CSCI-1680
Transport Layer II

Data over TCP: Flow Control

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti
• Sign up for IP grading: this week and next week
• TCP assignment: out now—start early!
  – Gear-up session soon, details forthcoming
  – The next few lectures will help you
  – Schedule Milestone I meeting by Thurs, April 14
• More details soon on what happens after TCP
Topics for today

From before break
• TCP: connection setup
• Sockets

New stuff: How to send data
• Flow control: how to send data without overwhelming receiver
• Congestion control: how to send data without overwhelming network
TCP – Transmission Control Protocol
TCP – Transmission Control Protocol
TCP provides a “reliable, connection oriented, full duplex ordered byte stream”
TCP Header

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Source Port</td>
<td>Destination Port</td>
<td></td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sequence Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Acknowledgment Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>U</td>
<td>A</td>
</tr>
<tr>
<td></td>
<td>Offset</td>
<td>Reserved</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>G</td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Checksum</td>
<td>Urgent Pointer</td>
<td></td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Options</td>
<td>Padding</td>
<td></td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

WHERE THIS PACKET IS IN DATA STREAM (BYTES)

NEXT SEQ EXPECTED

NOW MUCH DATA WE CAN RECEIVE.

WHERE THIS NEXT SEQ EXPECTED
Most important header fields

- **Ports**: multiplexing
- **Sequence number**
  - Correspond to bytes, not packets!
- **Acknowledgment Number**
  - Next expected sequence number
- **Window**: willing to receive
  - Lets receiver limit SWS (even to 0) for flow control
- **Checksum**: a (really weak) checksum, see RFC
Header Flags

- URG: whether there is urgent data
- ACK: ack no. valid (all but first segment)
- PSH: push data to the application immediately
- RST: reset connection
- SYN: synchronize, establishes connection
- FIN: close connection
TCP State Diagram

- **SYN RECEIVED**
  - Upon receipt of a SYN packet, the state transitions to LISTEN.
  - If no SYN is received within a timeout, the state transitions to CLOSED.

- **LISTEN**
  - Transition to CONNECT/SYN if SYN is sent.
  - Transition to CLOSED if a RST is received.

- **CONNECT/SYN**
  - (Step 1 of the 3-way handshake)
  - Transition to SYN SENT with a SYN/ACK.

- **SYN SENT**
  - (Step 2 of the 3-way handshake)
  - Transition to ESTABLISHED with a SYN/ACK.

- **ESTABLISHED**
  - Data exchange occurs.
  - Transition to CLOSED/FIN on FIN packet.
  - Transition to CLOSED on FIN/ACK packet.

- **CLOSE/FIN**
  - Transition to CLOSED on FIN/ACK packet.
  - Transition to LAST ACK on ACK packet.

- **LAST ACK**
  - Transition to CLOSED on ACK packet.

- **CLOSED**
  - (Go back to start)

- **Active CLOSE**
  - Transition to CLOSING on FIN packet.
  - Transition to TIME WAIT on FIN/ACK packet.

- **CLOSING**
  - Transition to TIME WAIT on FIN/ACK packet.

- **TIME WAIT**
  - Transition to LAST ACK on FIN/ACK packet.

- **CLOSE WAIT**
  - Transition to LAST ACK on FIN/ACK packet.

- **LAST ACK**
  - Transition to CLOSED on ACK packet.

- **FIN WAIT 1**
  - Transition to FIN WAIT 2 on FIN/ACK/ACK packet.

- **FIN WAIT 2**
  - Transition to TIME WAIT on FIN/ACK packet.

- **CLOSE/FIN**
  - Transition to CLOSED on ACK packet.

- **FLOW CONTROL**
  - Send/Recv Data

- **TEARDOWN**
  - Connection Tear Down

- **SETUP**
  - Connection Setup
Establishing a Connection

- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: server may send RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout
Establishing a Connection

- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: server may send RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout
Establishing a Connection

- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: server may send RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout
Establishing a Connection

- Three-way handshake
  - Two sides agree on respective initial sequence nums
- If no one is listening on port: server may send RST
- If server is overloaded: ignore SYN
- If no SYN-ACK: retry, timeout
Sequence numbers

How to pick the initial sequence number?

• Protocols based on relative sequence numbers based on starting value

• But why not start at 0?

• Instead, pick an arbitrary number
State for a TCP connection kept in Transmission Control Buffer (TCB)
• Keeps initial sequence numbers, connection state, send/recv buffers, status of unACK’d segments, ...
• When to allocate?
  – Client: Initiating a connection (sending a SYN)
  – Server: accepting a new connection (receiving SYN)
  – Listening on a socket*
Each connection has an associated TCB in the kernel

For each packet kernel maps the 5-tuple (tcp/udp, src IP, src port, dst IP, dst port) to a socket
SYN flooding

- What happens if you send a huge number of SYN packets?
  - Denial of Service
  - Exhaust memory
  - Exhaust available sockets
A hacky solution: SYN cookies

- Don’t allocate TCB on first SYN
- Encode some state inside the initial sequence number that goes back to the client (in the SYN+ACK)
- What gets encoded?
  - Coarse timestamp
  - Hash of connection IP/port
  - Other stuff (implementation dependent)
- Better ideas?
Sending data

