questions above.

Our intent is to have you explore networking issues, rather than write a program that manages to pass a given set of acceptance tests. We will grade you on this material in two ways. First, we will look at your writeup and the C source code. The printout, your description of what is going on, your answers to the discussion questions, and the design clarity of your program, give us a very good idea of how well you got the program working and understand the issues that it explores. Second, we will sometimes compile and run your programs. We will do this cases that we are unsure of after looking at the source and your printout. We also reserve the right to randomly test programs to see that they produce the output that you claim they produce. When we test, it isn't our intent to penalize you significantly for small errors in execution, as opposed to larger errors in understanding of the concepts and design of the solution, but we will deduct some points for small errors or not following our instructions. We will also sometimes include optional material in the assignments, and hope that you choose to explore some of the issues this material raises, but we won't grade optional material.