Fishnet Assignment 3: Reliable Transport

Due: Friday Nov 15, 2002 at the beginning of class. Out: Monday Oct 28, 2002.
CSE/EE461 Autumn 2002; Wetherall.

The purpose of this assignment is for you to understand and implement a protocol for reliable transport, your own version of TCP. You do this in two phases. First, you develop a transport protocol within your node, along with a command to transfer data that you can use to test the protocol. The rest of your node implements the routing, flooding and naming functionality that you have already built up. Second, you retarget your implementation to the Amphibian application interface to allow applications that run in other processes to use your protocol to transfer data. We provide you with a test application. Make sure you obtain a new Fishnet distribution before you start this assignment.

1 Reliable Transport

Your job is to design and implement a transport protocol with the following features:

- *Reliability*. Each packet should be transmitted reliably by using sequence number and acknowledgement number fields and timeouts and retransmissions. The sequence number advances in terms of bytes of data that are sent. The acknowledgement number operates as in TCP to give the next expected in-order sequence number, and an acknowledgement should be sent every time that a data packet is received. That is, the acknowledgement number does not increase when a packet has been lost until that packet has been retransmitted and received. You can use constants in fish.h for the value of the timeout and the maximum number of times to try retransmitting before giving up. We suggest that you initially implement a "stop and wait" style scheme where only one packet can be transmitted at a time. This is all that is required. A better implementation would use a fixed size sliding window of, say, 4 packets. An excellent implementation would use a dynamically sized sliding window depending on acknowledgements and loss, mimicking the operation of TCP. We consider the latter challenging – please let us know if you are able to achieve this.

- *Connection state machine*. A connection is identified by a four-tuple, the combination of a source and destination fishnet address plus a source and destination port value. Each connection should be established before data is transferred, and torn down after all data has been transferred. The SYN flag on a packet sent from A to B is used by A to request that the half of the connection from A to B be established. That half is established when B acknowledges the SYN packet sent by A. The half of the connection in the reverse direction is established similarly. Similarly, The FIN flag on a packet sent from A to B is used by A to request that the half of the connection from A to B be torn down. That half is torn down when B acknowledges the FIN packet sent by A. The half of the connection in the reverse direction is torn down similarly. Finally, the RST flag on a packet sent from A to B is used to abruptly abort the connection in case of error. It should also be used to indicate "connection refused"

when there is no application awaiting connections on the destination port. A good implementation will support multiple, concurrent connections. We suggest that you define your own transport connection structure, with your node having an array of such structures, one per connection in use. The structure will encode all state associated with a connection, including sequence numbers, buffered data (both sent awaiting acknowledgment and possible retransmission, and received awaiting processing by the application), and connection state (such as established, the SYN has been sent but not acknowledged, etc.).

- *Flow control*. Your protocol should use the advertised window field along with the sequence and acknowledgement numbers to implement flow control so that a fast sender will not overwhelm a slow receiver. The advertised window field tells the other end how much buffer space is available to hold data that has not yet been consumed by the application. Note that this will not be much of an issue until you reach the second part of the assignment, as the in-node protocol may process data immediately, as it arrives, rather than having to buffer it until a separate application is ready to receive it.

You must use only packets of type `struct transport_packet` to build this protocol. As usual, you should strive to come up with a design that will interoperate with other students' nodes.

We suggest that you take the following steps to ease the development of your protocol:

1. Build a "transfer" command into your node that, on the sender side, sends sets up a connection to another node and sends a well-known test pattern to the other side, and tears down the connection. On the receiver side, your node should check that the test pattern is expected and provide feedback about the success or failure of the overall transfer. This command is purely an expedient way to test your transport protocol. An example of a transfer pattern is a configurable number of fixed sized packets, say with 512 bytes of data, whose contents are all bytes with values of N for the Nth packet, e.g., all 1s for the first packet, 2s for the second, etc., and using a specific port, say 1 for both source and destination.

2. At the sender and receiver, print the following single letter codes, without a newline, when a packet of the appropriate type is sent or received:
    - "`S`" for a SYN packet
    - "`F`" for a FIN packet
    - "`.`" for a regular data packet
    - "`!`" for a retransmission at the sender or duplicate at the receiver
    - "`:`" for an acknowledgement packet that advances the acknowledgement field
    - "`?`" for an acknowledgement packet that does not advance the field

    These codes will give you visual feedback to help you gauge the progress of a transfer. You should read about and call `flush()` after printing single letter codes so

that they appear on the console without delay. If you print the codes as specified above, a successful connection will appear as a sequence of mostly dot characters marching over your screen.

3. Run your Fishnet with a relatively high level of packet loss (10%, say) to check that lost data is successfully retransmitted. Packet loss is a command-line option to fishhead.

2 Amphibian

For the second half of this assignment, your goal is to modify your transport protocol so that it can be used by applications that run on your IPAQ but outside of your Fishnet node. This is necessary because applications that send data are not usually implemented in the same program as the transport protocol itself in the way that you have a command to send a test pattern as part of your Fishnet node. Rather, Web servers and so forth run in separate processes than the kernel, where TCP/IP is implemented.

Specifically, your job is to get the provided fishperf application running using your transport protocol. To do this, you will need to learn about Amphibian, a new part of Fishnet that provides an interface between applications and protocols. Amphibian works differently when used in applications and inside a Fishnet node. You only need to write code using Amphibian inside your Fishnet node, but you need to understand both uses to do so.

Applications include the header file amphibian_app.h and link against libamphibian_app. They use a faux "socket" API to access the Fishnet, e.g., fish_socket() instead of socket(). Sockets are the API use by real applications such as Web servers to exchange data across the network. We have provided you with fishperf in the distribution, an application that uses Amphibian to send data between Fishnet nodes and times the transfer to see how well the transport protocol is working. You can look at its source code. You will use fishperf to test your transport protocol when you have completed it, rather than the in-node transfer of a test pattern.

To make fishperf work, Fishnet nodes need to do three things. First, include the header file amphibian.h to access Amphibian functionality (but you still link against libfish, as before). Second, register callbacks with Amphibian so that it knows which function to call when applications open connections, send data, close connections, and so forth. Third, call other Amphibian functions directly to initialize Amphibian, deliver data to a waiting application, complete a waiting connection, and so forth.

You should proceed using the following steps:

- Read amphibian.h to learn more about the API. As with fish.h, the file amphibian.h is the definitive version of the Amphibian in-node API.

- Look at your test pattern command. Wherever that command calls or is called by your transport protocol code there is an interaction between an application and your transport protocol that you will need to revise to use Amphibian.

- Revise your code until you can run the fishperf program and use it to drive your transport protocol. Note that this will probably cause your "transfer" command to cease working, which is fine. However, leave the single-letter printouts in the transport protocol for turn-in.

3 Discussion Questions

- Your transport protocol implementation picks an initial sequence number when establishing a new connection. This might be 1, or it could be a random value. Which is better, and why?
- Your transport protocol implementation picks the size of a buffer for received data that is used as part of flow control. How large should this buffer be, and why?

4 Turn In

Turn-in all of your source code for your entire node electronically. It should make using the target hw3. Bring a printed copy of your transport protocol code to class (you don't need to include the routing and naming code from earlier assignments). Bring a printout of the fishperf application running using your nodes on a two node private fishnet, plus answers to the discussion questions.

—END—