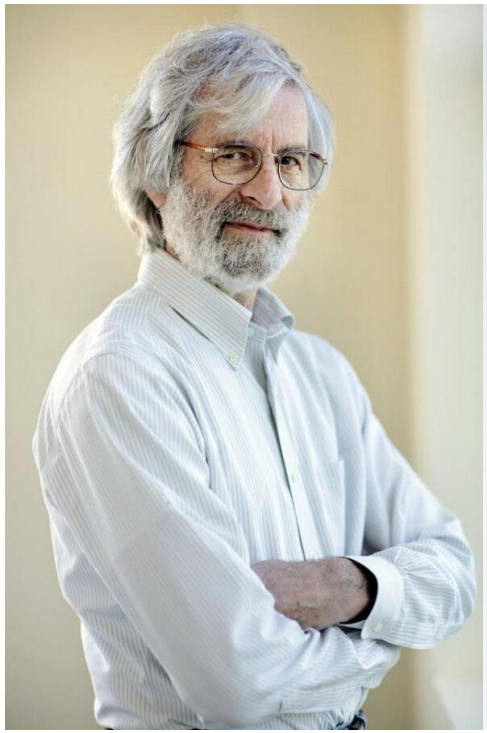
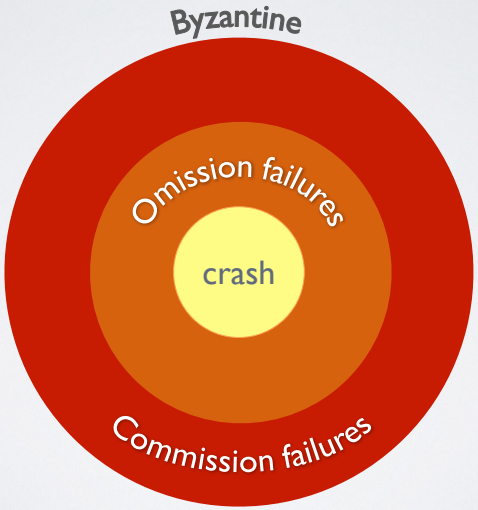


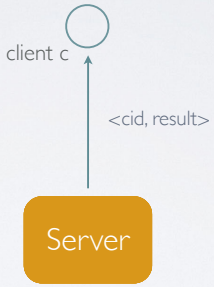
"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Leslie Lamport

FAILURE MODELS



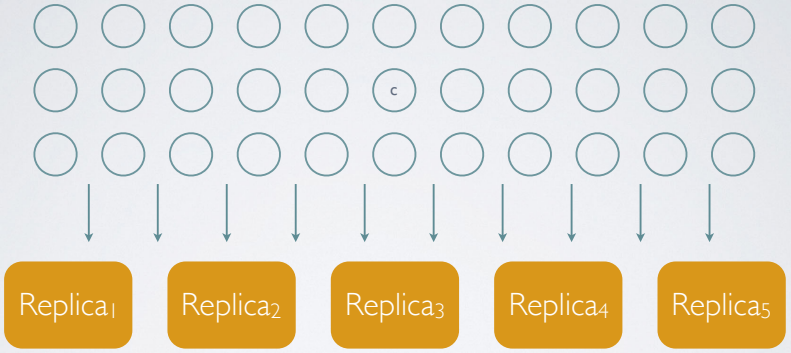
THE BIG PICTURE



THE BIG PICTURE

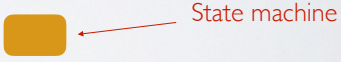
THE BIG PICTURE

THE BIG PICTURE



STATE MACHINE REPLICATION

1. Make server deterministic (state machine)

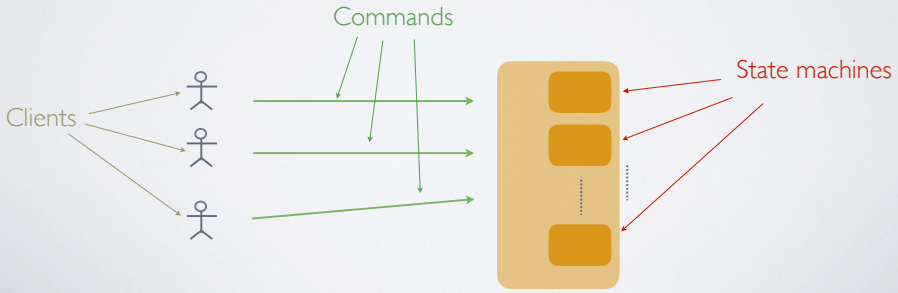


STATE MACHINE REPLICATION

1. Make server deterministic (state machine)
2. Replicate server

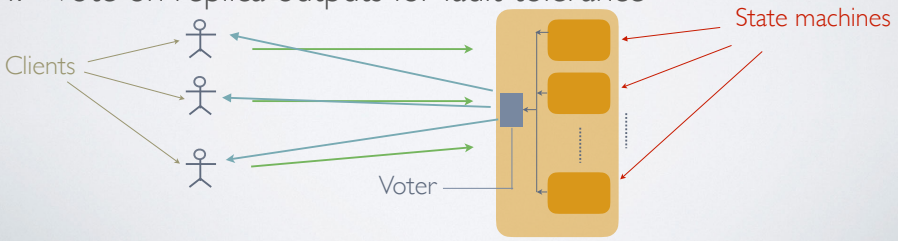
STATE MACHINE REPLICATION

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions

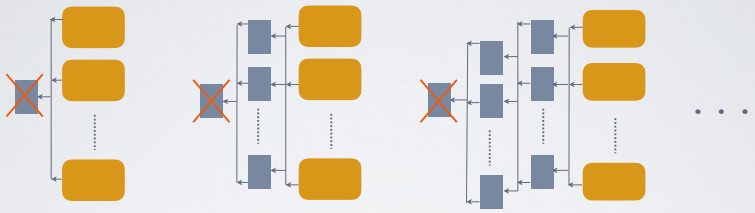


STATE MACHINE REPLICATION

1. Make server deterministic (state machine)
2. Replicate server
3. Ensure correct replicas step through the same sequence of state transitions
4. Vote on replica outputs for fault-tolerance

