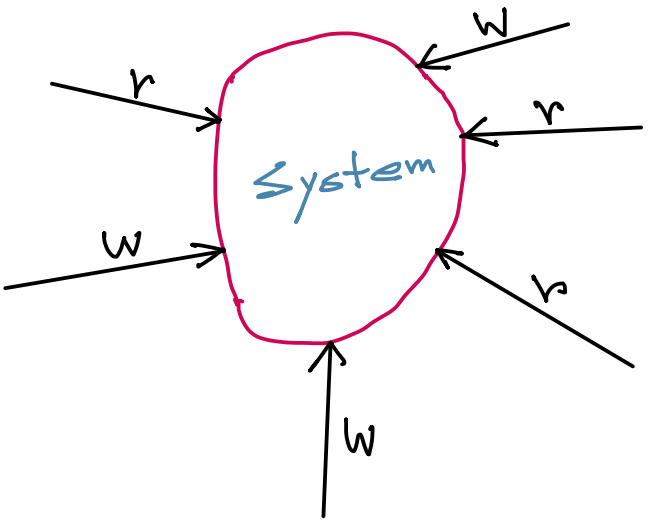
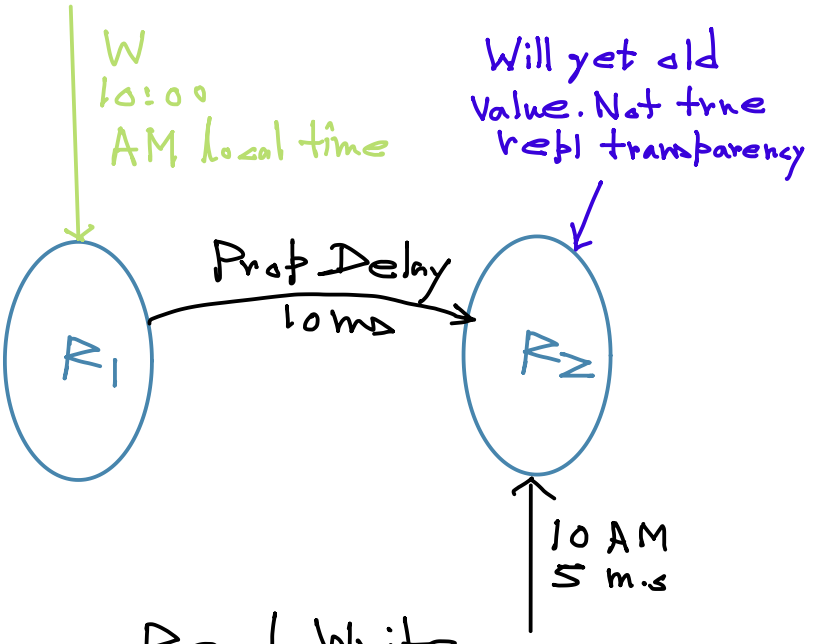
"single copy" view = reads get latest write





Consistency Model:-

Guarantee given by system on Read-Write operation behaviour on shared data.

Model only tells what will happen not how it will happen.

Data Centric Consistency Models:-

- Linearizability
- Sequential Consistency
- Eventual Consistency.

"Data Centric" ⇒ focus on what is happening to data irrespective of who is doing that.

Client-Centric Consistency Models:-

Focus is on one client, What happens on it's R & W

Motivate - Shared data that is usually updated by one entity or even if updated by many rarely, write conflicts can be resolved by some other mechanism.

