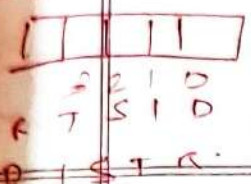
• ENDIANNESS



(doesn't the digital clock work on battery? ↳ req power, isn't?)

classmate Date 28/07/25 Page

Are they hardware? or focus on a device? (Combination of both)

i) Distributed systems: hardware components as networked computers which communicate by message passing
→ collection of independent computers (hardware) or processors (software) that appears to be coherent system to achieve a goal
→ multiple# computers, interconnected & shared system (3 features)
→ failure prone: as all of them try to achieve the same goal
→ it is layered over networks ⇒ for each layer specific protocols present

ii) Perfect synchronization not possible ← due to delays in network

iii) HTTP
↳ one obj. per TCP
↳ non-persistent connection
↳ persistent "
autonomous, programmable, asynchronous & failure-prone

→ Stateless (as complex if they have to manage) - Servers maintains no info about past client requests
→ uses TCP transport service

iv) Issues Related to Distributed Systems
• No global clock: no single notion of correct time (same ones)
• unpredictable failure of components
• highly variable bandwidth
• Possibly large & variable latency → (in Internet, it's high)
→ for memory (smtg), it's in ns
• large no. of hosts — challenges → concurrent* execution, independent failure*, autonomous administration, heterogeneous

v) Network communication — challenges → Asynchronous*, unreliable*, insecure

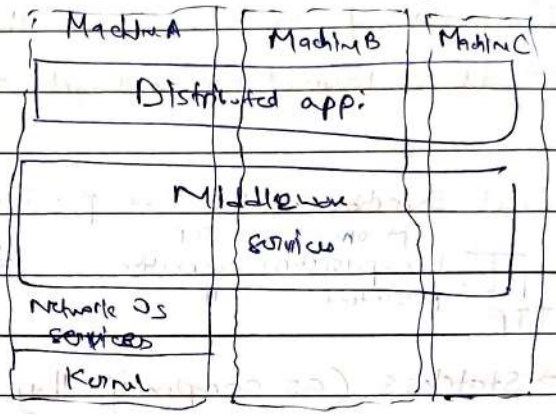
Common goal — (concurrency)*, transparency

* real issue

vi) Middleware — (DS ~~software~~) (may be DS req it?)
• Software programs
• Why necessary? — Distributed systems: functional distribution → different functions exist
inherent
↑ application differs

viii) Challenges & Goals of Distributed Systems → ex. remote method invocation, group communication, replication

- Heterogeneity – can be handled by Middleware
  - a SW layer providing a programming abstraction as well as masking the heterogeneity of the underlying networks, HW, OS, & programming languages
  - (variety & difference of underlying Network/ Infrastructures)



Software layers in DS

- Openness – determines whether system can be extended & reimplemented in various ways
- Security – DS must protect shared info & resources
- Scalable – Can handle additional no. of users/resources without suffering noticeable loss of performance
- Failure Handling – Failure in DS is partial
  - (try Redundancy)
  - making handling of failures difficult ⇐ {some components fails while others continue to function
  - How to handle ? → (i) Masking failure – trying retransmission
    - (ii) Detecting "
    - (iii) Tolerating "
  - How to recover ? → (i) Rollback
    - (ii) undo/redo in transmission
- Concurrency – multiple users – Stale data in shared system
  - ↳ can lead to deadlock.

- ~~Hetro~~ Transparency — concealing the heterogenous & distributed nature of the system so that it appears to the user like one system

challenges? → categories

((i) Access — access local & remote resources using identical operations) (:....)

28/07/25

Topic : System Model → describes

(ii) Architectural Model : structure of system in terms of components (like roles, how interact ← (In high level) then done, & how responsibilities divided) **

↑ tells what connected to that & how mapping of ~~server & client~~ processor to processor not 1 to 1 as a ~~server~~ processor can provide many services

Distributed System Architecture — 2 types

Client - Server Architecture → distinction betn Client & Servers (note : there are machines)

V/s

Distributed Object Architecture → no distinction → any one can ask/provide

→ tells how the system behaves

Fundamental Models — a system model

i) the entities : defines
ii) their interactions
iii) Characteristics that affect their behavior

→ Purposes : (i) Explicitly states all relevant assumptions
(ii) generalise what is possible or not based on the assumptions.

* Concerned with properties common to all architectural model

↓

refer slides for 3 key features

(i) No global time in a DS
(ii) communication occur by messages so vulnerable to attacks, delays etc
(iii) failure prone

bounds on processing time (min & max), msg transmission delay, local clock drift rate

→ Synchronous (strong assumptions about time)

→ Asynchronous (no assumptions)

classmate Page

- 3 models are: (i) Interaction model – deals with performance & the difficulty of setting time limits in a DS

types of failure
(i) omission
(ii) arbitrary
(iii) timing

(ii) Failure model – attempts to give a precise specification of the faults that can be exhibited by a process & communication channels
→ defines reliable communication & correct processes
(Message eventually reaches the destination) : validity ← features → integrity : no manipulation of messages

(iii) Security model – discusses the possible threats to processes & communication channels
→ introduces the concept of a secure channel, which is secure against those threats

achieved by:
(i) access rights
(ii) principal (??)

- Performance of communication channel maintained by:
  (i) latency: Sender to receiver (time taken)
  → includes propagation delays – propagation in a network
  network delays – delay in accessing the network
  OS delays – time taken by OS
  (ii) bandwidth: capacity of communication channel
  (iii) jitter: variation in time taken to deliver a series of messages

- Failure Types
  (i) Omission failure → process → crash stop (fail-stop)
  → crash recovery
  → communication → send omission: loss betn sending process & outgoing msg buffers
  → channel omission
  → receive "

Can be masked

(ii) Arbitrary failures → process – omits intended processing steps or takes unintended processing steps
→ Channel – msgs corrupted, duplicated, large delays etc

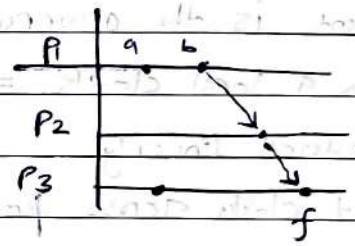
(are) Byzantine failure ← both examples
caused by MITM, viruses, worms etc

classmate
Date 30/07/25
Page

Topic : Time in Distributed System

- Berkey Algo : server after computation sends the offset & not the time to its clients
- Network Time Protocol : Best works with Symmetric mode
- causality : means cause & effect
- 2 local events in 2 different machines can't be directly compared unless explicit relationship !

~~Solar~~ Solar clocks also?



relationship betn a & c ?
→ Can't say !

- Physical Clocks

i) International Atomic Time (TAI) : no. of ticks of cesium 133 atom since 1/1/58 (atomic second)

ii) Atom Clock : One second defined as (since 1967) 9,192,631,770 transitions of the atom cesium 133

iii) Slowdown of earth $\xrightarrow{\text{lead to}}$ Introducing leap seconds
⇒ Correction of TAI is called Universal Coordinated Time (UTC)

iv) if $C(t)$ be a perfect clock

(i) Clock $C_i(t)$ is called correct at time t if $C_i(t) = C(t)$

(ii) " " " " accurate " " " if $\frac{d(C_i(t))}{dt} = \frac{d(C(t))}{dt} = 1$

(iii) 2 clocks $C_i(t)$ & $C_k(t)$ considered synchronize at time t if $C_i(t) = C_k(t)$

- Clocks

→ Computers contain physical clocks (Crystal oscillator)
→ Clock skew : Instantaneous difference betn the readings of two clocks
→ Clock drift : Crystal based clocks count time at different rates & so diverge
↳ relative difference in clock frequencies (rates) of 2 processes

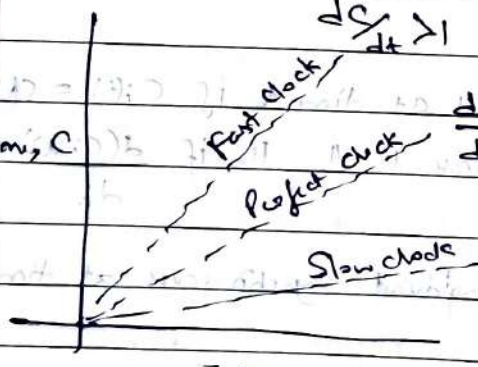
- Why Synchronisation? — Correctness
  fairness

- Process : present in asynchronous DS
  - has a state
  - takes actions which change its state
  - Event is the occurrence of an action
  - has a local clock ⇒ events within a process can be ordered linearly
  - but clocks across process not synchronised

- Max drift rate bet' 2 clocks with similar MDR is 2*MDR (MDR)

NOTE: A non-zero clock drift causes skew to increase (eventually)

- Synchronization → External (synchronised with an external source of time)
  → Internal (synchronised with one another)

- 

← relation bet' clock time & UTC when clock ticks at different rates

- for synchronizing (Asynchronous DS) → Cristian's Algo
  Berkeley Algo
  Internal Synchronisation
  NTP (Network Time Protocol)

(Check for more on next page)

classmate
Date
Page

↳ If 2 processes needn't req exact time

- Logical Clocks → kind of says ordering more important
  ↳ & provides timestamps //
  but if clocks need to have sanction ⇒ physical clock
  → uses the happens before relation
  → assigns logical timestamps to events

Lamport Logical clock → each process has a local counter
- (on each event it increments its clocks)
- When sending a message, it sends its clock too
- When receiving, it sets it's clock to max(local, received) + 1

Topic: Vector Clocks, Global State 04/08/25

- Cut consistency: ⇒ if receive message is recorded, so sent message should also be recorded!

NOTE: if $a \to b$ (events in same process) $\Rightarrow L(a) < L(b)$
but if $L(a) < L(b) \not\Rightarrow a \to b$
because a & b could be events in 2 different processes & still be concurrent though logical clock not!!

- Vector Logical Clocks
  i) address the issue as described above (↑)
  ii) instead of using a single logical clock, each process has N logical clocks for the other (N-1) processes
  ↳ (kind of vector clock)
  → without synchronization, captured using causality

- Global State = States of all Processes + state of all communication channels
  → captures the instantaneous state of each process + each communication channel (i.e messages in transit on the channels)

(Consistent cut ⇐ is if & only if it corresponds to consistent cut)

Topic : Chandy-Lamport Algorithm (CLA)

- NOTE: Channel marked as empty ⇒ no messages has been received in there
- Linearization may not be unique
- To tell happen before relationship ⇒ (vector comparison can be done)
- CLA results in a consistent global state
  ↳ i.e if e_j is included in cut & $e_i \rightarrow e_j$, ⇒ $e_i \in$ cut
  proved by: fifo ordering of channel

⇒ if $e_j \rightarrow P_j$ records its state
but we say $e_j \leftarrow P_j$ records it state

⇓

Since P_j recorded its state
it will send marker
& first message sent to $\int$ in C_{ij} channel
is marker then message for e_j to occur
⇒ $e_j \leftarrow P_j$ recorded its state

⇓

Contradiction

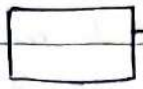
06/08/25

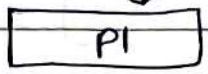
Previous : Synchronization //

Topic : Process Communication

- how ↓ to implement multicast in DS : reliable or ordered
  from one ↓ process to a group of process
- Distributed debugging
  → it is about capturing consistent global state to check whether meaningful conditions (like $|x_i - x_j| \leq \delta$) actually held during execution not just due to time artifacts.
  ↳ capture state + monitor + evaluates
  global state → i.e predicate → definitely / possibly

11/08/25

-  → B multicast
  i.e send (m) to group g



p1 receive (m)
& B-deliver (m) at p
i-e ~~Someone in the network first gets it & then deliver the same~~
process p deliver it to the application

- Reliable multicast
  - Integrity : one time sending
  - validity : a process which sends m correctly also delivers
    - guarantees liveness m (+ itself)
  - agreement (to the sender) : if some one correct process delivers message m then all other correct processes in group(m) will eventually deliver m //
    - either all or nothing

