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
• HW3: Due yesterday
• IP project grading: meetings should be done, grades out soon
• TCP assignment: out now—start early!
  – Last lecture, and the next few lectures, will help you
  – Schedule your Milestone I meeting on/before Monday, November 7
What do you need for the milestone?

Initial design/implementation for establishing connections

• Socket table and per-connection state
• Using the TCP header => 3-way handshake
  – No checksum yet
• Some design questions (see assignment for details)
  – What state will you store for each connection?
  – How will you use threads?

No grading server tests… just do as much as you can and bring it to your meeting => we will give feedback!
TCP: The story so far

Last lecture
• Sockets
• TCP: connection setup

Today
• Basic flow control: How to send data
• Connection teardown
TCP – Transmission Control Protocol

TCP provides a “reliable, connection oriented, full duplex ordered byte stream”
TCP Header

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement Number
- Data Offset
- Reserved
- URG
- ACK
- PSH
- RST
- SYN
- FIN
- Window Size
- Checksum
- Urgent Pointer
- Options
- Data
Important Header Fields

• **Ports: multiplexing**
• **Sequence number**
  – Where segment is in the stream (in **bytes**)
• **Acknowledgment Number**
  – Next expected sequence number
• **Window**
  – How much data you’re willing to receive
• **Flags**…
Important Header Fields: Flags

- **SYN**: establishes connection ("synchronize")
- **ACK**: this segment ACKs some data (all packets except first)
- **FIN**: close connection (gracefully)
- **RST**: reset connection (used for errors)
- **PSH**: push data to the application immediately
- **URG**: whether there is urgent data
Less important header fields

- **Checksum:** Very weak, like IP
  - Has weird semantics ("pseudo header"), more on this later…

- Data Offset: used to indicate TCP options (mostly unused)
- Urgent Pointer
TCP HANDSHAKE: Review

1. 1.2.3.4
   CLIENT
   DIAL ON
   CONNECT

   SYN
   SEQ: Y

   SYN+ACK
   SEQ: Y
   ACK: X+1

   ACK
   SEQ: X+1
   ACK: Y+1

ESTABLISHED
Review: 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
Summary of TCP States
Summary of TCP States

Connection Establishment

EVENT/ACTION

1. Passive open
2. Active open/SYN
3. Send/ SYN
4. SYN/ SYN + ACK
5. SYN/ SYN + ACK
6. SYN/ SYN + ACK
7. SYN + ACK/ACK
8. SYN + ACK/ACK
9. ACK
10. FIN/ACK
11. Close/FIN
12. Close/FIN
13. Time out after two segment lifetimes
14. Timeout after two segment lifetimes
15. ACK
16. ACK
17.ACK
18. ACK
19. ACK
20. ACK
21. ACK
22. ACK
23. ACK
24. ACK
25. ACK
26. ACK
27. ACK
28. ACK
Summary of TCP States

MOST DATA TRANSFER OCCURS NEXT.

Active open/SYN

Connection Establishment

CLOSED

Passive open

Close

LISTEN

SYN/SYN + ACK

SYN/SYN + ACK

Send/SYN

SYN_SENT

SYN + ACK/ACK

ESTABLISHED

FIN_WAIT_1

Close/FIN

FIN_WAIT_2

ACK + FIN/ACK

ACK

CLOSE_WAIT

FIN/ACK

CLOSE

LAST_ACK

ACK

TIME_WAIT

ACK

Timeout after two segment lifetimes

CLOSED
Summary of TCP States

Active close:
Can still receive
Summary of TCP States

Passive close:
Can still send!

Active close:
Can still receive

ONE SIDE DECIDES TO BE DONE!
TCP State Diagram

Edge notation: Event/Action
(see event X / do this and change state)
How to pick the initial sequence number?

