Internet Programming & Protocols Lecture 7

Reliable streams

TCP header

Reliable streams

- How to build a transport protocol that provides a reliable stream of bytes on top of IP?
- IP is based on datagrams
  - Not circuit-based
  - Not a stream of bits
- IP is a best-effort protocol, packets can be
  - Mangled (bit errors)
  - Delayed
  - Duplicated
  - Arrive out of order
  - lost

Building a stream on top of packets

- Application writes or reads a sequence of bytes
  - Pointer to start of user data
  - Length of data
- Transport protocol for sender must divide the user data into a sequence of packets (roughly MTU sized), and send each packet to the receiver
  - Since the data to be sent may not be a multiple of MTU, each packet should carry a length field.
- At the receiver, the protocol just needs to pass each packet up to the application's read() and provide the length to the application.
  - Application may have to do multiple read's to collect all the bytes

Building a reliable stream

- For reliability, protocol must
  - Let the sender know that receiver received the packet
    - Our protocol needs an acknowledgement (ACK) packet
  - Insure that packets received have no errors
    - So our protocol must add a checksum or CRC to each packet
    - Sender must calculate the checksum for each packet and append to packet. So now we have a packet header with a CRC
    - Receiver must re-calculate the checksum on each packet and compare. If checksum fails, receiver needs to inform the sender. We can do this with a negative acknowledgement (NAK) packet.
    - If sender receives an ACK, then send the next packet. If NAK is received, retransmit the last packet
    - What if ACK or NAK packets get mangled (probably need our checksum for these packets as well)?
      - If sender receives mangled ACK/NAK, just re-transmit last segment until good ACK received.

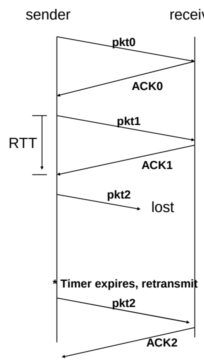
Stop-and-wait

- So far, we have a stop-and-wait protocol based on ACK's and NAK's
  - Send a data packet, wait til ACK or NAK arrives
  - If NAK arrives, re-send packet.
  - If ACK arrives, send next packet
- What if network duplicates one of the ACK packets?
  - Sender could think last packet was received OK, when in fact what arrived first was a duplicate of the previous ACK …
  - Also a problem if a data packet is duplicated. Receiver would think it's the next packet, but it's not!
  - We should add a packet number to our protocol that accompanies each packet and is included in the ACK/NAK
  - Receiver should always ACK the packet number, even for duplicate data packets.
- Our packet header consists of
  - Type (data, ACK, NAK)
  - Packet number
  - Packet length
  - checksum

Handling packet loss

- What if data packet is lost?
  - Receiver never sends ACK/NAK, sender waits forever ☹
- What if ACK or NAK is lost?
  - Sender waits forever ☹
- Sender needs a timer
  - After sending packet, sender sets a timer
  - If the timer expires before ACK/NAK arrives, then resend data packet
  - If ACK/NAK arrives, cancel timer.
  - How long to wait?
    - Ideally round-trip time, but that can vary, and we may not know what RTT is. So often pick something "long enough" – 3 seconds?
  - Note if ACK was lost, receiver will get a duplicate packet, but our packet number in the header will allow him to discard already accepted packets. Receiver must ACK such duplicate packets.
  - What if ACK was delayed and arrives shortly after we retransmitted? Too bad, packet is already in flight, we can't zap it. Sender will eventually receive another ACK for duplicated packet. Sender just ignores duplicate ACK's for packets that have already been ACK'd

Our reliable protocol



• What is the data rate of our protocol?

one packet/RTT

Example:

packet size: 1000 bytes (8000 bits)

100 mbs path, RTT 40 ms

data rate = 8000/0.04 = 200 kbs

path utilization 0. 2/100 == 0. 2%

this is poor, even for government work.

Utilization gets worse with longer RTT!

