Fishnet Assignment 1: Ping and Flood

Due: Friday, Jan 17, 2003 at the beginning of class. Out: Monday, Jan 6, 2003.

CSE/EE461 Winter 2003; Wetherall.

In this assignment, you will work in teams of two to develop a single C program that is a Fishnet node. Your node will participate with those written by other students in an ad-hoc network run on the IPAQs and exchange simple messages. We will extend the nodes and network with functionality throughout the quarter. The goals of this first assignment are to become familiar with the Fishnet development environment and to understand packet forwarding concepts.

1 Preliminaries

Before you write any code, make sure you work through the ***Introduction to Fishnet*** handout. This covers basics such as using the CSE Lab machines, installing the Fishnet development environment, and how to set up a Fishnet network. It also tells you key things to do **before** you try to run your node on the live class network using your IPAQ. You will receive your IPAQ and address for the class network in section. You don't actually need to use it until you have working code, since you develop and test your code on the CSE Lab machines; the IPAQ is mainly for using the network.

2 What To Develop

Your assignment is to write a C program in a file hw1.c that acts as a Fishnet node with the following three types of functionality:

1. ***Flooding***. Your node "floods" each packet it handles so that the packet reaches all other nodes connected to the network. You must decide exactly how your flooding function will work (there is probably more than one choice) but it must be accomplished subject to the following constraints:

    - The flooding logic must work for all kinds of packets but ONLY use packet information accessible from packets of type `struct packet` as defined in fish.h. You must implement functionality to fill in all of the fields in these packets as they are intended. This is necessary so that your flooding implementation can interoperate with our implementation as well as that of other students.
    - Flooding must deliver a copy of every packet that is sent by any node to all of the other nodes. At each node, the copy of the packet that is delivered should be processed further only if the packet is destined for them. It may be destined for them either directly, if the destination address of the packet is their node address, or as part of a network-wide broadcast, if the destination address is the broadcast address.
    - Notwithstanding the above, flooding will fail to deliver a copy of a packet to all other nodes that will process it in a couple of situations. First, packets can

be lost during transmission, e.g., corrupted due to noise on the wireless link, detected as a checksum failure. Second, as part of using packets of type `struct packet` you must implement functionality to update the TTL field when packets are forwarded. This will cause packets to be discarded without being forwarded if their TTL reaches zero.

- Your flooding design must prevent packets from circulating indefinitely. This requires that your node work out when it has already performed its flooding work for a given packet and not flood it again. The header fields in packet (`pktid` and `source` in particular) are suited to this task.
- Your flooding must work for all network topologies including those that change as nodes move, yet without a priori knowledge of the topology. You should not build up maps of which nodes are where in the network. Building up maps, called routing, can greatly improve the efficiency of communication but can be surprisingly complicated – we will study it in the next assignment.
- As a tip, you can use broadcast as the first parameter to `fish_send()` to relay packets between nodes. This allows you to send a packet to all of your neighboring nodes (within radio range or directly connected as part of the managed overlay) without knowing their individual node addresses. There is no other way to send your neighboring nodes a message until you have discovered their identities, which you will not be able to do until you have completed this assignment (except a priori for simple networks that you set up yourself).
- As a tip, to simplify your implementation you can assume that the network contains only a small number of nodes (less than 100), all of which have small addresses (less than 100). A better implementation will work with larger node addresses and networks, but this is not necessary for the first assignment.

The reason you are implementing flooding is that, without it or some more complicated alternative, one node cannot send a message to a distant node in the network that is out of radio range. Flooding uses other, in-between nodes to relay or forward the message. Flooding variations are used as components of other network protocols, such as link-state routing (Peterson 4.2) that we will study later in the course. They are normally used to provide point-to-all communication, and are very inefficient as a way to send a message from one node to another, as we will for ping below. However, flooding will get us started with networking and motivate the need for more efficient techniques (routing) in subsequent assignments.

A good flooding design (or network protocol in general) will use no more packet transmissions or node resources than necessary to accomplish its function. As you develop your design, you should check that it works (all nodes receive a copy of each transmission they need to process and yet the flood eventually stops) and see how many packets you are using to make it work. You should do this for several different topologies (use `--help` to fishhead to see your options).

2. ***Ping***. Your node can "ping" another node (bounce a packet off of it) to check that it is working. Ping is a well-known Internet utility that is used to check that a remote host

is alive – try "`man ping`" and see RFC 792 Internet Control Message Protocol (http://www.rfc-editor.org/rfc.html) for inspiration. You decide exactly how your ping function will work (there may be only one choice here!) within these constraints:

- You must use ONLY packets of type `struct echo_packet` (defined in fish.h) as they are intended. This is necessary so that your node can interoperate with other nodes.
- If you ping using a packet with the address of a particular node as the destination, then only that node should respond to the ping. If you ping using a packet with the network broadcast address as the destination then all nodes in the network should respond.
- As before, you do not need to worry about reliability in the case that ping packets are lost, e.g., due to corruption on the wireless link. In real life, packets are occasionally dropped on the Internet, and so ping tries to bounce several packets off of a remote host to see if it is alive.
- As a tip, you will probably want to use keyboard (or other) input as a command to cause you to "ping" a node as well as output. See the hello.c program for an example of a command. You may also want to print output, and perhaps make a sound on the IPAQ, when you successfully ping or are pinged by another node so that you notice someone is communicating with you.