Client = C

read after read consistency

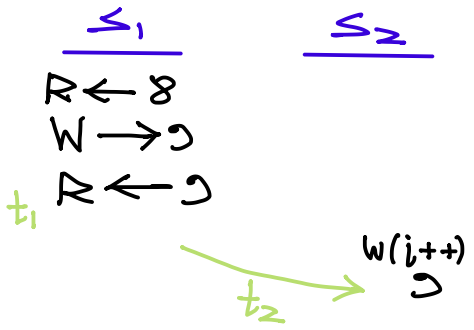
- If C reads from a server S_1 at time t_1 & then reads from a server S_2 at time $t_2 > t_1$, C should get a value that is at least as old as or more recent than the value read at t_1

Read after Write Consistency:-

If C writes to S_1 at t_1, & reads from S_2 at t_2, C should get at least the value written at S_1.

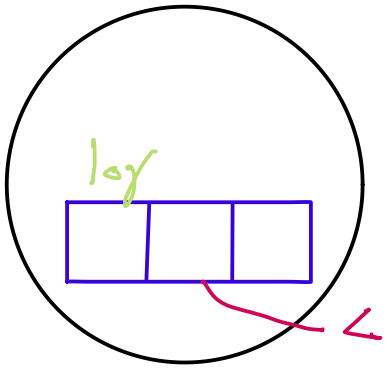
Write after Read Consistency:-

Write after Write Consistency:-



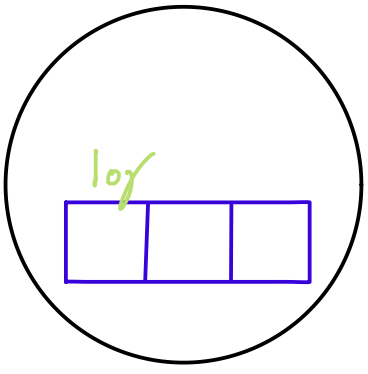
Q) Can we use ZPC to do linearizability?

State Machine Replication



So, goal is for all n-f nodes, the element in the i-th position of their logs will be the same.

Client Command/ Request



How to do?

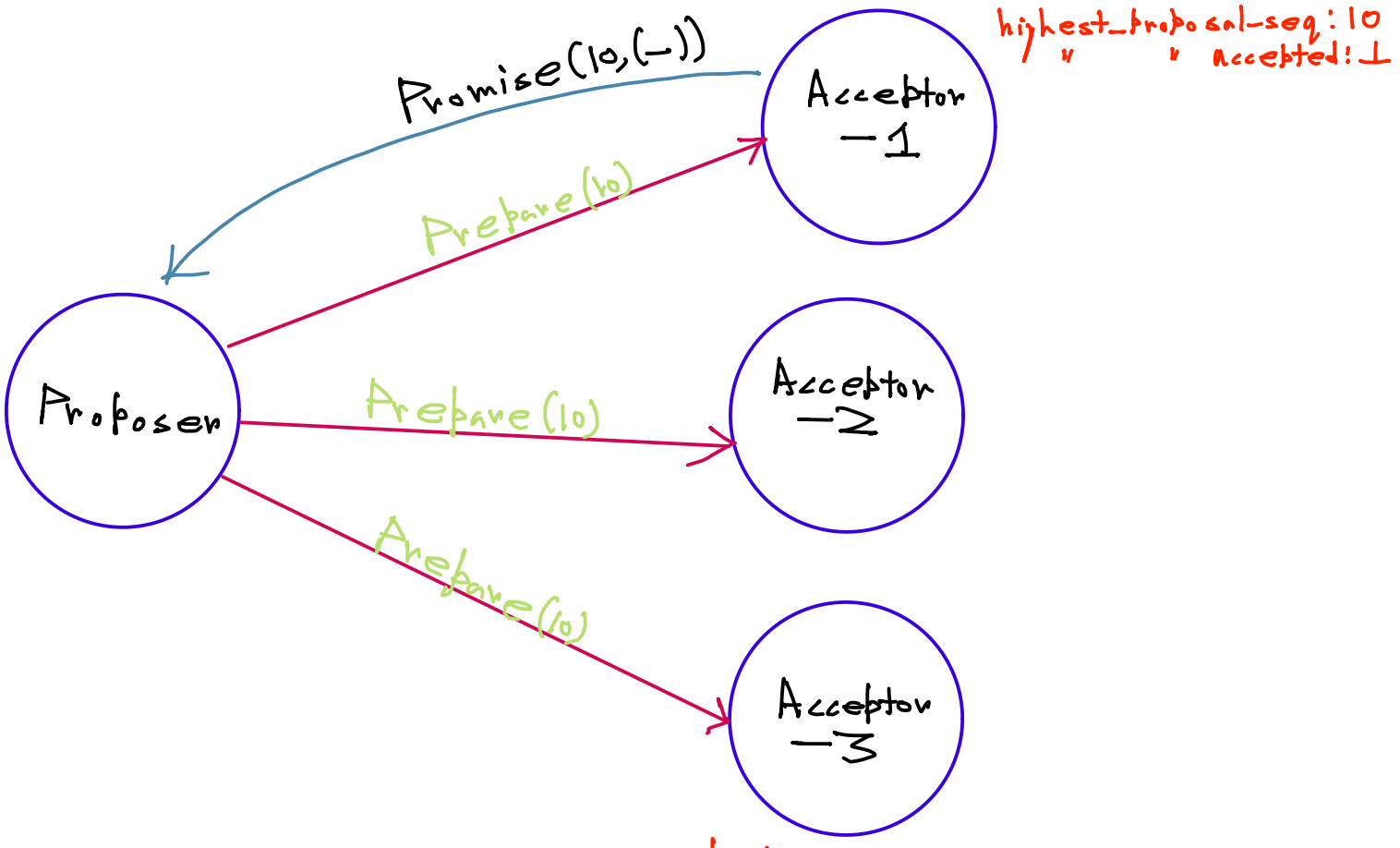
For every Client command, Run Consensus on which position in the log it should get in.

