CS414/415 Section 5
Project 4: Reliable networking

Krzysztof Ostrowski

krzys@cs.cornell.edu

Slides modified from previous years' slides

What do you have to do?

- A connection-based, stream-like reliable communication, similar to TCP.
- Features:
  - Connection-based (open, close etc.).
  - Guarantee that packets are delivered.
    - At least once (not lost in the network).
    - At most once (no duplicates).
    - Guarantees are not absolute (cannot be).

What do you have to do?

- Features (continued):
  - Ordering: deliver in sequence (FIFO).
  - Congestion control (sort of a sliding window with size one).
  - Stream-like (as opposed to packet-like) semantics:
    - Can send data blocks of any size: should fragment.
    - Can receive packets of any size.

How is it related to project #3 ?

- Unreliable/reliable protocols will co-exist.
  - Two separate APIs, sender can use both simultaneously.
    - Unrel.: ”`minimsg.h`”, rel.: ”`minisocket.h`”
    - One may rely on the other, but…
    - …keep code reasonably separate and isolated!
  - Receiver recognizes type of communication.
    - *Demultiplexing*: a new *protocol type* field in the base header, passing control to the right place.

How is it related to project #3 ?

- Common base abstractions:
  - We are still using ports as communication endpoints for both protocols.
  - Common base packet header (addr., ports)
    - Headers for other protocols staked on top of it
  - We will use miniports for unreliable, sockets for reliable networking.
    - User cannot use same ports for both kinds of communication (miniports hidden in sockets).

A connection-based protocol

- Connection identified by port numbers
- A typical scenario has multiple stages:
  - Server waiting for a client to connect
  - Client connects, information about connection (eg. port numbers) exchanged via hand-shaking
  - Client and server can send data in any direction
  - Client explicitly closes the connection
    - However, server can do it too (eg. when shutting down)
  - After connection closed, no send/receive succeeds
  - Socket is destroyed: accepting more clients requires that a new socket be created

Achieving reliability

- At least once:
  - Delivery confirmation: acknowledgement for every packet received (ACK).
  - Resending if not confirmed (timeout with an exponential back-off).
- At most once:
  - Suppressing duplicates
    - Receiver: duplicate data packets when ACK lost
    - Sender: duplicate ACKs when data believed lost but not really lost (only delayed)
- Not only for data, but also for control packets!
  - Duplicate requests to open/close connection etc.

Ordering and flow control

- Ordering: sequence numbers in packets
  - Normally: buffering, variable-size window, suppressing delivery of packets etc.
- Flow control: dropping packets
  - Normally: we use sliding window with size adjusted to bandwidth
- You can make window size to be equal to one
  - Simplifies implementation tremendously
  - One data packet in transit at all times (very slow)
  - Still need to keep / check sequence numbers

Achieving stream-like semantics

- Send data of any size: fragmentation
  - We simply cut in small pieces (arbitrarily)
    - Assume packet boundaries as determined by the sender app. are meaningless (byte stream)
    - Don’t put “reassembling” information in packets
    - Receiver treats all incoming data as parts of a single infinite byte stream
      - Ordering is essential to guarantee correctness here!
      - When data requested, may need to merge data from a few packets to fill the buffer given by client application

Achieving stream-like semantics

- Receive data of any size:
  - Specify maximum amount of data to receive
    - May consume only a part of an incoming packet
    - An “unused” part of the packet is left in the socket for another *receive* operation to consume
  - May receive less data than requested
    - Since it’s a stream, the exact size doesn’t matter
    - Client application is assumed smart enough to know what to do with the incoming stream
      - For example, it could add some “merging” information
      - We don’t care about it

Achieving stream-like semantics

- One-sender-to-one-receiver, but…
  - …multiple threads can send, and what's worse…
  - …multiple threads can issue *receive* to the same socket concurrently
    - Threads will form a receiver queue to the socket
    - Independent threads can receive random pieces of data!
    - They will need to know how to reassemble them
      - We don't care about it, it's up to the application to assemble
  - Still, all control communication (ACKs etc.) will need to be handled globally, in parallel
    - May require dedicated some threads for this purpose

Minisockets API (to implement)

- Creating socket on the server:

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

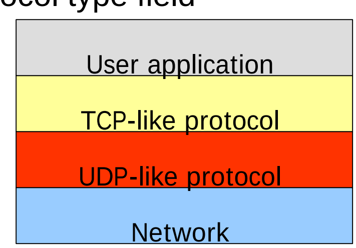
  - Server installed on a specific port (may fail)
  - Waits for incoming connection
    - Blocking (completes only after client connected)
    - Returns a complete socket connected with client
  - Simplification: one-to-one communication
    - Only a single client can connect (so it could fail)
    - Once a client connected, connecting not allowed

Minisockets API (to implement)

```
void minisocket_initialize();

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

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);

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

Our protocol stack

- Base header from **“miniports.c”** acts as our UDP/IP-like protocol header
  - Need it to identify communication endpoints
  - Need to extend it with protocol type field
- Add a new, TCP-like header on top of that
  - Don’t mix info about the reliable communication with the base header!



Implementation approaches

- Approach #1: “TCP” on top of “UDP/IP”
  - “UDP/IP” doesn’t know much about “TCP”
    - May need to add miniport “types” to represent various types of protocols stacked on top
    - Still should check if the packets received match the type of receiver (application directly / “TCP”)
  - “TCP” uses “UDP” via “send/receive”
    - Any interaction via interface defined in header.
    - May need a separate thread for each port to handle control traffic (sending ACKs etc.)
  - Arguably simpler to implement…
    - …but that’s not the way things are done in real life.

Implementation approaches

- Approach #2: “TCP”, “UDP/IP” in parallel
  - Neither of the two protocols “knows” about the other or “uses” the other:
    - Two separate modules for the two protocols
    - Demultiplexing in the network handler:
      - passing control to the right module based on “type”
    - Both are using the same “ports” infrastructure
    - Some code will inevitably be duplicated…
      - …but we should still keep the two modules reasonably separate, their code shouldn’t be intermingled.

Retransmission scheme

- Send, wait for ACK for a given timeout
- Complete if ACK is received on time
  - We don't send ACK to ACK here!
- Resend if ACK not arrived on time
  - Attempt up to 7 times, then give up (error)
  - Start with 100ms delay, then x2 each time
  - If an "old" ACK arrives now, it's still OK

Hand-shaking protocol

- Stages:
  - Client sends OPEN_CONNECTION
  - Server responds with ACK
    - It might also respond with error:
      - Socket in use by another client
      - Socket does not exist
      - Socket exists, not in use, but no thread waiting
  - Client confirms ACK with his own ACK
- Retransmission scheme applies here!

Implementation hints

- Think about it as a state machine
  - States / state:
    - Server: waiting for a client to arrive, client is connected, closing the socket, …
    - Client: waiting for server to accept connection, …
    - Server/Client: various stages while sending a packet (ACK received? Which resending attempt? What timeout period?)
    - …
  - State transitions:
    - Packets received
    - An API function called
    - Timeout expired
    - …

Questions?

- Ask if not sure.
- By all means, come to office hours.