3. ***Neighbor Discovery***. Your node continuously probes the network at a low-rate to discover its immediate neighbors in the network topology. Again, you devise a solution within these constraints:

- You must not use any additional kinds of packets. This task can be achieved by a careful combination of the functions that you have already built (ping, flooding, the network broadcast address, and the TTL field).
- Neighbors disappear (when nodes move away or are turned off) as well as appear (when nodes move into range or are turned on). You will want to print out the current list of neighbors so you can see who they are, perhaps only printing when there is a change.
- As a tip, you will need to use timers, e.g., `fishnet_scheduleevent()`, to implement continuous, low-rate background activity. Be very careful with automated mechanisms, especially when using flooding and broadcast! They should operate on the timescale of at least tens of seconds (tens of thousands of milliseconds in the API calls!) or you may inadvertently bring down the building wireless network.

The reason you are implementing neighbor discovery is to provide some way to find out when your IPAQ comes into contact with other class nodes. You may want to play a sound on your IPAQ to alert you this situation. Once you know your neighbors, you can ping them directly using their addresses. This is what you will need to do as part of turn-in.

A good design will use very little bandwidth yet keep a reasonably accurate set of neighbors. As a challenge, note that we can implement neighbor discovery with echo packets and flooding without having packets be processed at any nodes except the neighboring nodes!

The above constitutes the intellectual bulk of your assignment. It is probably simplest to implement the functions in the order they are given above. As you write your program, note that our (fairly verbose) sample solution is not especially long. If you are writing hundreds of lines of code then you are probably doing something the hard way and should talk to us about it. Make sure you comment your code so that your protocol designs are apparent to us; we expect these comments in lieu of separate, detailed written descriptions. 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.

3 Design Philosophy

There are two key issues to bear in mind as you design your solutions, for this assignment and all future ones.

Robustness Principle. An important rule of thumb in building network protocols is "Be conservative in what you do, be liberal in what you accept from others." (RFC 793, Transmission Control Protocol, http://www.rfc-editor.org/rfc.html). This helps different implementations of a protocol (e.g., the sample solution, your program, and your classmates' programs) to work together reliably. This principle means you should be careful to send packets that strictly comply with the intended usage of the packet formats as described in fish.h so that other nodes can handle them, but you should do the best you can with whatever packets you receive from other nodes. Of course, you will get it right, but *they* will send broken packets. In particular, other nodes shouldn't be able to crash your node by sending it bad packets. If your node does crash, it's your responsibility to find out what happened and fix it.

Interoperable Designs. It is also possible that different students will design protocols that are not compatible with each other in the class Fishnet, even if everyone's code is robust as defined above. We hope that strict adherence to the packet formats and their intended usage will result in compatible protocols, without any further need for specifications or constraints on your designs, but we can't guarantee this. Therefore, you must check that your design is legal in that it interoperates with the reference executable that we provide. You must do this in your own, private fishnet before attempting to join the shared, class network running on the IPAQs or you may interfere with the proper functioning of the class network. It is everyone's responsibility to work towards compatible, interoperable protocol designs by checking for incompatibilities and discovering what further design details we need to make together, as a class. This is a very real issue in the Internet, and part of what you will learn during the course.

4 Discussion Questions

1. Describe one advantage and one disadvantage of the "upcall" style of programming.

2. Flooding includes a mechanism to prevent packets from circulating indefinitely, and the TTL field provides another such mechanism. Why have both? What would happen if we only had flooding checks? What would happen if we had only the TTL?
3. When your node pings a remote node using its particular destination address (rather than the network broadcast address), how many requests does the remote node receive and why? How many responses does your node receive and why?
4. When your node pings a remote node using its particular destination address (rather than the network broadcast address), how many request and response packets do other nodes handle? How many of these packets are unnecessary, and could probably be eliminated with smarter networking protocols?
5. Describe one design decision you could have made differently (you are allowed to change Fishnet) and the pros and cons compared to the decision you made.

Write only a few, short sentences for each of these questions!

5 Turn-In

For this and future assignments, you need to demonstrate that your IPAQ works in the class network as well as turn in both electronic and paper material as follows.

1. Run a three node private fishnet (using your program only, not the sample solution), with each node running in a separate window. Make the nodes form a chain network (see `./fishhead --help` and look for `chain`). From one node, ping another node using its specific address. Capture the entire output of the three sessions using, for example, the `script` command. Make sure the debugging level is FISHNET_DEBUG_ALL (the default) so that we can see what packets are being sent and received. Mark up the printout to tell us what is going on.
2. Use your IPAQ to ping our fishnet node running in 324 (the Systems' Lab) directly, by using its specific address. You just need to get within radio range and discover its address. You can identify our node as the one sending periodic pings to neighboring nodes with packet contents "Fishnet is alive".
3. Use the `turnin` program on the Linux servers to electronically submit one or more C files containing the source code of your solution, including the Makefile if you have changed it. You must do this before class on the day that it is due. The code you send us should compile as is with the command "make hw1".
4. In class on the due date, hand in one stapled paper write up, with both partners' names on it, containing:
    a. A printout of the source code you submitted electronically.
    b. A printout of any output we have asked you to capture.
    c. Short answers to the discussion questions.
    d.

—END—