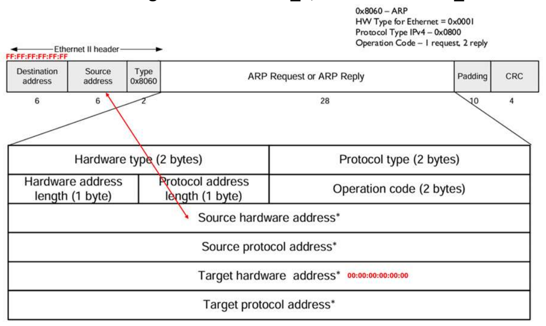
Subnets
- IPv4 format: 32-bits X.X.X.X/Y, Y = subnet mask
- Interfaces within the same subnet will have the same subnet part of IP address
- Subnets purpose: Packets don’t need to go through unnecessary routers, minimizing network traffic
- **Special IP addresses**
  - **First** IP address of network: Network Identification
  - **Last** IP address of network: Broadcast address
  - 0.0.0.0: Current host (e.g. initial IP address in DHCP protocol)
  - 255.255.255.255: Can’t be used
  - **First** and **last** subnets in a network cannot be used due to ambiguity (*In reality, subnet mask removes the ambiguity):
    - First IP address: network-id of subnet 0 or entire network?
    - Last IP address: broadcast to subnet or entire network?
  - Network part all zeros: Host on this network
- Classful addresses
  - Class A: **0**XXXXXXX.**<u>X.X.X/8</u>** (0.0.0.0 – 127.255.255.255)
  - Class B: **10**XXXXXX.X.**<u>X.X/16</u>** (128.0.0.0 – 191.255.255.255)
  - Class C: **110**XXXXX.X.X.**<u>X/24</u>** (192.0.0.0 – 233.255.255.255)
  - Class D: **1110**XXXX.X.X.X (only used for routing protocols)
  - Class E: **1111**XXXX.X.X.X (not used)
- Classless addressing (used today): Variable length subnet mask (VLSM)
- **Supernetting**: Combine multiple subnets to form a larger network
  - Before supernetting, if there are 2^k subnets, the router needs to have 2^k entries in its routing table
  - Route aggregation: Move k bits from host id to network id (bitwise shift left network mask by k bits)

Network Address Translation
- Enables private IP networks to communicate with hosts with public IP addresses outside its network
- One NAT table per router (not per interface)

1. Host A (IP_A, port) sends datagram to NAT router
2. NAT router determines output interface (e.g. X) using the destination address, then modifies (IP_A, port) → (IP_NAT_X, unique port) and sends the datagram
3. NAT saves (IP_NAT_X, unique port) → (IP_A, port) mapping in its NAT translation table
4. Server C responds back with src = IP_C, dest = (IP_NAT_X, unique port)
5. NAT translates (IP_NAT_X, unique port) → (IP_A, port) and sends datagram with src = IP_C, dest = (IP_A, port) back to Host A

ARP Protocol
- For hop-to-hop delivery; only relevant within LAN
- To determine MAC address of destination host, given its IP address

Same Subnet (Host A → Host B)
1. Host A’s ARP table contains IP_B?
   - Yes: Send ethernet frame <destMAC = MAC_B>. **DONE**
   - No: Broadcast **ARP REQUEST** packet <dest MAC = broadcast>
2. Switch receives a frame from Host A
   - Switch records <MAC_A, Interface_X, TTL> in its switching table
   - Frame = ARP REQUEST?
     - Broadcast to all interfaces
   - Frame ≠ ARP REQUEST?
     - Switch table contains MAC_B?
       - Yes: Forward frame to table[MAC_B].Interface. **DONE**
       - No: Broadcast frame to all interfaces. **DONE**
3. Host B receives the frame and sends **ARP REPLY** to Host A
   - Host B records <IP_A, MAC_A> in its ARP cache
4. Switch receives frame from Host B
   - Switch records <MAC_B, Interface_Y, TTL> in its switching table
5. Switch forwards frame to Interface_X (Host A)
6. Host A records of <IP_B, MAC_B > in its ARP cache.
7. Host A sends datagram <destIP = IP_B, destMAC = MAC_B> to Host B



- If ARP REQUEST is made for non-existing host, it will retry with increasing time intervals in-between until it eventually gives up
- A host will periodically send ARP REQUESTS to all addresses listed in its ARP cache to refresh its own ARP cache content
- **Gratuitous ARP requests**: Send ARP request to its own IP address
  - To check if it’s IP address has been taken by another host
  - To update the ARP caches in all the hosts within the network

Proxy ARP
- Allow communication between hosts (e.g. Host A → Host B) from different subnets
- In the ARP protocol, the **router will send the ARP REPLY instead** of the destination host, indicating its own (the router) MAC address
- When Host A wants to send a packet to Host B, it sends <destIP = IP_B, destMac = router MAC> to the router which will forward it to Host B

ARP Vulnerabilities
1. Lack of authentication: ARP packets can be forged
2. ARP is stateless: ARP REPLY can be sent without corresponding ARP REQUEST
3. Any host receiving a ARP REQUEST/REPLY must update its ARP cache if it already has an entry for the IP address in the cache

- These can lead to **ARP poisoning**: Sending fake ARP packets to change the IP to MAC mapping, linking the attacker’s MAC address with an IP address in the network

DHCP Protocol
- Application layer protocol
- Dynamically obtain IP address when host joins a network
- Also returns address of first-hop router, name and IP address of DNS server, and network mask
- Uses **UDP**, port **68** for client, port **67** for server
- ID identifies the client

Client REBOOT/INIT Stages (DORA): [ ] = UDP header, < > = DHCP payload
1. **DHCPDISCOVER**: client A broadcasts
   [src=0.0.0.0, dest=255.255.255.255]
   <CIADDR=0, YIADDR=0, SIADDR=0, CHADDR=CLIENT_MAC>
2. **DHCPOFFER**: DHCP server(s) respond with
   [src=SERVER_IP, dest= 255.255.255.255]
   <CIADDR=0, YIADDR=YIADDR_X, SIADDR=SERVER_IP, CHADDR=CLIENT_MAC>
   - Each DHCP server will ping the entire LAN with yiaddr_X to check if yiaddr_X is taken and sends DHCPOFFER if ping has no reply
   - Broadcast?: Some clients cannot accept unicast until configured to do so
3. **DHCPREQUEST**: client picks any one of the DHCPOFFER and broadcasts
   [src=0.0.0.0, dest= 255.255.255.255]
   <CIADDR=0, YIADDR= YIADDR_X, SIADDR=0, CHADDR=CLIENT_MAC>
   - Broadcast?: to inform all other DHCP servers to withdraw any offers and return their offered IP addresses to the IP address pool
4. **DHCPACK**: server_X responds and sends the **lease duration**
   [src= SERVER_IP, dest= 255.255.255.255]
   <CIADDR=0, YIADDR= YIADDR_X, SIADDR= SERVER_IP, CHADDR=CLIENT_MAC>
5. Client sends **3 ARP probes** to check if YIADDR is taken
6. If no reply, client sends **1 ARP announcement** to officially claim YIADDR