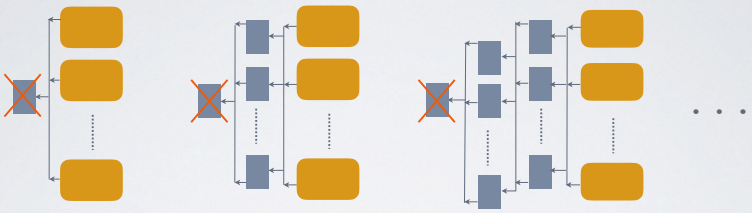


A CONUNDRUM

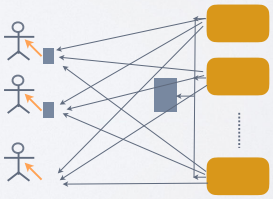


A: voter and
client share fate!

A CONUNDRUM



A: voter and
client share fate!



REPLICA COORDINATION

All non-faulty state machines receive all commands in the same order

- Agreement: Every non-faulty state machine receives every command
- Order: Every non-faulty state machine processes the commands it receives in the same order

HOW?

The Dear Leader

The Parliament

PRIMARY-BACKUP

- Clients communicate with the Dear Leader (the Primary)
- The Primary:
  - sequences clients' requests
  - updates as needed other replicas (backups) with sequence of client requests or state updates
  - waits for acks from all non-faulty clients
- Timeouts detect failure of primary
- On primary failure, a backup is elected as new primary

PRIMARY-BACKUP vs PARLIAMENT SYSTEM

1. More fault tolerance: $f < N$ vs $f < N/2$
2. Easier to develop, debug, tune, and maintain
3. Less communication and computation
4. Can handle non determinism (!)
5. Needs a stronger failure model (fail stop)

SOME LIKE IT HOT

- Hot Backups process information from the primary as soon as they receive it
- Cold Backups log information received from primary, and process it only if primary fails
- Rollback Recovery implements cold backups cheaply:
  - Primary logs information needed by backups directly to stable storage
  - Backups are generated "on demand" upon promary's crash

Deterministic Replay for Multiprocessors

Why?

Record and reproduce multithreaded executions

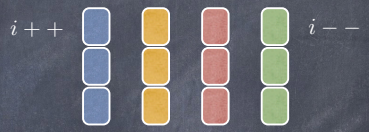
- Debugging
- Program analysis
- Forensics and Intrusion Detection
- Fault tolerance

What is hard?

On a uniprocessor

- Only one thread at a time updates shared memory
  - concurrency is simulated
- Record scheduling decisions
  - fewer than SM accesses

On a multiprocessor



- Threads actually update shared memory concurrently

Instrument each memory operation
10-100X

Detect dependencies using memory protection bits
Up to 9x

?

Reduced logging + offline search
Slow replay

Hardware support
Custom HW

What to replay?

- Exact reproducibility is hard and expensive...
- ... but no need to replay the exact execution
  - Aim for observationally indistinguishable
    - produce same set of states S
    - produce same set of outputs O
    - match a possible execution of the program that would have produced S and O

When to replay?

- Online: in parallel with the original execution
  - fault tolerance, parallel security check
- Offline: after the original execution has completed
  - debugging, forensics, etc

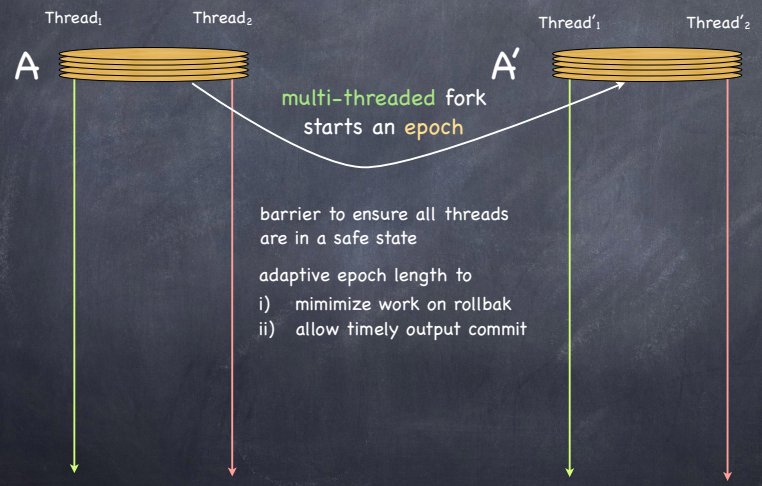
Online Multiprocessor Replay

Respec, ASPLOS '10

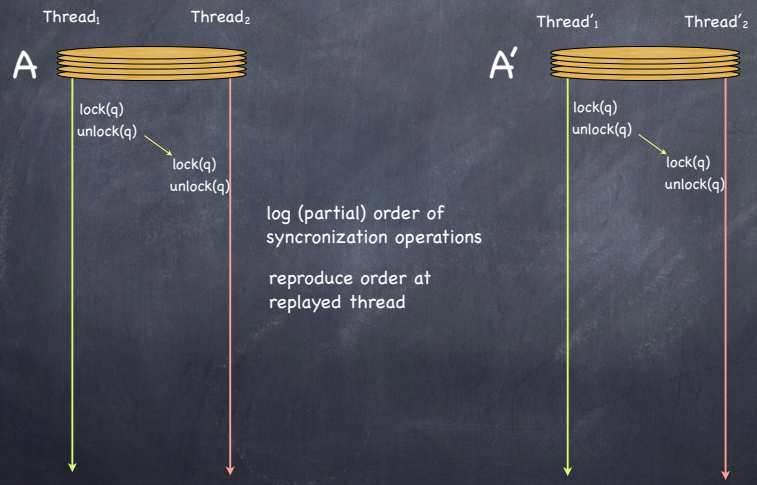
Key idea: "~~trust~~ but ~~verify~~" (speculate, check)