- Order of multicast
  - FIFO : (m) delivered first then m' get delivered (per sender) — if multicast(g,m) first issued
  - Causal : multicast(g,m) → multicast(g,m') (2 people) ⟹ m delivered first then m'
  - Total : if m first then m' (for all)
    - for all it's same the order

FIFO (per sender)	Causal ordering	Total ordering
(nothing about different senders) • messages from the same sender are delivered in the order they were sent	If one msg could have influenced another, deliver them in causal order	All processes deliver all messages in the exact same order, no matter what !

classmate
Date 12/08/25
Page

Flooding

Initially do: if pid = root, then ← [Case 1: for 1 msg
Case 2: for multiple msg
↑ modify (1)
send_msg ← true
Send M to all neighbours
else
send_msg ← false } *

Upon receiving M do: ← it's counter (containing 0 or 1 bit)
if send_msg = false then
send_msg ← true
Send M to all neighbours
← time to complete

Time Complexity: Diameter (farthest distance btn any 2 nodes) - O(n)
Space " :
Message " : Edges - O(E)
↑ how many messages were transferred

Case 3: Setting the parent pointer
A → B
↑ sets parent pointer to A
Need modification in Case 1
Initially do: if: pid = root, then parent ← root
Send M to all neighbours
* → else
parent ← ⊥ (null)

Upon receiving M do:
if parent == ⊥ then
parent = p
Send M to all neighbours

#To know all the children

initially do:

non_children = ϕ

if pid = root, then

parent ← root

children ← {root}

Send M to all neighbours

else

parent ← $\perp$

Children ← ϕ

Upon receiving M from p do:

If parent = $\perp$ then

parent ← p

Send ⟨ack⟩ to p // to mark as child

Snd M to all neighbours

else

Send ⟨nack⟩ to p // to mark as non-child

$\perp$

upon receiving ⟨ack⟩ from p do:

children ← children ∪ {p}

Upon receiving ⟨nack⟩ from p do:

non-children ← non-children ∪ {p}

Correctness: Since everyone sends the message so eventually everyone gets it & set the children/Non children in $O(D)$

Hamilton Path: path connecting all nodes visiting only once (NP complete)

Euler Path: edges visited only once (Polynomial)

Construction of spanning tree (Algo)

→ use BFS or DFS

→ Using DFS

Case 1: designated root (P_i) (discussed)

Case 2: multiple initiator ← (Only 1 will Spanning tree should be constructed (by some coordinated actions))

- Initially assume: parent ← ⊥, children = ϕ, unexplored includes all the neighbours of P_i

for root:

- upon receiving no message
  - if i = r and parent = ⊥ then
    - parent ← i
    - let P_j be a process in unexplored
    - remove P_j from unexplored
    - send M to P_j

- upon receiving M from neighbour P_j
  - if parent = ⊥ then
    - parent ← j
    - remove P_j from unexplored // as it received it from
    - if unexplored ≠ ϕ then
      - let P_k be a process in unexplored
      - remove P_k from unexplored
      - send M to P_k
    - else
      - send <parent> to parent
  - else:
    - send <reject> to P_j

• Upon receiving <parent> or <reject> from neighbour P_j:

if received <parent> then

add j to children

If unexplored $= \phi$ then

if parent $\neq i$ then send <parent> to parent

terminate ← for current process P_i

else

let P_k be a process in unexplored

remove P_k from explored

Send M to P_k

How do we know overall termination happened?

→ Central Node which collects all terminated detail

No. of messages passed = $O(E) \approx O(4M)$

$4*E$ (worst case)

Time complexity = $O(E) \approx O(M)$

13/08/25

Constructing DFS spanning tree

w/o Specified root (wrt P_i) (below steps)

• Initially: Parent ← ⊥, leader ← P_i

Children ← ϕ

Unexplored ← all neighbours

• Upon receiving no message:

If parent = ⊥, then

leader ← id, parent ← i

let P_j be a process in unexplored

remove P_j from unexplored

Send <leader> to P_j

(leader variable)

new_id

- Upon receiving <~~search~~> from P_j

  if leader < ~~search~~ new_id, then

    leader ← new_id, parent ← j

    unexplored ← all neighbours of P_i except P_j

    if unexplored ≠ ϕ then

      let P_k be a process in unexplored

      remove P_k from unexplored

      Send <leader> to P_k

    else

      Send <parent> to parent

  else if leader = new_id, then

    Send <already> to P_j

    // otherwise leader > new_id and the DFS for new id stalled

    of <already>

- Upon receiving <parent> from P_j:

  If received <parent> then:

    add j to all children

    if unexplored = ϕ then

      if parent ≠ i, then Send

        <parent> to parent

      else terminate as root of the spanning tree

    else let P_k be a process in unexplored

      remove P_k from unexplored

      Send <leader> to P_k

If received <already> then

  {nothing to do}

- Time complexity = $O(n \times M)$ (per process)

Message complexity = $O(\text{no. of nodes} \times M)$

Topic: Mutual Exclusion :-

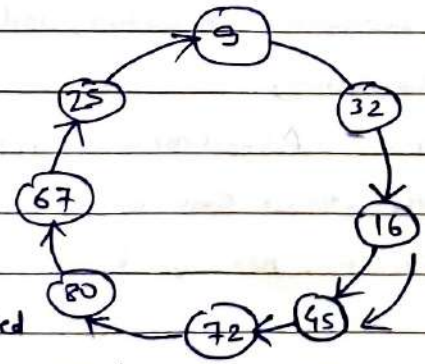
↓ a piece of code : at most 1 client executing it at any point of time

- ensured through message passing based protocols
  - → Enter()
  - → accessResource()
  - → exit()
- Requirements
  - → Safety : only 1 process
  - → liveness : should be actively arriving
  - → ordering : req. granted in the order they are made
- DS mutual exclusions performance evaluation criteria
  - → Bandwidth : # messages sent in each entry & exist operation
  - → Client delay : (req operation requested & granted) time duration
  - → Synchronization delay : (entry & exit of process)

- In Token Ring Approach
  - → there is no separate Entry() message
  - → only has the exit() message which corresponds to token being passed

18/08/25

- # Ring Based Algorithm : for leader election
  - → each process has an Id



(i) Send election msg <election, id>

(ii) if $P_i < x$, let's say P_j ↔ P_i

<election, 16> <election, j>

at i, Check i<j pass elseif replace j with i

Leader realisation

→ if a process receives it's own msg, that means it was not blocked & it's leader

→ pass <elected, 80> : to inform others → (termination)

circulation of highest PID then it receives its own msg (ii) after that it pass elected (iii)

classmate Date Page

(If the max pid process is adjacent to the process who starts the election) !! : Worst case

Best case: When highest PID starts

- How many messages req here? $3n-1$ $2n$

→ Safety property - ~~bbb~~ single process is elected as a leader

→ Liveliness " - if any process starts the election process, then a leader is elected at the end of process. Provided no process failure

- What if a process in the ring has failed?

→ Some process should detect it

MIDSEM

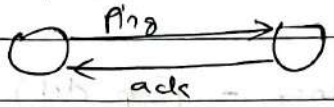
L1 · DS - hardware components / software - located at networked computers - communicate by message passing

- Characteristics:
  (i) multiple computers (ii) interconnections
  (iii) shared state
- HTTP Stateless - Server maintain no info about past client req
  ↳ persistent vs non-persistent
- Types of distribution
  (i) Functional (ii) Inherent
- DS issues
  (i) no global clock (ii) unpredictable failure of components
  (iii) highly variable bandwidth (iv) Possibly large & variable latency
  (v) Large no. of hosts
- Challenging Properties
  (i) Multiple computers - Concurrent execution, independent failure
  (ii) Common goal - Consistency
  (iii) Network communication - Asynchronous, unreliable
- RFC - specifications of the Internet communication protocol specifications for applications run over them

- Heterogeneity – ensured by Middleware (ensures communication & resource sharing)
  Openness – by RFCs, CORBA
  Scalability – 3 metrics : (i) no. of users / resources
  (ii) distance between the farthest nodes in the system (network radius)
  (iii) no. of organisations exerting control over the pieces of the system
  Transparency – (app def) heterogeneous nature of system should be hidden

L2

- DS Architecture Type : (i) Client – server
  (ii) peer to peer
- Fundamental Models : (i) Interaction Model → Synchronous DS / Asynchronous DS
  (ii) Failure "
  (iii) Security "
- Performance : (i) Latency – delay between sending of a message by one process & its receipt by another
  in classes, propagation – time taken for first string of bit to reach destination
  Network – delay in accessing the network
  OS – t.t by OS communication services at both sending & receiving process
  (ii) Bandwidth : capacity of the communication channel
  (iii) Jitter : Variation in time taken to deliver a series of messages
- Synchronous – processing time has a bound !!
  same with message transmission delay (+ local clock drift rate)
- Types of failure – (i) omission – when a process or a channel fails to perform actions that it is supposed to do → ex. process crash, message drop
  (ii) Arbitrary (Byzantine) Failures – any type of error
  (iii) Timing Failures

- Omission Failure - (i) Send-omission : b/w sending process & outgoing msg buffer
  (ii) Channel ll : loss in the communication channel
  (iii) receive ll : loss in incoming msg buffer & receiving process
- Reliable comm- defined by: (i) Validity : eventual reaching
  (ii) Integrity : (msg received == msg sent)
- Detecting a crashed process
  M1: 

  (i) Synchronous :
  timeout (Δ_1) = 2 (max network delay)
  (ii) asynchronous :
  timeout (Δ_2) = k * (max observed RTT)

M2: 

after which heartbeat send
Timeout value = $T + \Delta_2$

Synchronous : Δ_2 = max network delay - min network delay

asynchronous : Δ_2 = k (observed delay)

- Correctness of failure detection
  (i) completeness : every failed process is eventually detected
  (ii) Accuracy : every detected failure corresponds to a crashed process (no mistakes)

both achievable in synchronous but not in asynchronous

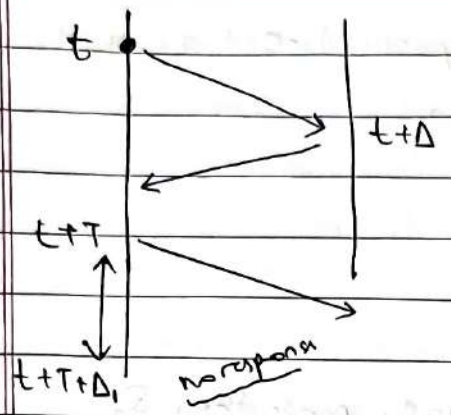
↳ piggy heartbeat : complete
→ for accuracy & incomplete : never report failure ☺

Worst case for failure detection
- ping ack : $T + \Delta_1 - \Delta$: Δ is t.t for last ping from p to reach q
- heartbeat : $\Delta + T + \Delta_2$: Δ is t.t for last ping from q to reach p

$|A-B| < D \Rightarrow |A-b| - |C-b| < 0$

$|C-B| < D \Rightarrow A - b \leq C + b < 0$

classmate
Date
Page



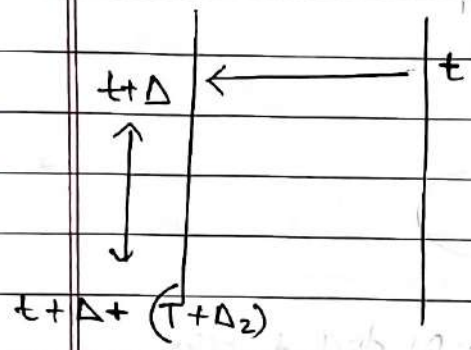
#Synchronous system

$\Rightarrow t + T + \Delta_1 - (t + \Delta)$

$\Rightarrow T + \Delta_1 - \Delta$ //

Δ → min network delay

Δ_1 → worst case time delay for failure detection



$\Rightarrow (t + \Delta) + (T + \Delta_2) - t$

$\Rightarrow \Delta + T + \Delta_2$

Δ → max network delay //

NOTE: In case of asynchronous system, Increasing Δ_1 or $\Delta_2 \Rightarrow$ Increase in accuracy ~~of~~ but increase in failure time

