Internet Architecture

Entities

- Hosts: end systems (e.g. laptops, smartphones)
- Communication links (e.g. radio satellite, fiber, copper)
- Packet switches (e.g. routers, switches)
- Network edge: Hosts
- Network Core: interconnected routers

Delays

- Store-and-forward: entire packet must arrive at a router before it can be transmitted on next link
- **End-to-End** delay: total time taken to send one packet from src to dest
  $d_{node} = d_{prop} + d_{queue} + d_{trans} + d_{proc}$
- **Transmission** delay = L/R
- **Queuing** delay: Total = $n(n-1)(L/(2R))$

Latency (s)	**Bandwidth** (bps)	**Throughput** (bps)
Time for 1 bit of data to travel from one endpoint to another	Maximum number of bits that can be transmitted per unit time	Number of bits transmitted per unit time

- End-to-end throughput = transmission rate of bottleneck link
- Average and instantaneous throughput is <u>independent</u> of file size

Application Layer

- Resides in end-systems
- IP addresses: uniquely identifies host
- Port: identifies a process/service running on the host
- Socket: software interface to allow applications to send and receive messages

HTTP Protocol

- <u>H</u>yper<u>t</u>ext <u>T</u>ransfer <u>P</u>rotocol
- Uses TCP, port **80**
- Stateless

Non-Persistent HTTP

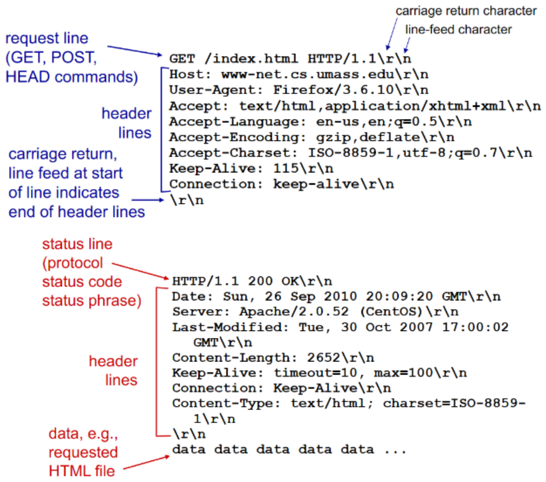
- Non-pipelined:
  1. Inititate TCP connection (1 RTT)
  2. GET HTML file (1 RTT + trans)
  3. Connection closes
  4. Repeat for each *n* additional ojects
- Pipelined (*m* files at once):
  1. Initiate TCP connection (1 RTT)
  2. GET HTML file (1 RTT + trans)
  3. Connection closes
  4. Repeat for $\lceil {}^{n}/_{m} \rceil$ times:
     a. Initiate TCP connection (1 RTT)
     b. GET *m* objects (1 RTT + *m* trans)
     c. 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 ⌈\sfrac{n}{m} ⌉ times:
   3.1. GET *m* objects (1 RTT + *m* trans)
4. Connection closes

HTTP Request and Response



Cookies

1. Client sends HTTP request to server for the first time
2. Server responds with cookie-ID in response header
3. Client accepts cookie?
   3.1. Yes: Cookie saved in client until TTL expires. Server creates entry in backend database
4. [If accepted] Client sends next HTTP request, with 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 proxys

1. Client sends request to proxy server
2. Proxy server checks if response is cached:
   2.2. Cached: response with cached response. Done
   2.3. Step 3
3. Proxy server forwards original request to origin server
4. Proxy server caches the response from origin server and responses back to client

- **Conditional GET**: Allows proxy server to query origin server to check if the cached response data has changed since it was last cached
  1. Client sends GET request to proxy server
  2. Proxy server sends Conditional GET request to origin server with "If-modified-since: <date-in-cached-response>" in header
  3. Origin server responds with:
     3.1. 304 Not Modified if response data was modified after date-in-cached-response. <u>Empty response body</u>
     3.2. 200 OK if data was not modified
  4. Proxy server updates the new response (if any) and responds back to client

