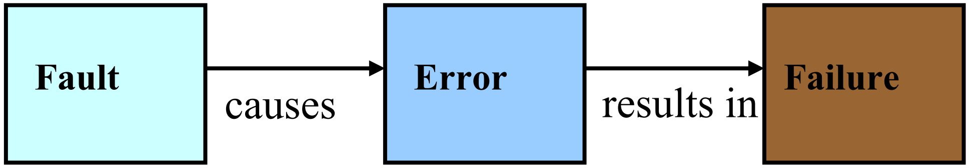
Fault Tolerance Middleware for Distributed Systems

CompSci 237
Prof. Nalini Venkatasubramanian
(with slides/animations adapted from Prof. Ghosh, Uof Iowa and Prof. Gupta, UIUC, Prof. Birman, Cornell Univ., Prof. Lynch, MIT, Prof. Zhang, ETS Montreal)

Fundamentals



Fault

causes

Error

results in

Failure

Fault - defect within the system (hardware/software/network)

Error – is observed by a deviation from the expected behavior of the system

Failure occurs when the system can no longer perform as required (does not meet spec)

Fault Tolerance – is ability of system to provide a service, even in the presence of errors

Why fault tolerance?

- Availability, reliability, dependability

Characterizing faults

– fault tolerance and limits

How to provide fault tolerance ?

- Replication/ Redundancy
- Checkpointing/rollback and message logging
- Hybrid

Attributes, Consequences and Strategies

Attributes

- Availability
- Reliability
- Safety
- Confidentiality
- Integrity
- Maintainability

What is a Dependable system?

How to distinguish faults?

Consequences

- Fault
- Error
- Failure

How to handle faults?

Strategies

- Fault prevention
- Fault tolerance
- Fault recovery
- Fault forcasting

Attributes of a Dependable System

- **System attributes:**
  - **Availability** – system always ready for use, or probability that system is ready or available at a given time
  - **Reliability** – property that a system can run without failure, for a given time
  - **Safety** – indicates the safety issues in the case the system fails
  - **Maintainability** – refers to the ease of repair to a failed system
- Failure in a distributed system = when a service cannot be fully provided
- System failure may be partial
- A single failure may affect other parts of a system (failure escalation)

Types of Fault (wrt time)

Hard or Permanent Faults

- repeatable error, e.g. failed component, power fail, fire, flood, design error, sabotage

Soft Faults

- **Transient –** occurs once or seldom, often due to unstable environment (e.g. bird flies past microwave transmitter)
  - **(Hardware)** Arbitrary perturbation of the global state. May be induced by power surge, weak batteries, lightning, radio-frequency interferences, cosmic rays etc.
  - **(Software)** Heisenbugs temporary internal faults, intermittent. They are essentially permanent faults whose conditions of activation occur rarely or are not easily reproducible
- **Intermittent –** occurs randomly, but where factors influencing fault are not clearly identified, e.g. unstable component
- **Operator error –** human error

Over 99% of bugs in IBM DB2 production code are non-deterministic and transient (Jim Gray)

Types of Fault (wrt attributes)

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Amnesia crash	Lost history, reboot
Pause crash	Remembers state before crash, can be recovered
Halting crash	Hardware failure, must be replaced or re-installed
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure	The server's response is incorrect
Value failure	The value of the response is wrong
State transition failure	The server deviates from the correct flow of control
Arbitrary failure (Byzantine)	A server may produce arbitrary responses at arbitrary times (includes malicious behavior)

Crash failures

Crash failure = the process halts. It is *irreversible*.

- Synchronous systems: easy to detect crash failure (using *heartbeat signals* and timeout).
- Asynchronous systems: Hard. Not possible to distinguish between a process that has crashed, and a process that is running very slowly.

Some failures may be complex and nasty. Fail-stop failure is a *simple abstraction* that *mimics* crash failure when program execution becomes arbitrary. Implementations help detect which processor has failed. If a system cannot tolerate fail-stop failure, then it cannot tolerate crash.

More Failure Classification

Fail-stop Failures

- crash failures that can be *reliably* detected

Fail-noisy Failures

- crash failures that can *eventually* be detected
- there may be some a priori *unknown time* in which the detection is unreliable

Fail-silent Failures

- can not *distinguish* crash failures from omission failures

Fail-safe Failures

- *arbitrary* failures but *benign*

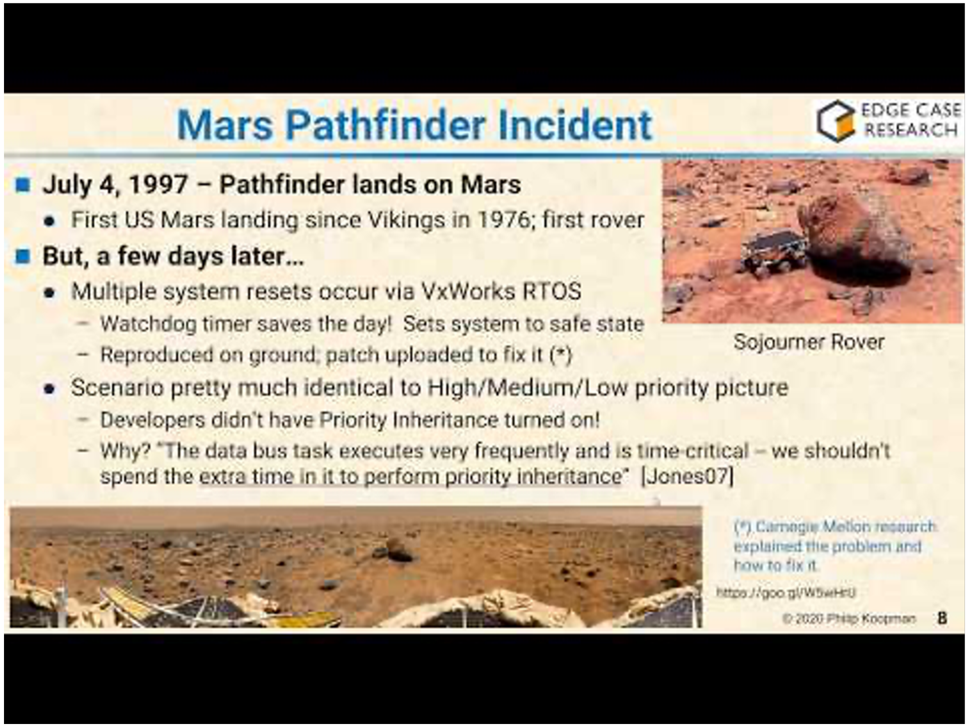
Fail-arbitrary Failures

- failures may be *unobservable* in addition to being *harmful*

Temporal failures

Inability to meet deadlines – correct results are generated, but too late to be useful. Very important in real-time systems.

- Poor algorithms
- Poor design strategy
- Loss of synchronization among the processor clocks



Mars Pathfinder Incident

- July 4, 1997 – Pathfinder lands on Mars
  - First US Mars landing since Vikings in 1976; first rover
- But, a few days later...
  - Multiple system resets occur via VxWorks RTOS
    - Watchdog timer saves the day! Sets system to safe state
    - Reproduced on ground; patch uploaded to fix it (*)
  - Scenario pretty much identical to High/Medium/Low priority picture
    - Developers didn't have Priority Inheritance turned on!
    - Why? "The data bus task executes very frequently and is time-critical – we shouldn't spend the extra time in it to perform priority inheritance" [Jones07]

Sojourner Rover

(*) Carnegie Mellon research explained the problem and how to fix it.

https://goo.gl/W5wHrU

© 2020 Philip Koopman 8

Strategies to Handle Faults

- Fault avoidance

Techniques aim to **prevent** faults from entering the system during design stage

- Fault removal

Methods attempt to **find** faults within a system before it enters service

- Fault detection

Techniques used during service to **detect** faults within the operational system

- Fault tolerance and recovery

Techniques designed to **tolerant** faults, i.e. to allow the system operate correctly in the presence of faults.

Actions to identify and remove errors:

- Design reviews
- Testing
- Use certified tools
- Analysis:
- Hazard analysis
- Formal methods - proof & refinement

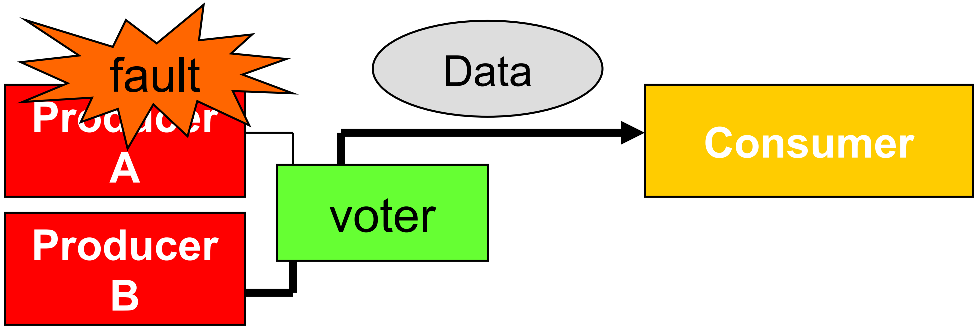
- No non-trivial system can be guaranteed free from error
- Must have an expectation of failure and make appropriate provision

Redundancy - handling failures

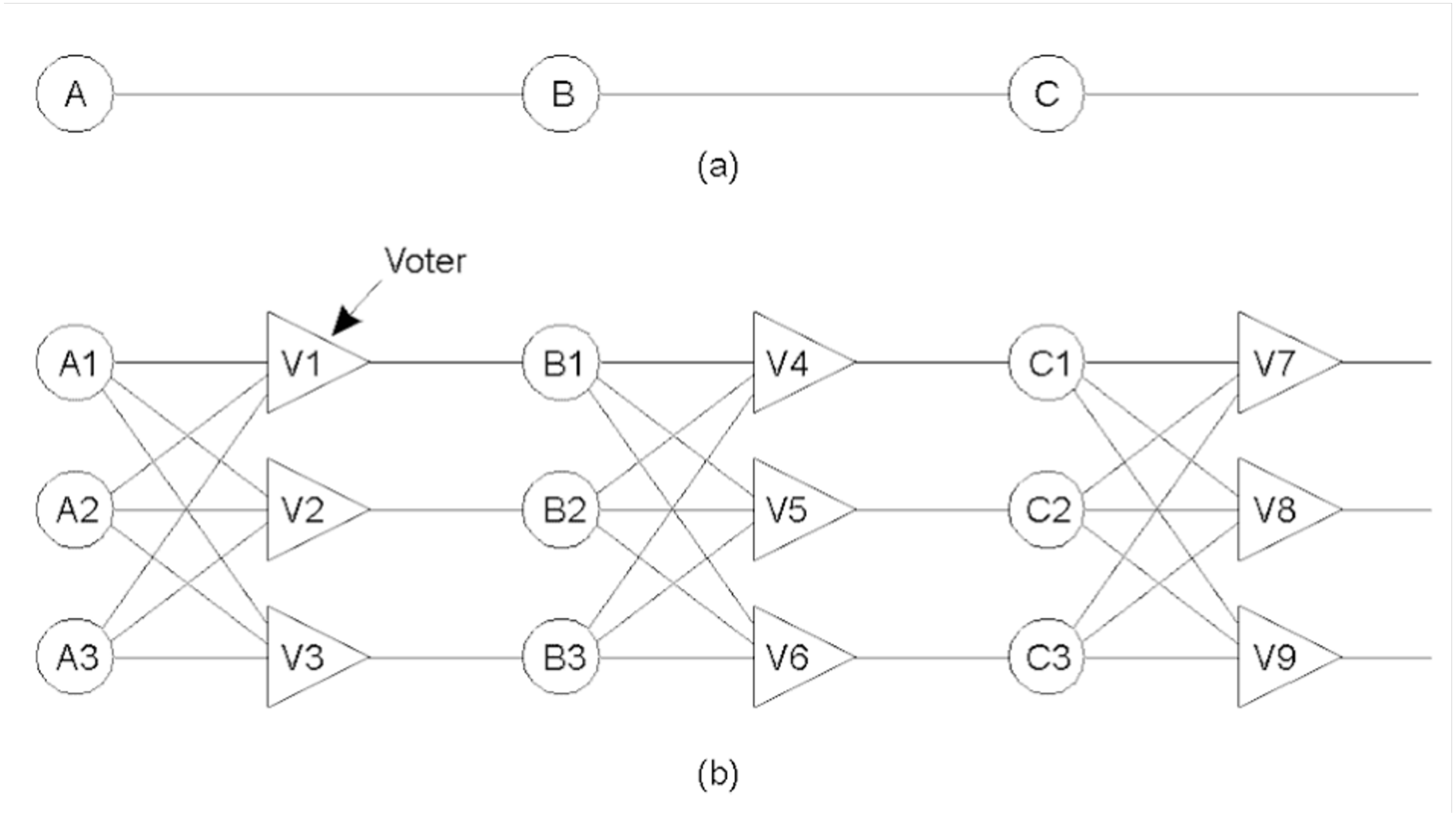
- Hardware redundancy
  - Use more hardware: RAID, Triple Modular Redundancy …
- Software redundancy
  - Use more software
- Information redundancy, e.g.
  - Parity bits
  - Error detecting or correcting codes
  - Checksums
- Temporal (time) redundancy
  - Repeating calculations and comparing results
  - For detecting transient faults

Modular Redundancy

- Modular Redundancy
  - Multiple identical replicas of hardware modules
  - Voter mechanism
    - Compare outputs and select the correct output
- Tolerate most hardware faults
- Effective but expensive



Triple Modular Redundancy

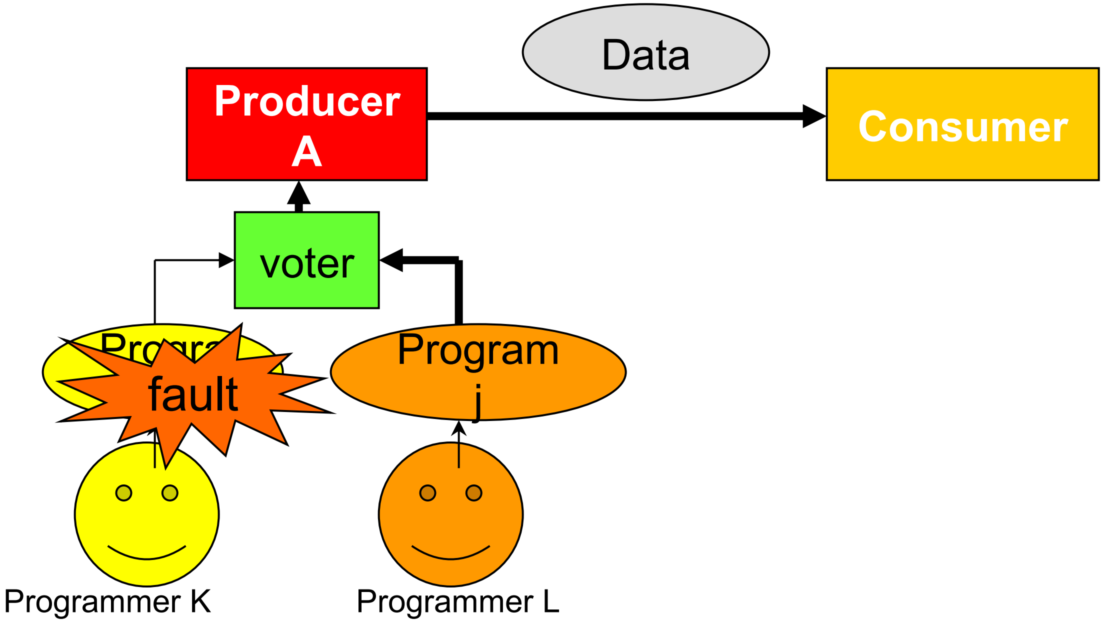


(a) Original circuit

(b) Triple modular redundancy

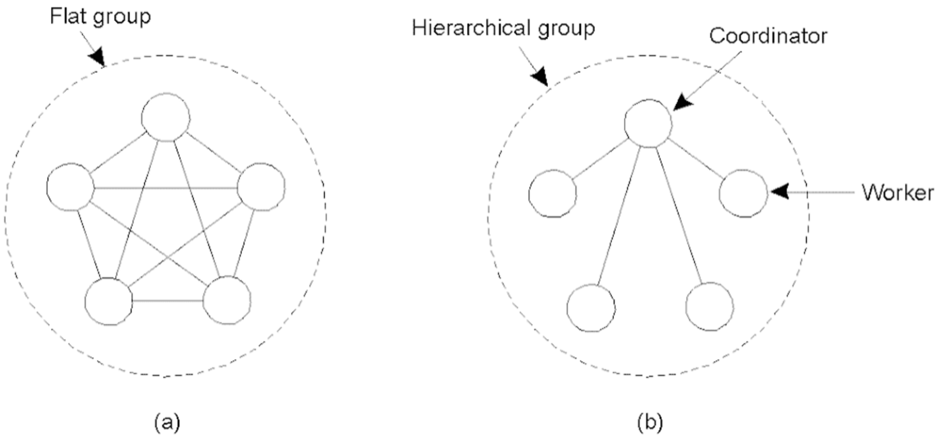
Software Redundancy: N-version Programming

- N-version Programming
  - Different versions by different teams, different implementations of the same specification
  - Different versions may not contain the same bugs
- Voter mechanism
- Tolerate some software bugs



Software Redundancy: Process Groups

- Organize several identical processes into a group - provides redundancy
  - Multicast/group communication ensures all members receive all messages (atomic, ordered)
  - Group membership
- If one process in a group fails, another process can take over
  - Processes dynamically join/leave group - replace failed group members??
  - Membership protocol ensures agreement on group membership at any given time
- Design Issue : Reaching agreement within a process group when one or more of its members cannot be trusted to give correct answers.



Fault Tolerance with Process Group

- A system is said to be k fault tolerant if it can survive faults in k components and still meets its specification.
- If the components (processes) fail silently, then having k + 1 of them is enough to provide k fault tolerant.
- If processes exhibit Byzantine failures (continuing to run when sick and sending out erroneous or random replies, a minimum 2k + 1 processes are needed.
- If we demand that a process group reaches an agreement, such as electing a coordinator, synchronization, etc., we need even more processes to tolerate faults .

Failure Detection: Synchronous vs. Asynchronous

- Single system – everything stops; Distributed system - some parts may continue

- Synchronous Distributed System
  - Each message is received within bounded time
  - Each step in a process takes lb < time < ub; Each local clock's drift has a known bound
  - Example: Multiprocessor systems

- Asynchronous Distributed System
  - No bounds on message transmission delays
  - No bounds on process executionThe drift of a clock is arbitrary
  - Example: Internet, wireless networks, datacenters, most real systems

Failure detection

The design of fault-tolerant algorithms will be simple if processes can detect failures.

- Impossibility results assume failures cannot be observed.
- In synchronous systems with bounded delay channels, crash failures can definitely be detected using timeouts.
- In asynchronous distributed systems, the detection of crash failures is imperfect.

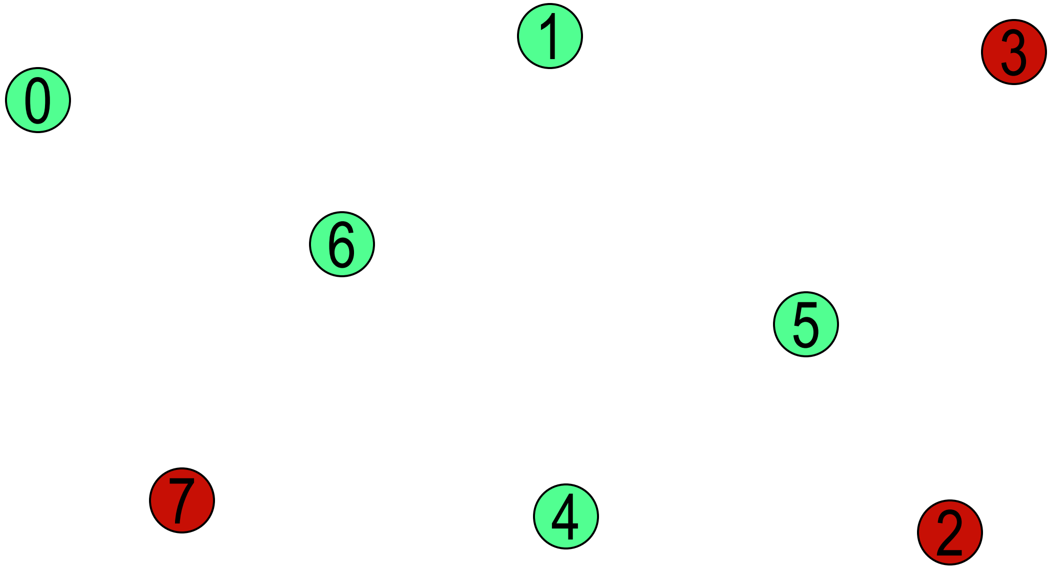
Designing failure detectors

Processes carry a *Failure Detector* to detect crashed processes.

Desirable Properties of a failure detector:

- *Completeness* – Every crashed process is suspected
- *Accuracy* – No correct process is suspected.
- *Other factors*
  - Speed -- time to first detection of a failure
  - Overhead -- load on member process, network message load

Example



0 suspects {1,2,3,7} to have failed.
Does this satisfy completeness?
Does this satisfy accuracy?

Classification of completeness

- **Strong completeness.** Every crashed process is eventually suspected by *every* correct process, and remains a suspect thereafter.

- **Weak completeness.** Every crashed process is eventually suspected by *at least one* correct process, and remains a suspect thereafter.

Note that we don't care what mechanism is used for suspecting a process.

Classification of accuracy

- **Strong accuracy.** No correct process is ever suspected.

- **Weak accuracy.** There is at least one correct process that is never suspected.

Eventual accuracy

A failure detector is *eventually strongly accurate*, if there exists a time **T** after which no correct process is suspected.

(*Before that time, a correct process be added to and removed from the list of suspects any number of times*)

A failure detector is *eventually weakly accurate*, if there exists a time **T** after which at least one process is no more suspected.

Classifying failure detectors

Perfect P. (Strongly) Complete and strongly accurate

Strong S. (Strongly) Complete and weakly accurate

Eventually perfect ◊P.

(Strongly) Complete and eventually strongly accurate

Eventually strong ◊S

(Strongly) Complete and eventually weakly accurate

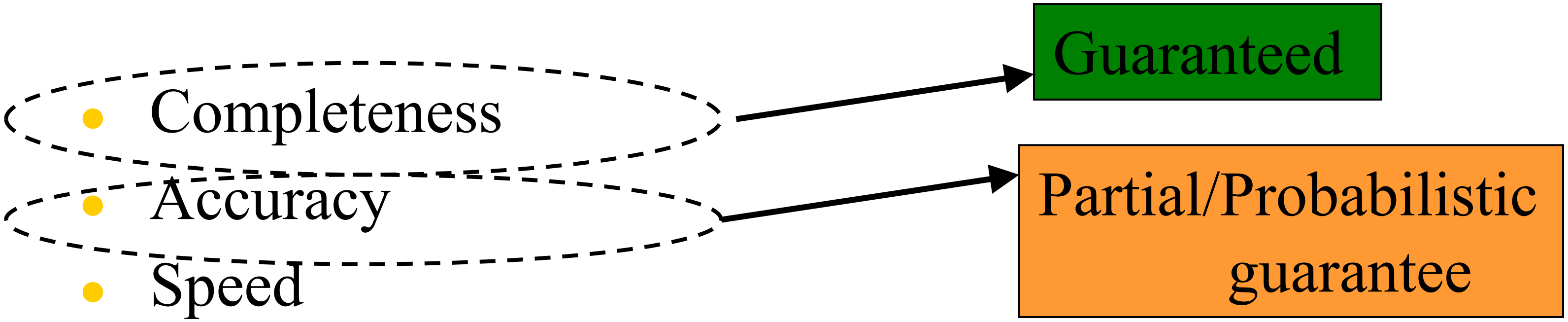
Other classes are feasible: W (weak completeness) and weak accuracy) and **◊W**

Failure detector properties

- **Completeness** = every process failure is eventually detected (no misses)
- **Accuracy** = every detected failure corresponds to a crashed process (no mistakes)

- Completeness and Accuracy
  - Can both be guaranteed 100% in a synchronous distributed system
  - Can never be guaranteed simultaneously in an asynchronous distributed system

What Real Failure Detectors Prefer

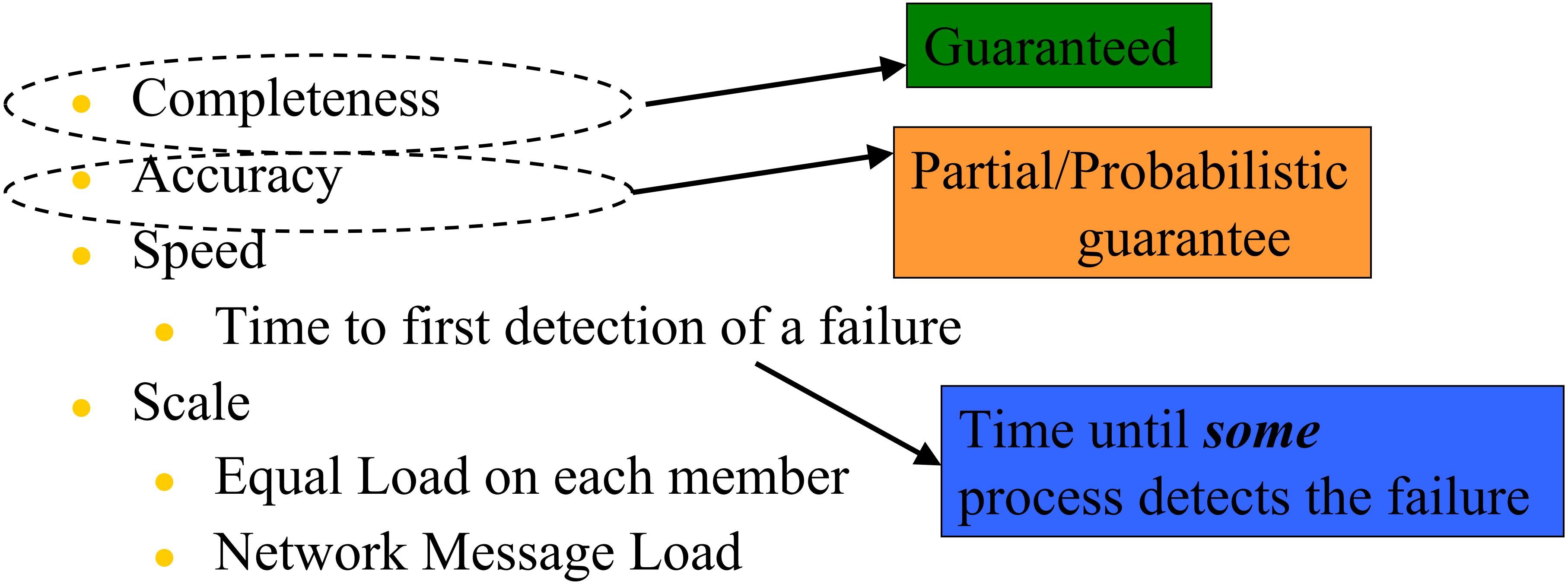


- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Guaranteed

Partial/Probabilistic guarantee

What Real Failure Detectors Prefer



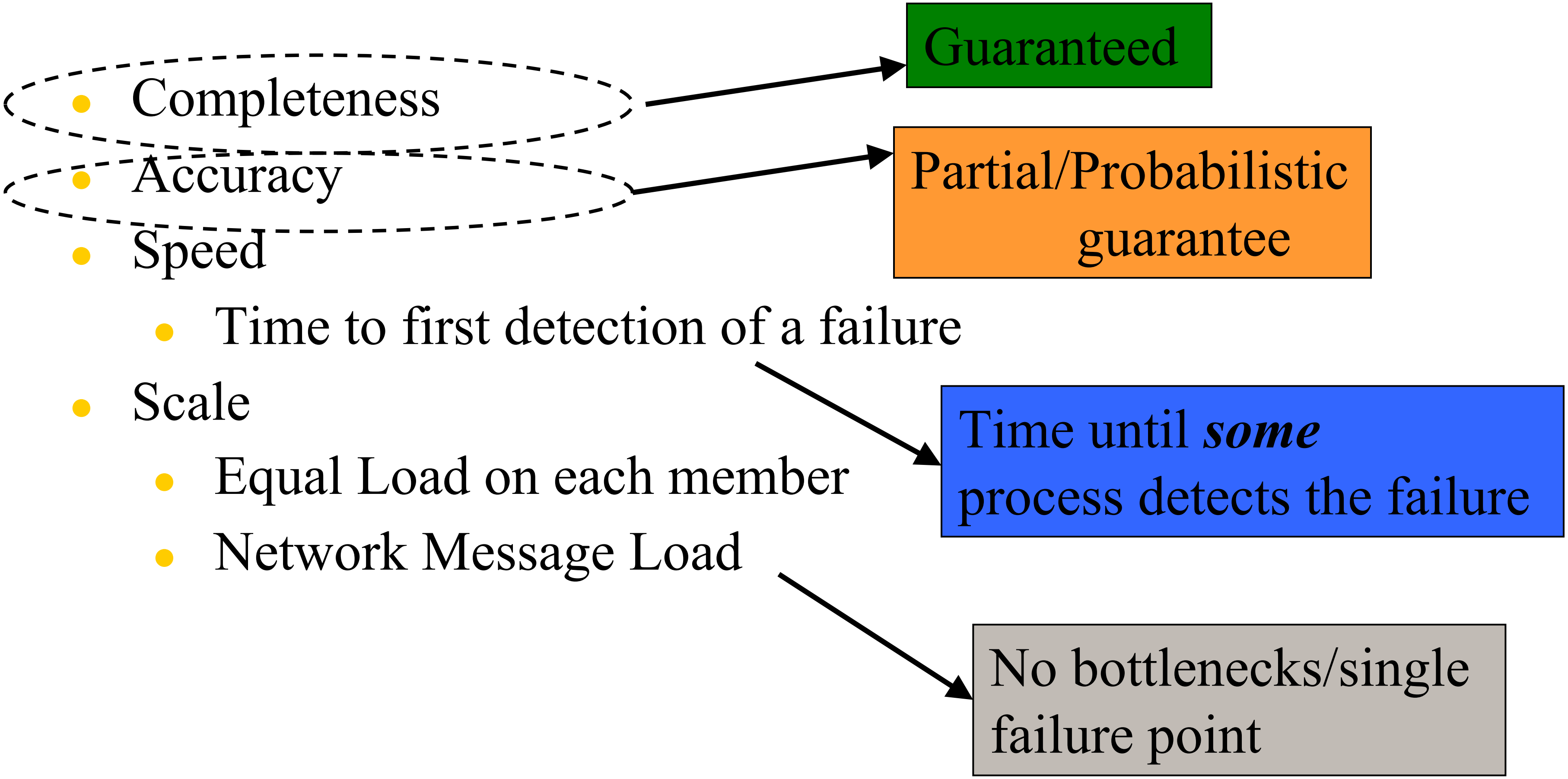
- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Guaranteed

Partial/Probabilistic guarantee

Time until ***some*** process detects the failure

What Real Failure Detectors Prefer



- Completeness
- Accuracy
- Speed
  - Time to first detection of a failure
- Scale
  - Equal Load on each member
  - Network Message Load

Guaranteed

Partial/Probabilistic guarantee

Time until ***some*** process detects the failure

No bottlenecks/single failure point

Detecting failures

p_i

p_j

Detecting failures

p_i

p_j

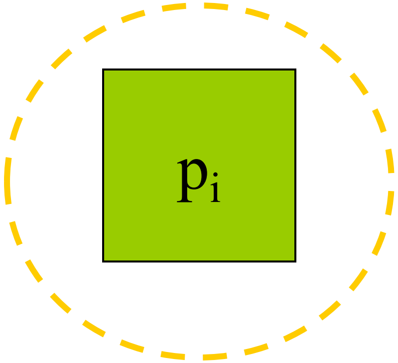
X

Crash-stop failure (p_j is a *failed* process)

Detecting failures

needs to know about pj's failure
(pi is a *non-faulty* process or
alive process)

crash-stop failure
(p_j is a *failed* process)

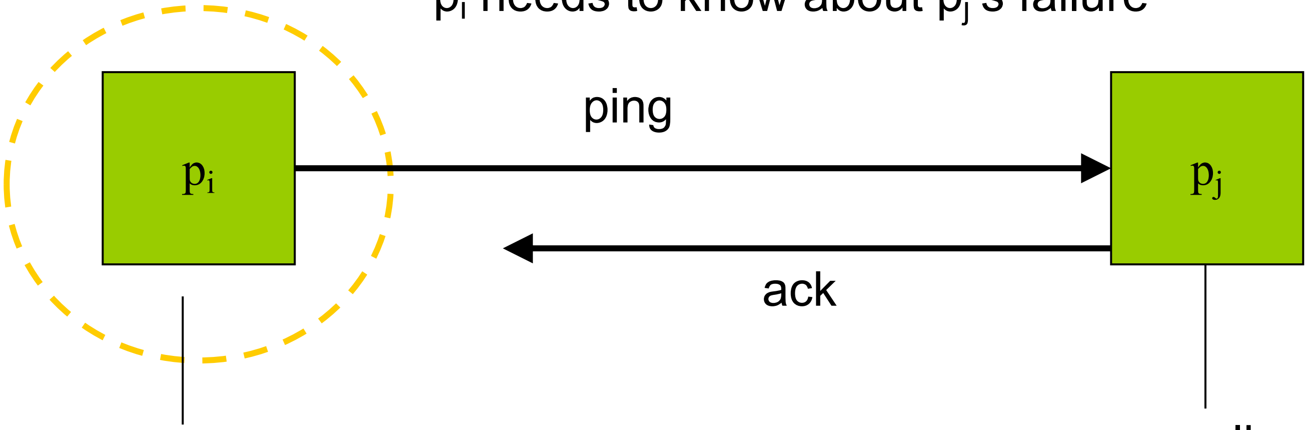


There are two main flavors of **failure detectors**

1. Ping-Ack (proactive)
2. Heartbeat (reactive)

Ping-ack protocol

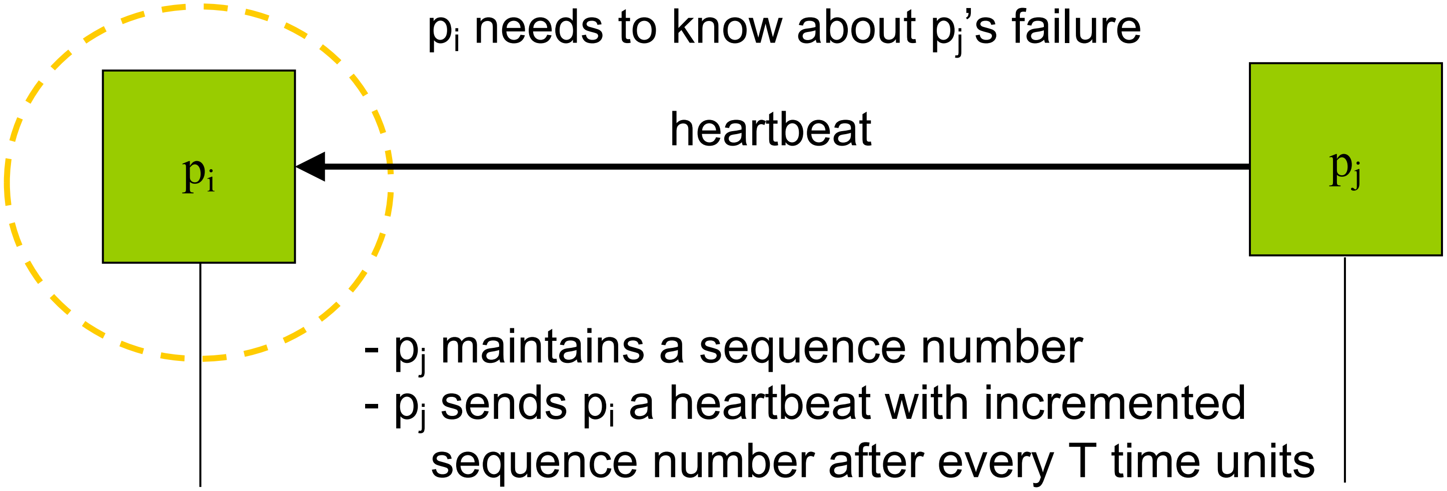
p_i needs to know about p_j's failure



- p_j replies

- p_i queries p_j once every T time units
- If p_j does not respond within another T time units of being sent the ping, p_i detects p_j as failed
- Worst case Detection *time* = *2T,* if p_j fails, then within *T* time units, p_i will send it a ping message. p_i will time out within another T time units.
- The waiting time *T* can be parameterized.

Heartbeat protocol



- p_j maintains a sequence number
- p_j sends p_i a heartbeat with incremented sequence number after every T time units

If p_i has not received a new heartbeat for the past, say *3T* time units, since it received the last heartbeat, then p_i detects p_j as failed

If T >> round trip time of messages, then worst case detection time ~ 3*T (why?)

The *3* can be changed to any positive number since it is a parameter

Synchronous DS case

- The Ping-ack and Heartbeat failure detectors are always "correct"
  - If a process p_j fails, then p_i will detect its failure as long as p_i itself is alive
- Why?
  - **Ping-ack**: set waiting time *T* to be > round trip time upper bound
    - p_i □ p_j latency + p_j processing + p_j □ p_i latency + p_i processing time
  - **Heartbeat**: set waiting time *3T* to be > round trip time upper bound

Satisfying completeness and accuracy in asynchronous DS

- **Impossible** because of arbitrary message delays & message losses
  - If a heartbeat/ack is dropped (or several are dropped) from p_j, then p_j will be mistakenly detected as failed □ inaccurate detection
  - How large would the T waiting period in ping-ack or *3T* heartbeat waiting period, need to be to obtain 100% accuracy?
  - In asynchronous systems, delay/losses on a network link are impossible to distinguish from a faulty process
- Heartbeat – satisfies completeness but not accuracy
- Ping-Ack – satisfies completeness but not accuracy

Completeness or accuracy in asynchronous DS

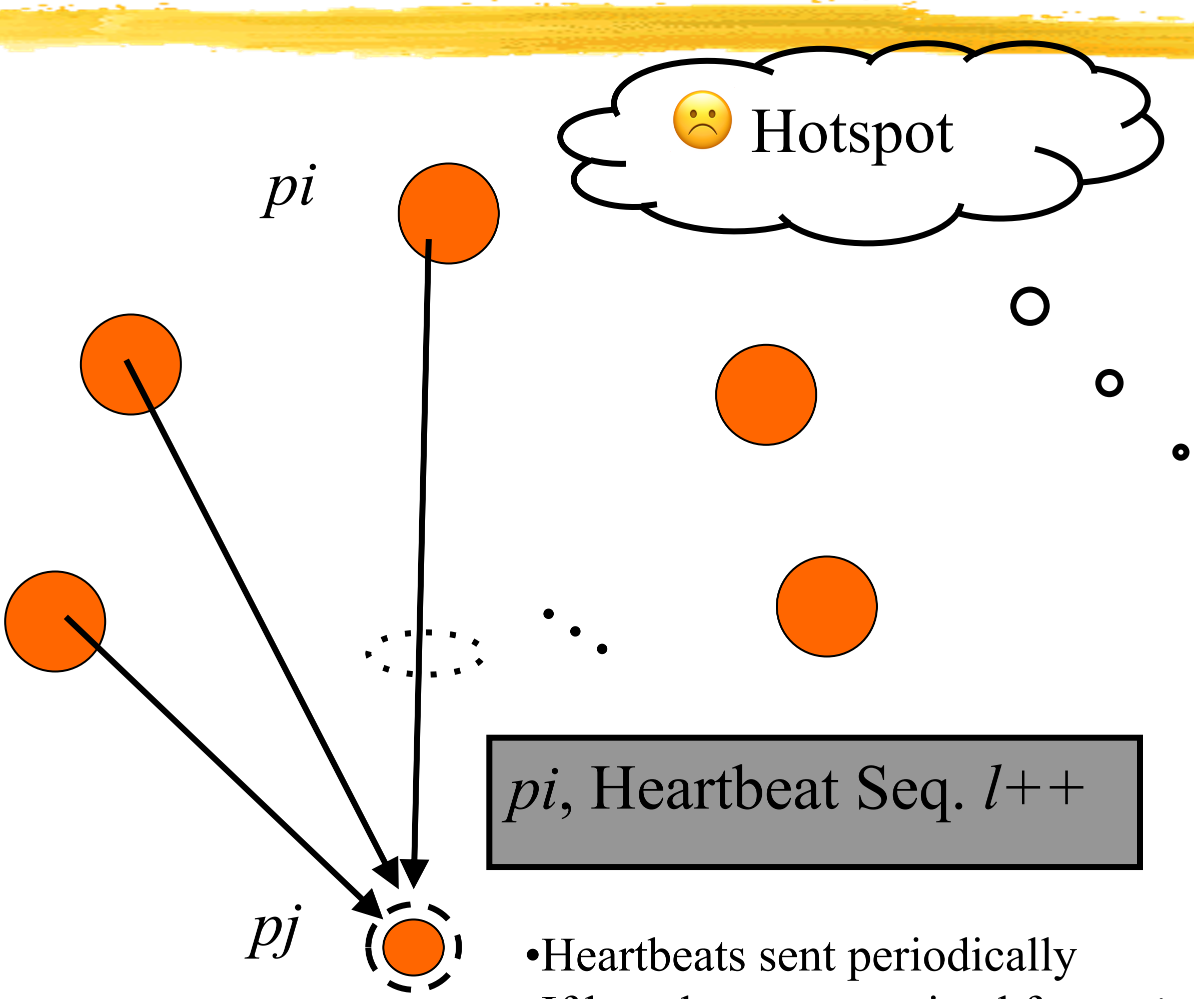
- Most failure detector implementations are willing to **tolerate some inaccuracy**, but **require 100% completeness**
- Many distributed apps designed assuming 100% completeness, e.g., P2P systems
  - "Err on the side of caution"
  - Processes not "stuck" waiting for other processes
- If error in identifying is made then victim process rejoins as a new process and catches up
- Hearbeating and Ping-ack provide
  - **Probabilistic accuracy**: for a process detected as failed, with some probability close to 1.0 (but not equal) it is true that it has actually crashed

Failure detection across the DS

- We want failure detection of not merely one process (p_j), but all processes in the DS
- Approaches:
  - Centralized heartbeat
  - Ring heartbeat
  - All-to-all heartbeat

Who guards the failure detectors?

Centralized Heartbeating

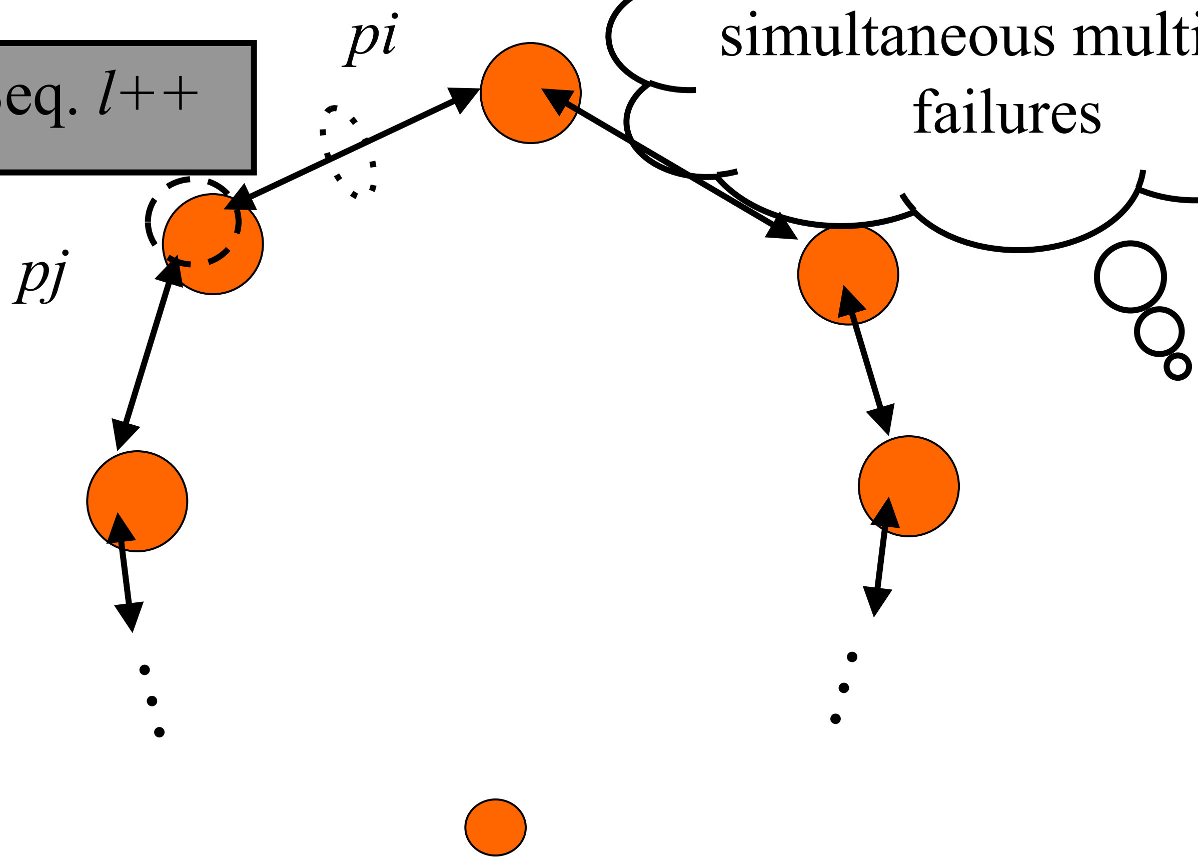


pi, Heartbeat Seq. *l*++

- Heartbeats sent periodically
- If heartbeat not received from *pi* within timeout, mark *pi* as failed

Ring Heartbeating

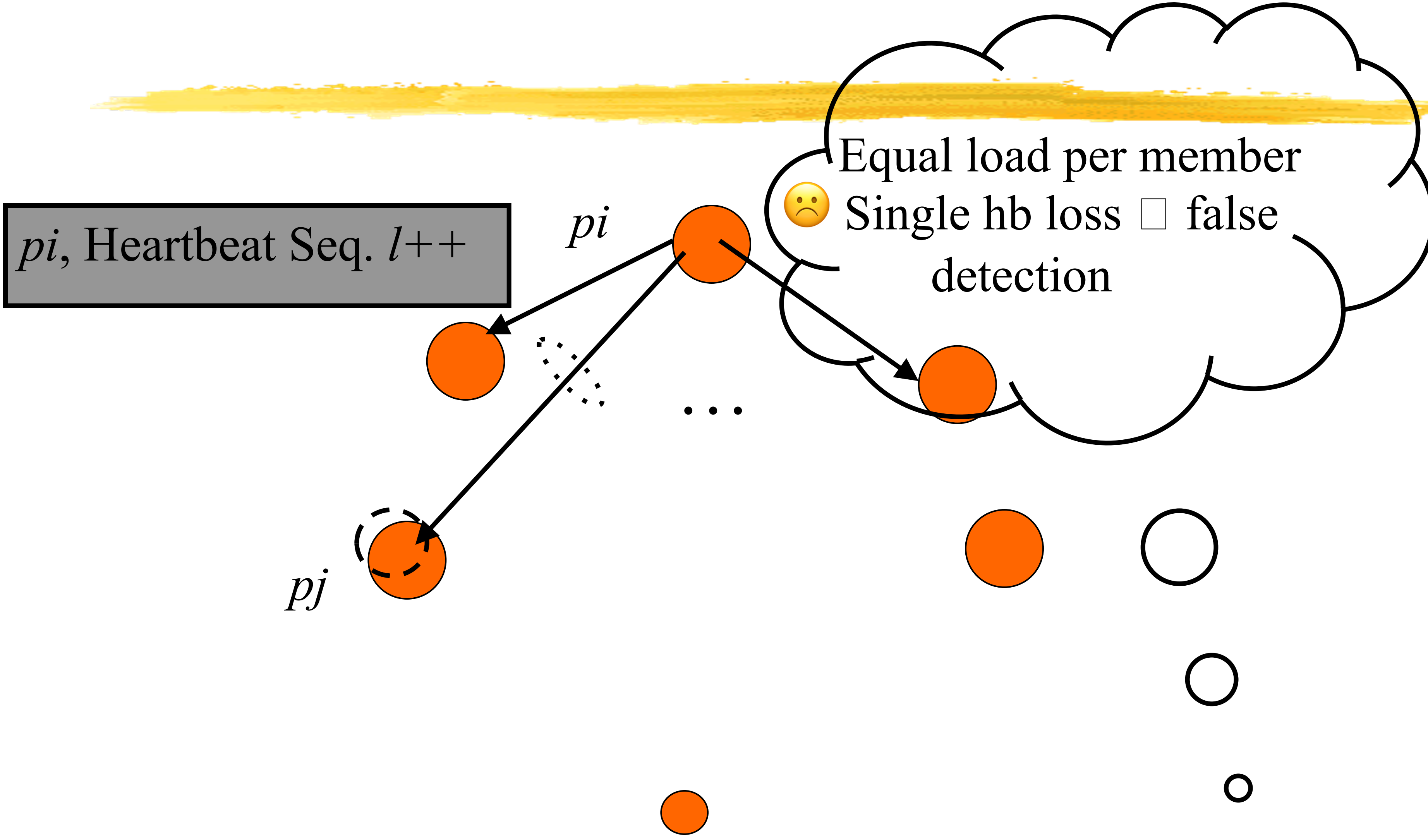
pi, Heartbeat Seq. *l*++



☹ Unpredictable on simultaneous multiple failures

Approach used in cluster settings

All-to-All Heartbeating



pi, Heartbeat Seq. *l*++

Equal load per member

☹ Single hb loss □ false detection

Variant - ***gossip style heartbeating*** (heartbeats with a member subset) -- AWS???
Determine gossip-period; send o(N) heartbeats to a subset every gossip period

Detection of omission failures

For **FIFO** channels: Use sequence numbers with messages.

(1, 2, 3, 5, 6 …) ⇒ message 4 is missing

Non-FIFO bounded delay channels - use timeout

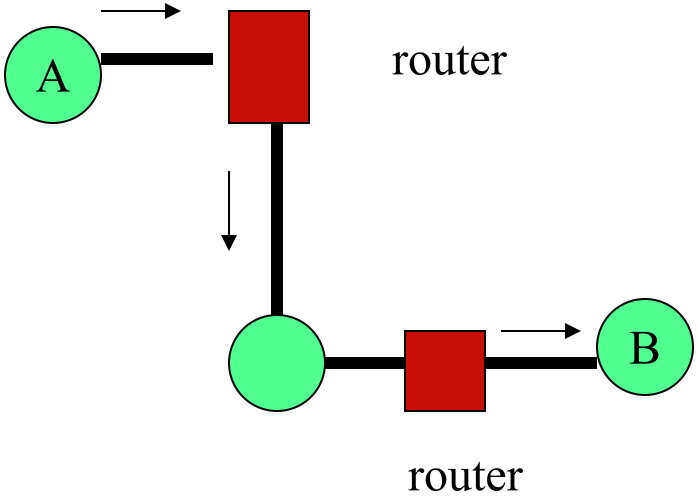
What about non-FIFO channels for which the upper bound of the delay is not known?

Use *unbounded sequence numbers* and acknowledgments.

But acknowledgments may be lost too!

Tolerating omission failures
A real example

A central issue in networking



Routers may drop messages, but reliable end-to-end transmission is an important requirement. If the sender does not receive an ack within a time period, it retransmits (it may so happen that the was not lost, so a duplicate is generated). This implies, the communication must tolerate **Loss, Duplication, and Re-ordering** of messages

Detection efficiency metrics

- **Bandwidth**:
  - the number of messages sent in the system during steady state (no failures)
  - Small is good
- **Detection time**
  - Time between a process crash and its detection
  - Small is good
- **Scalability**:
  - How do bandwidth and detection properties scale with N, the number of processes?
- **Accuracy**
  - Large is good

Accuracy metrics

- **False Detection Rate/False Positive Rate** (inaccuracy)
  - Multiple possible metrics
    1. Average number of failures detected per second, when there are in fact no failures
    2. Fraction of failure detections that are false

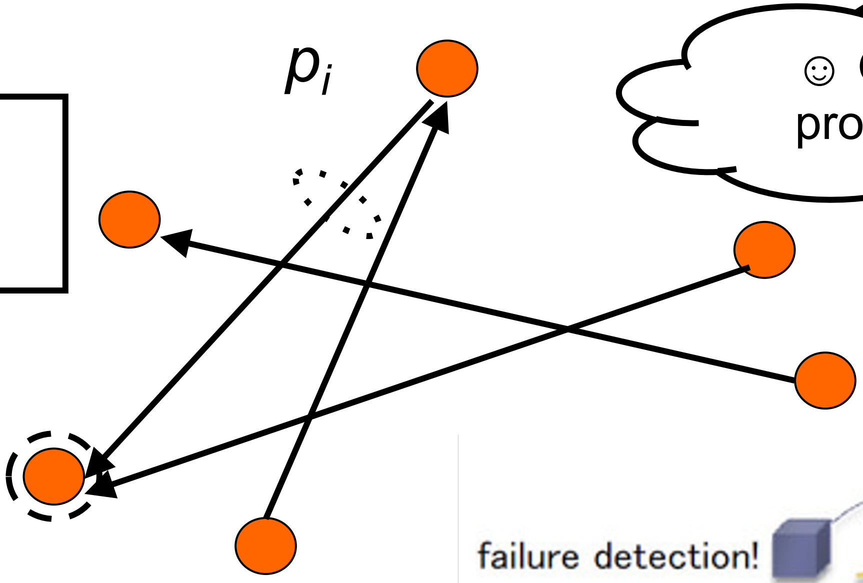
- **Tradeoffs**: If you increase the T waiting period in ping-ack or $3T$ waiting period in heartbeating what happens to:
  - Detection Time?
  - False positive rate?
  - Where would you set these waiting periods?

