# CS5412: PAXOS

Lecture XIII

Ken Birman

# 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 $\leq t$ failures, deploy $2t+1$ 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 (for cs5412: probably Vsync).
  - 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

- Then will look at faster but more complex reliable multicast options (many of them...)

# Key idea in Paxos: 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.
    - Updates would wait, potentially forever!
    - If a Read sees a copy that hasn't received some update, it returns the wrong value
- How can we manage replicated data in this setting?

# 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



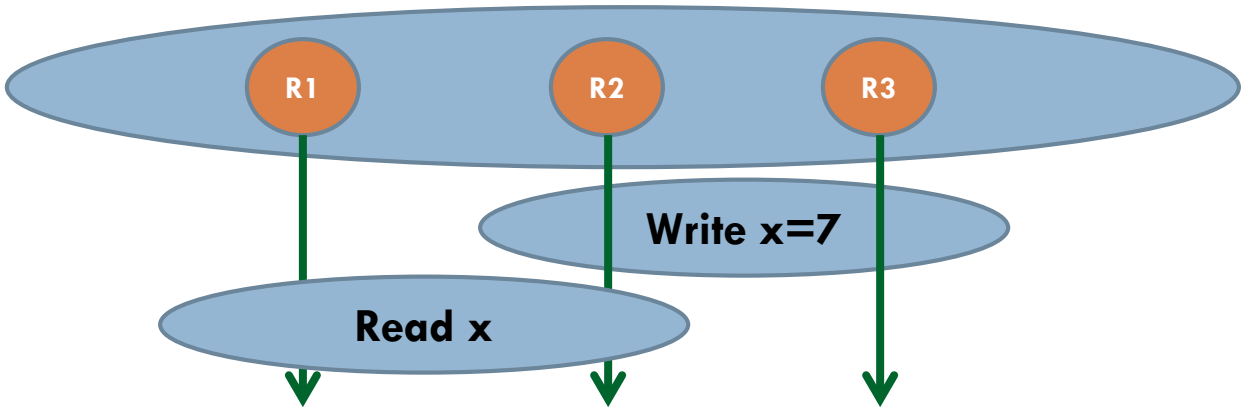
- To compensate for the risk that some replicas lack some writes, we must read multiple replicas
  - … enough copies to compensate for gaps

- 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
$Q_W = 2$
$Q_R = 2$

# 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
  - Actual scheme was used in nearly identical form by Oki and Liskov in a paper on "Viewstamped Replication" and a protocol by Dwork, Stockmeyer and Lynch is also equivalent
  - Isis "gbcast" protocol (from 1985) bisimulates Paxos, but looks very different, and proof was sloppy
- Lamport's key innovation was the step by step protocol development ("by refinement") and the rigorous proof methodology he pioneered for Paxos

# Paxos: Step by step



- Paxos is designed to deal with systems that
  - Reach ***agreement*** on what "commands" to execute, and on the order in which to execute them in
  - Ensure ***durability***: once a command becomes executable, the system will never forget the command. In effect, the data ends up in a database that Paxos is used to update.
  - Durability usually means "persisted into non-volatile storage"
- The term command is interchangable with "message" and the term "execute" means "take action"
- Paxos is *not* a reliable multicast protocol. It normally holds the entire state of the application and the front-end systems normally need to obtain (**learn**) state by querying Paxos.

