CS2105

Introduction to Computer Networks

AY2022/23 Semester 1

Notes by Jonathan Tay

Last updated on September 7, 2022

Contents

Part I

Introduction

1 Network Edge

Hosts (end systems) access the Internet through **access networks**, running network applications, and communicating over **links**.

Wireless access network use access points to connect hosts to routers, either via wireless LANs, e.g. Wi-Fi, or wide-area wireless access, e.g. 4G.

Hosts can connect directly to an access network physically via guided media, e.g. twisted pair cables and fiber optic cables, or over-the-air via unguided meia, e.g. radio.

2 Network Core

A mesh of interconnected routers which forward data in a network.

Transmitting data through a network takes place via **circuit switching** or **packet switching**.

2.1 Circuit Switching

Circuits along the path are reserved before transmission can begin, which mean that no other circuit can use the same path, but performance can be guaranteed.

However, there is a finite number of circuits, so the network is limited in its capacity. This approach is used in telephone networks.

2.2 Packet Switching

Messages are broken into smaller chunks, called **packets**. Packets are transmitted onto a link at a **transmission rate**, also known as **link capacity** or **bandwidth**.

The **packet transmission delay** **(**d_{trans}**)** is the time needed to transmit an L-bit packet into the link at a transmission rate R.

$$d_{\text{trans}} = \frac{L \text{ in bits}}{R \text{ in bits/sec}} \text{ seconds}$$

Packets are passed from one **router** to the next across links on the path from the source to the destination.

This incurs a **propagation delay** (d_{prop}), which depends on the length d of the physical link, and the propagation speed s in the medium.

$$d_{\text{prop}} = \frac{d}{s \approx 2 \times 10^8 \ m/s}$$

At each router, packets are **stored and forwarded**, which means an entire packet must arrive before being transmitted onto the next link.

Therefore, with P packets and N links, the **end-to-end delay**:

$$d_{\text{end-to-end}} = (P + N - 1) \cdot \frac{L}{R}$$

At the router, packets are checked for bit errors and the output link is determined using **routing algorithms**. This incurs a **nodal processing delay** (d_{proc}).

Therefore, packets have to **queue** in a **buffer** at each router, also incurring a **queueing delay** (d_{queue}), which is the time spent waiting in the queue before transmission.

In general,

$$d_{\text{end-to-end}} = d_{\text{trans}} + d_{\text{prop}} + d_{\text{queue}} + d_{\text{proc}}$$

2.3 Packet Loss

Router buffers have a finite capacity and packets arriving to a full queue will be **dropped**, resulting in **packet loss**. This is known as **buffer overflow**.

Packets can be corrupted in transit or due to noise.

2.4 Throughput

The number of bits that can be transmitted the per unit time.

Each link has its own **bandwidth** R, so throughput is measured for end-to-end communication.

$$\text{throughput} = \frac{1}{\sum_{i=1}^{n} \frac{1}{R_i}} \text{ where } n \text{ is the number of links}$$

Peak throughput and other throughput calculations are not covered in this module.

3 Network Protocols

The format and order of messages exchanged, and the actions taken after messages are sent and received.

The protocols in the Internet are arranged in a stack of **5 layers**:

1. **application**, e.g. HTTP, SMTP
2. **transport**, e.g. TCP, UDP
3. **network**, e.g. IP
4. **link**, e.g. ethernet, 802.11
5. **physical**, e.g. bits on the wire

Part II

Application Layer

Application layer protocols define the:

1. **types of messages exchanged**, e.g. requests, responses
2. **message syntax**, e.g. message fields and delineation
3. **message semantics**, i.e. meaning of information in fields
4. **rules** for when and how applications send and respond to messages

4 Architectures

In the **client-server** architecture, a **client initiates contact** with a **server**, which waits for the request before providing a service back to the client.

This relies on data centers for scaling, and clients are usually implemented in web browsers.