Failure Detection

L-3

- Clock skew: relative difference between two clock values
  Clock drift rate: change in skew from a perfect reference clock per unit time
- 2 type of synchronization: (i) external : with an authoritative clock
  (ii) internal
- NOTE:

$|A-b| \leq D$ and $|C-b| \leq D$ $\Rightarrow$ $|A-C| = |(A-b) - (C-b)|$

$\rightarrow |A-C| \leq |A-b| + |C-b|$

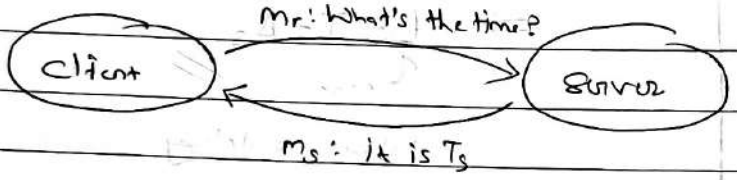
$\leq 2D$ //

- 2 clocks if drifting from UTC in opposite direction at a time Δt after they are synchronised,
  $\Rightarrow$ they are $2p\Delta t$ apart

  where,
  p : maximum drift rate

Also, if 2 clocks shouldn't differ by more than δ,
$\Rightarrow$ it should be reset every $\delta/2p$ seconds atleast

NOTE:



What time T_c should adjust its local clock to after receiving m_s ?

$$T_c = T_s + (min + max)/2,$$

$$Skew(client, server) \leq (max - min)/2$$

How to synchronise //

(i) Cristian Algo

$$T_c = T_s + (T_{round}/2)$$

$$Skew \leq (T_{round}/2) - min$$

↖ Minimum 1 way network delay (≈ 0)

$$\leq (T_{round})/2$$

(ii) Berkeley Algo : only internal synchronisation

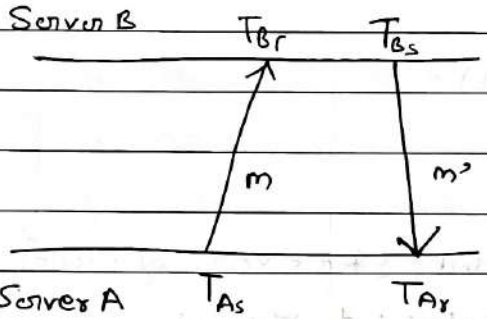
- Server asks all their time
- Christian algo on each
- avg the local time

→ Send the offset to all

(iii) Network Time Protocol

- multicast timestamps - low accuracy
- procedure call (christian algo) - higher accuracy
- Symmetric mode - highest accuracy

⇒ • A & B exchange messages
• record the send & receive timestamps
• calculate the offset bet one another



$$T_{B_r} = T_{A_s} + t + O$$

$$T_{A_r} = T_{B_s} + t' - O$$

t & t' : transmission time
O : offset of B relative to A

Normal clocks req synchronisation

L-4

⇓

logical clocks : time sync not req
order of events req

- Happened Before Relationship (HB) : denoted by $\longrightarrow$

$e \to e'$ means e happened before e'

$e \to_i e'$ means e happened before e' as observed by p_i

→ if $\exists p_i$, $e \to_i e'$ then $e \to e'$ $\{\Rightarrow l(e) < l(e')\}$

(x) true

as $e \to e'$ or $e' \to e$

NOTE: Algorithm (Lamport's Timestamping)

(A)

(i) Set $L_i = 0$

(ii) increment on any event

(iii) on send message → send L_i

(iv) receiver: Set $L_i = \max(L_i, t)$ ← received

(v) then increment L_i before timestamping receive msg

Limitation: Partial order

$a \rightarrow b \Rightarrow L(a) < L(b)$

$\not\Leftarrow$

⇓ hence,

Vector clocks (instead of just 1 clock, store a vector of clocks)

⇒ similar to Algo (A) but extended to vector of clocks on receiving a msg

note: increment just own (but take max for all)

- $VT_1 = VT_2 \Rightarrow VT_1[i] = VT_2[i] \ \forall i$

  $VT_1 \leq VT_2 \Rightarrow VT_1[i] \leq VT_2[i] \ \forall i$

  $VT_1 < VT_2 \Rightarrow VT_1[i] \leq VT_2[i] \ \forall i$

  && $VT_1[j] < VT_2[j] \ \exists j$

⇒ VT_1 is concurrent with VT_2

iff $(\text{not}(VT_1 \leq VT_2) \ \& \ \text{not}(VT_2 \leq VT_1))$

{NOTE: ordering of events based on timestamps

i.e $VT_1 < VT_2$} *

- Causality violation: occurs when order of messages causes an action based on information that another host has not yet received

⇓

⇒ If incoming vector timestamp < local V.T.S

Global State (== state of all processes + state of all communication channels)

- Includes instantaneous state of each process

how to get it without synchronising the clocks?

- 3 types of events: (i) local computation
  (ii) sending a message
  (iii) receiving a message

e_i^n : nth event at p_i

NOTE:

(i) History $(p_i) \Rightarrow h_i = \langle e_i^0, e_i^1, \ldots \rangle$

(ii) prefix history $(p_i^k) = h_i^k = \langle e_i^0, e_i^1, \ldots e_i^k \rangle$

S_i^k : p_i's state immediately after kth event

(iii) For a set of processes $\langle p_1, p_2, \ldots p_n \rangle$

$H = \cup_i (h_i)$ {global history}

NOTE: a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup h_3^{c_3} \ldots h_n^{c_n}$

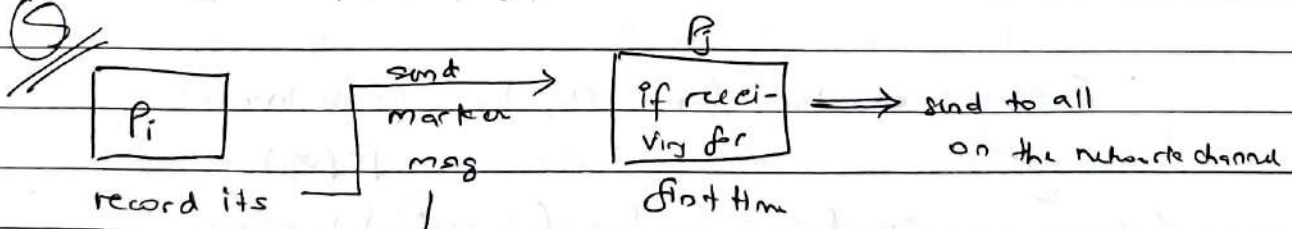
Also frontier of $C = \{ e_i^{c_i}, i = 1, \ldots n \}$

global state S corresponds to cut $C = \cup_i (S_i^{c_i})$

Consistent: iff $\forall e \in C$ (if $f \rightarrow e$ then $f \in C$)

Consistent iff it corresponds to a consistent cut!!

(L-5) Chandy Lamport Algo



- ensure consistent cut: Proof by contradiction (delivery of marker & a message): FIFO
- ~~channel~~ state?

(Start recording all msg in that channel until received a marker)

↑ (incoming)

Run – a total ordering of events in H that is consistent with each h_i's ordering

Linearisation – run consistent with happens – before ($\rightarrow$) relation in H

It passes through consistent global states

NOTE: A global state S_k reachable from S_i if there is a linearisation that passes through S_i & then through S_k

The d.s evolves as a series of transition between global states $S_0, S_1 \ldots$

(L – 6)

- Liveness: Something good will happen eventually

  Liveness($P(S_0)$) = $\forall L \in$ linearisations from S_0, L passes through S_L & $P(S_L)$ = true

  (S_0: State S_0 satisfy liveness property P)

- Safety: Something bad will never happen

  Safety($P(S_0)$) = $\forall S$ reachable from S_0, $P(S)$ = true

  (State S_0 satisfy the safety property)

- Stable: once true, forever stays true (stable liveness property)

  (if a state is true, all the states following it will remain true)!

  Once false, always false (Stable non-Safety property)

Distributed Debugging: to a monitor, all global state must

Possibly ϕ : There is a consistent global state S through which a linearisation of H passes such that $\phi(S)$ is True

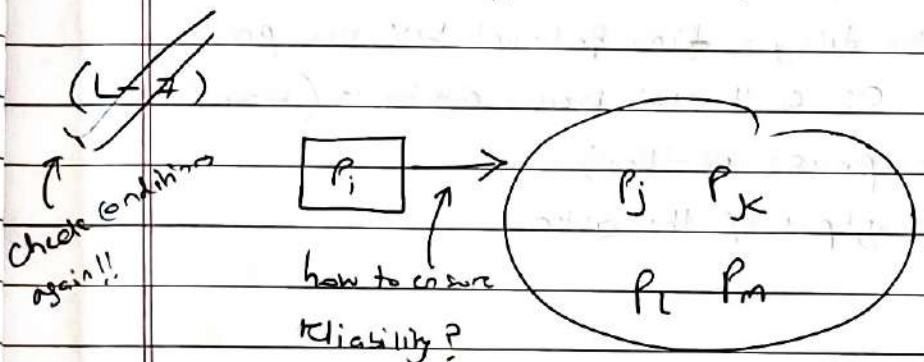
Definitely ϕ: for all linearisations L of H, there is a consistent global state S through which L passes such that $\phi(S)$ is True

NOTE: S is a consistent global state iff

$$V[s_i[i]] \geq V[s_j[i]] \quad \forall i,j = 1, 2, \ldots N$$

NOTE: $S = [s_1, s_2 \ldots s_n]$

This means no. of p_i's events known at p_j when it sent s_j should not be more than no. of events that have occured at p_i when it sent s_i



(Only for multicast it should have happened before relationship)

How to achieve ordering

(i) FIFO

(ii) Causal

FIFO ⇌ Causal

(iii) Total

M1 Reliability { (i) B-Multicast (ii) R-Multicast : • integrity: a correct process p delivers a message m at most once

• validity: If a correct process multicast message m, then it will eventually deliver m itself

• Agreement: if a correct process delivers message m, then all other correct process in group(m) will eventually deliver m

M2 Tree based (multicast) ← MST

(no two replies)

What if sender fails → reconstruct

Overhead

(B Gossip Send to B random targets)

overall liveness

implementation

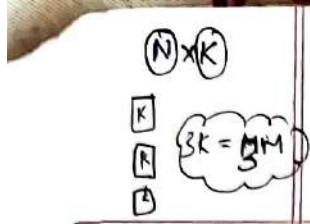
→ Sequence vector

FIFO, Causal → Multicast

Total Order → delivery of message

↳ 2 ways: central server / ISIS

↳ prof



classmate
Date
Page

L-8 Mutual Exclusion — in single OS : Semaphore, Mutex, Conditional Variables, Monitor

Safety ←
liveness ← in DS — central server algo
Ordering ← Ring based "
↳ as in how it was sent (not received) Ricart-agrawala " — Causality & Multicast
Maekawa " — no liveness

Performance Eval metrics

(i) Bandwidth — total no. of messages sent in each enter & exit operation

(ii) Client delay — delay incurred by a process at each enter & exist operation (when no other process is in CS or waiting)

(iii) Synchronization delay — time interval b/t one process exiting the CS & the next process entering it (when there is only one process waiting)

measures the throughput of the system

L-9

Leader Election

→ algorithm should guarantee : safety
liveness

↳ Ring election algo

Bully algo ~~→~~ only to seniors (if no replies declare itself) & wait still if no → rerun!!

