Distributed Systems

10. Quorum-Based Consensus: Paxos

Paul Krzyzanowski

Rutgers University

Fall 2018

Consensus Goal

Allow a group of processes to agree on a result

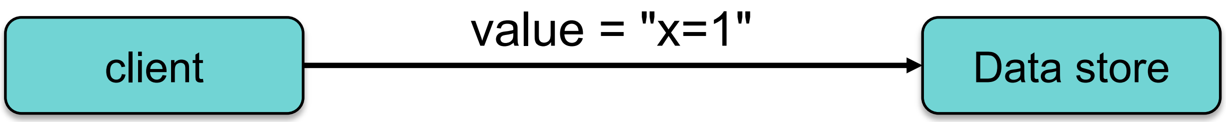
- All processes must agree on the same value
- The value must be one that was submitted by at least one process (the consensus algorithm cannot just make up a value)

We saw versions of this

- Mutual exclusion
  - Agree on who gets a resource or who becomes a coordinator
- Election algorithms
  - Agree on who is in charge
- Other uses of consensus:
  - Synchronize state to manage replicas: make sure every group member agrees on the message ordering of events
  - Manage group membership
  - Agree on distributed transaction commit
- General consensus problem:
  - *How do we get unanimous agreement on a given value?*

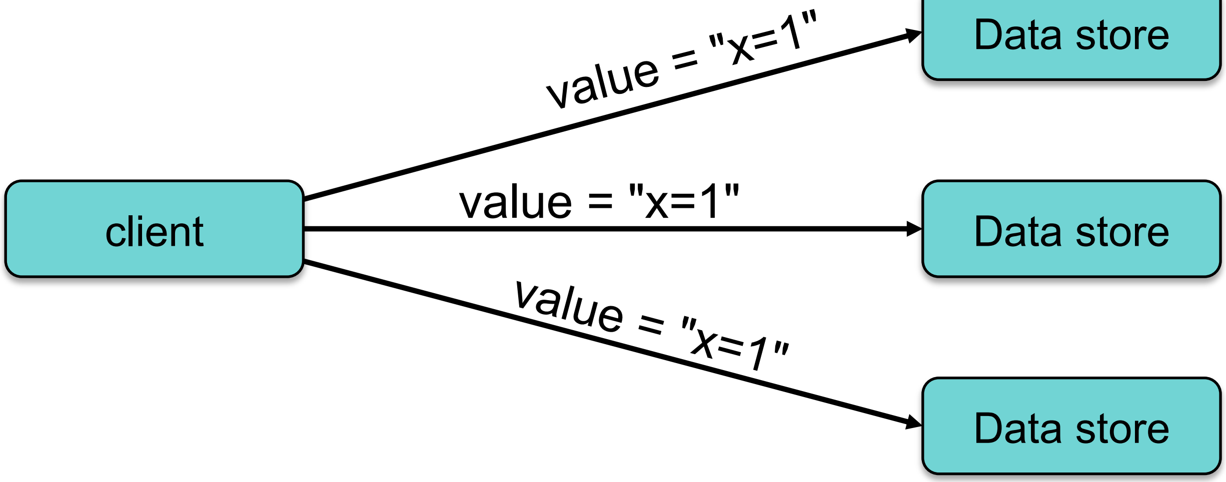
value = sequence number of a message, key=value, operation, whatever…

Achieving consensus seems easy!



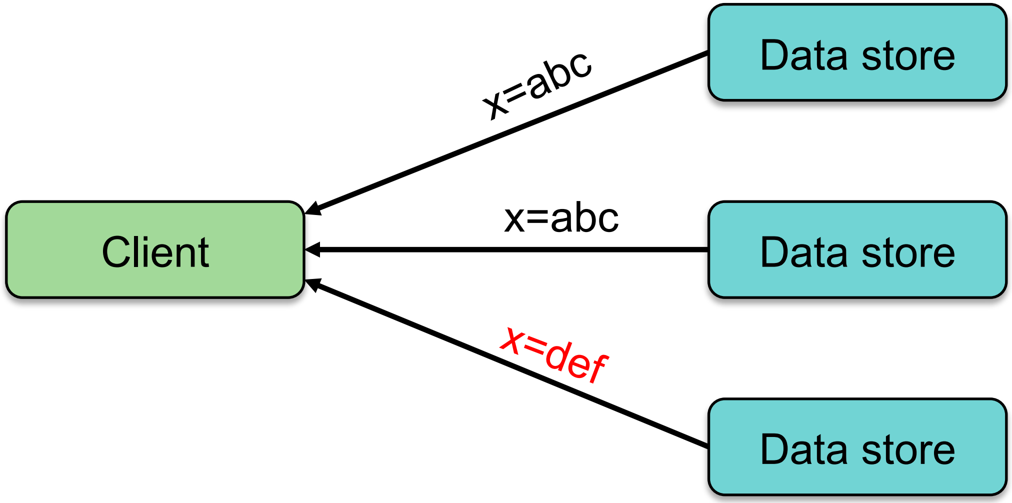
- One request at a time
- Server that never dies

Servers might die – let's add replicas



- One request at a time

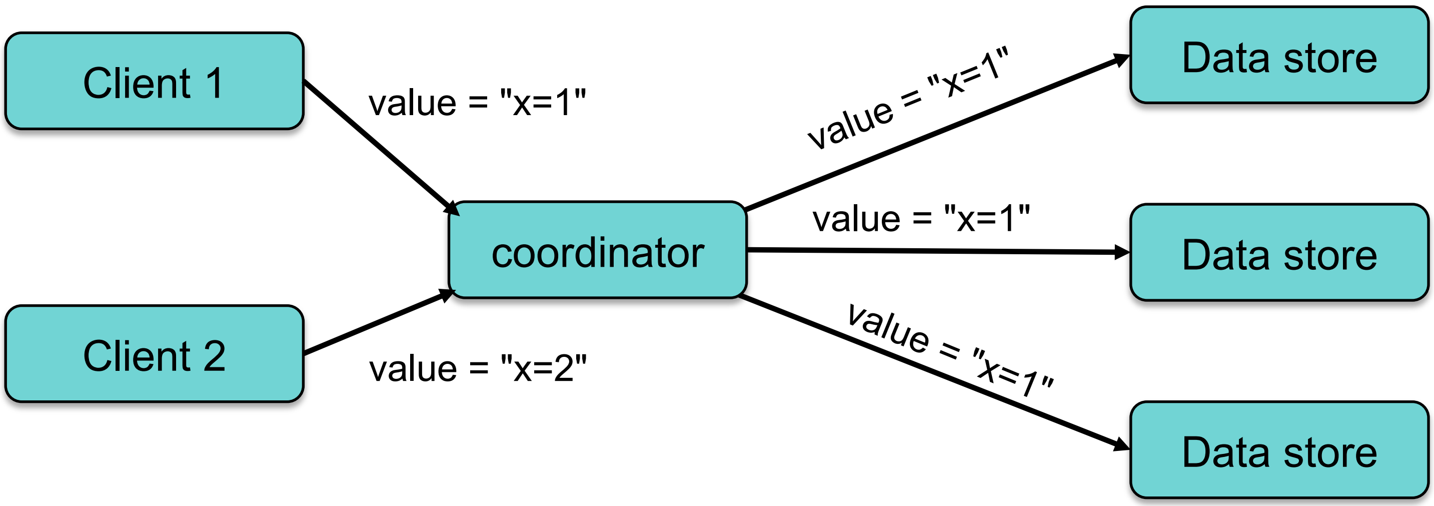
Reading from replicas is easy



We rely on a quorum (majority) to read successfully

No quorum = failed read!

What about concurrent updates?



- Coordinator processes requests one at a time
- But now we have a single point of failure!
- We need something safer

Consensus algorithm goal

- Create a fault-tolerant consensus algorithm that does not block if a majority of processes are working

- Goal: agree on one result among a group of participants
  - Processors may fail (some may need stable storage)
  - Messages may be lost, out of order, or duplicated
  - If delivered, messages are not corrupted

Quorum: majority (>50%) agreement is the key part: If a majority of coins show heads, there is no way that a majority will show tails at the same time.

If members die and others come up, there will be one member in common with the old group that still holds the information.

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

Consensus algorithm: Paxos

Paxos

Goal: provide a consistent ordering of events from multiple clients

- All machines running the algorithm **agree on a proposed value** from a client
- The *value* will be associated with an event or action
- Paxos ensures that no other machine associates the value with another event

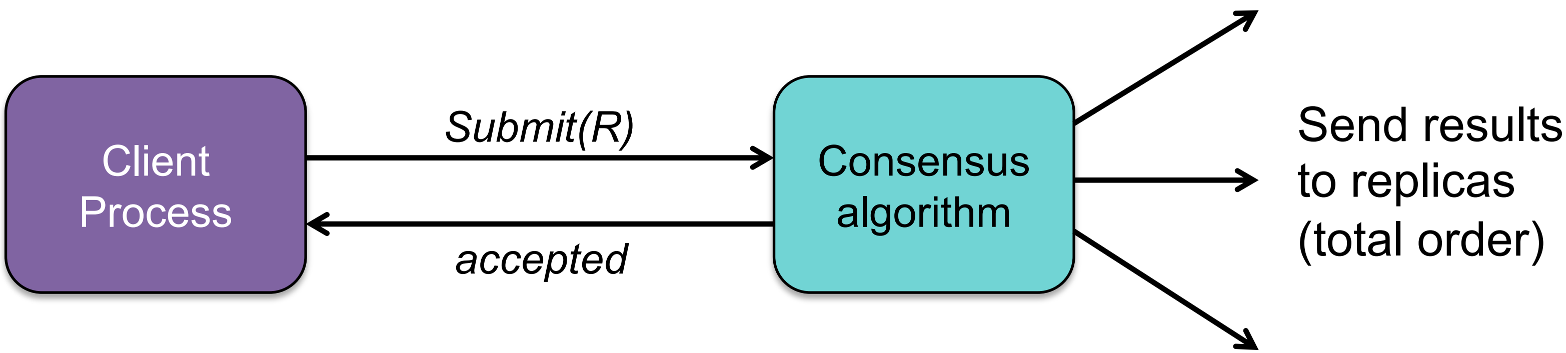
Fault-tolerant distributed consensus algorithm