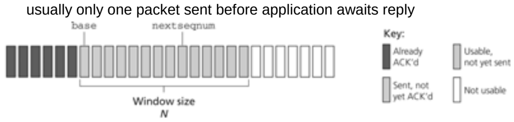
Pipelining our protocol

- Stop-and-wait data rate == packetsize/RTT
  - Could use bigger packet size to improve data rate (MTU limited?)
  - Alter speed of light to make RTT smaller ☺
- Improve data rate by allowing sender to initially send N packets
  - Potentially improve data rate by factor of N (up to link bandwidth)
- Cost:
  - Sender and receiver both need N packet buffers
  - ACK/NAK handling more complex
    - Which packets have been ACK'd, which are in flight
    - Current left edge of window

- Sender needs N packet buffer because, packets may need to be retransmitted if NAK or lost
- Receiver needs to buffer packets in case left edge packet is lost. Since it's a byte stream, can't pass data to application if there are holes
  - Example: packets 1 and 2 arrive and are passed to application. Packets 4, 5, and 6 arrive (but not 3), receiver must buffer these til sender times out and resends packet 3.

Sliding window protocol

- Sender sends a new packet only if left edge packet ACK'd, otherwise receiver could need infinite receive window.
- Sender still needs timer. If timer expires and left edge packet has not been ACK'd, resend left-edge packet and restart timer. When left edge ACK packet received by sender, move left edge (slide window) to next un-ACK'd packet, send that many more new packets, and restart timer.
- For bulk transfers (ftp) window will fill, for interactive sessions (ssh), usually only one packet sent before application awaits reply



© Kurose

How big a window?

- N=1, it's our stop-and-wait protocol, performance sucks – worsens with RTT
- Ideally, choose N based on path bandwidth and path delay
  - The **bandwidth-delay product**
    - If path bandwidth is 100 mbs and RTT is 10 ms, bandwidth delay product is 100 * .01 megabits = 1 million bit buffer. With packet size of 8,000 bits, want N to be 125 packets (125 Kbytes)
    - If RTT is 100 ms, N needs to be 1250 packets (1.2 Mbytes)
    - RTT 100 ms and GigE (1000 mbs) path, need 12 Mbytes of buffer!
- Alas, this value will vary for each pair of hosts and current route (RTT)
  - We don't usually know either RTT or path bandwidth
  - Most applications just take the OS default (like 32 KB)
  - In TCP, SNDBUF and RCVBUF define window values
- On wide-area links, the window size is usually the performance bottleneck!! We'll have lots more to say about the bandwidth-delay product

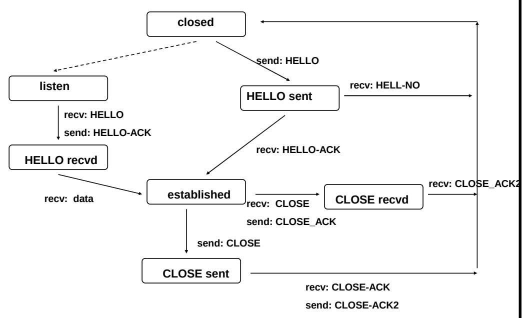
Starting and stopping our reliable stream protocol

- How does receiver know sender has sent the last packet?
  - Need another packet type, CLOSE
    - Needs to be checksum'd and reliable
    - Needs to be ACK'd by other end, CLOSE-ACK
    - need a CLOSE-ACK ACK (CLOSE-ACK2)
      - When CLOSE-ACK2 received, that end can close
      - After sending CLOSE-ACK2, close.
- Need a startup (connect/accept-refuse) protocol to establish "connection" (allocate buffers, initialize state, e.g., packet counters)
  - Need HELLO packet type
  - If receiver is listening, replies with HELLO-ACK packet, or if no application listening, responds with HELL-NO packet
- HELLO packet causes listener to allocate buffers, initialize state, etc.
- CLOSE allows both ends to release resources (buffers, state info)