In the **peer-to-peer** architecture, arbitrary end systems communicate directly with each other, requesting and returning services.

This is **self-scalable** as new peers bring service capacity and demand. However, this architecture is more complex as peers are connected intermittently.

Regardless of which architecture is used, the application layer ride on the **transport layer** protocols — **TCP** or **UDP** — for data integrity, throughput, timing, and security.

5 Hypertext Transfer Protocol (HTTP)

The application layer protocol of the Internet.

HTTP uses the client-server architecture and TCP as the transport service. The client must **initiate a TCP connection** with the server before sending a **request message**.

The server receives the request message and sends the **response message** with the requested object back to the client.

The **round-trip time** (RTT) is the time taken for a packet to travel from a client to the server and back.

The **HTTP response time** in general takes one RTT to establish the TCP connection, one more RTT for the HTTP request to be fulfilled, plus the file transmission delay.

5.1 Request Message

```
1 GET /index.html HTTP/1.1\r\n
2 Host: www.example.org\r\n
3 Connection: keep—alive\r\n
4 ...
5 \r\n
6 <body>
```

Line 1 is the **request line**, specifying the **method**, **URL**, and **HTTP version**.

Lines 2 to 4 are the **header lines**, each specifying the **header field name** and **value**. Only the `Host` header is required.

The extra blank line (line 5) indicates the end of the header lines, after which the body follows.

5.2 Response Message

```
1 HTTP/1.1 200 OK\r\n
2 Date: Wed, 23 Jan 2019 13:11:15 GMT\r\n
3 Content—Length: 606\r\n // in bytes
4 Content—Type: text/html\r\n
5 ...
6 \r\n
7 <data>
```

Line 1 is the **status line** specifying the **HTTP version** and the **response status code**.

Lines 2 to 5 are the **header lines**, and lines 7 and onward contain the data requested, e.g. the HTML file.

5.3 HTTP/1.0 (non-persistent HTTP)

At most one object is sent over a TCP connection, after which the connection is closed.

Downloading multiple objects therefore requires multiple connections, incurring 2 RTTs per object in addition to the overhead for each TCP connection, which some browsers may parallelize.

5.4 HTTP/1.1 (persistent HTTP)

Unlike HTTP/1.0, the server leaves the TCP connection open after sending the response, which is reused for subsequent messages.

Persistent connections with pipelining allow multiple objects to be requested even before the server has responded to previous requests.

This reduces the total response time to as low as one RTT.

5.5 Conditional GET

Avoiding unnecessary requests for cached and up-to-date objects.

Clients send an additional `If-Modified-Since` header with the request, containing the date of last modification.

If the requested object has been modified after the date specified, then the server responds with a `200 OK` along with the requested object data.

Otherwise, the server responds with a `304 Not Modified`, which means the client can use its cached version.

5.6 Cookies

Maintaining state despite the stateless nature of HTTP.

Cookies are sent using the `Cookie` and `Set-Cookie` header fields in requests and responses respectively.

They are created by servers, stored client-side, and managed by browsers.

Cookies are sent to the server in subsequent requests, which the server can then use to execute **cookie-specific actions**, e.g. retrieve a shopping cart.

6 Domain Name System (DNS)

Computers use **IP addresses** to identify hosts and communicate, but **hostnames** (e.g. www.example.org) are easier for humans to remember.

The **domain name system** translates between a hostname and its IP addresses — multiple IPs are usually used for load-balancing.

DNS runs on the **UDP** transport protocol (chosen for its speed), which means queries can get lost or corrupted in transmission, but due to the locality of queries, such incidents are rare.

Furthermore, in the event of a query loss, browsers can simply re-issue the query, or even issue multiple queries right from the start.

Use `nslookup` or `dig` at the command line to perform a DNS query.

6.1 DNS Servers

DNS servers store resource records in distributed databases implemented in a heirarchy of many name servers.