- Does not block if a majority of processes are working
- The algorithm needs a majority (*2P+1*) of processors survive the simultaneous failure of *P* processors

Paxos provides **abortable consensus**

- A client's request may be rejected
- It then has to re-issue the request

A Programmer's View



If your request is not accepted, you can submit it again later:

```
while (submit_request(R) != ACCEPTED) ;
```

Think of R as a key:value pair in a database where multiple clients might want to modify the same key

Paxos players

- **Client**: makes a request
- **Proposers**:
  - Get a request from a client and run the protocol to get everyone in the cluster to agree
  - Leader: elected coordinator among the proposers
    (not necessary but simplifies message numbering and ensures no contention) – we don't need to rely 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

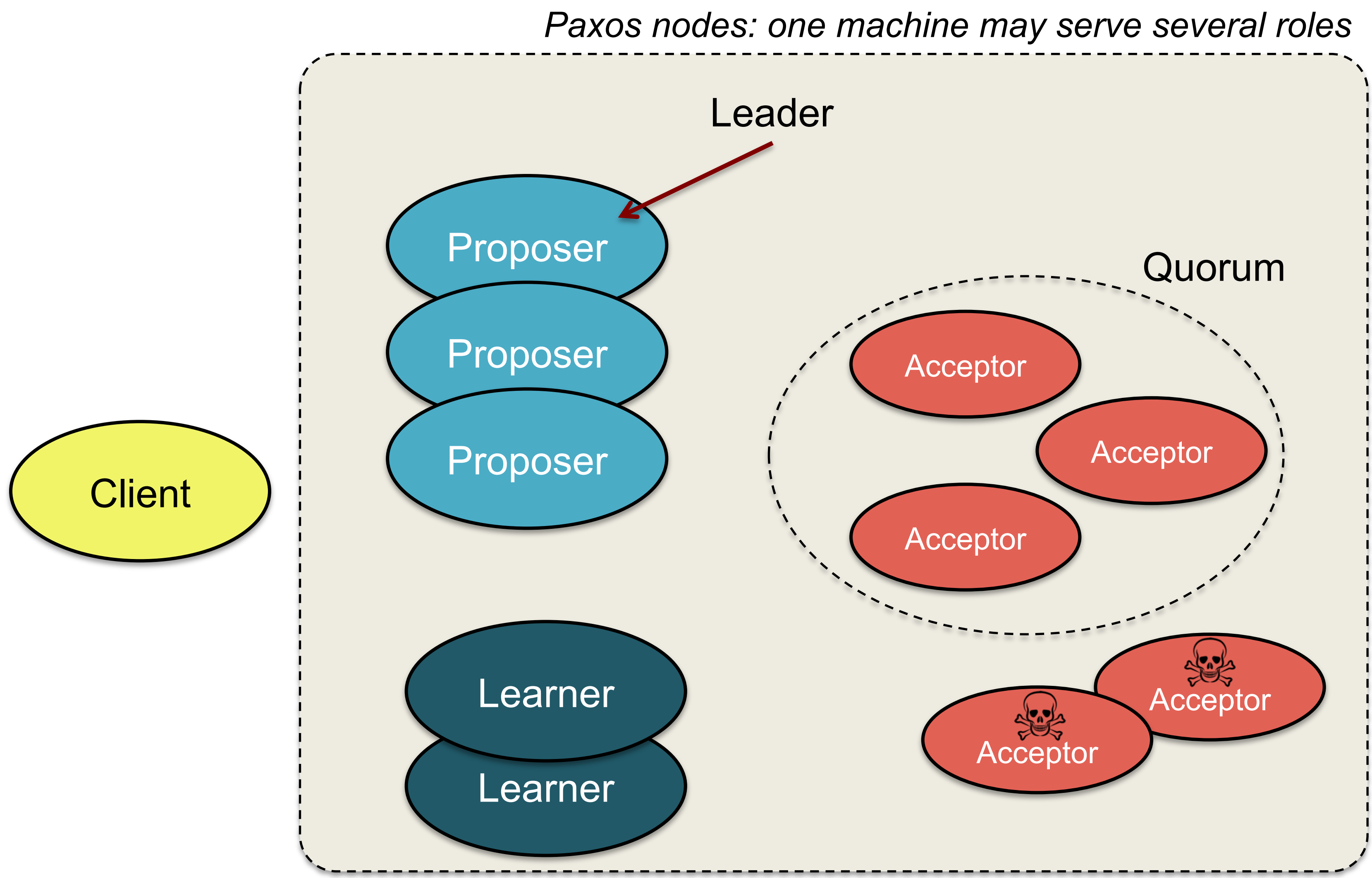
Proposal numbers

- **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 proposal number (event ID) it knows about
    - ID = sequence number
  - Asks all the acceptors to reserve that proposal #
- **Acceptors**
  - A majority of acceptors have to accept the requested proposal #

Proposal Numbers

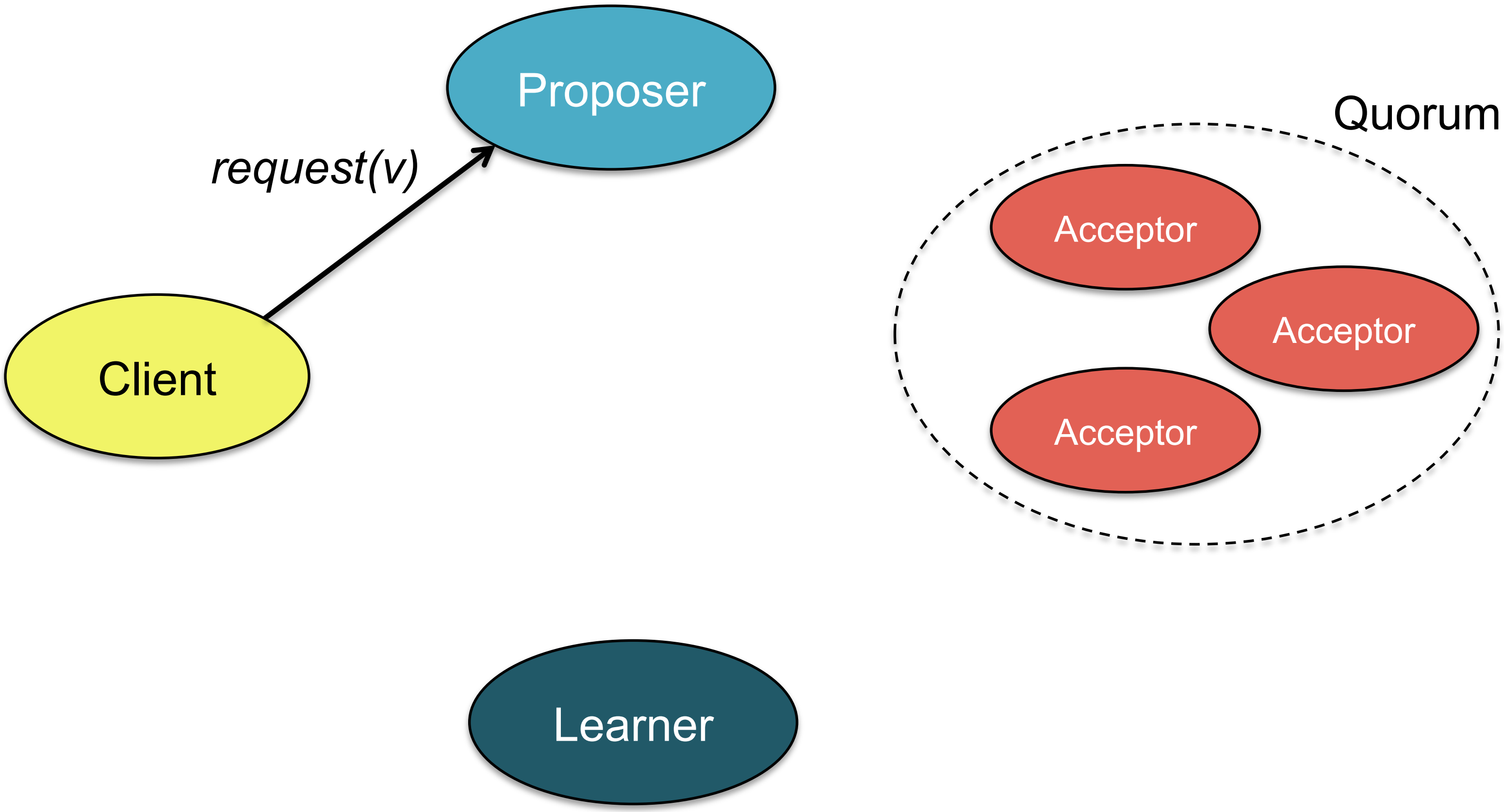
- Each proposal has a unique number (created by proposer)
  - Must be unique (e.g., <sequence #>.<process_id>)
- Newer proposals take precedence over older ones
- Each acceptor
  - Keeps track of the largest number it has seen so far
  - Lower proposal numbers get rejected
    - Acceptor sends back the *{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 in action