# Terminology

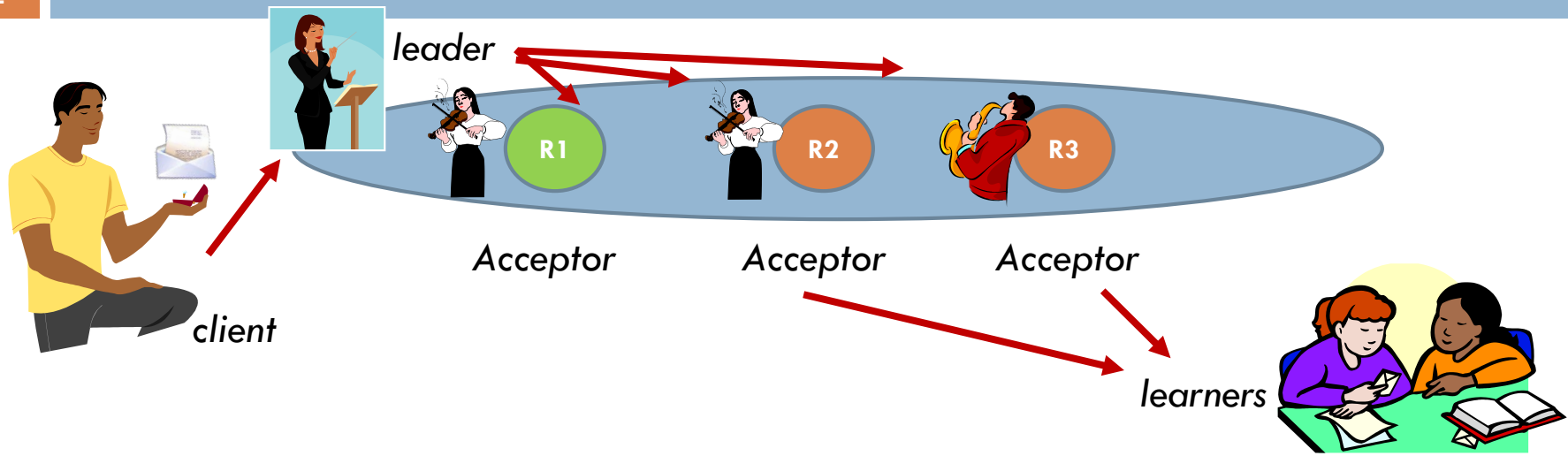


- In Paxos we distinguish several roles
  - A single process might (often will) play more than one role at the same time
  - The roles are a way of organizing the code and logic and thinking about the proof, not separate programs that run on separate machines
  - Client. Not "modelled" but is the application using Paxos.

- These roles are:
  - Leader (a process that runs the update protocol),
  - Acceptor (a participant), and
  - Learner (a protocol for obtaining the list of committed commands)

# Visualizing this






- The client asks the leader for help: "I would like the Paxos system to accept this command." Paxos is like a "postal system". A leader passes the command in.
- The leader and acceptors group "thinks" about the letter for a while (replicating the data and picking a delivery order)
- Once the commands and their ordering is "decided" it can be learned by running a learner protocol
- Usually the learners then take some action: they carry out the command

# Why even mention clients/learners?



- We need to "model" the application that uses Paxos
- It turns out that correct use of Paxos requires very specific behavior from that application
- You need to get this right or Paxos doesn't achieve your application objectives
  - In effect, Paxos and the application are "combined"
  - Many research papers get this wrong!

# Client request initiation role



- When an application wants the state machine to perform some action, it prepares a "command" and gives it to a process that can play the leader role.
  - A leader is whatever program runs the Paxos protocol
  - Ideally there is just one leader, but nothing bad happens if there happen to be two or more for a while
  - Leader is like the coordinator in a 2PC protocol
- The command is application-specific and might be, e.g., "dispense $100 from the ATM in Statler Hall"

# Client learner role



- Something has to run the learner protocol to extract the state from Paxos.
- Having learned the state, a client can take action
- For example, the client could be the ATM in Statler and it might actually "dispense" $100
  - Paxos itself doesn't look at the command per-se

# Leader role



- It runs the Paxos protocol, which has two phases
  - Phase 1 "prepares" the acceptors to commit some action. Several tries may be required
  - Phase 2 "decides" what command will be performed. Sometimes the decision is that no command will be executed.

- We run this protocol again and again. Each time we run it for one in a series of "slots" that jointly constitute a list of commands the system has decided

- Once decided, the commands are performed in the order corresponding to the slot numbers by "learners"

# So… Paxos is like an append log



- In effect each command is added to the end of a write-only log that gets longer and longer
- In fact there is a way to garbage collect from the front of the log, but surprisingly rarely used
- Thinking of Paxos as a way to make a durable log of messages is the right way to view the protocol

