IF3230 – Sistem Paralel dan Terdistribusi
Fault Tolerant dan Consensus

Achmad Imam Kistijantoro (imam@staff.stei.itb.ac.id)
Judhi Santoso (judhi@ staff.stei.itb.ac.id)
Anggrahita Bayu Sasmita (bayu.anggrahita@ staff.stei.itb.ac.id)

Fault Tolerant

- Partial Failure
  - Failure yang terjadi pada salah satu komponen sistem terdistribusi
  - Tidak terjadi pada single-machine system

- An **important** goal in distributed systems design is to construct the system in such a way that it can **automatically recover** from **partial failures** without seriously affecting the overall performance.

Fault Tolerance Terms

Fail : system cannot meet its promises

Error : a part of a system's state that may lead to a failure

Fault : the cause of an error

Fault tolerance

- System can provide its services even in the presence of faults
- Dependable systems

Type of fault :

- Transient : hanya terjadi satu kali kemudian hilang
- Intermittent : terjadi berulang kali
- Permanent : terjadi terus menerus hingga komponen diganti

Dependable Systems

4 requirement of Dependable Systems (Kopetz and Verissimo, 1993)

- Availability
  - System is ready to be used immediately
- Reliability
  - system can run continuously without failure
- Safety
  - situation that when a system temporarily fails to operate correctly, nothing catastrophic happens
- Maintainability
  - how easy a failed system can be repaired

Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	A server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Failure Detection

- To properly mask failures, we need to detect them
- 2 mechanisms:
  - Actively send "are you alive" message to each other
  - Passively wait message come in from other process
- There has been a huge body of **theoretical work** on failure detectors
  - **Timeout** mechanism is used to check whether a process has failed.

Failure Masking by Redundancy

Jika failure terjadi, sedapat mungkin failure disembunyikan dari proses lain/user

- Key technique for masking faults is to use redundancy

- Information redundancy
  - Adding extra bits to allow recovery from garbled bits
  - Hamming code
- Time redundancy
  - Mengulangi action
  - Berguna untuk fault jenis transient atau intermitten
- Physical redundancy
  - Adding extra processes or extra component to replace fault component

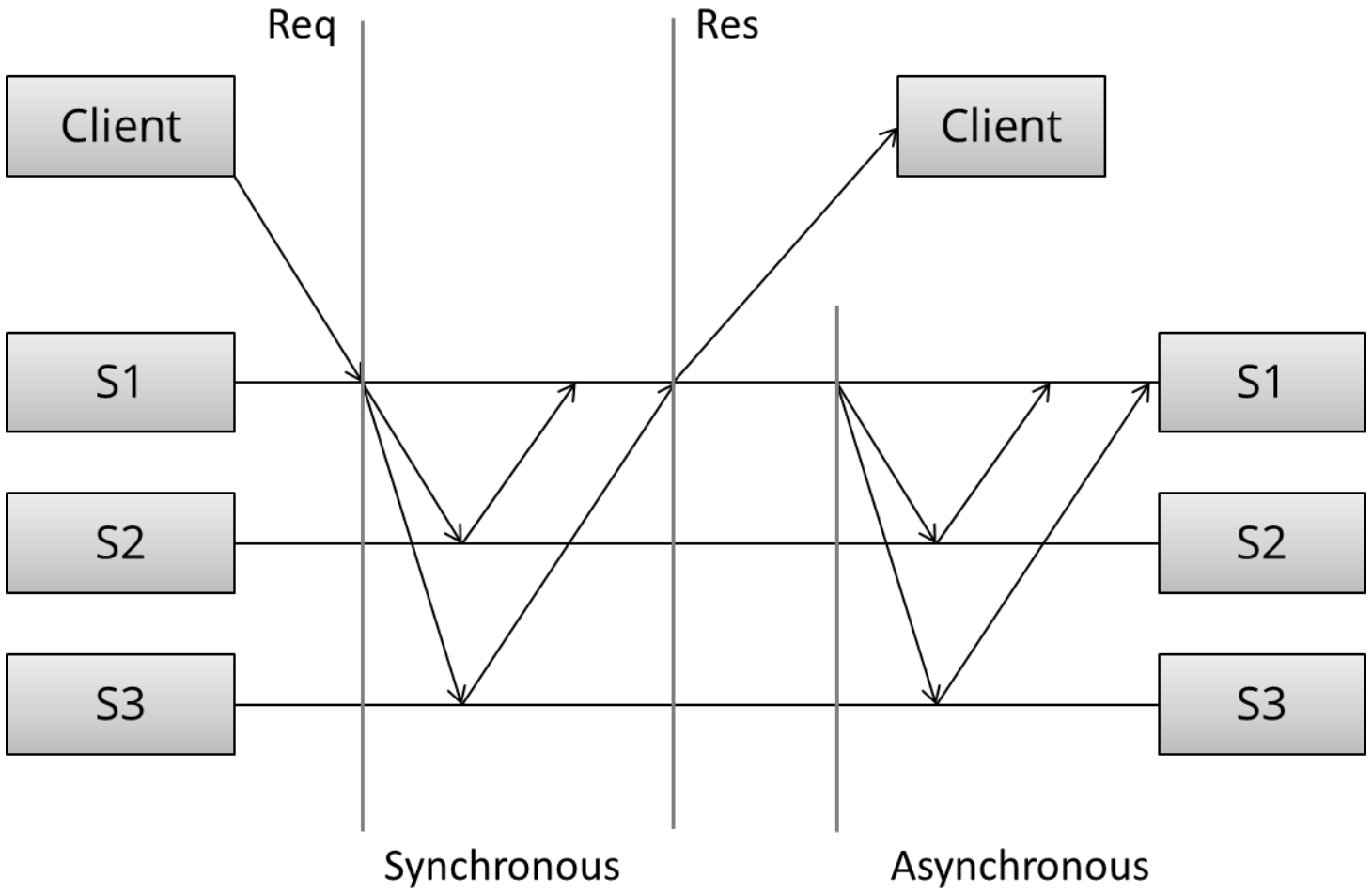
Failure Masking and Replication

Pendekatan utama untuk menangani proses yang faulty adalah dengan menjalankan beberapa proses yang identik dalam sebuah grup

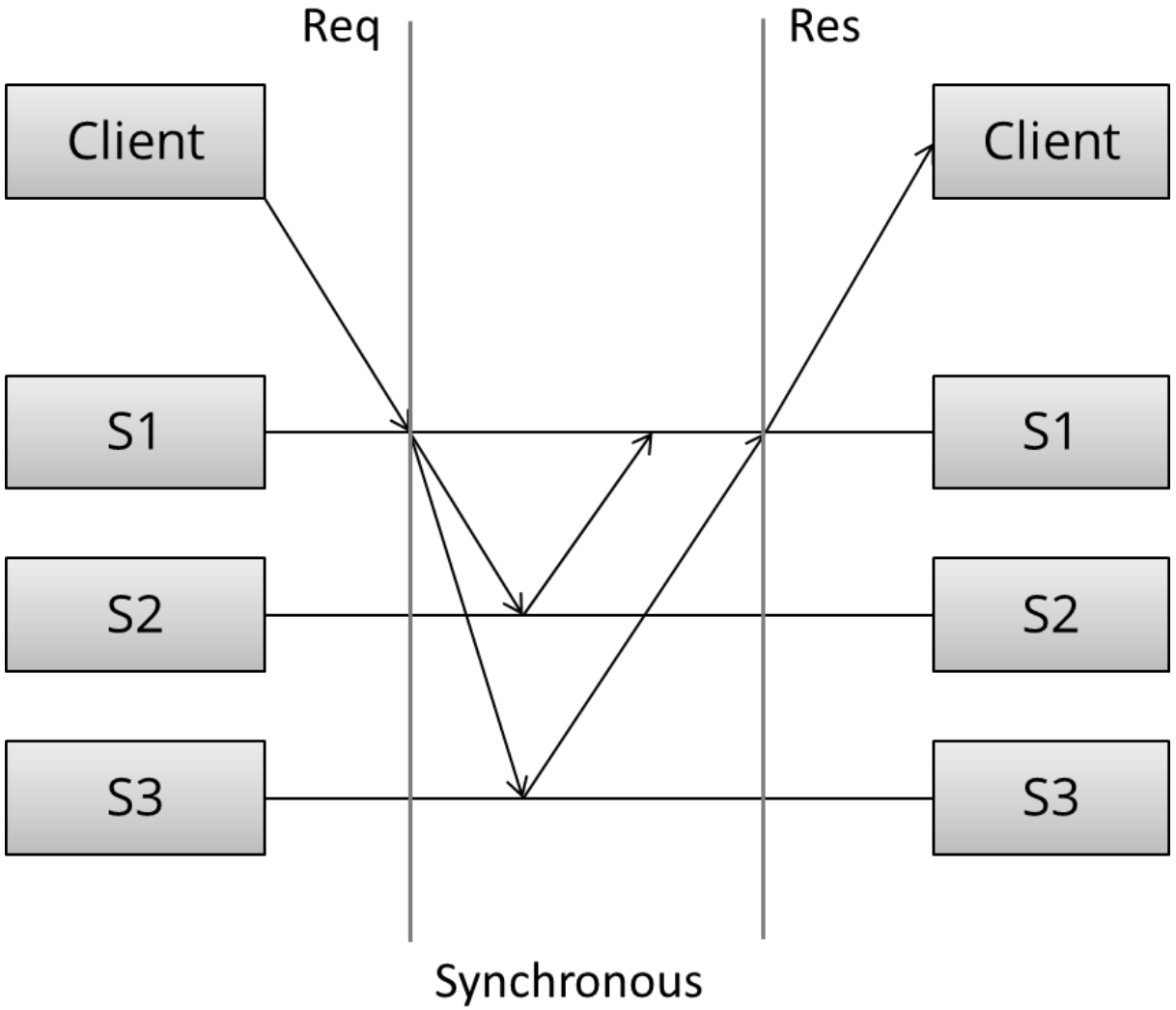
- saat sebuah pesan dikirim ke grup, semua anggota grup menerimanya
- Jika salah satu proses pada grup fail, ada proses lain yang dapat mengambil alih

- Sebuah sistem disebut **k fault tolerant** jika dia dapat survive faults pada *k* components dan tetap berjalan sesuai spesifikasi
  - Untuk proses secara umum, dengan ***k* + 1** backup process cukup untuk menyediakan *k* fault tolerance.
  - Untuk proses tertentu (misal: paxos), minimum ***2k* + 1** correct process dibutuhkan untuk mencapai *k* fault tolerance.