Paxos in action: Phase 0

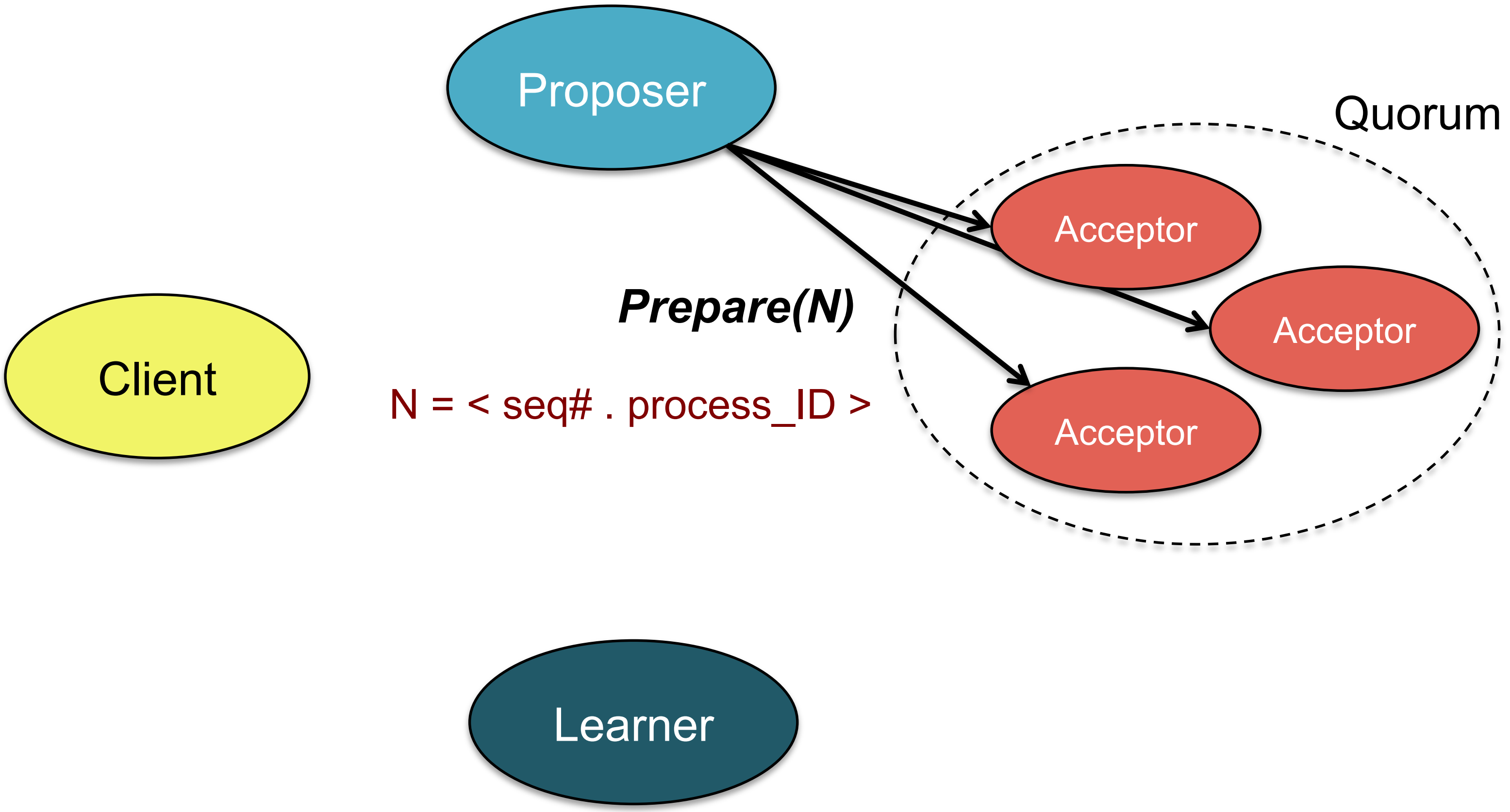
Client sends a request to a proposer



Paxos in action: Phase 1a – *PREPARE*

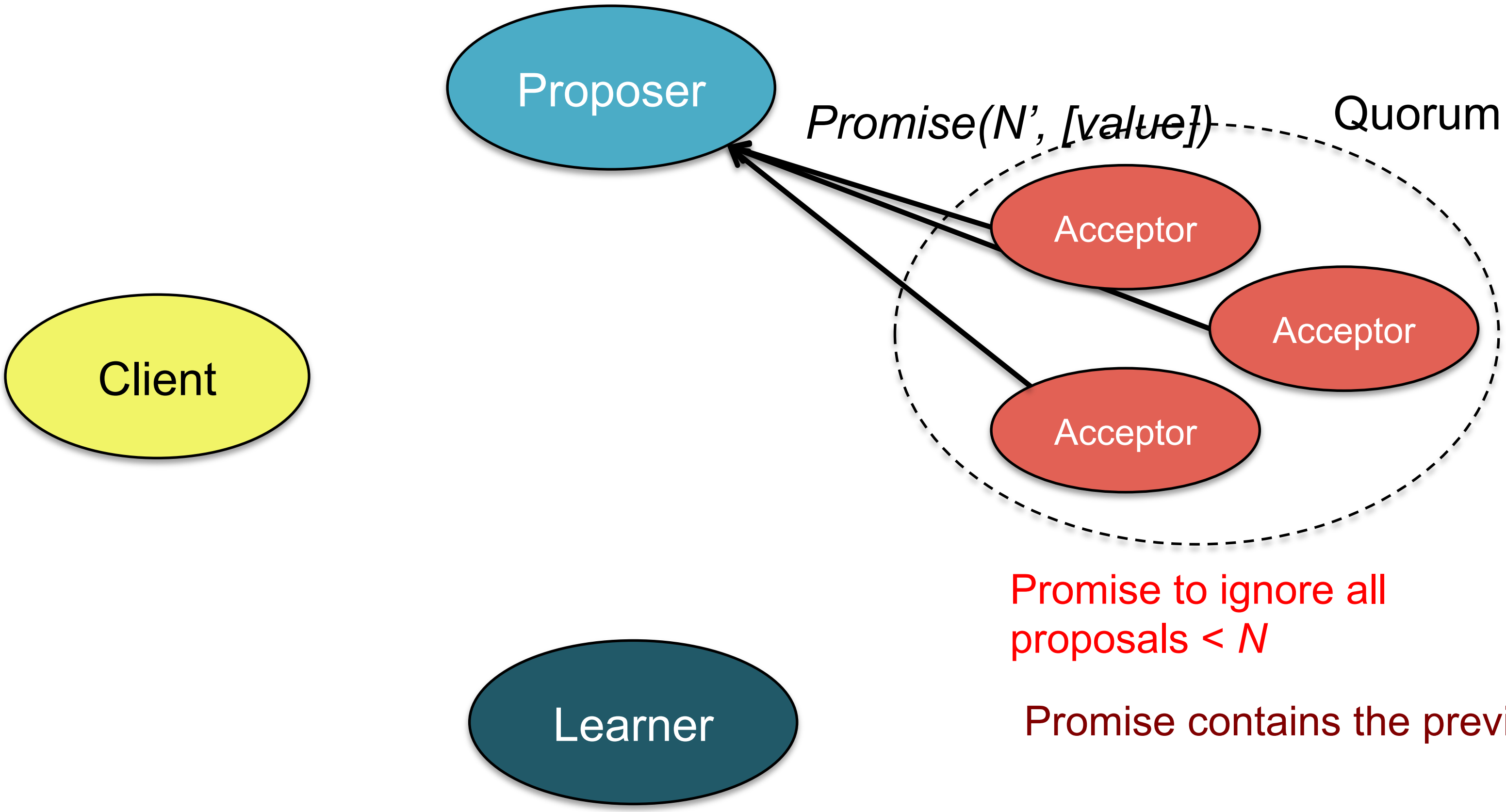
Proposer: creates a *proposal #N (N acts like a Lamport time stamp)*, where *N* is greater than any previous proposal number used by this proposer

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



Paxos in action: Phase 1b – *PROMISE*

Acceptor: if proposer's ID > any previous proposal
promise to ignore all requests with IDs < N
reply with info about highest past proposal: { *N', value* }



Promise to ignore all proposals < *N*

Promise contains the previous *N*

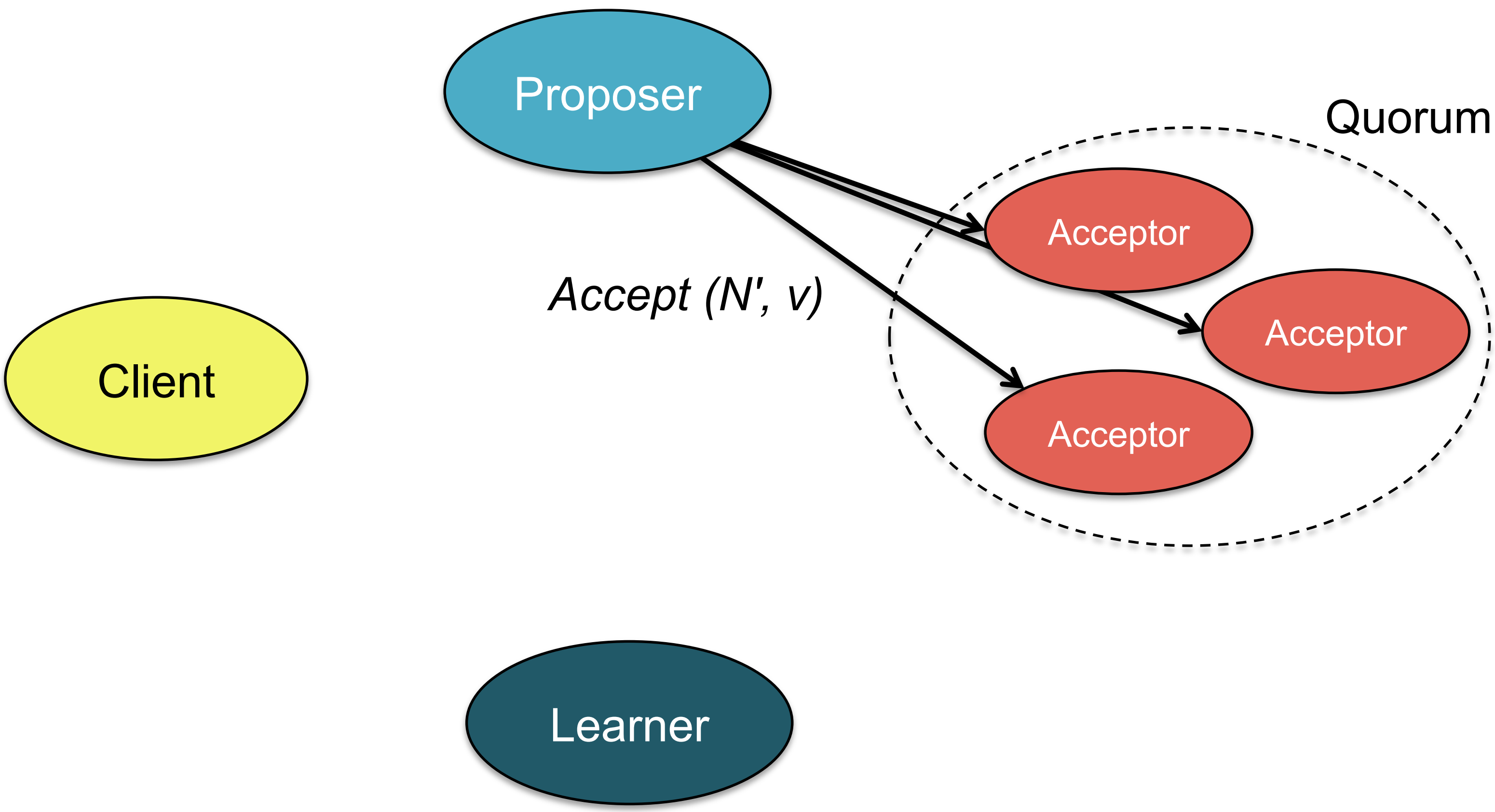
Paxos in action: Phase 2a – *ACCEPT*

Proposer: if proposer receives promises from the quorum (majority):

Attach a value *v* to the proposal (the event).

