Introduction to Fishnet

Tom Anderson

Fishnet is a software project for teaching the core principles of network protocol design and implementation. This document describes our goals in designing Fishnet, its software architecture, and how it can be used to study network protocol design. Other documents describe the specific assignments in detail. Fishnet was originally developed by Neil Spring, David Wetherall, and Janet Davis. We have completely re-written the system for this quarter, converting the implementation from C to Ruby, a powerful object-oriented scripting language. We have also substantially added to the functionality of Fishnet in the latest version, but much of the original approach has been preserved.

Goals

We had several goals in mind in developing the latest version of Fishnet:

Simplicity. A primary goal was to provide a simple, clean software environment for developing, testing, and running network protocols. The simpler the interface, the faster students can focus on the more fundamental issues of network protocol design. This implies that we necessarily must abstract away some of the more tedious details of protocol implementations on systems such as Linux and Windows.

Completeness. Fishnet can be used to design, implement, and evaluate any type of network protocol, from the media access layer, through routing and transport, up to application-level protocols such as web transfers and email. In this class, we will use Fishnet to develop all of the protocol software (above the MAC layer) needed to support a walkie-talkie application on an ad hoc wireless network. Simplicity and completeness go hand in hand – if we were to ask students to grok the full complexity of a full TCP implementation, we would have time for little else. We note that few software developers hack TCP implementations in practice, but much, much larger numbers need to know how to develop protocols that work, since protocols are an essential aspect of any distributed system. So we focus on the principles underlying protocol design, rather than the mechanics of any specific protocol.

Uniform interface. Fishnet is unique in that it provides the network protocol developer a single, uniform interface to running their code in a variety of environments: as a simulation in a single process, an emulation across multiple processes, or on a physical network (such as a collection of IPAQs). This simplifies development, as different issues are best studied in each context. Simulation is best for initial protocol development and testing, as execution runs are completely reproducible. Of course, reproducibility is not typical of real networks! Emulation adds a level of realism and unpredictability, as nodes come and go and packets are corrupted and dropped in an unplanned fashion, but users are still able to use traditional debugging tools to study the system's behavior. Of course, the end goal is deploying the system on a real network, which inevitably exposes additional issues. By allowing the same network protocol code to be used in all three contexts, Fishnet enables students to focus on the protocol code, not the tedious details of converting from one environment to the next.

Quantitative evaluation of protocol robustness and efficiency. A major issue in the design of any network protocol is adaptability to the physical environment -- failures, topologies, capacities, and workloads. In Fishnet's simulation and emulation modes, the topology, capacities, and failures are controlled via a separate file. A topology and failure generator is provided to help automate testing. The result is to make it easy for students to test their protocols in a variety of situations. Of course, on a physical network such as the IPAQ's, the topology is not under the student's control – it is determined by the physical location of the various IPAQ's to each other. Any failures that occur are likewise unpredictable. Thus, it is especially important to test protocols against as wide a range of operating conditions as possible.

Event-driven software architecture. Network protocol implementations typically come in two flavors: process-oriented and event-driven. In the former, threads or lightweight processes synchronize access to shared network protocol state using mutual exclusion. This is the "OS-centric" perspective, since operating systems must already deal with multiple processes (to support the execution of multiple concurrent user programs), they often cast network code in the same light. However, this leads to fairly complex implementations that often have subtle race conditions due to the difficulty of designing correct synchronization. Instead, we use a completely non-threaded, event-driven model, where the protocol code must completely deal with an incoming packet and return before the next packet can be processed. Explicit state must be saved to enable the protocol code to keep track of what to do next; in a threaded system, the thread stack automatically stores the protocol state. In truth, the event-driven and thread models are duals of each other – exactly the same protocol can be expressed in either context. We believe network code is simpler and easier to understand and implement correctly when cast in an event-driven context. Some commercial web servers, for example, have been re-implemented in an event-driven model, resulting in better performance and fewer bugs. Avoiding thread synchronization also means that we don't need to assume an operating systems course as a pre-requisite.

Extensibility and interoperability. A key factor in our choice to move Fishnet to Ruby was to enable students to study protocol extensibility and interoperability at the same time. Students have complete control over protocol design in simulation mode and when emulating a network, if the student specifies the behavior of all of the emulated nodes. However, Fishnet also supports interoperability – given standard packet formats, students can test their protocol implementations to see if they work with those of other groups, by having each node in the emulation, or each IPAQ on the physical network, run code from a different student group. This exposes students to one of the key issues in protocol engineering – how do we design protocols so as to make them robust against the inevitable software implementation errors of others? Ruby lets us take this one step further – network protocol implementations can be easily shipped around the network, so that peers participating in a network protocol can interoperate using a new, more advanced protocol. For example, all groups could implement a basic transport protocol, while some groups take the further step of exploring a more advanced protocol, where the code for the more sophisticated protocol is shipped to all the participating nodes in the physical network.