- Protocols based on relative sequence numbers based on starting value
- Why not start at 0?
  - Security: Avoid "seq. num. prediction" attacks
  - Robustness: Guard against reuse of same numbers

- RFC9293, Sec 3.4.1: Procedure for picking ISN, based on timer and cryptographic hash

  => For project, just pick a random integer :)
### Relative Sequence Numbering

<table>
<thead>
<tr>
<th>Source Port: 49719</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination Port: 22</td>
</tr>
<tr>
<td>[Conversation completeness: Complete, WITH_DATA (31)]</td>
</tr>
<tr>
<td>[TCP Segment Len: 0]</td>
</tr>
<tr>
<td><strong>Sequence Number: 0</strong> (relative sequence number)</td>
</tr>
<tr>
<td><strong>Sequence Number (raw): 2000828645</strong></td>
</tr>
<tr>
<td>[Next Sequence Number: 1 (relative sequence number)]</td>
</tr>
<tr>
<td>Acknowledgment Number: 0</td>
</tr>
<tr>
<td>Acknowledgment number (raw): 0</td>
</tr>
<tr>
<td>1011 .... = Header Length: 44 bytes (11)</td>
</tr>
</tbody>
</table>

### Actual Number in Packet

### Relative Seq Num.
Keeping state: the TCB

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?

A collection of state for one TCP connection.
Keeping state: the TCB

State for a TCP connection kept in Transmission Control Buffer (TCB)

- Keeps initial sequence numbers, connection state, send/recv buffers, status of unACK’ed segments, ...

- When to allocate?
  - Server: listening on a connection*
  - Client: Initiating a connection (sending a SYN)
  - Server: accepting a new connection (receiving SYN)
Recall: the socket table

- Each connection has an associated TCB in the kernel.
- For each packet, kernel maps the 5-tuple (tcp/udp, local IP, local port, remote IP, remote port) to a socket.
- Depending on socket type, the socket contains a TCB.
Two “types” of TCP sockets

- **Listen sockets**
  - Created by `server` to **accept new connections**
  - When client connects, kernel queues info, allocates **new socket** on **accept**

- **“Normal” sockets**
Two “types” of TCP sockets

- “Normal” sockets:
  - Connection between two specific endpoints
  - Can send/recv data

- Listen sockets
  - Created by server to accept new connections
  - When a client connects, client info gets queued by kernel
  - When server process calls accept(), a new ("normal") socket is created between the server and that client
NOTA BENE: This diagram is only a summary and must not be taken as the total specification. Many details are not included.
SYN flooding

What happens if you send a huge number of SYN packets?

- Can SYN result memory, port numbers.
- Denial of service attack (DoS)
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)
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?

$\Rightarrow \text{ Ev. RATE LIMITING # OF CONNECTIONS PER TIME.}$
Sending data

Flow control: don’t send more data than the receiver can handle
• TCP stack divides data into packets called segments
Sending data

Flow control: don’t send more data than the receiver can handle
- TCP stack divides data into packets called segments

Questions
- When to send data?
- How much data to send?
Sending data

Flow control: don’t send more data than the receiver can handle
• TCP stack divides data into packets called segments

Questions
• When to send data?
• 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

\[ \text{MSS} = 1400 \text{ bytes} - \text{LEN(IP HDR)} - \text{LEN(TCP HDR)} \]
**SIMPLI ST SENDER**: STOP & WAIT, *(IDEAL CASE)*

**S**ender

1. SYN Seq=0, Ack=0
2. SYN-ACK Seq=0, Ack=1
3. ACK Seq=1, Ack=1
4. Seq=1, Ack=1

**R**ceiver

- **S**: SEND SEGMENT, WAIT FOR ACK *(EX. 1, 3)*
- **R**: WAIT FOR SEGMENT, SEND ACK *(EX. 2, 4)*

KEY FIELDS *(FOR SOME X, Y)*

- **Seq**: SEGMENT STARTS AT POSITION X IN DATA STREAM
- **Ack**: "I HAVE UP TO BYTE (Y-1), I EXPECT BYTE Y NEXT"
- **Window**: HOW MANY BYTES LEFT IN R'S RECEIVED BUFFER (MORE ON THU SOON!)
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

IF No ACK within some time (RTO), timeout and retransmit

(RTO time adapts to network conditions, more on this later.)
Sequence number example

