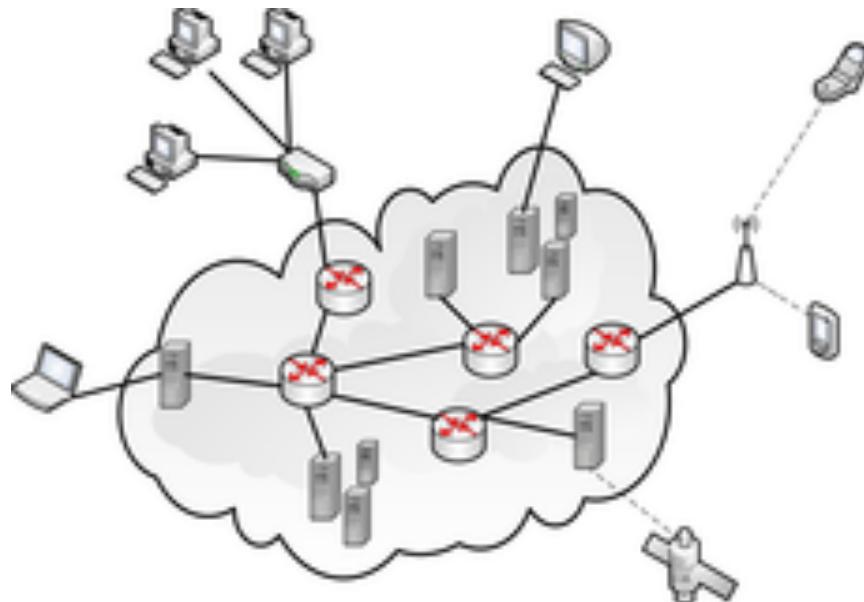


Distributed Systems: Introduction, Motivation & Overview

Hans-Arno Jacobsen

Application & Middleware Systems Research Group

<https://www.i13.in.tum.de/>



In a nutshell

Moodle: <https://www.moodle.tum.de/course/view.php?id=49093>

For slides, information, questions, etc.

Tutorials (starting soon)

About 15 lectures

Lecturer:
Concepts,
principles,
algorithms, etc.

Cloud-DB Lab,
Middleware Course,
Blockchain Seminar

12 Assignments
(starting soon)

Reading material

Exam (in February)

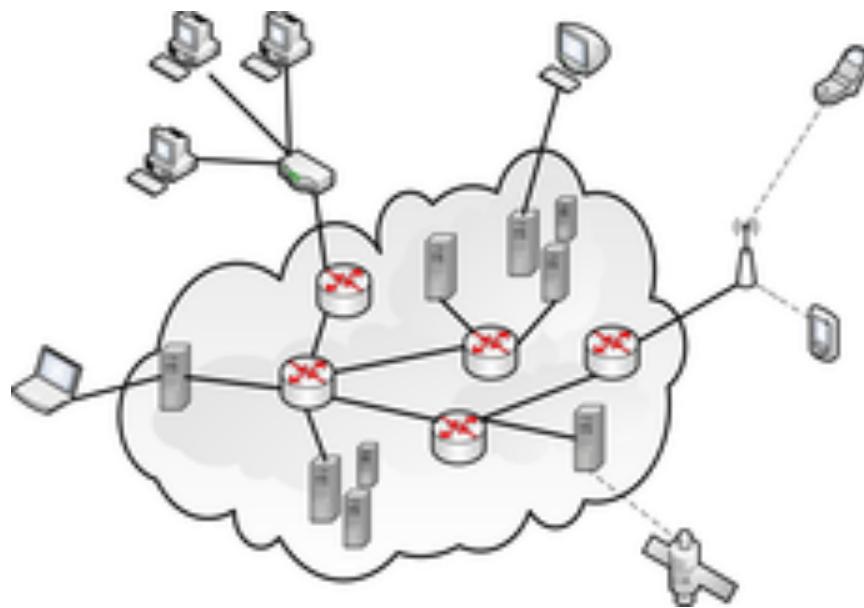
Instructor Team

Dr. Doblander, Pezhman Nasirifard and myself



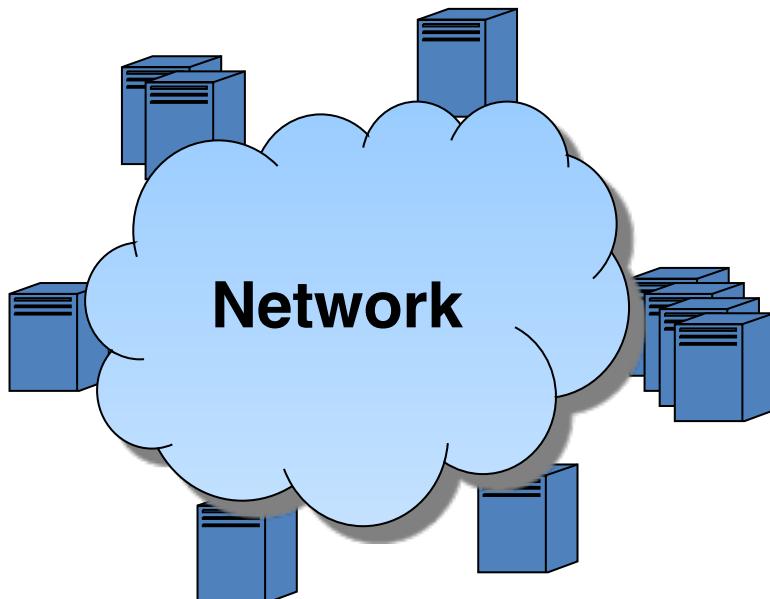
Additional tutors for tutorials

What is a distributed system?



Working definition for this course

A distributed system is a system that is comprised of several **physically disjoint compute resources** interconnected by a **network**.

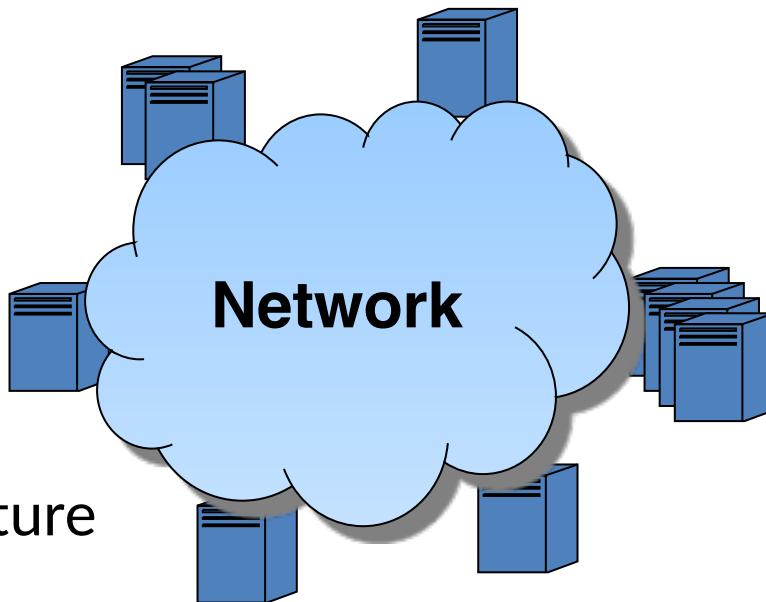


Working definition for this course

A distributed system is a system that is comprised of several **physically disjoint compute resources** interconnected by a **network**.

MapReduce
(Hadoop)

Google infrastructure
(BigTable)



P2P Networks
(Bitcoin, BitTorrent)

World Wide Web
(Akamai CDNs)

Other definitions & views

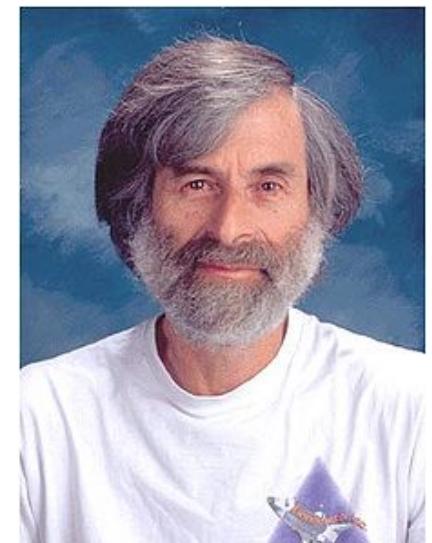
- A distributed system is one in which **hardware or software components** located at **networked computers** communicate and coordinate their actions only by **passing messages**.
 - *By Coulouris et al.*
- A distributed system is a **collection of independent computers** that appears to its users as a **single coherent system**.
 - *By Tanenbaum & van Steen.*

“Introduction to distributed systems design”

- A distributed system is an **application** that executes a collection of **protocols** to **coordinate the actions** of multiple **processes** on a **network**, such that all components **cooperate** together to perform a **single** or small set of **related tasks**.
 - By [Google Code University](#)

Leslie Lamport's anecdotal remark

- “*A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*”
- Father of distributed systems
 - Turing award 2013
 - Inventor of LaTeX



Why build a distributed system?

- Centralized system is simpler in all respects:
 - Local memory, storage
 - Failure model
 - Maintenance
 - Data security

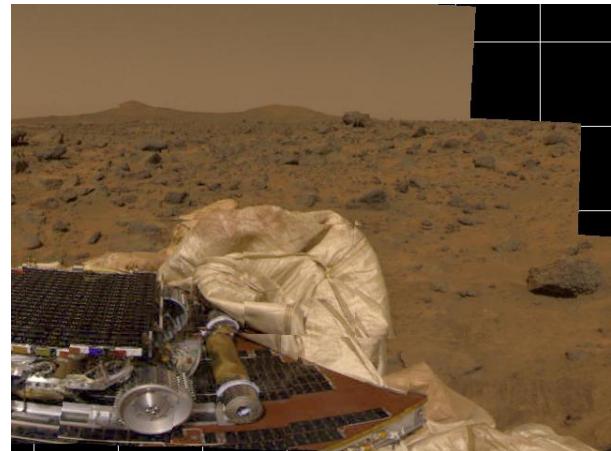


Why build a distributed system?

But...

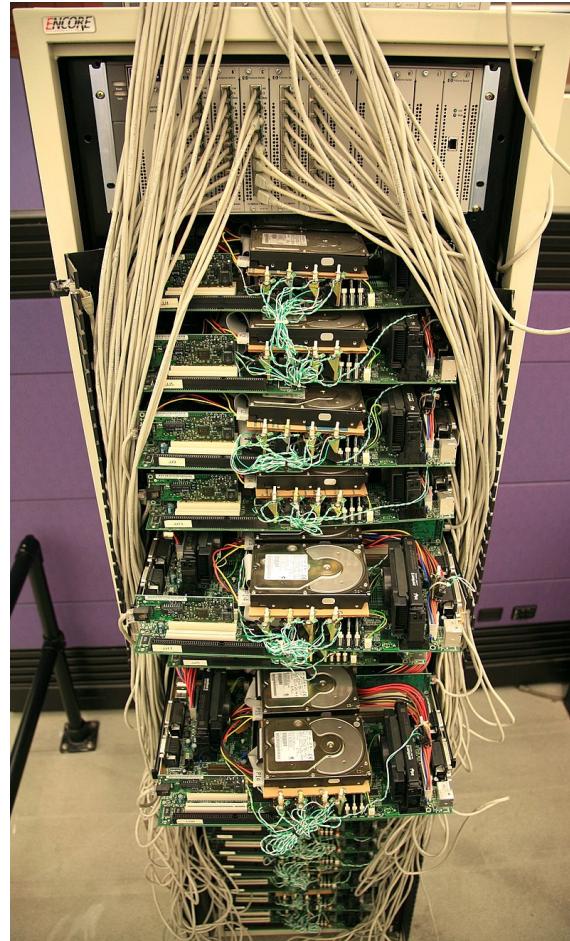
Mars Pathfinder, July 1997

- **Vertical scaling** costs more than **horizontal scaling**
 - Availability and redundancy
 - Single point of failure
-
- Many resources are inherently distributed
 - Many resources used in a shared fashion

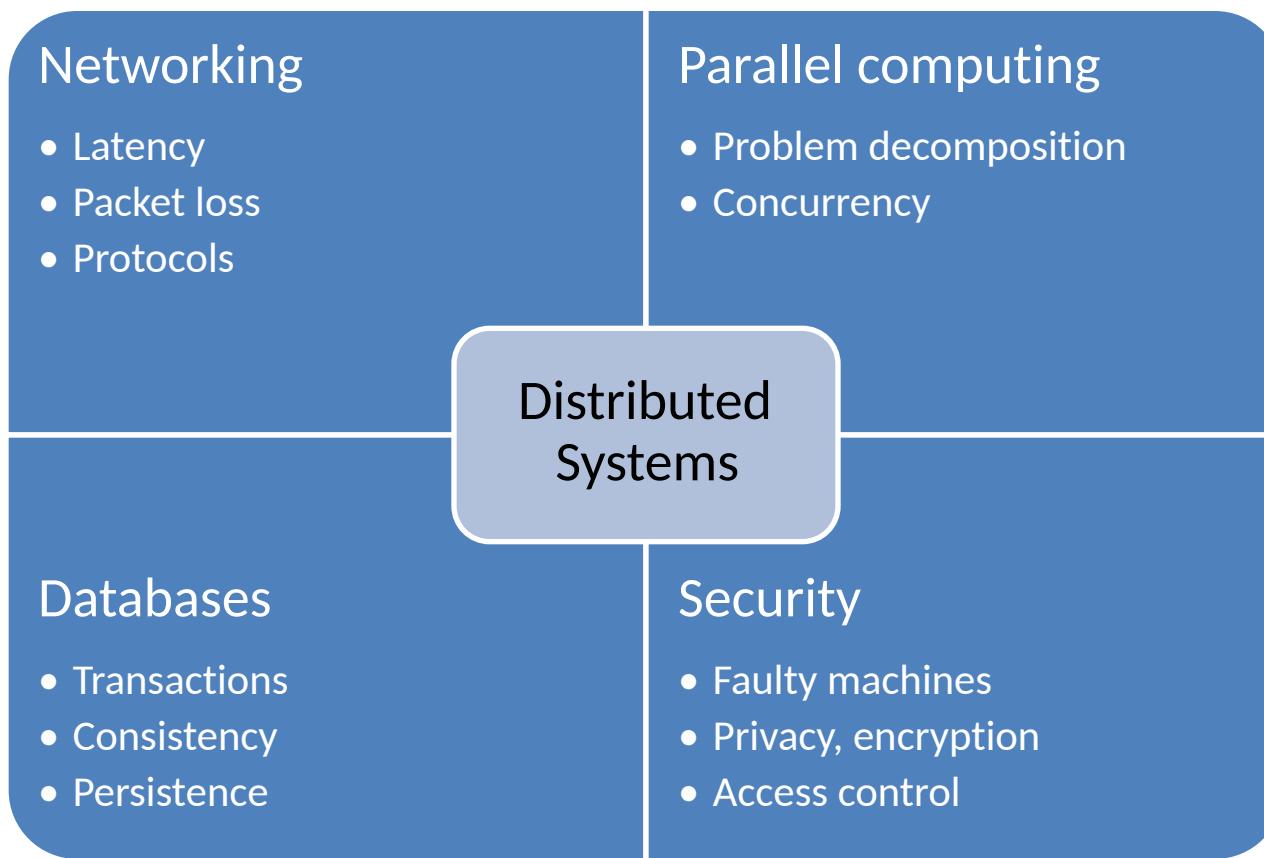


First Google computer, cluster

(<http://www.computerhistory.org/collections/catalog/102662167>)



Related disciplines



Learning objectives

Learning objectives

- Design and develop **architectures for common distributed systems** and applications

Learning objectives

- Design and develop **architectures for common distributed systems** and applications
- Apply **foundational principles** in the development of distributed systems

Learning objectives

- Design and develop **architectures for common distributed systems** and applications
- Apply **foundational principles** in the development of distributed systems
- Understand **properties of common building blocks** applicable to distributed systems design

Learning objectives

- Design and develop **architectures for common distributed systems** and applications
- Apply **foundational principles** in the development of distributed systems
- Understand **properties of common building blocks** applicable to distributed systems design
- Understand the **complexities** involved in developing a distributed system (e.g., machine and network failures, concurrency, etc.)

Learning objectives

- Design and develop **architectures for common distributed systems** and applications
- Apply **foundational principles** in the development of distributed systems
- Understand **properties of common building blocks** applicable to distributed systems design
- Understand the **complexities** involved in developing a distributed system (e.g., machine and network failures, concurrency, etc.)
- Study **advanced distributed systems concepts** for **popular specialized distributed applications**

Today's Agenda: Distributed Systems

- What is a distributed systems?
- Why build distributed systems?
- Why study distributed systems?
- **Characteristics of distributed systems**
- Fallacies of distributed systems design
- Case study: BigTable et al.
- Administrative remarks

Characteristics of distributed systems

- Reliable
- Fault-tolerant
- Highly available
- Recoverable
- Consistent
- Scalable (horizontal)
- Predictable performance
- Secure
- Heterogeneous
- Open

*Also known as
the **ilities***

*(non-functional
requirements)*

*Many of them
still pose
significant
challenges in
theory and in
practice!*

Reliability

- Probability of a system to **perform its required functions under stated conditions for a specified period of time.**
- To run continuously without failure
- Expressed as
Mean Time Between Failure
(MTBF), failure rate



Availability & high-availability

- Proportion of time a system is in a **functioning state**, i.e., can be used, (**1 - unavailable**).
- **Ratio** of time usable over entire time
 - Informally, uptime / (uptime + downtime)
 - System that **can be used 100 hrs out of 168 hrs** has **availability of 100/168**
- Specified as decimal or percentage
 - Five nines is 0.99999 or 99.999% available

Nines or Class of 9

# Nines	Avail. (%)	Downtime per			
		year	month	week	day
1 x 9	90	36.5 d	3 d	16.8 h	2.4 h
2 x 9	99	3.65 d	7.2 h	1.68 h	14.4 mins
4 x 9	99.99	52.56 min	4.32 min	60.48 s	8.64 s
5 x 9	99.999	5.256 min	25.92 s	6.048 s	864 ms
6 x 9	99.9999	31.536 s	2.592 s	604.8 ms	86.4 ms
9 x 9	99.9999999	31.536 ms	2.592 m s	604.8 μ s	86.4 μ s

Nines or Class of 9

- Formerly used for telecommunication systems et al.
- Favorite marketing term
- Does not capture impact or cost of downtime

*“According to Google, its Gmail service was **available** 99.984 percent of the time in 2010 ...” by P. Lilly*

Between **52** and **5.2 minutes** downtime per year.

Availability is not equal to reliability

- System going down 1 ms every 1 hr has an availability of more than 99.9999%
 - Highly available, but also highly unreliable
- A system that never crashes, but is taken down for two weeks
 - Highly reliable, but only about 96% available

Fallacies of distributed systems design

- **Assumptions** (novice) designers of distributed systems often make **that turn out to be false**
- Originated in 1994 by Peter Deutsch, Sun Fellow, Sun Microsystems
- Also see “*A Note on Distributed Computing*”
- **The 8 fallacies**
 - The network is reliable.
 - Latency is zero.
 - Bandwidth is infinite.
 - The network is secure.
 - Topology doesn't change.
 - There is one administrator.
 - Transport cost is zero.
 - The network is homogeneous

Today's Agenda: Distributed Systems

- What is a distributed systems?
- Why build distributed systems?
- Why study distributed systems?
- Characteristics of distributed systems
- Fallacies of distributed systems design
- **Case study: BigTable et al.**
- Administrative remarks

CASE STUDY OF A LARGE-SCALE DISTRIBUTED SYSTEM: BIGTABLE

Key-value stores

- *What is a key-value-store?*
- *Why is a key-value store needed?*
- Key-value-store client interface
- Key-value stores in practice
 - Common features
 - Common non-features
 - Apache HBase
 - Apache Cassandra
 - Google Spanner

What mechanisms make them work?

What are key-value stores?

- Container for key-value pairs (Databases)
- Distributed, multi-component, systems
- NoSQL semantics (non-relational)
- KV-Stores offer **simpler semantics** in exchange for **increased scalability, speed, availability, and flexibility.**



DBMS (SQL)

Students Table	
Student	ID *
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table		
ID *	Activity *	Cost
084	Swimming	\$17
084	Tennis	\$36
100	Squash	\$40
100	Swimming	\$17
182	Tennis	\$36
219	Golf	\$47
219	Swimming	\$15
219	Squash	\$40

Key-value store

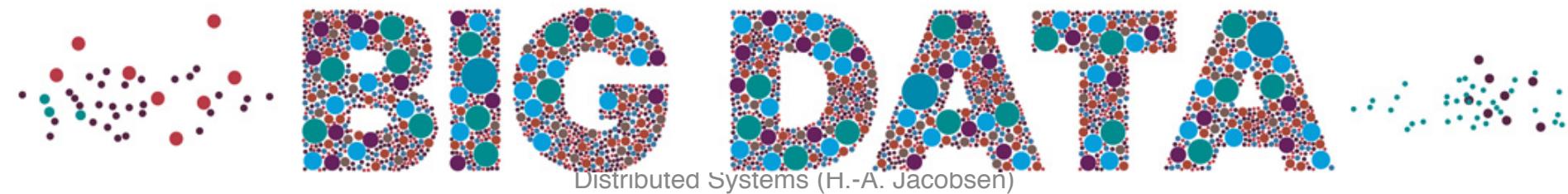
Key	Value
John Smith	{Activity:Name=Swimming}
Jane Bloggs	{Activity:Cost=57}
Mark Anthony	{ID=219}

- Relational data schema
- Data types
- Foreign keys
- Full SQL support

- No data schema
- Raw byte access
- No relations
- Single-row operations

Why are key-value stores needed?

- Today's internet applications
 - Huge amounts of stored data ($1 \text{ PB} = 10^{15} \text{ bytes}$)
 - Huge number of Internet users (e.g., 3.4 billion)
 - Frequent updates
 - Fast retrieval of information
 - Rapidly changing data definitions
- Ever more users, ever more data



Why are key-value stores needed?

- Horizontal scalability
 - User growth, traffic patterns change
 - Adapt to number of requests & data size
- Performance
 - High speed for single-record read and write operations
- Flexibility
 - Adapt to changing data definitions
- Reliability
 - Thousands of components at play
 - Uses commodity hardware: failure is the norm
 - Provide failure recovery
- Availability and geo-distribution
 - Users are worldwide
 - Guarantee fast access

Key-value store client interface

- Main operations
 - Write/update **put**(key, value)
 - Read **get**(key)
 - Delete **delete**(key)
- Usually no aggregation, no table joins, no transactions!

Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");
```

```
HTable table = new HTable(conf, „MyBaseTable”);
```

```
Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes(„value”), Bytes.toBytes(200));
table.put(put);
```

```
Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value”));
System.out.println("Value: " + Bytes.toInt(val));
```

Hbase: Key-value store client interface

Initialization
Using
ZooKeeper

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");
```

```
HTable table = new HTable(conf, „MyBaseTable”);
```

```
Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes(„value”), Bytes.toBytes(200));
table.put(put);
```

```
Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes(„value”));
System.out.println("Value: " + Bytes.toInt(val));
```

Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");
```

Initialization
Using
ZooKeeper

```
HTable table = new HTable(conf, „MyBaseTable“);
```

Column Family:
“Schema”

```
Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("value"), Bytes.toBytes(200));
table.put(put);
```

```
Get get = new Get(Bytes.toBytes("key1"));
```

```
Result result = table.get(get);
```

```
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes("value"));
```

```
System.out.println("Value: " + Bytes.toInt(val));
```

Hbase: Key-value store client interface

```
Configuration conf = HBaseConfiguration.create();
conf.set("hbase.zookeeper.quorum", "192.168.127.129");
```

Initialization
Using
ZooKeeper

```
HTable table = new HTable(conf, „MyBaseTable“);
```

Column Family:
“Schema”

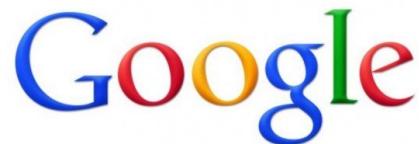
```
Put put = new Put(Bytes.toBytes("key1"));
put.add(Bytes.toBytes("colfam1"), Bytes.toBytes("value"), Bytes.toBytes(200));
table.put(put);
```

Column:
Defined at run-time
(“wide column” stores)

```
Get get = new Get(Bytes.toBytes("key1"));
Result result = table.get(get);
byte[] val = result.getValue(Bytes.toBytes("colfam1"), Bytes.toBytes("row1"));
System.out.println("Value: " + Bytes.toInt(val));
```

Key-value store in practice

- BigTable
- Apache HBase
- Apache Cassandra
- Redis
- Amazon Dynamo
- Yahoo! PNUTS



Common elements of key-value stores

(Covered in detail in subsequent lectures)

- Failure detection & failure recovery (failure models)
- Replication
 - Store multiple copies of data
- Memory store & write ahead log (WAL)
 - Keep data in memory for fast access
 - Keep a commit log as ground truth
- Versioning
 - Store different versions of data
 - Timestamping (logical clocks)

Common features of key-value stores

- Flexible data model with column families
- Horizontal partitioning (key ranges)
- Store big chunks of data as raw bytes
- Fast column-oriented single-key access (read, write, update)

Common non-features

- Integrity constraints
- Transactional guarantees, i.e., ACID
 - cf. *CAP theorem and distributed transactions*
- Powerful query languages
- Materialized views

Google

YAHOO!

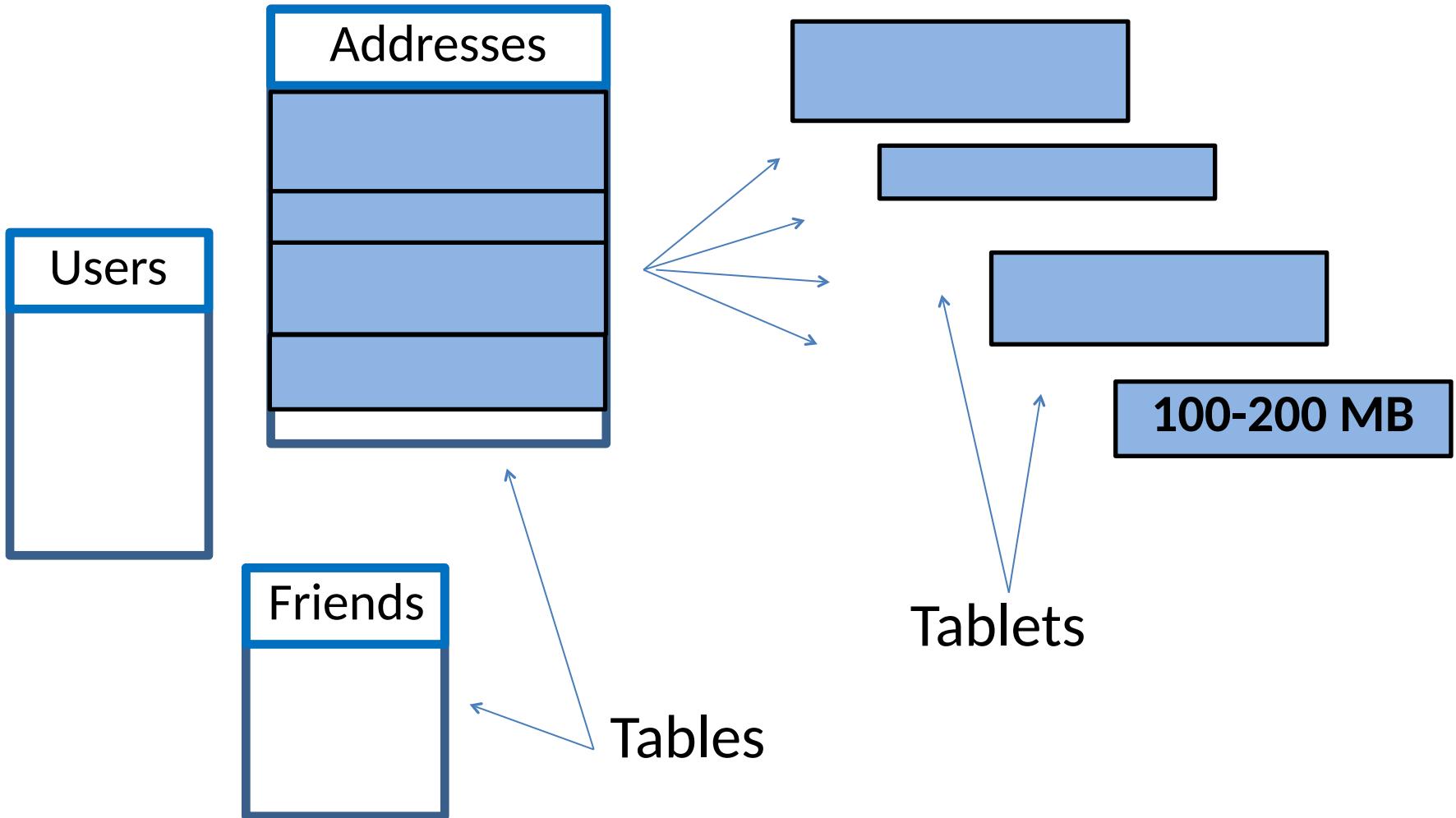


BIGTABLE / HBASE

BigTable

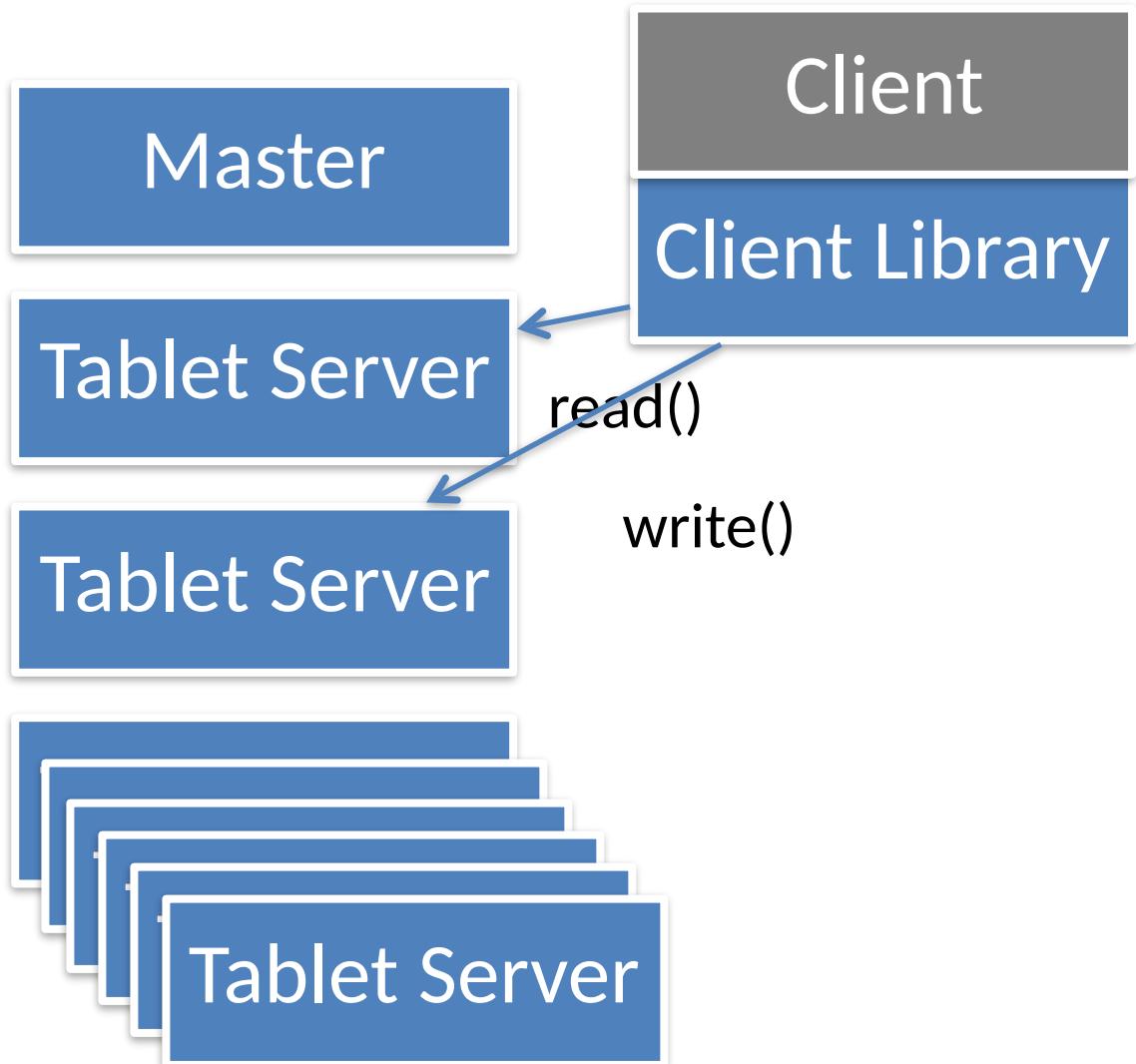
- Started in 2004, publicly available since 2015 (as Hbase), extended by F1/Spanner
- Designed for the petabyte scale
- Used for many Google applications: used for web indexing, personalized search, Google Earth, Google Analytics, Google Finance
- Both BigTable and MapReduce operate on GFS for different purposes.
 - BigTable: read/write web data
 - MapReduce: offline batch processing
 - Google File System: common persistent storage layer

BigTable: Tables & Tablets



BigTable components

- Client library
- Master
 - Metadata operations
 - Load balancing
- Tablet server
 - Data operations



Master

- Assigns tablets to tablet servers
- Detects addition and expiration of tablet servers
- Balance tablet server load
- Etc.

Master

Tablet Server

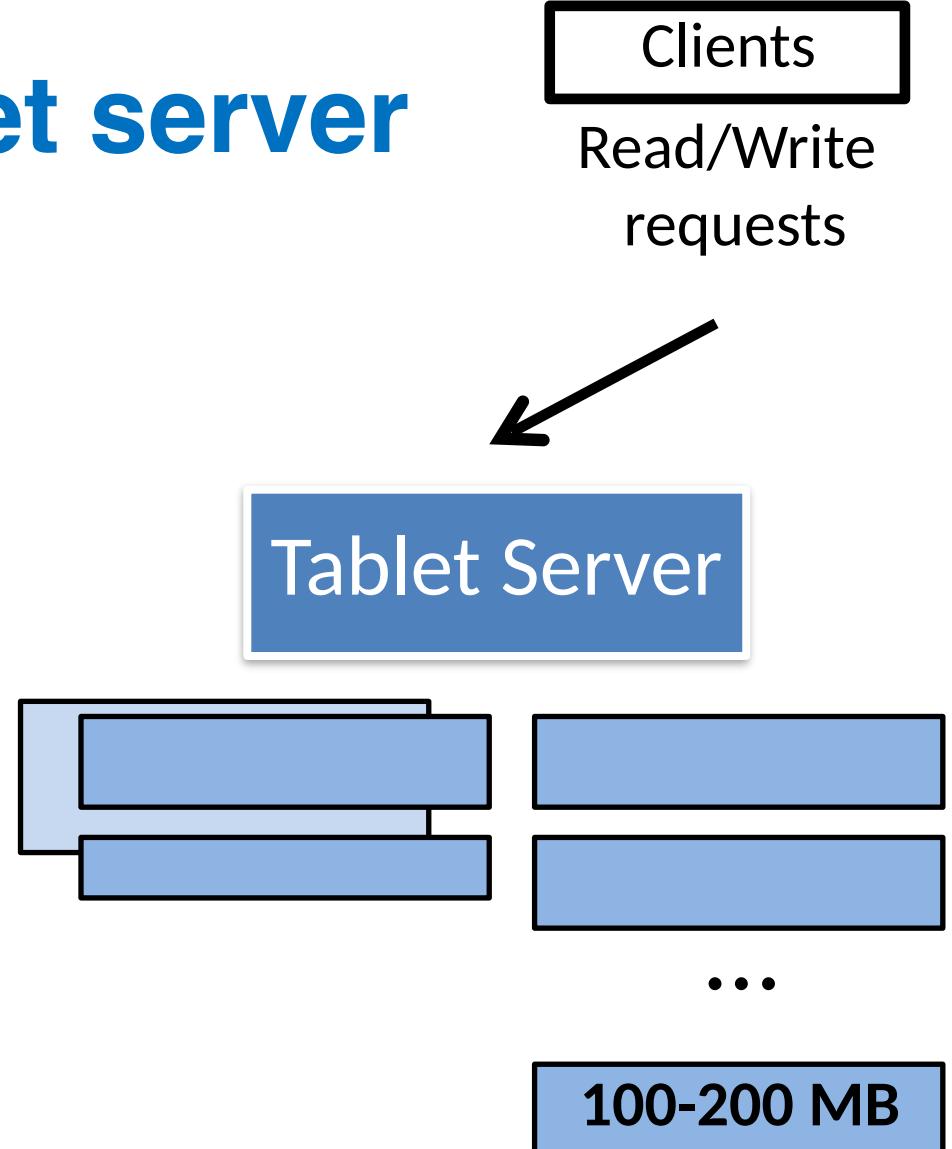
Tablet Server

Tablet Server

Tablet Server

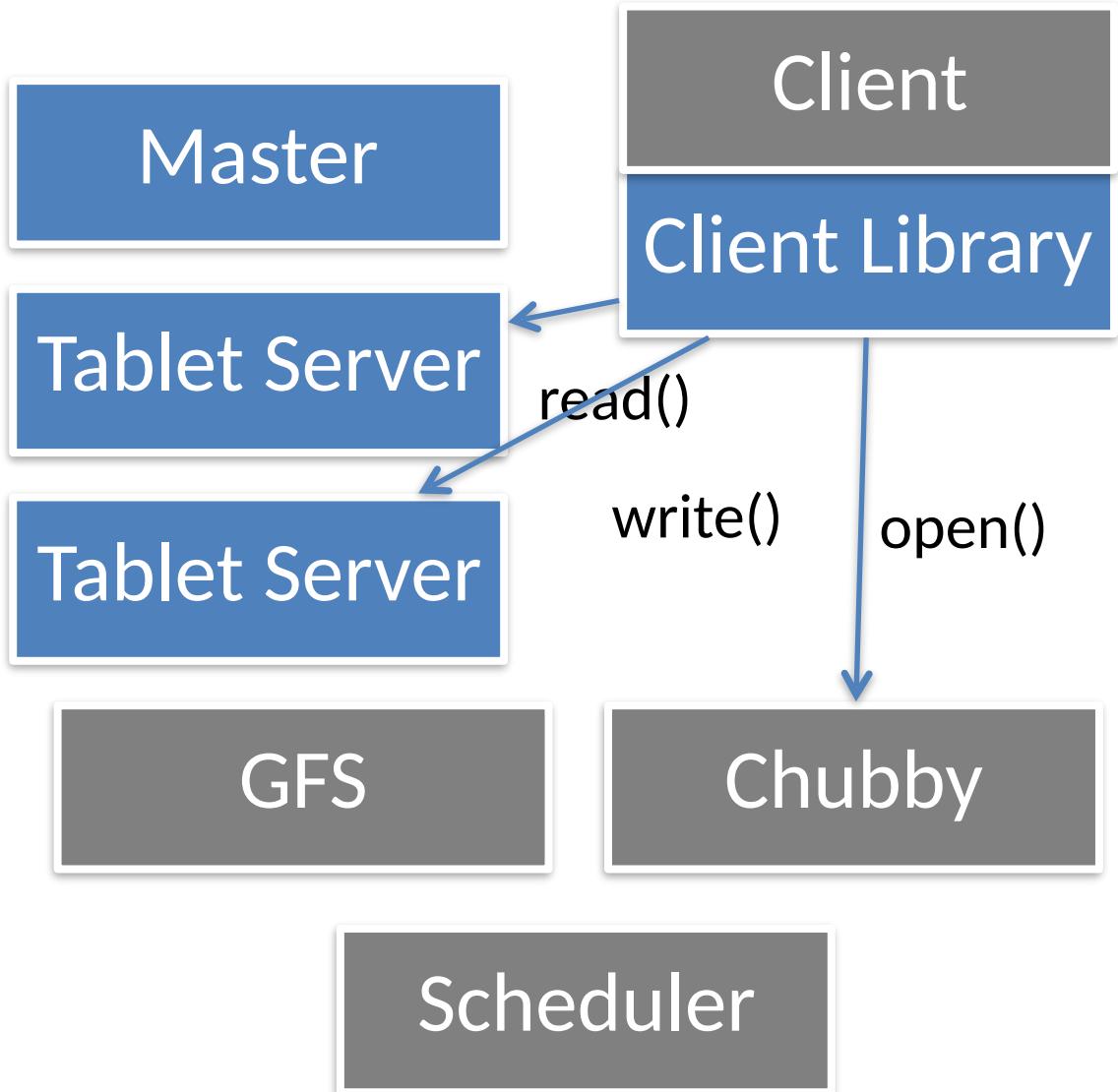
Tablet server

- Manages a set of tablets (up to a thousand)
- Handles read and write requests for the tablets it manages
- Splits tablets that have grown too large



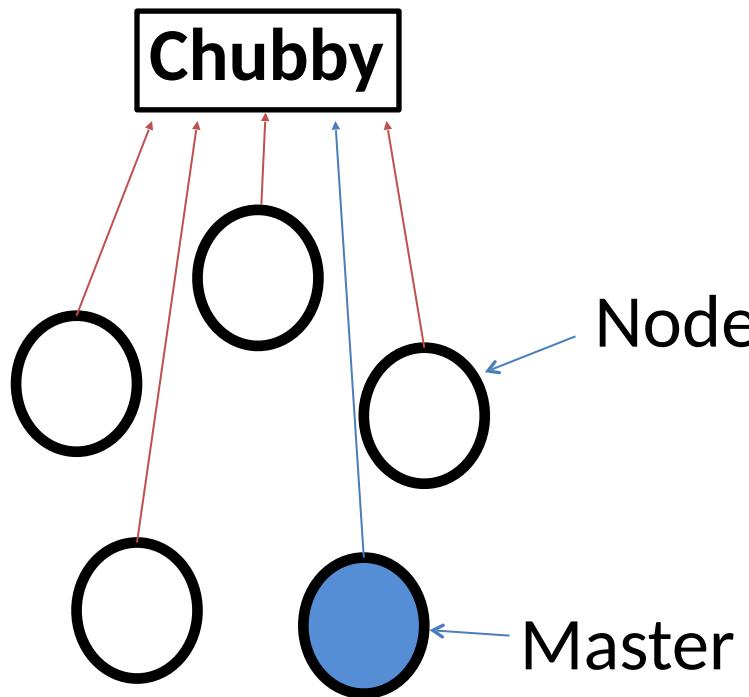
BigTable building blocks

- Chubby
 - Lock service
 - Metadata storage
- GFS
 - Data, log storage
 - Replication
 - Uses Sorted Strings Table files (SSTables)
- Scheduler
 - Monitoring
 - Failover



Chubby lock service

Highly-available, persistent, distributed lock and coordination service



Sample use in BigTable

- Ensure at most one active BigTable master at any time
- Store bootstrap location of data (root tablet)
- Discover tablet servers (manage their lifetime)
- Store schema information

Lock service operational model

- Knows about and manages directories, **files**
- Directories, files can serve as **locks**
- Reads, writes are **atomic**

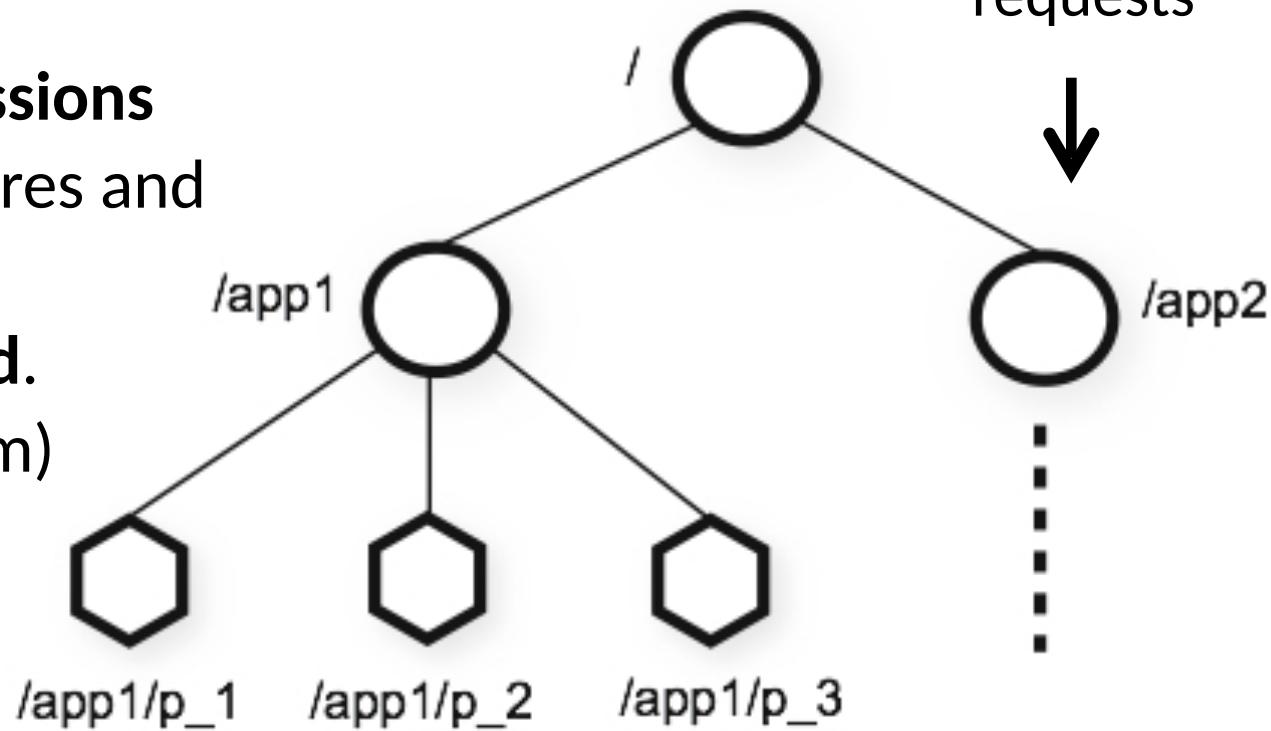
Clients maintain **sessions**

If session lease expires and
can't be renewed,

locks are released.

(timeout mechanism)

Chubby
Clients
Read/Write
requests



Lock service availability

- Comprised of five active replicas
 - Consistently replicate writes (cf. **Paxos**)
- One replica is designated as master
 - We need to **elect** the master (leader)
 - Chubby master is different from BigTable master!
- Service is up when:
 - Majority of replicas are running and
 - A quorum of replicas needs to be established
 - Can communicate with one another

Core mechanisms

- Ensure one active BigTable master at any time
 - **Mutual exclusion** but in a distributed setting
- Keep replicas consistent in face of failures
 - **Paxos algorithm** based on replicated state machines (RSM)
 - Atomic broadcast

Chubby example: leader election

- Electing a master (leader) node: supported by acquiring an exclusive lock on a file (**clients represent partaking nodes**)
- Clients **open a lock file** and attempt to acquire the lock in write mode
- One client **succeeds** (i.e., becomes the **leader**) and writes its name to the file
- Other clients **fail** (i.e., become **replicas**) and discover the name of the leader by reading the file

Chubby example: leader election

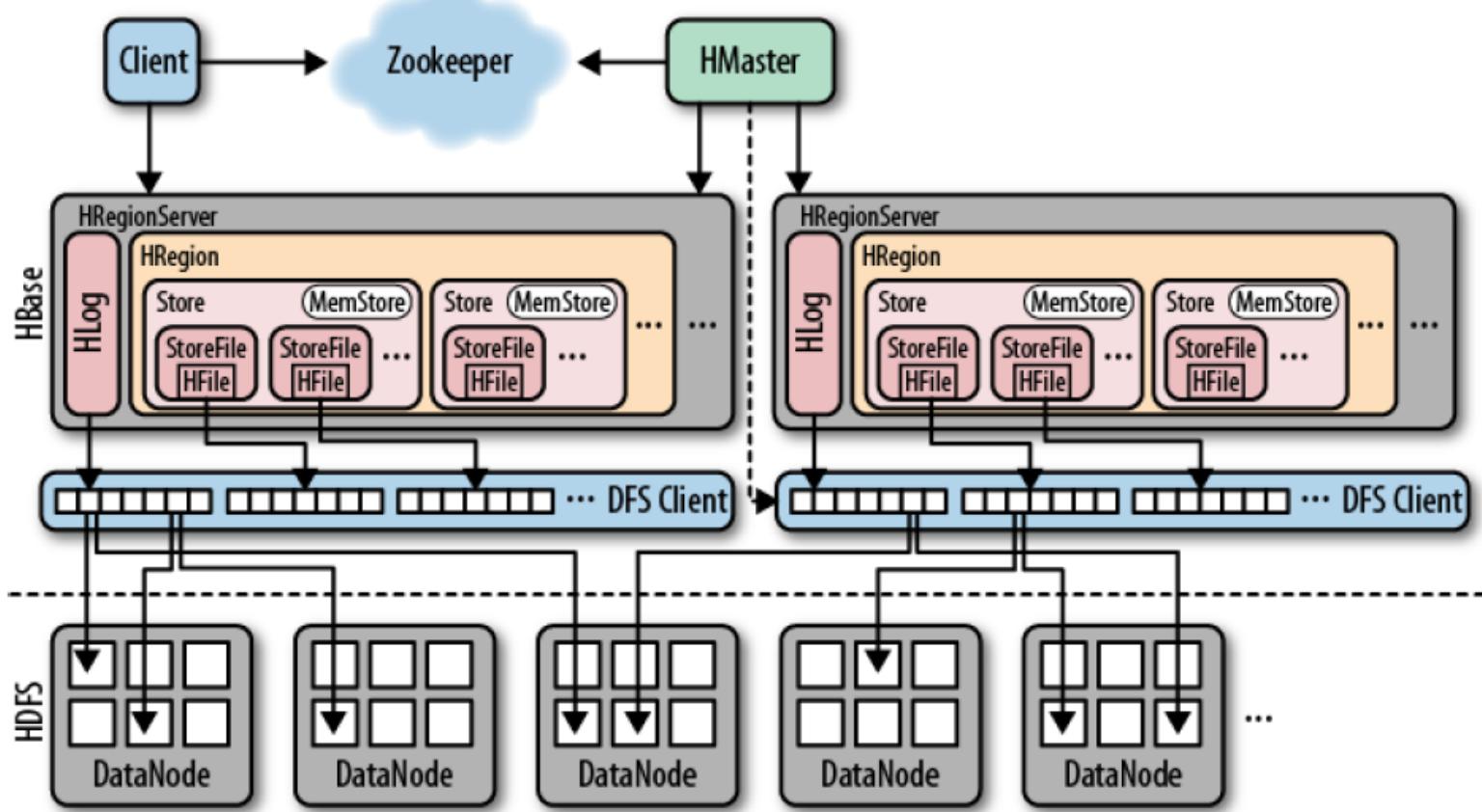
```
Open(„/ls/cell1/somedir/file1“,  
      „write mode“) → obtain file handle  
  
