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
Administrivia

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

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Source Port      | Destination Port |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Sequence Number  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Acknowledgment Number |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Data |      |U|A|P|R|S|F|
| Offset| Reserved |R|C|S|S|Y|I| Window |
| | | |G|K|H|T|N|N| |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Checksum | Urgent Pointer |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Options | Padding |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| data |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
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

- **CLOSED**: (Start) 
- **LISTEN**: 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
- **FIN WAIT 1**: FIN/ACK
- **CLOSING**: FIN+ACK/ACK
- **TIME WAIT**: Timeout

---

**States and Transitions**

- **UNUSUAL EVENT**: client/receiver path
- **SERVER/SENDER PATH**: unusual event
- **RST/−**: (simultaneous open)
- **SYN/−**: (Start)
- **SYN+ACK/−**: SYN/SYN+ACK
- **SEND/−**: SEND/SYN
- **CONNECT/−**: (Step 1 of the 3-way-handshake)
- **FIN/−**: FIN/ACK
- **FIN+ACK/−**: FIN+ACK/ACK
- **TIME OUT**: (Go back to start)

**States**

- **CLOSED**
- **FIN WAIT 1**
- **CLOSING**
- **TIME WAIT**
- **LAST ACK**
- **CLOSED**

**Actions**

- **LISTEN**
- **SYN SENT**
- **ESTABLISHED**

---

**Notes**

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

---

**TCP States and Transitions**

- **CLOSED**: (Start)
- **LISTEN**: 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
- **FIN WAIT 1**: FIN/ACK
- **CLOSING**: FIN+ACK/ACK
- **TIME WAIT**: Timeout

---

**States and Transitions**

- **UNUSUAL EVENT**: client/receiver path
- **SERVER/SENDER PATH**: unusual event
- **RST/−**: (simultaneous open)
- **SYN/−**: (Start)
- **SYN+ACK/−**: SYN/SYN+ACK
- **SEND/−**: SEND/SYN
- **CONNECT/−**: (Step 1 of the 3-way-handshake)
- **FIN/−**: FIN/ACK
- **FIN+ACK/−**: FIN+ACK/ACK
- **TIME OUT**: (Go back to start)

---

**States**

- **CLOSED**
- **FIN WAIT 1**
- **CLOSING**
- **TIME WAIT**
- **LAST ACK**
- **CLOSED**

**Actions**

- **LISTEN**
- **SYN SENT**
- **ESTABLISHED**

---

**Notes**

- **ACTIVE CLOSE**
- **PASSIVE CLOSE**
Establishing a Connection

- Three-way handshake
  - Two sides agree on respective initial sequence numbers
- 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
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?
  – 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 someone 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?
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
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 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\[N+1\]).
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>

(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:

A

SYN
SYN+ACK
ACK
“abc”
ACK
“defg”
ACK
“foobar”
“hello”
“goodbye”
FIN
ACK
ACK
FIN
ACK

B

SYN
SYN+ACK
ACK

In terms of the sequence and acknowledgment numbers, SYNs count as 1 byte, as do FINs. Thus, the SYN counts as sequence number 0, and the first byte of data (the “a” of “abc”) counts as sequence number 1. Similarly, the ack=21 sent by the B side is the acknowledgment of “goodbye”, while the ack=22 is the...
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
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:
RFC 793, Sec 3.3
Flow control: receiver

**AdvertisedWindow**

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

**Useful Sliding Window**

Terminology: RFC 793, Sec 3.3
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)

- Data received, acknowledged and delivered to application
- Data received, acknowledged, but not yet delivered to application
- Data received, but not acknowledged
- Unfilled buffer

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=lk27yiTOvU
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 \( \geq \) 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 \( \leq \) 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