CS 438 MT1 Review

ACM @ UIUC

September 26, 2024

Disclaimers and Logistics

- **Disclaimer:** We have not seen the exam. We have no idea what the questions are. However, we've taken the course and reviewed material/practice exams, so we have **suspicions** as to what the questions will be like.
- This review session is being recorded. Recordings and slides will be distributed on Piazza and the ACM site at the end.
- **Agenda:** We'll review all topics likely to be covered, working through some examples that may look like exam questions as we go, then review individual topics by request.
  - Questions are designed to be written in the same style as previous exams but to be *slightly* harder, so don't worry if you don't get everything right away!
- Please let us know if we're going too fast/slow, not speaking loud enough/speaking too loud, etc.
- If you have a question anytime during the review session, please ask! Someone else almost surely has a similar question.
- We'll provide a feedback form at the end of the session.

Foundations I: Resource Sharing

- A **network** is just a set of **elements** (servers, routers, etc.) connected together, that implements a set of **protocols** for the purpose of **sharing resources** at the end hosts
  - Visualized as graph with elements as nodes and **links** as the edges connecting them

Foundations I: Resource Sharing

- A **network** is just a set of **elements** (servers, routers, etc.) connected together, that implements a set of **protocols** for the purpose of **sharing resources** at the end hosts
  - Visualized as graph with elements as nodes and **links** as the edges connecting them
  - Only one goal- deliver the data between edge nodes

Foundations I: Resource Sharing

- A **network** is just a set of **elements** (servers, routers, etc.) connected together, that implements a set of **protocols** for the purpose of **sharing resources** at the end hosts
  - Visualized as graph with elements as nodes and **links** as the edges connecting them
  - Only one goal- deliver the data between edge nodes
- **Statistical Multiplexing**: combining demands to share resources without overbuilding capacity, as aggregate of peak demand is much bigger than peak of aggregate demand. Two approaches:

Foundations I: Resource Sharing

- A **network** is just a set of **elements** (servers, routers, etc.) connected together, that implements a set of **protocols** for the purpose of **sharing resources** at the end hosts
  - Visualized as graph with elements as nodes and **links** as the edges connecting them
  - Only one goal- deliver the data between edge nodes
- **Statistical Multiplexing**: combining demands to share resources without overbuilding capacity, as aggregate of peak demand is much bigger than peak of aggregate demand. Two approaches:
  - **Reservations/Circuit Switching**: Source sends call request, path between source and destination reserved + blocked off, communication happens, then circuit teardown. Used in some telephone + ATM protocols. Can share a channel using FDM (split frequencies) or TDM (round-robin whole resource)

Foundations I: Resource Sharing

- A **network** is just a set of **elements** (servers, routers, etc.) connected together, that implements a set of **protocols** for the purpose of **sharing resources** at the end hosts
  - Visualized as graph with elements as nodes and **links** as the edges connecting them
  - Only one goal- deliver the data between edge nodes
- **Statistical Multiplexing**: combining demands to share resources without overbuilding capacity, as aggregate of peak demand is much bigger than peak of aggregate demand. Two approaches:
  - **Reservations/Circuit Switching**: Source sends call request, path between source and destination reserved + blocked off, communication happens, then circuit teardown. Used in some telephone + ATM protocols. Can share a channel using FDM (split frequencies) or TDM (round-robin whole resource)
  - **Packets/Datagrams**: Packets contain data (body) + information on how/where to send it and where it came from (headers). No underutilization/blocked connections/setup costs and can route around link failures, but no guarantees on availiability/delay, and overhead from headers. Used basically everywhere.

Foundations II: Internet as IPC

- Main problem: **inter-process communication**, where processes may not be on the same machine

End-To-End Story

Foundations II: Internet as IPC

- Main problem: **inter-process communication**, where processes may not be on the same machine

End-To-End Story

- Program opens **socket** which allows it to connect to **network stack**

Foundations II: Internet as IPC

- Main problem: **inter-process communication**, where processes may not be on the same machine

End-To-End Story

- Program opens **socket** which allows it to connect to **network stack**
- **DNS** maps name of target to **address**

Foundations II: Internet as IPC

- Main problem: **inter-process communication**, where processes may not be on the same machine

End-To-End Story

- Program opens **socket** which allows it to connect to **network stack**
- **DNS** maps name of target to **address**
- Network stack embeds address and **port** of source/destination in datagram headers

Foundations II: Internet as IPC

- Main problem: **inter-process communication**, where processes may not be on the same machine

End-To-End Story

- Program opens **socket** which allows it to connect to **network stack**
- **DNS** maps name of target to **address**
- Network stack embeds address and **port** of source/destination in datagram headers
- **Routers** create **routing tables** to decide which **outgoing link** to send packets along (knowing only local information). When link is free, forward packet to next router