1. Speculate execution is data race free
2. Check efficiently for mis-speculation
3. On mis-speculation, rollback and retry

Speculate



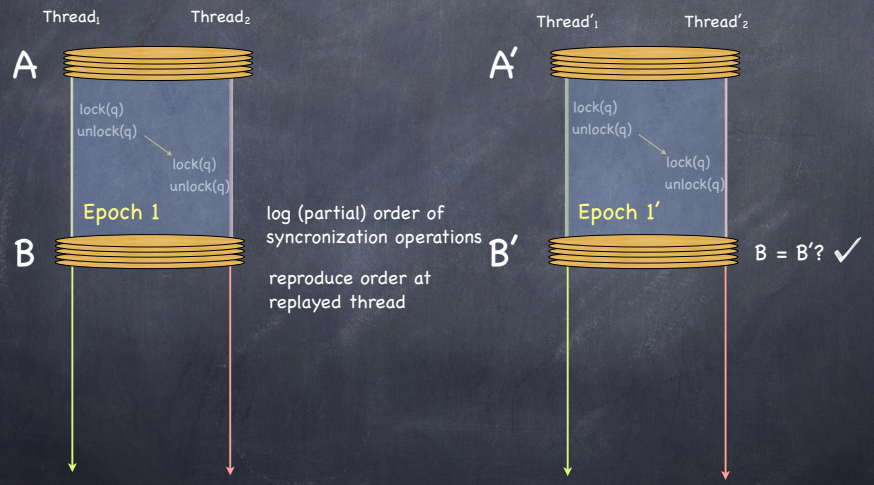
Speculate



log (partial) order of syncronization operations

reproduce order at replayed thread

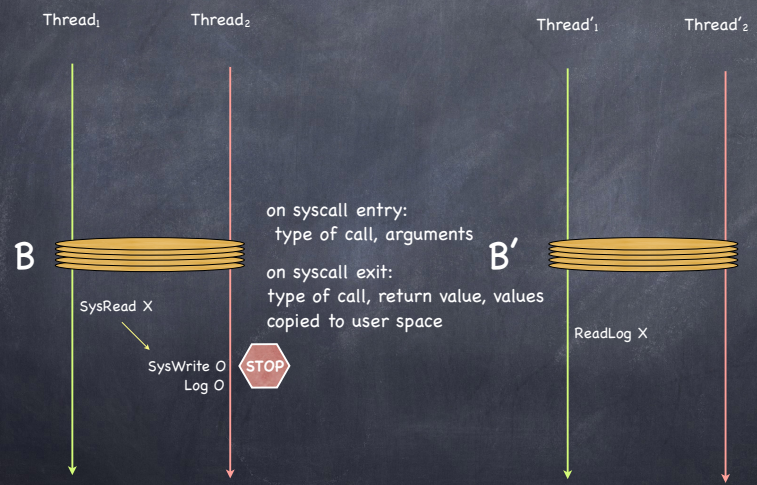
Speculate



log (partial) order of syncronization operations

reproduce order at replayed thread

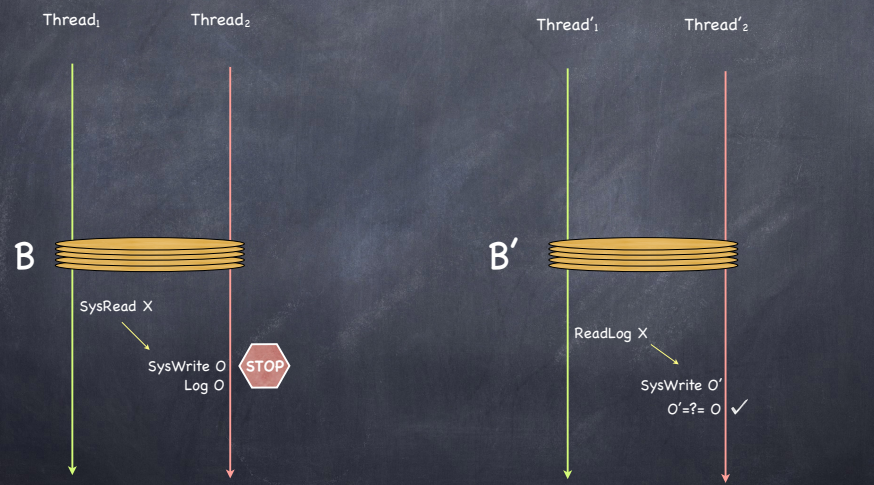
Speculate



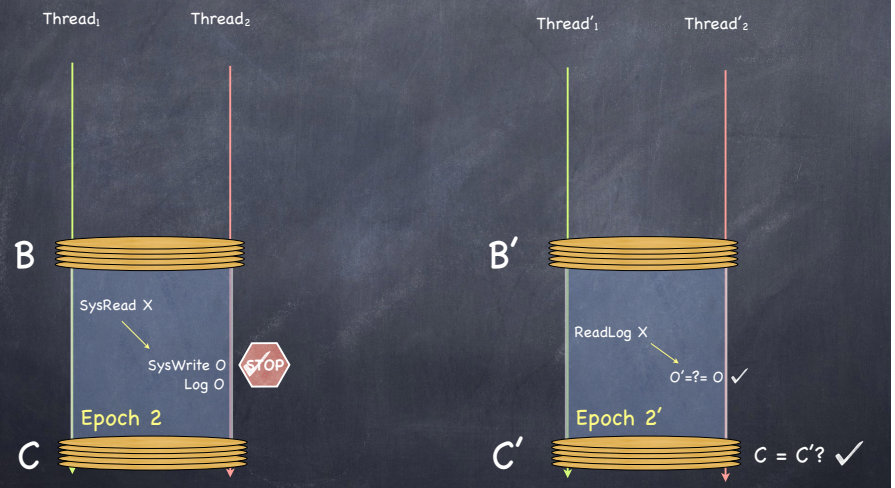