DHCP RENEWAL
- When lease time is 50% expired

1. **DHCPREQUEST**: Unicast, server name in DHCP packet must be filled
   [src=CLIENT_IP, dest= SERVER_IP]
   <CIADDR= CLIENT_IP, YIADDR=0, SIADDR= SERVER_IP, CHADDR=CLIENT_MAC>
2. **DHCPACK**: server responds with a unicast and updates the new lease time
   [src=SERVER_IP, dest= CLIENT_IP]
   <CIADDR= CLIENT_IP, YIADDR= CLIENT_IP, SIADDR= SERVER_IP, CHADDR=CLIENT_MAC>

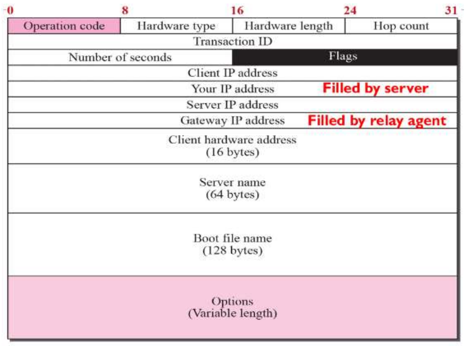
DHCP REBIND
- When lease time is 87.5% expired (in the case where DHCP renewal fails), client broadcasts DHCPREQUEST
- If client receives **DHCPNACK**, it will need to obtain new IP address

DHCP RELEASE
1. **DHCPRELEASE**: client sends
   <`
2. Server adds CLIENT_IP back to the available IP address pool

DHCP Relay Agent
- A client can communicate with a DHCP server from another subnet via a DHCP relay agent (a router listening on port 67)
- Relay agent intercepts DHCPDISCOVER

- Sets GIADDR = IP address of interface which receives the client's broadcast and increment hop-count by 1
- Forwards the DHCPDISCOVER (**<u>UNICAST</u>**) to one or more DHCP servers
- DHCP server(s) use GIADDR to determine client's subnet and offers a corresponding IP address by sending DHCPOFFER to the router (unicast)
- Router forwards the DHCPOFFER to the client (**<u>UNICAST</u>**)

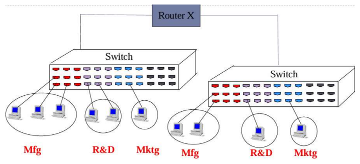
DHCP Packet Format

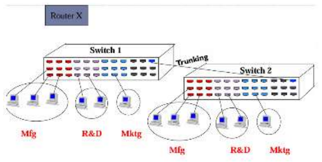


- **Server Name**: If SIADDR or server name are non-zero, only the server with matching SIADDR or server name will answer the DHCP request
- **Flags:** Only first bit is used; 0 = unicast, 1 = broadcast
  - Used to ask the server to broadcast if client cannot accept unicast
- **Options**: For additional info to client/vendor specific info
  - Tag | Length of value (in bytes) | Value
  - Tag = 0: For padding; Tag = 255: Indicates end of an option
  - Tag = 51: Lease time

<u>Virtual LAN (VLAN)</u>

- Motivation: LAN requires all hosts to be in the same physical location. VLAN allows hosts in different geographical locations to be grouped together (logical grouping)
- **Trunking**: Trunking is connecting switches with one another (via **trunk ports**), allowing them to bridge traffic within the same VLAN
  - Without trunking, communication between two hosts in the same VLAN but different switches need to be routed via the router. With trunking, it only goes through the switch, which is faster (since it is layer 2)
  - Switches **<u>do not</u>** bridge traffic across different VLANS; communication between hosts in different VLANs still need to be routed via the router, with or without trunking





Communication across VLANs

- How the router knows which port and which switch to route the data to?
- **Frame Filtering**
  - Examines information about each frame (MAC address or layer 3 protocol type)
- **Frame Tagging**
  - Switches place a unique ID in VLAN tag of the header of each frame it receives
  1. First (ingress) switch adds tag containing the VLAN ID to all incoming frames
  2. Intermediate switches do not recompute the VLAN id
  3. Last (egress) switch removes the tags from all outgoing frames
  - Frame tag format: 4 bytes
    1. Tag Protocol Identifier (TPID): 2 bytes; 0x8100 = IEEE 802.IQ
    2. Priority code point (PCP): 3-bit frame priority
    3. Drop eligible indicator (DEI): 1 = eligible, 0 = non-eligible
    4. VID: VLAN ID: 12 bits

Static VLAN

- Port-based; each port is assigned a VLAN number which is static (unless re-configured)
- Each switch maintains a bridging table (VLAN table) to track which ports are for which VLANs: (VLAN id, Switch Interface/Port)
- Configurable with VLAN Management System (VMS), or manually using CLI

Dynamic VLAN

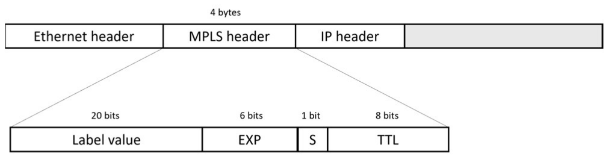
- VLANs dynamically assigned based on MAC address
  - When a host connects to a port on the switch, the switch sends a VLAN Query Protocol (VQP) packet to a database on the VLAN Membership Policy Server (VMPS) for VLAN membership
- The VMS will maintain a MAC-to-VLAN mapping table: (MAC address, VLAN id)

Layer-3 based VLAN

- VLANs dynamically assigned based on the layer-3 information in the packets being sent at the switch port (based on layer-3 <u>protocol type</u> field or <u>subnet</u> field)

<u>Multiprotocol Label Switching (MPLS)</u>

- Motivation: Traditional IP routing requires routers to inspect the packet's IP dest addresses and then use the routing table to determine the next hop, which is slow. MPLS streamlines this by using short labels instead of IP addresses to make forwarding faster
- Layer 2 protocol; adds a new MPLS header to ethernet frame



<u>HTTP Protocol</u>

- Uses TCP, port 80; stateless (can manage state with cookies)

