Seminar on

# "*Fault Tolerance in Cloud Computing*"

by

*Prof. Tommaso Cucinotta*
Real-Time Systems Laboratory (ReTiS)
Scuola Superiore Sant'Anna

# History, background & skills

## Let me introduce myself…

- **2000**: MSc in Computer Engineering
  - Thesis: PKCS#11 module for Netscape
- **2004**: PhD in Computer Engineering
  - Interoperability in open-source **smart-card solutions**
  - Open-source **MuscleCard** framework → RedHat CoolKey
- **2004-2012**: Researcher et al. at the **ReTiS**
  - Adaptive scheduling for soft real-time systems
  - Deadline-based scheduler for the Linux kernel for improved responsiveness of soft real-time, multimedia & virtualized services
- **2012-2014**: MTS in **Bell Labs**: research on security and real-time performance of cloud applications (NFV/IMS)
- **2014-2015**
  - SDE in **AWS DynamoDB**: real-time performance and scalability of DynamoDB
- **2016-...**
  - **Associate Professor** at the **ReTiS**
  - Joint UniPi/SSSA MSc degree on Embedded Computing Systems

# AWS DynamoDB

# AWS DynamoDB

## What is DynamoDB

- Fully-managed 24/7 **NoSQL DB** service supporting
  - Single-digit ms latency with **guaranteed read/write throughput**
  - **Elastic growth of tables** up to arbitrary size

## Data model

- A **table** is a collection of **items**, composed of **attributes**
- Primary key
  - **partition hash key**: looked up by exact key (query)
  - optional **sort key**: within each partition, items sorted by sort key

## Other options

- Secondary indexes
- Support for structured JSON contents
- DynamoDB Streams

# AWS DynamoDB

## Performance model

- At table creation time, you specify RCUs and WCUs
  - **R/W beyond the provisioned capacity results in user errors**
- Support for **dynamic change of RCUs and WCUs**
  - Increasing a table capacity, as well as growing it with more and more data, will cause it to split into more and more **partitions**

## Consistency model

- **Consistent read** of <4KB consumes a single RCU
- **Eventually consistent read** of <4KB consumes 0.5 RCU
- Once you get OK (200) from a write, you can safely sleep
  - data is already stored durably on (more) SSD disks

# AWS DynamoDB

## Operations

- "System" composed of zillions of machines
- Geo-distributed: DynamoDB available in ~all AWS DCs
- Tables can grow arbitrarily in size => they continuously split into more and more partitions
- **Failures** continuously happen at all levels
  - Hardware failures (host, power, CPU, memory, network, disk)
  - Software failures (application, libraries, middleware, kernel)
    - <u>replication protocol provably correct</u> in tolerating peers' failures
- **Software upgrades** continuously happen
  - DynamoDB components, as well as thousands of dependencies within the AWS eco-system, need continuous upgrades
- **Operators' mistakes** do happen as well
- <u>Performance and availability SLA under all said conditions</u>

# Failures

## Individual server failure examples

- server unreachable due to network issues
- server down due to unplanned restart (e.g., server crash / killed by OS / by an operator / kernel panic & machine reboot)
- down due to planned maintenance (software upgrade)
  - rolling an upgrade in a region gracefully by sub-clusters takes several hours
  - several services/components continuously patched/upgraded across several geographical regions

## Whole data-center failure example

- power outage
- cooling failure
- network failure (fiber cut)
- natural disasters

# Operations tasks

**Primary task/goal: keep service up & healthy 24/7**

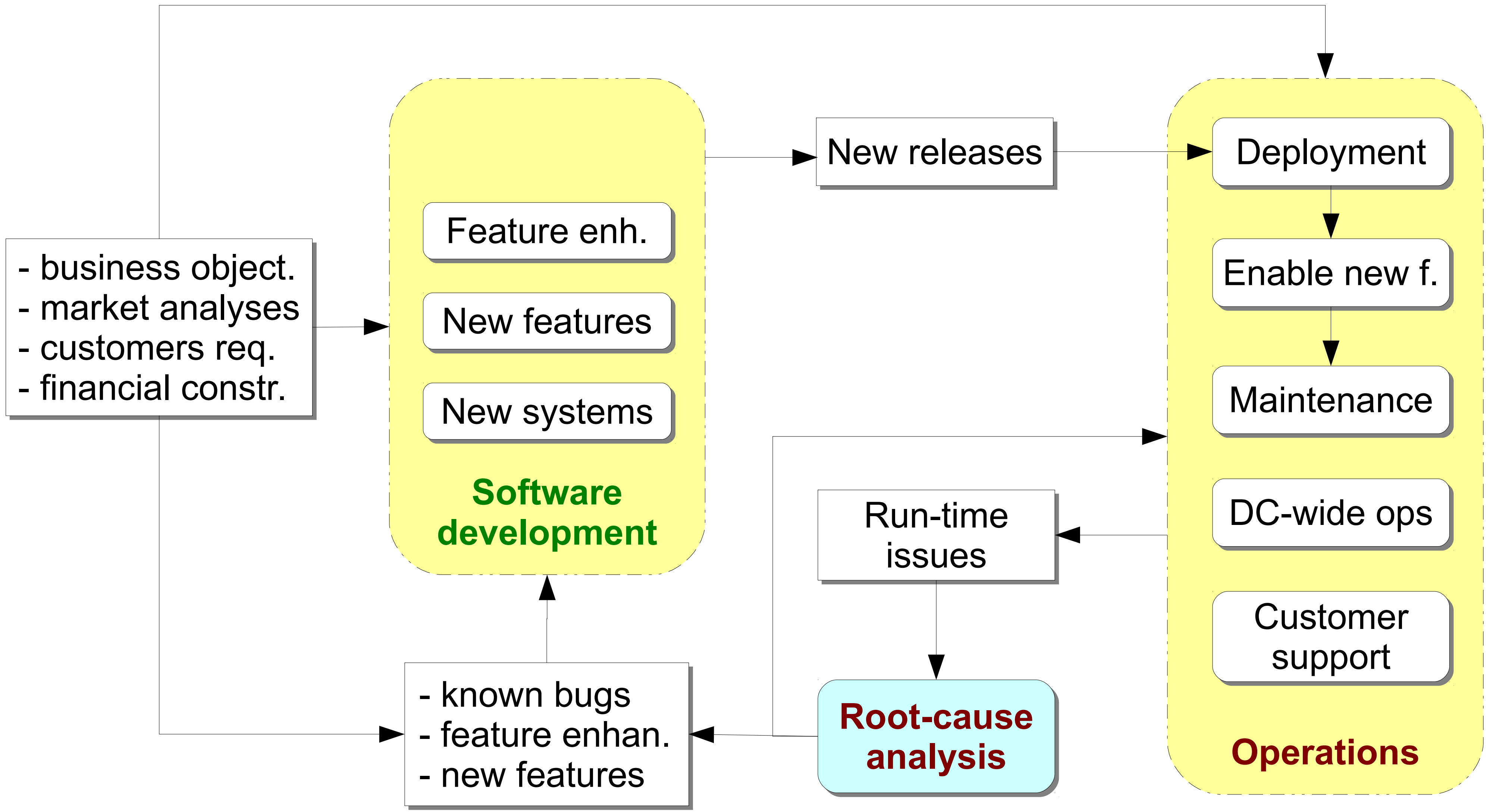
- up == availability
- healthy == satisfying functional & non-functional requirements
  - implies resolve promptly issues reported by customers

**Secondary tasks:**

- **root-cause analysis (RCA)**
- feed information back to developers to reduce operations' burden (mostly hit bugs need urgent fixes)
- **enhance automation** in operations handling

**Escalation path for primary operators to engage secondary ones and experts when needed**

# DevOps Life Cycle



# Operations


## Idealistic view

- fully automated cloud system (infrastructure / service)

## Realistic view

- a number of problems need operators' intervention
- infrastructure is huge => significant operations load

## Operations challenges for big cloud infrastructures

- reduction of operations (human) load
- continuous improvement of automation
  - infrastructure expands as business grows
  - operations workload grows with infrastructure size
  - DC-wide operations (new DC / DC shutdown) need plenty of manual work

# Distributed Consensus

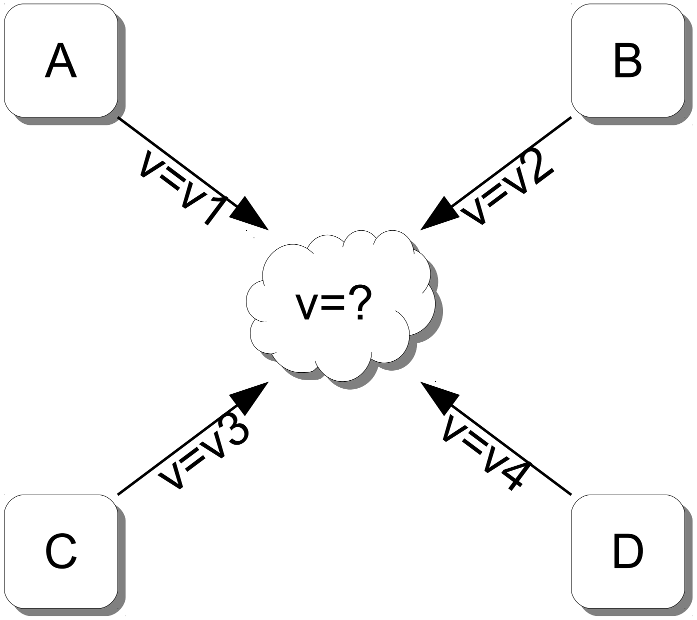
# Distributed Consensus

## The problem

- A set of processes can propose values
- They need to agree on one of the proposed values
- They should learn what value has been chosen

## Safety requirements

- Only one value among proposed ones can be chosen
- A process cannot learn a value that has not been chosen