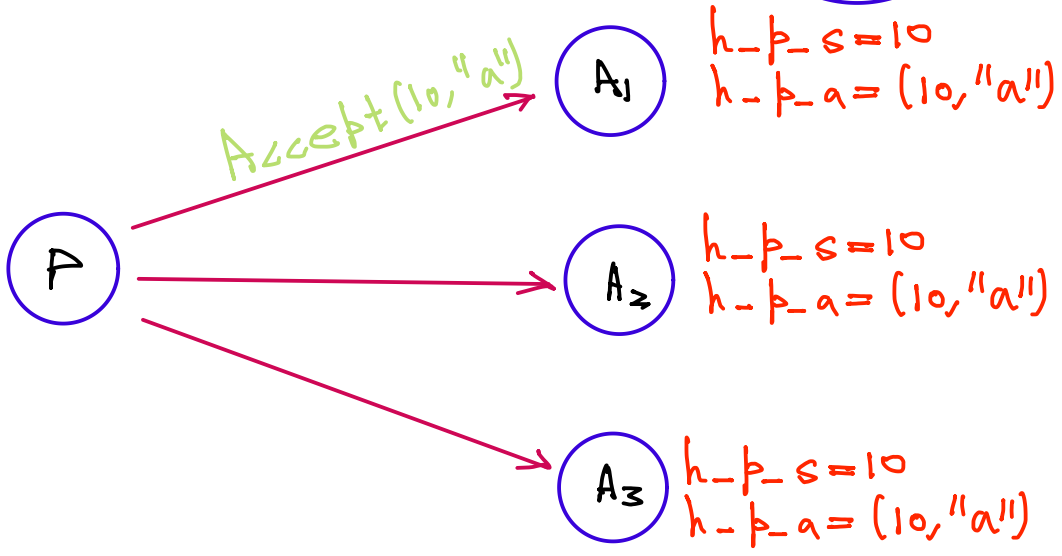
Paxos

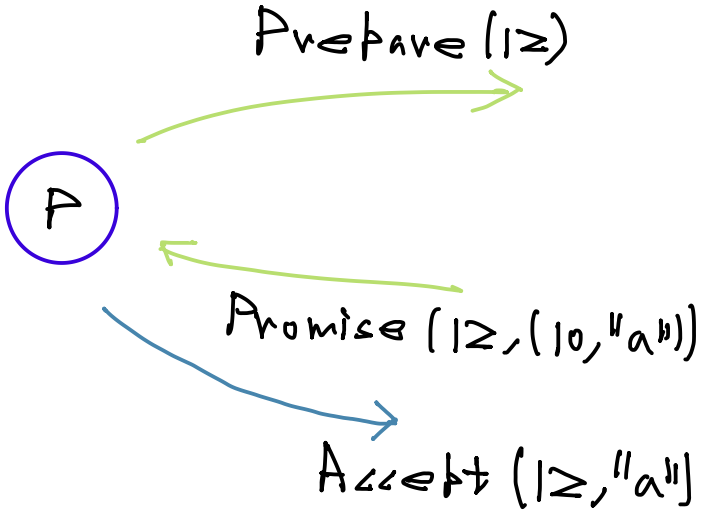
Proposal — <Prop No, Value>

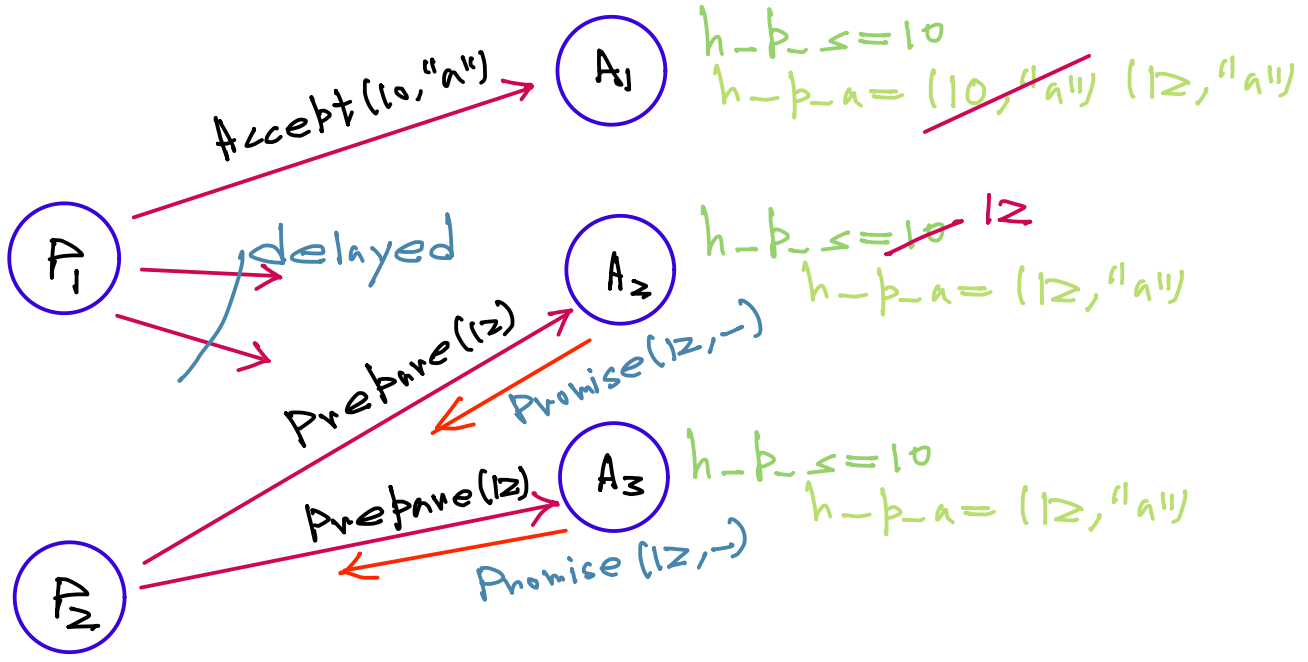
Value that consensus is needed on

Unique across the system (also not worry how)


highest_proposal_seq: 10
" " accepted: ⊥
Promise(10,(–))
Acceptor –1
Prepare(10)
Proposer
Prepare(10)
Acceptor –2
Prepare(10)
Acceptor –3