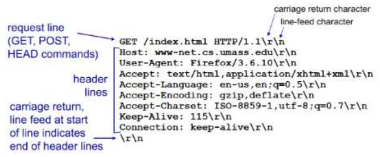
Non-Persistent HTTP

- Non-pipelined:
  1. Inititate TCP connection (1 RTT)
  2. GET HTML file (1 RTT + trans)
  3. Connection closes
  4. Repeat for each *n* additional ojects
- Pipelined (*m* files at once):
  1. Initiate TCP connection (1 RTT)
  2. GET HTML file (1 RTT + trans)
  3. Connection closes
  4. Repeat for $\lceil n/m \rceil$ times:
     - 4.1. Initiate TCP connection (1 RTT)
     - 4.2. GET *m* objects (1 RTT + *m* trans)
     - 4.3. Connection closes

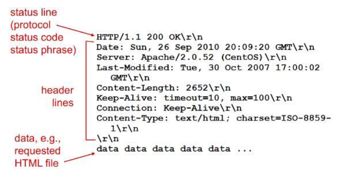
Persistent HTTP

- Non-pipelined:
  1. Inititate TCP connection (1 RTT)
  2. GET HTML file (1 RTT + trans)
  3. Repeat for *n* times:
     - 3.1. GET object (1 RTT + trans)
  4. Connection closes
- Pipelined (*m* files at once):
  1. Initiate TCP connection (1 RTT)
  2. GET HTML file (1 RTT + trans)
  3. Repeat for $\lceil n/m \rceil$ times:
     - 3.1. GET *m* objects (1 RTT + *m* trans)
  4. Connection closes

HTTP Packet Format



```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```



```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

Cookies

- HTTP is stateless; cookies are used to manage user identity and state

1. Client sends HTTP request to server for the first time
2. Server responds with `Set-Cookie: cookie-ID` in response header
3. Client accepts cookie?
   - 3.1. Yes: Client saves cookie in cookie file. Server creates an entry for that cookie in the backend database
4. [If accepted] Client sends next HTTP request, with `cookie: cookie-ID` in request header
5. [If accepted] Server stores and/or retrieves user-specific information from database using the cookie-ID and sends user-specific response

Web Proxy

1. Client sends request to proxy server
2. If response is not cached:
   2.1. Proxy server forwards the request to origin server
   2.2. Origin server sends response with `last-modified: date`
   2.3. Proxy server caches the response and forwards it back to client
3. If response is cached:
   3.1. Proxy server sends **Conditional GET** request to origin server with `if-modified-since: <date>` in header
      - `date`: `last-modified` date in the cached response

   3.2. If the object is not modified before `date`, origin server sends **304 Not Modified** and proxy responds to client with the cached response
   3.3. If the object is modified after `date`, origin server sends **200 OK**. Proxy server caches the response and forwards it back to client

HTTP Versions

- HTTP 1.1: Multiple, pipelined GETS over single TCP connection
  - Issues: server responds in-order to GET requests; small objects need to wait for transmission behind large objects (head-of-line (HOL) blocking)
- HTTP 2: Mitigates HOL blocking
  - Divide objects into frames and transmits objects based on client-specified object priority (instead of FCFS)
  - Can push unrequested objects to client
  - Issues: Using a single TCP connection means that when there is a packet loss, objects need to wait for recovery before they can be transmitted
- HTTP3: Uses UDP to decrease delay
  - Uses Quick UDP Internet Connections (QUIC)

Email Protocols

- **SMTP (Simple Mail Transfer Protocol)**
  - Port 25
  - Run in Message Transfer Agents (MTA) and used to push mail
- **POP3 (Post Office Protocol v3)** or **IMAP4 (Internet Mail Access Protocol v4)**
  - POP3 on port 110; IMAP4 on port 143
  - Run in Message Access Agents (MAA) and used to pull mail

1. MTA client in sender's User Agent (i.e. mail readers e.g. outlook) pushes mail via SMTP to sender's MTA server
2. Sender's MTA server saves the mail in a backend database. MTA client in the sender's MTA server forwards the mail to the receiver's MTA server via SMTP
3. Receiver's MTA server saves the mail in its storage system
4. Receiver's User Agent uses its MAA client which uses POP3/IMAP4 to pull the mail from the receiver's MAA server

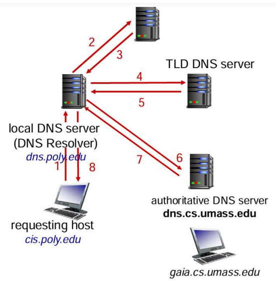
- Web-based emails use HTTP to push and pull email instead of SMTP and POP3/IMAP4

Domain Name System

- DNS name space: Set of all domains registered in the DNS, organised into a hierarchical, tree-like structure and stored and replicated in distributed databases
  - Hierarchy: **Root** > **TLD** (e.g. .com, .sg, .org) > **Authoritative** (amazon.com)
- Domain: Subtree in the DNS name space
- DNS server uses port 53
- DNS runs over TCP if response message is > 512 bytes, otherwise it uses UDP
- **Top-level Domain Servers (TLD)**
  - Responsible for .com, .org, .edu, etc and all top-level country domains e.g. .sg, .uk, etc.
- **Authoritative Servers**
  - Organisation's own DNS servers
- **Local DNS server (DNS resolver)**
  - Does not belong to the DNS hierarchy
  - Each ISP has one, also called the "default name server" (get its IP address through DHCP)
  - Acts as a proxy to forward DNS queries to the hierarchy and caches the name-to-address mappings

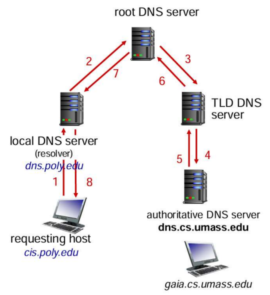
DNS name resolution

Iterative



1. Client first queries the local DNS server
2. If the local DNS server is unable to resolve, it will forward it to the root server
3. Root responds with the IP address of the TLD server to the local DNS server
4. Local DNS server queries the TLD server
5. TLD responds with the IP address of the authoritative server
6. Local DNS server queries the authoritative server
7. Authoritative server reponds with the IP address of the FQDN
8. Local DNS server forwards IP address to client

Recursive



1. Client first queries the local DNS server
2. If the local DNS server is unable to resolve, it will forward it to the root server
3. Root forwards it to TLD
4. TLD forwards it to authoritative
5. Authoritative returns the IP address of the FQDN to the TLD
6. TLD forwards the IP address to root
7. Root forwards the IP address to the local DNS server
8. Local DNS server forwards IP address to client