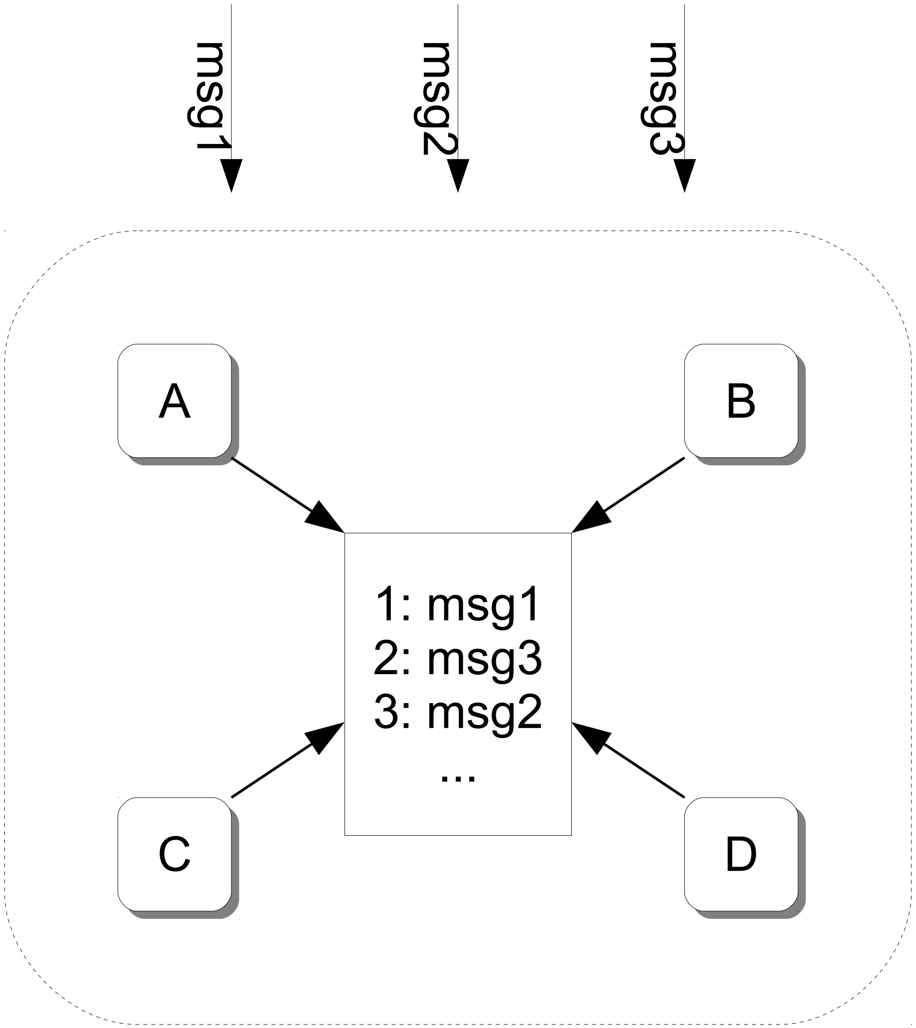
# Distributed Consensus

## Scenario: replicated SM

- SM replicas get messages causing state switches
- SM replicas agree on the sequence of transitions and SM state

## Scenario: replicated DB

- DB replicas get requests
- DB replicas agree on committed transactions and DB contents



# Traditional ACID transaction properties

## Atomicity

- All (commit) or nothing (abort)

## Consistency

- Always consistent

## Isolation

- w.r.t. parallelism, each transaction executes as if isolated; => serializability

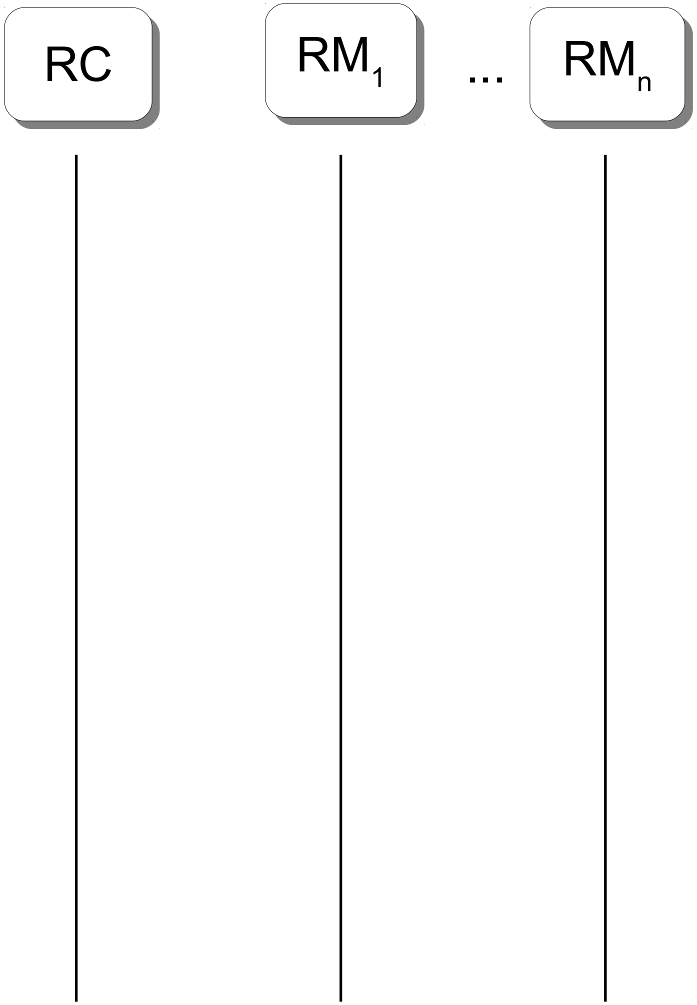
## Durability

- Results are preserved and durable despite failures

# Two phase commit

## 2PC roles

- Resource Coordinator (RC)
- Resource Managers (RMs)



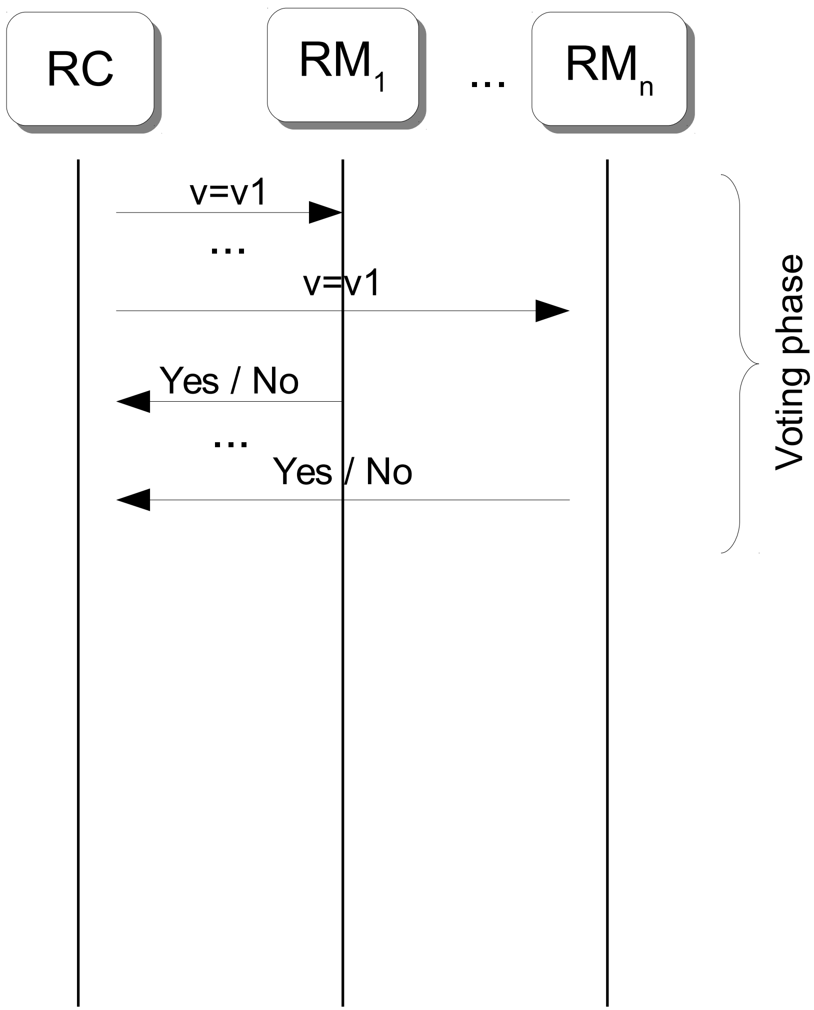
# Two phase commit

## 2PC roles

- Resource Coordinator (RC)
- Resource Managers (RMs)

## 2PC protocol

- **voting phase**: RC suggests a value to all RMs, which respond whether they agree



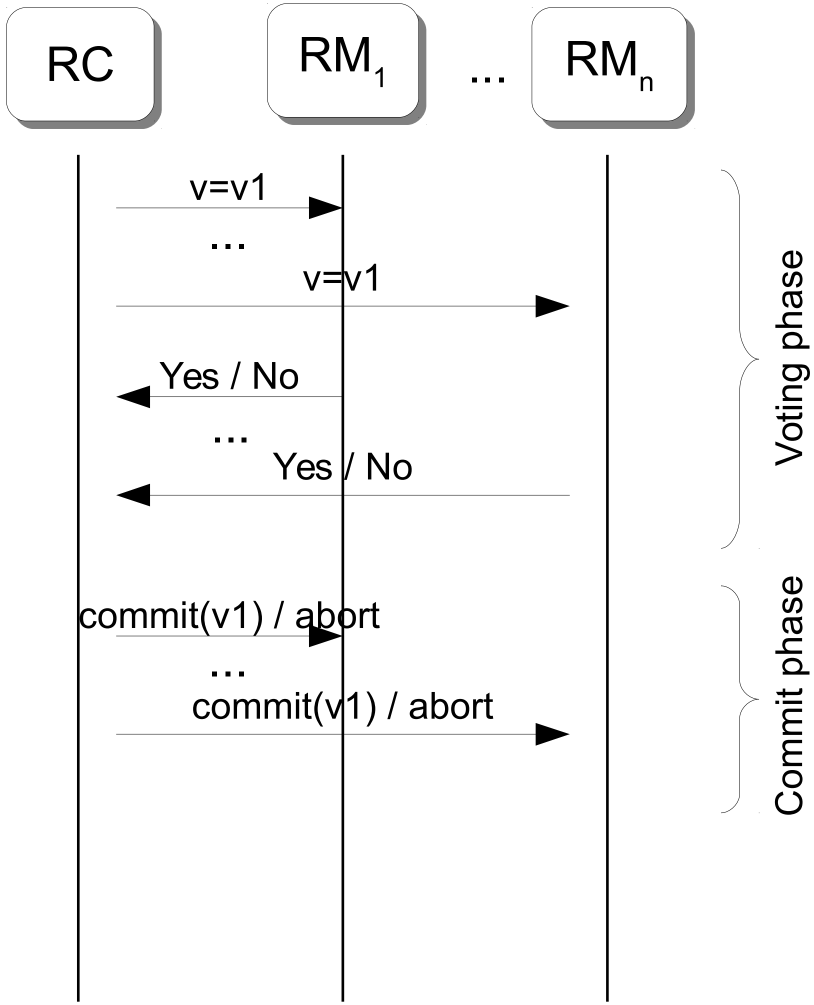
# Two phase commit

## 2PC roles

- Resource Coordinator (RC)
- Resource Managers (RMs)

## 2PC protocol

- **voting phase**: RC suggests a value to all RMs, which respond whether they agree
- **commit phase**: RC sends commit to all RMs if all RMs replied yes (or sends abort otherwise)