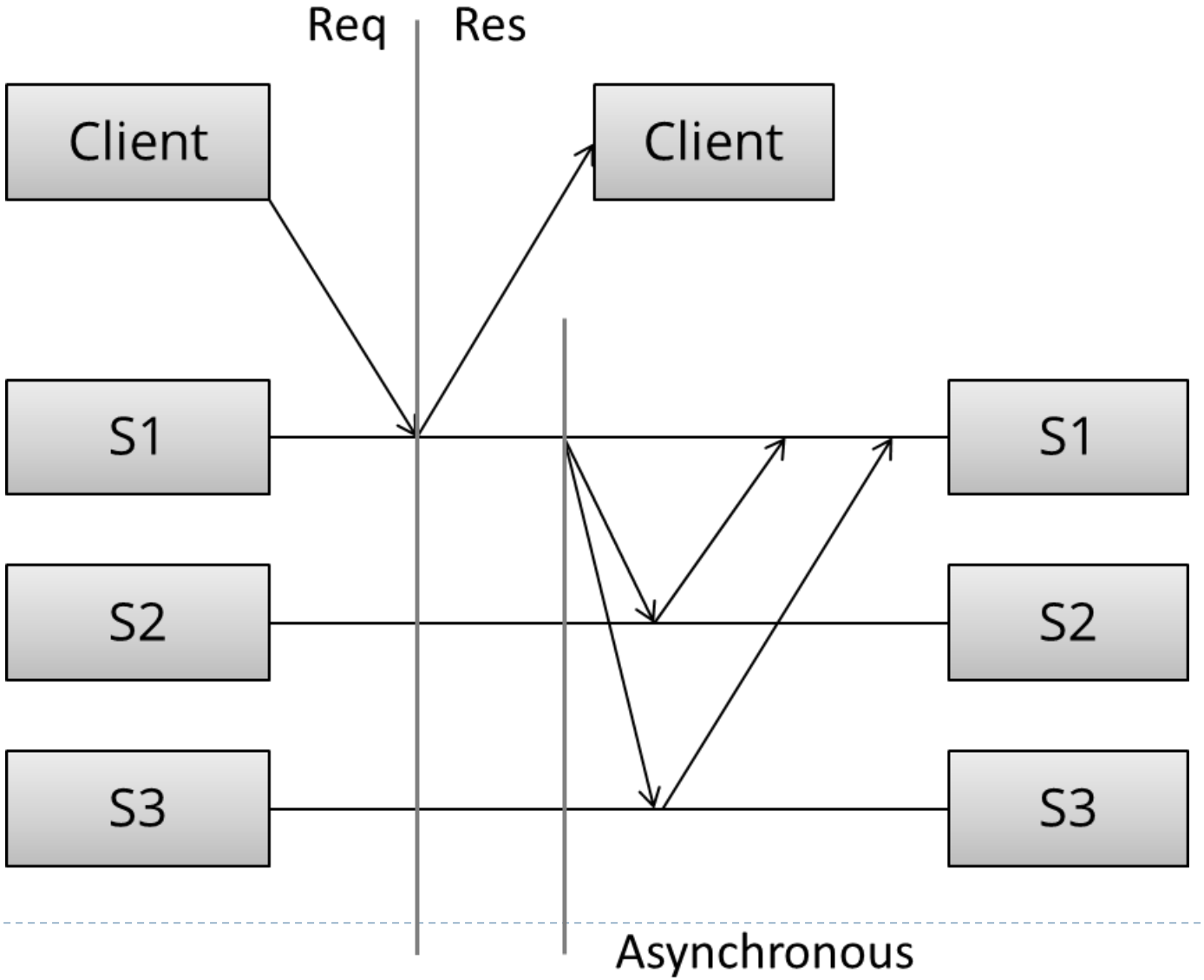
Replikasi


Req
Res
Client
Client
S1
S2
S3
S1
S2
S3
Synchronous
Asynchronous

Synchronous Replication (active, eager, push, pessimistic)



Asynchronous replication (lazy, passive, pull, optimistic)



Replikasi Primary/Backup

- Sering disebut Master/Slave replication, Log Shipping replication
- Update dilakukan pada master/primary, dan log operasi (atau state change) dikirimkan ke replika backup
- Ada 2 variant:
  - asynchronous primary backup replication
  - synchronous primary backup replication

- Asynchronous primary/backup berpotensi kehilangan update, jika master fail sebelum mengirimkan backup
- Synchronous primary/backup berpotensi bermasalah jika master fail sesaat sebelum memberikan ack ke client, namun backup sudah commit
- **Split-brain** terjadi jika antara primary dan backup terputus koneksi

Consensus

- Failure masking dapat dilakukan dengan redundancy
  - Misal menambah jumlah server untuk meningkatkan performansi dan memastikan sistem tetap berjalan meskipun terjadi failure pada suatu server
  - Perlu sinkronisasi agar semua server memiliki pemahaman yang sama (agreement) atas suatu nilai

- Solusi : consensus
  - Contoh algoritma consensus: Paxos

Consensus Goal

Allow a group of processes to agree on a result

- All processes must agree on the same value
- The value must be one that was submitted by at least one process (the consensus algorithm cannot just make up a value)

- Consensus requirements
  - **Validity**: Only proposed values may be selected
  - **Uniform agreement**: No two nodes may select different values
  - **Integrity**: A node can select only a single value
  - **Termination** (progress): Every node will eventually decide on a value

2 Phase Commit

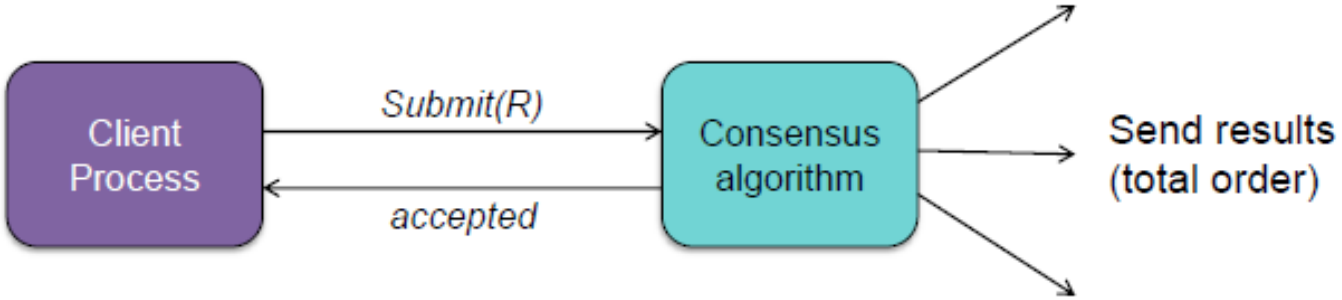
- umum digunakan pada database
- memastikan partisipan sepakat atau tidak untuk commit
- Phase 1:
  - koordinator: prepare
  - partisipan: Ok/Not Ok
- Phase 2:
  - koordinator: decision (abort/commit)
  - partisipan: Ack dan menerapkan decision

- berpotensi blocking jika koordinator fail saat phase 2

Paxos

- Fault-tolerant distributed consensus algorithm
  - tidak block jika mayoritas proses berjalan
  - algoritma membutuhkan mayoritas (2P+1) proses berjalan untuk menangani kegagalan P proses
- Goal: menyediakan konsisten event order untuk multiple client
  - Setiap mesin menjalankan algoritma agree pada proposed value dari client
  - semua value akan diasosiasikan dengan event/action
  - paxos menjamin tida; ada mesin yang mengasosiasikan value denganevent lain
- Abortable consensus
  - permintaan client dapat ditolak
  - client harus mengirimkan ulang request

Programmer's View



while (submit_request(R) != ACCEPTED) ;

Think of R as a key:value pair in a database

Paxos Player

- Client: makes a request
- Proposers:
  - Get a request from a client and run the protocol
  - Leader: elected coordinator among the proposers (not necessary but simplifies message numbering and ensures no contention) – we don't count on the presence of a single leader
- Acceptors:
  - Multiple processes that remember the state of the protocol
  - Quorum = any majority of acceptors
- Learners:
  - When agreement has been reached by acceptors, a Learner executes the request and/or sends a response back to the client