DNS

- <u>D</u>omain <u>N</u>ame <u>S</u>ystem
- Maps host name to unique IP address
- Uses UDP, port **53**
- Local DNS servers do not belong to the global hierarchy of distributed servers
- Root DNS servers: top most in the hierarchy
- Top-level domain (TLD) servers: .com, .net, .edy etc.
- Authoritative DNS servers: organisations' servers e.g. .nus
- DNS resolution:
  1. Host makes request to Local DNS server
  2. Cached?
     2.2. Yes: Responds host. Done
     2.3. No: Step 3
  3. Local DNS server forwards query to root DNS server
  4. Root DNS server responds local DNS server with the next lower level DNS server X it should query next (e.g. .com)
  5. Local DNS server queries DNS server X…
  6. Repeat iteratively until host name is resolved
- Local DNS servers process DNS queries in a recursive manner, where as the distributed servers do so in an iterative manner

Socket Programming

- UDP: SOCK_DGRAM; TCP: SOCK_STREAM
- TCP: client calls .connect() to establish handshake
- UDP: uses sendto() with destination addr and port as parameters
- TCP: uses send()
- Client's port is automatically chosen by OS; Server needs to bind to correct port with .bind()
- Client can call .bind() to manually choose which port to use

Transport Layer

- Provides process-to-process communication between hosts
- Resides in end-systems

UDP Protocol

- <u>U</u>ser <u>D</u>atagram <u>P</u>rotocol
- Unreliable data transfer; connection-less
- Not full duplex
- UDP socket is identified by (dest IP, dest Port)

UDP header	
Source Port (2 bytes)	**Dest Port** (2 bytes)
Length (2 bytes)	**Checksum** (2 bytes)

- Both length and checksum <u>includes</u> header
- Checksum:
  - Sender: 1's complement addition with every 16-bit integers (i.e. includes carry-out, inverting bits after sum)
  - Receiver: sum every 16-bit integer, then add the sum to checksum. If all bits of result is 1, no error

Principles of Reliable Data Transfer

- Utilisation: fraction of time sender busy sending

$$U = \frac{L/R}{RTT + L/R}$$

Stop-and-Wait Protocols

- Low utilisation

rdt1.0

- Assumptions: underlying channel is perfectly reliable
- **Sender**:
  - Sends packet
- **Receiver**:
  - Receives packet

rdt2.0 (Checksum, ACK/NAK)

- Assumptions: bits may be flipped
- Checksum: detect errors
- ACK/NAK: recover from errors
- **Sender**:
  - Sends packet
  - ACK/NAK is corrupted or NAK: retransmit
- **Receiver**:
  - Packet is not corrupted: Responds with ACK
  - Packet is corrupted: Responds with NAK
- Flaw: Does not deal with duplicate packets received at the receiver

rdt2.1 (Checksum, ACK/NAK, seq #)

- seq #: receiver can detect duplicates from retransmission by sender
- **Sender**:
  - Sends packet
  - ACK/NAK is corrupted or NAK: retransmit
- **Receiver**:
  - Packet is not corrupted: Responds with ACK
  - Packet is corrupted: Responds with NAK
  - Packet is a duplicate: Discards packet and responds with ACK

rdt2.2 (Checksum, ACK, seq #)

- **Sender**:
  - Sends packet
  - ACK is corrupted or duplicate: retransmit
- **Receiver**:
  - Responds with ACK # = last packet received
  - Packet is a duplicate: Discards packet and responds with ACK # = last packet received

rdt3.0 (Checksum, ACK, seq #, Timer)

- Assumptions: bits may be flipped and packets can be lost
- **Sender**:
  - Sends packet
  - Timeout: retransmit
  - ACK is corrupted or duplicate: ignore, do nothing
- **Receiver**:
  - Responds with ACK # = last packet received
  - Packet is a duplicate: Responds with ACK # = last packet received
- May introduce packet reordering; solution: use larger range of sequence numbers