Foundations II: Internet as IPC

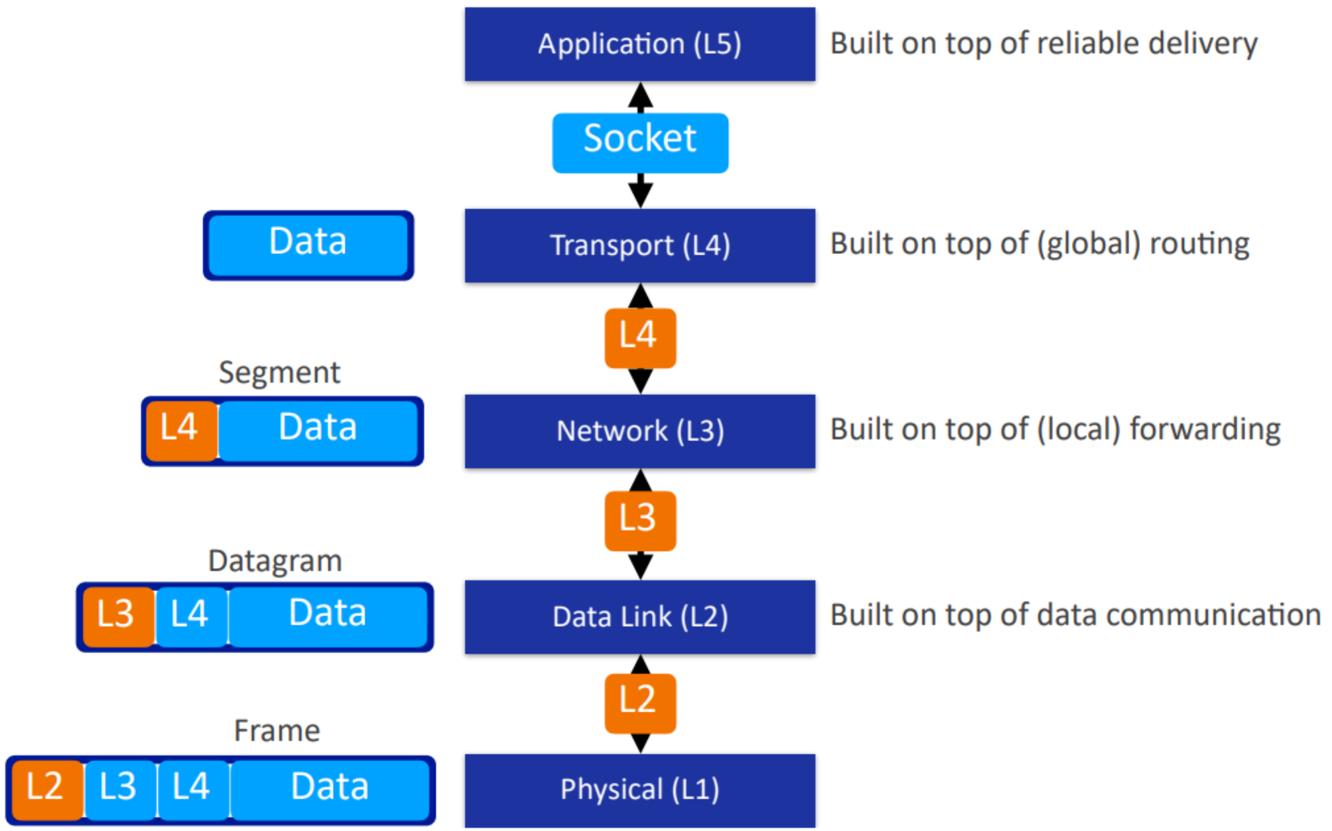
- Main problem: **inter-process communication**, where processes may not be on the same machine

End-To-End Story

- Program opens **socket** which allows it to connect to **network stack**
- **DNS** maps name of target to **address**
- Network stack embeds address and **port** of source/destination in datagram headers
- **Routers** create **routing tables** to decide which **outgoing link** to send packets along (knowing only local information). When link is free, forward packet to next router
- When packet arrives at destination, forward to correct application

- Goal: Nodes shouldn't have to worry about the implementation details of other nodes, just the correct decision locally (modularity)

Foundations III: Layering



Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1 - (1 - 10^{-6})^{10240} = 1.0\%$

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1 - (1 - 10^{-6})^{10240} = 1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1 - (1 - 10^{-6})^{10240} = 1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get
  - **Throughput**: Number of *total* correctly delivered bits in unit time

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1-(1-10^{-6})^{10240}=1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get
  - **Throughput**: Number of *total* correctly delivered bits in unit time
  - **Goodput**: Number of *application-layer* correctly delivered bits in unit time

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1 - (1 - 10^{-6})^{10240} = 1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get
  - **Throughput**: Number of *total* correctly delivered bits in unit time
  - **Goodput**: Number of *application-layer* correctly delivered bits in unit time
- **Timing**: some applications need packets to be delivered right away (video games/live conferencing/etc.). **Per Packet Latency** = Transmit time + Propagation delay + Process time + Queuing Delay. Assuming packet length L, transmit rate R bps

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1 - (1 - 10^{-6})^{10240} = 1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get
  - **Throughput**: Number of *total* correctly delivered bits in unit time
  - **Goodput**: Number of *application-layer* correctly delivered bits in unit time
- **Timing**: some applications need packets to be delivered right away (video games/live conferencing/etc.). **Per Packet Latency** = Transmit time + Propagation delay + Process time + Queuing Delay. Assuming packet length L, transmit rate R bps
  - **Transmit Time**: Time used by transmitter to write packet to wire ($\frac{L}{R}$)

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1 - (1 - 10^{-6})^{10240} = 1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get
  - **Throughput**: Number of *total* correctly delivered bits in unit time
  - **Goodput**: Number of *application-layer* correctly delivered bits in unit time
- **Timing**: some applications need packets to be delivered right away (video games/live conferencing/etc.). **Per Packet Latency** = Transmit time + Propagation delay + Process time + Queuing Delay. Assuming packet length L, transmit rate R bps
  - **Transmit Time**: Time used by transmitter to write packet to wire ($\frac{L}{R}$)
  - **Propagation Delay**: Time for packet to travel from transmitter to receiver ($\frac{\text{distance}}{\text{velocity}}$)