on syscall entry:
type of call, arguments

on syscall exit:
type of call, return value, values copied to user space

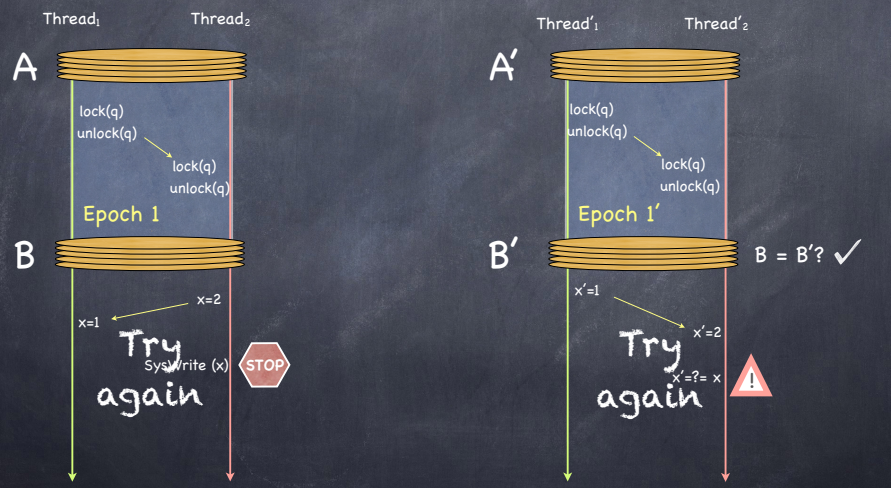
Check



Check



Mis-speculation



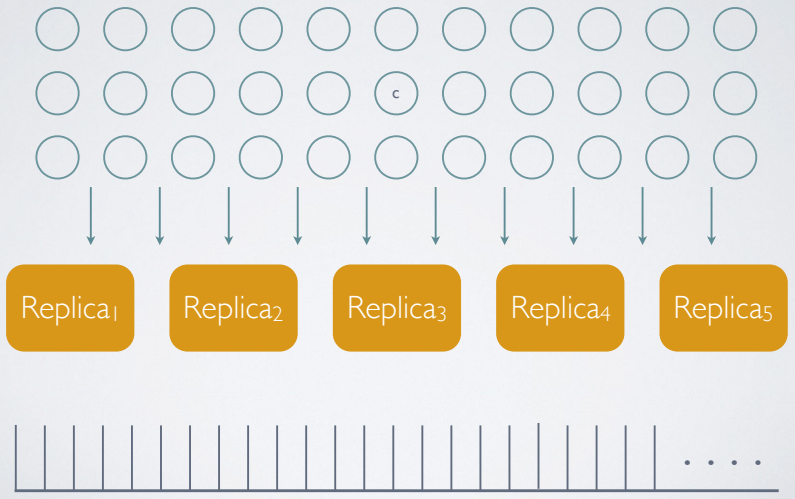
Liveness

- Could record individual accesses...

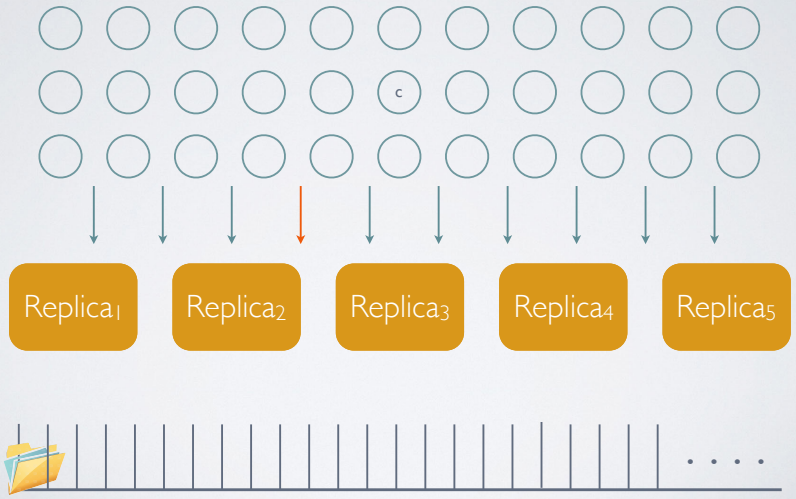
Instead

- Switch to uniprocessor execution
  - Record and replay one thread at a time, recording preemption point
  - Parallel execution resumes after failed epoch completes