# Acceptor role: Maintain this log



- The Paxos acceptors maintain lists of commands, and these are stored on disk, not in memory
- Each has a subset of the total command list but may have gaps, so they aren't identical replicas
  - Think of the list as a vector indexed by "slot number"
  - Slots are integers numbered 0, 1, ....
  - While running the protocol, a given acceptor might have a command in a slot, and that command may be in an "accepted" state or in a "decided" state
- Acceptors each have distinct logs: each holds its own subset of the total set of data

# Ballot numbers



- Goal is to reach agreement that a specific command will be performed in a particular slot
- But it can take multiple rounds of trying (in fact, theoretically, it can take an unlimited number, although in practice this won't be an issue)
- These rounds are numbered using "ballot numbers"

# Basic idea of the protocol



- Leader proposes a specific command in a specific slot in a particular ballot
  - If two leaders compete the one with the higher ballot will always dominate.
  - If two leaders compete with the same slot # and ballot #, at most one (perhaps neither) will succeed
  - Also, when leaders notice that they are competing, one of them yields to the other we soon end up with just one leader. Paxos is fastest and most efficient with just one "elected" leader
- We never talk about a command without slot and ballot #s
  - Paxos is about agreeing to execute the "Withdraw $100" first, and then the "Deposit $250" second
  - Slot # is the order in which to perform the commands

# Commands go through "states"



- Initially a command is known only to client & leader
- Then it gets sent to "acceptors" [illegible] asked to "prepare" to execute the co[illegible]
- If a quorum is reached, then the accep[illegible]rs are told that the command has been "accepted".
- A command is "decided" by running a second pha[illegible]
- A decided command can be executed (unless you overdraw your account)


# Applying commands to the state machine



- The learner watches and waits until new commands become committed (decided)
  - As slots become decided, the learner is able to find out if a decided slot has a command, or nothing in it.
    - Goes to the next slot if "no command"
    - Performs the command if a command is present
  - Can't skip a slot: learner takes one step at a time

- Little known but important: *after a crash, a recovering learner is shown the whole log from the start and is supposed to ignore commands that were already done.*

# Core protocol



- Phase 1: Leader sends *prepare* (slot,b,c) to acceptors
  - It believes this slot is free, and it uses the next ballot number
  - An acceptor looks at the slot and ballot number
    - If it hasn't previously voted in this slot, for this ballot number, it votes to accept the ballot and remembers the command
    - Otherwise it votes against the ballot and sends back the command it previously accepted

# Core protocol



- Leader wants to achieve a write quorum
  - If it succeeds, it starts phase 2 by asking acceptors to commit (slot,b,c) for the ballot number on which it got a quorum
  - Acceptor agrees if this is the highest ballot number for which it has been asked to participate in phase 2, otherwise rejects the request
  - If it again achieves a quorum of acknowledgments, the request has been decided and the coordinator sends out a "decide" ("commit") message
  - Otherwise it retries phase 1 with the next ballot number

# Incrementing the ballot number



- A leader can retry a command in some slot with the next largest ballot number, but only if there was a tie and no command was able to get a $Q_w$ majority

- Once some command is accepted by $Q_w$ acceptors no other command can be accepted into that slot

# Failed command



- If two or more leaders both run phase 2, at most one command can be accepted into any given slot
- The leaders that fail will need to retry with some other slot number
- There is also a case in which no leader is able to succeed and all move to the next slot number

# Things to notice



- If a command is decided in some slot, for some ballot number, no other command can be accepted into that same slot (for any ballot number)
  - To prove this, observe that for this to be violated, some acceptor would need to accept a phase 1 message after accepting a phase 2 message
  - This is because $Q_W+Q_W > N$

# More things to notice



- A leader may not actually realize that its command was accepted by a majority!
  - Messages are unreliable so the accepted messages can be lost, just like "yes" votes in 2PC
  - This would cause the leader to retry the same command with some other ballot number
  - Nothing bad will happen

# Things to notice about phase 2



- Two leaders could both try to enter phase 2 with different commands
  - One with ballot number b
  - Another with some ballot number b' > b
- In phase 2, only the latter could succeed and commit because there won't be a "surviving" quorum that have voted for command c with ballot b
  - Even though *some* acceptors might phase for the earlier command in phase 2, that leader definitely can't get a quorum and will fail
  - The case that leads to a "nothing" decision combines this scenario with an actual failure, so that both leaders enter phase 2, and neither can decide