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1 - (1 - 10^{-6})^{10240} = 1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get
  - **Throughput**: Number of *total* correctly delivered bits in unit time
  - **Goodput**: Number of *application-layer* correctly delivered bits in unit time
- **Timing**: some applications need packets to be delivered right away (video games/live conferencing/etc.). **Per Packet Latency** = Transmit time + Propagation delay + Process time + Queuing Delay. Assuming packet length L, transmit rate R bps
  - **Transmit Time**: Time used by transmitter to write packet to wire ($\frac{L}{R}$)
  - **Propagation Delay**: Time for packet to travel from transmitter to receiver ($\frac{\text{distance}}{\text{velocity}}$)
  - **Process Time**: Time required for router to read header + decide route

Foundations IV: Desiderata

- **Data Loss**: some applications (video/audio) can tolerate it, others (file transfer/ssh/etc.) can't
  - **Packet Error Rate** = $\frac{N}{P}$ for N unrecoverable bit errors for P packets. Packets can be recovered through error-correcting coding, depending on schema.
  - Example: if bit error rate is 10^{-6}, packet 10kb, no ECC, then PER is $1-(1-10^{-6})^{10240}=1.0\%$
- **Bandwidth**: some applications (video streaming) require some amount to be effective, but "elastic apps" can use whatever bandwidth they get
  - **Throughput**: Number of *total* correctly delivered bits in unit time
  - **Goodput**: Number of *application-layer* correctly delivered bits in unit time
- **Timing**: some applications need packets to be delivered right away (video games/live conferencing/etc.). **Per Packet Latency** = Transmit time + Propagation delay + Process time + Queuing Delay. Assuming packet length L, transmit rate R bps
  - **Transmit Time**: Time used by transmitter to write packet to wire ($\frac{L}{R}$)
  - **Propagation Delay**: Time for packet to travel from transmitter to receiver ($\frac{\text{distance}}{\text{velocity}}$)
  - **Process Time**: Time required for router to read header + decide route
  - **Queuing Delay**: Time that a packet waits in queue because link is busy. In expectation, proportional to $\frac{La}{r}$ with a packets in queue.

Foundations V: Signal Processing

- **Frequency Band** refers to the range of frequencies used for a signal. **Bandwidth** is the width of this band, is proportional to data rate.
- Frequency bands may be divided up into smaller **channels** for simultaneous communication

Foundations V: Signal Processing

- **Frequency Band** refers to the range of frequencies used for a signal. **Bandwidth** is the width of this band, is proportional to data rate.
- Frequency bands may be divided up into smaller **channels** for simultaneous communication
- **Carrier Frequency**: Fixed (higher) frequency used to carry signal. Options include **Amplitude Shift Keying**, **Frequency Shift Keying**

Foundations V: Signal Processing

- **Frequency Band** refers to the range of frequencies used for a signal. **Bandwidth** is the width of this band, is proportional to data rate.
- Frequency bands may be divided up into smaller **channels** for simultaneous communication
- **Carrier Frequency**: Fixed (higher) frequency used to carry signal. Options include **Amplitude Shift Keying**, **Frequency Shift Keying**
- **Signal to Interference and Noise Ratio**: $\frac{P_{\text{signal}}}{P_{\text{noise}}+P_{\text{interference}}}$. Bit error rate is a function of this.

Theorem (Shannon Capacity)

$C = B \log_2(1 + SINR)$

- *Capacity (C) in bits per second*
- *Bandwidth (B) in Hz*

Application Layer: Transport Protocols

- Applications can choose between TCP and UDP

Application Layer: Transport Protocols

- Applications can choose between TCP and UDP
- TCP
  - Uses connection setup between processes
  - Provides a **reliable transport** guarantee w.r.t. correctness/order/duplication
  - **Flow control**: sender won't overwhelm receiver
  - **Congestion control**: Will slow down to avoid network overload
  - No guarantees on timing, bandwidth

Application Layer: Transport Protocols

- Applications can choose between TCP and UDP
- TCP
  - Uses connection setup between processes
  - Provides a **reliable transport** guarantee w.r.t. correctness/order/duplication
  - **Flow control**: sender won't overwhelm receiver
  - **Congestion control**: Will slow down to avoid network overload
  - No guarantees on timing, bandwidth
- UDP
  - Unreliable data transfer between sender and receiver. No fancy control/ordering systems.

Application Layer: Transport Protocols

- Applications can choose between TCP and UDP
- TCP
  - Uses connection setup between processes
  - Provides a **reliable transport** guarantee w.r.t. correctness/order/duplication
  - **Flow control**: sender won't overwhelm receiver
  - **Congestion control**: Will slow down to avoid network overload
  - No guarantees on timing, bandwidth
- UDP
  - Unreliable data transfer between sender and receiver. No fancy control/ordering systems.
- Most internet protocols (HTTP/FTP/SMTP/etc.) are built on TCP, but a lot of video streaming/VoIP/trading systems use UDP

Application Layer: DNS

- A **distributed database** implemented as a hierarchy of **name servers** to **resolve** domain names as IP addresses at the application layer
  - 13 **root** DNS servers, thousands of TLD DNS servers (.com/.edu/.org/.uk), **authoritative** DNS servers set up by organization or service provider providing an authoritative source for organization’s servers (web, mail, etc.)