↳ Performance Analysis

(i) Bandwidth usage : Total no. of messages sent

(ii) Turnaround time : The no. of serialised message transmission time between the initiation & termination of a single run of the algorithm

consensus impossible in asynchronous system
(so is election!)

classmate
Date
Page

→ = Reliable & Totally Ordered Multicast

(L-10) E11 Consensus = Interactive Consistency Problem = BGt

(for synchronous : soln exist)

(for synchronous system : soln exist)

- Properties: (i) Termination
  (ii) Agreement
  (iii) Integrity

Synchronous: Send to all & wait for $E + (f+1) * T$ for upto f failed process

(after (f+1) rounds, algorithm will stop)

$f \leq \lfloor \frac{N-1}{3} \rfloor$ ⇒ $3f < N$

(Proof by contradiction)

Byzantine Problem

Asynchronous: Paxos, Raft → either safety or liveness (used for deciding a single value)

→ consensus safety

(L-11E) → Proposer + acceptor + learner

What kind of Qs? Paxos — for single value

If multiple ⇒ LogConsensus (Raft)

(L-13) E14 Raft

Objective: Whenever possible, select the alternative that is the easiest to understand

(Re-read or watch some vids)

leader based!

Candidate

follows us

(L-15)

- RPC — Invocation semantics — atleast once, atmost once, maybe once!!
- Marshalling & unmarshalling (as each system have its own format)
  → external data representation

to EDR — from EDR

(to/from) (platform dependent)

(L-16 & 17)

Transactions → ACID properties
- → all or nothing principle
- → Seq of op - by client - transform Server data from one consistent state to another
- → every step is an RPC
- → isolation major issue

option 1: Serial execution {concurrency reduced}
⇓
option 2: serial equivalence : criterion for correct Concurrent execution

i.e $T_1 = \{op_1, op_2, op_3\}$

$T_2 = \{op_4, op_5, op_6\}$

$O = \{op_1, op_2, op_4, op_3, op_5, op_6\}$
↳ is serially equivalent if,
result same as either

$\{op_1, op_2, op_3, op_4, op_5, op_6\}$

or

$\{op_4, op_5, op_6, op_1, op_2, op_3\}$

- Conflicting operations : read & write
  write & read
  write & write
  read & read (X)
  } on same object

in serial equivalent, all the conflicting operations will have an order → i.e either (T_1 & T_2) or (T_2 & T_1)

Ways to achieve concurrency control

pessimistic → a) Locking
optimistic → b) Optimistic CC
c) Timestamp ordering *

- Read → write : must commit! / abort!
- Write → Read/write : must commit / abort!

→ Not serial equivalent ⇒ 2 Phase Locking

NOTE: In read-write locks, read → write promotion happens only if both operations in same transaction!! (otherwise blocked!!)

↓ {gives serial equivalence & locks} (Proof)

2 phase locking → (strict 2 phase includes release locks only at commit point)

(i) growing phase → acquiring / promotion

(ii) Shrinking phase

downside: locking!! → deadlocks

↓ how to handle?

- locks all objects at start
- locks timeout
- deadlocks detection: keep track of wait for graph & find cycles

hit ⇒ breaks the (ii) point

3 ^(necessary) cond^n s: (i) exclusive lock modes
(ii) ~~locks~~ no preemption
(iii) circular wait in wait for graph
((iv) Hold & wait) too!!

(L-18)

Distributed Transaction req coordinator.

↳ how to commit?

→ an issue: • One phase commit - not so effective (how recover?)
• 2 phase commit = coordination → each local server maintains a permanent storage

(phantom deadlocks) ← {how to avoid deadlocks? ← | → as locks not released until commit or abort

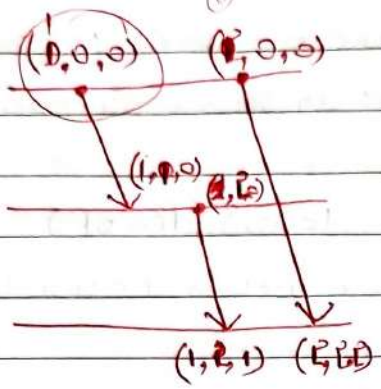
How Recovery?
- recovery manager
- recovery file

→ Centralised detection: global graph
→ decentralised deadlock detection
(: edge chasing
(if a server notices a transaction waiting for another transaction and it waiting for another ⇒ starts by sending probe req)!

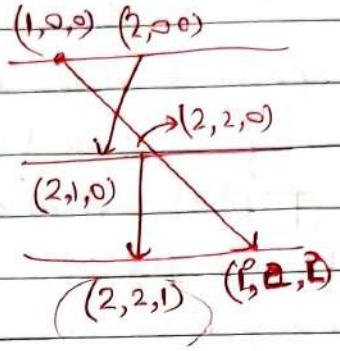
on detection, lowest priority transaction ⇐ {one or more cycles aborted to break the cycle!

Ex.

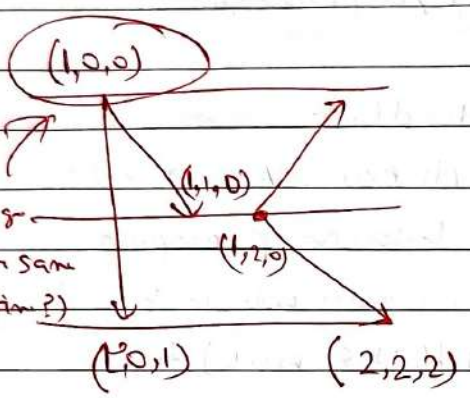
(i)



(ii)

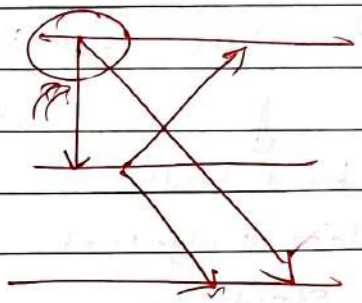


(iii)



Will it go at the same time?

(iv)



(Lec 16 - 17)

- Optimistic Concurrency control
  - a) 3 phases

working → validation → update phase

working	validation	update phase
Stores tentative version (each obj) = transaction	validated to see if any conflicts with other transactions	if no conflict, result/ transactions made is updated

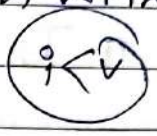
  - b) validation rules

Until commited:
- read - ~~read~~ write
- write - read
- write - write

→ done by backward or forward validations

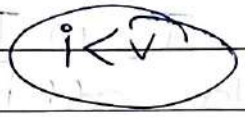
c) Backward validation

Read set of T_v compared with write set of T_i ↖ prev tractsaction



d) Forward validation

Write set of T_v compared with read set of T_i



NOTE: Write - write conflicts don't matter for OCC validation

because they don't cause stale read or break serial order, it's just that the last writer wins.

- Timestamped Ordering

a) each transaction assigned an id

b) If write → check last read & write on that obj is lesser than incoming one

If read → Check last write is lower id

c) Never results in deadlocks

d) ordering rules

write read — Write if ≥ read max timestamp

Write write
Read Write
} Similar conditions

d) T.O : Write rule

if $T_c \geq$ max read T.S on D
& $T_c >$ write timestamp on committed version of D

{ → list of tentative writes sorted by ids(asc)
if T_c in T.W list → update
else add it
}
else
// drop

e) T.O : Read rule

if $T_c >$ write T.S on committed version of D
{
D_s = max write T.S $< T_c$ on D

read D_s
if (D_s committed) → add T_c to RTS list
↓
list of ids that have read committed value

else if D_s written by T_c → read D_s

else wait till D_s committed or aborted then reapply rule
} else {
// abort

(L18-20)

- Distributed Transaction Atomicity == Consensus (→ either all or nothing)
  - requires coordinator : to whom req is fwd
    other servers serve as participants

- It's an asynchronous system !!
  → Req safety property : never have some agreeing to commit and others agreeing to abort

- First cut : One phase commit

Client —commit/abort→ Coordinator —Inform→ (Servers)
(Inform ← X : disadvantage)

→ blocked until a failed coordinator recovers

- Two phase commit Protocol → participants save the state in mem. *
  Phase 1 : Collect all commits
  Phase 2 : If all replies commit ⇒ tells to Commit
  Else sends abort message
  → Coordinator logs all votes & decisions *

how to handle with loses
(i) participant crashes *

(ii) CanCommit? loss : Participant after a timeout eventually says 'No'

(iii) Yes or no loss : aborts transactions Coordinator after some timeout

(iv) do commit loss : After saying yes, it cannot abort by itself → has to Send ~~DoCommit~~ (get Decision) !
→ retry !!

- Phantom deadlock
  → false detection of deadlocks that don't actually exists !!
  ⇓
  as edge chasing messages may have stale data
  → leads to spurious aborts

- Recovery
  → maintains recovery file consisting of object, transaction status, intentions list
  ↓
  list of values & objects altered by that transaction

- Logging
  → contains history of all transactions performed by a server
  → this is filled during commit or abort
  prepares to commit
  ↓
  first ^ transaction
  at that time, objects added in intentionlist
  then commits → status updated

Log → checkpoint : saved snapshot of the database or system at a certain moment
Updates
preparation
Commit
⇓
Why req ?
⇒ else need to replay all log from the start

- After a crash, 2 Ways to recover (by reading log):
  - (i) Forward Recovery (Redo based)
    - → Start at the last checkpoint & move forward in the log that is committed
    - → (like literally apply all the log events)
  - (ii) Backward Recovery (undo based)
    - → start from the end of the log & move backwards
    - → i.e undo committed transactions' updates

- Shadow Versions
  - → make a copy of the data when committing instead of modifying the original
  - → during prepare phase – creates a shadow version & on commit phase – replaces the old map with new one

(Lec 24 + 26)

a) Security Threats
- Leakage – Confidentiality needs to be ensured
- Tampering – Integrity
- Vandalism – DOS – Authority / Availability

b) Security Policy indicates which actions each entity (user, data, service) is allowed or prohibited to take

⇓

Security mechanism implements and enforces the policy

c) Cryptography
- Private/ Shared Key Cryptography
  → how to share key?
  → key length: 128 bits

- Public Key Cryptography
  → key length: 512, 1024 bits etc

d) Encryption
→ Intruder Types
- Passive - only listens to encrypted texts
- Active - can alter messages

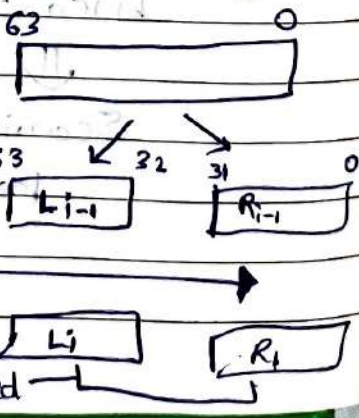
→ Substitution Cipher
DES: Data Encryption Standard
- encrypts data in 64 bit chunks
- Key length: 56 bits (actually 64 - 8 bits for parity)
- it's a 16-round Feistal Cipher - 19 stages of transformations
- each round has its own keys
  ↑ generated from original key: (48 bit key used)