Membership protocols

- Maintain a list of other alive (non-faulty) processes at *each process* in the system
- Failure detector is a component in membership protocol
  - Failure of p_j detected □ delete p_j from membership list
  - New machine joins □ p_j sends message to everyone □ add p_j to membership list
- Flavors
  - Strongly consistent: all membership lists identical at all times (hard, may not scale)
  - Weakly consistent: membership lists not identical at all times
  - Eventually consistent: membership lists always moving towards becoming identical eventually (scales well)

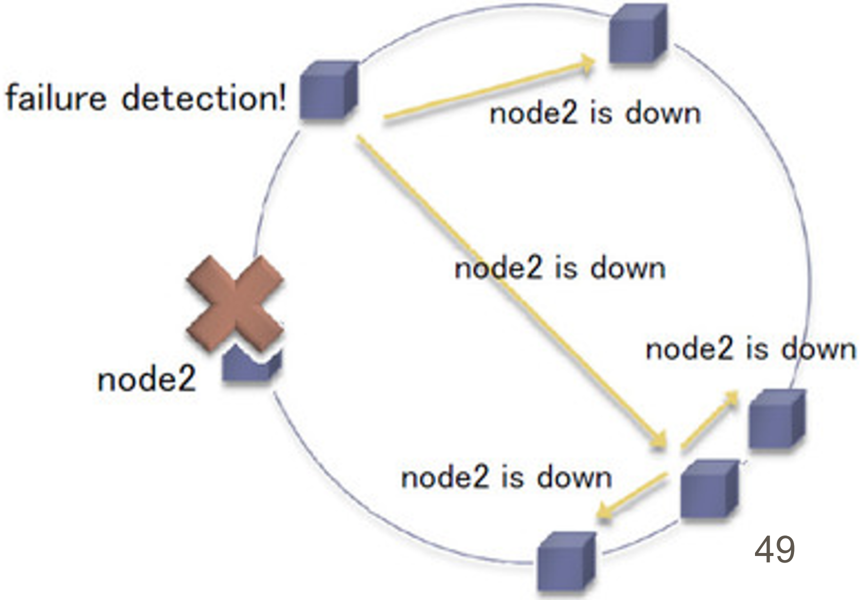
Gossip protocols

Array of Heartbeat Seq. *i* for member subset



☺ Good accuracy properties

- Mimic gossip in a social network, efficient to use due to DS large scale
- In a random search the **access time to any VM** is of at most n^3 for a regular graph and a third degree polynomial for any graph



Gossip based failure detection

1	10120	66
2	10103	62
3	10098	63
4	10111	65

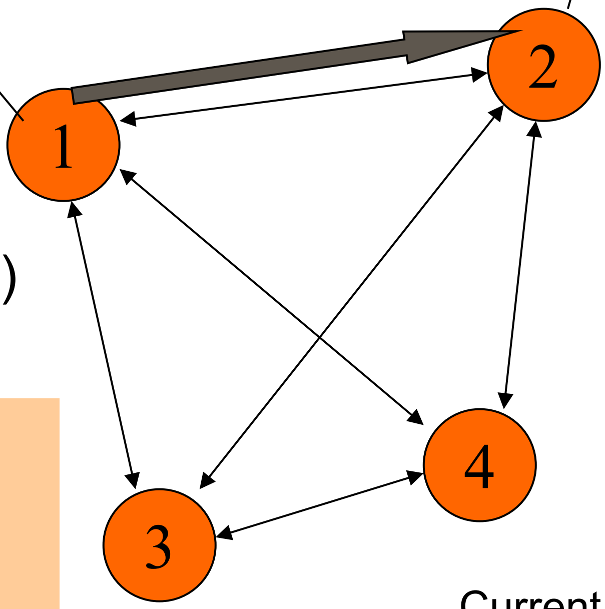
Address

Heartbeat Counter

Time (local)

Protocol

- Each process maintains a membership list
- Each process periodically increments its own heartbeat counter
- Each process periodically gossips its membership list
- On receipt, the heartbeats are merged, and local times are updated



1	10118	64
2	10110	64
3	10090	58
4	10111	65

1	10120	70
2	10110	64
3	10098	70
4	10111	65

Current time : 70 at node 2

(asynchronous clocks)

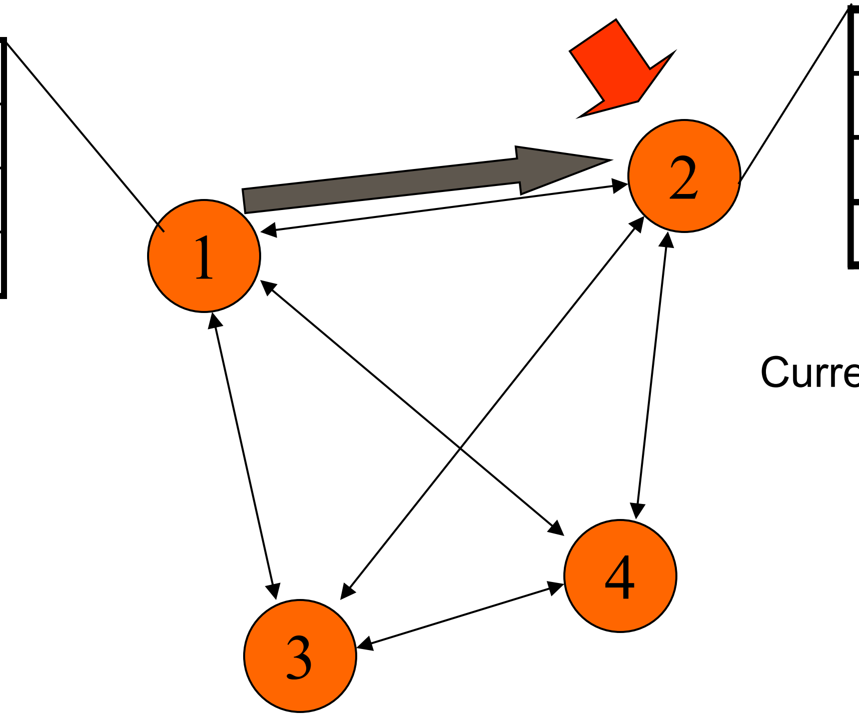
Gossip based failure detection

- *O(log(N))* time for a heartbeat update to propagate to everyone with high probability
- Very **robust** against failures – even if a large number of processes crash, most/all of the remaining processes still receive all heartbeats
- **Failure detection**: If the heartbeat has not increased for more than T_{fail} seconds,
  the member is considered failed
  - T_{fail} usually set to *O(log(N))*.
  - But entry not deleted immediately: wait another $T_{cleanup}$ seconds (usually = T_{fail})
    - Why?

Gossip based failure detection

- What if an entry pointing to a failed node is deleted right after T_{fail} (=24) seconds?

1	10120	66
2	10103	62
3	10098	55
4	10111	65



1	10120	66
2	10110	64
3 4	10098 10111	75 65
4	10111	65

Current time : 75 at node 2

- Solution: remember for another $T_{cleanup}$

What's the Best/Optimal we can do?

- *Worst case* load L* per member in the group (messages per second)
  - as a function of *T*, *PM(T)*, N
  - Independent Message Loss probability p_{ml}

$$\boxed{L^* = \frac{\log(PM(T))}{\log(p_{ml})} \cdot \frac{1}{T}}$$

Heartbeating

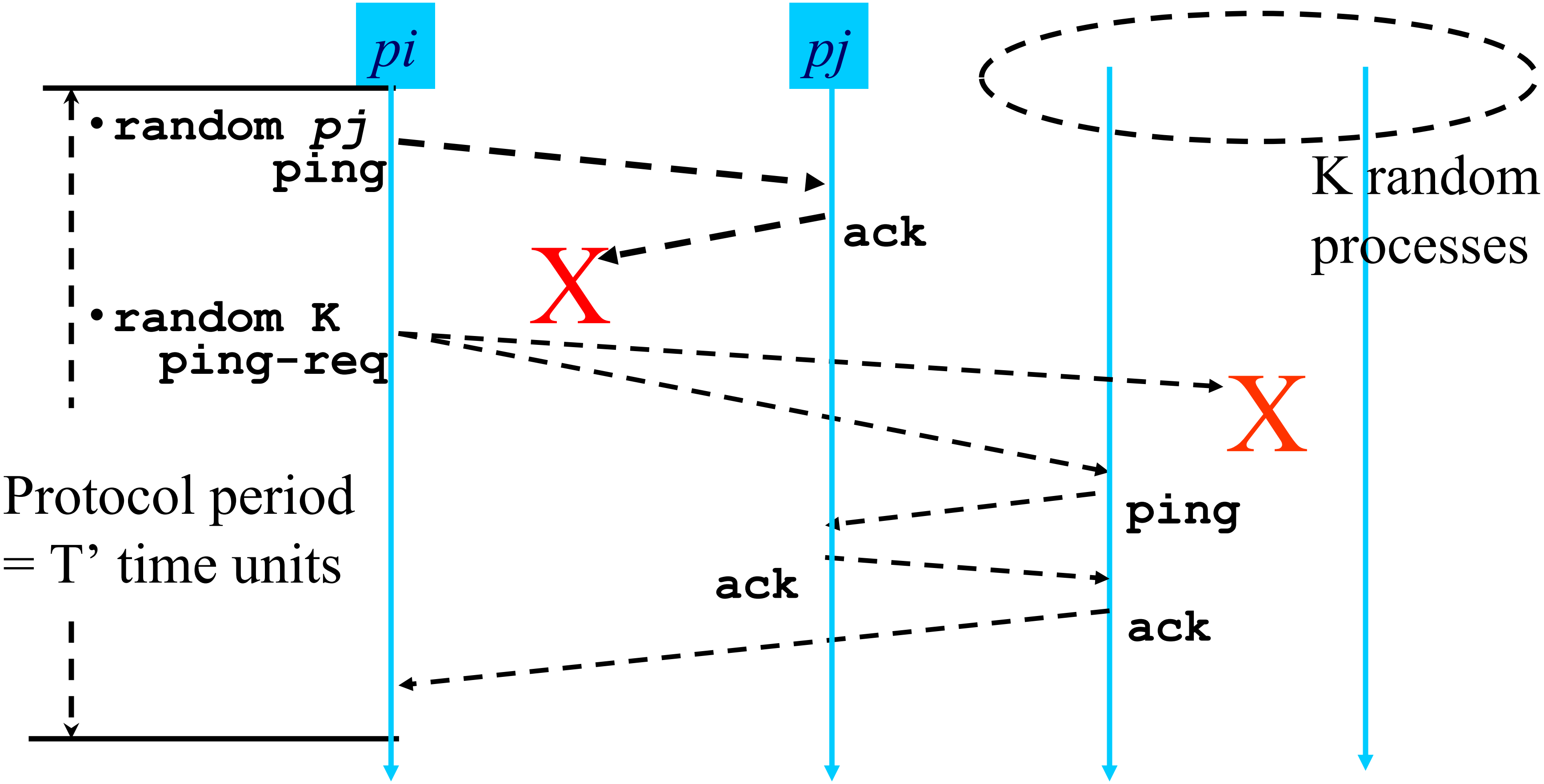
- Optimal L is independent of N (!)
- All-to-all and gossip-based: sub-optimal
  - L=O(N/T)
  - try to achieve simultaneous detection at ***all*** processes
  - fail to distinguish *Failure Detection* and *Dissemination* components

⍰Can we reach this bound?

⍰Key:

- Separate the two components
- Use a non heartbeat-based Failure Detection Component

SWIM Failure Detector Protocol



Detection Time

- Prob. of being pinged in T'= $1-(1-\frac{1}{N})^{N-1}=1-e^{-1}$
- $E[T]=T'.\frac{e}{e-1}$
- Completeness: *Any* alive member detects failure
  - Eventually
  - By using a trick: within worst case *O(N)* protocol periods

Accuracy, Load

- *PM(T)* is exponential in -*K*. Also depends on *pml* (and *pf*)
  - See paper
- $\boxed{\frac{L}{L^*} < 28}$ $\boxed{\frac{E[L]}{L^*} < 8}$ for up to 15 % loss rates

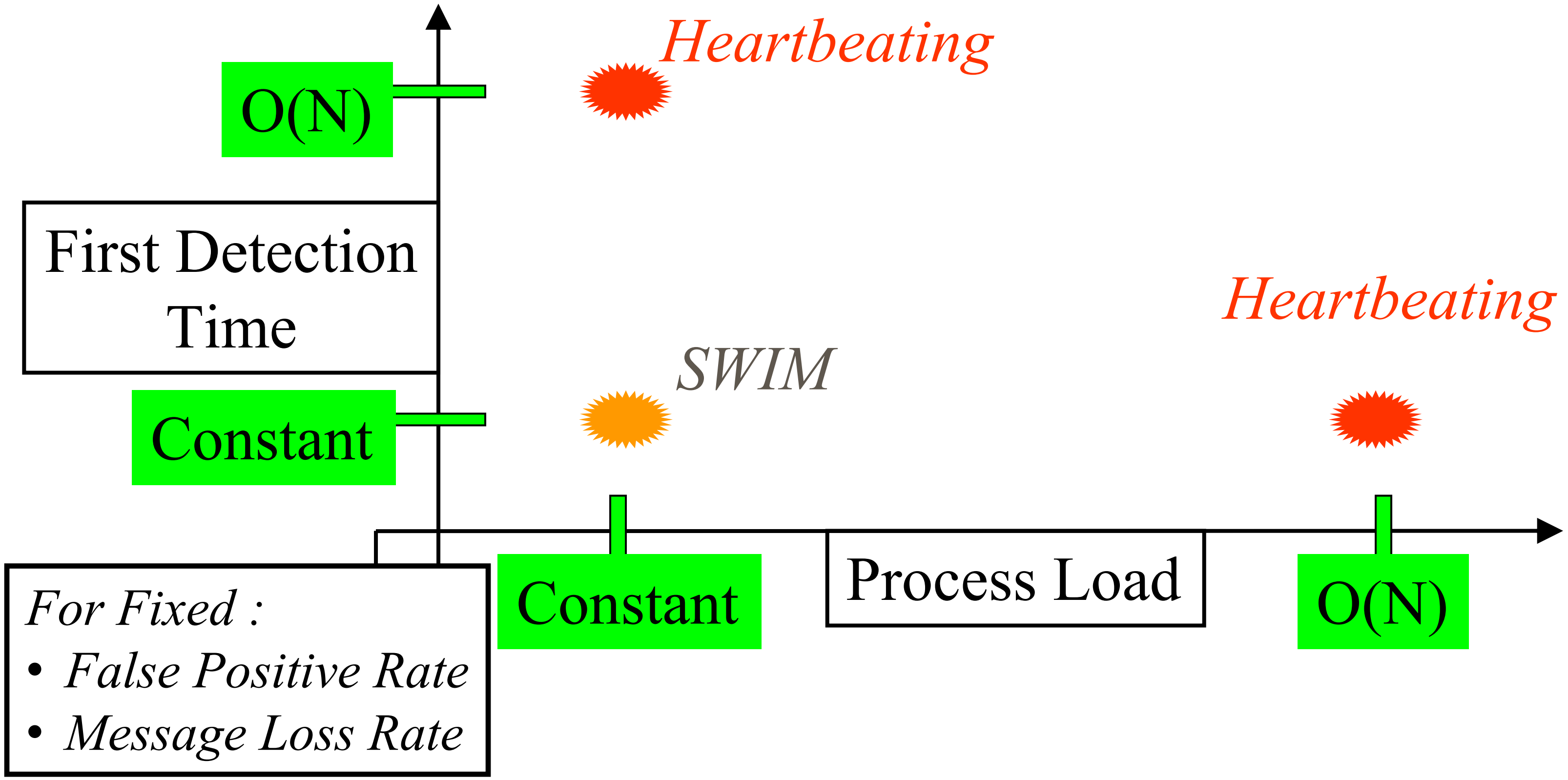
SWIM Failure Detector

Parameter	**SWIM**
First Detection Time	• Expected $\left[\frac{e}{e-1}\right]$ periods • Constant (independent of group size)
Process Load	• Constant per period • < 8 L* for 15% loss
False Positive Rate	• Tunable (via K) • Falls exponentially as load is scaled
Completeness	• Deterministic time-bounded • Within O(log(N)) periods w.h.p.

Time-bounded Completeness

- Key: select each membership element once as a ping target in a traversal
  - Round-robin pinging
  - Random permutation of list after each traversal
- Each failure is detected in worst case 2N-1 (local) protocol periods
- Preserves FD properties

SWIM versus Heartbeating

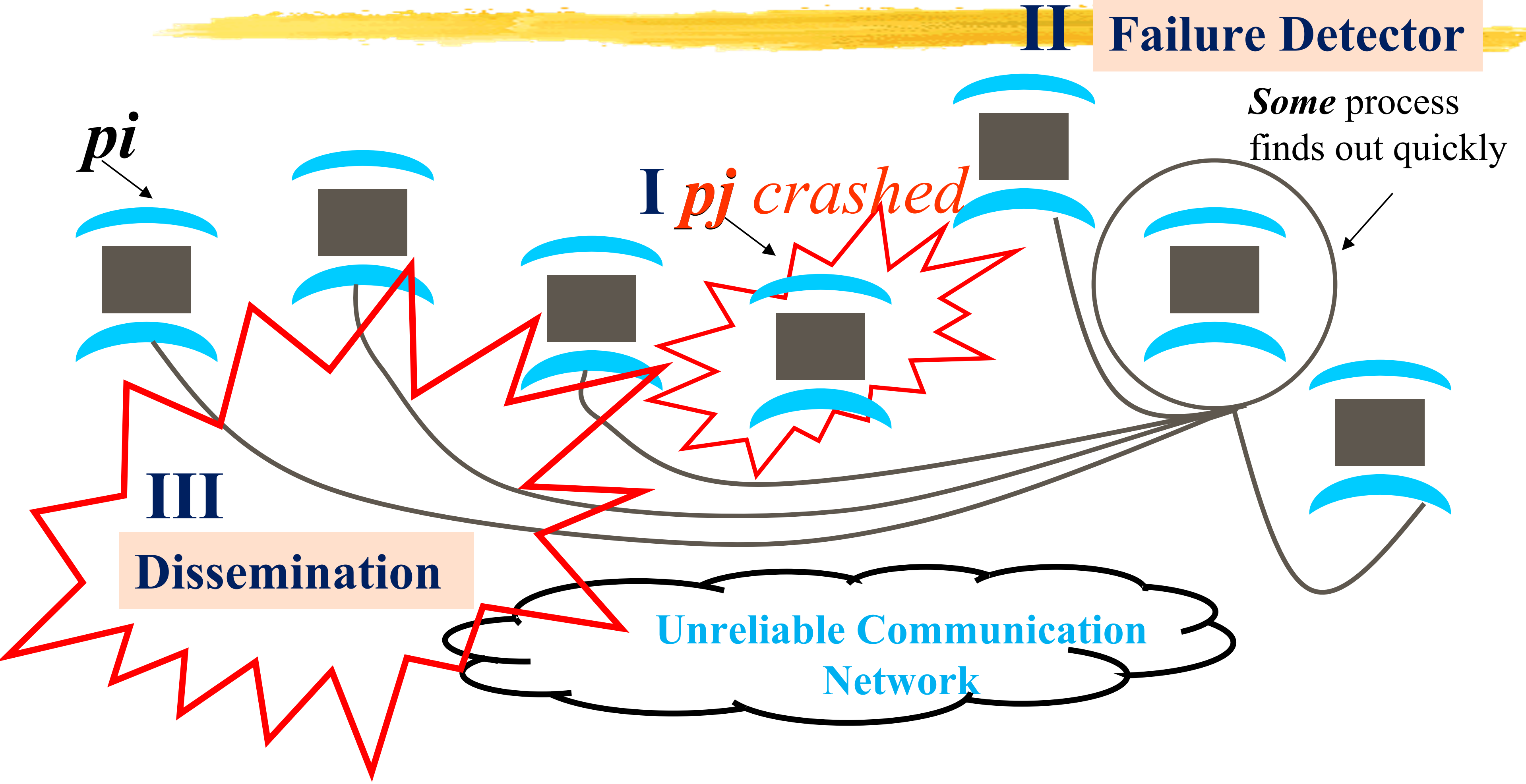


For Fixed :
- *False Positive Rate*
- *Message Loss Rate*

Next

- How do failure detectors fit into the big picture of a group membership protocol?
- What are the missing blocks?

Group Membership Protocol

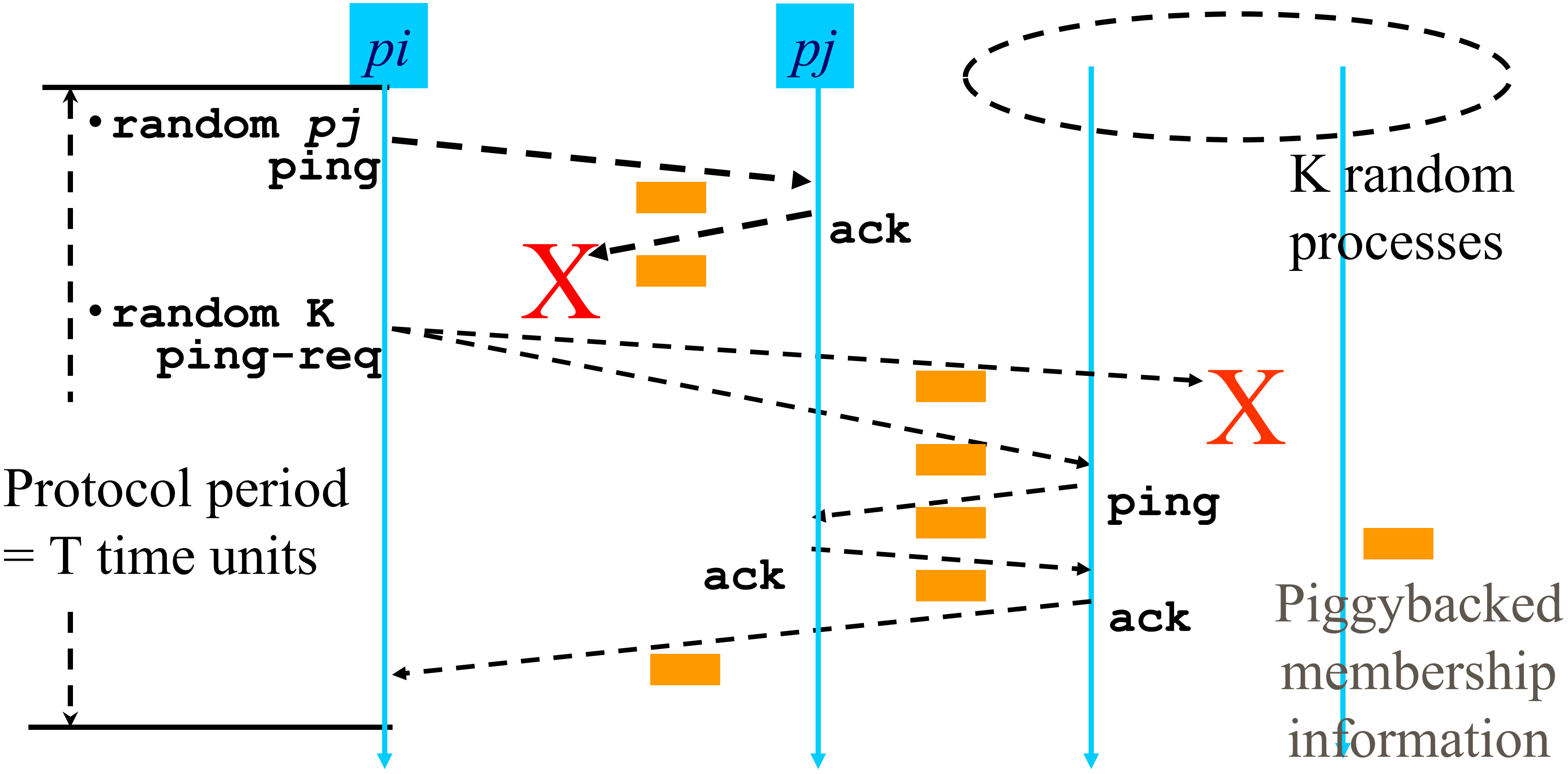

II Failure Detector
Some process finds out quickly
pi
I *pj crashed*
III Dissemination
Unreliable Communication Network

Fail-stop Failures only

Dissemination Options

- Multicast (Hardware / IP)
  - unreliable
  - multiple simultaneous multicasts
- Point-to-point (TCP / UDP)
  - expensive
- Zero extra messages: Piggyback on Failure Detector messages
  - Infection-style Dissemination

Infection-style Dissemination



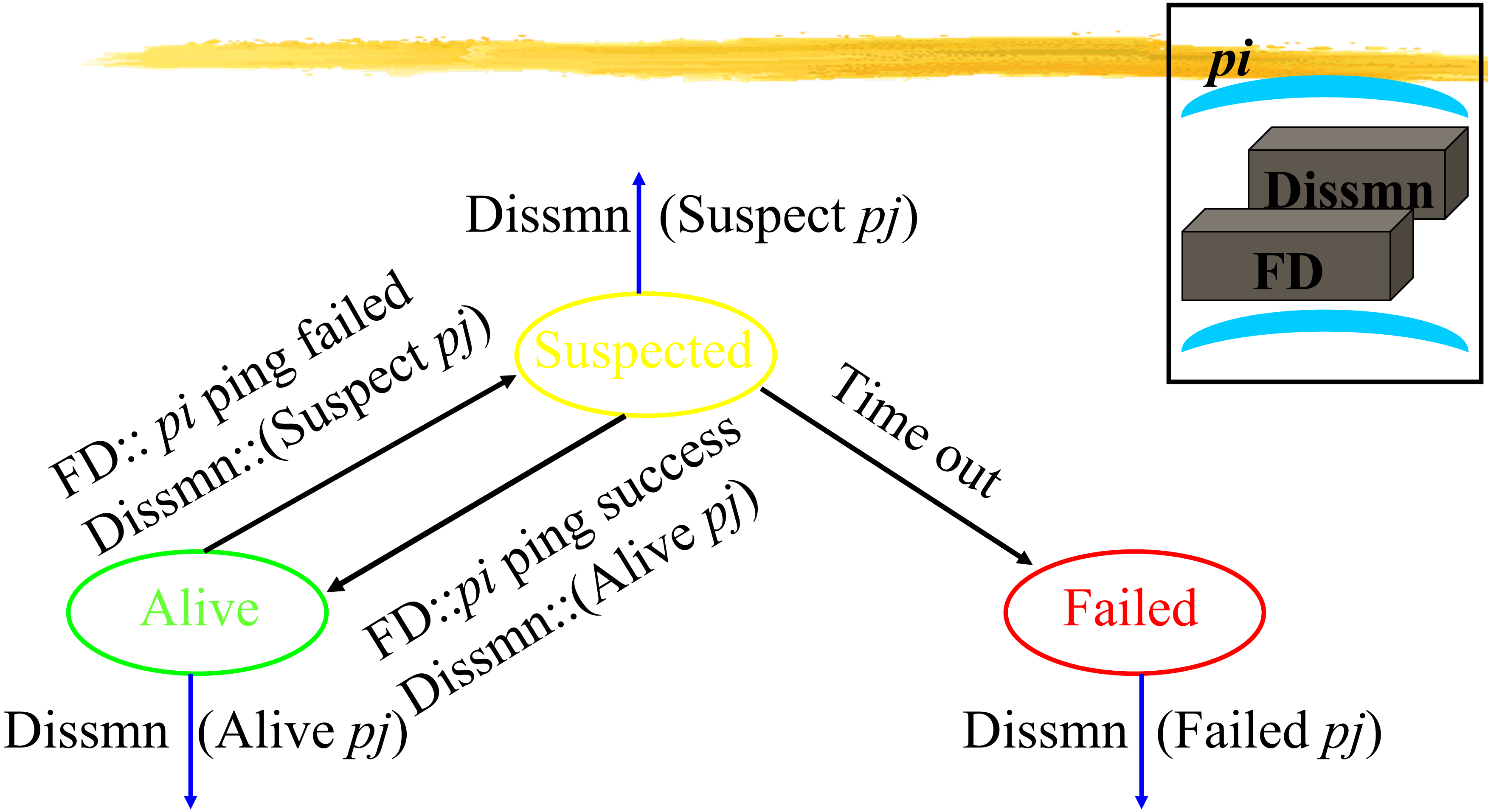
Infection-style Dissemination

- Epidemic/Gossip style dissemination
  - After $\lambda.\log(N)$ protocol periods, $N^{-(2\lambda-2)}$ processes would not have heard about an update
- Maintain a buffer of recently joined/evicted processes
  - Piggyback from this buffer
  - Prefer recent updates
- Buffer elements are garbage collected after a while
  - After $\lambda.\log(N)$ protocol periods, i.e., once they've propagated through the system; this defines weak consistency

Suspicion Mechanism

- False detections, due to
  - Perturbed processes
  - Packet losses, e.g., from congestion
- Indirect pinging may not solve the problem
- Key: *suspect* a process before *declaring* it as failed in the group

Suspicion Mechanism



pi

Dissmn

FD

Dissmn (Suspect *pj*)

Suspected

FD:: *pi* ping failed

Dissmn::(Suspect *pj*)

Time out

Alive

FD::*pi* ping success

Dissmn::(Alive *pj*)

Failed

Dissmn (Alive *pj*)

Dissmn (Failed *pj*)

Suspicion Mechanism

- Distinguish multiple suspicions of a process
  - Per-process *incarnation number*
  - *Inc* # for *pi* can be incremented only by *pi*
    - e.g., when it receives a (Suspect, *pi*) message
  - Somewhat similar to DSDV (routing protocol in ad-hoc nets)
- Higher inc# notifications over-ride lower inc#'s
- Within an inc#: (Suspect inc #) > (Alive, inc #)
- (Failed, inc #) overrides everything else

SWIM In Industry

- First used in Oasis/CoralCDN
- Implemented open-source by Hashicorp Inc.
    - Called “Serf”
    - Later “Consul”
- Today: Uber implemented it, uses it for failure detection in their infrastructure
    - See “ringpop” system

Wrap Up

- Failures the norm, not the exception in datacenters
- Every distributed system uses a failure detector
- Many distributed systems use a membership service

- Ring failure detection underlies
  - IBM SP2 and many other similar clusters/machines

- Gossip-style failure detection underlies
  - Amazon EC2/S3 (rumored!)

CAP Theorem (1)

- In 2000, **Eric Brewer** introduced the idea that there is a fundamental trade-off between
  - Consistency
  - Availability
  - Partition tolerance.
  -
- This trade-off, which has become known as the **CAP** Theorem

CAP Theorem (2)

- **C**onsistency
- **A**vailability
- **P**artition tolerance.

Theoretical context

- CAP Theorem
  - a general trade-off that appears everywhere in the study of **distributed computing**
  - the impossibility of guaranteeing both **safety** and **liveness** in an **unreliable** distributed system

Theoretical context

- **Safety**: nothing bad ever happens
  - Consistency is a classic safety property

- **Liveness:** eventually something good happens
  - Availability is a classic liveness property

- **Unreliable:**
  - Partitions
  - crash failures
  - message loss
  - malicious attacks (or Byzantine failures) etc

- **The CAP Theorem:** you cannot achieve both **safety** and **liveness** in an **unreliable** distributed system.

Practical implications

- It is necessary in practice to sacrifice either
  - Consistency
  - Availability

- There are systems that
  - Guarantee strong consistency and provide best effort availability
  - Guarantee availability and provide best effort consistency
  - May sacrifice both consistency and availability

The CAP Theorem

- Brewer’s original conjecture
- CAP Theorem in the context of a web service
- A set of distributed servers
- Clients make requests for the service
- Server receives a request and sends a response
- The CAP Theorem: a trade-off between Consistency, Availability and Partition tolerance

The CAP Theorem- Consistency

- **Consistency**:
  - each server returns the right response to each request
  - meaning of consistency depends on the service
- **Trivial services:**
  - no coordination
  - return the value of the constant PI=3.1416……
- **Weakly consistent services:** weaker consistency requirements that still provide useful services and yet avoid sacrificing availability
  - A distributed web cache

The CAP Theorem- Consistency

- **Simple services:**
  - **Sequential specification:** defines a service in terms of its execution on a single, centralized server
  - **Atomic:** for every operation, there is a single instant in between the request and the response at which the operation appears to occur
- **Complicated services:**
  - cannot be specified by sequential specifications
  - complicated coordination
  - transactional semantics

The CAP Theorem- Consistency

- Lets focus on a service that implements a read/write atomic shared memory:
  - The service provides its clients with a single (emulated) register
  - Each client can read or write from that register

The CAP Theorem-Availability

- **Availability:** second requirement of the CAP Theorem
  - means that each request eventually receive a response
  - a faster response is better
  - But, requiring an eventual response is sufficient to create problems

The CAP Theorem- Partition-tolerance

- **Partition-tolerance:** third requirement of the CAP theorem
  - Communication not reliable
  - Partitioned into multiple groups
  - Messages delayed
  - Messages lost forever

The CAP Theorem- Stated

‟ In a network subject to communication failures, it is impossible for any web service to implement an atomic read/write shared memory that guarantees a response to every request ”

The CAP Theorem- Proof sketch

- servers are partitioned into two disjoint sets: {p1} and {p_2, …, p_n}
- Some client sends a read request to server p_2
- p_1 is in a different component of the partition from p_2, every message from p_1 to p_2 is lost
- Thus, it is impossible for p_2 to distinguish the following two cases:
  - There has been a previous write of value v_1 requested of p_1 and p_1 has sent an ok response.
  - There has been a previous write of value v_2 requested of p_1 and p_1 has sent an ok response

The CAP Theorem- Proof sketch

- No matter how long p_2 waits, it cannot distinguish these two cases, and hence it cannot determine whether to return response v_1 or response v_2
- It has the choice to either
  - eventually return a response (and risk returning the wrong response)
  - or to never return a response.

The CAP Theorem- Theoretical Context

- Connection of Consistency and safety
  - A safety property is one that states nothing bad ever happens
  - Consistency requirements are almost always safety properties
  - every response is correct

- Connection of Availability and Liveness
  - Liveness property is one that states that eventually something good happens
  - Availability is a classic liveness property
  - eventually, every request receives a response

Concept: Consensus

Reaching Agreement is a fundamental problem in distributed computing

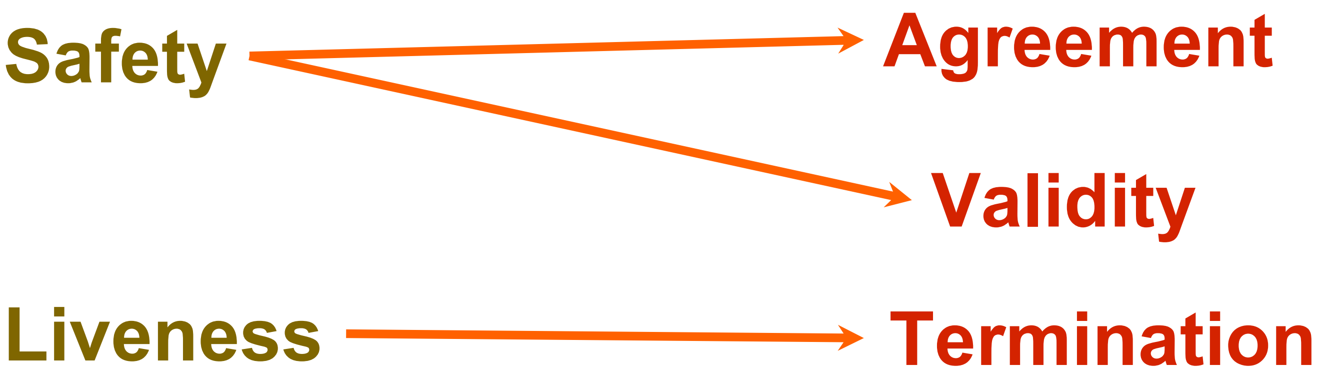
- Mutual Exclusion
  - processes agree on which process can enter the critical section
- Leader Election
  - processes agree on which is the elected process
- Totally Ordered Multicast
  - the processes agree on the order of message delivery
- Commit or Abort in distributed transactions
- Reaching agreement about which processes have failed
- Other examples
  - Air traffic control system: all aircrafts must have the same view
  - Spaceship engine control – action from multiple control processes("proceed" or "abort")
  - Two armies should decide consistently to attack or retreat.

Agreement is Impossible

- In 1985, Fischer, Lynch, and Paterson showed that:
  - fault-tolerant agreement is impossible in an asynchronous system
- They focused on the problem of consensus
  - each process p_i begins with an initial value v_i
  - all processes have to agree on one of those values

Consensus

- There are three requirements:
  - **agreement**: every process must output the same value
  - **validity**: every value output must have been provided as the input for some process
  - **termination**: eventually, every process must output a value.



Safety

Agreement

Validity

Liveness

Termination

Consensus

- Heart of the replicated state machine approach
- To improve availability, a service may be replicated at a set of servers
- to maintain consistency, the servers agree on every update to the service
- The safety requirements of consensus are strictly harder than simply implementing an atomic read/write register
- CAP Theorem also implies that you cannot achieve consensus in a system subject to partitions

Variant of Consensus Problem

- Consensus Problem (C)
  - Each process proposes a value
  - All processes agree on a single value

- Byzantine Generals Problem (BG)
  - Process fails arbitrarily, byzantine failure
  - Still processes need to agree

- Interactive Consistency (IC)
  - Each process propose its value
  - All processes agree on the vector

Solving Consensus

- No failures – trivial
  - All-to-all broadcast followed by applying a choice function
- With failures
  - One assumption: Processes fail only by *crash-stop*ping

- Synchronous system: Possible?
- Asynchronous system: ???

What about other failures??

- Omission Failures
- Byzantine Failures

Consensus

- Fischer et al. considered a system
  - no partitions
  - one (unknown) process in the system may fail by crashing
  - communicate reliably
- They Concluded
  - consensus is impossible

Coping with the Safety/Liveness Trade-off for Consensus

- After the publication of Fischer's work
  - researchers in distributed computing began examining this inherent trade-off between safety and liveness in more depth

- Knowing safety and liveness are impossible in systems that are sufficiently unreliable, gave birth to two questions

Question no 1

- Under what conditions it is possible to achieve both safety and livness?
    - what level of synchrony is necessary to avoid the inherent trade-off?
    - How many failures can be tolerated?
    - what level of network reliability is needed to achieve both consistency and availability?

Question no 2

- focuses on the question of consistency
- given that the network is unreliable, what is the maximum level of consistency that can be achieved?

Under what conditions it is possible to achieve both safety and livness?

Answer to Question 1

- network synchrony
- A network is synchronous if it satisfies the following properties-
  - every process has a clock, and all the clocks are synchronized
  - every message is delivered within a fixed and known amount of time
  - every process takes steps at a fixed and known rate

Under what conditions it is possible to achieve both safety and livness?

Answer to Question 1

- network synchrony
- systems as progressing in rounds
- within each round, each process:
  - sends some messages
  - receives all the messages that were sent to it in that round
  - performs some local computation

Answer to Question 1 Time complexity

Under what conditions it is possible to achieve both safety and livness?

- Synchronous system
  - consensus can be solved
  - consensus requires f + 1rounds, if up to f servers may crash
- asynchronous system
  - consensus is impossible
  - How much synchrony is needed to solve consensus?
  - Do real systems provide that necessary level of synchrony?
  - Dwork et al. attempted to answer this question

Answer to Question 1 Time complexity

Under what conditions it is possible to achieve both safety and livness?

- Dwork et al. introduced the idea of
- Eventual synchrony
  - a system may experience some periods of synchrony
  - some periods of asynchrony
  - eventually stabilizes and maintains synchrony for a sufficiently long period of time
- If a system is Eventually synchronous, we can solve consensus

Answer to Question 1 Time complexity

Under what conditions it is possible to achieve both safety and livness?

- How long a "window of synchrony" is necessary to solve consensus?
- Dutta and Guerraoui showed that at least f + 2 rounds are necessary
- Alistarh et al. recently showed that f + 2 rounds of synchrony are also sufficient.

Answer to Question 1 Time complexity

Under what conditions it is possible to achieve both safety and livness?

- connection between the synchrony of a system and the crash-tolerance
- synchronous system:
  - can solve consensus for any number of failures
- asynchronous system:
  - consensus is impossible for even one failure
- eventually synchronous system:
  - can solve consensus if there are < n/2 crash failures
  - n is the number of servers

Answer to Question 1 Failure detectors

Under what conditions it is possible to achieve both safety and livness?

- Different approach
- Chandra et al. introduced the idea of a failure detector
- An oracle that provides sufficient information for processes to solve consensus in an asynchronous crash-prone system
- particular failure detector Ω is the weakest failure detector for solving consensus.
- The failure detector essentially encapsulates a leader election service

Answer to Question 2

- What is the strongest form of consistency we can guarantee in a system with f crash failures?
- Chaudhuri introduced the problem of set agreement
  - each process begins with some value
  - eventually chooses an output
  - the validity and termination conditions are identical to concensus
  - some disagreement in the output is allowed.
  - k-set agreement: there may be up to k-different output values

What is the strongest form of consistency we can guarantee in a system with f crash failures?

Answer to Question 2

- 1-set agreement: consensus
- n-set agreement: trivial (i.e., each process simply outputs its own initial value)
- 1-set agreement is impossible if there is even one crash failure
- n-set agreement can tolerate an arbitrary number of crash failures

What is the strongest form of consistency we can guarantee in a system with f crash failures?

Answer to Question 2

- Borowski, Gafni, Herlihy, Saks, Shavit, and Zaharoglou showed that k-set agreement can be solved if and only if there are at most k -1crash failures
- Chaudhuri et al showed: in a synchronous system with t failures, at least |t/k|+1rounds are necessary and sufficient for k-set agreement

Practical Implications

- implication of the CAP Theorem:
  - we cannot achieve consistency and availability in a partition-prone network.
- CAP theorem indicates
  - the difficulties in providing distributed services.
- Yet, distributed services exist !!!
- practitioners building distributed services
  - successfully build and deploy distributed systems
  - overcome challenges posed by the CAP Theorem in various ways
    - Best Effort Availability
    - Best Effort Consistency
    - Trading Consistency for Availability
    - Segmenting Consistency and Availability

Best Effort Availability

- a common approach to dealing with unreliable networks
  - design a service that guarantees consistency
    - provide correct operation
  - optimize the service to provide best effort availability,
    - be as responsive as is possible given the current network conditions.
- more suitable to networks with
  - reliable and timely communication
- a good approach
  - when all the servers running a service are located in the same data center

Best Effort Availability (contd.)

- A recent popular approach: ***Chubby Lock Service***
  - built by Google and used extensively in the Google infrastructure
  - supports the Google File System, BigTable, etc
- Chubby provides strong consistency
  - has a distributed database, based on a primary-backup design.
  - consistency among the servers is ensured by using a replicated state machine protocol to maintain synchronized logs.
  - continues to operate as long as no more than half the servers fail

Best Effort Availability (contd.)

- Chubby guaranteed to make progress
  - whenever the network is reliable.
- Chubby is optimized for the case where
  - there is a stable primary and there are no partitions.
    - delivers a very high degree of availability.
- Chubby “cell” is deployed in a single data center
- Chubby cell
  - communication fast and reliable
  - failure of a primary is not too frequent.

Best Effort Consistency

- for some application sacrificing availability is not acceptable
- when the application is deployed over a wide area
  - the level of availability that can be achieved by a strongly consistent service may degrade rapidly.
- in such case, sacrifice consistency:
  - a response (preferably fast) is guaranteed at all times.
- the response may not always be correct.
  - consistency is provided only in a best effort sense.
- The classic example: Web Caching

Best Effort Consistency (contd.)

- web content are cached on servers
  - placed in data centers throughout the world.
- whenever user requests a given web page,
  - the content can be delivered from a nearby web cache.
- guarantees a very high level of availability
  - the responses are rapid
  - network connectivity issues rarely prevent a response.
- the consistency guarantees are (potentially) quite minimal.
  - after a web page update, propagating new content to all the cache servers needs some time
  - no guarantee that all users accessing a web page at any given time receive the exact same content.
  - content viewed by a user is slightly out-of-date, induces little harm.
  - less time for loading a web page is desired

Trading Consistency for Availability

- tune the trade-off between consistency and availability.
  - it may be acceptable for some content to be one hour out of date, but not one day out of date.
  - with good network connectivity for some time period
    - a service should be able to provide this level of consistency with good availability
  - for long-lasting partitions (e.g., more than one day)
    - availability is impossible
- specify the CAP trade-off
  - set the threshold for how out of date the data can be

Trading Consistency for Availability (contd.)

- TACT toolkit
  - enables replicated applications to specify exactly the desired consistency
- most of the seats on the airplane are available,
  - safe for the reservation system to rely on somewhat out-of-date data
  - with a few seats booked, new reservation can likely be accommodated.
  - As the plane is filled the reservation system requires increasingly accurate data to ensure that the plane is not overbooked.
- TACT enables the reservation system to
  - request increasing levels of consistency as the number of available seats diminishes.
- such a system provides neither strong consistency nor guaranteed availability
  - data can be out of date
  - yet with network partition, the service may still be unavailable.

Segmenting Consistency and Availability

- Many systems do not have a single uniform requirement.
  - Some aspects of the system require strong consistency, and some require high availability.
- segment the system into components that provide
  - different types of guarantees.
- a service that, as a whole, guarantees neither consistency nor availability.
  - each part of the service provides exactly what is needed.
- some of the dimensions along which a system might be partitioned.
  - Data partitioning
  - Operation partitioning
  - Functional partitioning
  - User partitioning
  - Hierarchical partitioning

Segmenting Consistency and Availability (contd.)

- Different types of data may require different levels of consistency and availability.
  - an on-line shopping cart may be
    - highly available, responding rapidly to user requests;
    - but occasionally inconsistent, losing a recent update in anomalous circumstances.
  - on-line product information for an e-commerce site may be
    - somewhat inconsistent: users will tolerate somewhat out-of-date inventory information.
    - The check-out/billing/shipping records, however, have to be strongly consistent: a user will be very unhappy if a finalized order does not reflect her intended purchase

Segmenting Consistency and Availability (contd.)

- Different operations may require different levels of consistency and availability.
    - consider a system that guarantees
        - high availability for read-only operations,
            - a “query” operation might return out of date data.
        - modify database operations may not respond during network partitions
            - a “purchase” operation should guarantee consistency,
    - PNUTS system, Yahoo!

Segmenting Consistency and Availability (contd.)

- Many services can be divided into different subservices which have different requirements.
  - an application might use a service such as Chubby for
    - distributed coordination (i.e., strong consistency)
  - at the same time it uses a service such as DNS to handle naming:
    - relatively weak consistency but high availability.
  - The same service might use yet a third subservice
    - with a different consistency/availability trade-off for content distribution.

Segmenting Consistency and Availability (contd.)

- Network partitions, and poor network performance correlate with real geo-graphic distance:
  - users that are far away are more likely to see poor performance.
- Some applications are organized hierarchically
  - At the top level, an application encompasses the entire world or the entire database;
  - subsequent levels of the hierarchy partition the world into geographically smaller pieces, or the database into smaller parts.
  - At each level of the hierarchy, the system may provide a different level of performance:
    - better availability toward the leaves, or less consistency toward the root.
  - For example, as you descend a geographically-organized hierarchy, the limitations of the CAP Theorem becomes less and less onerous as the relevant servers become better and better connected.

The CAP Theorem in Future Systems

- Scalability
  - scalable systems grow while using new resources efficiently to handle more load.
  - inherent trade-offs between scalability and consistency.
    - in order to efficiently use new resources, there must be coordination among those resources;
    - the consistency required for this coordination appears subject to the CAP Theorem trade-offs.

The CAP Theorem in Future Systems (contd.)

- Tolerating Attacks
  - The CAP Theorem focuses on **network partitions**
  - A denial-of-service attack cannot be modeled as a network partition
  - malicious users hack servers, disrupt major internet services
- Tolerating these more problematic forms of disruption requires
  - a somewhat different understanding of the fundamental consistency/availability trade-offs.

The CAP Theorem in Future Systems (contd.)

- Mobile Wireless Networks
  - The CAP Theorem initially focused on wide-area internet services
  - wireless communication is notoriously unreliable
    - partitions are less common
    - unpredictable message loss is very common
    - message latencies can vary significantly
  - the types of applications being deployed in wireless networks may be somewhat different

Conclusion

- the CAP Theorem
  - a fundamental trade-off between safety and liveness in fault-prone systems.
  - helps device systems that can be designed to meet an application’s needs
- despite unreliable networks, software architects have explored
  - strongly consistent solutions, with best-effort availability;
  - weakly consistent solutions with high availability
  - systems that mix both weaker availability and weaker consistency in varying ways.
- highlight on future research directions

Dolev-Strong Byzantine Agreement Problem

Assumptions:

-
  - N processors
  - One processor s is *the sender* with the initial value p_s
  - Each non faulty processor i need to decide on a value p_i^*
  - Up to f processors can be faulty *(Byzantine failure)*
  - Reliable communication network
  - Synchronization systems: time is discrete into *rounds, e*ach longer than the longest delay due to message transmission and processing
  - Authenticated setting

Requirements:

- *Agreement*: If i, j are non-faulty then $p_i^* = p_j^*$
- *Validity*: If the sender is honest then $p_i^* = p_s$ for all non-faulty players i

Dolev-Strong Authenticated Broadcast Protocol

f+1 rounds

- Round 1:
  - The leader 1 signs and sends v to all other processors: $< v, sign(v, 1) >$
- For $\mathrm{r} = 2$ to $f + 1$:
  - If a processor i receives a message m which is a valid r –signature chain on an **unseen** value v then send $< m, sign(m, i) >$ to all. Set $S \leftarrow S \cup \{v\}$
  - Otherwise, remain silent
- If $|S| = 1$ then output $v \in S$, otherwise output 'confused' or 'default' message

Dolev Strong Protocol Analysis

- A signature chain consists of signature from distinct nodes.
- A round i signature chain will be of length i
- If there is a signature chain of length $f + 1$:
  - It will certainly contain an honest signature from honest node h.
  - Party h can send the value to all other nodes.
- A message received by node i at the end of round r is valid if:
  - The first signature of the chain is from the leader
  - All signatures in the chain are from distinct nodes
  - Node i is not in the chain
  - All signatures are valid
  - The signature chain is of length r

Dolev Strong Protocol Correctness

- **Termination**:
  - Deterministic and terminate in $f + 1$ rounds
- **Validity:**
  - The honest leader signs only one value and send to other honest nodes in the next round
- **Agreement:**
  - If an honest party has $v \in S$ at round $r \leq f$ then all other nodes will have $v \in S$ at $r + 1$
  - If an honest node receives v at round $f + 1$ in a signature chain of size $f + 1$:
    - The $f + 1$ signature chain must contain an honest node h
    - The honest node h must already broadcast value v to other honest nodes in an earlier round
  - All honest nodes have identical S

Consensus Synchronous vs. Asynchonous Models

Synchronous Distributed System

- Drift of each process' local clock has a known bound
- Each step in a process takes lb < time < ub
- Each message is received within bounded time

Consensus is possible in the presence of failures!!

- Asynchronous Distributed System
  - No bounds on process execution
  - The drift rate of a clock is arbitrary
  - No bounds on message transmission delays

Consensus is impossible with the possibility of even 1 failure!!

Defining Consensus

N processes

- Every process contributes a value
- Goal: To have all processes decide on the same (some) value
  - Once made, the decision cannot be changed.

Each process p has

- input variable xp : initially either 0 or 1
- output variable yp : initially b (b=undecided) – can be changed only once

Consensus problem: design a protocol so that either

1. all non-faulty processes set their output variables to 0
2. Or non-faulty all processes set their output variables to 1
3. There is at least one initial state that leads to each outcomes 1 and 2 above

Consensus Properties/Terms

- **Termination**
  - Every non-faulty process must eventually decide.
- **Integrity**
  - The decided value must have been proposed by some process
- **Validity**
  - If every non-faulty process proposes the same value v, then their final decision must be v.

- **Agreement**
  - The final decision of every non-faulty process must be identical.
- **Non-triviality**
  - There is at least one initial system state that leads to each of the all-0’s or all-1’s outcomes

Consensus in a Synchronous System

- Possible
  - With one or more faulty processes

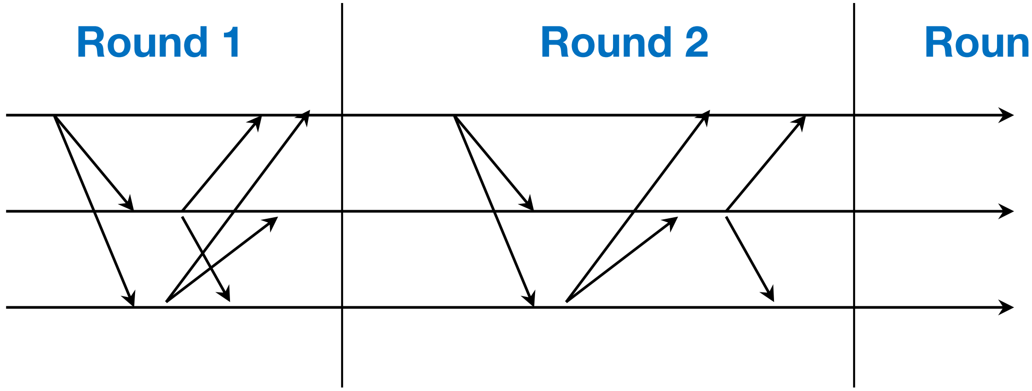
- Solution - Basic Idea:
  - all processes exchange (multicast) what other processes tell them in <u>several rounds</u>

- To reach consensus with *f* failures, the algorithm needs to run in *f* + 1 rounds.

Consensus in Synchronous Systems

For a system with at most *f* processes crashing

- All processes are synchronized and operate in "rounds" of time. Round length >> max transmission delay.
- The algorithm proceeds in *f+1* rounds (with timeout), using reliable communication to all members



Consensus with at most f failures : Synchronous Systems

$Value_i^r$: the set of proposed values known to p_i at the beginning of round r.

```
Initially $Value_i^0 = \{\}$ ; $Value_i^1 = \{v_i\}$
    for round = 1 to $f+1$ do
        multicast ($Value_i^r - Value_i^{r-1}$) # iterate through processes, send each a message
        $Value_i^{r+1} \leftarrow Value_i^r$
        for each $V_j$ received
            $Value_i^{r+1} = Value_i^{r+1} \cup V_j$
        end
    end
    $d_i$ = minimum ($Value_i^{f+1}$) # consistent minimum based on say, id (not minimum value)
```

Proof: Consensus in Synchronous Systems (extra)

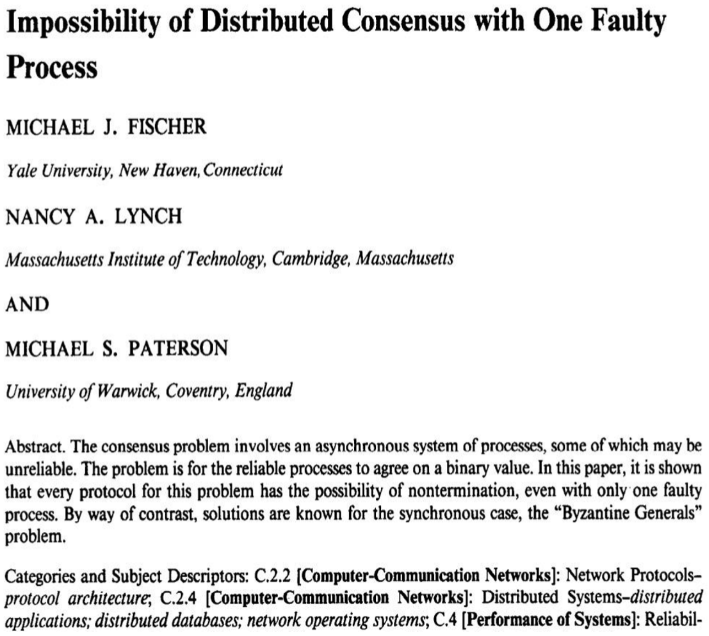
After f+1 rounds, all non-faulty processes would have received the same set of Values.

Proof by contradiction.

- Assume that two non-faulty processes, say p_i and p_j , differ in their final set of values (i.e., after f+1 rounds)
- Assume that p_i possesses a value v that p_j does not possess.
  - p_i must have received v in the very last round; else, p_i would have sent v to p_j in that last round
  - So, in the last round: a third process, p_k, must have sent v to p_i, but then crashed before sending v to p_j.
  - Similarly, a fourth process sending v in the last-but-one round must have crashed; otherwise, both p_k and p_j should have received v.
  - Proceeding in this way, we infer at least one (unique) crash in each of the preceding rounds.
  - This means a total of f+1 crashes, while we have assumed at most f crashes can occur => contradiction.