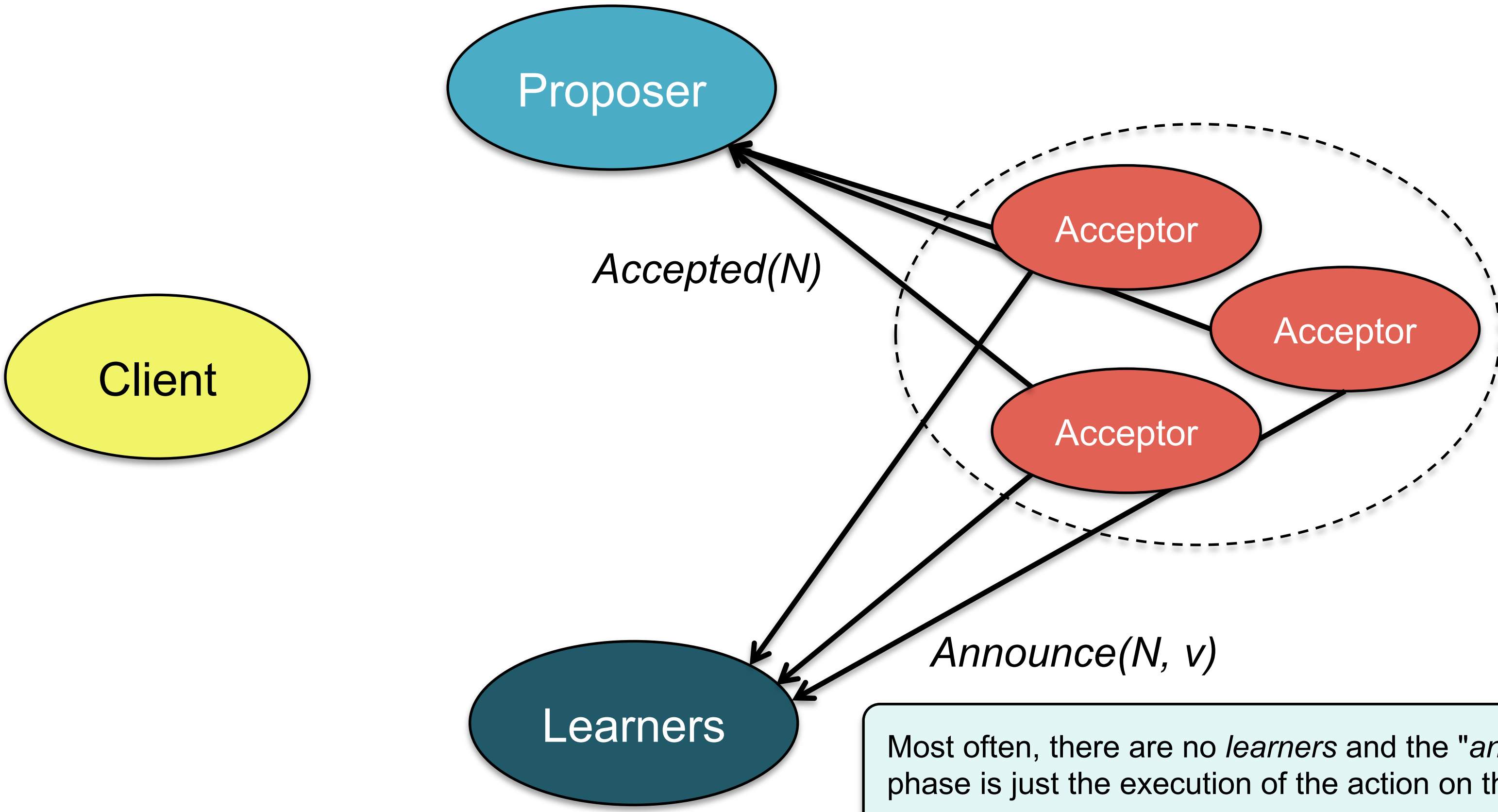
Send ***Accept*** to quorum with the **chosen** value

If promise was for another {*N′, v*}, proposer MUST accept *v* for the highest accepted proposal



Paxos in action: Phase 2b – *ANNOUNCE*

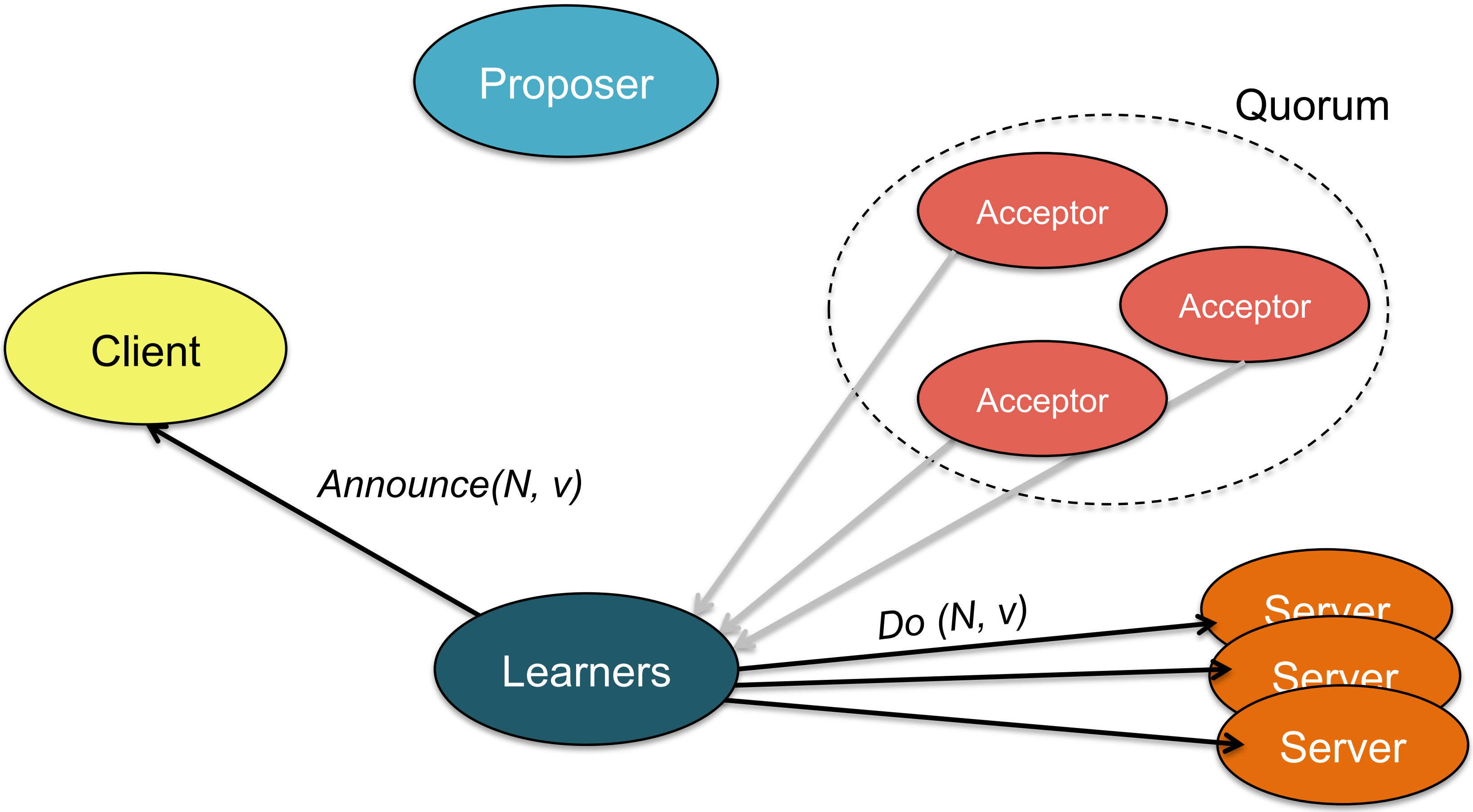
Acceptor: if the promise still holds, then announce the value *v*
Send ***Accepted*** message to Proposer and every Learner
BUT: if a higher proposal # may have been received during this time
then send *NACK* to proposer so it can try again



Most often, there are no *learners* and the "*announce*" phase is just the execution of the action on the local server

