TCP and Congestion Control

Networked Systems (H)
Lecture 7
Lecture Outline

• TCP and the Berkeley Sockets API
• Congestion control principles
• Congestion control in the Internet
  • TCP congestion control
  • Alternative approaches
TCP and the Berkeley Sockets API

SYNOPSIS
#include <sys/socket.h>

int
socket(int domain, int type, int protocol);

DESCRIPTION
The Berkeley Sockets API

• Widely used low-level C networking API

• First introduced in 4.3BSD Unix
  • Now available on most platforms: Linux, MacOS X, Windows, FreeBSD, Solaris, etc.
  • Largely compatible cross-platform

• Recommended reading:
Concepts

- Sockets provide a standard interface between network and application
- Two types of socket:
  - Stream – provides a virtual circuit service
  - Datagram – delivers individual packets
- Independent of network type:
  - Commonly used with Internet Protocol sockets, so stream sockets map onto TCP/IP connections and datagram sockets onto UDP/IP, but not specific to the Internet protocols
TCP Sockets

```c
int fd = socket(...)
connect(fd, ..., ...)
send(fd, data, datalen, flags)
recv(fd, buffer, buflen, flags)
close(fd)
```

```c
int fd = socket(...)
bind(fd, ..., ...)
listen(fd, ...)
connfd = accept(fd, ...)
recv(connfd, buffer, buflen, flags)
send(connfd, data, datalen, flags)
close(connfd)
```
What services do TCP sockets provide?

• TCP provides five key features:
  • Service differentiation
  • Connection-oriented
  • Point-to-point
  • Reliable, in-order, delivery of a byte stream
  • Congestion control

• These are provided by the operating system, via the sockets API
Client-server or peer-to-peer?

- Sockets initially unbound, and can either accept or make a connection

- Most commonly used in a client-server fashion:
  - One host makes the socket \texttt{listen()} for, and \texttt{accept()}, connections on a well-known port, making it into a server
    - The port is a 16-bit number used to distinguish servers
    - E.g. web server listens on port 80, email server on port 25
  - The other host makes the socket \texttt{connect()} to that port on the server
  - Once connection is established, either side can \texttt{send()} data into the connection, where it becomes available for the other side to \texttt{recv()}
  - Simultaneous connections are possible, using TCP in a peer-to-peer manner
Role of the TCP Port Number

Servers must listen on a known port; IANA maintains a registry. Distinction between system and user ports ill-advised – security problems resulted. Insufficient port space available (>75% of ports are registered).

TCP clients traditionally connect from a randomly chosen port in the ephemeral range. The port must be chosen randomly, to prevent spoofing attacks. Many systems use the entire port range for source ports, to increase the amount of randomness available.

http://www.iana.org/assignments/port-numbers
TCP Connection Setup

- Connections use 3-way handshake
  - The SYN and ACK flags in the TCP header signal connection progress
  - Initial packet has SYN bit set, includes randomly chosen initial sequence number
  - Reply also has SYN bit set and randomly chosen sequence number, acknowledges initial packet
  - Handshake completed by acknowledgement of second packet
  - Happens during the `connect()`/`accept()` calls

- Combination ensures robustness
  - Randomly chosen initial sequence numbers give robustness to delayed packets or restarted hosts
  - Acknowledgements ensure reliability

Similar handshake ends connection, with FIN bits signalling the teardown
Reading and Writing Data on a TCP Connection

- **Call `send()` to transmit data**
  - Will block until the data can be written, and returns actual amount of data sent
  - Might not be able to send all the data, if the connection is congested
  - Returns -1 if error occurs, sets `errno`