Asynchronous Consensus

- Messages have arbitrary delay, processes arbitrarily slow
- Impossible to achieve!
  - a slow process indistinguishable from a crashed process
- Result due to Fischer, Lynch, Patterson (commonly known as FLP 85).



Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

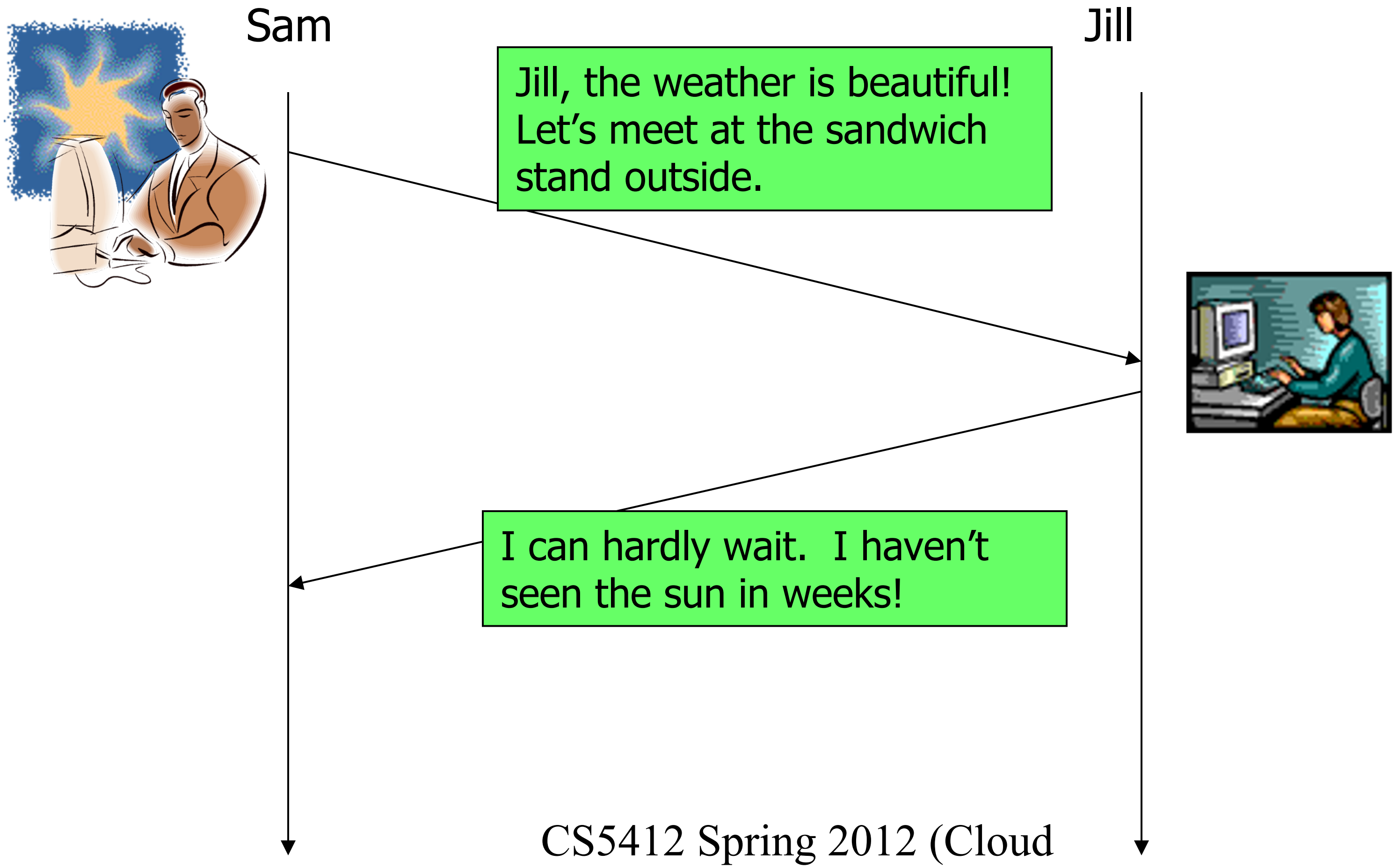
Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols–*protocol architecture*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems–*distributed applications; distributed databases; network operating systems*; C.4 [**Performance of Systems**]: Reliabil-

Theorem: In a purely asynchronous distributed system, the consensus problem is impossible to solve if even a single process crashes.

Intuition Behind FLP Impossibility Theorem

- Jill and Sam will meet for lunch. They'll eat in the cafeteria unless both are sure that the weather is good
  - Jill's cubicle is inside, so Sam will send email
  - Both have lots of meetings, and might not read email. So she'll acknowledge his message.
  - They'll meet inside if one or the other is away from their desk and misses the email.
- Sam sees sun. Sends email. Jill acks's. Can they meet outside?

Sam and Jill



Sam

Jill

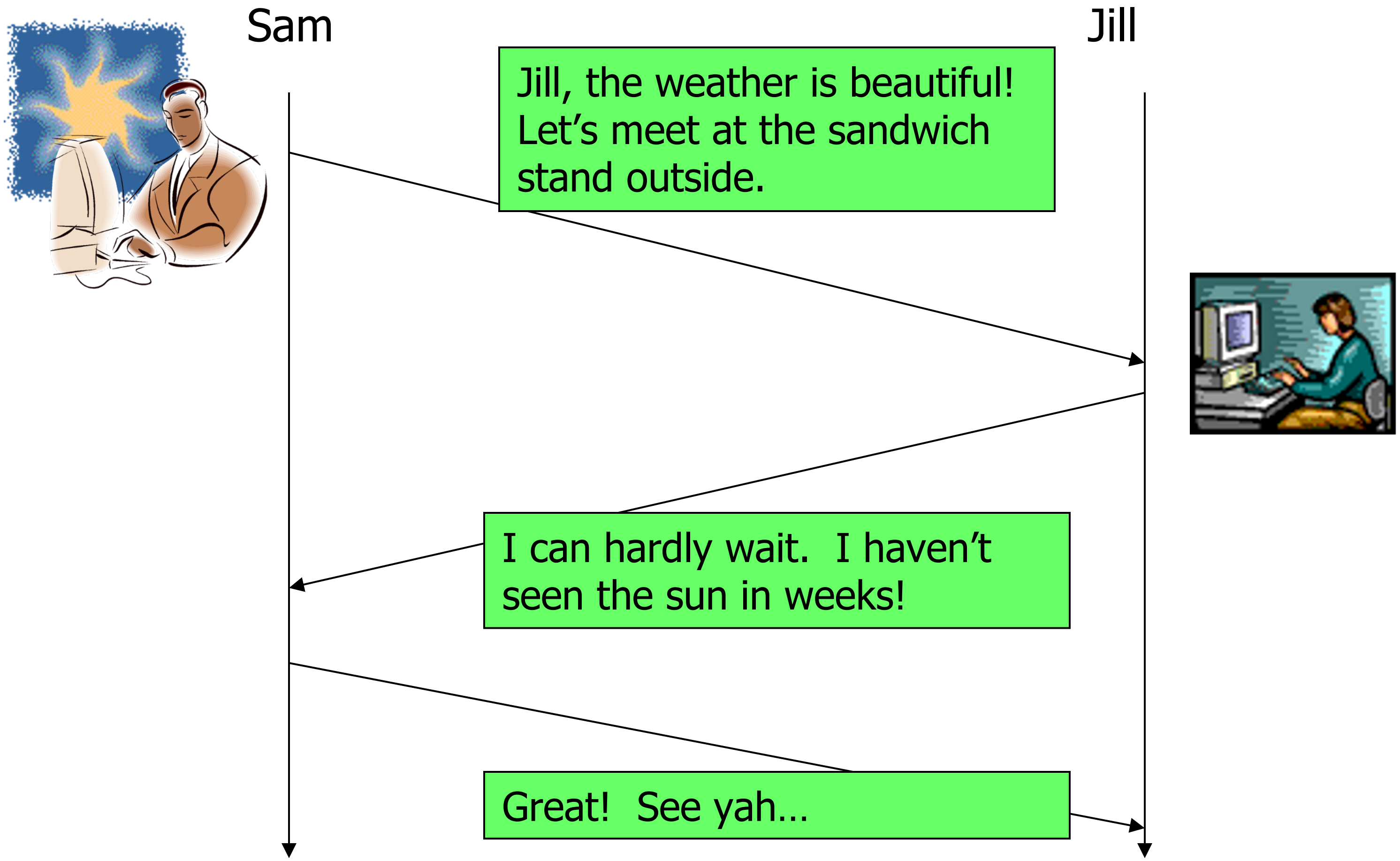
Jill, the weather is beautiful! Let’s meet at the sandwich stand outside.

I can hardly wait. I haven’t seen the sun in weeks!

They eat inside! Sam reasons:

- "Jill sent an acknowledgement but doesn't know if I read it
- "If I didn't get her acknowledgement I'll assume she didn't get my email
- "In that case I'll go to the cafeteria
- "She's uncertain, so she'll meet me there

Sam had better send an Ack



Sam

Jill

Jill, the weather is beautiful! Let’s meet at the sandwich stand outside.

I can hardly wait. I haven’t seen the sun in weeks!

Great! See yah…

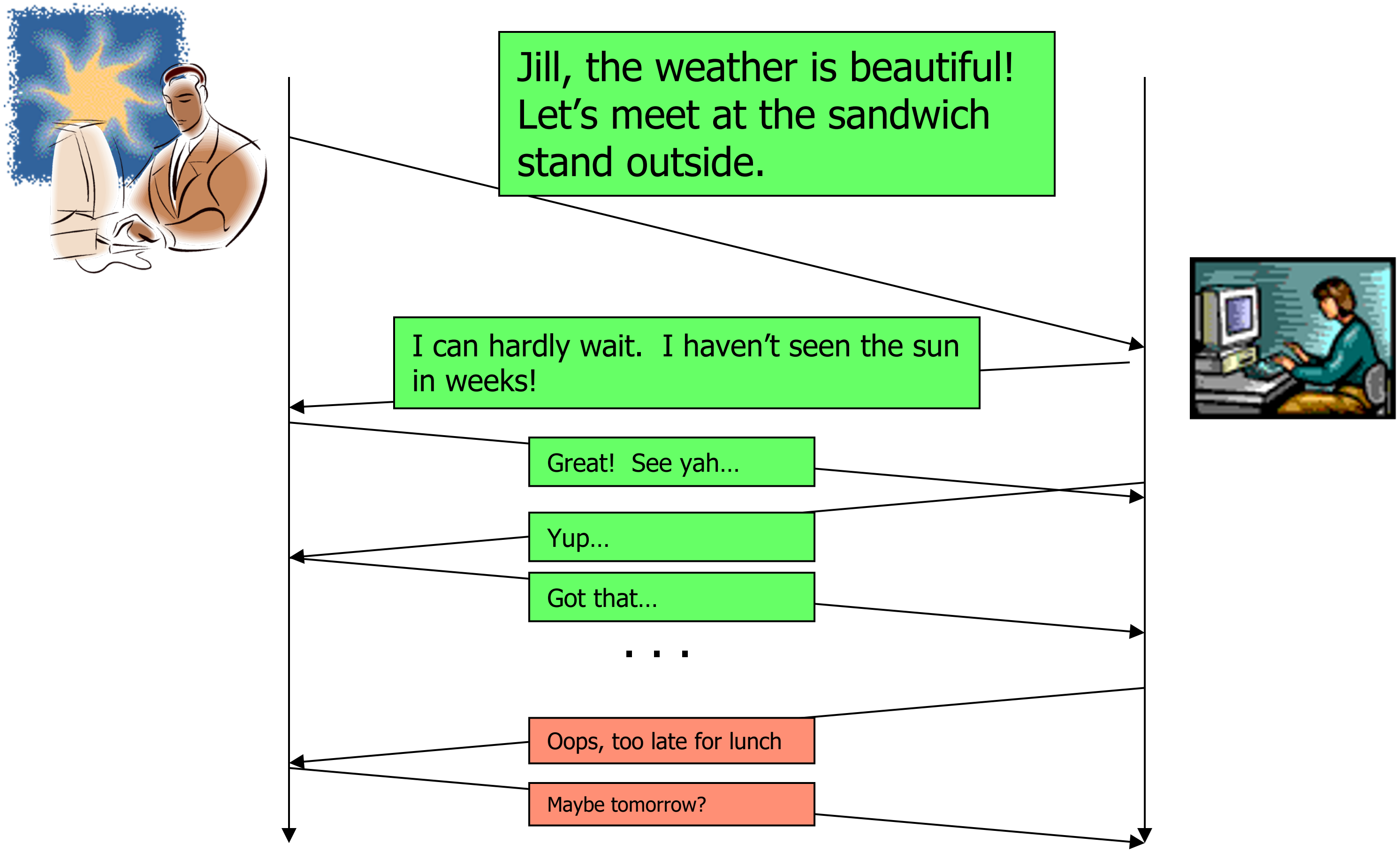
Why didn’t this help?

- Jill got the ack… but she realizes that Sam won’t be sure she got it
- Being unsure, he’s in the same state as before
- So he’ll go to the cafeteria, being dull and logical. And so she meets him there.

New and improved protocol

- Jill sends an ack. Sam acks the ack. Jill acks the ack of the ack….
- Suppose that noon arrives and Jill has sent her 117’th ack.
  - Should she assume that lunch is outside in the sun, or inside in the cafeteria?

How Sam and Jill's romance ended



Jill, the weather is beautiful! Let's meet at the sandwich stand outside.

I can hardly wait. I haven't seen the sun in weeks!

Great! See yah…

Yup…

Got that…

. . .

Oops, too late for lunch

Maybe tomorrow?

Things we just can’t do

- We can’t detect failures in a trustworthy, consistent manner
- We can’t reach a state of “common knowledge” concerning something not agreed upon in the first place
- We can’t guarantee agreement on things (election of a leader, update to a replicated variable) in a way certain to tolerate failures

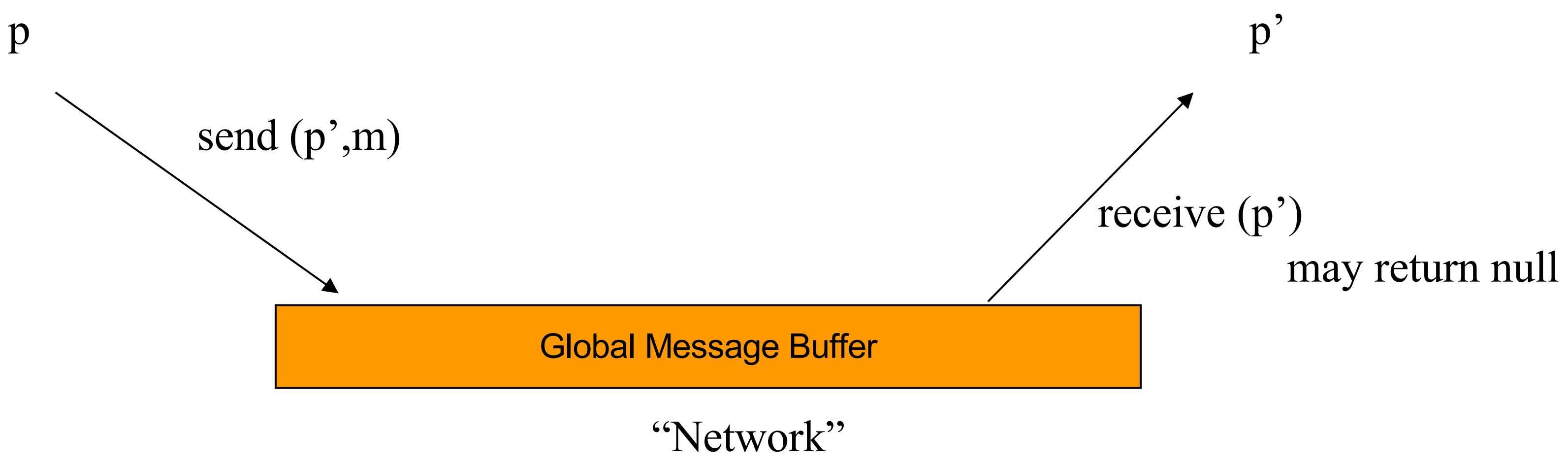
But what does it mean?

- In formal proofs, an algorithm is totally correct if
  - It computes the right thing
  - And it always terminates
- When we say something is possible, we mean "there is a totally correct algorithm" solving the problem
- FLP proves that any fault-tolerant algorithm solving consensus has runs that never terminate
  - These runs are extremely unlikely ("probability zero")
  - Yet they imply that we can't find a totally correct solution
  - And so "consensus is impossible" ("not always possible")
- In practice, fault-tolerant consensus is ..
  - Definitely possible.
  - E.g. **Paxos [Lamport 1998, 2001] that has become quite popular – discussed later!**

Proof Setup

- For impossibility proof, OK to consider:
    - more restrictive system model, and
    - easier problem
- Why?

Network



p

p’

send (p’,m)

receive (p’)

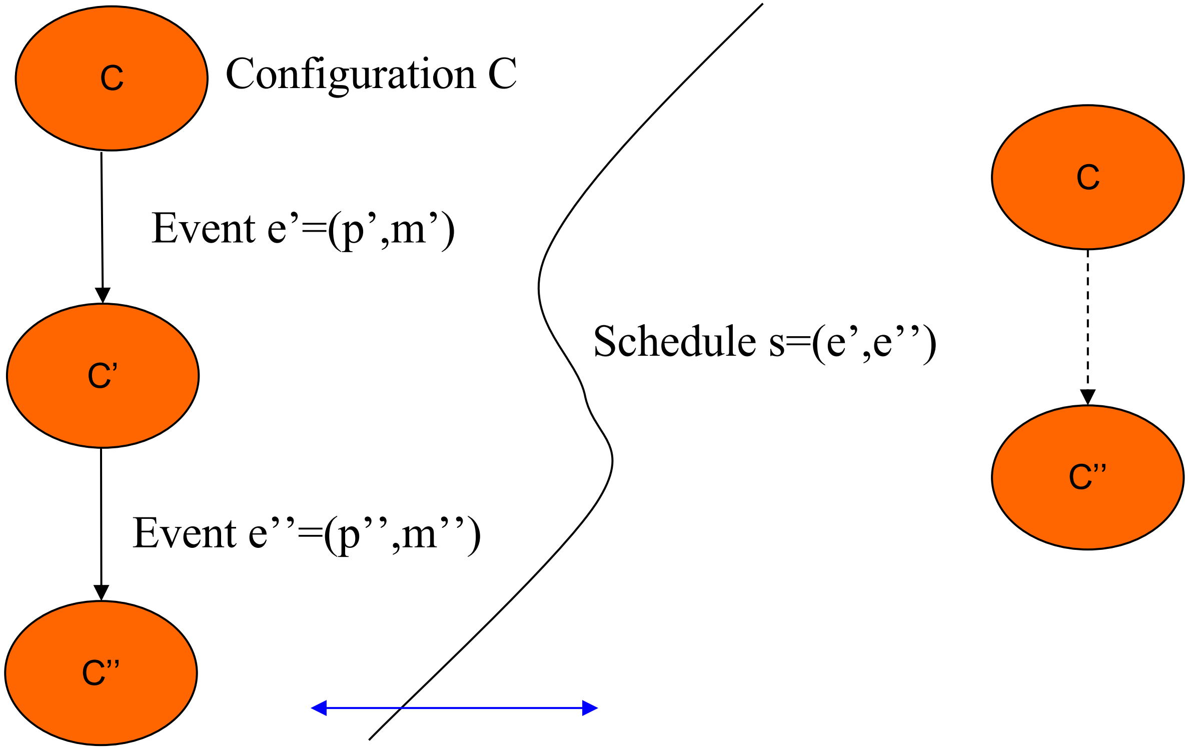
may return null

Global Message Buffer

“Network”

State

- State of a process
- Configuration = global state. Collection of states, one for each process; alongside state of the global buffer.
- Each Event (different from Lamport events) is atomic and consists of three steps
  - receipt of a message by a process (say p)
  - processing of message (may change recipient's state)
  - sending out of all necessary messages by p
- Schedule: sequence of events



Configuration C

Event e’=(p’,m’)

Event e’’=(p’’,m’’)

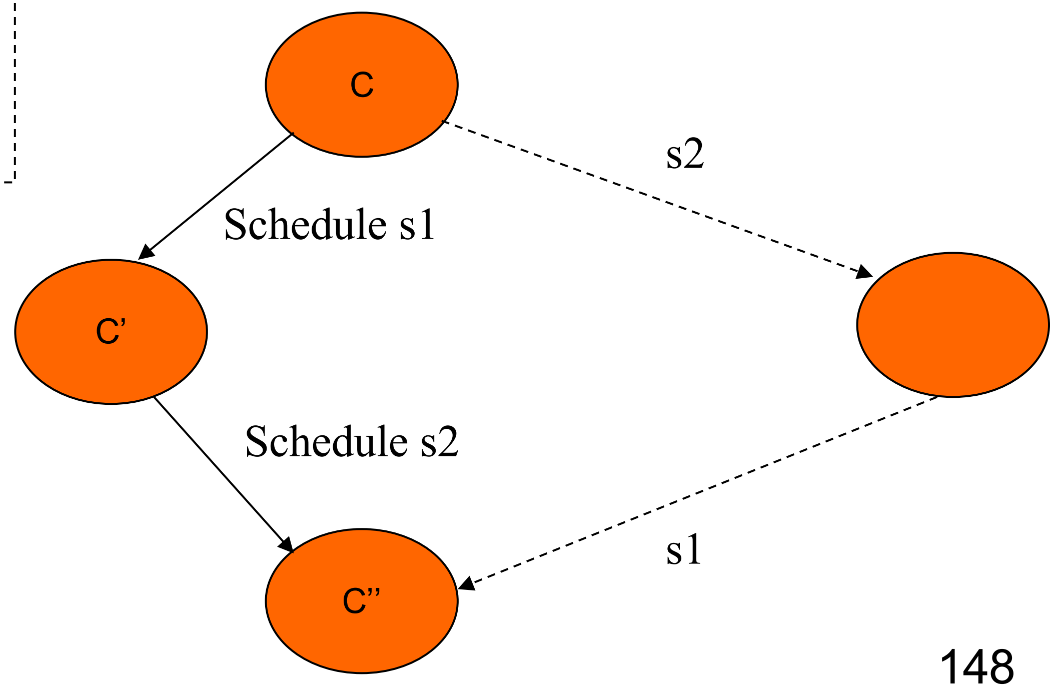
Schedule s=(e’,e’’)

Equivalent

Lemma 1

s1 and s2 involve disjoint sets of receiving processes, and are each applicable on C

Disjoint schedules are commutative



Easier Consensus Problem

- Easier Consensus Problem: some process eventually sets yp to be 0 or 1
- Only one process crashes – we're free to choose which one

Easier Consensus Problem

- Let configuration C have a set of decision values V reachable from it
    - If |V| = 2, configuration C is bivalent
    - If |V| = 1, configuration C is 0-valent or 1-valent, as is the case
- Bivalent means outcome is unpredictable

What the FLP proof shows?

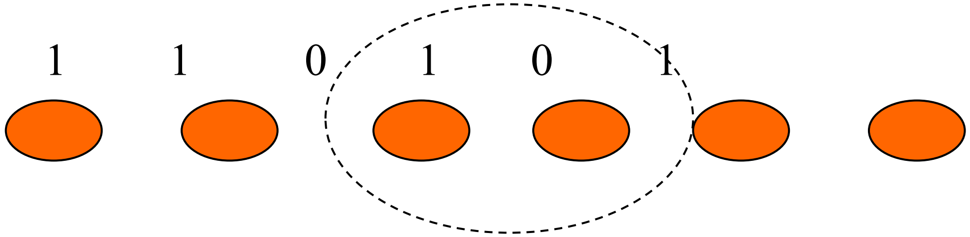
1. There exists an initial configuration that is bivalent

1. Starting from a bivalent configuration, there is always another bivalent configuration that is reachable

Lemma 2

Some initial configuration is bivalent

- Suppose all initial configurations were either 0-valent or 1-valent.
- If there are N processes, there are 2^N possible initial configurations
- Place all configurations side-by-side (in a lattice), where adjacent configurations differ in initial xp value for exactly one process.

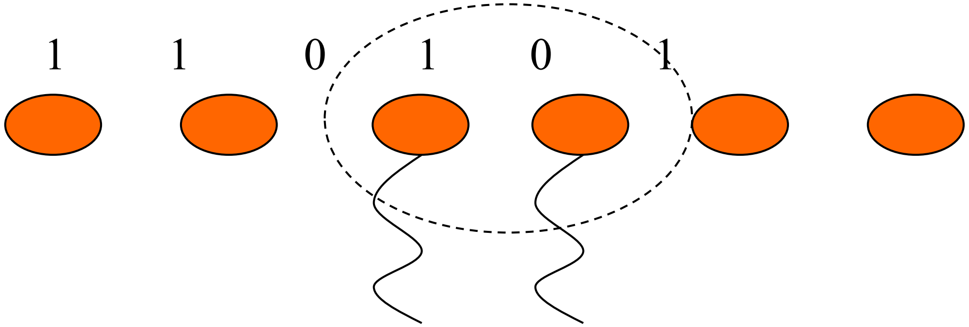


- There has to be some adjacent pair of 1-valent and 0-valent configurations

Lemma 2

Some initial configuration is bivalent

- There has to be some adjacent pair of 1-valent and 0-valent configs.
- Let the process p, that has a different state across these two configs., be the process that has crashed (i.e., is silent throughout)



- Both initial configs. will lead to the same config. for the same sequence of events
- Therefore, both these initial configs. are bivalent when there is such a failure

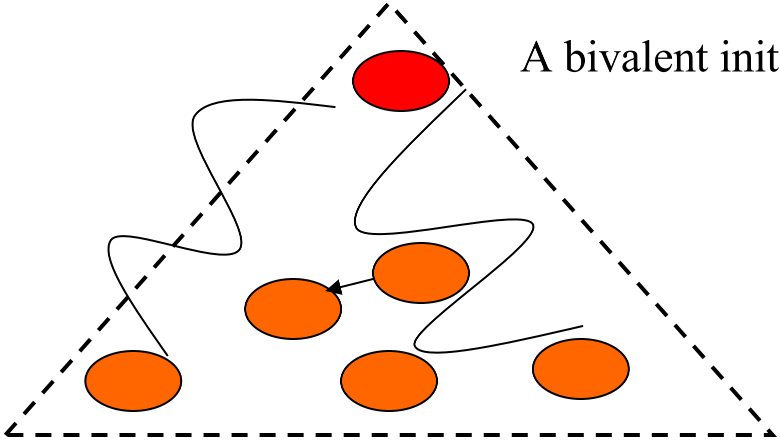
What we’ll show

1. There exists an initial configuration that is bivalent

1. Starting from a bivalent configuration, there is always another bivalent configuration that is reachable

Lemma 3

Starting from a bivalent configuration, there is always another bivalent configuration that is reachable



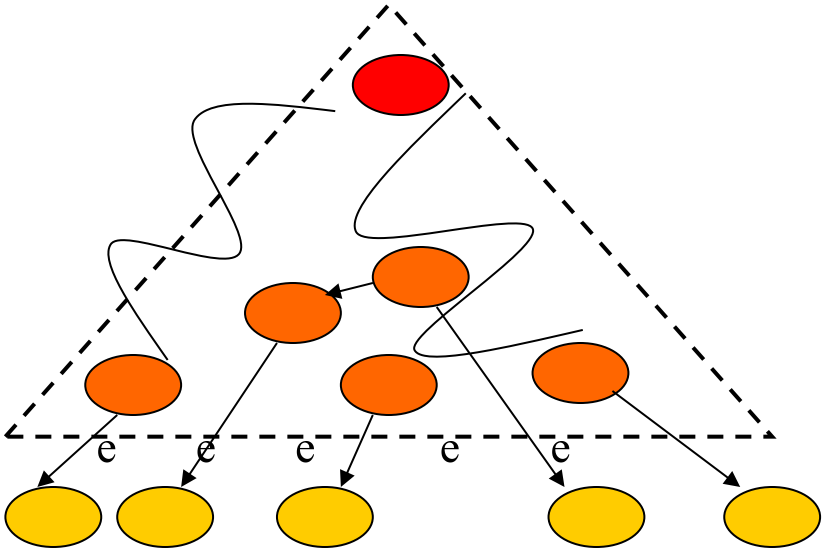
A bivalent initial config.

let e=(p,m) be some event applicable to the initial config.

Let ***C*** be the set of configurations reachable **without** applying e

Lemma 3

A bivalent initial config.

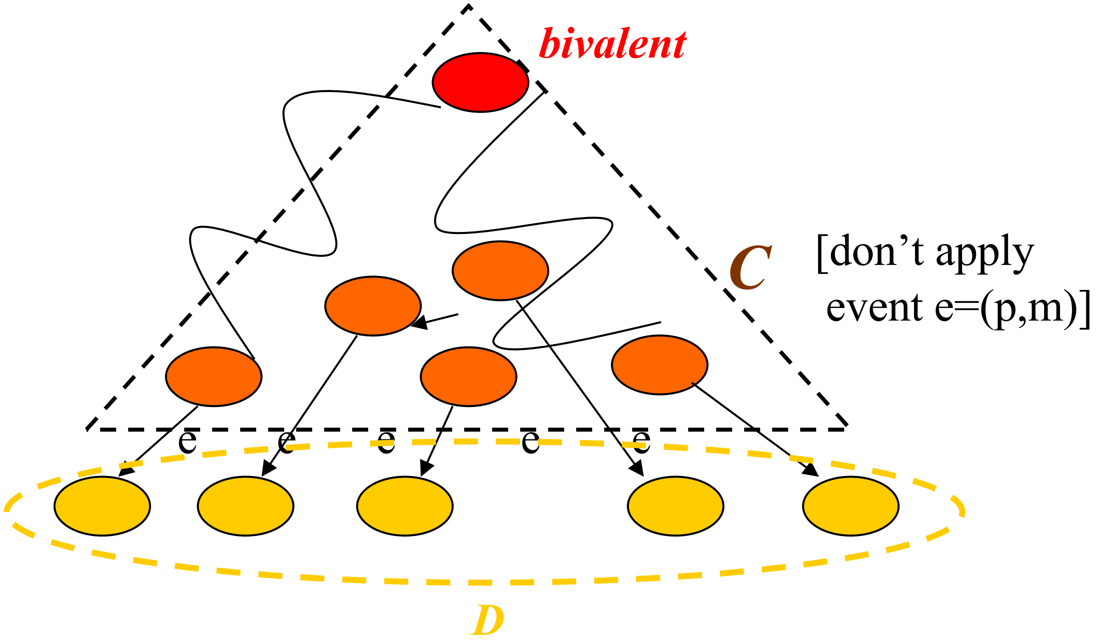


let e=(p,m) be some event applicable to the initial config.

Let *C* be the set of configurations reachable **without** applying e

Let *D* be the set of configurations obtained by **applying e** to some configurations in *C*

Lemma 3

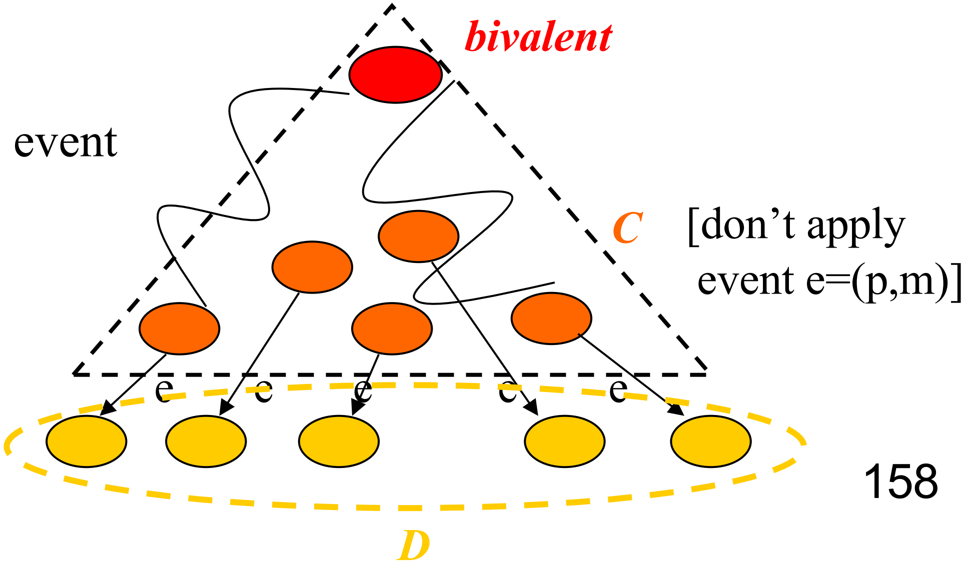


Claim. Set D contains a bivalent configuration

Proof. By contradiction. That is, suppose ***D*** has only 0- and 1- valent states (and no bivalent ones)

- There are states D0 and D1 in ***D***, and C0 and C1 in ***C*** such that
  - D0 is 0-valent, D1 is 1-valent
  - D0=C0 foll. by e=(p,m)
  - D1=C1 foll. by e=(p,m)
  - And C1 = C0 followed by some event e'=(p', m')

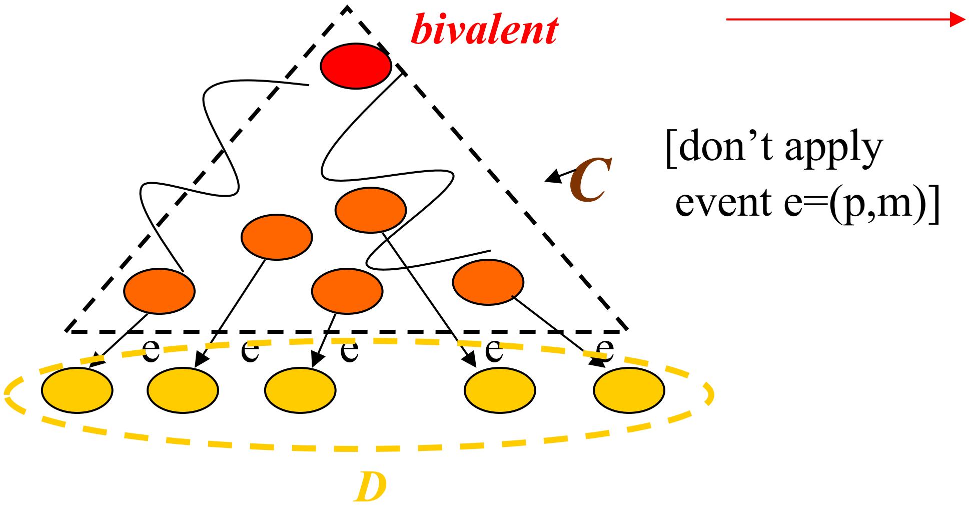
(why?)



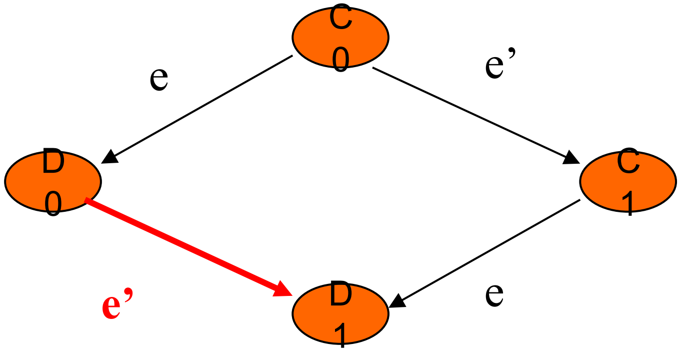
Proof. (contd.)

- Case I: p’ is not p

- Case II: p’ same as p



[don’t apply event e=(p,m)]

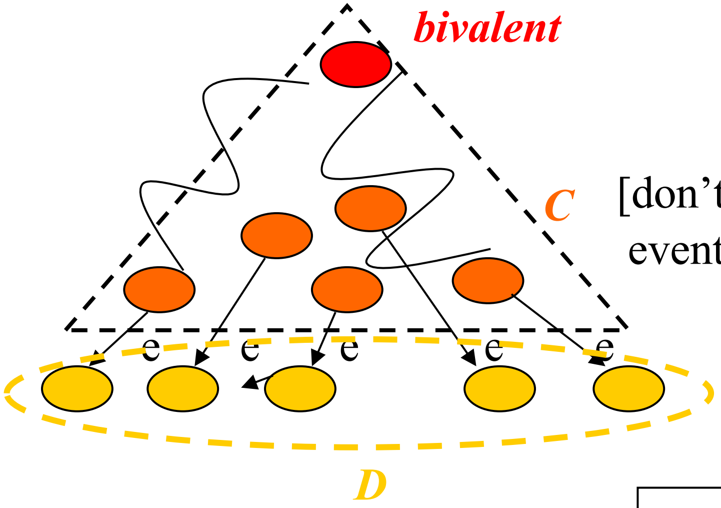


Why? (Lemma 1)
But D0 is then bivalent!

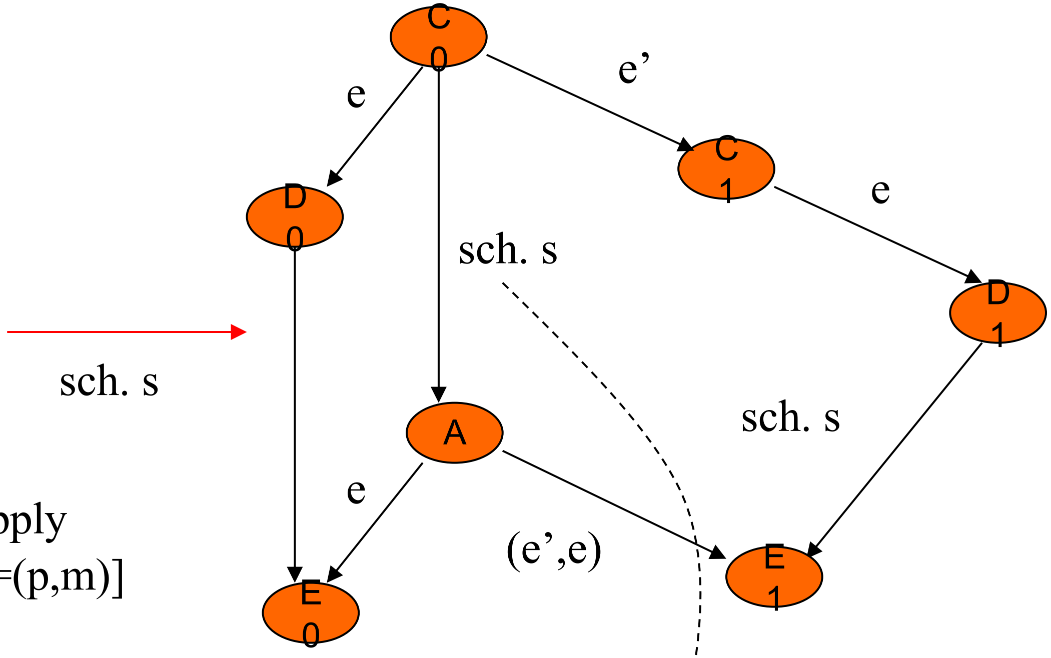
Proof. (contd.)

- Case I: p’ is not p

- Case II: p’ same as p



[don’t apply
event e=(p,m)]



But A is then bivalent!

sch. s
- finite
- **deciding run** from C0
- *p takes no steps*

FLP Proof Sketch:

- **Bivalent and Univalent states:** A decision state is bivalent, if starting from that state, there exist two distinct executions leading to two distinct decision values 0 or 1. Otherwise it is univalent.

  Bivalent ---> outcome is unpredictable

- **Process:** has state
- **Network:** Global buffer (processes put and get messages)
- **Configuration** -- global state (state for each process + state of global buffer)
- **Atomic Events** -- receipt of message by process p, processing of message (may change state), send out all needed messages from p
- **Schedule**: sequence of atomic events

Lemma 1: Schedules are commutative
Lemma 2: There exists an initial configuration that is bivalent.

Lemma 3: Starting from a bivalent config., there is always another bivalent config. that is reachable

Summary

- Consensus Problem
  - Agreement in distributed systems
  - Solution exists in synchronous system model (e.g., supercomputer)
  - Impossible to solve in an asynchronous system (e.g., Internet, Web)
    - Key idea: with even one (adversarial) crash-stop process failure, there are always sequences of events for the system to decide any which way
    - Holds true regardless of whatever algorithm you choose!
  - FLP impossibility proof
- One of the most fundamental results in distributed systems

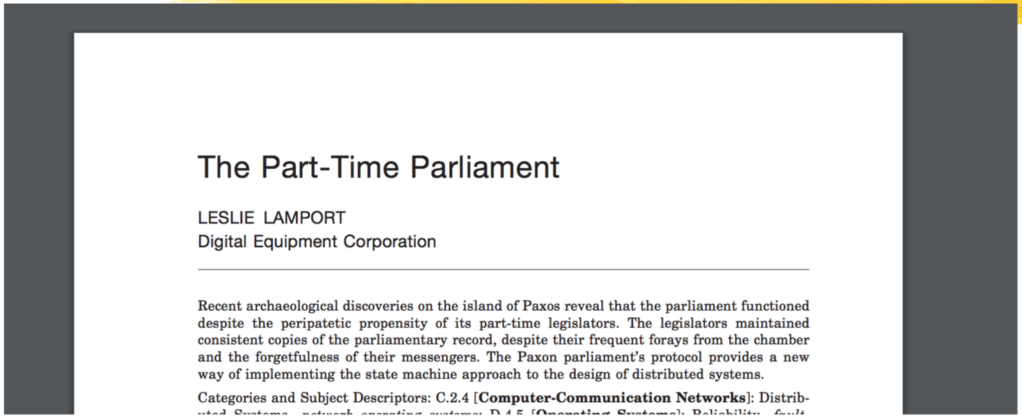
The PAXOS Algorithm
-- Towards a Practical Approach to Consensus

Landmark papers by Leslie Lamport (1998)

- Does not solve pure consensus problem (impossibility); But, provides consensus with a twist
- Paxos provides safety and eventual liveness
    - Safety: Consensus is not violated
    - Eventual Liveness: If things go well sometime in the future (messages, failures, etc.), there is a good chance consensus will be reached. But there is no guarantee.
- FLP result still applies: Paxos is not *guaranteed* to reach Consensus (ever, or within any bounded time)
- Used in Zookeeper (Yahoo!), Google Chubby, and many other companies

Brieft History

- People thought that Paxos was a joke.
- Lamport published it 8 years after it was written in 1990. – Title: *The Part-Time Parliament* [1]
- People did not understand the paper.
- Lamport gave up and wrote another paper that explains Paxos in simple English.
  - Title: *Paxos Made Simple* [2]
  - Abstract: "The Paxos algorithm, when presented in plain English, is very simple."
- It's still not the easiest algorithm to understand.
- People have written papers and lecture notes to explain *Paxos Made Simple*. (e.g., *Paxos Made Moderately Complex* [4], *Paxos Made Practical* [5], *etc.*)

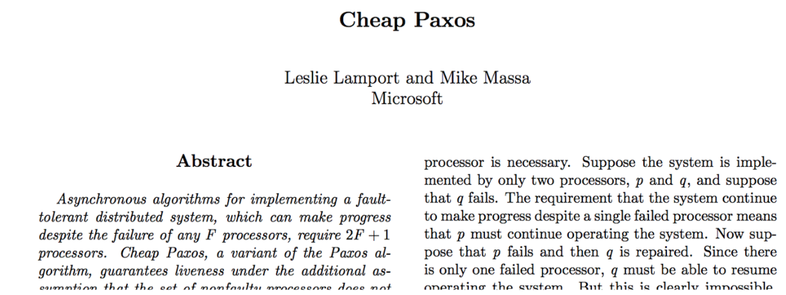


The Part-Time Parliament

LESLIE LAMPORT
Digital Equipment Corporation

Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—network operating systems; D.4.5 [**Operating Systems**]: Reliability—fault-



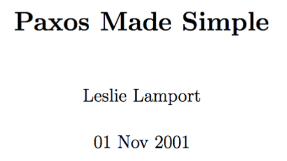
Cheap Paxos

Leslie Lamport and Mike Massa
Microsoft

Abstract

Asynchronous algorithms for implementing a fault-tolerant distributed system, which can make progress despite the failure of any F processors, require 2F + 1 processors. Cheap Paxos, a variant of the Paxos algorithm, guarantees liveness under the additional assumption that the set of nonfaulty processors does not

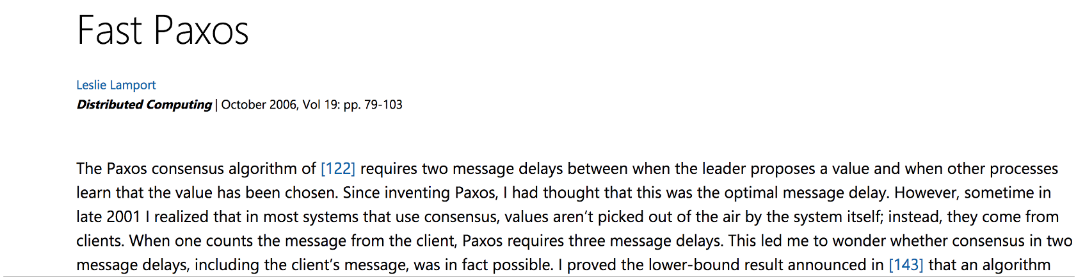
processor is necessary. Suppose the system is implemented by only two processors, p and q, and suppose that q fails. The requirement that the system continue to make progress despite a single failed processor means that p must continue operating the system. Now suppose that p fails and then q is repaired. Since there is only one failed processor, q must be able to resume operating the system. But this is clearly impossible



Paxos Made Simple

Leslie Lamport

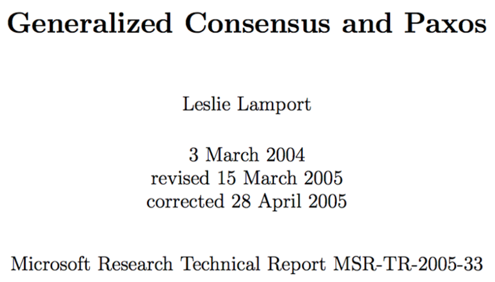
01 Nov 2001



Fast Paxos

Leslie Lamport
Distributed Computing | October 2006, Vol 19: pp. 79-103

The Paxos consensus algorithm of [122] requires two message delays between when the leader proposes a value and when other processes learn that the value has been chosen. Since inventing Paxos, I had thought that this was the optimal message delay. However, sometime in late 2001 I realized that in most systems that use consensus, values aren't picked out of the air by the system itself; instead, they come from clients. When one counts the message from the client, Paxos requires three message delays. This led me to wonder whether consensus in two message delays, including the client's message, was in fact possible. I proved the lower-bound result announced in [143] that an algorithm

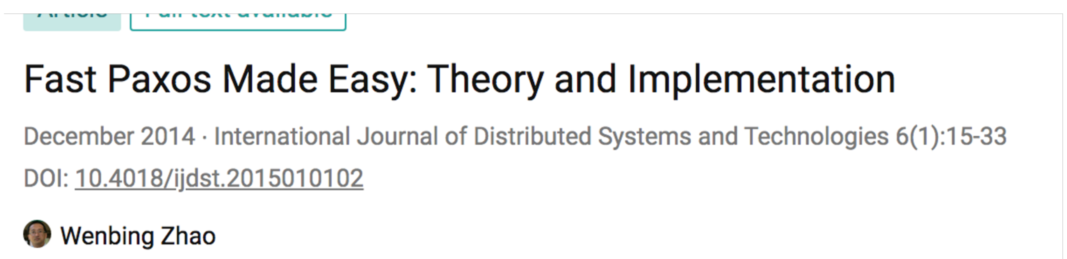


Generalized Consensus and Paxos

Leslie Lamport

3 March 2004
revised 15 March 2005
corrected 28 April 2005

Microsoft Research Technical Report MSR-TR-2005-33



Article | Full text available

Fast Paxos Made Easy: Theory and Implementation

December 2014 · International Journal of Distributed Systems and Technologies 6(1):15-33

DOI: 10.4018/ijdst.2015010102

Wenbing Zhao

more: Stoppable Paxos, Vertical Paxos, Egalitarian Paxos, …

Recall the Consensus Problem

N processes

- Every process contributes a value
- Goal: To have all processes decide on the same (some) value
  - Once made, the decision cannot be changed.

Each process p has

- input variable xp : initially either 0 or 1
- output variable yp : initially b (b=undecided) – can be changed only once

Consensus problem: design a protocol so that either

1. all non-faulty processes set their output variables to 0
2. Or non-faulty all processes set their output variables to 1
3. There is at least one initial state that leads to each outcomes 1 and 2 above

Assumptions

- Failures
  - "Fail Stop" assumption
    - When a node fails, it ceases to function entirely.
    - May resume normal operation when restarted.
  - Messages
    - May be lost.
    - May be duplicated.
    - May be delayed (and thus reordered).
    - May ***not*** be corrupt.
- Stable Storage
  - preserves info recorded before a failure

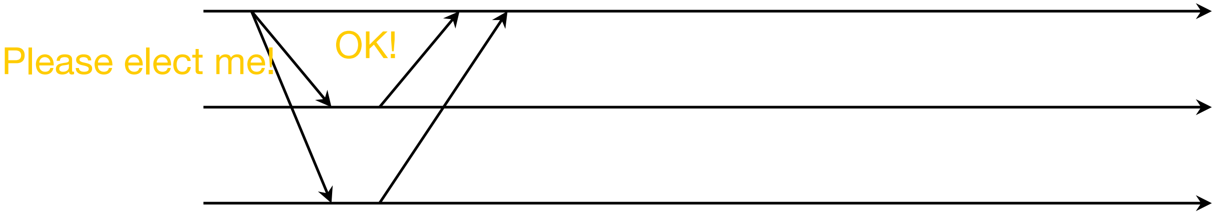
The Paxos Strategy

- Paxos has rounds; each round has a unique ballot id
- Rounds are asynchronous
  - Time synchronization not required
  - If you're in round *j* and hear a message from round *j+1*, abort everything and move over to round *j+1*
  - Use timeouts (then - eventually synchronous)
- Each round itself broken into phases
  - Phase 1: A leader is elected (Election)
  - Phase 2: Leader proposes a value to others (acceptors), processes ack (Bill)
  - Phase 3: Leader multicasts final value (Law)

- http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf

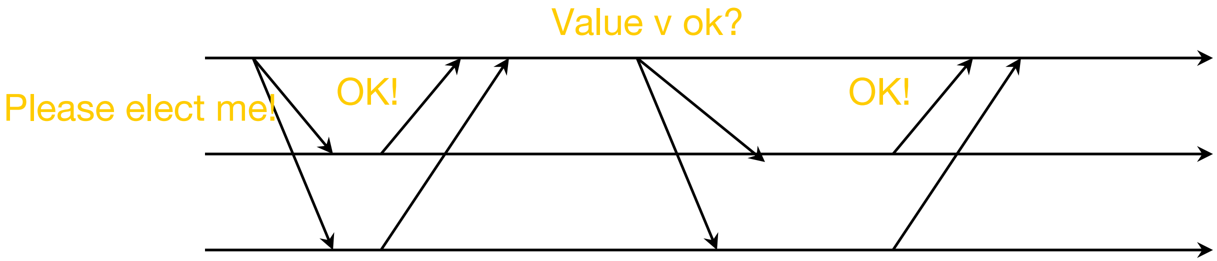
Paxos Strategy
Phase 1 – Election

- Potential leader chooses a unique ballot id, higher than seen anything so far
- Sends to all processes
- Processes wait, respond once to highest ballot id
  - If potential leader sees a higher ballot id, it can't be a leader
  - Paxos tolerant to multiple leaders, but we'll only discuss 1 leader case
  - Processes also log received ballot ID on disk
- If a process has in a previous round decided on a value v', it includes value v' in its response
- If majority (i.e., quorum) respond OK then you are the leader
  - If no one has majority, start new round
- (If things go right) A round cannot have two leaders (why?)