We should not send more data than the receiver can take: flow control

• When to send data?
  – Sender can delay sends to get larger segments

• How much data to send?
  – Data is sent in MSS-sized segments
    • MSS = Maximum Segment Size (TCP packet that can fit in an IP packet)
    • Chosen to avoid fragmentation
Simplest method: Stop and Wait

Consider sending one packet at a time
- S: Send packet, wait
- R: Receive packet, send ACK
- S: Receive ACK, send next packet
OR
No ACK within some time (RTO), timeout and retransmit
It. LOST DATA

LOST ACK
DELAYED ACK
What can go wrong?
Sequence number example

<table>
<thead>
<tr>
<th>A sends</th>
<th>B sends</th>
</tr>
</thead>
<tbody>
<tr>
<td>seq=0</td>
<td>SYN+ACK, seq=0, ack=1 (expecting)</td>
</tr>
<tr>
<td>ACK</td>
<td>ACK, seq=1, ack=4</td>
</tr>
<tr>
<td>&quot;abc&quot; ACK</td>
<td>seq=1, ack=8</td>
</tr>
<tr>
<td>&quot;defg&quot; ACK</td>
<td>seq=1, ack=8</td>
</tr>
<tr>
<td>&quot;foobar&quot; ACK</td>
<td>seq=1, ack=14, &quot;hello&quot;</td>
</tr>
<tr>
<td>FIN</td>
<td>seq=6, ack=21, &quot;goodbye&quot;, crossing packets</td>
</tr>
<tr>
<td>ACK</td>
<td>seq=6, ack=22, FIN</td>
</tr>
<tr>
<td>FIN</td>
<td>seq=6, ack=22, FIN</td>
</tr>
<tr>
<td>ACK</td>
<td>seq=22, ack=7, ACK of FIN</td>
</tr>
</tbody>
</table>
Better Flow Control: Sliding window

- Part of TCP specification (even before 1988)
- Send multiple packets at once, based on a window
- Receiver uses window header field to tell sender how much space it has
Receive Window

Send end adds data

Data ready for app to read

Last byte received

Next byte expected

(Number sent in ACK)

App removes data by calling recv()

Window size to advertise in ACK =

(MAX_RECV_BUFFER) -

(NEXT BYTE EXPECTED - 1) - (LAST BYTE READ)
Flow Control: Sender

Invariants

- $\text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow}$
- $\text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{BytesInFlight})$
- $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
Flow Control: Sender

Invariants
- LastByteSent – LastByteAcked <= AdvertisedWindow
- EffectiveWindow = AdvertisedWindow – (BytesInFlight)
- LastByteWritten – LastByteAcked <= MaxSendBuffer

Terminology:
RFC 793, Sec 3.3

Useful Sliding Window
AdvertisedWindow

\[ = \text{MaxRcvBuffer} - ((\text{NextByteExpected}-1) - \text{LastByteRead}) \]
Flow control: receiver

**AdvertisedWindow**

\[ \text{AdvertisedWindow} = \text{MaxRcvBuffer} - ((\text{NextByteExpected}-1) - \text{LastByteRead}) \]

Useful Sliding Window
Terminology:
RFC 793, Sec 3.3
Flow Control

(a) Sending application

LastByteWritten
TCP

LastByteAked
LastByteSent

(b) Receiving application

LastByteRead
TCP

NextByteExpected
LastByteRcvd
Flow Control

• Advertised window can fall to 0
  – How?
  – Sender eventually stops sending, blocks application

• Sender keeps sending 1-byte segments until window comes back > 0
Some Visualizations

- Normal conditions:  https://www.youtube.com/watch?v=zY3Sxvj8kZA
- With packet loss:   https://www.youtube.com/watch?v=Ik27yiITOvU
How do ACKs work?

- ACK contains next expected sequence number
- If one segment is missed but new ones received, send duplicate ACK
- If receiver gets 3 dup ACKs, retransmit

- How to know when to retransmit? Compute based on observed RTT, more on this later
When to Transmit?

- Nagle’s algorithm
- Goal: reduce the overhead of small packets
  
  ```
  if (there is data to send) and (window >= MSS)
      Send a MSS segment
  else
      if there is unAcked data in flight
          buffer the new data until ACK arrives
      else
          send all the new data now
  ```
- Receiver should avoid advertising a window <= MSS after advertising a window of 0
Delayed Acknowledgments

- Goal: Piggy-back ACKs on data
  - Delay ACK for 200ms in case application sends data
  - If more data received, immediately ACK second segment
  - Note: never delay duplicate ACKs (if missing a segment)
Delayed Acknowledgments

• **Goal:** Piggy-back ACKs on data
  – Delay ACK for 200ms in case application sends data
  – If more data received, immediately ACK second segment
  – Note: never delay duplicate ACKs (if missing a segment)

• **Warning:** can interact badly with Nagle for some applications
  – Nagle waits for ACK until send => Temporary deadlock
  – App can disable Nagle with `TCP_NODELAY`
  – App should also avoid many small writes
Limitations of Flow Control

- Network may be the bottleneck
  - Signal from receiver not enough!
- Sending too fast will cause queue overflows, heavy packet loss
- Flow control provides correctness
- Need more for performance: congestion control
Second goal

- We should not send more data than the network can take: congestion control