if (successful) { // master  
    setContents(primary_identity) → write to file  
} else {           // replica  
    Open(„/ls/cell1/somedir/file1“,  
          „read mode“,  
          „file-modification event“) → subscribe to  
                                         modification event  
    On modification notification  
        primary = getContentsAndStat() → read from file  
}
```

Apache HBase

- Open-source implementation of BigTable
- Facebook Messenger uses Hbase
- Different names for the same components
 - GFS → HDFS
 - Chubby → Zookeeper
 - BigTable → Hbase
 - MapReduce → Hadoop



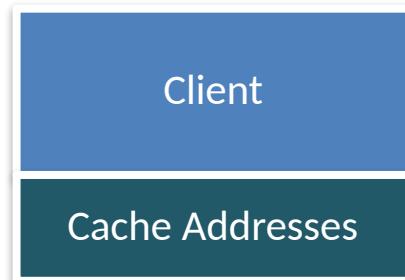
HBase architecture



Lars George: Hbase: The Definitive Guide O'Reilly Media, Inc., 2011

HBase read-path

Client wants to read
key k_1 from table t_1



ZooKeeper

Region Server 1

-ROOT-:
Addresses of
meta tables

Region Server 2

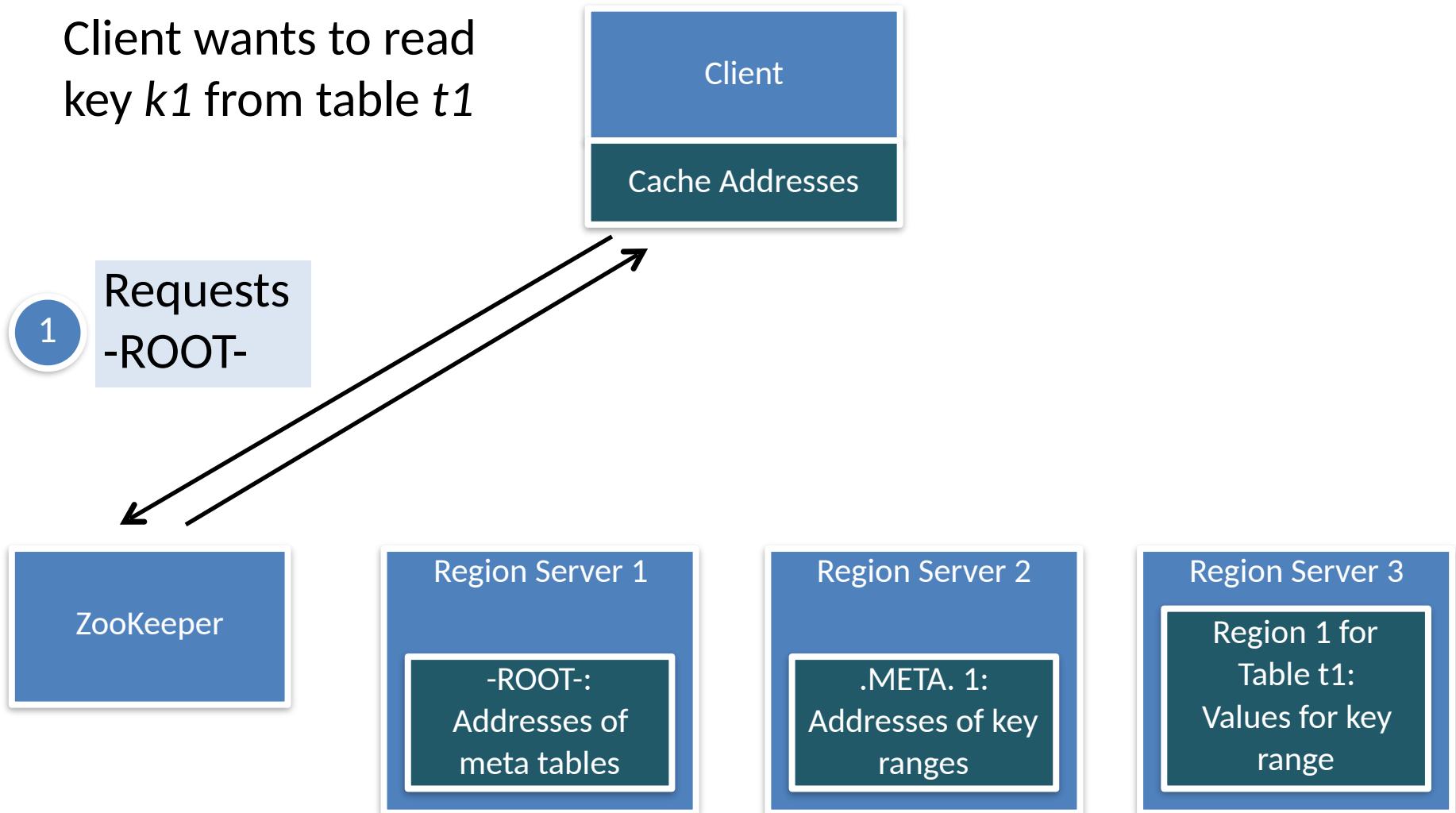
.META.
Addresses of key
ranges

Region Server 3

Region 1 for
Table t_1 :
Values for key
range

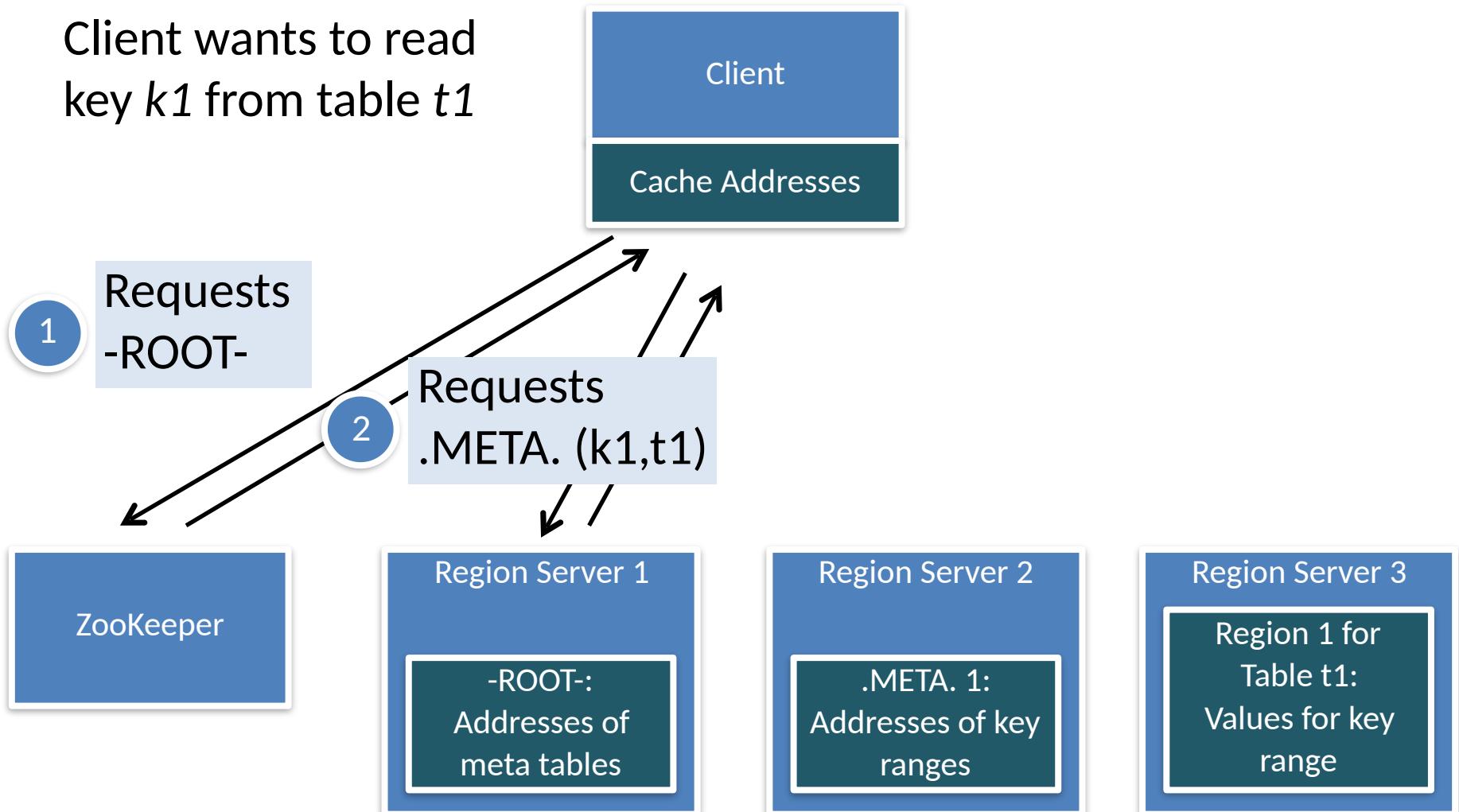
HBase read-path

Client wants to read
key k_1 from table t_1



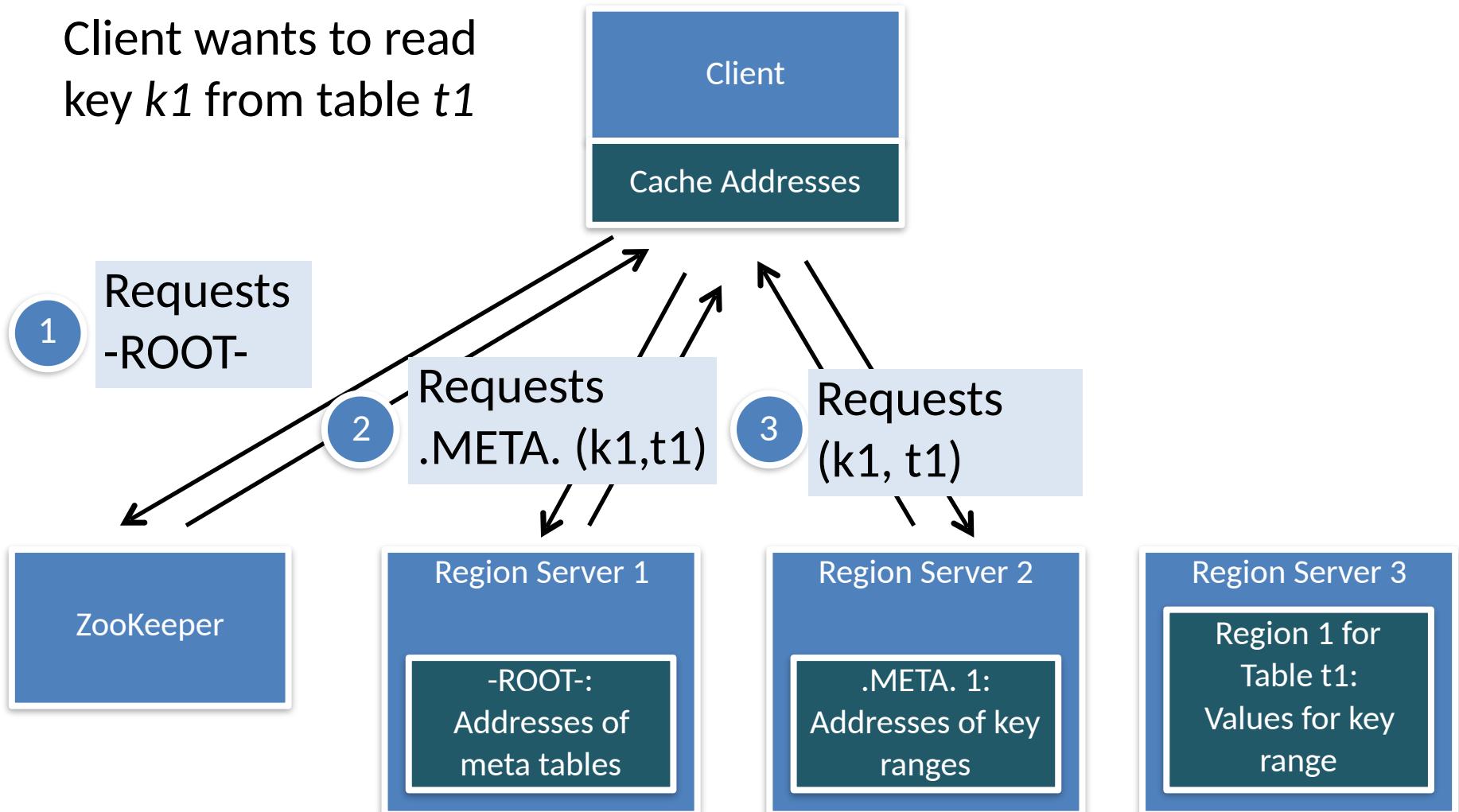
HBase read-path

Client wants to read key k_1 from table t_1



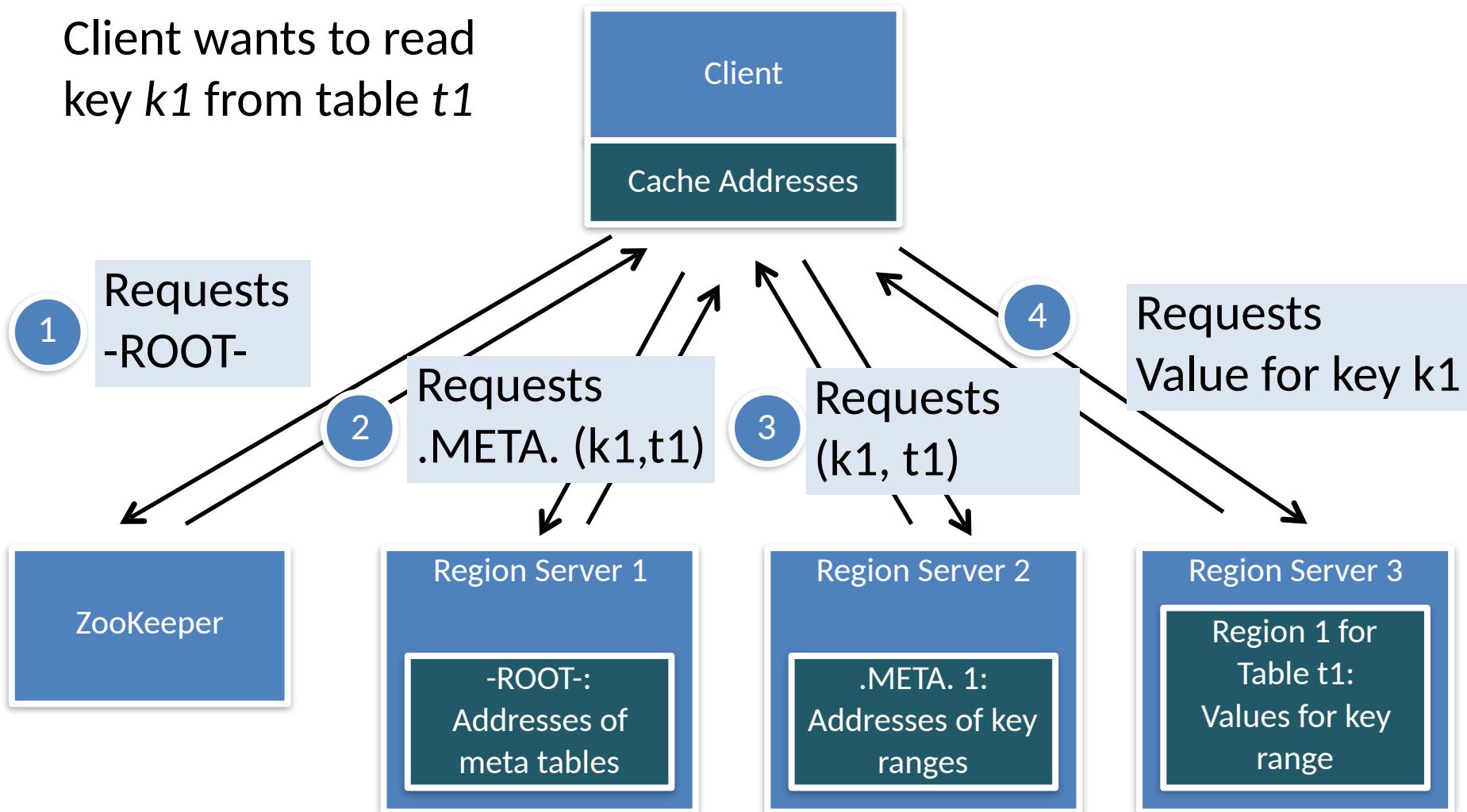
HBase read-path

Client wants to read key k_1 from table t_1

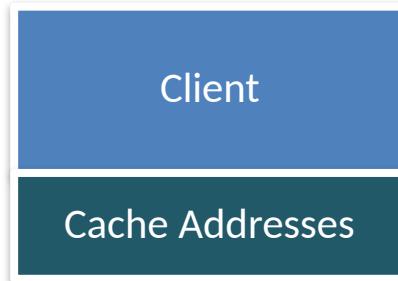


HBase read-path

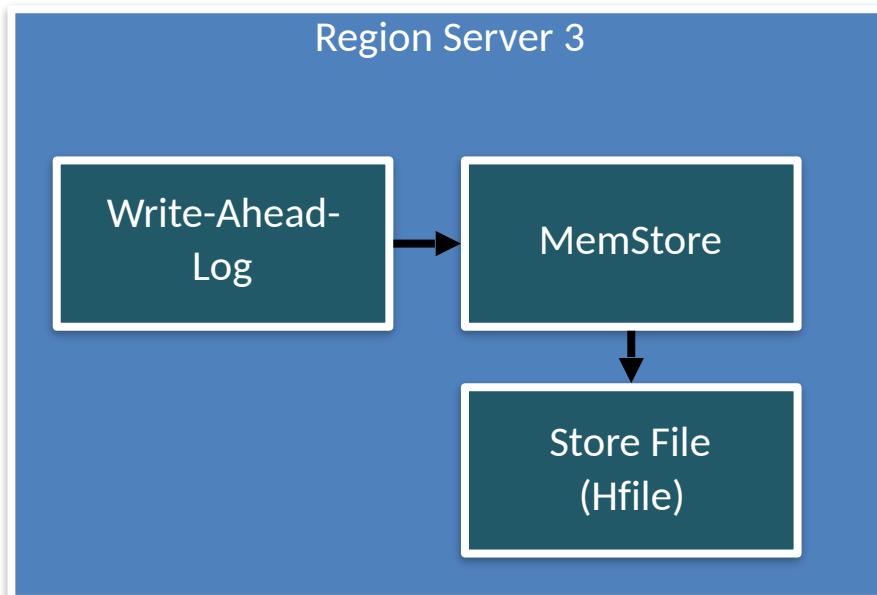
Client wants to read key k_1 from table t_1



HBase write-path



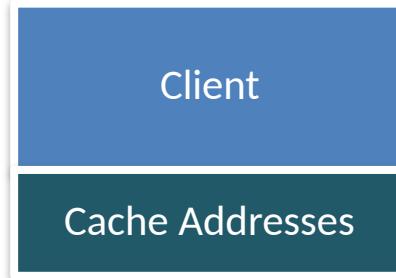
Client wants to store
a value under key $k1$
in table $t1$



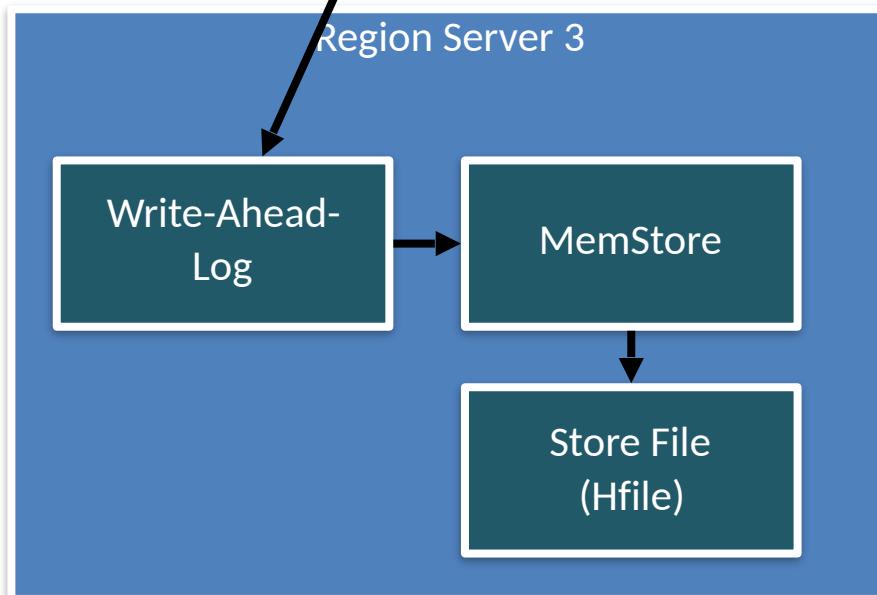
HBase write-path

1

Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS



Client wants to store a value under key $k1$ in table $t1$



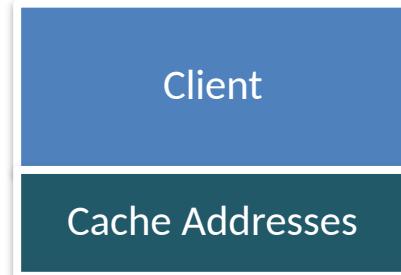
HBase write-path

1

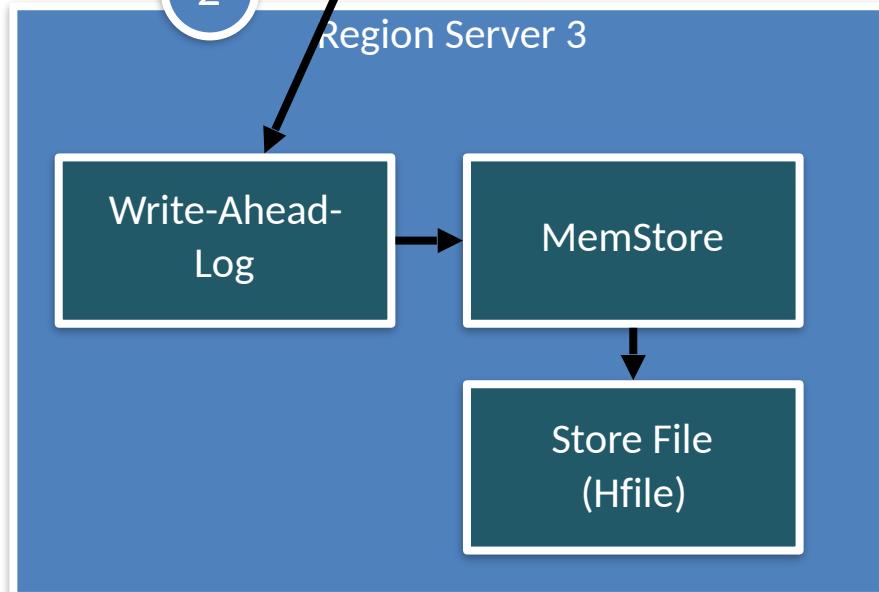
Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS

Send put(k_1 , value)
to RS

Client wants to store a value under key k_1 in table t_1



2



HBase write-path

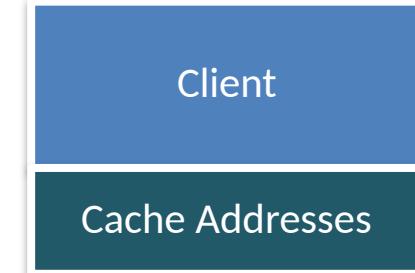
1

Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS

Send put(k_1 , value) to RS

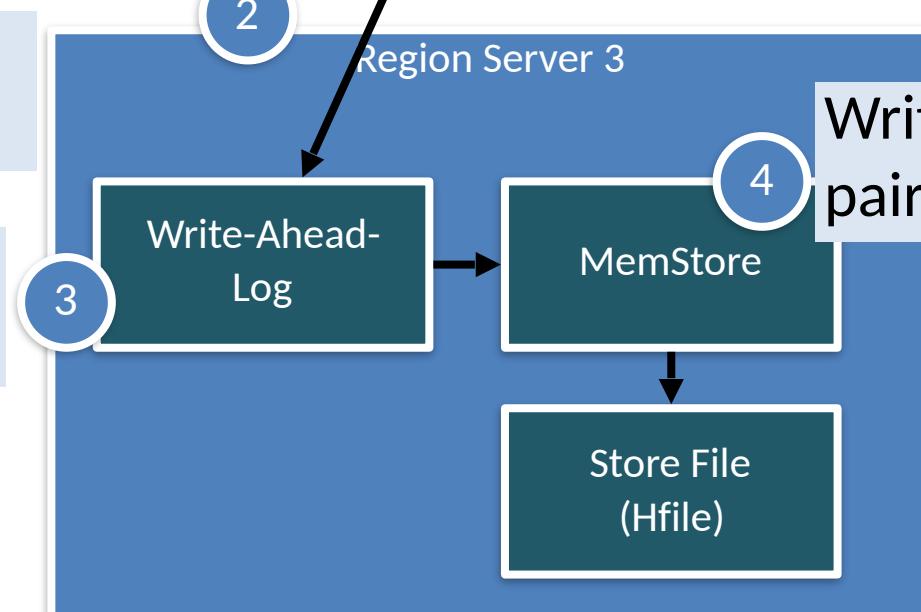
Key-value pair is written to WAL

Client wants to store a value under key k_1 in table t_1



2

Region Server 3



3

Write key-value pair to MemStore

HBase write-path

1

Perform the same four steps as on read-path. If address of region server (RS) is cached go directly to RS

Send put(k_1 , value) to RS

Key-value pair is written to WAL

Client

Cache Addresses

2

Region Server 3

Write-Ahead-Log

3

Client wants to store a value under key k_1 in table t_1

4

MemStore

Write key-value pair to MemStore

5

Store File (Hfile)

Flush key-value pair to HDFS

Summary on BigTable / Hbase I

- Partitioning of data for **horizontal scalability**
 - Tables → Regions (Tablets)
 - **Load-balanced** amongst Region Servers (TabletServer)
 - Write-Ahead-Log for **failure recovery**
- Centralized management
 - Master (HMaster)
 - Backup masters for failover, **leader election** needed
 - Not involved in read/write path (not a bottleneck)

Summary on Bigtable / Hbase II

- **Coordination**
 - Chubby (ZooKeeper) lock service
 - Leader election, server status, region directory, ...
 - Sessions (leases) for timeout (failure detection)
 - Highly available and reliable
 - Paxos, atomic broadcast to replicate state
 - Caching responses to avoid frequent communication
- **Distributed file system**
 - GFS (HDFS)
 - Store data as SSTables (Hfiles)
 - Data is replicated for availability

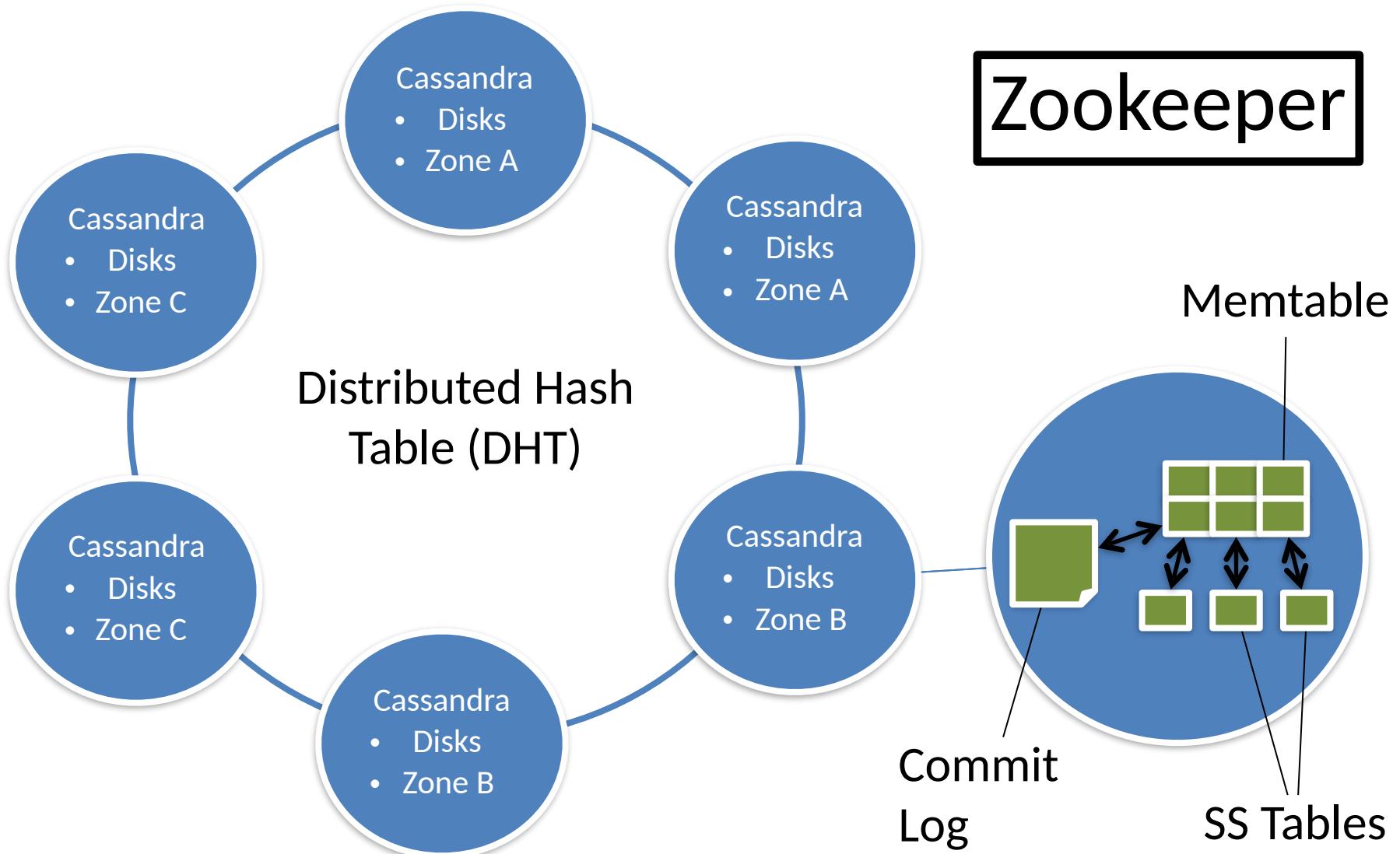


DYNAMO / CASSANDRA

Cassandra

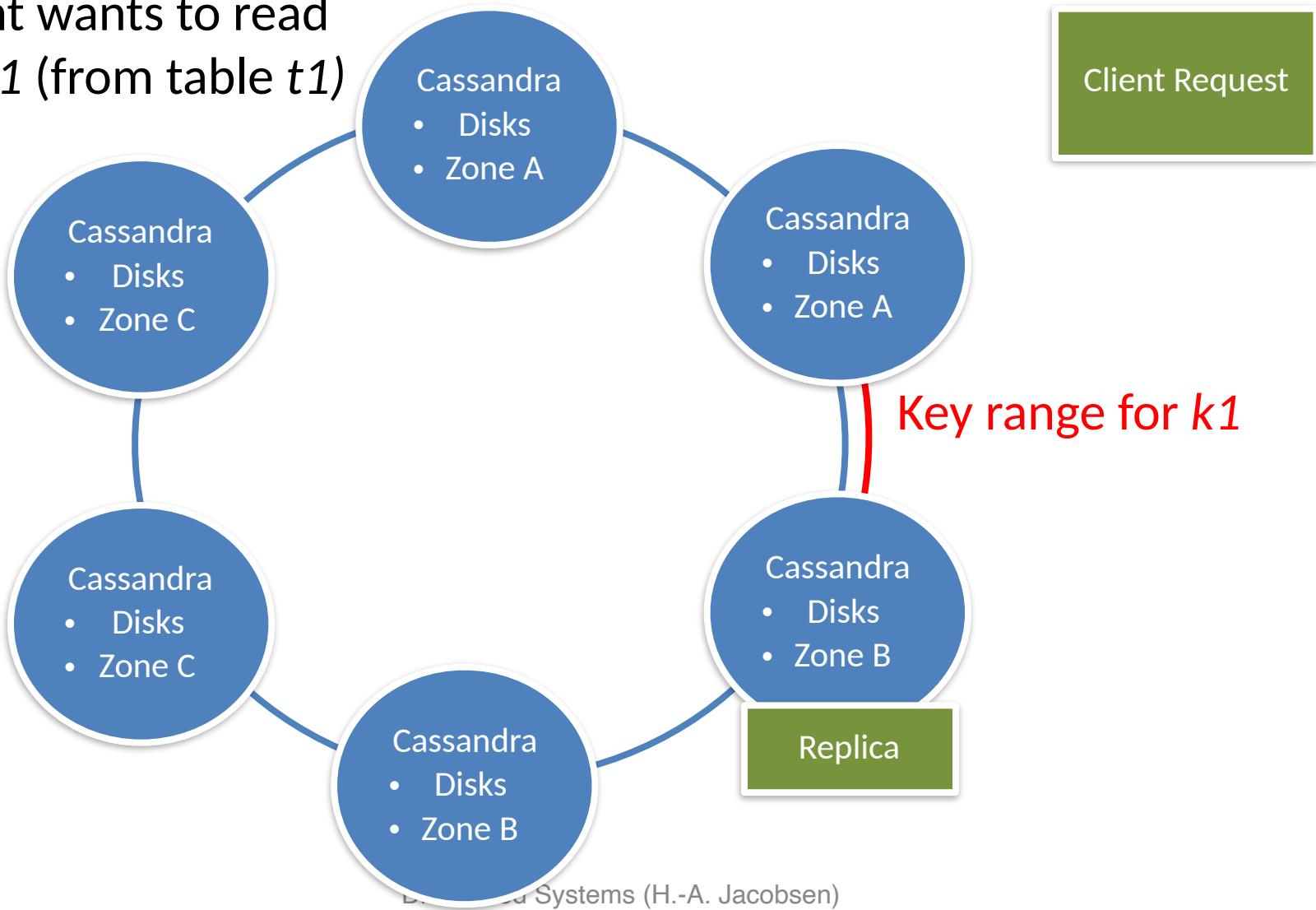
- Developed by Facebook
- Based on Amazon Dynamo (but open-source)
- Structured storage nodes (no DFS used)
- Decentralized architecture (no master)
- **Consistent hashing** for load balancing
- **Eventual consistency**
- **Gossiping** to exchange information

Cassandra architecture overview



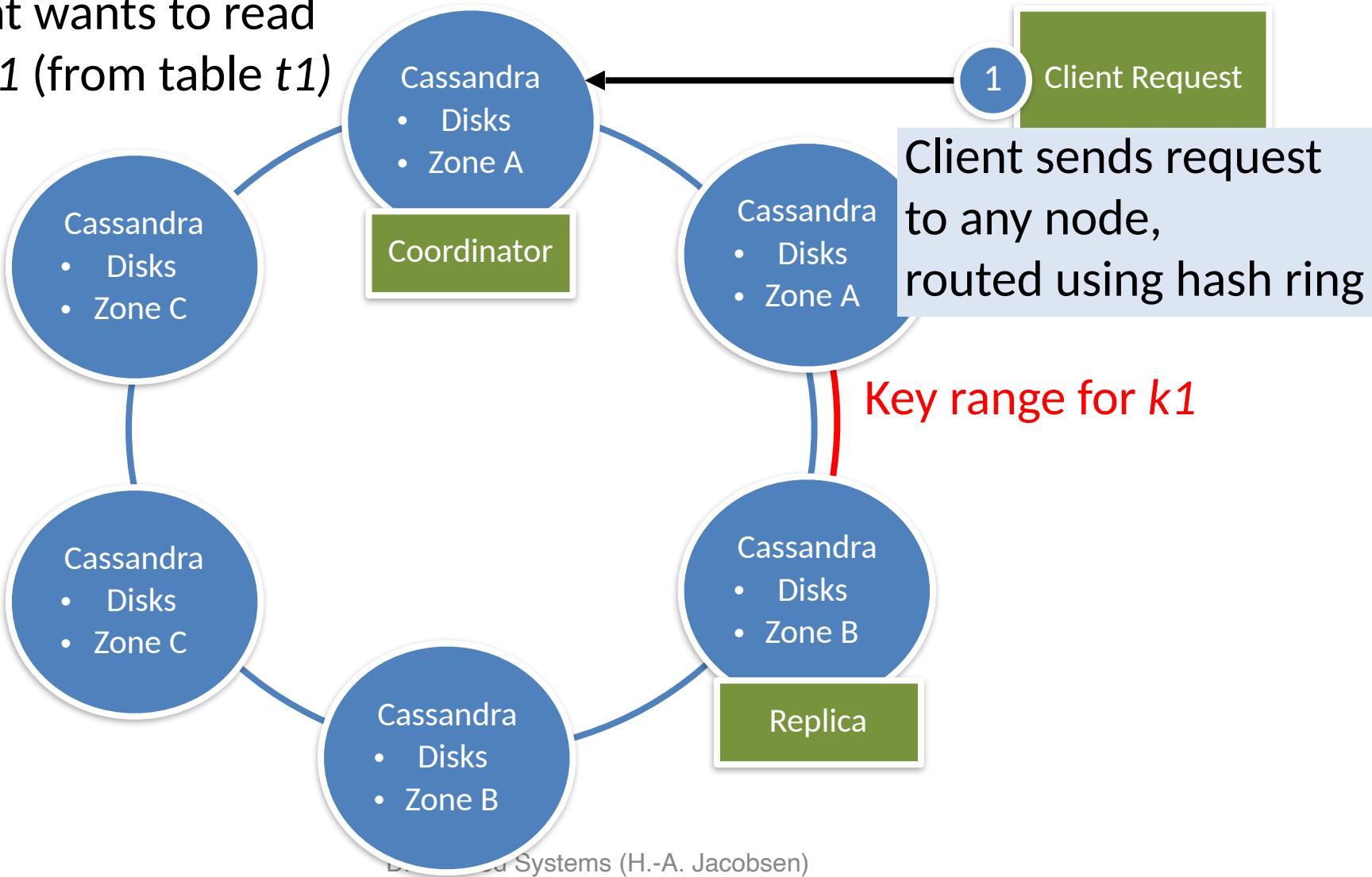
Cassandra global read-path

Client wants to read key $k1$ (from table $t1$)



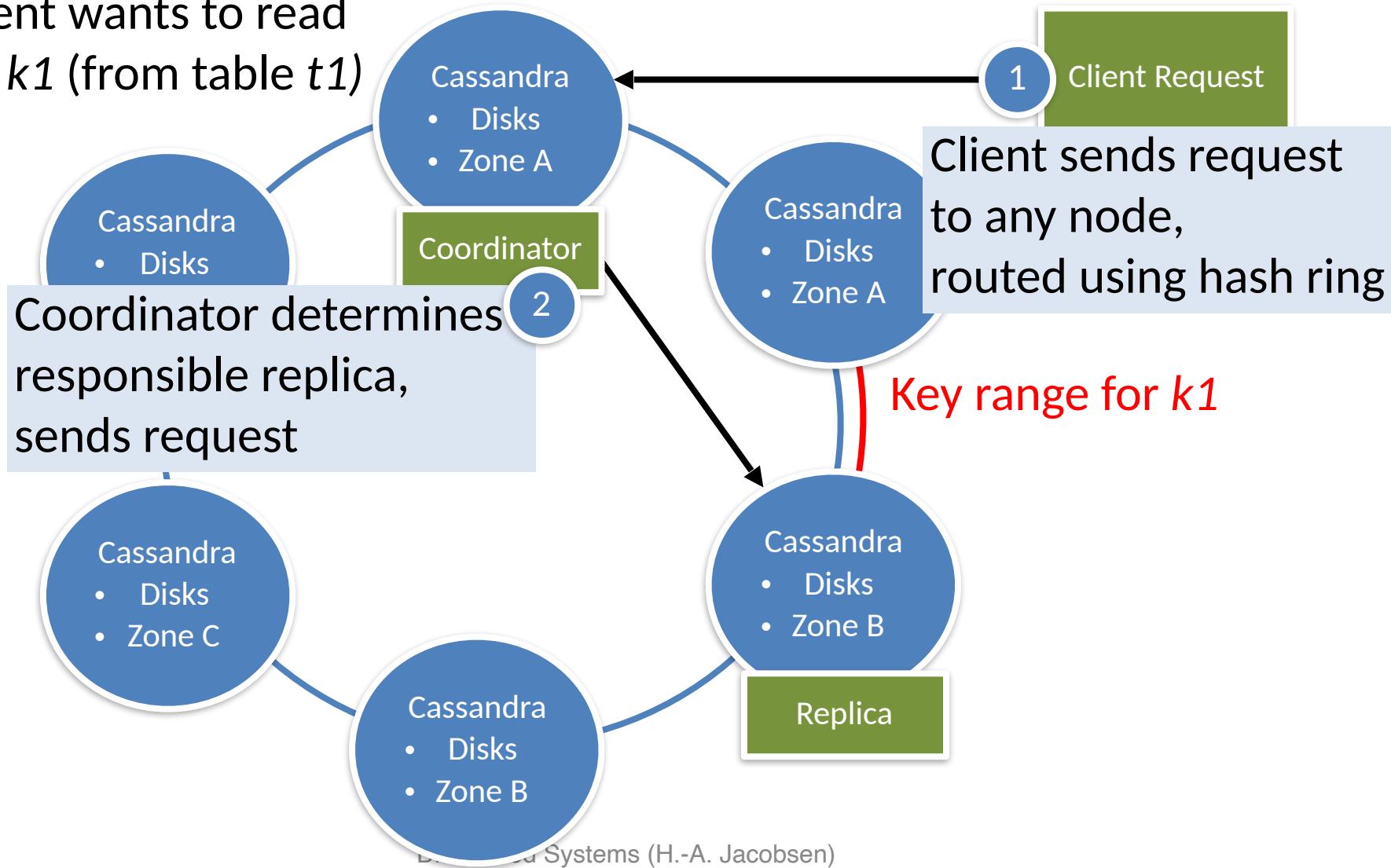
Cassandra global read-path

Client wants to read key k_1 (from table t_1)



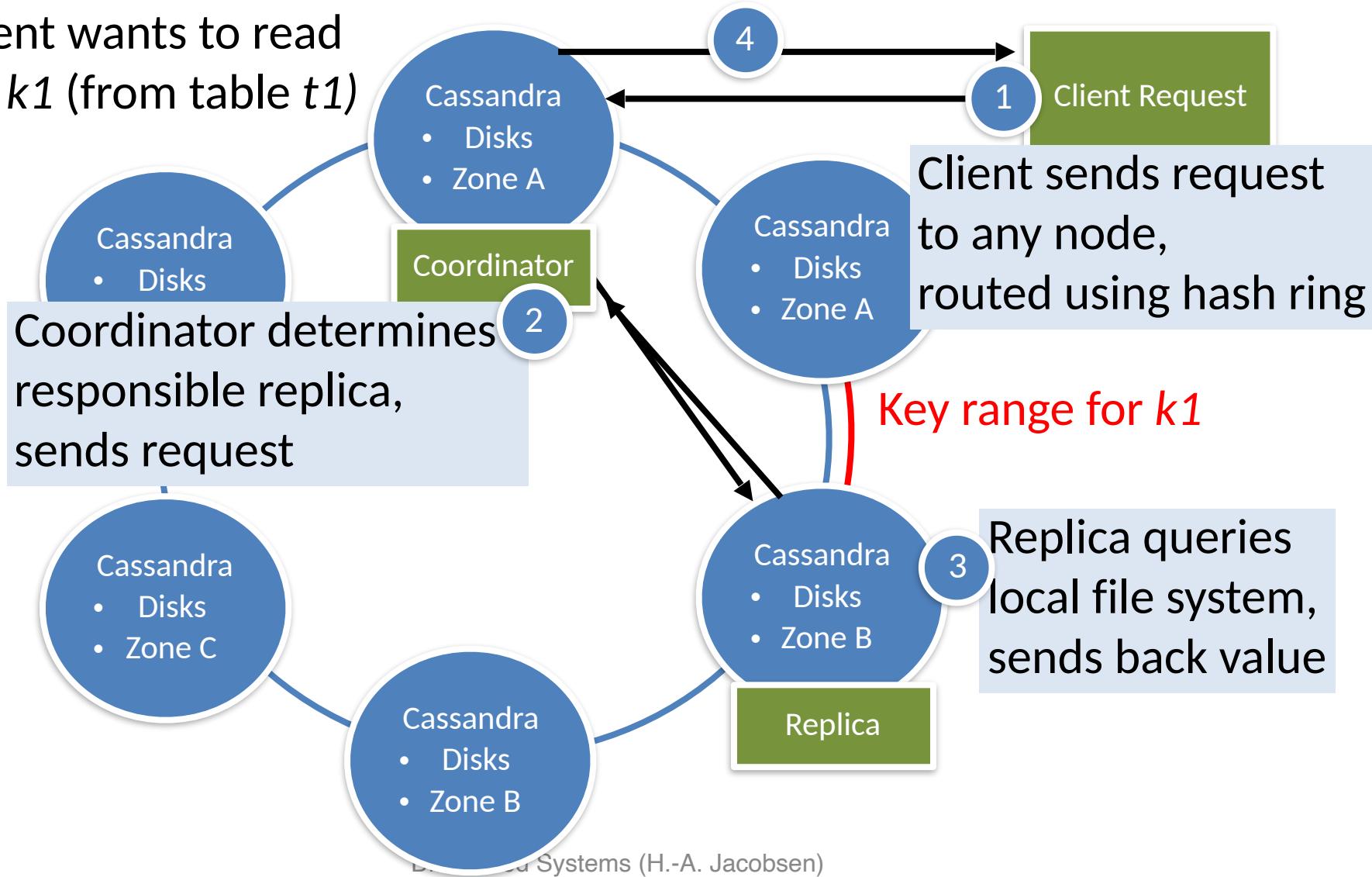
Cassandra global read-path

Client wants to read key k_1 (from table t_1)



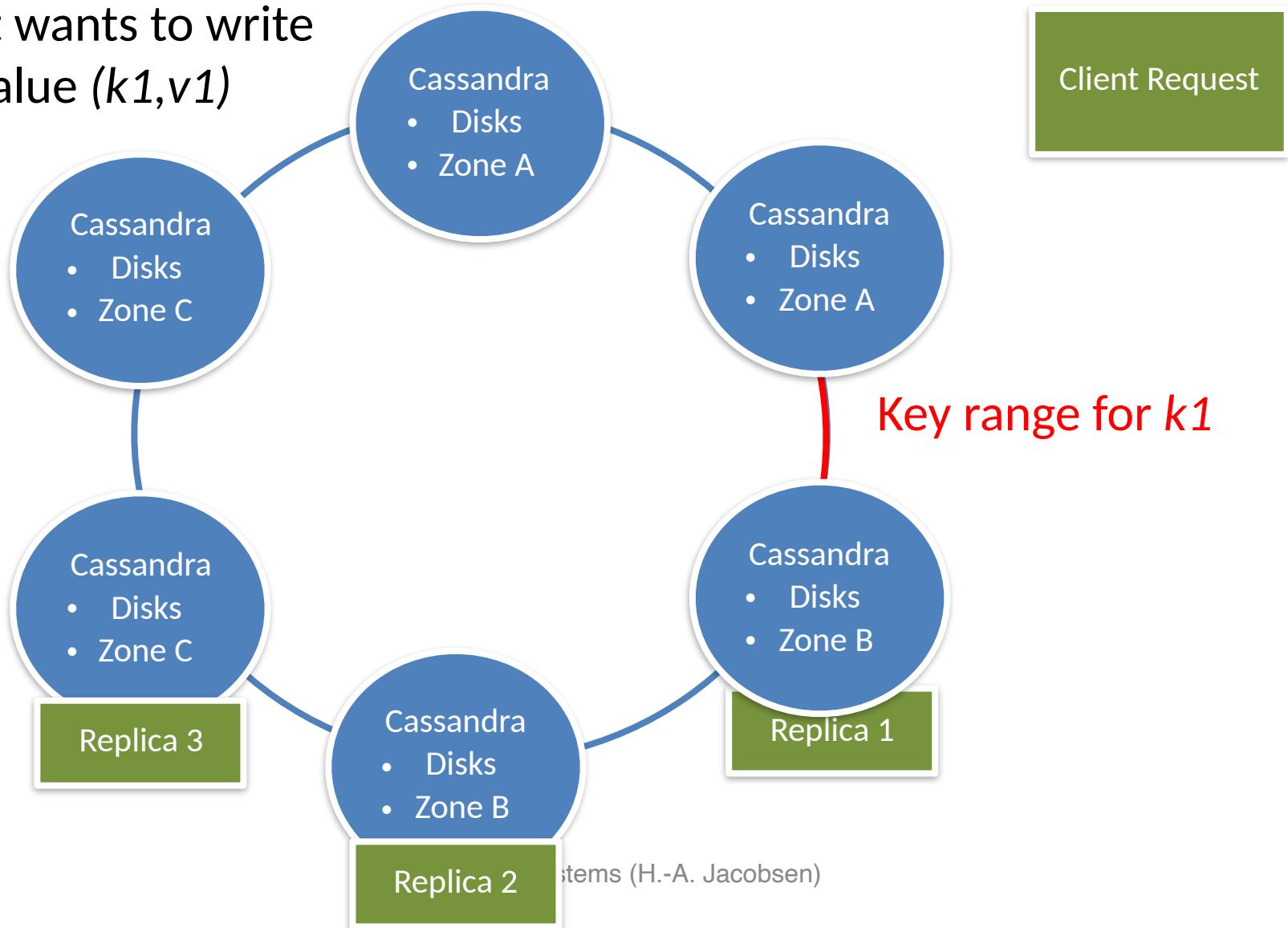
Cassandra global read-path

Client wants to read key k_1 (from table t_1)



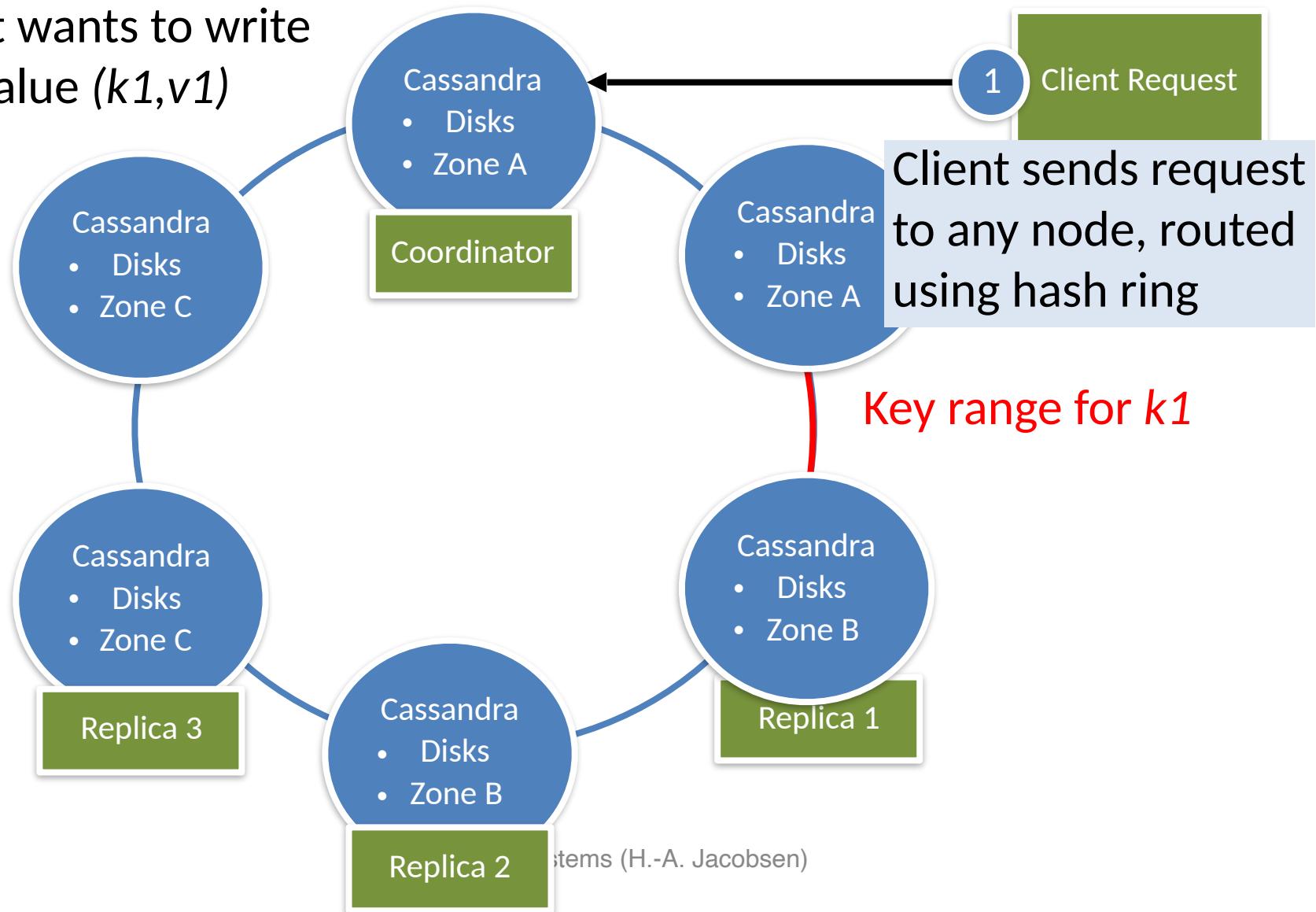
Cassandra global write-path

Client wants to write key-value ($k1, v1$)



Cassandra global write-path

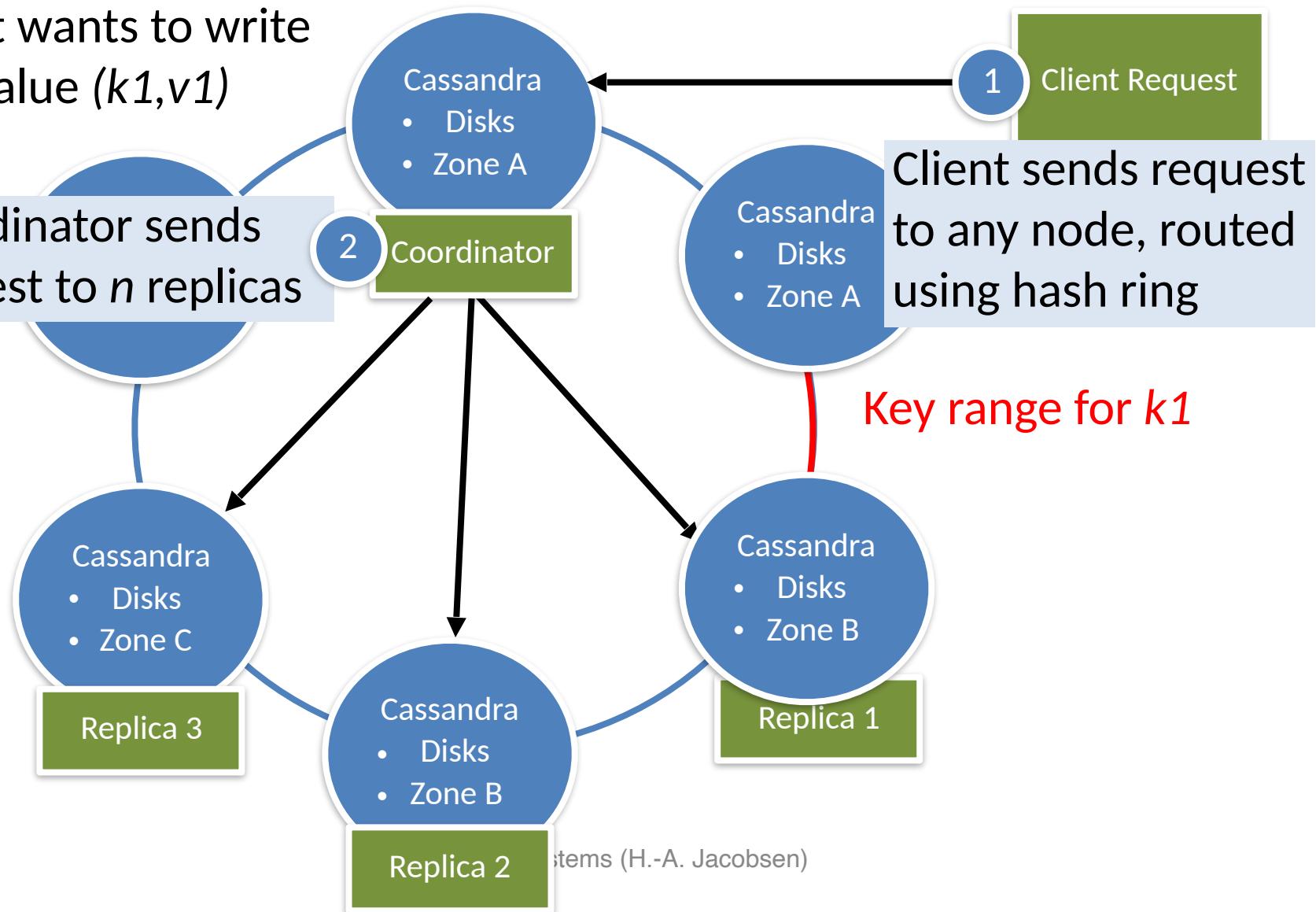
Client wants to write key-value ($k1, v1$)



Cassandra global write-path

Client wants to write key-value ($k1, v1$)

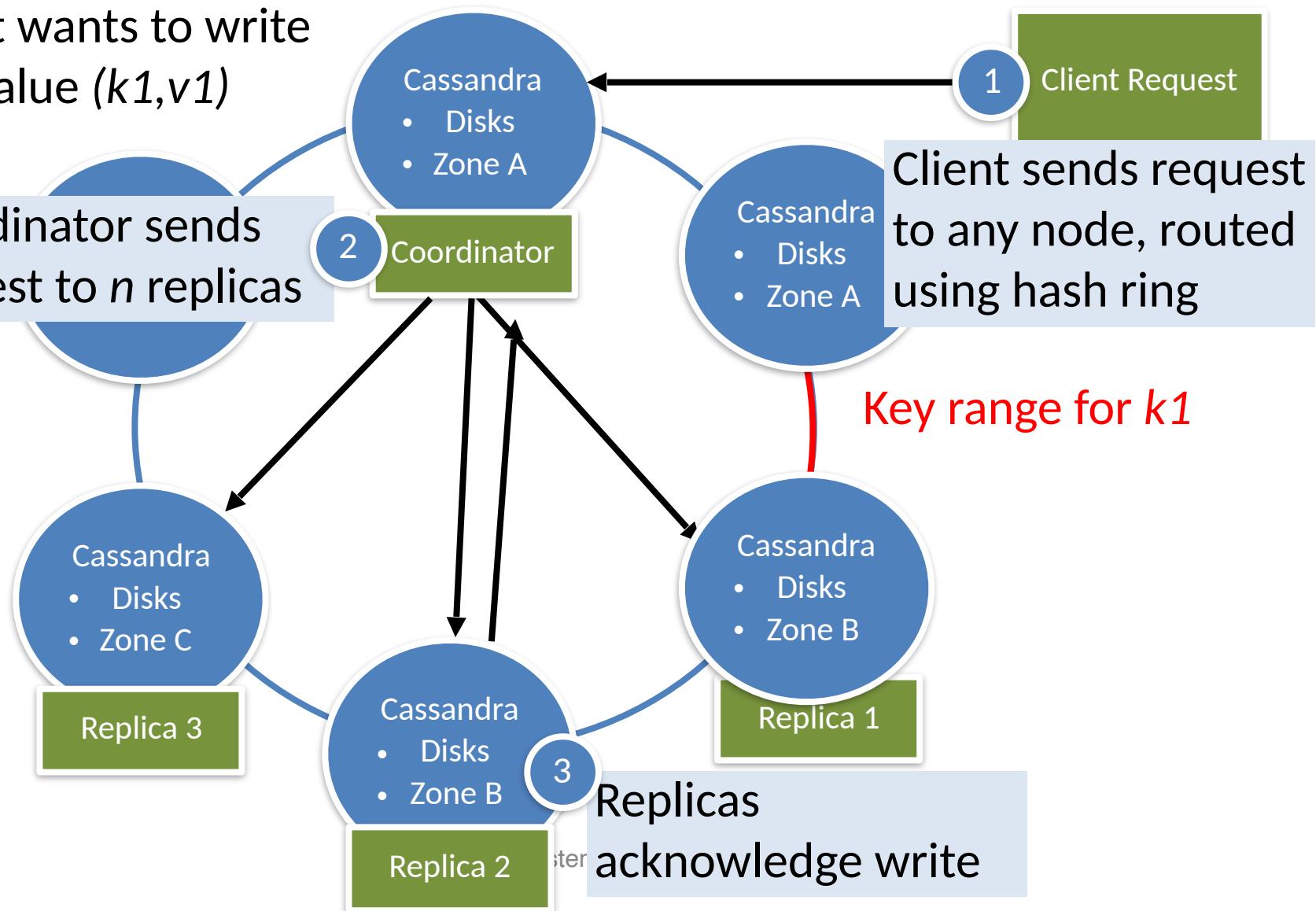
Coordinator sends request to n replicas



Cassandra global write-path

Client wants to write key-value ($k1, v1$)

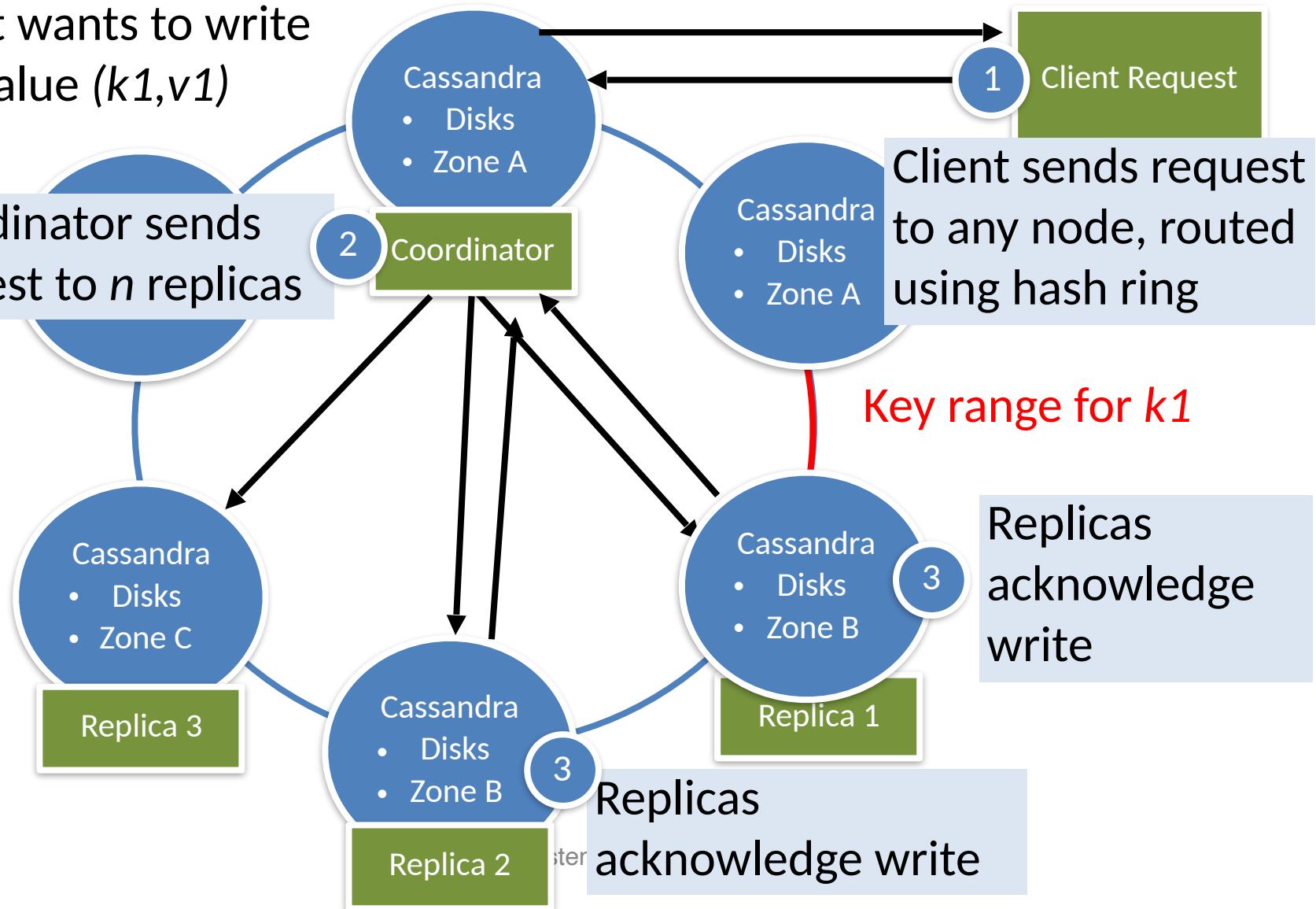
Coordinator sends request to n replicas



Cassandra global write-path

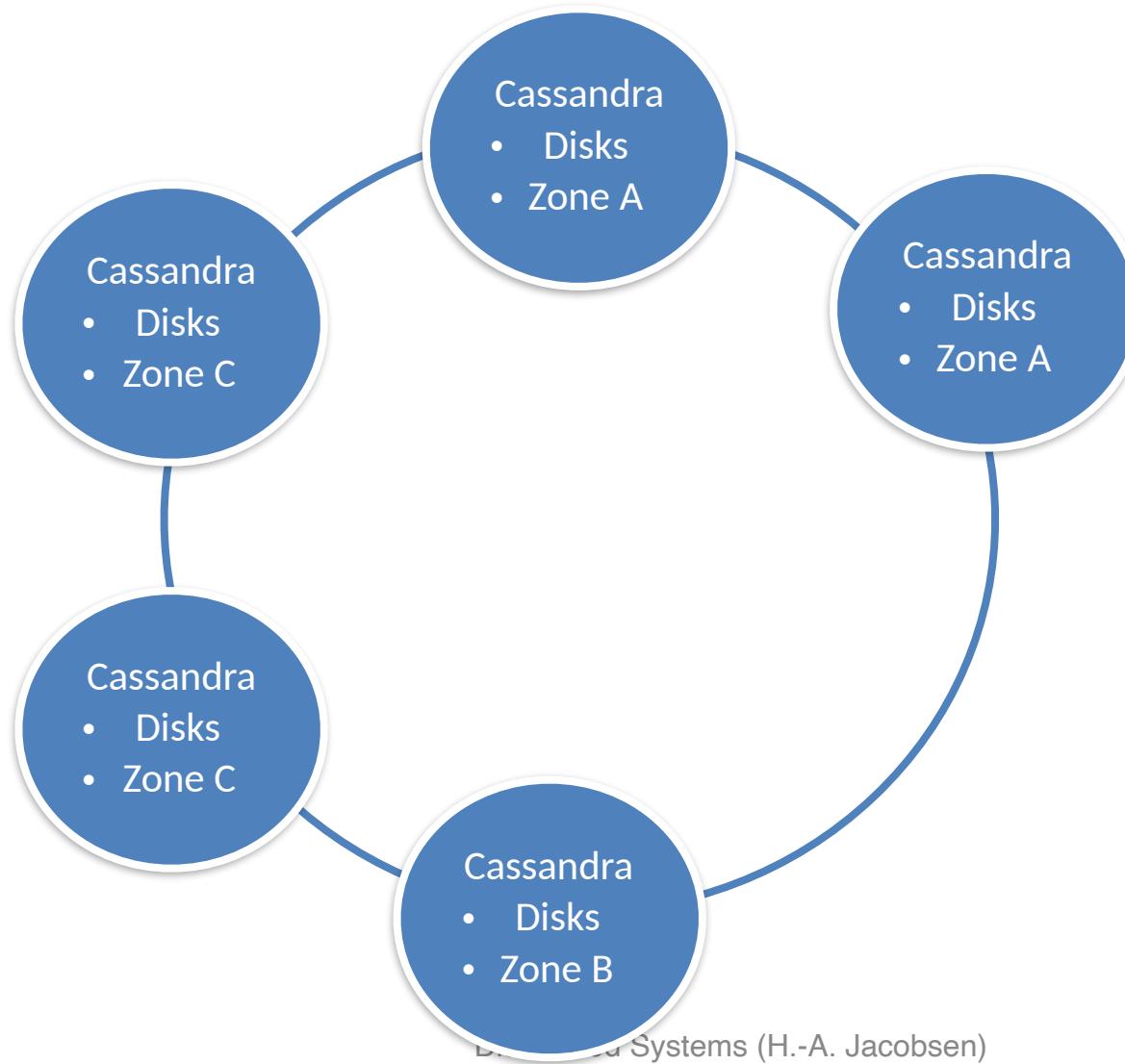
Client wants to write key-value ($k1, v1$)

Coordinator sends request to n replicas



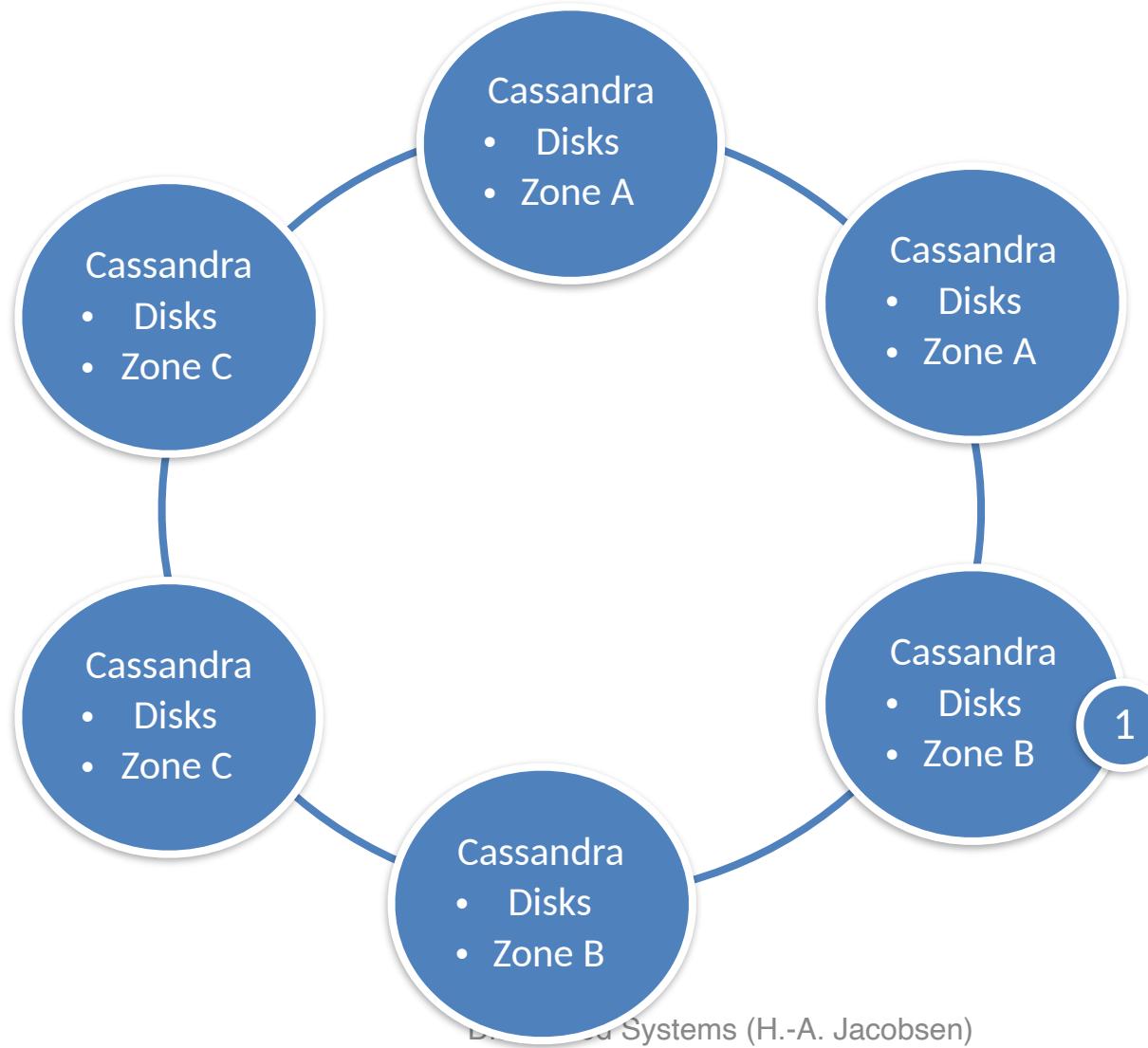
Incremental scaling in Cassandra

(i.e., adding a storage unit)



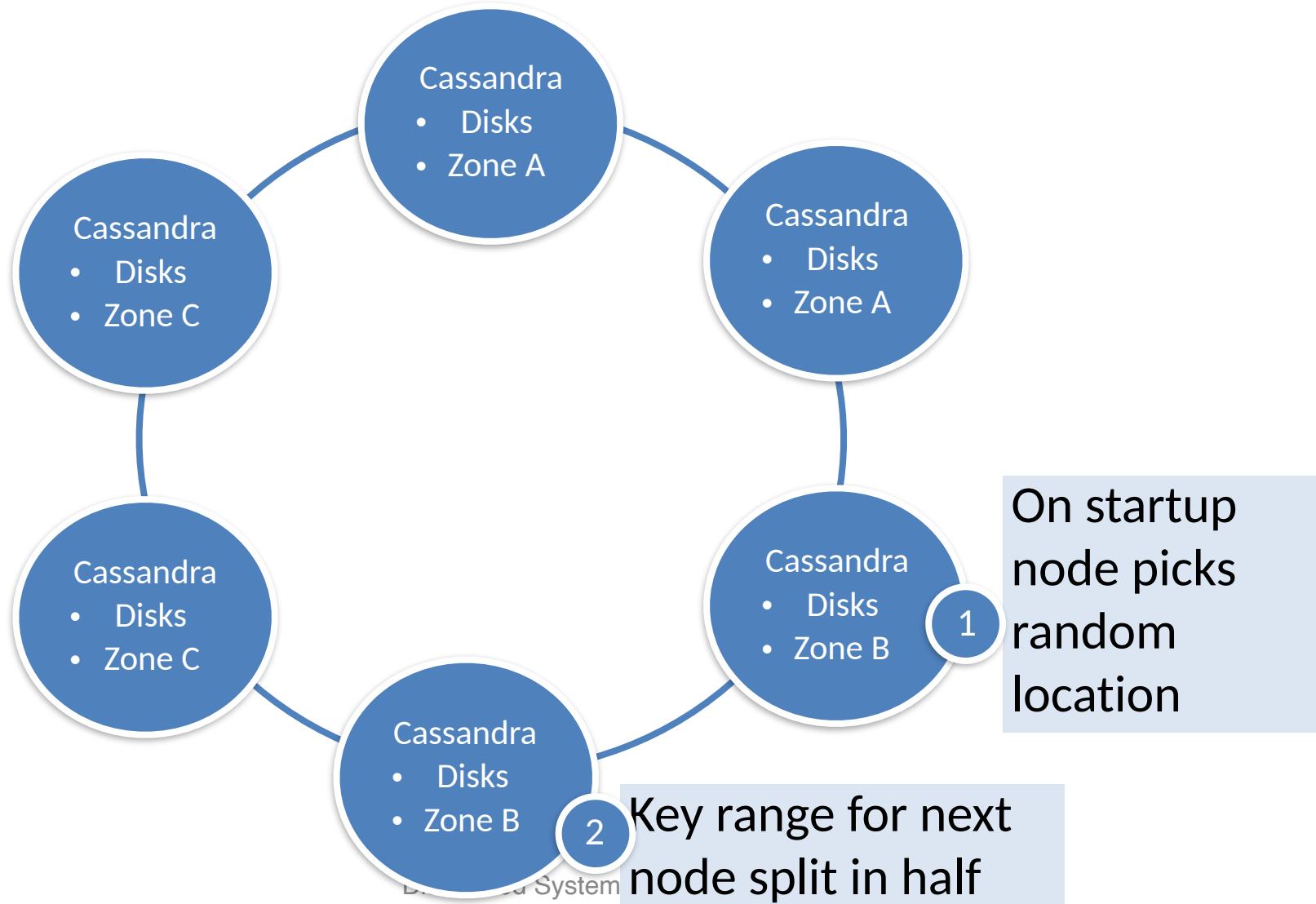
Incremental scaling in Cassandra

(i.e., adding a storage unit)



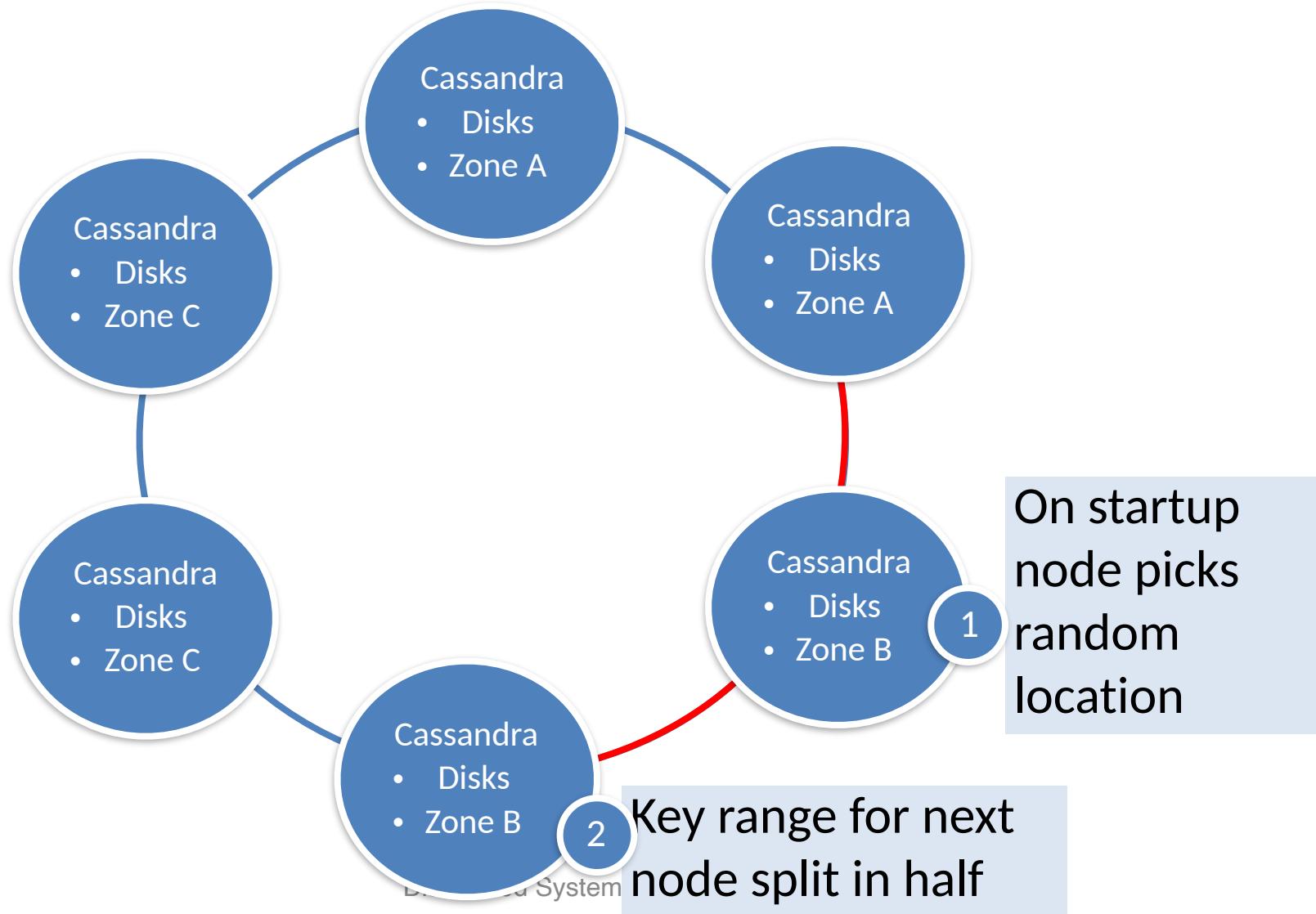
Incremental scaling in Cassandra

(i.e., adding a storage unit)



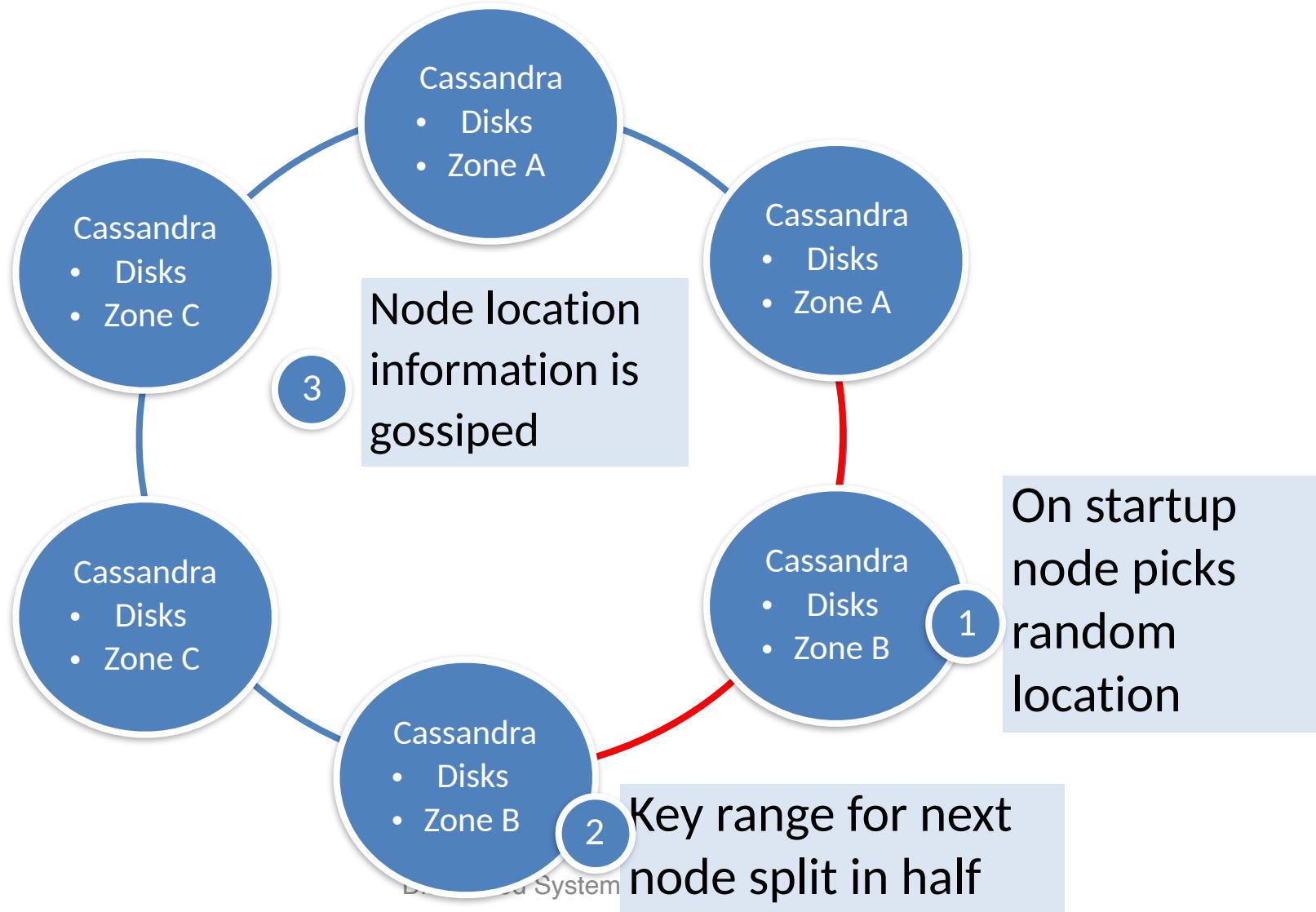
Incremental scaling in Cassandra

(i.e., adding a storage unit)



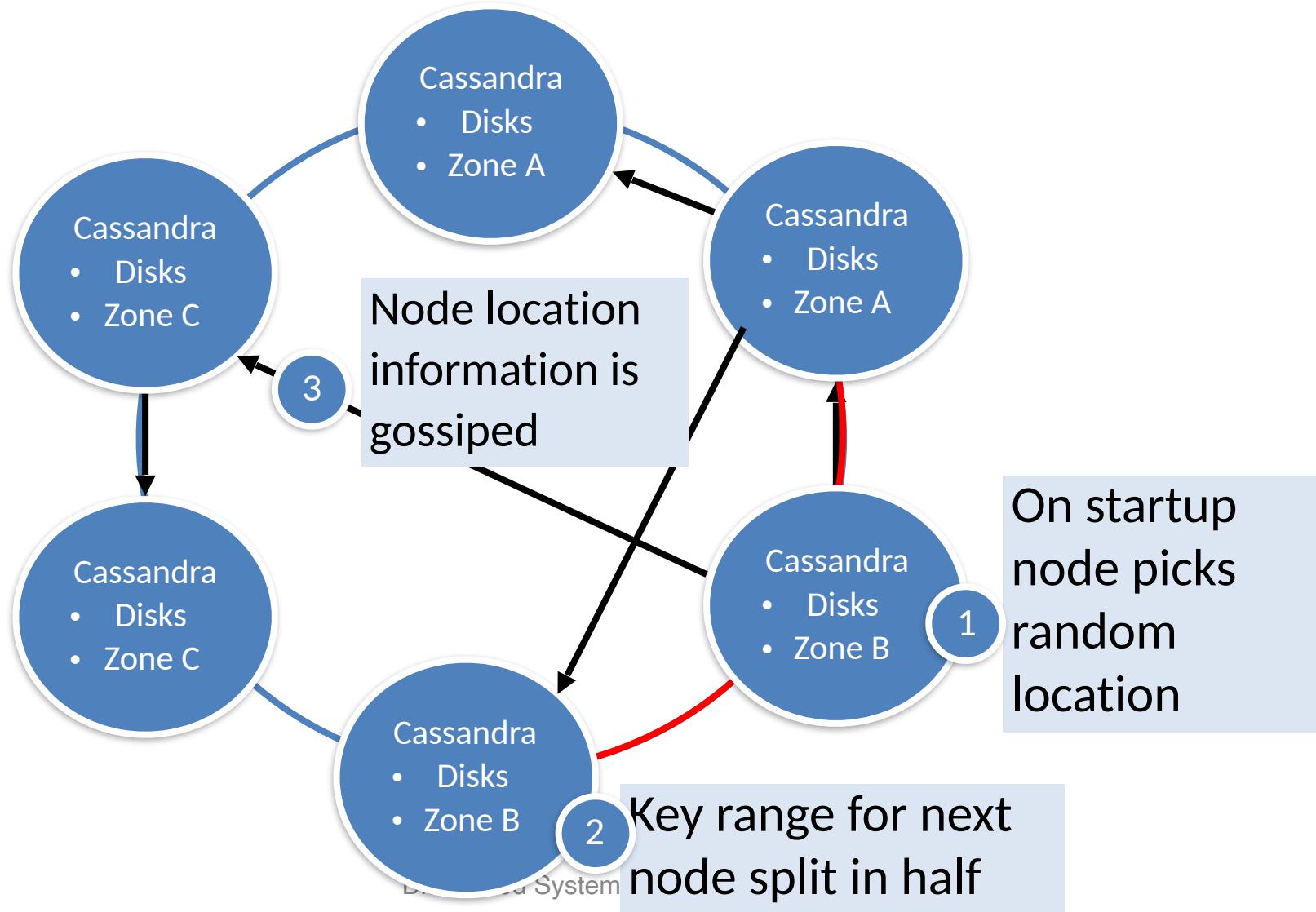
Incremental scaling in Cassandra

(i.e., adding a storage unit)

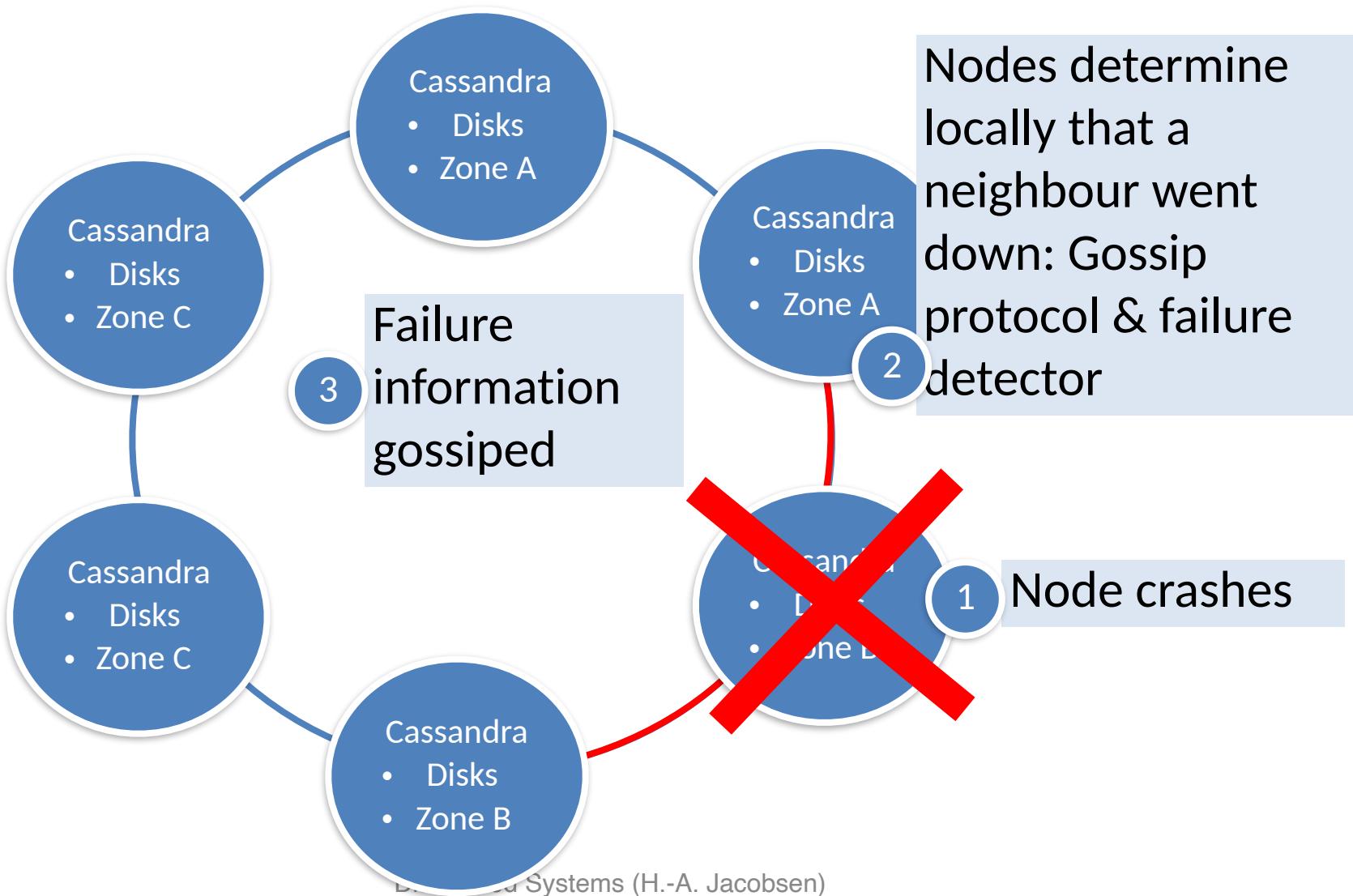


Incremental scaling in Cassandra

(i.e., adding a storage unit)



Storage unit failure



Core mechanisms

- Decentralized load balancing and scalability
 - Cf. [Consistent hashing](#) and [peer-to-peer](#)
- Read/write reliability
 - Cf. [Replication](#)
- Membership management
 - Cf. [Gossip protocols](#)
- Eventual consistency model
 - Cf. [Consistency](#)

ADMINISTRATIVE REMARKS

In a nutshell

Moodle: <https://www.moodle.tum.de/course/view.php?id=49093>

For slides, information, questions, etc.

Tutorials (starting soon)

About 15 lectures

Lecturer:
Concepts,
principles,
algorithms, etc.

Cloud-DB Lab,
Middleware Course,
Blockchain Seminar

12 Assignments
(starting soon)

Reading material

Exam (in February)

Instructor Team

Dr. Doblander, Pezhman Nasirifard and myself



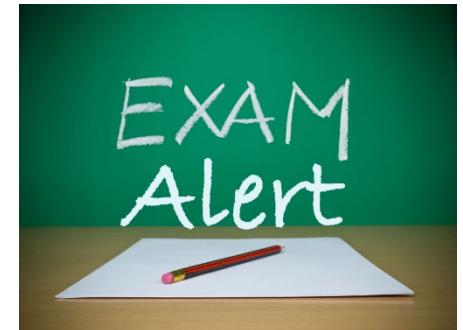
Additional tutors for tutorials

Assignments

- About one assignment set per lecture throughout the semester
- Set is usually discussed in the tutorial following the week of its release
- Assignments are **not marked**, solutions are discussed in tutorials

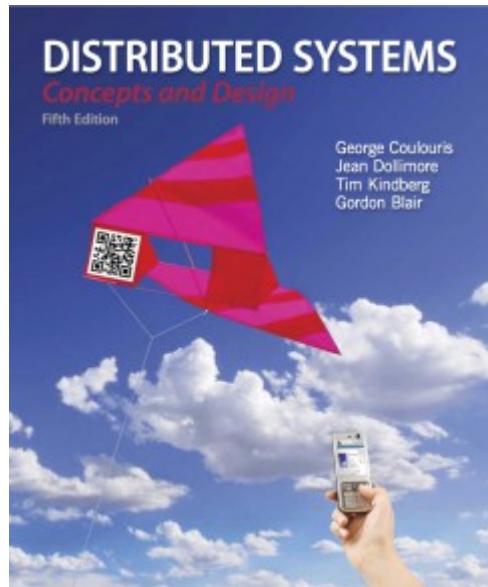
Assignments start soon

- First set of assignments released soon
- Tentative list of assignment sets on:
 - Time & clock synchronization
 - Coordination
 - RSM & Paxos
 - Replication & gossip
 - Consistency & transactions
 - Consistent hashing
 - Peer-to-peer principles
 - MapReduce
 - Publish/Subscribe
 - Blockchains
 - Cloud computing

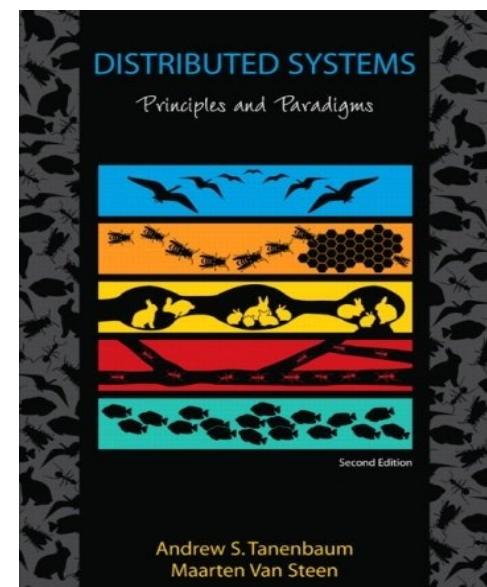


Reading material

- **Required reading** posted with each lecture (for the exam)
- **Recommended reading** posted with each lecture (for life)
- Reading draws from selected sections of the below two books and from online resources



Distributed Systems (H.-A. Jacobsen)



Andrew S. Tanenbaum
Maarten Van Steen

Final exam

- **11.02 and 17.04 (no guarantee, check online)**
- Closed book
- Covers lectures, tutorials and assignments
- You are responsible for
 - All **required reading** material assigned
 - All **lecture and tutorial** material
- Final review (last lecture)

Out of scope for us

- Many interesting middleware abstractions
(see Middleware course IN2258)
- Parallel and concurrent programming
- Networking / Socket Programming
(see Cloud-DB course IN0012, IN2106,
IN4163)
- Security – “The network is secure, is it ☺?”

Tentative course outline

- Time in distributed systems
- Coordination and agreement
- Consensus, Paxos, replication
- Consistency and transactions
- Consistent hashing, CAP theorem, web caching
- Distributed file systems (GFS)
- MapReduce, Spark
- Peer-to-peer systems, distributed hash tables (DHTs)
- Distributed publish/subscribe systems
- Blockchains
- Cloud technologies, software-defined networking

THE END

Distributed Systems: Time

"Time has been invented in the universe so that everything would not happen at once."

"There is no change without the concept of time, and there is no movement without time."

Agenda

- Clocks & time in computers
- Synchronizing physical clocks
- Logical clocks
 - Lamport clocks
 - Vector clocks

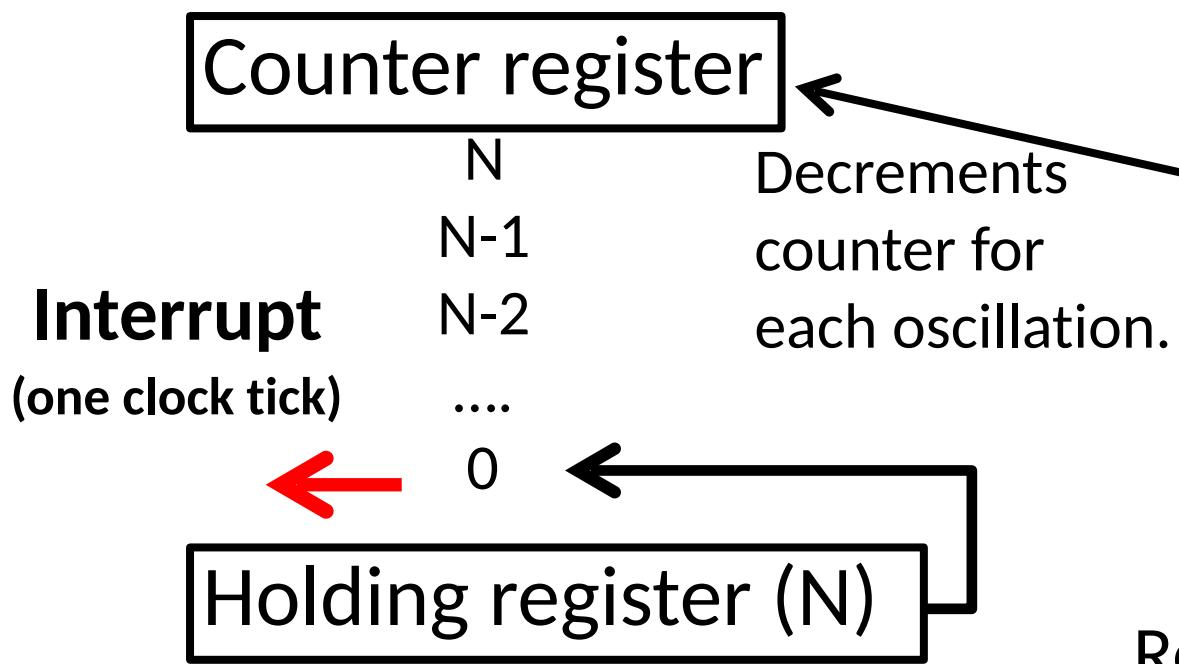
Why is time important?

- In distributed systems, we require...
 - Cooperation between nodes: must **agree** on certain things
 - **High degree of parallelism**: nodes should work independently to achieve the most progress
- Time gives us...
 - A **point of reference** every machine knows how to keep track of...
 - Without explicit communication!
- However...
 - Time-keeping is not perfect ☺
 - How do we efficiently synchronize time to achieve our goals?

Computer “clocks”

Computer clocks count oscillations
of a crystal at a defined frequency

Crystal oscillator
(quartz crystal)

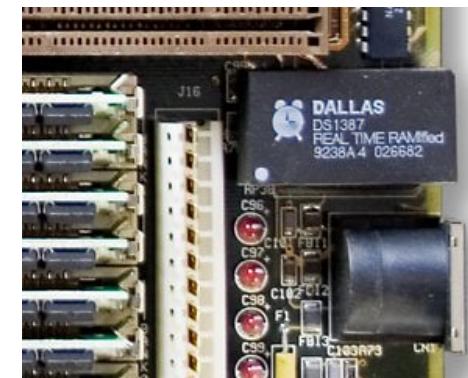


frequency often
32.768 kHz
(2^{15} cycles per sec.)

Reload upon interrupt

Real time clock (RTC, CMOS, HWC)

- RTC is used even when the PC is hibernated or switched off
 - Based on alternative low power source
 - Cheap quartz crystal (<\$1), inaccurate (+/- 1-15 secs/day)
- Referred to as “**wall clock**” time
 - Synchronizes the **system clock** when computer on
 - Should not be confused with **real-time computing**
- IRQ 8
- sysfs interface **/sys/class/rtc/rtc0 ... n**
UNIX: cat /sys/class/rtc/rtc0/since_epoch
/sys/class/rtc/rtc0/wakealarm



Source: Wikipedia

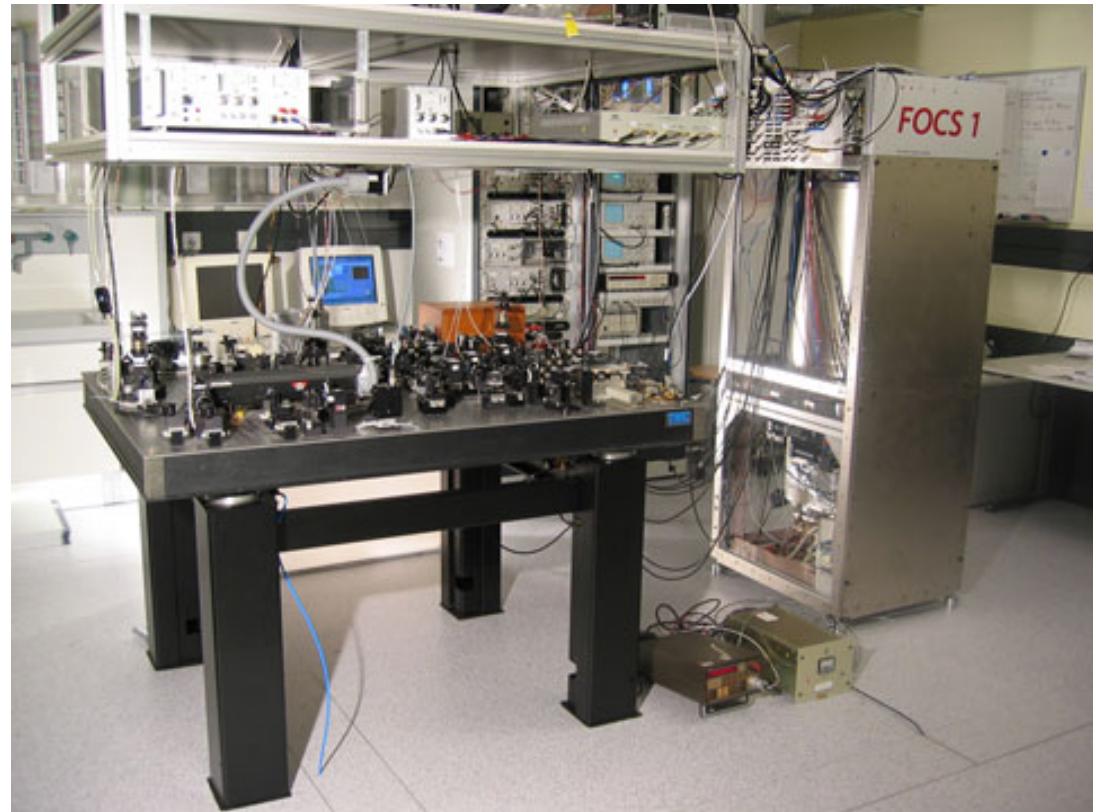
Universal Time Coordinated (UTC)

Temps Universel Coordonné

- **Universal**
 - Standard used around world & Internet (e.g., NTP)
 - Independent from time zones (UTC 0)
 - Converted to local time by adding/subtracting local time zone (EST: UTC-5; CET: UTC+2)
- **Coordinated**
 - 400 institutions contribute their estimates of current time (using **atomic clocks**)
 - UTC is built by combining these estimates

Caesium-133 fountain atomic clock in Switzerland

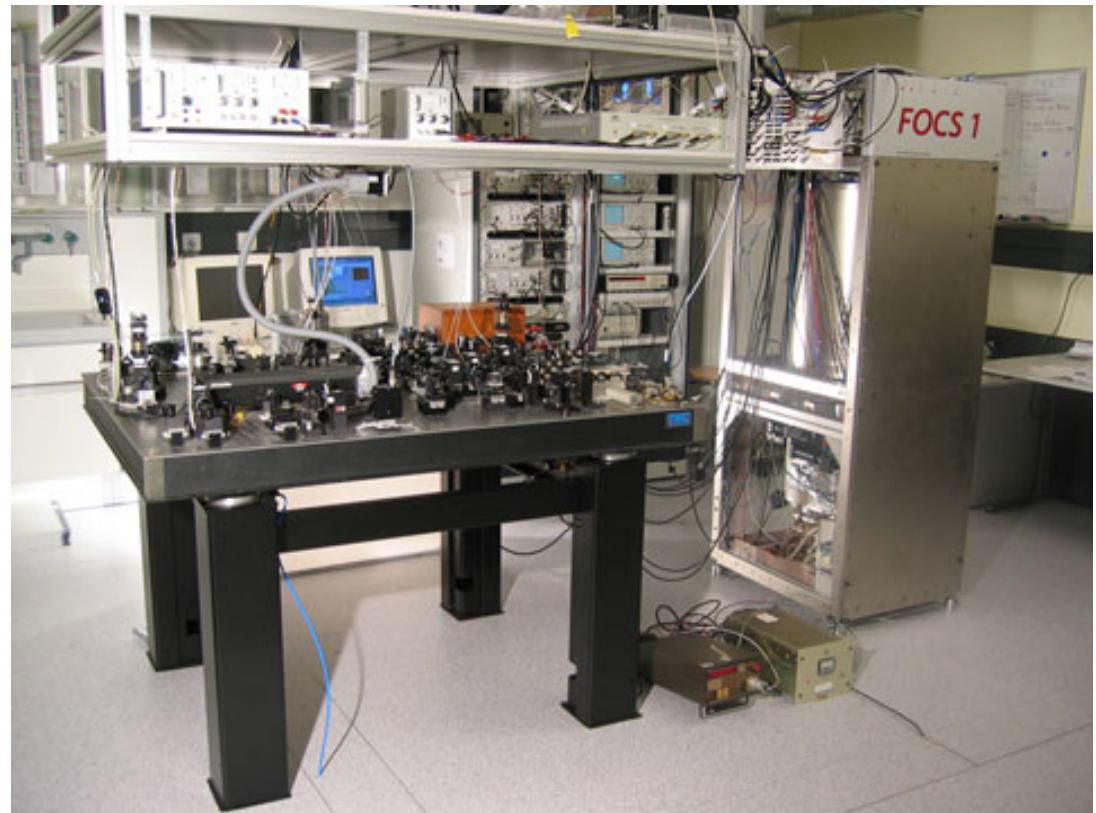
Uncertainty of one second in 30 million Years!



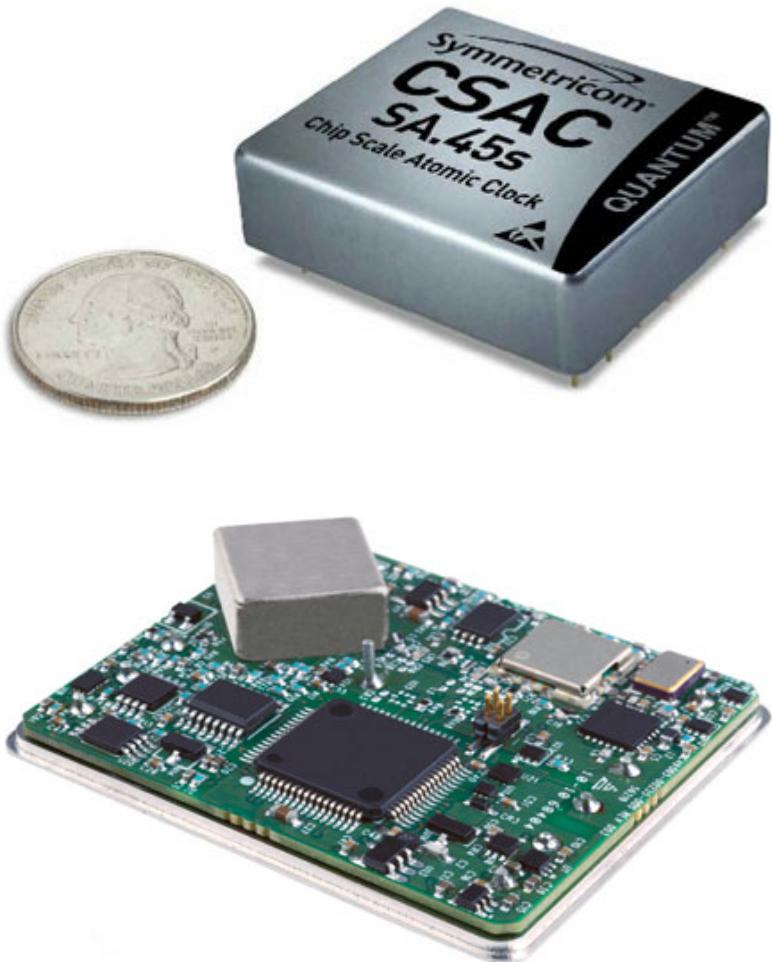
Caesium-133 fountain atomic clock in Switzerland

Uncertainty of one second in 30 million Years!

(Probably the ``Rolex`` of atomic clocks ☺)



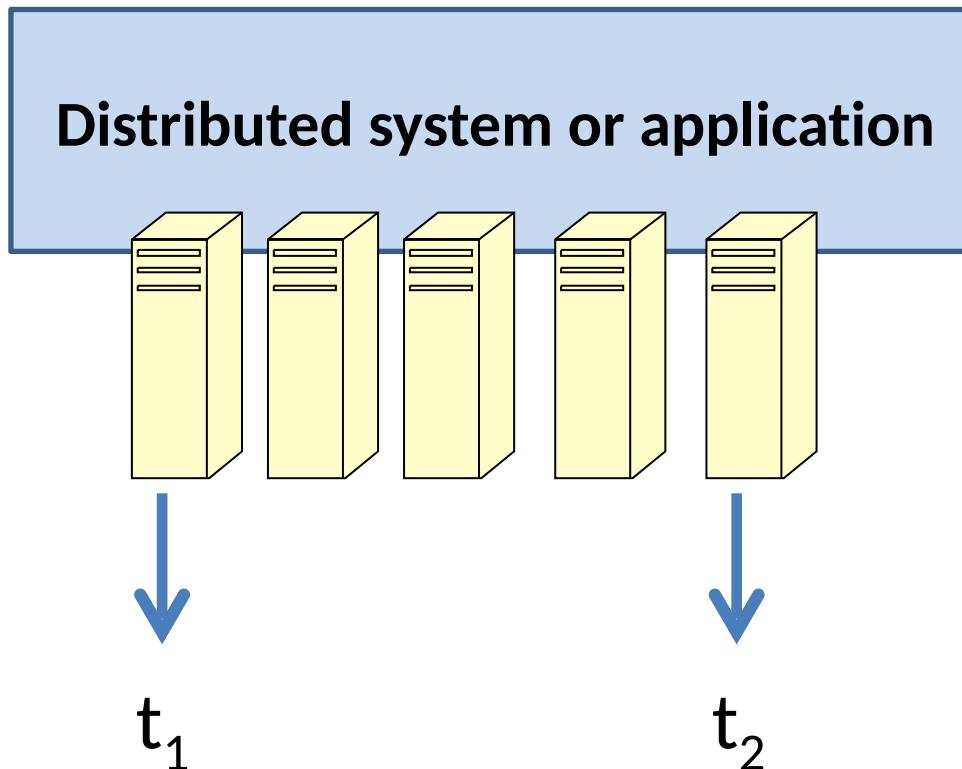
Atomic clocks



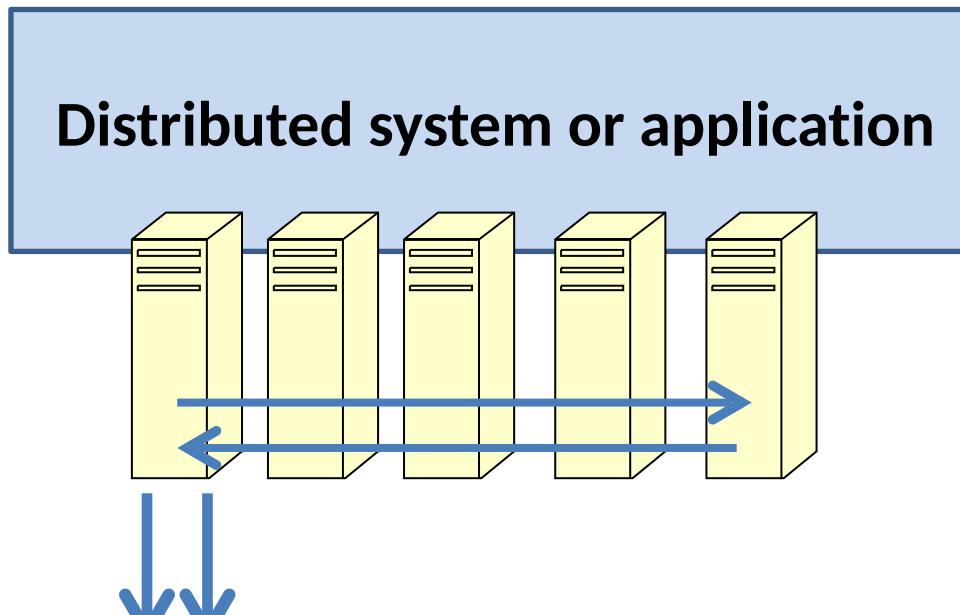
Atomic clock on the market
May 11, 2011.
Quoted \$1500 with an accuracy of less than 0.5 micro seconds per day.

Chip-Scale Atomic Clock. The ultimate in precision--the caesium clock--has been miniaturized By Willie D. Jones
Posted 16 Mar 2011.

Measuring latency in distributed systems experiments



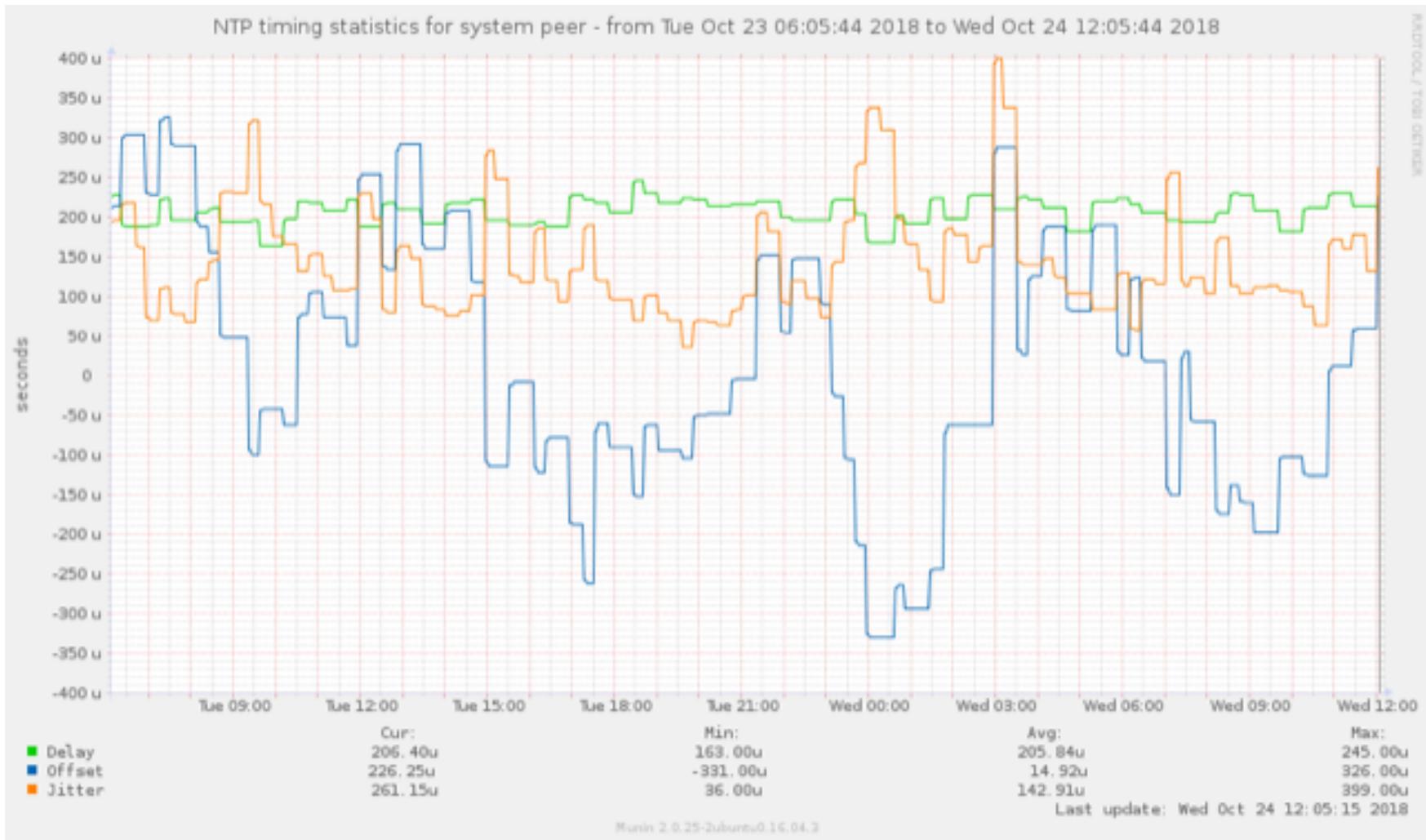
Measuring latency in distributed systems experiments



host=node-1 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-2 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-3 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-4 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-5 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-6 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-7 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-8 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-9 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-10 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-11 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-12 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-13 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-14 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-15 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-16 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-17 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-18 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-19 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-20 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-21 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-22 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-23 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-24 rtt=750(187)ms/0ms delta=0ms/
0ms /0ms delta=0ms/0ms

host=node-25 rtt=750(187)ms/0ms delta=1ms/1ms
host=node-26 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-27 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-28 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-29 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-30 rtt=750(187)ms/0ms delta=-1ms/-1ms
host=node-31 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-32 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-33 rtt=750(187)ms/0ms delta=-1ms/-1ms
host=node-34 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-35 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-36 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-37 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-38 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-39 rtt=562(280)ms/0ms delta=0ms/0ms
host=node-40 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-41 rtt=750(187)ms/0ms delta=0ms/0ms
host=node-42 rtt=562(280)ms/0ms delta=290ms/290ms
host=storage-1 rtt=562(280)ms/0ms delta=0ms/0ms
host=storage-2 rtt=750(187)ms/0ms delta=0ms/0ms
host=storage-3 rtt=750(187)ms/0ms delta=0ms/0ms
host=storage-4 rtt=562(280)ms/0ms delta=0ms/0ms

NTP timing statistic in microseconds



Clock skew & drift

- **Clock skew:** Instantaneous difference between readings of two clocks
- **Clock drift:** Rate at which clock skew increases between a clock and some reference clock

Synchronization mechanisms

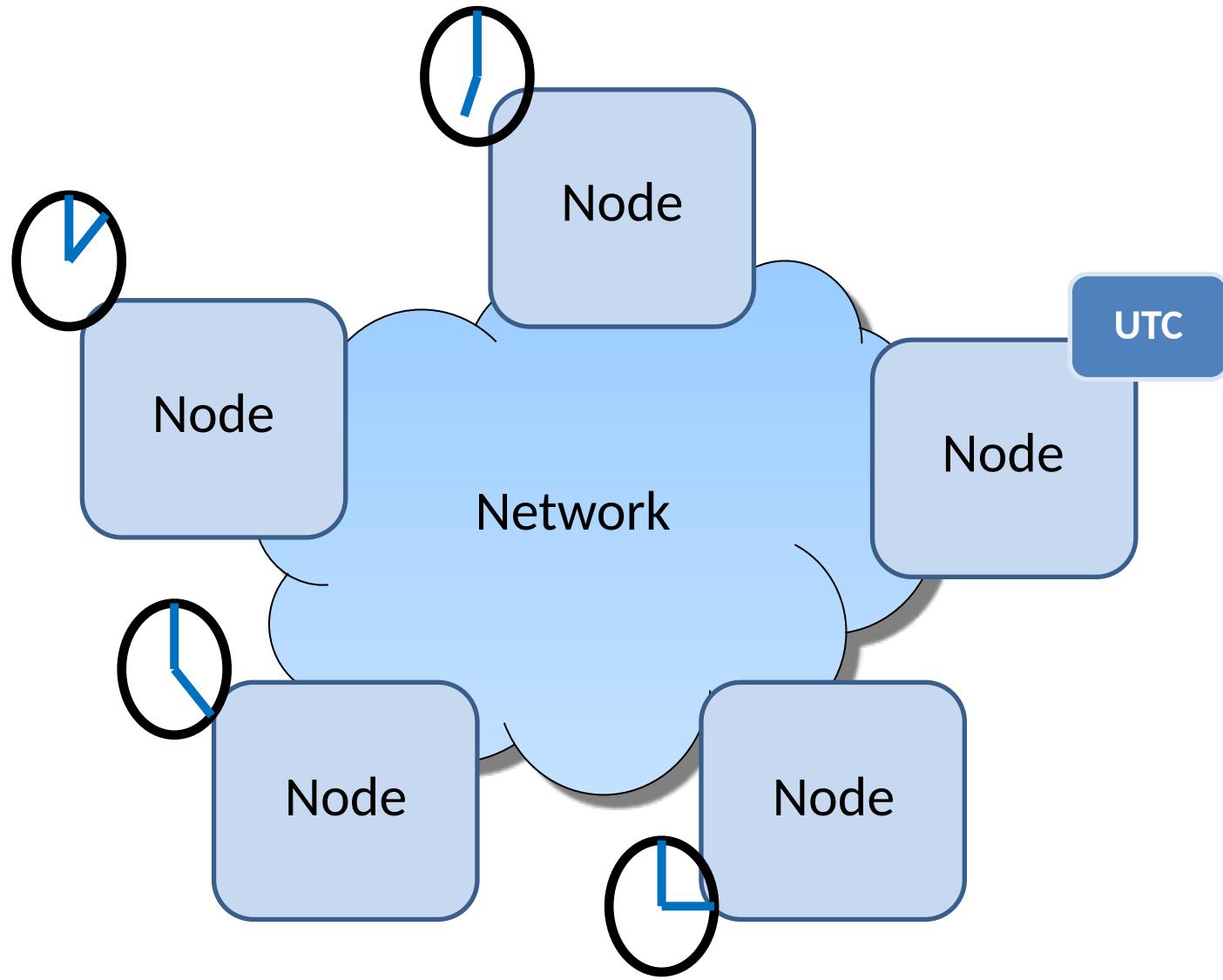
- Hardware support: Radio receivers, GPS receivers, atomic clocks, shared backplane signals
 - Tight synchronization
($\pm 10\text{ns}$ for GPS)
 - Negligible overhead
 - Costly
 - Quartz is good enough



Software synchronization properties

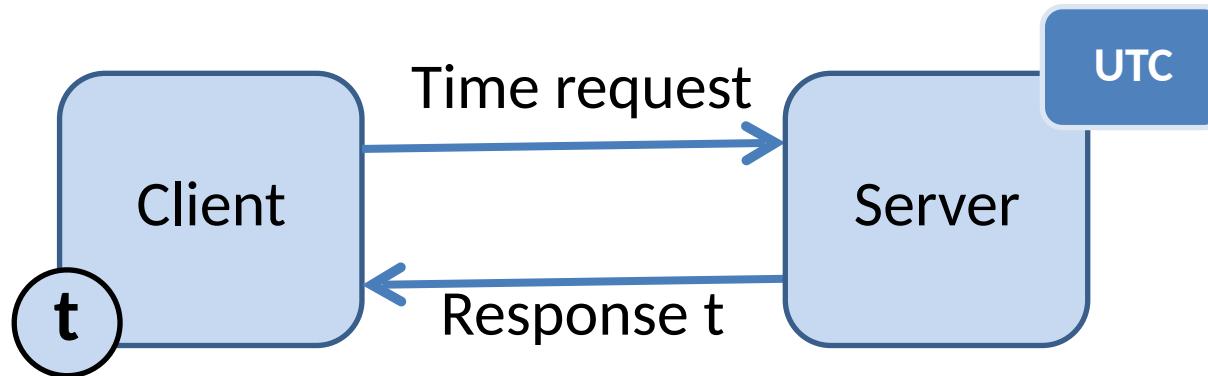
- External & internal synchronization
- Adjusting the clock is not straight forward
 - Set *clock* = *new clock value*
 - Time must **increase monotonically**
- **Can't go back to the past**
 - Timestamps are important, can't repeat them
- Time should not **show sudden jumps**
 - Cannot jump on your way to the future
 - Lose the present time & miss a deadline

Problem: External synchronization



Synchronization request and reply

- Request to reference clock source for time
 - Request involves network round trip time (RTT)
 - Response is wrong by the time client receives it



- Client must adjust response based on knowledge of network RTT

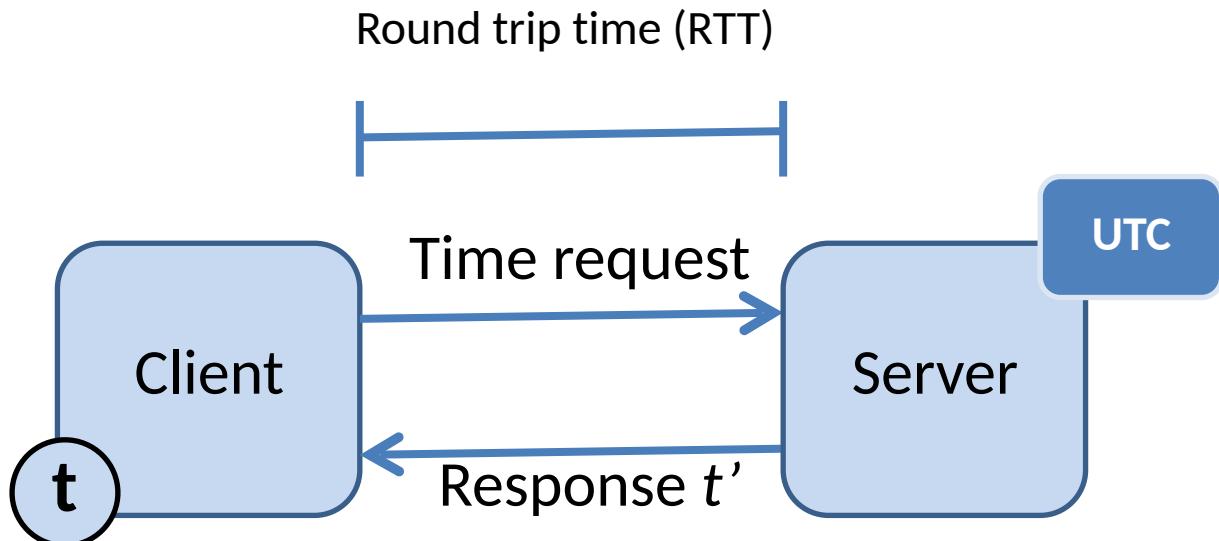
Probabilistic clock synchronization

Proposed by IBM, 1989

- External clock synchronization
- Time server connected to a time reference source (UTC)
- Transmission time is unbounded, but usually reasonably short
- Synchronization is achieved to the degree that network delays are short compared to desired accuracy
- Primarily intended for operation on LANs

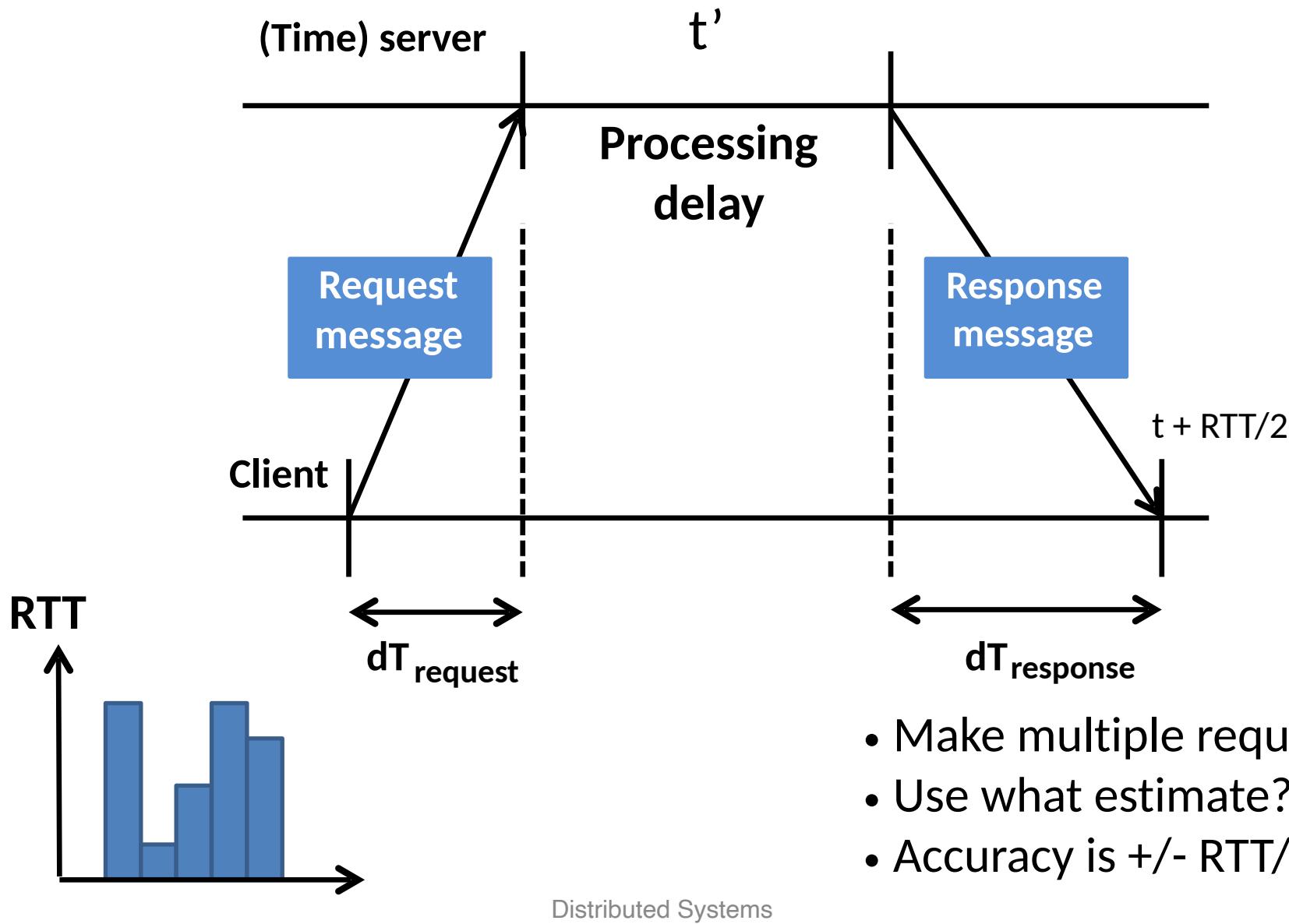
Cristian's algorithm (1989)

- Client measures **round trip time** for request to server
- Server responds with time value
- Client assumes transmission **delays split equally**
- Could factor in time to process request at server



$$t = t' + \text{RTT}/2$$

RTT: Accuracy estimation



New time

- Time request at T_0
- Time response at T_1
- Assume network delays are symmetric ($RTT/2$)
- Estimated overhead due to network delay is $(T_1 - T_0)/2$
- $T_{\text{new}} = T_{\text{server}} + (T_1 - T_0)/2$

Result accuracy bound

- T_{\min} minimum network delay
- T_0, T_1 as above, assume T_{\min} for propagation
- **Earliest time** server could generate time stamp: $T_0 + T_{\min}$
- **Latest time** the server could generate the time stamp: $T_1 - T_{\min}$
- Range: $T_1 - T_{\min} - (T_0 + T_{\min}) = T_1 - T_0 - 2T_{\min}$
- **Accuracy:** $\pm |(T_1 - T_0)/2 - T_{\min}|$

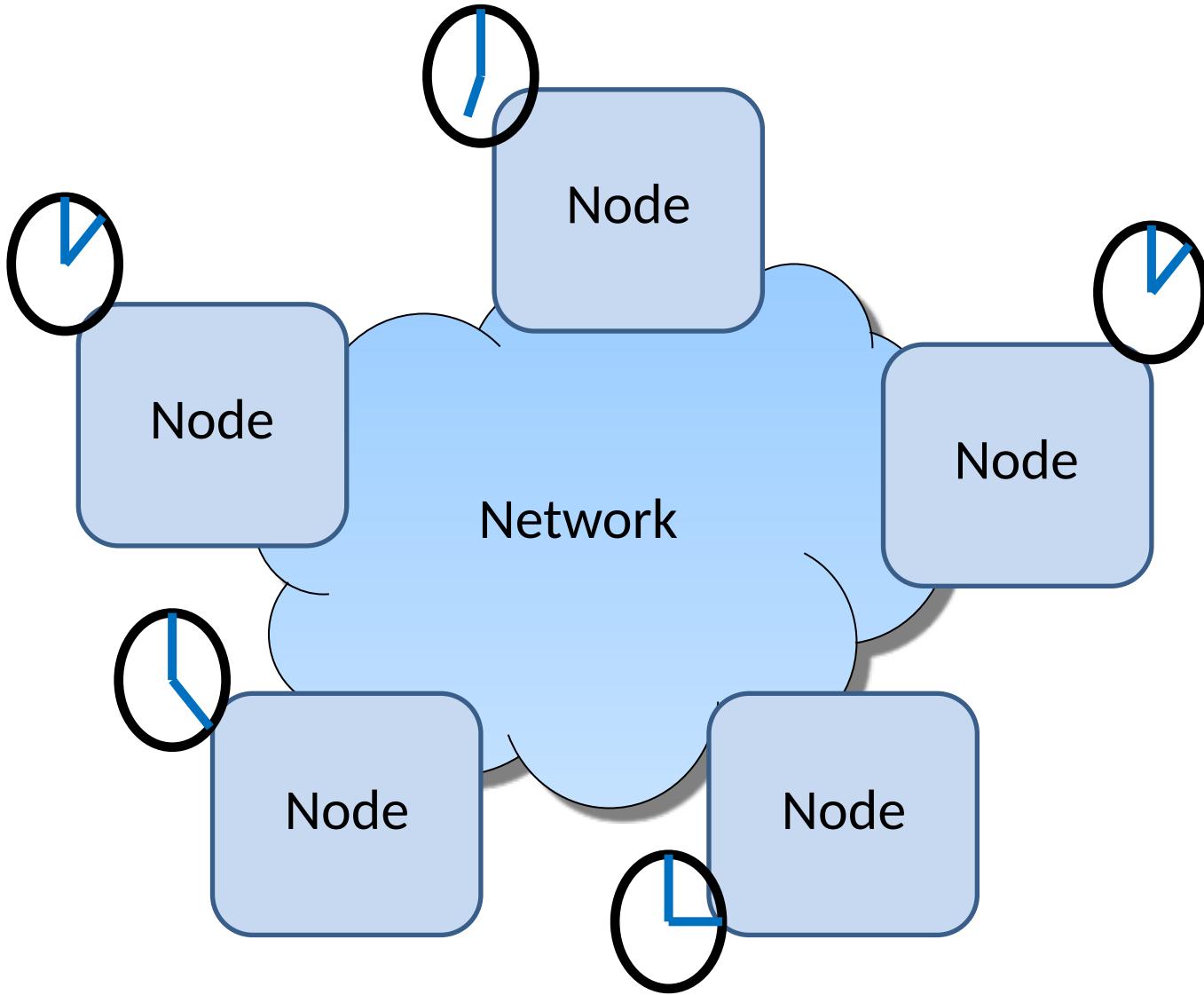
Errors are cumulative

- Say node A synchronizes time with node B with an accuracy of +/- 5 msec
- Node B synchronizes its time with node C with an accuracy of +/- 7 msec
- Then, the net accuracy at node A is $+/- (5+7)$ msec = $+/- 12$ msec

Problems with Cristian's algorithm

- **Centralized time server**
 - Distribution possible via broadcast to many servers
 - Communication with single server is preferred
 - Simpler approach
 - Better estimates based on series of requests
- Time server is trusted & **single point of failure**
 - Malicious, failed server would wreak havoc

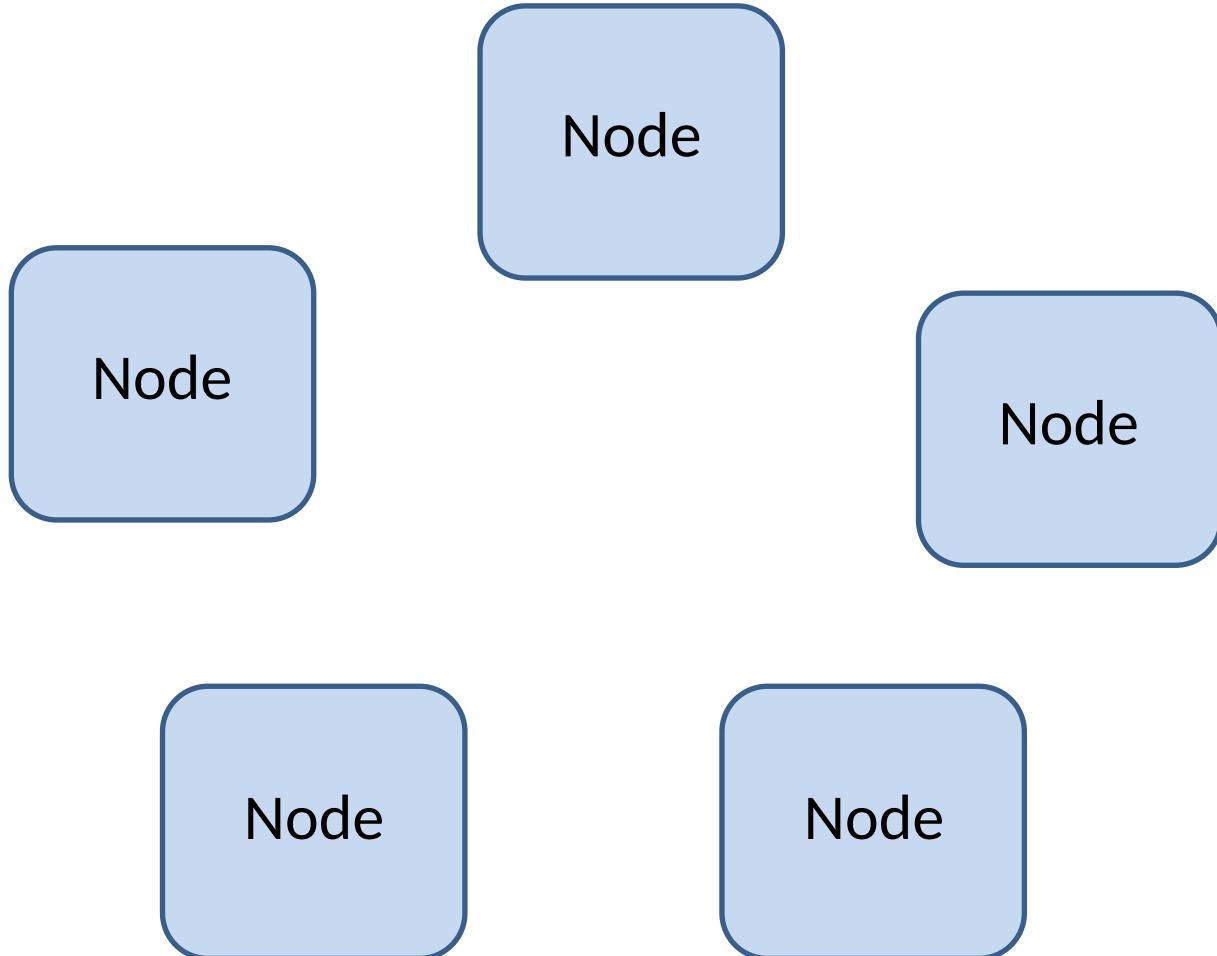
Problem: Internal synchronization



Berkeley algorithm overview

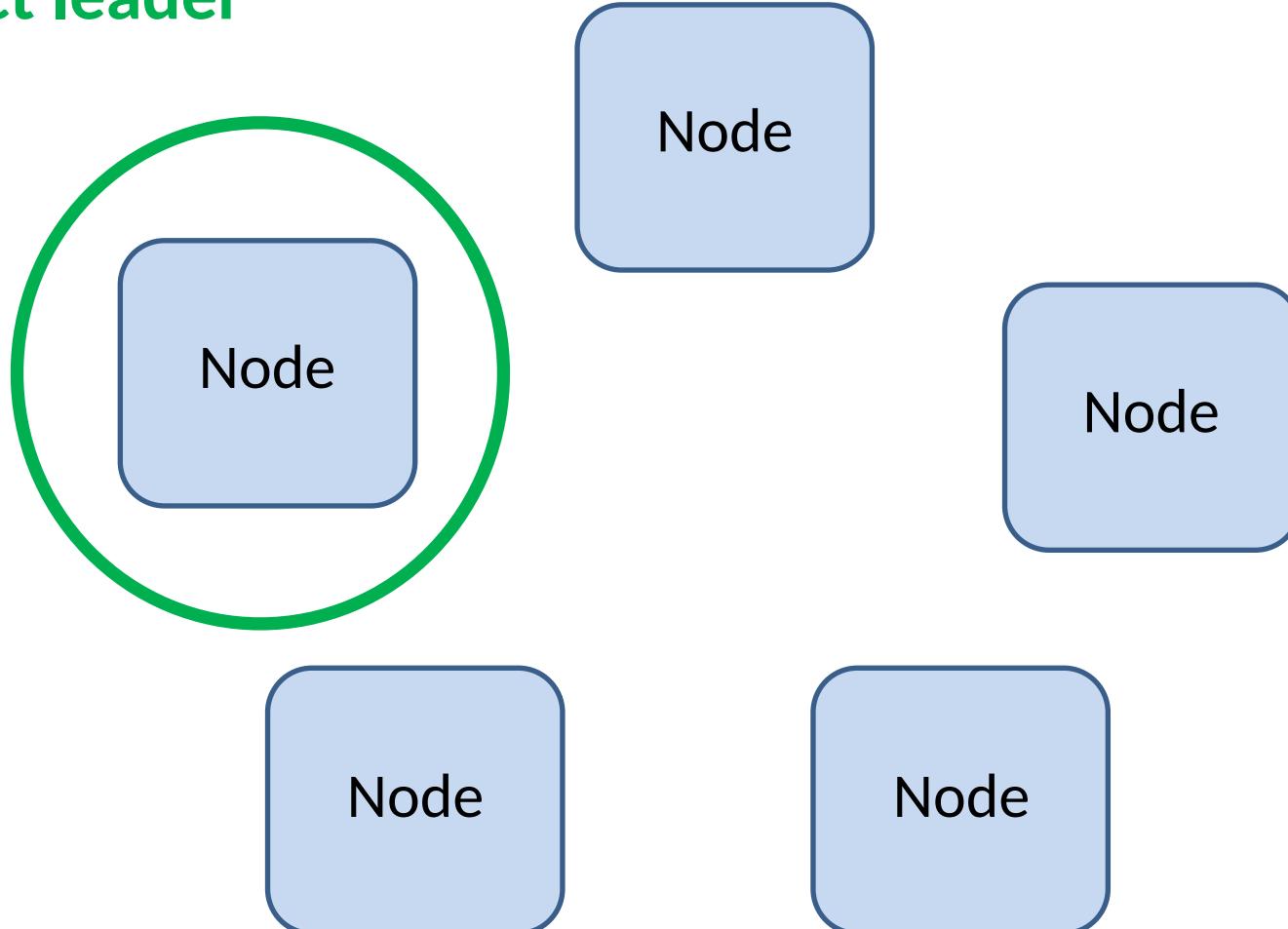
- **Clock synchronization** algorithm developed in 1989 as part of Berkeley Unix efforts
- **Internal synchronization**
- Assumes no machine has accurate time source
- Performs internal synchronization to **set clocks** of all machines **to within a bound**
- Intended for use in intranets / LANs
- Experimentally: Synchronization of clocks to within 20-25 ms on LAN

Berkeley algorithm: Request phase



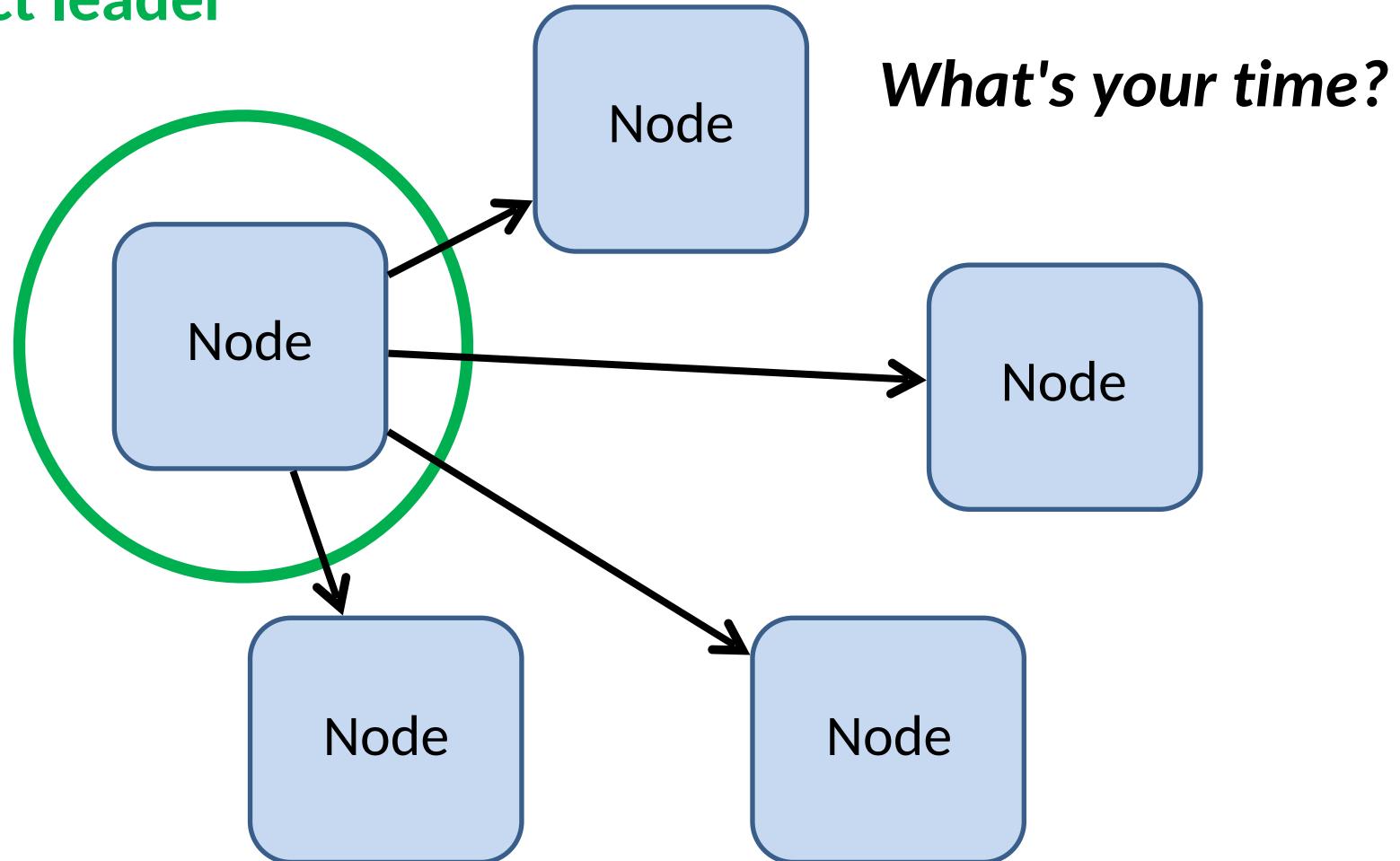
Berkeley algorithm: Request phase

Elect leader

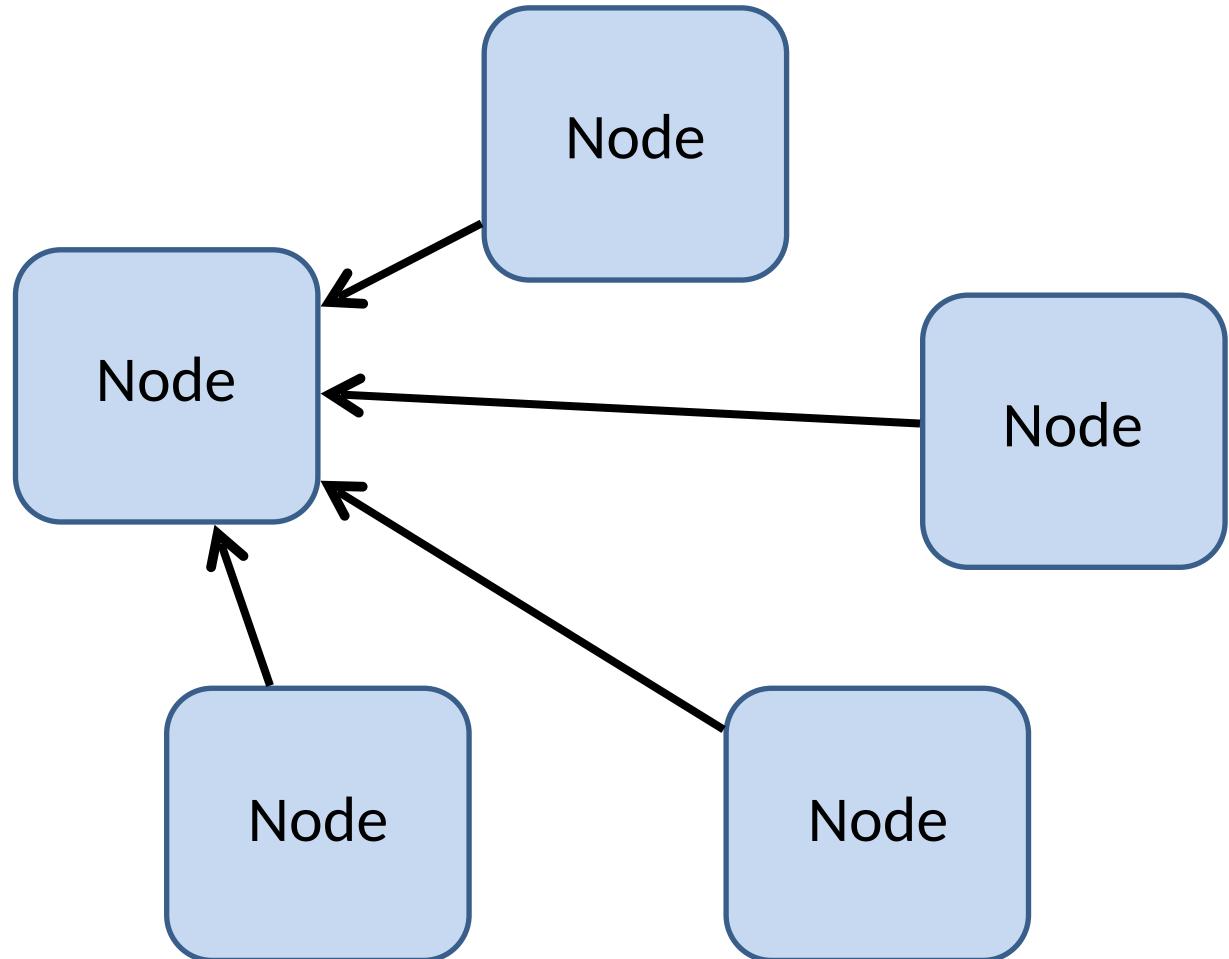


Berkeley algorithm: Request phase

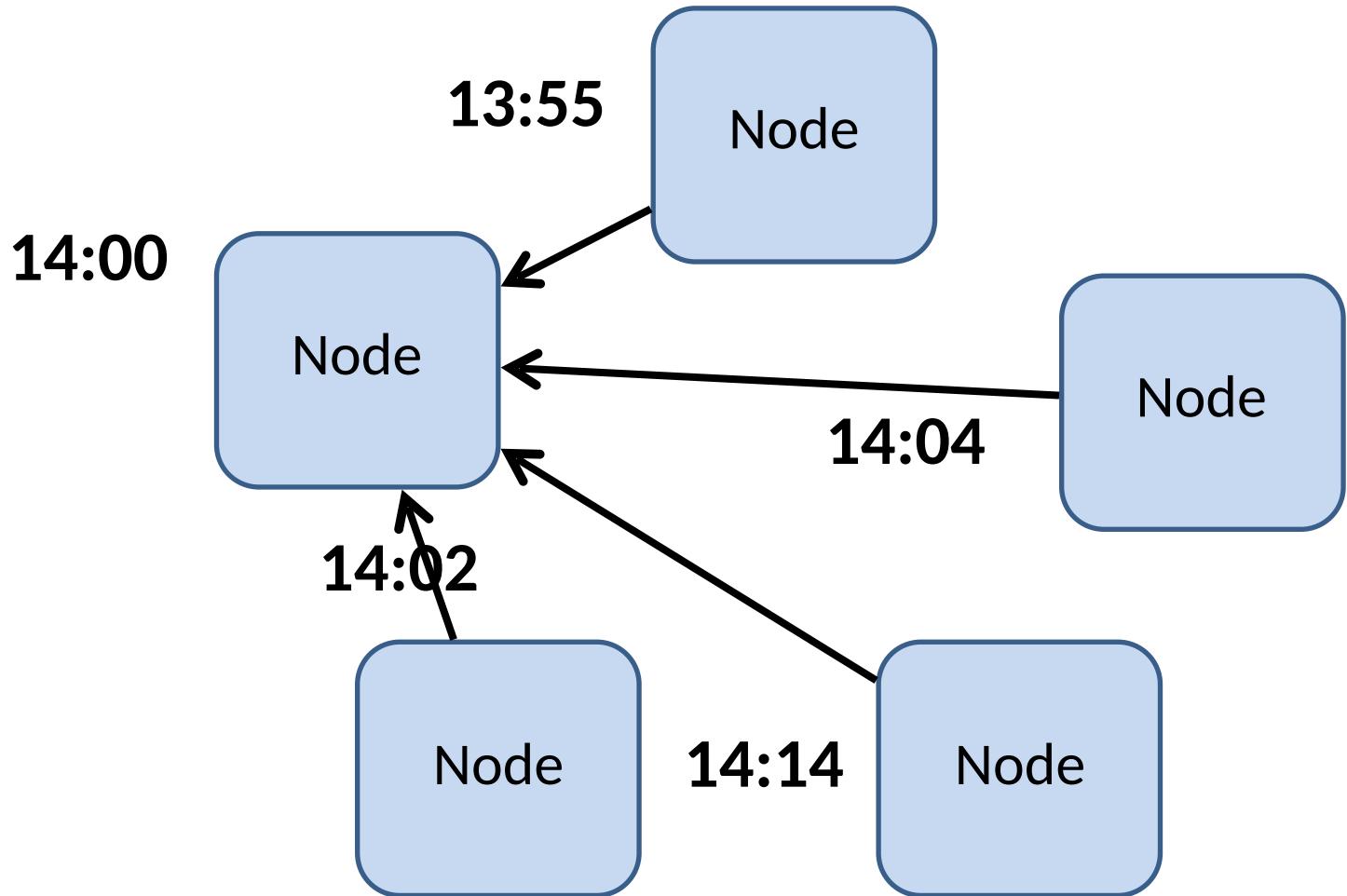
Elect leader



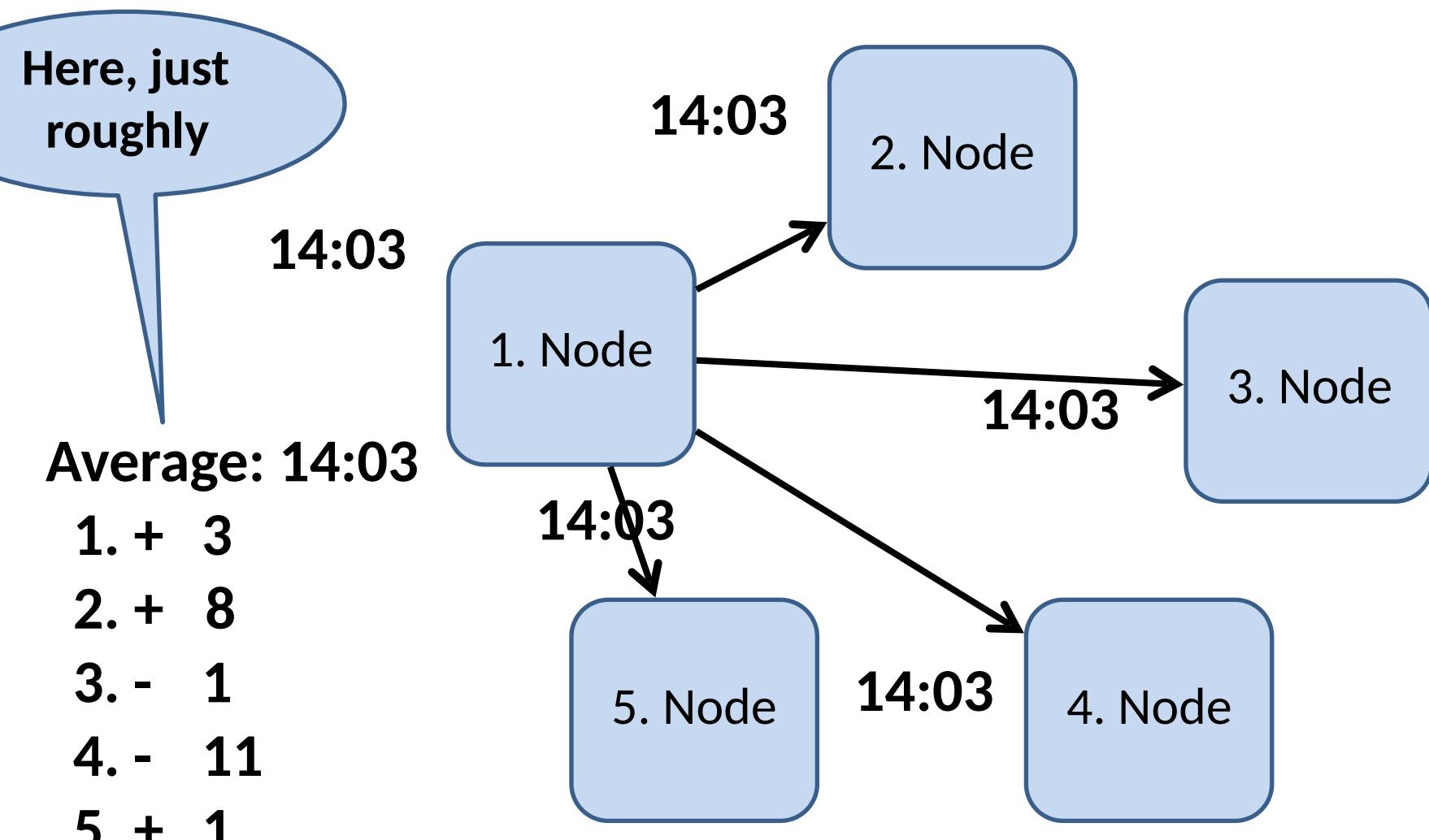
Berkeley algorithm: Reply phase



Berkeley algorithm: Reply phase



Berkeley algorithm: Clock adjustment



Berkeley algorithm: Summary

- Leader is chosen (via an **election process**)
- Leader polls nodes who reply with their time
- Leader **observes RTT** of messages and estimates time of each node and its own
- Leader **averages clock times, ignoring any values it receives far outside values of others** (include its own)
- Leader sends out **amount** (positive or negative) that each node must adjust its clock
- Avoids further uncertainty due to RTT at nodes
- Accuracy originally reported was 20-25 ms (15 nodes)

Network time protocol (NTP)

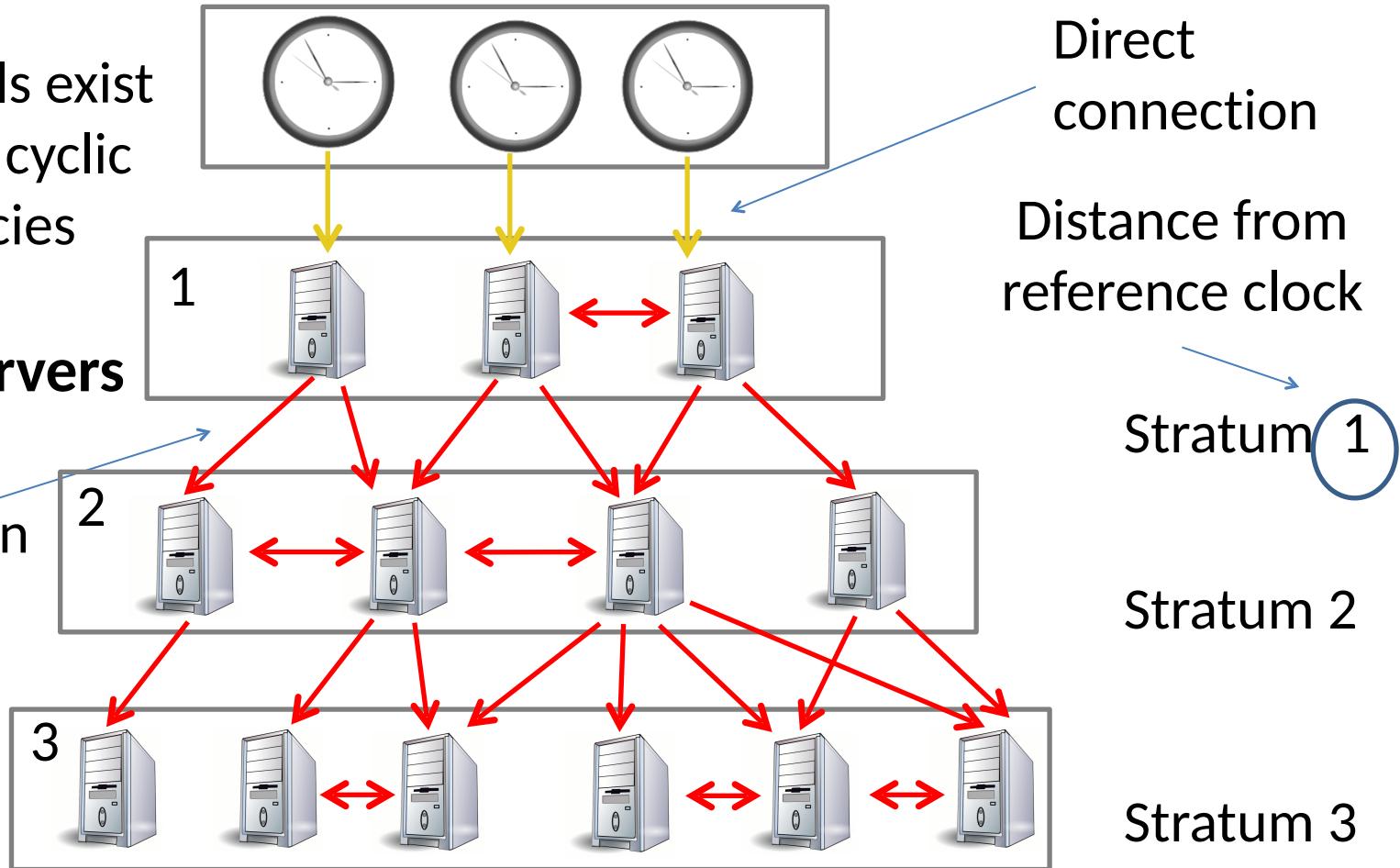
[D. Mills, 1994++]

- Service that provides UTC time over networks
- Hierarchical distributed system which provides scalability, fault-tolerance and security
- Time service and dissemination of time on Internet
- Maintains time to within **tens of milliseconds over Internet**
- Achieves **1 millisecond accuracy in LANs** under ideal conditions
- Supported by daemon (ntpd) in user space (kernel support) via UDP on port 123

NTP: Clock Strata

Reference clock

Strata levels exist to prevent cyclic dependencies



Up to 16 levels used, more possible (stratum 16 unsynchronized)

CLOCK ADJUSTMENT

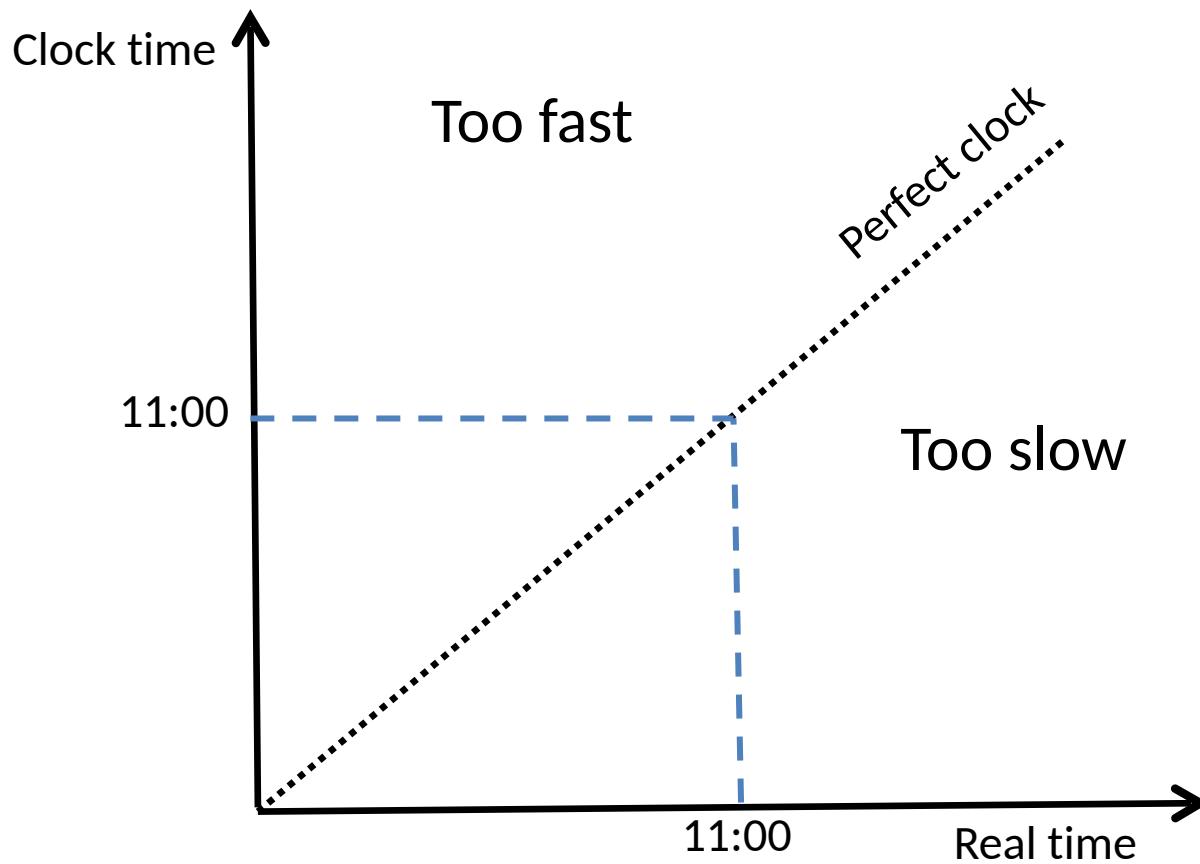
Hardware and software clocks

- At real time, t , OS reads time on computer's **hardware clock** $H_i(t)$
- Calculates time on its *software clock*:
$$C_i(t) = a * H_i(t) + b \quad (\text{for process } i)$$
- Monotonicity requirement:
$$t' > t: C(t') > C(t)$$
- Can achieve monotonicity by adjusting a and b in
$$C_i(t) = a * H_i(t) + b$$

Clock adjustment

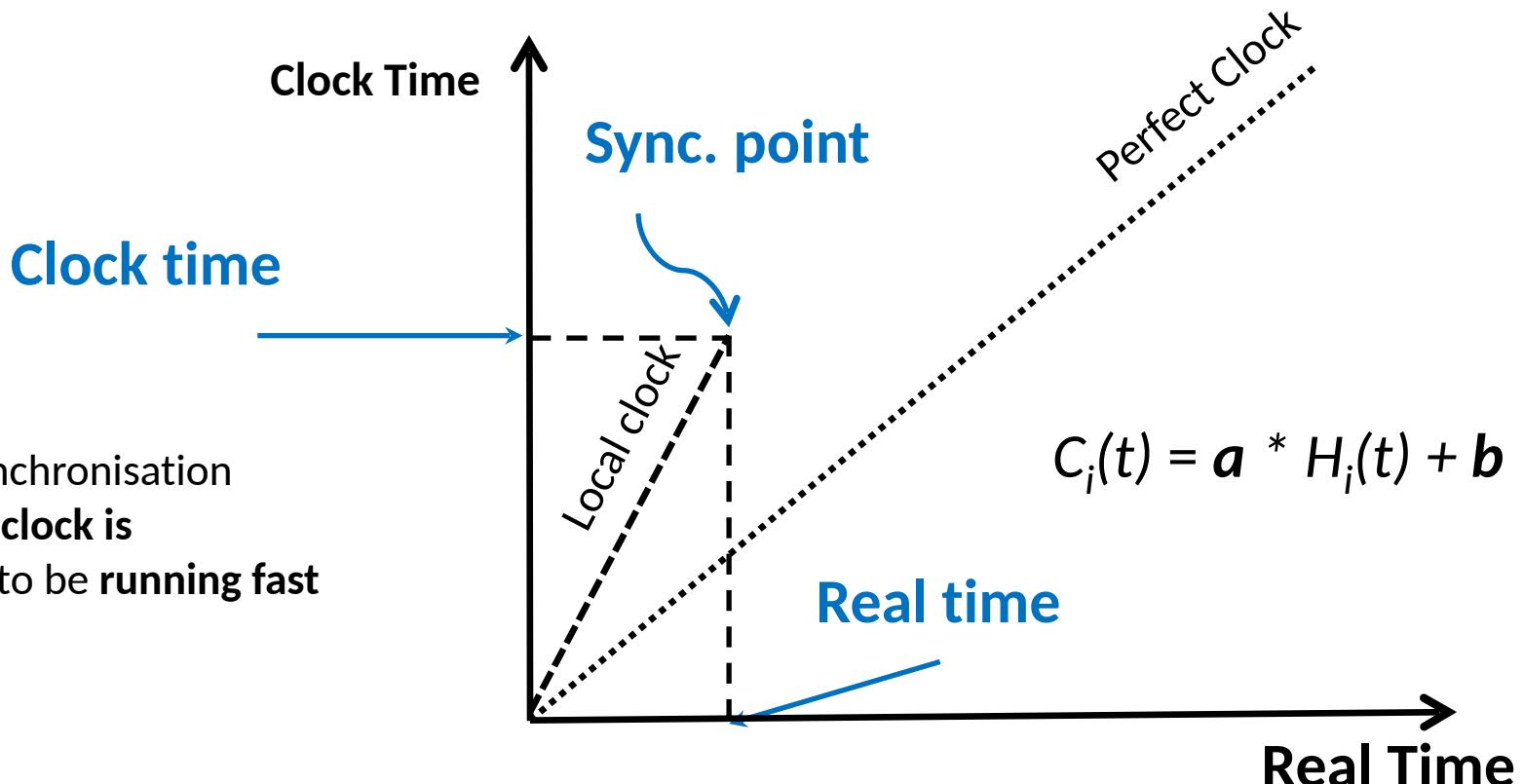
- Two parameters are a constant **offset** and a **linear slope**:
 - $C_i(t) = a * H_i(t) + b$
- The “catch up” value for scaling local time down or up in a linear fashion

Perfect time



Clock too fast

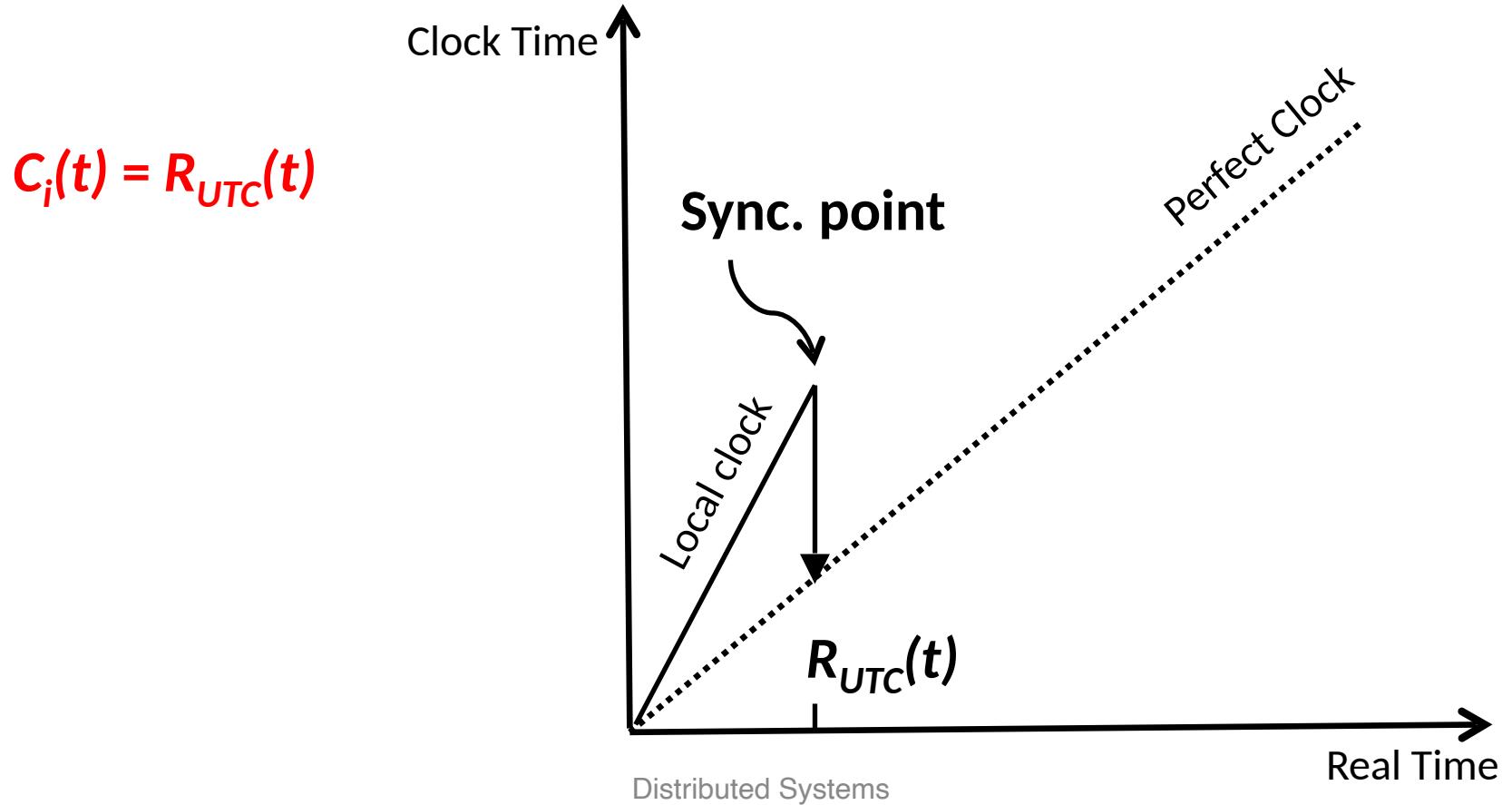
- At some synchronisation point, local **clock** is discovered to be **running fast**



- True time according to a UTC source, for example

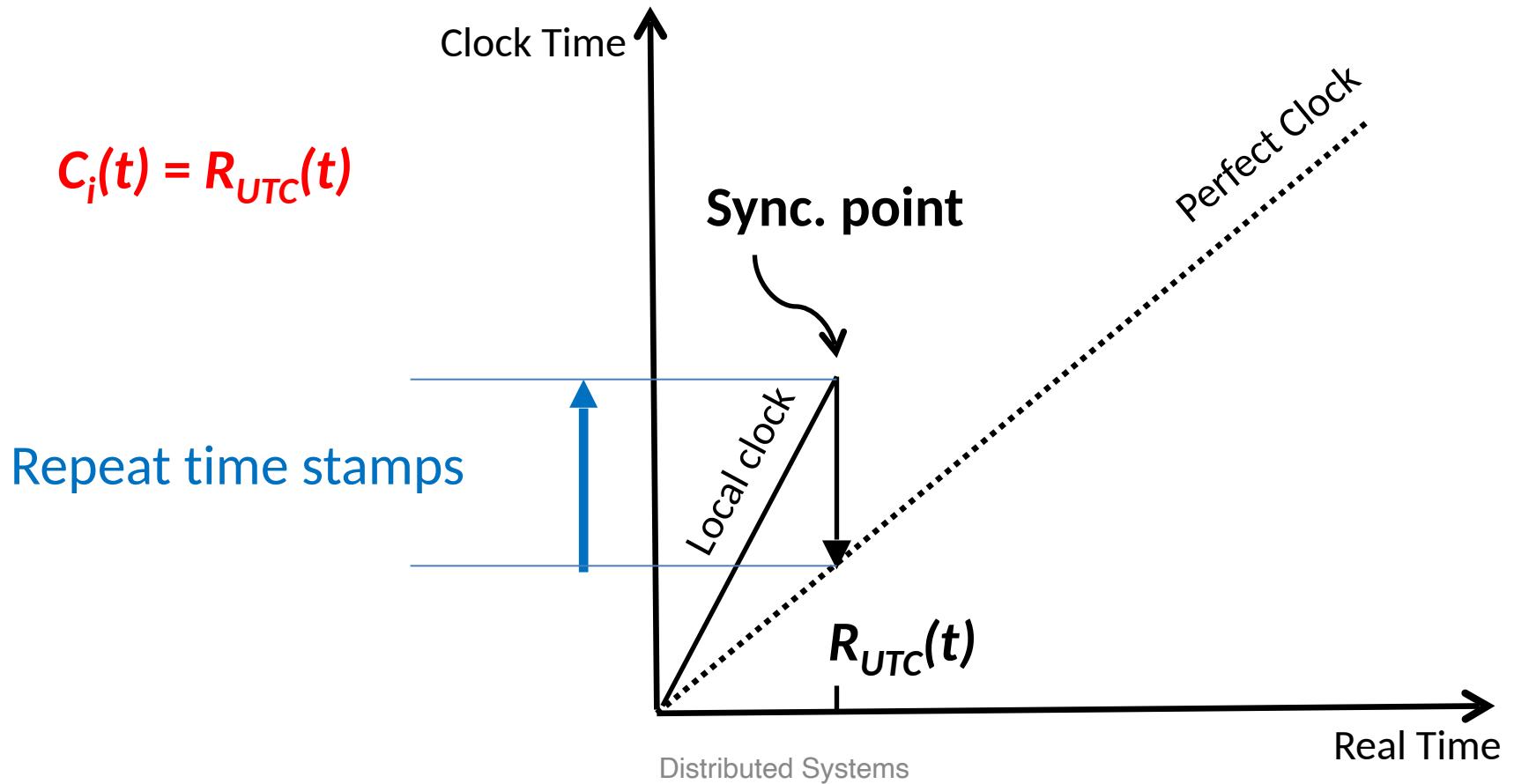
“Catch up” Example: Reset clock

- Can't just set local clock to be equal to “real time”



“Catch up” Example: Reset clock

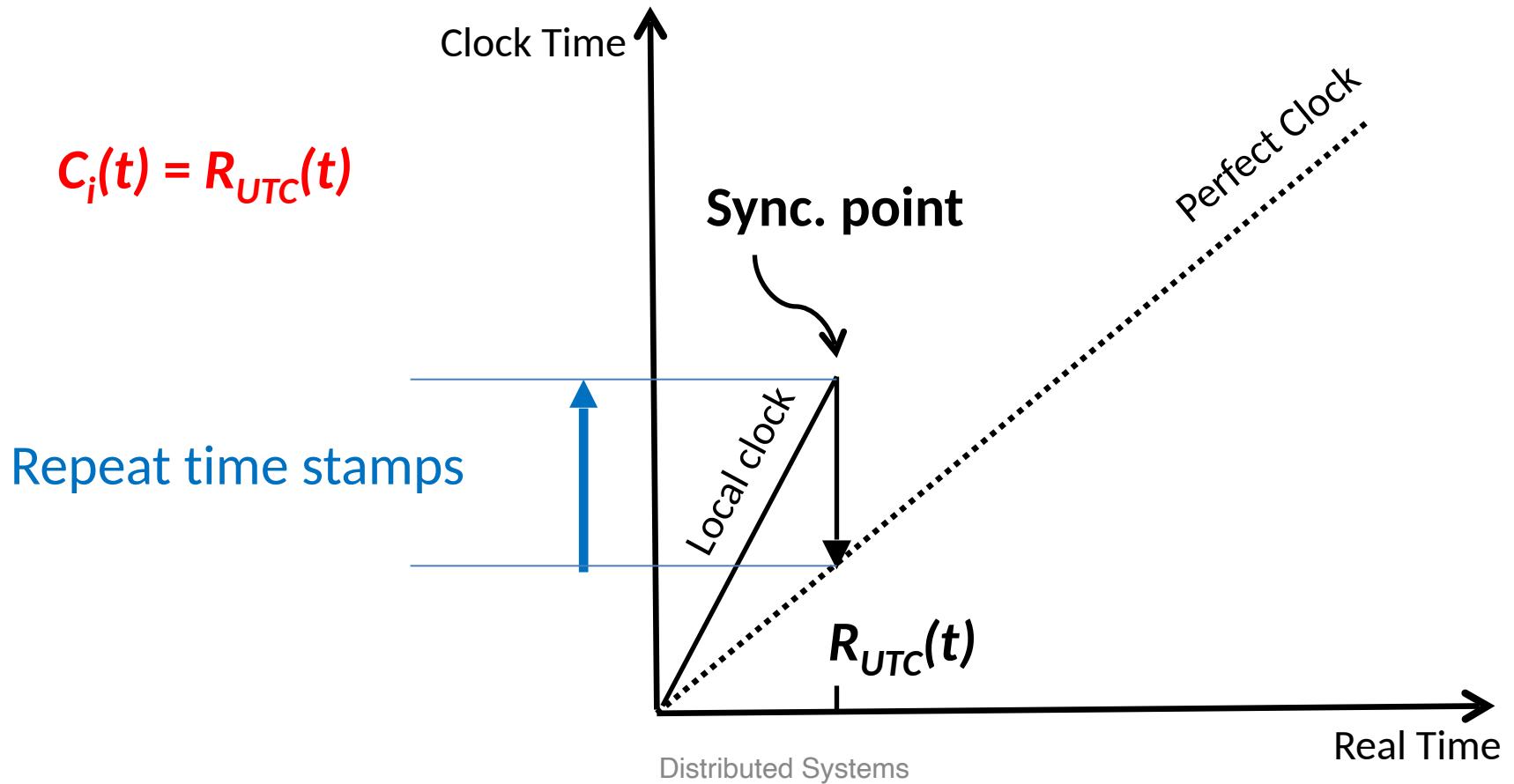
- Can't just set local clock to be equal to “real time”



“Catch up” Example: Resync

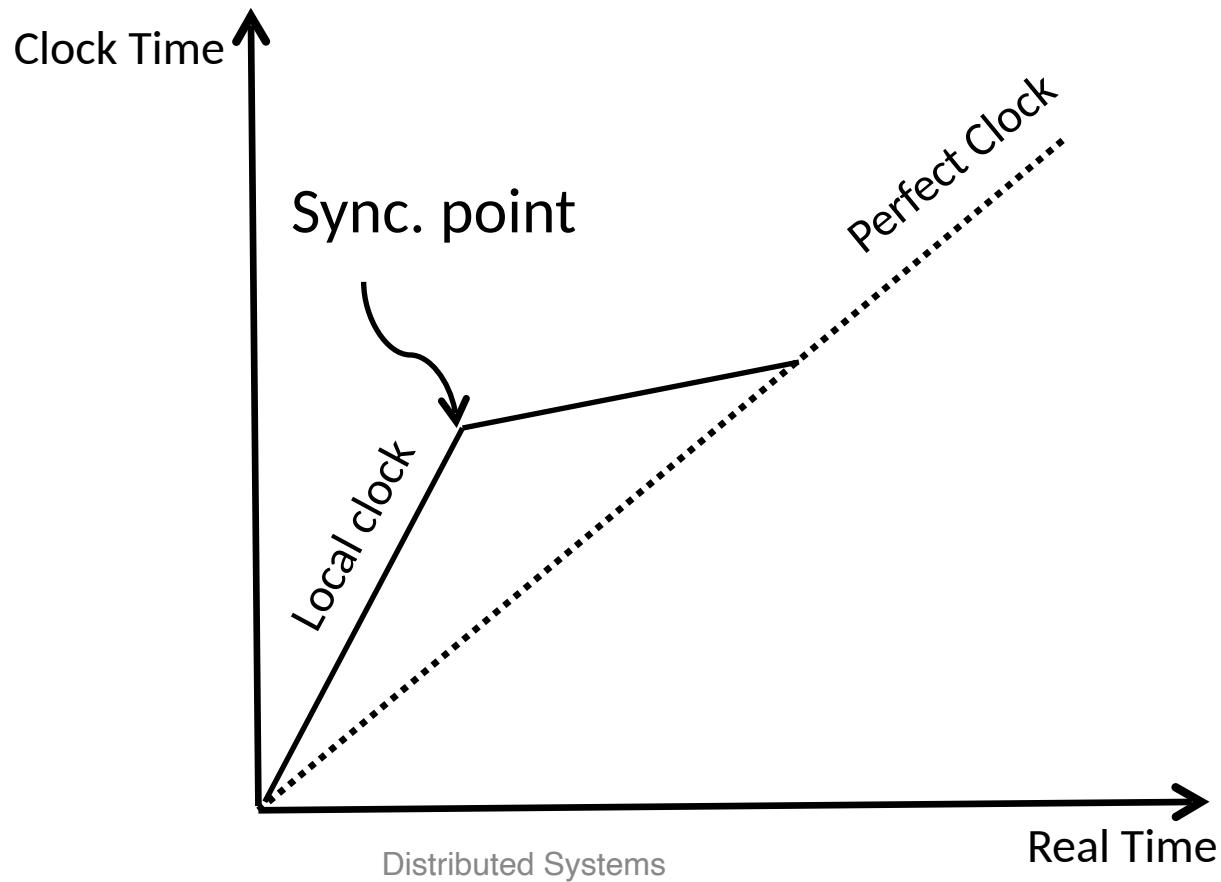


- Can't just set local clock to be equal to “real time”



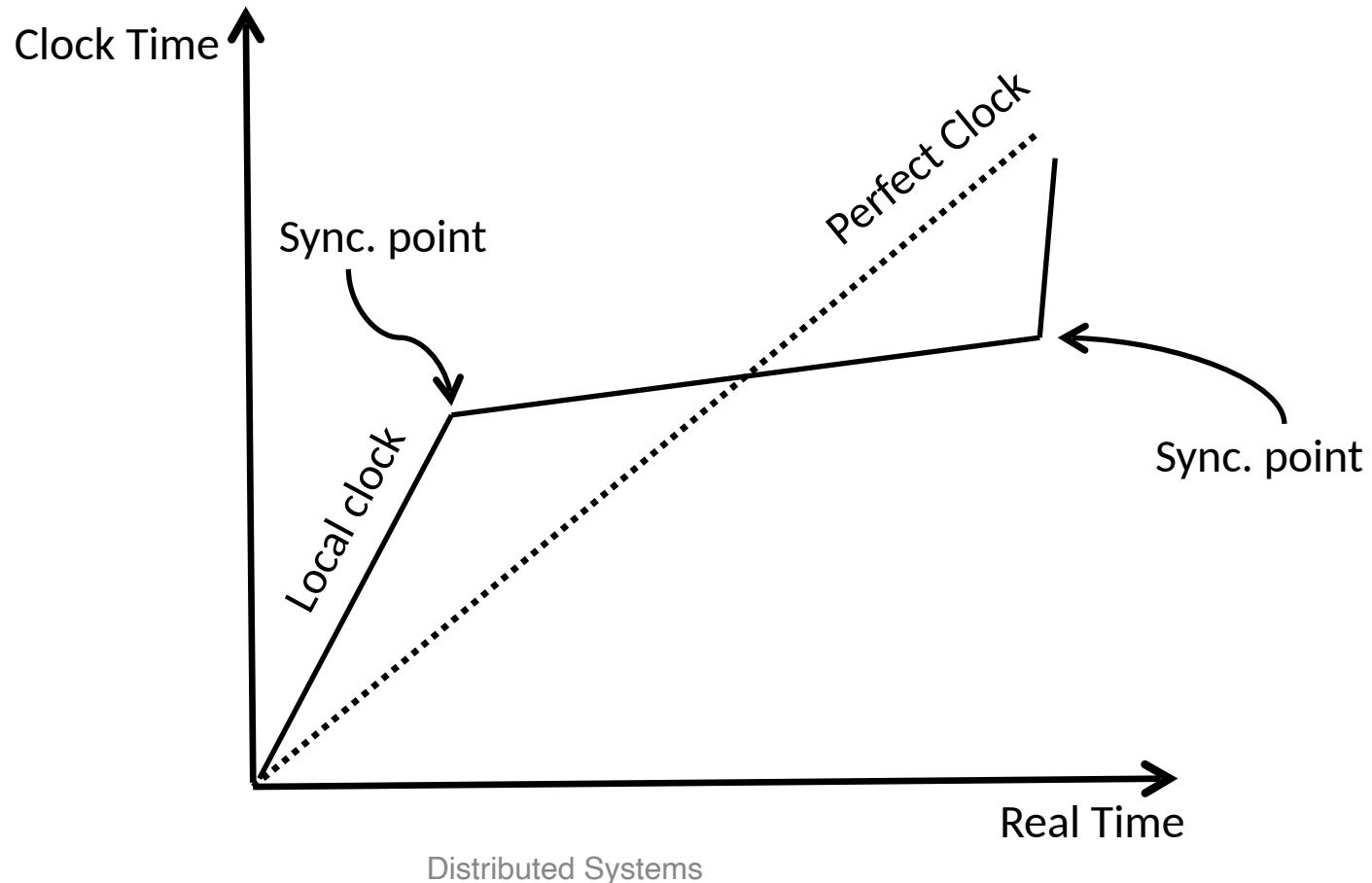
“Catch up” Example: Slow down clock

- Instead, **slow down** the local clock by not updating the full amount at each clock interrupt



“Catch up” Example: Implications

- Imperfect timing of synchronization points may lead to saw tooth behaviour (too slow, then too fast)



LOGICAL CLOCKS

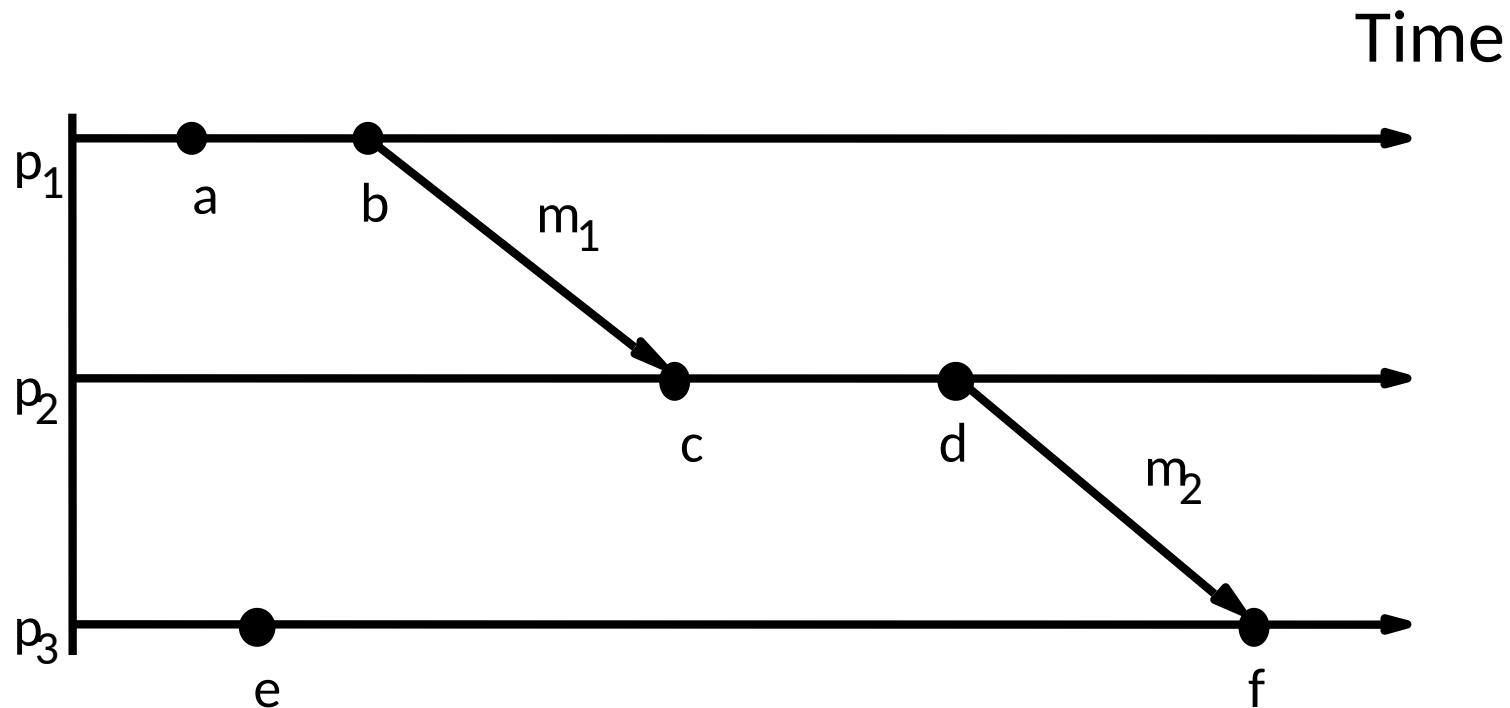
Events and event ordering

- Often sufficient to know the **order of events**
 - An **event** may be an instruction execution, method call, order entry, etc.
 - Events include **message send & receive**
- Within a **single process** order determined by **instruction execution** sequence
- Between **two processes** on same computer order determined using **local physical clock**
- Between **two different computers** order cannot be determined using **local physical clocks**, since those clocks **cannot be synchronized perfectly**

Logical clocks

- Key idea is to **abandon the idea of physical time**
- Only know **order of events**, not *when* they happened (or *how much time between events*)
- Lamport (1978) introduced **logical (virtual) time** and methods to synchronize logical clocks

Events in a distributed system



The happened-before relation

Denoted by “ \rightarrow ”

- Describes a **(causal) ordering of events**
- If **a and b are events in the same process and a occurred before b** then $a \rightarrow b$
- If **a is the event of sending a message m in one process and b is the event of receiving m in another process** then $a \rightarrow b$
- Relation “ \rightarrow ” is **transitive**: If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- If neither $a \rightarrow b$ nor $b \rightarrow a$ then **a and b are concurrent**, denoted by $a \parallel b$
- For any two events a and b , either $a \rightarrow b$, $b \rightarrow a$, or $a \parallel b$

Causality of “ \rightarrow ”-relation

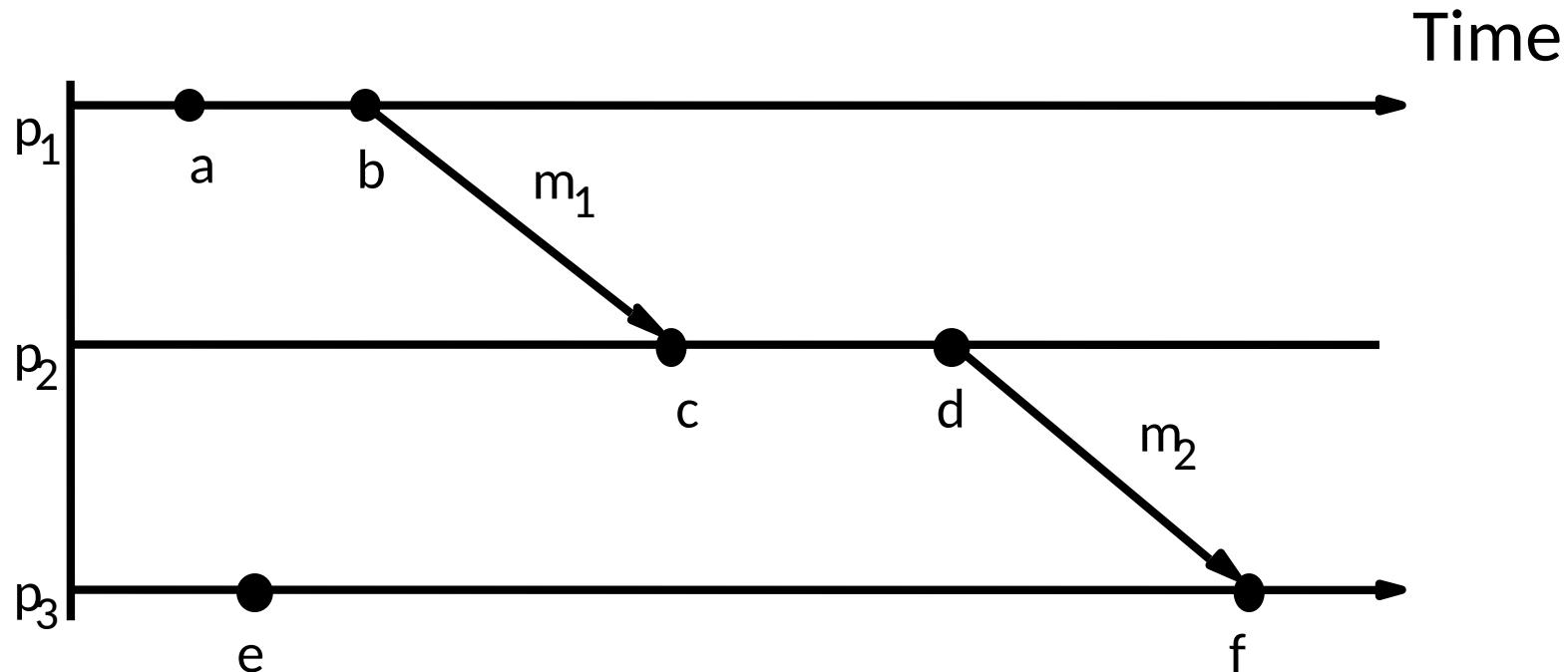
(a.k.a. causality relation)

- Intuitively, past events influence future events
- Influence among causally related events is referred to as **causal effects**
- If $a \rightarrow b$, event a **may** causally effect event b
- Concurrent events **do not causally effect** each other (e.g., neither $a \rightarrow b$ nor $b \rightarrow a$)

Potential causality of “ \rightarrow ”-relation

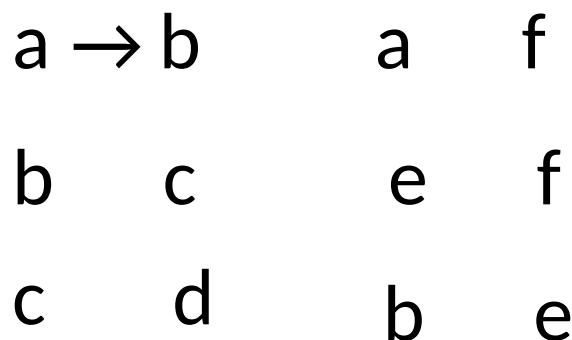
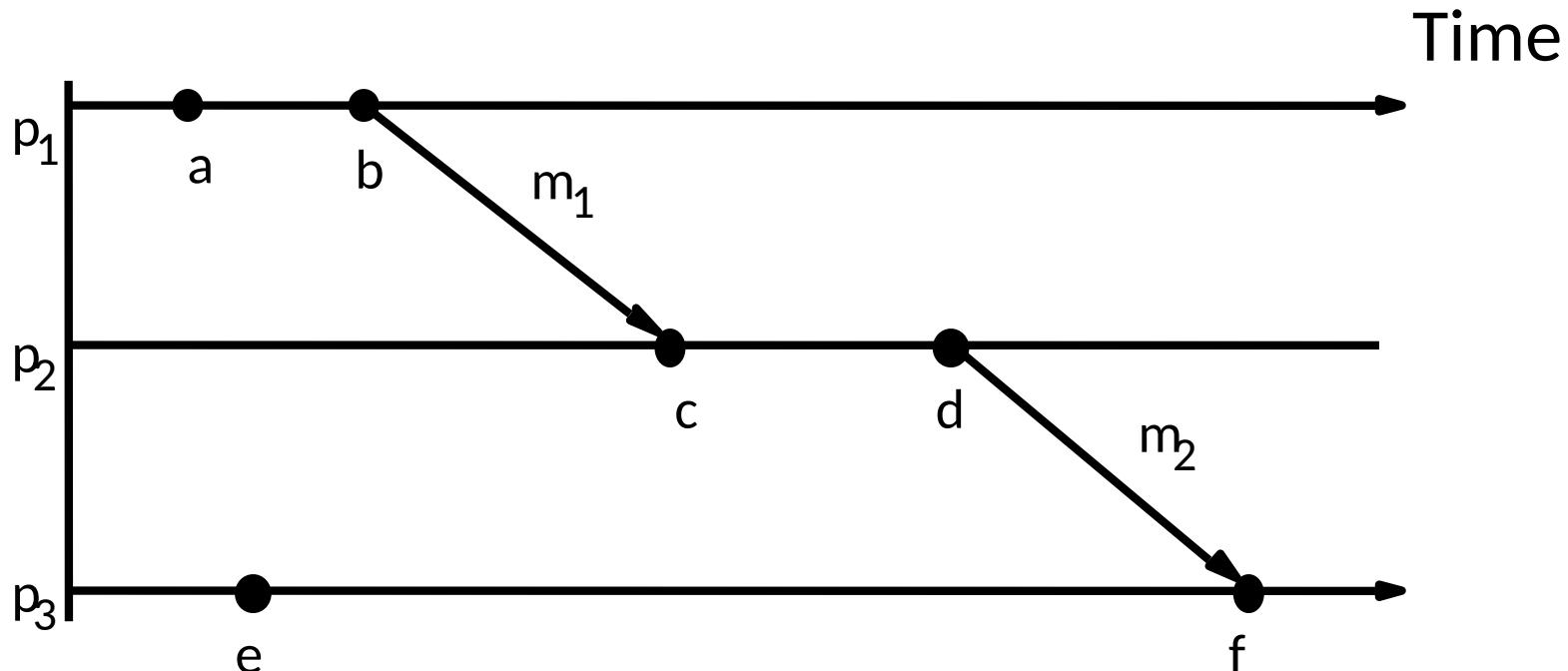
- “ \rightarrow ” captures potential flow of information between events
- Information may have flown in ways other than via message passing (not modeled by “ \rightarrow ”)
- In $a \rightarrow b$, a might or might not have caused b (relation assumes it has, but we don't know for sure)

Happened-before relation

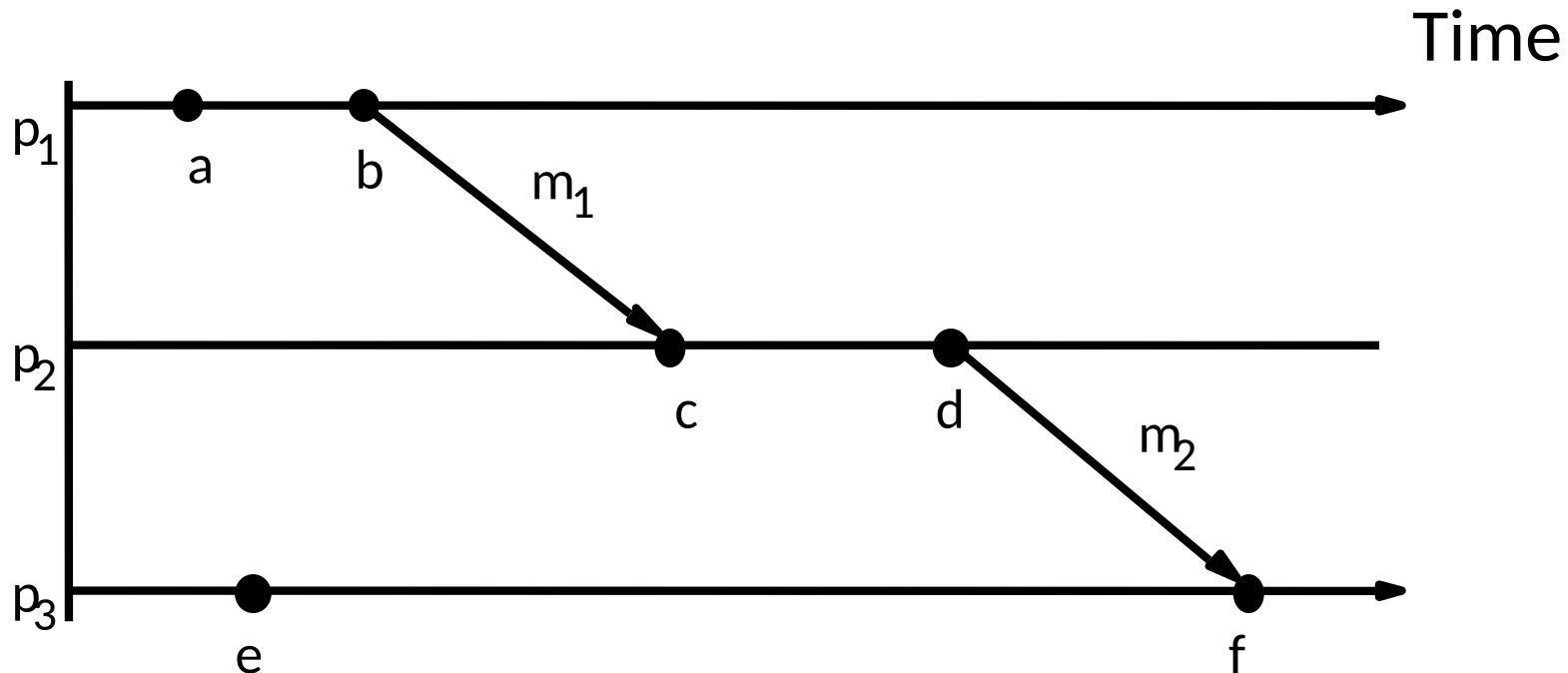


a	b	a	f
b	c	e	f
c	d	b	e

Happened-before relation

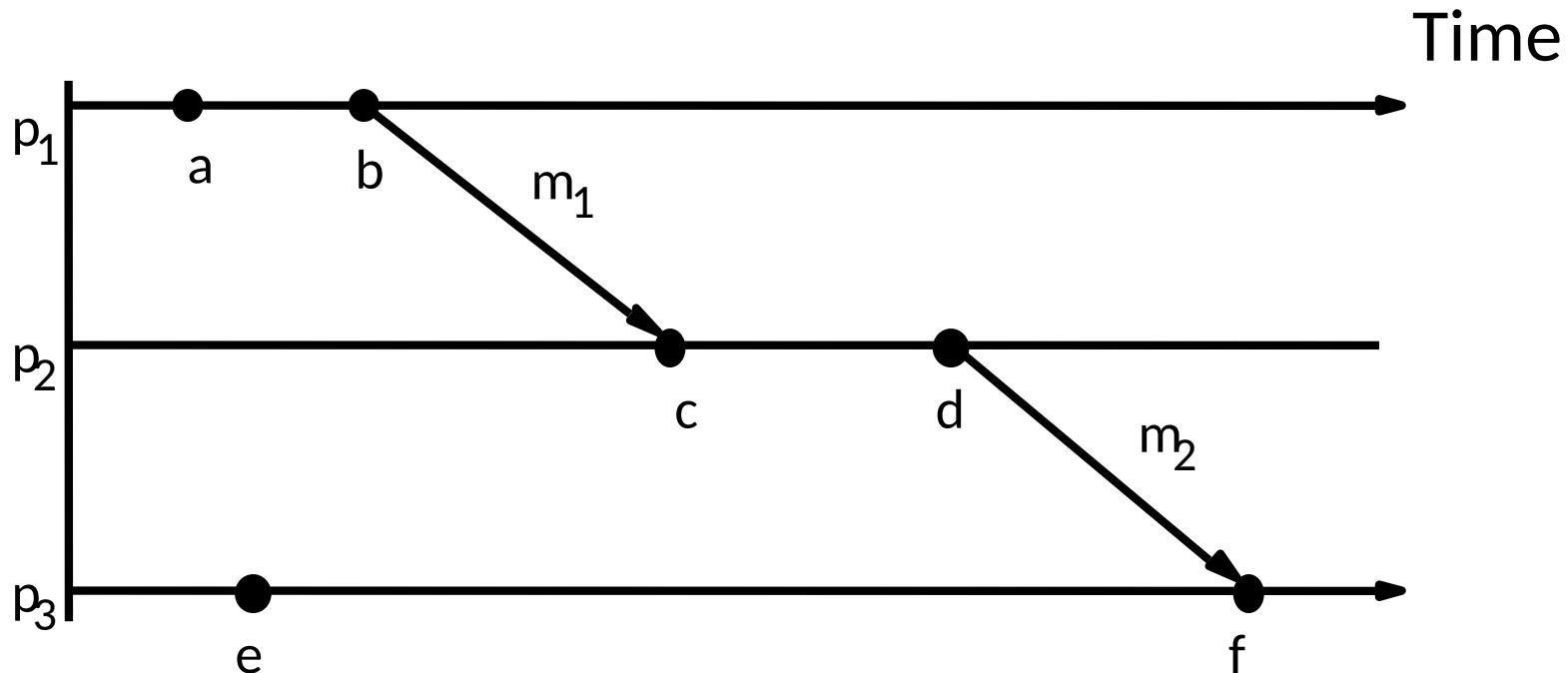


Happened-before relation



$a \rightarrow b$	$a \rightarrow f$
$b \rightarrow c$	$e \rightarrow f$
$c \rightarrow d$	$b \rightarrow e$

Happened-before relation



$a \rightarrow b$

$a \quad f$

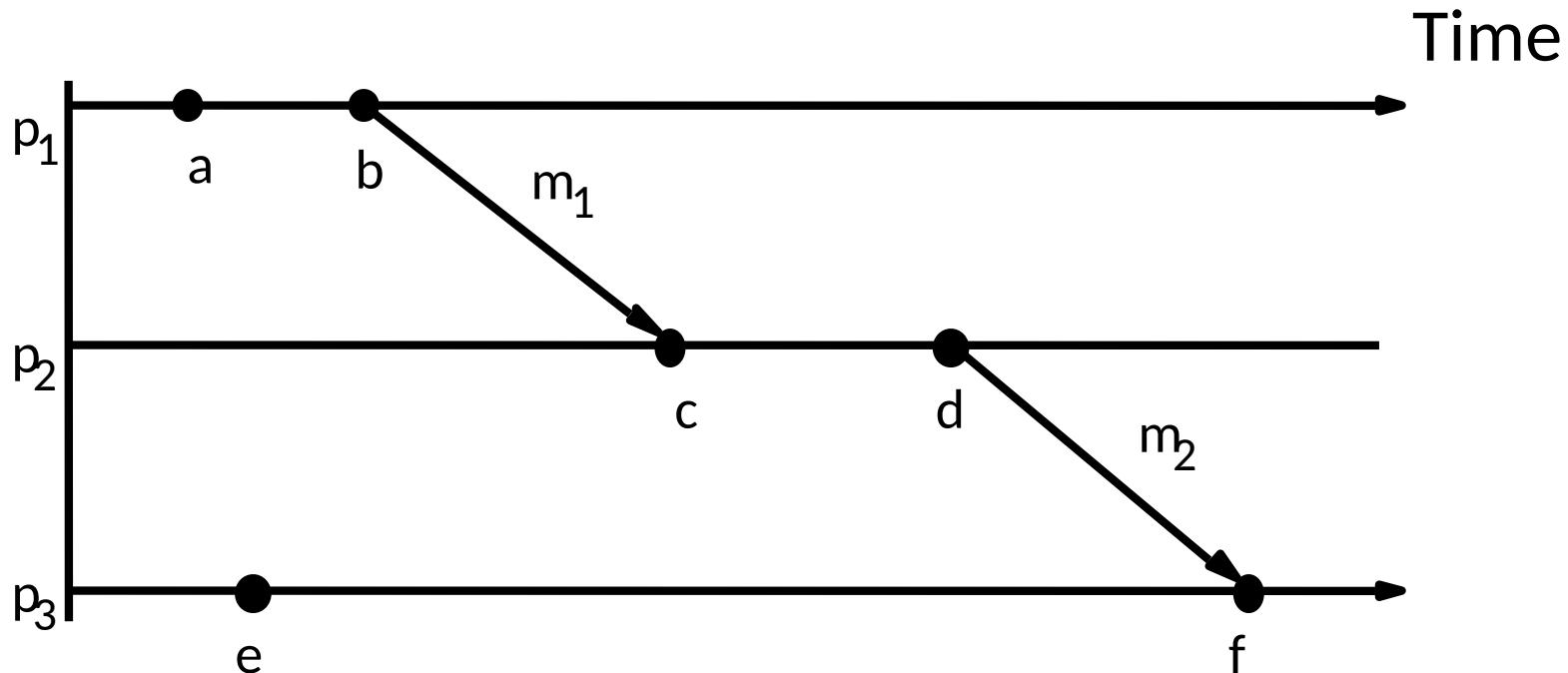
$b \rightarrow c$

$e \quad f$

$c \rightarrow d$

$b \quad e$

Happened-before relation



$a \rightarrow b$

$a \rightarrow f$

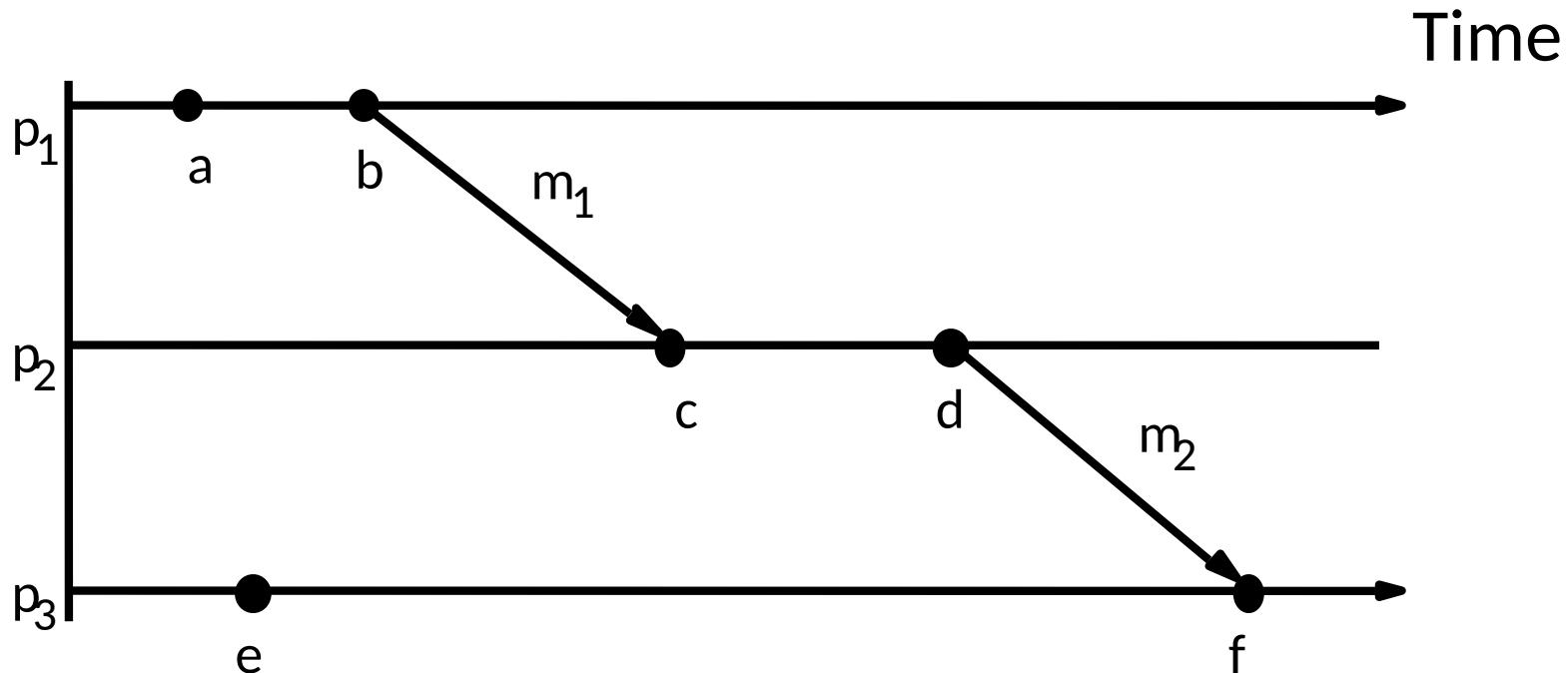
$b \rightarrow c$

$e \quad f$

$c \rightarrow d$

$b \quad e$

Happened-before relation



$a \rightarrow b$

$a \rightarrow f$

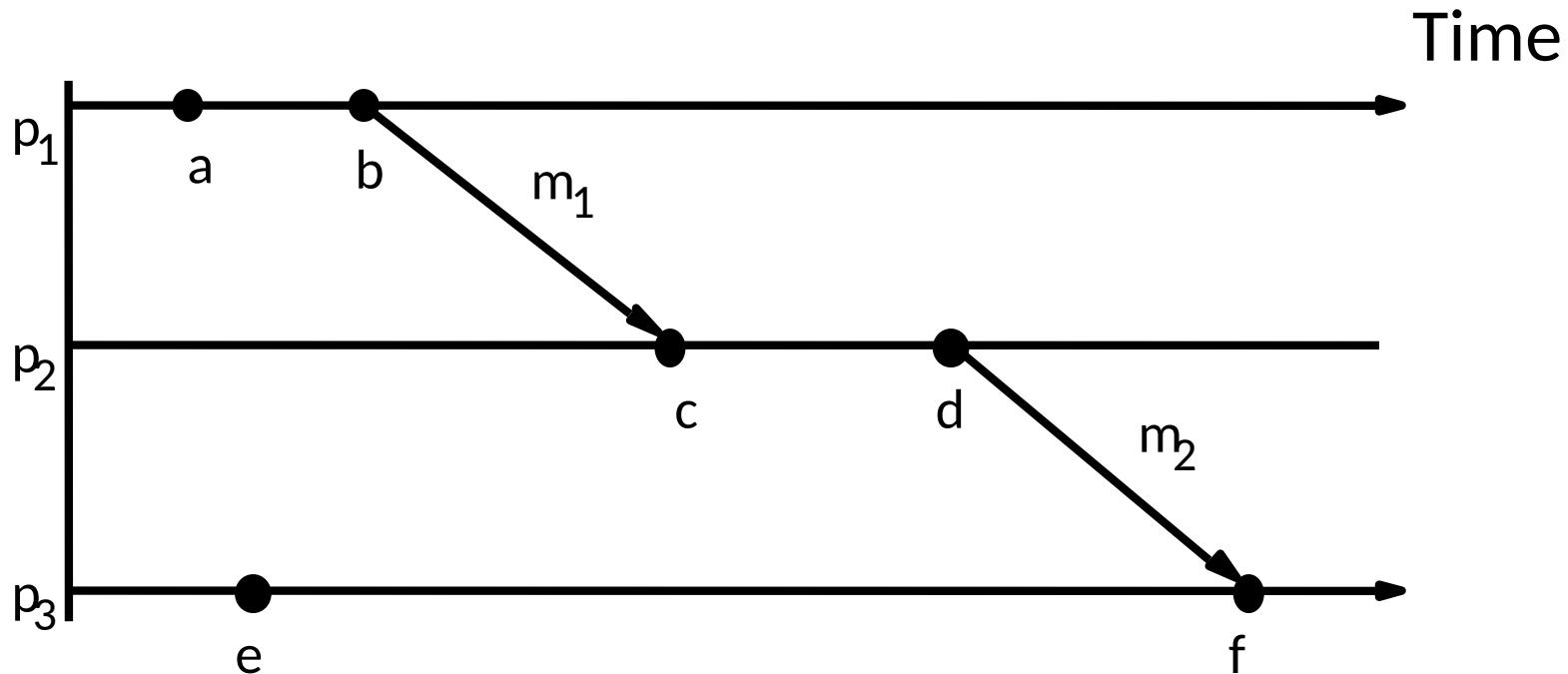
$b \rightarrow c$

$e \rightarrow f$

$c \rightarrow d$

$b \quad e$

Happened-before relation



$a \rightarrow b$

$a \rightarrow f$

$b \rightarrow c$

$e \rightarrow f$

$c \rightarrow d$

$b \parallel e$

Lamport clock (logical clock)

- Implementation of clock that tracks “ \rightarrow ” numerically
- Each process P_i has a **logical clock C_i** (a counter)
- Clock C_i can assign a **value $C_i(a)$** to any event a in process P_i
- Value $C_i(a)$ is the **timestamp of event a** in process P_i
- Timestamps have **no relation to physical time**, which leads to the term **logical clock**
- Logical clocks **assign monotonically increasing timestamps**
- Can be implemented by a **simple integer counter**

Correctness condition

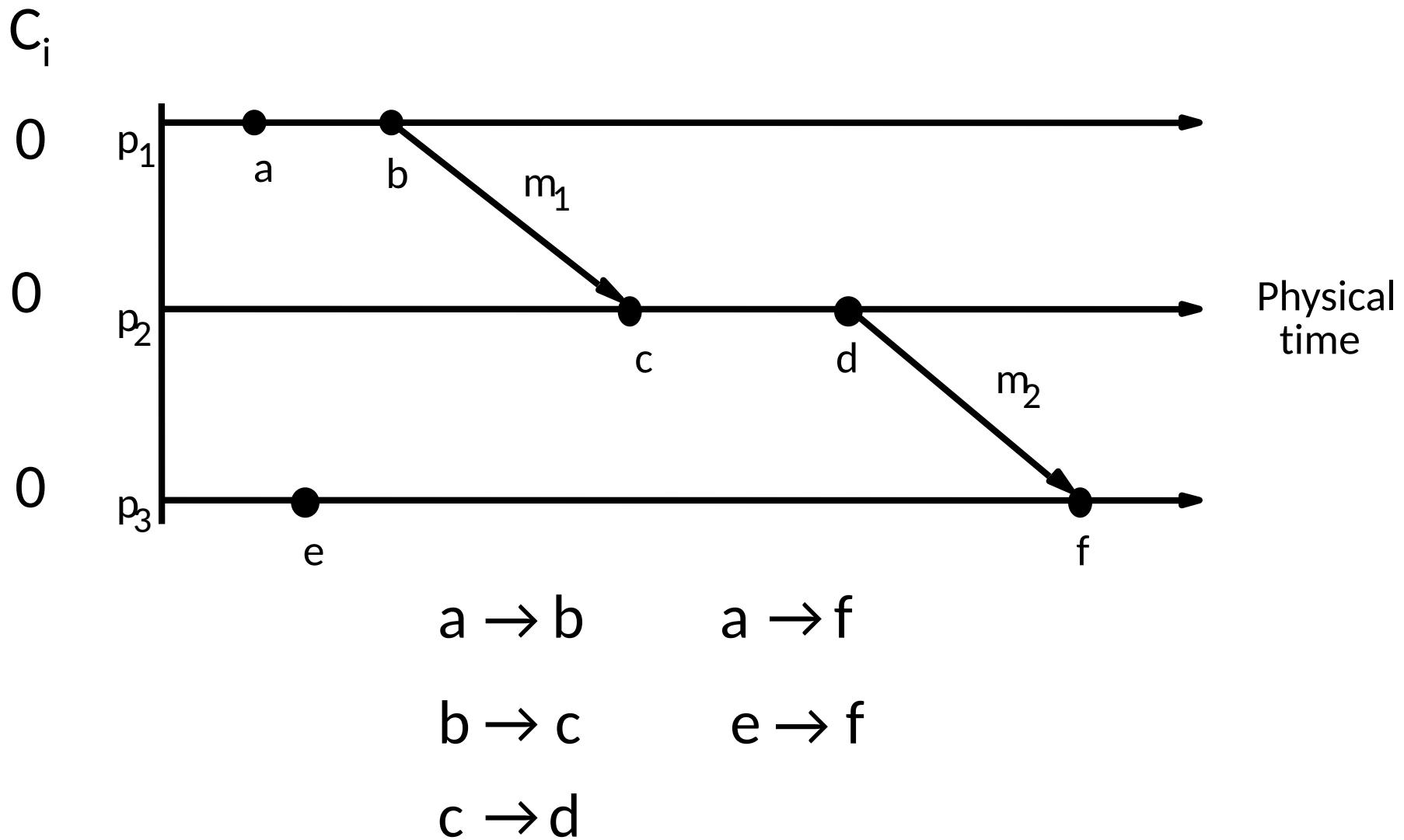
Correctness condition

- Clock condition
 - If $a \rightarrow b$ then $C(a) < C(b)$
 - But not: If $C(a) < C(b)$ then $a \rightarrow b$
- Correctness conditions
 - For any two events a and b in the same process P_i , if $a \rightarrow b$ then $C_i(a) < C_i(b)$
 - If a is the event of sending a message in process P_i and b is the event of receiving that same message in a different process P_j then $C_i(a) < C_j(b)$

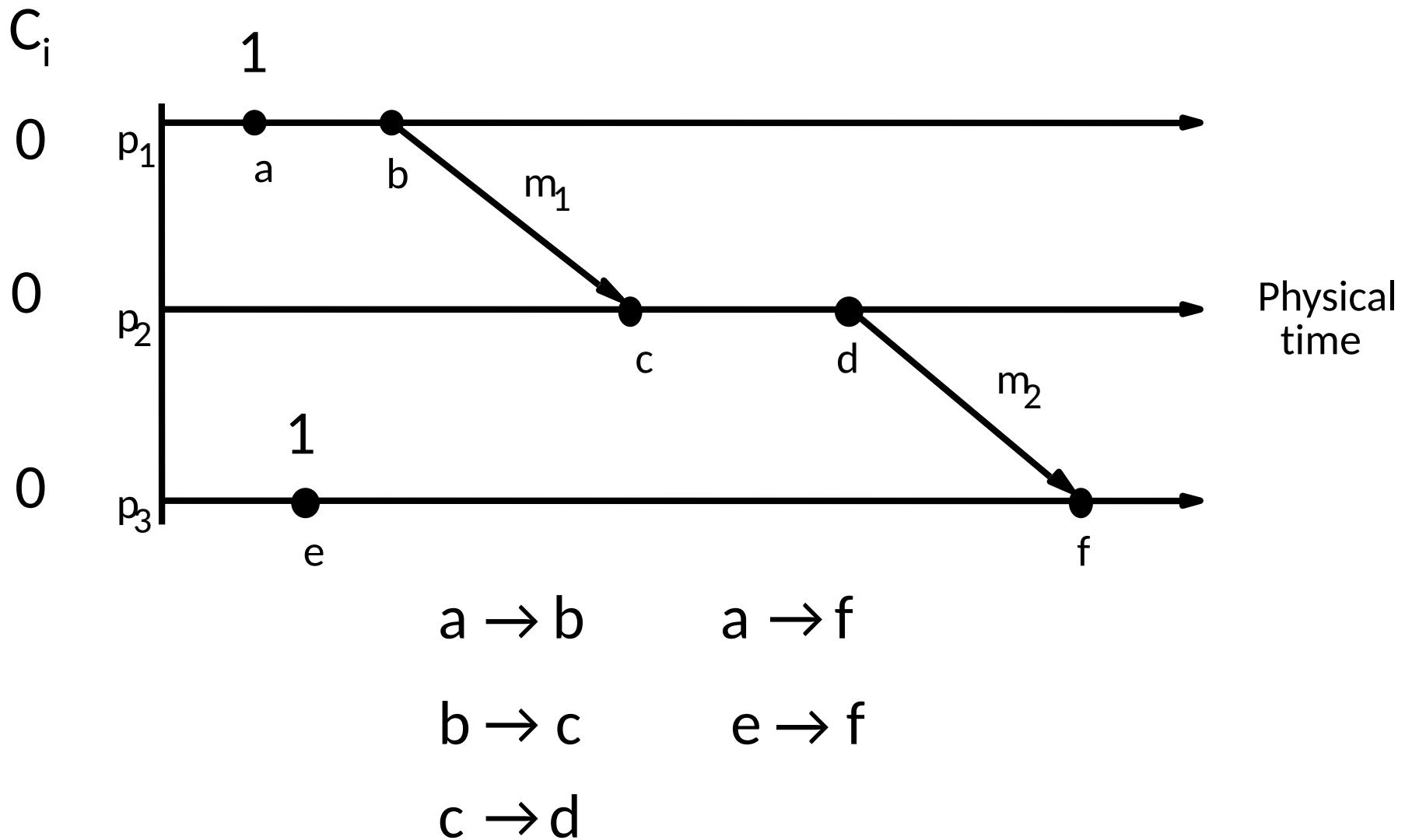
Implementation of logical clocks

- Clock C_i must be **incremented** before an event occurs in process P_i (before event is executed)
 - $C_i = C_i + d$ ($d > 0$, d usually 1)
- If a is the event of **sending a message m** in process P_i then **m is assigned a timestamp**
 - $T_m = C_i(a)$
- When **that same message m is received** by a different process P_k , **C_k is set to a value greater than its present value** (prior to the message receipt) and greater than T_m
 - $C_k = \max\{C_k, T_m\} + d$ ($d > 0$, d usually 1)

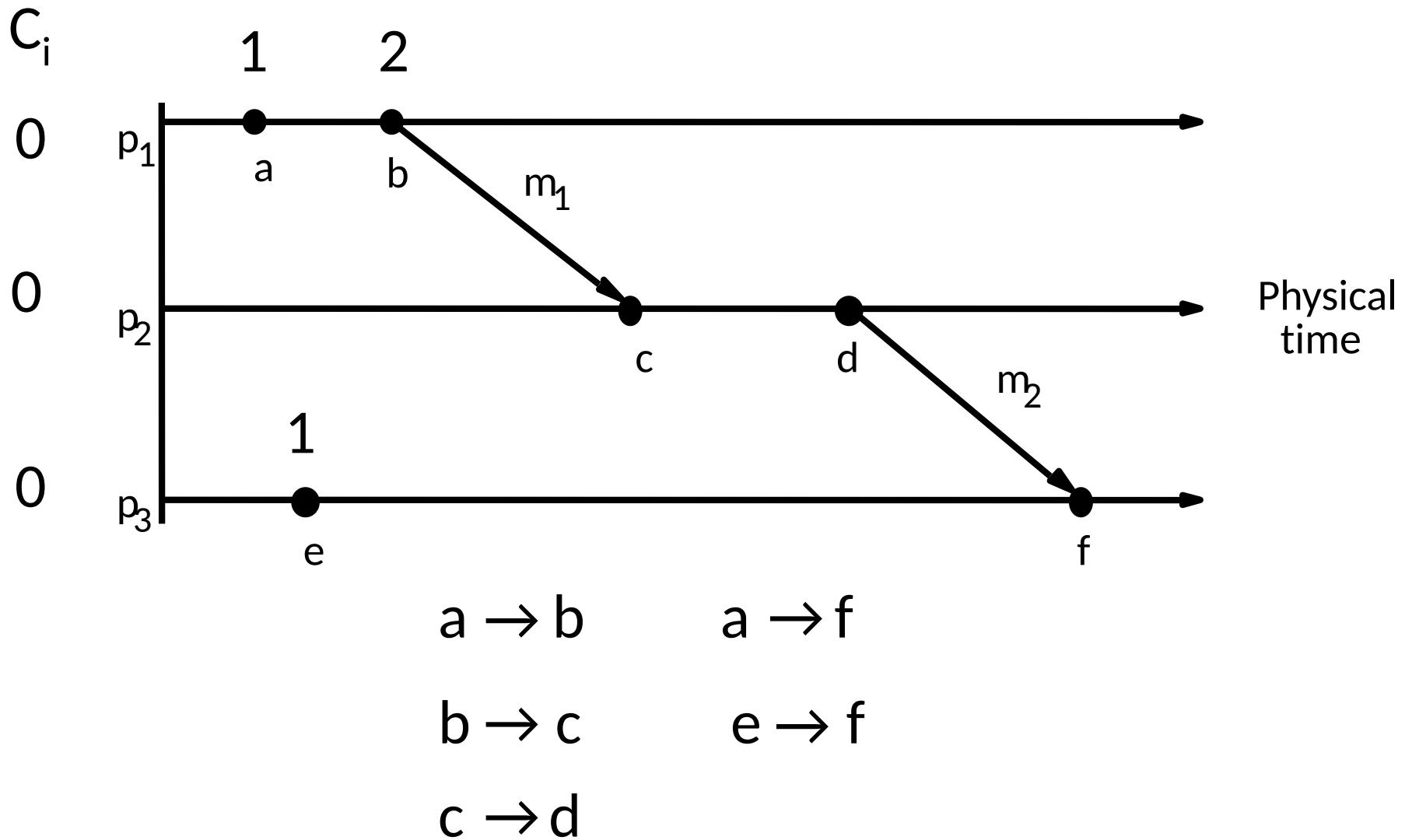
Logical clock example



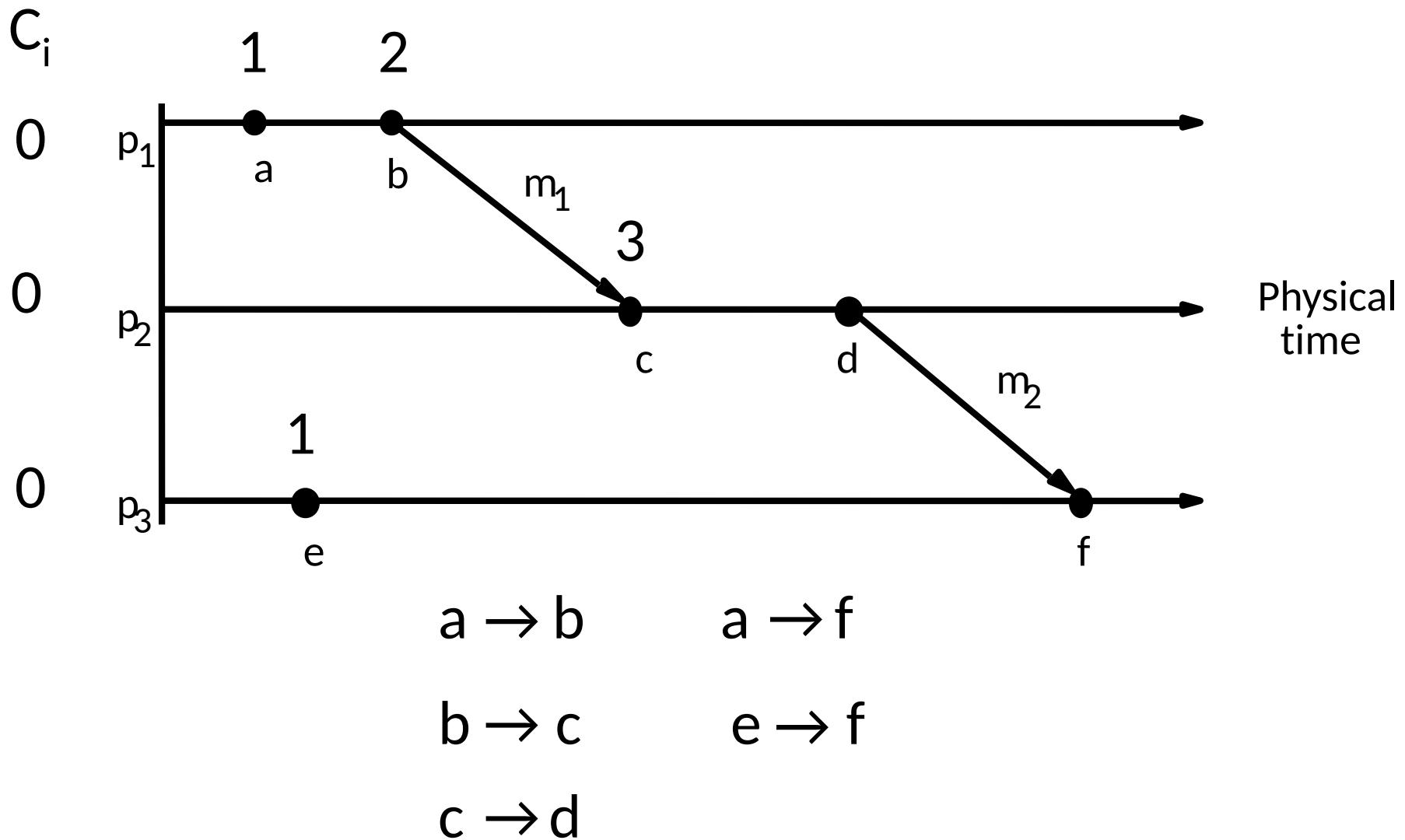
Logical clock example



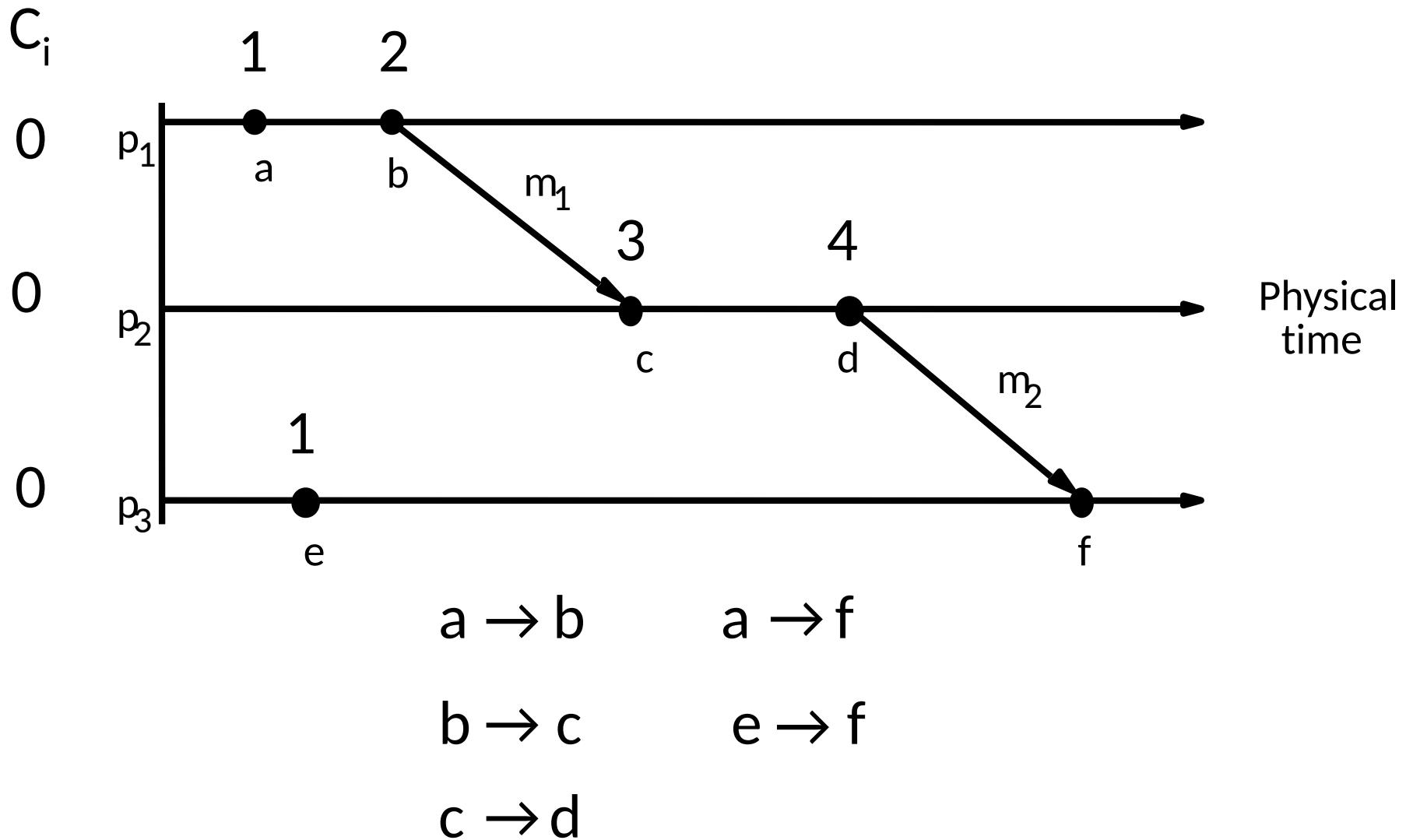
Logical clock example



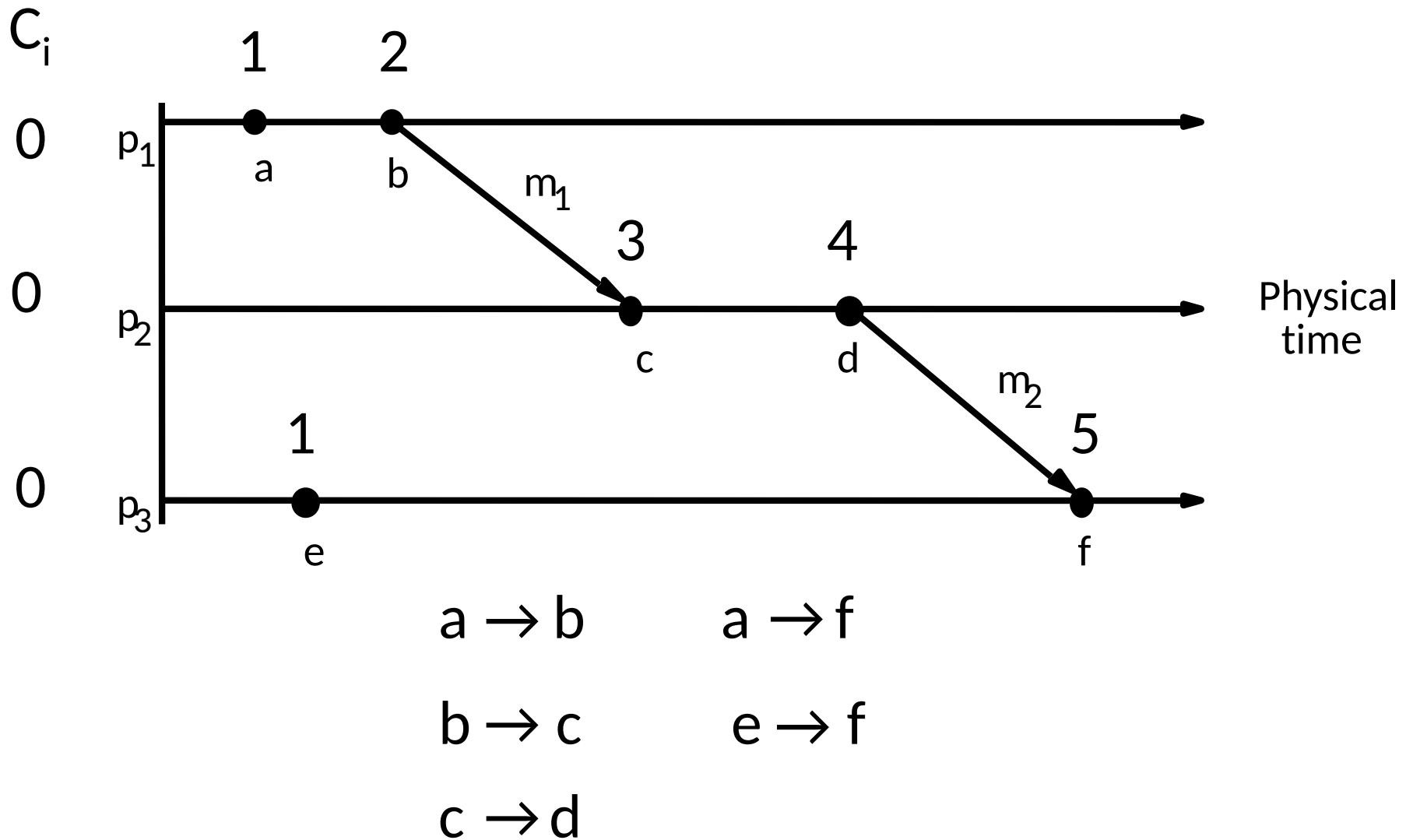
Logical clock example



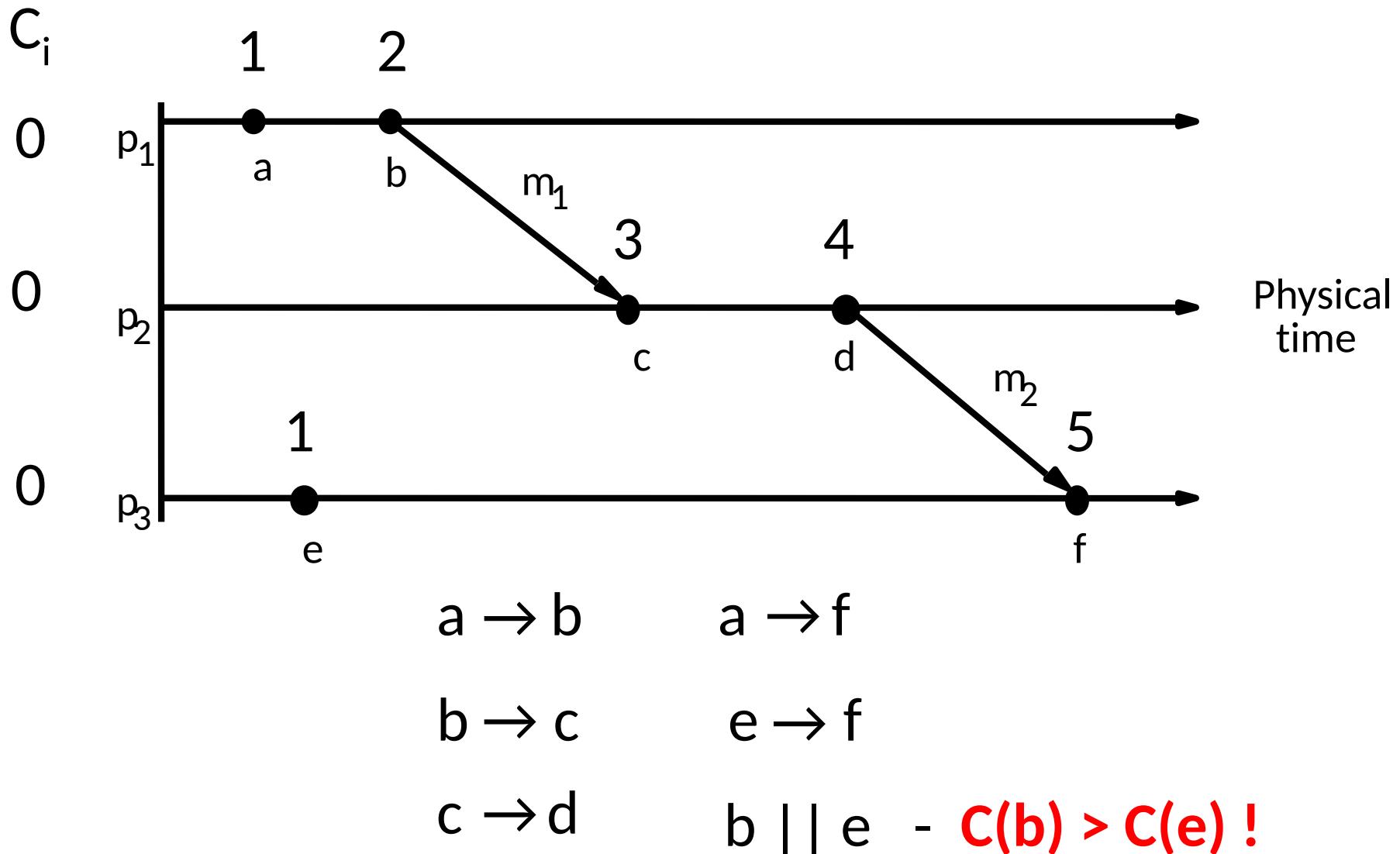
Logical clock example



Logical clock example



Logical clock example



Induced total order

- $C_1(a) = 1$ and $C_3(e) = 1$ can't be ordered according to happened-before relation!
- Happened-before is a **unique partial order** of events
- Induce a non-unique total order as follows
 - Use logical time stamps to order events
 - Break ties by using an arbitrary total ordering of processes, e.g., $P_1 < P_2$ (process identifiers)

Total order for events

- Let a be an event in P_i and b an event in P_j
then $a \Rightarrow b$ if and only if either
 - (i) $C_i(a) < C_j(b)$ or
 - (ii) $C_i(a) = C_j(b)$ and $P_i < P_j$
- Results in total order of all events in system

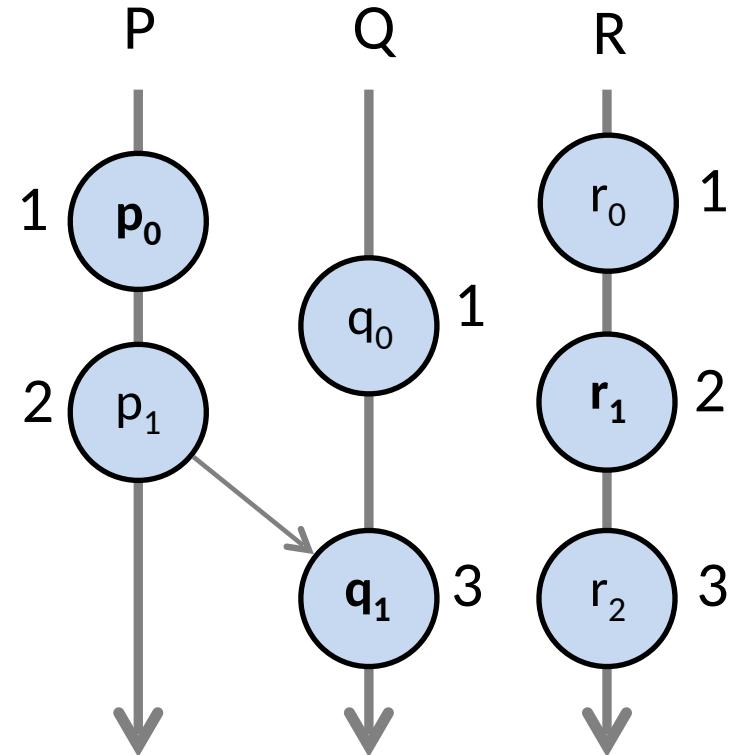
Application of logical clocks

(Distributed mutual exclusion)

- System with n processes, single exclusive-use resource
- We require:
 1. Resource is used exclusively by a single process at a time
 2. Requests to the resource must be granted in the order they are made
 3. Every request must eventually be granted
- Use system of logical clocks and induced total order of events
- Order request, release operations for resource by total order **See lecture on coordination for details.**

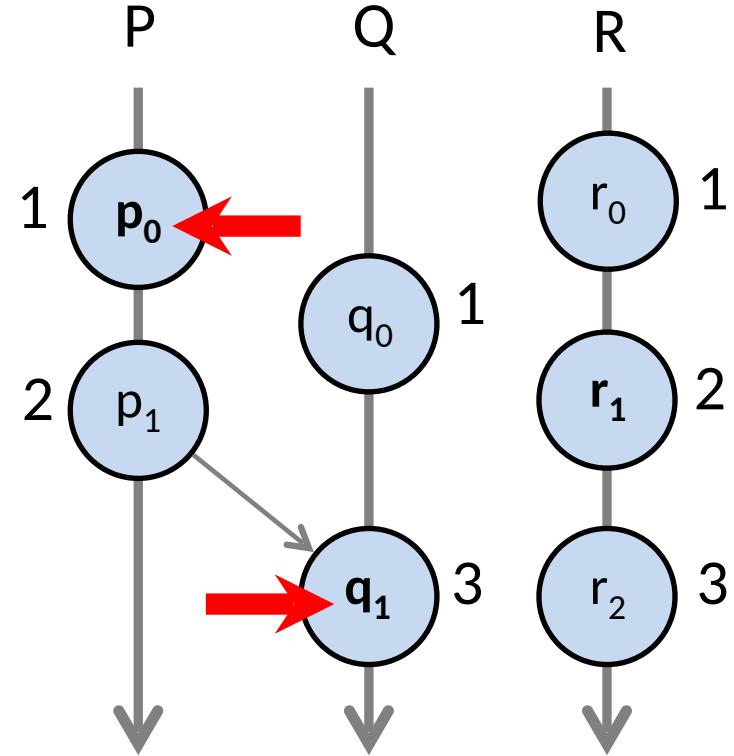
Limitation of Logical Clocks

- If $C(a) < C(b)$ then a may or may not happen-before b
- Example illustrating this limitation
 - $C(p_0) < C(q_1)$ and $p_0 \rightarrow q_1$ is true
 - $C(p_0) < C(r_1)$ but $p_0 \rightarrow r_1$ is false
- **One cannot determine whether two events are causally related from timestamps alone**



Limitation of Logical Clocks

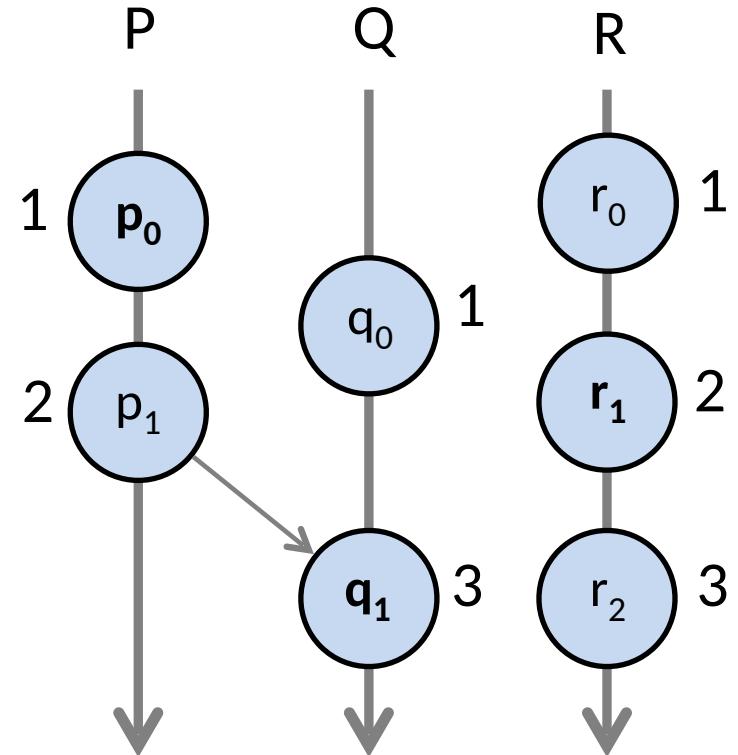
- If $C(a) < C(b)$ then
 a may or may not happen-before b
- Example illustrating this limitation
 - $C(p_0) < C(q_1)$ and $p_0 \rightarrow q_1$ is true
 - $C(p_0) < C(r_1)$ but $p_0 \rightarrow r_1$ is false



- **One cannot determine whether two events are causally related from timestamps alone**

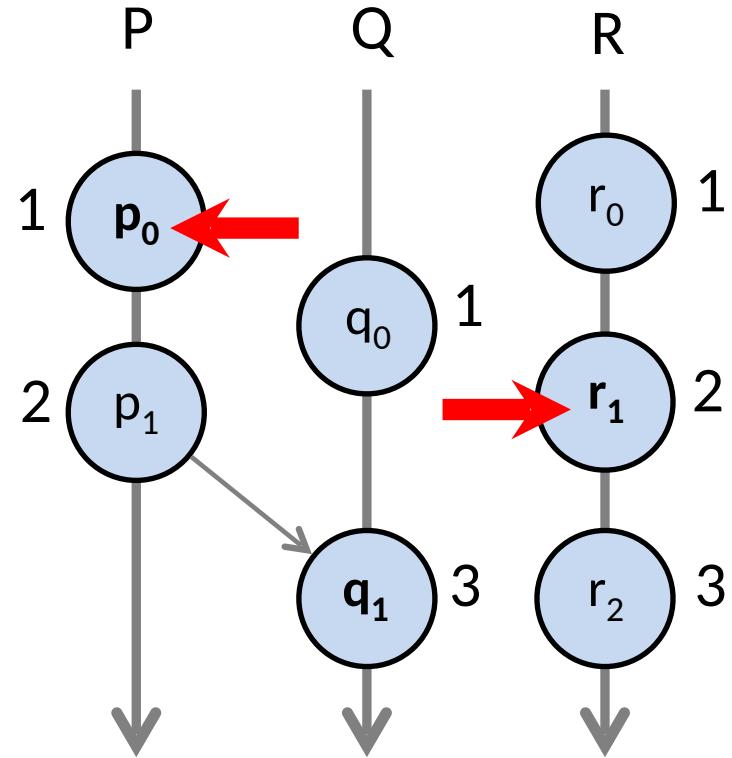
Limitation of Logical Clocks

- If $C(a) < C(b)$ then a may or may not happen-before b
- Example illustrating this limitation
 - $C(p_0) < C(q_1)$ and $p_0 \rightarrow q_1$ is true
 - $C(p_0) < C(r_1)$ but $p_0 \rightarrow r_1$ is false
- **One cannot determine whether two events are causally related from timestamps alone**



Limitation of Logical Clocks

- If $C(a) < C(b)$ then
 a may or may not happen-before b
- Example illustrating this limitation
 - $C(p_0) < C(q_1)$ and $p_0 \rightarrow q_1$ is true
 - $C(p_0) < C(r_1)$ but $p_0 \rightarrow r_1$ is false



- One cannot determine whether two events are causally related from timestamps alone

VECTOR CLOCKS

Vector clocks I

- System with n processes
- Each process P_i has a **clock C_i** , which is an **integer vector of length n** :

$$C_i = (C_i[1], C_i[2], \dots, C_i[n])$$

- $C_i(a)$ is the **timestamp** (clock value) of **event a** at process P_i (a vector)
- $C_i[i]$, entry i of C_i , is P_i 's **logical time**
- $C_i[i]$ represents the number of events that process P_i has timestamped

$$P_i : (C_i[1], \dots, C_i[i], \dots, C_i[n])$$

Vector clocks II

- $C_i[k]$, entry k of C_i (where $k \neq i$), is P_i 's “guess” of the logical time at P_k
- $C_i[k]$ is the number of events that have occurred at P_k that P_i has potentially been affected by

$$P_i : (C_i[1], \dots, C_i[k], \dots, C_i[n])$$

Implementation of vector clocks

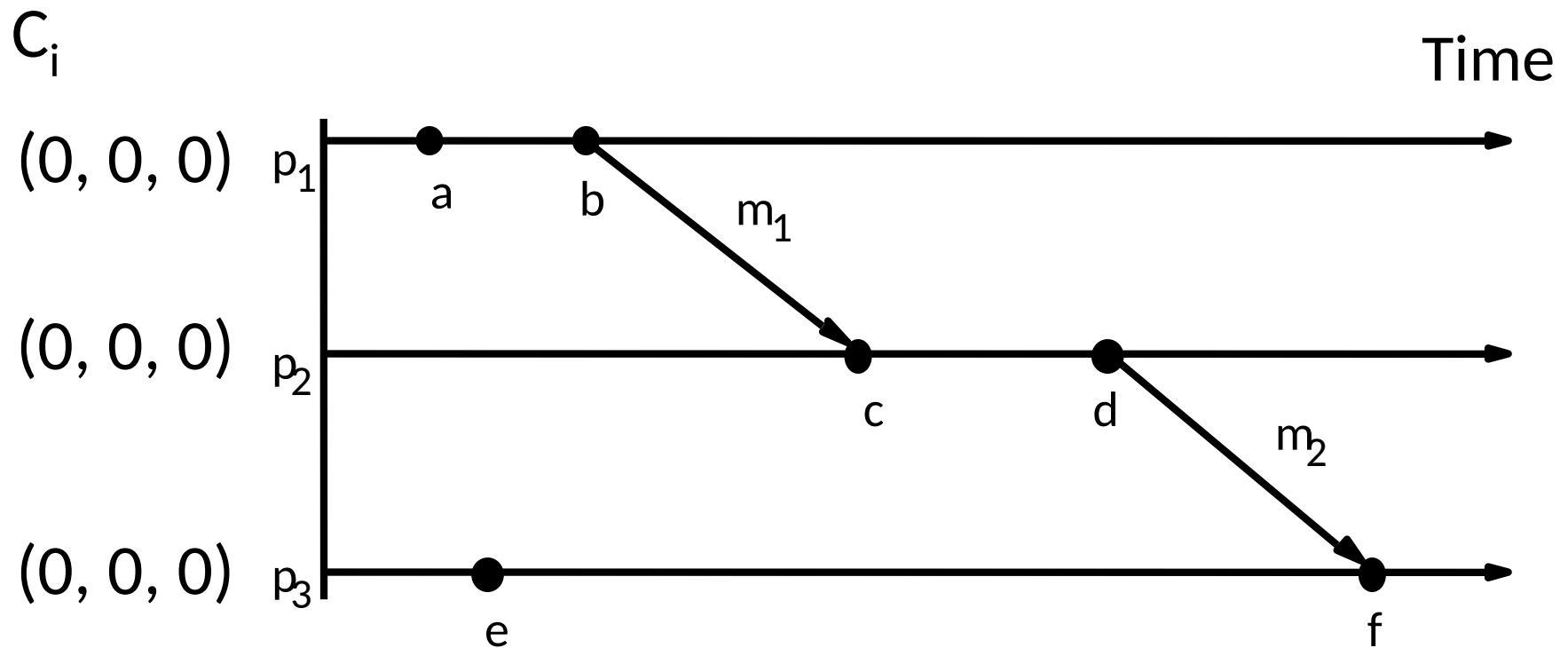
- Clock C_i is incremented before an event occurs in process P_i

$$C_i[i] = C_i[i] + d \quad (d > 0, d \text{ usually } 1)$$

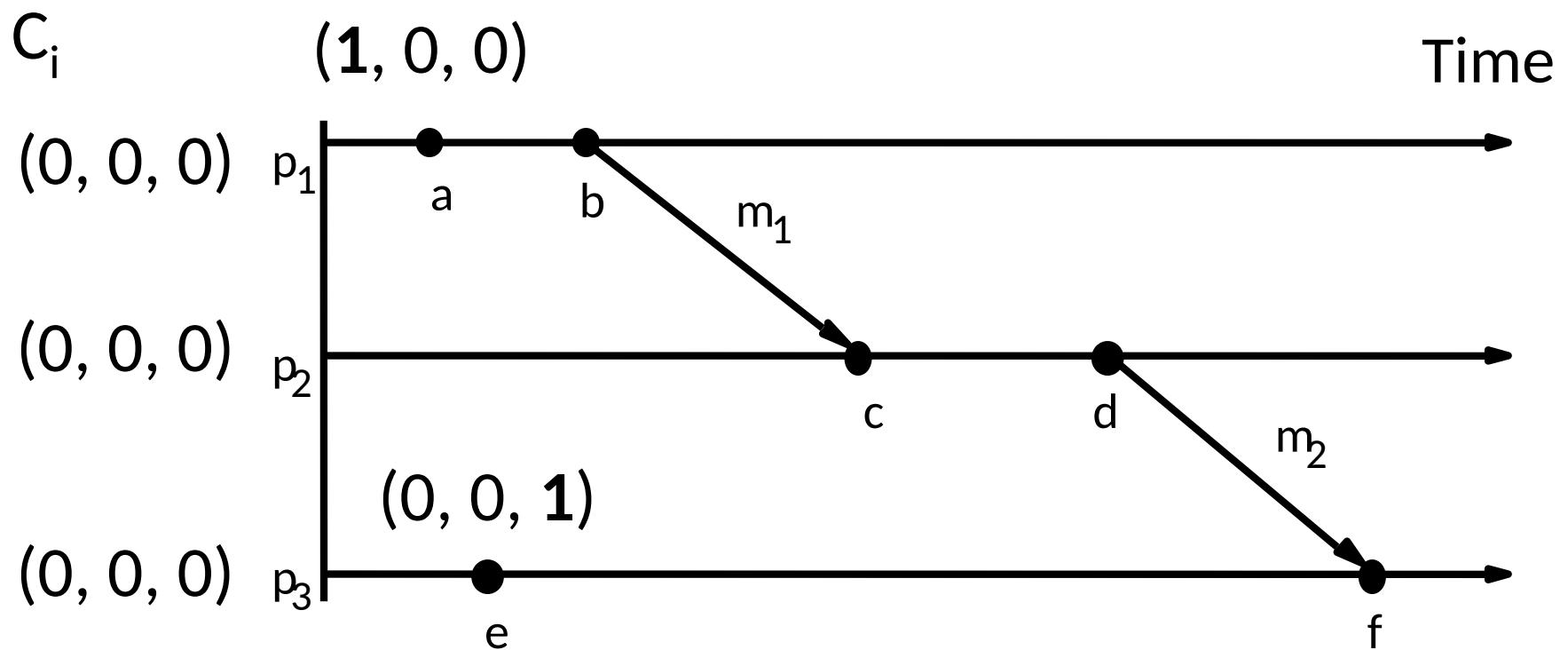
- If event a is the event of sending a message m in process P_i , then message m is assigned a vector timestamp $T_m = C_i(a)$
- When that same message m is received by a different process P_k , C_k is updated as follows:

$$\text{For all } j, C_k[j] = \max\{ C_k[j], T_m[j] \}$$

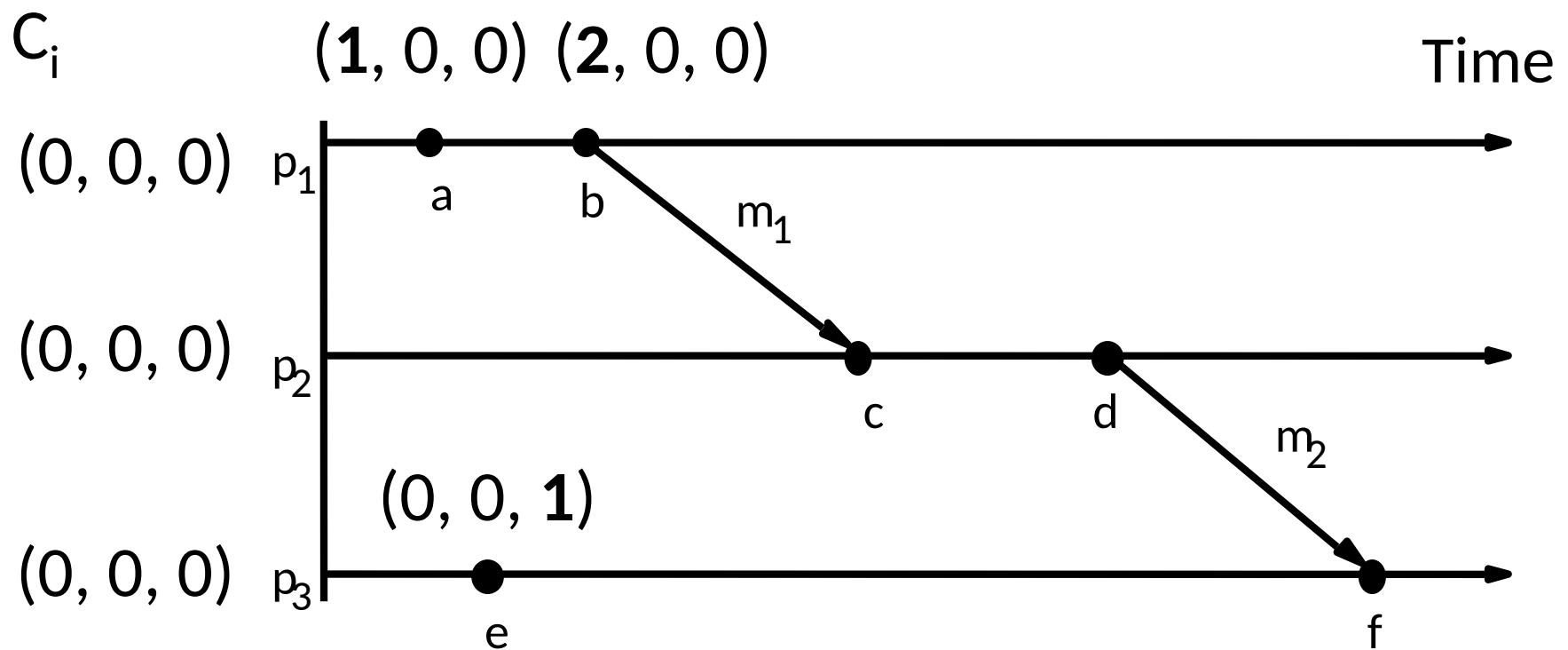
Vector clock example



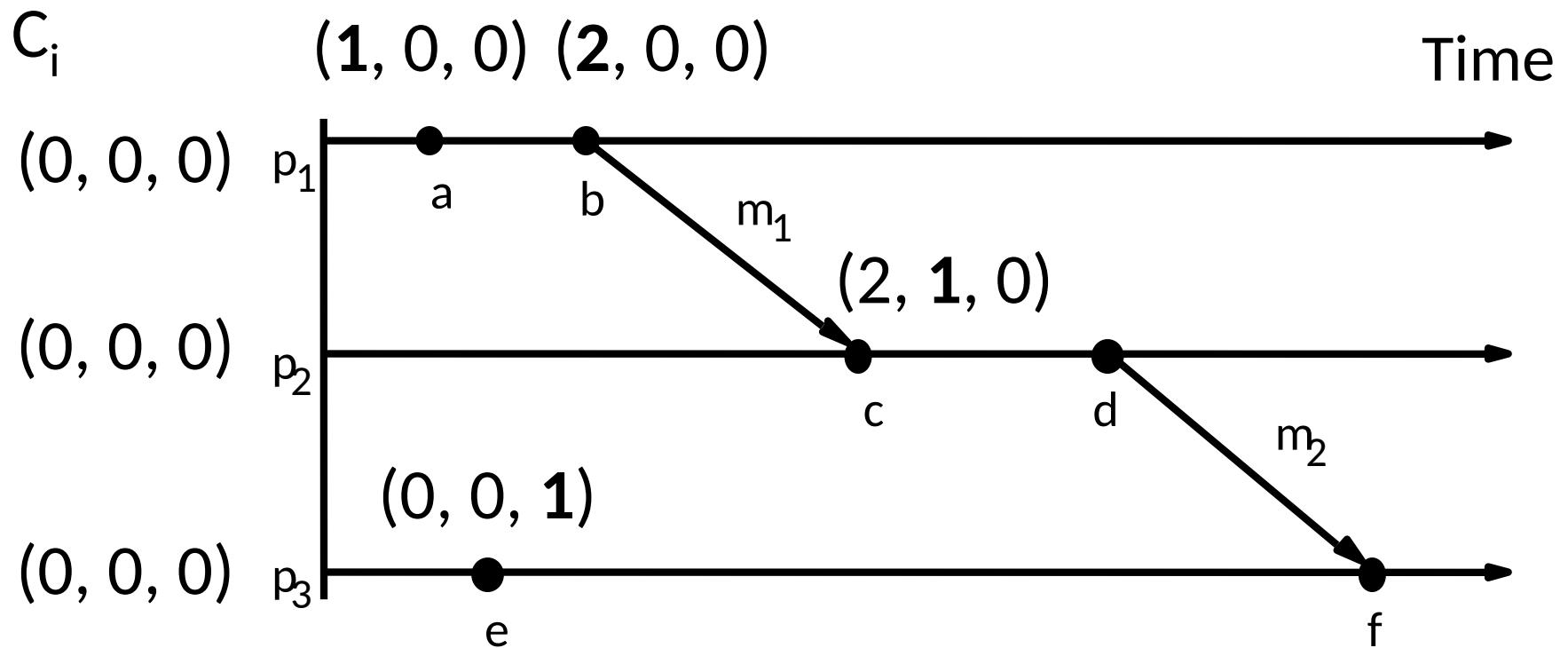
Vector clock example



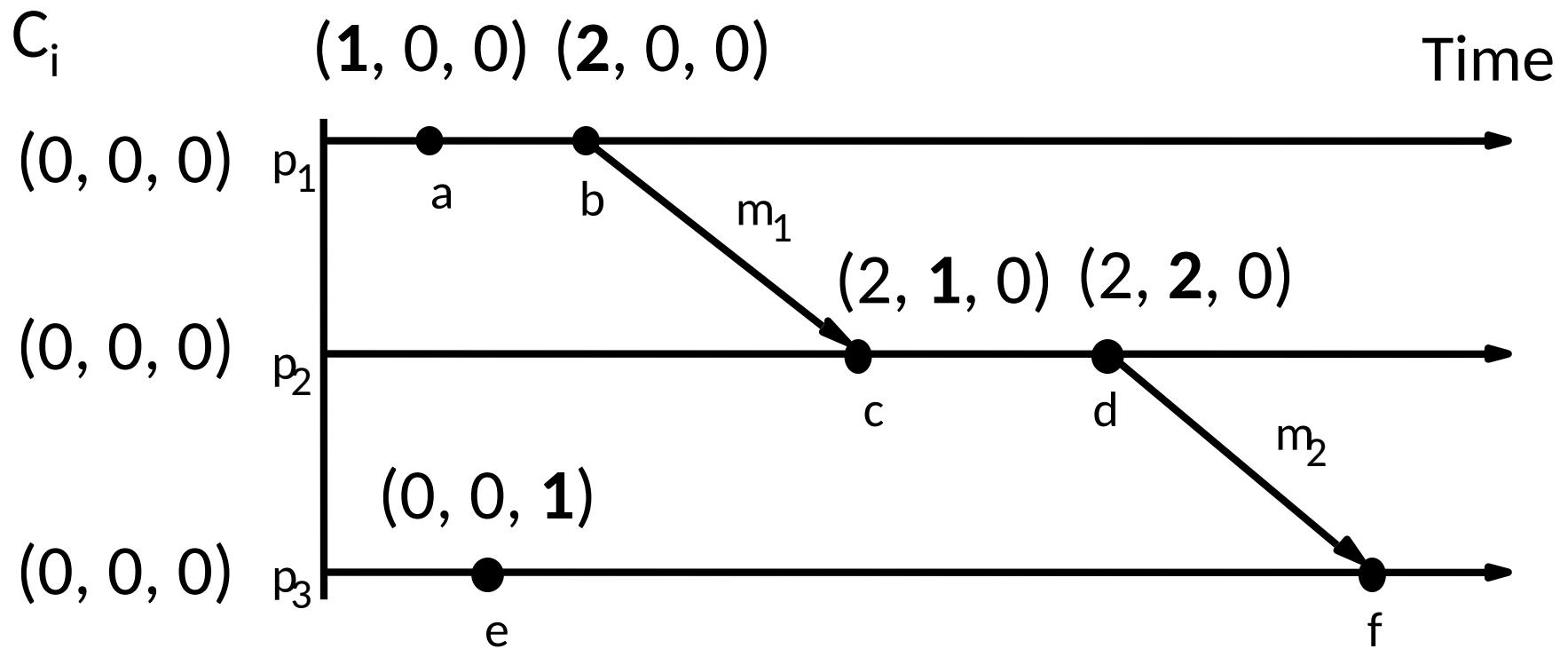
Vector clock example



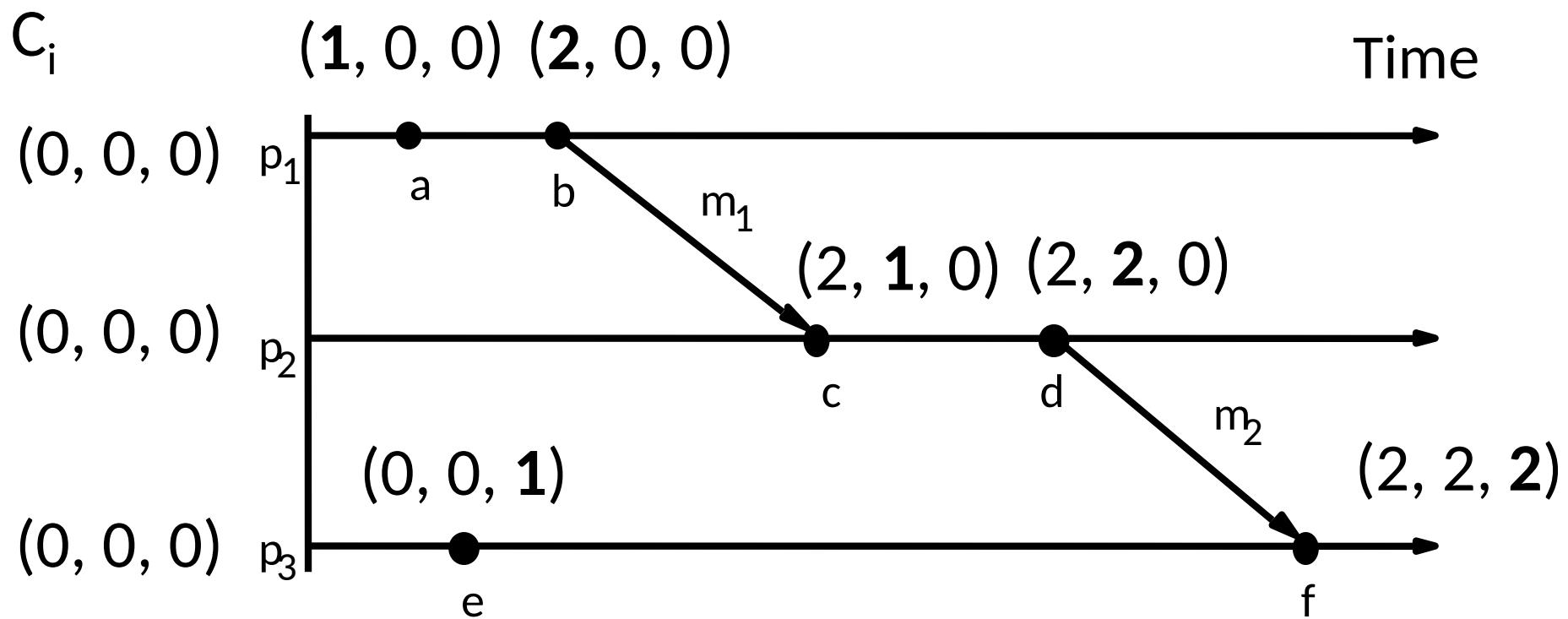
Vector clock example



Vector clock example

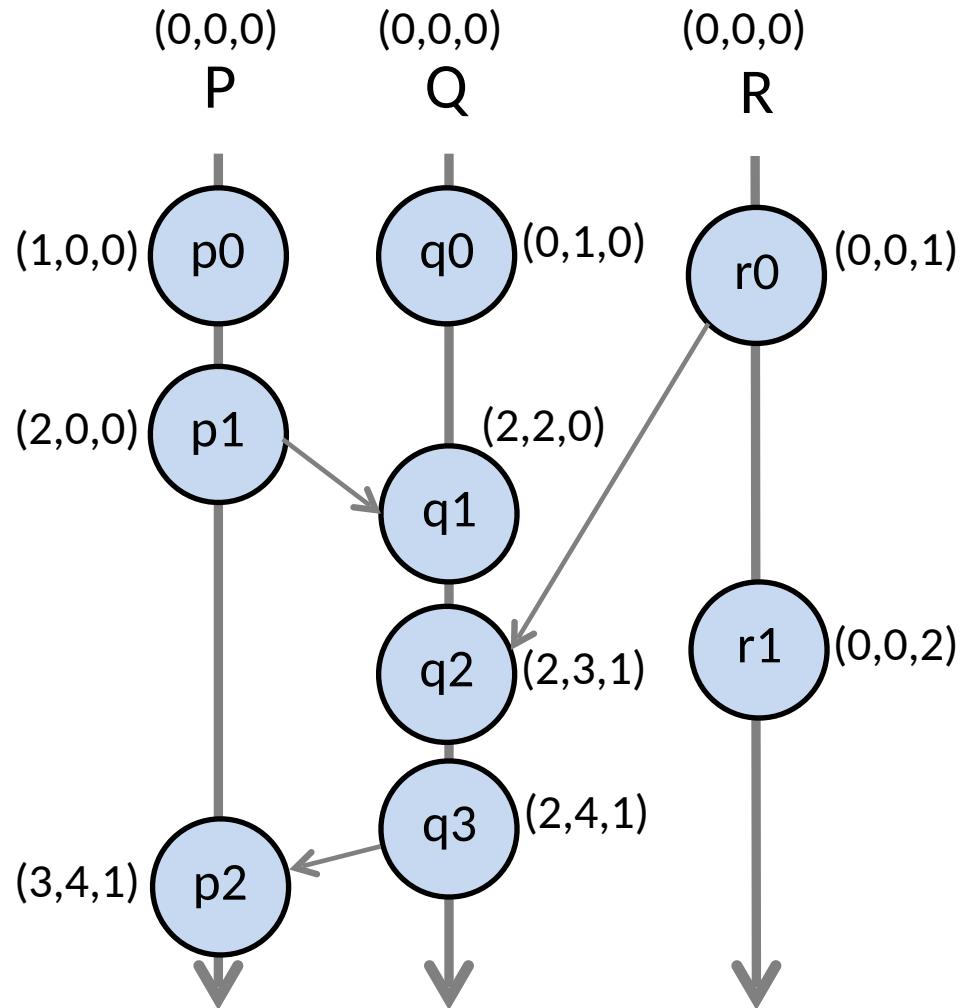


Vector clock example



Example

- Events p_0 , p_1 , q_0 , and r_0 updated by first implementation rule
- Received q_1 updated by both implementation rules
- Received p_2 tells P about Q and R
- R's clock is old, but better than nothing!



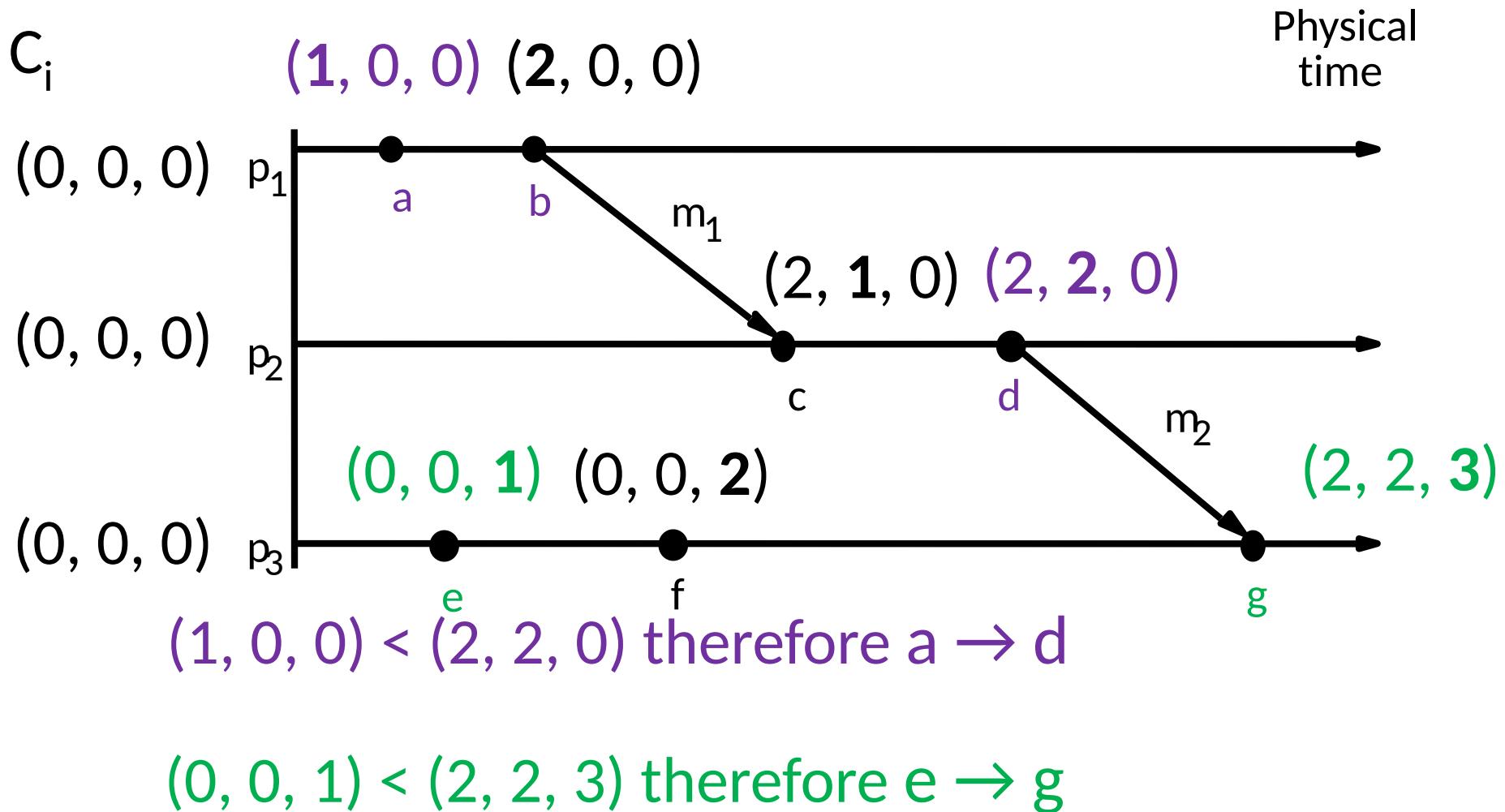
Relations for comparing vector clocks

- C_a, C_b two vector timestamps
- $C_a = C_b \quad \text{iff for all } i: \quad C_a[i] = C_b[i]$
- $C_a \leq C_b \quad \text{iff for all } i: \quad C_a[i] \leq C_b[i]$
- $C_a < C_b \quad \text{iff } C_a \leq C_b, \exists i: C_a[i] < C_b[i]$
- $C_a || C_b \quad \text{iff } \neg(C_a[i] < C_b[i]) \text{ and } \neg(C_b[i] < C_a[i])$

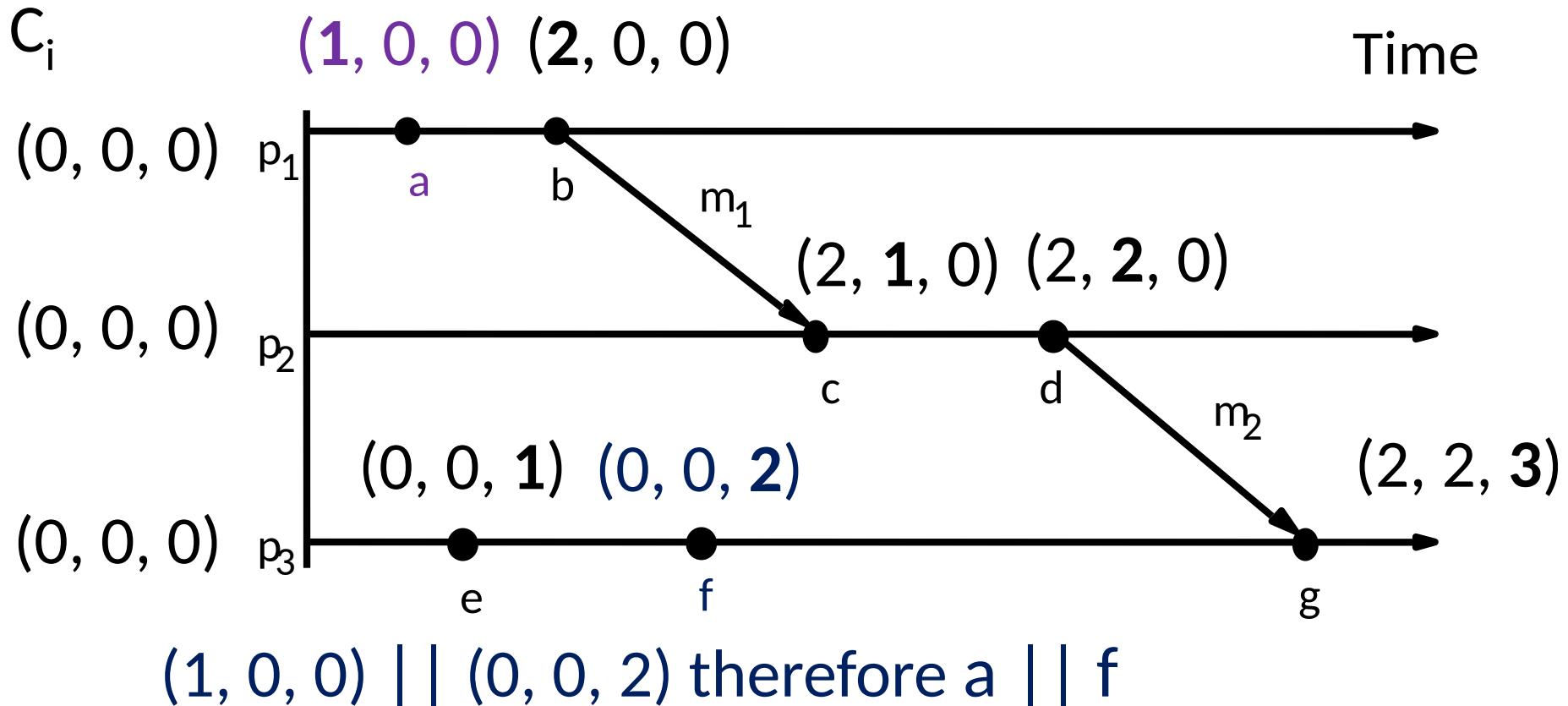
Example

- $(1 \ 1 \ 2 \ 3) = (1 \ 1 \ 2 \ 3)$
- $(1 \ 1 \ 2 \ 3) \leq (1 \ 1 \ 2 \ 4)$ and $(1 \ 1 \ 2 \ 3) \leq (1 \ 1 \ 2 \ 3)$
- $(1 \ 1 \ 2 \ 3) < (1 \ 1 \ 2 \ 4)$
- $(1 \ 1 \ 3 \ 3) \mid\mid (1 \ 1 \ 2 \ 4)$

Vector clock example



Vector clock example



With Lamport: $1 < 2$, but not $a \rightarrow f$?

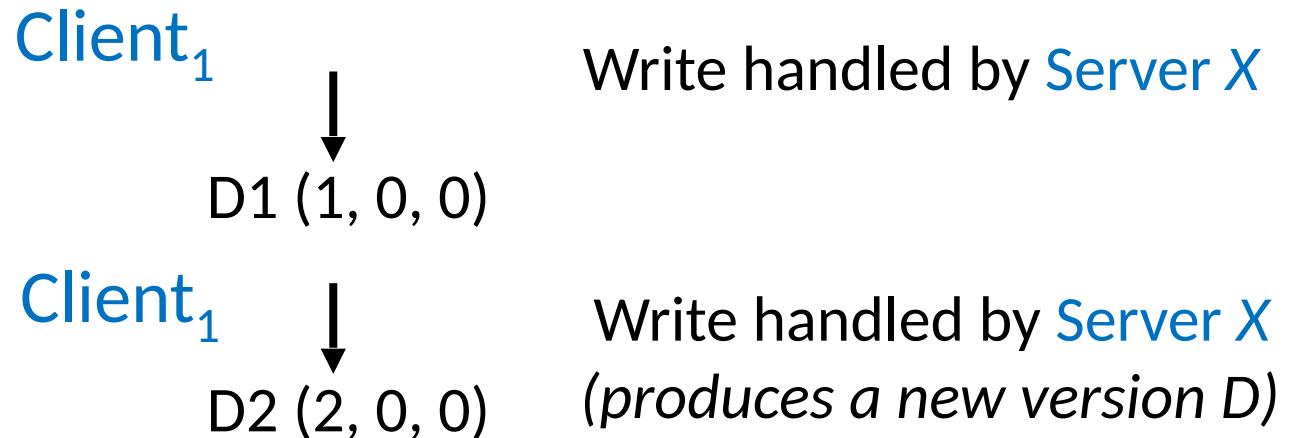
Properties of vector clocks

- Let a, b be two events with vector time stamps C_a, C_b then:
 - $a \rightarrow b$ iff $C_a < C_b$
 - $a \parallel b$ iff $C_a \parallel C_b$

Application of vector clocks in Dynamo

Versioning data objects

Assume three servers X, Y, Z , a data object D with object versions represented by a **vector clock** (X, Y, Z)

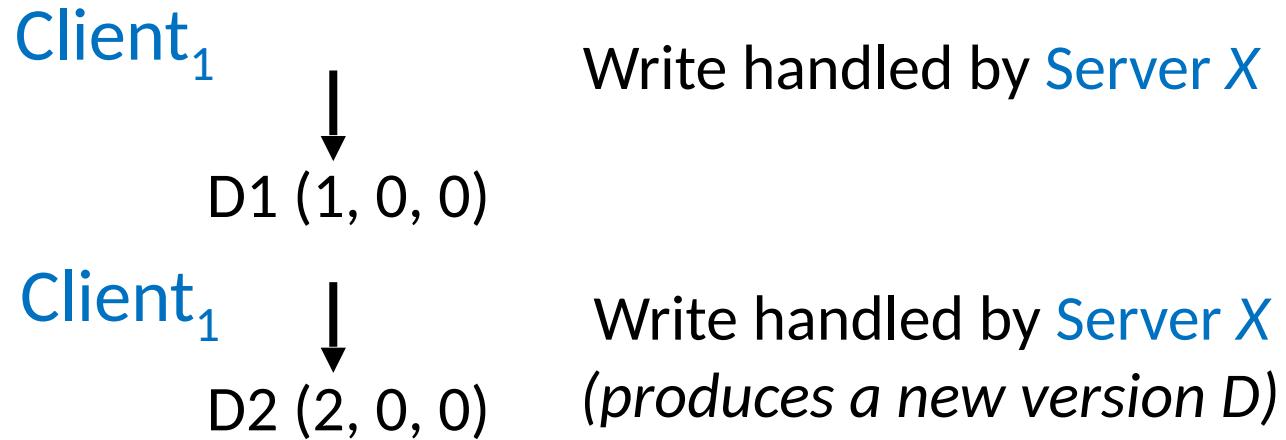


Application of vector clocks in Dynamo

Versioning data objects

Assume three servers X, Y, Z, a data object D with object versions represented by a **vector clock**

(X, Y, Z)



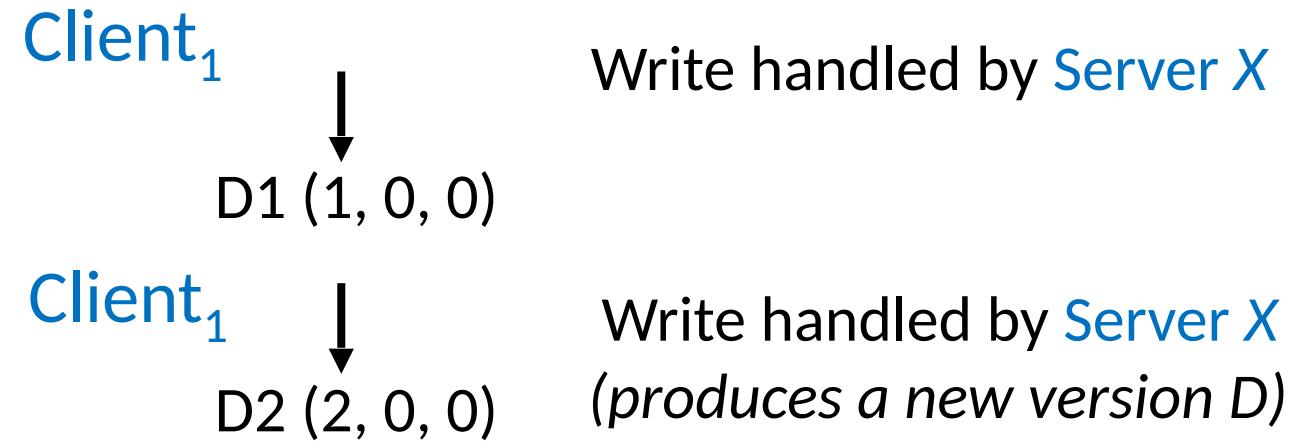
Dynamo can **syntactically reconcile** data (overwrite D1 with D2), D2 is a strictly newer version of D1.

Application of vector clocks in Dynamo

Versioning data objects

Assume three servers X, Y, Z, a data object D with object versions represented by a **vector clock**

(X, Y, Z)



Dynamo can **syntactically reconcile** data (overwrite D1 with D2), D2 is a strictly newer version of D1.

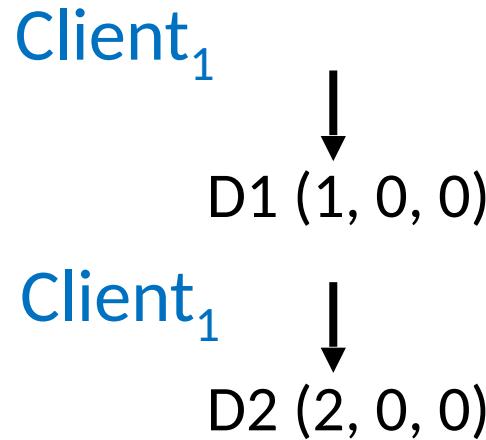
Internal to Dynamo, **updates** to data are **replicated** asynchronously to other servers.

Application of vector clocks in Dynamo

Versioning data objects

Assume three servers X, Y, Z, a data object D with object versions represented by a **vector clock**

(X, Y, Z)



Write handled by Server X

Write handled by Server X
(produces a new version D)

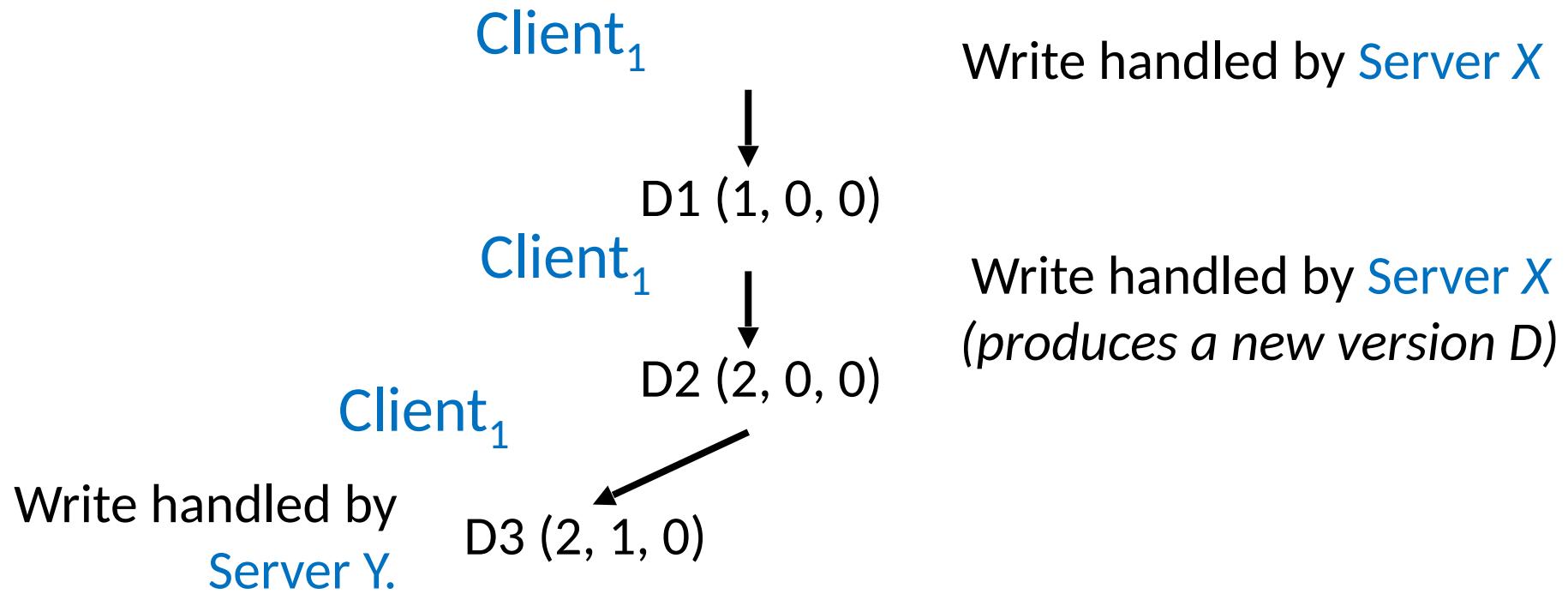
Dynamo can **syntactically reconcile** data (overwrite D1 with D2), D2 is a strictly newer version of D1.

Internal to Dynamo, **updates** to data are **replicated** asynchronously to other servers.

There may exist replicas of D1 that have not yet seen D2.

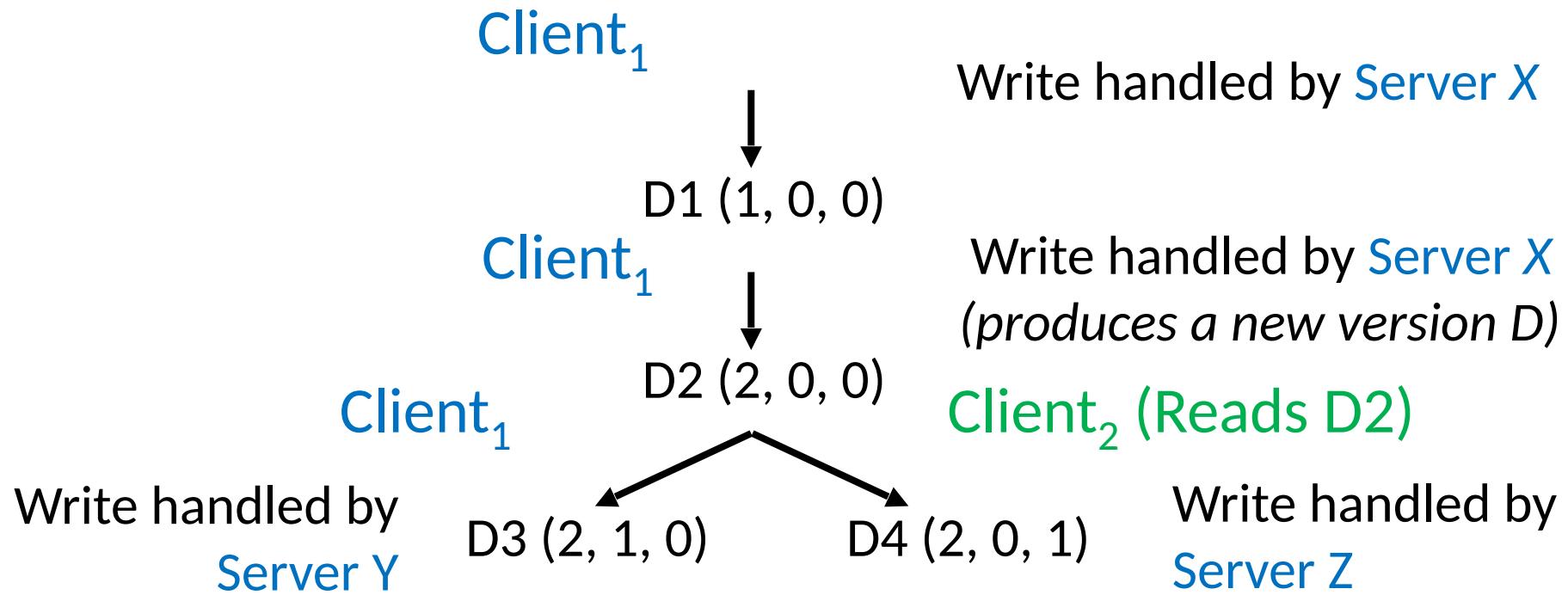
Application of vector clocks in Dynamo

Versioning data objects



Application of vector clocks in Dynamo

Versioning data objects



Application of vector clocks in Dynamo

Versioning data objects

- A node seeing D1 (1, 0, 0) and D2 (2, 0, 0)
 - $(1, 0, 0) \leq (2, 0, 0)$ (overwrite D1 with D2)

Application of vector clocks in Dynamo

Versioning data objects

- A node seeing D1 (1, 0, 0) and D2 (2, 0, 0)
 - $(1, 0, 0) \leq (2, 0, 0)$ (overwrite D1 with D2)
- A node seeing D1 (1, 0, 0), D2 (2, 0, 0) and D4 (2, 0, 1) (or D3 (2, 1, 0))
 - E.g., $(2, 0, 0) \leq (2, 0, 1)$ (overwrite D2 with D4)

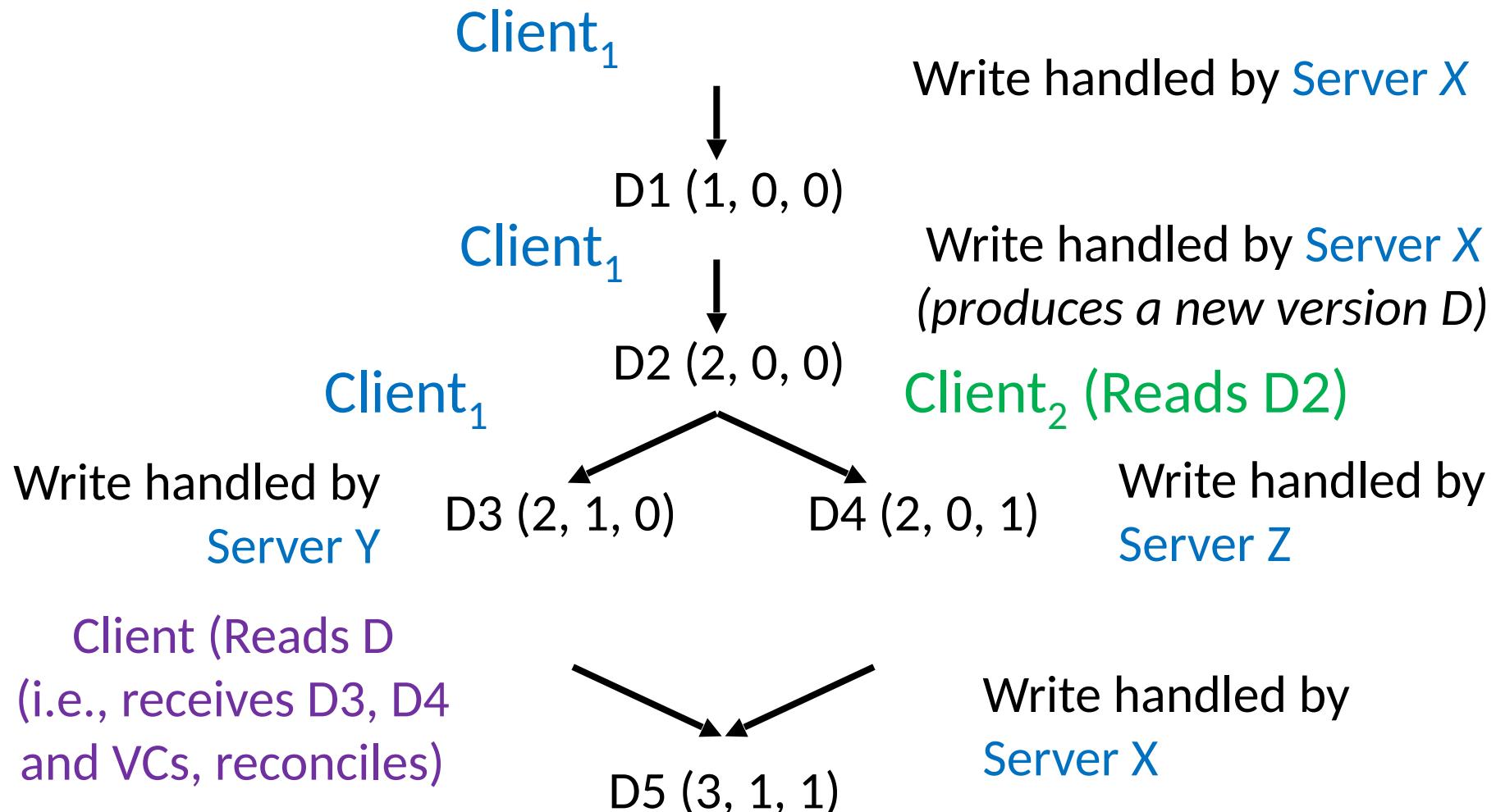
Application of vector clocks in Dynamo

Versioning data objects

- A node seeing D1 (1, 0, 0) and D2 (2, 0, 0)
 - $(1, 0, 0) \leq (2, 0, 0)$ (overwrite D1 with D2)
- A node seeing D1 (1, 0, 0), D2 (2, 0, 0) and D4 (2, 0, 1) (or D3 (2, 1, 0))
 - E.g., $(2, 0, 0) \leq (2, 0, 1)$ (overwrite D2 with D4)
- A node aware of D3 (2, 1, 0), receiving D4 (2, 0, 1)
 - $(2, 1, 0) \parallel (2, 0, 1)$ (no causal relation!)
 - Exist changes in D3, D4 that are not reflected in each others' version of the data
 - Both versions must be kept and presented to client for **semantic reconciliation**

Application of vector clocks in Dynamo

Versioning data objects



{}

Add beer
{beer}



Add chips
{beer, chips}

Delete beer
Add diapers
{chips, diapers}

Delete chips
Add sausages
{beer, sausages}

Reconciliation is to
“merge” shopping carts.
{beer, chips, diapers, sausages}

**Didn't loose “Add,” loose
“Delete”!**

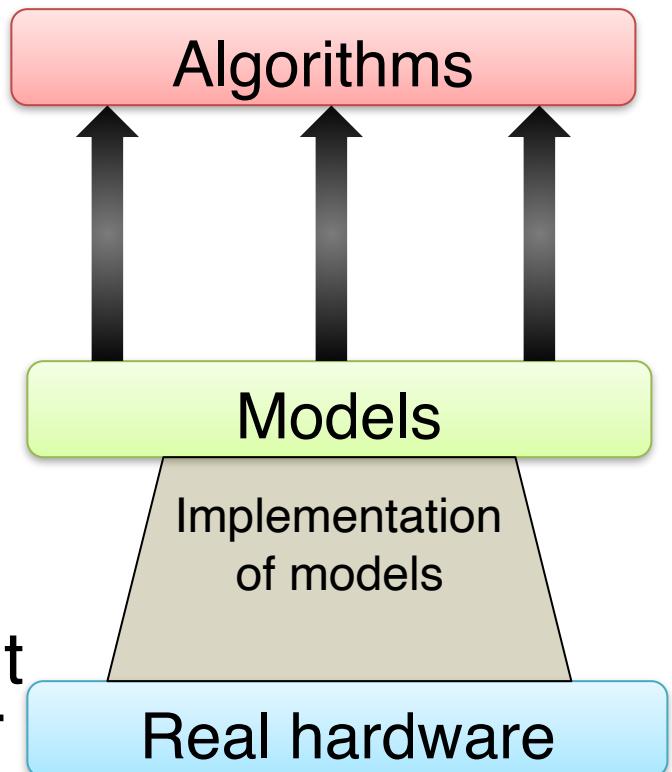
Agenda

- Synchronous vs. asynchronous systems
- Distributed mutual exclusion
 - Mutual exclusion with shared variables (recap)
 - Algorithms for distributed mutual exclusion
- Leader election
 - Problem definition
 - Algorithms

DISTRIBUTED SYSTEM MODELS

Distributed system model

- Model captures all the **assumptions** made about the system
- Including network, clocks, processor, etc.
- Algorithms always assume a certain model
- Some algorithms are only possible in stronger models
- Model is **theoretical**: whether it can be implemented is another question!



Synchronous vs. asynchronous model

Property	Synchronous system model	Asynchronous system model
Clocks	Bound on drift	No bound on drift
Processes	Bound on execution time	No bound on execution time
Network	Bound on latency	No bound on latency

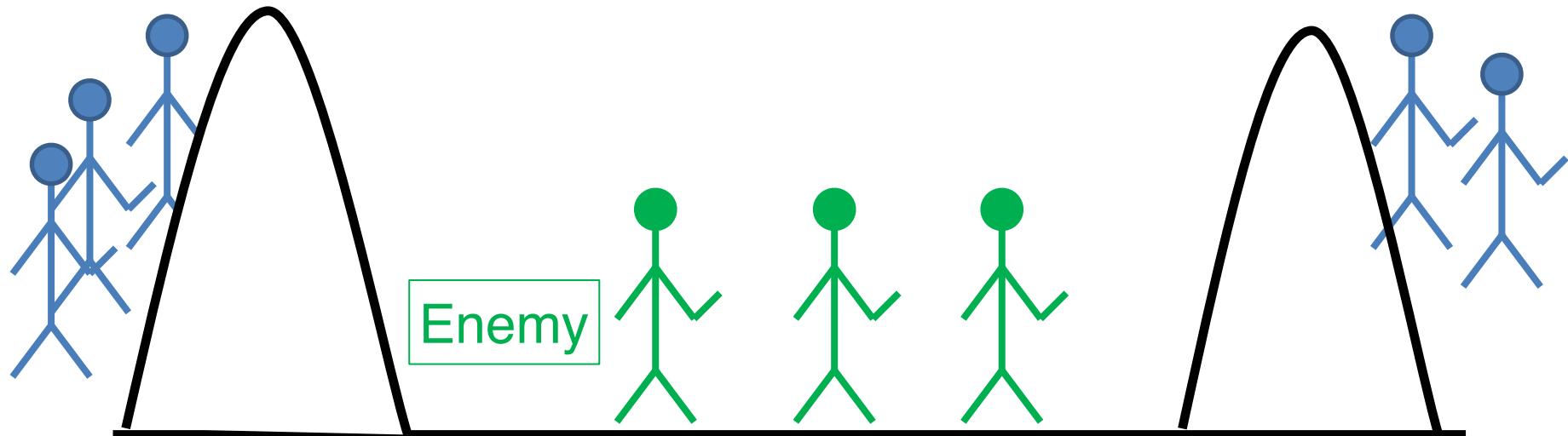
Two General's Problem (Agreement)

Armies need to agree on:

A thought experiment

Army 1

Army 2



Armies are safe if they
don't attack (or win)
(Safety)

Armies need to
coordinate attack to win.
(Liveness)

Two General's Problem (Agreement)

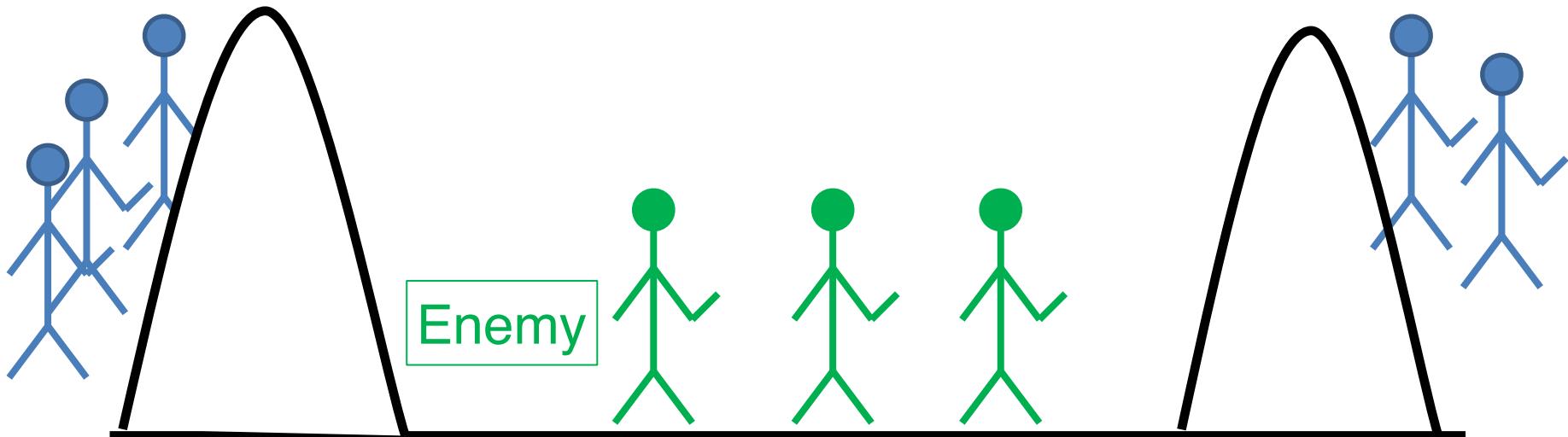
Armies need to agree on:

A thought experiment

Who leads the attack?
When to attack?

Army 1

Army 2



Armies are safe if they
don't attack (or win)
(Safety)

Armies need to
coordinate attack to win.
(Liveness)

Two General's Problem (Agreement)

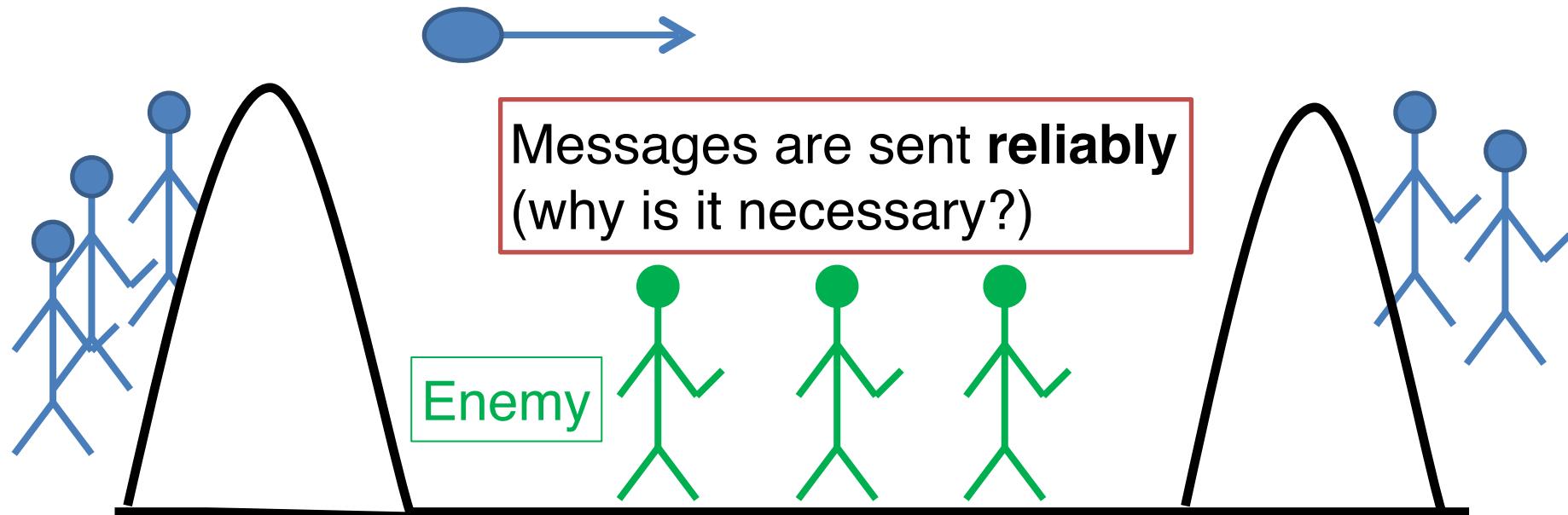
Armies need to agree on:

A thought experiment

Who leads the attack?
When to attack?

Army 1

Army 2



Armies are safe if they
don't attack (or win)
(Safety)

Armies need to
coordinate attack to win.
(Liveness)

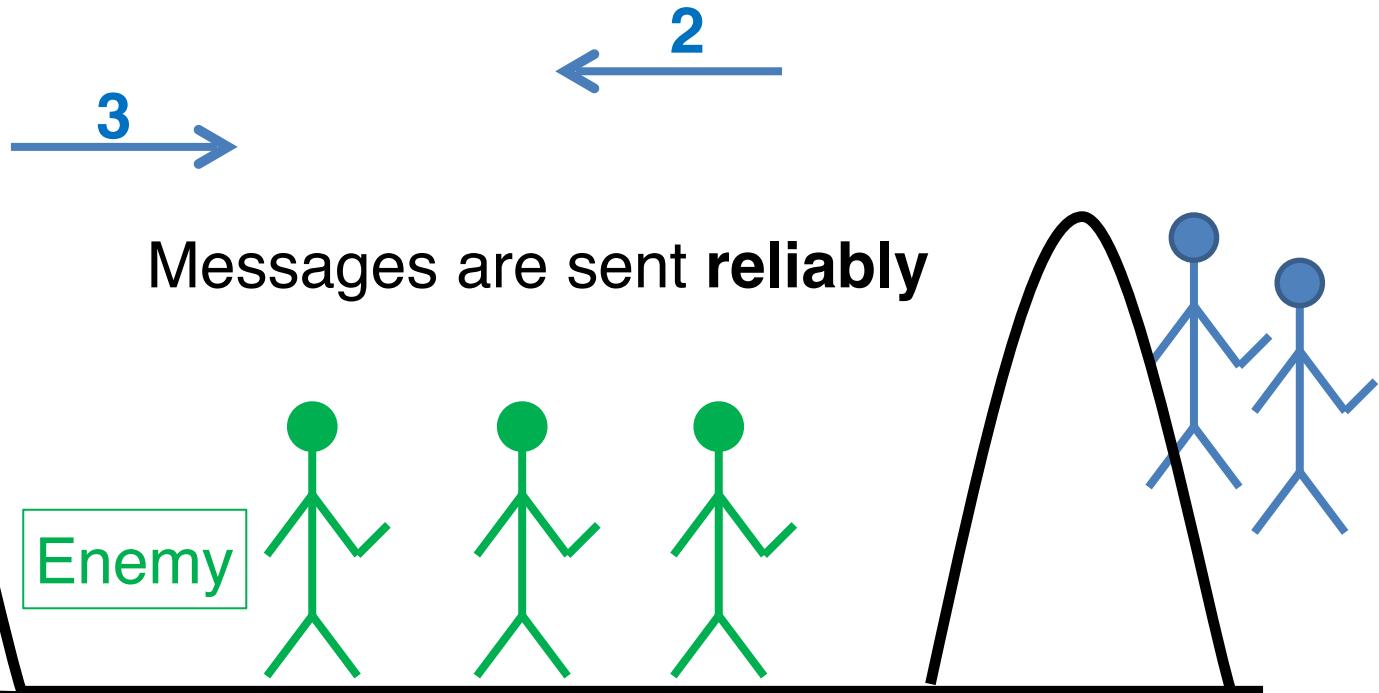
Synchronous vs. asynchronous agreement

Who leads the attack?

- Largest army
- If tied, Army 1

Army 1

Army 2



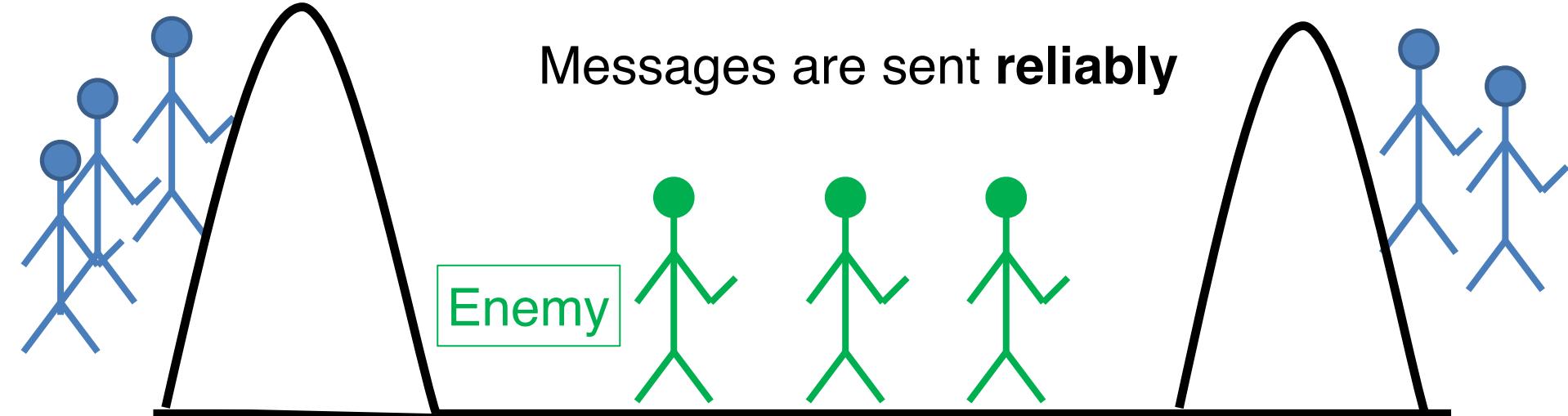
Asynchronous agreement

When to attack? No bound on delivery!

Army 1

Army 2

Messages are sent **reliably**



Asynchronous agreement

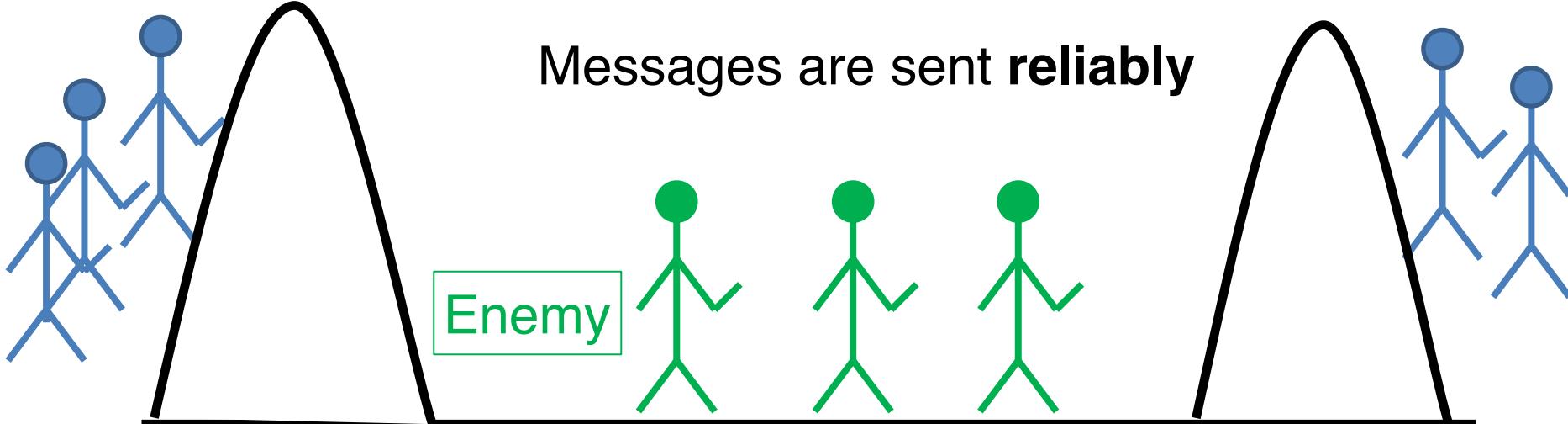
When to attack? No bound on delivery!

Army 1

Army 2

Attack!
→

Messages are sent **reliably**

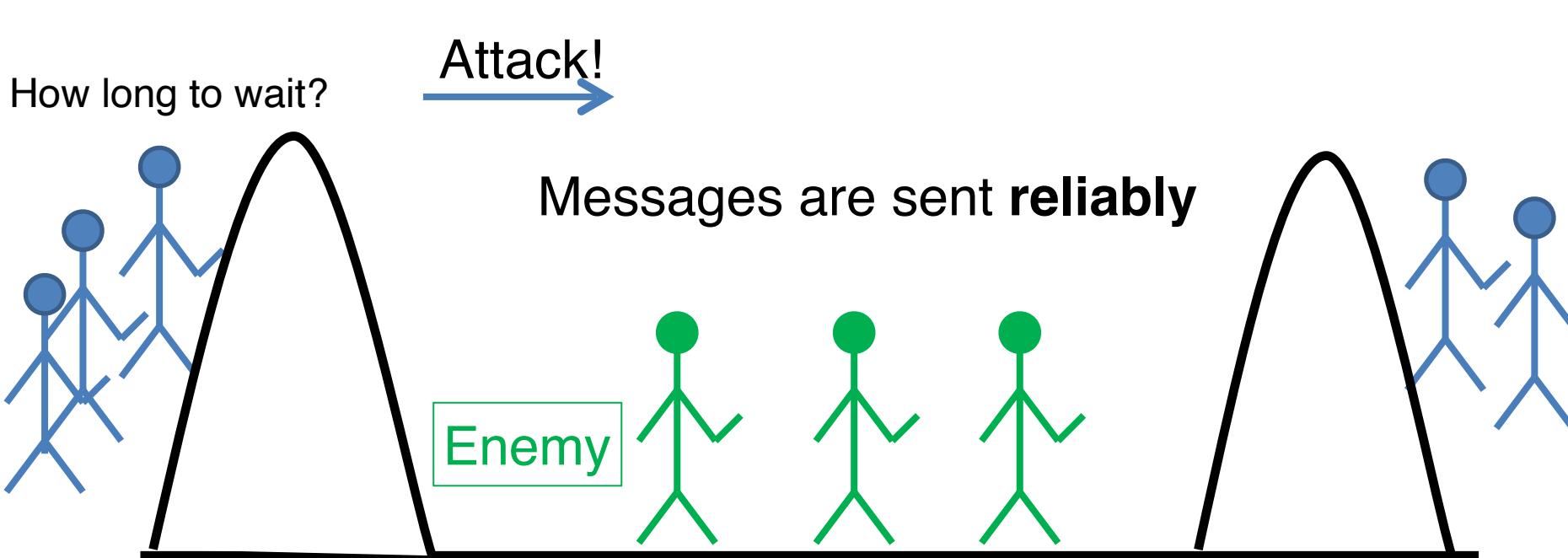


Asynchronous agreement

When to attack? No bound on delivery!

Army 1

Army 2



Synchronous agreement

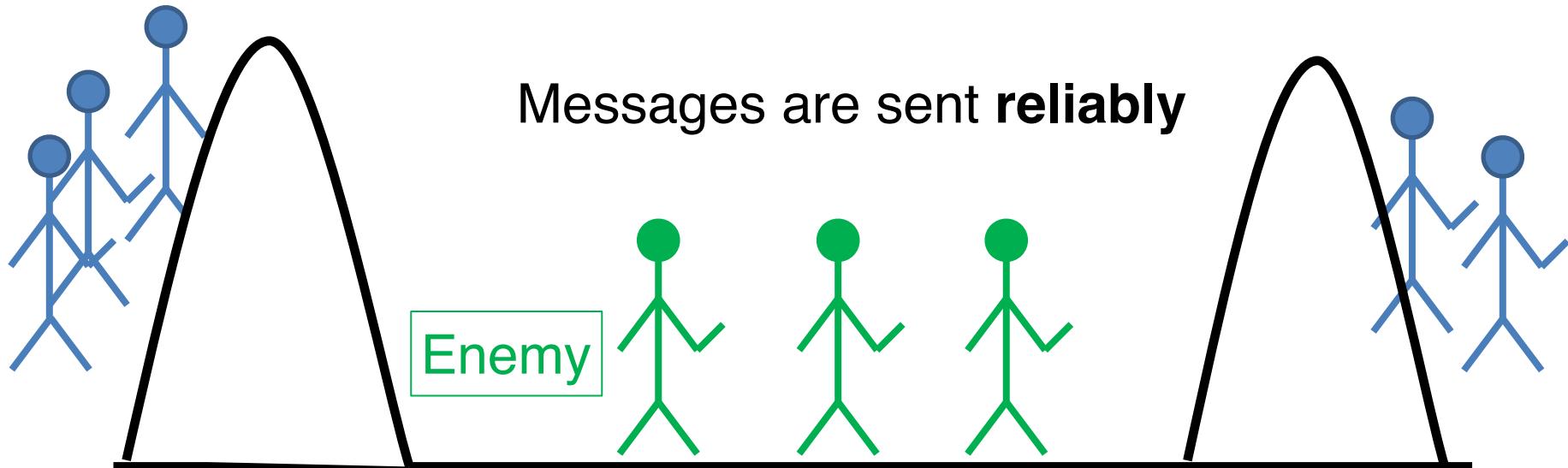
When to attack?

Assum
ing no
clocks

Army 1

Army 2

Messages are sent **reliably**



Synchronous agreement

When to attack?

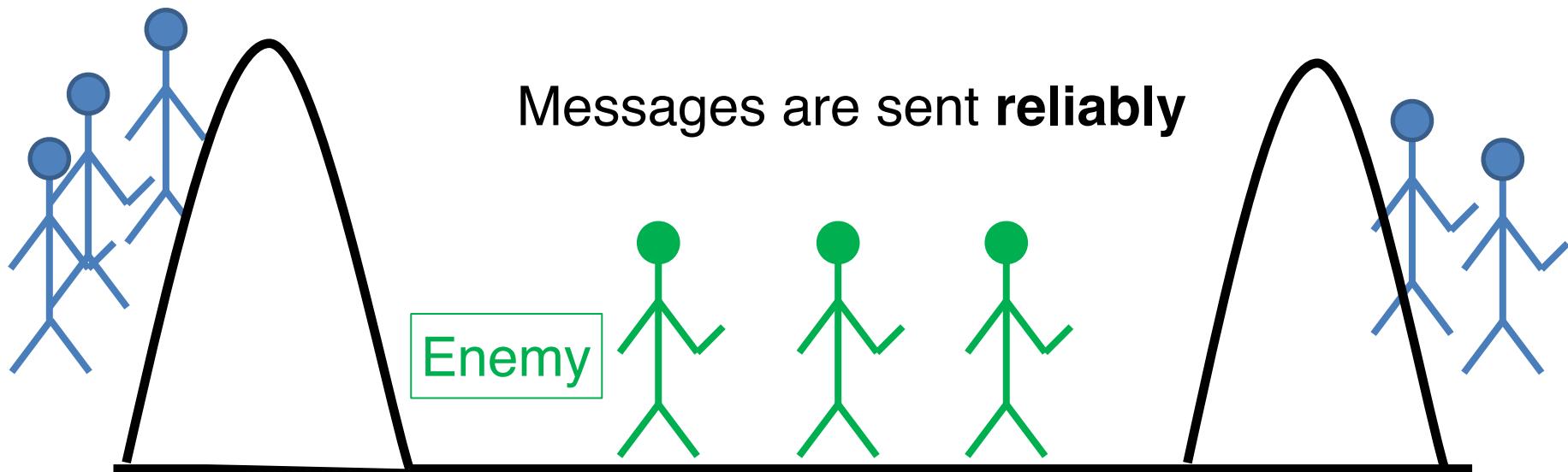
Assum
ing no
clocks

Army 1

Attack!

Army 2

Messages are sent **reliably**



Synchronous agreement

When to attack?

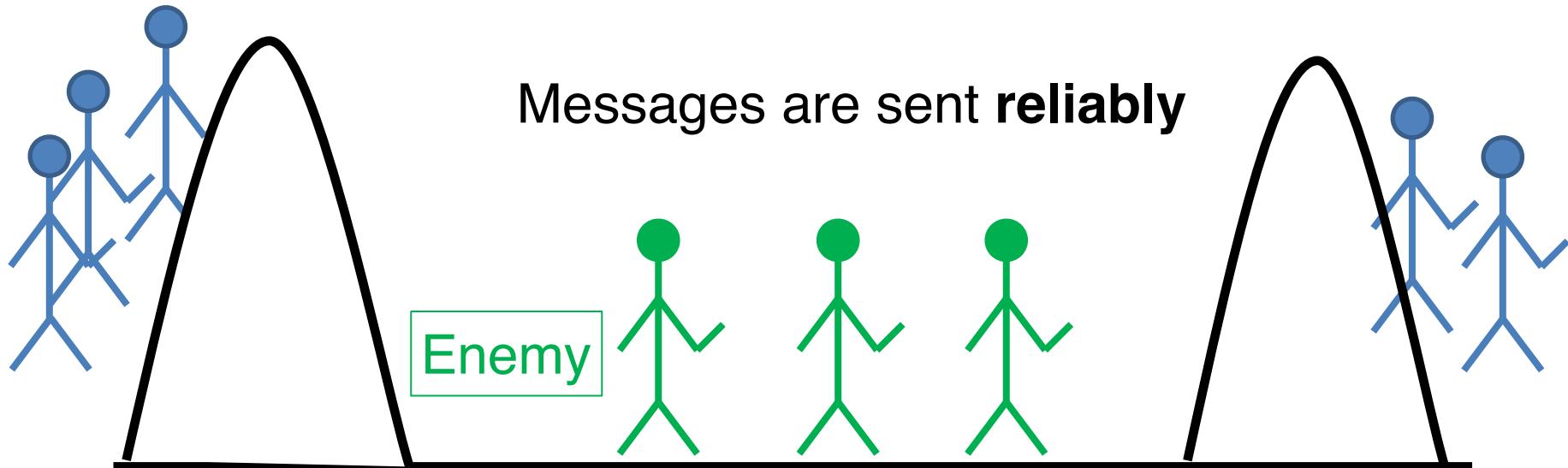
Assuming no clocks

Message takes at least **min** time
and at most **max** time to arrive

Army 1

Army 2

Attack!



Synchronous agreement

When to attack?

Assuming no clocks

Message takes at least **min** time
and at most **max** time to arrive

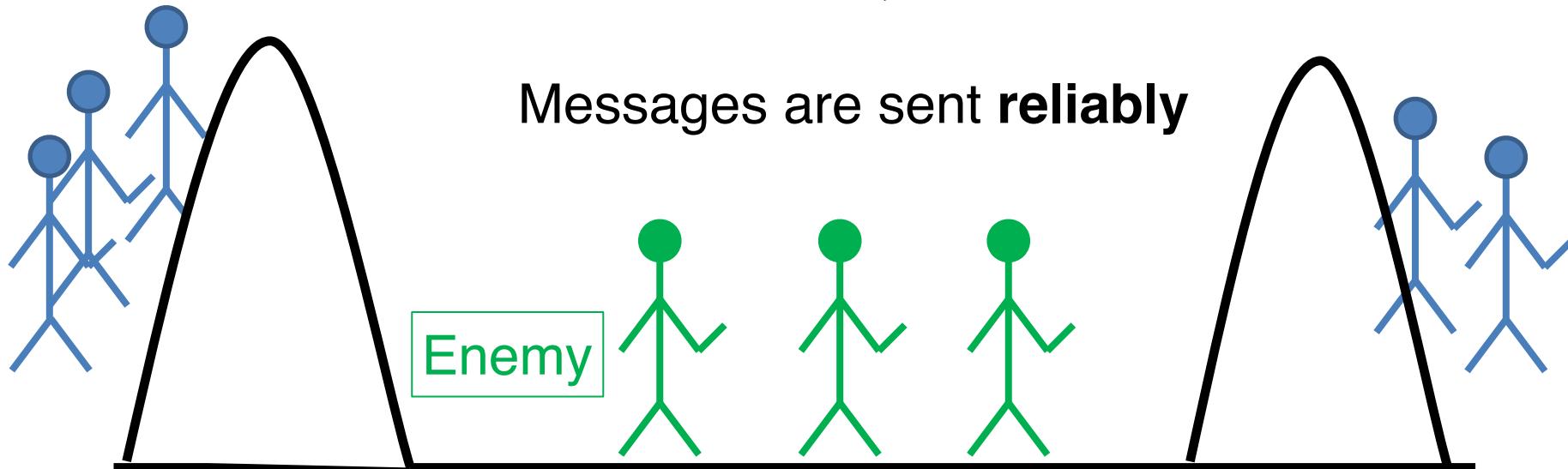
Army 1

Army 2

Attack!

Waits for min time, then attacks

Messages are sent **reliably**



Synchronous agreement

When to attack?

Assuming no clocks

Message takes at least **min** time
and at most **max** time to arrive

Army 1

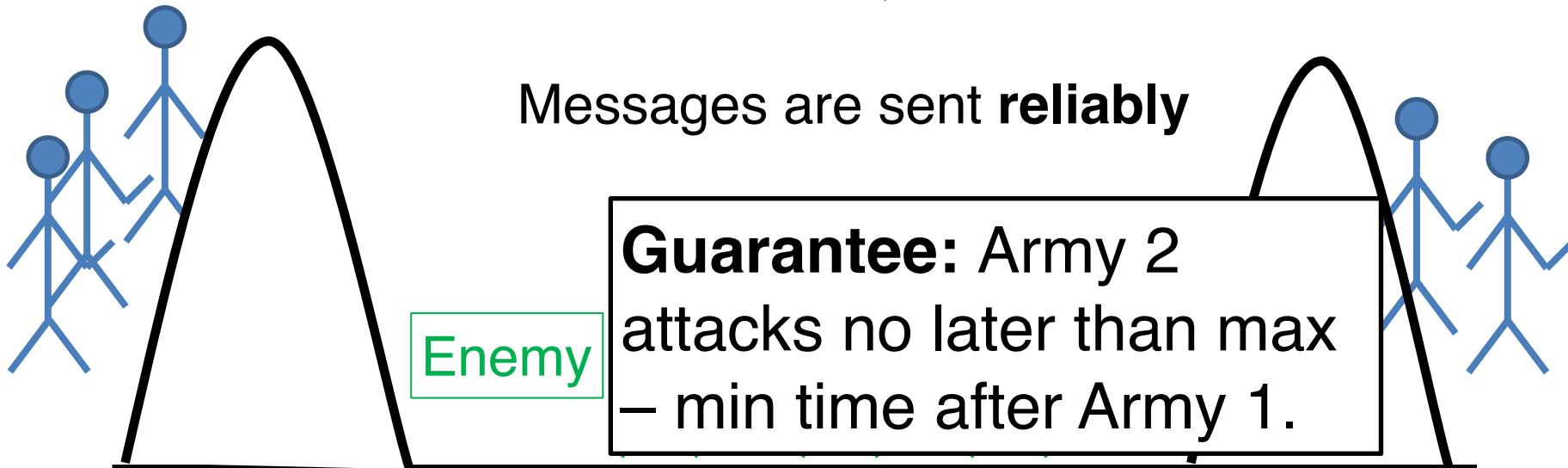
Army 2

Attack!

Waits for min time, then attacks

Messages are sent **reliably**

Guarantee: Army 2
attacks no later than max
– min time after Army 1.



Some takeaways

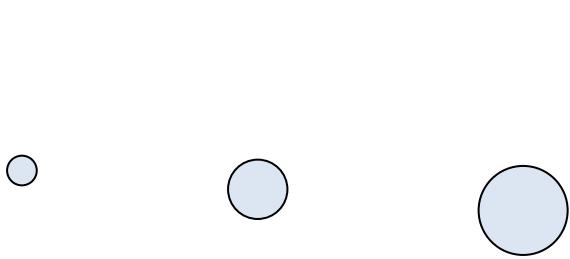
- Internet and many practical distributed applications are closer to asynchronous than synchronous model
- A solution valid for asynchronous distributed systems is also valid for synchronous ones (synchronous model is a stronger model)
- Many design problems cannot be solved in an asynchronous world (e.g., *when* vs. who leads attack)
- Apply **timeouts** and **timing assumptions** to reduce uncertainty and to bring elements of the synchronous model into the picture

You should have come
across this before. E.g.,
*Operating Systems or
Computer Organization*

RECAP: MUTUAL EXCLUSION

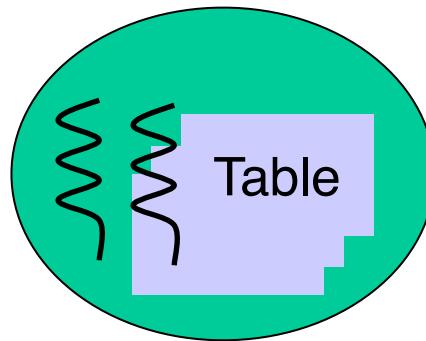
Problem: Access to shared variables

- Imaging a **globally shared variable** counter in a process accessible to multiple threads
 - For example, the **key-value records** managed by a storage server (or more complex data structure)



Let's dissect
the issue in
detail

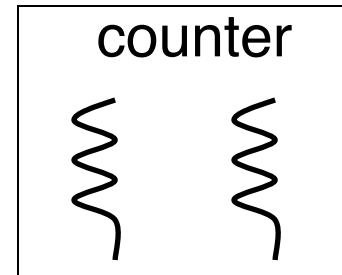
Shared data & synchronization



What may happen if multiple threads concurrently access shared process state (e.g., variables in memory)?

Concurrently manipulating shared data

- Two threads execute concurrently as part of the same process
- Shared variable (e.g., global variable)
 - counter = 5
- Thread 1 executes
 - counter++
- Thread 2 executes
 - counter--
- *What are all the possible values of counter after Thread 1 and Thread 2 finish executing?*



Machine-level implementation

- Implementation of “counter++”

register₁ = **counter**

register₁ = **register₁ + 1**

counter = **register₁**

- Implementation of “counter--”

register₂ = **counter**

register₂ = **register₂ – 1**

counter = **register₂**

Possible execution sequences



Context Switch



Context Switch



Interleaved execution

- Assume **counter** is 5 and interleaved execution of counter++ (in T_1) and counter-- (in T_2)

$T_1: r_1 = \text{counter}$ ($\text{register}_1 = 5$)

$T_1: r_1 = r_1 + 1$ ($\text{register}_1 = 6$)

$T_2: r_2 = \text{counter}$ ($\text{register}_2 = 5$)

$T_2: r_2 = r_2 - 1$ ($\text{register}_2 = 4$)

$T_1: \text{counter} = r_1$ ($\text{counter} = 6$)

$T_2: \text{counter} = r_2$ ($\text{counter} = 4$)

Context
switch

- The value of **counter** may be 4 or 6, whereas the correct result should be 5!

Race condition

- **Race condition**
 - **Several threads manipulate shared data concurrently.** The final value of the data depends upon which thread finishes last.
- In our example (interleaved execution) of counter++ with counter--
- To prevent race conditions, concurrent processes must be **synchronized!**

The moral of this story

- The statements
`counter++;`
`counter--;`
must each be executed ***atomically***.
- Atomic operation means an operation that **completes in its entirety without interruption**.
- This is achieved through **synchronization primitives**
- Shared variable accessed in **critical section**,
protected by synchronization primitives
- Known as the **critical section problem** or as **mutual exclusion**

“Don’t worry”, in distributed systems (the ones we look at), there is no shared memory – but,

...

DISTRIBUTED MUTUAL EXCLUSION

Distributed mutual exclusion

- In distributed systems, mutual exclusion is more complex due to lack of:
 - Shared memory
 - Timing issues
 - Lack of a global clock
 - Event ordering
- Applications
 - Accessing a shared resource in distributed systems
 - One active Bigtable master to coordinate

Critical section (CS) problem: No shared memory

- System with n processes
- Processes access shared resources in CS
- **Coordinate access to CS via message passing**
- Application-level protocol for accessing CS
 - Enter_CS() – enter CS, block if necessary
 - ResourceAccess() – access shared resource in CS
 - Exit_CS() – leave CS

Assumptions

(Theoretical nature, unless stated otherwise)

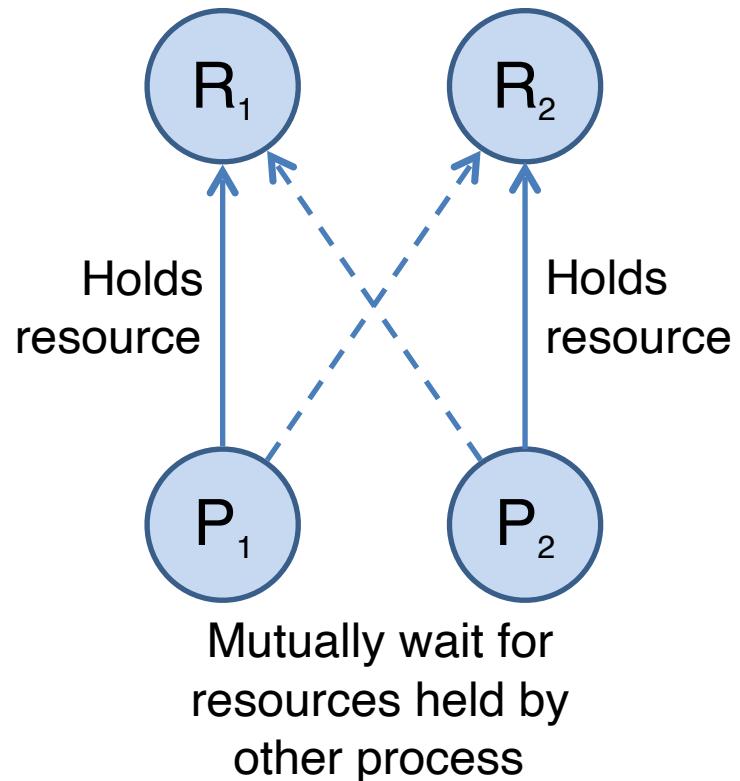
- System is asynchronous
 - No bound on delays, no bound on clock drift, etc.
- Processes do not fail
- Message delivery is reliable
 - Any message sent, is eventually delivered intact and exactly once
- Client processes are well-behaved and spent finite time accessing resources in CS

Mutual exclusion requirements

- **Safety**
 - At most one process in the critical section at a time
- **Liveness**
 - Requests to enter & exit CS eventually succeed
 - No deadlock
- **Fairness** (order & starvation)
 - If one request to enter CS **happened-before** another one, then entry to CS is granted in that **order**
 - Requests are ordered such that no process enters the critical section twice while another waits to enter (i.e., **no starvation**)

Deadlock & starvation

- **Deadlock:** Two or more processes become **stuck indefinitely** while attempting to enter and exit the CS, by virtue of their mutual dependency
- **Starvation:** The **indefinite postponement** of entry to CS for a process that has requested it.



Possible performance metrics

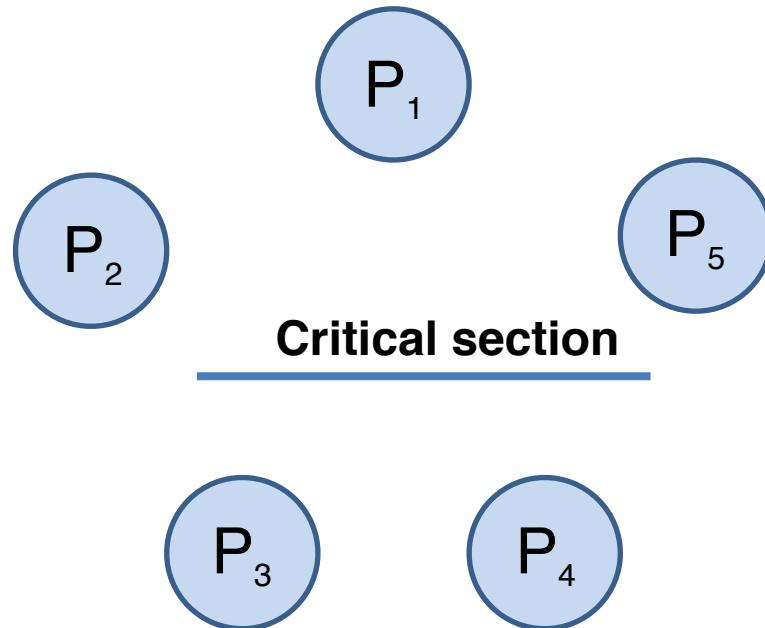
- **Bandwidth:** Number of messages sent, received or both
- **Synchronization delay:** Time between one process **exiting** the critical section and the next **entering**
- **Client delay:** Delay at entry and exit (response time)
- We do not measure client access to resources protected by the critical section (**assume finite**)

Solution strategies

- **Centralized strategy**
 - Divide processes into **master** and **slaves**, master dictates actions of slaves
- **Distributed strategy:** Each process independently decides actions, based on local knowledge of others' state
 - **Token-based:** A node is allowed in the critical section (CS) if it has a token. Tokens are passed from site to site, in some (priority) order.
 - **Non-token-based:** A node enters CS when an **assertion becomes true**. A node communicates with other nodes to obtain their states and decides whether the assertion is true or false.

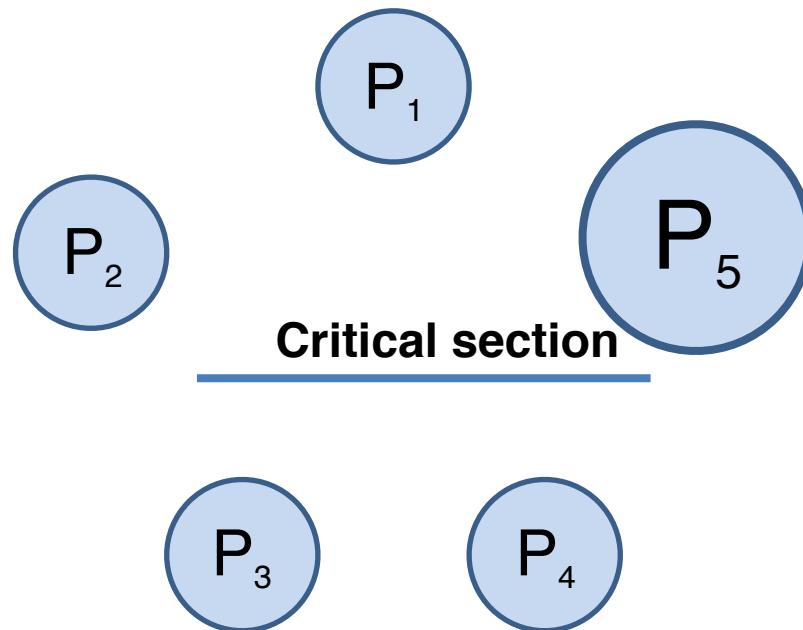
Centralized strategy

1. Elect a leader (details, cf. soon)



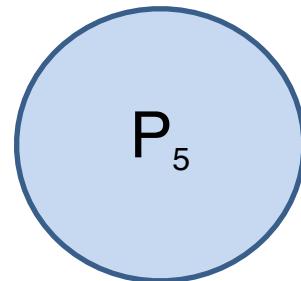
Centralized strategy

1. Elect a leader (details, cf. soon)

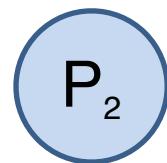
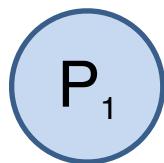


Centralized strategy: Empty CS

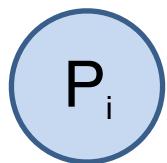
Server / coordinator



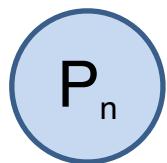
Critical section



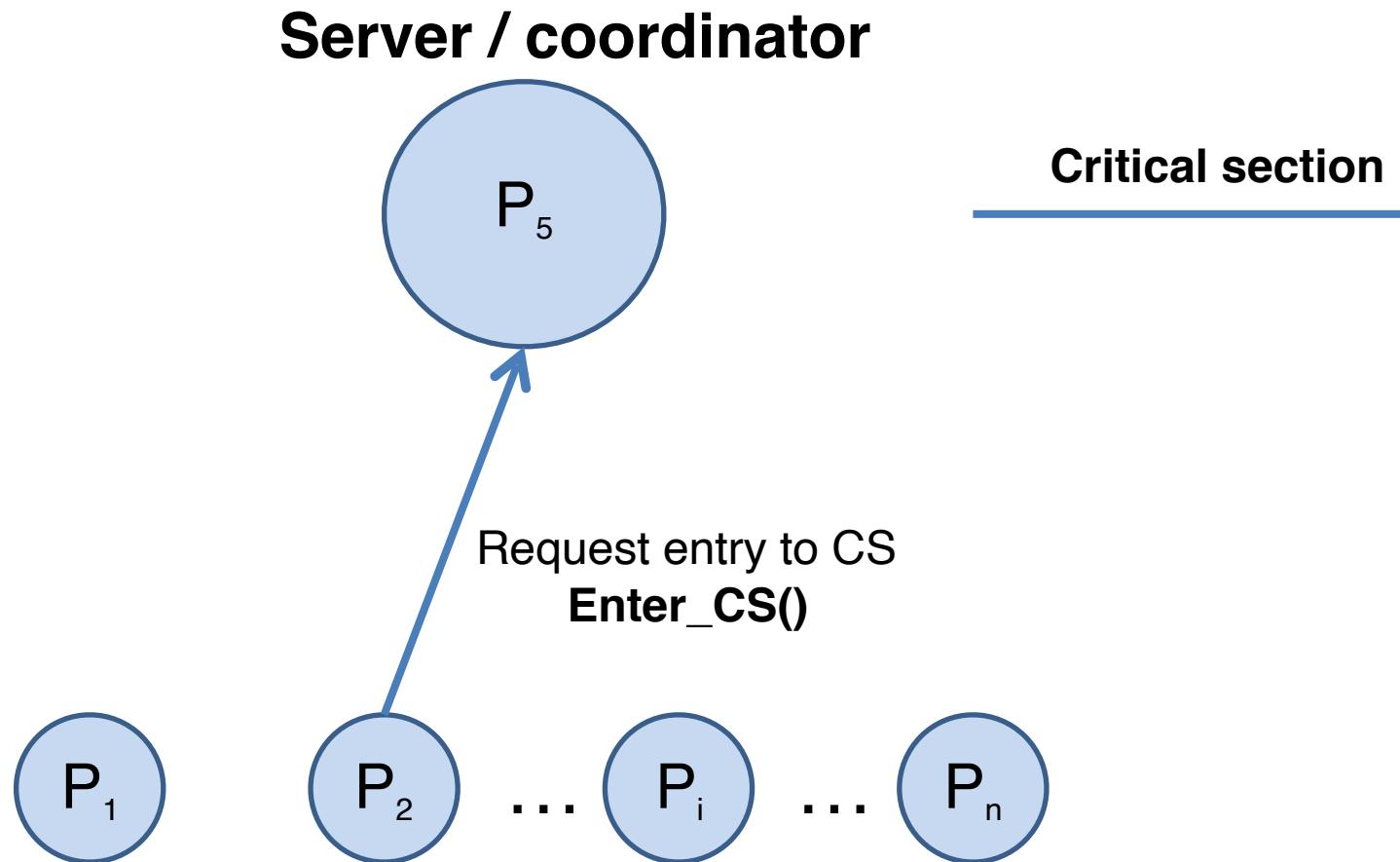
...



...

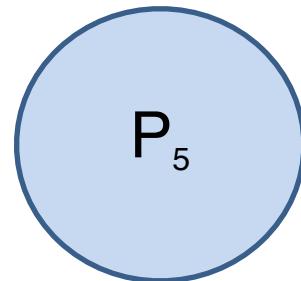


Centralized strategy: Empty CS

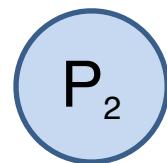
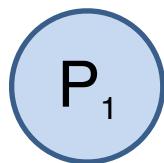


Centralized strategy: Empty CS

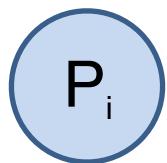
Server / coordinator



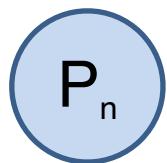
Critical section



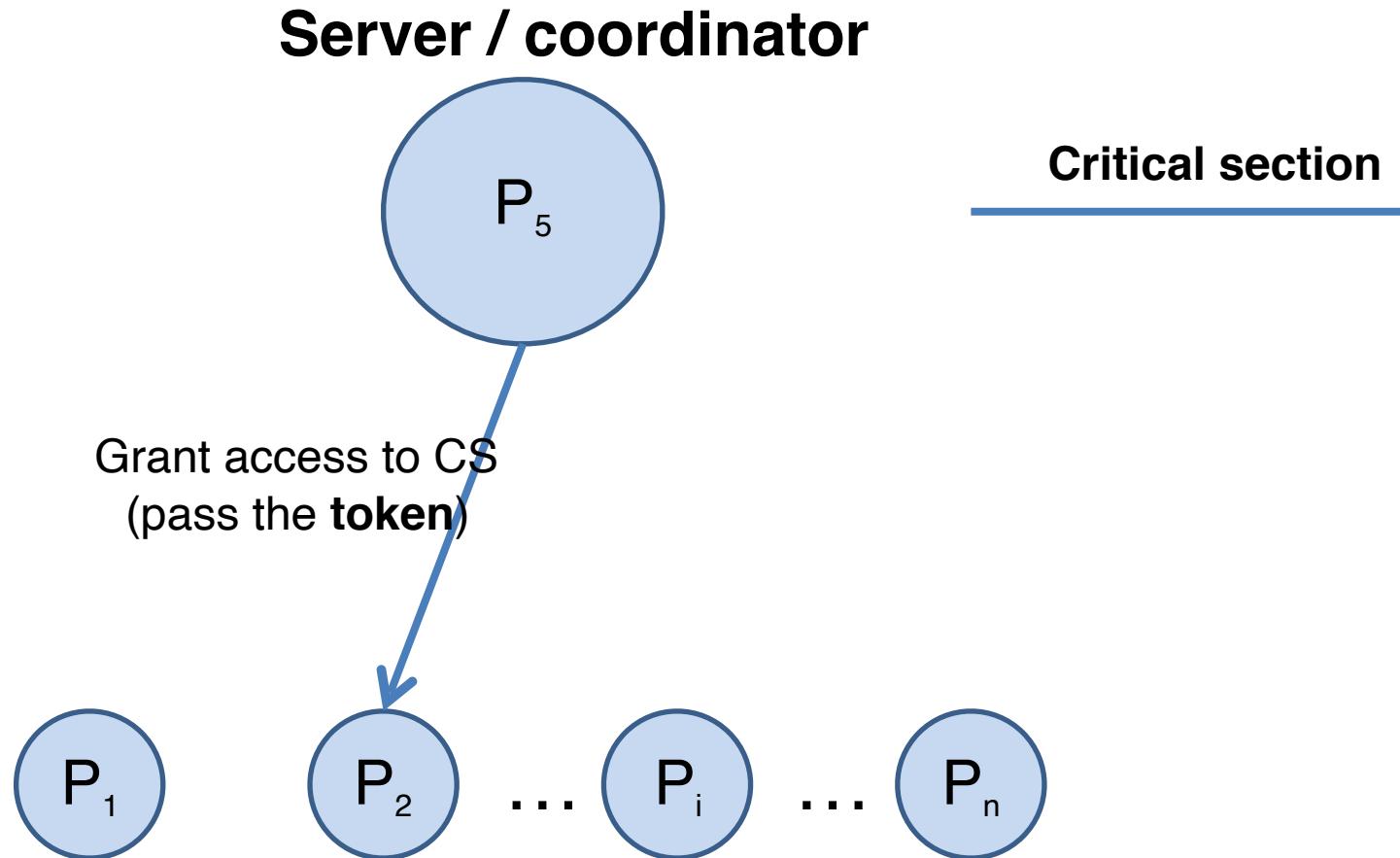
...



...

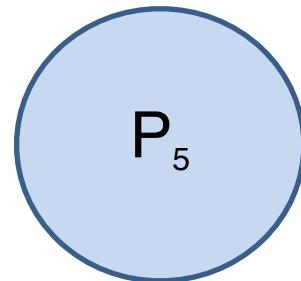


Centralized strategy: Empty CS

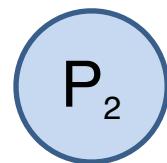
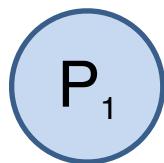


Centralized strategy: Empty CS

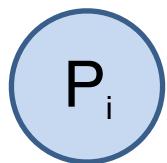
Server / coordinator



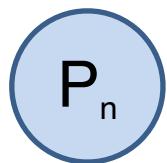
Critical section



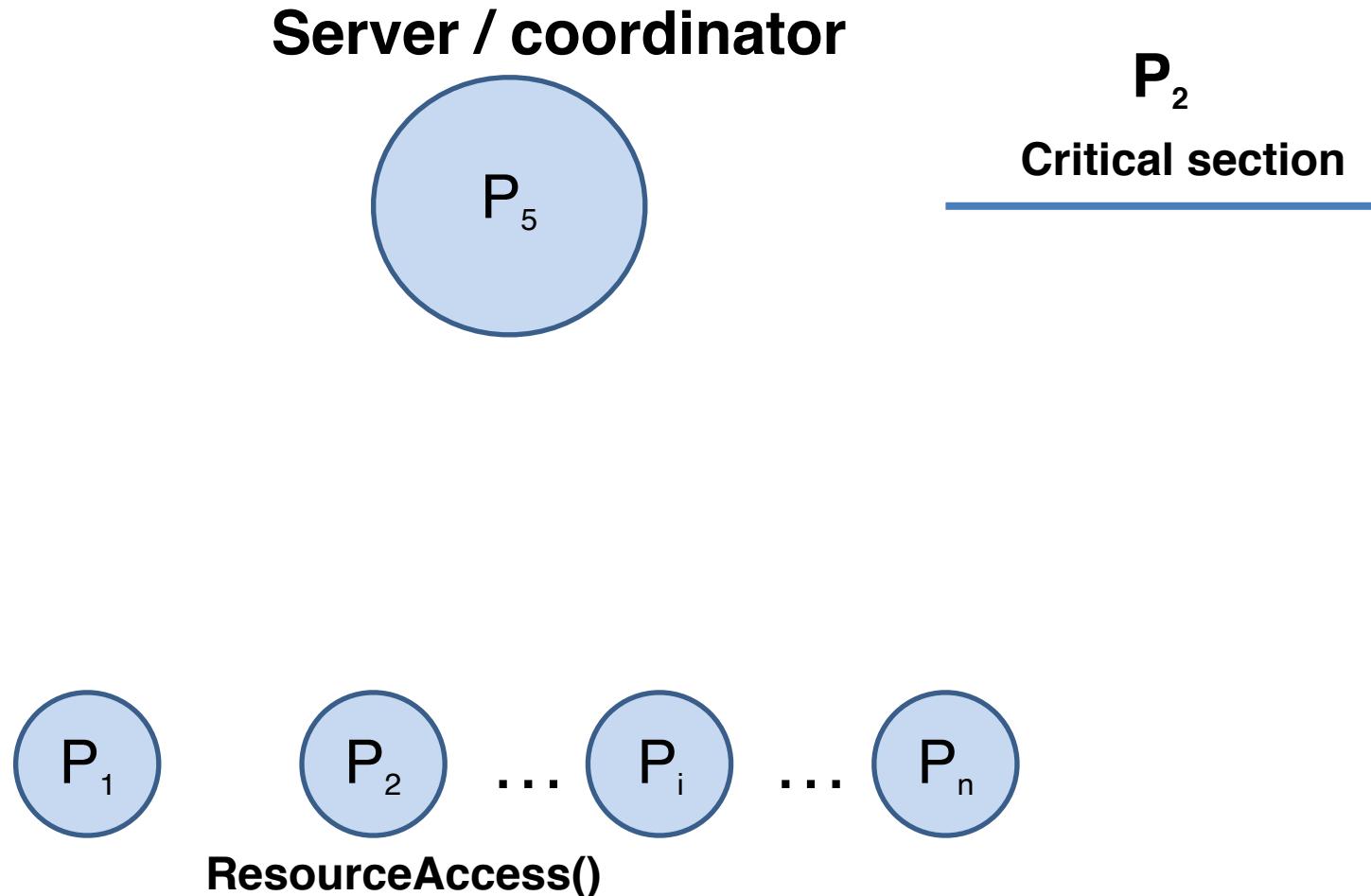
...



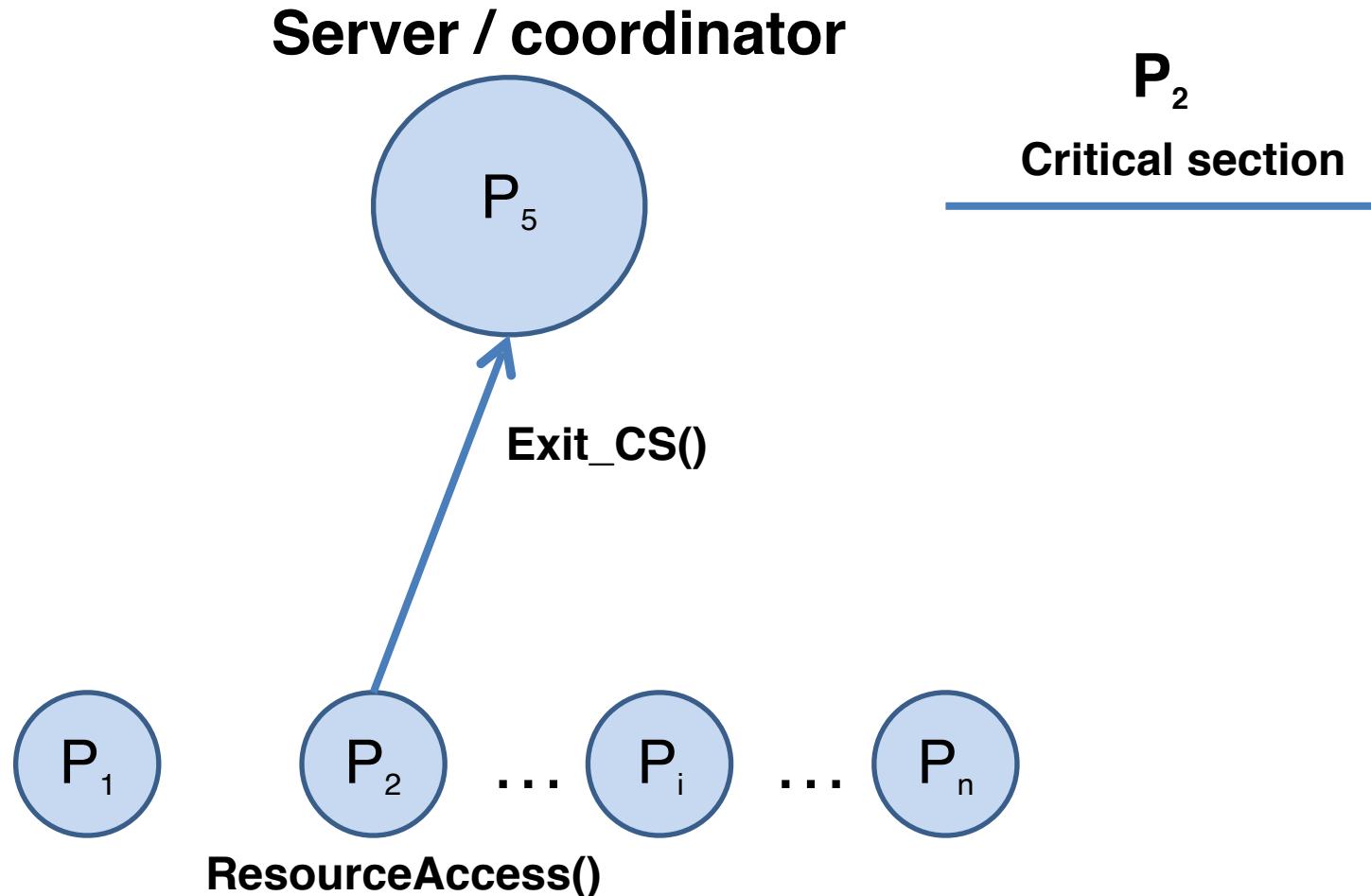
...



Centralized strategy: Empty CS

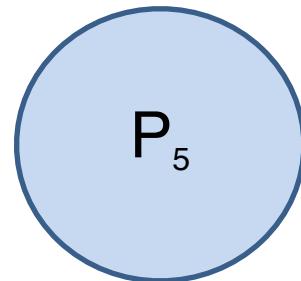


Centralized strategy: Empty CS

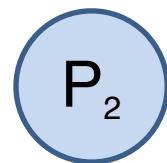
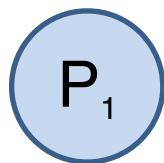


Centralized strategy: Empty CS

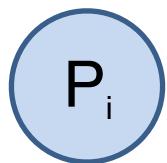
Server / coordinator



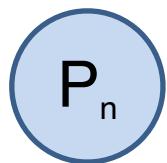
Critical section



...



...



Centralized strategy: Non-empty CS

Pending requests of processes waiting for entry to CS

Queue

Server / coordinator

P_5

P_2

Critical section

P_1

P_2

...

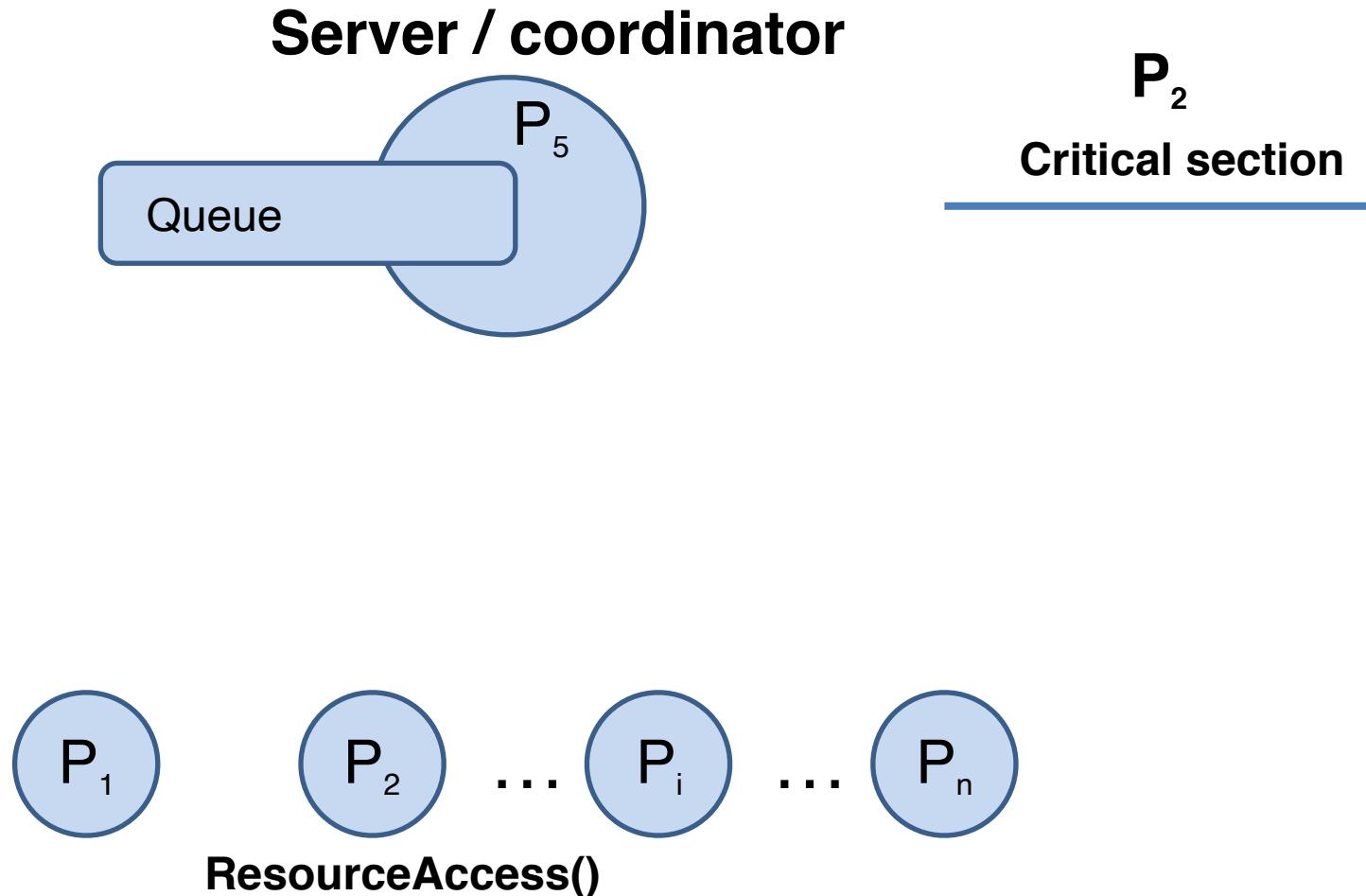
P_i

...

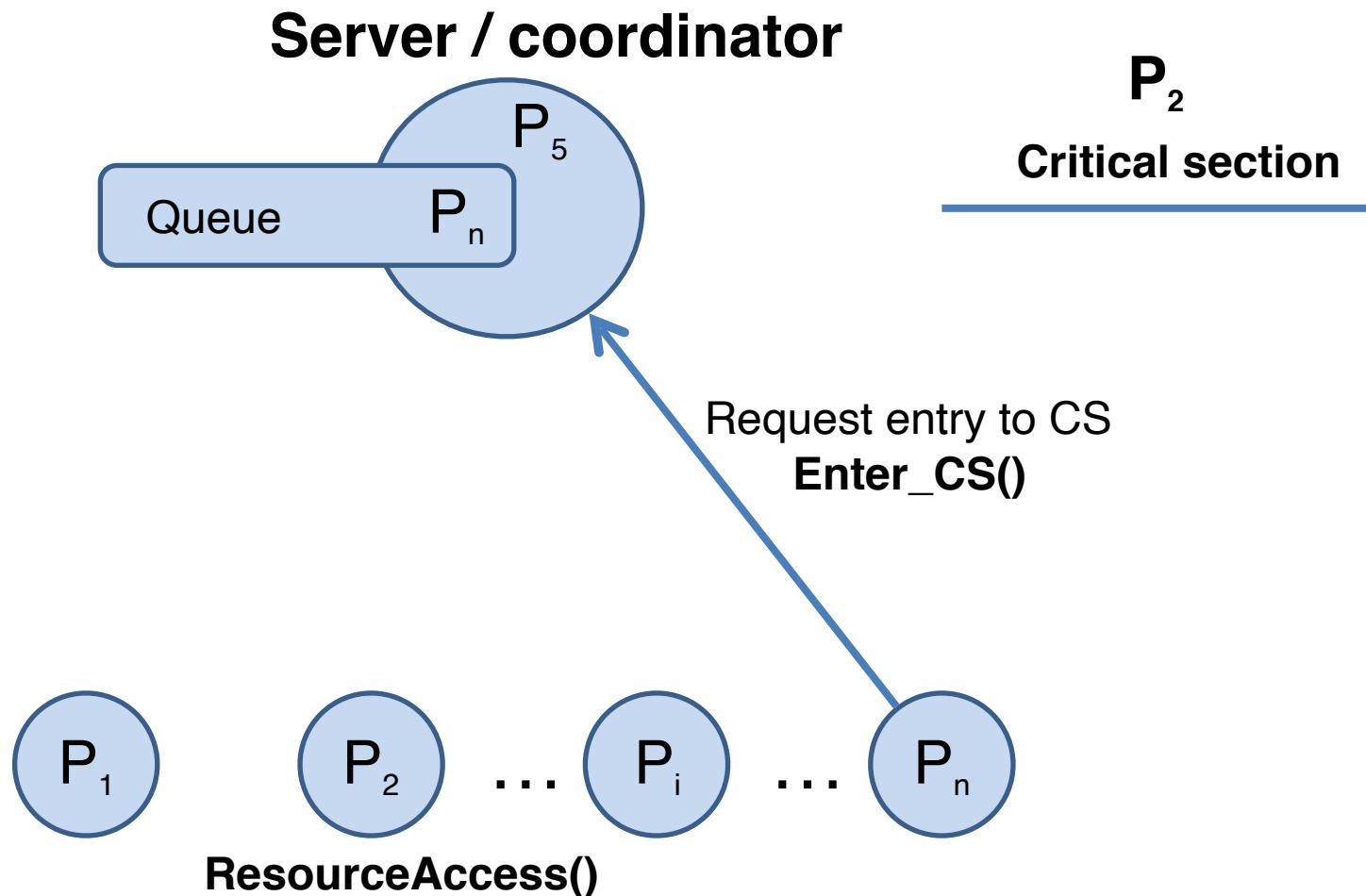
P_n

ResourceAccess()

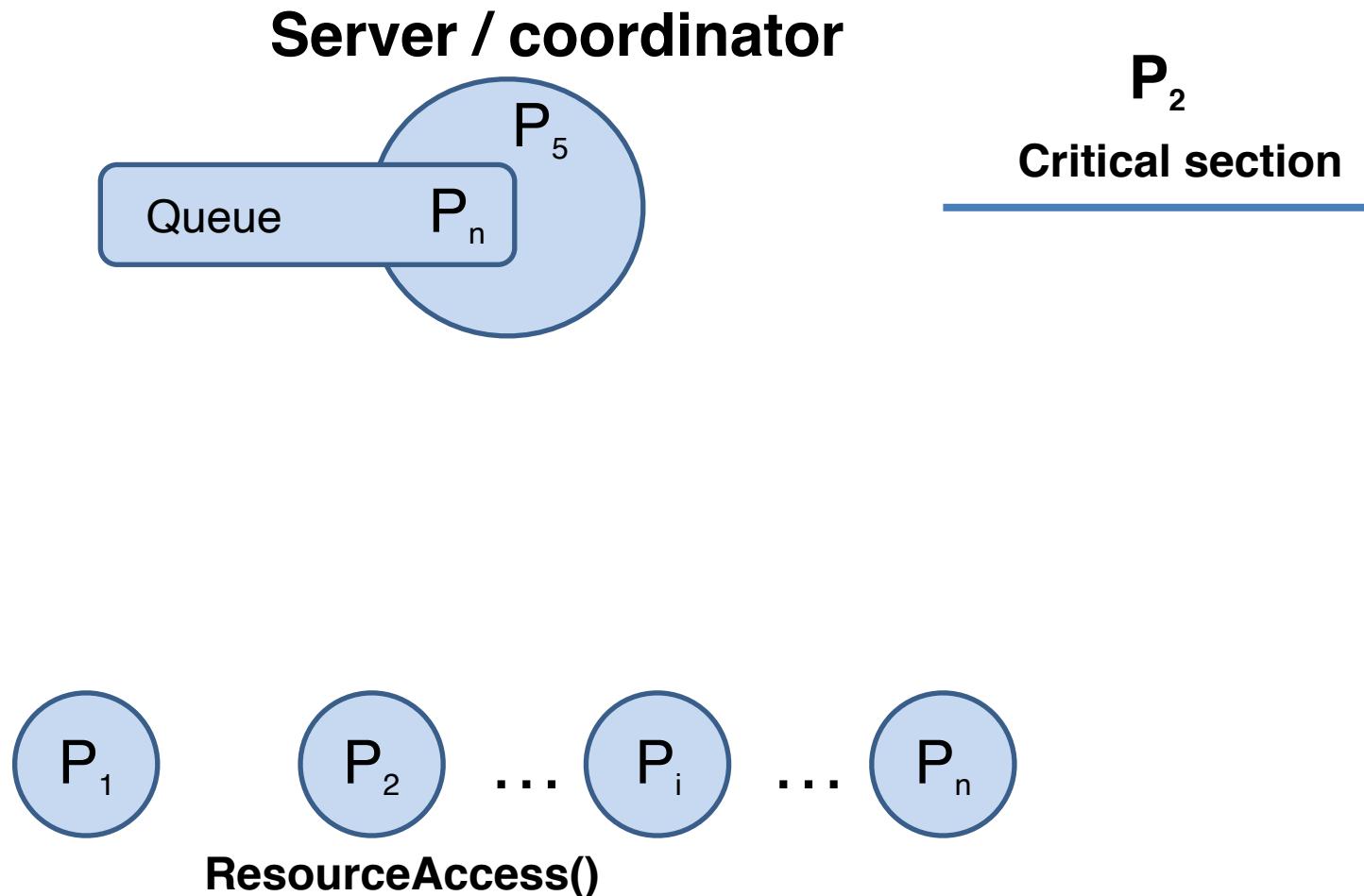
Centralized strategy: Non-empty CS



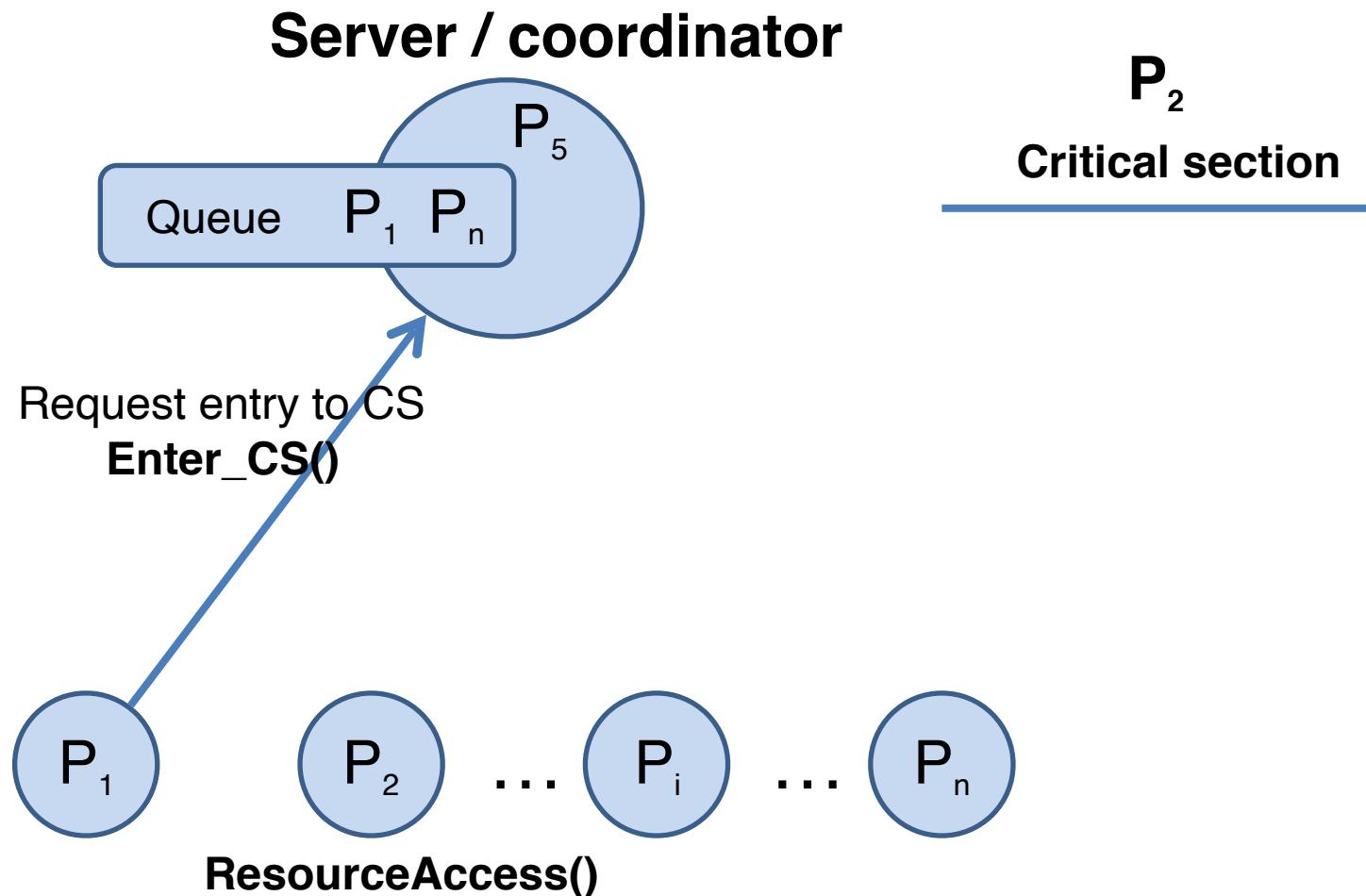
Centralized strategy: Non-empty CS



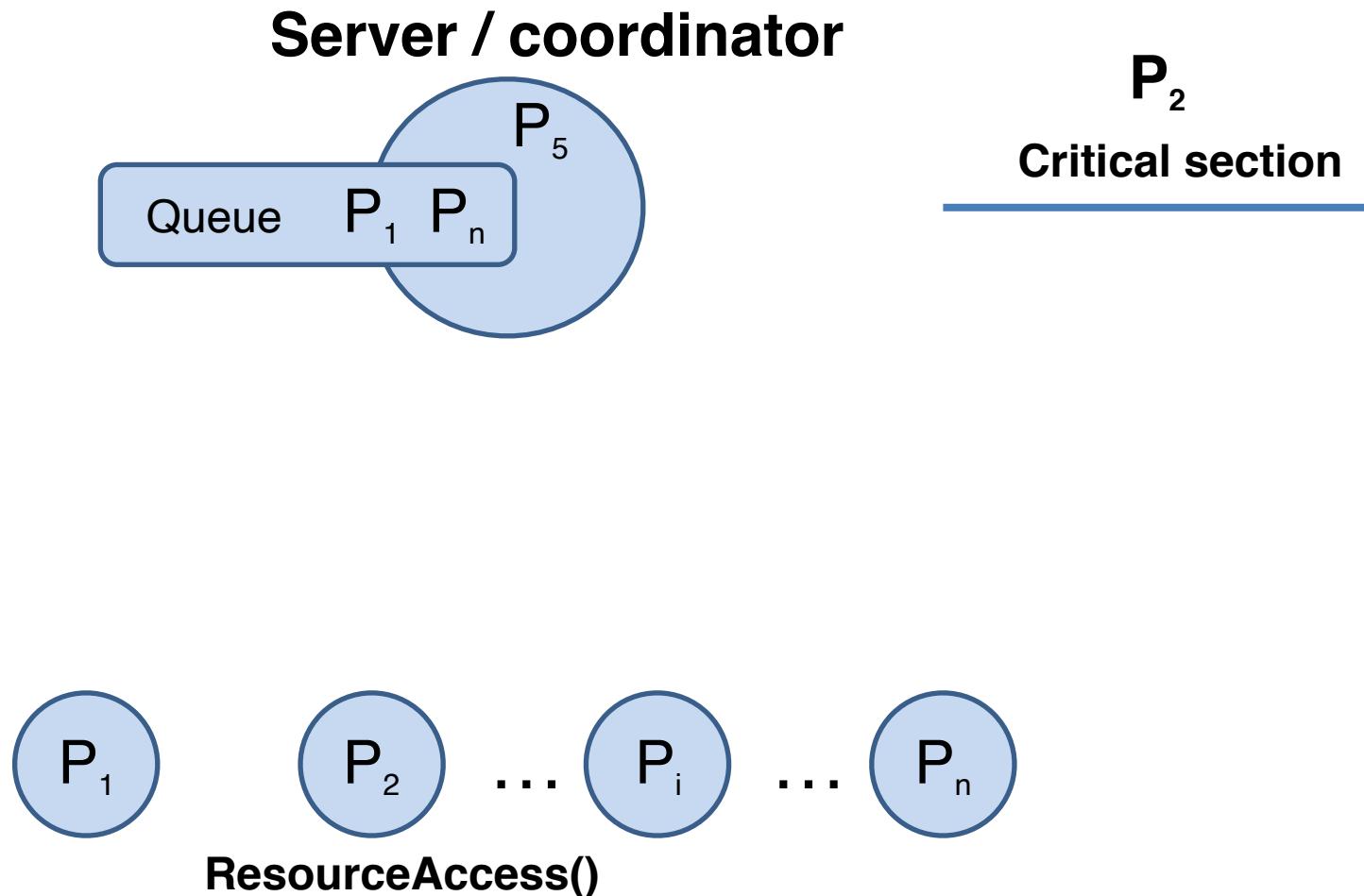
Centralized strategy: Non-empty CS



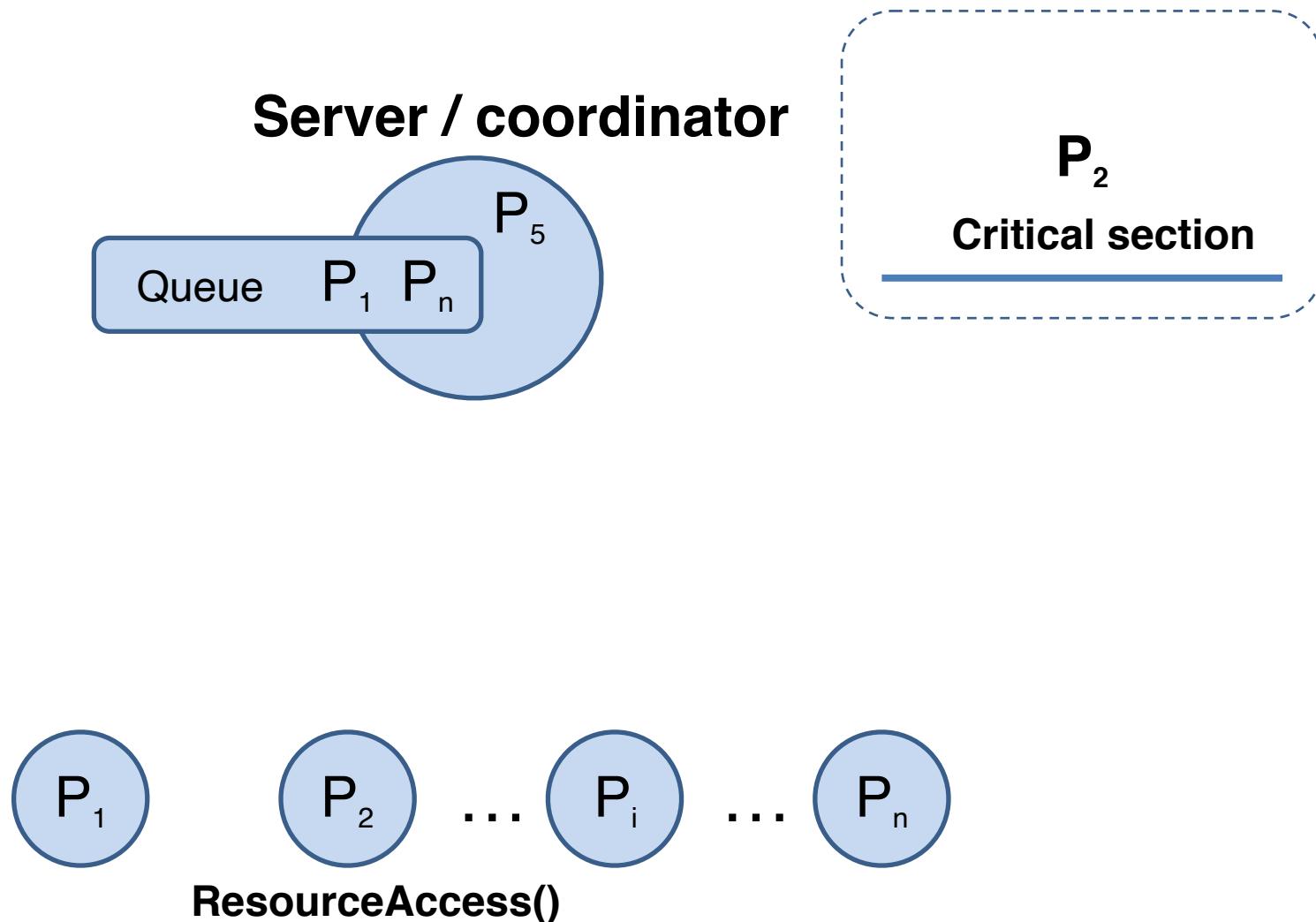
Centralized strategy: Non-empty CS



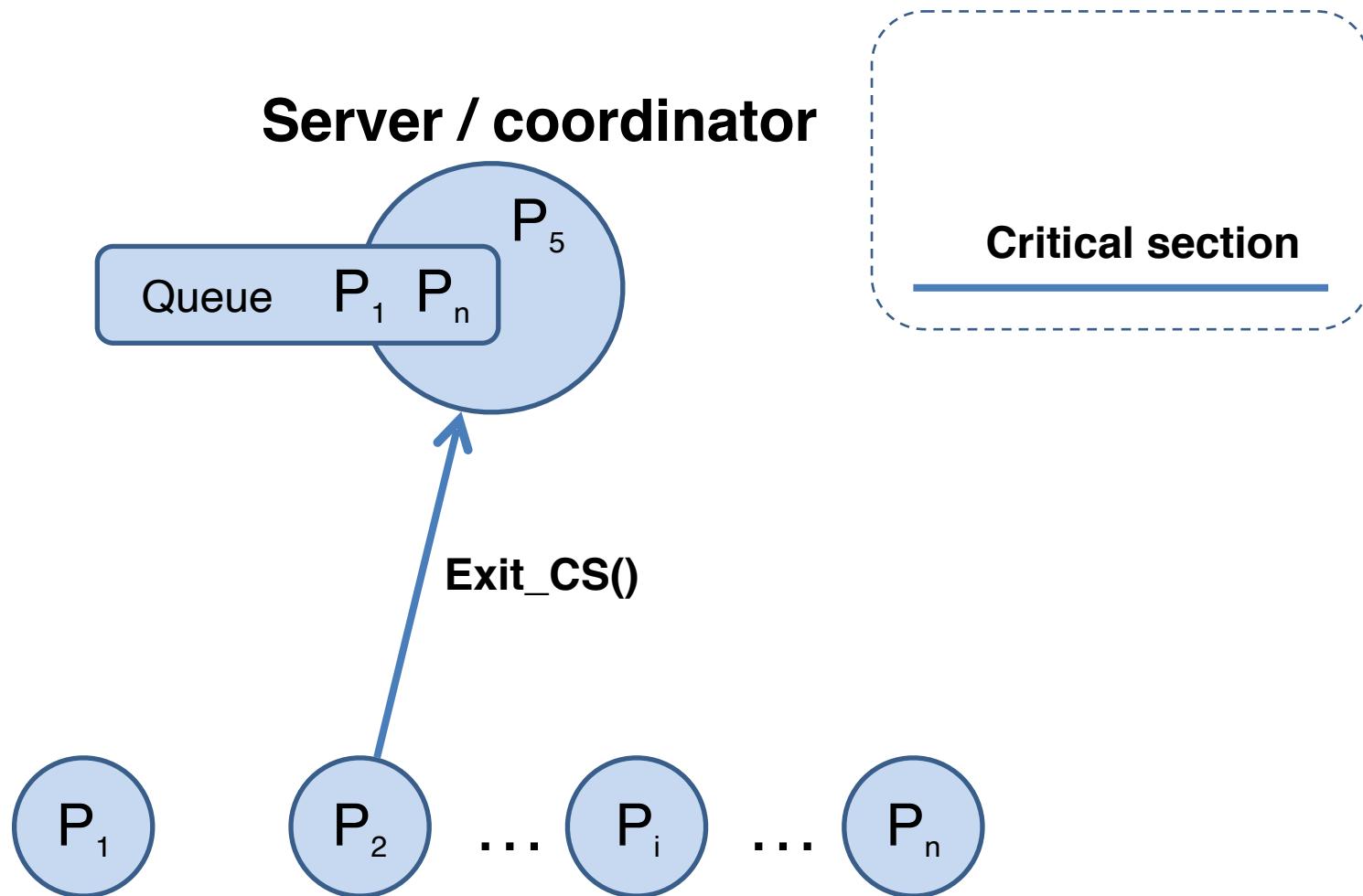
Centralized strategy: Non-empty CS



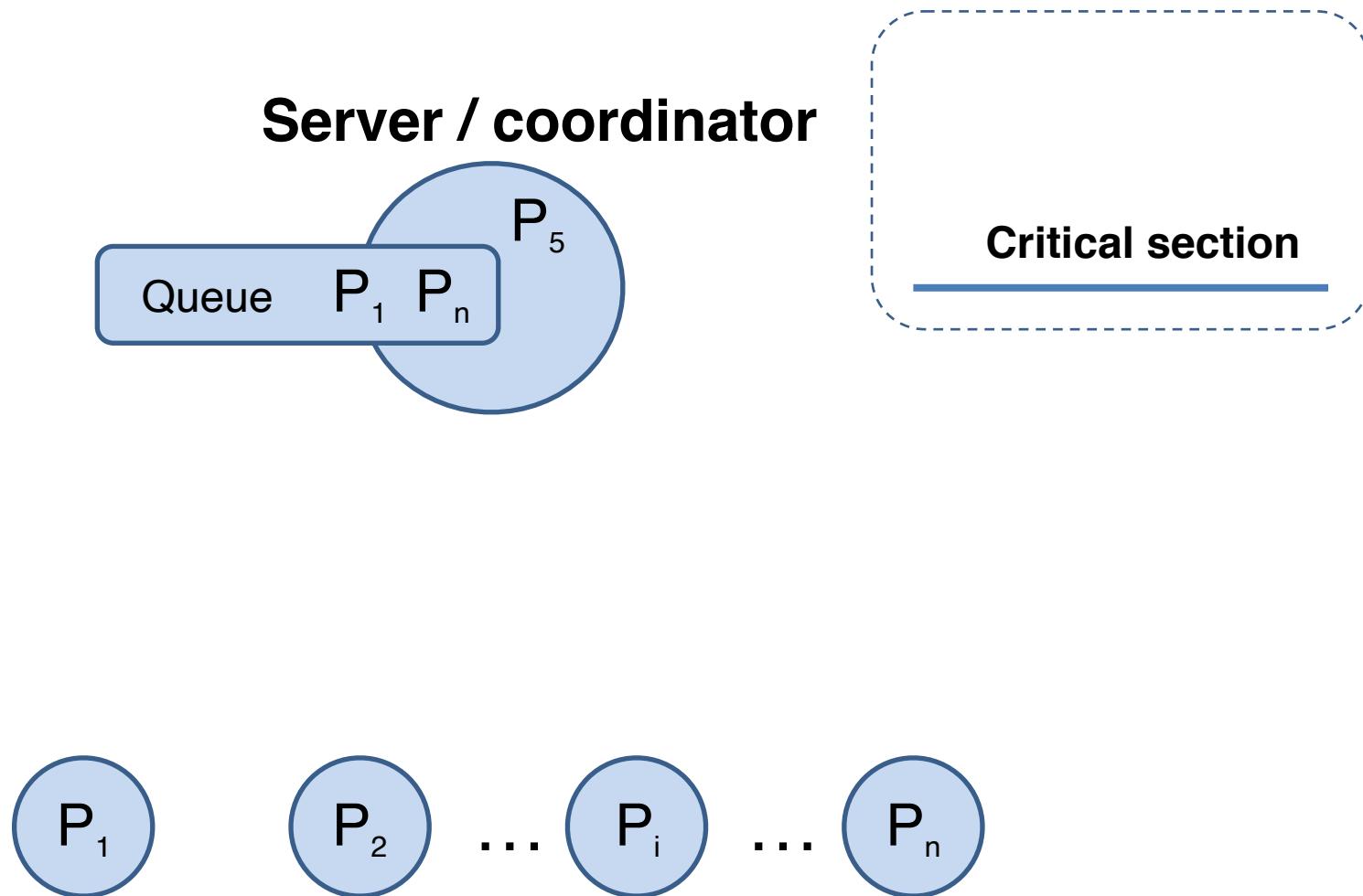
Centralized strategy: Exit CS



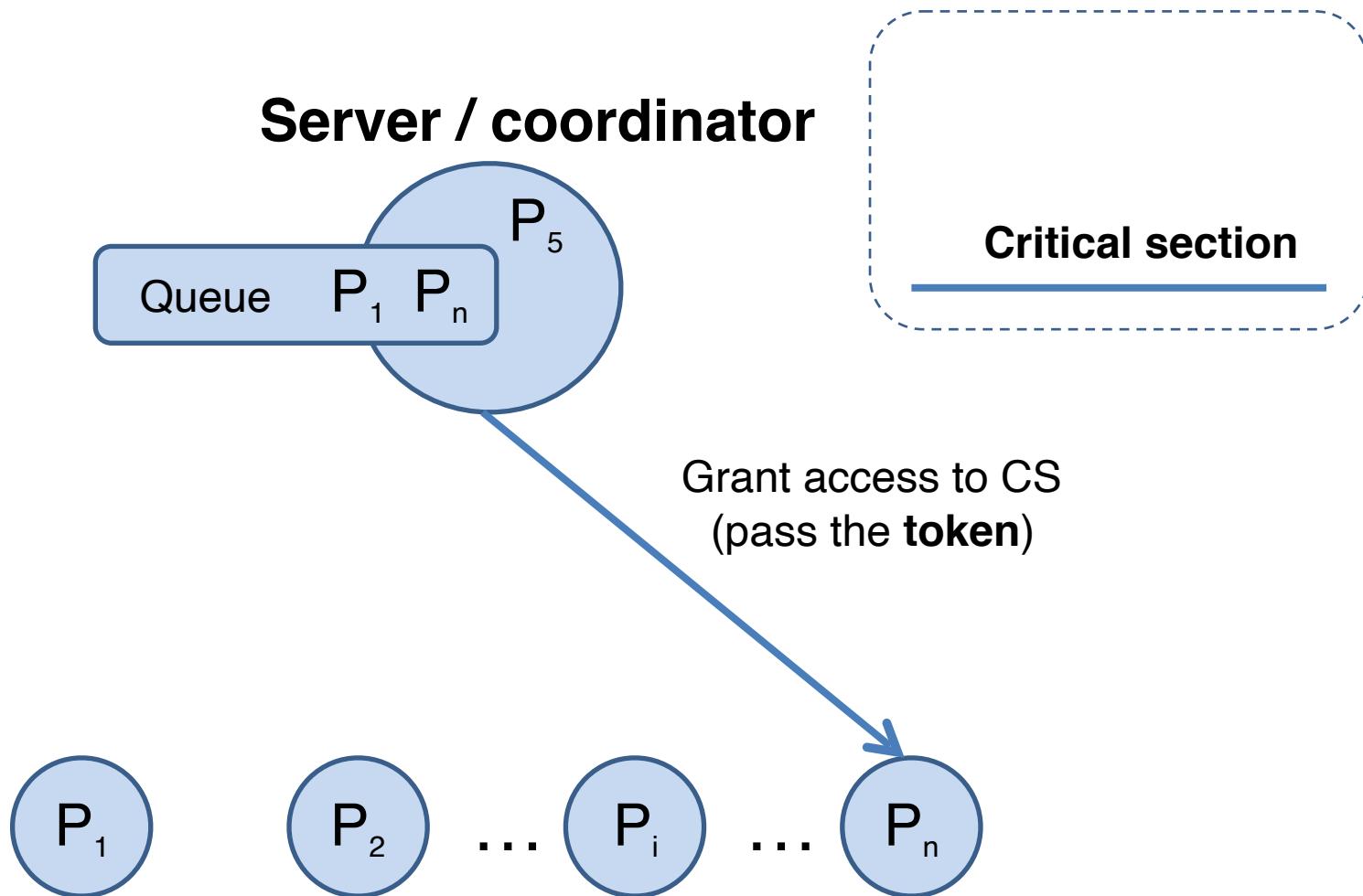
Centralized strategy: Exit CS



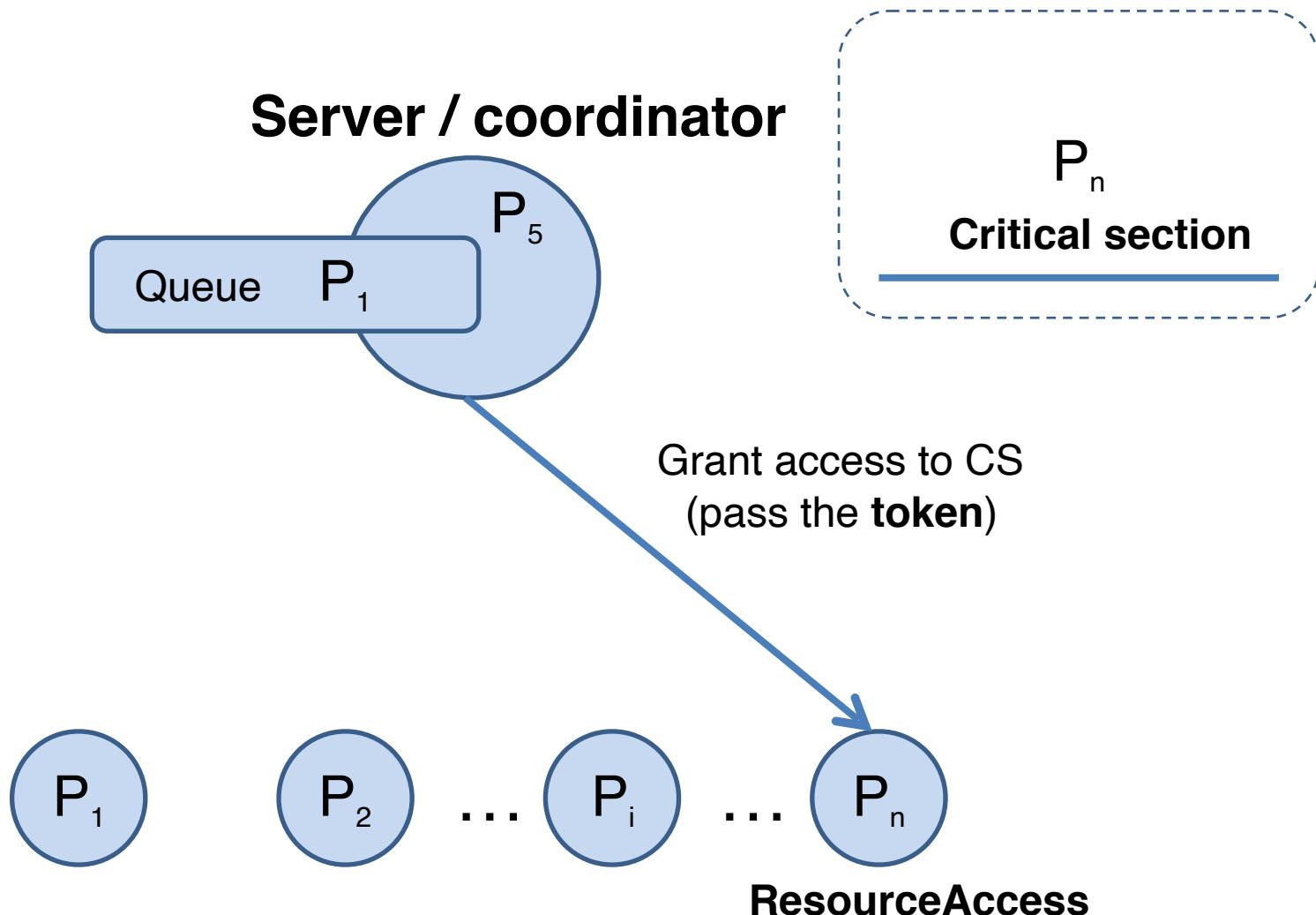
Centralized strategy: Exit CS



Centralized strategy: Exit CS



Centralized strategy: Exit CS



Centralized strategy analysis I

- Meets requirements: Safety, liveness, no starvation
- ***Does solution meet the ordering requirement?***
- Advantages
 - **Simple to implement**
- Disadvantages
 - **Single point of failure**
 - Bottleneck, network congestion, timeout
- Deadlock potential for multiple resources with separate servers

Centralized strategy analysis II

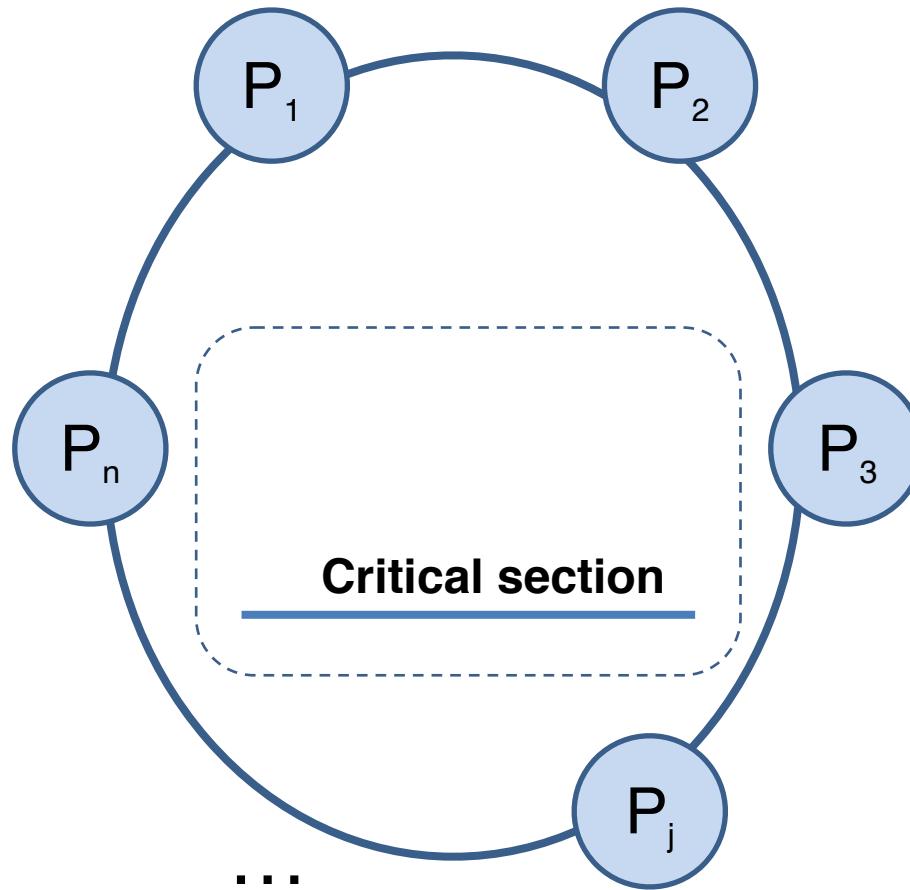
- Enter_CS()
 - **Two messages:** Request & Grant
 - One round of communication (RTT delay)
- Exit _CS()
 - **One message:** Release message
 - No delay for the process in CS

Distributed strategy

- In our distributed strategies, **the same decision made on each node**, independent of the other nodes in the system.
- Selected algorithms
 - Ring-based algorithm
 - **Logical clocks-based algorithm (Lamport, 1976)**
 - Ricart & Agrawala, 1981
 - Maekawa, 1985
 - Many more

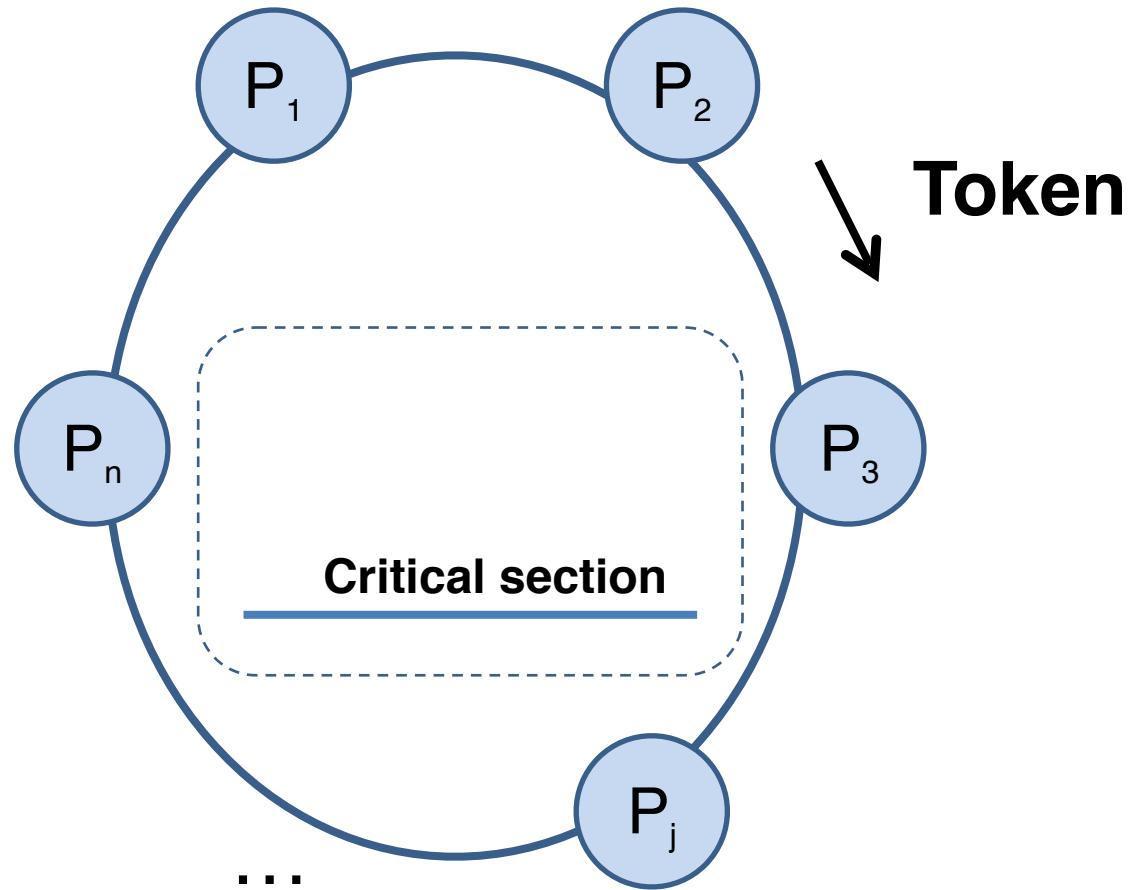
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



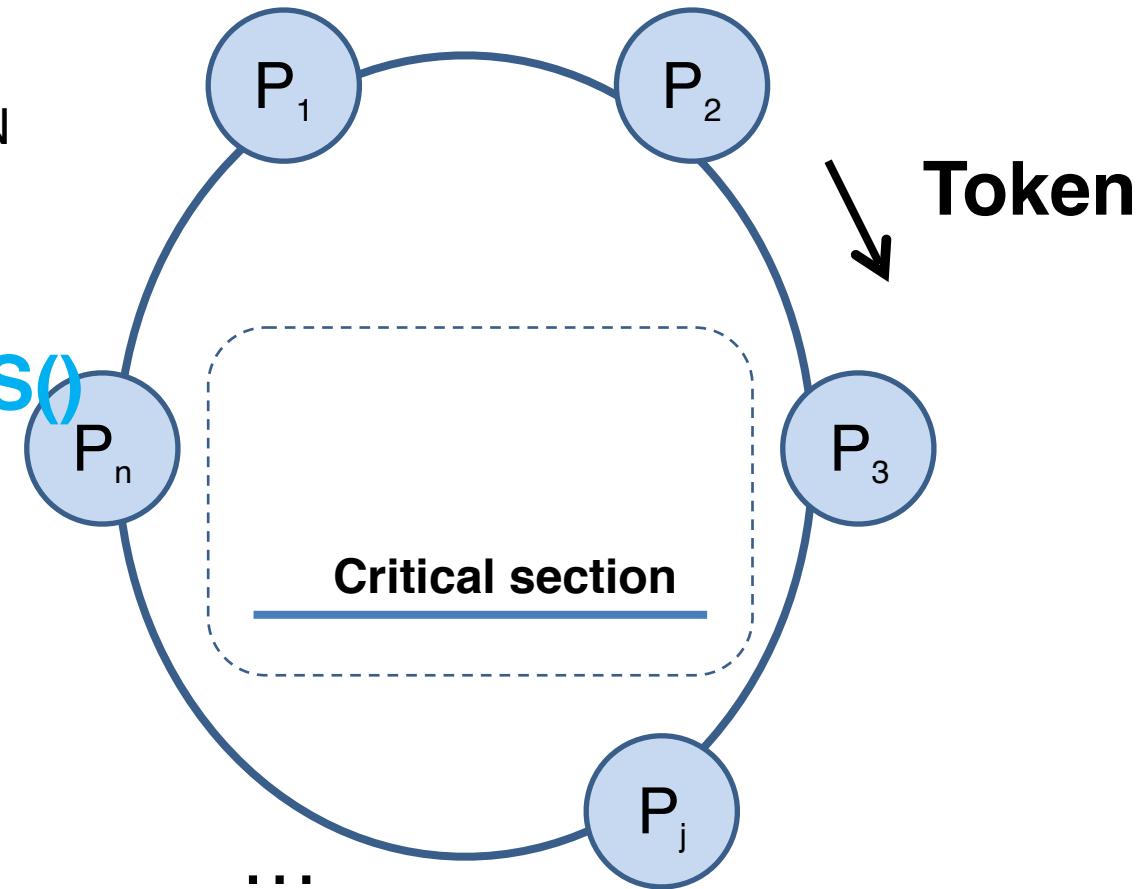
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical topology



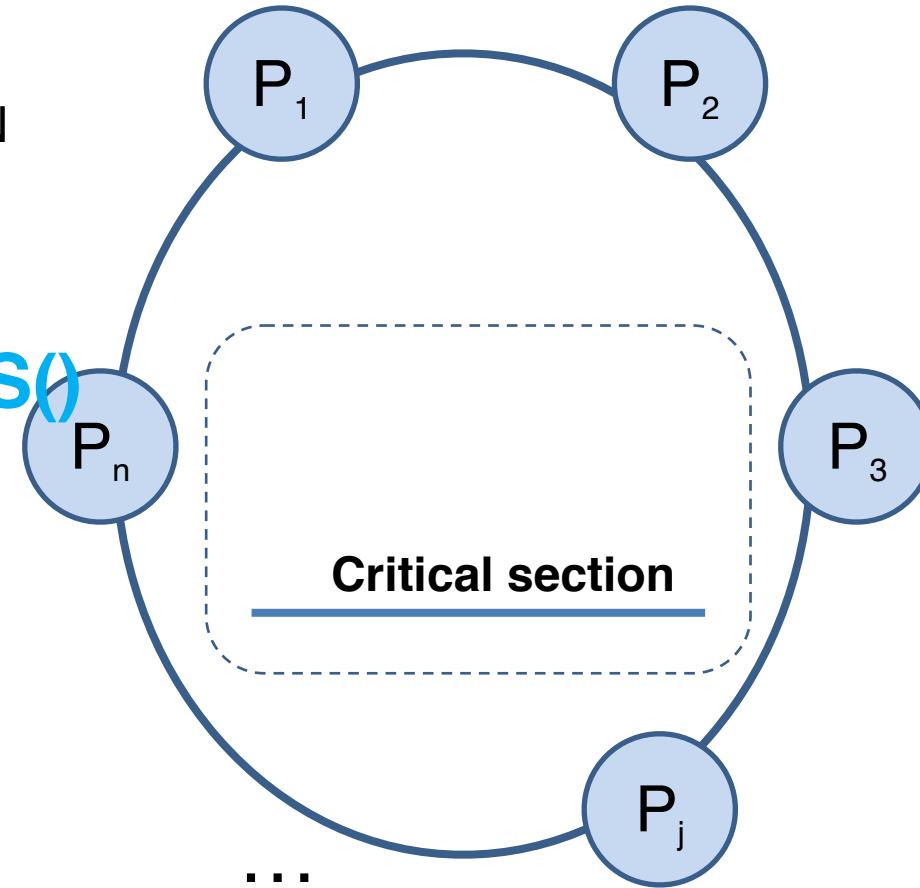
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical **Enter_CS()** topology



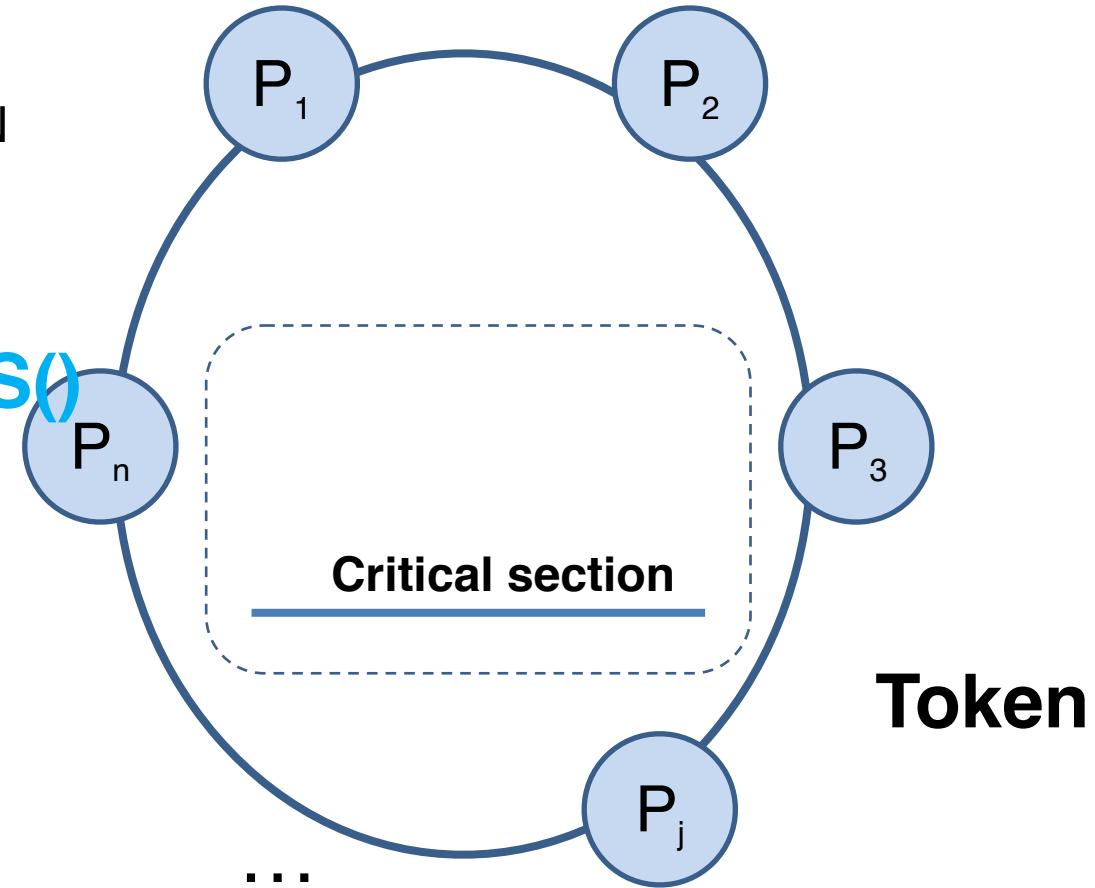
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical `Enter_CS()` topology



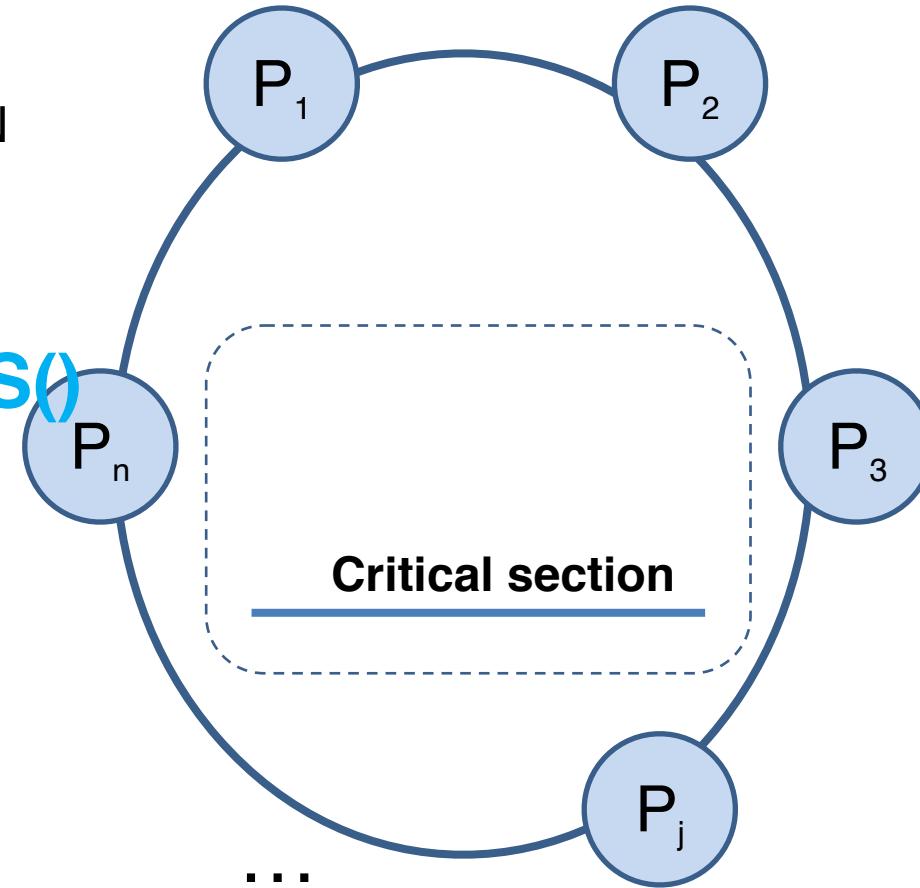
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical **Enter_CS()** topology



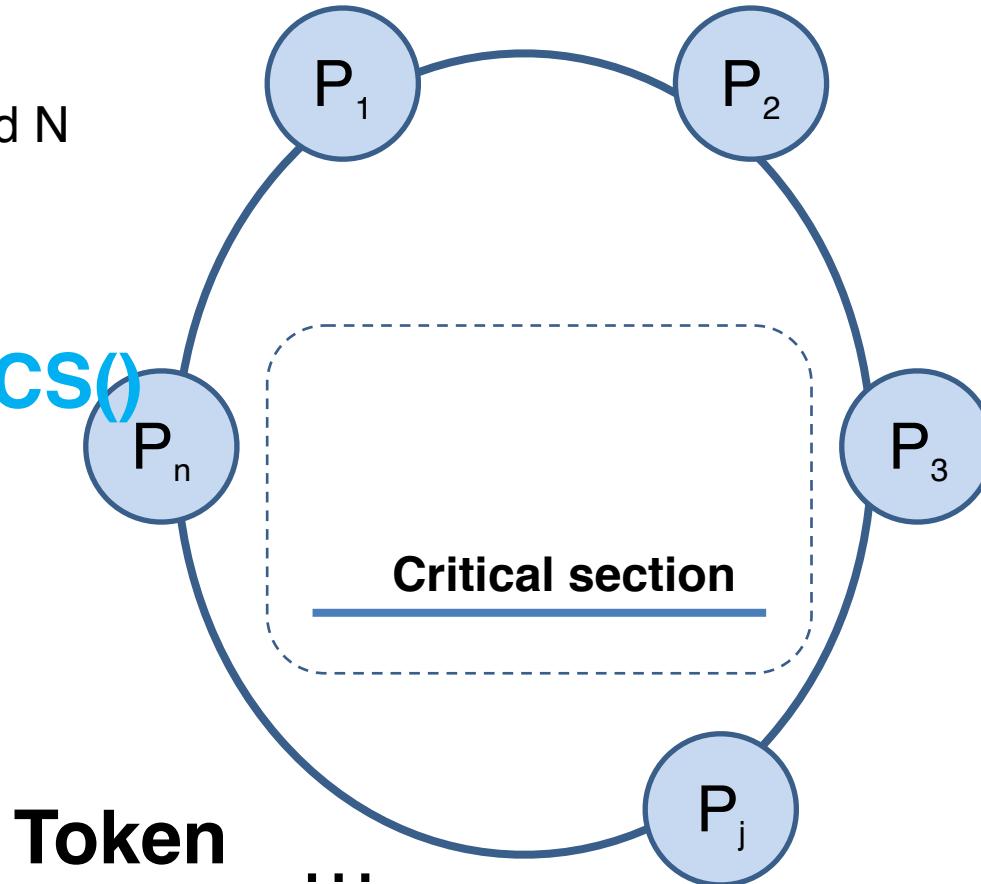
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical **Enter_CS()** topology



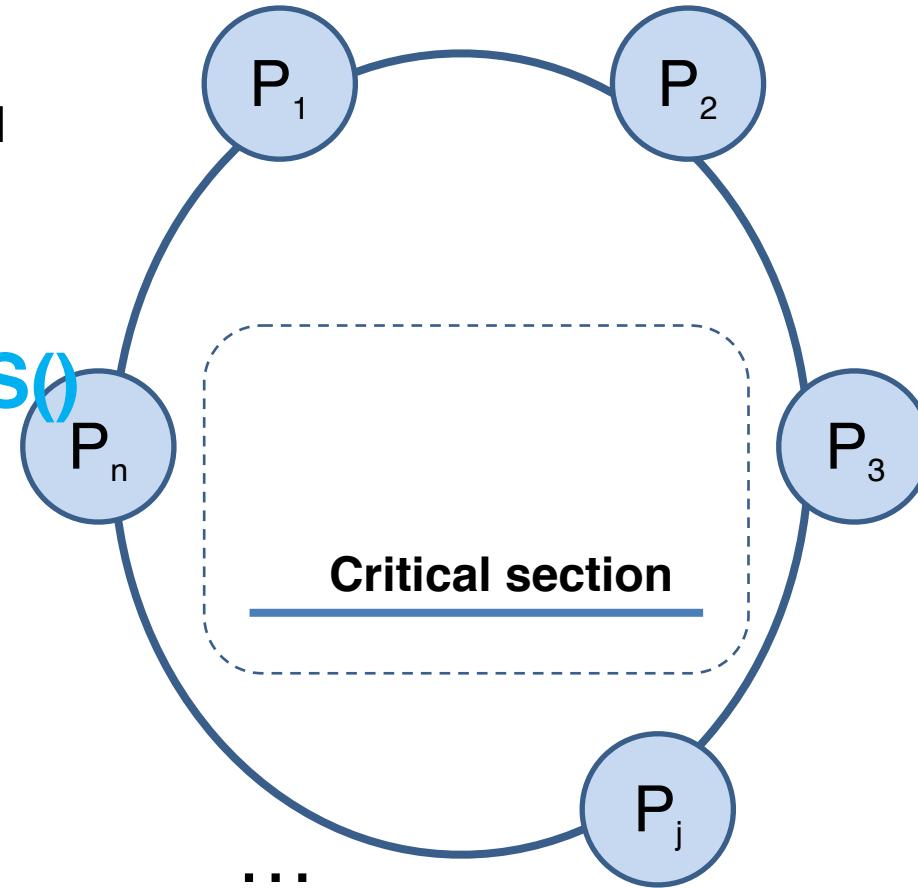
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical **Enter_CS()** topology



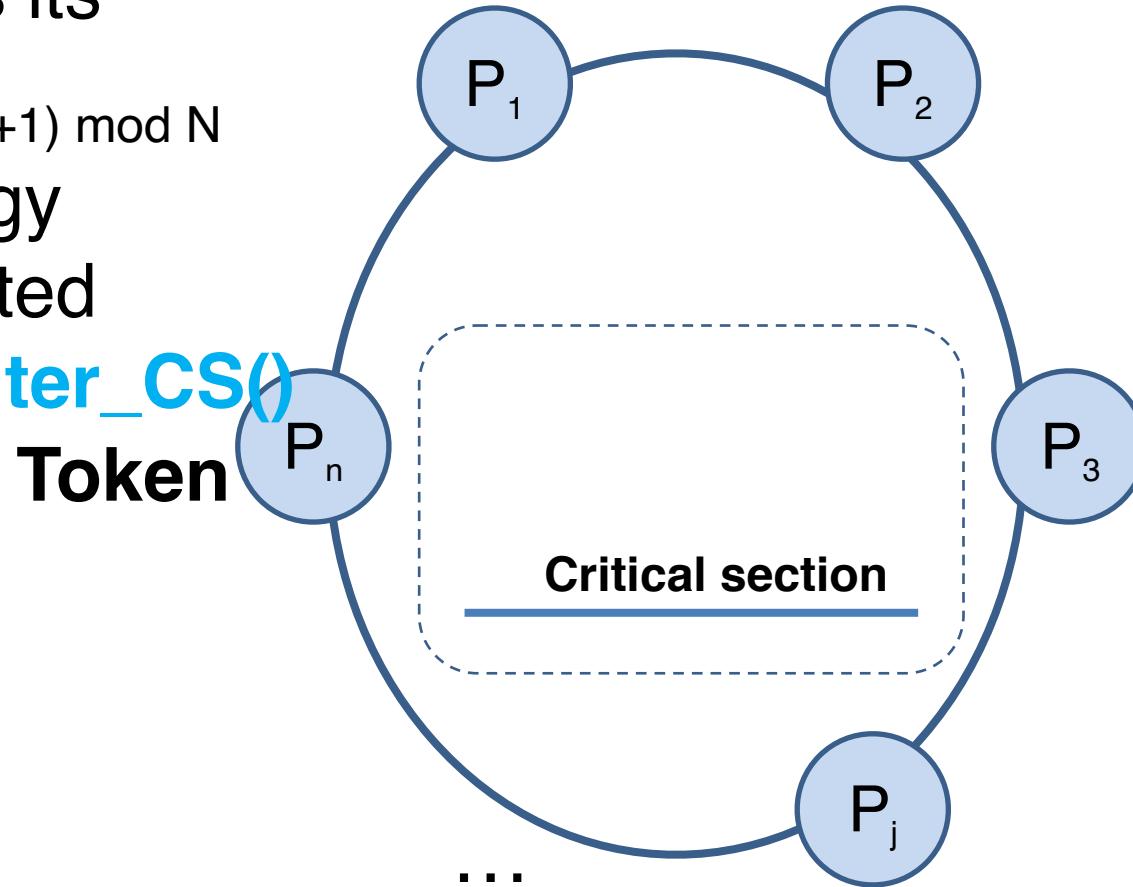
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical **Enter_CS()** topology



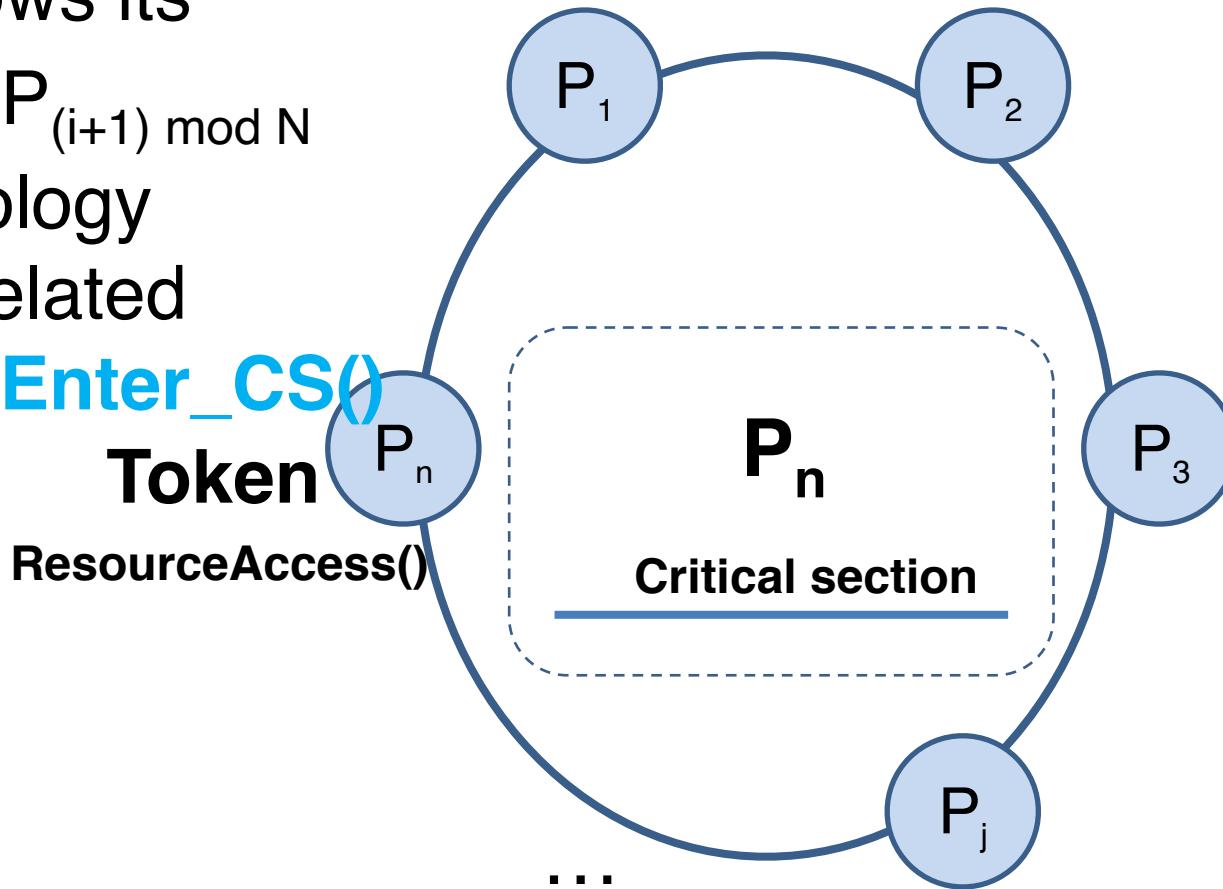
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical **Enter_CS()** topology



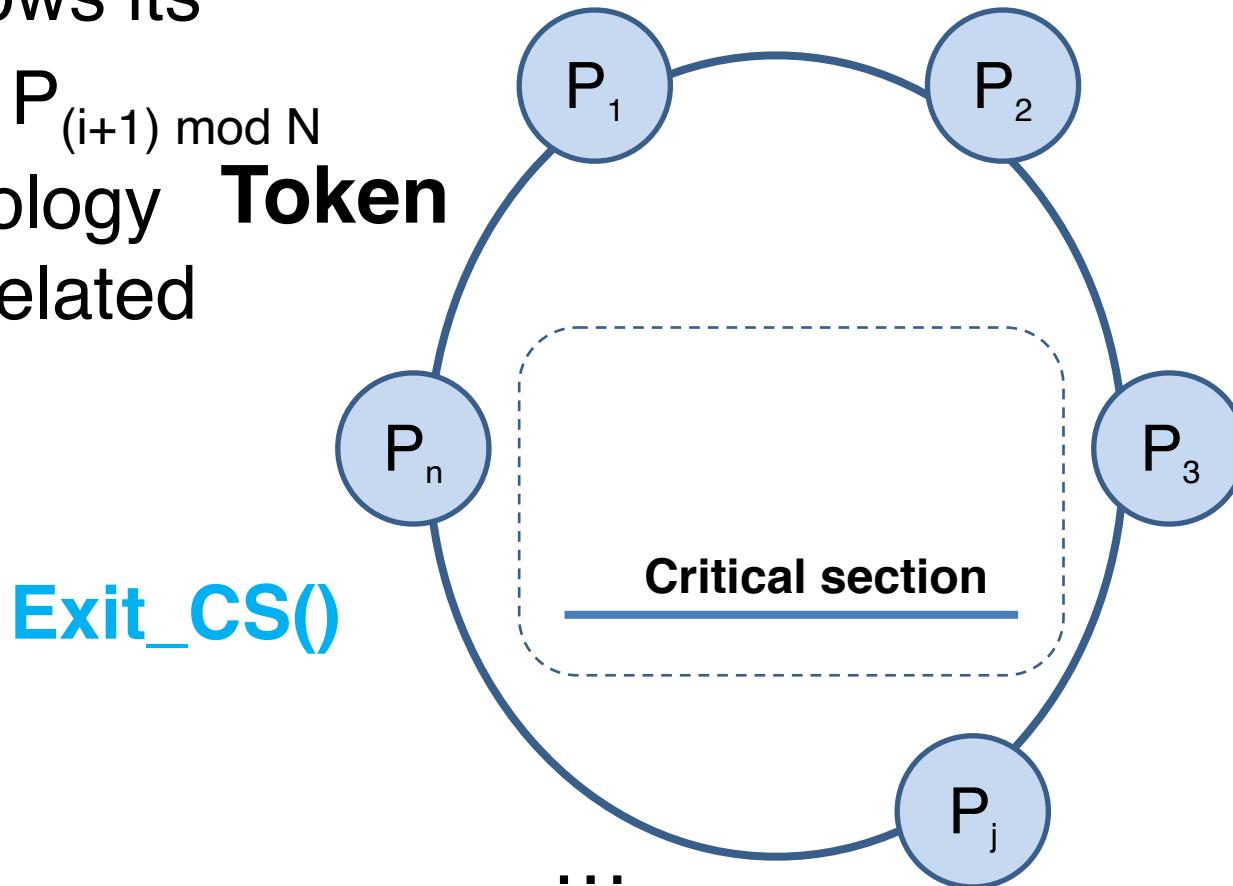
Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology a priori unrelated to physical **Enter_CS()** topology



Ring-based algorithm

- Logical ring of processes
- Each P_i knows its successor, $P_{(i+1) \bmod N}$
- Logical topology **Token** a priori unrelated to physical topology



Ring-based algorithm analysis

- **Safe:** Node enters CS only if it holds the token
- **Live:** Since finite work is done by each node (can't re-enter), the token eventually gets to each node
- **Fair:** Ordering is based on ring topology, no starvation (pass token between accesses)
- **Performance**
 - **Constantly consumes network bandwidth**, even when no processes seek entry, except when inside the CS
 - **Synchronization delay:** Between 1 and N messages
 - **Client delay:** Between 0 and N messages; 0 for exit (asynchronous)

Potential problems with ring-based algorithm

(Due to our assumption, do not apply here.)

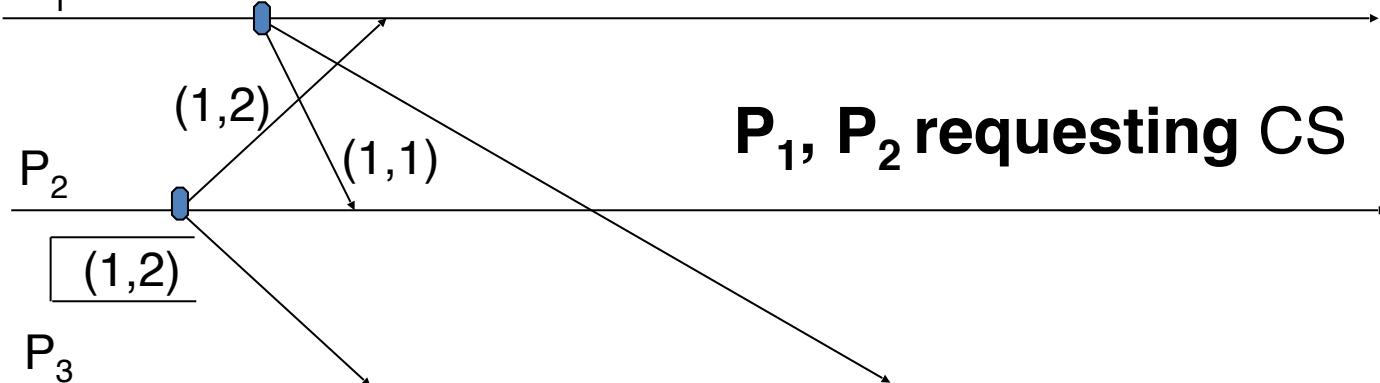
- Lost token
- Duplicate token
- Timeouts on token passing

Lamport's algorithm

- System of n processes
- (ts_i, i) – logical clock timestamp of process P_i
- Non-token based approach uses logical timestamps to order requests for CS
- Smaller timestamps have priority over larger ones
- Request queue maintained at each process ordered by timestamp (**a priority queue!**)
- Assume message delivered in FIFO order

Lamport's algorithm

- P_i requesting CS
 - Broadcast **REQUEST(ts_i, i)** message to all processes
 - Place request in *request_queue_i*,
 - (ts_i, i) denotes timestamp of request
- When P_k receives a REQUEST(ts_i, i) message from P_i ,
 - It places P_i 's request into *request_queue_k*
 - Sends a timestamped REPLY message to P_i



Lamport's algorithm

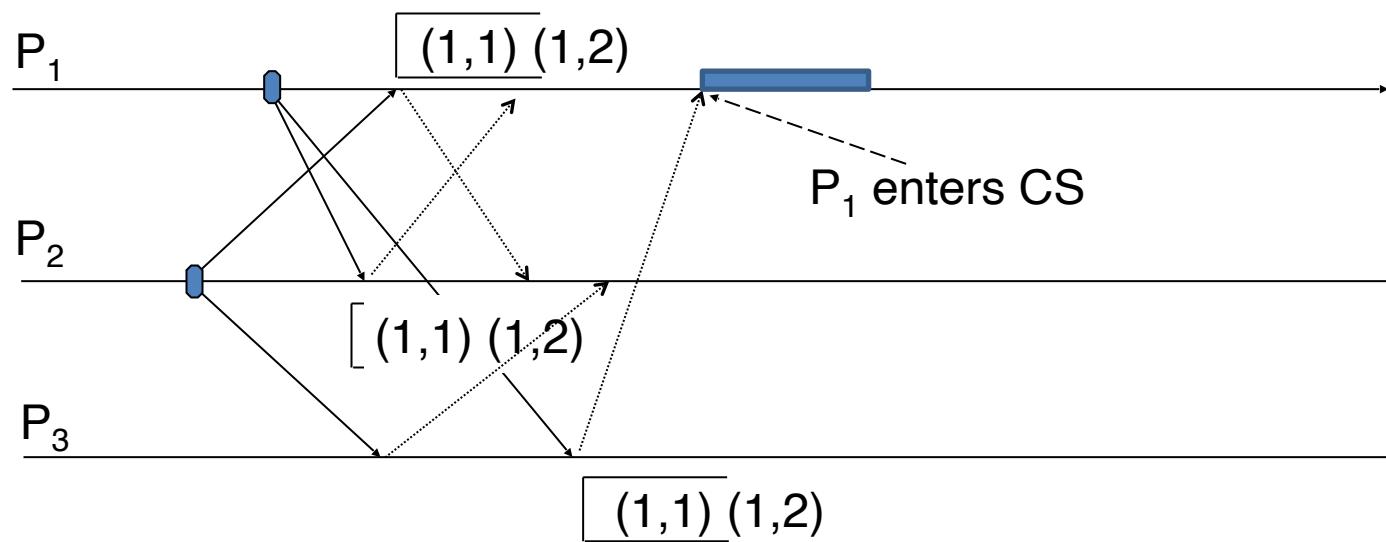
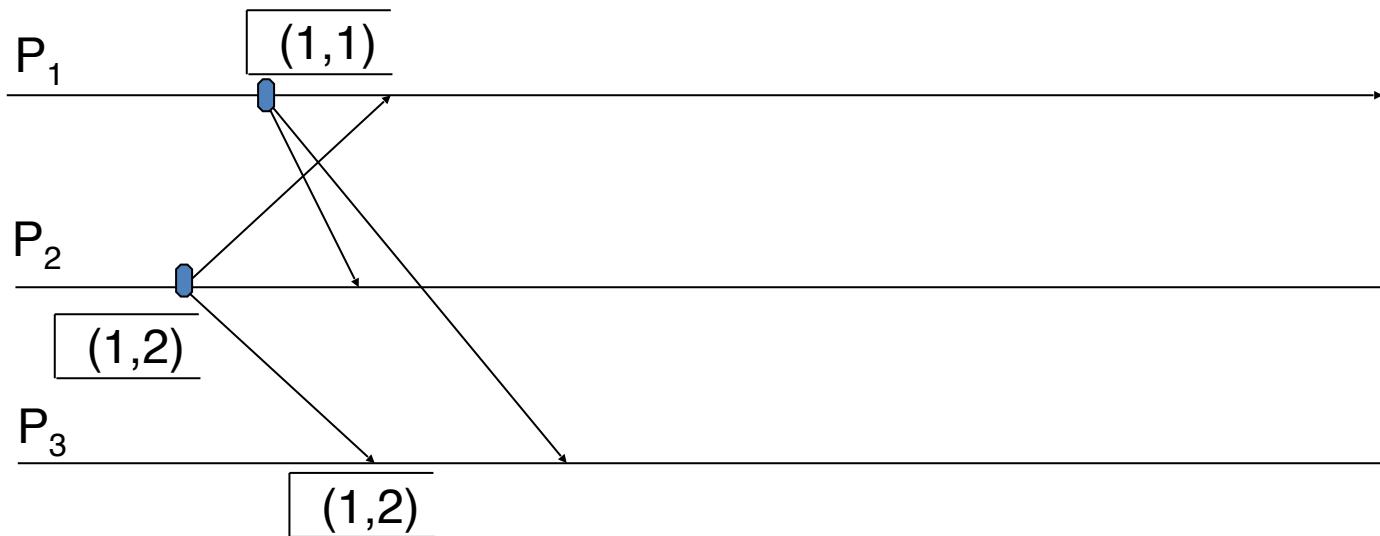
P_i requesting CS – P_i enters when following conditions hold

1. P_i has received a message with timestamp larger than (ts_i, i) from every other processes
2. P_i 's request is at top of *request_queue_i*,

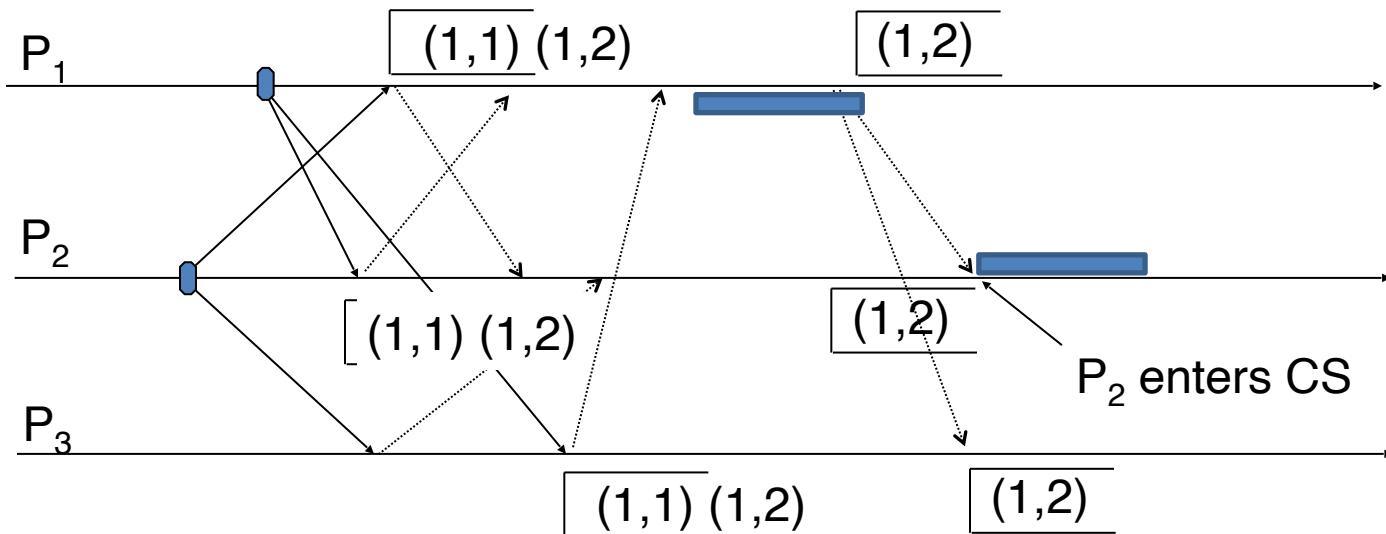
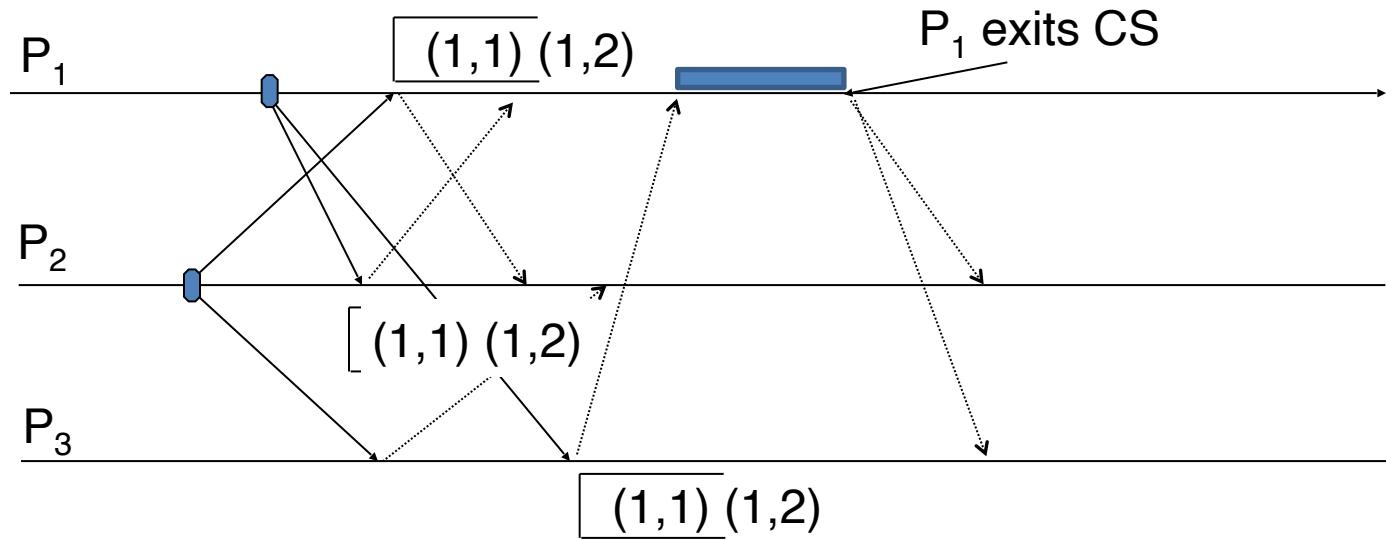
P_i releasing CS

- Removes request from top of the request queue
- Broadcasts a **RELEASE** message to all processes
- When P_k receives a **RELEASE** message from P_i , P_k removes P_i 's request from its request queue

Example



Example...



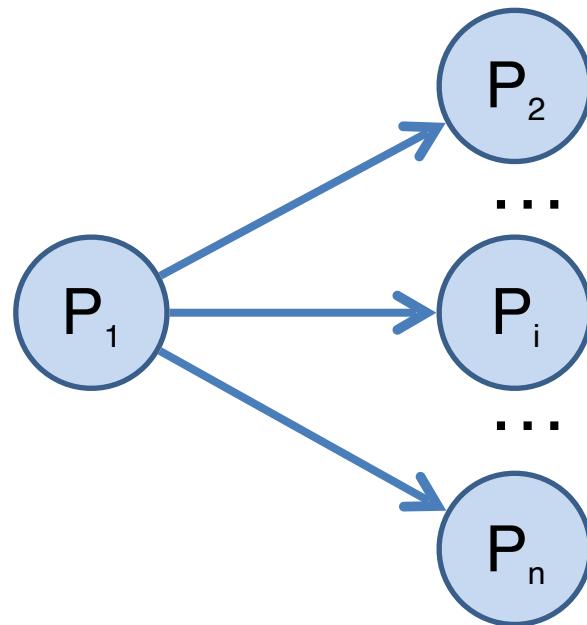
Performance

- $3(N-1)$ messages per CS invocation
 - $(N - 1)$ REQUEST
 - $(N - 1)$ REPLY
 - $(N - 1)$ RELEASE messages

Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among n processes)

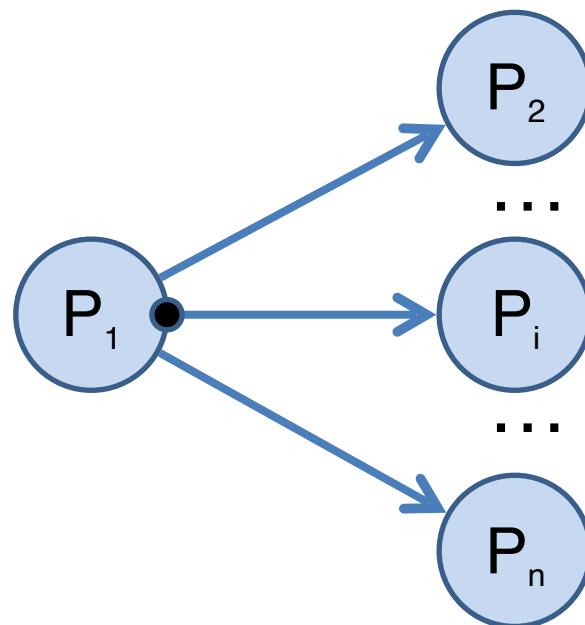
- Basic idea
 - Processes wanting to enter CS, **broadcast a request to all processes**
 - Enter CS, once **all have granted request**
- Use **Lamport timestamps** to order requests: $\langle T, P_i \rangle$, T the timestamp, P_i the process identifier



Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among n processes)

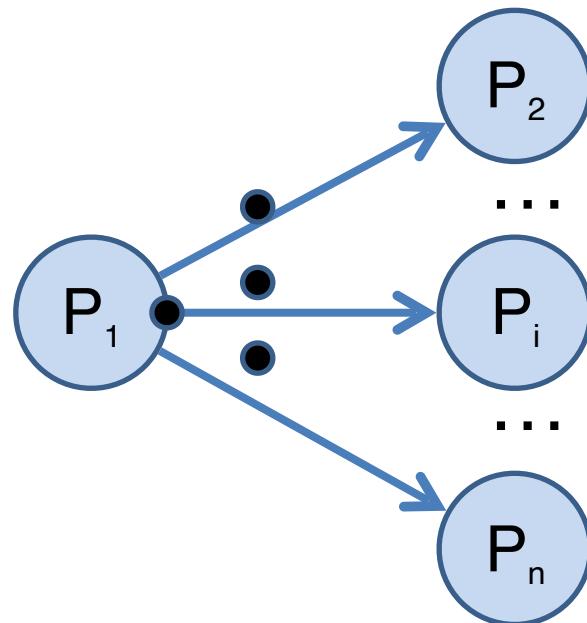
- **Basic idea**
 - Processes wanting to enter CS, **broadcast a request to all processes**
 - Enter CS, once **all have granted request**
- Use **Lamport timestamps** to order requests: $\langle T, P_i \rangle$, T the timestamp, P_i the process identifier



Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among n processes)

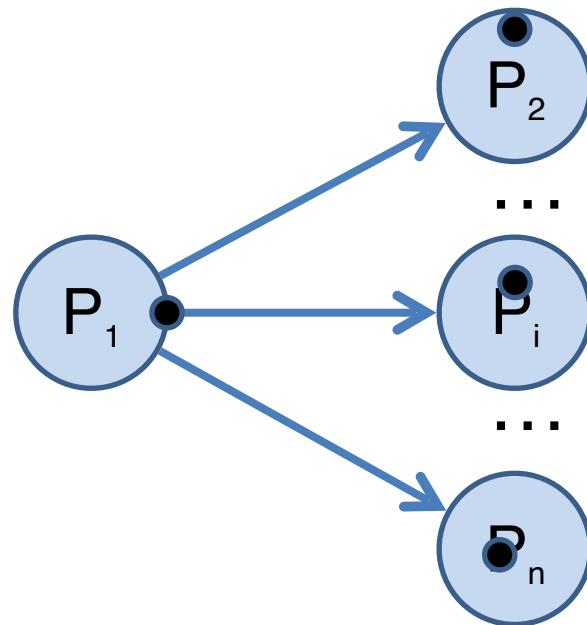
- **Basic idea**
 - Processes wanting to enter CS, **broadcast a request to all processes**
 - Enter CS, once **all have granted request**
- Use **Lamport timestamps** to order requests: $\langle T, P_i \rangle$, T the timestamp, P_i the process identifier



Ricart & Agrawala, 1981, algorithm

(Guarantees mutual exclusion among n processes)

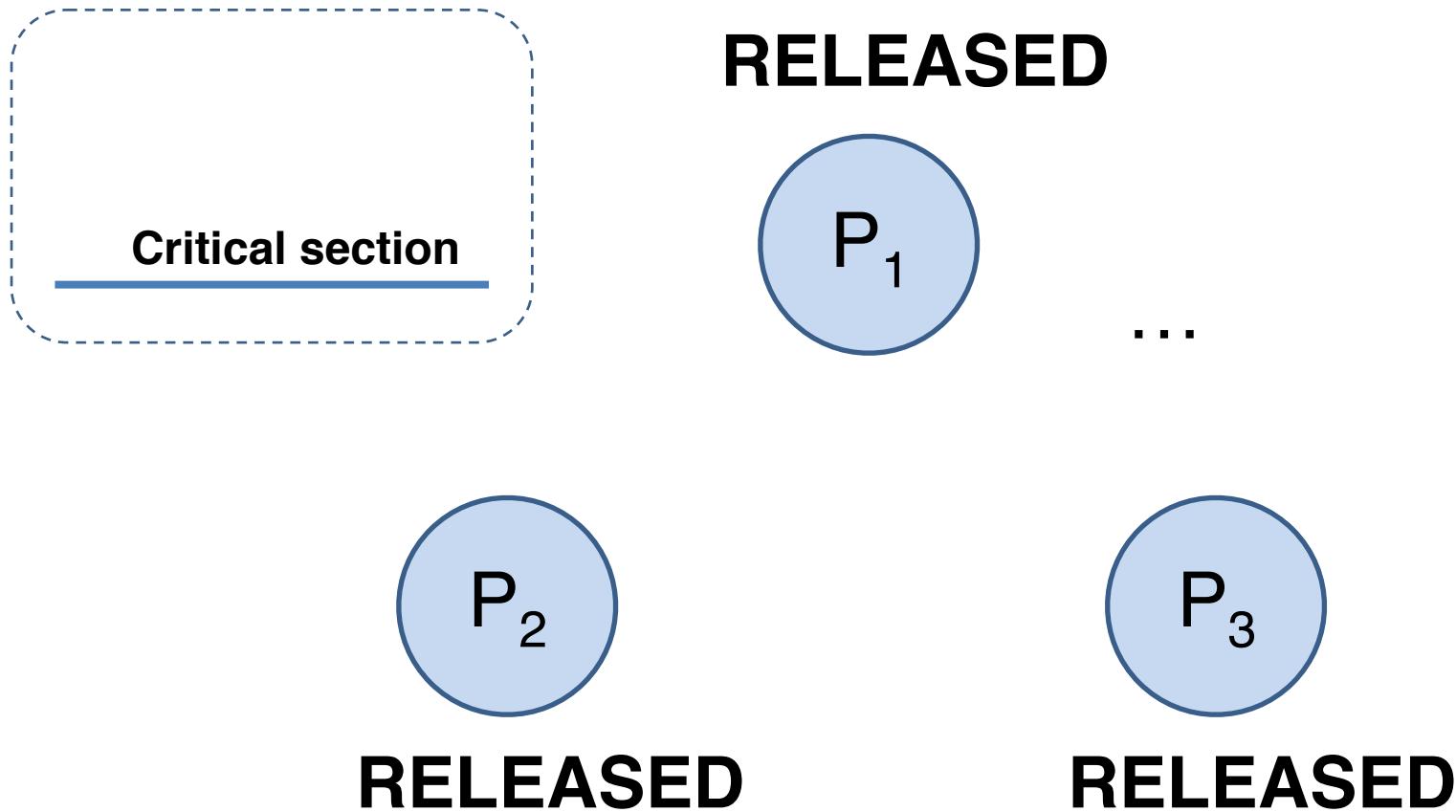
- **Basic idea**
 - Processes wanting to enter CS, **broadcast a request to all processes**
 - Enter CS, once **all have granted request**
- Use **Lamport timestamps** to order requests: $\langle T, P_i \rangle$, T the timestamp, P_i the process identifier



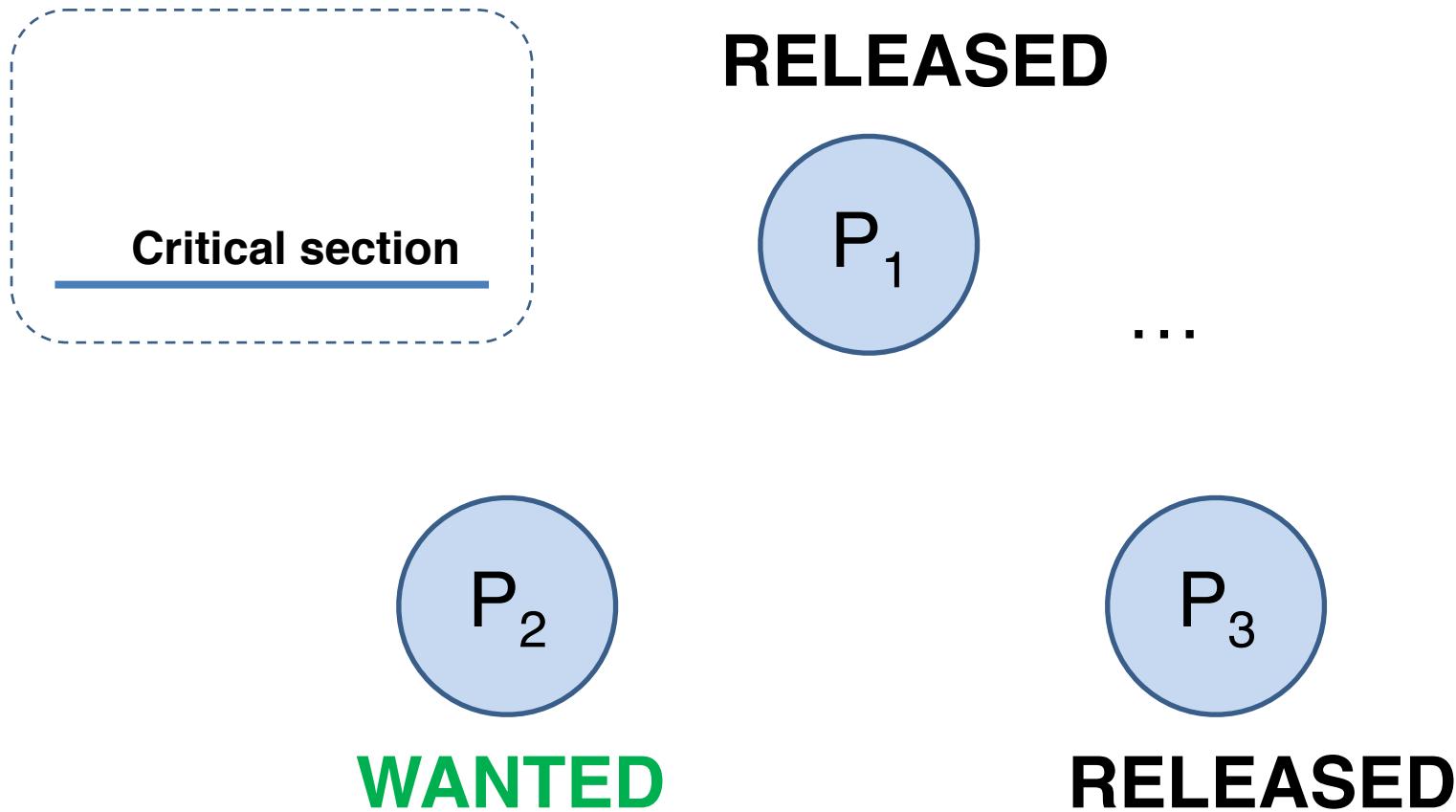
Ricart & Agrawala: Distributed strategy

- Each process is in one of three states
 - **Released** - Outside the CS, Exit_CS()
 - **Wanted** - Wanting to enter CS, Enter_CS()
 - **Held** - Inside CS, RessourceAccess()
- If a process requests to enter CS and **all other processes are in the *Released state*, entry is granted** by each process
- If a process, P_i , requests to enter CS and another process, P_k , is inside the CS (***Held state***), then P_k will not reply, until it is finished with the CS

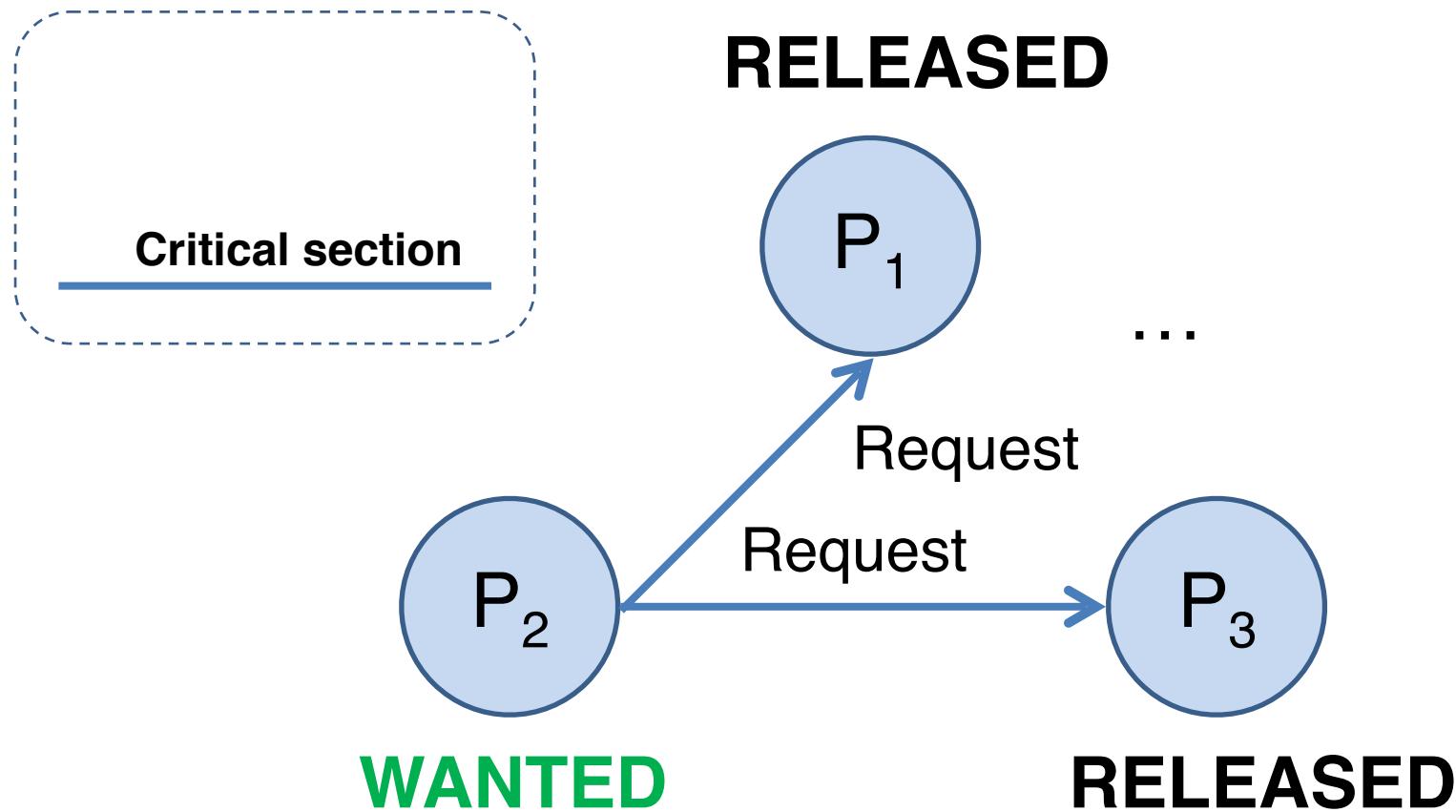
Initialization



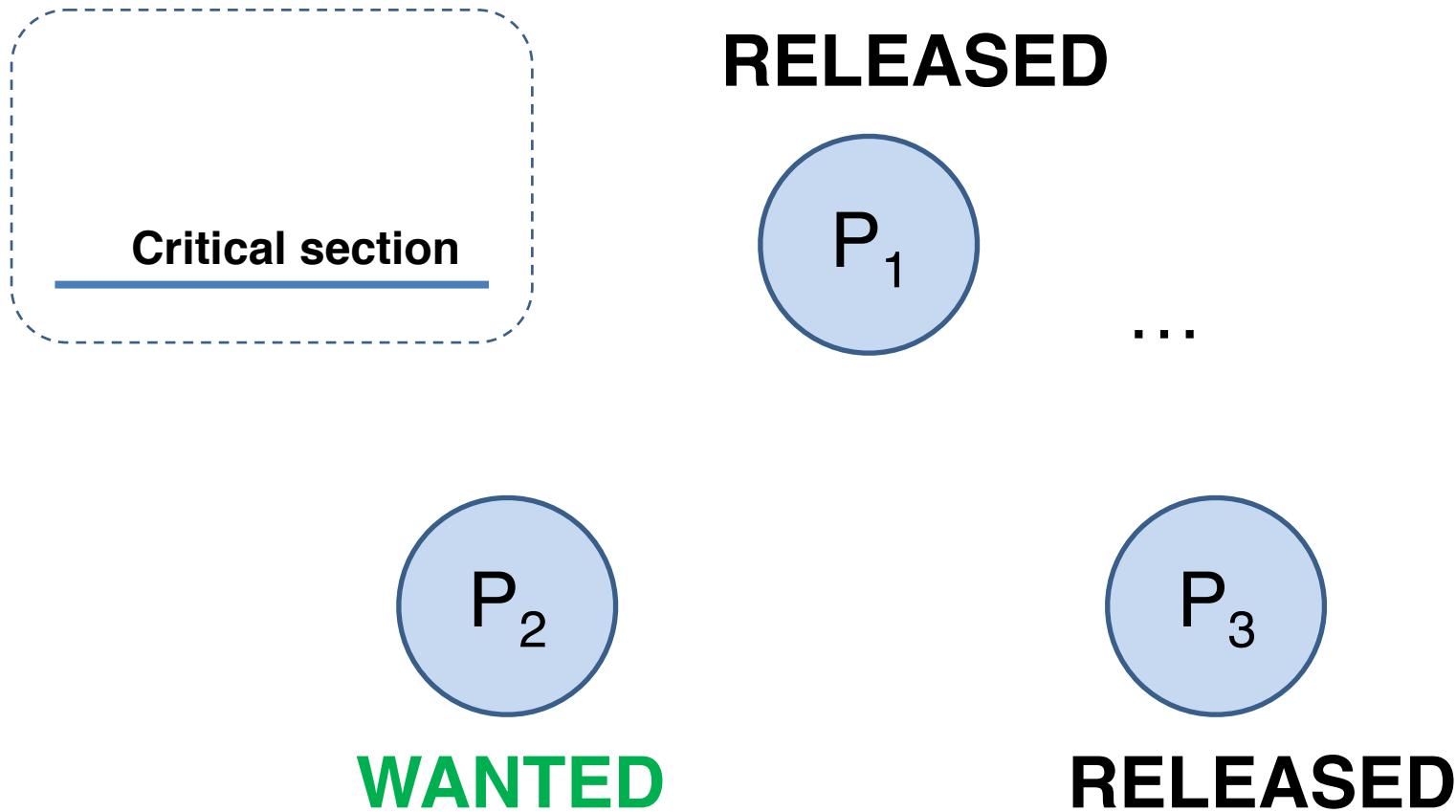
Requesting entry to CS: *All Released*



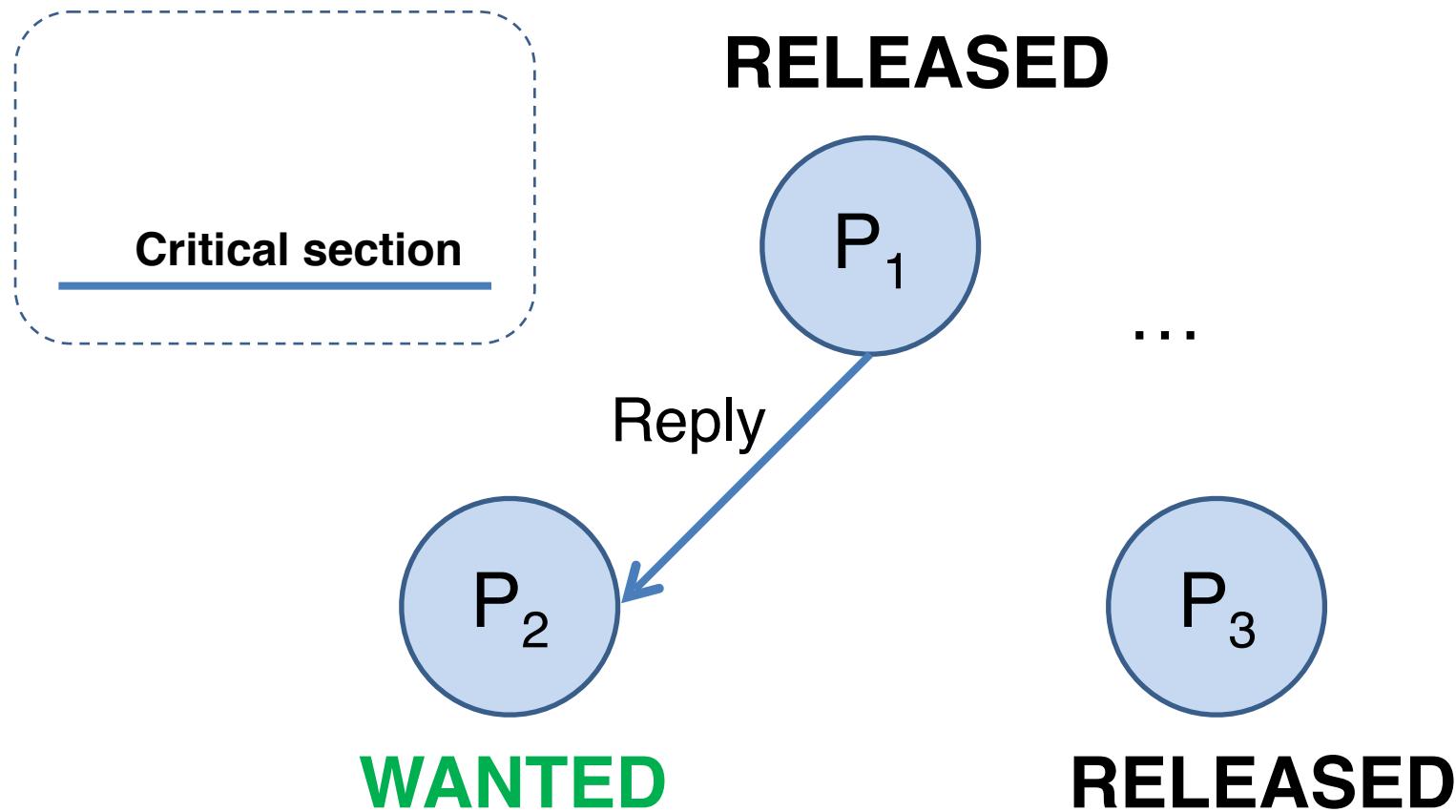
Requesting entry to CS: *All Released*



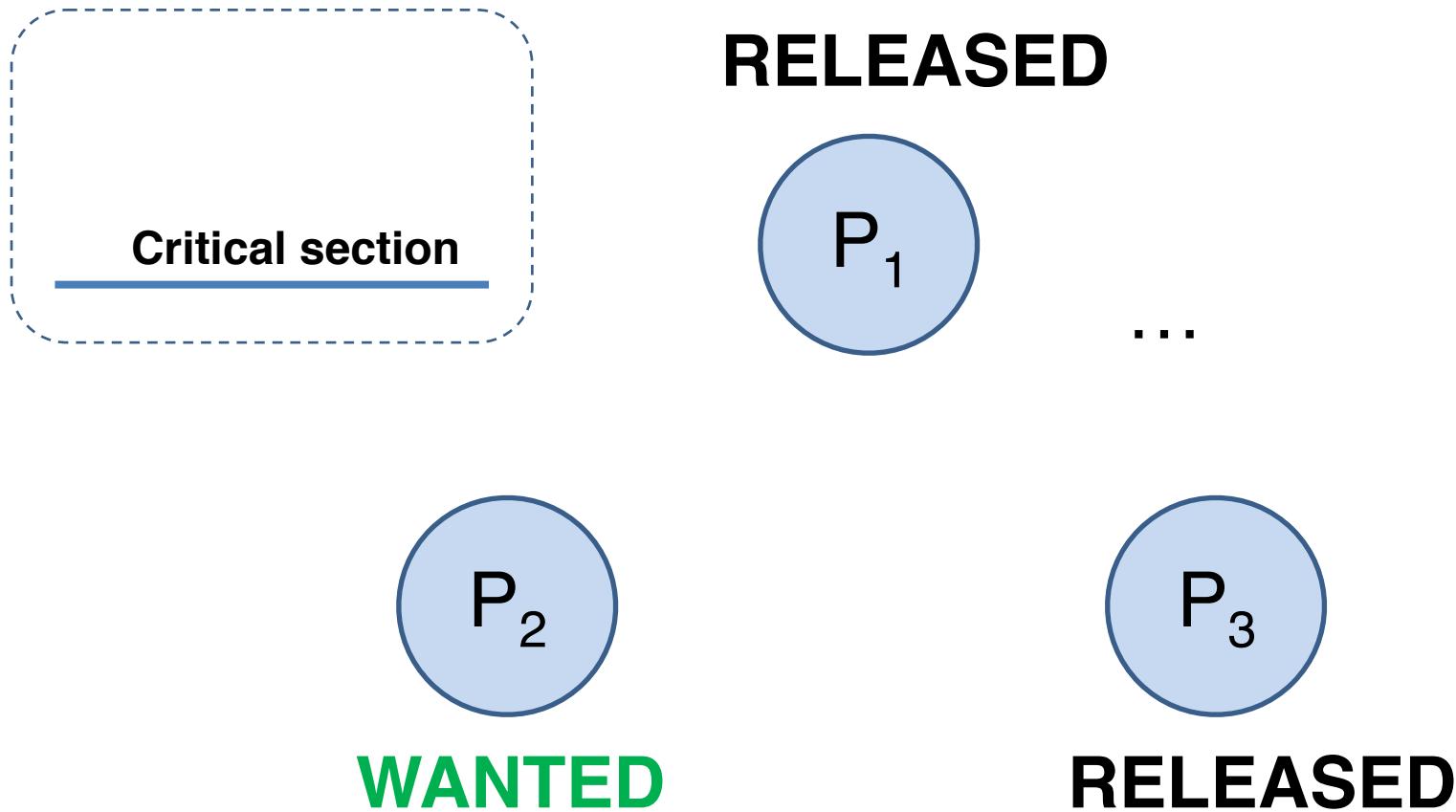
Requesting entry to CS: *All Released*



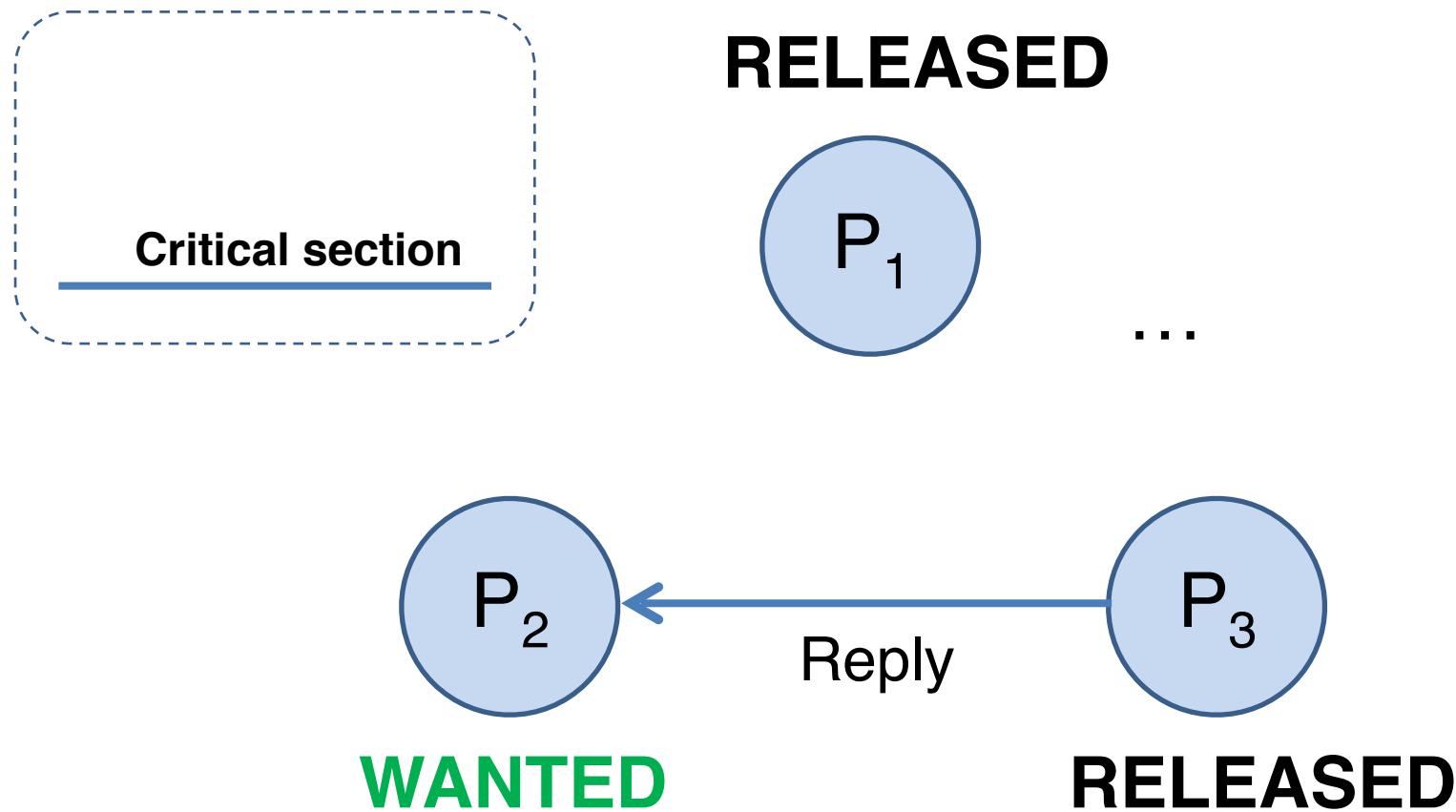
Requesting entry to CS: *All Released*



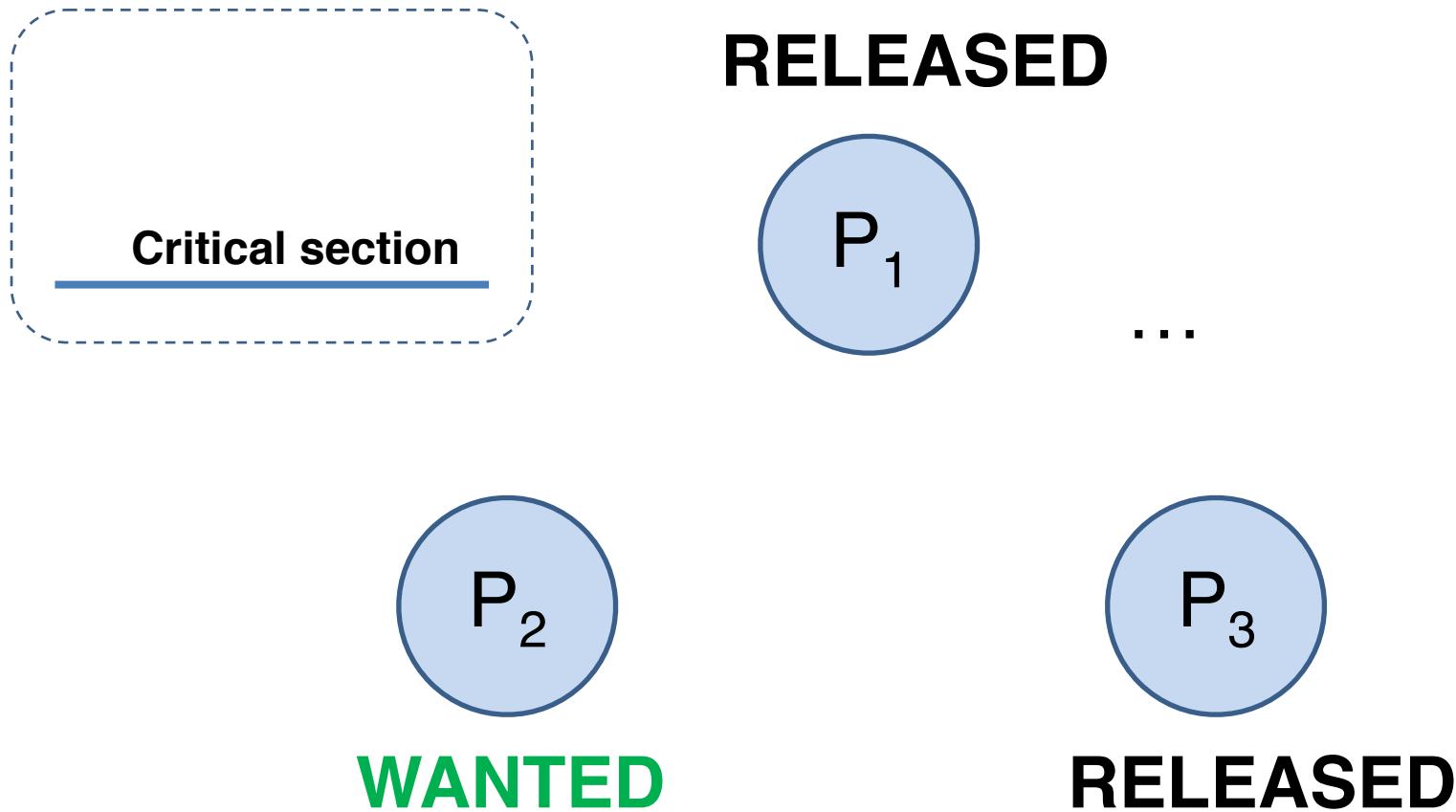
Requesting entry to CS: *All Released*



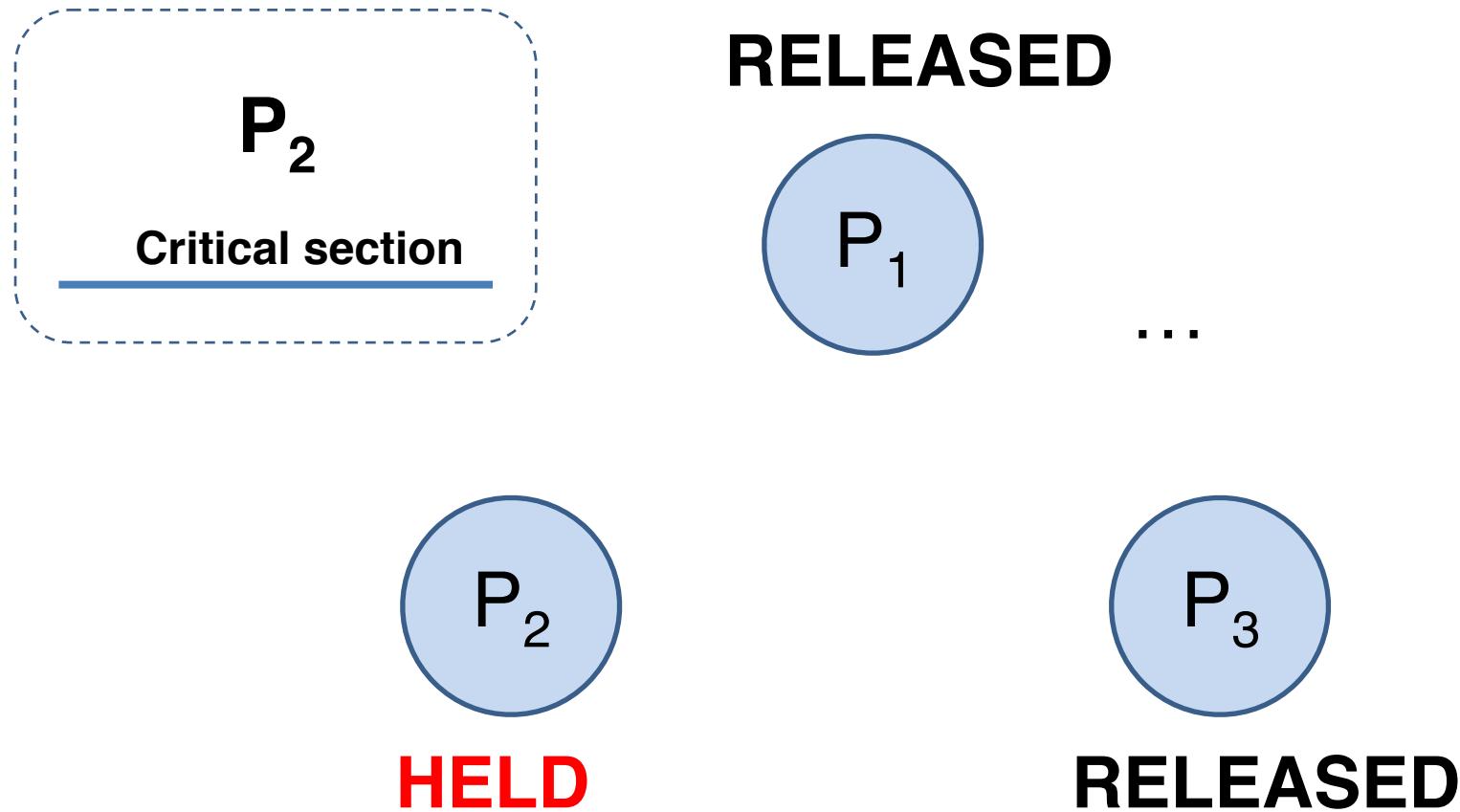
Requesting entry to CS: *All Released*



Requesting entry to CS: *All Released*



Requesting entry to CS: *All Released*



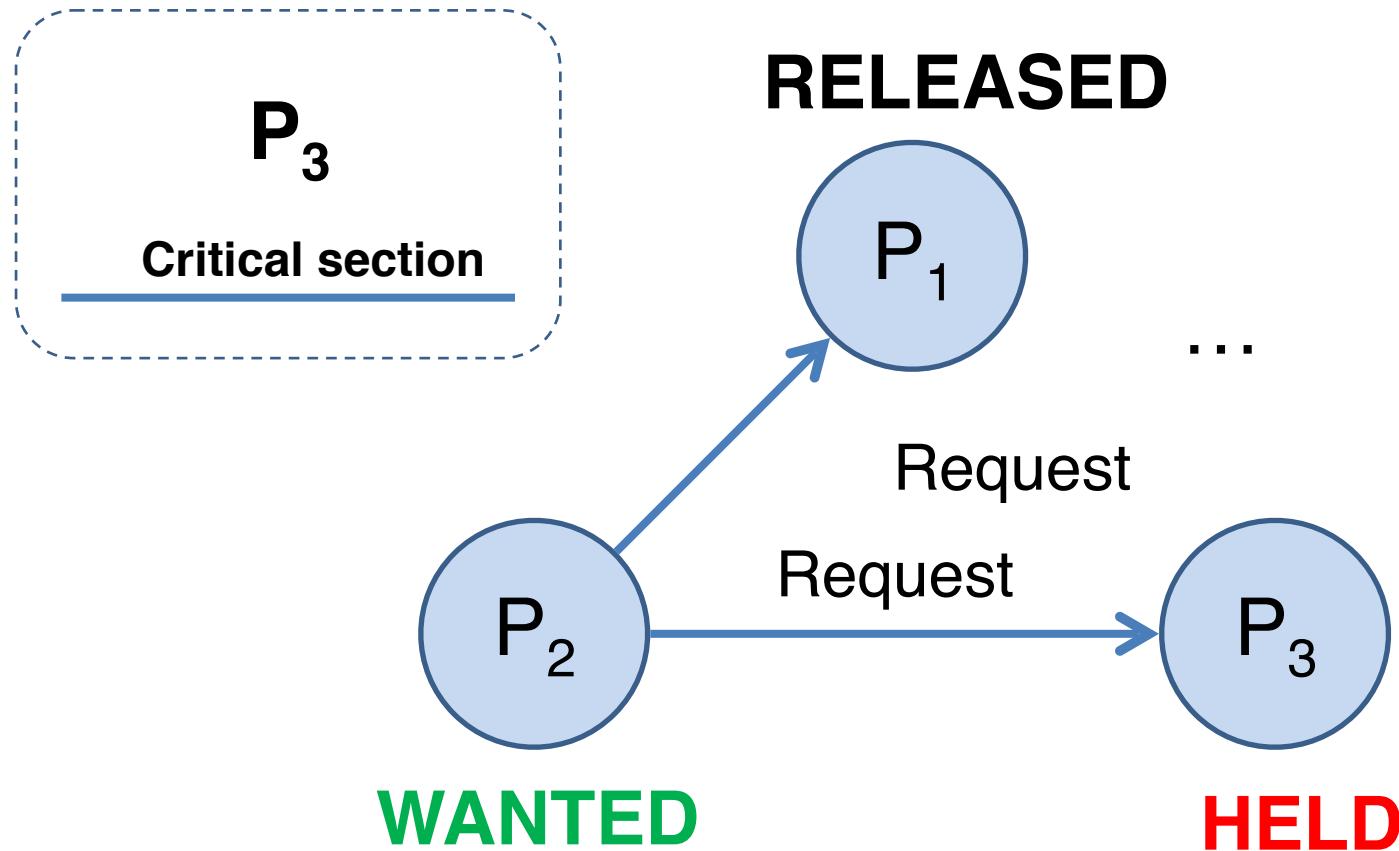
Requesting entry to CS:

Request while Held



Requesting entry to CS:

Request while Held