Fishnet Programs

We next turn to discuss the various components of Fishnet. There are three separate Fishnet-related programs:

fishnet – This is the main code implementing the Fishnet programming model. Using command line arguments (or equivalently, by redefining ARGV in Ruby before the Fishnet code is invoked), a simulation/emulated node/ipaq node is started. In simulation mode (-s), all nodes run in the same address space, and the simulator switches between them by scheduling and delivering packets between the various nodes. The command line arguments specify the number of nodes to simulate and the topology file to use for interconnecting the nodes. In emulated mode (-e), fishnet starts a single node that connects to the trawler (see below); the command line arguments specify where to find the trawler. In IPAQ mode (-i), fishnet starts a single node that uses and listens to wireless broadcasts to determine the local topology; the command line arguments specify the fishnet address to use – this is the number written on the bottom of the IPAQ.

trawler – The trawler manages a Fishnet emulation. A student can run their own Fishnet trawler; in this case, the student would run all of the nodes in the emulation. Note that the trawler and each of the emulated nodes should be started in their own window, to keep the input and output distinct. Once this is working, a student can connect to the shared, course-wide trawler; in this case, students provide create one or two emulated fishnet nodes (again, in their own window) and demonstrate that their solutions interoperate with the implementations provided by other students (and faculty). The trawler will run indefinitely waiting for nodes to connect to it; students must explicitly kill any trawlers and any emulated nodes that they start, or they too will run forever.

A key aspect of the trawler is that you have to pick a port for the trawler to listen for incoming emulated node connections, and another port for every emulated node to listen for incoming packets from other emulated nodes. These ports can be arbitrary numbers less than 64K, as long as they don't conflict with other ports that are in use on the same machine. In general, many of the ports less than 1024 are in active use by the host operating system, e.g., for receiving normal Internet packets for such services as email, so you want to pick port #'s higher than that. If you happen to pick a port that is already in use, you'll get back an error message and you can try again with a different number.

Note that the trawler assigns each emulated node its own unique fishnet address. This is equivalent to the Internet's DHCP protocol. On the IPAQs, since we don't necessarily have access to a server in ad hoc mode, students provide each IPAQ's fishnet address via the command line; please use the number written on the bottom of each IPAQ. In simulation mode, the simulator assigns each node a unique fishnet address.

Topogen – This utility generates sample topology files under command line control. The generated topology files are in the format expected by the Fishnet simulator and trawler. Note that an emulated Fishnet node does not control its topology; the trawler specifies which nodes are neighbors. Likewise, a Fishnet node running on an IPAQ gets its neighbors from wireless broadcasts – any Fishnet node that can receive the signal is a neighbor.

Fishnet Files

The distribution has two groups of files. In the proj directory, you will find the files implementing the network protocol code that runs on Fishnet. These files are modifiable by students. In the lib directory, you will find the files implementing the Fishnet runtime environment supporting student network protocol code. These should NOT be modified by students. In particular, we are likely to need to provide updated versions of these files to support later assignments, fix bugs, etc, and so any changes you make will need to be remade to other versions. Further, any changes to lib may prevent your code from interoperating with that of other student groups.

However, students should read and understand the code in the lib directory, as it will be useful during debugging. It is ok to add temporary debug messages into these files, if that helps track down the source of a problem. In general, though, students should limit their changes to the files in proj (and files loaded into the files in proj).

All Ruby files are given a .rb suffix. Ruby files are interpreted, so no separate compilation step is needed; neither do you need any Makefiles. You will need to execute the file lib/fishnet.rb to run Fishnet, and lib/topogen.rb to run the topology generator, etc. Exactly the same code runs on Windows, UNIX boxes, and the IPAQ's, so we do not need to maintain separate executables for each environment (as we would if we were programming in C or C++). Note in particular that the IPAQ's use an ARM CPU, so binary files built for an x86 Linux will not run on those devices.

Files in proj/

Node.rb – this is the code that the students write to implement a protocol on a single node. (Feel free to use multiple files, using "require" to load the files into Node.rb.) Node.rb uses routines defined in fishnet.rb (the Fishnet class) for speaking to the network; in general, the user software in Node cannot and should not know whether it is running inside a simulation, as a separate node in an emulation, or on a physical IPAQ.

Files in lib/

Packet.rb – this file defines the common packet formats to be used by students, for interoperability with the implementations of other students.

Fishnet.rb – main routines for parsing command line options and starting either a fishnet simulation, fishnet emulated node, or fishnet on an ipaq. Note that since it is difficult to type on the ipaq, feel free to rename the fishnet.rb file to something shorter (e.g., f) on the ipaq's for easier typing.

Simulator.rb – routines for implementing a Fishnet simulation.

Physical.rb – routines for supporting an emulated Fishnet node or a Fishnet node running on an IPAQ. Both simulator.rb and physical.rb implement instances of a common abstract Fishnet class – that is, they support the same set of methods to send and receive packets, to set timers, etc.

