CSCI 5673
Distributed Systems

Lecture Set Seven

Paxos
(Distributed transactions, Quorums)

Lecture Notes by
Shivakant Mishra
Computer Science, CU Boulder
Last update: February 23, 2017

Acknowledgement: *Some of these slides are adapted from Prof. Ken Birman's Cloud Computing course (CS 5412 Spring 2012)*

Island of Paxos

Reference: Lamport. The Part-Time Parliament. ACM TOCS, May 1998.

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators.

- No one in Paxos was willing to devote his life to Parliament
- The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary chamber
- Bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems

Read the paper for details …

Leslie Lamport's vision

- Centers on *state machine replication*
  - We have a set of replicas that each implement some given, deterministic, state machine and we start them in the same state
  - Now we apply the same events in the same order. The replicas remain in the identical state
  - To tolerate ≤ *t* failures, deploy 2*t+1* replicas (e.g. Paxos with 3 replicas can tolerate 1 failure)
- How best to implement this model?

Two paths forwards...

- One option is to build a totally ordered reliable multicast protocol, also called an “atomic broadcast” protocol in some papers
  - To send a request, you give it to the library implementing that protocol (Examples: Isis, Totem, ...).
  - Eventually it does *upcalls* to event handlers in the replicated application and they apply the event
  - In this approach the application “is” the state machine and the multicast “is” the replication mechanism
- Use “state transfer” to initialize a joining process if we want to replace replicas that crash

Two paths forwards...

- A second option, explored in Lamport's Paxos protocol, achieves a similar result but in a very different way

- We'll look at Paxos first because the basic protocol is simple and powerful, but we'll see that Paxos is slow
  - Can speed it up... but doing so makes it very complex!
  - The basic, slower form of Paxos is currently very popular
- Reference: Lamport. Paxos Made Simple. ACM SIGACT 32 (4), December 2001.