# Two phase commit

## 2PC roles

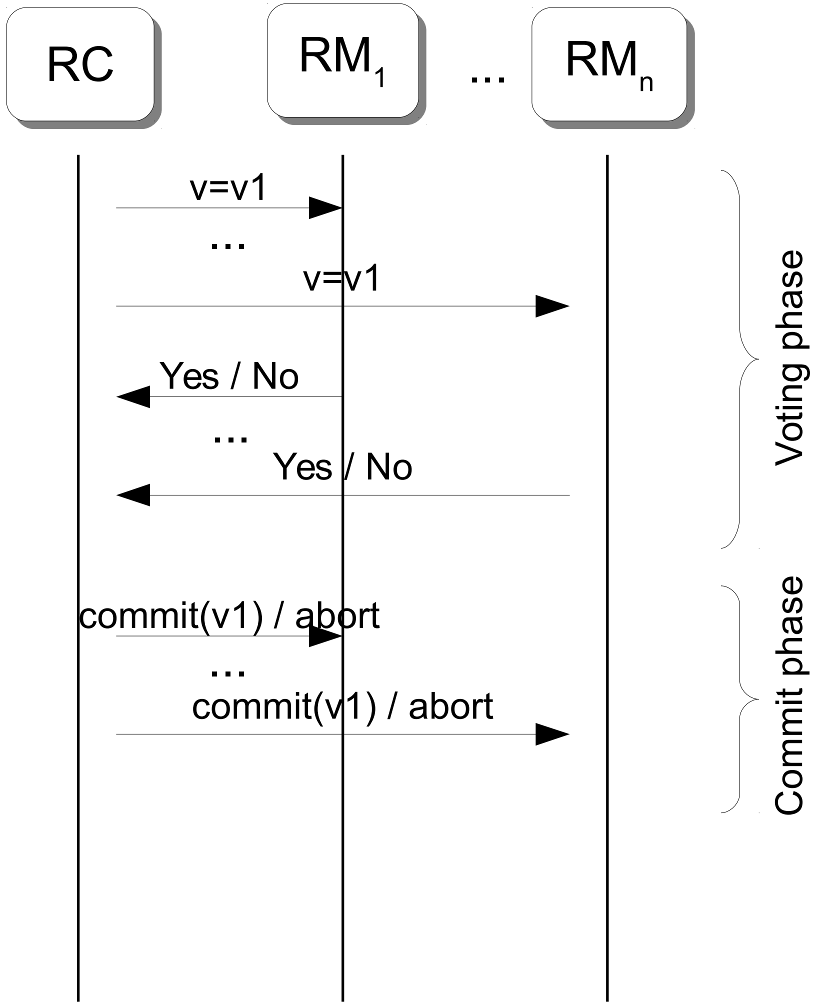
- Resource Coordinator (RC)
- Resource Managers (RMs)

## 2PC protocol

- **voting phase**: RC suggests a value to all RMs, which respond whether they agree
- **commit phase**: RC sends commit to all RMs if all RMs replied yes (or sends abort otherwise)

## Efficiency

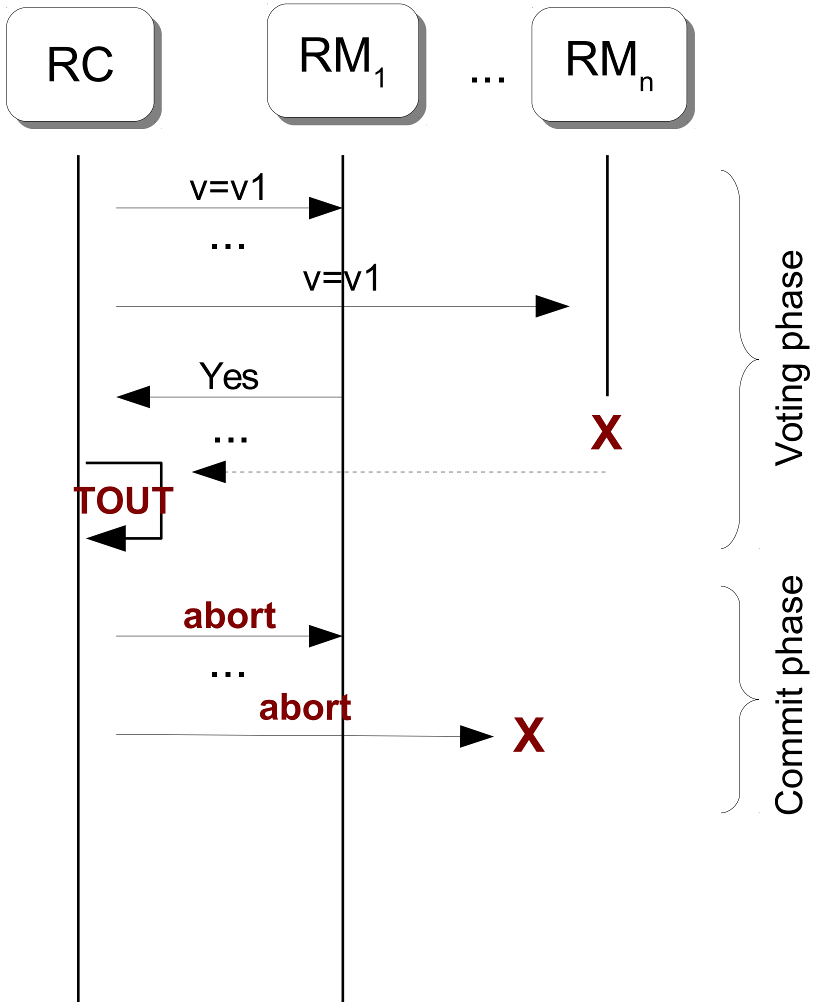
- 3n



# Two phase commit
# Failure scenarios

## RM crashes before replying yes/no
## (or after having replied no)

- RC aborts and notifies



# Two phase commit
# Failure scenarios

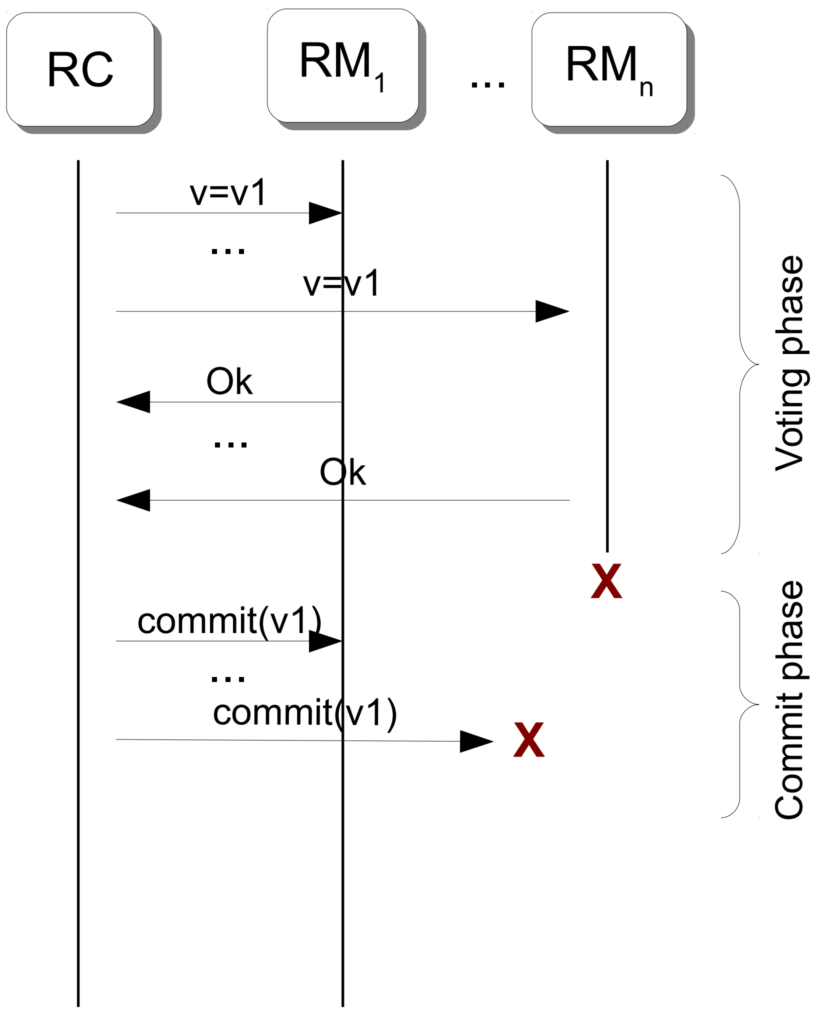
**RM crashes after having replied yes, before having seen the commit/abort**

- RC commits/aborts
- other RMs successfully complete

**Subsequent consensus instances need to restore membership**

- Orthogonal problem

**This is assuming fail-stop**



# Failures

**Fail-stop**

- Power, CPU, motherboard, NIC failure
- Host is shut down, repaired, wiped out and re-enrolled

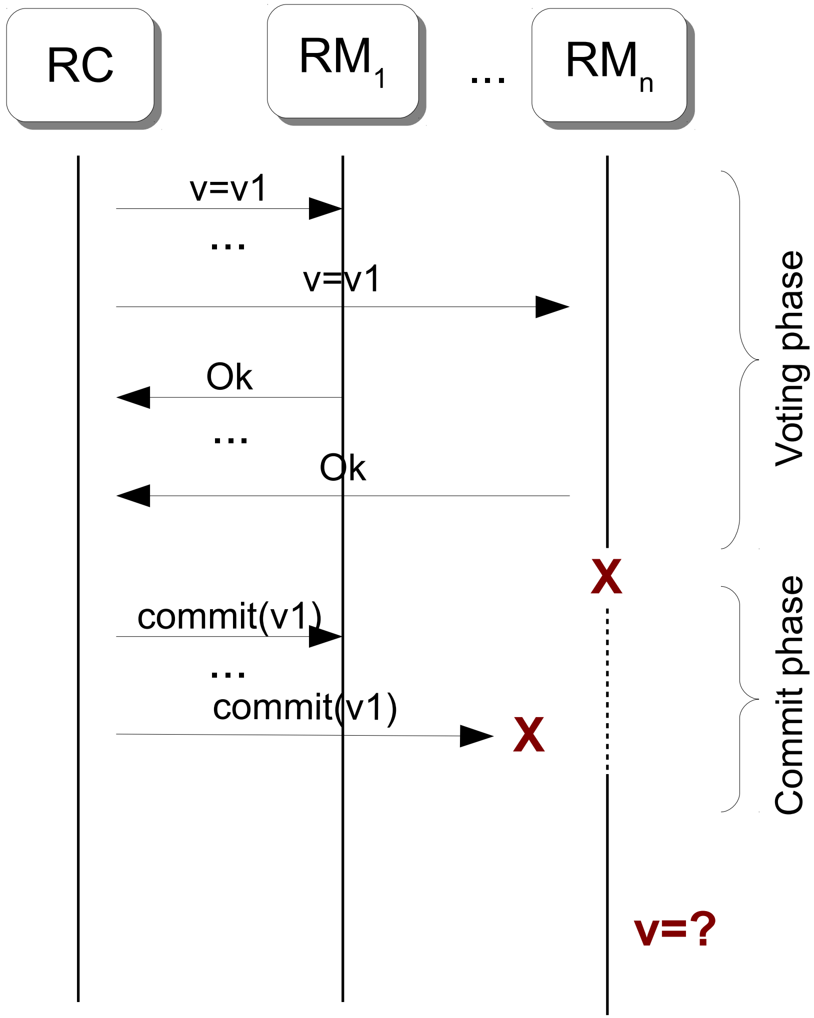
**Fail-recovery**

- Transient network issue
  - Network element crashes and is rebooted
  - Network element fails, backup element takes its place, old primary gets replaced and repaired
  - Network span cut/damaged, traffic is rerouted
  - Software/configuration upgrade, element restarted
- Kernel panic, host is rebooted
- Host becomes temporarily slow due to uncommon overload (software updates, log catch-up, operators' actions, ...)