each round of

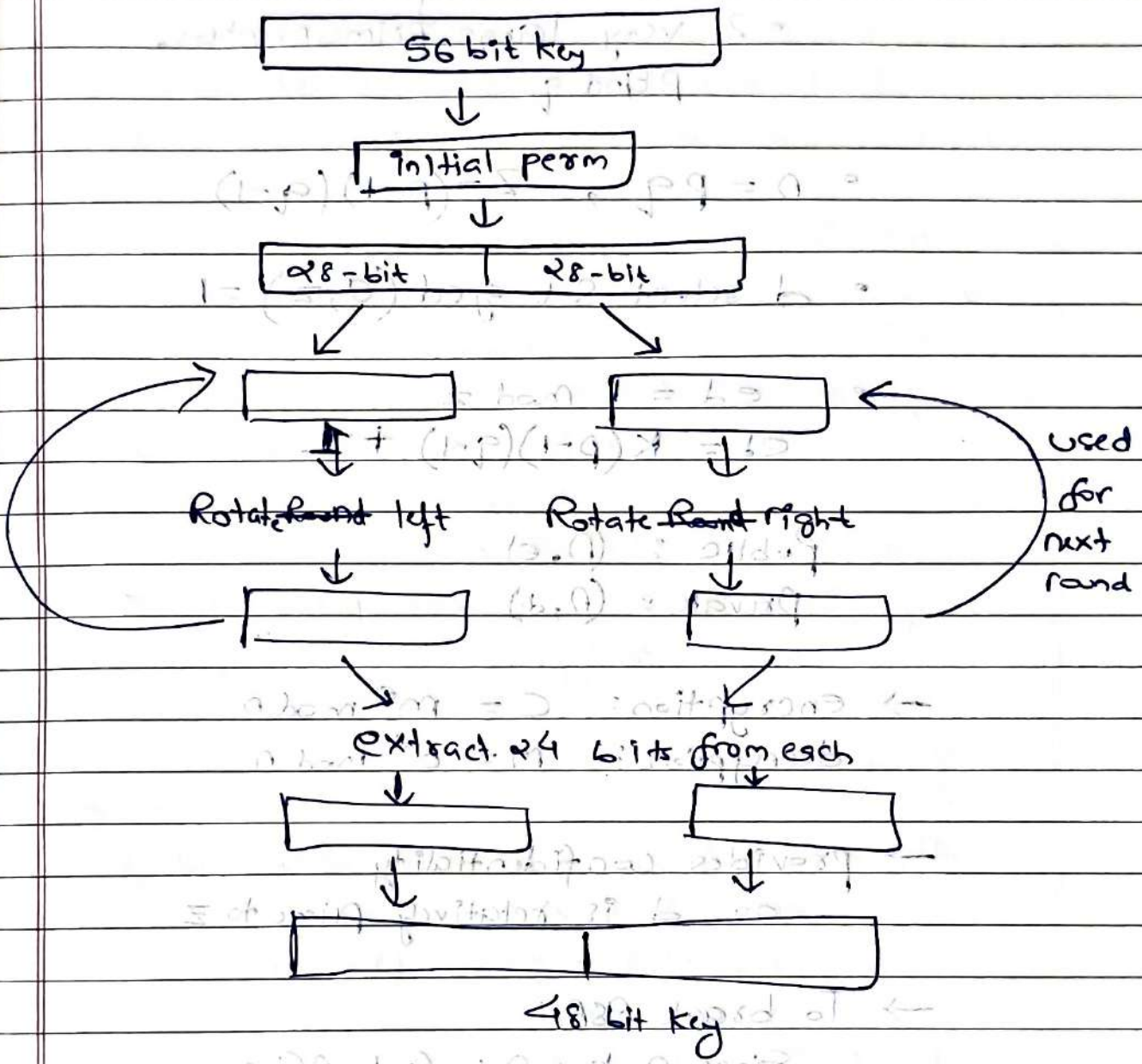
F_i (function given by)

$L_i = R_{i-1}$

$R_i = L_{i-1} \oplus F(R_{i-1}, K_i)$



- Key generation



- provides confidentiality – (56! : Reason)

Under this only

e) Key Distribution Problem

f) Trusted centralized Agent

→ only need to know about trusted agents key

Public Key Cryptosystems : RSA

→ how to generate key

- 2 very large prime numbers, p and q
- $n = pq$, $z = (p-1)(q-1)$
- d select s.t $\gcd(d, z) = 1$
- $ed = 1 \bmod z$

  $ed = k(p-1)(q-1) + 1$

Public : (n, e)

Private : (n, d)

→ encryption : $C = M^e \bmod n$

decryption : $M = C^d \bmod n$

→ provides confidentiality as d is relatively prime to z

→ To break RSA :

Since n known : find p & q

Hash Functions

(1) Secure Digest functions

$h = H(M)$ is SDF provided;

a) Given M, it is easy to compute h

b) Given h, it is hard to compute M

c) Given M, it is hard to find another message M', such that $H(M) = H(M')$

example MD5

input broken into 448 bit blocks

padded to turn it into 512 bit blocks

Message digest size : 128 bits

proves * ← ↓ done through PKC

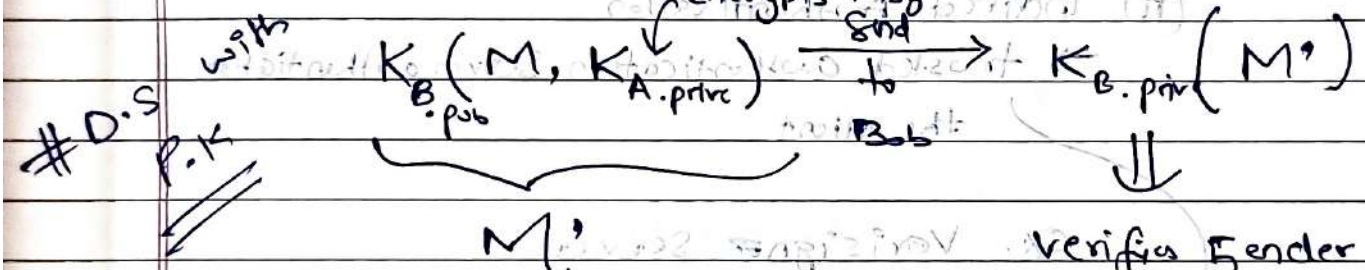
ii) Digital Signature

→ to protect sender receiver from each other

→ Sender encrypts the message using their own private key

and receiver can verify it using public key

NOTE: If confidentiality needed



→ provides data integrity
message authentication } *
non-repudiation

→ generally if Unsecure channel
⇒ PKC

else if secure Channel
⇒ establish the shared secret key
Using hybrid method
⇒ use it to produce low cost signatures
- Message Authentication Code
(a short piece of info generated using Symmetric keys)
provides integrity & message auth but not non-repudiation (!!!)

as signatures ← Shared(!)

as if we can verify a MAC, we can also create it

→ proves identity using a trusted CA

iii) Digital Certificates
→ a document containing a statement (usually short) signed by a principal
→ binds a public key to a real identity

for pt's proof

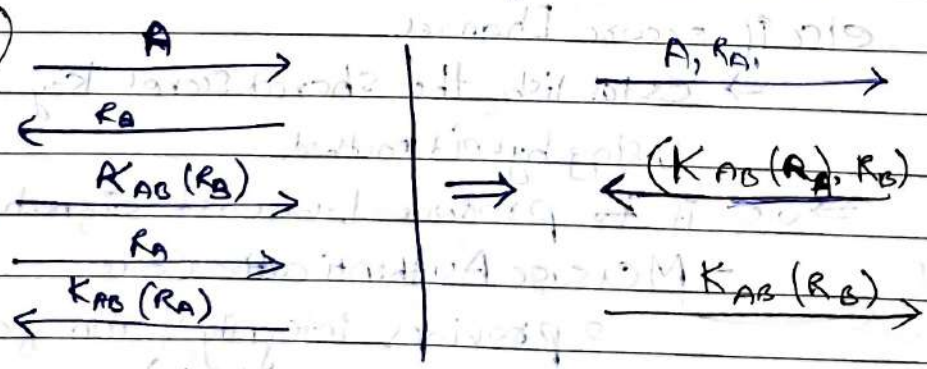
°Authentication
(i) Direct Authentication
→ Server & Clients contact
→
(ii) Indirect Authentication
→ trusted authentication. Server authenticates the client

Ex. Verisign Servers

Solution based on Secret key Cryptography
(*) → 3 Way handshaking
→ Trusted third party (KDC)

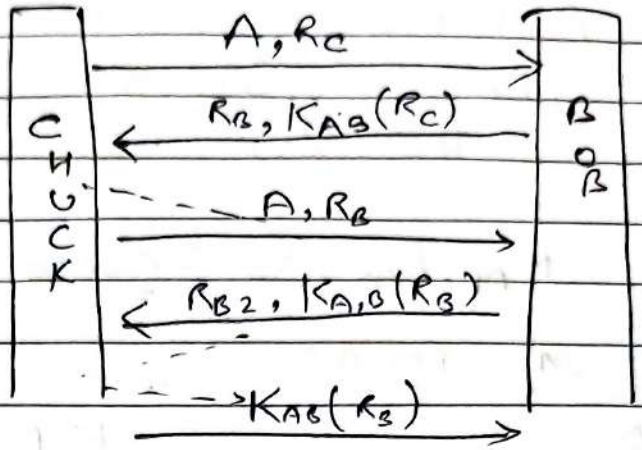
Soln based on Public Key Cryptography
→ Public Key Authentication

(*)



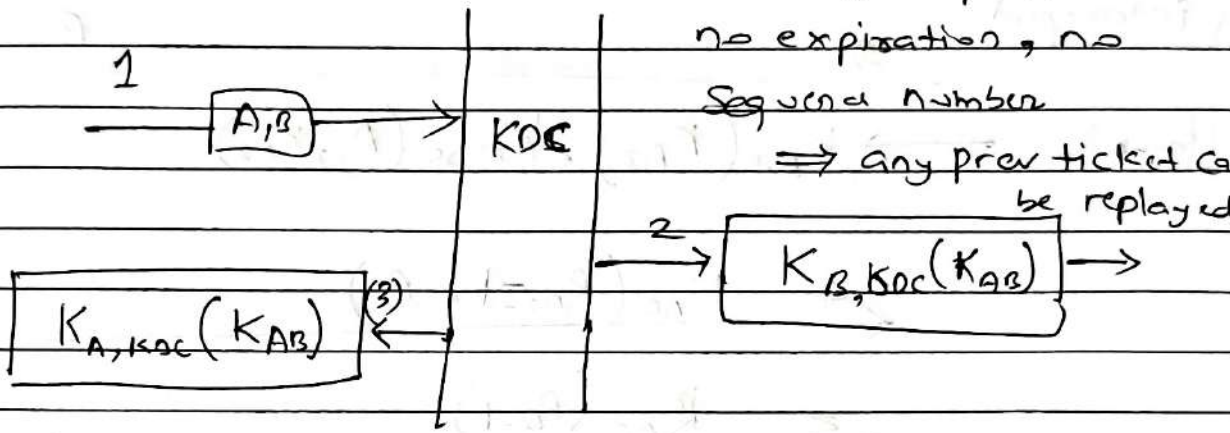
Replay ⟹

Reflection Attack



KDC ^ based Protocol : no timestamps, no nonce, no challenge response, no expiration, no sequence number

⟹ any previous ticket can be replayed



Pb: A may communicate to B even before B got the key from KDC

soln

~~A~~ $K_{A,KDC}(K_{AB})$, ticket $K_{B,KDC}(K_{AB})$

⇒ let Alice maintain setting up a connection to Bob (vulnerable) if $K^{old}_{B,KDC}(K_{AB})$ released

Send to B: $(A, K_{B,KDC}(K_{AB}))$ ⇓ It can establish a session as no verification!

→ for session establishment

Authentication using KDC (Needham Schroeder Protocol)

R_{A1}, R_{A2}, R_B : nonces : purpose of freshness

1) R_{A1}, A, B → KDC

2) ← $K_{A,KDC}(R_{A1}, B, K_{AB}, K_{B,KDC}(A, K_{A,B}))$

if not present similar to prev case without nonce situation

A

if B missing { Intercepted / Knows K_{AB} }

3) → B : $K_{A,B}(R_{A2}), K_{B,KDC}(A, K_{AB})$

4) ← $K_{AB}(R_{A2}-1, R_B)$

5) → $K_{AB}(R_B-1)$

Vulnerable to replay if K_{AB} (released)

Case 1: • If he knows $K_{B,KDC}$ old without R_{A1} present
• then he can get K_{AB}.