Application Layer: DNS

- A **distributed database** implemented as a hierarchy of **name servers** to **resolve** domain names as IP addresses at the application layer
  - 13 **root** DNS servers, thousands of TLD DNS servers (.com/.edu/.org/.uk), **authoritative** DNS servers set up by organization or service provider providing an authoritative source for organization’s servers (web, mail, etc.)
- **Iterative Querying**: Server either returns the record, or an address of a DNS server who might

Application Layer: DNS

- A **distributed database** implemented as a hierarchy of **name servers** to **resolve** domain names as IP addresses at the application layer
  - 13 **root** DNS servers, thousands of TLD DNS servers (.com/.edu/.org/.uk), **authoritative** DNS servers set up by organization or service provider providing an authoritative source for organization's servers (web, mail, etc.)
- **Iterative Querying**: Server either returns the record, or an address of a DNS server who might
- **Recursive Querying**: Server returns record, asking other servers if needed

Application Layer: DNS

- A **distributed database** implemented as a hierarchy of **name servers** to **resolve** domain names as IP addresses at the application layer
  - 13 **root** DNS servers, thousands of TLD DNS servers (.com/.edu/.org/.uk), **authoritative** DNS servers set up by organization or service provider providing an authoritative source for organization's servers (web, mail, etc.)
- **Iterative Querying**: Server either returns the record, or an address of a DNS server who might
- **Recursive Querying**: Server returns record, asking other servers if needed
- When any DNS server learns a mapping, it **caches** it (which times out and disappears eventually). Most common TLD servers are often cached locally, meaning root name server unusual.

Application Layer: DNS

- A **distributed database** implemented as a hierarchy of **name servers** to **resolve** domain names as IP addresses at the application layer
  - 13 **root** DNS servers, thousands of TLD DNS servers (.com/.edu/.org/.uk), **authoritative** DNS servers set up by organization or service provider providing an authoritative source for organization's servers (web, mail, etc.)
- **Iterative Querying**: Server either returns the record, or an address of a DNS server who might
- **Recursive Querying**: Server returns record, asking other servers if needed
- When any DNS server learns a mapping, it **caches** it (which times out and disappears eventually). Most common TLD servers are often cached locally, meaning root name server unusual.
- 4 types of records: A (hostname `name` is IP address `value`), NS (authoritative name server for `name` can be found at `value`), CNAME (the "real" name for alias `name` is the canonical name `value`), MX (the mailserver for `name` has name `value`)

Application Layer: DNS

- A **distributed database** implemented as a hierarchy of **name servers** to **resolve** domain names as IP addresses at the application layer
  - 13 **root** DNS servers, thousands of TLD DNS servers (.com/.edu/.org/.uk), **authoritative** DNS servers set up by organization or service provider providing an authoritative source for organization's servers (web, mail, etc.)
- **Iterative Querying**: Server either returns the record, or an address of a DNS server who might
- **Recursive Querying**: Server returns record, asking other servers if needed
- When any DNS server learns a mapping, it **caches** it (which times out and disappears eventually). Most common TLD servers are often cached locally, meaning root name server unusual.
- 4 types of records: A (hostname `name` is IP address `value`), NS (authoritative name server for `name` can be found at `value`), CNAME (the "real" name for alias `name` is the canonical name `value`), MX (the mailserver for `name` has name `value`)
- Inserting Records: Provide registrar with name and IP of authorative name server, registrar inserts NS record for auth server name and A record for auth server IP

Application Layer: Architectures

- **Client-Server**
  - Server: always-on host with constant address, Clients: communicate only with servers and not with each other, may disconnect/reconnect, change IP addresses.

Application Layer: Architectures

- **Client-Server**
  - Server: always-on host with constant address, Clients: communicate only with servers and not with each other, may disconnect/reconnect, change IP addresses.
- **Peer-to-Peer**
  - No always-on server, peers might disconnect, change addresses. Scalable, but sometimes difficult to manage. Examples: CHORD, Gnutella

Application Layer: Architectures

- **Client-Server**
  - Server: always-on host with constant address, Clients: communicate only with servers and not with each other, may disconnect/reconnect, change IP addresses.
- **Peer-to-Peer**
  - No always-on server, peers might disconnect, change addresses. Scalable, but sometimes difficult to manage. Examples: CHORD, Gnutella
- Many services use hybrid (ex: video conferencing/instant messaging: users directly connect with each other but use central server to register/look up *where* users are)

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.
- **HTTP is “stateless”**. Servers maintain no information about previous requests.

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.
- **HTTP is “stateless”**. Servers maintain no information about previous requests.
- **Non-Persistent HTTP** (HTTP/1.0): Connection closes after one response. What’s the total response time?

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.
- **HTTP is “stateless”**. Servers maintain no information about previous requests.
- **Non-Persistent HTTP** (HTTP/1.0): Connection closes after one response. What’s the total response time? **2 $\times$ RTT + file transmit time** per object.

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.
- **HTTP is "stateless"**. Servers maintain no information about previous requests.
- **Non-Persistent HTTP** (HTTP/1.0): Connection closes after one response. What's the total response time? **$2 \times$ RTT + file transmit time** per object.
- **Persistent HTTP without pipelining**: Connection stays open, but waits for one message response before the next one is sent. What's the total response time?

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.
- **HTTP is "stateless"**. Servers maintain no information about previous requests.
- **Non-Persistent HTTP** (HTTP/1.0): Connection closes after one response. What's the total response time? **2 $\times$ RTT + file transmit time** per object.
- **Persistent HTTP without pipelining**: Connection stays open, but waits for one message response before the next one is sent. What's the total response time? **(# of referenced objects + 1) $\times$ RTT + data transmit time**

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.
- **HTTP is “stateless”**. Servers maintain no information about previous requests.
- **Non-Persistent HTTP** (HTTP/1.0): Connection closes after one response. What’s the total response time? **2 $\times$ RTT + file transmit time** per object.
- **Persistent HTTP without pipelining**: Connection stays open, but waits for one message response before the next one is sent. What’s the total response time? **(# of referenced objects + 1) $\times$ RTT + data transmit time**
- **Persistent HTTP *with* pipelining** (HTTP/1.1 default): Connection stays open, *and* client requests a file as soon as it’s referenced. What’s the min possible total response time?