# Two phase commit
# Failure scenarios

## RM comes back after having replied yes (fail-recover)

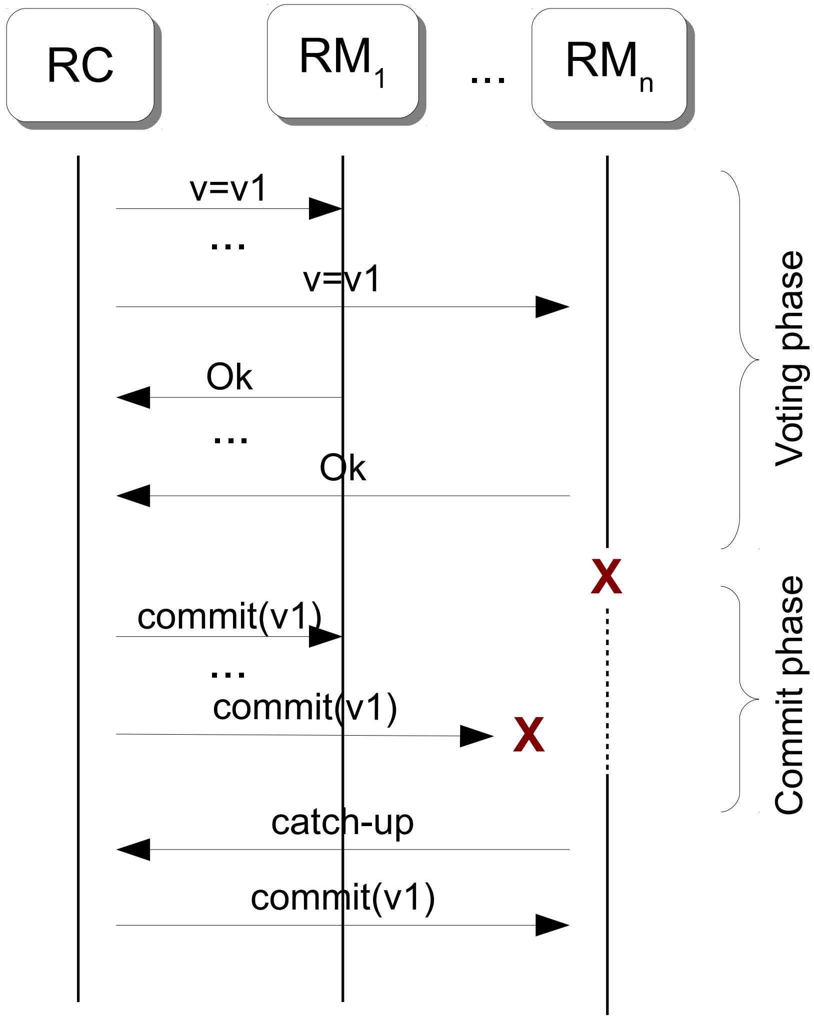
- Commit/abort message is lost :(



# Two phase commit Failure scenarios

## RM comes back after having replied yes (fail-recover)

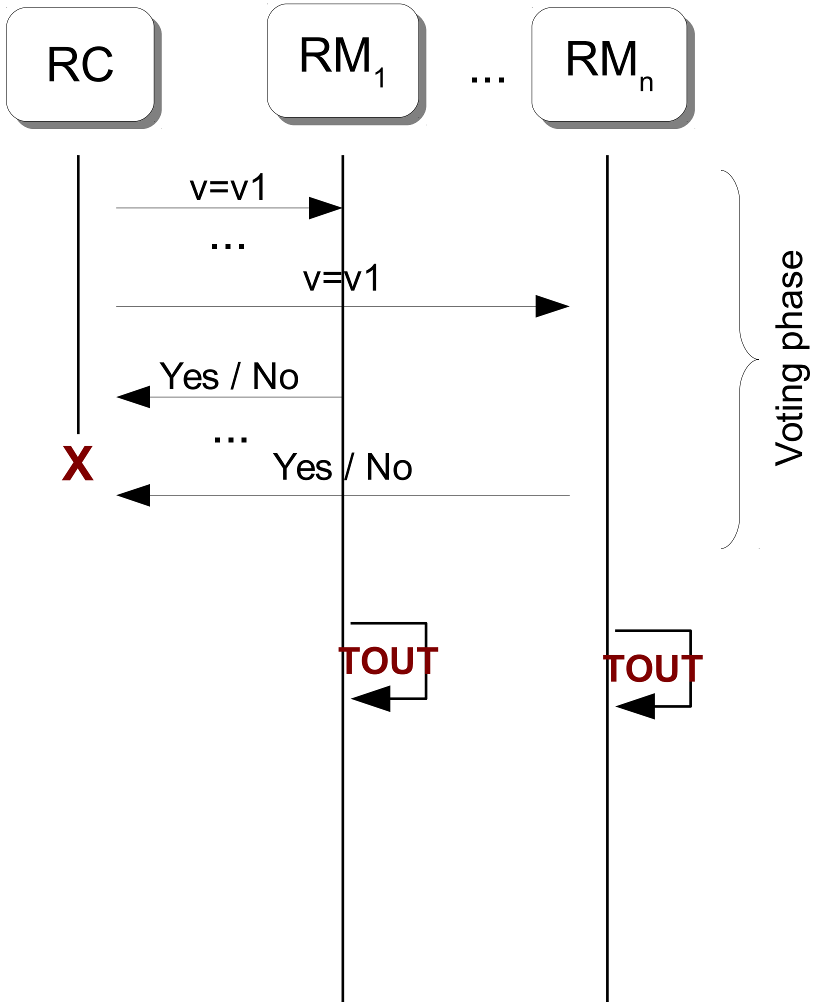
- Commit/abort message is lost :(
- Need for a **catch-up** mechanism, e.g.:
  - RMs persist their Yes/No replies
  - Crashed RM asks back to RC
- When is it safe to discard catch-up info?



# Two phase commit Failure scenarios

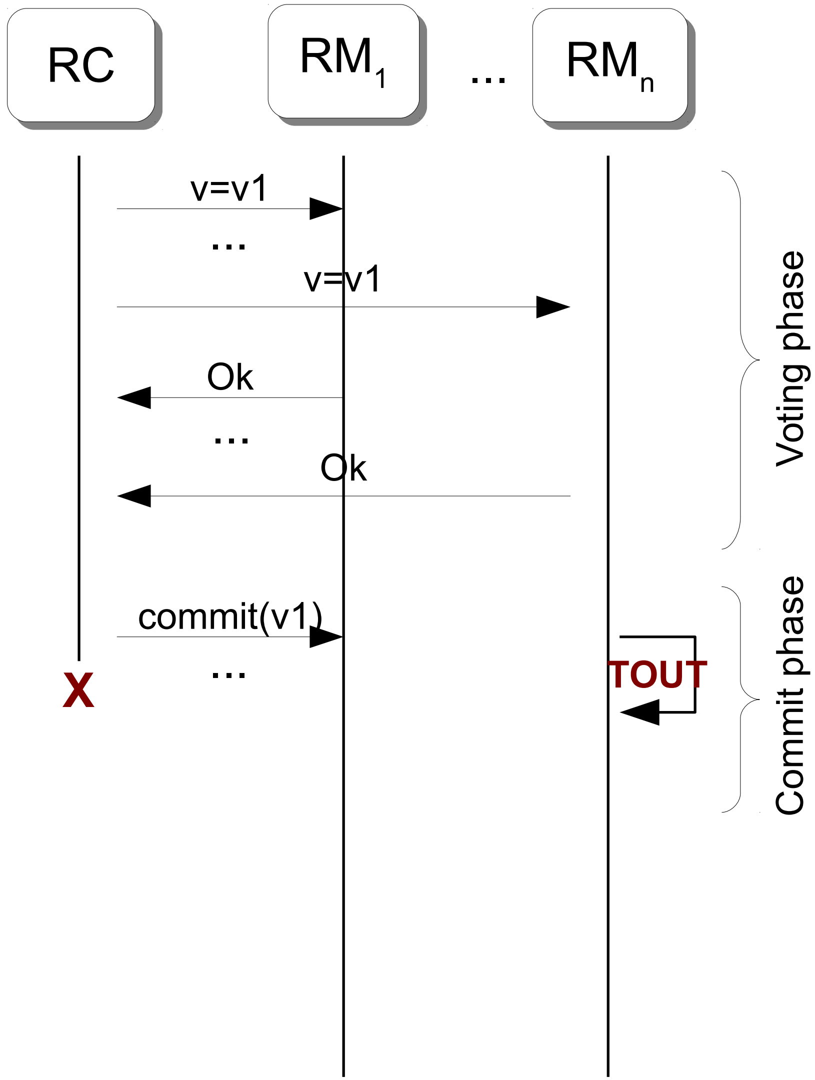
## RC crashes before seeing all replies

- No commit/abort sent
- RMs abort on TOUT



## RC crashes while sending out commits

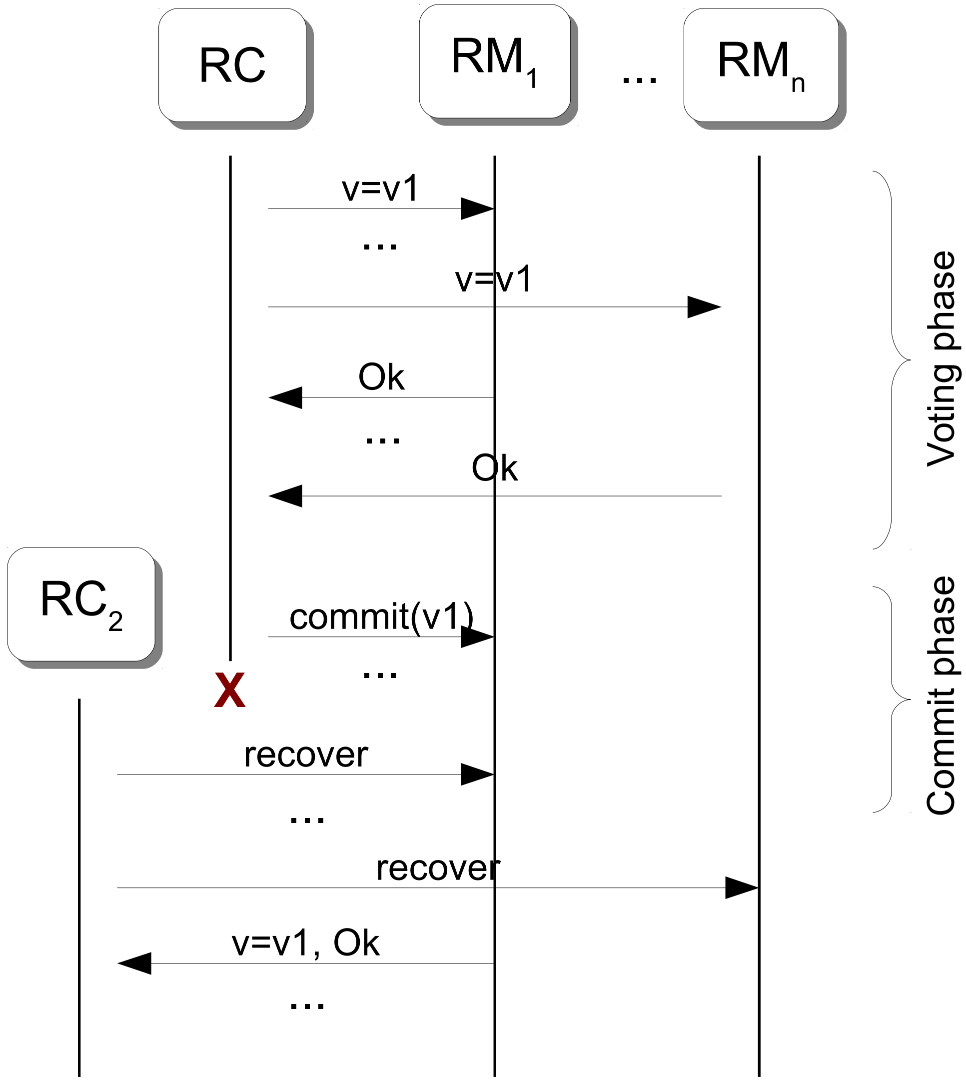
- Problem: some RMs completed the protocol, others no!