1. **Root servers**: answer requests for records in the root zone, returning a list of authoritative name servers for the appropriate **top-level domain** (TLD).
2. **Top-level domain servers**: answer requests for `.com`, `.org`, etc., and the top-level country domains, e.g. `.sg`.
3. **Authoritative servers** belong to organizations and service providers, mapping an authoritative hostname to IP addresses.
4. **Local servers**: cache answers to DNS queries for faster access, and act as proxies to forward DNS queries if the answer is not cached.

6.2 Resource Records (RR)

Resource records format: `(name, value, type, ttl)`.

type	name	value
`A`	hostname	IP address
`CNAME`	alias name	canonical (real) name
`NS` (name server)	domain	hostname of authoritative name server
`MX` (email server)	mail server	name of mail server

`ttl`, a.k.a. **time-to-live**, is the number of seconds that a record is valid for after it is cached in a local server.

When the TTL reaches zero, it is invalidated and removed from the cache.

This also means that changes to an IP address of a host may not be immediately reflected until the TTL expires.

6.3 Domain Name Resolution

In a **recursive query**, the query is forwarded from the client through the local server, root server, TLD server, and then to the authoritative server, and the response is forwarded back to the client.

In an **iterative query**, the local DNS server handles the queries and responses directly, pinging the root, TLD, and lastly the authoritative servers, without any forwarding.

Both query methods are valid, but iterative querying is used in practice.

7 Socket Programming

A **process** is a program running on a host.

Within the same host, the OS can define inter-process communication, but processes on different hosts communicate by exchanging messages.

Processes are identified by an **IP address *and* port number**.

type	number of bits
IPv4	32
IPv6	128
port	16

A **socket** is the software interface — a set of APIs — between processes and transport layer protocols.

Processes send and receive messages via a socket.

7.1 via UDP

The sender attatches the **destination IP address *and* port number** to *each packet*, which the receiver extracts.

1. Client creates `clientSocket`.
2. Server creates `serverSocket`.
3. Client creates packet with `serverIP` and port `x`, sent via `clientSocket`.
4. Server reads the **datagram** from `serverSocket`.
5. Sever sends the reply specifying the client address and port number via `serverSocket`.
6. Client reads the datagram from `clientSocket`.
7. `clientSocket` is closed.

No connection is established between the client and the server. This method of transmission via datagrams over UDP is *unreliable*.

7.2 via TCP

When contacted by a client, the server TCP connection creates a *new socket* for the server process to communicate with that client.

This allows the server to communicate with *multiple clients simultaneously*.

1. Server creates `serverSocket` on port `x`.
2. Client creates `clientSocket`, connecting to `serverIP` on port `x`.
3. Client and server set up a TCP connection.
4. Client and server exchange requests using `clientSocket` and `connectionSocket` respectively.
5. Server closes `connectionSocket`.
6. Client closes `clientSocket`.

Part III

Transport Layer

Transport layer services deliver messages between processes on different hosts, while packet switches check the destination IP address for routing.

The **sender** breaks messages into **segments**, and passes them to the *network layer*.

The **receiver** reassembles the segments into the message, and passes it the *application layer*.

8 User Datagram Protocol (UDP)

UDP transmission is **unreliable** but fast, often used only by loss-tolerant and rate-sensitive applications, e.g. multimedia.

UDP adds the following on top of IP:

- **multiplexing** at sender,
- **de-multiplexing** at receiver, and
- **checksum** at sender and receiver.

At the receiver, every datagram with the same *destination port number* will be directed to the *same UDP socket*.

UDP adds the following headers to each segment:

1. **source port number**,
2. **destination port number**,
3. **segment length**, and
4. **checksum**.

UDP exists because it is:

- **fast**: no connection establishment $\implies$ no delay,
- **simple**: no connection state,
- **small**: header size is small,
- **unlimited**: no congestion control.

8.1 UDP Checksum

Checksums are used to detect transmission errors in each segment, e.g. flipped bits.

