# 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 ≤ *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 Isis$^2$).
  - 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.
  - 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



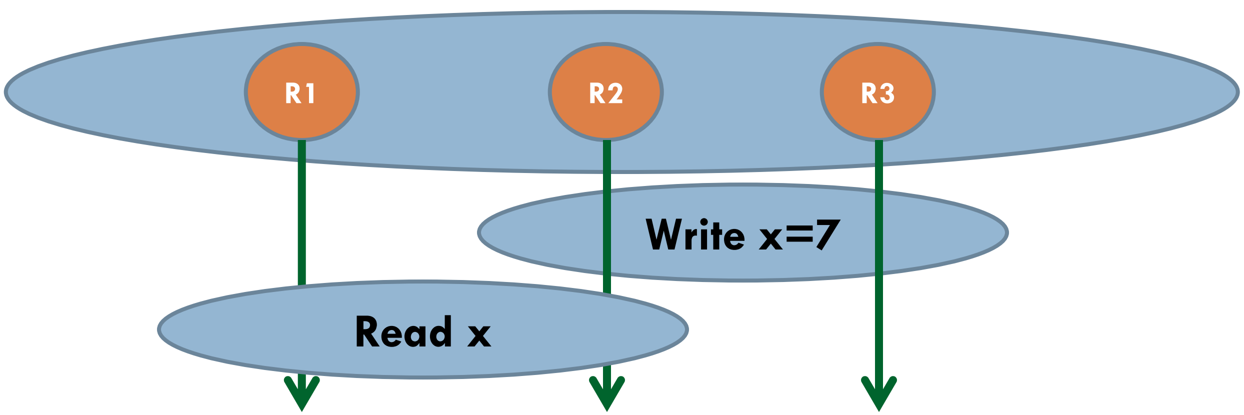
- To compensate for the risk that some replicas lack some writes, we must read multiple replicas
  - … enough copies to compensate for gaps

- 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




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

# 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: 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

- The term command is interchangable with "message" and the term "execute" means "take action"
- But we will see later that Paxos is *not* a reliable multicast protocol. It normally needs to be part of a replicated system, not a separate library

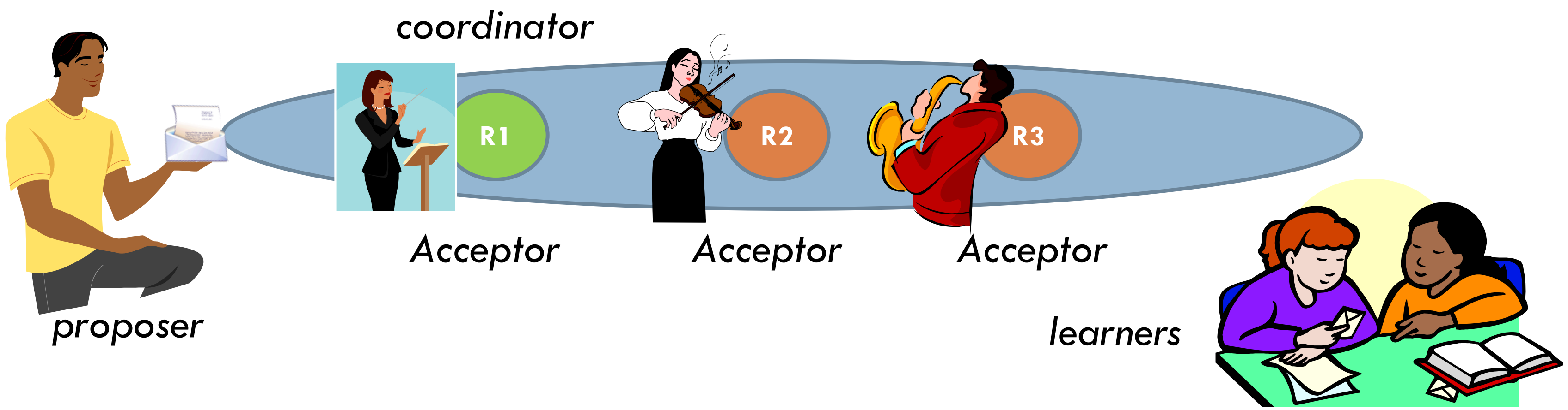
# Terminology



- 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
- These roles are:
  - Proposer, which represents the application “talking to” Paxos
  - Coordinator (a leader that runs the protocol),
  - Acceptor (a participant), and
  - Learner, which represents Paxos “talking to” the application

