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

<table>
<thead>
<tr>
<th>Field</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>16 bits</td>
<td>Source port number</td>
</tr>
<tr>
<td>Destination Port</td>
<td>16 bits</td>
<td>Destination port</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>32 bits</td>
<td>Sequence number</td>
</tr>
<tr>
<td>Acknowledgement Number</td>
<td>32 bits</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>Window Size</td>
<td>16 bits</td>
<td>Window size</td>
</tr>
<tr>
<td>Checksum</td>
<td>16 bits</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>16 bits</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options</td>
<td>32 bits</td>
<td>Options</td>
</tr>
<tr>
<td>Data</td>
<td>40 bytes</td>
<td>Data</td>
</tr>
</tbody>
</table>
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
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

- **Passive close:** Can still send!

- **Active close:** Can still receive

Connection Establishment:
- Passive open
- Active open/SYN

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

(Start)

CLOSED

LISTEN

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

LISTEN

CLOSE/-

LISTEN/

CLOSE/-

SYN

RECEIVED

SYN/SYN+ACK

(Step 2 of the 3-way-handshake)

SYN/SENT

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

ESTABLISHED

Data exchange occurs

SYN

SENT

RST/-

SYN/SYN+ACK (simultaneous open)

SYN/SYN+ACK

(Step 2 of the 3-way-handshake)

CLOSE/FIN

CLOSE/FIN

FIN/ACK

(Step 3 of the 3-way-handshake)

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

FIN WAIT 1

FIN WAIT 2

CLOSED

TIME OUT

F IN/ACK

F IN+ACK/ACK

F IN/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

F IN+ACK/ACK

(End state)

Fin/ACK

Fin/ACK
Sequence numbers

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

• RFC9293, Sec 3.4.1: Procedure for picking ISN, based on timer and cryptographic hash
  => For project, just pick a random integer :)


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?
  - 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) => socket
• Depending on socket type, socket contains TCB

```
deemer@vesta ~ % netstat -anl
Active Internet connections (including servers)
Proto Recv-Q Send-Q Local Address             Foreign Address         (state)
tcp4  0   0     172.17.48.121.56915           192.168.1.58.7000        SYN_SENT
tcp4  0   0     172.17.48.121.56908           142.250.80.35.443        ESTABLISHED
tcp4  0   0     172.17.48.121.56887           13.225.231.50.80         ESTABLISHED
... tcp4  0   0     *.22                        *.*                          LISTEN
```
Two “types” of TCP sockets

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

<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>172.17.48.121:56887</td>
<td>13.225.231.50:80</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp4</td>
<td>0</td>
<td>0</td>
<td>*:22</td>
<td><em>:</em></td>
<td>LISTEN</td>
</tr>
</tbody>
</table>

- **“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.

RFC 9293, Sec 3.3.2
SYN flooding

What happens if you send a huge number of SYN packets?
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?
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
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\[N+1\]).
A sends

1. SYN, seq=0
2. SYN+ACK, seq=0, ack=1 (expecting)
3. ACK, seq=1, ack=1 (ACK of SYN)
4. “abc”, seq=1, ack=1
5. ACK, seq=1, ack=4
6. “defg”, seq=4, ack=1
7. “foobar”, seq=8, ack=1
8. “hello”, seq=14, “hello”
9. “goodbye”, seq=14, ack=6, “goodbye”
10. seq=14, ack=6, “goodbye”
11,12. seq=21, ack=6, FIN
13. seq=6, ack=21 ;; ACK of “goodbye”, crossing packets
14. seq=6, ack=22 ;; ACK of FIN
15. seq=22, ack=7 ;; ACK of FIN

B sends

1. SYN+ACK, seq=0, ack=1
2. ACK, seq=1, ack=1
3. ACK, seq=1, ack=4
4. ACK, seq=1, ack=8
5. ACK, seq=1, ack=14, “hello”
6. ACK, seq=1, ack=21
7. ACK, seq=6, ack=21
8. ACK, seq=6, ack=22
9. ACK, seq=6, ack=22, FIN
10. FIN, seq=6, ack=22
11. FIN, seq=22, ack=7
12. FIN, seq=22, FIN

(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:
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

• \( \text{LastByteSent} - \text{LastByteAcked} \leq \text{AdvertisedWindow} \)
• \( \text{EffectiveWindow} = \text{AdvertisedWindow} - (\text{BytesInFlight}) \)
• \( \text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer} \)

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

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

Useful Sliding Window
Terminology:
RFC 9293, Sec 3.4
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

![Diagram showing TCP flow control](image)
Unfilled buffer

Data received, but not acknowledged

Data received, acknowledged and delivered to application

Sequence numbers
(Circumference = 0 to $2^{32}$ slots)

Initial sequence number

Data received, acknowledged, but not yet delivered to application

Window shifts

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=lk27yiITOvU
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 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)

• 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*