# Learning (aka "Deciding")



- To learn the state of Paxos the learner combines data from $Q_r$ logs
- By merging the logs, it can be sure that it learns of every committed write, and in the proper order
- The leader can't just trust any single log because since $Q_w < N$, a single log might lack some updates

# Failures?



- Paxos "rides out" many kinds of failures
  - As long as a quorum remains available, Paxos can make progress in the sense of adding entries to the command list
  - But keep in mind 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 leader didn't reach that acceptor, but obtained a write quorum anyhow

# Failed leader?



- If a leader crashes, the next time a leader tries to run, it will notice any pending but undecided commands in the history
- It completes those interrupted protocol instances on behalf of the failed leader
- This way Paxos makes progress under conditions where a traditional 2PC might get stuck

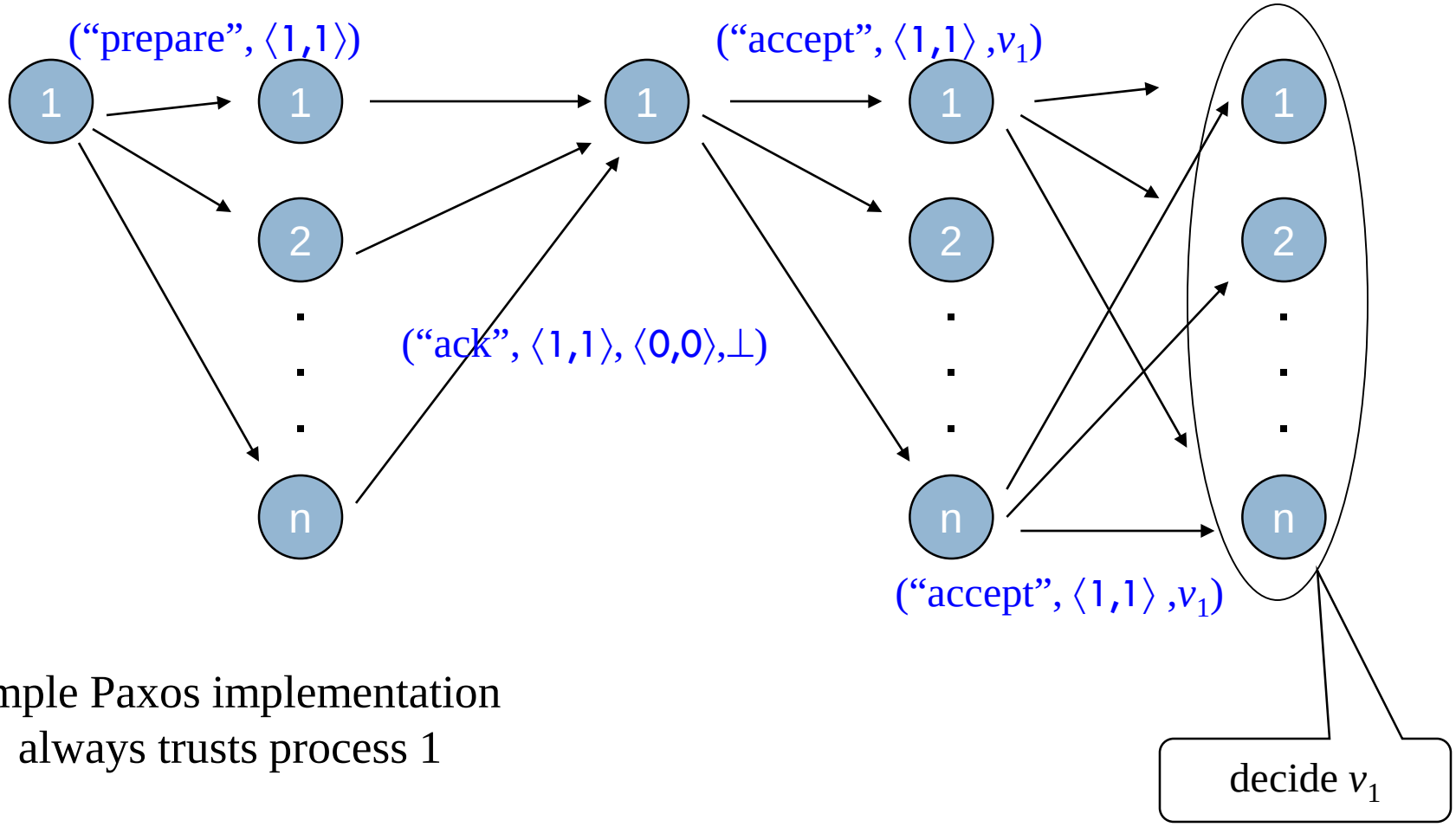
# FLP



- It turns out that the Paxos acceptor protocol can advance without blocking
- But if you combine the acceptor protocol with the learner protocol, FLP applies.
- In effect, Paxos can definitely accept data but there is an attack that could block you from learning the committed commands.

# In Failure-Free Synchronous Runs






Simple Paxos implementation always trusts process 1

# Looks like a 3-phase commit!



- The pattern is indeed like a 3-phase commit.
- Many people understand Paxos in these terms.
- But we don't normally have 3-phase commit with multiple leaders competing to own a slot, so in this sense the protocol isn't really identical, just similar.

# Reconfigurable Paxos



- Lamport extended Paxos to support changing membership

- Basically, this entails
  - Suspending the current configuration ("wedge" it)
  - Reaching agreement on the initial state (initial command list and new quorum configuration policy ($N$, $Q_W$, $Q_R$) that will be used in the new state machine)
    - A version of the learner role
    - In effect, the members of the new configuration learn the outcome of the prior configuration
    - Then can start the new configuration
  - The old wedged configuration has been "terminated"

# Odd behavior



- When reconfiguring to add a server, we normally would copy the log from some server to the new one.
  - This is safe if we copy the log of a server being taken out of the group. But what if it caught fire and the state was lost?
  - A natural thing is to copy the log of some other server
- But this is unsafe!
  - An "empty" slot could now suddenly show a command that seemingly achieved $Q_w$ accepts and committed
  - Thus long after command **c** failed to commit in slot **s**, there is a case where **c** would "reappear".