- Not ideal; heavy load at upper levels of hierarchy

DNS Zones

- DNS zone: A specific portion (path) in the DNS hierarchy which an organisation has authority over
- The DNS server for a domain can either (i) store the IP address for every node in the domain, or (ii) divide the domain into sub-domains and delegate part of its authority to other servers
- **Primary** DNS server: Main authoritative server and holds the master copy of the DNS zone file
- **Secondary** DNS server: Also authoritative for the zone they serve; keeps a copy of the DNS zone file (via zone transfer; the file is only read-only here); provides redundancy

DNS resource records

- Resource records (RR) are data entries in the DNS database, in the format of (name, value, type, ttl)
- **A**
  - name: hostname; value: IPv4 address(es)
  - `example.com. 3600 IN` **`A`** `93.184.216.34`
    Maps `example.com.` domain to `93.184.216.34`
- **AAAA**
  - name: hostname; value: Ipv6 address(es)
  - `example.com. 3600 IN` **`A`** `2606:2800:220:1:248:1893:25c8:1946`
    Maps `example.com` domain to `2606:2800:220:1:248:1893:25c8:1946`
- **NS**
  - name: domain; value: hostname(s) of an authoritative nameserver(s) for this domain (primary & secondary)
  - `example.com. 3600 IN` **`NS`** `ns1.example.com`
    Maps `example.com` domain to `93.184.216.34`
  - If `dig foo.example.com NS` returns a SOA record, it means that foo.example.com is a subdomain, so it returns the authoritative nameserver
- **SOA**
  - name: domain; value: hostname of the **primary** authoritative nameserver for this domain
  - `example.com. 3600 IN` **`SOA`** `ns1.example.com. admin.example.com. 2024091001 7200 3600 1209600 3600`
    `ns1.example.com` is the primary authoritative nameserver for the `example.com` domain, and `admin@example.com` is the email address of the admin
- **CNAME**
  - name: alias; value: canonical ("real") name
  - `www.nus.edu.sg. 3600 IN` **`CNAME`** `mgnzsqc.x.incapdns.net.`
    `www.nus.edu.sg.` is an alias for `mgnzsqc.x.incapdns.net.`
- **MX**
  - name: domain; value: hostname(s) of the mailserver(s)
  - `nus.edu.sg. 3600 IN` **`MX`** `10 nus.in.tmes.trendmicro.com.`
    `nus.in.tmes.trendmicro.com.` is a mailserver for the domain `nus.edu.sg`, and it has a priority of 10 (lower number, higher priority)
- **PTR**
  - Used for reverse DNS lookups (given IP address, find the domain name)
  - `4.4.8.8.in-addr.arpa. 3600 IN` **`PTR`** `dns.google.`
    The domain name for IP address `8.8.4.4` is `dns.google.`

CSMA/CD

- Link layer protocol (communication acoss one-hop)
- Used by wired Ethernet
- Hosts sense the communication channel before transmission
  - Channel is idle: Transmit
  - Channel is busy: Wait until idle
- Collision detection:
  - A transmitting host continues to sense the channel while it is transmitting.
  - If it detects that the signal on the channel is not what it is transmitting, it detects a collision
  - When collision is detected, transmission is **aborted** and tried again with **exponential backoff**
  - After 16 failed attempts, the frame is discarded
- Min. transmission time: Hosts must have a min. transmission time equals to 2 × propagation delay in order for collision to be detected

CSMA/CA

- Used by wiressless LANs (**WLANs**)
  - CSMA/CD is avoided because CD is hard in wireless networks.
    - A wireless transmitter/receiver cannot sense the channel while simultaneously transmitting/receiving
    - Signal attenuation: signals weaken as they travel through air/obstacles (vs stay relatively constant in controlled channels like physical wires)

1. Hosts sense the communication channel before transmission
   1.1. Channel is idle: Wait **Inter-Frame Space (IFS) time** to ensure that the channel is idle across the entire channel (something like propagation delay")
   1.2. Channel is busy: Wait until idle
2. If channel is still idle, choose a random number R from 0 to $2^k - 1$ and wait for R time. This is to minimise collision in the case where two hosts both sense the channel at the same time (collision can still happen if 2 hosts pick the same R, but the chances are miniscule)
   2.1. Channel is idle: Send frame
   2.2. Channel is busy: Wait until idle
3. Wait for ACK
   3.1. ACK received: Done

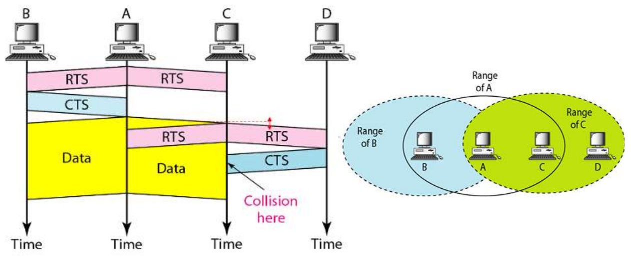
- ACK is not received: If failed 16 times, abort. Otherwise, restart with exponential backoff

Hidden node problem

- Two nodes (e.g. B and C) are out of each other's range but are within range of a common receiver (e.g. A), leading to collisions
  - Nodes B and C sense that the channel is idle (because they are too far from each other) so they transmit to Node A. At node A, collision happens
- **RTS/CTS (solves hidden node problem)**
  - Node B sends Request-to-Send (RTS) along with Network Allocation Vector (NAV) to indicate that it wishes to reserve the channel for NAV time
  - Node A replies with Clear-to-Send (CTS). Node B transmits
  - Node C hears CTS and sets NAV timer, during which it will refrain from transmitting

Exposed node problem (no solution)

- Assume we have new Node D which is only in Node C's range
- Node A sends RTS (sensed by nodes B and C, but not D)
- Node B sends CTS and Node A transmits to the channel
- Node C wants to (and can, because Node D is out of range of Nodes A and B) transmit to Node D, but it does not because it sensed an RTS in the channel, leading to underutilisation
  - Even if Node C starts to transmit (sends RTS and receives CTS), there will be a collision at Node C (between the CTS from Node D and data from Node A)



Interframe Space (IFS) values

- Distributed IFS (DIFS): Longest IFS
  - Used as the delay during access contention
- Short IFS (SIFS): Shortest IFS
  - Used for immediate response actions (e.g. ACK, CTS, Polling)
  - Shorter → priority is given for such response messages over sending new messages (which use DIFS)
- Point coordination function IFS (PIFS): Mid-length IFS
  - Used by centralised controller in PCF scheme when using polls

MAC Sublayer Modes (DCF, PCF)