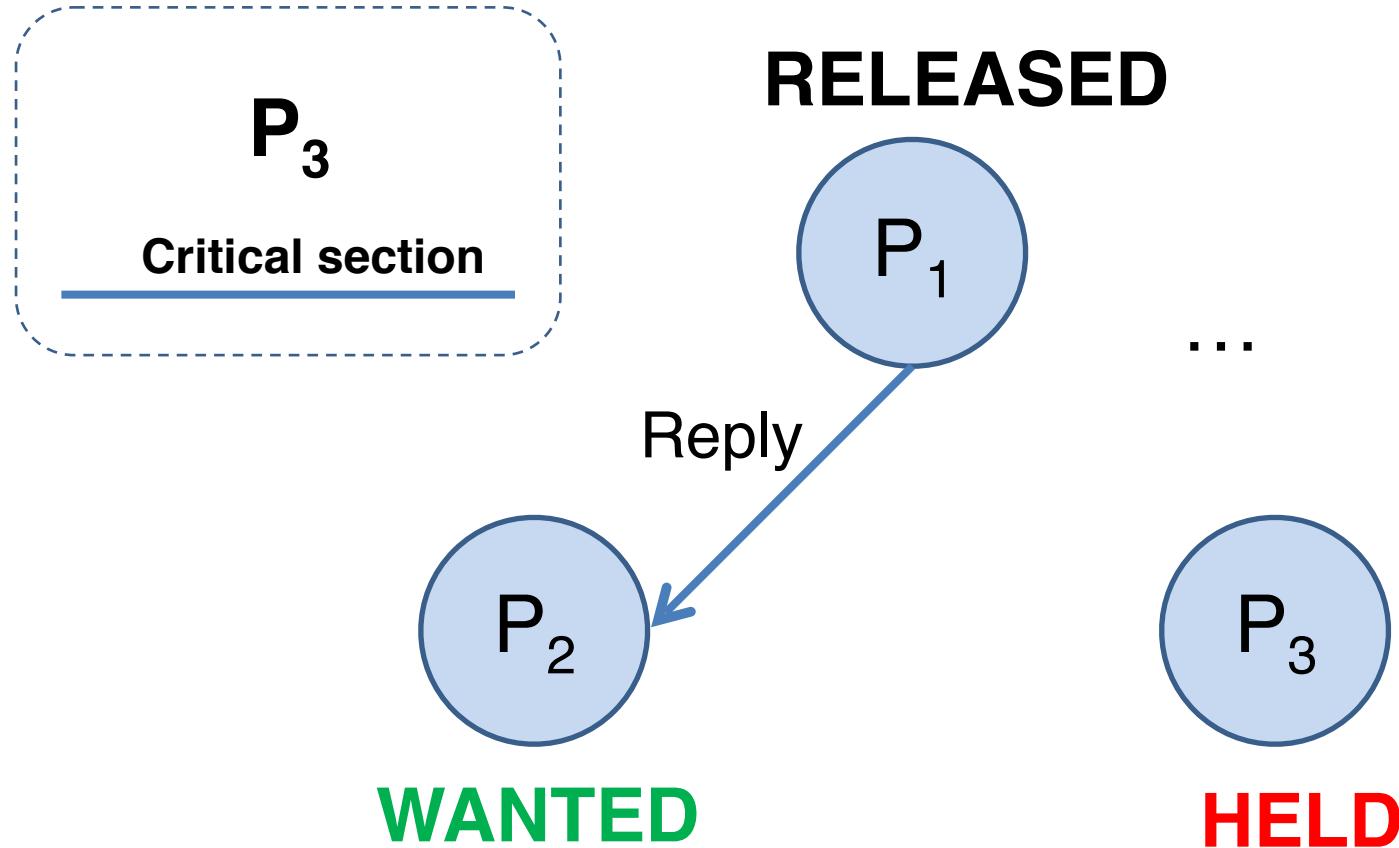
Requesting entry to CS:

Request while Held



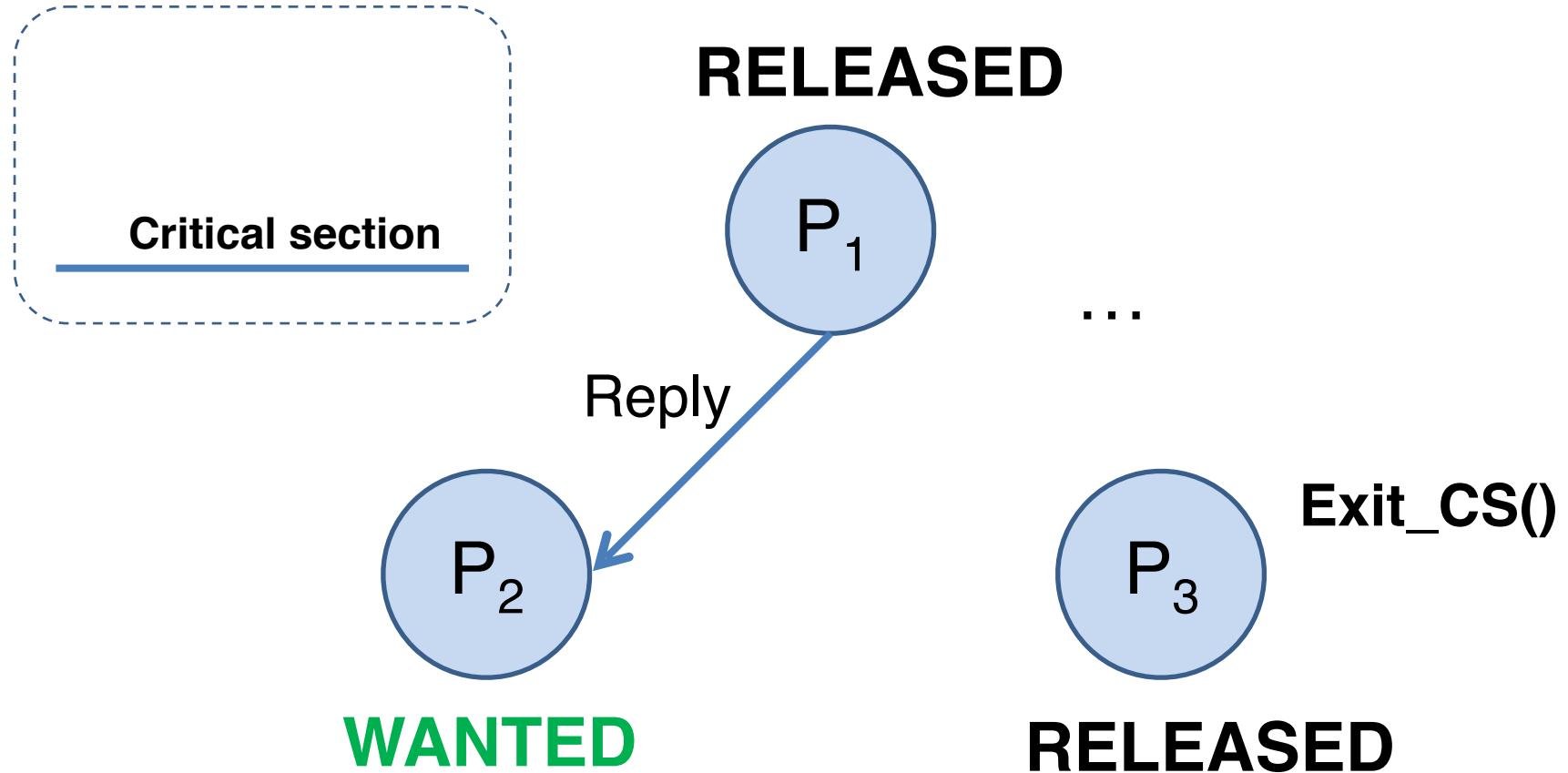
Requesting entry to CS:

Request while Held



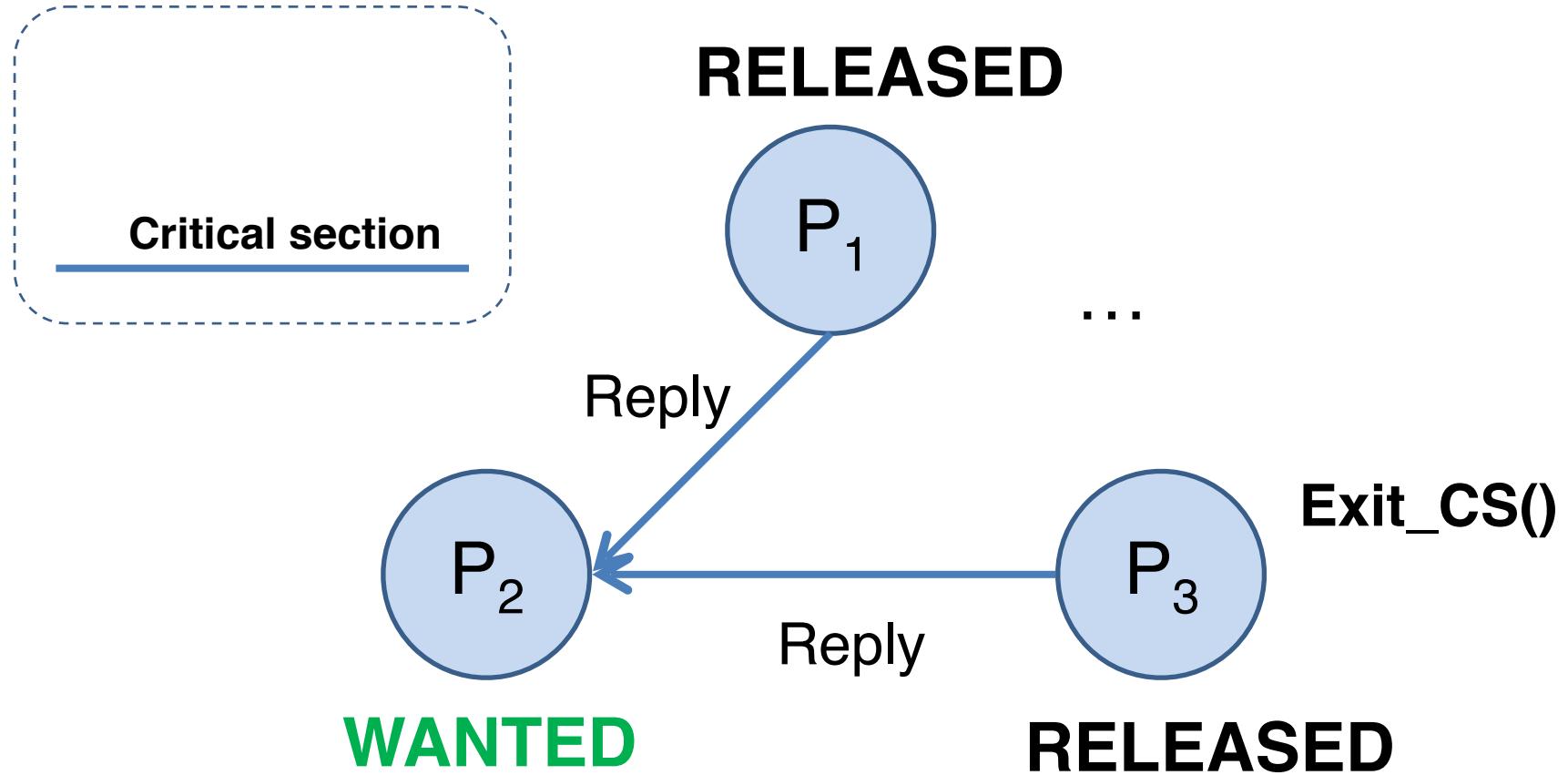
Requesting entry to CS:

Request while Held



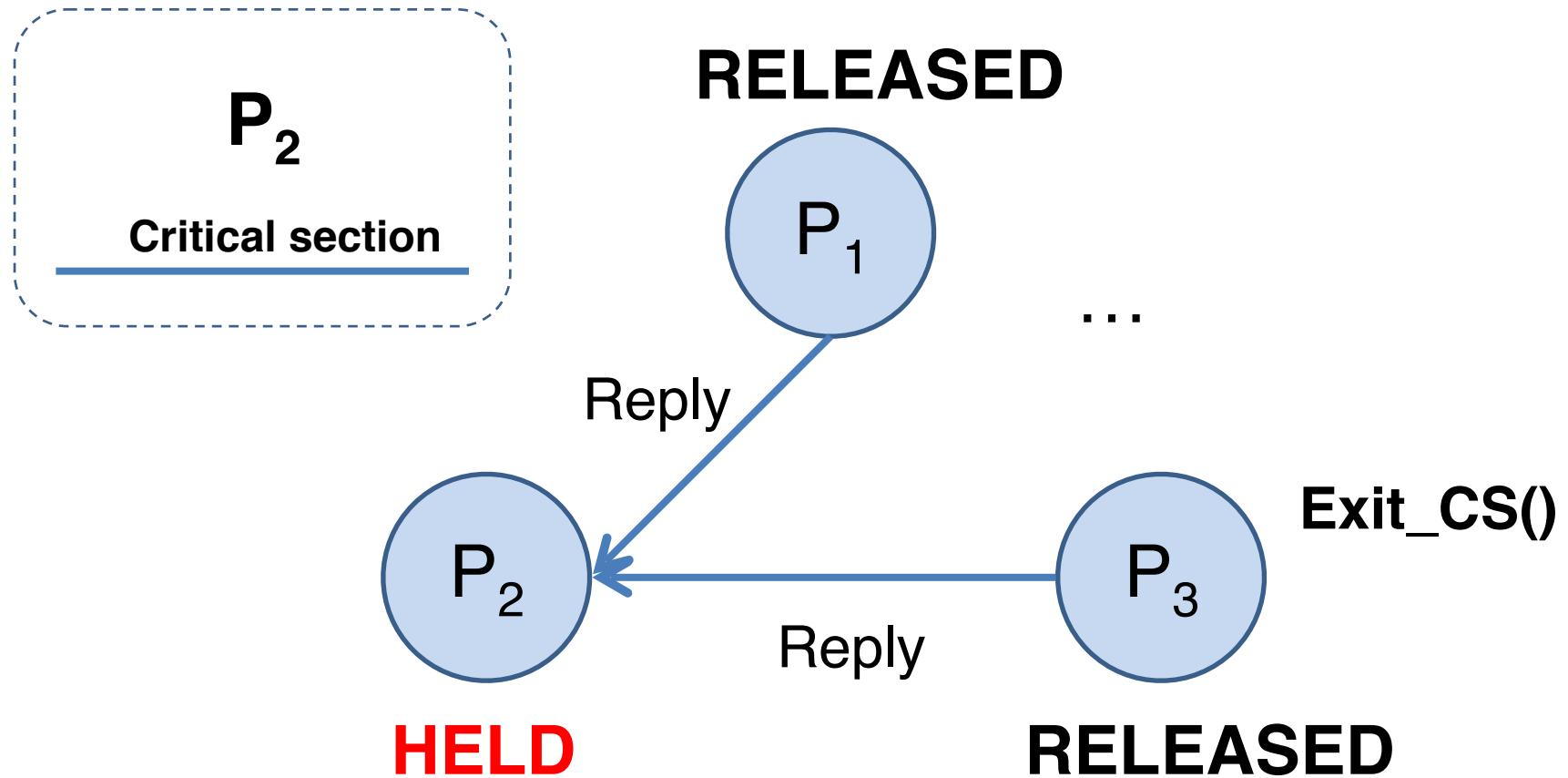
Requesting entry to CS:

Request while Held



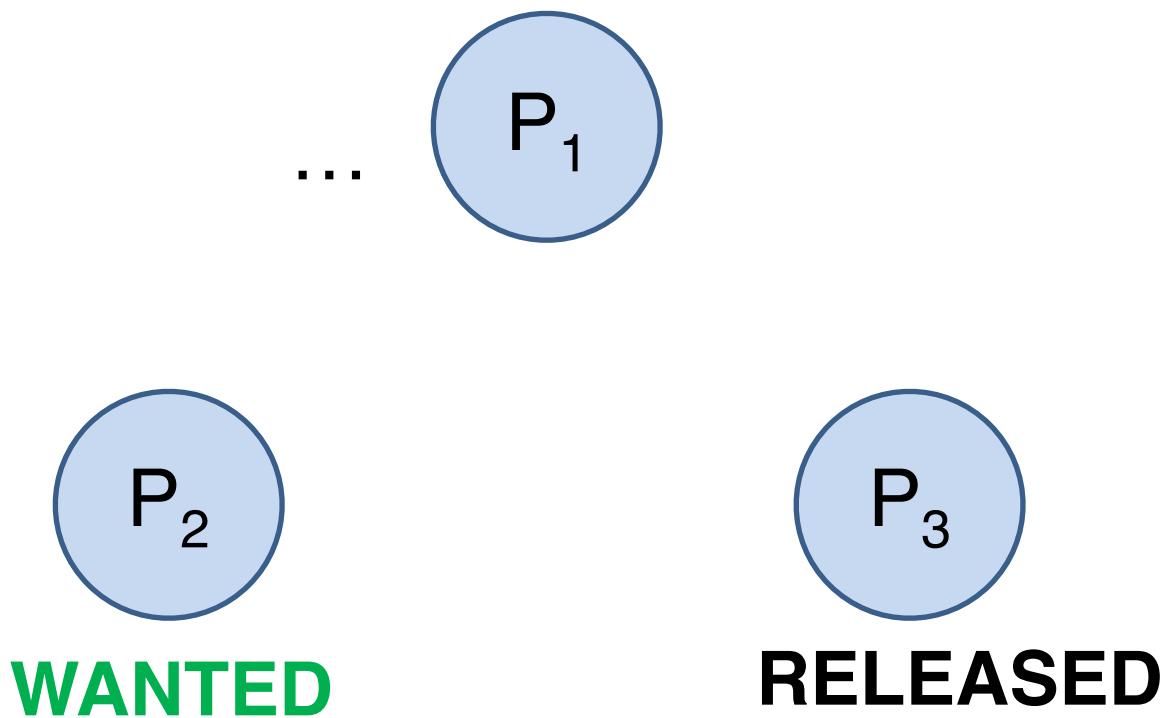
Requesting entry to CS:

Request while Held



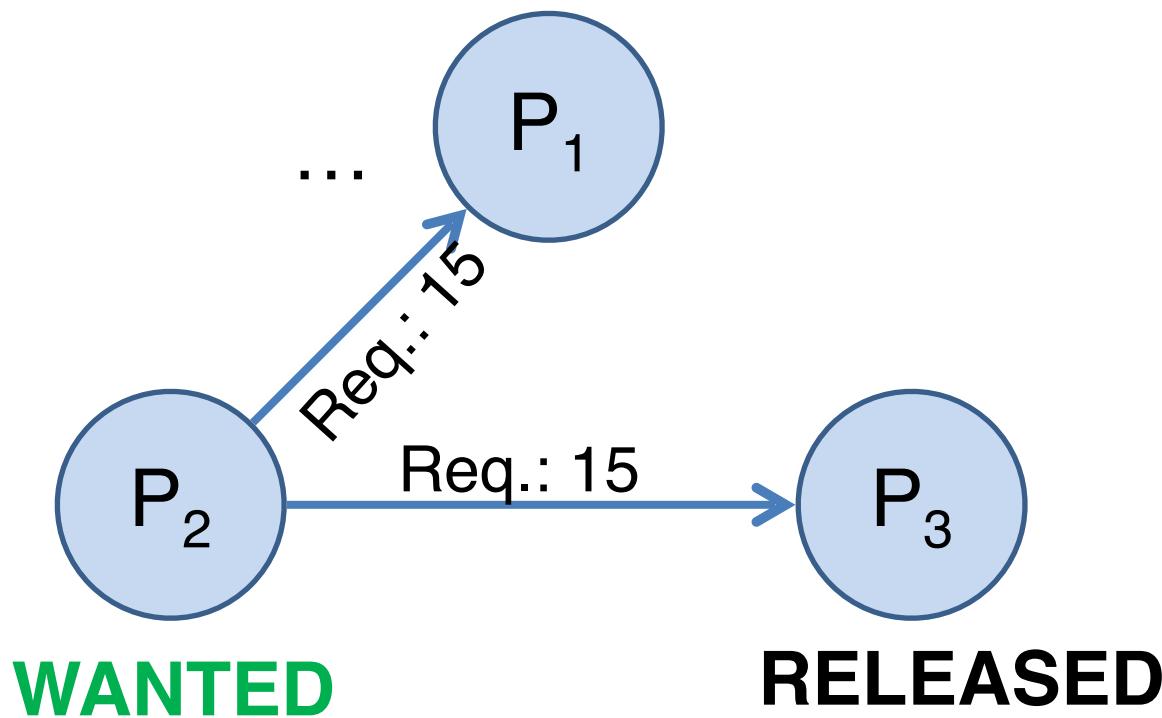
Concurrent entry requests to CS

P_2 and P_3 request entry
to CS concurrently **RELEASED**



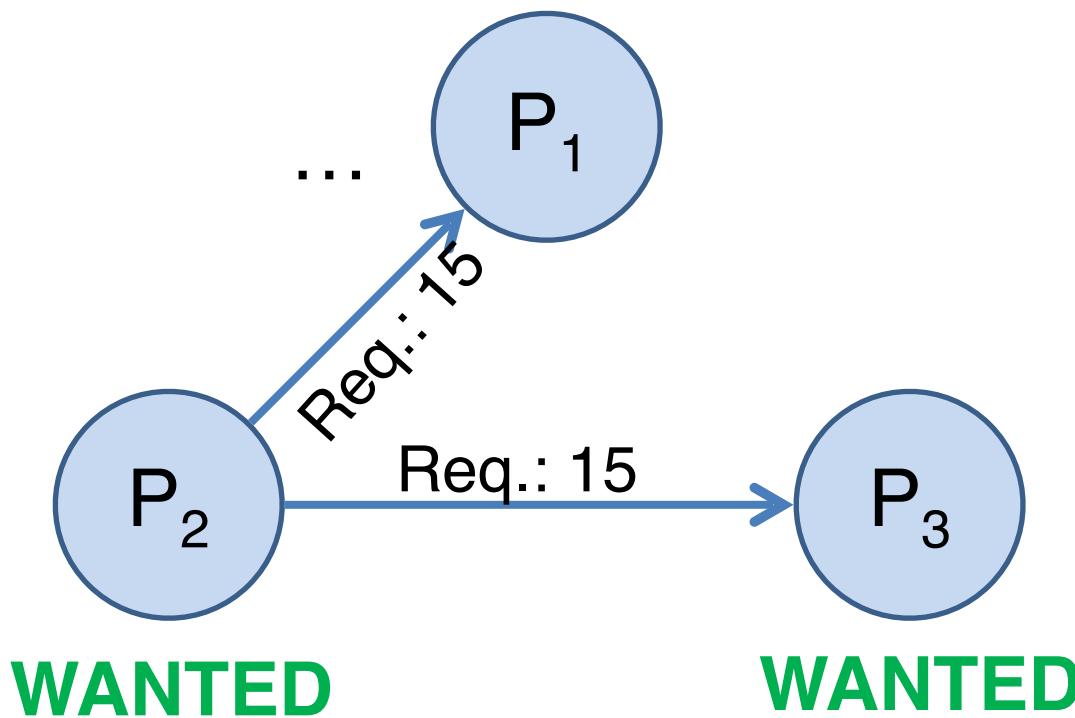
Concurrent entry requests to CS

P_2 and P_3 request entry
to CS concurrently **RELEASED**



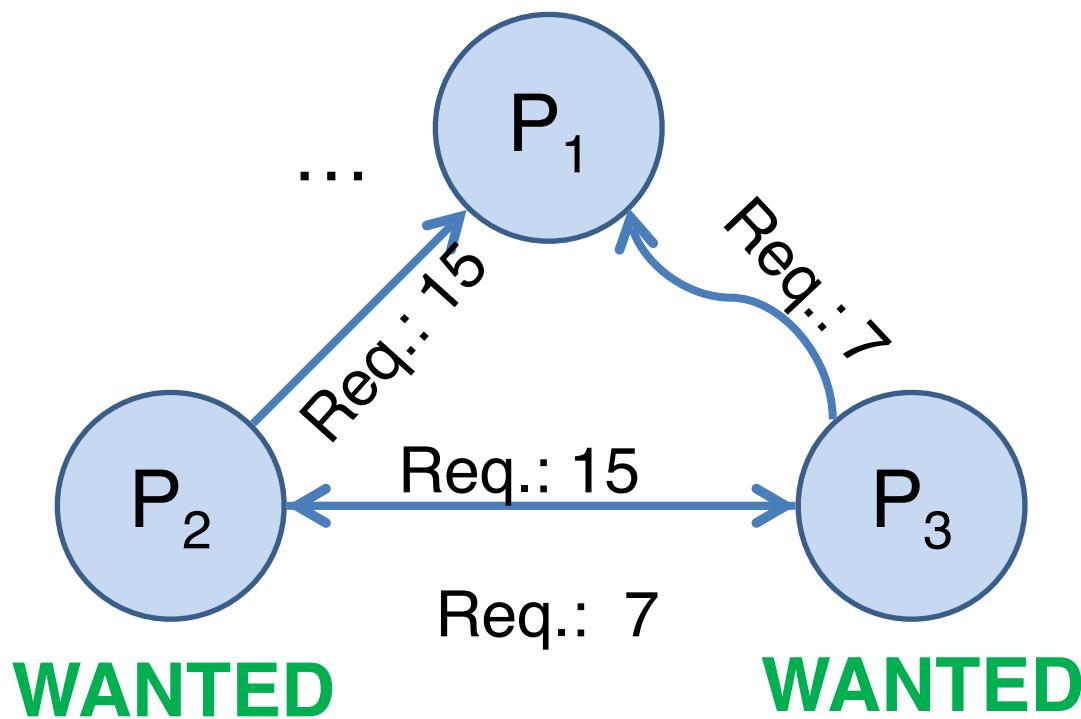
Concurrent entry requests to CS

P_2 and P_3 request entry
to CS concurrently RELEASED



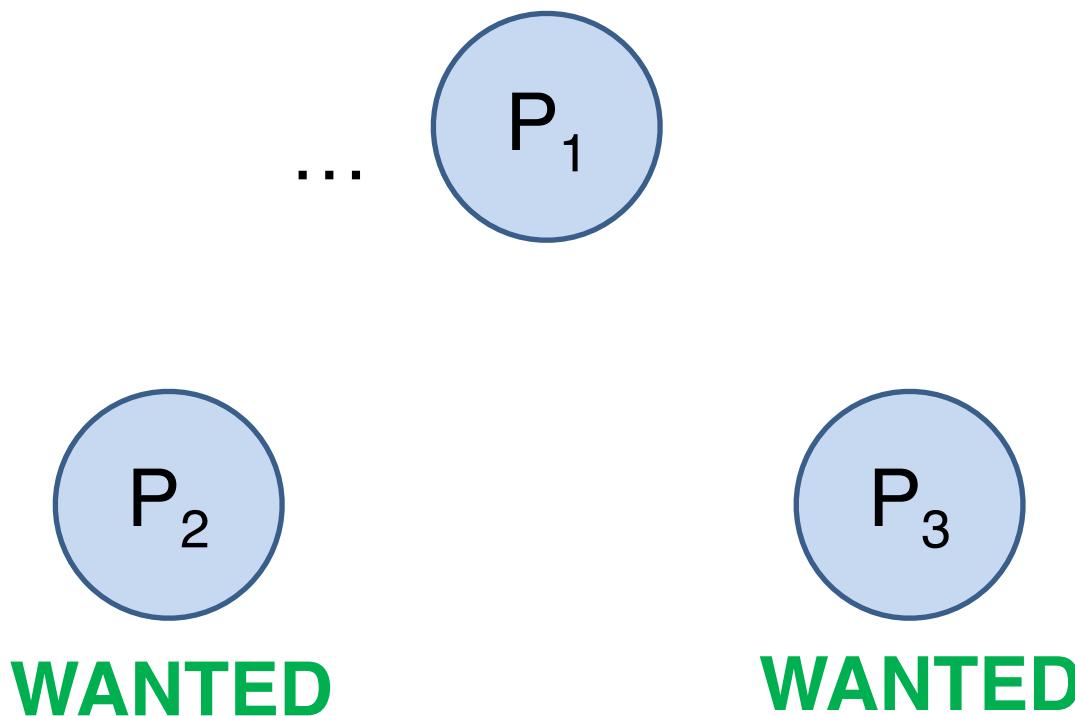
Concurrent entry requests to CS

P_2 and P_3 request entry
to CS concurrently RELEASED



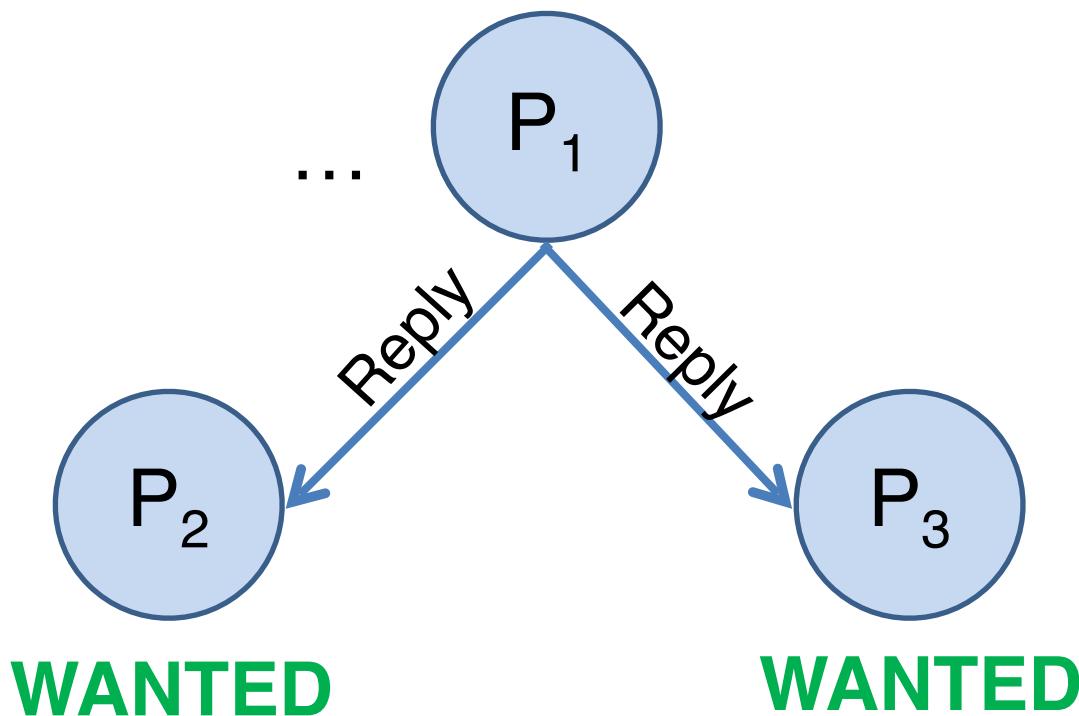
***Concurrent* entry requests to CS**

P_2 and P_3 request entry
to CS concurrently RELEASED



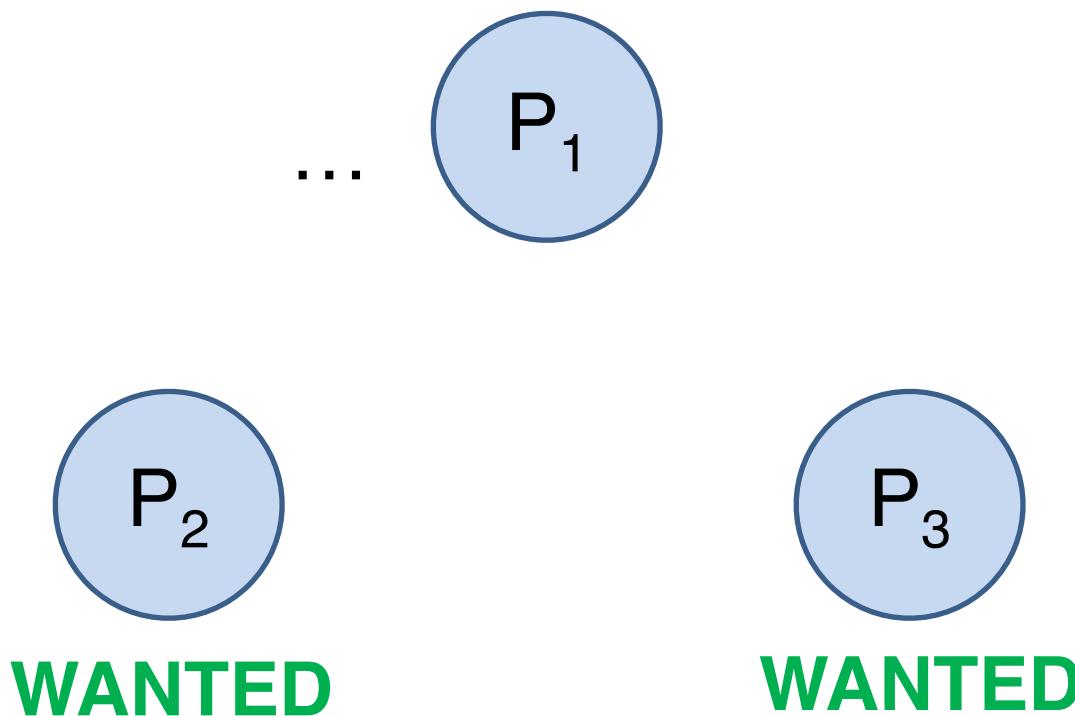
***Concurrent* entry requests to CS**

P_2 and P_3 request entry
to CS concurrently RELEASED



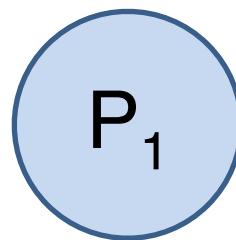
***Concurrent* entry requests to CS**

P_2 and P_3 request entry
to CS concurrently RELEASED

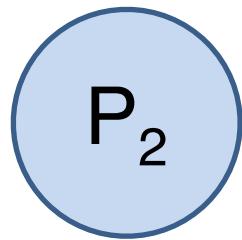


Concurrent entry requests to CS

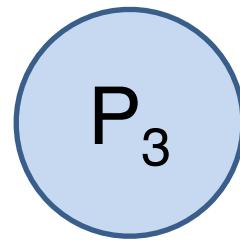
**P₂ and P₃ request entry
to CS concurrently RELEASED**



@P₃: 7 from P₃ < 15 from P₂,
Own timestamp lower,
therefore, hold on to reply to P₂



WANTED



WANTED

Concurrent entry requests to CS

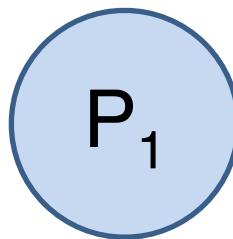
**P₂ and P₃ request entry
to CS concurrently RELEASED**

@P₂: 7 from P₃ < 15 from
P₂,

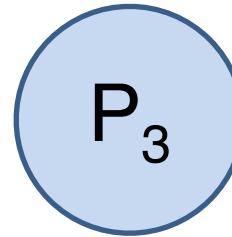
therefore grant P₃

access first
P₂

WANTED



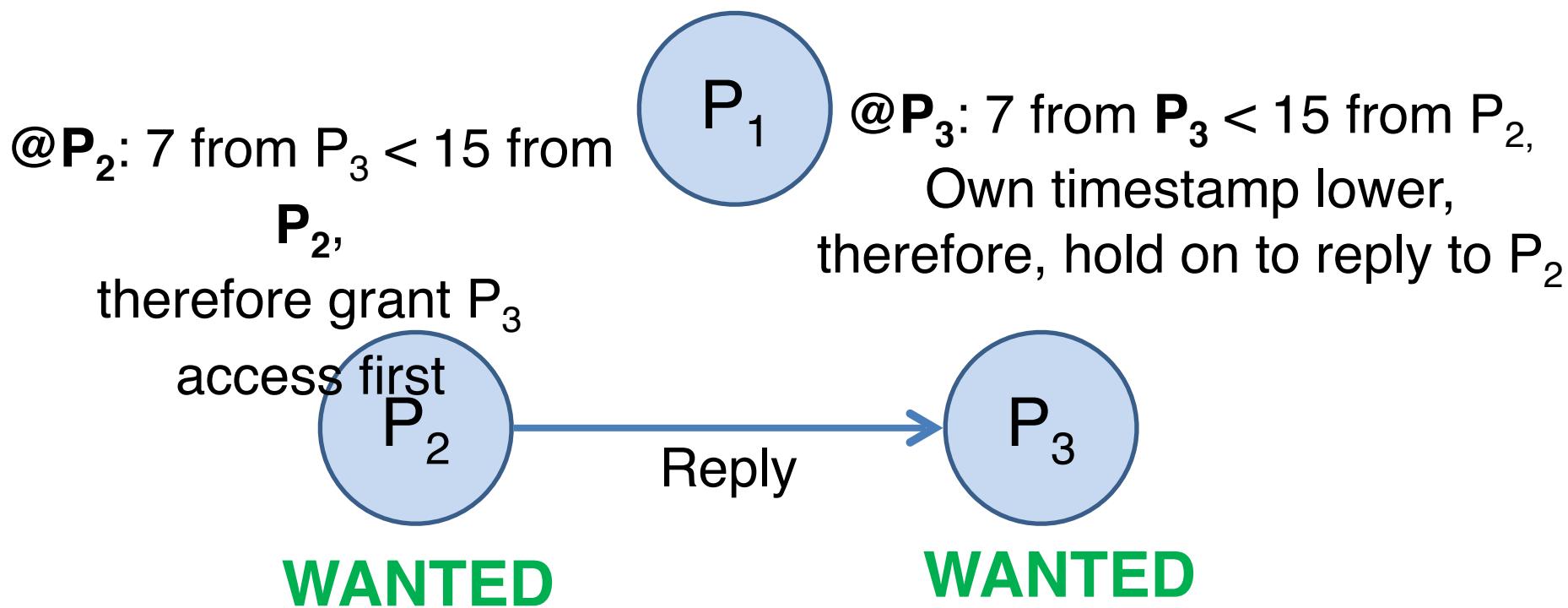
@P₃: 7 from P₃ < 15 from P₂,
Own timestamp lower,
therefore, hold on to reply to P₂



WANTED

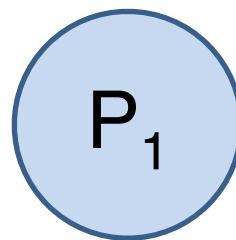
Concurrent entry requests to CS

**P₂ and P₃ request entry
to CS concurrently RELEASED**

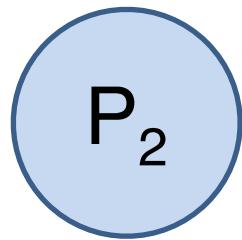


Concurrent entry requests to CS

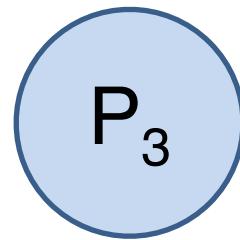
**P₂ and P₃ request entry
to CS concurrently RELEASED**



@P₃: 7 from P₃ < 15 from P₂,
Own timestamp lower,
therefore, hold on to reply to P₂



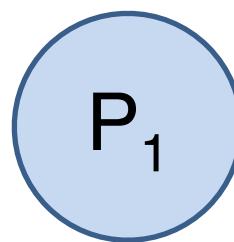
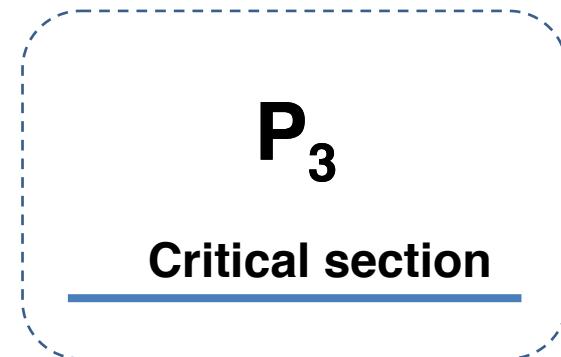
WANTED



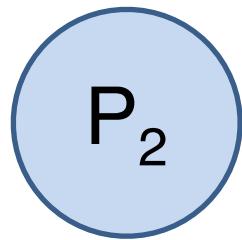
WANTED

Concurrent entry requests to CS

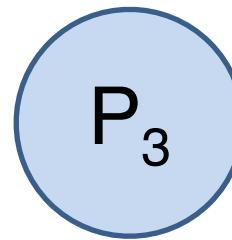
**P₂ and P₃ request entry
to CS concurrently RELEASED**



@P₃: 7 from P₃ < 15 from P₂,
Own timestamp lower,
therefore, hold on to reply to P₂



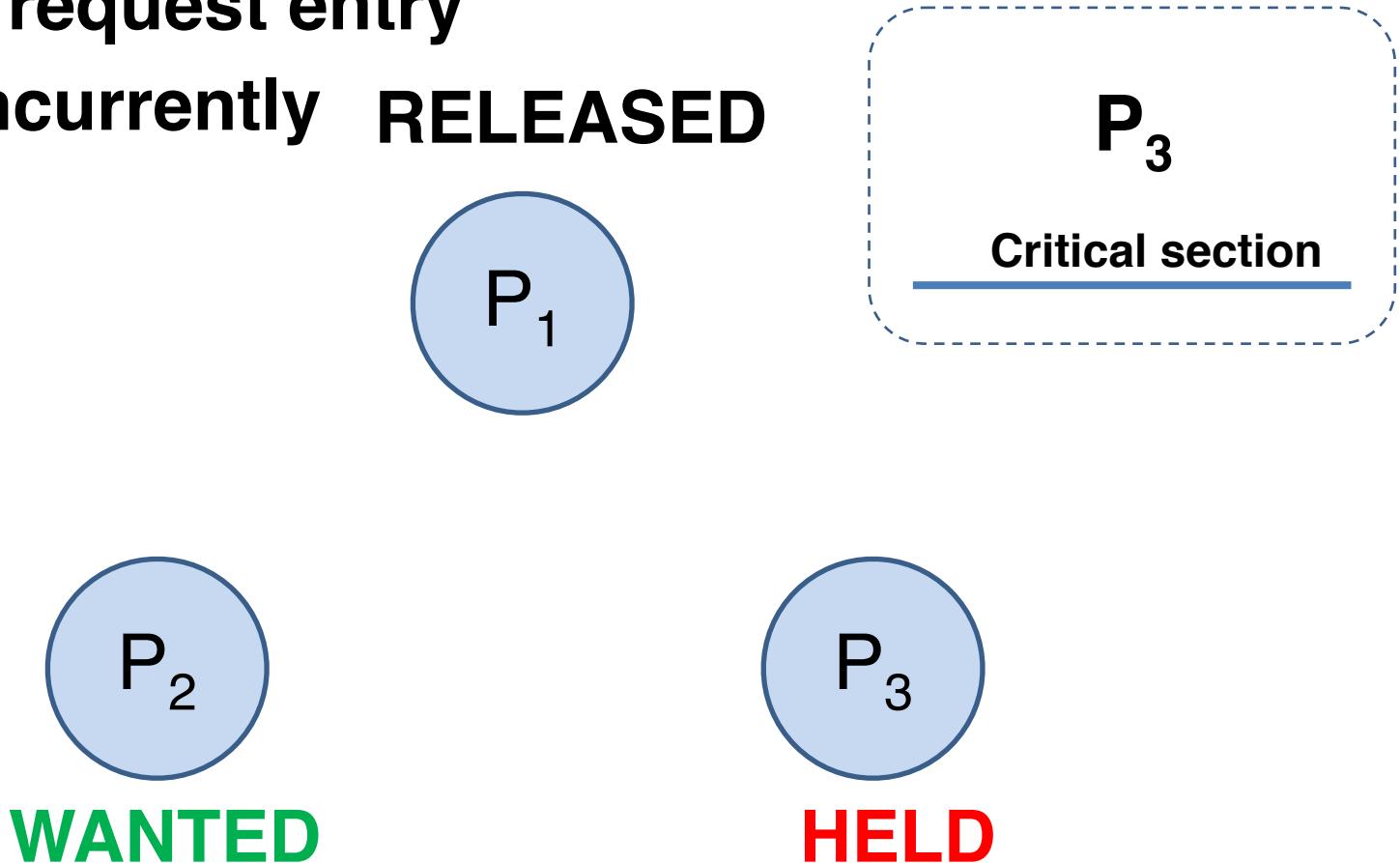
WANTED



HELD

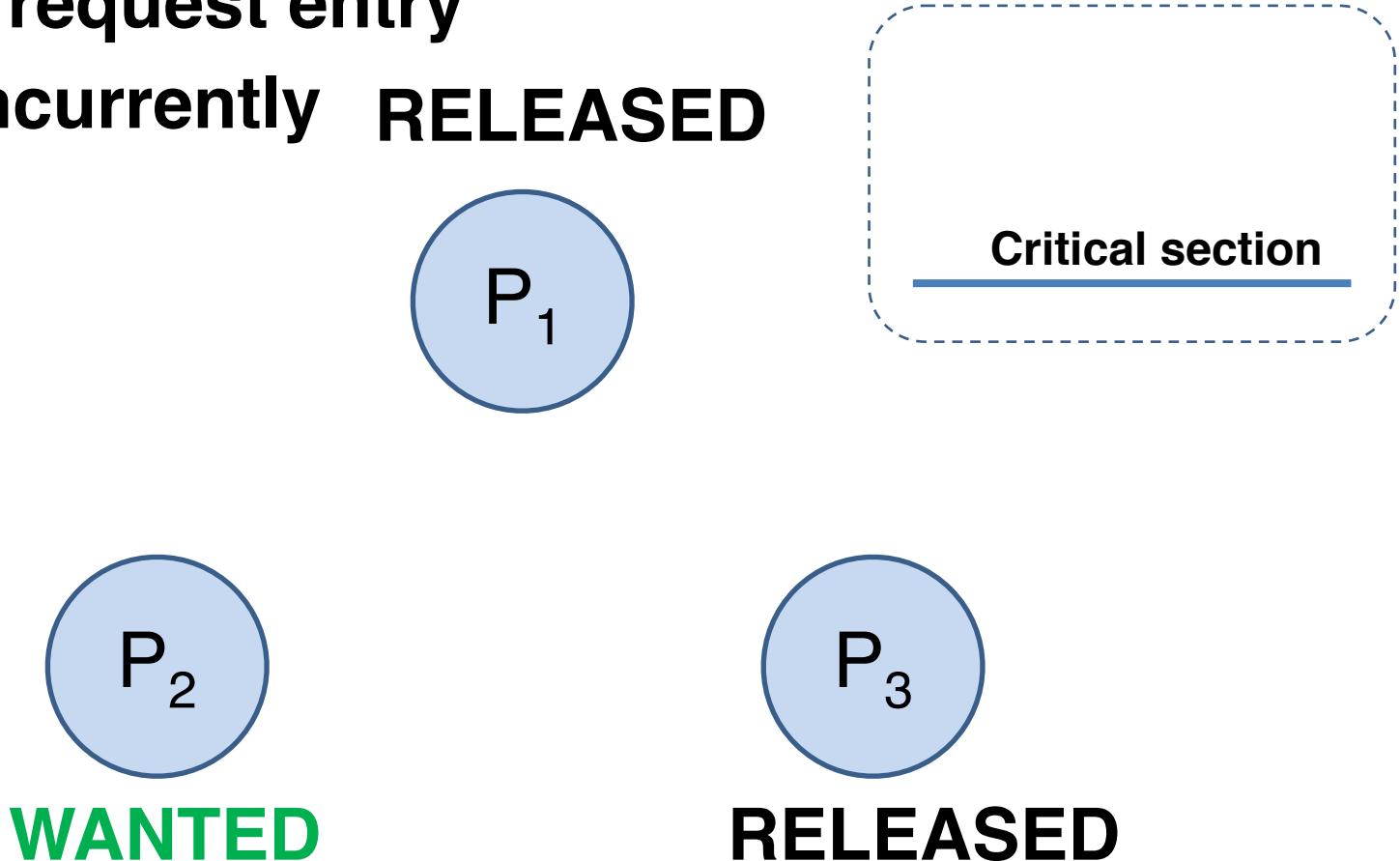
***Concurrent* entry requests to CS**

**P₂ and P₃ request entry
to CS concurrently RELEASED**



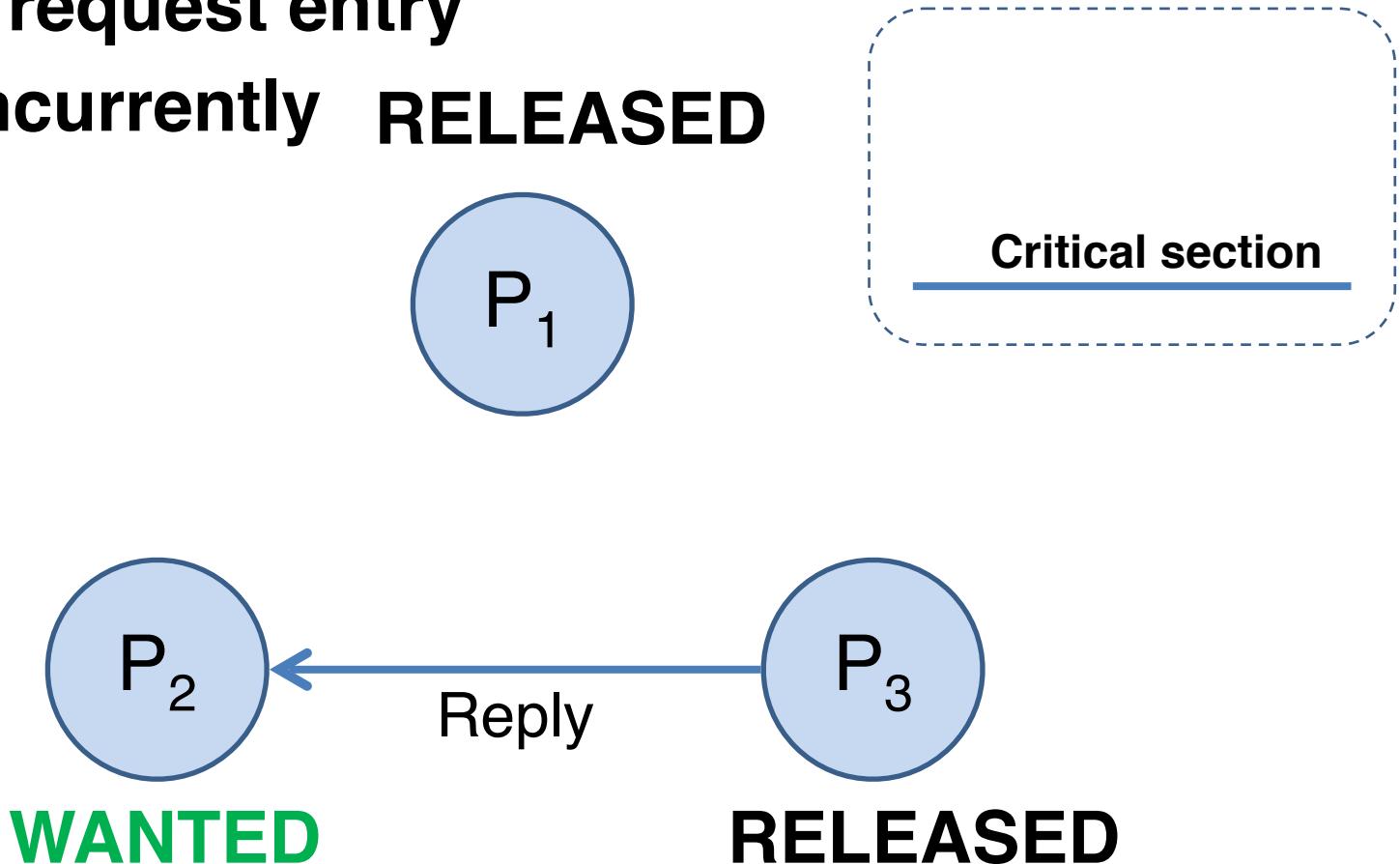
***Concurrent* entry requests to CS**

**P₂ and P₃ request entry
to CS concurrently RELEASED**



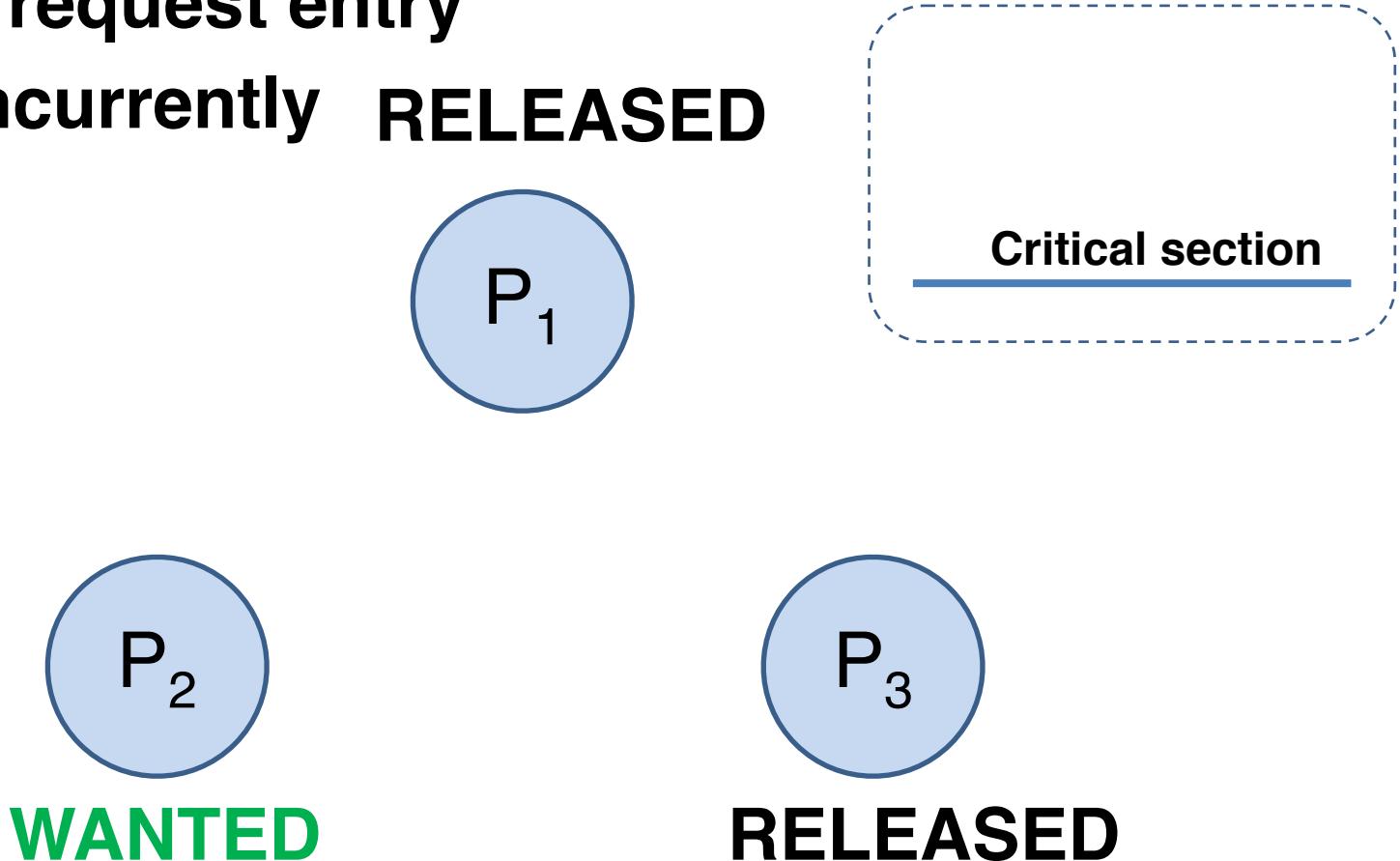
Concurrent entry requests to CS

P_2 and P_3 request entry
to CS concurrently **RELEASED**



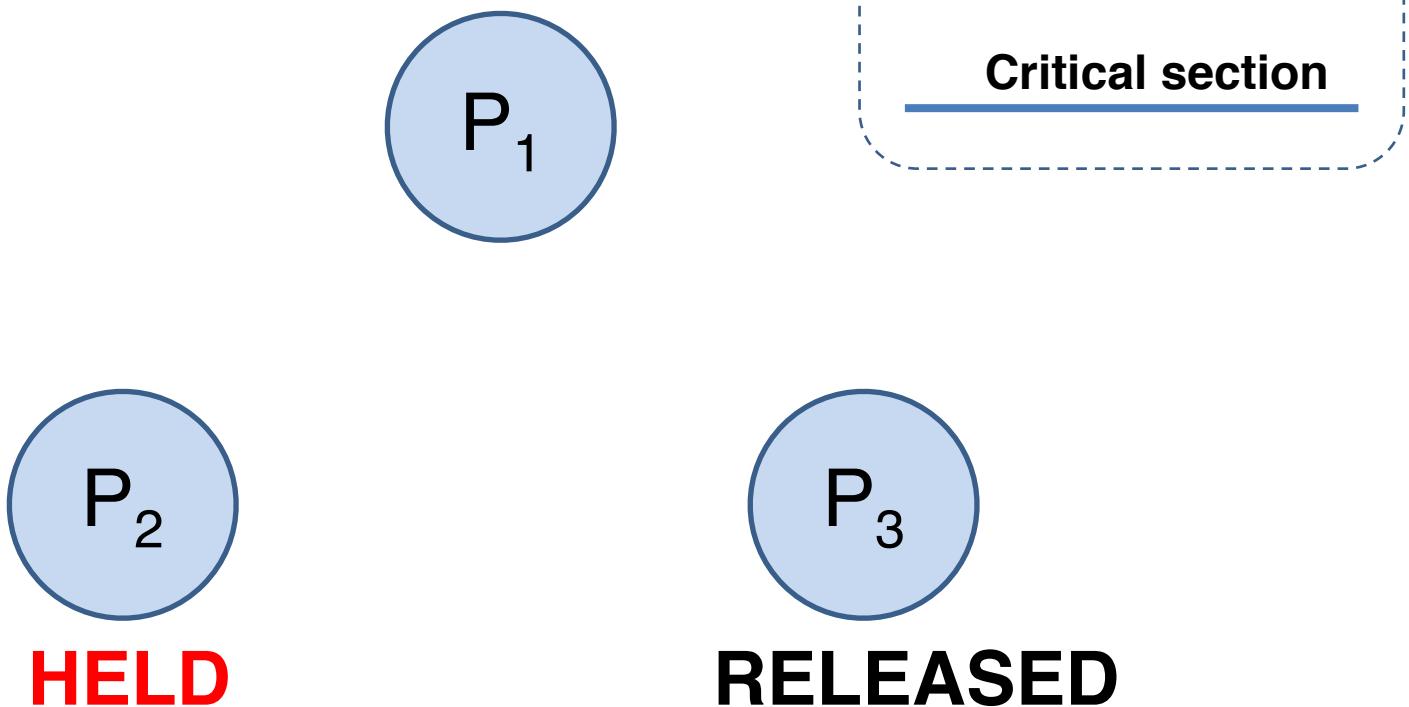
***Concurrent* entry requests to CS**

**P₂ and P₃ request entry
to CS concurrently RELEASED**



Concurrent entry requests to CS

P_2 and P_3 request entry
to CS concurrently **RELEASED**



Reminder: Subtlety about timestamps

- Use **Lamport timestamps** to order requests: $\langle T, P_i \rangle$, T the timestamp, P_i the process identifier
- If for two timestamps
 - $\langle T, P \rangle$ and $\langle S, Q \rangle$, $T=S$, then break ties by looking at process identifiers
 - Gives rise to a total order over timestamps

Ricart & Agrawala algorithm

Ricart & Agrawala algorithm

On initialization
state =
RELEASED

Ricart & Agrawala algorithm

Enter_CS()

state = WANTED

Multicast **request** to all processes

T = request's timestamp

wait until ((n-1) acks received)

state = HELD

On initialization

state =

RELEASED

Ricart & Agrawala algorithm

Enter_CS()

state = WANTED

Multicast **request** to all processes

T = request's timestamp

wait until ((n-1) acks received)

state = HELD

On receiving a request with $\langle T_i, P_i \rangle$ at $P_j (i \neq j)$

if (state==HELD or (state==WANTED and $(T, P_j) < (T_i, P_i)$))

 queue request from P_i without replying

else

 send a reply to P_i

j

On initialization

state =

RELEASED

Ricart & Agrawala algorithm

Enter_CS()

state = WANTED

Multicast **request** to all processes

T = request's timestamp

wait until ((n-1) acks received)

state = HELD

On receiving a request with $\langle T_i, P_i \rangle$

if (state==HELD or (state==WANTED and $(T, P_j) < (T_i, P_i)$))

 queue request from P_i without replying

else

 send a reply to P_i

j

On initialization

state =

~~RELEASED~~

Exit_CS()

state =

RELEASED

Reply to all queued requests at $P_i (i \neq j)$

Ricart & Agrawala algorithm

Enter_CS()

state = WANTED

Multicast **request** to all processes

T = request's timestamp

wait until ((n-1) acks received)
**j's timestamp
(its own)**

state = HELD

On receiving a request with $\langle T_i, P_i \rangle$

if (state==HELD or (state==WANTED and $\langle T_i, P_i \rangle < (T_j, P_j)$))

queue request from P_i without replying

else

send a reply to P_i

On initialization

state =

~~RELEASED~~

Exit_CS()

state =

~~RELEASED~~

Reply to all queued

requests

~~at $P_i \neq j$~~

j

Ricart & Agrawala algorithm

Enter_CS()

state = WANTED

Multicast **request** to all processes

T = request's timestamp

wait until ((n-1) acks received)

state = HELD

On receiving a request with $\langle T_i, P_i \rangle$

if (state==HELD or (state==WANTED and $(T, P_j) < (T_i, P_i)$))

 queue request from P_i without replying

else

 send a reply to P_i

j

On initialization

state =

~~RELEASED~~

Exit_CS()

state =

RELEASED

Reply to all queued requests at $P_i (i \neq j)$

Potential fault-tolerance issues

(Due to our assumption, do not apply here.)

- None of the algorithms tolerate message loss
- Ring-based algorithm cannot tolerate crash of single process
- Centralized algorithm can tolerate crash of processes that are neither in the CS, nor have requested entry
- Lamport, R&A can not tolerate faults either

LEADER ELECTION

Leader election

- **Problem:** A group of processes, P_1, \dots, P_n , **must agree** on some **unique P_k** to be the “**leader**”
- Often, the leader then **coordinates another activity**
- Election runs when leader (a.k.a., coordinator) failed
- Any process who hasn’t heard from the leader in some predefined time interval **may call for an election**
- **False alarm** is a possibility (new election initiated, while current leader still alive)
- **Several processes** may initiate **elections concurrently**
- Algorithm should allow for process crash during election

Applications of leader election

- Berkeley clock synchronization algorithm
- Centralized mutual exclusion algorithm
- Leader election for choosing the master in Hbase / Bigtable (using ZooKeeper/Chubby)
- Choosing the master among the 5 servers of Chubby or ZooKeeper cell
- *Primary-backup replication algorithms*
- *Two-phase commit protocol*

As compared to mutual exclusion

- Losers return to what they were doing ...
 ... instead of waiting
- Fast election is important ...
 ... not starvation avoidance
- All processes must know the result ...
 ... not just the winner

As compared to mutual exclusion

- Losers return to what they were doing ...
 ... instead of waiting
- Fast election is important ...
 ... not starvation avoidance
- All processes must know the result ...
 ... not just the winner

ME can be reduced to LE!
(e.g., Hbase wants LE,
ZooKeeper provides ME)

Process identifier

- Elected **leader** must be **unique**
- Active process with the **largest identifier** wins
- Identifier could be any “useful value”
 - i.e., **unique & totally ordered** (e.g. **unique integers**)
- E.g., based on OS process identifiers, IP adr., port
- E.g., based on least computational load
 - $\langle 1/\text{load}, i \rangle$, $\text{load} > 0$, i is process ID to break ties
- Each process, P_i , has a variable **elected**, that holds the value of the leader or is “ \perp ” (undefined)

Election algorithm requirement

- **Safety**
 - A participating process, P_i , has variable $\text{elected}_i = \perp$ or $\text{elected}_i = P$, where P is chosen as the non-crashed process at the end of the election run with the **largest identifier**.
(Only one leader at a time!)
- **Liveness**
 - **All processes participate** in the election and eventually either set $\text{elected}_i \neq \perp$ or **crash**.

Chang & Roberts Ring-based algorithm, 1978

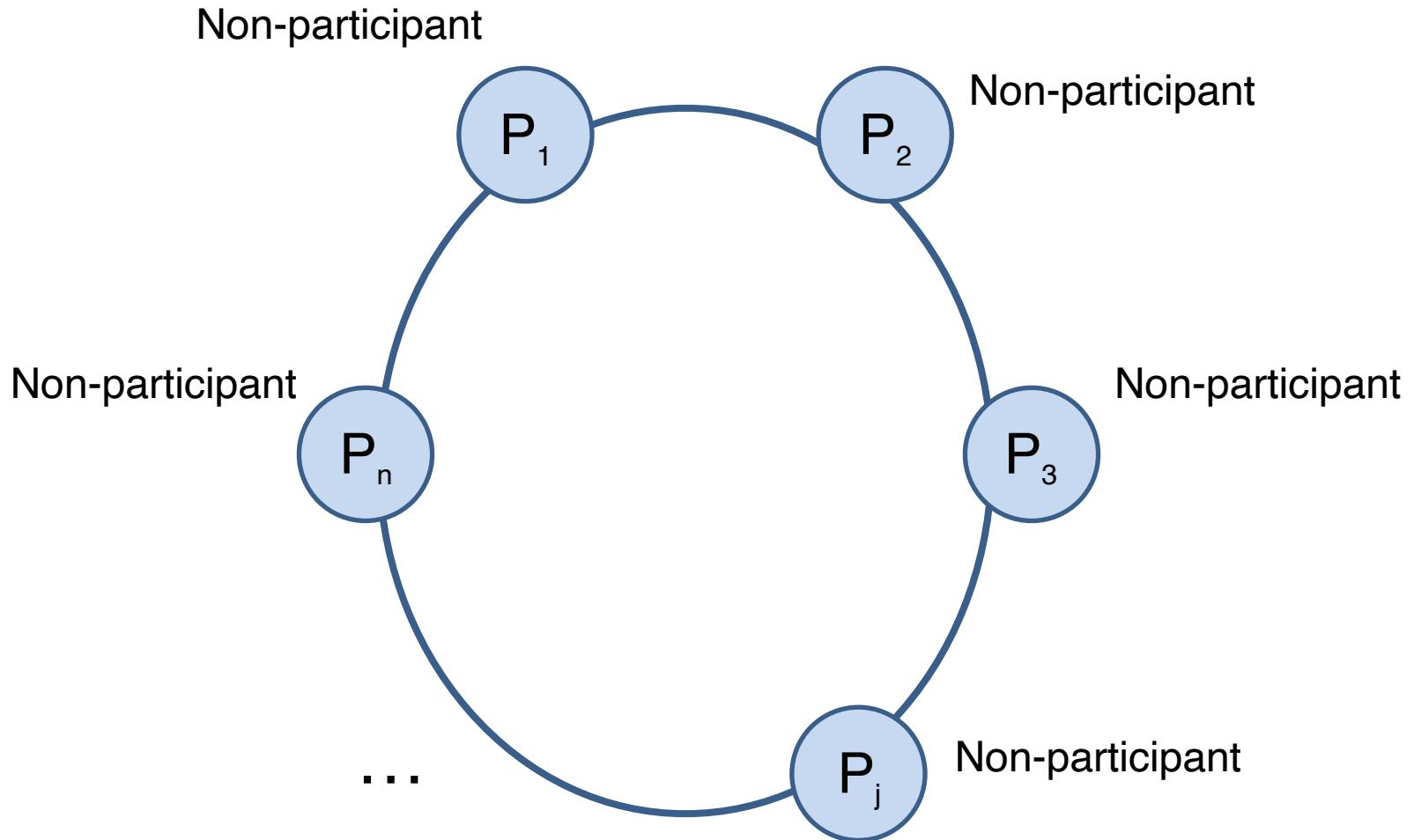
Essentially three phases

1. Initialization
2. Election phase (concurrent calls for election)
 - Determine election winner (voting phase)
 - Reach point of message originator
3. Announce the leader phase (victory announcement phase)

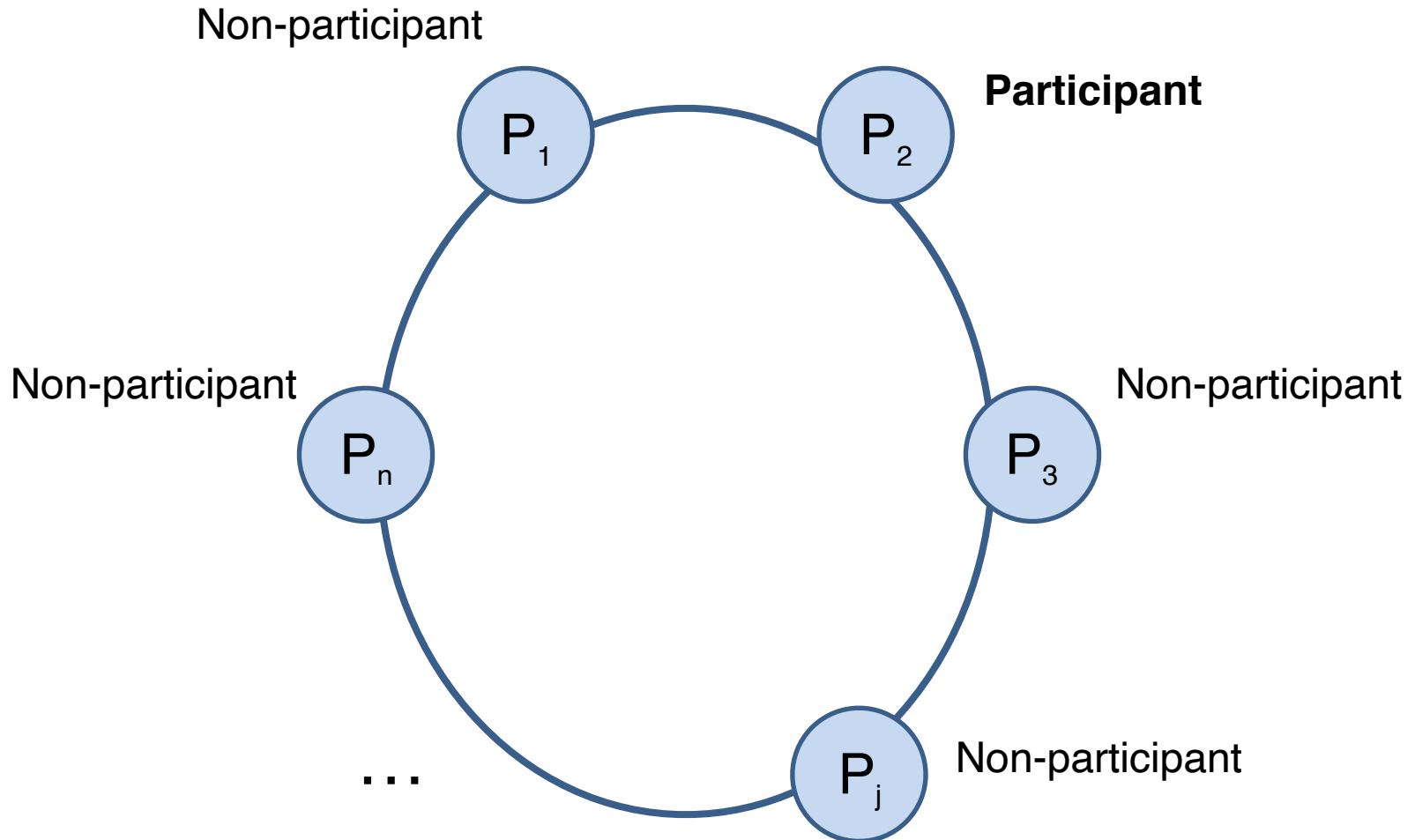
Ring-based election algorithm

- Construct a ring (cf. ring-based mutual exclusion)
- Assume, each P has **unique ID (e.g., unique integer)**
- Assume, no failures and asynchronous system, **but failures can happen before the election!**
- Any P_i may begin an election by sending an **election message** to its successor (i.e., after detecting leader failure)
- Election message holds P_i 's ID
- Upon receiving an election message, P_k **compares its own ID to ID in received message**
 - If received message ID is **greater**: **Forward** election message
 - If ... **smaller**: **Forward** election message **with P_k 's ID, unless P_k has already sent a message (participated) in the current election run**
 - If ... **equal**: P_k **is now leader**. **Forward victory message** to notify all other processes

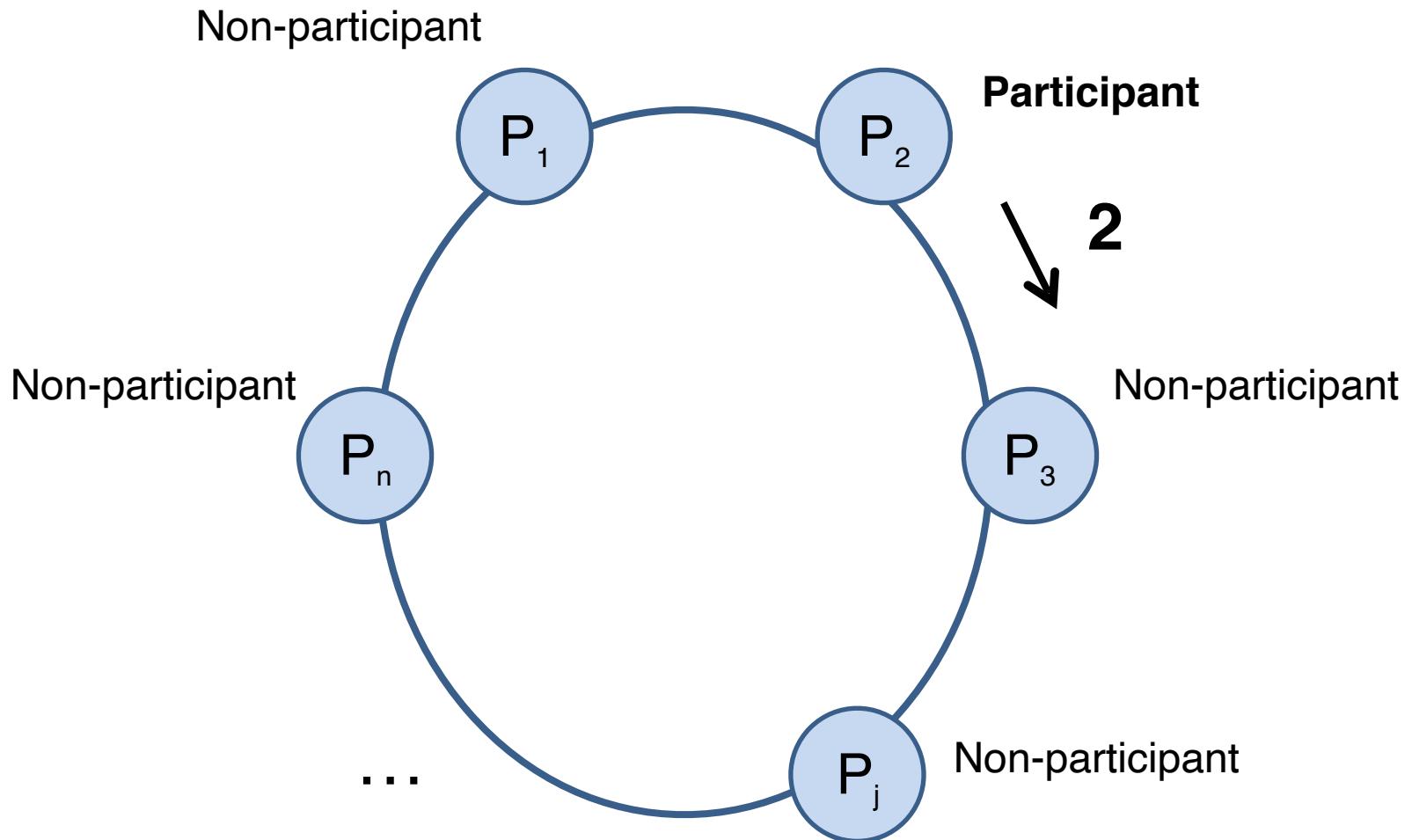
Ring-based algorithm: Calling an Election (determine winner)



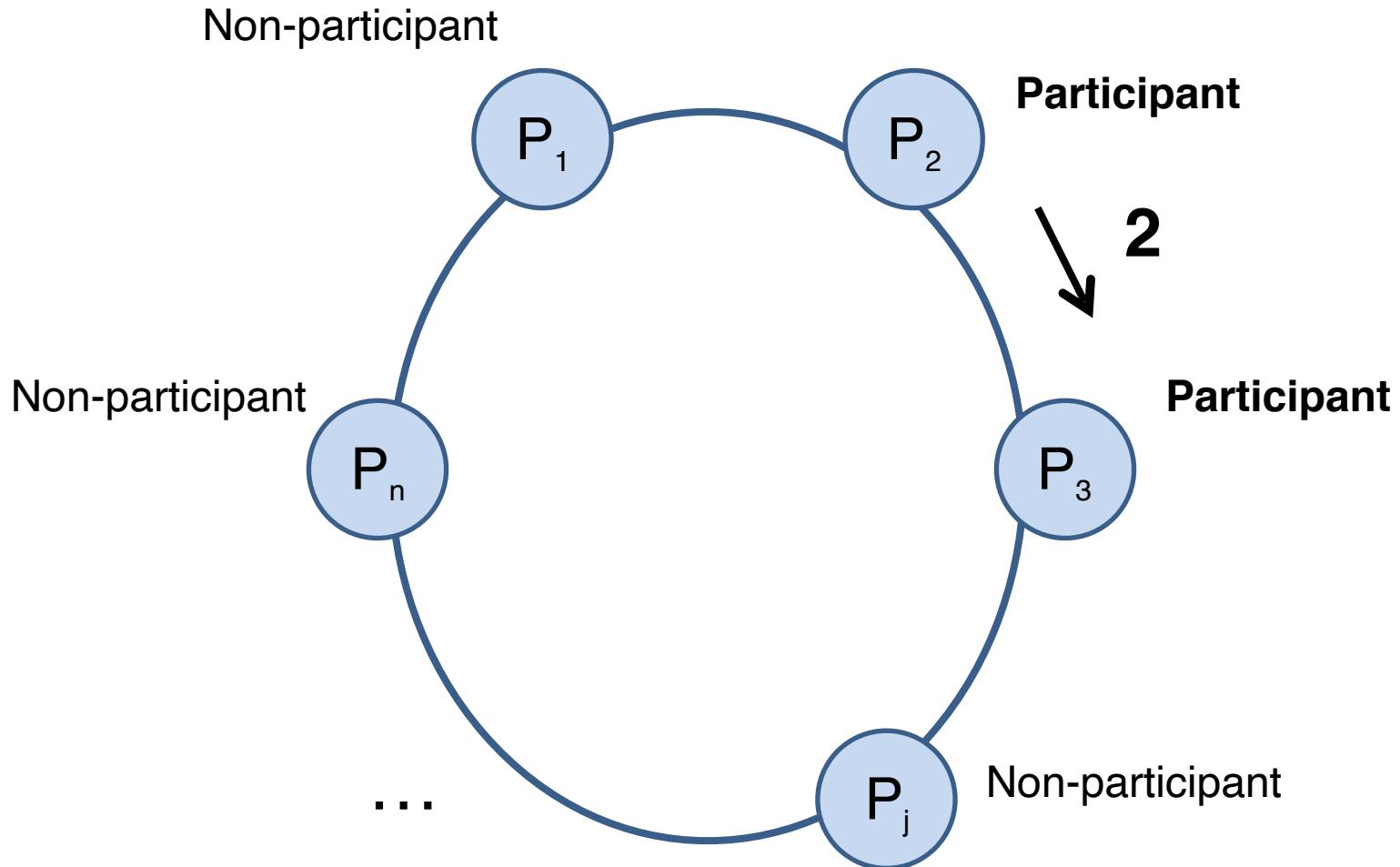
Ring-based algorithm: Calling an Election (determine winner)



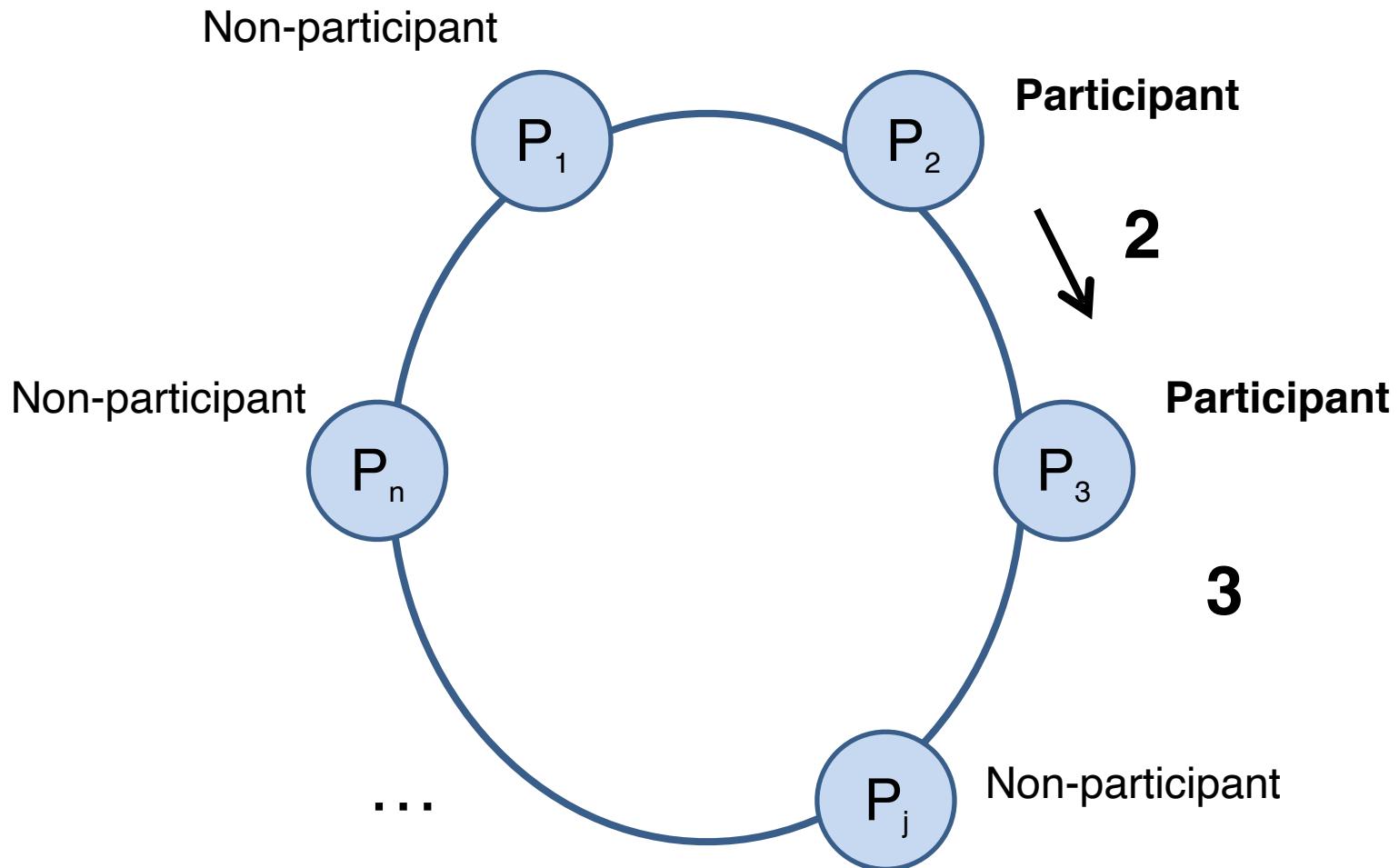
Ring-based algorithm: Calling an Election (determine winner)



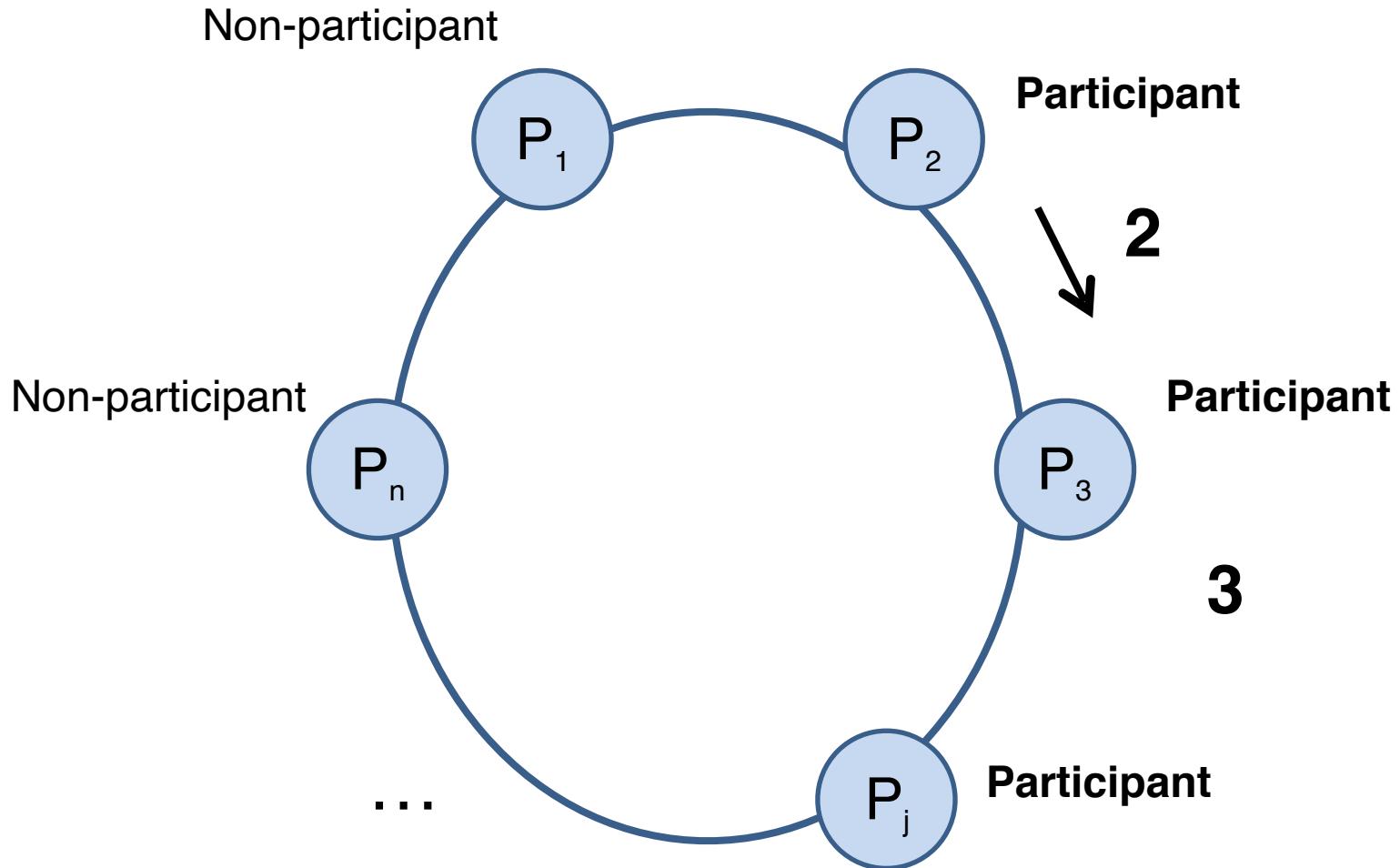
Ring-based algorithm: Calling an Election (determine winner)



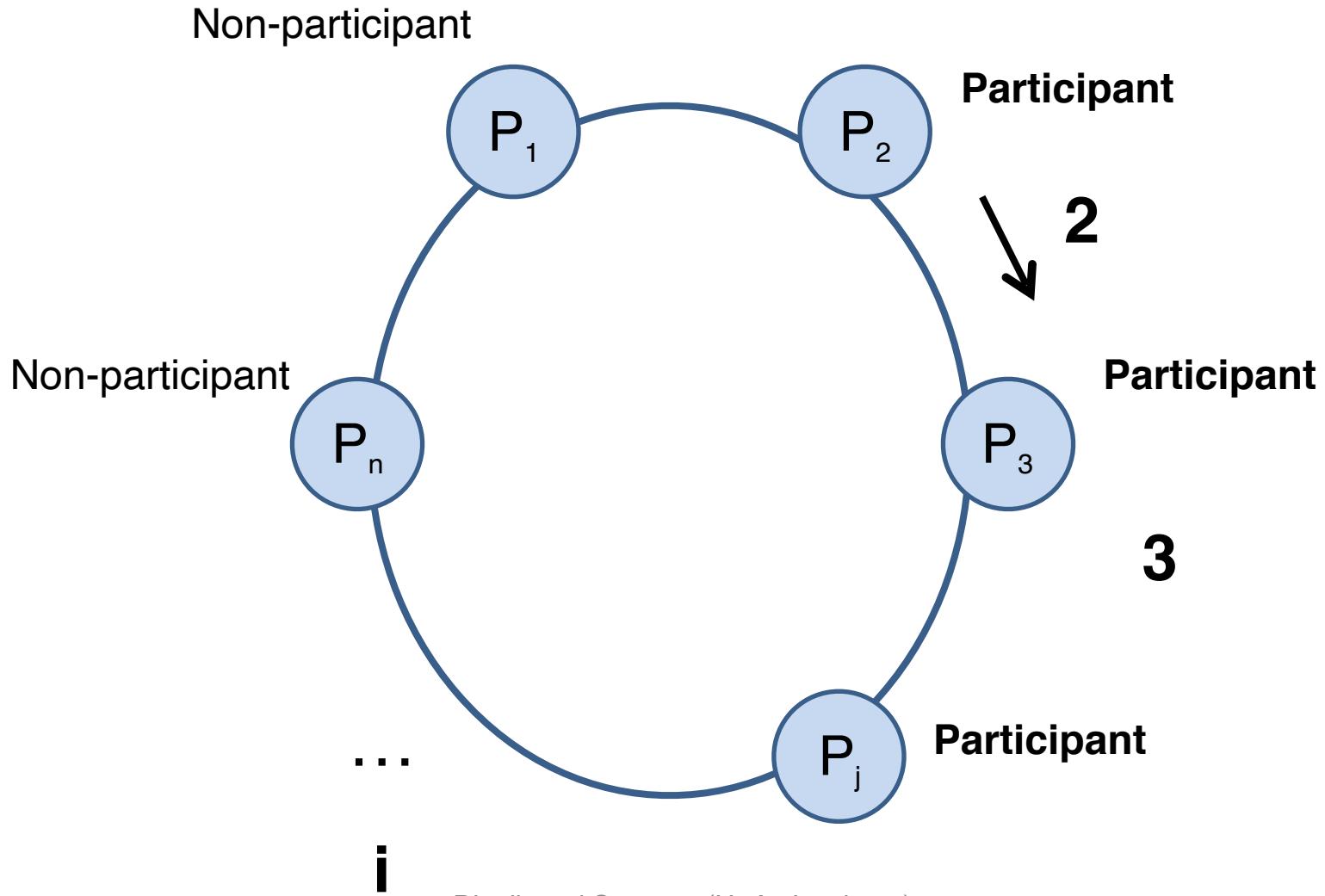
Ring-based algorithm: Calling an Election (determine winner)



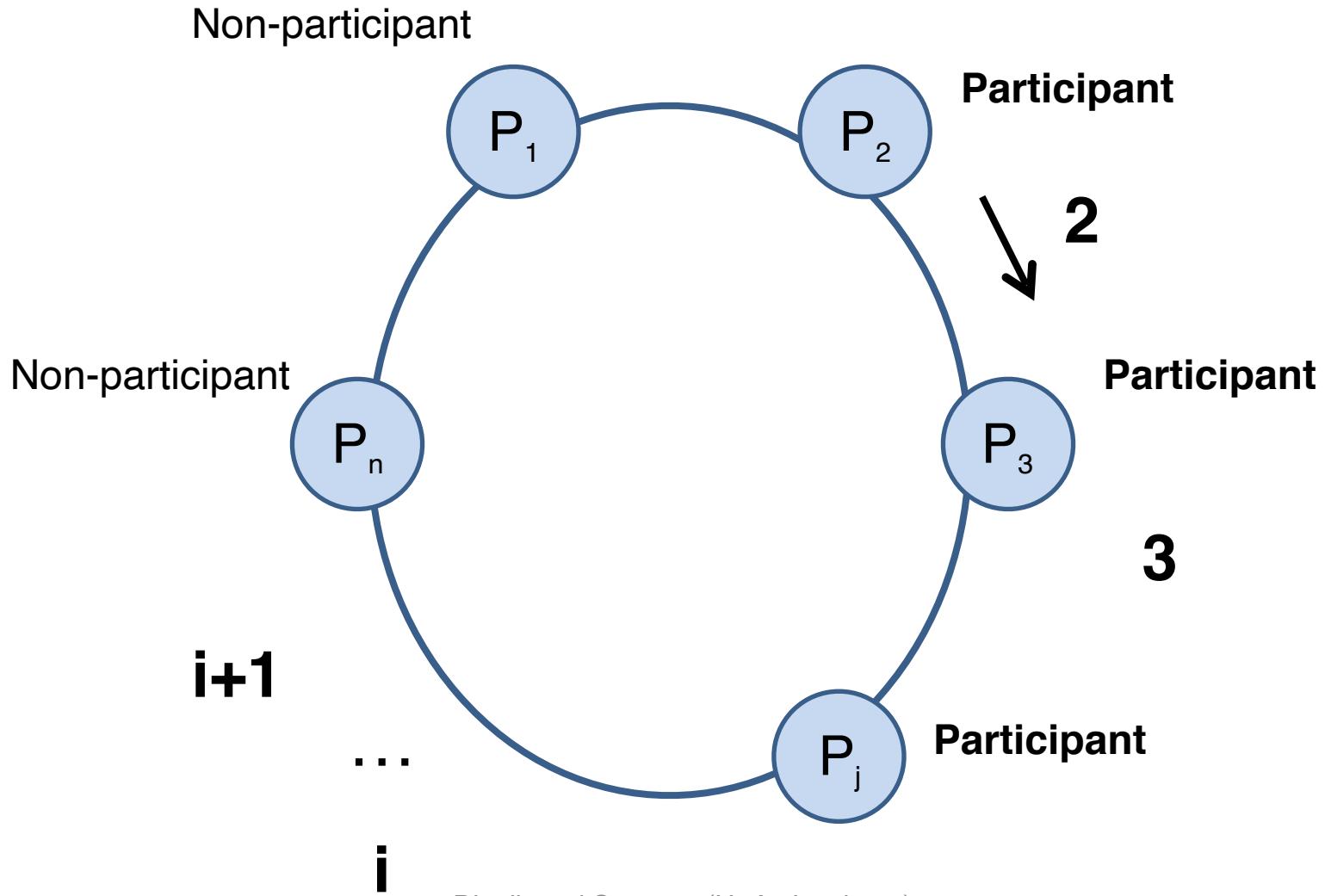
Ring-based algorithm: Calling an Election (determine winner)



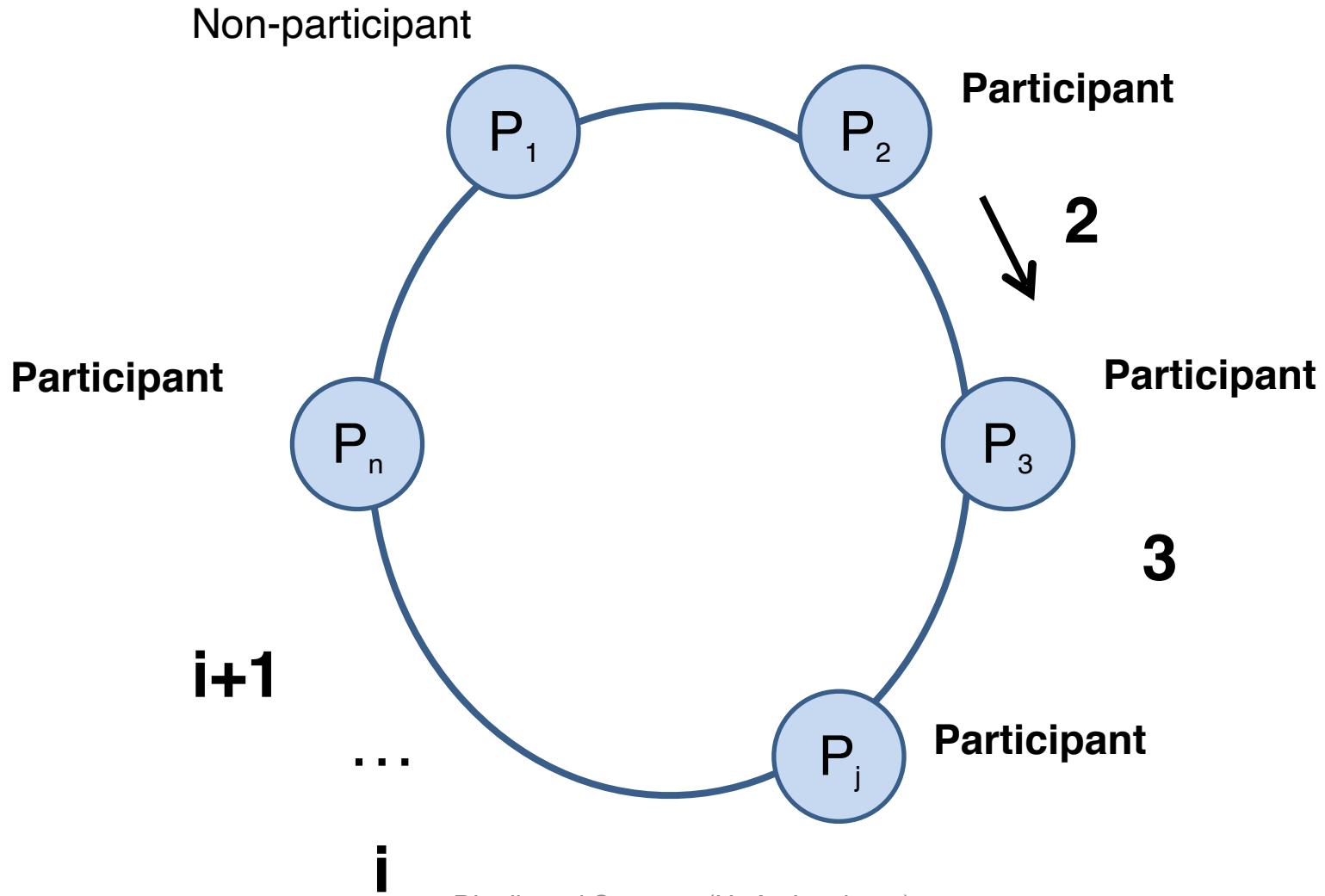
Ring-based algorithm: Calling an Election (determine winner)



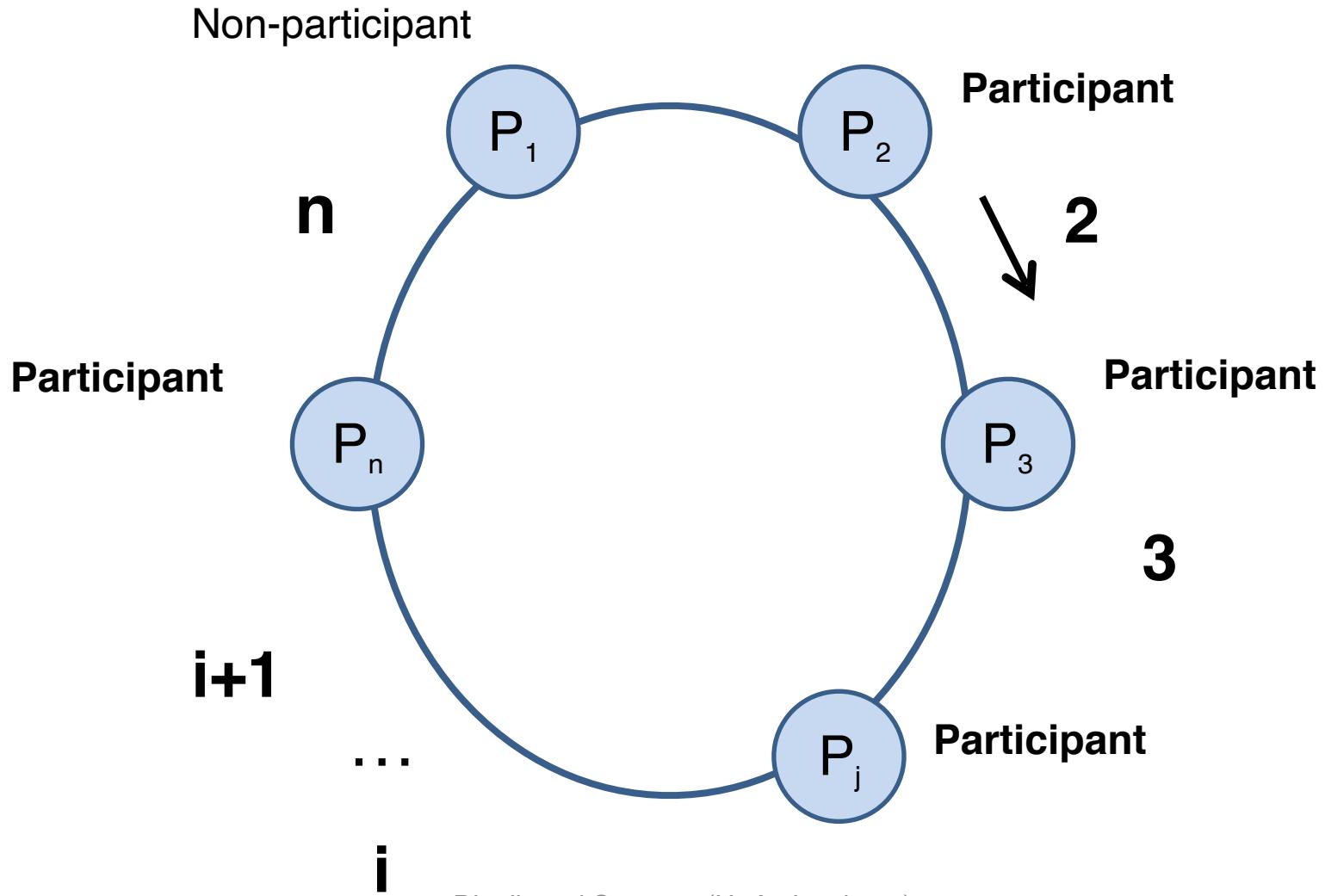
Ring-based algorithm: Calling an Election (determine winner)



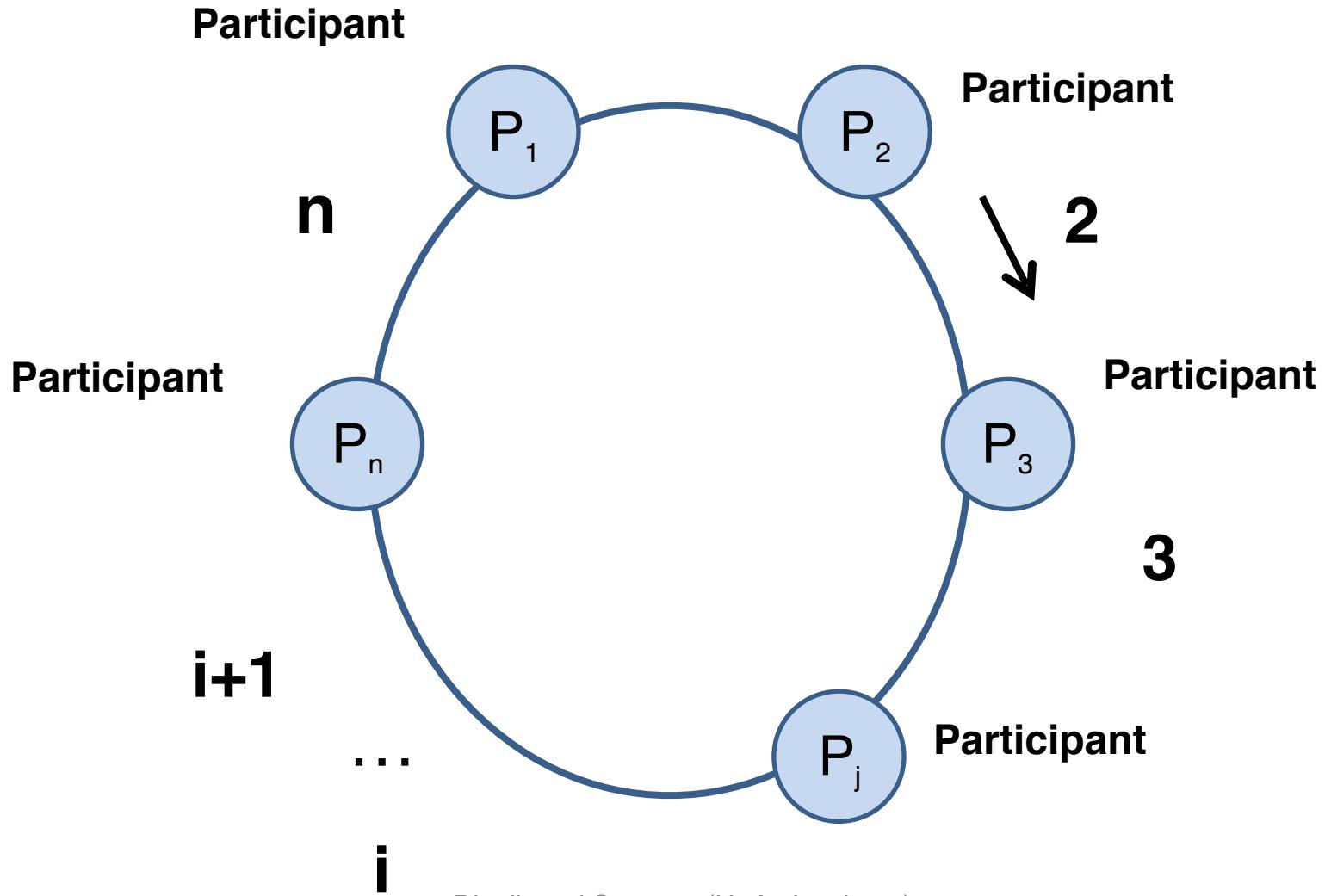
Ring-based algorithm: Calling an Election (determine winner)



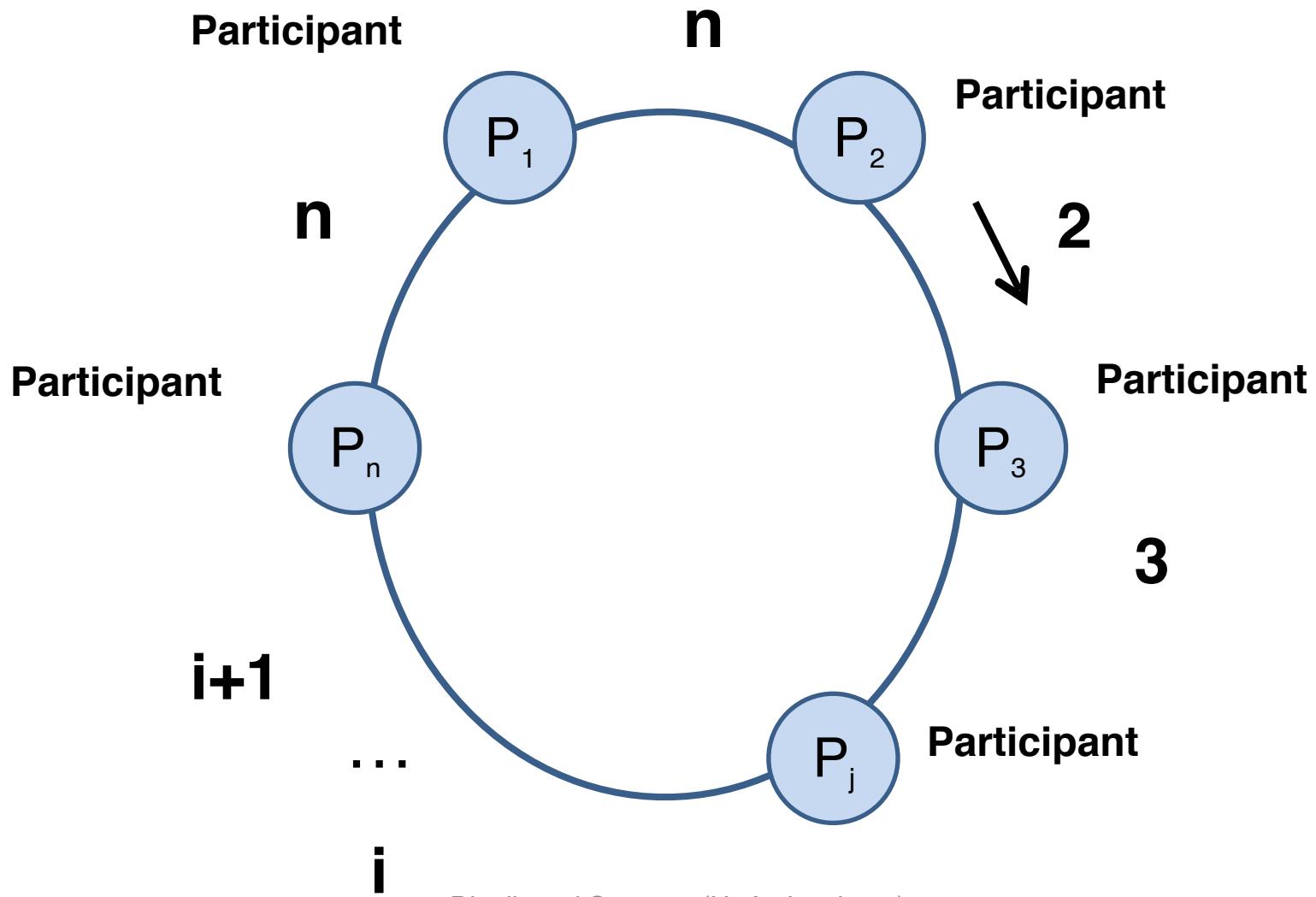
Ring-based algorithm: Calling an Election (determine winner)



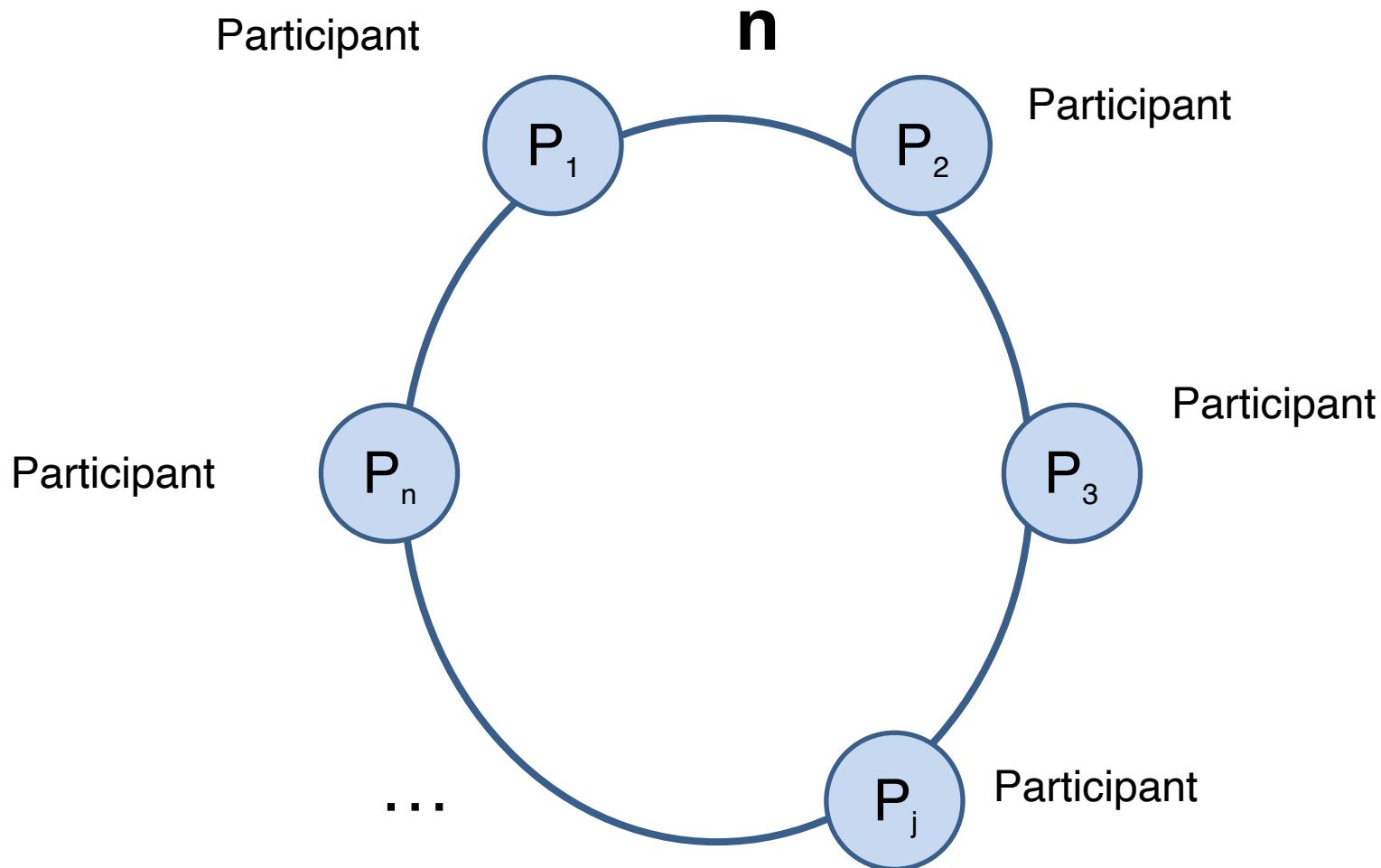
Ring-based algorithm: Calling an Election (determine winner)



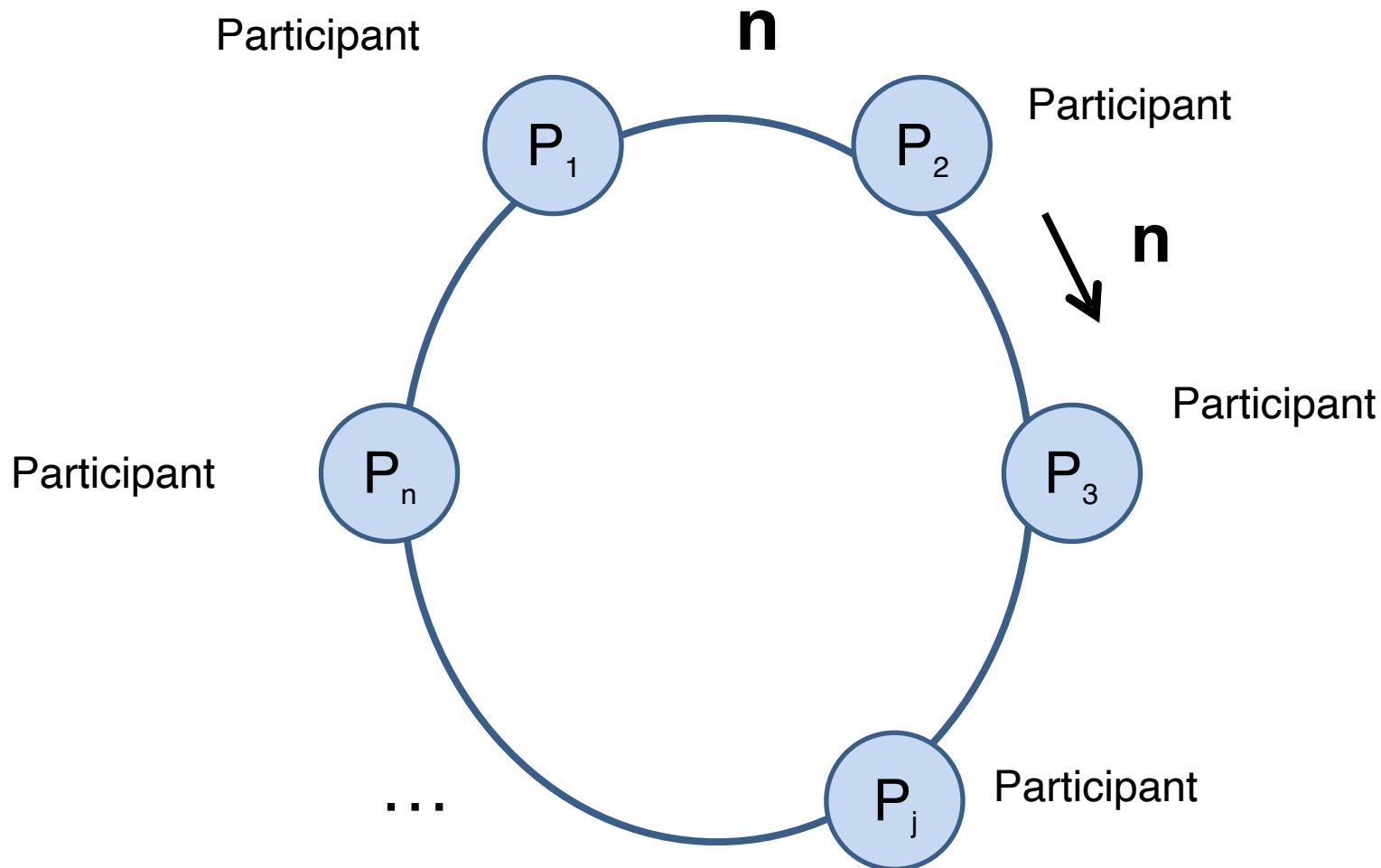
Ring-based algorithm: Calling an Election (determine winner)



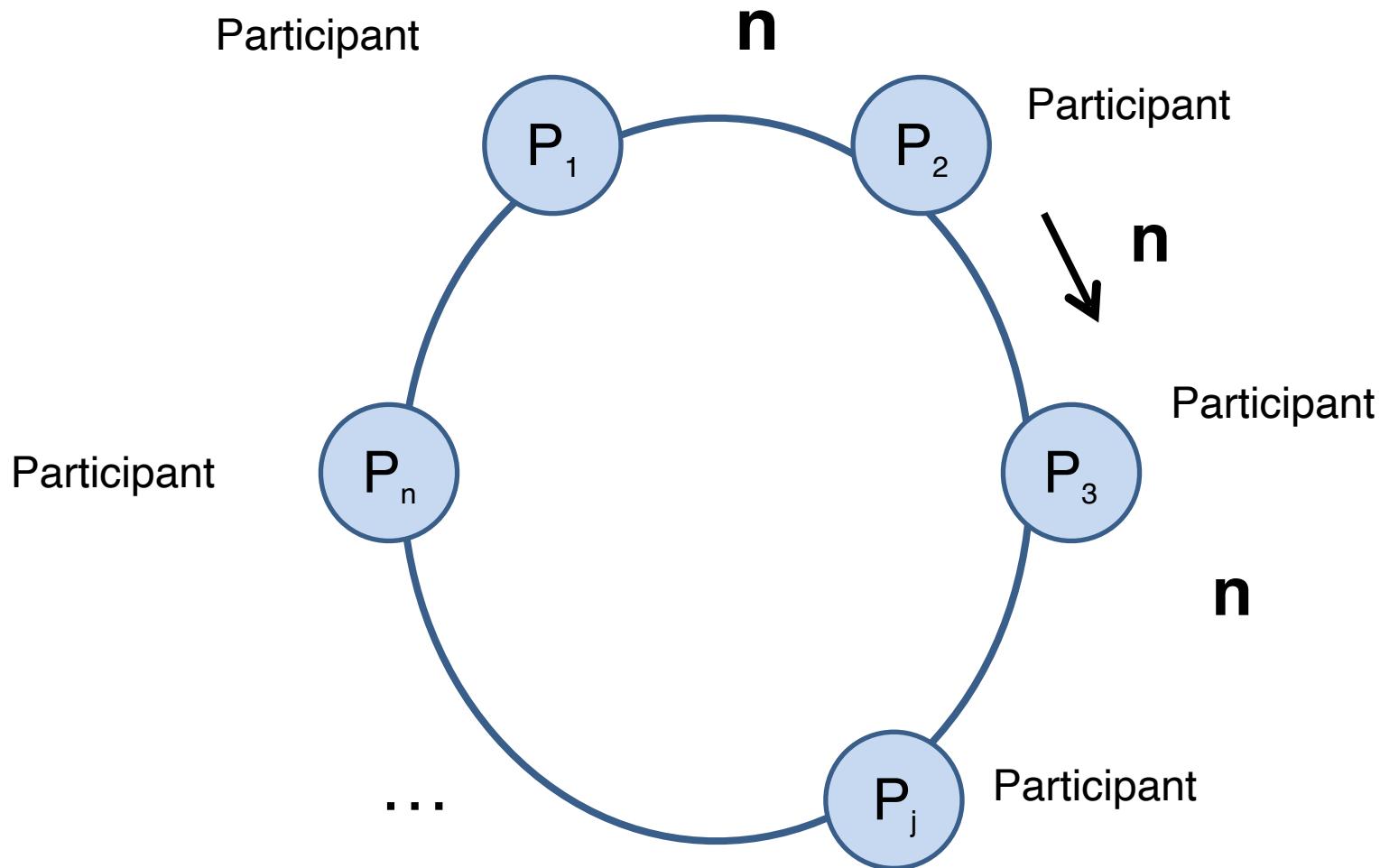
Ring-based algorithm: Calling an Election (origin & victory)



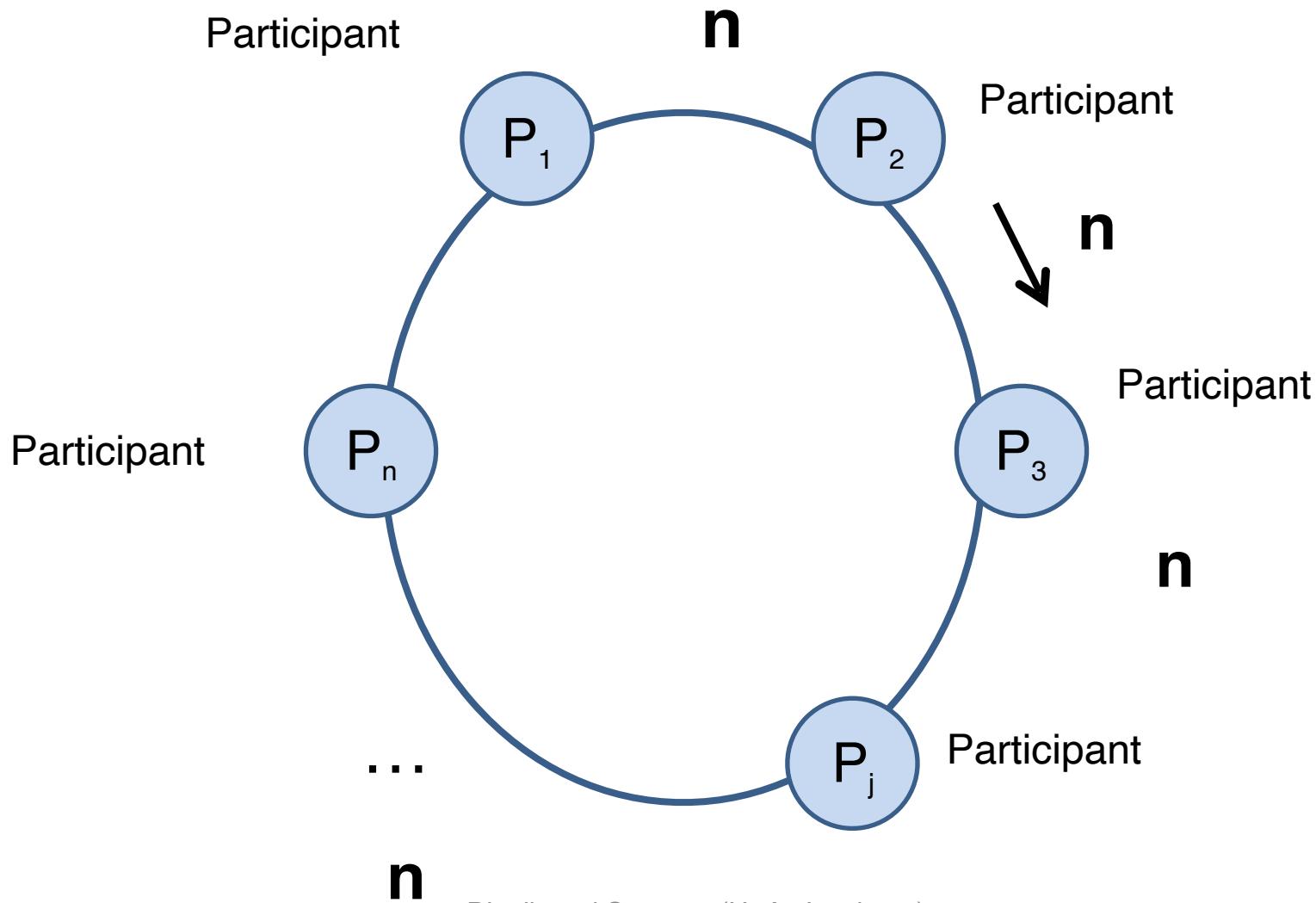
Ring-based algorithm: Calling an Election (origin & victory)



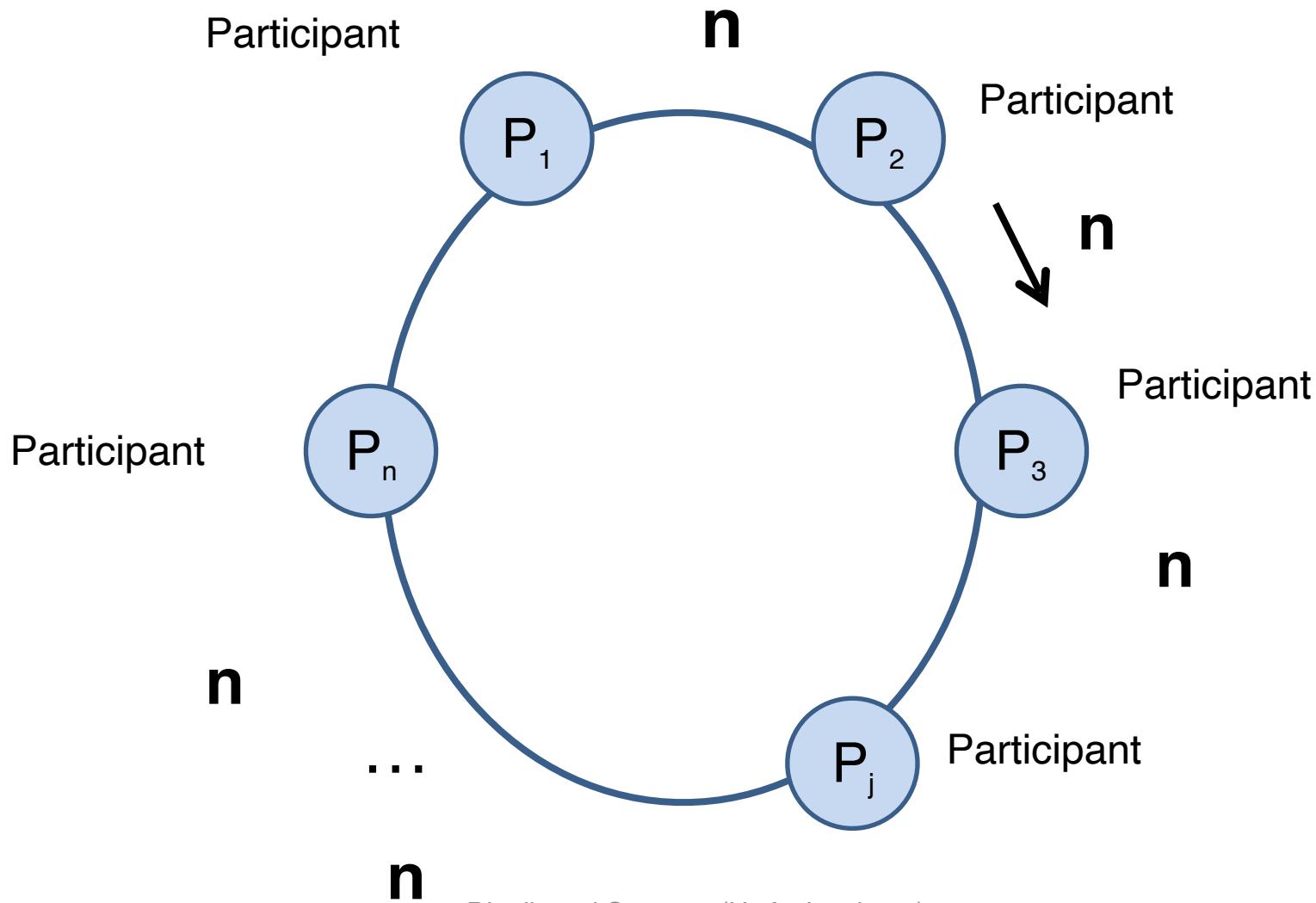
Ring-based algorithm: Calling an Election (origin & victory)



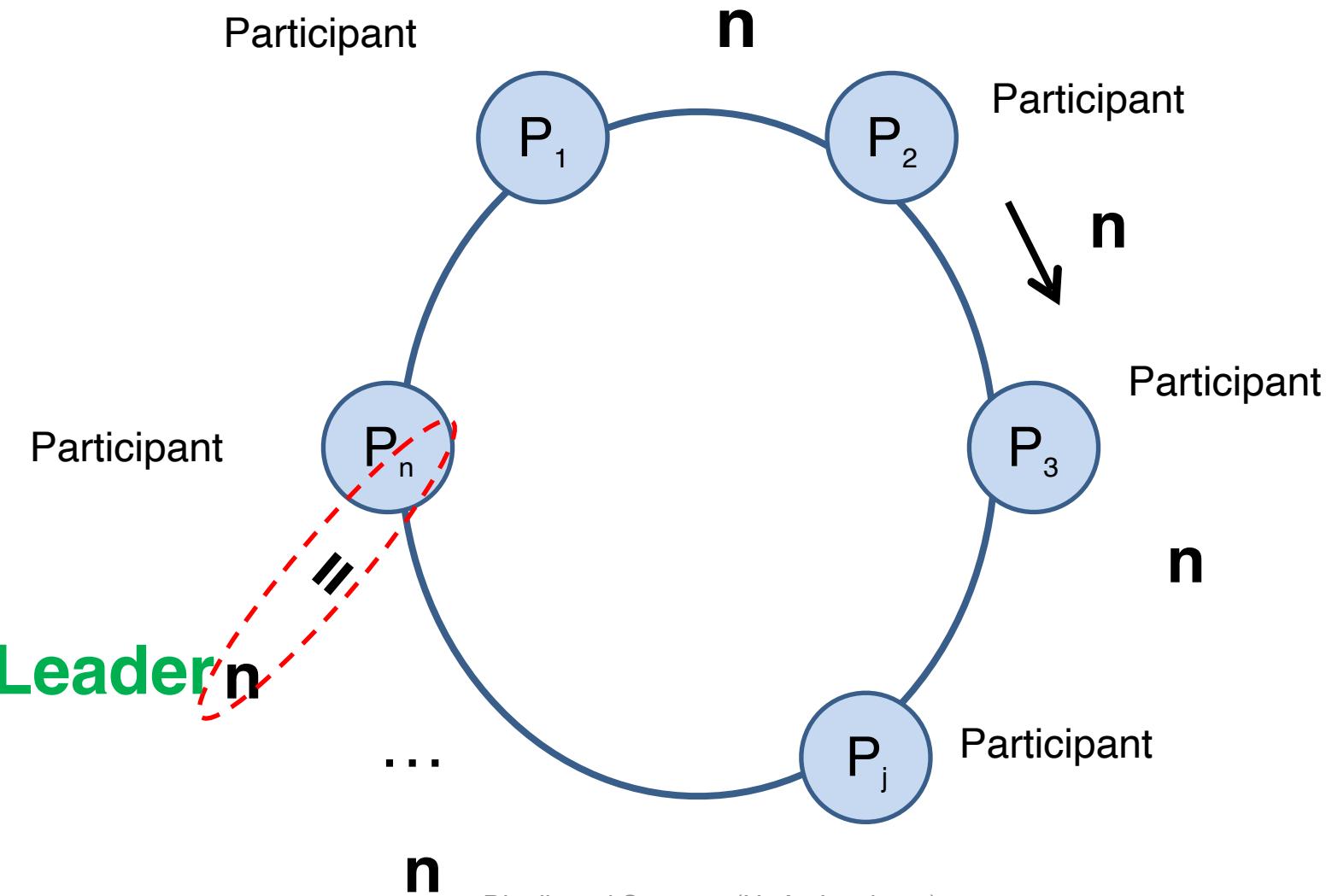
Ring-based algorithm: Calling an Election (origin & victory)



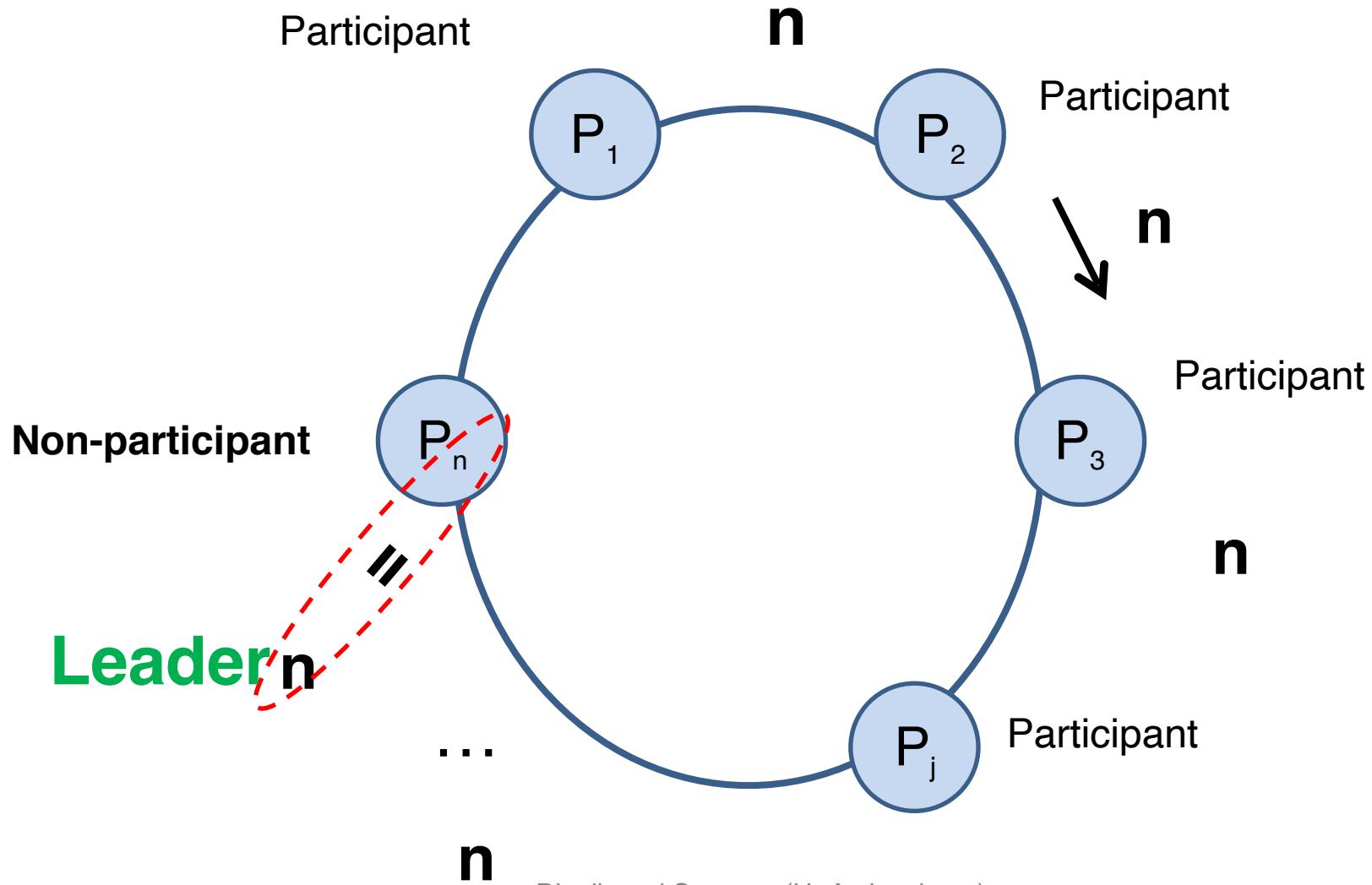
Ring-based algorithm: Calling an Election (origin & victory)



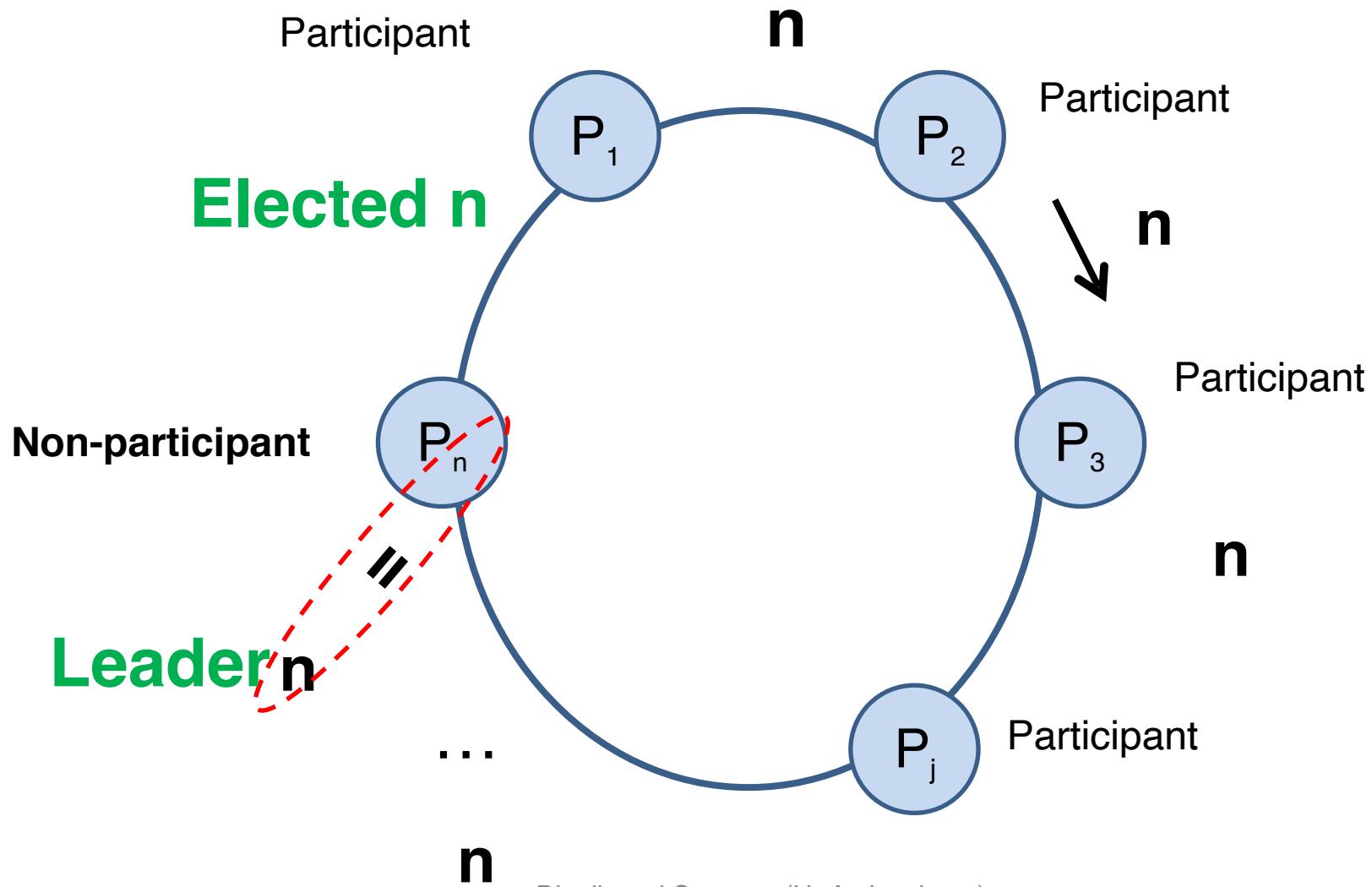
Ring-based algorithm: Calling an Election (origin & victory)



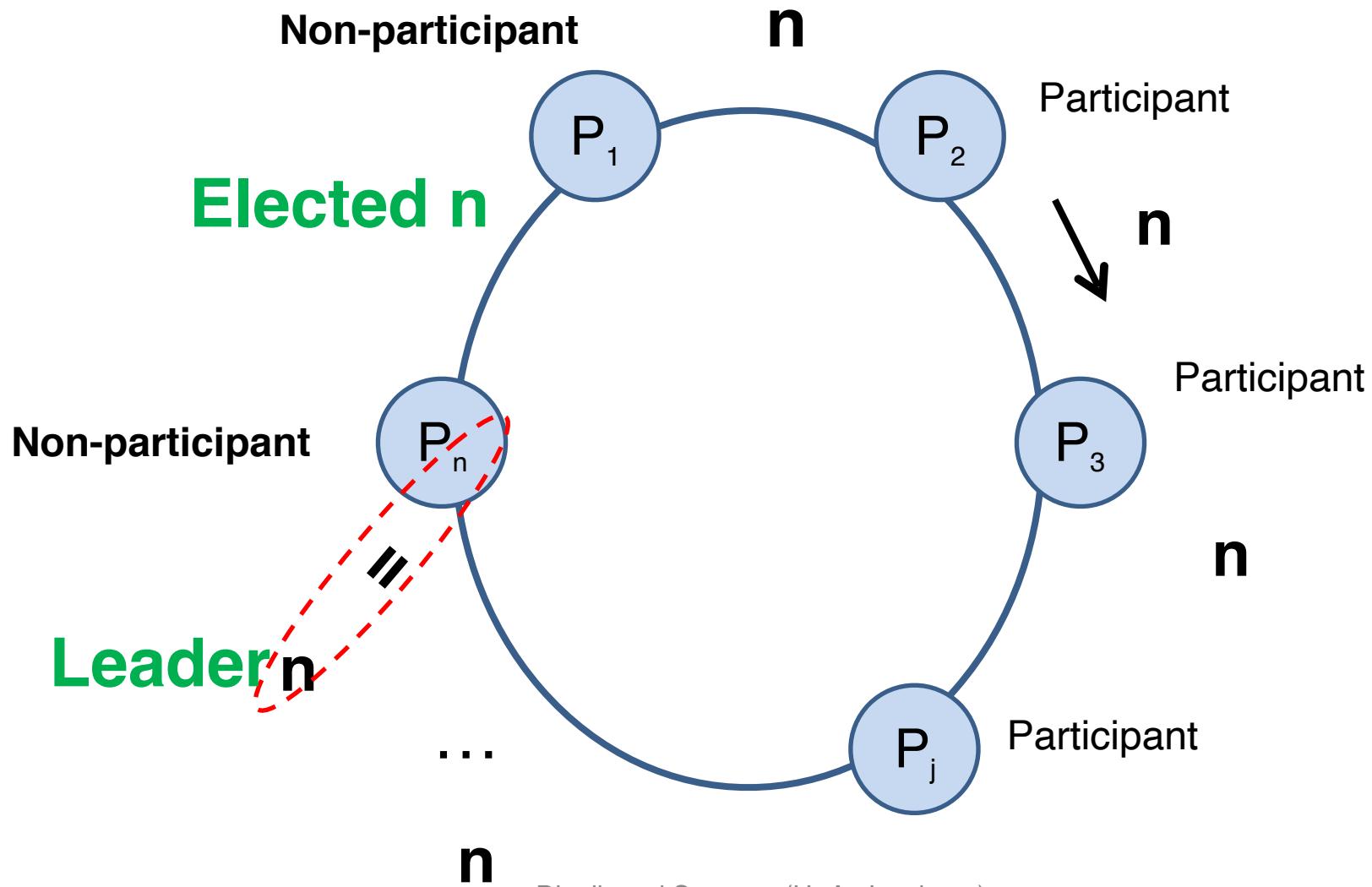
Ring-based algorithm: Calling an Election (origin & victory)



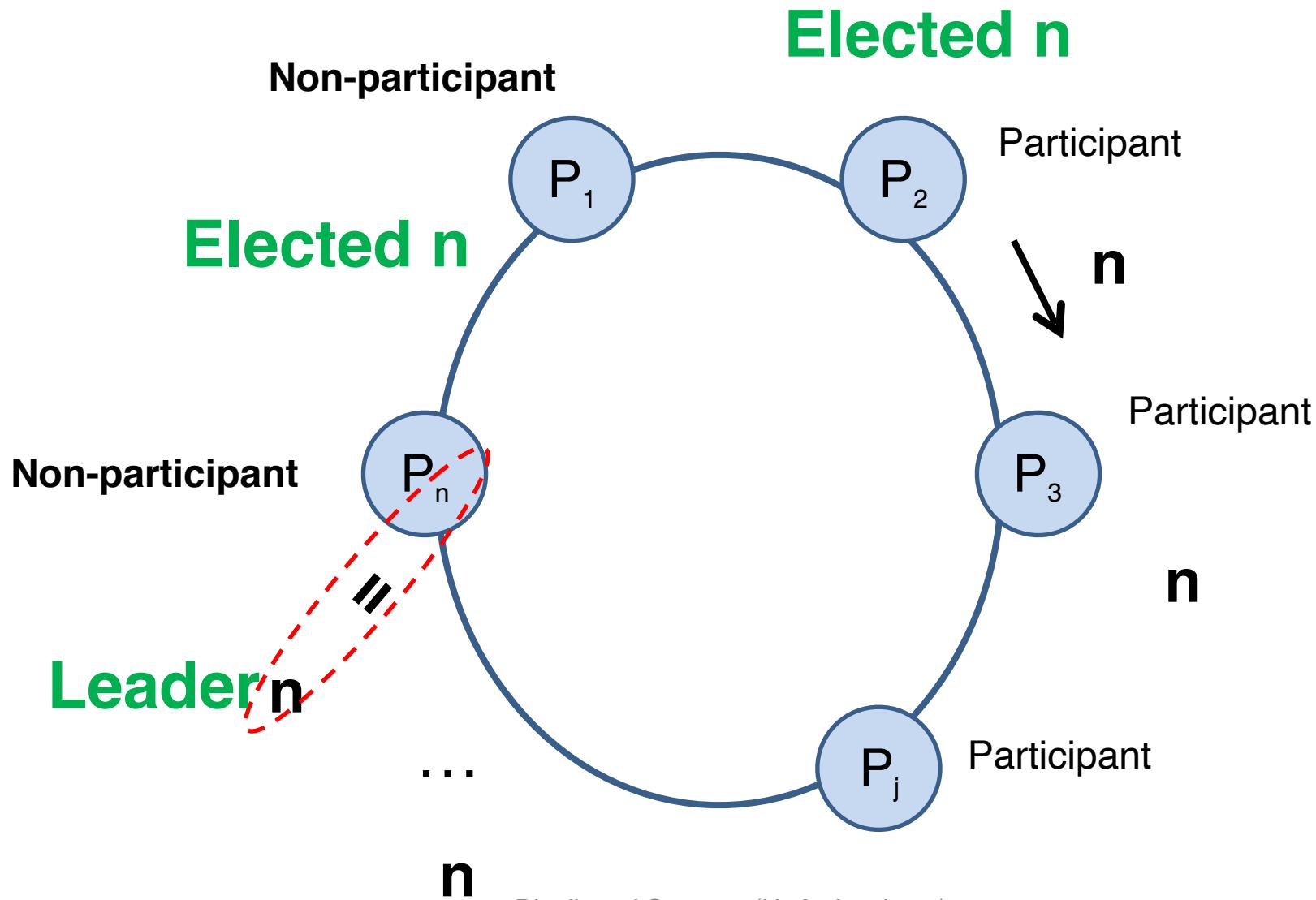
Ring-based algorithm: Calling an Election (origin & victory)



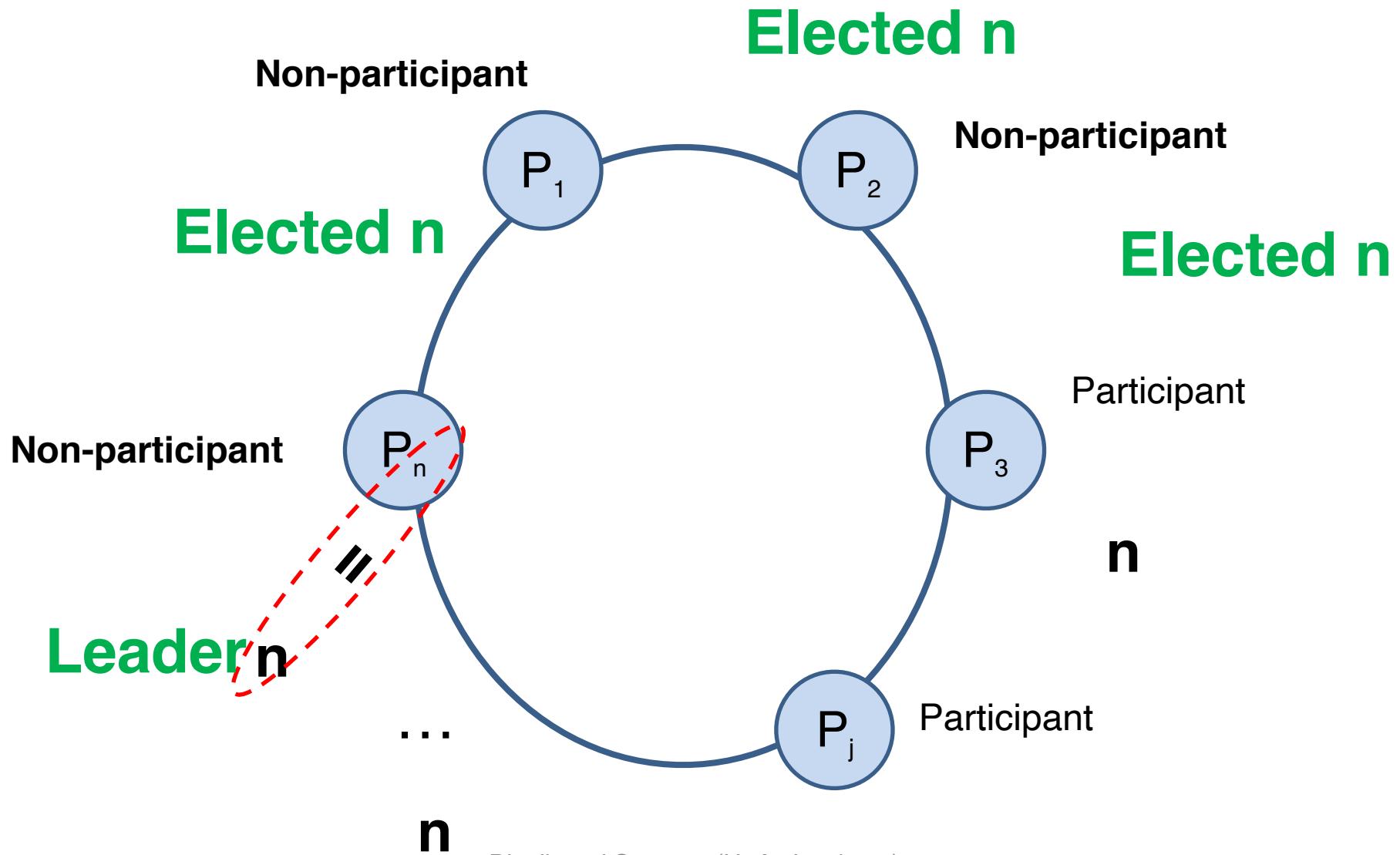
Ring-based algorithm: Calling an Election (origin & victory)



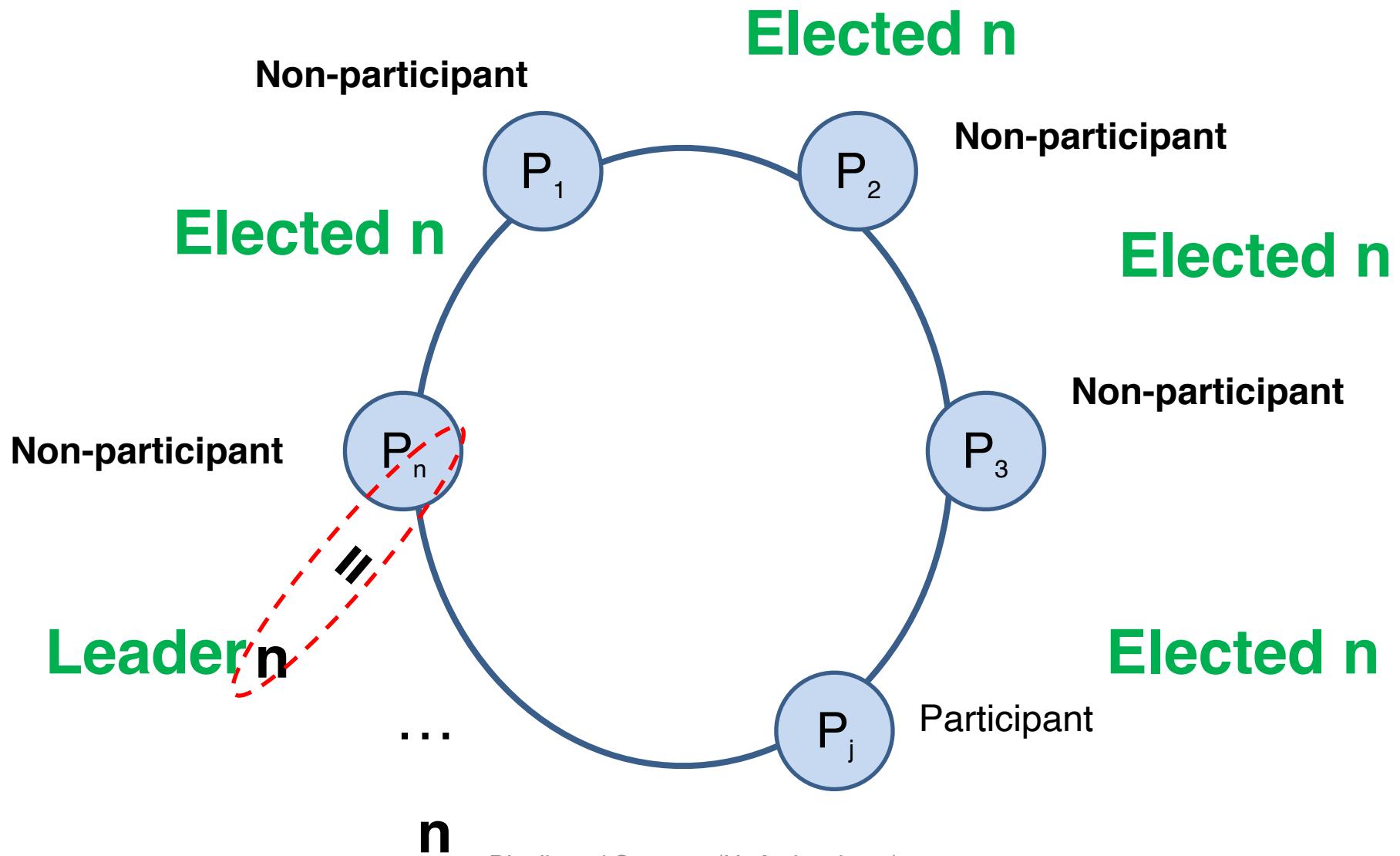
Ring-based algorithm: Calling an Election (origin & victory)



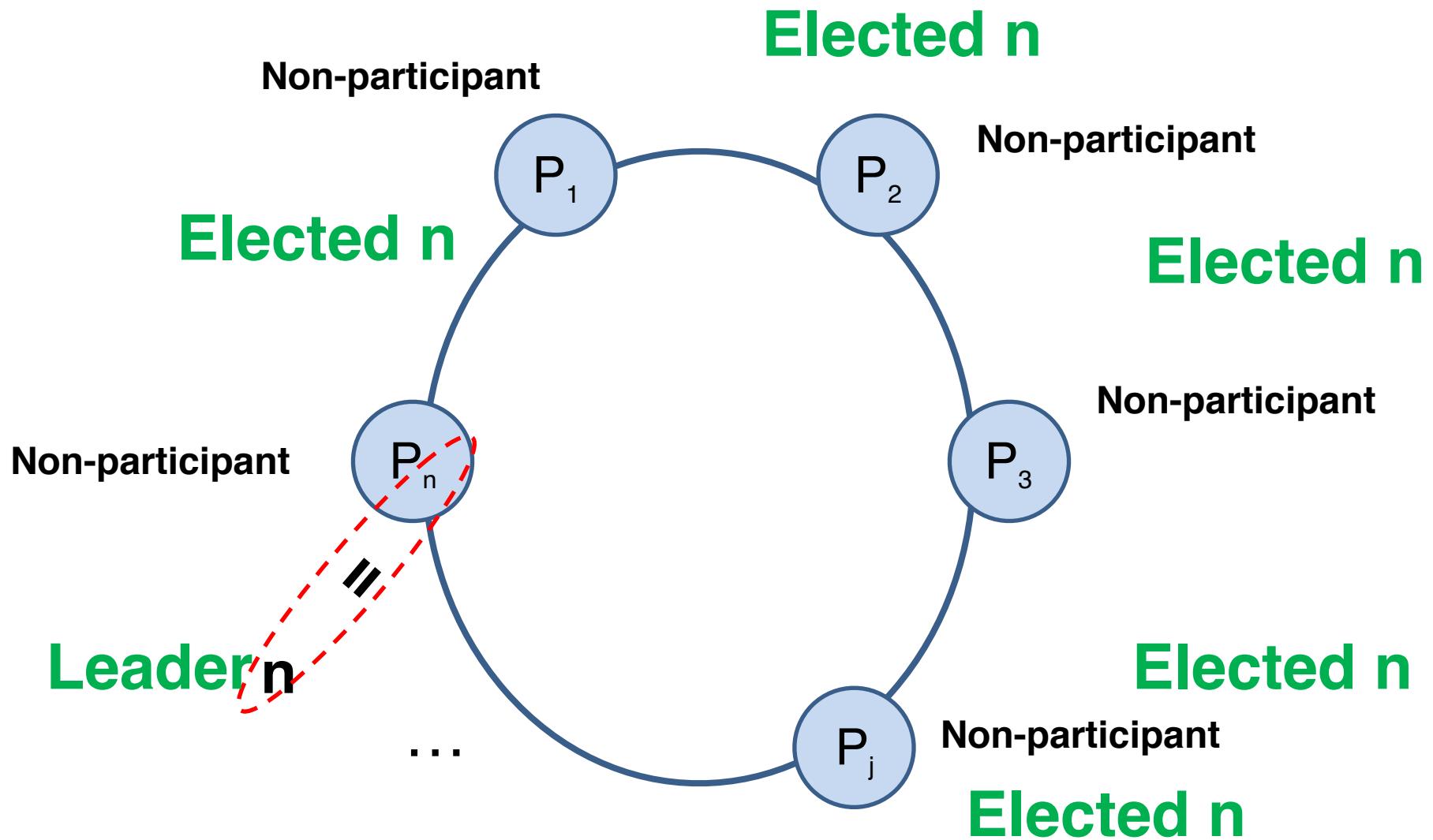
Ring-based algorithm: Calling an Election (origin & victory)



Ring-based algorithm: Calling an Election (origin & victory)



Ring-based algorithm: Calling an Election (origin & victory)



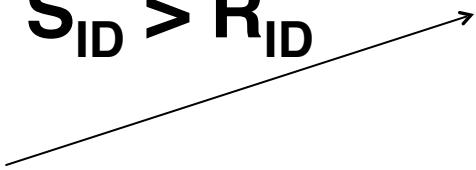
Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant



Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant
Forward S_{ID}



Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant
Forward S_{ID}

$$S_{ID} > R_{ID}$$

Non-participant

Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant
Forward S_{ID}

$$S_{ID} > R_{ID}$$

Non-participant
Forward S_{ID}

Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant
Forward S_{ID}

$$S_{ID} > R_{ID}$$

Non-participant
Forward S_{ID}

$$S_{ID} < R_{ID}$$

Non-participant

Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant
Forward S_{ID}

$$S_{ID} > R_{ID}$$

Non-participant
Forward S_{ID}

$$S_{ID} < R_{ID}$$

Non-participant
Forward R_{ID}

Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant
Forward S_{ID}

$$S_{ID} > R_{ID}$$

Non-participant
Forward S_{ID}

$$S_{ID} < R_{ID}$$

Participant

$$S_{ID} < R_{ID}$$

Non-participant
Forward R_{ID}

Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant
Forward S_{ID}

$$S_{ID} > R_{ID}$$

Non-participant
Forward S_{ID}

$$S_{ID} < R_{ID}$$

Participant
No forwarding (own ID already sent)

$$S_{ID} < R_{ID}$$

Non-participant
Forward R_{ID}

Different cases

S (sender)

$$S_{ID} > R_{ID}$$

R (receiver)

Participant

Forward S_{ID}

If $S_{ID} = R_{ID}$, it follows
R elected as leader

$$S_{ID} > R_{ID}$$

Non-participant

Forward S_{ID}

$$S_{ID} < R_{ID}$$

Participant

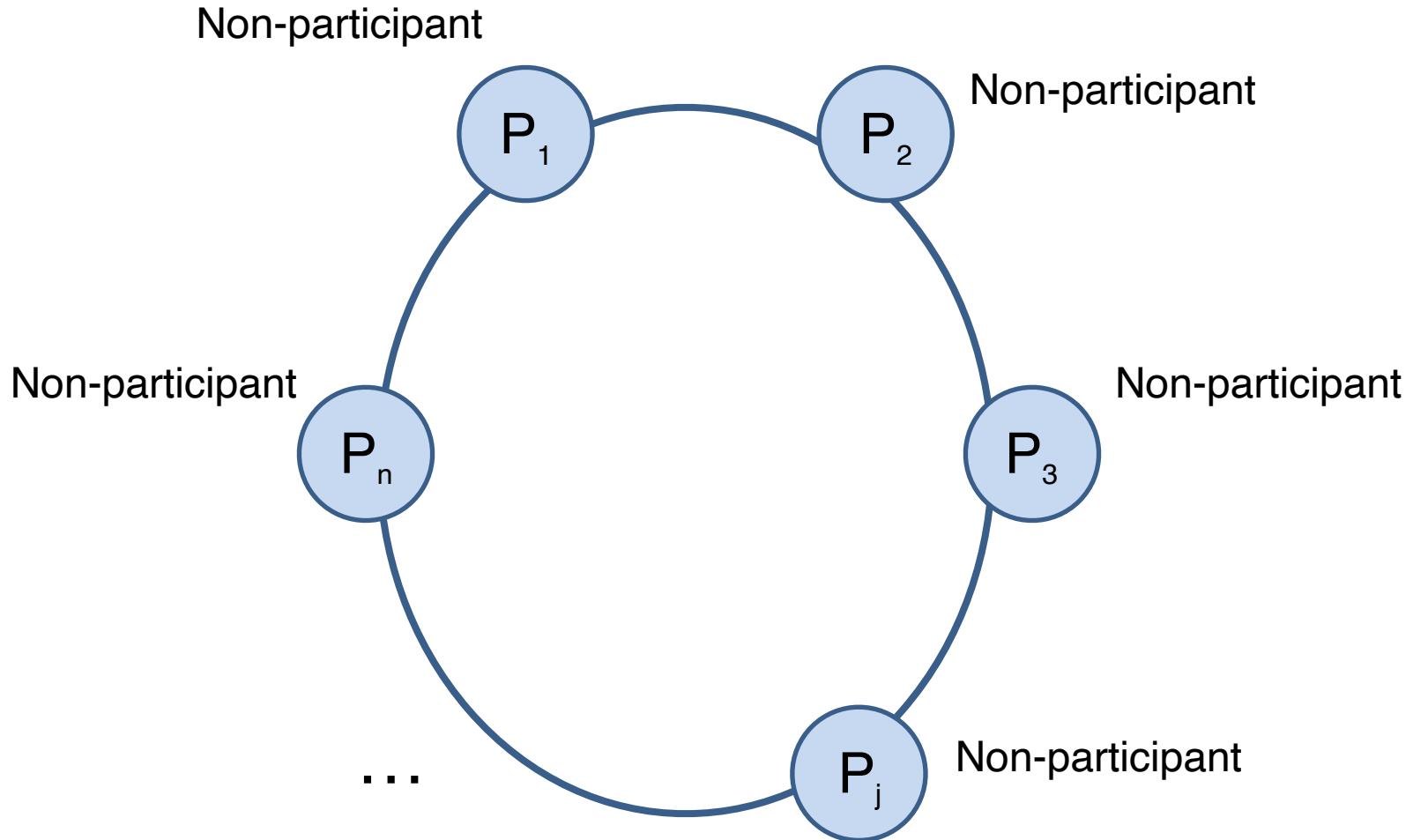
No forwarding (own ID already sent)

$$S_{ID} < R_{ID}$$

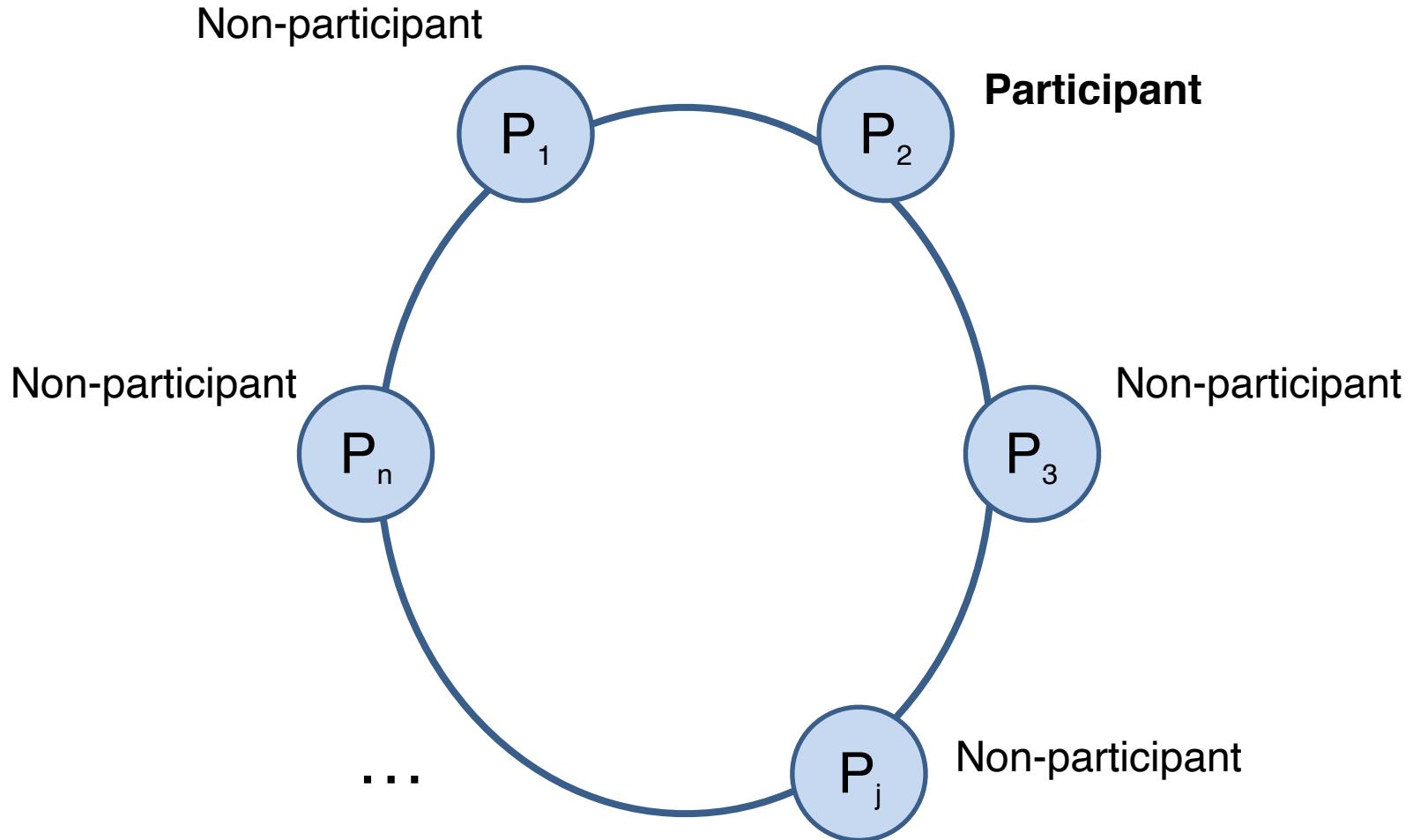
Non-participant

Forward R_{ID}

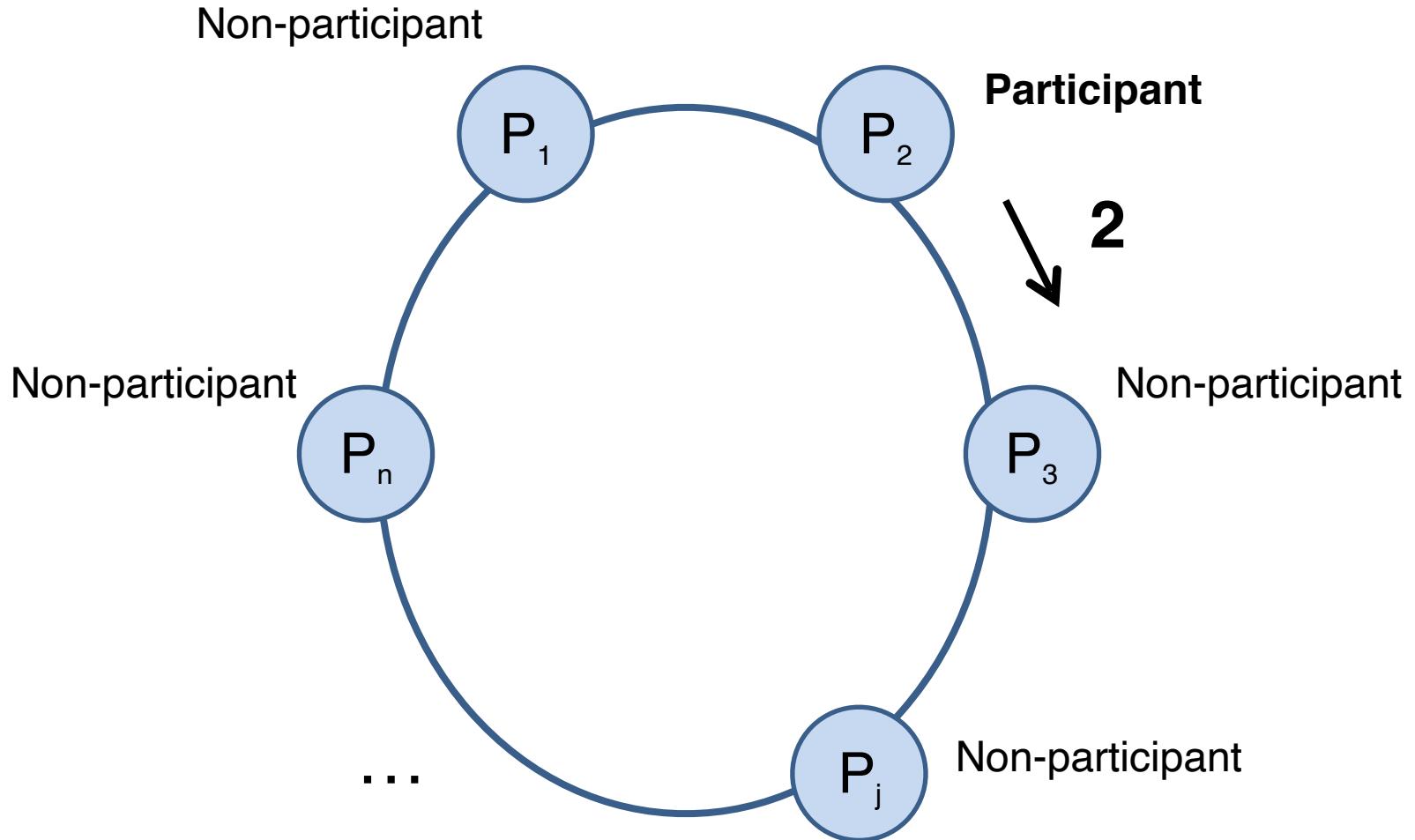
Ring-based algorithm: Concurrent election start



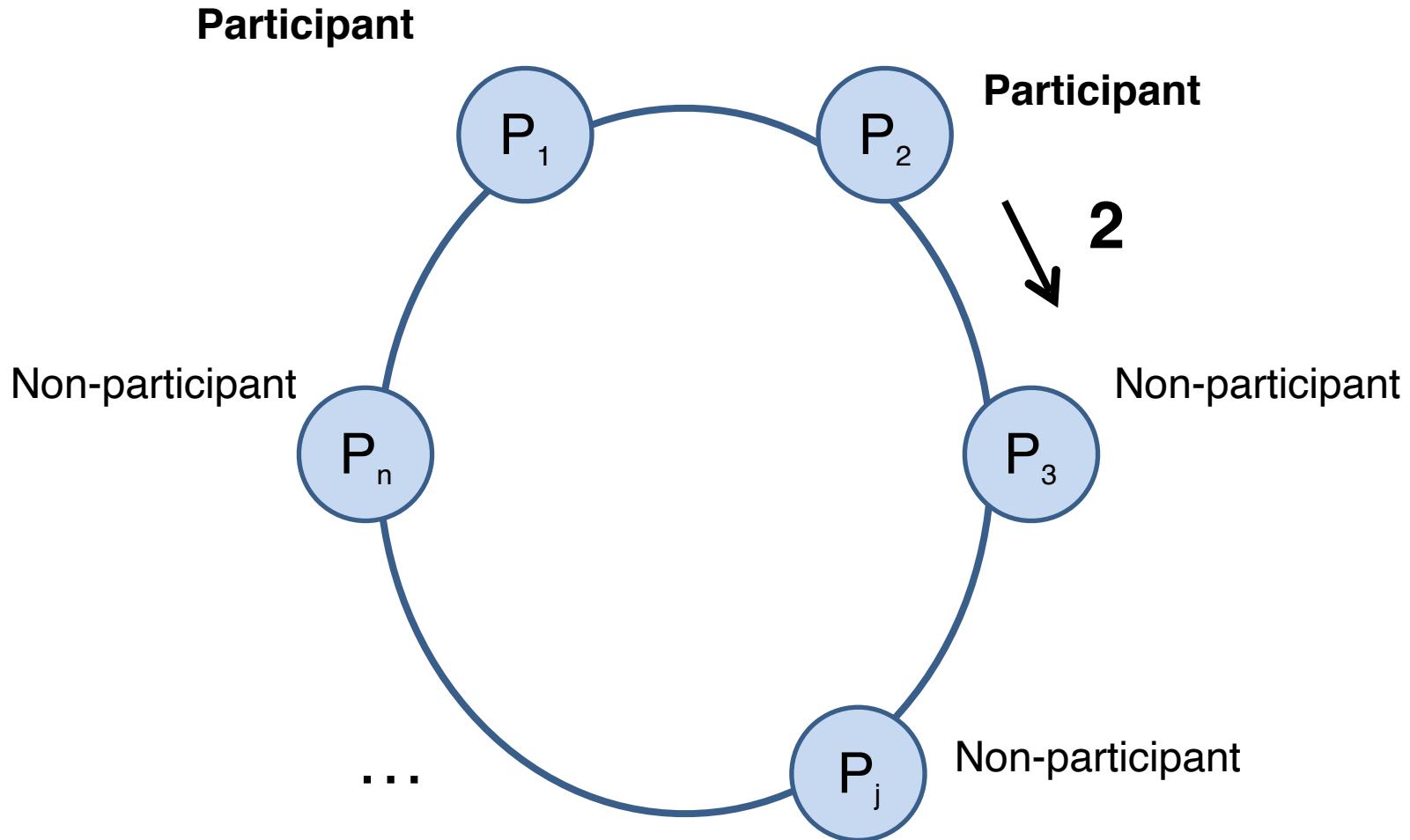
Ring-based algorithm: Concurrent election start



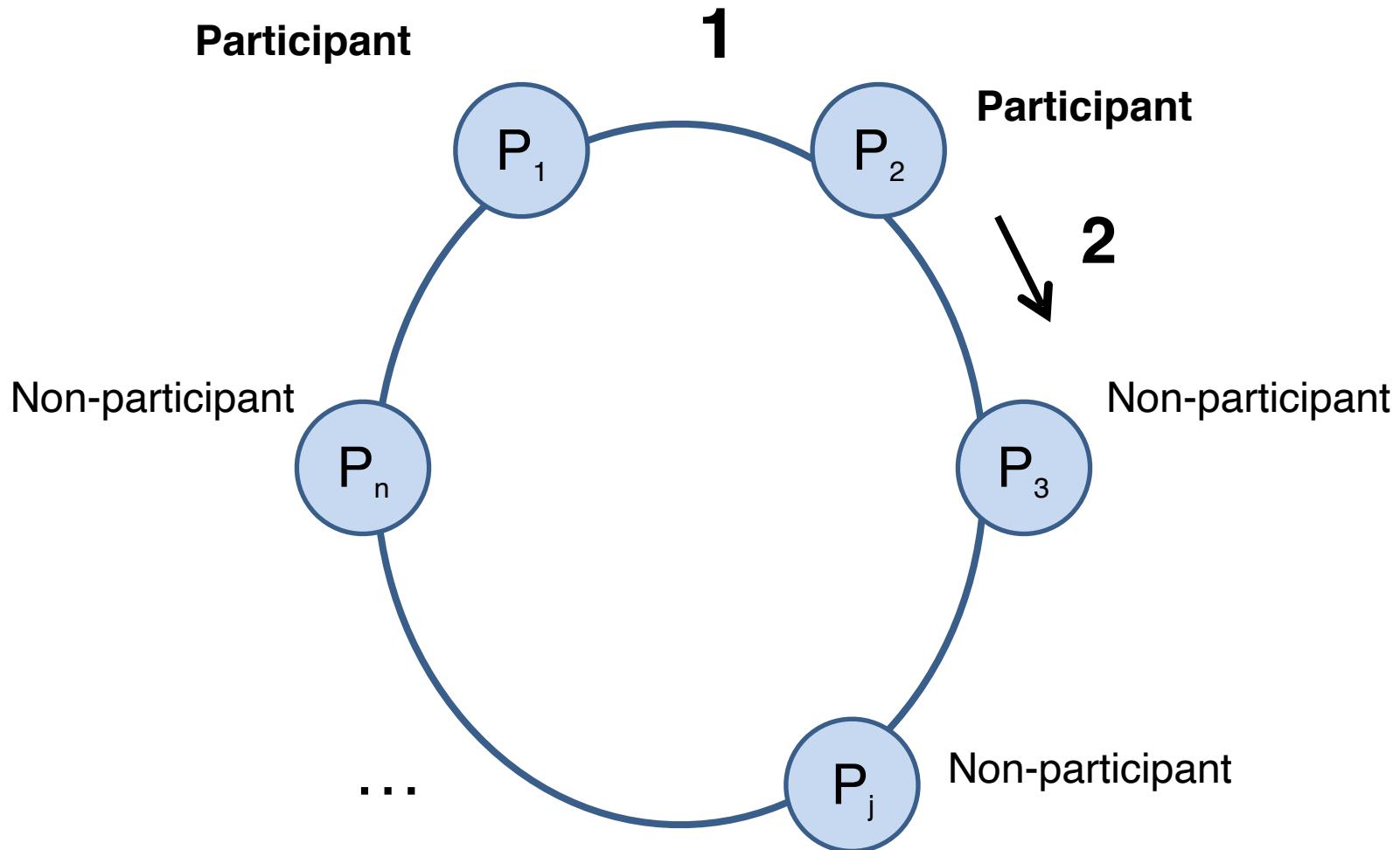
Ring-based algorithm: Concurrent election start



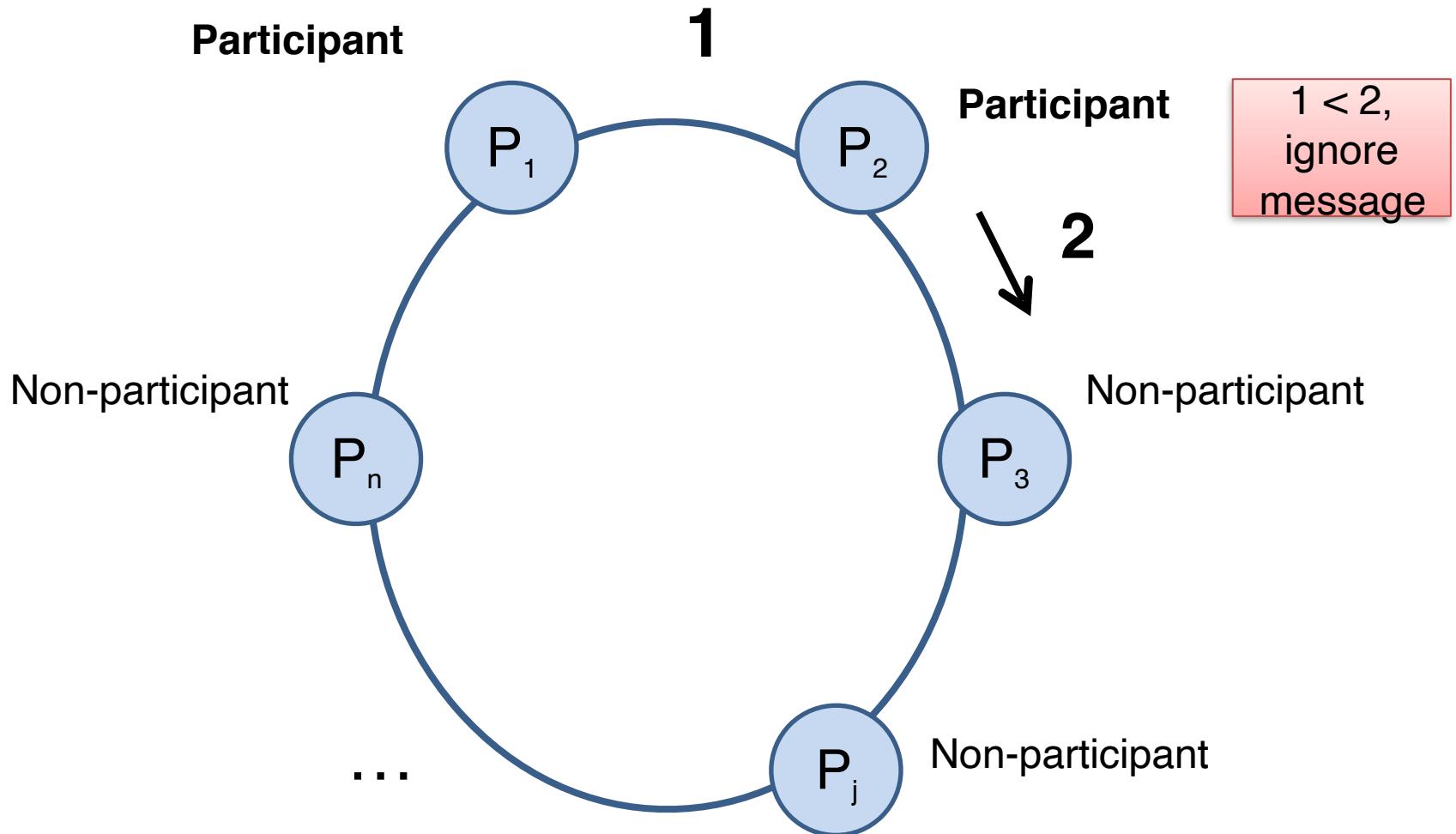
Ring-based algorithm: Concurrent election start



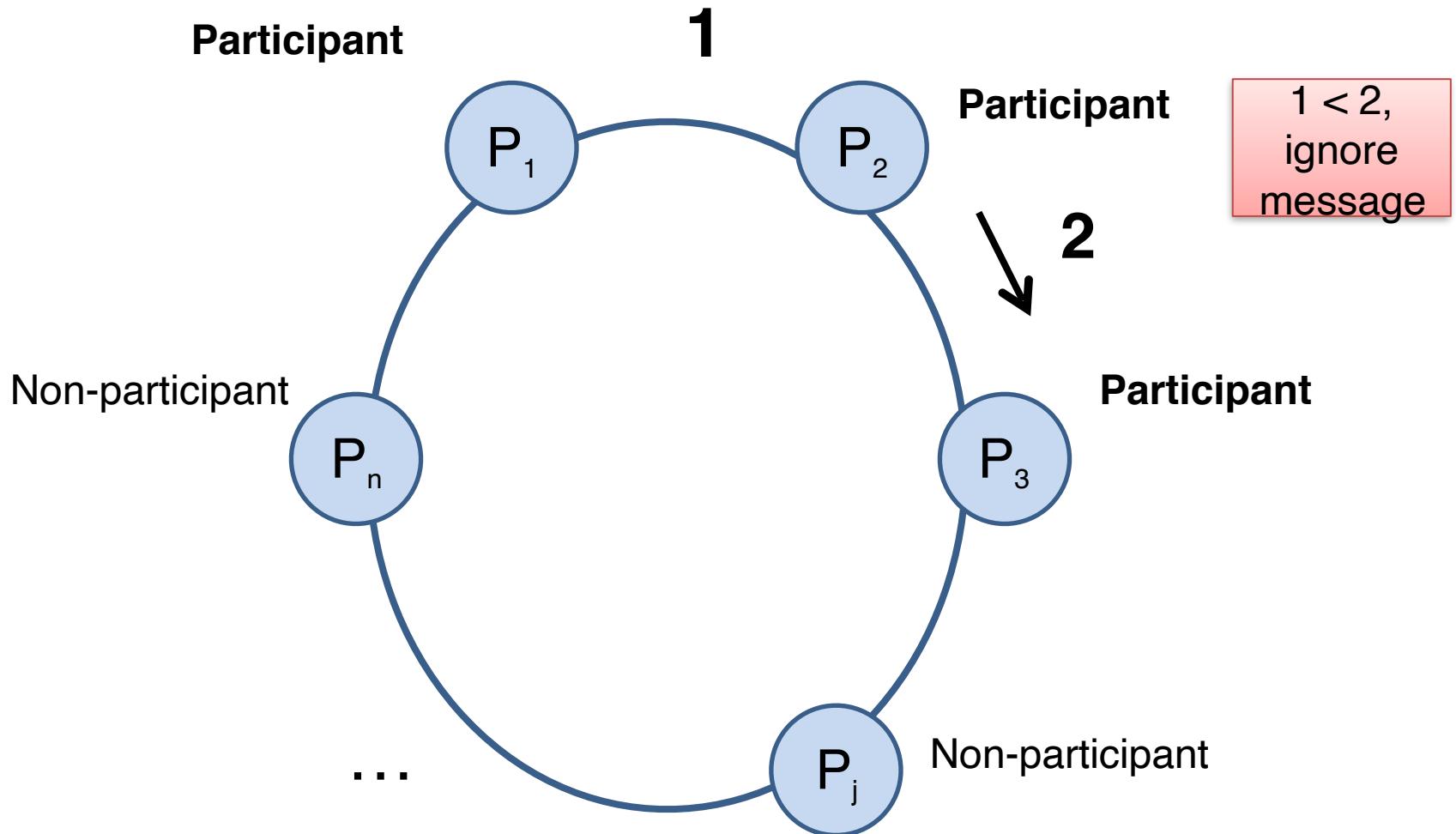
Ring-based algorithm: Concurrent election start



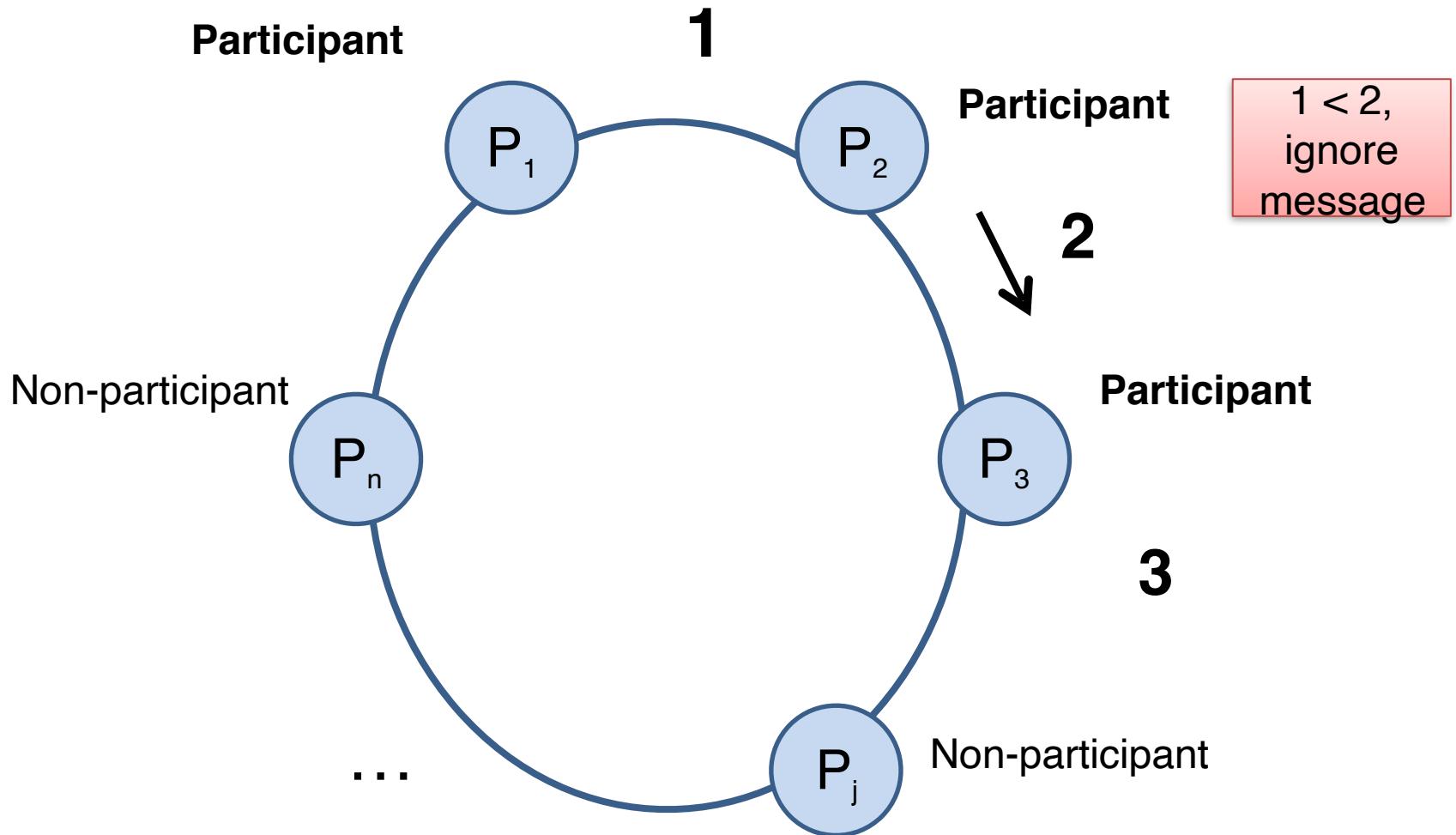
Ring-based algorithm: Concurrent election start



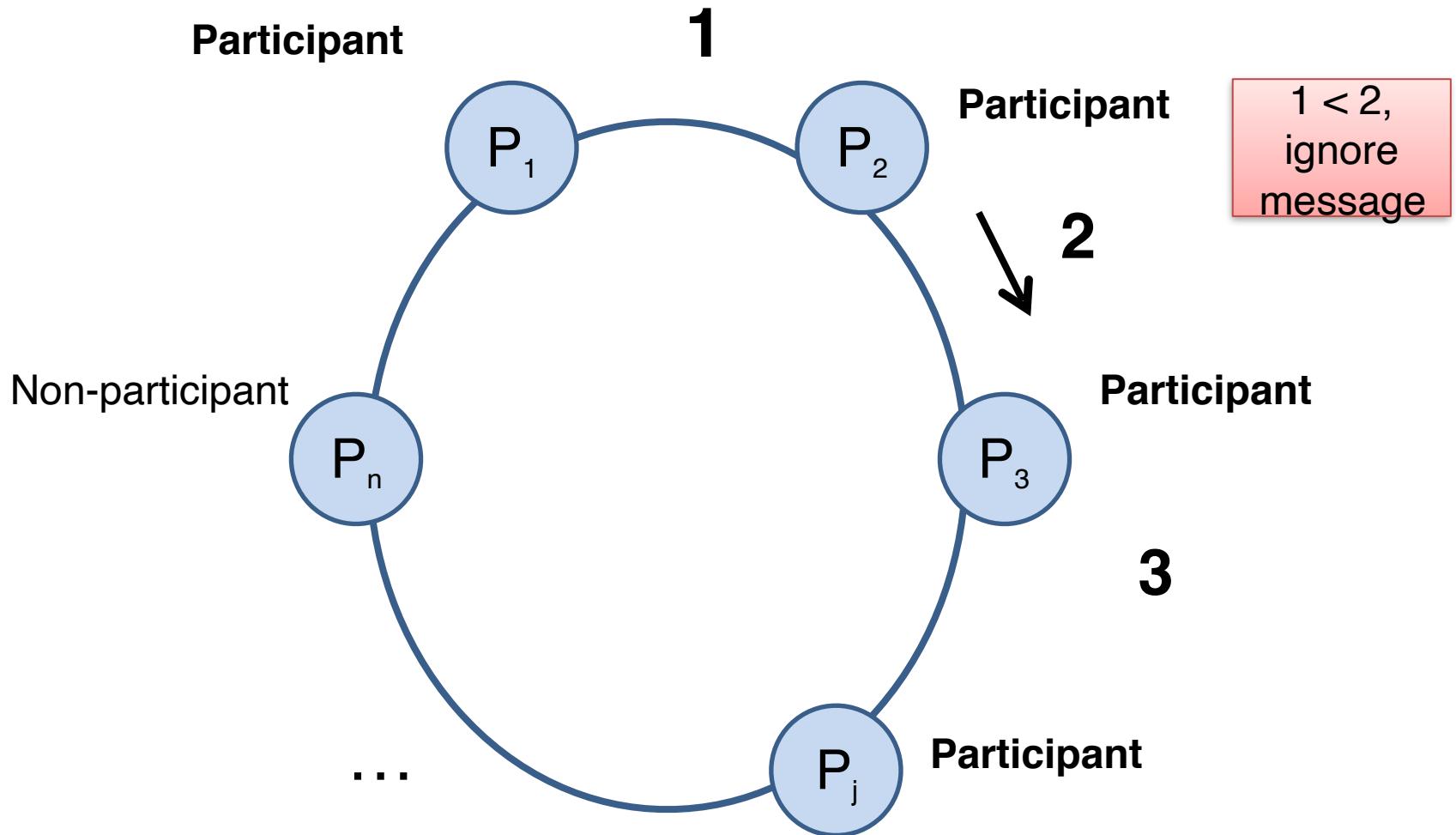
Ring-based algorithm: Concurrent election start



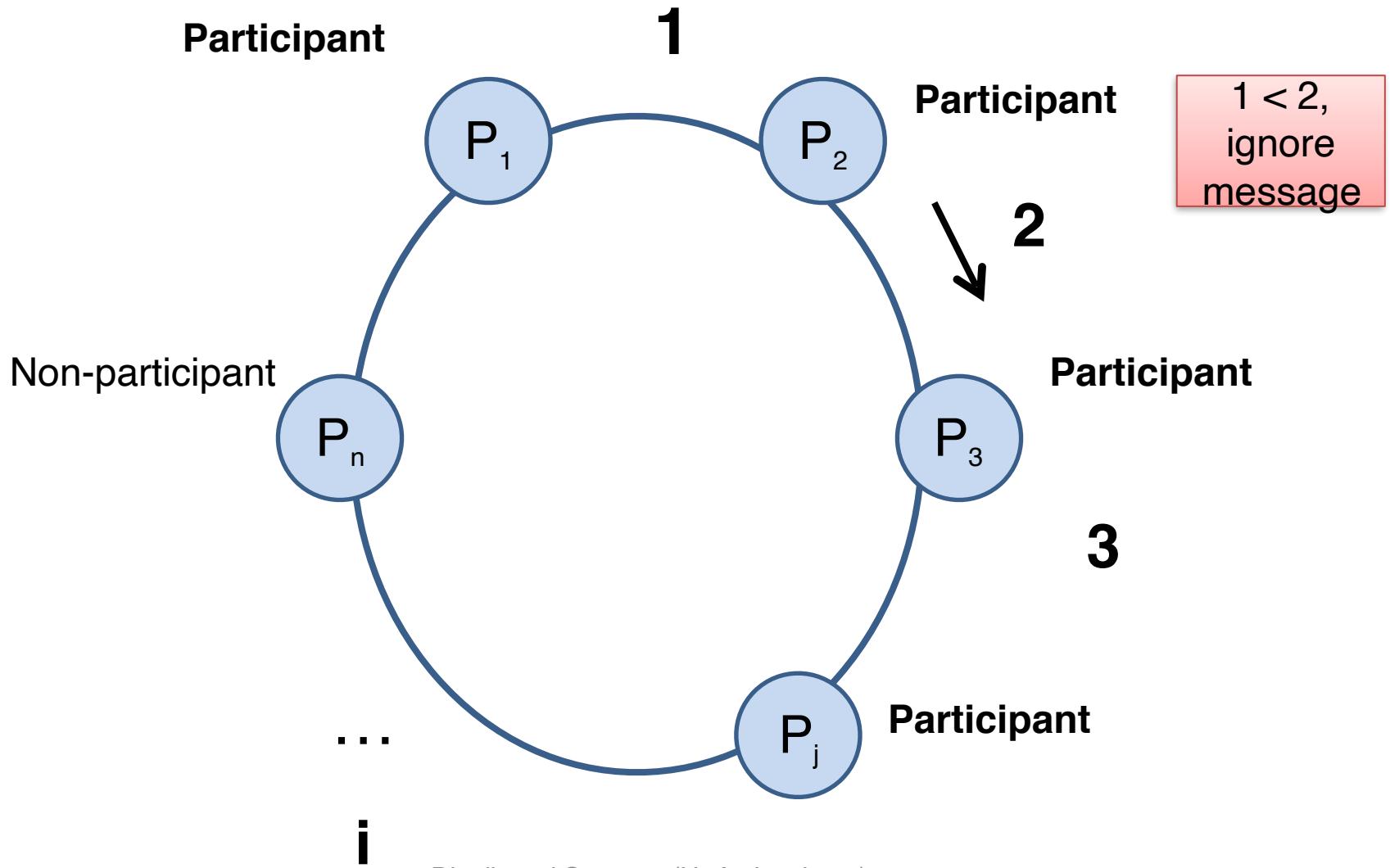
Ring-based algorithm: Concurrent election start



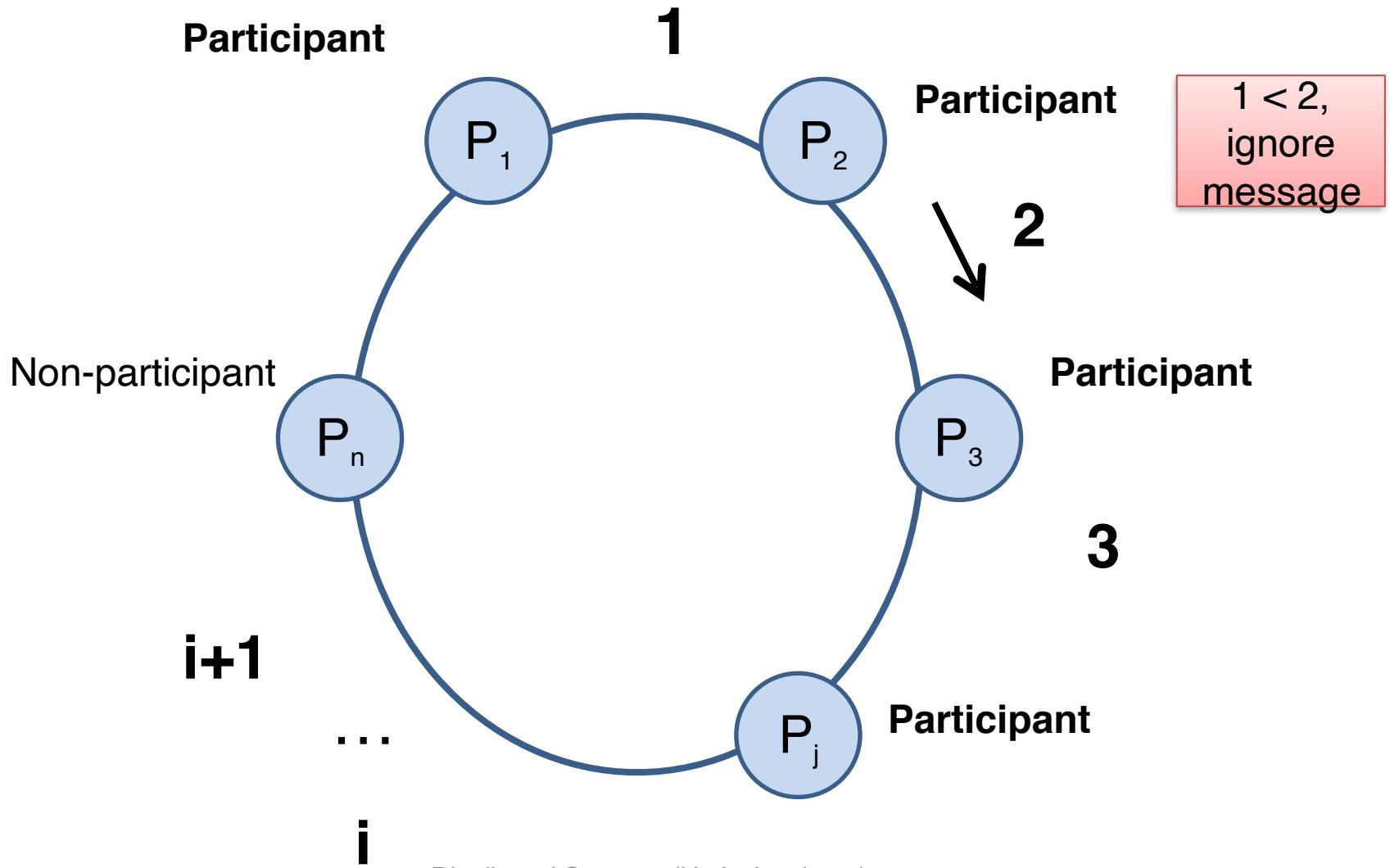
Ring-based algorithm: Concurrent election start



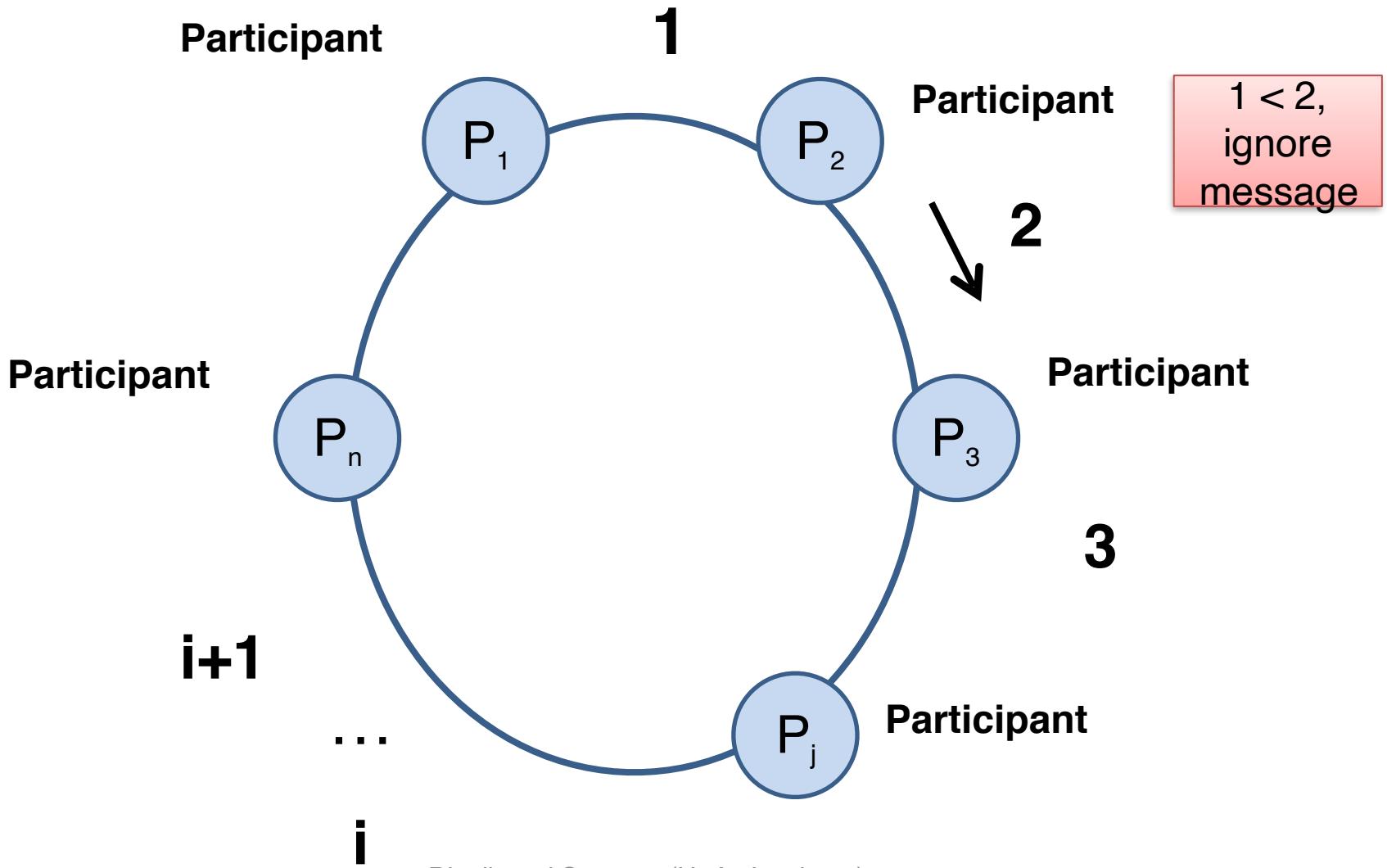
Ring-based algorithm: Concurrent election start



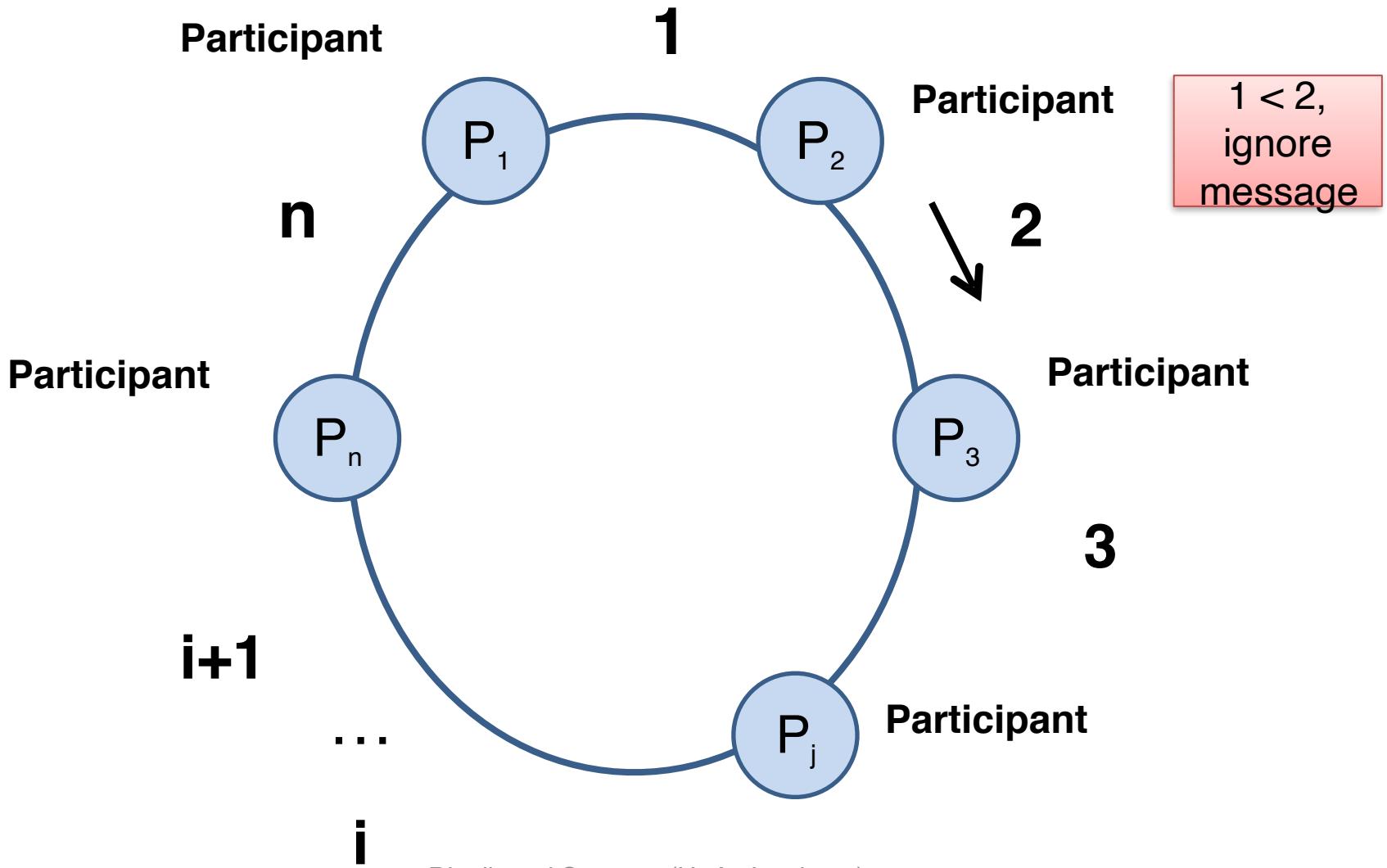
Ring-based algorithm: Concurrent election start



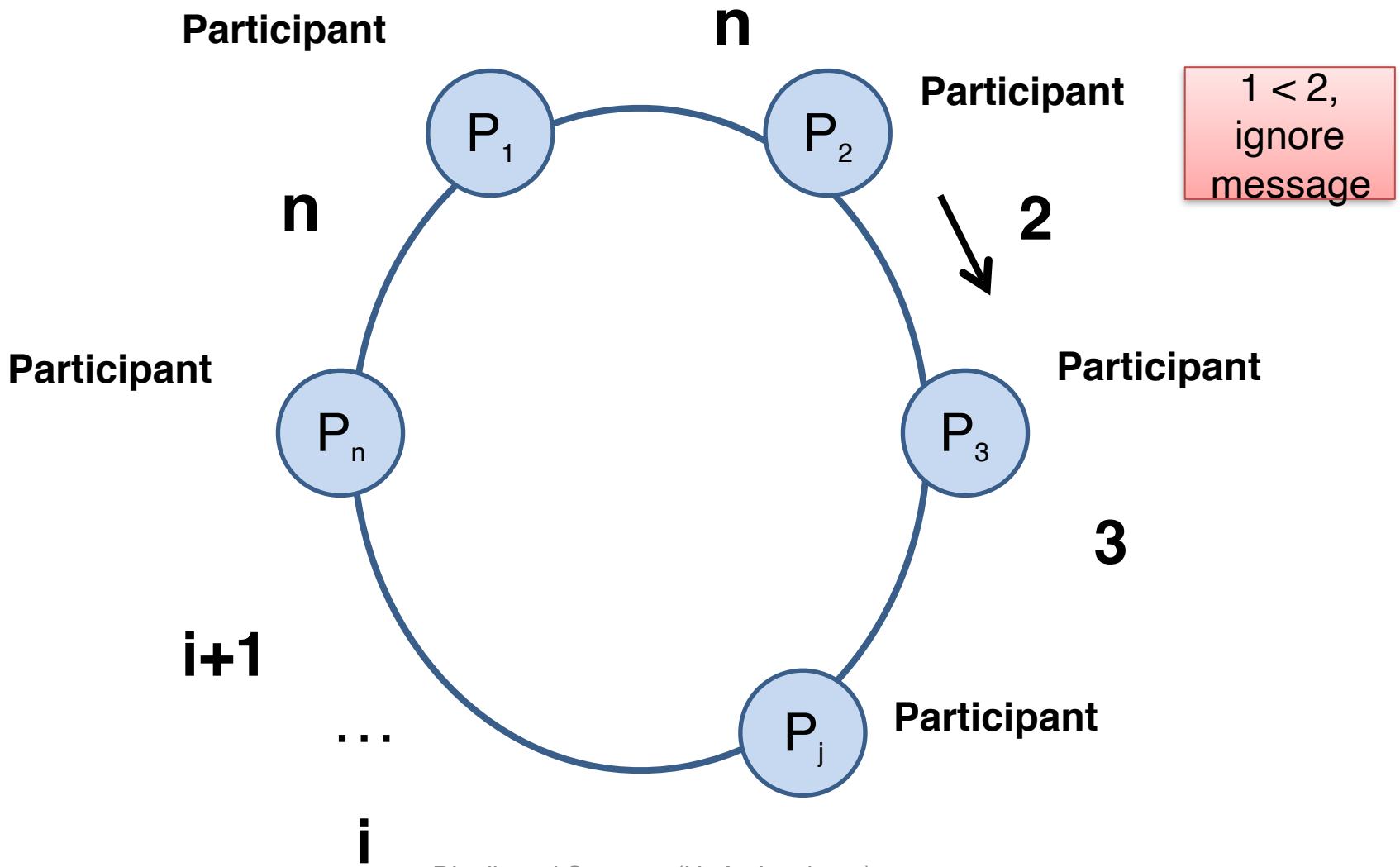
Ring-based algorithm: Concurrent election start



Ring-based algorithm: Concurrent election start



Ring-based algorithm: Concurrent election start



Ring-based election algorithm analysis

- Worst case: $3N - 1$ messages
 - $N-1$ messages to reach highest ID from lowest ID
 - N messages to reach point of origin
 - Leader announcement takes another N messages
- Safety: even with multiple processes starting election
- Liveness: guaranteed progress *if no failures during the election occur*
- Fault-tolerance: Can't tolerate token loss

Bully algorithm, Garcia-Molina, 1982

- Assumes each process has a unique ID, reliable message delivery, and synchronous system
- Assumes processes know each others' IDs and can communicate with one another
 - Higher IDs have priority
 - Can “bully” lower numbered processes
- Initiated by any process that **suspects failure** of the leader
- Tolerates processes crashing during elections

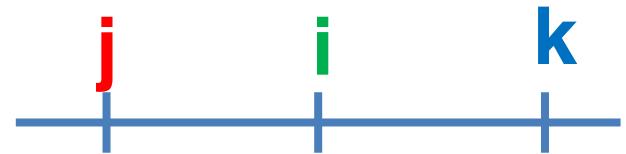


Bully algorithm messages

- Operates with three types of **messages**
 - ***Election*** announces an election
 - ***Answer*** responds to an election message
 - ***Coordination*** announces identity of leader
- Algorithm is triggered by any process that detects leader to have crashed (synchronous system)

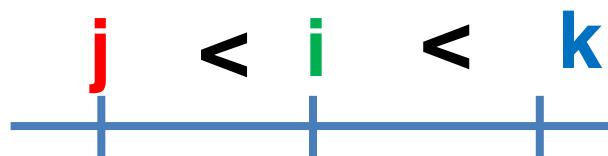
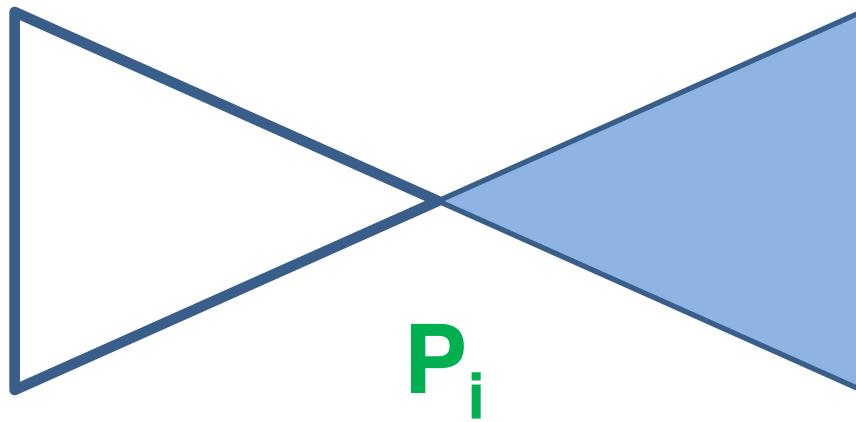
@ P_i P_i detects failure of leader

For any $j < i$ and any $i < k$

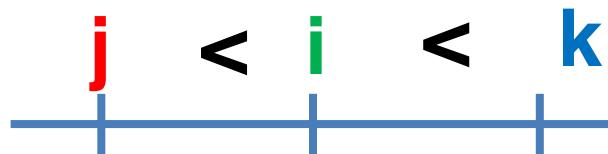
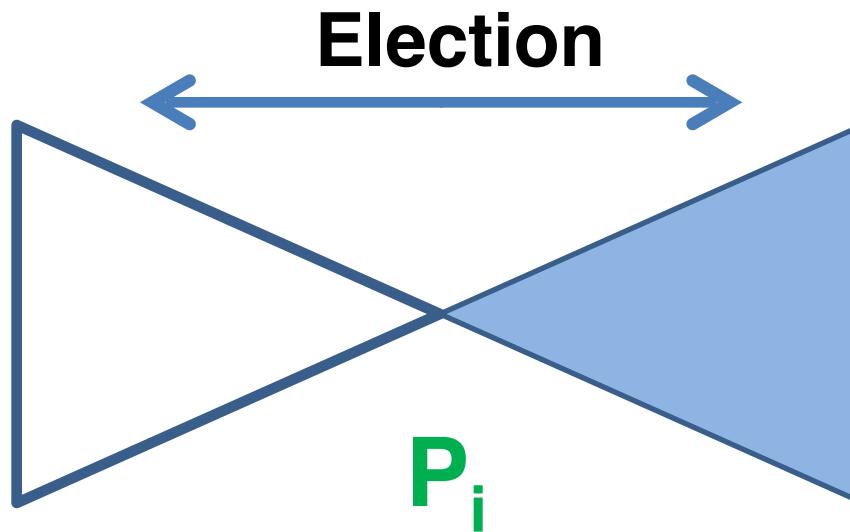


1. Broadcasts **election message**
2. Any P_k receiving election message **responds with answer message** and starts another election
3. Any P_j receiving election message **does not respond**
4. If P_i does not receive any answer message (timeout) then it **broadcasts victory via coordination message**
5. If P_i does receive answer message(s) then waits to receive **coordination message**
6. Restarts election, if no coordination message received (failure happened)

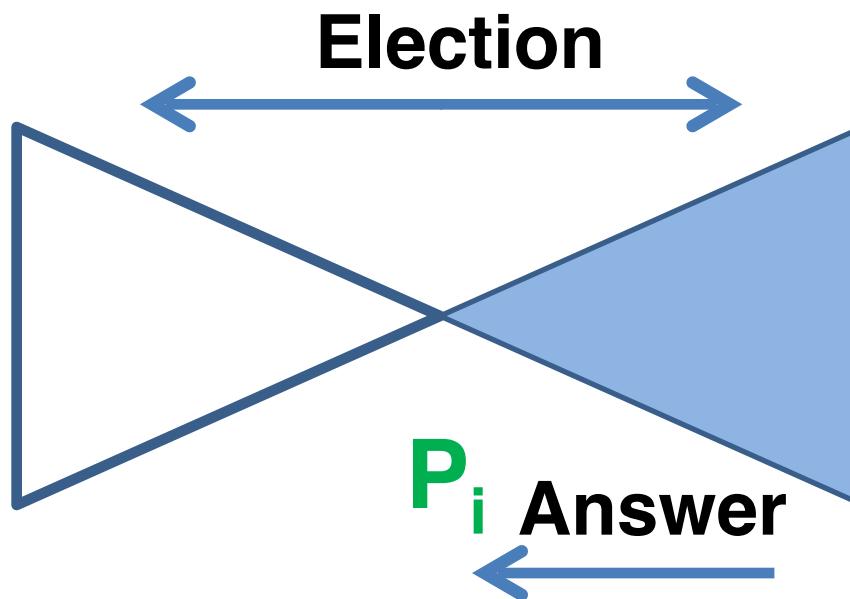
Election & answer



Election & answer



Election & answer



$$j < i < k$$

A horizontal number line with three tick marks. Above the first tick mark is the letter j in red. Between the first and second tick marks is the symbol $<$. Above the second tick mark is the letter i in green. Between the second and third tick marks is the symbol $<$. Above the third tick mark is the letter k in blue.

Upon crash of a process

- Suppose process eventually recovers (*no problem if it stays down, why?*)
- Process may determine that it has the highest identifier, thus, pronounce itself as leader
 - Even though system may have an existing leader (elected during crash)
- New process “**bullies**” current leader



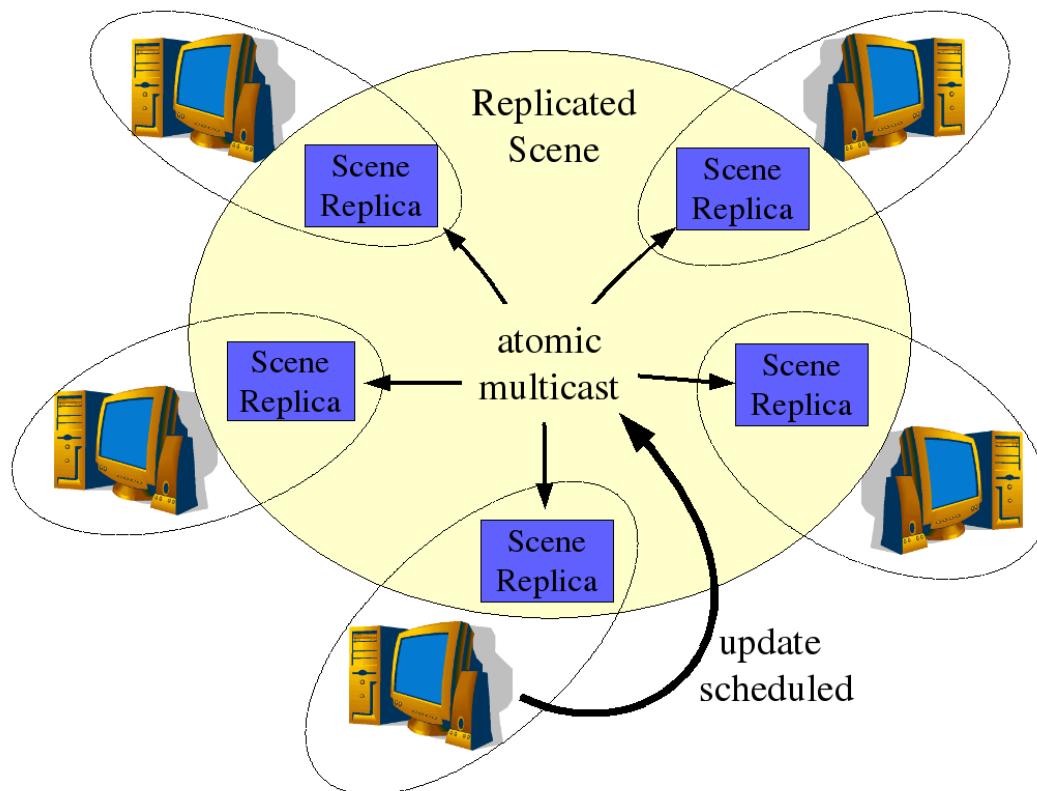
Logical clock summary

- Clocks that are not based on real-time
- Logical time progresses using events
 - Local execution of some code
 - Send/receive messages
- Happened-before relationship
 - Tracks causality between events across processes
- Lamport clock
 - Single counter updated at every event
 - Introduces false positive causality
- Vector clock
 - Size of timestamp is relative to number of processes
 - Can determine causality more accurately than Lamport clock

Replication

Pezhman Nasirifard, Hans-Arno Jacobsen

Application & Middleware Systems Research Group

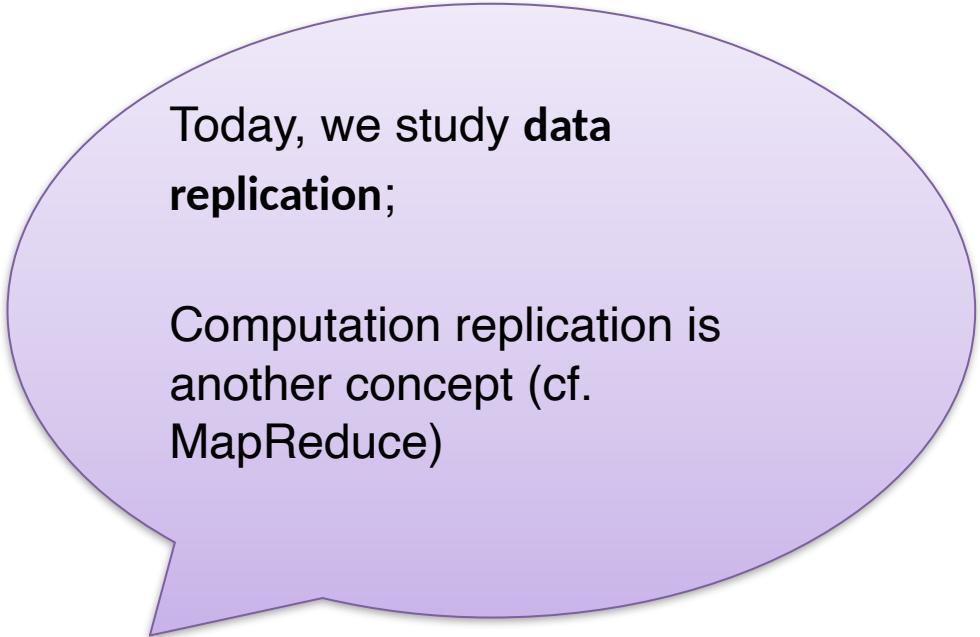


Outline

- Data replication techniques
 - Active replication
 - Passive replication
 - Multi-primary replication
 - Gossiping
 - Chain Replication

Why Replication?

- Performance
- High availability
- Fault tolerance

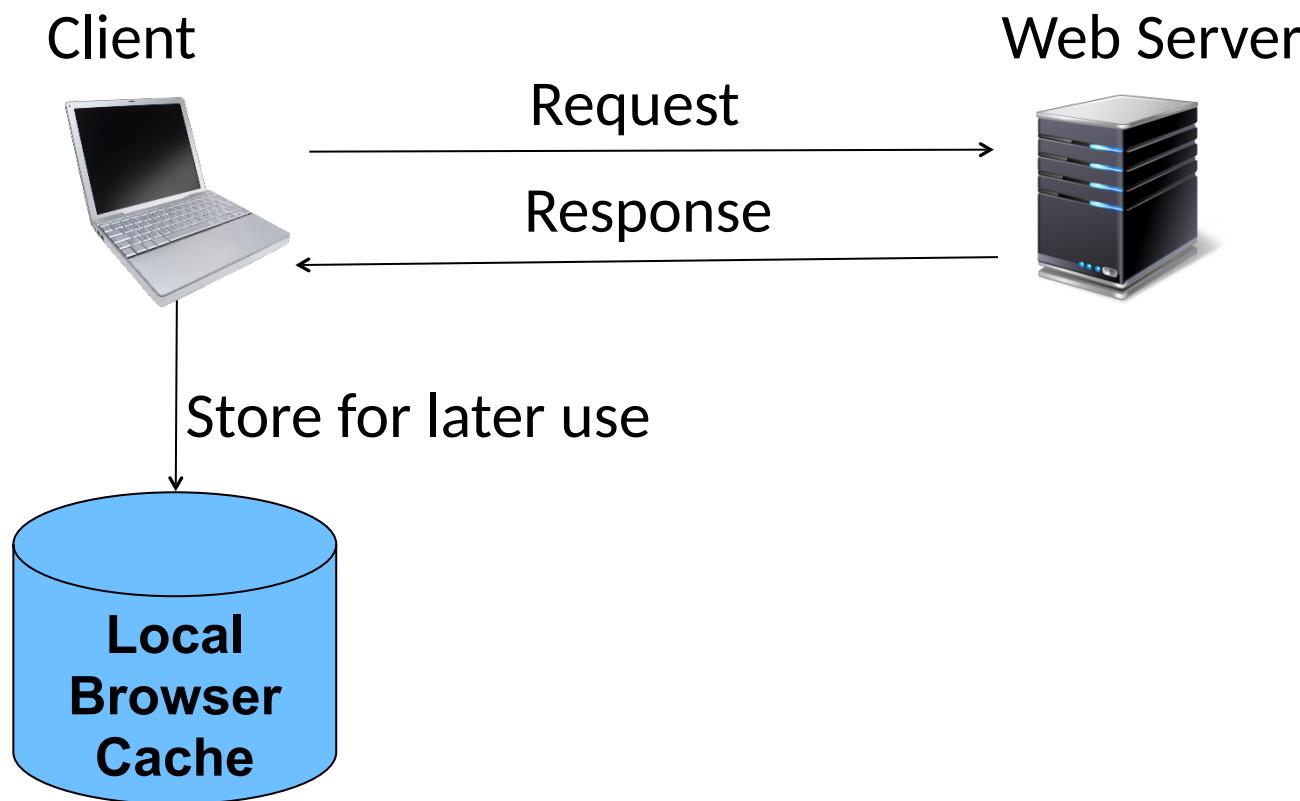


Today, we study data replication;

Computation replication is another concept (cf. MapReduce)

Performance

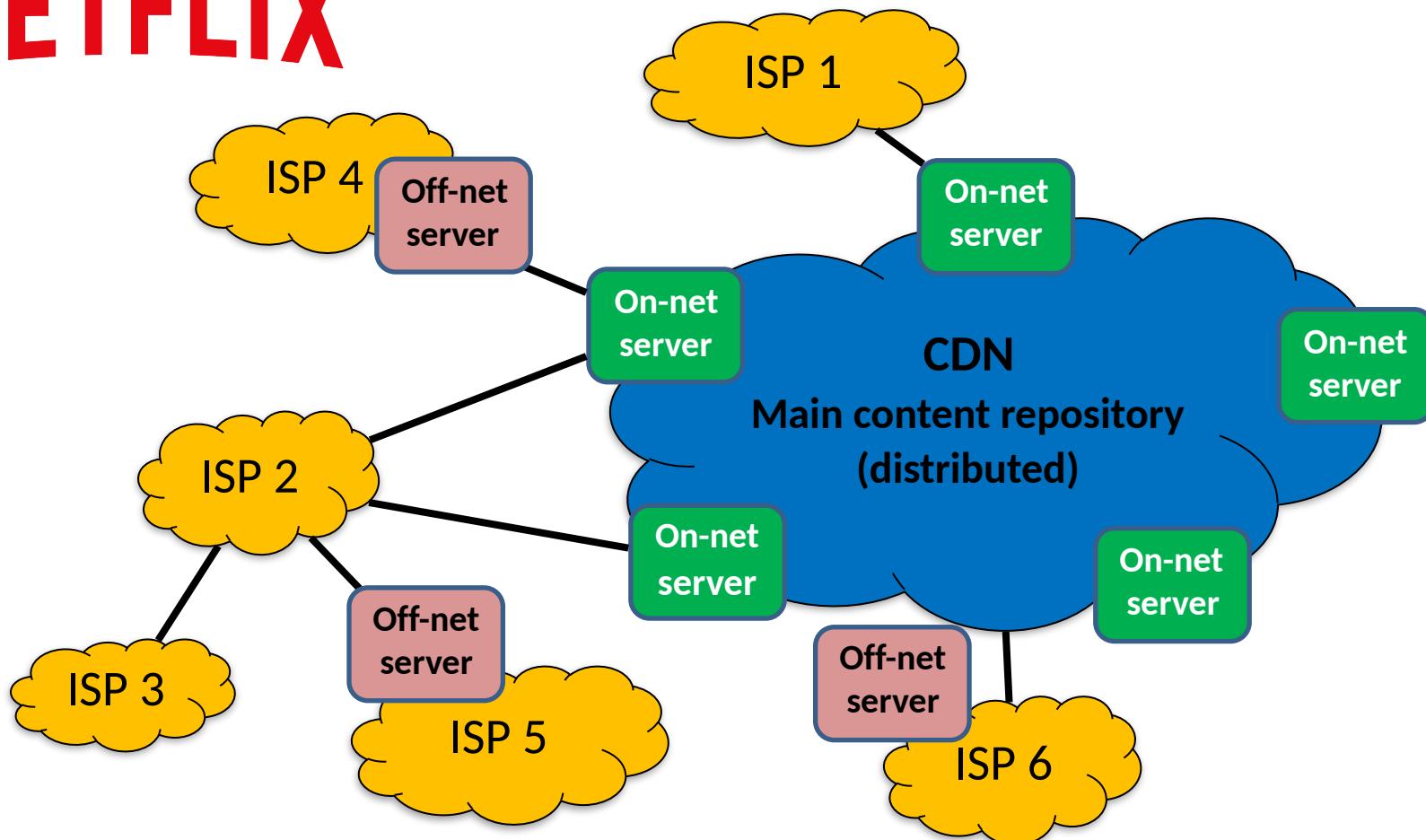
Caching data at browsers and proxy servers



 YouTube
NETFLIX

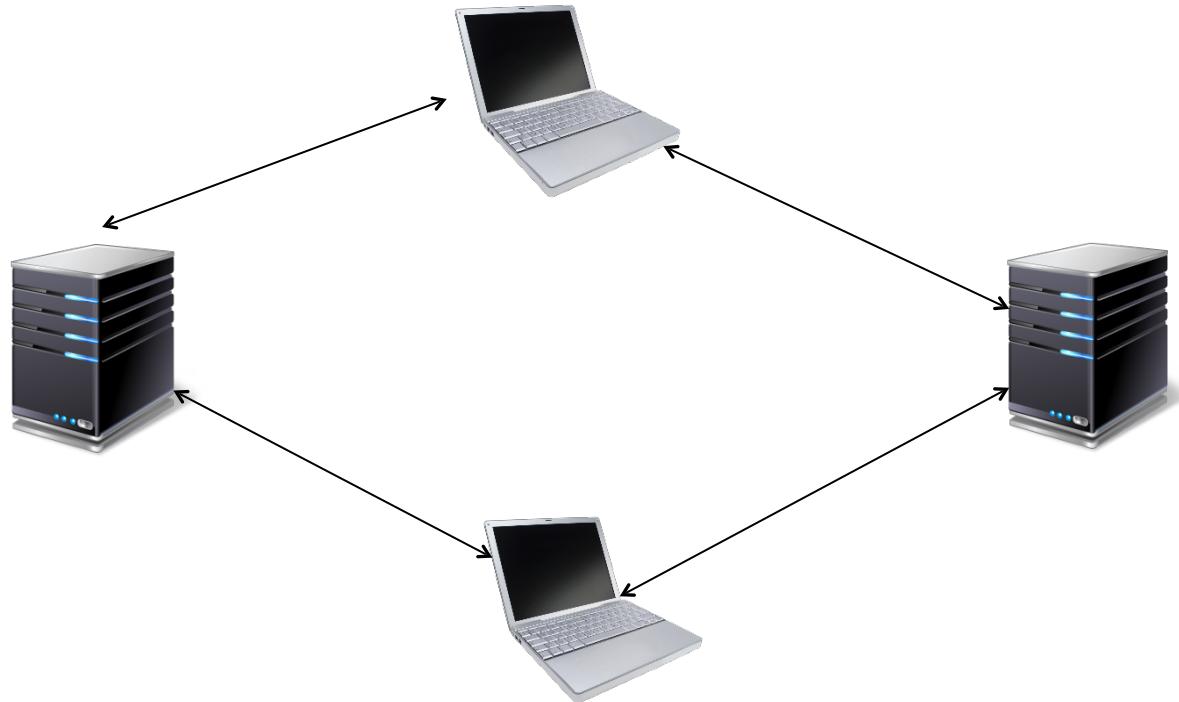
Typical (video) CDN

(Content Delivery Network)



High Availability

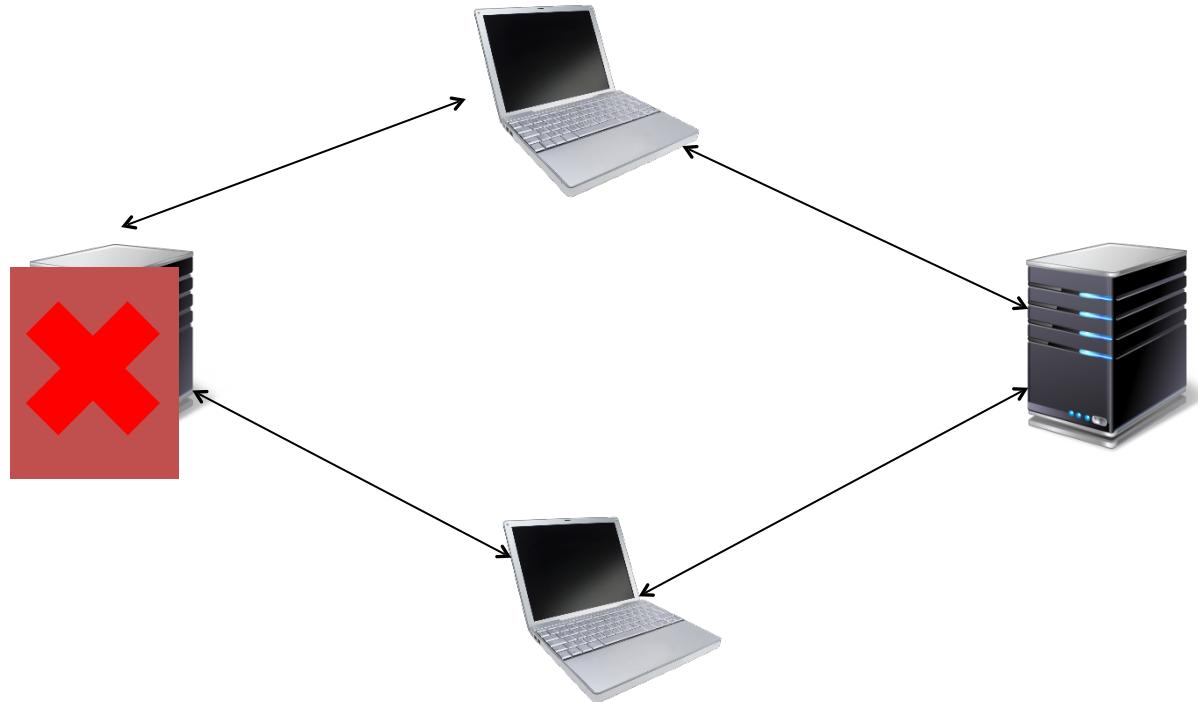
Upon crashes, service offered by replica



[1] The availability of a service by replicating its servers would grow.

High Availability

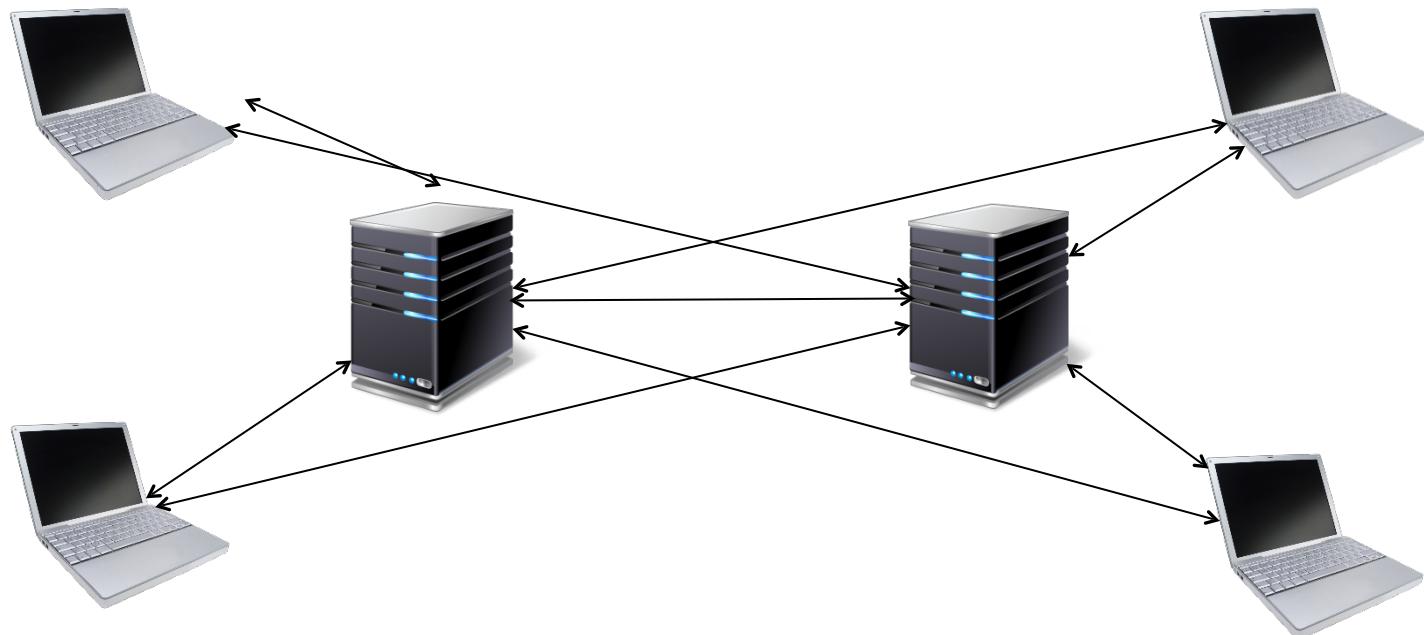
Upon crashes, service offered by replica



[1] The availability of a service by replicating its servers would grow.

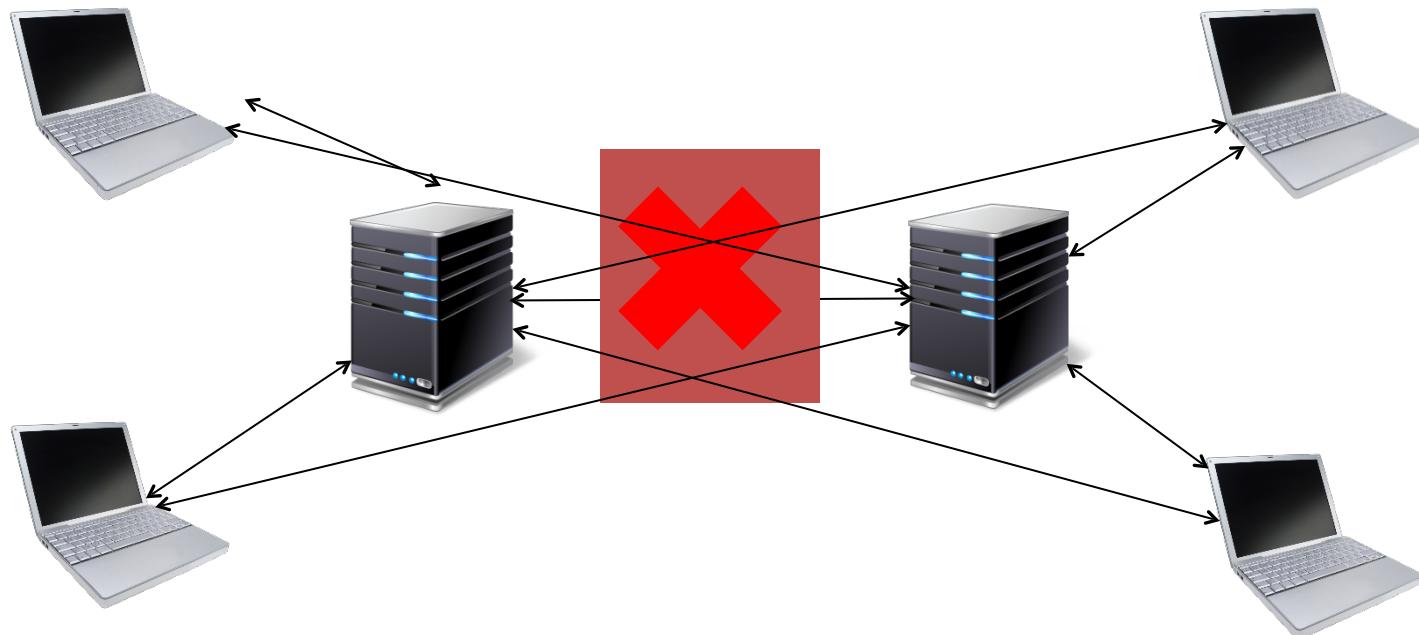
High Availability: Network Partitioning

Upon network partition, service available to clients in partition



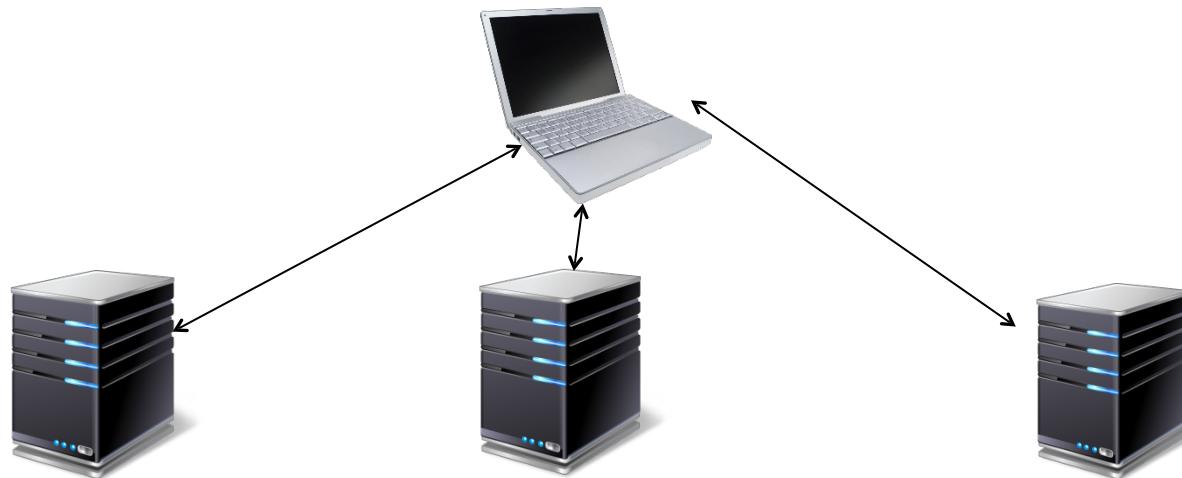
High Availability: Network Partitioning

Upon network partition, service available to clients in partition



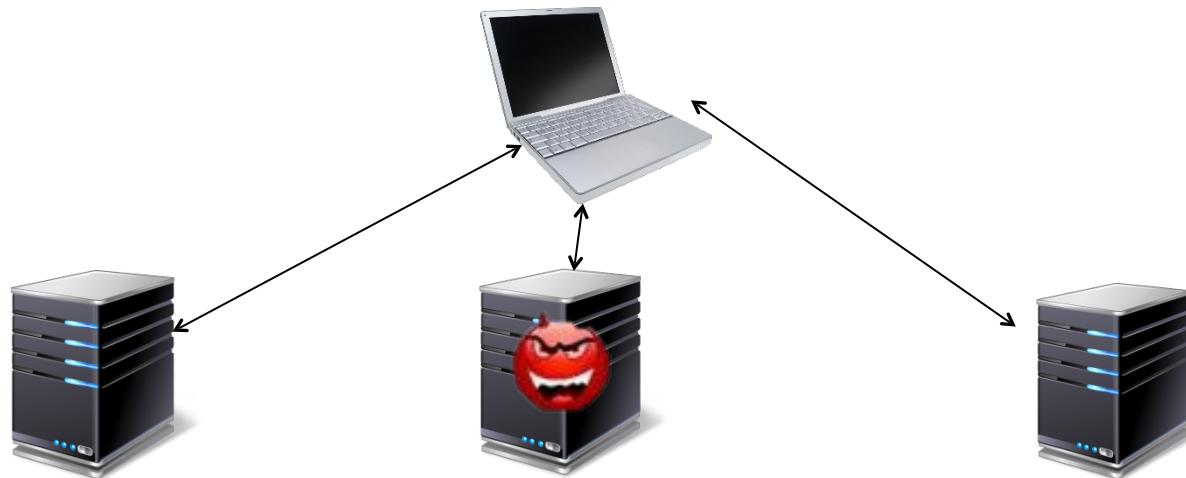
High Availability: Fault Tolerance

Providing reliable service in face of faulty servers
(not just crashes, but also arbitrary failures!)



High Availability: Fault Tolerance

Providing reliable service in face of faulty servers
(not just crashes, but also arbitrary failures!)



The “cost” of replication

- Not just cost of storing additional copies of the data
- Cost to **keep replicas up to date** in face of updates?
- E.g., additional bandwidth, number of messages exchanged, higher latency (i.e., service-response time), complexity of code, etc.
- How can the **problem of stale** (out-of-date) **data at replicas** be solved?

State Machine Replication (RSM)

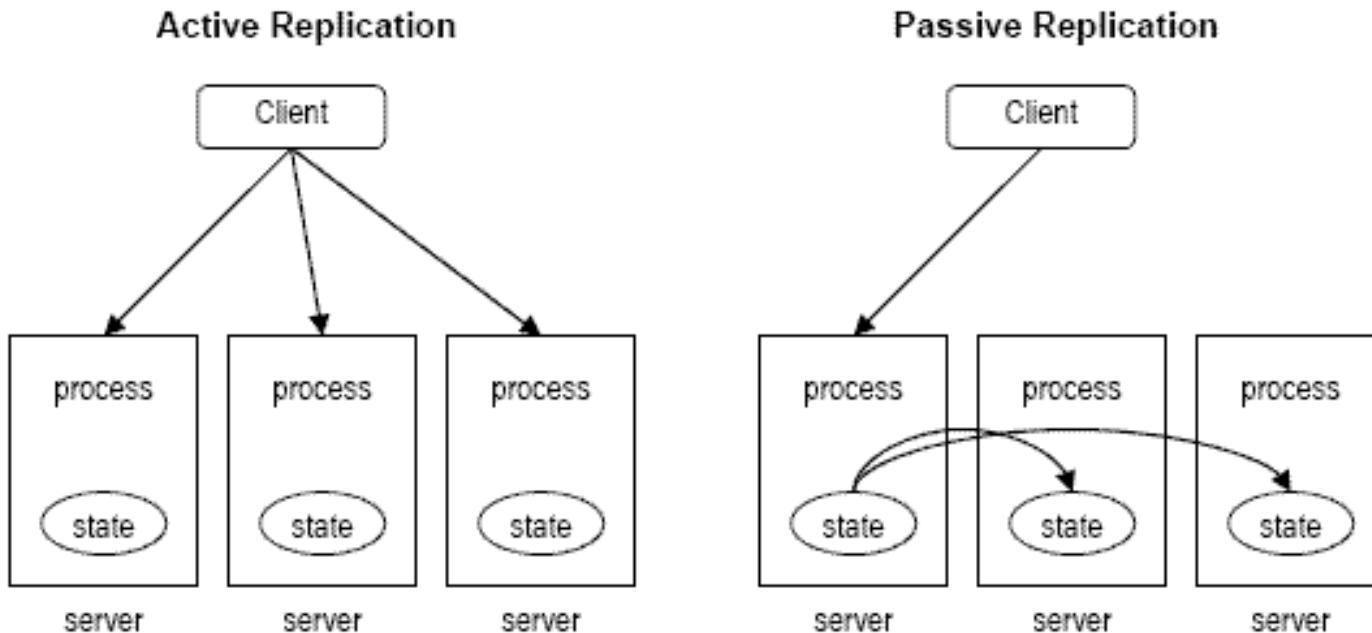
- Also known as Replicated State Machine (RSM)
- Every replica starts at the same initial state.
- They execute all client commands **in the same order**.
- Commands are assumed to be deterministic, therefore all replicas end up in the **same order**.
- A client can query the state of any replica, since they are always the same.
- RSM is widely used: Chubby (Paxos), Bitcoin (Blockchain)

System Description

- Processes are either **clients** or **replicas (servers)**
- Clients send requests to replicas, and replicas send responses back to the clients
 - Requests are updates or queries
 - Clients are potentially able to send to any replica
- Processes can crash

RSM Replication Techniques

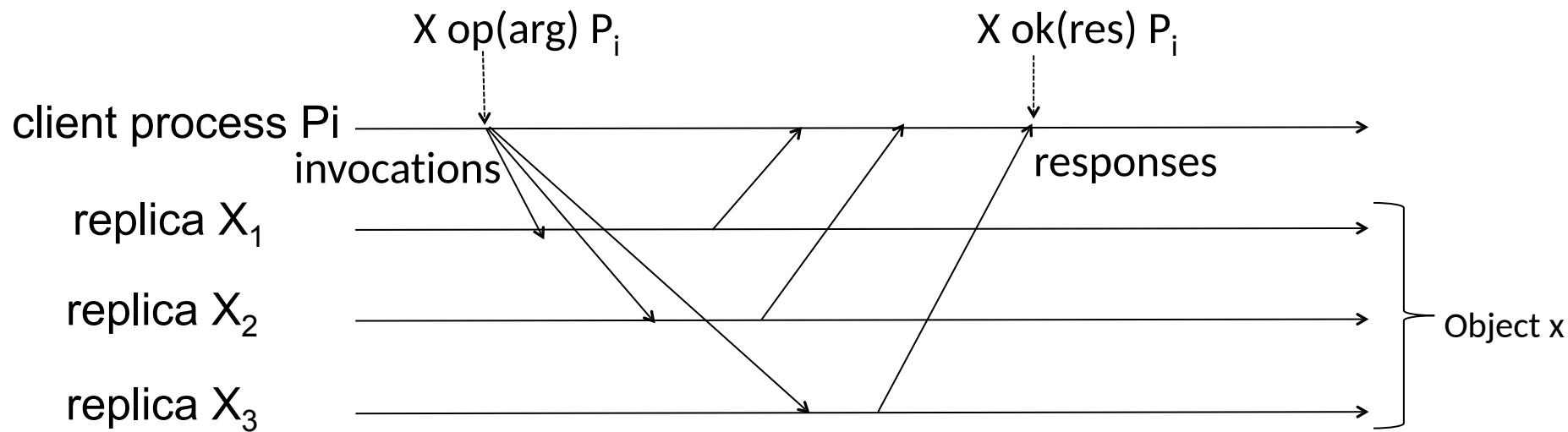
- Active Replication
- Passive Replication
 - Primary-backup
 - Multi-primary



Active Replication

- The client must send their request to every replica
- Requires total order broadcast messaging to guarantee each replica will receive ALL requests (**from ALL clients**) in the same order
- Since this is a RSM, every replica will send the same response back
- Easier to implement since it is like client-server
- Also faster to get a response, since the first result received can be used

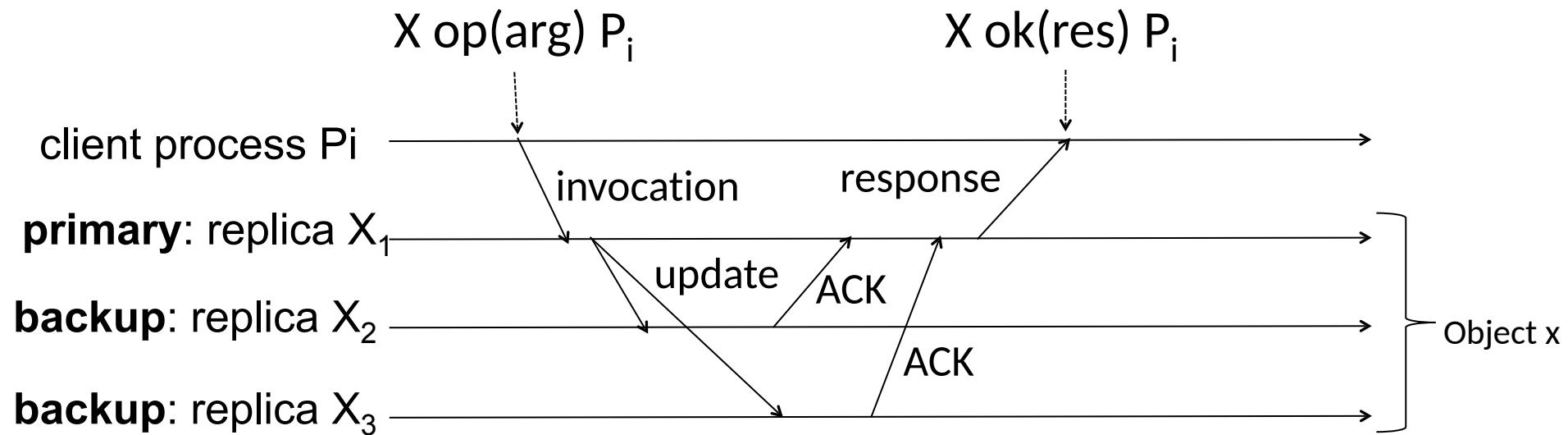
Active Replication



Passive Replication: Primary-Backup

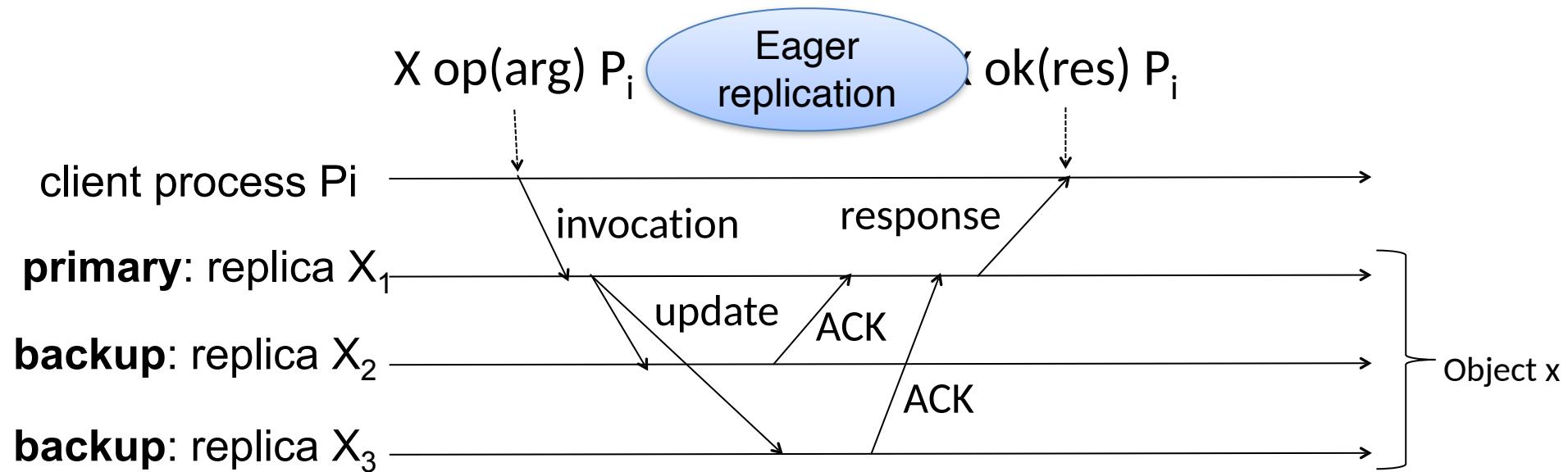
- A Replica is chosen to be the Primary (use Leader Election)
- **Primary**
 - Receives invocations from clients
 - Execute requests and sends back answers
 - Replicates the state to other replicas
- **Backup**
 - Interacts with the primary only
 - Is used to replace the primary when it crashes
- Called **Eager** replication if the replication is performed within the transaction boundaries (e.g. before the reply is sent)

Primary Backup Scenario



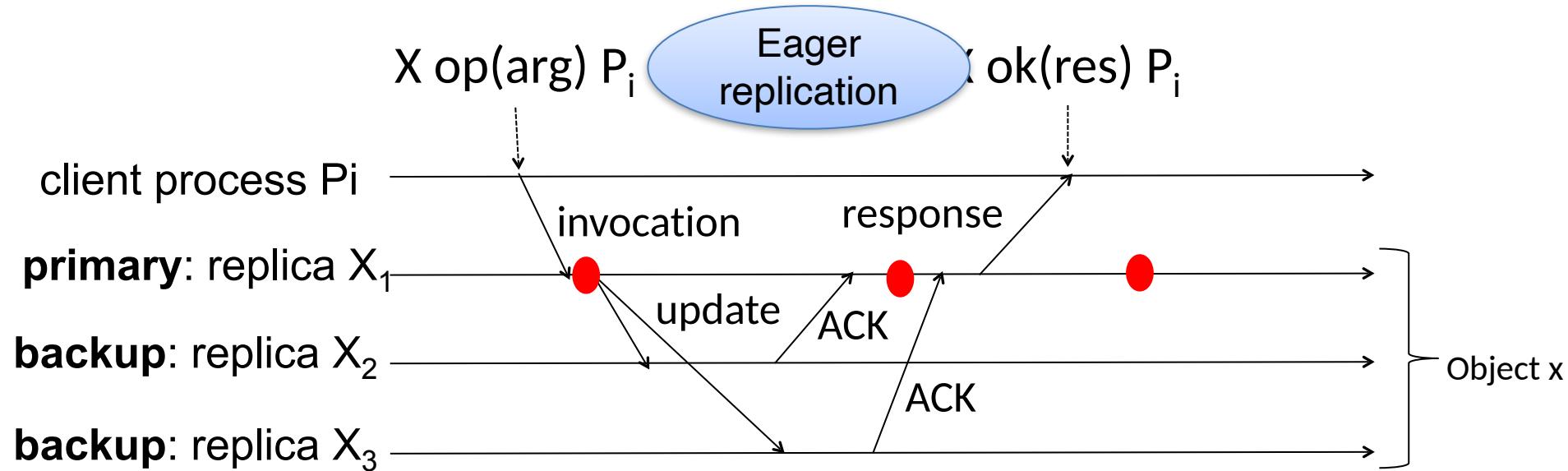
Guarantee sequential consistency: Order in which the primary receives clients' invocations define the order of the operation on the data item(s). (cf. lecture on consistency)

Primary Backup Scenario



Guarantee sequential consistency: Order in which the primary receives clients' invocations define the order of the operation on the data item(s). (cf. lecture on consistency)

Primary Backup Scenario

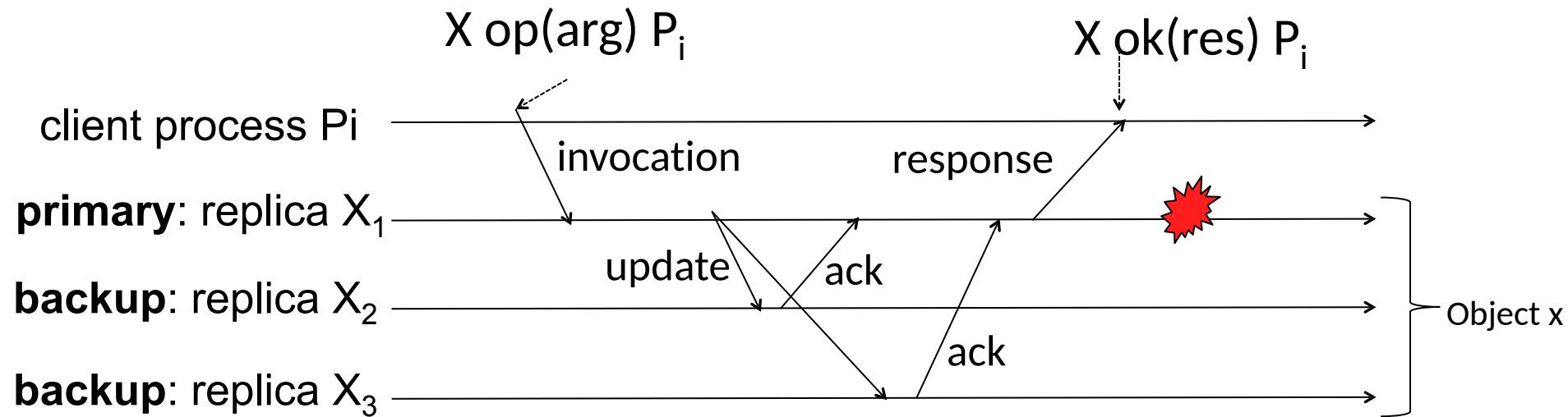


Guarantee sequential consistency: Order in which the primary receives clients' invocations define the order of the operation on the data item(s). (cf. lecture on consistency)

Primary Backup: Presence of Crash

- Three scenarios
 - Primary fails after the client receives the answer
 - Primary fails before sending update messages
 - Primary fails after sending update messages and before receiving all the ACK messages
- In all cases, a new **primary is elected** from among the backups

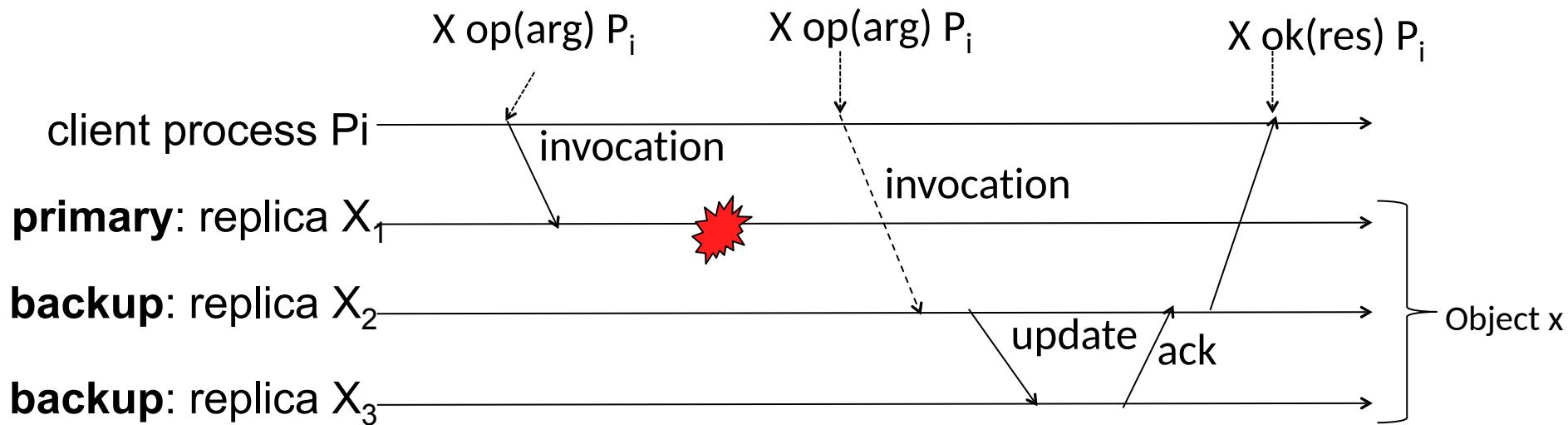
Scenario 1



Scenario 1

- Primary fails **after** the client receives the answer
- Nothing bad happens since the client has already received the response
- A new primary is elected

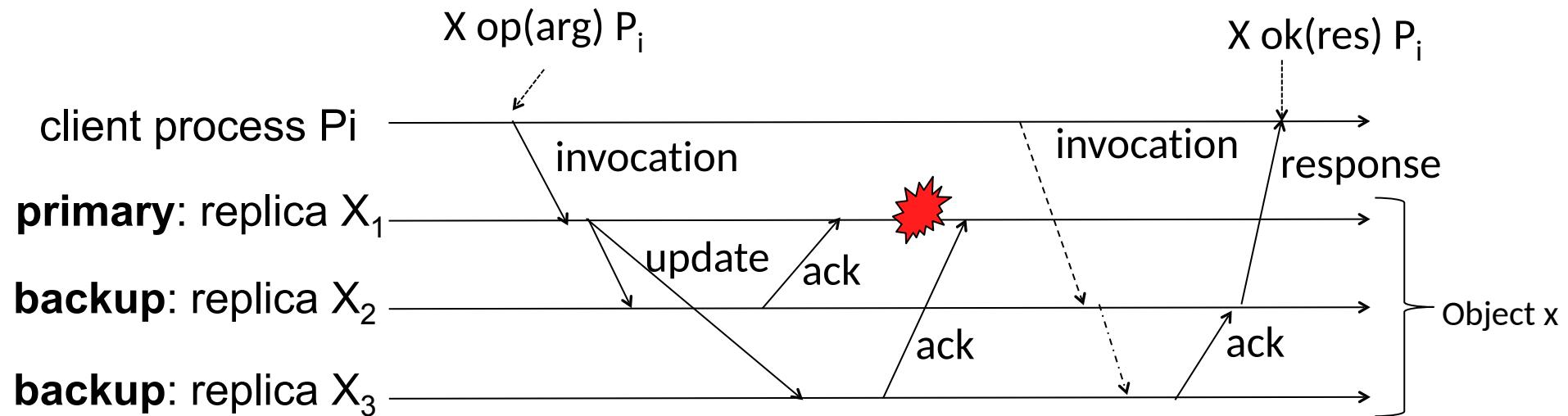
Scenario 2



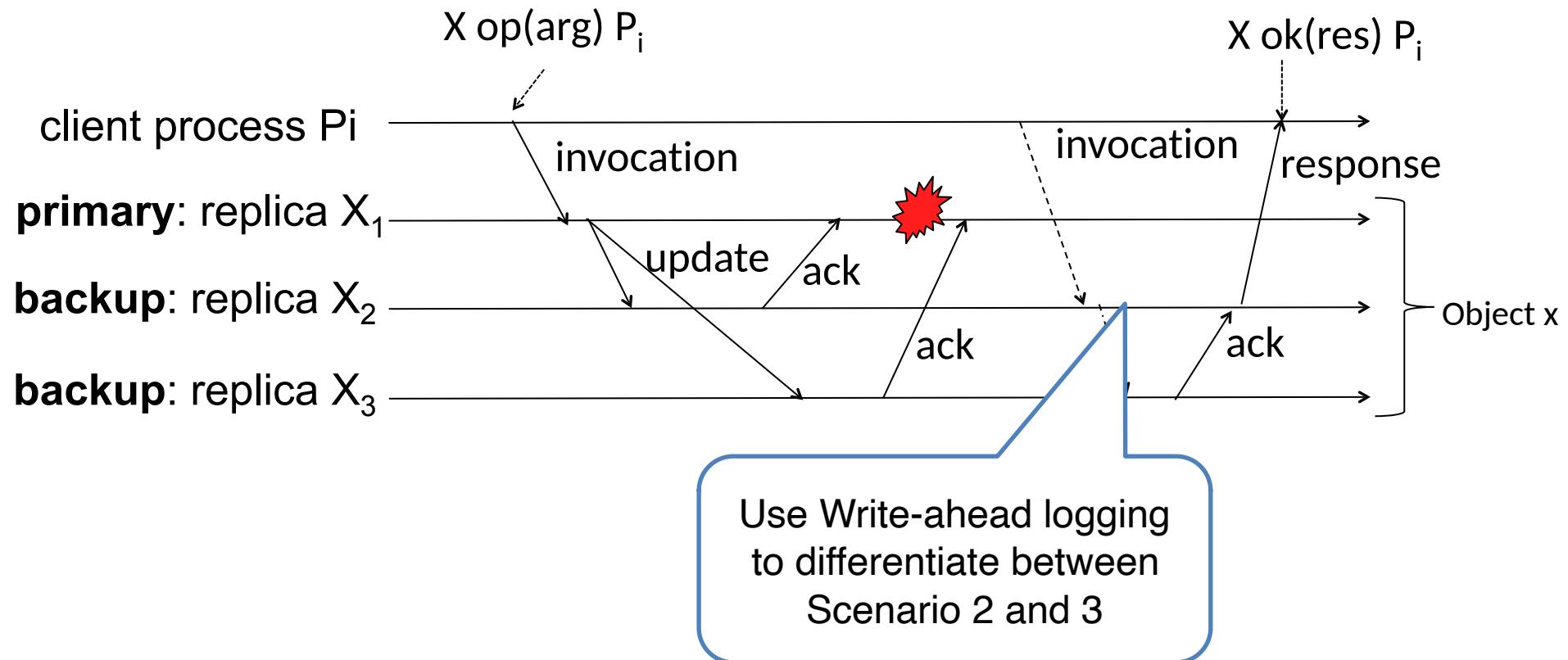
Scenario 2

- Primary fails **before sending update messages**
- Client does not get an answer and **resends the requests after a timeout**
- The newly elected primary will handle the request as new

Scenario 3



Scenario 3



Scenario 3

- Primary fails **after sending some update messages and before receiving all ACK messages**
- When a primary fails, a new primary is elected by the backup replicas
- Client does not get an answer and **resends the requests after a timeout**
- Each replica holds a **write-ahead log** (also called **commit log**) which tracks all processed operations.
- When the new primary and the remaining replica receives the operation, they do not execute it again if it is in their log and directly sends a response back.

Multi-primary Replication (MPR)

- Primary-backup is **not scalable**, since only a single process handles client requests
 - Inefficient use of replica resources
- Multi-primary solves that by allowing every replica to handle client requests
 - The replicas then have to figure out how to order the requests (e.g., using consensus)
 - Conflict-free replicated data type – CRDTs
- If the replication is done **eager**, the processes have to agree on the order of operations before they execute any command and respond to the clients
 - This can be slow since it locks processes

Optimistic Lazy MPR

- To improve response times, replication is often lazily
 - The replica first executes locally and returns a result to the client right away
 - The replicas asynchronously propagate updates they made
- This type of replication is called **optimistic**
 - Replicas may diverge, which can introduce inconsistencies, aborts, and rollbacks
 - The opposite is called **pessimistic**, where the client receives a response only if the outcome is **clear**

Gossiping protocols

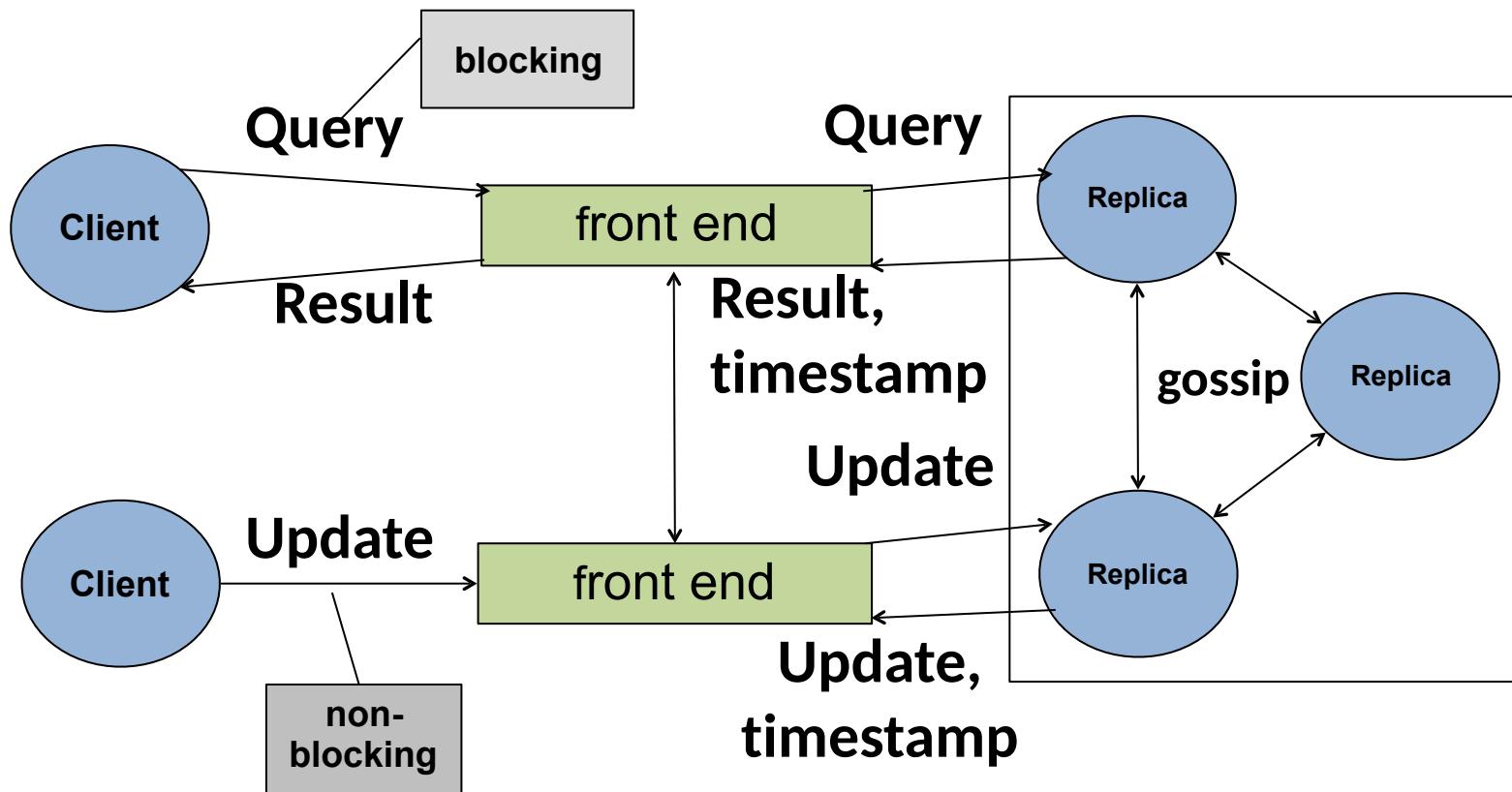


- Gossiping protocols disseminate information in an **incremental** manner
 - Avoids overloading processes with heavy broadcast messages
 - The drawback is that it takes more time to fully propagate some information
- Each peer maintains a **partial view** of other peers
- During each gossip round, each peer chooses a **random** node from its view to exchange information about:
 - Some application data (e.g., current state)
 - Its partial view
- The peers then update their state and partial view based on the information received
- Gossiping happens **periodically** and **non-deterministically**
- Used in Cassandra for propagating the status of each node (alive/dead)

Lazy replication using gossiping

- Replicas want to gossip about the operations it has processed
- During a gossip round, two replicas compare their operation log and apply any operations they have not seen yet
- To preserve consistency, **vector clocks** can be used to track causality between operations
- If the system quiesces (i.e., no more operations are processed), then each replica will **eventually** converge to the same state by gossiping enough times

Gossip Architecture



Message Types in Gossip Architecture

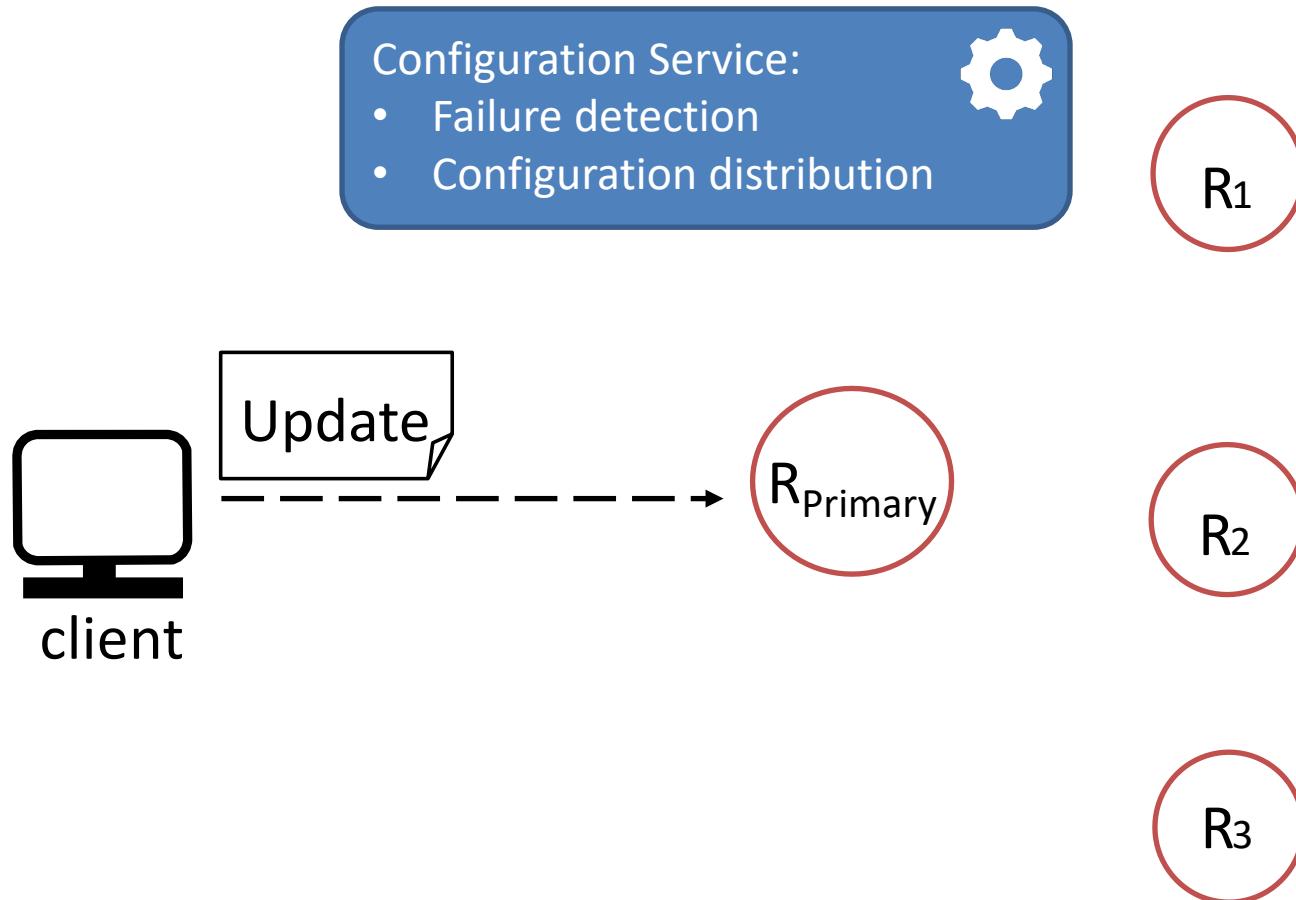
- **Query message:** A message from a client to its closet replica to determine the state of the system
- **Update message:** A message from a client to the closet replica to update the state of the system
- **Gossip message:** This message contains some updates that a replica thinks that some other replica might not have received (send to a number of other replicas)

Operation of Gossip Architecture

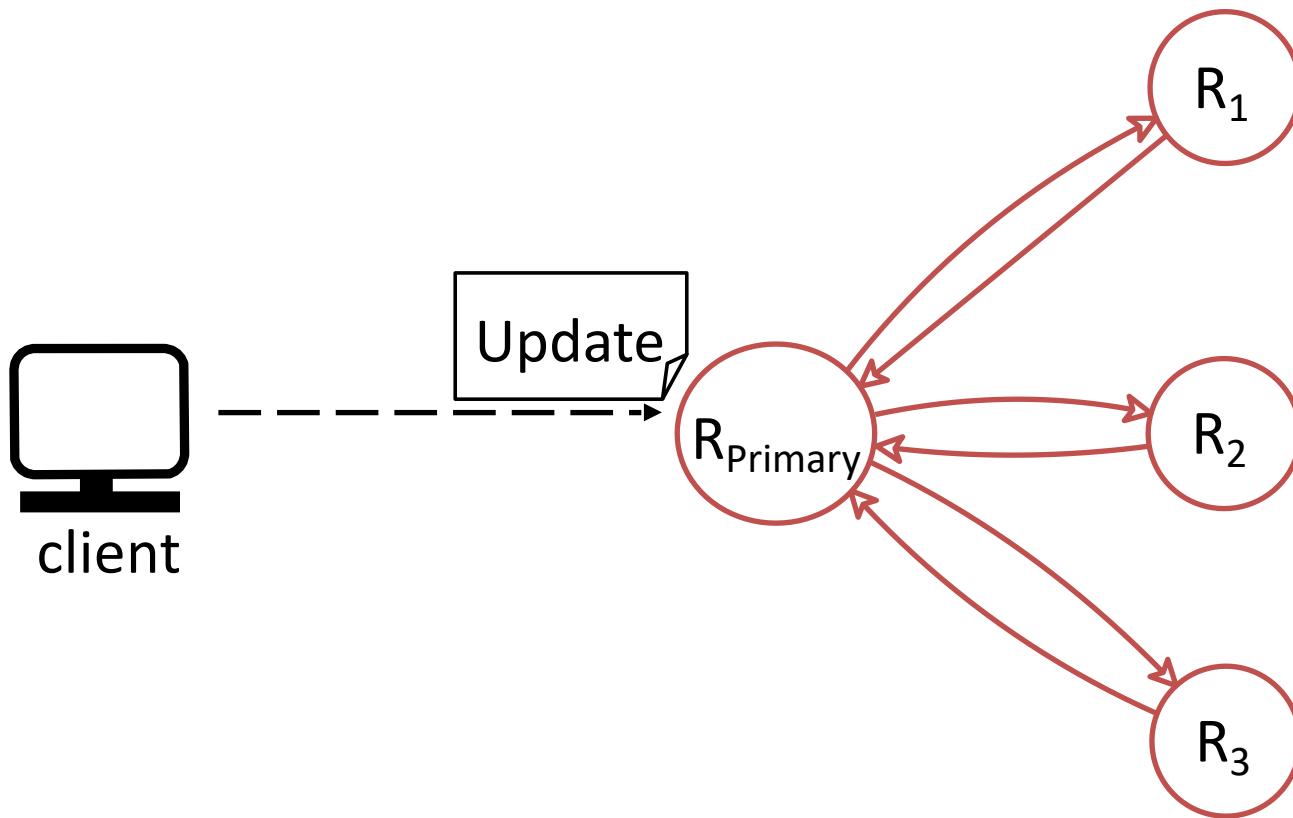
- How it works
 - Front end sends request to a single replica manager at a time, with timestamp
 - If *query*, front-end/client blocks waiting for reply
 - If *update*, return to client immediately; updates propagated in the background
 - Request processed according to ordering constraints
 - Replica managers update each other by sending gossip messages
- Timestamps
 - Uses vector timestamps (arrays of logical clocks, e.g. ,(2,2,5), one per manager)

CHAIN REPLICATION

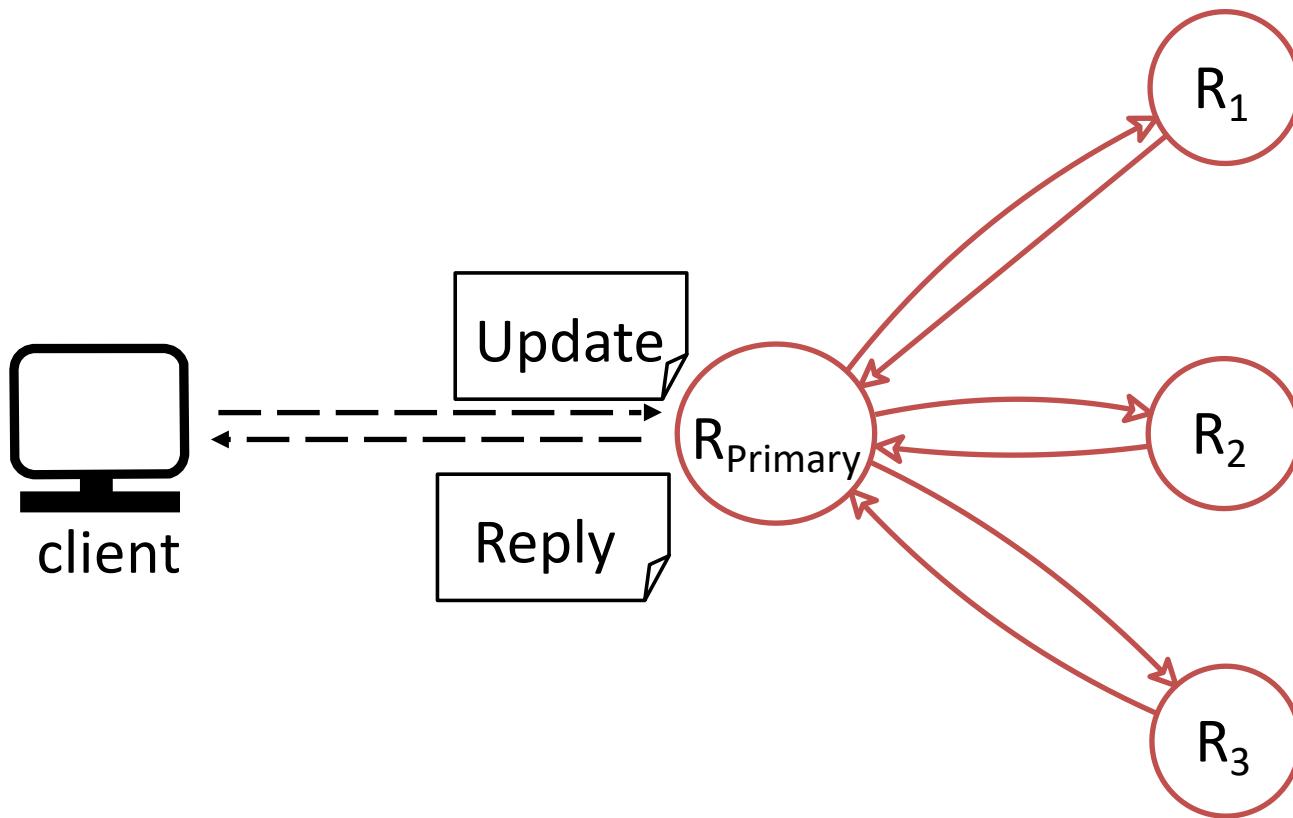
Primary-Backup Replication



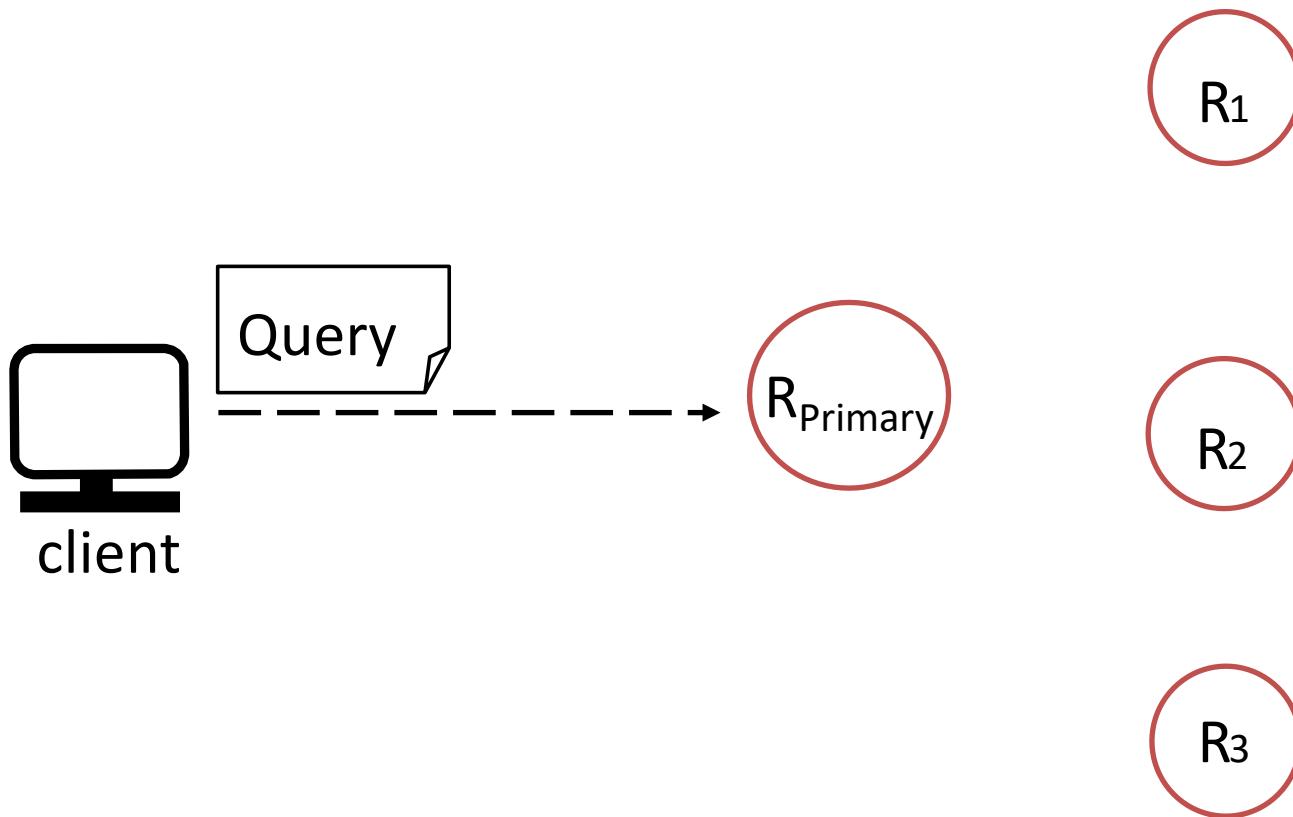
Primary-Backup Replication



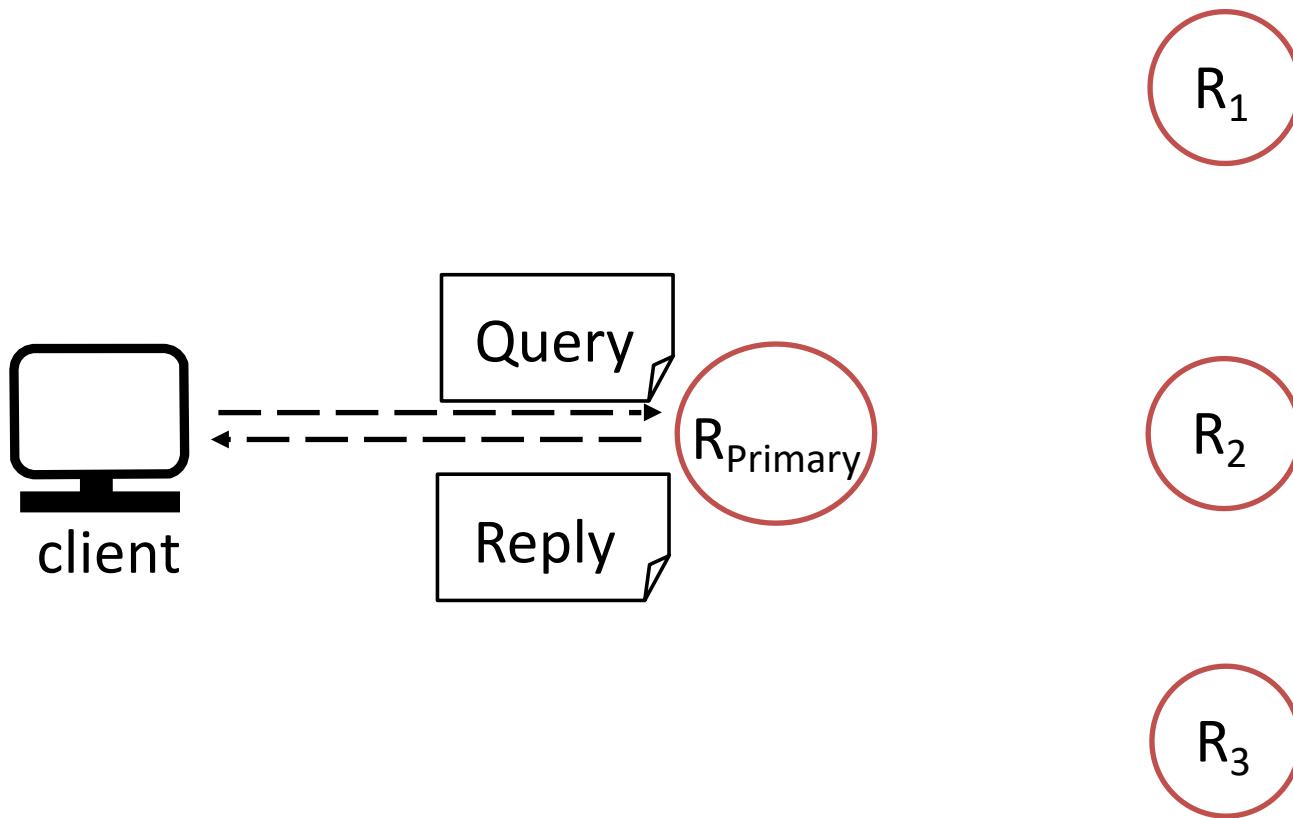
Primary-Backup Replication



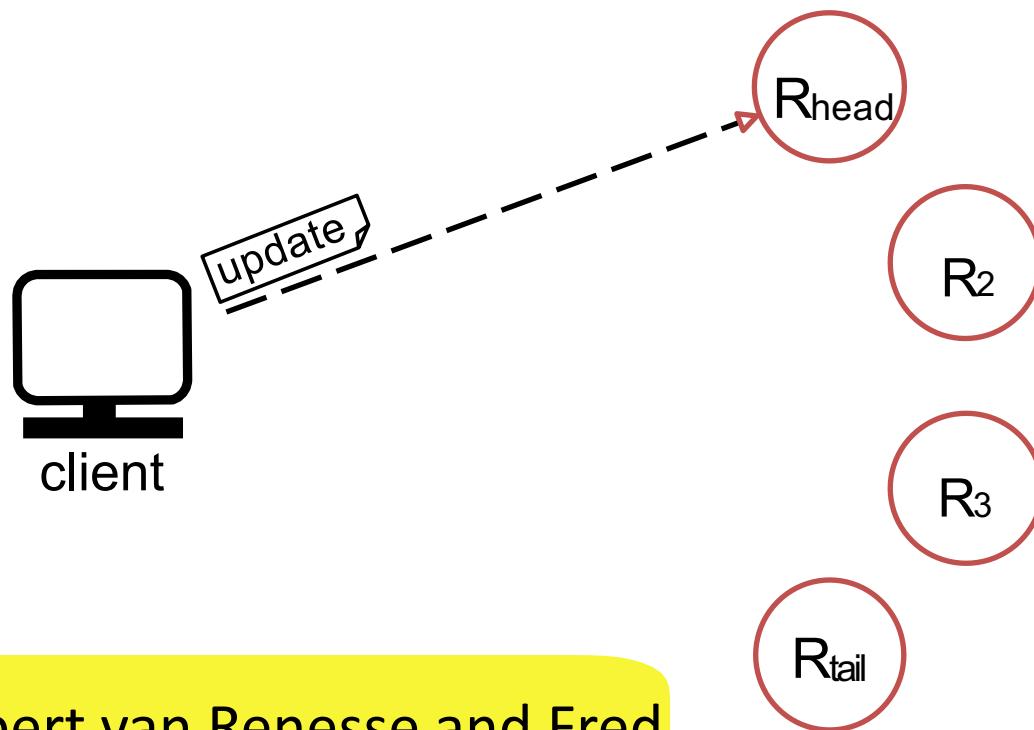
Primary-Backup Replication



Primary-Backup Replication

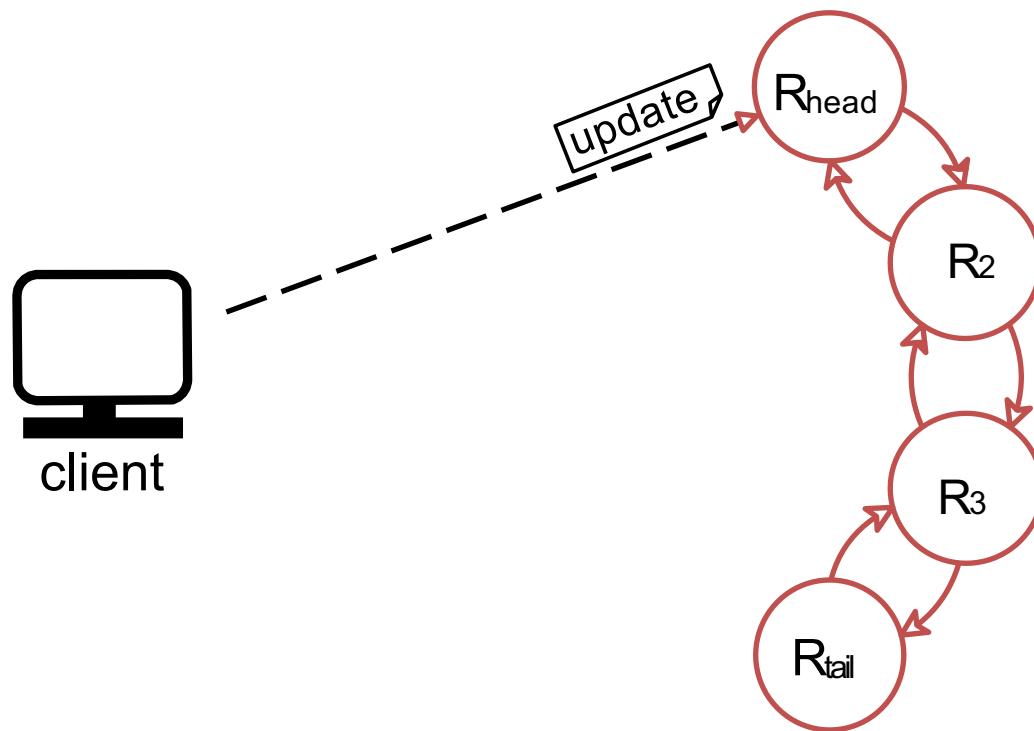


Chain Replication

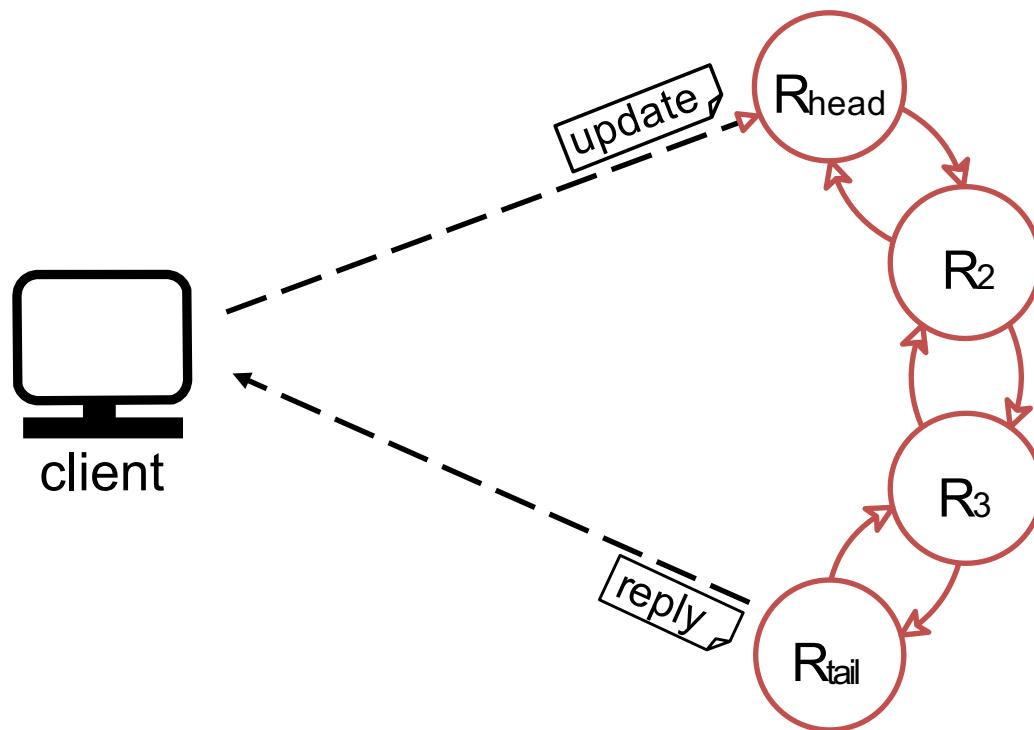


By Robbert van Renesse and Fred
B. Schneider in 2004

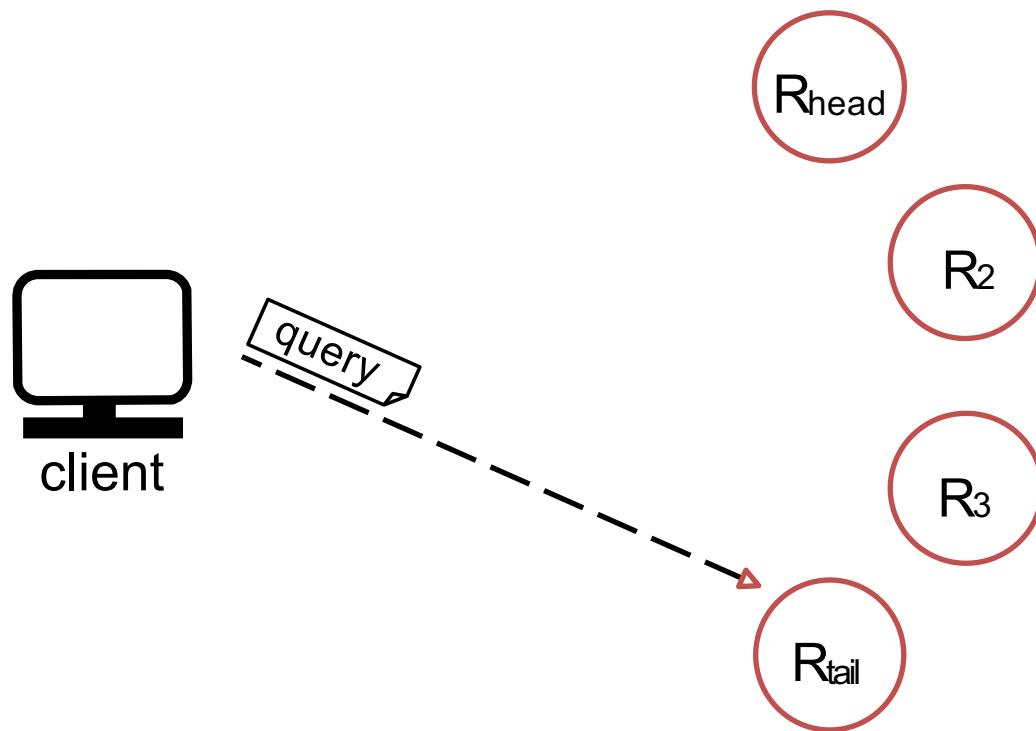
Chain Replication



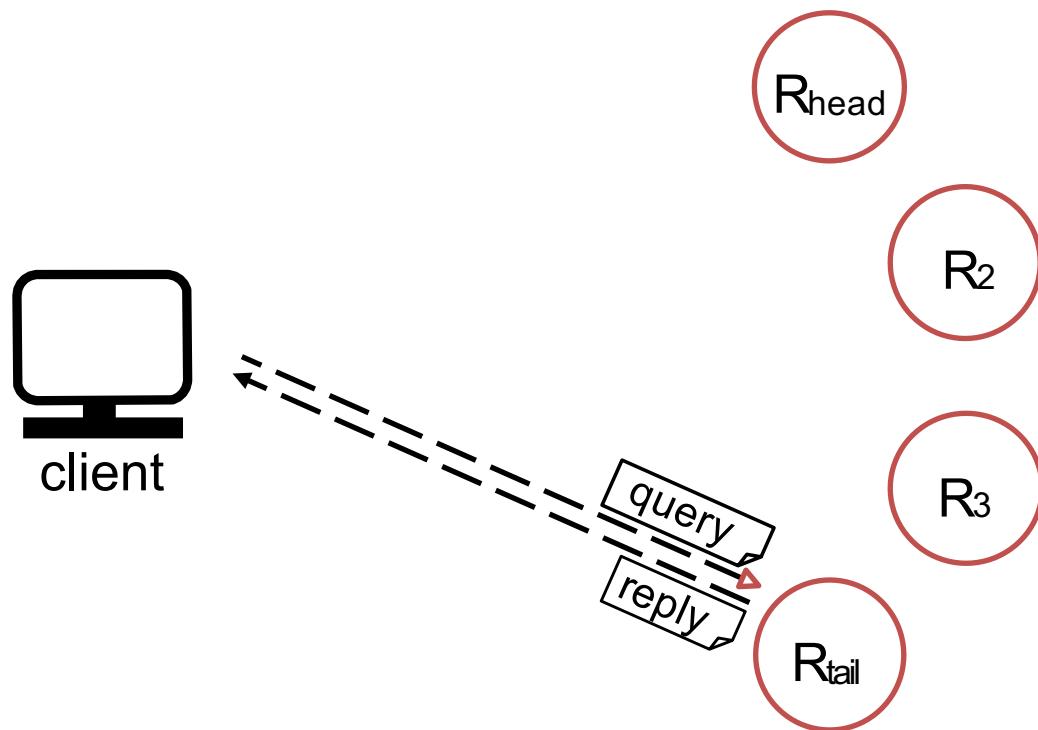
Chain Replication



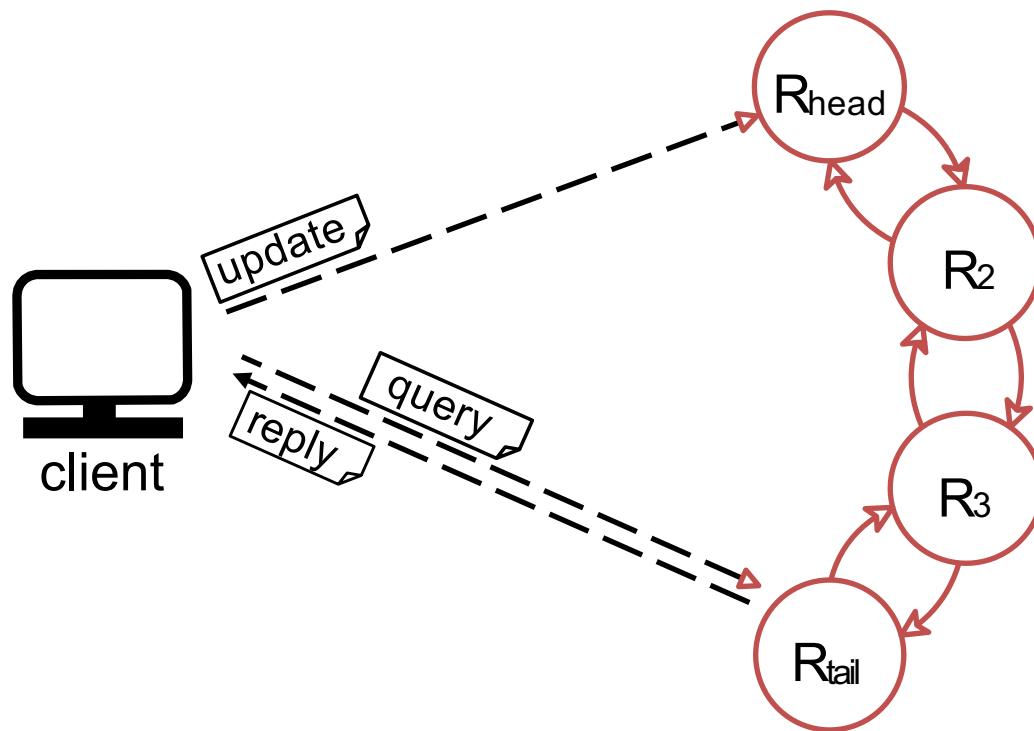
Chain Replication



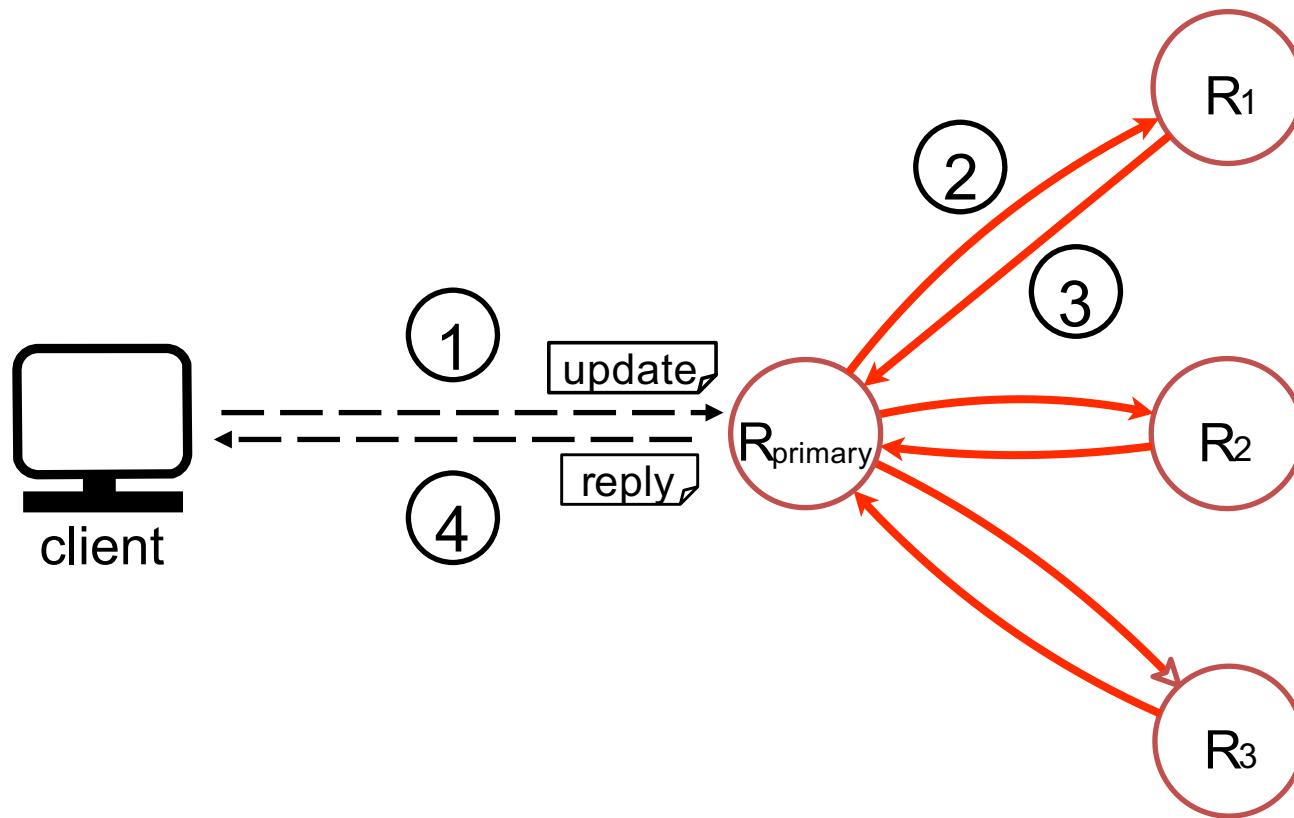
Chain Replication



Chain Replication

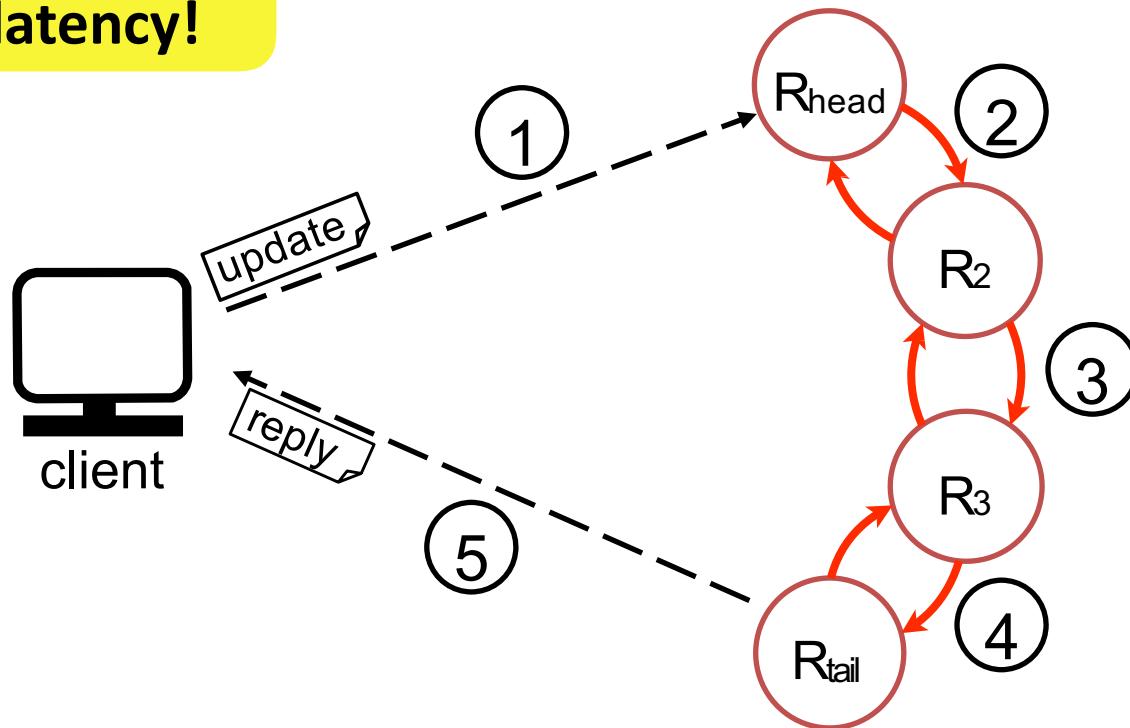


Primary-Backup Replication

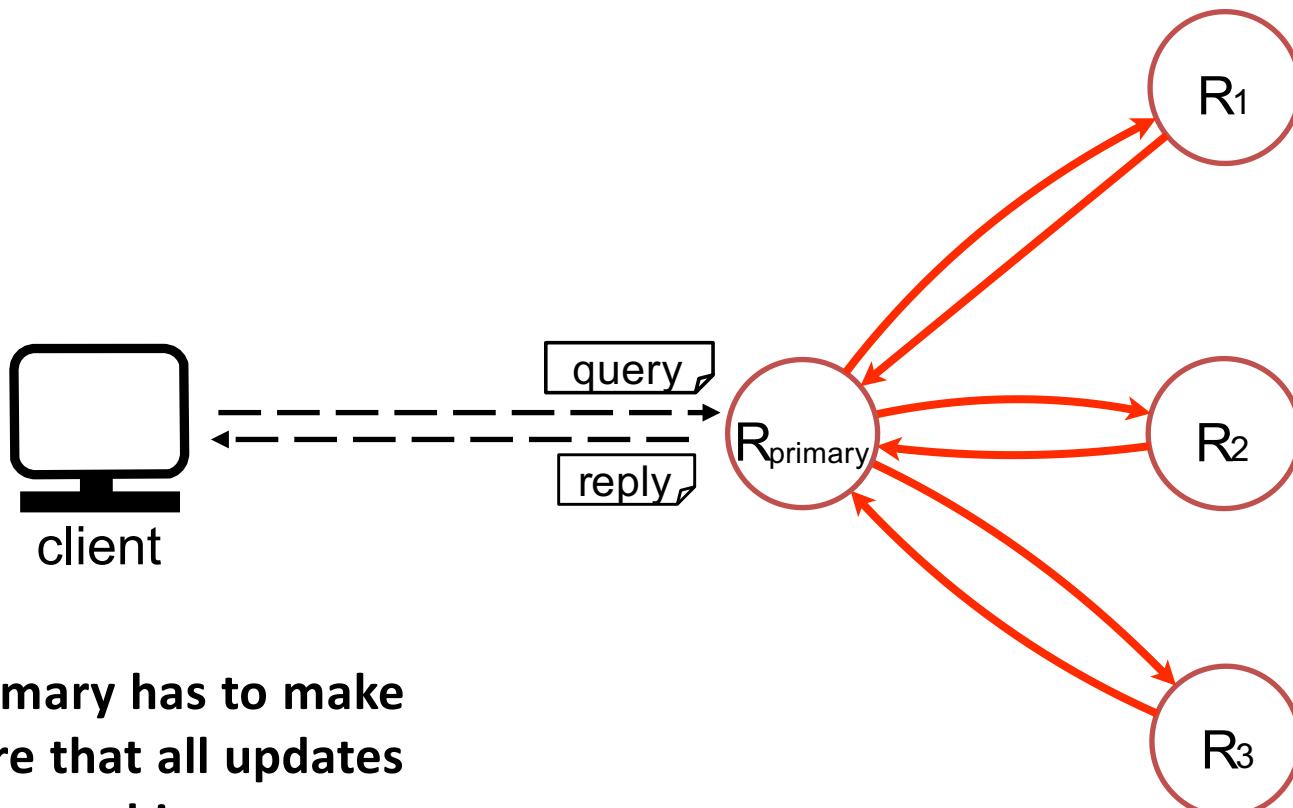


Chain Replication

Higher latency!

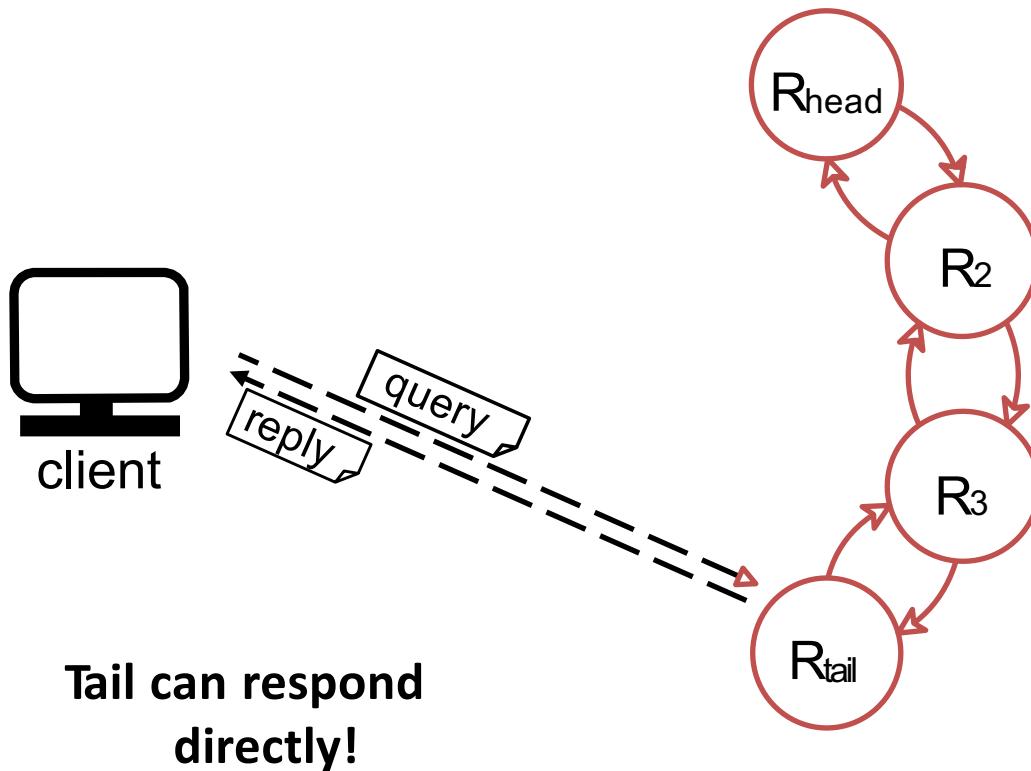


Primary-Backup Replication



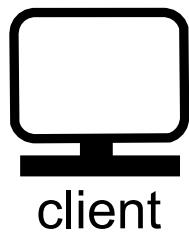
**Primary has to make
sure that all updates
prior to this query are
done!**

Chain Replication

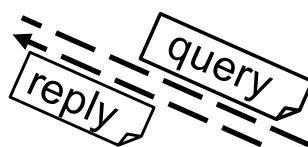


Chain Replication

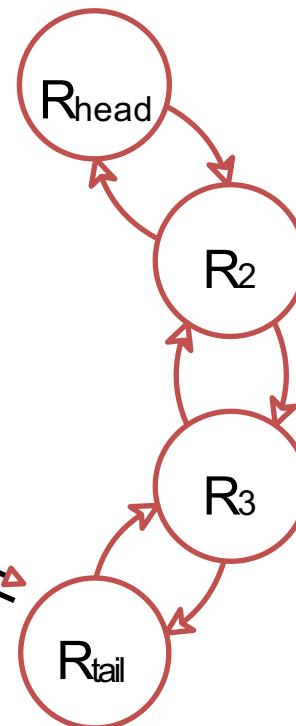
Higher throughput!



client



Tail can respond directly!



Chain Replication: Operations

- Chain replication operations:
 - Updates
 - Queries
 - Failures
 - Reconfigurations

Updates

Updates

Speculative
History

Speculative
History

Speculative
History

Speculative
History

R_{head}

R_2

R_3

R_{tail}

Stable
History

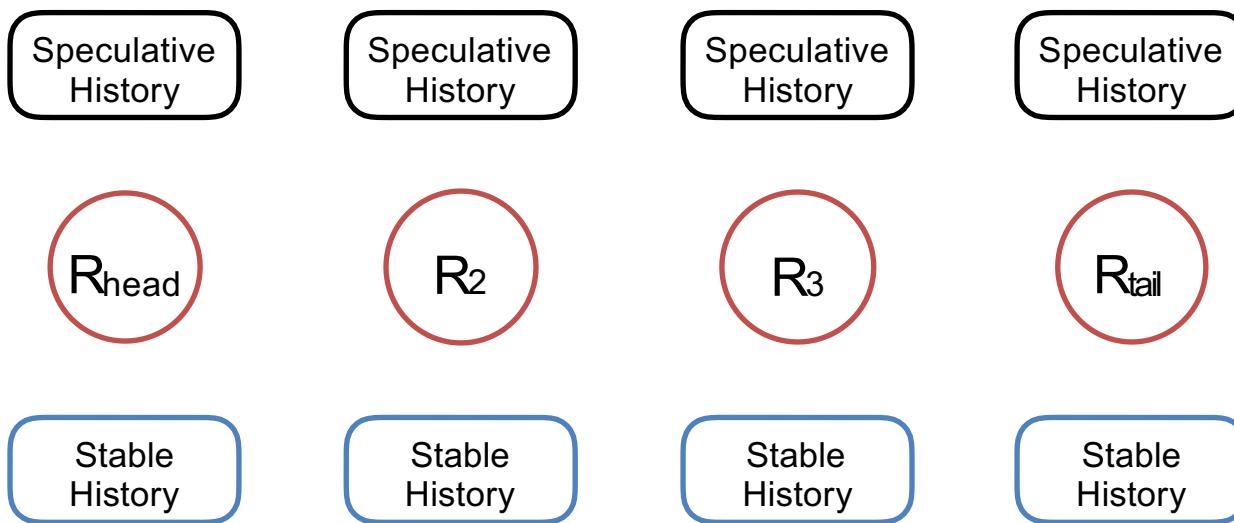
Stable
History

Stable
History

Stable
History

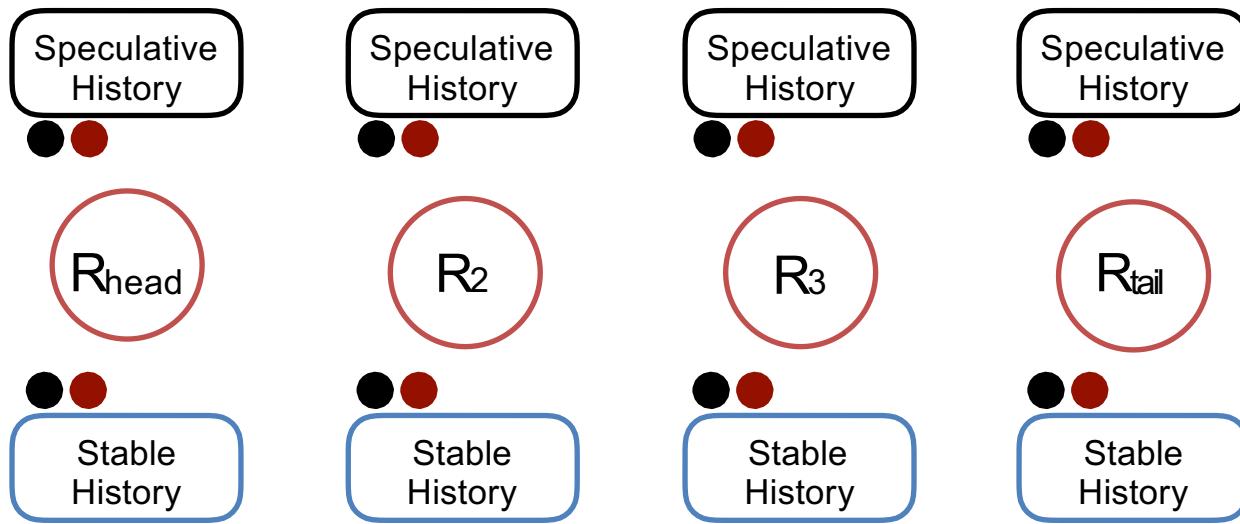
Updates

R_2 is the **predecessor** of R_3

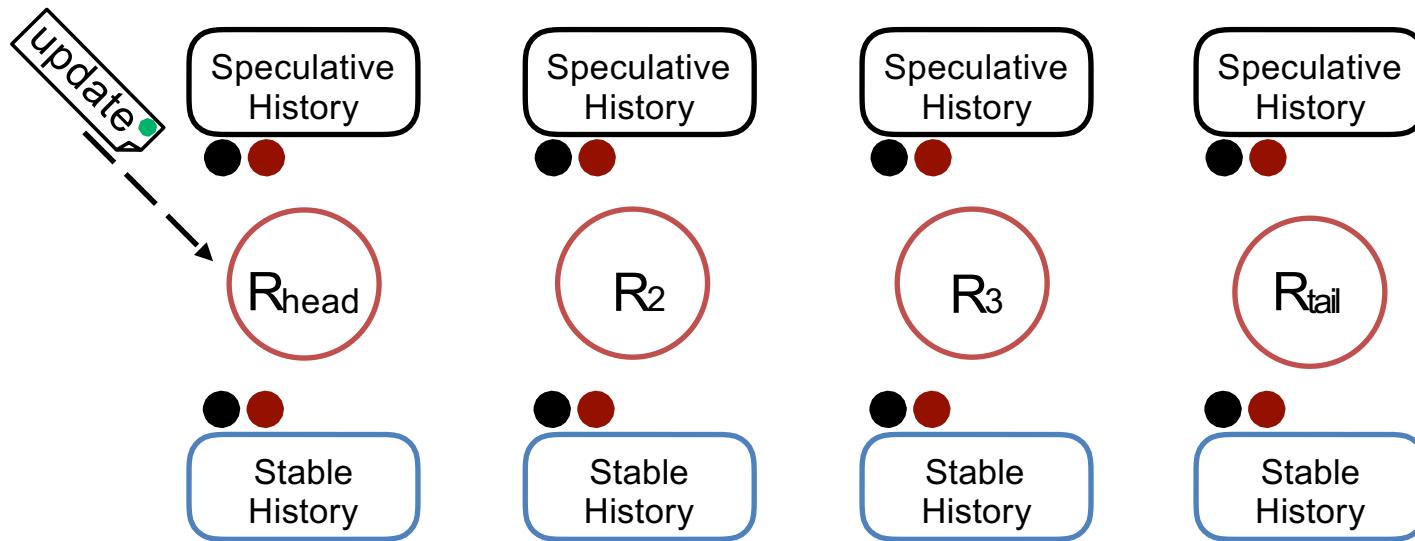


R_3 is the **successor** of R_2

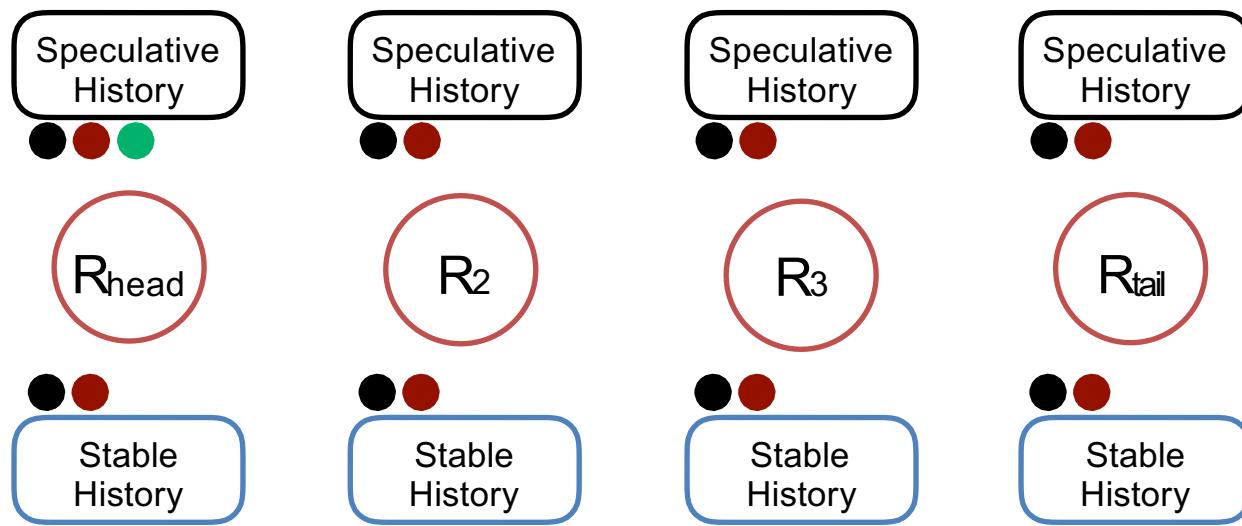
Updates



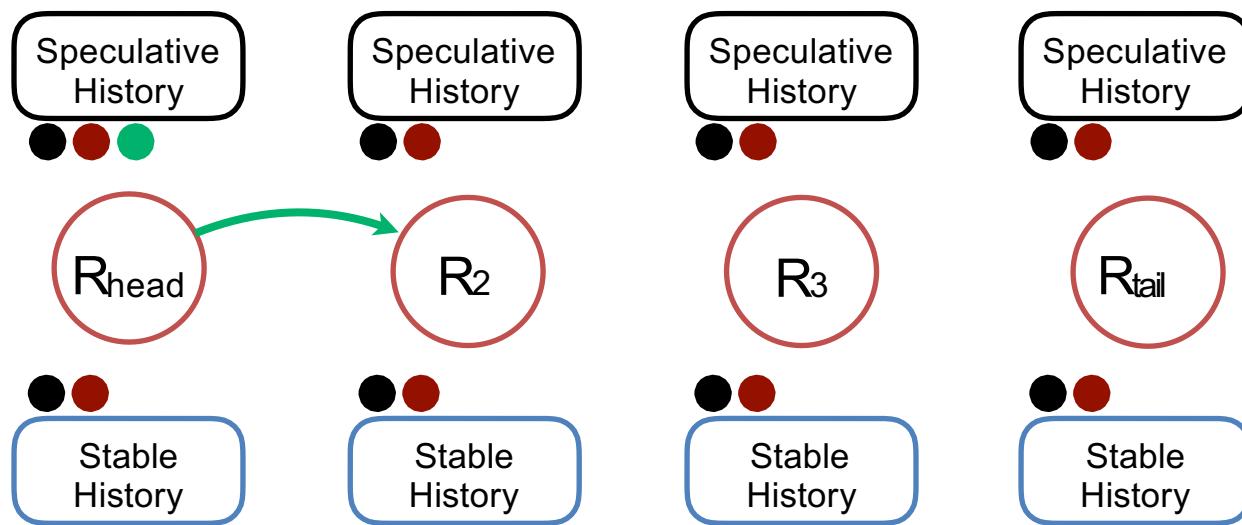
Updates



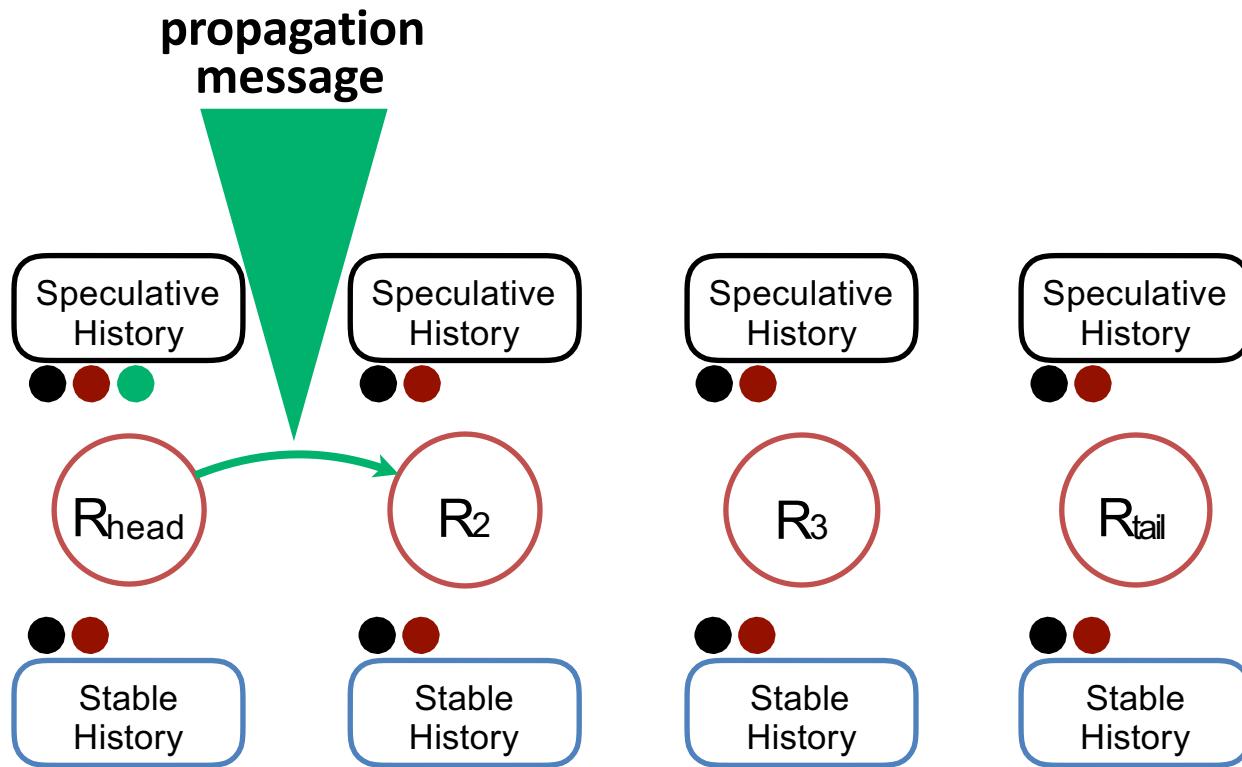
Updates



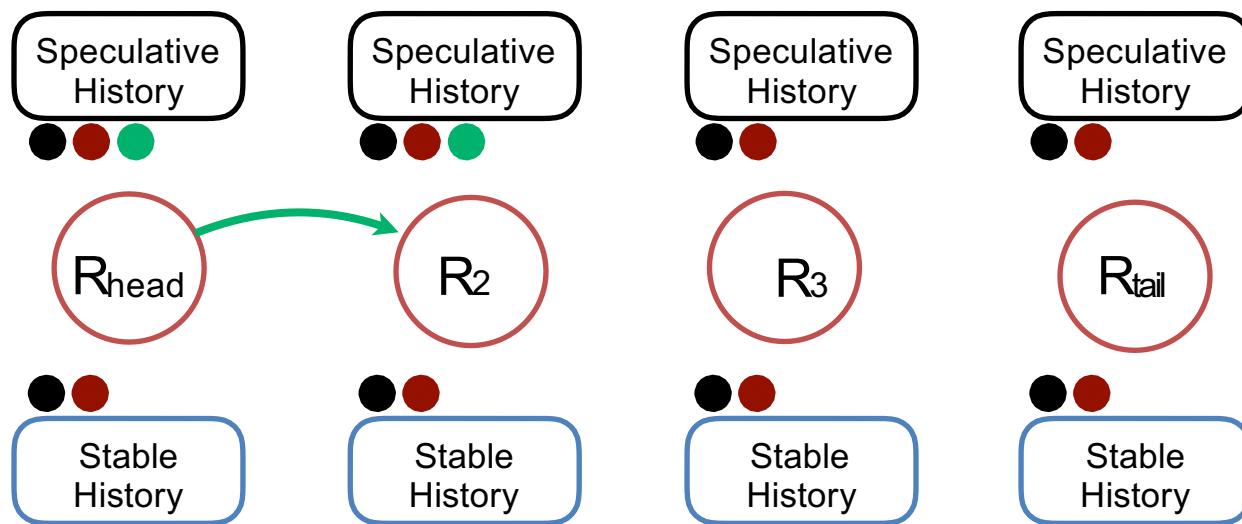
Updates



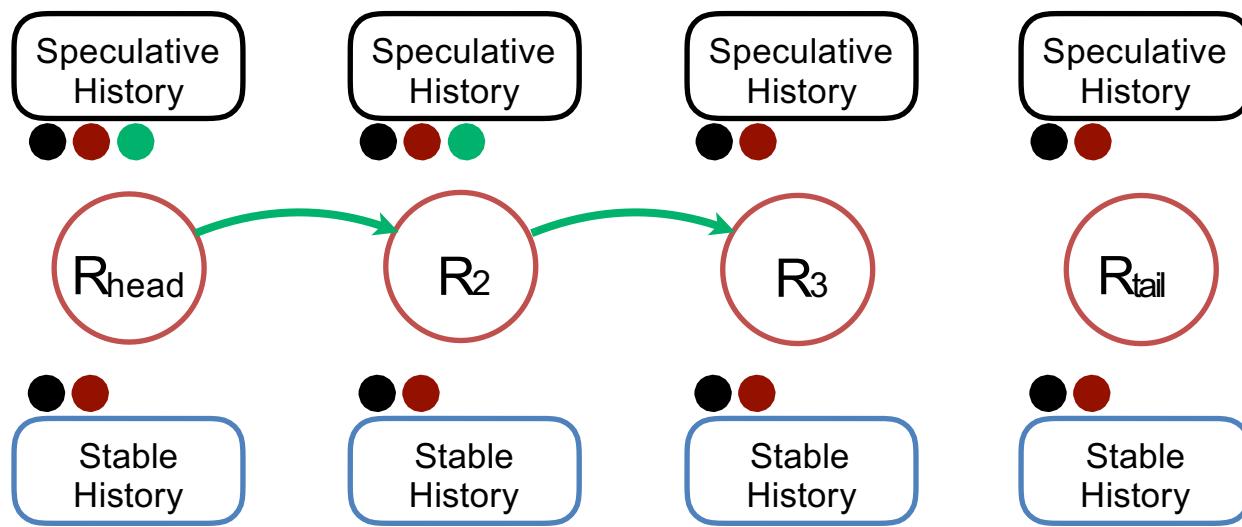
Updates



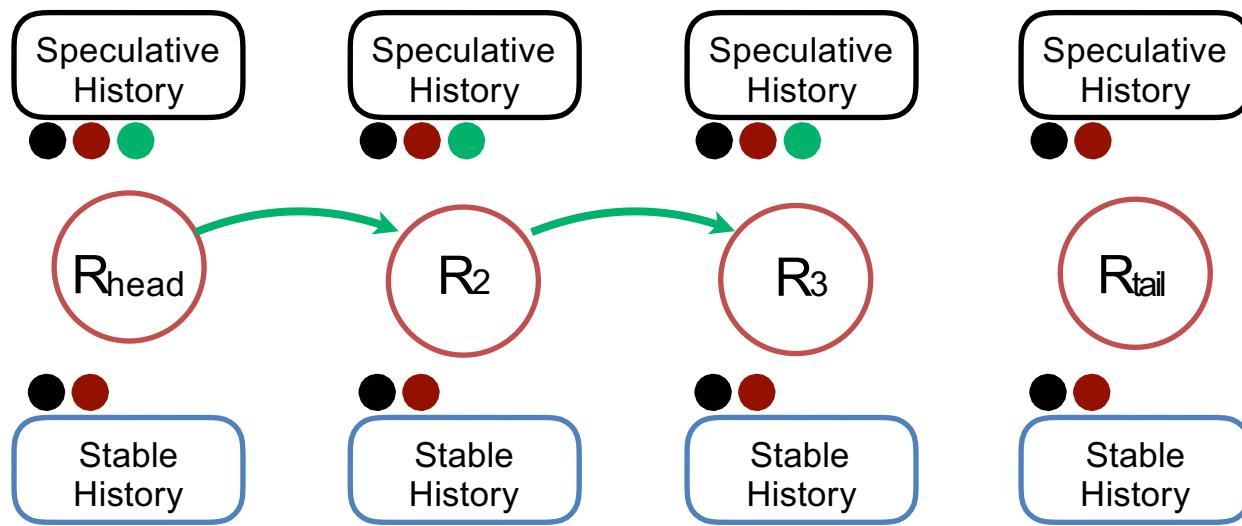
Updates



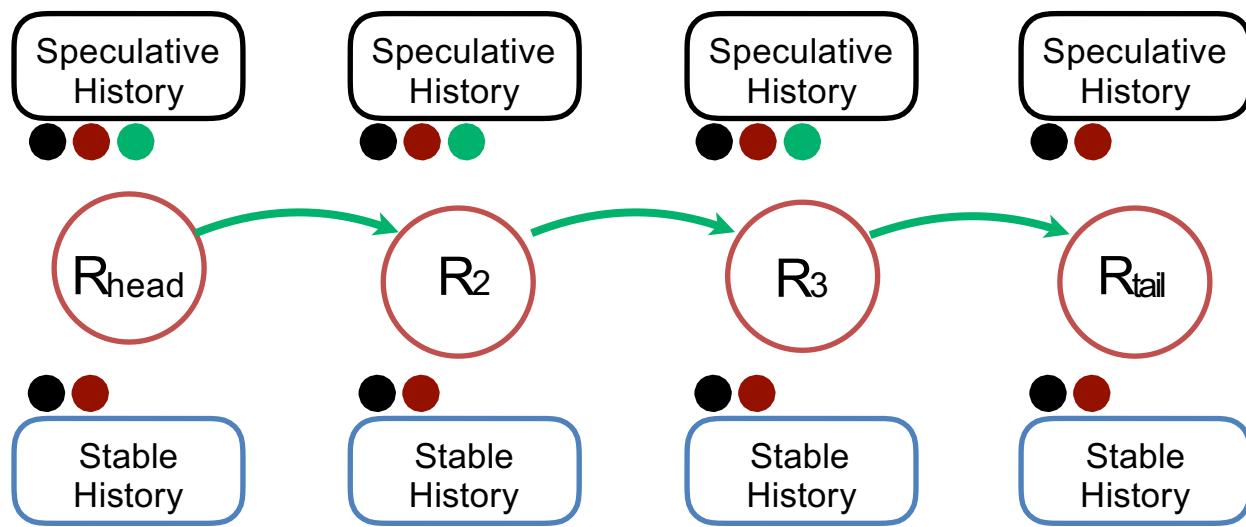
Updates



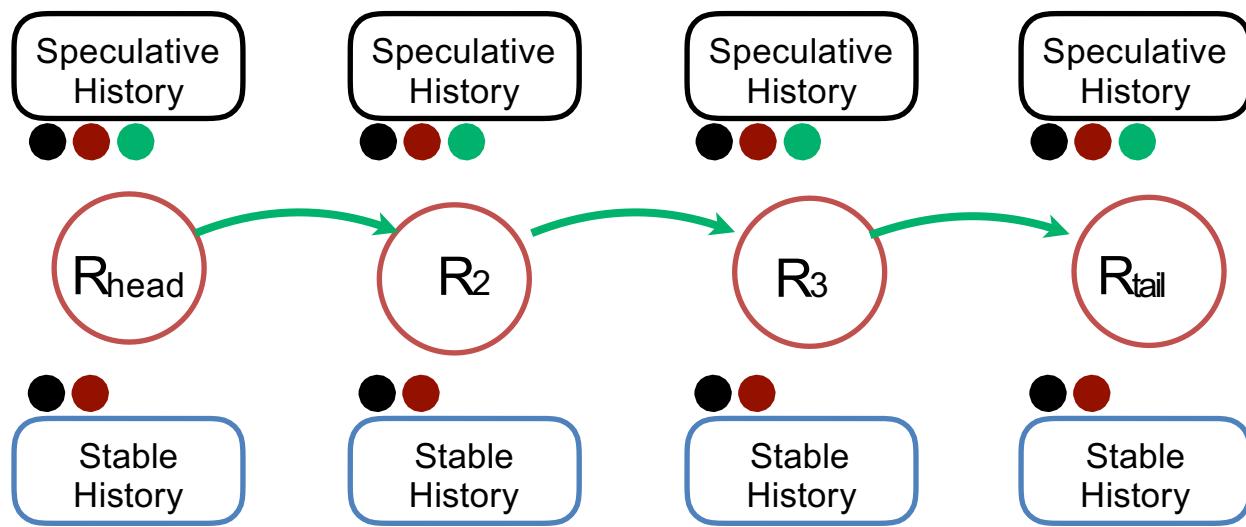
Updates



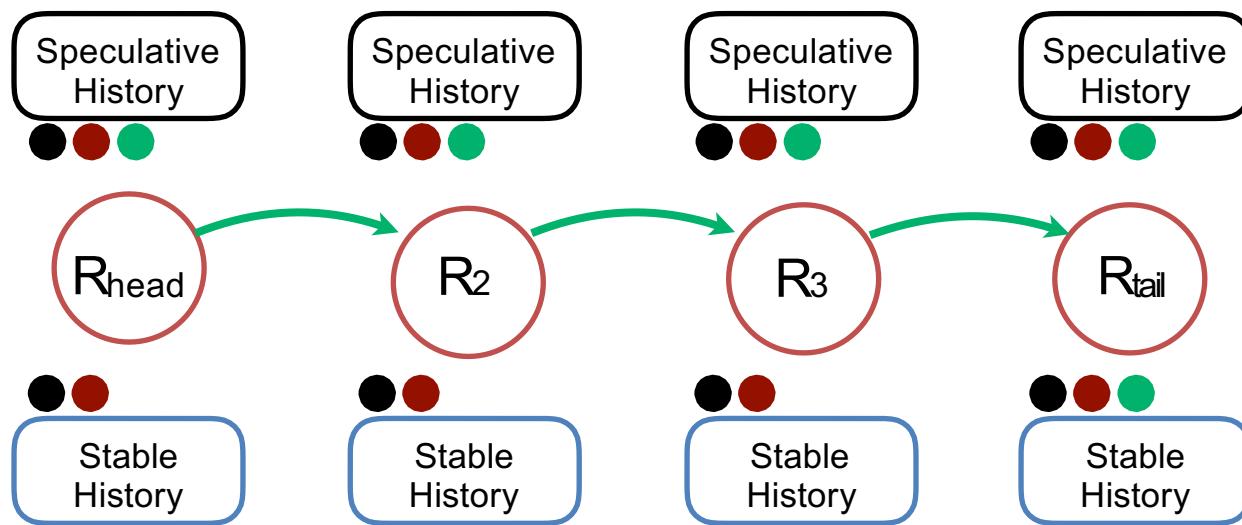
Updates



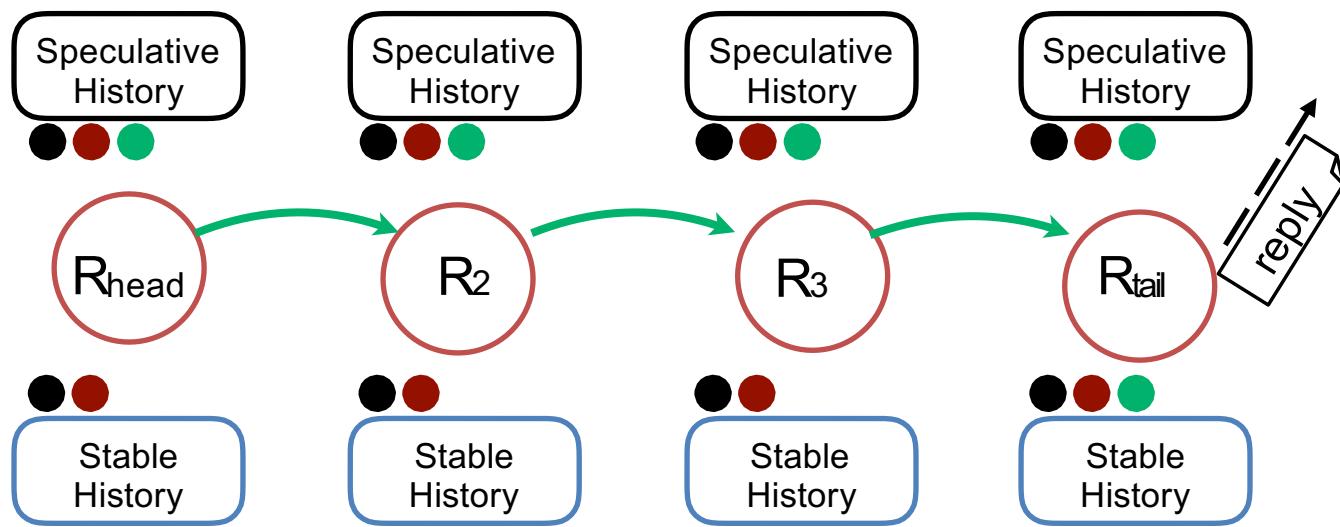
Updates



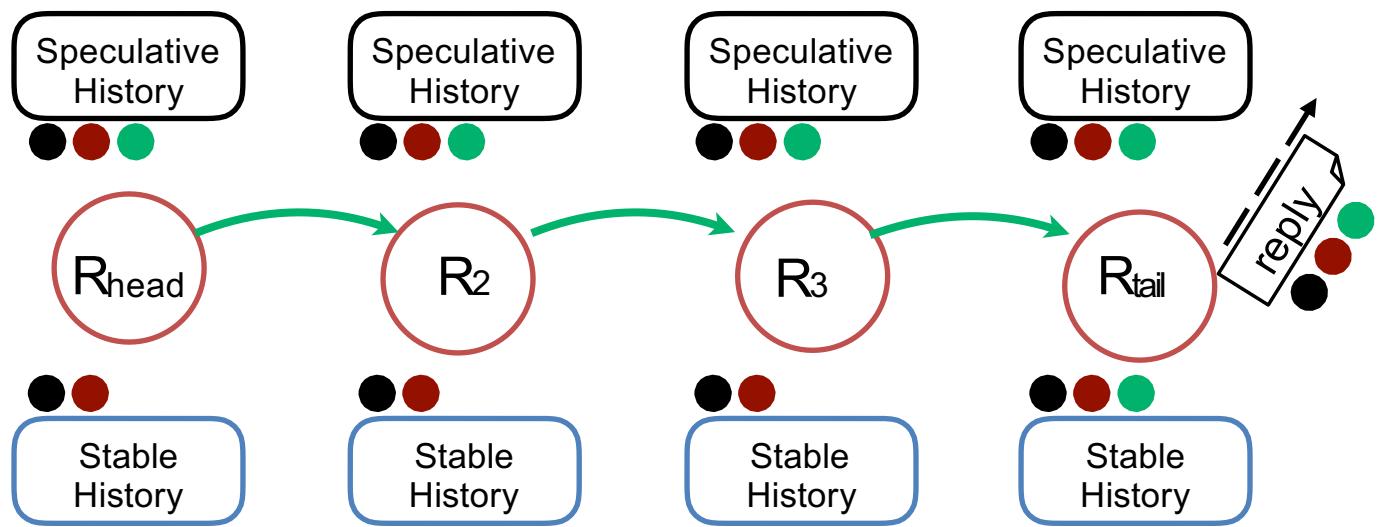
Updates



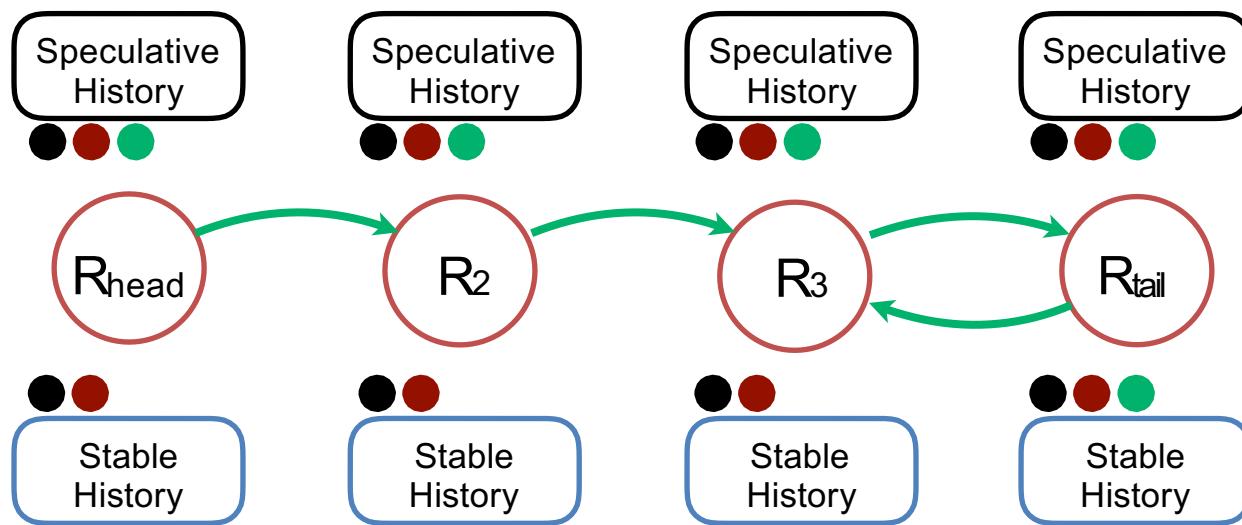
Updates



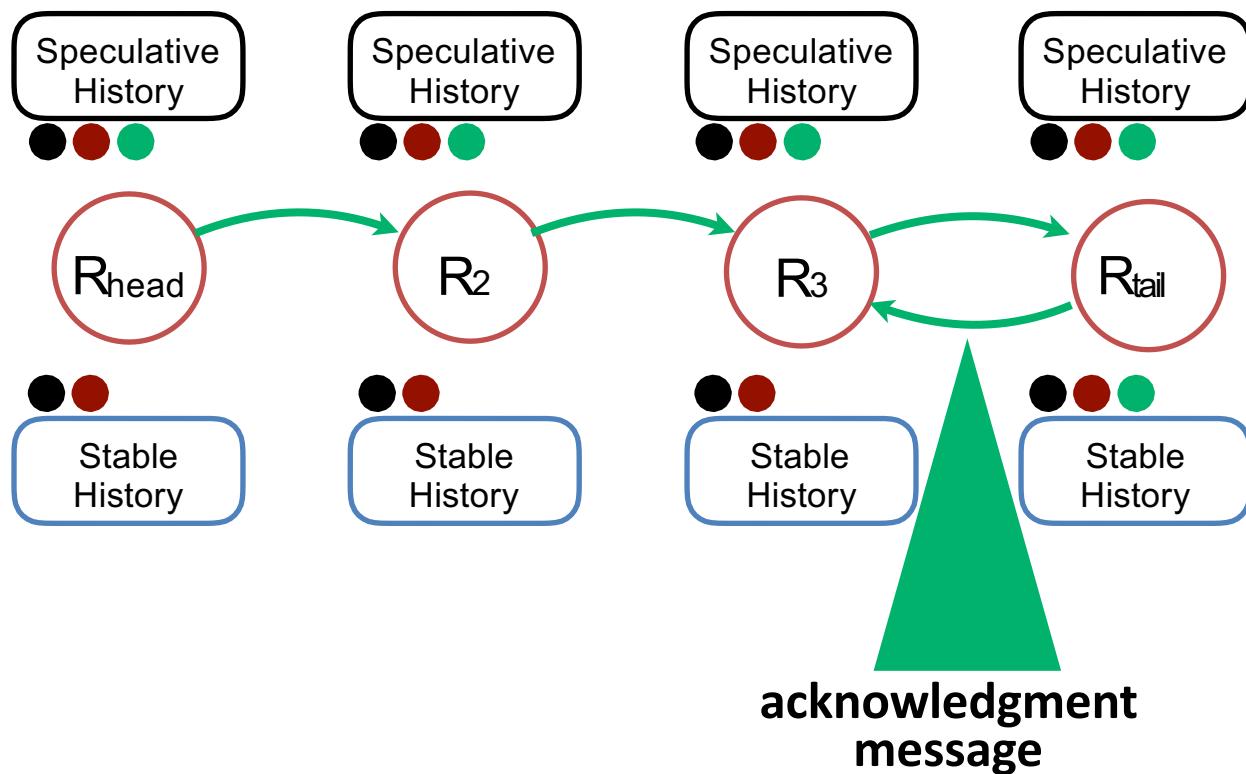
Updates



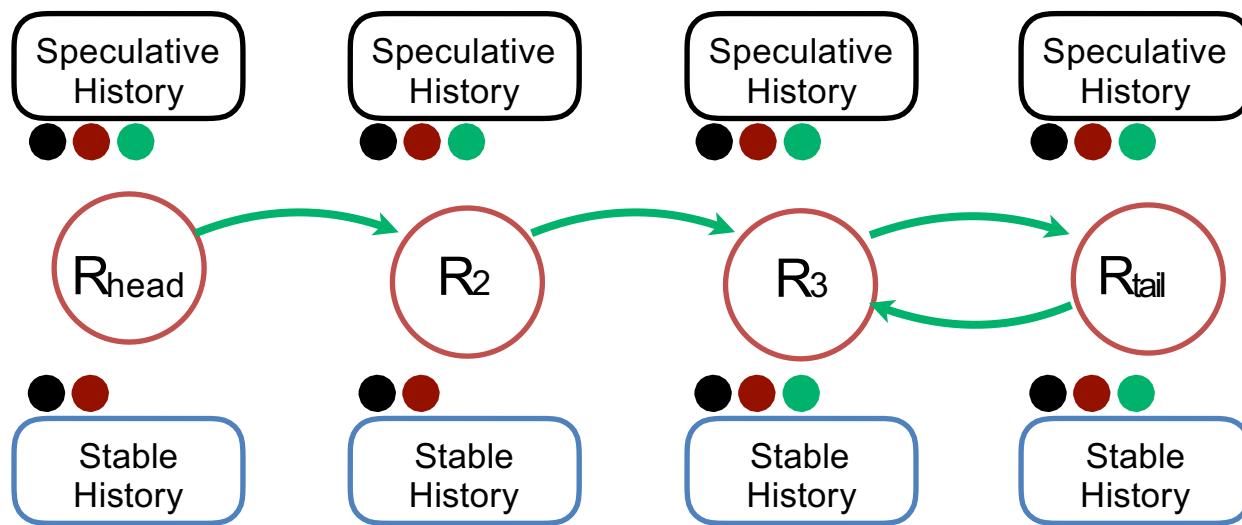
Updates



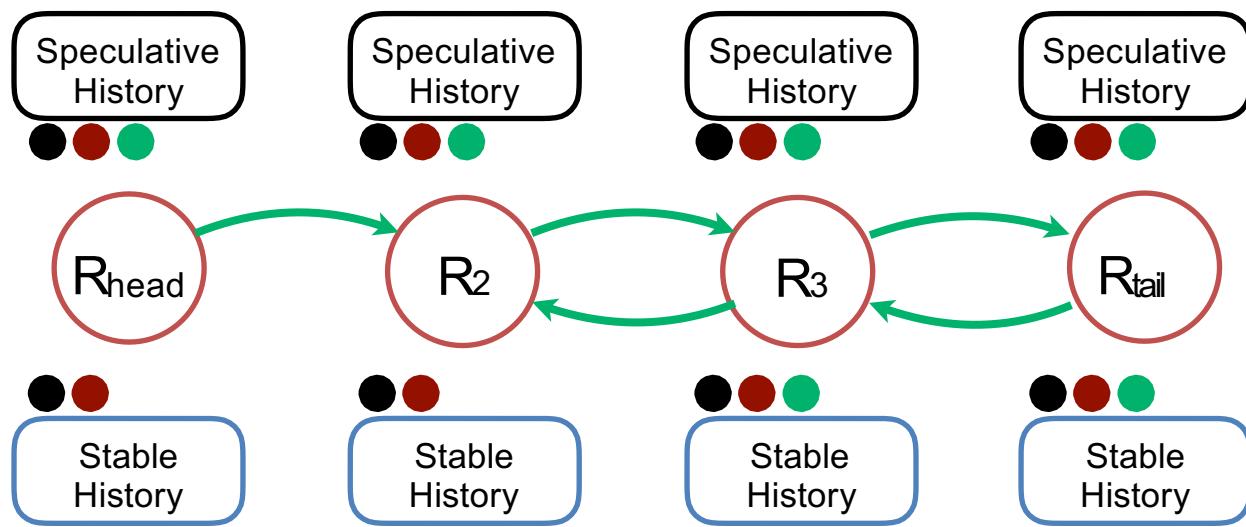
Updates



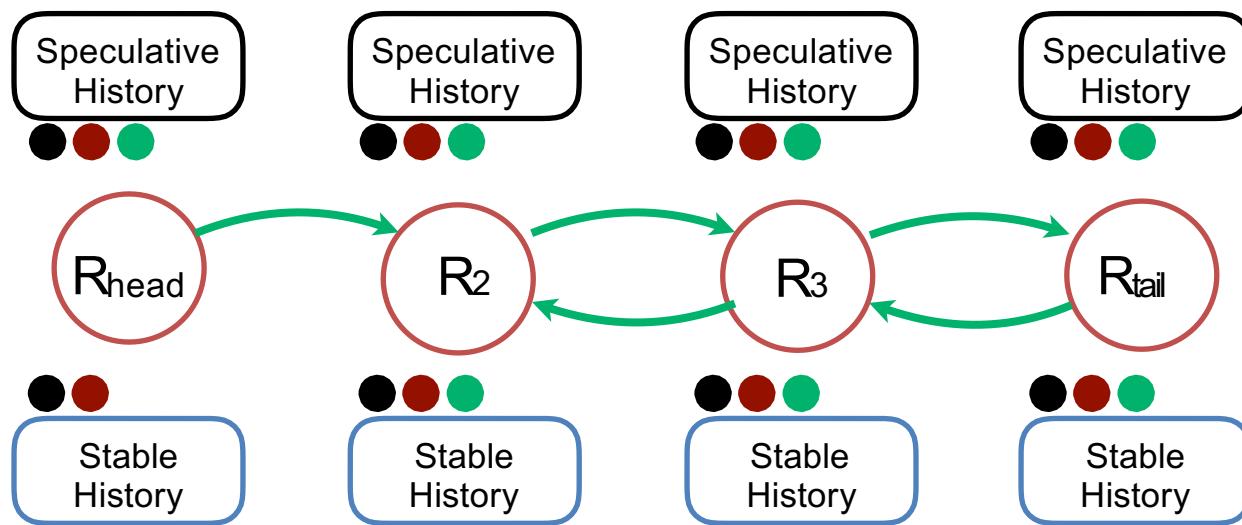
Updates



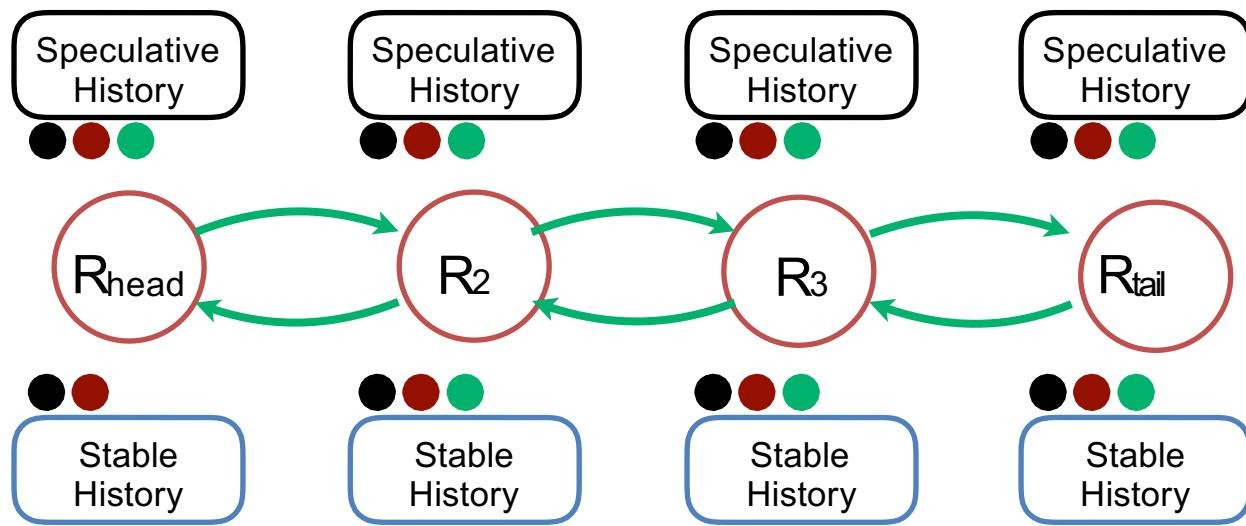
Updates



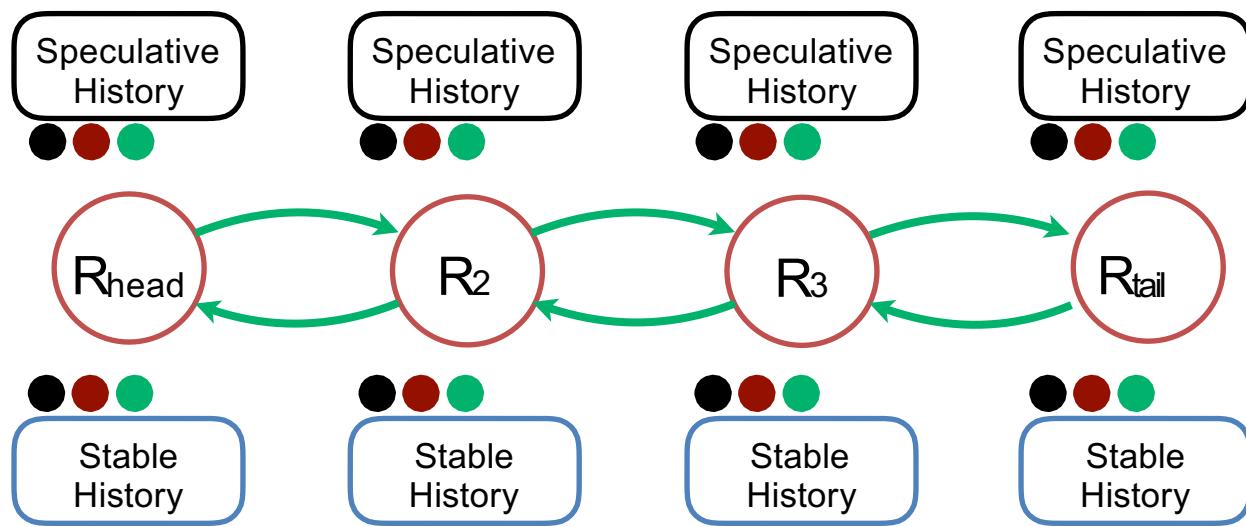
Updates



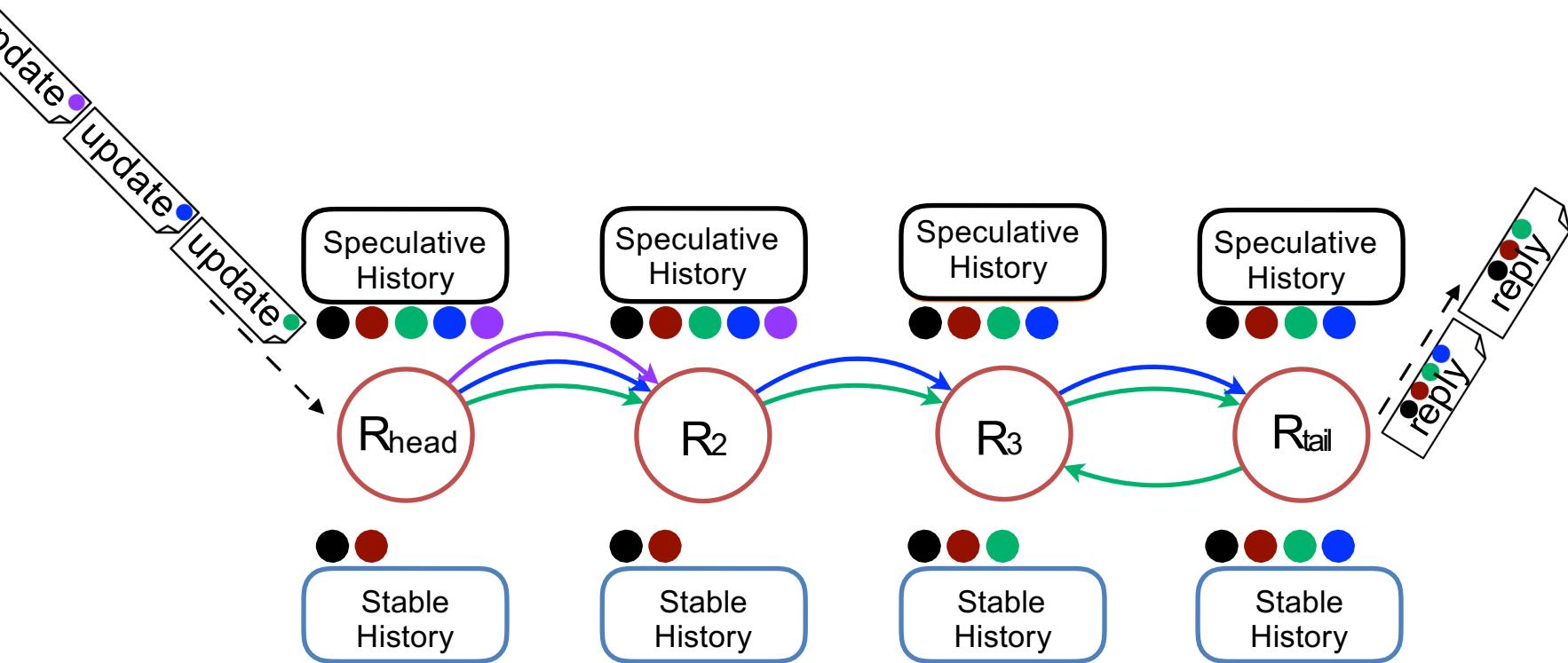
Updates



Updates

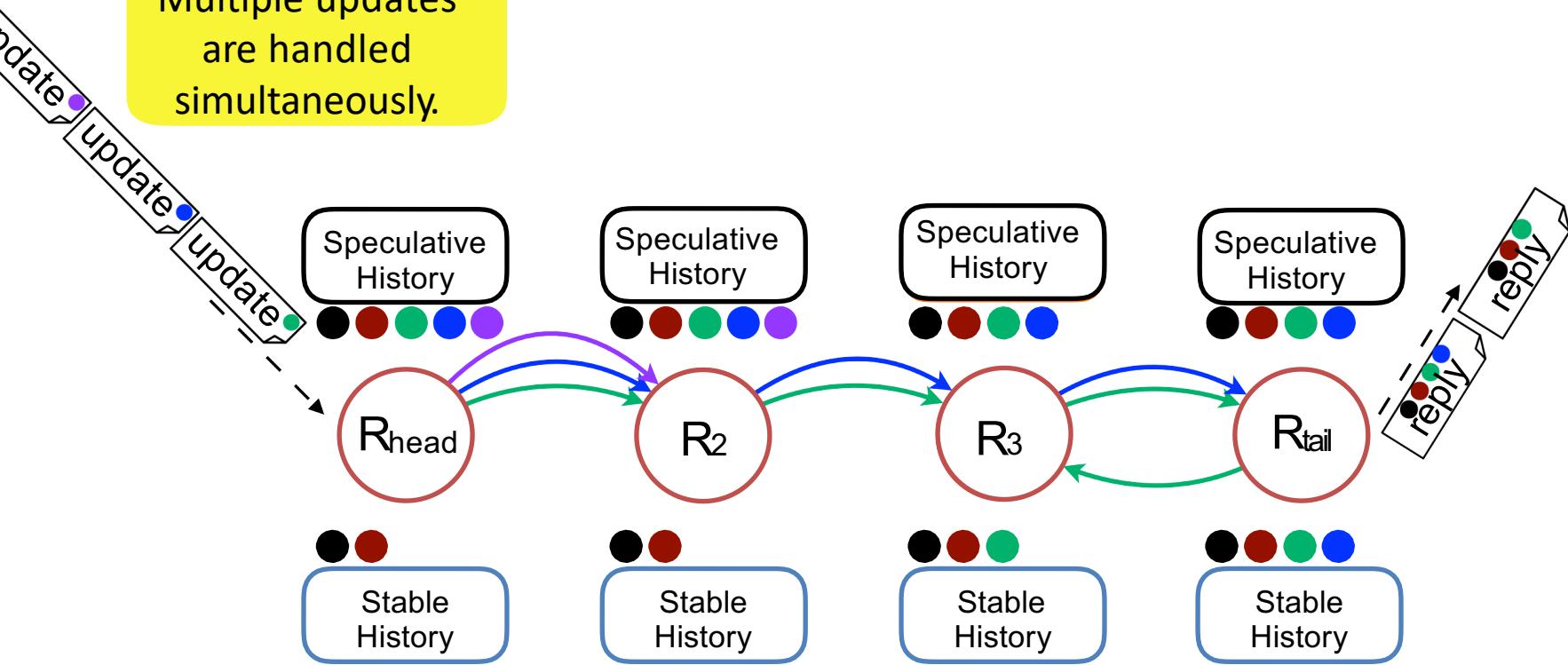


Updates

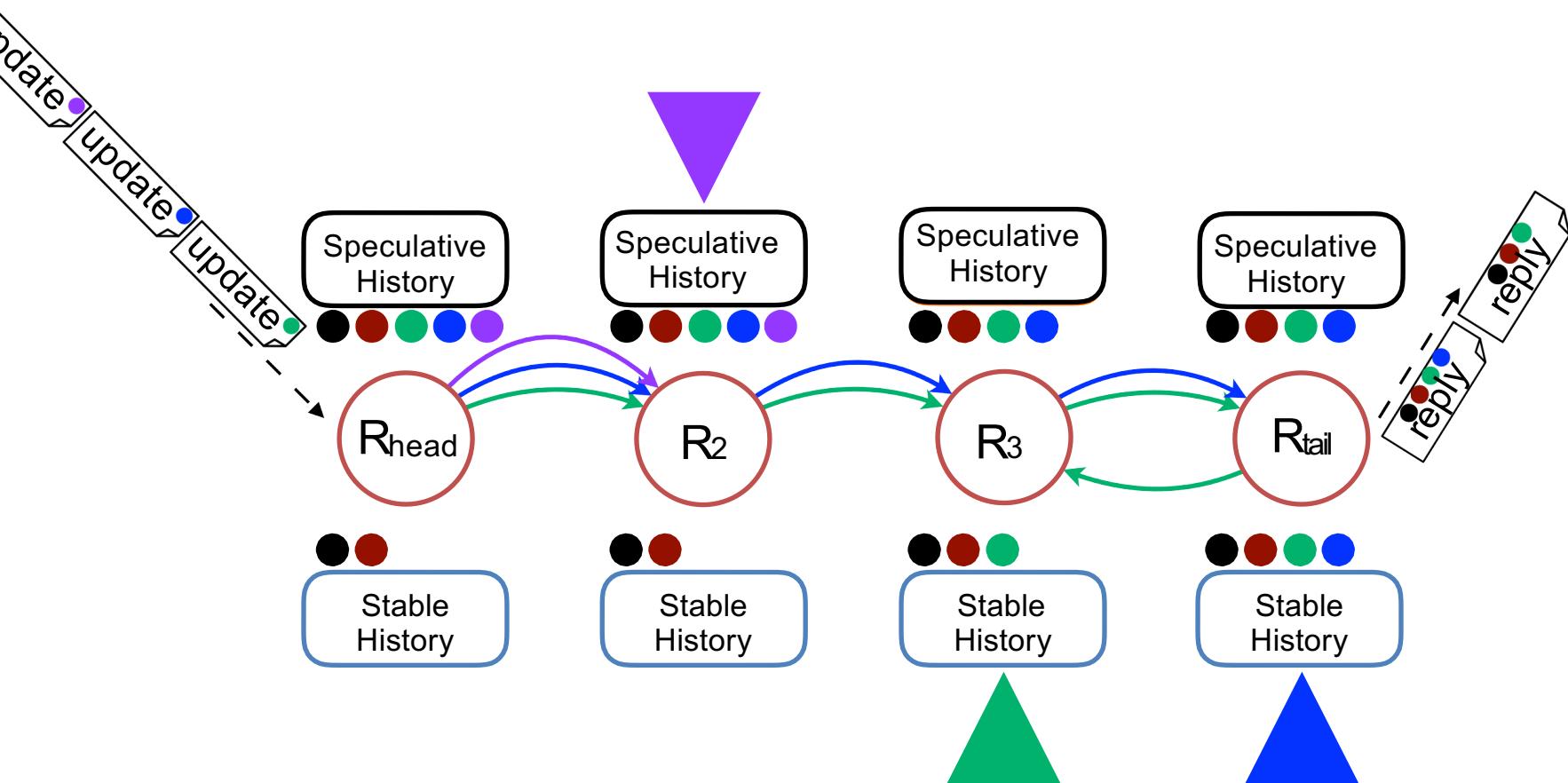


Updates

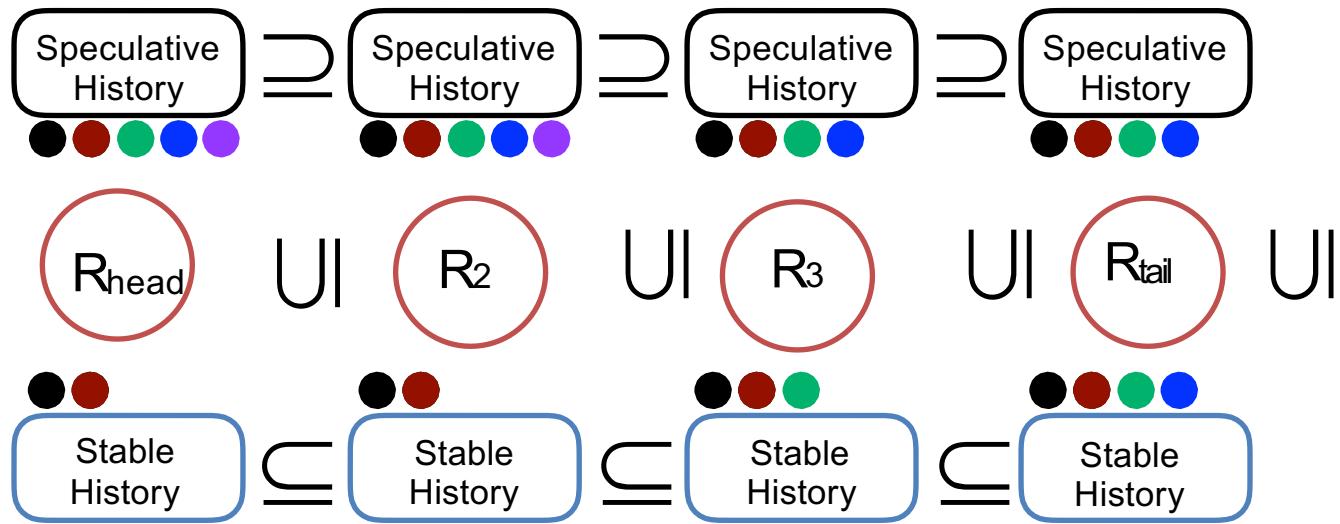
Multiple updates
are handled
simultaneously.

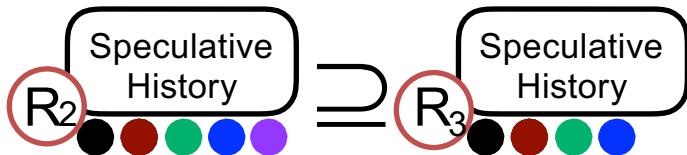


Updates

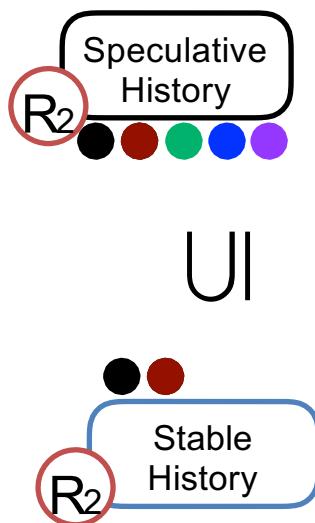


Updates

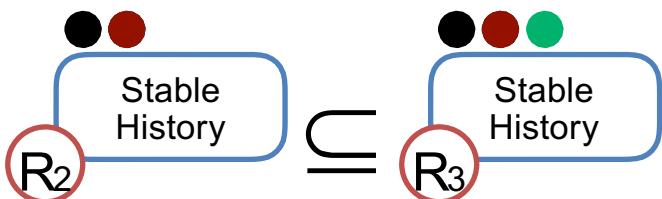




The speculative history of a node's successor is a subset of that node's speculative history.



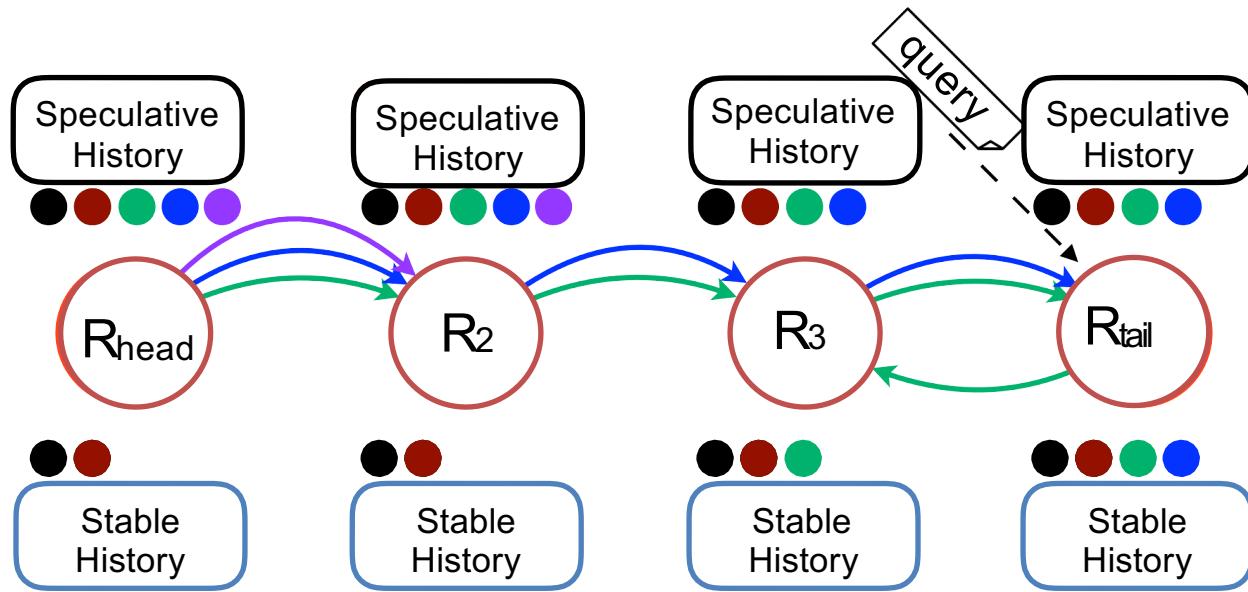
The speculative history of a node is a superset of its stable history.



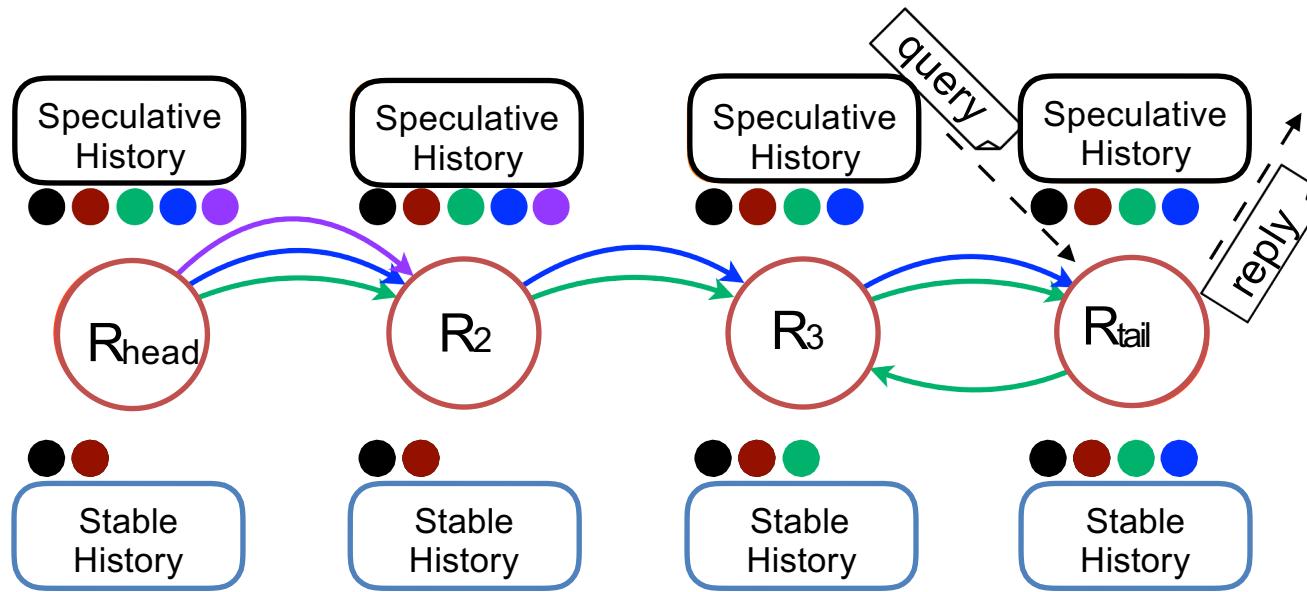
The stable history of a node's successor is a superset of that node's stable history.

Queries

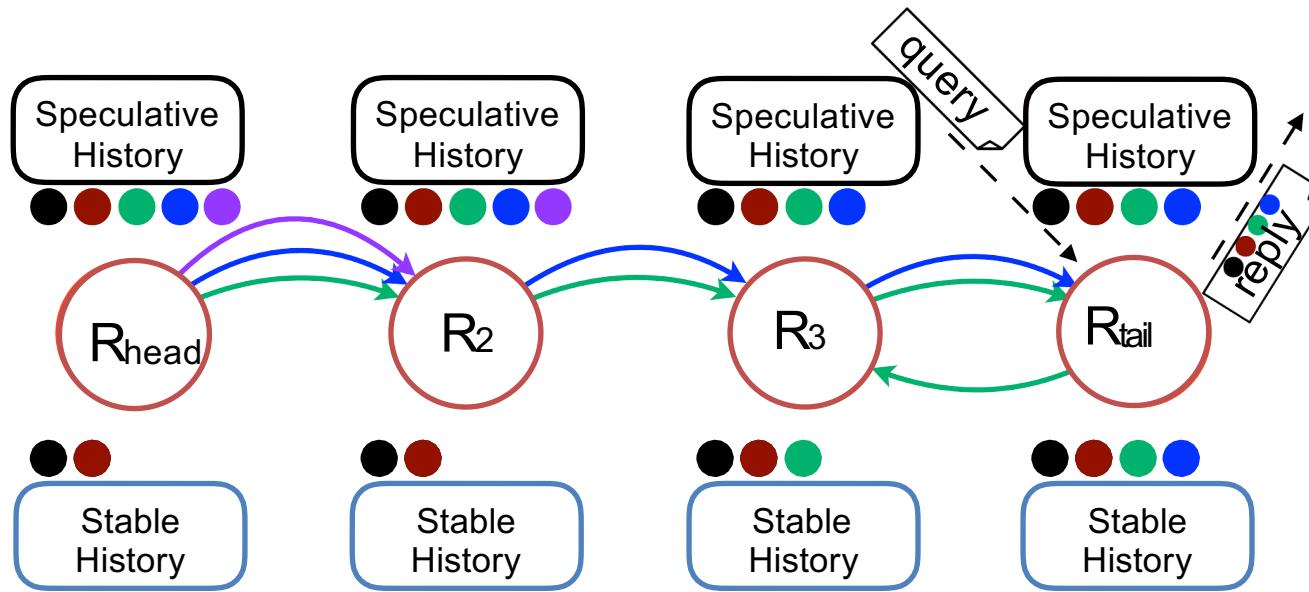
Queries



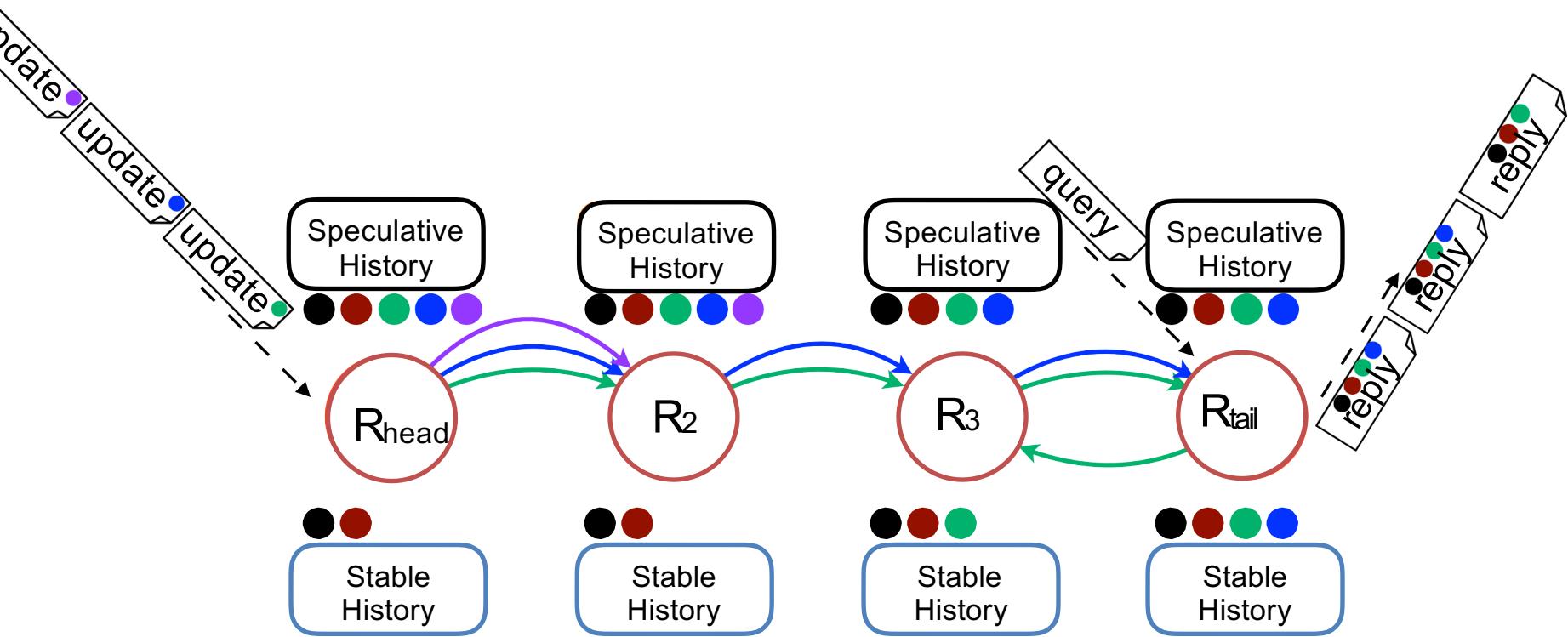
Queries



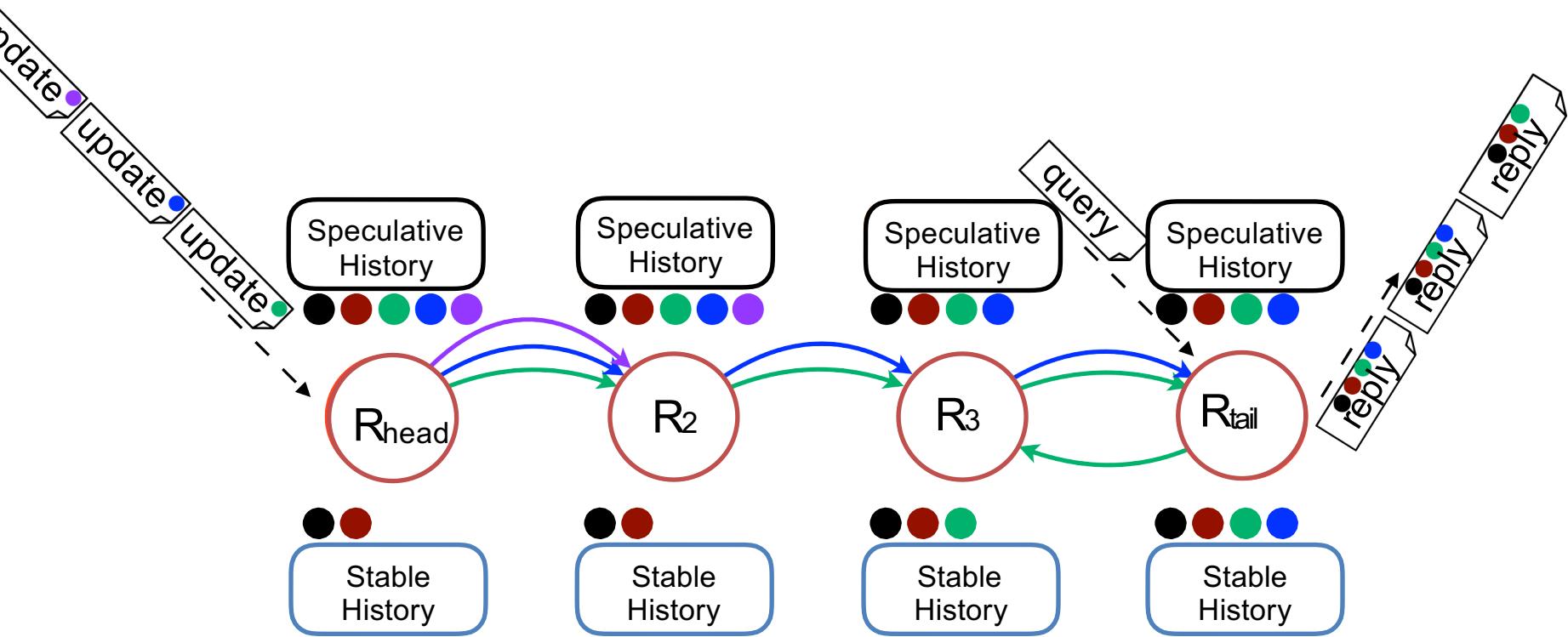
Queries



Queries



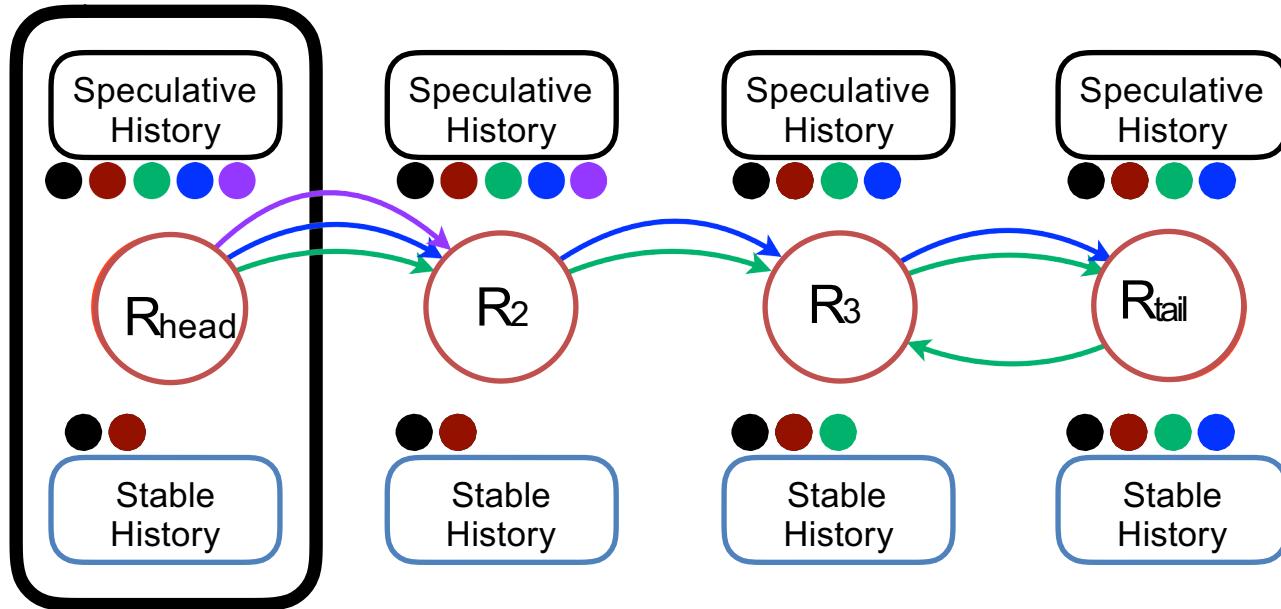
Queries



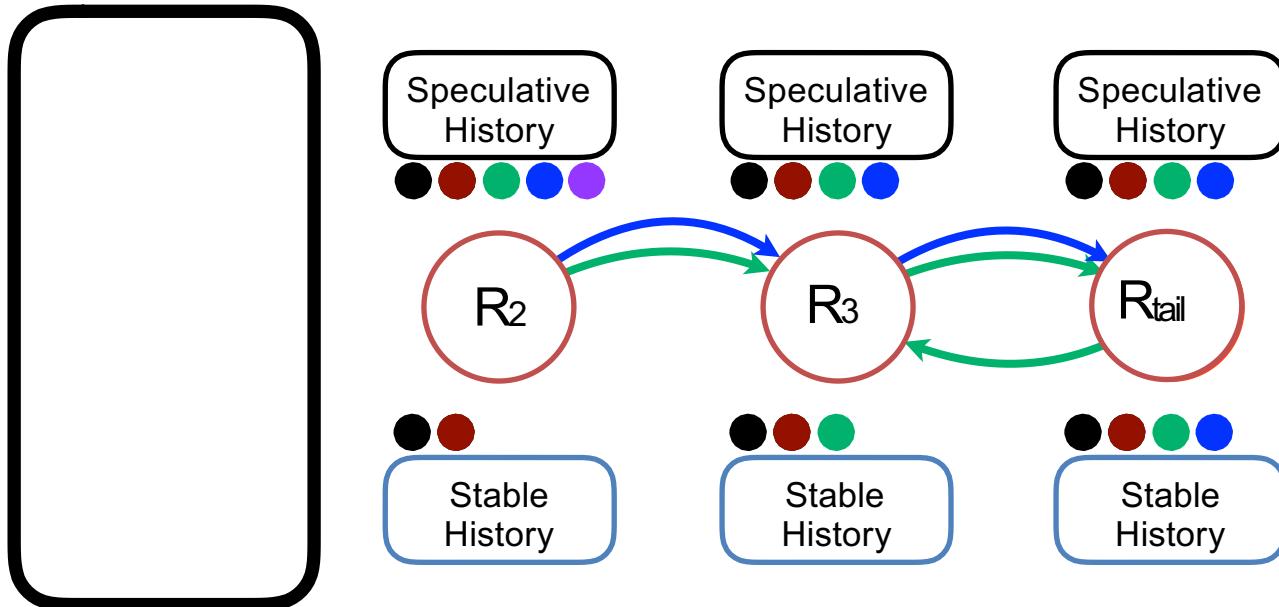
The tail is the point of linearization!

Failures

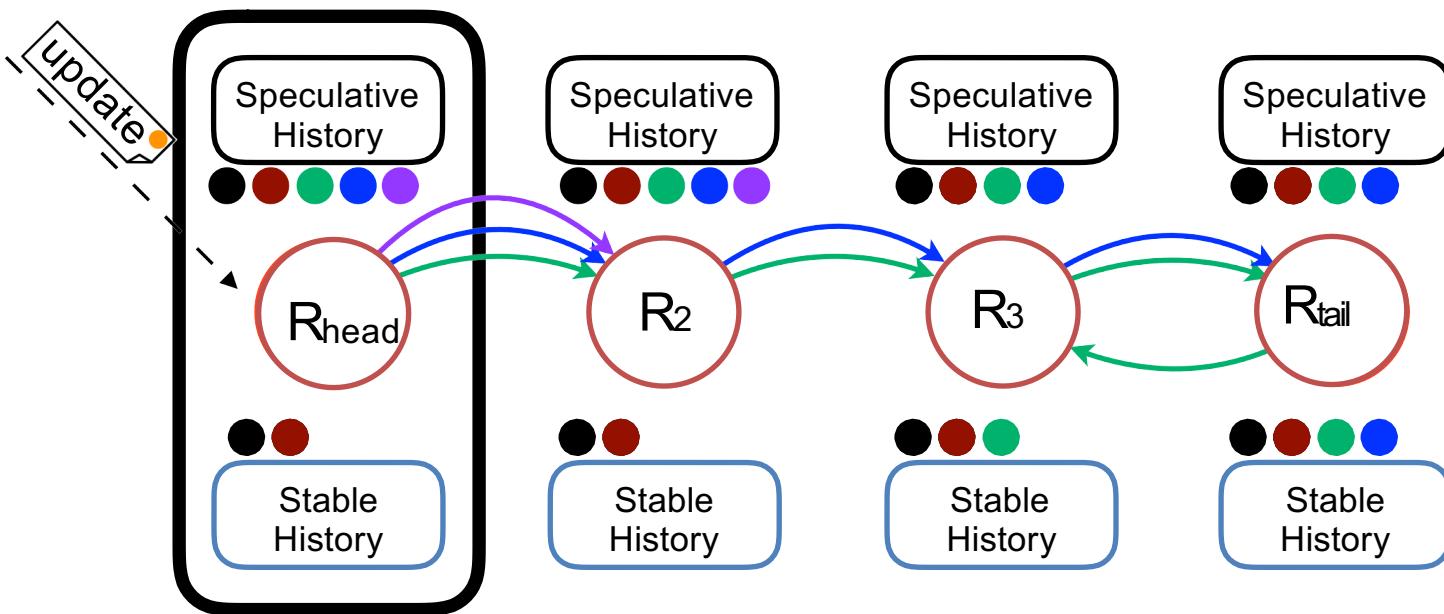
Head Failures



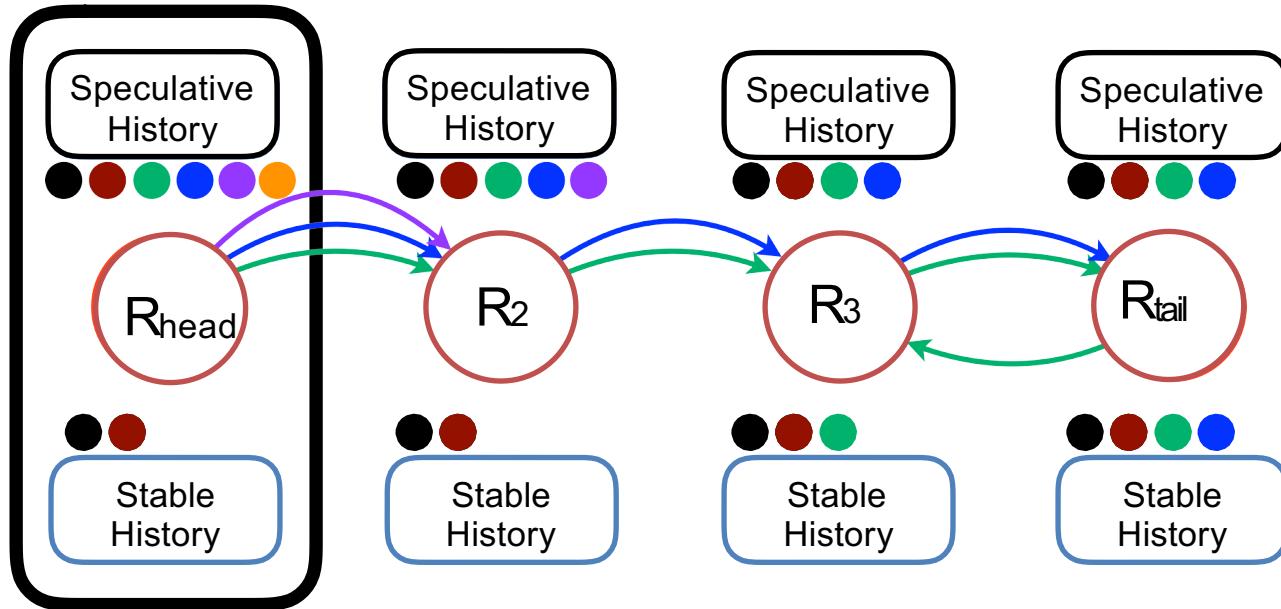
Head Failures



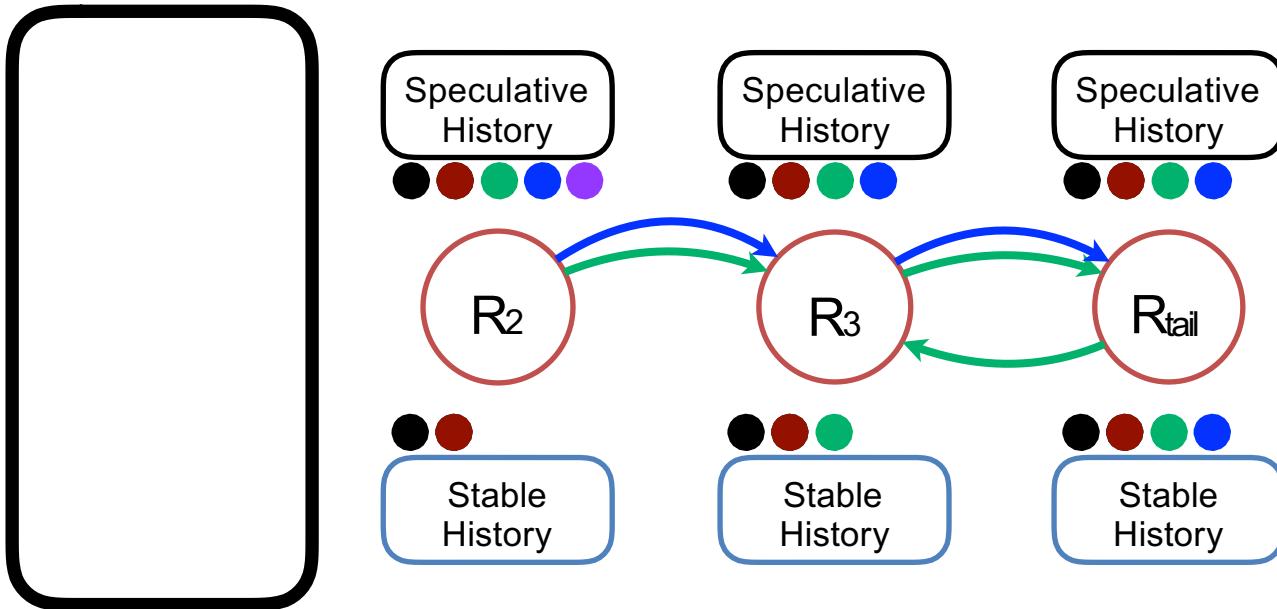
Head Failures



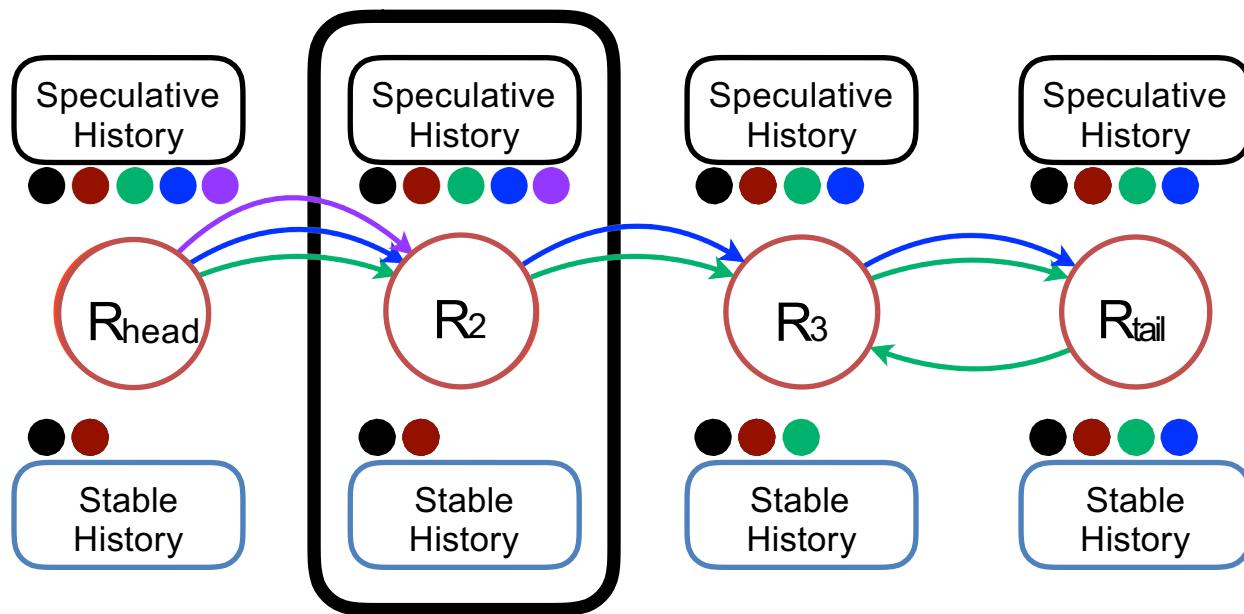
Head Failures



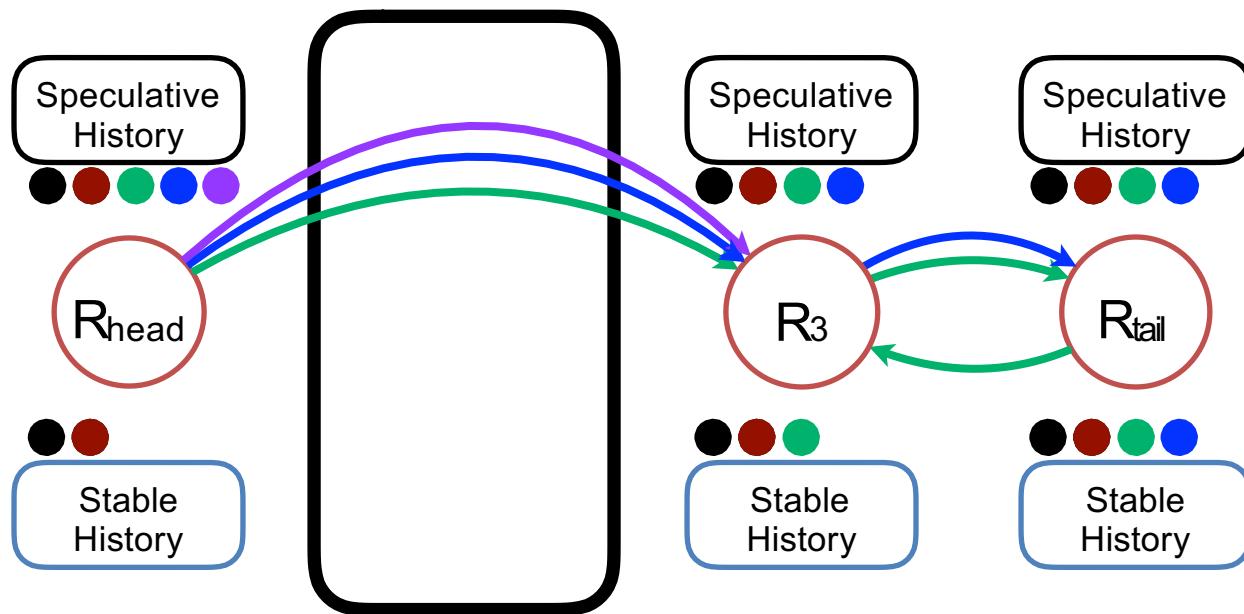
Head Failures



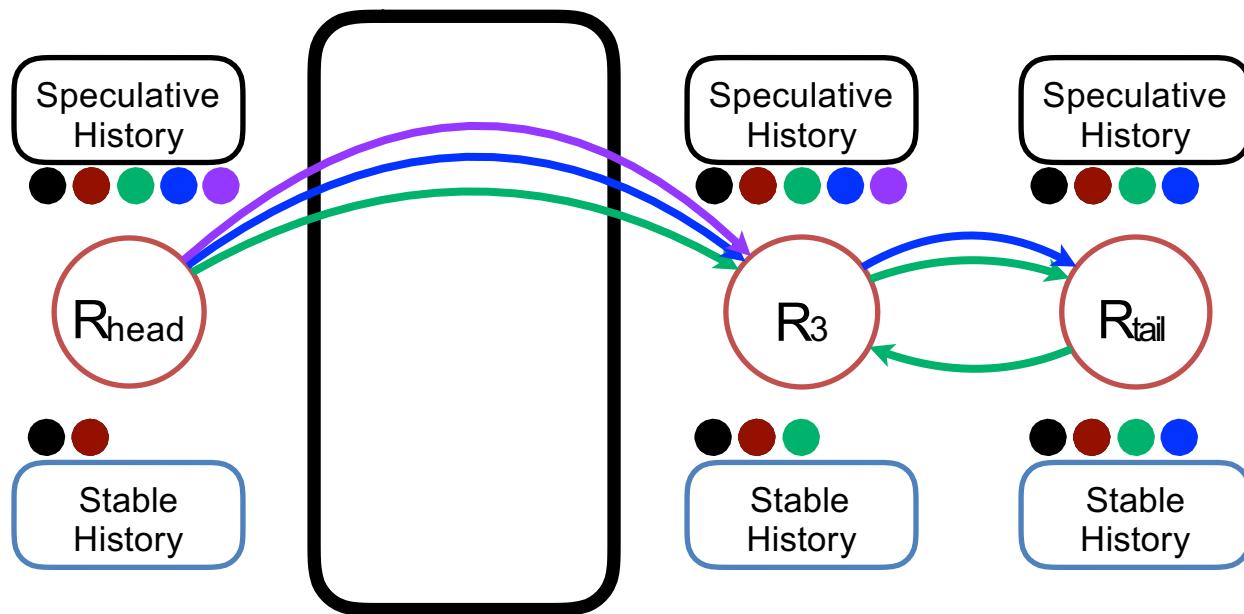
Middle Node Failures



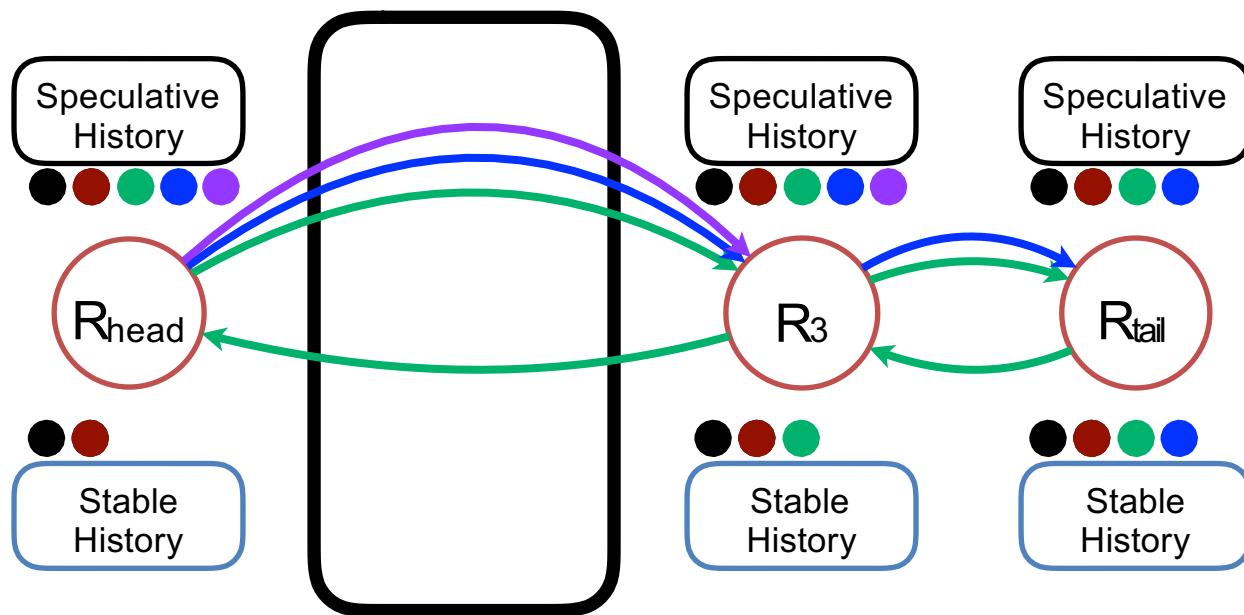
Middle Node Failures



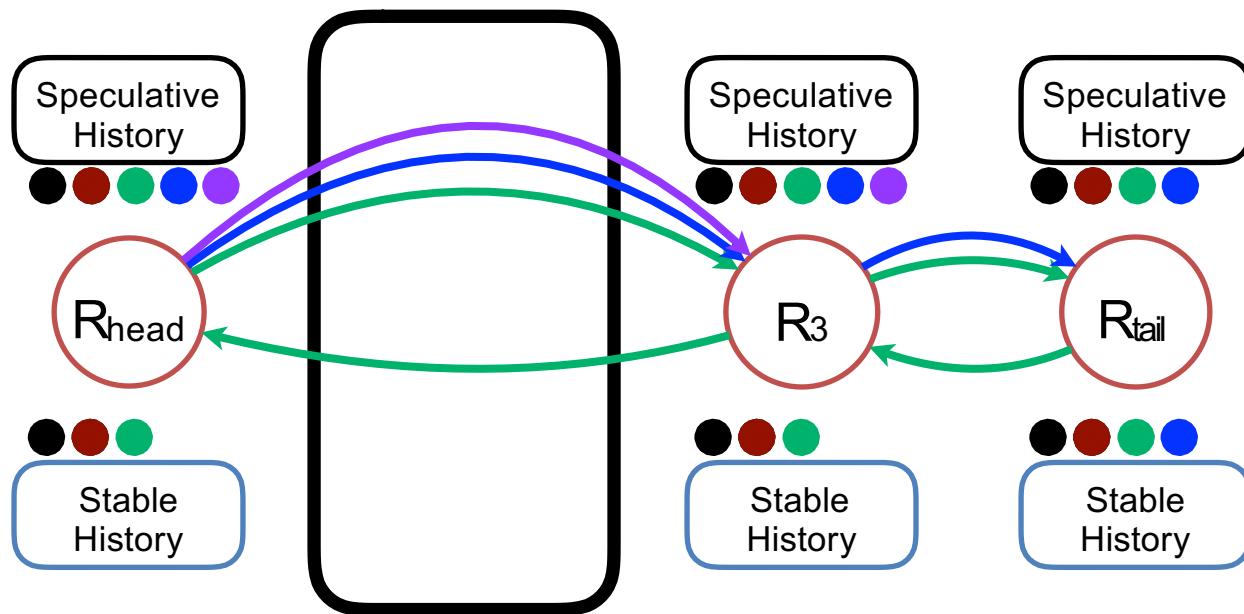
Middle Node Failures



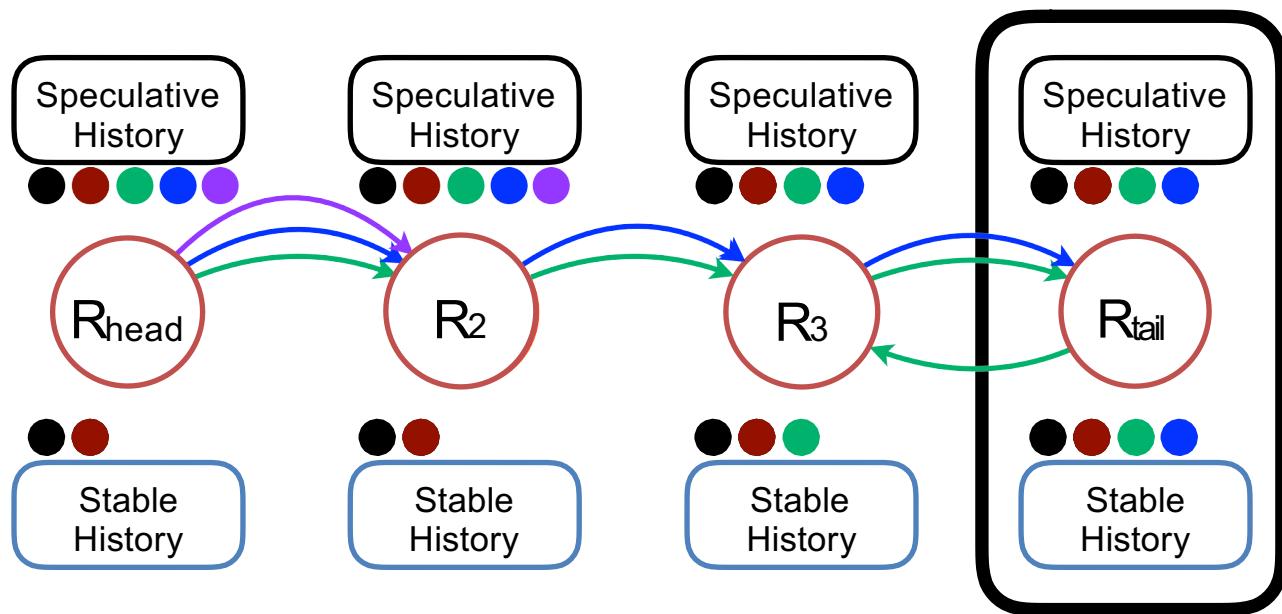
Middle Node Failures



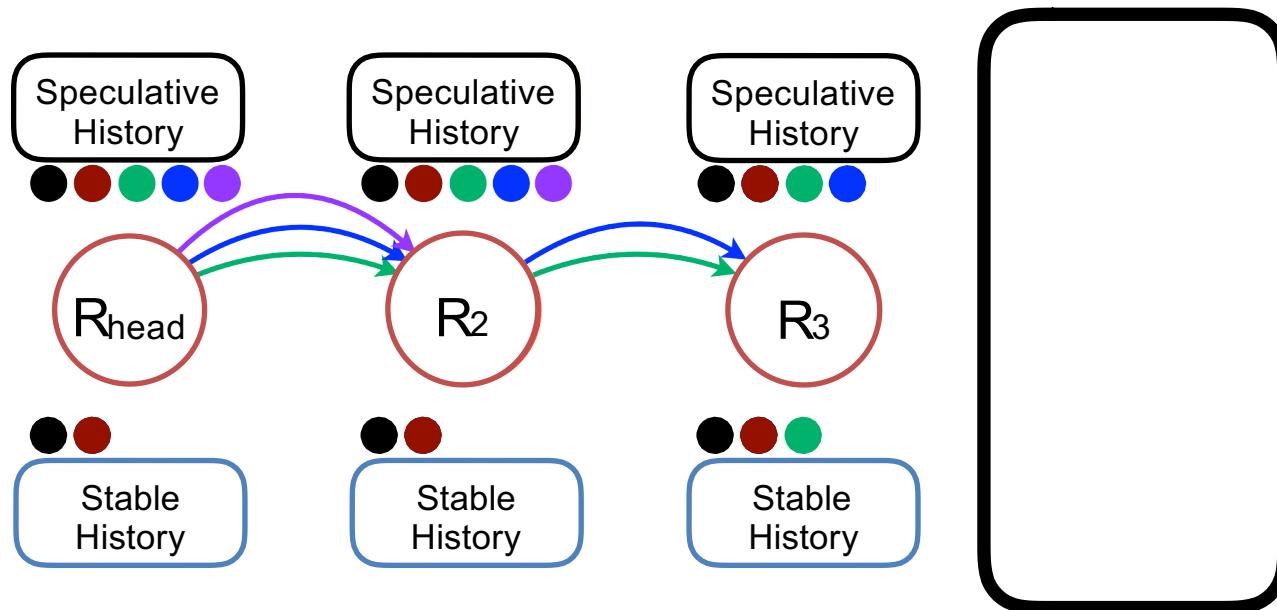
Middle Node Failures



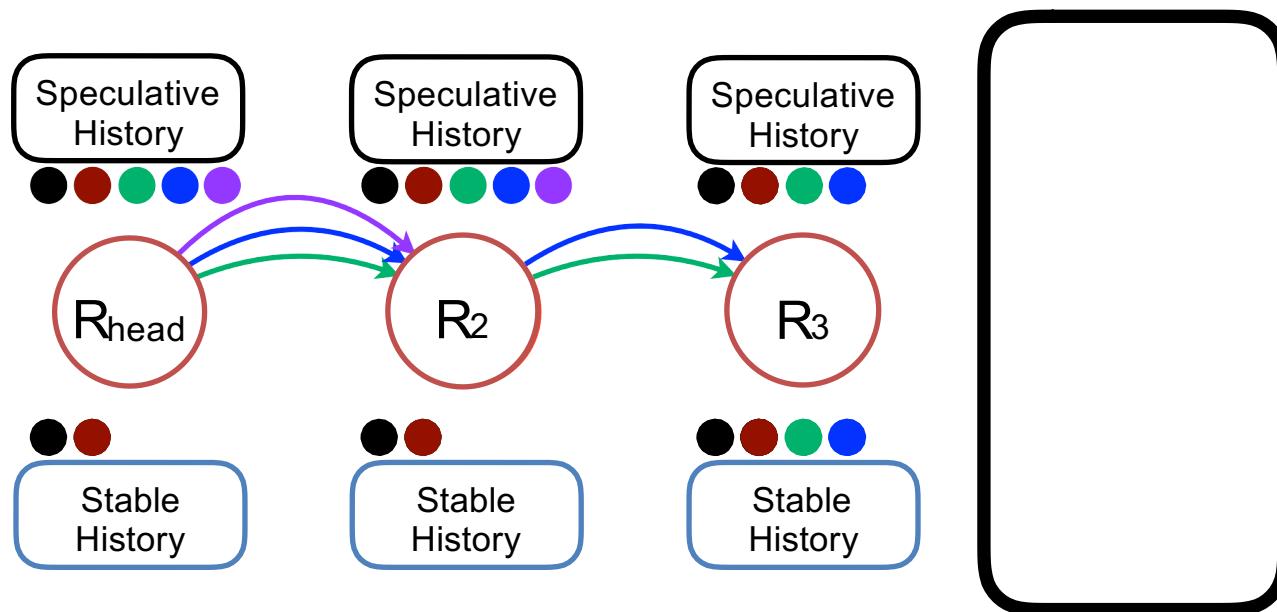
Tail Failures



Tail Failures

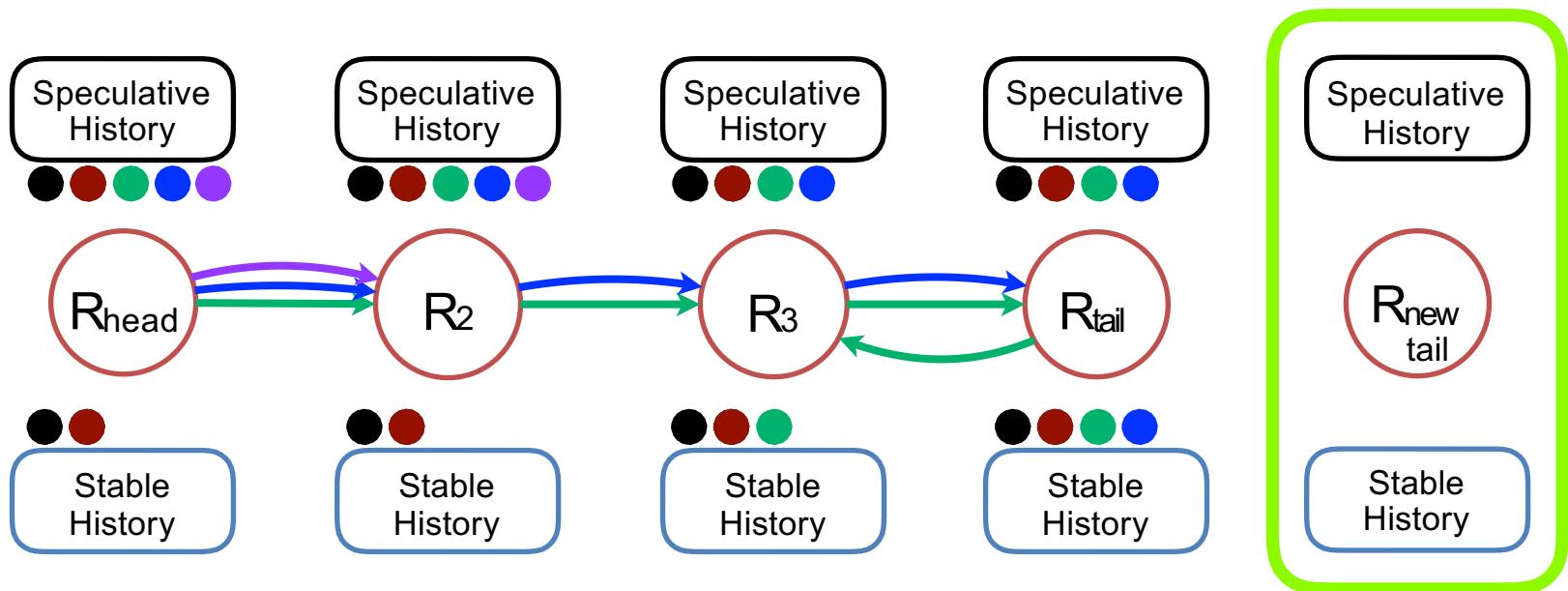


Tail Failures

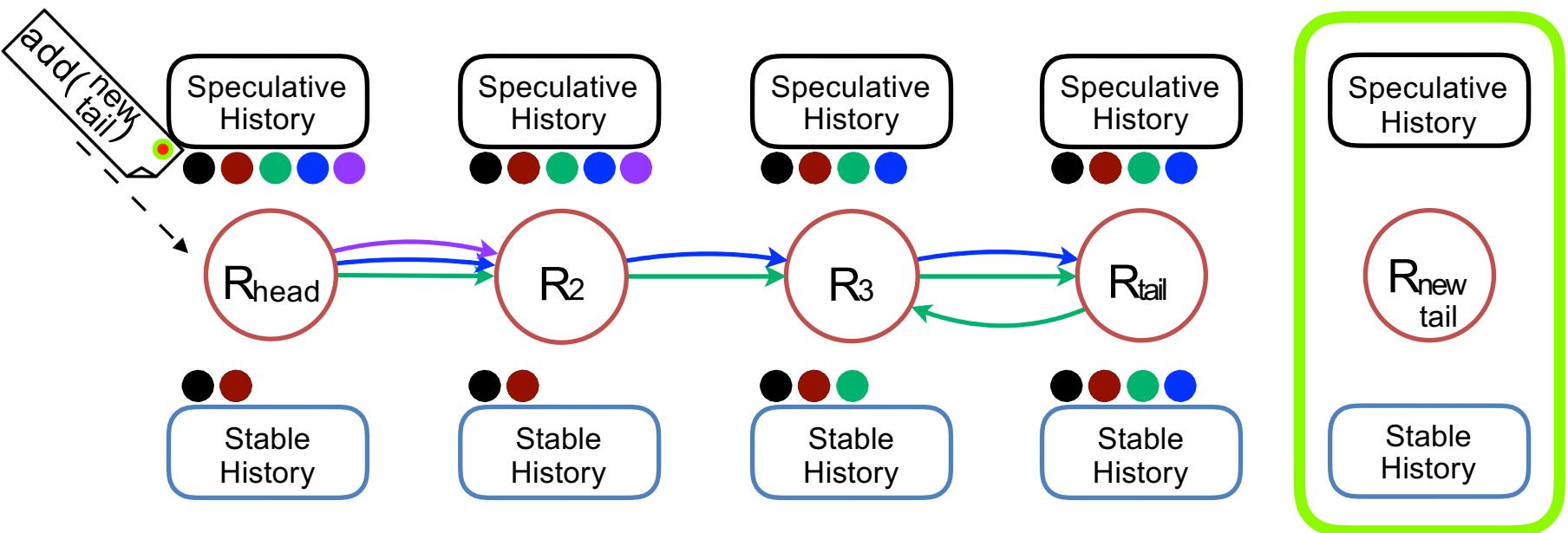


Reconfigurations

Adding a New Node

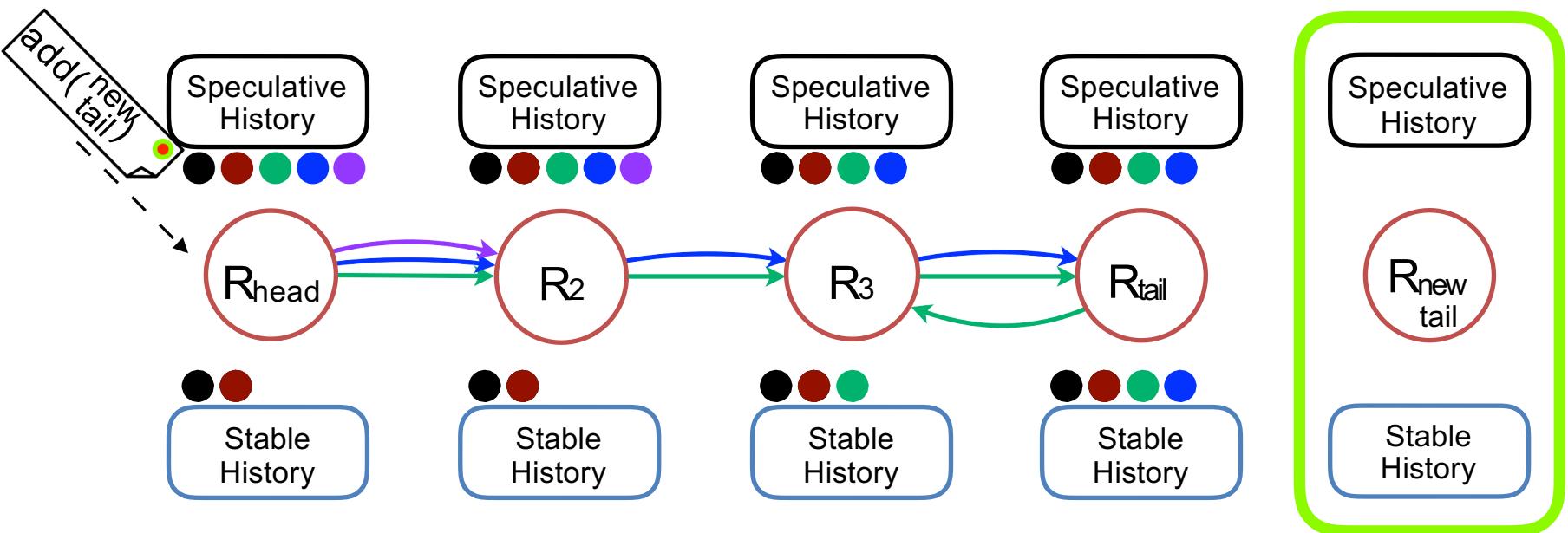


Adding a New Node



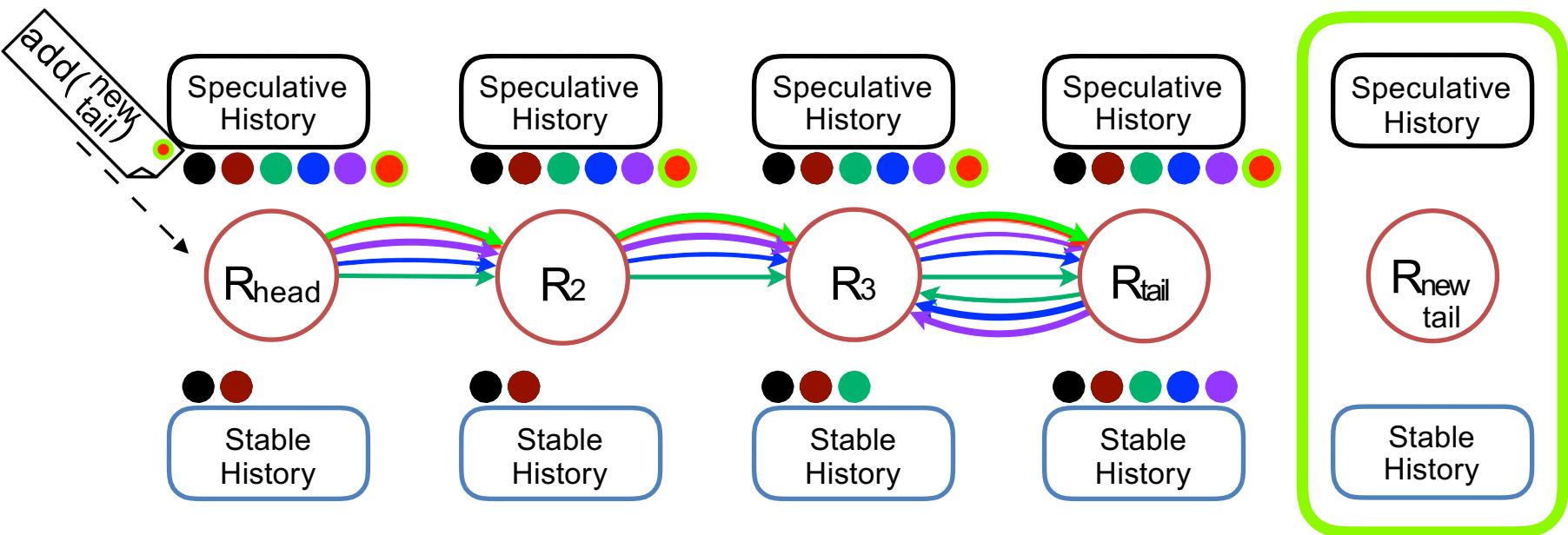
- new nodes are added to the chain with special configuration updates that are added to the history: **add(nodeid)**

Adding a New Node

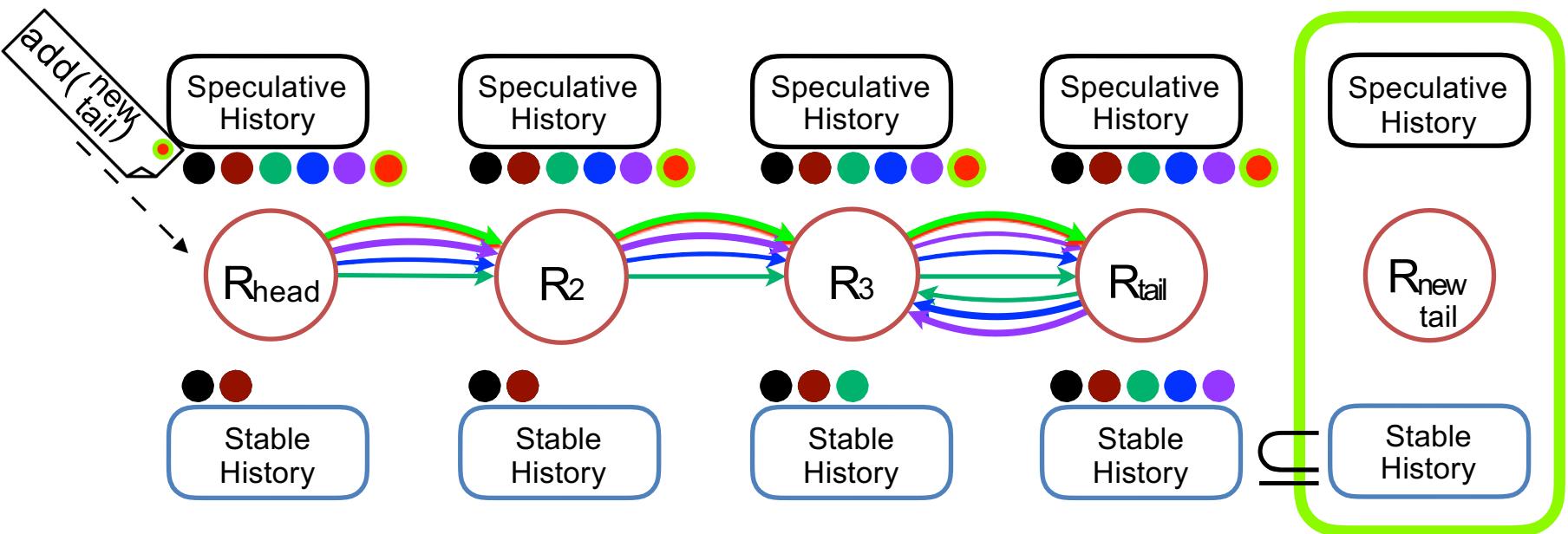


- new nodes are added to the chain with special configuration updates that are added to the history: **add(nodeid)**
- by looking at the order of these updates, a node can determine the configuration of the chain

Adding a New Node

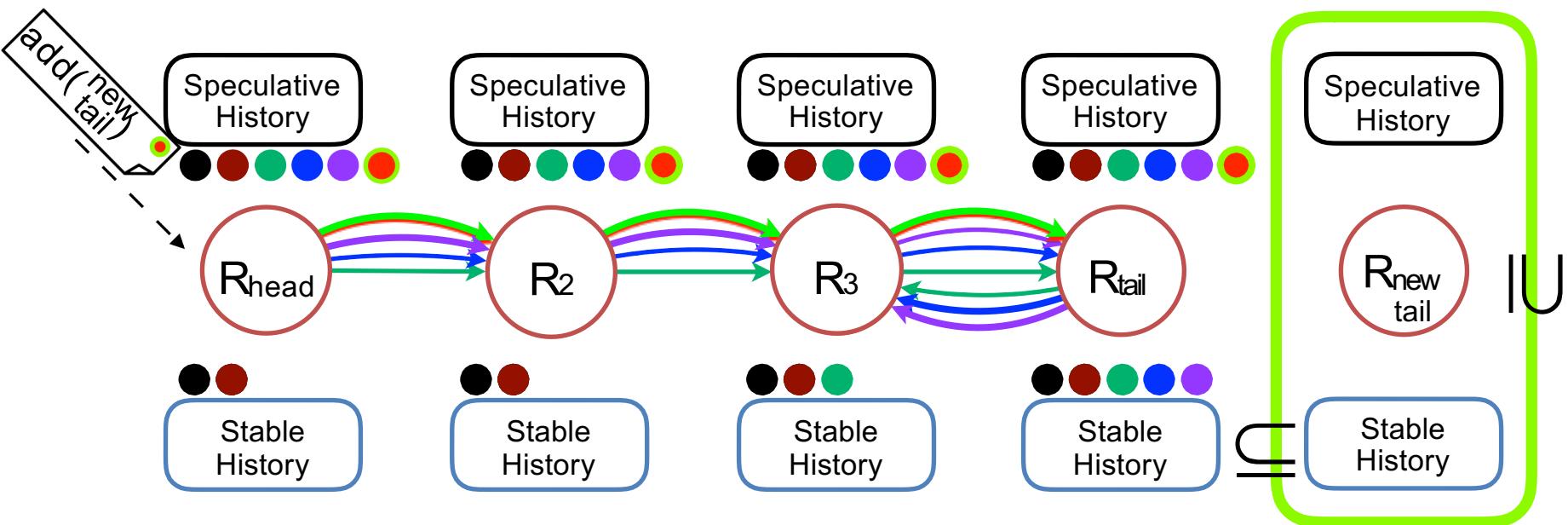


Adding a New Node



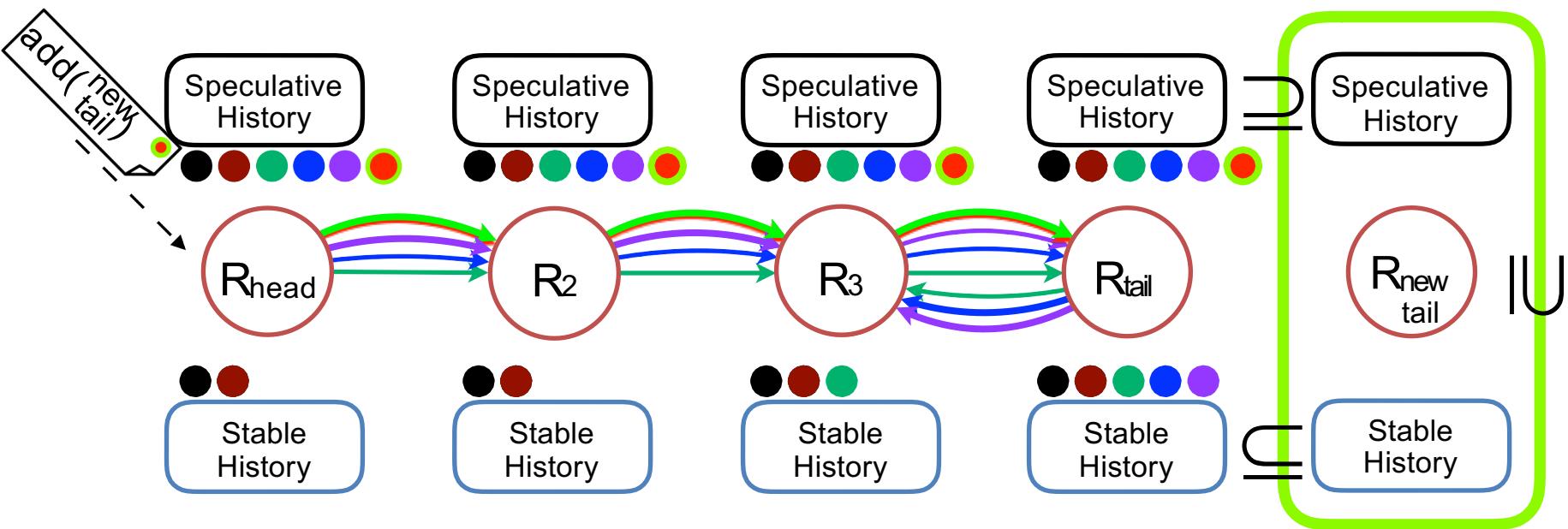
- stable history of new tail should be a superset of the stable history of tail.

Adding a New Node



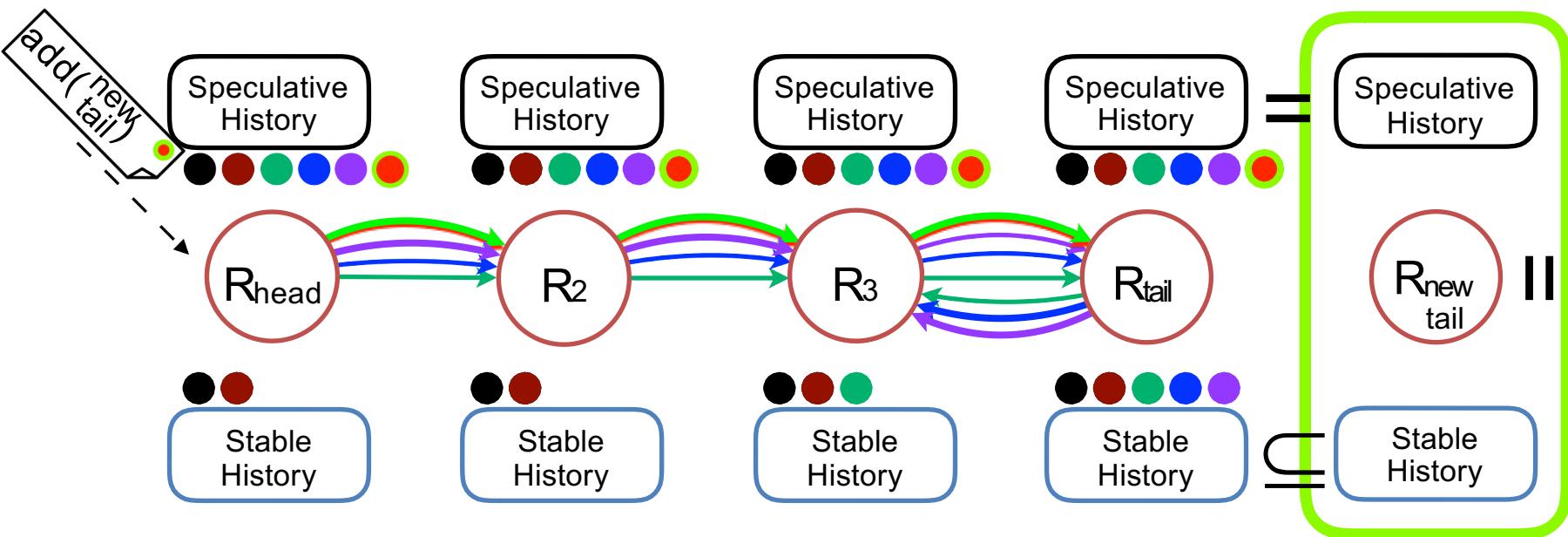
- stable history of new tail should be a superset of the stable history of tail.

Adding a New Node



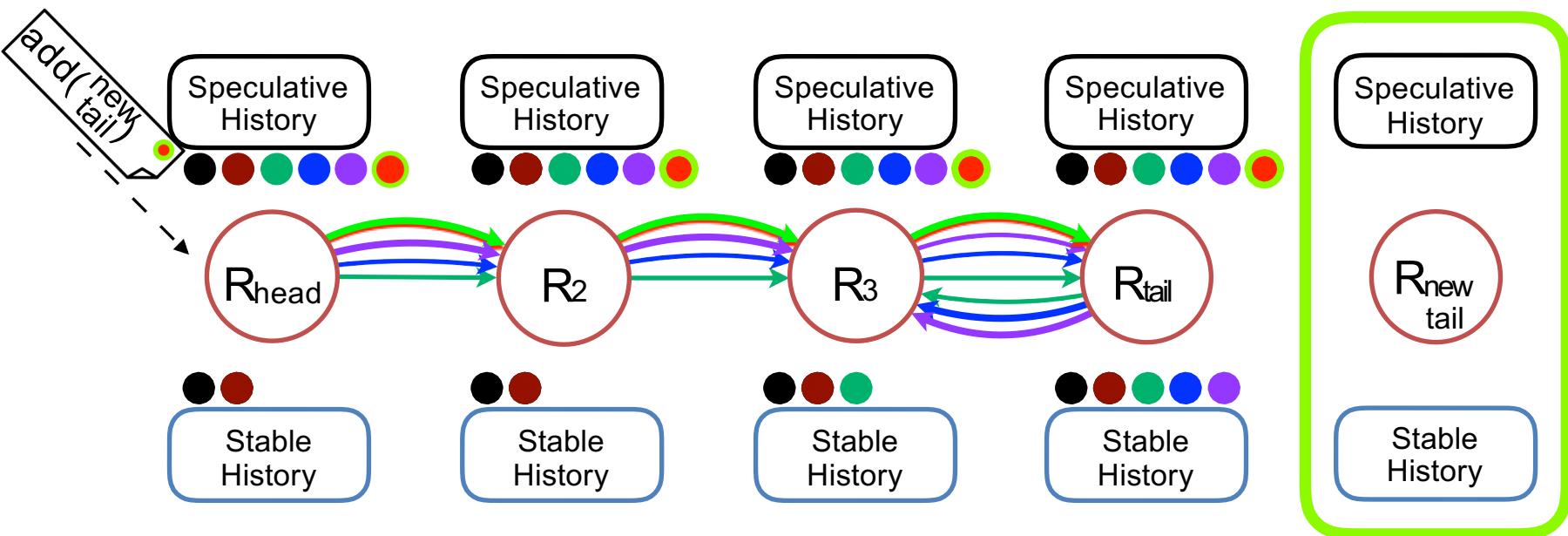
- stable history of new tail should be a superset of the stable history of tail.
- speculative history of new tail should be a superset of its stable history.

Adding a New Node



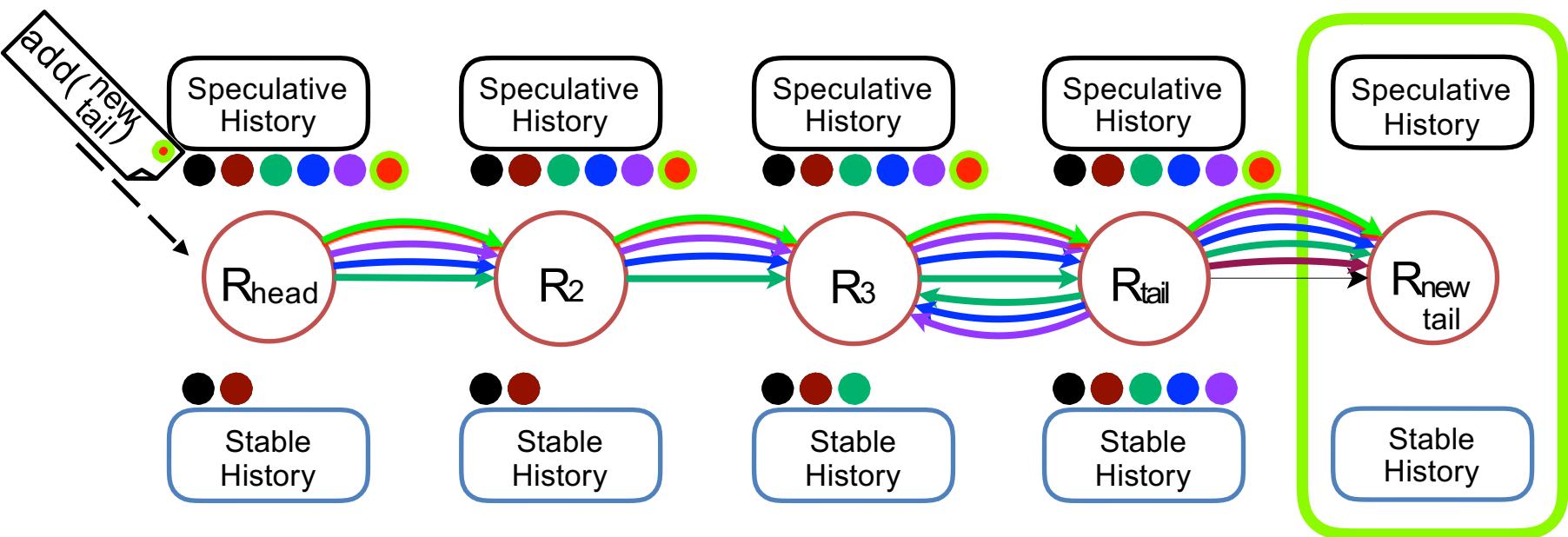
- stable history of new tail should be a superset of the stable history of tail.
- speculative history of new tail should be a superset of its stable history.
- speculative and stable histories of new tail should be equal to the speculative history of tail

Adding a New Node

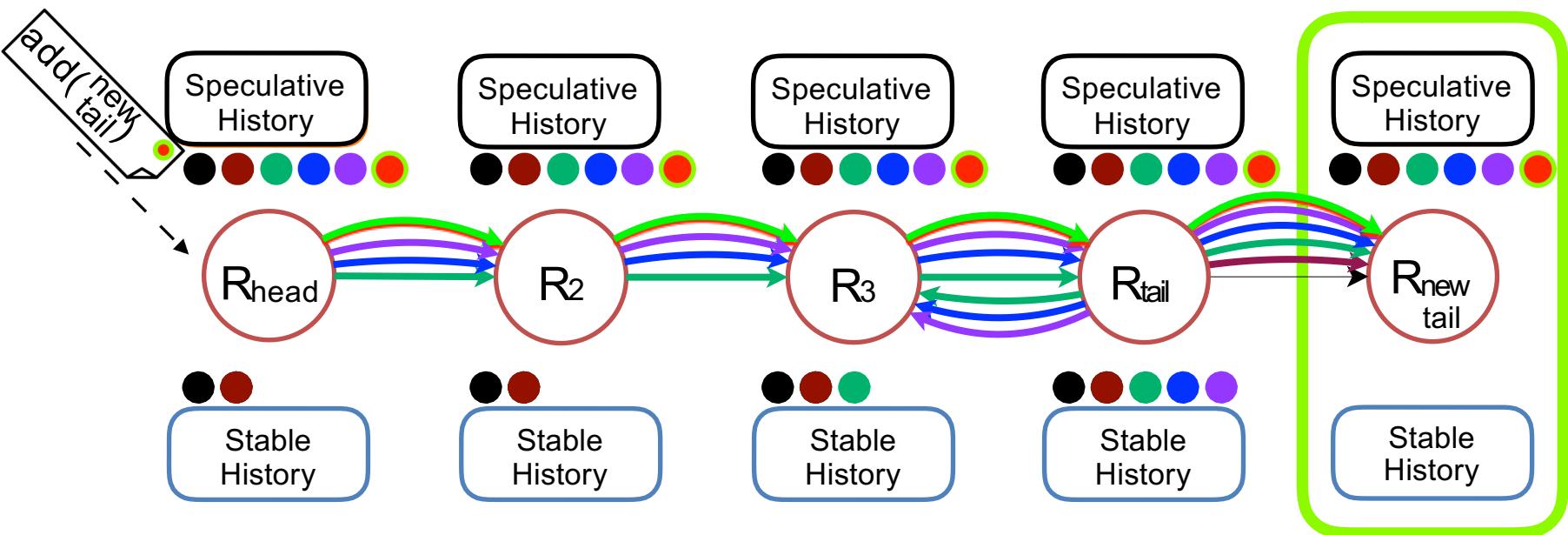


- stable history of new tail should be a superset of the stable history of tail.
- speculative history of new tail should be a superset of its stable history.
- speculative and stable histories of new tail should be equal to the speculative history of tail
- old tail should not answer to queries when the new tail should.

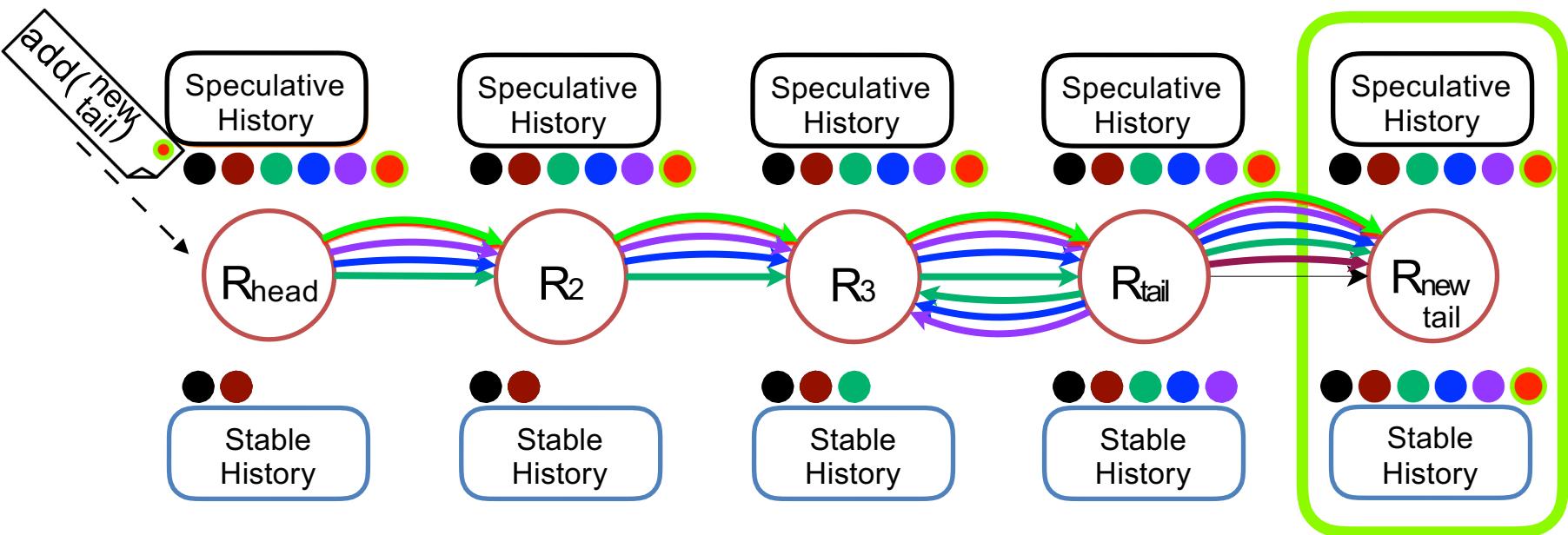
Adding a New Node



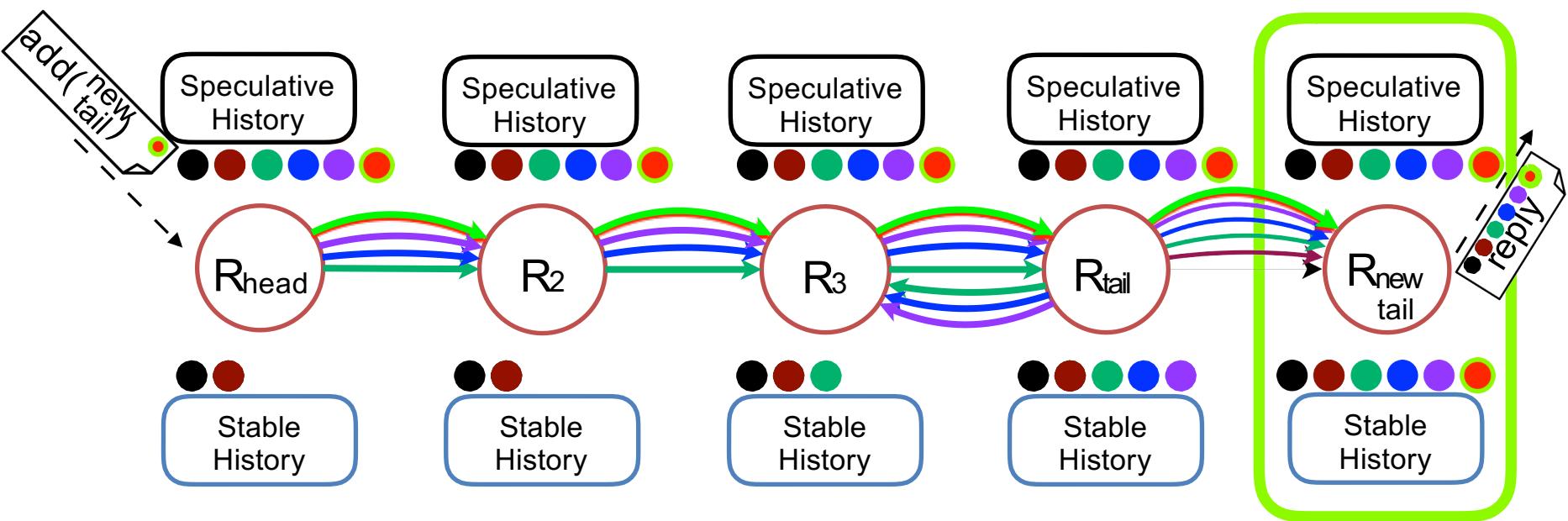
Adding a New Node



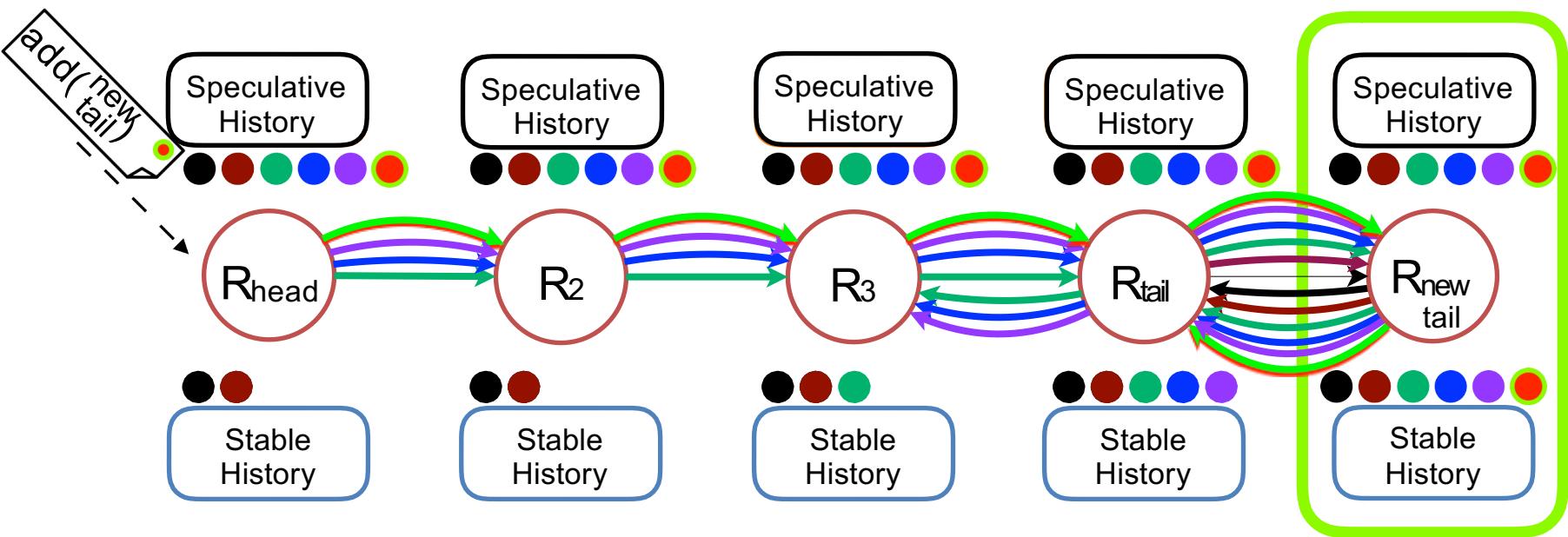
Adding a New Node



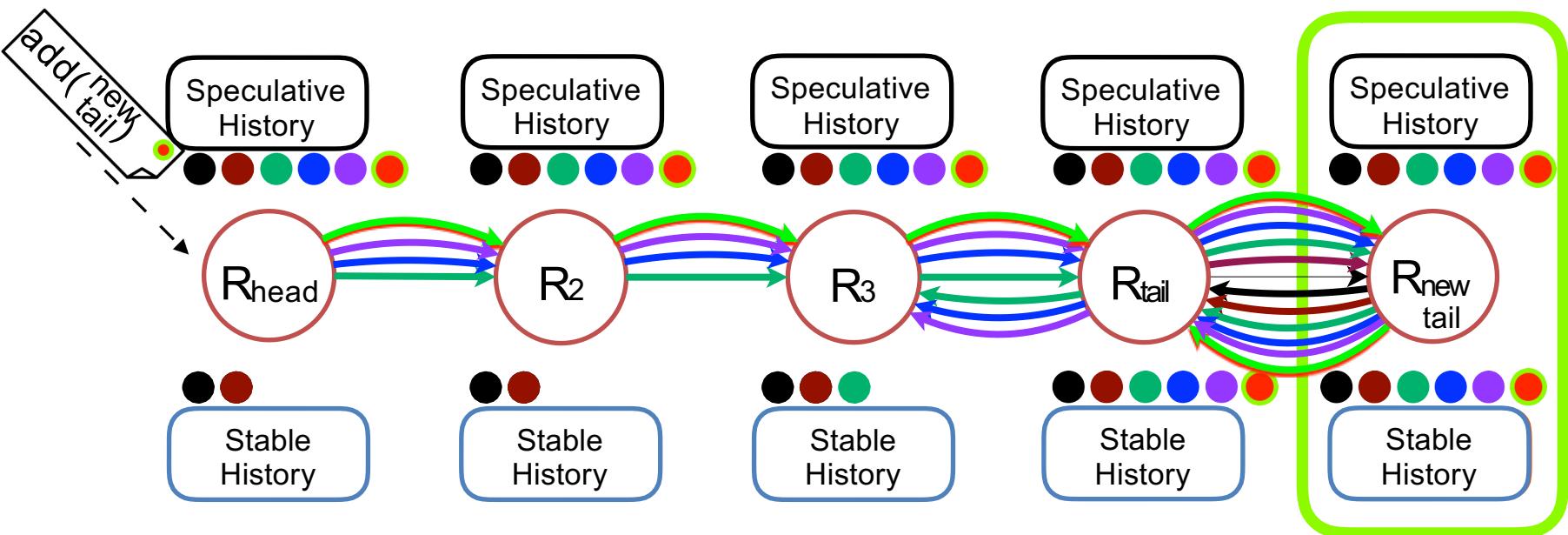
Adding a New Node



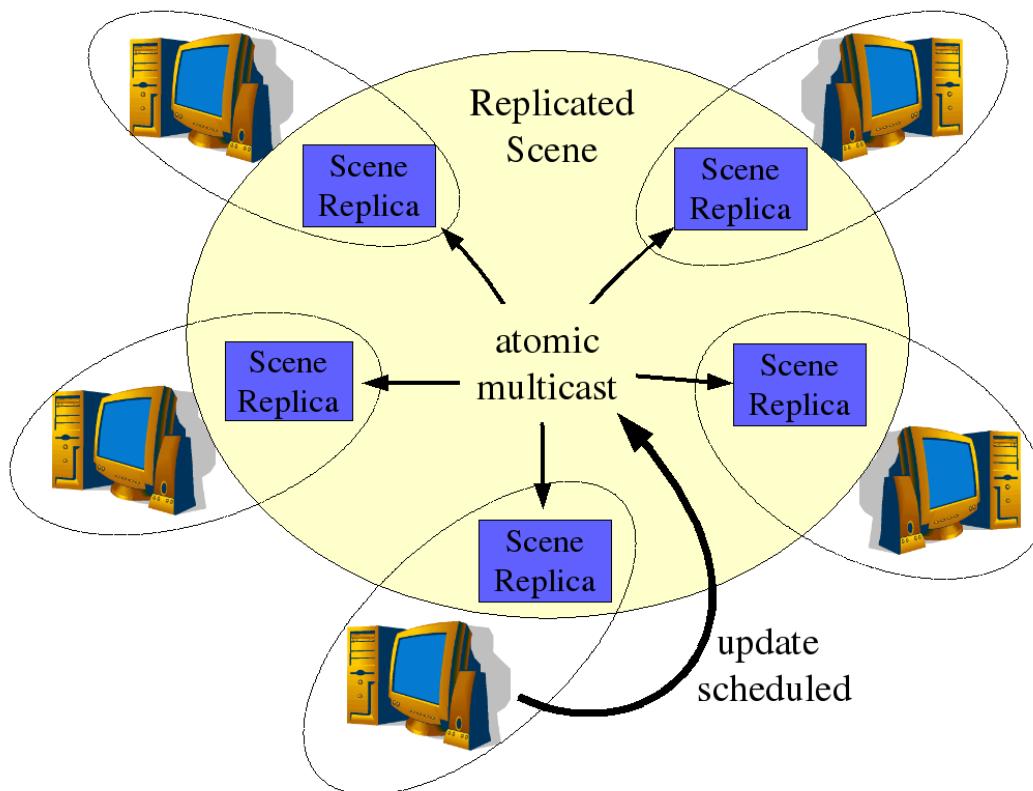
Adding a New Node



Adding a New Node



The Paxos Consensus Algorithm



Algorithm name	Type	DS and Failure Models
Centralized strategy	Mutual Exclusion	Asynchronous, only inactive nodes can crash , <u>reliable messages</u>
Token-based ring algorithm	Mutual Exclusion	Asynchronous, no nodes failures , <u>reliable messages</u>
Ricart & Agrawala	Mutual Exclusion	Asynchronous, no nodes failures , <u>reliable messages</u>
Changs & Roberts (Ring-based)	Leader Election	Asynchronous, no failures during election , <u>reliable messages</u>
Bully algorithm	Leader Election	Synchronous, process failures, <u>reliable messages</u>
Dolev-Strong algorithm	Consensus (a.k.a. agreement)	Synchronous, f failures with $f+1$ rounds, <u>reliable messages</u>

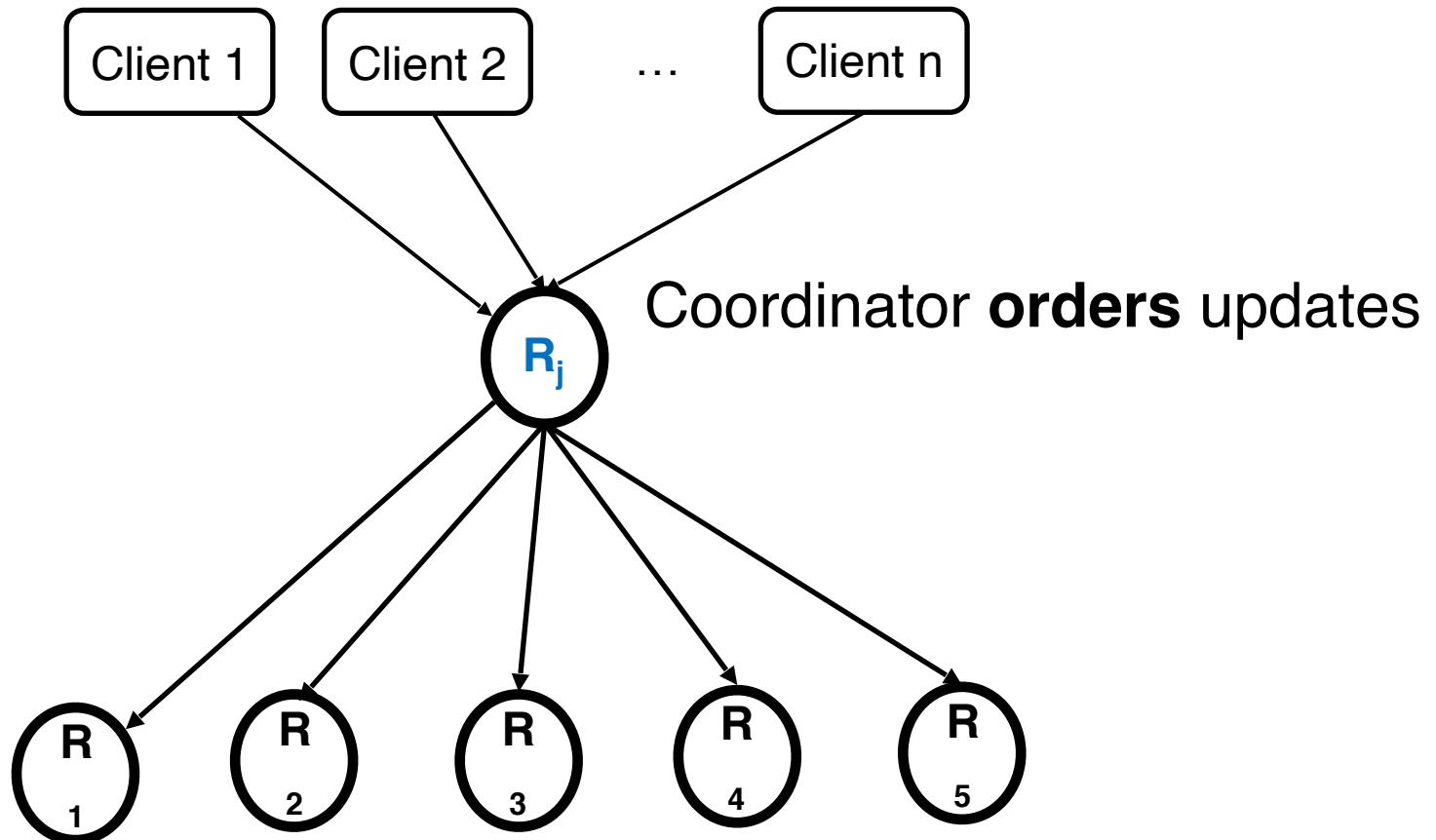
Consensus problems

- Desire **all nodes to agree** on a value after one or more nodes **proposed a value**
- Also known as **problems of agreement**
- **Challenges**
 - Reach consensus, even in the presence of **node failures** and **unreliable networking**

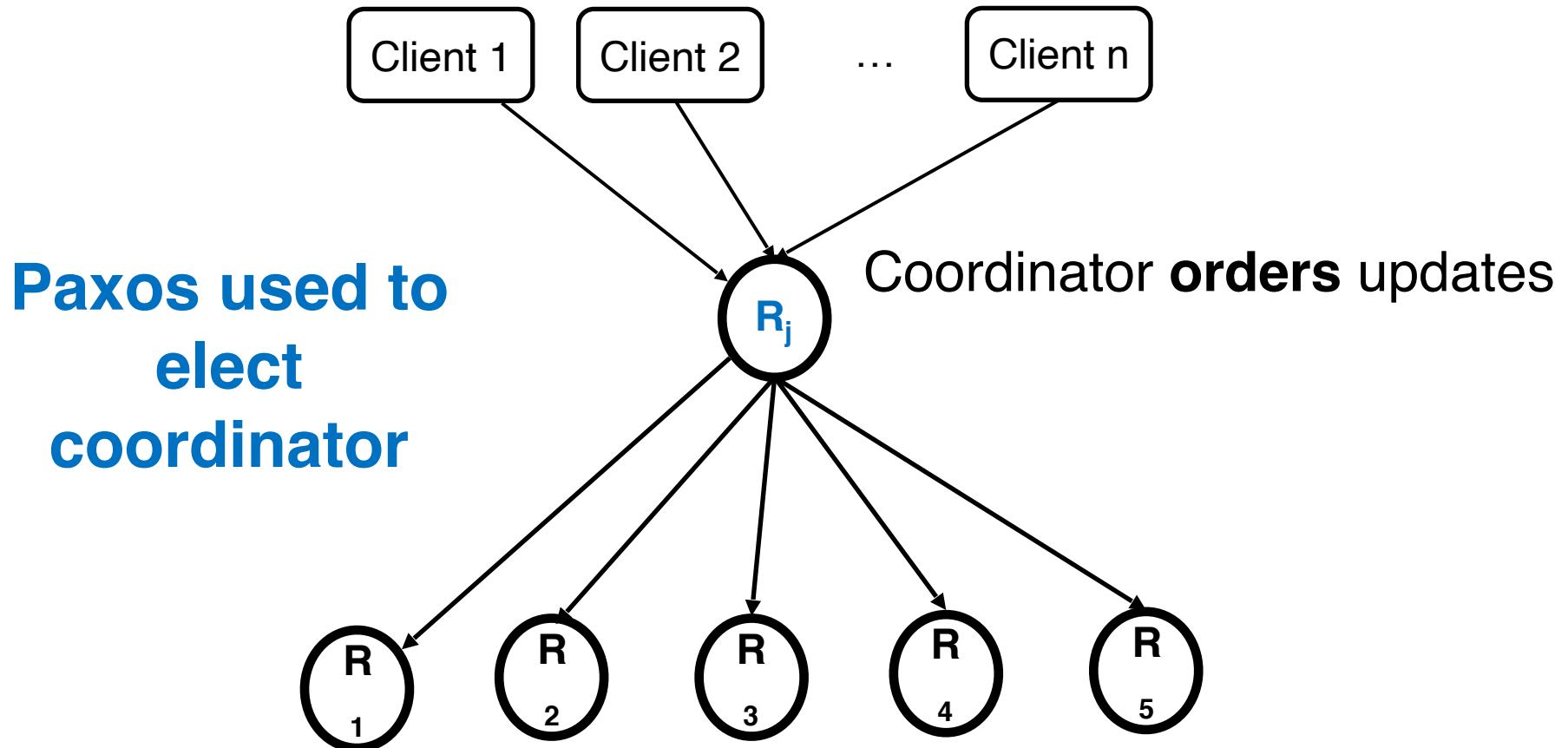
Consensus problem examples

- Two armies **agree** consistently to attack or to retreat (thought experiment)
- **Mutual exclusion:** Processes **agree** on who can enter CS
- **Leader election:** Processes **agree** on who is elected
- Replicated state machines **agree** on commands to execute
- Multi-primary replication **agree** on order of updates at replicas
- Totally ordered multicast: Processes **agree** on the order of messages delivered
- ATM and bank's servers **agree** what should happen to bank account balance when withdrawing money
- Transaction managers **agree** to commit or abort (e.g., two-phase commit)
- Flight computers **agree** to “abort” (reboot) or “proceed”
- Reactor safety systems **agree** on position of control rods

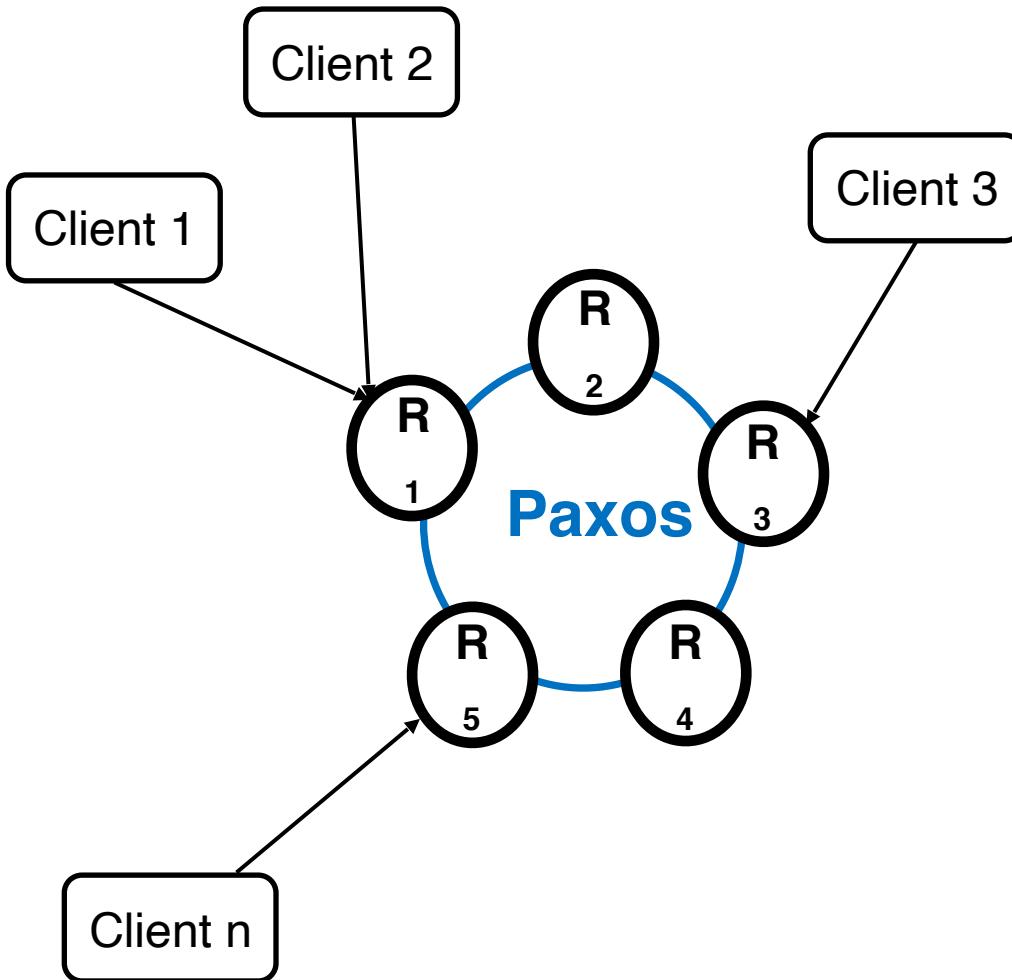
Replication



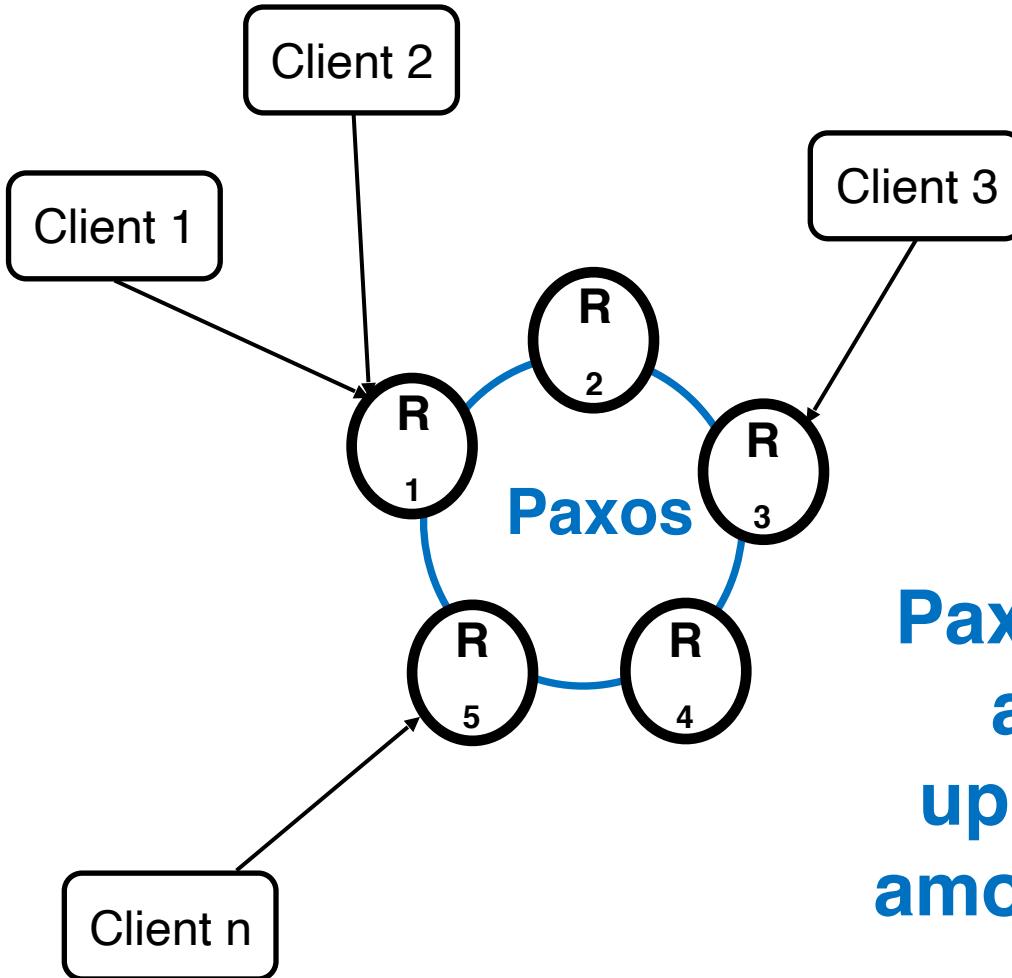
Replication



Replication



Replication



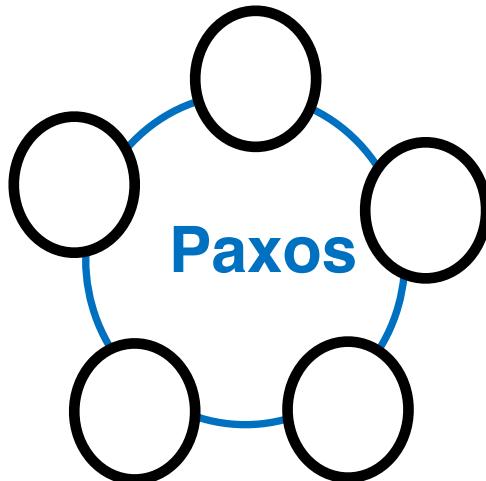
**Paxos used to
agree on
update order
among replicas**

Coordination services

(Paxos inside)

Highly-available and persistent, a.k.a.
distributed lock service

Chubby / ZooKeeper*

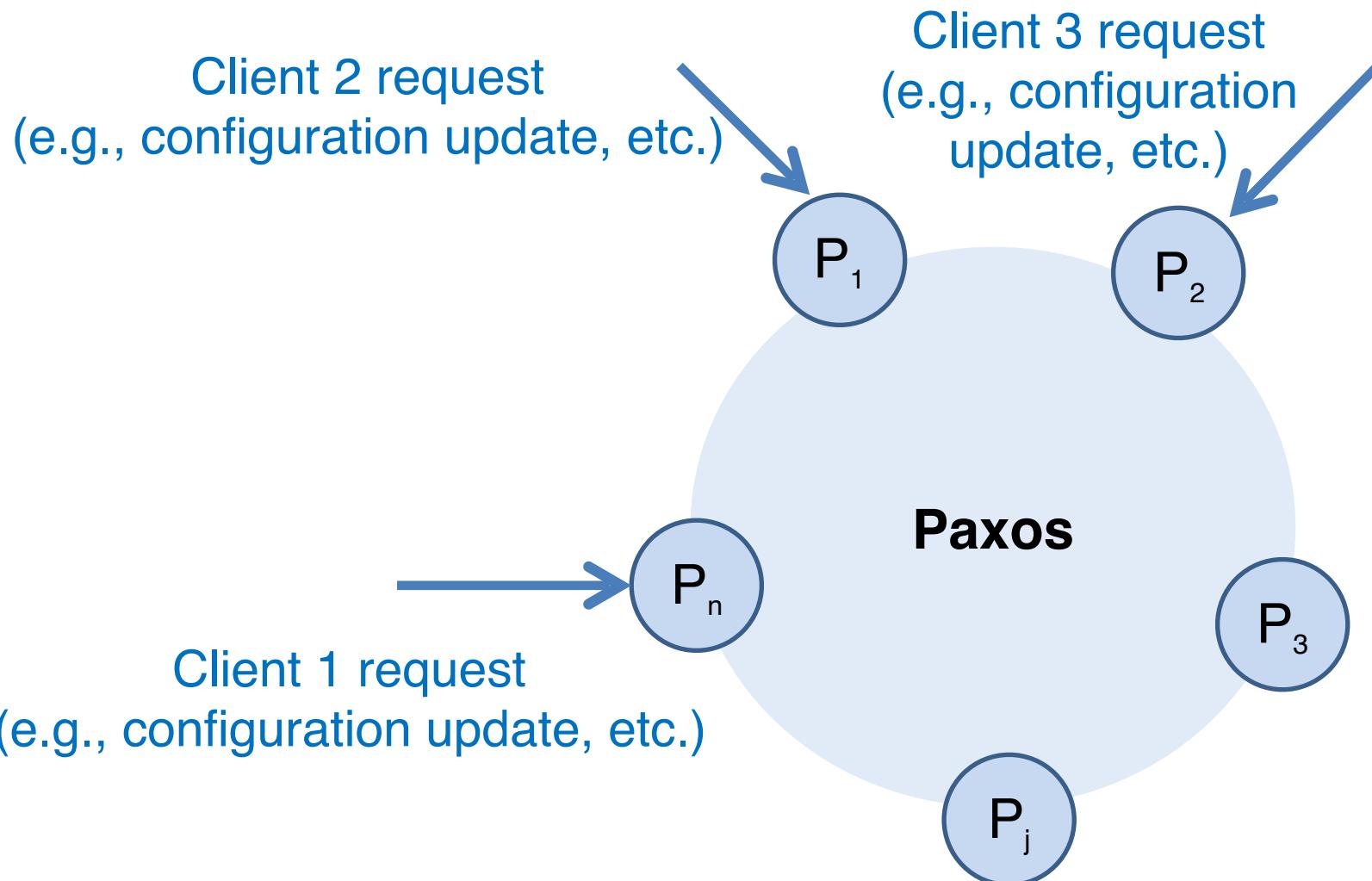


Use in Bigtable/HBase

- Ensure **at most one active** Bigtable **master** at any time
- Store **bootstrap** location of **data** (root tablet)
- **Discover** tablet servers (manage their lifetime)
- Store **configuration** **information**

*ZooKeeper uses ZAB (ZK. Atomic Broadcast, not Paxos)

As coordination service



Consensus problem

- One or more nodes may propose some value. ***How do we get a collection of nodes to agree on exactly one of the proposed values?***
- Requirements for consensus:
 - **Agreement:** Every correct node (not failing) must agree on the same value.
 - **Validity:** Only proposed values can be decided. If a node decides on some value, V , then some node must have proposed V .
 - **Integrity:** Each node can decide a value at most once.
 - **Termination:** All correct nodes eventually decide on a value.



Basic Paxos

"Then we are agreed nine to one that we will say our previous vote was unanimous!"

Brief history of Paxos

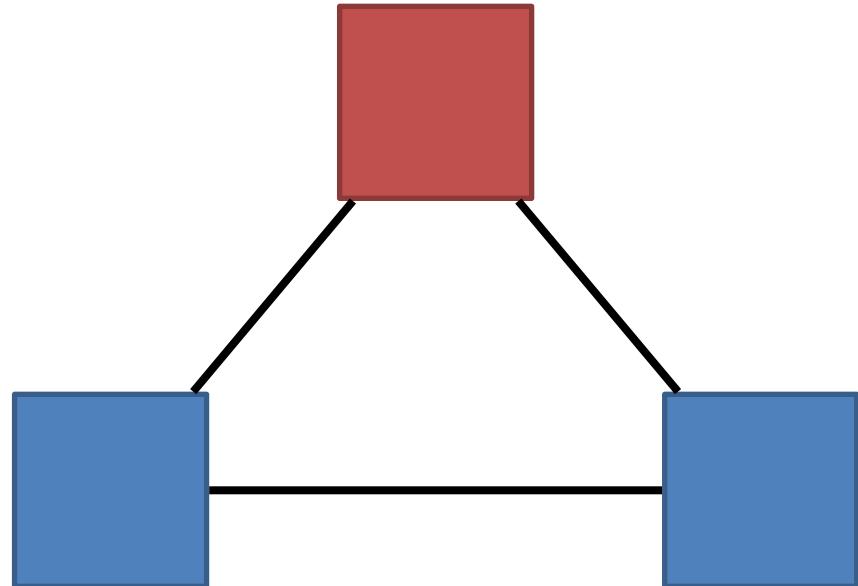
- Solves **consensus** in **asynchronous systems**, with **node** and **network omission failures***
 - Better than any algorithms we've seen so far!
 - But... (FLP, *cf.* next)*
- Original paper: “*The Part-Time Parliament*,” by Lamport (submitted in 1990, published in 1998?!)
- Fictional legislative **consensus system** for the Greek island of Paxos
- Rewritten into “plain English” as “*Paxos Made Simple*”
- Many variations of the algorithm now exist in the Paxos family (*cf.* “Paxos family” slide)

*Paxos in light of FLP result

- In **asynchronous systems**, with even **one node crashing**, there is no guarantee to reach consensus.
- Paxos guarantees **safety**, but **not liveness**
- “*Conditions that could prevent progress are difficult to provoke.*”
- Paxos **attempts to make progress** even during periods when some bounded number of nodes are unresponsive

Assumptions for Basic Paxos

- Nodes
- Network
- Number of nodes

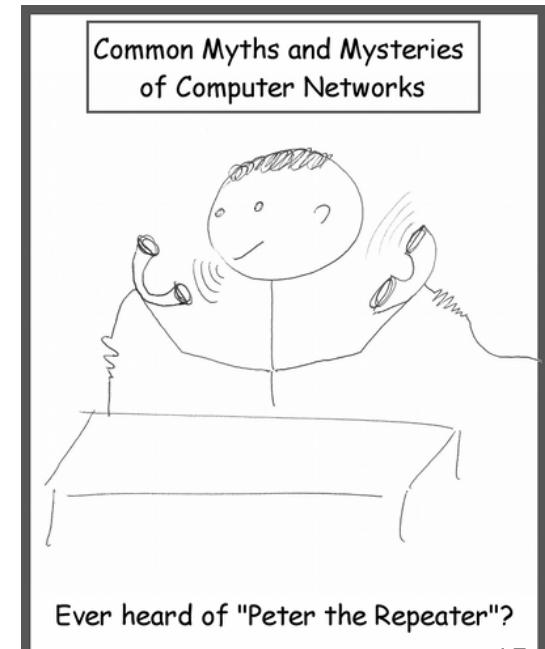


Nodes

- Operate at **arbitrary speed** (i.e., asynchronous model)
- May experience **failures** (i.e., crash failures)
- Nodes **may** re-join protocol after failures (i.e., following a **crash-recovery failure model**; remember their previous state)
- Do not collude, lie, or otherwise attempt to subvert the protocol (i.e., **no Byzantine failures**)

Network

- Nodes can send messages to any other node via unicast
- Messages are **sent asynchronously**
(i.e., may take arbitrarily long)
- Messages may be **lost, reordered, or duplicated**
- Network may **partition**
- Messages are **delivered without corruption**
(i.e., **no Byzantine failures**,
cf. *Byzantine Paxos*)



Number of nodes

- In general, Paxos is functional given **$2f+1$** nodes (i.e., majority)
- Despite the **simultaneous failure** of any **f** nodes
- E.g., 5 nodes, **resilient against 2 failing** (e.g., 5 replicas in Chubby)



TOWARDS UNDERSTANDING PAXOS

Part 2

Roles in Paxos

Like client & server roles in client-server

Roles in Paxos

Like client & server roles in client-server

- Client triggers protocol
 - Express protocol in terms of **roles**
 - Proposer
 - Acceptor
 - Learner
 - Single process may play **one or more roles** at the same time
-
- Leader (one of P)

Roles in Paxos

Like client & server roles in client-server

- Client triggers protocol
- Express protocol in terms of **roles**
 - Proposer
 - Acceptor
 - Learner
 - Leader (one of P)
- **Single process** may play **one or more roles** at the same time
- Has no effect on protocol correctness

Roles in Paxos

Like client & server roles in client-server

- Client triggers protocol
- Express protocol in terms of **roles**
 - Proposer
 - Acceptor
 - Learner
 - Leader (one of P)
- **Single process** may play **one or more roles** at the same time
- Has no effect on protocol correctness
- **Roles are commonly coalesced** to improve latency, number of messages exchanged, etc.

Paxos informal, high-level overview

- A **proposer** starts a new proposal by sending a ***proposal*** (message) to **acceptors**
- **Acceptors** send a ***promise*** back if they can accept the proposal
- If a **proposer** collects a **majority** of ***promises***, it sets a value to the proposal and sends it to a majority of **acceptors**
- **Acceptors** send the decided value to all **learners** if they can accept the value
- **Learners** learn the decided value when they collect a **majority** of messages from **acceptors** for a given value

Paxos in context: Roles, proposals, value

Client 2 request
(e.g., update to process, etc.)

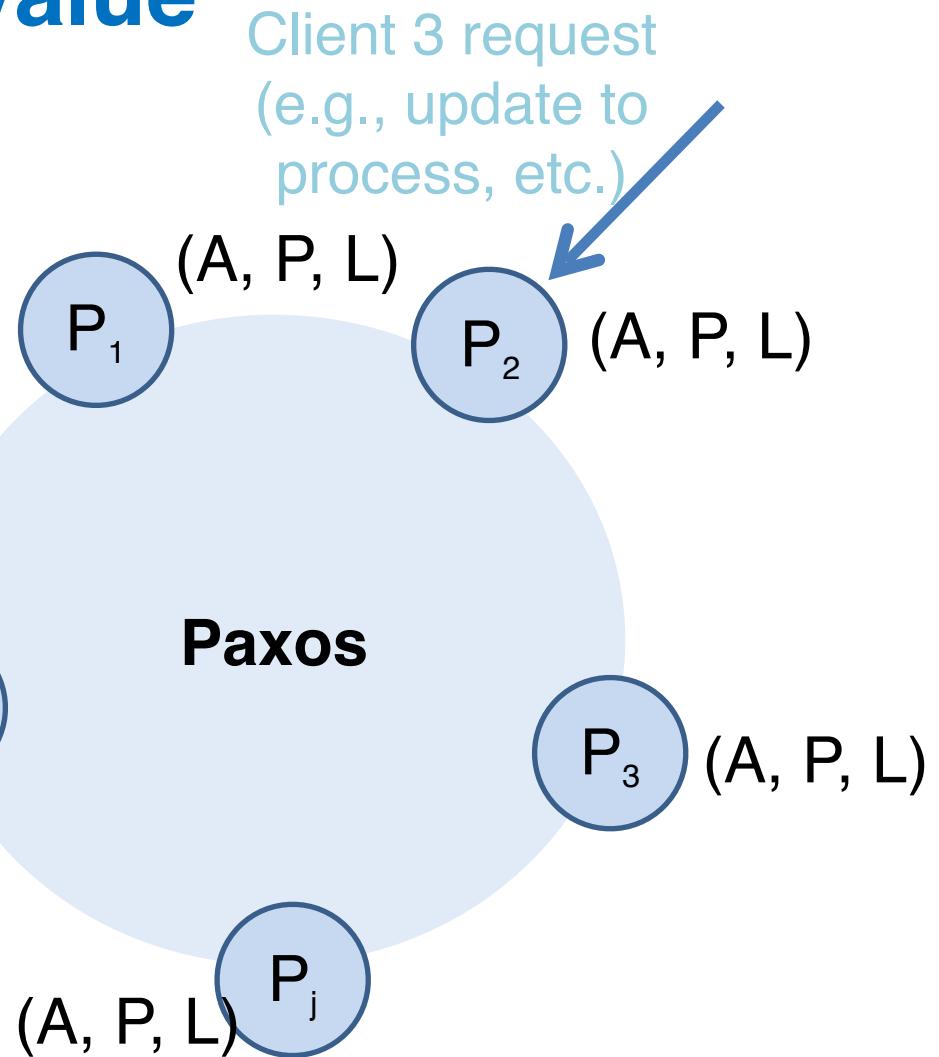
Client triggering protocol by submitting its “value” to a proposer.

Client 1 request
(e.g., update to process, etc.)

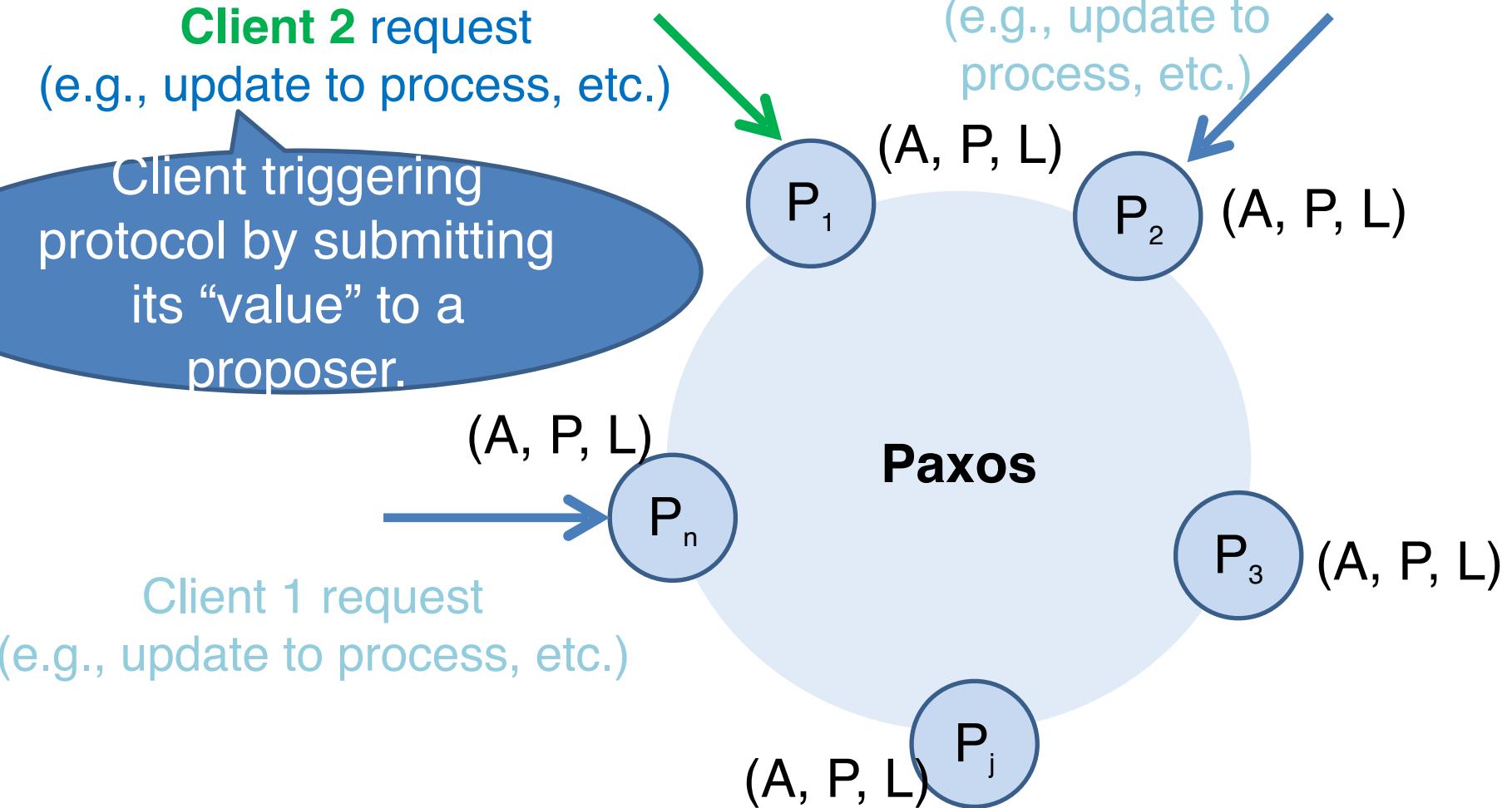
(A, P, L)

\longrightarrow

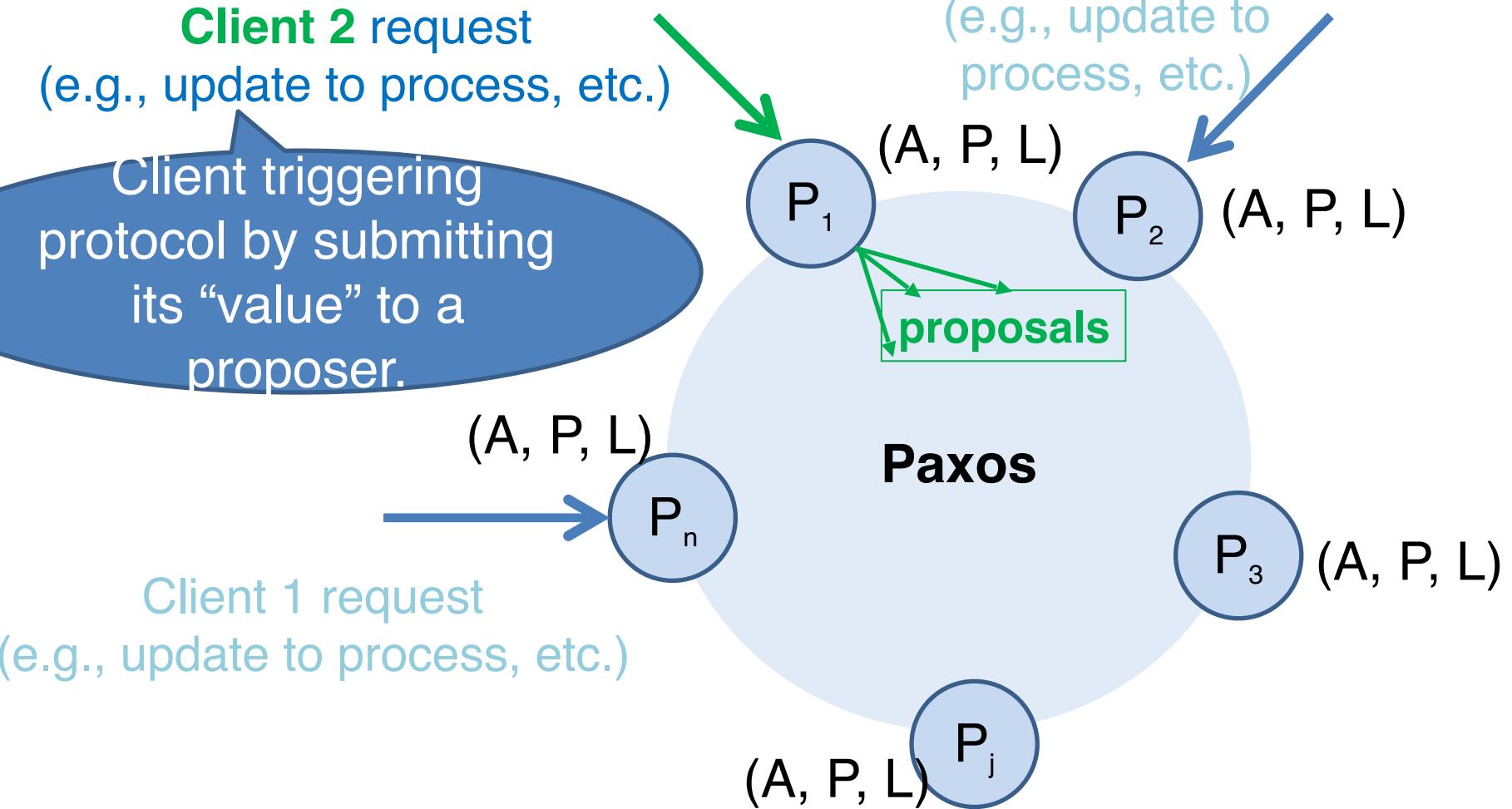
P_n



Paxos in context: Roles, proposals, value



Paxos in context: Roles, proposals, value

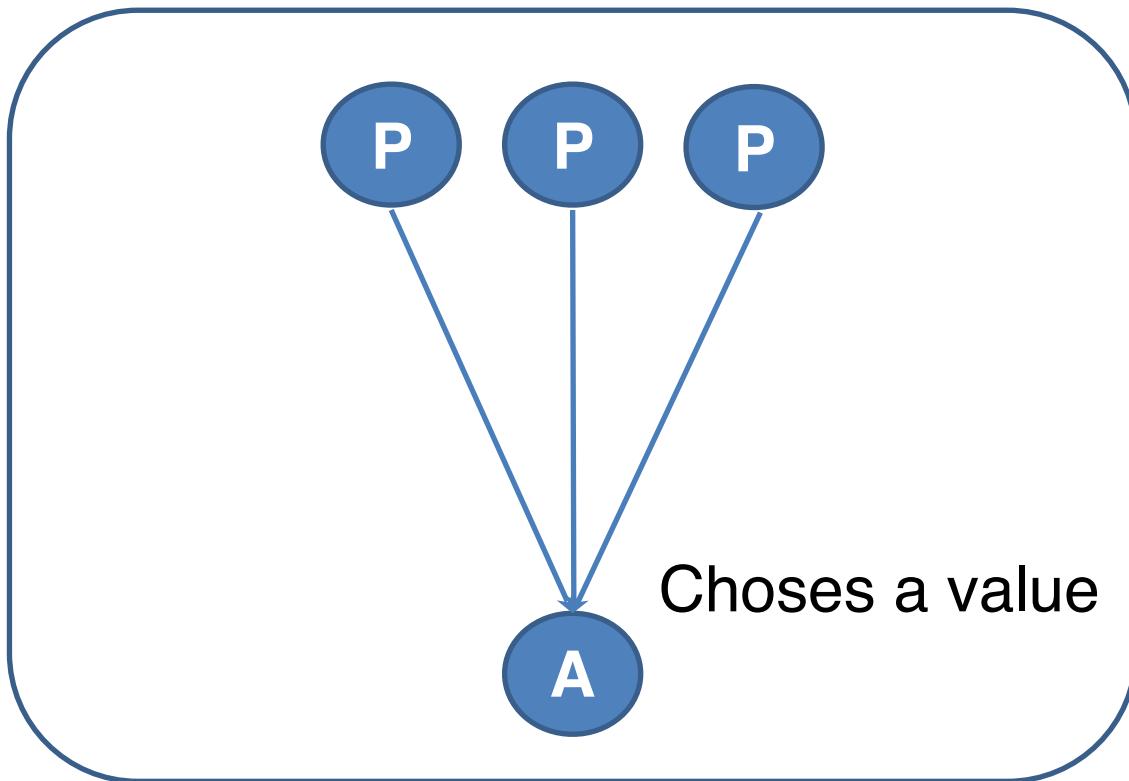


Corner cases addressed by Paxos

- *What if multiple proposers make competing proposals simultaneously?*
- *What if a proposer proposes a different value than an already decided value?*
- *What if there is a network partition?*
- *What if a proposer crashes in the middle of soliciting promises?*
- *What if a proposer crashes after collecting a majority of promises but before setting a value to its proposal?*
- ...

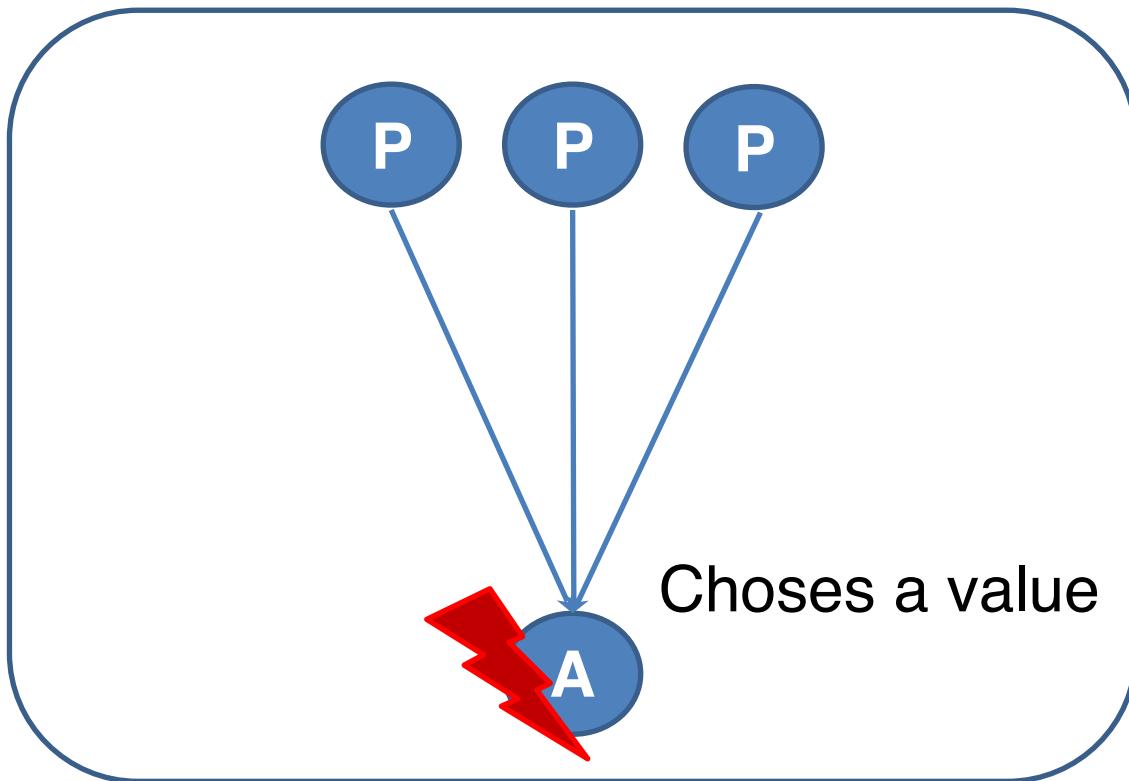
Strawman 1

(Multiple proposers, single acceptor)



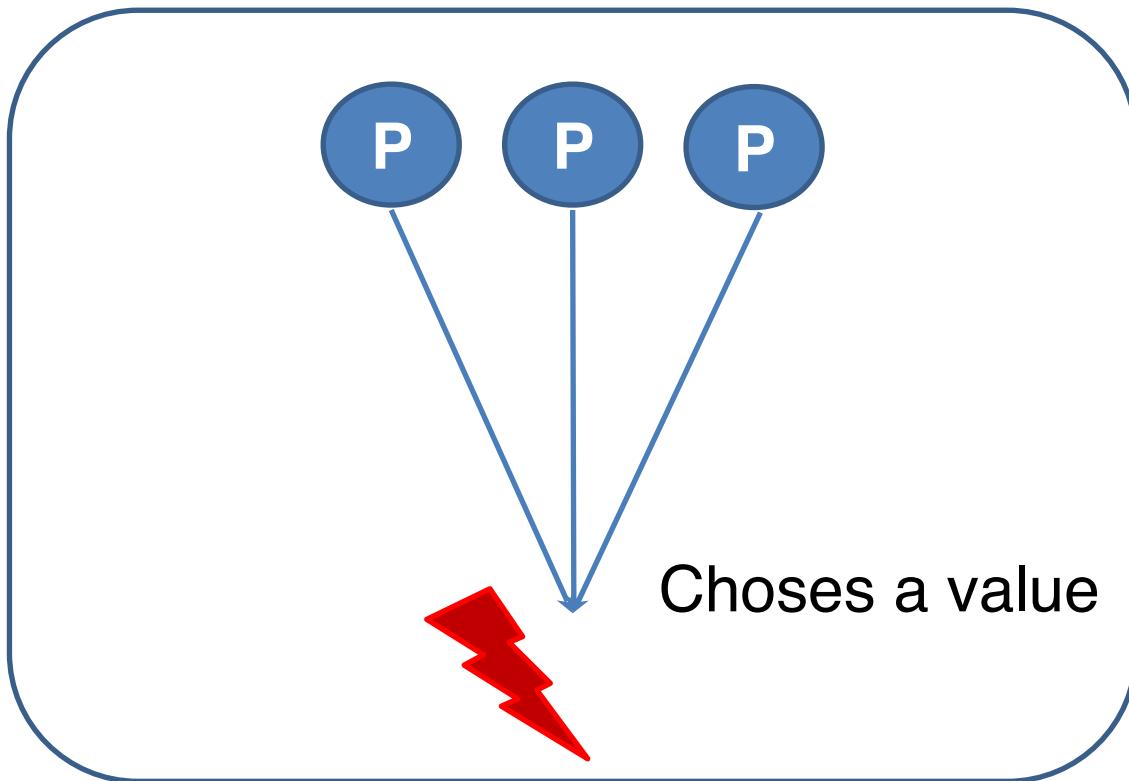
Strawman 1

(Multiple proposers, single acceptor)



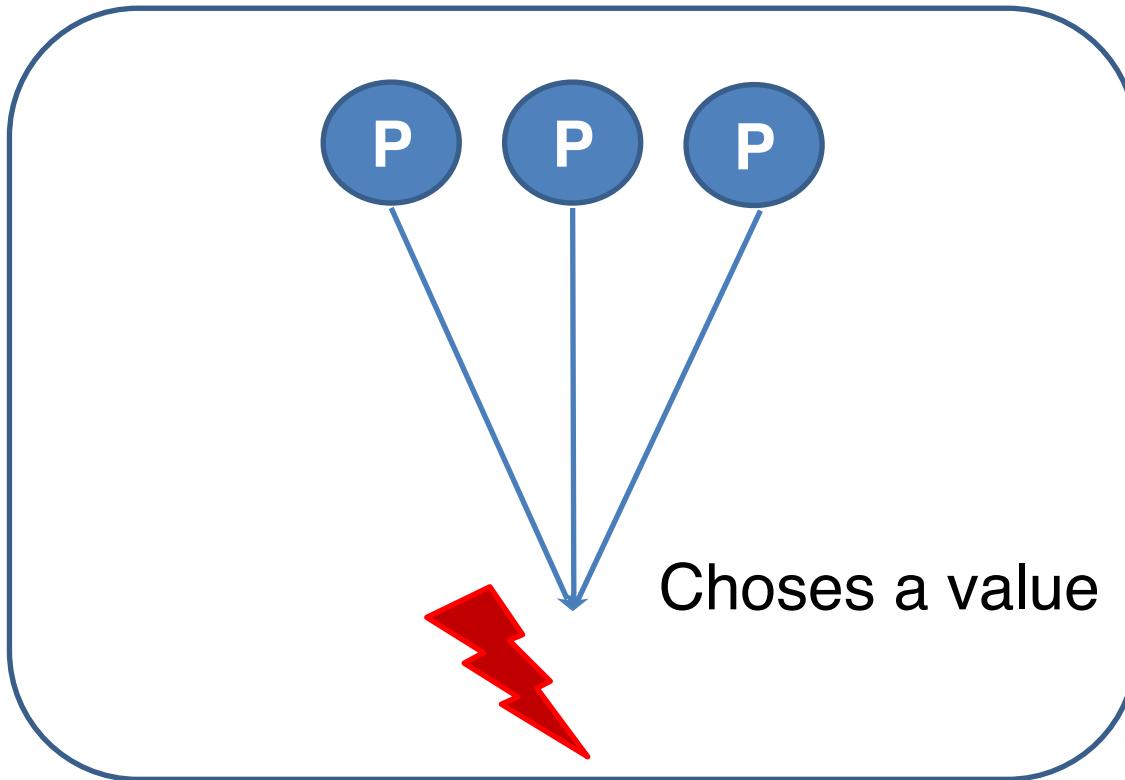
Strawman 1

(Multiple proposers, single acceptor)



Strawman 1

(Multiple proposers, single acceptor)

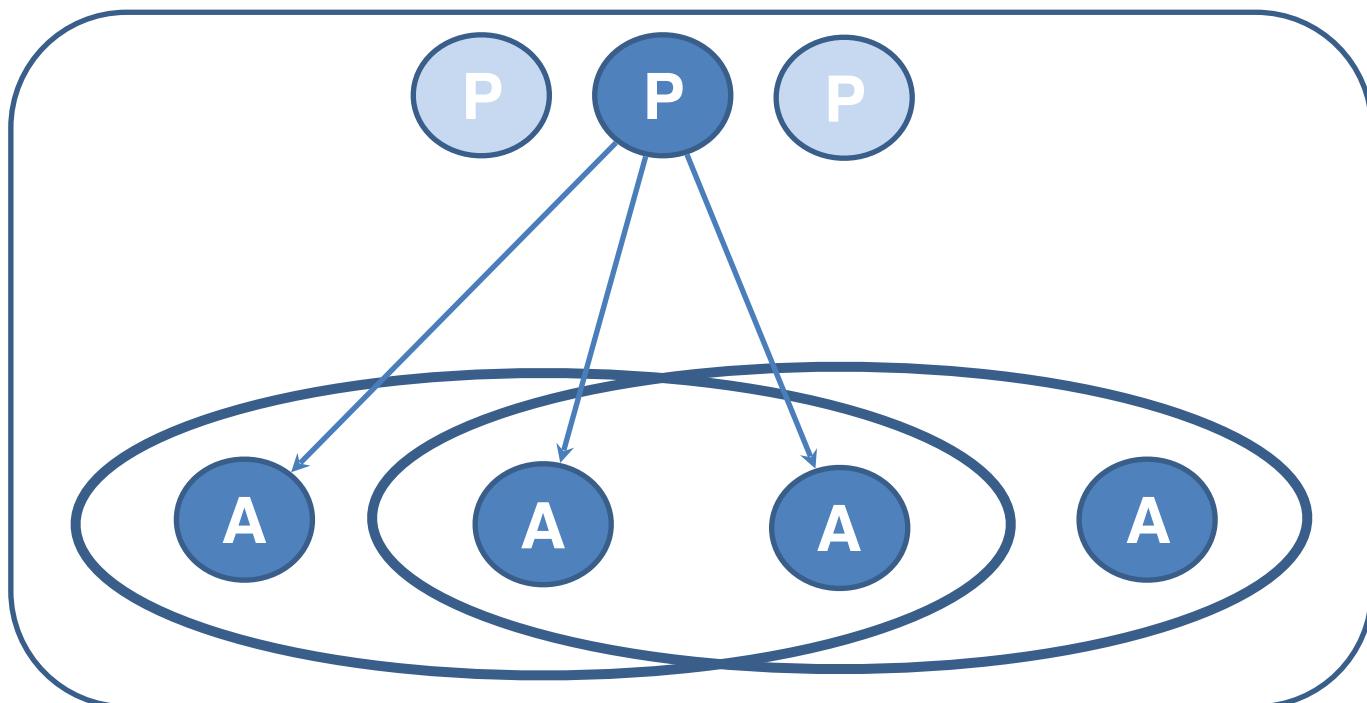


We need multiple acceptors to tolerate faults.

Strawman 2

(Multiple proposers, multiple acceptors)

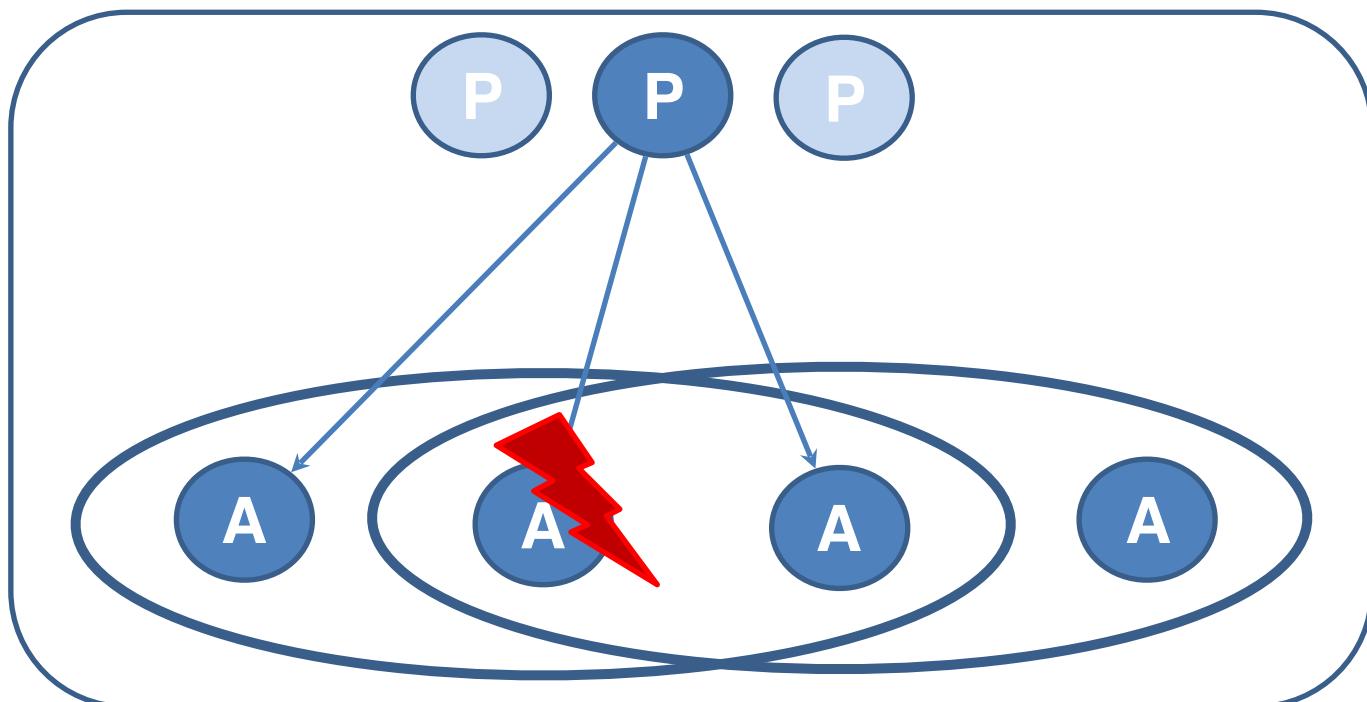
- Proposer sends proposal to a **majority of acceptors** or more
- If a majority of acceptors chooses a particular proposal (value) then that value is considered chosen



Strawman 2

(Multiple proposers, multiple acceptors)

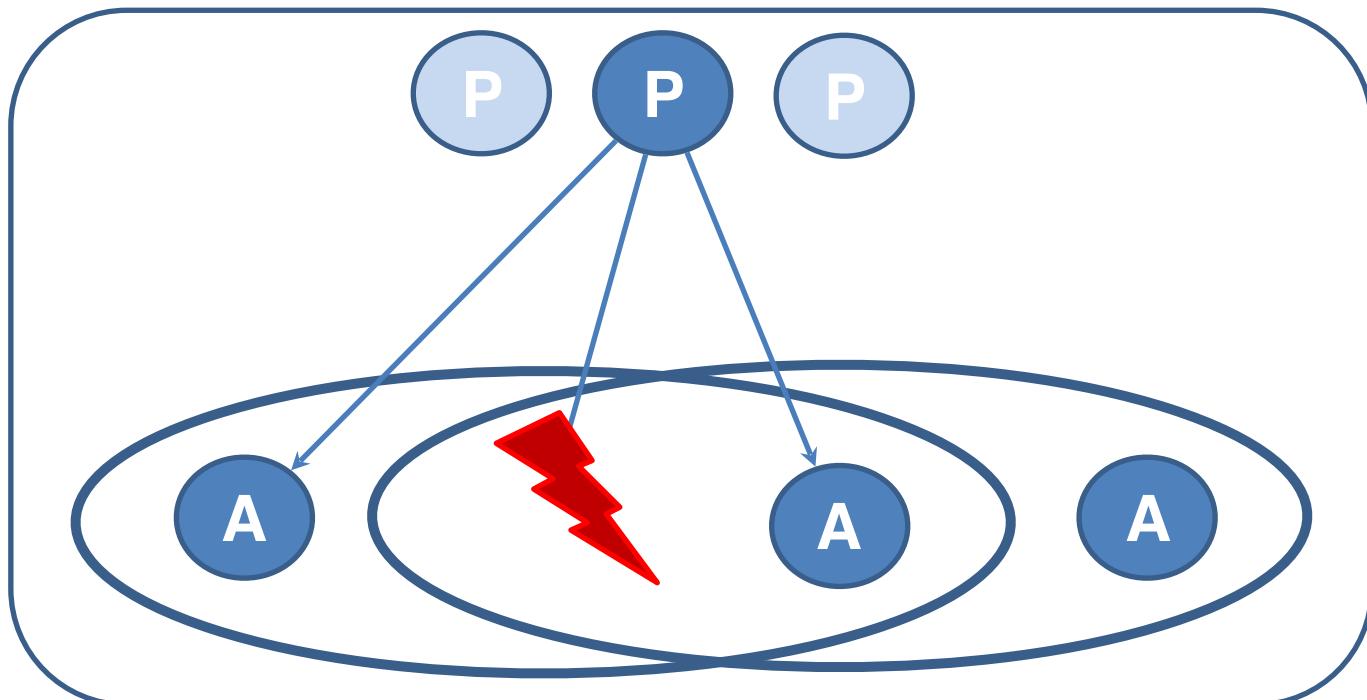
- Proposer sends proposal to a **majority of acceptors** or more
- If a majority of acceptors chooses a particular proposal (value) then that value is considered chosen



Strawman 2

(Multiple proposers, multiple acceptors)

- Proposer sends proposal to a **majority of acceptors** or more
- If a majority of acceptors chooses a particular proposal (value) then that value is considered chosen



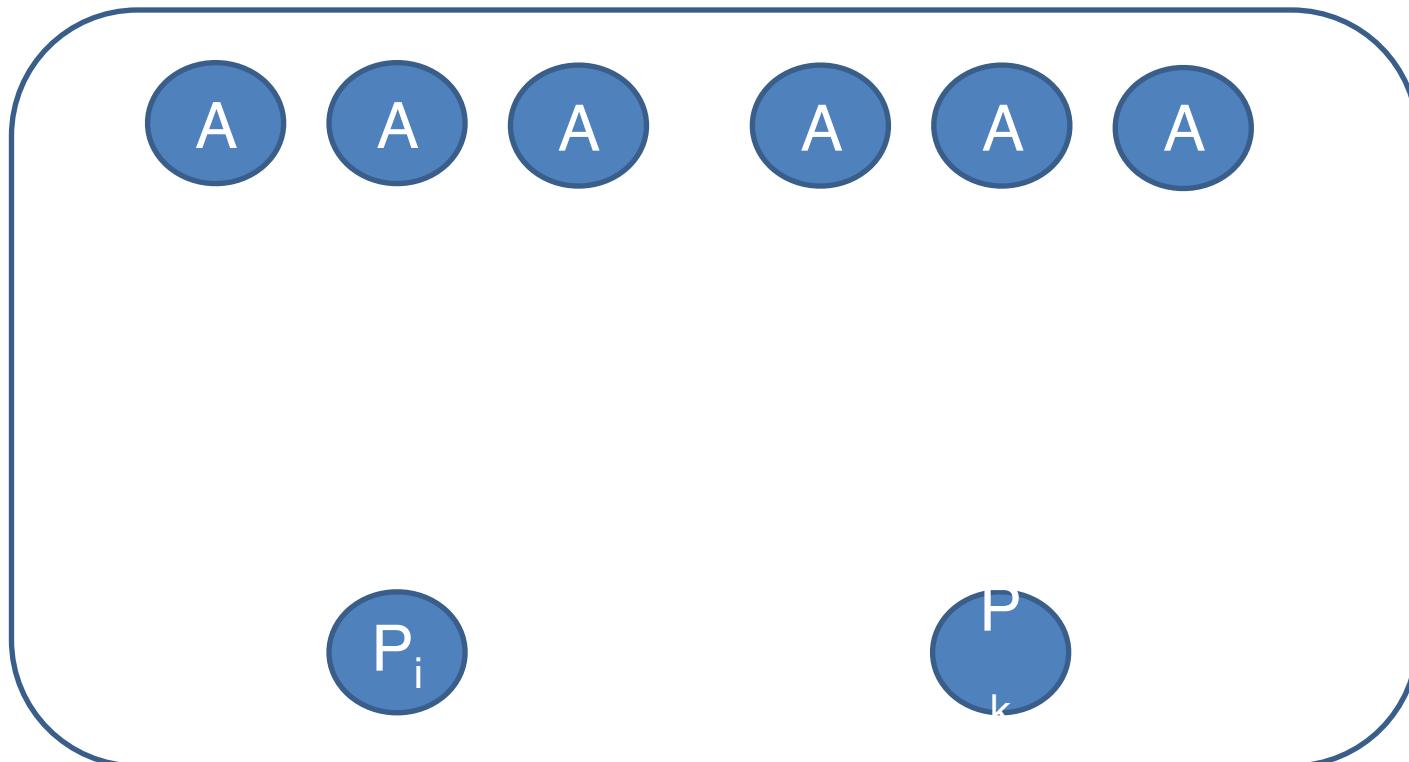
Strawman 2

(Multiple proposers, multiple acceptors)

- Proposer sends proposal to a **majority of acceptors** or more
- If a majority of acceptors chooses a particular proposal (value) then that value is considered chosen
- ***But what proposal to chose?***
- Each acceptor **accepts the first proposal** it receives and rejects the rest
- A proposer who receives a majority of replies (i.e., votes) from acceptors chooses its own **value**
 - Only one proposal will get a majority, so only one value will be chosen
- Proposer sends the chosen value to everyone

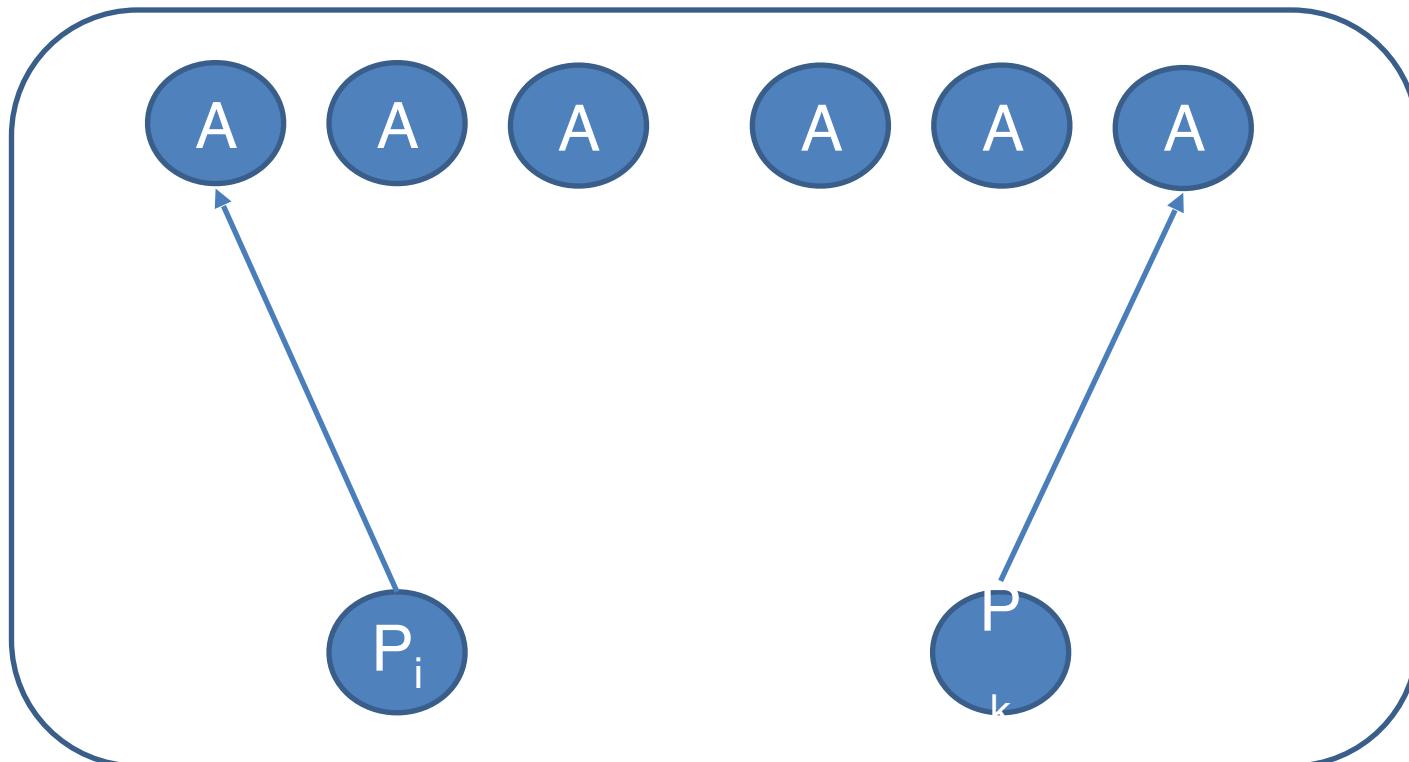
Strawman 2

(Multiple proposers, multiple acceptors)



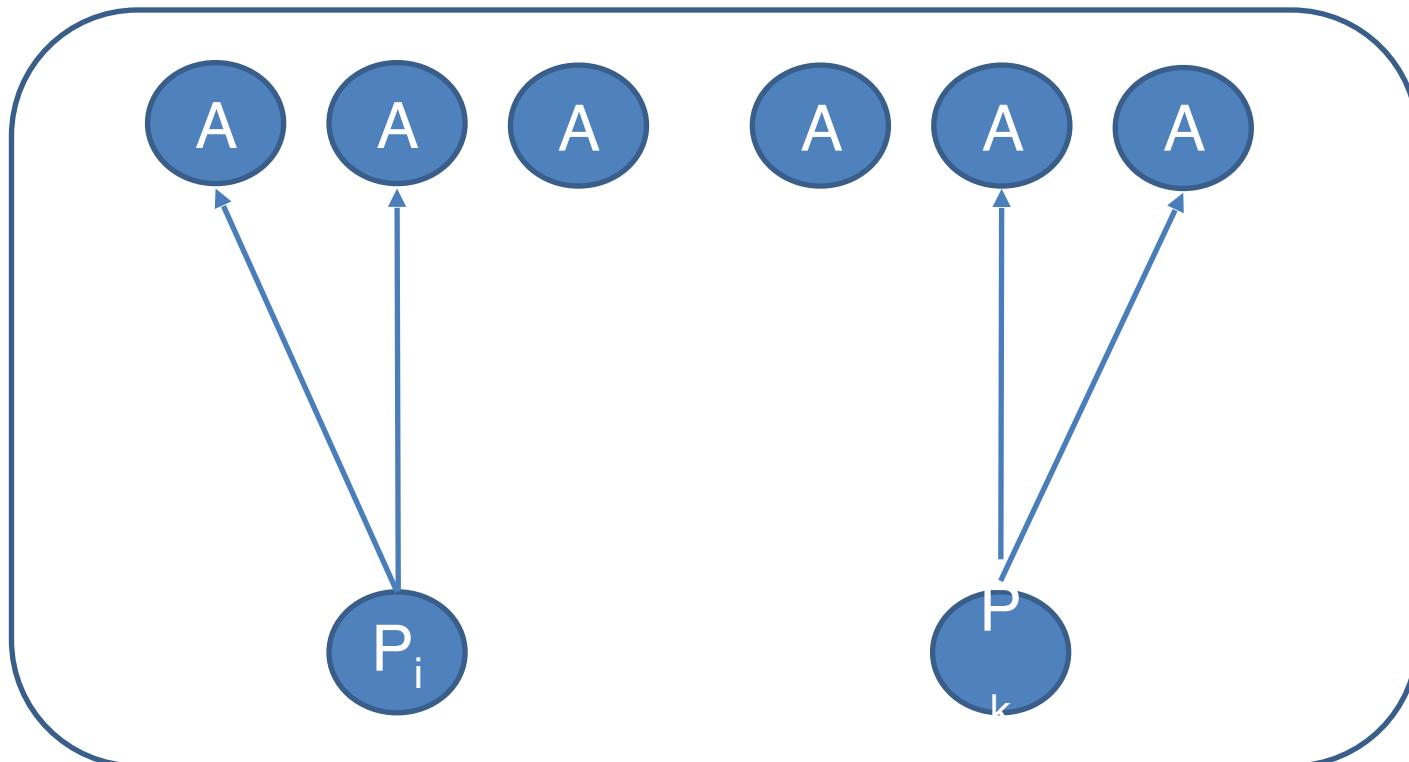
Strawman 2

(Multiple proposers, multiple acceptors)



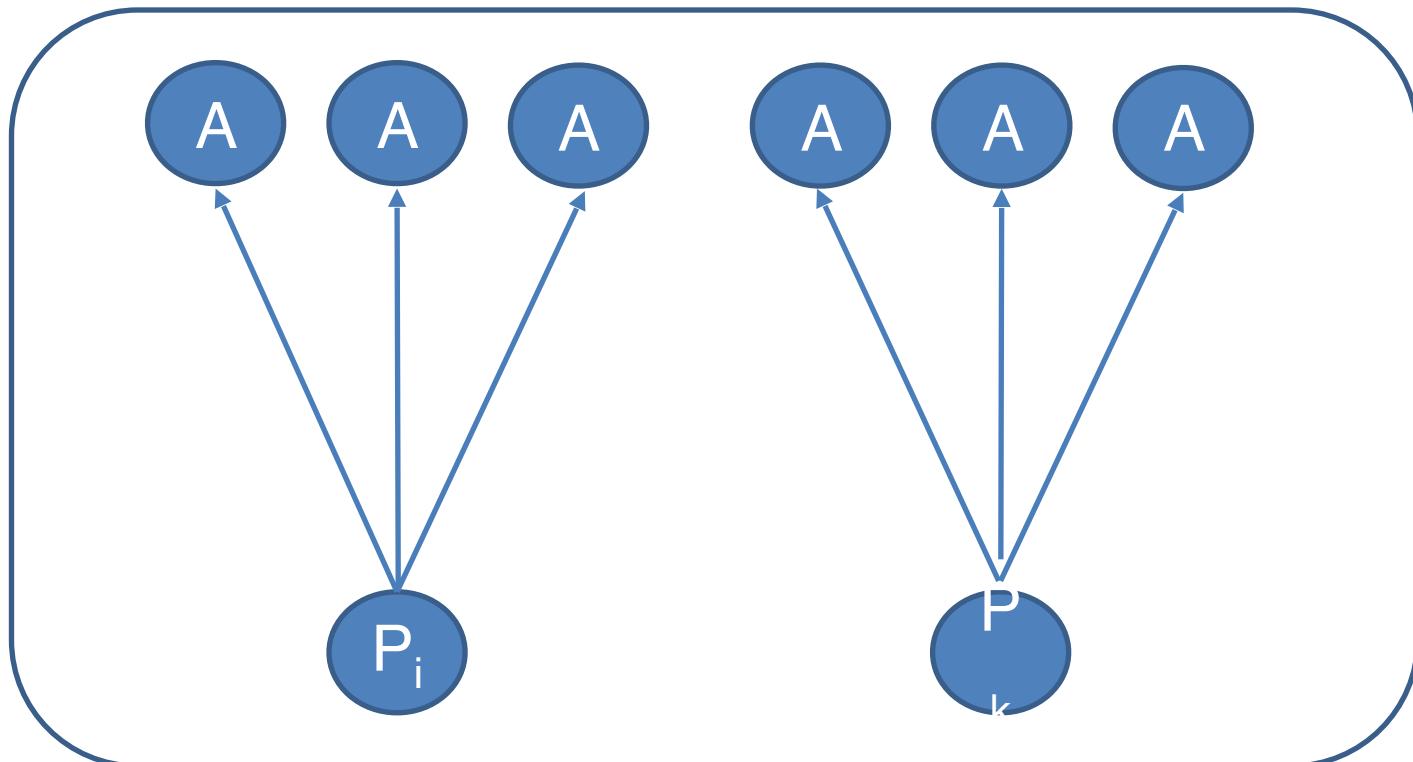
Strawman 2

(Multiple proposers, multiple acceptors)



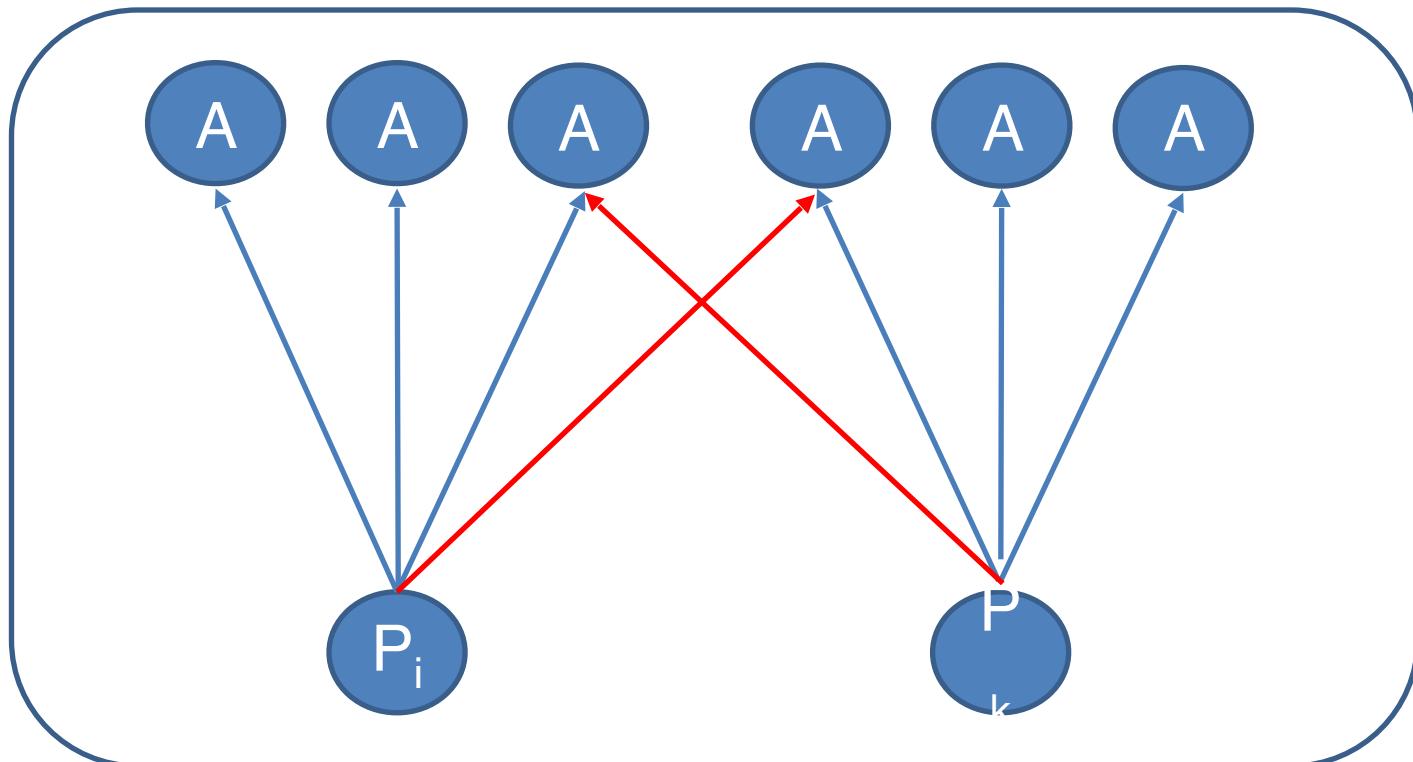
Strawman 2

(Multiple proposers, multiple acceptors)



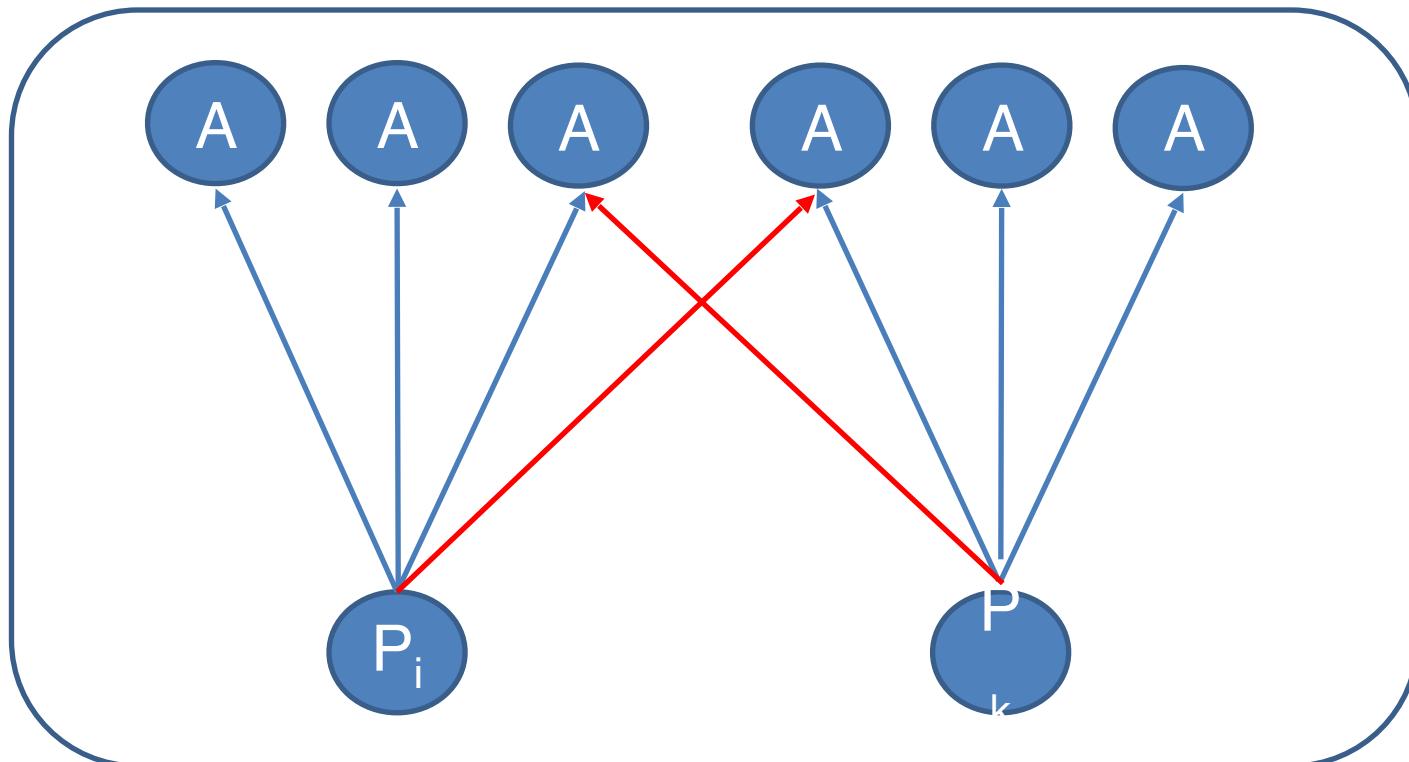
Strawman 2

(Multiple proposers, multiple acceptors)



Strawman 2

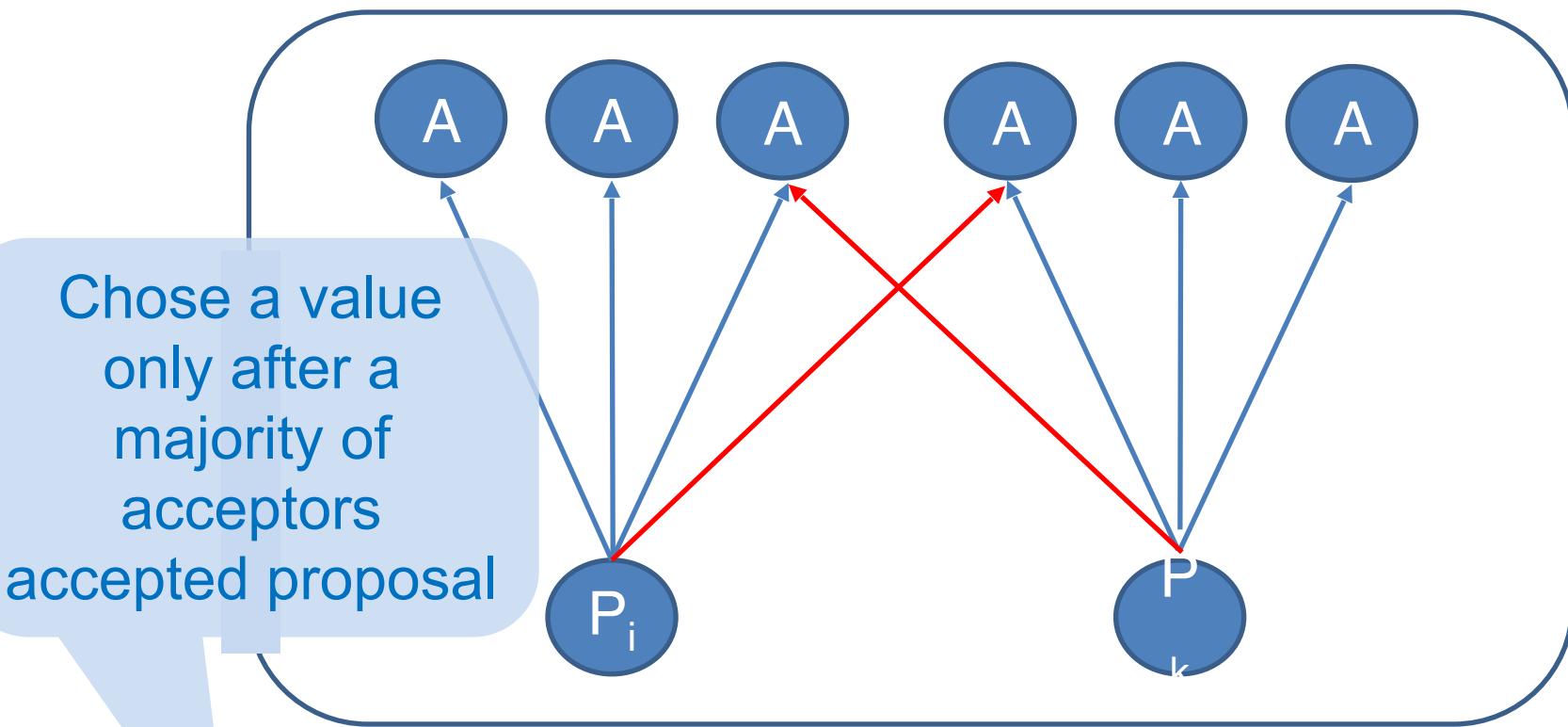
(Multiple proposers, multiple acceptors)



**Multiple proposers send proposals concurrently,
no one collects a majority!**

Strawman 2

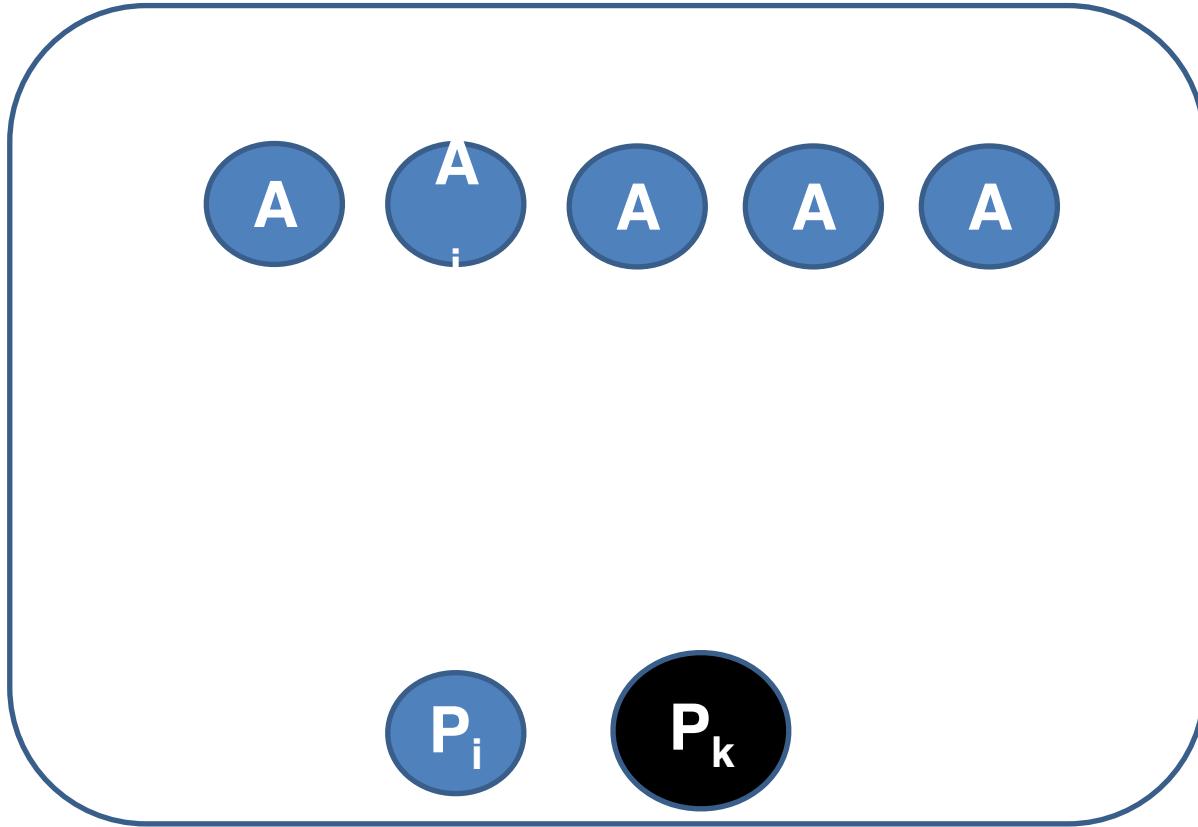
(Multiple proposers, multiple acceptors)



Need to allow for acceptors to change their mind about the proposal they accept.

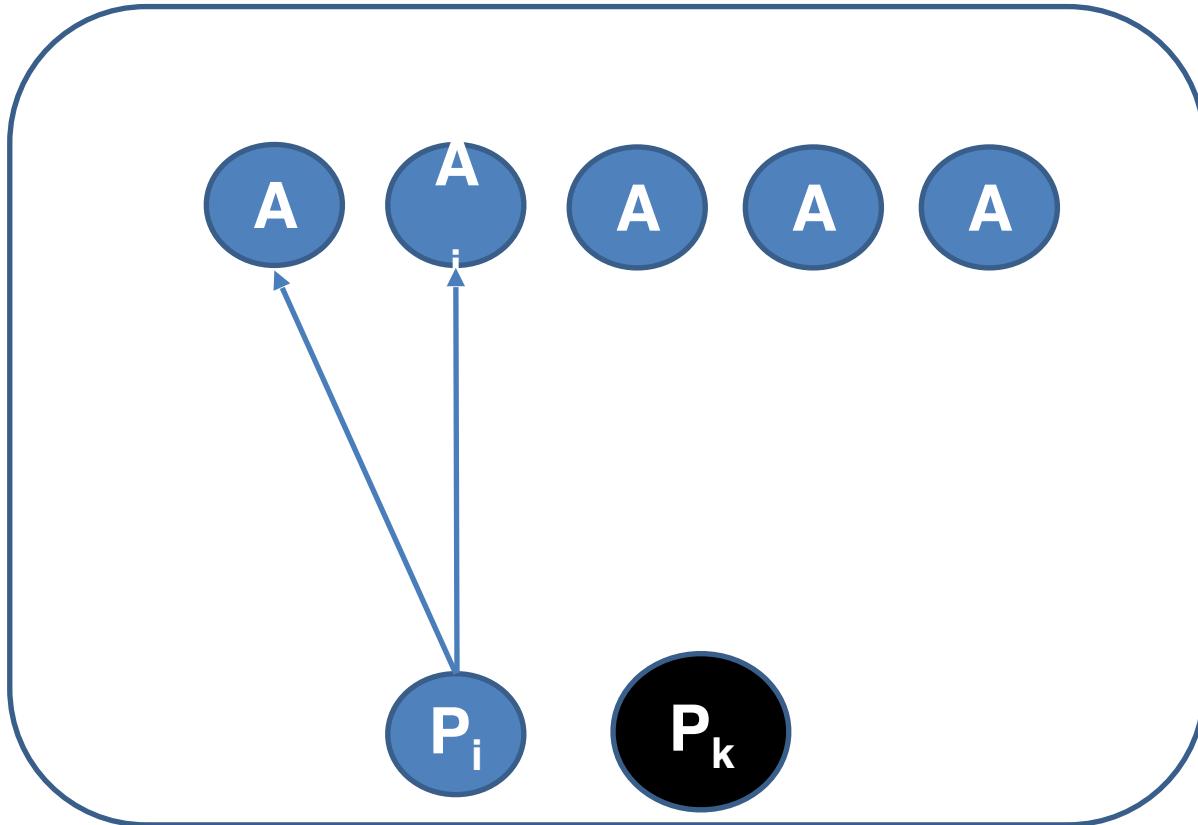
Strawman 3

(Acceptors change their mind, accept any proposal)



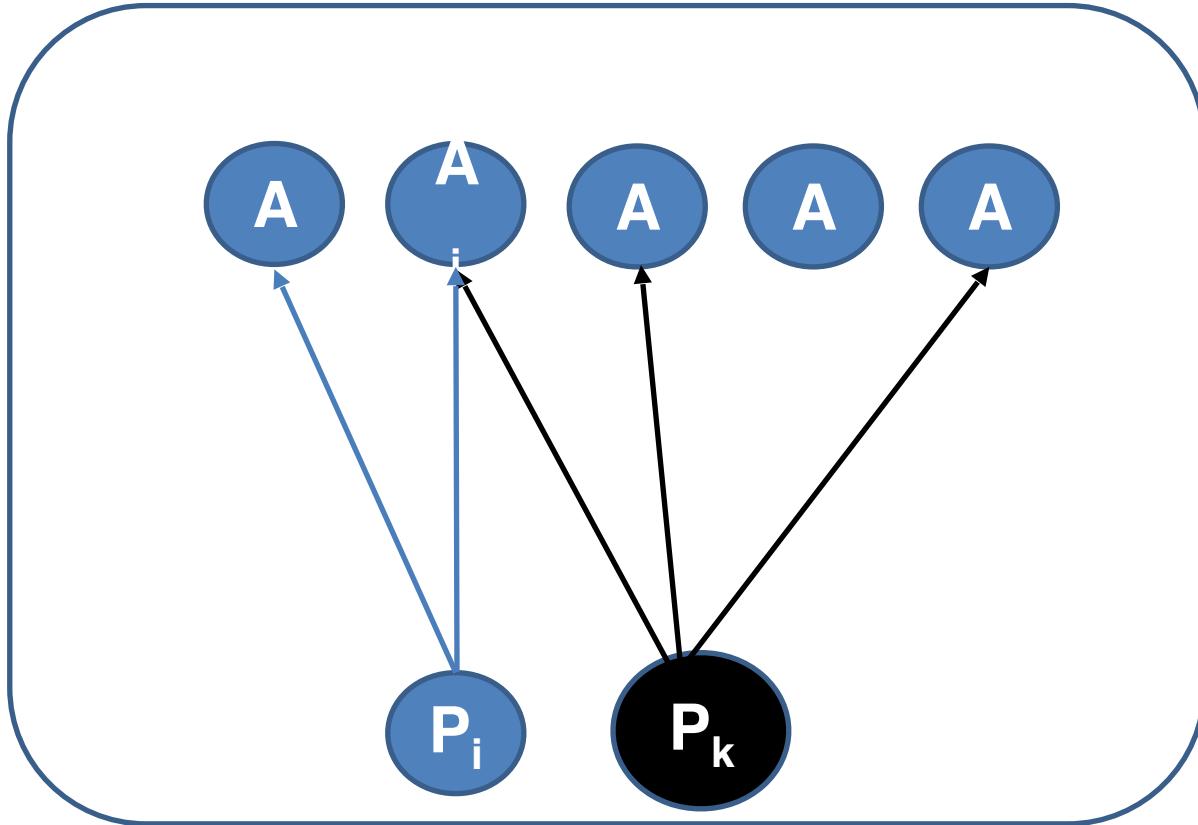
Strawman 3

(Acceptors change their mind, accept any proposal)



Strawman 3

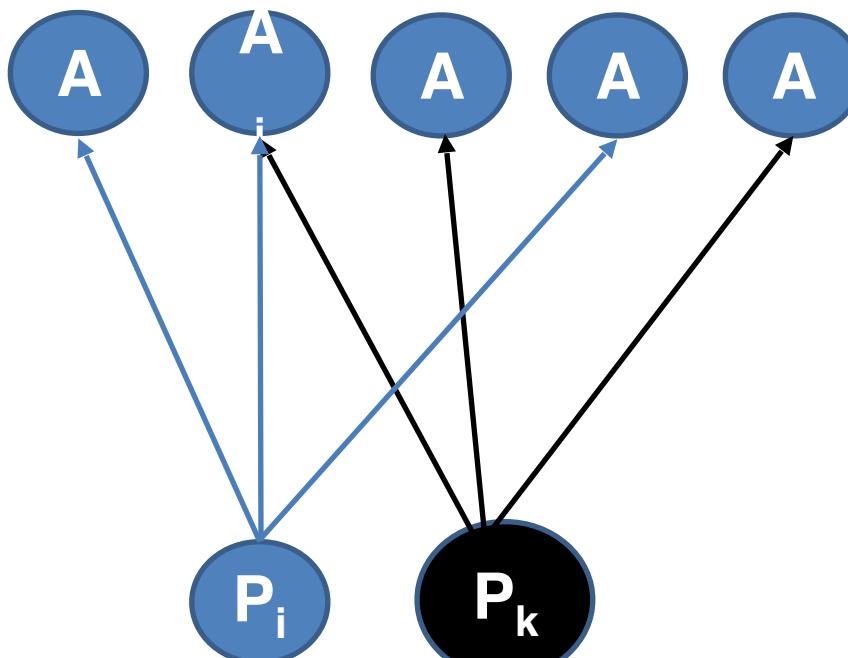
(Acceptors change their mind, accept any proposal)



Strawman 3

(Acceptors change their mind, accept any proposal)

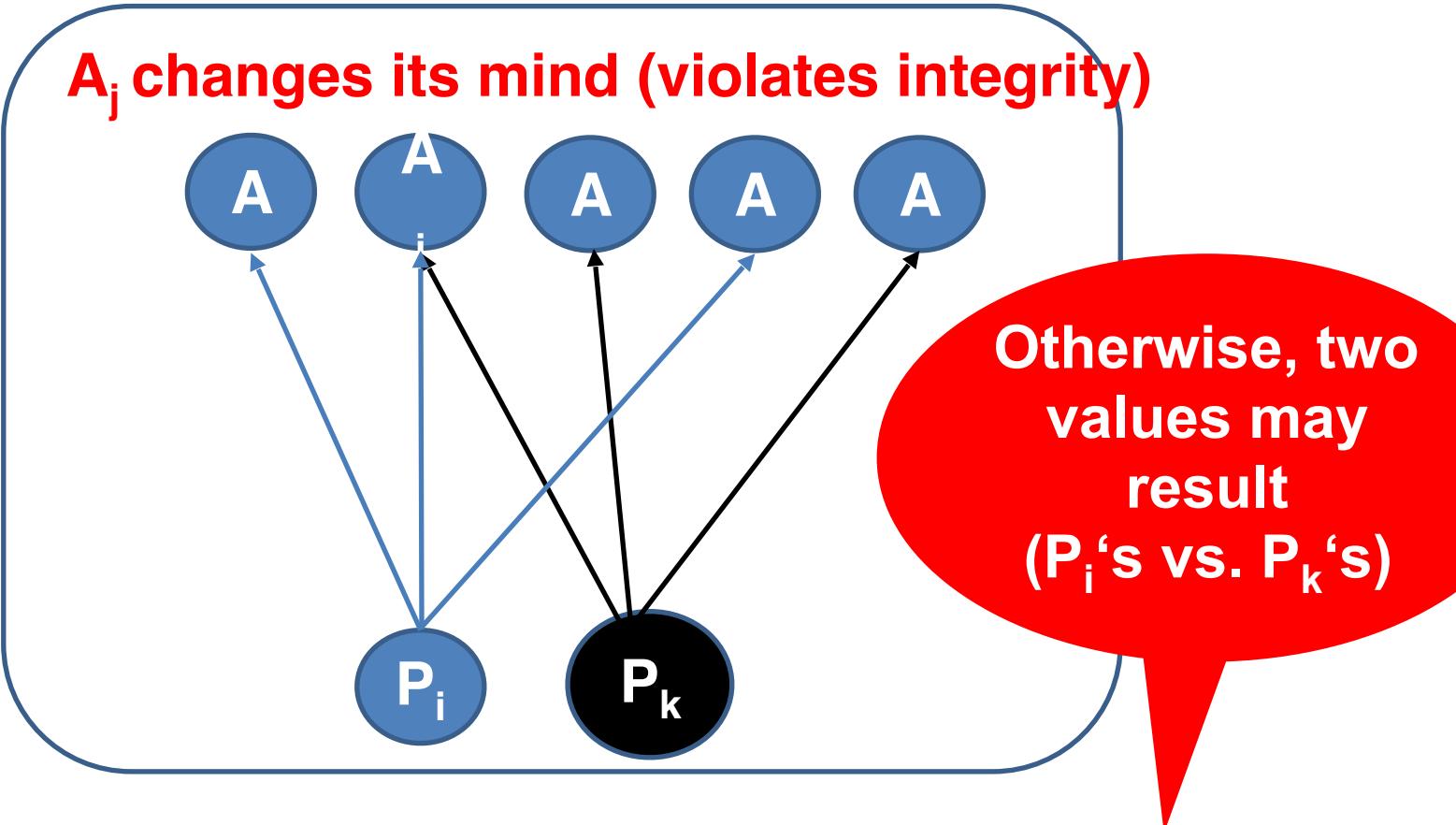
A_j changes its mind (violates integrity)



Integrity: Each node can decide a value at most once.

Strawman 3

(Acceptors change their mind, accept any proposal)

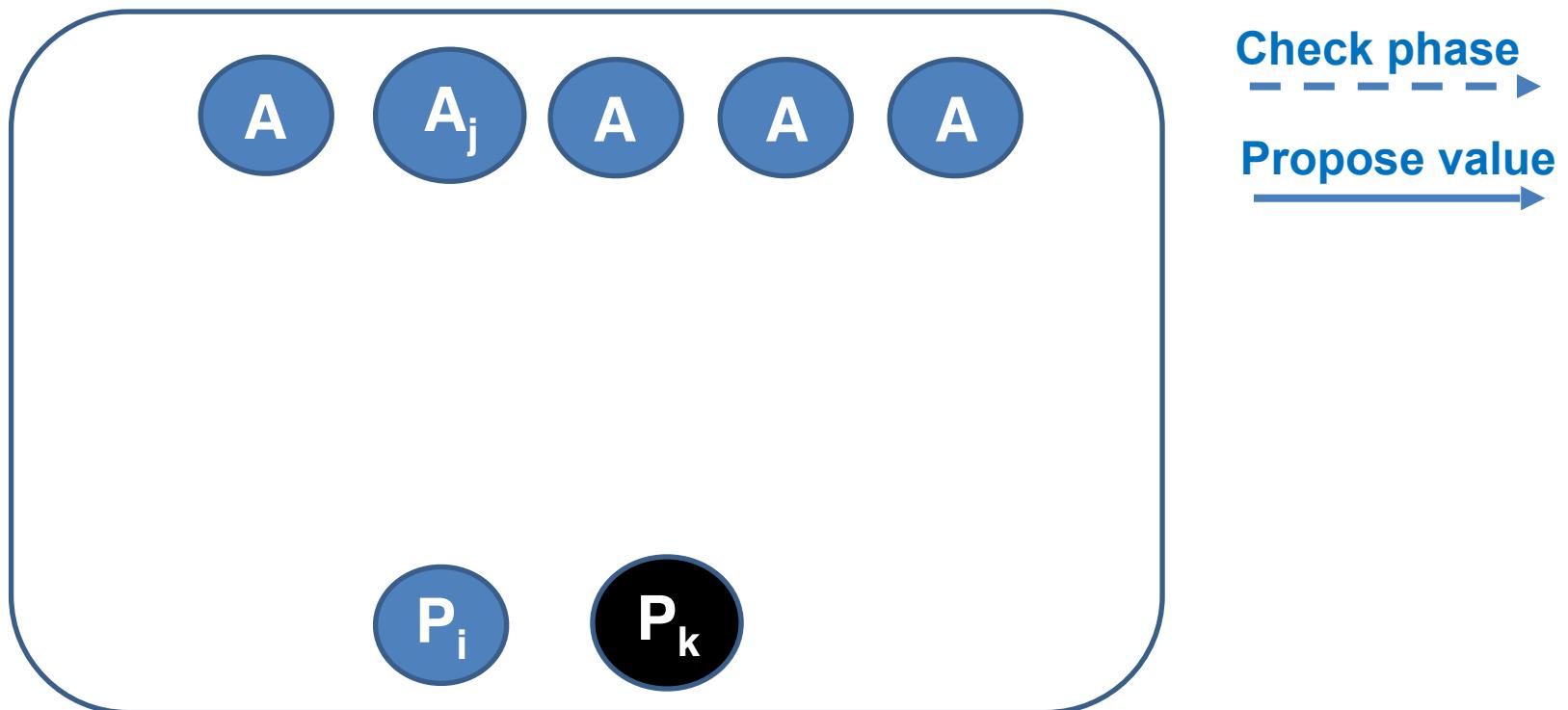


Strawman 3

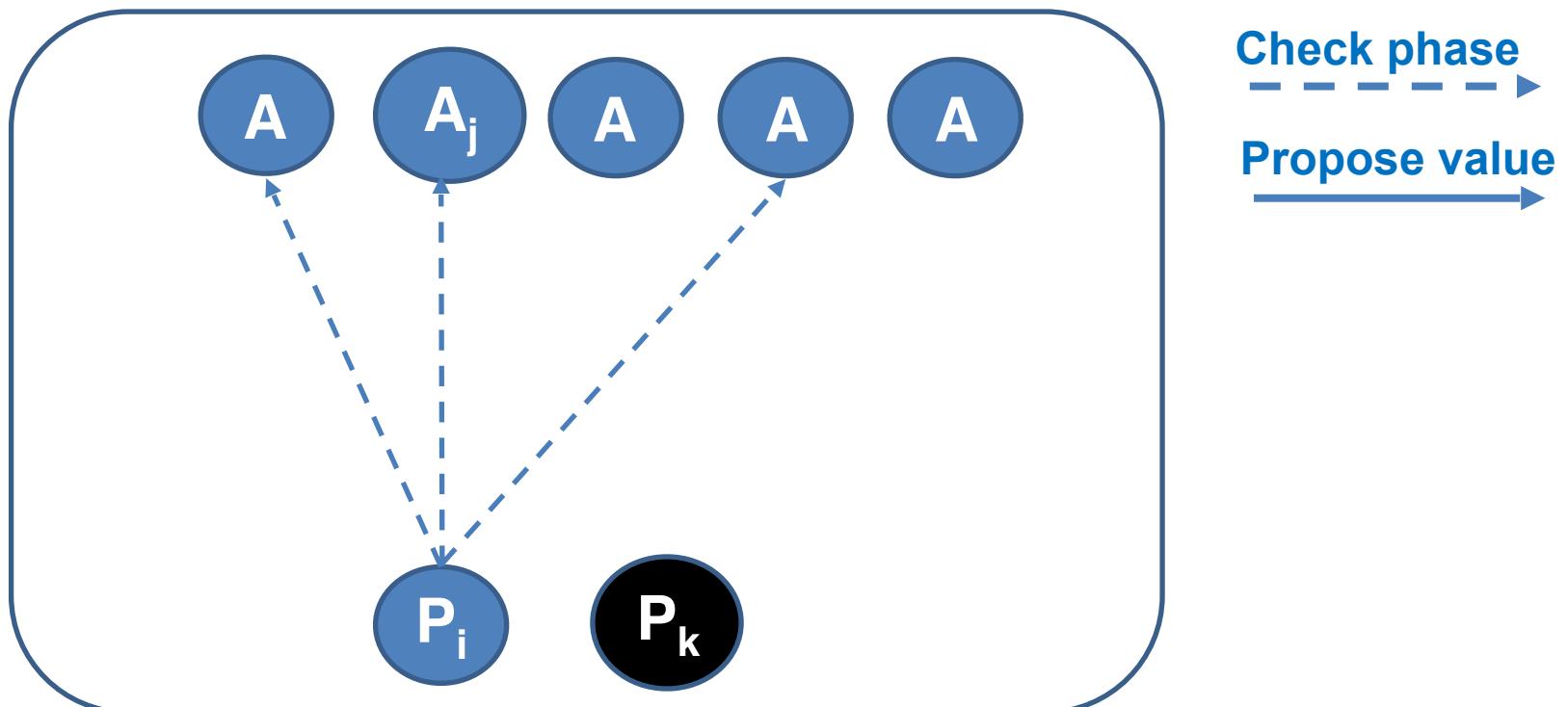
(Acceptors change their mind, accept any proposal)

- A proposer can't just propose a value
- Proposer needs to **first check with a majority of acceptors** whether a value has already been chosen
- If yes, proposer must adopt this value and propose it to a majority of ...
- If no, proposer is free to propose its value
- **Requires a two-phase protocol:** Check, act accordingly

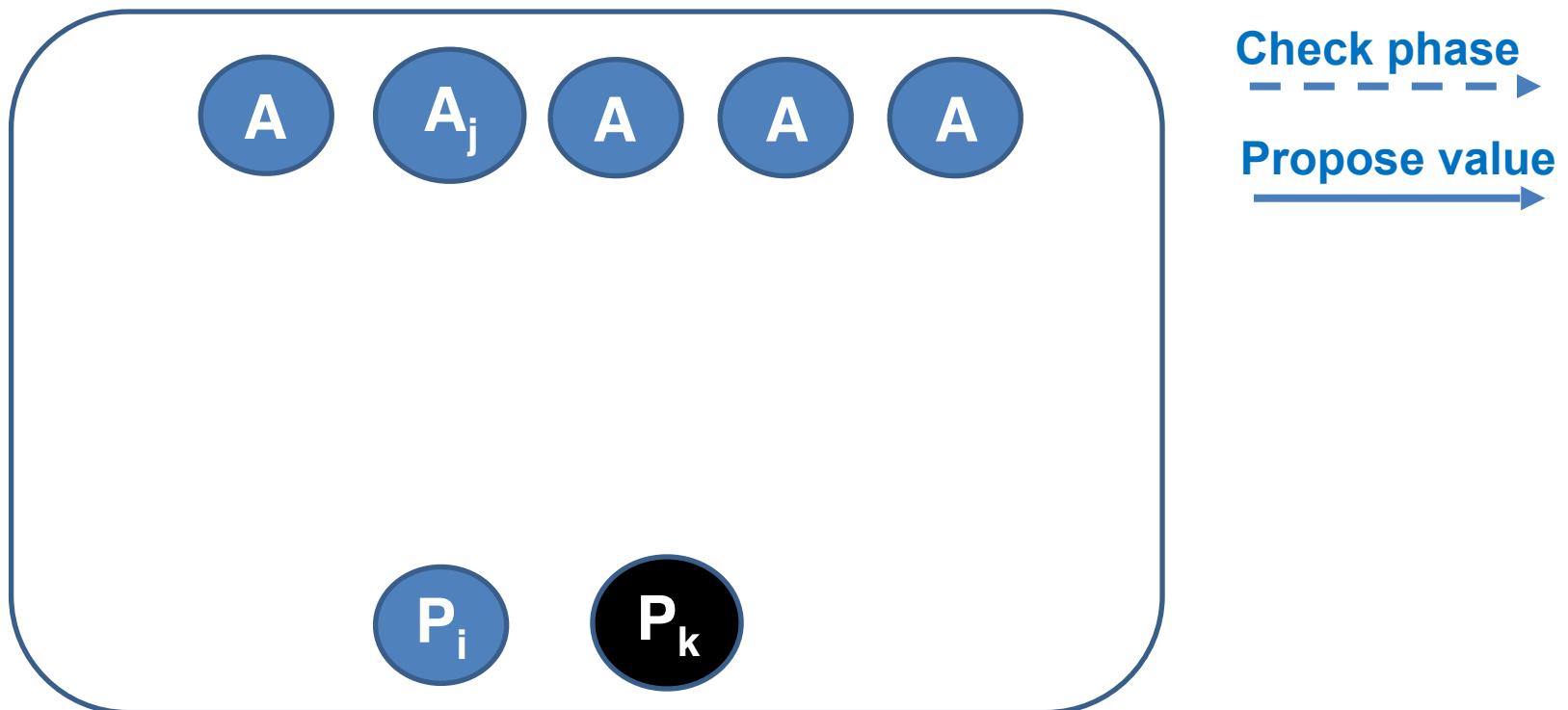
Strawman 4



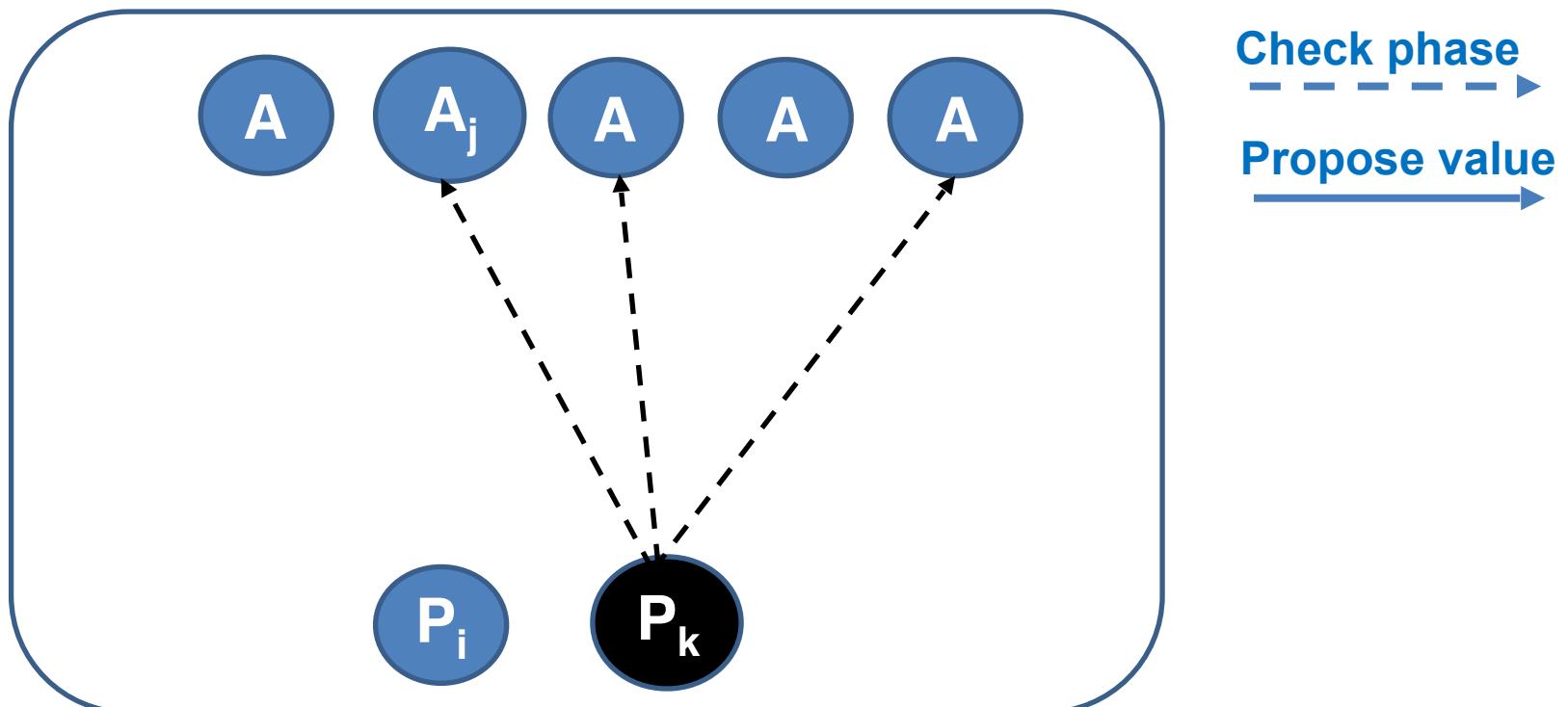
Strawman 4



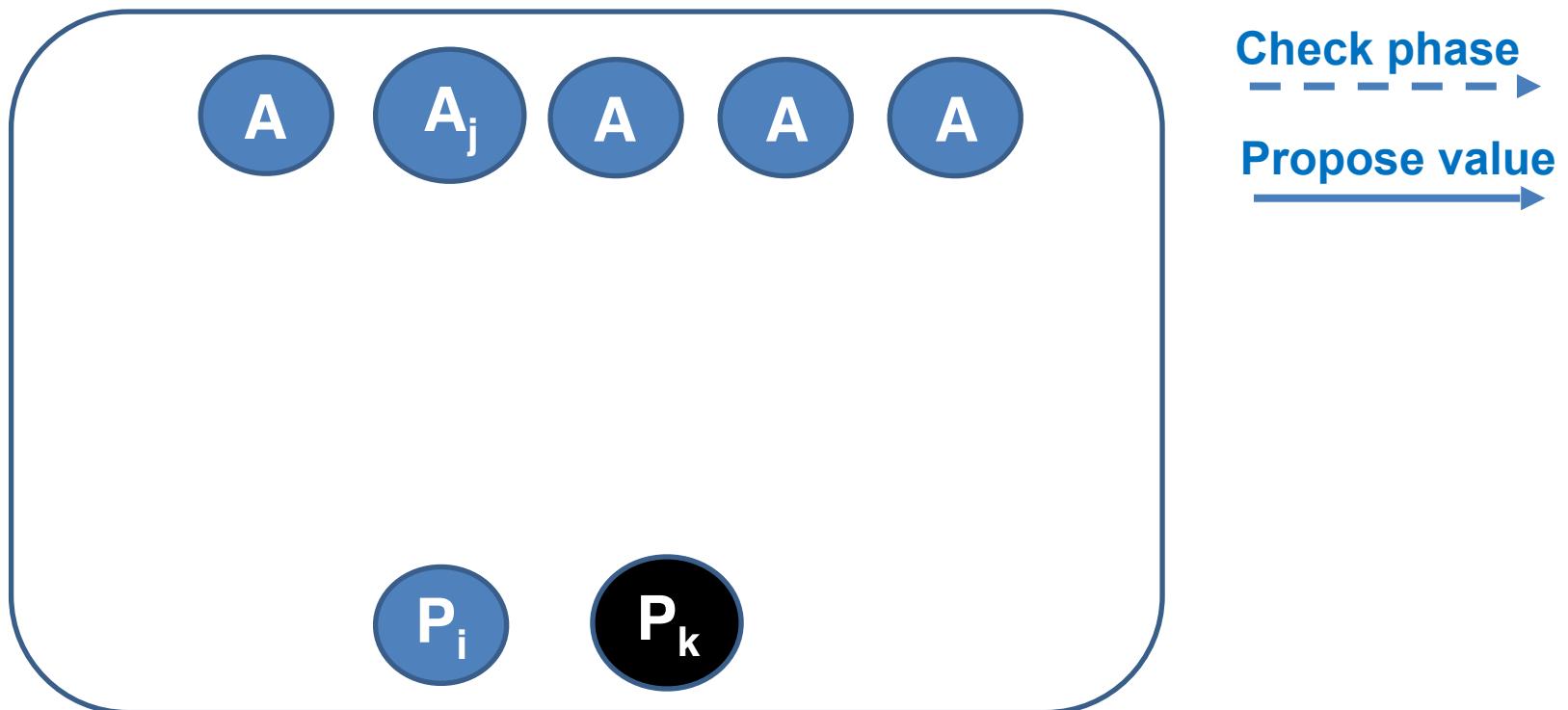
Strawman 4



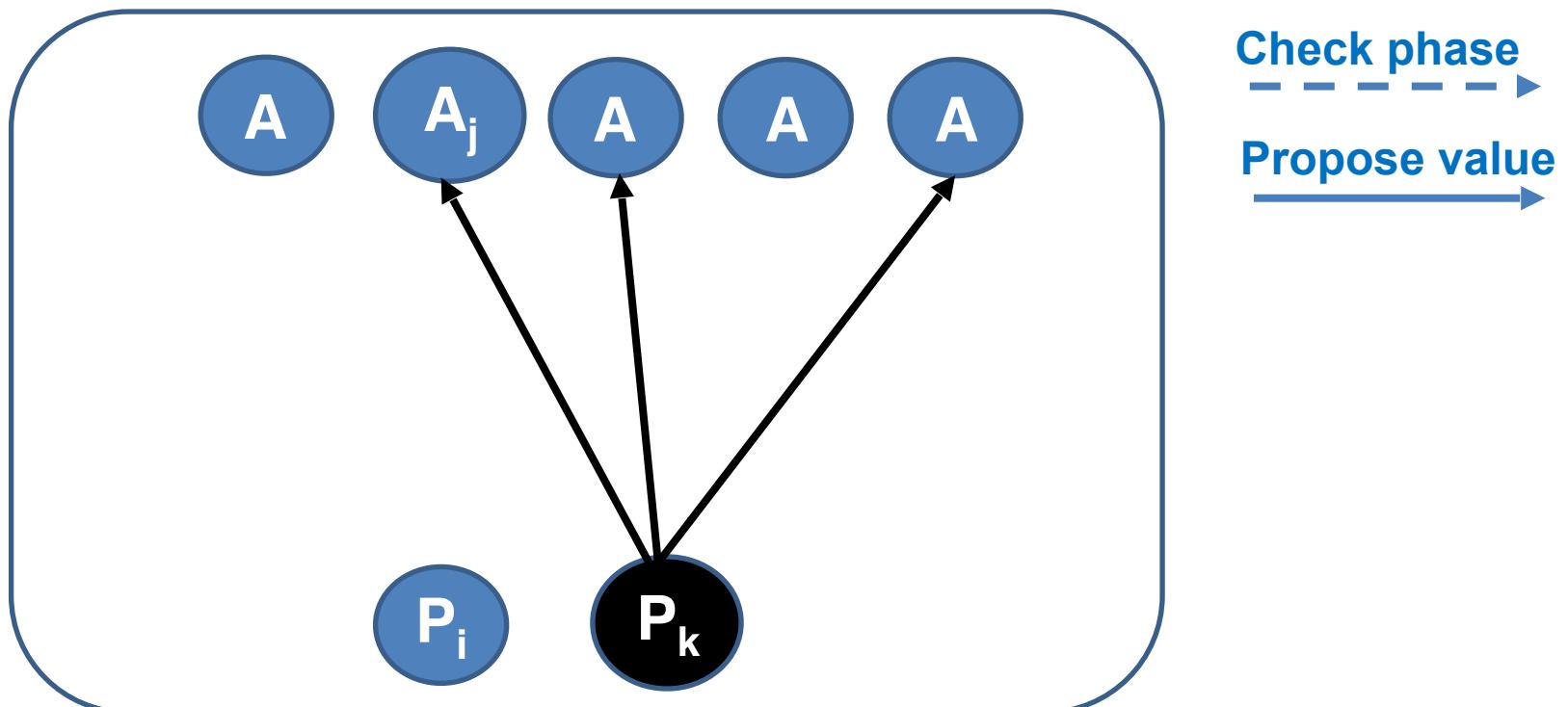
Strawman 4



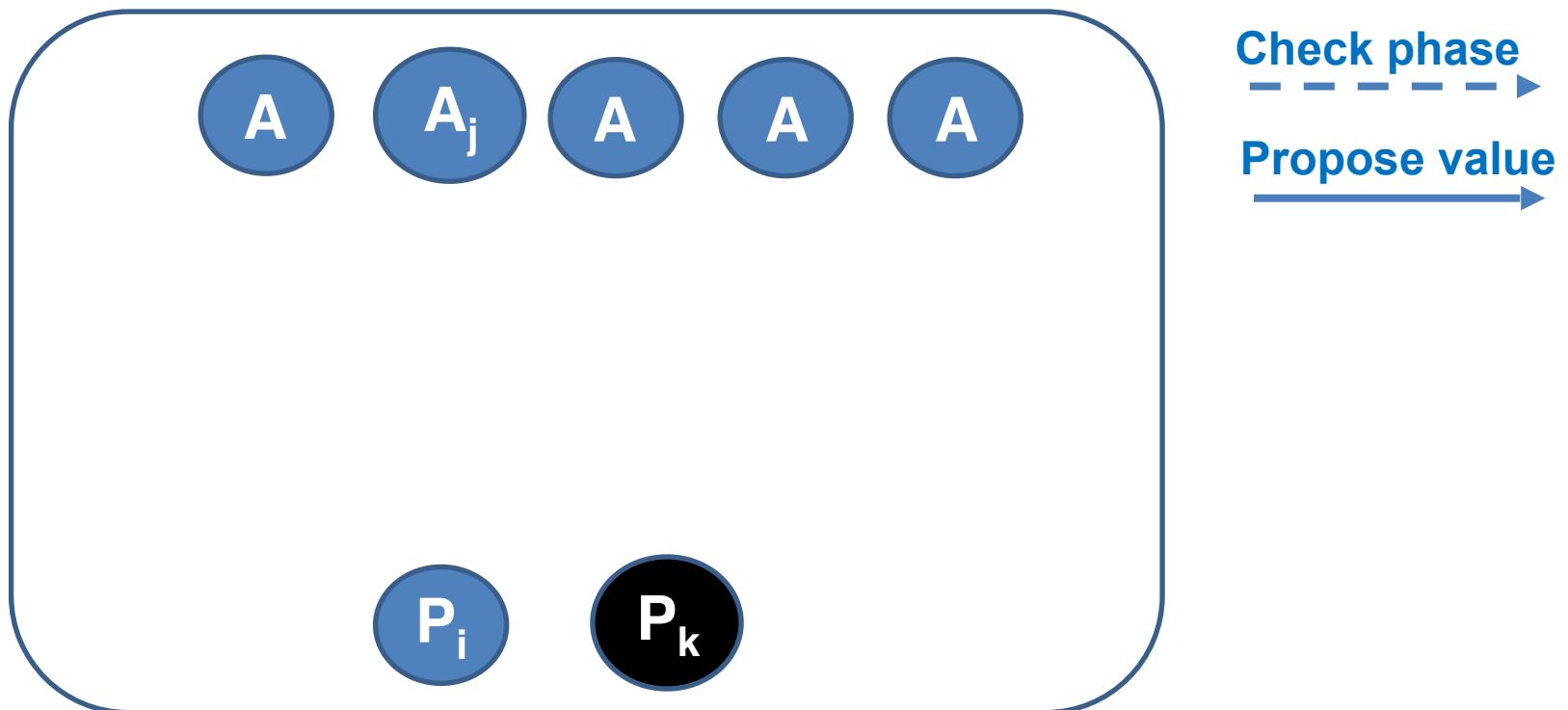
Strawman 4



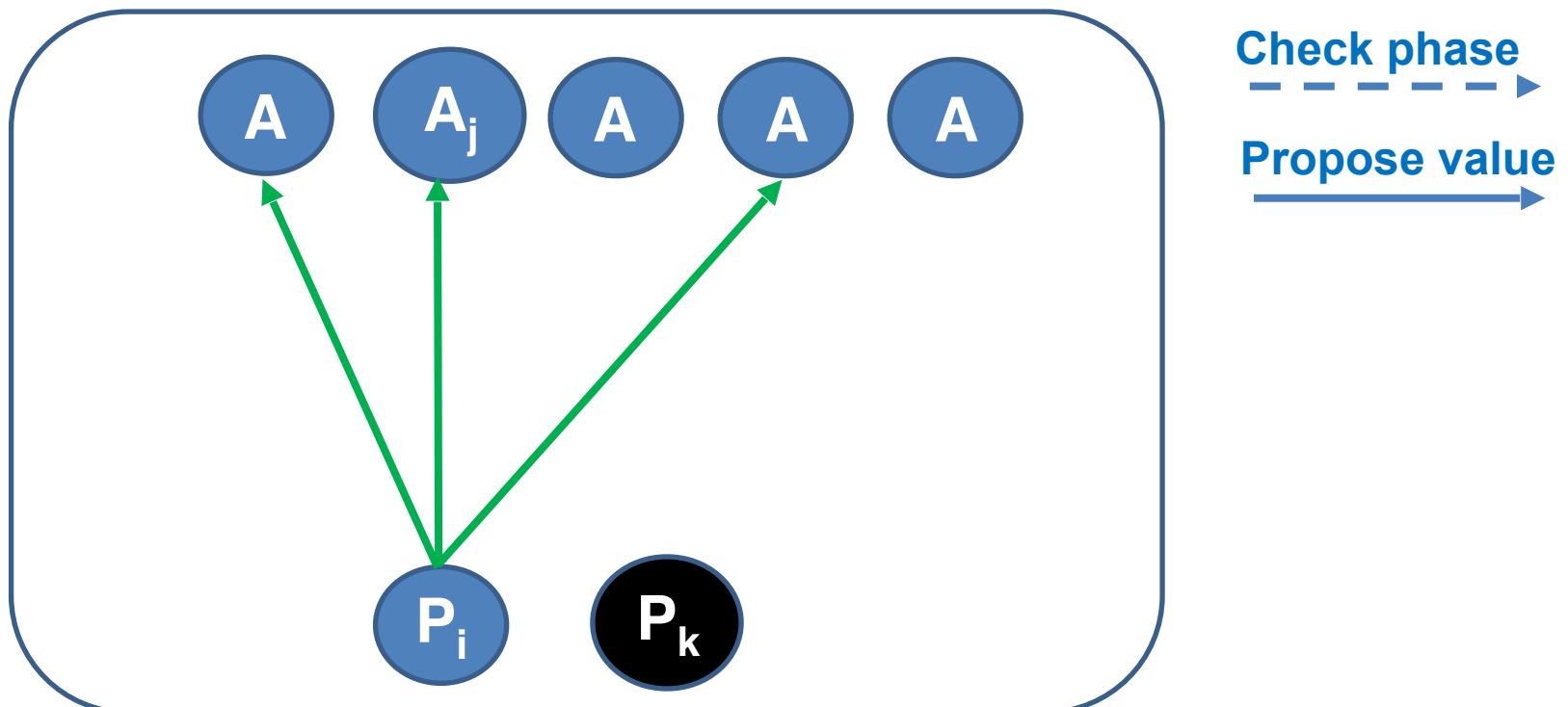
Strawman 4



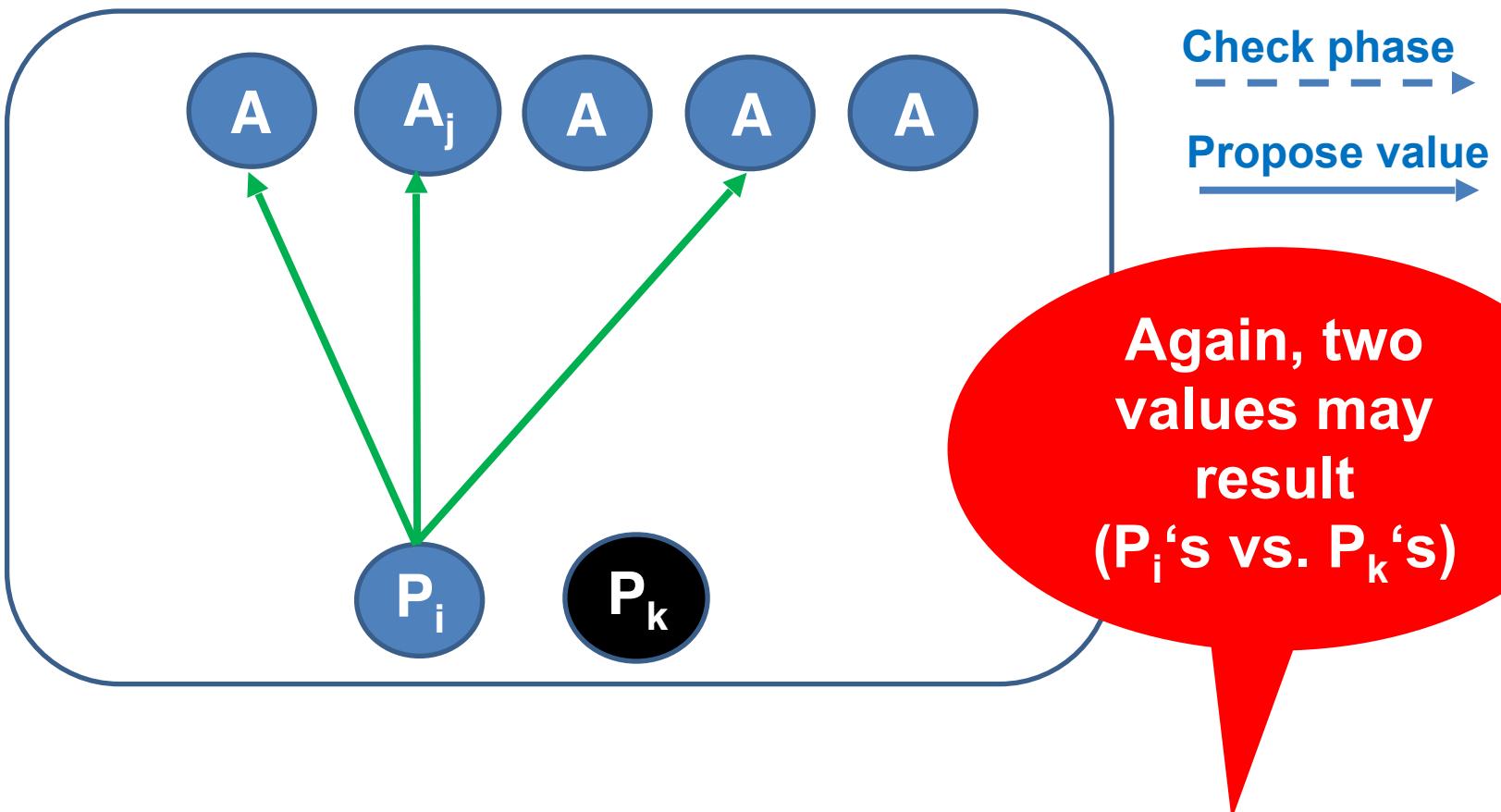
Strawman 4



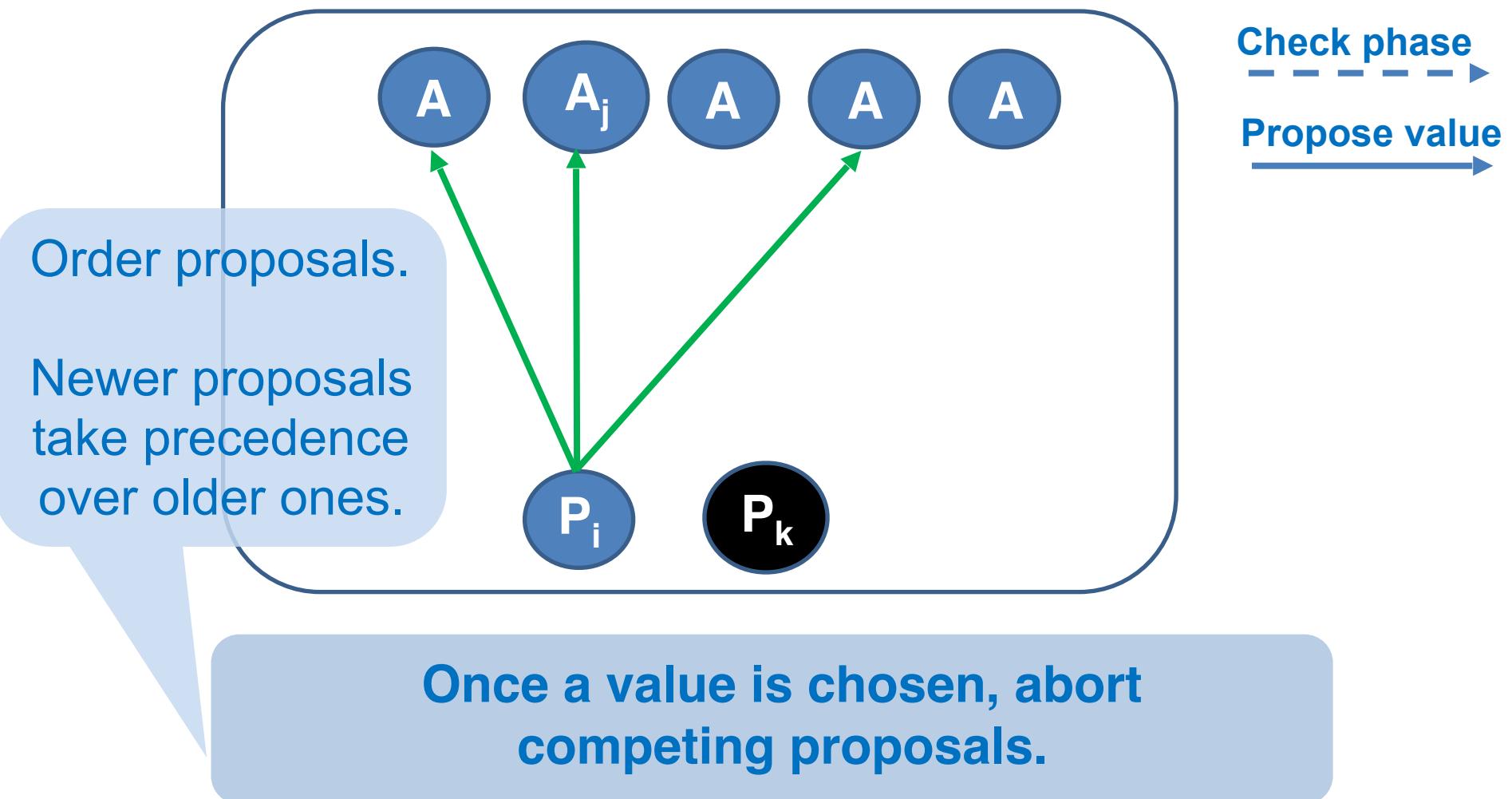
Strawman 4



Strawman 4



Strawman 4



Lessons learned from Strawmen 1-4

- Require multiple acceptors to tolerate faults
- Require votes on a value from a majority of acceptors to break ties
- Need to allow for acceptors to change their mind about proposals they accept to establish a majority
- Require a two-phased protocol to check for prior decisions
- Require means to order proposals to ensure single value results

The Paxos way

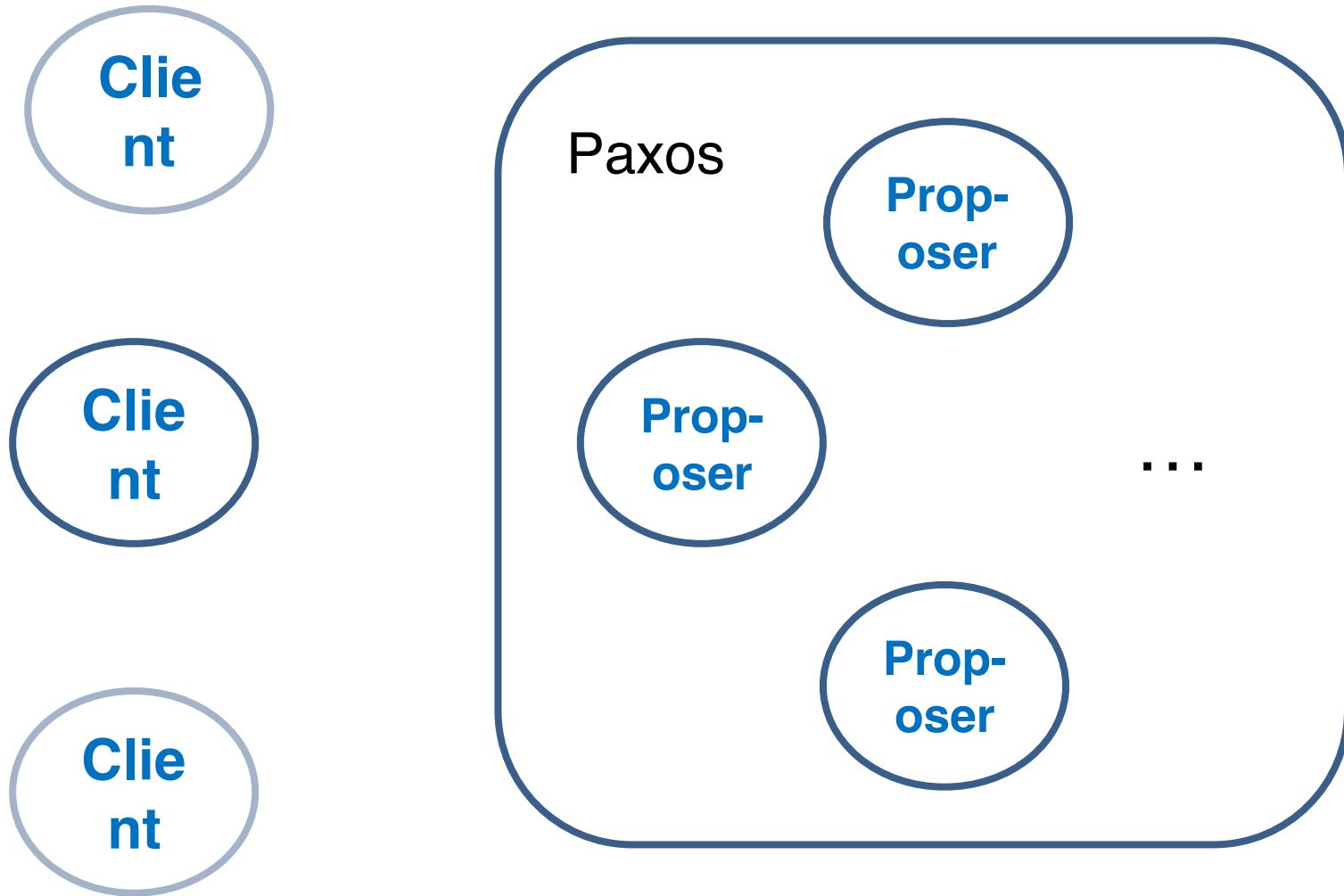
Preview

- Each acceptor may be able to accept multiple proposals **based on ordering**
- But, each acceptor must follow two rules:
 - If it **promised** to one proposal with a P# n , it can only **accept** a **newer proposal** with a P# m , where $m > n$.
 - If it promises to a newer proposal, it will ask the proposer to **set the value of the newer proposal** to the **same value as an older accepted proposal**.

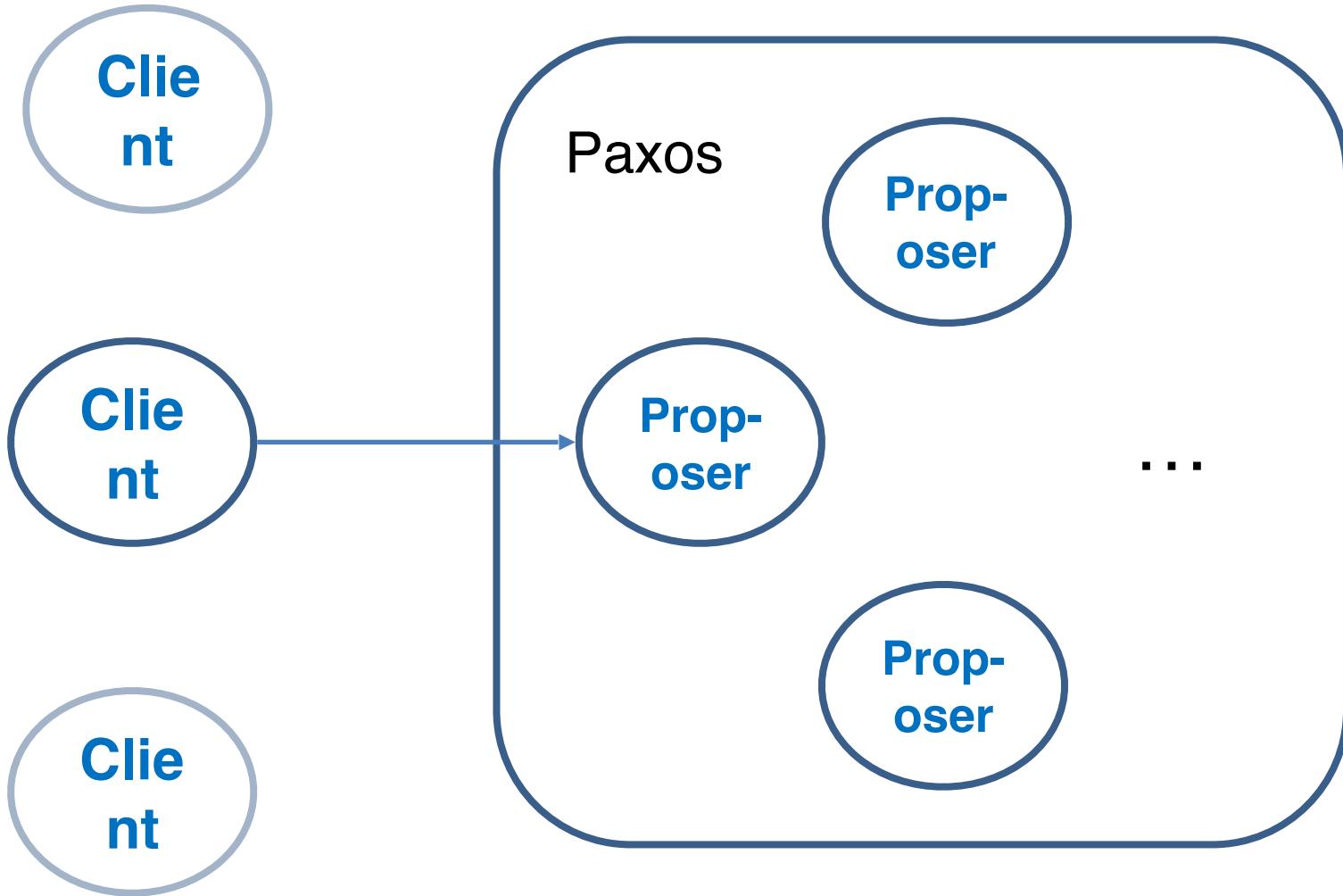
BASIC PAXOS IN DETAIL

Part 3

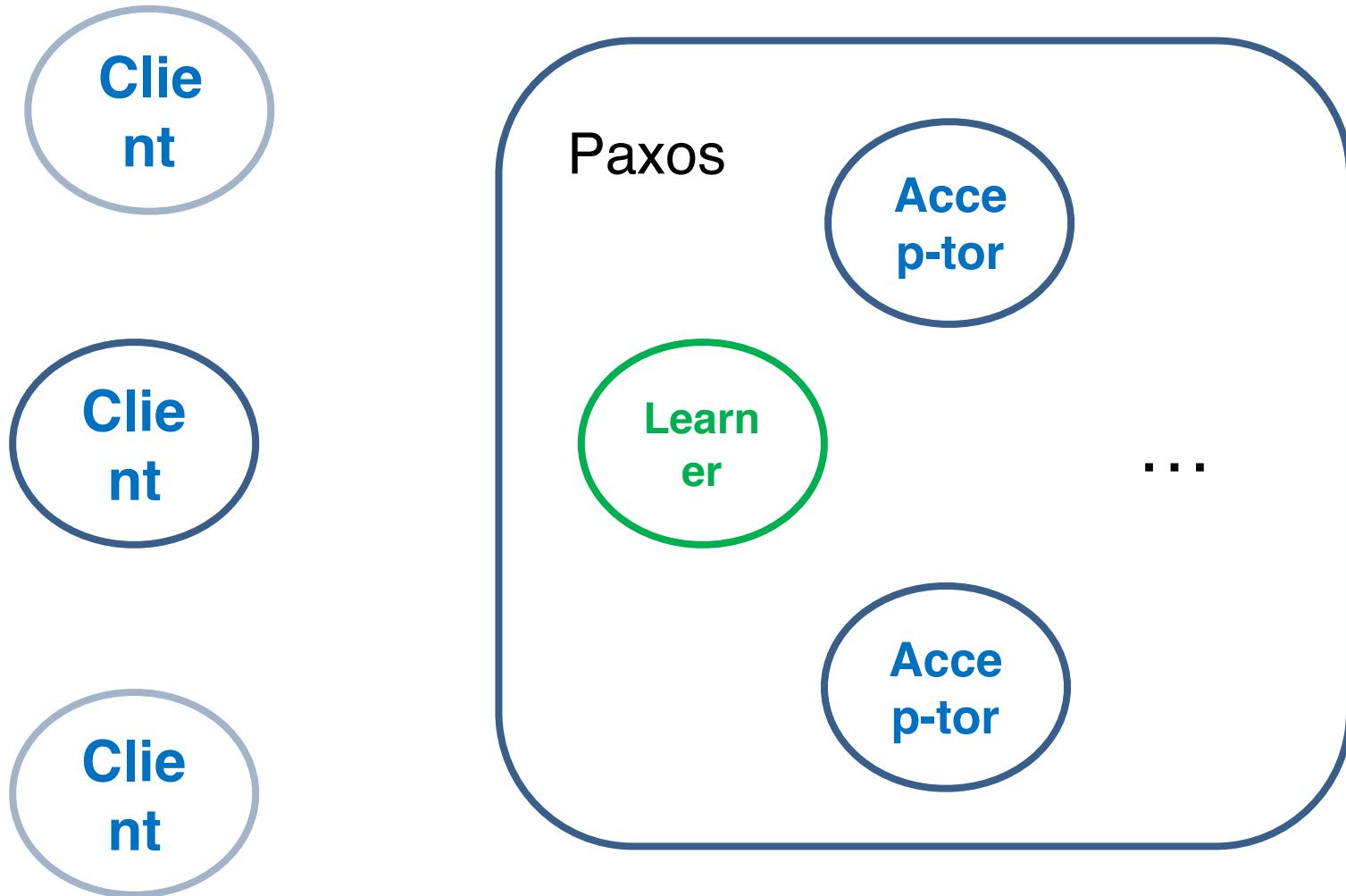
Clients



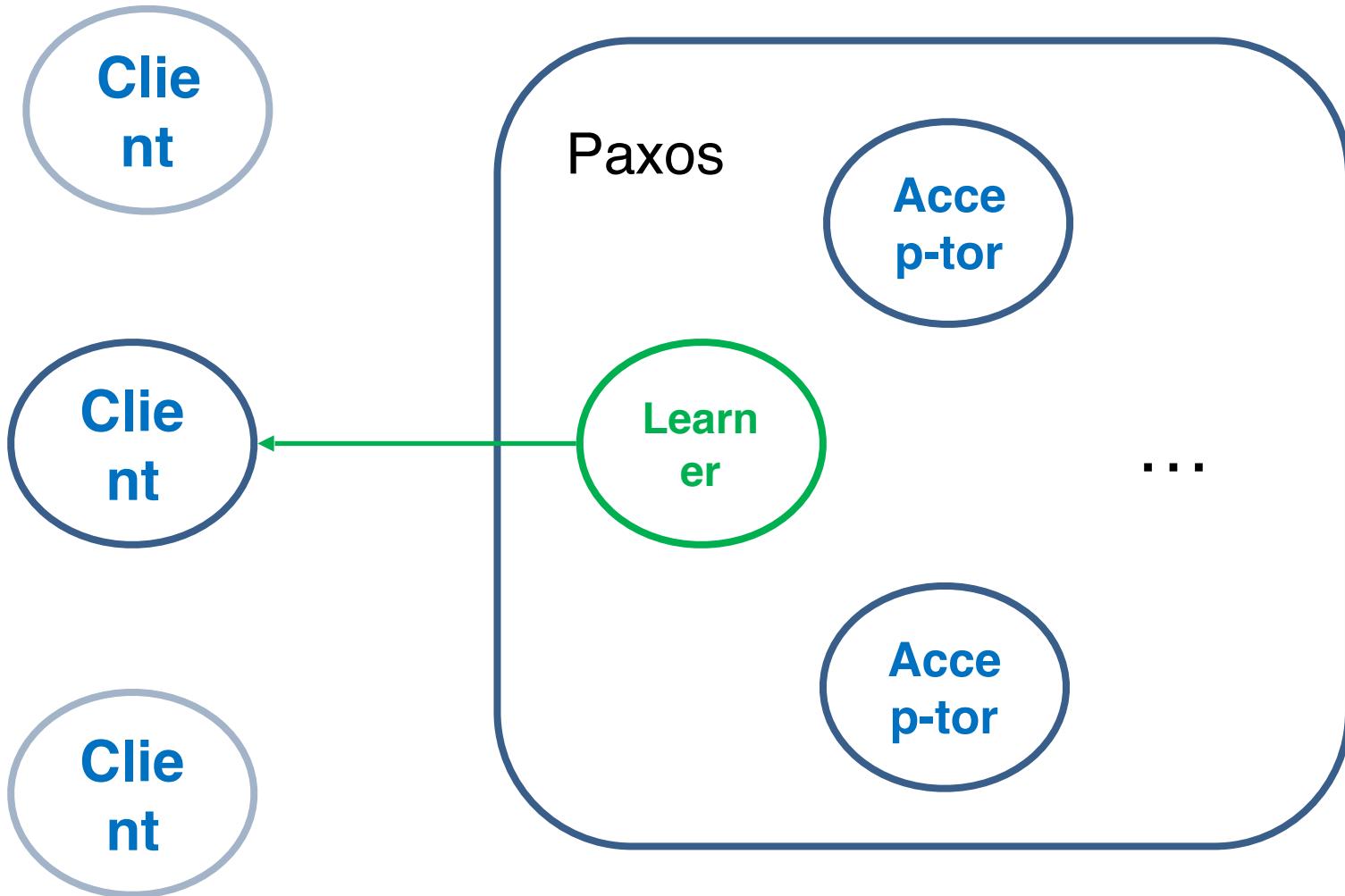
Clients



Clients



Clients



Clients

- Processes which interact with the Paxos protocol:
Do not participate in protocol
- Clients send **requests to proposer** and wait for **response from learner**
- Basic Paxos **decides which request is accepted** (in multi-paxos, a sequence is decided)
- The nature of requests depends on how Paxos is used

Role of proposers in Paxos

- Convince acceptors to agree on **client requests**
- Paxos refers to **client request** simply as **values**
- Proposer issues proposals for a value to ultimately obtain agreement from Paxos
- There could be several different client requests (values) that are concurrently being proposed by different proposers
- One instance of Paxos determines one value that all correct nodes agree on
- Proposers need to initially check whether any other value has been decided on by acceptors (first phase of protocol)

Proposal number & agreed value

- **Messages** sent by proposer
 - prepare (P# (N))** First phase
 - acceptReq (P# (N), Value (V))** Second phase
- where **P#** refers to the **proposal number** (cf. below)
- Attempt to define an agreed value **V** via **proposals**
- Proposals may or may not be accepted by **acceptors**
- **Proposers** assign monotonically increasing numbers to each proposal (called the **P#**), e.g., 1,3,4,8,...*
- **Value V** represents the information that is to be agreed on (i.e., input from the client, client request)

*Proposal number

- **Must be unique** (required)
 - No two proposers can have the same number.
- Should be larger than any previous proposal number (by any proposer in instance of protocol)
(desirable)
 - If this is not the case, proposer would find out by having its proposal rejected (proposer would have to re-try)
- Use a globally unique node/process identifier in lower-order bits and an incrementing counter in high-order bits
 - E.g., (logical clock, unique node/process identifier)-pair
 - E.g., (2345 1921681215555)

Acceptors

- **Messages** sent by acceptor
 - promise** ($P\#$, old $P\# (N)$, old Value (V)) First phase
 - accepted** ($P\#, V$) Second phase
- Acceptors decide which proposal to accept and memorize its value (i.e., write it to stable storage)
- Any proposal must be sent to a **quorum of acceptors** (e.g., **3 out 5 acceptors**)
 - Usually, proposal is sent to every acceptor
- A proposal **must** be accepted by a **quorum** in order to pass
- Note: Paxos provides **consensus**, but **internally it only requires majority** from acceptors!
 - This makes this step more fault-tolerant than strict unanimity
 - I.e., tolerates failures

Quorum

- A quorum is a subset of acceptors such that two quorums share at least one member.
- Using quorum ensures there exists at least one acceptor who can decide between two proposals (**safety**)
- Typically, any **majority of participating acceptors**
- **Example**, given **acceptors {A,B,C,D}**:
 - **Majority quorum** would be any three **acceptors** (or more):
 {A,B,C}, {A,C,D}, {A,B,D}, {B,C,D}, {A,B,C,D}
 - **Weighted quorum**, if $w_A = 2$, else 1, weight 3 needed:
 {A,B}, {A,C}, {A,D}, {B,C,D}, {A,B,C,D}

Learner

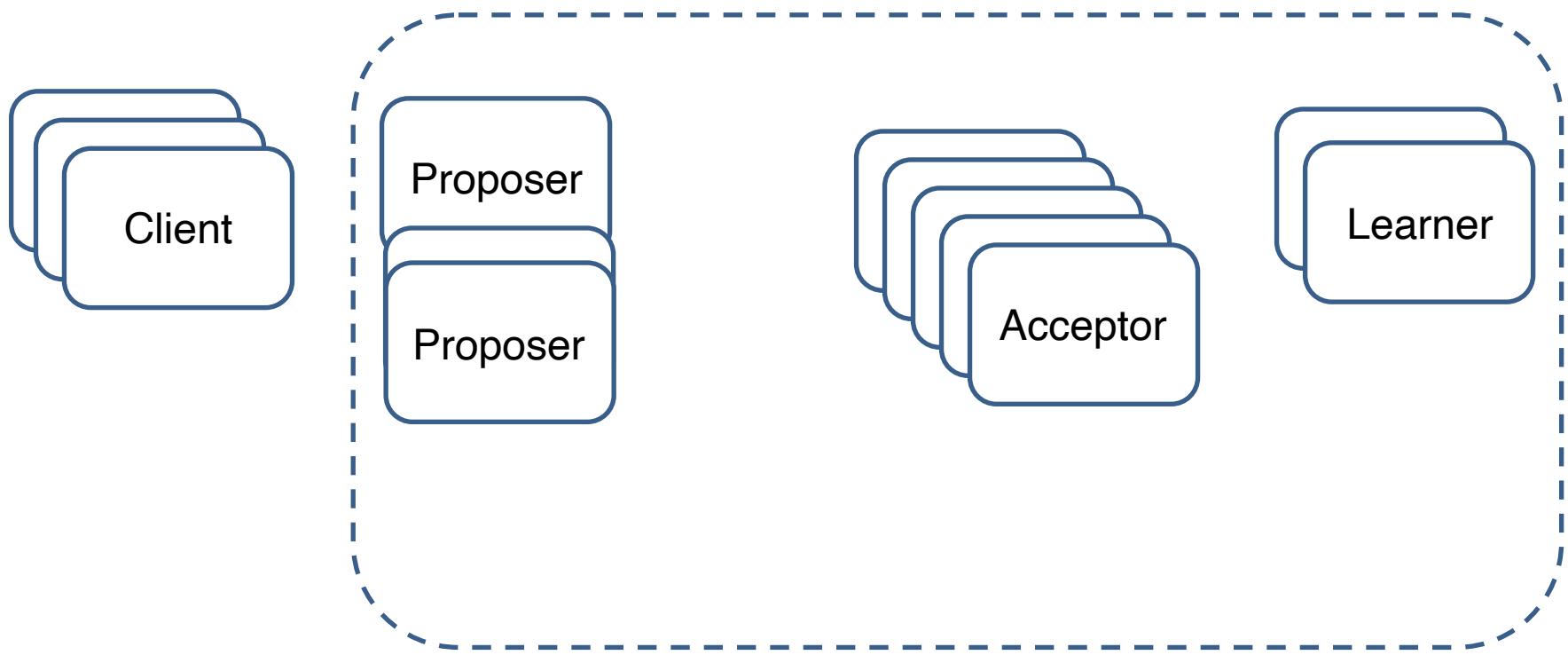
- **Messages** sent by learner
ClientRes (Value (V))
- A **learner** acts as a storage point for decisions made by acceptors
- Paxos converts a decision made by only a **quorum** of acceptors into a consensus for **learners** (for Paxos)
- Because the decision has already been made by acceptors, it is easier for all learners to agree



Safety and liveness properties

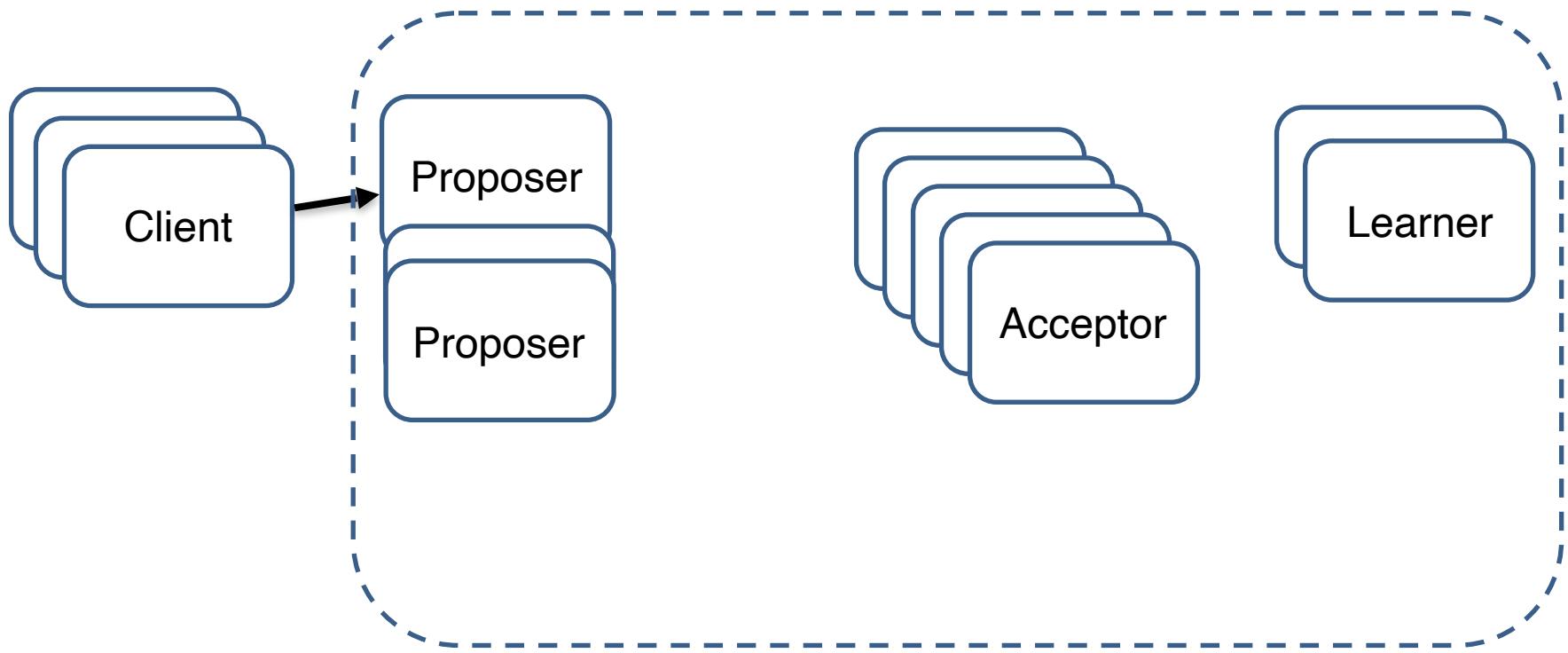
- **Non-triviality:** Only proposed values can be learned
- **Safety:** At most one value can be learned (i.e., two different **learners** cannot learn different values)
- **Liveness:** If value V has been proposed, then eventually **learner L** will learn *some* value (if sufficient processes remain non-faulty)
- Paxos **ensures safety** property holds, regardless of the pattern of failures (subject to $2f+1$ requirement)
- Paxos may **get stuck in a livelock** (then, no progress; **liveness property is not guaranteed**)

Summary of Paxos roles



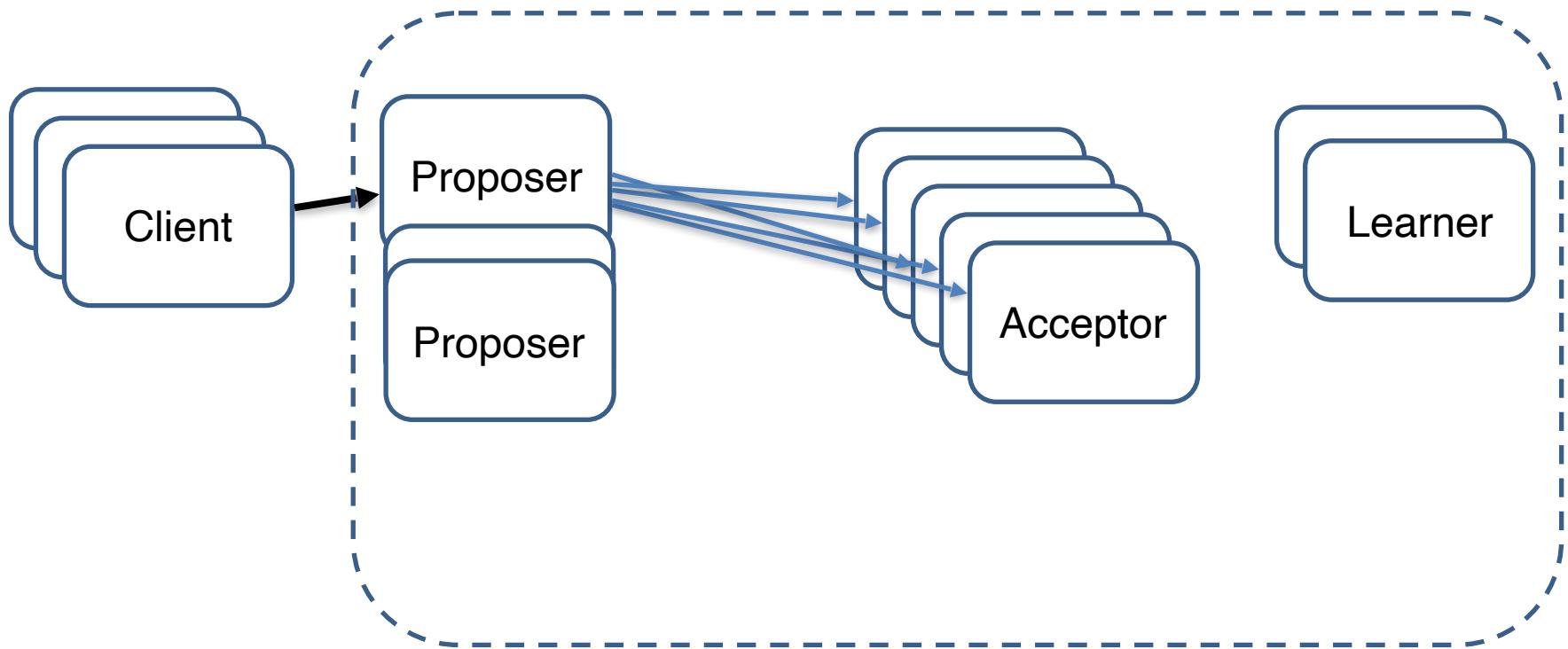
Summary of Paxos roles

1. clientReq (Value V)



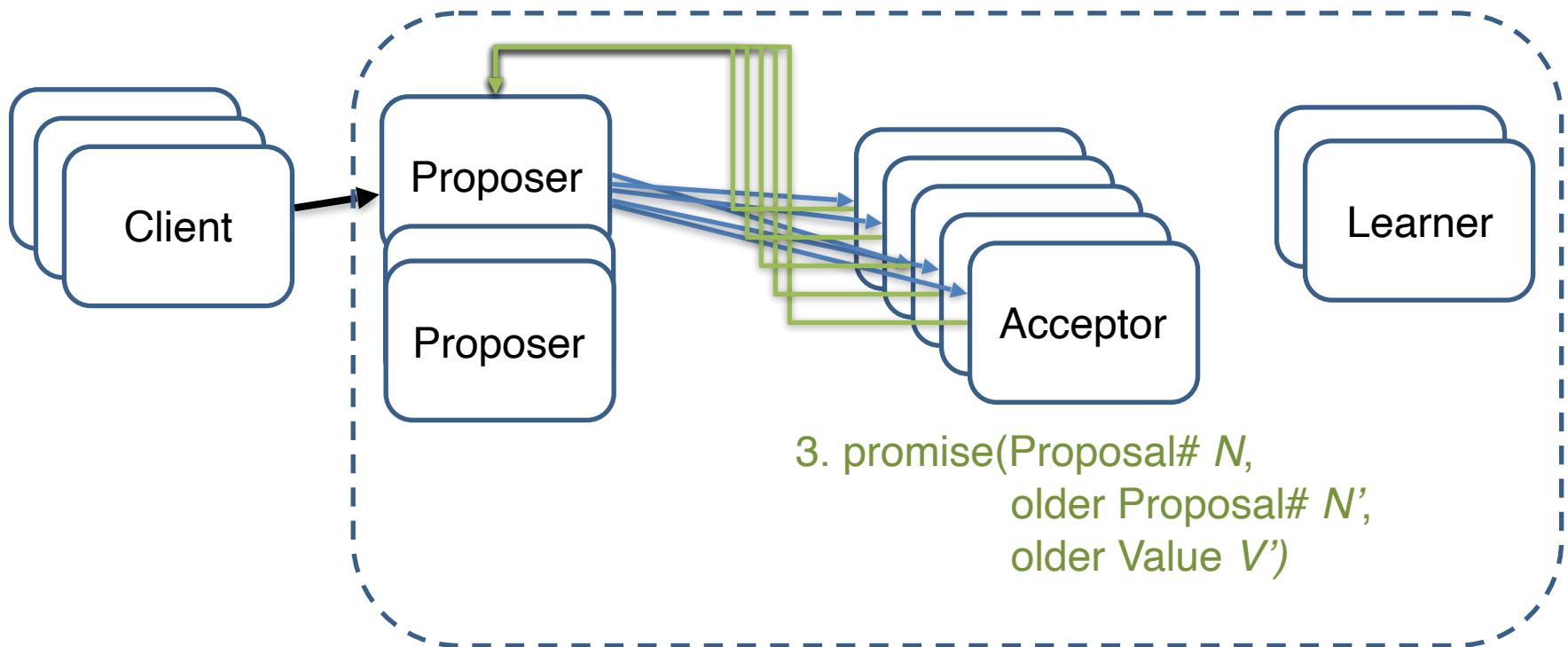
Summary of Paxos roles

1. clientReq (Value V)
2. prepare (Proposal# N)



Summary of Paxos roles

1. clientReq (Value V)
2. prepare (Proposal# N)

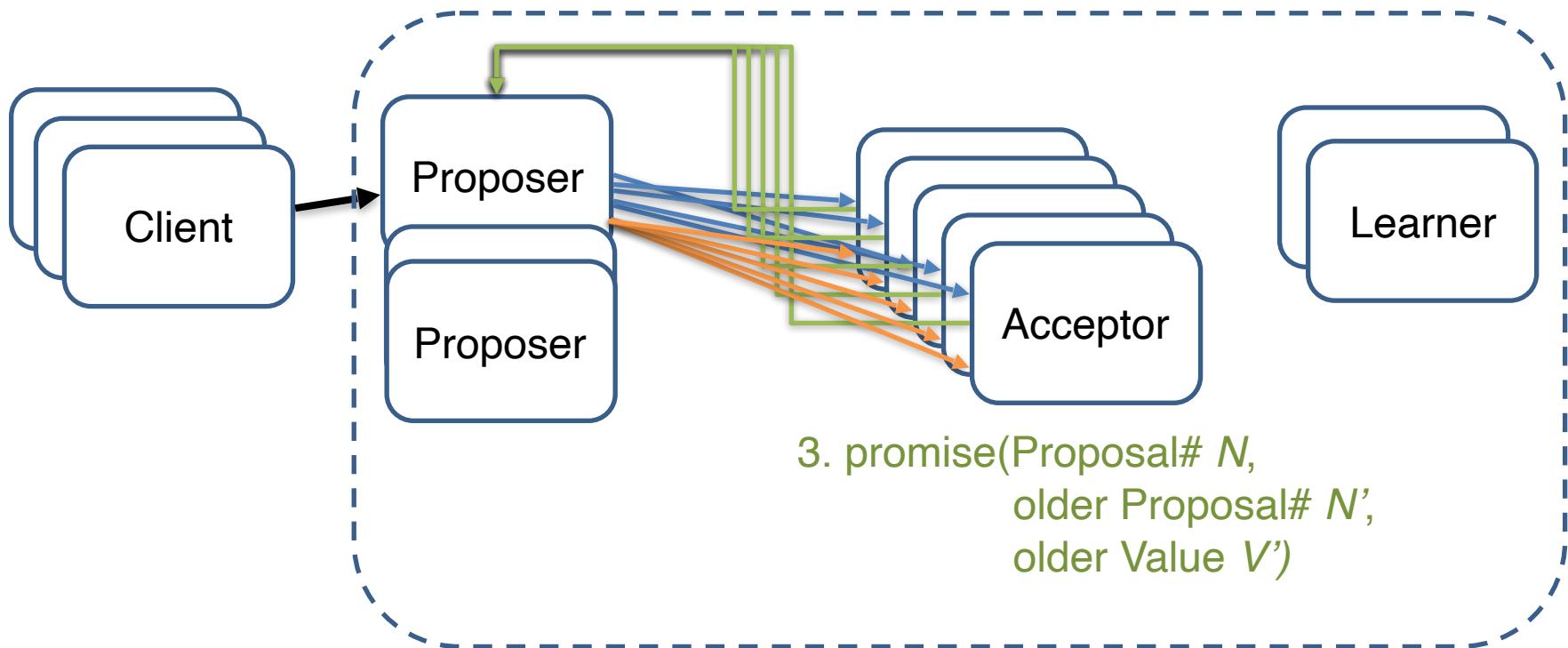


Summary of Paxos roles

1. clientReq (Value V)

2. prepare (Proposal# N)

4. acceptReq (N, V)

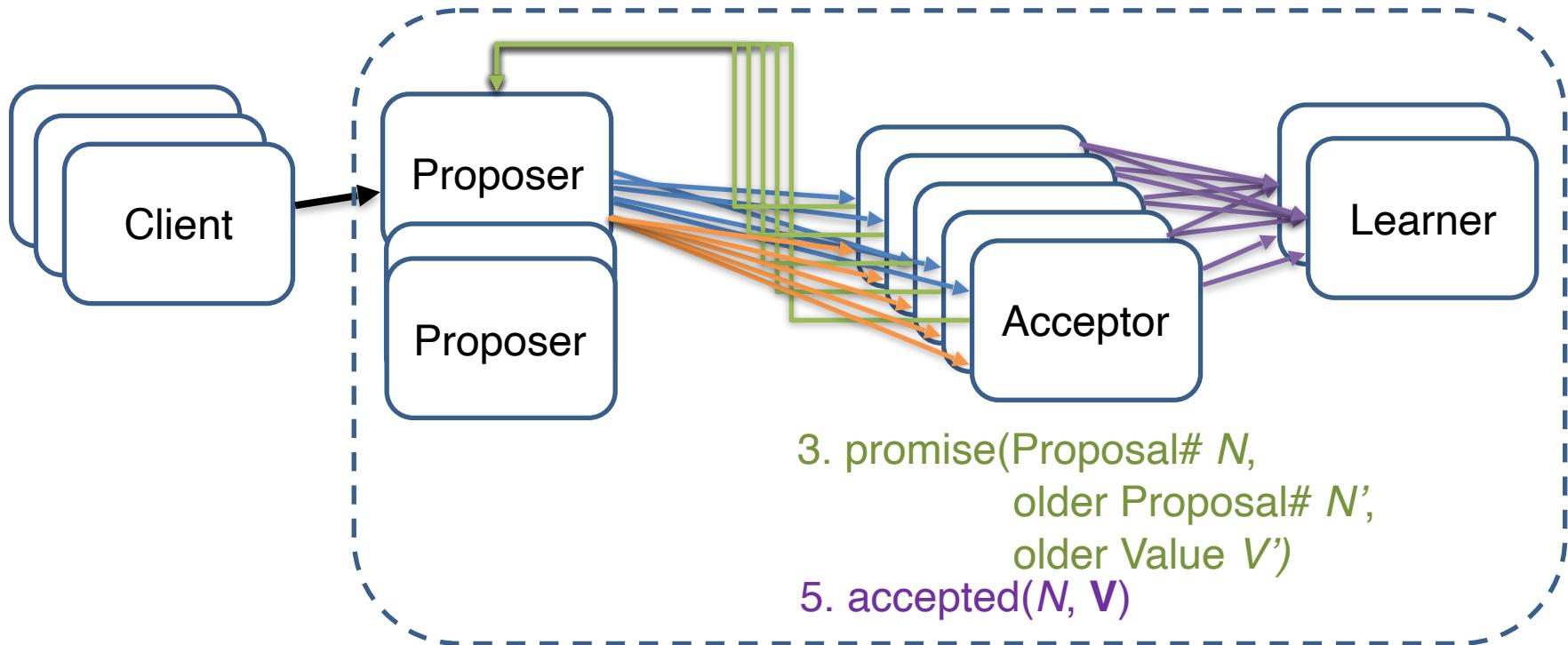


Summary of Paxos roles

1. clientReq (Value V)

2. prepare (Proposal# N)

4. acceptReq (N, V)



3. promise(Proposal# N ,
older Proposal# N' ,
older Value V')

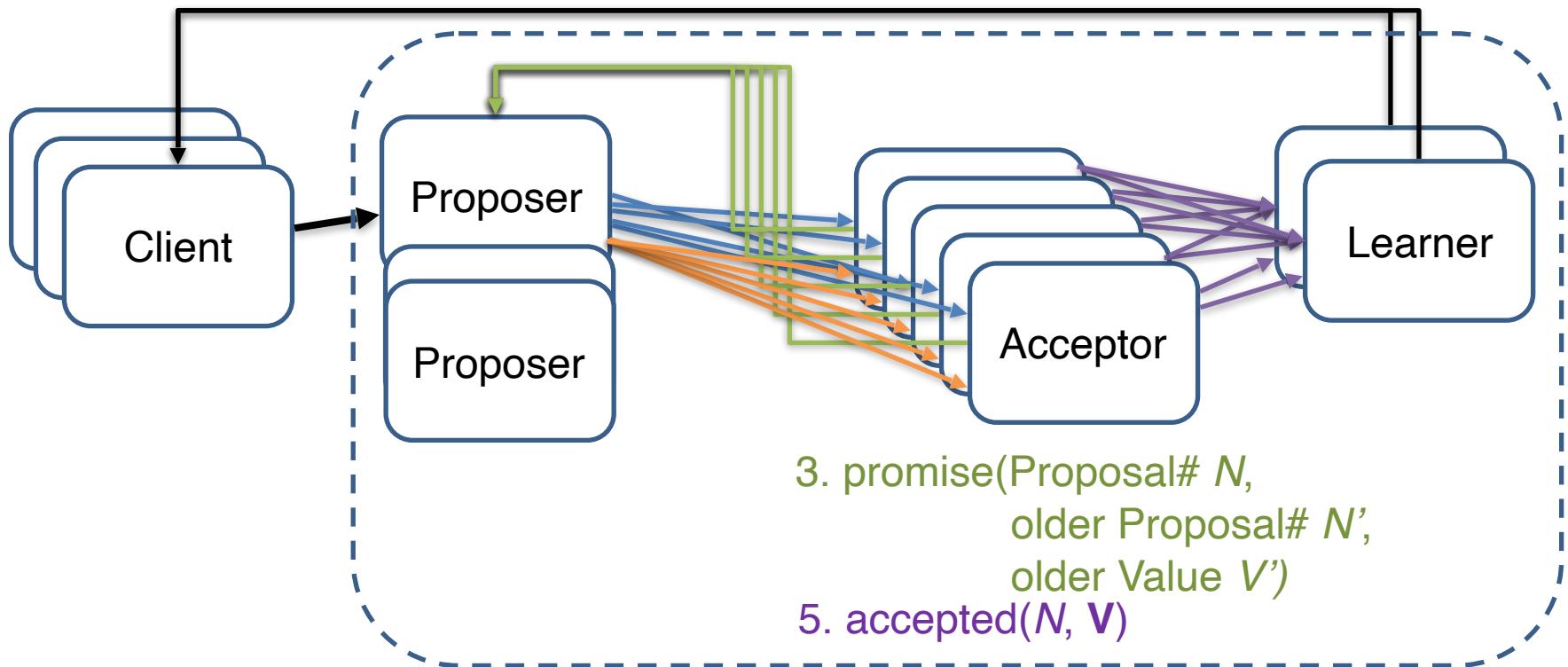
5. accepted(N, V)

Summary of Paxos roles

1. clientReq (Value V)

4. acceptReq (N, V)

6. clientRes (Decided Value V)



Basic Paxos

- Each protocol invocation **instance** decides on a **single output value**
- Protocol may **proceed** over several **rounds**
- A successful round has **two phases**
 - prepare (P) and promise (A) (first phase)
 - acceptRequest (P) and accepted (A) (second phase)
- Once learner collects a majority of accepts for a value it issues clientResponse (L) (“third phase”)
- **Proposer** needs to communicate with at least a **quorum of acceptors**

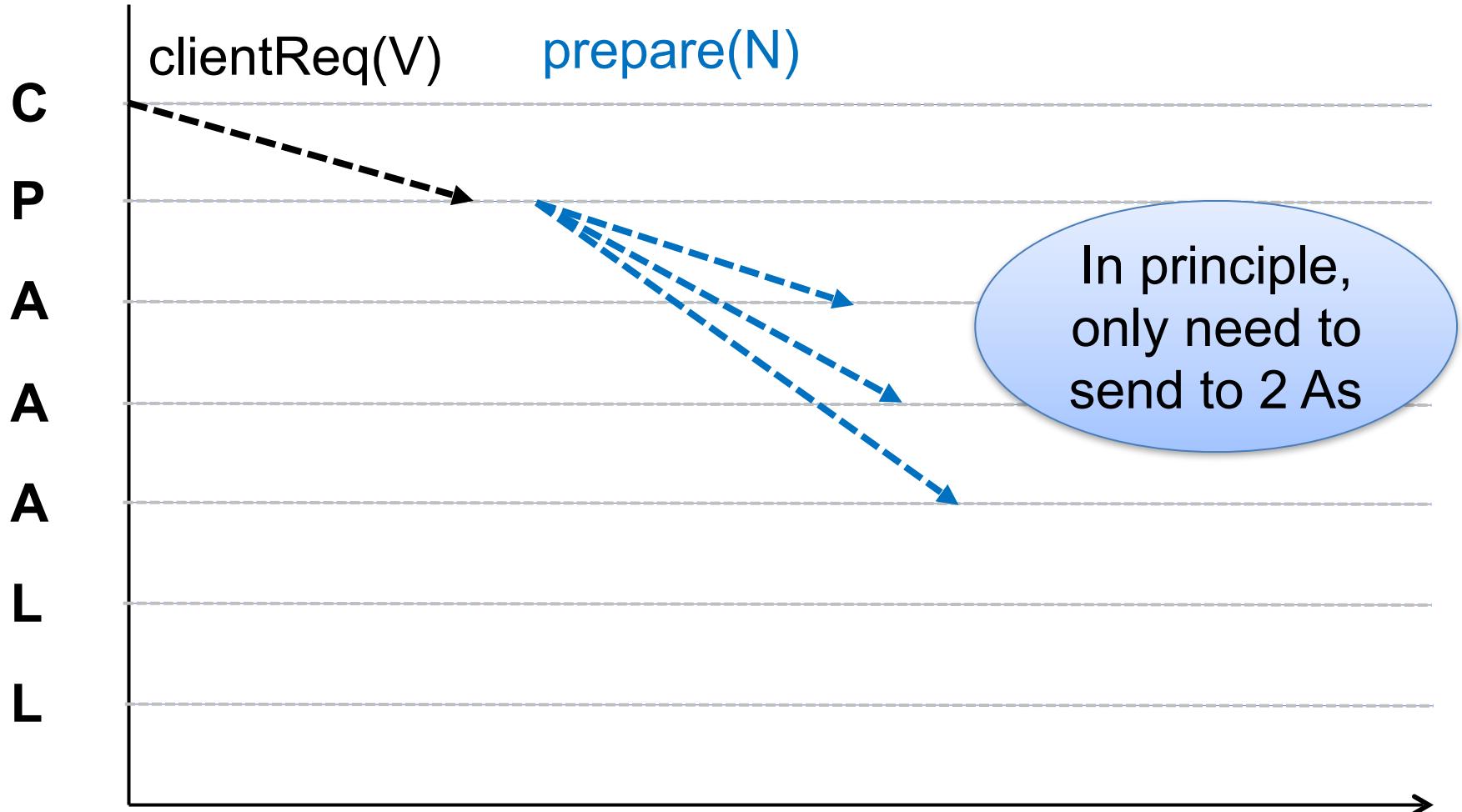
Phase 1a: Prepare (@ Proposers)

- **Proposer** creates proposal identified by P# N
- Any proposer could send proposals at any time
- P# N must be **greater than any previous P# N'** by this proposer
- **Proposer** sends a **prepare message** with P# N to quorum of **acceptors**:
prepare(N)
- **Proposer** decides who (among the acceptors) is in the quorum
(could be every acceptor)

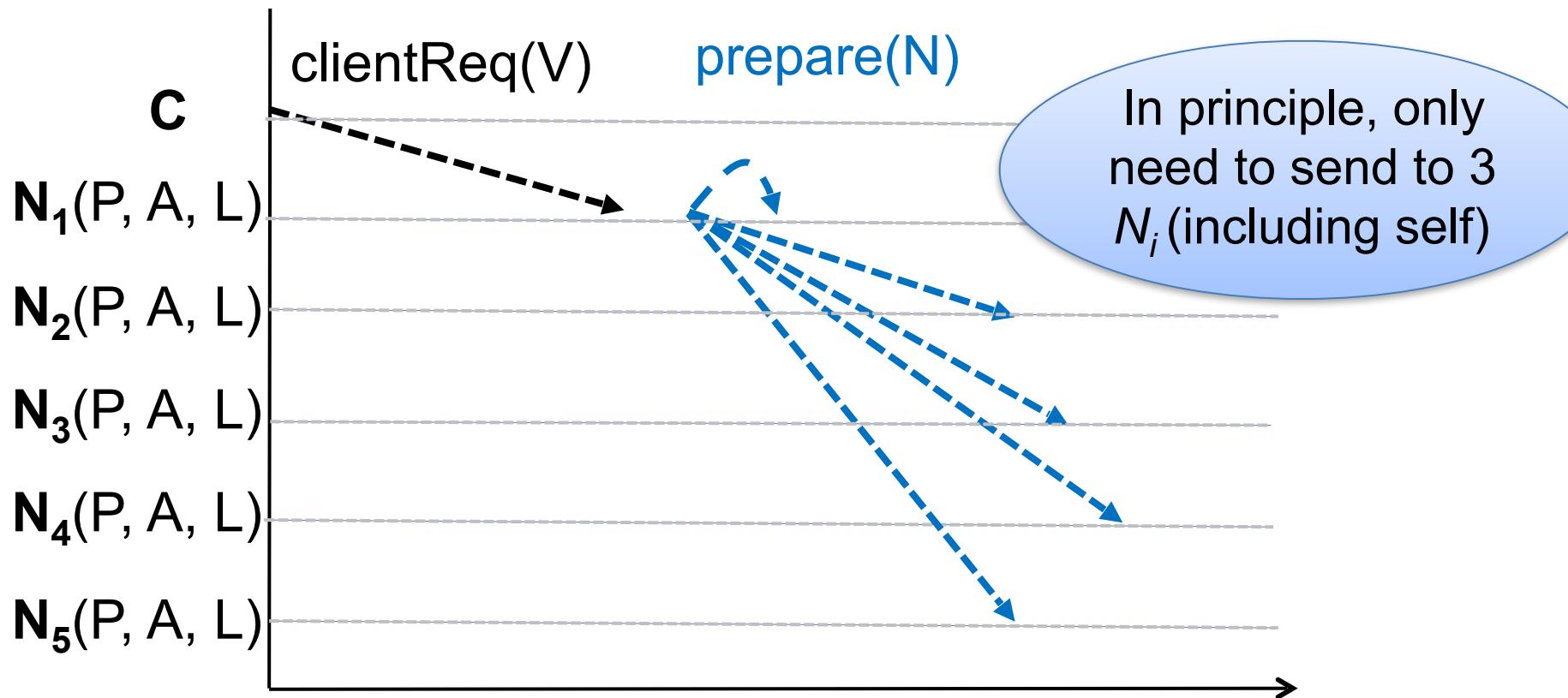


I'm not sure we have a quorum now that half our Board has been arrested.

Phase 1a: Prepare



Phase 1a: Prepare (Roles coalesced)



Phase 1b: Promise (@ Acceptors)

Received: `prepare(N)`

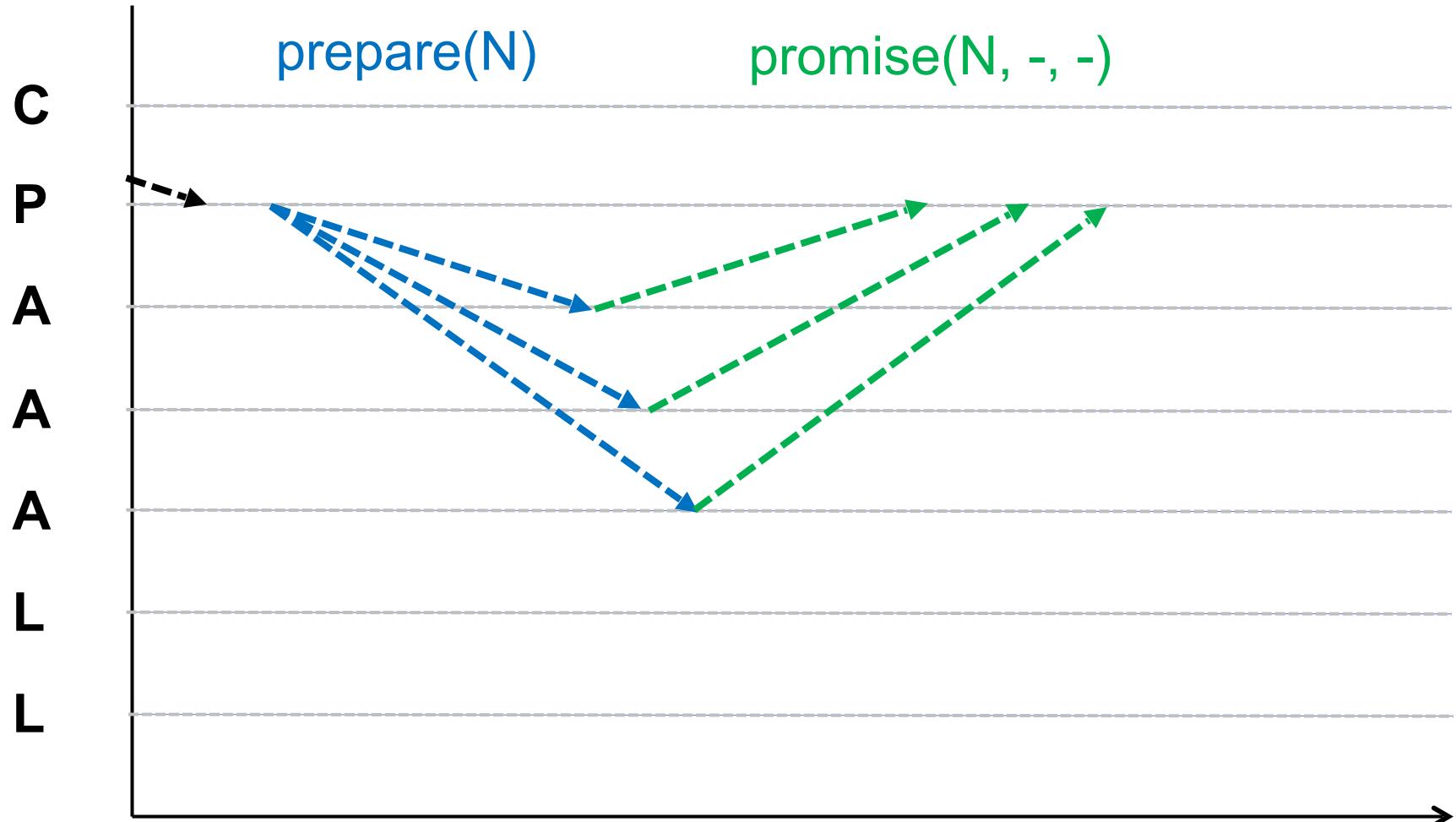
- If N is larger than any previous N' received from **any proposer** by this acceptor, then the acceptor must return a **promise to ignore all future proposals having a number less than N**
`promise(N, -, -)`
- If the **acceptor** accepted a proposal at some point in the past, it must include the previous N' and previous value V' in its response to the **proposer**
`promise(N, N', V')`
- Otherwise, the **acceptor** sends a negative reply (NACK)

Optional

Phase 1b: Promise (@ Acceptors)