- **Call `recv()` to read up to BUFSIZE** bytes of data from a connection
  - Will block until some data is available or the connection is closed
  - Returns the number of bytes read from the socket; 0 if the sender closed the connection; or -1 and sets `errno` if an error occurred
  - Received data is *not* null terminated – potential security risk

```c
#define BUFSIZE 1500
...
ssize_t i;
ssize_t rcount;
char    buf[BUFSIZE];
...
rcount = recv(fd, buf, BUFSIZE, 0);
if (rcount == -1) {
    // Error has occurred
    ...
} else if (rcount < BUFSIZE) {
    // Couldn’t send it all, retry unsent
    ...
}
...
for (i = 0; i < rcount; i++) {
    printf("%c", buf[i]);
}

char data[] = “Hello, world!”;
int  datalen = strlen(data);
...
int sent = send(fd, data, datalen, 0);
if (sent == -1) {
    // Error has occurred
    ...
} else if (sent < datalen) {
    // Couldn’t send it all, retry unsent
    ...
}
...
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv) {
    const char *filename = argv[0];
    printf("%s
", filename);
    char data[] = “Hello, world!”;
    int  datalen = strlen(data);
    ...
    int sent = send(fd, data, datalen, 0);
    if (sent == -1) {
        // Error has occurred
        ...
    } else if (sent < datalen) {
        // Couldn’t send it all, retry unsent
        ...
    }
    ...
    ...
Reading and Writing Data on a TCP Connection

- The `send()` call enqueues data for transmission.
- This data is split into *segments*, each segment is placed in a *TCP packet*, that packet is sent when allowed by the congestion control algorithm.
  - Segments have sequence numbers → acknowledged by the receiver.
- If the data in a `send()` call is too large to fit into one segment, the TCP implementation will split it into several segments; similarly, several `send()` requests might be aggregated into a single TCP segment.
  - Both are done transparently by the TCP implementation and are invisible to the application.
- Implication: the data returned by `recv()` doesn’t necessarily correspond to a single `send()` call.
Application Level Framing

The `recv()` call can return data in unpredictably sized chunks – applications must be written to cope with this.

HTTP/1.1 200 OK
Date: Mon, 19 Jan 2009 22:25:40 GMT
Server: Apache/2.0.46 (Scientific Linux)
Last-Modified: Mon, 17 Nov 2003 08:06:50 GMT
ETag: "57c0cd-e3e-17901a80"
Accept-Ranges: bytes
Content-Length: 3646
Connection: close
Content-Type: text/html; charset=UTF-8

Example: HTTP/1.1 response
Ideally all headers received in one `recv()` call, then parsed to extract the Content-Length, then read entire body

TCP might split the response arbitrarily – parsing becomes more complex
TCP Reliability

- TCP connections are reliable
  - Each TCP packet has a sequence number and an acknowledgement number
    - Sequence number counts how many bytes are sent (this example is unrealistic, since it shows one byte being sent per packet)
  - Acknowledgement number specifies the next byte expected to be received
    - Cumulative positive acknowledgement
    - Only acknowledge contiguous data packets (sliding window protocol, so several data packets in flight)
    - If a packet is lost, receipt of subsequent packets will trigger duplicate acknowledgements
  - TCP layer retransmits lost packets – this is invisible to the application

![Diagram showing TCP reliability with sequence and acknowledgement numbers]
TCP Reliability: How is Loss Detected

- Triple duplicate ACK → some packets lost, but later packets arriving
  - Triple duplicate = Four identical ACKs in a row

- Timeout → send data but acknowledgements stop returning
  - Either the receiver or the network path has failed
TCP Reliability and Packet Reordering

- Packet delay leading to reordering will also cause duplicate ACKs to be received.
- Gives appearance of loss, when the data was merely delayed.
- TCP uses triple duplicate ACK as indication of packet loss to prevent reordered packets causing retransmissions.
  - Assumption: packets will only be delayed a little; if delayed enough that a triple duplicate ACK is generated, TCP will treat the packet as lost and send a retransmission.
Head of Line Blocking in TCP

- Data delivered in order, even after loss occurs
- TCP will retransmit the missing data, transparently to the application
- A `recv()` for missing data will block until it arrives; TCP always delivers data in an in-order contiguous sequence
A Complete TCP Connection

Initial 3-way handshake
- seq = 0, SYN
- seq = 0, ack = 1
- seq = 1500
- seq = 3000
- seq = 4500
- seq = 6000

Sending data
- seq = 7500
- seq = 9000
- seq = 4500
- seq = 4500
- seq = 10500, FIN

Closing 3-way handshake
- seq = 10500, ack = 10501

Sender
- SYN
- ACK
- SYN+ACK
- FIN
- ACK
- FIN+ACK

Receiver
- ack = 1, seq = 0, SYN
- ack = 1500
- ack = 3000
- ack = 4500
- ack = 4500
- ack = 10500
- ack = 10501, FIN

recv() blocks
recv() → 1500 bytes
recv() → 1500 bytes
recv() → 1500 bytes
recv() → 1500 bytes
recv() → 6000 bytes
recv() → 6000 bytes
Congestion Control