Distributed Coordination Function (DCF) Contention Service

- Contention-based; as in CSMA/CA

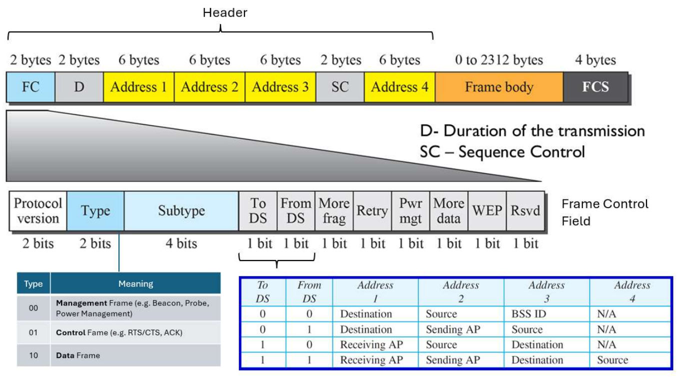
Point Coordination Function (PCF) Polling Service

- Used for time sensitive transmissions
- Centralised, contention-free polling access method
- Point Coordinator (PC) at the access point (AP) performs the polling
  - Nodes request that AP register them on polling list
- Nodes registered for polling will refrain from contention; AP will be contenting on their behalf with other unregistered nodes operating in DCF mode
- To give priority to AP, AP uses PIFS which is shorter than DIFS
- Once AP has control over the channel, it enters PCF mode, polling nodes in the polling list to deliver traffic to the polled node
- Always using PIFS can starve other nodes in DCF. So, AP alternates between using PIFS and DIFS to give other nodes a chance

IEEE 802.11

- **Basic Service Set (BSS)**: Set of stations controlled by a single AP
  - **Ad-hoc network/Independent BSS**: BSS without AP. Stations can communicate with one another directly
    - BSSID = 48-bit random number (MAC address)
  - **Infrastructure BSS**: BSS with AP. All traffic between stations must pass through the AP
    - BSSID = MAC address of radio in AP
  - **BSSID** (MAC address format) =
- **Extended Service Set (ESS)**: Set of ≥ 1 BSS interconnected by a Distribution System (DS)
  - Traffic between BSS must flow through AP

IEEE 802.11 Frame Format

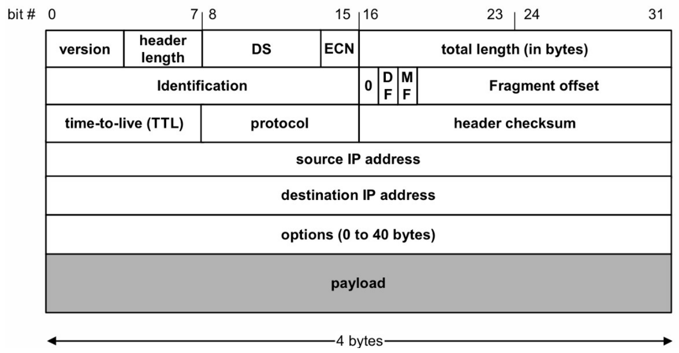


Type	Meaning
00	**Management** Frame (e.g. Beacon, Probe, Power Management)
01	**Control** Fame (e.g. RTS/CTS, ACK)
10	**Data** Frame

To DS	*From DS*	*Address 1*	*Address 2*	*Address 3*	*Address 4*
0	0	Destination	Source	BSS ID	N/A
0	1	Destination	Sending AP	Source	N/A
1	0	Receiving AP	Source	Destination	N/A
1	1	Receiving AP	Sending AP	Destination	Source

IP Packet Format & IP Fragmentation

IP Datagram Format

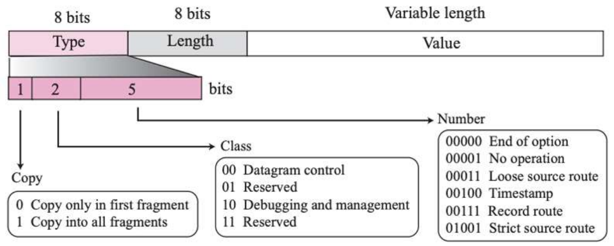


- Min header size = 20 bytes; Max header size = 20 + 40 ≈ 64 bytes
- Min total length = 20 bytes; Max total length = 2^{16} = 65536 bytes
- Datagram transmitted in network byte order (**big endian**)
- **DS/ECN**: Differentiated Services (8-bits)
  - Used to be called "Type-of-Service", but has since been redefined
  - **Differentiated Service (DS)** (6-bits): Contains Differentiated Services Code Point (**DSCP**) code which controls the priority and quality of service of the packet as it travels across the network
  - **Explicit Congestion Notification (ECN)** (2-bits): Feedback mechanism used by TCP
    - 00: Non ECN-Capable Transport (Non-ECT)
    - 01/10: ECN Capable Transport (ECT(0) & ECT(1))
    - 11: Congestion Encountered (CE)

- **HLEN**: Header length (4-bits)
  - Length of IP header, in **4-byte** increments (i.e. range: 0 – 63 byes)
- **DF**: Don't Fragment flag (1-bit)
  - Set to 1 to indicate not to fragment an IP datagram
  - If DF = 1 and datagram size > MTU, it will be dropped
- **MF**: More Fragments flag (1-bit)
  - MF = 0: This is the last fragment
  - MF = 1: There are more fragments to follow
- **Fragment offset**: (5-bits)
  - Offset of fragment's data relative to beginning of the original IP datagram's data
  - Because it is 5-bits, data length of each fragment must be divisible by 8 (implicit 3-bit zeros are added)
  - Actual fragment offset (bytes) = fragment offset * 8
- **Total Length**: (16-bits)
  - Total length of the IP datagram (i.e. header + data)
- **Identification**: (16-bits)
  - Used in IP fragmentation to identify the original IP datagram a fragment belongs to

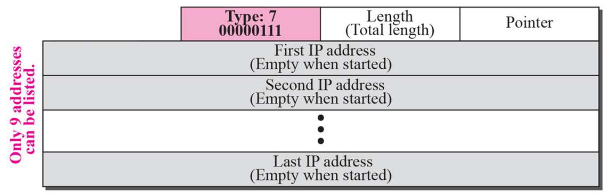
IP Fragmentation
- Maximum Transmission Unit (**MTU**): Size of largest packet (for IP, **includes the IP header**) that can be transmitted. Exists to prevent any node from hogging a transmission channel
- IP fragmentation happens at a router or original sending host
- Fragments reassembled at destination host
- Max Fragment size (including IP) = MTU
  - Data length of each fragment (except last fragment) **must be divisible by 8** (e.g. if MTU = 256, max data length = 256 – 20 = 236, but revised to 232 to make it divisible by 8)
- # Fragments = ⌈ IP datagram size - 20 / Max data length per fragment ⌉