Pipelined Protocols

Go-Back-N

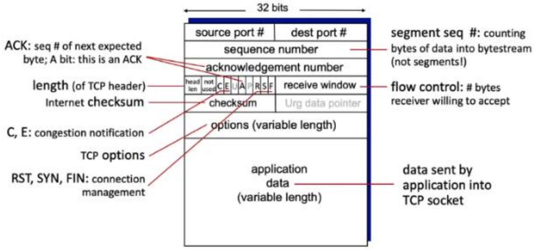
- Uses cumulative ACK
- Single timer; timer is for the oldest unACKed packet (in the window)
- **Sender**
  - Window will always contain only unACKed packets
  - ACK *n* → advance window base to *n* + 1, then sends packets newly added into the window
  - ACK is corrupted or duplicate: ignore, do nothing
  - Timeout: retransmit all packets in the window
- **Receiver**
  - Window size = 1
  - Responds with ACK # = last <u>correctly</u> received, <u>in-order</u> packet
  - Packet is corrupted, duplicate or out-of-order: discard, responds with appropriate ACK

Selective Repeat

- Uses individual ACKs
- Uses individual timers
- **Sender**
  - Window can contain both unACKed and ACKed packets
  - Window base is always at the oldest unACKed packet
  - ACK *base#* →advance window base to # of oldest unACKed packet
  - Sends packets newly added into the window
  - ACK not in window → ignore, do nothing
  - ACK is corrupted or duplicate: ignore, do nothing
  - Timeout: retransmit single packet
- **Receiver**
  - Responds with ACK # = # of packet, if uncorrupted
  - Packet is corrupted: discard, no ACK is sent
  - Packet is out-of-order: buffer, then ACK
  - Packet is in-order: delivers all buffered, in-order packets and advance window to # of next not-yet
  - Packet is duplicate: discard, then ACK

TCP Protocol

- <u>T</u>ransmission <u>C</u>ontrol <u>P</u>rotocol
- Reliable data transfer; connection-oriented
- TCP socket is identified by (src IP, src Port, dest IP, dest Port)
- Full duplex



Data transfer

- MSS = maximum size of <u>payload</u>
- Seq # = i^{th} position of the first byte in payload in full data; need not start at 0
- ACK # = expected next seq #
- Uses cumulative ACK
- Single timer; timer is for the oldest unACKed packet (in the window)
- **Sender**
  - Window will always contain only unACKed packets
  - ACK *n* → advance window base to *n* + 1, then sends packets newly added into the window
  - Timeout: retransmit single packet
- **Receiver**
  - Packet is out-of-order: buffer or discard, then respond with duplicate ACK
  - Packet is in-order: responds with appropriate ACK
  - Packet is corrupted: discard, no ACK is sent
- Congestion control: timeout interval doubles after each timeout for each packet
- Fast retransmit: if sender receives 4 consecutive duplicate ACKs, retransmit the corresponding packet

Receiver	Action	
Arrival of in-order packet with expected seq #	Delayed ACK (500ms) and wait for next packet. If no packet, send ACK	
Arrival of in-order packet with expected seq #, the one before has pending ACK	Immediately send single cumulative ACK, essentially ACK-ing both in-order packets with one ACK	
Arrival of out-of-order packet higher than expected seq #	Immediately send duplicate ACK	
Arrival of out-of-order packet higher than expected seq # that partially/completely fills gap	Immediately send ACK	

Timeout Estimation

$TimeoutInterval = EstimatedRTT + 4 \times DevRTT$
$EstimatedRTT = (1 - \alpha) \times EstimatedRTT + \alpha \times SampleRTT$
$DerivedRTT = (1 - \beta) \times DevRTT + \beta \times |SampleRTT - EstimatedRTT|$

- Typically, $\alpha = 0.125, \beta = 0.25$

Three-way Handshake

1. Client sends SYN segment with SYN = 1 (seq # = client_isn)
2. Server sends SYNACK segment with SYN = 1 (seq # = server_isn, ACK # = client_isn)
3. Client sends segment with SYN = 0 (seq # = client_isn + 1, ACK # = server_isn + 1)

Connection Termination

1. Client sends segment with FIN = 1 (seq # = x)
2. Server reponds with segment with ACKbit = 1 (ACK # = x + 1)
3. Server sends segment with FIN = 1 (seq # = y)
4. Client responds with segment with ACKbit = 1 (ACK # = y + 1)
5. Connection is terminated

- Either client or server can initiate the connection termination
- Once a FIN segment is send, the host can no longer send application data except for retransmitting segments sent before the FIN segment