```
// N larger than any P# previously received
IF N > max_PN
    // Yes, remember P#, reply with promise
    max_PN = N
    reply: promise(N, -, -)
ELSE
    // No
    do not reply (or reply with a NACK)
```

Phase 1b: Prepare



Phase 1b: Promise (@ Acceptors)

Acceptor accepted proposal in the past

- If N is larger than any previous N' received from **any proposer** by this acceptor, then the acceptor must return a **promise to ignore all future proposals having a number less than N**
promise(N , -, -)
- If the **acceptor** accepted a proposal at some point in the past, it must include the previous N' and previous value V' in its response to the **proposer**
promise(N , N' , V')
- Otherwise, the **acceptor** sends a negative reply (NACK)

Phase 1b: Promise (@ Acceptors)

Acceptor accepted proposal in the past

```
// N larger than any P# previously received
// N > max_PN
IF N ≤ max_PN
    // !(N > max_PN)
    do not reply (or reply with a NACK)
ELSE
    // (N > max_PN), was proposal accepted before?
    IF (proposal_accepted == true)
        reply: promise(N, N', V')
    ELSE
        reply: promise(N, -, -)
```

Phase 2a: Accept Request (@Proposer)

- If a **proposer** receives **enough promises** from **acceptors** (a quorum), it needs to **set a value** to its **proposal**
- If any acceptor had previously accepted a proposal, then it would have sent its values to the **proposer**, who now must **set the value** of its **proposal** to the value associated with the **highest proposal number** reported by any **acceptor**

Phase 2a: Accept Request (@Proposer)

cont.'d.

- If none of the **acceptors** had accepted a proposal up to this point, then the **proposer** may **choose any value** for its proposal (i.e., the one from the client request)
- **Proposer** sends an **accept request message** to a **quorum of acceptors** with the chosen V (value) for its proposal

acceptReq(N, V)

- **Proposer** may choose **quorum** from all acceptors in the system, not just ones that promised back

Phase 2a: Accept Request (@Proposer)

cont.'d.

Were promises from a majority of acceptors received?

IF yes

Did any reply contain accepted values
(from other proposals) ?

IF yes

//Take value from promise with highest PN
V = **accepted_VALUE**

ELSE

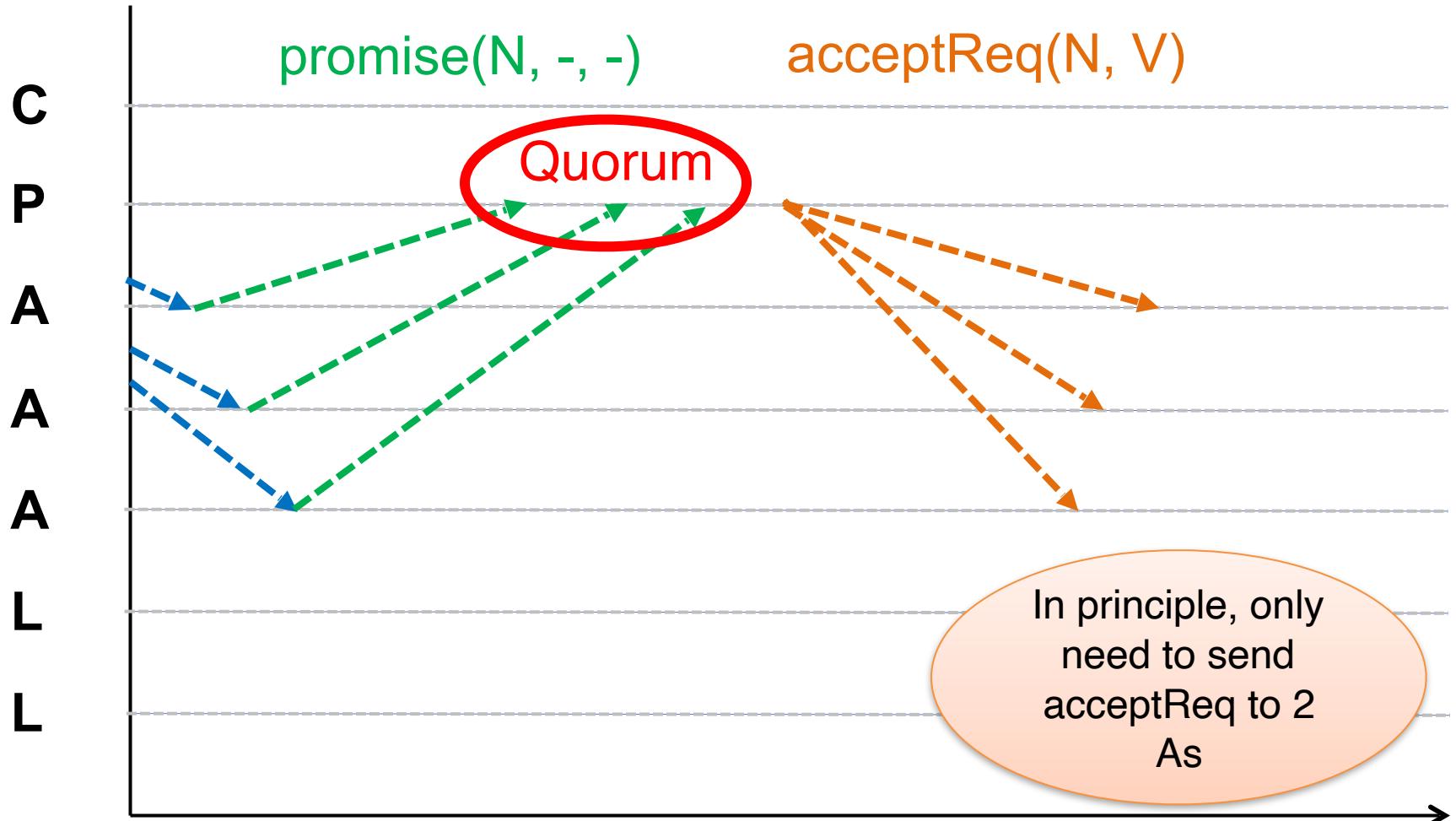
// Use own value (client's input)

V = **OUR_VALUE**

reply **acceptReq(N, V)** to majority of acceptors
(or more)

Phase 2a: Accept Request

Quorum size is 2



Phase 2b: Accepted (@Acceptor)

(Accept Request – acceptReq)

- If an **acceptor** receives an **acceptReq message** for a proposal number N , it must accept it **if and only if** it has not already promised to only consider proposals having a proposal number greater than N
- In this case, it registers the corresponding value V and sends an **accepted message** to the **proposer** and **every learner**
accepted(N, V)
- Otherwise, it **ignores** the **acceptReq** and sends a **NACK** to the proposer

Phase 2b: Accepted (@Acceptor)

(Accept Request – acceptReq)

- If an **acceptor** receives an **acceptReq message** for a proposal number N , it must accept it **if and only if** it has not already promised to only consider proposals having a proposal number greater than N
- In this case, it registers the corresponding value V and sends an **accepted message** to the **proposer** and **every learner**
accepted(N, V)
- Otherwise, it **ignores** the **acceptReq** and sends a **NACK** to the proposer

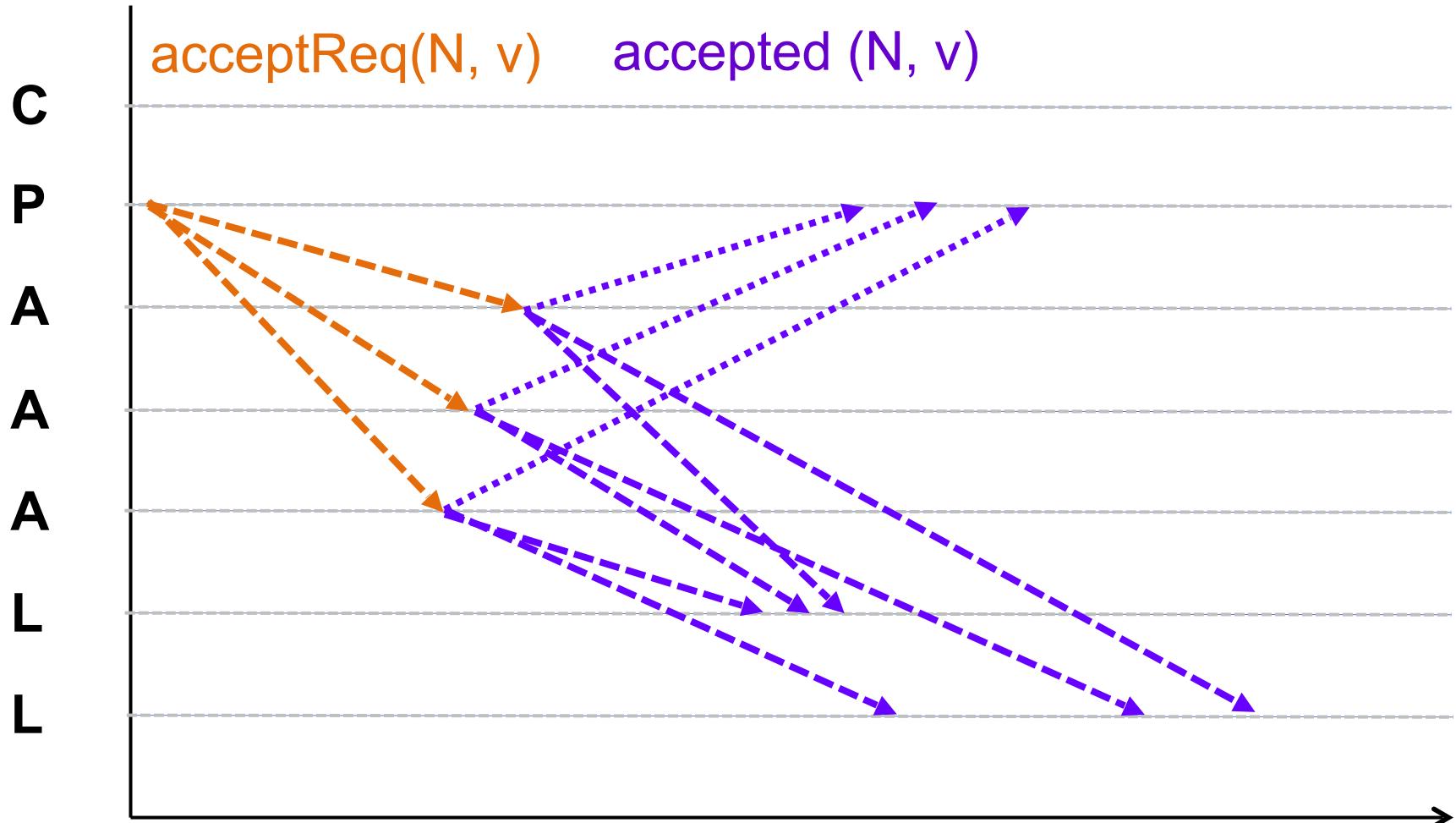
Optional

Phase 2b: Accepted (@Acceptor)

(Accept Request – acceptReq)