Paxos in action: Phase 3 – *ANNOUNCE*

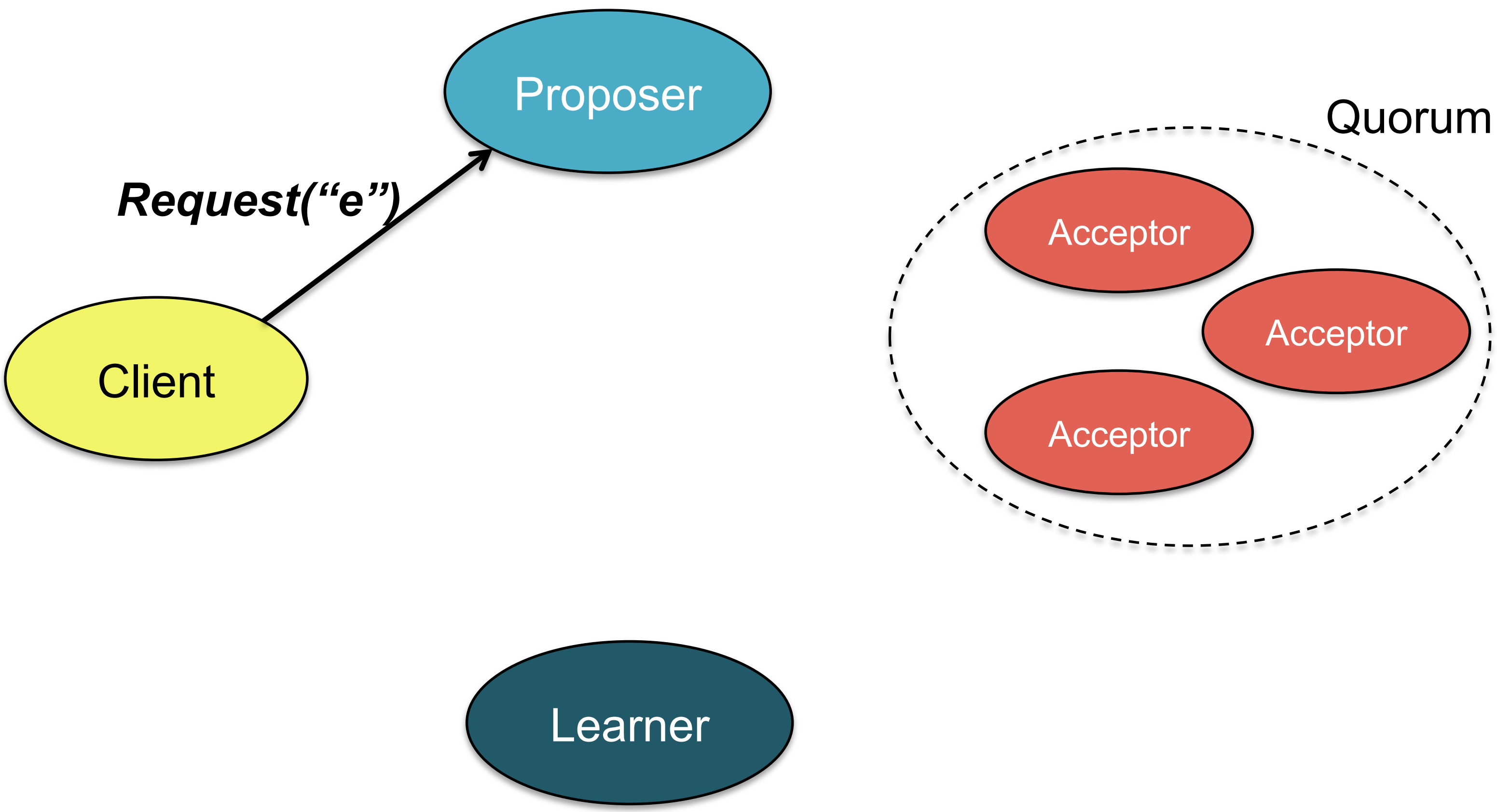
Learner: Respond to client and/or take action on the request