IP Options



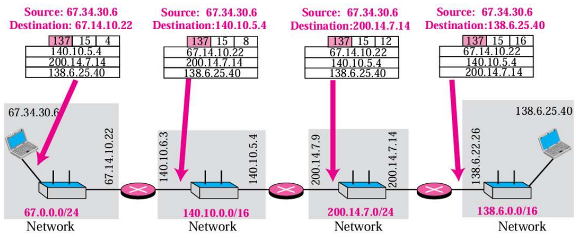
- Type:
  - Copy: Copy IP options to all fragements or only first fragment
  - Option Number: Represents the option (up to 32 options)
  - Length (optional): Length, in bytes, including Type and Length field itself

Record Route
- Record the route an IP packet takes
- Only up to 9 addresses (36 bytes can represent 9 addresses)
- Pointer (1-byte): Indicates where the next router's address will be recorded within the option (address offset from start of options)



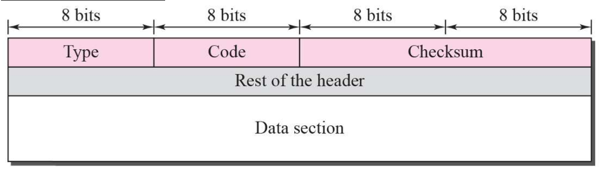
- Whenever the packet reaches a router, place the router's IP address in the options field inside the address pointed to by Pointer. Then, increment Pointer by 4
- Once Length exceeded, stop recording
  - E.g. if Length = 15, can record at most (15 – 3) / 4 = 3 addresses

Strict-source-route
- Options contain IP addresses of routers which form a path the IP packet must follow. Must strictly adhere to this specific route
- Whenever the packet reaches a router, increment Pointer by 4. The next hop must be the router with the IP address pointed to by Pointer (by swapping the IP address pointed to by Pointer with 'Desination address' in IP header)



Loose-source-route
- Same as strict-source-route, except routers can choose to deviate from the path as long as they eventually visit all of the listed routers

ICMP Protocol



- Allows routers and hosts to send error or control messages to other routes or hosts
- Can only report condition back to the original source (source that created and sent that IP packet), i.e. error messages are not propagated to other devices along the path

ICMP error messages

Type	Message
3	Destination unreachable
4	Source quench
11	Time exceeded
12	Parameter problem
5	Redirection

- Payload: Entire IP header + first 8 bytes of the IP packet (of packet causing error)
  - IP header: To identify the IP packet causing the error
  - First 8 bytes of IP packet: Contains source and dest port inside UDP/TCP header; to identify the application sending the IP packet
- A datagram carrying ICMP error messages will not trigger another ICMP error message if it has an error
- Only 1 ICMP error message for fragmented datagram
- No ICMP error messages for a datagram whose destination address is multicast/broadcast
- No ICMP error messages for a datagram whose source address is not a single host (e.g. 0.0.0.0, 127.x.x.x. broadcast or multicast address)
- **ICMP source quench message**:
  - Informs the source that a datagram has been discarded due to congestion
  - One message for each discarded datagram
  - Source most slow down the sending of datagrams
- **ICMP time-exceeded message**:
  - When router received datagram with TTL = 0, it discards it and sends this ICMP message
  - When destination host does not receive all of the dragments in a set time, it discards all the received fragments and sends this ICMP message
- **ICMP redirection message**:
  - Helps hosts to optimise their routing tables
  - E.g. Host A sends to Router R1 which forwards it to R2. If R1 forwards the packet in the same interface, it knows that Host A can directly send the packet to R2 instead. R1 sends ICMP redirection to Host A, which then updates its routing table

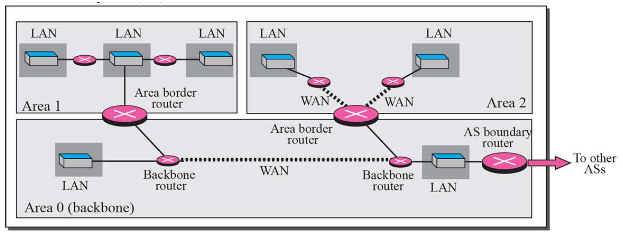
ICMP query messages

Type	Message
8 or 0	Echo request (8) or reply(0)
13 or 14	Timestamp request (13) or reply (14)

- **ICMP echo request/reply**
  - Ping; tests reachability of a host
- **ICMP timestamp request/reply**
  - Helps to synchronise two clocks in two machines
  - Original, Receive, Transmit timestamp

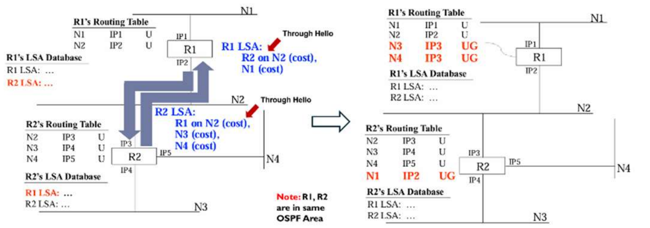
OSPF Protocol

- Used for **intradomain** routing (domain = autonomous system (AS))
- OSPF uses IP directly (no transport protocol)
- Uses **Link State Routing**, which uses Dijkstra's algorithm to locally and independently compute shortest path to each node
  - As such, routers need to know entire network topology
  - To do so, routers gather information about its neighbours, then share that info to all other routers in the same area through LSAs
- Routers learn about the entire network topology through
- AS is split into smaller Areas (Area 0 = backbone). Each router in an area has no knowledge of network topology outside its area
  - Link state exchange happens only between routers within the area; helps to limit the amount of link-state info exchanged
  - Relies on Area Border Routers (**ABR**) and AS Boundary Routers (**ASBR**) to summarise information outside of its area



Neighbour Discovery
- To discover neighbours of a router
- Router sends ***Hello*** messages through all its network interfaces:
  - Message is **multicasted** to all **OSPF routers** through AllSPFRouters address **224.0.0.5**, every 10 seconds
  - Failure to receive any Hellos from a neighbour for 40s: assume link failed or neighbour crashed
  - Message includes router's IP address for that interface, Hello interval, list of neighbours whose Hellos the sender has already heard
- Router 1 receives ***Hello*** message from Router 2
  - Router 1 now knows that Router 2 is its neighbour

Link State Advertisements (LSA)
- For routers to share information about their local links and state to other routers, through LSA packets
- LSAs are sent when either of these happen:
  - Router discovers a new neighbours
  - Link to a neighbour goes down
  - Cost of a link changes
  - Basic refresh every 30mins
- LSAs are distributed to all routers in the area by reliable flooding
  - Explicitly ACKed, sequenced and timestamped (like TCP)



