Hail

Stella Chung (SID 004277565) Ky-Cuong Huynh (SID 204269084)

Stella handled `server.c` and Ky-Cuong hanled `client.c` . All other work was split equally.

Hail is a transport-layer protocol for reliable data transfer, implemented atop UDP.

Hail Protocol Design

Hail is based upon TCP, one of the twin backbone protocols of the modern Internet.

At a high level, we:

- Try to establish a connection, with a maximum number of attempts to do so.
- Break-up any data to be sent into chunks, called *packets*, then send them to another host. We send up to *flight-window* size many packets at a time.
- The receiver reorders received packets within a buffer before writing the data back to disk as a file.
- The receiver positively acknowledges received content.
- The sender retransmits if they do not receive the acknowledgement after a set time-out interval.

Segment Format

A Hail packet has the following structure:

```
[ Sequence number         ]  1 byte
[ Acknowledgement number  ]  1 byte
[ Control code            ]  1 byte
[ Hail version            ]  1 byte
[ File size               ]  8 bytes
[ File data               ]  500 bytes
```

The total size of any Hail network packet is 512 bytes. Having a power-of-2 size allows for easy alignment in memory. There is no padding between fields or before/after the packet. While the above is aligned on systems with a 4-byte (32-bit) word-size, it is not aligned for systems with a 8-byte (64-bit) word-size. To prevent automatic padding, we specify that the struct be "packed" instead (see the `hail.h` for details).

An explanation of the field design decisions:

- Sequence number: this is the most-accessed field and so placed first. It identifies the order of Hail packets as originally sent. It wraps around upon hitting 255, which gives an internal limit of 256 * 500 bytes == 128 MB on file size before this edge case is encountered and handled.

- Acknowledgement number: a receiver echoes a sequence number as its acknowledgement. A sender knows to retransmit a particular packet upon not seeing its echoed sequence number after some time-out interval.

- Control code: our control codes mark a packet as being internal to the protocol's functioning. They carry no data, and their functions are explained in the relevant sections. Up to 256 such control codes can exist, providing for the standard's extensibility.
    - OK
    - SYN
    - SYN ACK
    - ACK
    - CLOSE
- Hail version: designed for graceful backwards compatibility handling, the version (from 0 to 255) lets hosts decide what level of features are available. As Hail is still in an alpha stage, the version is kept at 0 for now.
- File size: this is simply an unsigned integer, 8 bytes in size to match the size limit of 2 EiB (exbibytes, 2^64 bits, or approximately 2.3 exabytes for those who prefer base-10) for individual files on most operating systems
- File data: these are the contents of a file transported by the Hail packet. As these are simply raw bytes, applications can apply their own higher-level logic

Note that we assume octet bytes, i.e., each is 8 bits in size. When sent, this struct is packed into a buffer (marshalling/serialized) and unpacked (unmarshalled/deserialized) at the destination.

Connection Establishment

Hail opens with a three-way handshake to negotiate a connection. At the moment, both hosts only confirm that the version number is between 0 and 2, reserved for the development versions of Hail. The handshake begins when a sender sends a packet with the `SYN` code set (synchronize). The receiver replies with `SYN ACK` (synchronize acknowledge), the sender acknowledges with `ACK`.

The time-out interval for any of the handshake messages is 50 ms. A maximum of 3 retransmission attempts is made at each step. If any step fails to receive acknowledgement for 3 time-outs, then the other host is deemed unreachable, and an error message will be emitted.

Chunking into Packets

The receiver allocates a reordering buffer as large as the file to be received, based on the file-size reported in any of the incoming packets. Later, we can enhance MiniFTP with streaming writes to disk (where we write a run of sequence numbers to disk before moving to the next one), error checks for disagreement on file size, etc.

NOTE: We will need streaming writes to disk anyway for large files, as the heap size is limited to ~1-4 MB on most systems, and even a 1-byte sequence number gives us:

```
256 packets * 500 bytes/packet == 128 MB filesize send limit
```

The sender sends a chunk of the file at a time. It packs these chunks into Hail packets and fires them off. It can do so again for more of the file up until the *flight window* limit is hit. The flight window limits how many packets may be "in-flight" on the network at any one time. Once the limit is hit, the sender must receive acknowledgements of previously sent packets before they send on new ones. This allows the sender to easily retransmit any lost or corrupted packets from the most recent window.

The overall result is a *sliding window mechanism*, where the sender and receiver are tightly coupled together. Imagine the file to be sent as already being places into its constituent packets. We can view these a steady sequence of numbered packets:

```
[ 0 1 2 ] 3 4 5 ... 255
```

Where `[` and `]` mark our flight window boundaries.