Reliable stream protocol summary

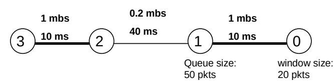
- Our protocol specifies how to open/close a connection and how to send and acknowledge packets. The sender uses a timer.
- Our packet header:
  - Type (HELLO, HELLO-ACK, HELL-NO, CLOSE, CLOSE-ACK, CLOSE-ACK2, DATA, ACK, NAK)
  - Packet length
  - Packet number
  - Checksum
- Sliding-window implementation requires buffers at both ends
- Finite state machine defines transitions for open/established/close

Reliable stream finite state machine



Animations

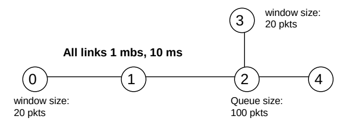
- ACK protocols are self-clocking ANIMATION (ns simulations)



What would happen if queue size was only 10 pkts ?

- ACK protocols adapt to changing available bandwidth ANIMATION

0 transmits to 4, later 3 sends to 4, 3 finishes



Extending our protocol

- To support multiple applications on a host, our protocol needs a source and destination port number in the packet header
- Our protocol is flawed. The ACK from the receiver says the transport protocol has received the packet, BUT the application may not be ready to read the data from the OS. We need to stop the sender if the receiver application has not drained the receive buffer. We need flow control.
- Our protocol header will be extended with an “available window” field.
  - Starts out as size of receive buffer
  - Receiver updates it to reflect available space
    - If receiving application is not reading data from net buffer, the available window could shrink to 0. As application reads data, window will re-open.
  - Sender inspects “available window” in each ACK packet from the receiver and insures that it does not send more packets than the available window allows.

Our protocol vs TCP

- TCP provides a reliable stream on top of IP
  - Uses timers, sequence numbers, checksum, ACK, sliding window, flow control, congestion control
- TCP differs from our protocol
  - We count packets, TCP counts bytes
  - TCP is bidirectional, byte counters for both directions
  - TCP uses cumulative ACKs, no NAKs (more later)
  - TCP tries to estimate an appropriate timeout value from observed RTT
  - TCP provides an out-of-band delivery (URGENT)

TCP header

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Hdr  |           |U|A|P|R|S|F|                               |
| length| Reserved  |R|C|S|S|Y|I|            Window             |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent offset         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                             data                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Usually 5 32-bit words, but options used on initial connect.

Ethernet	IP	TCP/UDP	Application

TCP header C struct

/usr/include/netinet/tcp.h

```
struct tcphdr {
    u_short th_sport;               /* source port */
    u_short th_dport;               /* destination port */
    tcp_seq th_seq;                 /* sequence number */
    tcp_seq th_ack;                 /* acknowledgement number */
#ifdef _BIT_FIELDS_LTOH
    u_int   th_x2:4,                /* (unused) */
    th_off:4;               /* data offset */
#else
    u_int   th_off:4,               /* data offset */
    th_x2:4;                /* (unused) */
#endif
    u_char  th_flags;
#define TH_FIN  0x01
#define TH_SYN  0x02
#define TH_RST  0x04
#define TH_PUSH 0x08
#define TH_ACK  0x10
#define TH_URG  0x20
    u_short th_win;                 /* window */
    u_short th_sum;                 /* checksum */
    u_short th_urp;                 /* urgent pointer */
};
```

TCP header fields

- 16-bit source and destination ports
- 32-bit sequence number, byte number of first data byte in packet
- 32-bit ACK number, byte number +1 of last data byte received
- 4-bit Hdr length number of 4-byte words in header
- FLAGS
  - URG urgent byte (MSG_OOB) coming, urgent offset valid
  - ACK ack of data up to ack number
  - PSH all data in buffer sent, push data up to application (meaningless)
  - RST reset, no port or service aborted
  - SYN synchronize, start connection (connect())
  - FIN finish, end connection (close())
- window, space available in my receive buffer (initially, SO_RCVBUF)
- checksum over TCP pseudo header and data
- 16-bit urgent offset of byte of urgent data within stream (added to seq. number)

TCP header notes

- URG flag and offset allow application to insert one byte of “out of band” data. Receiving application can be notified with signal() or other mechanisms to retrieve the byte “out of order” – e.g., to send a ctrl-c ahead of a bunch of keystrokes in the buffer. NOT really used, API is messy.
- TCP options | type | length | value |
  - Header length normally 5 words
  - negotiated in SYN sequence
  - variable number of options, padded to 4-byte boundary with no-ops
  - newer options, both sides must support
  - 1-byte type code