```
// Is P# N the largest seen so far?  
IF (N == max_PN)  
    // note that we accepted a proposal  
    proposal_accepted = true  
    // save the accepted proposal number  
    accepted_ID = N  
    // save the accepted proposal value  
    accepted_VALUE = V  
    reply: accepted(N, V) to proposer & learners  
ELSE  
    do not respond (or respond with a NACK)
```

Phase 2b: Accepted



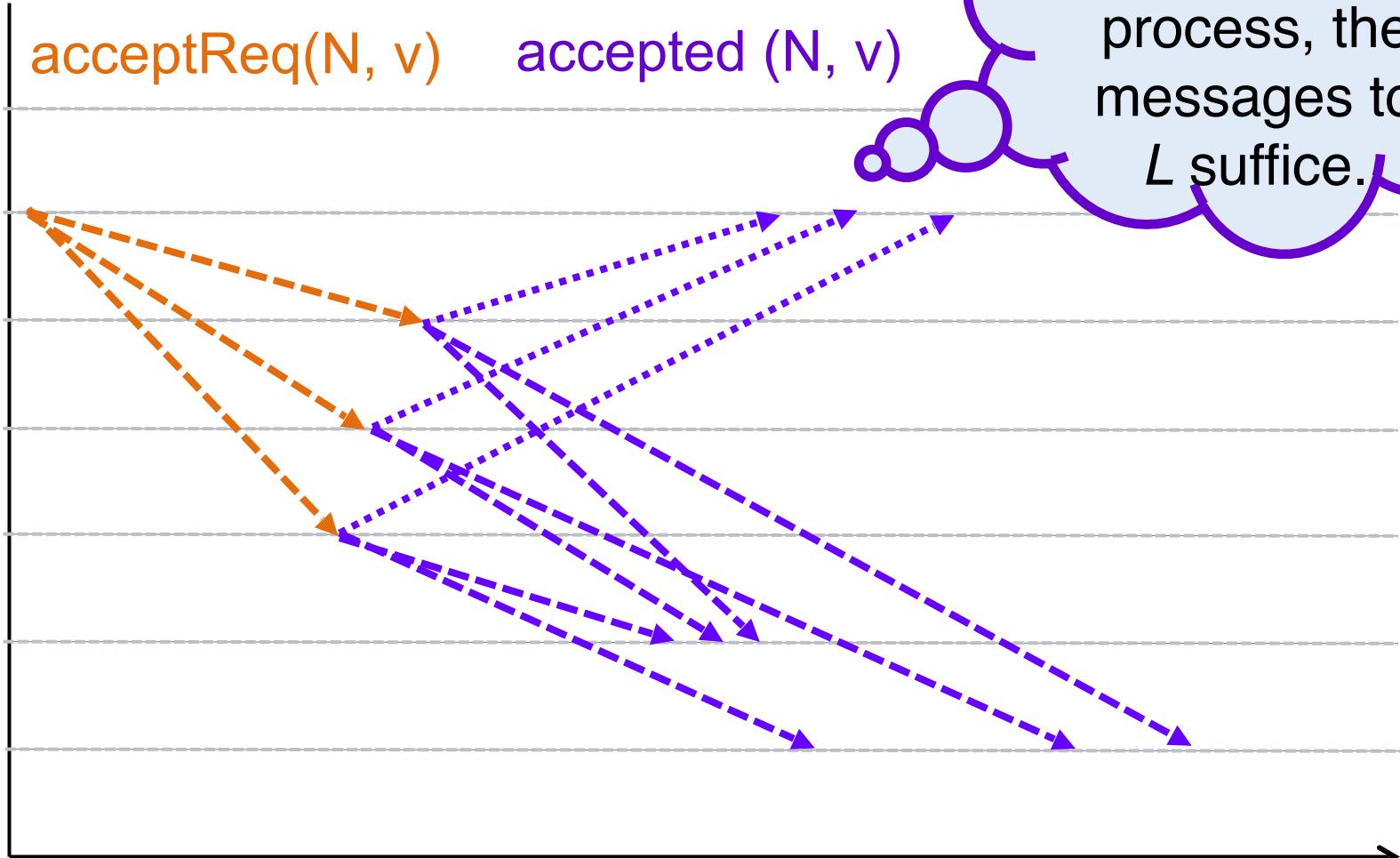
Phase 2b: Accept

C
P
A
A
L
L

acceptReq(N, v)

accepted (N, v)

If role P and L are held by the same process, the messages to L suffice.



Notes on Phase 2b

- An acceptor can accept **multiple proposals**
- **Paxos guarantees that acceptors ultimately agree on a single value**

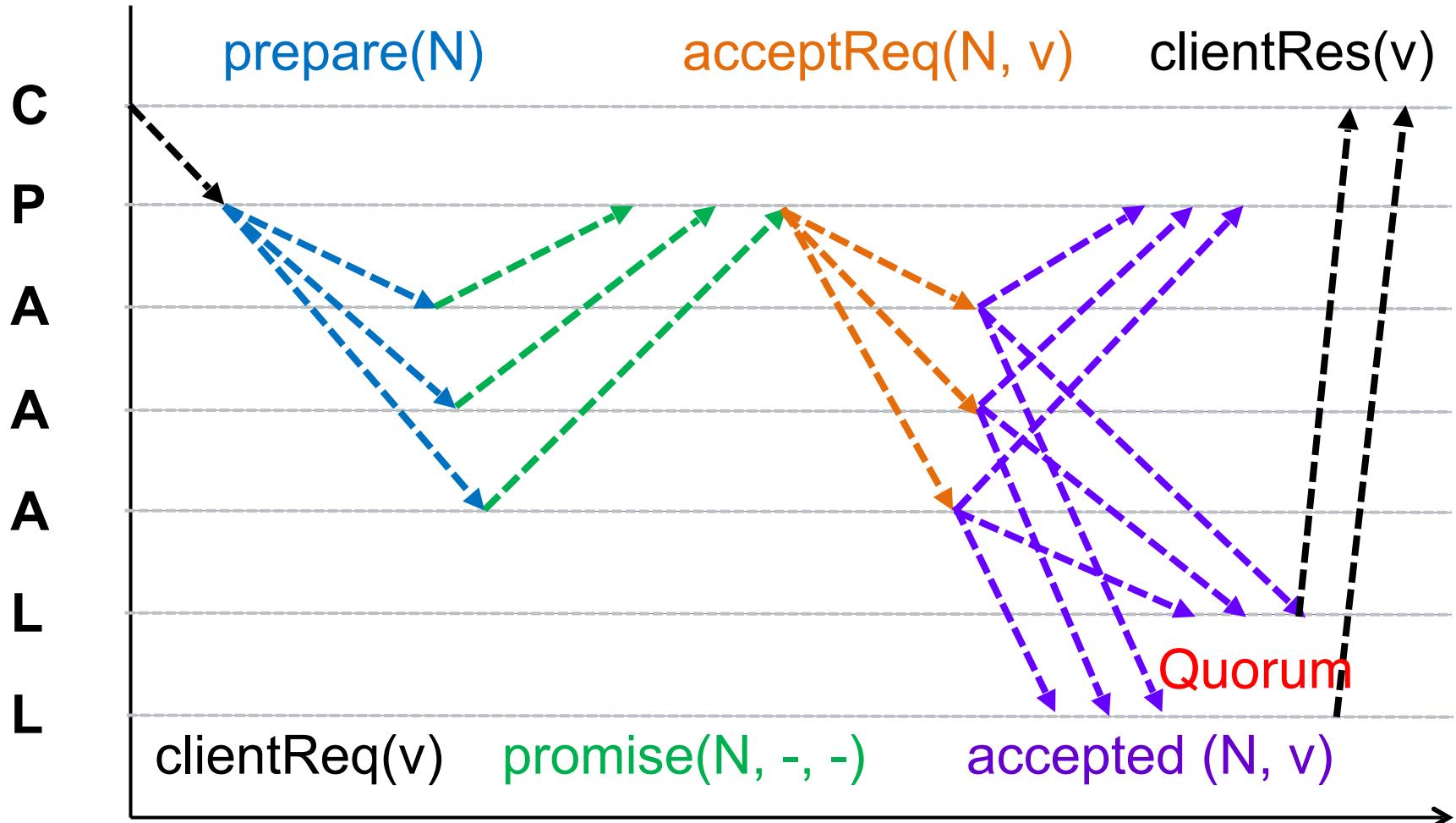
Notes on Phase 2b

- Rounds fail when **multiple** proposers send **conflicting prepare messages** which invalidate **acceptReq** messages
- Rounds fail when proposer **does not receive a quorum of replies** (promise or accepted messages)
- In these cases, **another round must be started** with a higher proposal number

Phase 3: Decided (@Learner)

- A learner **learns** a decided value if it receives accept messages from a **quorum of acceptors**
- A quorum is required to avoid conflicting accepts from multiple concurrent proposals
- Once a learner has **learned** a decided value, it can execute the associate command and/or communicate with a client

Basic Paxos protocol timeline



FAILURE HANDLING IN PAXOS

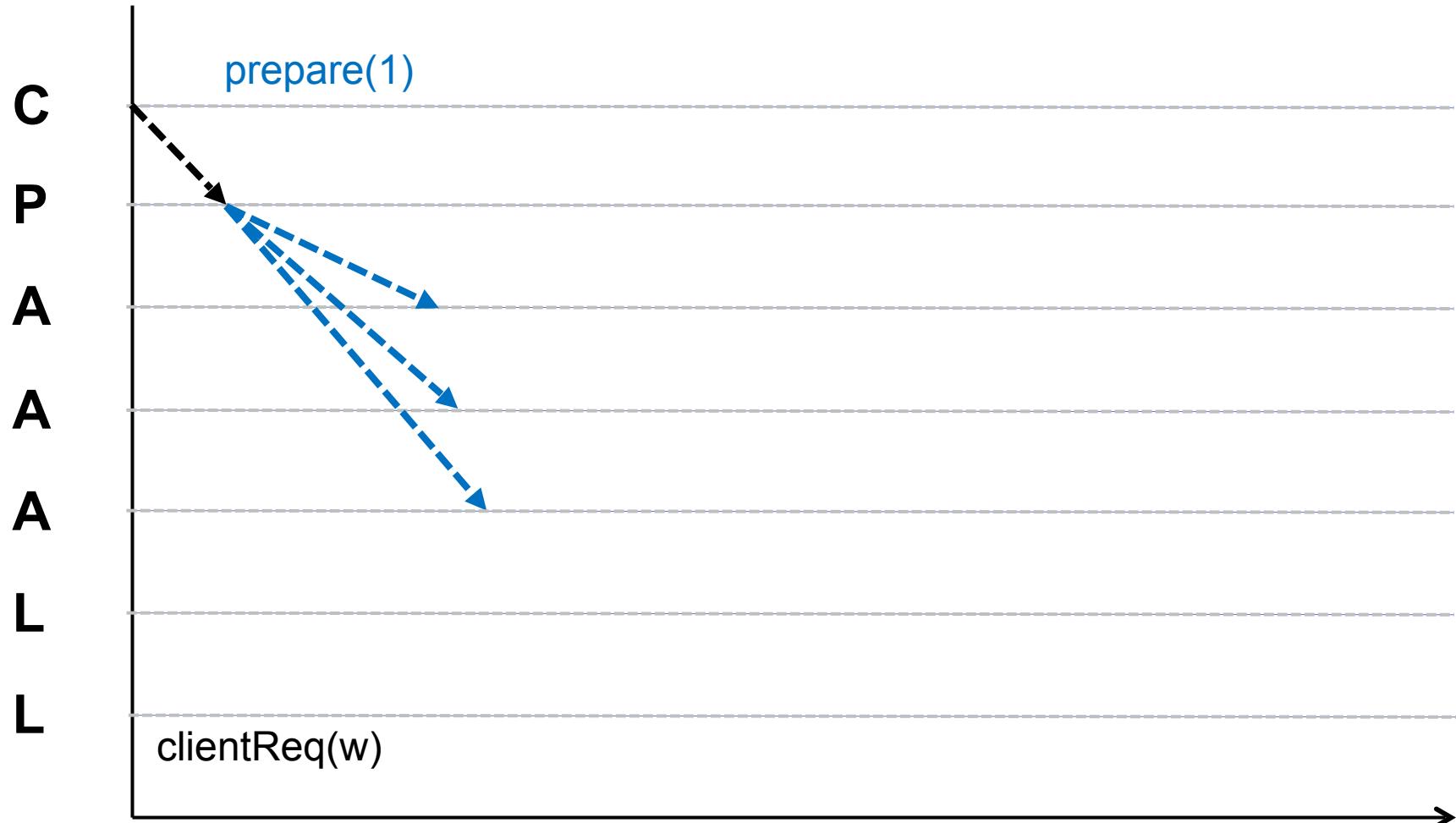
Part 4

Failure scenarios

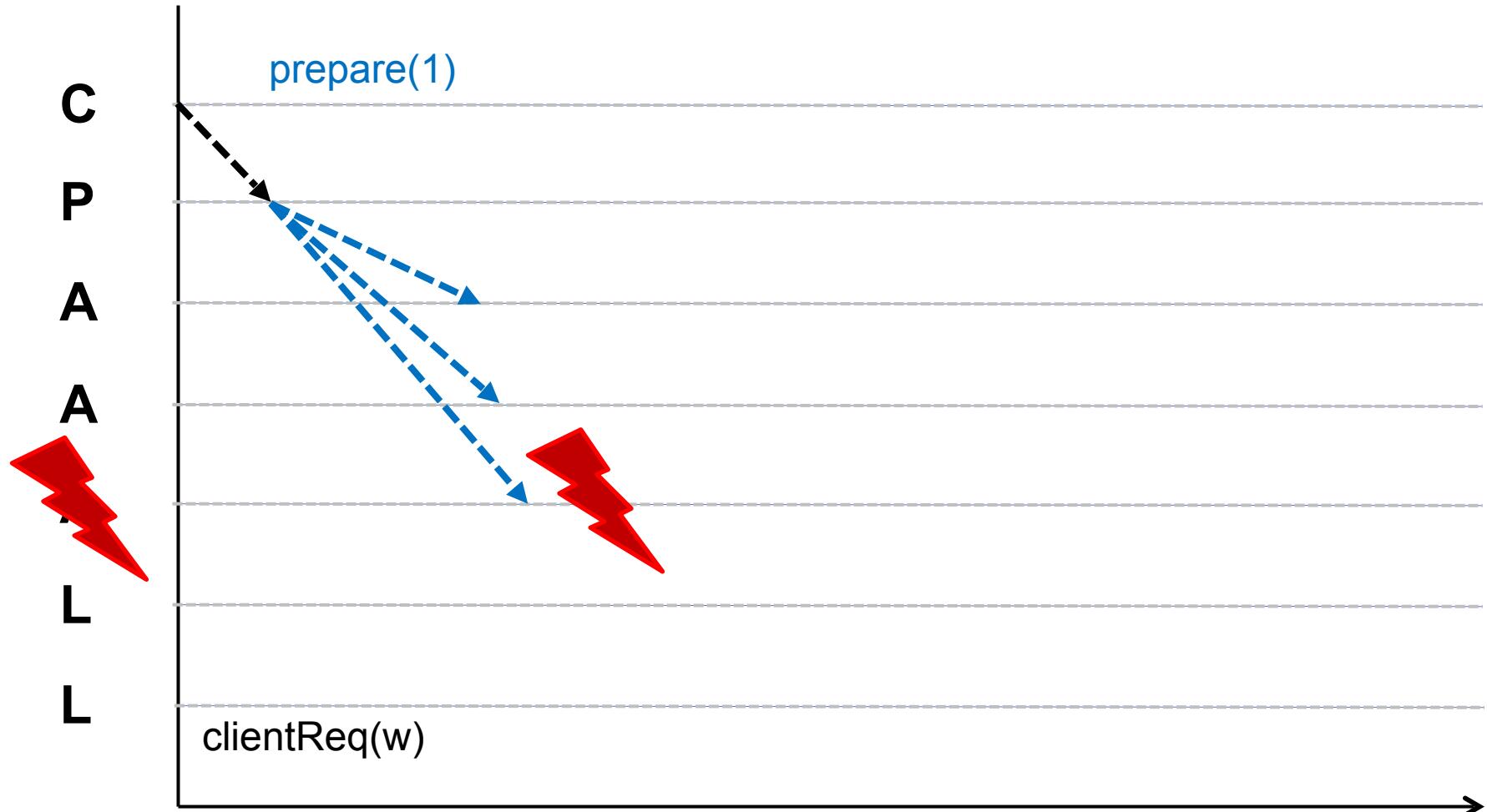
- Failure of **acceptor**
- Failure of redundant **learner**
- Failure of **proposer**
- Dueling **proposers**
- Communication failures



Basic Paxos: Failure of acceptor I

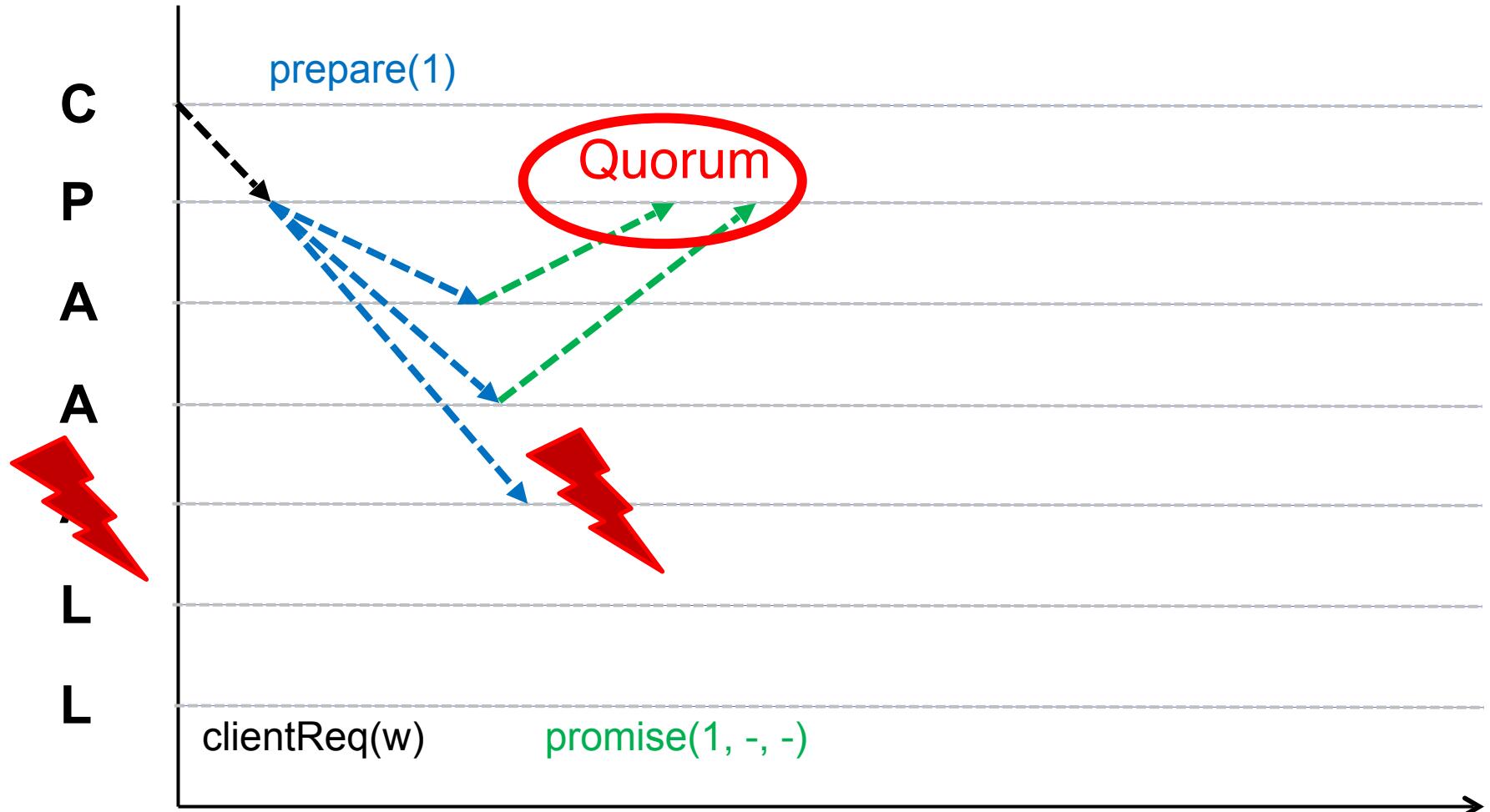


Basic Paxos: Failure of acceptor II



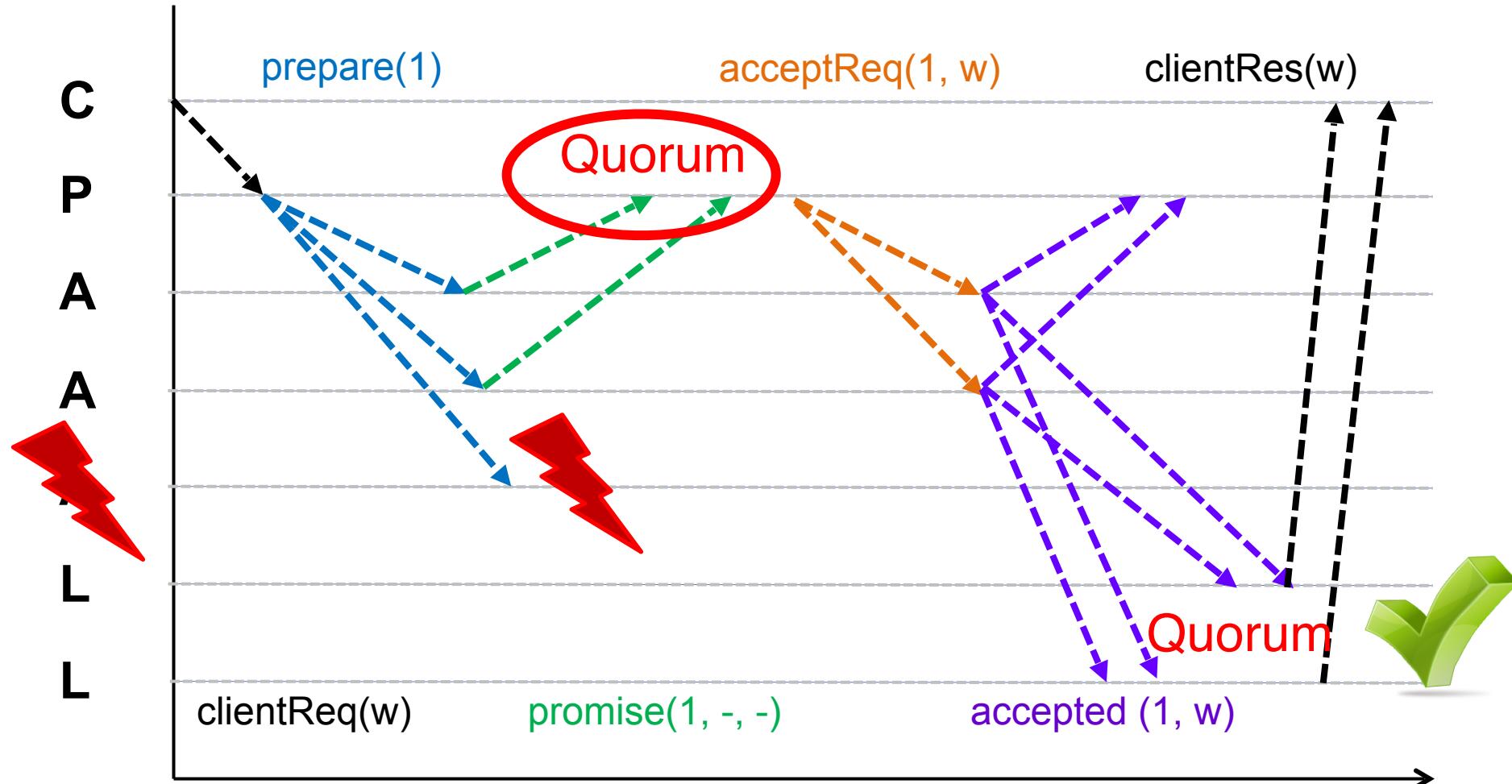
Basic Paxos: Failure of acceptor III

Quorum size is 2



Basic Paxos: Failure of acceptor IV

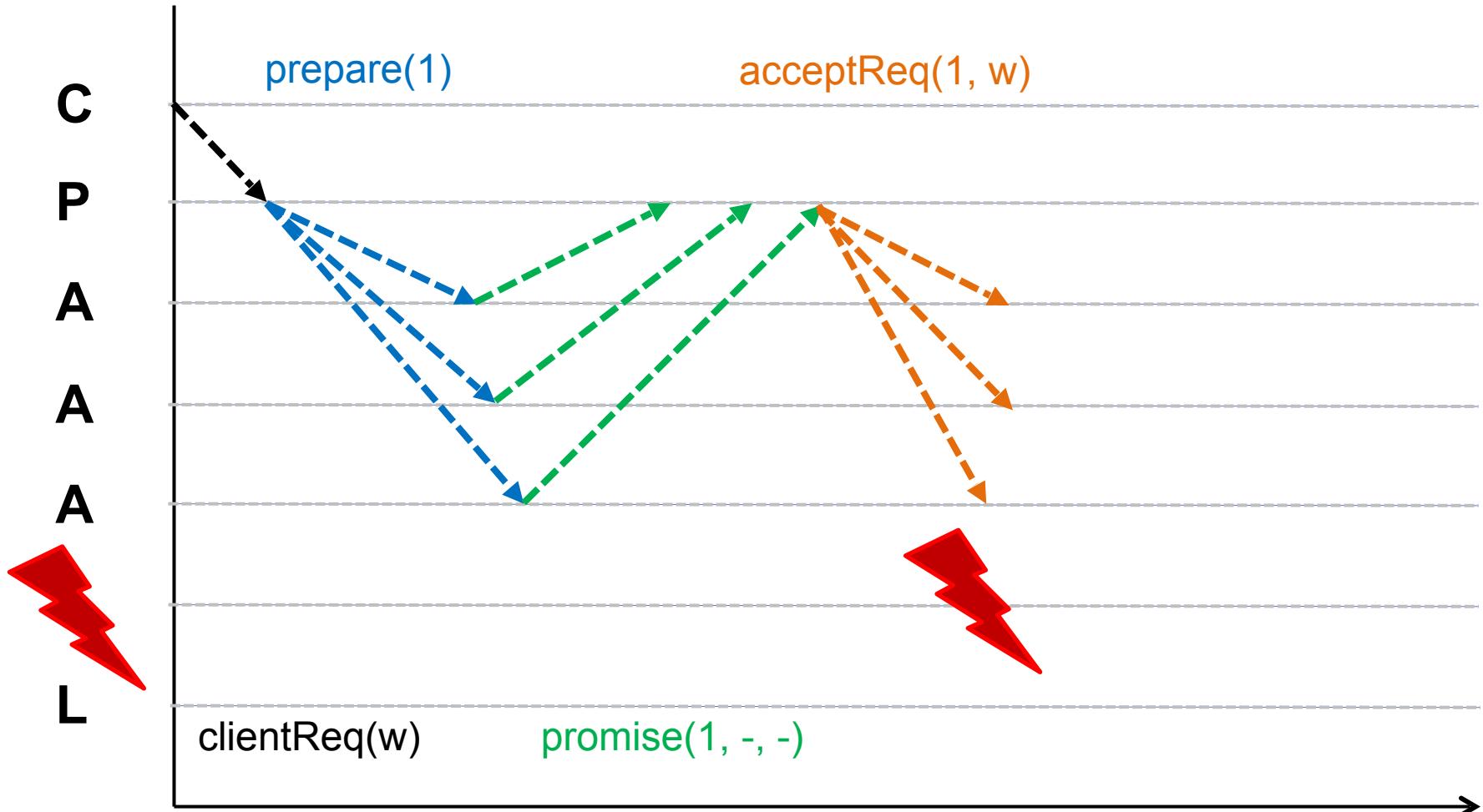
Quorum size is 2



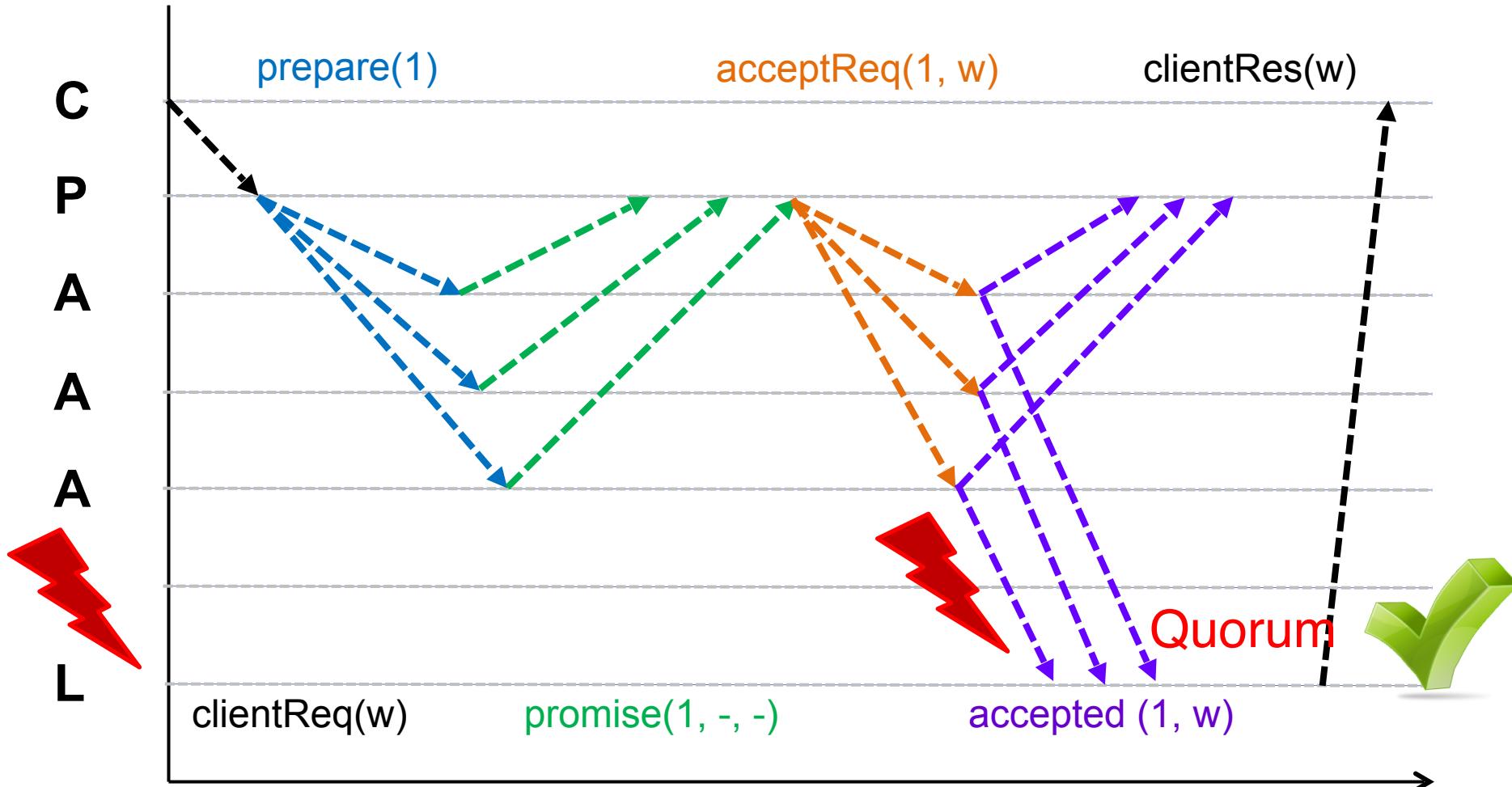
Basic Paxos: Failure of redundant learner



Failure of redundant learner I



Failure of redundant learner II

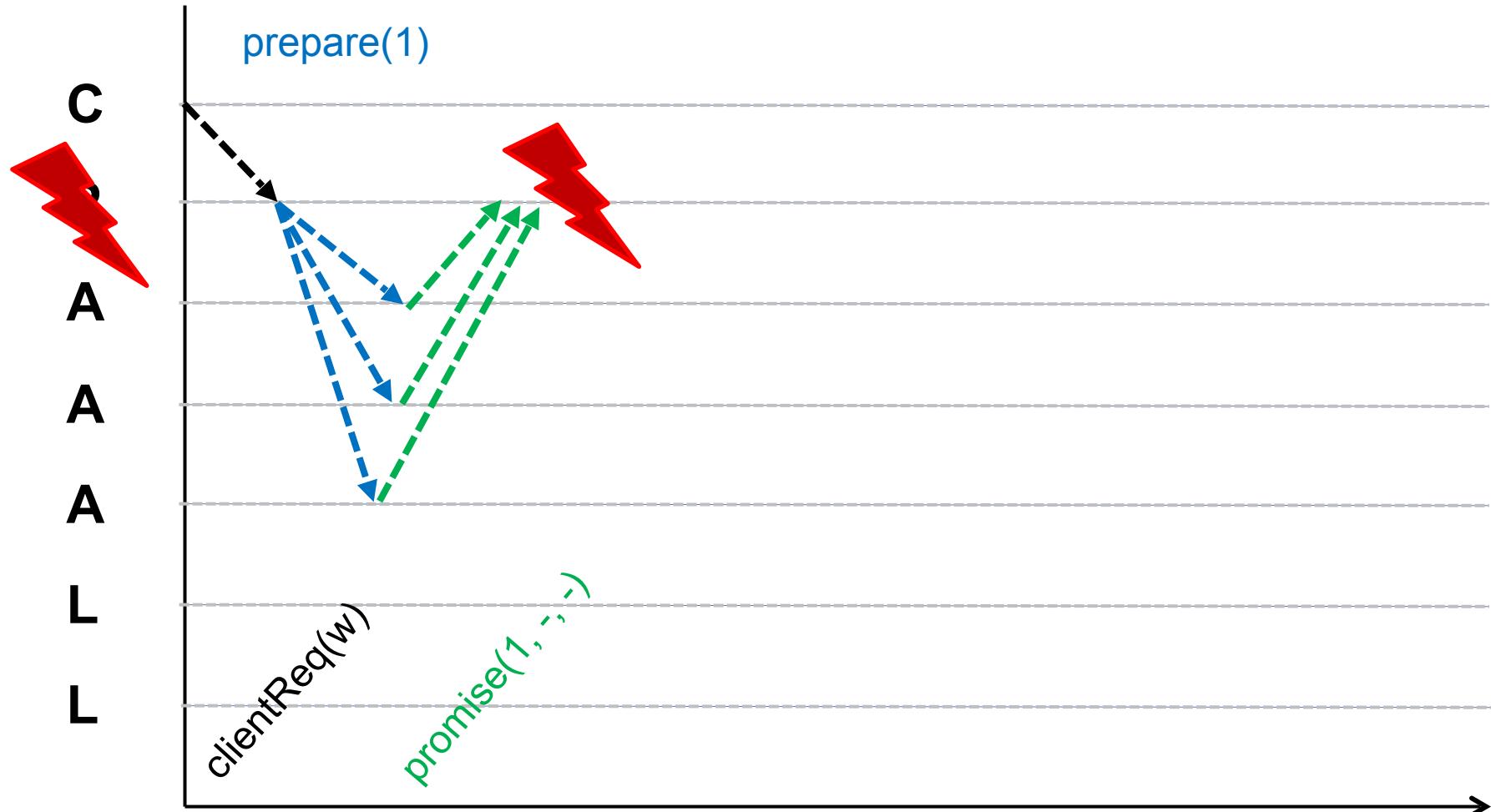


Basic Paxos: Failure of proposer

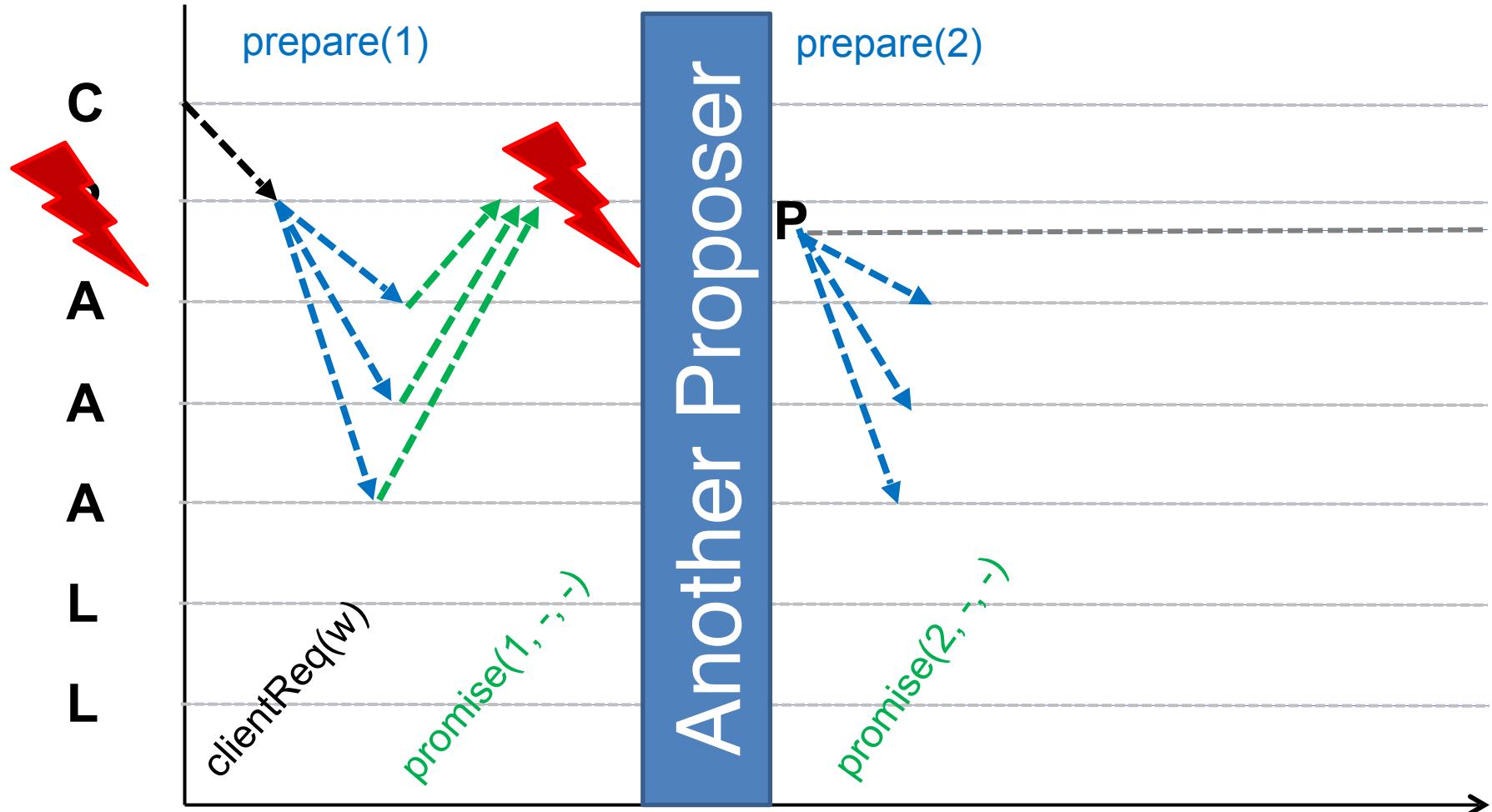


Basic Paxos: Failure of proposer I

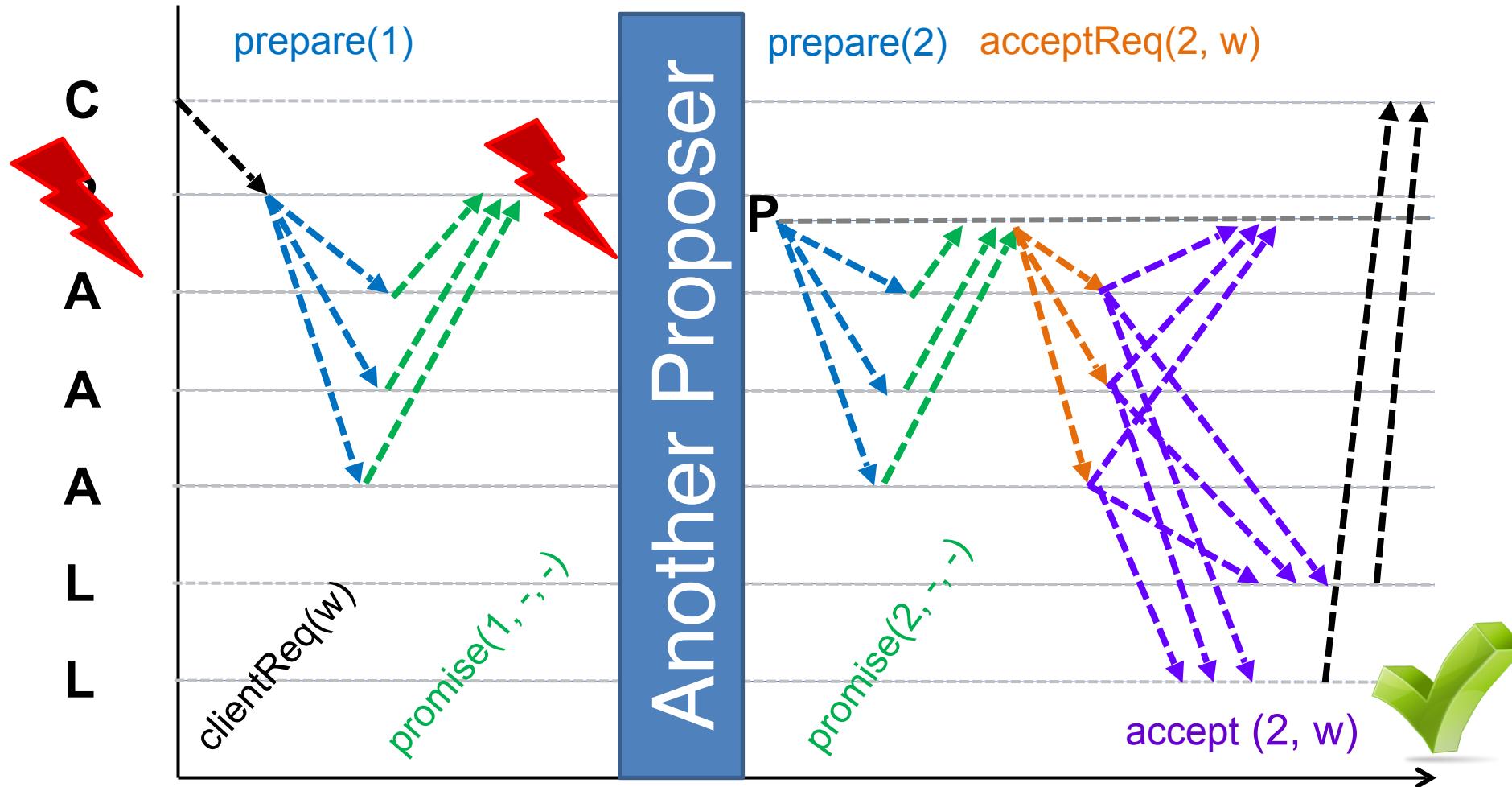
In prepare phase



Basic Paxos: Failure of proposer II



Basic Paxos: Failure of proposer III

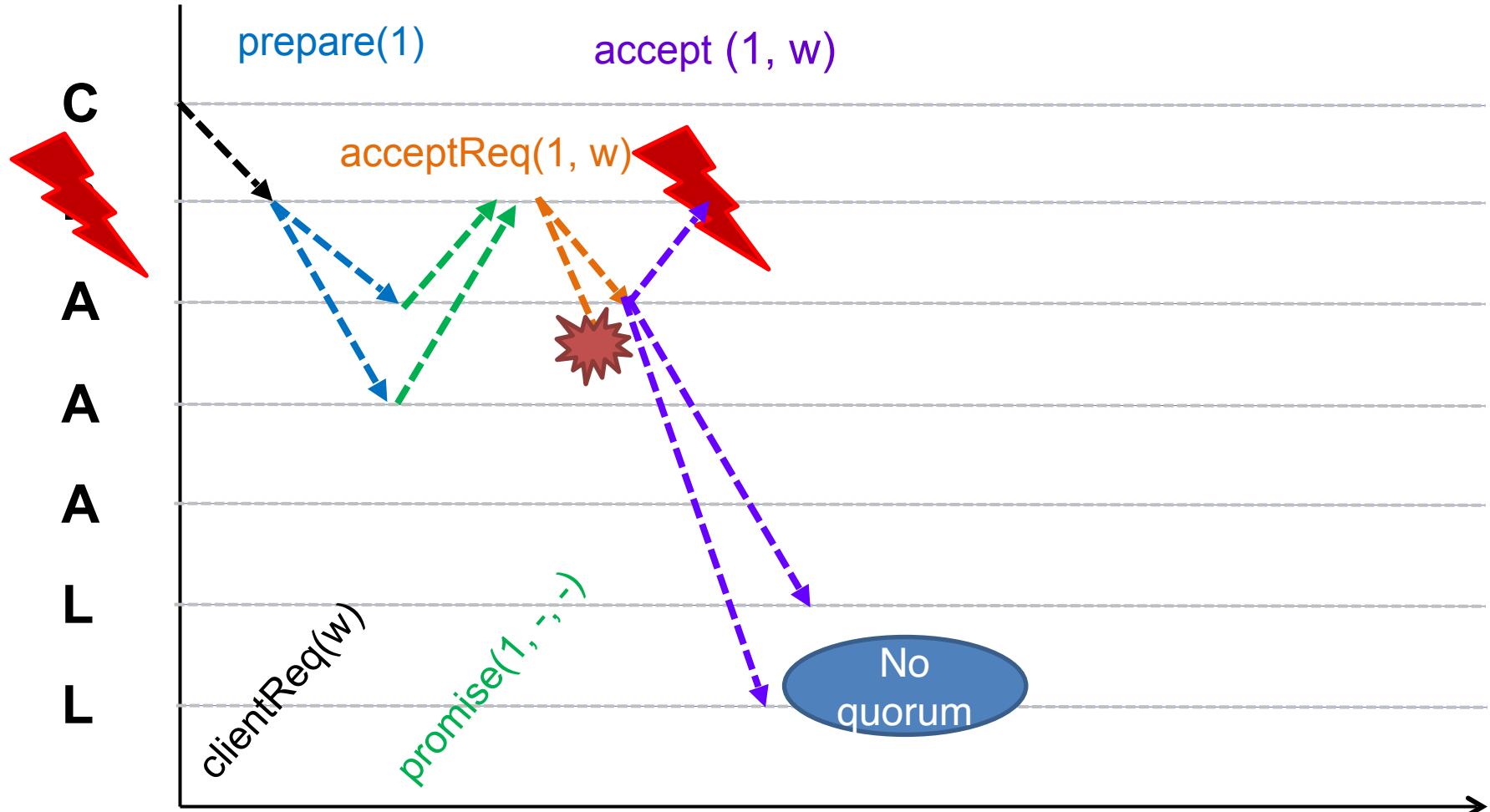


Basic Paxos: Failure between accepts

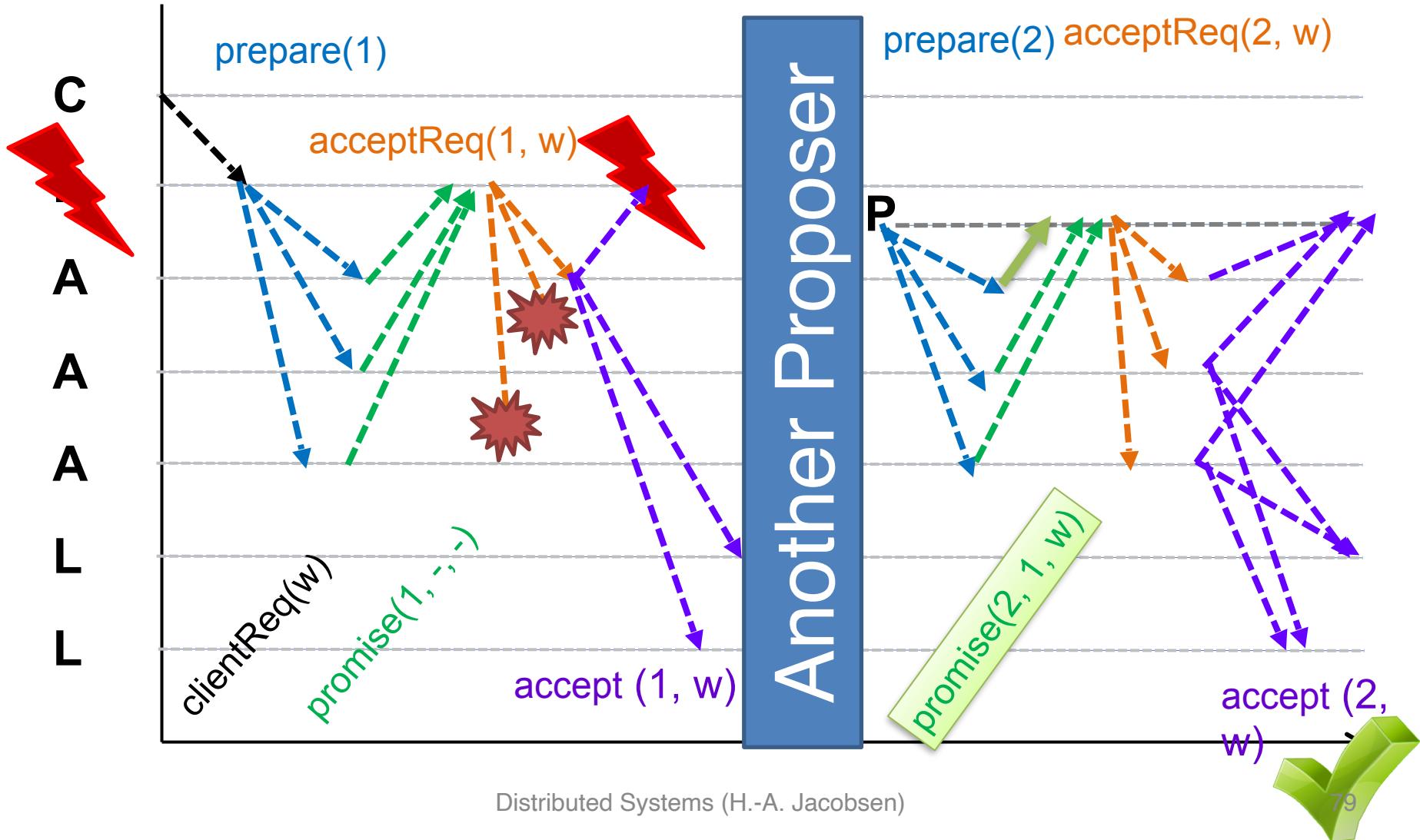
In accept phase



Basic Paxos: Failure in Phase 2



Basic Paxos: Failure in Phase 2



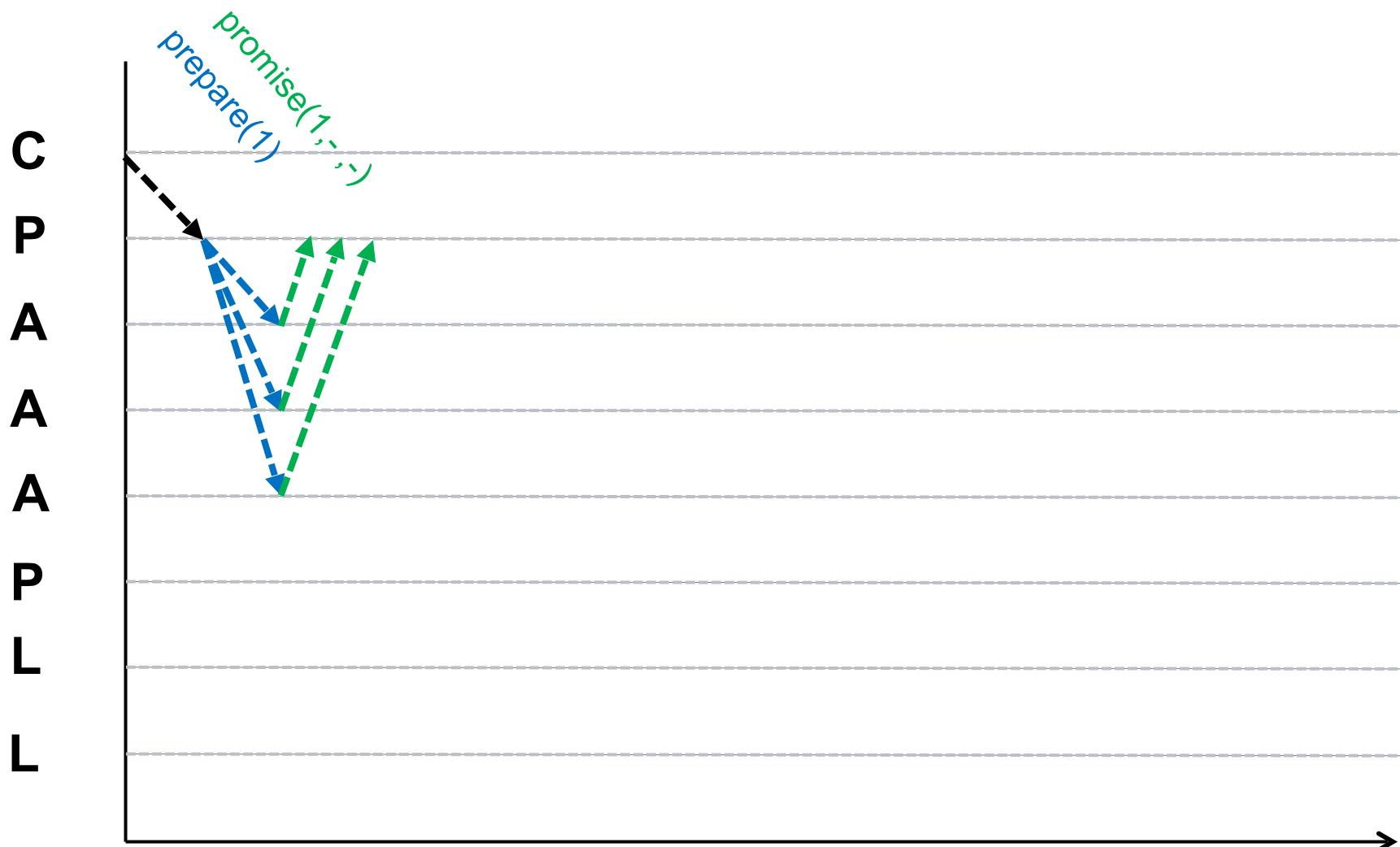
Basic Paxos: Dueling proposers



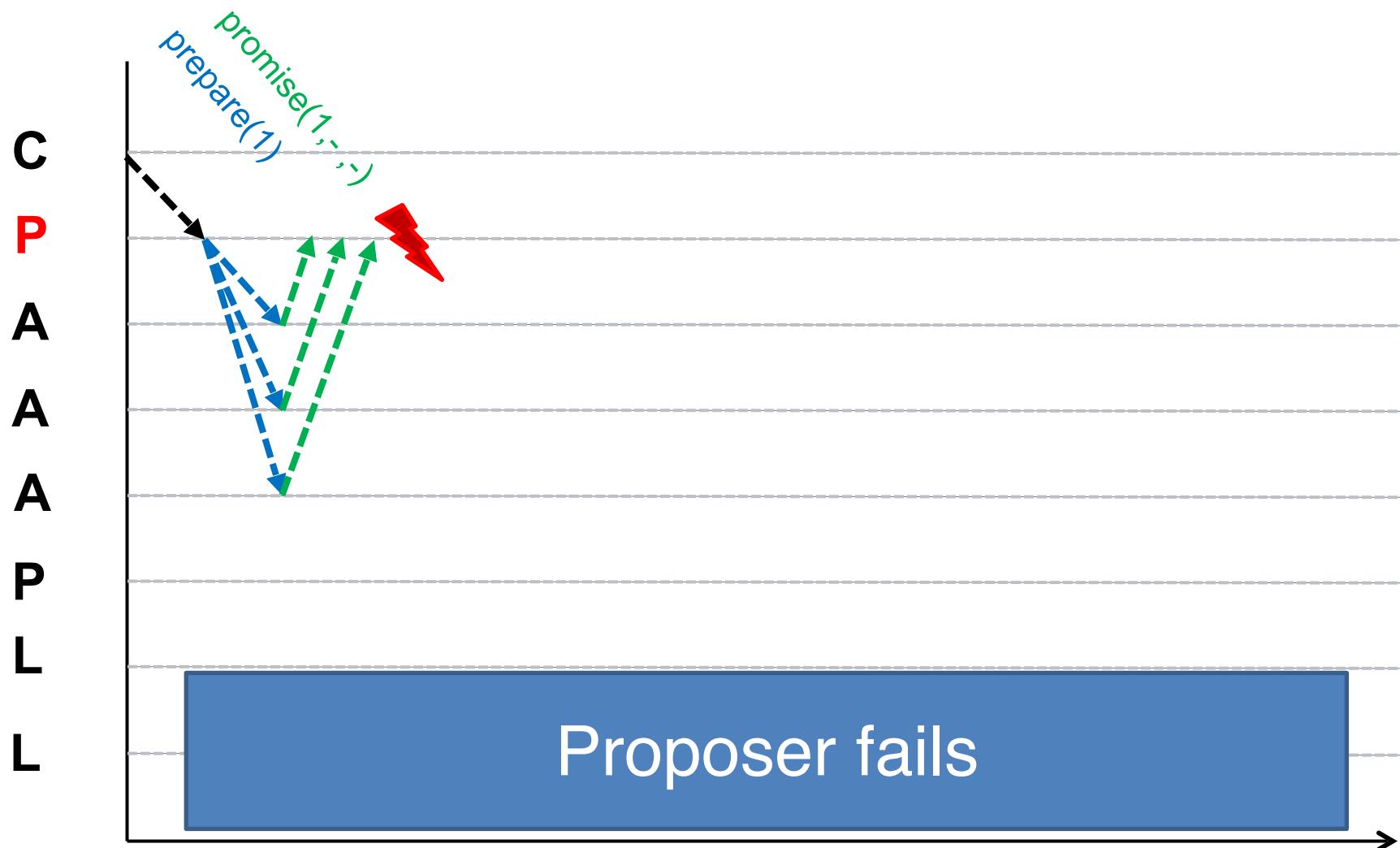
Dueling proposers



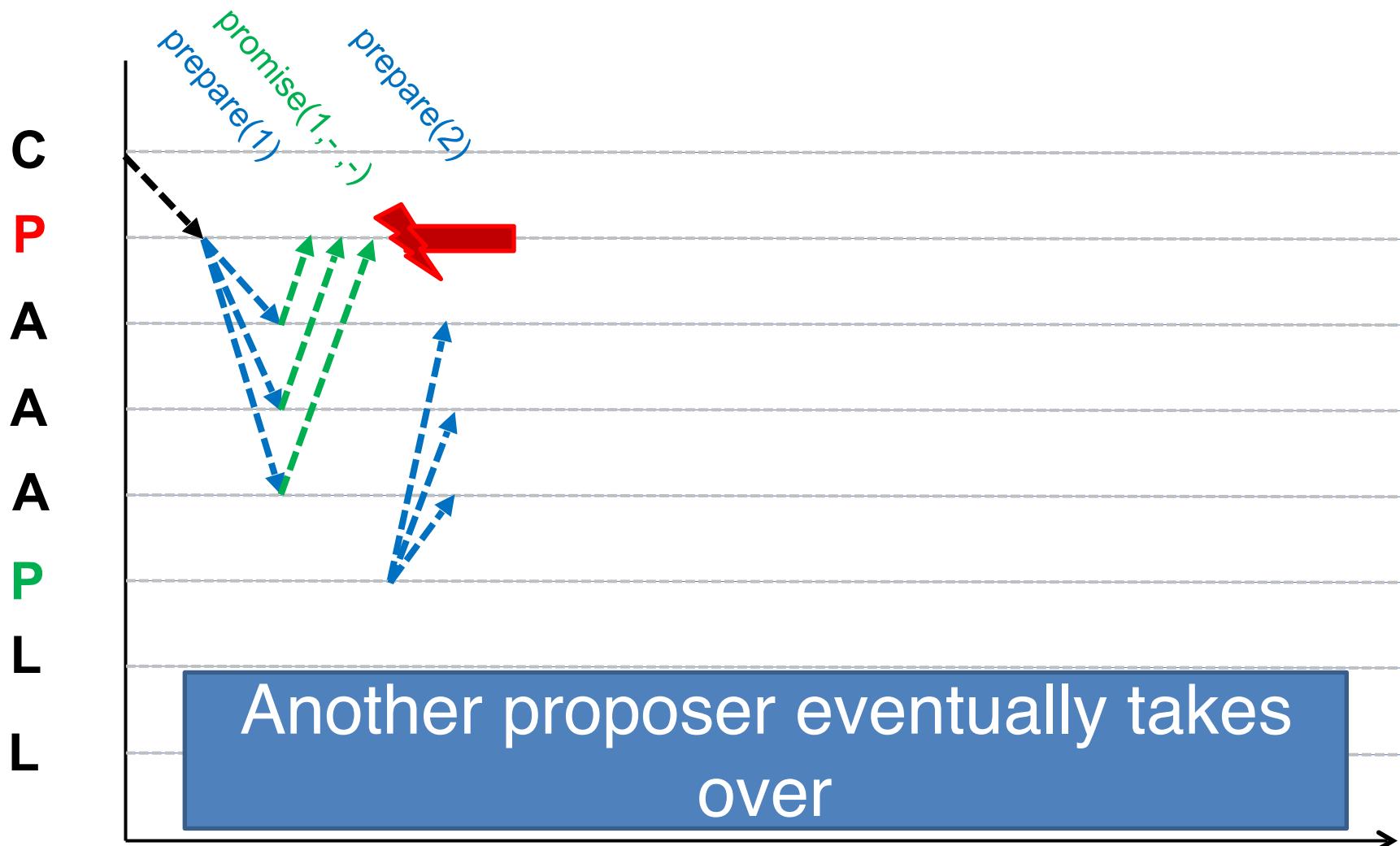
Dueling proposers



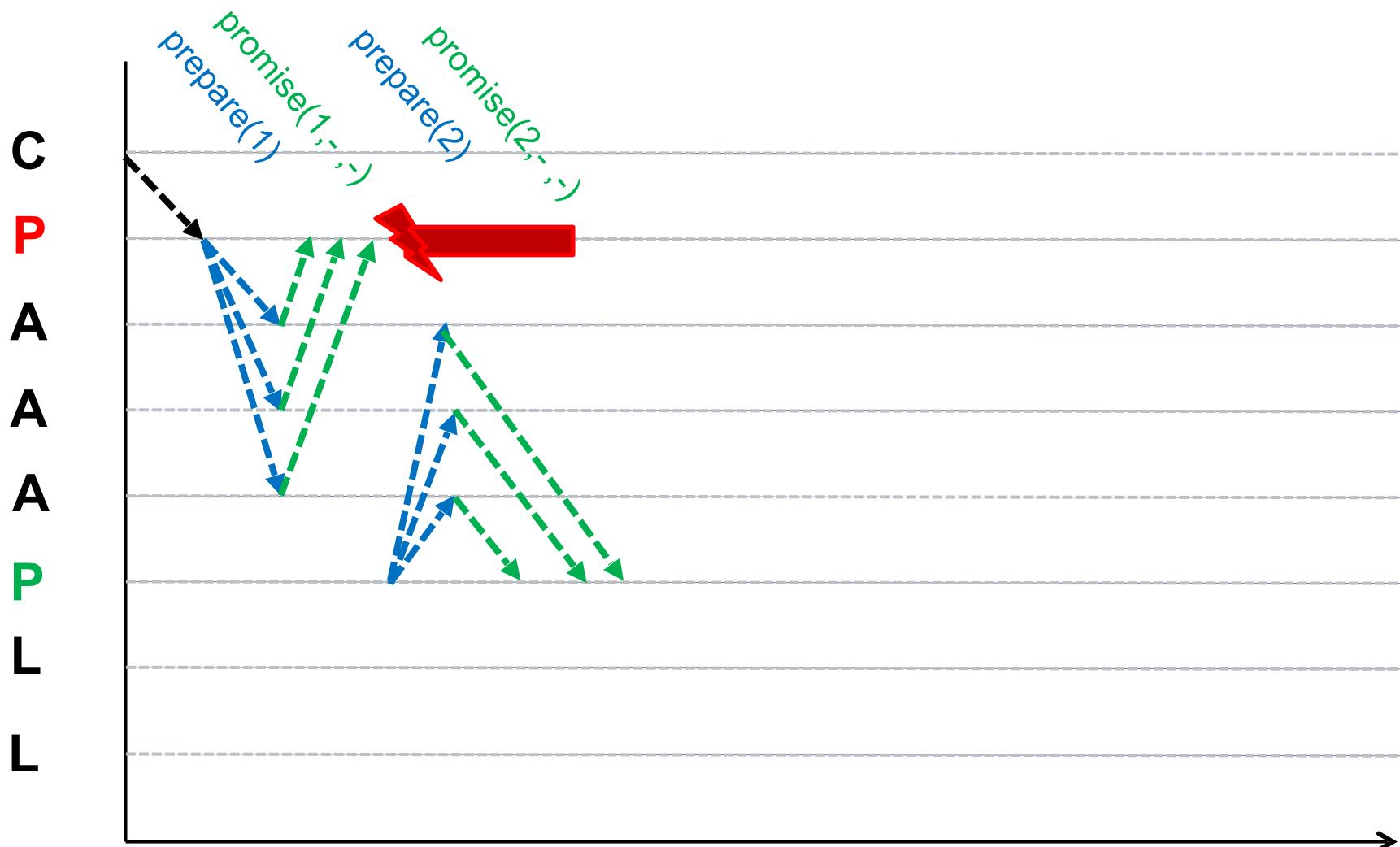
Dueling proposers



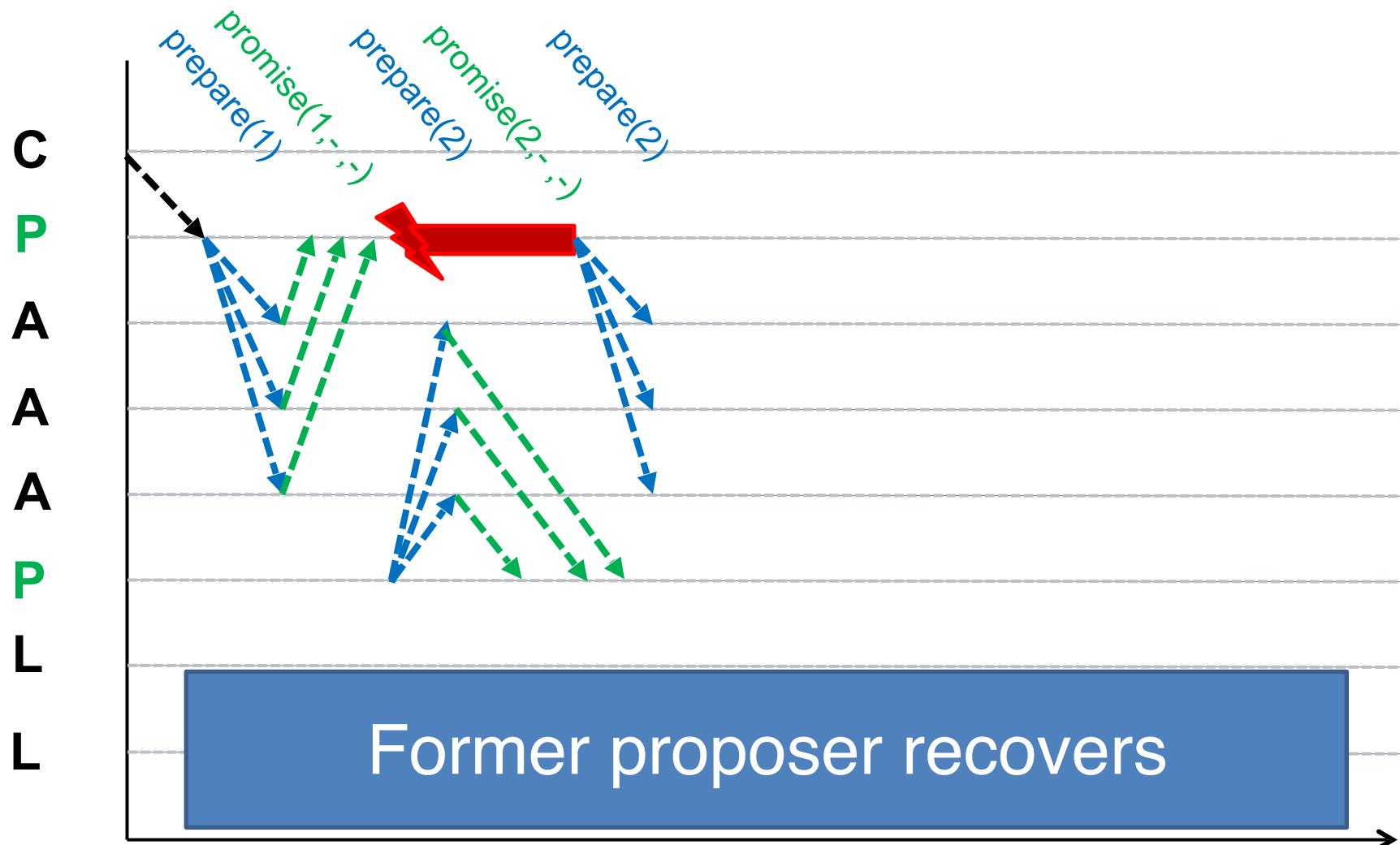
Dueling proposers



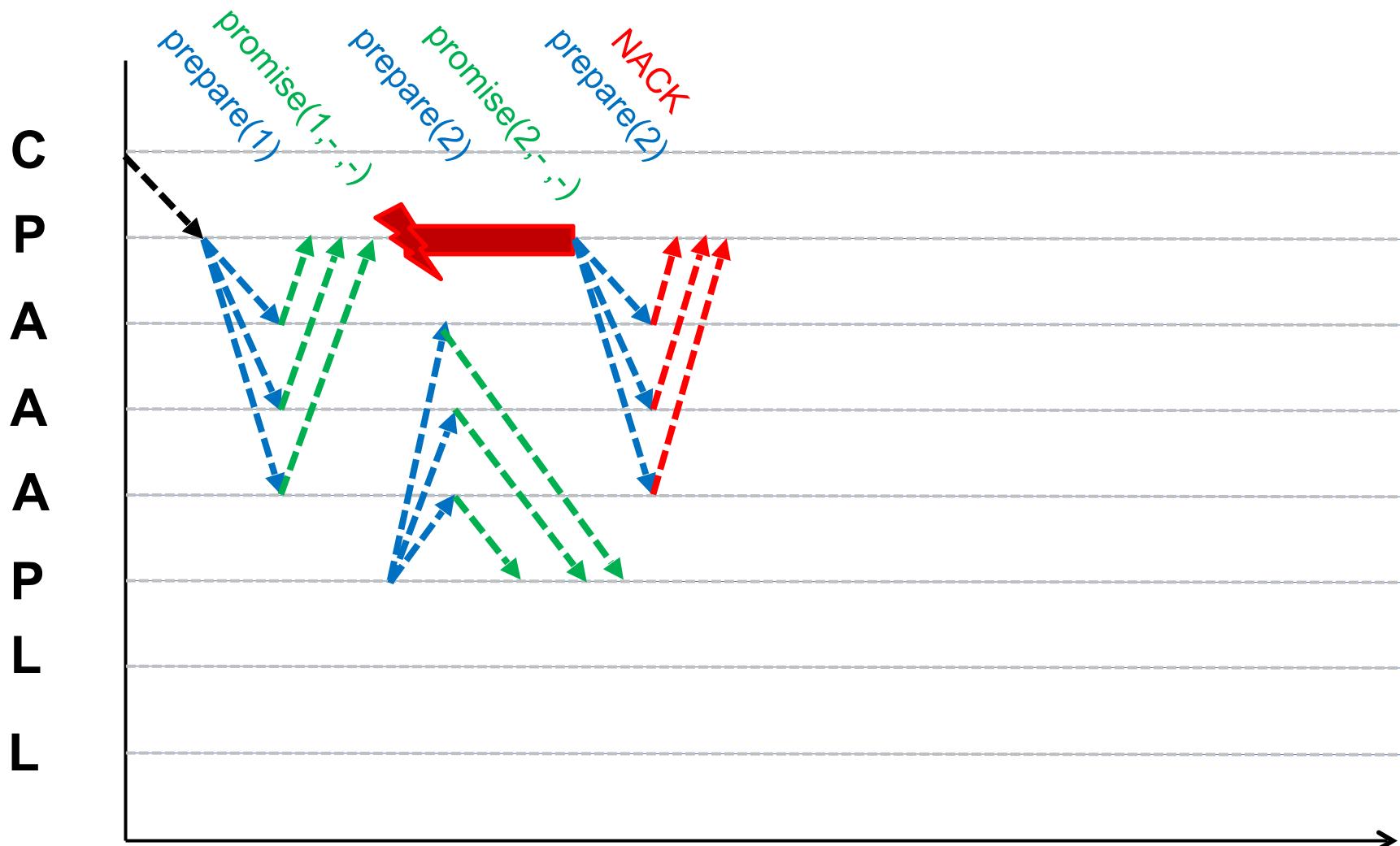
Dueling proposers



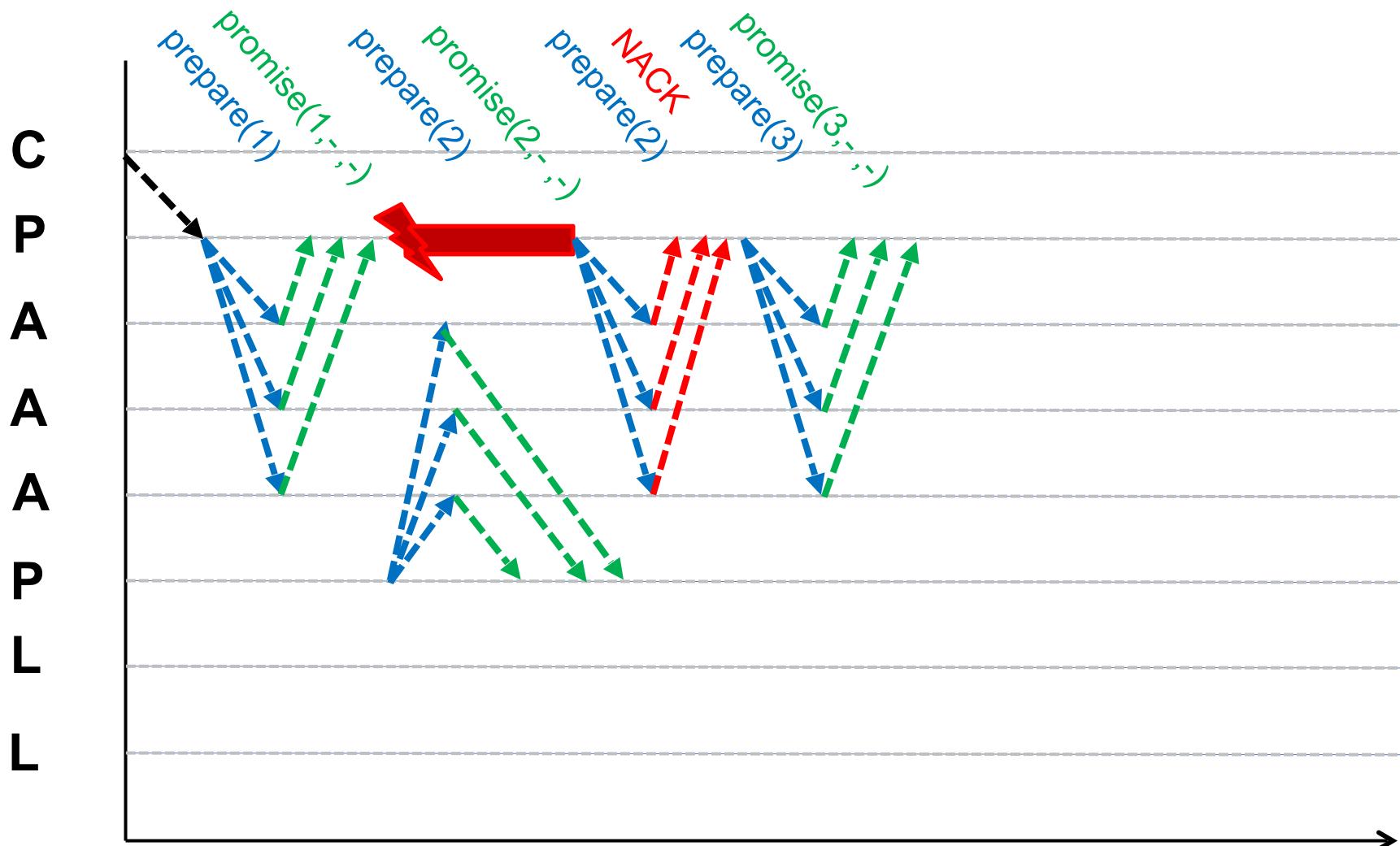
Dueling proposers



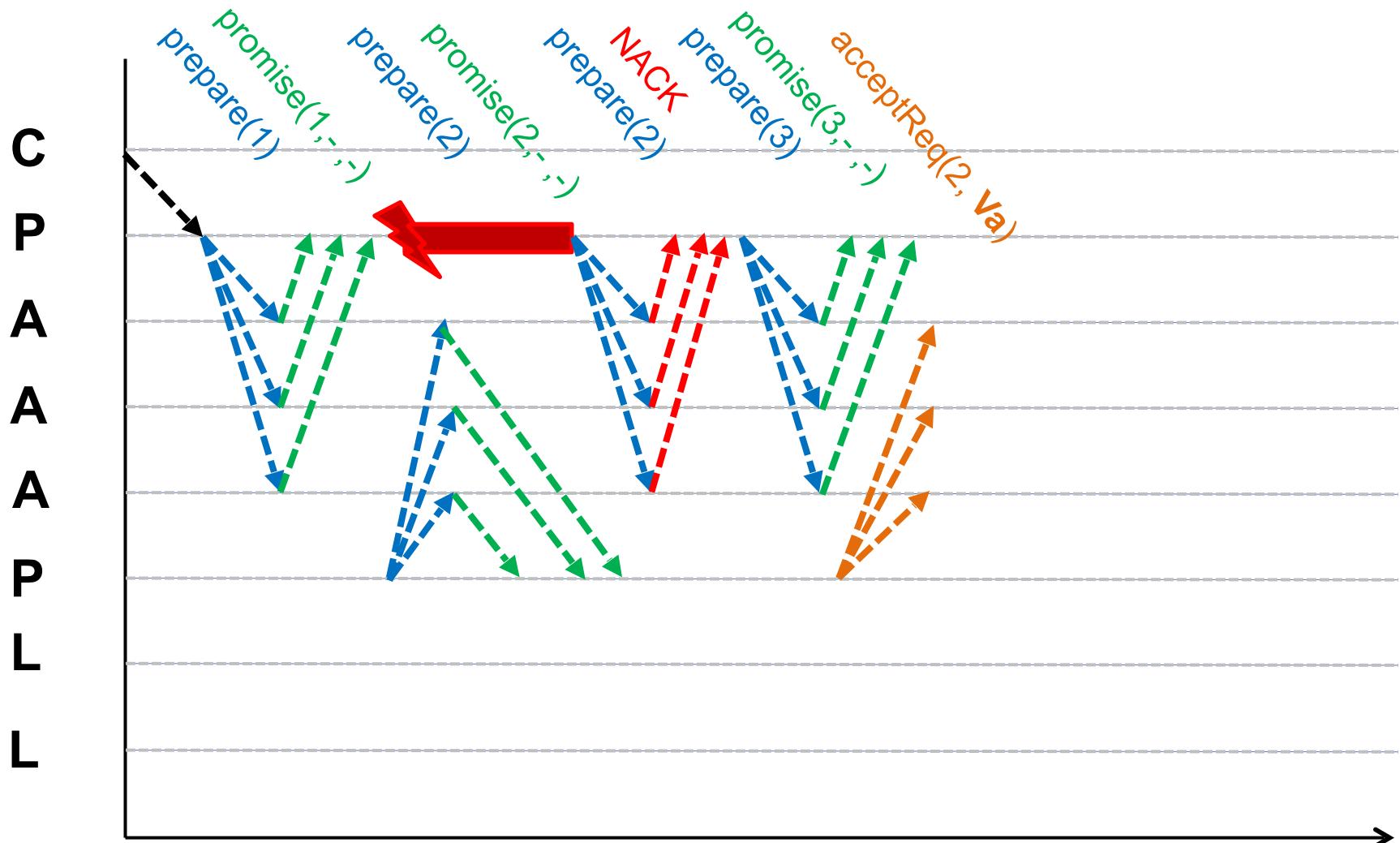
Dueling proposers



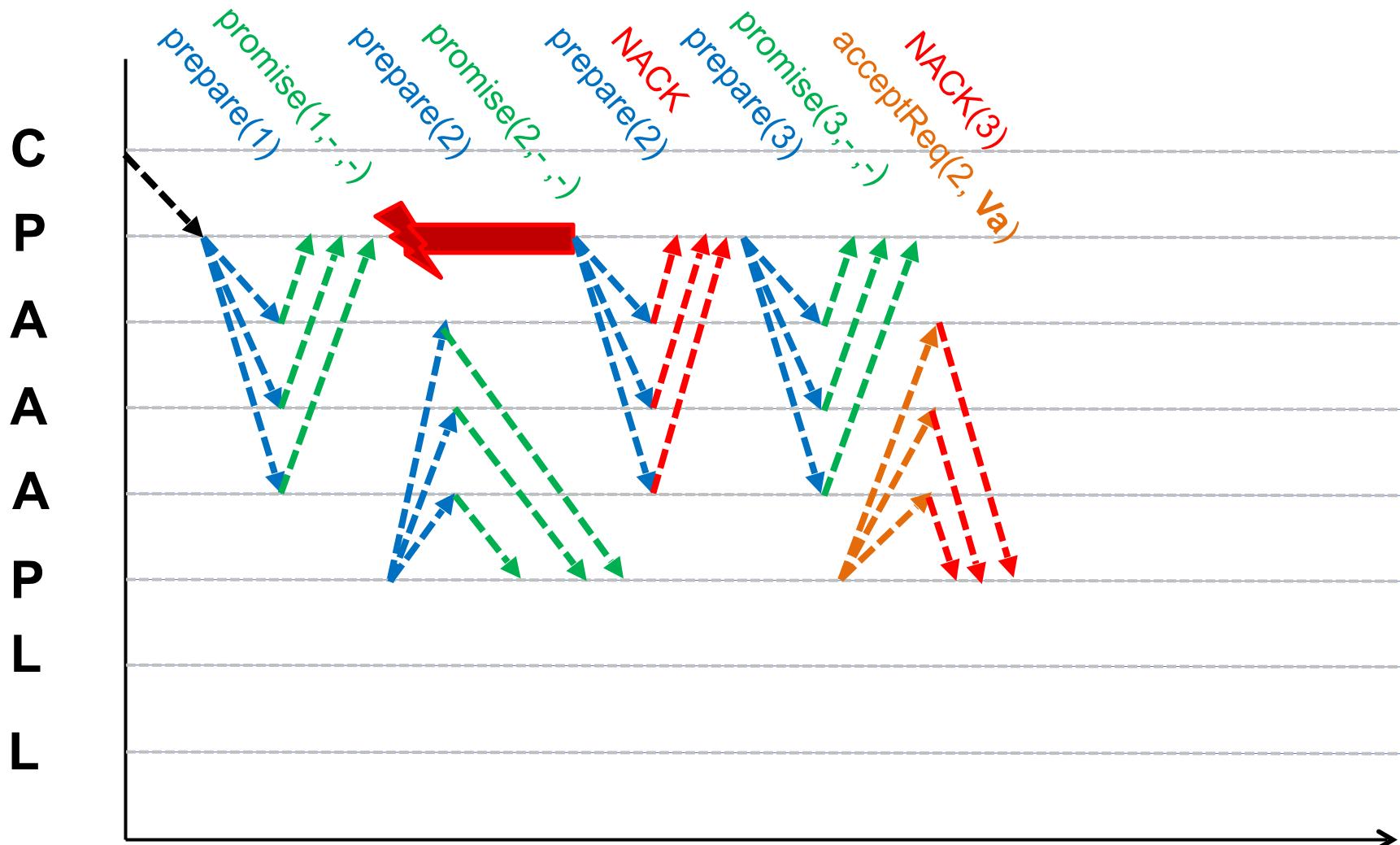
Dueling proposers



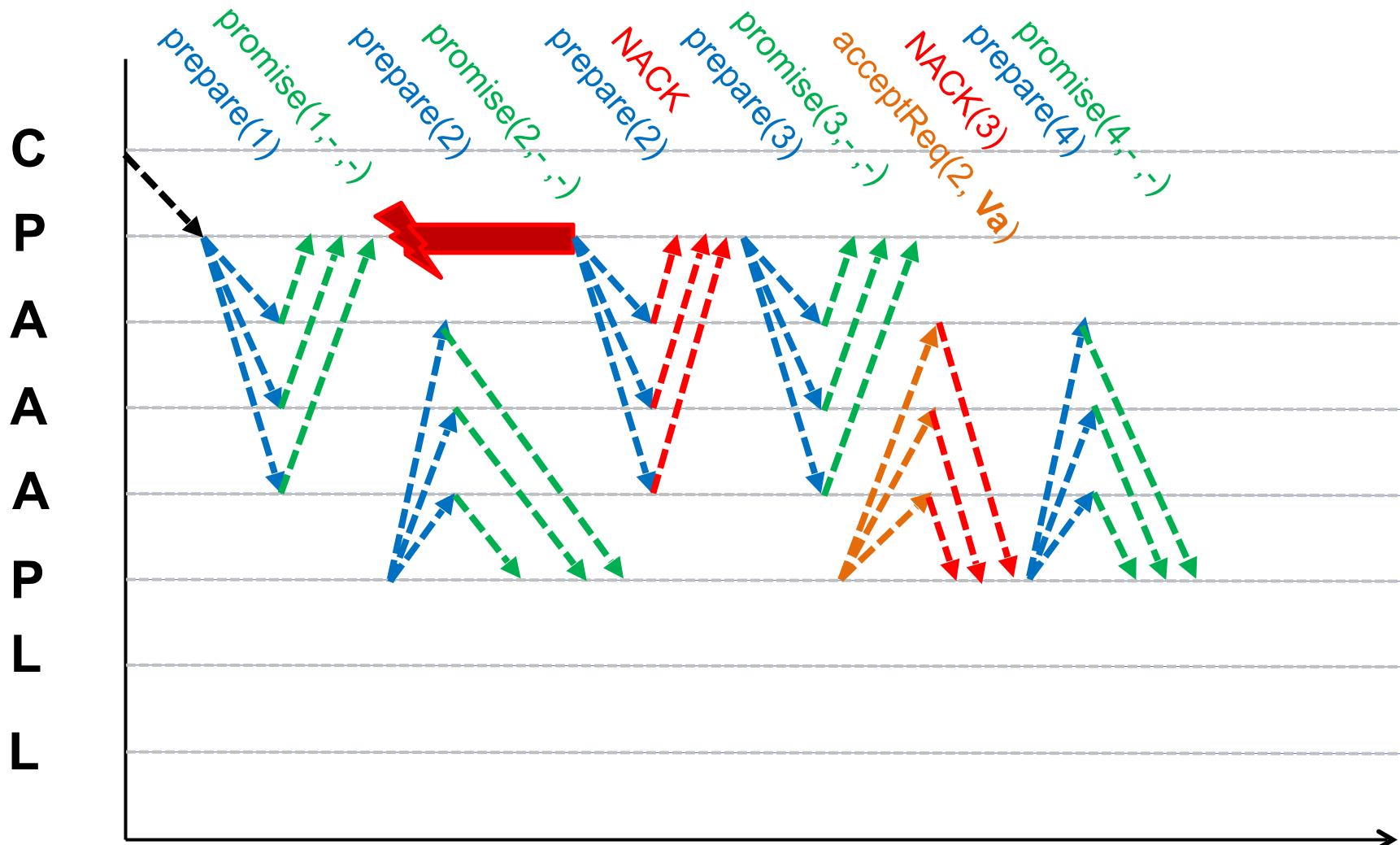
Dueling proposers



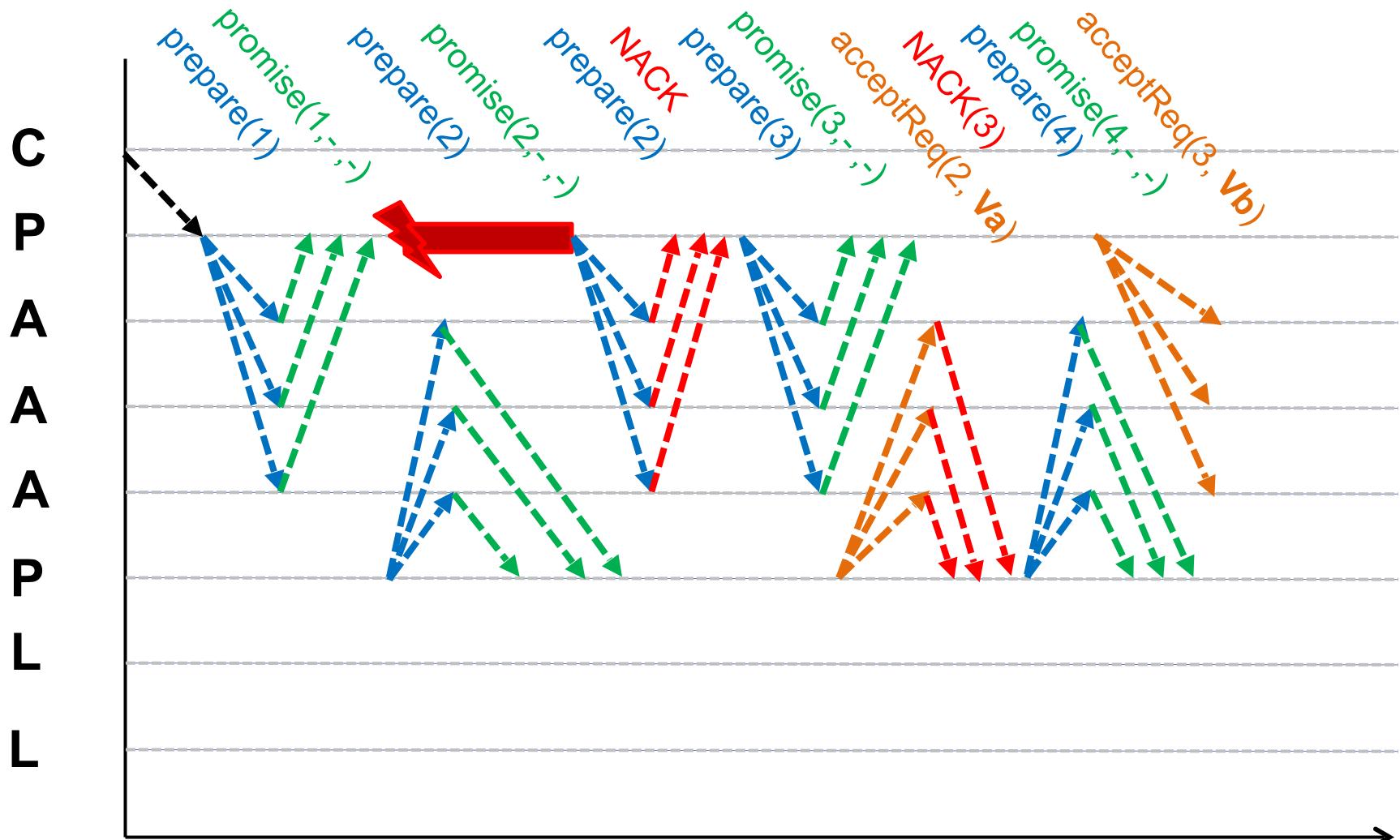
Dueling proposers



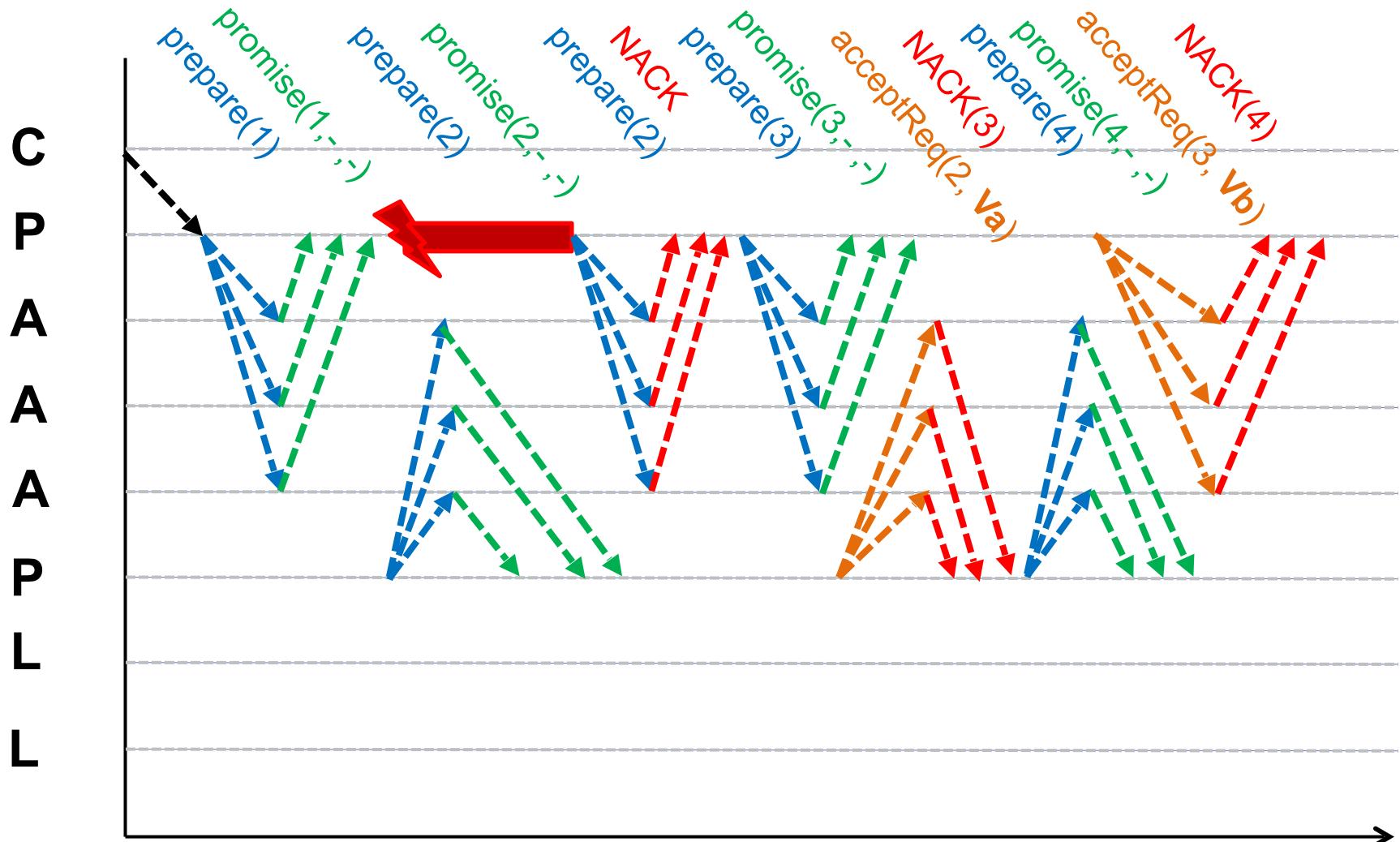
Dueling proposers



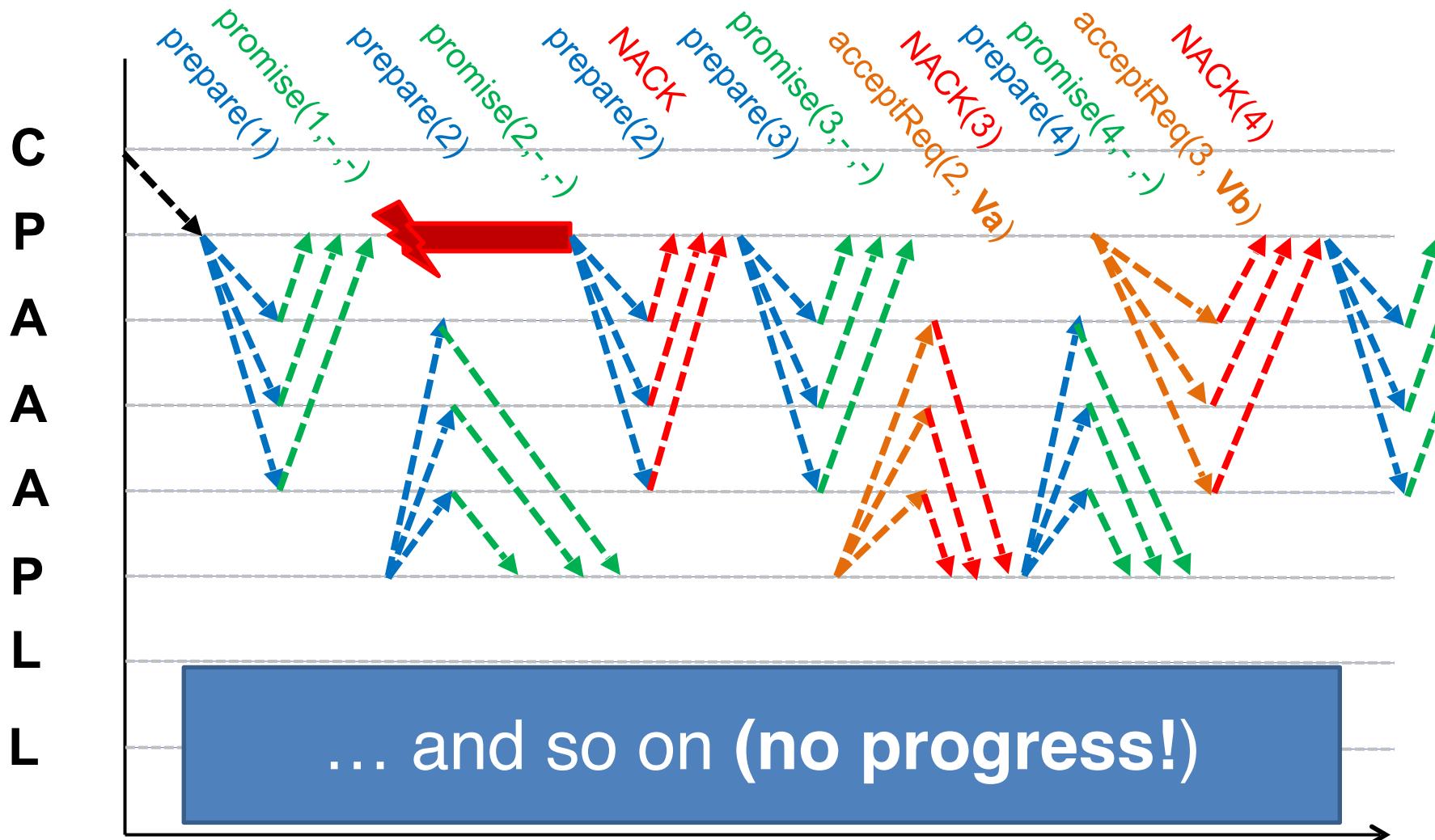
Dueling proposers



Dueling proposers



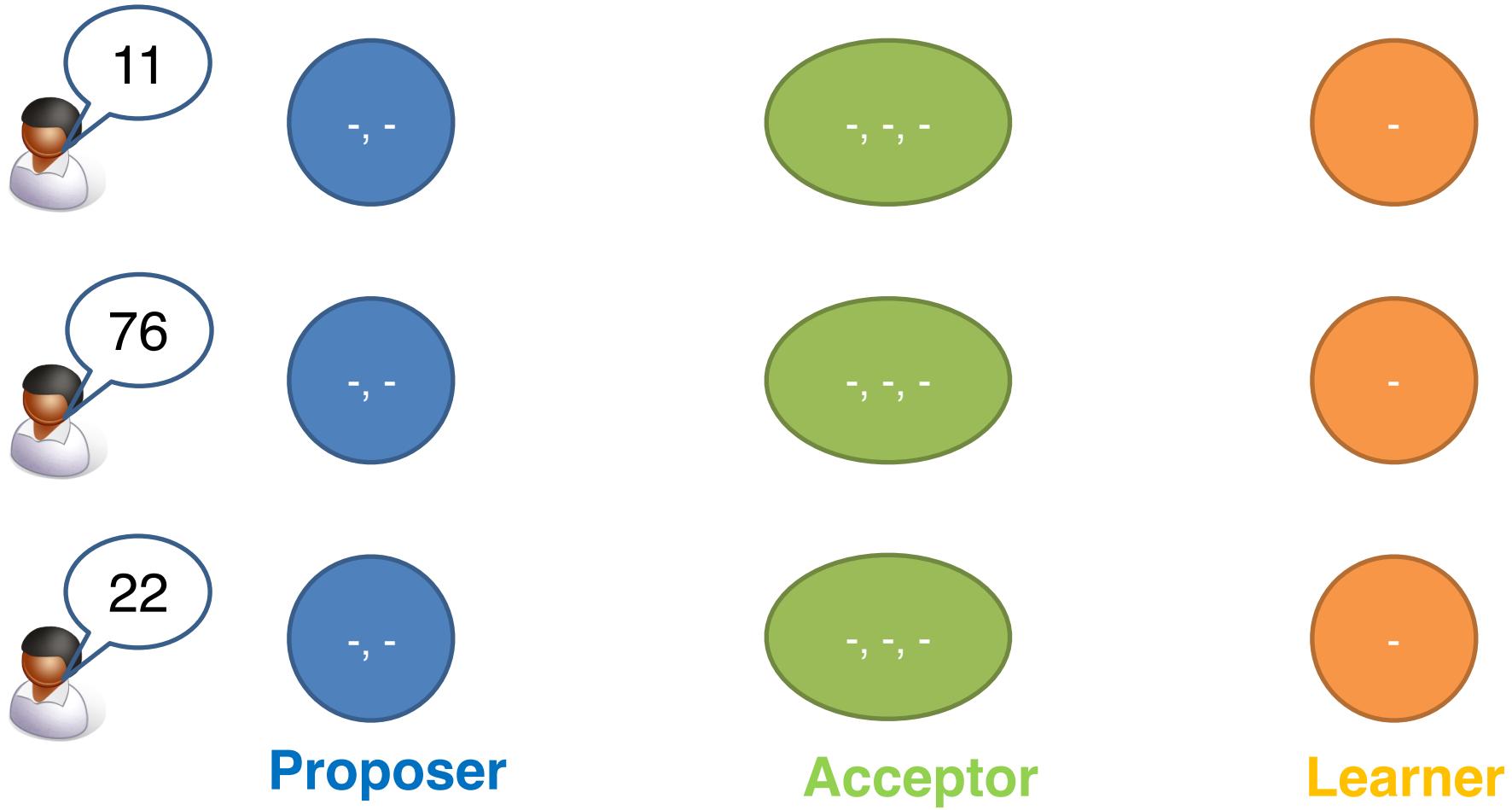
Dueling proposers



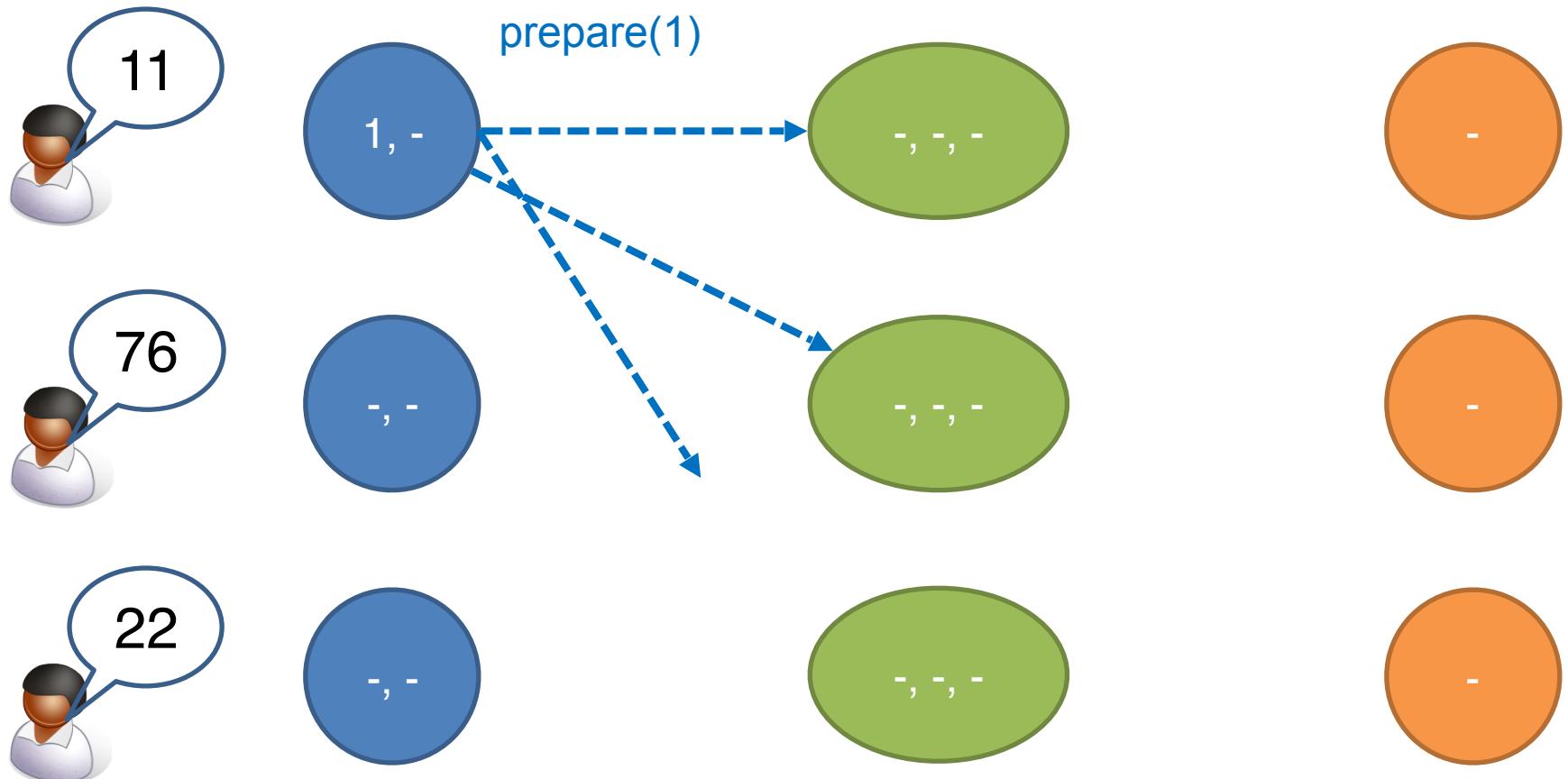
Basic Paxos: Communication failures



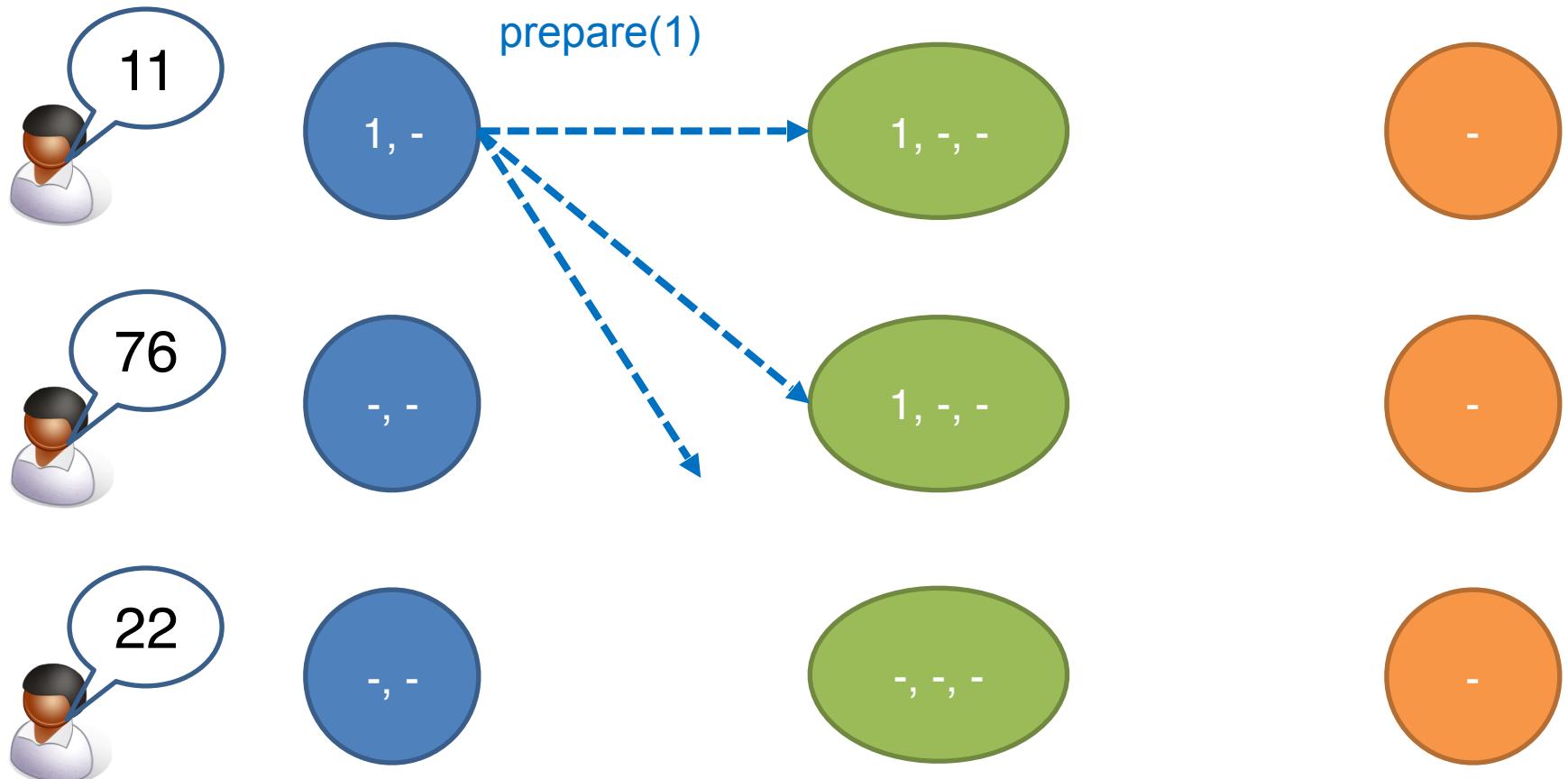
Basic Paxos with comm. failures



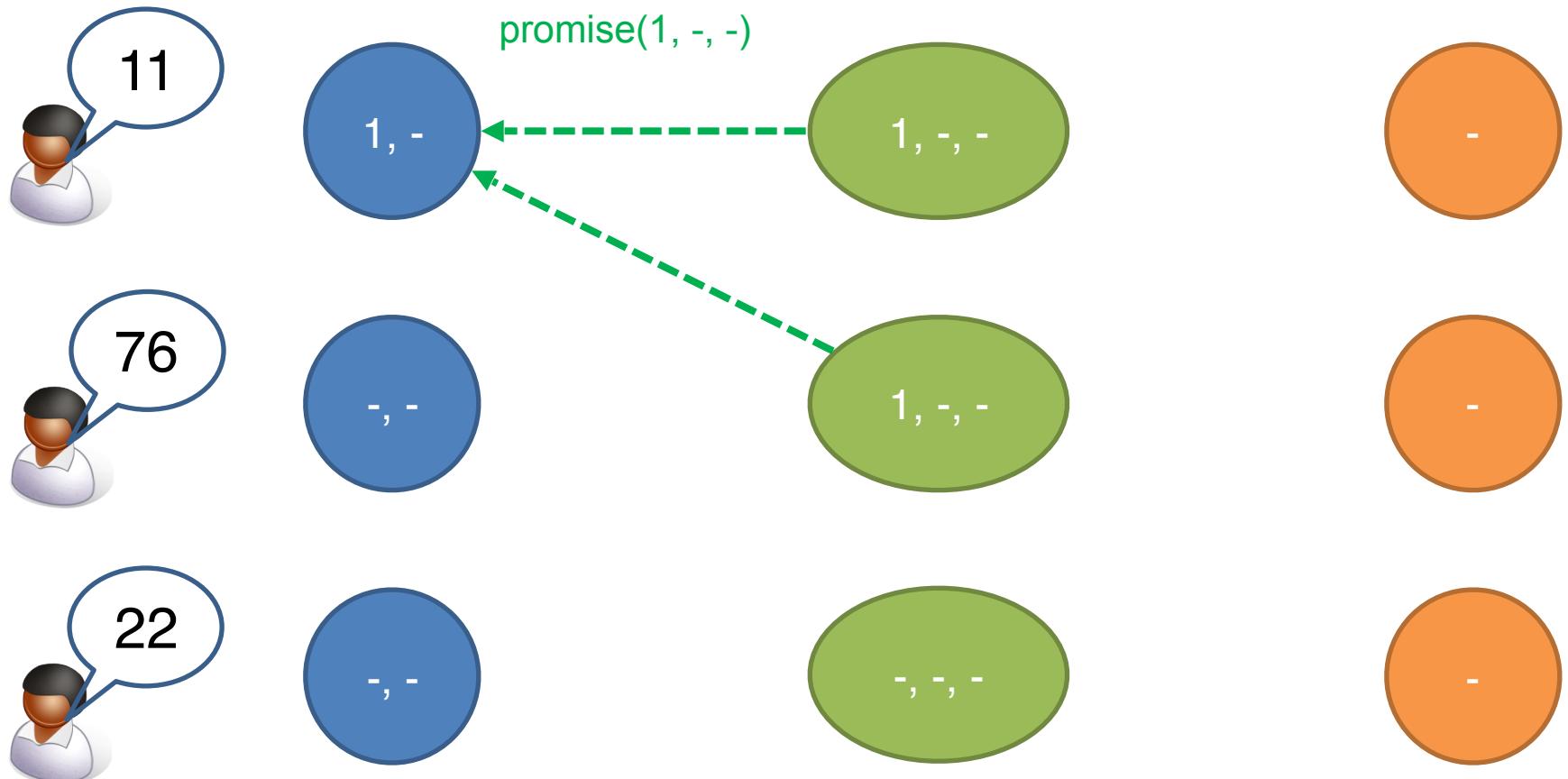
Basic Paxos with comm. failures



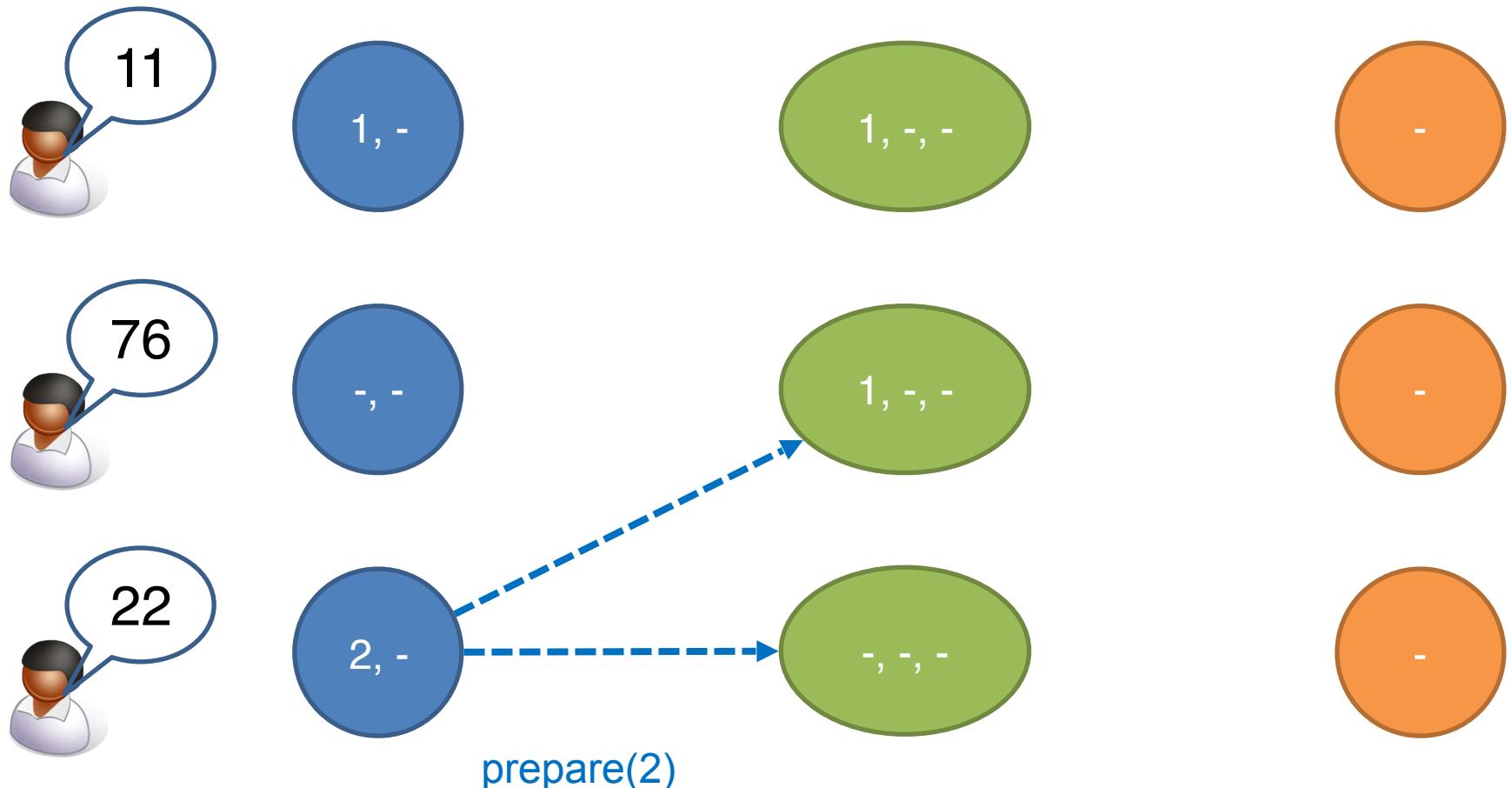
Basic Paxos with comm. failures



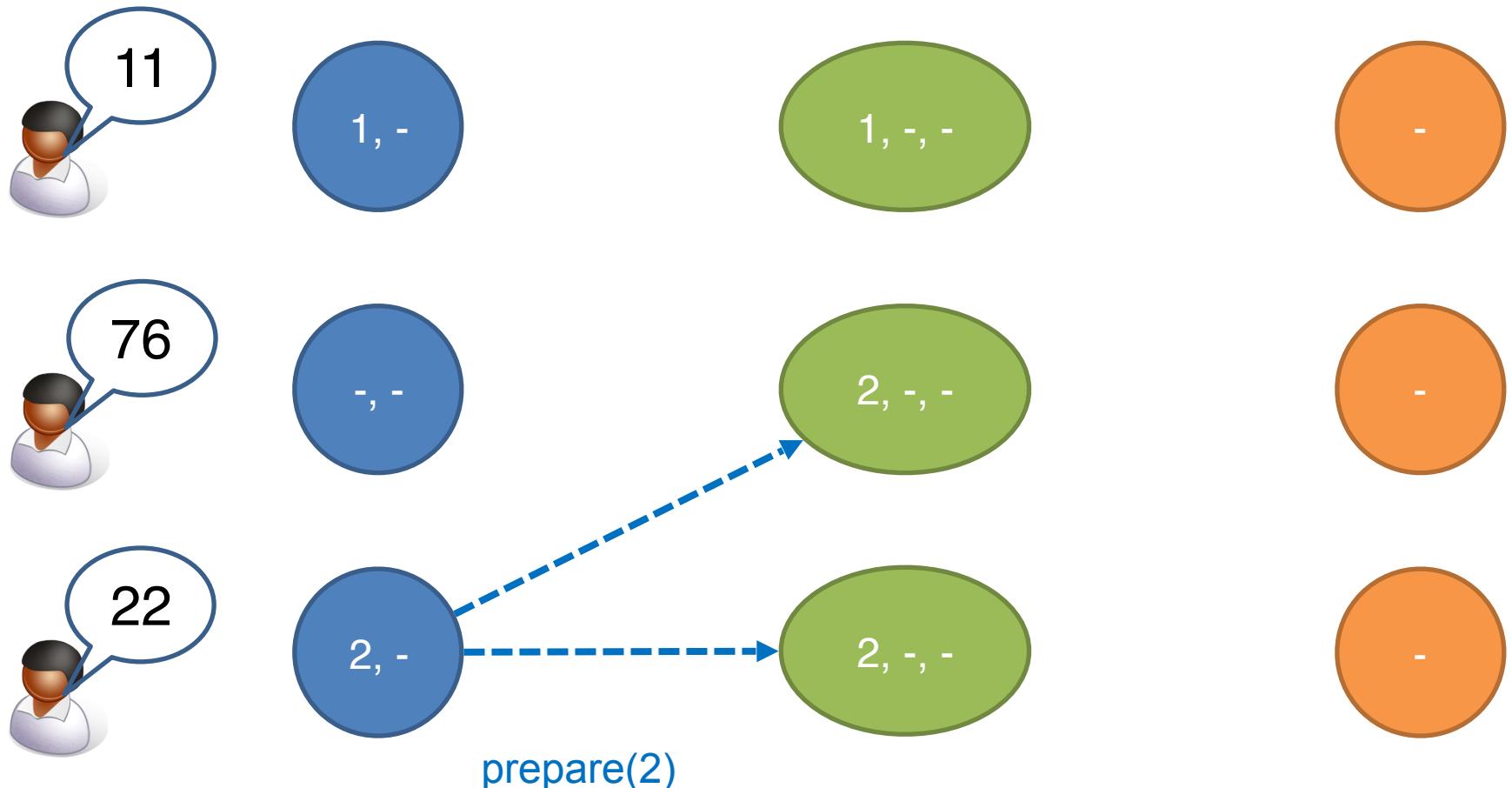
Basic Paxos with comm. failures



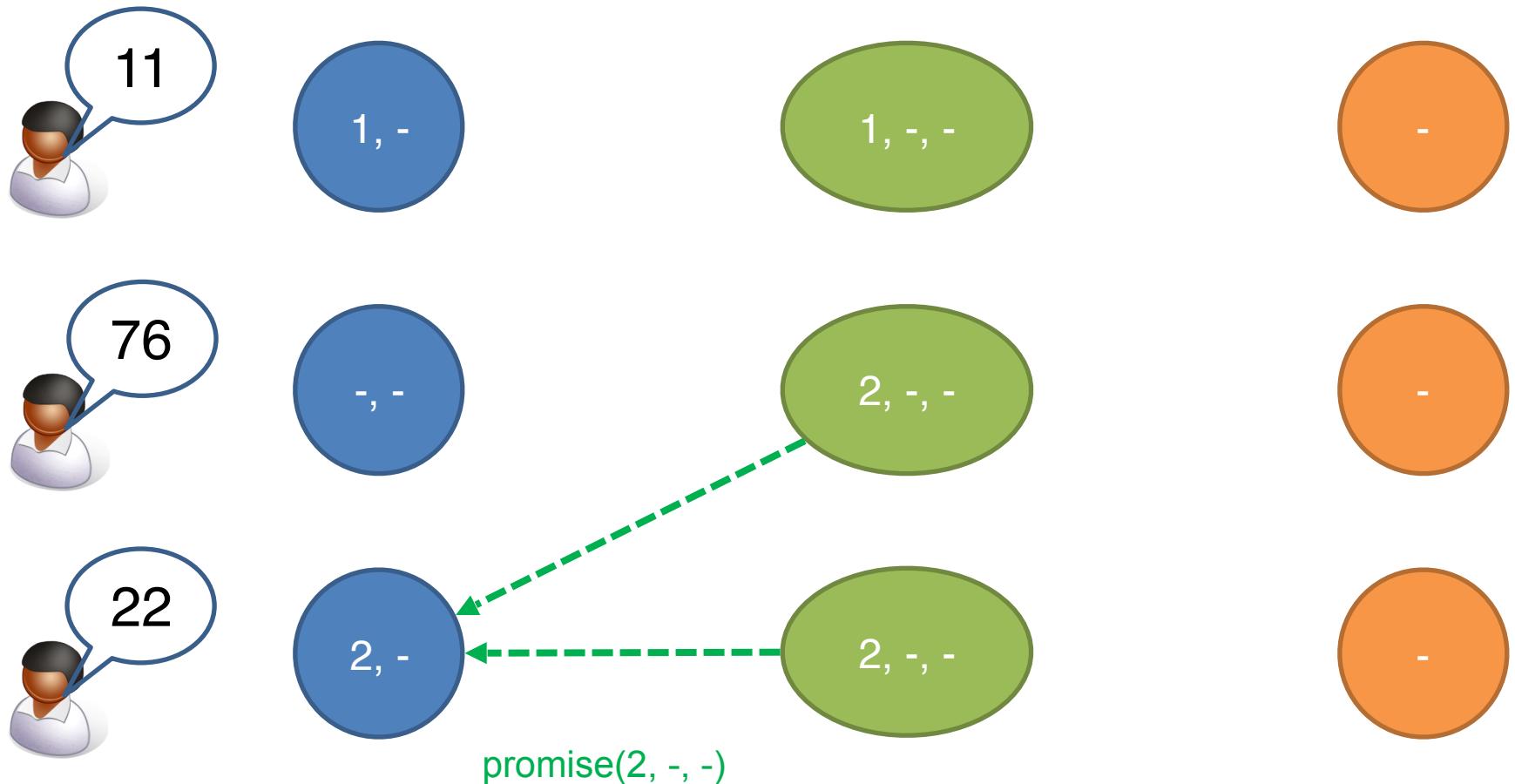
Basic Paxos with comm. failures



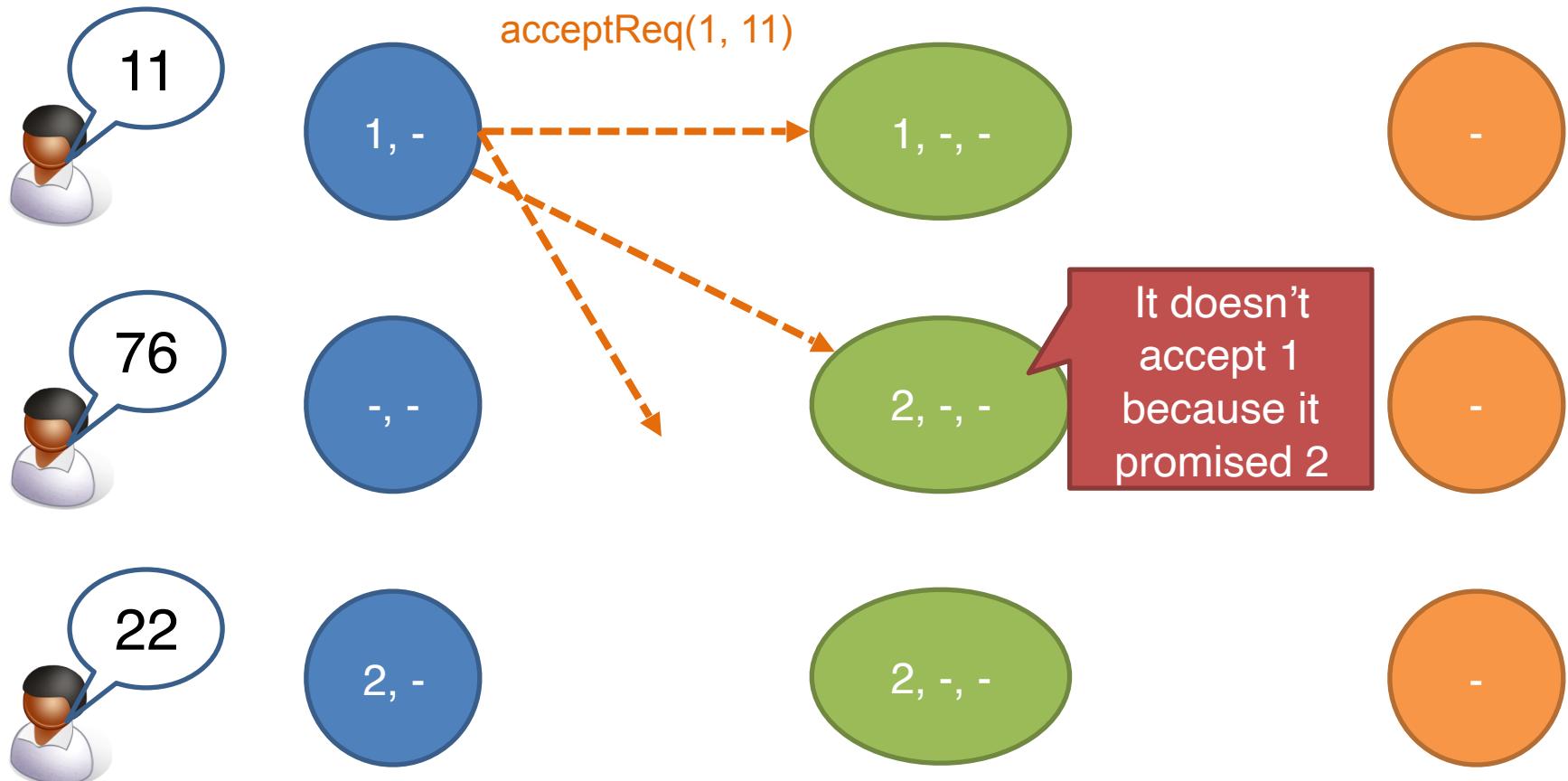
Basic Paxos with comm. failures



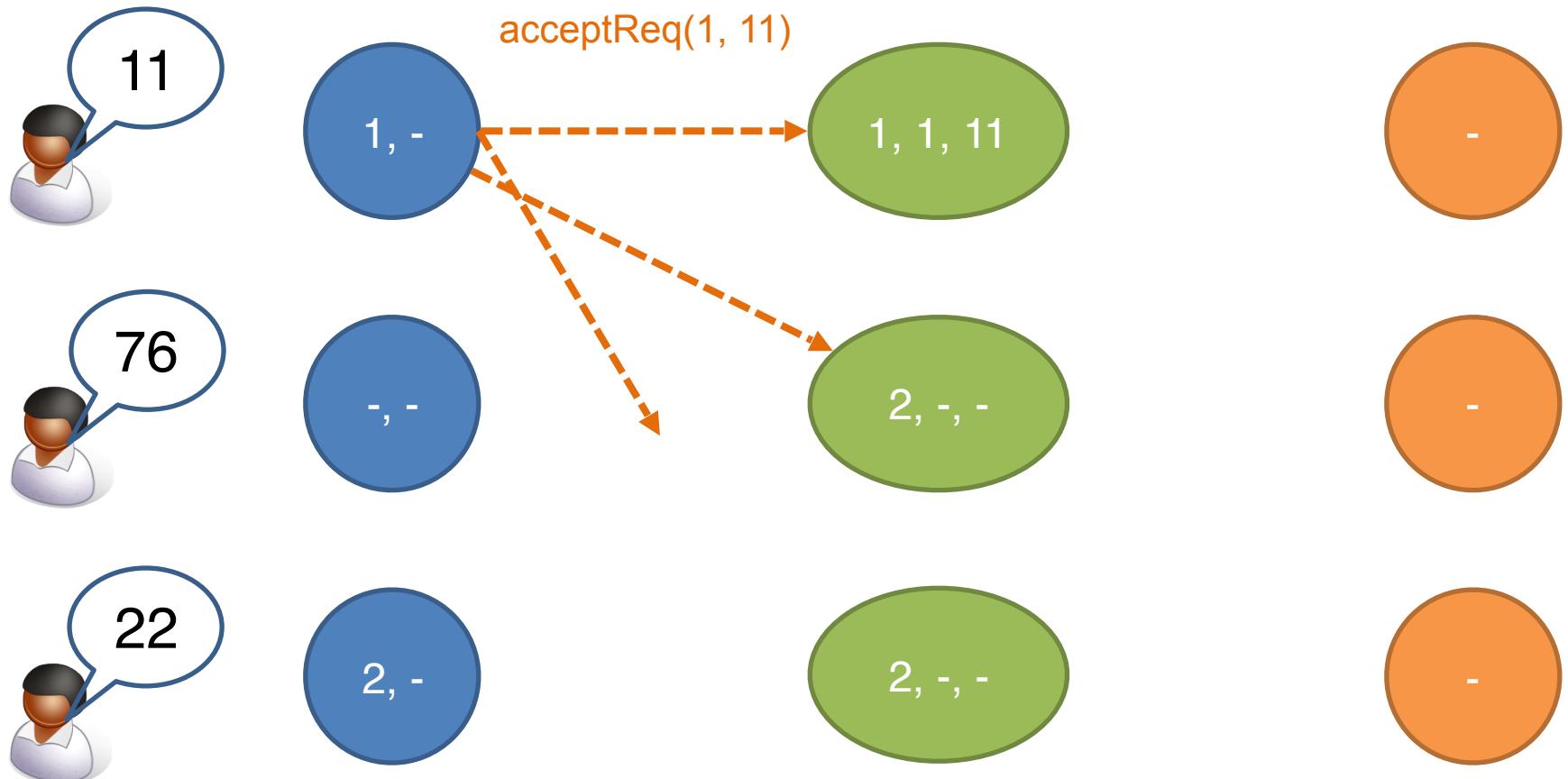
Basic Paxos with comm. failures



Basic Paxos with comm. failures

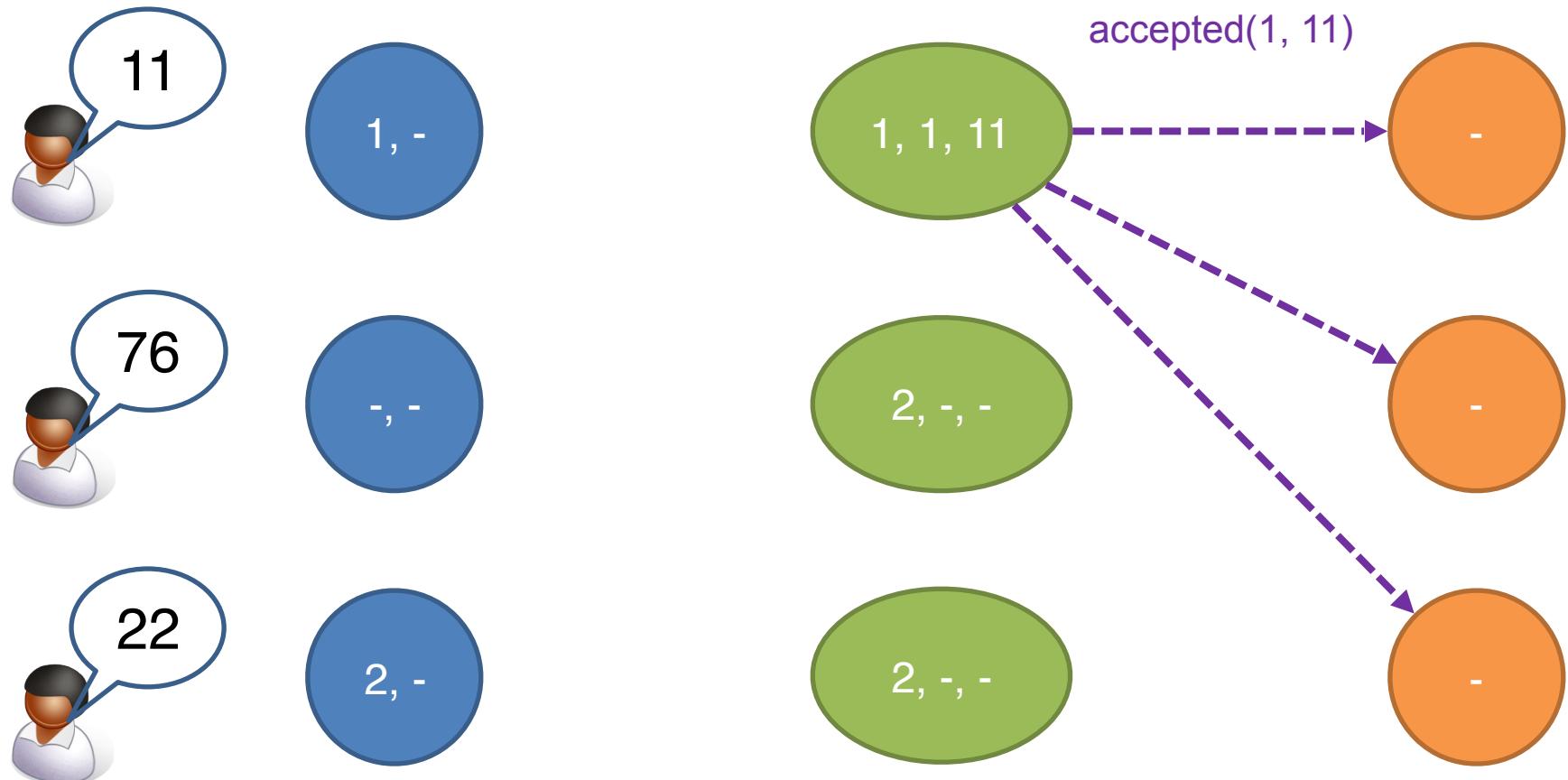


Basic Paxos with comm. failures

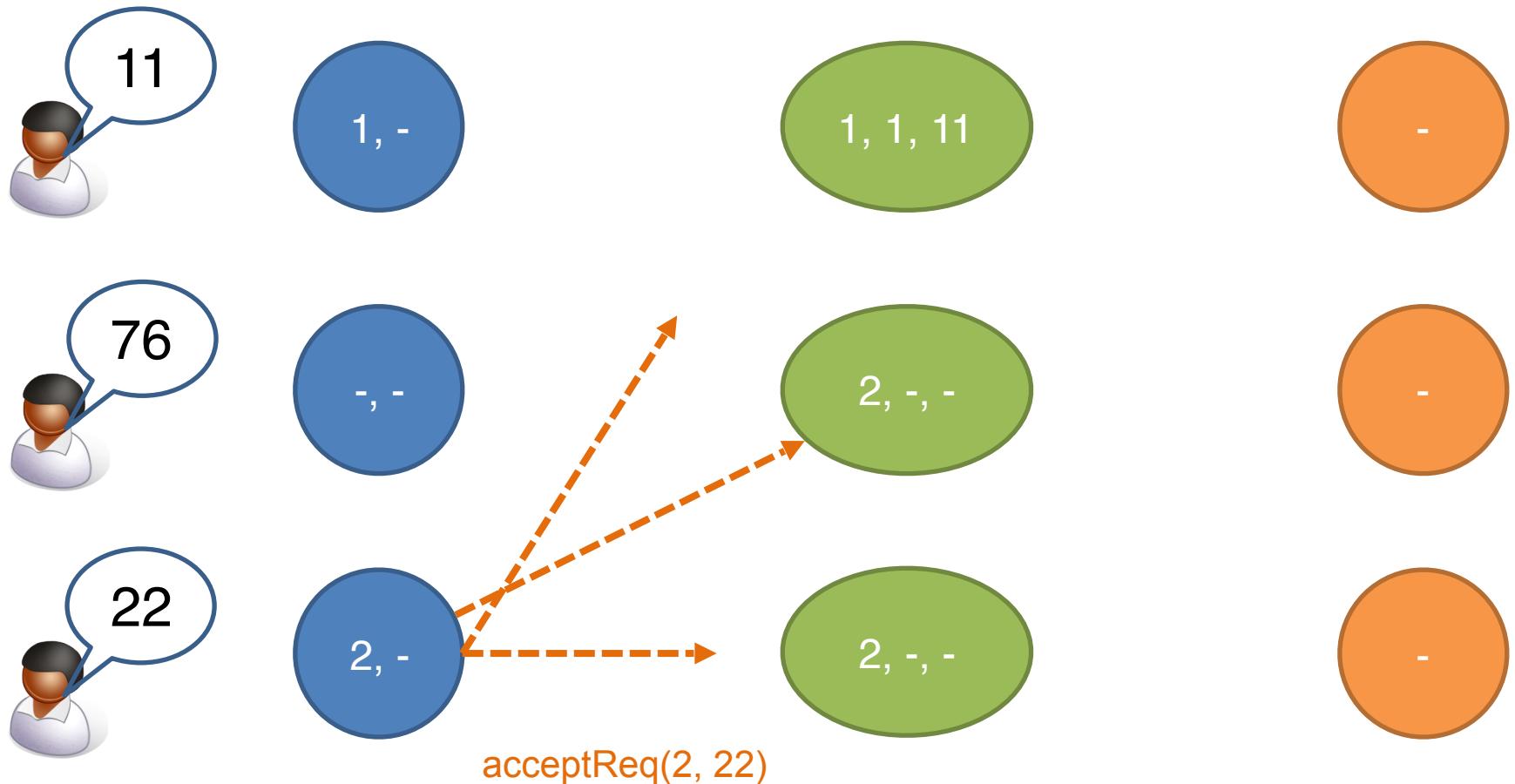


Basic Paxos with comm. failures

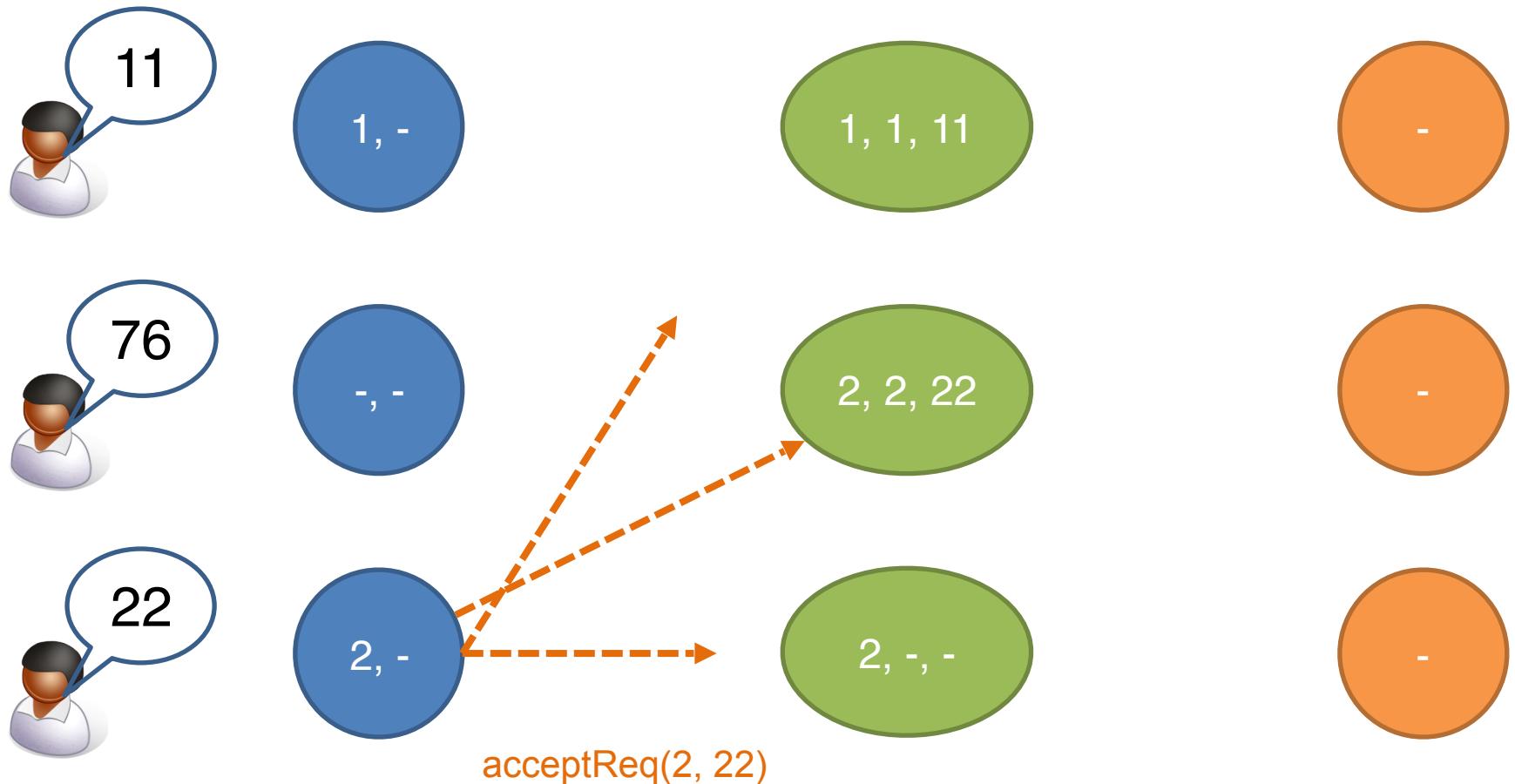
accepted(1, 11) message also send to all **proposers** (not shown here)



Basic Paxos with comm. failures

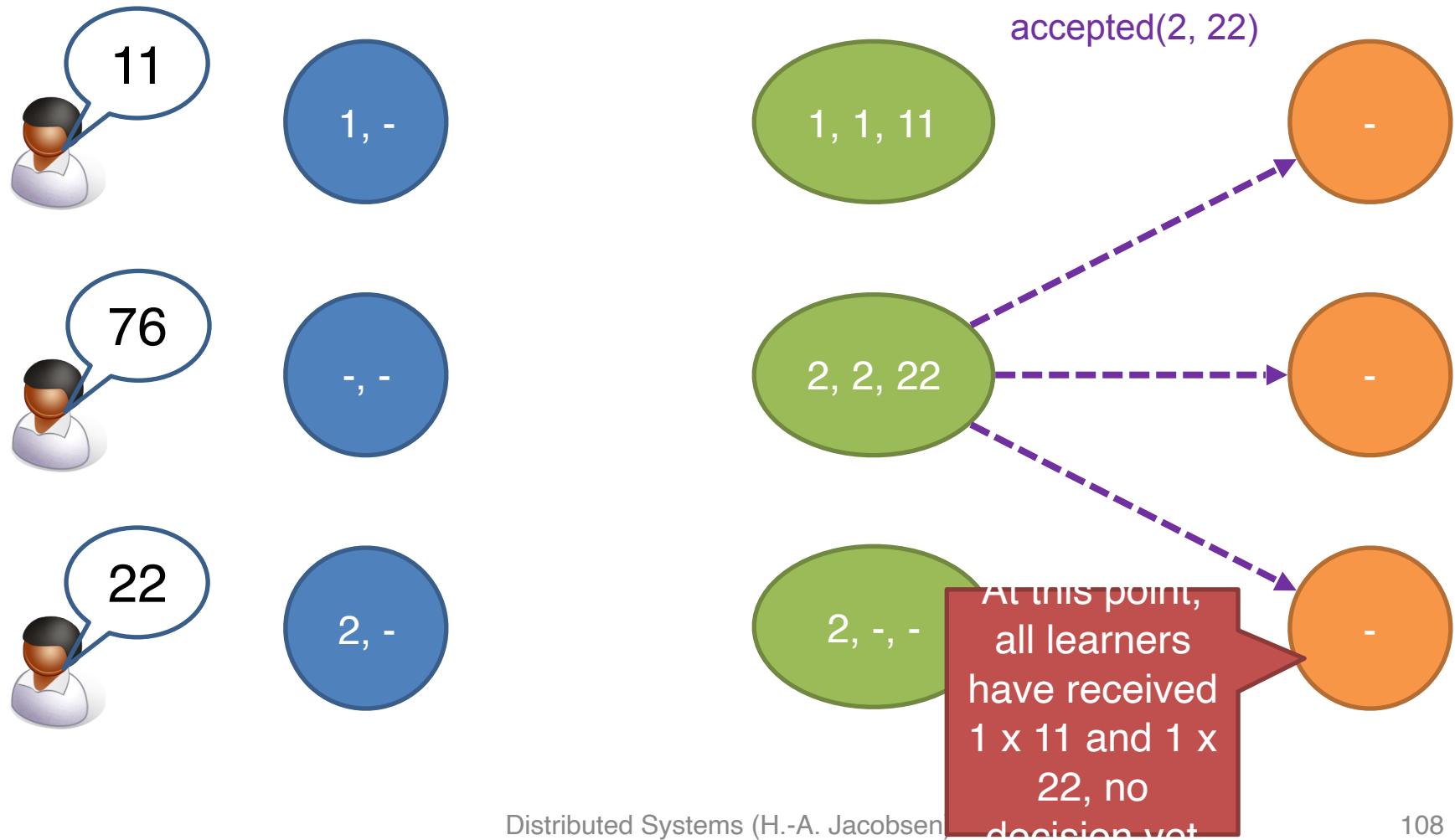


Basic Paxos with comm. failures

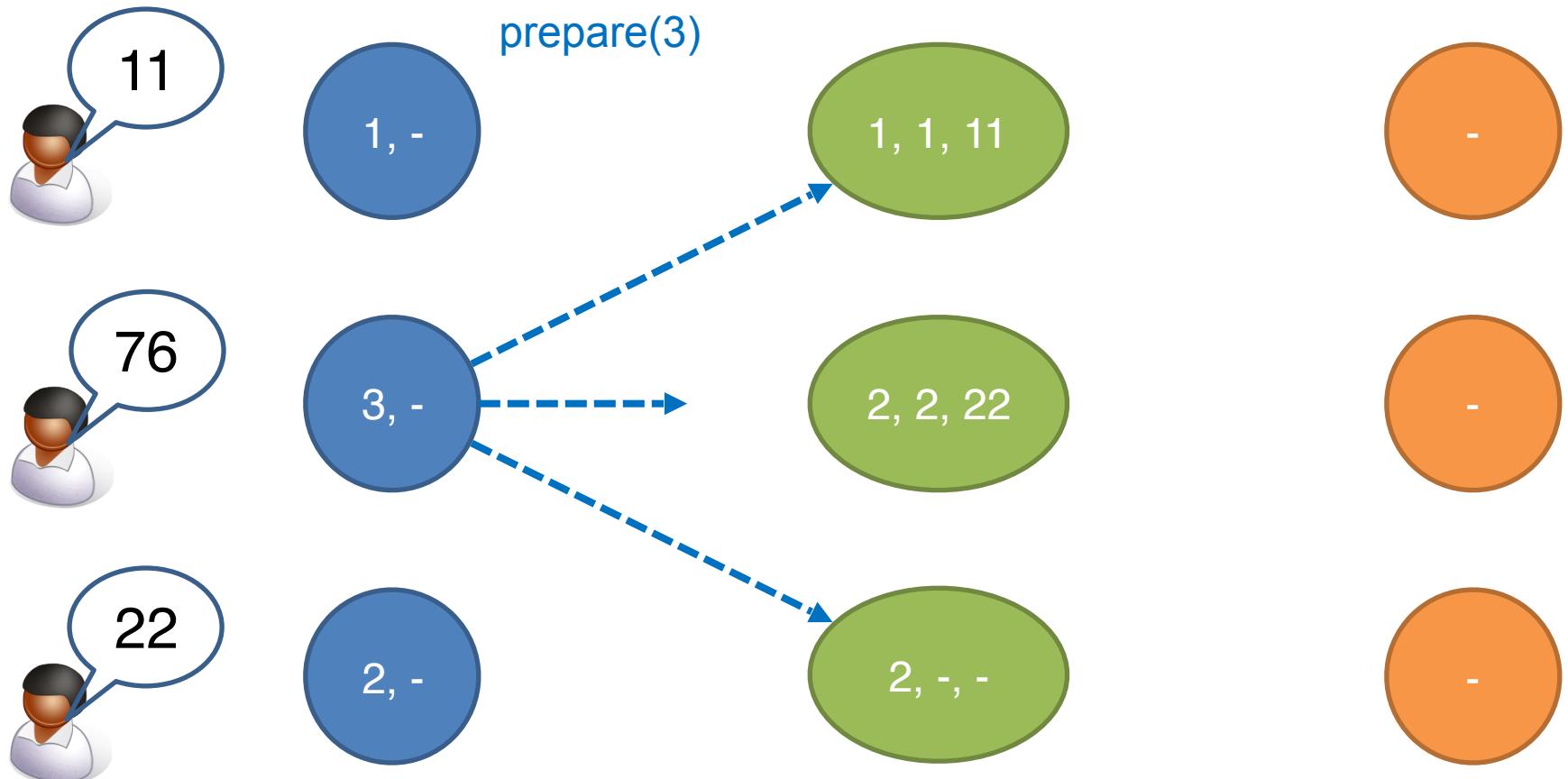


Basic Paxos with comm. failures

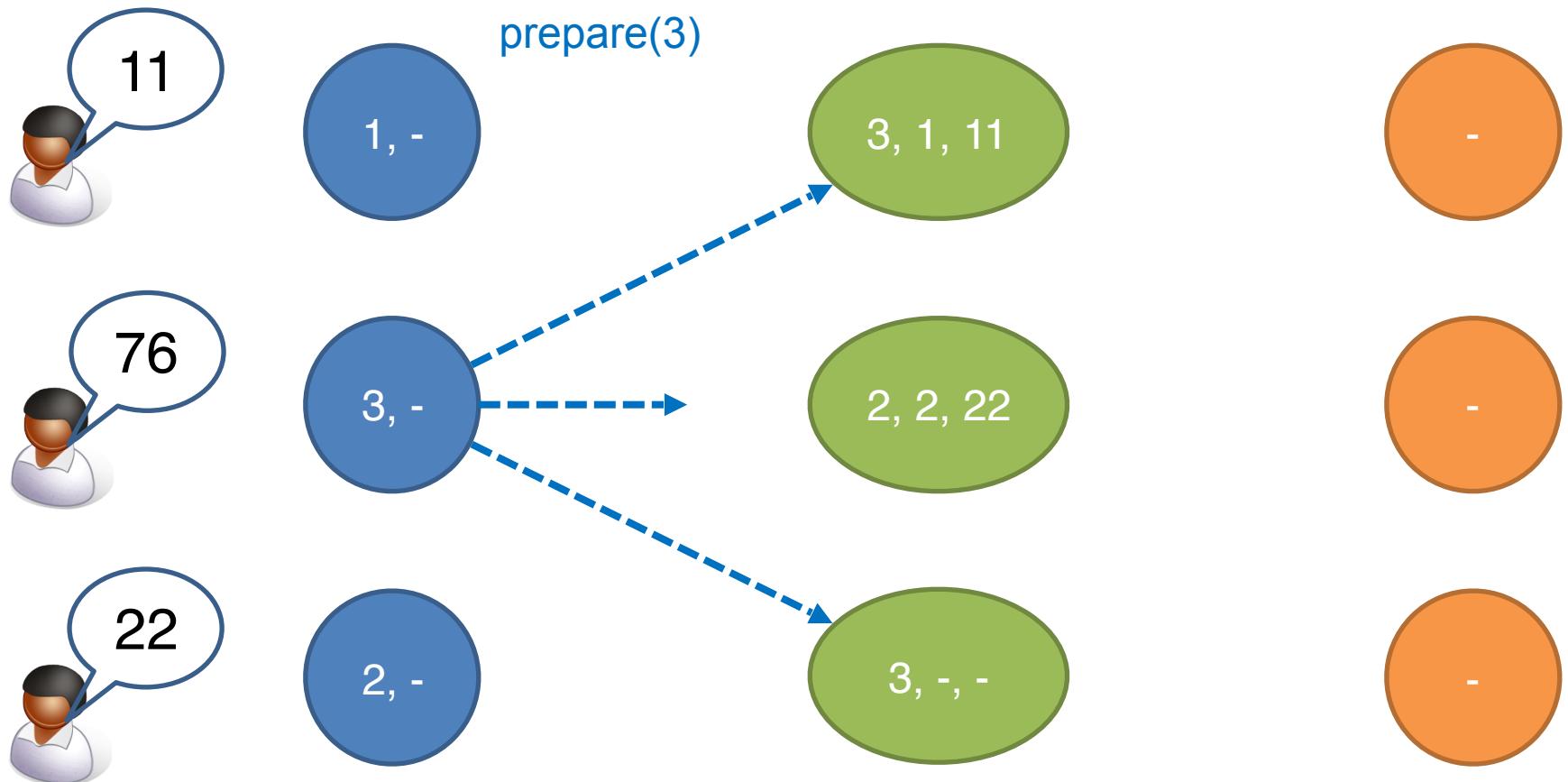
accepted(2, 22) message also send to all **proposers** (not shown here)



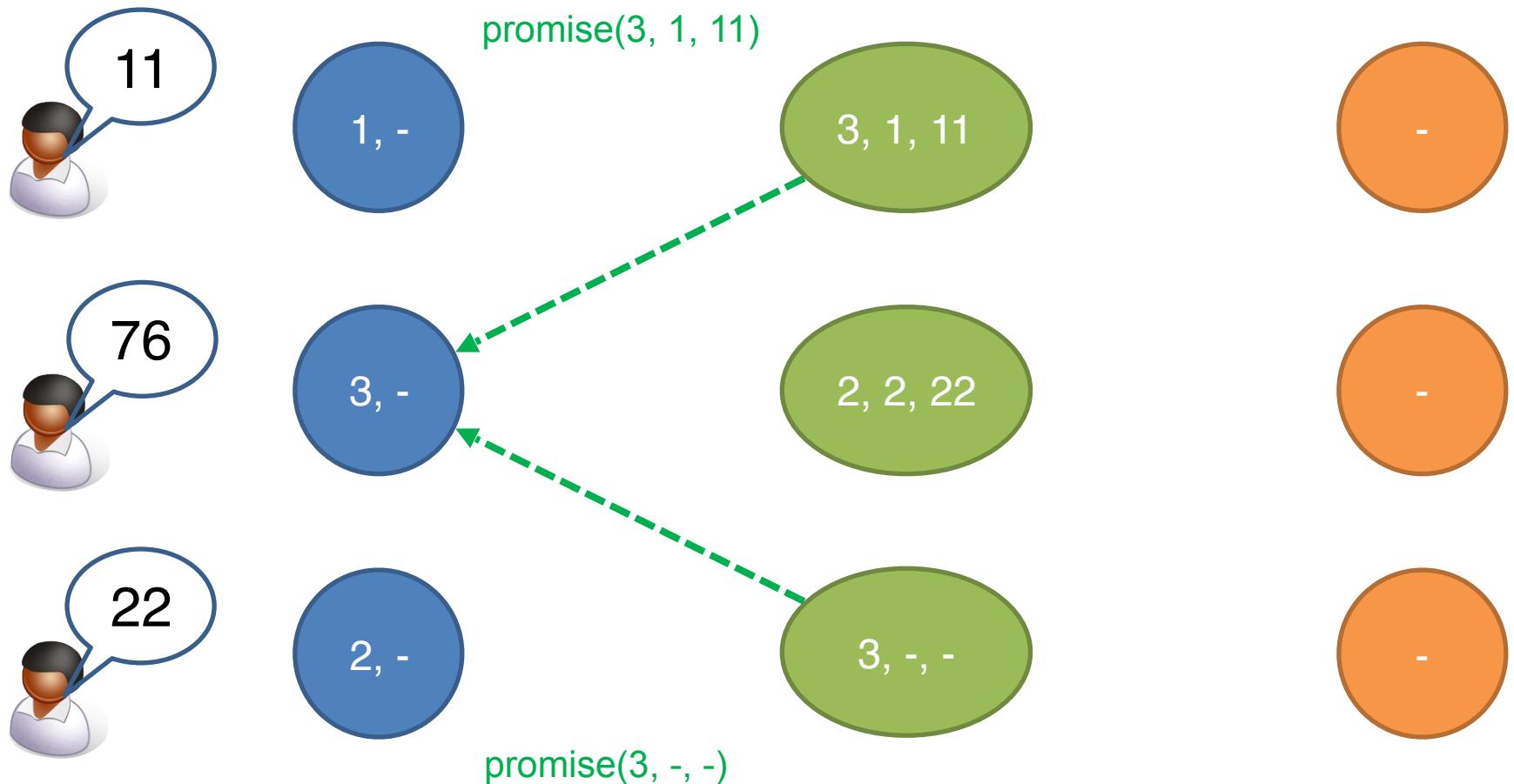
Basic Paxos with comm. failures



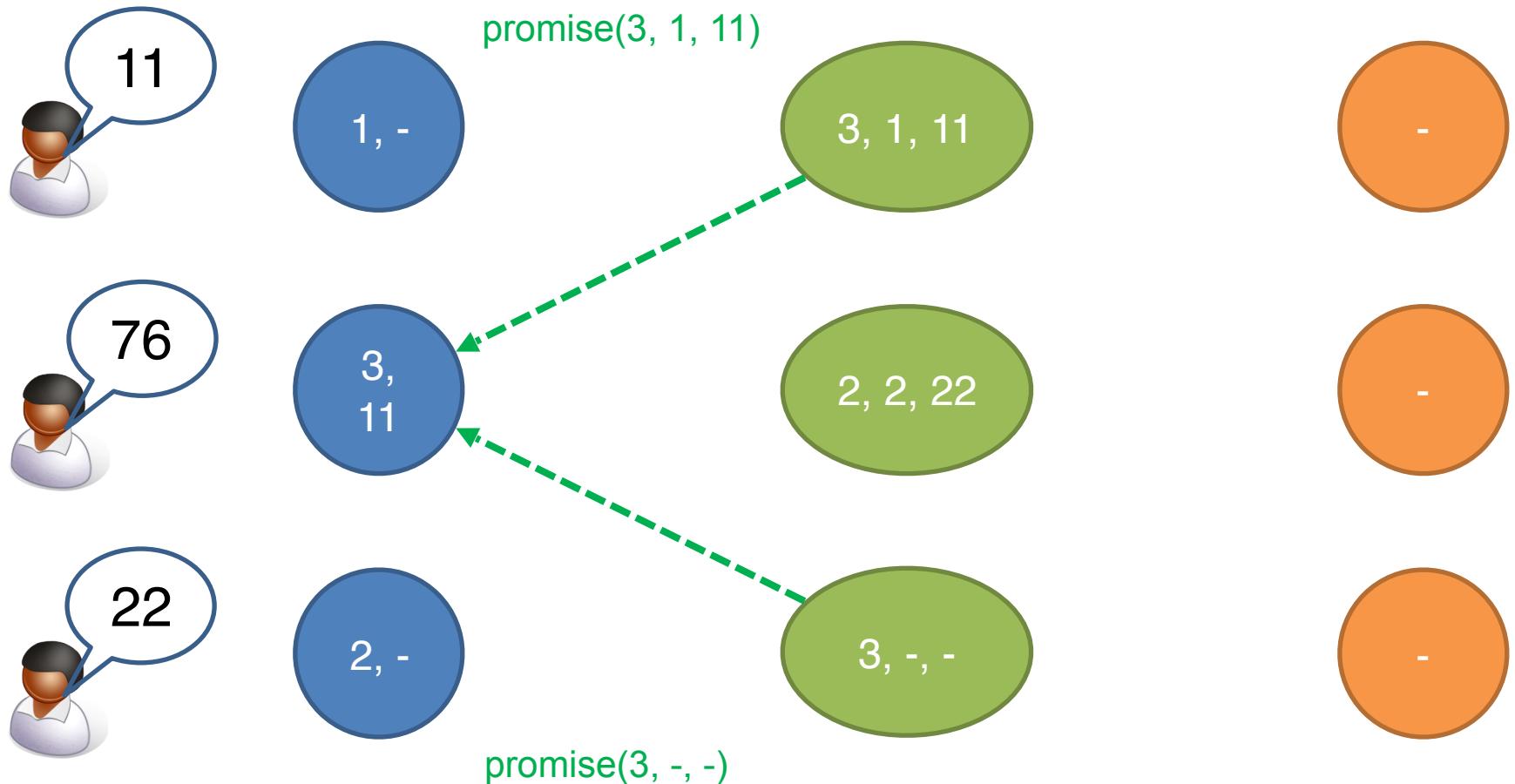
Basic Paxos with comm. failures



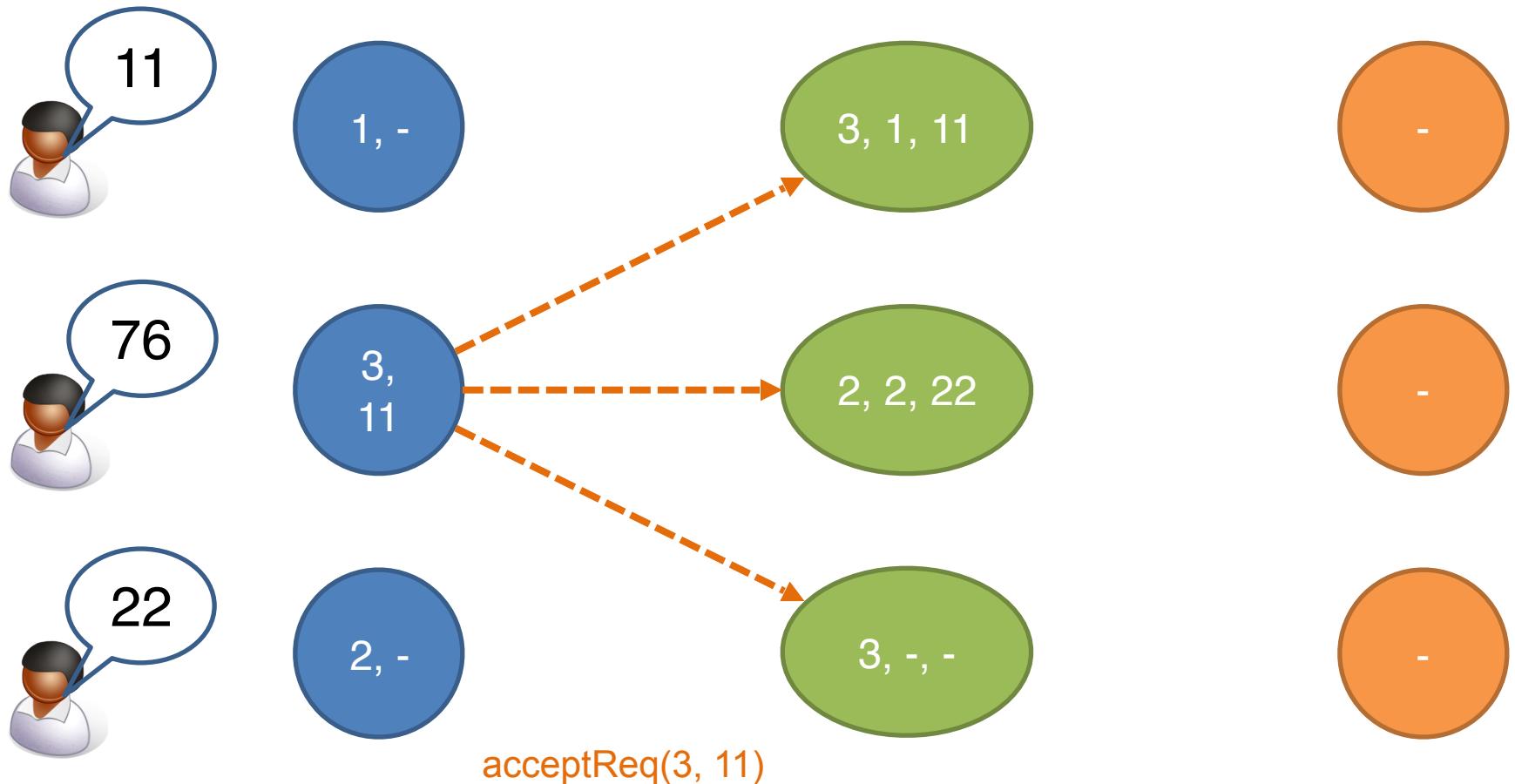
Basic Paxos with comm. failures



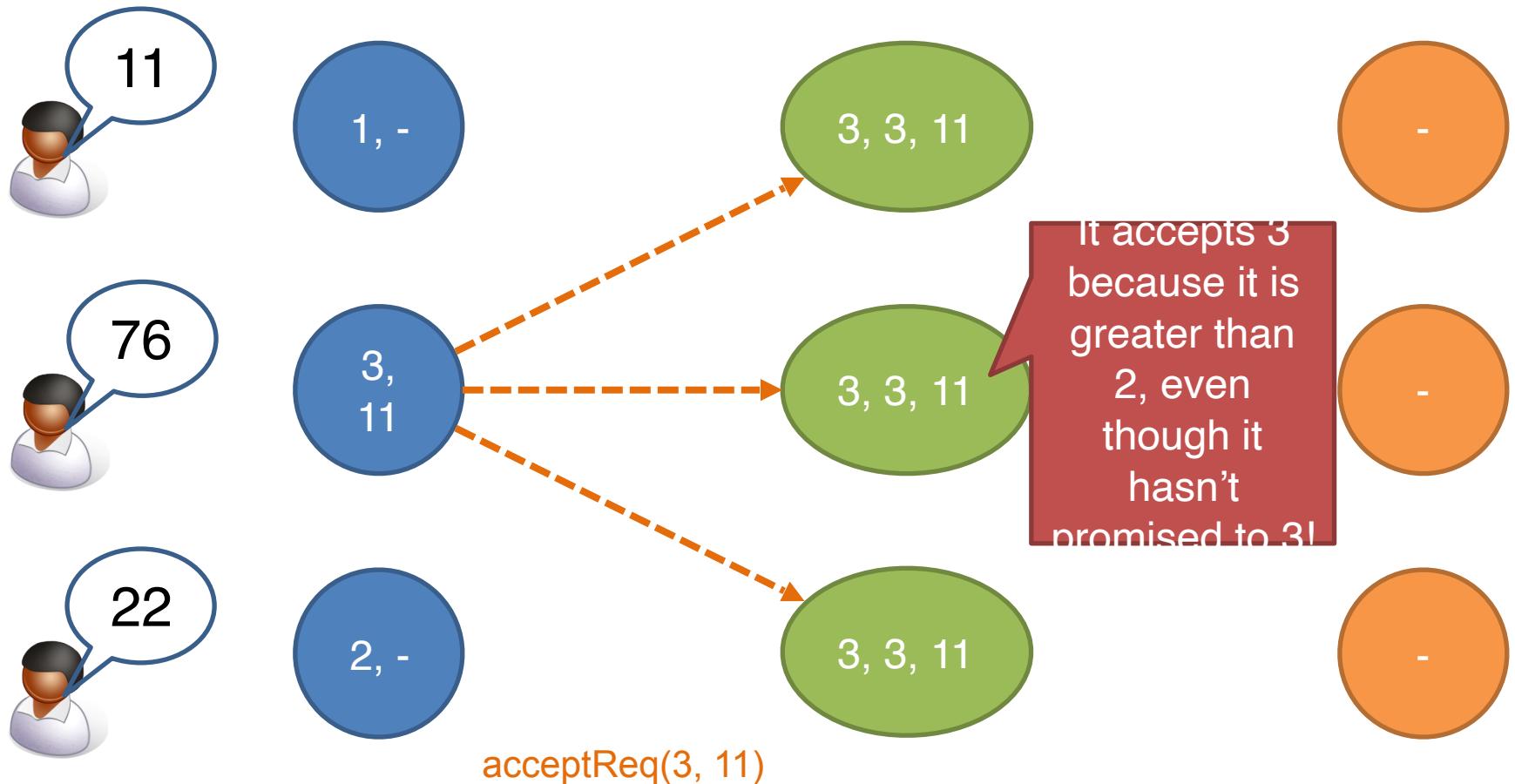
Basic Paxos with comm. failures



Basic Paxos with comm. failures

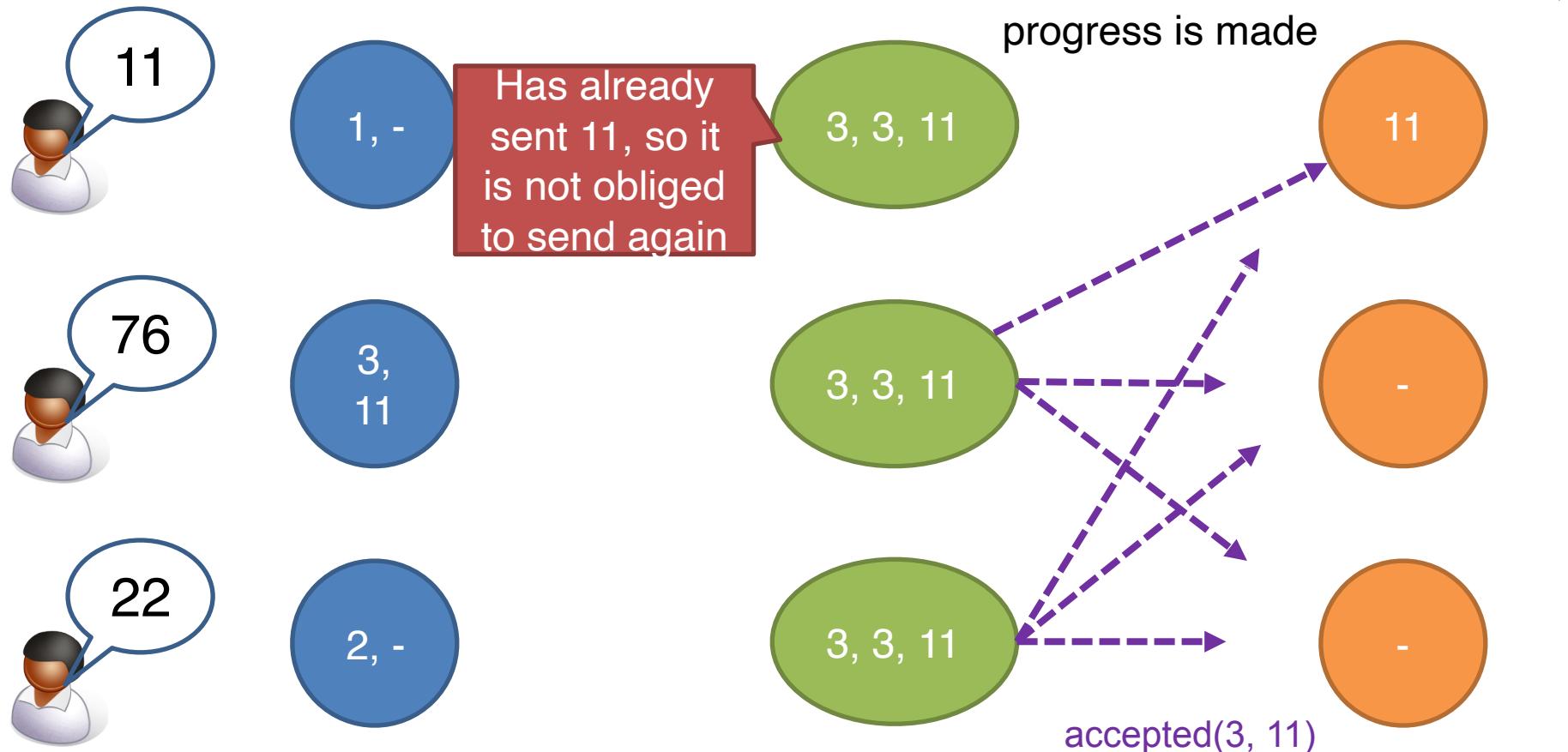


Basic Paxos with comm. failures

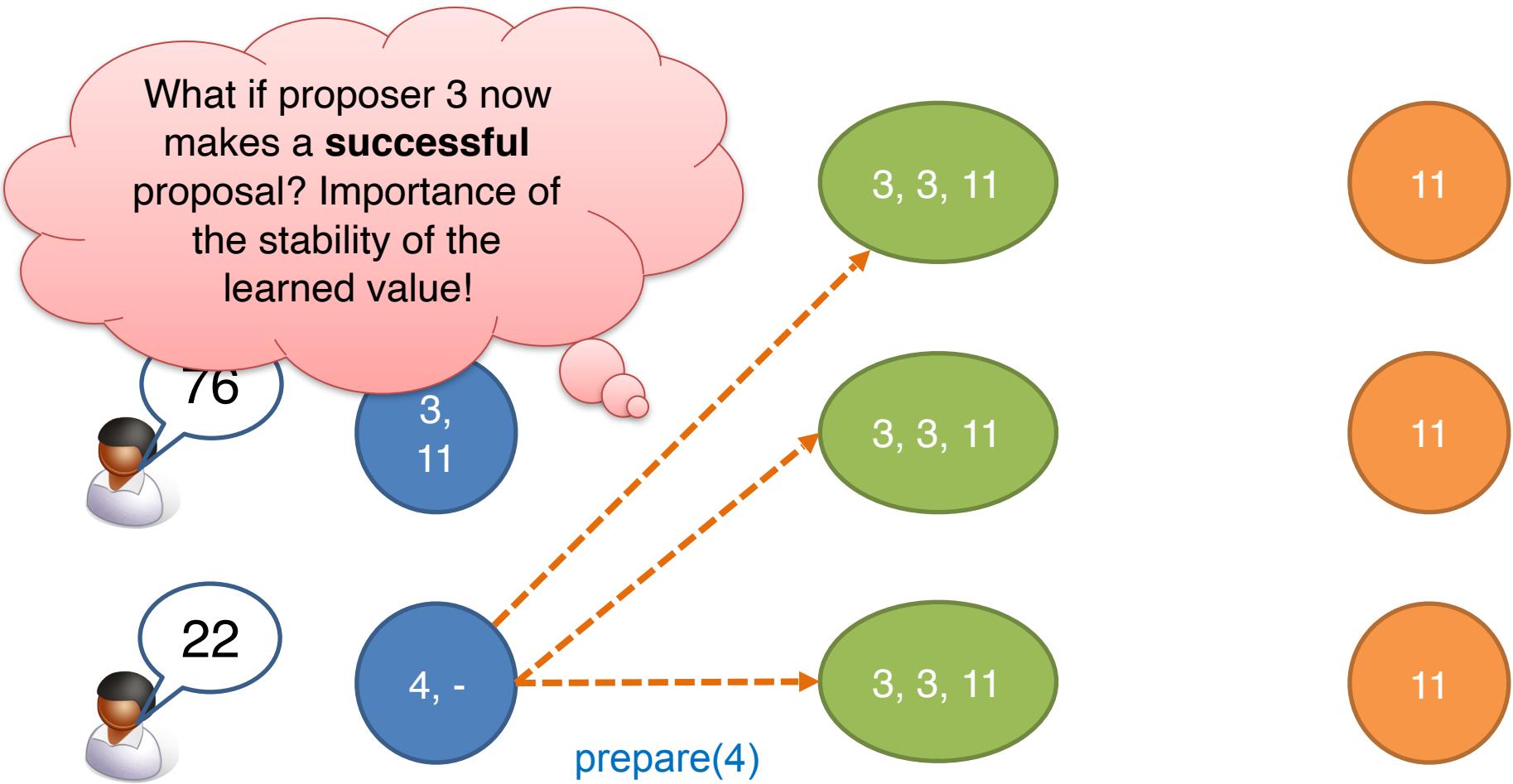


Basic Paxos with comm. failures

accepted(3, 11) message also send to all **proposers**
(not shown here)



Basic Paxos with comm. failures



“Paxos family”

- **Basic Paxos, Multi-Paxos, Cheap Paxos, Fast Paxos, Byzantine Paxos etc.**
- Protocols with **spectrum of trade-offs** between
 - Number of processes
 - Number of message delays before learning the agreed value
 - Activity level of individual participants
 - Number of messages sent
 - Types of failures tolerated

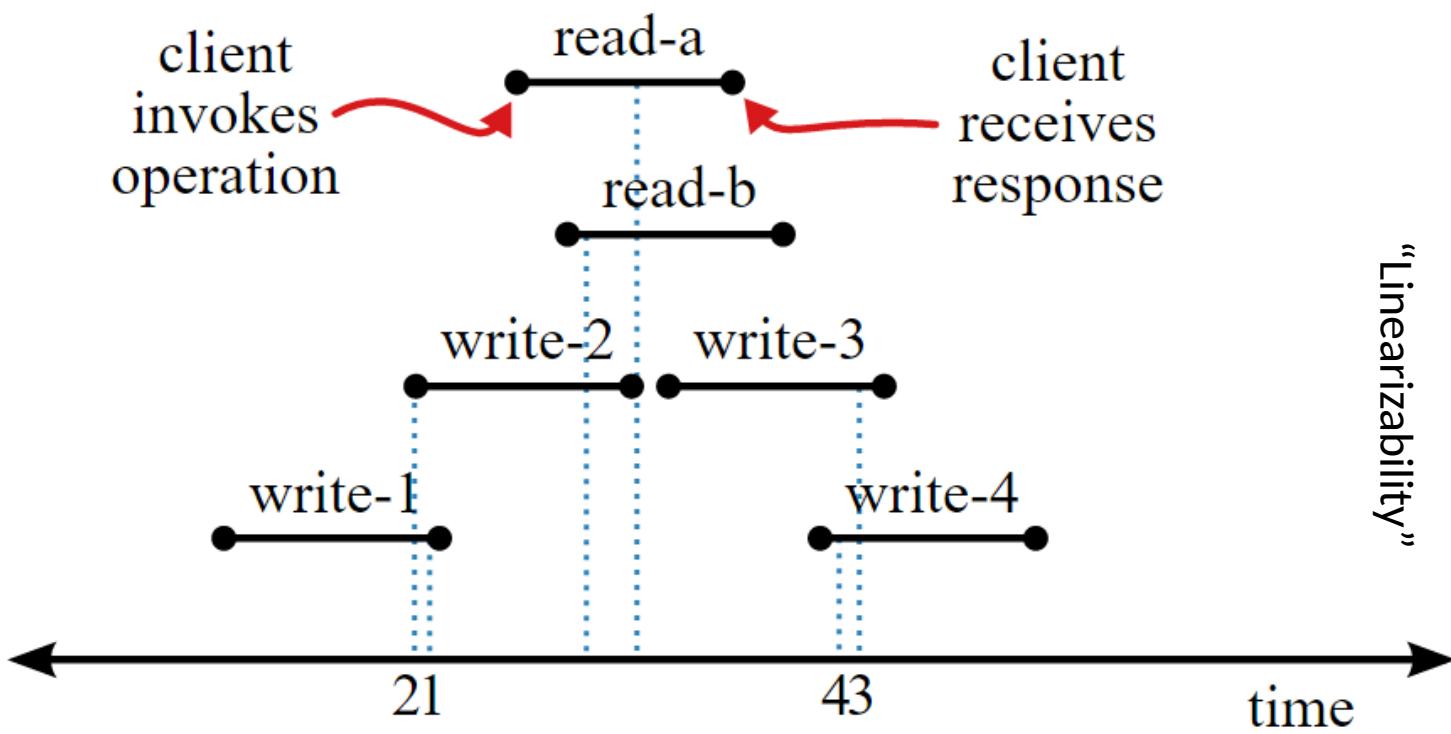
Discussion

- Configuration management not addressed by Basic Paxos (n assumed fix in protocol instance)
- Roles are coalesced in practice, i.e., same node acts a proposer, acceptor and learner
- A client request that does not become the result (consensus value) of an instance of Paxos will have to be retried by the client

Summary

- Consensus in an asynchronous environment with failures and unreliable network assuming $2f+1$ nodes
- Better than previously seen coordination algorithms
- Paxos guarantees safety, but progress may not hold
- Core mechanisms of Paxos are **ordering using proposal numbers** and **quorum accepts** without rollback (e.g., future proposals take older accepted values)
- Paxos is at the heart of Internet-scale systems deployed by major industry players (e.g., Chubby)
- Paxos protocol family has many members; here, Basic Paxos was reviewed (decides only on one value)

Replication & Consistency

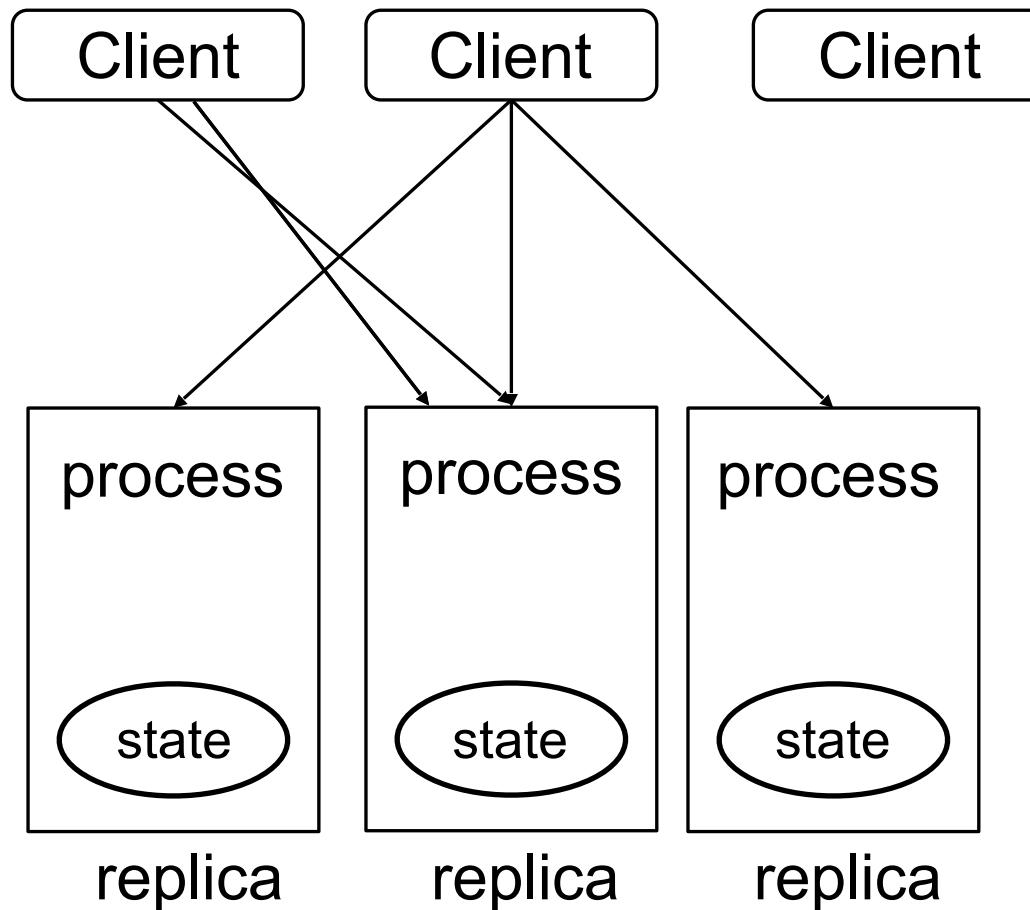


Agenda

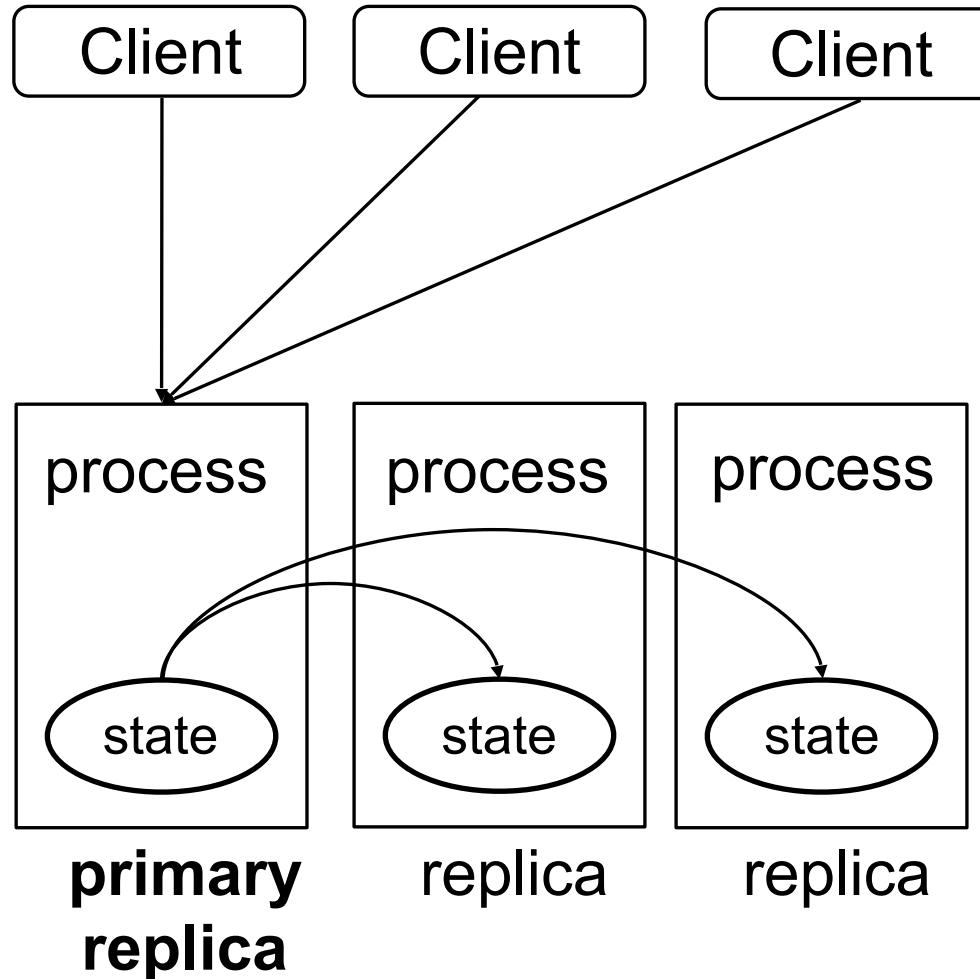
- Consistency models
- Data-centric consistency
- Client-centric consistency

CONSISTENCY MODELS

Active Replication

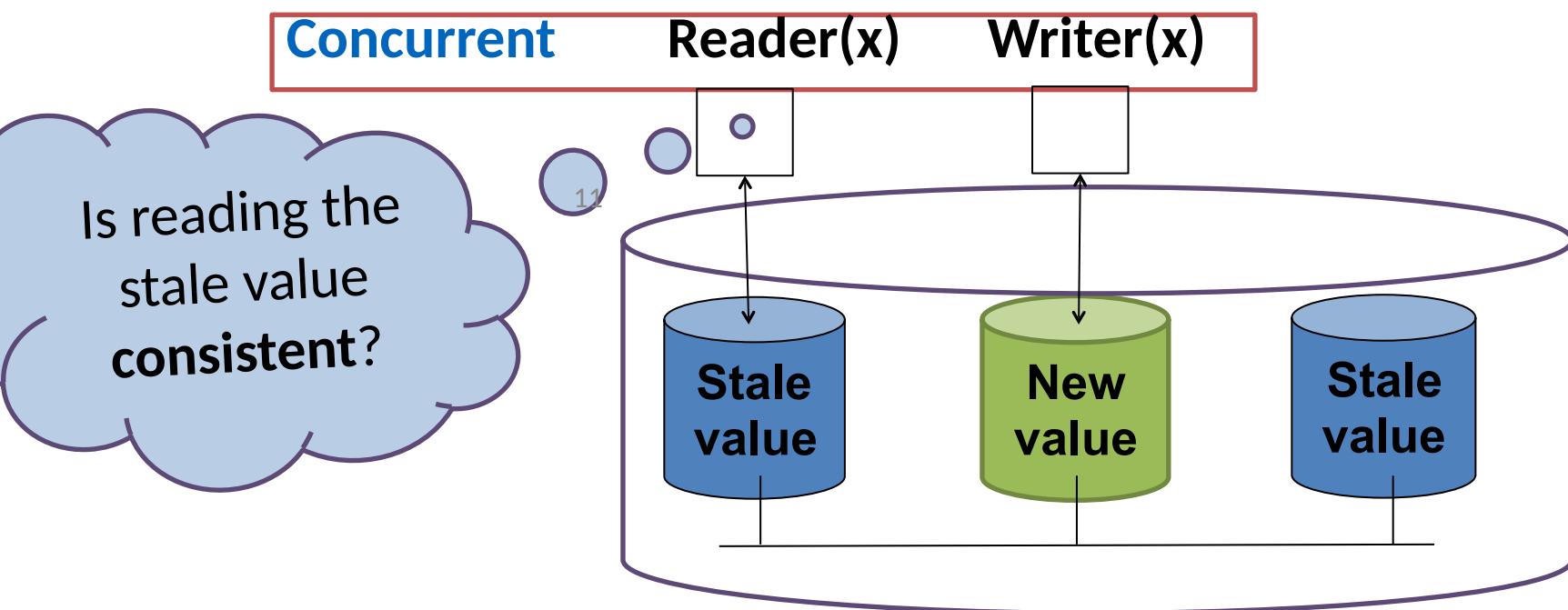


Passive Replication



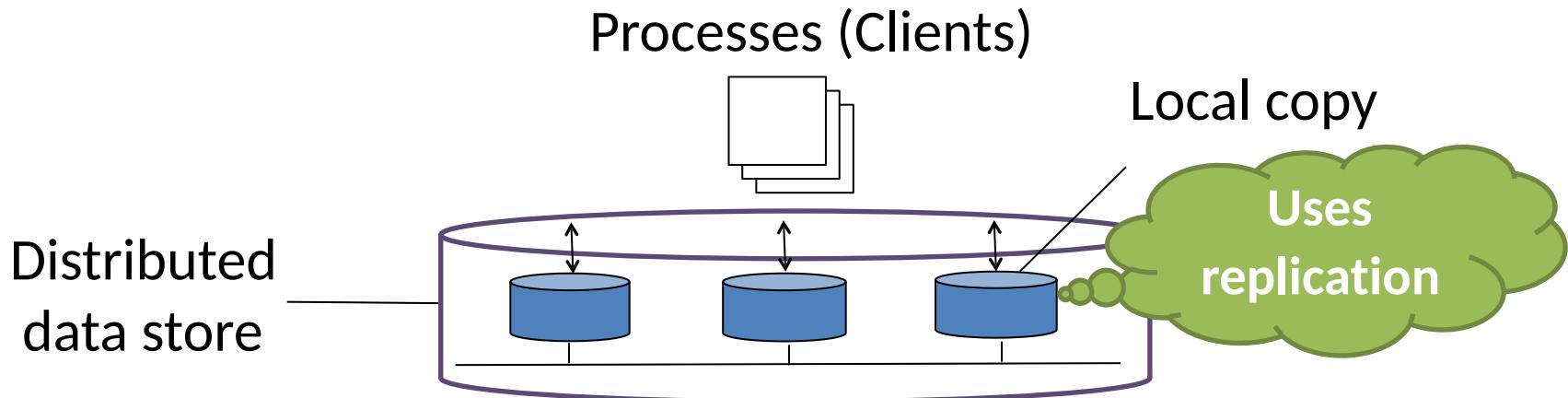
Concurrency and Replication

- All operations must be applied in a **specific order** to all replicas
- **Global ordering is too costly** and **not scalable** (e.g., using consensus, or single primary)
- **Solution:** Avoid global ordering using **weaker consistency requirements suitable** for the application



Consistency Models

- Definition (Data-centric Consistency model)
 - A **contract** between a distributed data store and a set of processes which specifies what the **results of concurrent read/write operations** are



Distributed data store as synonym for replicas, distributed database, shared memory, shared files, etc.

Data-Centric Consistency Models

- **Data-centric** consistency models dictate the outcome of concurrent reads and writes (r/w and w/w **conflicts**):
 - Strict consistency
 - Sequential consistency
 - Linearizable consistency
 - Causal consistency
 - FIFO consistency
 - Weak consistency

Strict Consistency

- **Definition:** Any *read* on a data item x returns a value corresponding to the result of the **most recent write** on x
- Uni-processor systems have traditionally observed strict consistency, ...
 - $a = 1; a = 2; \text{print}(a);$ **Output?**
- Definition assumes existence of **absolute global time** for unambiguous determination of "most recent".

Strict Consistency

- **Definition:** Any *read* on a data item x returns a value corresponding to the result of the **most recent write** on x
- Uni-processor systems have traditionally observed strict consistency, ... *but what about replicated systems?*
 - $a = 1; a = 2; \text{print}(a);$ **Output?**
- Definition assumes existence of **absolute global time** for unambiguous determination of "most recent".

Interpretation of Strict Consistency

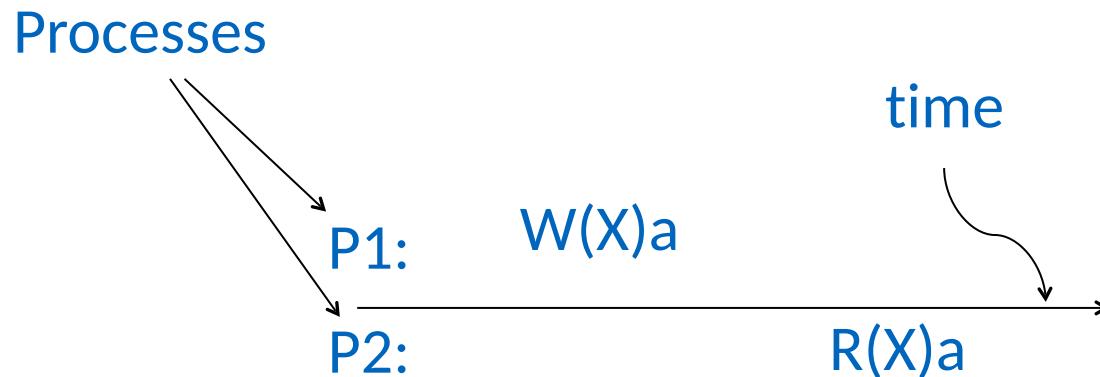
- Under strict consistent, all writes are **instantaneously visible to all processes** and **absolute global time order is maintained**
- If a **replica is updated**, ...
 - all subsequent reads, **see the new value**, no matter how soon after the update the reads are done
 - and no matter which process is doing the reading and where it is located
- Similarly, if a **read** is done, then it **gets the most recent value**, no matter how quickly the next write is done

Notation

$W(X)a$: Represents **writing** the value a to data X (*memory*)

$R(X)a$: Represents **reading** data X , which returns the value a

Initial value of X is ***NIL***



Strict Consistency Example

P1: $w(X)a$

P2: $\xrightarrow{R(X)a}$

P1: $w(X)a$

P2: $\xrightarrow{R(X)NIL \ R(X)a}$

Strict Consistency Example

P1: $W(X)a$
P2: $\frac{}{R(X)a}$



P1: $W(X)a$
P2: $\frac{}{R(X)NIL \ R(X)a}$

Strict Consistency Example

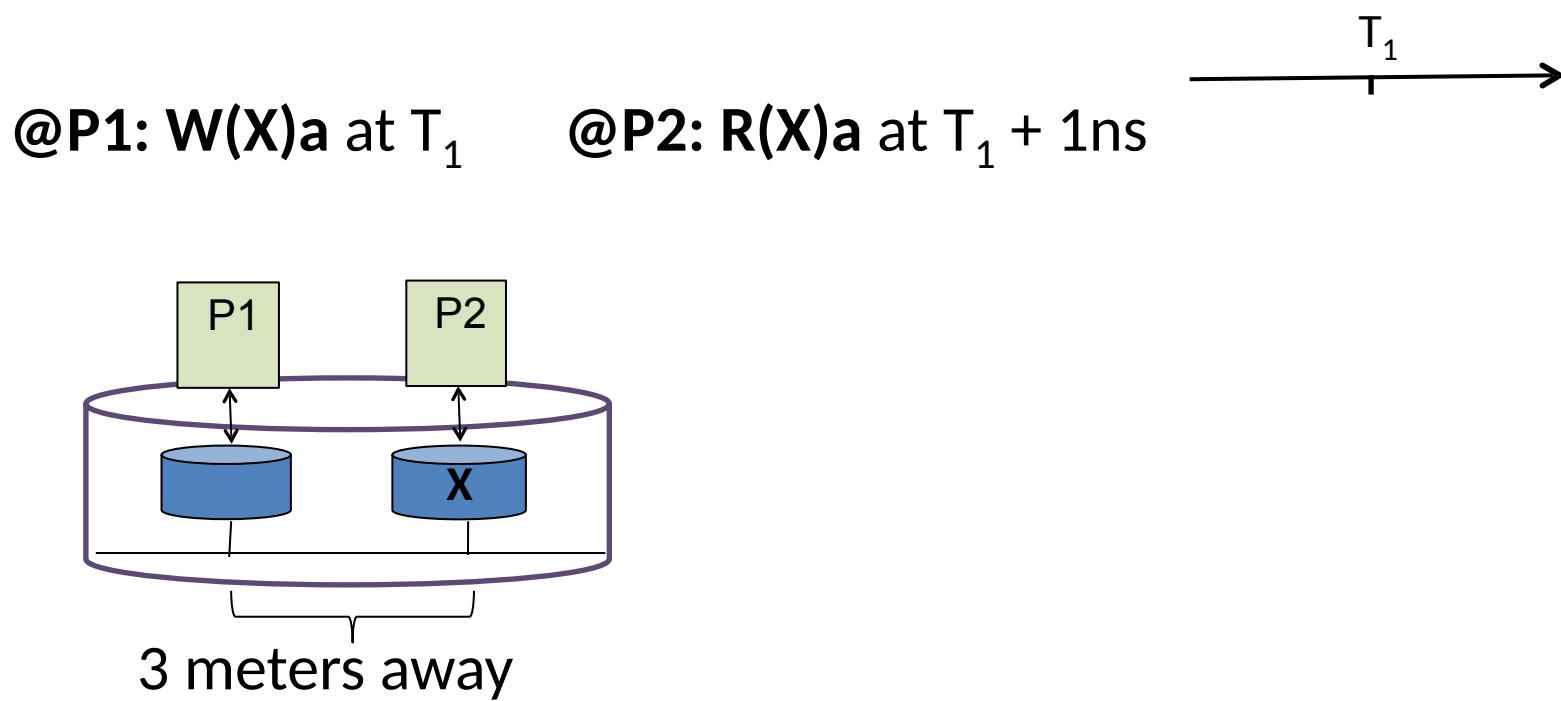
P1: $W(X)a$
P2: $\frac{}{R(X)a}$



P1: $W(X)a$
P2: $\frac{}{R(X)NIL \quad R(X)a}$



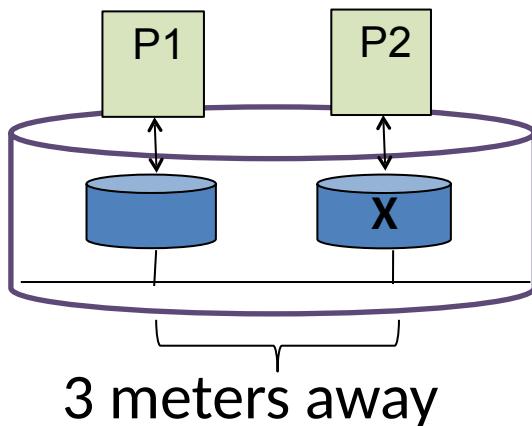
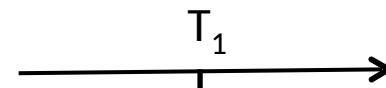
Strict Consistency: Thought Experiment



Strict Consistency: Thought Experiment

- To satisfy strict consistency, laws of physics may have to be violated! Obviously, not an option!!!
- Example:

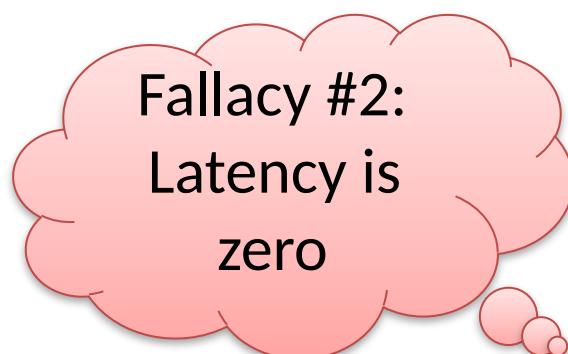
@P1: $W(X)a$ at T_1 @P2: $R(X)a$ at $T_1 + 1\text{ns}$



To realize strict consistency in this case, $W(X)a$ would have to travel at 10 times the speed of light!

Strict Consistency: The Bad News

- It is **impossible** to perfectly synchronize clocks
 - How to accurately determine the time of each operation?
- It is **impossible** to instantaneously replicate operations



H.-A. Jacobsen



Definition of Sequential Consistency

- The result of any execution is the same as if the 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**
- Weaker than strict consistency: **logical time** instead of **physical time**

Sequential Consistency Example

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)b$ $R(x)a$

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

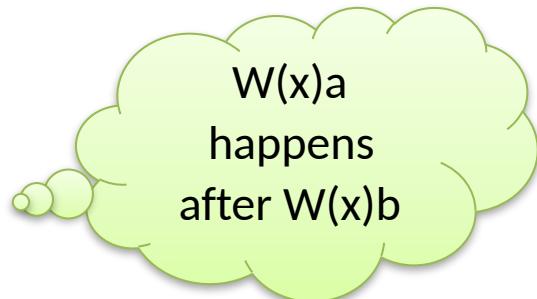
Sequential Consistency Example

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$

P4:  $R(x)b$ $R(x)a$



P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$

P4: $R(x)a$ $R(x)b$

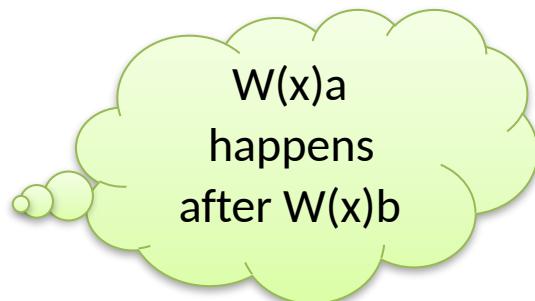
Sequential Consistency Example

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$

P4:  $R(x)b$ $R(x)a$



Cyclic
dependency
between the
writes!

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$

P4:  $R(x)a$ $R(x)b$

Definition of Linearizability

- 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.
- In addition, if $ts_{OP_1}(x) < ts_{OP_2}(y)$, then **operation $OP_1(x)$ should precede $OP_2(y)$** in this sequence
- **$ts_{OP}(x)$ denotes the timestamp assigned to operation OP that is performed on data item x and OP is either read (R) or write (W).**

Time stamp is modeled as a duration.

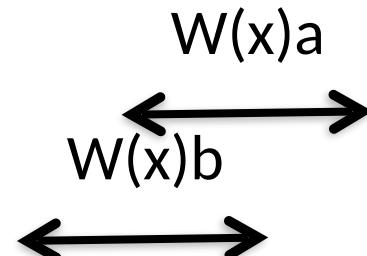
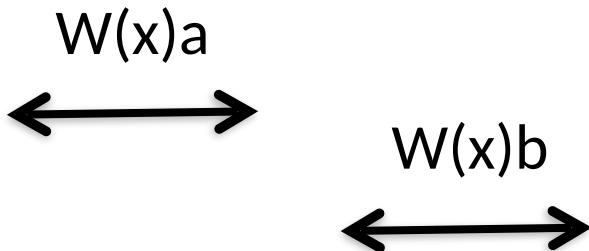
Definition of Linearizability

- 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 **Sequential consistency**: operations of each individual process appear in this sequence in the order specified by its program.
- In addition, if $ts_{OP_1}(x) < ts_{OP_2}(y)$, then **operation $OP_1(x)$ should precede $OP_2(y)$** in this sequence
- $ts_{OP}(x)$ denotes the timestamp assigned to operation OP that is performed on data item x , and OP is either read (R) or write (W).**

Time stamp is modeled as a duration.

Linearizability

- Like strict consistency, **assumes global time**, but not absolute global time
- Assumes processes in the system have **physical clocks** synchronized to within an **bounded error** (captured by duration timestamp)
- If $W(x)b$ was the **most recent** write operation and there is no other write operation **overlapping** with $W(x)b$, then any **later** read should return b
- If $W(x)a$ and $W(x)b$ were two **most-recent overlapping** write operation, then **any later read** should **return either a or b** , not something else

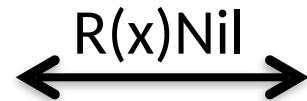


Linearizability

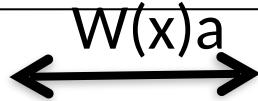
- A linearizable data store is also **sequentially consistent**
- I.e., linearizability is more restrictive (**stronger**)
- Difference is the ordering according to a set of synchronized clocks
- Linearizability prevents **stale reads**, since it guarantees correct reads after a write is completed

Example 1

P1:



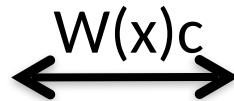
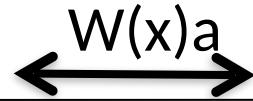
P2:



P3:



P1:



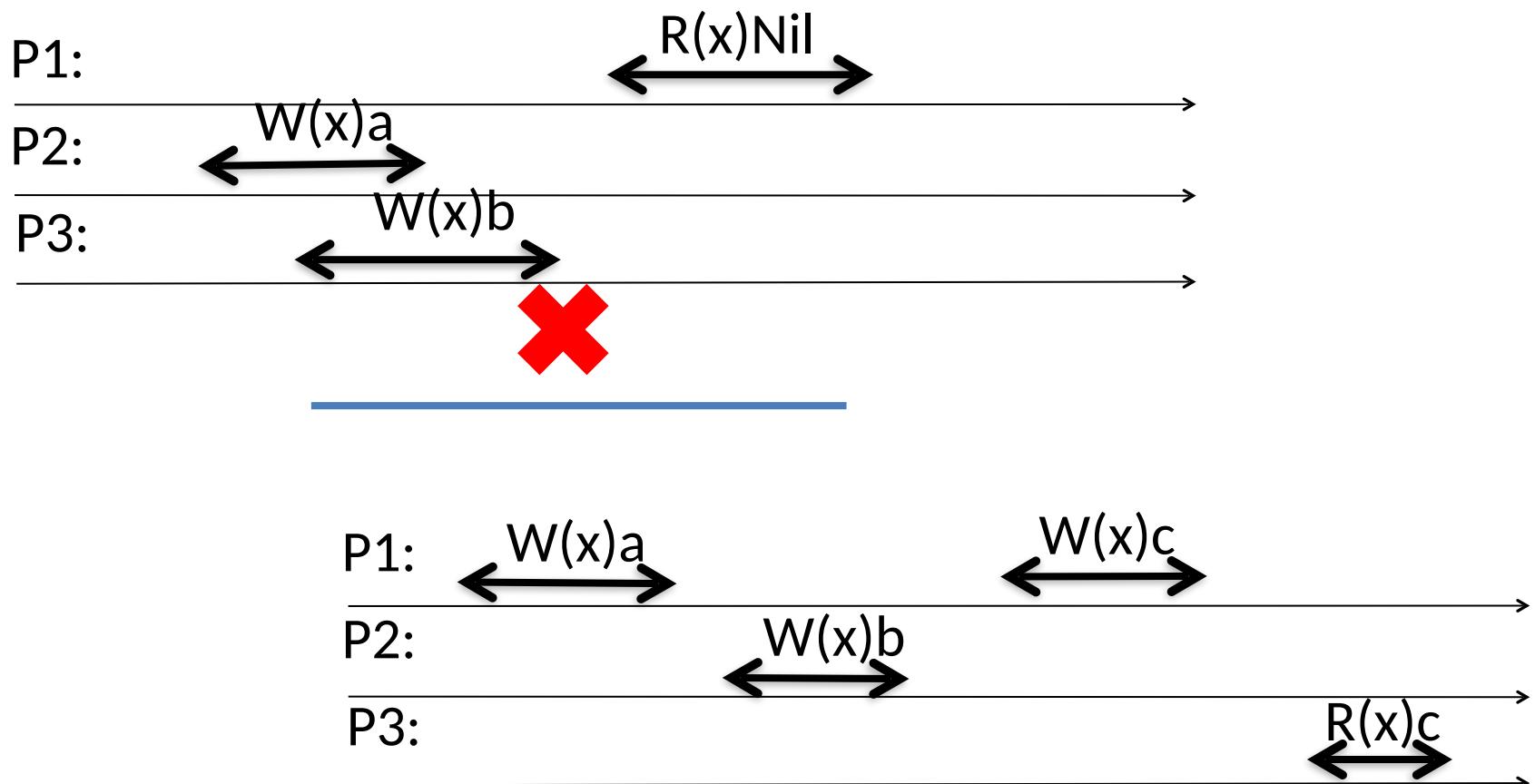
P2:



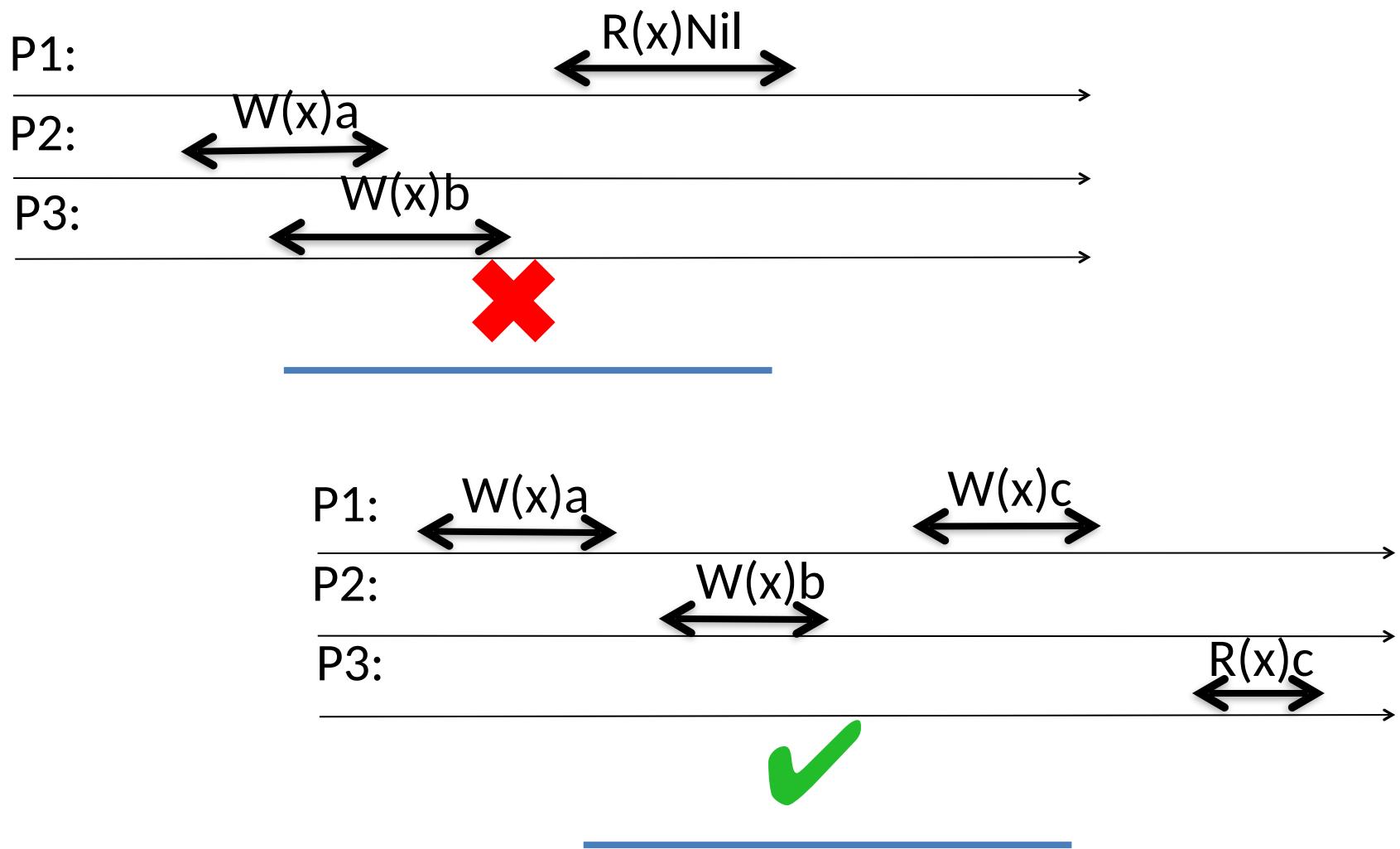
P3:



Example 1



Example 1



Example 2

P1:

$$\xleftarrow{\quad} \xrightarrow{\quad} R(x)a$$

P2:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)a$$

P3:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)b$$



P1:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)a$$

$$\xleftarrow{\quad} \xrightarrow{\quad} R(x)b$$

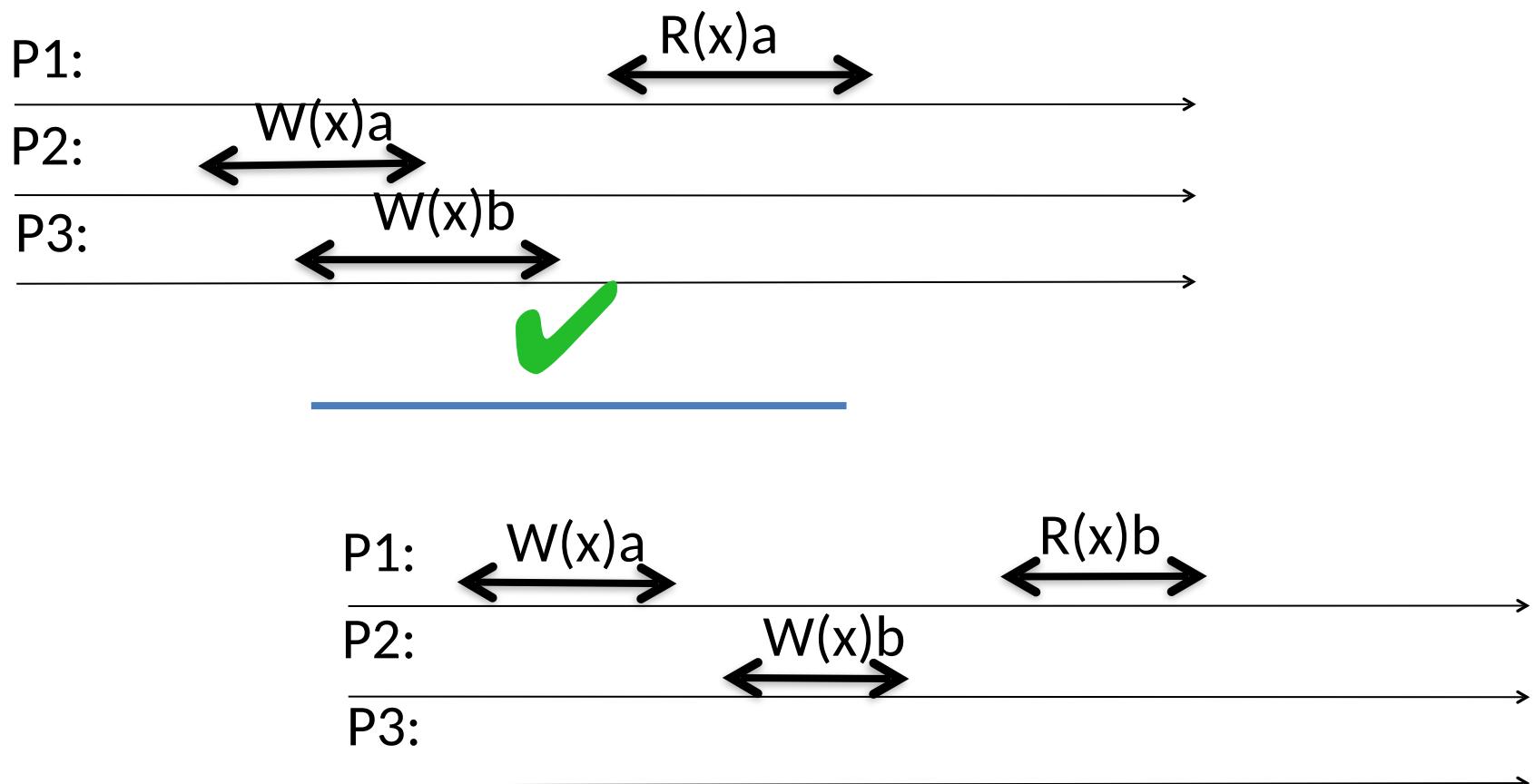
P2:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)b$$

P3:



Example 2



Example 2

P1:

$$\xleftarrow{\quad} \xrightarrow{\quad} R(x)a$$

P2:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)a$$

P3:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)b$$



P1:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)a$$

$$\xleftarrow{\quad} \xrightarrow{\quad} R(x)b$$

P2:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)b$$

P3:



Example 3

P1:

$$\xleftarrow{\quad} \xrightarrow{\quad} R(x)b$$

P2:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)a$$

P3:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)b$$



P1:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)a$$

$$\xleftarrow{\quad} \xrightarrow{\quad} R(x)a$$

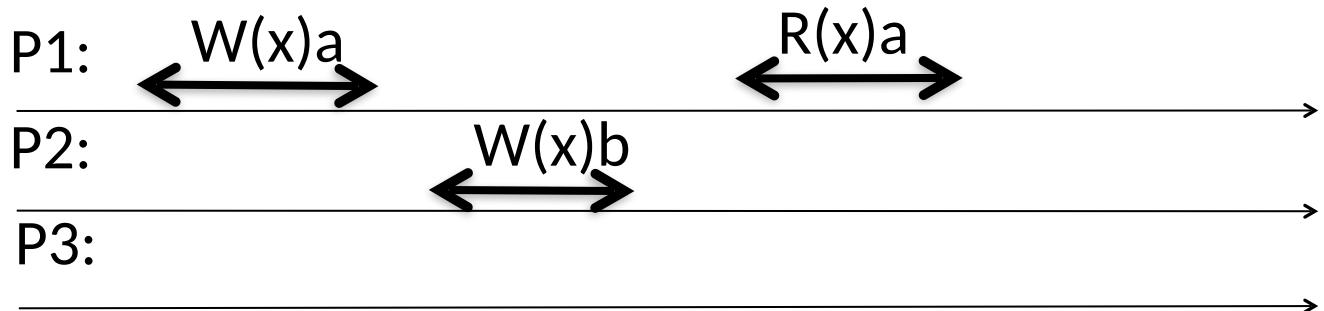
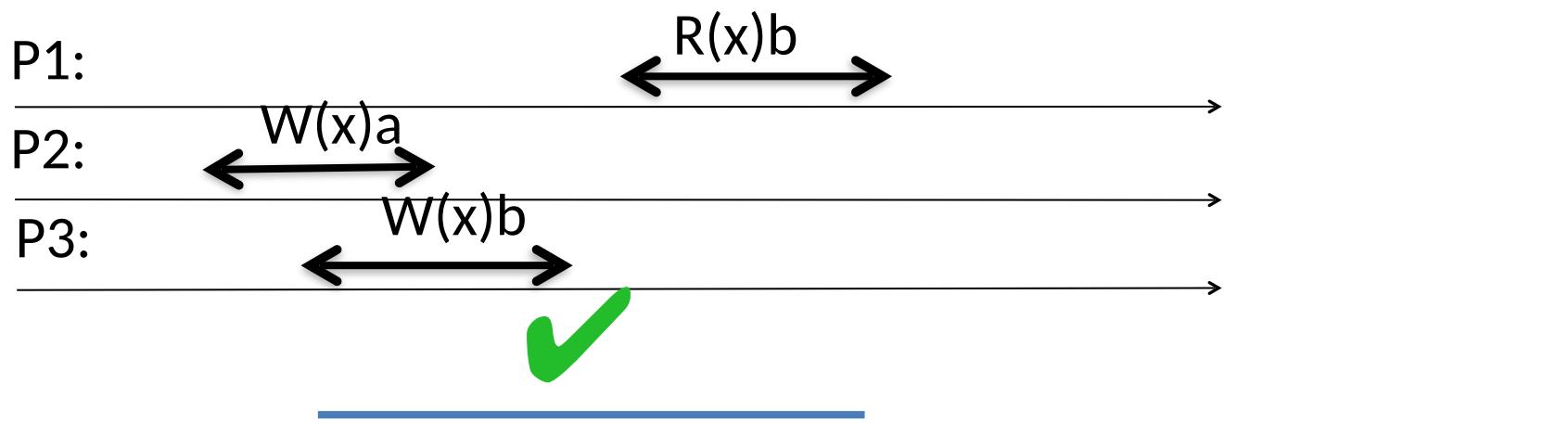
P2:

$$\xleftarrow{\quad} \xrightarrow{\quad} W(x)b$$

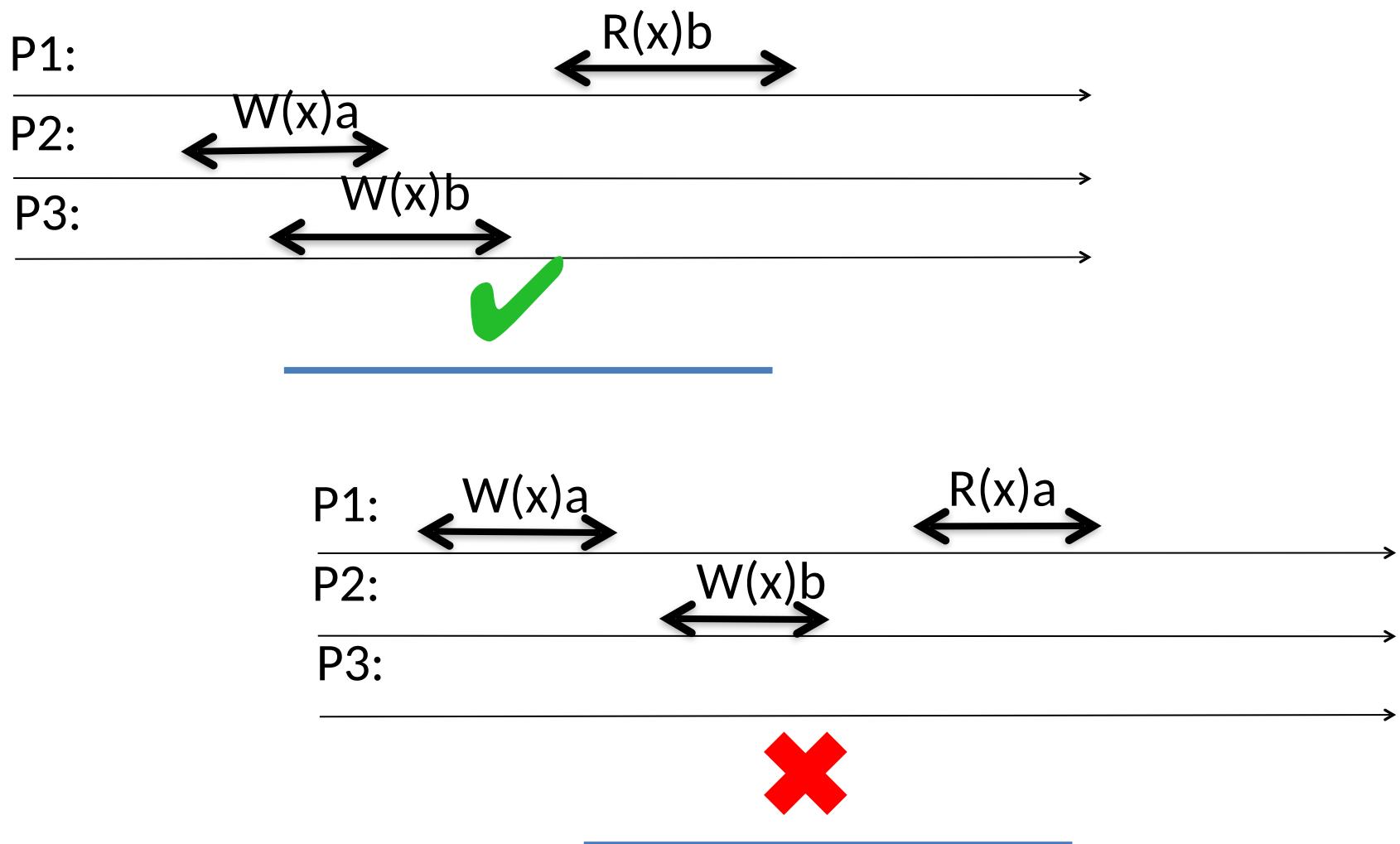
P3:



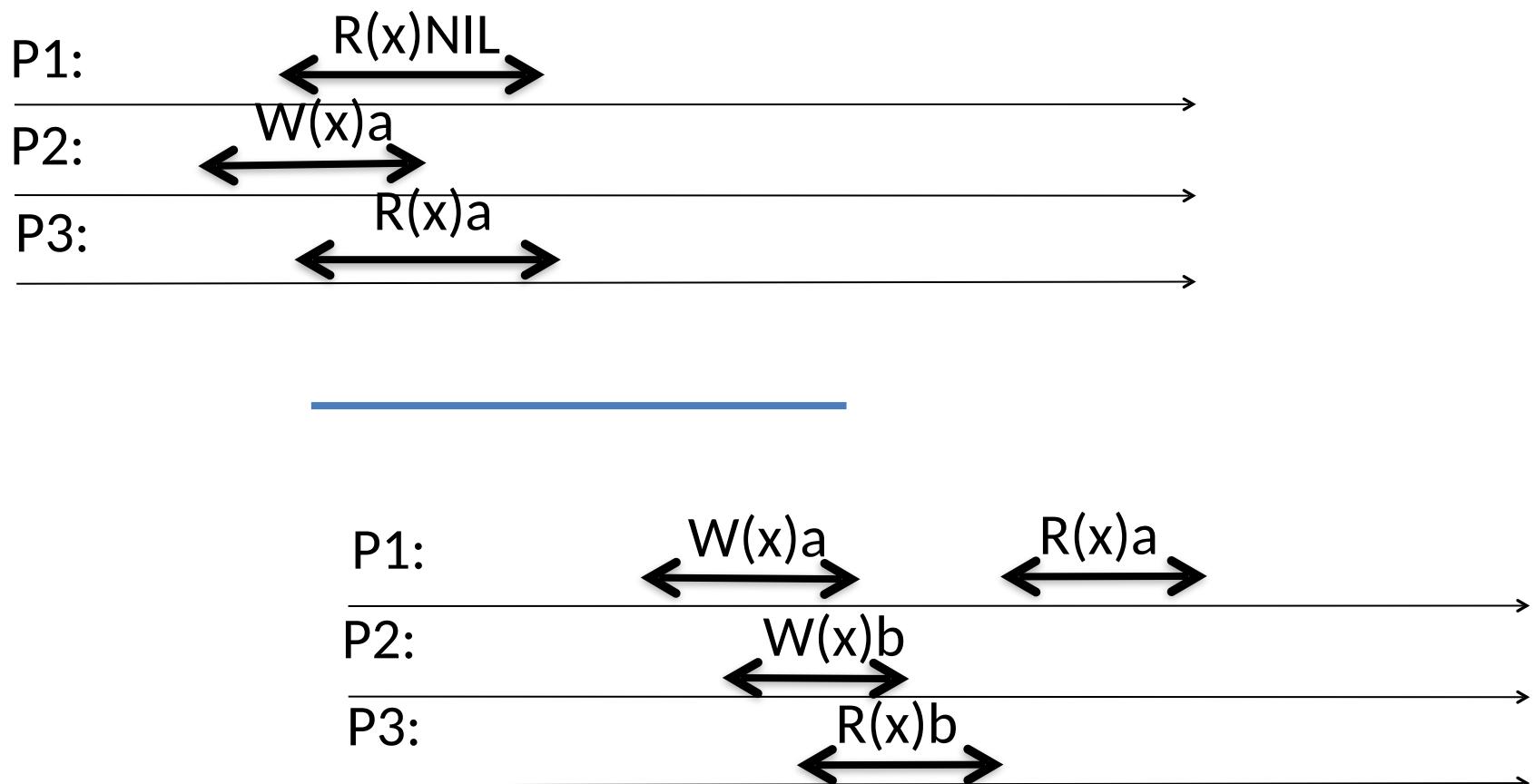
Example 3



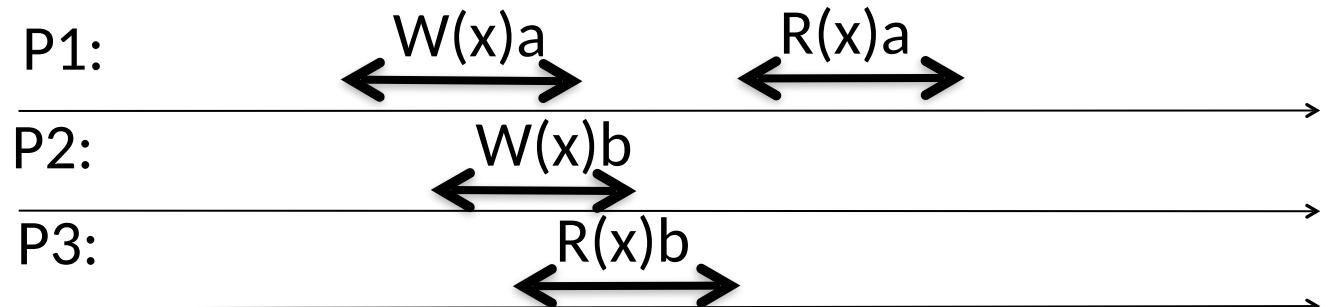
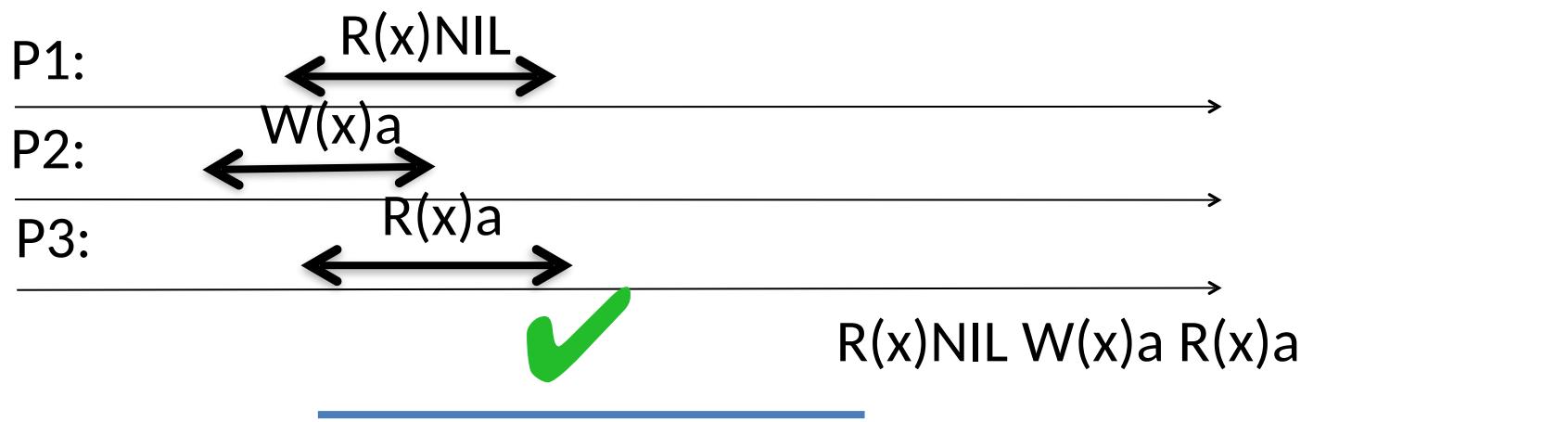
Example 3



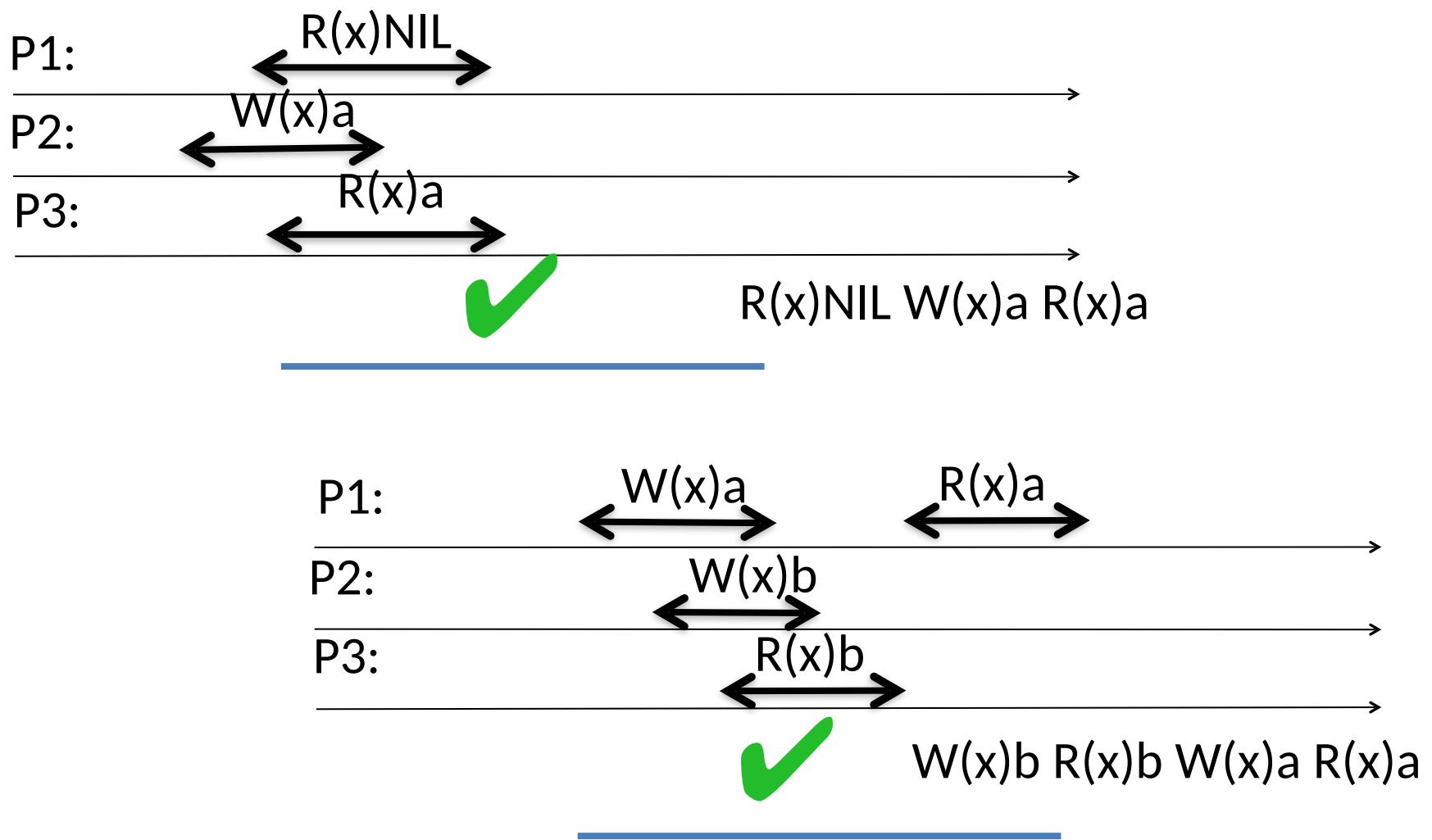
Example 4



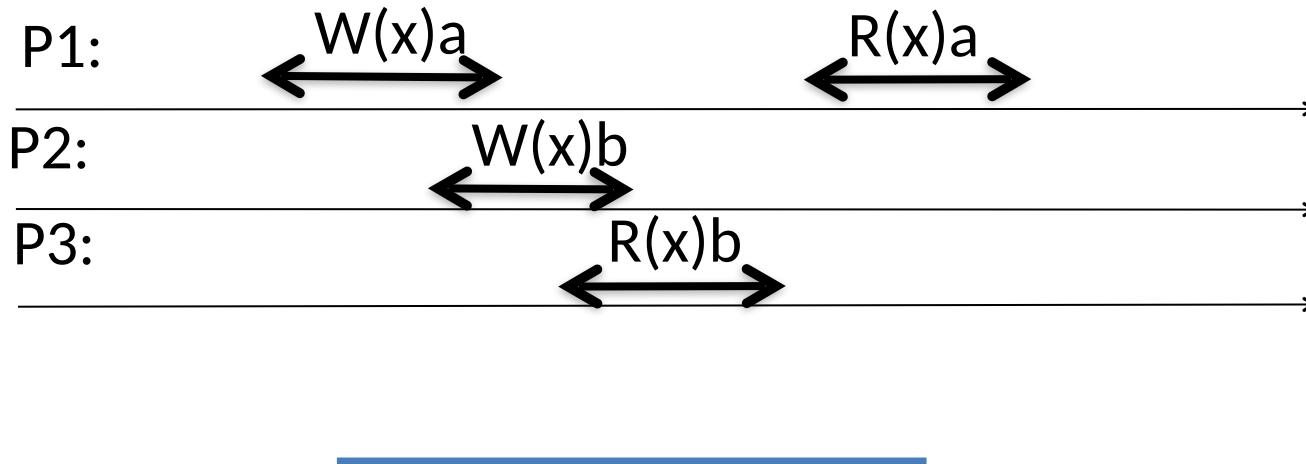
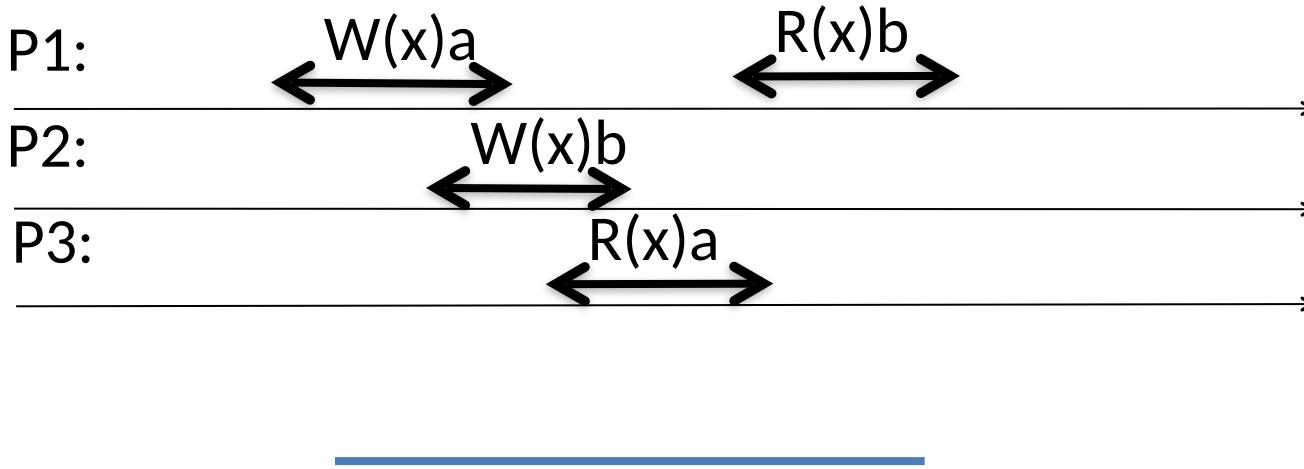
Example 4



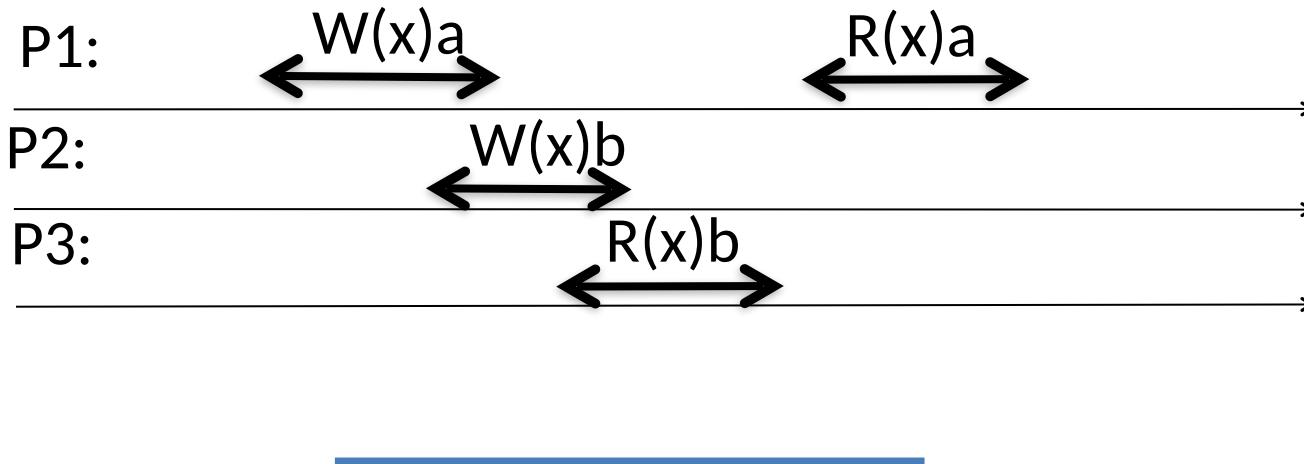
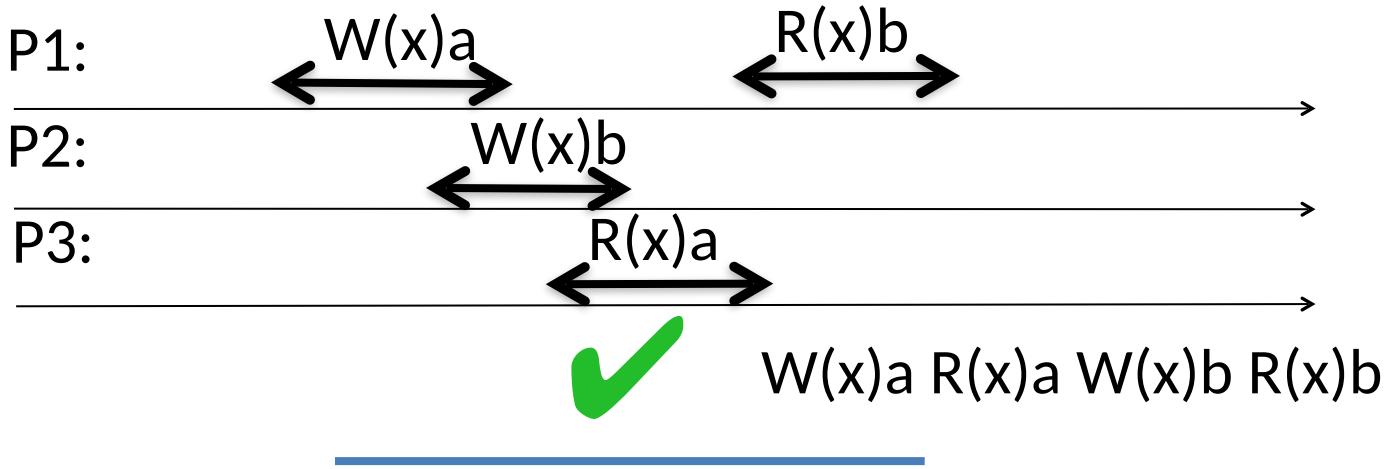
Example 4



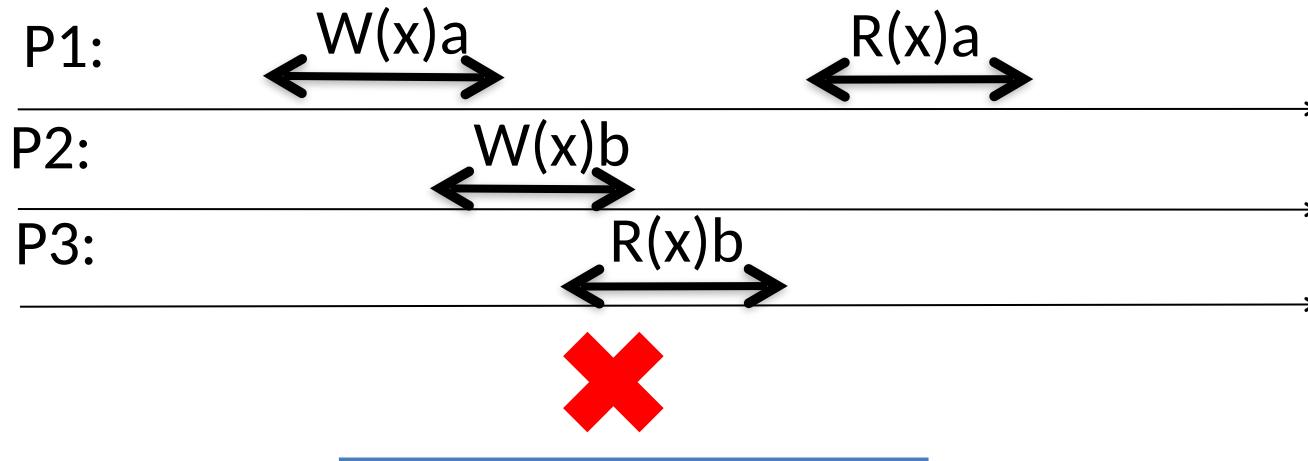
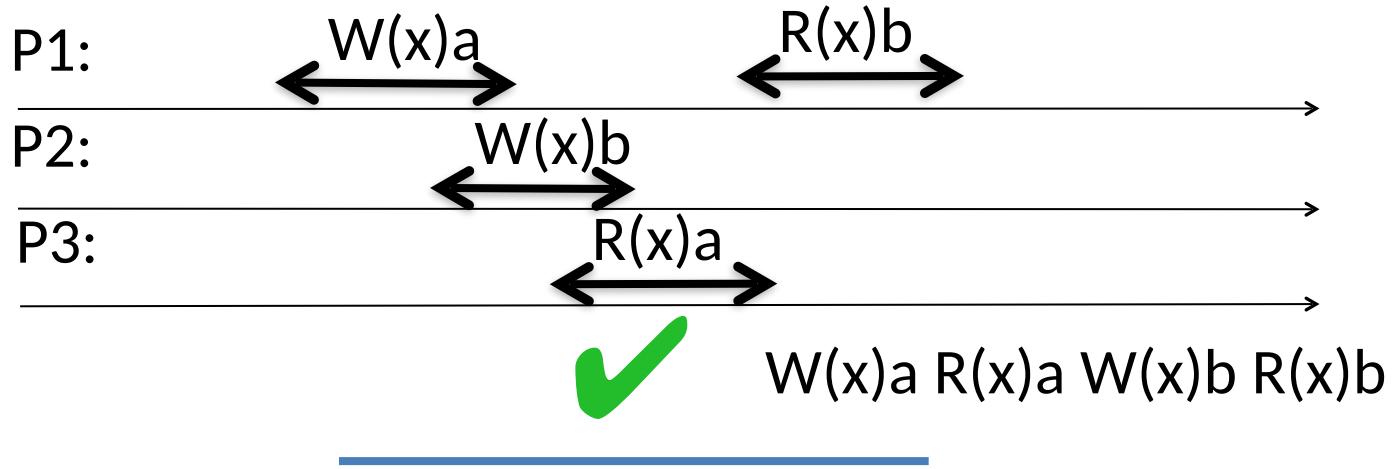
Example 5



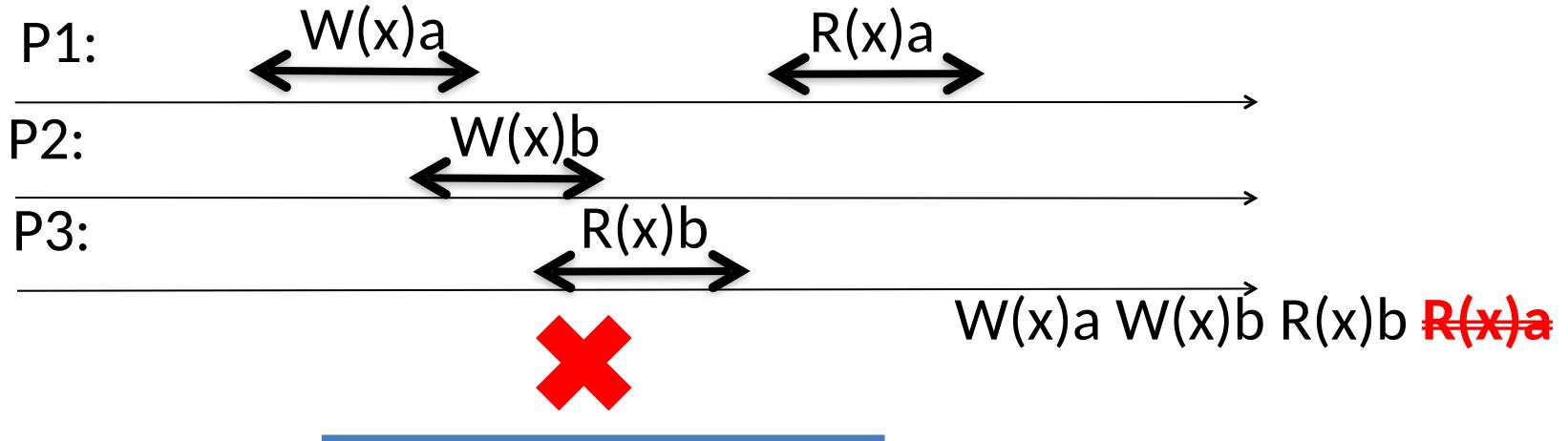
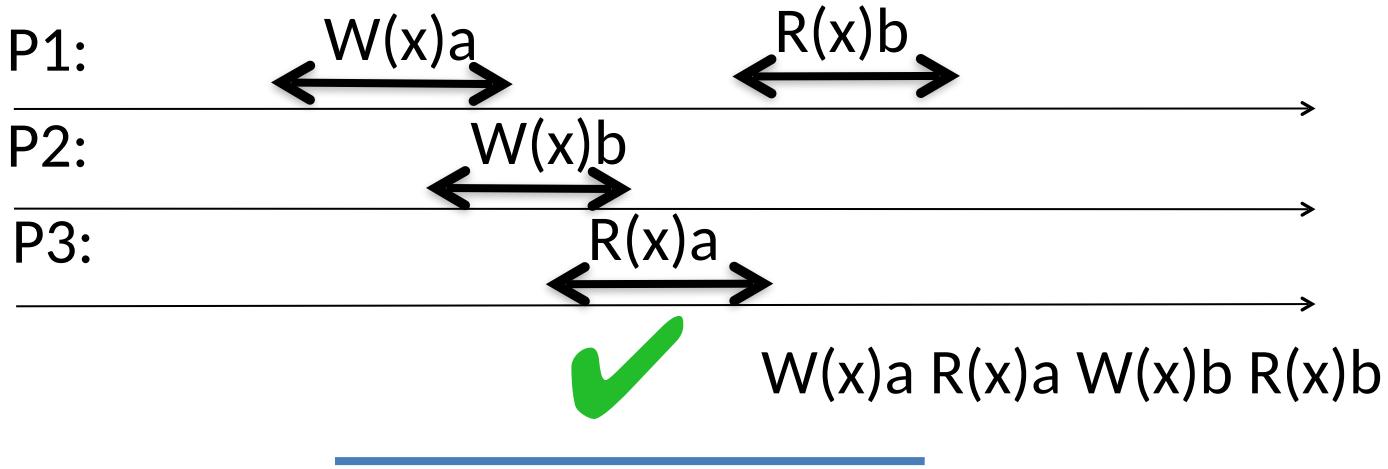
Example 5



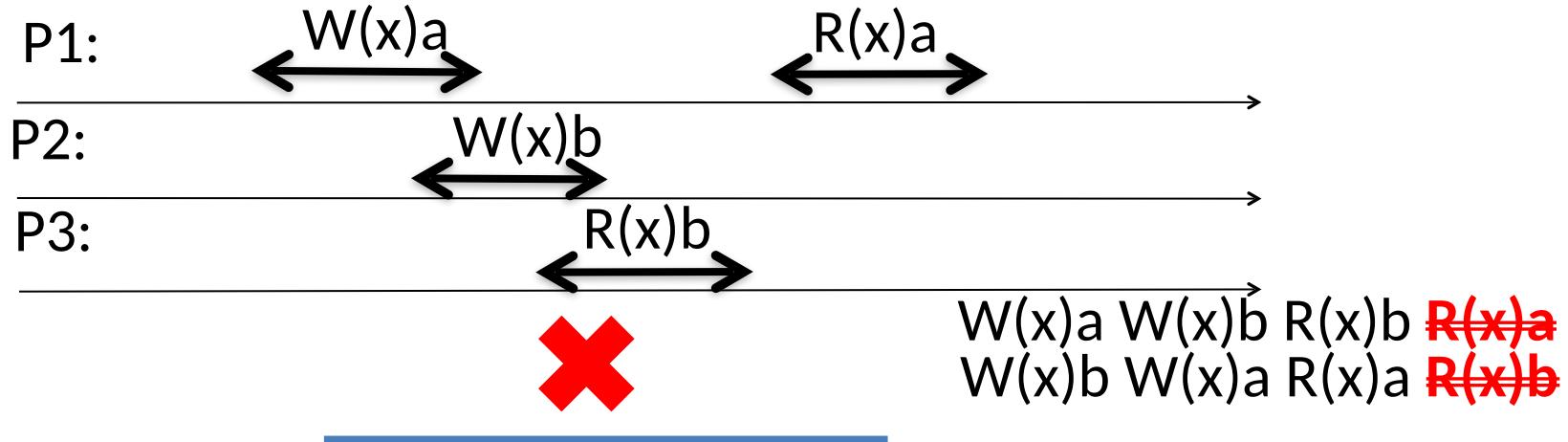
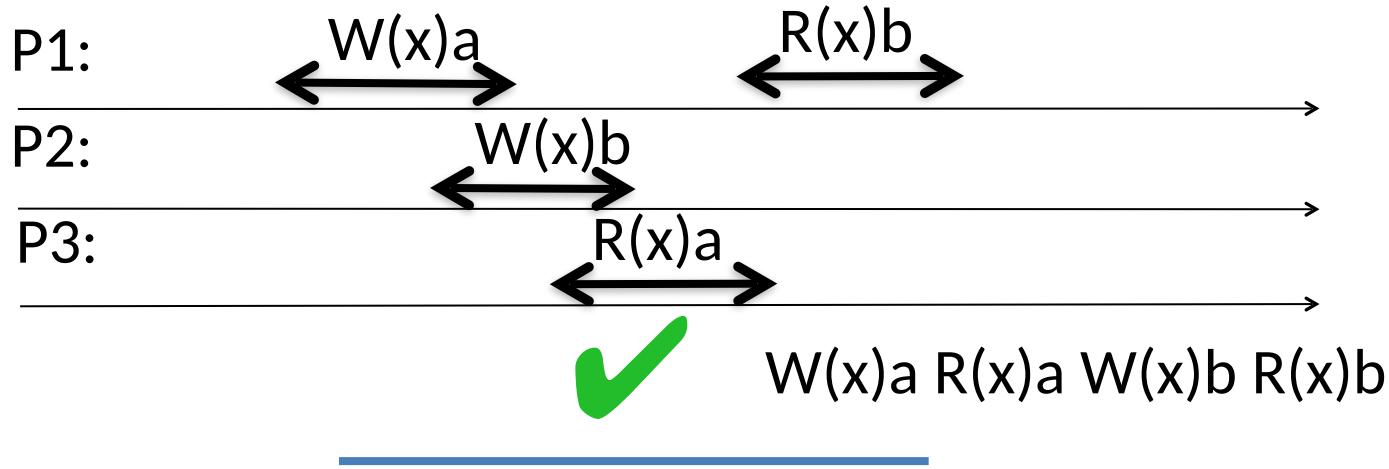
Example 5



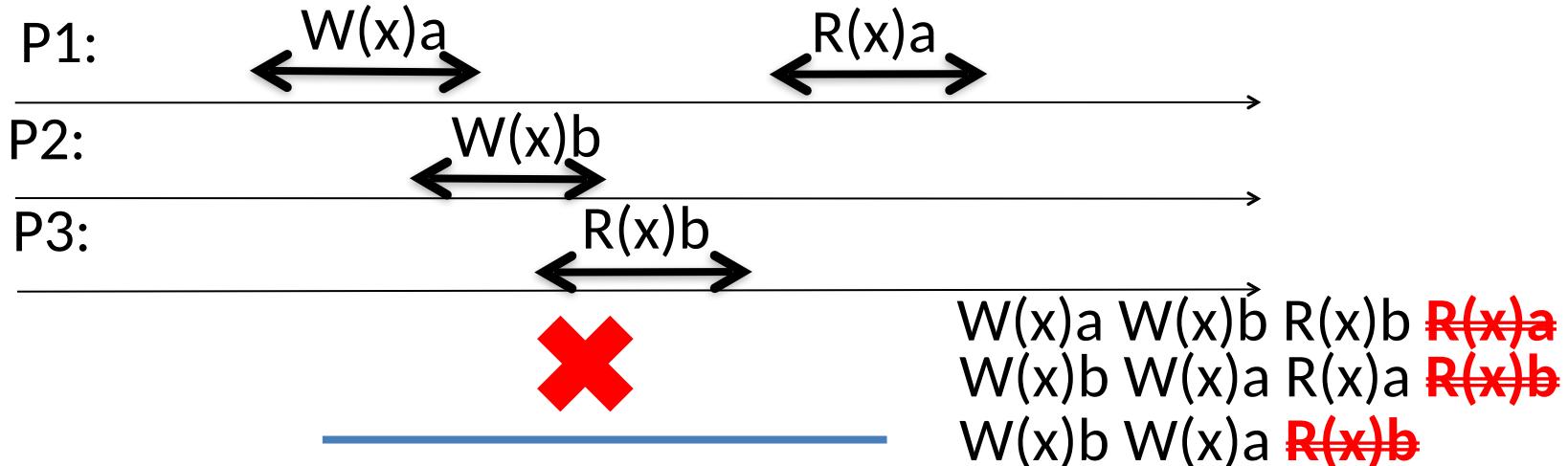
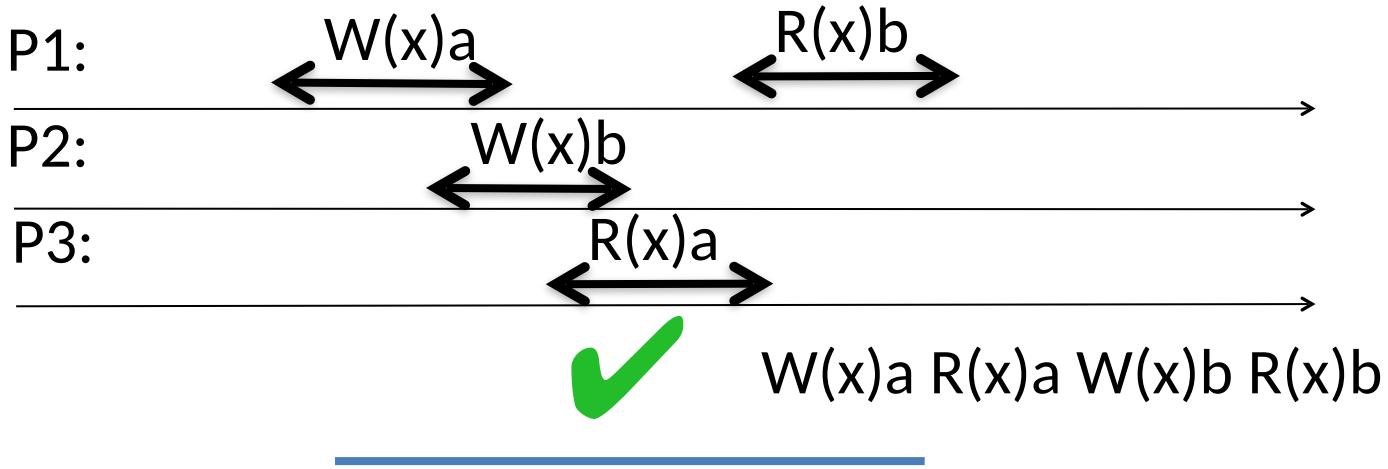
Example 5



Example 5



Example 5



Intuition for Causal Consistency

- **Weaker** than sequential consistency
- Distinguish events that are potentially **causally related** and those that are not (concurrent events)
- Similar to the **happens-before** relationship (cf. Lamport clock), but with reads and writes instead of messages
- If Event *B* is influenced (caused) by an earlier Event *A* ($A \rightarrow B$), causal consistency requires that **every process first sees A then B**
- **Concurrent events** (i.e., writes) may be seen in a different order on different machines

Causal Relationship

- **Read followed by write** in the same process, the two are causally related
 - Example: $R(x)a \rightarrow W(y)b$ (e.g., it may be that $y=f(x)$)
- Read is **causally related** to write that provided the value read got (across processes)
 - Example: $W(x)a \rightarrow R(x)a$
- Transitivity: if $Op1 \rightarrow Op2$, $Op2 \rightarrow Op3$, then $Op1 \rightarrow Op3$
- Independent writes by two processes on a variable are not causally related (they are concurrent)
 - Example: $W(x)a \parallel W(x)b$

“Potentially” Causally Related

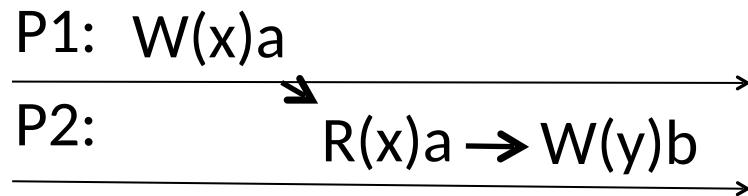
- Example:

$$\begin{array}{c} \text{P1: } W(x)a \\ \xrightarrow{\hspace{10em}} \\ \text{P2: } R(x)a \quad W(y)b \\ \xrightarrow{\hspace{10em}} \end{array}$$

- Reading of x and writing of y are **potentially** causally related
- Computation of y may have depended on value of x read by P2 (written by P1); e.g., $y = f(x)$
- On the other hand, y may not have depended on x , yet potential causality still holds in our formalization!

“Potentially” Causally Related

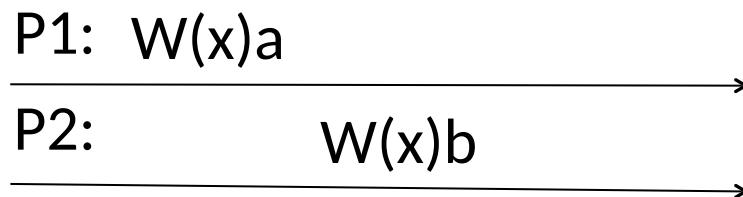
- Example:



- Reading of x and writing of y are **potentially** causally related
- Computation of y may have depended on value of x read by P2 (written by P1); e.g., $y = f(x)$
- On the other hand, y may not have depended on x , yet potential causality still holds in our formalization!

Definition of Causal Consistency

- “Potentially” causally related writes must be seen by all processes in the same order
- Concurrent writes may be seen in a different order by different processes



Causal Consistency Example I

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:		$R(x)a$		$R(x)b$ $R(x)c$

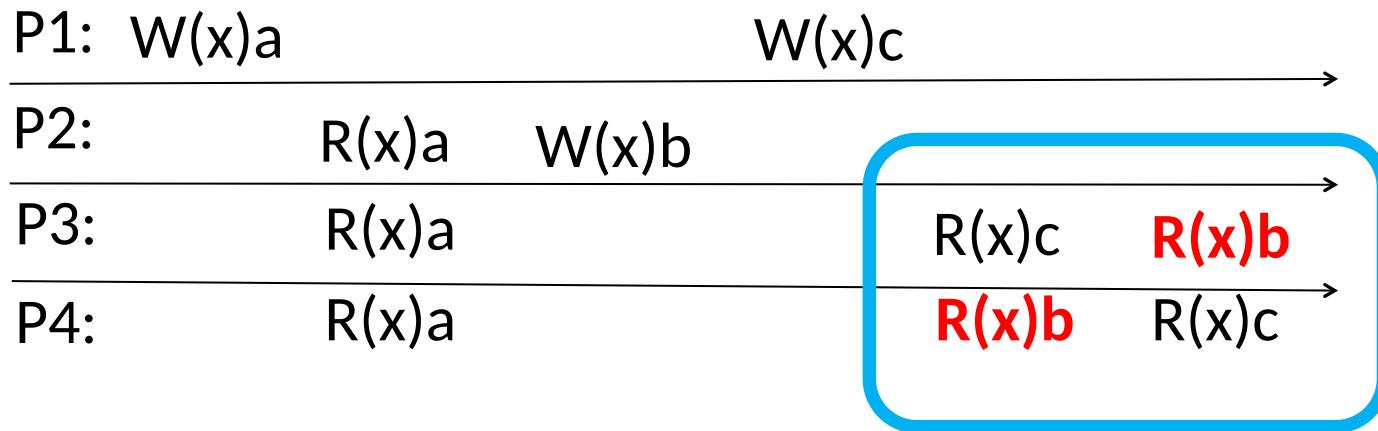
Is this strictly, sequentially or causally consistent?

Causal Consistency Example I

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:		$R(x)a$		$R(x)b$ $R(x)c$

Neither **strictly**, nor sequentially consistent, ...

Causal Consistency Example I

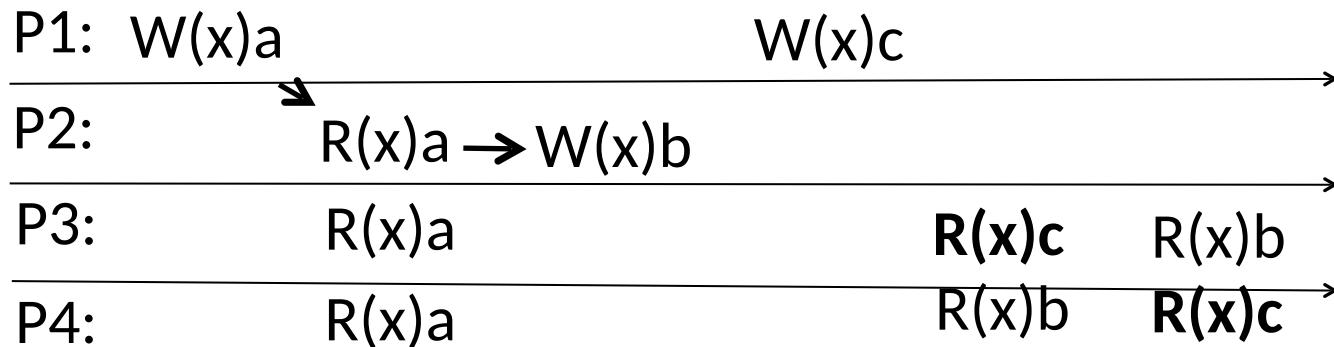


Neither **strictly**, nor **sequentially** consistent, ...

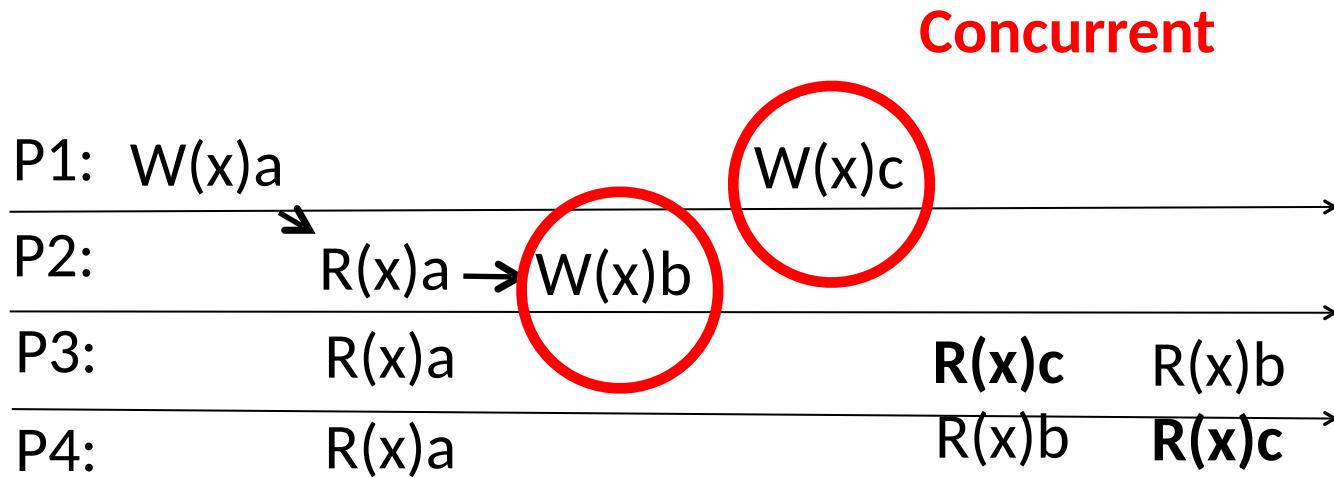
Causal Consistency Example I

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:		$R(x)a$	$R(x)b$	$R(x)c$

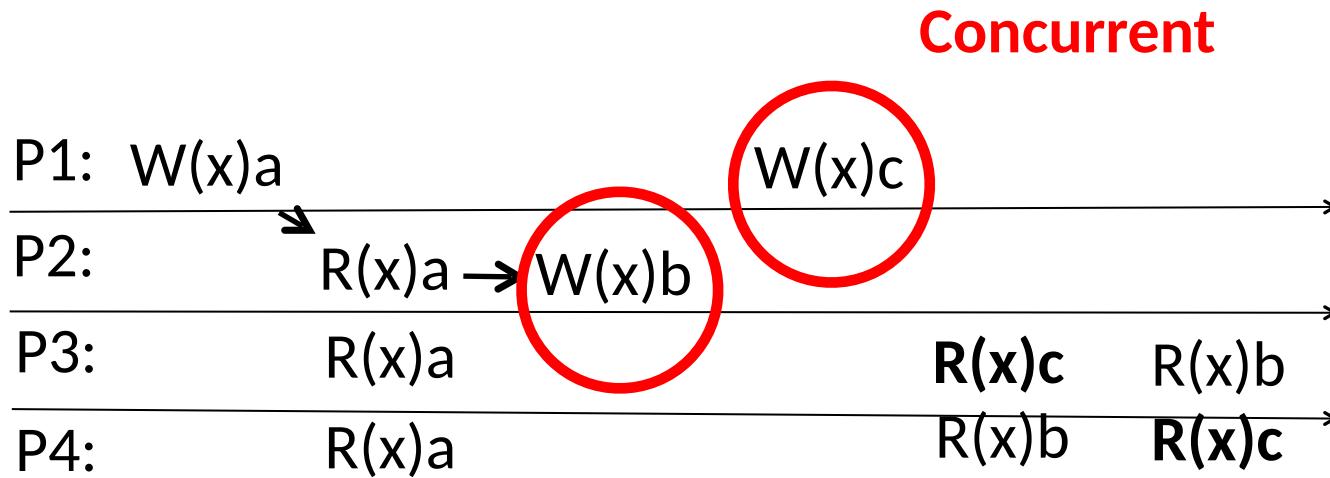
Causal Consistency Example I



Causal Consistency Example I



Causal Consistency Example I

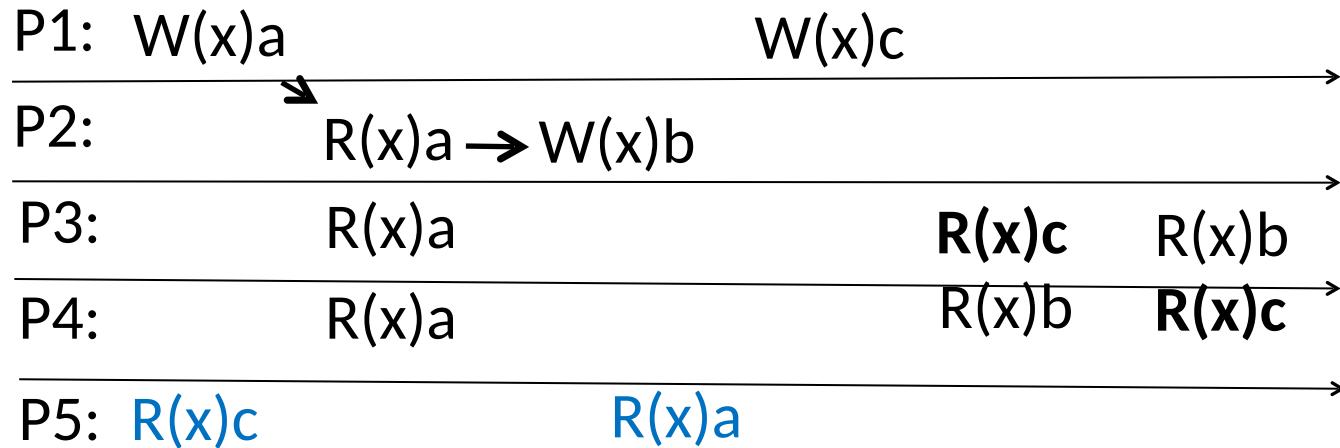


Neither strictly, nor sequentially consistent,
but causally consistent!

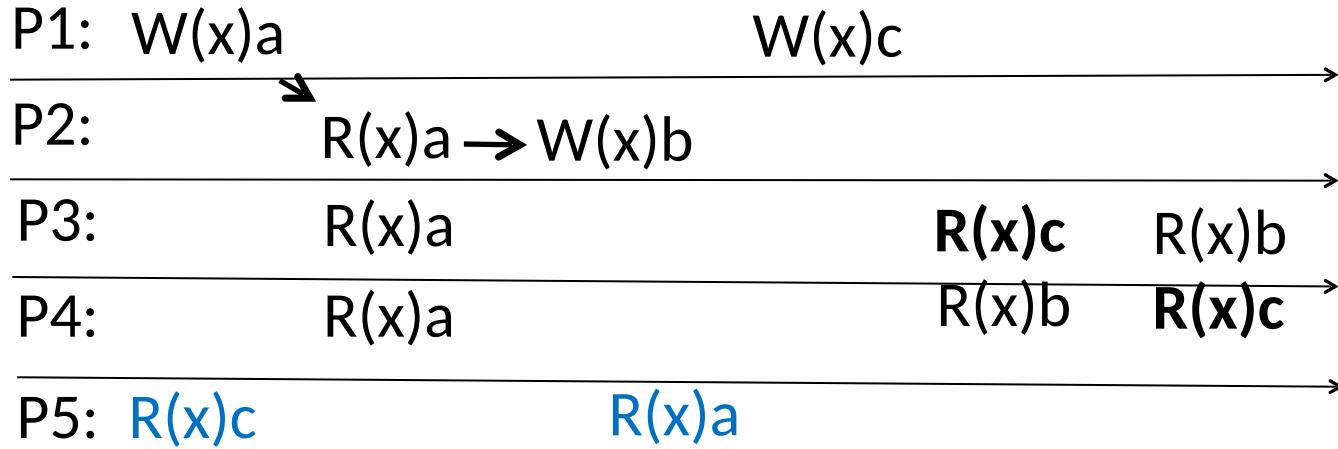
Causal Consistency Example I

P1:	$W(x)a$		$W(x)c$		\rightarrow
P2:		$R(x)a$	$W(x)b$		\rightarrow
P3:		$R(x)a$		$R(x)c$	$R(x)b$
P4:		$R(x)a$		$R(x)b$	$R(x)c$
P5:			$R(x)c$	$R(x)a$	\rightarrow

Causal Consistency Example I



Causal Consistency Example I



Not causally consistent, $W(x)a$ happens before $W(x)c$ in P1

Causal Consistency Example II

P1: $W(x)a$

P2: $R(x)a$ $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

P1: $W(x)a$

P2: $W(x)b$

P3: $R(x)b$ $R(x)a$

P4: $R(x)a$ $R(x)b$

Causal Consistency Example II

P1:

$W(x)a$



P2:

$R(x)a \rightarrow W(x)b$

P3:

$R(x)b \rightarrow R(x)a$



P4:

$R(x)a$

$R(x)b$



P1: $W(x)a$



P2:

$W(x)b$



P3:

$R(x)b$

$R(x)a$

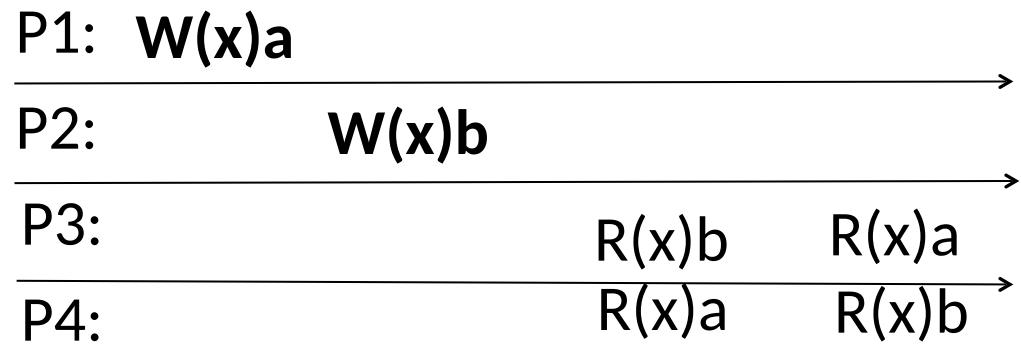
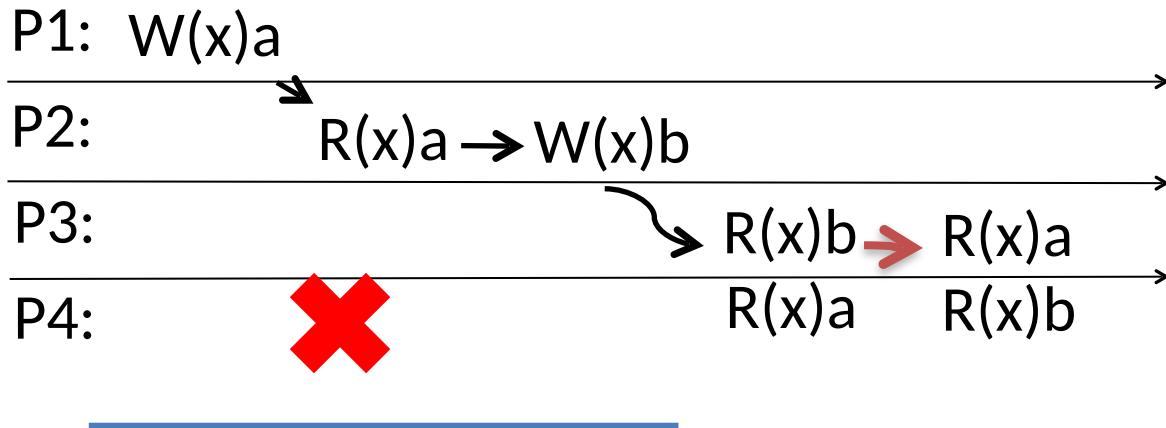


P4:

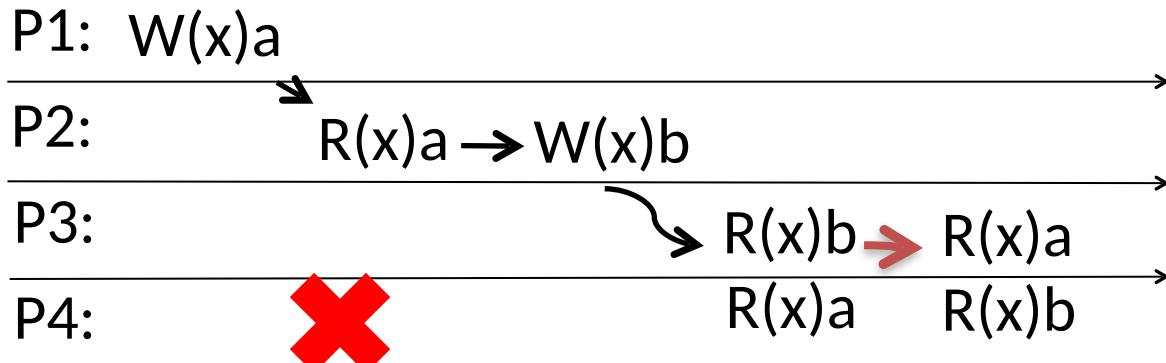
$R(x)a$

$R(x)b$

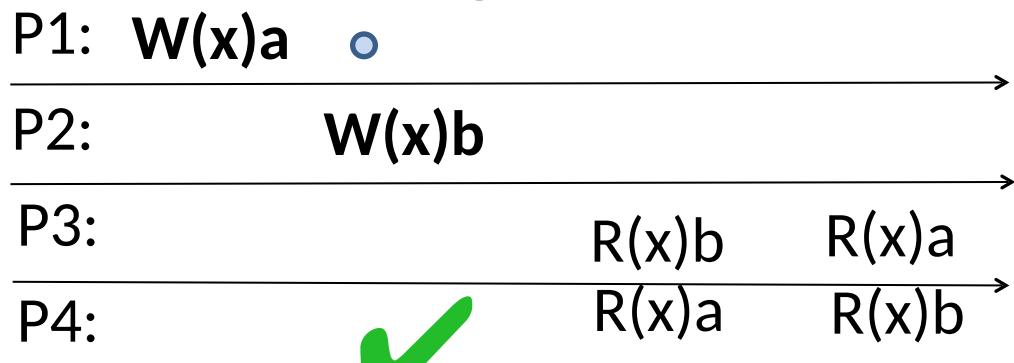
Causal Consistency Example II



Causal Consistency Example II



This wouldn't be allowed in sequential consistency



Definition of FIFO Consistency

- Writes by a single process are seen by all other processes in the order in which they were issued
- Writes from different processes may be seen in a different order by different processes
- Easy to maintain: simply send writes in FIFO order from each process to each replica (e.g. using TCP)

FIFO Consistency

P1: $W(x)a$

P2: $R(x)a \quad W(x)b \quad W(x)c$

P3: $R(x)b \quad R(x)a \quad R(x)c$

P1: $W(x)a$

P2: $R(x)a \quad W(x)b \quad W(x)c$

P3: $R(x)c \quad R(x)a \quad R(x)b$

FIFO Consistency

P1: $W(x)a$

P2: $R(x)a \quad W(x)b \quad W(x)c$

P3: $R(x)b \quad R(x)a \quad R(x)c$



P1: $W(x)a$

P2: $R(x)a \quad W(x)b \quad W(x)c$

P3: $R(x)c \quad R(x)a \quad R(x)b$

FIFO Consistency

P1: $W(x)a$

P2: $R(x)a \quad W(x)b \quad W(x)c$

P3: $R(x)b \quad R(x)a \quad R(x)c$



P1: $W(x)a$

P2: $R(x)a \quad W(x)b \quad W(x)c$

P3: $R(x)c \quad R(x)a \quad R(x)b$



Weak Consistency

- Not all processes need to see all writes, let alone in the same order
- Based on **synchronization variable S** with single operation **Synchronize(S)**
- Any process can perform read/write operations and synchronize
- Order of operations before **synchronization** is not consistent
- After synchronization, all processes see the **same outcome of operations** preceding the synchronization point

Weak Consistency Interpretation

- Process forces the just written value out to all synchronizing replicas
- Process can be sure to get the most recent value written before it reads
- Model enforces **consistency on a group of operations** as opposed to individual reads and writes
- Care about the **effect of a group** of reads and writes

Weak Consistency

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

P1:	W(x)a	W(x)b	S	
P2:			S	R(x)a

Weak Consistency

P2 & P3 have yet to synchronize, no guarantees about values read

P1: $W(x)a$ $W(x)b$ S

P2: R(x)a R(x)b S

P3: R(x)b R(x)a S



P1: $W(x)a$ $W(x)b$ S

P2: S R(x)a

Weak Consistency

P2 & P3 have yet to synchronize, no guarantees about values read

P1: $W(x)a$ $W(x)b$ S

P2: R(x)a R(x)b S

P3: R(x)b R(x)a S



P1 propagates value b

P1: $W(x)a$ $W(x)b$ S

...

P2:

S R(x)a



Weak Consistency

Agreement on value is implementation dependent.

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

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

Weak Consistency

Agreement on value is implementation dependent.

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

✓

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

Weak Consistency

Agreement on value is implementation dependent.

P1:	W(x)a	W(x)b	S	R(x)a	
-----	-------	-------	---	-------	--

P2:		W(x)a	S	R(x)a	
-----	--	-------	---	-------	--

P3:			R(x)b	R(x)a	S
-----	--	--	-------	-------	---

P1 & P2 agree
on value a



P1:	W(x)a	W(x)b	S	R(x)b	
-----	-------	-------	---	-------	--

P2:		W(x)a	S	R(x)b	
-----	--	-------	---	-------	--

P3:			R(x)b	R(x)a	S
-----	--	--	-------	-------	---

P1 & P2 agree
on value b



Weak Consistency

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

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

Weak Consistency

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



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

Weak Consistency

P1: W(x)a W(x)b S R(x)a

P2: W(x)a S R(x)b

P3: R(x)b R(x)a S



P1: W(x)a W(x)b S R(x)b

P2: W(x)a S R(x)a

P3: R(x)b R(x)a S



Summary of Consistency Models

Strongest

Consistency	Description
Strict	Absolute time ordering of all shared accesses
Linearizability	All processes see all shared accesses in the same order . Accesses ordered according to a (non-unique) global timestamp
Sequential	All processes see all shared accesses in the same order . Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order
FIFO	All processes see writes from each other in the order they were issued . Writes from different processes may not always be seen in that order
Weak	Shared data can be counted on to be consistent only after a synchronization is done

Weakest

When to use...?

- **Linearizability:** Strongest distributed solution, possible with eager replication (synchronous), Hbase, Bigtable
- **Sequential:** System-wide consistent reads, e.g., everyone sees replies to a post in same order
- **Causal:** Everyone sees posts before replies
- **FIFO:** Reading all messages from each friend in order but not across friends
- **Weak:** Responsibility left to the developer who must explicitly enforce synchronization

CLIENT-CENTRIC CONSISTENCY

Client-Centric Consistency

- So far, the goal was to maintain consistency in presence of **concurrent read and write** operations
- There are use cases with **no (few) concurrent writes**, or consistency of write operations are secondary
 - **DNS**: **No write-write conflicts** since there is a single authority updating each domain (disjoint partitions)
 - **Key-value stores**: **Usually no write-write conflicts** since updates partitioned by keys, e.g., Dynamo, Cassandra (also, optimistic concurrency control)
 - **WWW**: Heavy use of client-side caching, reading stale pages is acceptable in many cases

Client-Centric Consistency

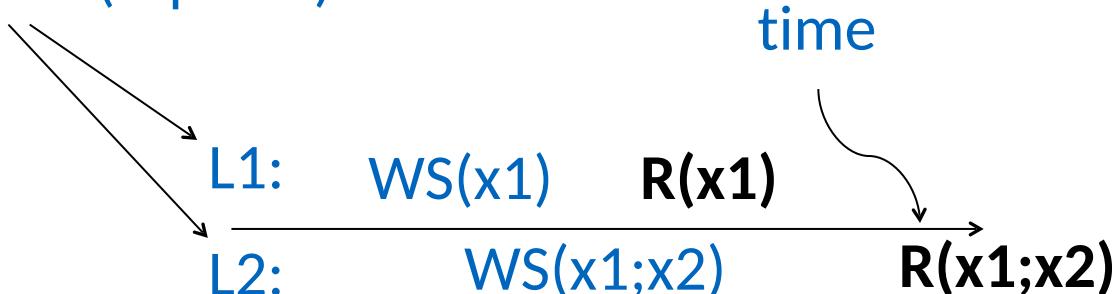
- Client-centric consistency puts the emphasis on maintaining a **consistent view** for a **single process**, instead of on the data stored in the system
- Emphasis on **read-write conflicts**, and we assume **write-write** conflicts do not exist (i.e., no concurrent writes)
- Client-centric consistency models describe what happens when a **single client** writes/reads from **multiple replicas**

Eventual Consistency

- Eventual consistency states that all replicas **eventually** converge when write operations stop
 - E.g., lazy replication using gossiping (cf. replication)
- Very weak form of consistency with no time bound, but **highly available** (i.e., always return a value, but could be stale)
- Works fine if a client always reads from the **same** replica...
- ...but gives “weird” results if client reads from **multiple replicas**:
 - In a mobile scenario
 - During replica failure
- Client-centric consistency models describe what happens when a **single** client writes/reads from **multiple replicas**

Notation

Different locations (replicas)



- **Reads and writes by one client at different locations**
- WS represents a series of write operations at L_i
- E.g., $WS(x1)$ denotes that only op $x1$ has been written; $WS(x1;x2)$ denotes that op $x1$ and then later op $x2$ have been written
- $R(x1;x2)$ means the read returns a value formed by operation $x1$ followed by $x2$
- $W(x1), R(x1)$ are write to and read from $x1$

Read-Your-Writes Consistency

- Informally, ... $W \dots R \dots$ (by same process!)
- If a read R follows a write then W is included in the set of writes read by R
- **A write operation is always completed before a successive read operation by the same process, no matter where the read operation takes place**

Read Your Writes

Example: Updating your password on a cluster

L1: **W(x1)**

L2:

WS(x1)

R(x1)

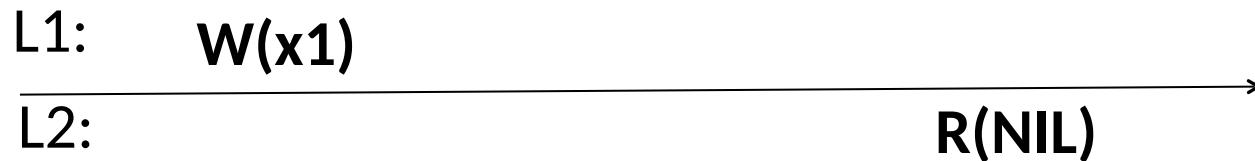
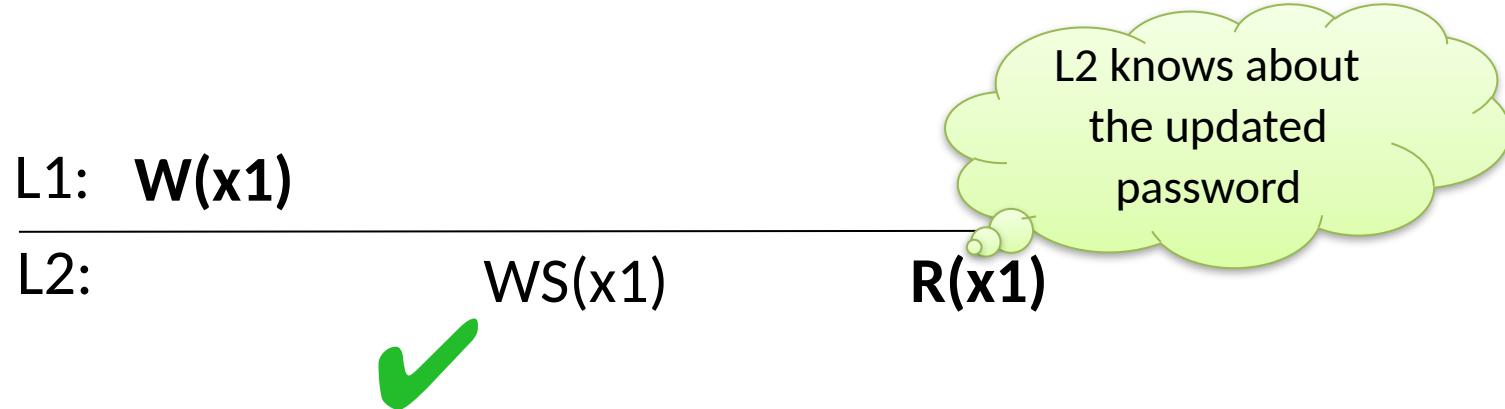
L1: **W(x1)**

L2:

R(NIL)

Read Your Writes

Example: Updating your password on a cluster



Read Your Writes

Example: Updating your password on a cluster

L1: $W(x_1)$

L2:

$WS(x_1)$



$R(x_1)$

L2 knows about
the updated
password

L1: $W(x_1)$

L2:

$R(NIL)$



L2 doesn't know
about the
password;
cannot login!

Monotonic-Reads Consistency

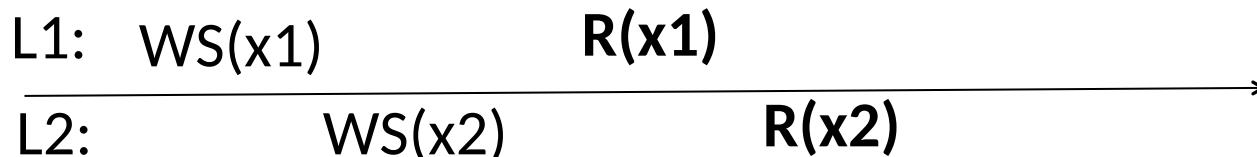
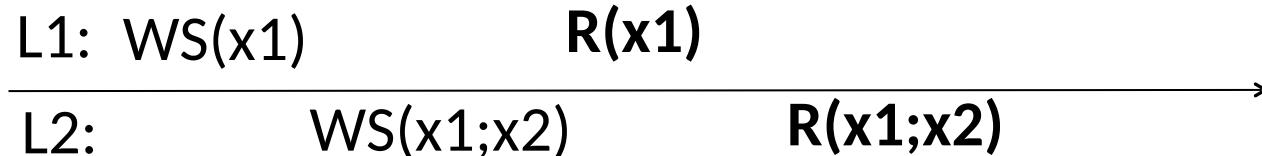
- ... $WS1(..) R \dots WS2(..) R \dots$ (s.t. $WS1(..) \subseteq WS2(..)$)
- If a process reads a value formed by a set of operations, WS , any successive read operation of that process will always return a **value** formed from a **superset** of WS
- A process always sees **more recent data** (but not necessarily fresh!)
- If a process reads again from another replica that replica must have already received the relevant older operations

Example: Monotonic Reads

Example: E-mail client

Each read returns the list of emails

Each write adds one e-mail



Example: Monotonic Reads

Example: E-mail client

Each read returns the list of emails

Each write adds one e-mail

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_1; x_2)$



$R(x_1; x_2)$

L2 knows
about all e-
mails

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_2)$ $R(x_2)$

Example: Monotonic Reads

Example: E-mail client

Each read returns the list of emails

Each write adds one e-mail

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_1; x_2)$



$R(x_1; x_2)$

L2 knows
about all e-
mails

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_2)$



$R(x_2)$

L2 doesn't
know about
the e-mail
 x_1 !

Writes-Follow-Reads Consistency

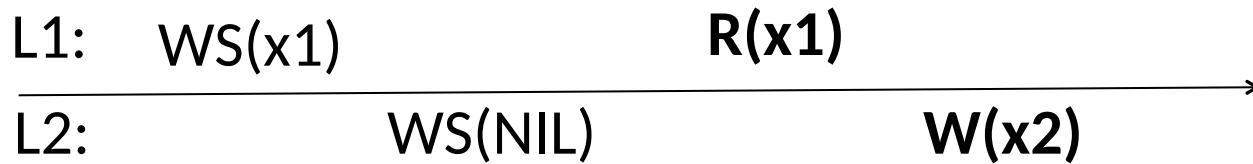
- ... $WS(X_1, \dots, X_n) R_1 \dots WS(X_1, \dots, X_n) W_2 \dots$
- If a read R_1 precedes a write W_2 then at any replica, if W_2 is written, it is preceded by WS , the write set read by R_1
- Any successive write operation by a process will be performed on a state that is **up to date** with the value **most recently read** by that process

Writes Follow Reads

Example: Facebook Wall

Reading a post

Replying to a post

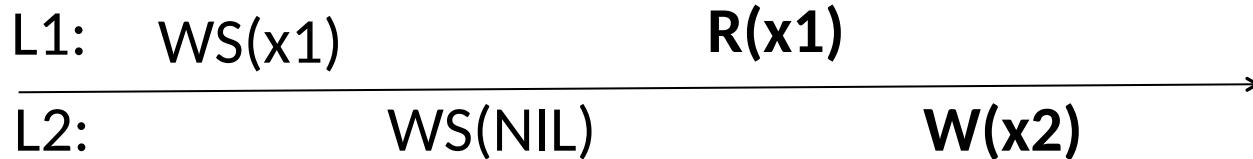
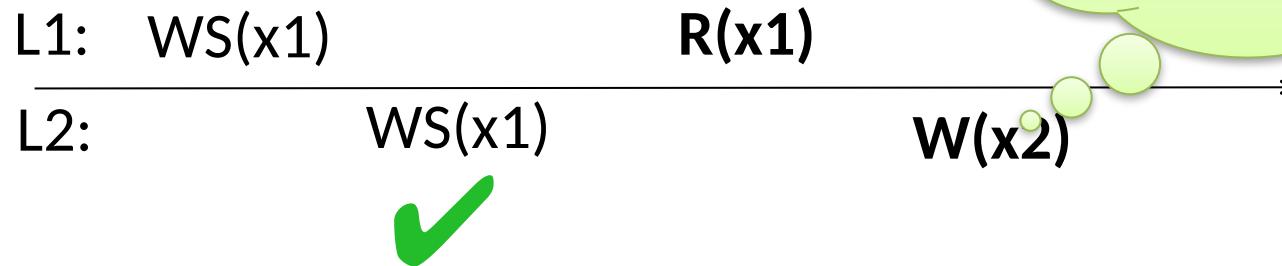


Writes Follow Reads

Example: Facebook Wall

Reading a post

Replying to a post



Writes Follow Reads

Example: Facebook Wall

Reading a post

Replying to a post

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(x_1)$ $W(x_2)$



L2 knows post (x_1),
writes reply (x_2) to
known post

L1: $WS(x_1)$ $R(x_1)$

L2: $WS(NIL)$ $W(x_2)$



L2 doesn't know
about the post
to reply to!

Monotonic-Writes Consistency

- Informally, ... $W_1 \dots W_2 \dots$
- If a write W_1 precedes a write W_2 then on any replica, W_1 must precede W_2
- In other words, the **writes by the same process** are performed in the same order at every replica
- Resembles FIFO consistency model

Monotonic Writes

Example: Replicating software code

$W(x_1)$ adds a new method to a program

$W(x_2)$ calls the new method

L1: $W(x_1)$

L2: $WS(x_1)$ $W(x_2)$

L1: $W(x_1)$

L2: $W(x_2)$

Monotonic Writes

Example: Replicating software code

$W(x_1)$ adds a new method to a program

$W(x_2)$ calls the new method

L1: $W(x_1)$

L2:

WS(x_1)

$W(x_2)$



L2 knows
about the
new method

L1: $W(x_1)$

L2:

$W(x_2)$

Monotonic Writes

Example: Replicating software code

$W(x_1)$ adds a new method to a program

$W(x_2)$ calls the new method

L1: $W(x_1)$

L2: WS(x_1)



$W(x_2)$

L2 knows
about the
new method

L1: $W(x_1)$

L2:



$W(x_2)$

L2 doesn't
know the new
method! The
code does not
compile

Summary: Client-centric consistency

- Eventual consistency can be used when:
 - **Few** concurrent write occurs
 - Used in Dynamo, Cassandra, Riak
- Eventual consistency provides **high availability** and scalability
- Client-centric consistency dictates how reads and writes should look from the client's perspective
 - Read your writes
 - Monotonic reads
 - Monotonic writes
 - Writes follow reads

MapReduce



Agenda

- MapReduce
- Hadoop Ecosystem
- Related: Spark



[fathomdelivers.com]



MAPREDUCE

Origins

- **Google** faced the problem of having to analyzing huge sets of data (order of petabytes)
- Standard tasks: PageRank, web access logs, distributed grep, etc.
- Algorithms to process data often reasonably simple
- But to finish computation in an acceptable amount of time, task must be split and forwarded to thousands of cheap **worker** machines (commodity hardware)

I ❤
(lisp)



Distributed Systems (H.-A. Jacobsen)



Origins cont'd

- **Common tasks** for processing large amounts of data
 - Split data
 - Forward data and code to participant nodes
 - Check node state to react to errors
 - Retrieve and reorganize results
- Need: **Simple large-scale data processing abstraction**
- Inspiration from **functional programming** and **scatter/gather** in distributed/grid computing
- Difference with previous works is enforcing data to be in **key-value** format, which simplifies the design
- Google published MapReduce paper in 2004 at OSDI

Functional Programming (Quick Digression)

- MapReduce is “*functional programming meets distributed processing on steroids*”
 - Not a new idea, FP dates back to the 50’s
- What is functional programming?
 - Computation as application of **functions**
 - Theoretical foundation provided by lambda calculus
 - Kind of **declarative programming**
- How is it different (from imperative programming)?
 - Traditional notions of “data” and “instructions” are not applicable
 - Data flows are implicit in program
 - Different orders of execution are possible

Lisp Basics

(Lisp is List Processing)

- Lists are primitive datatypes

(list 1 2 3 4 5)

(list (list 'a 1) (list 'b 2) (list 'c 3))

(nth (list 1 2 3 4 5) 0) -> 1

(nth (list (list 'a 1) (list 'b 2) (list 'c 3)) 3) -> nil

- Function evaluation is written in prefix

(+ 1 2) -> 3

(* 3 4) -> 12

(Math/sqrt (+ (* 3 3) (* 4 4))) -> 5

(def x 3) -> x

(* x 5) -> ??

15

Clojure is a Lisp Dialect
which runs on JVM or
using ClojureScript on a
Javascript runtime

Try out now!

<https://clojurescript.io> or
install clojure on your
computer

Lisp Functions

- Functions are defined by binding **lambda expressions** to variables

