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: The story so far

Last lecture
• Sockets
• TCP: connection setup

Today
• Basic flow control: How to send data
• Connection teardown
TCP provides a “reliable, connection oriented, full duplex ordered byte stream”
TCP Header

0  15  16  31
Source Port  Destination Port

Sequence Number

Acknowledgement Number

Data Offset  Reserved  U  R  G  A  C  K  P  A  S  T  R  S  T  N  F  S  Y  N  F  I  N

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
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
Summary of TCP States

Connection Establishment

- Passive open
- Active open/ SYN
- Connection Establishment

Passive close:
- Can still send!

Active close:
- Can still receive

States:
- CLOSED
- LISTEN
- SYN_RCVD
- SYN_SENT
- ESTABLISHED
- FIN_WAIT_1
- FIN_WAIT_2
- CLOSING
- TIME_WAIT
- CLOSE_WAIT
- LAST_ACK
- CLOSED

Transitions:
- Passive open
- Close
- Active open
- Send/SYN
- SYN/SYN + ACK
- SYN/ SYN + ACK
- SYN + ACK/ACK
- Close/FIN
- FIN/ACK
- FIN/ACK
- FIN/ACK
- Timeout after two segment lifetimes
- Close/FIN
- ACK
- ACK
- ACK
- ACK
TCP State Diagram

Closed

Listen

Connect/SYN (Step 1 of the 3-way-handshake)

SYN

SYN/SYN+ACK

(Step 2 of the 3-way-handshake)

Data exchange occurs

SYN+ACK/ACK

(Step 3 of the 3-way-handshake)

Established

FIN/ACK

FIN+ACK/ACK

Close/FIN

Active CLOSE

Passive CLOSE

Active CLOSE

Timeout

Close/FIN

Close/FIN

Closing

Time Wait

Last ACK

Fin Wait 1

Fin Wait 2

Unusual event

Client/receiver path

Server/sender path

Simultaneous open

Timeout

(Closed)

(Closed)
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?
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?
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
Recall: the socket table