# Two phase commit
# Failure scenarios

## RC crashes while sending out commits

- Problem: some RMs completed the protocol, others no!
- Another RC might take over, but needs to query RMs again about their votes
  - When can RMs forget about a completed protocol instance?



# Three phase commit

## 3PC phases

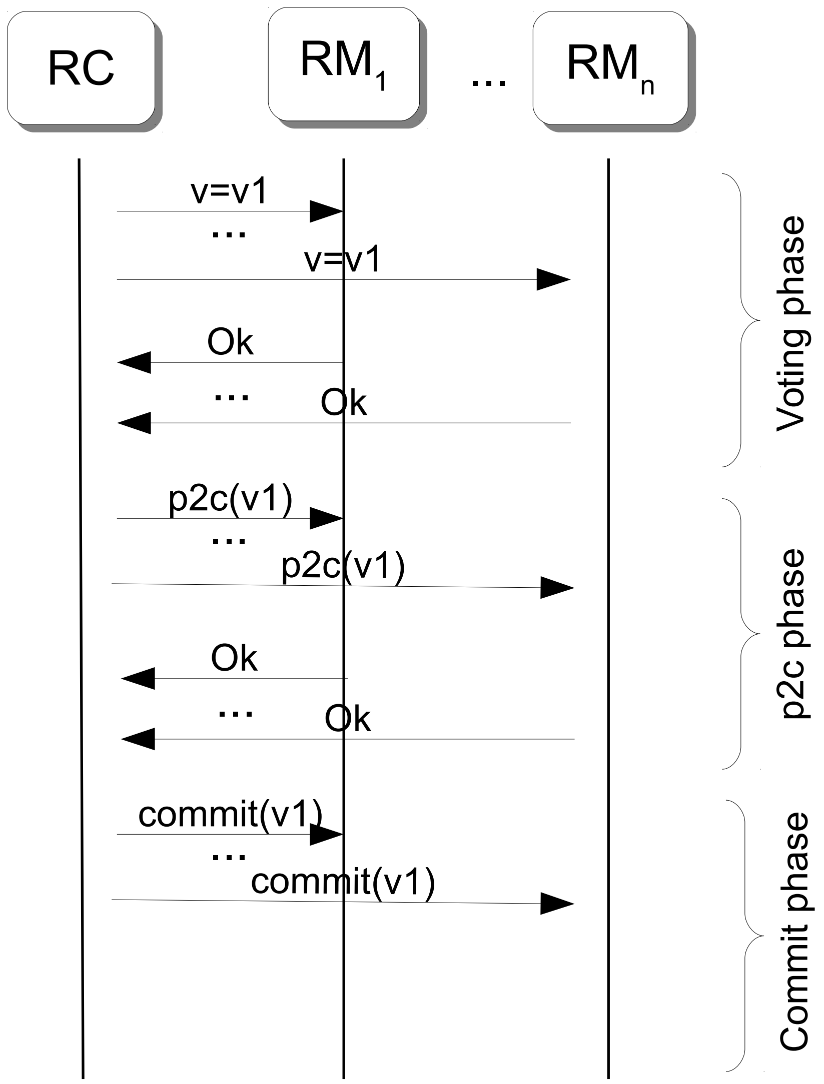
- Voting
- Prepare to commit (p2c)
- Commit

## Key idea

- Each RM is able to take over as RC

## Efficiency

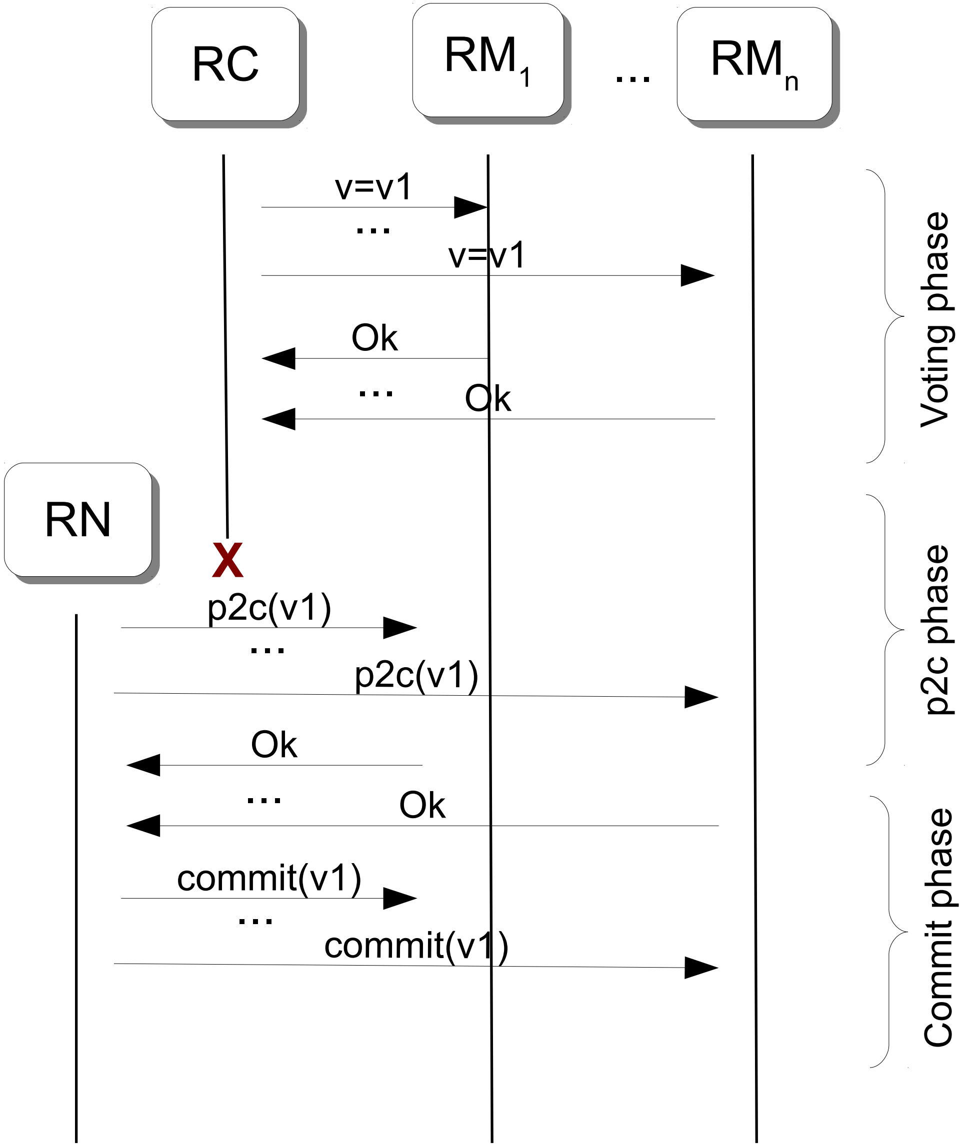
- 5n



# Three phase commit
# Failure scenario

**RC crashes while sending out prepare-to-commit**

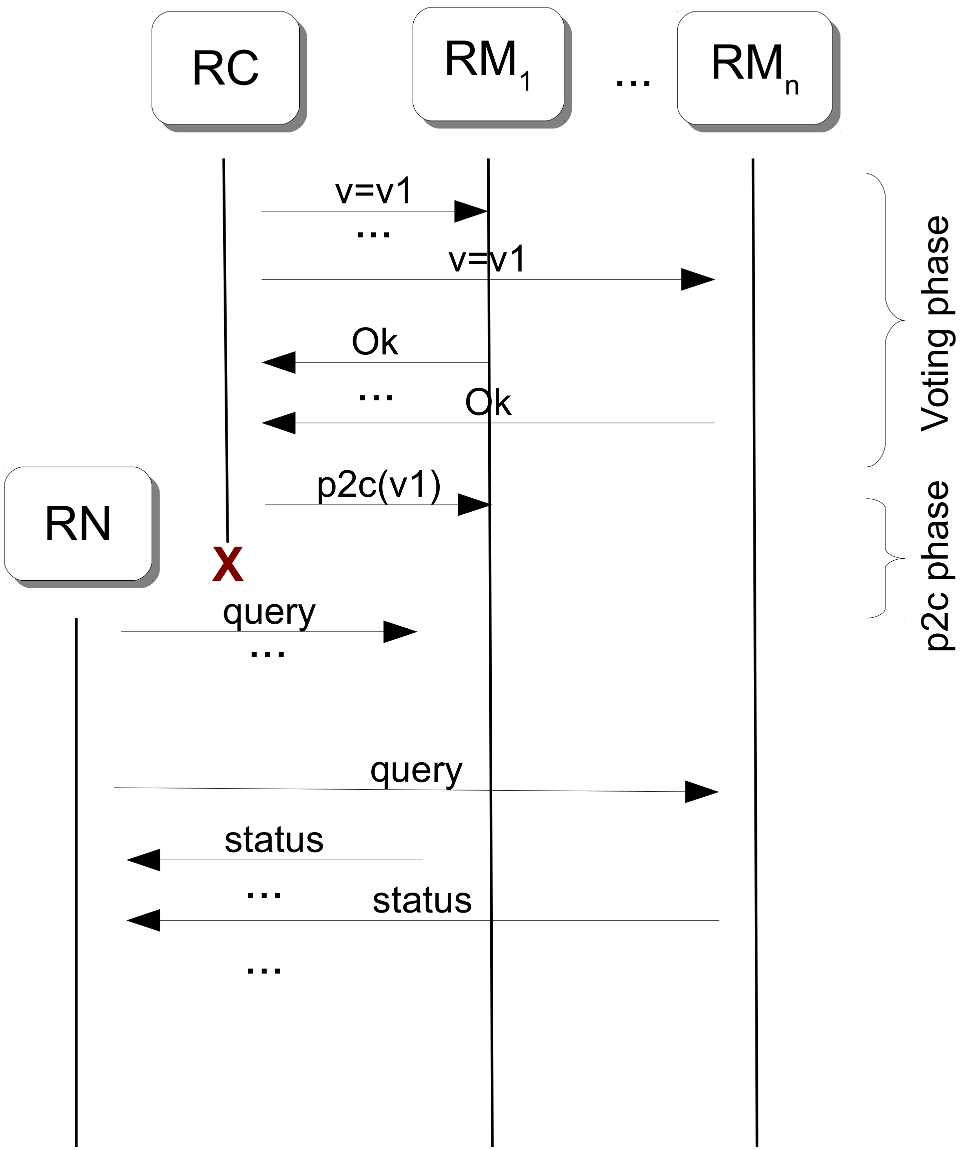
- Any RM can take over as recovery node (RN)
- If RN had received p2c or commit before, it just sends out all p2c and commit to everybody