```
(def foo
```

```
  (fn [x y] (Math/sqrt (+ (* x x) (* y y)))))
```

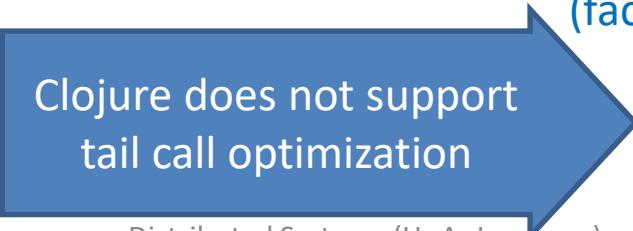
- Once defined, function can be applied:

```
(foo 3 4) -> 5
```

- Generally expressed with **recursive** calls (instead of loops):

```
(def factorial (fn [n]
  (if (= n 1)
    1
    (* n (factorial (- n 1))))))
(factorial 6) -> 720
```

```
(def factorial (fn [n]
  (reduce * (range 1 (inc n)))))
(factorial 6) -> 720
```

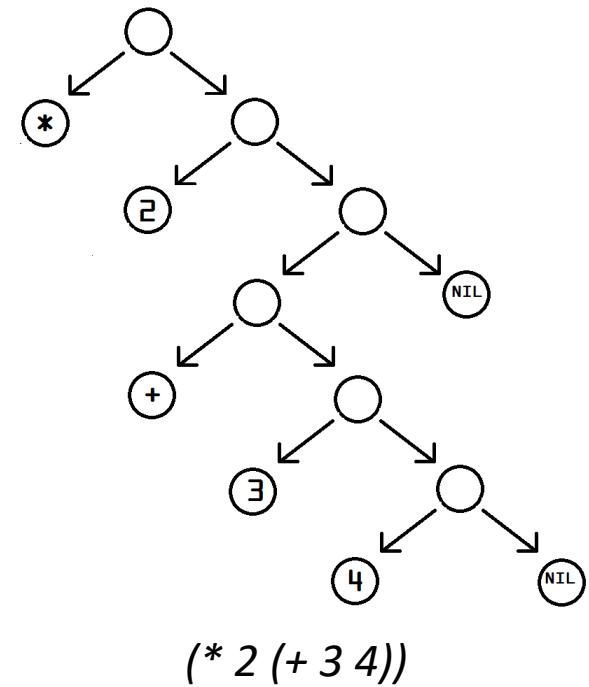


Clojure does not support
tail call optimization

Lisp Features

- Examples are from Clojure, a Lisp dialect
- Everything is an s-expression
 - No distinction between “data” and “code”, called [“Homoiconicity”](#)
 - Operators, lists, values...
 - Easy to write self-modifying code
- Higher-order functions
 - Functions that take other functions as arguments

```
(def adder (fn [x] (fn [a] (+ x a)))  
(def add-five (adder 5))  
(add-five 11) -> 16
```

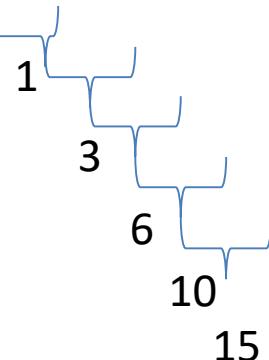


Clojure Reduce

“+” is a function taking 2 arguments

(reduce + 0 (list 1 2 3 4 5))

Identity Element (optional, depends on the function if implemented or not)



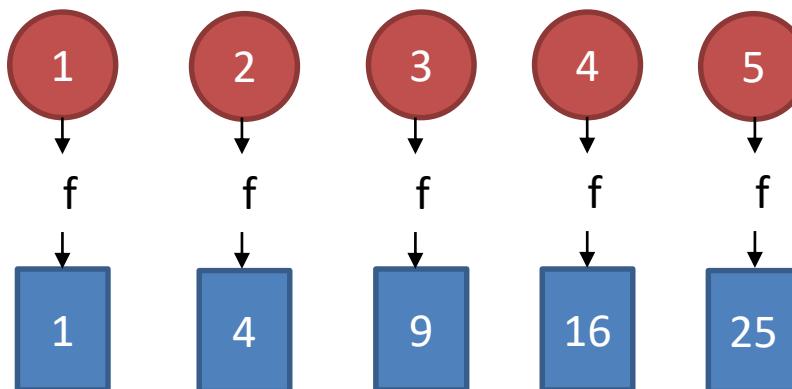
From Lisp to MapReduce I

- Why use functional programming for large-scale computing?
 - **Hide** distribution and coordination from analytics code
 - Define functions to capture **core application logic**
 - Let **framework** (MapReduce) execute functions across many machines
 - Thus, avoids tricky bugs due to distribution, parallelism, coordination
- Two important concepts in functional programming about lists
 - **Map**: Do something to everything in a list
 - **Fold**: Combine results of a list in some way
 - MapReduce distributes the content of lists to workers

Map

```
(map (fn [x] (* x x)) (list 1 2 3 4 5))  
-> '(1 4 9 16 25)
```

- Map is a **higher-order function** (takes one or more functions as arguments)
- How map works:
 - Function is applied to every element in a list
 - Result is a new list

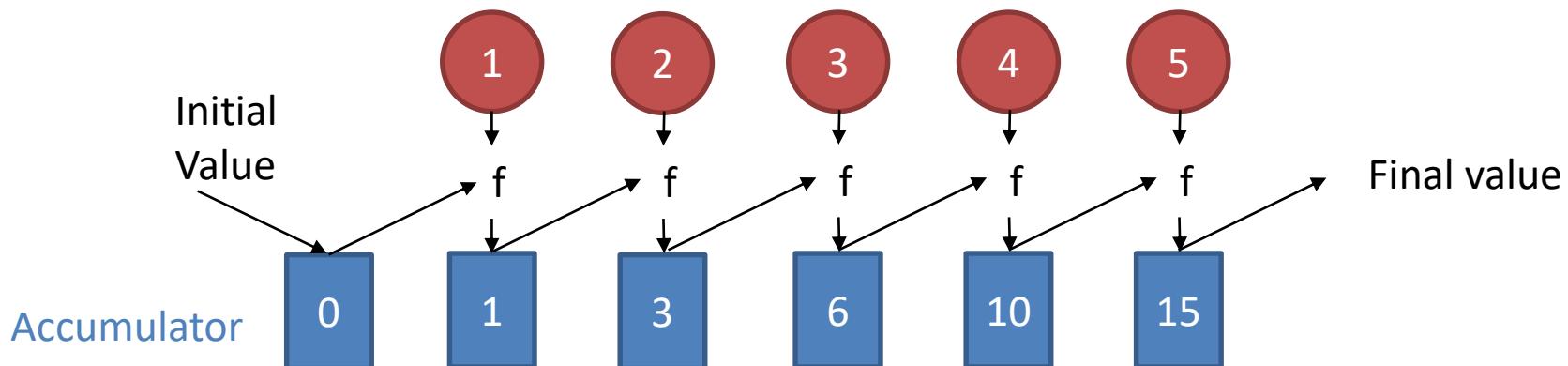


Fold

```
//In clojure also called reduce  
(reduce + 0 (list 1 2 3 4 5))  
(reduce * 1 (list 1 2 3 4 5))
```

- Fold is also a **higher-order function**
- How fold works:
 - **Accumulator** (generalization of a counter) set to initial value
 - Function applied to list element and accumulator
 - Result stored in accumulator
 - Repeated for every item in list
 - Result is the final value in accumulator

What would happen if this is set to 0?



Map/Fold in Action

- Map example:

```
(map (fn [x] (* x x)) (list 1 2 3 4 5))  
-> '(1 4 9 16 25)
```

- Fold (In Clojure called Reduce with accumulator argument):

```
(reduce + 0 (list 1 2 3 4 5)) -> 15  
(reduce * 1 (list 1 2 3 4 5)) -> 120
```

- Sum of squares:

```
(def sum-of-squares (fn [v] (reduce + 0 (map (fn [x] (* x x)) v)))  
(sum-of-squares (list 1 2 3 4 5)) -> 55
```

From Lisp to MapReduce II

- Let's assume a long list of records: imagine if...
 - We could **distribute** the execution of map operations to multiple nodes
 - We had a mechanism for bringing map results **back together** in the fold operation
- That is MapReduce! (and Hadoop)
- **Implicit parallelism** (due to functional paradigm):
 - **Parallelize execution** of map operations; they are isolated
 - **Reorder folding** if the fold function is commutative and associative
 - **Commutative**: Change order of operands: $x*y = y*x$
 - **Associative**: Change order of operations: $(2+3)+4 = 2+(3+4)=9$

More Than A Buzzword

1. Message-passing interface (MPI)
 - Library with basic communication elements
 - Highly portable, popular for scientific computing
2. Remote procedure calls (RPC)
 - A method to call a function/process on another machine
 - Popular with commercial clusters & client/server applications
3. MapReduce
 - A simple programming model that abstracts most complexity of programming clusters
 - Provides fault-tolerance
 - Tradeoff: Generality?



[google.com]

MapReduce

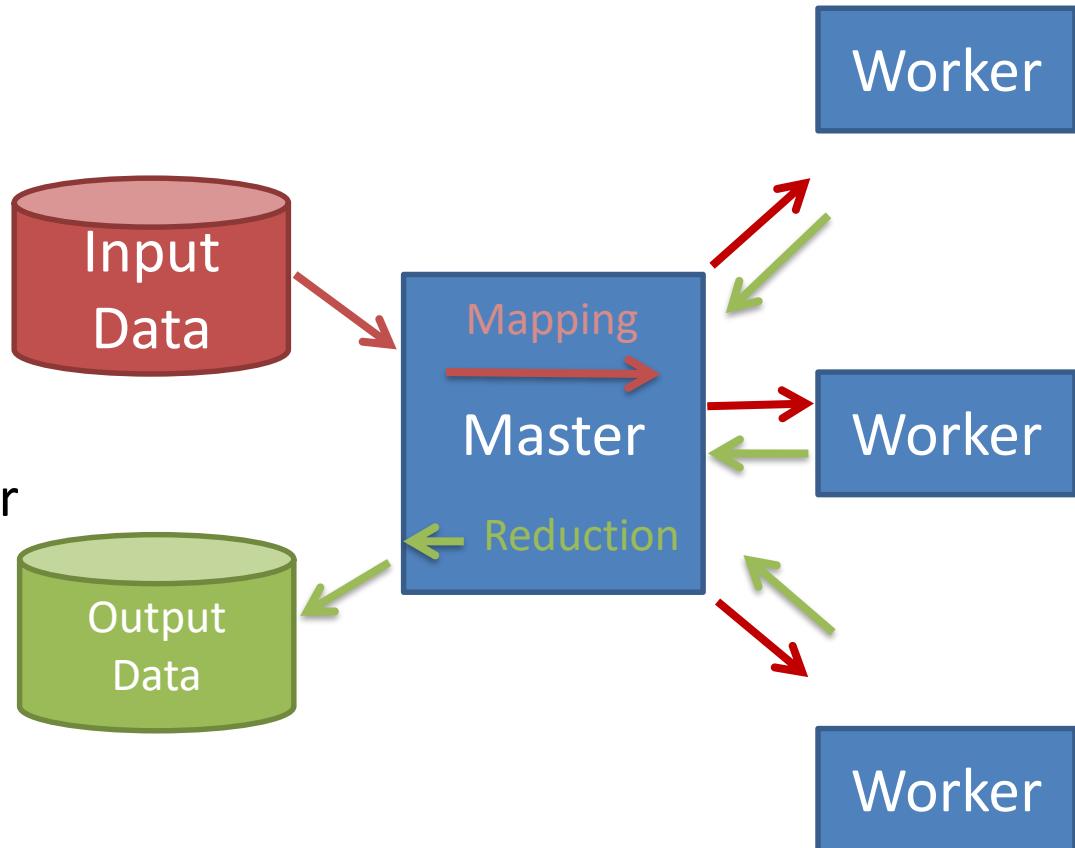
- Programming model & runtime system for processing large data-sets
 - E.g., search algorithms (2005 – 200TB indexed)
 - Goal: Simplify use of 1000s of CPUs and TBs of data with commodity hw.
- Inspiration: Functional programming languages
 - Programmer specifies only “what”
 - System determines “how”
 - System deals with scheduling, parallelism, locality, communication..
- Ingredients
 - Automatic parallelization and distribution
 - Fault-tolerance
 - I/O scheduling
 - Status reporting and monitoring

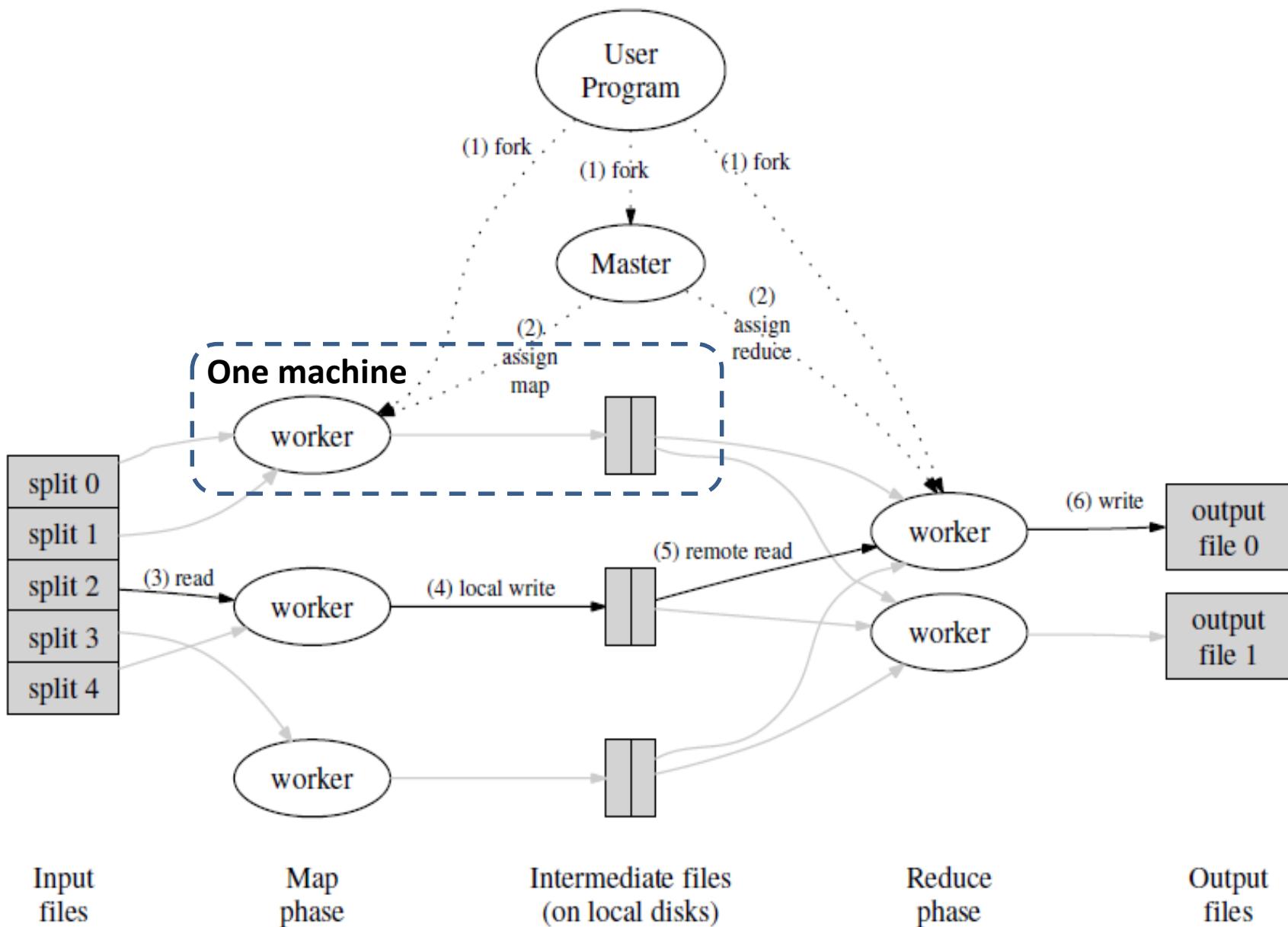
Examples Map Tasks

- Examples shown are tiny, but Map Tasks can be computationally heavy
- Feature extraction for machine learning
 - Scaling each raw image to a smaller size
 - Run Canny edge detector on each image in training data set
- Recoding
 - Recode each video from source format to webm file in 4 different resolutions
- Natural language processing
 - Translate each webpage and index it
 - Sentiment analysis of each webpage, tweet, ...

Simplified View of Architecture

- Input data is distributed to workers (i.e., available nodes)
- Workers perform computation
- Master coordinates worker selection & fail-over
- Results stored in output data files





MapReduce Programming Model (Programmer Specified)

- Input & output: Each a set of **key/value pairs**
- `map (in_key, in_value) -> list(out_key, intermediate_value)`
 - Processes input key/value pair
 - Produces set of intermediate pairs
- `reduce (out_key, list(intermediate_value)) -> list(out_value)`
 - Combines all intermediate values for a particular key
 - Produces a set of merged output values (usually just one)
- *Two optional optimizations for tuning MapReduce*
 - `combine (key, values) -> output(key, sum(values))`
 - `partition (out_key, number of partitions) -> partition id for out_key`

Programming Model

Map and Reduce Signatures

- Map: $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
- Reduce: $(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$
- MapReduce framework “re-shuffles” the output from Map to conform to the input of reduce

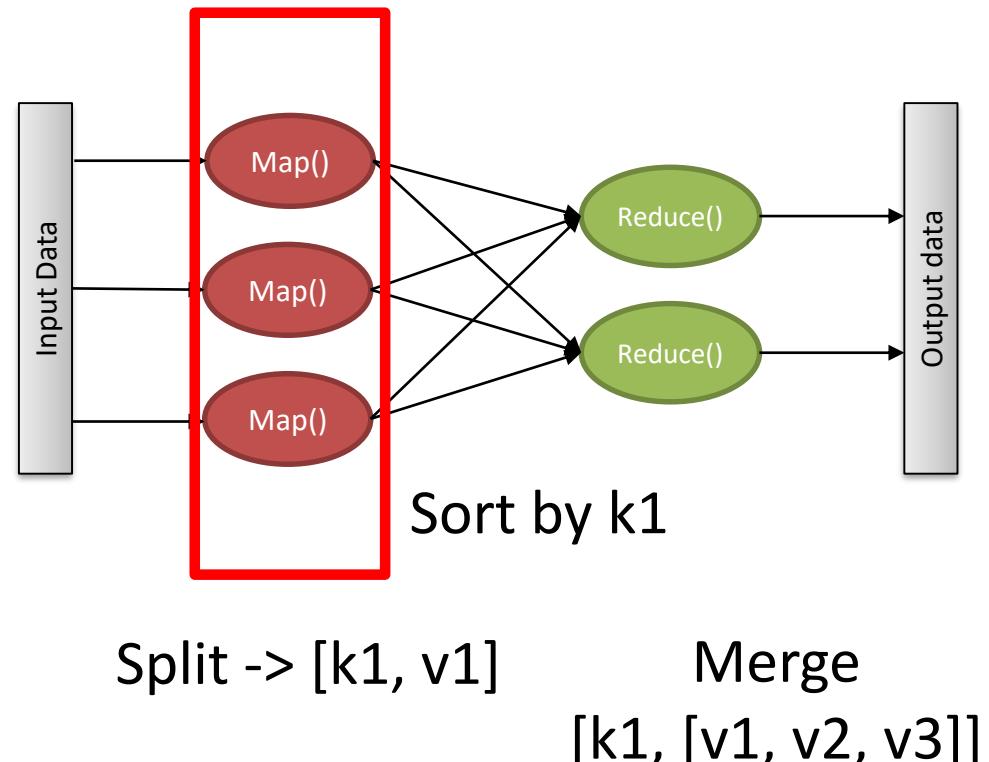
Programming Model

High-level Example

- Map: $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
 - (filename, file content) $\rightarrow \text{list}(\text{word}, 1)$
- Reduce: $(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$
 - (word, list(1, 1, ...)) $\rightarrow \text{count}$

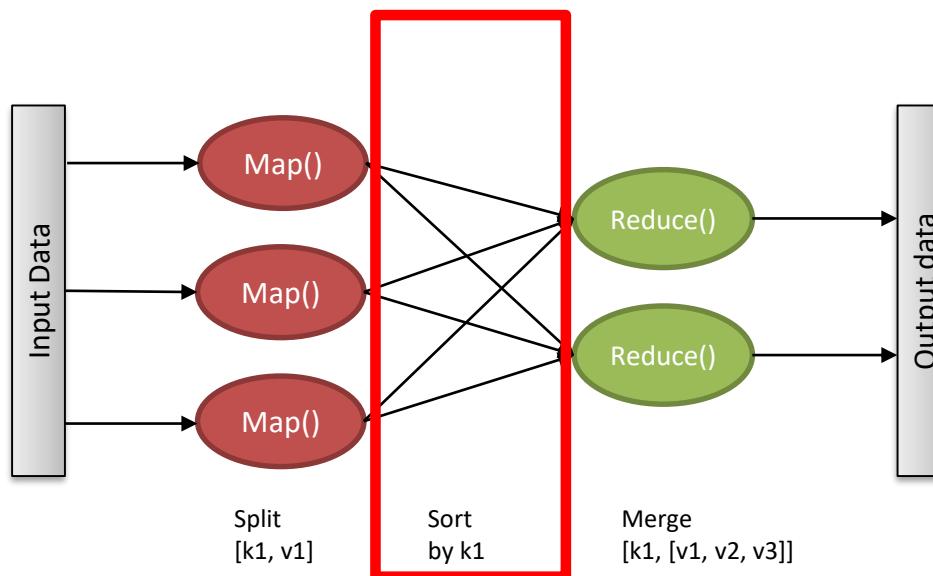
Map()

- Records from data source (lines out of files, rows of a database table, etc.) are fed into Map as key-value pairs: (filename, line)
- Map() produces one or more intermediate values along with an output key from the input



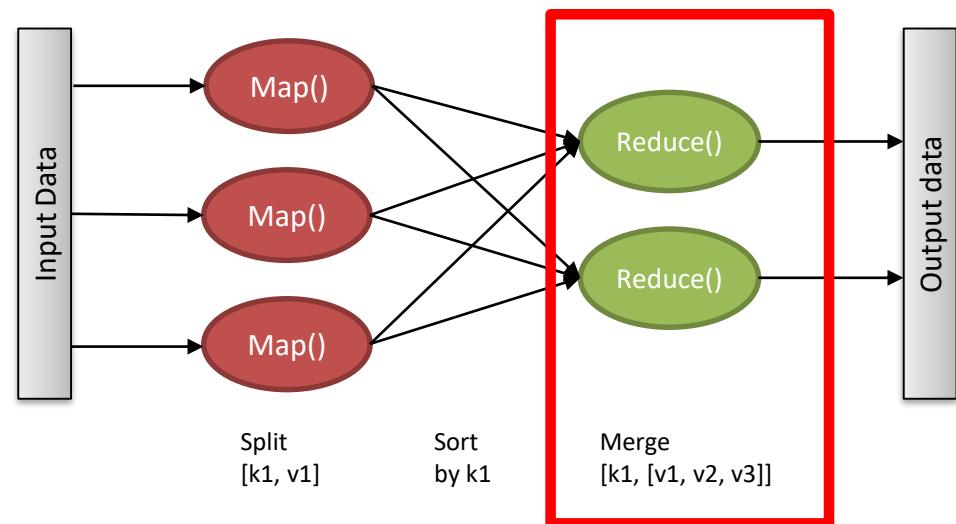
Sort and Shuffle

- MapReduce framework
 - Shuffles and sorts intermediate pairs based on key
 - Assigns resulting streams to reducers



Reduce()

- After map phase is over, all intermediate values for a given output key are combined into a list
- Reduce() combines those intermediate values into one or more final values for that same output key
- Often, only one final value per key



MapReduce Execution Stages

- **Scheduling:** Assigns workers to map and reduce tasks
- **Data distribution:** Moves processes to data (Map)
- **Synchronization:** Gathers, sorts, and shuffles intermediate data (Reduce)
- **Errors and faults:** Detects worker failures and restarts

MapReduce Example: Word Count

(Count word frequencies across set of documents)

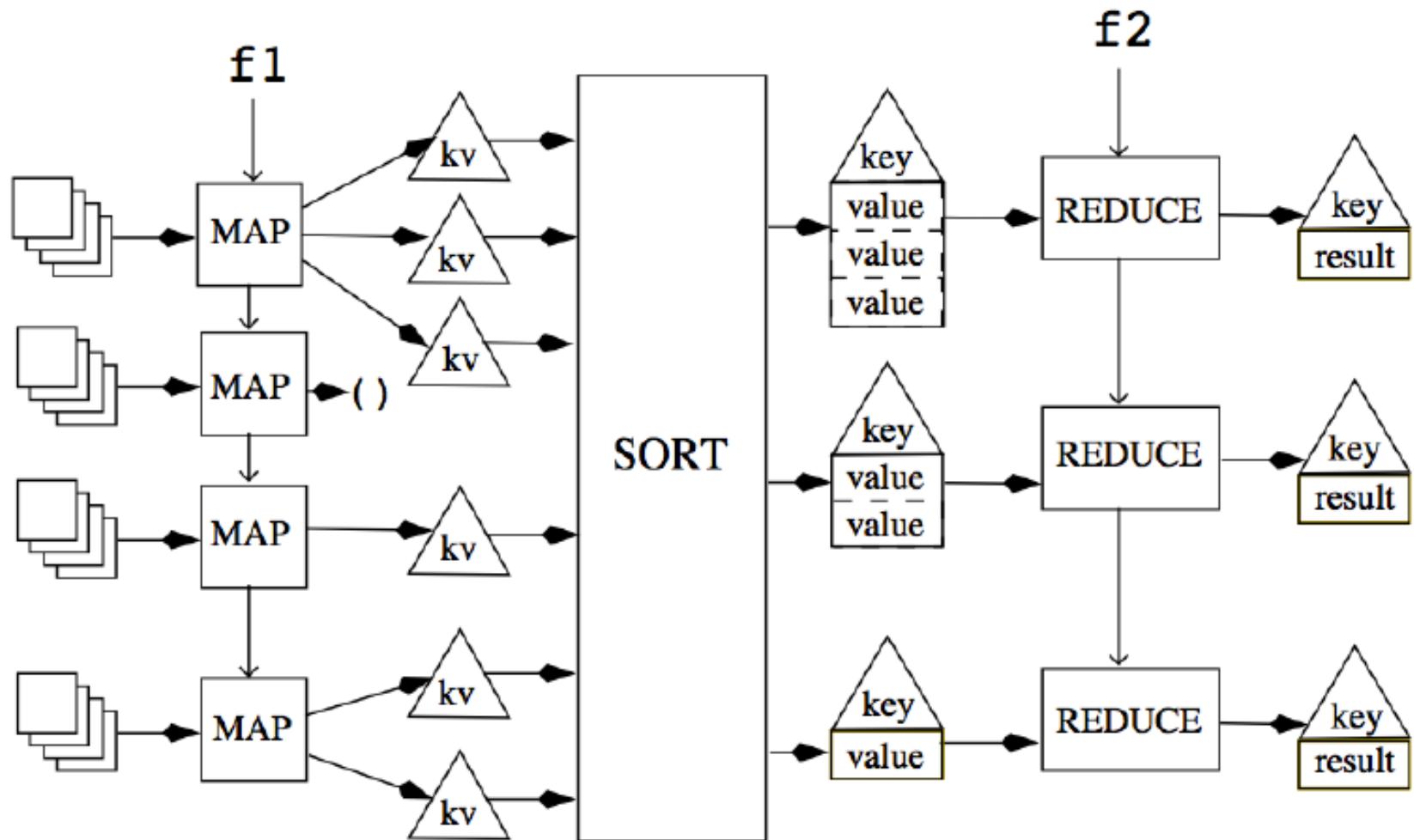
MAP: Each map gets a document. Mapper generates key/value pairs each time a word appears, i.e. $(word, "1")$

- **INPUT:** $(\text{FileName}, \text{FileContent})$, where FileName is the *key* and FileContent is the *value*
- **OUTPUT:** $\text{List}(\text{Word}, \text{WordAppearence})$, where Word is *key* and WordAppearence is the *value*

REDUCE: Combines the values per key and computes the sum

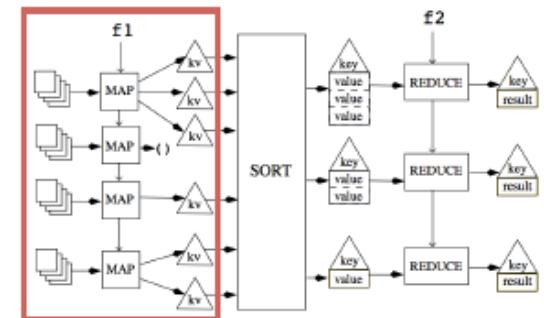
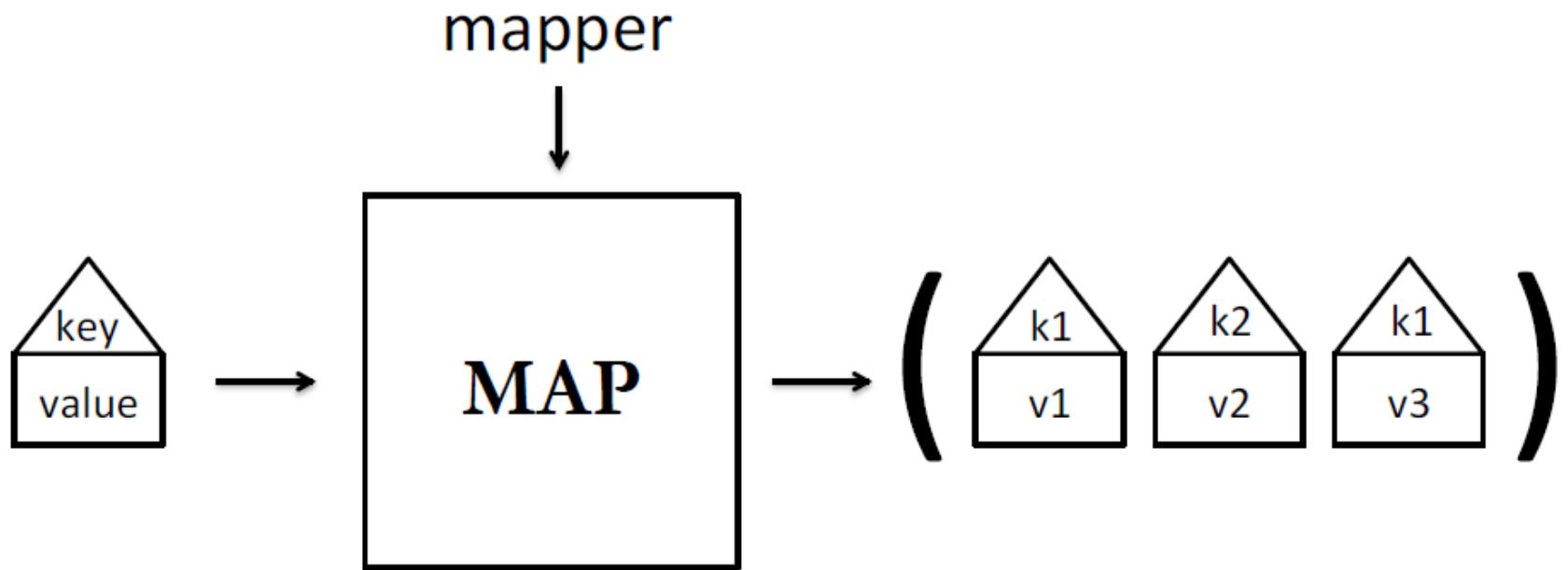
- **INPUT:** $(\text{Word}, \text{List}<\text{WordAppearence}>)$ where Word is the *key* and List<WordAppearence> are the *values*
- **OUTPUT:** $(\text{word}, \text{sum}<\text{WordAppearence}>)$ where Word is the *key* and sum<WordAppearence> is the *value*

MapReduce



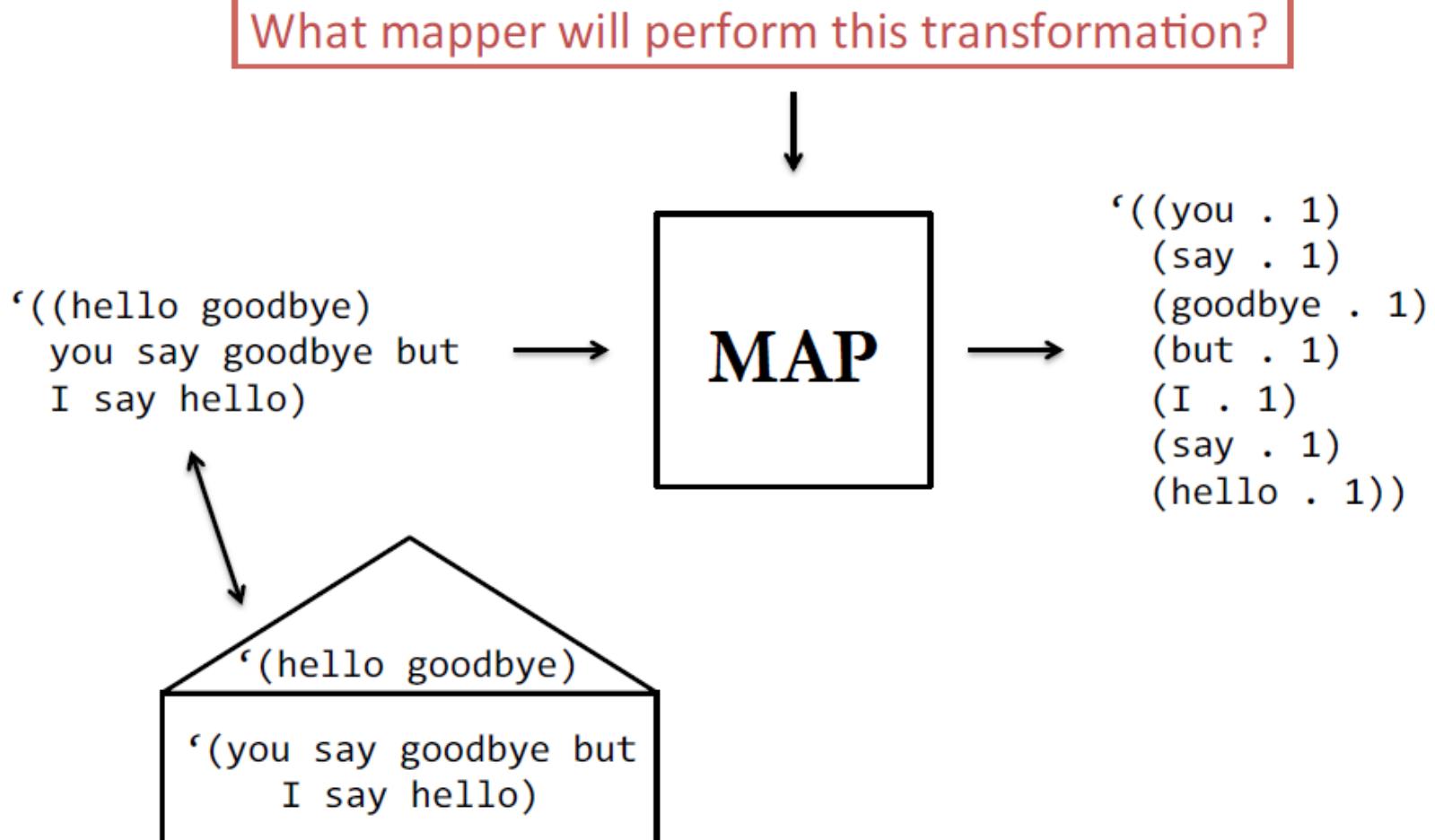
[Eric Tzeng]

MapReduce – Map Phase



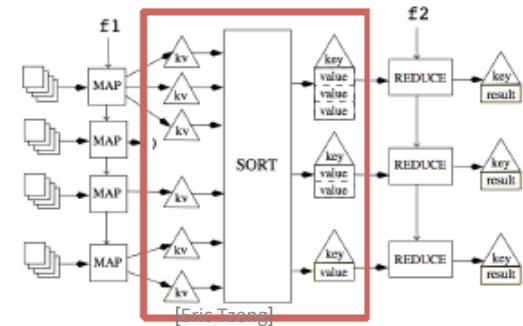
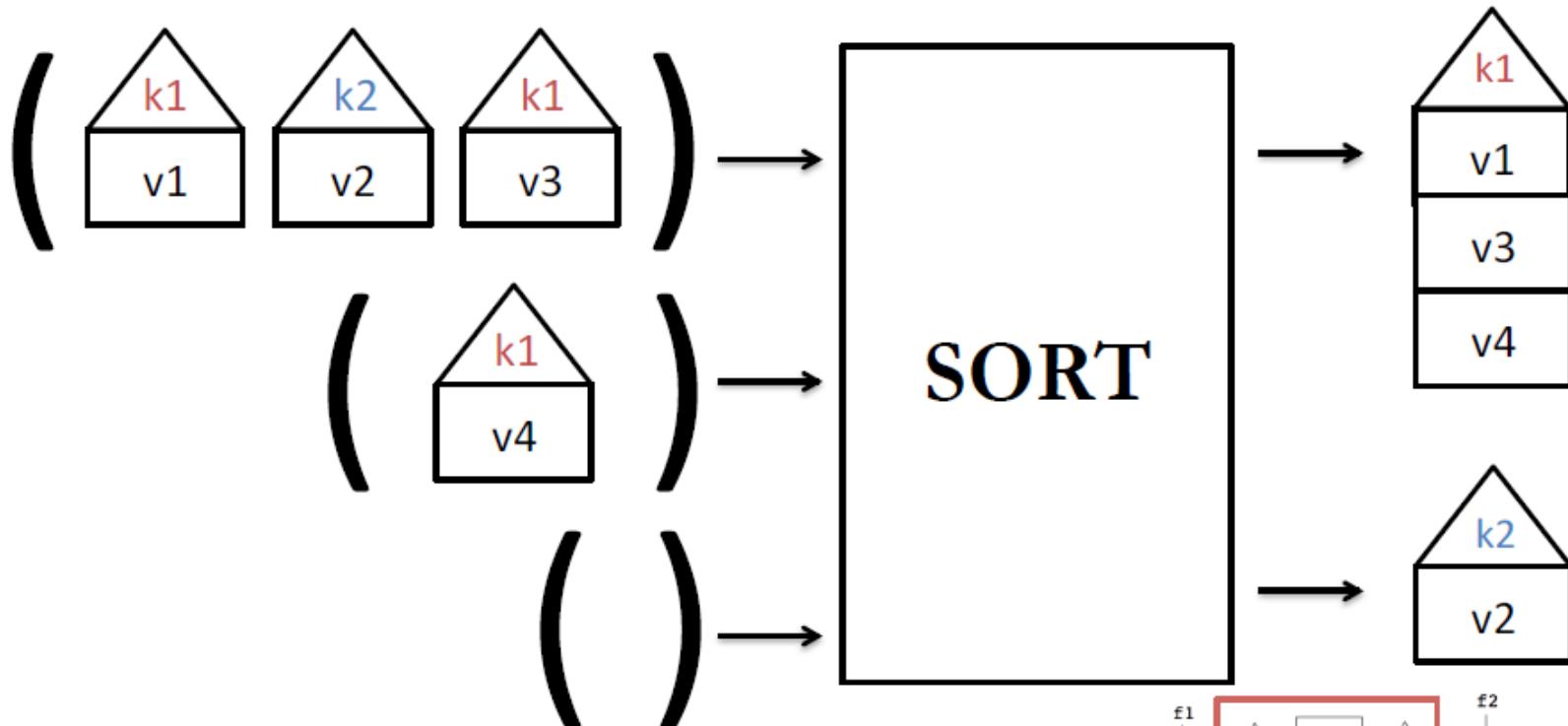
[Eric Tzeng]

Map Phase – Example: Word Count

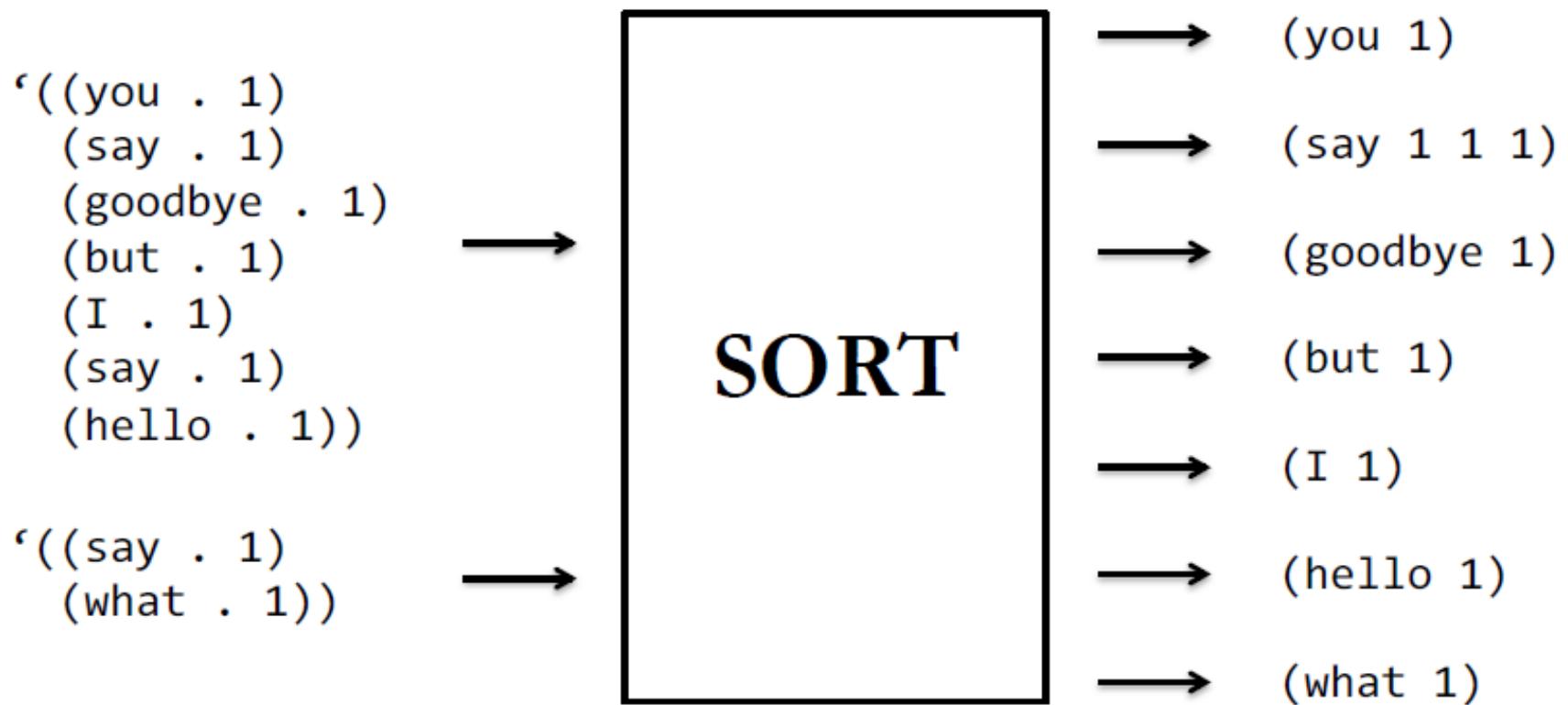


[Eric Tzeng]

MapReduce – Sort Phase

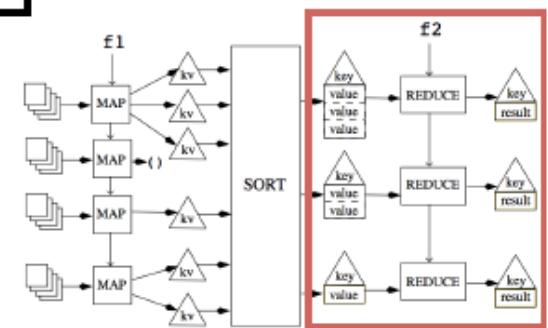
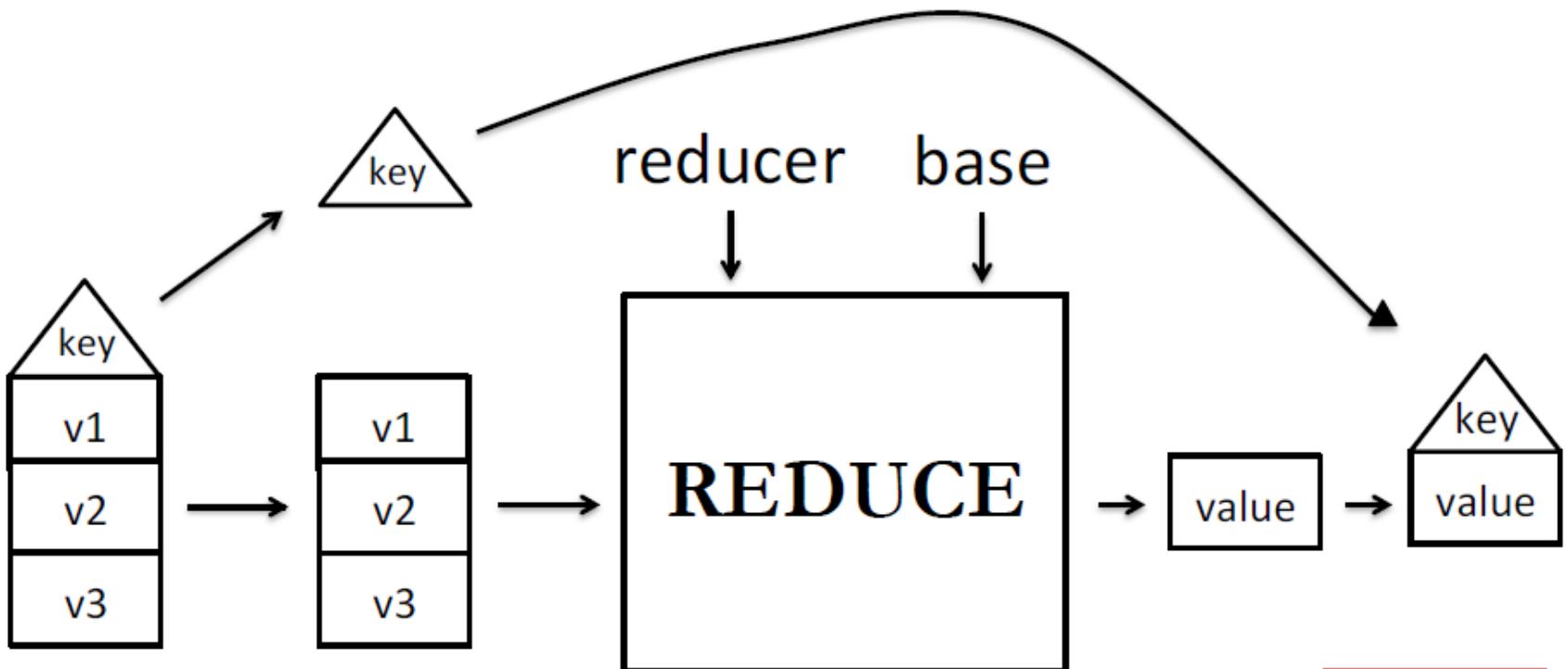


Sort Phase – Example: Word Count

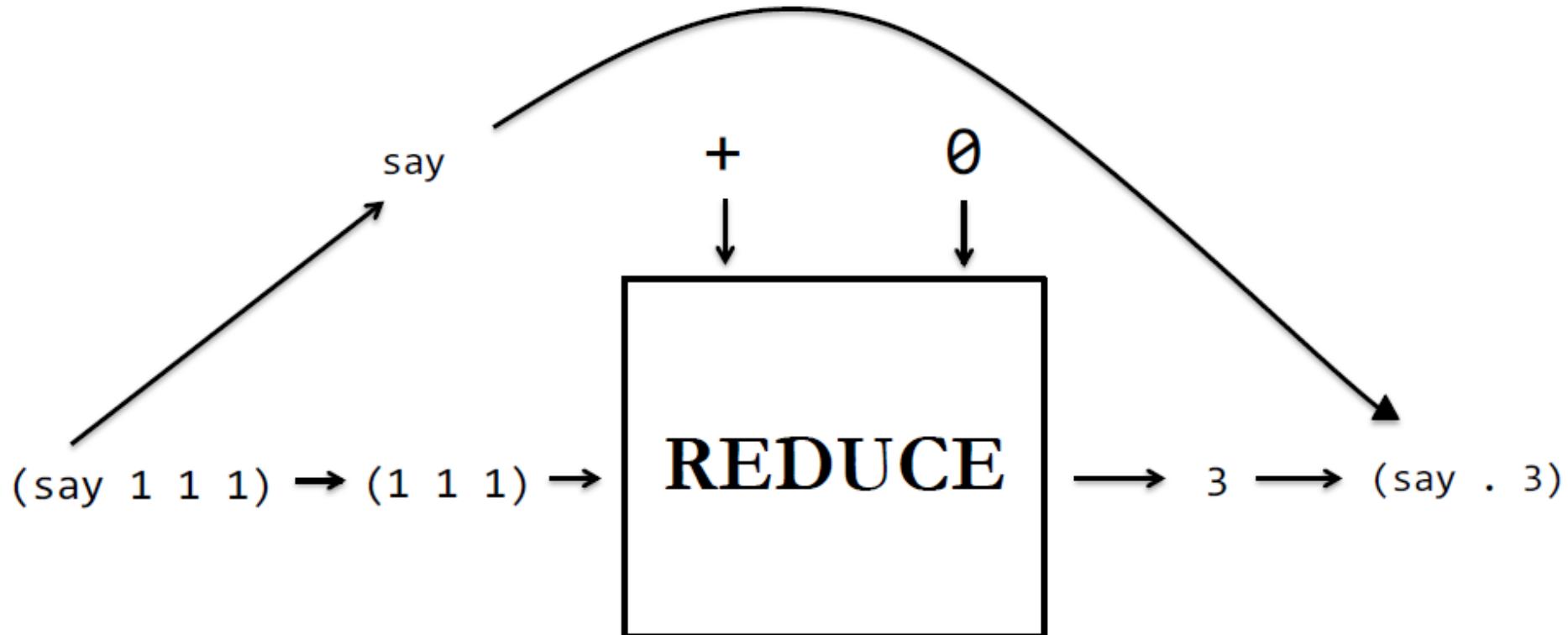


[Eric Tzeng]

MapReduce – Reduce Phase



Reduce phase – Example: Word Count



[Eric Tzeng]

Combiners

- Often a map task will produce many pairs of the form $(k, v_1), (k, v_2), \dots$ for the same key k
 - E.g., popular words in Word Count like (“the”, 1)
- For associative ops. like sum, count, max, can save bandwidth by pre-aggregating at mapper
- Decreases size of intermediate data
- Example: local counting for Word Count:

```
def combine(key, values): output(key, sum(values))
```

Partition Function

- Input to map are created by contiguous splits of input files
 - For reduce, we need to ensure that records with the same intermediate key end up at the same worker
 - System uses a default partition function: **hash(key) mod R**
 - Distributes the intermediate keys to reduce workers “randomly”
 - Sometimes useful to override
 - Balance load manually if distribution of keys known
 - Specific requirement on which key-value pair should be in the same output files
- def partition(key, number of partitions):** partition id for key

Parallelism

- map() tasks **run in parallel**, creating different intermediate values from different input data sets
- reduce() tasks **run in parallel**, each working on a different output key
- All values are **processed independently**, taking advantage of distribution
- **Bottleneck**: Reduce phase can't start until map phase is **completely finished**
- If some workers are slow, they slow down the entire process down: **Straggler** problem
- Start **redundant workers** and take result of fastest one

Google's MapReduce Implementation

- Runs on Google clusters (state 2004/5):
 - 100s-1000s of 2-CPU x86 machines, 2-4 GB of memory, local-based storage (GFS), limited bandwidth (commodity hardware)
- C++ library linked to user programs (on top of RPC)
- Scheduling/runtime system (a.k.a. master)
 - Assign tasks to machines: typically # map tasks > # of machines
- Often use 200,000 map/5,000 reduce tasks, 2000 machines
 - Minimizes time for fault recovery
 - Can pipeline shuffling with map execution
 - Better dynamic load balancing
- Other MapReduce implementations
 - Hadoop: Open-source, Java-based MapReduce framework
 - Phoenix: Open-source MapReduce framework for **multi-core**
 - Spark: MapReduce-like framework with **in-memory processing** in Scala

Locality Considerations

- Master divides up tasks based on location of data:
 - Tries to run map() tasks on same machine where physical input data resides (based on GFS)
 - At least same rack, if not same machine
- map() task inputs are divided into 64 MB blocks
 - Same size as GFS chunks
- GFS is Google File System
 - Distributed file system

Fault Tolerance & Optimizations

- In cluster with 1000s of machines, failures are common
- Worker failure
 - Detect failure via periodic heartbeating
 - Re-execute completed and in-progress *map* tasks
 - Re-execute in-progress *reduce* tasks
 - Task completion committed through master
- Design does not deal with master failures
- Optimization for fault-tolerance and load-balancing
 - Slow workers, **stragglers**, significantly lengthen completion time
 - Due to other jobs on machines, disk with errors, caching issues, ...
 - Other jobs consuming resources on machine
 - Solution: Near end of phase, spawn backup copies of tasks
 - Whichever one finishes first "wins"

Fault Tolerance & Optimizations cont'd

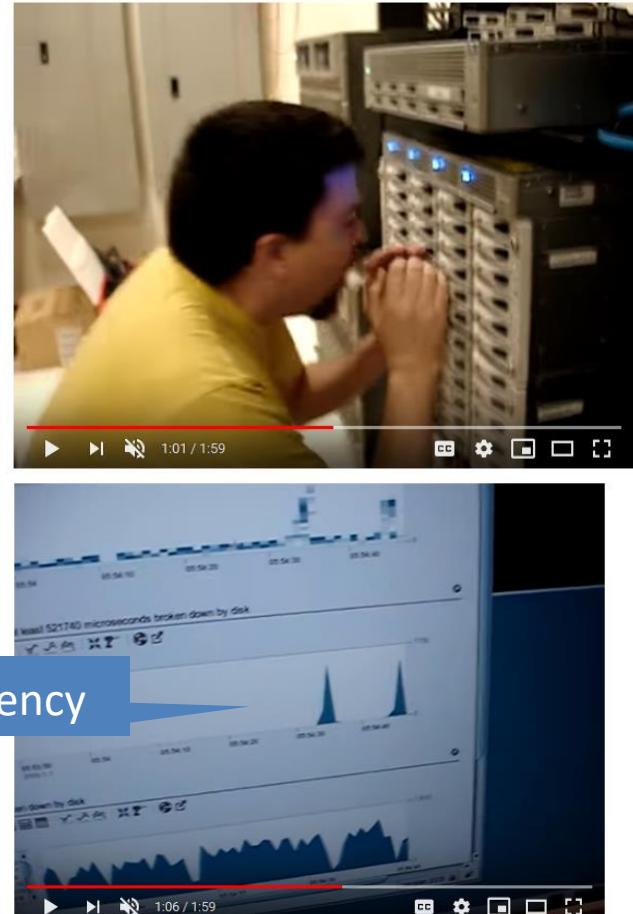
- Scheduling for locality
 - Asks GFS for locations of replicas of input file blocks
 - Map tasks schedule to node with copy of input data
 - Effect: Thousands of machines read input at local disk speed
- Skipping “bad” records in input
 - Map/Reduce functions sometimes fail for particular inputs
 - Best solution is to debug & fix, but not always possible because problems are in the input data and third party libraries
 - Solution
 - On seg fault, notify master about the record processed before fault
 - If master sees 2 failures for same record, it notifies workers to skip
 - Effect: Can work around bugs in third-party libraries
- Other
 - Compression of intermediate data

Criticism of MapReduce

- Too low level
 - Manual programming of per record manipulation
 - As opposed to declarative model (SQL)
- Nothing new
 - Map and reduce are classical Lisp or higher order functions
- Low per node performance
 - Due to replication and data transfer
 - Expensive shuffling process (to be minimized if possible)
 - A lot of I/O to GFS
- Batch computing, not designed for incremental/streaming tasks
 - Data must be available before job is started
 - Cannot add more input during job execution

What are Stragglers?

- Problem: Reduce phase can only begin when all map jobs are completed
- Mitigation of stragglers
 - Speculatively executing duplicate work on multiple systems
 - Enabled per default in Hadoop for MapReduce Jobs
- Reduce sources of straggling
 - Reduce bursts on the network through different configurations
 - Reduce data structure contention
 - Coordination of garbage collection (e.g., in Java)



Shouting in the Datacenter
1,674,884 views 9.6K likes 111 dislikes SHARE SAVE ...

 Bryan Cantrill
Published on Dec 31, 2008

SUBSCRIBE 1.9K

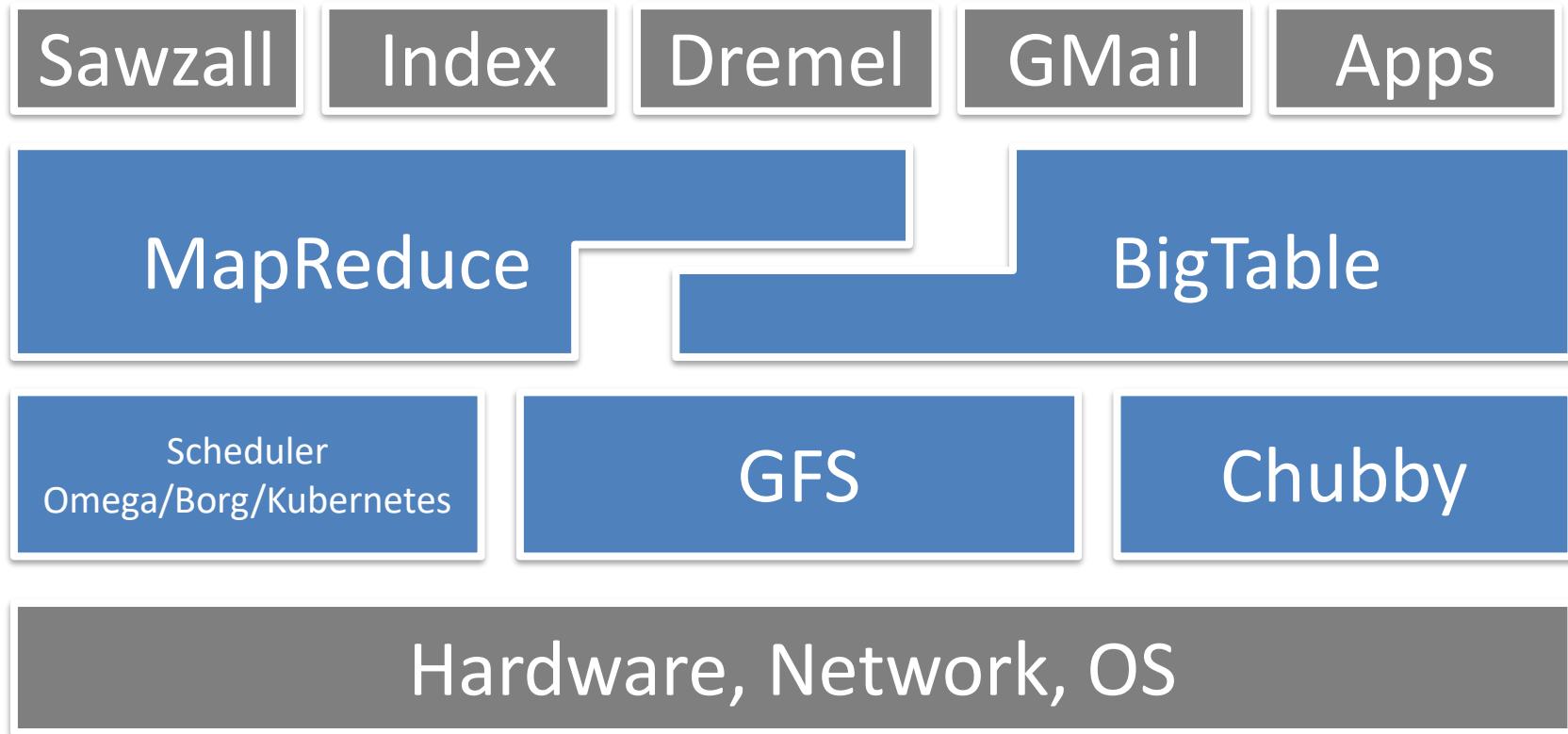
[Link](#) [Blog](#)

GOOGLE AND HADOOP ECOSYSTEMS

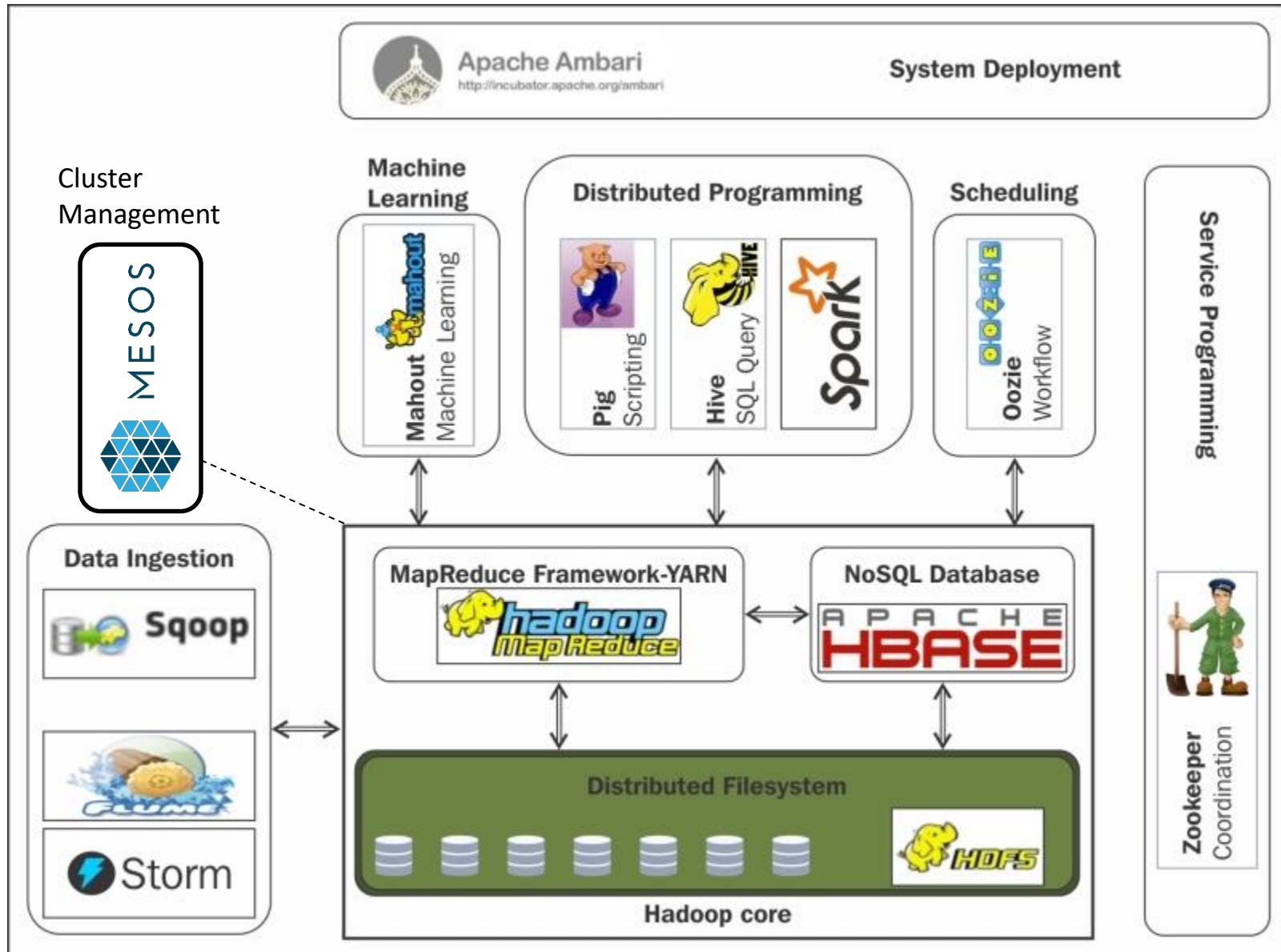
Beyond MapReduce

“Unfortunately, no one can be told what the Matrix is. You have to see it for yourself.”

Google's stack

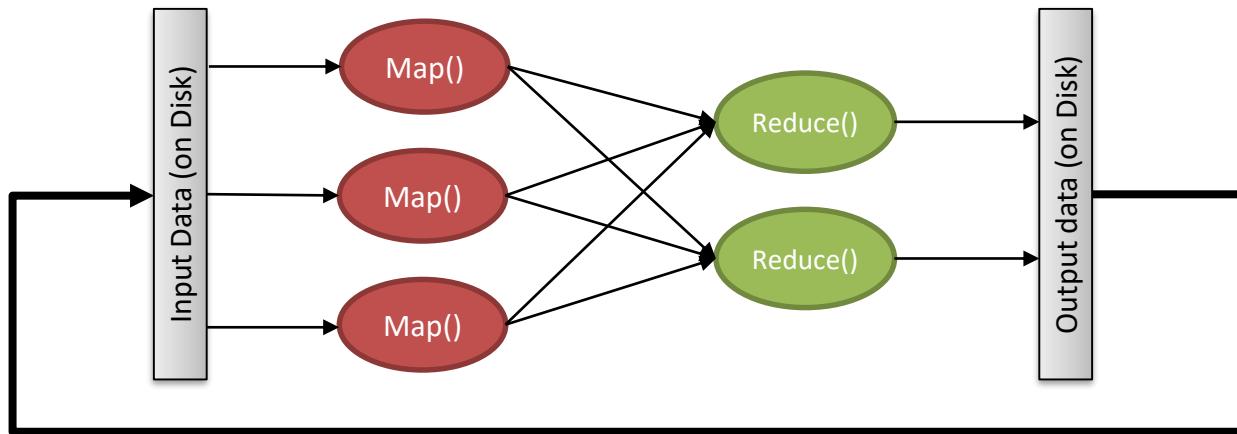


Hadoop Ecosystem

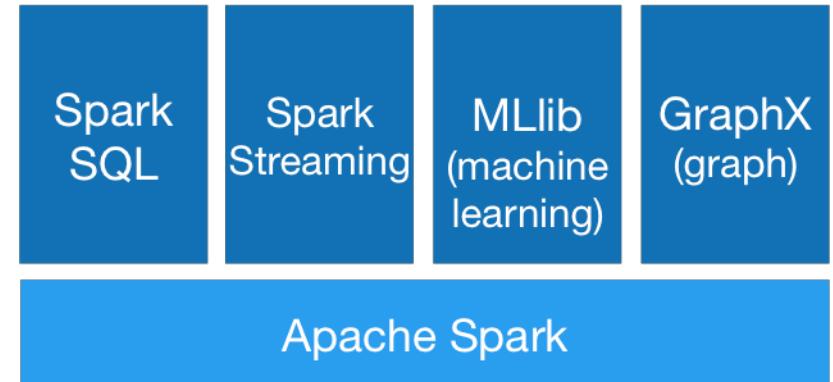
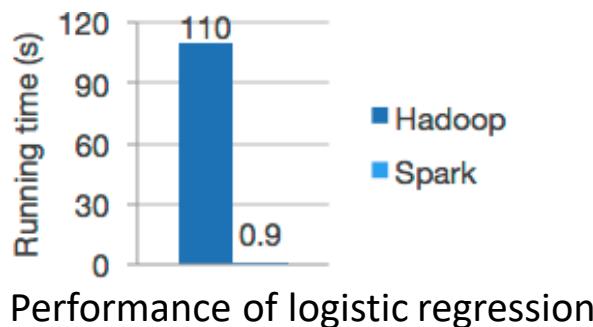


Iterative MapReduce

- Iterative algorithms: Repeat map and reduce jobs
- Examples: PageRank, SVM and many more



- In MapReduce, the only way to share data across jobs is stable storage (i.e., via GFS)
- Some proposed solutions include: Twister and **Spark**

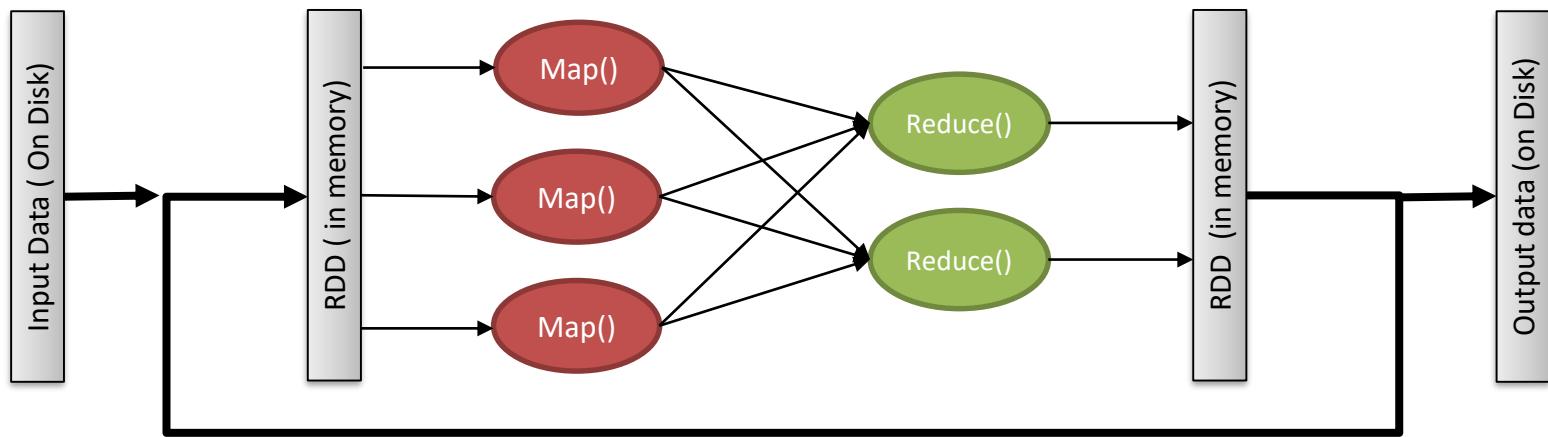


- Up to 100x faster than Hadoop MapReduce
- Several programming interfaces (Java, Scala, Python, R)
- Powers a stack of useful libraries (see picture to the right)
- Runs on top of a Hadoop cluster (uses HDFS)
- In-memory computation vs. stable storage (MapReduce)
- Fault tolerance (for crashes & stragglers)
- Extremely well documented

Spark Basics

Based on the concept of a **resilient distributed dataset (RDD)**:

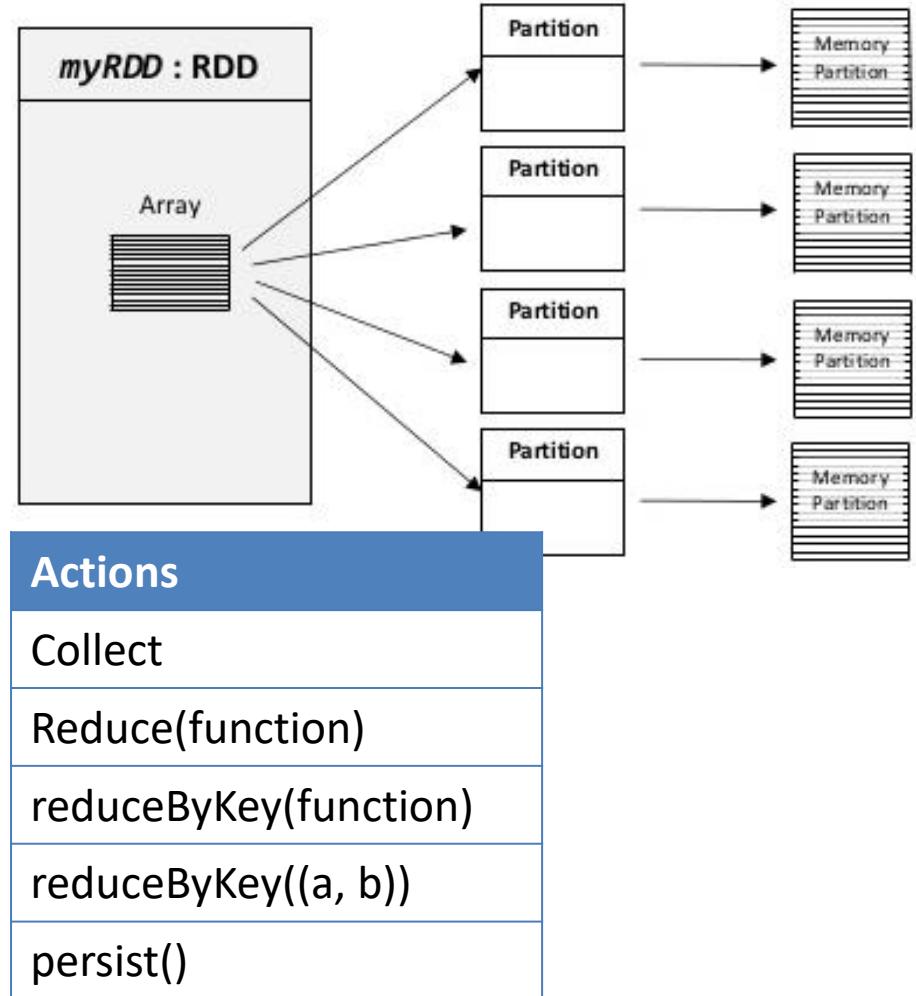
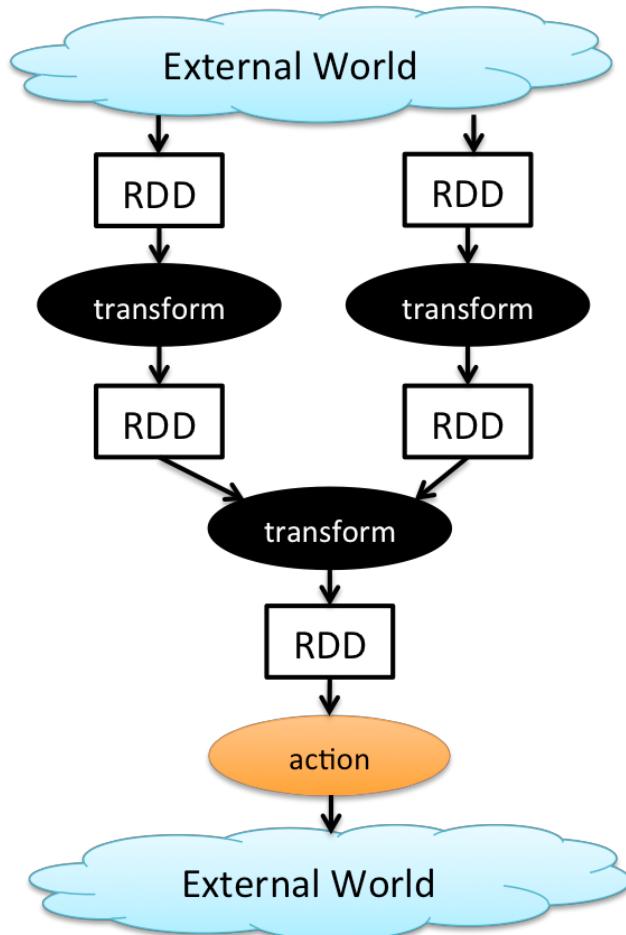
- **Read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost**
- Created by transforming data in RDDs using data flow operators (map, filter, group by, ...)
- A handle to an RDD contains enough information to compute the RDD starting from data in reliable storage
- Can be cached across parallel operations



RDDs

- Good fit for batch applications that apply the same operation to all elements of a dataset
- Less suitable for applications that make asynchronous fine grained updates to shared state (storage system for web applications)

Spark Processing Flow



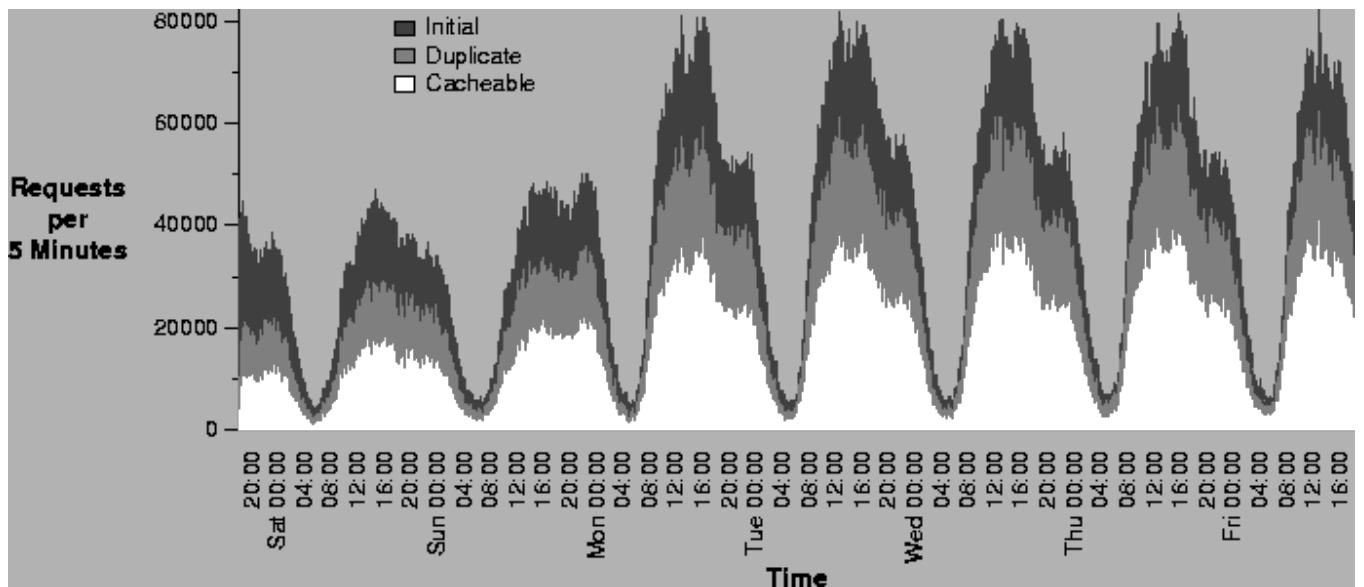
Spark Examples

Word Count