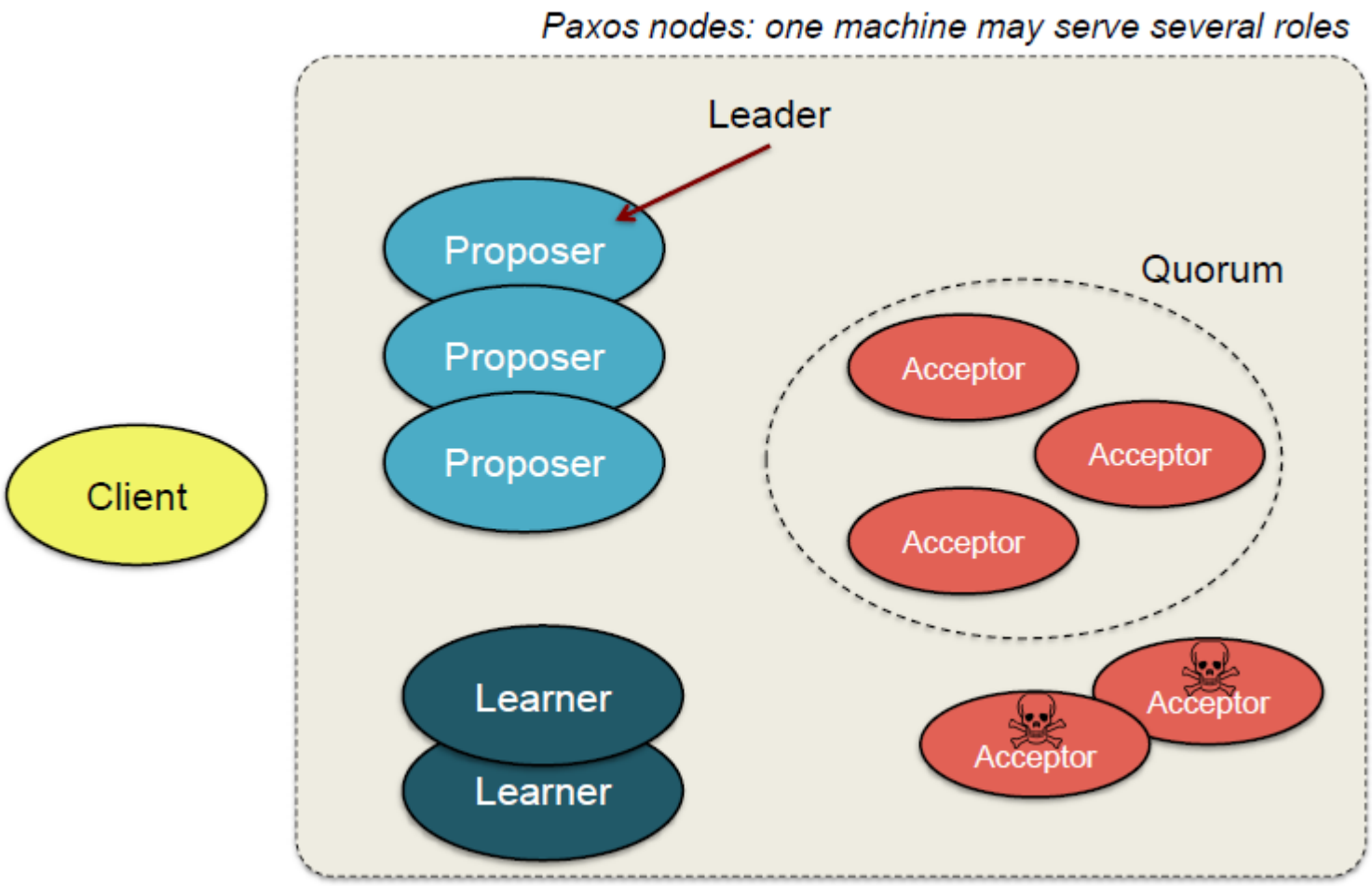
These different roles are usually part of the same system

What Paxos Does

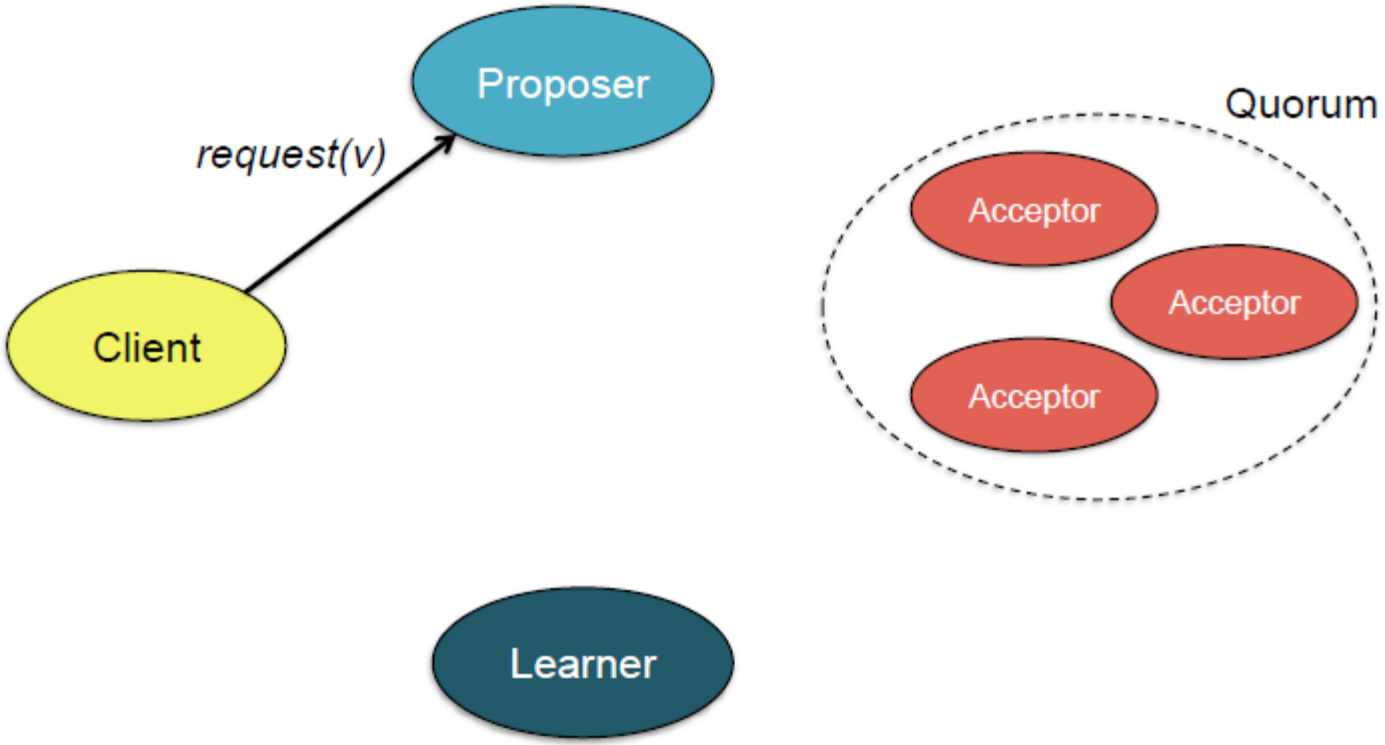
- **Paxos ensures a consistent ordering in a cluster of machines**
  - Events are ordered by sequential event IDs (*N*)
- Client wants to log an event: sends request to a Proposer
  - E.g., *value, v* = "*add $100 to my checking account*"
- **Proposer**
  - Increments the latest event ID it knows about
    - ID = sequence number
  - Asks all the acceptors to reserve that event ID
- **Acceptors**
  - A majority of acceptors have to accept the requested event ID

Proposal Numbers

- Each proposal has a number (created by proposer)
  - Must be unique (e.g., <*sequence#*>.<*process_id*>)
- Newer proposals take precedence over older ones
- Each acceptor
  - Keeps track of the largest number it has seen so far
- Lower proposal numbers get rejected
  - Acceptor sends back the {number, value} of the currently accepted proposal
  - Proposer has to "play fair":
    - It will ask the acceptors to accept the {number, value}
    - Either its own or the one it got from the acceptor


Paxos nodes: one machine may serve several roles
Leader
Proposer
Proposer
Proposer
Quorum
Acceptor
Acceptor
Acceptor
Client
Learner
Learner
Acceptor
Acceptor