<table>
<thead>
<tr>
<th>A sends</th>
<th>B sends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SYN, seq=0</td>
</tr>
<tr>
<td>2</td>
<td>SYN+ACK, seq=0, ack=1 (expecting)</td>
</tr>
<tr>
<td>3</td>
<td>ACK, seq=1, ack=1 (ACK of SYN)</td>
</tr>
<tr>
<td>4</td>
<td>“abc”, seq=1, ack=1</td>
</tr>
<tr>
<td>5</td>
<td>ACK, seq=1, ack=4</td>
</tr>
<tr>
<td>6</td>
<td>“defg”, seq=4, ack=1</td>
</tr>
<tr>
<td>7</td>
<td>seq=1, ack=8</td>
</tr>
<tr>
<td>8</td>
<td>“foobar”, seq=8, ack=1</td>
</tr>
<tr>
<td>9</td>
<td>seq=1, ack=14, “hello”</td>
</tr>
<tr>
<td>10</td>
<td>seq=14, ack=6, “goodbye”</td>
</tr>
<tr>
<td>11, 12</td>
<td>seq=21, ack=6, FIN</td>
</tr>
<tr>
<td>13</td>
<td>seq=6, ack=21 ;; ACK of “goodbye”, crossing packets</td>
</tr>
<tr>
<td>14</td>
<td>seq=6, ack=22 ;; ACK of FIN</td>
</tr>
<tr>
<td>15</td>
<td>seq=22, ack=7 ;; ACK of FIN</td>
</tr>
</tbody>
</table>
Stop & wait: what can go wrong?

Lost data

Lost ACK

Sequence numbers + retransmissions allow us to recover from all of these!
What can go wrong?

- **Lost Data**: Sender sends Data[N], Receiver receives ACK[N], Sender time out.
- **Lost ACK**: Sender sends Data[N], Receiver sends ACK[N], Sender time out.
- **Late ACK**: Sender sends Data[N], Receiver sends ACK[N], Sender expects ACK[N+1], Receiver sends Data[N+1].
WHAT'S THIS ABOUT A BUFFER?

TCP BUFFERING: NETWORK-LEVEL OVERVIEW

SEND BUFFER (CIRCULAR)

TCP STACK DECIDES NOW/WHEN TO SEND OUT
(REMOVES DATA ONCE RECEIVER ACKS IT)

APP ADDS DATA TO BUFFER.
CONN. WRITE.

RECV BUFFER (CIRCULAR)

TCP STACK ADDS DATA AS IT'S RECEIVED
(MIGHT BE OUT OF ORDER . . .)
CONN. READS DATA FROM BUFFER
(REMOVING IT)

MORE ON THIS LATER! . . .
EXTRA CONTENT

FOR LATER

FEEL FREE TO READ AHEAD!
Connection Termination

• When you have no more data to send, send a FIN
  – Sent by `close()` or `shutdown()`

• Both sides close connection separately!
Connection Termination

- When you have no more data to send, send a FIN
  - Sent by `close()` or `shutdown()`
- Both sides close connection separately!
- `TIME_WAIT`: initiating side should wait for $2 \times MSL$ before deleting TCB
Connection Termination

- When you have no more data to send, send a FIN
  - Sent by close() or shutdown()
- Both sides close connection separately!
- **TIME_WAIT**: initiating side should wait for 2*MSL before deleting TCB
  - MSL = Longest time a segment might be delayed (configurable, ~1min)
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
Sliding window (for later)
Flow Control: Sender

Invariants

- LastByteSent − LastByteAcked ≤ AdvertisedWindow
- EffectiveWindow = AdvertisedWindow − (BytesInFlight)
- LastByteWritten − LastByteAcked ≤ MaxSendBuffer
Flow Control: Sender

Invariants

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

Useful Sliding Window
Terminology:
RFC9293, Sec 3.4
Flow control: receiver

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

**AdvertisedWindow**

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

Useful Sliding Window
Terminology:
RFC 9293, Sec 3.4
Flow Control
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
Sequence numbers (Circumference = 0 to $2^{32}$ slots)

Initial sequence number

Data received, acknowledged and delivered to application

Data received, acknowledged, but not yet delivered to application

Data received, but not acknowledged

Unfilled buffer

Window shifts

Receiver's window (Allocation buffer) Up to $2^{16}-1$ slots
Some Visualizations

• Normal conditions: https://www.youtube.com/watch?v=zY3Sxvj8kZA

• With packet loss: https://www.youtube.com/watch?v=lk27yilTOvU
Sliding window: 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 unAced 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