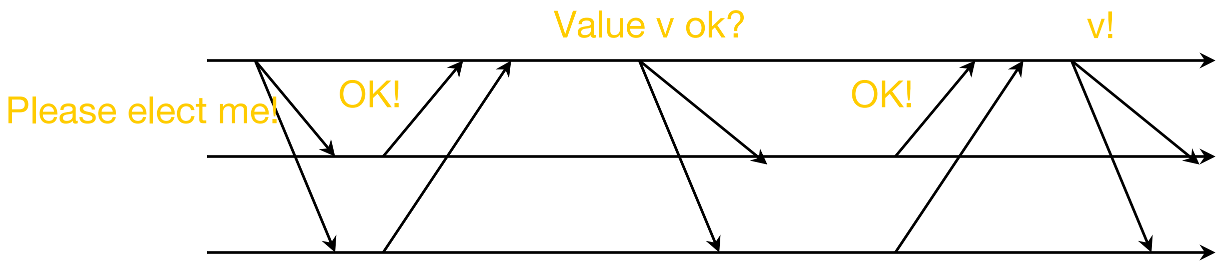
Paxos Strategy Phase 2 – Proposal

- Leader sends proposed value v to all
  - use v=v' if some process already decided in a previous round and sent you its decided value v'
  - If multiple such v' received, use latest one
- Recipient logs on disk; responds OK



Paxos Strategy
Phase 3 – Decision

- If leader hears a majority of OKs, it lets everyone know of the decision
- Recipients receive decision, log it on disk
- Consensus is reached



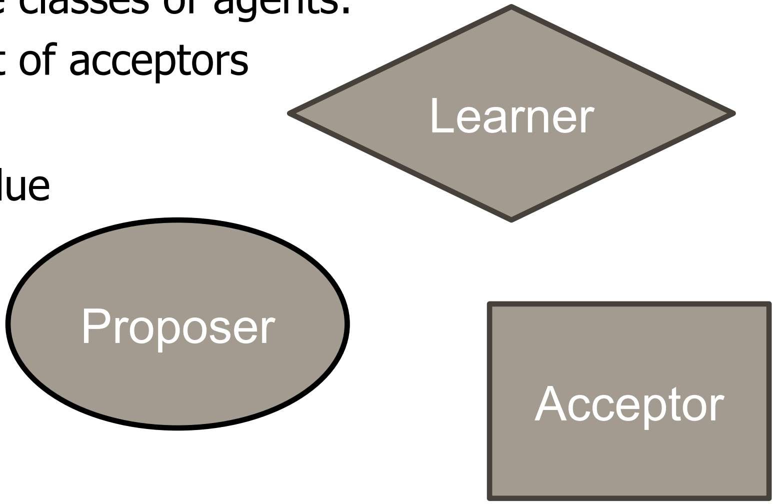
Paxos Algorithm

Assumptions

- Asynchronous system
- Non-Byzantine model
- Information can be remembered by a process that has failed and restarted

Paxos Agents:

- The algorithm is performed by three classes of agents:
  - *Proposer*: send a value to a set of acceptors
  - Acceptor: can accept a value
  - Learner: learn the accepted value



Paxos Algorithm: choose a value

The 2 phase algorithm for choosing a value

-

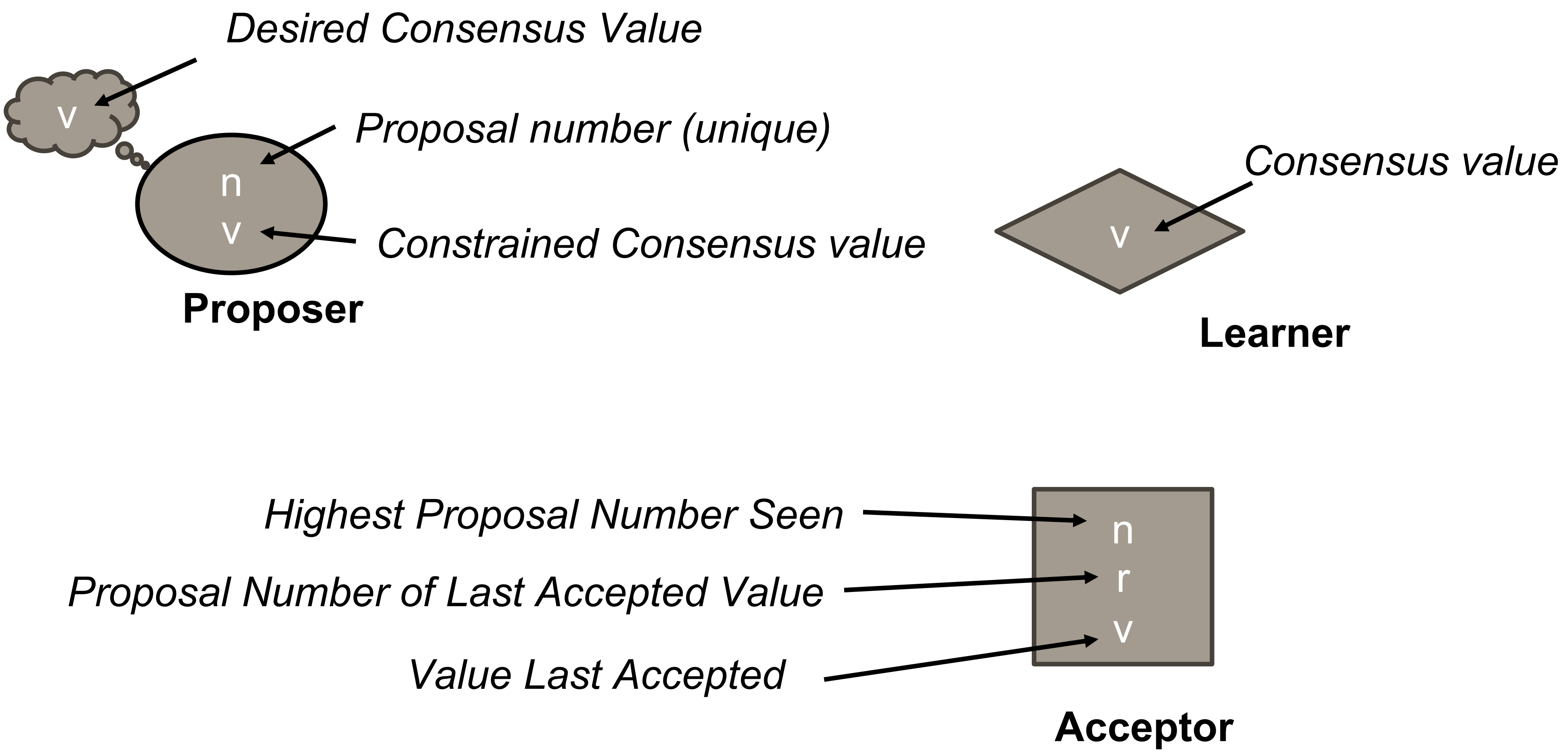
Phase 1:

- A proposer selects a proposal number n and sends a $prepare(n)$ request to a majority of acceptors.
- An acceptor who receives a $prepare(n)$ with the proposal number n is greater than that of any $prepare$ request it has responded, it will response a $promise(n, r, val)$ to not accept any more proposals numbered less than n
  - r is proposal number of last accepted value
  - val is last accepted value

Phase 2:

- If the proposer receives a response to its $prepare(n)$ requests from a majority of acceptors, then it sends an $accept(n, v)$ request to each of those acceptors

Paxos Agents



Desired Consensus Value

Proposal number (unique)

Constrained Consensus value

Proposer

Consensus value

Learner

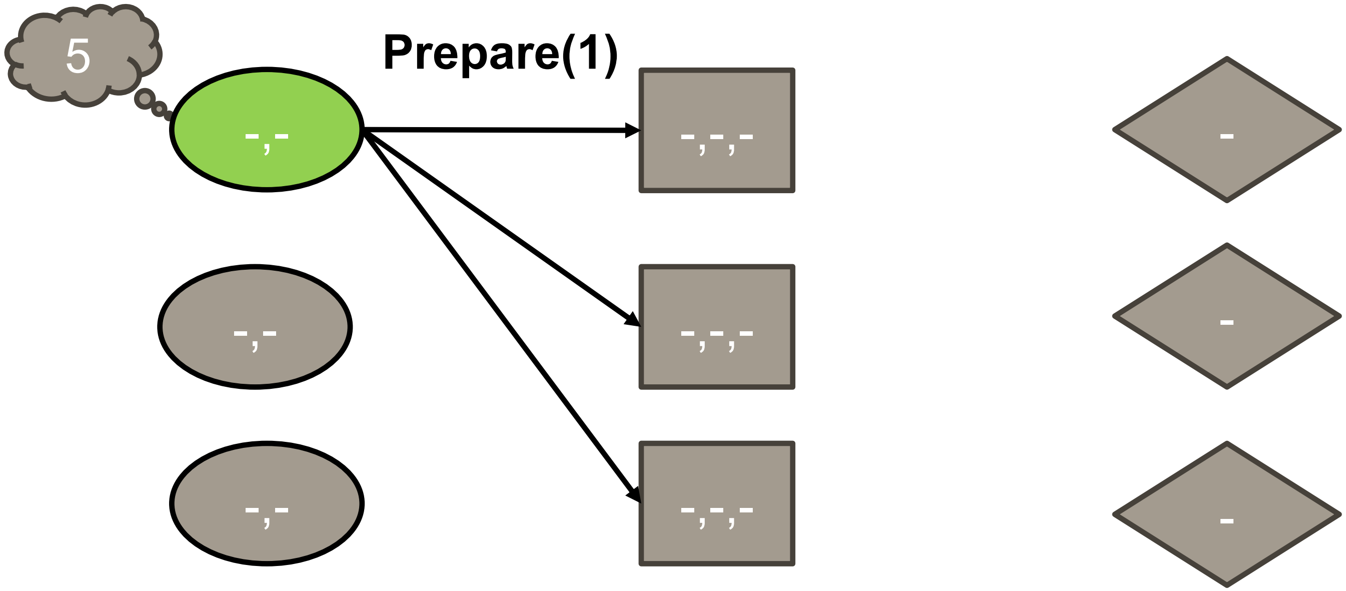
Highest Proposal Number Seen

Proposal Number of Last Accepted Value

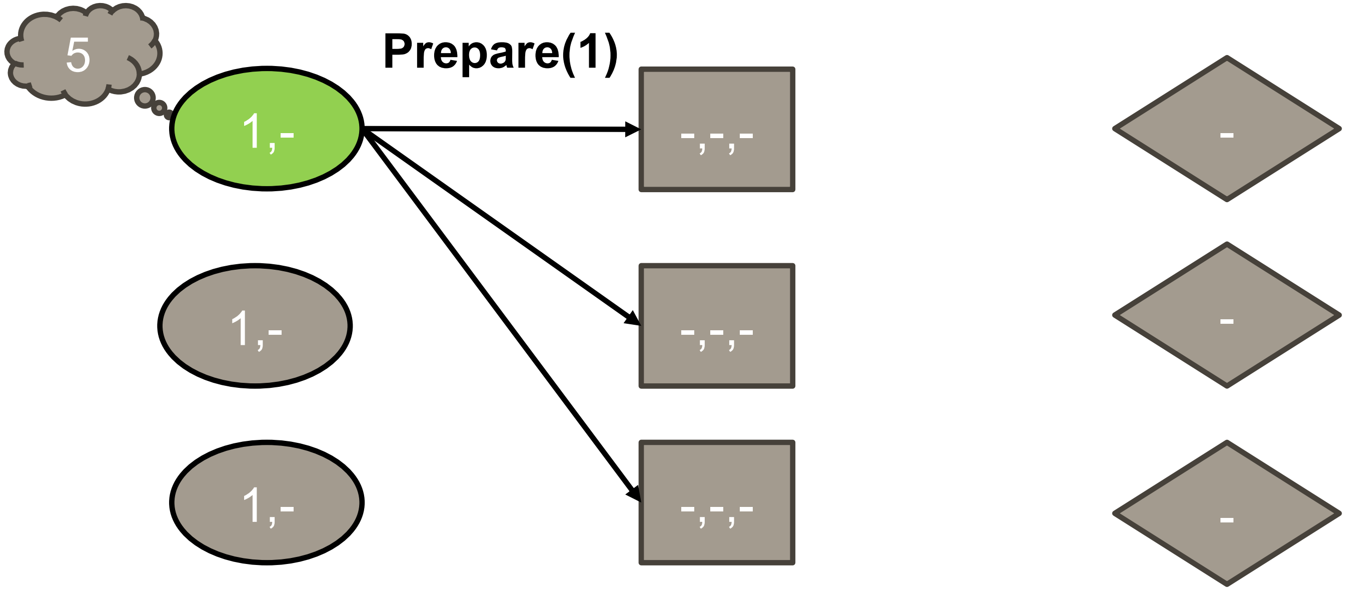
Value Last Accepted

Acceptor

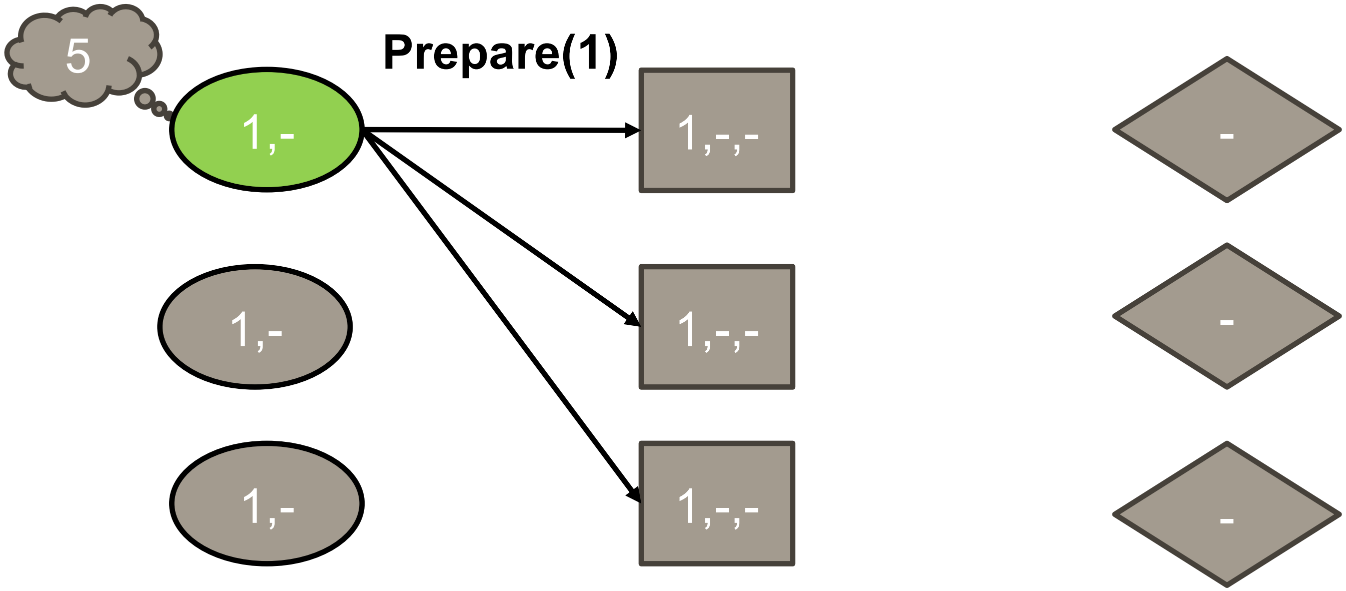
Paxos Execution Example



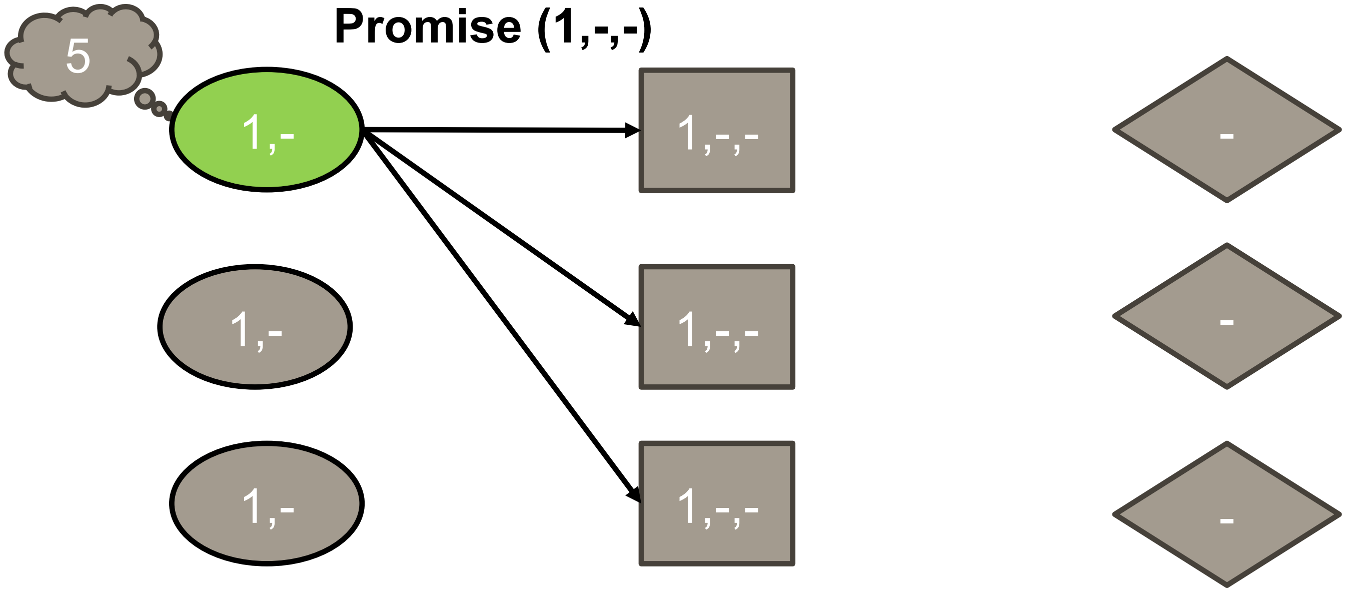
Paxos Execution Example



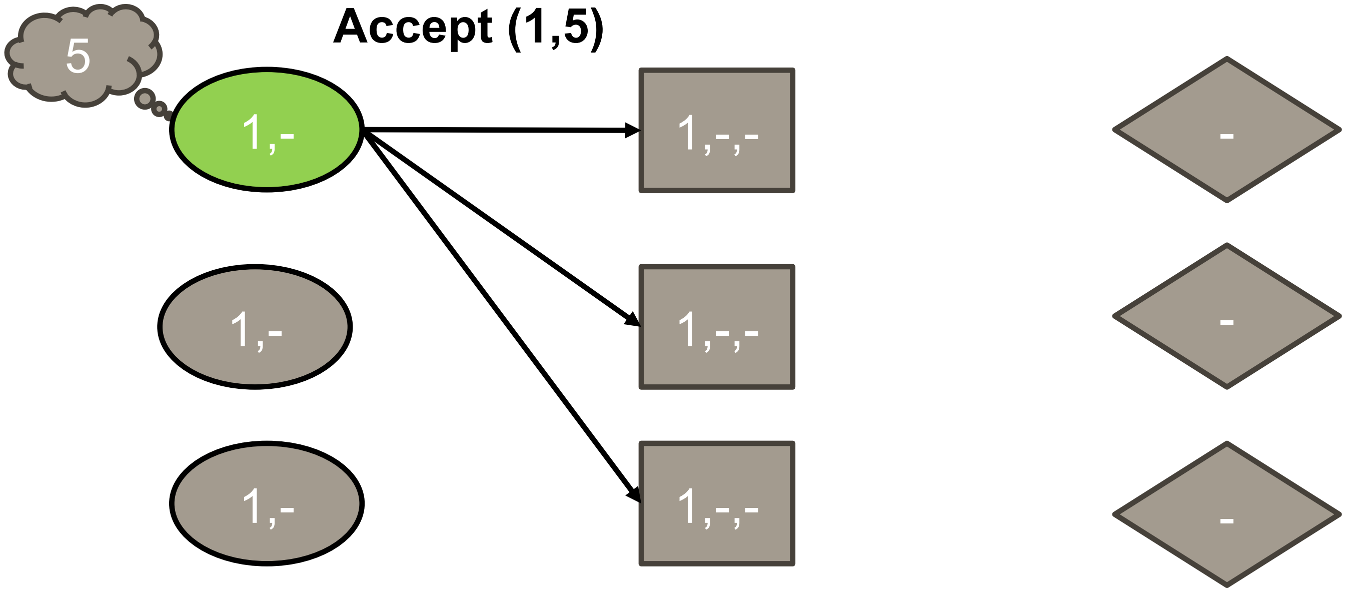
Paxos Execution Example



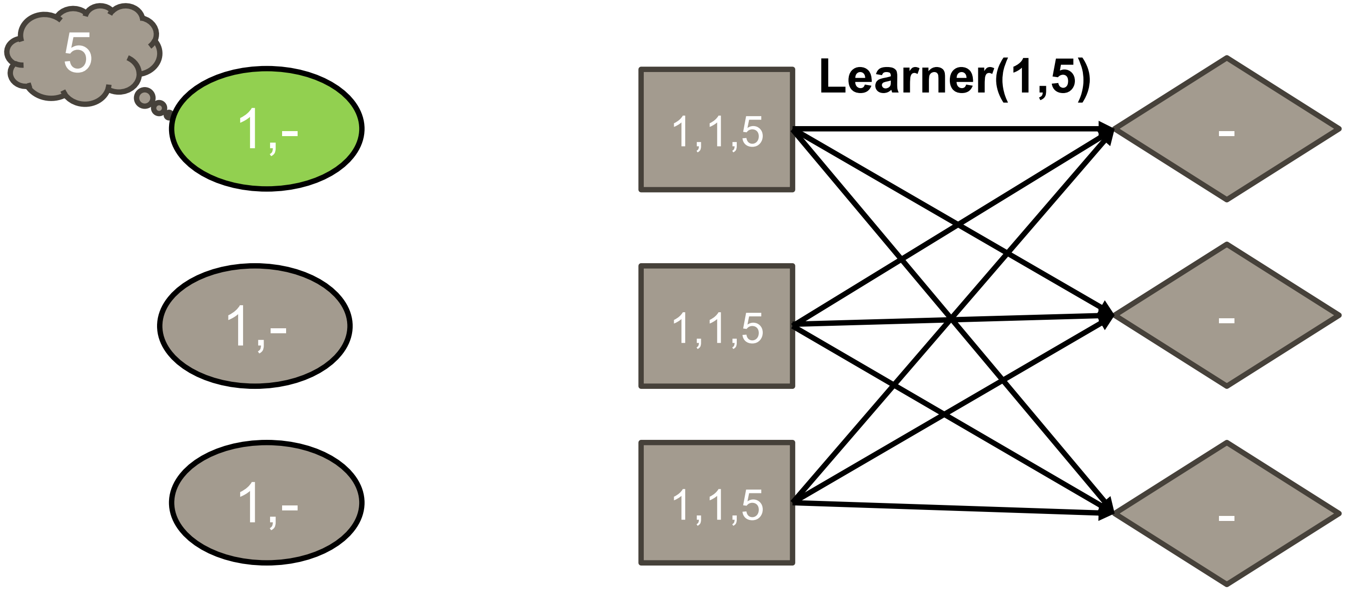
Paxos Execution Example



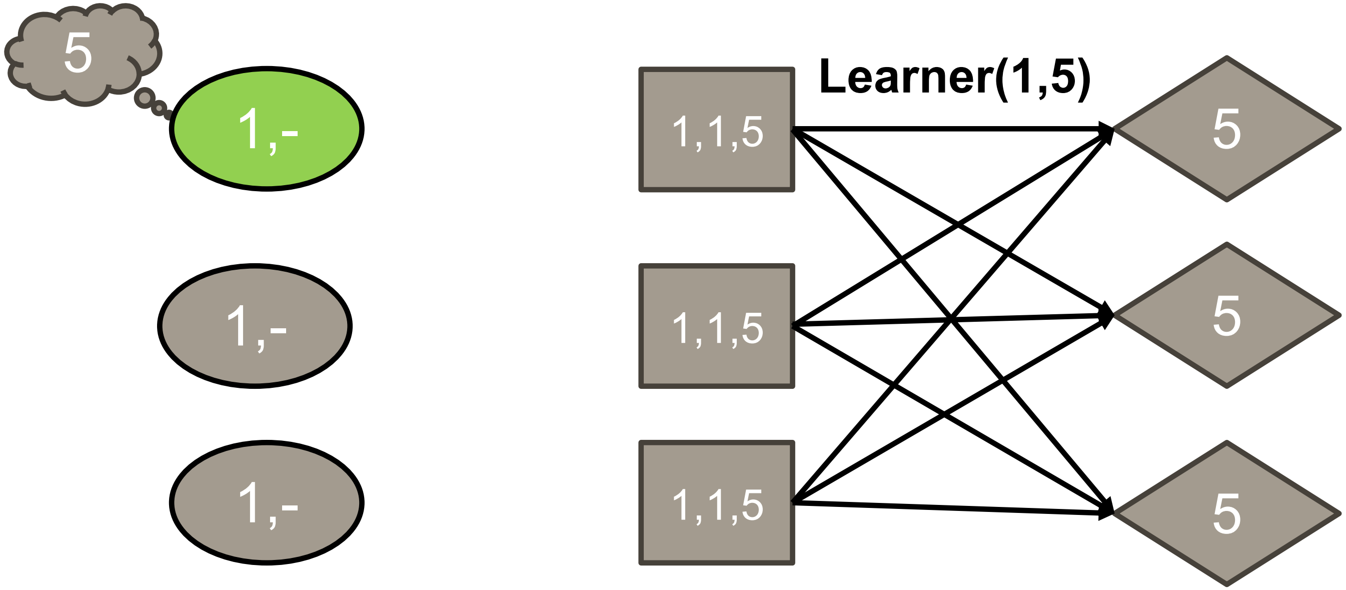
Paxos Execution Example



Paxos Execution Example



Paxos Execution Example



Paxos Protocol Implementation - Terms

- ***Proposer*** P1
  - Suggests values for consideration by Acceptors.
  - Advocates for a client.
- ***Acceptor*** A1
  - Considers the values proposed by proposers.
  - Renders an accept/reject decision.
- ***Learner***
  - Learns the chosen value.
- In practice, each node will usually play all three roles.

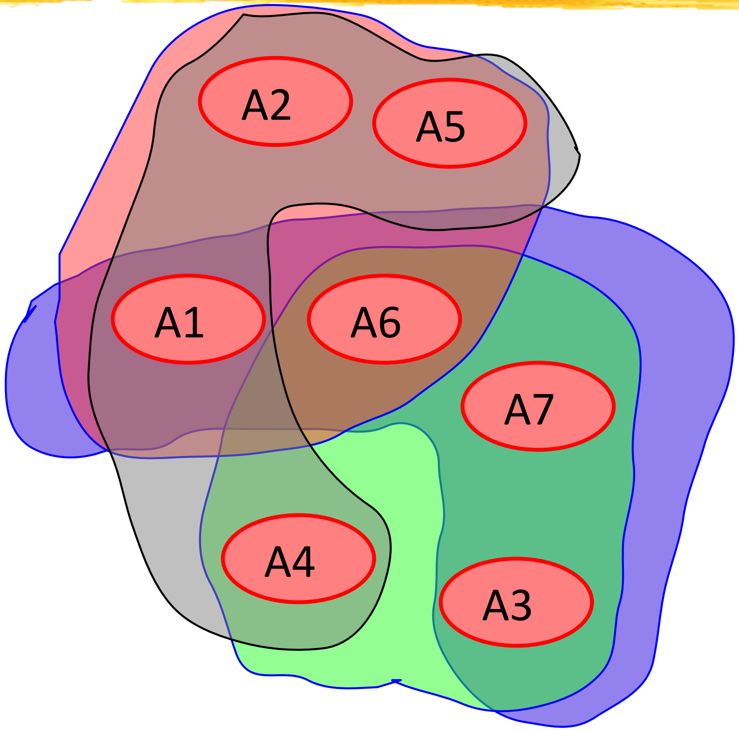
- ***Proposal***
  - An alternative proposed by a proposer.
  - Consists of a unique ***number*** and a proposed ***value.***

```
( 42, B )
```

- **We say a value is *chosen* when consensus is reached on that value.**

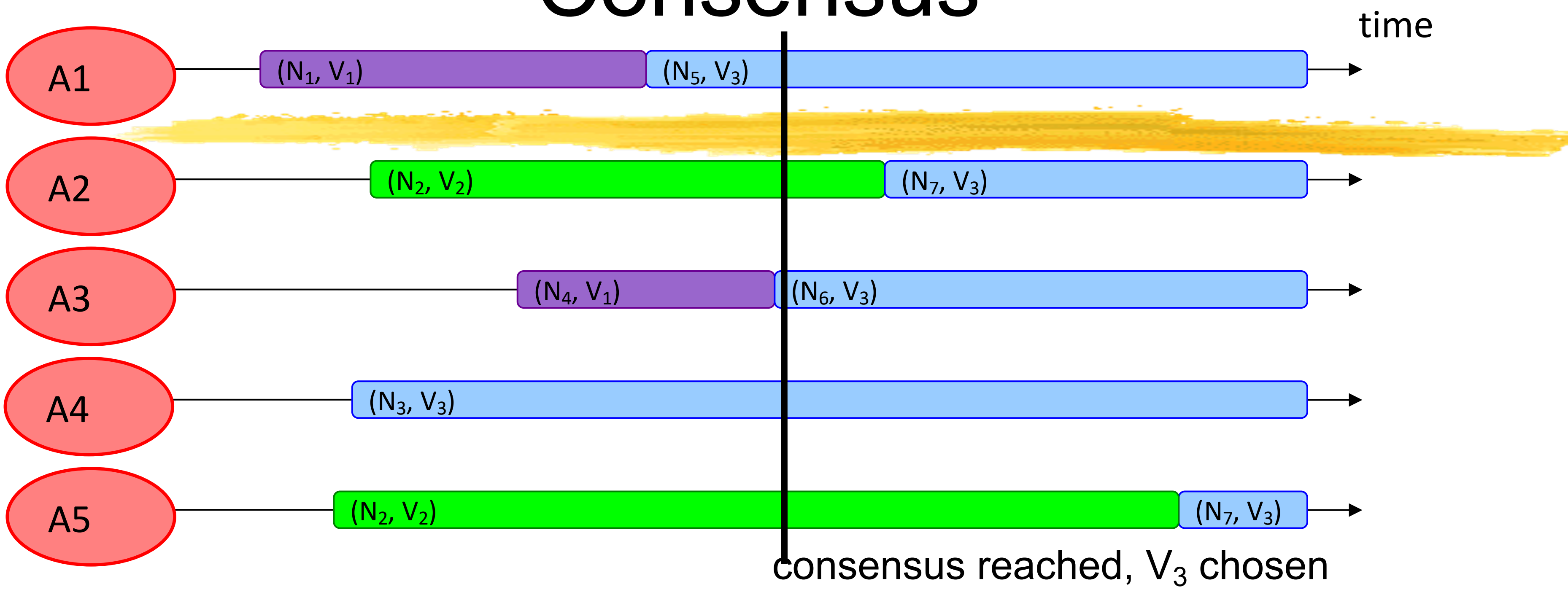
Strong Majority

- “Strong Majority” / “Quorum”
  - A set of acceptors consisting of more than half of all acceptors.
- Any two quorums have a nonempty intersection.
- Helps avoid “split-brain” problem.
  - Acceptors decisions are not in agreement.
  - Common node acts as “tie-breaker.”
- In a system with 2F+1 acceptors, F acceptors can fail and we'll be OK.



Quorums in a system with seven acceptors.

Consensus



- Values proposed by proposers are constrained so that once consensus has been reached, all future proposals will carry the chosen value.
- P2c . For any v and n, if a proposal with value v and number n is issued, then there is a set S consisting of a majority of acceptors such that either:
  - (a) no acceptor in S has accepted any proposal numbered less than n, or
  - (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by the acceptors in S.

Basic Paxos Algorithm

Phase 1a: **“Prepare”**
Select proposal number* *N* and send a ***prepare(N)*** request to a quorum of acceptors.

Proposer

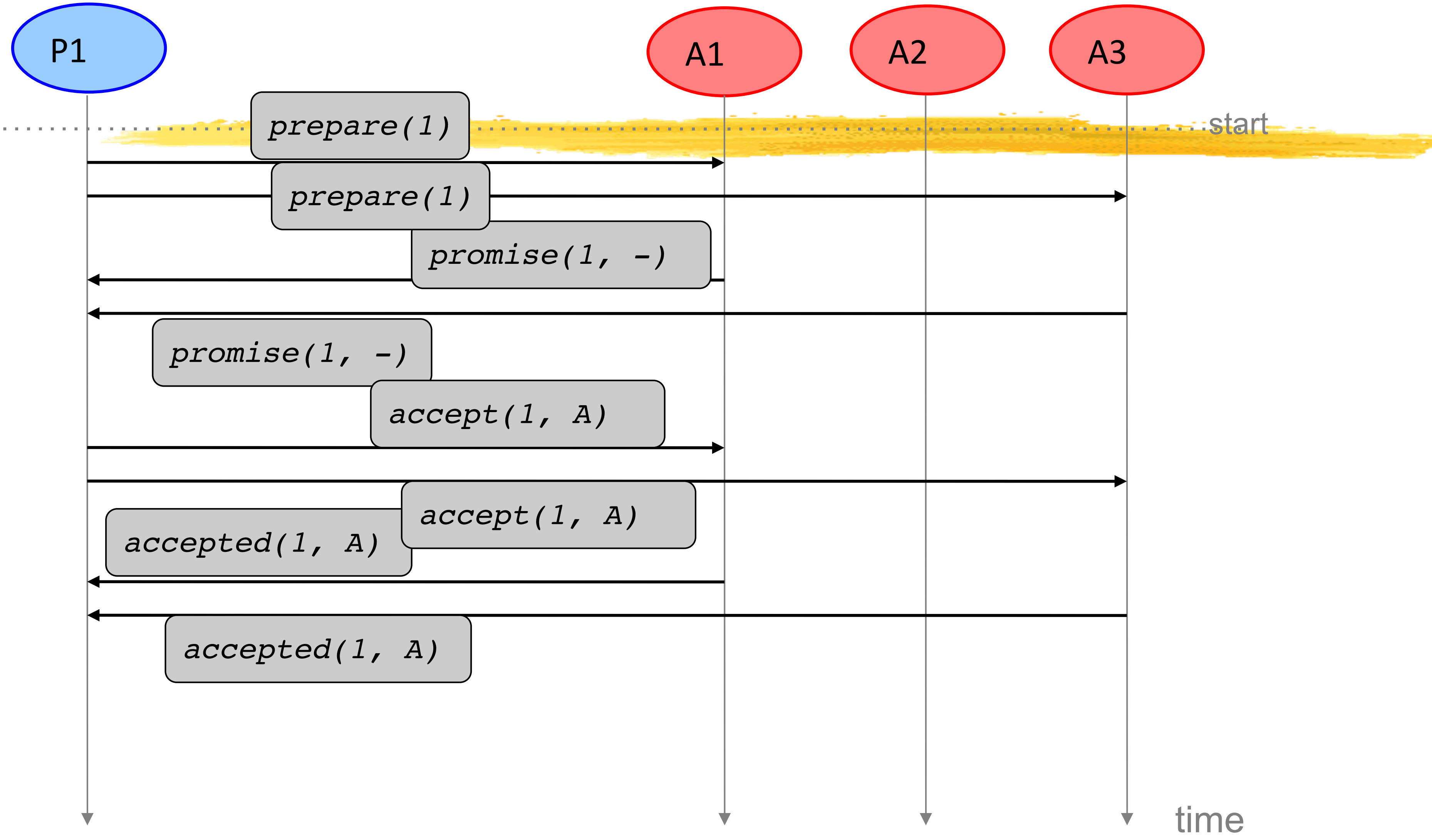
Phase 1b: **“Promise”**
If N > *number of any previous promises or acceptances*,
* promise to never accept any future proposal less than *N*,
- send a ***promise(N, U)*** response
(where *U* is the highest-numbered proposal value accepted so far (

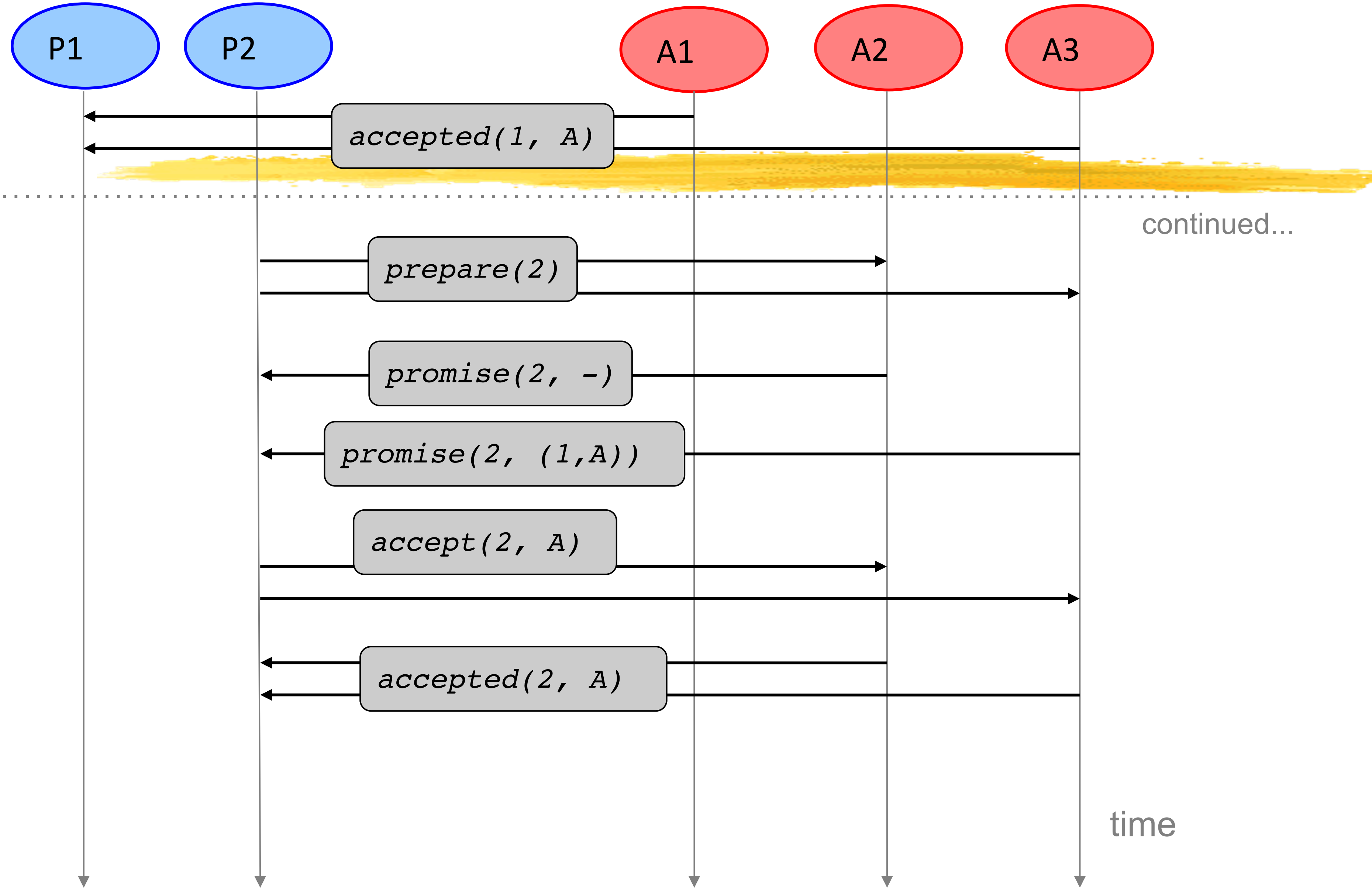
Phase 2a: **“Accept!”**
If proposer received promise responses from a quorum,
- send an ***accept(N, W)*** request to those acceptors
(where ***W*** is the value of the highest-numbered proposal among the ***promise*** responses, or any value if no ***promise*** contained a proposal)

Acceptor

Phase 2b: **“Accepted”**
If N >= *number of any previous promise*,
* accept the proposal
- send an ***accepted*** notification to the learner

* = record to stable storage


P1
A1
A2
A3
prepare(1)
start
prepare(1)
promise(1, -)
promise(1, -)
accept(1, A)
accept(1, A)
accepted(1, A)
accepted(1, A)
time


P1
P2
A1
A2
A3
accepted(1, A)
continued...
prepare(2)
promise(2, -)
promise(2, (1,A))
accept(2, A)
accepted(2, A)
time

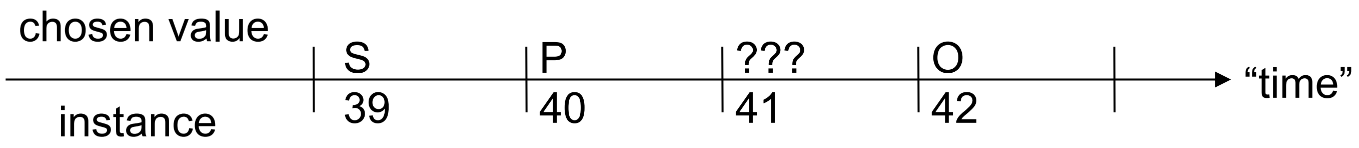
Other Considerations

- Liveness
  - Can't be guaranteed in general.
  - Distinguished Proposer
    - All proposals are funneled through one node.
  - Can re-elect on failure.

- Learning the Chosen Value
  - Acceptors notify some set of learners upon acceptance.
  - Distinguished Learner

- A node may play the role of both distinguished proposer and distinguished learner – we call such a node the ***master***.

Multi-Paxos



- A single ***instance*** of Paxos yields a single chosen value.
- To form a sequence of chosen values, simply apply Paxos iteratively.
  - To distinguish, include an instance number in messages.
- Facilitates replication of a state machine.

Paxos Variations

- **Cheap Paxos**
  - Reconfiguration
    - Eject failed acceptors.
    - Fault-tolerant with only F+1 nodes (vs 2F+1)
    - Failures must not happen too quickly.

- **Byzantine Paxos**
  - Arbitrary failures – lying, collusion, fabricated messages, selective non-participation.
  - Adds an extra “verify” phase to the algorithm.

- **Fast Paxos**
  - Clients send *accept* messages to acceptors.
  - Master is responsible for breaking ties.
  - Reduces message traffic.

Raft distributed consensus

https://www.cs.rutgers.edu/~pxk/417/notes/raft.html

Consensus problem

- The problem of consensus is getting a group of processes to *unanimously* agree on a *single* result. There are four requirements to such an algorithm:
  - **Validity**. The result must be a value that was submitted by at least one of the processes. The consensus algorithm cannot just make up a value.
  - **Uniform agreement**. All nodes must select the same value.
  - **Integrity**. A node can select only a single value. That is, a node cannot announce one outcome and later change its mind.
  - **Termination**. Also known as **progress**, every node must eventually reach a decision.

Replicated state machines

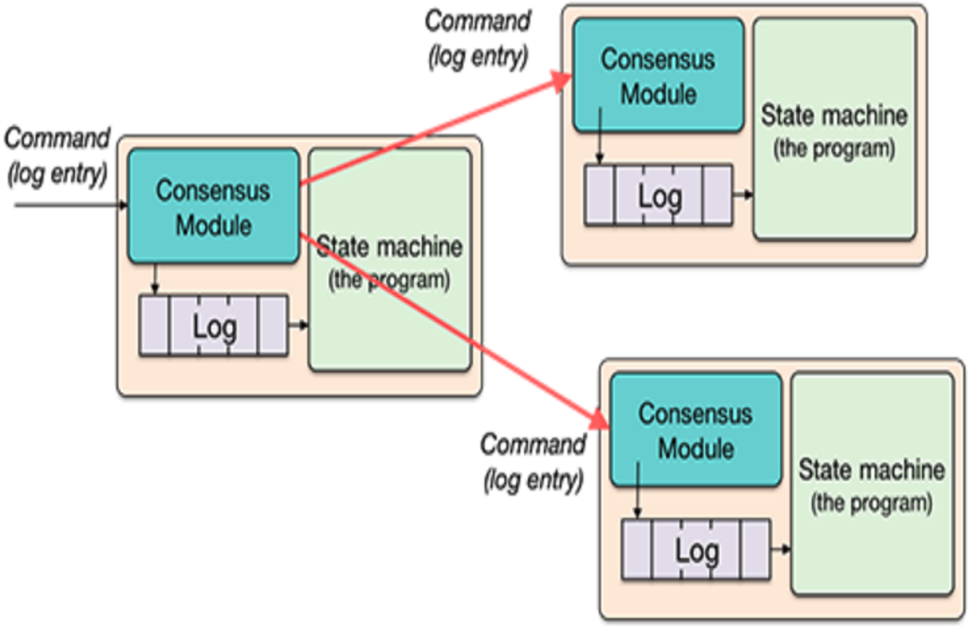
- Multiple systems in an identical state so that the system can withstand the failure of some of its members and continue to provide its service
- Replicated state machines is a fault tolerant technique which is commonly used distributed systems where a central coordinator is needed:
  - Google Chubby provides a centralized namespace and lock management service for the Google File System and various other services
  - Apache ZooKeeper coordinates various Apache services, including the HDFS
  - Big data processing frameworks such as MapReduce, Apache Spark, and Kafka

State Machine

- *State machines* are programs that store "state" – data that changes based on inputs received by the programs
- Programs are deterministic as multiple identical copies of the program will all modify their data ("state") in the same way when presented with the same input
- State machine system examples:
  - Key-value store
  - Database updates
  - File updates

REPLICATED LOGS

- A way to implement **replicated state machines**
- Log is a list of commands that are received and stored by each state machine and is used as input by the state machine.
- The commands in the log must be identical and in the same order across all replicas to ensure state machine synchronization
- **Consensus algorithm** is applied to keep the logs consistent



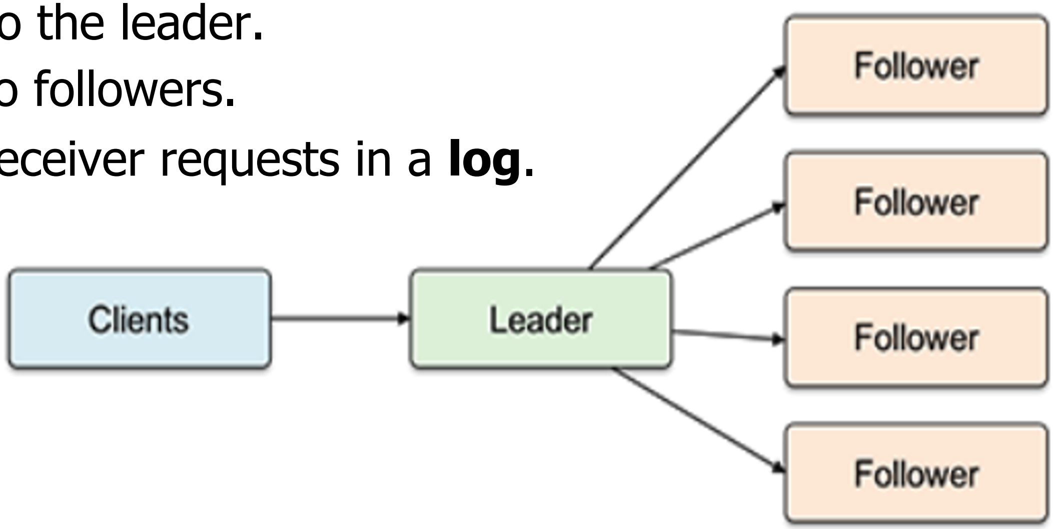
Consensus modules

RAFT

- A consensus algorithm designed for managing a replicated log
- Created at Stanford University in 2014 by Diego Ongaro and John Ousterhout
- An alternative to Paxos for log replication but also easier to understand, implement, and validate
- Raft separates the functions of leader election and log replication.

Raft environment

- A group of servers containing:
  - The state machine (the service that the server provides)
  - A log that contains inputs fed into the state machine
  - The **Raft protocol**
- One server is *elected* to be the **leader**
- Other servers function as **followers**
- Clients send requests only to the leader.
- The leader forwards them to followers.
- Each of the servers stores receiver requests in a **log**.



Server states

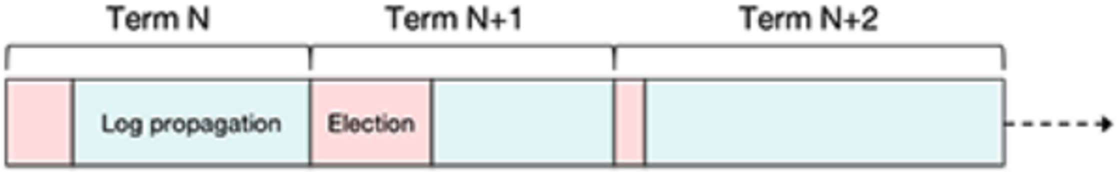
- A server operates in one of three states:
  - **Leader**. The leader handles all client requests and responses. There is only one leader at a time.
  - **Candidate**. A server may become a candidate during the election phase. One leader will be chosen from one or more candidates. Those not selected will become followers.
  - **Follower**. The follower does not talk to clients. It responds to requests from leaders and candidates.

RAFT Messages

- The Raft protocol comprises two remote procedure calls (RPCs):
  - **RequestVotes:** Used by a candidate during elections to try to get a majority vote
  - **AppendEntries:** Used by leaders to communicate with followers to:
    - Send log entries (data from clients) to replicas.
    - Send commit messages to replicas. That is, inform a follower that a majority of followers received the message
    - Send heartbeat messages. This is simply an empty message to indicate that the leader is still alive.

RAFT Terms

- The timeline of operations in Raft is broken up into ***terms***
- Each **term** has a unique number and begins with a leader election phase
- After a leader is elected, it propagates log entries to followers



Terms

- If, at some point in time, a follower ceases to receive RPCs from the leader or a candidate:
  - Another election takes place and another term begins with an incremented term number
- If a server discovers that its current term number is smaller than that in a received message, it updates its term number to that in the received message
- If a leader or candidate receives a message with a higher term number then it changes its state back to a *follower* state.

Leader election

- All servers start up as *followers* and wait for a *message* from the leader:
  - *AppendEntries* RPC
- If a follower does not receive a message from a leader within a specific time interval, it becomes a *candidate* and starts an election to choose a new leader:
  - Sends *RequestVotes* messages to all the other servers asking them for a vote
  - If it gets votes from a majority of servers then it becomes a *leader*

Leader election: timeouts

- Each follower sets an **election timeout:**
  - maximum amount of time that a follower is willing to wait without receiving a message from a leader before it starts an election
  - **Randomized election timeout** per follower to reduce the chance that multiple servers will start elections at the same time
- A follower starts an election to choose a new leader when timeout:
  - Increments its current term
  - Set itself to the *candidate* state.
  - Sets a timer
  - Sends *RequestVote* RPCs to all the other servers
- If a server receives a *RequestVote* message and hasn't yet voted, it votes for that candidate and resets its election timeout timer.

Leader election: timeouts (2)

- If a follower receives a majority votes from the group, including itself:
  - becomes the *leader*
  - starts sending out *AppendEntries* messages to the other servers at a **heartbeat** interval
- If a candidate receives an *AppendEntries* message while it is waiting for its votes:
  - If the term number in the message is the same or greater than the candidate’s term then the candidate will recognize the sender as the legitimate leader and become a follower.
  - If the term number in the message is smaller than the candidate’s term then the candidate rejects the request and continues with its election.
- **Election split vote**: if two or more nodes started elections with neither receiving a majority vote:
  - Each node time out waiting for a majority and start a new election

Log Replication

- Clients only communicate with that leader

- If the client query is a read:
  - the server can simply respond to the client

- If the client query is an update:
  - the leader will add the request to its log
  - the leader will add the request to its log

- Each server maintains a log of requests, each contains:
  - The client request (the command to be run by the server).
  - The term number when the command was received by the leader (to detect inconsistencies)
  - An integer that identifies the command’s position in the log.

Log Replication (2)

- A log entry is considered **committed** when the message has been replicated to the followers:
  - The leader can then execute the request and return any result to the client
  - Followers execute the same requests and keep their state machines in sync with the leader and each other

Log replication: consistency checks

- *AppendEntries* message from the leader to a follower contains:
  - The client request (the command to be run by the server)
  - The index number that identifies the command's position in the log.
  - The current term number.
  - The index number and term number of the preceding entry in the log.

- The follower does a consistency check when receives an *AppendEntries* message:
  - If the leader's term (in the message) is less than the follower's term then reject the message – some old leader missed an election cycle.
  - If a follower does not see the preceding index and term number in its log then it rejects the message
  - If the log contains a conflicting entry at that index – a different term number, delete the entry and all following entries from the log.

- If the message passes the consistency check, the follower will add the entry to its log and acknowledge the message
- When a log entry has been accepted by a majority of servers, it is considered **committed**

Log replication: making logs consistency

- Logs may get inconsistent: leader crashes, followers crash
- The leader is responsible for bringing a follower’s log up to date if inconsistencies are detected:
  - The leader finds out the latest log entry in common between the leader’s and follower’s log
  - The leader can send subsequent entries to synchronize the log and make it consistent

Elections: ensuring safety

- During the voting phase, a candidate cannot win an election if its log does not contain all committed entries
- The *RequestVotes* RPC sends information about the log length and the term of the latest log entry:
  - If a server receives a *RequestVotes* message and the candidate has an earlier term then the server will reject the vote.
  - If the term numbers are the same but the log length of a candidate is shorter than that of the server that receives the message, the server will reject the vote.

References

- Diego Ongaro and John Ousterhout, In Search of an Understandable Consensus Algorithm, 2014 USENIX Annual Technical Conference, June 2014, pp. 305–319
- Presentation video: Diego Ongaro and John Ousterhout, In Search of an Understandable Consensus Algorithm, 2014 USENIX Annual Technical Conference, June 2014.
- Video: Designing for Understandability: The Raft Consensus Algorithm, John Ousterhout, August 29, 2016.
- Brian Curran, What are the Paxos & Raft Consensus Protocols? Complete Beginner’s Guide, Blockonomi, November 14, 2018.
- Leslie Lamport, The Part-Time Parliament, August 2000. *This is the original Paxos Paper*.
- Leslie Lamport, Paxos Made Simple, November 2001.
- David Mazières, Paxos Made Practical, Stanford University, 2007
- The Raft Consensus Algorithm, Project page @ github: RaftScope visualization and lots of links to papers, videos, and implementations.
- Parikshit Hooda, Raft Consensus Algorithm, GeeksforGeeks, 2018

Replication: What and Why

- Replication = An object has identical copies, each maintained by a separate server
  - Copies are called "replicas"

- Why replication?
  - Fault-tolerance: With *k* replicas of each object, can tolerate failure of any (*k-1*) servers in the system
  - Load balancing: Spread read/write operations out over the *k* replicas => load lowered by a factor of *k* compared to a single replica
  - Replication => Higher Availability

Availability

- If each server is down a fraction *f* of the time
  - Server's failure probability

- With no replication, availability of object

  = Probability that single copy is up

  = $(1 - f)$

- With *k* replicas, availability of object

  Probability that at least one replicas is up

  = 1 – Probability that all replicas are down

  = $(1 - f^k)$

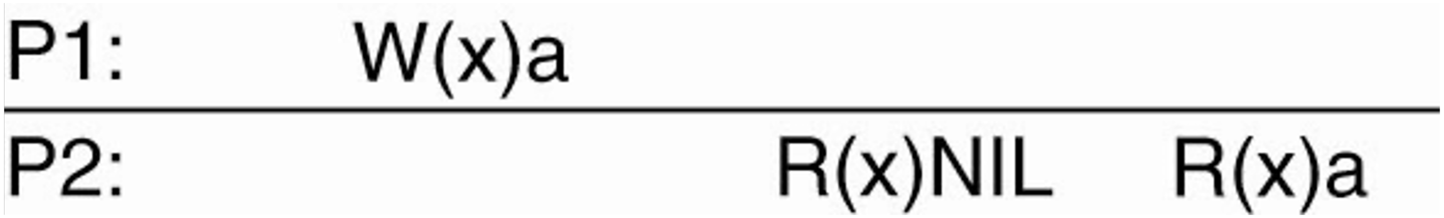
Transactions and Replication

- One-copy serializability
  - *A concurrent execution of transactions in a replicated database is one-copy-serializable if it is equivalent to a serial execution of these transactions over a single logical copy of the database.*
  - (Or) The effect of transactions performed by clients on replicated objects should be the same as if they had been performed one at a time on a single set of objects (i.e., 1 replica per object).

- In a non-replicated system, transactions appear to be performed one at a time in some order.
  - Correctness means serial equivalence of transactions

- When objects are replicated, transaction systems for correctness need one-copy serializability

Sequential Consistency (1)



P1: W(x)a

P2: R(x)NIL R(x)a

- Behavior of two processes operating on the same data item. The horizontal axis is time.
- P1: Writes “W” value a to variable “x”
- P2: Reads ` NIL’ from “x” first and then ` a’

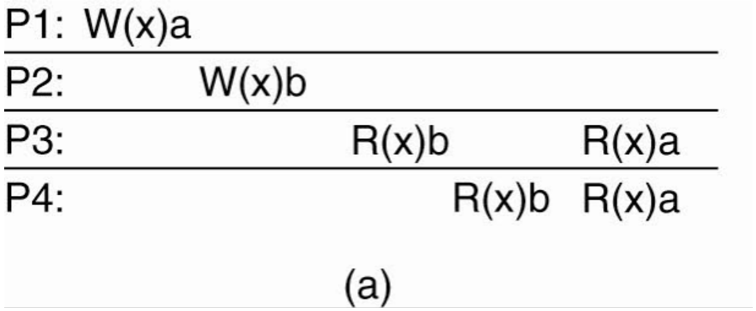
Tanenbaum & Van Steen, Distributed Systems: Principles and Paradigms, 2e, (c) 2007 Prentice-Hall, Inc. All rights reserved. 0-13-239227-5

Sequential Consistency (2)

- A data store is sequentially consistent when:

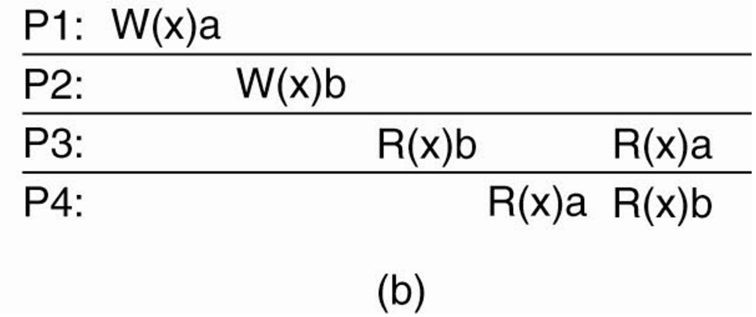
- The result of any execution is the same as if the (read and write) operations by all processes on the data store …
  - Were executed in some sequential order and …
  - the operations of each individual process appear …
    - in this sequence
    - in the order specified by its program.