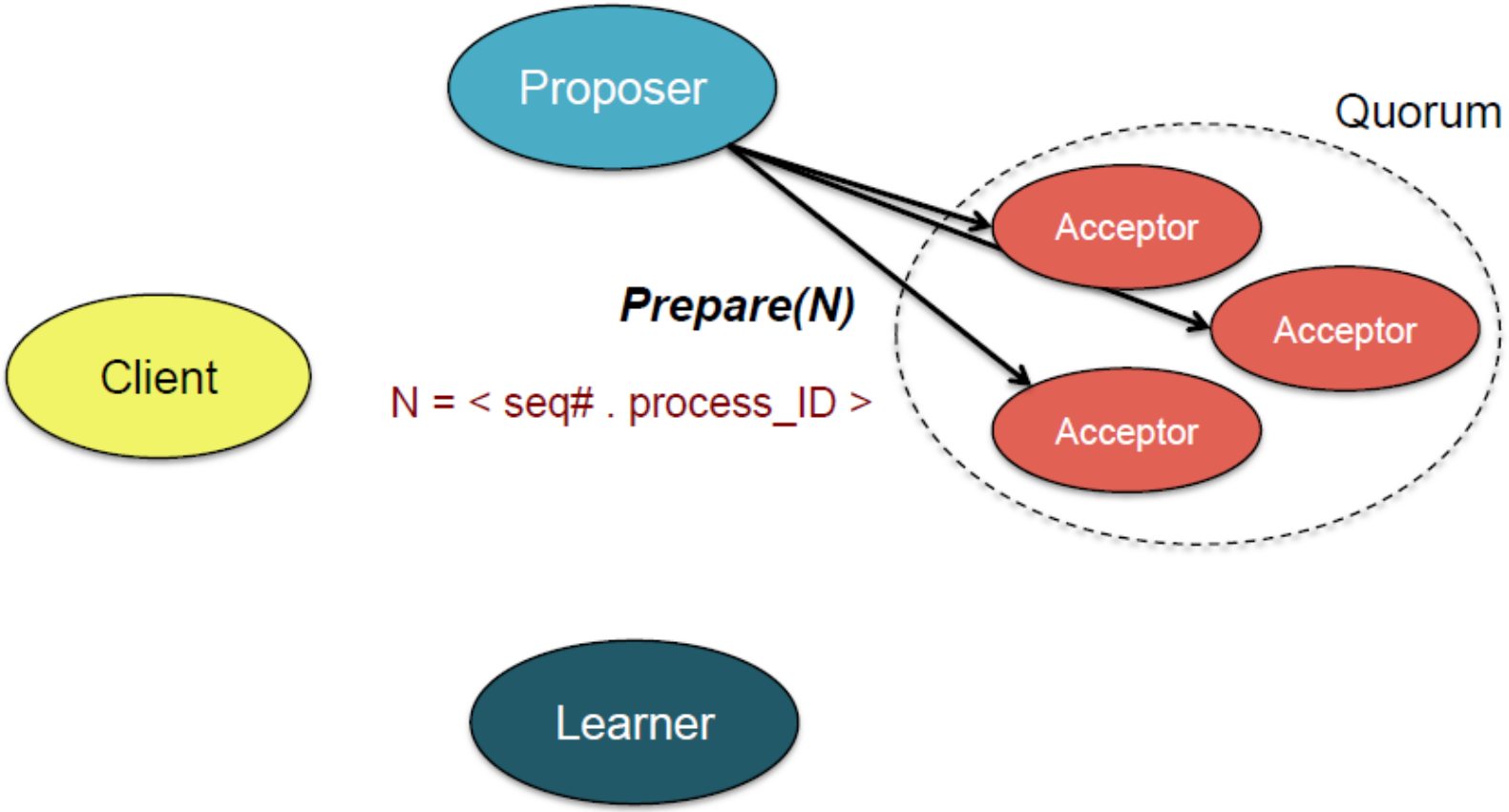
Phase 0


Proposer
request(v)
Client
Quorum
Acceptor
Acceptor
Acceptor
Learner

Phase 1a : Prepare

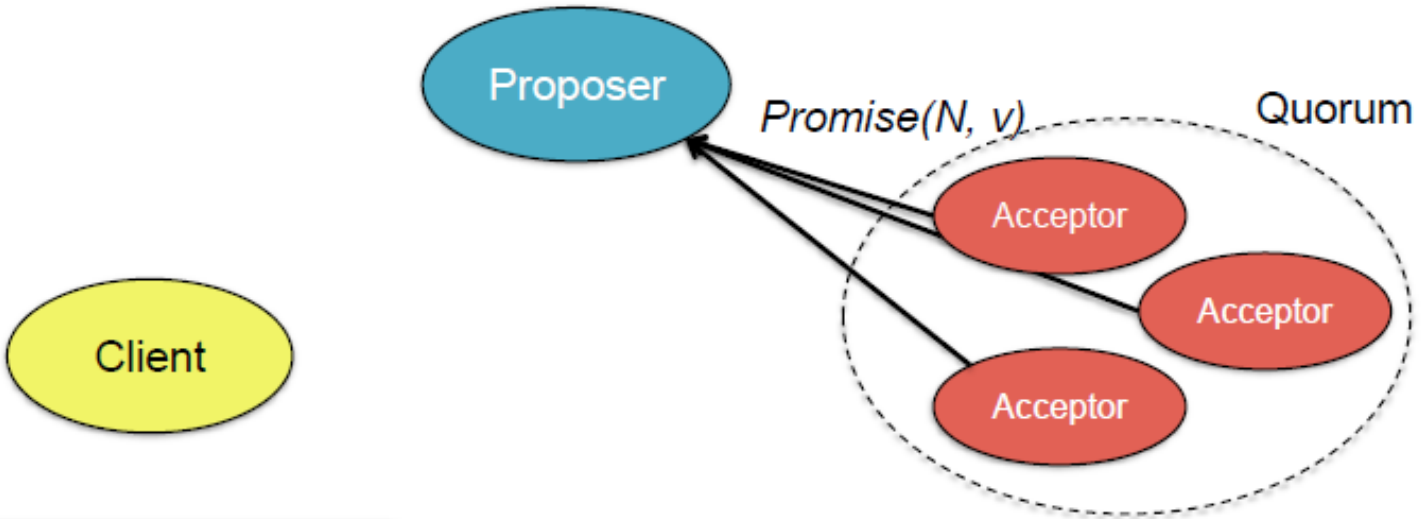
Proposer: creates a *proposal #N (N acts like a Lamport time stamp)*, where *N* is greater than any previous proposal number used by this proposer

Send to Quorum of Acceptors (however many you can reach – but a majority)



Phase 1b : Promise

Acceptor: if proposer's ID > any previous proposal
promise to ignore all requests with IDs < N
reply with info about highest accepted proposal, if there is one: { *N, value* }
else *Reject* the proposal



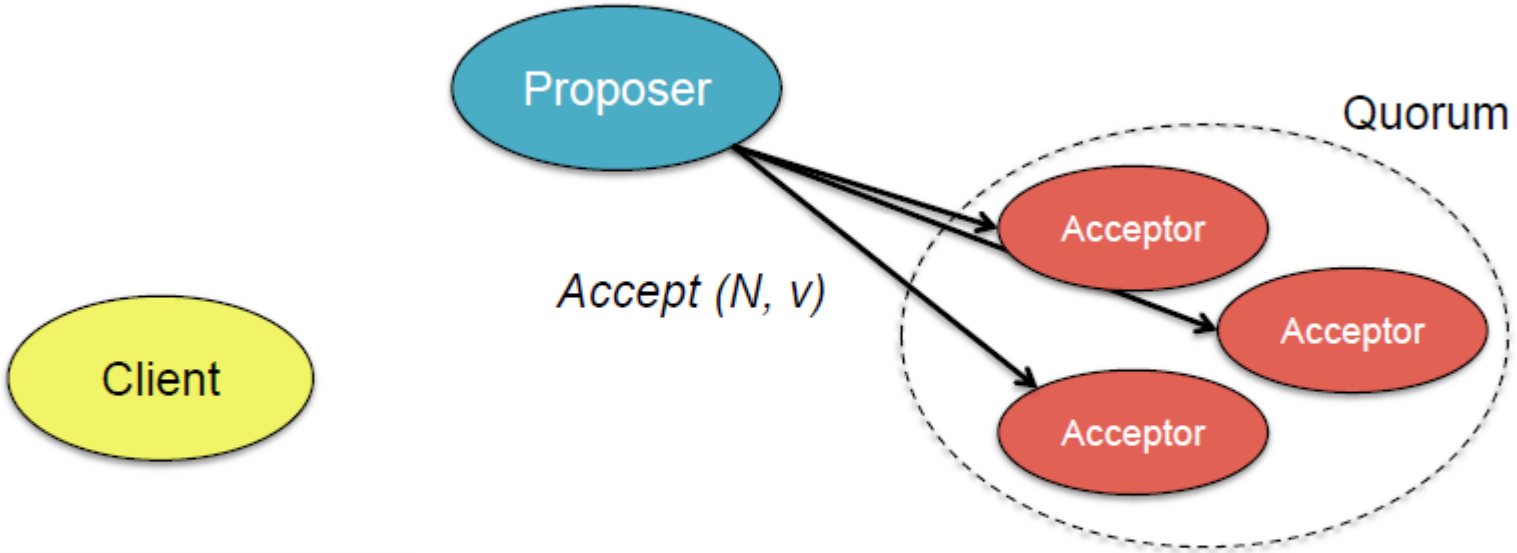
This step:
(a) Tells proposers about any value that has already been accepted.
(b) Rejects older proposals.

Promise to ignore all proposals < *N*

Promise contains the previous *N*

Phase 2a

Proposer: if proposer receives promises from the quorum (majority):
Attach a value *v* to the proposal (the event).
Send ***Accept*** to quorum with the **chosen** value
If promise was for another {N, v}, proposer MUST accept that instead

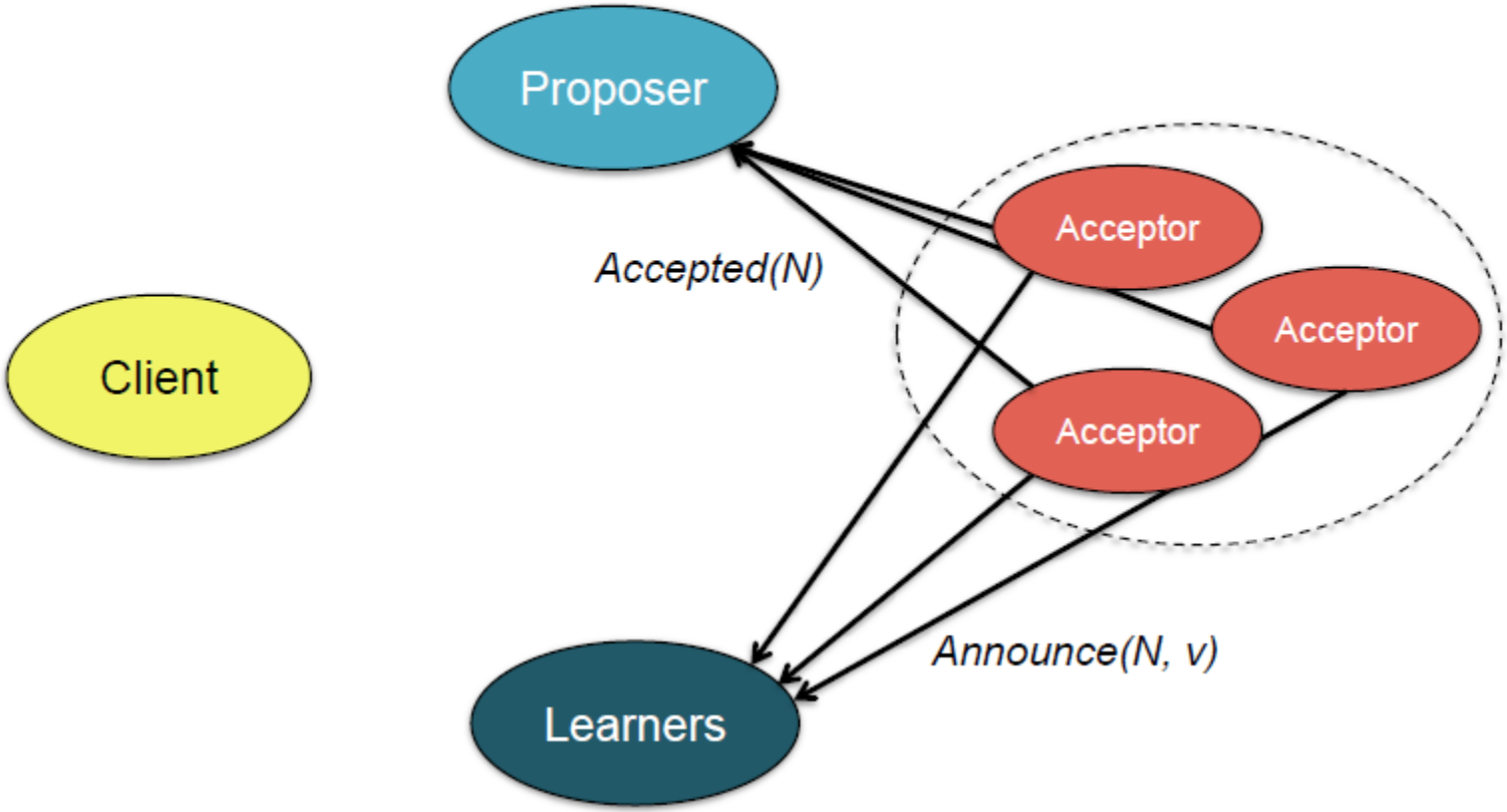


If the acceptor returned any (*N*, *v*) sets then the proposer must agree to accept one of those values intead of the value it proposed. It picks the *v* for the highest *N*.