```
text_file = spark.textFile("hdfs://...")  
counts = text_file.flatMap(lambda line: line.split(" ")) \  
    .map(lambda word: (word, 1)) \  
    .reduceByKey(lambda a, b: a + b)  
counts.saveAsTextFile("hdfs://...")
```

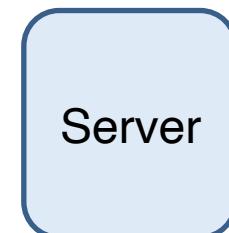
Consistent Hashing

*Organization-Based
Analysis
of Web-Object Sharing and
Caching*
Alec Wolma et al.



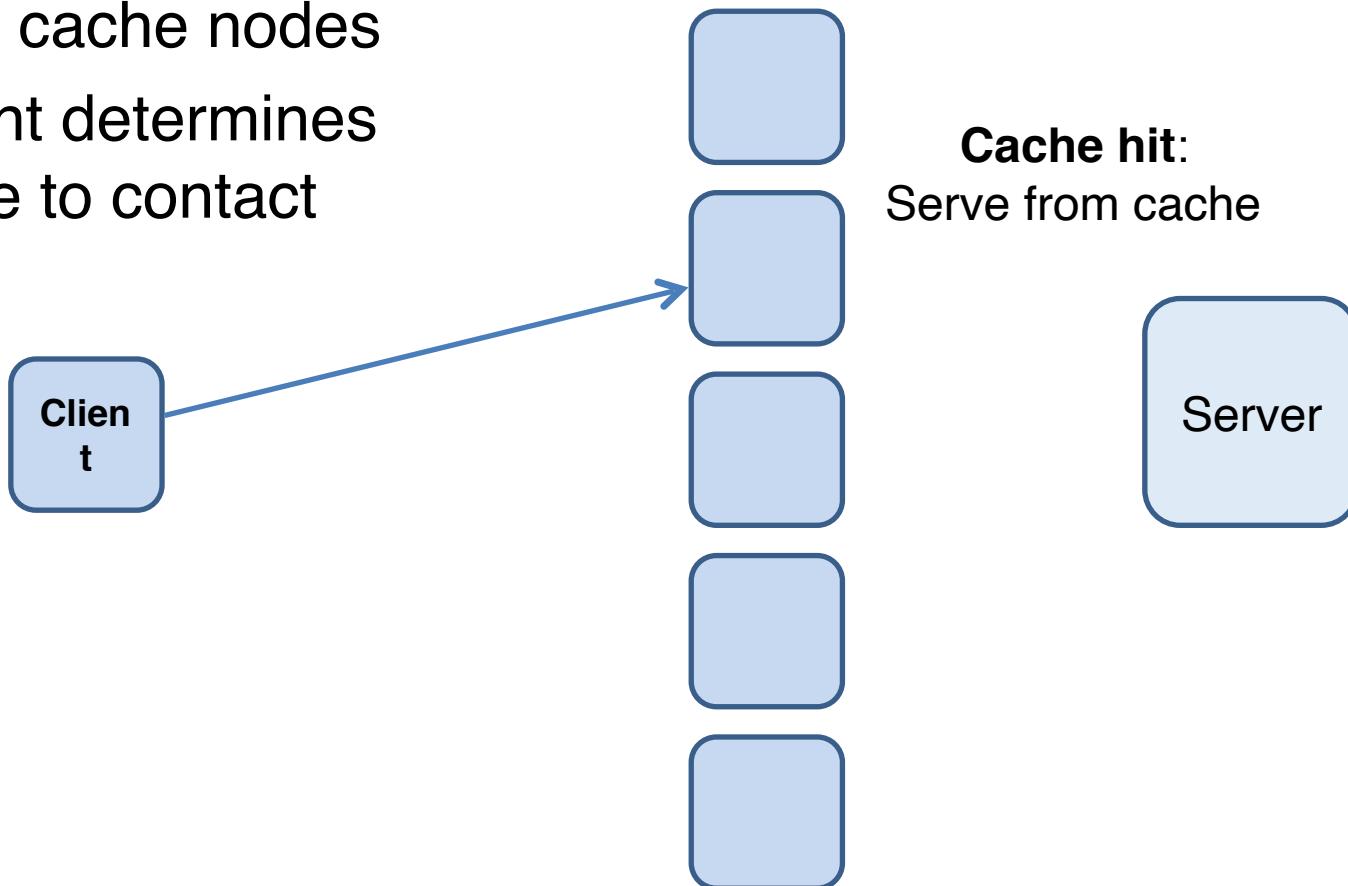
Caching

- Five cache nodes
- Client determines node to contact



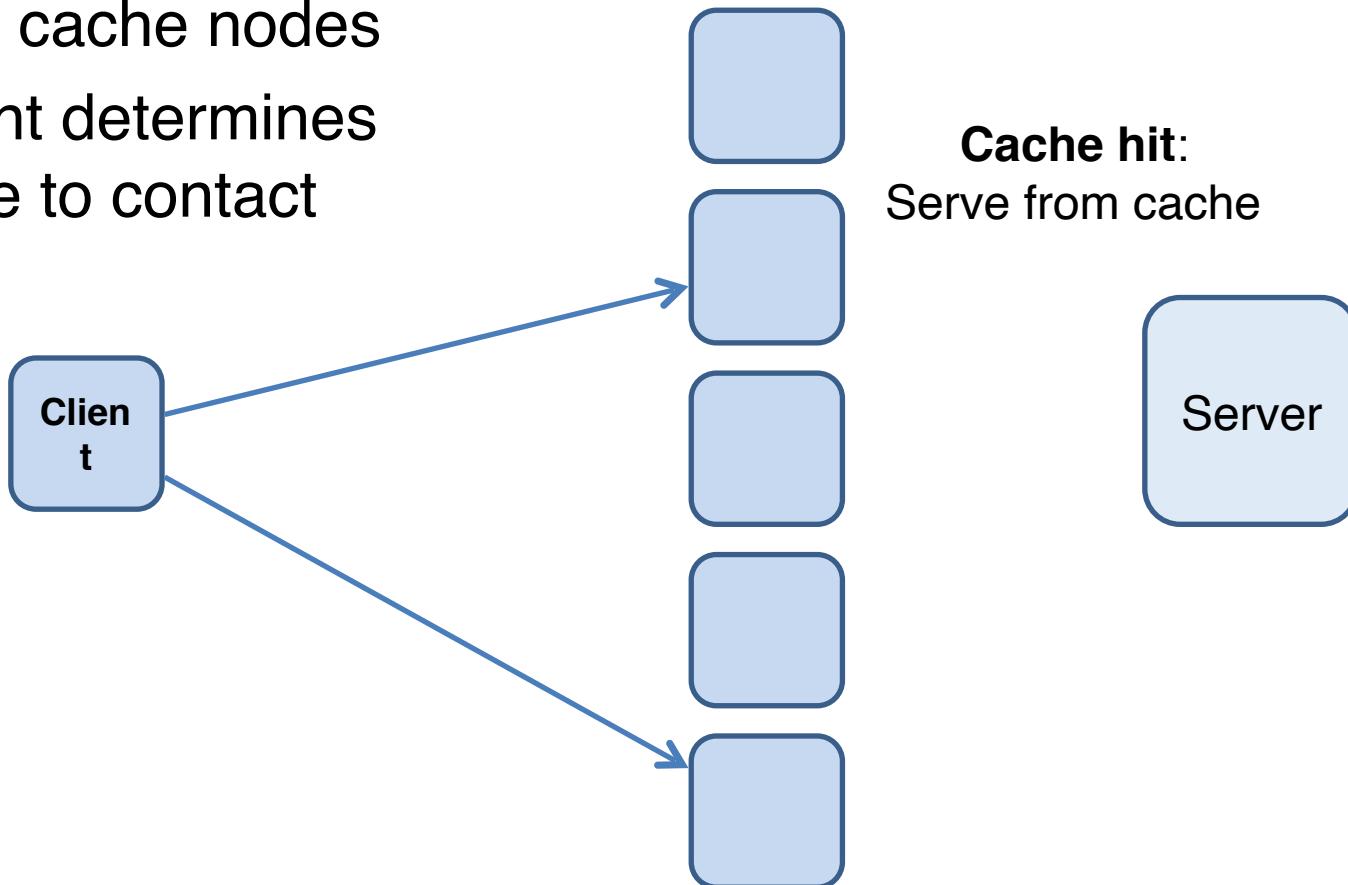
Caching

- Five cache nodes
- Client determines node to contact



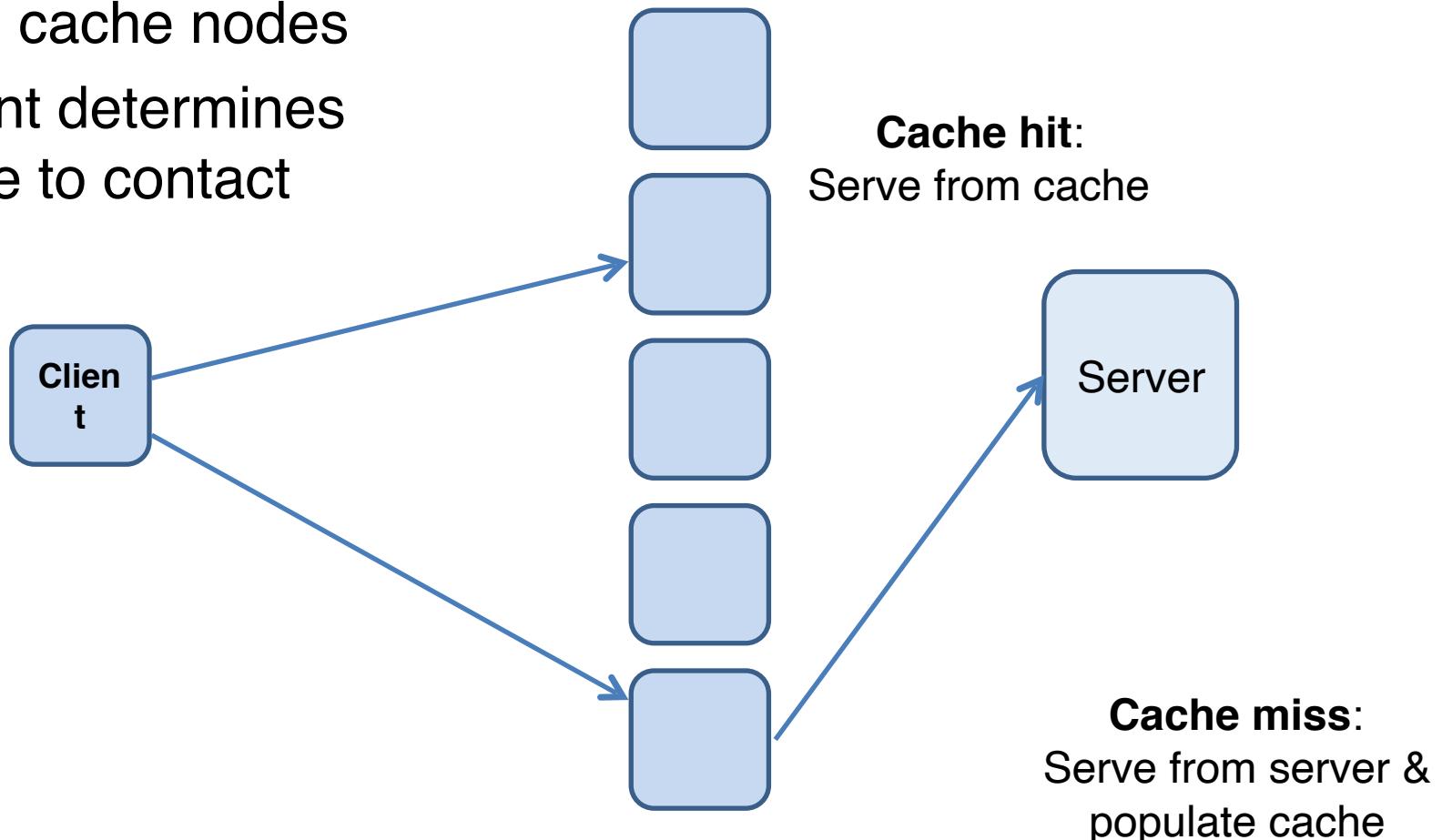
Caching

- Five cache nodes
- Client determines node to contact



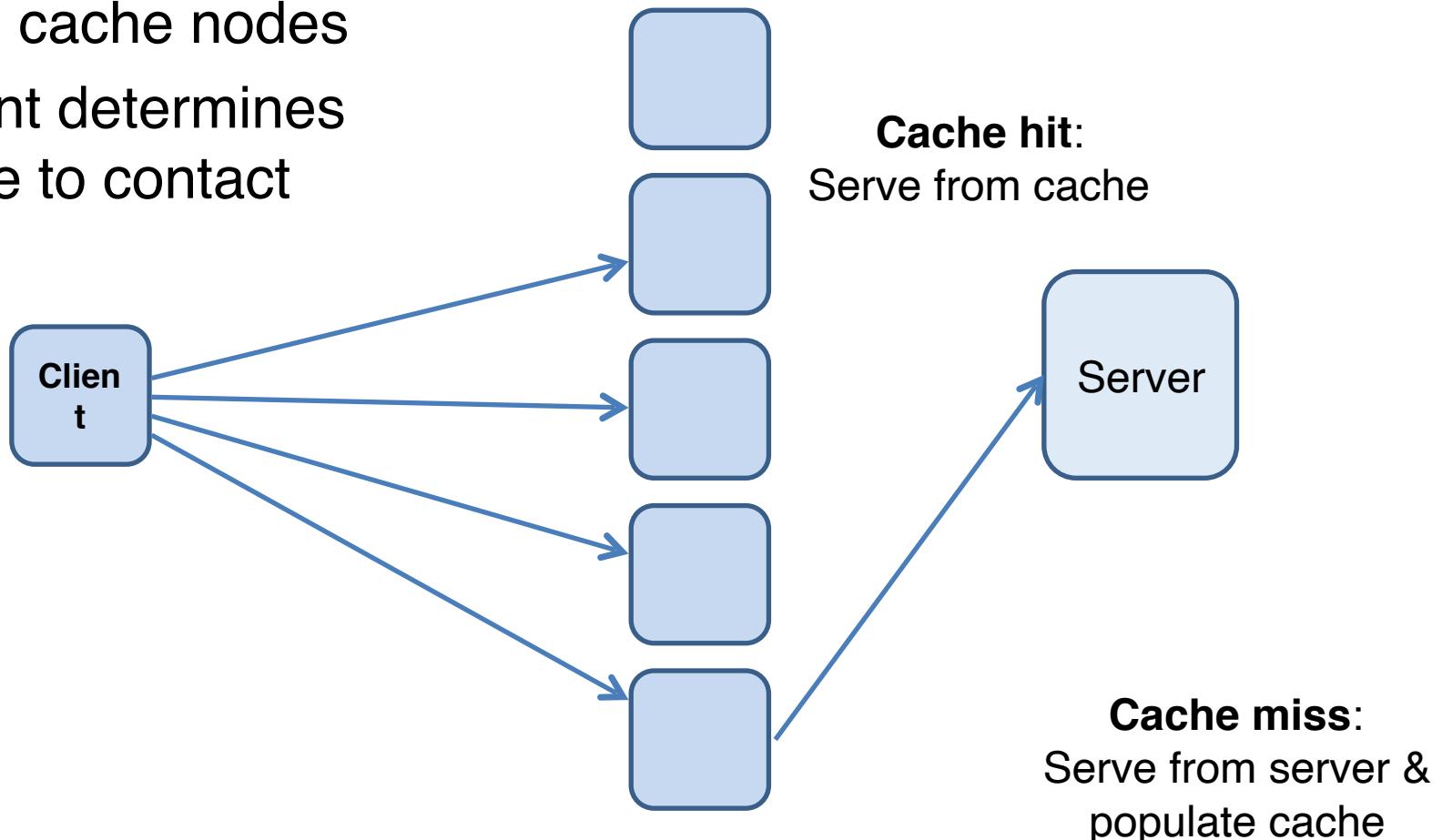
Caching

- Five cache nodes
- Client determines node to contact



Caching

- Five cache nodes
- Client determines node to contact



Problem: Mapping objects to caches

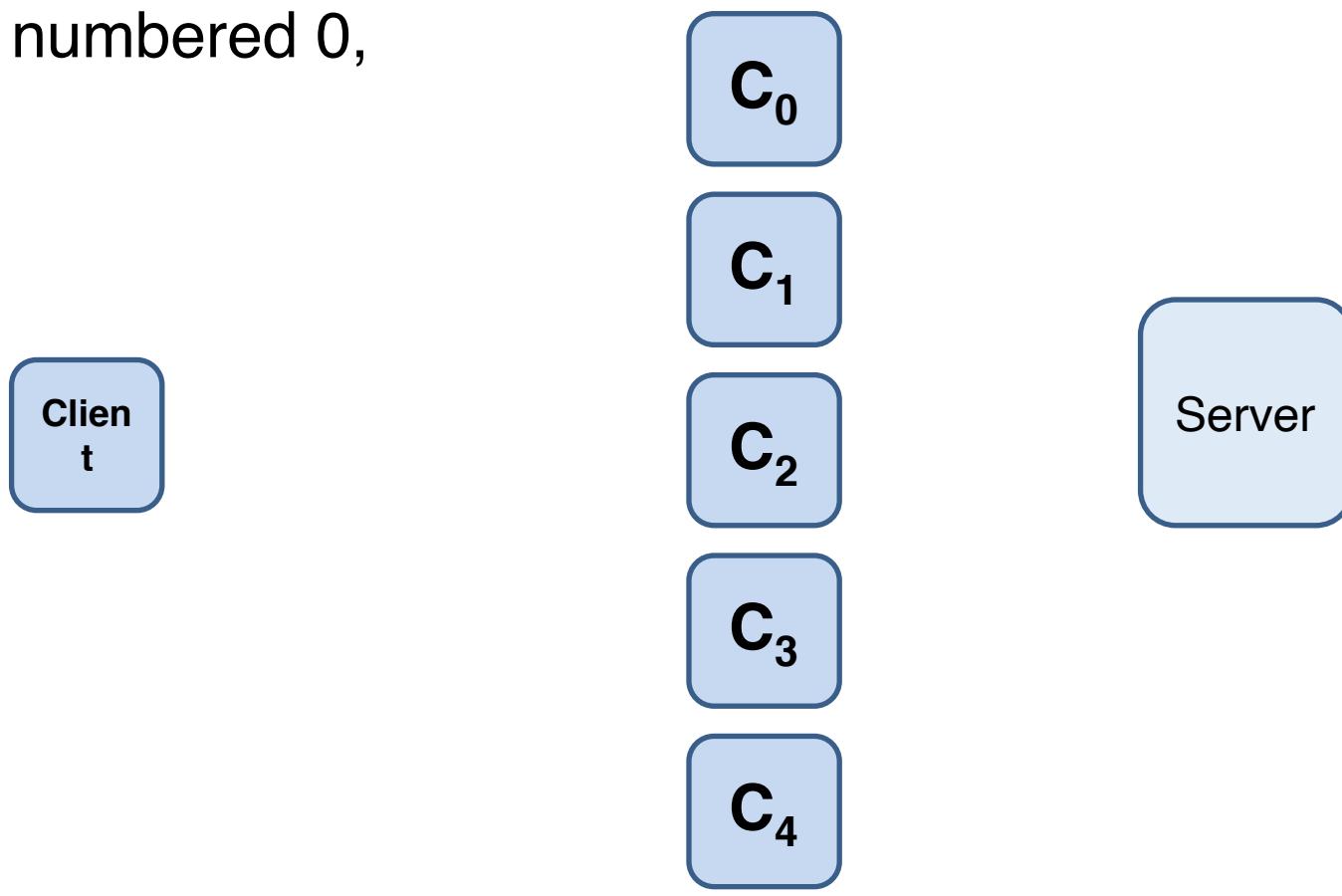
- Given a number of caches (e.g., cooperative caching, CDNs, etc.)
- Each cache should carry an **equal share of objects**
- **Clients** need to **know what cache to query** for a given object
- **Horizontally partition** (shard) object ID space
 - Doesn't work with **skewed distributions**: e.g., 10 servers, each handles 100 IDs, but all objects have IDs between 1-100 or 900-1000
- **Caches** should be able to **come and go** without disrupting the whole operation (i.e., non-effected caches)

Solution attempt: Use hashing

- **Map object ID (e.g., URL u) into one of the caches**
- Use a hash function that **maps u to node $h(u)$**
 - For example, $h(x) = (ax + b) \text{ mod } p$, where p is range of $h(x)$, i.e., the number of caches
 - Interpret u as a number based on bit pattern of object ID (or URL)
- Hashing tends to **distribute input uniformly** across range of hash function
 - Objects (URLs) are **equally balanced** across caches, even if object IDs are skewed (i.e., highly clustered in ID space)
- No one cache responsible for an **uneven share of objects/URLs**
- No disproportionately loaded node (potential bottleneck)

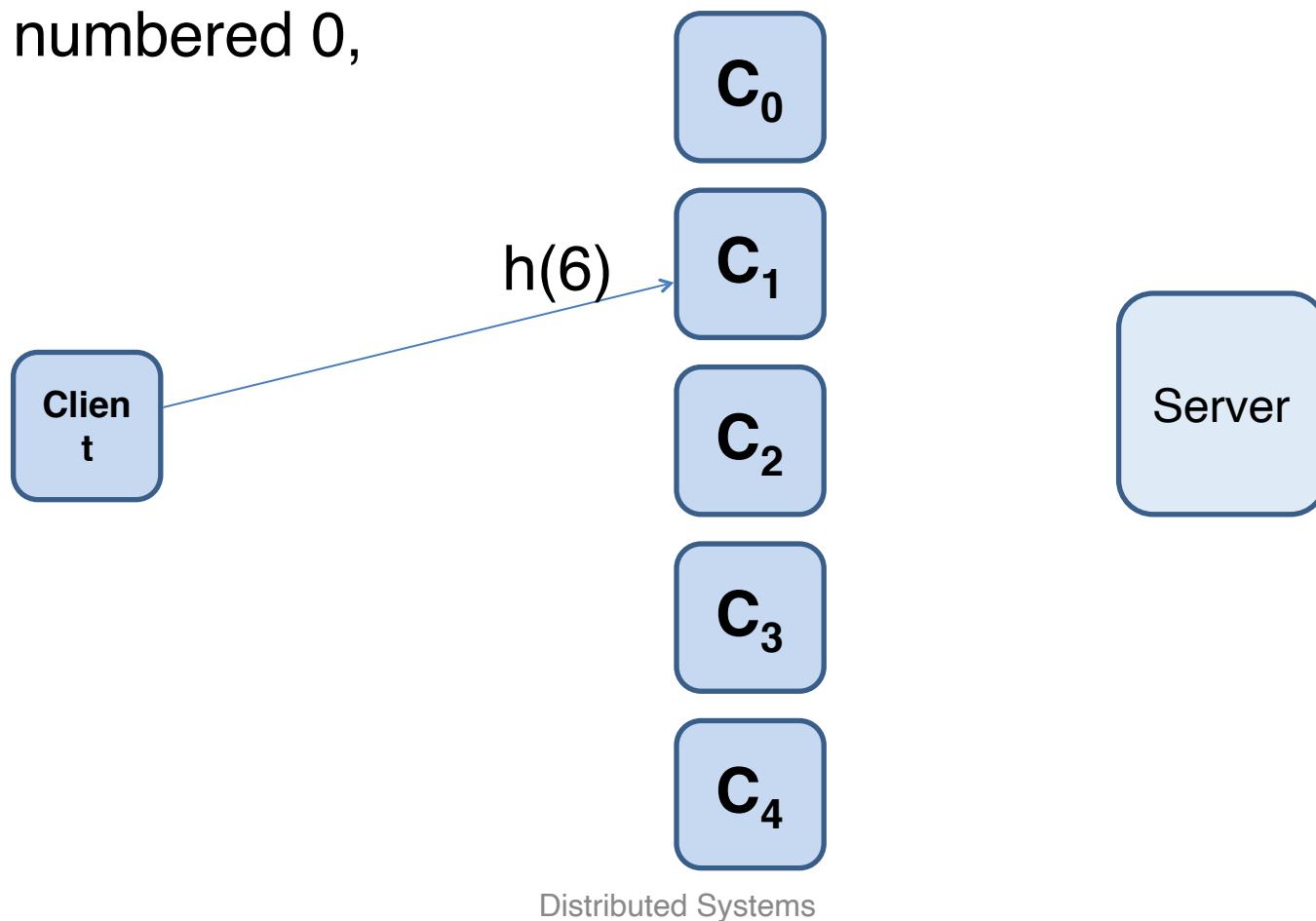
$$h(u) = (7u + 4) \bmod 5$$

Assume, we have **five caches**, numbered 0, ..., 4.



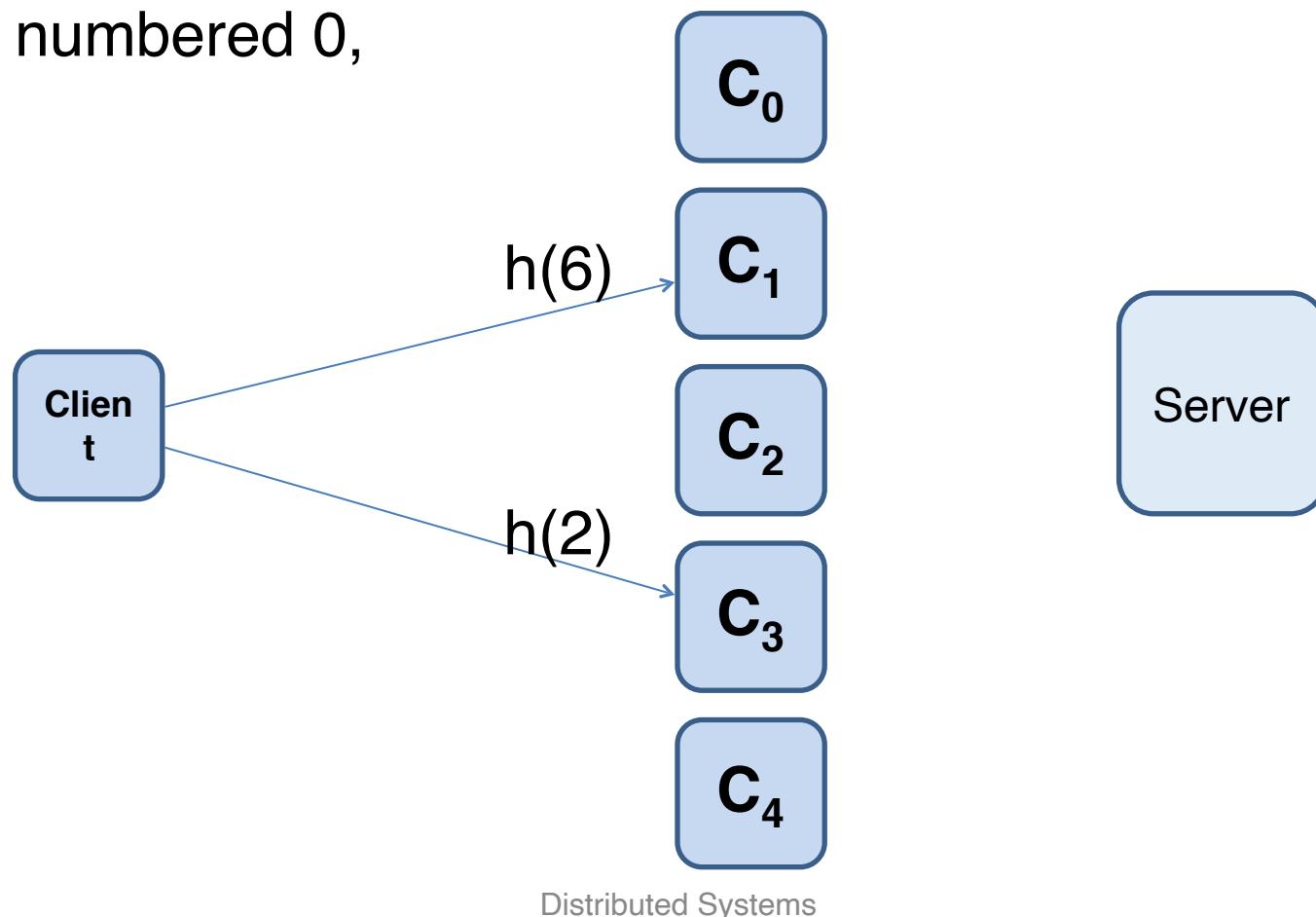
$$h(u) = (7u + 4) \bmod 5$$

Assume, we have **five caches**, numbered 0, ..., 4.



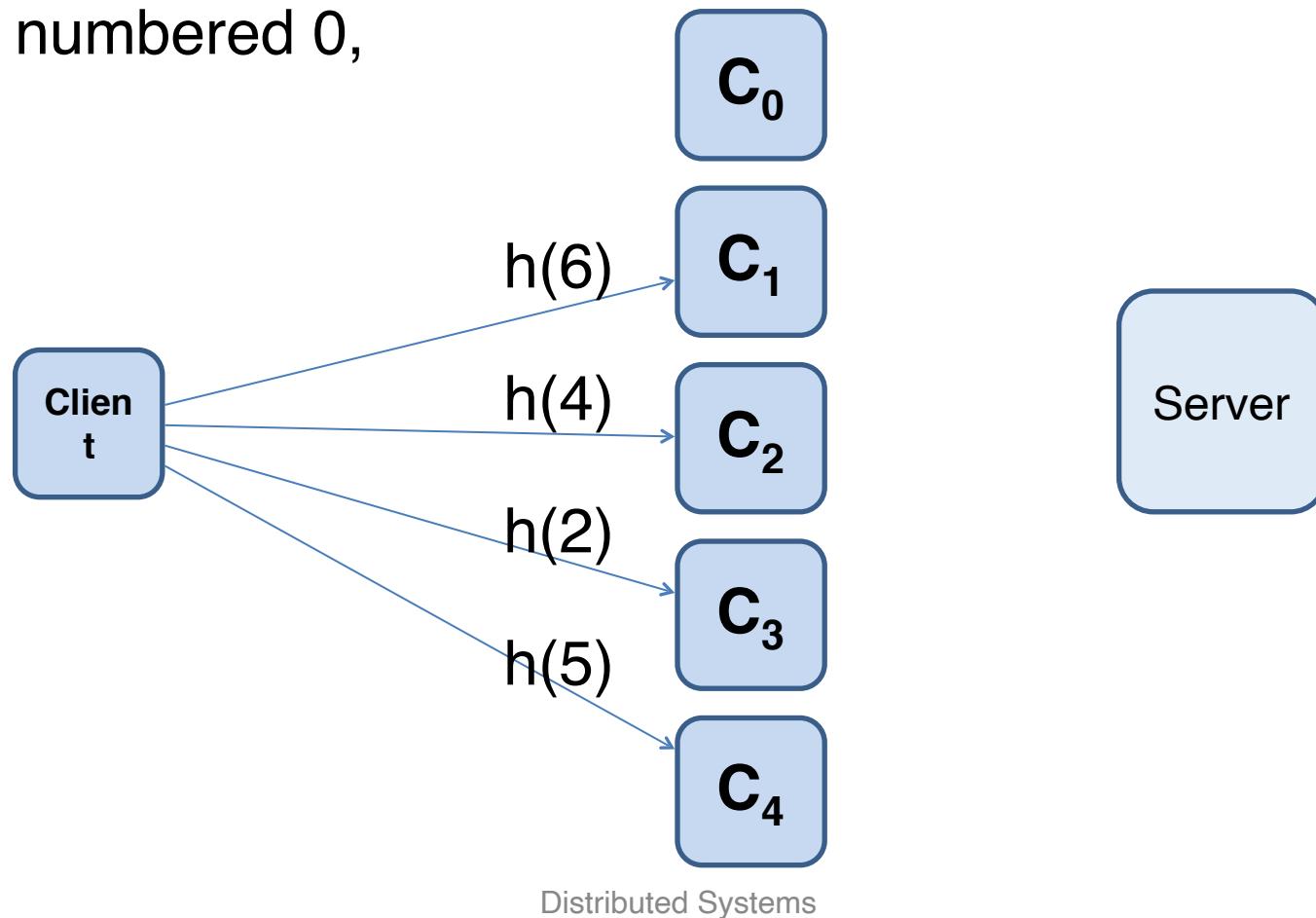
$$h(u) = (7u + 4) \bmod 5$$

Assume, we have **five caches**, numbered 0, ..., 4.



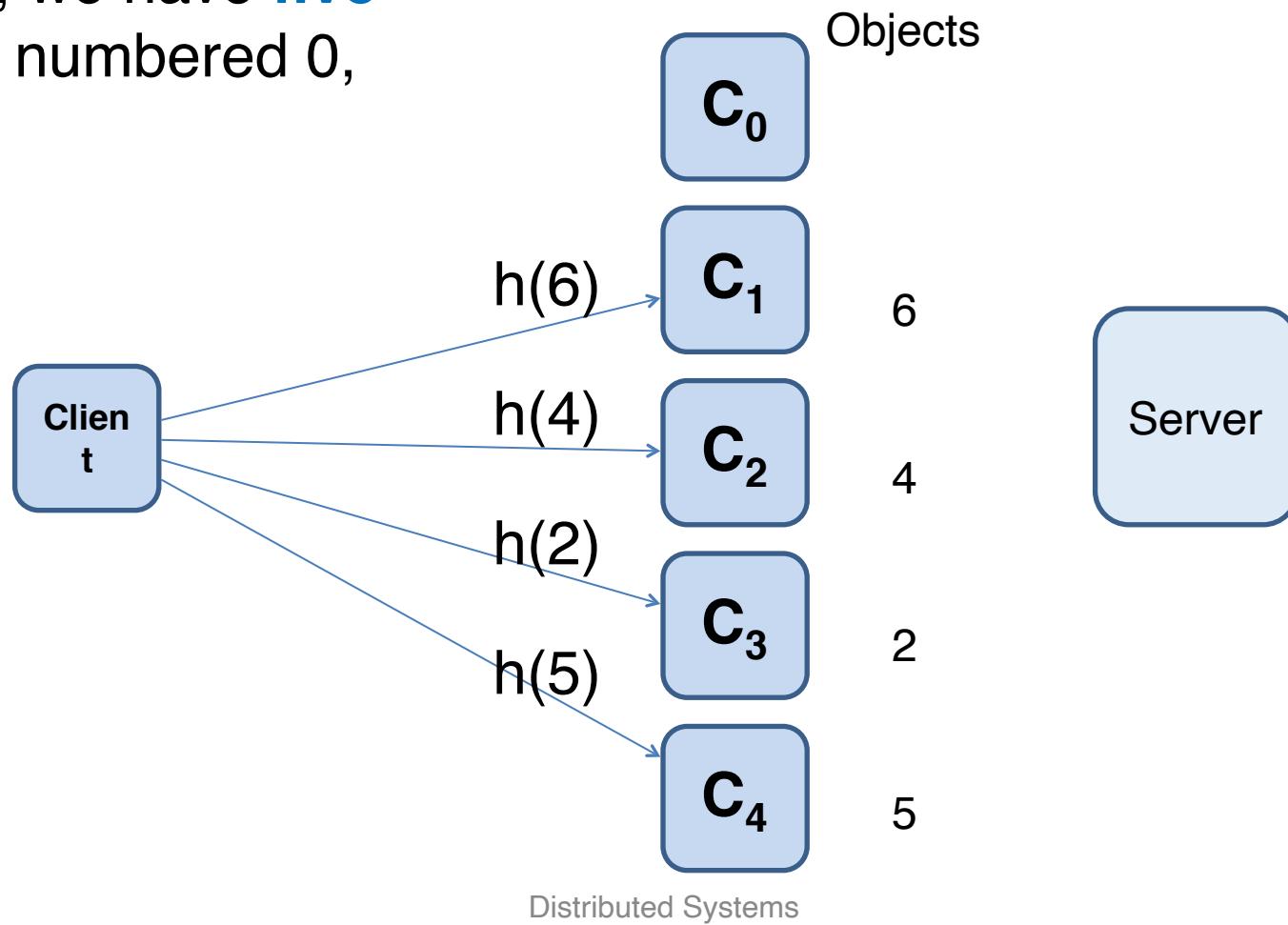
$$h(u) = (7u + 4) \bmod 5$$

Assume, we have **five caches**, numbered 0, ..., 4.



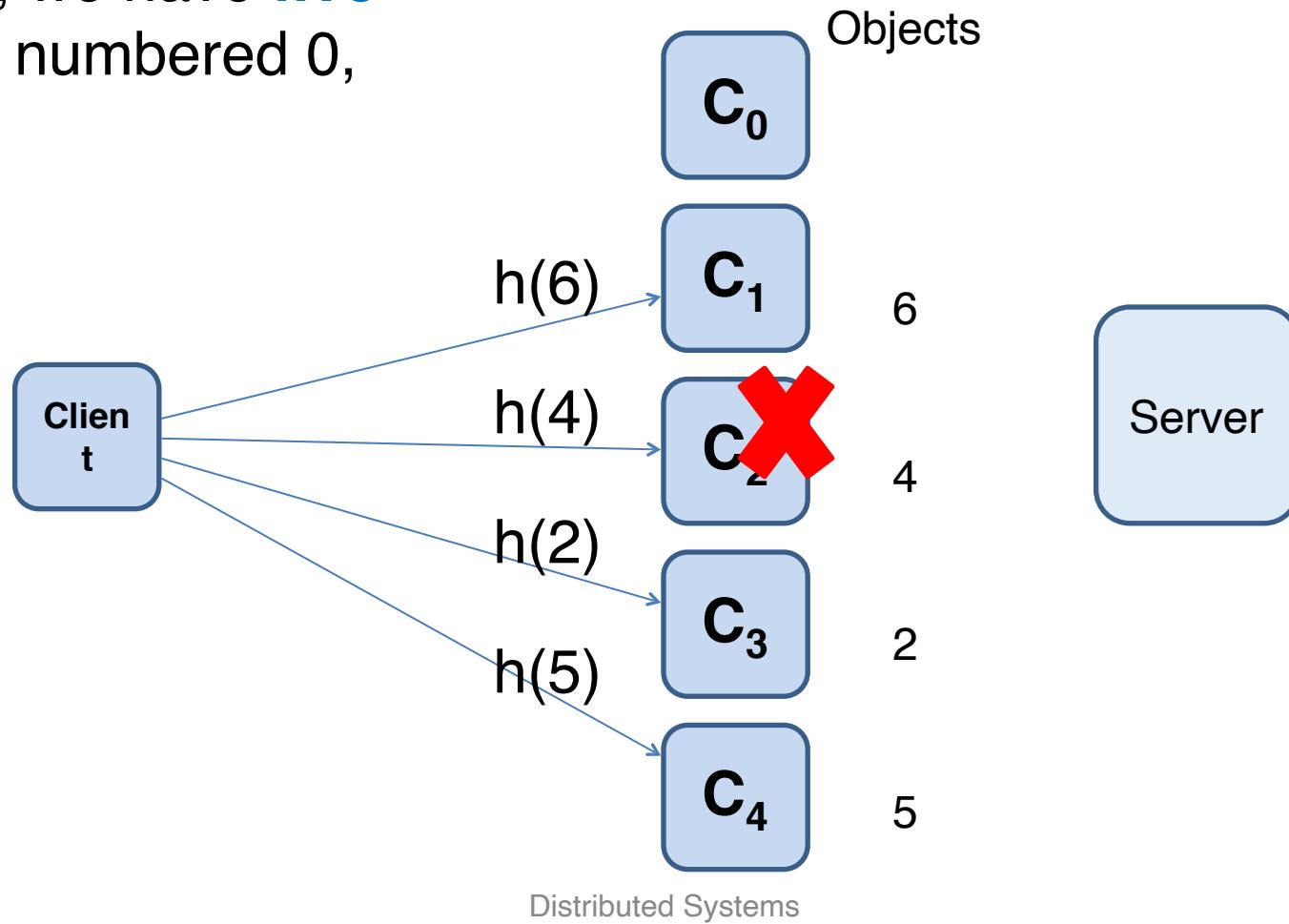
$$h(u) = (7u + 4) \bmod 5$$

Assume, we have **five caches**, numbered 0, ..., 4.



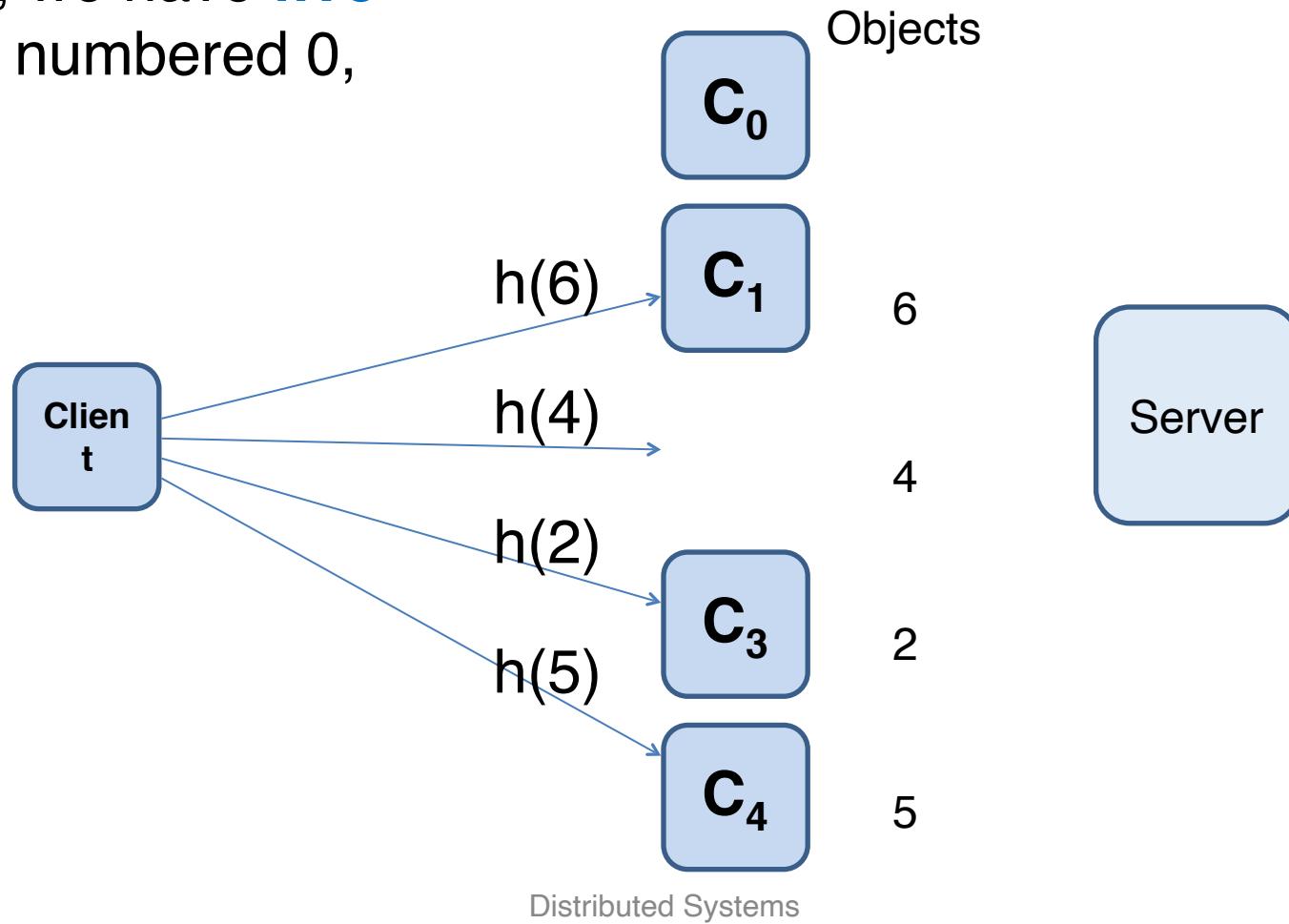
$$h(u) = (7u + 4) \bmod 5$$

Assume, we have **five caches**, numbered 0, ..., 4.



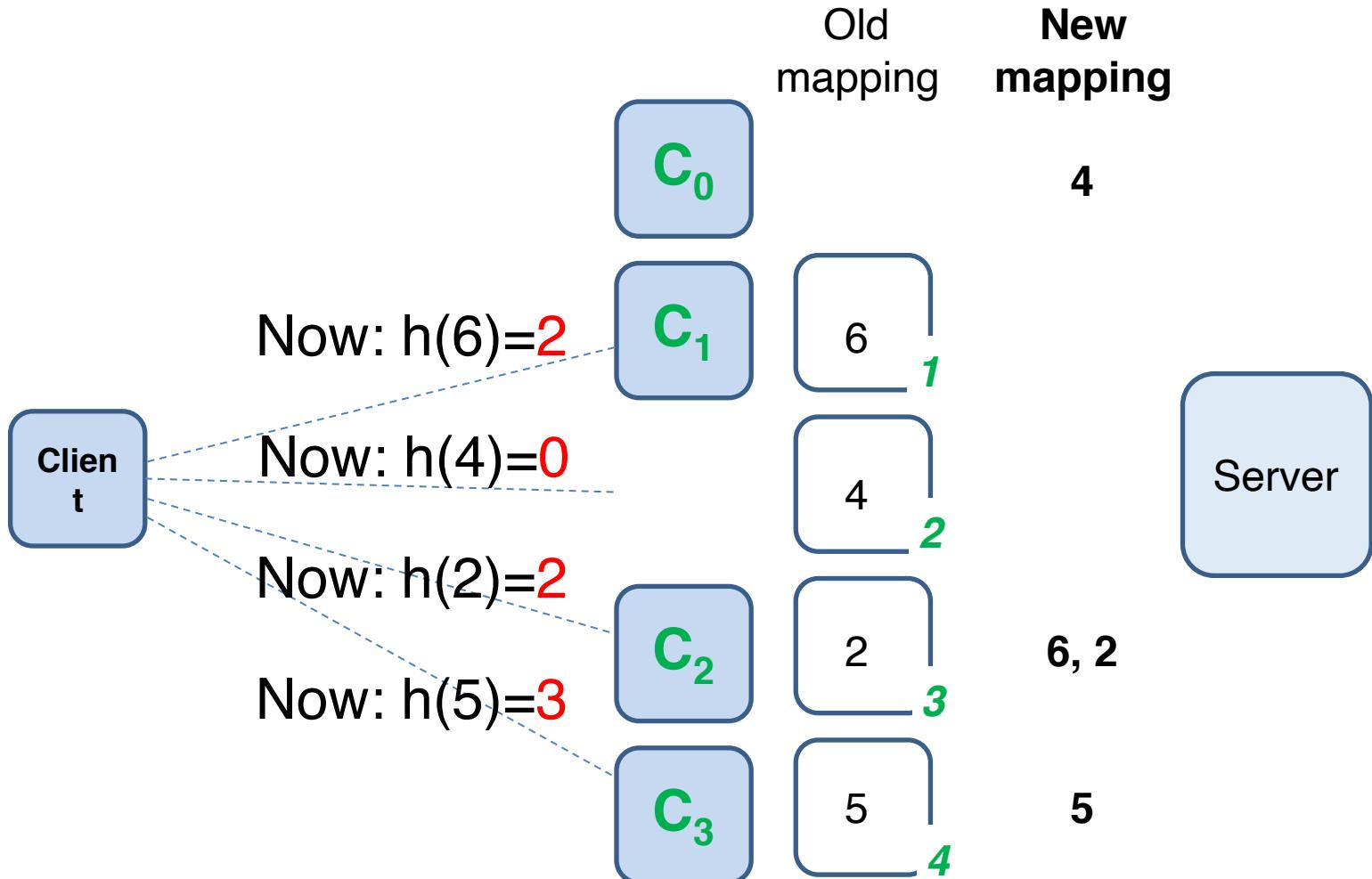
$$h(u) = (7u + 4) \bmod 5$$

Assume, we have **five caches**, numbered 0, ..., 4.



$$h(u) = (7u + 4) \bmod 4$$

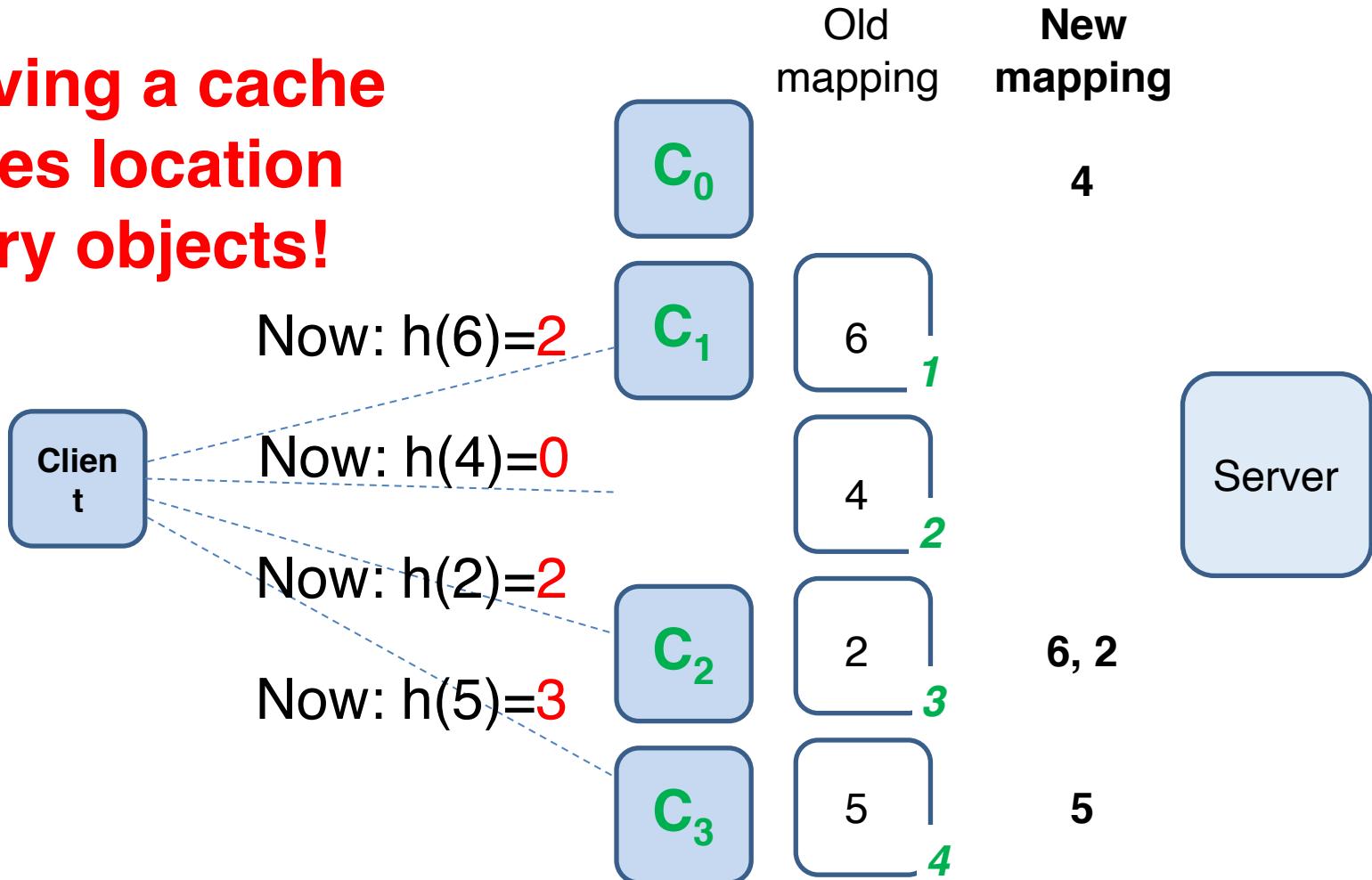
(now have to map across 4 caches)



$$h(u) = (7u + 4) \bmod 4$$

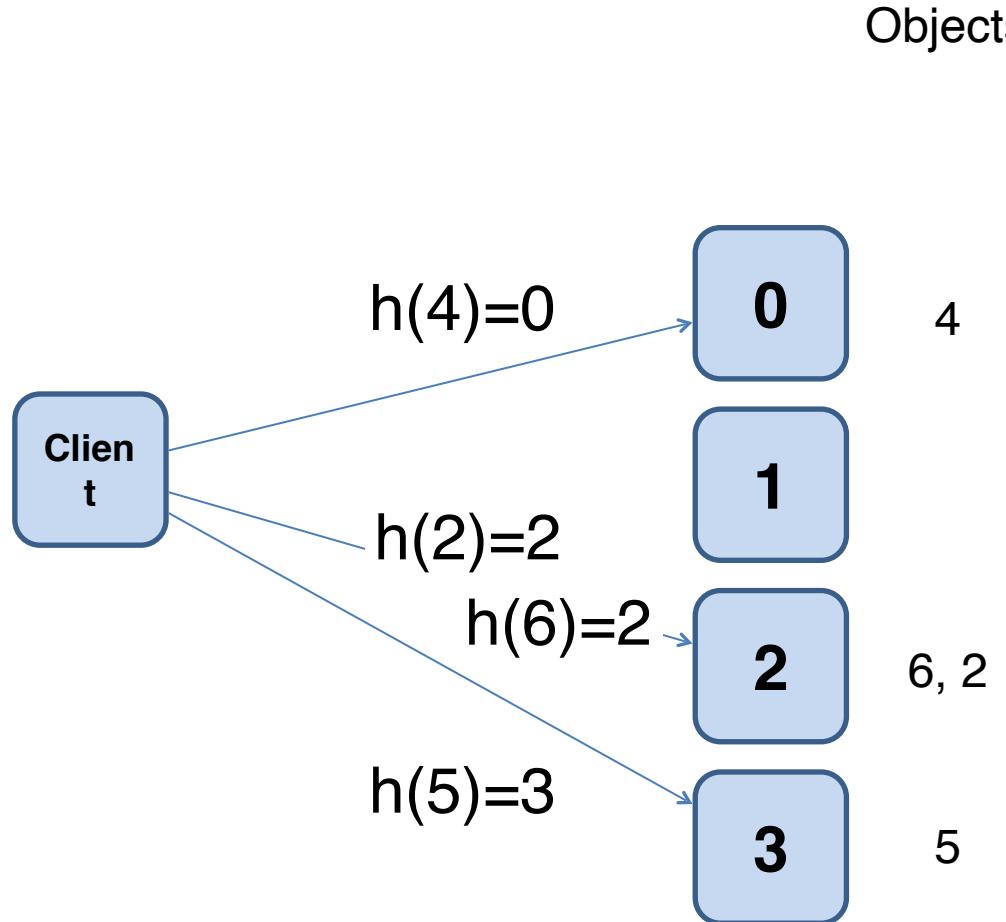
(now have to map across 4 caches)

Removing a cache changes location of every objects!



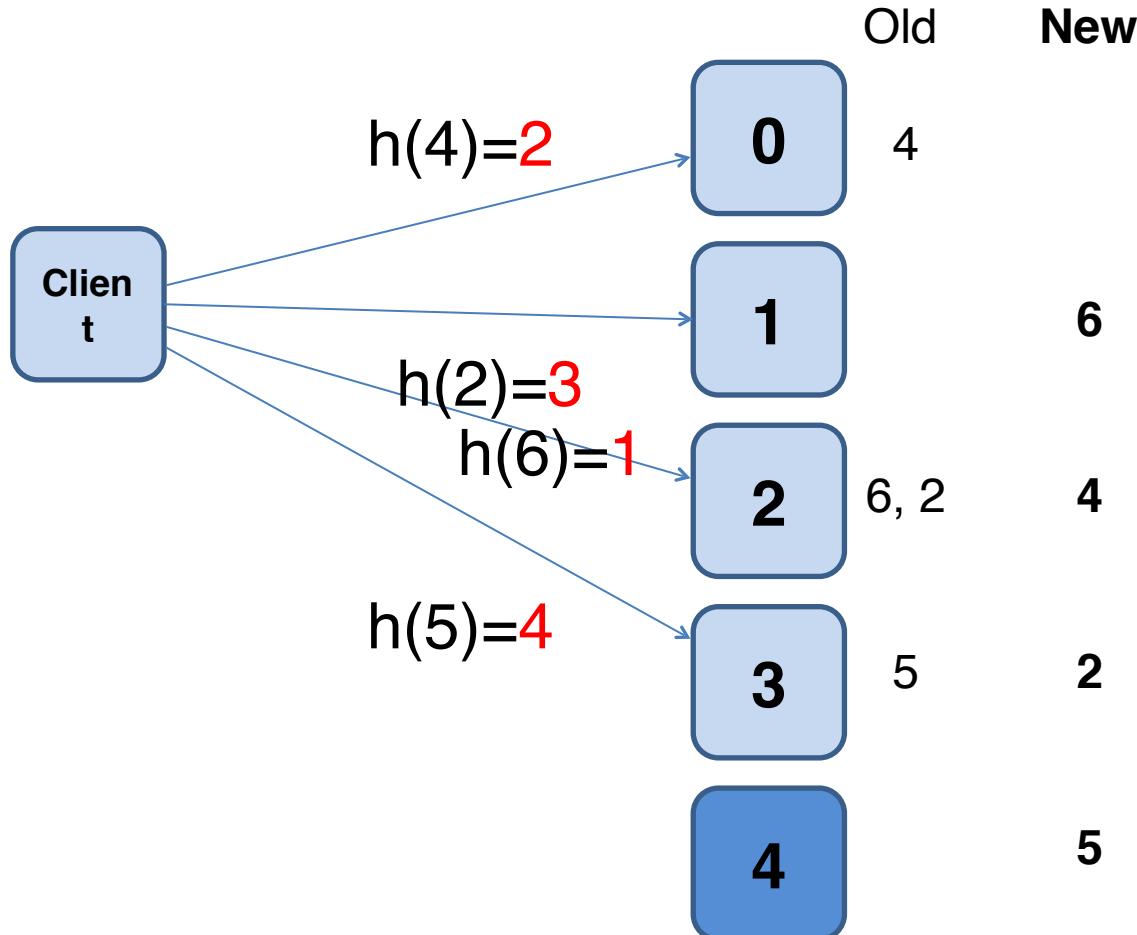
$$h(u) = 7u + 4 \bmod 4$$

(mapped across 4 nodes)



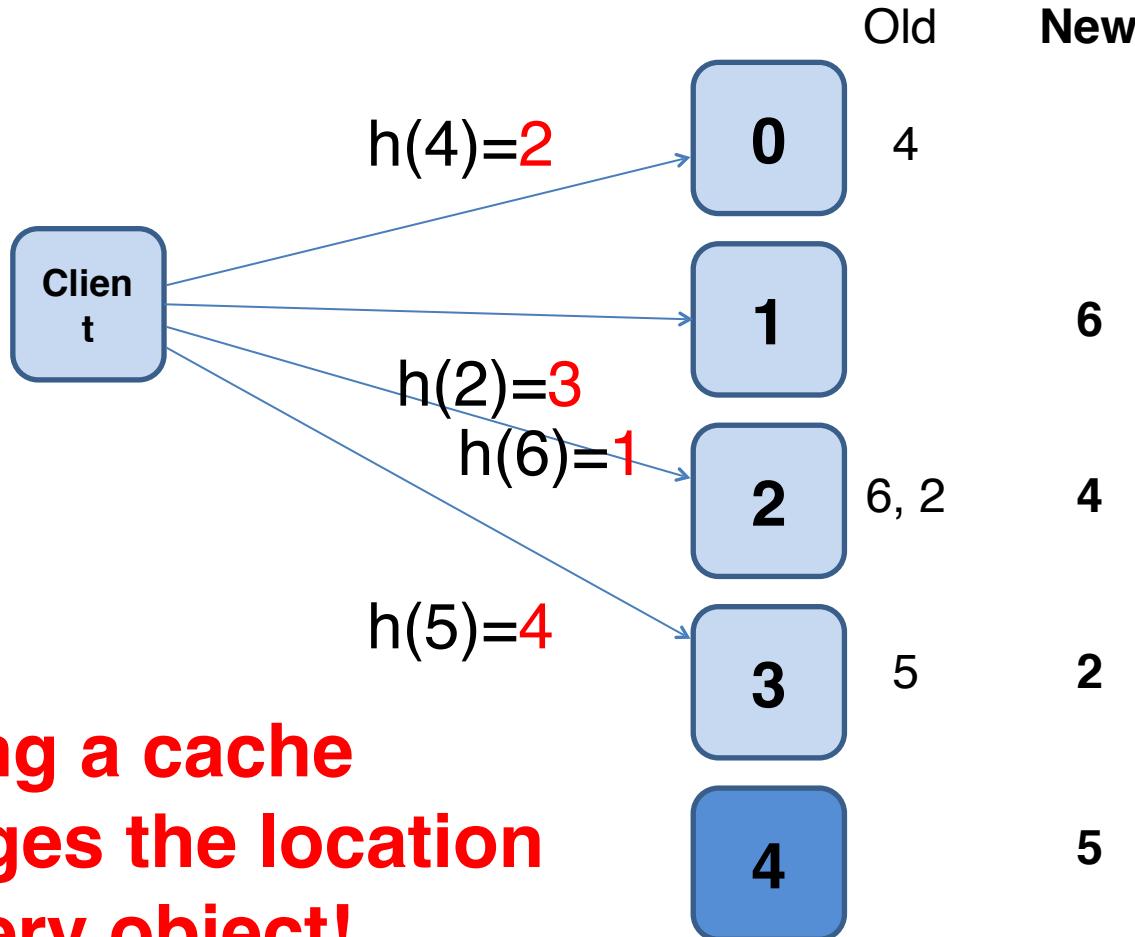
$$h(u) = (7u + 4) \bmod 5$$

(adding a cache again)



$$h(u) = (7u + 4) \bmod 5$$

(adding a cache again)



**Adding a cache
changes the location
of every object!**

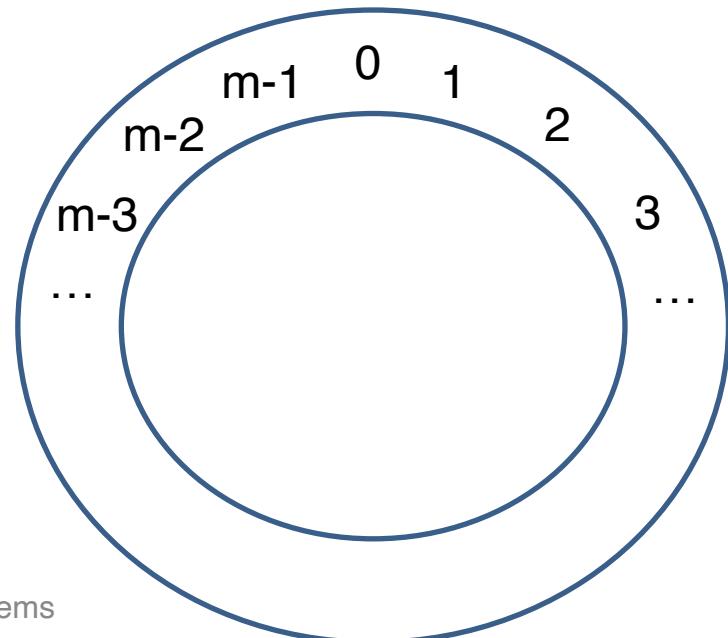
Consistent hashing

- **Goals**
 - Uniform distribution of objects across nodes
 - Easily find objects
 - Let any client perform a local computation mapping a URL to the node that contains referenced object
 - **Allow for nodes to be added/removed without much disruption**
- D. Karger *et al.*, MIT, 1997
- Basis for Akamai
 - CDN company (content distribution network)
 - Web cache as a service

Consistent hashing

Key idea intuition

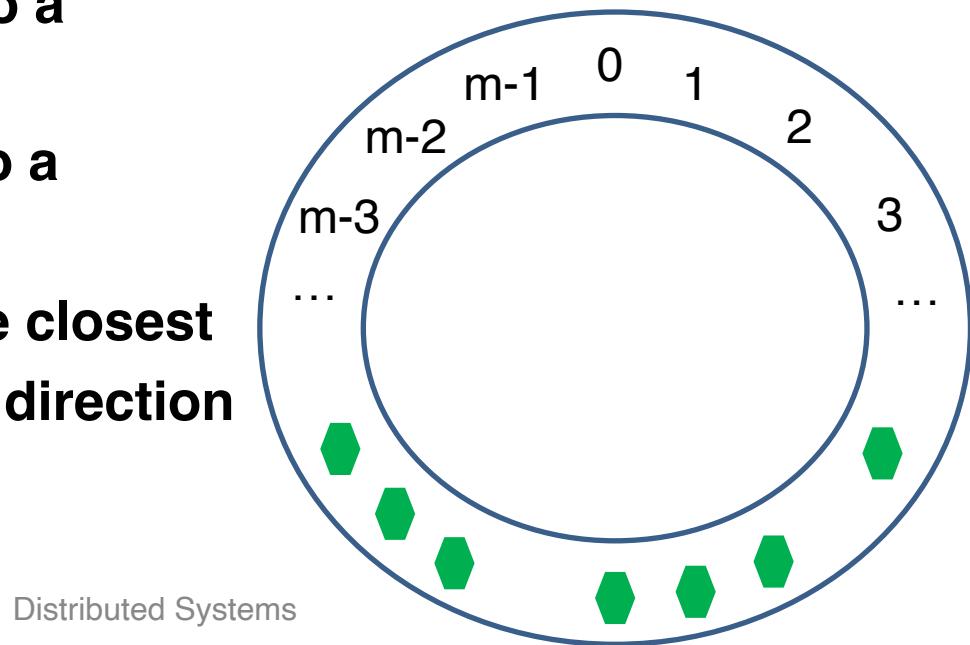
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, m-1]$
- *E.g., $h(x) = (ax + b) \bmod m$*
- Interpret range of $h(..)$ as array that wraps around (i.e., a circle)
- $h(..)$ gives slot in array (circle) and wraps around at $m-1$ to 0
- Each **object** is mapped to a **slot** via $h(..)$
- Each **cache** is mapped to a **slot** via $h(..)$
- Assign **each object to the closest cache slot in clockwise direction** on the circle



Consistent hashing

Key idea intuition

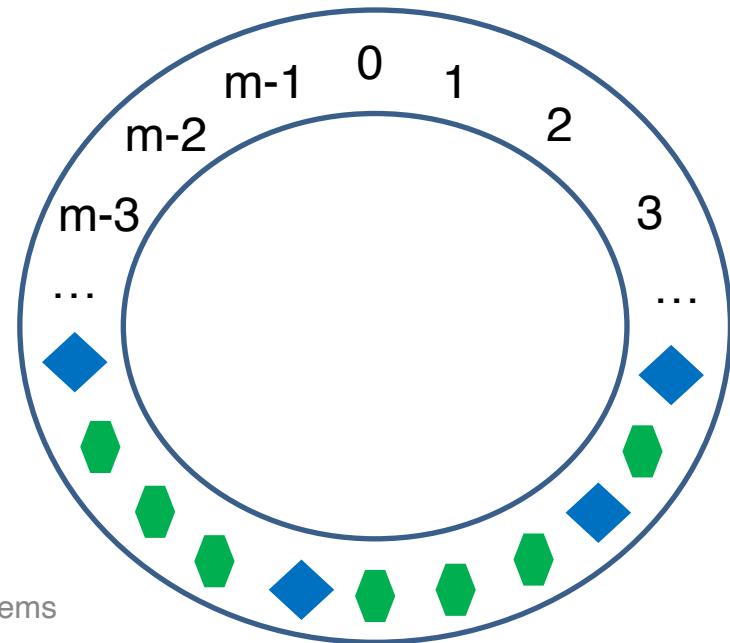
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, m-1]$
- *E.g., $h(x) = (ax + b) \bmod m$*
- Interpret range of $h(..)$ as array that wraps around (i.e., a circle)
- $h(..)$ gives slot in array (circle) and wraps around at $m-1$ to 0
- Each **object** is mapped to a **slot** via $h(..)$
- Each **cache** is mapped to a **slot** via $h(..)$
- Assign **each object to the closest cache slot in clockwise direction** on the circle



Consistent hashing

Key idea intuition

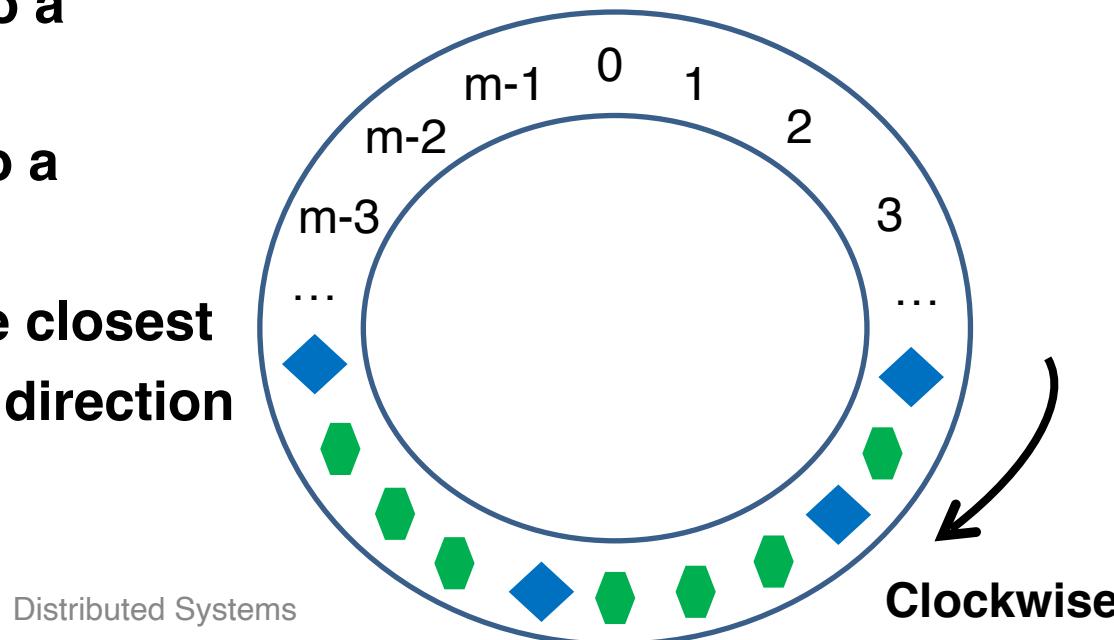
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, m-1]$
- *E.g., $h(x) = (ax + b) \bmod m$*
- Interpret range of $h(..)$ as array that wraps around (i.e., a circle)
- $h(..)$ gives slot in array (circle) and wraps around at $m-1$ to 0
- Each **object** is mapped to a **slot** via $h(..)$
- Each **cache** is mapped to a **slot** via $h(..)$
- Assign **each object to the closest cache slot in clockwise direction** on the circle



Consistent hashing

Key idea intuition

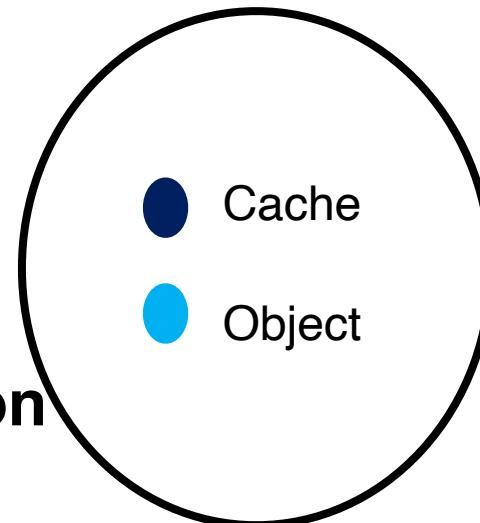
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, m-1]$
- *E.g., $h(x) = (ax + b) \bmod m$*
- Interpret range of $h(..)$ as array that wraps around (i.e., a circle)
- $h(..)$ gives slot in array (circle) and wraps around at $m-1$ to 0
- Each **object** is mapped to a **slot** via $h(..)$
- Each **cache** is mapped to a **slot** via $h(..)$
- Assign **each object to the closest cache slot in clockwise direction** on the circle



Consistent hashing

Original interpretation

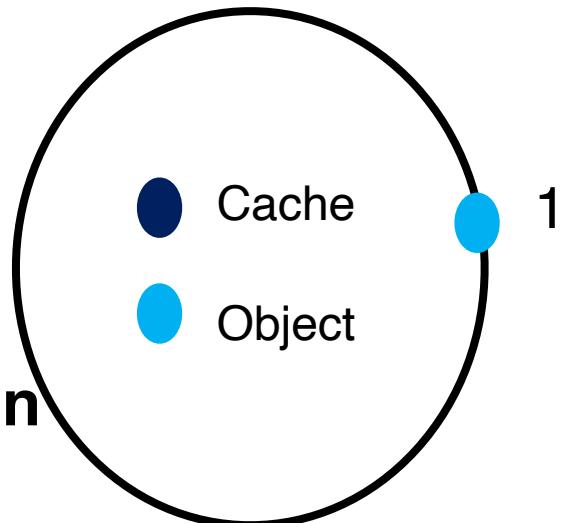
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object is mapped to a point** on the unit circle via $h(..)$
- Each **cache is mapped to a point** on the unit circle via $h(..)$
- Assign **each URL to the closest cache point in clockwise direction** on the circle



Consistent hashing

Original interpretation

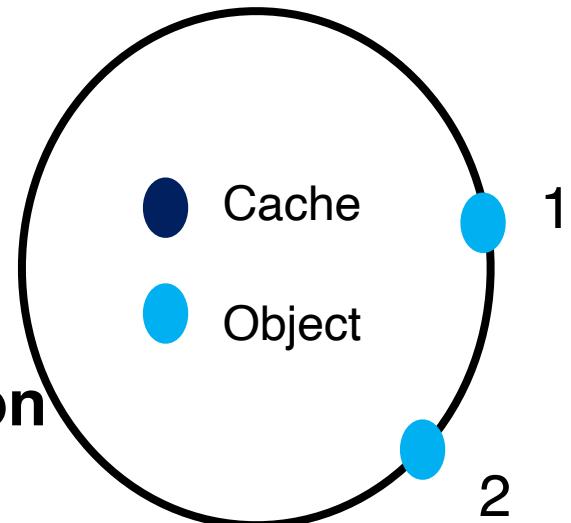
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a **point** on the unit circle via $h(..)$
- Each **cache** is mapped to a **point** on the unit circle via $h(..)$
- Assign each **URL** to the closest **cache point** in **clockwise direction** on the circle



Consistent hashing

Original interpretation

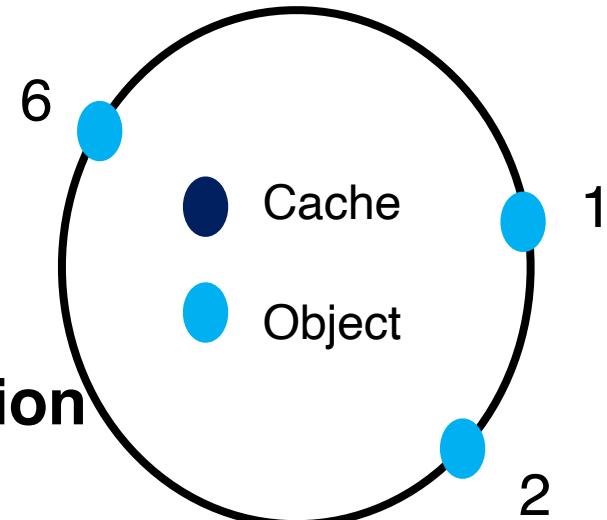
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a **point** on the unit circle via $h(..)$
- Each **cache** is mapped to a **point** on the unit circle via $h(..)$
- Assign each **URL** to the closest **cache point** in **clockwise direction** on the circle



Consistent hashing

Original interpretation

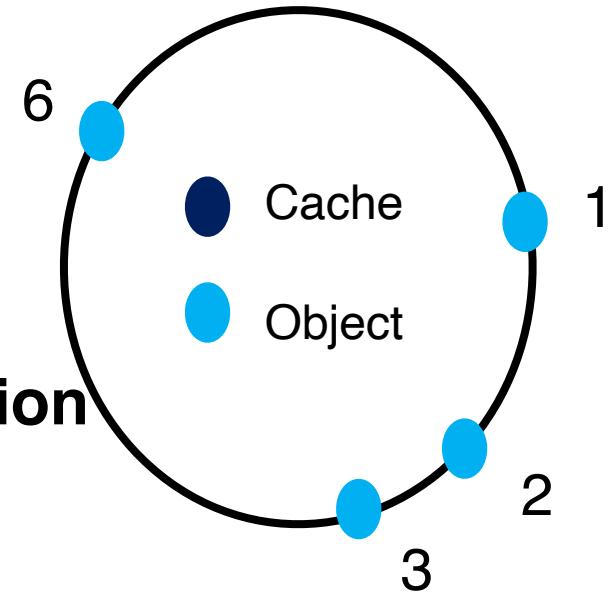
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a point on the unit circle via $h(..)$
- Each **cache** is mapped to a point on the unit circle via $h(..)$
- Assign each URL to the closest cache point in clockwise direction on the circle



Consistent hashing

Original interpretation

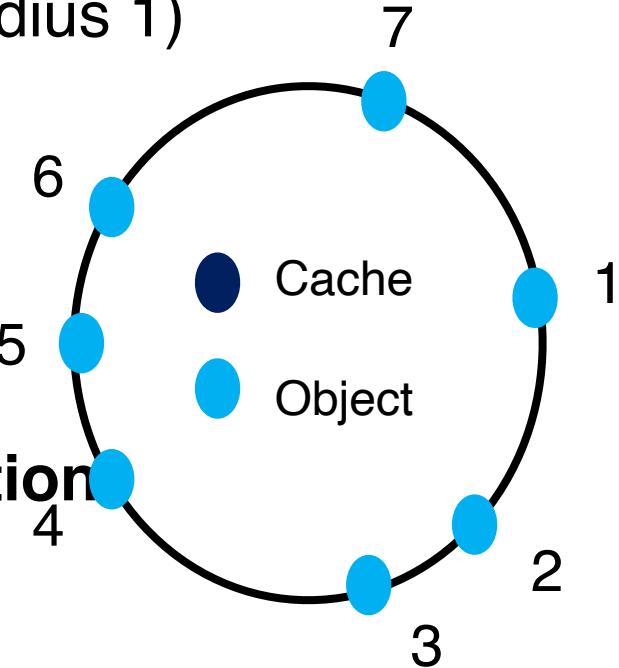
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a point on the unit circle via $h(..)$
- Each **cache** is mapped to a point on the unit circle via $h(..)$
- Assign each URL to the closest cache point in clockwise direction on the circle



Consistent hashing

Original interpretation

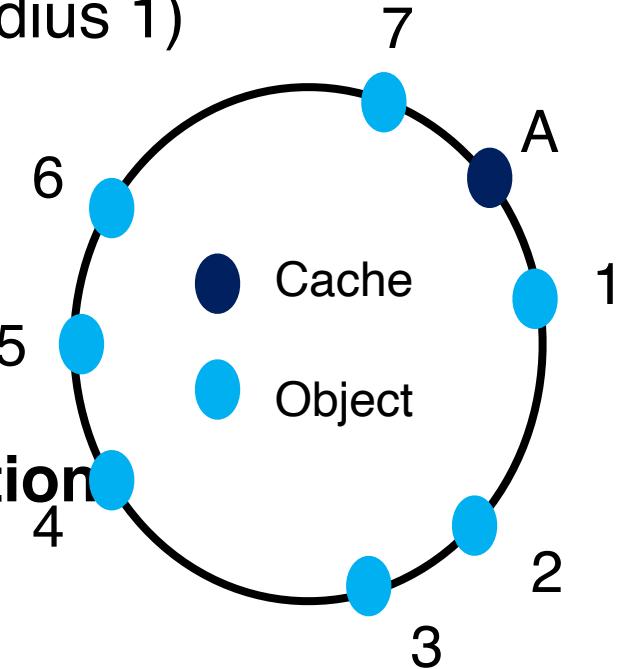
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a point on the unit circle via $h(..)$
- Each **cache** is mapped to a point on the unit circle via $h(..)$
- Assign each URL to the closest cache point in clockwise direction on the circle



Consistent hashing

Original interpretation

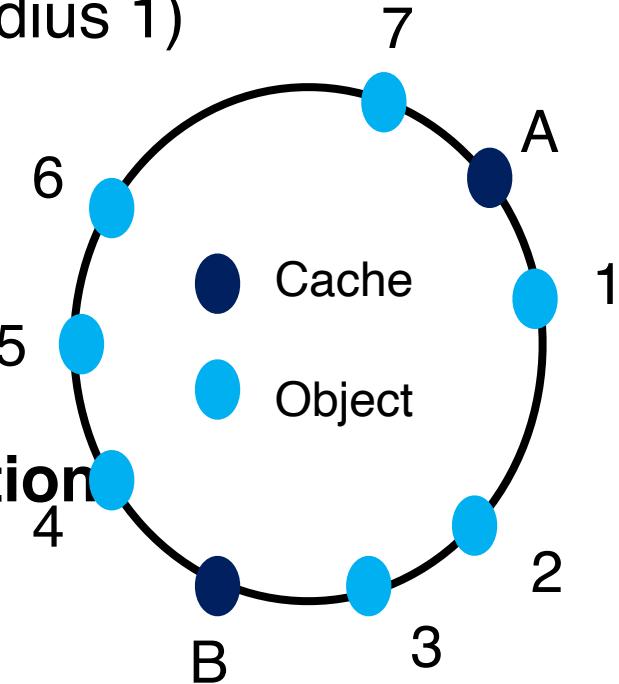
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a point on the unit circle via $h(..)$
- Each **cache** is mapped to a point on the unit circle via $h(..)$
- Assign each **URL** to the closest cache point in **clockwise direction** on the circle



Consistent hashing

Original interpretation

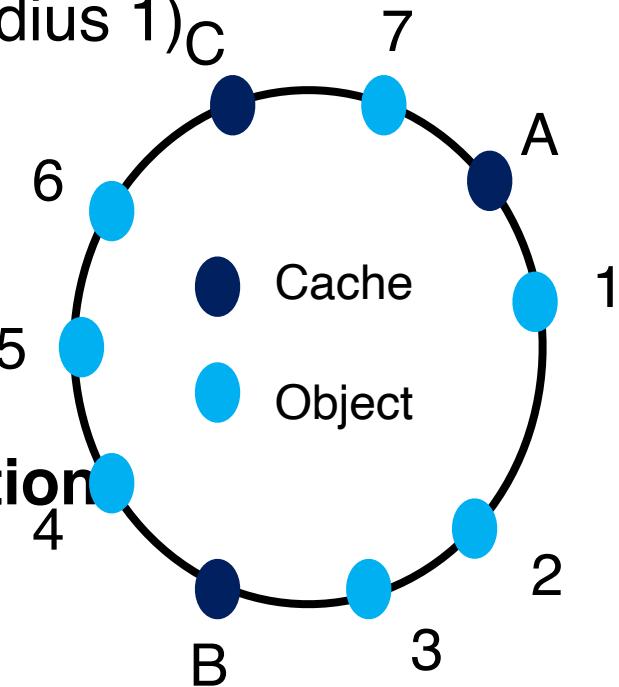
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a point on the unit circle via $h(..)$
- Each **cache** is mapped to a point on the unit circle via $h(..)$
- Assign each **URL** to the closest **cache point in clockwise direction** on the circle



Consistent hashing

Original interpretation

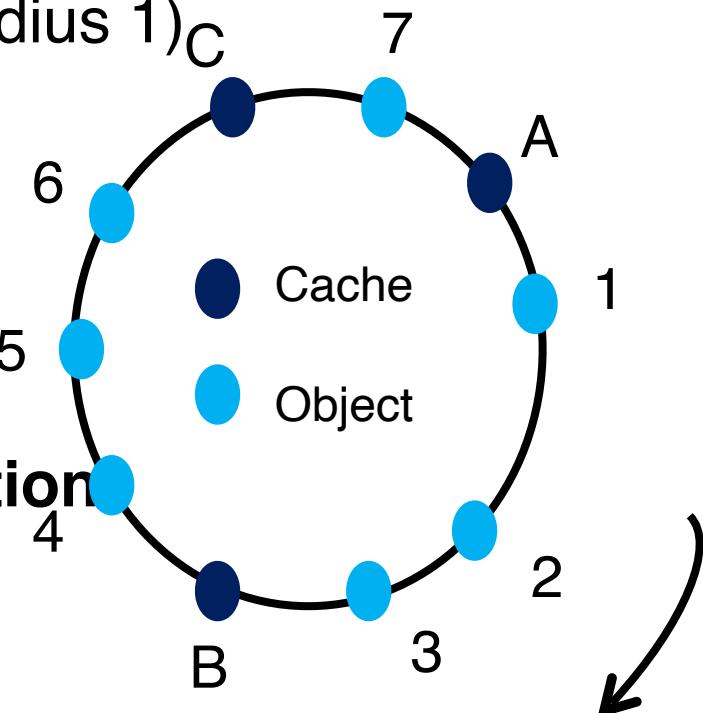
- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a point on the unit circle via $h(..)$
- Each **cache** is mapped to a point on the unit circle via $h(..)$
- Assign each **URL** to the closest **cache point in clockwise direction** on the circle



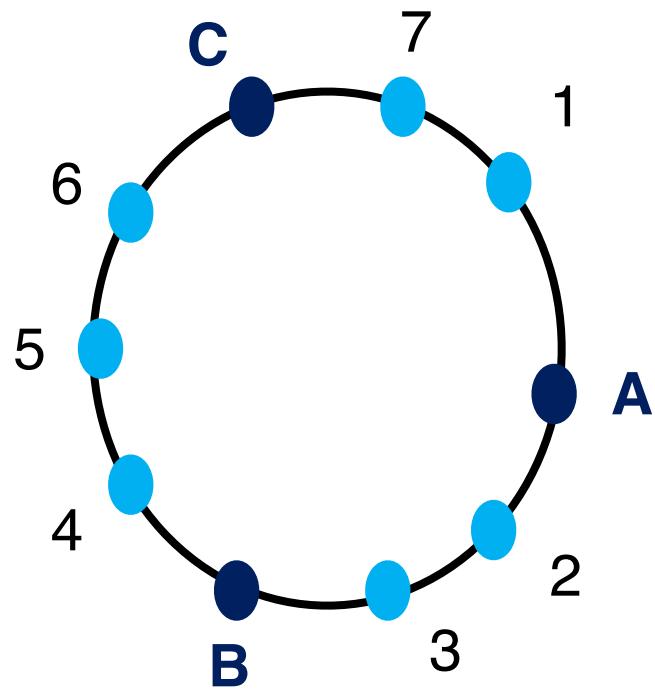
Consistent hashing

Original interpretation

- Select a **base hash function** that maps input identifier to the number range $[0, \dots, M]$
- Divide by M , re-mapping $[0, \dots, M]$ to $[0, 1]$
- Interpret this interval as the **unit circle**: Here, circle with circumference 1 (normally radius 1)
- Each **object** is mapped to a point on the unit circle via $h(..)$
- Each **cache** is mapped to a point on the unit circle via $h(..)$
- Assign each **URL** to the closest **cache point in clockwise direction** on the circle



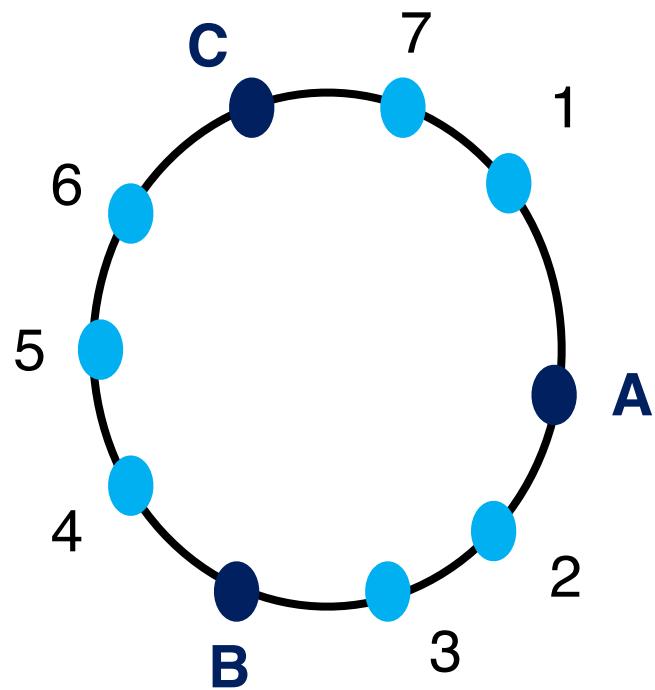
Mapping items to caches



Items 2, 3 mapped to **B**
Items 4, 5, 6 mapped to **C**
Items 7, 1 mapped to **A**

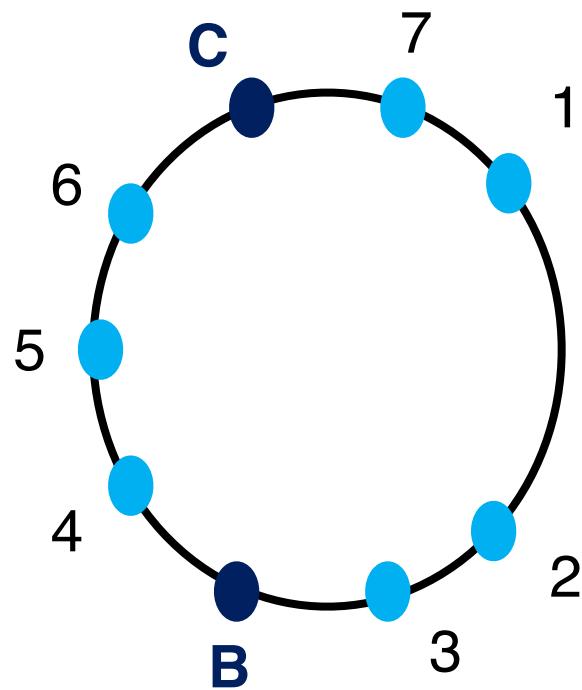
- Cache
- Object

Removing a cache



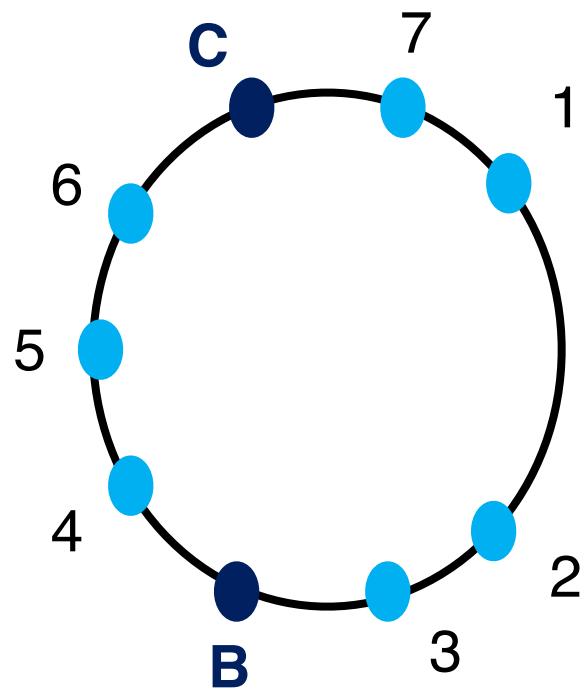
Items 2, 3	mapped to B
Items 4, 5, 6	mapped to C
Items 7, 1	mapped to A

Removing a cache



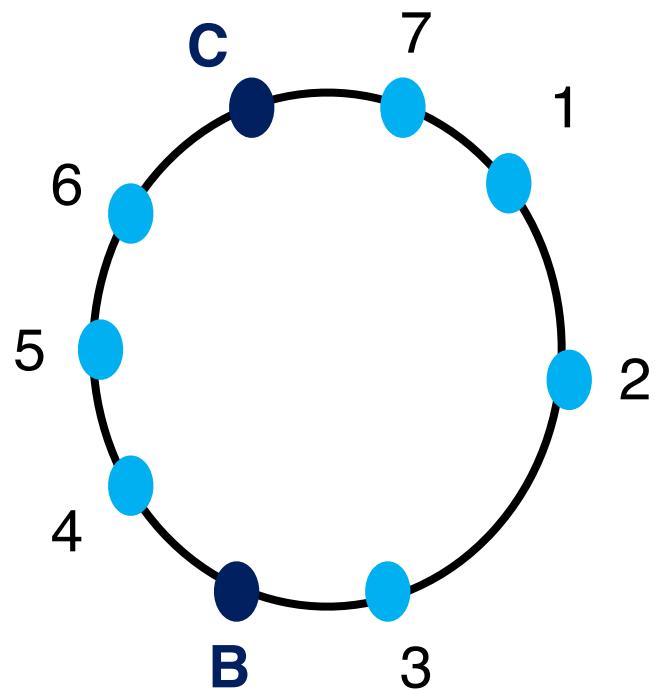
Items	2, 3	mapped to B
Items	4, 5, 6	mapped to C
Items	7, 1	mapped to A

Removing a cache



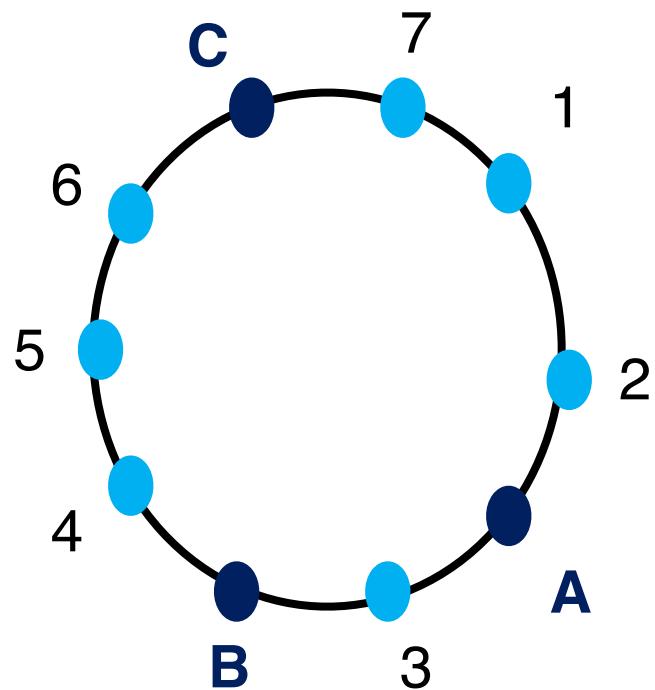
Items 2, 3 **7, 1** mapped to **B**
Items 4, 5, 6 mapped to **C**

Adding a cache



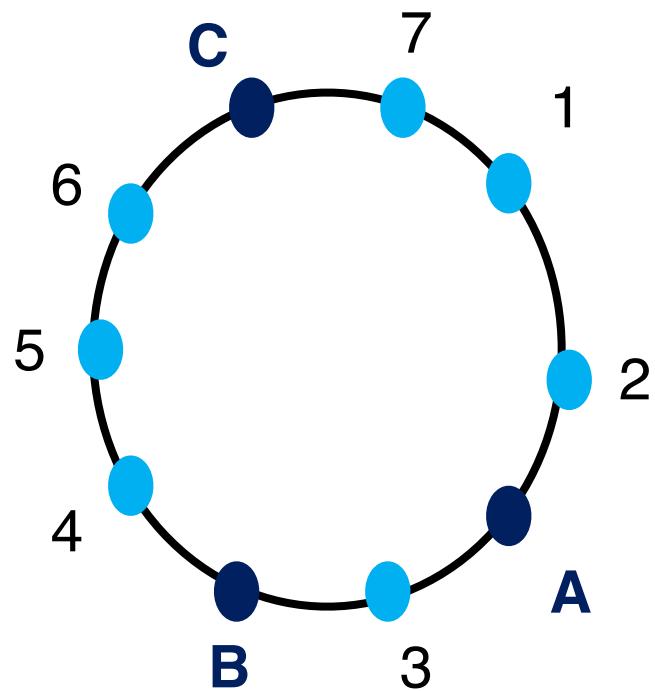
Items 7, 1, 2, 3 mapped to B
Items 4, 5, 6 mapped to C

Adding a cache



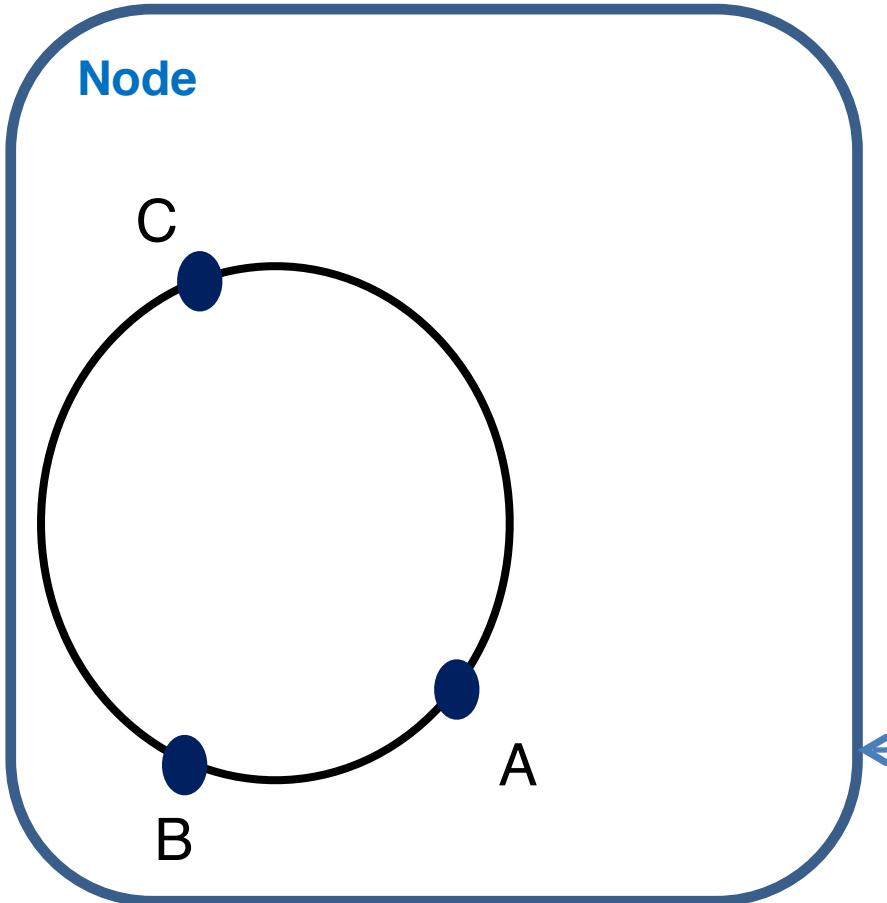
Items 7, 1, 2, 3 mapped to B
Items 4, 5, 6 mapped to C

Adding a cache



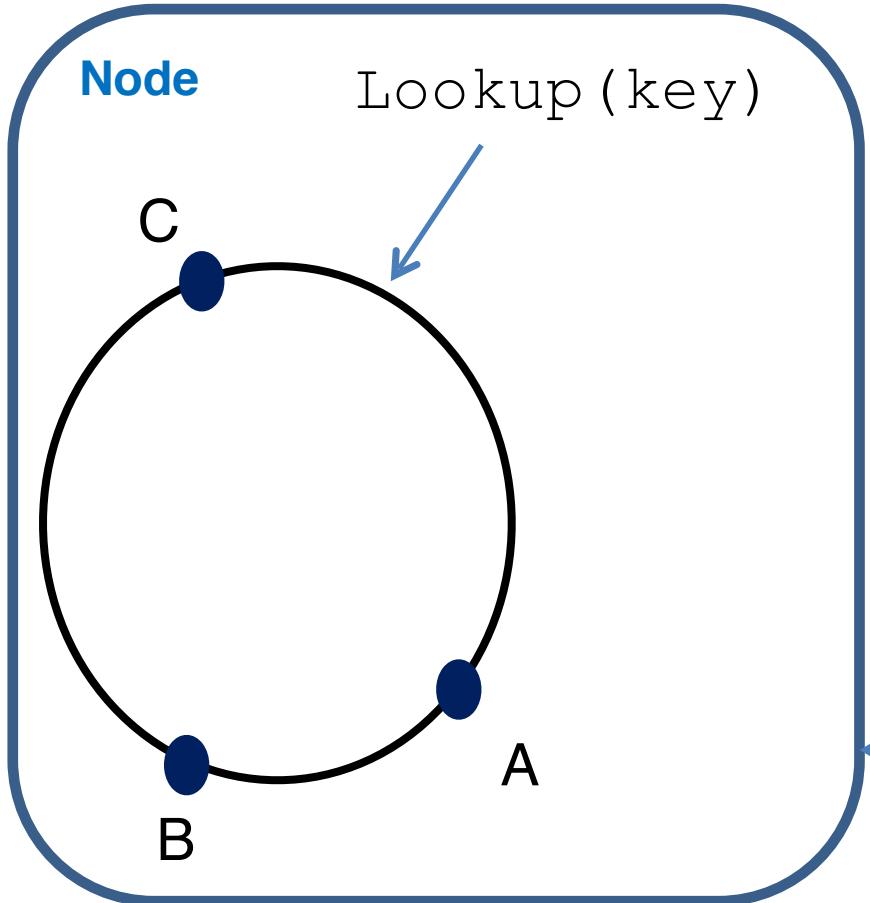
Items 3	mapped to B
Items 4, 5, 6	mapped to C
Items 7, 1, 2	mapped to A

Processing a Lookup (key)



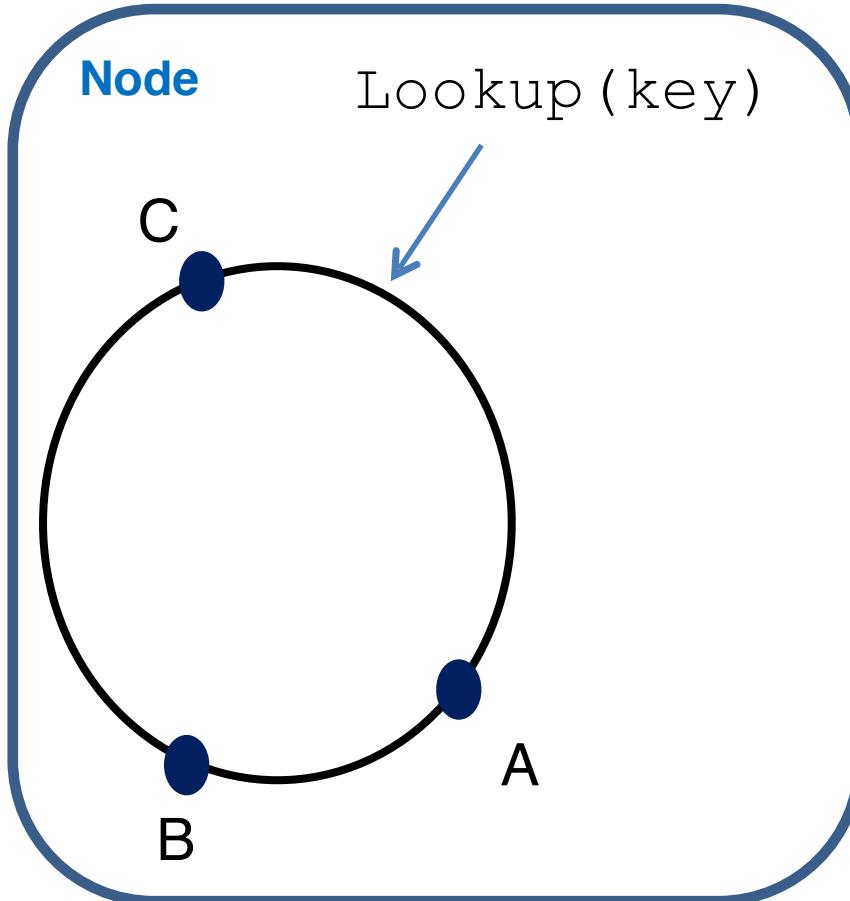
Information about
node addition & removal
(e.g., via gossiping or via a
coordination service)

Processing a Lookup (key)



Information about
node addition & removal
(e.g., via gossiping or via a
coordination service)

Processing a Lookup (key)



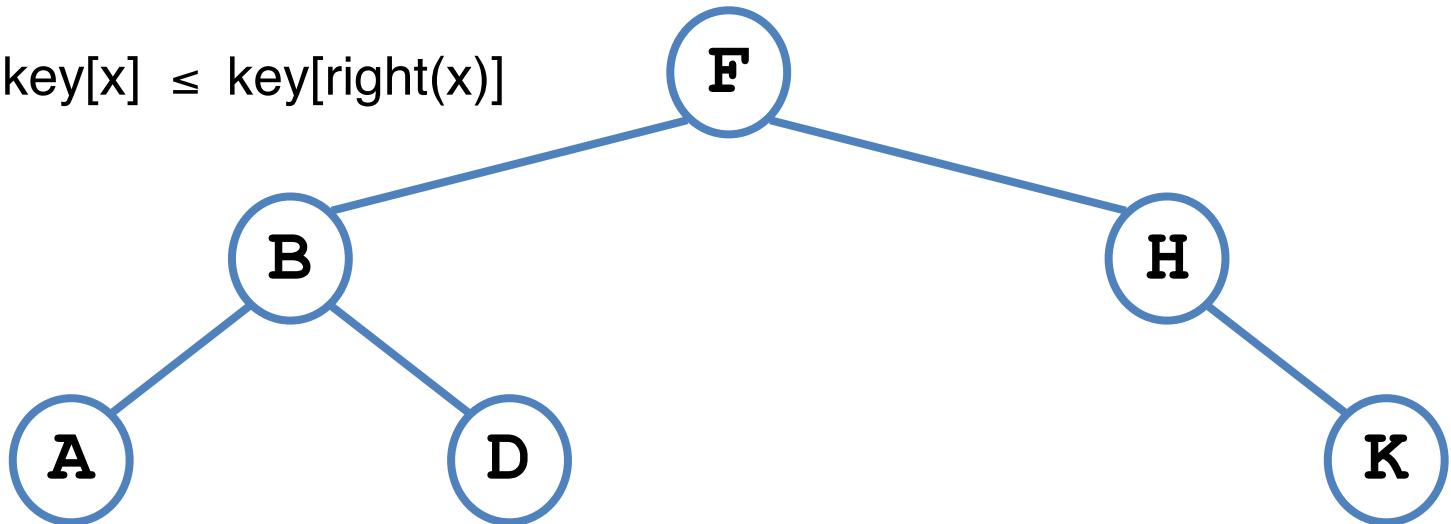
Retrieve **object** with
key from **A**

Information about
node addition & removal
(e.g., via gossiping or via a
coordination service)

Cache lookup data structure at each node

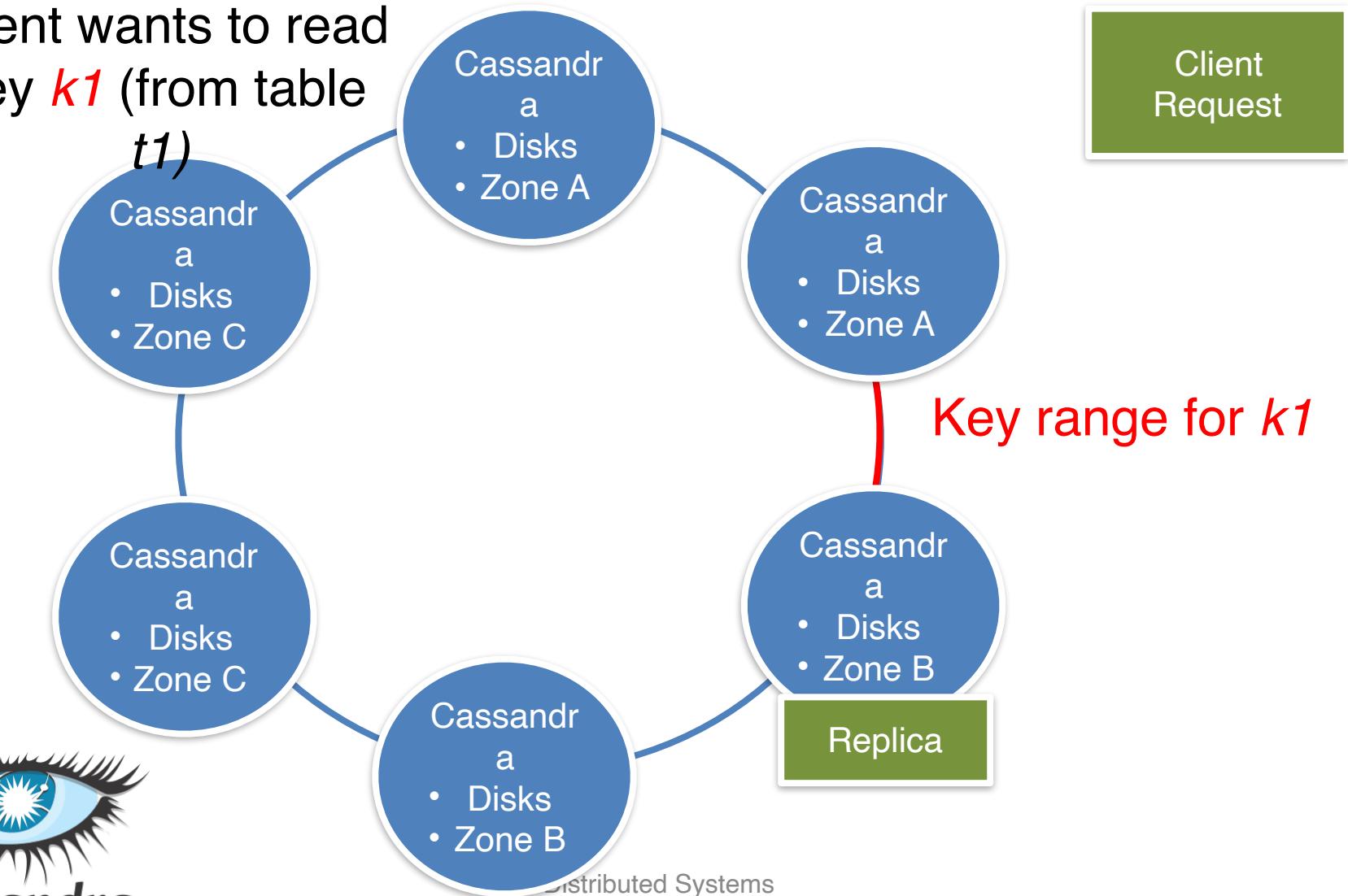
- Store **cache points** in a **binary tree**
- Find **clockwise successor** of a **URL point** by single search in tree (takes **$O(\log n)$ time**)
- For a constant time technique, cf. Karger et al., 1997

$\text{key}[\text{left}(x)] \leq \text{key}[x] \leq \text{key}[\text{right}(x)]$



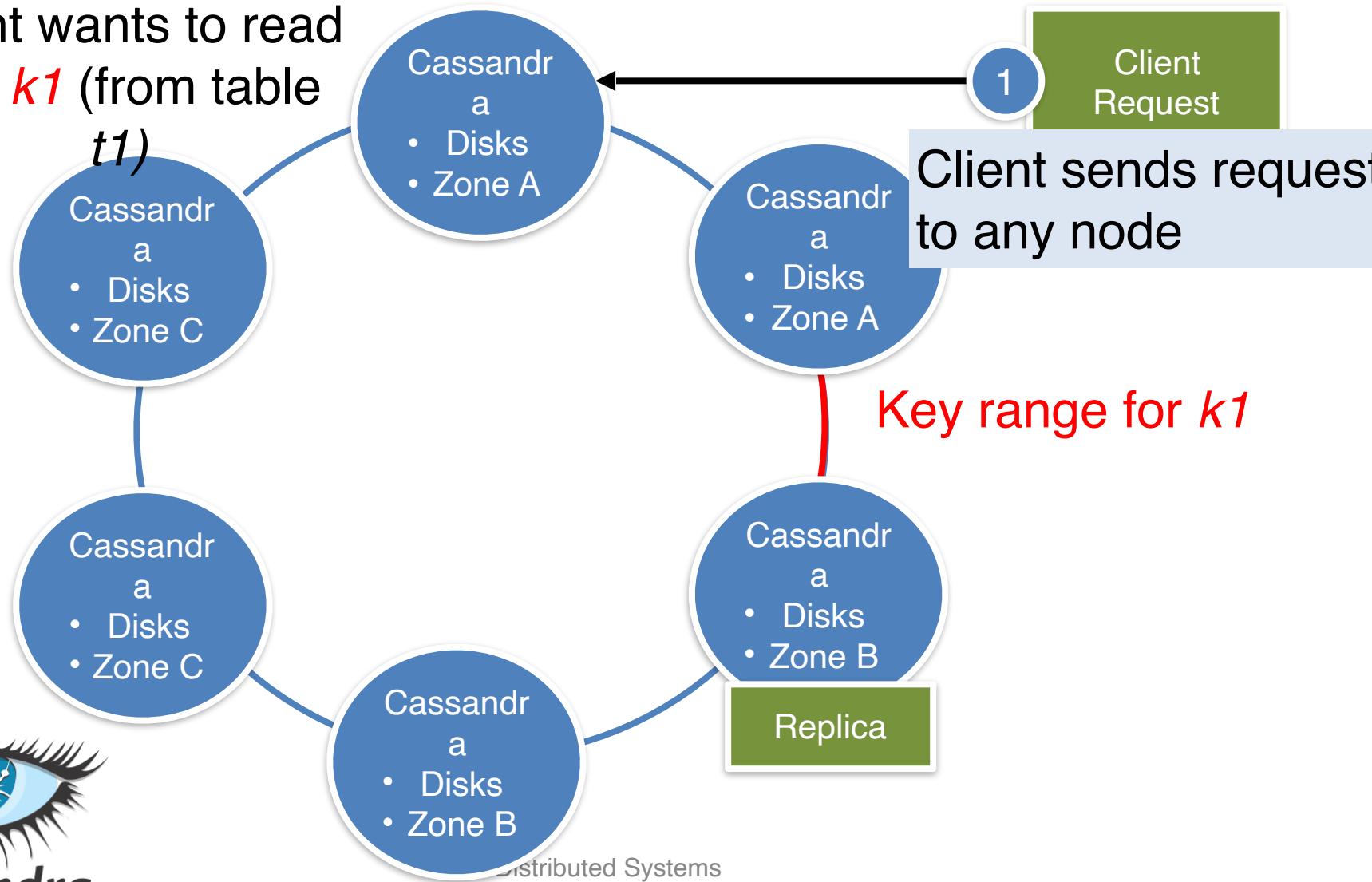
Cassandra global read-path

Client wants to read
key *k1* (from table
t1)



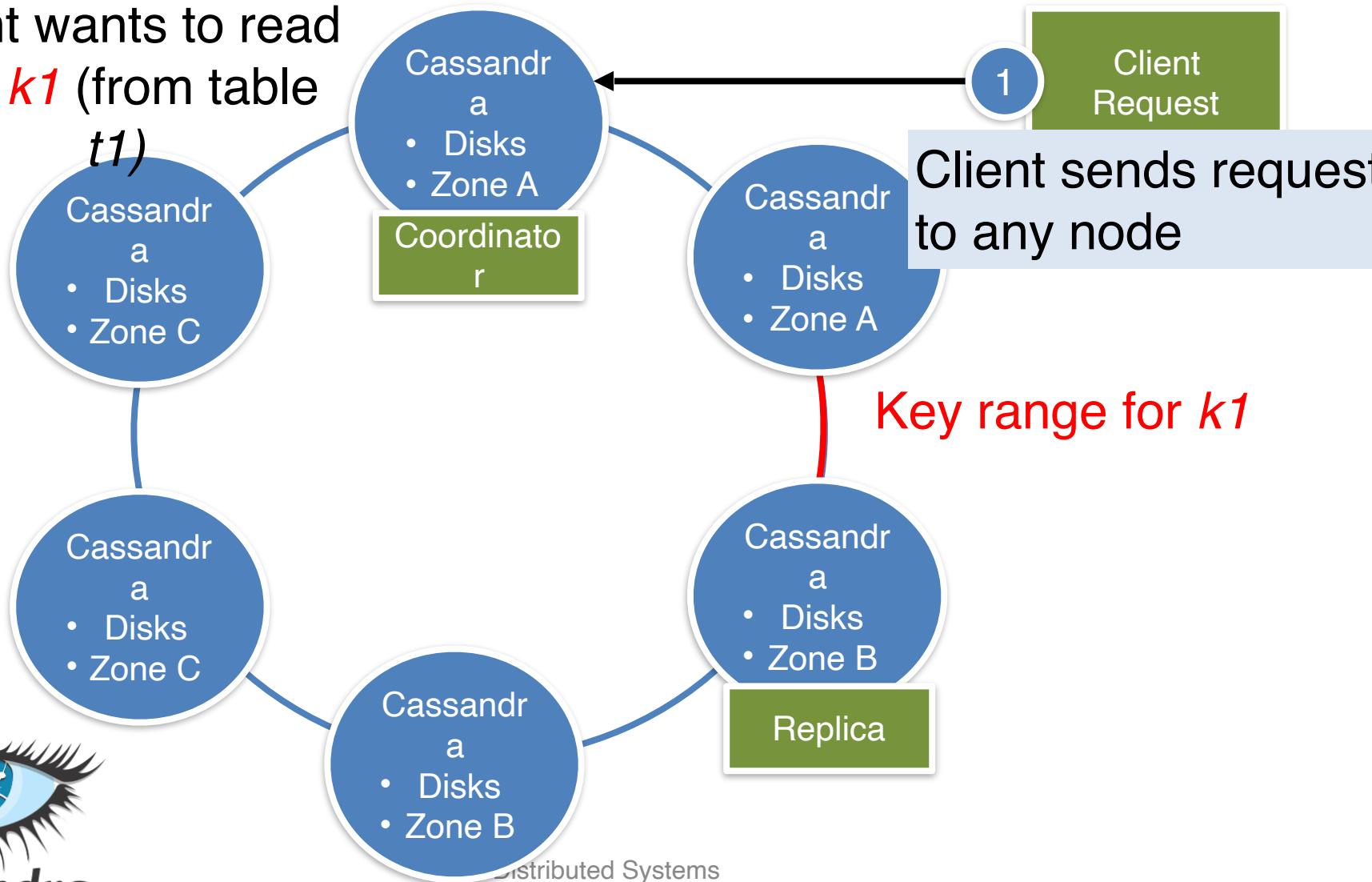
Cassandra global read-path

Client wants to read key $k1$ (from table $t1$)



Cassandra global read-path

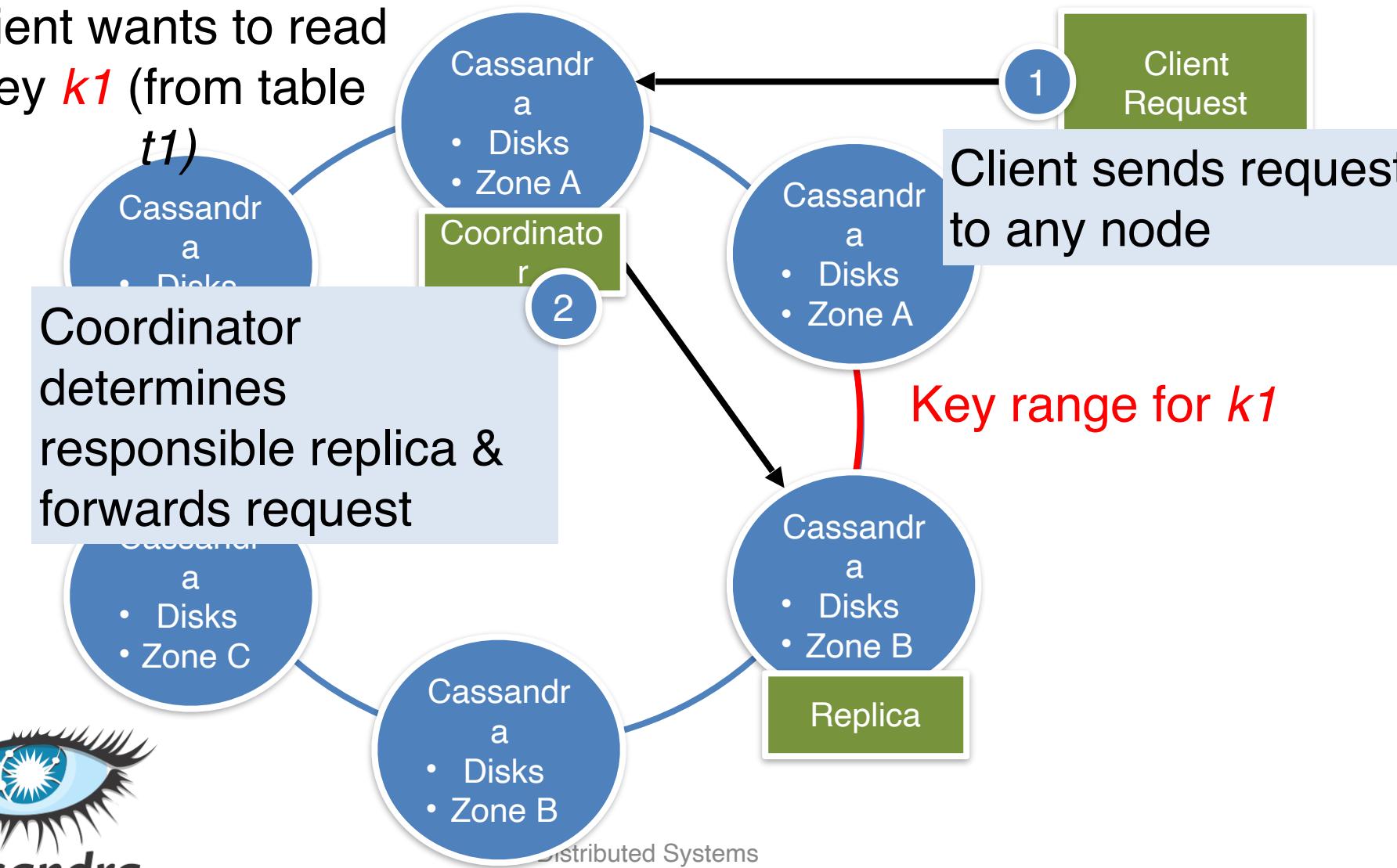
Client wants to read key *k1* (from table *t1*)



Cassandra global read-path

Client wants to read key *k1* (from table *t1*)

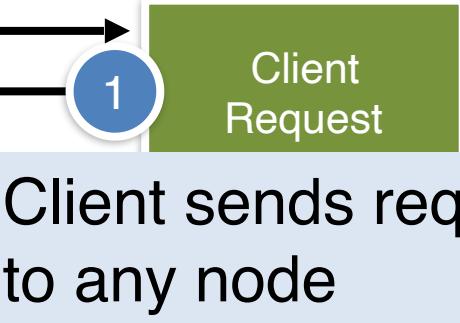
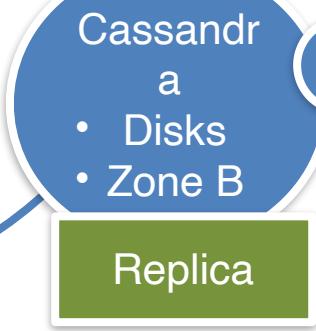
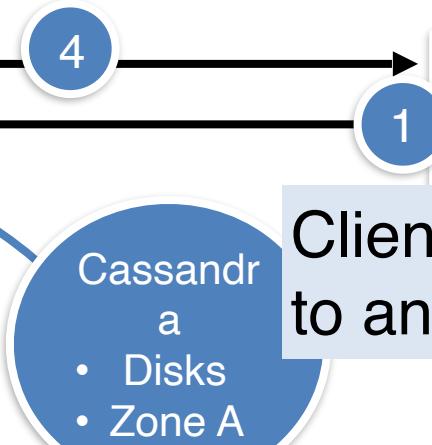
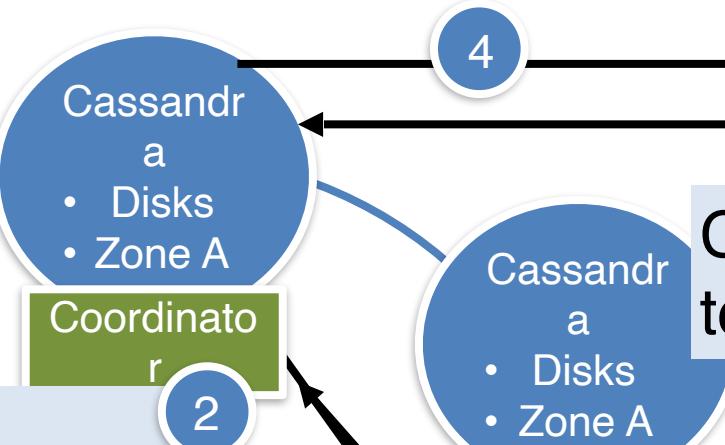
Coordinator determines responsible replica & forwards request



Cassandra global read-path

Client wants to read key *k1* (from table *t1*)

Coordinator determines responsible replica & forwards request



Client sends request to any node

Key range for *k1*

Replica queries local file system, Returns value



Base hash function: MD5

- **Message Digest 5 (MD5)**, R. Rivest, 1992 (MD1, ..., MD6)
- **Hash function** that produces a **128-bit (16-byte) hash value**
- Maps variable-length message into a **fixed-length output**
- MD5 hash is typically expressed as a hex number (32 digits)
- It's been shown that **MD5 is not collision resistant**
- US-CERT about MD5 “***should be considered cryptographically broken and unsuitable for further use***” (for security, not for caching)
- SHA-2 is a more appropriate cryptographic hash function
- For consistent hashing, MD5 is sufficient

MD5 examples

- MD5(*“The quick brown fox jumps over the lazy dog”*) =
9e107d9d372bb6826bd81d3542a419d6
- MD5(*“The quick brown fox jumps over the lazy dog.”*) =
e4d909c290d0fb1ca068ffaddf22cbd0
- MD5("") = d41d8cd98f00b204e9800998ecf8427e

<http://en.wikipedia.org/wiki/MD5#Algorithm>

MD5 examples

- MD5(*“The quick brown fox jumps over the lazy dog”*) =
9e107d9d372bb6826bd81d3542a419d6
- MD5(*“The quick brown fox jumps over the lazy dog.”*) =
e4d909c290d0fb1ca068ffaddf22cbd0
- MD5("") = d41d8cd98f00b204e9800998ecf8427e

<http://en.wikipedia.org/wiki/MD5#Algorithm>

Distributed File Systems

Application & Middleware Systems
Research Group

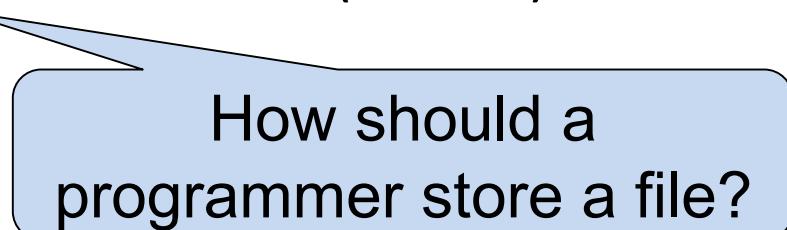
This week

- Basics of FS: POSIX, physical, and EXT2
- User-oriented FS: Network file systems (NFS)
- Big Data Distributed FS: GFS / HDFS / Ceph
- Break
- Erasure coding

BASICS OF FILE SYSTEMS

Interaction with file systems

- Motivation: Provide abstractions a programmer can use to achieve platform independence
- POSIX
 - POSIX, “The Single UNIX Specification”
 - Aligns with the ISO C 1999 standard (stdio.h)
 - File locking
 - Directories



How should a
programmer store a file?

Basic concepts POSIX

- Files
- Directories
- Links
- Metadata
- Locks

```
chris@xr2d2 ~ % tree -L 1
.
├── bin
├── boot
├── cdrom
├── core
├── dev
├── etc
├── home
├── initrd.img -> boot/initrd.img-4.2.0-19-generic
├── initrd.img.old -> boot/initrd.img-4.2.0-18-generic
├── lib
├── lib64
├── lost+found
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
└── tmp
└── vmlinuz -> boot/vmlinuz-4.2.0-19-generic
└── vmlinuz.old -> boot/vmlinuz-4.2.0-18-generic

21 directories, 5 files
```

File System Operations

- File operations:
 - Open
 - Read
 - Write
 - Close
- Directory operations:
 - Create file
 - Mkdir
 - Rename file
 - Rename dir
 - Delete file
 - Delete dir

POSIX Files <stdio.h>

```
FILE *fopen(const char * filename, const char * mode)
```

Modes

- r open text file for reading
- w truncate to zero length or create text file for writing
- a append; open or create text file for writing at end-of-file
- rb open binary file for reading
- wb truncate to zero length or create binary file for writing
- ab append; open or create binary file for writing at end-of-file
- r+ open text file for update (reading and writing)
- w+ truncate to zero length or create text file for update
- a+ append; open or create text file for update, writing at end-of-file

- r+b or rb+ open binary file for update (reading and writing)
- w+b or wb+ truncate to zero length or create binary file for update
- a+b or ab+ append; open or create binary file for update, writing at end-of-file

```
int fflush(FILE *stream); //Any unwritten buffered data is physically persisted  
int fclose(FILE *stream); //Flushed and file closed
```

POSIX Directories <stat.h>

```
int mkdir(const char* path, mode_t mode)

/* example
mkdir("/home/cd/distributed_systems", S_IRUSR | S_IWUSR | S_IXUSR | S_IRWXG );

...
S_IRUSR  read permission, owner
S_IWUSR  write permission, owner
S_IXUSR  execute/search permission owner
S_IRWXG  read, write, execute/search by group
...
*/
```

POSIX File Locking <fcntl.h>

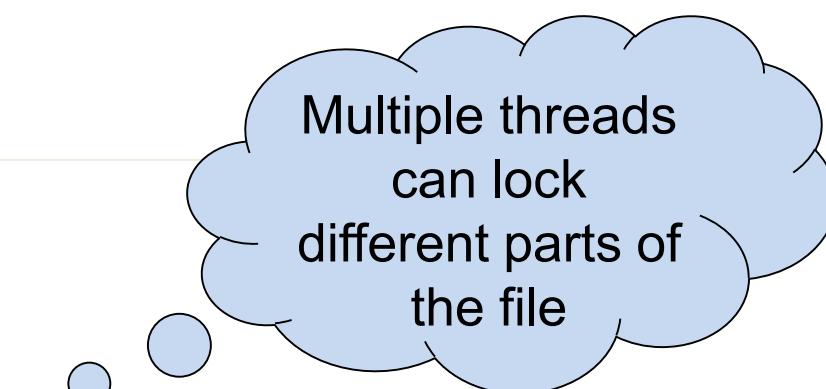
```
int fcntl(int fildes, int cmd, ...);

/* small example
int fd;
struct flock fl;

fd = open("/home/cd/test.txt");

fl.l_type = F_WRLCK;      //write lock
fl.l_whence = SEEK_SET
fl.l_start = 500;          //start at byte 500
fl.l_len = 100;            //next 100 bytes

fcntl(fd, F_SETLK, &fl); //acquire lock
/*
```



Multiple threads
can lock
different parts of
the file

Types of locks:

F_RDLCK Shared or read lock

F_UNLCK Unlock

F_WRLCK Exclusive or write lock

POSIX File Metadata <stat.h>, <unistd.h>

```
//access permissions
int chmod(const char * file, mode_t mode)

/* Example modes
777 read, write, execute for all
664 sets read and write and no execution access for owner and
group, and read, no write, no execute for all others
*/
//e.g. chmod("file.txt", S_IRUSR | S_IWUSR)
```

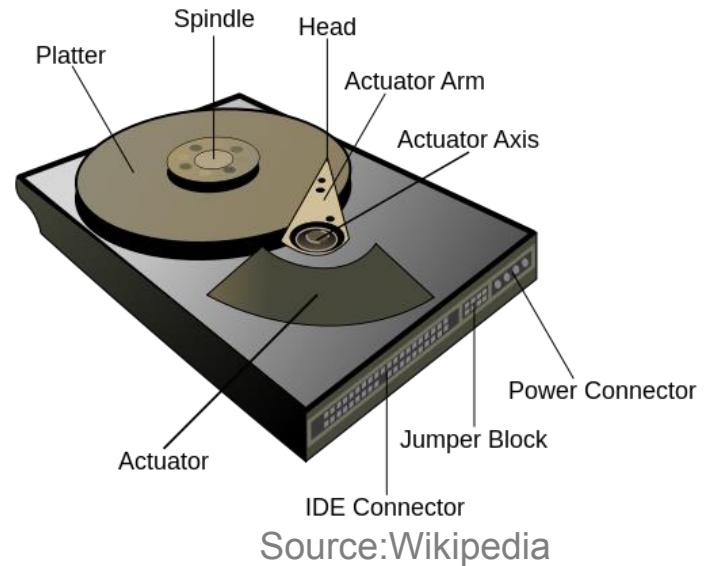
```
//change ownership of a file
int chown(const char *, uid_t, gid_t)
```

```
//e.g. chown("file.txt", getpwnam("christoph"), -1)
```

```
chris@xr2d2 ~ % ls -l
total 10084
drwxr-xr-x  2 root root   12288 Okt 23 08:50 bin
drwxr-xr-x  3 root root    4096 Dez  2 08:53 boot
drwxrwxr-x  2 root root    4096 Jun 19 12:06 cdrom
-rw-----  1 root root 10203136 Aug 19 16:53 core
```

HDD

- Magnetic discs
- Cache (8MB – 128MB)
- Cost
 - ~38€/TB (1.1.2014)
 - ~30€/TB (4.12.2014)
 - ~29€/TB (7.12.2015)
 - ~22€/TB (29.11.2017)
 - ~21€/TB (9.12.2019)
- 5400 rpm – 15000 rpm
- Seek 4-9ms
- Connected via SATA, SCSI/SAS ...



SSD

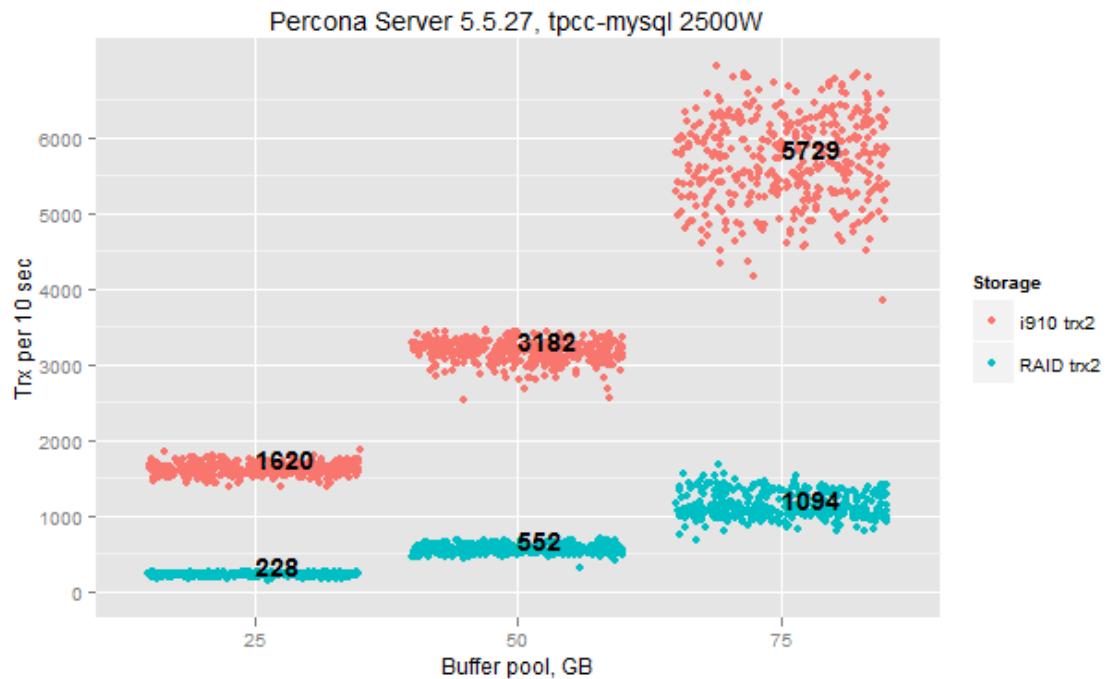
- DRAM (NAND-based flash memory)
- No moving mechanical components
- Cache (16MB – 512MB)
- Cost
 - ~600€/TB (1.1.2014)
 - ~350€/TB (4.12.2014)
 - ~260€/TB (7.12.2015)
 - ~250€/TB (29.11.2017)
 - ~90€/TB (9.12.2019)
- Can also be connected via PCI Express
- Low-level operations differ compared to HDD
 - On SSD's overwriting costs more => TRIM Command
 - Deleting is delegated to internal firmware which has a garbage collector



Source:OCZ

Benchmarks

- [1] “Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King”, Jim Gray, Microsoft



RAID10 over 8 HDD
vs. software RAID 2xSSD

[1] http://research.microsoft.com/en-us/um/people/gray/talks/flash_is_good.ppt

Source: <http://www.mysqlperformanceblog.com/2012/09/11/intel-ssd-910-vs-hdd-raid-in-tpcc-mysql-benchmark/>

How common are actually HDD failures?

Backblaze Lifetime Hard Drive Annualized Failure Rates

For hard drive models in service as of September 30, 2019

Reporting period April 2013 - September 2019 inclusive

AFR: $\frac{\text{Drive Failures}}{\text{Drive Days}}$ / 365.25

MFG	Model	Drive Size	Drive Count	Avg. Age	Drive Days	Drive Failures	AFR*
HGST	HMS5C4040ALE640	4TB	2,707	42.0	11,420,392	161	0.51%
HGST	HMS5C4040BLE640	4TB	12,641	35.6	18,409,871	233	0.46%
HGST	HUH728080ALE600	8TB	1,001	22.3	746,311	16	0.78%
HGST	HUH721212ALE600	12TB	1,560	4.8	183,560	4	0.80%
HGST	HUH721212ALN604	12TB	10,849	6.1	1,923,518	25	0.47%
Seagate	ST4000DM000	4TB	19,330	47.3	50,839,992	3,724	2.67%
Seagate	ST6000DX000	6TB	886	53.9	2,821,207	83	1.07%
Seagate	ST8000DM002	8TB	9,839	36.3	10,910,157	316	1.06%
Seagate	ST8000NM0055	8TB	14,416	26.8	11,856,443	386	1.19%
Seagate	ST10000NM0086	10TB	1,200	24.3	897,426	14	0.57%
Seagate	ST12000NM0007	12TB	37,116	15.4	17,458,380	1,102	2.30%
Toshiba	MD04ABA400V	4TB	99	52.3	225,739	5	0.81%
Toshiba	MG07ACA14TA	14TB	1,220	11.9	441,195	9	0.74%
Totals		112,864		128,134,191	6,078	1.73%	

* AFR - Annualized Failure Rate



AFR: 1.73% fail

Reliability

Mean Time
To Failure

$$\text{MTTF of a Disk Array} = \frac{\text{MTTF of a Single Disk}}{\text{Number of Disks in the Array}}$$

Without fault tolerance, large assemblies of disks are too unreliable to be useful.

[Source: A case for redundant arrays of inexpensive disks \(RAID\)](#)

Bitrot on HDD

- Bitrot means silent corruption of data
- HDD specifications predict an Uncorrectable bit Error Rate (UER) of 10^{15}
- Evaluation [1]
 - 8x100GB HDD
 - After 2 PB reads
 - 4 read errors where observed
- How to protect against bitrot?
 - Erasure codes

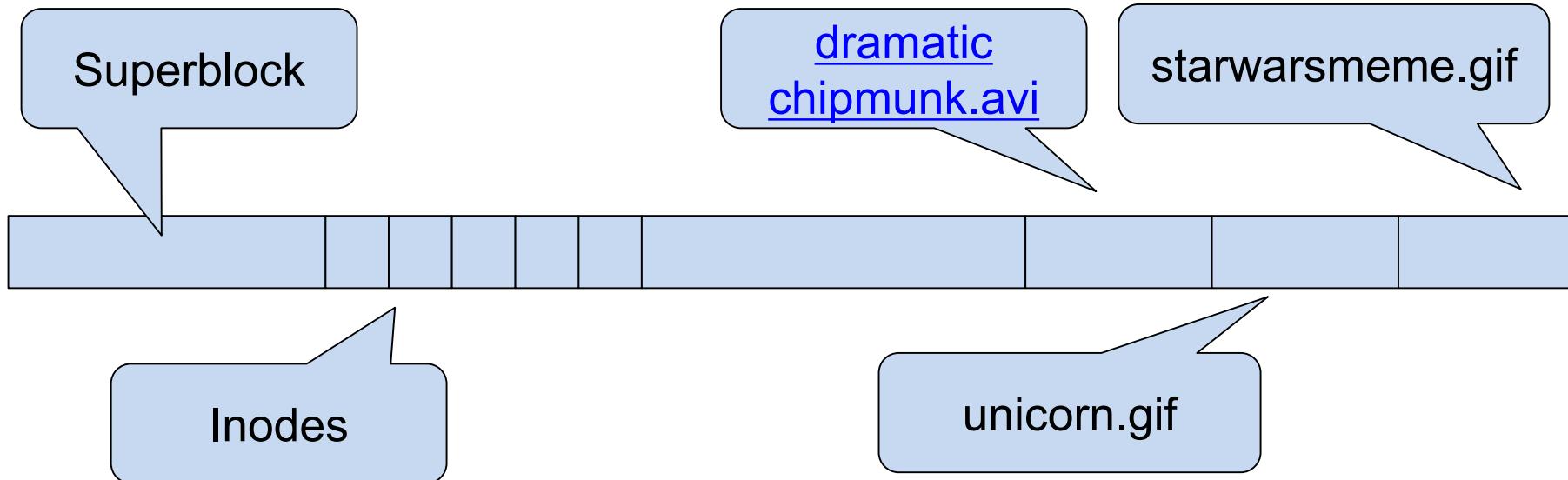
[1] <http://research.microsoft.com/pubs/64599/tr-2005-166.pdf>

Disk file systems

- Linux
 - EXT4, EXT3, EXT2
 - JFS, XFS,
 - BTRFS, ZFS
 - Pooling, snapshots, checksums
- Windows
 - NTFS
 - FAT, FAT32, exFAT, ReFS
- For simplicity only EXT2 covered

Linux EXT2

- Superblock
- Inodes (one per file or directory)
- Data blocks



EXT2 Inode

- Owner and group identifiers
- File length
- File type and access rights
- Number of data blocks
- Array of pointers to data blocks
- Timestamp
- Types
 - File
 - Directory
 - Symbolic link

Linux/fs/ext2/ext2.h

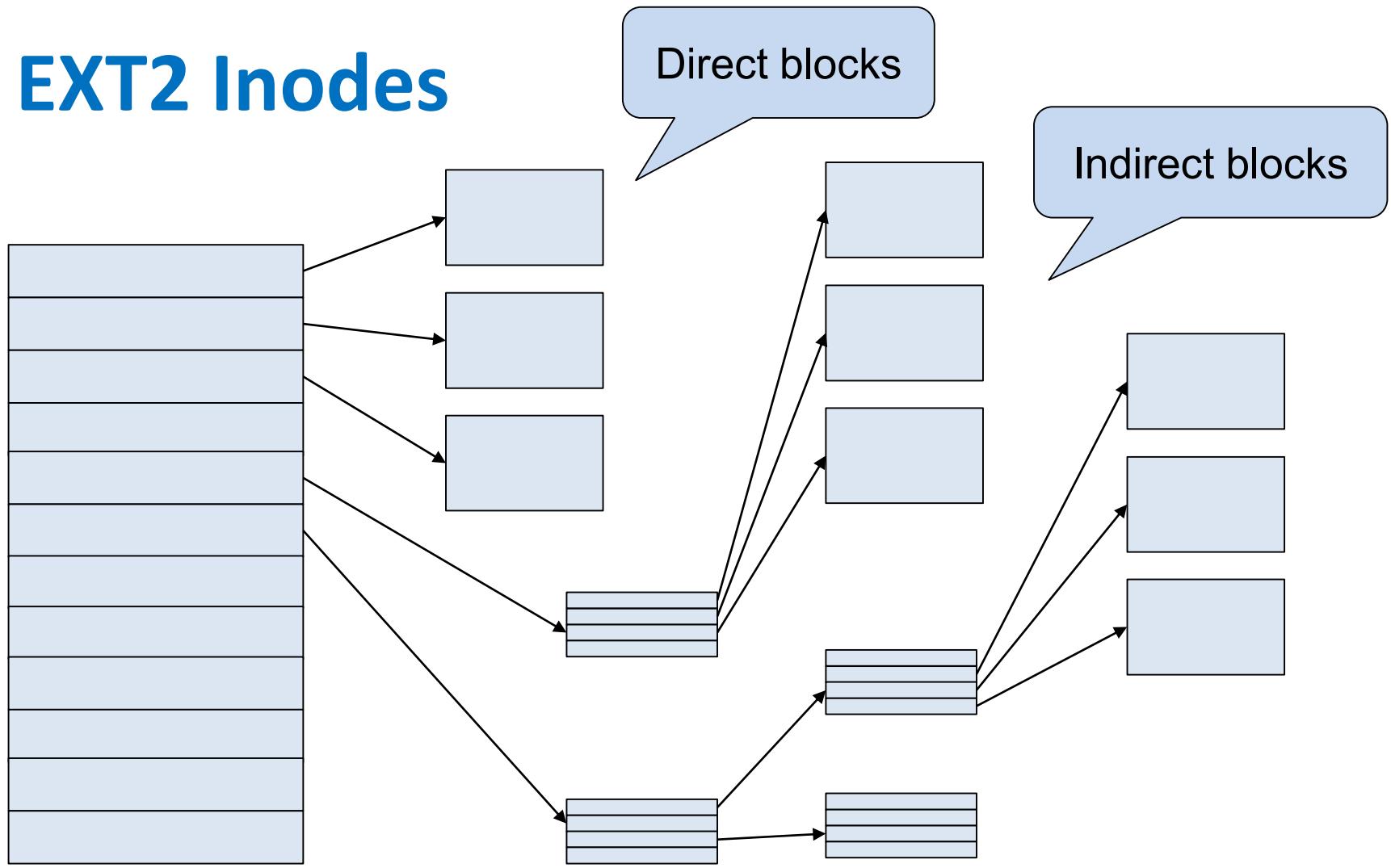
```
/*
 * Structure of an inode object
 */
struct ext2_inode {
    __le16 i_mode;      /* File mode */
    __le16 i_uid;       /* Owner Uid */
    __le32 i_size;      /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __le32 i_mtime;     /* Modification time */
    __le32 i_dtime;     /* Deletion Time */
    __le16 i_gid;       /* Group Id */
    __le16 i_links_count; /* Links count */
    __le32 i_blocks;    /* Blocks count */
    __le32 i_flags;     /* File flags */
    ...
}

struct ext2_dir_entry {
    __le32 inode;        /* Inode number */
    __le16 rec_len;      /* Directory entry length */
    __le16 name_len;    /* Name length */
    char name[];         /* File name, up to EXT2_NAME_LEN */
};
```

File metadata

Source: <https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h>

EXT2 Inodes



NETWORK FILE SYSTEM

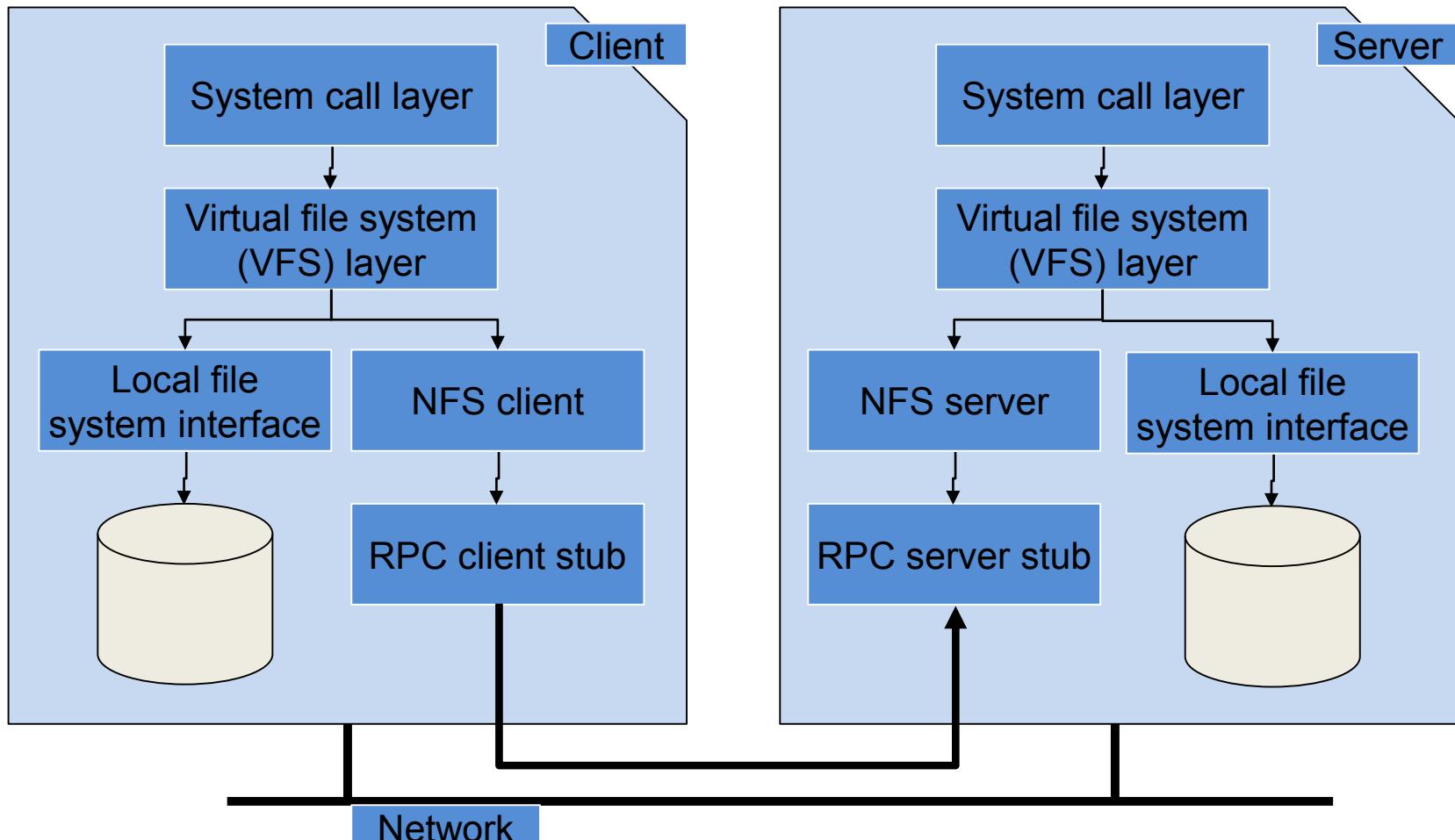
Motivation for Distributed FS

- Collaborating
 - Shared file directory for projects, companies
- Sharing resources
 - Pooling resources across multiple devices
 - Incremental scalability by adding hardware over time
- Challenges
 - Performance
 - Scalability
 - Consistency

Network File System (NFS)

- Goals:
 - **Consistent namespace** for files across computers
 - **Authorized users** can access their files from any computer
- Protocol designed for local LANs
- NFS creates a remote access layer for file systems
 - Each file is hosted at a **server**, and accessed by **clients**
 - The namespace is **distributed** across servers
 - Each client treats remote files as local ones (“virtual files”)
- NFS has a **user-centric** design:
 - Most files are privately owned by a **single user**
 - Few **concurrent access**
 - Reads are **more common** than writes

Basic NFS



Sending commands

- Essentially, NFS works as replicated system using **Remote Procedure Calls** to propagate FS operations from the clients to the servers
- Naïve solution: forward **every RPC** to the server
 - Server orders all incoming operations, performs them, returns results
- Similar to Replicated State Machine: too costly!
 - Good: concurrent clients will be consistent
 - Bad: High access latency due to the number of RPCs
 - Terrible: Server is overloaded by RPCs

Solution: Caching

- Clients use a cache to store a copy of the remote files
- Clients can then periodically synchronize with the server
- This is essentially **multi-primary replication**:
 - How should synchronization be done? (eager/lazy)
 - What is the right consistency level?

Original version: Sun NFS

- Developed in 1984
- Uses in-memory caching:
 - File blocks, directory metadata
 - Stored at both clients and servers
- Advantage: no network traffic for open/read/write/close if done locally
- Problems: **failures and cache consistency**

Caching & Failures

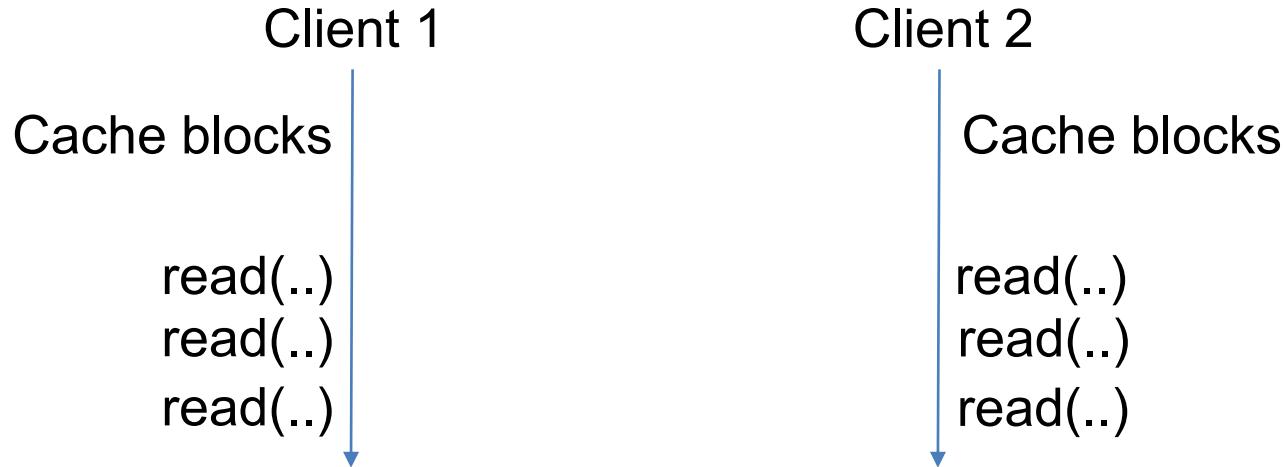
- Server crash
 - Any data not persisted to disk is lost
 - What if client does seek(); [server crash] read();?
 - Seek sets a position offset in the opened file
 - After the crash, the **server forgets** the offset, **reads return incorrect data**
- Communication omission failures
 - Client A sends delete(foo), server processes it
 - ACK for the delete is lost, meanwhile another client B sends create(foo)
 - A times out and send delete(foo) again, **deleting the file created by B!**
- Client crash
 - Since caching is in memory, **lose all updates by the client not synched to the server**

Solution: Stateless RPC

- RPC commands are **stateless**: the server does not maintain state across commands in a “session”
- e.g., `read()` is stateful (server needs to remember `seek()`) → **read(position)** is **stateless** (server has all the information needed to make the correct read)
- With stateless RPC, server can fail and continue to serve commands without recovering the old state
- NFS’ RPCs are also **idempotent**, so repeating a command has no side effect: `delete("foo")` becomes **delete(someid)**, so it cannot wrongly delete a new file named “foo”

Cache Consistency

- Clients can cache file blocks, directory metadata, etc.



- What happens if both clients want to write?

Solution: Time-bounded consistency

- Flush-on-close: When a file is **closed**, the modified blocks are sent to the server **synchronously** (i.e., `close()` does not return until update is finished)
- Each client periodically checks with the server for updates
- Clients will synchronize their cache **after some bounded time** if there are no more updates; otherwise they could always read stale data
- (Optional) **close-to-open** consistency: force clients to check with the server when opening a file, trading performance for consistency

Concurrent Writes in NFS

- NFS does not provide any guarantees for concurrent writes!
- The server may update using one client's writes, the other's writes, or a mix of both!
- Not usually a concern due to the **user-centric** design: assumed there are no concurrent writes.

NFS Summary

- Transparent remote file access using the **virtual file system**
- Client-side caching for improved performance
- Stateless and idempotent RPCs for fault-tolerance
- Periodical synchronization with the server, with flush-on-close semantics
- No guarantees for concurrent writes

GOOGLE FILE SYSTEM

GFS (The Google File System)

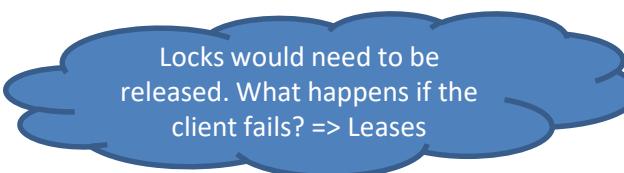
- Designed for big data workloads:
 - Huge files, mostly appends, concurrency, huge bandwidth
- Fault tolerance while running on inexpensive commodity hardware
 - 1000s machines where **failure is the norm**
- Introduces an API which is designed to be implemented scalably (non-POSIX)
- Architecture: one master, many chunk (data) servers
 - Master stores metadata, and monitors chunk servers
 - Chunk servers store and serve chunks



Published 2003
(We can assume that major revisions of GFS)

Definitions

- Mutation
 - The GFS paper uses the word „mutation“ for any modification done to a file. This can be an in-place update, an append or a file creation, hence we also use it in the slides.
- Leases
 - Ownership for a specified length of time
 - Leases can be renewed or extended by the owner
 - When the owner fails, the lease expires
 - The owner can end the lease early



Locks would need to be released. What happens if the client fails? => Leases

Design assumptions

- Inexpensive commodity components that often fail
- Typical file size 100MB or larger, no need to optimize for small files
- Read workload
 - Large streaming reads
 - Small random reads
- Write workload
 - Append to files
 - Modification supported but a design goal
 - Hundreds of concurrently appending clients
- Bandwidth is more important than low latency

Interface

- Does not support full POSIX interface
 - POSIX requires many guarantees which are hard or impossible to fulfill in distributed applications
- Supported operations
 - Create, delete, open, close, read, write
 - Snapshot
 - Creates a copy of a file or a directory tree at low costs
 - Record append
 - Allows multiple clients to append data to the same file while guaranteeing atomicity

Architecture

- Files
 - Divided into fixed-size chunks
 - Identified by an immutable and unique id (chunk handle)
- Single master
 - Maintains file system metadata
 - Namespace, access control information, mapping from files to chunks, location of chunks
 - Garbage collection (deferred deletion of files)
 - Sends heartbeat messages to chunk server
- Multiple chunk servers
 - Chunks are stored on disks as files
 - Each chunk is replicated to multiple chunk servers (depending on the replication factor of the region)

Architecture overview

(1) Using file name and chunk index, the GFS master responds where to find it

In case the client wants a specific byte range, it needs to calculate the chunk index. For this the client has to know the chunksize.

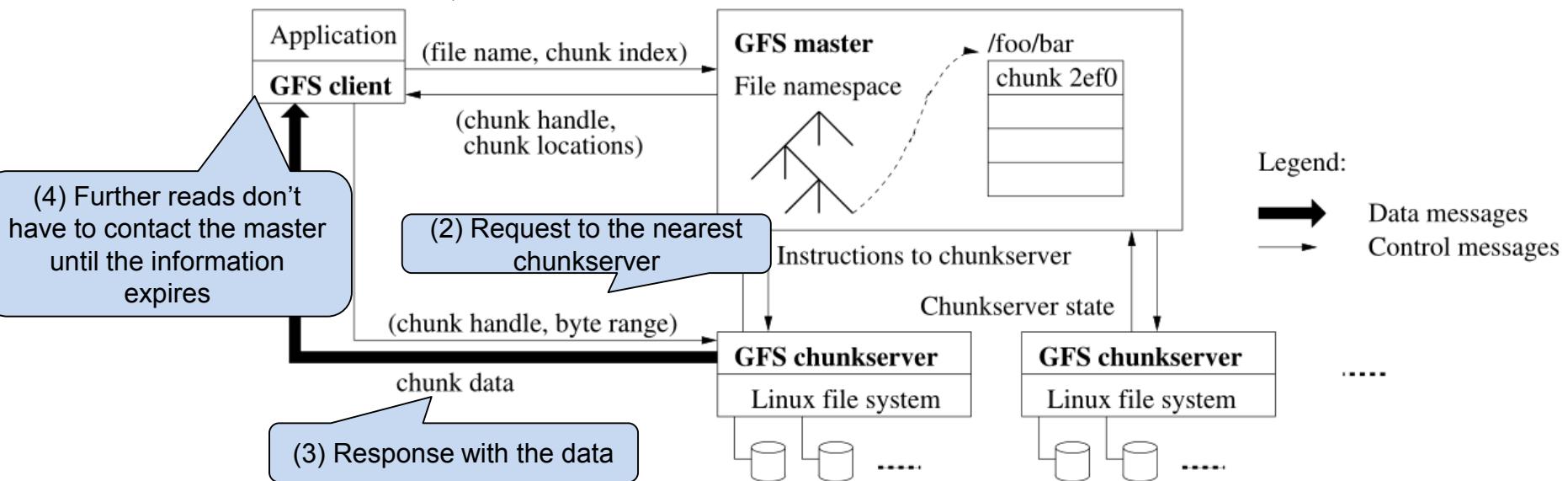
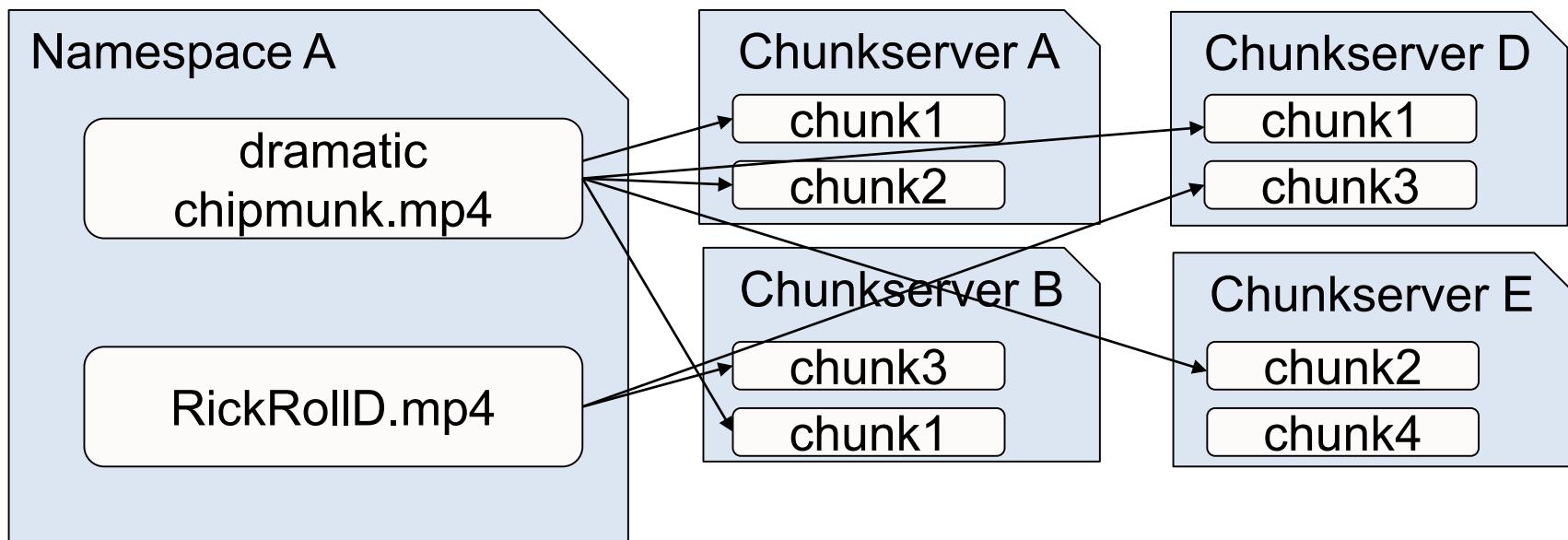


Figure 1: GFS Architecture

Source:[2]

Metadata (Master)

- File and chunk namespaces
- Mapping from files to chunks
- Location of each chunk's replicas



Metadata (Replication)

- Replicated to a shadow master
 - Namespaces
 - File to chunk mapping
- Location of the chunks is in memory only
 - In case of failover or start, the master asks the chunkservers which chunks they have to rebuild the location/chunk mapping
 - Periodic scanning is used to implement
 - Garbage collection (when files are deleted)
 - Re-replication (chunkserver failure)
 - Chunk migration (to balance load and disk space)
 - Metadata has to fit in memory (64bytes/chunk)

Operation Log (Master)

- Contains a historical record of metadata changes
- Changes of metadata are only visible to clients after they are persisted
- Replicated to standby master
- Recovering from failure by replaying operation log
- Check pointing is used to minimize replaying effort

Consistency model

- Relaxed consistency model
- Applications using GFS have to accomodate for the model
 - Instead of mutating parts of a file, append to the end
 - Another component then sequentially reads the file and creates a single new one
 - Metadata is atomically, e.g., rename

Guarantees provided by GFS

- File namespace mutations are atomic
 - e.g., file creation, ...
 - Uniquely handled by master
 - Masters operation log defines a global total order of the operations
- File manipulations
 - The state of a file region after a data mutation depends on the type of the action
- Definitions
 - Consistent – all clients see the same data, regardless of which replica they read from
 - Defined – consistent and also the clients see what the file mutation writes in its entirety
 - Inconsistent – multiple clients see different kinds of data

When higher guarantees needed, user has to implement coordination using other services

How is fault-tolerance achieved?

- Master
 - Operation Log, Replication to shadow master
- Chunk server
 - All chunks are versioned
 - Version number updated when a lease is granted
 - Chunks with old versions are not served and are deleted
- Chunks
 - Re-replication triggered by master maintains replication factor
 - Rebalancing
 - Data integrity checks

How is high-availability achieved?

- Fast recovery of master
 - Check pointing and operation log
- Heartbeat messages
 - Include piggybacked information
 - Can trigger re-replication
 - Share current load
 - Can trigger garbage collection
 - => Chunkservers can fail any time
- Diagnostic tools

How to use the API within an application?

- Typical usages
 - Append, instead of overwriting
 - Write to temporary file from beginning to end
 - Atomically renames the file from the temporary name to a permanent name after writing the data
 - Periodically checkpoints how much has been successfully written
 - Application-level checksum
- In essence
 - The developer of an application which uses GFS has to be aware of these limitations and has to work with/around them

Summary on GFS

- Highly concurrent reads and appends
- Highly scalable
- On cheap commodity hardware
- Built for map-reduce kind of workloads
 - Reads
 - Appends
- Developers have to understand the limitations and may have to use other tools to work around
- No POSIX API, would require many guarantees which are difficult to fulfill in DS

HADOOP DISTRIBUTED FILE SYSTEM

HDFS

- The Hadoop Distributed File System
- Originally built as infrastructure for the Apache Nutch web search engine
- Quite similar to GFS
- Main Reference used
 - [3] Shvachko, K.; Hairong Kuang; Radia, S.; Chansler, R., "**The Hadoop Distributed File System**"

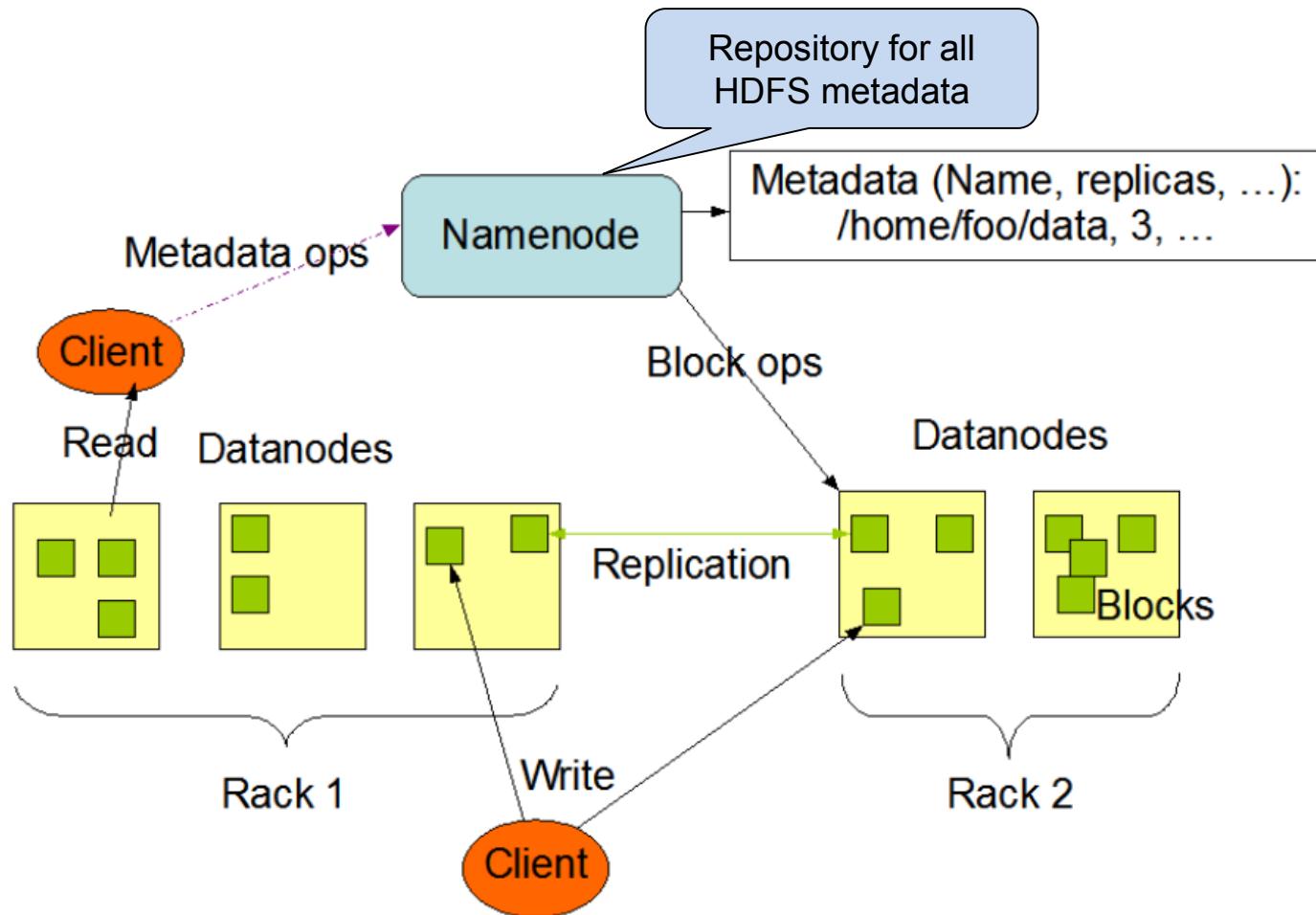
Used by

- <http://wiki.apache.org/hadoop/PoweredBy>
- Facebook
 - One cluster with 8800 cores and 12 PB raw storage
- LinkedIn
 - Multiple clusters and also PB of data
- Twitter
- Powerset (bought by Microsoft)
- ...

Use cases

- MapReduce kind of workloads, similar to GFS
- Backend for HBase
- Many more uses
 - http://hadoopilluminated.com/hadoop_illuminate_d/Hadoop_Use_Cases.html

HDFS Architecture



Source: http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf

Roles

- HDFS Client
- NameNode
 - Only one
 - Main task is to coordinate
 - Metadata
- DataNode
 - Many datanodes
 - Store data

HDFS Client – Example code

```
//Some operations

FSDataInputStream open(Path f)
FSDataOutputStream create(Path f, EnumSet<CreateFlags> flags, CreateOpts options)

void rename(Path src, Path dst, Renameoptions options)
Boolean delete(Path f, Boolean recursive)
FileStatus getFileStatus(Path f)
```

<http://hadoop.apache.org/docs/current2/api/>

Streaming writer – Example code

- org.apache.Hadoop.io.SequenceFile
- Flat files consisting of binary key/value pairs

```
SequenceFile.Writer writer =  
    new SequenceFile.Writer(fs, conf, out, Text.class, Text.class);  
  