# More odd behavior



- Reconfigurable Paxos can also continue to commit messages in the old wedged configuration.
  - The new configuration doesn't know when it is safe to start
  - Sometimes a very slow and visible hiccup

- Recently, a mixture of virtual synchrony and Paxos showed how to eliminate this issue. See Chapter 22 in the textbook. Leslie calls this **virtually synchronous Paxos.** We'll see more on this topic soon.

# Paxos optimizations



- Using a leader-election scheme we can reduce the risk of having two leaders that intefere with each other (if that happens, they can repeatedly abort)
- We can also batch requests and do several a time
- We can have multiple leaders and run them all at the same time, but assign them to distinct slots

- The trick is that we build this as incremental steps so the "correctness" of the core protocol is unchanged

# Comments on Paxos



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

# How slow?



- At best we need
  - Leader to acceptors: 1 to N multicast
  - Acceptors back to leader: N to 1 unicasts, plus scheduling delays for leader to read these in
  - Leader discovers it has a quorum of replies
  - Leader to acceptors: 1 to N multicast again
  - Acceptors to leader: N to 1 unicasts ("prepared"), more scheduling delays
  - Leader to acceptors: 1 to N multicast ("commit")
- We will see that Vsync has a 10x faster reliable protocol but it isn't identical in its guarantees.

# Paxos with a disk



- Very often we want a system to survive complete crashes where all members go down, then recover
- An "in-memory" Paxos won't have this property
- Accordingly, the command list must be kept on a disk, as a disk log: leads to **Corfu** system
  - Now accept and commit actions involved disk writes that must complete before next step can occur
  - Further slows the protocol down
  - In Vsync implemented by SafeSend DiskLogger durability plugin (enabled via g.SetDurabilityMethod)

# Paxos in Vsync



- Access via the g.SafeSend API
  - You chose between in-memory and disk Paxos
  - Must also tell the system how many acceptors to use

- Is SafeSend really Paxos?
  - Yes… but… it includes an optimization that simplifies the protocol and speeds up learners
  - Discussed in Chapter 22 of textbook
  - The properties are exactly those of standard Paxos

# Paxos isn't a reliable multicast!



- Consider the following common idea:
  - Take a file, or a database… Make N replicas
  - Now put a program that runs Paxos in front of the replicated file/db
  - Learner just asks the file to do the command (a write or append), or the DB to run an update query
  - Would this be correct? Why?
    - … no, because after a crash, Paxos will replay commands and the file or database might be updated twice if it isn't "idempotent". Many researchers don't realize this.