Sequential Consistency (3)



P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)b	R(x)a

(a)



P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)a	R(x)b

(b)

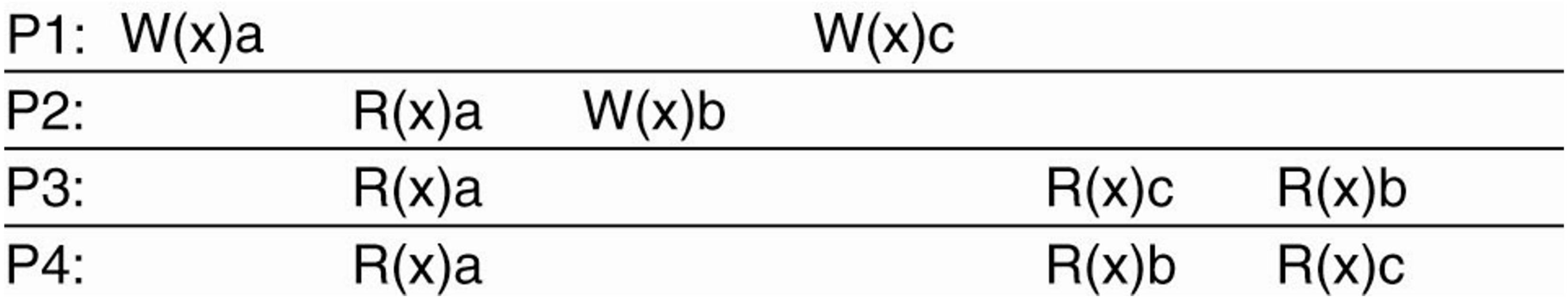
(a) A sequentially consistent data store.

(b) A data store that is not sequentially consistent.

Causal Consistency (1)

- For a data store to be considered causally consistent, it is necessary that the store obeys the following condition:
  - Writes that are potentially causally related …
      - must be seen by all processes
      - in the same order.
  - Concurrent writes …
    - may be seen in a different order
    - on different machines.

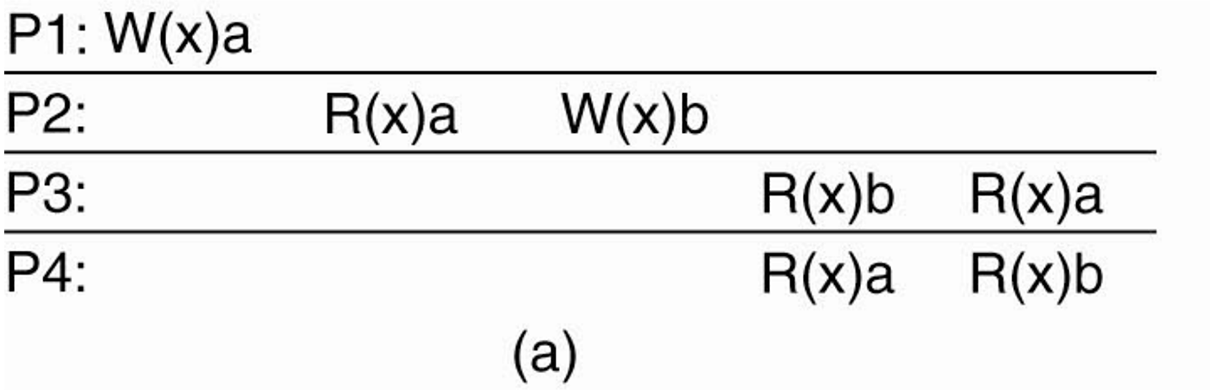
Causal Consistency (2)



P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

Causal Consistency (3)



P1:	W(x)a				
P2:		R(x)a	W(x)b		
P3:				R(x)b	R(x)a
P4:				R(x)a	R(x)b

(a)

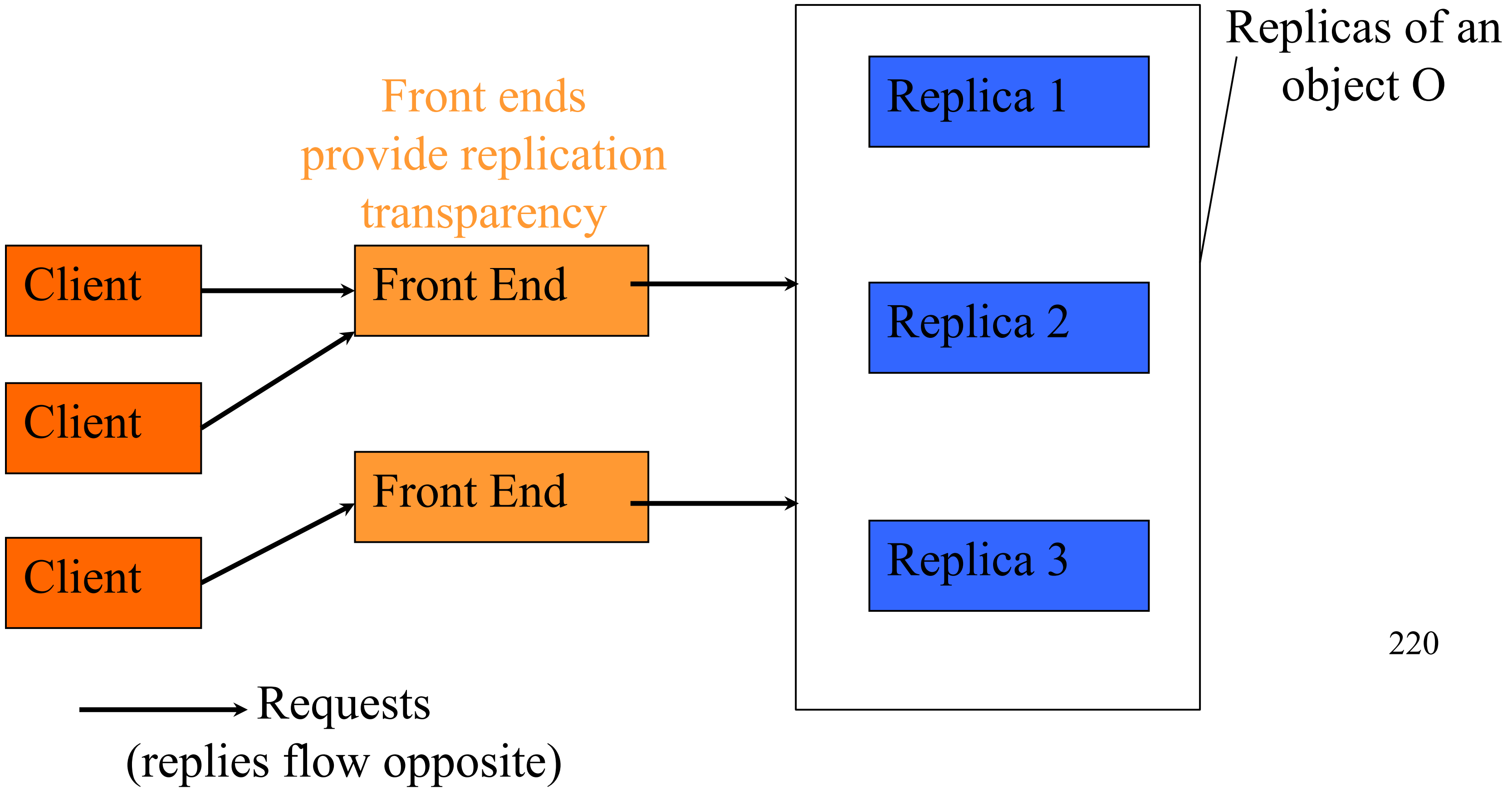
- A violation of a causally-consistent store.

Challenge?

- Maintain two properties:

1. Replication Transparency
   - A client ought not to be aware of multiple copies of objects existing on the server side
2. Replication Consistency
   - All clients see single consistent copy of data, in spite of replication
   - For transactions, guarantee ACID

Replication Transparency



Replication Consistency

- Two ways to forward updates from front-ends (FEs) to replica group
  - Passive Replication: uses a primary replica (master)
  - Active Replication: treats all replicas identically

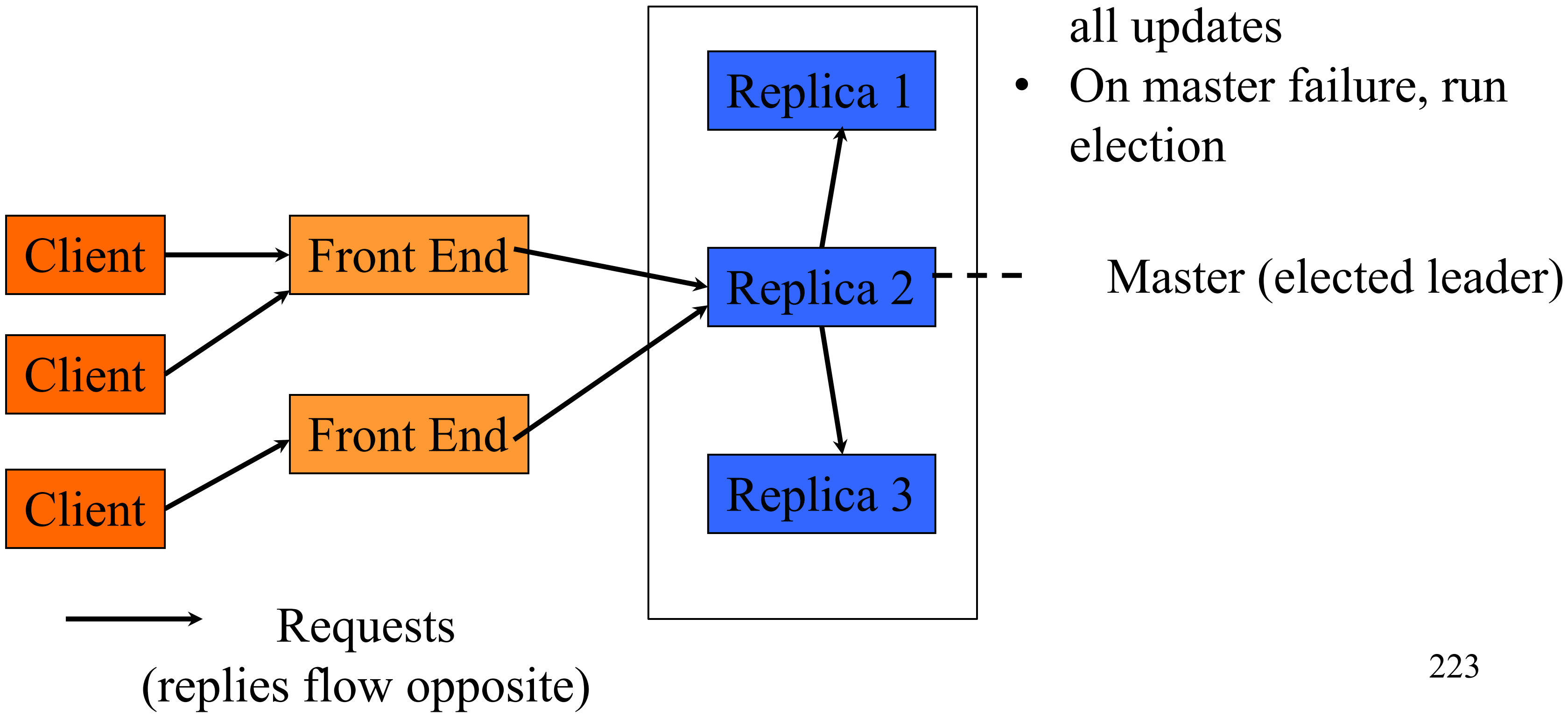
- Both approaches use the concept of "Replicated State Machines"
  - Each replica's code runs the same state machine
  - *Multiple copies of the same State Machine begun in the Start state, and receiving the same Inputs in the same order will arrive at the same State having generated the same Outputs.* [Schneider 1990]

Replication Model

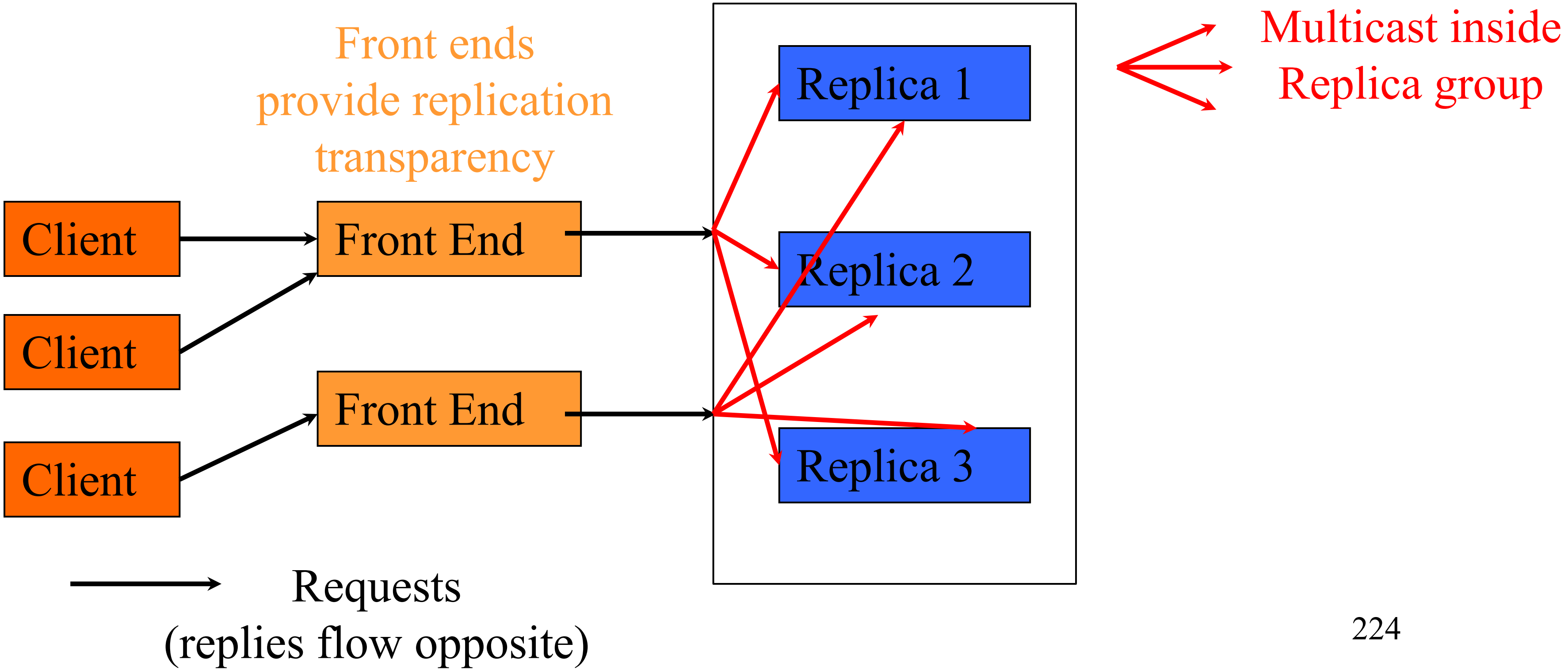
- Passive Replication (Primary-backup)
  - Operations handled by primary, it streams copies to backup(s)
  - Replicas are “passive”, i.e. follow the primary
  - Good: Simple protocol. Bad: Clients must participate in recovery.
- Active Replication (Quorum consensus)
  - Designed to have fast response time even under failures
  - Replicas are “active” - participate in protocol; there is no master, per se.
  - Good: Clients don’t even see the failures. Bad: More complex.

Passive Replication

- Master => total ordering of all updates
- On master failure, run election



Active Replication

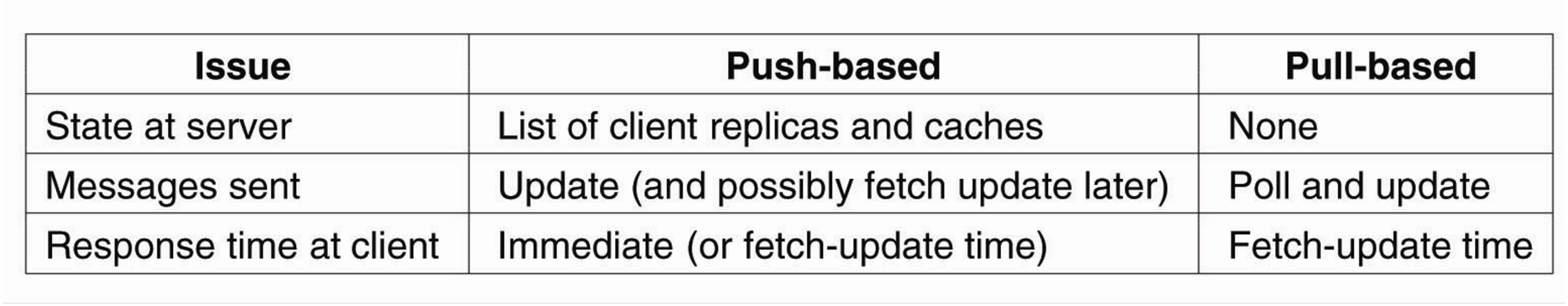


Front ends provide replication transparency

Multicast inside Replica group

Requests (replies flow opposite)

When to Replicate: Pull versus Push Protocols

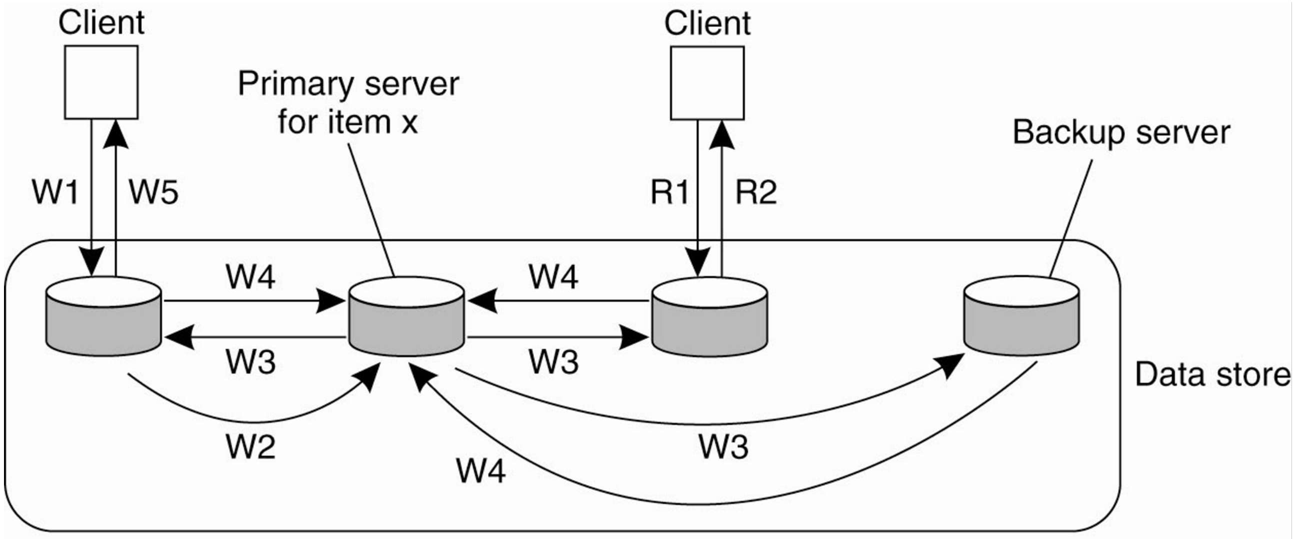


Issue	Push-based	Pull-based
State at server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Comparison between push- and pull-based protocols in the case of multiple-client, single-server systems.

- Pull Based: Replicas/Clients poll for updates (caches)
- Push Based: Server pushes updates (stateful)

Remote-Write PB Protocol

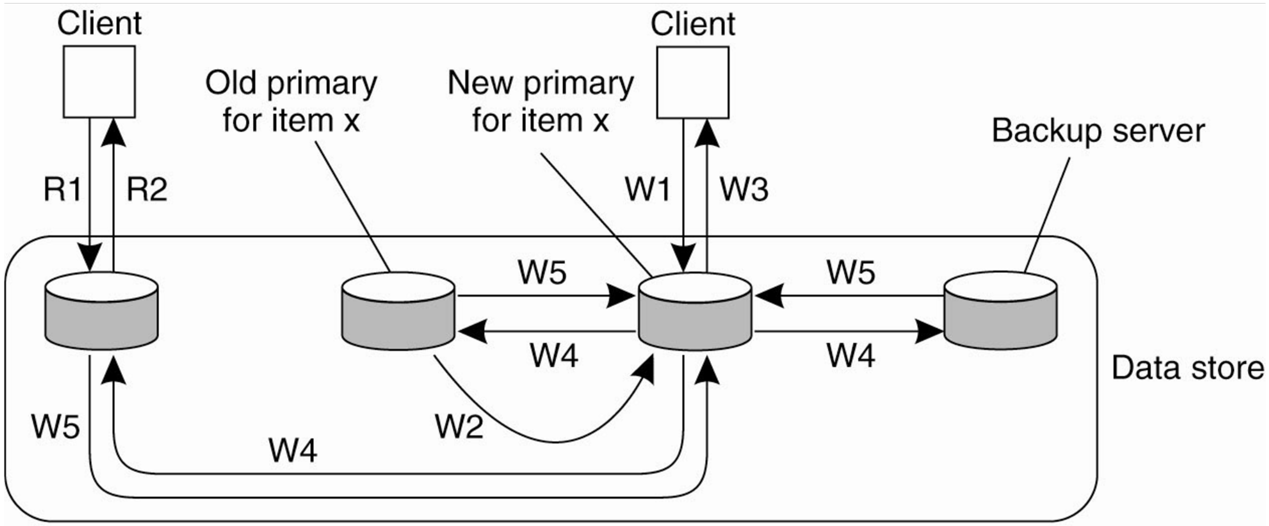


W1. Write request
W2. Forward request to primary
W3. Tell backups to update
W4. Acknowledge update
W5. Acknowledge write completed

R1. Read request
R2. Response to read

Updates are blocking, although non-blocking possible

Local-Write P-B Protocol



W1. Write request
W2. Move item x to new primary
W3. Acknowledge write completed
W4. Tell backups to update
W5. Acknowledge update

R1. Read request
R2. Response to read

Primary migrates to the process wanting to process update
For performance, use non-blocking op.

What does this scheme remind you of?

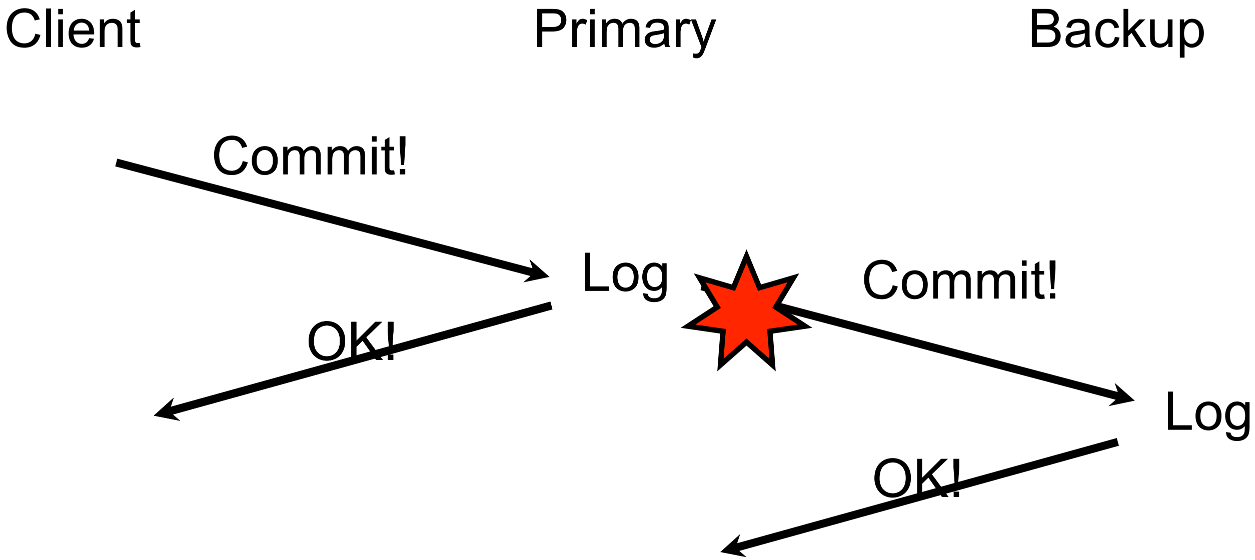
Primary-Backup

- Note: If you don't care about strong consistency (e.g., the "mail read" flag), you can reply to client *before* reaching agreement with backups (sometimes called "asynchronous replication").
- This looks cool. What's the problem?
  - What do we do if a replica has failed?
  - We wait... how long? Until it's marked dead.
  - Primary-backup has a strong dependency on the failure detector
- This is OK for some services, not OK for others
- Advantage: With N servers, can tolerate loss of N-1 copies

Implementing P-B

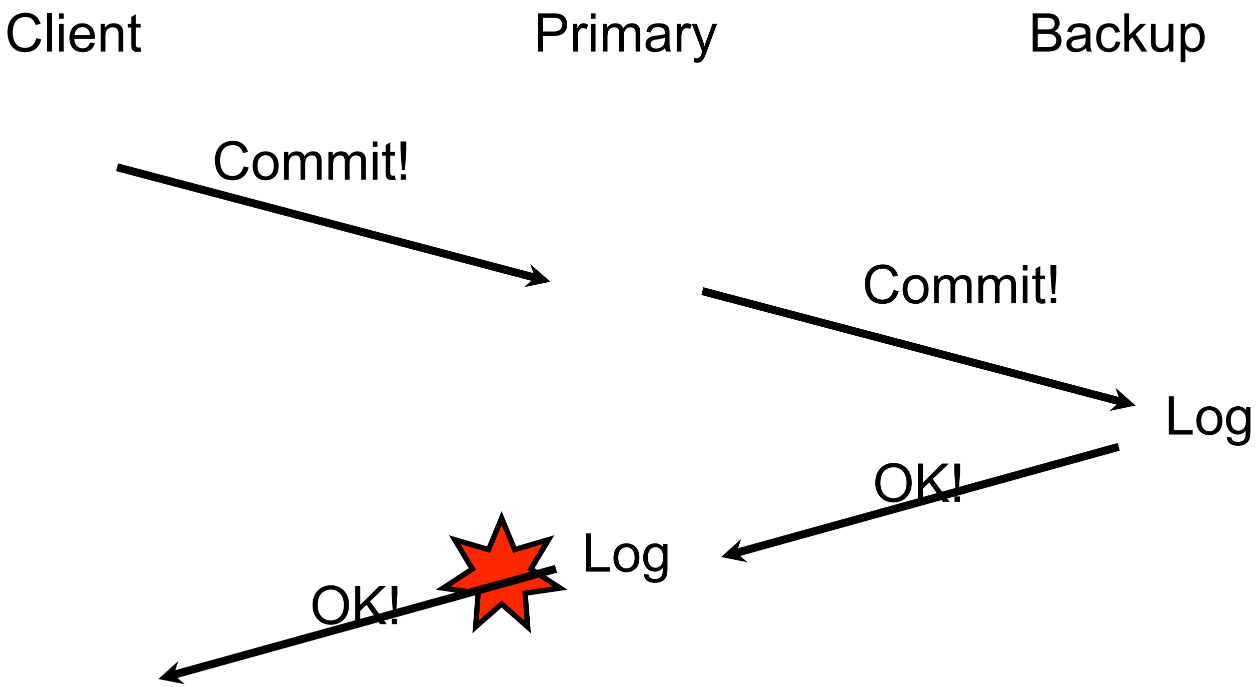
- Remember logging? :-)
- Common technique for replication in databases and filesystem-like things: Stream the log to the backup. They don’t have to actually apply the changes before replying, just make the log durable.
- You have to replay the log before you can be online again, but it’s pretty cheap.

p-b: Did it happen?



Failure here:
Commit logged only at primary
Primary dies? Client must re-send to backup

p-b: Happened twice



Failure here:
Commit logged at backup
Primary dies? Client must check with backup

(Seems like at-most-once / at-least-once... :)

Problems with p-b

- Not a great solution if you want very tight response time even when something has failed: Must wait for failure detector
- For that, *quorum* based schemes are used
- As name implies, different result: To handle *f* failures, must have 2*f* + 1 replicas (so that a majority is still alive)
- Also, for replicated-write => write to all replica's not just one.

Prof Srinivasan Seshan

School of Computer Science
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213-3891

http://www.cs.cmu.edu/~srini/15-440-all/2016.Fall/lectures/10-Logging-Recovery.ppt

Recovery Strategies

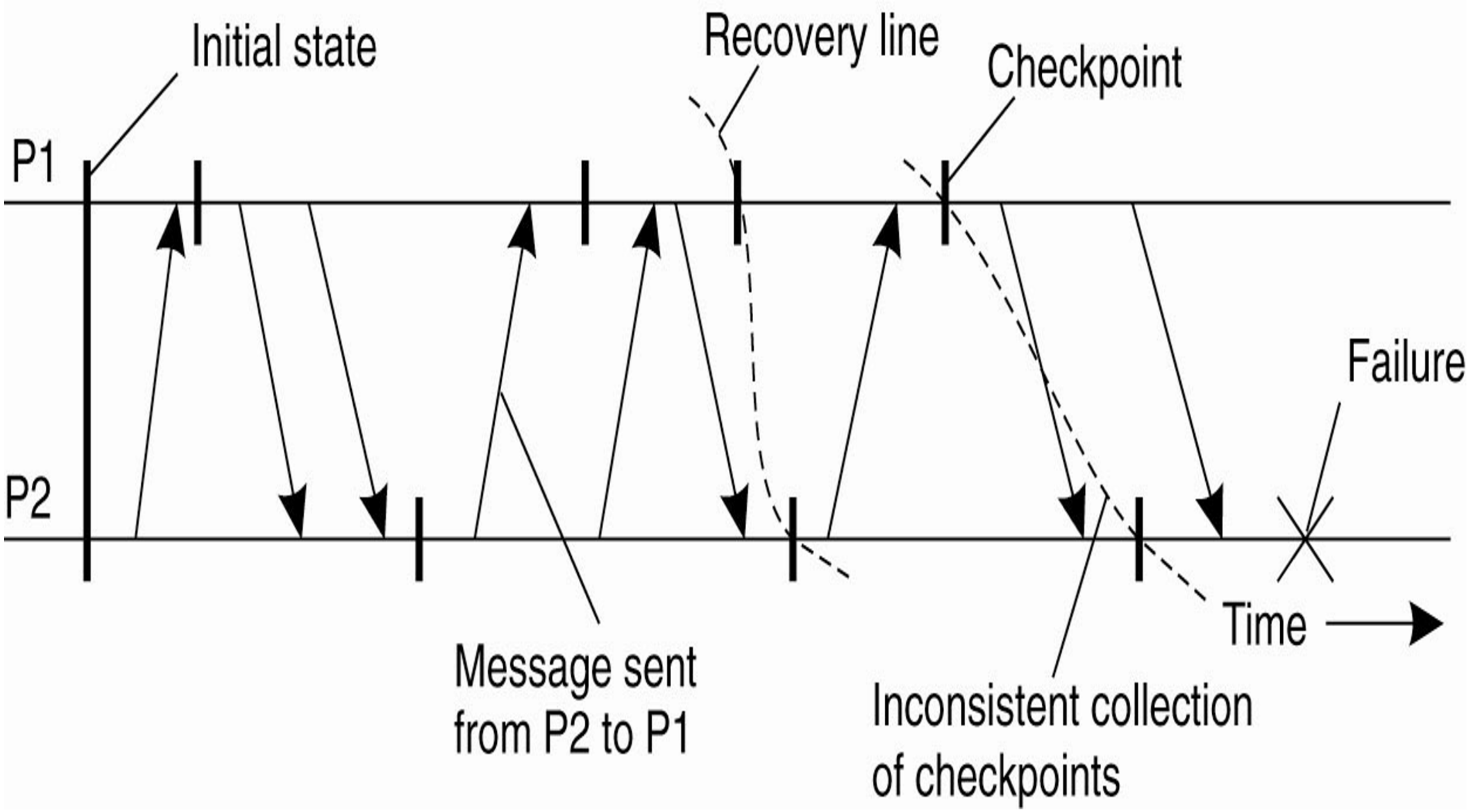
- When a failure occurs, we need to bring the system into an error free state (recovery). This is fundamental to Fault Tolerance.

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing. Example?
    - Packet retransmit in case of lost packet
2. **Forward Recovery**: bring the system into a correct new state, from which it can then continue to execute. Example?
    - Erasure coding, (n,k) where k < n <= 2k

Forward and Backward Recovery

- **Major disadvantage of Backward Recovery**:
  - Checkpointing can be very expensive (especially when errors are very rare).
  - [Despite the cost, backward recovery is implemented more often. The "logging" of information can be thought of as a type of checkpointing.].
- **Major disadvantage of Forward Recovery**:
  - In order to work, all potential errors need to be accounted for *up-front*.
  - When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

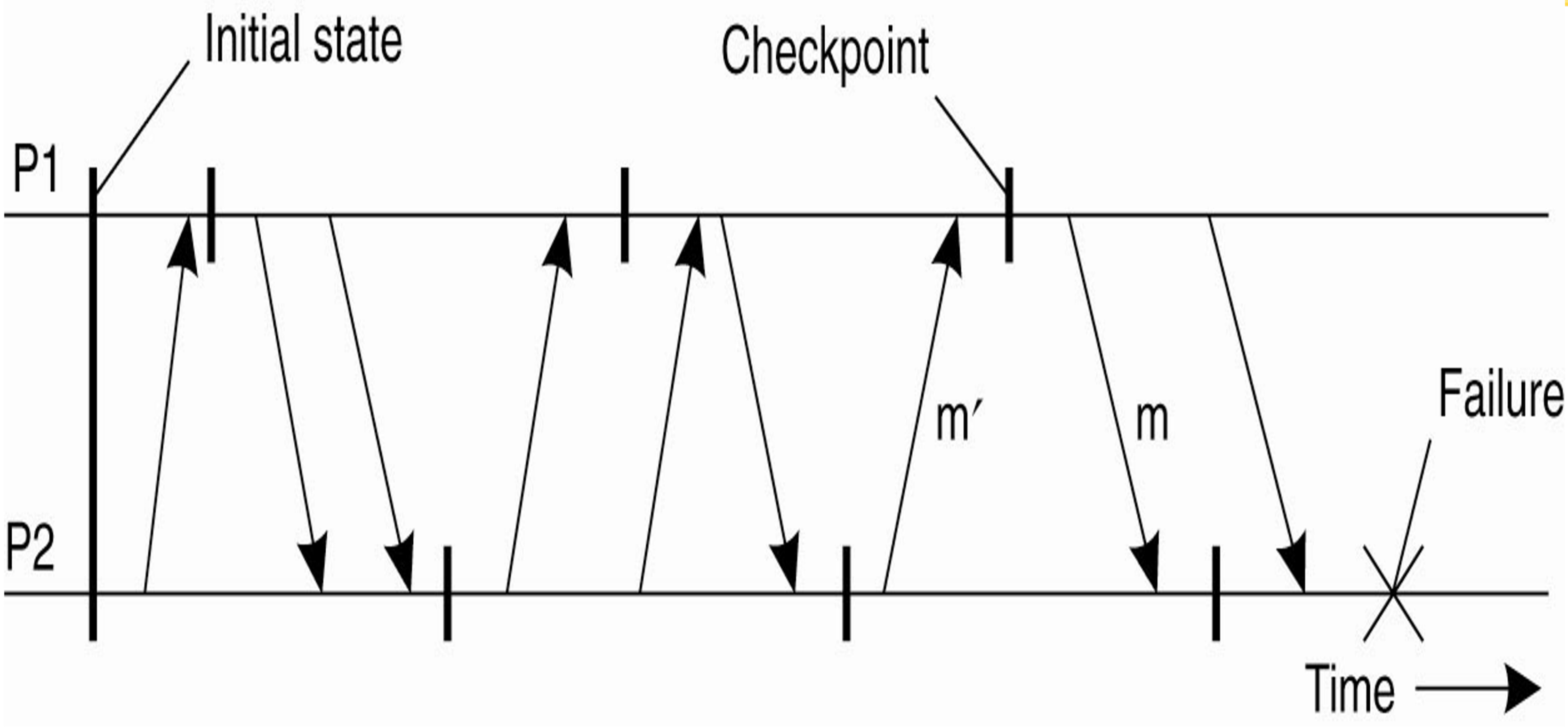
Checkpointing


Initial state
Recovery line
Checkpoint
P1
P2
Failure
Time
Message sent from P2 to P1
Inconsistent collection of checkpoints

A recovery line to detect the correct distributed snapshot

This becomes challenging if checkpoints are un-coordinated

Independent Checkpointing



The domino effect – Cascaded rollback

P2 crashes, roll back, but 2 checkpoints inconsistent (P2 shows m received, but P1 does not show m sent)

Coordinated Checkpointing

- Key idea: each process takes a checkpoint after a globally coordinated action. (why is this good?)

- Simple Solution: 2-phase blocking protocol
  - Co-ordinator multicast *checkpoint_REQUEST* message
  - Participants receive message, takes a checkpoint, stops sending (application) messages, and sends back *checkpoint_ACK*
  - Once all participants ACK, coordinator sends *checkpoint_DONE* to allow blocked processes to go on

- Optimization: consider only processes that depend on the recovery of the coordinator (those it sent a message since last checkpoint)

Logging

- …in the presence of failures
  - Machines can crash. Disk Contents (OK), Memory (volatile), Machines don't misbehave
  - Networks are flaky, packet loss, handle using timeouts

- If we store database state in memory, a crash will cause loss of "Durability".
- May violate atomicity, i.e. recover such that uncommited transactions COMMIT or ABORT.
- General idea: store enough information to disk to determine global state (in the form of a LOG)

Challenges:

- Disk performance is poor (vs memory)
  - Cannot save all transactions to disk
  - Memory typically several orders of magnitude faster

- Writing to disk to handle arbitrary crash is hard
  - Several reasons, but HDDs and SSDs have buffers

- Same general idea: store enough data on disk so as to recover to a valid state after a crash:
  - Shadow pages and Write-ahead Logging (WAL)

Shadow Paging Vs WAL

- Shadow Pages
  - Provide Atomicity and Durability, “page” = unit of storage
  - Idea: When writing a page, make a “shadow” copy
    - No references from other pages, edit easily!
  - ABORT: discard shadow page
  - COMMIT: Make shadow page “real”. Update pointers to data on this page from other pages (recursive). Can be done atomically
  - Essentially “copy-on-write” to avoid in-place page update

Shadow Paging vs WAL

- Write-Ahead-Logging
  - Provide Atomicity and Durability
  - Idea: create a log recording every update to database
  - Updates considered reliable when stored on disk
  - Updated versions are kept in memory (page cache)
  - Logs typically store both REDO and UNDO operations
  - After a crash, recover by replaying log entries to reconstruct correct state
  - WAL is more common, fewer disk operations, transactions considered committed once log written.

Write-Ahead Logging

- View as sequence of entries, sequential number
  - Log-Sequence Number (LSN)
  - Database: fixed size PAGES, storage at page level
- Pages on disk, some also in memory (page cache)
  - "Dirty pages": page in memory differs from one on disk
- Reconstruction of global consistent state
  - Log files + disk contents + (page cache)
- Logs consist of sequence of records
  - `Begin LSN, TID #Begin TXN`
  - `End LSN, TID, PrevLSN #Finish TXN (abort or commit)`
  - `Update LSN, TID, PrevLSN, pageID, offset, old value, new value`

Write-Ahead Logging

- Logs consist of sequence of records
  - To record an update to state
  - Update LSN, TID, PrevLSN, pageID, offset, old value, new value
  - PrevLSN forms a backward chain of operations for each TID
  - Storing “old” and “new” values allow REDO operations to bring a page up to date, or UNDO an update reverting to an earlier version

- Transaction Table (TT): All TXNS not written to disk
  - Including Seq Num of the last log entry they caused

- Dirty Page Table (DPT): all dirty pages in memory
  - Modified pages, but not written back to disk.

Write-Ahead-Logging

- Commit a transaction
  - Log file up to date until commit entry
  - Don't update actual disk pages, log file has information
  - Keep "tail" of log file in memory => not commits
  - If the tail gets wiped out (crash), then partially executed transactions will lost. Can still recover to reliable state

- Abort a transaction
  - Locate last entry from TT, undo all updates so far
  - Use PrevLSN to revert in-memory pages to start of TXN
  - If page on disk needs undo, wait (come back to this)

Recovery using WAL – 3 passes

- Analysis Pass
  - Reconstruct TT and DPT (from start or last checkpoint)
  - Get copies of all pages at the start
- Recovery Pass (redo pass)
  - Replay log forward, make updates to all dirty pages
  - Bring everything to a state at the time of the crash
- Undo Pass
  - Replay log file backward, revert any changes made by transactions that had not committed (use PrevLSN)
  - For each write Compensation Log Record (CLR)
  - Once you reach BEGIN TXN, write an END TXN entry

WAL can be integrated with 2PC

- WAL can integrate with 2PC
  - Have additional log entries that capture 2PC operation
  - **Coordinator:** Include list of participants
  - **Participant:** Indicates coordinator
  - Votes to commit or abort
  - Indication from coordinator to Commit/Abort

Optimizing WAL

- As described earlier:
  - Replay operations back to the beginning of time
  - Log file would be kept forever, (entire Database)

- In practice, we can do better with CHECKPOINT
  - Periodically save DPT, TT
  - Store any dirty pages to disk, indicate in LOG file
  - Prune initial portion of log file: All transactions upto checkpoint have been committed or aborted.

Summary

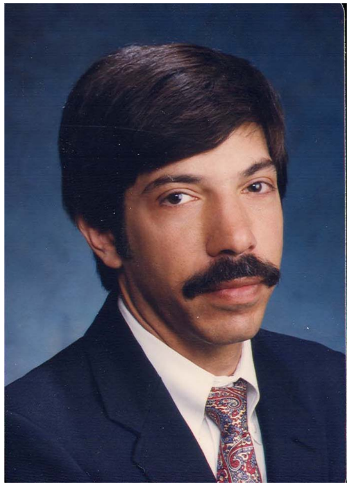
- Fault Tolerance – Backward recovery using checkpointing, both Independent and coordinated
- Fault Tolerance –Recovery using Write-Ahead-Logging, balances the overhead of checkpointing and ability to recover to a consistent state

State Machine Replication

Drew Zagieboylo

Authors

- Fred Schneider



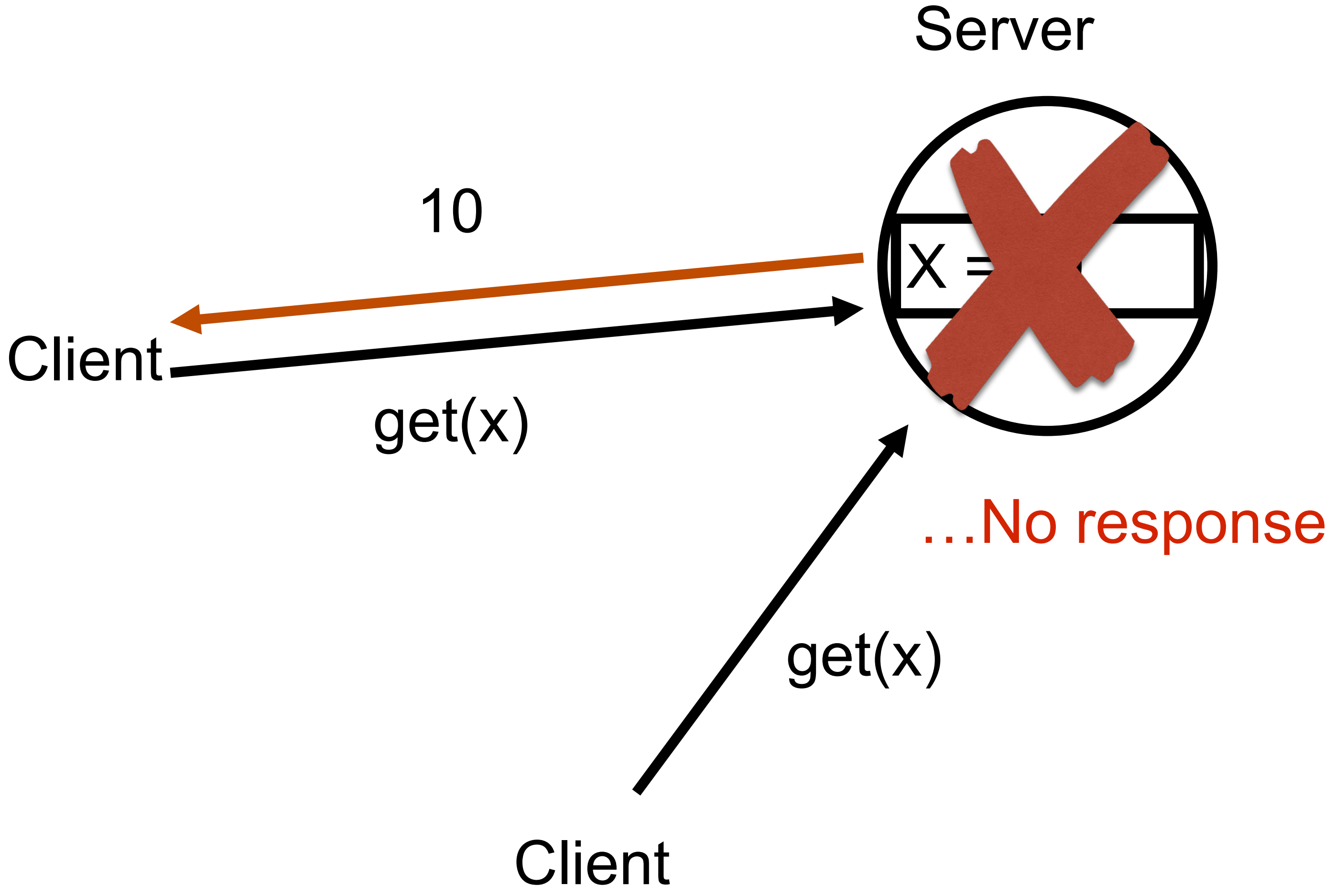
Takeaways

- Can represent ***deterministic*** distributed system as *Replicated State Machine*
- Each replica reaches the same conclusion about the system ***independently***
- Key examples of *distributed algorithms* that generically implement *SMR*
- Formalizes notions of fault-tolerance in *SMR*

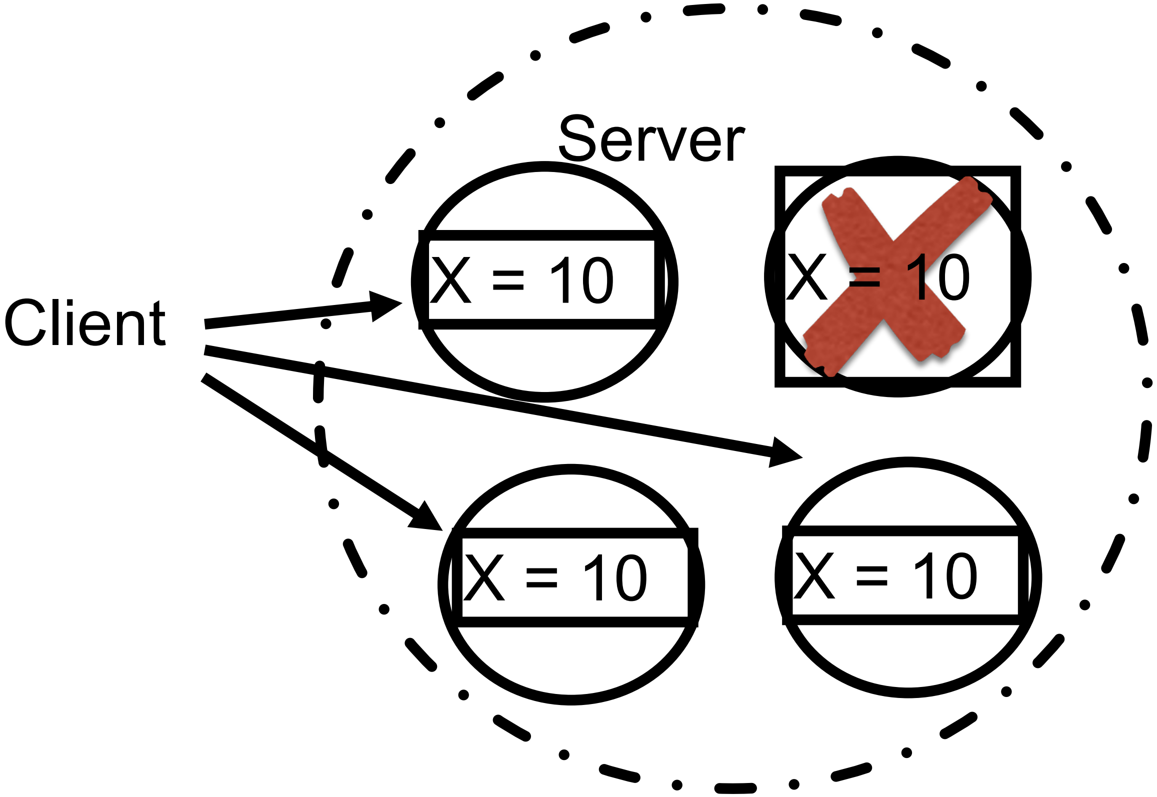
Outline

- Motivation
- State Machine Replication
- Implementation
- Fault Tolerance Requirements
- An Example - Chain Replication
- Evaluation

Motivation



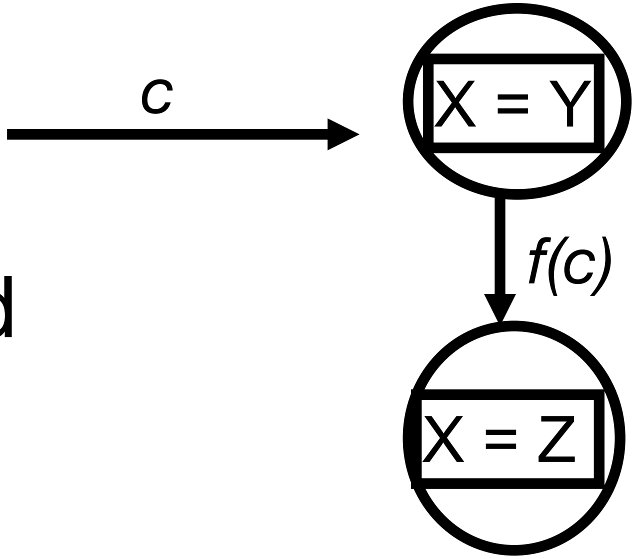
Motivation



Motivation

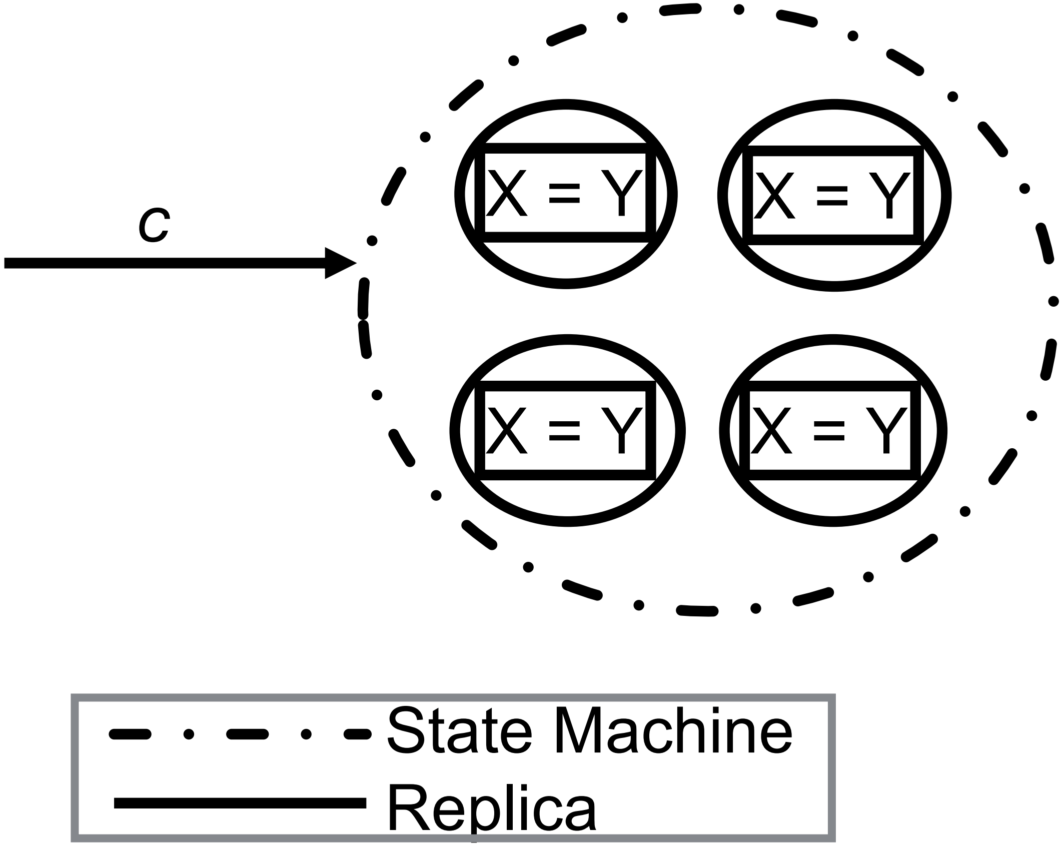
- Need replication for fault tolerance
- What happens in these scenarios without replication?
  - Storage - Disk Failure
  - Webservice - Network failure
- Be able to reason about failure tolerance
  - How badly can things go wrong and have our system continue to function?