Flow Control

- **Receiver**
  - Maintains `LastByteRcvd`, `LastByteRead`, `RcvBuffer`
  - Amount of data in buffer = `LastByteRcvd – LastByteRead`
  - Attaches `rwnd = RcvBuffer – (LastByteRcvd – LastByteRead)` to ACK packet
- **Sender**
  - Maintains `rwnd, LastByteSent, LastByteAcked`
  - Amount of data sent = `LastByteSent – LastByteAcked`
  - Ensures that `LastByteSent – LastByteAcked ≤ rwnd` whenever it wants to send a segment
- When `rwnd = 0`, sender continues to send 1 byte to receiver
  - Receive buffer is full: byte is dropped
  - Receive buffer is not full: Receiver sends ACK segment, letting sender know that buffer is not full

Network Layer

- Provides host-to-host communication
- Resides in network core
- **Forwarding**: move packets to appropriate router output port
- **Routing:** determine route taken from source to dest

IP address

- Per interface
- 32-bits
- a.b.c.d/x, x = subnet prefix
- Subnet mask: x MSB 1s, 32-x 0s
- 0.0.0.0/8: non-routable meta-address
- 255.255.255.255/32: broadcast address

Subnets

- Hosts can connect to one another without a router
- Hosts have same network prefix

DHCP Protocol

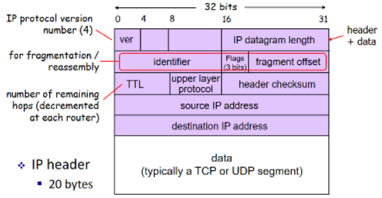
- Dynamic Host Configuration Protocol
- Dynamically obtain IP address when host joins a network
- Uses UDP, port **68** for client, port **67** for server
- ID identifies the client

1. DHCP discover: client sends
   <src: **0.0.0.0**, 68, dest: **255.255.255,255**, 67, yiaddr: **0.0.0.0**, ID>
2. DHCP offer: server responds
   <src: X.X.X.X, 68, dest: **255.255.255,255**, 67, yiaddr: **Y.Y.Y.Y**, ID, lease time>
3. DHCP request: client sends
   <src: **0.0.0.0**, 68, dest: **255.255.255,255**, 67, yiaddr: **Y.Y.Y.Y**, ID + 1, lease time>
4. DHCP ACK: server responds
   <src: X.X.X.X, 68, dest: **255.255.255,255**, 67, yiaddr: **Y.Y.Y.Y**, ID + 1, lease time>

- DHCP discover and offer are not mandatory
- Also returns address of first-hop router, name and IP of DNS server and network mask
- Client chooses any offer should there be multiple servers reponding

IP Protocol

- Internet Protocol



Fragmentation

- Max Fragment size = MTU of link-level frame
- Max payload size in fragment = MTU – 20 bytes
- # fragments = ⌈original size / MTU⌉
- **Flag**: 0 if is the last fragment, else 1
- **Offset**: Units of 8-bytes
- **ID**: ID of original datagram

Hierarchical Addressing

- Route aggregation: combine multiple IP addresses into a singe route advertisement → reduces size of forwarding table
- Routing table stores <Dest IP addr range, Next Hop> entries
- Longest prefix-matching

Intra-Autonomous Systems Routing

- Autonomous system: a group of routers under a common administrative domain
- Finds path between two routers within AS
- **Distance Vector Routing Algorithm**
  - Each router X knows the cost $C_{X,Y}$ of one hop
  - Each router X stores a routing table containing distance vectors $D_X(W)$ to all other routers in the AS network
  - Initial $D_X(W) = C_{X,W}$ if W is one-hop away, else $D_X(W) = \infty$
  - When $D_X(W)$ changes in router X, it updates its distance vectors then notifies its neighbours
  - Neighbours Y update their distance vectors then notifies their neighbours ……

- **Routing Information Protocol (RIP):** DV with $C_{X,Y}$ = # hops from X → Y
  - Uses UDP, port **520**
  - Routing table exchanged every 30s
  - No update from neighbour for 3 minutes → delete entry