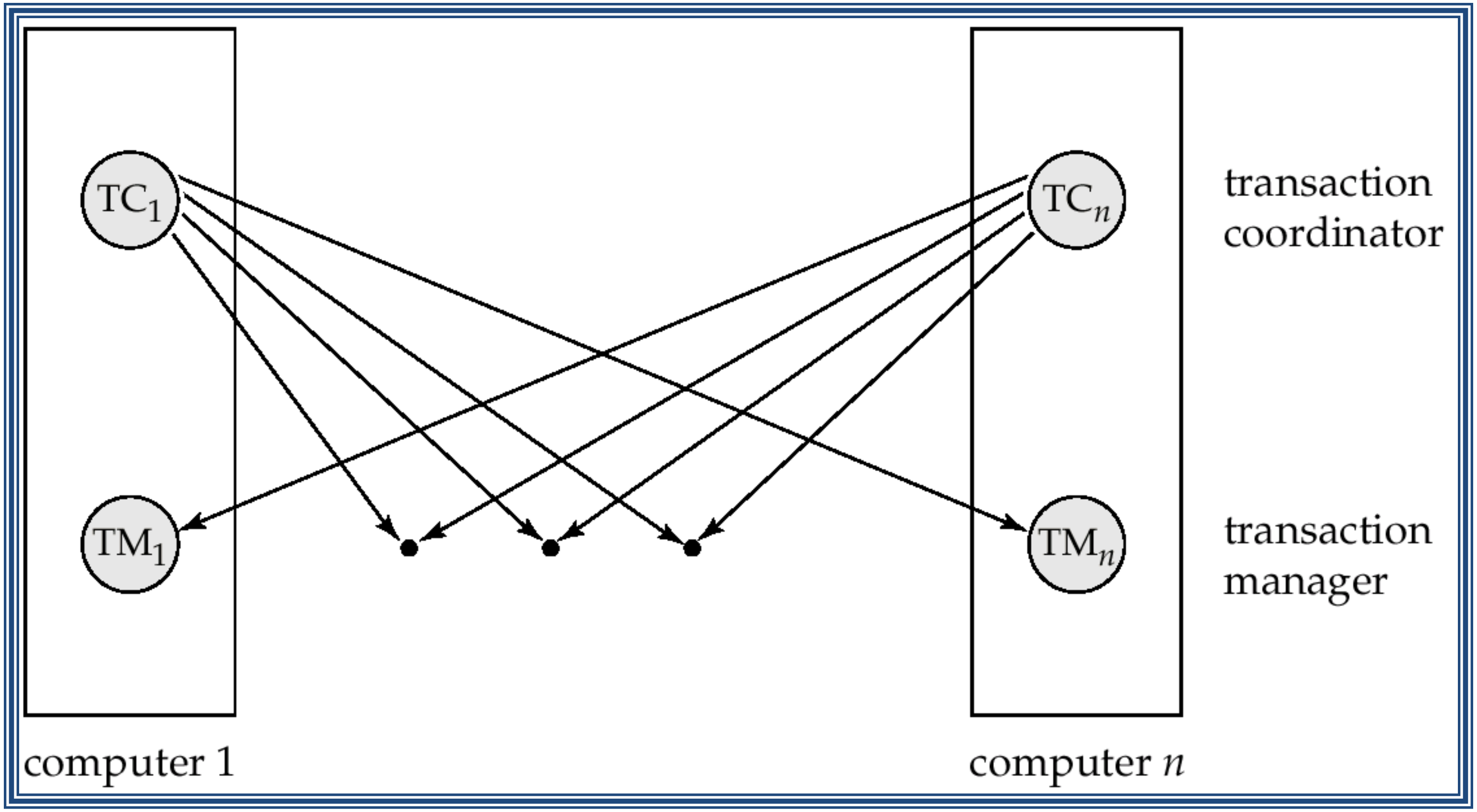
Paxos

- Uses two technologies
  - 2 Phase Commit Protocol (part of Distributed Transactions)
  - Distributed Quorum

Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local transaction manager:
  - Maintains a log for recovery purposes
  - Participates in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction coordinator:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions to sites for execution.
  - Coordinating the termination of each transaction that originates at the site: May result in the transaction being committed at all sites or aborted at all sites.

Transaction System Architecture



Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2 *PC*) protocol is widely used
- The *three-phase commit* (3 *PC*) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
    - add the record <**ready** *T*> to the log
    - force *all records* for *T* to stable storage
    - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- *T* can be committed of C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: site executes **redo** (*T*)
- Log contains <**abort** *T*> record: site executes **undo** (*T*)
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- The log contains no control records concerning *T* replies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_i must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.

2 Phase Commit

- **Blocking problem** : active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.

Handling of Failures - Network Partition

- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Three Phase Commit (3PC)

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most K sites (participants as well as coordinator) can fail
- Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1.
  - Every site is ready to commit if instructed to do so

Three Phase Commit (3PC)

- Phase 2 of 2PC is split into 2 phases, Phase 2 and Phase 3 of 3PC
  - In phase 2 coordinator makes a decision as in 2PC (called the pre-commit decision) and records it in multiple (at least K) sites
  - In phase 3, coordinator sends commit/abort message to all participating sites
- Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure
  - Avoids blocking problem as long as < K sites fail
- Drawbacks:
  - higher overheads
  - assumptions may not be satisfied in practice
- Won’t study it further

Distributed Quorums

- Starts with a simple observation:
  - Suppose that we lock down the membership of a system: It has replicas {P, Q, R, ... }
  - But sometimes, some of them can't be reached in a timely way.
  - How can we manage replicated data in this setting?
- Updates would wait, potentially forever!
- If a Read sees a copy that hasn't received some update, it returns the wrong value

Quorum policy: Updates (writes)

- To permit progress, allow an update to make progress without waiting for all the copies to acknowledge it.
  - Instead, require that a “write quorum” (or update quorum) must participate in the update
  - Denote by $Q_{W.}$ For example, perhaps Q_W=N-1 to make progress despite 1 failure (assumes N>1, obviously)
  - Can implement this using a 2-phase commit protocol

- With this approach some replicas might “legitimately” miss some updates. How can we know the state?