What is Congestion Control?

- Adapting speed of transmission to match available end-to-end network capacity
- Preventing congestion collapse of a network

Occurred in the Internet in 1986, before congestion control added.
Network or Transport Layer?

• Can implement congestion control at either the network or the transport layer
  • Network layer – safe, ensures all transport protocols are congestion controlled, requires all applications to use the same congestion control scheme
  • Transport layer – flexible, transports protocols can optimise congestion control for applications, but a misbehaving transport can congest the network
Congestion Control Principles

- Two key principles, first stated by Van Jacobson in 1988:
  - Conservation of packets
  - Additive increase and multiplicative decrease of the sending rate
- Together, ensure stability of the network
- Congestion control standards in IETF maintained by Sally Floyd for many years
  - High-speed TCP extensions, Quick start, SACK, ECN, etc.

Conservation of Packets

• The network has a certain capacity
  • The $bandwidth \times delay$ product of the path
• When in equilibrium at that capacity, send one packet for each acknowledgement received
  • Total number of packets in transit is constant
    • “ACK clocking” – each acknowledgement “clocks out” the next packet
  • Automatically reduces sending rate as network gets congested and delivers packets more slowly
AIMD Algorithms

- Adjust sending rate according to an additive increase/multiplicative decrease algorithm
  - Start slowly, increase gradually to find equilibrium
    - Add a small amount to the sending speed each time interval without loss
    - For a window-based algorithm $w_i = w_{i-1} + \alpha$ each RTT, where $\alpha = 1$ typically
  - Respond to congestion rapidly
    - Multiply sending window by some factor $\beta < 1$ each interval loss seen
    - For a window-based algorithm $w_i = w_{i-1} \times \beta$ each RTT, where $\beta = 1/2$ typically
- Faster reduction than increase $\rightarrow$ stability
Congestion in the Internet

• Network layer signals that congestion is occurring to the transport

• Two ways this is done:
  • Packet arrives at router, but queue for outgoing link is full → router discards the packet (this is the common case)
  • Packet arrives at router, queue for outgoing link is getting close to full, and transport has signalled that it understands ECN → router sets ECN-CE bit in the packet header

• Transport protocol (e.g., TCP) detects congestion signal and reacts
  • Receiver detects packet loss due to gap in sequence number space; or the receiver notices the ECN-CE mark in the packet header
  • When no congestion signal → gradual additive increase in the sending rate
  • When congestion signal received → multiplicative decrease in sending rate
  • AIMD algorithm, following Jacobson’s principles
TCP Congestion Control

• TCP uses a window-based congestion control algorithm
  • Maintains a *sliding window* onto the available data that determines how much can be sent according to the AIMD algorithm
  • Plus slow start and congestion avoidance
  • Gives approximately equal share of the bandwidth to each flow sharing a link

• The following slides give an outline of TCP Reno congestion control
  • The state of the art in TCP as of ~1990
  • See RFC 7414 (https://tools.ietf.org/html/rfc7414) for a roadmap of current TCP specifications (57 pages, referencing ~150 other documents)
  • “The world’s most baroque sliding-window protocol” – Lloyd Wood
Sliding Window Protocol: Stop and Wait

- Consider a simple stop-and-wait protocol
  - Transmit a packet of data, and then wait for acknowledgement from receiver
  - When acknowledgement received, send next packet
  - If no acknowledgement after some time out, retransmit packet

- Limits sender to one frame outstanding → poor performance
Sliding Window Protocol: Link Utilisation

• Why does stop-and-wait perform poorly?
• It takes time, $t_s$, to send a packet
  • $t_s = \text{packet size} / \text{link bandwidth}$
• Acknowledgement returns $t_{RTT}$ seconds later
• Link utilisation, $U = t_s / t_{RTT}$
  • The fraction of the time the link is in use sending packets – ideally, we want $U \approx 1.0$
  • Assume a gigabit link sending a 1500 byte packet from Glasgow to London:
    • $t_s = 1500 \times 8 \text{ bits} / 10^9 \text{ bits per second} = 0.000012s$
    • $t_{RTT} \approx 0.010 \text{ seconds}$
    • $U \approx 0.0012$
    • i.e., the link is in use 0.12% of the time
• Sliding window protocols improve on stop-and-wait by sending more than one packet before stopping for acknowledgement
Sliding Window Protocol

Improve link utilisation by allowing several frames to be outstanding
Sliding Window Protocol

Improve link utilisation by allowing several frames to be outstanding.
Sliding Window Protocol

Improve link utilisation by allowing several frames to be outstanding
Sliding Window Protocol

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