# Correct use of Paxos



- The learner needs to be a part of the application!
- By treating the learner as part of Paxos, we erroneously ignore the durability of actions in the application state, and this causes potential error
  - The *application* must perform every operation, at least once
  - Learner retries after crashes until application has definitely performed each action.
  - To avoid duplicated actions, application should check for and ignore actions already applied to the database
- Many Paxos-based replication systems are incorrect because they fail to implement this logic!

# How this works in Vsync



- The DiskLogger durability method has a "dialog" with the application
  - DiskLogger+application are like a learner
    - When DiskLogger delivers a message the application must "confirm" accepting that operation
    - E.g. might apply it to a database and wait until done
    - If a crash happens, DiskLogger will redeliver any unconfirmed messages until it gets confirmation
- With in-memory durability, SafeSend skips this step
  - But this is weaker than the way Paxos is "normally" used

# Details: Paxos oddities



- To increase performance, Paxos introduces a "window of concurrency" $\alpha$: as many as $\alpha$ commands might be concurrently decided
  - E.g. instead of proposing the next slot, we can allow proposals for slots s, s+1, … s+$\alpha$-1
  - But this adds an issue: when new configuration is defined, as many as $\alpha$-1 commands may still be decided "late", in the new configuration
  - This can be a problem for application with configuration-specific commands; they need to add "guards" like "As long as the configuration is still {P,Q,R} deduct $100 from the account and dispense the cash"
  - This is annoying and error-prone, so many run with $\alpha$=1but then run slowly because they can't leverage concurrency

# Details: Paxos oddities



- A really strange thing can happen if we add members in new configurations
  - Paxos requires that we "learn" the configuration
  - But some Paxos implementations short-cut this by copying some command list from an old member to a new one: "state transfer"
  - That's a mistake: some command that was marked as accepted but never committed (never decided) because it lacked a write quorum could *later* pass the write-quorum threshold retroactively!

# Details: Paxos oddities



- Example: command x reaches just P in {P,Q,R} in slot 17 on ballot 1.
  - x doesn't achieve a quorum and eventually slot 17 decides "nothing"
  - Some time later Q and R are replaced by S and T in a new configuration and S and T *initialize themselves from* rather than "learning" from {P,Q,R}
  - Now x is in P,Q,R's command list and hence has a quorum
  - So it sort of gets decided "very late" and at a time long in the past!
  - Causes serious bugs in applications that use Paxos reconfiguration if this style of reconfiguration plus state transfer is used. The version with a learner, though, can be slow and hard to implement!

# Paxos summary



- An important and widely studied/used protocol (perhaps the most important agreement protocol)
- Developed by Lamport but the protocol per-se wasn't really the innovation
  - Similar protocols were widely used prior to Paxos
  - Earliest was Isis "gbcast" but although it bisimulates Paxos, it looks very different. Then Oki/Liskov, Dwork/Stockmeyer/Lynch. Paxos came later.
- The key advance was the proof methodology
  - We touched on one corner of it
  - Lamport addresses the full set of features in his

# What did Lamport do with Paxos?



- His main focus was on using "model checking" to confirm that the protocol really can handle every possible mix of failures that could occur

- Surprisingly complex challenge. He uses "TLA+" for this. The work is very interesting.

- If a bug could occur, TLA+ prints out a sequence of events that will cause Paxos properties to break

# What about other uses of Paxos?



- Many people created members of what could almost be called a family of Paxos protocols
- Today there are dozens of such protocols (including the ones in Vsync). Some date to prior to Paxos, but we still tend to call this family of solutions Paxos
- They are all strong enough to solve consensus

# 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!*