Application Layer: HTTP

- Web pages consist of **objects** (HTML, images, javascript, etc.), each of which has a URL
- **HTTP uses a client-server model** where clients request + render, servers send objects in response to requests.
- **HTTP uses TCP**. Clients connect to port 80 of host, messages are exchanged between brower + server.
- **HTTP is "stateless"**. Servers maintain no information about previous requests.
- **Non-Persistent HTTP** (HTTP/1.0): Connection closes after one response. What's the total response time? **2 $\times$ RTT + file transmit time** per object.
- **Persistent HTTP without pipelining**: Connection stays open, but waits for one message response before the next one is sent. What's the total response time? **(# of referenced objects + 1) $\times$ RTT + data transmit time**
- **Persistent HTTP *with* pipelining** (HTTP/1.1 default): Connection stays open, *and* client requests a file as soon as it's referenced. What's the min possible total response time? **Setup + data transmit + 1 RTT for *all* objects**.

Application Layer: HTTP

- Two types of messages: **request**, **response**. Headers in ASCII (except for HTTP/2 or later versions).
  - Example Request:

```
GET / HTTP/1.1
Host: illinois.edu
User-Agent: curl/8.9.1
Accept: */*
```

Application Layer: HTTP

- Two types of messages: **request**, **response**. Headers in ASCII (except for HTTP/2 or later versions).
  - Example Request:

    ```
    GET / HTTP/1.1
    Host: illinois.edu
    User-Agent: curl/8.9.1
    Accept: */*
    ```

  - Method types: GET (gets requested file, can have information in URL parameters), POST (uploads body to server), HEAD (leaves everything but headers out), PUT (uploads file to path in URL field), DELETE (deletes file in URL field)

Application Layer: HTTP

- Two types of messages: **request**, **response**. Headers in ASCII (except for HTTP/2 or later versions).
  - Example Request:

```
GET / HTTP/1.1
Host: illinois.edu
User-Agent: curl/8.9.1
Accept: */*
```

  - Method types: GET (gets requested file, can have information in URL parameters), POST (uploads body to server), HEAD (leaves everything but headers out), PUT (uploads file to path in URL field), DELETE (deletes file in URL field)
  - Example Response:

```
HTTP/1.1 200 OK
Date: Mon, 21 Oct 2024 23:15:43 GMT
Server: Apache/2.4.57 (Red Hat Enterprise Linux) OpenSSL/3.0.7
Last-Modified: Mon, 23 Sep 2024 21:24:01 GMT
ETag: "eac6-622d001ecb792"
Accept-Ranges: bytes
Content-Length: 60102
Content-Type: text/html; charset=UTF-8
```

Application Layer: Caching

- **Goal**: Satisfy client request without involving origin server

Application Layer: Caching

- **Goal**: Satisfy client request without involving origin server
- Browser sends all requests to cache, which acts as both client and server. If cache has file, returns immediately, else requests from server and returns. **Which requests does this help?**

Application Layer: Caching

- **Goal**: Satisfy client request without involving origin server
- Browser sends all requests to cache, which acts as both client and server. If cache has file, returns immediately, else requests from server and returns. **Which requests does this help?**
- Can use **conditional GET requests**. Add `If-modified-since` field to headers; if not modified, return status 304, else return file. Ensures that requests are up-to-date while still saving bandwidth. **Why?**

Application Layer: SMTP

- Uses TCP on port 25 to send mail
- Sending mail server acts as “client”, while recieving server acts as “server”. This makes it a “push” protocol, rather than a “pull” protocol (like HTTP)
- Three phases of transfer: handshake, message transfer, closure. Commands in ASCII, response is status code + message.
- Users access email boxes via **user agents** (POP3/IMAP/webmail).
- Lots more details, but they’re highly unlikely to come up on an exam.

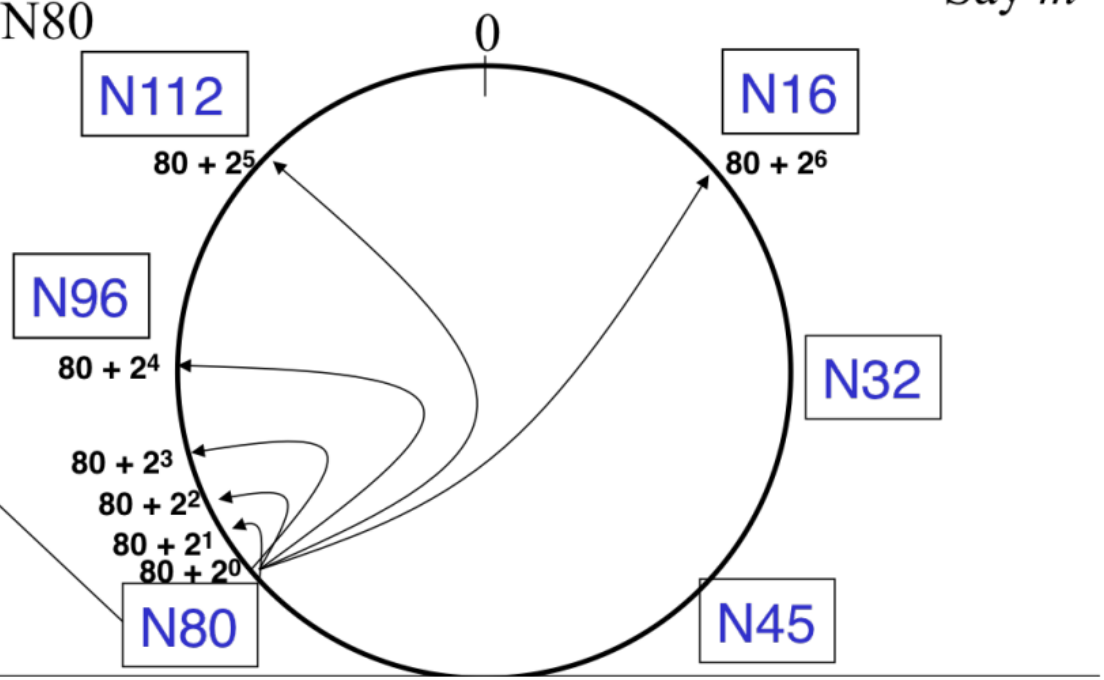
Application Layer: CHORD

- Each file assigned a hash and assigned to the next highest node, each server knows a "finger table" of nodes exponentially far away from current id, recursive lookup structure.