State Machines



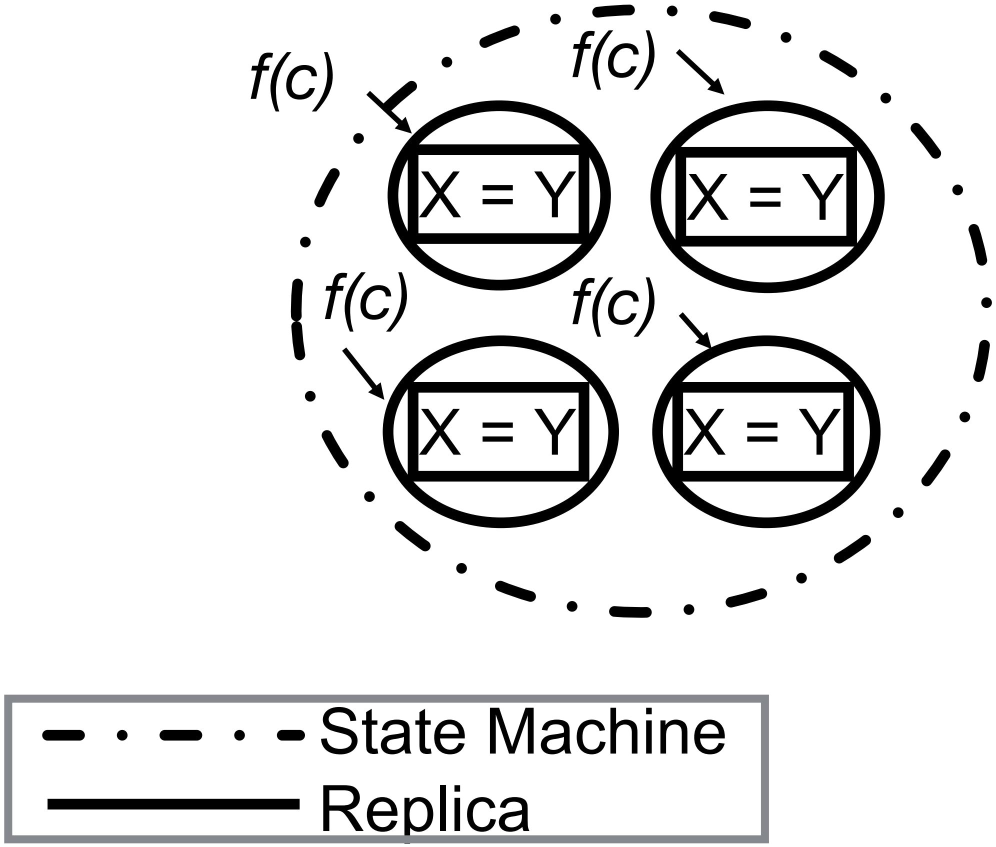
- c is a Command
- f is a Transition Function

State Machine Replication (SMR)



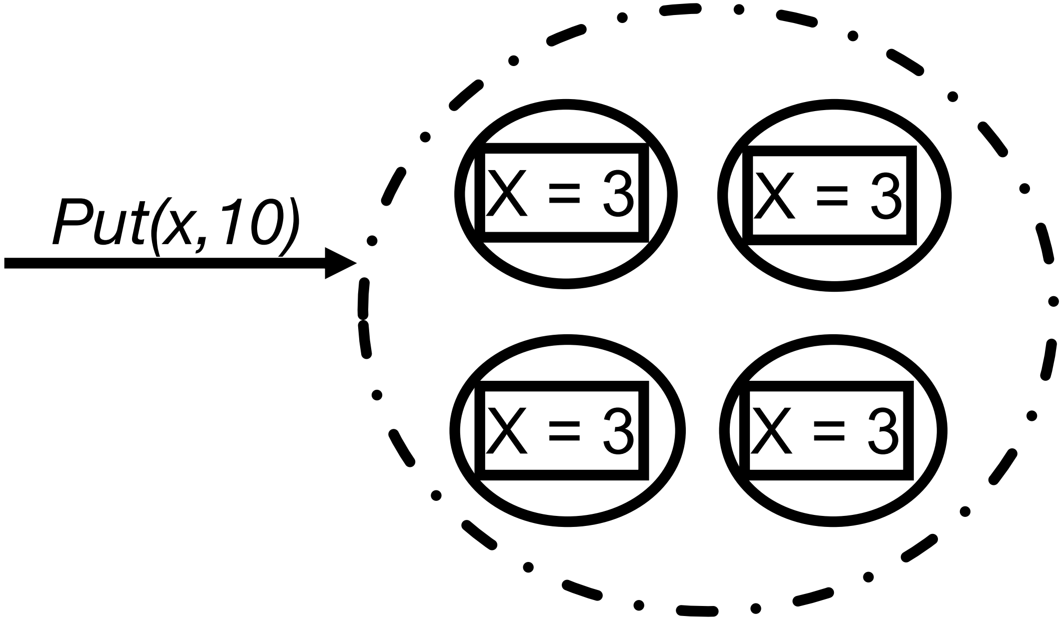
- The *State Machine Approach* to a fault tolerant distributed system
- Keep around *N* copies of the state machine

State Machine Replication (SMR)

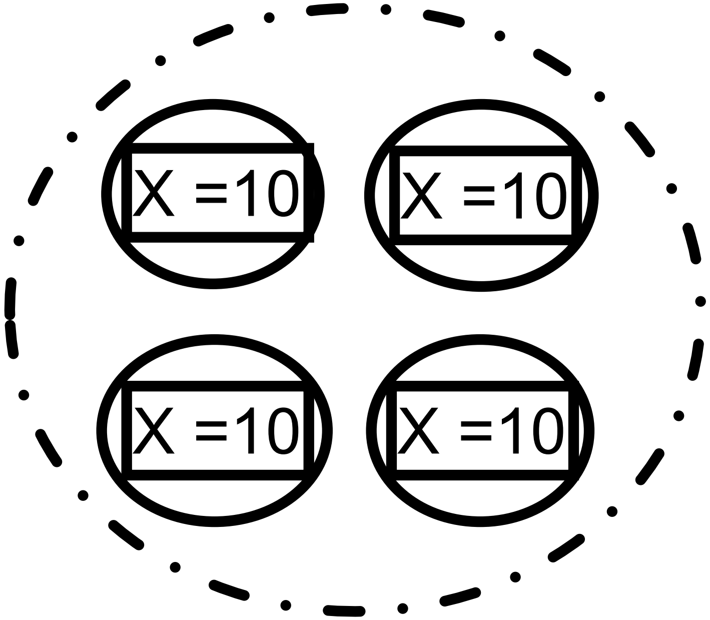


- The *State Machine Approach* to a fault tolerant distributed system
- Keep around *N* copies of the state machine

SMR Requirements

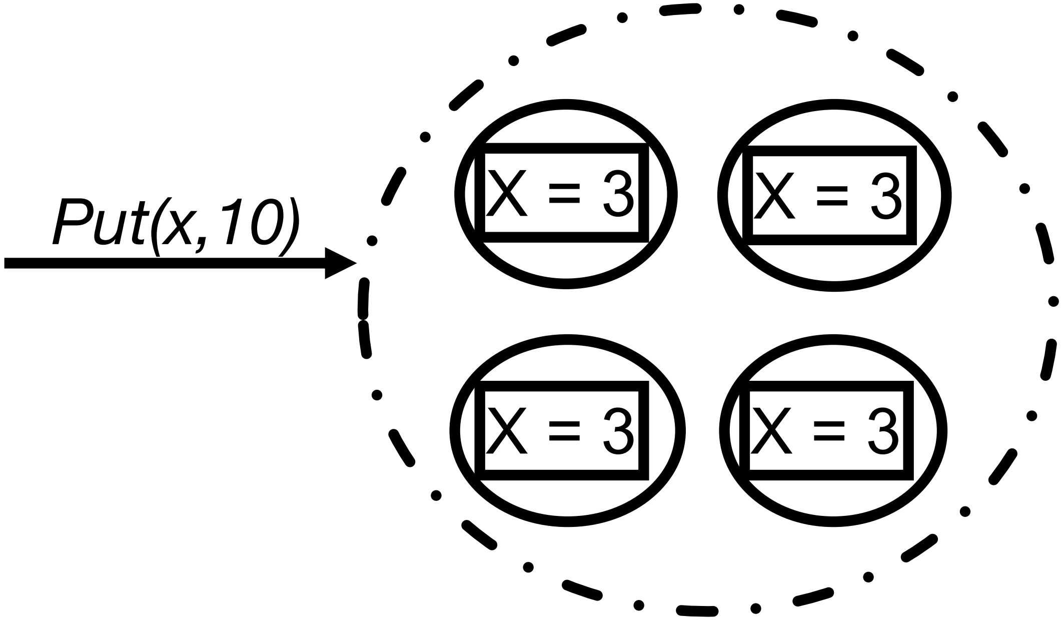


SMR Requirements

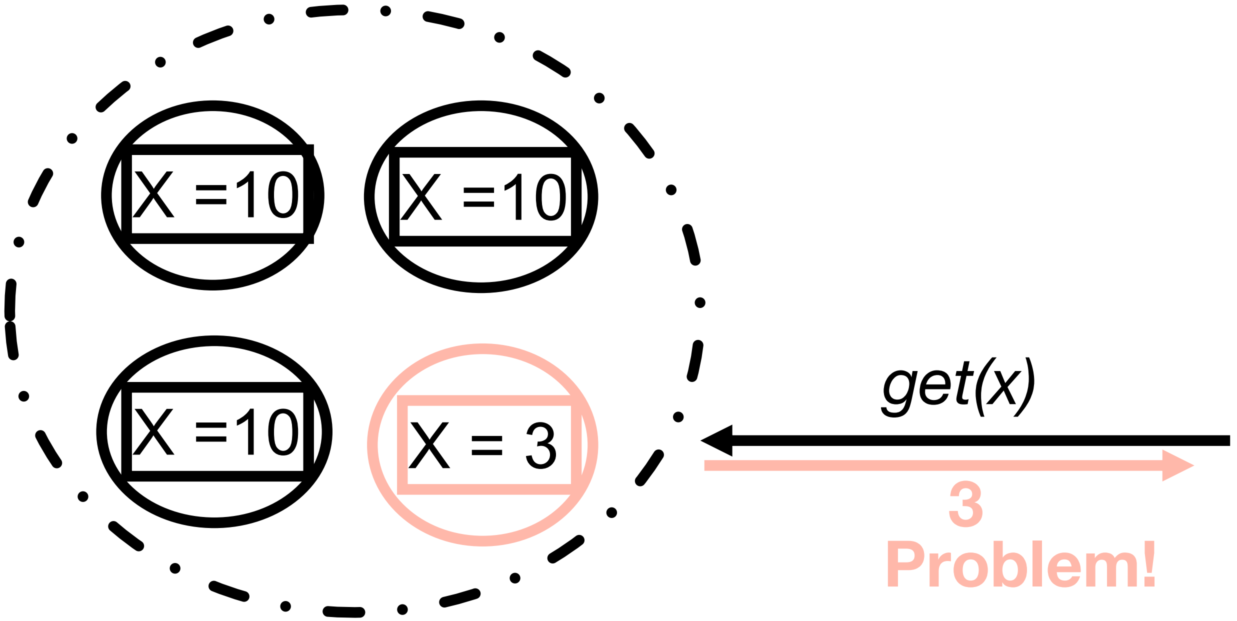


Great!

SMR Requirements

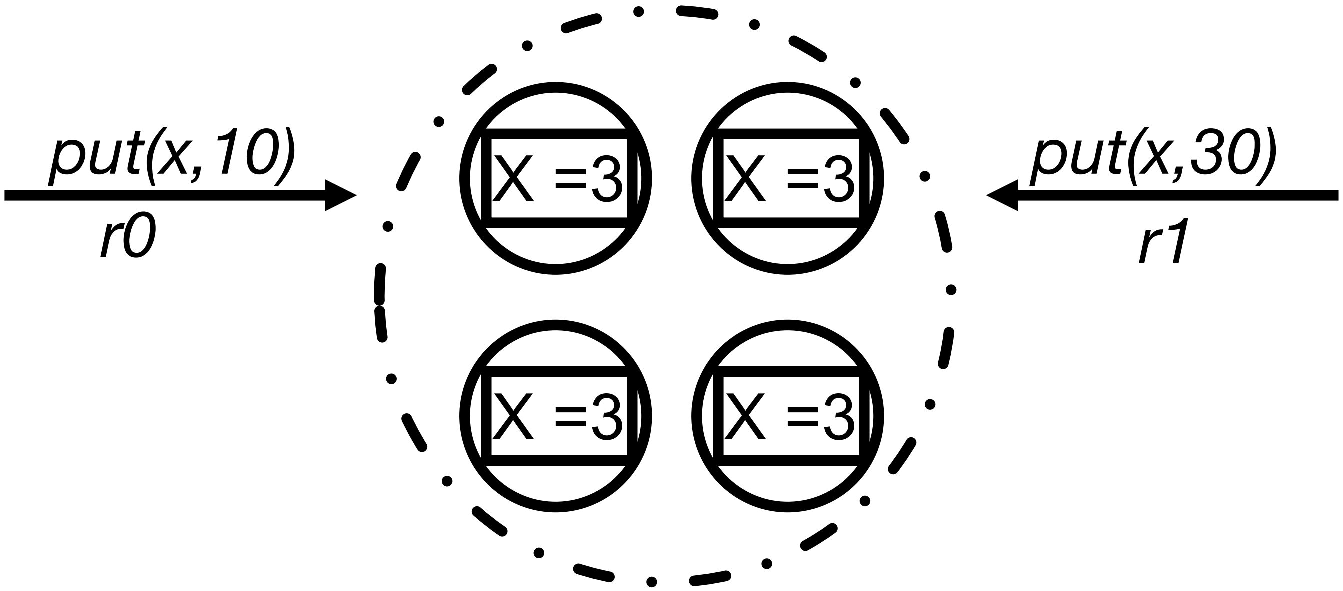


SMR Requirements

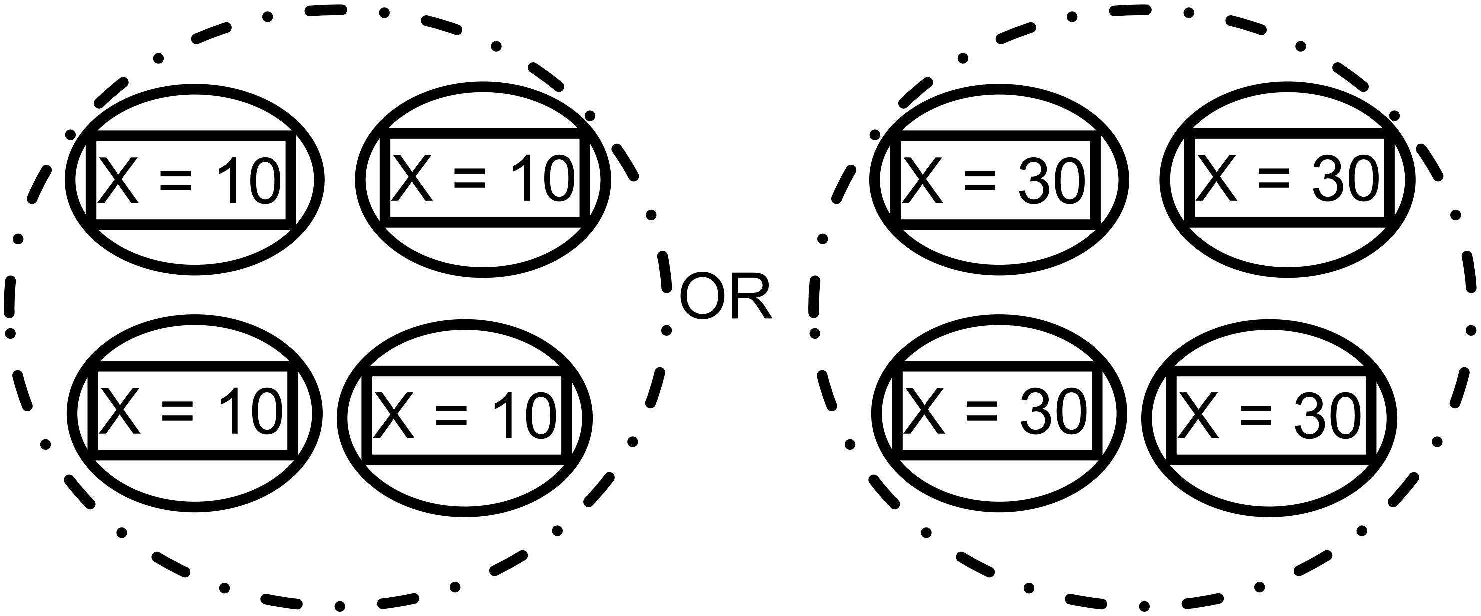


- Replicas need to ***agree*** on the which requests have been handled

SMR Requirements



SMR Requirements



X = 10

X = 10

X = 10

X = 10

OR

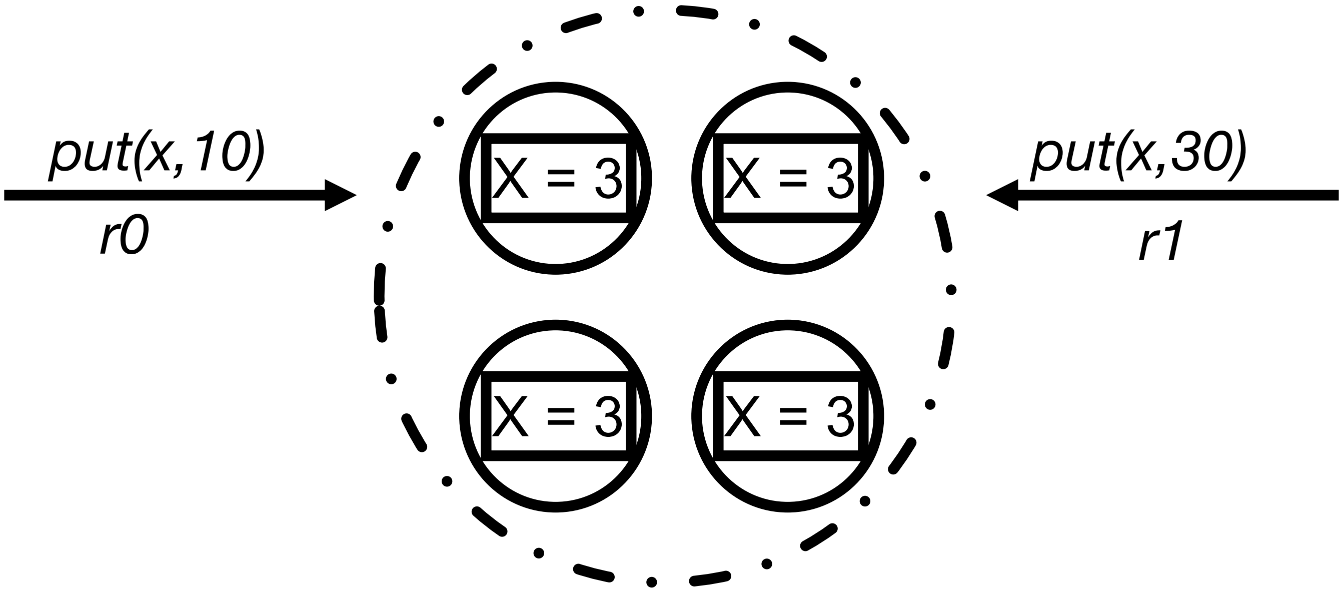
X = 30

X = 30

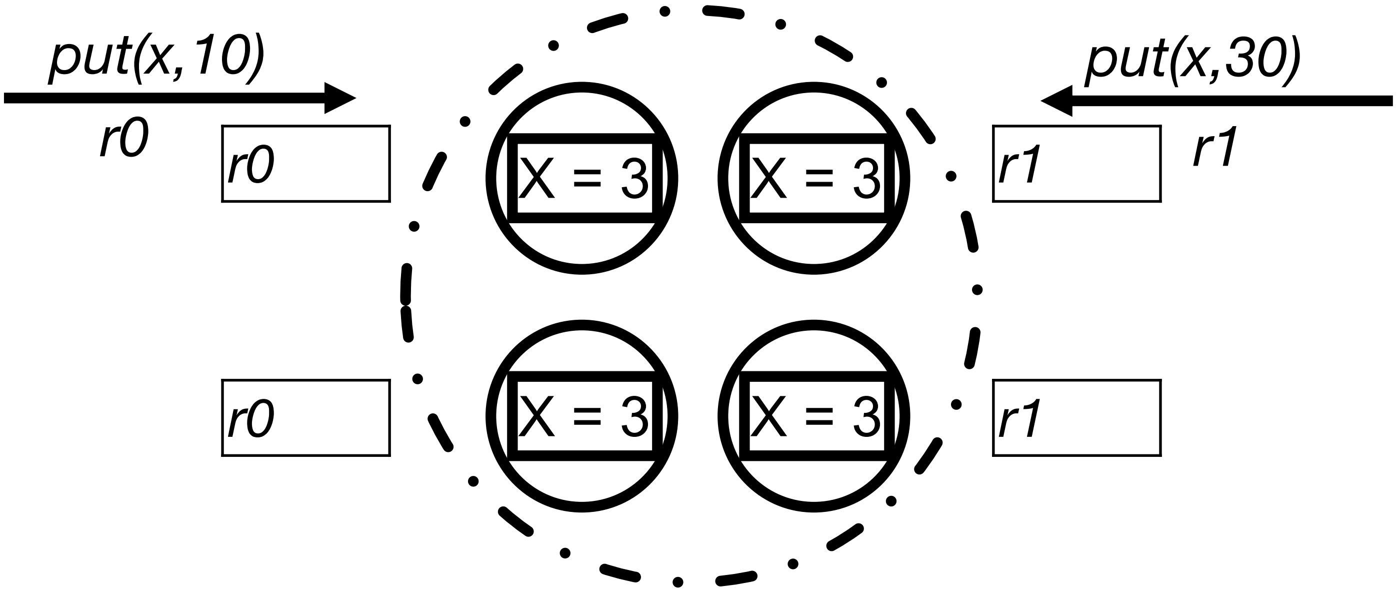
X = 30

X = 30

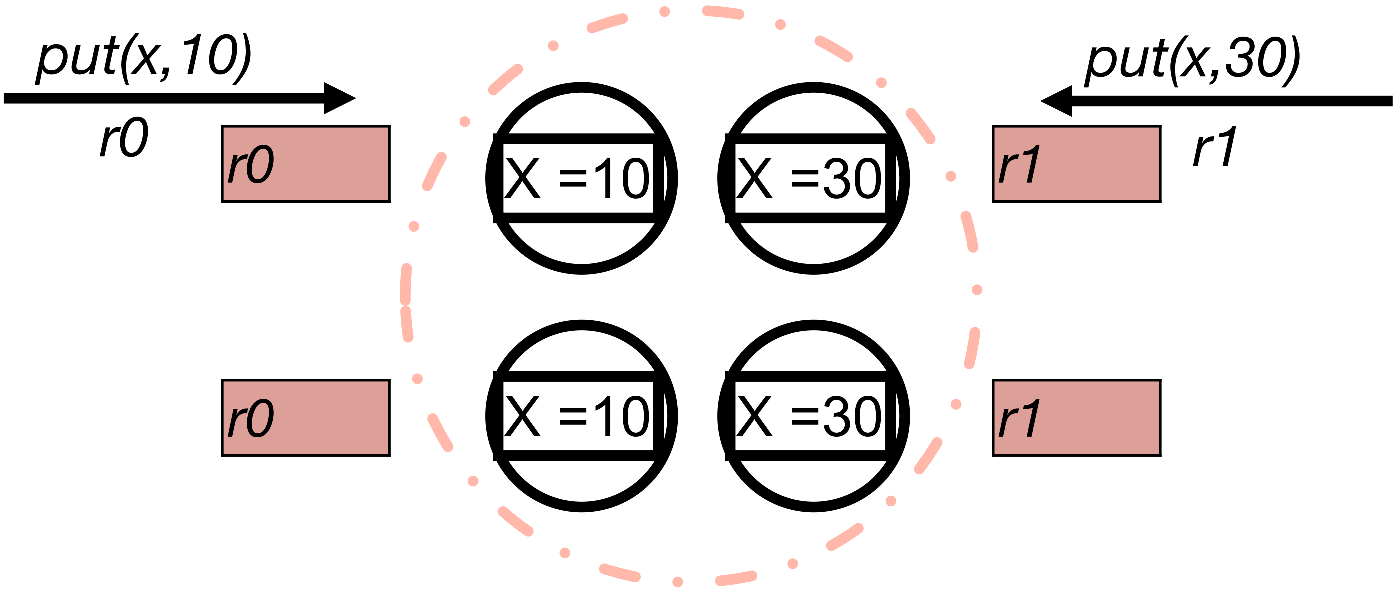
SMR Requirements



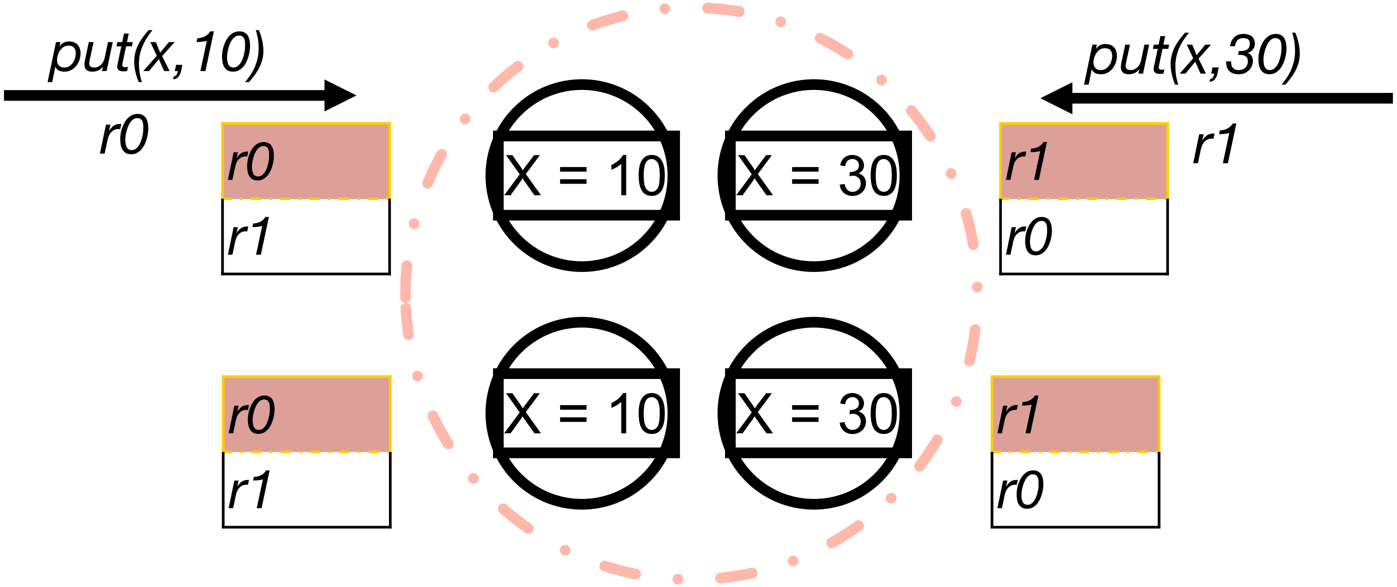
SMR Requirements



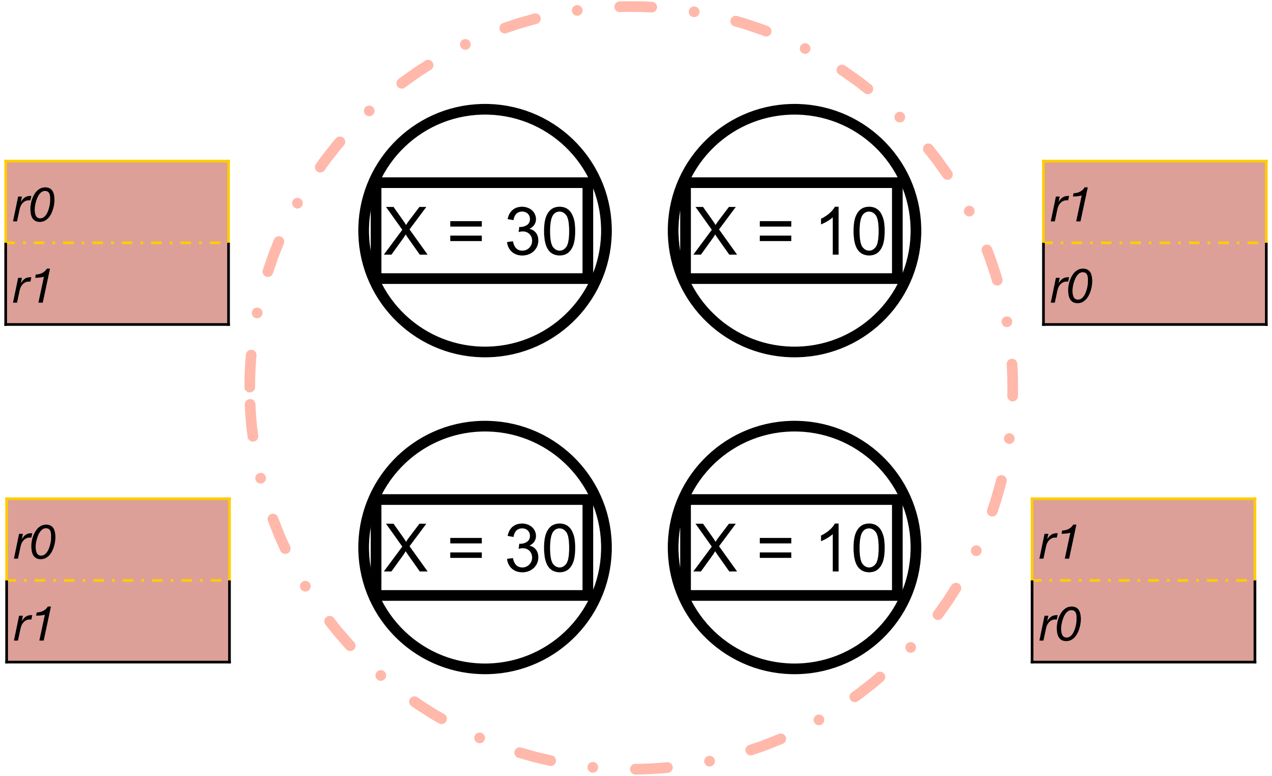
SMR Requirements



SMR Requirements



SMR Requirements



- Replicas need to handle requests in the same ***order***

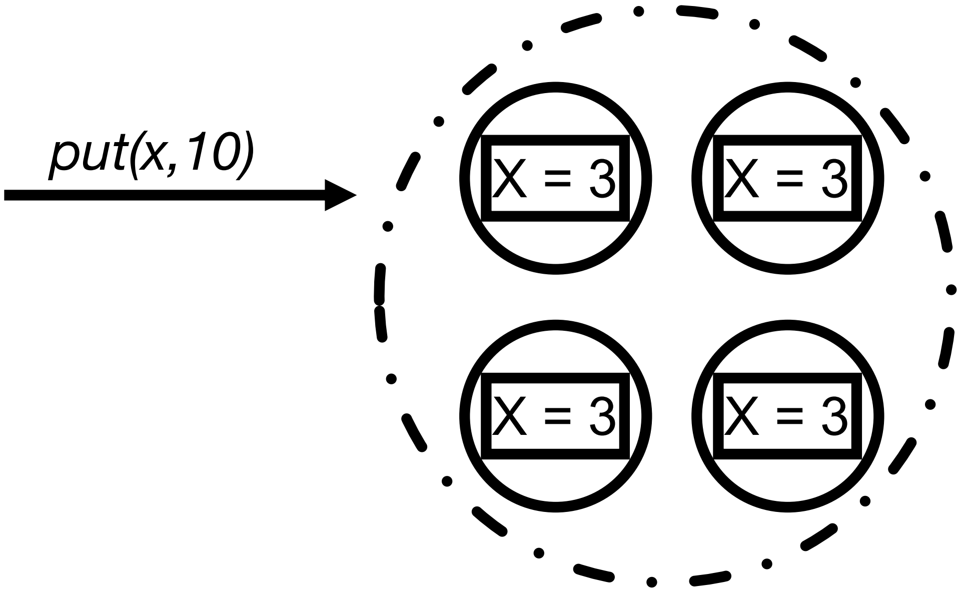
SMR

- All non faulty servers need:
  - Agreement
    - Every replica needs to accept the same set of requests
  - Order
    - All replicas process requests in the same relative order

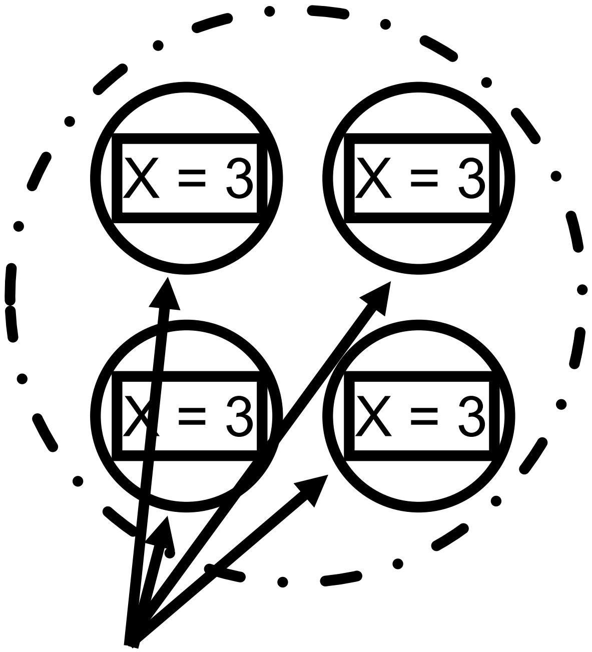
Implementation

- Agreement
  - Someone proposes a request; if that person is nonfaulty all servers will accept that request
  - Strong and Dolev [1983] and Schneider [1984] for implementations
  - Client or Server can propose the request

SMR Implementation



SMR Implementation



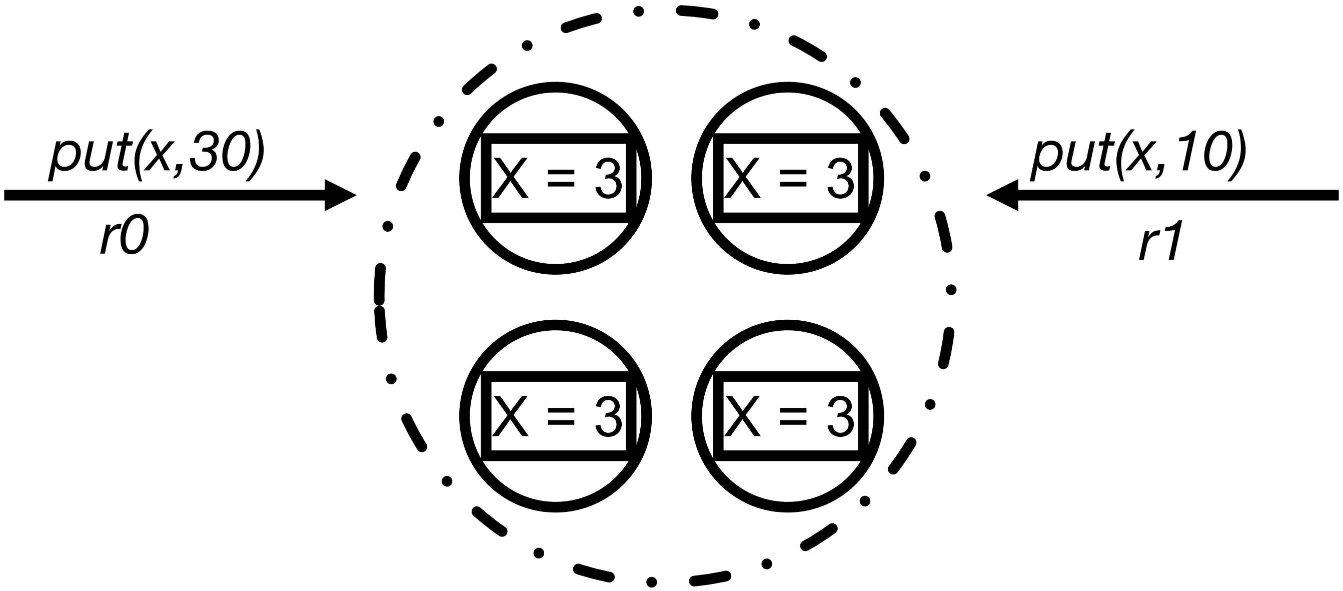
put(x,10)

Non-faulty Transmitter

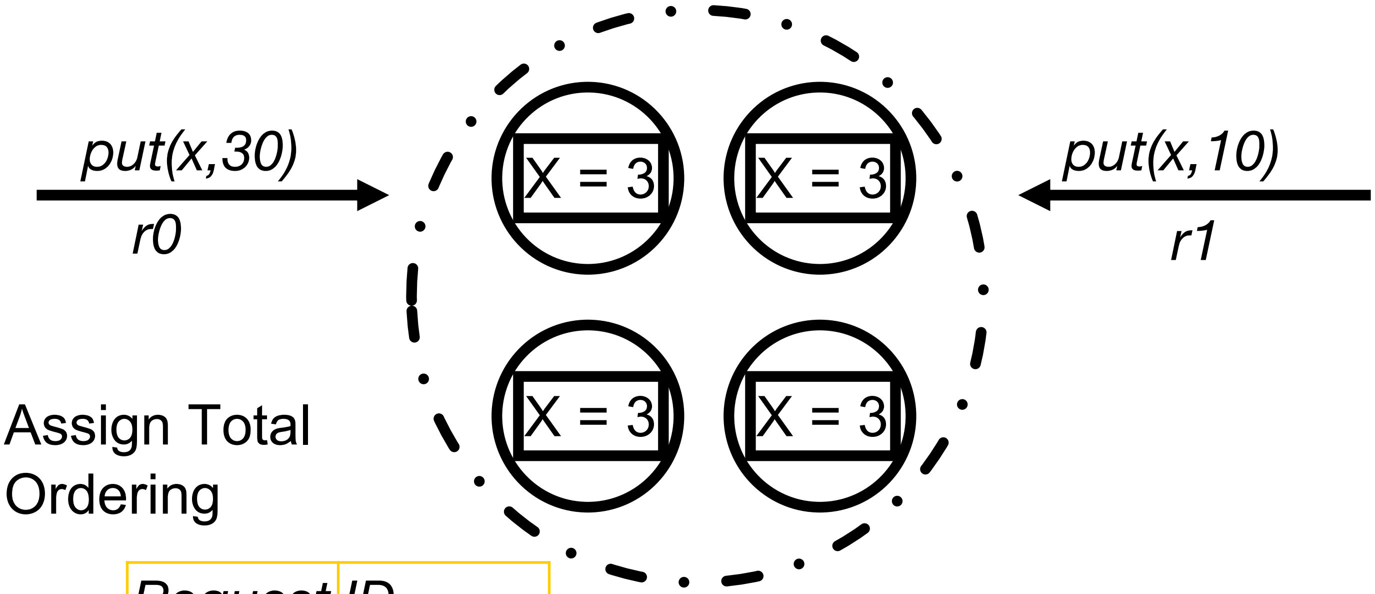
Implementation

- Order
  - Assign unique ids to requests, process them in ascending order.
  - How do we assign unique ids in a distributed system?
  - How do we know when every replica has processed a given request?

SMR Requirements



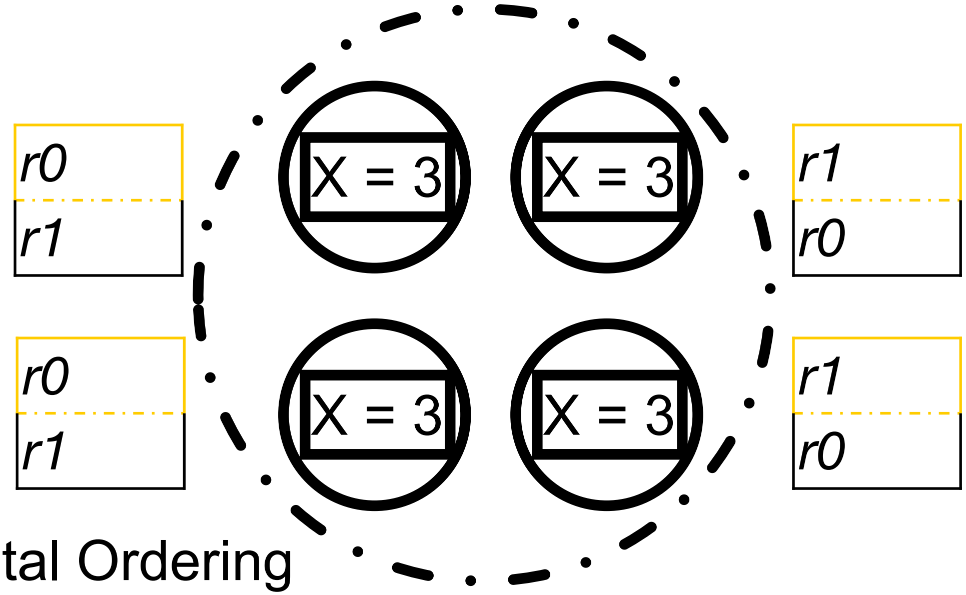
SMR Requirements



Assign Total Ordering

Request	*ID*
r0	*1*
r1	*2*

SMR Requirements



r0
r1

r0
r1

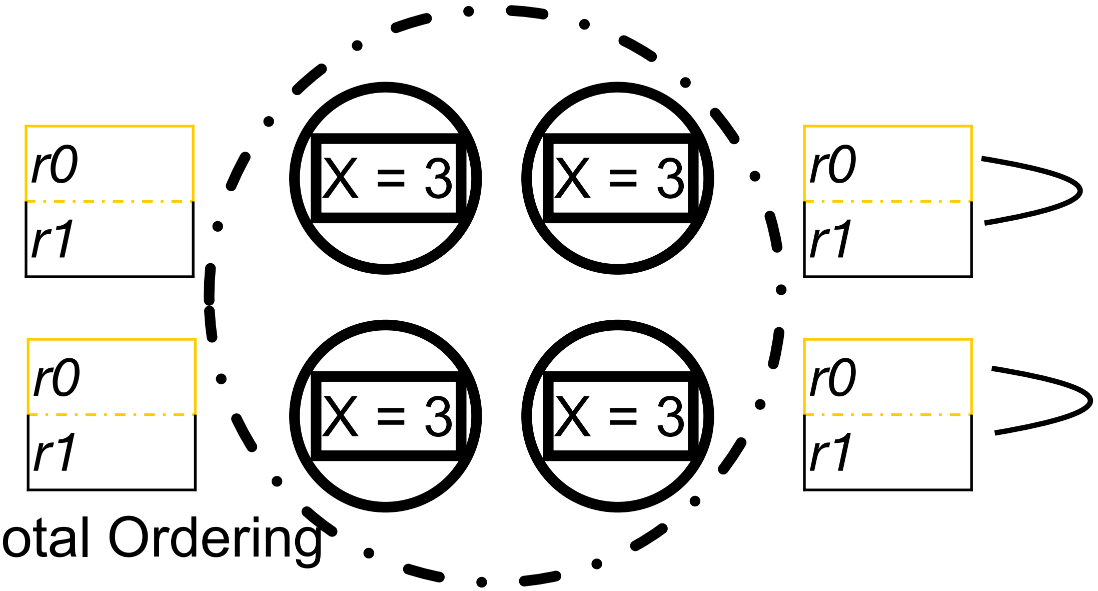
r1
r0

r1
r0

Assign Total Ordering

Request	*ID*
r0	*1*
r1	*2*

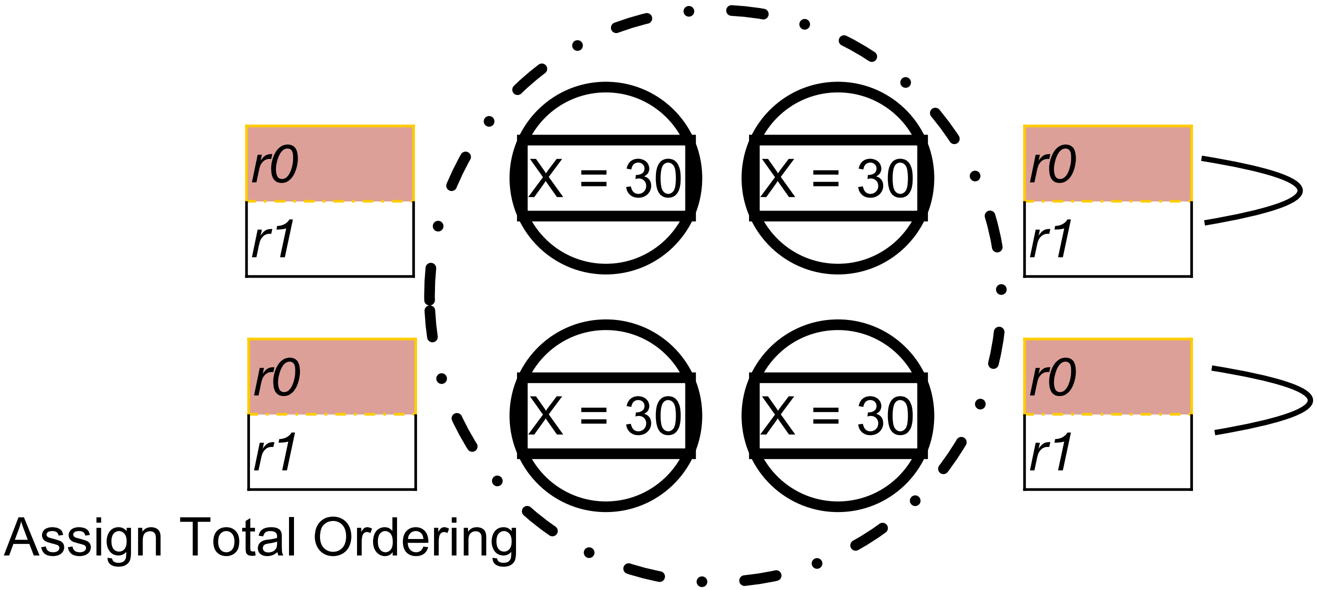
SMR Requirements



Assign Total Ordering

Request	*ID*
r0	*1*
r1	*2*

SMR Requirements

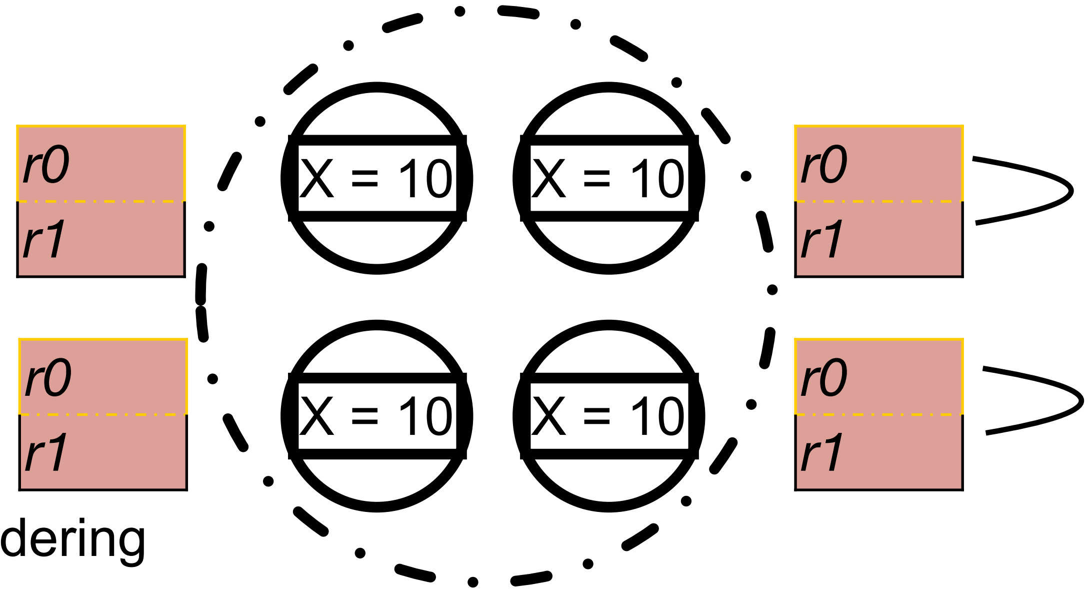


Assign Total Ordering

Request	*ID*
r0	*1*
r1	*2*

r0 is now stable!

SMR Requirements



Assign Total Ordering

Request	*ID*
r0	*1*
r1	*2*

r0 is now stable!
r1 is now stable!

