Project 4

Reliable Streams

Zhiyuan Teo

Slide heritage: Previous TAs → Robert Escriva

Cornell CS 4411, October 21, 2011
Updated November 3, 2011

Administrative Information

- Project 4 has been extended till 11.59pm Monday 7 November.
- An FAQ has been posted on the website. You may access it from here.
- Email the staff for help.
- Start early!

Assignment 4A

- This assignment is meant to get you started on understanding critical portions of the project early.
- Worth 5% of your project grade.
- It is an easy assignment!
- Submit onto CMS **individually** (every person needs to turn in a copy).

Reliable streams

- Why care about reliability at the OS layer?
- What does it mean to be reliable?
- What is a stream?

Best Effort vs. Reliability

- Embodies some notion of delivery guarantee.
- Allows the user to know if there was a failure.
- Data is delivered *at least* once (not lost in the network).
- Data is delivered *at most* once (no duplicates).
- Guarantees are not absolute; it is sufficient to just detect errors.

Stream protocols

- Connection-based (open, close, etc.).
- Terminology: server waits for a connection; client initiates one.
- The transport layer should treat messages as a sequence of bytes.
- Message boundaries are an application-level concept.

In real life: Transmission Control Protocol (TCP)

- TCP is a reliable, stream oriented protocol.
- Ordering is achieved by placing sequence numbers in packets.
  - Buffering, variable window size, and duplicate suppression are used in TCP.
- Flow control is (automatically) achieved by dropped packets.
  - Dropped packets usually indicate link capacity bottlenecks.
  - TCP adjusts to dropped packets by reducing its window size.
- Our implementation of minisockets will be a simplified version of TCP.

What does reliable streaming involve?

- Ordering: deliver in sequence (FIFO).
- Stream-like semantics:
  - User can send blocks of any size; minisockets should fragment the data if necessary.
  - User can ask to receive an arbitrary amount of data.

Relation to Project #3

- Unreliable and reliable protocols will co-exist.
- Some 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 packets to both protocols.
- It is up to you to *demultiplex* the network packets.

Base abstractions (both Project 3 and Project 4)

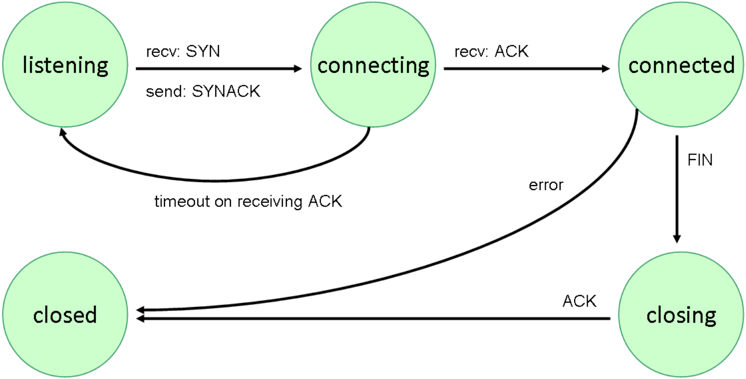
- Each protocol has the concept of a *port* as its endpoint.
- `network.h` is used by both.
- Ports are identified by number.
  - Ports for listening sockets are specified by the user and range from 0-32767.
  - Ports for connecting sockets are automatically allocated by the program and range from 32768-65535.

Differences with Project #3

- Separate miniheader format for reliable streams.
  - Format is a superset of the fields in unreliable datagrams.
- minisocket are bidirectional communication primitives.
  - Reads and writes are done to the same minisocket.
  - Port numbers are fixed when creating the client or server.
  - Remote address is fixed after connection.
- Data may need to be fragmented.
- You will need to keep (lots of) state at each endpoint.