Case 2: If B missing → replay attacks can happen

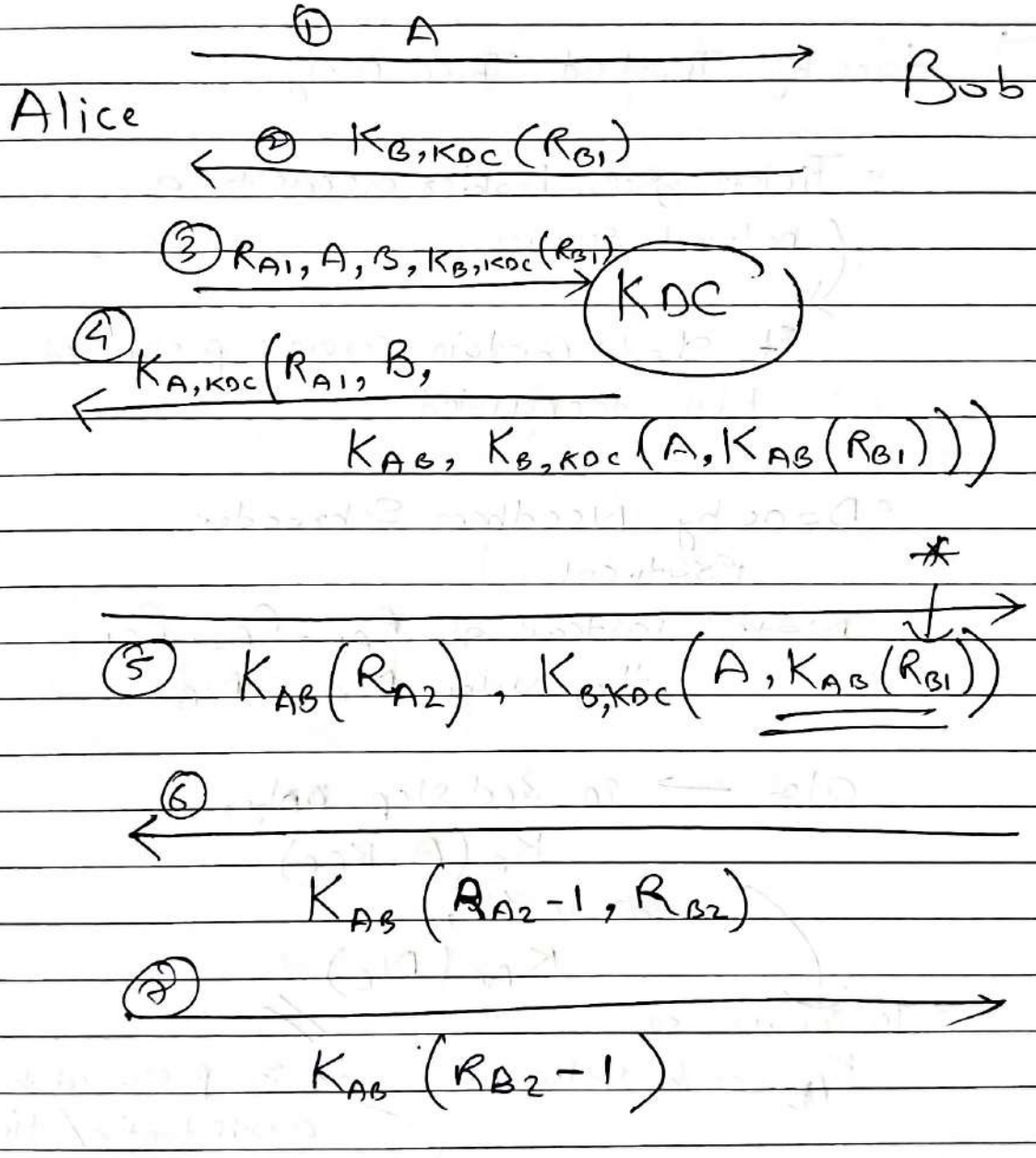
↓

Identifies who he wants to talk

Case 3: K_{AB}^{old} is released

- old message can be replayed
- and session establishment

Defense



Authenticating to Multiple Servers

- Ticket based system
  (else vulnerable)
  ↳ as in if every server knows the password ⇒ one broken, all unsafe

→ done by Trusted Third Party

- Ticket gives holder access to a network service
  → It should contain server's password but encrypted

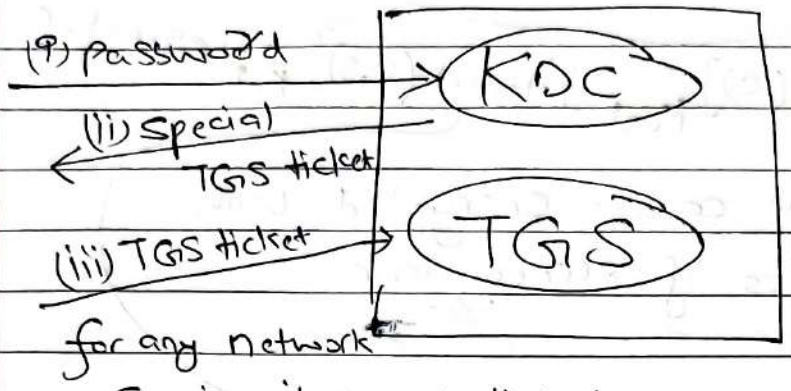
- Done by Needham Schroeder Protocol

Note: instead of R_{A1}, R_{A2}, R_{B1} they using N_A & N_B

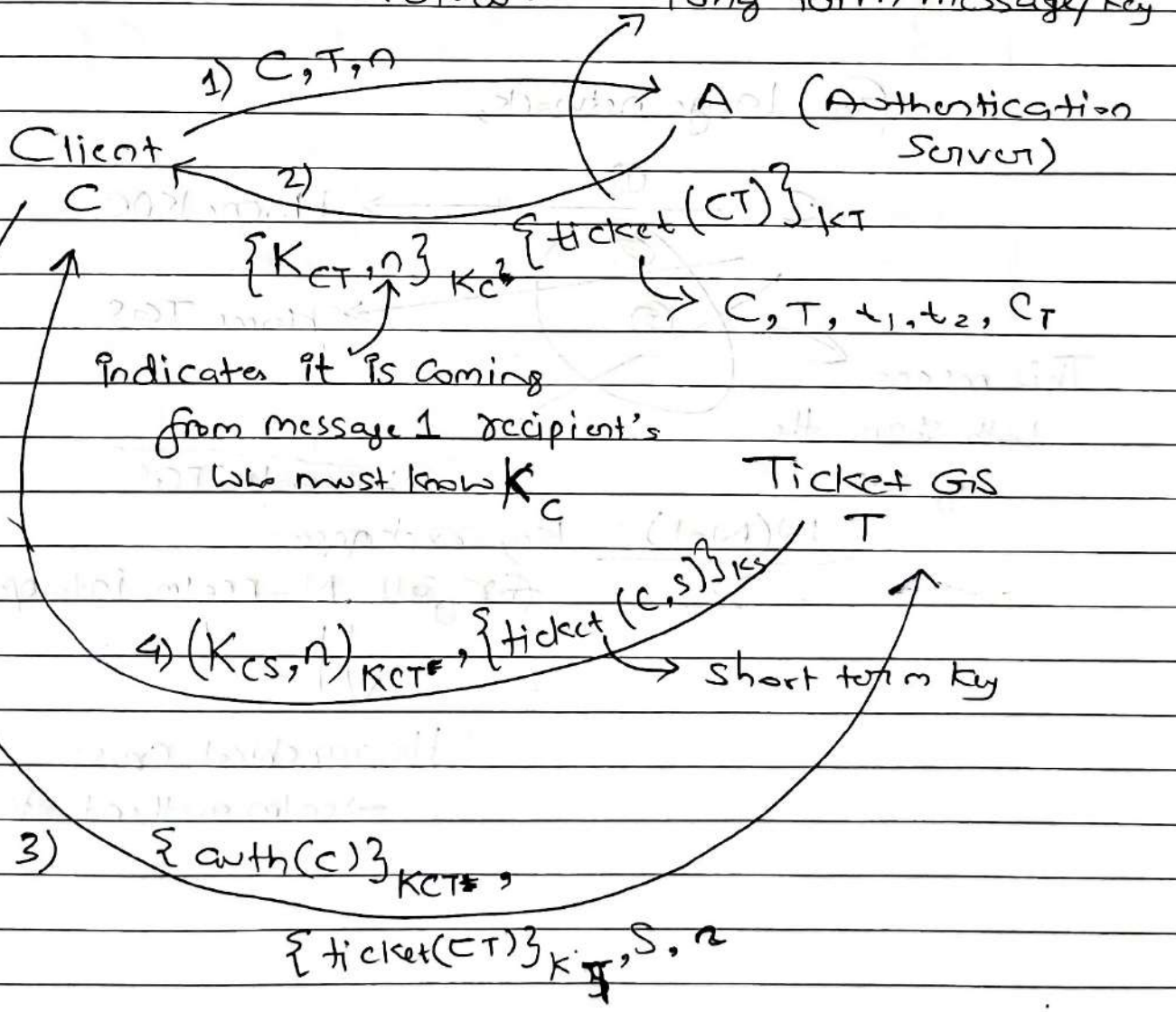
also → in 3rd step only, $K_B(A, K_{AB})$
so in 4th, $K_{AB}(N_B)$ //

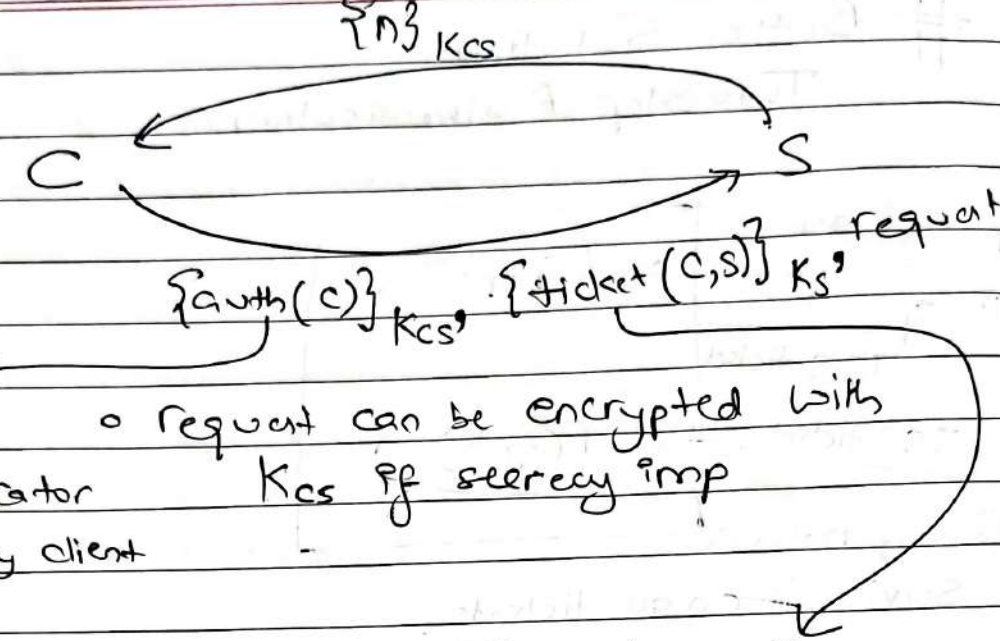
- In secure as K_{AB} can be stolen
- also need to send password each time as in password to authentication/ ticket generator for ticket generation //

Better Solution
Two Step Authentication



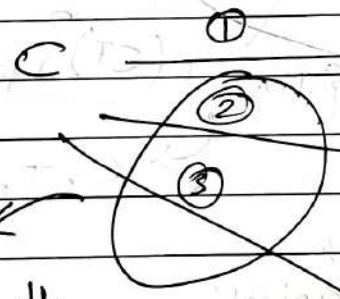
example: Kerberos
Based on Needham Schroeder Protocol