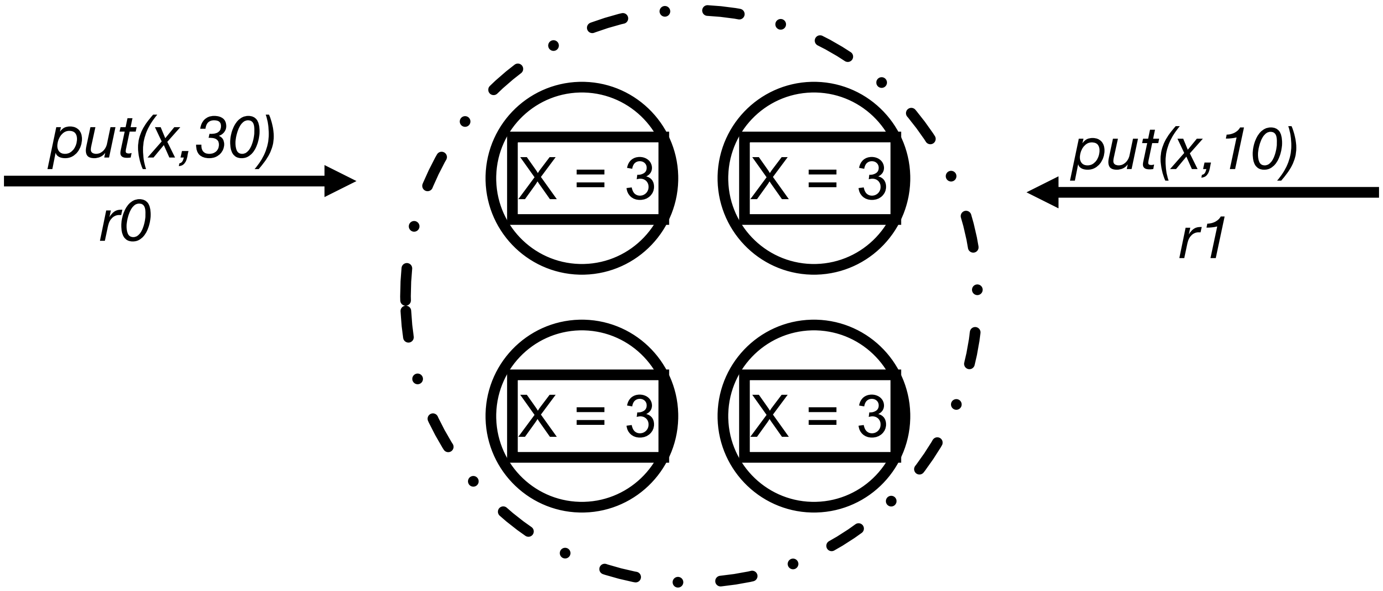
Implementation
Client Generated IDs

- Order via Clocks (Client timestamps represent IDs)
  - Logical Clocks
  - Synchronized Clocks
- Ideas from last week! [Lamport 1978]

Implementation
Replica Generated IDs

- 2 Phase ID generation
  - Every Replica proposes a *candidate*
  - One candidate is chosen and agreed upon by all replicas

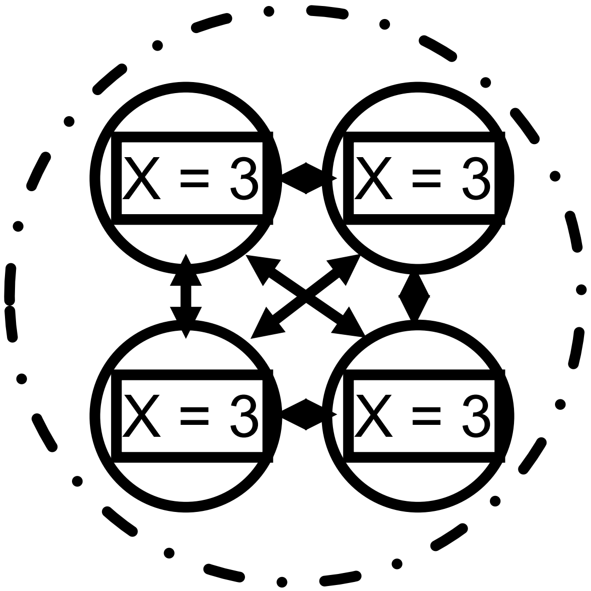
Replica ID Generation



Replica ID Generation

Req.	CUID	UID
r0	1.1	
r1	2.1	

Req.	CUID	UID
r0	1.2	
r1	2.2	



Req.	CUID	UID
r1	1.3	
r0	2.3	

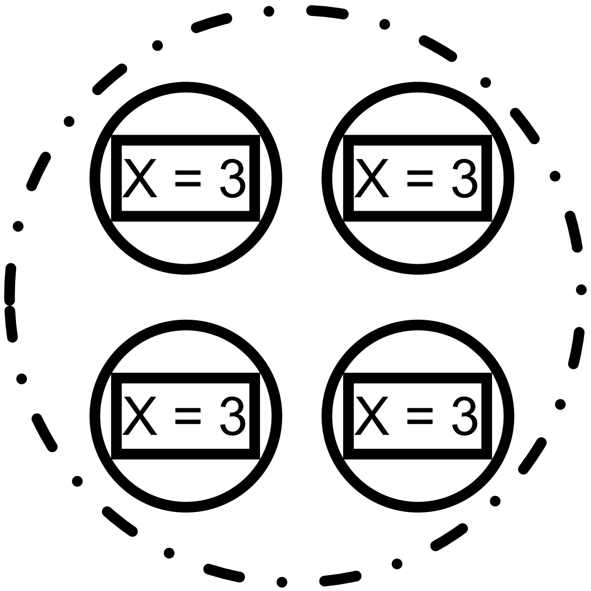
Req.	CUID	UID
r1	1.4	
r0	2.4	

1) Propose Candidates

Replica ID Generation

Req.	CUID	UID
r0	1.1	2.4
r1	2.1	

Req.	CUID	UID
r0	1.2	2.4
r1	2.2	



Req.	CUID	UID
r1	1.3	
r0	2.3	2.4

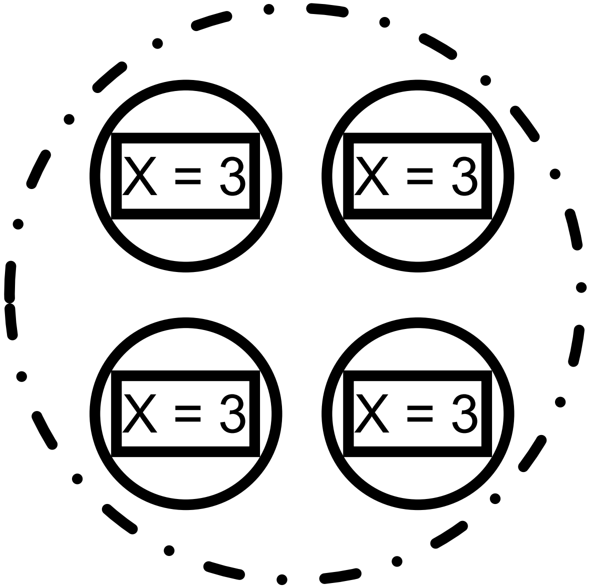
Req.	CUID	UID
r1	1.4	
r0	2.4	2.4

2) Accept *r0*

Replica ID Generation

Req.	*CUID*	*UID*
r0	*1.1*	*2.4*
r1	*2.1*	*2.2*

Req.	*CUID*	*UID*
r0	*1.2*	*2.4*
r1	*2.2*	*2.2*



Req.	*CUID*	*UID*
r1	*1.3*	*2.2*
r0	*2.3*	*2.4*

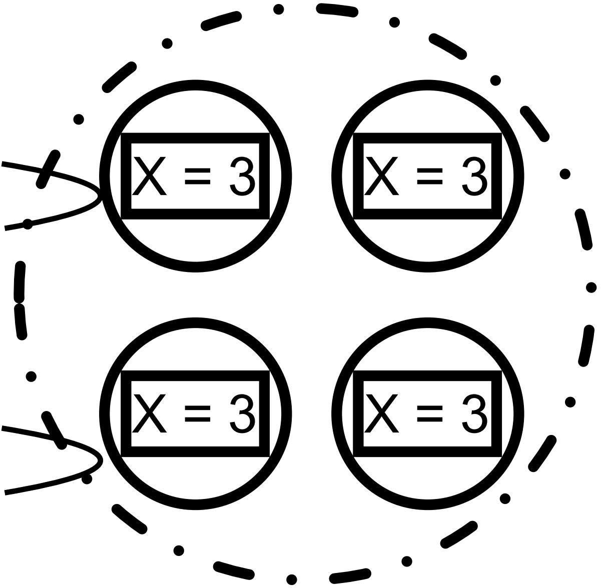
Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

3) Accept *r1*

Replica ID Generation

Req.	*CUID*	*UID*
r1	*2.1*	*2.2*
r0	*1.1*	*2.4*

Req.	*CUID*	*UID*
r1	*2.2*	*2.2*
r0	*1.2*	*2.4*



Req.	*CUID*	*UID*
r1	*1.3*	*2.2*
r0	*2.3*	*2.4*

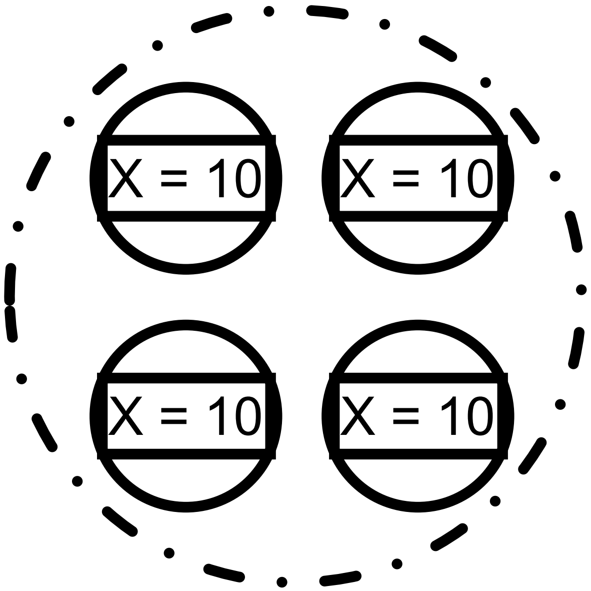
Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

r1 is now stable

Replica ID Generation

Req.	*CUID*	*UID*
r1	*2.1*	*2.2*
r0	*1.1*	*2.4*

Req.	*CUID*	*UID*
r1	*2.2*	*2.2*
r0	*1.2*	*2.4*



Req.	*CUID*	*UID*
r1	*1.3*	*2.2*
r0	*2.3*	*2.4*

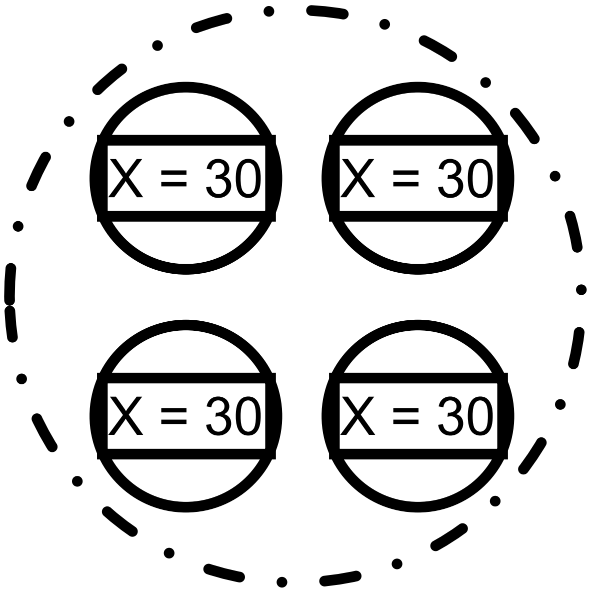
Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

4) Apply *r1*

Replica ID Generation

Req.	*CUID*	*UID*
r1	*2.1*	*2.2*
r0	*1.1*	*2.4*

Req.	*CUID*	*UID*
r1	*2.2*	*2.2*
r0	*1.2*	*2.4*



Req.	*CUID*	*UID*
r1	*1.3*	*2.2*
r0	*2.3*	*2.4*

Req.	*CUID*	*UID*
r1	*1.4*	*2.2*
r0	*2.4*	*2.4*

5) Apply *r0*

Implementation Replica Generated IDs

- 2 Rules for Candidate Generation/Selection
  - Any new candidate ID must be > the id of any *accepted* request.
  - The ID selected from the candidate list must be >= each candidate
- In the paper these are written as:
  - If a request *r'* is seen by a replica sm_i after *r* has been accepted by sm_i then $uid(r) < cuid(sm_i, r')$
  - $cuid(sm_i, r) <= uid(r)$

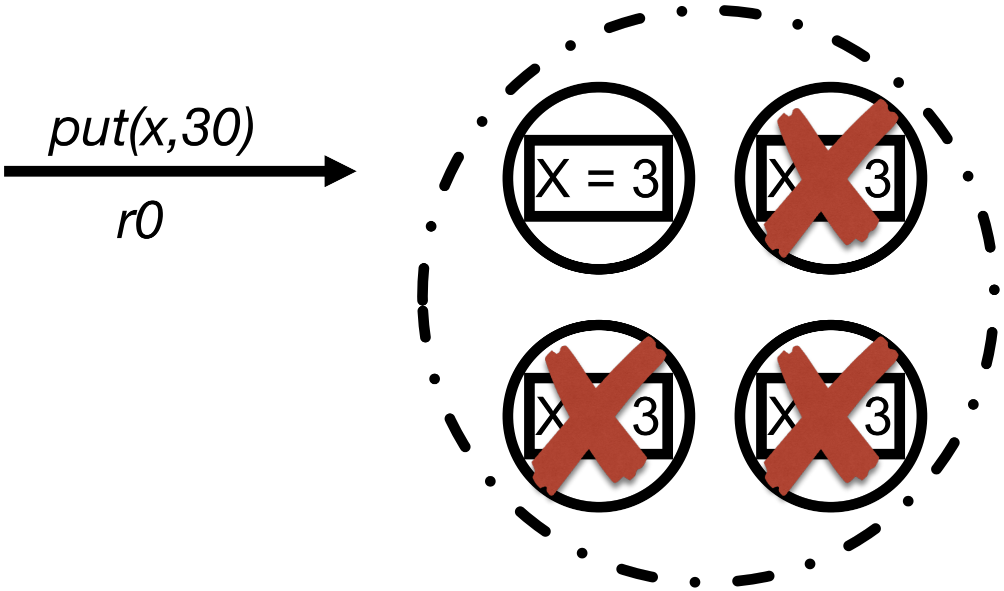
Implementation
Replica Generated IDs

- When do we know a candidate is *stable?*
  - A candidate is *accepted*
  - No other pending requests with smaller candidate ids

Fault Tolerance

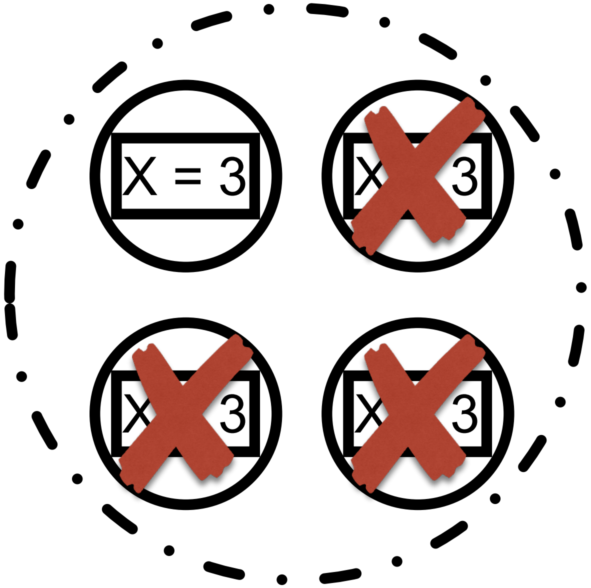
- Fail-Stop
  - A faulty server can be detected as faulty
- Byzantine
  - Faulty servers can do arbitrary, perhaps malicious things
- Crash Failures - NOT covered in paper
  - Server can stop responding without notification (subset of Byzantine)

Fail-Stop Tolerance



Fail-Stop Tolerance

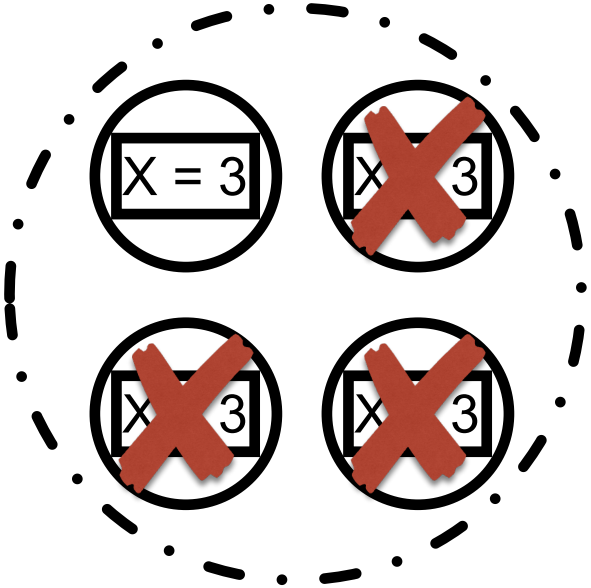
Req.	*CUID*	*UID*
r0	*1.1*	



1) Propose Candidates….

Fail-Stop Tolerance

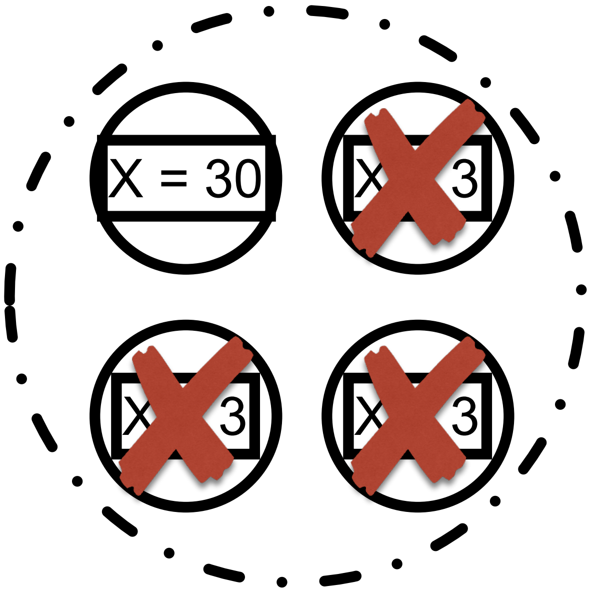
Req.	*CUID*	*UID*
r0	*1.1*	*1.1*



2) Accept *r0*

Fail-Stop Tolerance

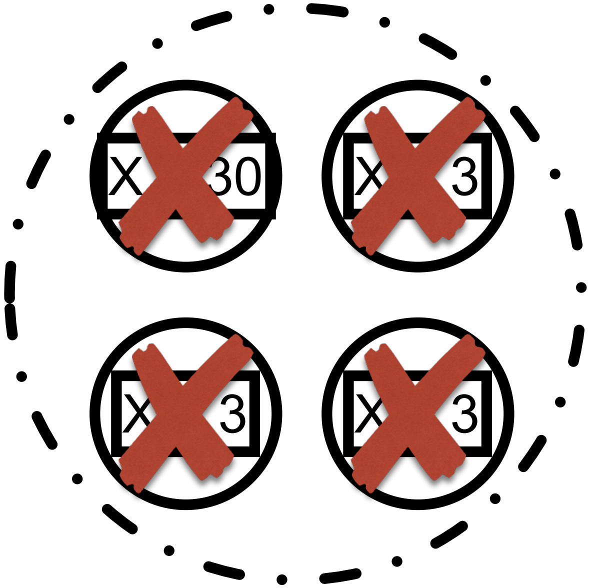
Req.	*CUID*	*UID*
r0	*1.1*	*1.1*



2) Apply *r0*

Fail-Stop Tolerance

GAME OVER!!!

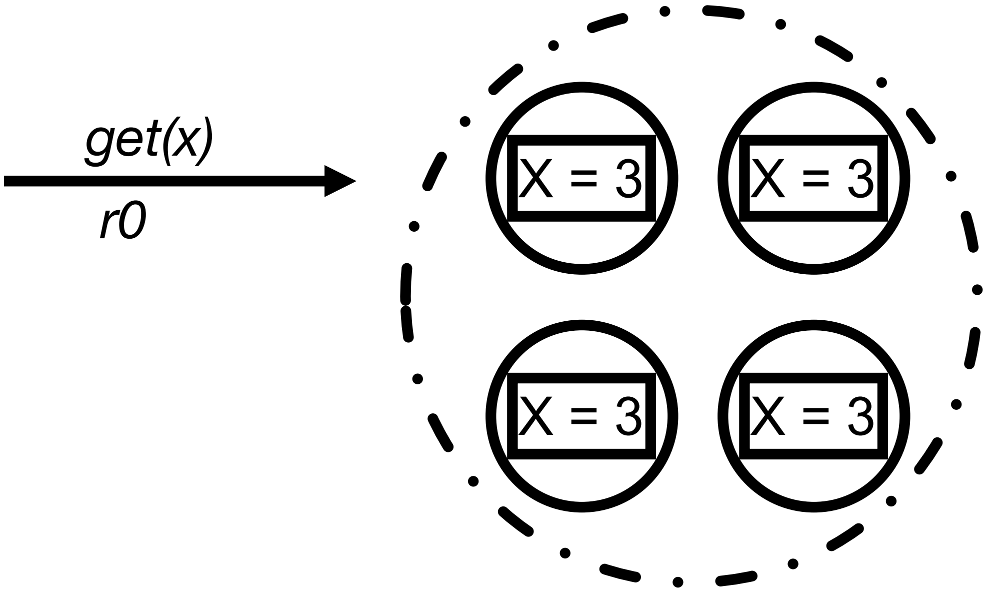


2) Apply *r0*

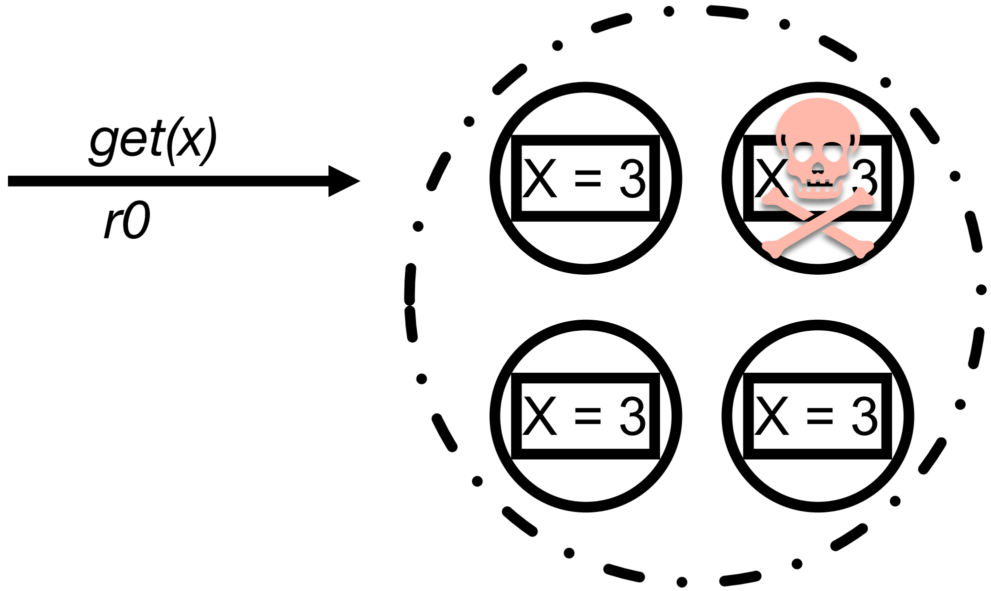
Fail-Stop Tolerance

- To tolerate *t* failures, need *t+1* servers.
- As long as 1 server remains, we’re OK!
- Only need to participate in protocols with other *live* servers

Byzantine Tolerance



Byzantine Tolerance



get(x)

r0

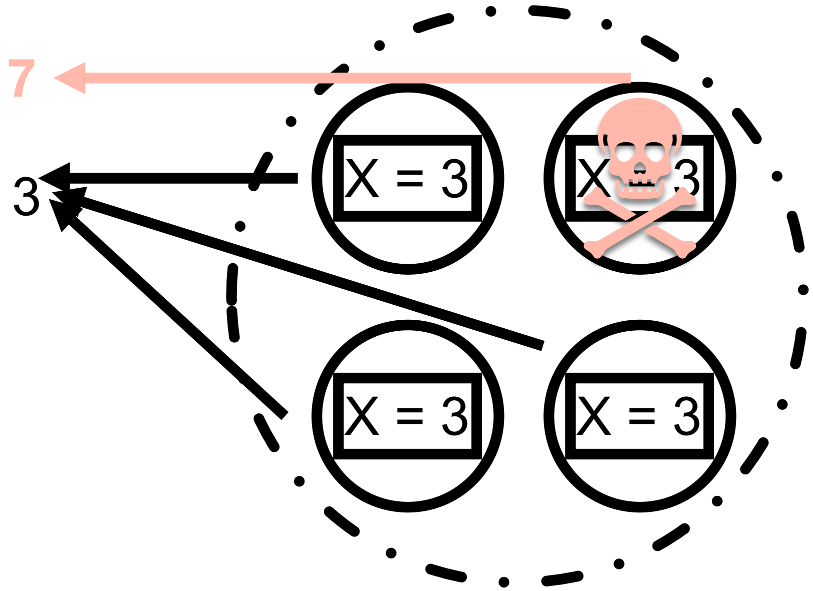
X = 3

X = 3

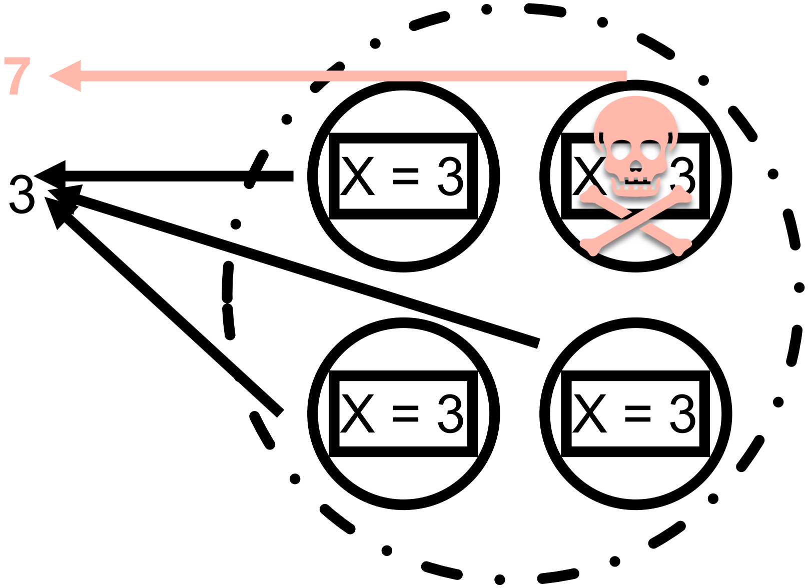
X = 3

X = 3

Byzantine Tolerance

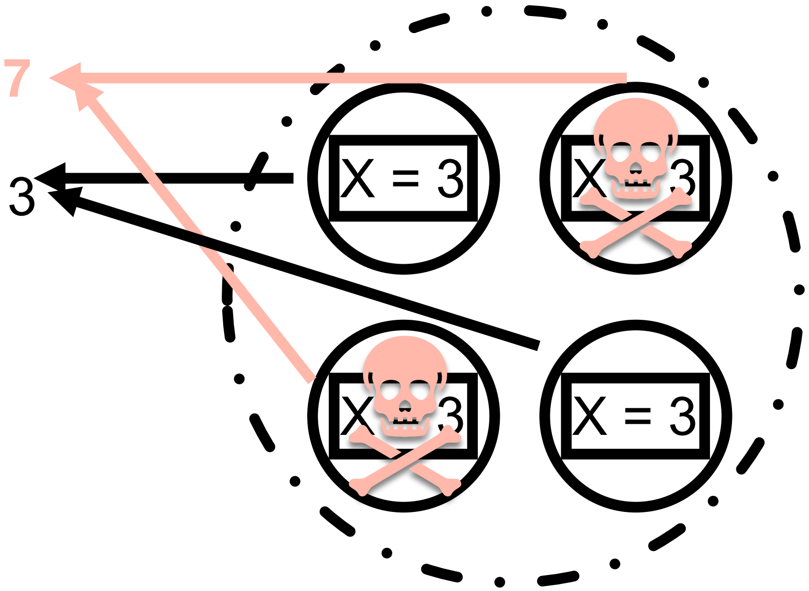


Byzantine Tolerance



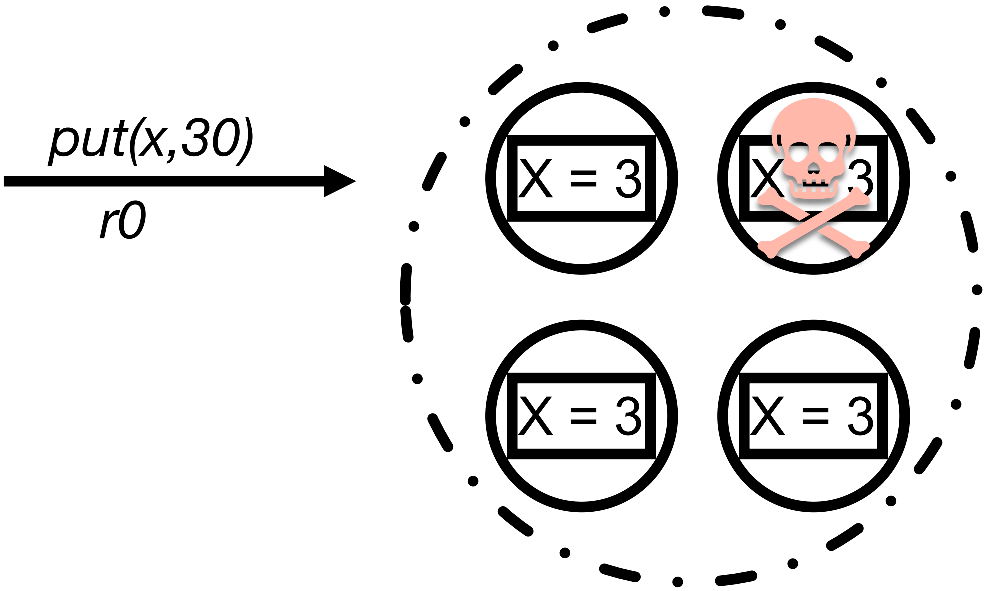
Client trusts the majority =>
Need majority to participate in replication

Byzantine Tolerance



Who to trust?? 3 or 7?

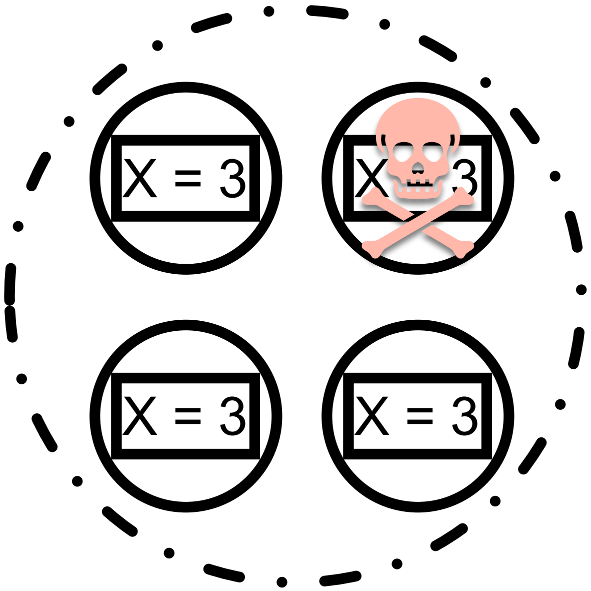
Byzantine Tolerance



Byzantine Tolerance

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	1.2	



Req.	CUID	UID
r0	???	???

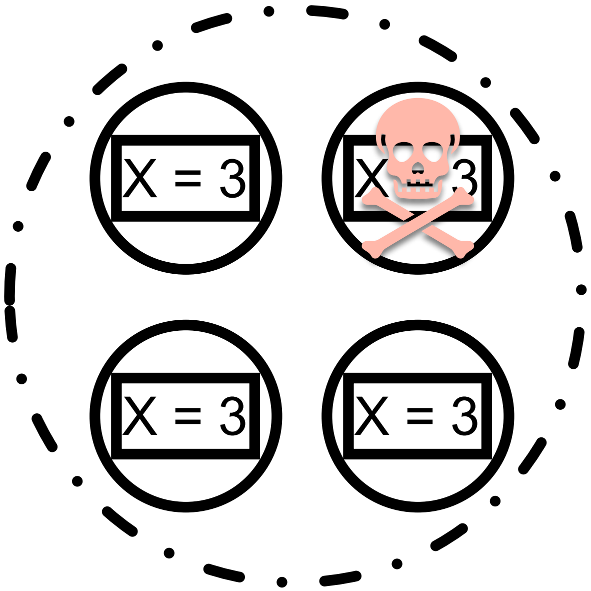
Req.	CUID	UID
r0	1.4	

1) Propose Candidates

Byzantine Tolerance
a) No response

Req.	*CUID*	*UID*
r0	*1.1*	

Req.	*CUID*	*UID*
r0	*1.2*	



Req.	*CUID*	*UID*
r0	*???*	*???*

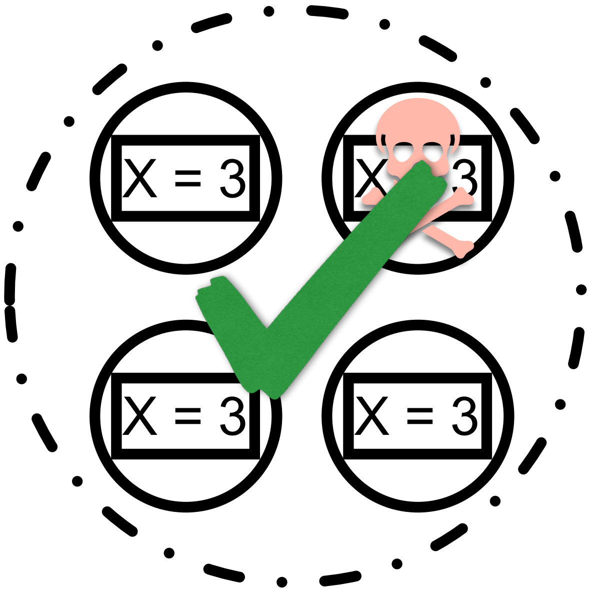
Req.	*CUID*	*UID*
r0	*1.4*	

a) Wait for majority candidates
Timeout long requests & notify others

Byzantine Tolerance
a) No response

Req.	*CUID*	*UID*
r0	*1.1*	*1.4*

Req.	*CUID*	*UID*
r0	*1.2*	*1.4*



Req.	*CUID*	*UID*
r0	*???*	*???*

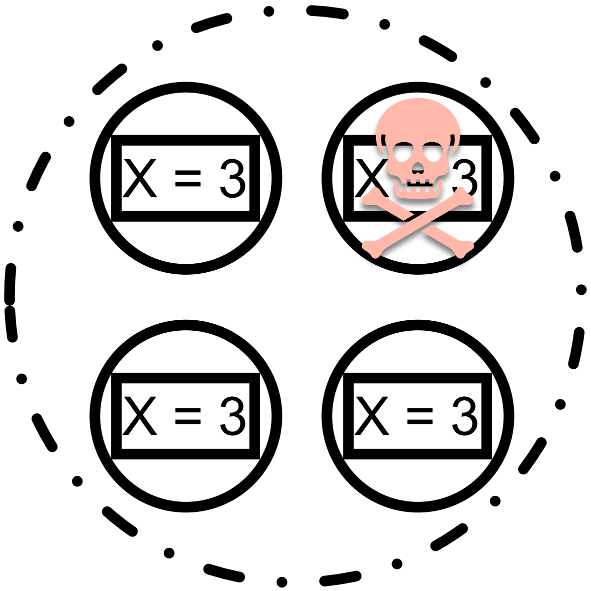
Req.	*CUID*	*UID*
r0	*1.4*	*1.4*

a) *Accept*
r0

Byzantine Tolerance
Small ID

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	-5	???



Req.	CUID	UID
r0	1.2	

Req.	CUID	UID
r0	1.4	

1) Propose Candidates

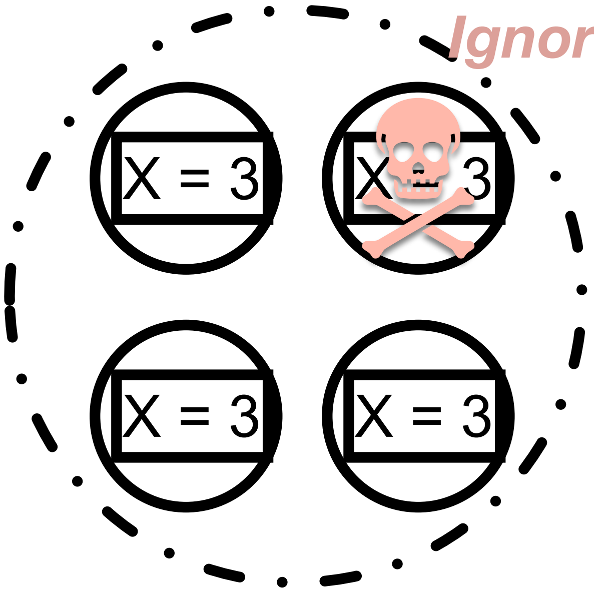
Byzantine Tolerance
Small ID

$uid = max(cuid(sm_i,r))$

Ignore low candidates!

Req.	*CUID*	*UID*
r0	*1.1*	

Req.	*CUID*	*UID*
r0	*1.2*	



Req.	*CUID*	*UID*
r0	*-5*	*???*

Req.	*CUID*	*UID*
r0	*1.4*	

2) *Accept*
r0

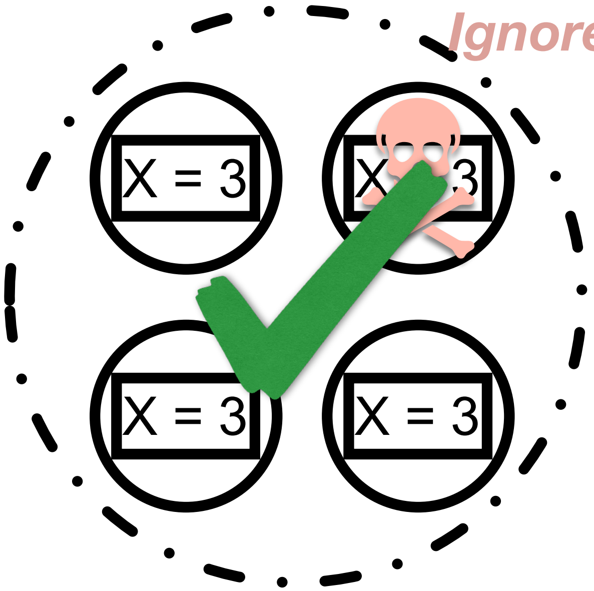
Byzantine Tolerance
Small ID

$uid = max(cuid(sm_i,r))$

Ignore low candidates!

Req.	*CUID*	*UID*
r0	*1.1*	*1.4*

Req.	*CUID*	*UID*
r0	*-5*	*???*



Req.	*CUID*	*UID*
r0	*1.2*	*1.4*

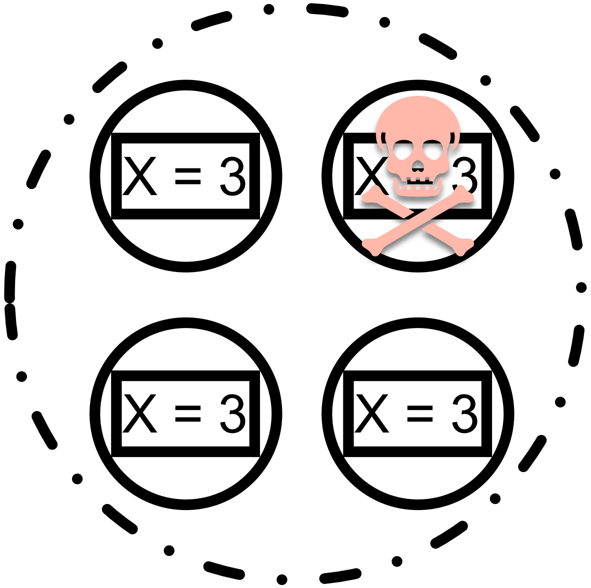
Req.	*CUID*	*UID*
r0	*1.4*	*1.4*

2) *Accept*
r0

Byzantine Tolerance
Large ID

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	1.2	



Req.	CUID	UID
r0	10	???

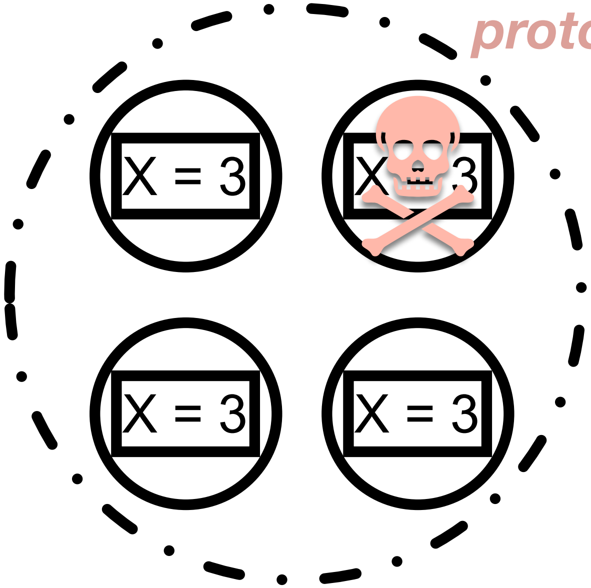
Req.	CUID	UID
r0	1.4	

1) Propose Candidates

Byzantine Tolerance
Large ID

Req.	CUID	UID
r0	1.1	

Req.	CUID	UID
r0	1.2	



Large numbers follow protocol!

Req.	CUID	UID
r0	10	???

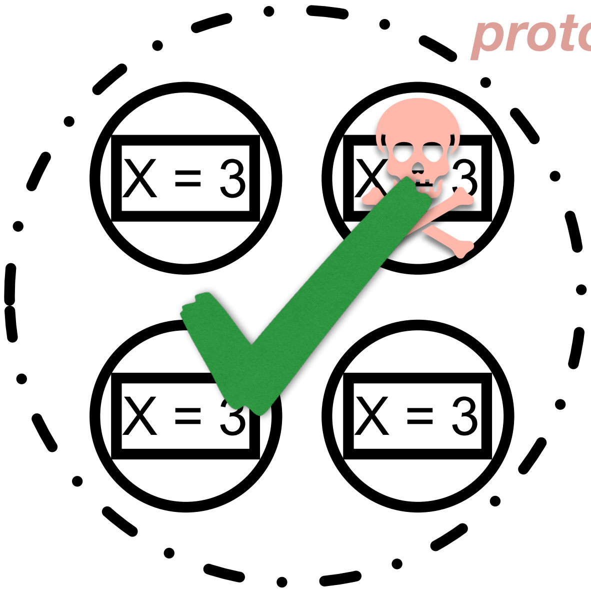
Req.	CUID	UID
r0	1.4	

Byzantine Tolerance
Large ID

Large numbers follow protocol!

Req.	*CUID*	*UID*
r0	*1.1*	*10*

Req.	*CUID*	*UID*
r0	*1.2*	*10*



Req.	*CUID*	*UID*
r0	*10*	*???*

Req.	*CUID*	*UID*
r0	*1.4*	*10*

2) *Accept*
r0

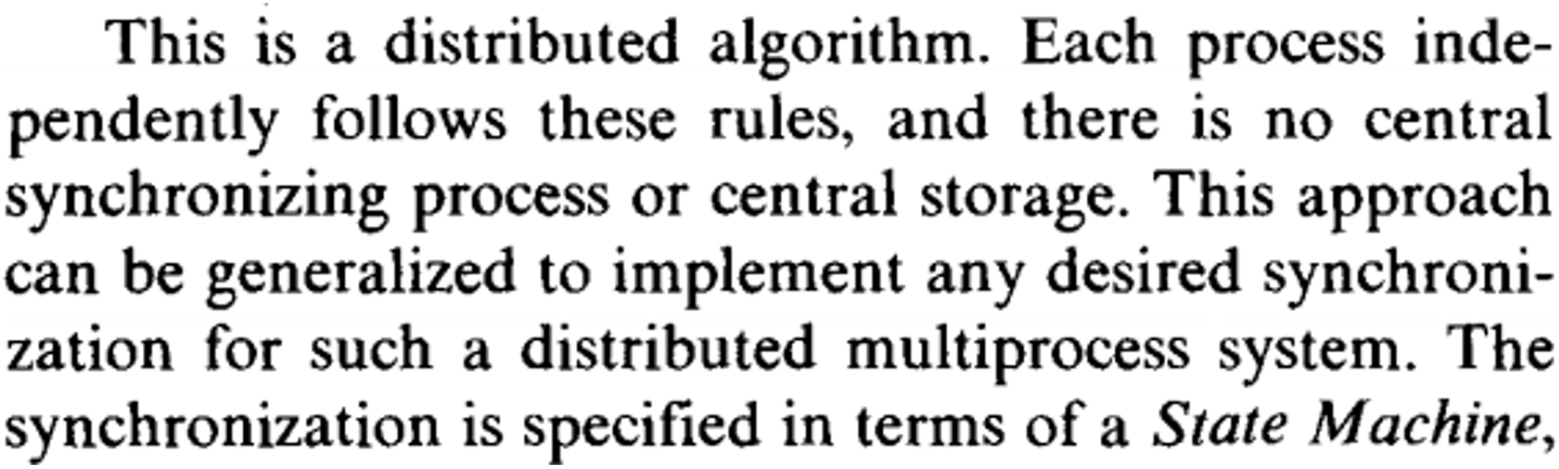
Fault Tolerance

- Byzantine Failures
  - To tolerate *t* failures, need *2t + 1* servers
  - Protocols now involve votes
    - Can only trust server response if the majority of servers say the same thing
  - *t + 1* servers need to participate in replication protocols

Other Contributions

- Tolerating Faulty Output Devices
  - (e.g. a faulty network, or user-facing i/o)
- Tolerating Faulty Clients
- Reconfiguration

Takeaways



This is a distributed algorithm. Each process independently follows these rules, and there is no central synchronizing process or central storage. This approach can be generalized to implement any desired synchronization for such a distributed multiprocess system. The synchronization is specified in terms of a *State Machine*,

Lamport 1978

Takeaways

- Can represent ***deterministic*** distributed system as *Replicated State Machine*
- Each replica reaches the same conclusion about the system ***independently***
- Key examples of *distributed algorithms* that generically implement *SMR*
- Formalizes notions of fault-tolerance in *SMR*

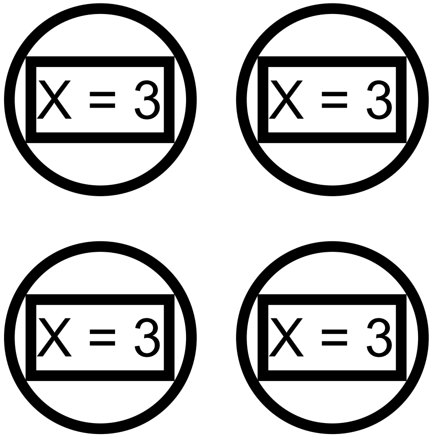
Chain Replication

- Authors
  - Robert Van Renesse (RVR)
  - Fred Schneider

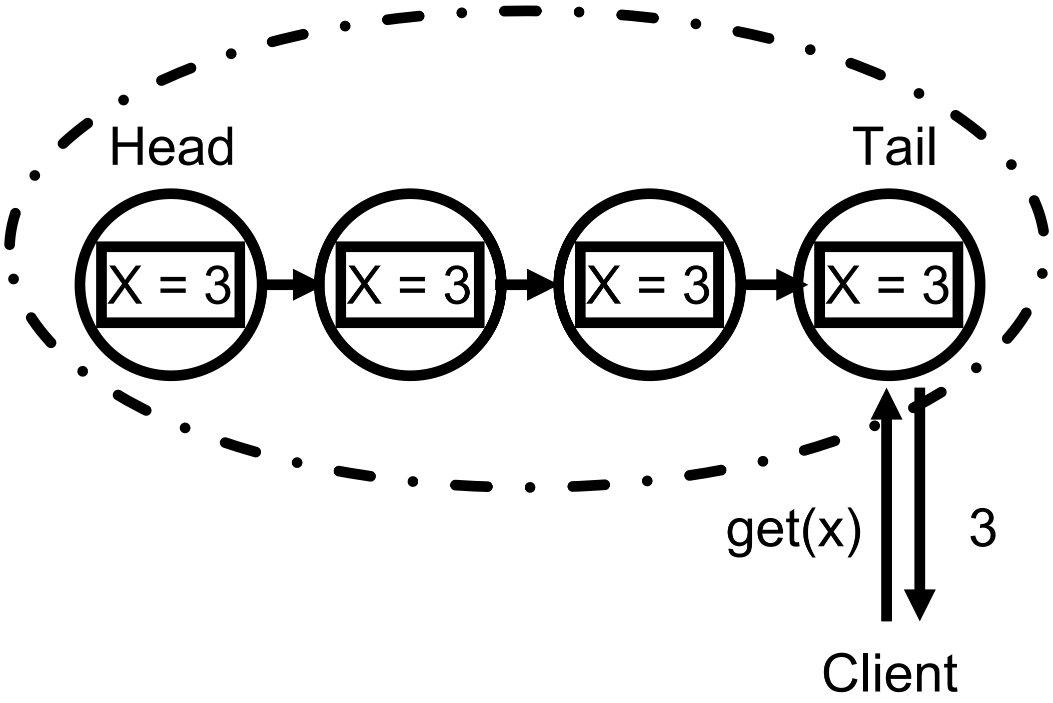
Chain Replication

- Fault Tolerant Storage Service (Fail-Stop)
- Requests:
  - Update(x, y) => set object *x* to value *y*
  - Query(x) => read value of object *x*

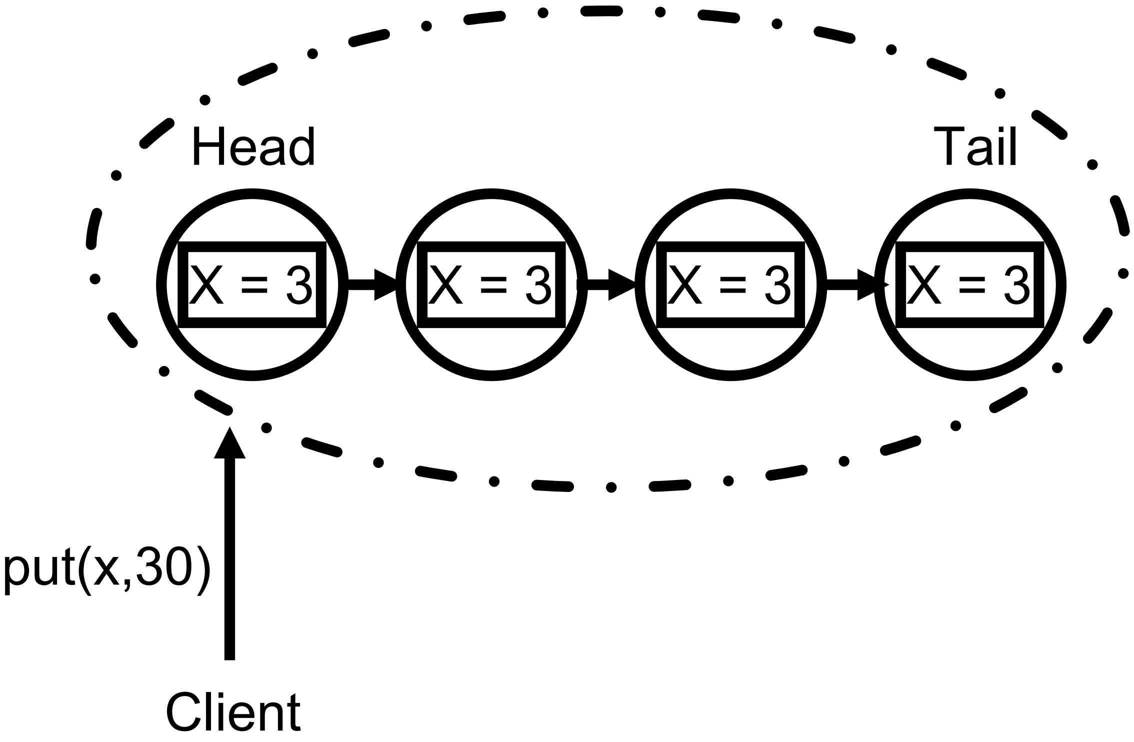
Chain Replication



Chain Replication

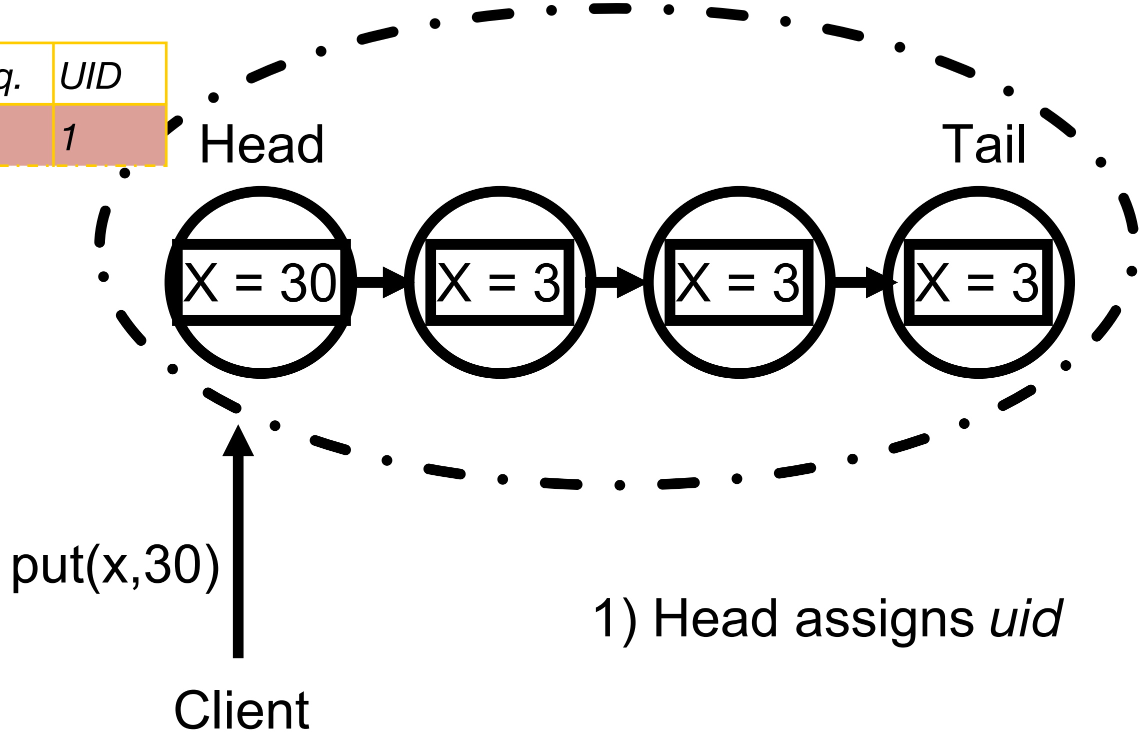


Chain Replication



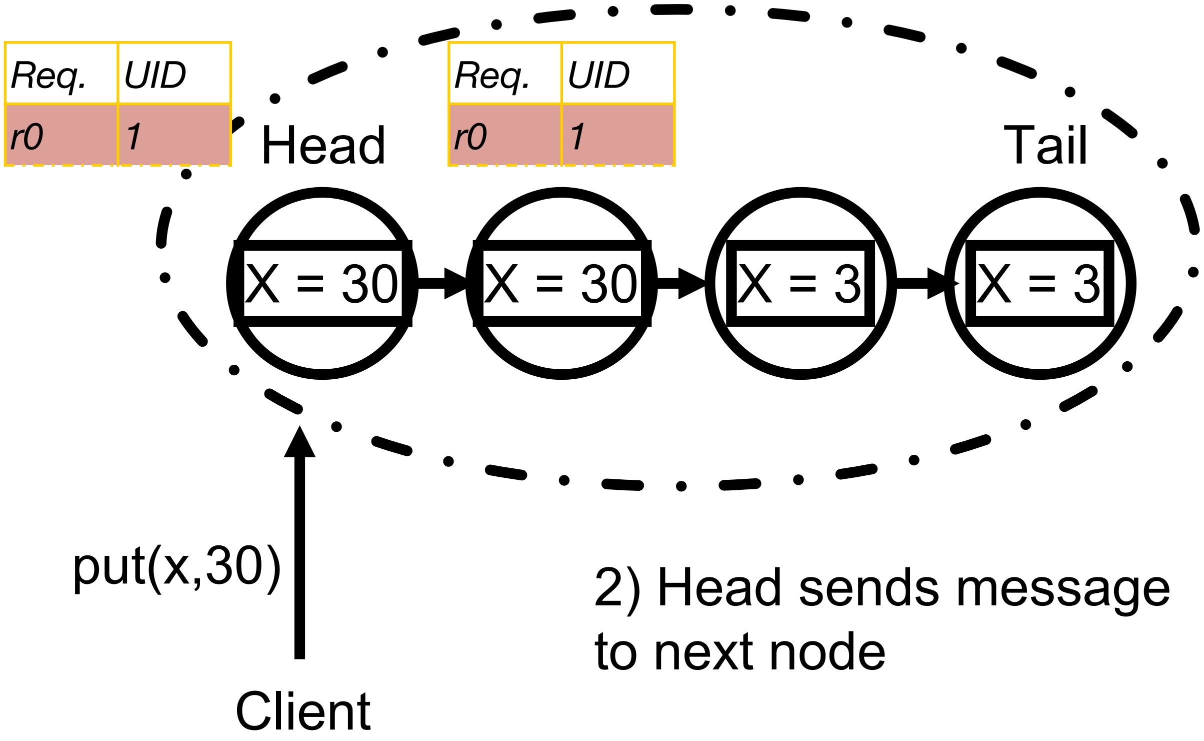
Chain Replication

Req.	*UID*
r0	*1*



1) Head assigns *uid*

Chain Replication

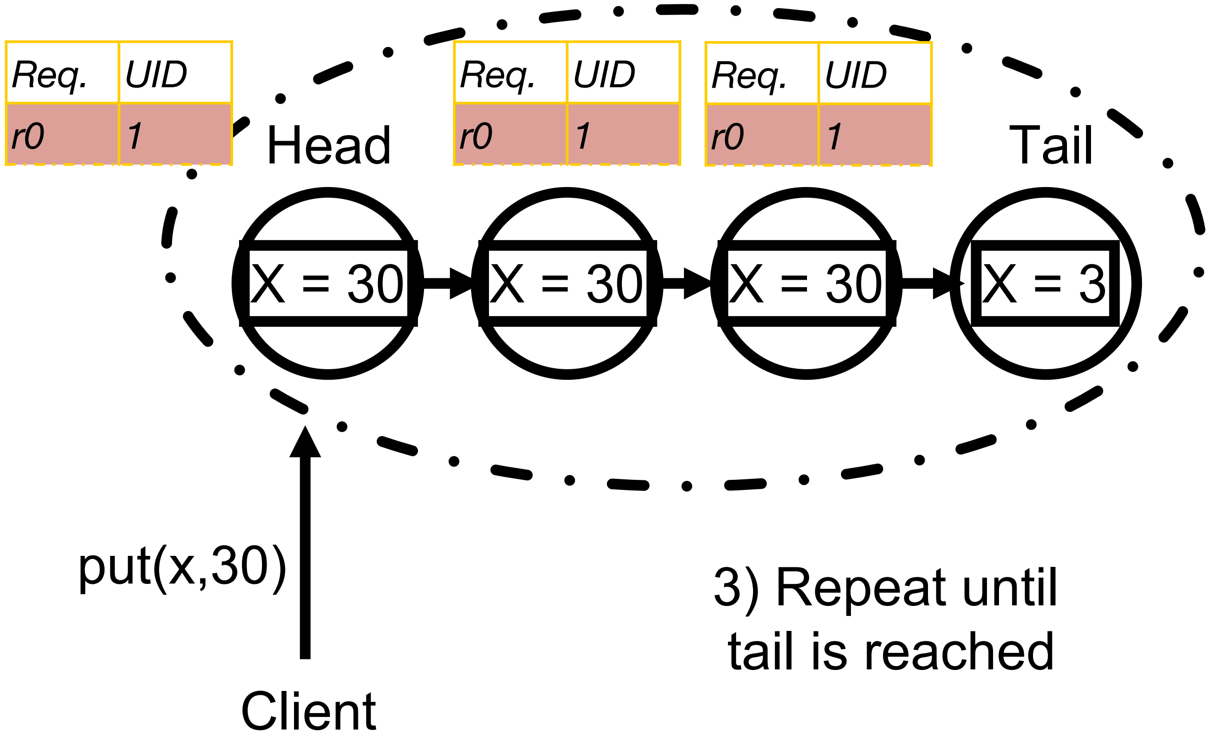


Req.	UID
r0	1

Req.	UID
r0	1

2) Head sends message to next node

Chain Replication



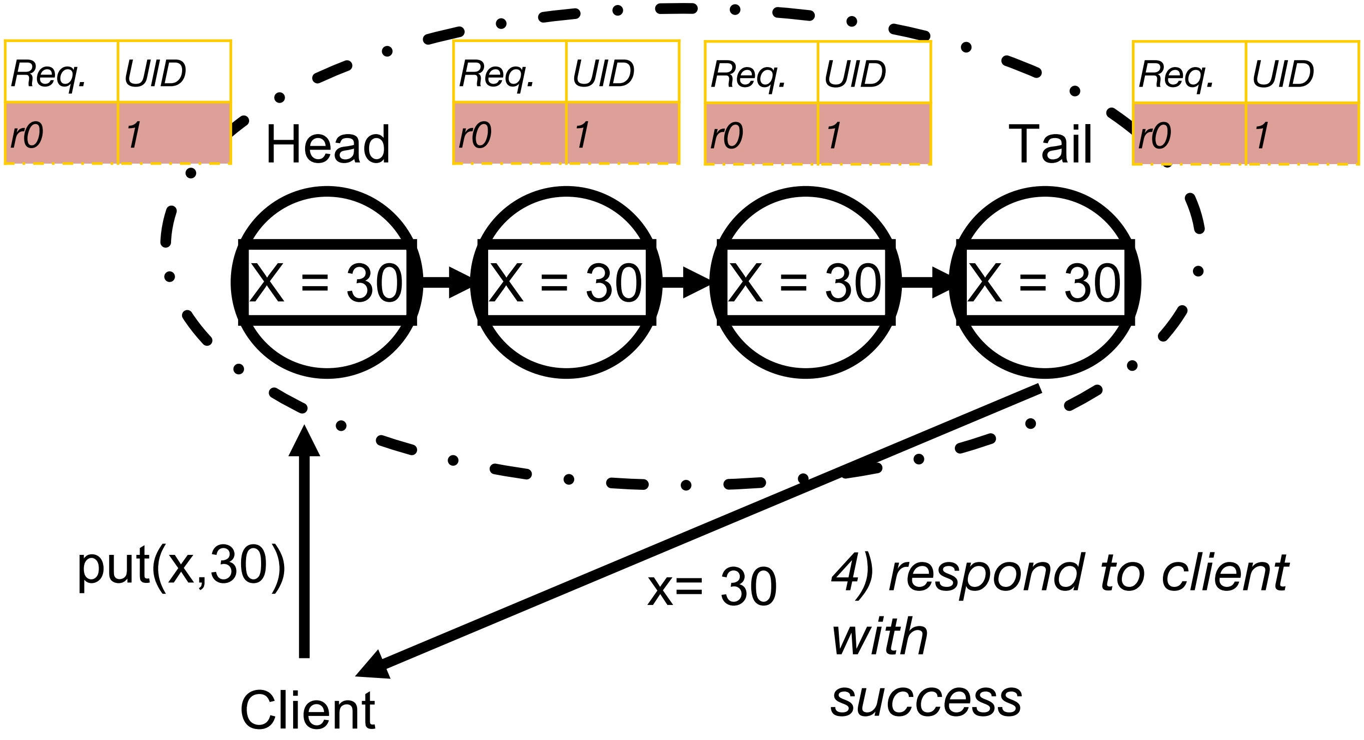
Req.	UID
r0	1

Req.	UID
r0	1

Req.	UID
r0	1

3) Repeat until
tail is reached

Chain Replication



Req.	UID
r0	*1*

Req.	UID
r0	*1*

Req.	UID
r0	*1*

Req.	UID
r0	*1*

4) respond to client with success

Chain Replication

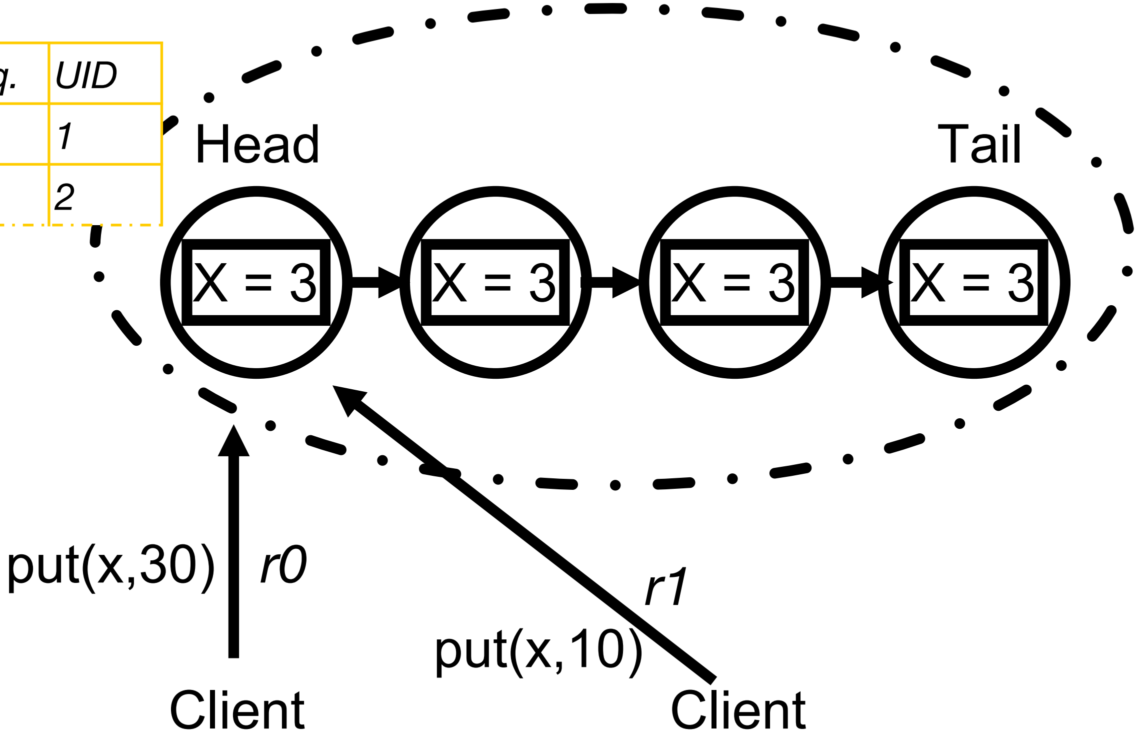
- How does Chain Replication implement State Machine Replication?
- Agreement
  - Only *Update* modifies state, can ignore *Query*
  - Client always sends *update* to *Head*. *Head* propagates request down chain to *Tail*.
  - Everyone accepts the request!