Type			
0	end of list		
1	no-op		
2	len(4),MSS (2 bytes)	max segment size	| 2 | 4 | MSS |
3	len(3),shiftcount	window scale factor	
4	len(2) permitted	selective ACK (experimental)	
8	len(10),timestamp(4),timestamp(4)	timestamp	| 8 | 10 | timestamp | timestamp |

TCP ACKs and sequence numbers

- No NAKs
- Data is numbered by bytes, not packets
- Number is unsigned and wraps!
- Bidirectional, so sequence/ACK numbers for both ends
- Sequence number of first byte in packet
- ACK number of last good contiguous byte received (+ 1)
- Initial segment number established at connect/accept
  - Start with new number (+128 or random) for each new connection!
  - Does NOT start at 0 (avoid earlier incarnations)

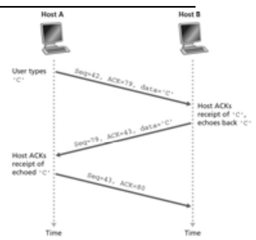
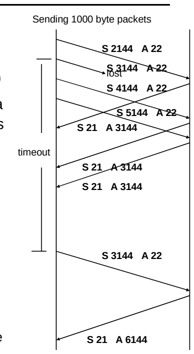


Figure 3.31 ♦ Sequence and acknowledgement numbers for a simpl Telnet application over TCP

© Kurose

Lost packets and cumulative ACK

- If packet arrives out of order, (e.g. lost packet), ACK number+1 of last good contiguous byte received (not in RFC 793)
- When lost packet arrives, receiver sends a cumulative ACK, ack’ing all the good bytes in its buffer
  - not in original RFC 793
  - Receiver could discard out of order data, have go-back-N protocol
- Good news: ACKs can be lost and next ACK can advance sender
- Bad news: duplicate ACKs during loss period conveys little information about packets that have arrived
  - Case 1: broken connection, no ACKs arrive
  - Case 2: lost packet, duplicate ACKs arrive



Establishing a TCP connection

- Uses 3-way handshake
- Client connect() sends SYN packet with initial sequence number (new/random)
  - Need new number to avoid old packets for same 5-tuple
  - SYN for SYNchronize end points
- Server replies with ACK and its own SYN and its new sequence number
- Client ACKs server’s SYN
- Connection established, send/receive buffers allocated
- If no reply, connect() times out after exponential backoff/retry
- If port unavailable, server sends RST

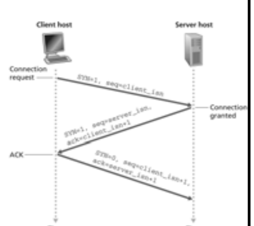


Figure 3.38 ♦ TCP three-way handshake: segment exchang

© Kurose

Closing a TCP connection

- Application issuing close (active) send FIN
- Other end (passive) ACKs FIN
  - FIN causes EOF on reads/writes
  - Application eventually closes, issuing a FIN
- Active end ACKs FIN and waits …
  - 2* maximum segment lifetime (30, 60 120s?)
  - Waiting for delayed packets?
  - Can’t re-use port for that period
    - Socket option SO_REUSEADDR
    - netstat –a (FIN_WAIT2)
- close() waits til packets in flight are handled
  - shutdown() for immediate close
- ctrl-c a net application and OS handles close

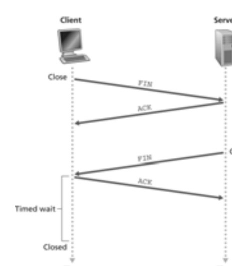


Figure 3.39 ♦ Closing a TCP connection

© Kurose

Variations on close

- close() returns immediately
  - no more sends/recvs; send buffer sent, then FIN sent
- SO_LINGER socket option effects behavior of close()

```
/usr/include/sys/socket.h

struct  linger {
int     l_onoff;               /* option on/off */
int     l_linger;              /* linger time */
};
```

