Project #2: FishNet Distance Vector Routing

Assignment documentation version: 1.0
Due Nov 18th, 2009

This informational handout covers introductory material describing the Fishnet software that you use, the Fishnet programs that you write, the development environment, and testing strategies.

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 pairs. If you follow the instructions it should be possible to interconnect nodes designed by each team into a single network.

1.1 Development Environment
We will be developing under Linux and writing C code. It is assumed that you will use a window system (either X windows natively under Linux or using Windows/MacOS and opening multiple ssh sessions in different windows) to allow multiple processes to be visible at once. Each node of the fishnet network will run as a separate process, but to distinguish their output it is convenient to run each process in a distinct window. The compiler we will use is gcc, and the debugger we recommend is gdb. You may use other tools, but we won't support them and we will grade your assignments using the default tools and Makefiles (i.e. its 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, originally designed at the University of Washington. Second, it is a running network of many nodes. This is what we most commonly mean by fishnet. Note, however, that there are many potential running fishnet networks. You will all start your own, private, fishnet as a managed overlay controlled by a single fishhead (a key component to be described shortly) to develop and test the code required for assignments. It will run as cluster of processes on one or more PCs. In this form your nodes will not interact with nodes of other students networks.

1.3 Fishnet Components
The Fishnet project code, like everything else you need, is available from the class Web site at:

http://www.cs.ucsd.edu/classes/fa09/cse123/fishnet/

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.)

As you develop your Fishnet node and test it, one of the first things it will do is to join a private 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 the nodes in the network can come and go. Type fishhead --help to get usage information.

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_send() 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 (including our own).

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. Getting Started
First, login to a linux box (e.g. ieng6) and download the fishnet tarball onto your by typing:
wget http://www.cs.ucsd.edu/classes/fa09/cse123/fishnet/fishnet-hw2.tar
and then expand it by typing:
tar xvf fishnet-hw2.tar

Next build the fishhead program and libfish.a library by cd’ing into the fishsrc directory and typing: make

You can now launch the fishhead program. There are a number of options (fishhead --help will provide a list) that are useful for testing your solutions under different topologies and under different loss conditions. For example: fishhead -t chain will arrange that fishhead nodes are linked in a chain topology: a -> b -> c -> d and so on, while fishhead -t ring will create a ring topology: a->b->c->a).
When your fishhead starts it will report the hostname and port it is running on. For example:

```
./fishhead -t ring
21:18:48.696751 fishhead started successfully on ieng6-202.ucsd.edu:7777
```

Terminate fishhead when done with ^C.

This indicates that the fishhead process is running on ieng6-202.ucsd.edu at port 7777. The fishhead program defaults to using port 7777, but if someone else is already using that port on that machine you will need to select another using the –port argument (select a port between 1024 and 32,000). Remember the machine name and port used by your fishhead because you will need them to connect your nodes to the fishnet network it is emulating. Please do not connect to other people’s fishhead processes without their permission.

Next 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.

Next, we have also provided two sample programs: hello and flood. You can compile either of these by typing: make hello or make flood.

Lets start with the hello program. Open a new window, and launch the hello program by typing:

```
./hello machine:port 1
```

Here “machine” and “port” are the two values your fishhead returned and the number 1 is the “address” of the node you are creating (the number 1 is arbitrary… it’s just a name). Repeat this in yet another window (you will now have three windows open) but using a node address of 2 instead of 1. There will be some status messages in each of the three windows.

The hello program takes input but accepts only one command “hi <dest>” where <dest> is the address of the node you wish to send a message to. Try sending “hi 1” from the second node. You’ll see messages indicating that a message is being sent on one window and received on the other. If you look at hello_console_command() you’ll see how this gets done by filling in a packet data structure and calling fish_send(). You can safely end each of the hello processes using Ctrl-C and the fishhead can remain running. Note that hello assumes that the node it is sending a “hi” to is a direct neighbor. This is not true in general (to test this out you can start up fishhead with the –t chain option and then create three hello nodes: 1, 2 and 3. If you try to send hi 1 from node 3 it will complain that 1 is not a valid neighbor).

Next look at the more complex example in flood.c. This program provides two commands “p <dest>” (which pings a node with an ECHO_REQUEST packet and
awaits a corresponding **ECHO_RESPONSE** packet in return) and “pa” (which pings all nodes). Unlike the **hello** program, **flood** will arrange that these services work no matter how many nodes are in the fishnet and no matter what topology is in place. It does this by forwarding each packet it receives to all of its neighbors – otherwise known as “flooding”. To ensure that a packet is not flooded by the same node multiple times each node keeps track of which packets it has seen before (using per-node sequence numbers that are monotonically increasing). There is some trickiness here so it’s worth looking through the code to understand it. Try creating larger networks (with 5 or 6 nodes) and turn on debugging messages to see how the packets are sent.

### 3. Project 2: Distance Vector Routing

For this project the goal is to improve on the blind flooding of the flood.c program and implement a real forwarding table, populated using a basic distance vector routing protocol to compute the next hop neighbor along the shortest path to the destination.

Your program should provide all of the functionality of flood.c and **additionally**:

- Maintain a routing table which is an array of MAX_ADVERTISEMENTS -1 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.
- Forward any packets received from neighboring nodes that are destined for other nodes, decrementing the TTL as in the flood program. However, instead of flooding the packet to all 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.
- Use the `routing_packet` structure defined in fish.h to periodically send a routing update to all neighbors every five seconds. The value of the packet header’s protocol field should be set to FISH_PROTOCOL_ROUTING_DV; the destination should be ALL_NEIGHBORS. The packet will contain the distance vector – it 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 current entry in the routing table (in the beginning there will only be the local node, but other entries will be learned over time). Note that the size of a routing packet is `PACKET_HEADER_SIZE + num_adv*sizeof(struct route_advertisement)`.
- Upon receiving a routing update message from a neighbor, each node should update their routing table as follows:
  - First, when considering any route advertisement always add one to its metric (hop count really) since the neighbor you heard it from is one hop away.
  - When you receive an advertisement for a new destination at a cost less than INFINITY, add it to the routing table with an age of 0.
When you receive an advertisement for a destination 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 destination from the current preferred neighbor and 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.

Every time a routing table entry is added, changed, or updated but not changed, its age should be refreshed to 0.

Periodically “age” the routing table every 5 seconds (you may want to combine aging and sending out the distance vector as part of the same timer). To age the routing table, you should remove any entries that have not been refreshed for the past three consecutive aging intervals (i.e. for 15 seconds).

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".

Your program code should reside entirely in a file called hw2.c (note that it is ok to include echo-protocol.c like flood.c does) and at the top of this file you MUST include a commend including the full names and student IDs of your group members. We have supplied a Makefile so you should be able to simply type “make hw2” to make your solution (this is what we will do to make your solution).

3. Some implementation suggestions
   - Use flood.c to start with (why not start with something that that works after all).
   - Leave the program performing flooding and add the code for the periodic routing timer. Look at fish.h to see the timer API calls (also look at the refreshSeqNumTimer example in flood.c). 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.
   - Define the structure for a routing table entry and add the routing table. Change the forwarding routine to forward selectively using the routing table (instead of sending to ALL_NEIGHBORS) 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).
• 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.
• 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.
• 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.
• 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 make, as well as anything non-obvious.
• At this stage you have a complete program to turn in.

4. Extra credit
• Only do these after you have a working solution and have saved it. If you attempt the extra credit then please include a comment at the top of the file: “Extra credit:” followed by “triggered” and/or “split” depending on which you have done.
• 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.
• Only do this after you have a working solution and have saved it) 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.