<table>
<thead>
<tr>
<th>Proto</th>
<th>Recv-Q</th>
<th>Send-Q</th>
<th>Local Address</th>
<th>Foreign Address</th>
<th>(state)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>172.17.48.121.56915</td>
<td>192.168.1.58.7000</td>
<td>SYN_SENT</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>172.17.48.121.56908</td>
<td>142.250.80.35.443</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>172.17.48.121.56887</td>
<td>13.225.231.50.80</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td><em>.</em></td>
<td><em>.</em></td>
<td>LISTEN</td>
</tr>
</tbody>
</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
  \((\text{tcp/udp}, \text{local IP, local port, remote IP, remote port})\) \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

```
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                        *:*               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!
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!

=> 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>
<th>Destination Port: 22</th>
<th>Seq: 0</th>
<th>Len: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port: 49719</td>
<td>Destination Port: 22</td>
<td>Seq: 0</td>
<td>Len: 0</td>
</tr>
</tbody>
</table>

- Sequence Number (raw): 200828645
- Next Sequence Number: 1
- Acknowledgment Number: 0
- Acknowledgment number (raw): 0
- Flags: 0x002 (SYN)
Observation: new connections use memory!
Observation: new connections use memory!

What happens if you send a someone lots of SYN packets?
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
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
  – Other stuff (implementation dependent)

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

1. **SYN RECEIVED**
   - **CLOSED**
   - **LISTEN**
   - **CONNECT/SYN**
     - (Step 2 of the 3-way-handshake)
   - **SYN/SYN+ACK**
     - (Simultaneous open)
   - **RST/-**
   - **SYN/SYN+ACK**
   - **SEND/SYN**
   - **SYN+ACK/ACK**
     - (Step 3 of the 3-way-handshake)
   - **Data exchange occurs**

2. **SYN SENT**
   - **CLOSE/FIN**
   - **CLOSE/FIN**
   - **FIN/ACK**
   - **Active CLOSE**
   - **Passive CLOSE**

3. **FIN WAIT 1**
   - **FIN/ACK**
   - **FIN+ACK/ACK**
   - **ACK/-**
   - **FIN WAIT 2**
   - **TIME WAIT**
   - **Timeout**
   - **CLOSED**

4. **CLOSING**
   - **FIN/ACK**
   - **FIN+ACK/ACK**
   - **ACK/-**

5. **CLOSE WAIT**
   - **CLOSE/FIN**
   - **LAST ACK**
   - **ACK/-**

6. **UNUSUAL EVENT**
   - **client/receiver path**
   - **server/sender path**

7. **RST/ACK**

8. **SYN/ACK**

9. **TIME OUT**

10. **CLOSED**

11. **LISTEN/-(Start)**

12. **CONNECT/SYN**
    - (Step 2 of the 3-way-handshake)

13. **SYN/SYN+ACK**
    - (Simultaneous open)

14. **SYN/SYN+ACK**
    - **SEND/SYN**

15. **SYN+ACK/ACK**
    - (Step 3 of the 3-way-handshake)

16. **Data exchange occurs**
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
Sending data: the basic idea
• Start: app calls Send(), loads send buffer
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?
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
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)

=> For project: just a fixed value
Simplest TCP sender: stop and wait
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
What can go wrong?

The right half of the diagram, by comparison, illustrates the case of a lost ACK. The receiver has received a duplicate Data[N]. We have assumed here that the receiver has implemented a retransmit-on-duplicate strategy, and so its response upon receipt of the duplicate Data[N] is to retransmit ACK[N].

As a final example, note that it is possible for ACK[N] to have been delayed (or, similarly, for the first Data[N] to have been delayed) longer than the timeout interval. Not every packet that times out is actually lost!

In this case we see that, after sending Data[N], receiving a delayed ACK[N] (rather than the expected ACK[N+1]) must be considered a normal event.

In principle, either side can implement retransmit-on-timeout if nothing is received. Either side can also implement retransmit-on-duplicate; this was done by the receiver in the second example above but not by the sender in the third example (the sender received a second ACK[N] but did not retransmit Data[N+1]).
(We will see below that this table is slightly idealized, in that real sequence numbers do not start at 0.)

Here is the ladder diagram corresponding to this connection:

<table>
<thead>
<tr>
<th>A sends</th>
<th>B sends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 SYN, seq=0</td>
<td>SYN+ACK, seq=0, ack=1 (expecting)</td>
</tr>
<tr>
<td>2</td>
<td>SYN+ACK, seq=0, ack=1 (expecting)</td>
</tr>
<tr>
<td>3 ACK, seq=1</td>
<td>ACK, seq=1, ack=1 (ACK of SYN)</td>
</tr>
<tr>
<td>4 “abc”, seq=1, ack=1</td>
<td>ACK, seq=1, ack=4</td>
</tr>
<tr>
<td>5</td>
<td>ACK, seq=1, ack=4</td>
</tr>
<tr>
<td>6 “defg”, seq=4, ack=1</td>
<td>seq=1, ack=8</td>
</tr>
<tr>
<td>7</td>
<td>seq=1, ack=8</td>
</tr>
<tr>
<td>8 “foobar”, seq=8, ack=1</td>
<td>seq=1, ack=14, “hello”</td>
</tr>
<tr>
<td>9</td>
<td>seq=1, ack=14, “hello”</td>
</tr>
<tr>
<td>10 seq=14, ack=6, “goodbye”</td>
<td>seq=6, ack=21 ;; ACK of “goodbye”, crossing packets</td>
</tr>
<tr>
<td>11,12 seq=21, ack=6, FIN</td>
<td>seq=6, ack=22 ;; ACK of FIN</td>
</tr>
<tr>
<td>13</td>
<td>seq=6, ack=22, FIN</td>
</tr>
<tr>
<td>14</td>
<td>seq=6, ack=22, FIN</td>
</tr>
<tr>
<td>15 seq=22, ack=7 ;; ACK of FIN</td>
<td>seq=6, ack=22, FIN</td>
</tr>
</tbody>
</table>
Problems?
Can we do better?
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
- 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

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

Useful Sliding Window Terminology:
RFC 9293, Sec 3.3.1
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
Unfilled buffer

Data received, but not acknowledged

Data received, acknowledged, but not yet delivered to application

Initial sequence number

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

Window shifts
Some Visualizations

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

• With packet loss: https://www.youtube.com/watch?v=lk27yiIOTOvU
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"):
    \[
    SRTT = (\alpha \times SRTT) + (1 - \alpha) \times RTT_{\text{Measured}}
    \]
    \[
    RTO = \max(RTO_{\text{Min}}, \min(\beta \times SRTT, RTO_{\text{Max}}))
    \]

\(\alpha\) = "Smoothing factor": .8-.9
\(\beta\) = "Delay variance factor": 1.3—2.0

RFC793, Sec 3.7
This is only the beginning…

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

• Problem 2: RTT can have high variance
  – Initial implementation doesn’t account for this
  – Congestion control: modeling network load