Commands.rb – routines for parsing input from the keyboard and/or the topology file. The specification of legal commands are defined at the beginning of the commands.rb file. One can define edges, give edges specific latency and capacity characteristics, cause edges or nodes to fail and be rebooted, and to control the relative timing of these events

in both simulated and emulated time. Further, since the student code may also need to process keyboard input, any uninterpreted commands are sent to the code in Node.rb.

Topology.rb – routines for tracking a simulated or emulated topology.

Trawler.rb – routines for implementing the code that manages an emulated Fishnet. The trawler uses the input topology file to tell each emulated node who its neighbors are; the emulated nodes use TCP to talk to the trawler, and UDP to directly communicate with their neighbors. In other words, the trawler just sets up the emulation – it does not mediate any of the actual packet deliveries.

Utility.rb – generically useful routines

Topogen.rb – the topology generator; produces a file that can be used as input to the simulator and the trawler.

An Example: Ping

In order to test out the various parts of Fishnet, we implemented a simple "hello world" program on Fishnet. This code takes keyboard commands, and sends the text to the specified node (e.g., the command "5: Hi there" will send the text "Hi there" in a packet to node 5). The receiving node then copies the contents of the incoming packet into a response packet sent back to the source. This implements a version of the Internet's "ping" protocol 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)

The "ping" example illustrates the various entry points to send messages and establish wakeup calls, as well as the various upcall entries into student code – to start a node, to receive a packet, to receive keyboard input, and to be woken up after a timer expires. Note that the node software is provided its fishnet address in Node::start; this allows your code to be generic whether the fishnet address is specified on the command line or provided dynamically.

As a first step, you should download the Fishnet code into your account, and get the ping program to successfully send messages in simulation mode, emulation mode, and between a pair of IPAQs. The Fishnet code is available on the course web site, as a .tar file. On Linux, use tar –xf file.tar to extract the files from the tar file (cf. the tar man page).

Ruby on Linux and Windows

Ruby is highly portable and freely available. This means you can work on your projects on any computer you like. If ruby is not already installed, you can download and install it in about 5 minutes from the main ruby web site, www.rubycentral.com.

Ruby on linux is pretty straightforward. As described in the Ruby book, linux supports the shebang rule – if you include a "! ruby" command on the first line of a ruby file, executing the file will cause the shell to invoke the command on the file. (This works for other commands too, such as csh, to allow shell scripts written in slightly different dialects to control which shell is used for interpreting the commands.)

For Windows, the distribution comes packaged with a syntax aware editor that lets you run ruby code from within the editor. This is pretty convenient, but it requires you to add

a line or two to the beginning of fishnet.rb, to specify the command line arguments that you would have been able to provide had you run it from the shell. Note however, that the Ruby syntax-aware editor on Windows appears to lack support for sockets; thus, you can run the simulator under the editor, but the emulator code must be run (on Windows) using the ruby executable or the interactive ruby shell (irb).

Debugging

Our goal in the programming assignments is to give you a grounding in real protocol design and implementation, not to have you spend all your waking hours debugging. As the saying goes: I hear and I forget, I see and I remember, I do and I understand. We are principally concerned with your design choices, and much less concerned with whether the code is completely bug free. Ruby is a very good language for developing a quick initial prototype, and less good for ensuring that every last bug has been found.

That said, the time you spend doing the assignments is principally dependent on how carefully you write code (thereby avoiding errors), and how quickly you can find and fix any problems that occur. Whatever you do to minimize the time spent fixing bugs is fine by us. Ruby is particularly good at reducing the run-debug-fix loop – the interactive ruby interpreter means that you can try out your code immediately, you can try out small bits of code incrementally to see whether the code behaves as you expect, and you can get parts of the system working without needing every last line to be free of errors.

The approach that we recommend is to do your initial debugging under the simulator. Once your code is working there, set up your own trawler, and use a small scale emulation as an additional level of verification. One can think of this as the scientific method – change one thing at a time, and test, before changing the next thing. Once you have a private emulation working, you can run a process that joins the course-wide Fishnet trawler. We will provide sample solutions, running continuously as emulated nodes on the course-wide trawler, for you to test your solutions against. (Note that we may also provide some "evil" nodes to see if your code is robust against misbehaving peers.) Once you have tested your code against our code and the code of other students in an emulation, we can then try connecting them all together on the IPAQ's. Since debugging on the IPAQ's is difficult, you will want to do everything you can to get any bugs out before you reach that point.

IPAQs

Instructions for dealing with the IPAQs can be found on the class web page. Once you have a working, interoperable program you are ready to join the live network on the IPAQs. There are three steps. First, make sure your IPAQ has its 802.11 card in managed mode so that it is reachable from the CSE Lab machines via the building wireless network. Second, copy the fishnet directory including all the ruby files to your IPAQ using a file transfer utility (e.g., scp). Finally, to join the live class Fishnet you must switch the 802.11 card on your IPAQ to ad-hoc mode so that it can communicate with other IPAQs rather than with the rest of the Internet. Once this is done, simply run your node program in the same vicinity as someone else's IPAQ and see what happens!