Say m=7

Finger Table at N80

i	$ft[i]$
0	96
1	96
2	96
3	96
4	96
5	112
6	16



Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order
  - Packets can be delayed arbitrarily long

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order
  - Packets can be delayed arbitrarily long
  - Packets can be duplicated

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order
  - Packets can be delayed arbitrarily long
  - Packets can be duplicated
  - Packets can be corrupted

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order
  - Packets can be delayed arbitrarily long
  - Packets can be duplicated
  - Packets can be corrupted
  - We can overwhelm a sending/receiving buffer (thereby dropping packets)

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order
  - Packets can be delayed arbitrarily long
  - Packets can be duplicated
  - Packets can be corrupted
  - We can overwhelm a sending/receiving buffer (thereby dropping packets)
  - If we transmit too much, we can interfere with other communication going on

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order
  - Packets can be delayed arbitrarily long
  - Packets can be duplicated
  - Packets can be corrupted
  - We can overwhelm a sending/receiving buffer (thereby dropping packets)
  - If we transmit too much, we can interfere with other communication going on
- We don't want to use any information about lower/higher layers

Transport Layer: Goals

- We need to provide a **reliable data stream** to the receiver's application from the sender's application. However, there's a lot that can go wrong:
  - Packets can be lost
  - Packets can arrive out-of-order
  - Packets can be delayed arbitrarily long
  - Packets can be duplicated
  - Packets can be corrupted
  - We can overwhelm a sending/receiving buffer (thereby dropping packets)
  - If we transmit too much, we can interfere with other communication going on
- We don't want to use any information about lower/higher layers
- Also, distributed consensus is *hard*. Some of what we want to do is the **Two Generals' Problem**: since message acknowledgments are as likely to be lost as messages, we'd potentially need infinite messages to come to consensus safely.

Transport Layer: Basic Validation

- Most protocols: to avoid out of sequence ordering, assign each packet a **sequence number**. Send an **acknowledgment** (ACK) if packet received.

Transport Layer: Basic Validation

- Most protocols: to avoid out of sequence ordering, assign each packet a **sequence number**. Send an **acknowledgment** (ACK) if packet received.
- Packets can be corrupted. Receiver can calculate **checksum** to verify validity, as well as run ECCs if protocol provides (Hamming codes, etc.) Some protocols send NACKs to indicate that packet has been rejected, but most just ignore packet.

Transport Layer: Basic Validation

- Most protocols: to avoid out of sequence ordering, assign each packet a **sequence number**. Send an **acknowledgment** (ACK) if packet received.
- Packets can be corrupted. Receiver can calculate **checksum** to verify validity, as well as run ECCs if protocol provides (Hamming codes, etc.) Some protocols send NACKs to indicate that packet has been rejected, but most just ignore packet.
- Idea: Instead of verifying message/ACK reception, have sender simply resend the packet if no ACK has been received after some time. If receiver receives duplicate packet (by sequence number), acknowledge but throw out. **How does this avoid two generals?**

Transport Layer: Basic Validation

- Most protocols: to avoid out of sequence ordering, assign each packet a **sequence number**. Send an **acknowledgment** (ACK) if packet received.
- Packets can be corrupted. Receiver can calculate **checksum** to verify validity, as well as run ECCs if protocol provides (Hamming codes, etc.) Some protocols send NACKs to indicate that packet has been rejected, but most just ignore packet.
- Idea: Instead of verifying message/ACK reception, have sender simply resend the packet if no ACK has been received after some time. If receiver receives duplicate packet (by sequence number), acknowledge but throw out. **How does this avoid two generals?** Receiver doesn't know (or care) *which* ACKs have been received, so no distributed consensus.

Transport Layer: Pipelined Protocols

- Validation + one-packet-at-a-time approach is correct, but is intolerably slow. **Solution:** allow *multiple* packets to be “in-flight” (forming a **context window**)

Transport Layer: Pipelined Protocols

- Validation + one-packet-at-a-time approach is correct, but is intolerably slow. **Solution:** allow *multiple* packets to be “in-flight” (forming a **context window**)
- **Go-Back-*N*** (GBN): Receiver keeps track of the first packet that has *not* been received (expected_seq_num).

procedure RECIEVE(k)
    if k = expected_seq_num **then**
        Send ACK(k); expected_seq_num $\leftarrow$ expected_seq_num $+1$
    else
        Send ACK(expected_seq_num - 1) ▷ *“cumulative ACK”*

  - On timeout: resend *all* packets in CW

Transport Layer: Pipelined Protocols

- Validation + one-packet-at-a-time approach is correct, but is intolerably slow. **Solution:** allow *multiple* packets to be “in-flight” (forming a **context window**)
- **Go-Back-*N*** (GBN): Receiver keeps track of the first packet that has *not* been received (expected_seq_num).

```
procedure RECIEVE(k)
    if k = expected_seq_num then
        Send ACK(k); expected_seq_num ← expected_seq_num +1
    else
        Send ACK(expected_seq_num - 1)                ▷ “cumulative ACK”
```

  - On timeout: resend *all* packets in CW
- **Selective ACK**: receiver individually acknowledges all correctly received packets (ACK(k)), buffers if needed for in-order delivery to application layer. Sender retransmits packets where no ACK received. If receiver receives a packet with the same sequence number as something in the buffer, throw out.