Paxos: A Simple Example – All Good

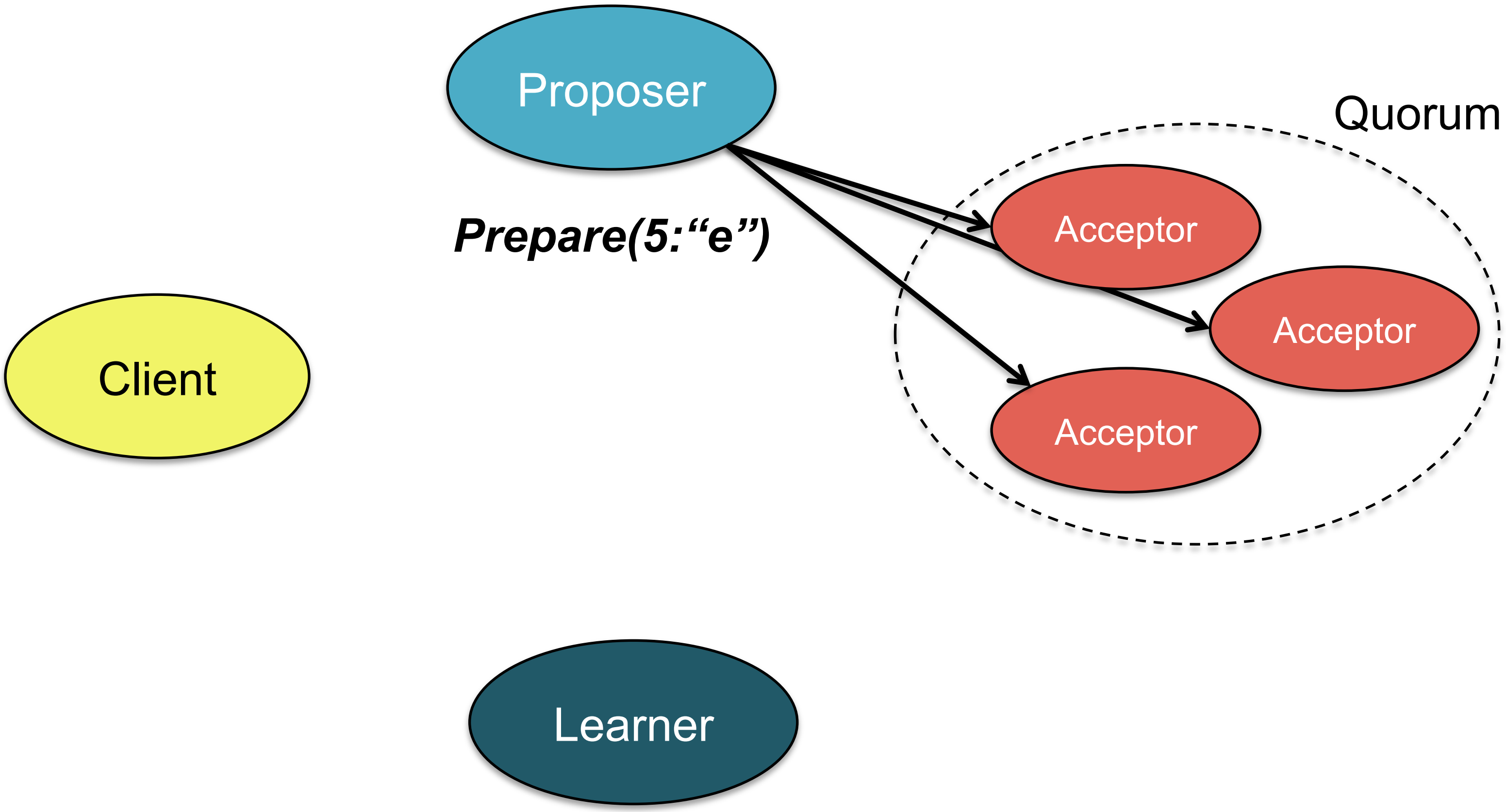
Paxos in action: Phase 0

Client sends a request to a proposer



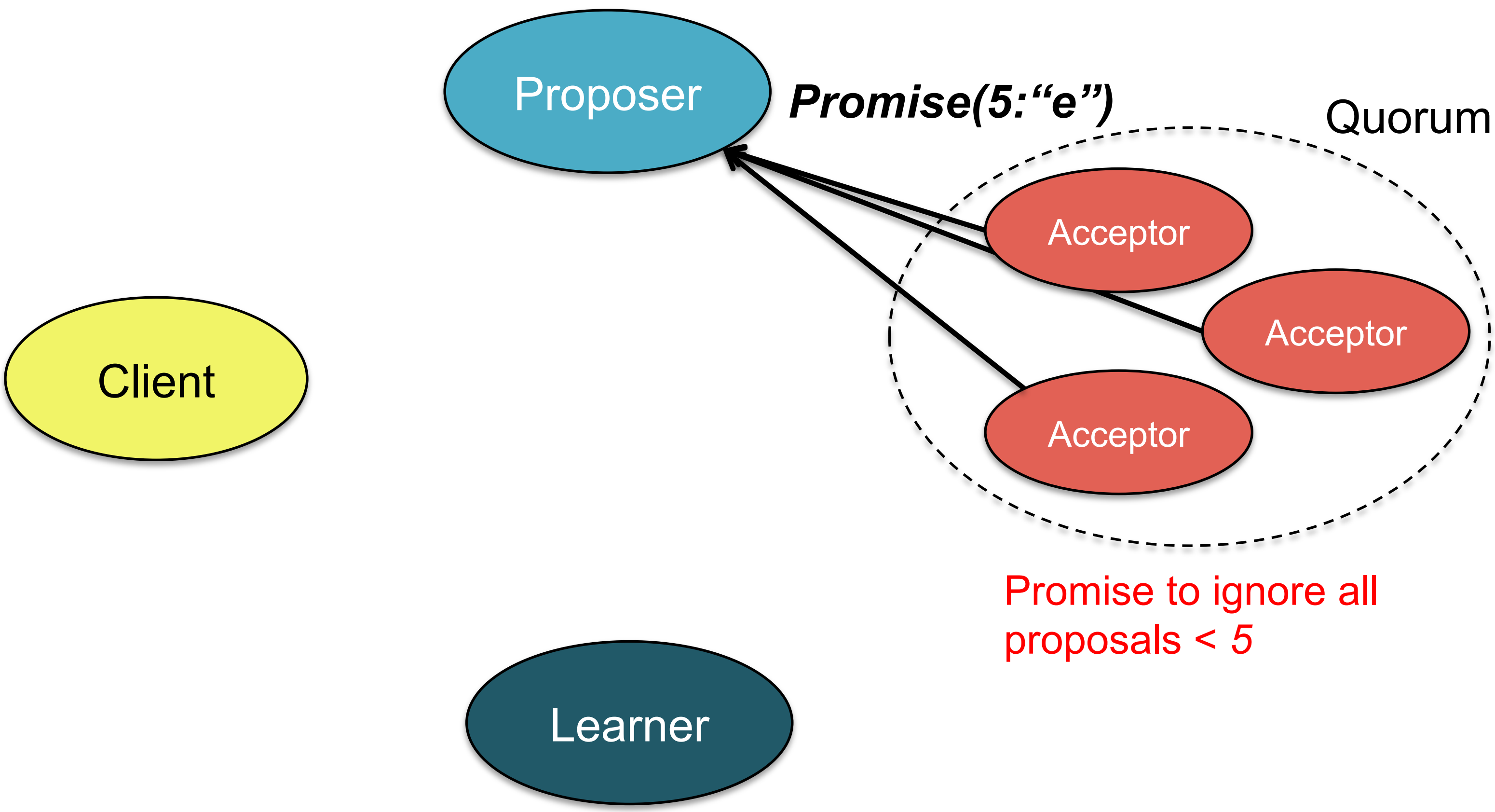
Paxos in action: Phase 1a – *PREPARE*

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



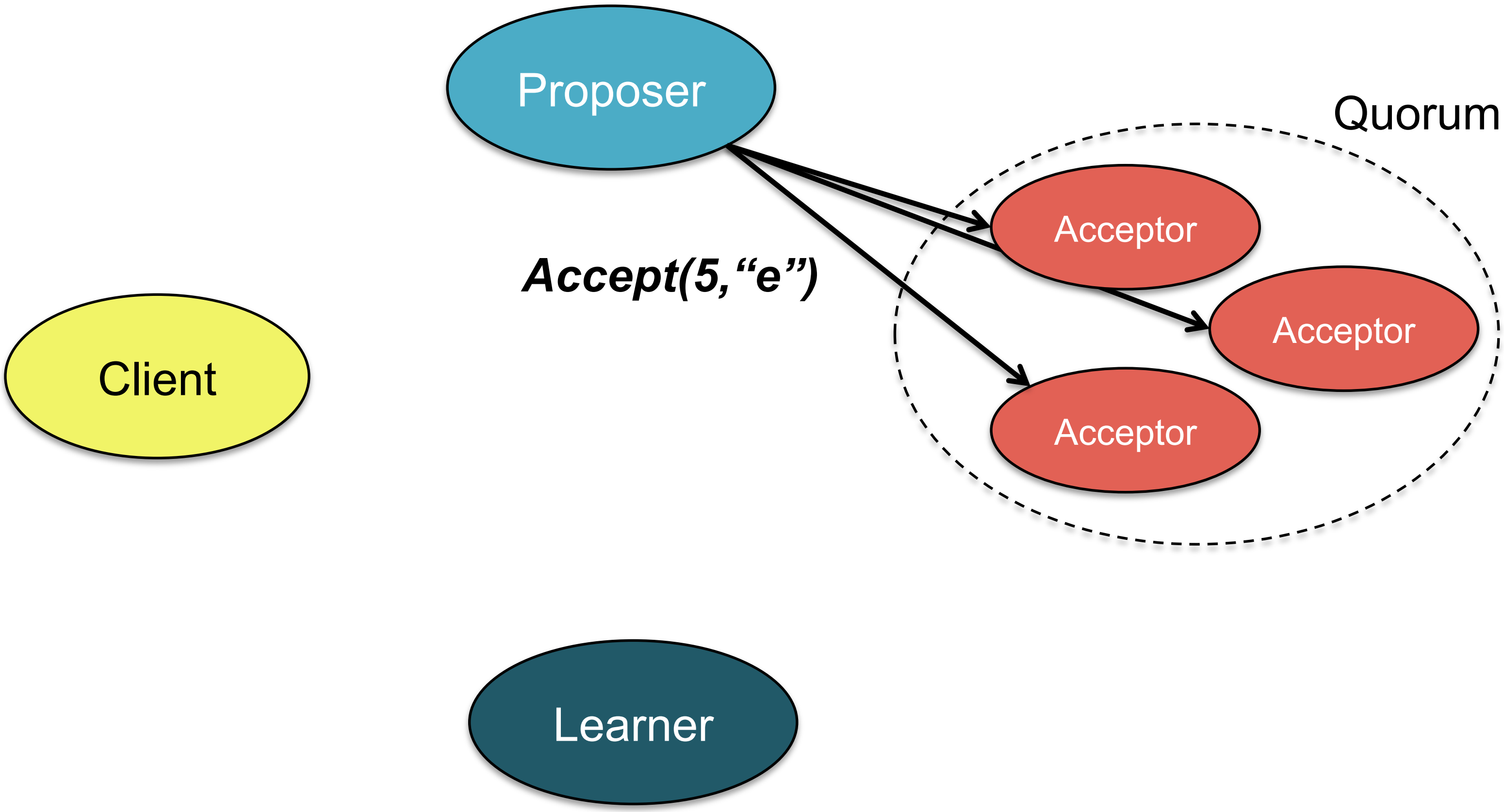
Paxos in action: Phase 1b – *PROMISE*

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



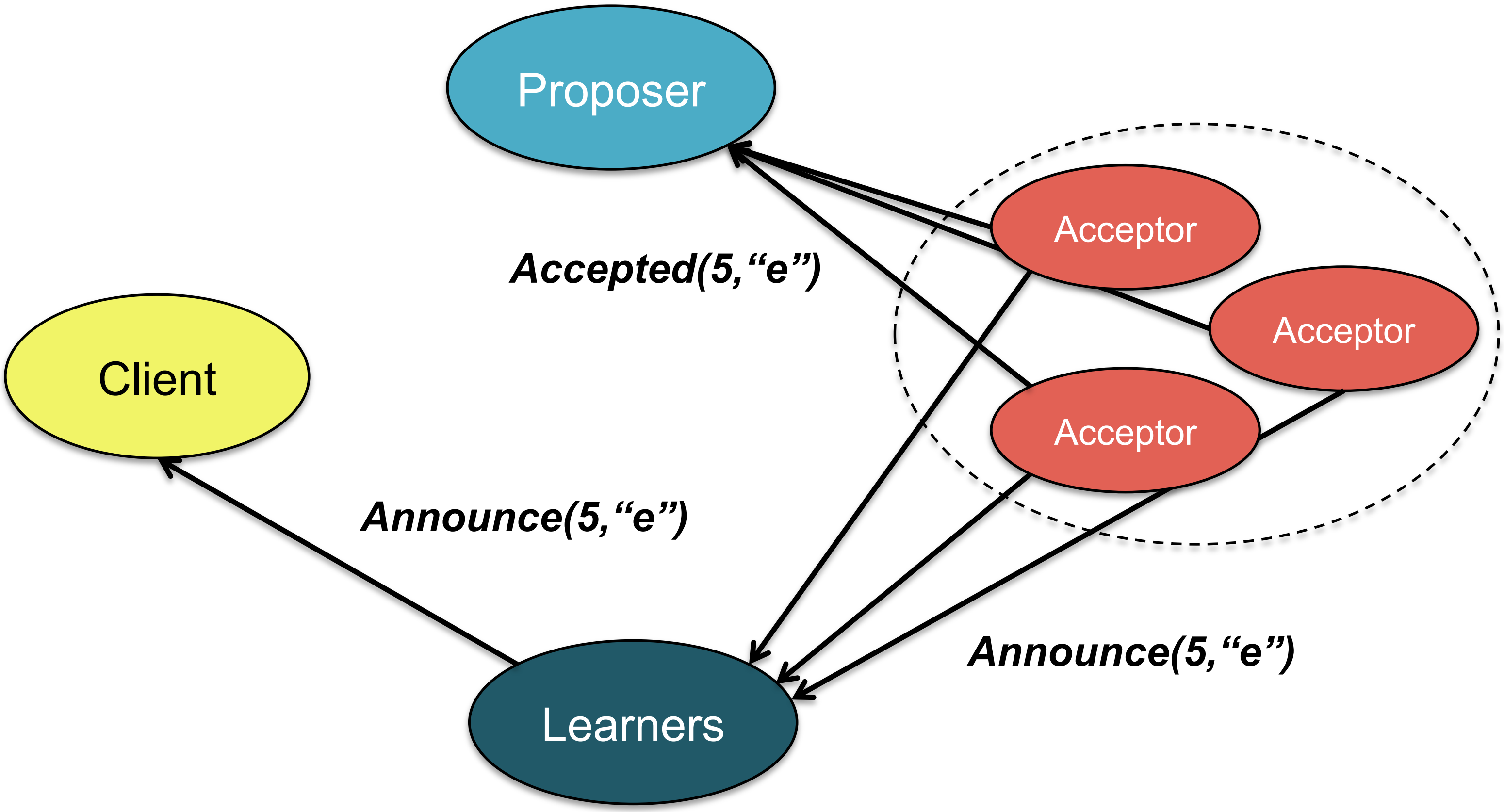
Paxos in action: Phase 2a – *ACCEPT*