Accept(10,"a")
A1
h_p_s=10
h_p_a=(10,"a")
P
A2
h_p_s=10
h_p_a=(10,"a")
A3
h_p_s=10
h_p_a=(10,"a")


Prepare(12)
P
Promise(12,(10,"a"))
Accept(12,"a")



At the begining:-

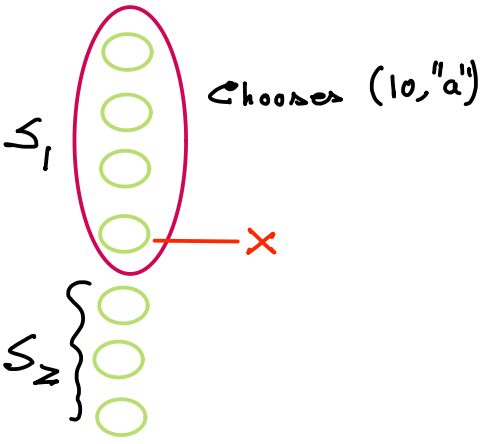
highest_proposed_accepted = $(\perp, \perp)$ for all

Think of the last value choosen majority has $(n, "a")$

S_1 = choose "a"
S_2 = say $\perp$

& $n > \perp$

Acceptors

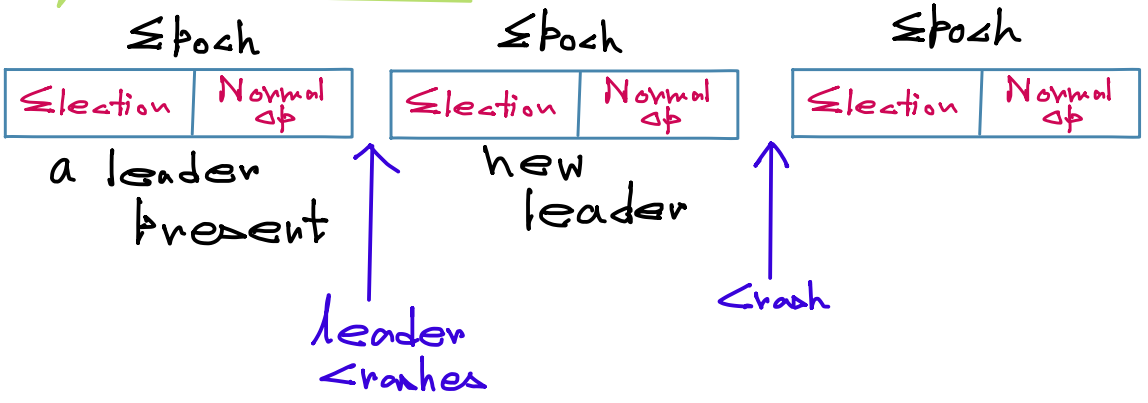


Multi Paxos:- Paxos round for each log position

Raft

Two main tasks
1. Leader Slection
2. log replication

Typical Operations



Term:- Name for each epoch:-

Term 1, Term 2, ...

Log:-

Term No			
Command			

Diff b/w putting in log and commiting (update the state m/c)

putting in log is just that.
committed (state actually changed) when leader is sure that at least a majority of servers has it in their logs.

Split Vote:- No one gets majority

Ex:- ≥ Candidates

6 total servers
Each gets 3 Votes

So, will timeout waiting for the i-th vote starts leader election again ≡ New Term

leader's log is the log

leader Keeps:-
nextindex[i] = index of next log entry to send to server i

Raft:- Once a log entry is committed, it will be there in all future leaders.