Our sequence number above (`seq_num` in the code) is 1 byte in size, so that gives us `2^8 == 256` possible values, `0-255` . We'll say a *run* of these number is sending enough packets that we use up all of the numbers in this range. For the first run, the receiver simply takes the contents of the received packet and slots it into the right spot in its reordering buffer. It does this based on the packet's sequence number. For the first run we go through, we just index based on sequence number, like so:

```
reorder_buffer[0 through 255]
```

where reorder_buffer is of (integral) size `ceil(file_size / HAIL_CONTENT_SIZE)` and of type `char* buffer` (since it must be dynamically alloated). The code for the contents being loaded looks like this:

```
reorder_buffer[unpacked_packet.seq_num] = unpacked_packet.contents
```

Sequence Number Wraparound

But what happens if our file is large, and we need more than 256 packets to send it? Our sequence number will inevitably wraparound (as we're limited in the size of its underlying data type), and we must handle this.

The key realization is that we can detect wraparound on the receiver side, and that the window size limits the circumstances under which this occurs.

After the first run of 256 packets, if we see another ‘0’ sequence number on the receiver side, we know that the sender has wrapped around. We are then possibly going to receive packets from the next run of sequence numbers.

So how can we tell if we are getting packets from the first 0-255 or the next 0-255? After all, they may arrive out of order, so what if we got two packets, one with number 127, and other with number 32, but from different runs? However, the finite flight/send window prevents this ambiguous overlap of ranges.

It limits what could possibly be in-flight at the time. For example, here’s the edge case just before we wrap around:

```
250 251 252 [ 253 254 255 ]
```

Then, upon ACK of 253:

```
251 252 253 [ 254 255 0 ]
```

So we know that anything 254-255 must be in the previous run of sequence numbers. And if we slide our window forward a bit a more:

```
[ 255 0 1 ] 2 3 4
```

Then now 255 must be in our previous run, and 0-1 in our next run, because we only have 3 packets in-flight at most at any time (if that were our window size).

So for the next run, we need to modify how we index into the reordering buffer. Instead of reorder_buffer[seq_num], we have:

```
reorder_buffer[(numRun * 256) + seq_num]
```

This in fact generalizes. We initialize numRun to 0, and so for our first run we have `reorder_nuffer[(0 * 256) + 0 thru 255]` , giving us the same indices as before.

But upon wraparound, we'll have:

```
reorder_buffer[(1 * 256) + 0 thru 255]
```

So our first index would be `reorder_buffer[256 + 0]` , which slots the first packet of the next run just after all of the last run, which is exactly what we wanted.

However, we have yet to handle the case where our window is still sliding over between the two runs.

During this straddling period, we cannot simply use the above indexing formula, or else we'd do something like `reorder_buffer[(1 * 256) + 255]` for the window:

```
[ 255 0 1 ] 2 3
```

(Actually, I think this is the only corner case, assuming we don't increment numRun until after encountering zero, which means 254 won't run into this)

That would cause us to place the last packet's contentes of the preceding run into the area reserved for the contents of the next run of packets. This is in fact the only special case. We can check for this. We use the first index formula if the left-edge of receive window is `SEQ_NUM_MAX` (255 here).

Sender-side Tracking

As for the sender's window, we just increment modulo the wraparound limit:

```
seq_num = ((seq_num + 1) % SEQ_NUM_MAX) + 1
```

(No '+ 1' if SEQ_NUM MAX is 256 instead of 255, but that's less clear for other use-cases of this constant).

Time-Outs

The server tracks the timestamps for sent files. Before it shifts the send window forward, it checks to make sure that all packets within the window have been acknowledged. If a packet has not been acknowledged and the difference between its associated timestamp and the current time is greater than or equal to 50 ms, then the packet is resent. The send window will not shift forward until that packet has also been acknowledged.

Closing the Connection

The server automatically closes the connection once all packets are sent and acknowledged. It then exits. The client automatically closes and exits after acknowledging all packets. The file has been received.

Difficulties Faced

Designing Hail

When we first began designing Hail, we had many design decisions to make:

- Would we use positive or negative acknowledgements? Both? Why? How?
- How would we coordinate a send/receive window across both sender and receiver?

And so on.

Importantly, it was hard to tell before begining what state we would need to track (and how). Thus, we both purposely and accidentally had too much tracked within Hail's structures. This led to wading through unnecessaryily messy code later while debugging.

We resolved this through careful removal of unneeded control codes, fields in Hail packets, etc. We had to recheck sending/receiving each time to ensure there were no regressions.

Generating Probabilities

Generating random numbers is a common task, but we hadn't had the need before to generate probabilities. We needed a way to take a random number and turn that into a probability in some way. Eventually, we realized that we needed to take a wider view, for a sequence of random numbers. Then, a random number between 0 and 100 is generated. We declare part of that range to correspond to "yes", and the inverse part to be "no". By checking if we are within the "yes" range, we have a probability.

Streaming Read/Write to/from Disk

Finally, we realized that we needed to handle files possibly not fitting in memory. We struggled with figuring out how to write only a portion at a time, and to coordinate which portion was written. Eventually, after careful diagramming and research, we realized that `lseek()` would come to the rescue. This allowed us to achieve streaming reads and writes on both server and client (respectively).

Conclusion

Project Hail was our most challenging project yet, but it brought us a much deeper understanding of networking.