Project 4

Reliable Streams

Tom Magrino

Slide heritage: Previous TAs → Krzysztof Ostrowski → Robert Escriva

Cornell CS 4411, October 19, 2010

Announcements

- Project 3 due Sunday at 11:59PM.
- Project 4 will be due November 3 at 11:59PM.
- Web page is being updated frequently; check for updates.
- Email `cs4410staff@systems.cs.cornell.edu` for help.

What we're looking for

Implementing TCP is non-trivial; instead, we've loosely specified a reliable stream protocol.

This protocol should co-exist with `minimsg`.

What is “reliable”?

What is “reliable”?

- Guarantee that if a packet is acknowledged (ACKed), it was delivered.
- Allows the user to know that there was a failure.
- Packets delivered *at least* once (not lost in the network).
- Packets delivered *at most* once (no duplicates).
- Guarantees are not absolute; it is sufficient to just detect errors.

What is “stream”?

What is “stream”?

- Connection-based (open, close, etc.).
- The user on each end should treat messages as a sequence of bytes.
- Message boundaries are an application-level concept.

What are does reliable streaming involve?

- Ordering: deliver in sequence (FIFO).
- Congestion control: a static window size is sufficient.
- Stream-like semantics:
  - User can send blocks of any size; minisockets should fragment the data.
  - Can ask to receive an arbitrary amount of data.

Relation to Project #3

- Unreliable and reliable protocols will co-exist.
- Code from Project 3 may be borrowed/reused where necessary ...
- ... but your code should be reasonably separate and isolated.
- The same network interrupt will be used to deliver both minimsg and miniports.
- It is up to you to modify the header so that you may *demultiplex* the network packets.

Base abstractions (both Project 3 and Project 4)

- Each protocol has the concept of a *port* as its endpoint.
- `network.h` is used by both.
- Ports are identified by number.

A typical example

1. Server listens for client connections.
2. Client connects by initiating a hand-shake.
3. Hand-shake completes.
4. Client and server can send data in any direction.
5. Client/server may explicitly close the connection.
6. No send or receive will succeed after a close.
7. Socket is destroyed at the end of communication; that is, a new socket must be created to repeat this process.

Reliability

- At least once:
  - Delivery confirmation: acknowledgement for every packet received (ACK).
  - Retransmission: failure to receive an ACK triggers a retransmission after timeout.
- At most once:
  - Delivery confirmation: acknowledgement for every packet received (ACK).
  - Retransmission: failure to receive an ACK triggers a retransmission after
- Control packets should be reliable as well (e.g. if the network duplicates a request to open a connection, the user should not see an error).

Stream-like semantics

- Ordering is achieved by placing sequence numbers in packets.
  - Buffering, variable window size, and duplicate suppression are used in TCP.
- Flow control is achieved by dropped packets.
  - TCP uses a dynamically changing window size that changes with available bandwidth.
- You may make the window size of minisockets equal to 1.
  - Significantly simplifies implementation.
  - Only one data packet is in transit at any given time (very slow).
  - Sequence numbers are still necessary to guarantee FIFO.

Fragmentation

- Cut the data into arbitrarily sized pieces.
- Assume that the sending application's boundaries are meaningless.
- Don't put "reassembling" information into the packets.
- Receiver will order the packets (by sequence number) and present it to the user as a continuous (potentially infinite) stream of bytes.

Receiving

- The receiver specifies an upper bound on the amount of data to receive.
- It is perfectly acceptable (and very common) for minisockets to provide fewer bytes.
- Any unconsumed data must be left for the next receiver.
- Because we are implementing a stream, *the exact amount of returned data does not matter*[*].
- Reconstructing messages is up to the client.

---

[*] Except if it exceeds the upper bound.

Concurrency

There is a one-to-one correspondence between server and client ports, but ...

- ... multiple threads can simultaneously send, and worse ...
- ... multiple threads can simultaneously receive.
  - The threads will need to be queued waiting on the socket.
  - Independent threads can receive random pieces of data.
  - It is up to the application to reassemble the pieces returned from concurrent reads.
- All control communication must be performed concurrent with all other communication.

Creating a socket

```
minisocket_t minisocket_server_create(
        int port,
        minisocket_error *error);
```

- The server is installed on a **specific port** (this may fail).
- Blocks pending a connection from a client.
- Returns a socket connected with a client.
- Simplification: one-to-one communication.
  - Only a single client may connect (further attempts will fail).
  - Once a client is connected, further connections are not allowed.

Connecting to a socket

```
minisocket_t minisocket_client_create(
        network_address_t addr,
        int port,
        minisocket_error *error);
```

- Connect to minisocket `port` on host `addr`.
- This may fail for reasons outlined above.
- Blocks until a successful connection is established (or it times out).

Sending and receiving

```
int minisocket_send(minisocket_t socket,
        minimsg_t msg,
        int len,
        minisocket_error *error);
int minisocket_receive(minisocket_t socket,
        minimsg_t msg,
        int max_len,
        minisocket_error *error);
```

- These block until the data has been ACKnowledged.

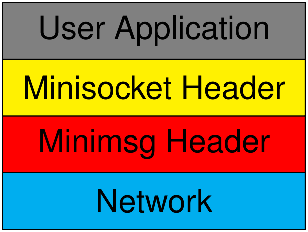
```
void minisocket_close(minisocket_t socket);
```

- This should *never fail*.
- This should wait until communication has successfully ended (or timed out).
- All future sends/receives will fail.

A new header

- Observation: Everything we need in the header for minimsg, we also need for minisocket.
- Header(minimsg) $\subset$ Header(minisocket)
- Can we structure our socket header to take advantage of our marshalling/unmarshalling from minimsg?

  - With care, yes we can.
  - Do not mix the socket header into the base header.



Retransmission

1. Set the initial timeout to occur 100ms after the first send.
2. Each time the timeout expires, resend the message and double the timeout interval.
3. After 12.7 seconds (seven timeouts), stop trying to send and return an error.
4. When a send is acknowledged, or aborted, reset the timeout value to 100ms.

Hand-shaking

- Client sends `OPEN_CONNECTION`
- Server responds with `OPEN_CONNECTION_ACK` or error:

  `SOCKET_BUSY` A client is already connected to this socket server.

  `SOCKET_NOSERVER` No server is waiting on this port.
- Client confirms `OPEN_CONNECTION_ACK` with its own `ACK`.
- This is subject to the retransmission scheme.

Implementation approach

- Separate the two modules.
- Put common code in a third file.
- Demultiplex in the network handler.
- Pass control to the right module based on “type”.
- Some code will inevitably be duplicated ...
- ... but, you should work hard to define module boundaries, and prevent intermingling of code.

Where to begin?

- Think about the process as a state machine:
  - Server: {waiting for client, client connected, closing socket, ...}
  - Client: {waiting for connection to establish, ...}
  - Server/Client: {sending packet, sending `ACK`, retransmit, ...}
- Transitions:
  - Packet received.
  - An API function is called.
  - Retransmit timeout expires.
  - ...

Looking for more of a challenge?

- Add a static window size > 1 to your implementation.
- Read about TCP's congestion control algorithm. Describe (or implement) the changes necessary to use this algorithm with minisockets.
- Demonstrate that your implementation is not subject to `ACK`-spoofing or Dup`ACK` attacks.

Moving forward

- There is lots to do on this project.
- Start early (this is not a single-weekend project).
- Ask if you are not sure.
  - Even better: Research how the problem you describe is handled in TCP; come to us with a proposed solution in-hand.
- Come to office hours.