So, suppose it does.
Say, an entry committed in term X.
Let Y = Smallest term > X when the entry is not there in the log.
So, in term Y, leader elected, got vote from majority
also, vote requires majority
⟹ ∃ at least one process P, that take part in both commit & vote
When P voted for term Y, it must have already committed the entry for term X

Also when P voted for term y leader, the leader's log must have been at least as up to date as P's

Case 1:- P's last log term entry = Same as leader's last log entry term.

Then, leader must have all entries of P ⇒ the committed entry also

Case 2:- leader's last entry term > P's last log entry term.

Q) Do we need to send reads to also?

linearizibility ⇒ All reads and Writes

Sequential ⇒ Can directly reply to read (No need for ordering)

Atomic Order or Total Order Multicast

Set of nodes sending msgs to all others.

Goal:- 1. If a msg is "delivered" by one node, it is delivered eventually by all nodes

2. All nodes deliver the msgs in the same order.

Normally, atomic multicast means atomic order mcast.

Sometimes, it refers to 1 msg from 1 node (then this means "atomic", the Cond 1 only)

Replication System:-

	Consistency
Passive Repl —	linearizable
Active Repl —	linearizibility Seq Consistency Eventual

Tool:-

- 2 ph Commit
- Consensus — Concept done
- Atomic Multicast — No Algo done
- Voting Based Protocols — Static, Dynamic

Voting Based Protocol:-

1. $Q_r + Q_w > N$
2. $2 Q_w > N$

Versioning with each data — Updated in Write

Jayodia–Muchler Majority Voting Algorithm

Normal protocol — "majority" may not be there if nodes fail, n/w gets partitioned etc.

Idea:- Redefine "majority" of "What".

Each Data: VN — Version No.
NN — No of nodes which participated in the last update.

A node is said to be in a "distinguished partition" if no of nodes in the partition is $> \frac{NN}{2}$ for highest version no.

Basically, no explicit vote assignment,
But replicas with older version gets 0 vote

A node can do a write if it is in the distinguished partition, otherwise rejects the req

	A	B	C	D	E
VN	≥0	≥0	≥0	≥0	≥0
NN	5	5	5	5	5

Say ABC DE gets partitioned

Write Req at A

Split

A decides it is in "distinguished partition"
→ Write OKⁿs

	A	B	C	D	E
VN	≥1	≥1	≥1	≥0	≥0
NN	3	3	3	5	5

Write Req at D
— 2 is not majority of 5
— Refused

Say AB C ↔ AB C

Write Req at A!

	A	B	C	D	E
VN	22	22	≥1	≥0	≥0
NN	2	2	3	5	5

Say, D,E joins back with AB

ABDE C

	A	B	C	D	E
VN	23	23	≥1	≥3	≥3
NN	4	4	4	5	4

CAP Theorem:-

In a distributed system, you can have atmost two of Consistency, Availability & Partition Tolerance

Eventual Consistency

Epidemic Protocol — Similar to flooding

Replication Topology — Chooses who replicates from who.

Simplest form — sp_tree, say children replicates from parent.

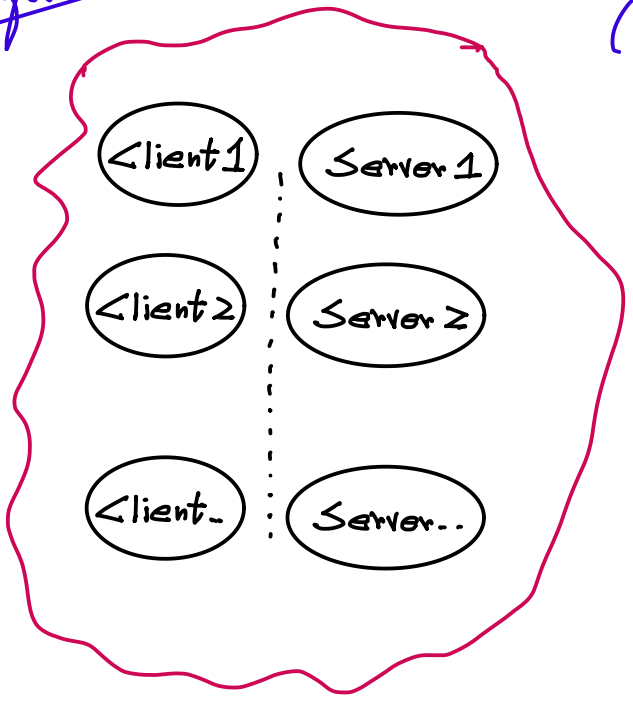
Scheduled Replication — Says who gets when