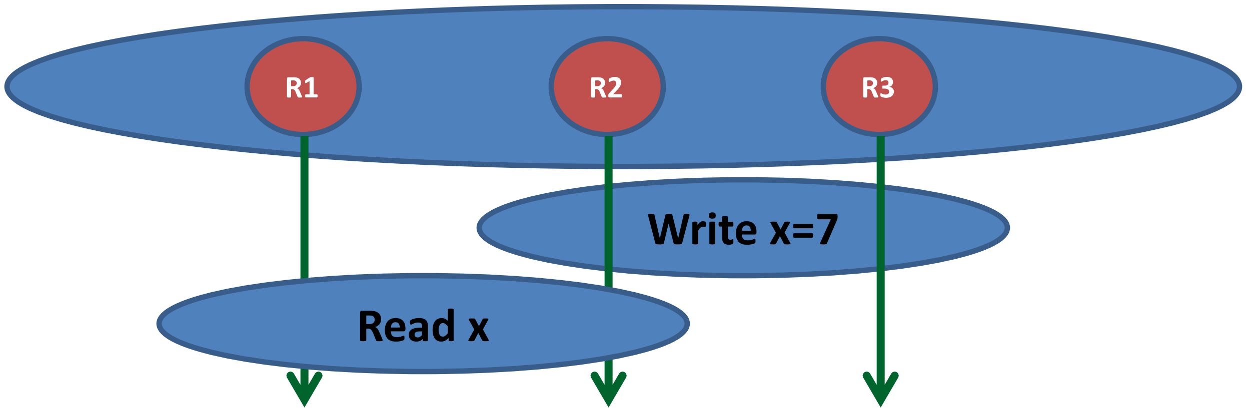
Quorum policy: Reads

- To compensate for the risk that some replicas lack some writes, we must read multiple replicas
  - … enough copies to compensate for gaps

- Accordingly, we define the read quorum, Q_R to be large enough to overlap with any prior update that was successful. E.g. might have $Q_R = 2$

Verify that they overlap

- So: we want
  - $Q_W + Q_R > N$: Read overlaps with updates
  - $Q_W + Q_W > N$: Any two writes, or two updates, overlap
- The second rule is needed to ensure that any pair of writes on the same item occur in an agreed order



$N = 3$
$Q_W = 2$
$Q_R = 2$

Reference: K. Gifford. Weighted Voting with Replicated Data. SOSP 1979.

Things that can make quorums tricky

- Until the leader sees that a quorum was reached, an update is pending but could “fail”
- This is why we use a 2PC protocol to do updates
- But what if leader fails before finishing phase 2?
  - If the proposer crashes, the participants might have a pending update but not know the outcome
  - In fact we need to complete such an interrupted 2PC
  - Otherwise subsequent updates can commit but we won’t be able to read the state of the system since we’ll be unsure whether the interrupted one succeeded or failed

Things that can make quorums tricky

- We might sometimes need to adjust the quorum sizes, or the value of N, while the system is running
  - This topic was explored in papers by Maurice Herlihy
  - He came up with an idea he called "Quorum Ratchet Locking" in which we use two quorum systems
    - One controls updates or reads (Q_W, Q_R)
    - A second one controls the *values* of N, Q_W, Q_R
    - While updating the second one we "lock out" the basic read and update operations. This is the "ratchet lock" concept
  - Paper on this appeared in 1986

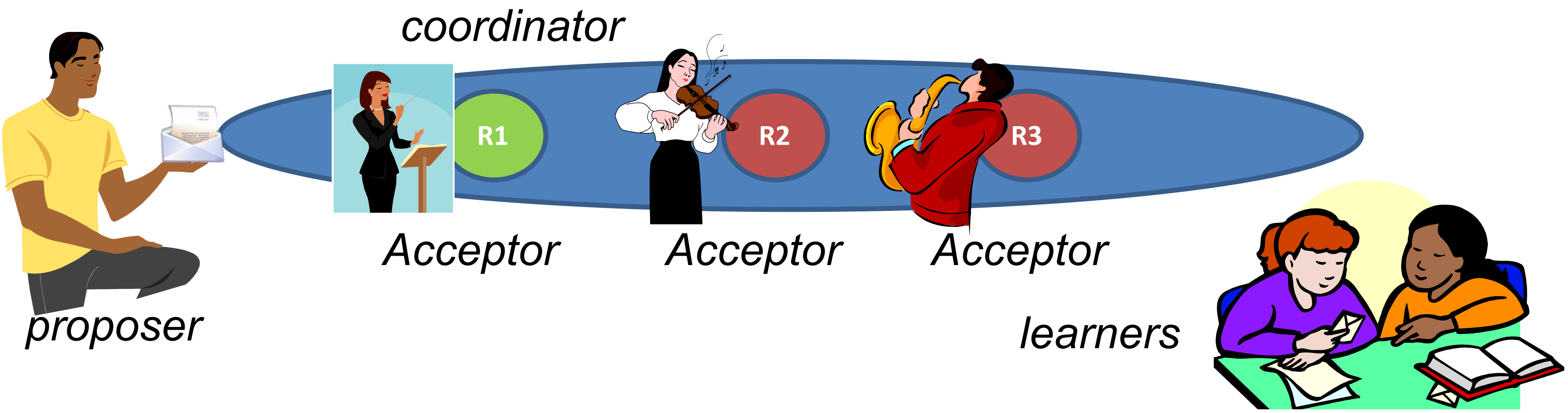
Paxos builds on this idea

- Lamport's work, which appeared in 1990, basically takes the elements of a quorum system and reassembles them in an elegant way
  - Basic components of what Herlihy was doing are there
  - Actual scheme was used in nearly identical form by Oki and Liskov in a paper on "Viewstamped Replication"
- Lamport's key innovation was the proof methodology he pioneered for Paxos