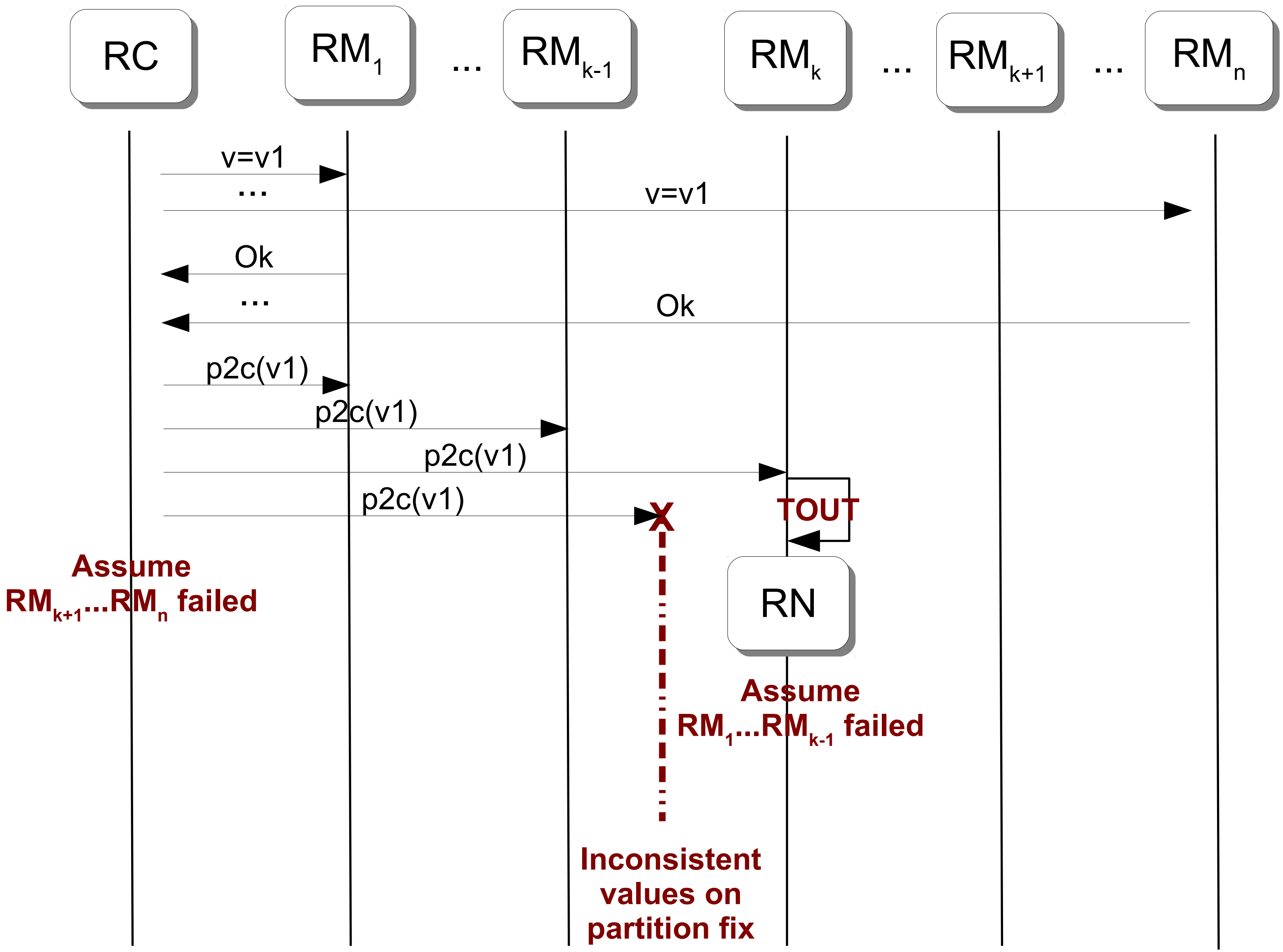
# Three phase commit
# Failure scenario

**RC crashes while sending out prepare-to-commit**

- Any RM can take over as recovery node (RN)
- If RN had seen no p2c nor commit before, it queries all other RMs
  - If any RM has seen a p2c or commit, RN completes the protocol sending out all missing p2c and commit
  - Otherwise, RN either aborts or triggers another vote on v=v1



# Three phase commit
# Network partition problem



# Shortcomings of 2PC/3PC

## 2PC

- tolerates RMs that fail-stop
- 2PC cannot tolerate RC failures

## 3PC

- tolerates RC failures
- less efficient w.r.t. 2PC (5n vs 3n messages)

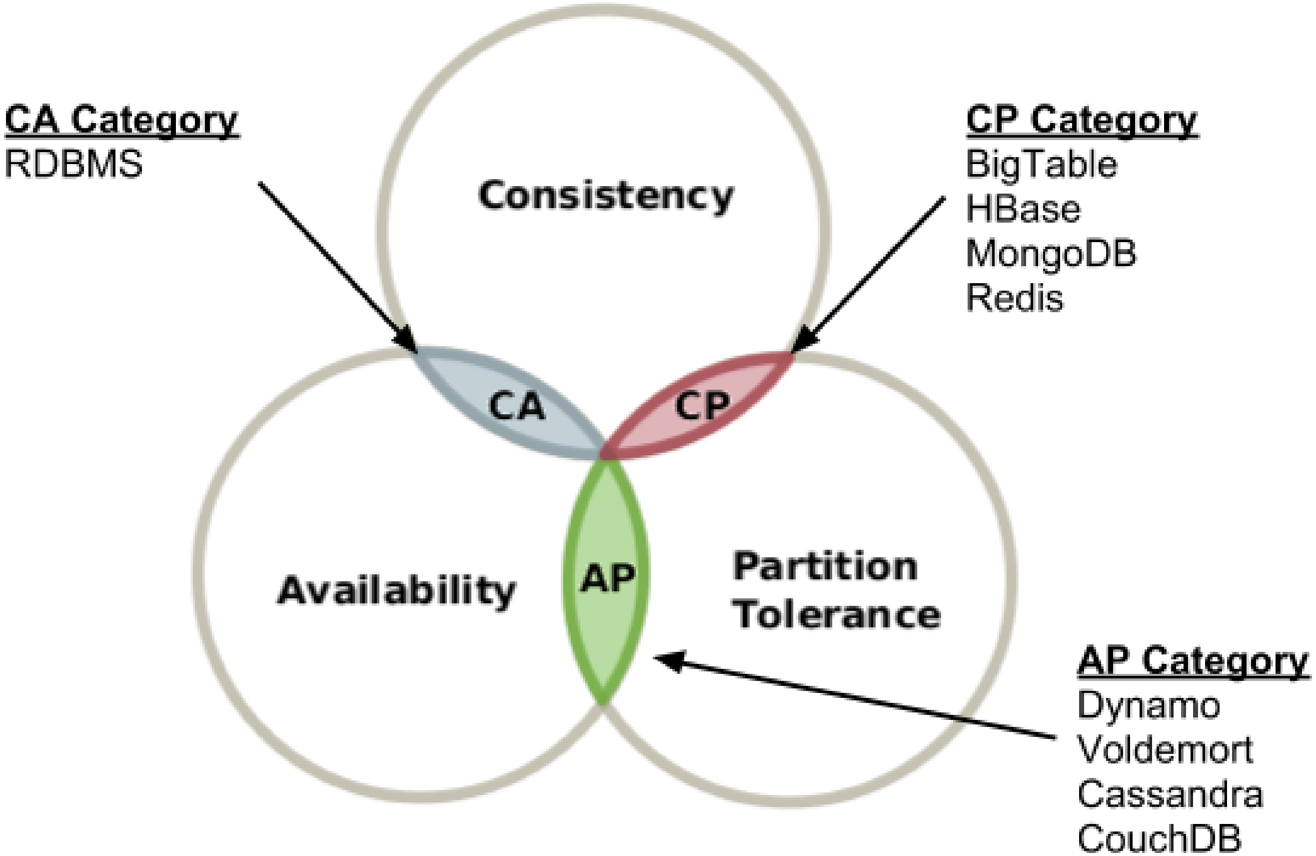
## Both

- cannot tolerate nodes that fail-recover
- cannot tolerate network partitions
- all nodes need to answer: bad with large networks with frequent failures

# CAP Theorem

## E. Brewer, "Towards Robust Distributed Systems," PODC00

"You can only have only two out of Consistency, Availability, Partition tolerance"



# ACID vs BASE

## BASE

- Basically available
- Soft-state (durability is not always needed)
- Eventually consistent

## ACID vs BASE and CAP

https://people.eecs.berkeley.edu/~brewer/PODC2000.pdf

https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

- BASE emphasizes availability over immediate consistency

## More variations on non-strong consistency

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html

- **weak consistency** => inconsistency window
- **eventual consistency**: weak consistency + we go to a consistent state if no further updates
- **causal consistency**: if A notifies B of an update, and B reads, it will see the udpate
- **read-your-writes consistency**: special case of causal consistency where A == B
- **session consistency**: read-your-writes only valid within the same session, if session falls, then no guarantee
- **monotonic read consistency**: once a newer value is observed, no old value can be seen
- **monotonic write consistency**: writes by the same process are serialized (or system impossible to use)

# More consistency models

## Amazon Dynamo consistency [SOSP'07]

- writes for each key coordinated by pre-determined node
  - if it is down, coordination taken over by other node(s)
- label each write with a (node, seq-n) pair (aka, version)
- multiple writes with same node are easily reconciled
  - value with attached the highest version wins
    a.k.a., **syntactic reconciliation** / conflict resolution
- multiple writes coordinated by different nodes are all kept and returned to the client on a get()
  - the client will explicitly resolve the conflict with its next put()
    a.k.a., (client-side) **semantic reconciliation** / conflict resolution

# 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.

## 1 The Problem

**PAXOS (Lamport, 1989)**

### 1.1 The Island of Paxos

Early in this millennium, the Aegean island of Paxos was a thriving mercantile center.[1] Wealth led to political sophistication, and the Paxons replaced their ancient theocracy with a parliamentary form of government. But trade came before civic duty, and no one in Paxos was willing to devote his life to Parliament. The Paxon Parliament had to function even though legislators continually wandered in and out of the parliamentary Chamber.

The problem of governing with a part-time parliament bears a remarkable correspondence to the problem faced by today's fault-tolerant distributed systems, where legislators correspond to processes and leaving the Chamber corresponds to failing.