Chain Replication

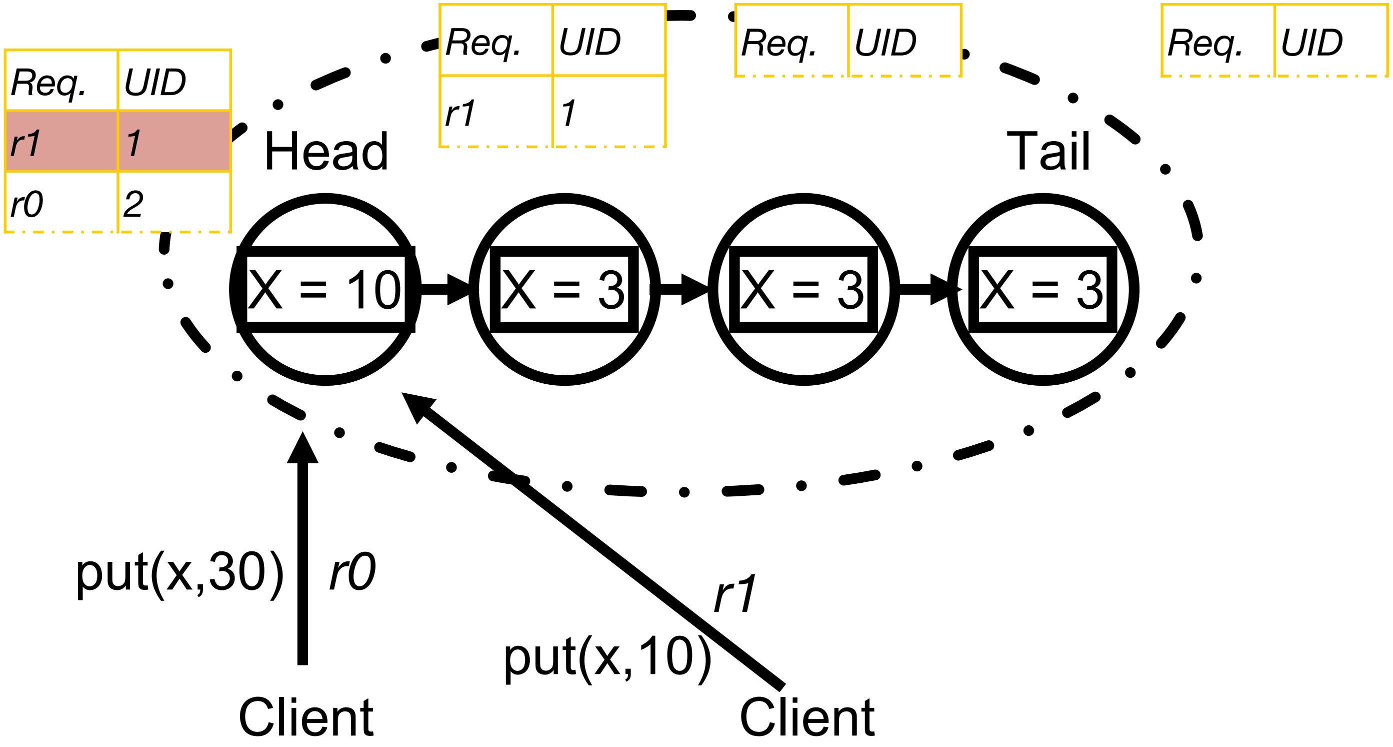
- How does Chain Replication implement State Machine Replication?
- Order
  - Unique IDs generated implicitly by *Head*'s ordering
  - FIFO order preserved down the chain
  - Tail interleaves *Query* requests
  - How can clients test stability? (How can clients tell when their *Updates* have been handled)

Chain Replication

Req.	UID
r1	1
r0	2



Chain Replication



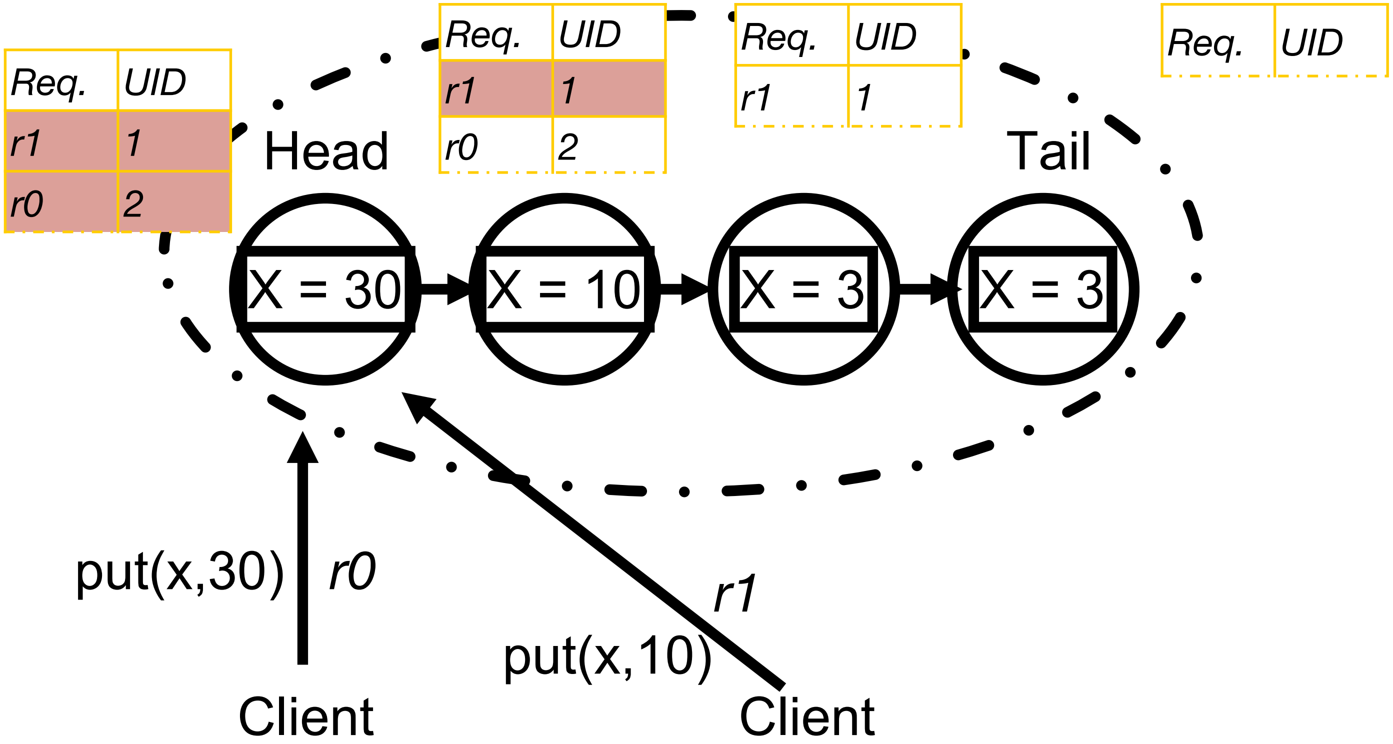
Req.	UID
r1	1
r0	2

Req.	UID
r1	1

Req.	UID

Req.	UID

Chain Replication



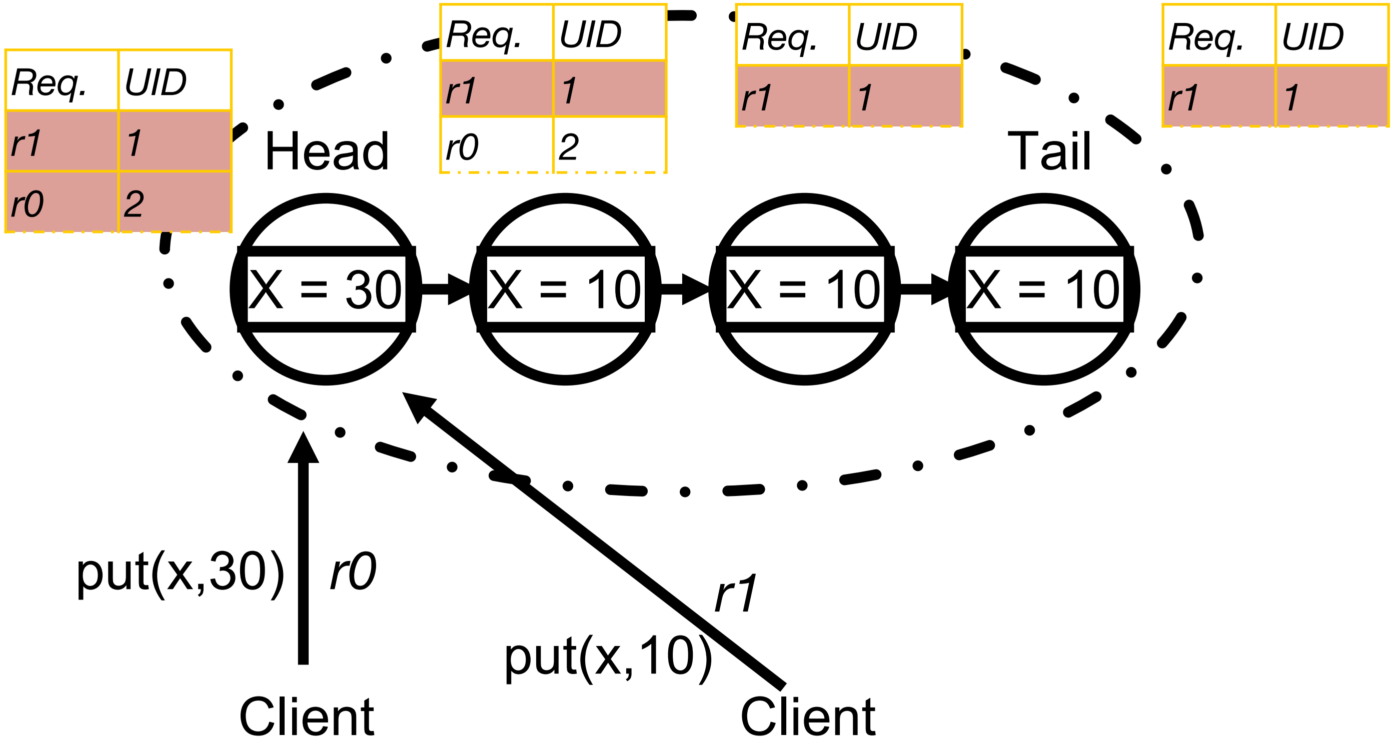
Req.	*UID*
r1	*1*
r0	*2*

Req.	*UID*
r1	*1*
r0	*2*

Req.	*UID*
r1	*1*

Req.	*UID*

Chain Replication



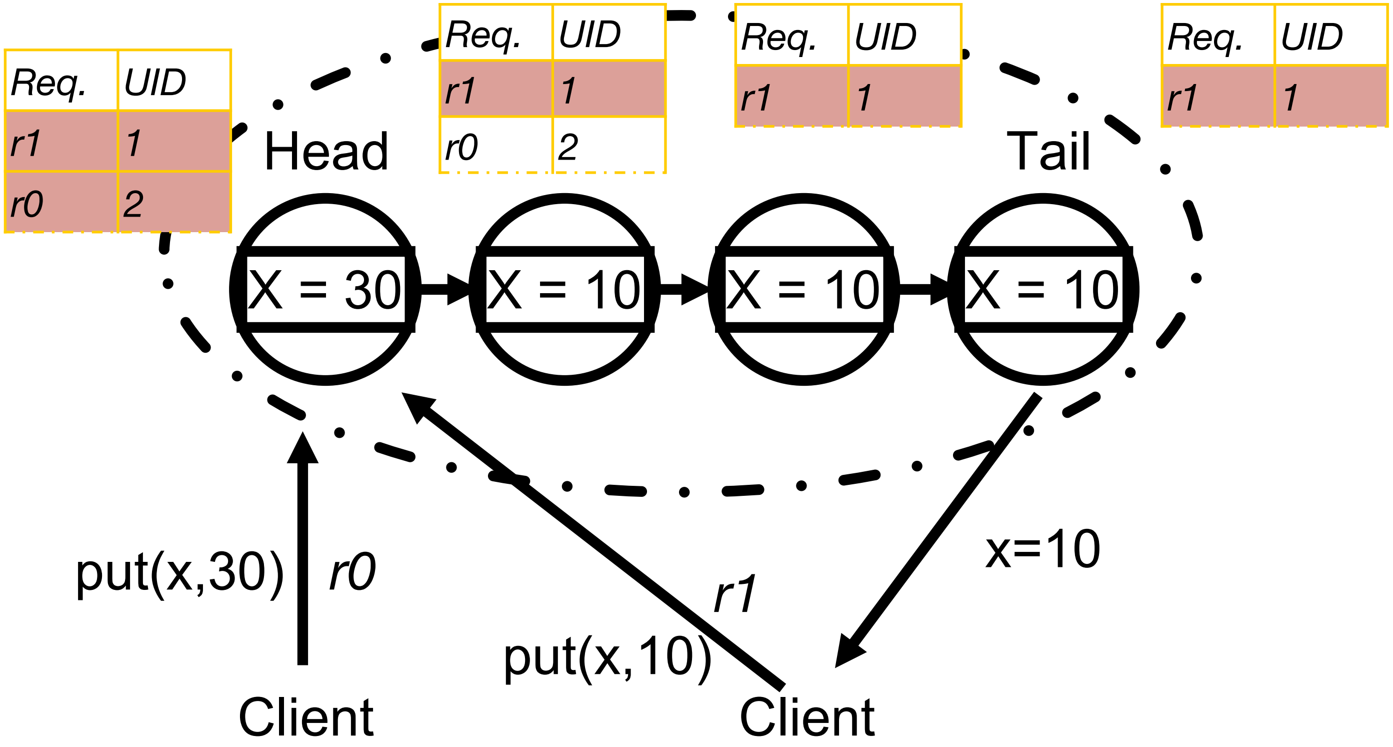
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1

Req.	UID
r1	1

Chain Replication



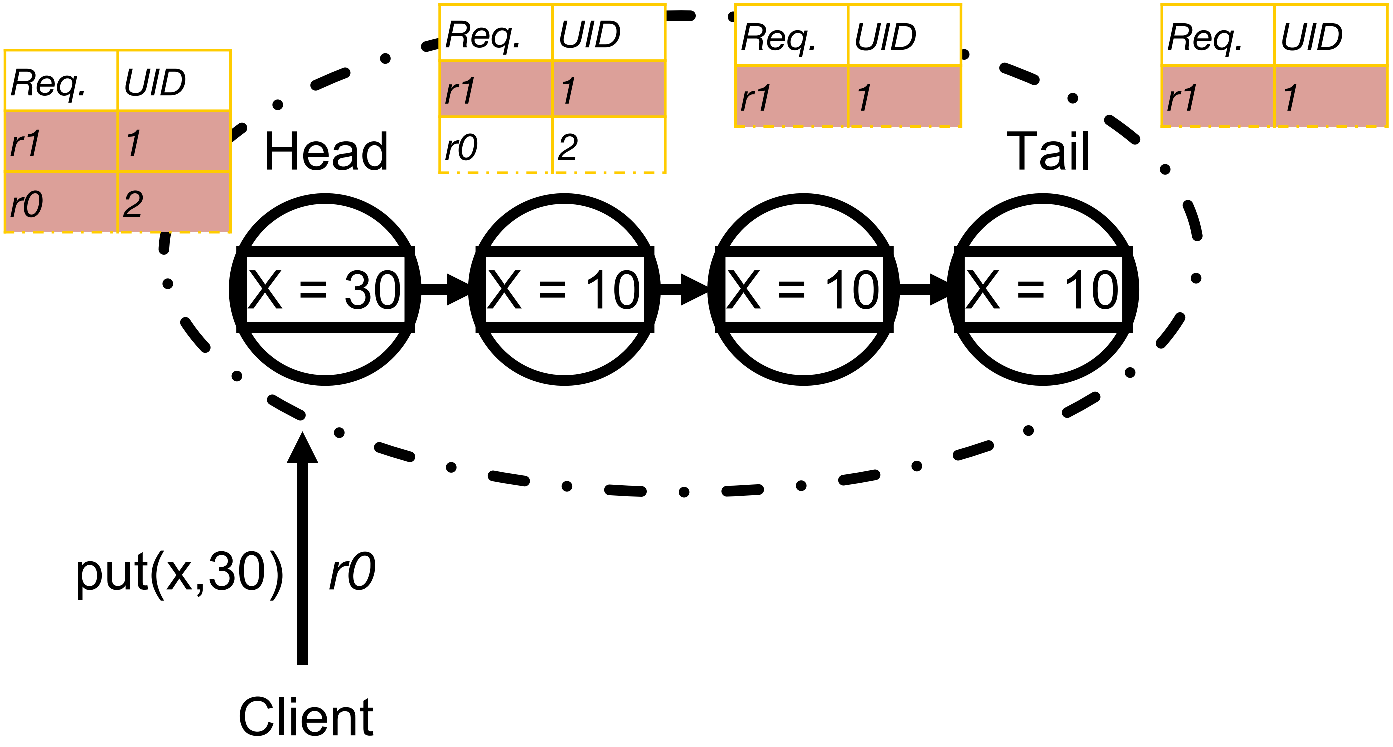
Req.	*UID*
r1	*1*
r0	*2*

Req.	*UID*
r1	*1*
r0	*2*

Req.	*UID*
r1	*1*

Req.	*UID*
r1	*1*

Chain Replication



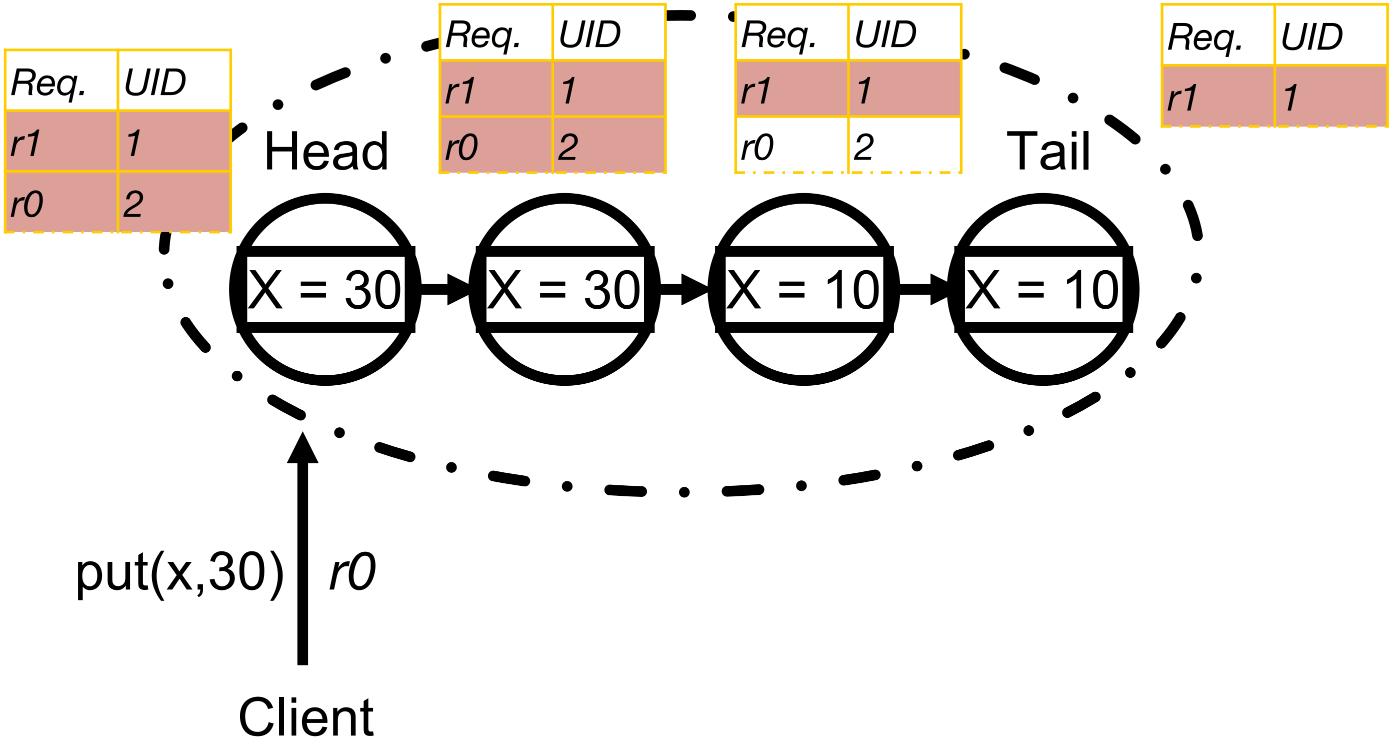
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1

Req.	UID
r1	1

Chain Replication



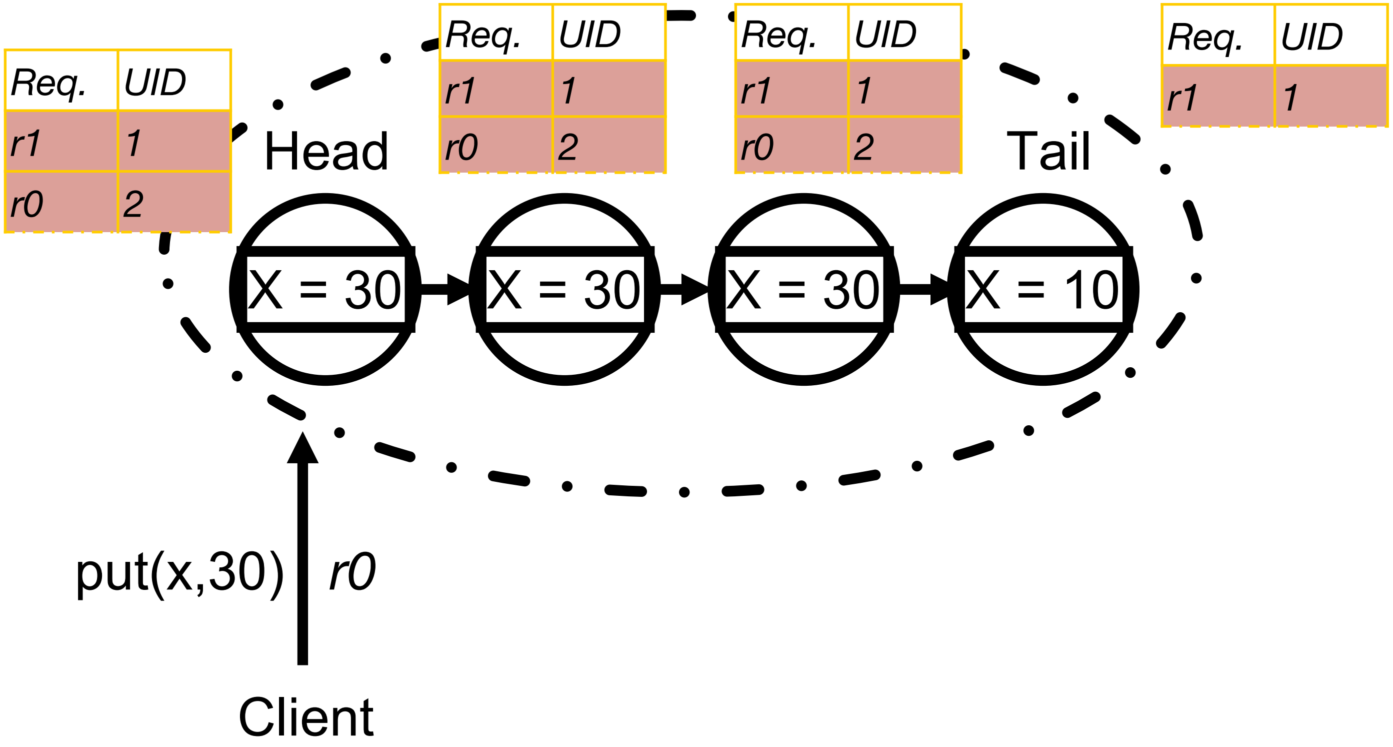
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1

Chain Replication



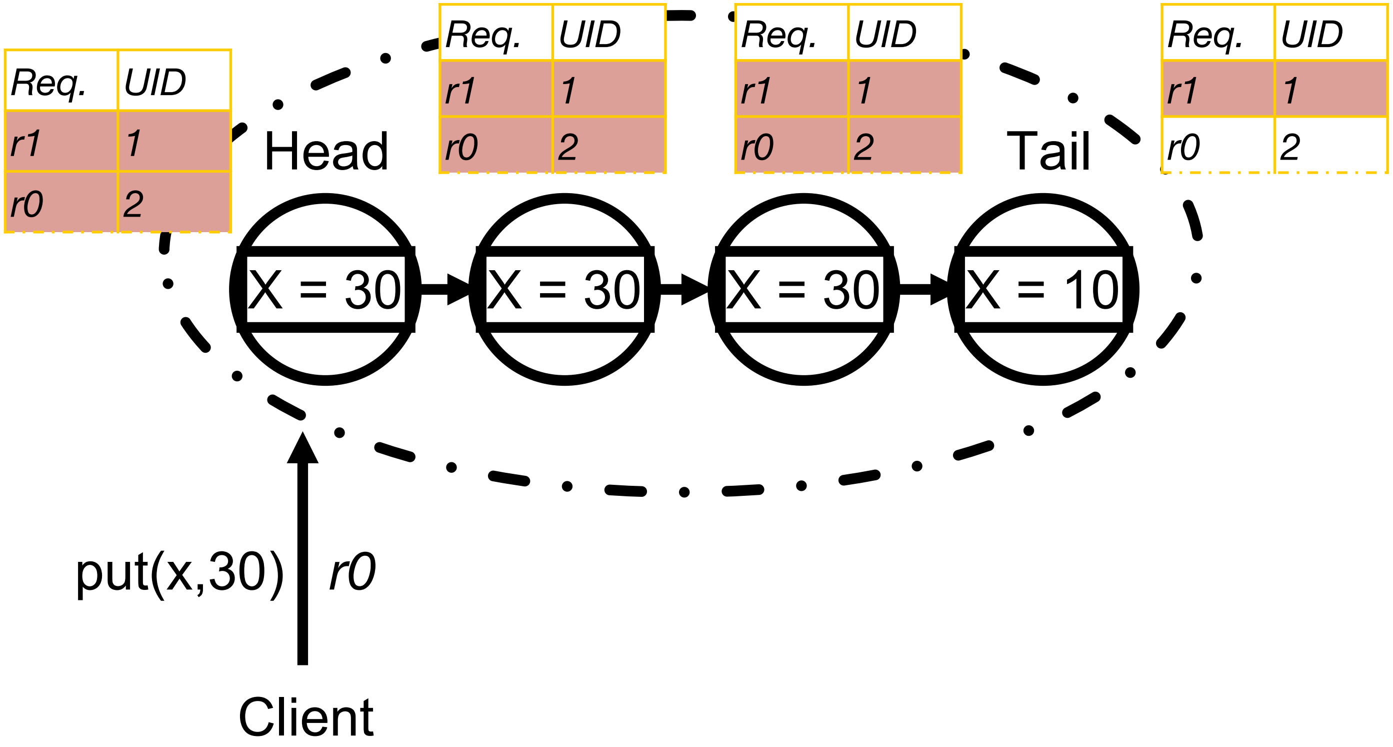
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1

Chain Replication



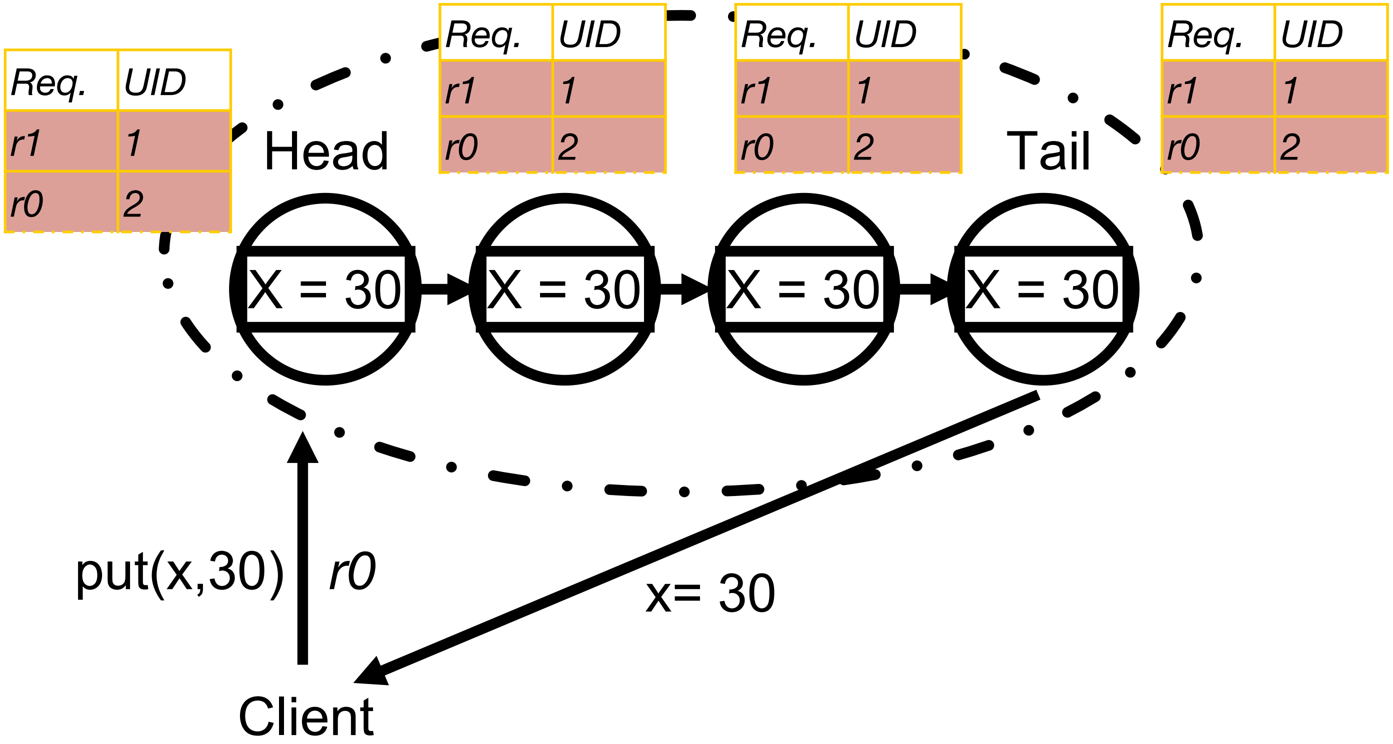
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Chain Replication



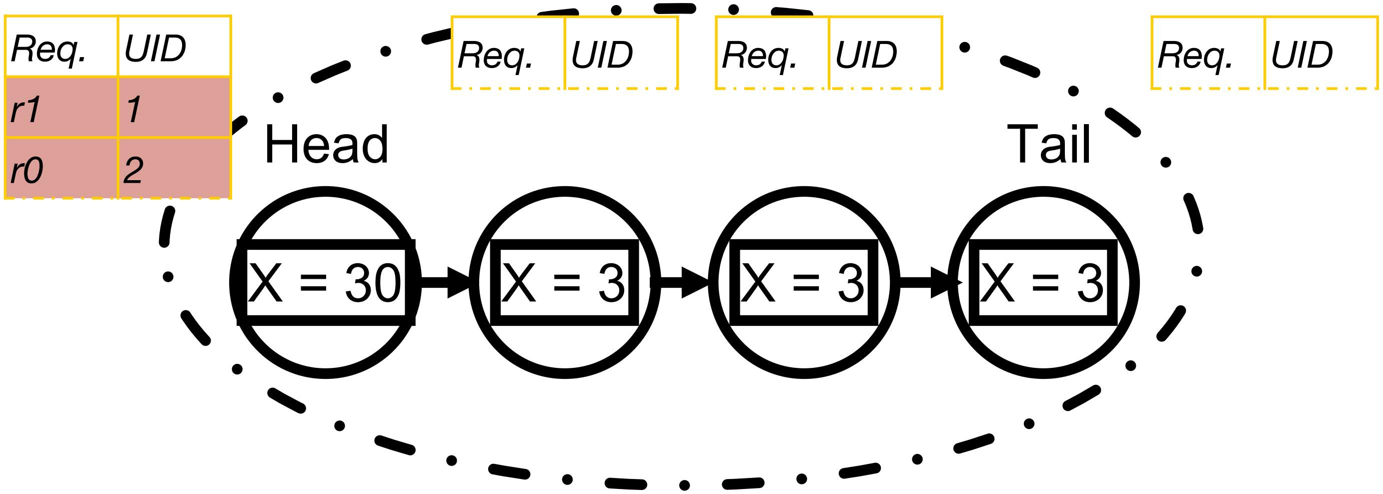
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Fault Tolerance



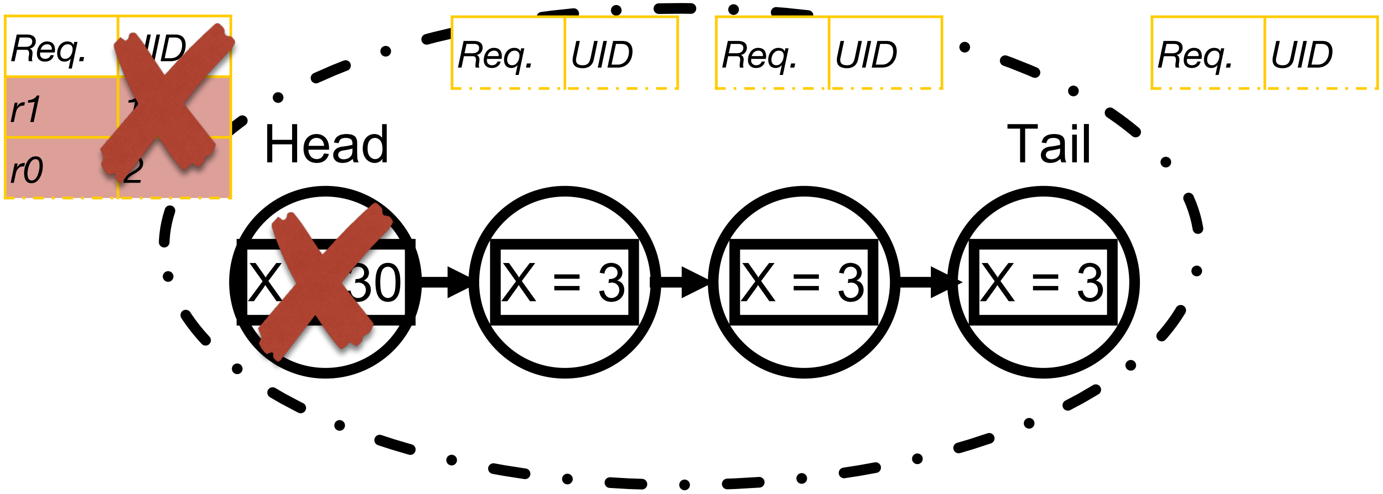
Req.	UID
r1	1
r0	2

Req.	UID

Req.	UID

Req.	UID

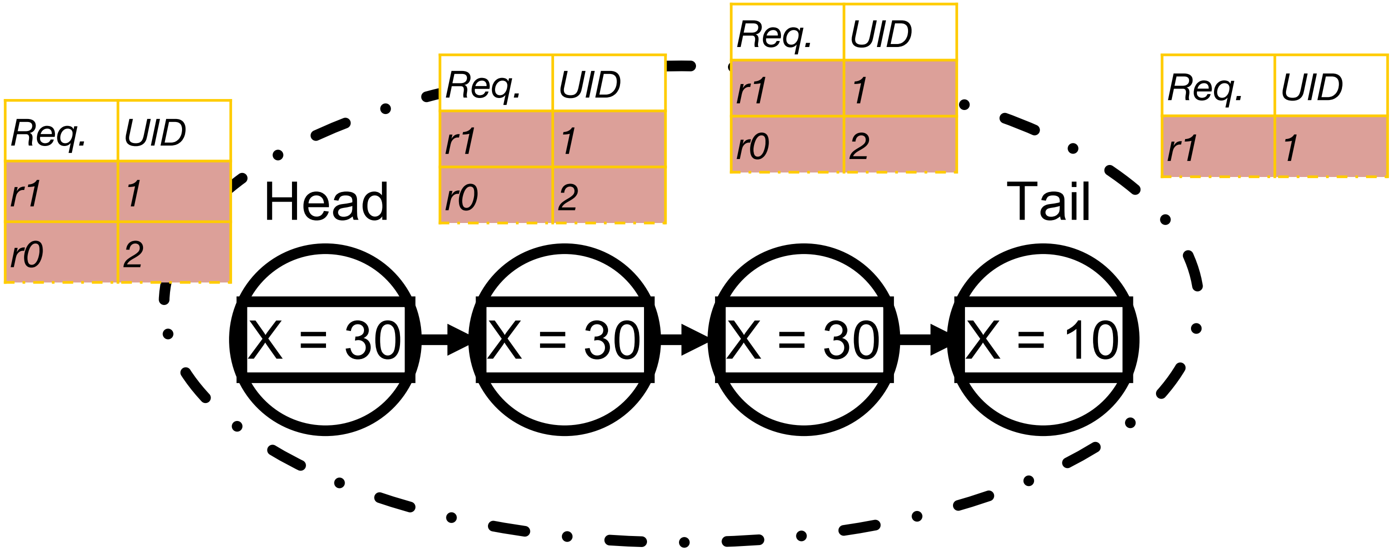
Fault Tolerance



Req.	UID
r1	1
r0	2

Dropped requests *r1* and *r0*

Fault Tolerance



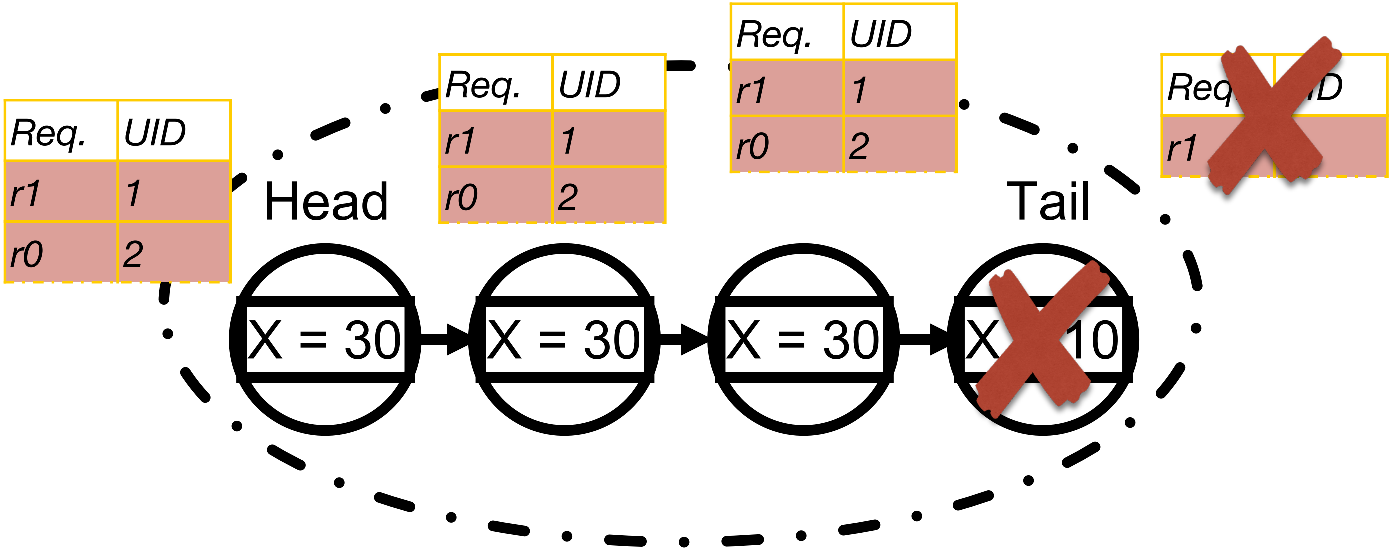
Req.	UID
r1	*1*
r0	*2*

Req.	UID
r1	*1*
r0	*2*

Req.	UID
r1	*1*
r0	*2*

Req.	UID
r1	*1*

Fault Tolerance



Req.	UID
r1	1
r0	2

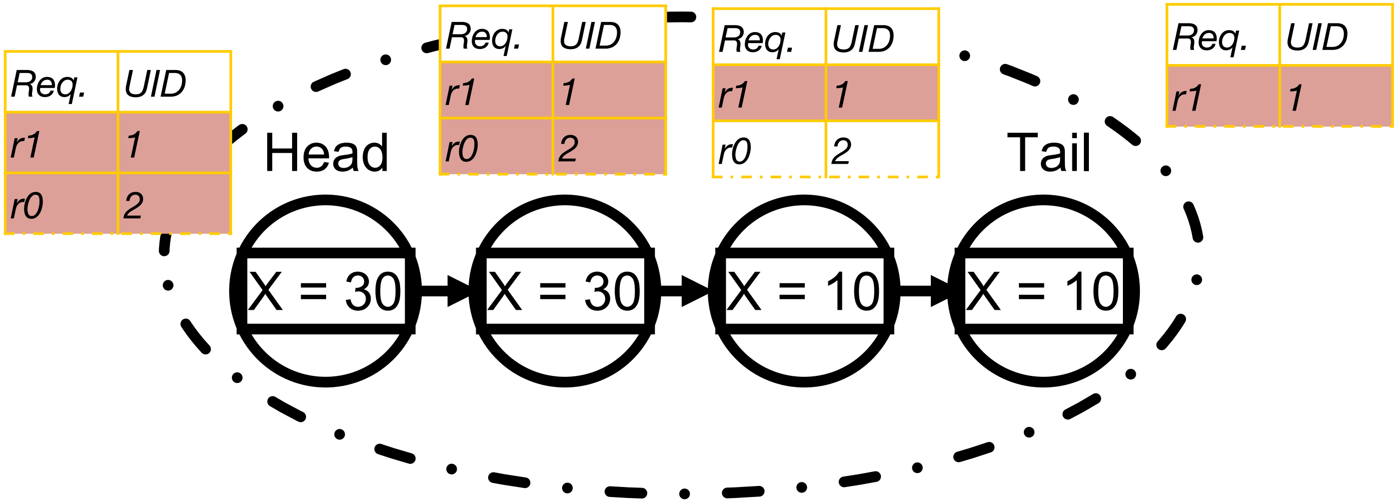
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req	ID
r1	

New tail is *stable* for superset of old tail's requests

Fault Tolerance



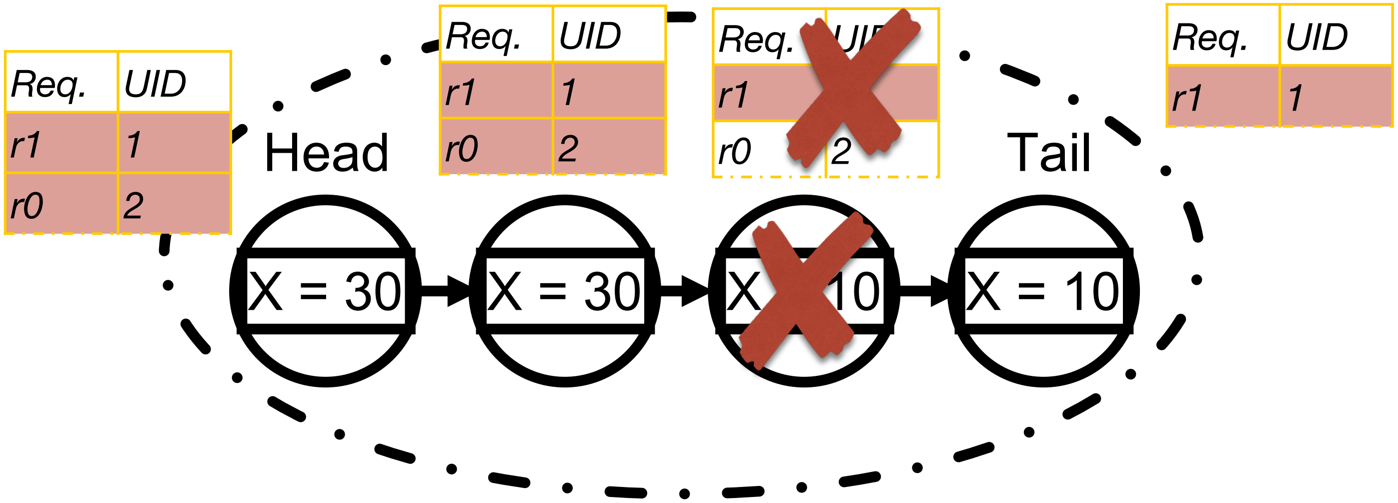
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	1

Fault Tolerance



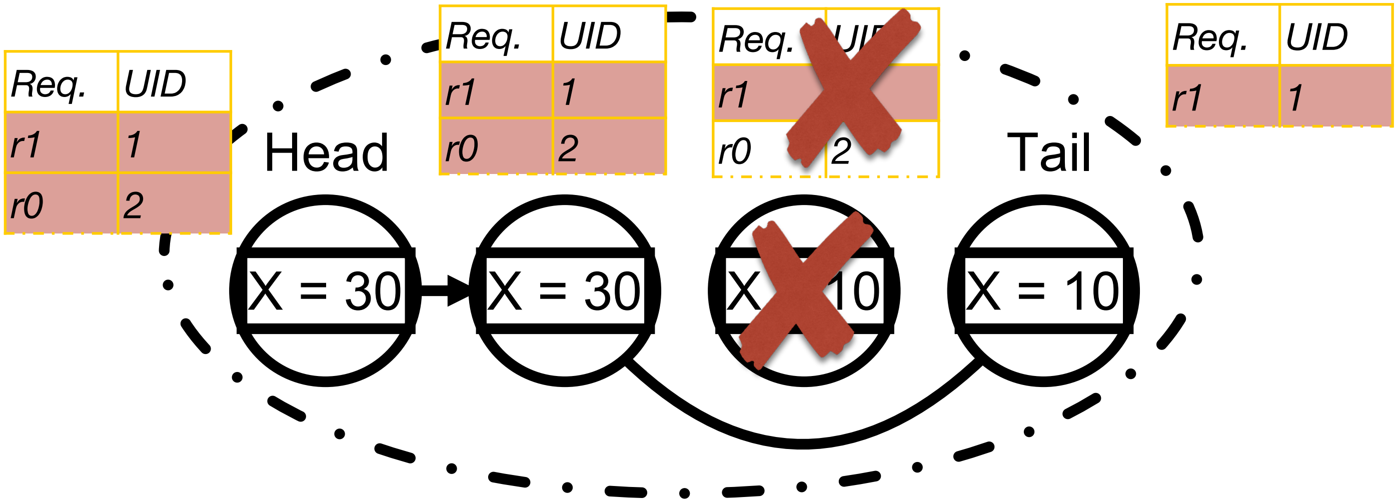
Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UID
r1	[illegible]
r0	2

Req.	UID
r1	1

Fault Tolerance



Req.	UID
r1	*1*
r0	*2*

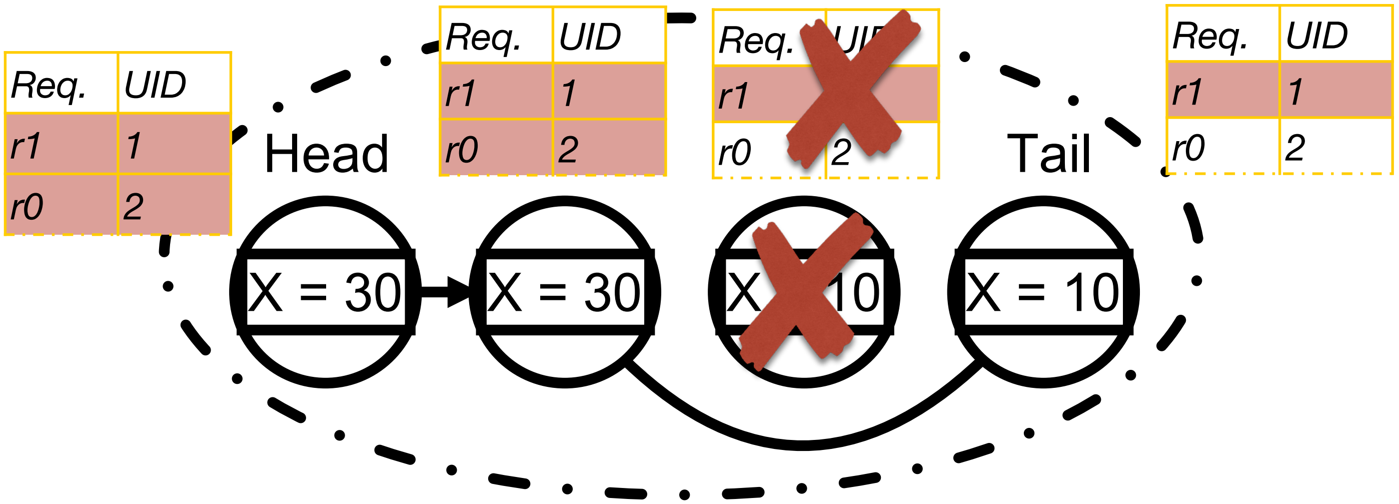
Req.	UID
r1	*1*
r0	*2*

Req.	UID
r1	[illegible]
r0	*2*

Req.	UID
r1	*1*

Need to *re-send r0*

Fault Tolerance



Req.	UID
r1	1
r0	2

Req.	UID
r1	1
r0	2

Req.	UI
r1	[illegible]
r0	2

Req.	UID
r1	1
r0	2

Need to *re-send r0*

How is all of this assignment managed?

Chain Replication
Fault Tolerance

- Trusted Master
  - *Fault-tolerant state machine*
  - Trusted by all replicas
  - Monitors all replicas & issues commands

Chain Replication Fault Tolerance

- Failure cases:
  - Head Fails
    - *Master* assigns 2nd node as Head
  - Tail Fails
    - *Master* assigns 2nd to last node as Tail
  - Intermediate Node Fails
    - *Master* coordinates chain link-up

Chain Replication Evaluation

- Compare to other primary/backup protocols
- Tradeoffs?
  - Latency
  - Consistency
- *Trusted Master*

Conclusions

- Implements the “exercise left to the reader” hinted at by Lamport’s paper
- Provides *some* of the concrete details needed to actually implement this idea
  - But still a fair number of details in real implementations that would need to be considered
  - Chain replication illustrates a “simple” example with fully concrete details
- Does some work to justify why such synchronization might be useful (plane actuators)
- A key contribution that bridges the gap between academia and practicality for SMR