Paxos Terminology

- In Paxos we distinguish several roles
  - A single process might (often will) play more than one role at the same time
  - The roles are a way of organizing the code and logic and are not separate programs that run on separate machines
- These roles are:
  - Proposer, which represents the application “talking to” Paxos
  - Acceptor (a participant), and
  - Learner, which represents Paxos “talking to” the application
- 2 phases
  - Phase 1: Prepare request ←→ Response
  - Phase 2: Accept request ←→ Response

Visualizing this



- The proposer requests that the Paxos system accept some command. Paxos is like a "postal system"
- It thinks about the letter for a while (replicating the data and picking a delivery order)
- Once these are "decided" the learners can execute the command

Phase 1: (prepare request)

(1) A proposer chooses a new proposal version number n , and sends a prepare request ("prepare",n) to a majority of acceptors:

(a) Can I make a proposal with number n ?

(b) if yes, do you suggest some value for my proposal?

The proposal is application-specific and might be, e.g., "dispense $100 from the ATM"

Phase 1: (prepare request)

(2) If an acceptor receives a prepare request (“prepare”, n) with n greater than that of any prepare request it has already responded to, it sends out (“ack”, n, n’, v’) or (“ack”, n, $\perp$, $\perp$)

(a) responds with a promise not to accept any more proposals numbered less than n.

(b) suggest the value v’ of the highest-number proposal that it has accepted if any, else $\perp$

Phase 2: (accept request)

(3) If the proposer receives responses from a majority of the acceptors, then it can issue an accept request (“accept”, n , v) with number n and value v:

(a) n is the number that appears in the prepare request.

(b) v is the value of the highest-numbered proposal among the responses

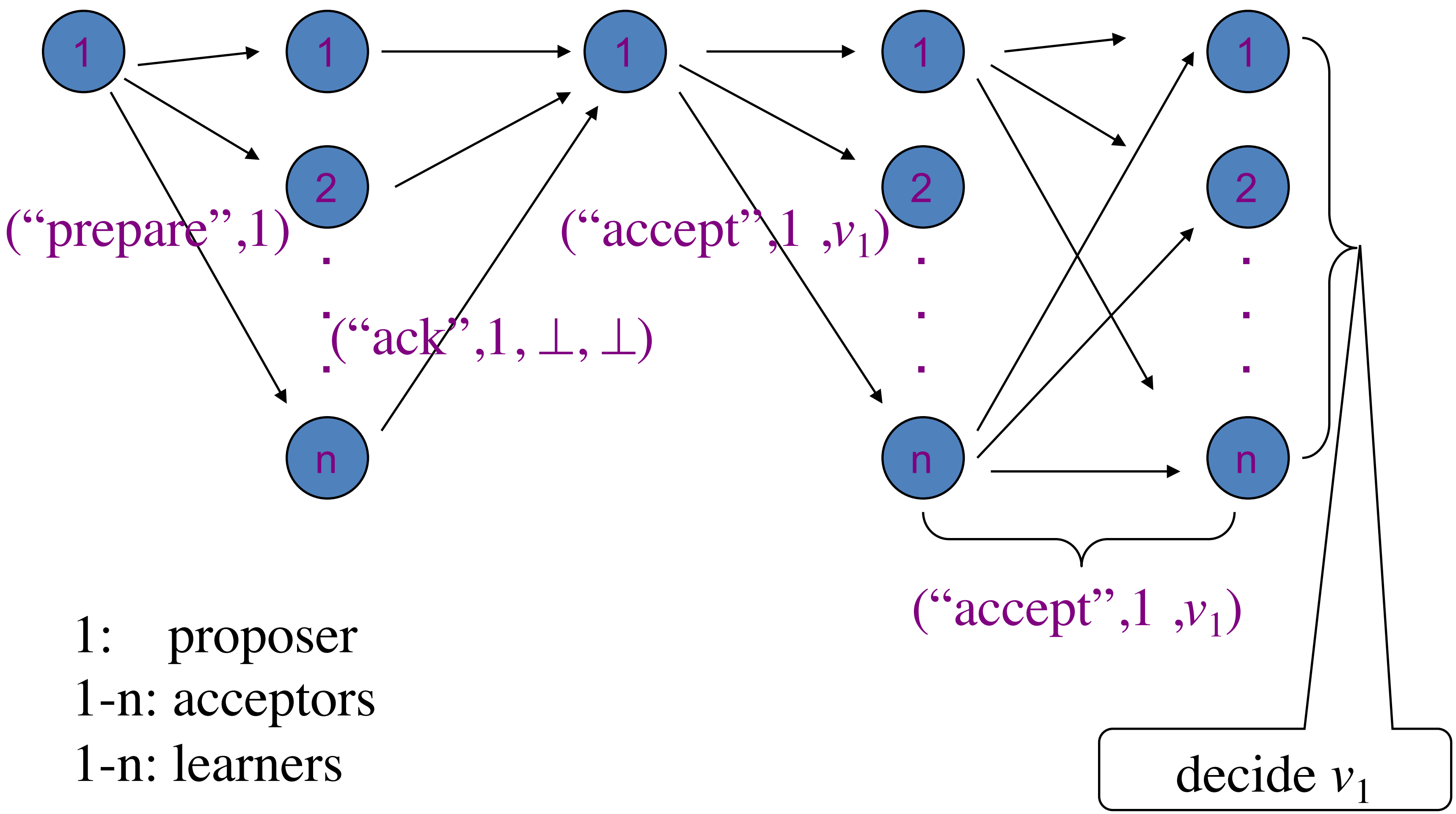
Phase 2: (accept request)

