CS433/533 Assignment Four: Reliable Transport and Congestion Control

Your assignment is to design and implement a protocol for reliable transport and congestion control, that is, a simplified version of TCP. This assignment consists of two perspectives: reliability and congestion control. We give out the two perspectives at the same time so that you may anticipate futures to your code.

- Some clarification points.
- Sample test.
- Rubric (see Grading guidelines at the end)
- Due
  - **Part 1: Design discussion with instructor or a TF: Nov. 11; Code check point: 11:55 pm, Nov. 15**
  - **Part 2: Design discussion with instructor or a TF: Nov. 14; Code and report: 11:55 pm, Nov. 19.**

General Introduction to Fishnet

For this assignment, we will usw the "Fishnet" software project infrastructure. Here is some general introduction to Fishnet:

- Fishnet introduction
- Modified Fishnet code distribution with link model(v1.8)
- Fishnet Java API documentation, which you can also download for offline use.
- A set of slides on the assignment (ppt, pdf)

Fishnet with link model

The original Fishnet distribution has limited simulation/emulation of buffering and loss characteristics of physical links. In simulation mode, only loss is simulated, and a link effectively has infinite buffer space. In emulation mode, there is no emulated buffering and loss. Therefore, if you run the emulation on zoo machines, you will almost experience no loss of packets. In order to help you stress test your implementation, we have modified Fishnet to provide simulation/emulation of buffering and loss characteristics of physical links. Please use the links above to download the new code distribution and API documentation.

The following is a summary of changes we have made to Fishnet:

- The unit used in the "edge" command to specify bandwidth has been changed to "B/s", instead of "KB/s". The default value is still 10KB/s. Please update the "edge" command "bw" options in your topology file if there are any. If you notice that your TCP throughput drops dramatically after switching to the new Fishnet, please check if you have updated your "bw" options in the "edge" commands.
- The "edge" command now supports an additional option "bt", which specifies the buffering capacity of the edge, in terms of buffering time in milliseconds. For example, suppose the bandwidth of an edge is 10KB/s, "bt 100" means the edge can buffer 100 milliseconds of data sent at 10KB/s, i.e.,

1000 bytes. The default buffering capacity is 250 milliseconds.
- The simulator now takes buffering capacity into account, and will drop packets when buffer overflows.
- The emulator now emulates both buffering and loss.
- The simulator/emulator will, on exit, print out statistics about dropped packets (due to buffer overflow) and lost packets (due to loss).

Part 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. The packet payload should be a packed object of class Transport.

Connection setup/teardown: A connection is identified by a four-tuple, the combination of a source and destination fishnet address plus a source and destination port value. Our connection state machine is considerably simpler than the one described for TCP. Specifically:

- First, 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.)
- Second, we assume a simpler hand-shake mechanism. As with TCP, each connection should be established before data is transferred and torn down after all data has been transferred. The sender initially sends a connection request to the receiver, by picking an initial sequence number and setting the SYN flag. 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 never carry payload data.

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.).

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.
- 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.)
- Note that packets carry both a sequence number in the packet header (which is unique among all packets sent by this node), and a sequence number in every transport header (which is the place within the byte stream for this data or ACK packet). These two roles for sequence number are semantically distinct, and in fact, the IP header has its own unique identifier field separate from the sequence number in TCP's header.
- *We recommend that you first implement a "stop and wait" style scheme, where only a single packet*

can be outstanding at a time. Once that is working, implement a fixed-size sliding window.

Requirements

As usual, you should strive to come up with a design that will inter-operate with other students' nodes. As you do, take the following steps:

- Implement the Fishnet socket APIs defined in the provided code. To help you test your implementation, the code implements, using these APIs, a transfer client and a transfer server. The transfer client sets up a connection to a transfer server, sends a test pattern to the server, and tears down the connection upon completion. The transfer server will check that the test pattern is expected. A "transfer" command and a "server" command are implemented by a node to start a client and a server, respectively. You're free to build more test cases to verify your implementation of the socket APIs.
    - The APIs are defined in Java classes TCPSock (in TCPSock.java) and TCPManager (in TCPManager.java). Class TCPSock defines an API that handles operations on a single socket, such as read/write and close. Class TCPManager defines a single function - "socket()" - that creates a new socket, and is supposed to handle connection management and/or any book-keeping that involves more than one socket.
    - During your implementation, it might be helpful to take a look at the code for the transfer client and server, to better understand the semantics of the APIs. The code for the transfer client is in "TransferClient.java", and the code for the server is in "TransferServer.java".
    - *NOTE: All of the socket APIs must be non-blocking*. For example, if a call of "TCPSock.write" cannot write all of the data (or even no data at all!) due to some reasons, the call should write whatever it can at the current moment, and return with the number of bytes successfully written. This is due to the fact that Fishnet is an event-driven simulator/emulator; thus if a call blocks, it also blocks the whole simulator/emulator. Again, it may be helpful to take a look at the code for the transfer client and server, which are also implemented in a event-driven fashion.
    - The four APIs that query the state of a socket are simple enough that their implementation is already provided. They should work fine, and you may not need to modify them.
- 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.