- requat can be encrypted with Kcs if secrecy imp
- authenticator created by client
  C
- prove client liveness (freshness check)
  ↓ prevents replay attacks

for large network,



This means both share the key

$\frac{N(N-1)}{2}$ key exchanges for full N-realm interoperation

↓↓

Hierarchical cross-realm authentication

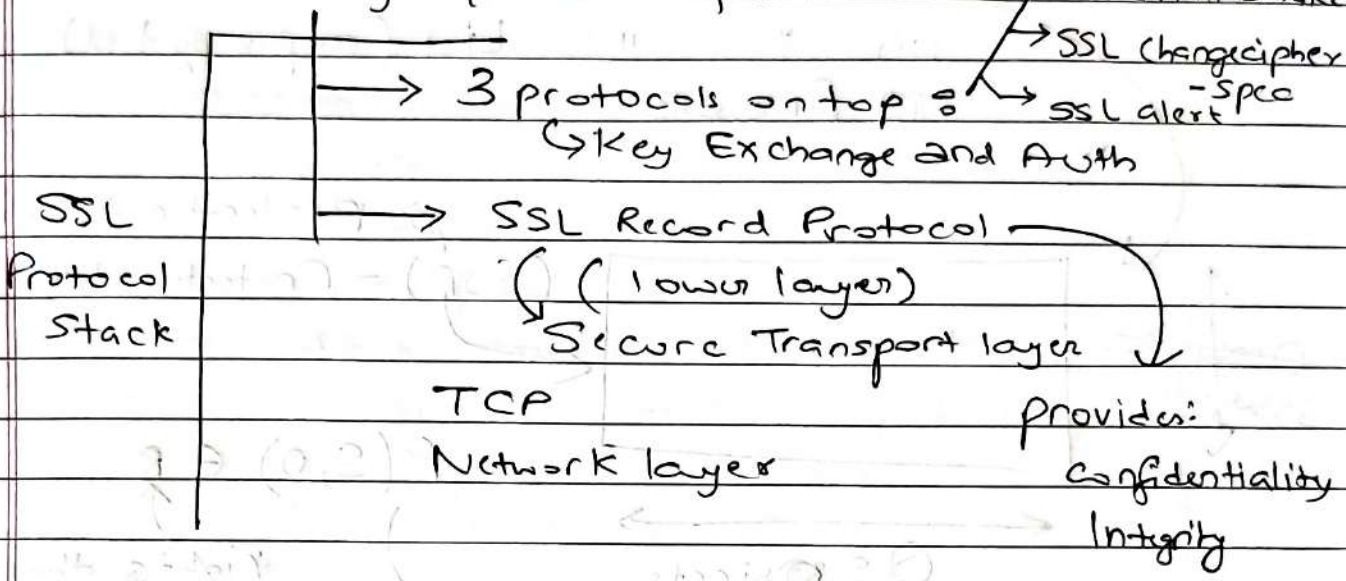
(Video)!

(uses) → not just tcp, SMTP over SSL (465 port no)

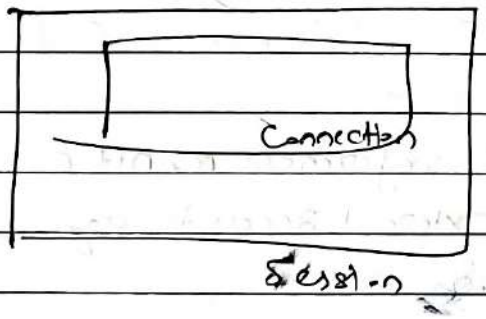
Secure Socket Layers

- layer on top of tcp
- tcp + SSL – authenticated, reliable, end to end, encrypted stream

- 2 layers {consist of 3



- provides confidentiality, integrity and authentication



(# Read one more time for SSL)
Specific to each layer

L24-26: (Slide 86 – 108)

→ User req smtg

⇓

to verify if can access ⇒ do access control

Access Control

→ verifying access rights

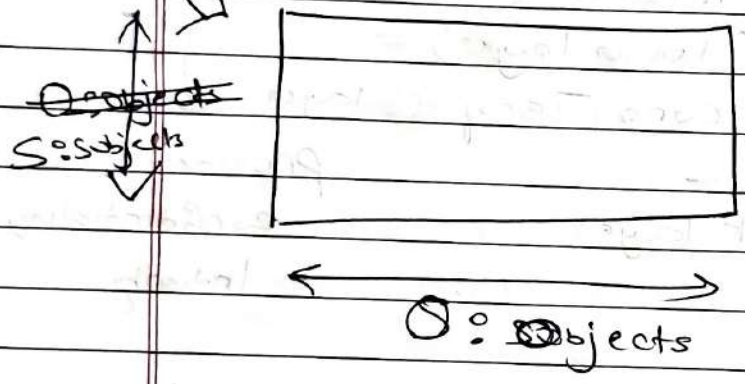
→ granting " " = authorization

→ access control models

(i) Access Control Matrix

(ii) " " List (capability Lists)

(iii) Firewalls



↗ protection system

(SOP) = Protection state

P(S,O) ∈ R

↑ rights that exists

(rights given to subject S to see O)

Firewalls

→ Special kind of reference monitor to control external access to any part of a d.s

→ models: (i) Packet filtering gateway

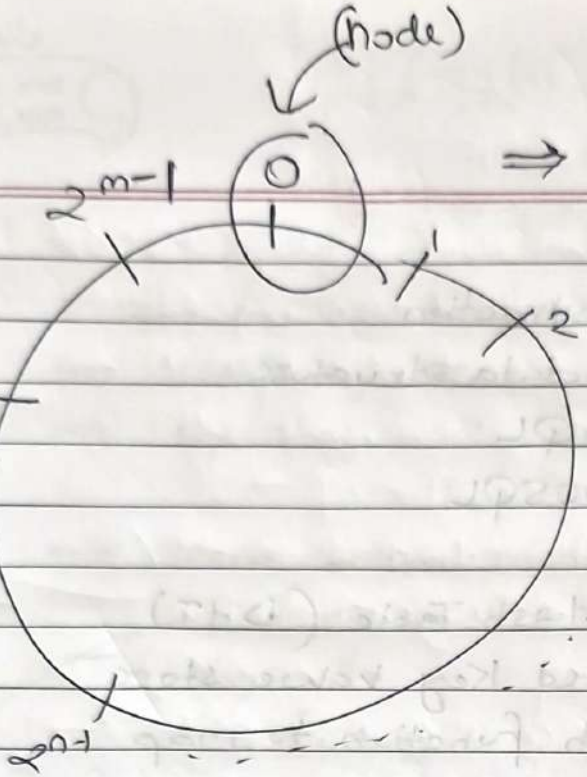
(ii) Proxy

(Lec 27 - 28)

a) Key - value Abstraction
→ a dictionary data structure
→ examples : SQL
NoSQL

b) Distributed Hash Table (DHT)
→ a distributed key value store
→ uses a hash function to map
key → node
↓ ↓
hash(file) value(file)
→ different protocols to implement DHTs
example. Chord, CAN, Pastry,
Tapestry

c) CHORD
→ properties : Load balance
Decentralized
Scalable — Cost (lookup) =
C(log N)
High availability
Flexible naming
(0 to 2^n) { → every node gets an ID } using hashmap
|| key || || ||
→ All IDs lie on circle (ring)
→ a key is stored on the first node clockwise from the key's ID → such that it's ≥ K //
↓
i.e called successor of key K

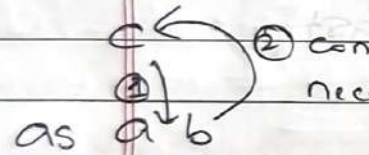


⇒ NOTE: m bit is the dp of the address space in ring

- find the id which $\geq k$ i.e successor of key k ($\because k \% 2^m$)
- if none present, go to smallest ID available

⇒ This is Consistent hashing

⇒ Useful: When a node enters or exists only $\frac{1}{n}(k)$ of them shifts (n = no of nodes)



NOTE: Key node 2^m stores the data we want to store

Note: 2^m possible locations for storing a node but only certain locations actually have a server (called node) !!

- That's why we search for successor(key) to store the value

→ helps in load balancing

→ Consistent Hashing

With K keys and N peers, each peer stores $O(K/N)$ keys (i.e $< c \cdot K/N$ for some constant c)

How to perform lookups

(i) $O(n)$ (T.C) + $O(1)$ S.C

(A)
Successor ← initial value (current node/ queried node)

i. Locate Successor (key) where key ≠ i :
   if (key ∈ (i, successor] then
      return (successor)
   * else return successor. Locate - successor(key)

(ii) Finger Table

- each node aware of ≈ m other nodes
- maintains a finger table with m entries

((

ith entry of node n's finger table =
successor $(n + 2^i)$

i ranges from 0 to m-1

Algorithm for look up

⇒ (A) remains same except (*) part gets updated:
else forward to the largest finger entry ≤ k

NOTE: Query is for key k !!

Total cost = $O(\log n)$

optimize routing

NOTE: finger table entries used for both lookup, insertion, deletion etc as long as the finger table & successor correct!

Also each nodes stores a successor list (> size r (= 2 log N) (nearest r successors) preserve ring successor

Pb of all nodes dying $\left(\frac{1}{2}\right)^r$

⇒ atleast 1 alive: $1-\left(\frac{1}{2}\right)^r$

⇒ ring atleast half alive $\left(1-\left(\frac{1}{2}\right)^r\right)^{N/2}$!!

Failure Recovery (for lookups)

NOTE: Successor list helps here by replacing it with first live entry in the list

Stabilize will correct finger tables and successor list entries pointing to failed node

Storing keys

NOTE: just not the successors stored we replicate key value pair at r successor & r predecessors!

New nodes joining!

→ Simple: add the node & then update finger tables

→ no. of finger tables affected due to new node addition = log N

→ No. of messages per node join (to initialise the new node's finger table)

$= O(\log(N) * \log(N))$

$\log(N)$ ↑ no. of entries; $\log(N)$ ↑ each req lookup

Concurrent Joins

i) updating every finger table perfectly is expensive

ii) Chord lookup — doesn't require every finger table to be correct

iii) Req only 2 invariants: → Successor(n)

Invariant 1: each node must have the successor pointer — i.e first entry in the finger table

Invariant 2: The correct successor of each key k must exist somewhere on the ring

→ Successor(k)

Stabilization Protocol

a) When a node n joins

next(n) = successor(n)

notify(next(n))

} joining node notifies its successor

b) When a node gets notified: n ← notify(n')

Node updates its predecessor when notified: If prev(n) is nil or n' lies bet^n prev(n) & n: prev(n) = n'

c) Stabilize()
→ Sets up the Successor
→ It is run by every node periodically

$x = prev(next(n))$
If x in $(n, next(n))$ then $next(n) = x$
notify next(n)

→ also each node n periodically updates a random finger entry

pick a random i in $[0, m-1]$
Lookup successor $(n + 2^i)$

Lec - MapReduce

How to run large scale distributed computations over key-value stores

⇒ Map Reduce Programming Abstraction

Key value Stores in Cloud Computing → is delivery of computing resources - like servers, databases, networking, software & analytics - over the Internet on demand usually on a pay as you go basis!

→ delivery models: SaaS, PaaS, IaaS

deployment models: public cloud
private "
Community "
hybrid "

→ Characteristics: On demand
Scalable

• (dynamic acquiring - elastic computing (and storage services) of compute resources)

→ What is a cloud?

Cloud = Lots of storage + Compute cycles nearby

↓

Services Include,

(i) managed clusters for distributed computing

(ii) managed distributed datastores

3 Tier architecture

(i) frontend

(ii) compute nodes

(iii) storage nodes

→ How to do processing on large datasets?

⇓

- distribute over multiple machines
- parallel computation

⇓

Map Reduce : it is an abstraction (a model) for doing large-scale computation easily in the cloud

Input : a set of key/value pairs

User supplies two functions:

(i) map(K,V) → list (K1,V1) — Intermediate key/value pairs

(ii) reduce (K1, list(V1)) → V2

↑ Output is the set of (K1,V2) pairs

Example:

```
map(key, value):
    // Key: metadata record #
    // value: (URL, size, ---)

    for each (URL, size) in value:
        emit(URL, size)

reduce(key, values):
    // Key: target
    // values: list of pages that link to it
    result = concatenate(values)
    emit(key, result)
```

- NOTE: Resource Manager (assigns map and reduce tasks to servers)
  - Example: Yarn, Kubernetes
  - → Treats each server as a collection of Containers
    - fixed CPU and memory (ex. Dockers)
  - → has 3 main components
    - (i) Global Resource Manager (RM)
      - → cluster scheduling
      - →
    - (ii) Per server Node manager (NM)
      - → daemon and Server specific functions

(iii) Per application (job) Application Master (AM)
→ container negotiation with RM and NMs
→ handling task failures of that job

- Fault Tolerance

(i) If NM fails ← (NM heartbeats RM)
→ it stops sending heartbeats to RM
→ RM informs AM

(ii) If AM fails ← (AM heart beats to RM)
→ RM restarts AM

(iii) If RM fails → use old checkpoints and bring up secondary RM

- Slow Servers

→ aka Stragglers
→ delay the entire job : as all maps must finish before Reduce tasks can start

→ Reason : (i) bad disk
(ii) low memory
(iii) low CPU
(iv) poor network
(v) temporary overload

→ How to handle?
- Uses speculative execution (MapReduce)

↓

If one task is too slow :
(1) run another copy of the same task on a different machine

ii) Whichever copy finishes first is accepted

iii) The other copy is killed

- Task Scheduling

→ try to schedule in the server where it is available

→ If not possible, in same rack
if not " , anywhere

⇒ otherwise, just delay !!