Transport Layer: Pipelined Protocols

- Validation + one-packet-at-a-time approach is correct, but is intolerably slow.
  Solution: allow *multiple* packets to be "in-flight" (forming a **context window**)
- **Go-Back-*N*** (GBN): Receiver keeps track of the first packet that has *not* been received (expected_seq_num).

  procedure RECIEVE(k)
      if k = expected_seq_num **then**
          Send ACK(k); expected_seq_num $\leftarrow$ expected_seq_num $+1$
      else
          Send ACK(expected_seq_num - 1) ▷ *"cumulative ACK"*

  - On timeout: resend *all* packets in CW
- **Selective ACK**: receiver individually acknowledges all correctly received packets (ACK(k)), buffers if needed for in-order delivery to application layer. Sender retransmits packets where no ACK received. If receiver receives a packet with the same sequence number as something in the buffer, throw out.
- TCP takes a **hybrid approach**, reports cumulative ACKs (lowest seq # not recieved - 1), but will accept out-of-order packets and reorder them.

Transport Layer: Pipelined Protocols

- Validation + one-packet-at-a-time approach is correct, but is intolerably slow. **Solution:** allow *multiple* packets to be "in-flight" (forming a **context window**)
- **Go-Back-N** (GBN): Receiver keeps track of the first packet that has *not* been received (expected_seq_num).

procedure RECIEVE(k)
    if k = expected_seq_num **then**
        Send ACK(k); expected_seq_num $\leftarrow$ expected_seq_num $+1$
    else
        Send ACK(expected_seq_num - 1)   ▷ *"cumulative ACK"*

  - On timeout: resend *all* packets in CW
- **Selective ACK**: receiver individually acknowledges all correctly received packets (ACK(k)), buffers if needed for in-order delivery to application layer. Sender retransmits packets where no ACK received. If receiver receives a packet with the same sequence number as something in the buffer, throw out.
- TCP takes a **hybrid approach**, reports cumulative ACKs (lowest seq # not recieved - 1), but will accept out-of-order packets and reorder them.
  - Sender considers multiple ACK(i)s as **dupACK**s, fresh i in ACK(i) **newACK**. Useful for estimating congestion.

Transport Layer: TCP Congestion Control

- **Main Problem**: How large do we make the context window?
  - If too large, then router queues fill up, dropped packets, no fair sharing.
  - If too small, then suboptimal performance.

Transport Layer: TCP Congestion Control

- **Main Problem**: How large do we make the context window?
  - If too large, then router queues fill up, dropped packets, no fair sharing.
  - If too small, then suboptimal performance.
- Initially, (after SYN/SYN-ACK/ACK), transmitter transmits small burst of packets, waits for ACK. Exponentially increases burst size (**slow start**).

Transport Layer: TCP Congestion Control

- **Main Problem**: How large do we make the context window?
  - If too large, then router queues fill up, dropped packets, no fair sharing.
  - If too small, then suboptimal performance.
- Initially, (after SYN/SYN-ACK/ACK), transmitter transmits small burst of packets, waits for ACK. Exponentially increases burst size (**slow start**).
- After some point (**SST**/**LIT**), switch from doubling the window size every burst to increasing it by 1 per burst.