- Run 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 (for example: edge 0 1 lossRate 0.05 delay 5 bw 10) 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.

Part 2: Flow Control and Congestion Control

Requirements:

- *Flow Control*: Your protocol should use the advertised window field along with the sequence and acknowledgment 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.
- *Congestion Control*: Design and implement AIMD congestion control. In addition, you will need to design some test cases to illustrate the behavior of your design.

Running Your Nodes in Fishnet

There are two ways to run your nodes: in the simulator (the 'simulate' mode) or in the emulator (the 'emulate' mode). Suppose your have implemented your "transfer" command with syntax "transfer addr port", where "addr" is the address of the receiver, and "port" is the port the receiver is listening to. The following shows two examples of potentially how to set up the Fishnet environment to run your nodes.

If you use the trawler, here is a topology file that contains only two connected node: two.topo. You can use this file when you run the trawler. Specifically, do the following:

- Open three new terminals
- In terminal 1, type: perl trawler.pl 8888 two.topo
- In terminal 2, type: perl fishnet.pl emulate localhost 8888 10000. You should see a line like the following, which tells you that this node has the address 0: Node 0 started
- In terminal 3, type: perl fishnet.pl emulate localhost 8888 10001. You should see a line like the following, which tells you that this node has the address 1: Node 1 started
- You can then proceed to test your nodes. For example, if node 0 listens on port 21 with backlog of 2, type "server 21 2" in terminal 2; to transfer 50000 bytes from node 1 port 40 to node 0 port 21, type "transfer 0 21 40 50000" in terminal 3.

If you use the simulator, there is a sample test script "transfertest.fish" under the "scripts" directory. You can run it using the command "perl fishnet.pl simulate 2 scripts/transfertest.fish". If you would like to know more details about the built-in commands and their syntax, please refer to the documentation at the beginning of CommandsParser.java.

FAQ

Here are some frequently asked questions and answers for this assignment:

Q: Are all source and destination ports always 1, or are they variable and must be specified in the various commands that you implement?

A: You should support arbitrary port numbers. Please note that the "transfer" command example outlined in the preceding description is only a test scenario suggested for you to test your design.

Q: Are we supposed to support multiple concurrent connections between the same 2 nodes in the same 1-way direction?

A: Yes.

Q: Must a server be listening first before a client can connect to it? Do we need a separate command to setup a server?

A: Yes, a server must be listening on a port before a client can connect to it. However, a separate command to setup a server is not strictly required. You may arrange your server to listen on a default port number. However, you're encouraged to implement such a command. Please clearly document your design decision when you submit your assignment.

Q: Should timeout values be constant or variable?

A: Timeout values should be variable. Please refer to the lecture slides for how to determine timeout values.

Q: Do we have to implement slow start?

A: No, slow start is optional.

Q: Should the sender and the reciever both print a packet in their trace

A: Yes.

Submission

Please submit using classv2 server. Turn in electronic and paper material as follows.

- Run a two-node fishnet emulation. Perform a reliable transfer of at least 100 packets. Capture the output (from both the sending node and the receiving node) 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.)
- You submission should include:
  - A brief design document.
  - A document containing any output we have asked you to capture.
  - Short answers to the discussion questions below: Diss1a, Diss1b, and Diss1c for the first submission; Diss2 for the second submission.

Discussion Questions

1. *Diss1a*: 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?
2. *Diss1b*: 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. *Diss1c*: What happens in your implementation when a sender transfers data but never closes a connection? (This is called a FIN attack.) How might we design the protocol differently to better handle this case?
4. *Diss2*: 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?

Grading guidelines

Each part of the project is graded on a 5 point (0-4) scale, multiplied by the weight of the part. The weighted grades from all parts of the project are added together to produce the final grade. There is an extra credit component to this project.

The five point scale is based on how well you show your understanding of the problem, and in case of code how well your implementation works:

0 - nothing turned in
1 - show minimal understanding of the problem / most things don't work
2 - show some understanding of the problem / some things work
3 - show a pretty good understanding of the problem / most things work
4 - show excellent understanding of the problem / everything works

The weights for the parts are:

Transport layer implementation: 1/2 of total grade
Write-up of your design decisions: 1/4 of total grade
Answers to discussion questions: 1/8 of total grade
Print-out of captured output: 1/8 of total grade

Last Update: October. 25, 2013 08:56:24 PM -0400