Fishnet Assignment 3: Reliable Transport

Out: Friday, February 8, 2002.
Due: Tuesday, February 26, 2002.
CSE/EE461 Winter 2002; Anderson.

In this assignment, you will work in teams of two to develop a Fishnet node that reliably transfers files to other nodes. The program you write builds on your solution so far. The goal of this assignment is for you to understand reliable transport.

START THIS ASSIGNMENT EARLY. Bugs are easy to write and challenging to understand.

1 What You Need To Write

Write a C program called hw3.c that implements a file transfer application on top of a bi-directional transport protocol, as described below. Continue to bear in mind the Robustness Principle: "Be conservative in what you send and liberal in what you accept." This specification may leave some points ambiguous; do what you think is best as long as your program can interoperate with the sample solution and other nodes, and document the design decisions you make.

- Builds on top of functionality from the first two assignments:
  - Takes three command line arguments as before, joins the Fishnet, performs the tasks below in any order, and runs until you type "exit", when libfish.a will end the program.
  - Accepts keyboard input commands of the form "`send <nnn> <message>`" and implements the Fishnet echo protocol, which is useful for testing.
  - Maintains an up-to-date routing table via the distance vector protocol and forwards packets using the routing table.
- Waits to get a line of input from the keyboard of the form "`put <nnn> <filename>`", then reliably transfers the contents of the file across the Fishnet to the destination node, where it is written to a file. To stress reliability mechanisms, the Fishnet in which you run will drop a significant proportion of packets. Successful transfer requires that you implement three pieces of functionality, each of which are described below: reliability using timers and retransmissions; connection setup and teardown; and a simple file transfer application. You should use transport packets as defined in fish.h for all of these tasks.
- *Reliability.* You will implement a reliable, bi-directional connection with the FISH_TRANSPORT_PROTOCOL. Reliable transmission should be achieved for each data packet within a connection, including those with SYN or FIN set (see below), by acknowledgement and retransmission.
  - *Sequence numbers.* Each packet that contains data or has the SYN or FIN flag set should have a non-zero sequence number. The setup protocol given below allows any initial sequence number, e.g., one or a random number, to appear in SYN

packets. Afterwards, each packet that is not a retransmission should have its sequence number incremented. (Don't worry about sequence number wrapping.)

  - *Sending data.* A sender should have only one data packet outstanding at a time (i.e., stop-and-wait). After the first packet, don't send a packet with a new sequence number until the previous one has been acknowledged.
  - *Acknowledgements.* Receivers should acknowledge every data packet that is part of a connection or that establishes a new connection, even if it is out-of-order or a duplicate. The acknowledgement number should always be the sequence number of the last **in-order** packet received.
  - *Retransmission and timers.* While sending data, a retransmission timer of RETRANSMIT_TIMEOUT seconds should be kept for each unacknowledged packet, and the packet should be re-sent verbatim if the timer fires. If MAX_RETRANSMIT copies of a packet have been unsuccessfully retransmitted, then the connection should be aborted by deleting the connection state. If a connection has seen no activity for IDLE_TIMEOUT seconds, then the connection should be similarly aborted. (Note that you can cancel timers using `fish_cancelevent`.)
  - *Sliding window.* The rules described above for stop-and-wait reliability are equivalent to using a sliding window of 1 packet. However, the acknowledgement rules are defined in such a way that your sender can support a larger sliding window of 8 packets (say) without any change in the receiver code. You may choose to make your sender use a larger sliding window, with or without the fast retransmit feature, and see how much faster it is. You may also allow the receiver to buffer a finite range of out-of-order packets.

- *Connections*. A connection is identified by the four-tuple of source and destination address plus source and destination port. Your application should be able to maintain up to MAX_CONNECTIONS simultaneous connections. Both the client (which initiates a connection) and the server (which accepts a connection) should be implemented within a single program.
  - *Connection setup.* Connection setup involves a three-way handshake. To begin a connection, the client should send a packet with the SYN flag turned on and wait for it to be acknowledged; all other packets in this connection should have this flag turned off. The server, upon receiving a packet with the SYN flag on for a destination port for which there is an application, should establish the connection state and reply with a packet that has the SYN flag set and acknowledges the client's SYN packet. The client should then acknowledge the server's SYN packet. Both SYN packets should establish an initial sequence number. (These may be different, but should both be greater than zero.)
  - *Connection teardown.* To implement connection teardown, the client or server should send a packet with the FIN flag turned on and wait for it to be acknowledged. When it later receives a packet with the FIN flag set, it should acknowledge the packet and wait IDLE_TIMEOUT seconds before clearing the connection state (in case the acknowledgement is lost and the FIN retransmitted).

Upon receiving a packet with the FIN flag set and the expected sequence number for an established connection, the client or server should acknowledge the packet and send FIN if it has no more data to send. After sending FIN, the program should wait for the packet to be acknowledged before clearing the connection state.