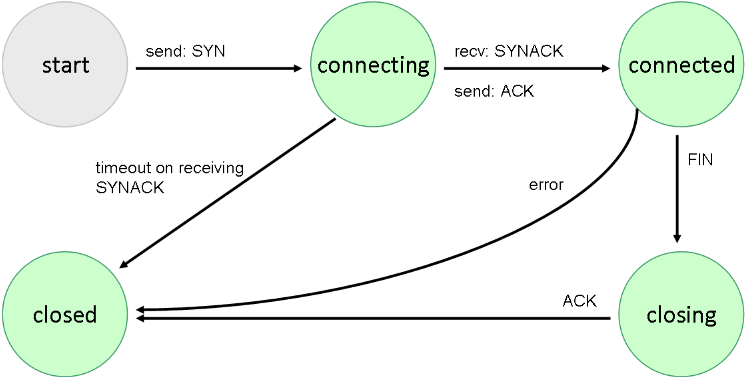
Listener (server) sockets

`minisocket_server_create()`

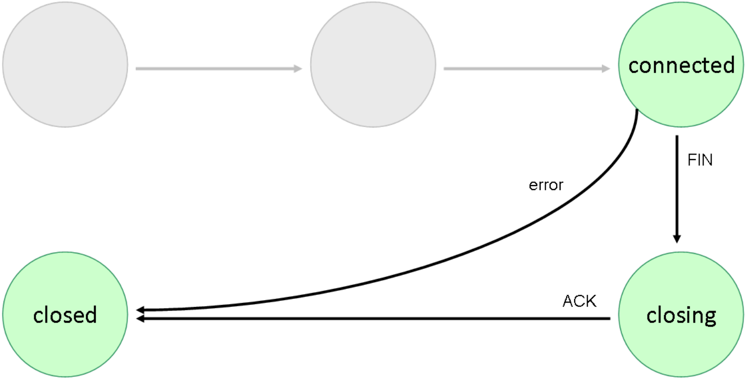


Connector (client) sockets

`minisocket_client_create()`



Common states



State transitions

- Note the similarity in state and state transitions once both ends are connected.
- Some special treatment will be required to get client/server socket into the connected state.
- But once in the connected state, there is no further need to distinguish server from client sockets.

Reliability

- At least once:
  - Delivery confirmation: acknowledgement for every packet received (ACK).
  - Retransmission: failure to receive an ACK triggers a retransmission after some timeout.
- At most once:
  - Keep track of the remote's sequence number and discard duplicate packets.
- 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).

Sequence numbers

- Both endpoints maintain their own sequence numbers.
- Sequence numbers start at 1.
- Each successive non-ACK packet generated by the local end increases the local sequence number by 1.
  - Packets sent locally do not affect the remote's sequence number.
  - Empty ACK packets are not given new sequence numbers.
- Retransmissions are resends of old packets and do not increase the sequence number.

Acknowledgement numbers

- Both endpoints also maintain a counter that describes the latest sequence number seen from the remote.
- This counter is known locally as the acknowledgement number.
- Accept a received packet only if packet's sequence number == local acknowledgement number + 1 (ie. sequence number of the packet is the next one you expect).
- If packet is accepted, update the acknowledgement number.
- For a window size of 1, the sequence number of the packet will not have a mismatch with the local acknowledgement number.

Fragmentation

- Cut the data into arbitrarily sized pieces.
- Assume that the sending application's boundaries are meaningless.
- 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

- Assume that there will be at most one sender writing to the socket at any time.
- But 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 communications must be performed concurrent with all other communications.
- Multiple distinct streams may operate in parallel.
  - You may use dedicated threads for handling control communications across all ports.

Creating a server

```
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.
  - Reject clients by sending `MSG_FIN`.

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 two reasons:
  - Connection was rejected because another client is already connected.
  - No response from server (even after retries): server is not listening on that port.
- Blocks until a successful connection is established (or it times out).

Connection Handshaking

- Client sends `MSG_SYN`.
- Server responds with `MSG_SYNACK` or error:
  - Send `MSG_FIN` if a client is already connected to this socket server.
  - Do not reply if there is no socket listening at the requested port.
- Client confirms `MSG_SYNACK` with its own `MSG_ACK`.
- This is subject to the retransmission scheme.

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);
```

- Send blocks until every fragment of transmitted data has been `ACK`nowledged.
  - Set the window size to 1.
- Receive blocks until data is available on the socket.
- Don't forget to `ACK` any data that you receive.

Closing a socket

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

- This should **never fail**.
- Inform the remote end of your intention to close the socket by sending `MSG_FIN`.
- Wait until either `MSG_ACK` is received or a timeout occurs on the transmission.
- All future sends/receives will fail.

Header for reliable streams

- Everything we need in the header for minimsg, we also need for minisocket.
- But we need more fields to indicate the packet type, sequence number and acknowledgement number.
- Message type is a `char`; set its value to one of the enums.
- Sequence and acknowledgement numbers are `unsigned ints`, make sure you pack them correctly using `pack_unsigned_int()`.
- With care, you might be able to reuse the code you developed for unpacking in minimsg.

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 12.8 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.

Other implementation specifics

- Return an error if no connector socket is available for assignment.
- Multiple threads may not simultaneously create server sockets on the same port.
- Set the window size to 1 (this simplifies the project substantially).

Simplifications

Do not worry about:

- Sequence numbers and acknowledgement numbers overflowing.
- Byzantine faults.

Testing

- Test your program by adjusting loss and duplication rates:
  - Set `synthetic_network` to 1 in `network.c`.
  - Adjust `loss_rate` and `duplication_rate`, also in `network.c`.

Advice

- There are many components in this project.
- Start early (this is not a single-weekend project).
- Ask if you are not sure.