(4) If the acceptor receives an accept request (“accept”, n , v) , it accepts the proposal unless it has already responded to a prepare request having a number greater than n.

Learning the decision

- Obvious algorithm: whenever acceptor accepts a proposal, respond to all learners (“accept”, n, v).
  - No Byzantine-Failures: Acceptors informs a distinguished learner and let the distinguished learner broadcast the result.
- Learner receives (“accept”, n, v) from a majority of acceptors, decides v, and sends (“decide”, v) to all other learners.
- Learners receive (“decide”, v), decide v

In Well-Behaved Runs



1: proposer
1-n: acceptors
1-n: learners

Failures?

- Paxos “rides out” many kinds of failures
  – As long as a quorum remain available, Paxos can make progress
  – But this also reminds us that no single command list will necessarily include every decided command
  – If we look at just one command list, we would often see gaps where some coordinator didn’t reach that acceptor, but didn’t turn out to need to do so

Comments on Paxos

- The solution is very robust
  - Guarantees agreement and durability
  - Elegant, simple correctness proofs
- FLP impossibility result still applies!
  - Question: How would an adversary “attack” Paxos?
- Paxos is quite slow. Quorum updates with a 2PC structure plus quorum reads to “learn” state

Paxos summary

- An important and widely studied/used protocol (perhaps the most important consensus protocol)
  - Chubby lock service.
  - Petal: Distributed virtual disks.
  - Frangipani: A scalable distributed file system
- Developed by Lamport but the protocol per-se wasn’t really the innovation
  - Similar protocols were widely used prior to Paxos
- The key advance was the proof methodology

Leslie Lamport’s Reflections

- “Inspired by my success at popularizing the consensus problem by describing it with Byzantine generals, I decided to cast the algorithm in terms of a parliament on an ancient Greek island.
- “To carry the image further, I gave a few lectures in the persona of an Indiana-Jones-style archaeologist.
- “My attempt at inserting some humor into the subject was a dismal failure.

The History of the Paper by Lamport

- “I submitted the paper to *TOCS* in 1990. All three referees said that the paper was mildly interesting, though not very important, but that all the Paxos stuff had to be removed. I was quite annoyed at how humorless everyone working in the field seemed to be, so I did nothing with the paper.”

- “A number of years later, a couple of people at SRC needed algorithms for distributed systems they were building, and Paxos provided just what they needed. I gave them the paper to read and they had no problem with it. So, I thought that maybe the time had come to try publishing it again.”

- *Along the way, Leslie kept extending Paxos and proving the extensions correct. And this is what made Paxos important: the process of getting there while preserving correctness!*