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
"Hi, I'd like to hear a TCP joke."
"Hello, would you like to hear a TCP joke?"
"Yes, I'd like to hear a TCP joke."
"OK, I'll tell you a TCP joke."
"Ok, I will hear a TCP joke."
"Are you ready to hear a TCP joke?"
"Yes, I am ready to hear a TCP joke."
"Ok, I am about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline."
"Ok, I am ready to get your TCP joke that will last 10 seconds, has two characters, does not have an explicit setting, and ends with a punchline."
"I'm sorry, your connection has timed out. ... Hello, would you like to hear a TCP joke?"
Administrivia

• IP project grading: happening now! Sign up for a meeting if you haven’t already

• TCP assignment: out now—start early!
  – Gearup I: Thursday 10/26 5-7pm
  – Milestone 1: schedule meeting on/before Thursday, November 2
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
- Window Size
- Checksum
- Urgent Pointer
- Options
- Data

Flags:
- URG
- ACK
- PSH
- RST
- SYN
- FIN

- INDEFINITE
- CONT
- ACK
- PSH
- RST
- SYN
- FIN

In Index into Data Stream < Next Byte Expected
- Both sides need to agree on their starting seq numbers
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
Establishing a Connection

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

- **CLOSED**
  - **CONNECT/SYN** (Step 1 of the 3-way-handshake)
  - **CLOSEd**
  - **LISTEN**
    - **SYN/SYN+ACK** (Step 2 of the 3-way-handshake)
  - **SYN SENT**
    - **SEND/SYN**
    - **SYN+SYN=ACK** (Simultaneous open)
    - **SYN+ACK=ACK** (Step 3 of the 3-way-handshake)
  - **SYN RECEIVED**
    - **RST=/-**
    - **ACK=/-**
  - **ESTABLISHED**
    - **Data exchange occurs**
  - **CLOSE/FIN**
  - **FIN/FIN**

- **FIN WAIT 1**
  - **FIN/FIN**
  - **FIN-ACK/FACK**
  - **ACK-**

- **FIN WAIT 2**
  - **FIN/FIN**
  - **FIN-ACK/FACK**
  - **ACK-**

- **CLOSING**
  - **TIME WAIT**
    - **Timeout**
    - **CLOSE/FIN**

- **CLOSE WAIT**
  - **LAST ACK**
    - **ACK-**

- **ACTIVE CLOSE**
  - **PASSIVE CLOSE**

(To go back to start)
We are now here
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?

**Server**: Listening on connection (LISTEN)

**Client**: Initiating connection (SYN)

**Server**: After each client connects (ACCEPT)
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?
– Server: listening on a connection*
– Client: Initiating a connection (sending a SYN)
– Server: accepting a new connection (receiving SYN)

⇒ When to deallocate?
UNTIL ALL DATA HAS BEEN ACK’D, CLOSE PROCESS IS DONE.
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?
– Server: listening on a connection*
– Client: Initiating a connection (sending a SYN)
– Server: accepting a new connection (receiving SYN)

When to deallocate?
Only after connection termination is fully completed (CLOSED state)
=> If no state, can’t meaningfully respond to packet!
NOTA BENE: This diagram is only a summary and must not be taken as
the total specification. Many details are not included.
Recall: the socket table

- Each connection has an associated TCB in the kernel
- Depending on socket type, socket contains TCB

⇒ For each packet, kernel maps the 5-tuple
  \((\text{tcp/udp, local IP, local port, remote IP, remote port})\) \text{ } \Rightarrow \text{ socket}
5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Conn
- Kernel maintains socket table: maps (5-tuple) => Socket

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
tcp4 0 0 *.22 *.* LISTEN
```

- If a 5-tuple is reused => new ISN, so sequence numbers likely out of range from past connection
5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Conn

- Kernel maintains socket table: maps (5-tuple) => Socket