ICMP Protocol

- Internet Control Message Protocol
- For error reporting or router signaling
- ICMP header: Type, Code, Checksum
- **ping**
  - Type = 8, Code = 0
  - Checks if connection to a remote host is successful
- TTL expired: Type = 11, Code = 0
- **Traceroute**
  - Returns route taken by a packer from one host to another by setting TTL = # hops

NAT

- Network Address Translation

1. LAN host (private IP, port) sends datagram to NAT router
2. NAT router modifies IP src IP and port in header: (private IP, port) → (NAT IP, unique port)
3. NAT saves (NAT IP, unique port) → (private IP, port) mapping in its NAT translation table
4. NAT sends modified datagram to server in WAN
5. Server responds back with datagram containing dest IP and port = (NAT IP, unique port)
6. NAT translates (NAT IP, unique port) → (private IP, port)
7. NAT forwards datagram to LAN host

Link Layer

- Resides in NIC card
- Involved adjacent nodes over a single link

Error Detection & Correction

1-D Parity Checking

- Detects all <u>odd</u> number of single bit errors, but cannot correct
- Poor performance with bursts or errors

2-D Parity Checking

- Detects all single or 2 bit errors
- Corrects all single bit errors

Cyclic Redundancy Check (CRC)

- Length of CRC = r
- Length of generator $G = r + 1$
- Sender:
  1. Append r number of 0s to data D
  2. Divide D by G (XOR)
  3. Remainder = CRC
- Receiver:
  1. Divide data D
  2. If remainder = 0, no errors
- Detects all <u>odd</u> number of single bit errors as well as all burst (consecutive bit) errors of $\leq r$ bits
- Detects burst errors of > r bits with $P = 1 – 0.5^r$

Multiple Access Protocols

- Ideal properties:
  1. Collision free
  2. Efficient: a transmitting node should transmit at max capacity
  3. Fairness: when n nodes want to transmit, each node transmits at a rate of R/n
  4. Fully decentralised: no external nodes

Channel Partitioning Protocols

Time Division Multiple Access (TDMA)

Collision Free	Efficient	Fairness	Decentralised
✓		✓	✓

- Divide into time frames and divide each time frame by # nodes
- Each node gets fixed length time slot to transmit at max throughput = R/n

Frequency Division Multiple Access (TDMA)

Collision Free	Efficient	Fairness	Decentralised
✓		✓	✓

- Divide channel spectrum into n frequency bands and each node transmits at one band at max throughput = R/n

Taking Turns Protocols

Polling Protocol

Collision Free	Efficient	Fairness	Decentralised
✓	(polling overhead)	✓	

- <u>Master node</u> polls which node to transmit next (and how many frames to transmit) in <u>round-robin</u> fashion

Token Passing Protocol

Collision Free	Efficient	Fairness	Decentralised
✓	(token passing overhead)	✓	✓

- Token (special frame) passed from one node to the next sequentially
- Only the node holding onto the token can transmit X # of frames
- If a node has nothing to transmit, it node immediately passes the token on to the next node
- Token loss possible → need to implement token recovery

Random Access Protocols

Slotted ALOHA

Collision Free	Efficient	Fairness	Decentralised
	✓	✓	✓

- Divide time into equal slots (of length ≥ time to transmit 1 frame); each node attempts to transmit at the beginning of a slot
- Collision: with probability p, transmit at next slot
- Efficiency ≈ 37%

Pure Division Multiple Access (TDMA)

Collision Free	Efficient	Fairness	Decentralised
	✓	✓	✓

- Nodes attempt to transmit at any given time
- Collision: wait for <u>1 frame transmission time</u>, then with probability p, transmit at next slot
- Efficiency ≈ 18%

Carrier Sense Multiple Access (CSMA)

Collision Free	Efficient	Fairness	Decentralised
	✓	✓	✓

- If channel is busy: wait for short while before sensing again
- If channel is idle: transmit
- Collision: Possible due to propagation delay; does not abort transmission

Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

Collision Free	Efficient	Fairness	Decentralised
	✓	✓	✓

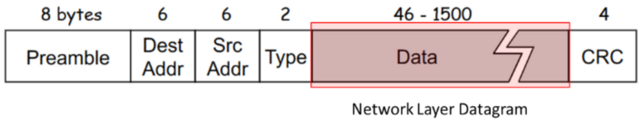
- If channel is busy: wait for short while before sensing again
- If channel is idle: transmit
- Collision: Possible due to propagation delay; <u>abort</u> transmission and retransmit after a <u>random</u> delay
- Exponential backoff: For each frame,
  - After the n^{th} collision, next delay is random $K \in \{1, 2, …, 2^n -1\}$ time units, up to $n = 10$
  - Time unit = 512 bit time (time taken to transmit 512 bits)
  - Actual delay = 512K/R
- Ethernet minimum frame size = 64 bytes to ensure $d_{trans} > d_{prop}$ and collisions are detected

Ethernet

MAC Address

- Burnt into NIC card (i.e. fixed and globally unique)
- A host can have multiple MAC addresses as long as it has multiple NIC
- 48-bit hexadecimal e.g. 5C-F9-DD-E8-E3-D2
- Broadcast address = FF-FF-FF-FF-FF-FF
- **LAN**: network that interconnects hosts within geographical area

Ethernet Frame



- src & dest addr: NIC only delivers packets up if dest MAC of received packet is broadcast address or own address
- Preamble: 7 bytes of 10101010 (AA) and 1 byte of 10101011 (AB) or "start-of-frame"
  - Synchronise sender and receiver clock rates
- Type: indicates the higher layer protocol (e.g. IP)

Physical Topology

- **Bus**:
  - Broadcast LAN, all devices connect to it
  - Bus is single point of failure, hard to troubleshoot and slow due to collisions
- **Star**:

Hub	Switch
Physical layer device	Link layer device
Acts on individual bits	Acts on frames (store-and-forward)
Collisions possible	Simultaneous transmissions without collisions
Bits are copied, boosted and broadcasted to all interfaces	Frames only transmitted to one interface
Cheap	Expensive

Ethernet Switch

- Uses CSMA/CD
- Transparent, plug-and-play (self-learning)
- Buffers frames
- Frame filtering/forwarding (Host A → Host B, A ≠ B)
  1. Switch receives frame from interface X occupied by Host A
  2. Switch stores <MAC_A, Interface_X, TTL> in switch table
  3. Does switch table contain MAC_B?
     - Yes: Forward to table[MAC_B].Interface
     - No: Broadcast to all interfaces
- If src MAC = dest MAC, frame is filtered

ARP Protocol

- <u>A</u>ddress <u>R</u>esolution <u>P</u>rotocol
- Maps IP address to MAC address
- ARP tables reside in each interface

Same Subnet (Host A → Host B)

1. Host A's ARP table contains IP_B?
   - Yes: Transmit ethernet frame with dest MAC = MAC_B
   - No: Transmit **ARP request** packet with dest MAC = FF-FF-FF-FF-FF-FF
2. Switch receives frame from Host A
   - Switch keeps a record <MAC_A, Interface_X, TTL> if there isn't one
   - Not ARP request?
     - Switch table contains MAC_B?
       - Yes: Forward frame to table[MAC_B].Interface. **DONE**
       - No: Broadcast frame to all interfaces. **DONE**
   - ARP request?
     - Broadcast frame to all interfaces
3. Host B receives the frame and responds with its MAC address
   - Host B keeps a record of <IP_A, MAC_A> in its ARP table if there isn't one
4. Switch receives frame from Host B
   - Switch keeps a record <MAC_B, Interface_Y, TTL> if there isn't one
5. Switch forwards frame to Interface_X (Host A)
6. Host A keeps a record of <IP_B, MAC_B, TTL>. Repeat from 1 with IP datagram instead of ARP request

Same Subnets (Host A → Host B)

1. Host A creates and transmits ethernet frame with src MAC = MAC_A and dest MAC = MAC_Router_Interface_A to router
2. Router has two interfaces: one resides in Host A's subnet (Interface_A) and the other in Host B's subnet (Interface_B)
3. Interface_A of router receives the frame, extracts the IP datagram (to see Host B's address)
4. Router checks Interface_B's ARP table for Host B's MAC address
   - If it does not exist, transmit ARP packet to switch in subnet B
5. Router modifies the frame and forwards the frame with src MAC = MAC_Router_Interface_B and dest = MAC_B to Host B

