CS2105
Introduction to Computer Networks

AY2022/23 Semester 1

Notes by Jonathan Tay

Last updated on August 19, 2022

Contents

1 Introduction **1**

1.1 Network Edge . . . 1

1.2 Network Core . . . 1

1.2.1 Circuit Switching . . . 1

1.2.2 Packet Switching . . . 1

1.2.3 Packet Loss . . . 1

1.2.4 Throughput . . . 1

1.3 Network Protocols . . . 1

2 Application Layer **1**

2.1 Architectures . . . 1

2.2 Hypertext Transfer Protocol (HTTP) . . . 2

2.2.1 Request Message . . . 2

2.2.2 Response Message . . . 2

2.2.3 HTTP/1.0 (non-persistent HTTP) . . . 2

2.2.4 HTTP/1.1 (persistent HTTP) . . . 2

2.2.5 Conditional `GET` . . . 2

2.2.6 Cookies . . . 2

2.3 Domain Name System (DNS) . . . 2

2.3.1 DNS Servers . . . 3

2.3.2 Resource Records (RR) . . . 3

2.3.3 Domain Name Resolution . . . 3

2.4 Socket Programming . . . 3

2.4.1 via UDP . . . 3

2.4.2 via TCP . . . 3

1 Introduction

1.1 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.

1.2 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**.

1.2.1 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.

1.2.2 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 routers, 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,

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

1.2.3 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.

1.2.4 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.

1.3 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

2 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

2.1 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.

2.2 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.

2.2.1 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.

2.2.2 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
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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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.

2.3 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.

2.3.1 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.

2.3.2 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.

2.3.3 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.

2.4 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.

2.4.1 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*.

2.4.2 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`.