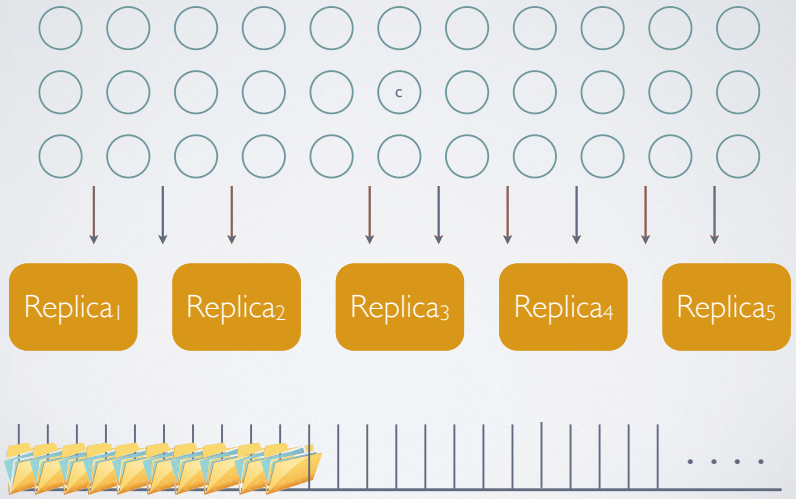
THE BIG PICTURE



THE BIG PICTURE



THE BIG PICTURE



The Dear Leader

The Parliament

CONSENSUS

- **Validity** – If a process decides v, then v was proposed by some process
- **Agreement** – No two correct process decide differently
- **Integrity** – No correct process decides twice
- **Termination** – Every correct process eventually decides some value

A simple Consensus algorithm

Process p_i:

Initially $V = \{v_i\}$

To execute propose(v_i)

1: send $\{v_i\}$ to all

decide(x) occurs as follows:

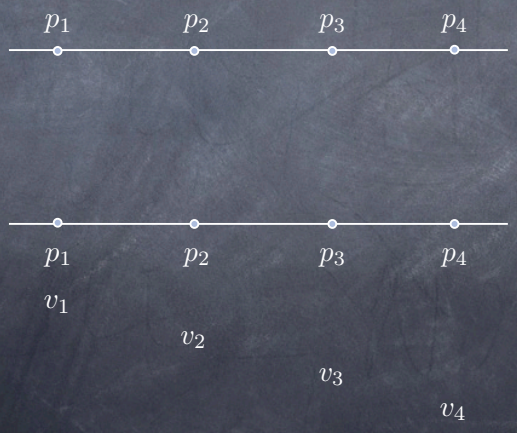
2: for all j, $0 \leq j \leq n-1$, $j \neq i$ do

3: receive S_j from p_j

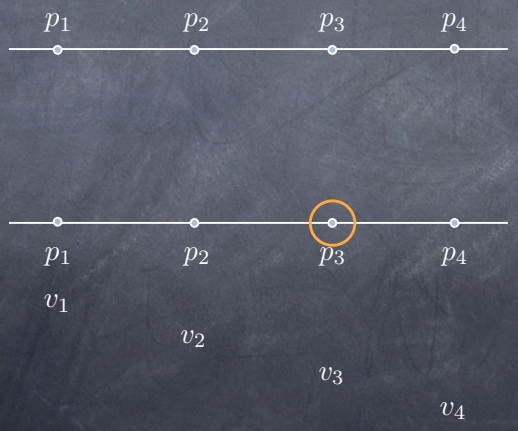
4: $V := V \cup S_j$

5: decide min(V)

An execution



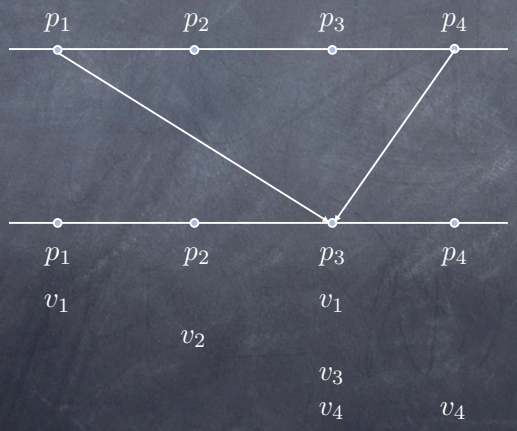
An execution



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

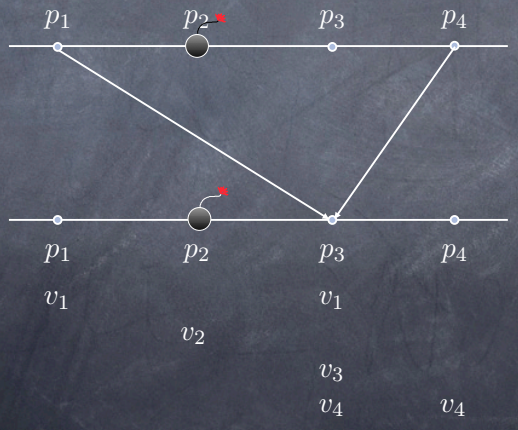
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

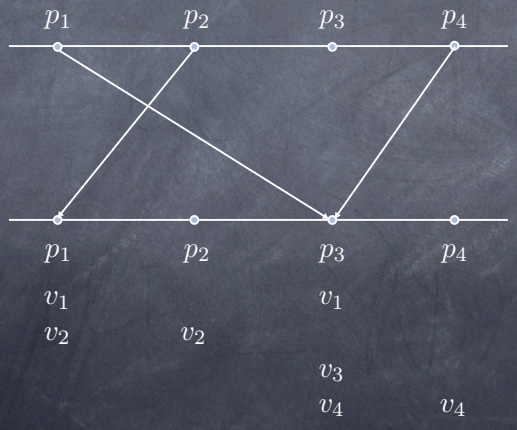
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