Learner

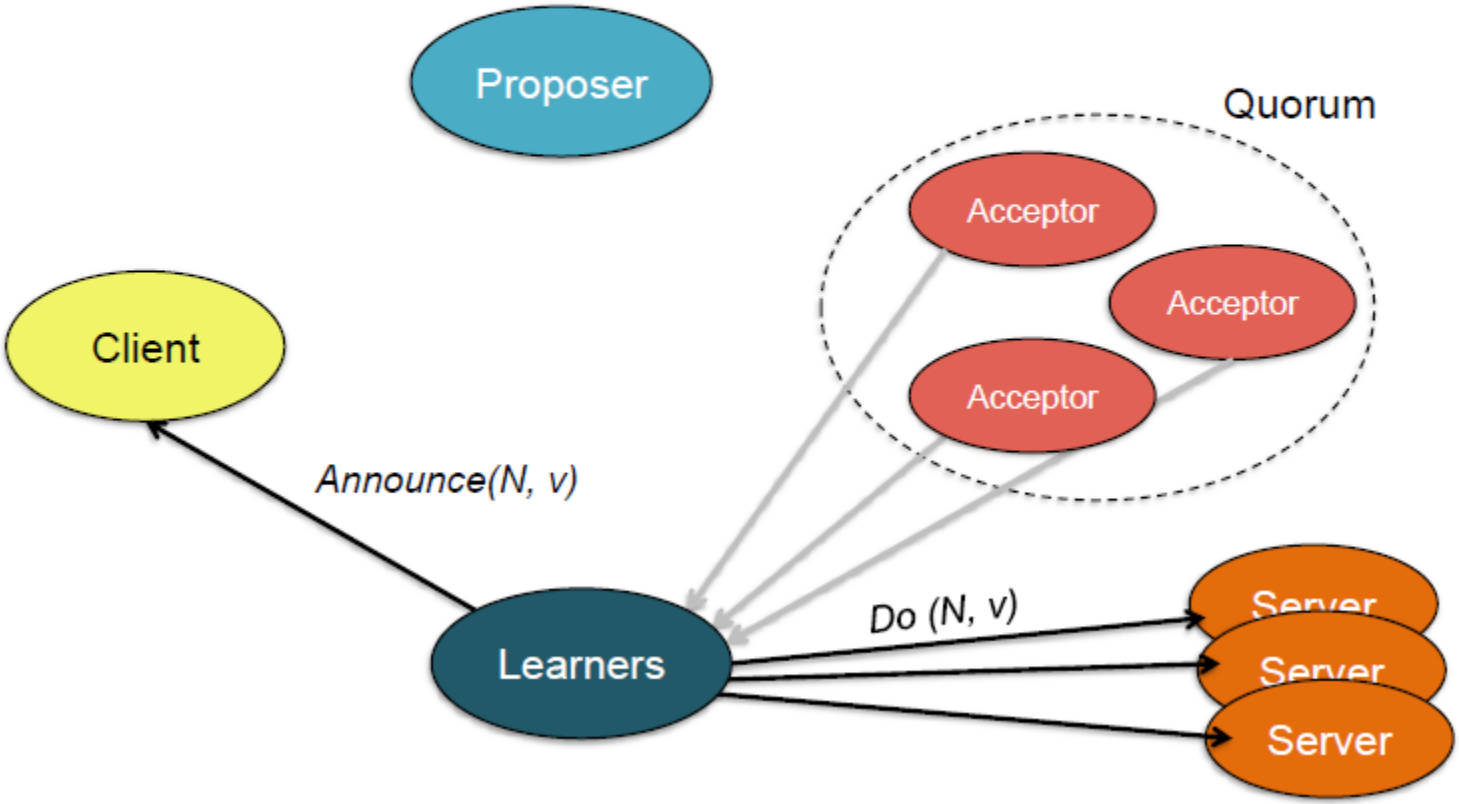
Phase 2b

Acceptor: if the promise still holds, then announce the value *v*
Send ***Accepted*** message to Proposer and every Learner
else ignore the message (or send *NACK*)