Transient Links
- Link with multiple routers
- To prevent flooding with LSAs, each router has only two neighbours: **Designated Router (DR)** and **Backup Designated Router (BDR)**.
- Reach router multicasts their LSAs to DR using AllDRouters address **224.0.0.6**
- DR and BDR have all routers in the network as neighbours
- When new router joins the transient link:
  1. DR sends **Database Description Packets** (containing a summary of its LSA db) to new router
  2. New router respons with **Link State Request Packet** (list of LSAs that it does not have or are outdated)
  3. DR forwards the full LSAs in the list to new router via **Network Link Advertisement Packet**
  4. New router responds with **Link State ACK Packet**
  5. New router sends its LSA to DR and BDR
  6. DR and BDR multicast the LSA to other routers in the link

NAT Traversal
- Techniques to enable outside world to communicate with a device behind NAT

Port Forwarding (Public → Behind NAT)
- Manually create a translation on the NAT router
  - Map private IP and port to public IP and port; external devices use the public IP and port to communicate
- Need to create a DNS 'A' record for the public IP so that external devices can use a hostname instead of having to know the IP address

Connection Reversal (Public → Behind NAT)
- Ask the host (A) behind NAT to initiate the connection (to B) instead by sending a connection request from Host B to A
- If Hosts A and B have previously connected, just use the previously established connection to send the connection request
- Otherwise, Host B relays the connection request to a connection server, which forwards it to Host A
  - Hosts A and B must be connected to the connection server

Proxy Server (Behind NAT → Behind NAT)
- Hosts communicate with one another via a proxy server which forwards their messages from one party to the other
- Very reliable, but not very efficient (extra hop)

UDP Hole Punching (Behind NAT → Behind NAT)
- Hosts A and B connect to a rendezvous server, S
- When Hosts A and B try to connect with each other, they send messages to S. This creates a record in their respective NAT translation table mapping their private IP and port to public IP and port
- Server S knows the public IP and port of both Hosts A and B; S sends B's public IP and port to A and likewise, sends A's public IP and port to B
- Hosts A and B can communicate directly with each other's public IP and port

TCP Hole Punching (Behind NAT → Behind NAT)
- Same as UDP Hole Punching, except both Hosts A and B need to send SYN packets to each other at the same time (**TCP simultaneous OPEN**)

Network Security
- **Confidentiality**: Only sender and intended receivers can understand message
- **Authentication**: Sender and receiver need to confirm each other's identity
- **Integrity**: Messages are not altered without detection

Confidentiality
- **Symmetric** Key Cryptography:
  - Sender and receiver use the same encryption (secret) key K_s
  - Sender and receiver use K_s to encrypt and decrypt
  - E.g. Block cipher (DES, AES), Caesar's cipher
- **Public** Key Cryptography:
  - Sender uses public key of receiver to encrypt the message (**$K^+(m)$**)
  - Receiver uses own private key to decrypt the message (**$m = K^-(K^+(m))$**)
  - E.g. RSA encryption

Integrity
- Apply hash(e.g. md5) to produce a message digest/modification detection code, **H(m)**
  - Hash function must be one-way, collision resistant and pre-image resistant
- Sender: send message along with the message digest i.e. **$m \oplus H(m)$**
- Receiver: calculates digest H(m) and compares with the digest H(m) it receives

Authentication
- Motivation: A malicious user can generate its own m' and send **$m' \oplus H(m')$** to the receiver. Receiver will not know who this message is from
- Message Authentication Code (**MAC**): Hash message along with a symmetric key i.e. **H(m + s)**
  - Sender: send message along with the message digest i.e. **$m \oplus H(m + s)$**
  - Receiver: calculates H(m + s) and compares with the H(m + s) it receives
- **Digitial signature**: Verifiable by receiver and unforgeable
  - Sender: encrypts digest with its own private key, i.e. **$m \oplus K^-(H(m))$**
  - Receiver: decrypts received digest with sender's public key, generates its own digest and checks if both match, i.e. **$K^+K^-(H(m)) = H(m)$**

Non-repudiation
- Motivation: Sender can deny having sending a message to receiver by denying ownership of public keys
- Sender: encrypts the digest with its own private key. i.e. **$m \oplus K^-(H(m))$**
- Trusted third party: uses sender's public key to decrypt digest and verifies the digest. Then, it encrypts the digest with its own private key and sends it to receiver, e.g. **$m \oplus K_T^-(H(m))$**
- Receiver: uses trusted third party's public key K_T^+ to decrypt the message

Challenge-Response Authentication Protocol
- A type if authentication protocol where one party (server) presents a challenge to another party (client) which must respond with the correct answer to prove its identity
- Nonce: A number R used only once-in-a-lifetime as the challenge
  - Uses only once to prevent replay attacks
- Challenge-Response with **Symmetric** Key:
  - Client wishes to authenticate with the server. Server responds with R
  - Client sends R encrypted with secret key shared between client and server, i.e. **$K_{CS}(R)$**

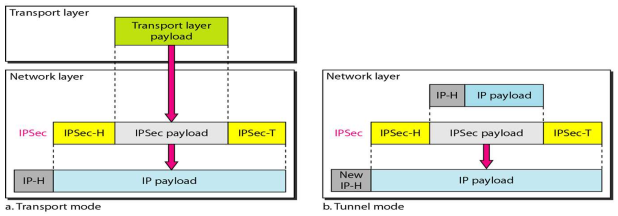
- Challenge-Response with **asymmetric** key:
  - Client wishes to authenticate with the server. Server responds with R
  - Client encrypts R with its own private key and sends it to the server i.e. $K_C^-(R)$
  - Server uses client's public key to decrypt R and verfies that R is the same one it has sent, i.e. $K_C^+(K_C^-(R)) = R$
  - Loophole: a MITM can intercept the response from server and send its own $K_M^-(R)$ to the server. When the server asks for client's public key, MITM can intercept again and send its own K_M^+