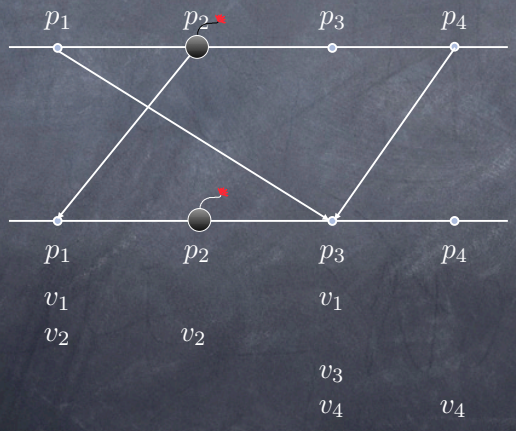
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

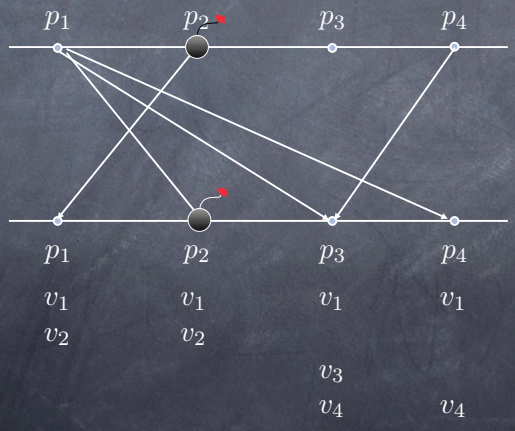
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

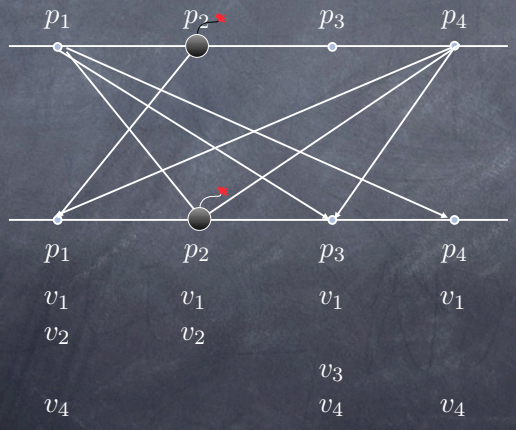
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

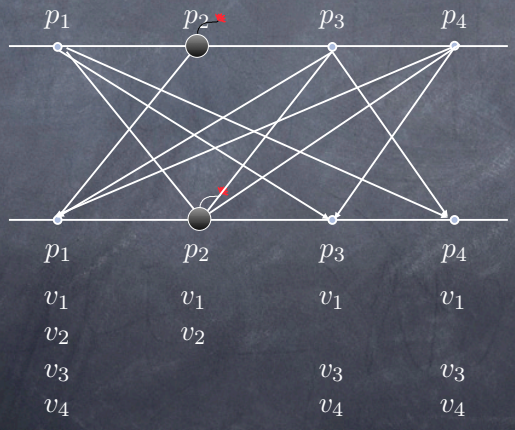
Can p_3 decide?



An execution

Suppose $v_1 = v_3 = v_4$ at the end of round 1

Can p_3 decide?



Echoing values

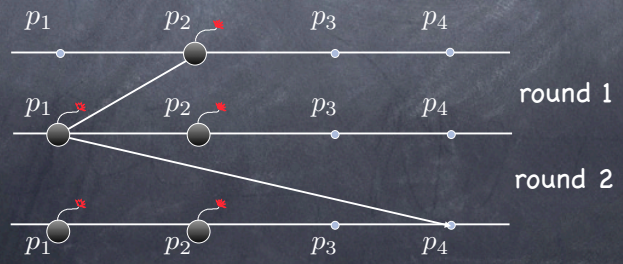
- A process that receives a proposal in round 1, relays it to others during round 2.

Echoing values

- A process that receives a proposal in round 1, relays it to others during round 2.
- Suppose p_3 hasn't heard from p_2 at the end of round 2. Can p_3 decide?

Echoing values

- A process that receives a proposal in round 1, relays it to others during round 2.
- Suppose p_3 hasn't heard from p_2 at the end of round 2. Can p_3 decide?



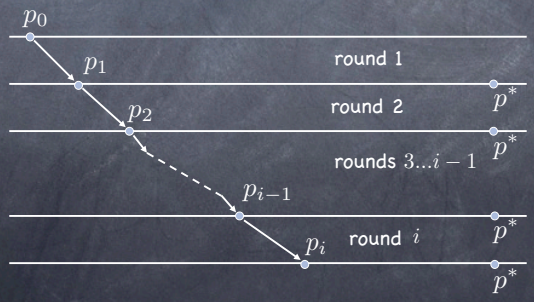
What is going on

- A correct process p^* has not received all proposals by the end of round i. Can p^* decide?
- Another process may have received the missing proposal at the end of round i and be ready to relay it in round $i+1$

Dangerous Chains

Dangerous chain

The last process in the chain is correct, all others are faulty



Living dangerously

How many rounds can a dangerous chain span?

- f faulty processes
- at most $f+1$ nodes in the chain
- spans at most f rounds

It is safe to decide by the end of round $f+1$!

Easy enough, right?

MESSAGES TAKE TIME

Does it matter how much?

AND YET...

Should it matter for
CORRECTNESS?

Assumptions are vulnerabilities!

