Fishnet 3: Reliable Transport (CSEP561; Spring 2005)

Due: Tue May 10 in class. (Out 4/26/05).

Your assignment is to design and implement a protocol for reliable transport, i.e., a simplified version of TCP. Your code will build on the code you wrote for previous assignments to do naming, flooding, and routing. The final assignment in the quarter will be to develop applications that use your Fishnet code.

1 Reliable Transport

Your job is to design and implement a transport protocol with the following features. For transport packets, use the `TRANSPORT_PKT` protocol number; the packet payload should be a packed object of class `Transport`.

- *Multiple connections*. A connection is identified by a four-tuple, the combination of a source and destination Fishnet address plus a source and destination port value. You must 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.).
- *Connection setup/teardown*. Each connection is setup before data is transferred and torn down after data has been transferred using a connection state machine that is considerably simpler than the one described in Peterson for TCP:
    - Connections are **one-way byte streams**, not two-way as in TCP. (Of course, you can easily build two-way byte streams on top of two one-way byte streams.)
    - **Setup** is simple. The sender opens the connection by sending a packet with the SYN flag set and some initial sequence number. The receiver then replies with an acknowledgment packet (with the ACK flag set). The transfer can then proceed.
    - **Teardown** is also simple. Either side closes the connection by sending a packet with the FIN flag set. FIN should also be used to indicate "connection refused" when there is no application awaiting connections on the destination port.
    - Unlike in TCP, packets with the SYN, ACK, or FIN flag set **do not** carry payload data.
- *Reliability and Sliding Window*. Each payload packet should be transmitted reliably by using sequence/acknowledgment number field and timeouts and retransmissions.
    - The **sequence number** advances in terms of bytes of data that are sent. Note that this sequence number is separate from and semantically different from the sequence number in the basic packet header.
    - 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. (Note that since an ACK never carries data, we put the acknowledgment number in the Transport header sequence number field.)
- *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 in this assignment (as the in-node protocol may process in-order data immediately, as it arrives, rather than having to buffer it until a separate application is ready to receive it) but it must work and will be needed by the next assignment.

As usual, you should strive to come up with a design that will interoperate with other students' nodes. Take the following steps:

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. For the transfer pattern you should use a configurable number of fixed sized packets, 512 bytes by default, 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. We will define destination port 1 to be the well-known test port.
2. Get something simple running first! We recommend that you implement a "**stop and wait**" style scheme, where only a single packet can be outstanding at a time, before a **fixed-size sliding window**. Another great simplification is to use retransmission instead of buffering: much of the complexity of buffer management at the receiver is for the case of out-of-order data following a loss. Instead of buffering this data, you could discard it, and the sender should retransmit it eventually.
3. Keep future applications in mind! You will soon build an application that uses your transport protocol. It will call into your transport code, e.g., to request that a connection be setup, data transferred, and the connection closed. Your transport code will call back into your application code when these actions are completed. If you structure your "transfer application" accordingly then it will be easy to add another application to your codebase.
4. 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 and give us a trace for your turnin. If you print the codes as specified above, a successful connection will appear as a sequence of mostly dot characters marching across your screen.
5. Test your Fishnet with a relatively high level of packet loss (5%, say) to check that lost data is successfully retransmitted. Packet loss on a given link can be specified in the topology file provided to the simulator or trawler (for example: edge 0 1 loss: 0.05 bw: 10 delay: 5) to establish a link between node 0 and 1, with a 5% loss rate, 10KB/s bandwidth, and 5 millisecond propagation delay. You can also generate sample topologies using topogen's "-l" flag.

2 Discussion Questions

a) 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?

b) 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?

c) Our connection setup protocol is vulnerable to the following attack. The attacker sends a large number of connection request (SYN) packets to a particular node, but never sends any data. (This is called a SYN flood.) What happens to your implementation if it were attacked in this way? How might you have designed the initial handshake protocol (or the protocol implementation) differently to be more robust to this attack?

3 Turn In

Turn in electronic and paper material as follows.

1. Run a two-node fishnet emulation with your node and our reference node. Perform a reliable transfer of at least 100 packets. Capture the output and mark it up to tell us what is going on. (It's fine if the output includes only your commands and the "SF.!:?" characters as described above.)
2. Use the turnin program to electronically submit one or more Ruby files containing the source code of your solution.
3. In class on the due date, hand in one stapled paper write up, with both partner's names on it, containing:
    a. A brief design document.
    b. A printout of your transport protocol code. (You don't need to include the flooding, routing, and naming code from earlier assignments.)
    c. A printout of the output we have asked you to capture.
    d. Short answers to the discussion questions.