Distributed Authentication:-

Within an Organization (Same admin domain)

Clients within an org wants to Connect to services (Servers) inside the organization

Organization



1. Use Password :-

Simple Way :- all servers have all user's password

— Too insecure (hack in one, hack in all)

— Inefficient (What if user changes password)

Goal :-

1. Type password as less as possible
2. Store " in as few machines as possible
3. Send " over net as few times as possible
4. User is unware of authentication most of the time
5. Scalable to large no of Clients & Services

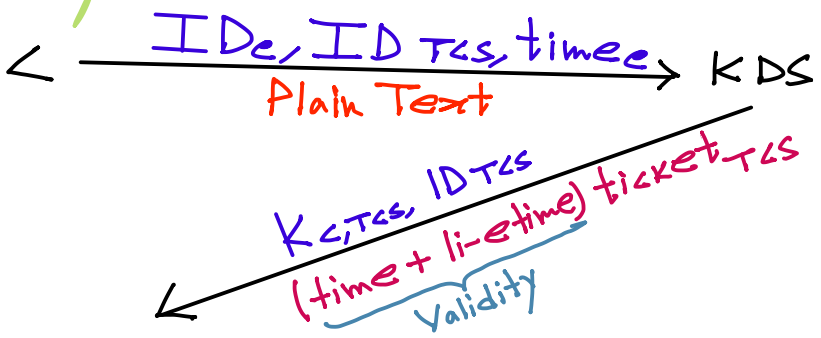
Who are we protecting against?

1. Malicious users (steals user credentials on a m/c)
2. Network spoofing attack (ex IP spoofing attack)
3.

KDC is also called AS (Authenticator Service)

KDC/AS

C logs in



Client gets it :-

If Client is indeed the Client, it claims to be (i.e typed in the right password, the K_C it generates will match the K_C KDC encrypted its response with