ASYNCHRONOUS SYSTEMS

NO centralized clock

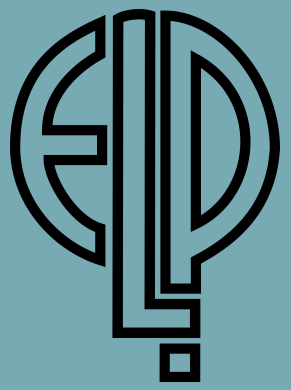
NO upper bound on the relative speed of processes

NO upper bound on message delivery time

CONSENSUS† IS IMPOSSIBLE IN AN ASYNCHRONOUS SYSTEM*

†deterministic

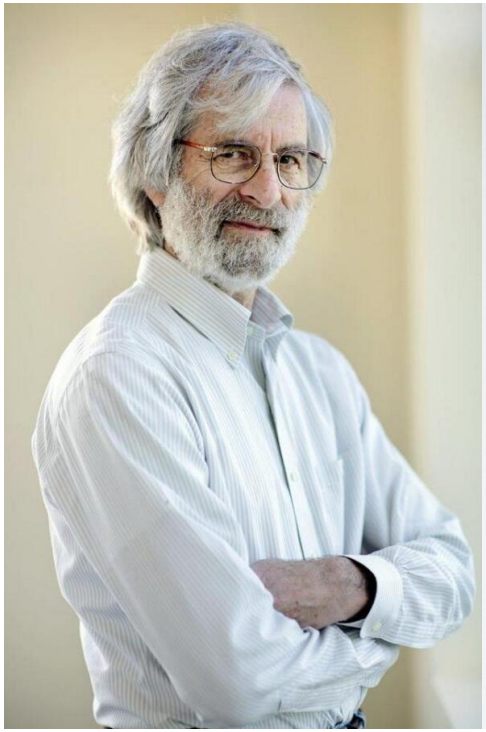
*in the presence of failures



CONSENSUS† IS IMPOSSIBLE IN AN ASYNCHRONOUS SYSTEM*

†deterministic

*in the presence of failures



Paxos

Always safe

Ready to pounce on liveness

The Part-Time Parliament

- Parliament determines laws by passing sequence of numbered decrees
- Direct democracy: Citizens/ Legislators leave and enter the chamber at arbitrary times
- No centralized records: each legislator carries a ledger recording the approved decrees

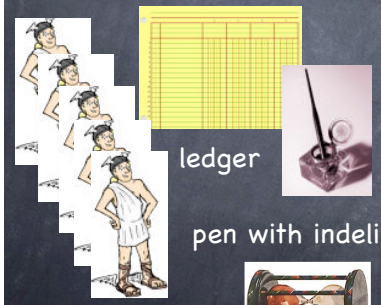
Government 101

- No two ledgers contain contradictory information
- If a majority of legislators are in the Chamber and no one enters or leaves the Chamber for a sufficiently long time, then
  - any decree proposed by a legislator is eventually passed
  - any passed decree appears on the ledger of every legislator

"In a world..."

Political intrigue!

Funky equipment!



ledger

pen with indelible ink

hourglass

messengers!

Λαμπσων

Δἴκστρα

Λισκωφ

Mysterious characters

Σθνειδερ

Σκεεν

Πνυελι

Δωλεφ

Δφωρκ

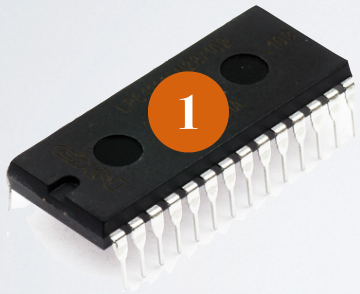
and more!

IT'S (ALMOST) EVERYWHERE!

Production use of Paxos [edit]

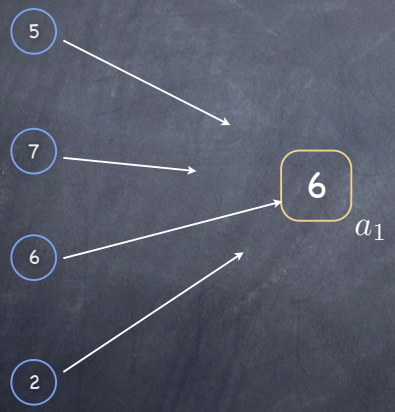
- The Petal project from DEC SRC was likely the first system to use Paxos, in this case for widely replicated global information (e.g., which machines are in the system).[20]
- Google uses the Paxos algorithm in their Chubby distributed lock service in order to keep replicas consistent in case of failure. Chubby is used by BigTable which is now in production in Google Analytics and other products.
- The Infinit peer-to-peer file system relies on Paxos to maintain consistency among replicas while allowing for quorums to evolve in size.
- Google Spanner and Megastore use the Paxos algorithm internally.
- The OpenReplica replication service uses Paxos to maintain replicas for an open access system that enables users to create fault-tolerant objects. It provides high performance through concurrent rounds and flexibility through dynamic membership changes.
- IBM supposedly uses the Paxos algorithm in their IBM SAN Volume Controller product to implement a general purpose fault-tolerant virtual machine used to run the configuration and control components of the storage virtualization services offered by the cluster.[*citation needed*]
- Microsoft uses Paxos in the Autopilot cluster management service from Bing.
- WANdisco have implemented Paxos within their DConE active-active replication technology.[21]
- XtreemFS uses a Paxos-based lease negotiation algorithm for fault-tolerant and consistent replication of file data and metadata.[22]
- Heroku uses Doozerd which implements Paxos for its consistent distributed data store.
- Ceph uses Paxos as part of the monitor processes to agree which OSDs are up and in the cluster.
- The Clustrix distributed SQL database uses Paxos for distributed transaction resolution.
- Neo4j HA graph database implements Paxos, replacing Apache ZooKeeper from v1.9
- VMware NSX Controller uses Paxos-based algorithm within NSX Controller cluster.
- Amazon Web Services uses the Paxos algorithm extensively to power its platform.[23]
- Nutanix implements the Paxos algorithm in Cassandra for metadata.
- Apache Mesos uses Paxos algorithm for its replicated log coordination.
- Windows Fabric used by many of the Azure services make use of the paxos algorithm for replication between nodes in a cluster
- Oracle NoSQL Database leverages Paxos-based automated fail-over election process in the event of a master replica node failure to minimize downtime.[24]

