JSTP: Jack-Steven Transfer Protocol

Jack Smith (smit2395) & Steven Kneiser (kneisers)

April 27, 2017

Section 1: Getting Started

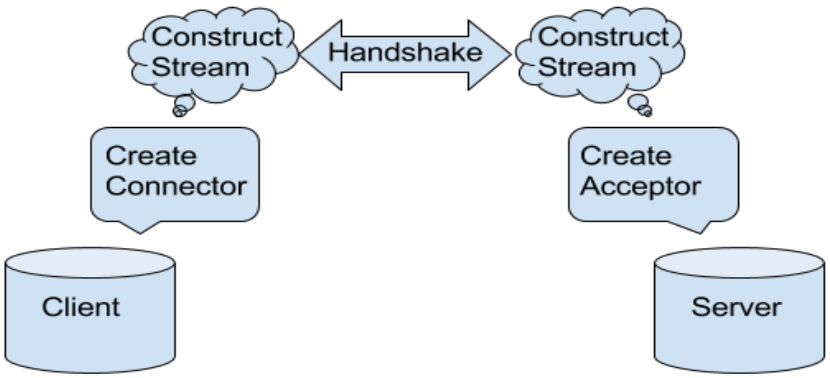
Navigate to the root of the project directory. Here you are going to build the client & server executables:

```
$ make
```

Next you can either append the project **bin/** to your **PATH** or you can simply move inside it for simplicity. There you can run the two executables with the following commands, where *P* is the probability of packet loss & *Cwnd* is the window size as measured in bytes:

```
$ cd ./bin
$ ./server port_number Cwnd P
$ ./client server_hostname port_number filename P
```

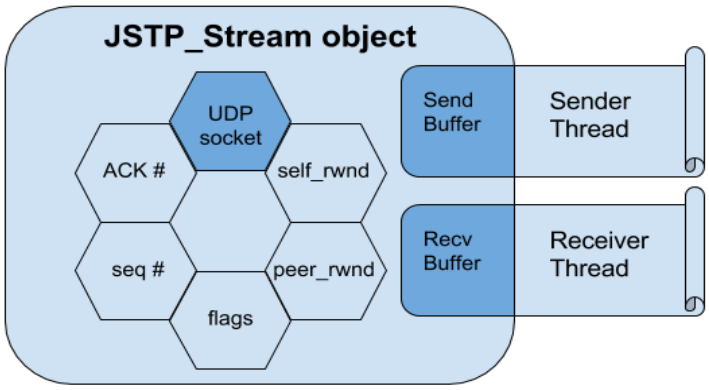
Section 2: Design



2.1 Protocol Overview

- All data packets are also ACK packets
- The client & server are fully symmetric once connected (full duplex)
- Can communicate with multiple clients since ports are uniquely re-assigned to each connection during initial handshake

2.2 JSTP_Stream object



Streams are constructed both from an acceptor on the server side and a connector on the client side. Over the course of the handshake, the server and client will discern an ephemeral port to exchange all future data over which frees up both of them to construct other streams.

User Interface:

- Constructed from Connector or Acceptor
- *Send()* adds data to the send buffer
- *Recv()* grabs data from the receive buffer

Run-time:

- Sender is by default sleeping
- When awaken, it sends as much data as it can, then returns to sleep
- (e.g., the receiver thread wakes the sender when new data is received)

2.3 JSTP_Segment object

A Segment is a digestible $\tilde{1}$ KB slice of a file that will sent over the network. They inherit serialize-able traits from an abstract base class for ease in sending over UDP sockets. Also, they come with the structured header:

- 32 bits: Sequence number
- 32 bits: ACK number
- 32 bits: Window size
- 16 bits: Flags

Other notable features:

- Sequences & ACKs are byte-oriented (like in TCP)
- This means a reasonable window size is large (1 KB or more)
- ACK number is the sequence number of the next expected byte
- Sequence number is the number of the first byte in the segment
- Out of order segments are discarded (waking the sender thread, thus forcing a duplicate ACK along with Go-Back-N behaviour)

2.4 UDP Socket Abstraction

This abstraction is worth mentioning for its simplification of both our loss simulation & interaction with the greater transport-layer functionality critical to the JSTP. Our UDP Socket abstraction helps to:

- Bind to local ports with *bind_local()* and *bind_local_only()*
- Send and receive any serialize-able object up to a configurable size limit (in this case, JSTP_Segment)
- Simulate loss on both ends along a Bernoulli distribution
- Save address of last peer (useful for handshake)

Section 3: Challenges

3.1 Big Design Up Front

Over the weeks, the JSTP design evolved as we fleshed out our implementation. At first, we cautiously crafted the JSTP_Segment so we could carefully partition entire files, and decided how best to encode/decode them. This week of implementing was as smooth as usual development on our familiar environments only with the daunting reality that we had not yet thought our solution entirely through.

3.2 C++0x

We began using git for version control to coordinate and, after grinding to a halt, had to painstakingly port our initial solution to the target environment which was proved surprisingly difficult despite it being the same one since Project 1. In reality, this project demanded much more intricate structures so we began this one, without hesitation, programming in the more batteries-included C++. However, some of the most convenient C++11 features we habitually took advantage of were nowhere to be found. Turns out this particular environment's compiler had come out not long before that standard and offered most of the features we were using, only under the yet-to-be-announced C++11 standard, *C++0x*. With that out of the way, all of our unsupported language-specific features required workarounds which dealt quite an irritating setback to the team.

3.3 Socket Programming

After polishing off the application-layer component to our protocol, we got to work constructing the underbelly, that is the more generalized transport-layer component to the JSTP. The next two weeks were predominately spent studying other implementations of data transfer at this layer. Only once we came across the Acceptor-Connector design pattern in networking could we finally had the knowledge brainstorm the rest of our architecture.

3.4 Playing Nice with Multiple Threads

While we had worked with primitive socket I/O in C for Project 1, our ambitious transport-layer component was fully symmetrical and required much more tedious use of *socketpairs*. This comes with not shortage along of headaches associated with isolating operations between threads and synchronizing this fluid two-way communication over the network. This was the week our demo began to take form.

With another productive week of research and working alongside the compiler, we overcome most of our complications. Ultimately, this packaged implementation of JSTP not only operates precisely as envisioned, but also transfers data with the performance to show for our close attention to detail.