```
5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Conn
- Kernel maintains socket table: maps (5-tuple) => Socket
```

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto  Recv-Q  Send-Q  Local Address               Foreign Address            (state)
tcp4   0       0       172.17.48.121:22           192.168.1.58:34452       SYN_SENT
tcp4   0       0       *.22                        *.*                      LISTEN
```

- If a 5-tuple is reused => new ISN, so sequence numbers likely out of range from past connection
5-tuple (proto., source IP, source port, dest IP, dest port) => 1 Conn

- Kernel maintains socket table: maps (5-tuple) => Socket

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address Foreign Address (state)
tcp4 0 0 172.17.48.121:22 192.168.1.58:34452 SYN_SENT
tcp4 0 0 172.17.48.121:22 142.250.80.35:11435 ESTABLISHED
tcp4 0 0 172.17.48.121:22 13.225.231.50:12345 ESTABLISHED
          ...          ...                      
tcp4 0 0 *.22 *.22 LISTEN
```

- If a 5-tuple is reused => new ISN, so sequence numbers likely out of range from past connection
Two "types" of sockets:

- "Normal" sockets
- Listen sockets
“Normal” sockets

– Connection between two specific endpoints
– Can send/recv data

Listen sockets

– Created by receiver 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
How to pick the initial sequence number?

- Protocols based on *relative* seq. numbers based on starting value
- Why not start at 0?
How to pick the initial sequence number?

- Protocols based on relative seq. numbers based on starting value
- Why not start at 0?
  
  => Someone might guess the value!

  => IF NUMBER REUSED (SYSTEM RESTART), CAN'T TELL IP BACK IT IS FOR DIFF CONNECTION.

  => 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

Transmission Control Protocol, Src Port: 49719, Dst Port: 22, Seq: 0, Len: 0

Sequence Number (raw): 2000828645

Flags: 0x002 (SYN)
Observation: new connections use memory!

What happens if you send a someone lots of SYN packets?

SYN flood => type of Denial of Service (DOS) attack
A SYN flood => type of Denial of Service (DOS) attack
=> Especially bad when attack traffic comes from multiple sources
(more on this later)
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?
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

Nowadays: filtering in kernel (or in network) on number of new connections per time (esp. on servers)
  => More on this later!
Sending data

Step 2 of the 3-way-handshake: SYN/SYN+ACK

Step 3 of the 3-way-handshake: SYN+ACK/ACK

Data exchange occurs
DROP PACKET. RETRANSMIT IF TIMEOUT (RTO)

LOST ACK: 11

DUP ACK/PACKET:

⇒ ALWAYS KNOW WHAT SEGMENT OR ACK TO EXPECT NEXT.
IF YOU DON'T RECEIVE AN EXPECTED SEGMENT, IGNORE IT.

"IN THE WINDOW" OF EXPECTED PKT/ACKS"
Sending data: the basic idea

• Start: app calls Send(), loads send buffer

> Part of TCB
Sending data: the basic idea

- Start: app calls Send(), loads send buffer
- TCP stack divides data into packets called segments
Sending data: the basic idea

- Start: app calls `Send()`, loads send buffer
- TCP stack divides data into packets called **segments**

Key challenges

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

⇒ Flow control (now): don’t send more data than the receiver can handle
⇒ Congestion control (much later) don’t send more data than the network can handle
Terminology: MSS

MSS: Maximum segment size

- Largest segment a TCP can send