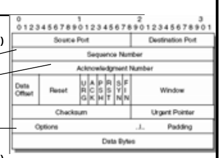
- l_onoff=0 -- default close
- l_onoff=1 and l_linger =0 -- connection aborted (RST), data discarded
- l_onoff=1 and l_linger =N -- process waits til all data sent and ACK'd, or until the timer (N) expires (N secs or .01 secs)
- shutdown() with SHUT_RD
  - recvd data discarded, can still send
- shutdown() with SHUT_WR
  - no more sends, can still recv, send buffer sent, then FIN
- be sure parent and child have closed socket

TCP data transfer protocol

- TCP uses sliding window protocol
- Initial window (usually) set from size of RCVBUF
- Header includes “available window” field
- Sender can have no more bytes in flight than minimum of sender’s SNDBUF or receiver’s current available window
- In-order bytes are ACKed, ACK can piggy-back on return data (best)
- Sender has retransmit timer
  - Multiple timeouts on same packet – exponential backoff (not in RFC 793)
- If receiver application is not reading data, window can go to zero
  - If no data to be sent by receiver, it has no way of telling sender window is open again
  - Sender must send window probes to receiver! (example later)

TCP 3-way handshake

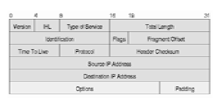
```
DENEB.1055 > ACHERNAR.discard: S 732608000:732608000(0)
win 8192 <mss 1460>
4500 002c c4f2 0000 3c06 fbfd 80a9 5e4a
          ports
80a9 5e3f 041f 0009 2baa b600 0000 0000
          cksum
6002 2000 d477 0000 0204 05b4

ACHERNAR.discard > DENEB.1055: S 728320000:728320000(0)
ack 732608001 win 8192 <mss 1460>
4500 002c f742 0000 3c06 c9ad 80a9 5e3f

80a9 5e4a 0009 041f 2b69 4800 2baa b601

6012 2000 60fd 0000 0204 05b4

DENEB.1055 > ACHERNAR.discard: . ack 1 win 8192
4500 0028 c4f3 0000 3c06 fc00 80a9 5e4a
80a9 5e3f 041f 0009 2baa b601 2b69 4801
5010 2000 78ba 0000
```



IP proto 6: TCP

TCP option: 2, MSS

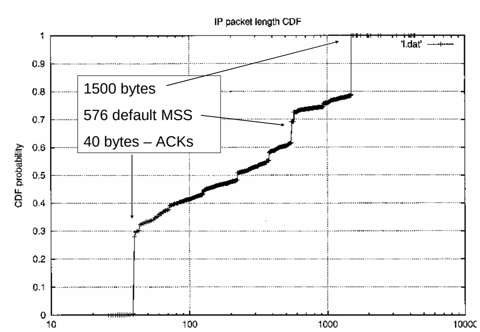
Window: 8192 (2000 hex)



MSS and path MTU discovery (sidebar)

- TCP tries to avoid IP fragmentation
  - Transport layer worrying about network/link layer issues ☹
  - Actually, there is application layer access socket option TCP_MAXSEG
- RFC 793 specified each end could provide the max segment size they would accept (MTU – IP/TCP header = 40)
  - If no MSS option, sender must use 536 byte segments
  - Of course, today, intervening links may have smaller MTU’s
- RFC 1191 proposed “path MTU” discovery
  - When full size datagram goes out, set DF IP bit (don’t fragment)
  - If router needs to fragment, it’s “supposed” to send back ICMP saying what its MTU is
  - Sender OS adjusts MSS (or at least turns off DF)
  - Cool if OS caches MTU info for path (typically in local routing table)
  - TROUBLE: routers don’t do it right, firewalls block ICMP – result **black hole**
    - You can ping remote and send small packets, but if you try to send big packets with DF, they disappear. OS should try with DF off
    - Maybe configure your OS to disable pMTUd

Packet size distribution



Next time ...

- TCP finite state machine
- Performance monitoring

- Assignments 3 and 4