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



Paxos in action: Phase 2b – *ANNOUNCE*

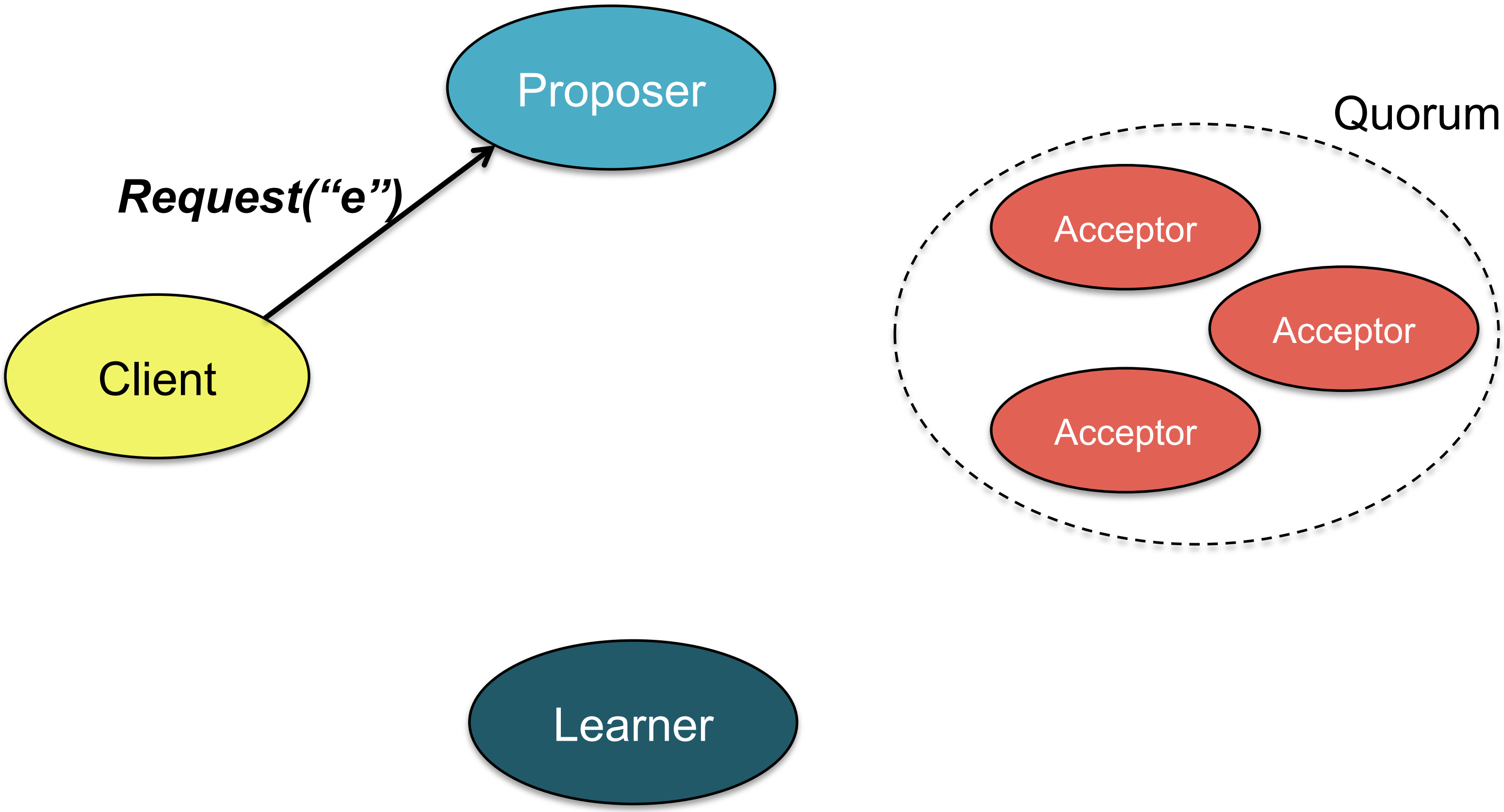
Acceptor: Acceptors state that they accepted the request



Paxos: A Simple Example – Higher Offer

Paxos in action: Phase 0

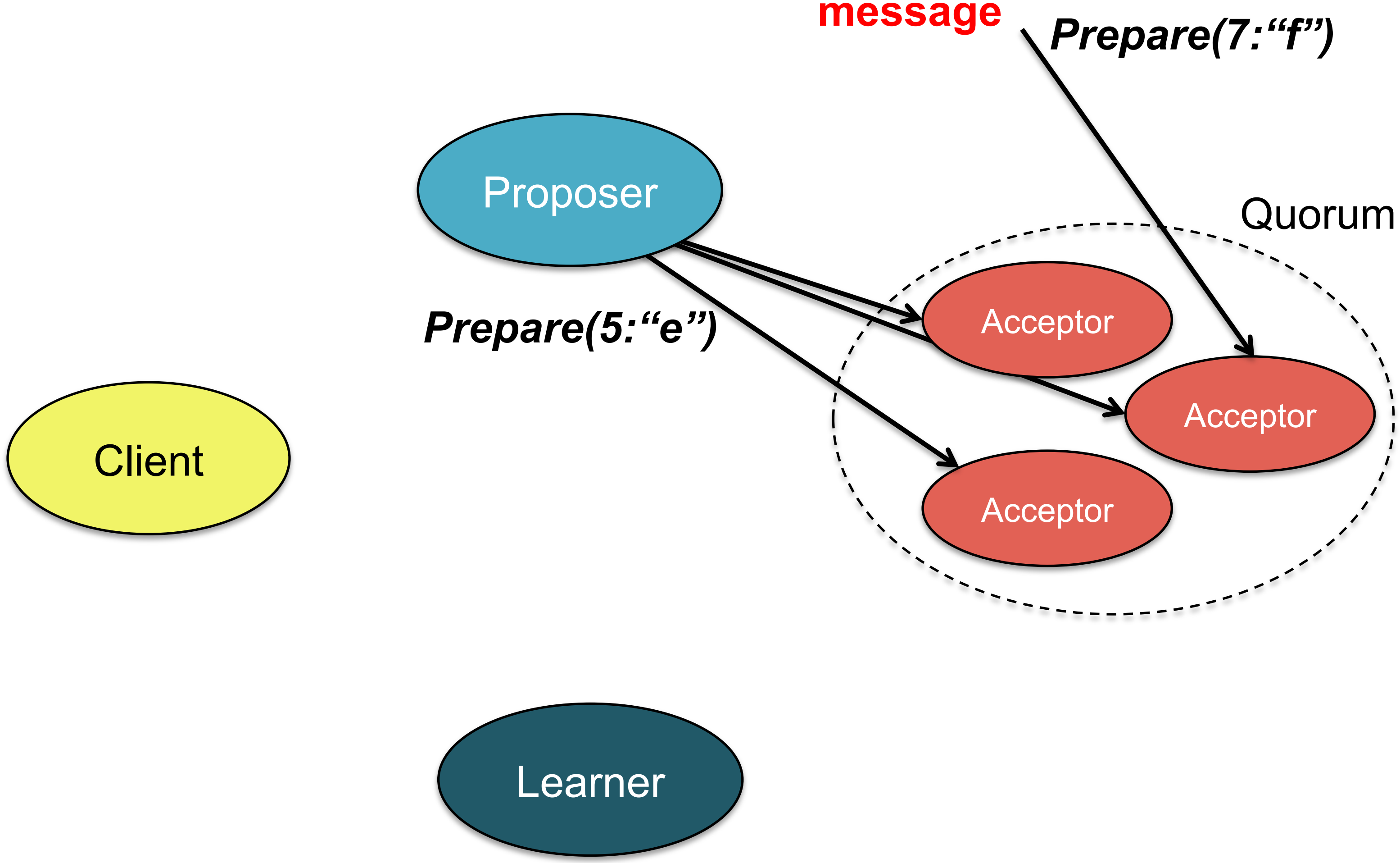
Client sends a request to a proposer



Paxos in action: Phase 1a – *PREPARE*

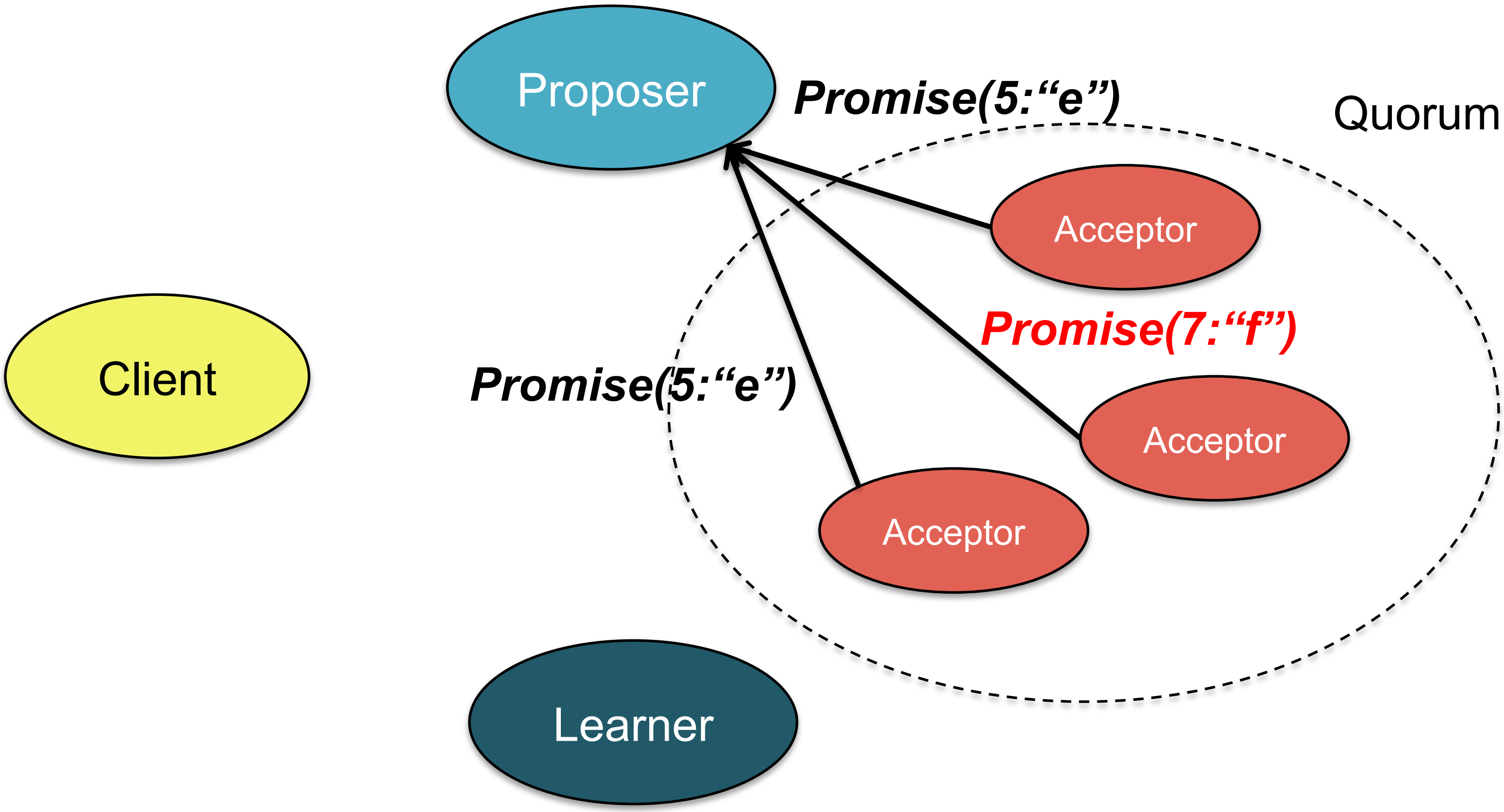
Proposer: picks a sequence number: 5
Send to Quorum of Acceptors