- Can be configurable

- Nowadays: sender and receiver negotiate using TCP options (out of scope for this class)
Terminology: MSS

MSS: Maximum segment size
- Largest segment a TCP can send
- Can be configurable

- Nowadays: sender and receiver negotiate using TCP options (out of scope for this class)

\[ \approx 1000 \text{ bytes} \]

=> For project: just a fixed value
Simplest TCP sender: stop and wait
**SIMPLIEST SENDER: STOP + WAIT**, (IDEAL CASE)

(S)ENDER

<R>ECIVER

<table>
<thead>
<tr>
<th>SYN</th>
<th>SEQ=0, ACK=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN-ACK</td>
<td>SEQ=0, ACK=1</td>
</tr>
<tr>
<td>ACK</td>
<td>SEQ=1, ACK=1</td>
</tr>
<tr>
<td>SEG=1, ACK=1</td>
<td></td>
</tr>
<tr>
<td>&quot;HELLO&quot;</td>
<td></td>
</tr>
<tr>
<td>SEG=1, ACK=6</td>
<td></td>
</tr>
<tr>
<td>&quot;WORLD&quot;</td>
<td></td>
</tr>
<tr>
<td>SEG=6, ACK=1</td>
<td></td>
</tr>
<tr>
<td>SEG=1, ACK=11</td>
<td></td>
</tr>
</tbody>
</table>

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

**KEY FIELDS (FOR SOME X, Y)**
- **SEG**: 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 FOR NEXT SEGMENT
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.)
Stop and wait: What can go wrong?

Sequence numbers + retransmissions allow us to recover from all of these!

Lost data

Lost ACK

Late ACK

SEQ: N

ACK = N+1

SEQ: N

ACK = N+1

Timeout (RTO)
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
S

| RTT |

S

R

DATA

ACK

DATA

ACK

| ONLY ONE SEGMENT "IN FLIGHT" (PENDING) AT ANY ONE TIME |

| DROP PACKET |

| LOST ACK |

| DUP PACKET |

| ACK |
What can go wrong?
Sequence number example

<table>
<thead>
<tr>
<th></th>
<th>A sends</th>
<th>B sends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SYN, seq=0</td>
<td>SYN+ACK, seq=0, ack=1 (expecting)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>ACK, seq=1, ack=1</td>
</tr>
<tr>
<td>3</td>
<td>“abc”</td>
<td>ACK, seq=1, ack=4</td>
</tr>
<tr>
<td>4</td>
<td>“defg”</td>
<td>seq=1, ack=8</td>
</tr>
<tr>
<td>5</td>
<td>“foobar”</td>
<td>seq=1, ack=14, “hello”</td>
</tr>
<tr>
<td>6</td>
<td>“goodbye”</td>
<td>seq=6, ack=21 ;; ACK of “goodbye”, crossing packets</td>
</tr>
<tr>
<td>7</td>
<td>FIN</td>
<td>seq=6, ack=22 ;; ACK of FIN</td>
</tr>
<tr>
<td>8</td>
<td>ACK</td>
<td>seq=6, ack=22, FIN</td>
</tr>
<tr>
<td>9</td>
<td>ACK</td>
<td>seq=6, ack=22, FIN</td>
</tr>
<tr>
<td>10</td>
<td>ACK</td>
<td>seq=6, ack=7 ;; ACK of FIN</td>
</tr>
</tbody>
</table>
STOP
AWAIT

WOULD YOU HAVE MORE SEGMENTS
"IN FLIGHT" AT ONE TIME TO
USE MORE NETWORK BANDWIDTH?

CHALLENGES:
- RECEIVER NEEDS TO PUT SEGMENTS
  IN ORDER
- SEGMENTS MIGHT BE
  RECEIVED OUT OF ORDER
- SENDER: FLOW CONTROL
Now we think about buffering.

Send buffer (circular buffer)

TCP stack decides how to send out

"In-flight"

Send, but not ACK'd.

Remove after ACK received.
**RECV BUFFER** (CIRCULAR BUFFER)

- EARLY ARRIVALS

TCP STACK

- ADDS DATA WHEN SEGMENTS RECEIVED

APP READS

- DATA FROM BUFFER (CONN.READ)

DATA RECEIVED IN ORDER, READY FOR APP

MAX AMOUNT APP CAN READ.
WHAT'S THIS ABOUT A BUFFER?

TCP BUFFERING: HIGH-LEVEL OVERVIEW

**SEND BUFFER (CIRCULAR)**

- App adds data to buffer.
- Conn. write.

TCP stack decides now when to send out.

- (Removes data once receiver acknowledges it)

**RECV BUFFER (CIRCULAR)**

- Conn. read.
- Reads data from buffer (removing it)

TCP stack adds data as it's recv'd (might be out of order...)

More on this later!
MORE NOTES