# PAXOS

## Participants

- **Proposers** propose values
- **Acceptors** accept/reject proposed values
- **Learners** are notified of accepted values

## Each node can play all of the roles at once

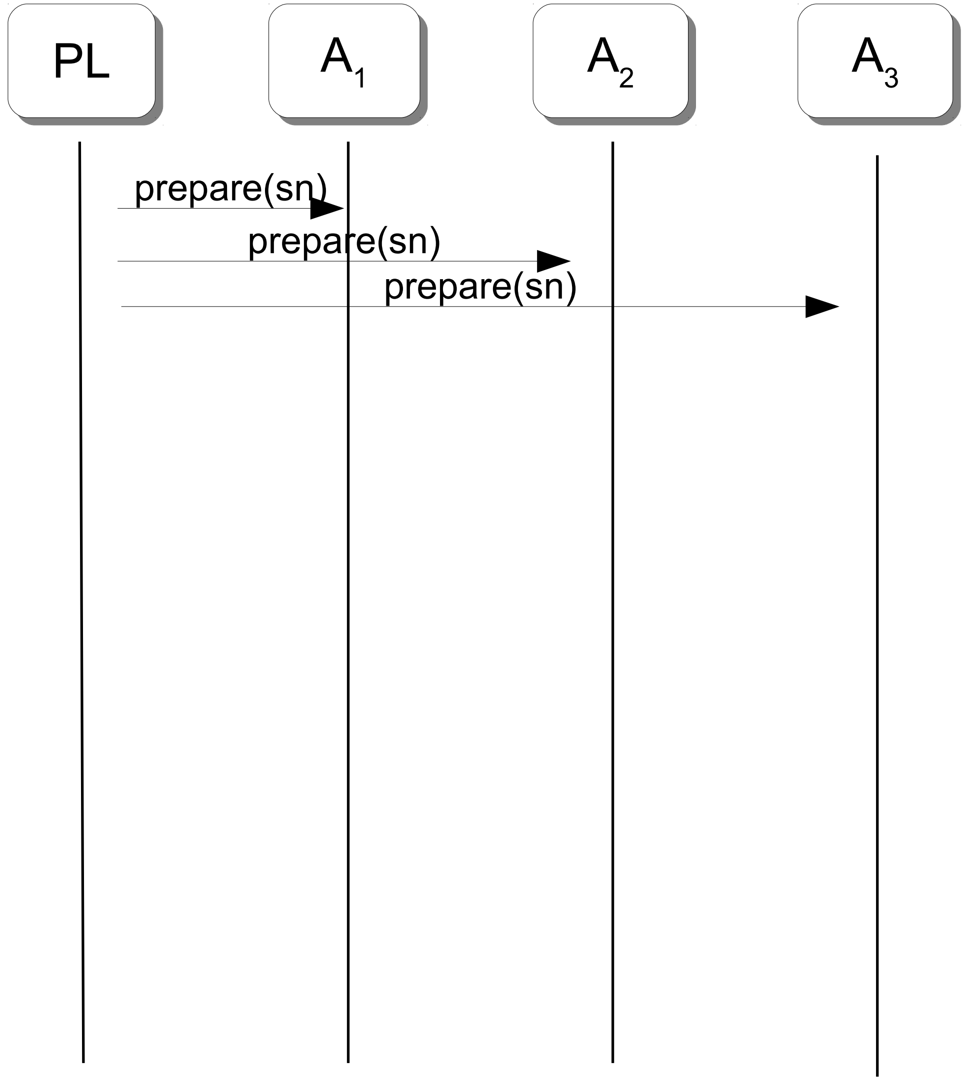
## Possible failures

- Nodes may operate arbitrarily slow, may fail and stop, may fail and recover
- Messages can be arbitrarily delayed, duplicated and lost, but not corrupted

# PAXOS

## Protocol

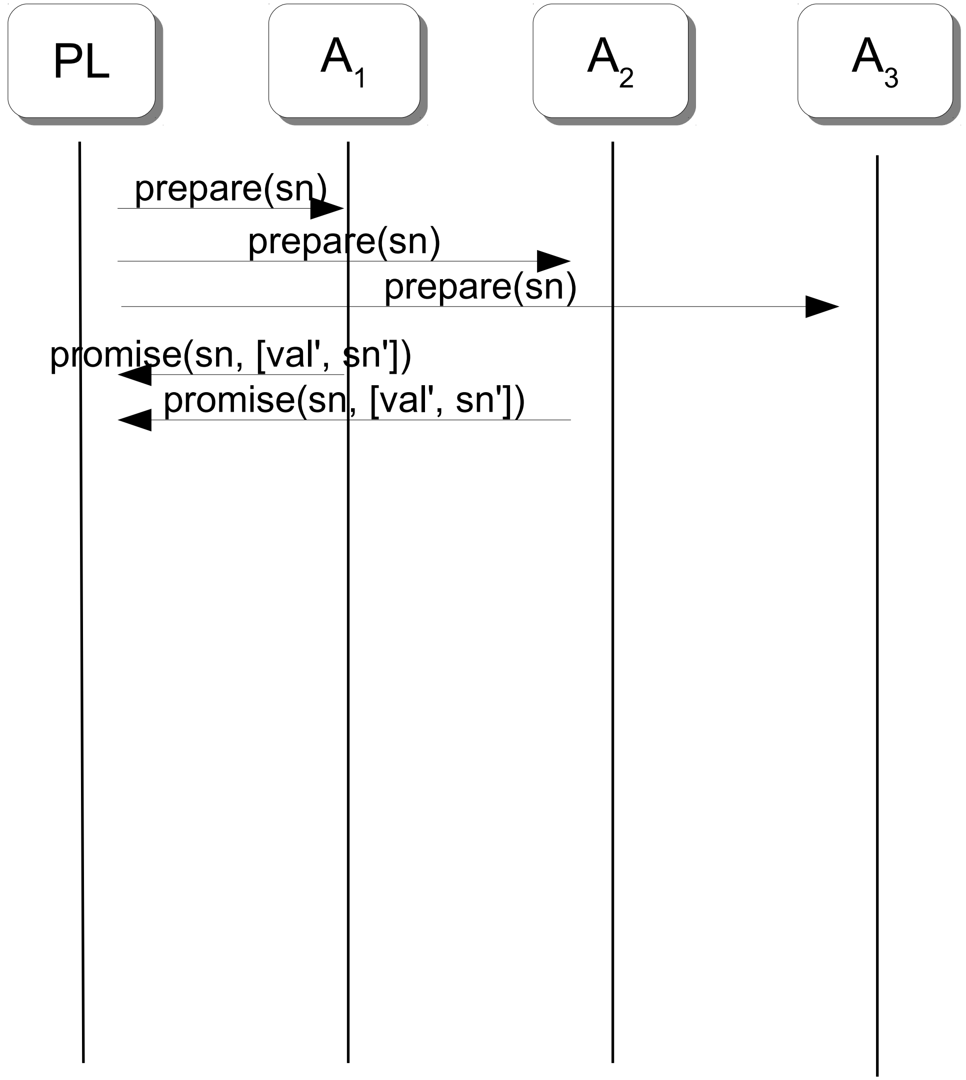
- A proposer acts as **leader**, it sends a unique and ever increasing sequence number (SN) to acceptors in a **prepare** request



# PAXOS

## Protocol

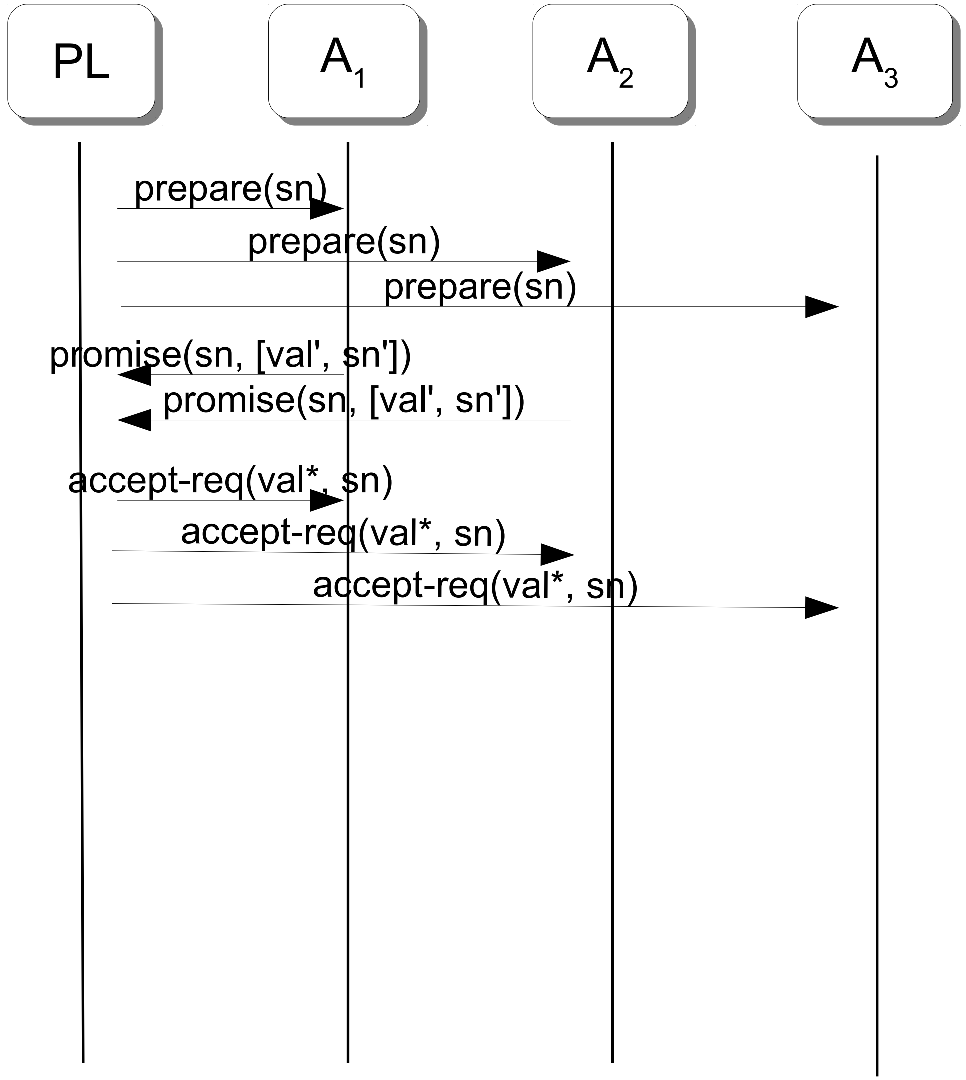
- A proposer acts as **leader**, it sends a unique and ever increasing sequence number (SN) to acceptors in a **prepare** request (**no value** sent now)
- Acceptors reply with a **promise** not to accept any value with lower SN, and include the previously accepted (SN, value) with the highest SN, if any
- Acceptors may ignore prepare requests with a lower SN than prepare requests they already replied to (but rejecting them would be better)