Key Distribution and Certification

- Key Distribution Centers: distribution of **symmetric** keys
  - Uses Kerberos; Easy to revoke
  - To establish symmetric key between two entities:
    - Entity A: sends $K_{A\text{-}KDC}(A, B)$ to KDC indicating that it wants to communicate with B
    - KDC: Responds with $K_{A\text{-}KDC}(R_1, K_{B\text{-}KDC}(A, R_1))$
    - Entity A: decrypts the response and sends $K_{B\text{-}KDC}(A, R_1)$ to B
    - Entity B: decrypts the response and now knows that A want to communicate with shared key R_1
- Diffie Hellman Algorithm:
  - To exchange symmetric keys **without** the use of KDCs
  - To establish symmetric key between two entities (values of g and p are public):
    - Entity A: sends $R_1 = g^x \bmod p$ to B
    - Entity B: sends $R_2 = g^y \bmod p$ to A
    - Entity A: computes $K = (R_2)^x \bmod p = g^{xy} \bmod p$
    - Entity B: computes $K = (R_1)^y \bmod p = g^{xy} \bmod p$
  - Ephemeral DH offers forward secrecy, unlike RSA
- Certification Authorities
  - Handles **asymmetric** key cryptography
  - CA signs an entity's public key to create a certificate, i.e. $K_{CA}^-(K_E^+)$
  - When entity A wants entity B's public key:
    - Entity A gets entity B's certificate $K_{CA}^-(K_B^+)$ from entity B
    - Entity A then decryts the certificate using CA's public key to get entity B's public key, i.e. $K_{CA}^+(K_{CA}^-(K_B^+)) = K_B^+$

Internet Security Practices

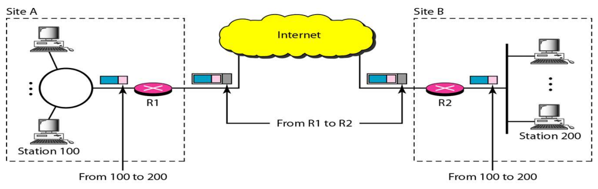
- **Transport mode**: Encrypt IP payload (i.e. transport layer packet) and add IPSec headers and trailer. IP header not encrypted
- **Tunnel mode**: Encrypt entire original IP packet (header + payload), add IPSec headers and trailer, then add new IP header in front of encrypted packet



a. Transport mode   b. Tunnel mode

Virtual Private Network (VPN)

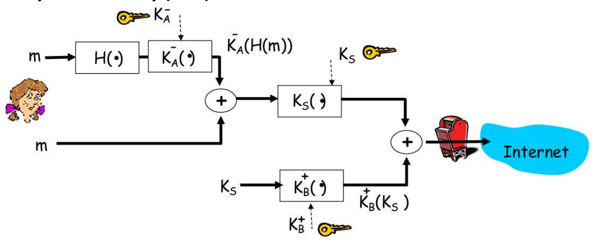
- Uses IPSec in tunnel mode to secure and encrypt connection between client's device and server over the public internet



Transport Layer Security (TLS)

- Transport layer security to any TCP-based app
- TLS handshake:
  1. **ClientHello**
     1.1. Establish TCP connection (TCP three-way handshake)
     1.2. Client sends TLS ClientHello to server
       - Includes supported cipher suites + hash algorithms
  2. **ServerHello**
     2.1. Server responds with ServerHello
       - Includes server's chosen cipher suite + hash algorithm and server's CA signed certificate
  3. **Key Derivation**
     3.1. Client receives and verifies server's certificate, decrypts it using CA's public key to get server's private key
     3.2. Client creates and sends pre-master secret to server. The secret is encrypted using server's public key
     3.3. Both client and server use the pre-master secret to generate a master secret
     3.4. Master secret is used to generate 4 keys:
       - K_{CS}: Encrypt messages from client to server
       - K_{SC}: Encrypt messages from server to client
       - M_C: MAC to authenticate client
       - M_S: MAC to authenticate server

Pretty Good Privacy (PGP)



TCP Congestion Control

- Goal: Efficiently use the available network bandwidth by transmitting at a rate close to congestion
- Congestion window (**cwnd**): Amount of data (in MSS) sender can transmit without receiving an ACK from receiver
  - Sender sends at a rate = min { cwnd, rwnd }

Loss-based Congestion Control

- **Capacity Probing**: Arrival of an ACK signals that one packet is removed from the network → can push a new packet into the network
- Lost packets indicate network congestion
- **Slow Start**: To quickly increase cwnd from a cold start to maximise use of bandwidth
  - Doubles the cwnd for each RTT (cwnd++ for each ACK received), until **ssthresh** is reached
- **Additive Increase**: To transmit at near congestion levels
  - Increment cwnd by 1 for each RTT (cwnd++ for one full window's worth of ACK received)
- **AIMD** (Additive Increase Multiplicative Decrease) Algorithm: TCP begins with Slow Start, with initial ssthresh set to 64KB/1GB, until ssthresh is reached, in which Additive Increase begins
- When congestion is detected, restart from Slow Start
- **TCP Tahoe**:
  - Uses AIMD
  - When congestion is detected via timeout or 3 duplicate ACKs:
    1. ssthresh is set to max { 2, ½ min { cwnd, rwnd })
    2. cwnd is set to **1**
    3. Fast retransmit if duplicate ACKs
- **TCP Reno**:
  - Uses AIMD + **Fast Recovery**
  - When congestion is detected via timeout:
    1. ssthresh is set to max { 2, ½ min { cwnd, rwnd })
    2. cwnd is set to **1**
  - [Fast Recovery] When congestion is detected via 3 duplicate ACKs:
    1. ssthresh is set to max { 2, ½ min { cwnd, rwnd })
    2. cwnd is set to **ssthresh**
    3. Fast retransmit and set cwnd = ssthresh + 3
    4. If there are additional duplicate ACKs, increment cwnd by 1 for each additional duplicate ACK
    5. One a new ACK arrives, set cwnd back to ssthresh from step 2
- **TCP Cubic**:
  - Uses AIMD with modified Additive Increase algorithm
    - Instead of increasing cwnd by 1, cwnd increases by $|(K - t)^3|$ where K = time when TCP window size will reach W_{max}, t = current time. W_{max} = window size at which congestion loss was detected previously
    - In essense, cwnd increases quickly initially but slows down nearer K
    - Once the cwnd exceed W_{max}, cwnd continues to increase by $|(K - t)^3|$ until congestion is detected again

Delay-based Congestion Control

- Uses RTT to determine the level of congestion in the network:
  - Normal RTT: Transmitting at the capacity, near congestion
  - Longer RTT: Network congestion
  - Smaller RTT: Bandwidth is underutilised
- **Bottleneck Bandwidth and Round-trip Propagation Time (BBR)**
  - Adjusts the cwnd based on measured throughput
    - Measured throughput = # bytes sent / $RTT_{measured}$
  - If measured throughput is very close to uncongested throughput → network is not congested → increase cwnd linearly
  - If measured throughput is far below uncongested throughput → network is congested → decrease cwnd inearly

Network-assisted Congestion Control

- Uses ECN (Explicit Congestion Notification) field in IP header:
  - 00: Non ECN-Capable Transport (Non-ECT)
  - 01/10: ECN Capable Transport (ECT(0) & ECT(1))
  - 11: Congestion Encountered (CE)