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



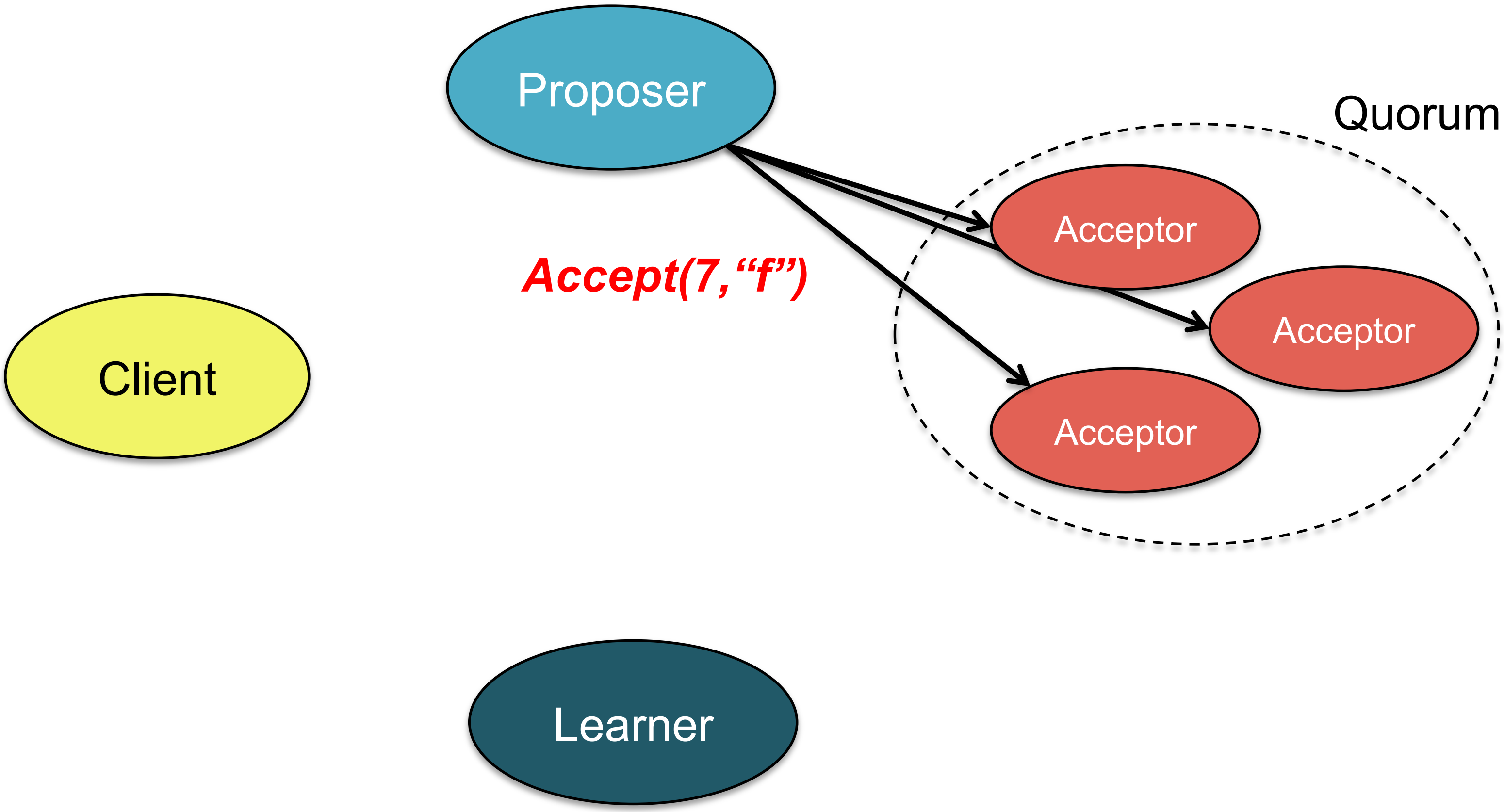
Paxos in action: Phase 1b – *PROMISE*

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



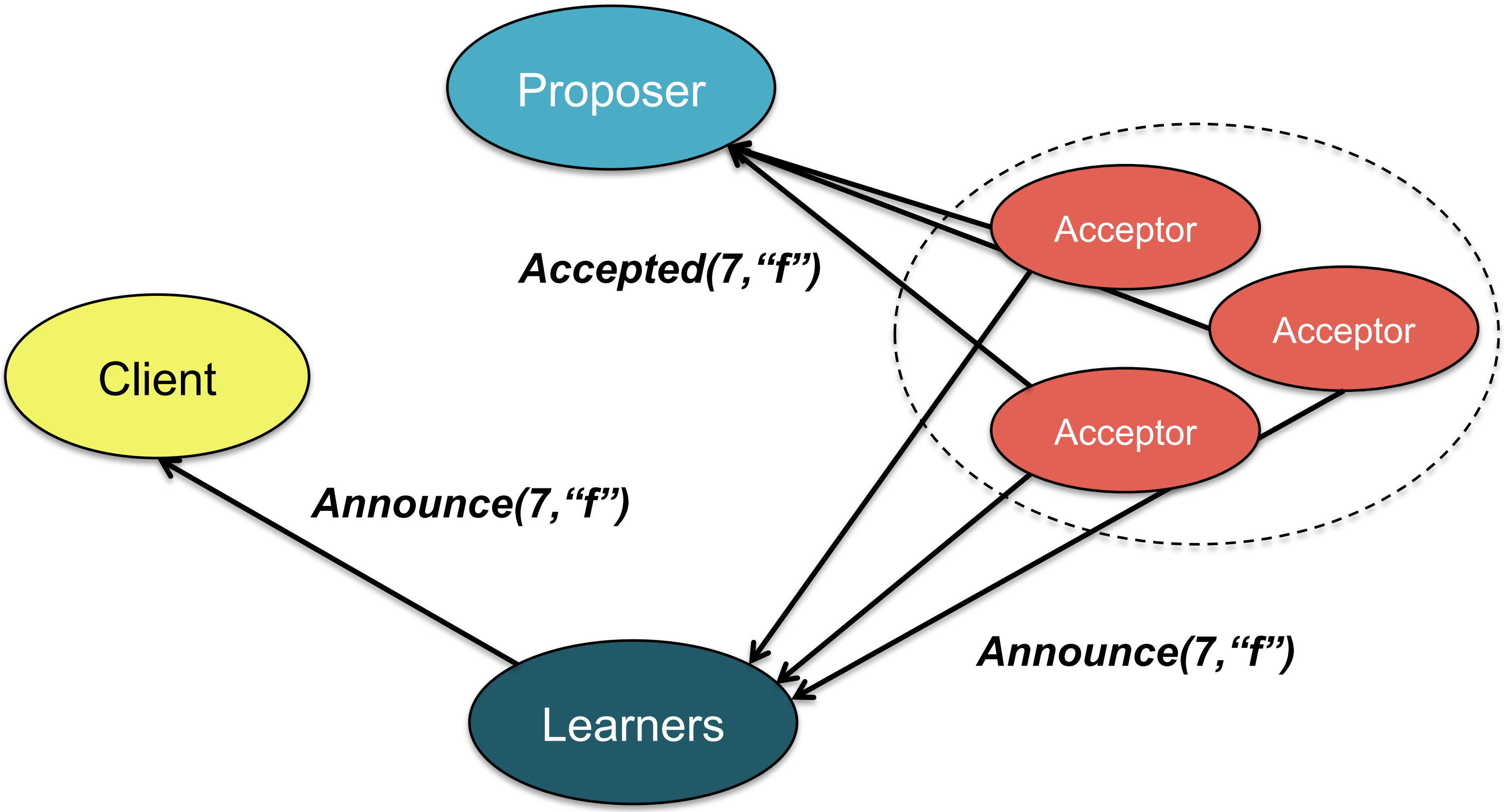
Paxos in action: Phase 2a – *ACCEPT*

Proposer: Proposer receives the higher # offer and **MUST** change its mind



Paxos in action: Phase 2b – *ANNOUNCE*

Acceptor: Acceptors state that they accepted the request



Paxos: Keep trying if you need to

- 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 summary

- 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:
  - Google Chubby lock manager / name server
  - Apache Zookeeper (clone of Google Chubby)
  - Cassandra lightweight transactions
  - Google Spanner, Megastore
  - Microsoft Autopilot cluster management service from Bing
  - VMware NSX Controller
  - Amazon Web Services, DynamoDB

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 has seen a higher event ID, it rejects the proposal & returns that higher event ID
  - 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

Implementation

- Use only one proposer at a time – the leader
  - Other nodes can be active backups just in case the leader dies
  - No need to worry about sync of proposal # – those are local per proposer
  - Acts like a fault-tolerant coordinator
    - Avoids failed proposals due to higher numbers from other proposers
- Alternatively, embed proposer logic into client library
  - Too many clients issuing concurrent requests can cause a large # of retries
- Learners rarely needed
  - Acceptors are often running on the system that processes the request (e.g., data store, log, …)
  - Just send an acknowledgement directly to the client.

The End