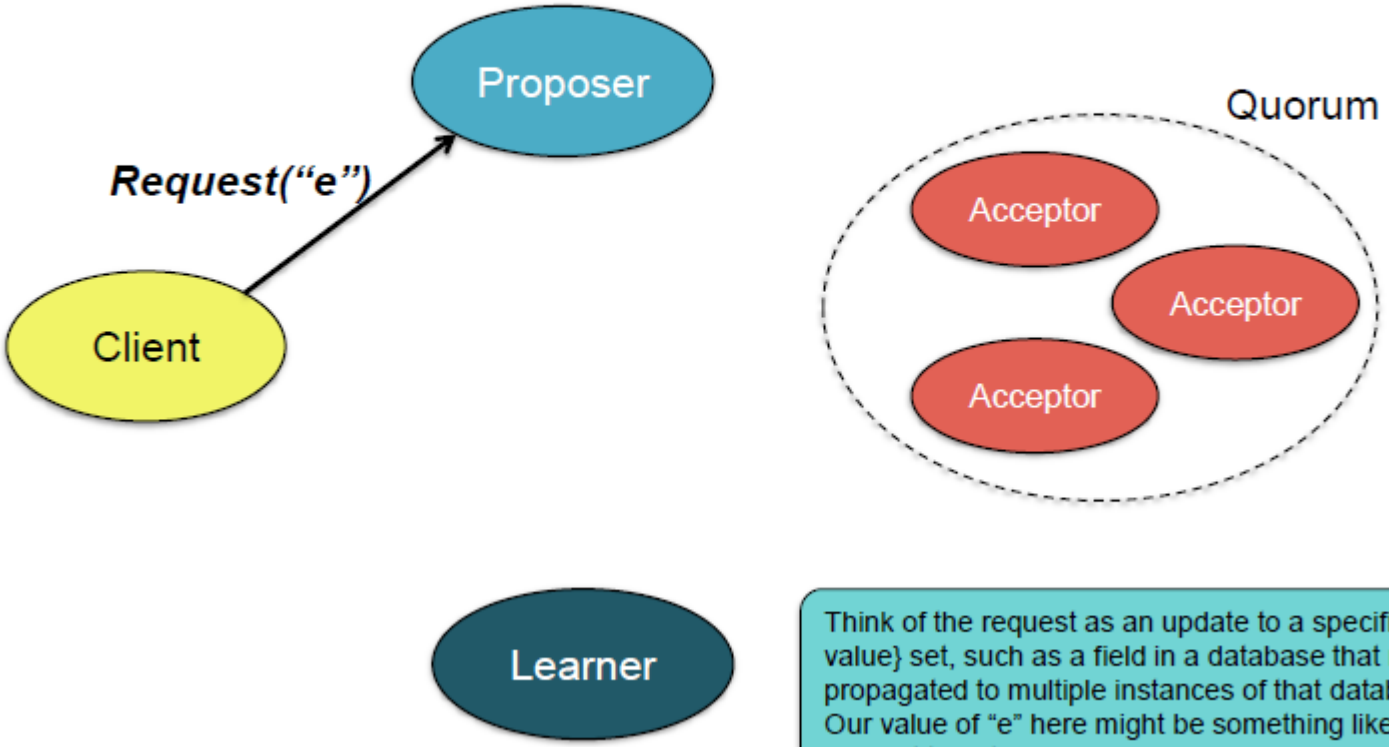
Phase 3

Learner: Respond to client and/or take action on the request



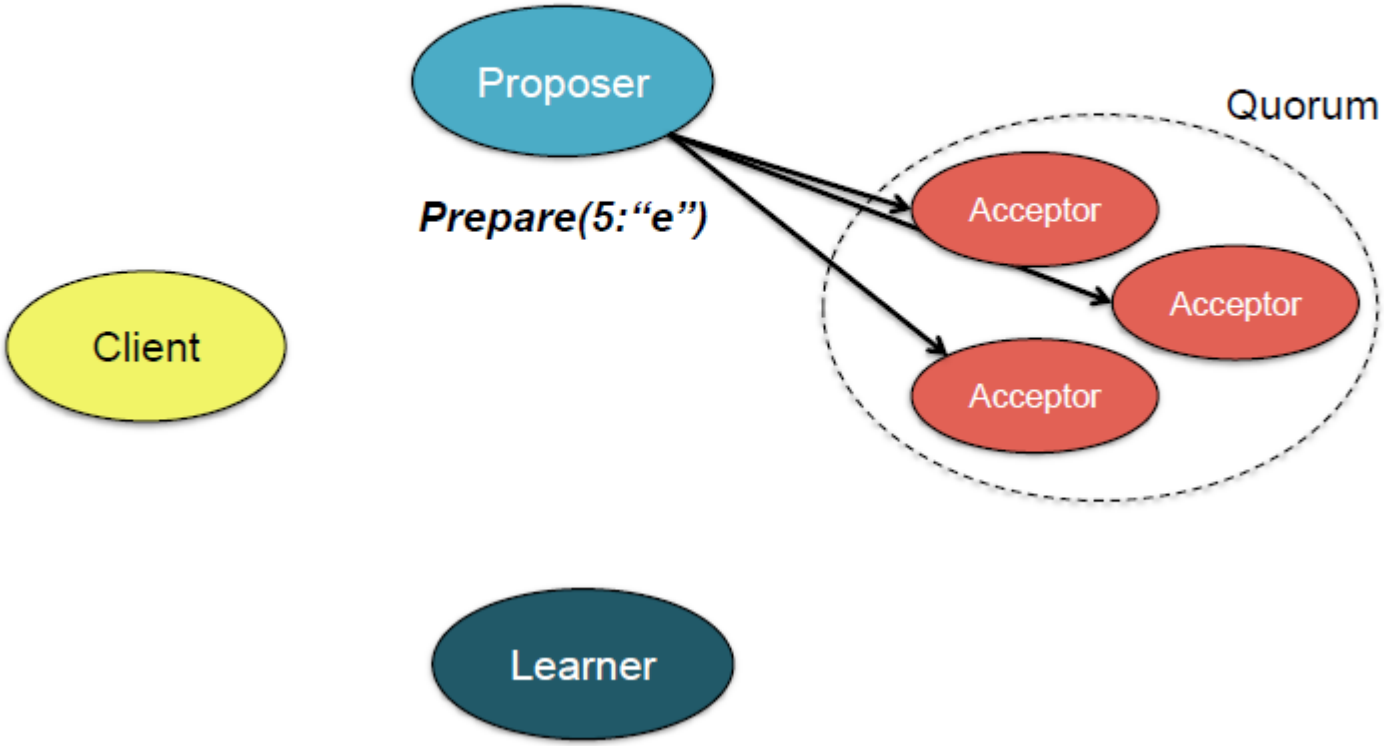
Contoh: phase 0

Client sends a request to a proposer

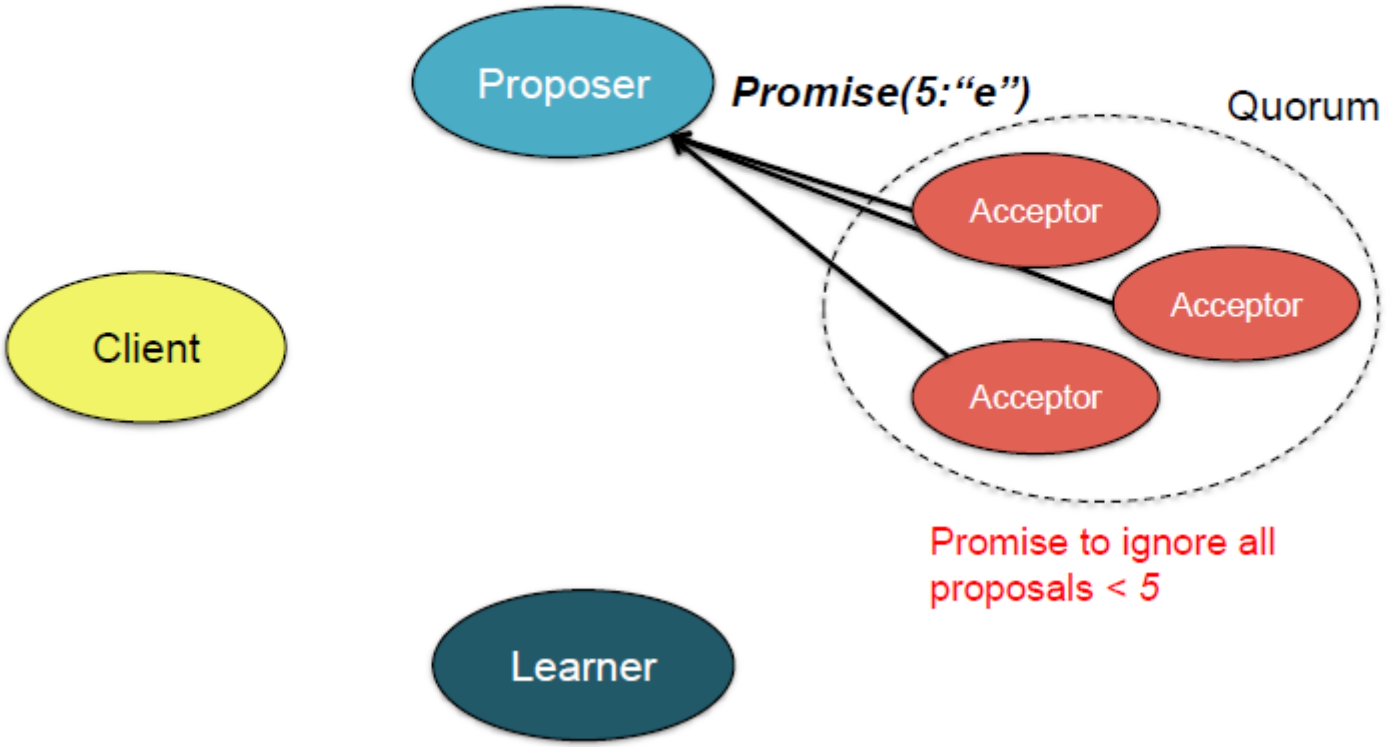


Think of the request as an update to a specific {key, value} set, such as a field in a database that may be propagated to multiple instances of that database. Our value of "e" here might be something like a request to set *name="e"*

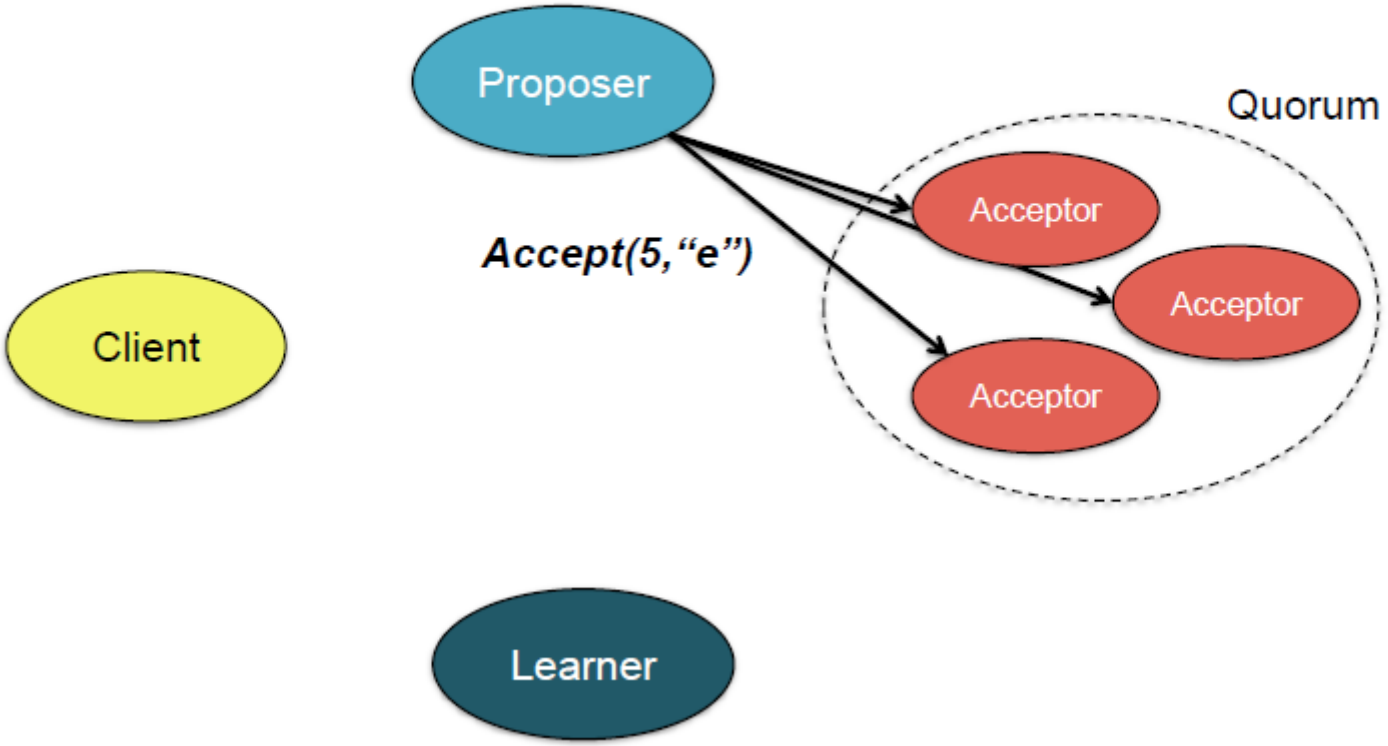
Proposer: picks a sequence number: 5
Send to Quorum of Acceptors



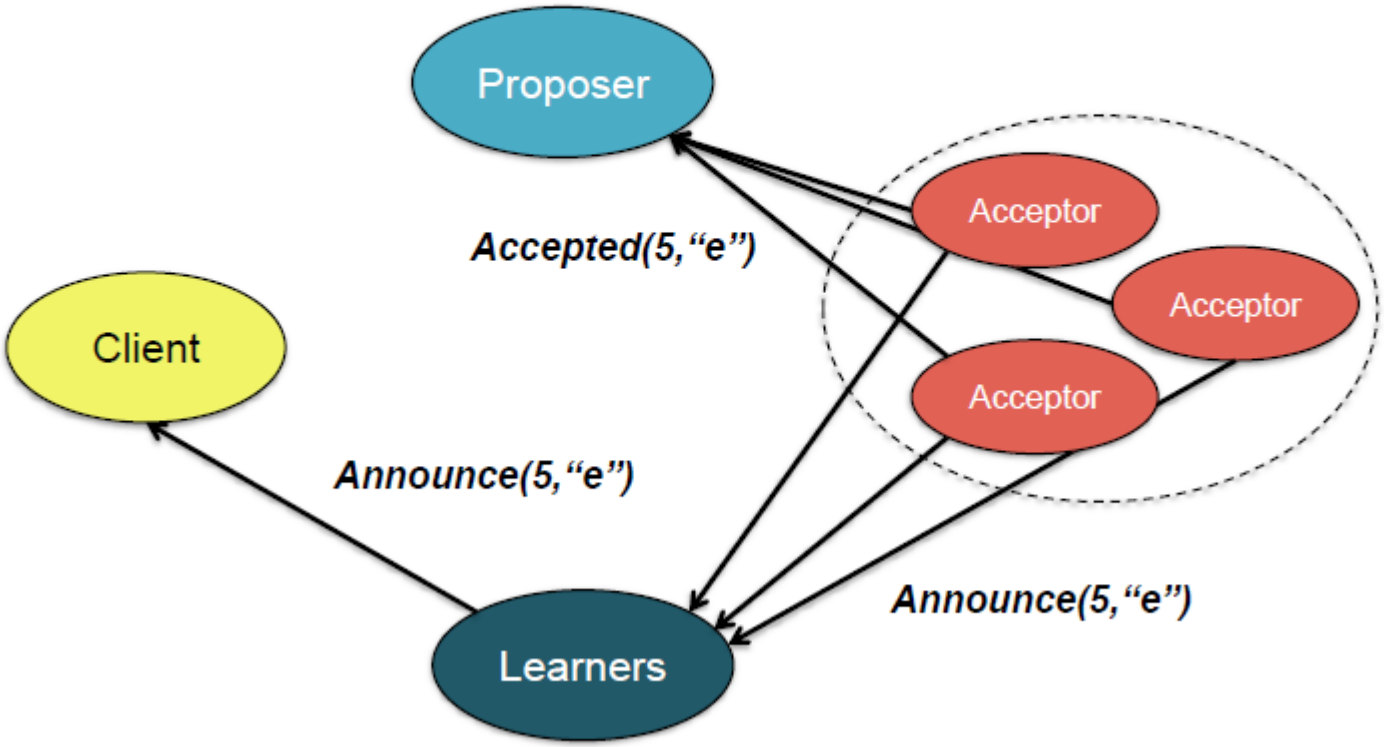
Acceptor: Suppose 5 is the highest sequence # any acceptor has seen
Each acceptor PROMISES not to accept any lower numbers