Improve link utilisation by allowing several frames to be outstanding

Acknowledgement received window slides along one packet
Sliding Window Protocol

Improve link utilisation by allowing several frames to be outstanding

Problem: how to size the window?
Should be bandwidth x delay of the path, but neither are known to the sender
TCP Congestion Control

• A sliding window protocol for TCP:
  • How to choose initial window?
  • How to find the link capacity?
    • Slow start to estimate the bottleneck link capacity
    • Congestion avoidance to probe for changes in capacity
Choosing the Initial Window

- How to choose initial window size, $W_{init}$?
  - No information $\rightarrow$ need to measure path capacity
  - Start with a small window, increase until congestion
    - $W_{init}$ of one packet per round-trip time is the only safe option, equivalent to stop-and-wait protocol, but is usually overly pessimistic
    - Traditionally, TCP used a slightly larger initial window: [RFC 3390] $W_{init} = \min(4 \times \text{MSS}, \max(2 \times \text{MSS}, 4380 \text{ bytes}))$ packets per RTT
      - e.g., Ethernet with MTU = 1500 bytes, TCP/IP headers = 40 bytes gives $W_{init} = \min(4 \times 1460, \max(2 \times 1460, 4380)) = 4380$ bytes (~3 packets)
    - Modern TCP uses an initial window of 10 packets per RTT [RFC 6928]
      - Experimental, but data from Google shows network capacity has increased enough so this is likely safe


MSS = Maximum Segment Size (MTU minus TCP/IP header size)
Finding the Link Capacity

• The initial window allows you to send
• How to choose the right window size to match the link capacity?
  Two issues:
  • How to find the correct window for the path when a new connection starts – *slow start*
  • How to adapt to changes in the available capacity once a connection is running – *congestion avoidance*
Slow Start

• Initial window, $W_{init} = 1$ packet per RTT
  • Or similar… a “slow start” to the connection

• Need to rapidly increase to the correct value for the network
  • Each acknowledgement for new data increases the window by 1 packet per RTT
  • On packet loss, immediately stop increasing window
Slow Start

- Two packets for each acknowledgement
- The window doubles on every round trip time – until loss occurs
- Rapidly finds the correct window size for the path
Congestion Avoidance

• Congestion avoidance mode used to probe for changes in network capacity
  • E.g., is sharing a connection with other traffic, and that traffic stops, meaning the available capacity increases

• Window increased by 1 packet per RTT
  • Slow, additive increase in window: $w_i = w_{i-1} + 1$
  • Until congestion is observed → respond to loss
Detecting Congestion

- TCP uses cumulative positive ACKs → two ways to detect congestion
  - Triple duplicate ACK → packet lost due to congestion
  - ACKs stop arriving → no data reaching receiver; link has failed completely somewhere
    - How long to wait before assuming ACKs have stopped?
    - $T_{rto} = \max(1 \text{ second}, \text{average } RTT + (4 \times RTT \text{ variance}))$
      - Statistical theory: 99.99% of data lies within 4σ of the mean, assuming normal distribution
        (where variance of the distribution = σ²)
Responding to Congestion

• If loss detected by triple-duplicate ACK:
  • Transient congestion, but data still being received
  • Multiplicative decrease in window: \( w_i = w_{i-1} \times 0.5 \)
  • Rapid reduction in sending speed allows congestion to clear quickly, avoids congestion collapse
Responding to Congestion

• If loss detected by time-out:
  • No packets received for a long period of time – likely a significant problem with network (e.g., link failed)
  • Return to initial sending window, and probe for the new capacity using slow start
  • Assume the route has changed, and you know nothing about the new path
Congestion Window Evolution

Typical evolution of TCP window, assuming $W_{init} = 1$

- **Slow start**
- **Congestion avoidance**

Converge on fair share of the path capacity
Congestion Window Evolution, Buffering, and Throughput

buffer size = bandwidth × delay

- Bottleneck queue never empty
- Bottleneck link never becomes idle → sending rate varies, but receiver sees continuous flow

Performance and Limitations of TCP

• TCP congestion control highly effective at keeping bottleneck link fully utilised
  • Provided sufficient buffering in the network: buffer size = bandwidth × delay
  • Packets queued in buffer → delay
  • TCP trades some extra delay to ensure high throughput

• Unless ECN used, TCP assumes loss is due to congestion
  • Too much traffic queued at an intermediate link → some packets dropped
  • This is not always true:
    • Wireless networks
    • High-speed long-distance optical networks
  • Much research into improved versions of TCP for wireless links
Summary

• Congestion control principles
  • Conservation of packets
  • Additive increase, multiplicative decrease (AIMD)

• TCP congestion control
  • Slow start
  • Congestion avoidance
  • AIMD