- *File transfer.* The file transfer application should read from the local file named in the put command described above, transfer its contents over the network, and write them to a local output file. The sender should transmit the null-terminated string "`put <filename>`" as the data of the SYN packet. The receiver should open the output file named by the source node address of the connection, followed by a dash, followed by the name of the file being transferred, e.g., "23-myfile.txt". The remaining packets in the connection should carry the contents of the file in order. The input and output file are closed when the connection is torn down. The FIN packet should not include data. (These rules about SYN and FIN packets carrying data are arbitrary, chosen only because we have a simple implementation in mind.) File transfer packets should be sent with destination port 20 (FTP Data) and a source port, 1000 or above, chosen by the sender.
    - Note that file transfer can be a significant security vulnerability, and you should take the following steps to avoid compromises. Do not send or accept filenames containing the character '/'. Make sure the permissions of the written file do not allow it to be executed. (A good set of permissions is 0422: user read-write, all others read-only). Write the file truncating any existing copy, e.g. using the `creat` system call. If you cannot open the file for writing, then you should consider the connection to be closed at the receiver (and the sender will soon time out and abort its connection).
    - Your implementation shouldn't care whether the file being transferred is text or binary. So, don't use functions like `strcpy` to manipulate data read from the file; use `memcpy` instead.
- Your program should print the following:
    - When a transfer is started, print "`PUT <filename>\n`" at the receiver. (At the sender, you will see the "`put <filename>`" command that you typed.) At the sender and receiver, print `"\n<xxx> bytes\n"` when the transfer is complete, showing how much data was transferred. At the sender, you should not print this message until the receiver has acknowledged the FIN packet. Then, you will know that the file has been successfully transferred when you see this message and the number of bytes matches the length of the file.
    - Print the following character codes using `fish_debugchar` with debug level `FISH_DEBUG_TRANSPORT` when a packet is sent or received:

Event	Sent	Received
SYN	`S`	`S`

FIN	F	F
Data packet	.	,
Acknowledgement	:	;
Retransmission or duplicate	!	*
Out-of-order		?
Connection state deleted	x	X

- ❑ Although you may wish to use the default debugging level (FISH_DEBUG_ALL) while debugging your program, use `fish_setdebuglevel` to set it to FISH_DEBUG_TRANSPORT before turning in your program. With debugging for routing disabled, you can use the "send" command to determine when a route between the client and server has been established.

- ▪ You may find the following system calls and library functions helpful in your implementation: memset, memcpy, strchr, sprintf, isprint, random, open(2) (do "`man 2 open`" to read the manpage), creat(2), read(2), write(2), and perror. Feel free to discuss them on the mailing list.

2 Step-by-Step Development and Test Instructions

Here is a suggested set of steps to develop the required functionality.

0. Start with hw2.c by copying it to the new file hw3.c
1. Run the fishhead with high loss (try "--help" or just "--loss 0.2") to stress the reliability mechanisms.
2. Code the "put" command syntax, but rather than sending the file, just send a series of dummy text packets reliably using acknowledgements and retransmissions and print the text to the console on the receiver. Rather than using full connection setup and teardown just yet, let the first received packet start the connection and the idle timer terminate the connection. Test between two nodes on your local fishnet.
3. Add connection setup on the sender and receiver, adding the relevant printouts, and test a dummy transfer. Then implement and test connection teardown. (The TCP state transition diagram on page 381 of Peterson and Davie may help you understand how to implement connection setup and teardown.)
4. Add the file transfer code by letting the SYN packet carry the filename, reading from the input file at the sender and writing to the output file at the receiver, keeping track of bytes transferred, and printing the remaining messages. In case your implementation is buggy at first, you may want to test by sending a file you don't particularly care about. Test with both text files and binary files (e.g., a picture or a copy of your executable). You can use the Unix `diff` command to verify that the received file is the same as the sent file.

5. If you haven't done so already, test your program for interoperability with the class reference solution. Then try joining the class Fishnet and transferring some files to nodes run by the TAs or your friends.
6. You're done! Read and do any turnin work now.

3 Turn In and Discussion Questions

Submit your source file(s) and the modified Makefile, if needed, using the turnin program. Hand in a paper copy of the discussion questions and test cases below as well as your source code.

1. Join the class Fishnet at `jimbo:7777` and use the `put` command to send a file to node 1. This file should have your username in its name, it should be at least 2KB, and it should be something you want other people to look at. It will be posted to a collaborative web site, which will be linked to from the main course web page.
   This transfer is the turnin test case. Save the output and print it for us. (Please use FISH_DEBUG_TRANSPORT for the debug level, as described above, to save paper and make the output easier for us to read.)
2. Leave a node running on the class Fishnet, as in the previous two assignments. You may want to run this node in a directory by itself (with no other files in it) so you can easily see what has been sent to you.
3. What difference (advantage or disadvantage in terms of reliability) does having a SYN flag make compared to simply using a sequence number of one to signify the start of a new connection?
4. What difference (advantage or disadvantage in terms of reliability) does having a FIN flag make compared to simply not signaling FIN and letting old state be deleted by the idle timeout?
5. In what ways does the system you have built fall short of perfect reliable file transfer? You should assume that the data in packets is protected by a 32 bit checksum while in transit (and this is in fact the case).