Proposer: Proposer receives the promise from a majority of acceptors
Proposer must accept that <seq, value>

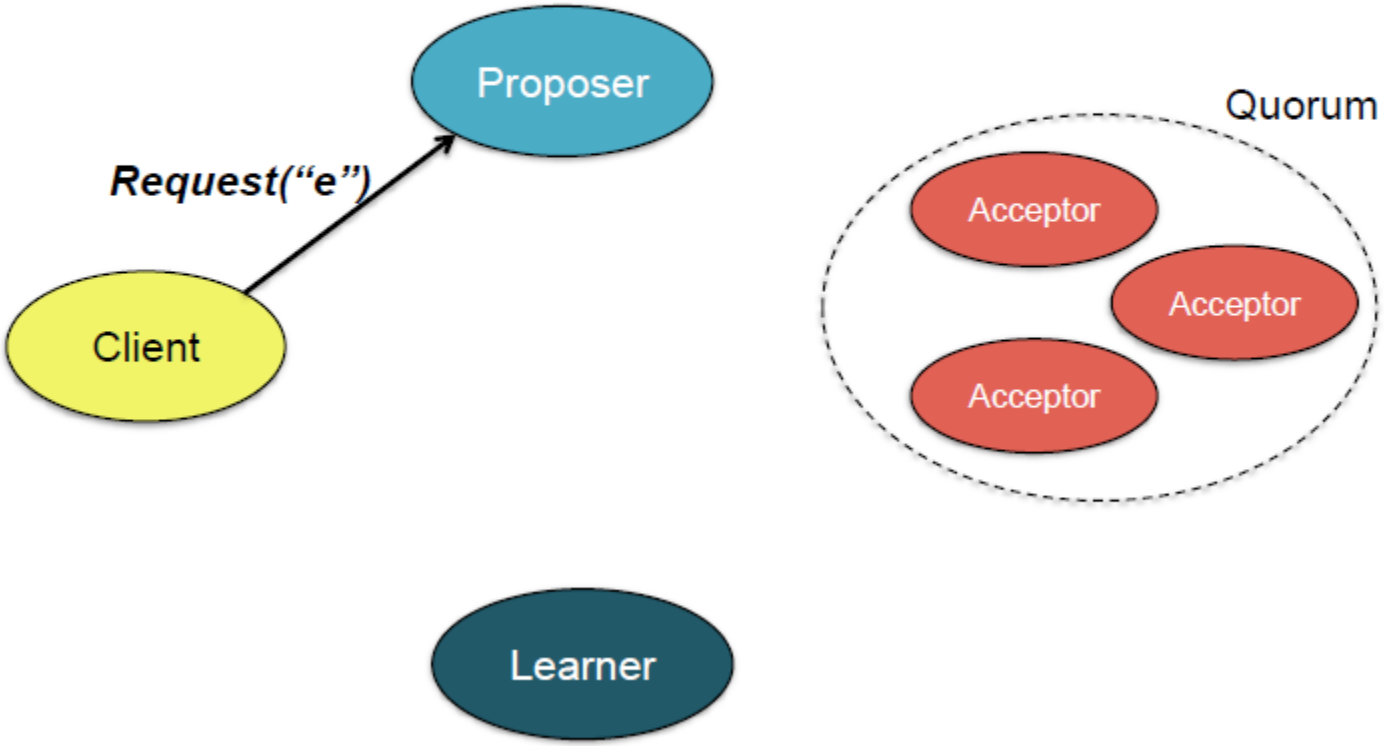


Acceptor: Acceptors state that they accepted the request



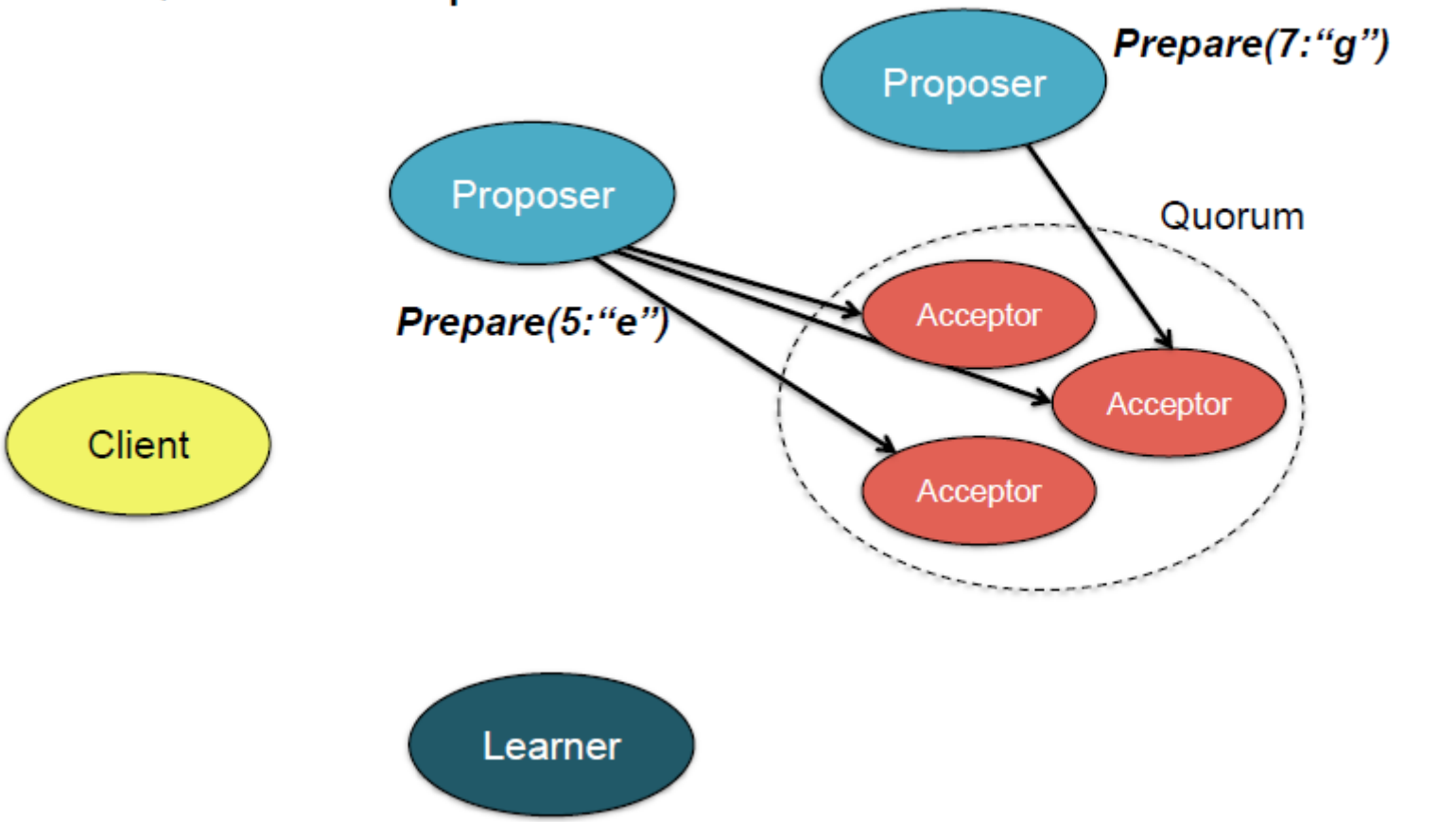
Contoh 2

Client sends a request to a proposer

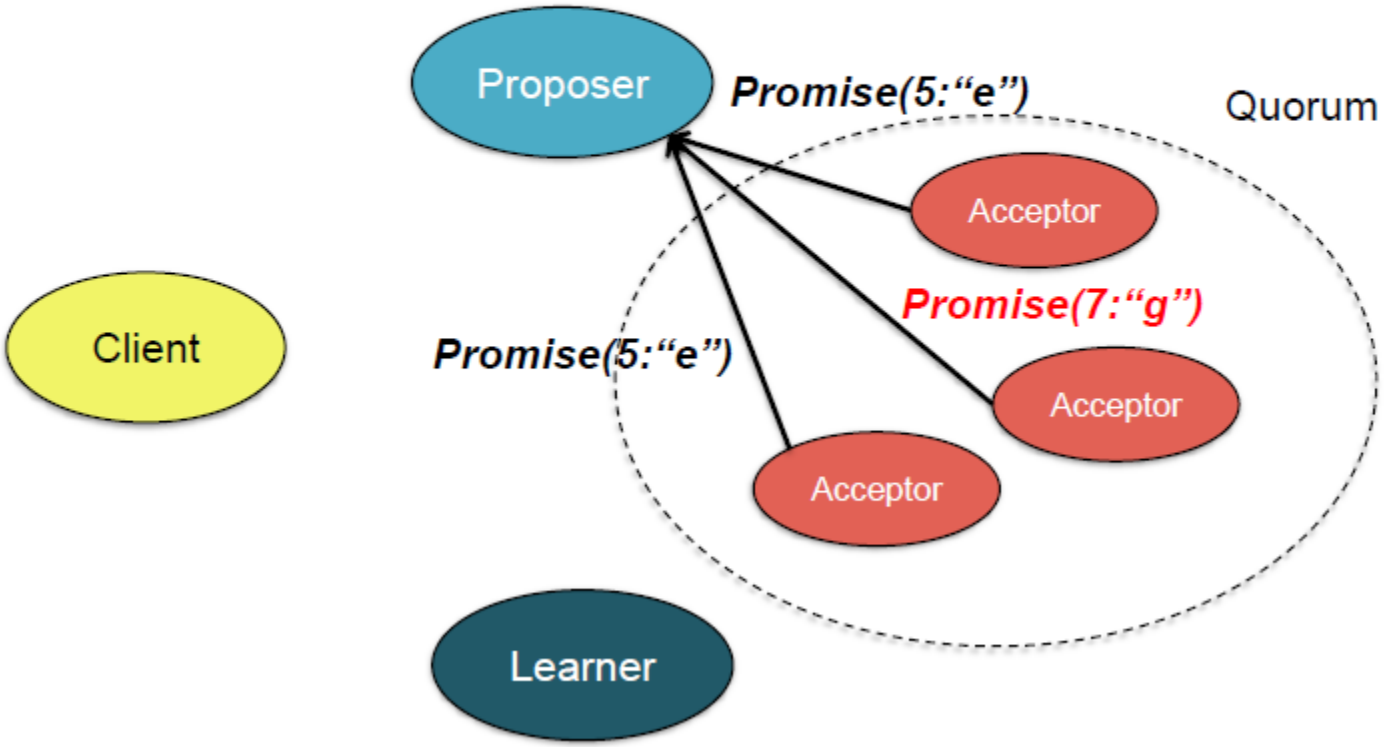


Proposer: picks a sequence number: 5
Send to Quorum of Acceptors

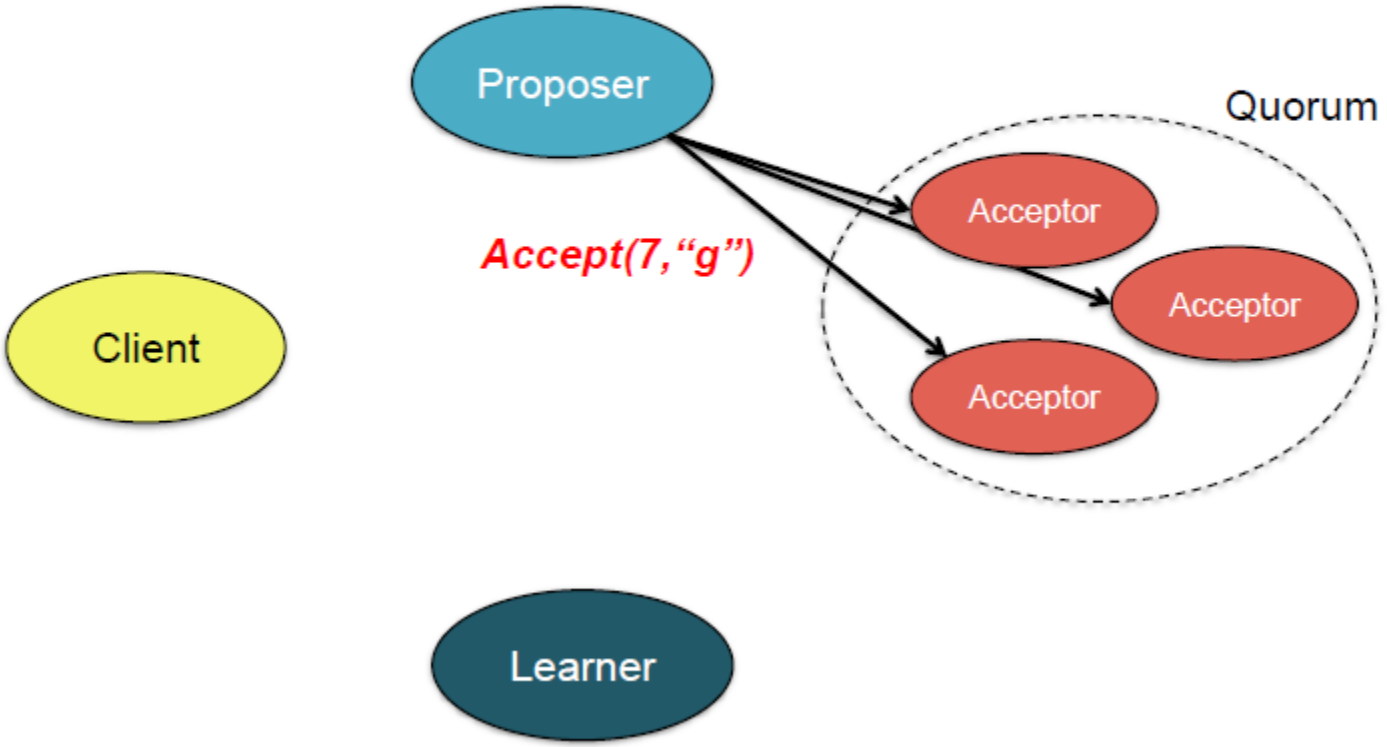
One acceptor receives a higher offer BEFORE it gets this PREPARE message



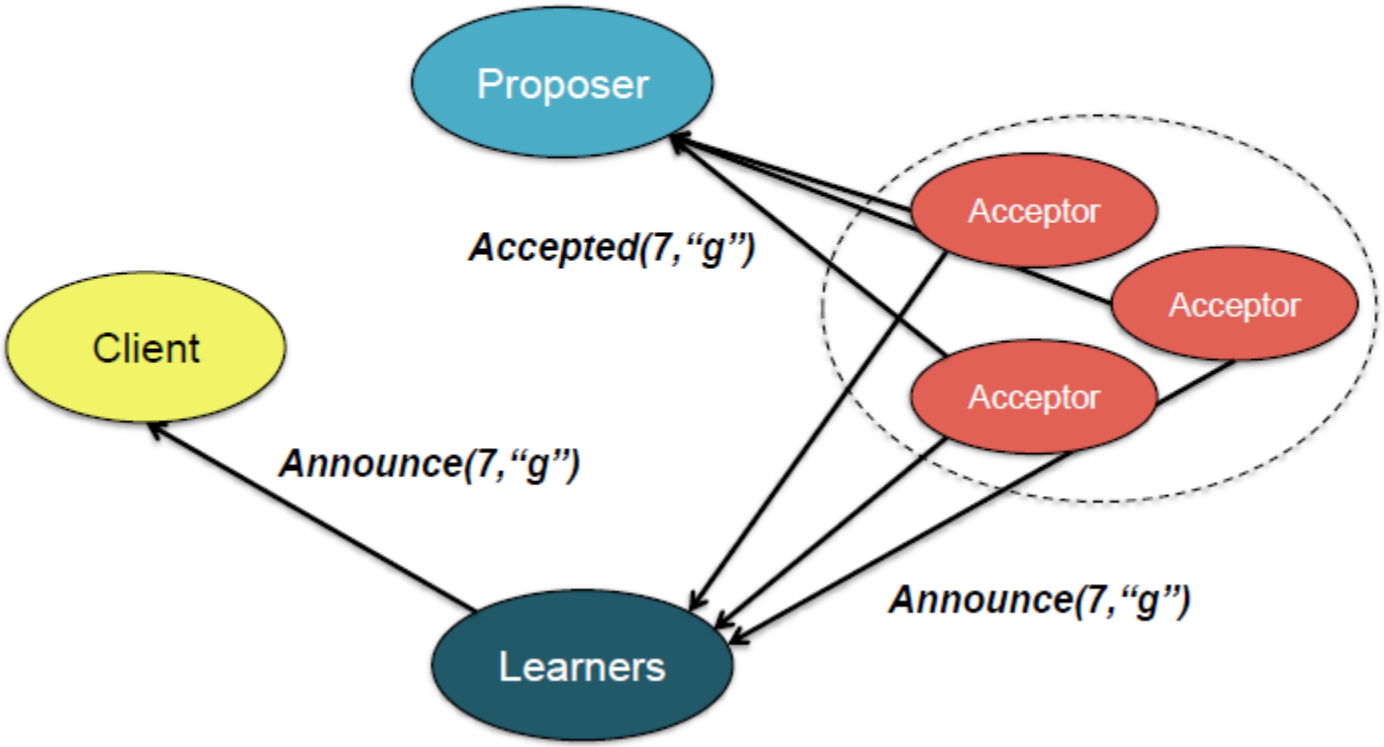
Acceptor: Suppose 5 is the highest sequence # any acceptor has seen
Each acceptor PROMISES not to accept any lower numbers



Proposer: Proposer receives the higher # offer and **MUST** change its mind and accept the highest offer that it received from any acceptor.



Acceptor: Acceptors state that they accepted the request. A learner can propagate this information



- A proposal *N may fail because*
  - The acceptor may have made a new promise to ignore all proposals less than some value *M* >*N*
  - A proposer does not receive a quorum of responses: either *promise* (phase 1b) or *accept (phase 2b)*
- Algorithm then has to be restarted with a higher proposal #

- Paxos allows us to ensure consistent (total) ordering over a set of events in a group of machines
  - Events = commands, actions, state updates
- Each machine will have the latest state or a previous version of the state
- Paxos used in:
  - Cassandra lightweight transactions
  - Google Chubby lock manager / name server
  - Google Spanner, Megastore
  - Microsoft Autopilot cluster management service from Bing
  - VMware NSX Controller
  - Amazon Web Services

Paxos Summary

- To make a change to the system:
  - Tell the *proposer (leader) the event/command you want to add*
    - Note: these requests may occur concurrently
    - Leader = one elected proposer. Not necessary for Paxos algorithm but an optimization to ensure a single, increasing stream of proposal numbers. Cuts down on rejections and retries.
  - The proposer picks its next highest event ID and asks all the acceptors to reserve that event ID
    - If any acceptor sees it's not the latest event ID, it rejects the proposal; the proposer will have to try again with another event ID
  - When the majority of acceptors accept the proposal, accepted events are sent to learners, which can act on them (e.g., update system state)
    - Fault tolerant: need *2k+1 servers for k fault tolerance*

References

- Andrew S. Tanenbaum and Maarten Van Steen. Distributed System Principles and Paragdims. 2007
- Paul Krzyzanowski. Distributed System Lectures Notes : Consensus Paxos. Rutgers University. Fall 2015