THE BASIC IDEA: THE WRITE-ONCE REGISTER



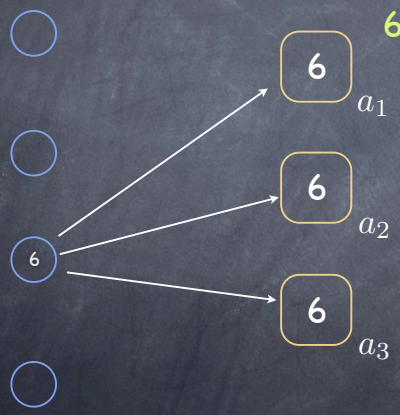
Participants forever
  read register
  if register contains a value then
    decide that value and halt
  else
    attempt to write own value

Deciding a value



Implement register using a single acceptor

What if the acceptor fails?



6 is decided!

- Decide only when a "large enough" set of acceptors accepts
- Using a majority set guarantees that at most one value is decided
- A value is decided once it is accepted by a majority of acceptors

Accepting a value

- Suppose only one value is proposed by a single proposer.
- That value should be chosen!
- First requirement:

  P1: An acceptor must accept the first proposal that it receives

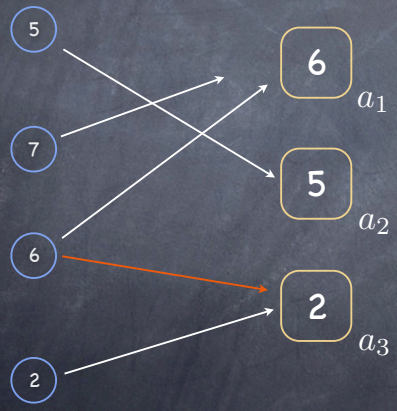
Accepting a value

- Suppose only one value is proposed by a single proposer.
- That value should be chosen!
- First requirement:

  P1: An acceptor must accept the first proposal that it receives

- ...but what if we have multiple proposers, each proposing a different value?

P1 & multiple proposers

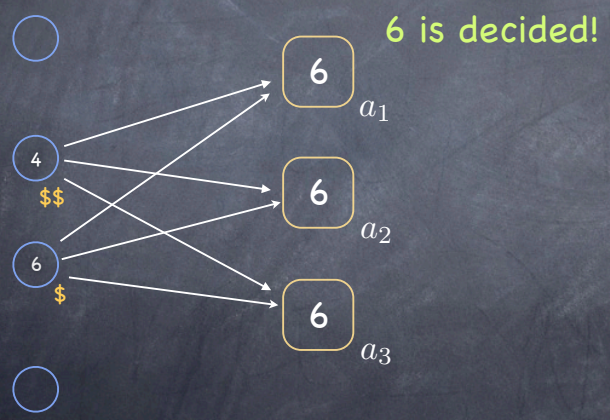


No value is decided!

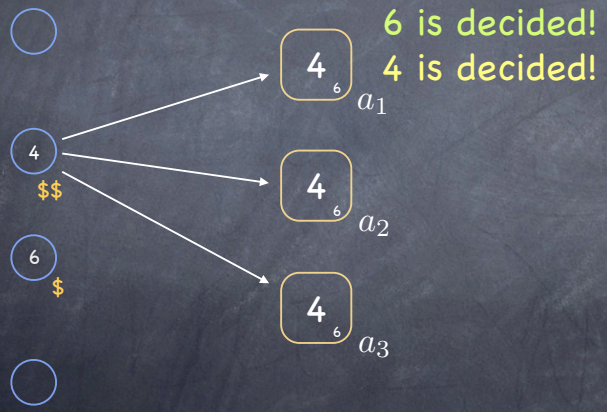
Sold!

- Acceptors sell their loyalty at auction
  - Once an acceptor agrees to a bid, it will never accept values proposed by a lower bidder
  - To have its value decided, a proposer must secure loyalty of (be the highest bidder for) a majority of acceptors

Write once?



Write once?



Easy!

- Once v is decided, acceptors cannot accept any value other than v!
  - but acceptors don't talk to one another...
- Once v is decided, proposers cannot propose any value other than v!

But how?

majority of acceptors

Proposers read ~~register~~ before attempting to write

Proposing

Phase 1

Read

Phase 2

Write

(ensuring one value)

Proposing

Phase 1

Proposer sends a bid

Acceptor response returns

- whether bid is accepted
- set of values previously accepted by acceptor (with corresponding bid)

Reading the register

- No acceptor returns a value associated with a lower bid
  - No value with a lower bid has been decided
  - No value with a lower bid will be decided
- Proposer can propose what it pleases!

Reading the register

- Acceptors returns one or more values associated with a lower bid


8


6


10

Proposing

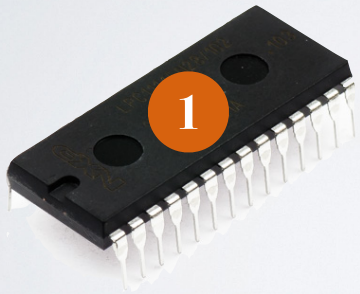
Phase 2

Proposer sends value determined in Phase 1 and corresponding bid

Acceptor response returns
- whether bid is accepted

If a majority of acceptors accepts bid, the corresponding value is decided (written to the register)

THE BASIC IDEA: THE WRITE-ONCE REGISTER



Participants forever
  read register
  if register contains a value then
    decide that value and halt
  else
    attempt to write own value