L – Peer to Peer

- P2P systems

→ refer to applications that take advantage of resources (Storage, cycles, content, ---) available at the end systems of the internet

- Overlay networks

→ refer to networks that are constructed on top of another network (eg. IP)

- P2P overlay networks

→ any overlay network that is constructed by the internet peers in the application layer on top of the IP network

→ Properties : (i) self organizing
(ii) load balancing
(iii) dealing with free riders

Lookup issue

Types of overlay networks

(i) Structured overlay networks — DHT

(ii) unstructured overlay networks

(iii) overlay multicast networks

Structured V/s	Unstructured (Join network with loose rules)
(i) Specific structure	(i) no specific structure — for data/placement
(ii) insertion/deletion have some overhead	(ii) no overhead in terms of both
(iii) fast lookup	(iii) lookup has high overhead
(iv) Complex queries difficult to support	(iv) Complex queries supported like keyword, range & attribute queries

- done through flooding or random walks

⇒ example. centralized directory based P2P systems, Pure P2P systems, hybrid P2P systems

Centralized Directory based P2P Systems

a) all peers connected to central entity

⇓

→ Central entity Serves as:
(i) to provide the service
(ii) Some kind of Index/group database
(iii) lookup/routing table

ex. Napster, Bittorrent

⇓

→ Single point of failure
limited scalability (server farms with load balancing)

→ Query is fast ⟹ upper bound for duration can be given

b) Gnutella
→ Pure P2P
→ no routing intelligence
→ Constrained broadcast
(i) TTL typically set to 7
ii) Packets have unique ids to detect loops

NOTE: fully distributed
→ edge : not a physical link

Scalability : (bandwidth ↑ wasted)
limited Scope flooding
→ Query flooding – fwd Query msg
Query hit sent over reverse path

(i)

Flooding ⟹ Random Walk

→ Scale free graph

- a network where
  - (i) a few nodes have very high degree (many connections)
  - (ii) most nodes have very low degree

} follows power law distribution

→ Random walk

(i) Choose a ~~node~~/neighbor at random

(ii) Move to it

(iii) Repeat

(iv) avoid immediately going back to the last node (to prevent loops)

Spreads the search through the network without flooding

→ Degree biased Random walk

(i) pick the neighbour with the highest degree that hasn't been visited yet

(ii) Replication

→ Spread copies of objects to peers : more popular objects can be found easier

→ replication strategies

a) Owner replication

b) path "

c) random "

(iii) Random Walkers
→ don't send to all neighbours
→ only to a subset of its neighbours

(iv) Unstructured Informed Searches
a) informed networks
- nodes do smarter routing by using info they already learned

- 2 types
→ Local indices
a) each node keeps info abt files within r hops
b) When querying it asks neighbours just outside r
c) This reduces flooding because we search only a local zone

→ Routing Indices
a) Each node keeps an index of
- what kinds of files each neighbor likely has
- how many matching files each neighbour has for a given theme

b) each neighbor gets a goodness score

c) The node forwards the query to the best neighbour

(V) Free Riding — Problem in P2P networks

→ Free riders are nodes that
- download but do not share
- or share useless/empty data

P2P Systems → even rare obj are found efficiently!

Structured P2P Systems

a) DHT

→ Hash table functionality in a P2P network: lookup of data indexed by keys

ex. routed queries done by Chord, Pastry etc

b) Routing Challenges

→ define a useful key nearness metric
→ Keep the hop count small
→ " " tables "
→ Stay robust despite rapid change

Freenet — Emphasizes anonymity
Chord — " efficiency and simplicity

c) Pastry → accounts hop count + network locality
- for routing efficiency

→ Self organizing overlay network of nodes
→ Pastry takes into 'account network locality'
↳ eg IP routing hops

→ no ring like structure : routing based on numeric closeness of identifiers

→ Identifier space
- $(0 \text{ to } 2^m - 1)$: m bit address
- but each digit in ID is (2^b) base

- keys are stored on the node whose ID is numerically closest to the key

$$\text{distance} = |\text{nodeID} - \text{KeyID}|$$

→ Routing Goal
- takes $\lceil \log_{2^b}(N) \rceil$ steps !! (for finding the node with the key)

- each node has 3 main elements
  (i) routing table (R)
  - similar to Chord's finger table
  - stores links to id-space
  - $\frac{m}{b}\left(\log_{2^b}(N)\right)$ ← rows
  - $2^b - 1$ entries per row !!
  (i.e $2^b - 1$ columns to represent the (digits))
  (ii) leafset
  - stores nodes close in id-space
  (iii) neighbour hood set
  - nodes that are closed together in terms of network locality

- each row corresponds to a prefix length
- each col stores nodes with the same prefix but differing next digit → i.e prefix length == row no then column should differ in row no index

(Only 1 node exists !! per rows)

→ Leaf Set L

- It keeps |L| nodes numerically closest to n (|L|/2 Smaller and |L|/2 larger)

- imp info for routing : acts as chord's successor list in case of node failure

→ Neighbourhood Set (M)

- concerned with nodes that are close to the current node with regard to the network proximity metric

↓

i.e how close they are in real network

Routing Rules → When node A receive msg for key D

a) if D is in A's leaf set range → use the leaf set

b) else → use routing table

- Compute L = shared prefix length (A, D)
- Then find to :
  - (i) routing table entry in row L
  - (ii) Column = digit of D at position L
    - → if it exists
- else
  - (i) Choose any node from (R ∪ M ∪ L)
  - (ii) with equal or longer prefix match
  - (iii) and numerically closer to D than A

↓

to prevent routing loops

* better as query routed to a node with a longer prefix than current node or to a numerically closer with same prefix!

→ Routing Performance

Case 1: R.T

- search space reduces by 2^b factor at least
- destination reached in $\log_{2^b} N$ steps

Case 2: L.S

- no. of hop += 1

Case 3: T V L U R

- no. of hop += 1

⟹ Complexity of routing $= O(\log_{2^b} N)$

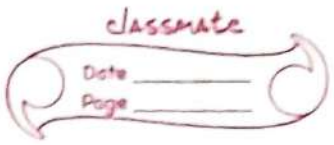
Join and failure

↓

- find numerically closest by routing
- build its 3 table based on the entries of nodes on the route !!

→ failed leaf node
: contact a leaf node on the side of the failed node and add appropriate new neighbour

failed table entry
: contact a live entry with same prefix as failed one
Or keep trying with longer prefix table entries



Self Organisation

- a new node knows another node K as closer to it via proximity

$\Rightarrow$ Consider it's neighbouring set as their own!

- then via (K) it finds a numerically closer node using routing $\xrightarrow{\text{with key}=n}$ takes it's leaf set as it's own

- next, for R.T it takes routing info from the nodes visited in that routing process!

Lazy Detection of Node Failures

$\Downarrow$

$\rightarrow$ as failure detected when a communication attempt with another node fails

$\downarrow$

and since routing involved $\Rightarrow$ lazy detection

Node Departure

- ~~asks~~ When node departs:
  - (i) remaining nodes need to repair their neighbourhood set M
  - (ii) they do this by asking other nodes for their M

Locality

$\rightarrow$ It means that the routing algo tries to use nodes that are physically close in the real network, not just close in the id space!

Locality in Routing Table
New nodes copy routing tables from nearby nodes →
good entries propagate

Route Locality
→ Each hop gets closer to target Id is physically near → no loops, efficient path

Nearest among k nodes
→ Switch to numeric closeness when near target to pick physically closest replica

Extra Notes

a) GFS (Google File System)

- DFS designed by Google — for large scale data processing
- optimized for huge files, high throughput and fault tolerance
- Uses a single master service that stores metadata
- files — Split 64 MB chunks stored on Chunkservers
- each chunk replicated (usually 3 times) for fault tolerance

- designed for write once, read many workloads
- ensures reliability through heartbeat messages, chunk versioning and automatic re-replication

b) Andrew File System

- DFS developed at CMU widely used in universities
- focuses on scalability and efficient file sharing across many clients
- uses whole file caching on clients → entire file copied to client machine
- reduces server load because most operations happen on cached copy
- uses callbacks : server tells clients when cached file is invalid → strong consistency
- file space organized into volumes, which can be moved easily for load balancing
- Authentication via Kerberos, supports secure access

- GHS
  - → distributed version of Prim's algo
  - → nodes form fragments & repeatedly find their least-weight-outgoing-edge (LWOE)
  - → fragments absorb or merge based on their levels, eventually forming the MST

  - → each fragment has a level
    - 2 fragments of same level merge → new level = L+1
    - lower level fragment absorbed into higher level fragment

  - → level tracks fragment size & controls merging

  - → Purpose of test/accept/reject messages
    To decide if an edge is an outgoing edge
    a) test - is this edge connected to a different fragment?
    b) accept - yes, outgoing
    c) reject - not, internal edge

  - → Uses broadcast & convergecast
    To ensure allow all nodes in a fragment to report their ~~that~~ candidate edges to the root so the fragment can find its LWOE

  - → branch and rejected edges
    - branch ↓ part of edges
    - rejected ↓ internal edge that cannot be used for MST (once rejected, permanent)

- Replication
  → Why ? availability
  reliability
  performance
  functioning even when nodes fail
  → diff bet linearizability and sequential consistency

linearizability	sequential consistency
↓	↓
realtime order is preserved	only program order preserved
Strongest practical model	real time may differ

Linearizability ⇒ Sequential
⇍

→ eventual consistency :
- a weak model where replicas may be temporarily inconsistent, but eventualy converge if no new updates occur
- used in large scale systems
- example: gossip, Dynamo

→ Passive (primary backup)
Primary executes operations, sends updates to backups; implements linearizability

→ Active
All replicas execute operations in same order via total order multicast; gives sequential consistency !!

→ Client centric consistency model

- consistency guarantees from the perspective of a single client
  - read after write
  - monotonic read
  - monotonic write
  - write follow read

- Self Stabilization
  → Self stabilizing system
  Start from arbitrary state
  Converges to a legal state in finite time & stays there

→ 2 properties
  (i) Closure: from a legal state, system stays legal
  (ii) convergence: from any state, system eventually reaches a legal state

→ Djikstra's ~~token ring~~ token ring is a self stabilizing system
  nodes arranged in a ring with states
  exactly one privileged node should exist (the token)
  rules ensure that regardless of initial state, eventually exactly one token exists & circulates — achieving self stabilization

→ Why an exceptional node in Djikstra's token ring ?

If all nodes identical, symmetry prevents system from ensuring a unique privileged node

one node must behave differently to break Symmetry

→ What type of faults handled by self-stabilization

Transient faults (like memory corruption) not permanent failures

- Distributed File Systems

→ a remote file storage system that allows clients to access files transparently over a network as if they were local

→ transparency requirements satisfied

(i) access transparency
(ii) location "
(iii) migration "
(iv) replication "
(v) performance & scaling "

→ NFS – Client-server model
RPC based
Stateless in V3
Stateful in V4

GFS – cluster based DFS

→ Caching improves performance by reducing remote access
Introduces consistency problems since multi cached copies may differ

→ file striping — (PVFS) (object based)
Striping a file into multiple blocks and storing them across multiple servers to establish parallel access