⟹ C will be able to decrypt

If not, fail

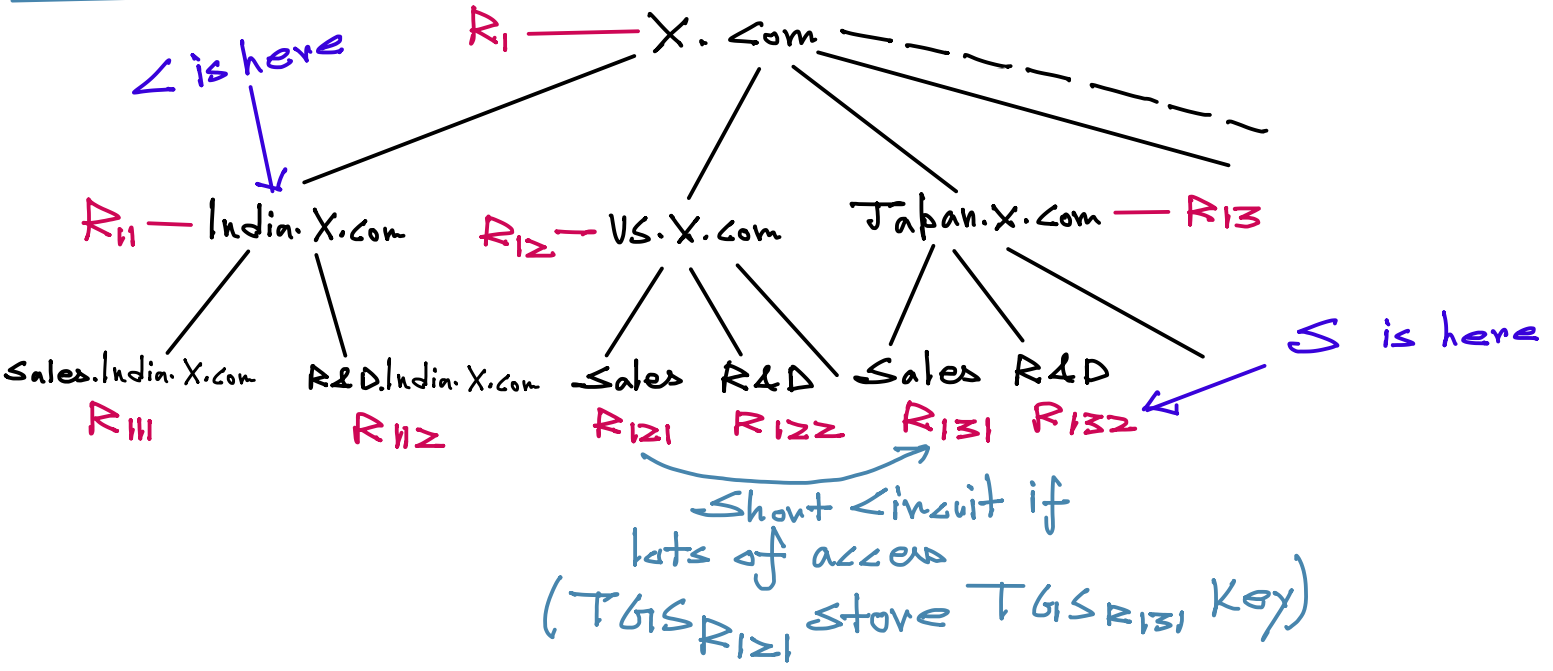
time + lifetime = Validity of the ticket.

Requirement:- Clocks must be synchronized.

V sending time_c+1 back ≡ Mutual Authentication

Ticket given by KDC for TGS = TGT (Ticket Granting Ticket)

Cross Realm Authentication



Ex. A client C in R11 wants access a service S in R132

$C \rightarrow TGS_{R11}$ — $(TGS_{R1}, ticket_{TGS_{R11}}, auth_C)$

$TGS_{R11} \rightarrow C$ — $(K_C, TGS_{R1}, TGS_{R11}, time_{TGS_{R11}}, ticket_{TGS_{R1}})$

$C \rightarrow TGS_{R1}$ — $(TGS_{R13}, ticket_{TGS_{R1}}, auth_C)$

$TGS_{R1} \rightarrow C$ — $(K_C, TGS_{R13}, TGS_{R13}, time_{TGS_{R1}}, ticket_{TGS_{R13}})$

$C \rightarrow TGS_{R13}$ — $(TGS_{R132}, ticket_{TGS_{R13}}, \ldots)$

$TGS_{R13} \rightarrow C$ — Give ticket for TGS of R132

Now C can ask TGS_{R132} for ticket of S.

What if R121 needs to access R131 lot?
But not vice-versa

→ Have TGS_{R131} stored in TGS_{R121}

"Renewable" Tickets

- has a "Current Expiring" time and a "find expiry" time
- Before Expiring, the ticket can be presented to TGS/AS for renewal & a new ticket can be obtained.

"Forwardable" Tickets

Ticket that can be forwarded by a service to another service on behalf of a Client.

Client $\xrightarrow{\text{access}}$ Web Server $\xrightarrow{\text{access}}$ DB Server for Client Data