Steps to compute the checksum:

1. Partition the segment into 16-bit integers.
2. Perform **binary addition** with every integer.
3. Add any *carry bits* to the next integer.
4. Take the 1s complement of the final integer to get the checksum.

9 Reliable Data Transfer (RDT)

The underlying network layer may *corrupt*, *drop*, *re-order*, or *delay* packets during transmission — it is **inherently unreliable**.

RDT protocols aim to *guarantee packet delivery* (in the order sent) and *correctness*.

RDT protocols are purely hypothetical — they only form the basis for real-life implementations.

9.1 RDT 1.0

We assume that the underlying channel is *perfectly reliable*.

The sender sends data through the channel, and the receiver reads the data from the channel.

9.2 RDT 2.0

We introduce **bit errors** into the otherwise perfect underlying channel.

Now the receiver has to validate the **checksum** to *detect* bit errors, and reply with either of the following:

- **acknowledgement** (ACK): if the packet was received correctly, or,
- **negative acknowledgement** (NAK): if the packet was corrupted, requesting re-transmission.

The sender will re-transmit the packet after receiving an NAK, *or* if either the ACK or NAK is corrupted.

However, this is a **stop-and-wait** protocol, as the sender has to wait for each acknowledgement before sending the next packet.

9.3 RDT 2.1

Because ACKs can get corrupted, causing duplicate packets to be transmitted to the receiver in RDT 2.0, we add a **sequence number** to each packet.

If the receiver identifies a duplicate packet, it simply *discards* it — not sending it to the application.

9.4 RDT 2.2

We want a **NAK-free** protocol — such that the receiver *only* sends ACKs.

The NAKs are replaced by ACKs, and *all* ACKs are appended with the sequence number of the last packet that was received correctly.

If the sender receives *duplicate* or *corrupted* ACKs, it re-transmits the current packet.

9.5 RDT 3.0

We introduce **packet loss** and **delays** to the underlying channel.

These are handled via **timeouts** which trigger re-transmission if an ACK is either corrupted, lost, or delayed.

Because re-transmission may duplicate packets, we still use the sequence number to discard duplicate packets.

However, this has **low throughput** and **low utilization**, because a sender spends most of its time waiting for an ACK.

9.6 Pipelined Protocols

We can increase throughput by **pipelining** — sending multiple yet-to-be-acknowledged packets at a time.

The **window size** controls the number of packets sent at a time.

This increases the *throughput* and *utilization*, but at a cost:

- increased range of sequence numbers
- buffering at sender and receiver
- increased design complexity

9.6.1 Go-back-*n* (GBN)

A GBN sender tracks 2 things:

1. a **sliding window** of n contiguous packets, and,
2. a **timer** for the oldest unACK'd packet, which is re-transmitted in the window when it expires.

A GBN receiver demands that all packets are received in order:

- **ACK** every successive packet that is received in order
- **discard** every out-of-order packet (i.e. if packet l is lost, every packet $> l$ is discarded until packet l is received correctly)

GBN is also known as **cumulative ACK** — an ACK means that every packet up to that point has been received correctly.

GBN behaves exactly like RDT 3.0 when $n = 1$.

The timeout has to be chosen correctly — too short and it results in **premature timeouts**, too large and the protocol becomes slow.

9.6.2 Selective Repeat (SR)

SR buffers out-of-order packets instead of dropping them.

An SR receiver ACKs individual packets.

An SR sender maintains a timer for each unACK'd packet, re-sending *only that packet* when its timer expires.

However, the sliding window of the sender cannot advance past any unACK'd packet until it is ACK'd.

10 Transmission Control Protocol (TCP)

TCP has several characteristics:

- **point-to-point**:
  one sender, one receiver
- **connection-oriented**:
  handshakes before transmission

- **full-duplex service**:
  bi-directional data flow
- **reliable, in-order byte stream**:
  bytes are labelled by sequence numbers

Every TCP connection (socket) is identified by a tuple:

1. the **source IP address**,
2. the **source port number**,
3. the **destination IP address**, and,
4. the **destination port number**.