IF YOU WANT TO READ AHEAD
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
TCP and buffering

Recall: TCP stack responsibilities

- Sender: breaking application data into segments
- Receiver: receiving segments, reassembling them in order

- Need to buffer data
Sliding window: in abstract terms

- Window of size $w$
- Can send at most $w$ packets before waiting for an ACK
Sliding window: in abstract terms

• Window of size $w$
• Can send at most $w$ packets before waiting for an ACK
• Goal
  – Network “pipe” always filled with data
  – ACKs come back at rate data is delivered => “self-clocking”
Sender example
Receiver example
Flow Control: Sender

Invariants

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

(a)
Flow Control: Sender

Invariants

- $LastByteSent - LastByteAcked \leq AdvertisedWindow$
- $EffectiveWindow = AdvertisedWindow - (BytesInFlight)$
- $LastByteWritten - LastByteAcked \leq MaxSendBuffer$

Useful Sliding Window

Terminology:

RFC 9293, Sec 3.3.1
Flow control: receiver

- Can accept data if space in window
- Available window = BufferSize - ((NextByteExpected-1) - LastByteRead)

- On receiving segment for byte S
  - if s is outside window, ignore packet
  - if s == NextByteExpected:
    - Deliver to application (Update LastByteReceived)
    - If next segment was early arrival, deliver it too
  - If s > NextByteExpected, but within window
    - Queue as early arrival

- Send ACK for highest contiguous byte received, available window
Flow control: receiver

- Can accept data if space in window
- Available window = BufferSize - ((NextByteExpected-1) - LastByteRead)

On receiving segment for byte S
  - if s is outside window, ignore packet
  - if s == NextByteExpected:
    - Deliver to application (Update LastByteReceived)
    - If next segment was early arrival, deliver it too
  - If s > NextByteExpected, but within window
    - Queue as early arrival

Send ACK for highest contiguous byte received, available window
Flow Control

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

- Resolution: zero window probing: sender sends 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=lk27yiITOvU
How do ACKs work?

- ACK contains next expected sequence number
- If one segment is missed but new ones received, send duplicate ACK
- Retransmit when:
  - Receive timeout (RTO) expires
  - Possibly other conditions, for certain TCP variants (eg. 3 dup ACKs)
- How to set RTO?
When to time out?

Should expect an ACK within one Round Trip Time (RTT)

- Problem: RTT can be highly variable

- Strategy: expected RTT based on ACKs received
  - Use exponentially weighted moving average (EWMA)
  - RFC793 version (“smoothed RTT”):
When to time out?

Should expect an ACK within one Round Trip Time (RTT)

- Problem: RTT can be highly variable

- Strategy: expected RTT based on ACKs received
  - Use exponentially weighted moving average (EWMA)
  - RFC793 version (“smoothed RTT”):
    \[
    SRTT = (\alpha \ast SRTT) + (1 - \alpha) \ast RTT_{\text{Measured}}
    \]
    \[
    RTO = \max(RTO_{\text{Min}}, \min(\beta \ast SRTT, RTO_{\text{Max}}))
    \]

- \(\alpha = “\text{Smoothing factor”}: 0.8-.9\)
- \(\beta = “\text{Delay variance factor”}: 1.3—2.0\)

RFC793, Sec 3.7
This is only the beginning…

- Problem 1: what if segment is a retransmission?
This is only the beginning...

- Problem 1: what if segment is a retransmission?
  - Solution: don’t update RTT if segment was retransmitted