<u>Network Security</u>

- **Confidentiality**: only sender and intended receivers can understand message contents
- **Authentication**: sender and receiver need to confirm each other's identity
- **Integrity**: messages are not altered without detection
- **Access & Availability**
- Cryptography Notation:

$$\underbrace{K_B}_{\text{decrypt}}(\underbrace{\underbrace{K_A}_{\text{encrypt}}(\underbrace{m}_{\text{plaintext}})}_{\text{ciphertext}}) = m$$

Confidentiality

- **Symmetric** Key Cryptography
  - Sender and receiver use the same encryption key (***K_s***)
  - Sender and receiver need to agree on same key
- **Public** Key Cryptography
  - Sender uses public key; receiver uses its own private key
  - Requirements
    - m = $\mathbf{K^-(K^+(m))}$
    - Given $\mathbf{K^+}$, it should be impossible to find $\mathbf{K^-}$

Caesar's cipher (Symmetric)

- A substitution cipher using fixed shift of alphabet
- K_s = # positions to shift (26)

Monoalphabetic cipher (Symmetric)

- A substitution cipher by substituting one letter for another
- K_s = subtitution cipher (26!)
- Can be cracked with Statistical Analysis

Polyalphabetic cipher (Symmetric)

- Using *n* substitution ciphers and a cycling pattern, substitute the i^{th} symbol in plaintext using the i^{th} cipher in the cyclic pattern
- K_s = *n* substitution ciphers + cycling pattern

Block cipher (Symmetric)

- Message is processed in blocks of *X* bits and each block is encoded in a one-to-one mapping
- K_s = block cipher (2^X!)
- DES (Data Encryption Standard)
  - 56-bit symmetric key, 64-bit block
- AES (Advanced Encryption Standard)
  - 128/192/256-bit symmetric key, 128-bit block

RSA Encryption (Public)

- Convert message to bit pattern then to integer (e.g. 00001100 = 12)
- $K^-(K^+(m)) = m = K^+(K^-(m))$

1. Choose two large prime numbers ***p***, ***q***
2. Compute ***n*** = *pq*, ***z*** = (*p* – 1)(*q* – 1)
3. Choose ***e*** s.t. *e* < *n* and has no common factors with z (relatively prime)
4. Choose ***d*** s.t. *ed* – 1 is exactly divisible by *z* (*ed* mod *z* = 1)
5. Receiver's $\mathbf{K^+}$ = (*n, e*), Receiver's $\mathbf{K^-}$ = (*n*, *d*)
6. Sender Encrypt: m^e mod *n* = c, Receiver Decrypt: c^d mod *n* = m

- RSA + DES/AES:
  1. Select <u>session key</u> K_s
  2. Use RSA to transfer K_s
  3. K_s is used as the symmetric key in DES/AES for encryption for the session
     - Faster, since RSA is a lot slower than DES/AES

Message Integrity

- **md5**: generates short, 128-bit digests
- Authentication key *s*: shared only between sender and receiver
- Sender sends Message Authentication Code, *H*(*m* + *s*), along with the message i.e. $m \oplus H(m + s)$
- Receiver calculates *H*(*m* + *s*) and compares it with the one it receives

Authentication

- Sender attaches digital signature which must be
  1. Verifiable by receiver
  2. Unforgeable
- Signature: Sender encrypts hashed message with its private key $\mathbf{K^-}$ i.e. $m \oplus K^-(H(m))$
- Receiver uses sender's private key $\mathbf{K^+}$ to decrypt the signature and checks if $K^+(K^-(H(m))) = H(m)$

Certification Authorities

- Public database containing a list of public keys $\mathbf{K^+_X}$, each signed by and certified by the CA
- CA signs public keys using its own private key i.e. $\mathbf{K^-(K^+_X)}$
- A list of trusted CA's public keys are hardcoded in the OS
- To get sender's private key $\mathbf{K^+_X}$, receiver gets sender's certificate $\mathbf{K^-(K^+_X)}$ and decrypts it using CA's public key ($\mathbf{K^+(K^-(K^+_X)) = K^+_X}$)