# 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

# Why even mention proposers/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"
  - In other words, *Paxos is not a multicast library.*

# Proposer 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 proposer role.
  - The coordinator will run the Paxos protocol
  - Ideally there is just one coordinator, but nothing bad happens if there happen to be two or more for a while
  - Coordinator is like the leader in a 2PC protocol
- The command is application-specific and might be, e.g., "dispense $100 from the ATM in Statler Hall"

# Coordinator 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 for a series of "slots" that 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"

# Acceptor role: Maintain "command list"



- The Paxos replicas maintain a long list of commands
  - Think of it as a vector indexed by "slot number"
  - Slots are integers numbered 0, 1, ....
  - While running the protocol, a given replica might have a command in a slot, and that command may be in an "accepted" state or in a "decided" state
- Replicas each have distinct copies of this 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



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

# Commands go through “states”



- Initially a command is known only to proposer & coordinator
- Then it gets sent to “acceptors” ... to “prepare” to execute the com...

Request denied: Exceeds current balance ($31.17)

- If a quorum is reached, then the acceptors are told that the command has been “accepted”.
- A command is “decided” by running a second phase
- A decided command can be executed (unless you overdraw your account)

# Learner role



- 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

# Core protocol



- Phase 1: Coordinator sends *prepare* (slot,b,c) to acceptors
  - It thinks this is a free slot and 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



- Coordinator 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

# Failed command



- If two coordinators both run phase 2, at most one command can be decided
- The coordinator that fails will need to retry with some other slot number

- There is also a case in which neither is able to succeed and both 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 coordinator 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 coordinator to retry the same command with some other ballot number
  - Nothing bad will happen

# Things to notice about phase 2



- Two coordinators 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 coordinator 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 coordinators enter phase 2, and neither can decide

# Learning (aka "Deciding")



- The learner might see a "decide" message, but if not can still advance by doing quorum reads
  - Its local replica of the command list, if it is also an acceptor,. might have gaps, or lack outcomes for some commands
  - By doing a quorum read, a learner can be certain to discover any committed command. If it also notices an unterminated entry in the history, it can fix it
- A learner <u>executes</u> an accepted (decided) command if
  - It knows the decision for every slot up to and including the slot in which that command was decided, and
  - It has executed every prior accepted command