# PAXOS

## Protocol

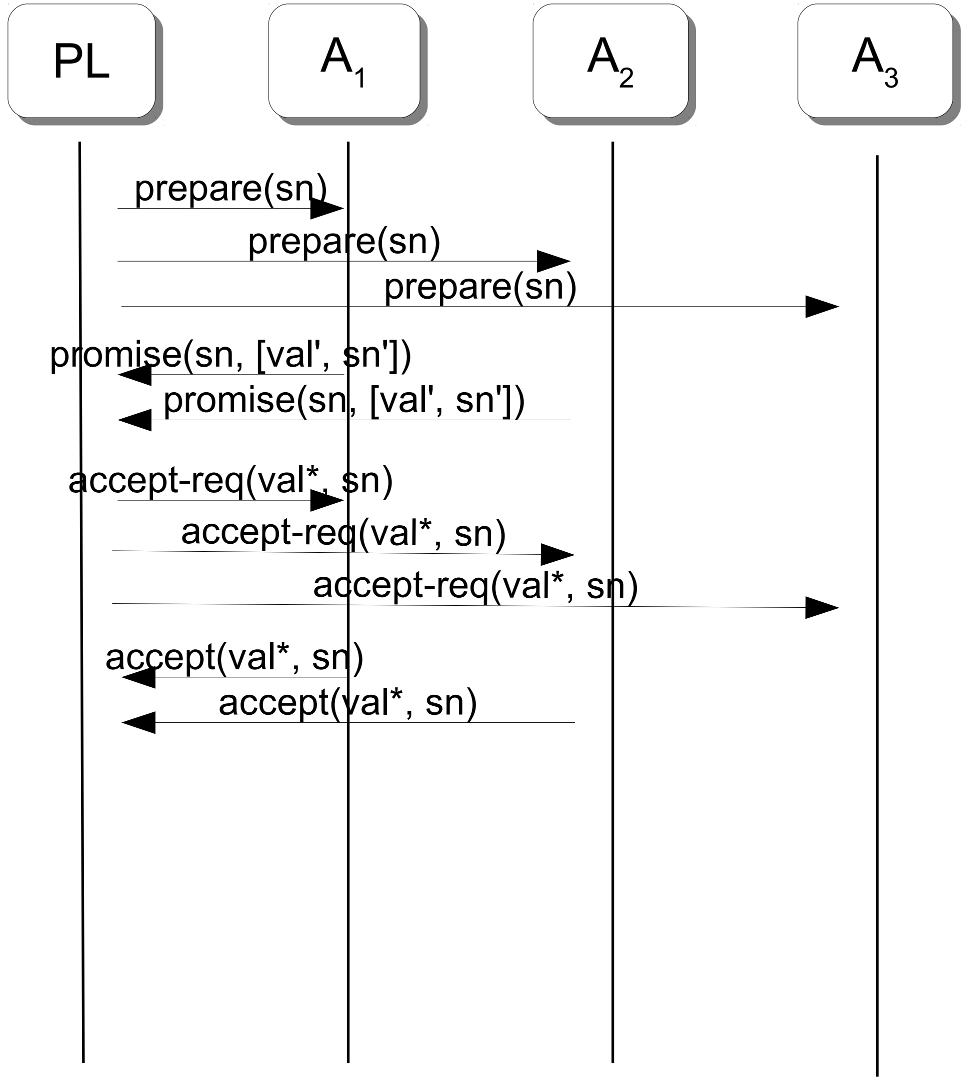
- If a **majority of acceptors** replied not to have accepted a higher SN, the leader continues: it broadcasts an **accept request**, with the highest-SN accepted value got from acceptors if any, or a new value



# PAXOS

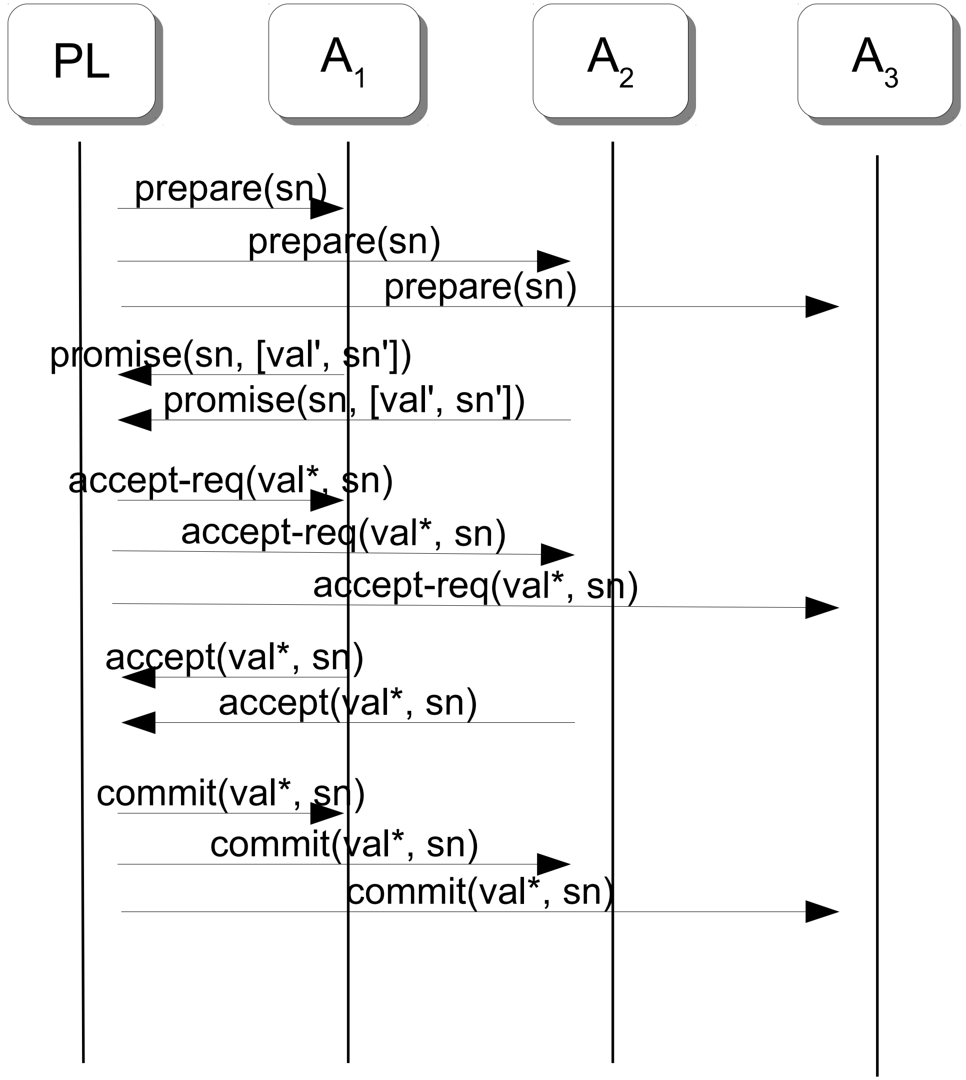
## Protocol

- If a **majority of acceptors** replied not to have accepted a higher SN, the leader continues: it broadcasts an **accept request**, with the highest-SN accepted value got from acceptors if any, or a new value
- Acceptors **accept** a proposal with a given SN iff they have not promised not to (iff they didn't reply to a promise with higher SN)



## Protocol

- If a **majority of acceptors** replied not to have accepted a higher SN, the leader continues: it broadcasts an **accept request**, with the highest-SN accepted value got from acceptors if any, or a new value
- Acceptors **accept** a proposal with a given SN iff they have not promised not to (iff they didn't reply to a promise with higher SN)
- PL sends **commit**



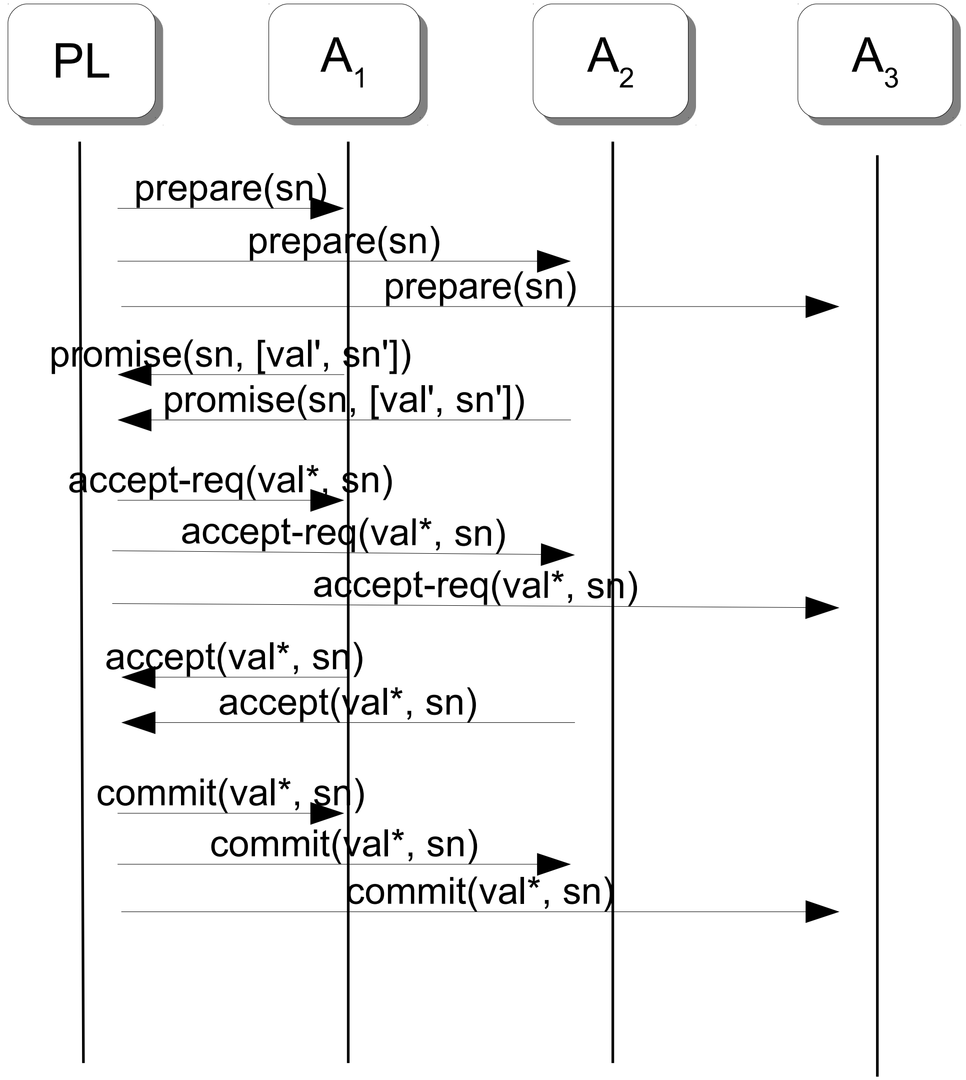
# PAXOS

## No-return (decision) point:

- Majority of acceptors accepted

## Learning accepted values

- #1 Acceptors, whenever accepting, reply to all learners
  - inefficient
- #2 Acceptors notify only a **distinguished learner**, who notifies all learners
  - What if the DL fails?
- #3 Acceptors notify a subset of distinguished learners



# PAXOS

## Notes

- How to change the sets of acceptors? Use PAXOS
- A full run of the protocol would need **5 disk syncs in the critical path** (prepare, promise, accept-req, accept, commit)
- **Chained/multi PAXOS** for enhanced efficiency
  - keep the same PL for as long as possible
  - pack prepare msgs within accept of prior instances
  - down to 1 disk sync (PL can sync after sending accept-req, acceptors need to sync before replying back with accept)
  - batch multiple requests
- **Egalitarian Paxos** (Moraru et al., 2013):
  - diff ops order across replicas unless precedence constraints
  - workload spreads better across replicas

# Thanks!

Questions?

# References

## References


- L. Lamport, *"The part-time parliament,"* TR 49, Digital SRC, 1989
- T. Chandra, R. Griesemer, J. Redstone, *"Paxos made live – an engineering perspective,"* ACM PODC, 2007