The **maximum segment size** is **1460 bytes**, but this is *not inclusive* of 20 bytes for the **TCP packet headers**:

1. **source port number**: 16 bits
2. **destination port number**: 16 bits
3. **sequence number (`seq`)**:
   32-bit byte number of the first byte in the segment
4. **acknowledgement number (`ack`)**:
   32-bit byte number of the next expected byte
5. **`ACK` bit (`A`)**:
   `1` if the ACK is in use, `0` otherwise
6. **`SYN` bit (`S`)**:
   `1` if this packet is establishing a new connection, `0` otherwise
7. **`FIN` bit (`F`)**:
   `1` if this packet is closing the connection, `0` otherwise
8. **checksum**: 32 bits

10.1 Connection

TCP employs a **3-way handshake** to *establish a connection* before exchanging application data:

1. **client → server**:
   `S = 1, seq = x`
2. **server → client**:
   `S = 1, seq = y, A = 1, ack = x + 1`
3. **client → server**:
   `A = 1, seq = x + 1, ack = y + 1,`
   `data = ...`

The client and server choose their **initial sequence numbers** x and y *randomly* and *independently*.

To *close a connection*:

1. **client → server**:
   `F = 1, seq = x`
2. **server → client**:
   `A = 1, ack = u + 1`
3. **server → client**:
   `F = 1, seq = t`
4. **client → server**:
   `A = 1, ack = t + 1`

10.2 Acknowledgements

TCP is also a **cumulative ACK** protocol, but handling of out-of-order segments is implementation-dependent.

Both the sender and receiver have a **buffer** for sent or received **segments**.

A TCP sender tracks 2 things:

1. the **next sequence number**, which is the first byte of the next segment to send, and,
2. a *single* **timer** for the oldest unACK'd packet, which is re-transmitted in the window when it expires.

There are 4 potential actions for a TCP receiver to take when it receives a segment:

1. **in-order, first non-ACK'd segment**:
   delay ACK for up to 500ms for the next segment, otherwise ACK with `ack = seq + len(data)`, which is equivalent to the next expected byte
2. **in-order, previous segment pending ACK**:
   send *single* cumulative ACK, ACKing both segments with `ack = seq` of the next expected byte
3. **out-of-order (gapped up) segment**:
   send *duplicate* ACKs with `ack = seq` of the expected byte
4. **gap-fill segment**:
   send ACK with `ack = seq` of the next expected byte if the packet fills the lowest end of the gap, otherwise it is still out-of-order, so repeat above.

ACK packets can also contain data (**piggy-back**), which is useful for bi-directional communication as it halves the number of packets needed.

10.3 Timeout

The timeout interval is constantly computed based on the **estimated RTT**:

$\mathrm{RTT}_{\mathtt{est}} = (1-\alpha)\,\mathrm{RTT}_{\mathtt{est}} + \alpha\,\mathrm{RTT}_{\mathtt{sample}}$
$\mathrm{RTT}_{\mathtt{dev}} = (1-\beta)\,\mathrm{RTT}_{\mathtt{dev}} + \beta\,\left\lvert \mathrm{RTT}_{\mathtt{sample}} - \mathrm{RTT}_{\mathtt{est}} \right\rvert$
$\mathtt{timeout} = \mathrm{RTT}_{\mathtt{est}} + 4 \times \mathrm{RTT}_{\mathtt{dev}}$

Typically, $\alpha = 0.125$ and $\beta = 0.25$.

The timeout interval has a **safety margin** factor of 4 $\mathrm{RTT}_{\mathtt{dev}}$.

TCP also employs **fast retransmission** — the sender resends segments immediately after receiving 4 ACKs for the same segment, assuming that they were lost.