# 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

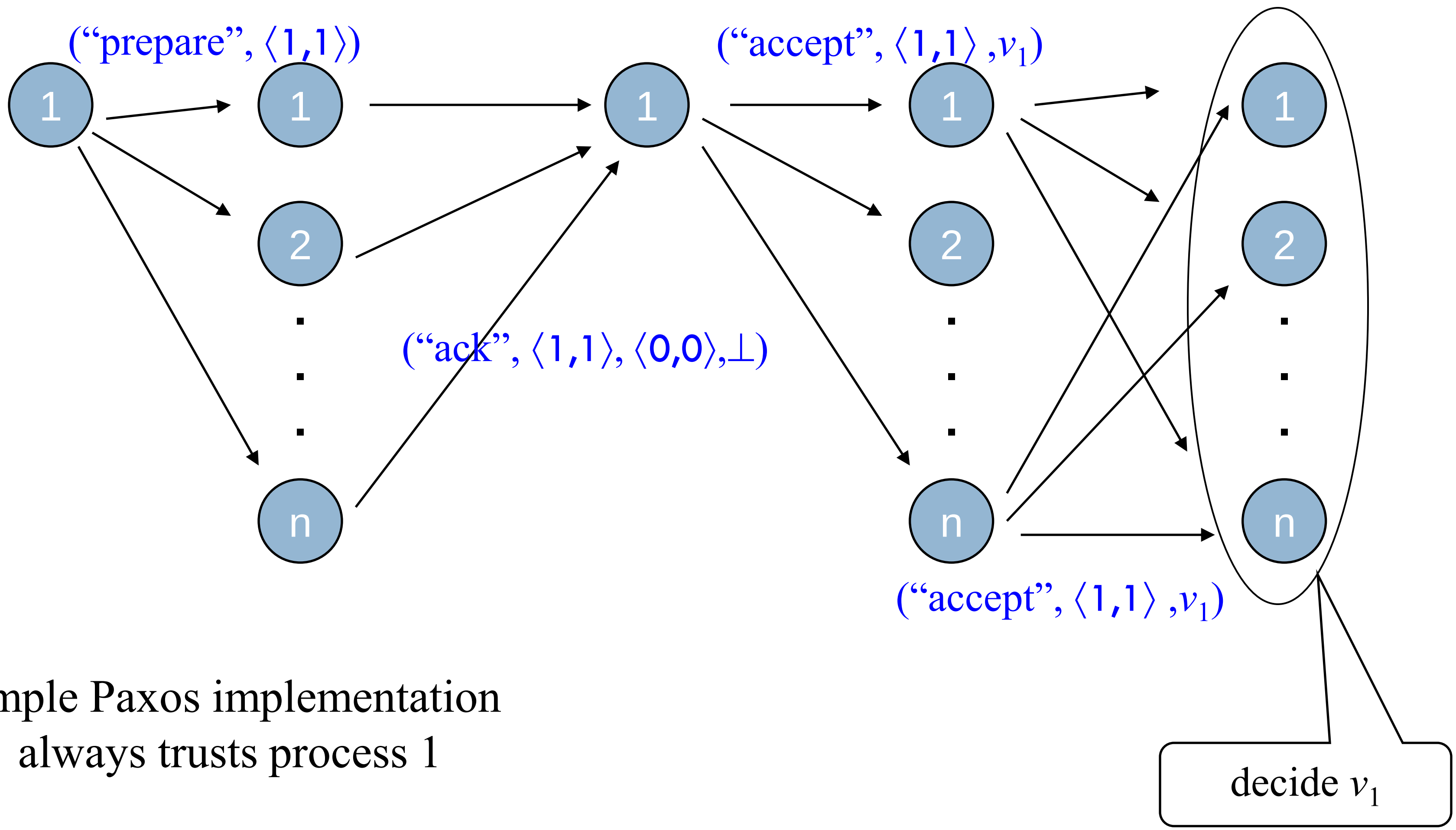
# Failed coordinator?



- If a coordinator crashes, the next time a coordinator 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 coordinator
- This way Paxos makes progress

# In Failure-Free Synchronous Runs






Simple Paxos implementation
always trusts process 1

# 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"

# Paxos optimizations



- Using a leader-election scheme we can reduce the risk of having two proposers that intefere with each other (if that happens, they can repeatedly abort)
- We can batch requests and do several a time
- We can combine several proposals and run them all at the same time, for 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

# 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 often be kept on a disk, as a disk log
  - Now accept and commit actions involved disk writes that must complete before next step can occur
  - Further slows the protocol down
  - In Isis$^2$ implemented by SafeSend DiskLogger durability plugin (enabled via g.SetDurabilityMethod)

# Paxos in Isis$^2$



- 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 Appendix A 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?

# 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 Isis$^2$



- 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

# Other 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

# Other 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!

# Other 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 *<u>initialize themselves from</u>* 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
- The key advance was the proof methodology
  - We touched on one corner of it
  - Lamport addresses the full set of features in his

# 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!*