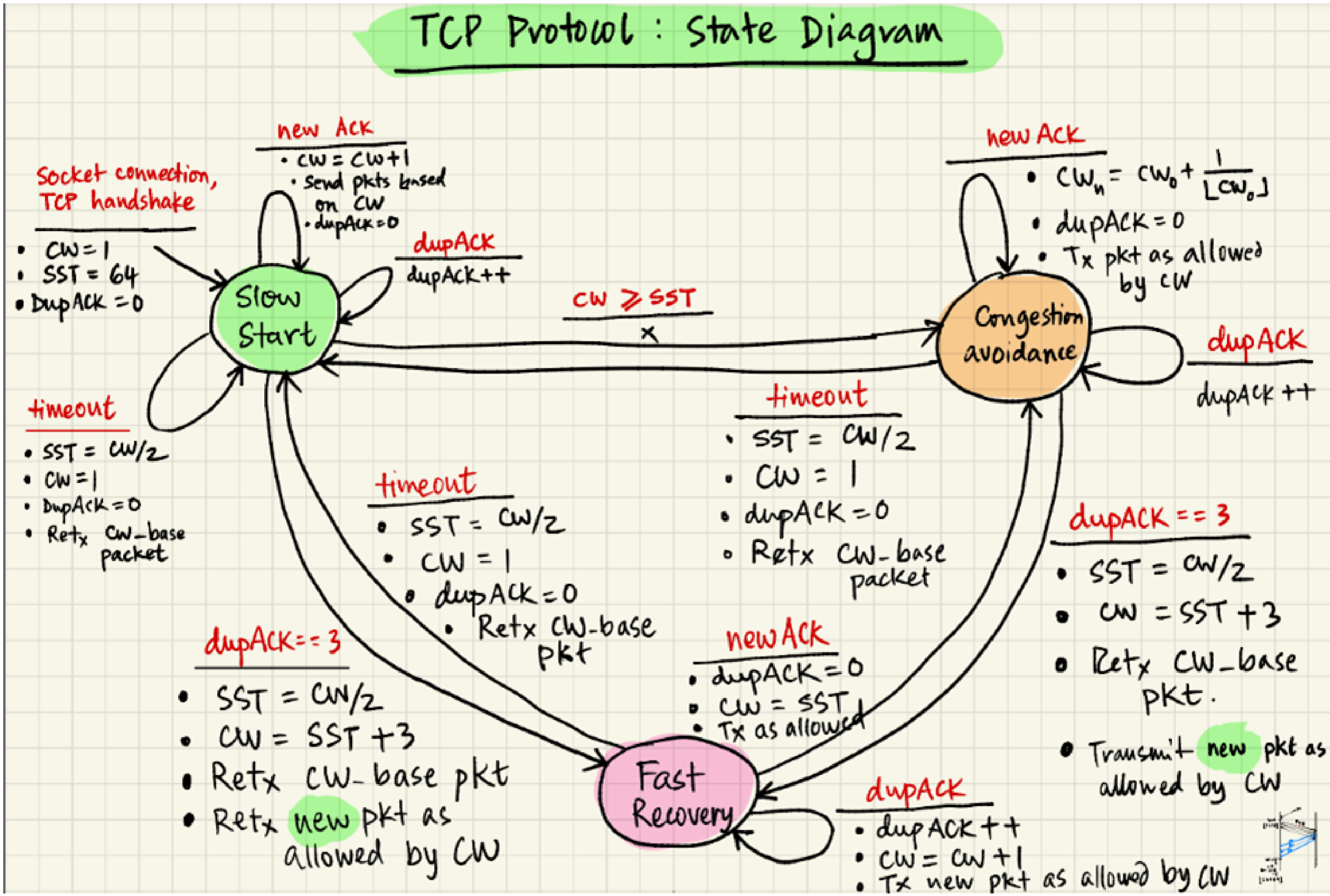
Transport Layer: TCP Congestion Control

- **Main Problem**: How large do we make the context window?
  - If too large, then router queues fill up, dropped packets, no fair sharing.
  - If too small, then suboptimal performance.
- Initially, (after SYN/SYN-ACK/ACK), transmitter transmits small burst of packets, waits for ACK. Exponentially increases burst size (**slow start**).
- After some point (**SST**/**LIT**), switch from doubling the window size every burst to increasing it by 1 per burst.
- Sender keeps a timer to interrupt for timeout. When CW shifted, increase the timer by the gap between packets. On timeout, drastically decrease CW, SST, resend packets.

Transport Layer: TCP Congestion Control

- **Main Problem**: How large do we make the context window?
  - If too large, then router queues fill up, dropped packets, no fair sharing.
  - If too small, then suboptimal performance.
- Initially, (after SYN/SYN-ACK/ACK), transmitter transmits small burst of packets, waits for ACK. Exponentially increases burst size (**slow start**).
- After some point (**SST**/**LIT**), switch from doubling the window size every burst to increasing it by 1 per burst.
- Sender keeps a timer to interrupt for timeout. When CW shifted, increase the timer by the gap between packets. On timeout, drastically decrease CW, SST, resend packets.
- DupACKs aren't necessarily a bad sign, but might be indicator of missed packets. If 3 dupACKs in a row, retransmit DupACK packet but don't reset SST, slightly cut CW (**fast recovery**).

Transport Layer: TCP State Machine



Transport Layer: TCP RTO Estimation

- How do we estimate how long timeout (RTO) should be?

Transport Layer: TCP RTO Estimation

- How do we estimate how long timeout (RTO) should be?
  - If too short, then premature timeout
  - If too long, then slow reactions to packet loss

Transport Layer: TCP RTO Estimation

- How do we estimate how long timeout (RTO) should be?
  - If too short, then premature timeout
  - If too long, then slow reactions to packet loss
- **Intuition**: should be AvgRTT + some “guard factor”.

Transport Layer: TCP RTO Estimation

- How do we estimate how long timeout (RTO) should be?
  - If too short, then premature timeout
  - If too long, then slow reactions to packet loss
- **Intuition**: should be AvgRTT + some "guard factor".
- AvgRTT estimated by rolling average: $RTT_{avg} \leftarrow (1-\alpha)RTT_{avg} + \alpha\, RTT_{packet}$

Transport Layer: TCP RTO Estimation

- How do we estimate how long timeout (RTO) should be?
  - If too short, then premature timeout
  - If too long, then slow reactions to packet loss
- **Intuition**: should be AvgRTT + some "guard factor".
- AvgRTT estimated by rolling average: $RTT_{avg} \leftarrow (1-\alpha)RTT_{avg} + \alpha\, RTT_{packet}$
- "guard factor" can be a deviation estimate:

$$
\begin{aligned}
devRTT_{avg} &\leftarrow (1-\beta)\, devRTT_{avg} + \beta(|RTT_{packet} - RTT_{avg}|) \\
RTO &\leftarrow RTT_{avg} + 4\, devRTT_{avg}
\end{aligned}
$$

Transport Layer: Flow Control and Fairness

- **Problem**: Receiver has a limited buffer. If many nodes transmitting to same receiver, losses may happen at receiver

Transport Layer: Flow Control and Fairness

- **Problem**: Receiver has a limited buffer. If many nodes transmitting to same receiver, losses may happen at receiver
- **Solution**: Receiver reports how much space left to sender in ACKs. Sender will deliberately use a smaller congestion window (while calculating CW as normal).

Transport Layer: Flow Control and Fairness

- **Problem**: Receiver has a limited buffer. If many nodes transmitting to same receiver, losses may happen at receiver
- **Solution**: Receiver reports how much space left to sender in ACKs. Sender will deliberately use a smaller congestion window (while calculating CW as normal).

- TCP guarantees **max-min fairness** (in stable state): All flows requesting less than fair share get their request. Remaining flows divide equally.

Feedback

http://go.acm.illinois.edu/cs438_mt1_feedback