writer.append(new Text(http://tum.de) , new Text("<html> ....</>"));  
...  
...  
writer.close()
```



The diagram illustrates a single entry in a SequenceFile. It consists of two blue rectangular boxes. The left box is labeled "Key" and has a blue arrow pointing from the word "writer.append" in the code above. The right box is labeled "Value" and has a blue arrow pointing from the word "Text" in the code above. Between these two boxes is a horizontal line with a small gap, representing the binary key-value pair stored in the file.

NameNode

- NameNode keep entire namespace in RAM
 - Hierarchy of files and directories
 - Inodes have attributes, permissions, modifications, access times, disk space quotas
- Checkpoints to the local file system, changes recorded as journal
- For improved durability redundant copies are stored on multiple local volumes or remote servers
- Batches operations for improved performance
 - Transactions are committed together

Checkpoint/Backup node

- NameNode can alternatively execute either of two other roles
- Role specified at node startup
- CheckpointNode
 - Combines existing checkpoint and journal
 - Returns the checkpoint to the NameNode
- BackupNode
 - Maintains an in-memory filesystem namespace which is synchronized with NameNode

DataNode

- Each block replica on a DataNode consists of two files
 - Block Metadata (Checksums for the data and timestamps)
 - Data (actual length)
 - File content is split into blocks, typically 64MB
 - If possible each chunk resides on a different DataNode
- Startup
 - DataNode connects to NameNode and performs handshake
 - Verify namespace and software version
 - DataNode is registered at the NameNode
 - DataNode sends block report (Block Metadata), renewed every hour
 - Begin sending heartbeat signals to NameNode
 - Heartbeat contains storage capacity, other statistics relevant for rebalancing

Node failure detection

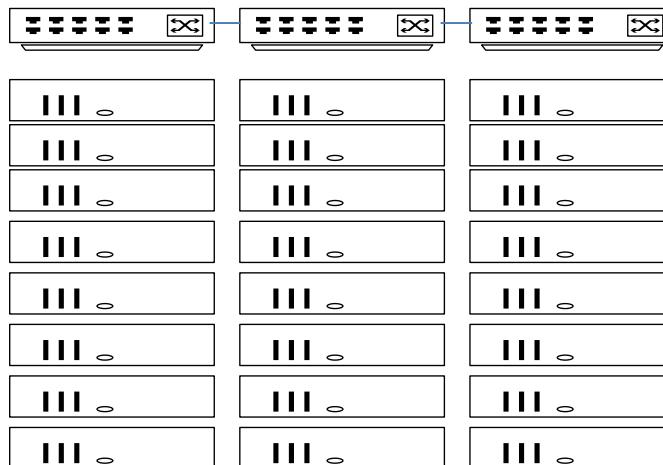
- NameNode
 - Prior to Hadoop 2.0, NameNode was single point of failure
 - Now: two redundant NameNodes in active/passive configuration
- DataNode
 - Heartbeat every 3 seconds to NameNode
 - If no heartbeat message received, DataNode is considered dead

Corrupted data

- Data includes checksum
- Periodically DataNodes send FileChecksum to the NameNode
- Try to resolve corrupted data
 - If no uncorrupted data can be found, it is still not deleted automatically, the user can still get the corrupted data and maybe resolve the issue manually

Block placement

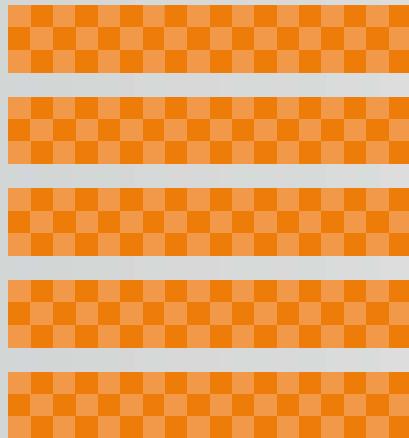
- Network bandwidth within rack greater than between racks
- HDFS estimates network bandwidth between two nodes by their distance
- Second and third replica typically placed in a different rack



CEPH

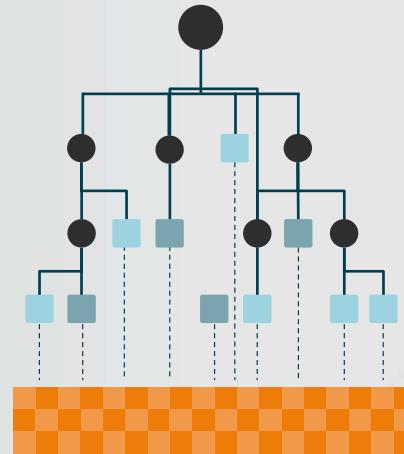
Distributed Systems (H.-A. Jacobsen)

Ceph Storage Platform



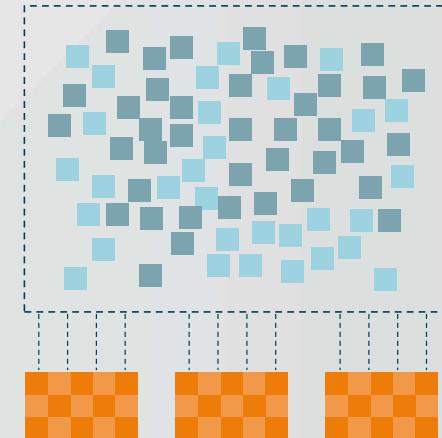
BLOCK STORAGE

Physical storage media appears to computers as a series of sequential blocks of a uniform size.



FILE STORAGE

File systems allow users to organize data stored in blocks using hierarchical folders and files.



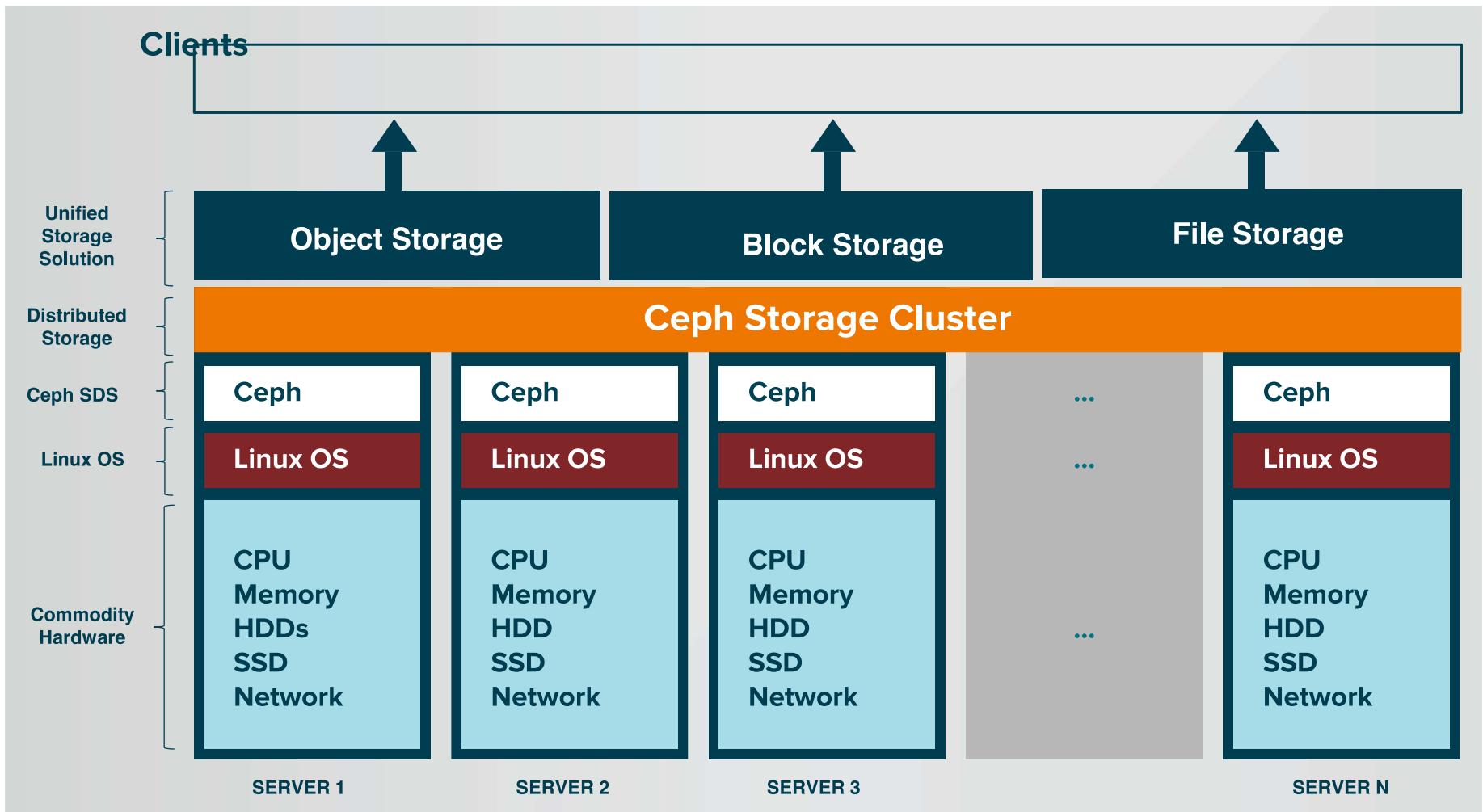
OBJECT STORAGE

Object stores distribute data algorithmically throughout a cluster of media, without a rigid structure.

Ceph Storage Platform

- Open Source Software Defined Storage
- Unified Storage Platform (Block, Object, File Storage)
- Runs on Commodity Hardware
- Self Managing, Self Healing
- Massively Scalable
- No Single Point of failure
- Designed for cloud infrastructure and emerging workloads

Stack Layout

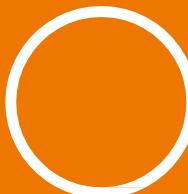


Core Components



OSDs (Object Storage Daemon)

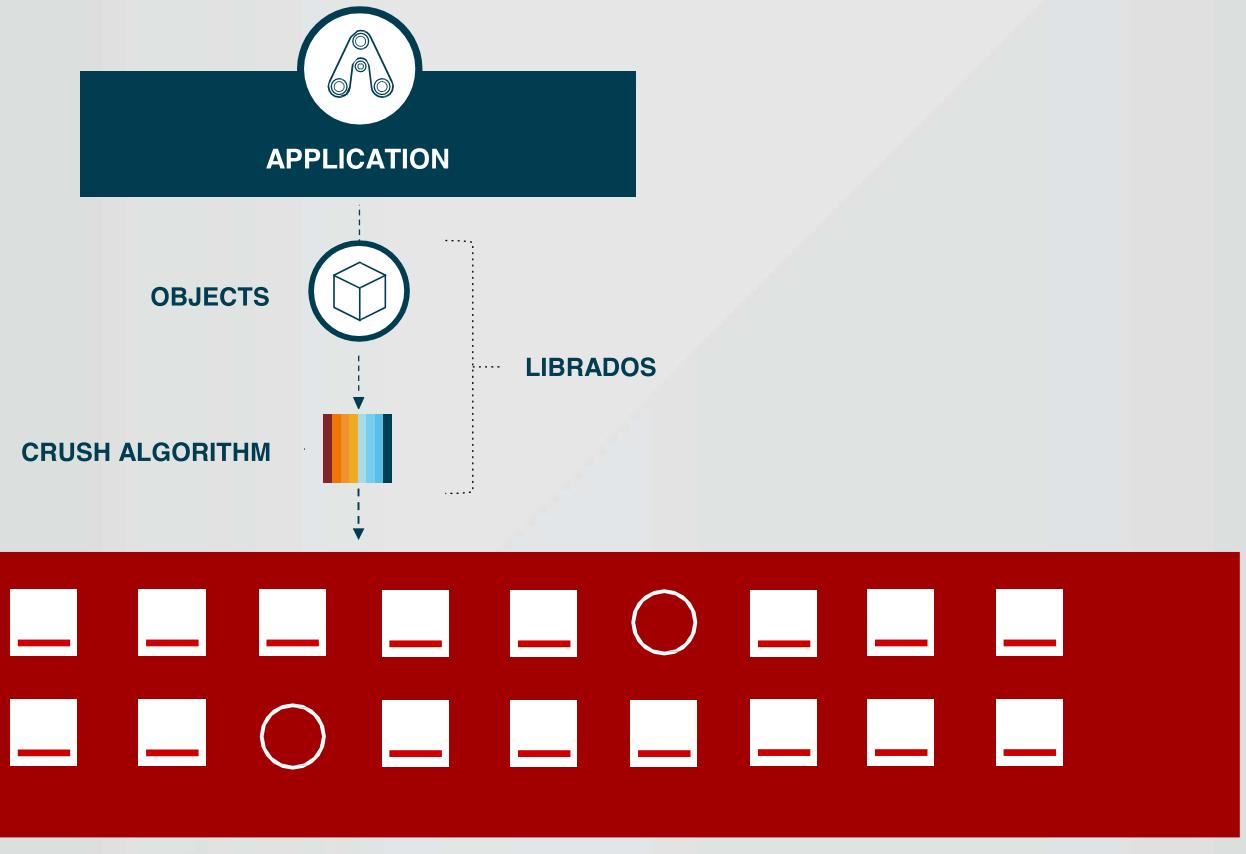
- 10s to 10000s in a cluster
- Typically one daemon per physical HDD
- Serve stored data to clients
- Intelligently peer for replication & recovery



Monitors

- Maintain cluster membership and state
- Provide consensus for distributed decision-making
- Small, odd number
- Do not store data

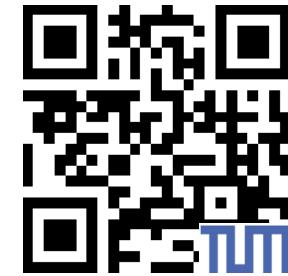
Data Distribution



ERASURE CODING

Types of Erasure Coding

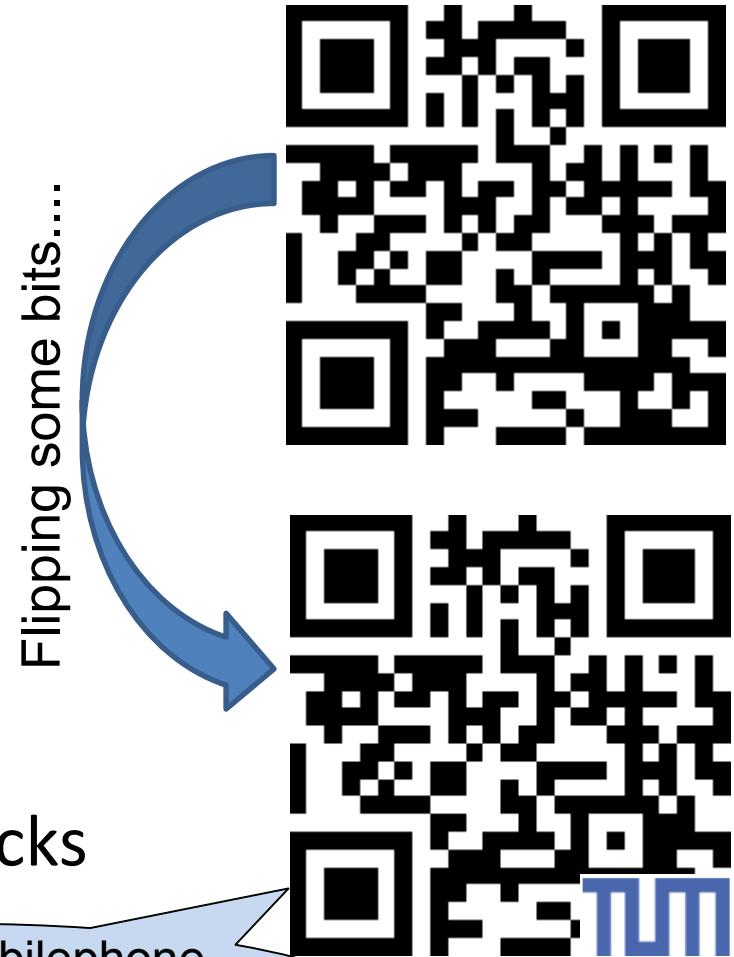
- **Linear block code**
 - Reed Solomon Code
 - Can sustain flipped bits or loss
 - Distributed file systems
- **Fountain code**
 - LT Codes
 - Can sustain missing chunks, e.g., missing packet
 - P2P systems, Torrents, Video streaming ...



Filling a bucket
from the firehose

Reed-Solomon

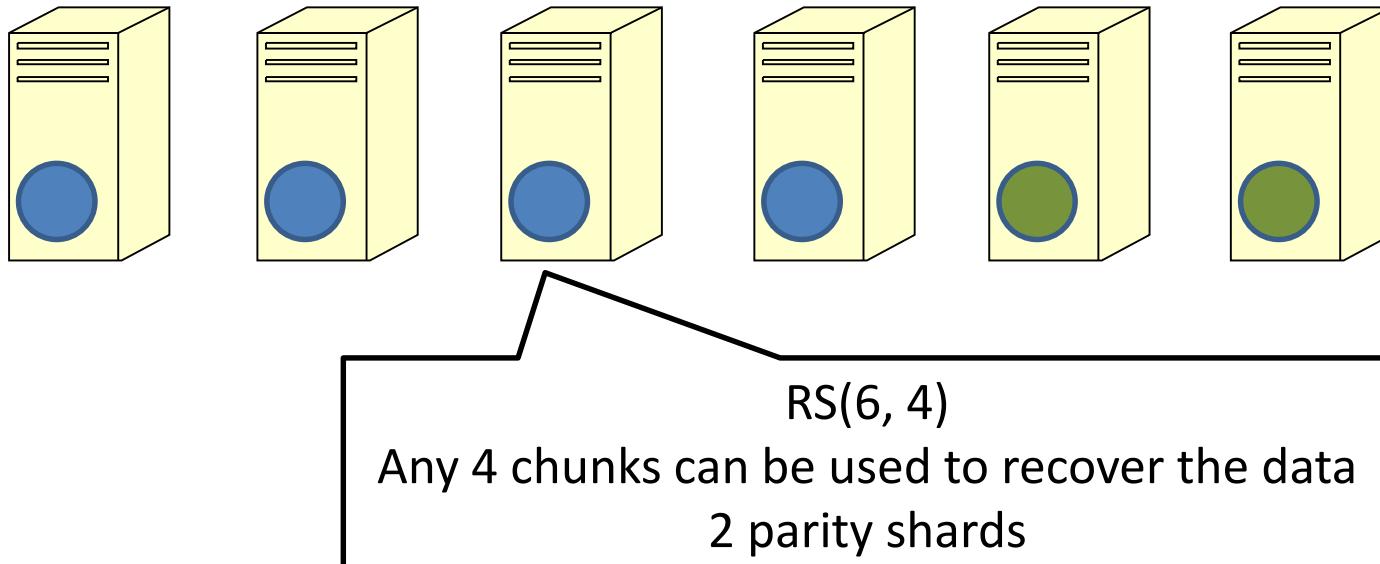
- Error-correcting code
 - Used in QR codes
- Block encoding
 - Data blocks
 - Error-correction blocks
 - Read data blocks first
 - Else, decode with error blocks



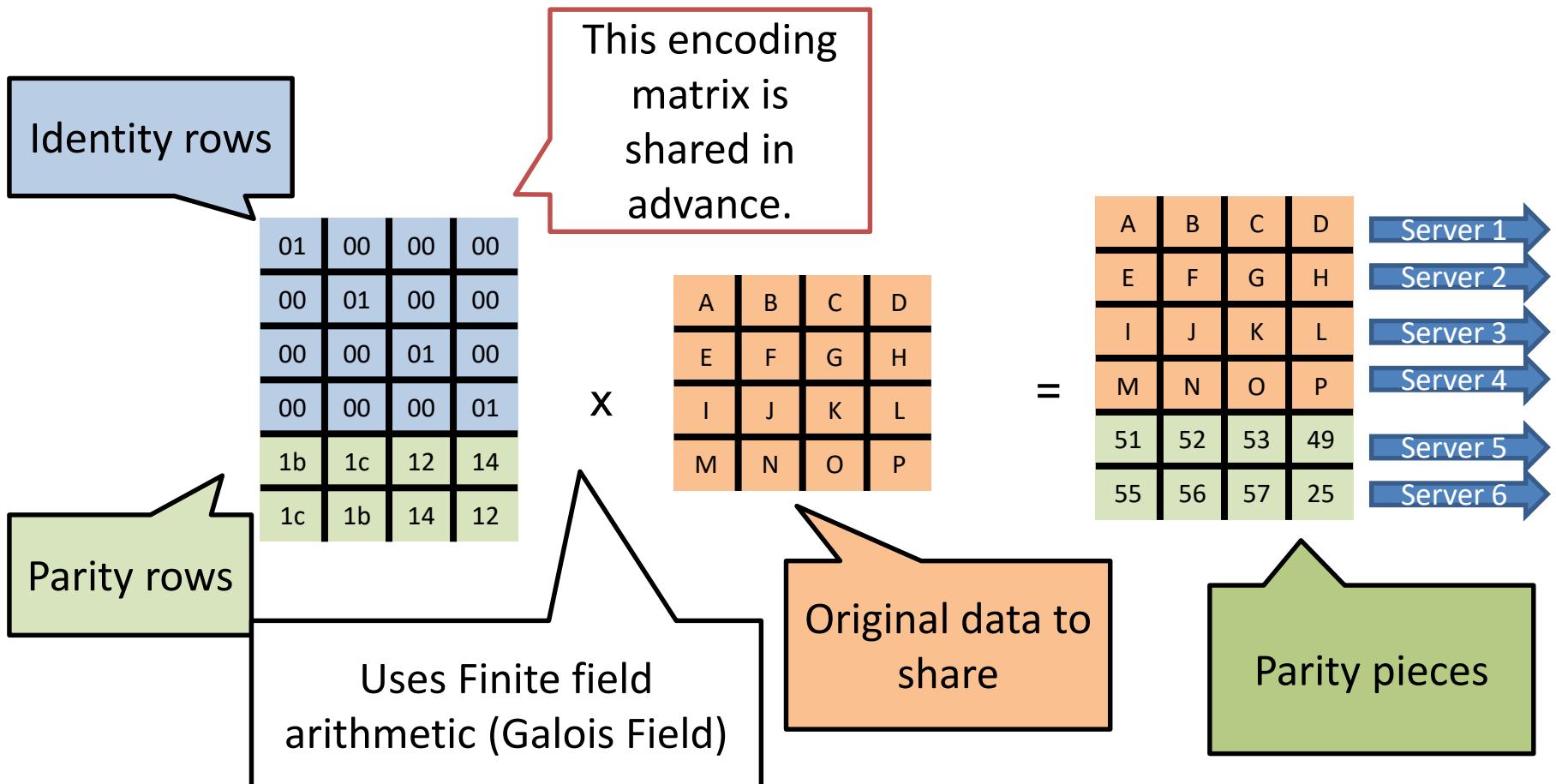
Take your mobilephone
and scan this QR-Code,
does it still work?

Reed Solomon (n, k)

- Any k symbols suffice for full data recovery
- k data symbols, $n - k$ parity checks



Reed-Solomon Encoding



Optimal encoding:
Any 4 pieces out of 6 can be used to rebuild the original data.

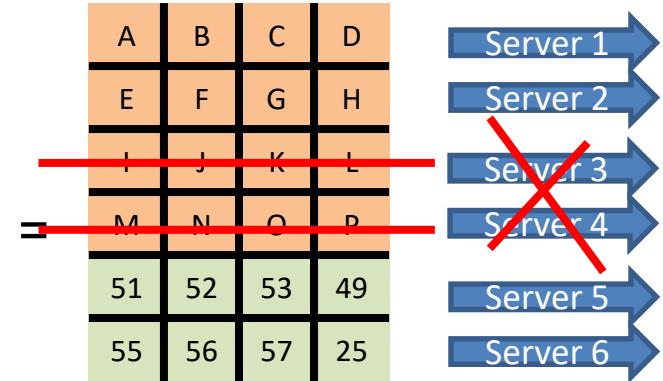
Data Loss Example

$$\begin{matrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ \hline 00 & 00 & 01 & 00 \\ \hline 00 & 00 & 00 & 01 \\ \hline \end{matrix} \times$$

Build the partial
encoding matrix.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Original data
to receive



Suppose only 4
received pieces
are intact.

Any 4 pieces out of 6 can be used to rebuild the original data.

Constructing the Decoding Matrix

The decoding and
the partial
encoding matrices
cancel out.

Build the decoding
matrix: inverse of the
partial encoding
matrix.

$$\begin{array}{|c|c|c|c|} \hline 01 & 00 & 00 & 00 \\ \hline 00 & 01 & 00 & 00 \\ \hline 8d & F6 & 7b & 01 \\ \hline F6 & 8d & 01 & 7b \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline 01 & 00 & 00 & 00 \\ \hline 00 & 01 & 00 & 00 \\ \hline 1b & 1c & 12 & 14 \\ \hline 1c & 1b & 14 & 12 \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline A & B & C & D \\ \hline E & F & G & H \\ \hline I & J & K & L \\ \hline M & N & O & P \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 01 & 00 & 00 & 00 \\ \hline 00 & 01 & 00 & 00 \\ \hline 8d & F6 & 7b & 01 \\ \hline F6 & 8d & 01 & 7b \\ \hline \end{array} \times \begin{array}{|c|c|c|c|} \hline A & B & C & D \\ \hline E & F & G & H \\ \hline 51 & 52 & 53 & 49 \\ \hline 55 & 56 & 57 & 25 \\ \hline \end{array}$$

Partial encoding matrix

Need to find the
inverse matrix

Reconstructing the Original Data

Uses Finite field arithmetic (Galois Field)

01	00	00	00
00	01	00	00
8d	F6	7b	01
F6	8d	01	7b

X

A	B	C	D
E	F	G	H
51	52	53	49
55	56	57	25

=

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

In exercise we use normal arithmetic for simplicity

To obtain the original data, multiply the decoding matrix with the 4 pieces.

Fountain codes

- Properties
 - The number of encoding symbols is limitless
 - **It doesn't matter what is received or lost, it only matters that enough is received**
- Encoder
 - Encoding symbols can be generated on the fly, called Droplets
- Decoder
 - Can recover data from any set of Droplets slightly longer than the data

M. Luby, 2002 <http://dx.doi.org/10.1109/SFCS.2002.1181950>

Luby Transform Codes

- Source splits data into equal chunks
 - In case data is sent over UDP, typically MTU size
- Encoder
 - Generates infinite stream of droplets by XOR
 - Each droplet has 1 or more degrees
- Decoder
 - Receives (or not) droplets

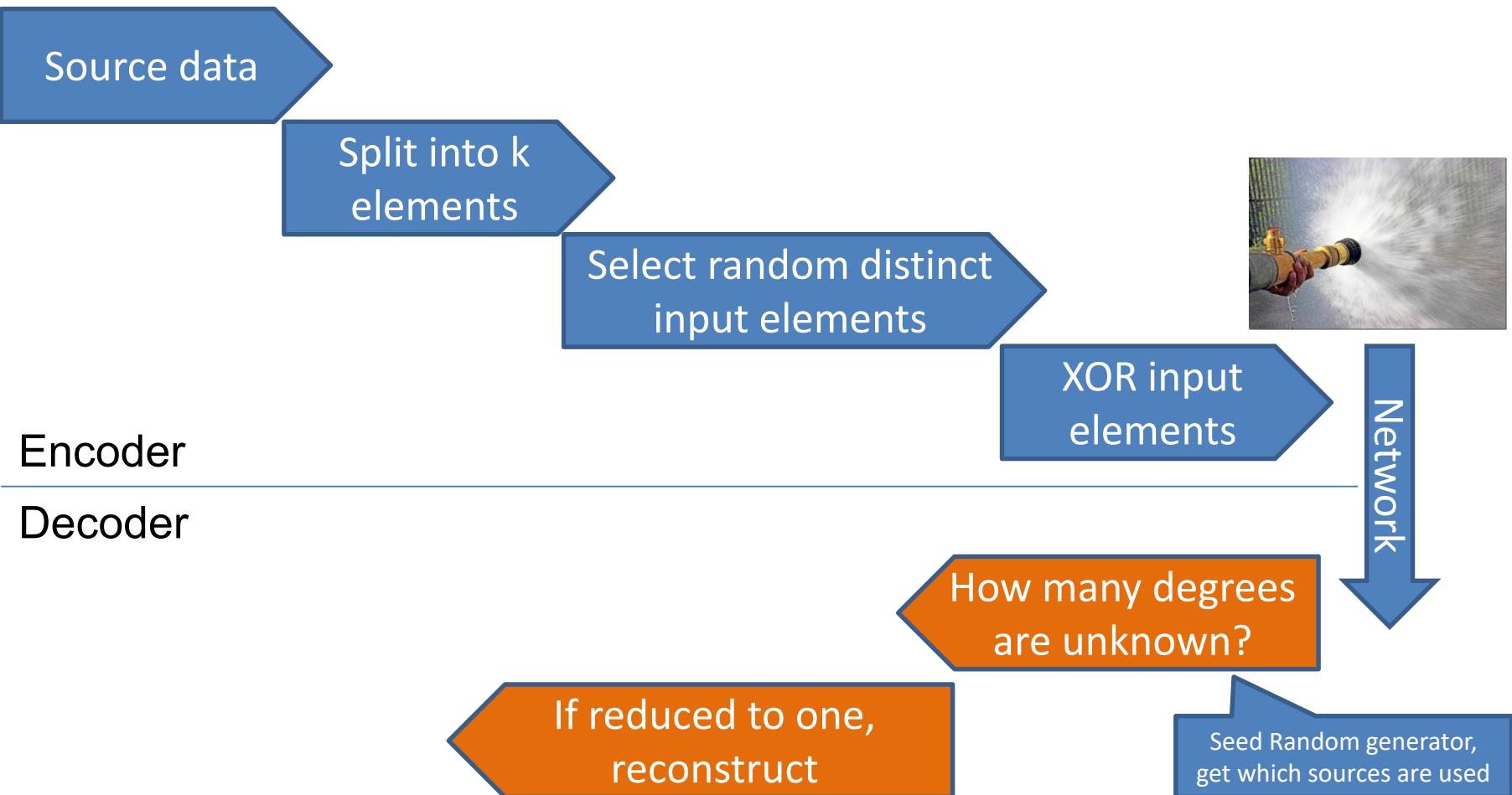
Luby Transform Codes - Encoding

- Randomly choose the degree from a degree distribution (based on Soliton distribution, e.g. 2)
- Choose uniformly random distinct input symbols (choose 2 symbols at random)
- Construct Droplet (xor these symbols)

Droplet

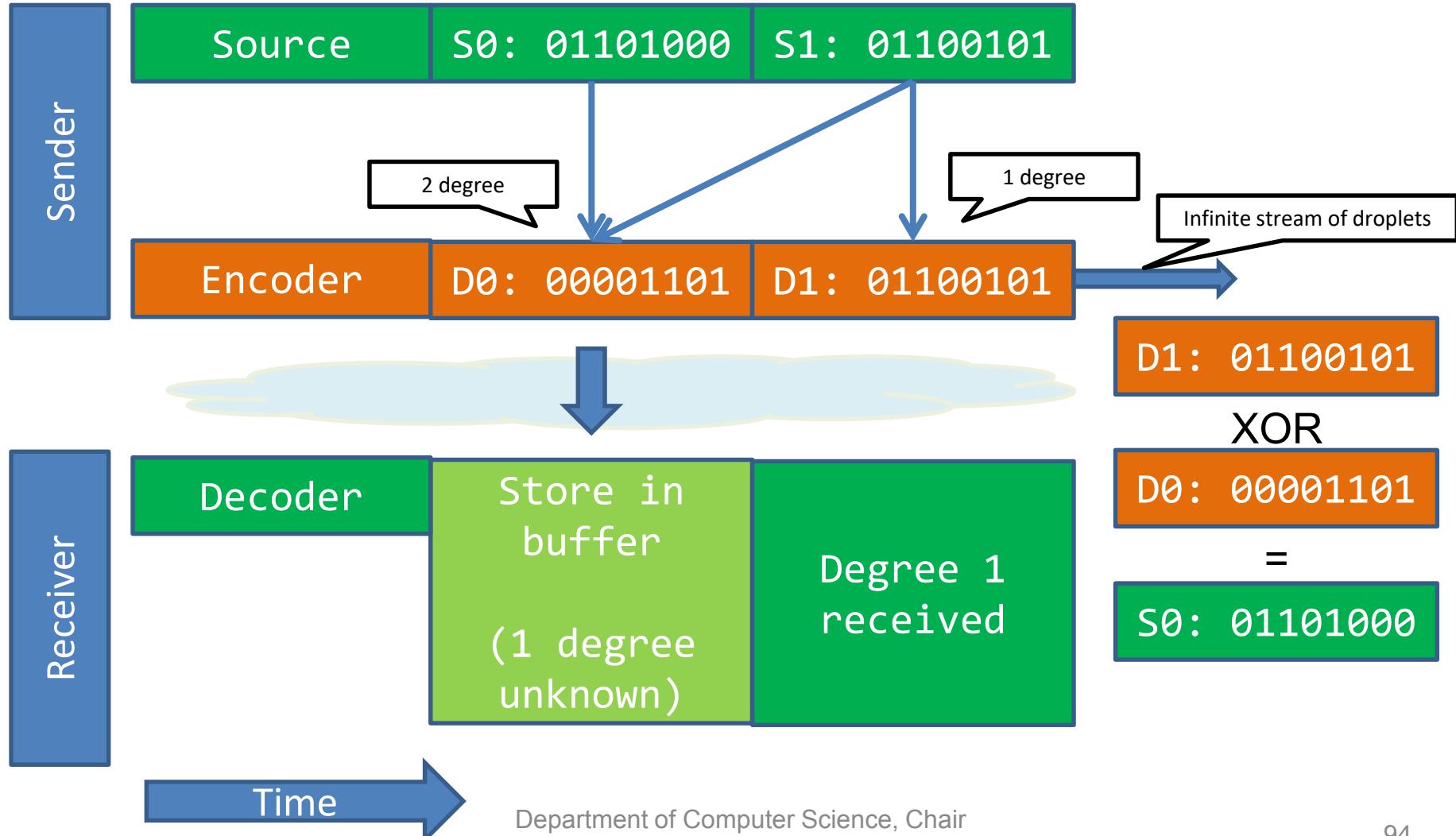
- Header
 - Contains information which source symbols are used
 - Bad: Header could become large when expressed as list
 - Alternative:
 - Both sender and receiver use same random number generator
 - Send number of sources + random seed
 - Same length independent of amount of degrees
- Data
 - If degree 1, just the data item
 - If degree > 1, XOR all data items

LT Code



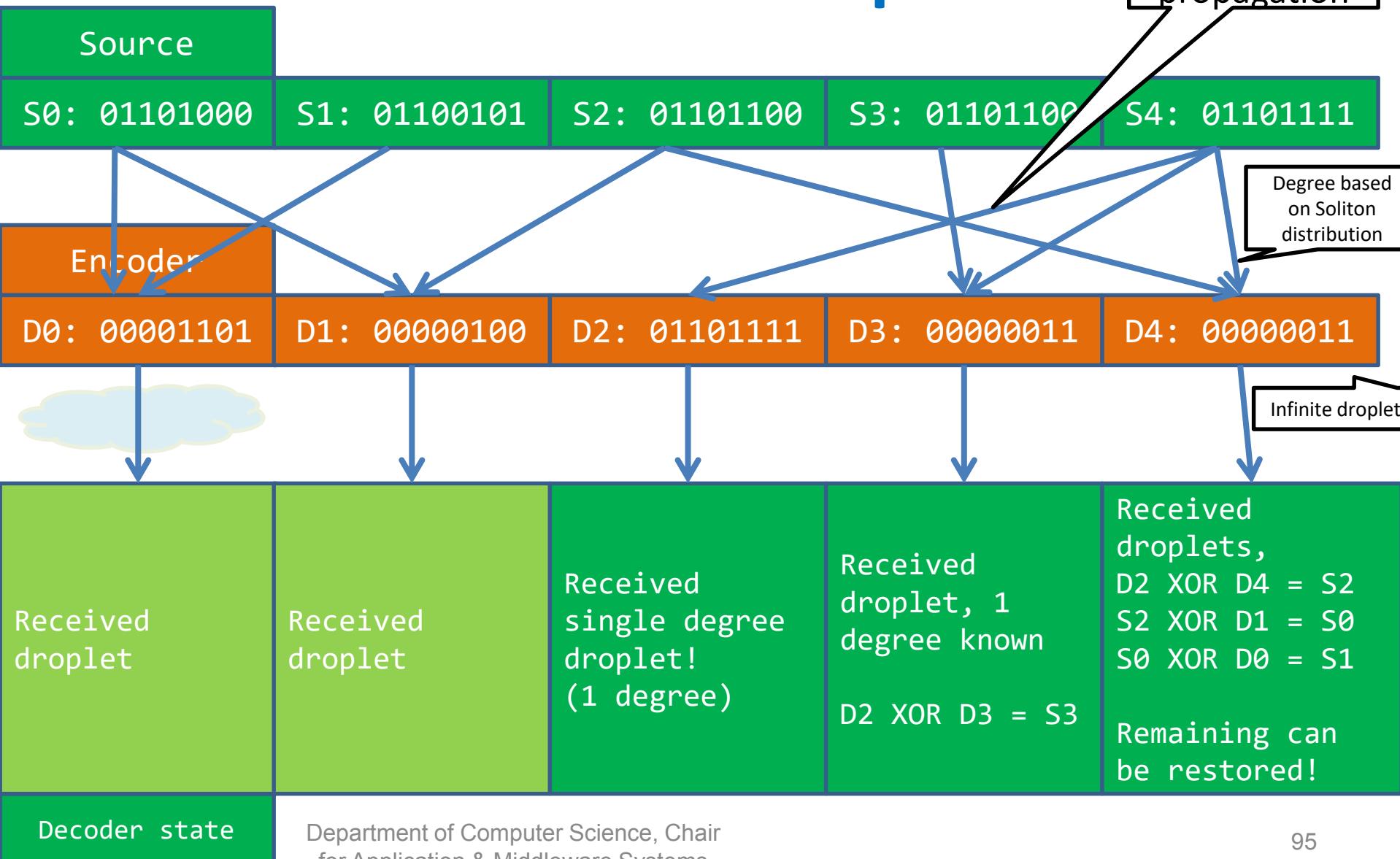
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Micro Example (LT Code)



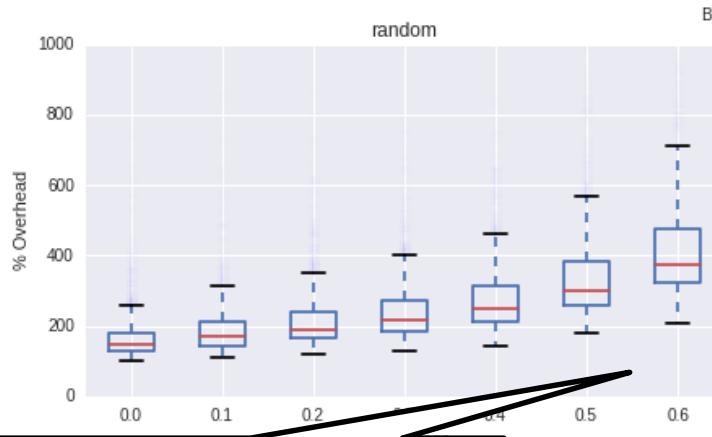
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

LT Code Example

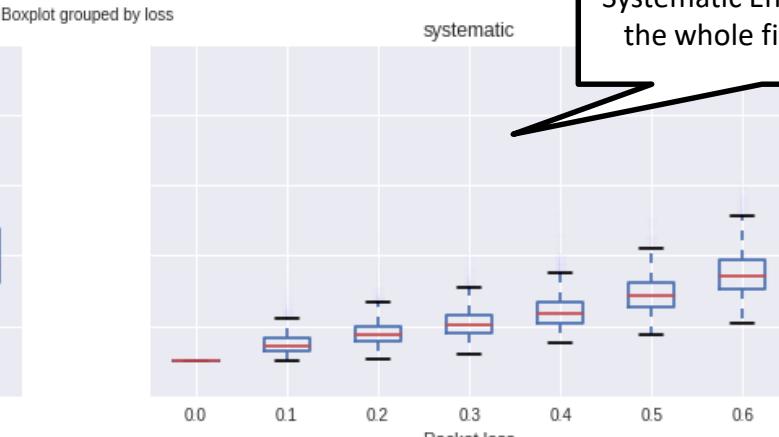


LT Code (Evaluation UDP)

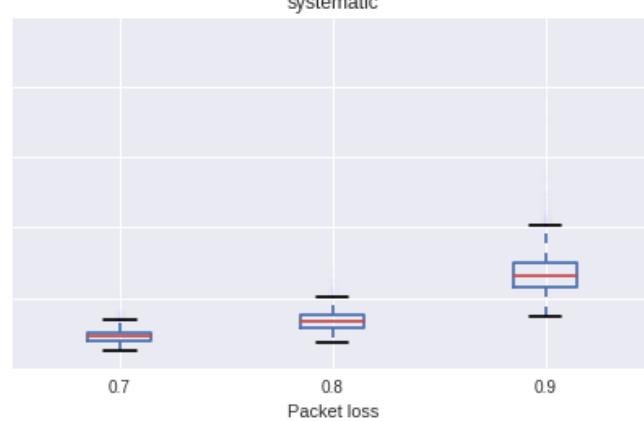
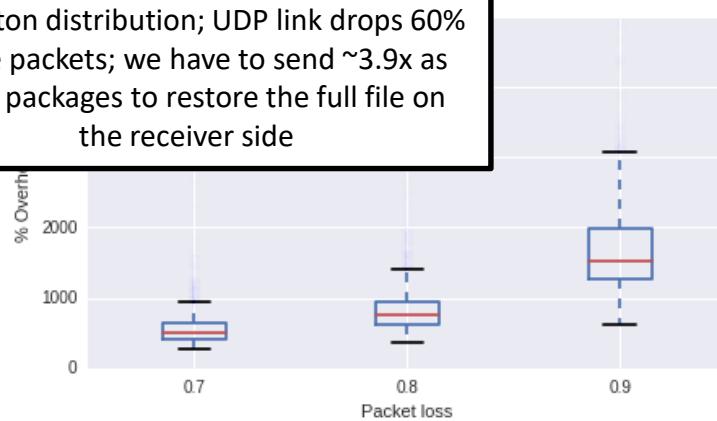
- Data: 1MB, 32 bit chunks, 1000 runs



Encoder randomly creates droplets based on Soliton distribution; UDP link drops 60% of the packets; we have to send ~3.9x as many packages to restore the full file on the receiver side

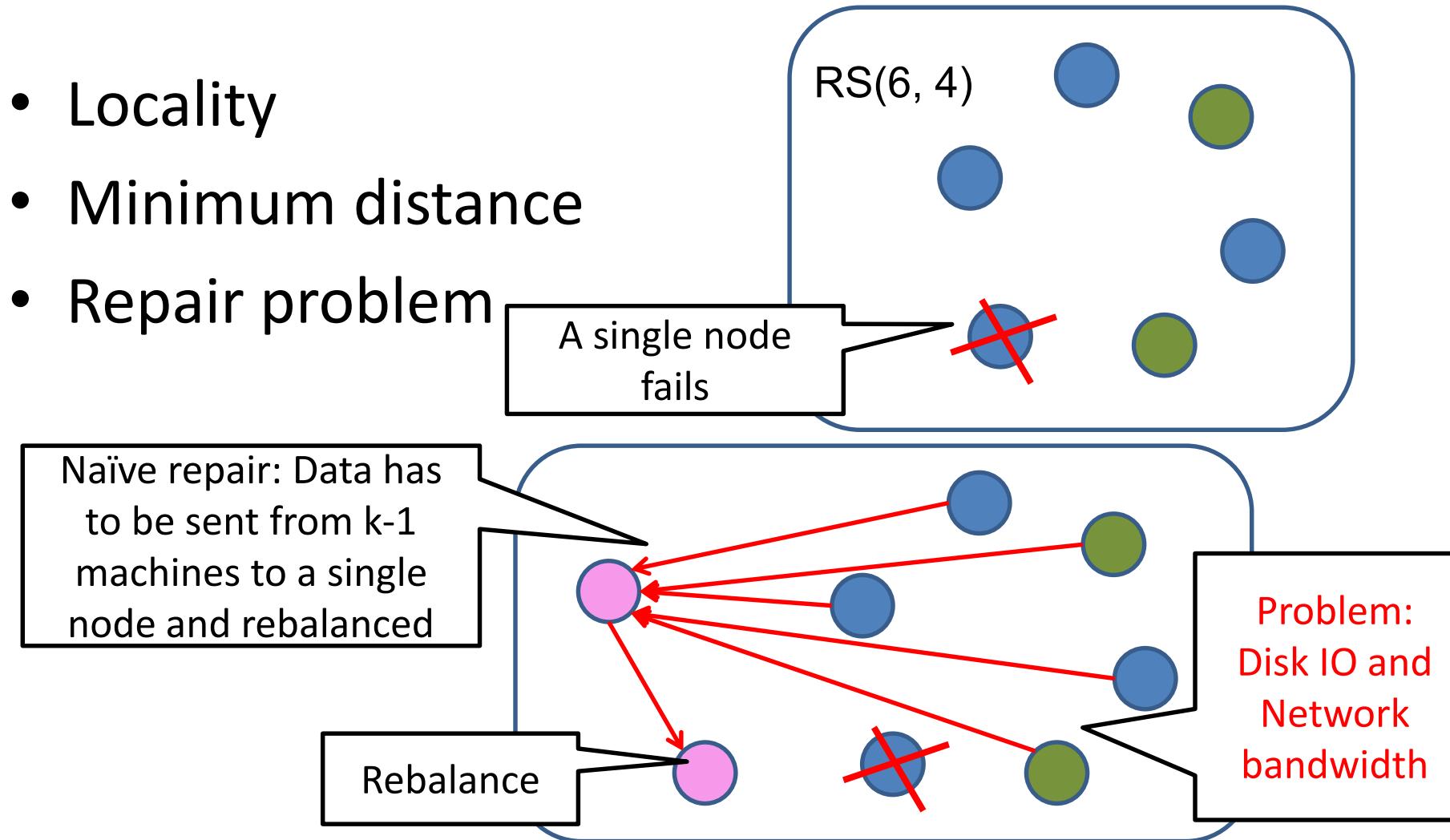


This implementation of a Systematic Encoder first sends the whole file in sequence.



Challenges of using RS in DS

- Locality
- Minimum distance
- Repair problem



As an Alternative to Replication

	Replication	Erasure Encoding
Storage	Make N full copies, Efficiency of: 1/N	K data pieces, N total pieces, Efficiency of: K/N
Computation	No encoding and decoding needed	Encoding required, decoding needed if using parity pieces
Fault-tolerance	Tolerate N-1 failures	Tolerate N-K failures
Communication	1 Node for full data	K Nodes for pieces

Use replication if **latency** is prioritized or computational resources are limited (e.g., mobile devices).

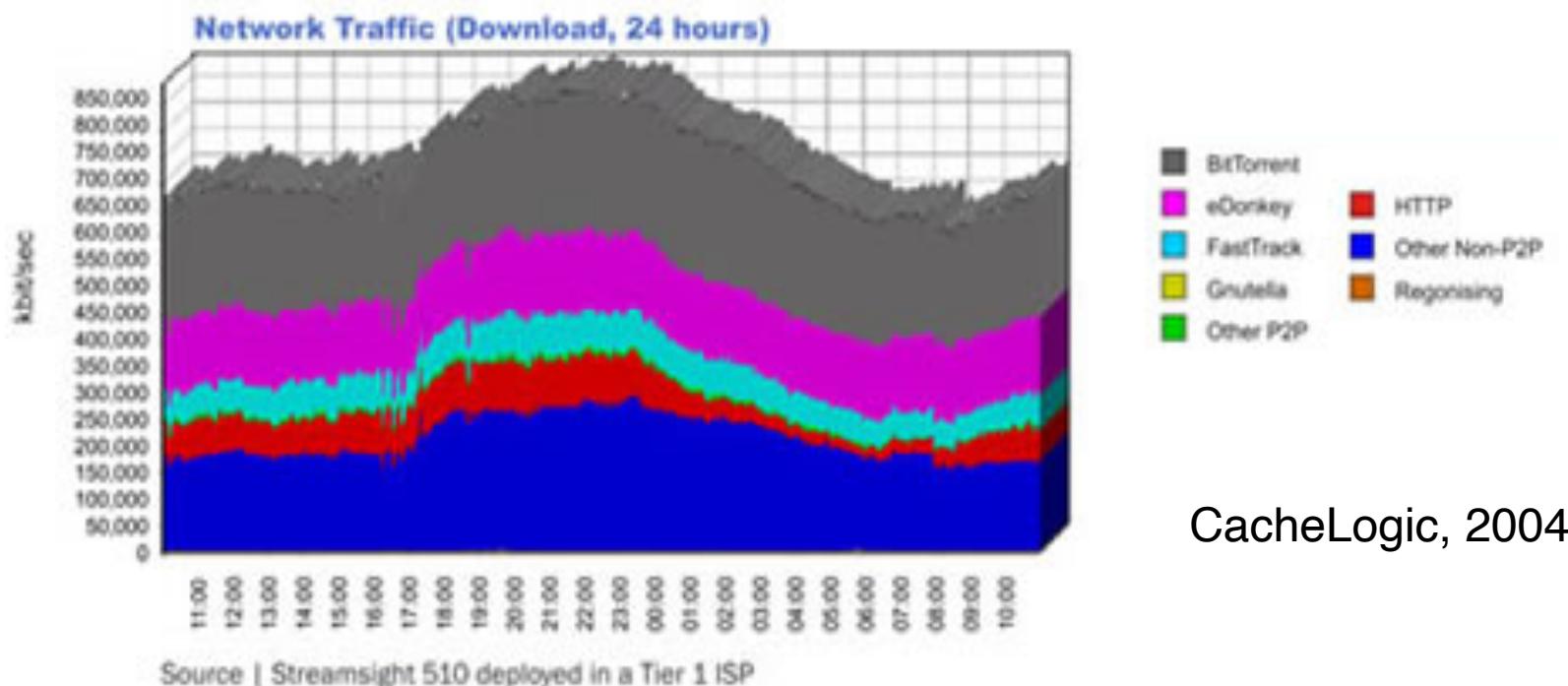
Use erasure coding if **storage** resources are limited (e.g., in-memory storage) or computational resources and communication is plentiful (e.g., HPC).

Other Types of Coding

- Checksum
 - Parity Bit
 - Fountain coding
 - Raptor code
 - Cauchy Reed-Solomon
 - Linear Network Coding
- Differences include:

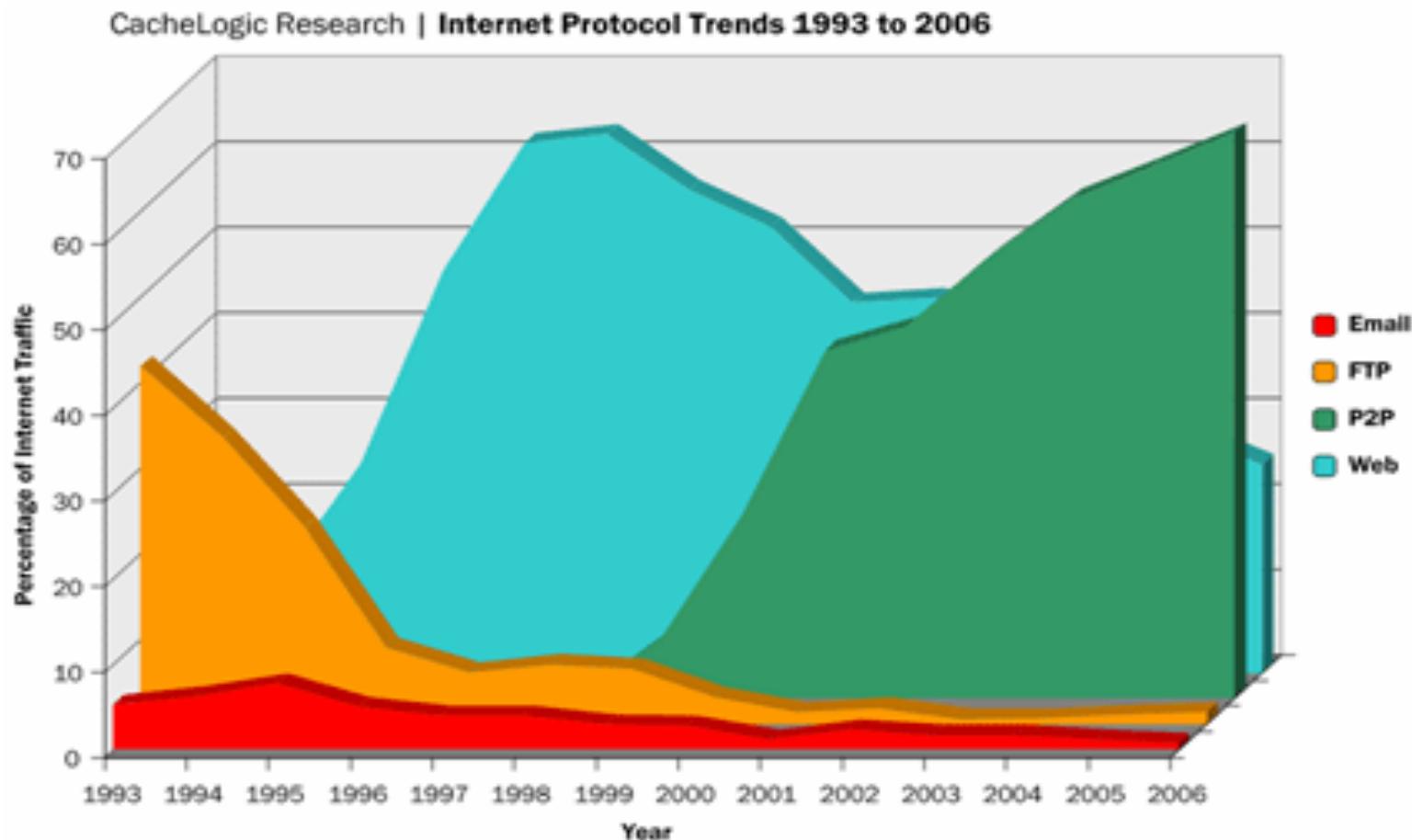
 - Error detection vs. correction
 - Encoding and decoding time
 - Optimal or near-optimal
 - Rateless
 - Maximizing bandwidth
-
- Custom combination possible
 - Reed Solomon to correct flipped bits + Fountain code to maximize bandwidth

Peer-to-Peer Applications



More data

(Source: CacheLogic)



Agenda

- Hybrid P2P applications
 - Spotify
 - BitTorrent
- Modern P2P technology: Blockchain
 - Bitcoin

Spotify - Large Scale, Low Latency, P2P Music-on-Demand Streaming

Gunnar Kreitz, Fredrik Niemelä
IEEE P2P'10

Following slides are adapted from authors'
slides at P2P in 2010 & 2011.

SPOTIFY

Spotify.com, 2004

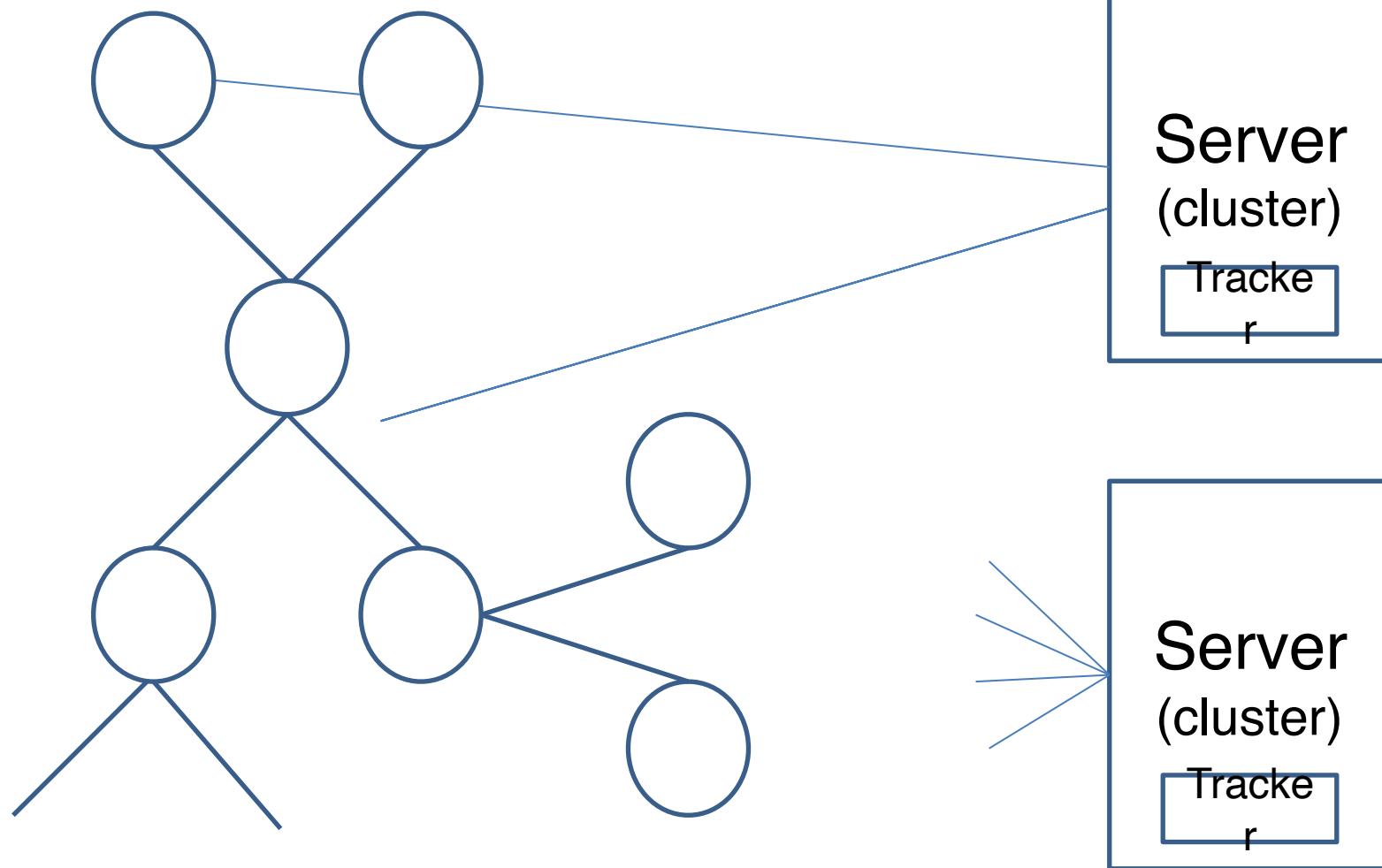


- Commercially deployed system, KTH start-up (Sweden)
- Peer-assisted on-demand music streaming
- Legal and licensed content only
- Large catalogue of music (over 15 million tracks)
- Available in U.S. & 7 European countries, over 10 million users, 1.6 million subscribers (in 2004)
- Fast (median playback latency of 265 ms)
- Proprietary client software for desktop & phone (not p2p)
- Business model: Ad-funded and free & monthly subscription, no ads, premium content, higher quality streaming

Overview of Spotify Protocol

- Proprietary protocol
- Designed for on-demand music streaming
- Only Spotify can add tracks
- 96–320 kbps audio streams (most are Ogg Vorbis q5, 160 kbps)
- Relatively simple and straightforward design
- Phased out in 2014: “*We’re now at a stage where we can power music delivery through **our** growing number of **servers** and ensure our users continue to receive a **best-in-class service**.*”
- Conclusion: *Commercially*, P2P technology is good for **startups** who demand more resources than their servers offer. Avoid “**death by success**”.

Spotify architecture: Peer-assisted



Why a Peer-to-peer Protocol?

Why a Peer-to-peer Protocol?

- **Improve scalability** of service

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources
- **Explicit design goal**

Why a Peer-to-peer Protocol?

- **Improve scalability** of service
- **Decrease load** on servers and network resources
- **Explicit design goal**
 - Use of peer-to-peer should not decrease overall performance (i.e., playback latency & stutter)

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
 - **Fast lookup required (Hybrid p2p)**
 - Let us do some rough estimates (ping times)

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
 - **Fast lookup required (Hybrid p2p)**
 - Let us do some rough estimates (ping times)
 - Latency UK – Netherlands ~ 10 ms and up

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
 - **Fast lookup required (Hybrid p2p)**
 - Let us do some rough estimates (ping times)
 - Latency UK – Netherlands ~ 10 ms and up
 - Latency across EU more like ~ 80 ms and up

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
 - **Fast lookup required (Hybrid p2p)**
 - Let us do some rough estimates (ping times)
 - Latency UK – Netherlands ~ 10 ms and up
 - Latency across EU more like ~ 80 ms and up
 - Latency US – Europe ~ 100 ms and up

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
 - **Fast lookup required (Hybrid p2p)**
 - Let us do some rough estimates (ping times)
 - Latency UK – Netherlands ~ 10 ms and up
 - Latency across EU more like ~ 80 ms and up
 - Latency US – Europe ~ 100 ms and up
 - Playback latency ~265 ms

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
 - **Fast lookup required (Hybrid p2p)**
 - Let us do some rough estimates (ping times)
 - Latency UK – Netherlands ~ 10 ms and up
 - Latency across EU more like ~ 80 ms and up
 - Latency US – Europe ~ 100 ms and up
 - Playback latency ~265 ms
 - <1% Playbacks have stuttering

Peer-to-peer Overlay Structure

- Unstructured overlay
- Does not use a DHT
 - **Fast lookup required (Hybrid p2p)**
 - Let us do some rough estimates (ping times)
 - Latency UK – Netherlands ~ 10 ms and up
 - Latency across EU more like ~ 80 ms and up
 - Latency US – Europe ~ 100 ms and up
 - Playback latency ~265 ms
 - <1% Playbacks have stuttering
 - **Simplicity of protocol design & implementation**

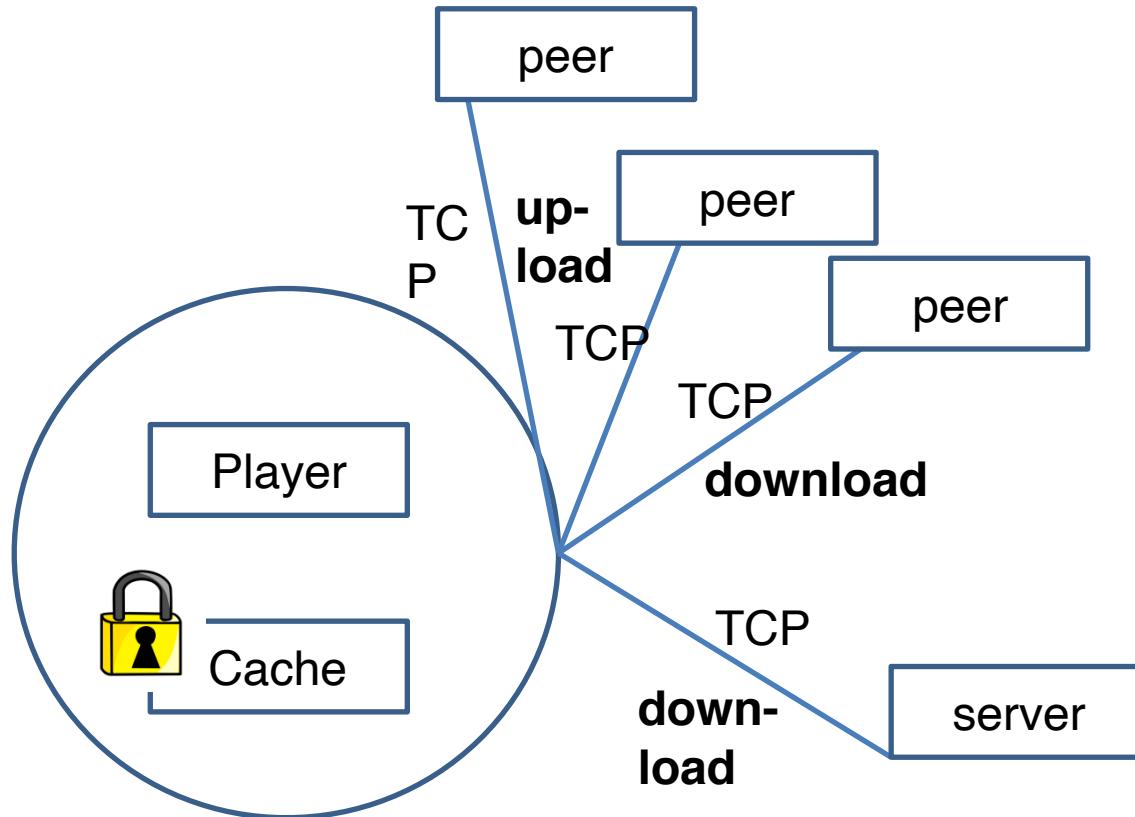
Peer-to-peer Overlay Structure

- Nodes have fixed maximum degree (60)
- Neighbour eviction by heuristic evaluation of utility
- Looks for and connects to new peers when streaming new track
- Overlay becomes (weakly) clustered by interest
- Client only downloads data user needs

Finding Peers

- Server-side tracker (cf. BitTorrent)
 - Only remembers 20 peers per track
 - Returns 10 (online) peers to client on query
- Clients broadcast query in small (2 hops) neighbourhood in overlay (cf. Gnutella)
- Client uses both mechanisms for every track

Peers



Protocol

- (Almost) everything encrypted
- (Almost) everything over TCP
- Persistent connection to server while logged in
- Multiplex messages over a single TCP connection

Caches

- Client (player) caches tracks it has played
- Default policy is to use 10% of free space (capped at 10 GB)
- Caches are often larger (56% are over 5 GB)
- Least Recently Used policy for cache eviction
- Over 50% of data comes from local cache
- Cached files are served in peer-to-peer overlay (if track completely downloaded)

Streaming a Track

- Tracks are decomposed into 16 kB chunks
- Request first piece of track from Spotify servers
- Meanwhile, search for peers that cache track
- Download data in-order (chunk by chunk via TCP)
- Send urgent pieces first (e.g. low buffer)
- Towards end of a track, start prefetching next track

Streaming a Track

- If a remote peer is slow, re-request data from new peers
- If local buffer is sufficiently filled, only download from peer-to-peer overlay
- If buffer is getting low, download from central server as well
 - Estimate at what point p2p download could resume
- If buffer is very low, stop uploading

Security Through Obscurity, ☹

- Music data lies encrypted in caches
- Client must be able to access music data
- Reverse engineers should not be able to access music data
- Details are secret and client code is obfuscated
- **Do not do this “at home”**
 - ***Security through obscurity*** is a bad idea
 - It is a matter of time until someone hacks the Spotify client (cf. the various Skype reverse engineering efforts)

Data sources: 8.8% from servers, 35.8% from p2p network, 55.4 % from caches

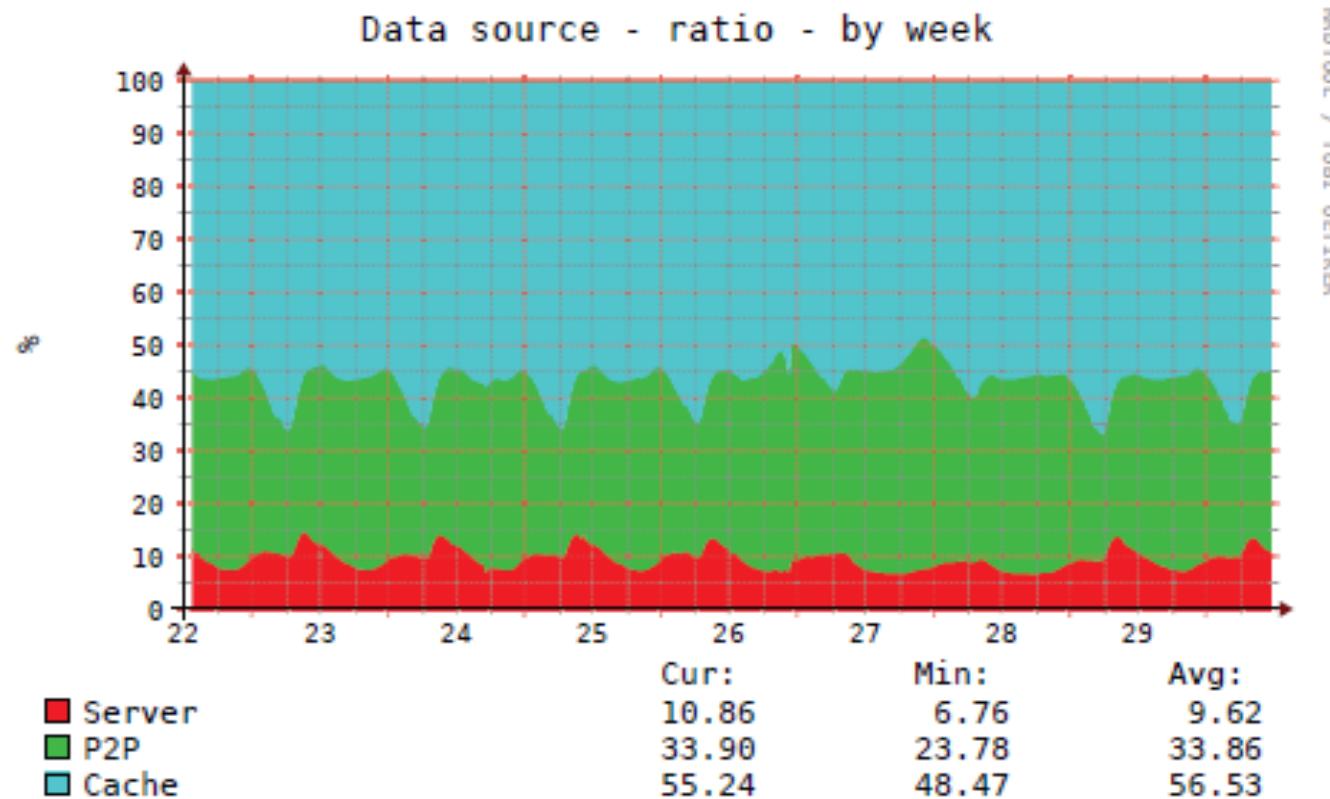
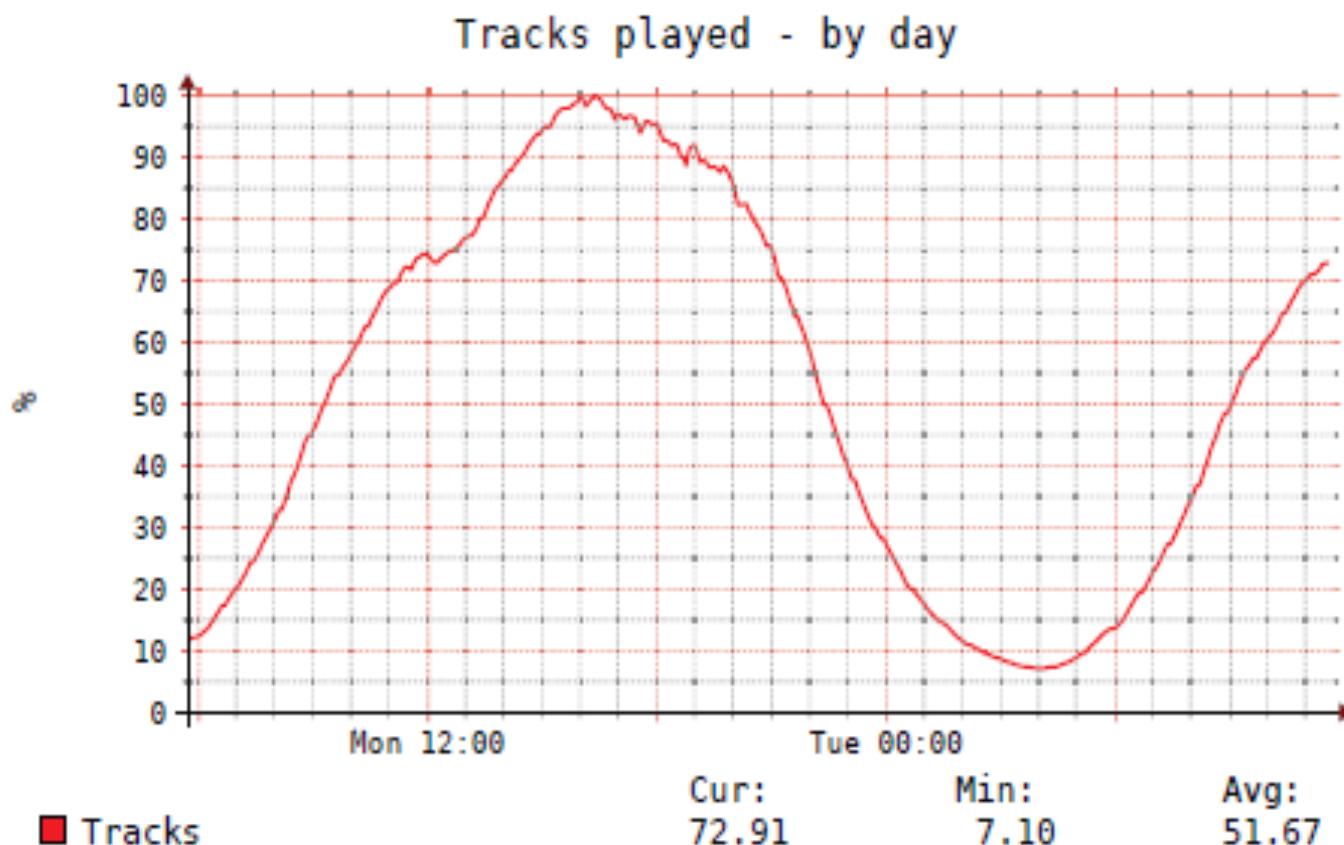


Figure 2. Sources of data used by clients

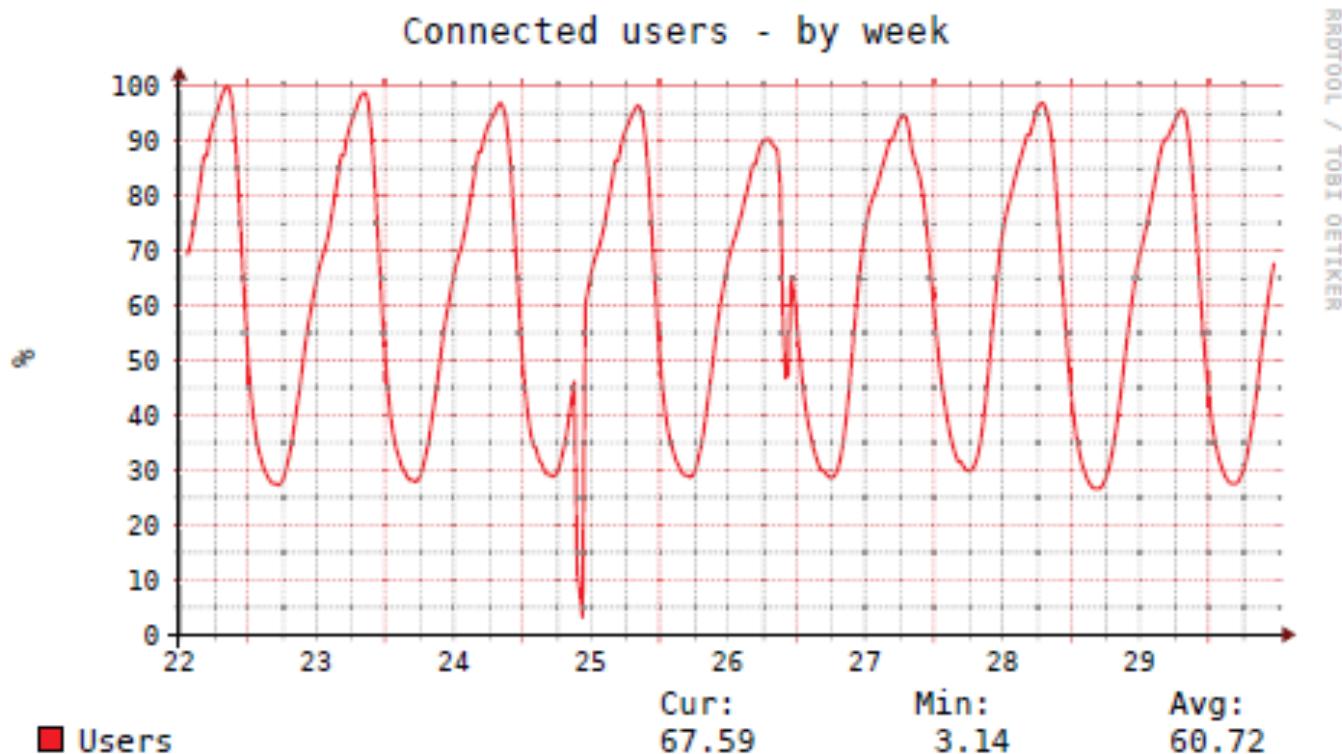
Tracks played

RRTOOL / TOBI OETIKER



(a) Tracks played

Users connected



(b) Users connected

Key Points

- Simplicity of architecture, protocol, design
- Peer-assisted, i.e., rely on centralized server
- Use of peer-to-peer techniques for scalability and avoid heavy, over-provisioned infrastructure
- Use of centralized tracker

Incentives build robustness in BitTorrent

by Bram Cohen

BitTorrent Protocol Specification
<http://www.bittorrent.org/protocol.html>



BitTorrent

- Written by Bram Cohen (in Python) in 2001
- Pull-based, swarming approach (segmented)
 - Each file is split into smaller pieces (& sub-pieces)
 - Peers request desired pieces from neighboring peers
 - Pieces are not downloaded in sequential order
- Encourages contribution by all peers
 - Based on a **tit-for-tat model**

BitTorrent Use cases

- File-sharing
- *What uses does BitTorrent support?*
 - Downloading (licensed only ☺) movies, music, etc.
 - *And ... ?*

BitTorrent Use cases

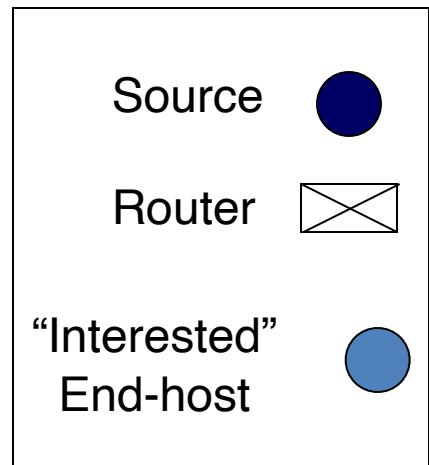
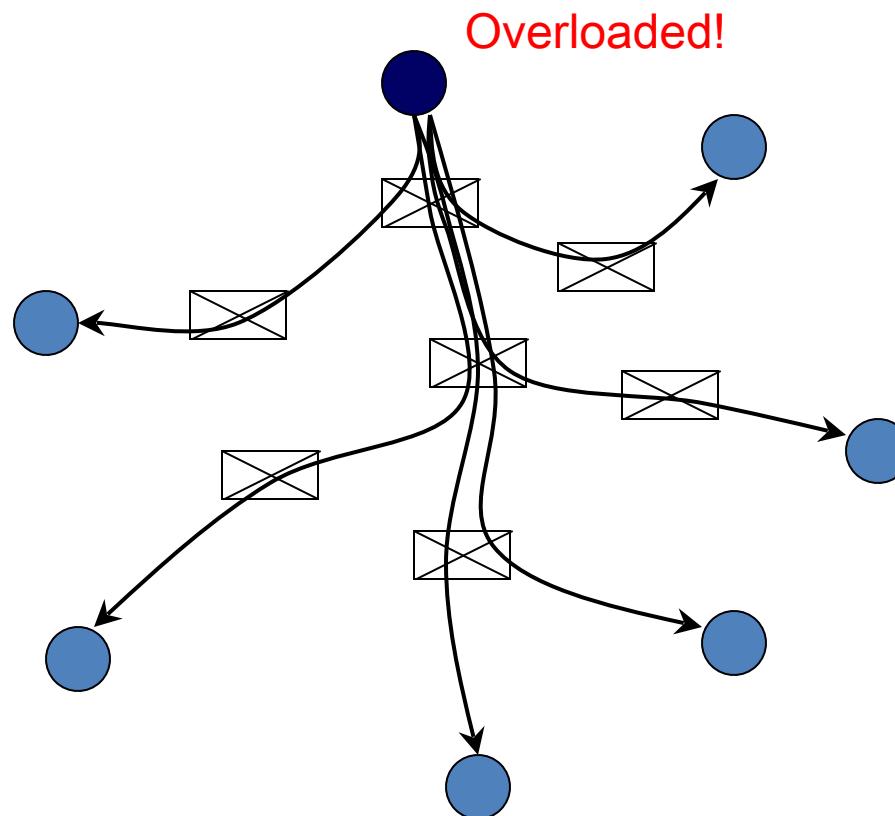
- File-sharing
- *What uses does BitTorrent support?*
 - Downloading (licensed only ☺) movies, music, etc.
 - *And ... ?*
 - Update distribution among servers at Facebook et al.
 - Distribution of updates and releases (e.g., World of Warcraft -> Blizzard Downloader)
 - ...



BitTorrent in numbers

- Nov.'2004, BitTorrent responsible for 35% of all Internet traffic
- Feb.'2013, BitTorrent responsible for 3.35% of all worldwide bandwidth
- Since 2010, more than 200,000 people have been sued for filesharing on BitTorrent
- 2013 study: Of 12,500 most popular torrent files, only two files were legally distributed on BitTorrent
- As of 2009, BitTorrent reportedly had about the same number of active users online as viewers of YouTube and Facebook combined
- As of Jan.'2012, BitTorrent is utilized by 150 million active users

Client-server



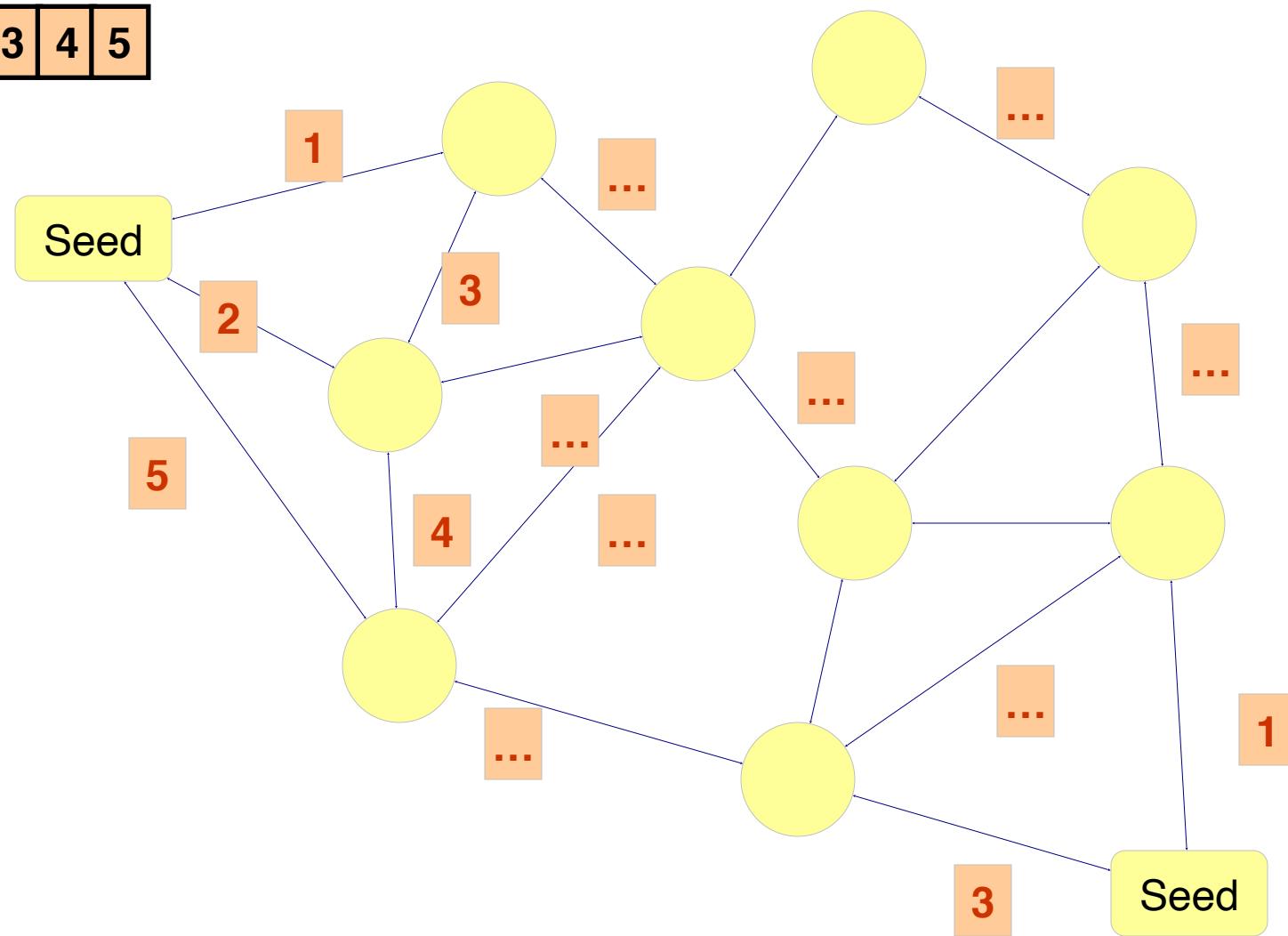
BitTorrent Swarm

- **Swarm**
 - Set of peers downloading the same file
 - Organized as a randomly connected mesh of peers
- Each peer knows list of pieces downloaded by neighboring peers
- Peer requests pieces it does not own from neighbors

BitTorrent Terminology

- **Seed**: Peer with the entire file
 - **Original Seed**: First seed for a file
- **Leech**: Peer downloading the file
 - Leech becomes a seed, once file downloaded, if the peer stays online & continues by protocol
- **Sub-piece**: Further subdivision of a piece
 - “Unit for requests” is a sub-piece (16 kB)
 - Peer uploads piece only after assembling it completely

BitTorrent Swarm



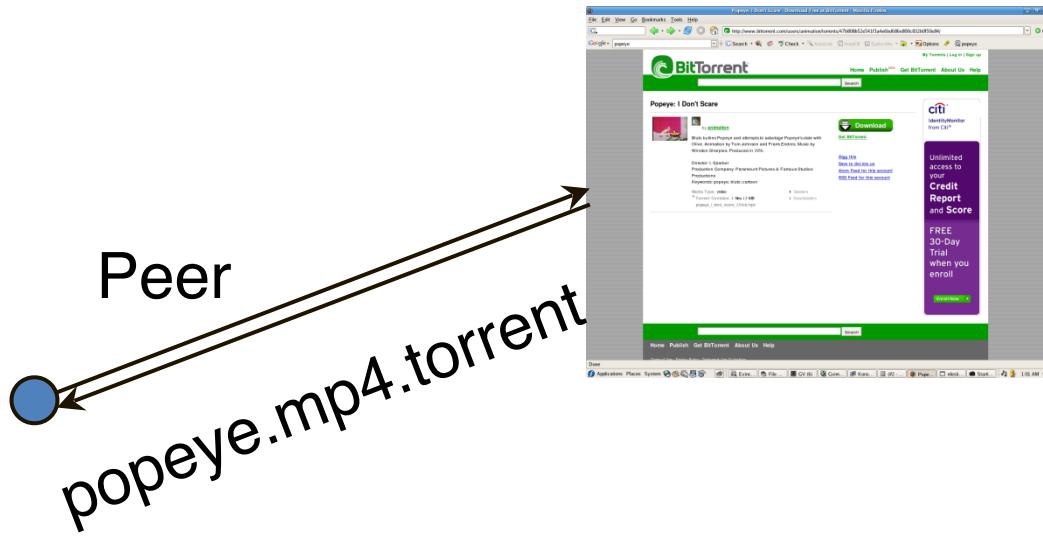
* From Analyzing and Improving BitTorrent by Ashwin R. Bharambe, Cormac Herley and Venkat Padmanabhan

Entering a Swarm for file “popeye.mp4”

- File **popeye.mp4.torrent** hosted at a (well-known) web server
- The **.torrent** has address of **tracker** for file
- The tracker, which runs on a web server as well, keeps track of all peers downloading file

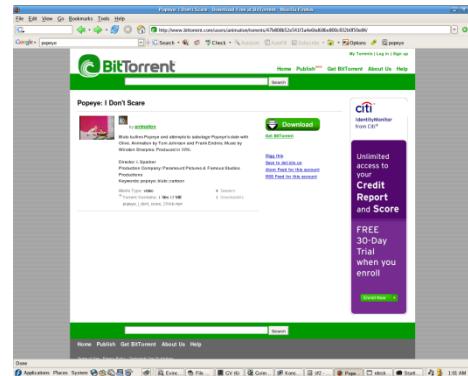
Find the Torrent for Desired Content

www.bittorrent.com or
anywhere



Contact the Tracker of Retrieved Torrent

www.bittorrent.com



Peer

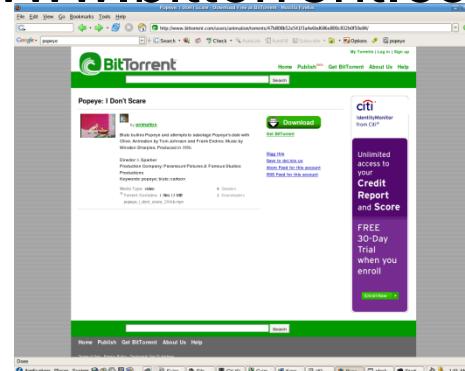


Addresses of peers

Tracker

Connect to Available Peers

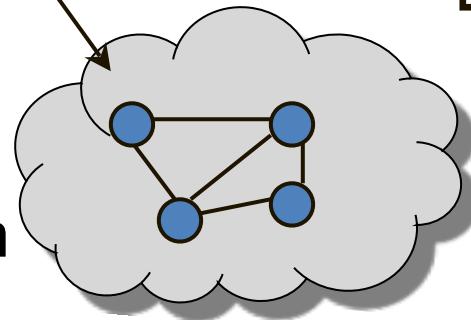
www.bittorrent.com



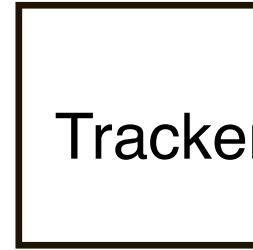
Peer



Swarm



Tracker



Contents of .torrent File

- URL of tracker
- Piece length – usually 256 KB
- SHA-1 hashes of each piece in file - ***Why?***

Download Phases

- **Beginning:** First piece
- **Middle:** Piece 2 to $n-x$
- **End-game:** Pieces $n-x$ to n

Piece (256 KB)



Middle: Choosing Pieces to Request

- *What is a good strategy?*
 - Most frequent first vs. rarest first?

Middle: Choosing Pieces to Request

- *What is a good strategy?*
 - Most frequent first vs. rarest first?
- **Rarest-first:** Look at all pieces at all neighboring peers and request a piece that's owned by the fewest peers – *Why?*

Middle: Choosing Pieces to Request

- *What is a good strategy?*
 - Most frequent first vs. rarest first?
- **Rarest-first:** Look at all pieces at all neighboring peers and request a piece that's owned by the fewest peers – **Why?**
 - **Increases diversity** in pieces downloaded
 - Avoids case where a node and each of its peers have exactly the same pieces
 - Increases system-wide throughput
 - Increases **likelihood all pieces still available** even if original seed leaves before any one node has downloaded entire file

Choosing Pieces to Request: Initial Piece

- First, pick a random piece (**random first policy**)
 - When peer starts to download, request random piece from a peer
 - When first complete piece assembled, **switch to rarest-first**

Choosing Pieces to Request: Initial Piece

- First, pick a random piece (**random first policy**)
 - When peer starts to download, request random piece from a peer
 - When first complete piece assembled, **switch to rarest-first**
 - Get first piece to quickly participate in swarm with upload
 - Rare pieces are only available at few peers (slows download down)
- **Strict priority policy**
 - Always **complete download of piece** (sub-pieces) before starting a new piece
 - Getting a complete piece as quickly as possible

Choosing Pieces to Request: Final Pieces

End-game mode

- Coupon's collector problem
- Send requests for the final piece to **all peers**
- Upon download of entire piece, cancel request for downloading that piece from other peers
- Speeds up completion of download, otherwise last piece could delay completion of download
- ***Why isn't this done all along?***

Why BitTorrent took off

- Working implementation (Bram Cohen) with simple well-defined interfaces for publishing content
- Open specification
- Many competitors got sued & shut down (Napster, KaZaA)
- Simple approach
 - Doesn't do “search” per se
 - Users use well-known, trusted sources to locate content

Pros & Cons of BitTorrent

- Proficient in utilizing partially downloaded files
- Discourages “freeloading”
 - By rewarding fastest uploaders
- Encourages diversity through “rarest-first”
 - Extends lifetime of swarm
- Works well for “hot content”

Pros & Cons of BitTorrent

- Assumes all interested peers active at same time
- Performance deteriorates if swarm “cools off”
- Too much overhead to disseminate small files

Pros & Cons of BitTorrent

- Dependence on centralized trackers
- *Is this good or bad?*

Pros & Cons of BitTorrent

- Dependence on centralized trackers
- *Is this good or bad?*
 - **Single point of failure**
 - New nodes can't enter swarm if tracker goes down
 - Simple to design, implement, deploy

Pros & Cons of BitTorrent

- Dependence on centralized trackers
- *Is this good or bad?*
 - **Single point of failure**
 - New nodes can't enter swarm if tracker goes down
 - Simple to design, implement, deploy
 - **Lack of a search feature**
 - Users need to resort to out-of-band search: Well known torrent-hosting sites, plain old web-search

Spotify vs. BitTorrent

- Live listening of streamed track
- One peer-to-peer overlay for all tracks
- Does not inform peers about downloaded blocks
- Downloads blocks in order
- Does not enforce fairness
- Informs peers about urgency of request
- Supports search
- Batch download of large files
- Essentially, a swarm (overlay) per torrent
- Exchange of downloaded blocks with peers
- Random download order
- Tit-for-tat (game theoretic roots)
- No notion of urgency
- No search feature

Peer-to-Peer Systems



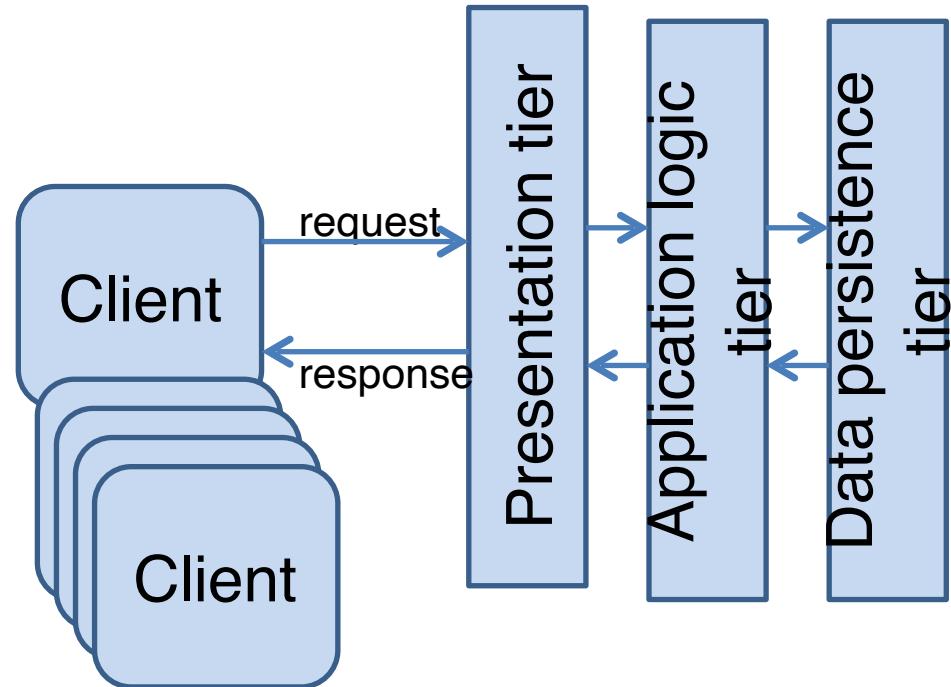
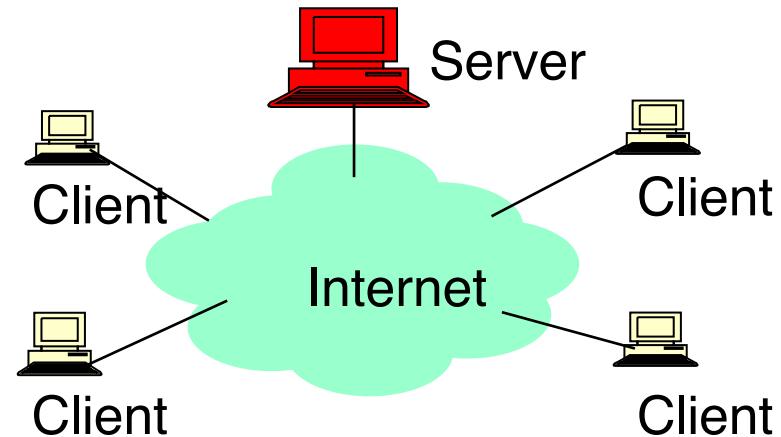
Agenda

- Peer-to-peer networking
 - Definition
 - Use cases
- Peer-to-peer overlays
 - Unstructured systems
 - Distributed hash table abstraction
 - Structured systems realizing DHTs

PEER-TO-PEER NETWORKING

Client-Server Architecture

- Well known, powerful, reliable server as data source
- Clients request data from server
- Very successful model
 - WWW (HTTP), FTP, Web services, etc.
- N-tier architecture



Client-Server Limitations

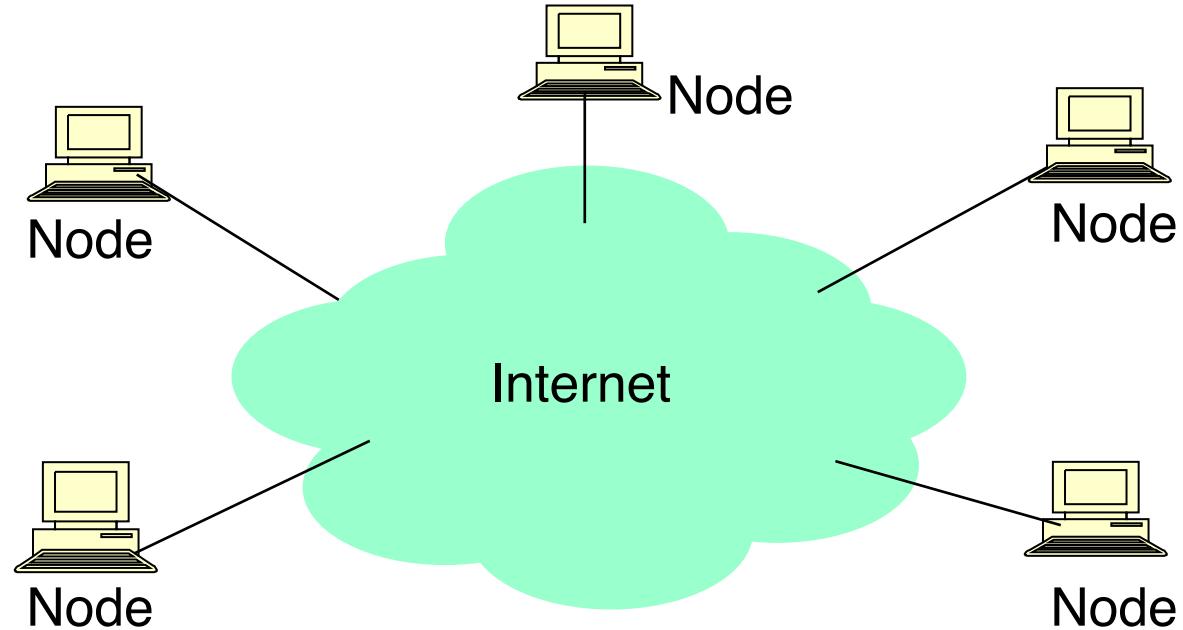
- Scalability is expensive (vertical vs. horizontal)
- Presents a single point of failure
- Requires administration
- Unused resources at network edge
- P2P systems try to address these limitations and leverage (otherwise) unused resources

P2P Computing

- P2P computing is the sharing of **computer resources** and **services** by **direct** exchange between nodes
- These resources and services include the **exchange of data, processing cycles, cache storage, and disk storage for files** (bandwidth, CPU, storage)
- P2P computing takes advantage of existing computing power, computer storage and networking connectivity, allowing users to leverage their **collective power to the ‘benefit’ of all**

* From (accessed June, 2004) http://www-sop.inria.fr/mistral/personnel/Robin.Groeneveld/Publications/Peer-to-Peer_Introduction_Feb.ppt

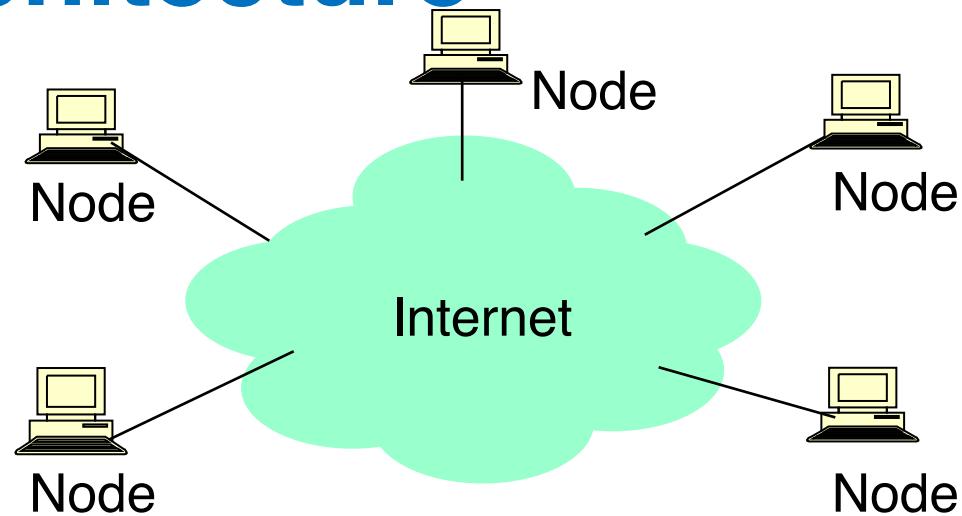
What is a P2P system?



- A distributed system architecture
 - No centralized control
 - Nodes are symmetric in function
- Larger number of unreliable nodes
- Enabled by technology improvements

P2P Architecture

- All nodes are both *clients* and *servers*
 - Provide and consume
 - Any node can initiate a connection
- No centralized data source
 - “The ultimate form of democracy on the Internet”
 - “The ultimate threat to copyright protection on the Internet”



- In practice, **hybrid models** are popular
 - Combination of client-server & peer-to-peer
 - Skype (early days, now unknown)
 - Spotify (peer-assisted)
 - BitTorrent (trackers)

P2P Benefits I

- Efficient use of resources
 - Unused bandwidth, storage, processing power at the edge of network
- Scalability
 - Consumers of resources also donate resources
 - Aggregate resources grow naturally with utilization
 - Organic scaling (more users, more resources)
 - “Infrastructure-less” scaling
- **Caveat:** It is not a one size fits all
 - Large companies are not switching in droves to P2P

P2P Benefits II

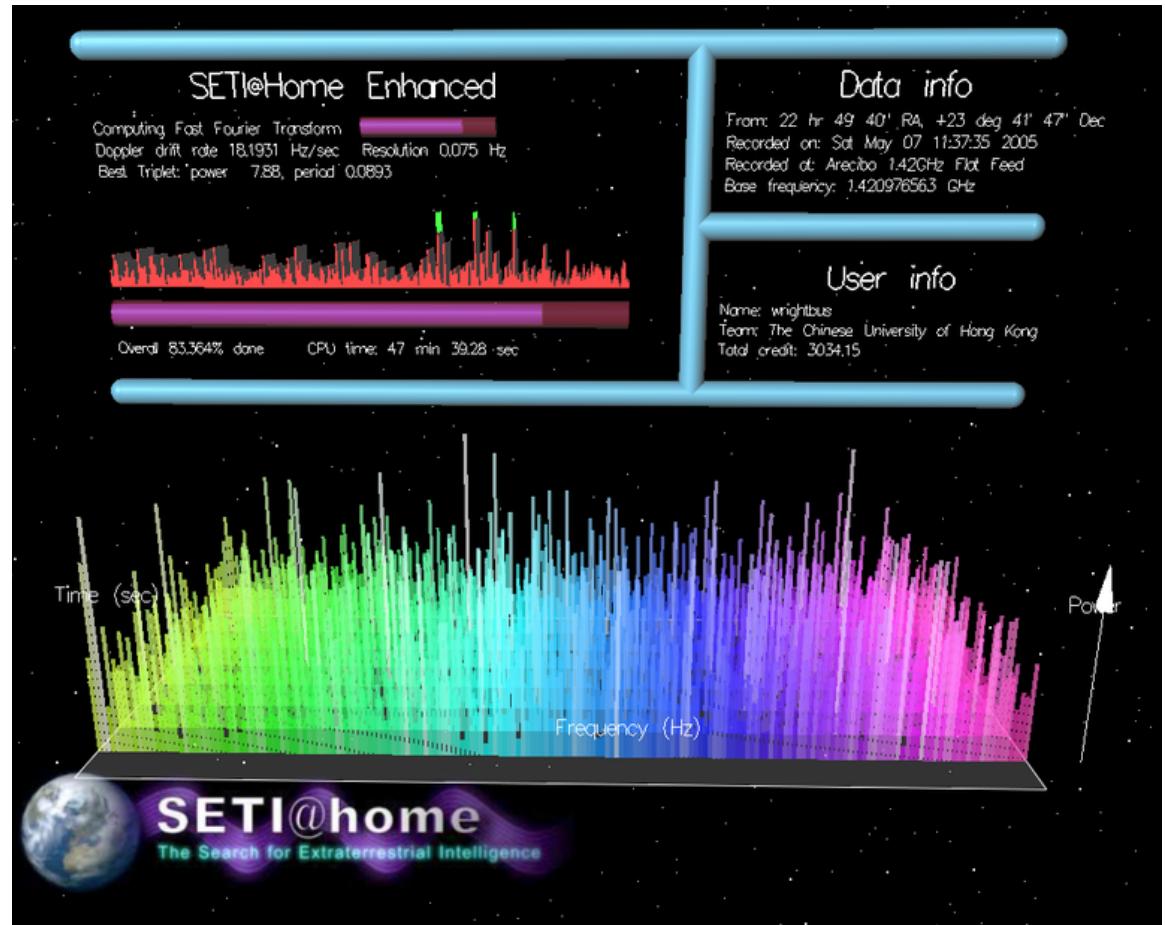
- Reliability (in aggregate)
 - Many replicas, redundancy
 - Geographic distribution
 - No single point of failure and control
- Ease of administration
 - Nodes self-organize
 - No need to deploy servers to satisfy demand
 - Built-in fault-tolerance, replication, and load balancing

Use Cases: Large-Scale Systems

- Some applications require immense resources
 - CPU: Scientific data analysis (*@home)
 - Bandwidth: Streaming, file sharing
 - Storage: Decentralized data, file sharing
- Thousands or even millions of nodes
 - How to efficiently manage such a large network?
 - One solution: P2P

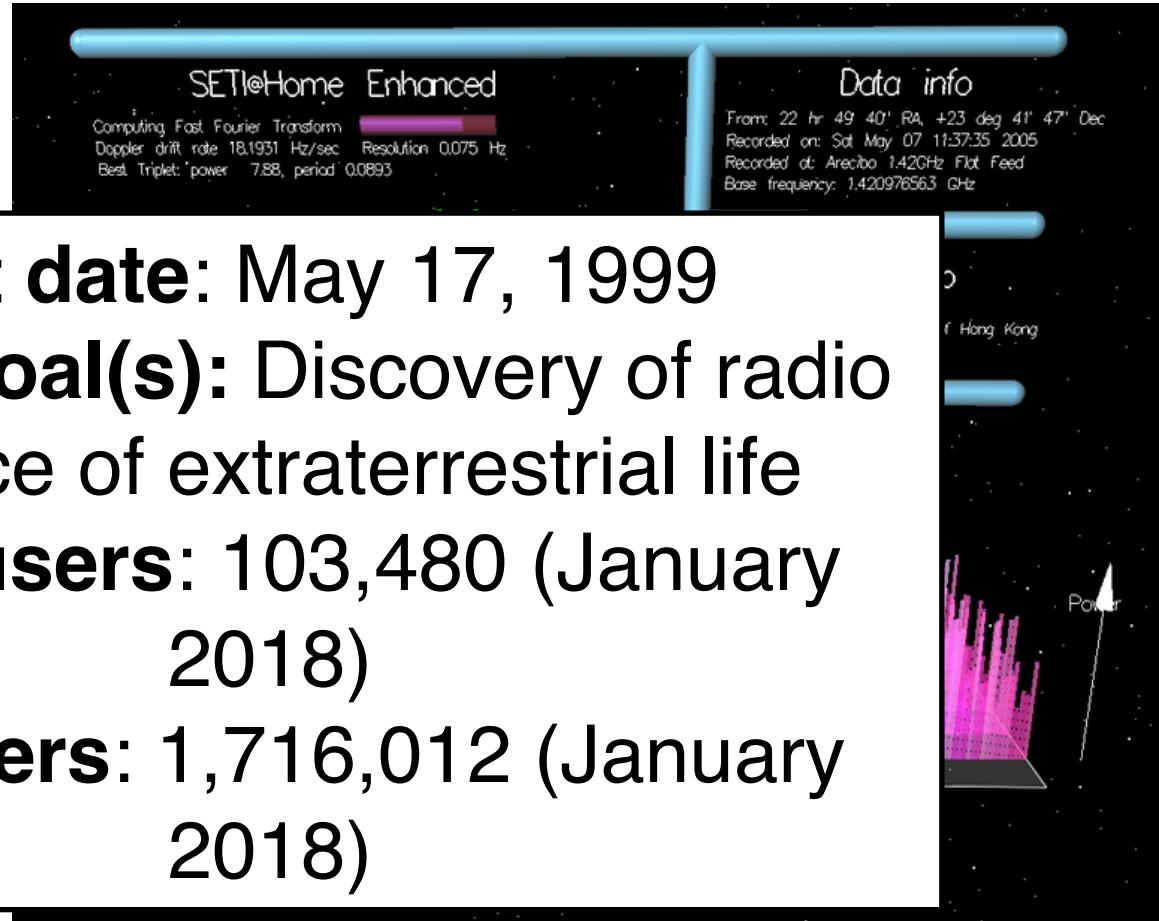
SETI@home – started in 1999

- 5.2 million participants worldwide
- On September 26, 2001, had performed a total of 10^{21} flops
- 35 GB/data per day, 140K work units
- 30 hours/WU
- 4.2M hours of computation/day
- Centralized database



SETI@home – started in 1999

- 5.2 million participants worldwide
- On Sept 17, 2001, SETI@home performed its first search of 10 hours.
- 35 GHz of computing power per day, from 100,000 units
- 30 hours of processing time
- 4.2M CPU hours per day
- Centralized database



P2P OVERLAYS & FILE SHARING

2015 NA Traffic

Traffic share in North America during peak hours

Rank	Upstream		Downstream		Aggregate	
	Application	Share	Application	Share	Application	Share
1	BitTorrent	28.56%	Netflix	37.05%	Netflix	34.70%
2	Netflix	6.78%	YouTube	17.85%	YouTube	16.88%
3	HTTP	5.93%	HTTP	6.06%	HTTP	6.05%
4	Google Cloud	5.30%	Amazon Video	3.11%	BitTorrent	4.35%
5	YouTube	5.21%	iTunes	2.79%	Amazon Video	2.94%
6	SSL - OTHER	5.10%	BitTorrent	2.67%	iTunes	2.62%
7	iCloud	3.08%	Hulu	2.58%	Facebook	2.51%
8	FaceTime	2.55%	Facebook	2.53%	Hulu	2.48%
9	Facebook	2.25%	MPEG - OTHER	2.30%	MPEG	2.16%
10	Dropbox	1.18%	SSL - OTHER	1.73%	SSL - OTHER	1.99%
		65.95%		78.69%		76.68%

 sandvine®

<https://www.sandvine.com>

2015 NA Traffic

Still #1 in upstream!

Traffic share in North America during peak hours

Rank	Upstream		Downstream		Aggregate	
	Application	Share	Application	Share	Application	Share
1	BitTorrent	28.56%	Netflix	37.05%	Netflix	34.70%
2	Netflix	6.78%	YouTube	17.85%	YouTube	16.88%
3	HTTP	5.93%	HTTP	6.06%	HTTP	6.05%
4	Google Cloud	5.30%	Amazon Video	3.11%	BitTorrent	4.35%
5	YouTube	5.21%	iTunes	2.79%	Amazon Video	2.94%
6	SSL - OTHER	5.10%	BitTorrent	2.67%	iTunes	2.62%
7	iCloud	3.08%	Hulu	2.58%	Facebook	2.51%
8	FaceTime	2.55%	Facebook	2.53%	Hulu	2.48%
9	Facebook	2.25%	MPEG - OTHER	2.30%	MPEG	2.16%
10	Dropbox	1.18%	SSL - OTHER	1.73%	SSL - OTHER	1.99%
		65.95%		78.69%		76.68%



<https://www.sandvine.com>

2015 NA Traffic

Still #1 in upstream!

Traffic share in North America during peak hours

But streaming is larger overall...

Rank	Upstream		Downstream		Aggregate	
	Application	Share	Application	Share	Application	Share
1	BitTorrent	28.56%	Netflix	37.05%	Netflix	34.70%
2	Netflix	6.78%	YouTube	17.85%	YouTube	16.88%
3	HTTP	5.93%	HTTP	6.06%	HTTP	6.05%
4	Google Cloud	5.30%	Amazon Video	3.11%	BitTorrent	4.35%
5	YouTube	5.21%	iTunes	2.79%	Amazon Video	2.94%
6	SSL - OTHER	5.10%	BitTorrent	2.67%	iTunes	2.62%
7	iCloud	3.08%	Hulu	2.58%	Facebook	2.51%
8	FaceTime	2.55%	Facebook	2.53%	Hulu	2.48%
9	Facebook	2.25%	MPEG - OTHER	2.30%	MPEG	2.16%
10	Dropbox	1.18%	SSL - OTHER	1.73%	SSL - OTHER	1.99%
		65.95%		78.69%		76.68%



<https://www.sandvine.com>

P2P File Sharing Systems

- Large-scale sharing of files
 - User *A* makes files (music, video, etc.) on their computer available to others
 - User *B* connects to “network,” searches for files and downloads files *directly* from User *A*
- P2P networks
 - Peers are connected to each other to form an **overlay network**
 - Peers communicate using links established in overlay
- Fallen out of favor (*RIP 1999-2015*)
 - Issues of copyright infringement
 - Cloud infrastructures has taken over (controlled resources)
 - Harder to exploit mobile and connected devices
 - Streaming makes file sharing obsolete (cf. P2P Streaming)

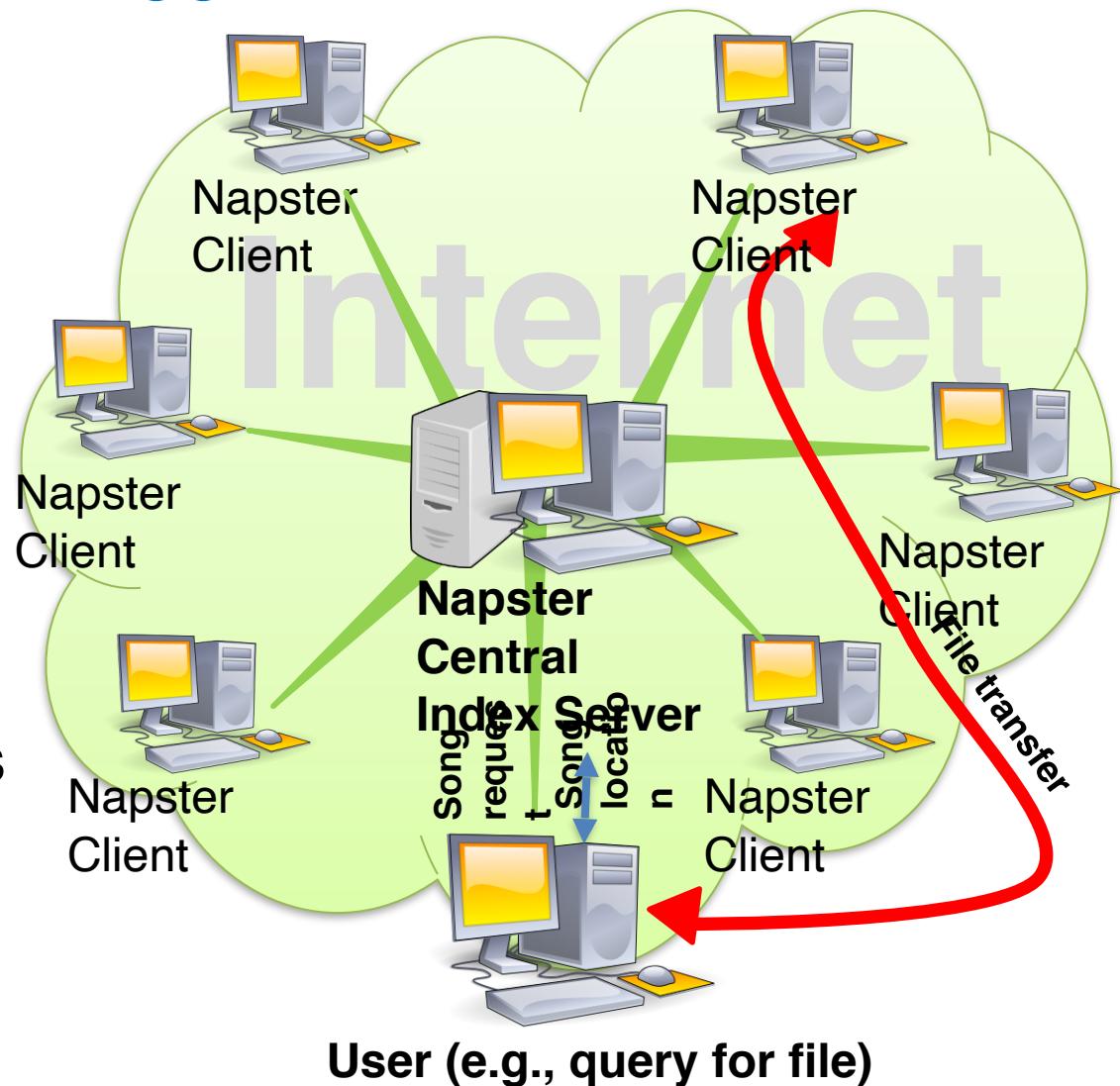
Types of P2P Networks

- Unstructured networks
 - No obvious structure in overlay topology
 - Peers simply connect to anyone in existing network
- First generation:
 - Centralized: **Napster**
 - Pure: **Gnutella**, Freenet
- Second generation:
 - Dynamic “supernodes”
 - Hybrid: **Skype**, **Spotify**, FastTrack, eDonkey, **BitTorrent**
- Structured networks
 - Topology of overlay is controlled: peers’ connections are fixed
 - Based on the distributed hash table abstraction (**DHT**)

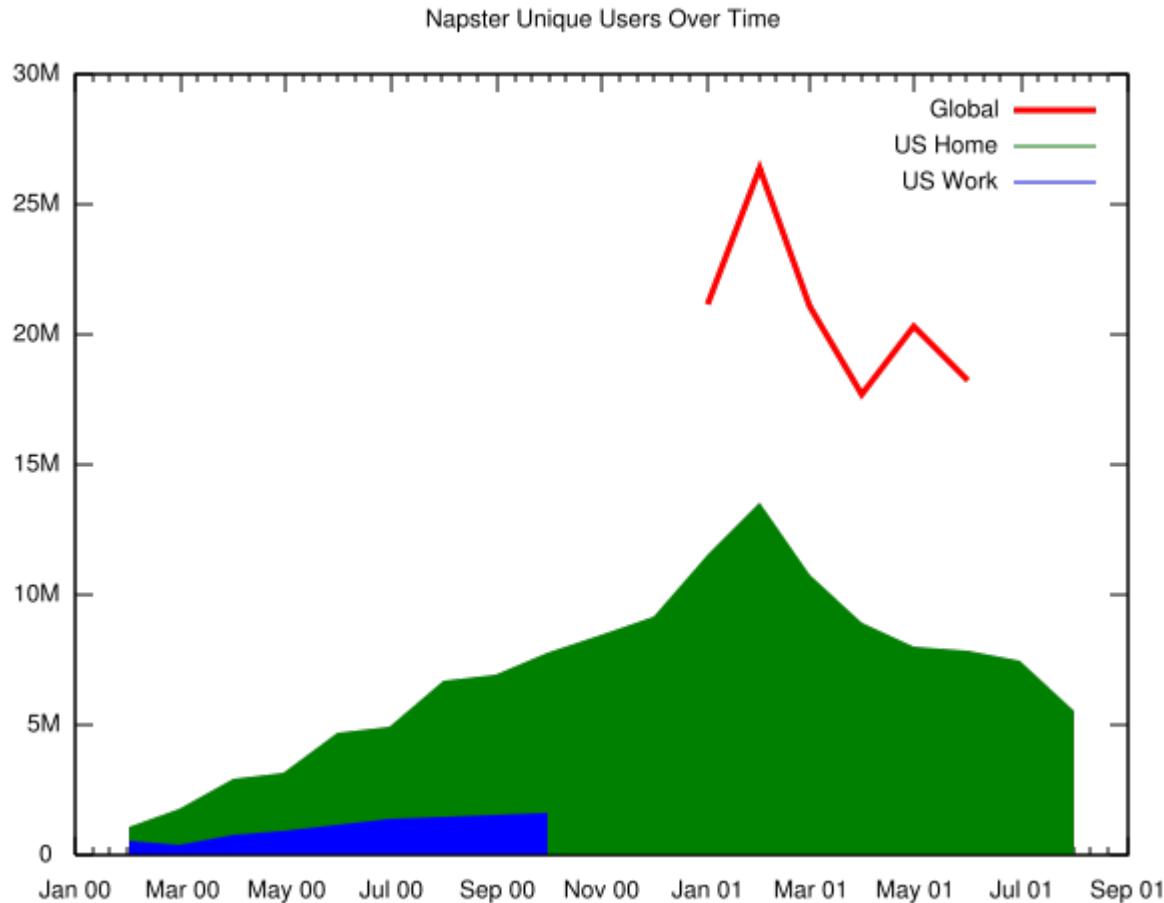
UNSTRUCTURED P2P SYSTEMS

Centralized: Napster “June 1999 – July 2001”

- Centralized search indexes music files
 - Perfect knowledge
 - Bottleneck
- Users query server
 - Keyword search (artist, song, album, bit rate, etc.)
- Napster server replies with IP address of users with matching files
- Querying users connect **directly** to remote node for file to download



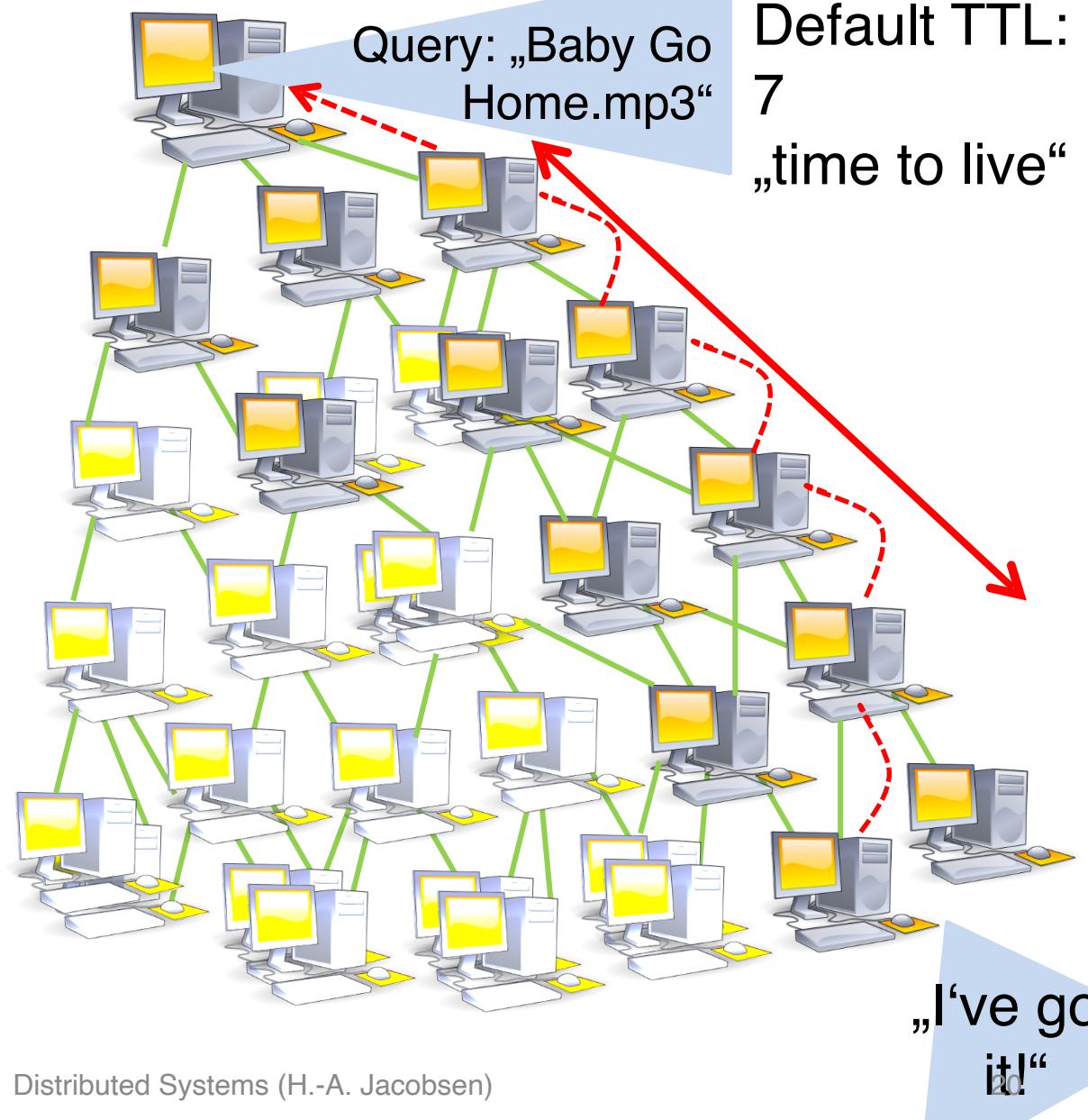
The Rise and Fall of Napster



RIAA shut down Napster easily because it used a central server!

Pure P2P: Gnutella 0.4 (2000 – 2001)

- Share any type of files
- Decentralized search
 - **Imperfect** content availability
- Client connect to (on average) 3 peers
- **Flood a QUERY** to connected peers
- Flooding propagates in network up to TTL
- Users with matching files **reply with QUERYHIT**
- Flooding wastes **bandwidth**: Later versions used more sophisticated search



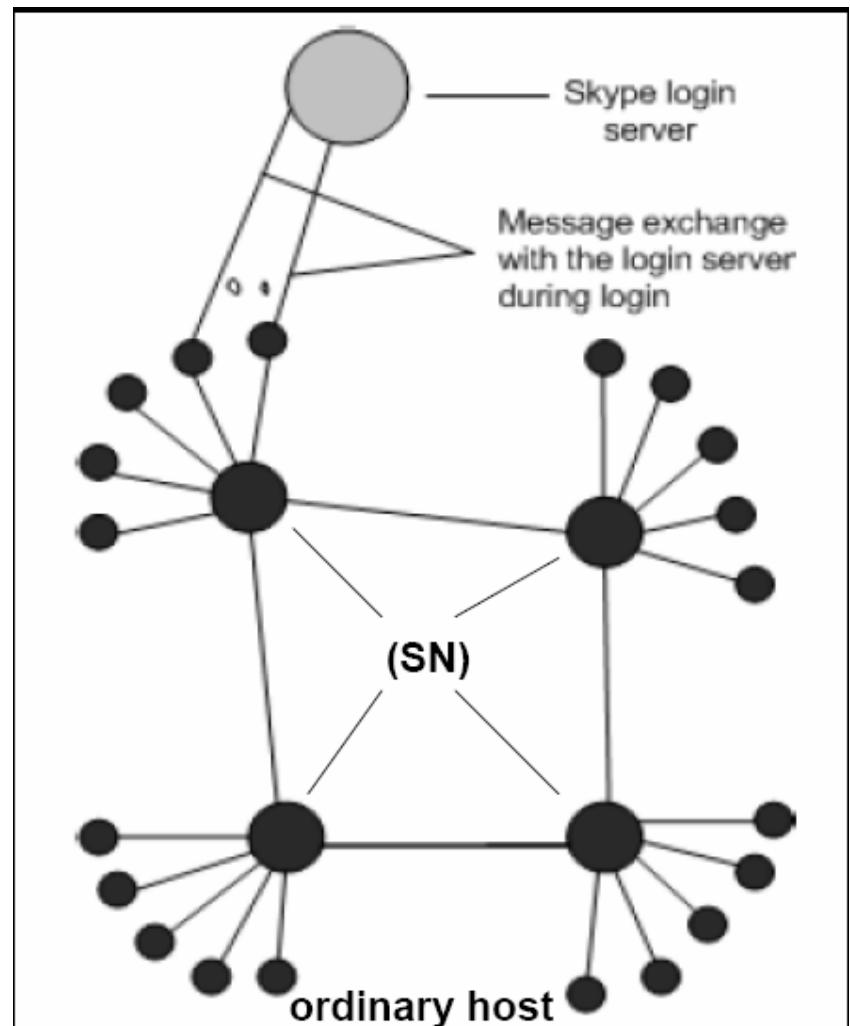
Hybrid P2P

- Both previous approaches have advantages and disadvantages
 - Centralized: single point of failure, easy to control, but perfect content availability
 - Pure: decentralized (resistant), costly and unreliable search
- Hybrid P2P combines both approaches
 - Hierarchy of peers
 - Superpeers with more capacity, discovered **dynamically**
 - Normal peers (leaf nodes) are users
- Superpeer responsibilities
 - Participates in search protocol, indexes and caches data
 - Improves content availability
 - Reduces message load

Skype Network

Around 2004

- **Super Nodes:** Any node with a public IP address having sufficient CPU, memory and network bandwidth is candidate to become a superpeer
- **Ordinary Host:** Host needs to connect to superpeer and must register itself with the Skype login server
- Login server and PSTN gateway (not shown) **are centralized components**



Responsibilities of Superpeers

In Skype Around 2004

- Indexes user directory
 - Distributed among superpeers
 - Communication among superpeers for lookup
- Communication relay
 - NAT traversal
- Phased out by Microsoft in 2011 (speculation)
 - Replaced with private servers
 - “[T]hat is in part why Skype has switched to server-based “dedicated supernodes”... nodes that **we control**, can handle orders of magnitudes **more clients per host**, are in **protected data centers** and up all the time, and **running code that is less complex** than the entire client code base. ”

Skype Impact

- Skype has shown, at least has suggested,:
 - **Signaling**, unique property of traditional phone system, is accomplished effortlessly with self-organizing P2P networks
 - **P2P overlay networks can scale** up to handle large-scale connection-oriented real-time services such as video and voice
- AT&T: “*The end of landlines* .



DISTRIBUTED HASH TABLE

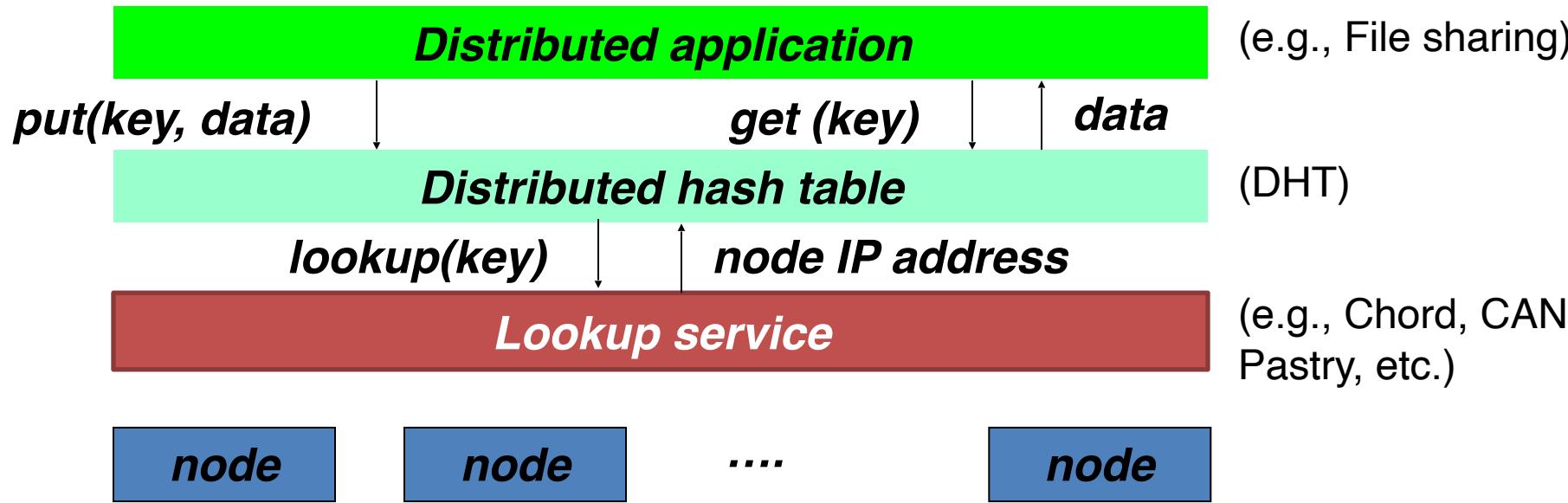
Structured Peer-to-Peer Systems

- Third generation P2P overlay networks
- Self-organizing, load balanced, fault-tolerant
- **Fast and efficient lookup guarantees**
 - $O(\log(n))$ lookup
 - $O(1)$: centralized, $O(n)$: pure, $O(\#S^*n)$: hybrid
- Based on a hash table interface (cf. KV-Store)
 - Put(Key, Data)
 - Get(Key)
- Systems: Chord, CAN, Pastry, etc.

Distributed Hash Tables (DHT)

- Distributed version of a hash table data structure
- Store and retrieve (key, value)-pairs
 - **Key** is like a filename, hash of name, hash of content (since name could change)
 - **Value** is file content
- Keys are hashed and mapped to nodes
 - Commonly uses consistent hashing

DHT Abstraction



- Application distributed over many nodes
- DHT distributes data storage over many nodes

DHT Interface

- Put(key, value) and get(key) → value
 - Simple interface!
- API supports a wide range of applications
 - DHT imposes neither structure nor meaning on keys
- Key-value pairs are persisted and globally available
 - Good availability, content stored at edge
 - Store keys in other DHT values
 - Thus, build complex data structures

DHT as Infrastructure or Service

- Many applications can share single DHT service
- Eases deployment of new applications
- Pools resources from many participants (P2P...)
- Essentially, a middleware service, a piece of distributed systems infrastructure

DHT-based Projects

- File sharing [CFS, OceanStore, PAST, Ivy, ...]
- Web cache [Squirrel, ..]
- Archival/Backup store [HiveNet, Mojo, Pastiche]
- Censor-resistant stores [Eternity,..]
- DB query and indexing [PIER, ...]
- Event notification (Publish/Subscribe) [Scribe, ToPSS]
- Naming systems [ChordDNS, Twine, ..]
- Communication primitives [I3, ...]
- Key-value stores [Cassandra*, Dynamo*, ...]

Common denominator:

- **Data is location-independent**
- **All leverage DHT abstraction**

* In as far as they use consistent hashing among nodes

STRUCTURED P2P SYSTEMS

DHT Desirable Properties

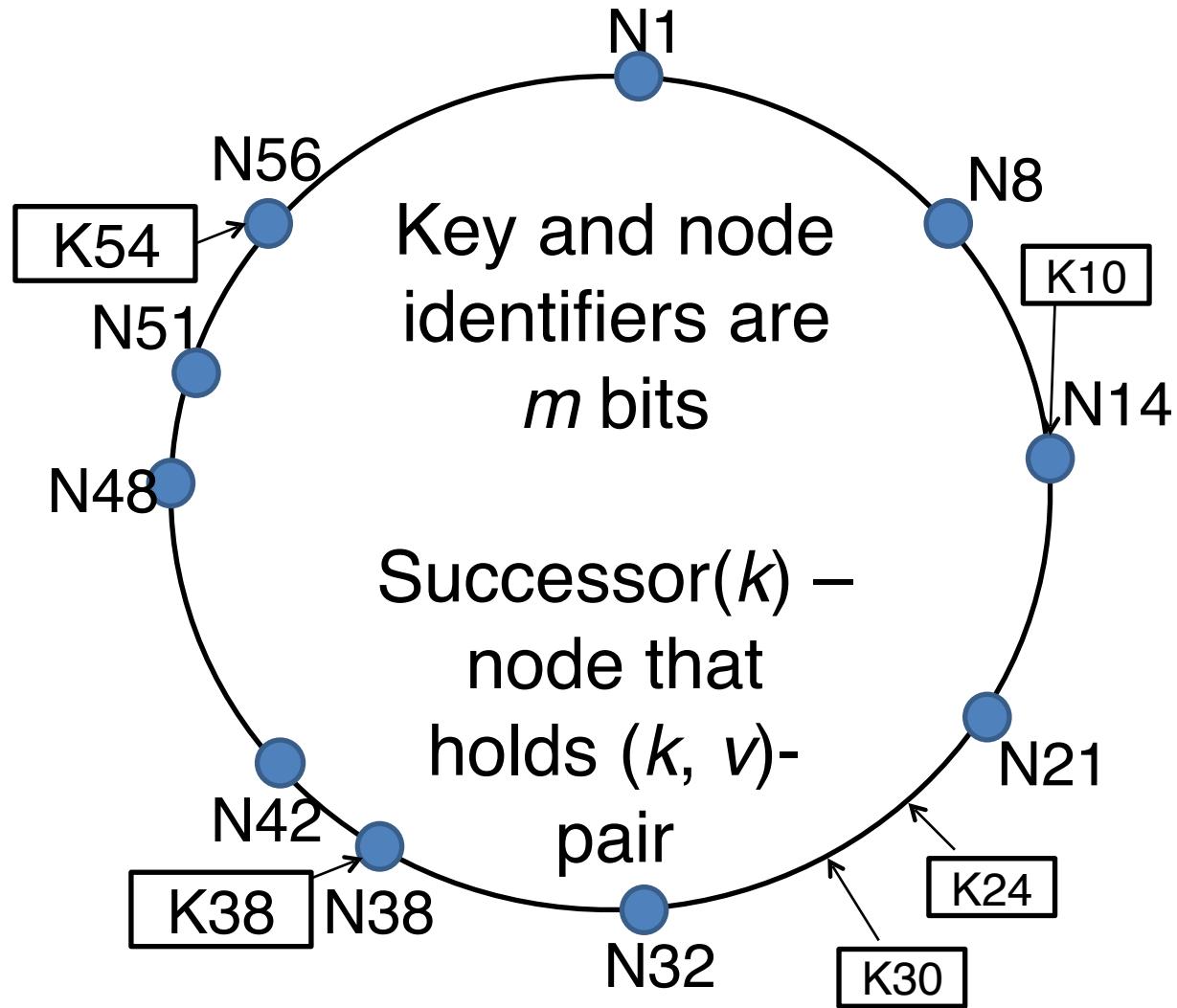
- Keys mapped evenly to all nodes in the network (load balancing)
- Node arrivals & departures only affect a few nodes (low maintenance)
- Each node maintains information about only a few other nodes (low maintenance)
- Messages can be routed to a node efficiently (fast lookup)

Lookup Services for DHTs

- Cover main intuition behind
 - Chord
 - Content addressable network

Chord Identifier Circle

- **Nodes** organized in an **identifier circle** based on **node identifiers**
- **Keys** assigned to their **successor** node in the identifier circle
- **Hash function** ensures **even distribution of nodes and keys** on the identifier circle
- Cf. **consistent hashing**
- With N nodes and K keys each **node is responsible for** roughly K/N keys



* All Chord figures from “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications”, Ion Stoica et al., IEEE/ACM Transactions on Networking, Feb. 2003.

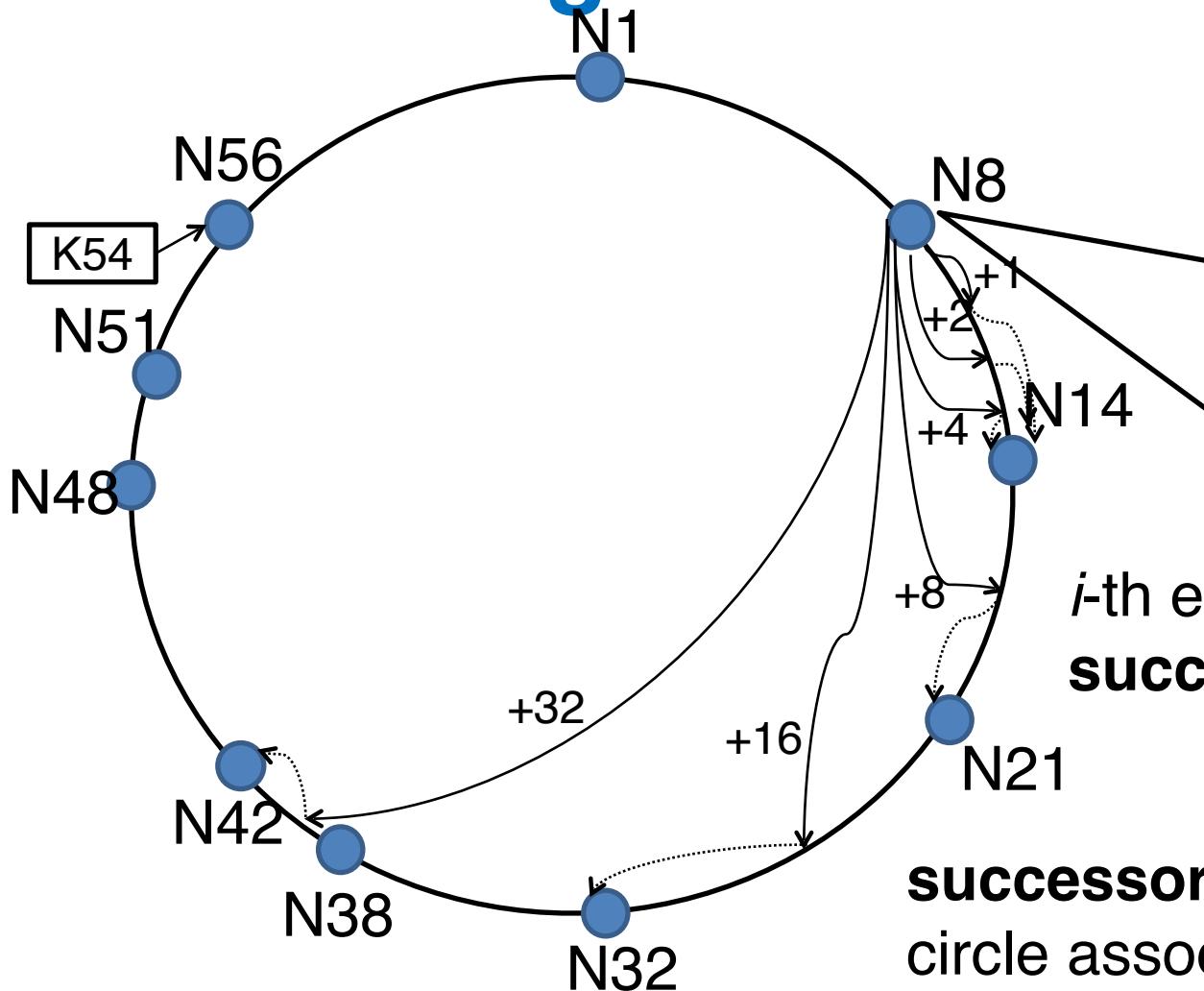
Node Joins & Leaves

- Nodes may disappear from the network (e.g., failure, departure)
- Each node records a whole segment of the circle adjacent to it, i.e., ***r nodes preceding and following*** it
- With high probability a node is able to correctly locate its successor or predecessor (even under high node churn)
- When a new node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands

Searching in Chord

- With knowledge of a single successor, a linear search through network could locate key (naïve search)
- Any given message may potentially have to be relayed through most of the network, i.e., cost is $O(n)$
- Faster search method requires each node to keep a "***finger table***" (FT) containing up to m entries
 - i -th entry in FT of node n contains the address of **successor($(n + 2^{i-1}) \bmod 2^m$)**
 - number of nodes that must be contacted to find a successor in an n -node network is **$O(\log n)$**

Chord Finger Table



Finger table

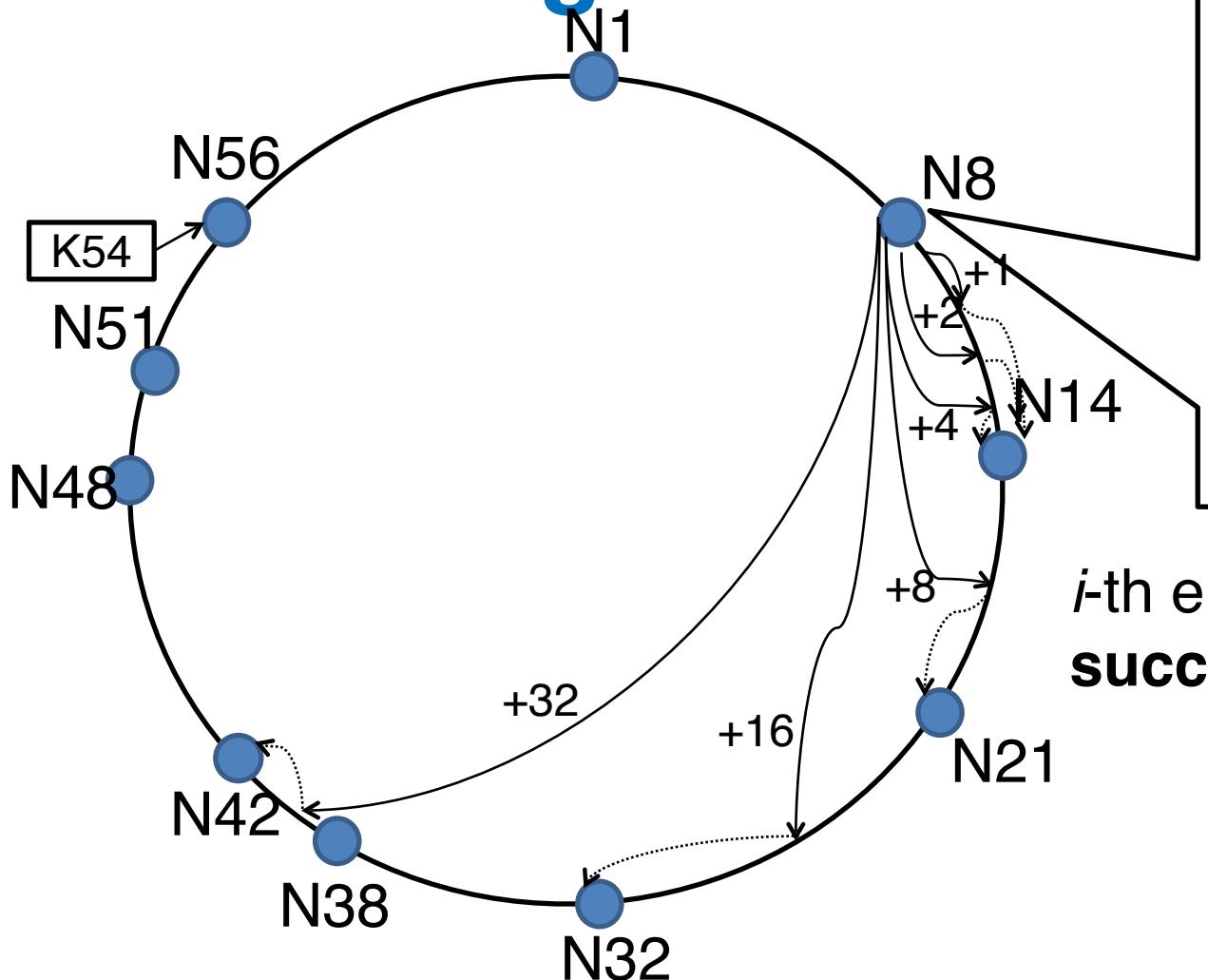
N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 + 16	N32
N8 + 32	N42

i-th entry is address of
successor((n + 2ⁱ⁻¹) mod 2^m)

successor(...) is the node on
circle associated with input
argument

For node n , i^{th} finger points to 1st node that succeeds n
by at least 2^{i-1} hops

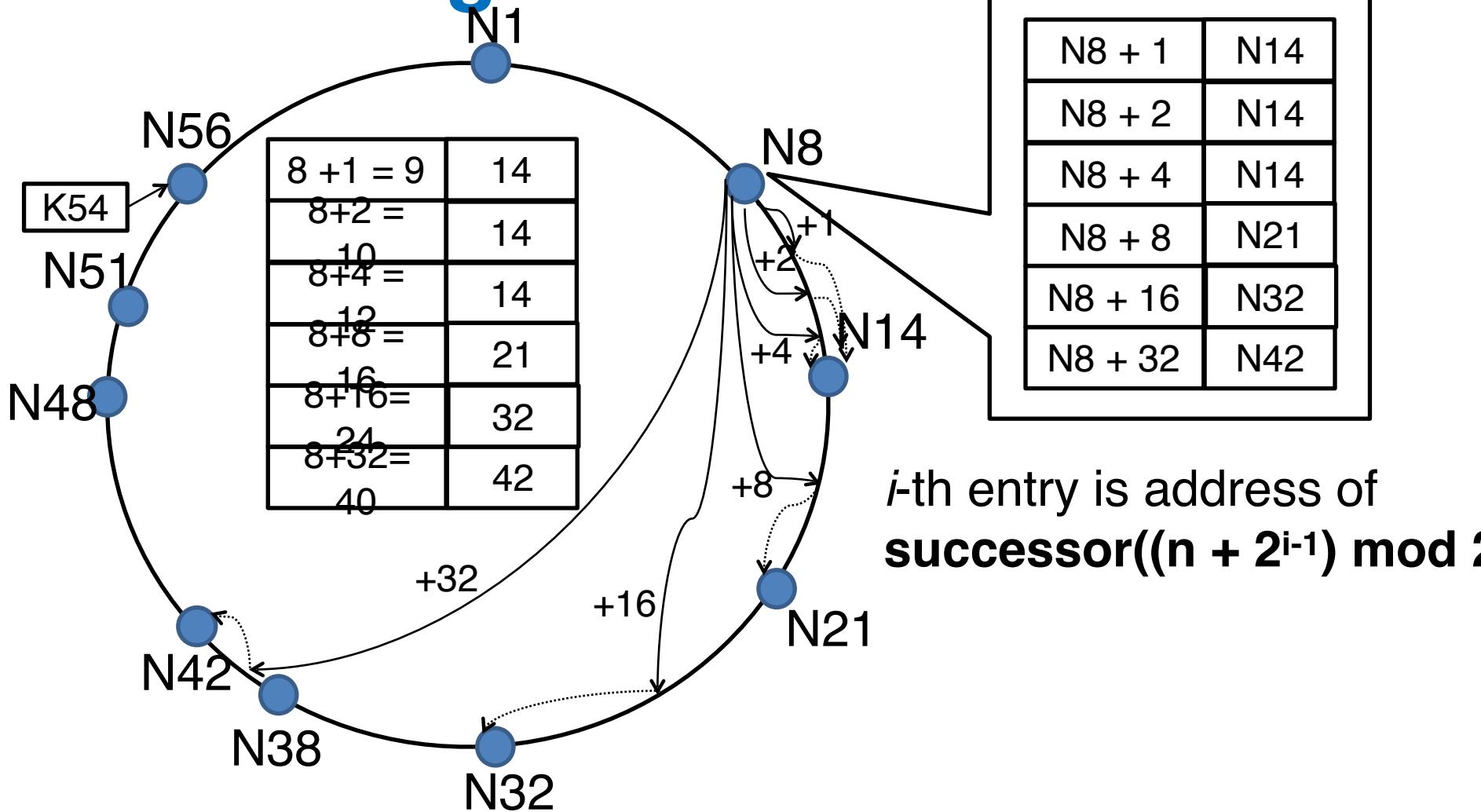
Chord Finger Table



i -th entry is address of
successor($(n + 2^{i-1}) \bmod 2^k$)

For node n , i th finger points to first node that succeeds n by at least 2^{i-1} hops

Chord Finger Table



Finger table

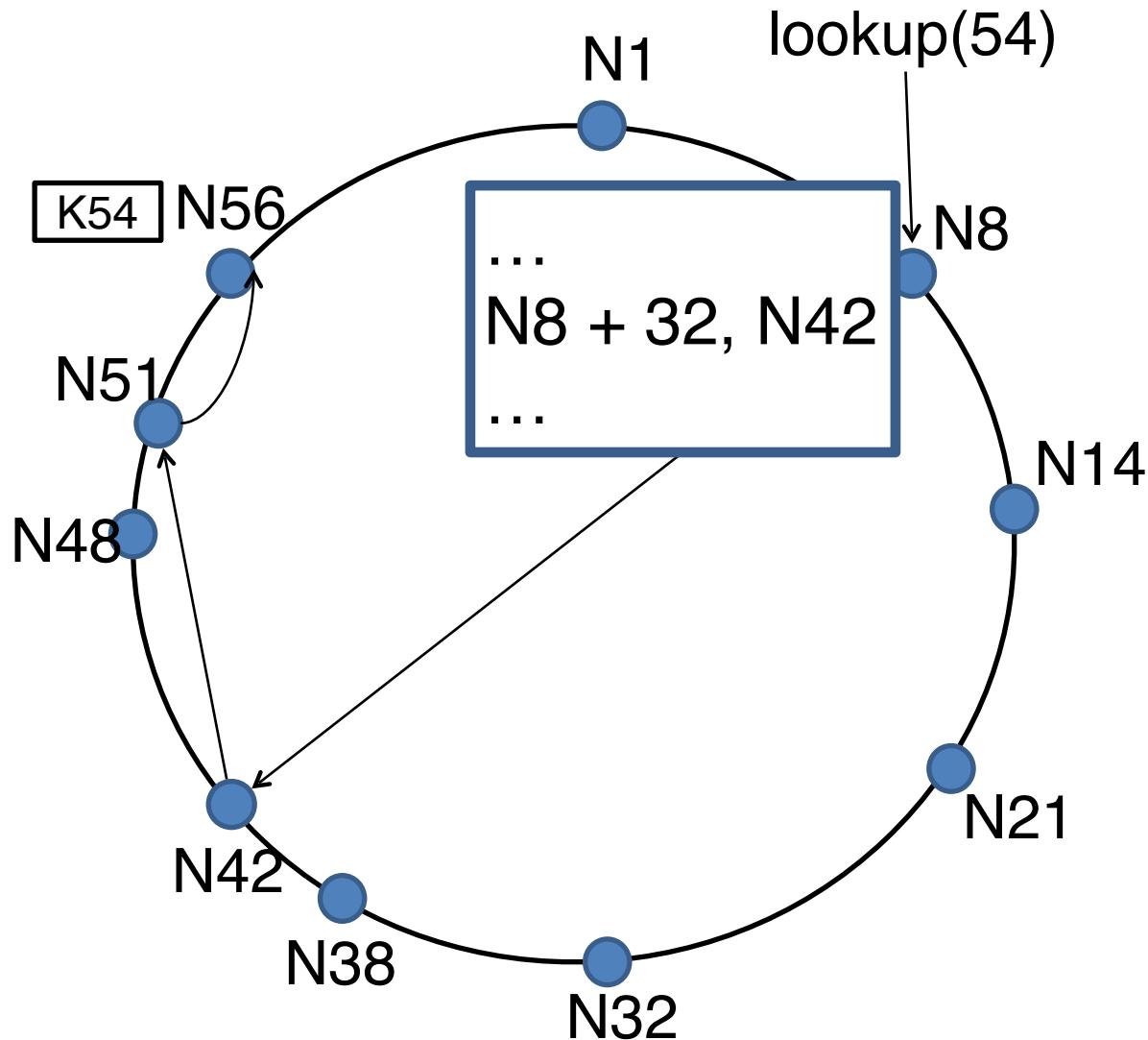
N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 + 16	N32
N8 + 32	N42

i -th entry is address of
successor($(n + 2^{i-1}) \bmod 2^m$)

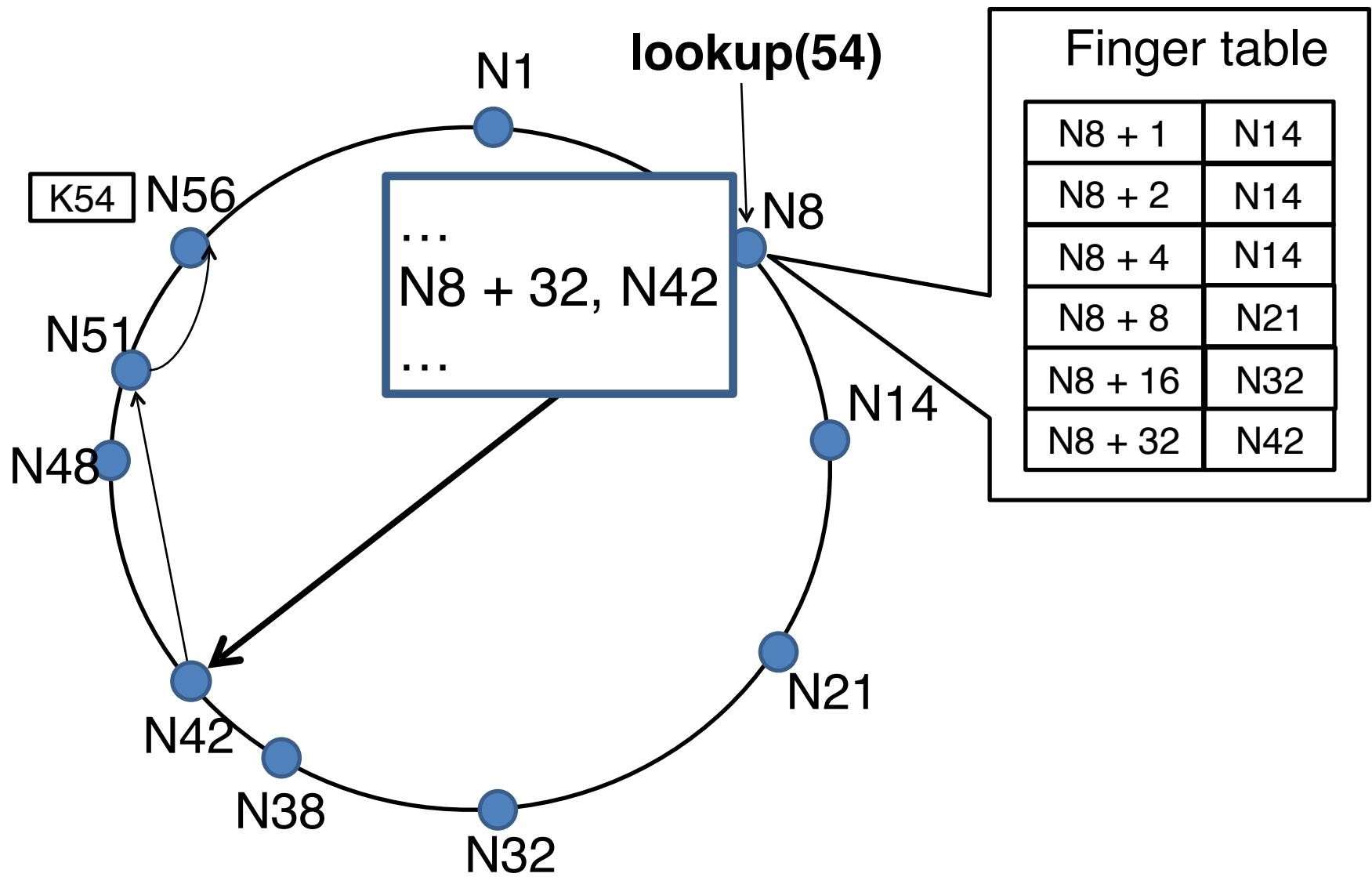
For node n , i th finger points to first node that succeeds n by at least 2^{i-1} hops

Chord Key Location

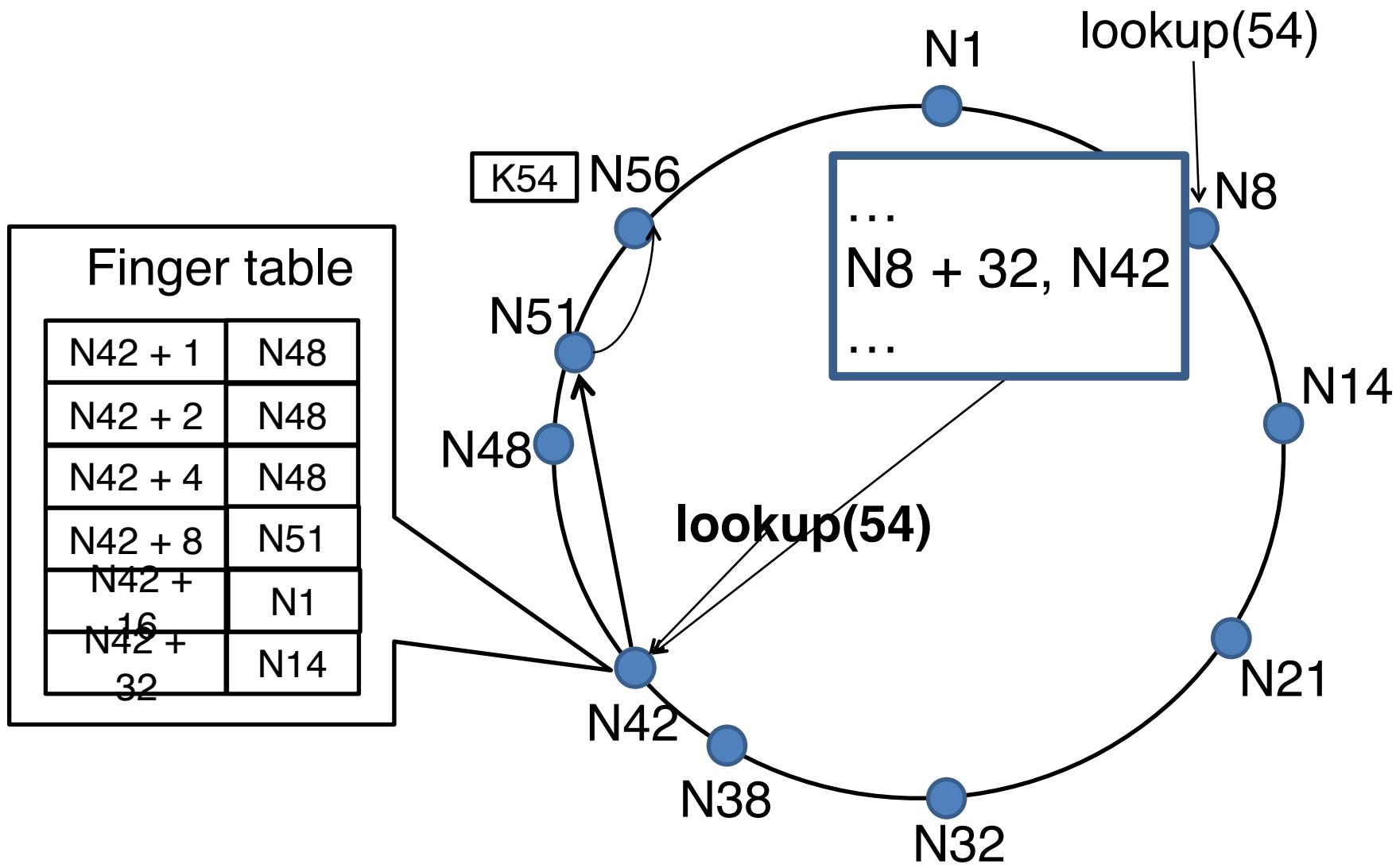
- Lookup in finger table the farthest node that precedes key
- Query homes in on target in $O(\log n)$ hops
- Each hop at least halves distance to destination



Lookup Example



Lookup Example (cont.)



Lookup Latencies

- While $O(\log n)$ is better than $O(n)$, it can still take considerable amount of time to find the target
- For example, $\log(1,000,000)$ hops which may be distributed anywhere
- Results in potentially high response latencies

Network locality

- Nodes close on ring can be far away in network

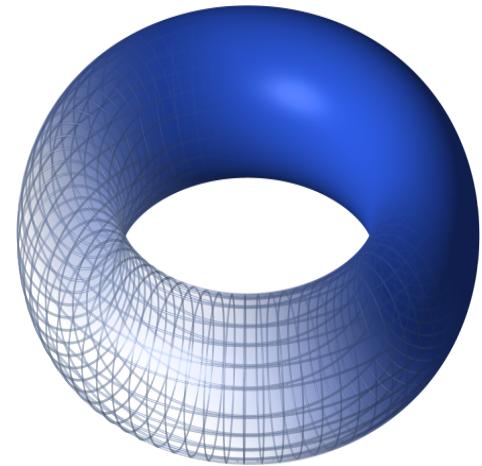


* Figure from <http://project-iris.net/dht-toronto-03.ppt>

CAN: Content addressable network

- Design is based on **virtual multi-dimensional Cartesian coordinate space** to organize overlay
- Nodes are layered on a multi-torus (i.e., **coordinates at edges wrap around**)
- Address space is **independent of physical location and physical connectivity** of nodes
- Points in the space are **identified with coordinates**
- General model is an n-dim. torus
 - Use dimensions for routing

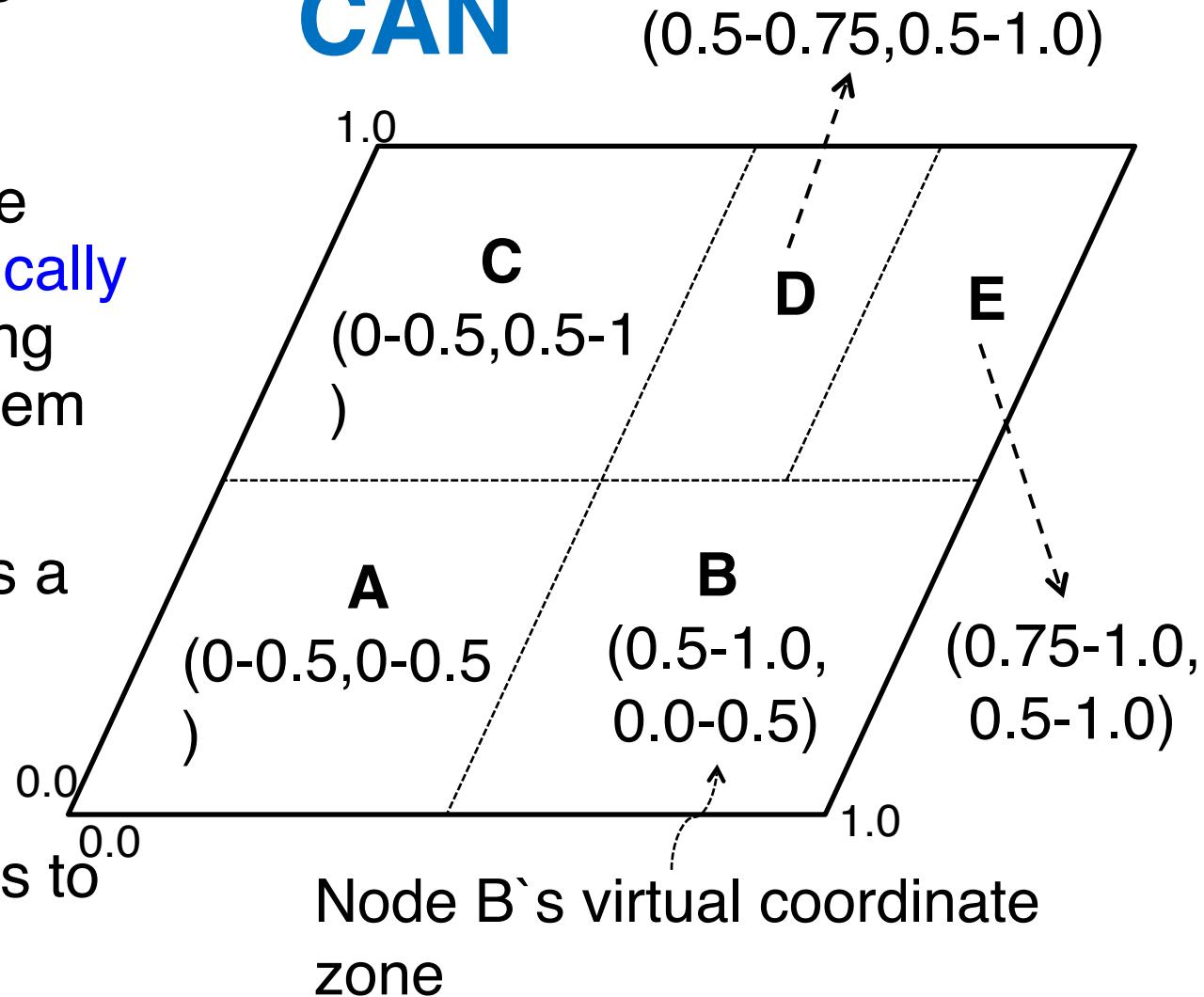
2001, SIGCOMM



Example 2-d space with 5 nodes

- Entire coordinate space is **dynamically partitioned** among all nodes in system
- Each node owns a distinct **zone** in space
- Each key hashes to a **point** in space

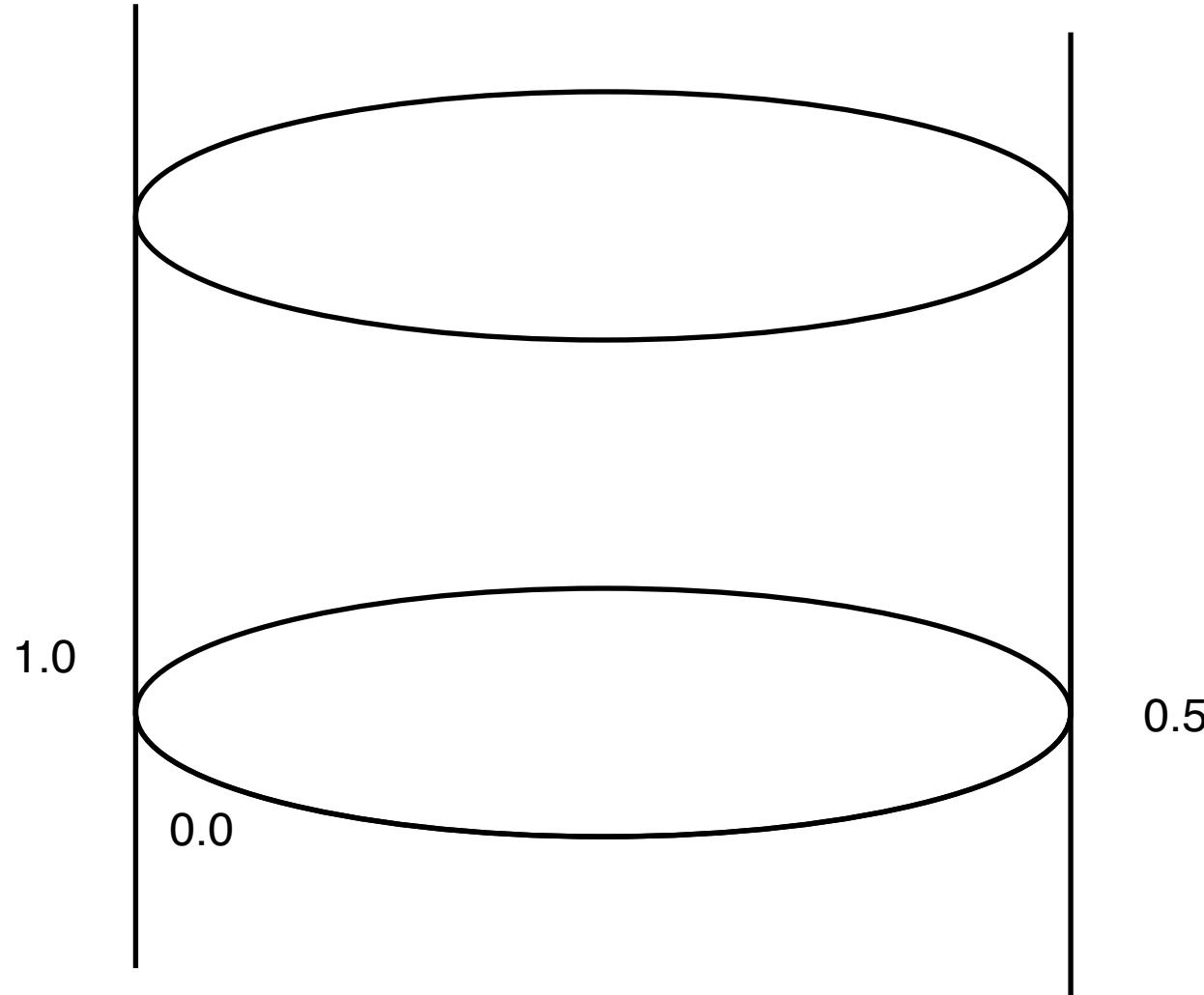
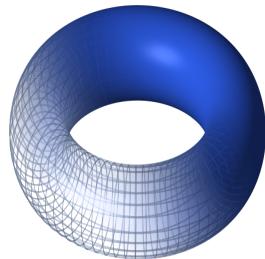
CAN



* All CAN figures from “A Scalable Content-Addressable Network”, S. Ratnasamy et al., In Proceedings of ACM SIGCOMM 2001.

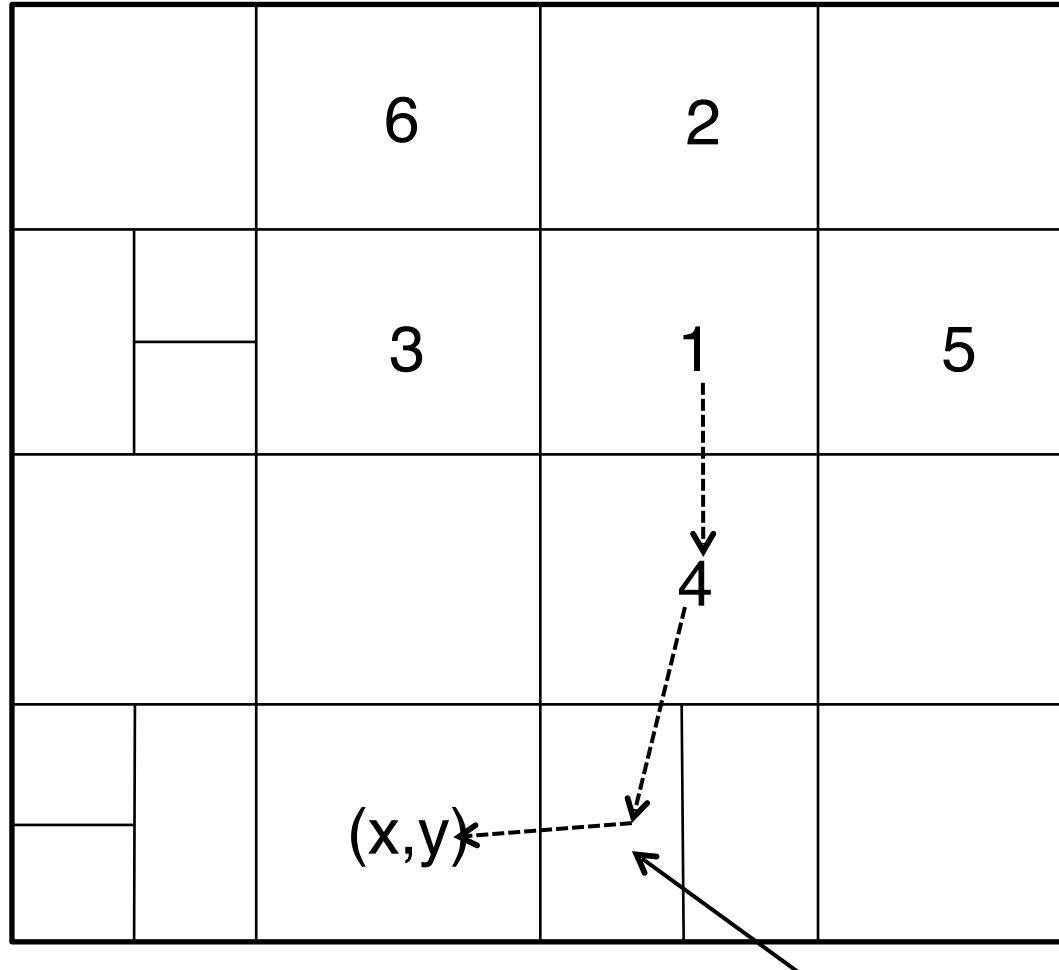
Coordinates Wrap Around

In all dimensions (only x-axis shown)



CAN Routing

- Put(key, data), get(key)
- Greedily forward message to neighbor closest to destination in Cartesian coordinate space
- Nodes maintain a routing table that holds IP address and zone of its neighbours



Sample routing path
from
node 1 to point (x, y)

CAN Routing

- Many possible routing paths exist between two points in space
- If a neighbour on a path crashes, simply pick the next best available (node) path

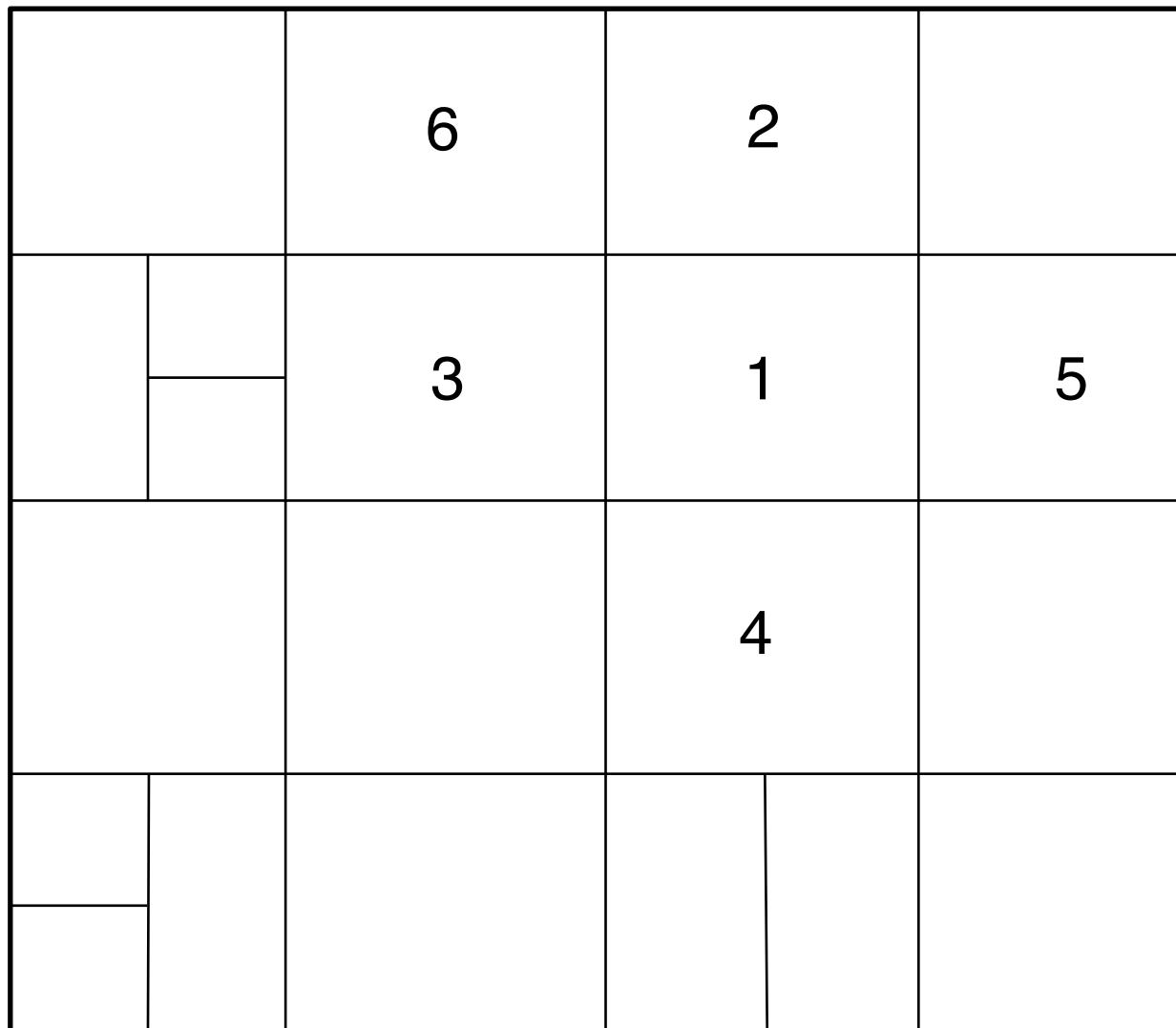
Average Path Length

- **d -dimensional space**, partitioned into **n equal-sized zones**,
average routing path length is: $d/4 * n^{1/d}$
- Each node maintains $2d$ neighbours
- Grow number of nodes, without affecting per node state
- Grow number of nodes, increases path length by $O(n^{1/d})$
- 2-dimensional space: $1/2 * n^{1/2}$ (average routing path)
- 3-dimensional space: $3/4 * n^{1/3}$ (average routing path)

Node Joining a CAN

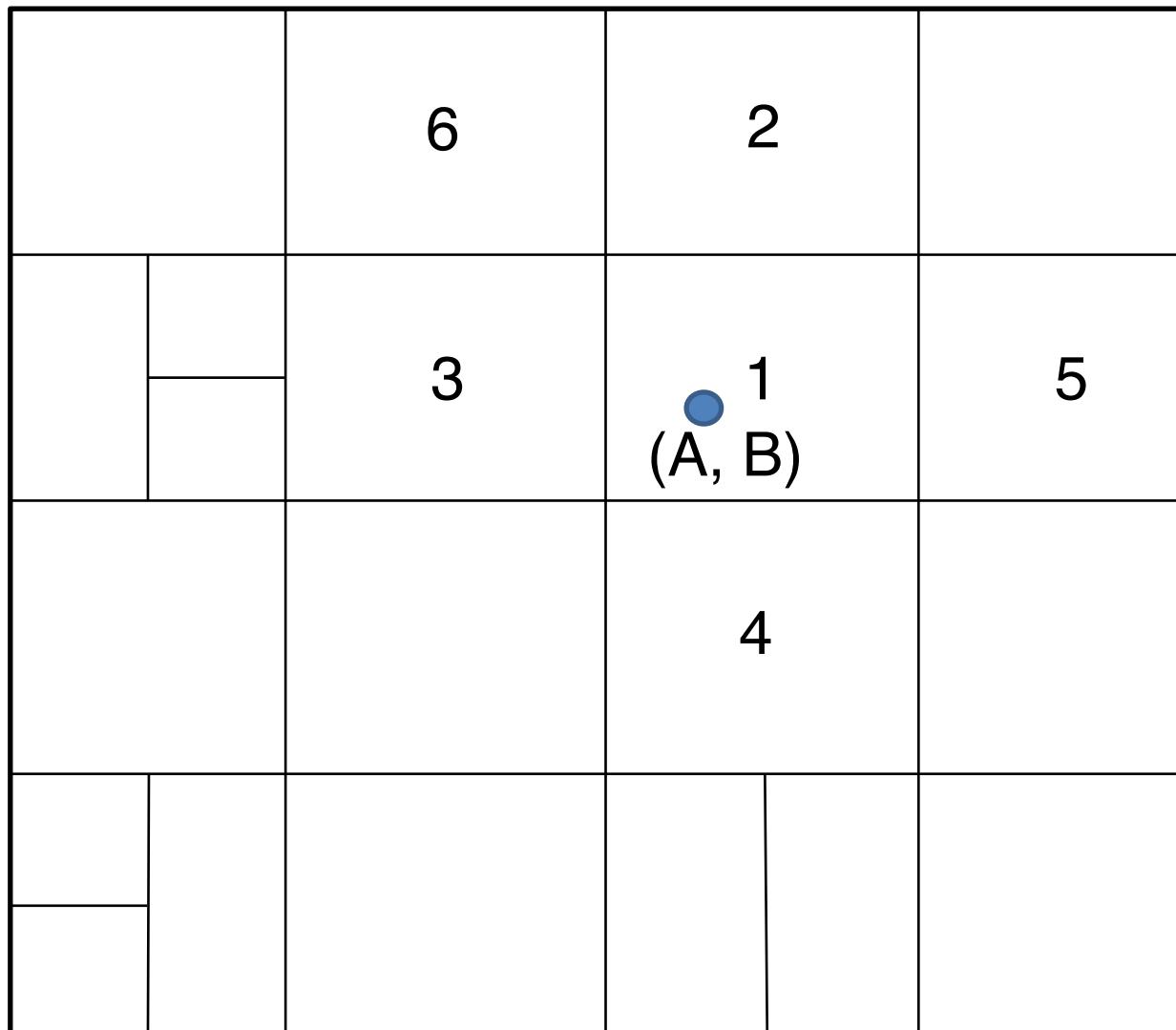
- Find a node already in overlay network
- Identify a zone that can be split
 - Pick random point
 - Route join request to node managing the point's zone
 - Initiate split of zone at that node
- Update routing tables of nodes neighbouring newly split zone
- If refused, try with a new random point

Zone Splitting Upon Node Joining



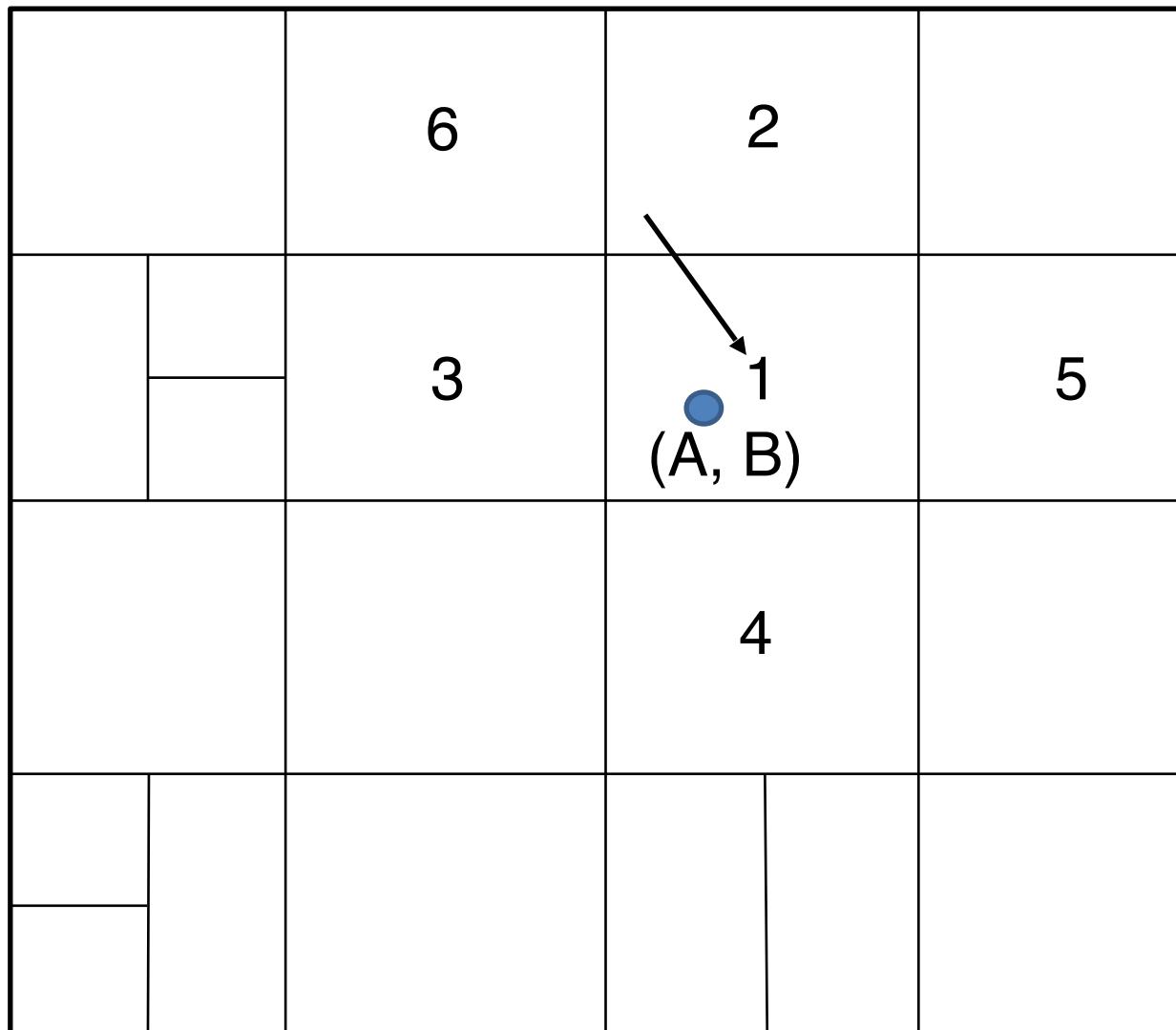
1's
coordinate
neighbor
 $\text{set} = \{2, 3, 4, 5\}$
Join request
Node 7

Zone Splitting Upon Node Joining



1's
coordinate
neighbor
 $\text{set} = \{2, 3, 4, 5\}$
Join request
Node 7
• Pick random
point (A, B)

Zone Splitting Upon Node Joining

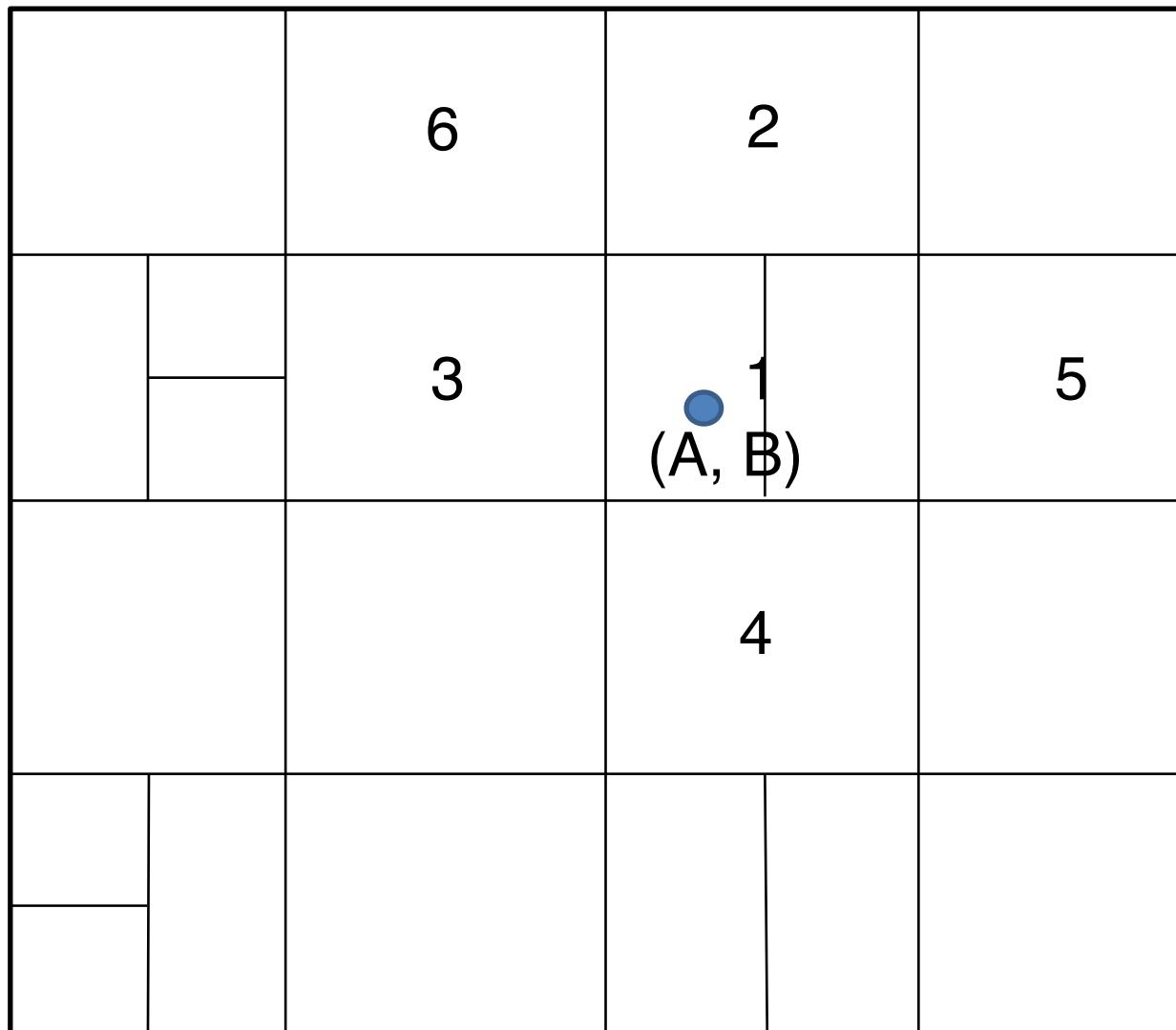


1's
coordinate
neighbor
 $\text{set} = \{2, 3, 4, 5\}$
Join request

Node 7

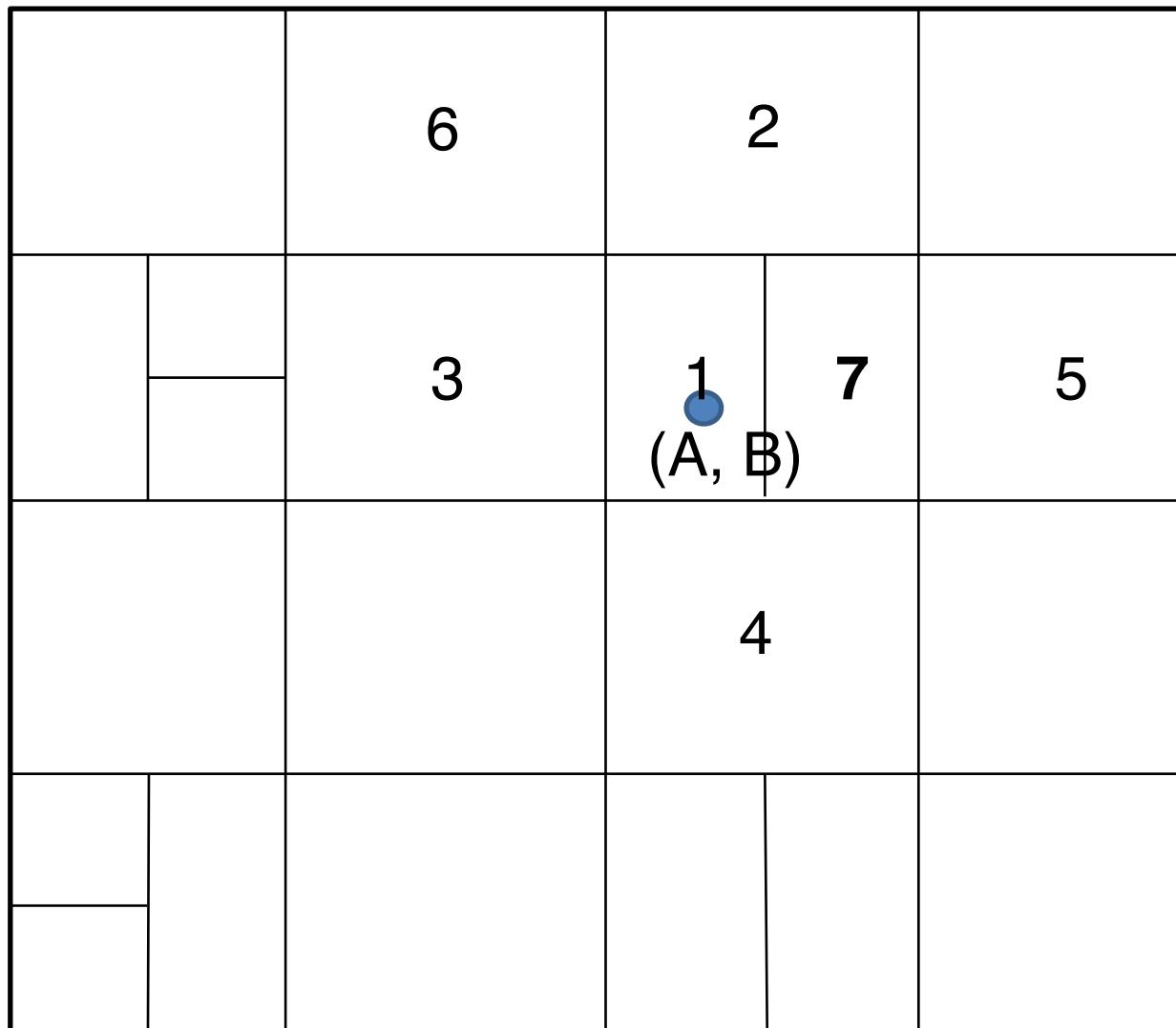
- Pick random point (A, B)
- Route join request of 7 to 1

Zone Splitting Upon Node Joining



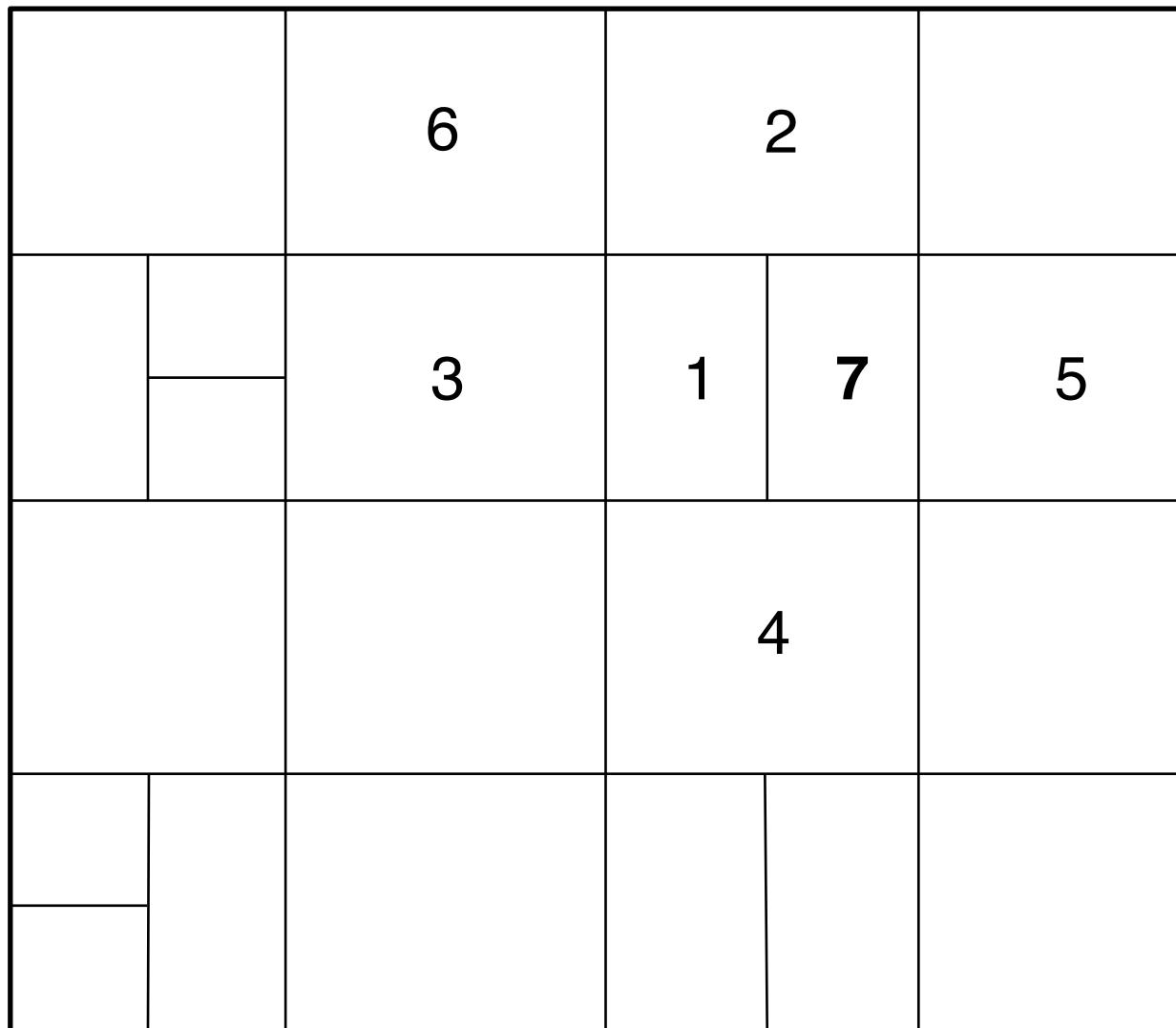
- 1's
coordinate
neighbor
 $\text{set} = \{2, 3, 4, 5\}$
Join request
Node 7
- Pick random point (A, B)
 - Route join request of 7 to 1
 - Initiate zone split

Zone Splitting Upon Node Joining



- 1's
coordinate
neighbor
 $\text{set} = \{2, 3, 4, 5\}$
Join request
Node 7
- Pick random point (A, B)
 - Route join request of 7 to 1
 - Initiate zone split

Zone Splitting Upon Node Joining



1's coordinate neighbor set = {2,3,4,7}
7's coordinate neighbor set = {1,2,4,5}

Update routing tables of nodes, transfer state, i.e., (k, v)-pairs (not shown)

Node Join Properties

- Only $O(d)$ nodes are effected when a node joins/leaves CAN (a node has $2d$ neighbours)
- Independent of n , number of nodes in CAN

DHT Routing Summary

- Chord
 - Finger table routing
 - Each hop at least halves distance (in identifier circle) to destination
- CAN
 - Neighbour routing
 - Forward to neighbor closest (in Cartesian coordinate space) to destination

Conclusions on P2P

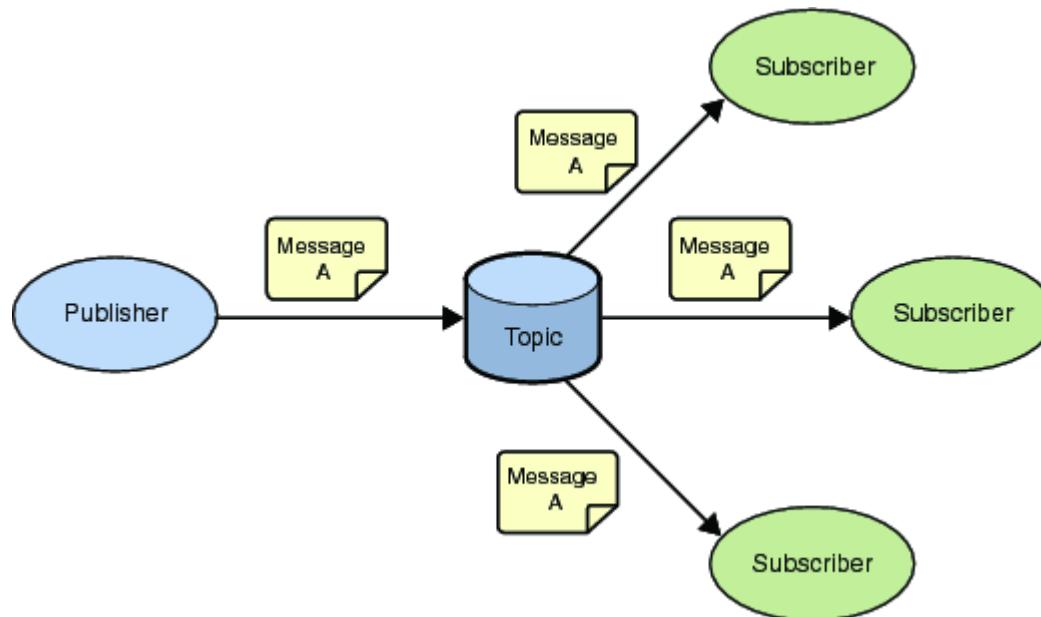
- Hugely popular area of research 2000-2010
- Large-scale companies (Amazon & Google et al.) prefer self-managed cloud infrastructures
- P2P principles, techniques and abstractions are used by large-scale systems (e.g., DHTs)
- Active applications: **BitTorrent**, **Bitcoin** *et al.*
- Peer-assisted, hybrid systems are probably here to stay: **Skype**, **Spotify**, etc.

Publish/Subscribe Systems

Pezhman Nasirifard

Hans-Arno Jacobsen

Application & Middleware Systems Research Group

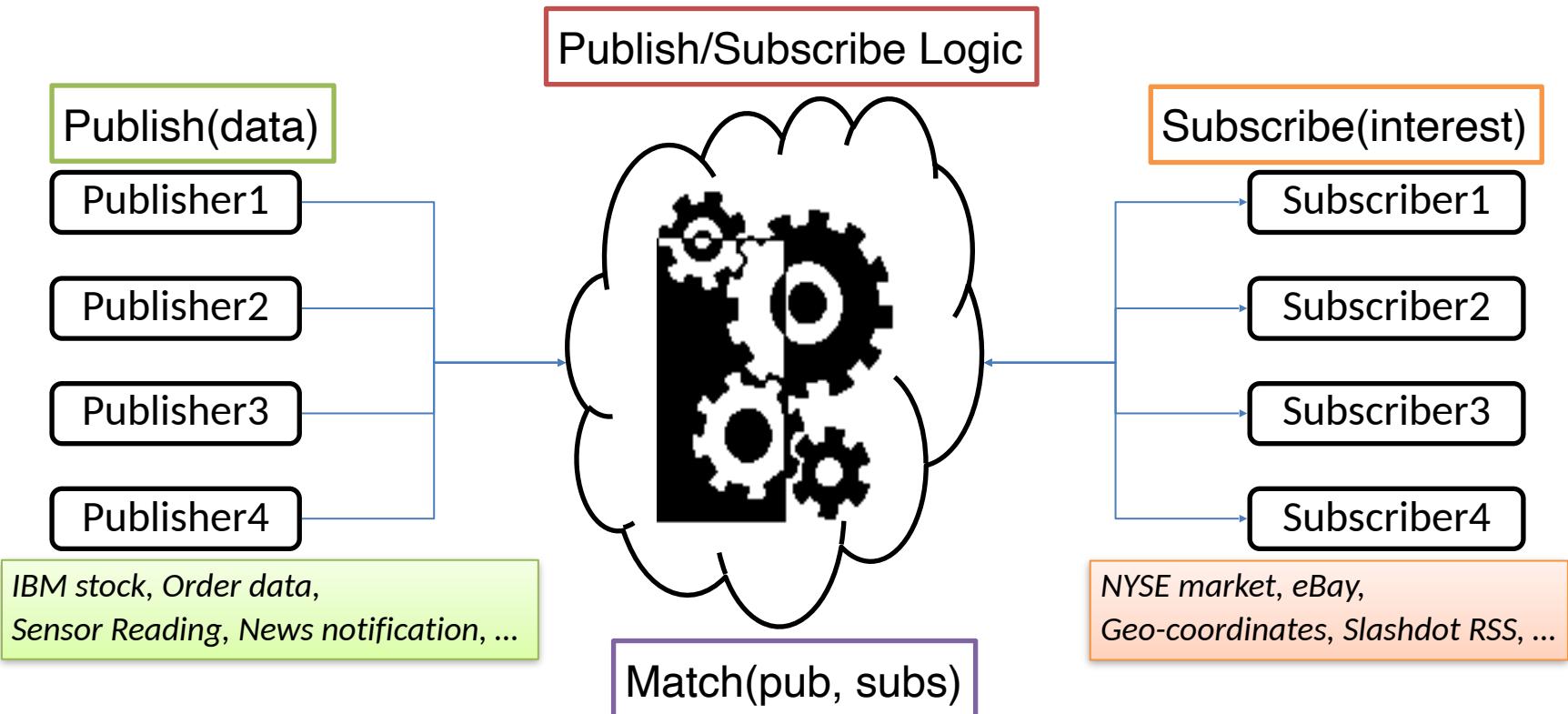


Agenda

- Publish/Subscribe
 - Definition
 - Decoupling
 - Applications
- Matching
 - Topic-based Model
 - Content-based Model
- Routing
 - Rendezvous-based
 - Overlay-based routing

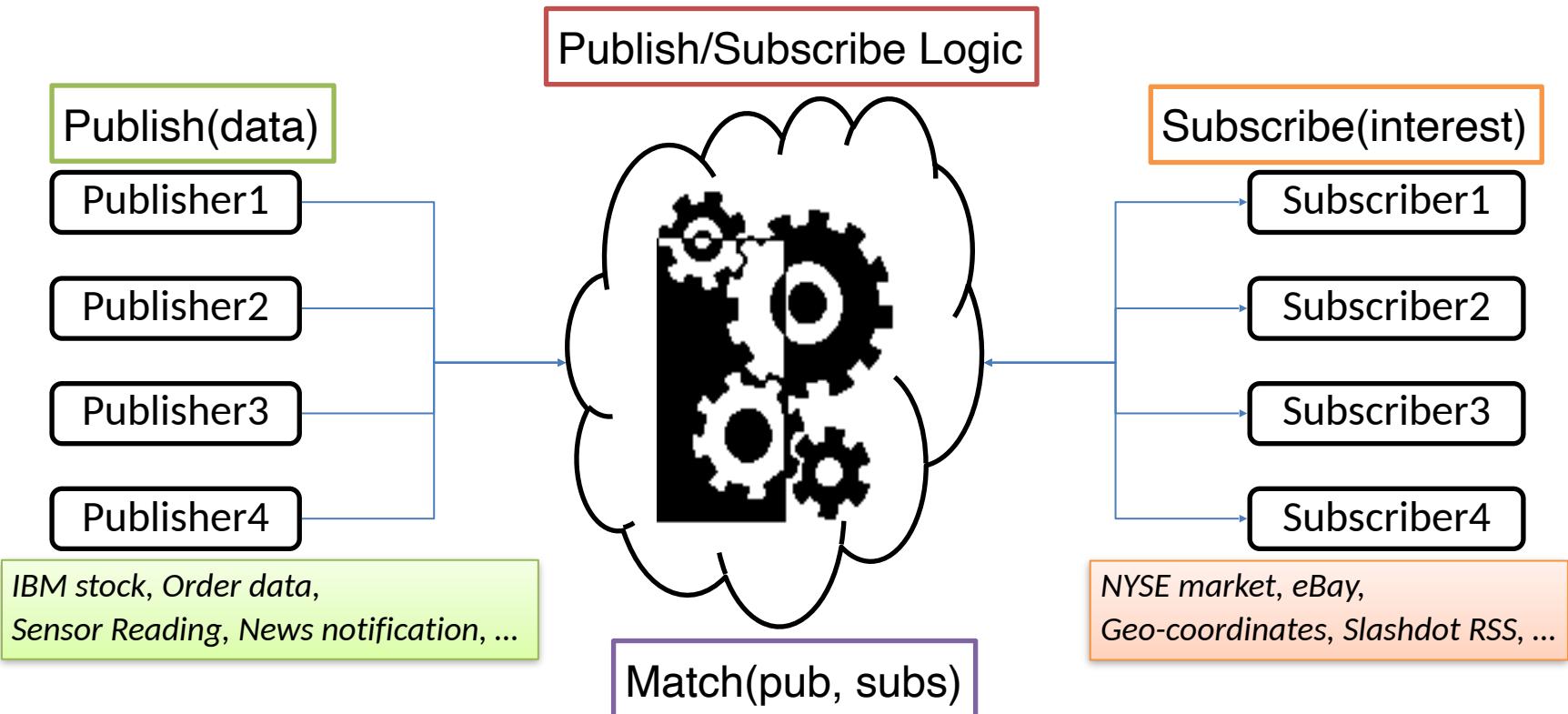
PUBLISH/SUBSCRIBE

Definition



“Publish/subscribe is a messaging pattern which allows **many-to-many** communication: publishers send **publications**, subscribers send **subscriptions** and receive publications of interest.”

Definition



“Publish/subscribe is a messaging pattern which allows **many-to-many** communication: publishers send **publications**, subscribers send **subscriptions** and receive publications of interest.”

Pub/Sub API

- Publish(data, metadata): publication metadata depends on the language (cf. Matching model)
- Subscribe(interest): subscription interest is defined depending on the language (cf. Matching model)
- Unsubscribe(interest): remove a subscription
- (Un-)Advertise(interest): publisher *advertises the type of content it will publish (optional)*

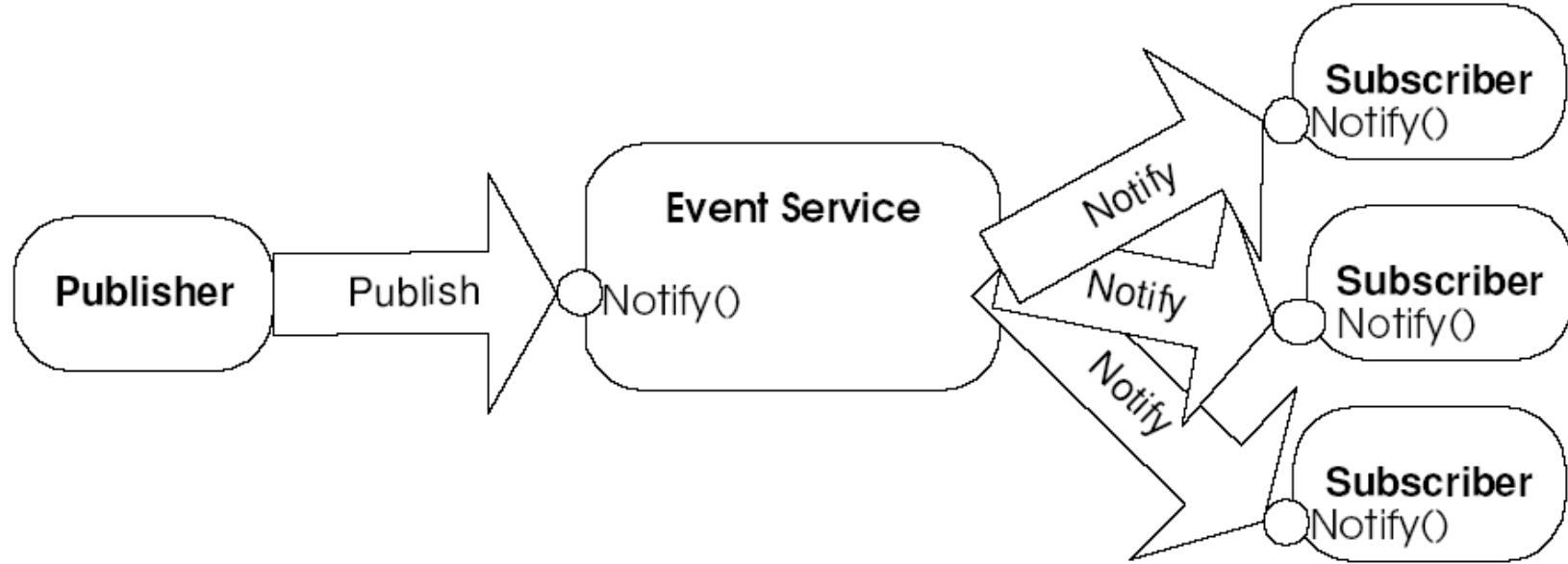
Design Decisions

- Matching-based communication:
 - Not host-to-host communication
 - Filtering is based on the **matching model**
- Optimized for **scalability** and **performance**
 - Large number of publishers and subscribers
 - High rate of publications
 - Fast matching: usually **stateless** (does not consider pub sequences)
 - Simple matching: commonly limited to **topics**
- **Decoupling** properties:
 - Time
 - Space
 - Synchronization

Why Decoupling?

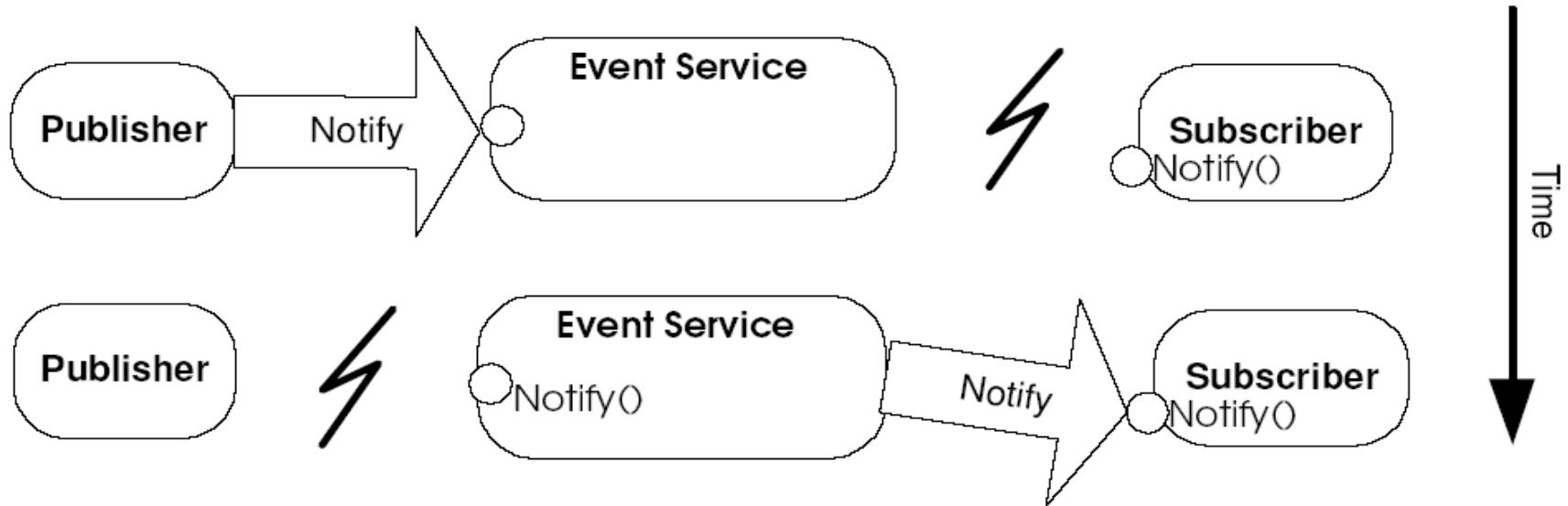
- Decouple publishing and subscribing clients
 - Removes explicit dependencies
 - Reduces coordination
 - Reduces synchronization
- Increases scalability of distributed systems
- Creates highly dynamic, decentralized systems
- Decoupling in at least three dimensions
 - Space
 - Time
 - Synchronization (flow)

Space Decoupling



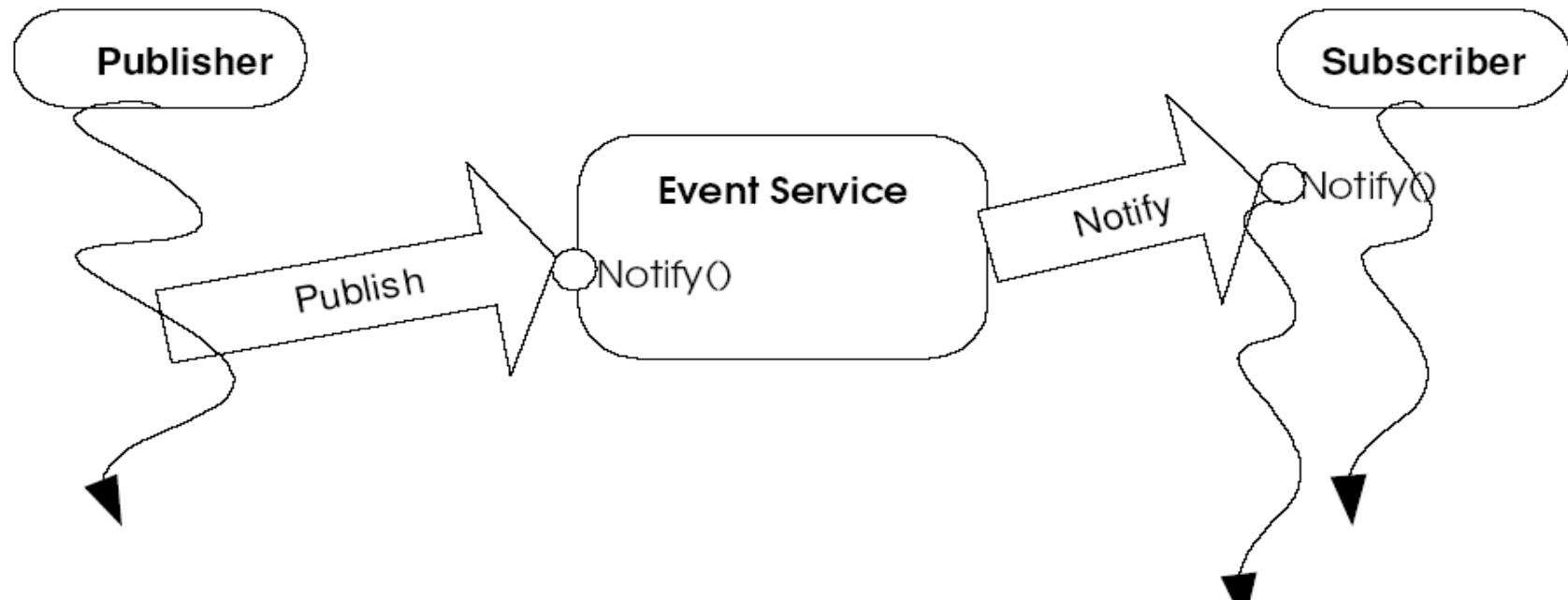
- No need for clients (publishers & subscribers) to hold references or even know each other
- Clients may be physically distributed

Time Decoupling



- No need for clients to be available at the same time

Synchronization Decoupling



- Control flow is not blocked by the interaction

Comparison of Communication Patterns

Abstraction	Space de-coupling	Time de-coupling	Synchronization decoupling
Message Passing	No	No	Producer-side
RPC/RMI	No	No	Producer-side
Asynchronous RPC/RMI	No	No	Yes
Future RPC/RMI	No	No	Yes
Notifications (Observer D. Pattern)	No	No	Yes
Tuple Spaces	Yes	Yes	Producer-side
Message Queuing (Pull)	Yes	Yes	Producer-side
Publish/Subscribe	Yes	Yes	Yes

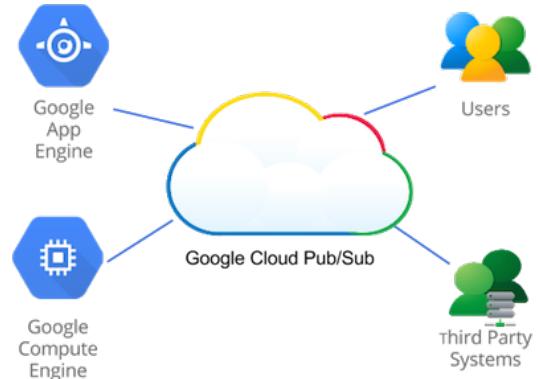
Applications



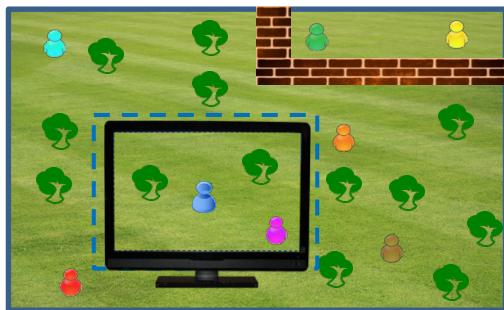
News Syndication



Stock Tracker



App Notifications



Network Engine for Games

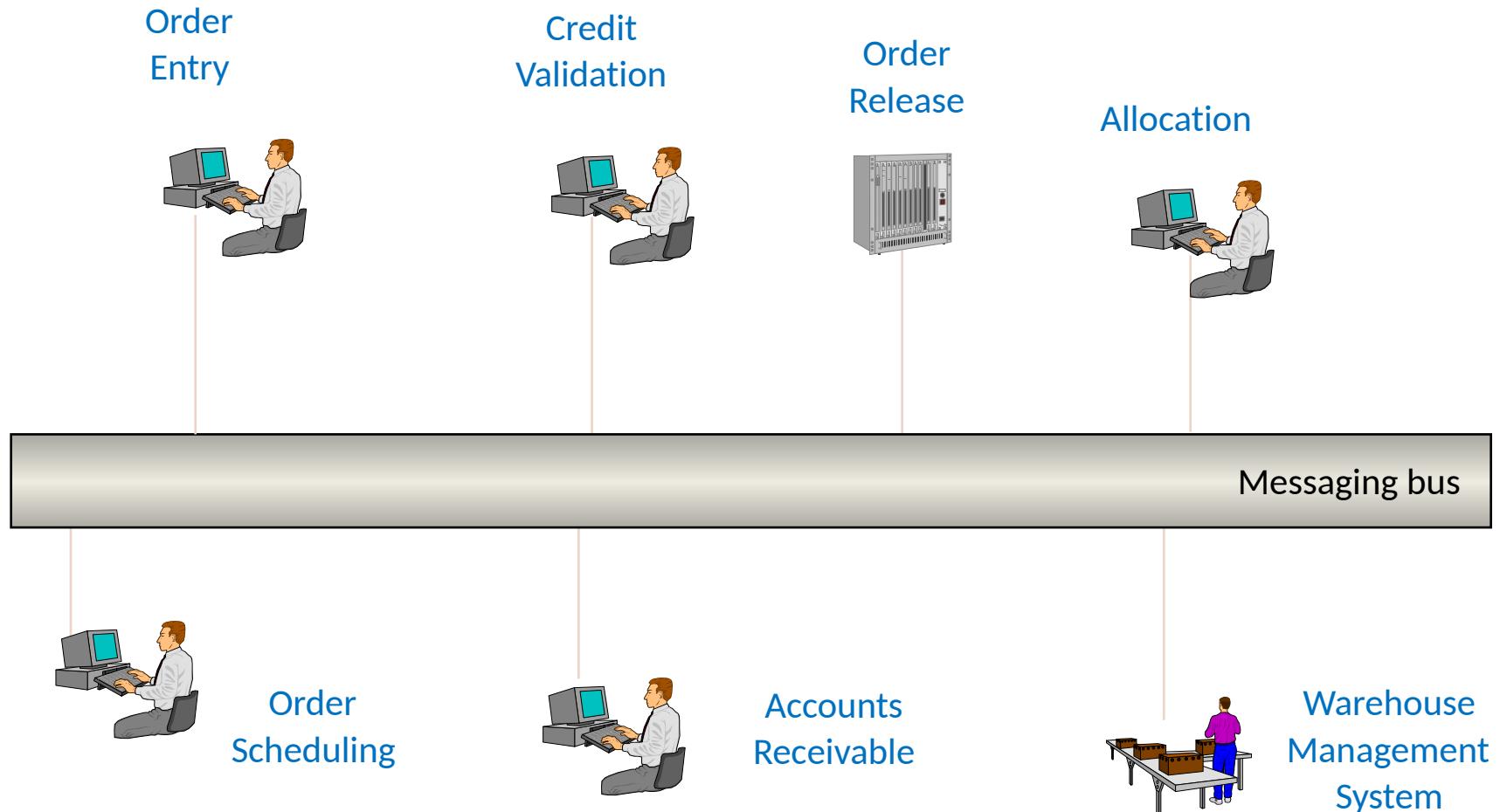


Social Networks

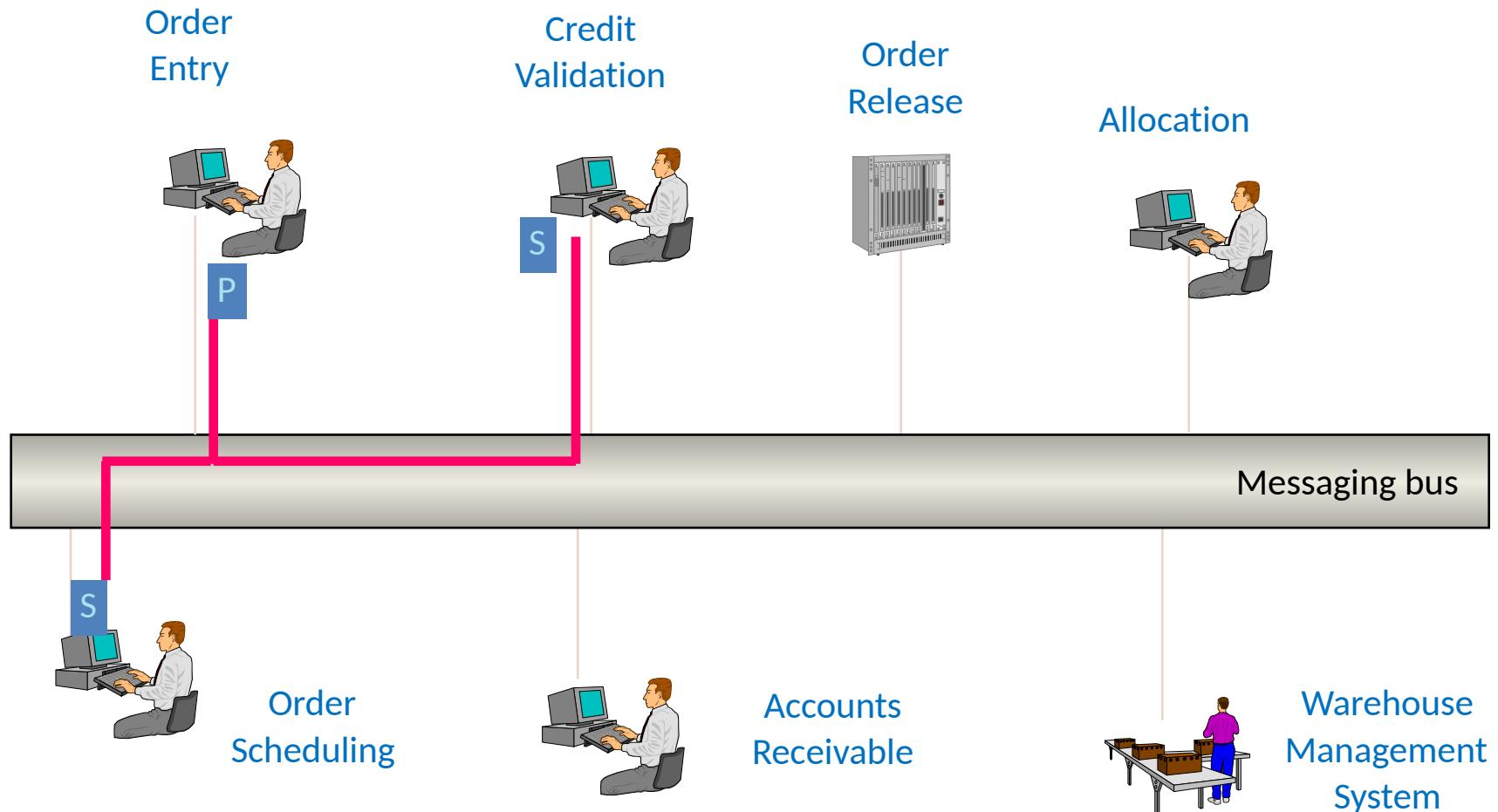


Traffic Monitoring

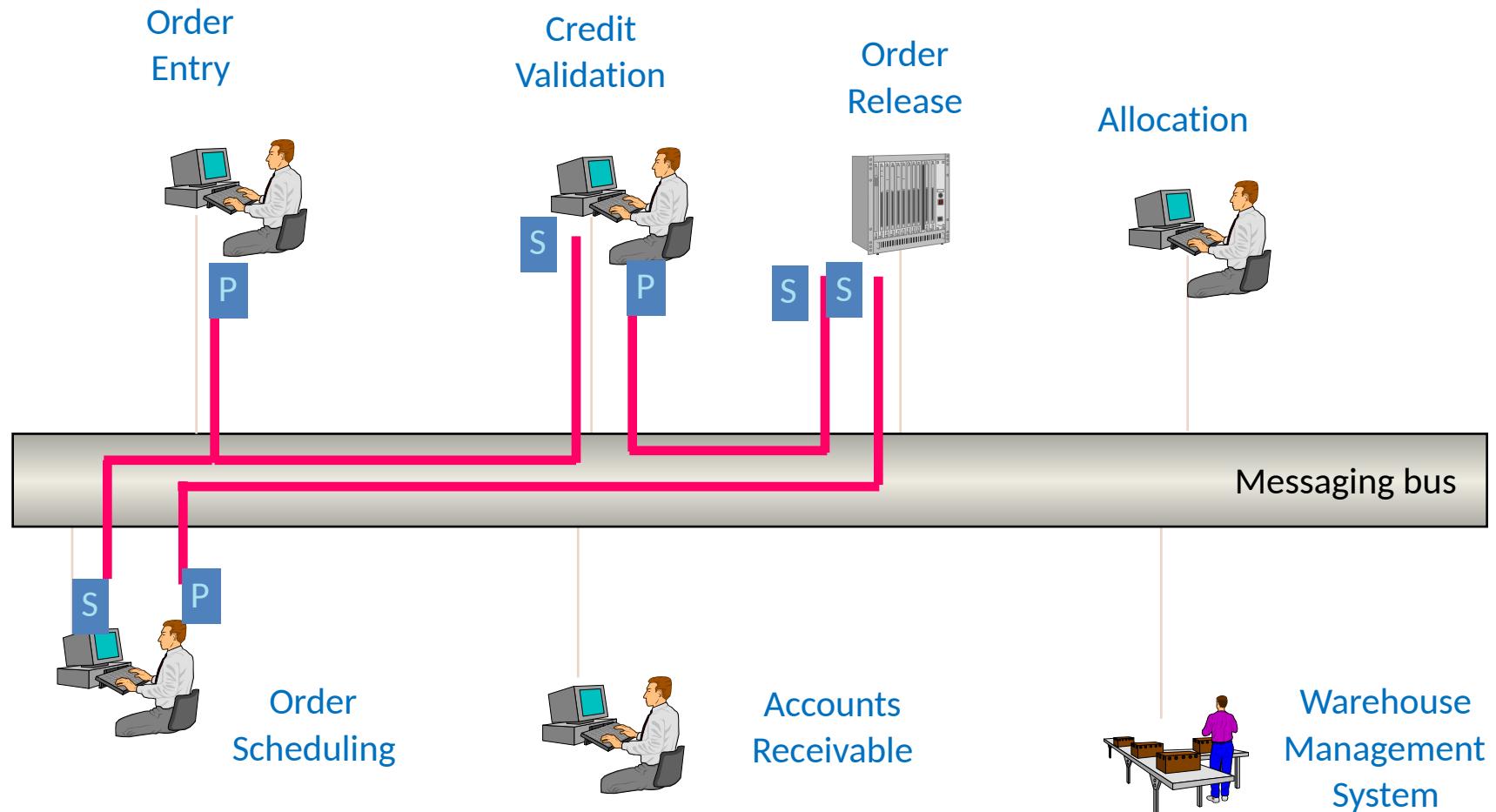
Enterprise Application Integration



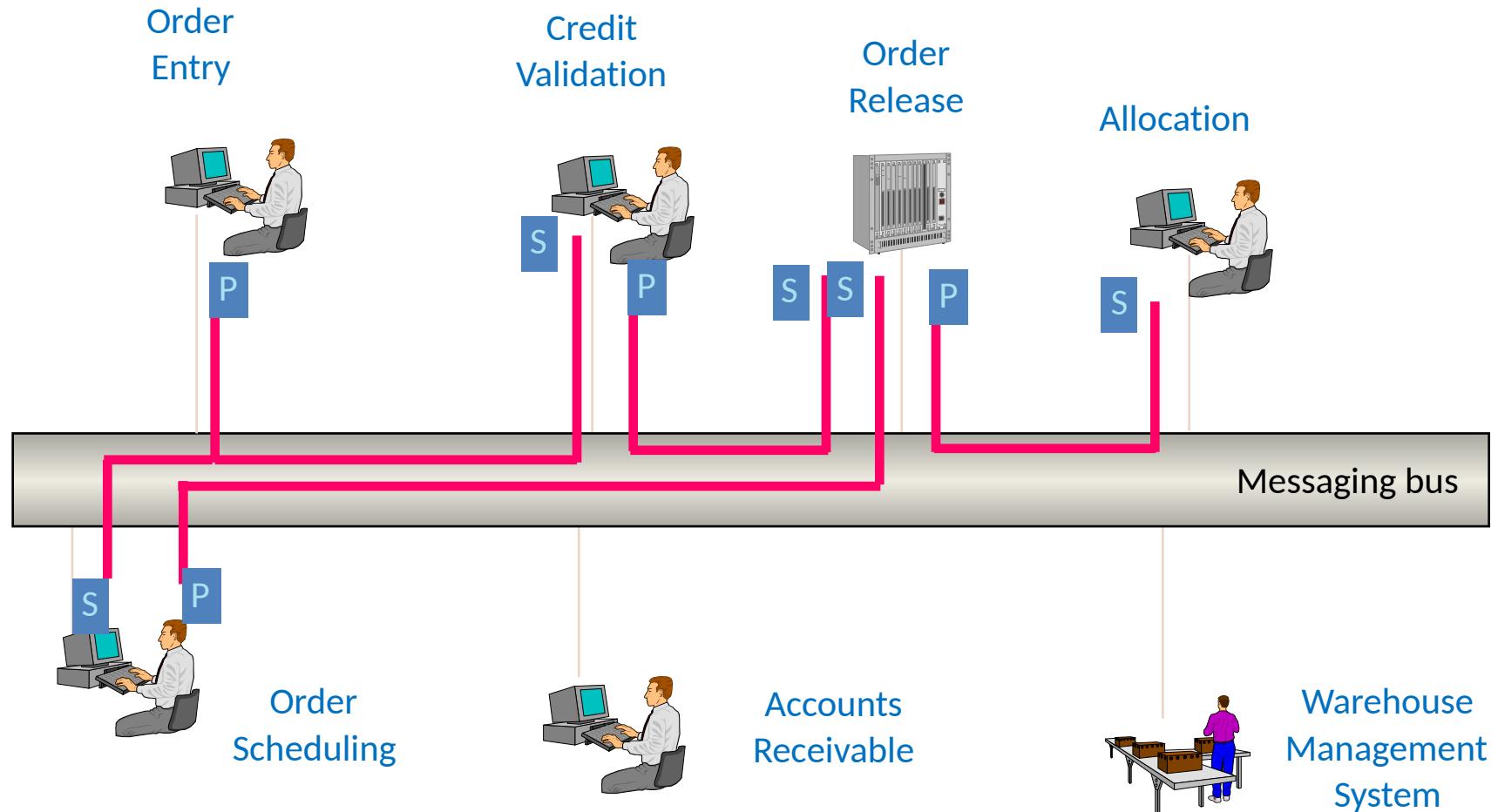
Enterprise Application Integration



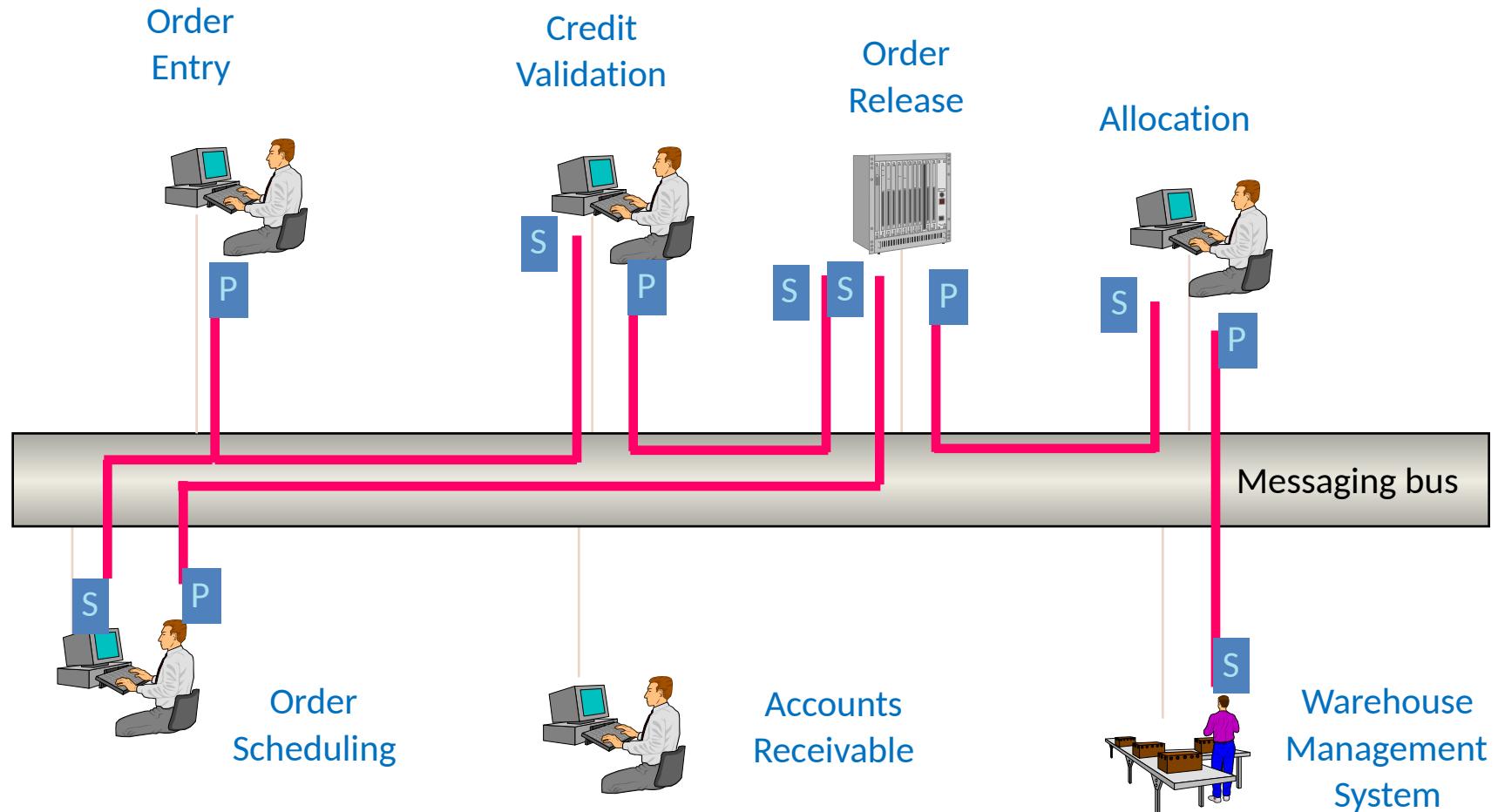
Enterprise Application Integration



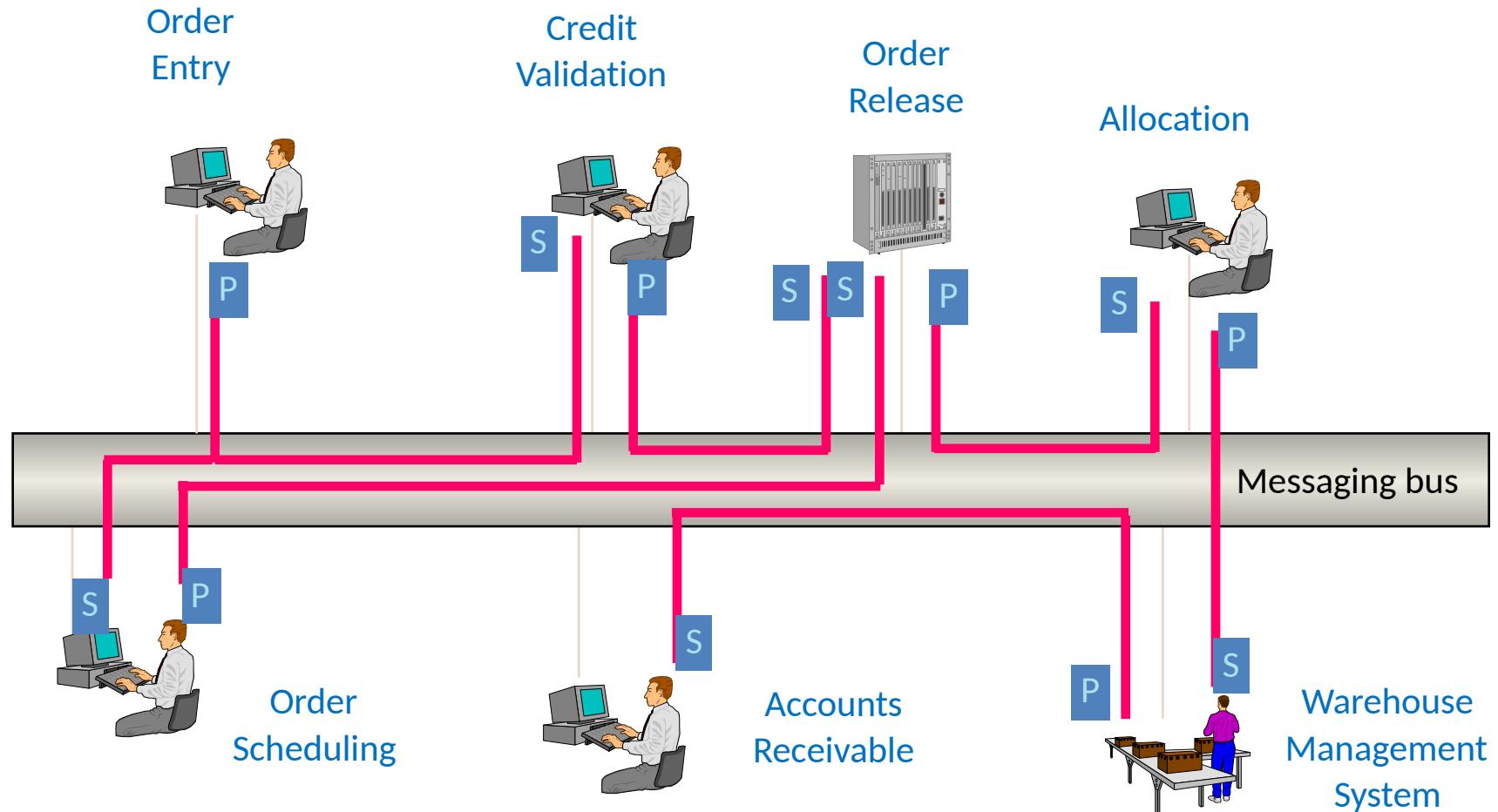
Enterprise Application Integration



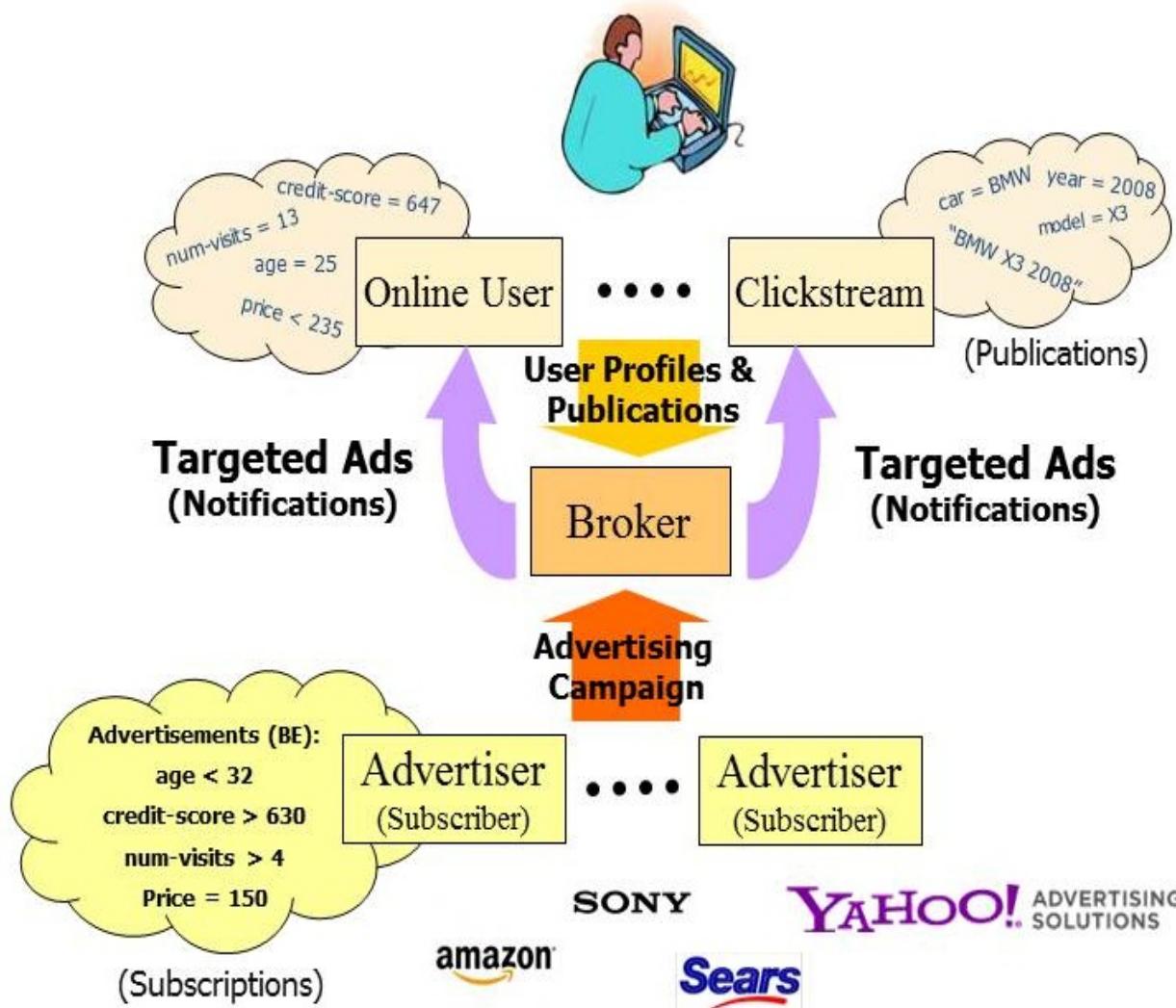
Enterprise Application Integration



Enterprise Application Integration



Computational Advertising: Filtering & Matching



Popular Systems in Industry



PubSubHubbub

Facebook Wormhole



ed Systems (H.-A. Jacobsen)



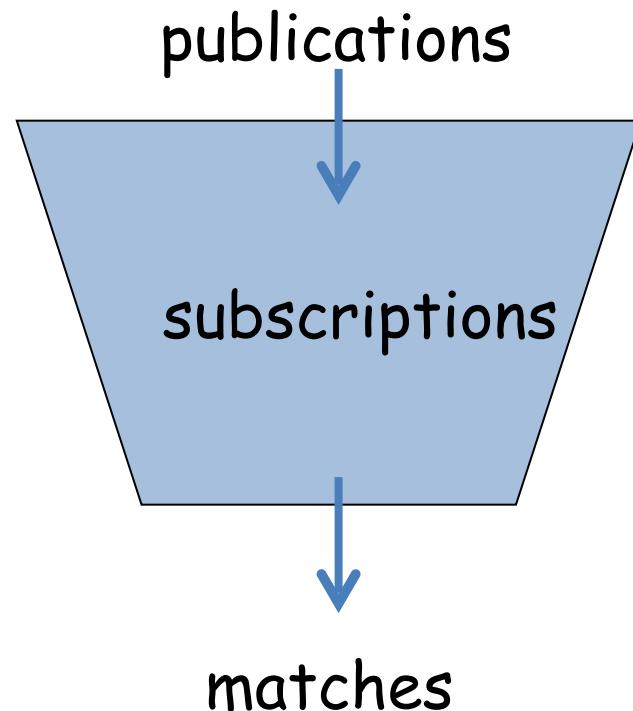
Summary of Pub/Sub

- Pub/sub provides **many-to-many** communication between publishers and subscribers.
- Communication is decoupled with respect to **time, space, and synchronization**.
- Many applications are **event-based** or require **event notifications**, which leverage pub/sub.
- Many flavors of pub/sub, which differ in **matching** and **routing**.

MATCHING MODELS

Matching & Filtering Problem in Pub/Sub

- **Matching problem:** Given a publication, e , and a set of subscriptions, S , determine all subscriptions, $s \in S$, that match e .
- Different for each model
 - Topic-based filtering
 - Content-based filtering
 - Type-based filtering
 - ...



Matching Model

- The matching model describes **how** subscriptions and publications are **expressed**.
- Each subscription contains **filters** which determine whether a publication matches or not.
- When a publication is sent, it is compared against each subscription to see which ones **match**.
- The publication is then delivered to **each subscriber with a matching subscription**.

Topic-Based Matching

- A.k.a. subject-based, group-based or channel-based event filtering (yet, there are subtle differences!)
- Publish(data, metadata) -> Publish(data, topic)
- Subscribe(interest) -> Subscribe(topic)
- A publication **matches** a subscription if it is published on the **same topic** as the subscription.
- Example:
 subscribe("IBM"),
 publish("price:175.31","IBM")

Topic-Based Matching II

- Static publisher & subscriber relationship
 - At publication time, the subscriber recipient set is known
 - For a given topic, unless topics or subscriptions change, every message published on the topic, is delivered to the same recipient set
- Topics can be hierarchically organized
 - E.g., sports/basketball/NBA
 - E.g., publish message to sports/basketball/NBA
- Topic subscriptions may contain wildcards
 - E.g., Subscribe to sports/basketball/*

Content-Based Matching

- **Subscription:** Conjunction of **predicates** (i.e., an attribute-operator-value triple)
$$(\text{subject} = \text{news}) \wedge (\text{topic} = \text{travel}) \wedge (\text{date} > 29.11.2011)$$
- **Publication (a.k.a. event):** Sets of attribute-value pairs
$$(\text{subject}, \text{news}), (\text{topic}, \text{travel}), (\text{date}, 21.2.2011), \dots$$

Content-Based Matching II

- Publish(data, metadata) -> Publish(data, list(attribute-value))
- Subscribe(interest) -> Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

publish("stock = IBM", "price = 180.0", "market = NYSE")

publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

Example	Tree-structured data	Graph-structured data	Unstructured data	Regular languages	Relational model
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) -> Publish(data, list(attribute-value))
- Subscribe(interest) -> Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")



publish("stock = IBM", "price = 175.31", "date = 31-12-1969")
publish("stock = IBM", "price = 175.5", "date = 1-1-2017")
publish("stock = IBM", "price = 180.0", "market = NYSE")
publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

NYSE")

Example	Tree-structured data	Graph-structured data	Unstructured data	Regular languages	Relational model
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) -> Publish(data, list(attribute-value))
- Subscribe(interest) -> Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

✗ publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

✓ publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

publish("stock = IBM", "price = 180.0", "market = NYSE")

publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market =
NYSE")

Example	Tree-structured data	Graph-structured data	Unstructured data	Regular languages	Relational model
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) -> Publish(data, list(attribute-value))
- Subscribe(interest) -> Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

 publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

 publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

 publish("stock = IBM", "price = 180.0", "market = NYSE")

publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market = NYSE")

Example	Tree-structured data	Graph-structured data	Unstructured data	Regular languages	Relational model
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Content-Based Matching II

- Publish(data, metadata) -> Publish(data, list(attribute-value))
- Subscribe(interest) -> Subscribe(predicates)
- A pub **matches** a sub if **all the predicates** are satisfied (conjunction).
- Example:

subscribe("stock = IBM", "price > 175.0", "date > 1-1-1970")

✗ publish("stock = IBM", "price = 175.31", "date = 31-12-1969")

✓ publish("stock = IBM", "price = 175.5", "date = 1-1-2017")

✗ publish("stock = IBM", "price = 180.0", "market = NYSE")

✓ publish("stock = IBM", "price = 177.5", "date = 1-1-2017", "market =

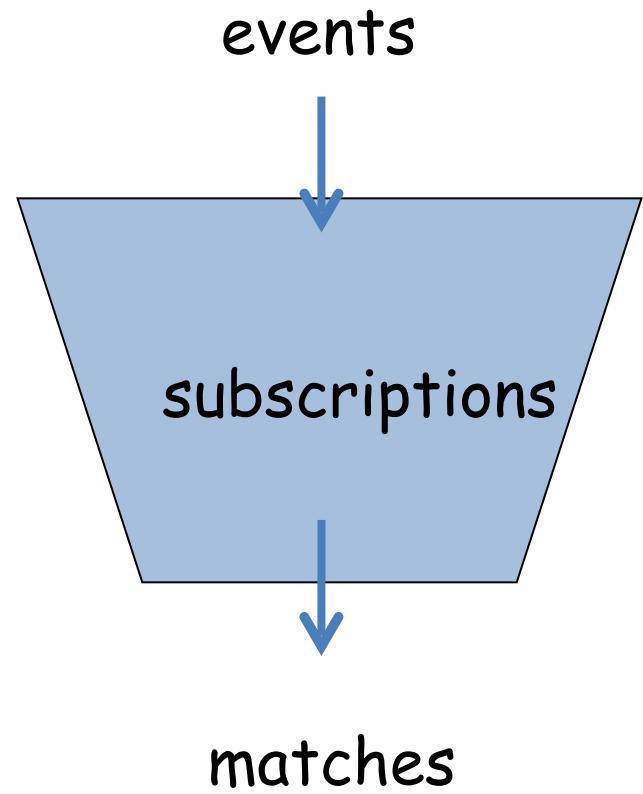
NYSE")

Example	Tree-structured data	Graph-structured data	Unstructured data	Regular languages	Relational model
Subscription	XPath	RDF Query	Keywords	Regular expressions	SQL
Publication	XML	RSS feeds	Text, documents	Sentences over some alphabet	DBs, i.e., tables

Language Tradeoffs

	Simple	Complex
Granularity	Coarse	Fine
Matching time	Fast	Slow
Communication	High	Low
Memory	Low	High

Matching problem: Given an event, e , and a set of subscriptions, S , determine all subscriptions, $s \in S$, that match e .



CONTENT-BASED MATCHING ALGORITHMS

Matching Algorithms

- Example algorithms
 - **Counting algorithm** [origins unknown to me; guess: 1970s]
 - *Clustering algorithm* [Fabret, Jacobsen et al., 2001]
- Both are **two-phased** matching algorithms
 1. Match all predicates (**predicate matching phase**)
 2. Match subscriptions from results of Phase 1 (**subscriptions matching phase**)
- Let us look at both phases separately
 - Predicate matching
 - Subscription matching

Two-Phased Matching

- **Decompose** subscriptions into predicates

$s_1 \ (subject^{p1} = \text{news}) \wedge (topic^{p2} = \text{travel}) \wedge (date >^{p3} 29.11.2011)$

$s_2 \ (subject^{p1} = \text{news}) \wedge (topic^{p4} = \text{stock}) \wedge (date >^{p5} 30.11.2011)$

- **Convert** subscriptions to reference predicates

S1: P1 \wedge P2 \wedge P3

S2: P1 \wedge P4 \wedge P5

Break down subscriptions into predicates, store and match them individually

PHASE 1: PREDICATE MATCHING

Predicate Matching Problem

- Given a set P of predicates and a publication e ,
identify all predicates p of P which evaluate to true under e .
- Example:

$e = \{\dots, (\text{price}, 5), (\text{color}, \text{white}), \dots\}$

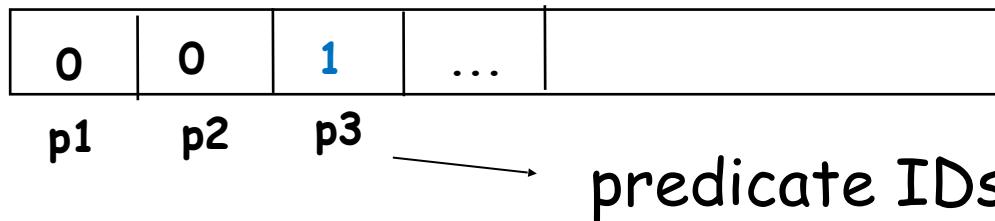
$p_1: \text{price} > 5; \quad p_2: \text{color} = \text{red}; \quad p_3: \text{price} < 7$

p_1 is false

p_2 is false

p_3 is true

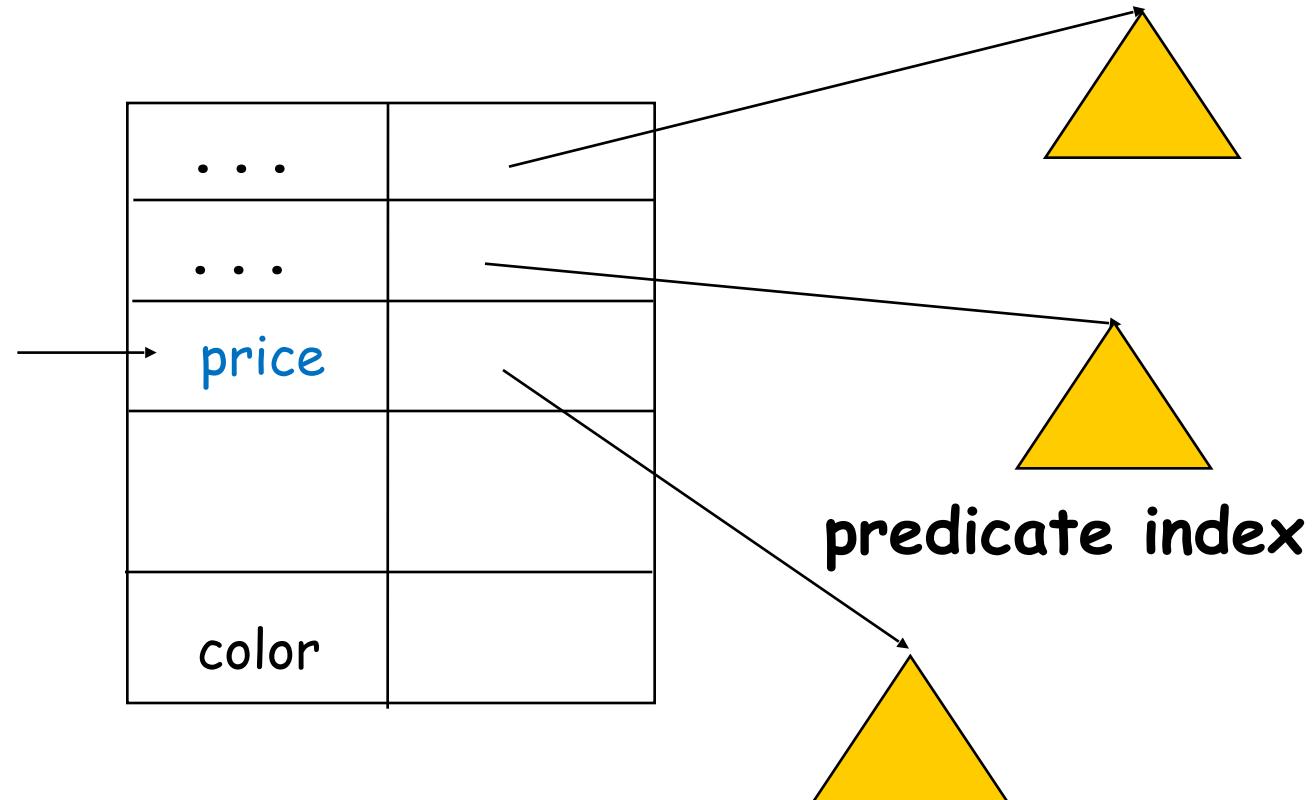
Predicate bit vector:

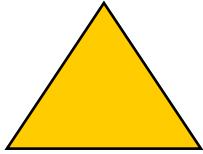


Predicate Matching: Top-level Data Structure

- Hash table on attribute name

$e = \{..., (\text{price}, 5), ...\}$



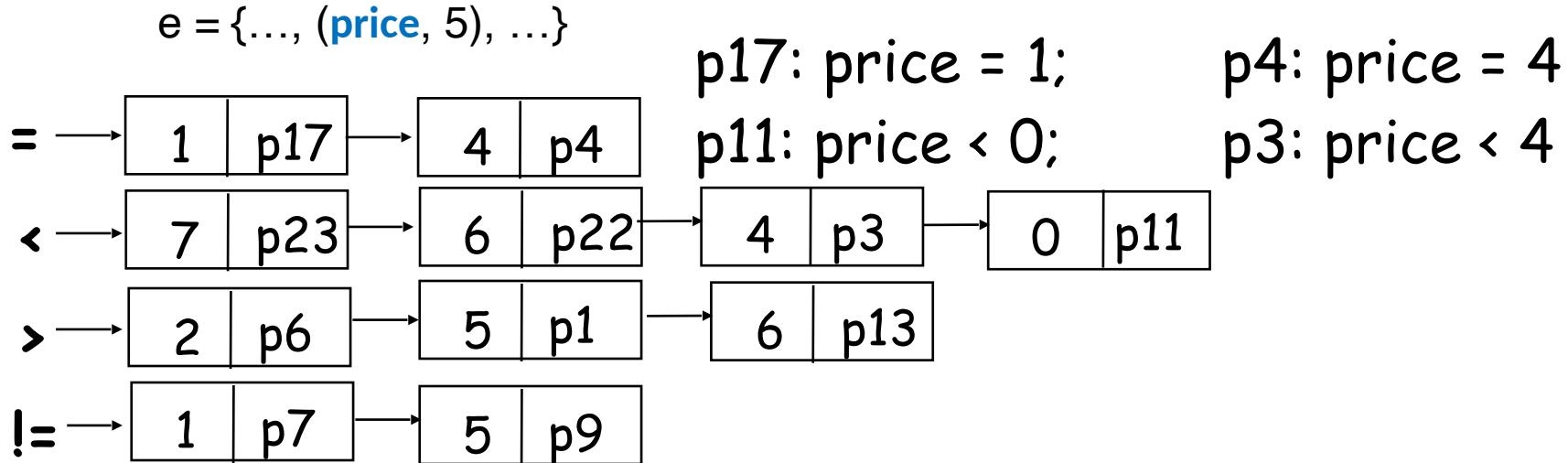


price

Predicate Index:

General Purpose Data Structure

Operators



- One ordered linked list per operator
- Insert, delete, match are $O(n)$ -operations (per attribute name in e and per operator)
- Alternatively, use a B-tree or B+-tree etc.

Finite Predicate Value Domain Types

- Countable domain types with small cardinality
 - Integer intervals
 - Collections (enums)
 - Set of tags (e.g., in XML)
- Examples
 - Price : [0, 1000], models variety of prices
 - Color, city, state, country, size, weight
 - All tags defined in a given XML schema
 - Predicate domain often context dependant, but limited in size
 - Prices of cars vs. prices of groceries

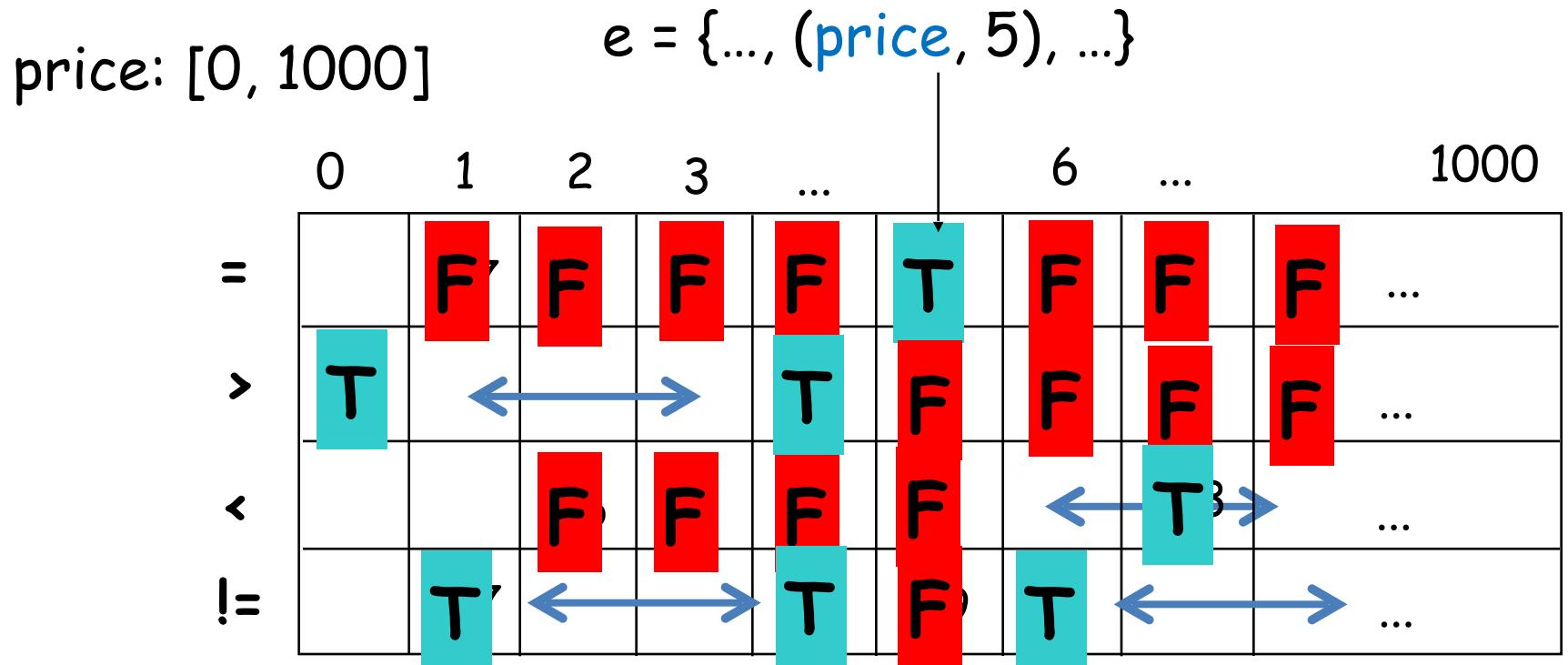
Predicate Matching For Finite Domains

price: [0, 1000] $e = \{\dots, (\text{price}, 5), \dots\}$

	0	1	2	3	...	6	...	1000
=		p17				p4		...
>			p6			p1	p13	...
<		p11				p3		...
!=			p7			p9		...

p1: price > 5; p3: price < 5; p4: price = 5; p9: price != 5

Predicate Matching Symmetries

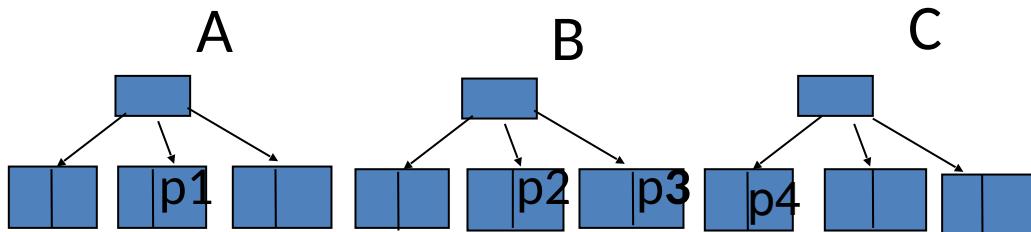


PHASE 2: SUBSCRIPTION MATCHING

Subscription Matching: Counting Algorithm

- Subscriptions consist of a set of predicates
 - S1: $(2 < A < 4) \& (B = 6) \& (C > 4)$ $\Rightarrow p1: (2 < A < 4), p2: (B = 6), p3: (C > 4)$
 - S2: $(2 < A < 4) \& (C = 3)$ $\Rightarrow p1: (2 < A < 4), p4: (C = 3)$
- Subscription matches the event if **all** its predicates are satisfied
- **Idea:** Count the number of satisfied predicates per subscription and compare with actual number

Counting Algorithm: Subscription Insertion



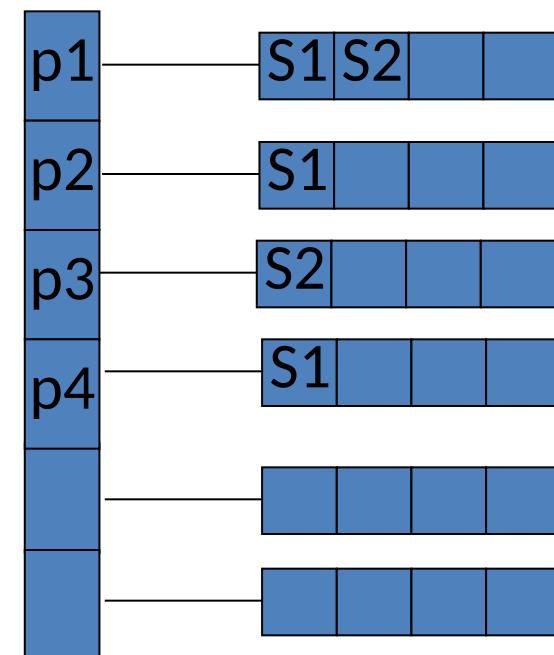
preds-per-sub hit count

S1	3
S2	2

S1	0
S2	0

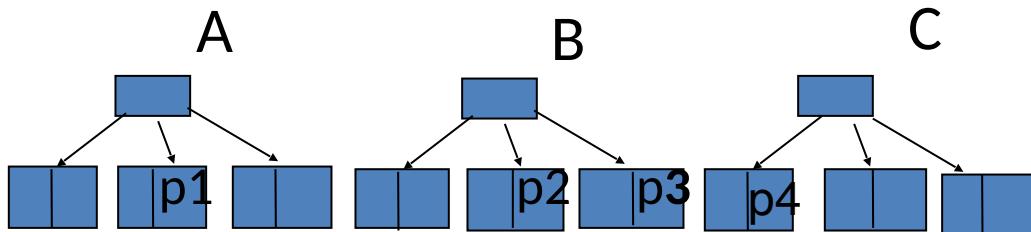
S1: p1, p2, p4
S2: p1, p3

Predicate vector



predicate-to-subscription association

Counting Algorithm: Subscription Insertion



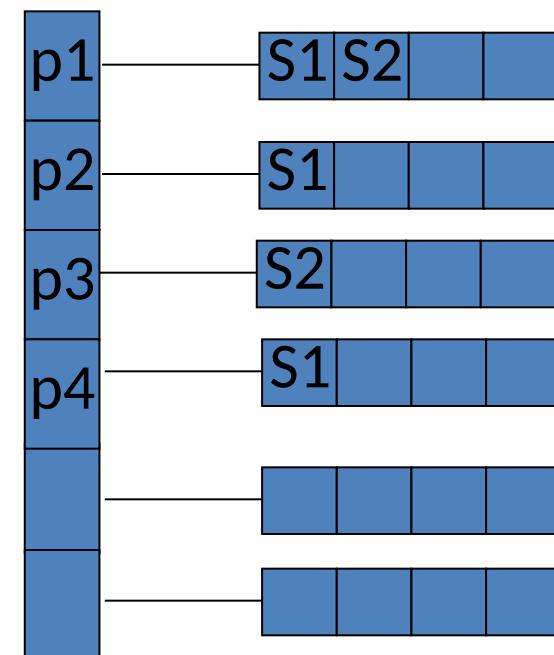
preds-per-sub hit count

S1	3
S2	2

S1	0
S2	0

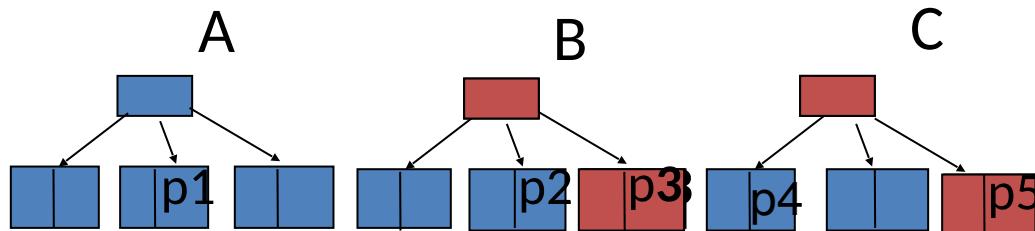
S1: p1, p2, p4
S2: p1, p3
S3: p3, p5

Predicate vector



predicate-to-subscription association

Counting Algorithm: Subscription Insertion



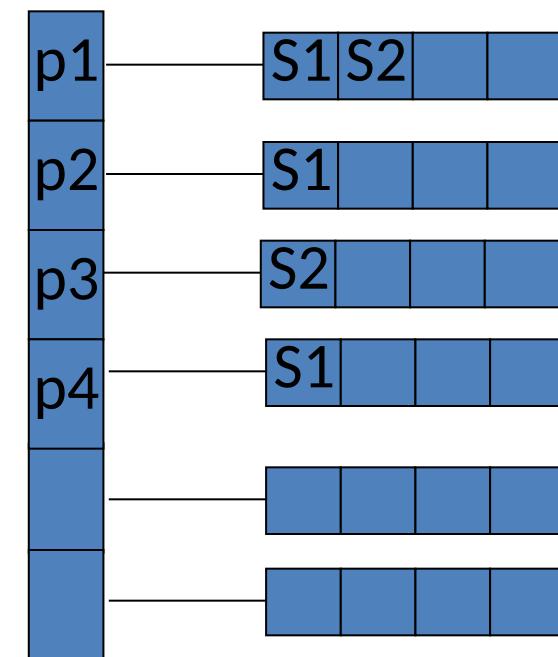
preds-per-sub

S1	3
S2	2

hit count

S1	0
S2	0

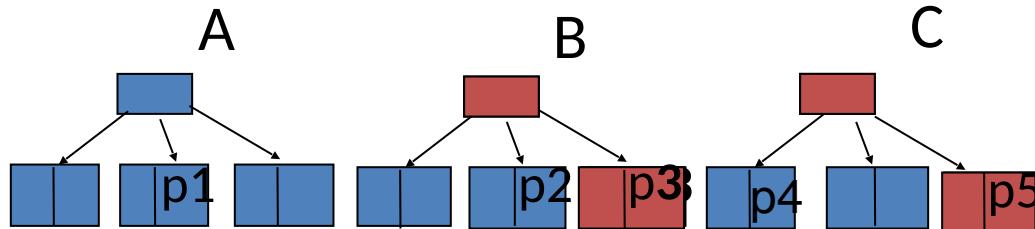
Predicate vector



S1: p1, p2, p4
S2: p1, p3
S3: p3, p5

predicate-to-subscription association

Counting Algorithm: Subscription Insertion



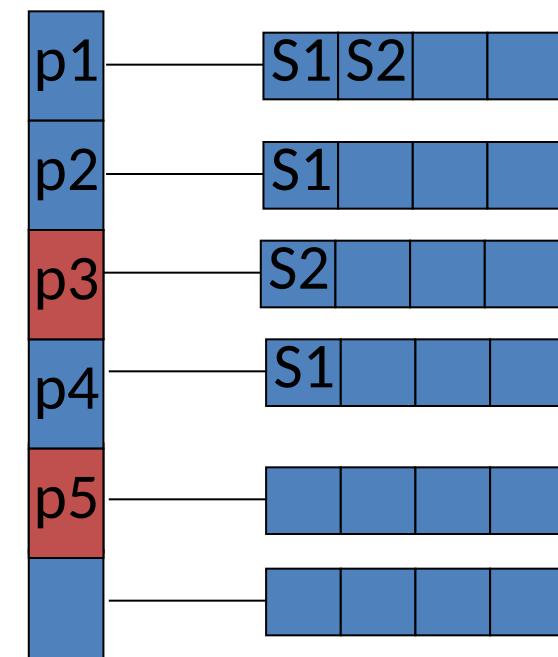
preds-per-sub

S1	3
S2	2

hit count

S1	0
S2	0

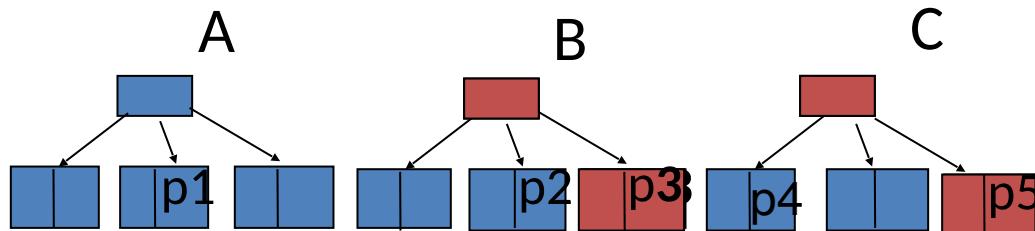
Predicate vector



S1: p1, p2, p4
S2: p1, p3
S3: p3, p5

predicate-to-subscription association

Counting Algorithm: Subscription Insertion

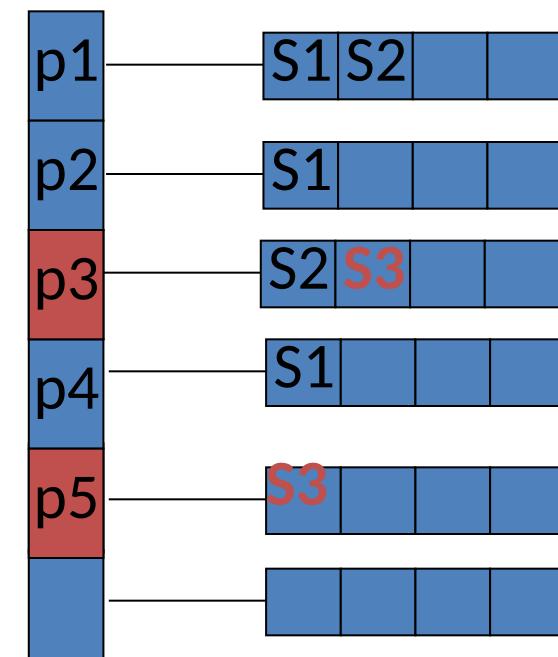


preds-per-sub hit count

S1	3
S2	2

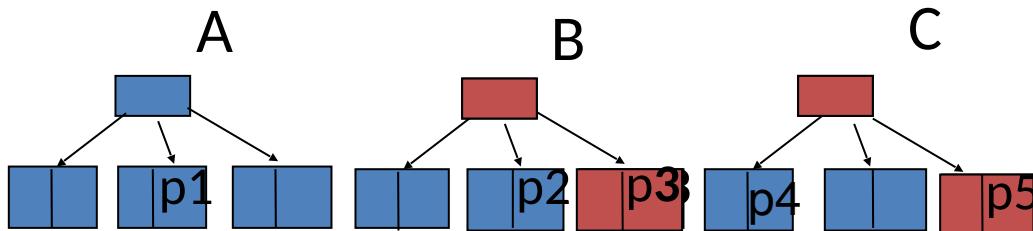
S1	0
S2	0

Predicate vector



S1: p1, p2, p4
S2: p1, p3
S3: p3, p5

Counting Algorithm: Subscription Insertion



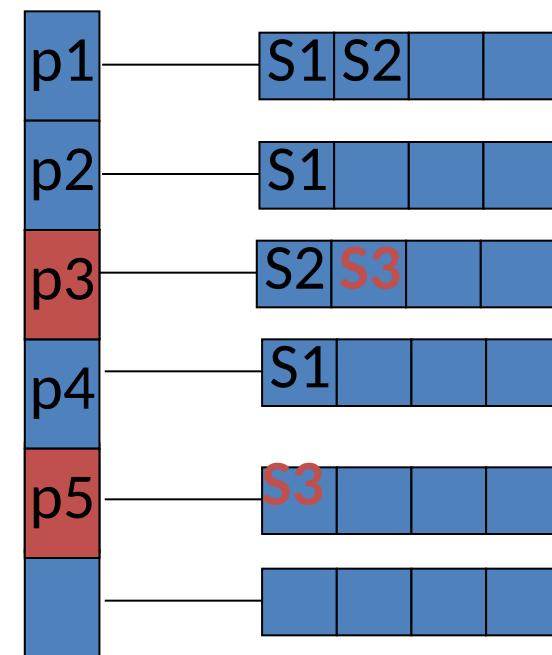
preds-per-sub hit count

S1	3
S2	2
S3	2

S1: p1, p2, p4
S2: p1, p3

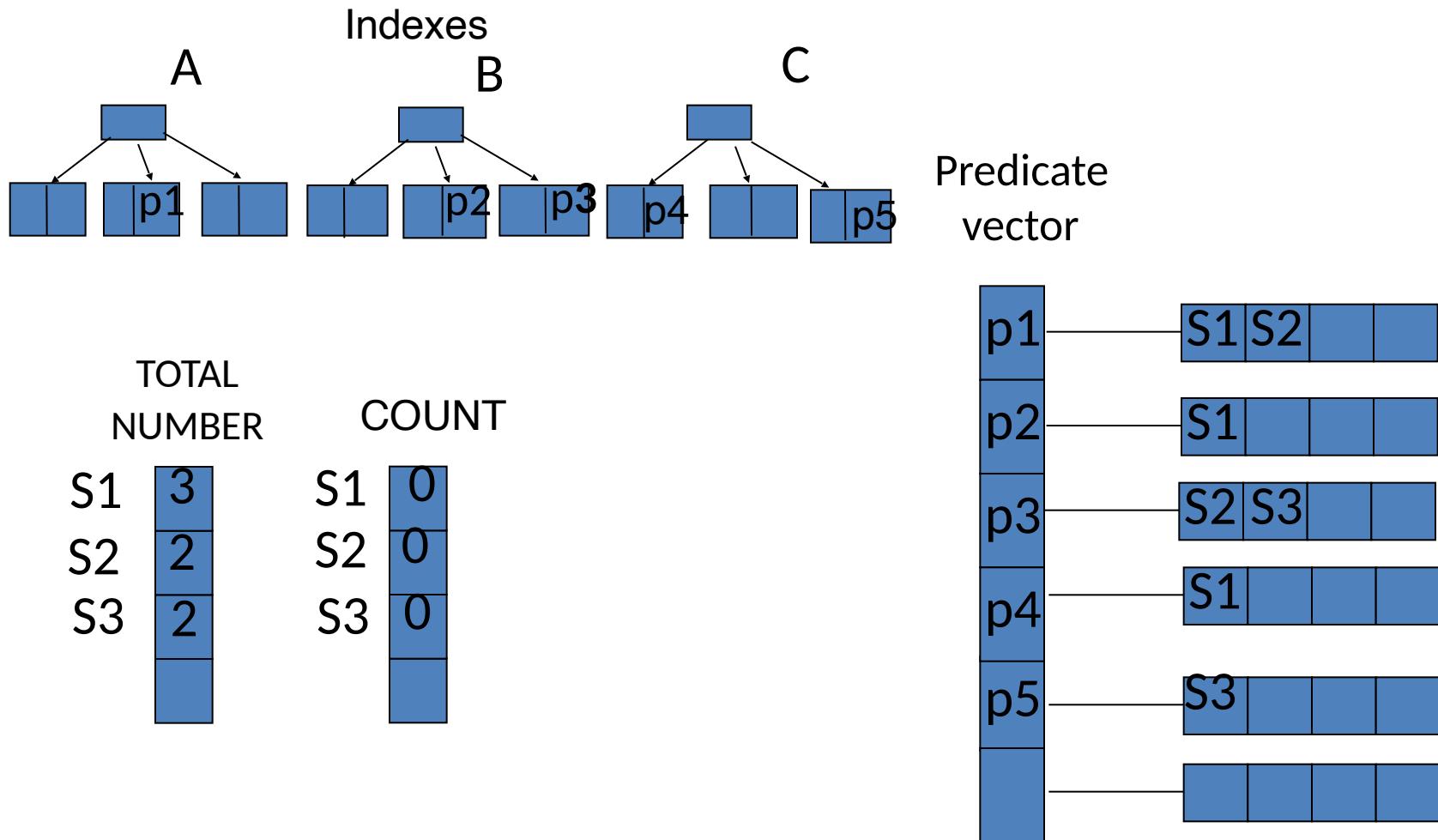
S3: p3, p5

Predicate vector

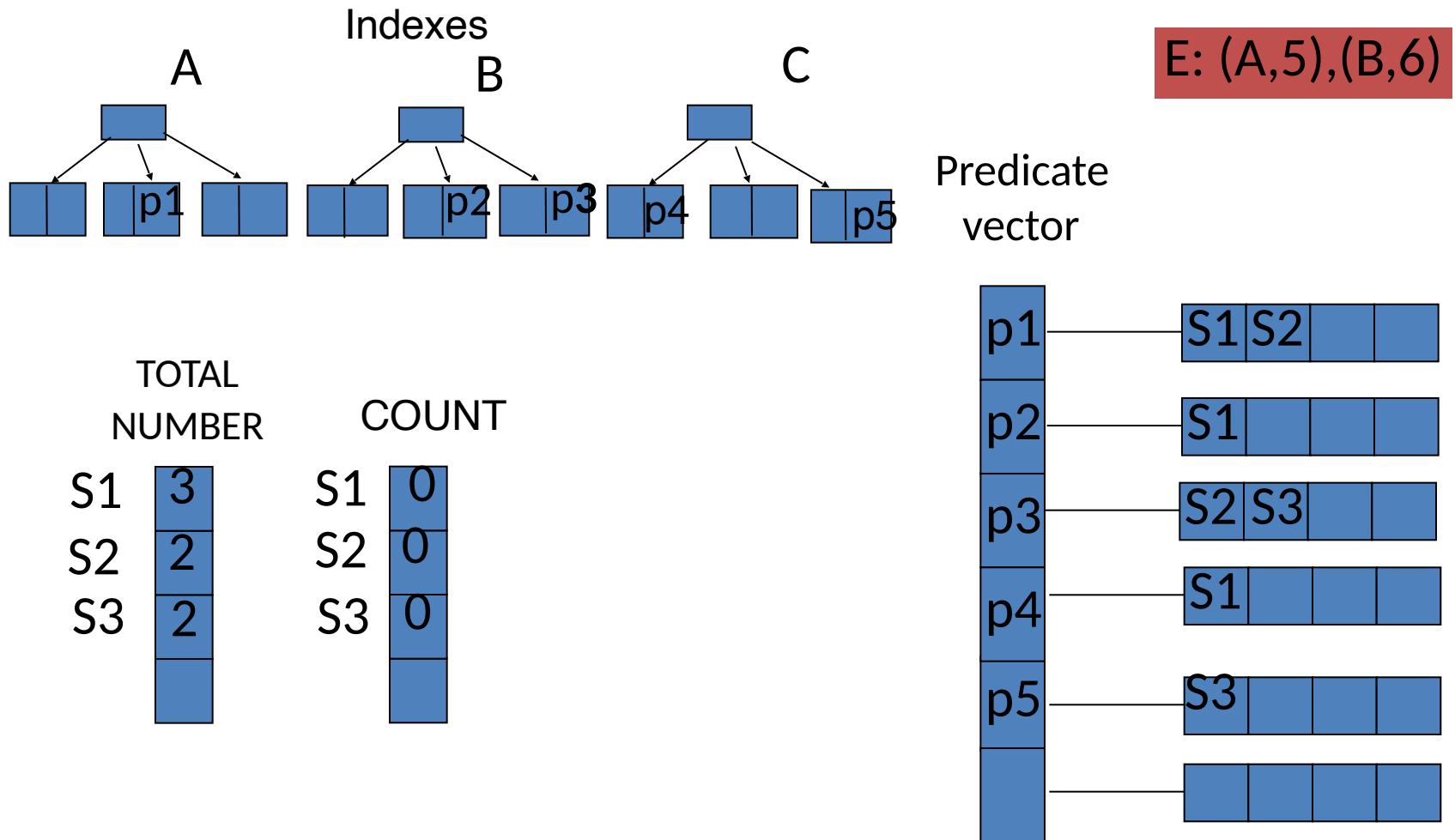


predicate-to-subscription association

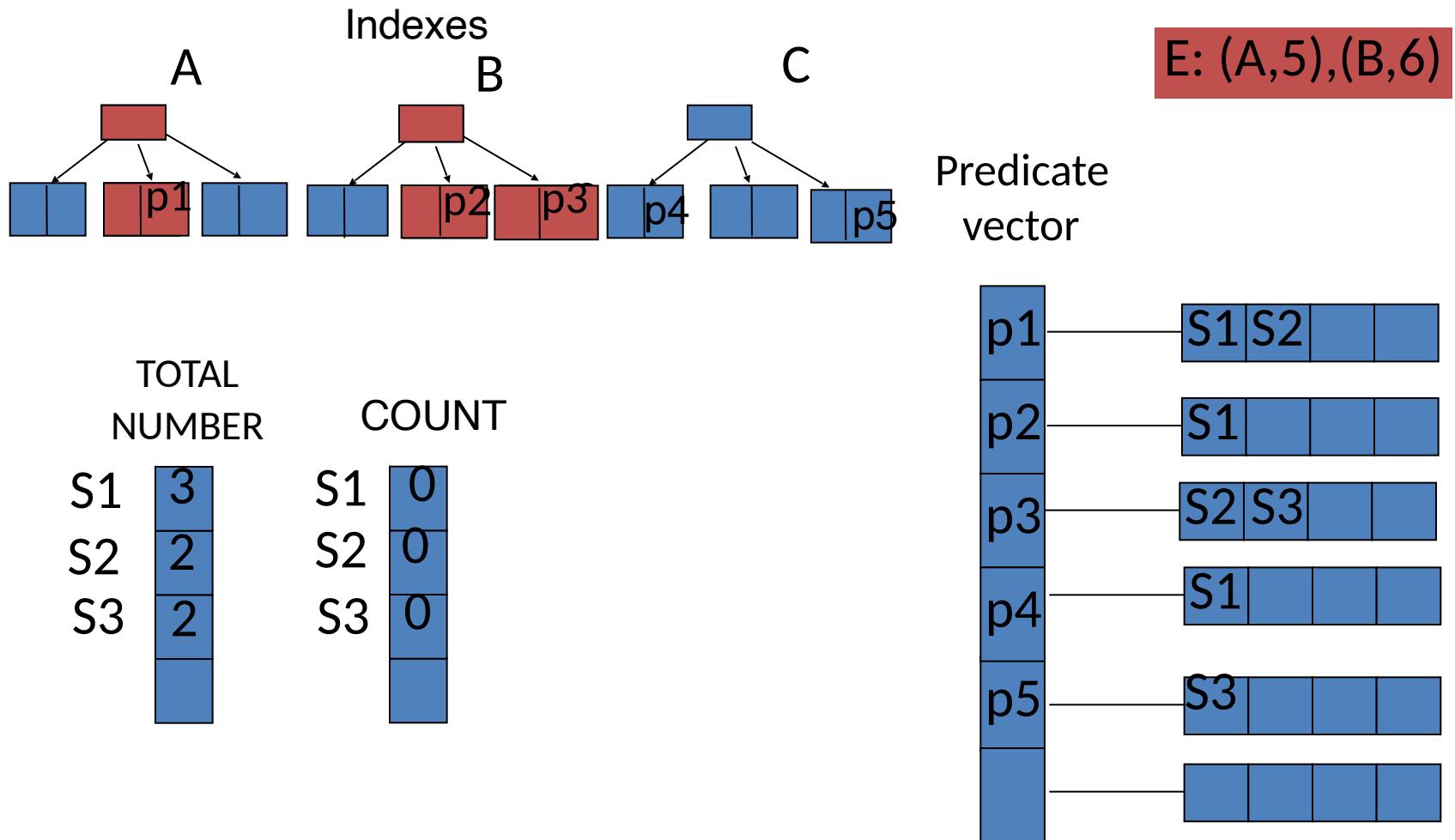
Counting Algorithm: Event Matching



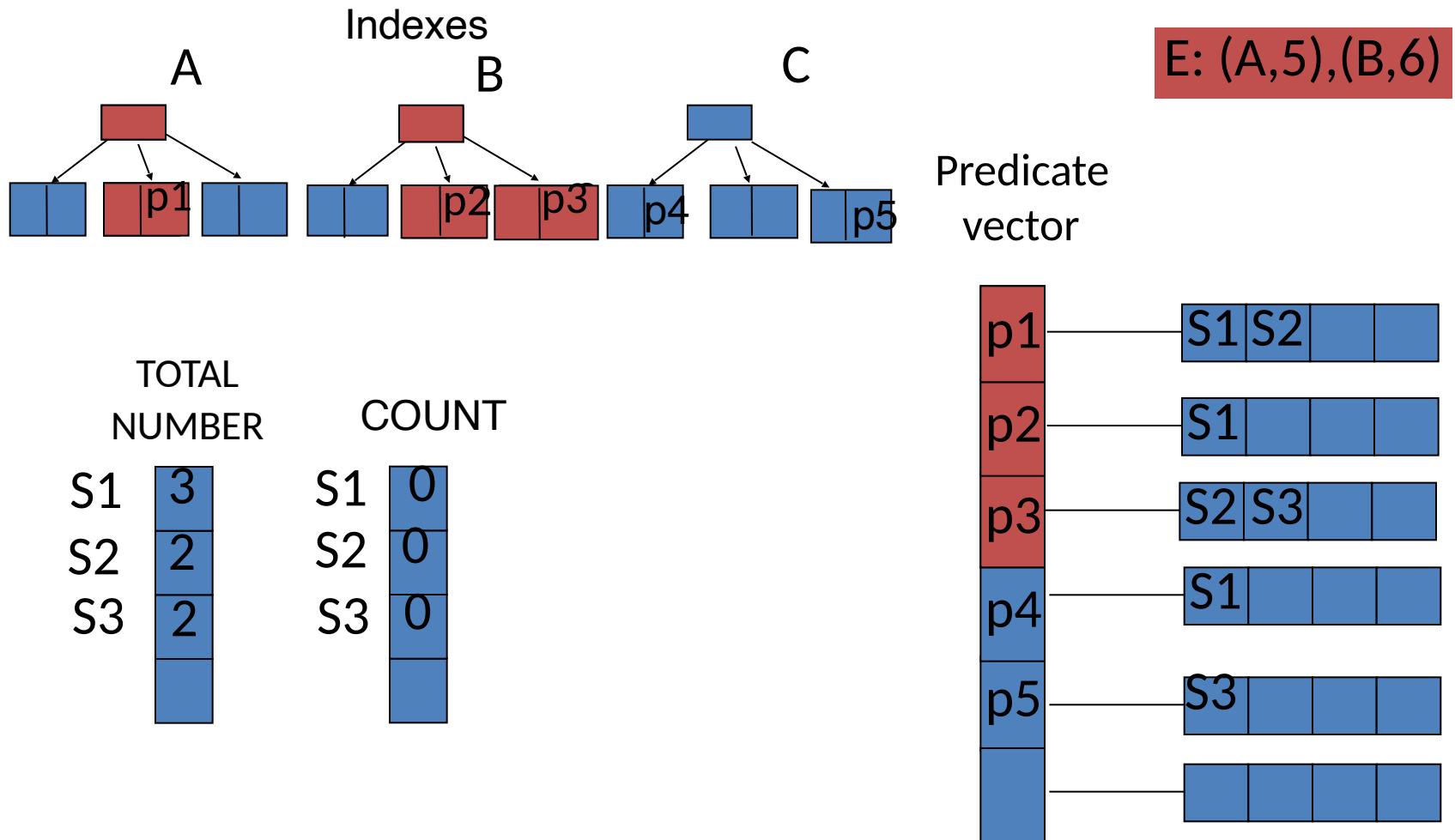
Counting Algorithm: Event Matching



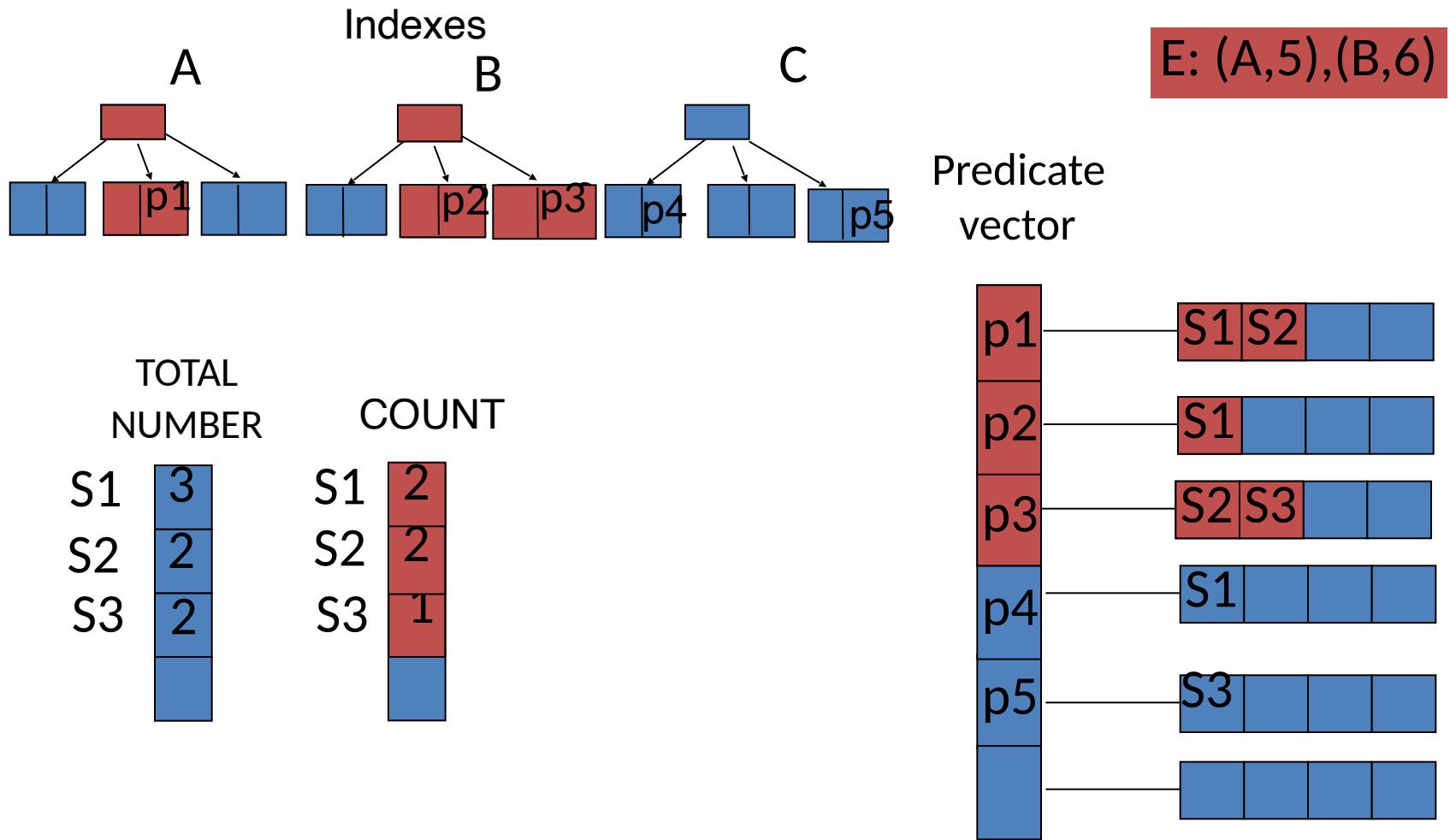
Counting Algorithm: Event Matching



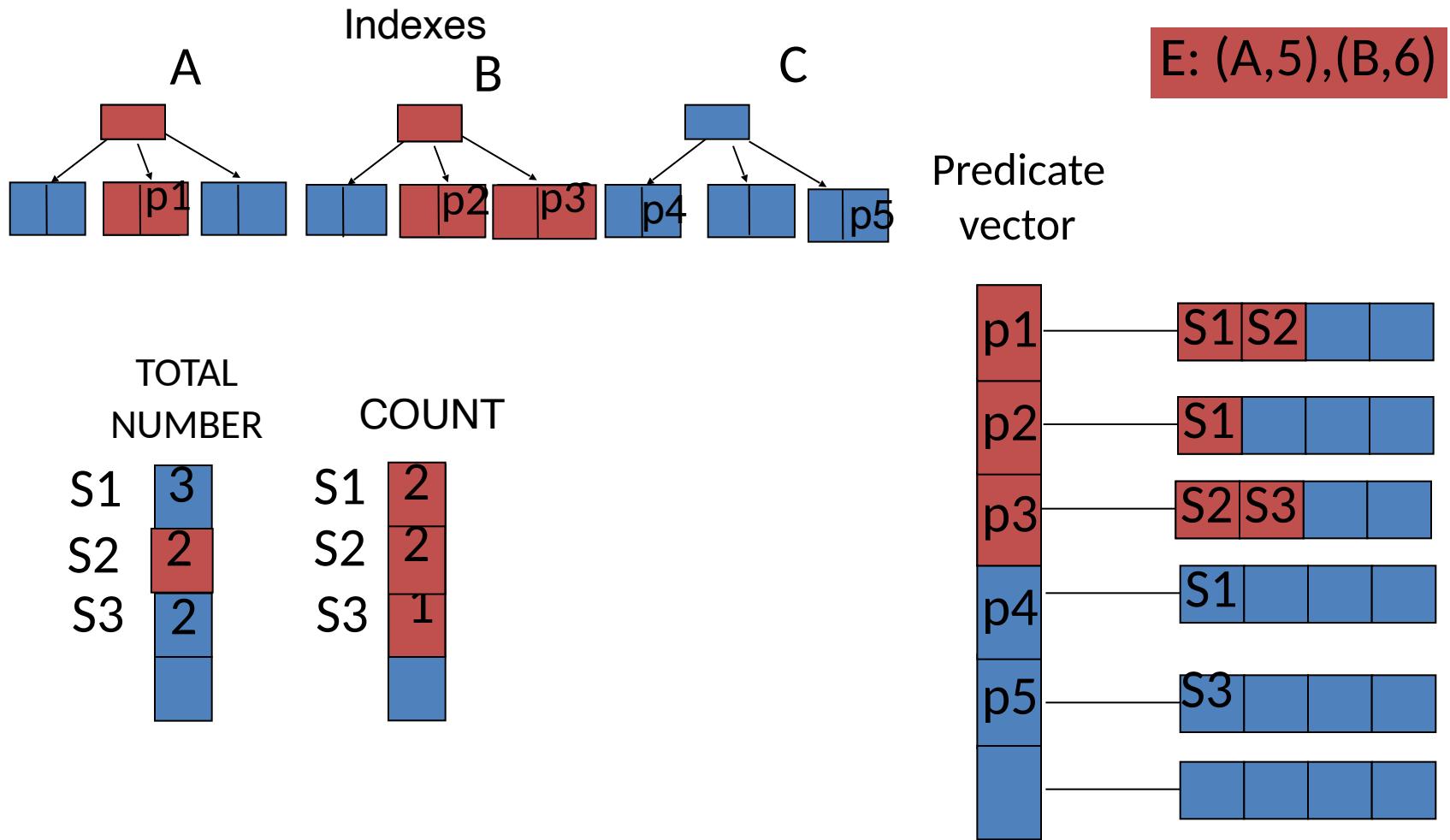
Counting Algorithm: Event Matching



Counting Algorithm: Event Matching



Counting Algorithm: Event Matching



Matching Models Summary

- **Matching model:** The language used to express pubs and subs in order to match them.
- **Matching problem:** Given a publication, p , and a set of subscriptions, S , determine all subscriptions, $s \in S$, that match e .
- **Topic-based vs. Content-based matching**
- **Tradeoff:** computation vs. communication
- **Two-phase matching** is an efficient technique for quickly finding matches in content-based model
- **Counting algorithm**

ROUTING MODELS

Pub/Sub Architecture

- Pub/Sub Middleware
 - Presence of a dedicated **pub/sub middleware**
 - Middleware in charge of receive all operations
 - Store subscriptions
 - Perform **publication matching and dissemination**
- Infrastructure-less
 - No dedicated middleware
 - Clients (publishers and subscribers) perform the tasks
 - **Peer-to-peer**

Centralized vs. Distributed

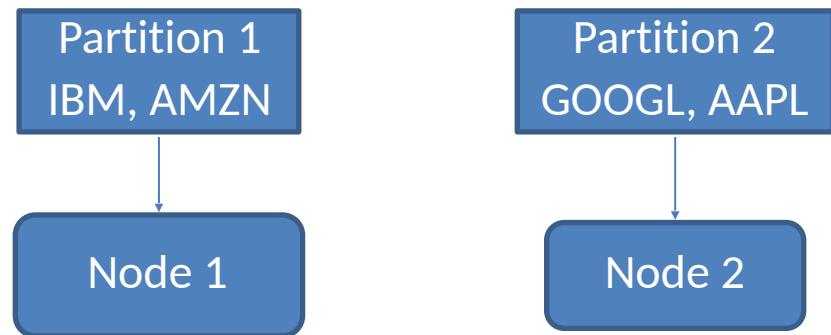
- Centralized
 - A single pub/sub entity
 - Client-server interactions
 - CORBA, JMS, ...
- Decentralized
 - Matching and dissemination tasks **distributed**
 - Either, multiple pub/sub entities (called **brokers**)
 - Or, P2P model without brokers (eg. DHT)

RENDEZVOUS-BASED ROUTING

Rendezvous-based Routing

- The “publication space” is partitioned
- Each node is in charge of a partition
- Publications and subscriptions belonging to a partition are sent to the corresponding node

Example for topic-based:

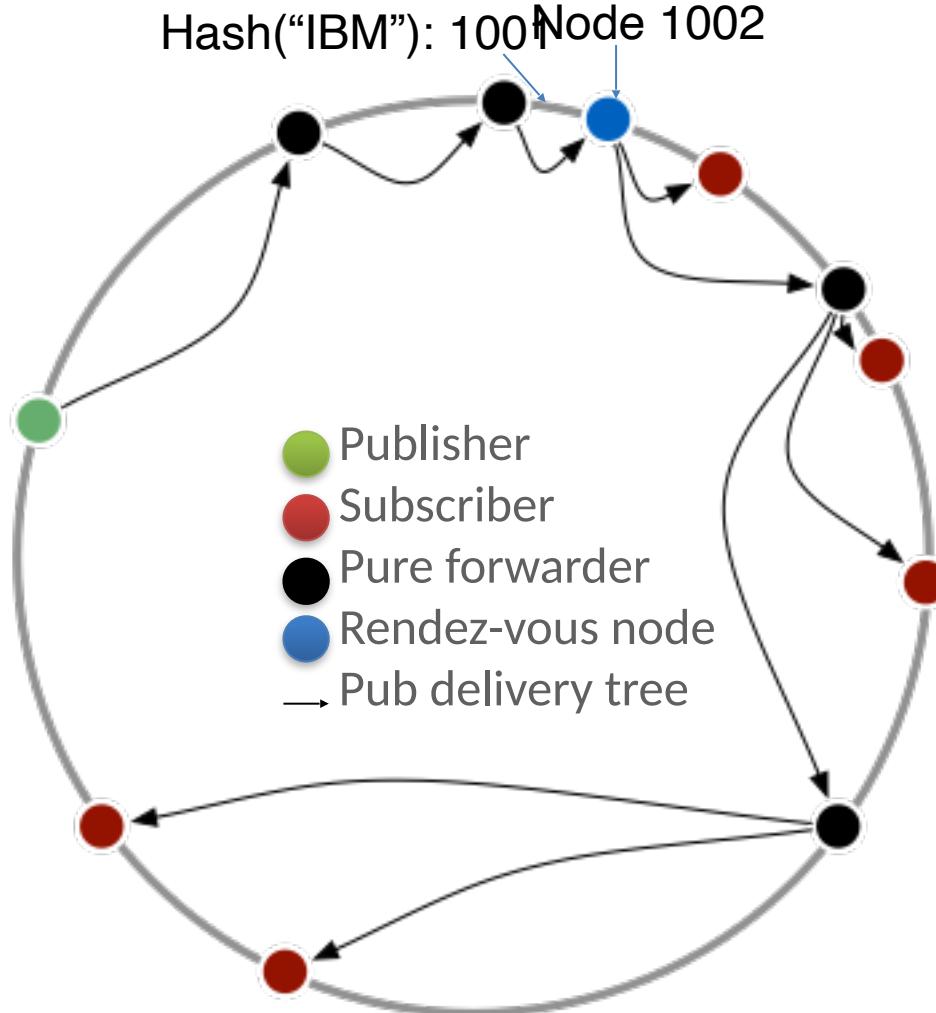


P2P Pub/Sub: Scribe (2001)

- Infrastructure-less, distributed, pub/sub network built on top of a P2P network
- Relies on a structured P2P DHT (Pastry)
- Supports topic-based pub/sub API on top of Pastry
- Topics are hashed in the DHT

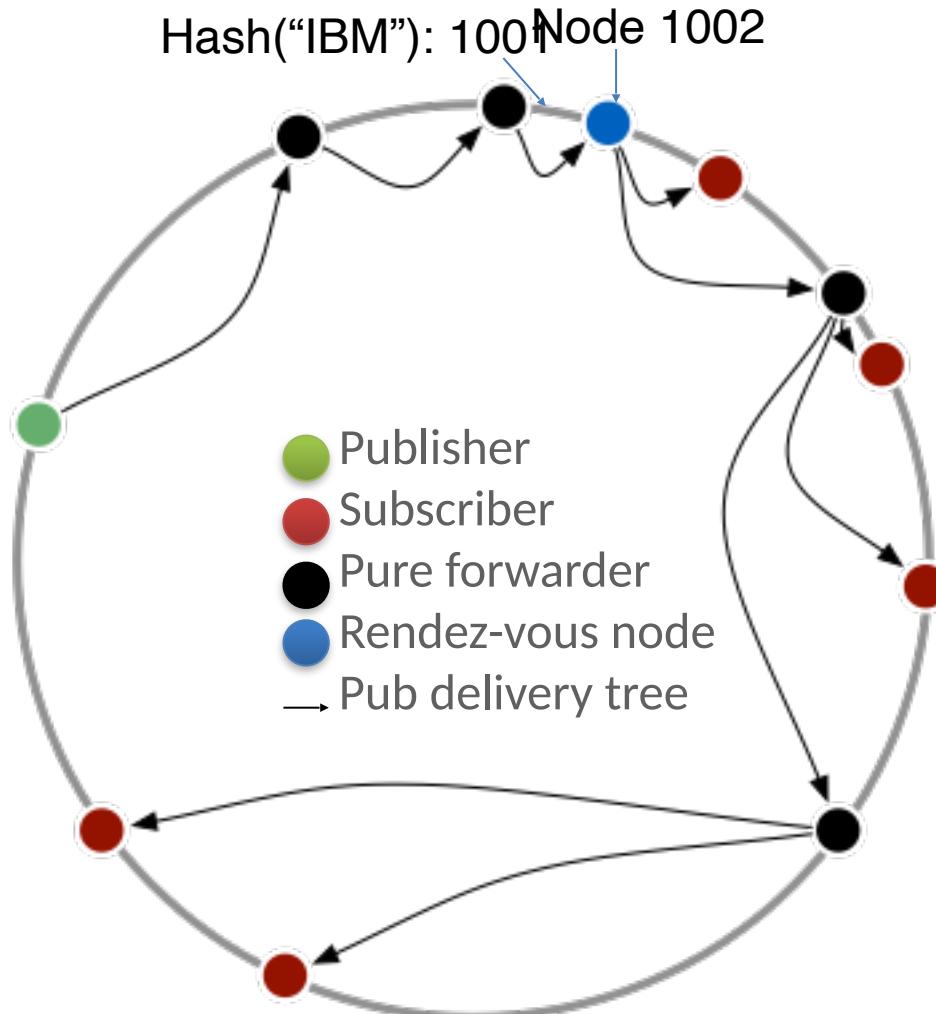
Scribe Routing

Hash("IBM"): 1001 Node 1002



Scribe Routing

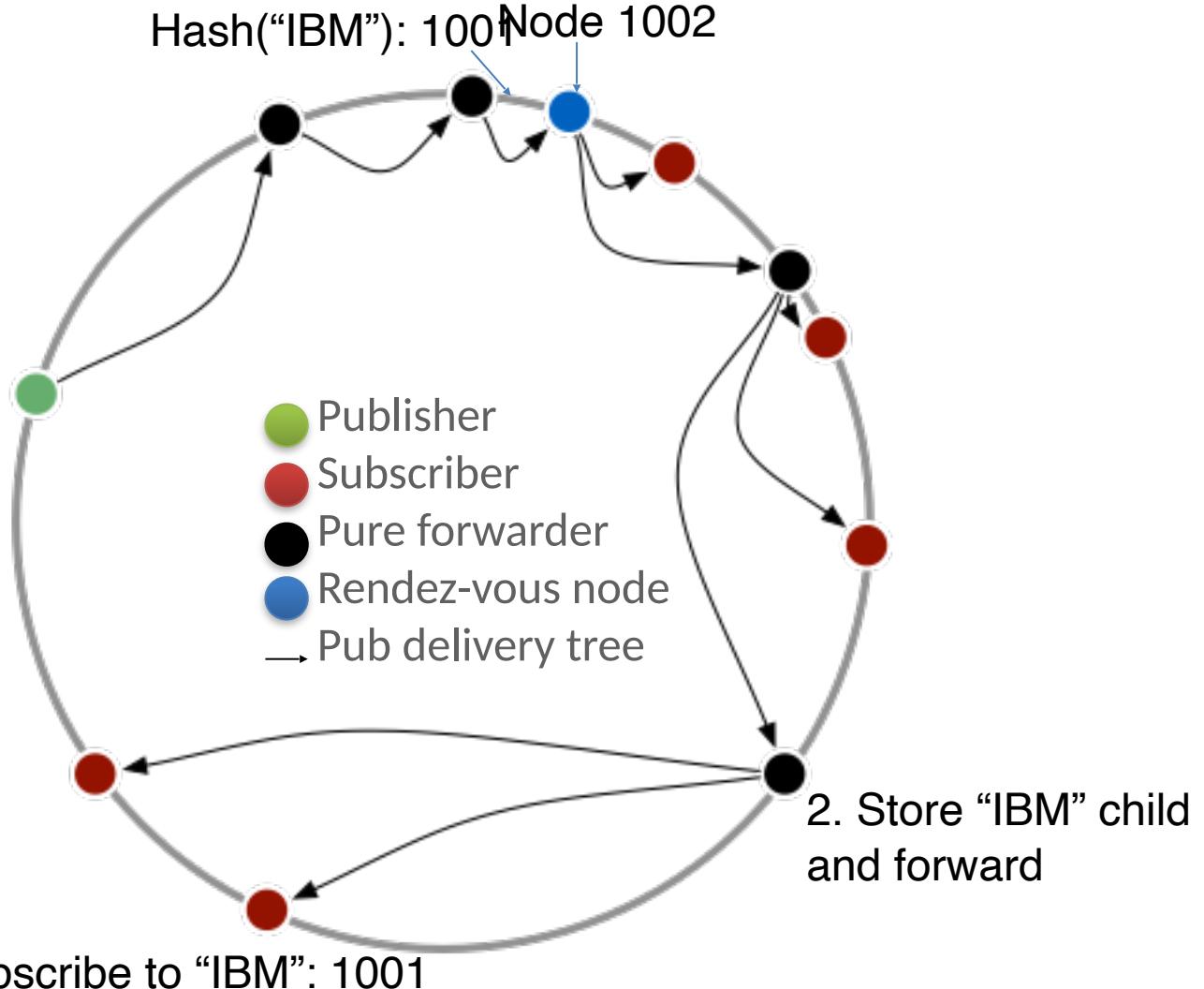
Hash("IBM"): 1001 Node 1002



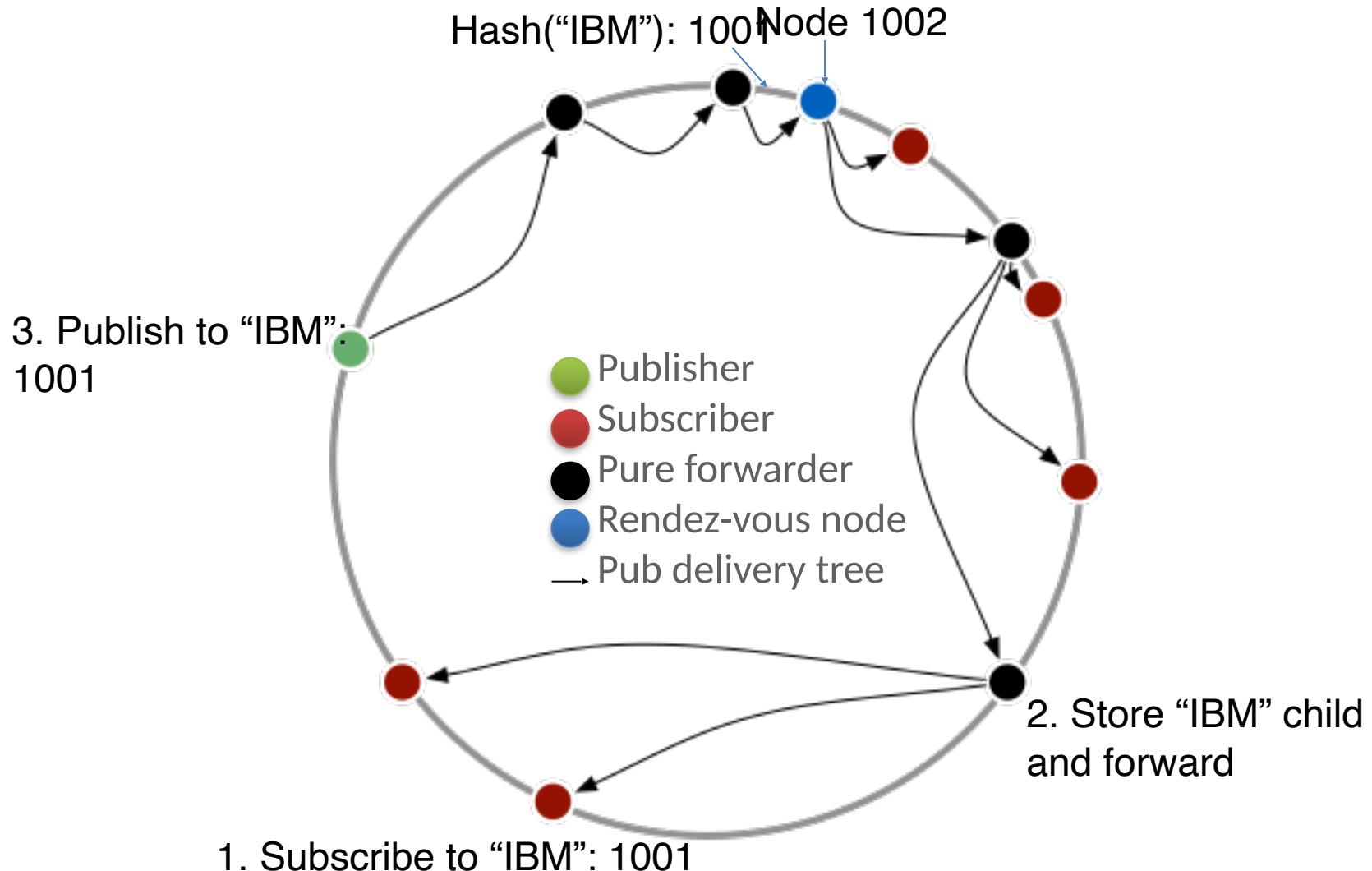
1. Subscribe to "IBM": 1001

Scribe Routing

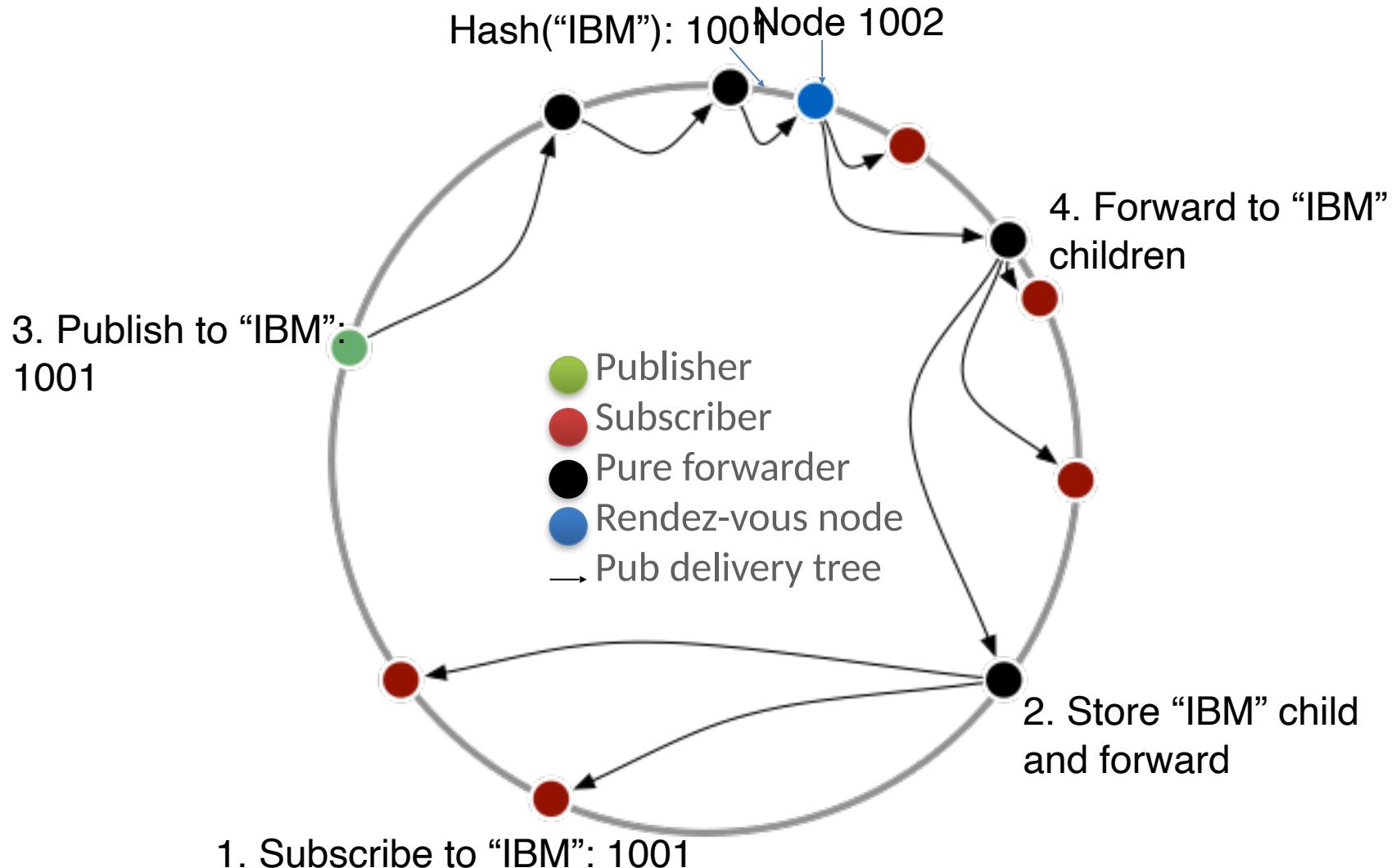
Hash("IBM"): 1001 Node 1002



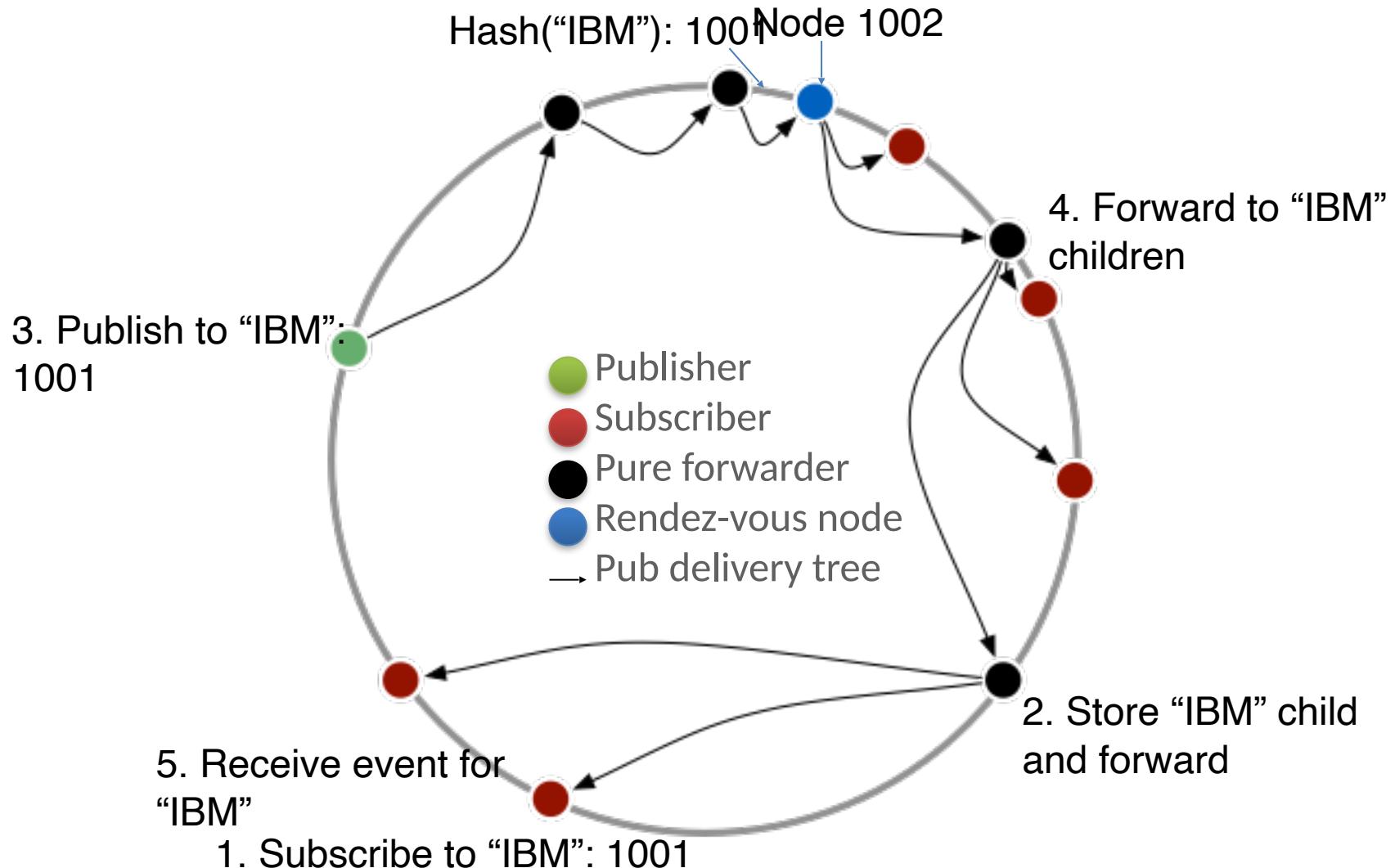
Scribe Routing



Scribe Routing



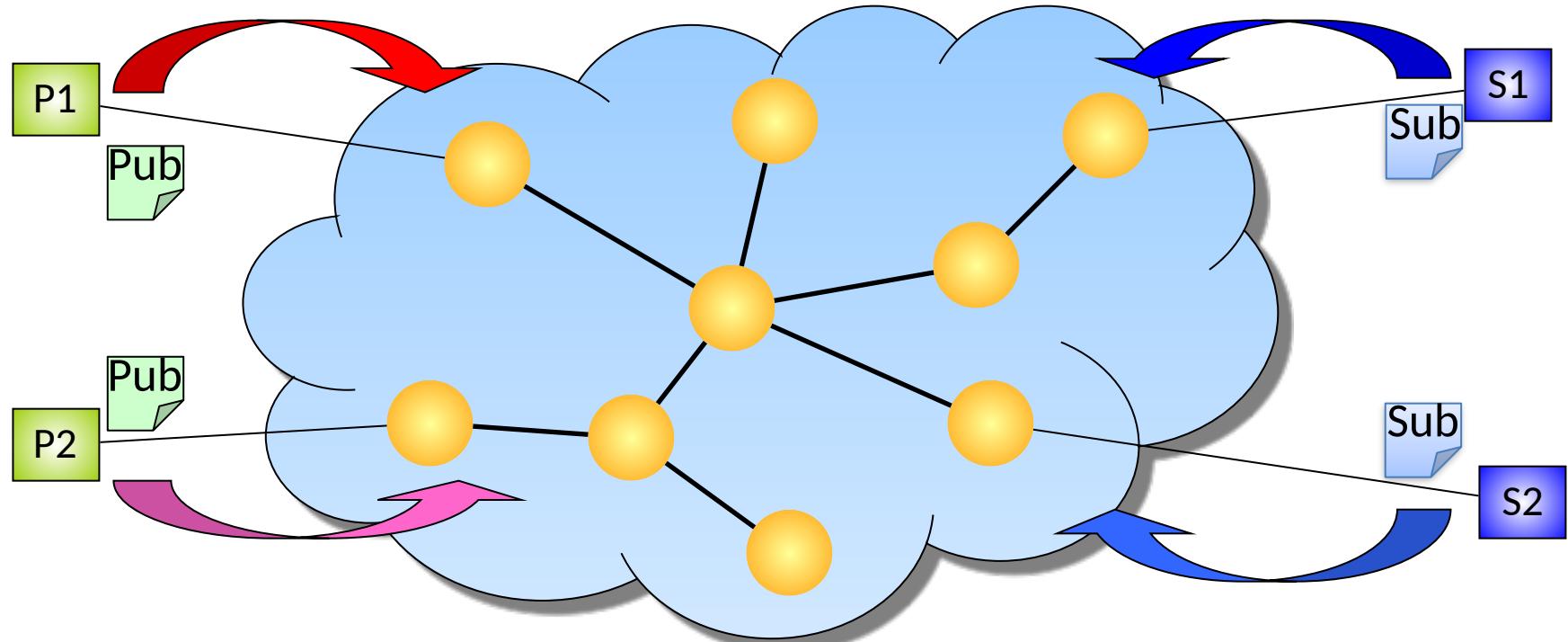
Scribe Routing



OVERLAY-BASED ROUTING

Overlay Broker Networks

- Messages flow based on overlay links
- Clients connect to any one broker

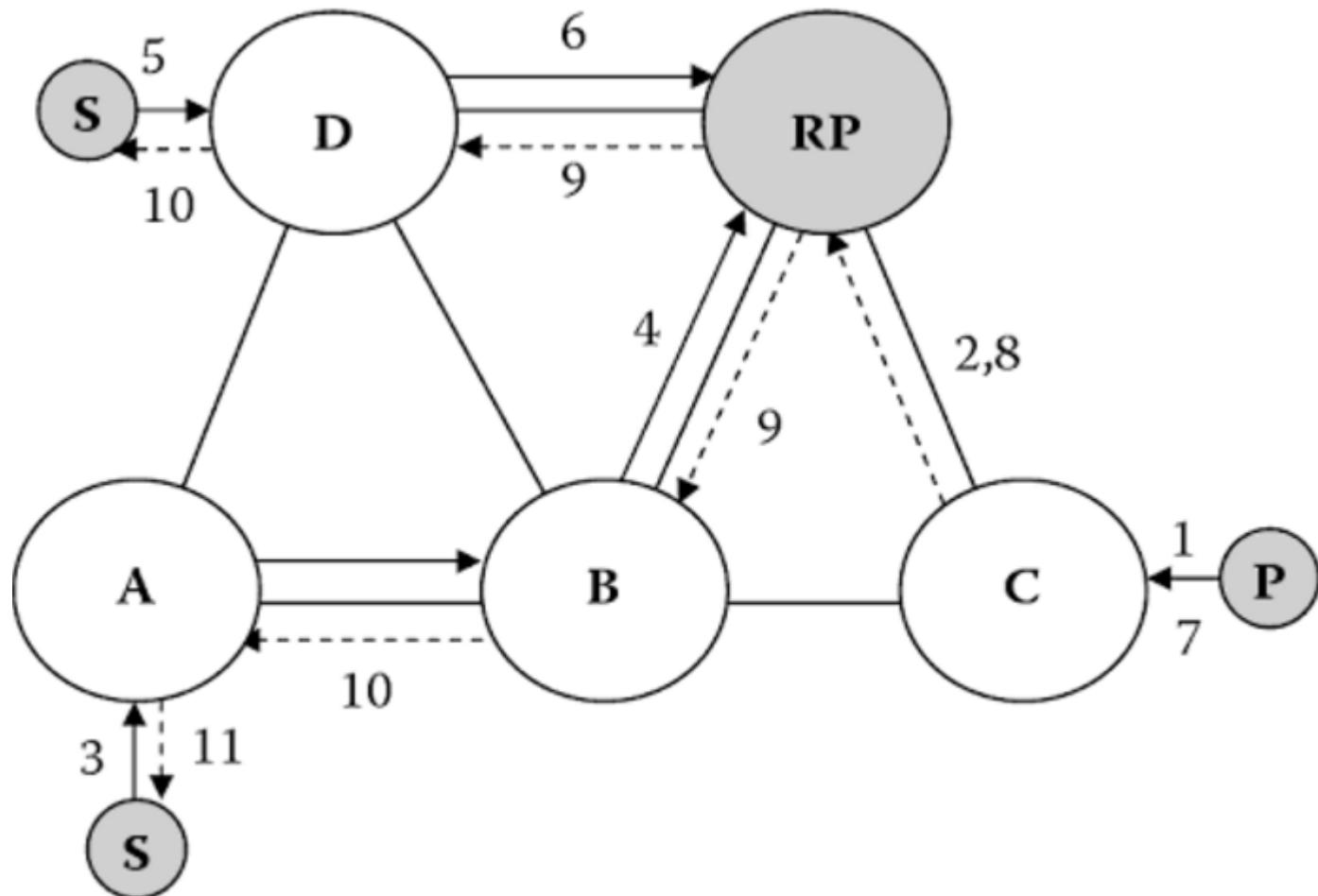


Matching and dissemination are performed inside the overlay

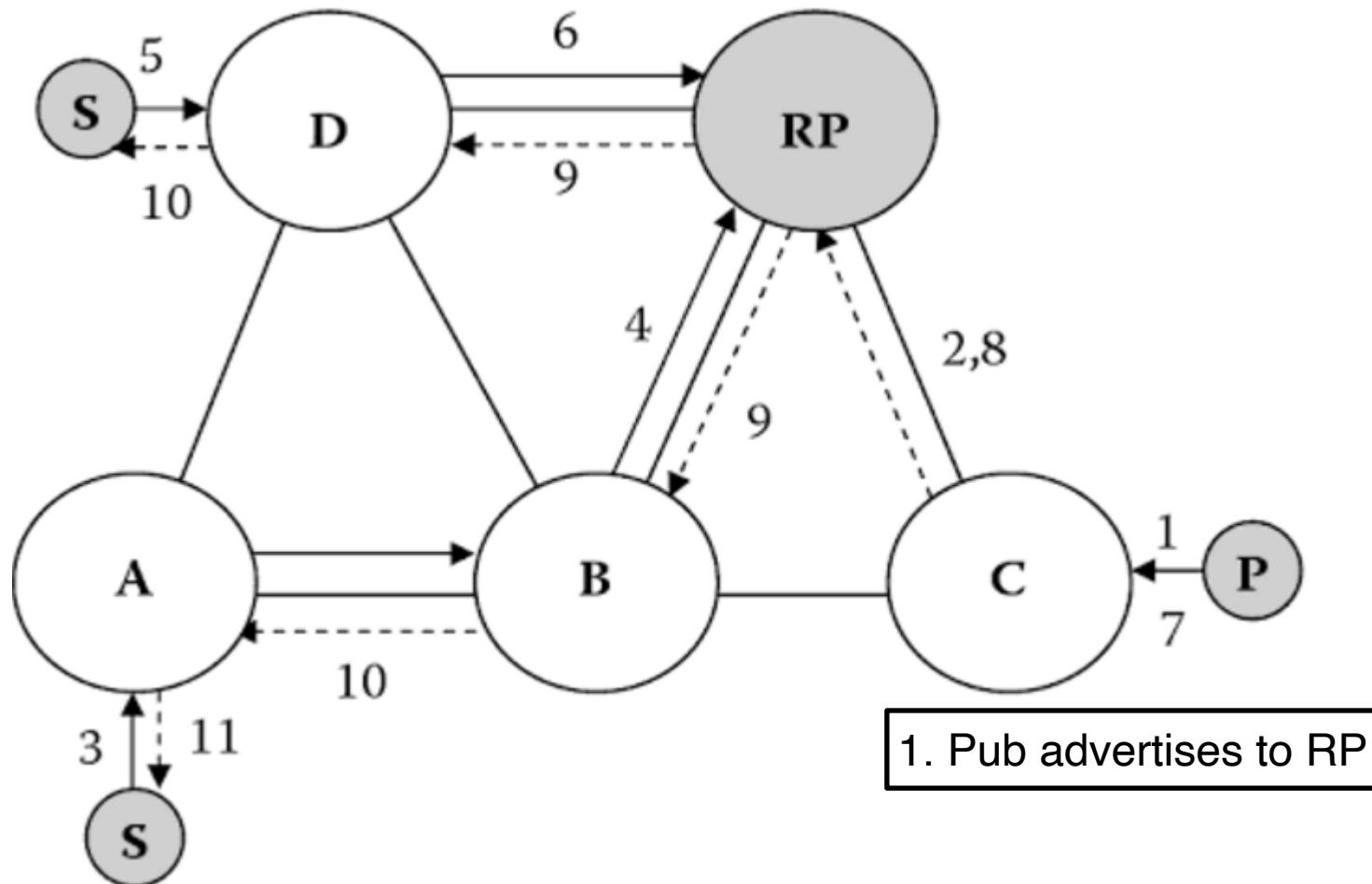
RDV-based Overlay: Hermes (2004)

- Supports **content-based** matching w/ ads
- One broker is the **rendezvous point**
- Each broker knows how to reach the rendezvous point and subscribers
- The rendezvous point performs **matching**
- The rendezvous point disseminate the pub towards matching subs.

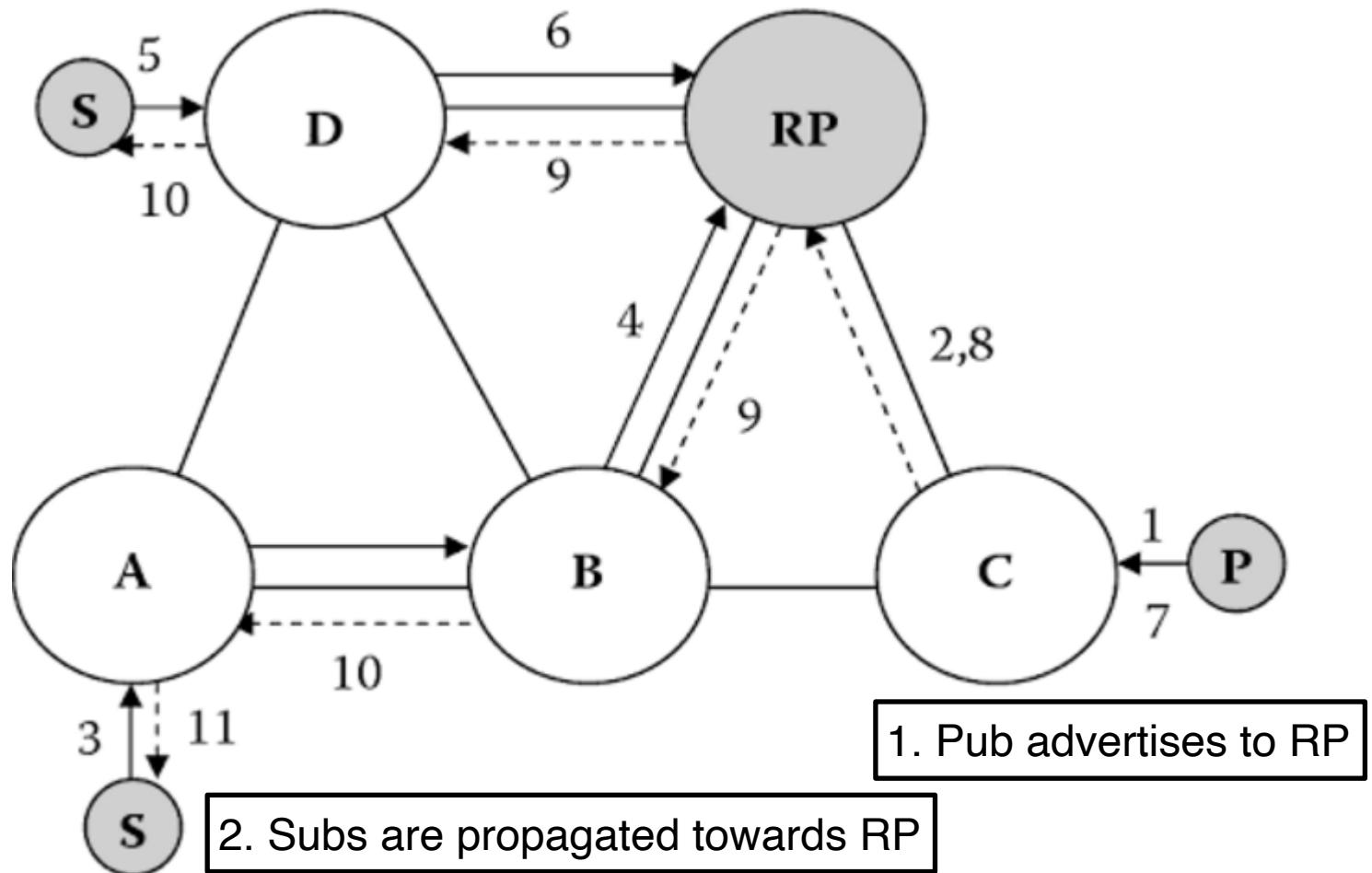
Hermes Routing



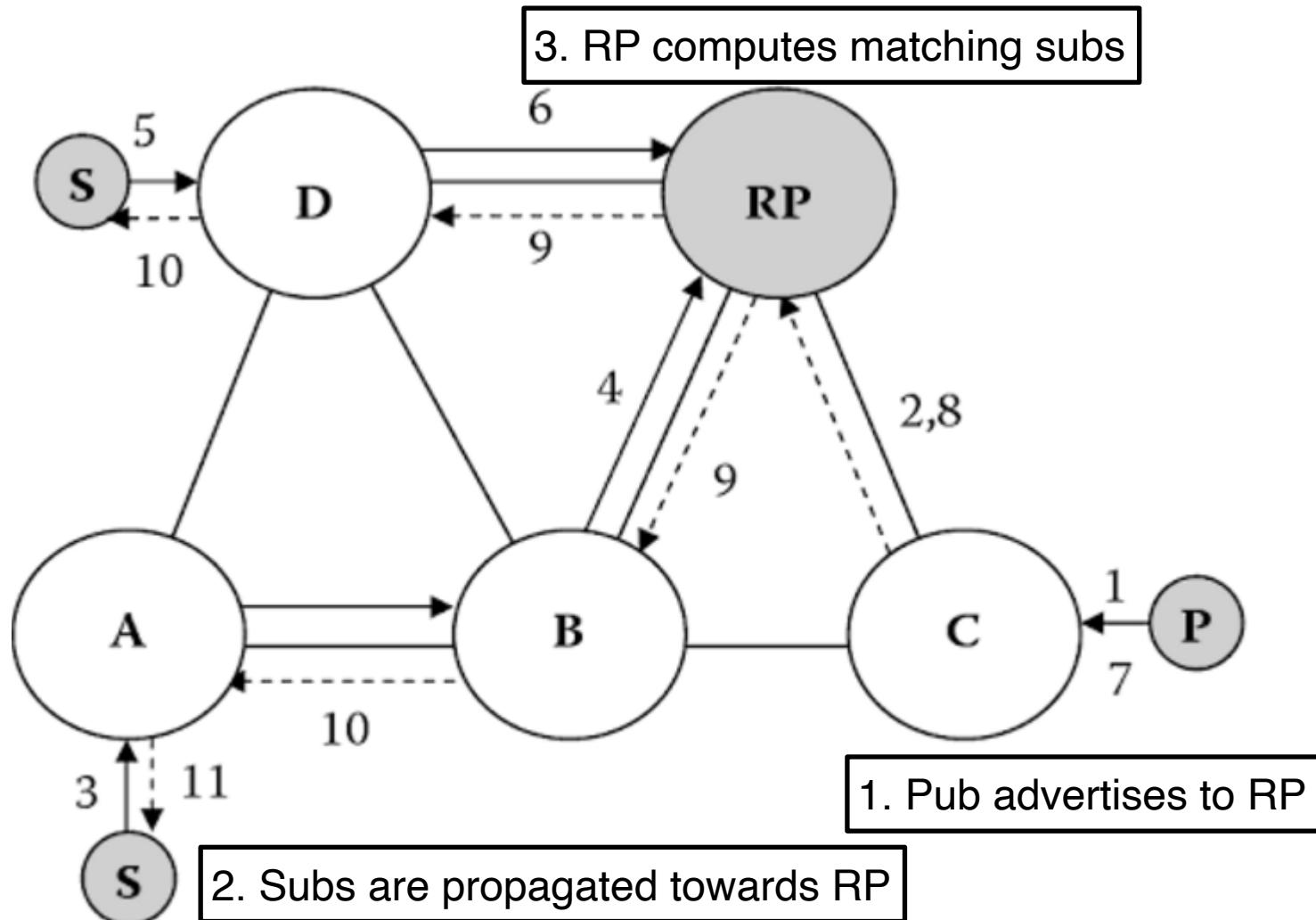
Hermes Routing



Hermes Routing



Hermes Routing



Possible Optimizations

- More than one rendezvous point
 - Use space partitioning
 - Increases state complexity of forwarding brokers
- **Bloom filters** using Link IDs
 - Reduces state complexity of forwarding brokers
 - Introduces false positive dissemination
 - Increases processing time at the RP

Bloom filters

- Efficient data structure for storing elements
- Represented as a **bit vector** with **fixed size**
- Elements can be inserted into the Bloom filter
- Can verify if an element belongs to the BF
- CANNOT obtain a list of all elements in the BF
- False positive: can wrongly identify an element as being part of the Bloom Filter

Bloom Filter Operations

Start with an m bit array, filled with 0s.

B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Hash each item x_j in S k times. If $H_i(x_j) = a$, set $B[a] = 1$.

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

To check if y is in S , check B at $H_i(y)$. All k values must be 1.

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Possible to have a false positive; all k values are 1, but y is not in S .

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Example of Bloom Filter

Start with an **16** bit array, filled with 0s.

B

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Hash1("id1") = 7, Hash2("id1") = 2, ... Hash8("id1") = 5

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Hash1("id2") = 7, Hash2("id2") = 11, Hash3("id2") = 4

B

0	1	0	1	1	0	1	0	0	1	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

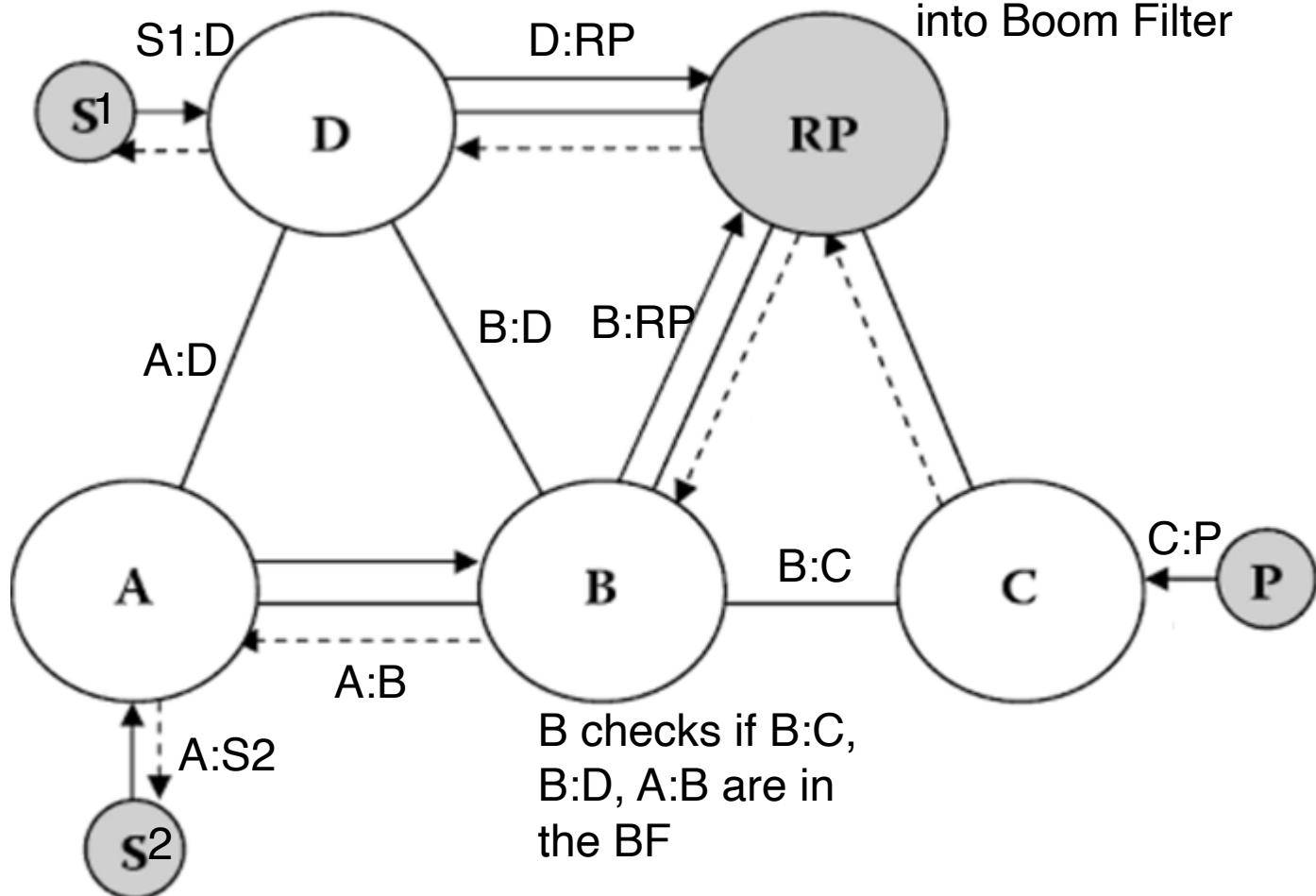
Hash1("id3") = 5, Hash2("id3") = 5,... Hash8("id3") = 7

B

0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

8 hash functions

Link IDs encoding in Bloom Filters (LIPSIN 2009)



Filtering-Based Routing: PADRES (2003)

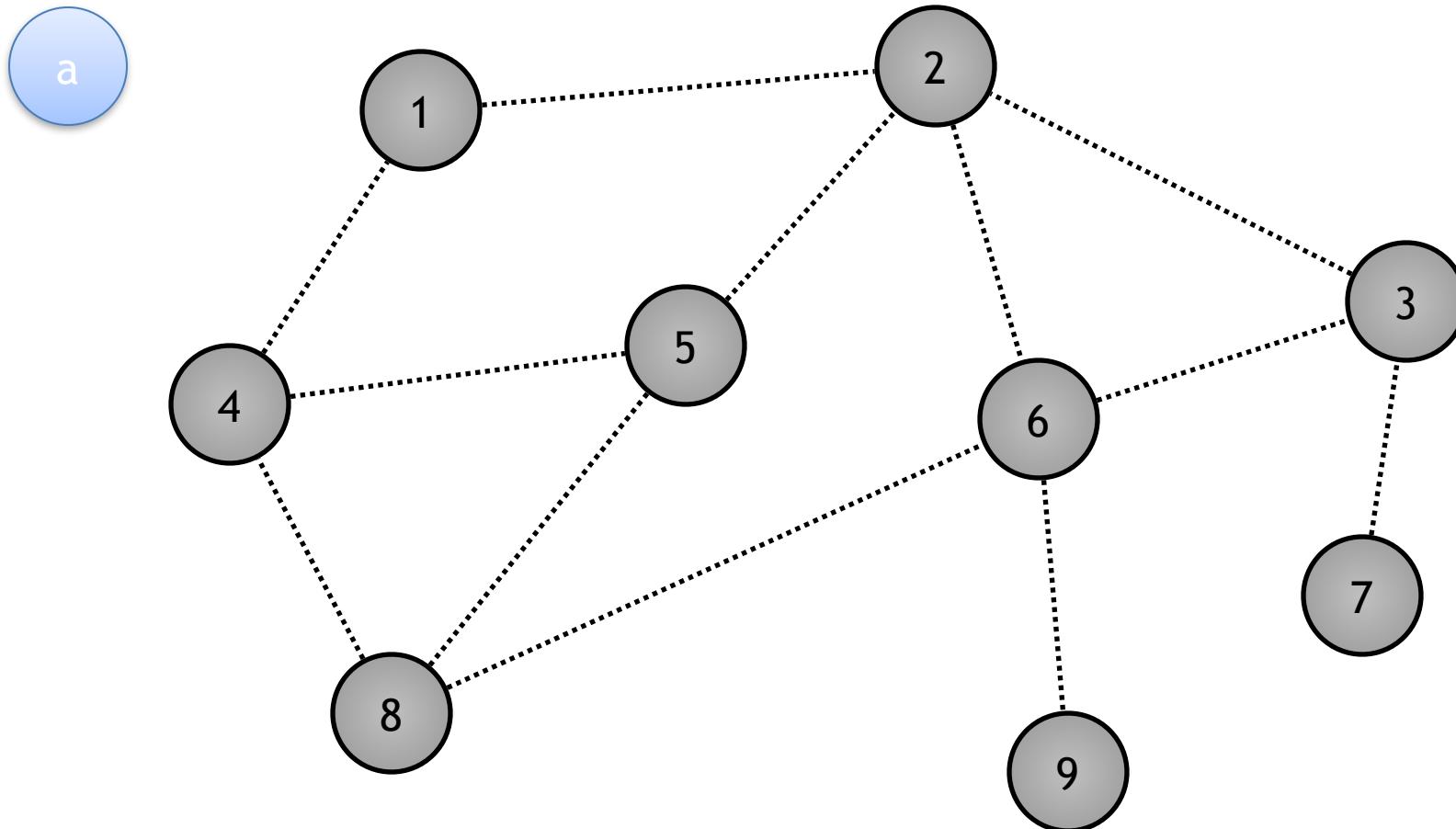
- Each broker performs filtering
- Based on the filtering, each broker disseminates publications towards matching subs
- The next hops repeat the process until subs are reached
- Supports any matching model (e.g. content-based)
- Subs are flooded to initialize **routing paths**
- **Optimization:**
 - Subscription covering
 - Advertising-based routing



<http://padres.msrg.org>

Filtering-Based Routing

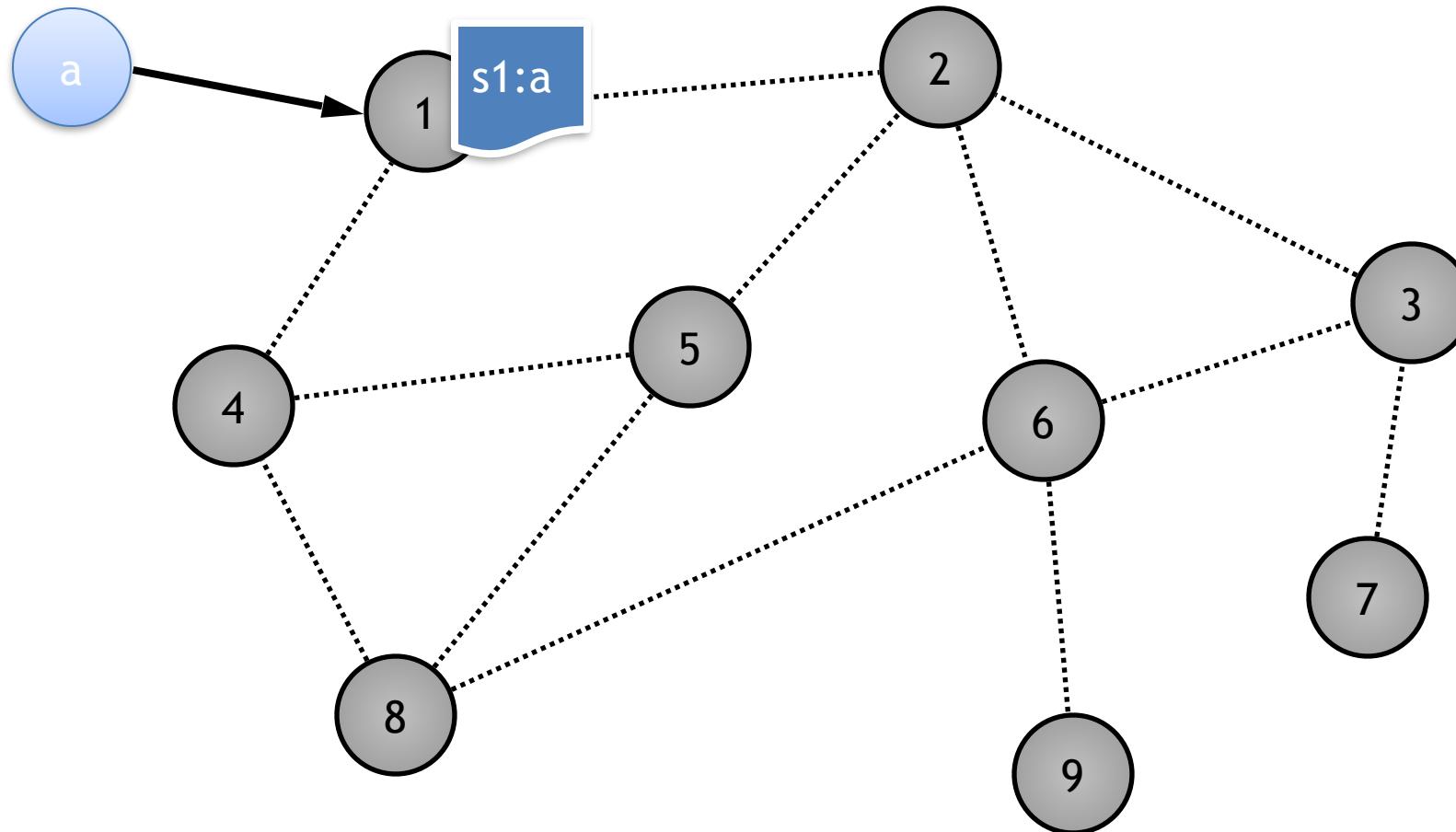
Subscription Flooding



Filtering-Based Routing

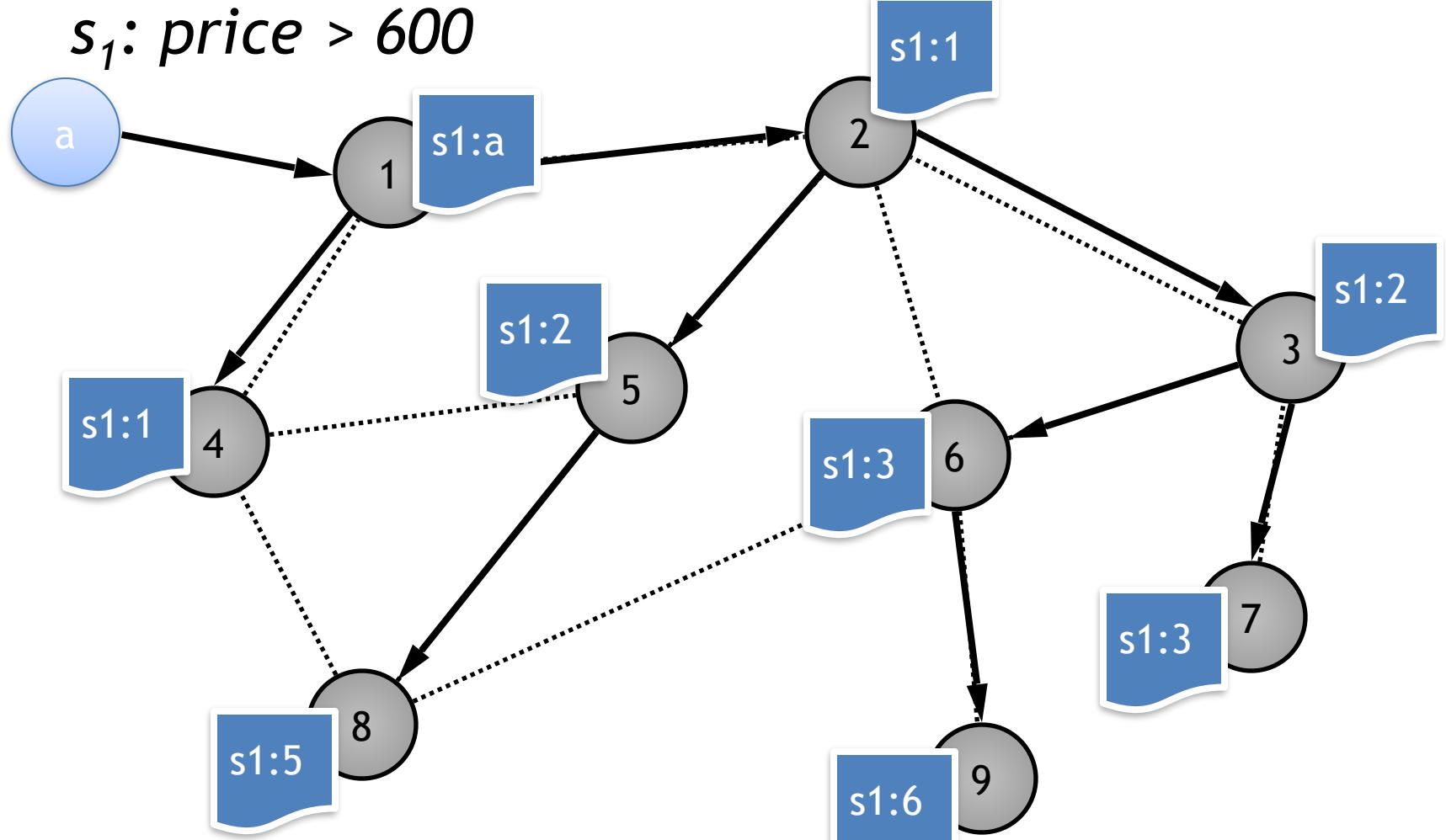
Subscription Flooding

$s_1: \text{price} > 600$



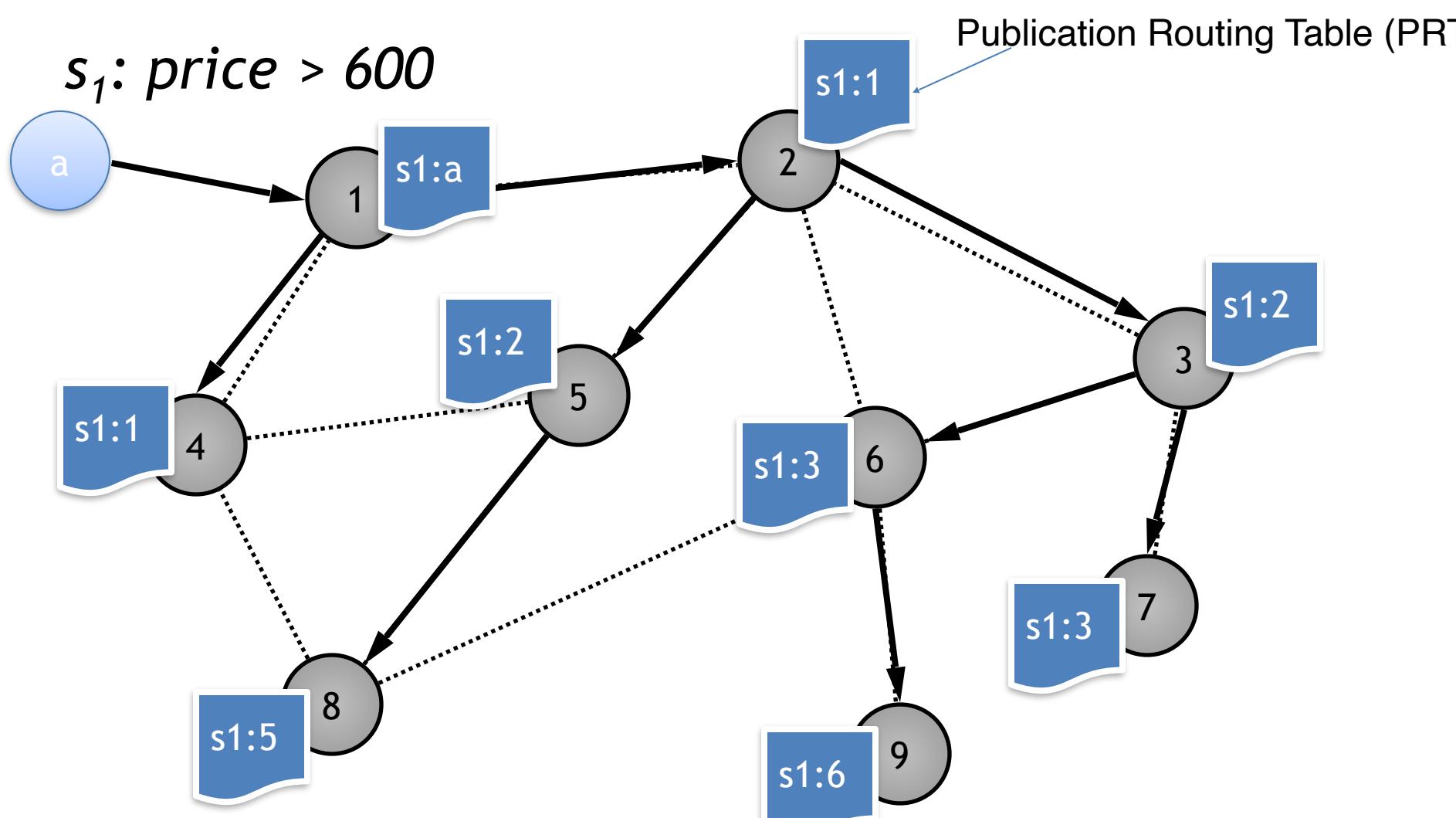
Filtering-Based Routing

Subscription Flooding



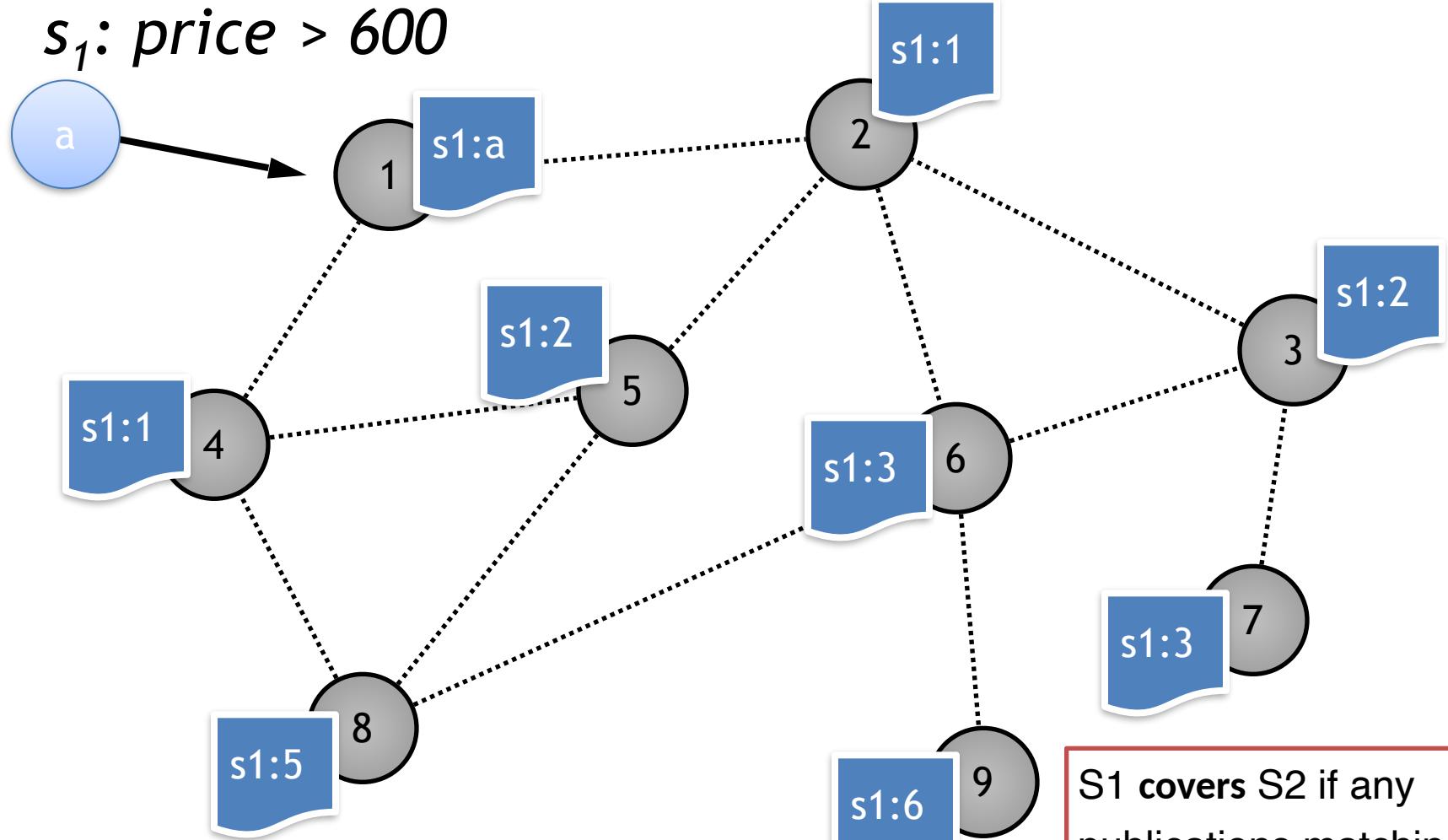
Filtering-Based Routing

Subscription Flooding



Filtering-Based Routing

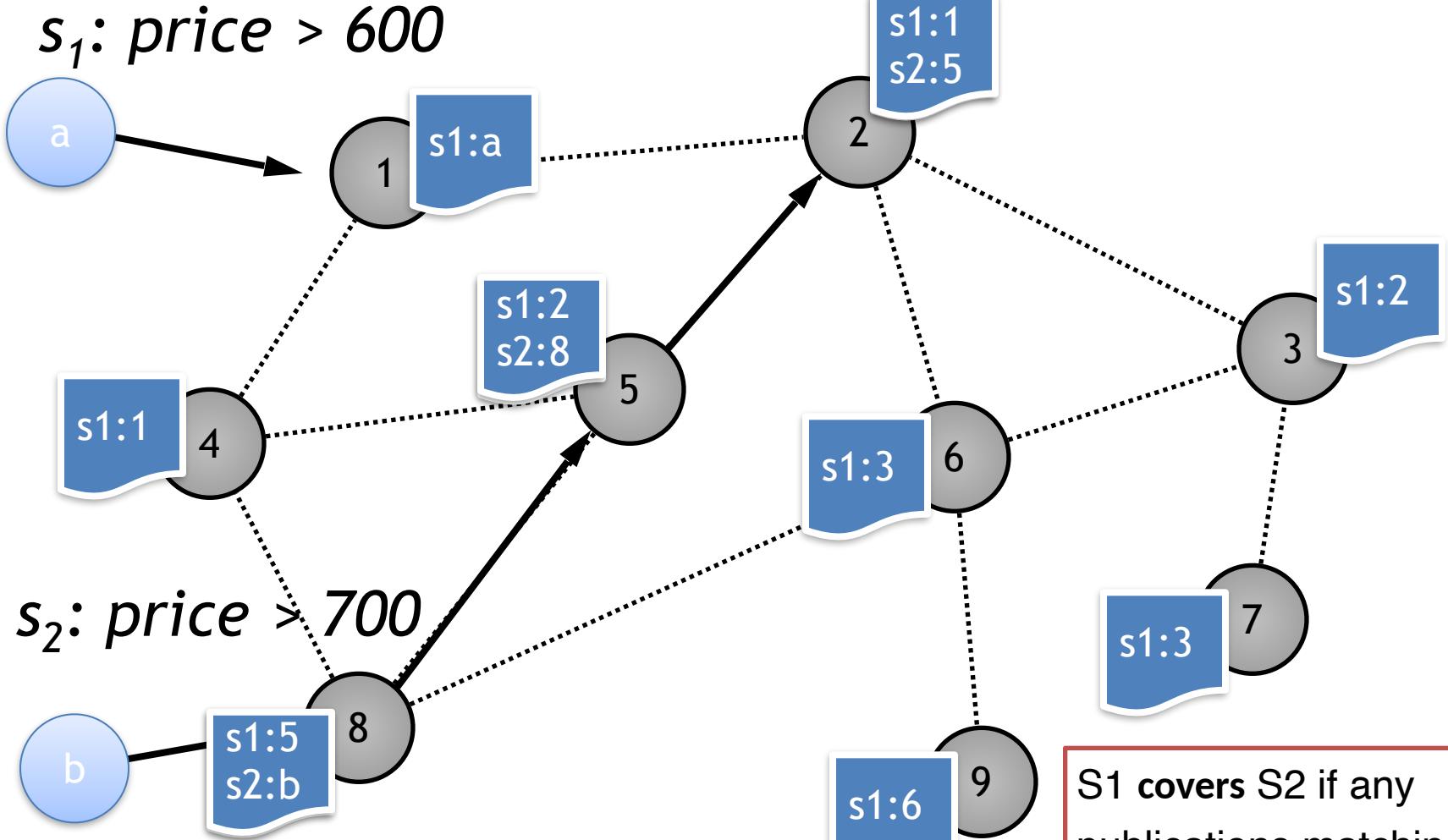
Subscription Covering



S1 covers S2 if any publications matching S2 also matches S1

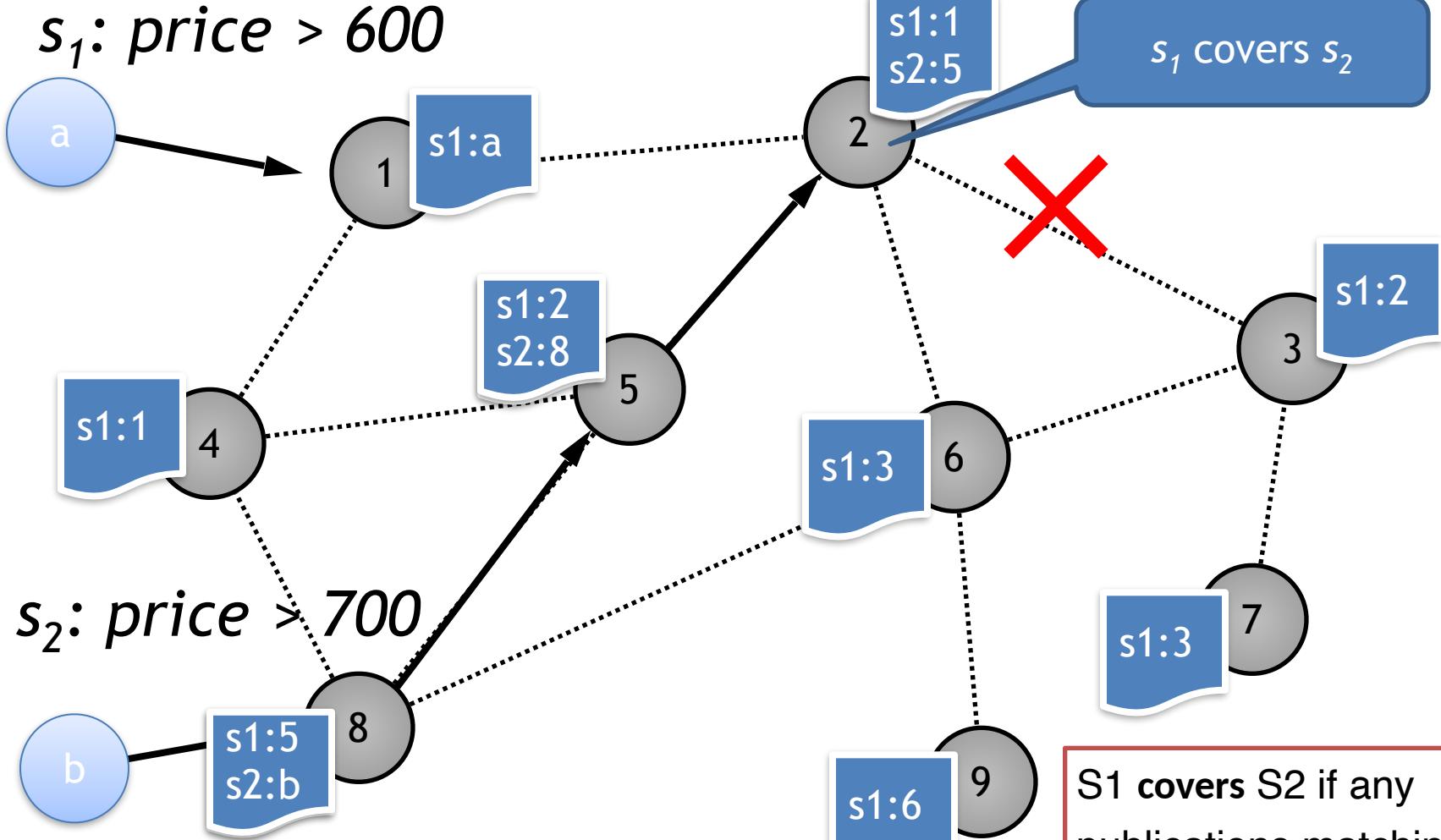
Filtering-Based Routing

Subscription Covering



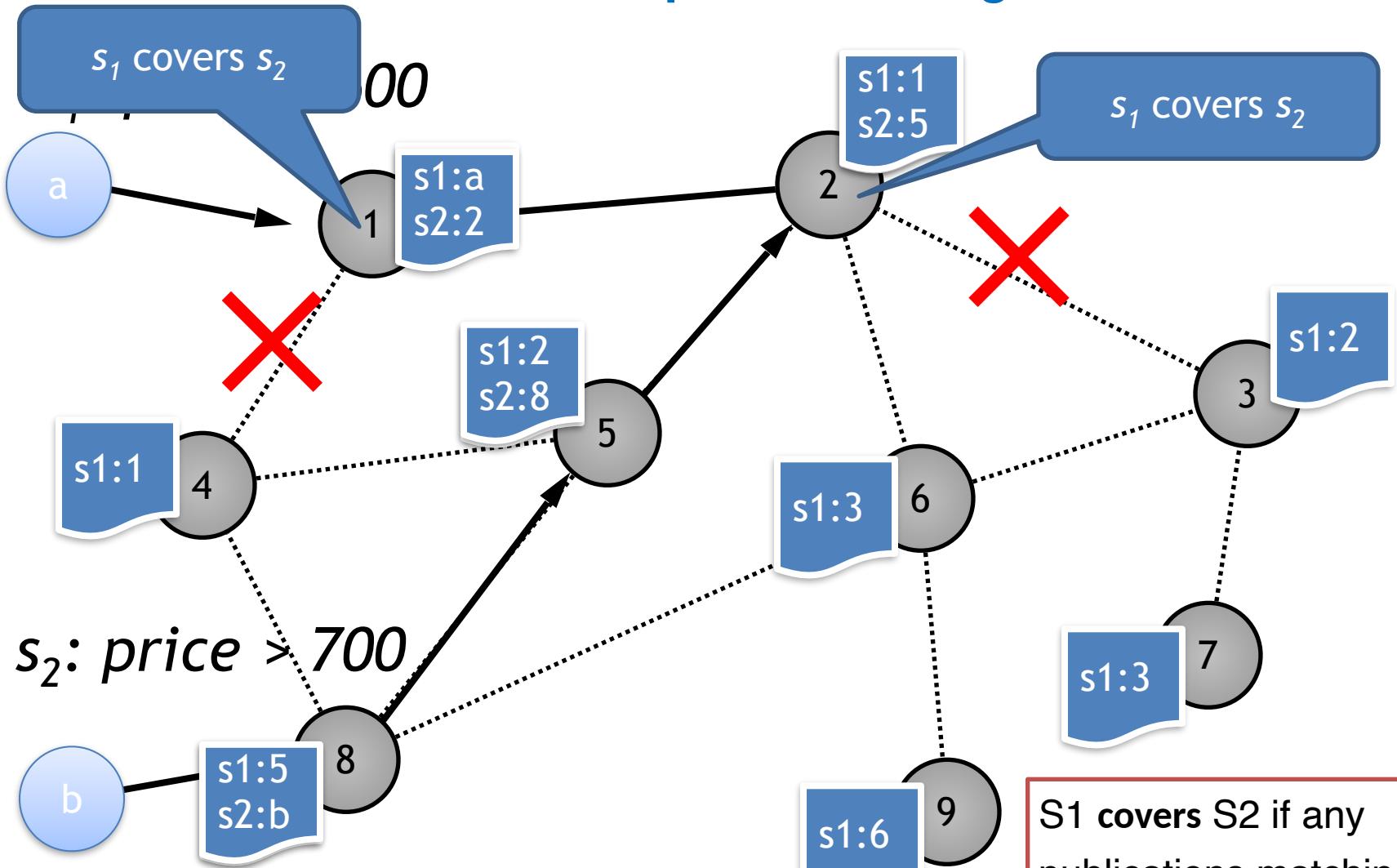
Filtering-Based Routing

Subscription Covering



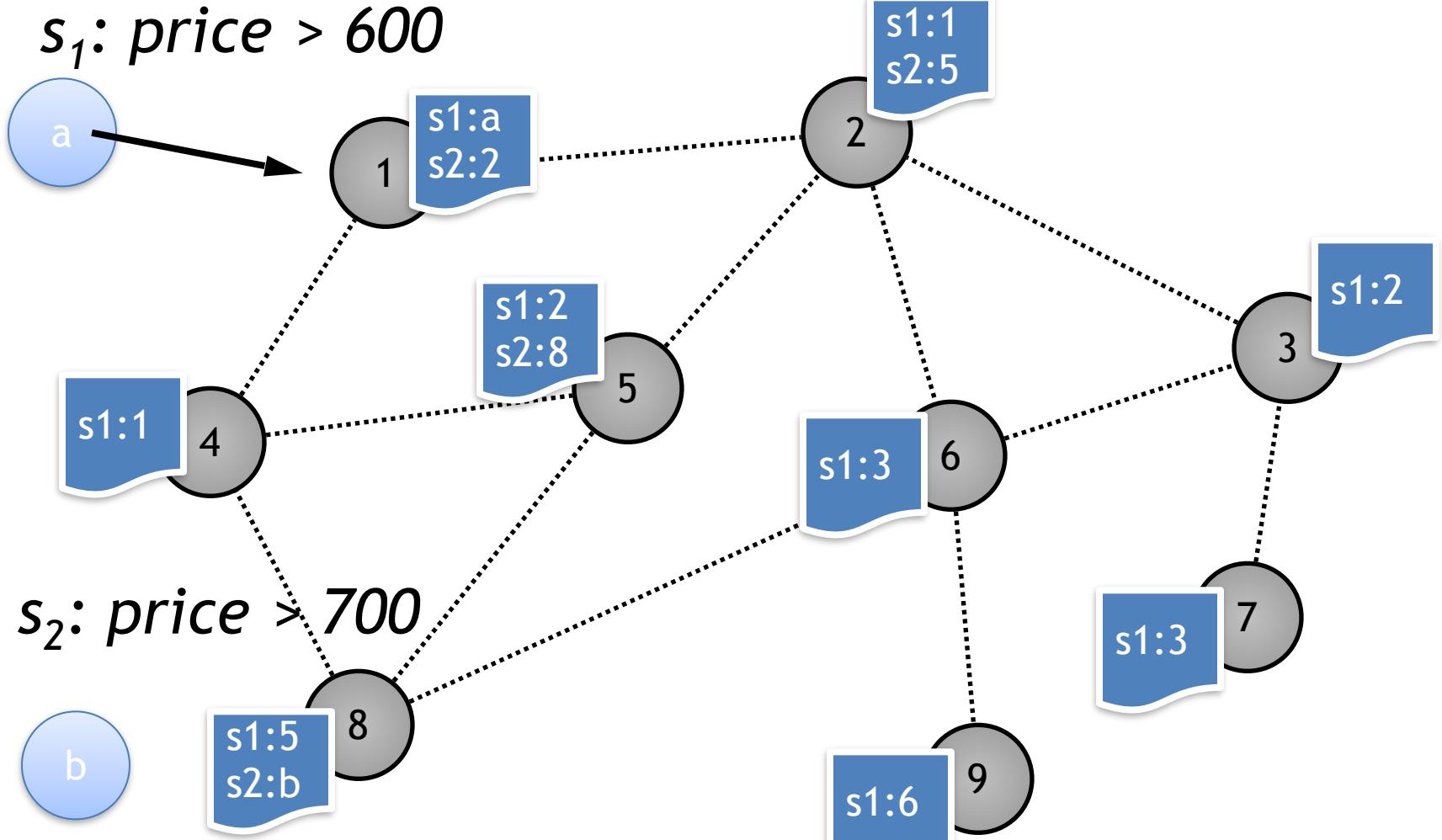
Filtering-Based Routing

Subscription Covering



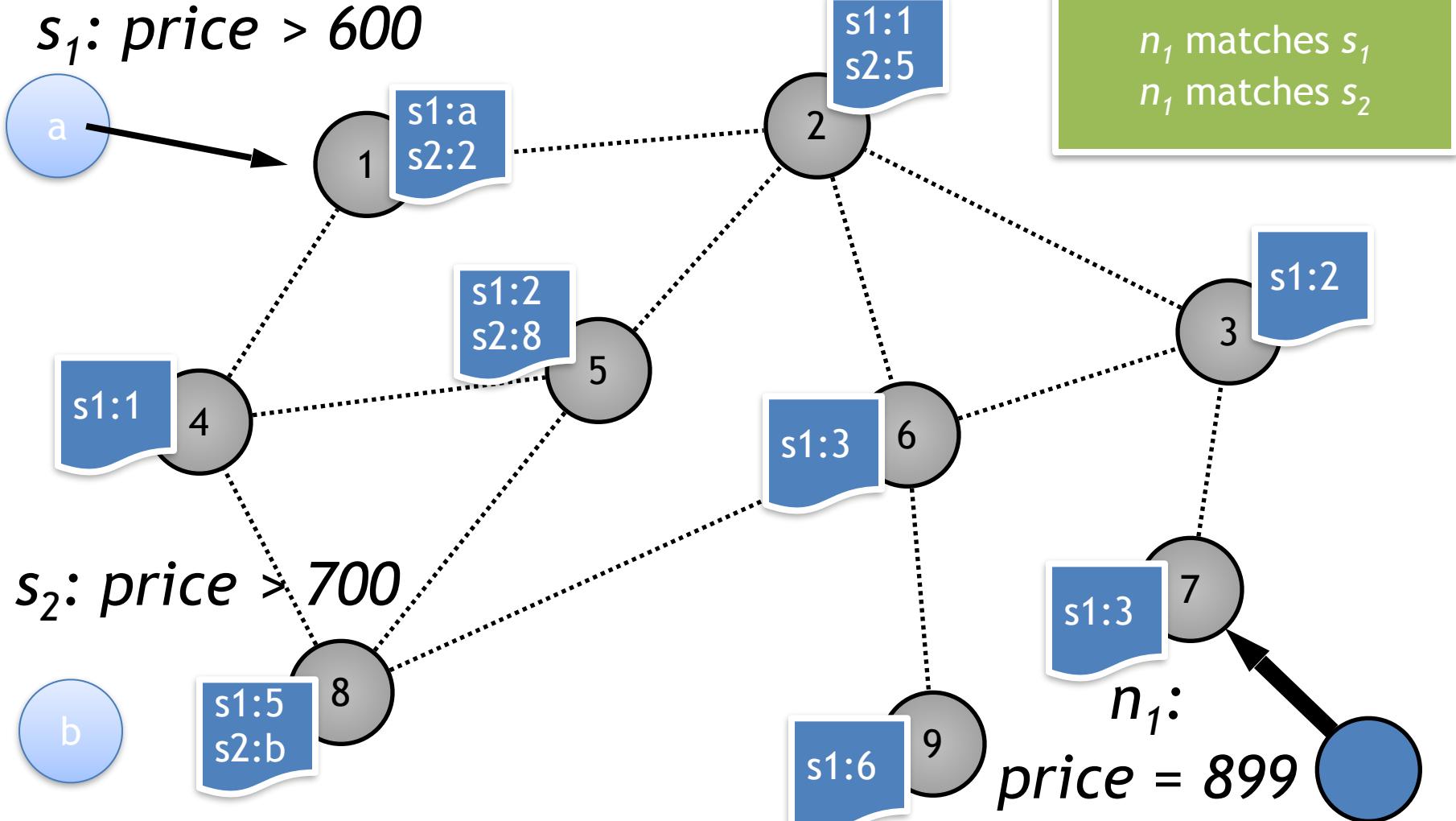
Filtering-Based Routing

Notification Delivery



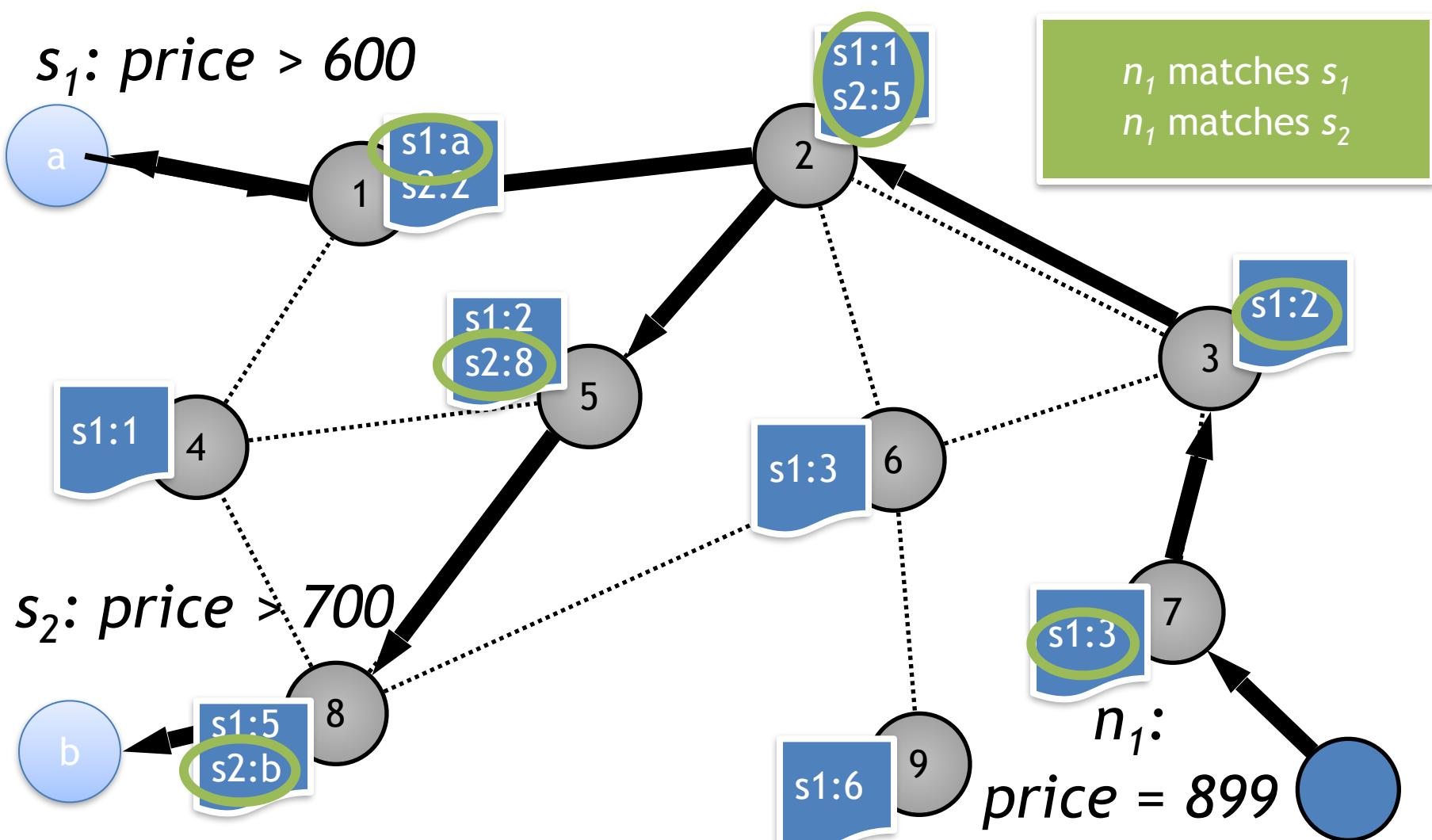
Filtering-Based Routing

Notification Delivery



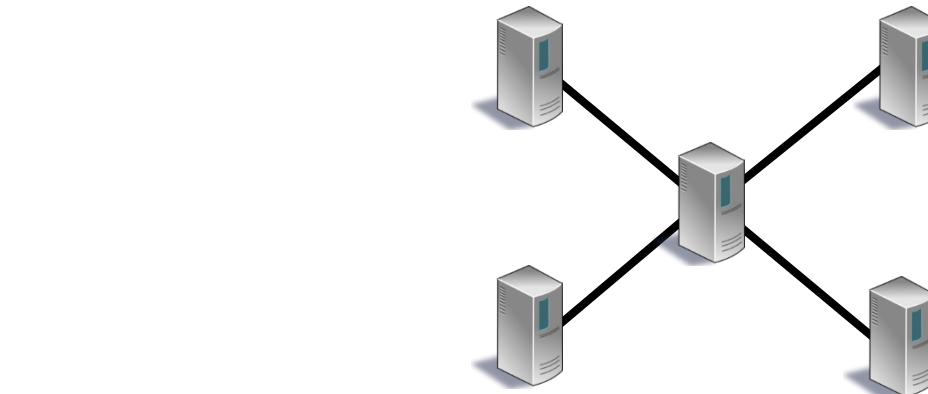
Filtering-Based Routing

Notification Delivery



Advertisement-Based Routing

Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisement path



Subscription path



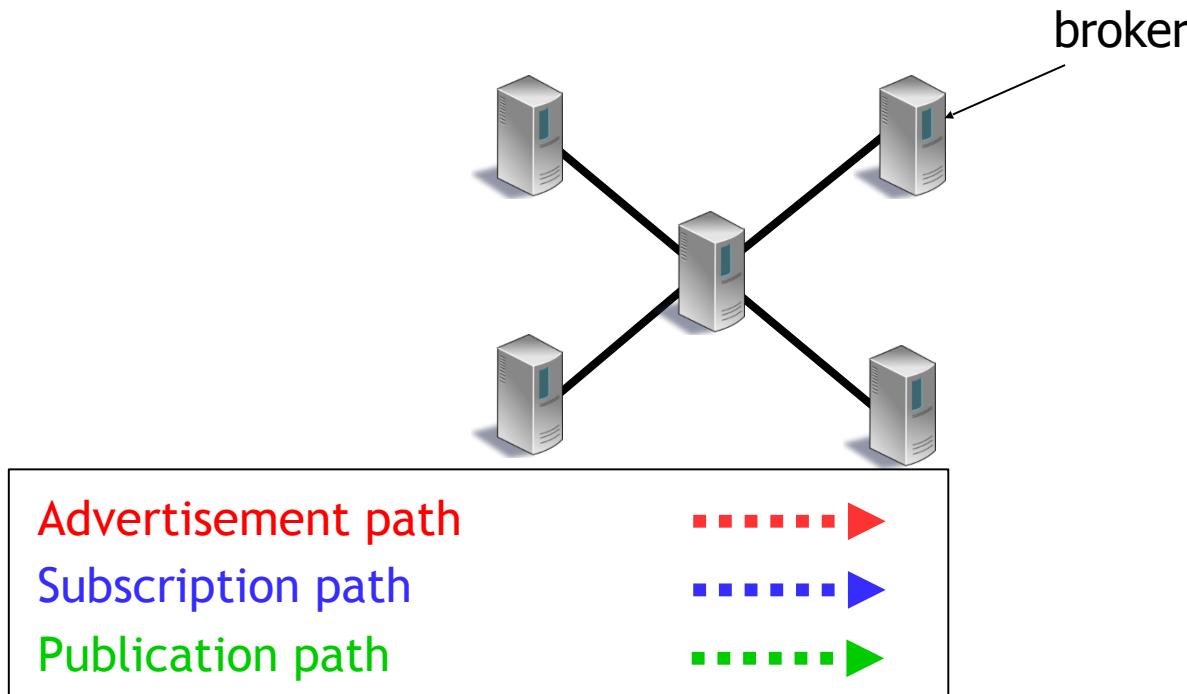
Publication path



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

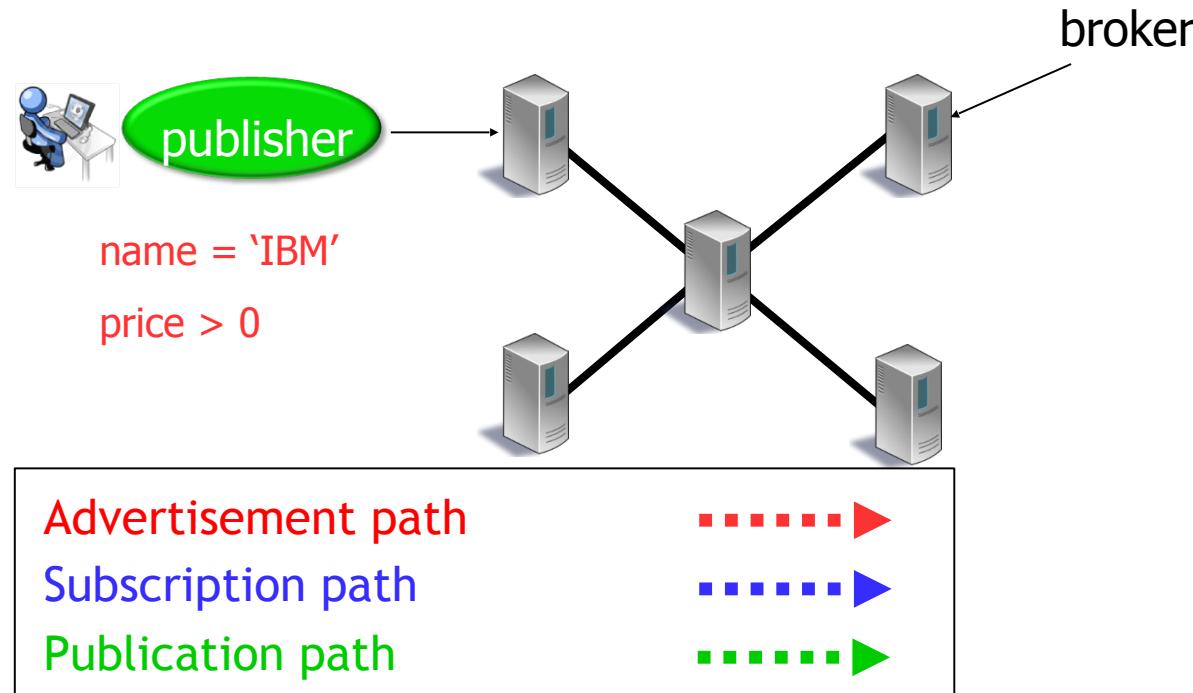
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

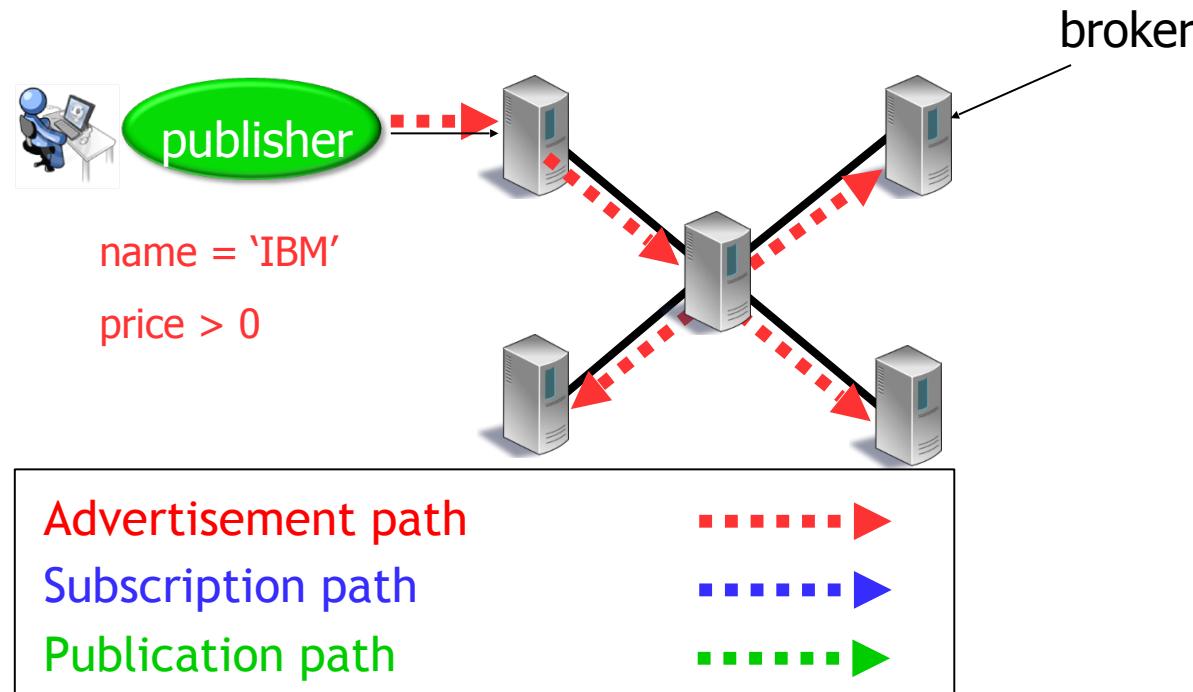
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

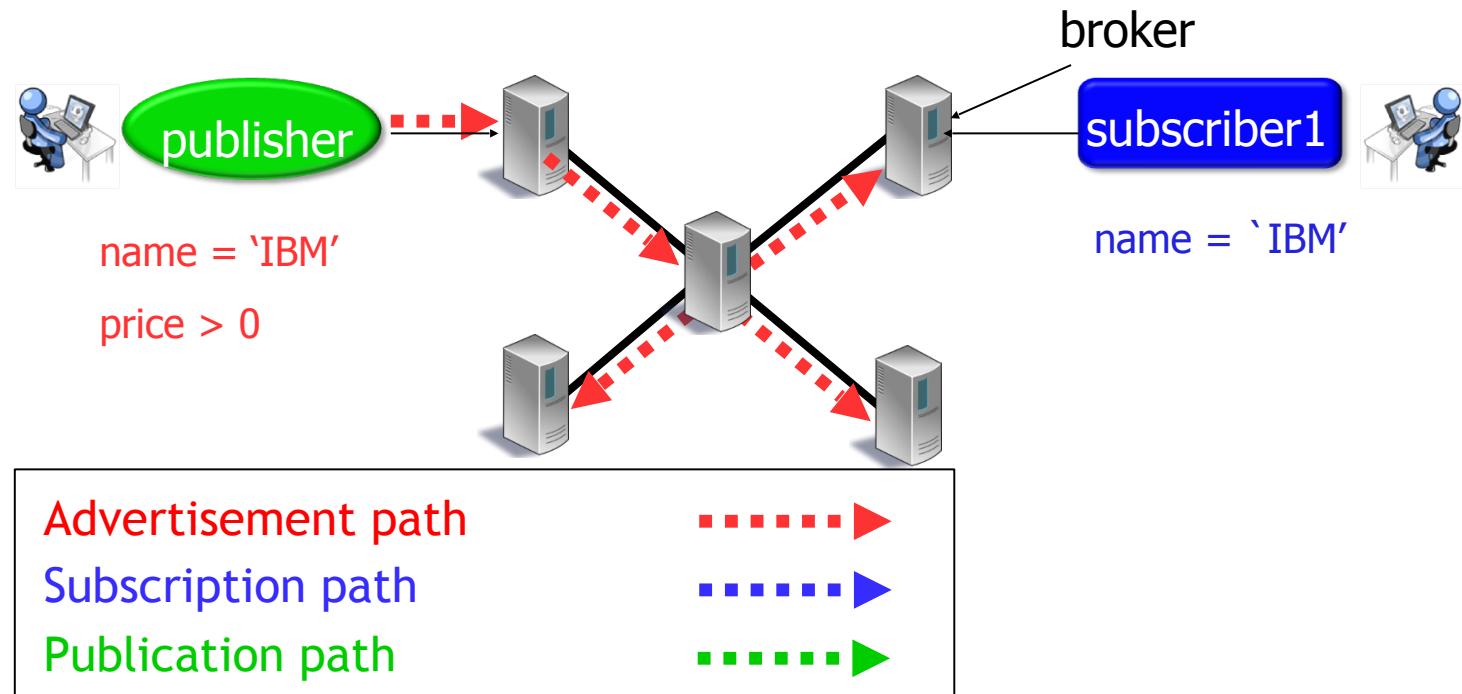
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

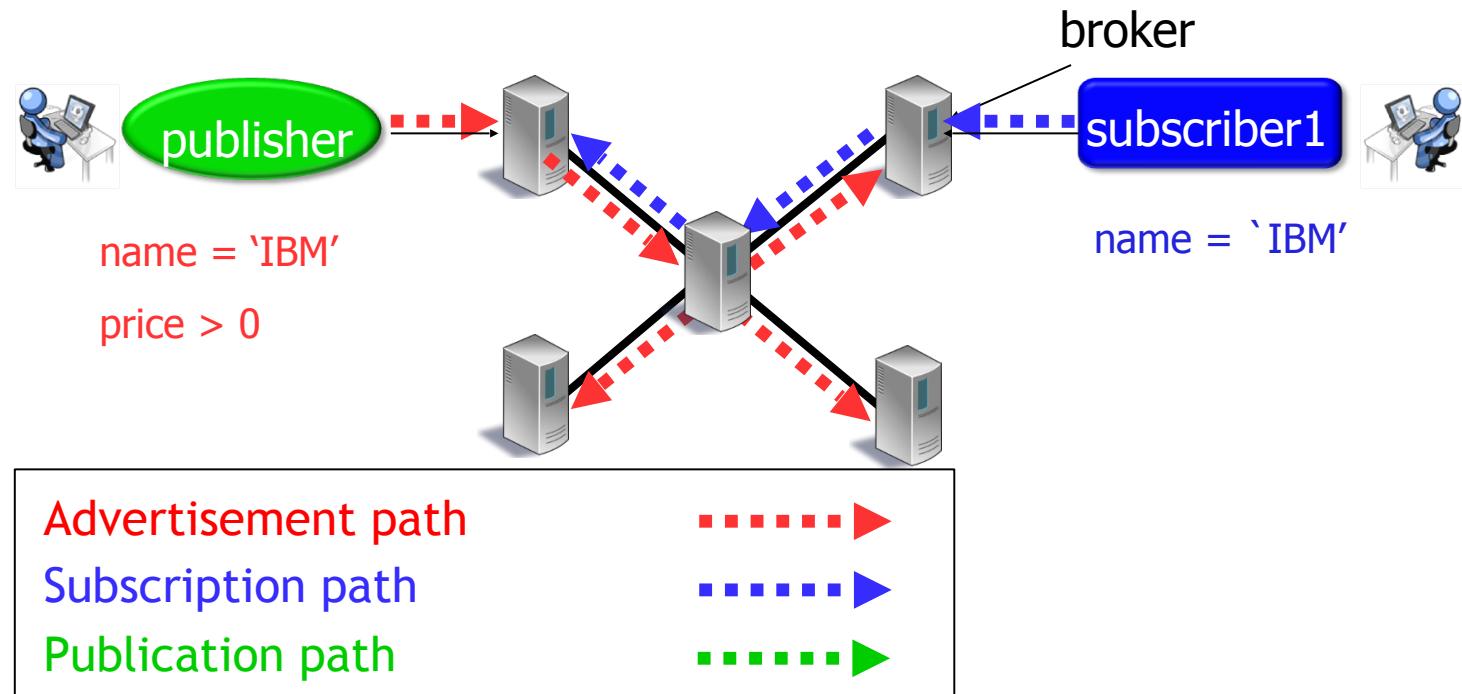
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

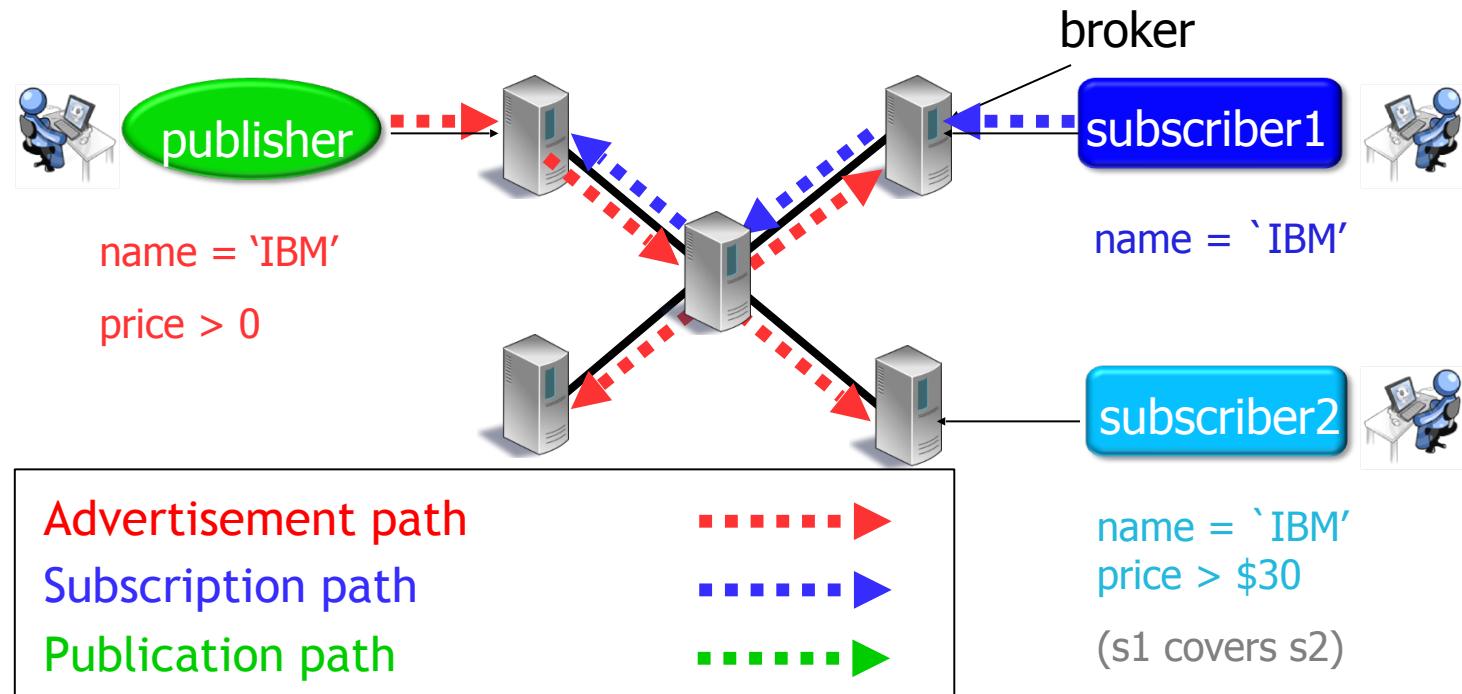
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

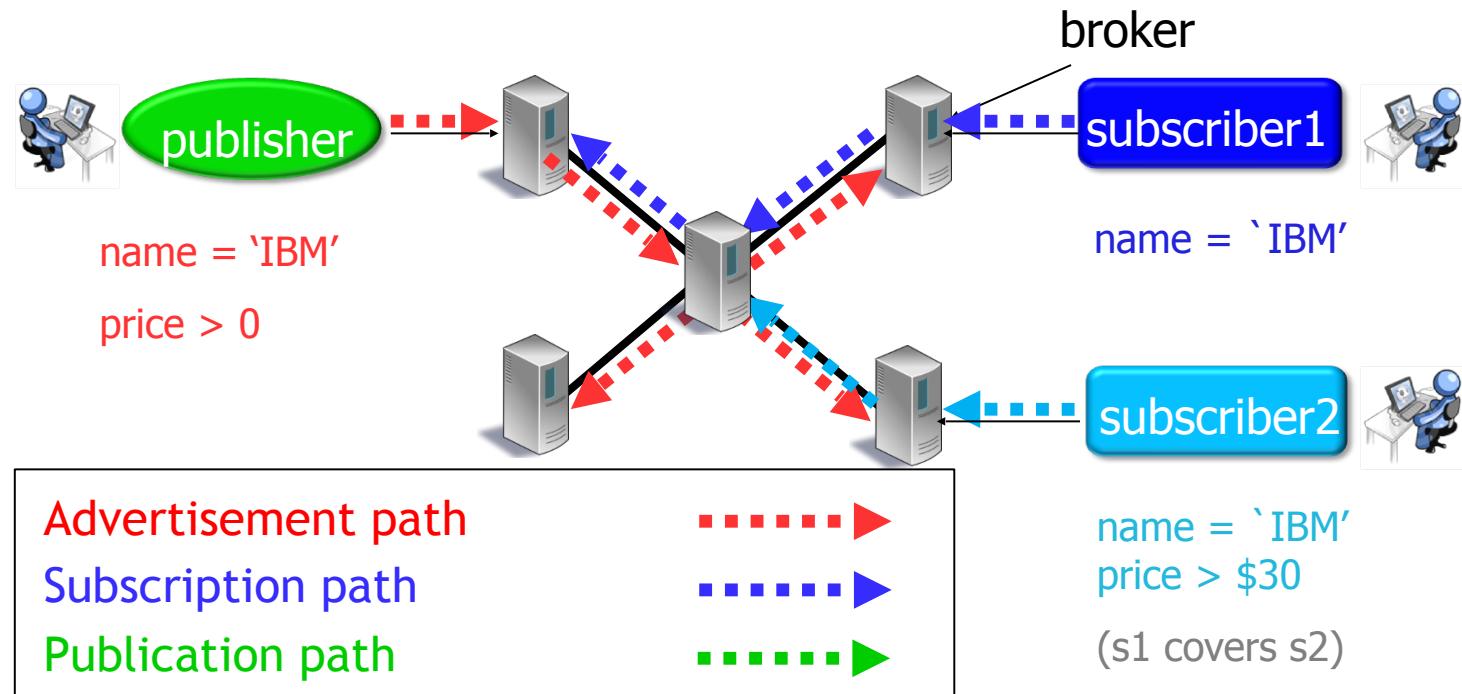
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

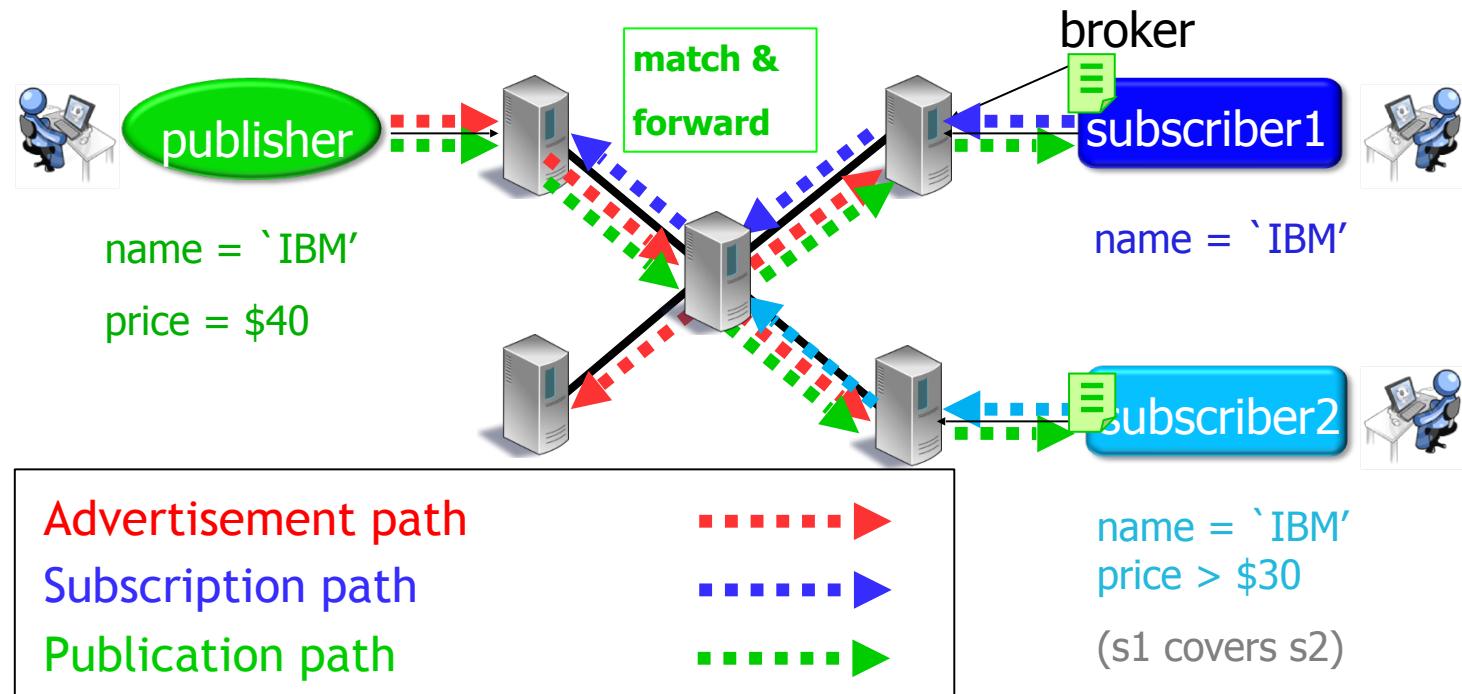
Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Advertisement-Based Routing

Publishers must first *advertise* the type of publications they are going to make.
Advertisements are stored at each broker in a Subscription Routing Table (SRT).



Advertisements are flooded, but not subscriptions

Routing Algorithms Summary

- Rendezvous-based
 - Partition-based: Simple, requires master
 - DHT-based: $\log(n)$ hops, structured network
- Overlay-based
 - Rendezvous-based: Efficient matching, inefficient communication, RPs are bottlenecks
 - Filtering-based: Computation intensive, efficient communication
 - Advertisement flooding: Useful if the number of publishers is much smaller than subscribers

Blockchain

PEZMAN NASIRIFARD

KAIWEN ZHANG

HANS-ARNO JACOBSEN

Outline

- Blockchain
 - Concepts: Mining, Proof-of-Work, Smart Contracts
 - Bitcoin, Ethereum, Hyperledger
 - Blockchain Applications

Final Exam Review (Feb 5th):
<https://forms.gle/JbyYk4S7XfUz2vT99>

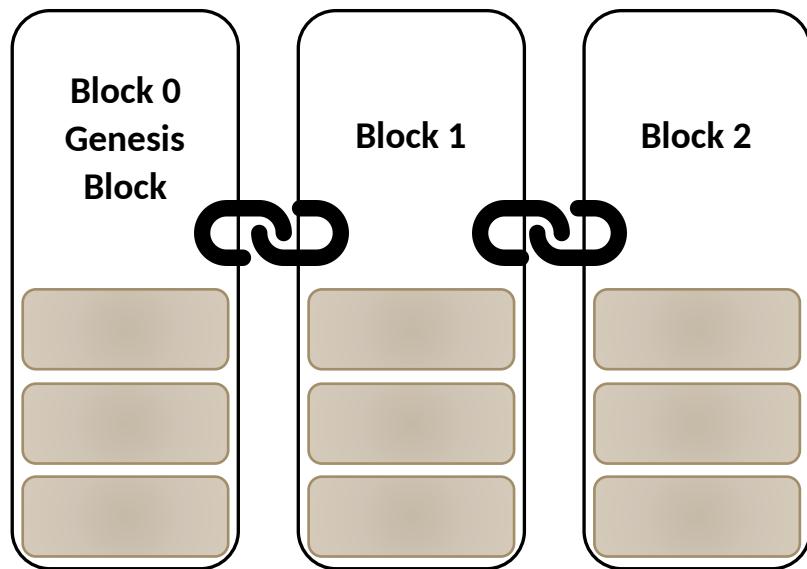
Blockchain Concepts

DEFINITIONS

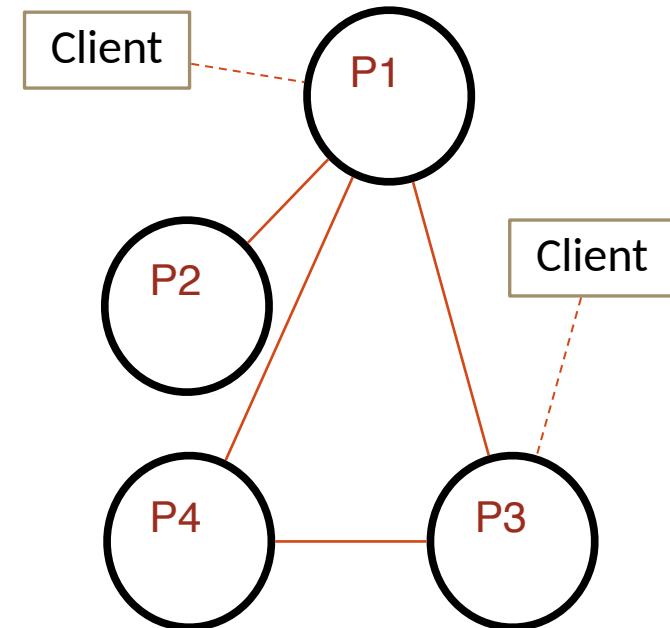
BITCOIN OVERVIEW

Blockchain 101: Distributed Ledger

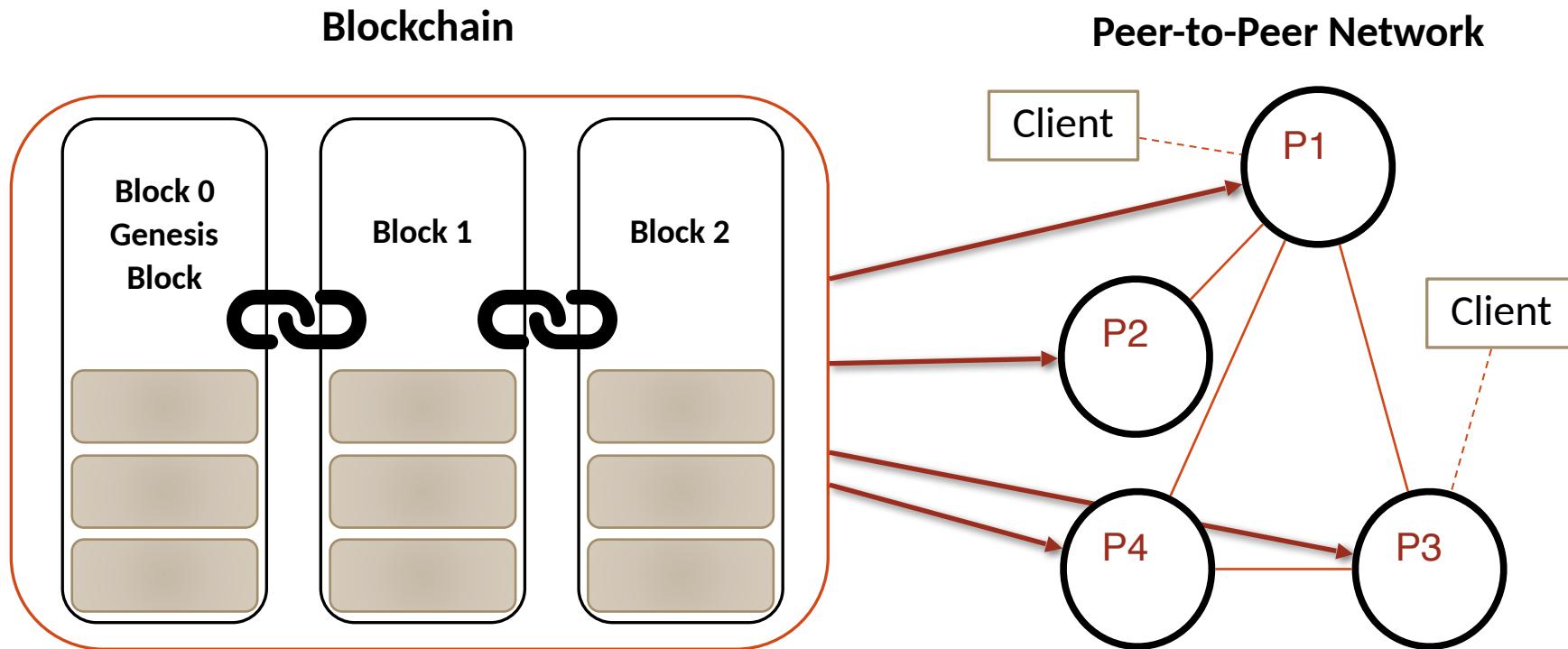
Blockchain



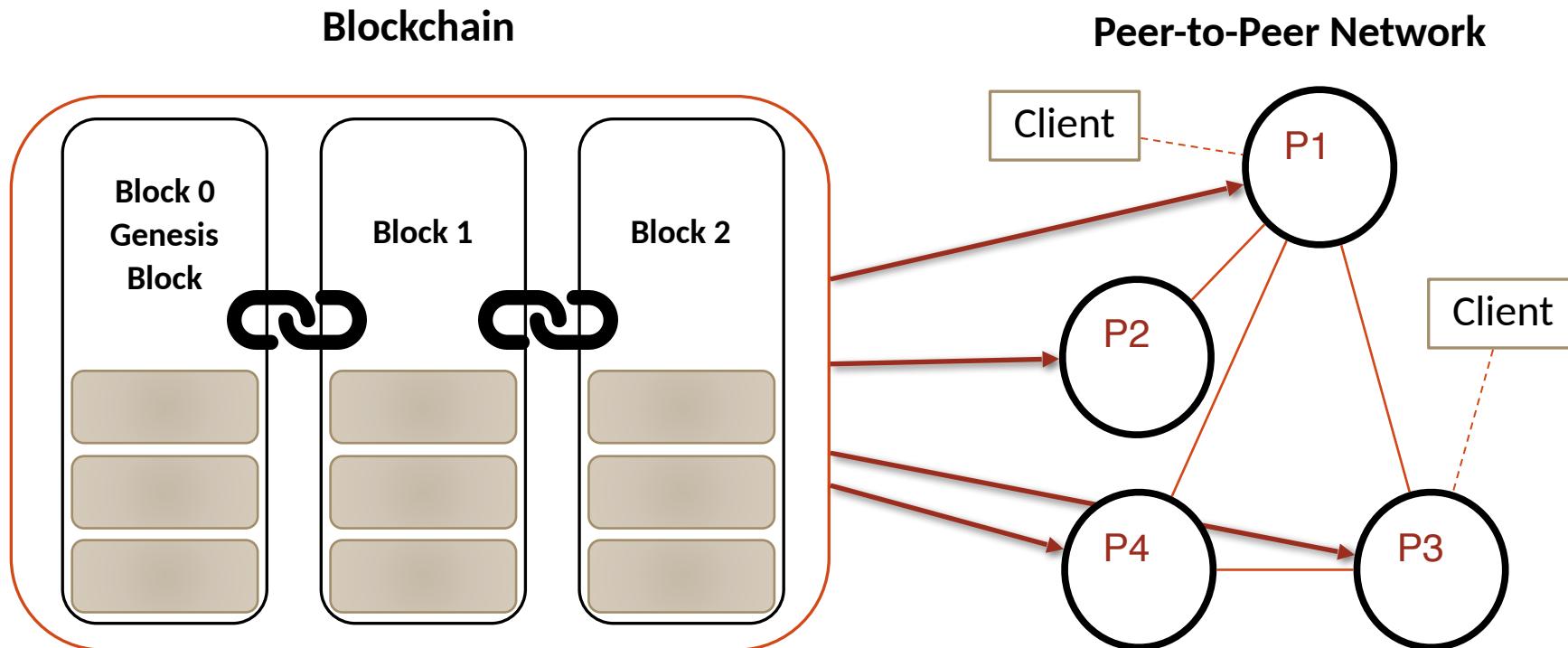
Peer-to-Peer Network



Blockchain 101: Distributed Ledger



Blockchain 101: Distributed Ledger

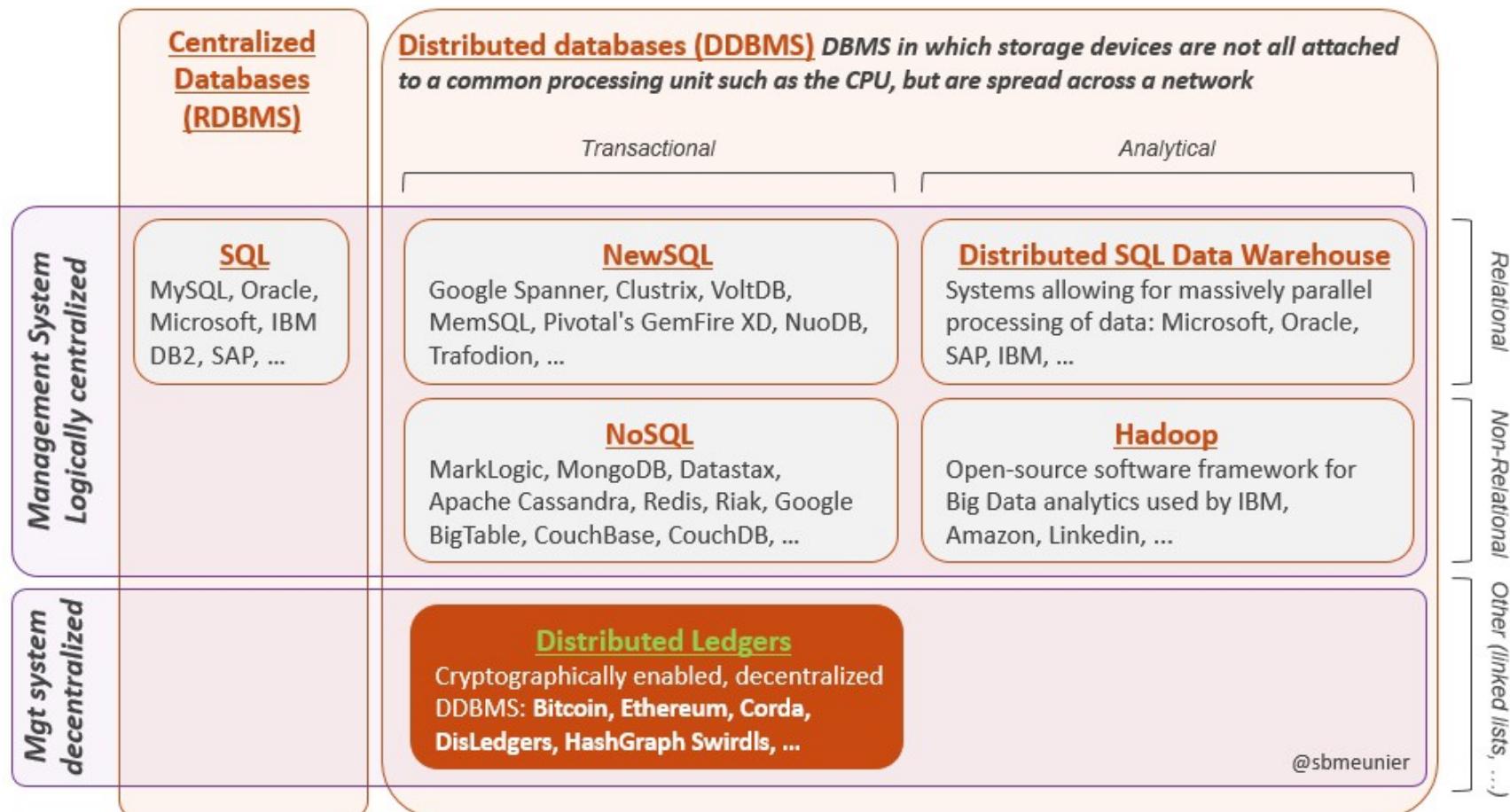


Cryptography is used to encrypt data, prevent modification, and replicate the distributed ledger.

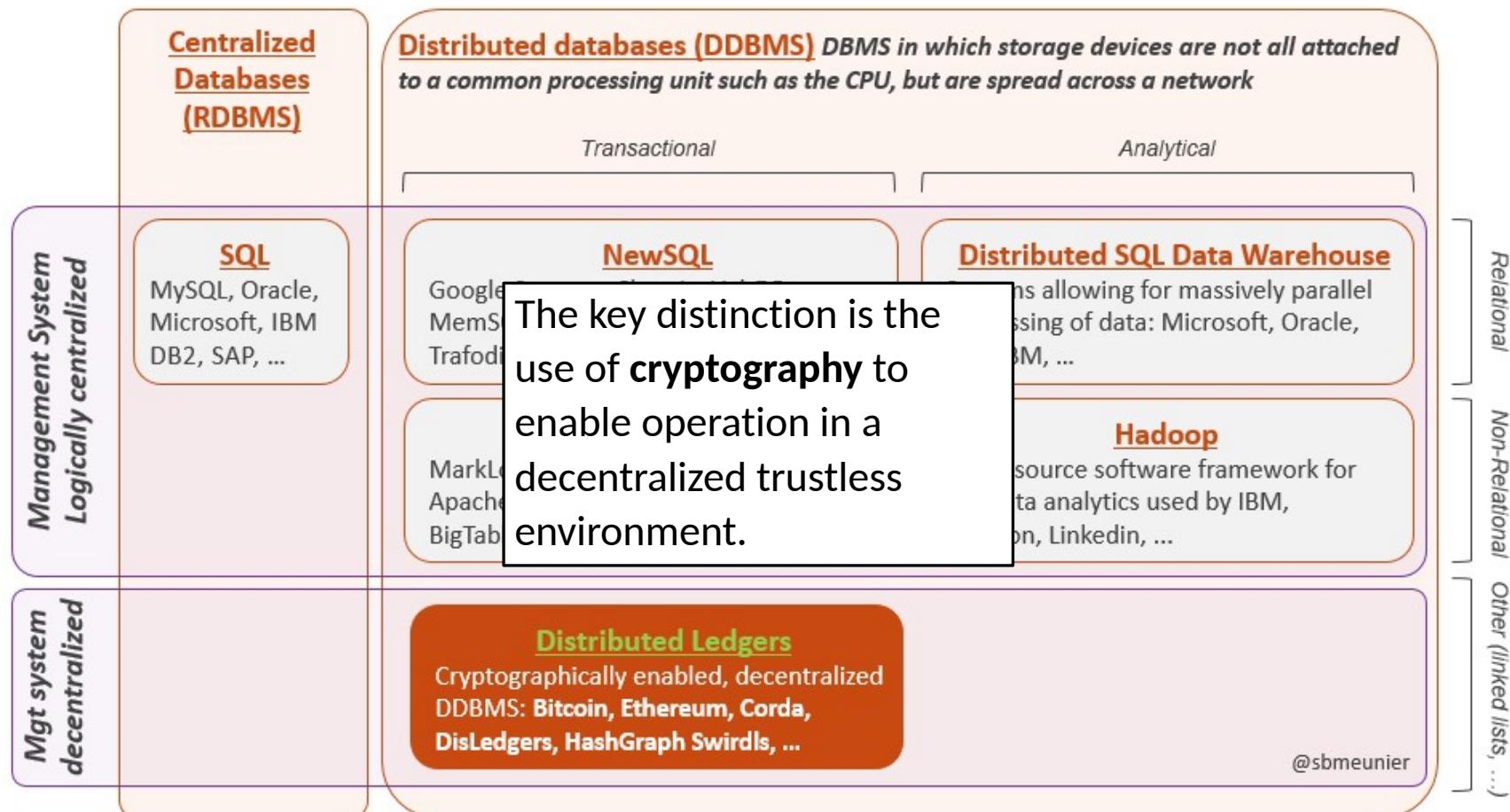
What is a Blockchain-based Distributed Ledger?

- An append-only log storing transactions
- Fully replicated across a large number of peers (called miners)
- Comprised of immutable blocks of data
- Deterministically verifiable (using the blockchain data structure)
- Able to execute transactions programmatically (e.g., Bitcoin transactions and smart contracts)
- A priori decentralized, does not rely on a third party for trust

Comparison with Databases

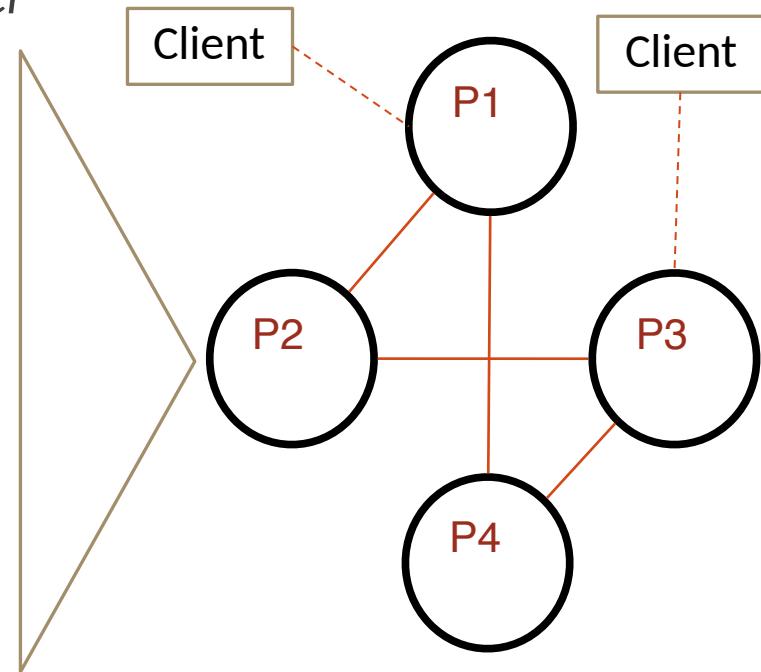
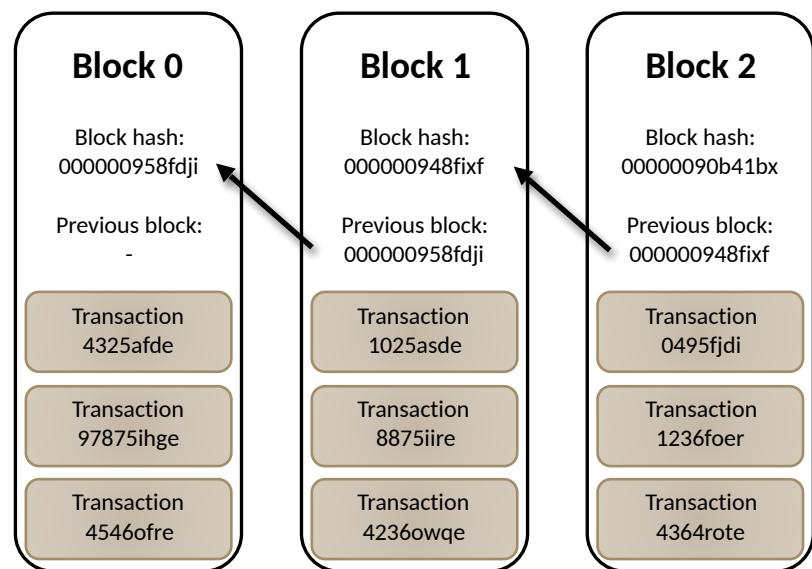


Comparison with Databases



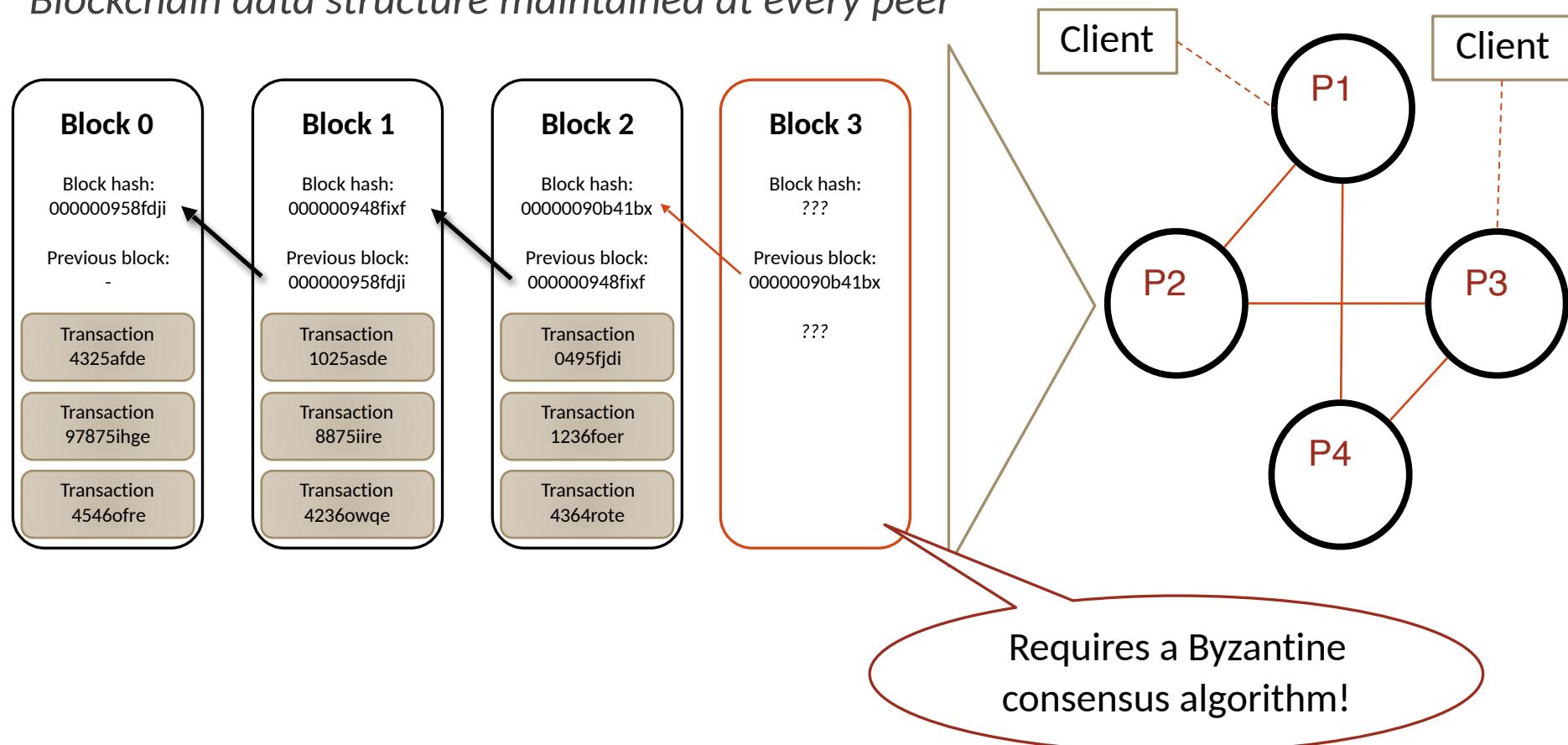
Handling Inserts

Blockchain data structure maintained at every peer



Handling Inserts

Blockchain data structure maintained at every peer



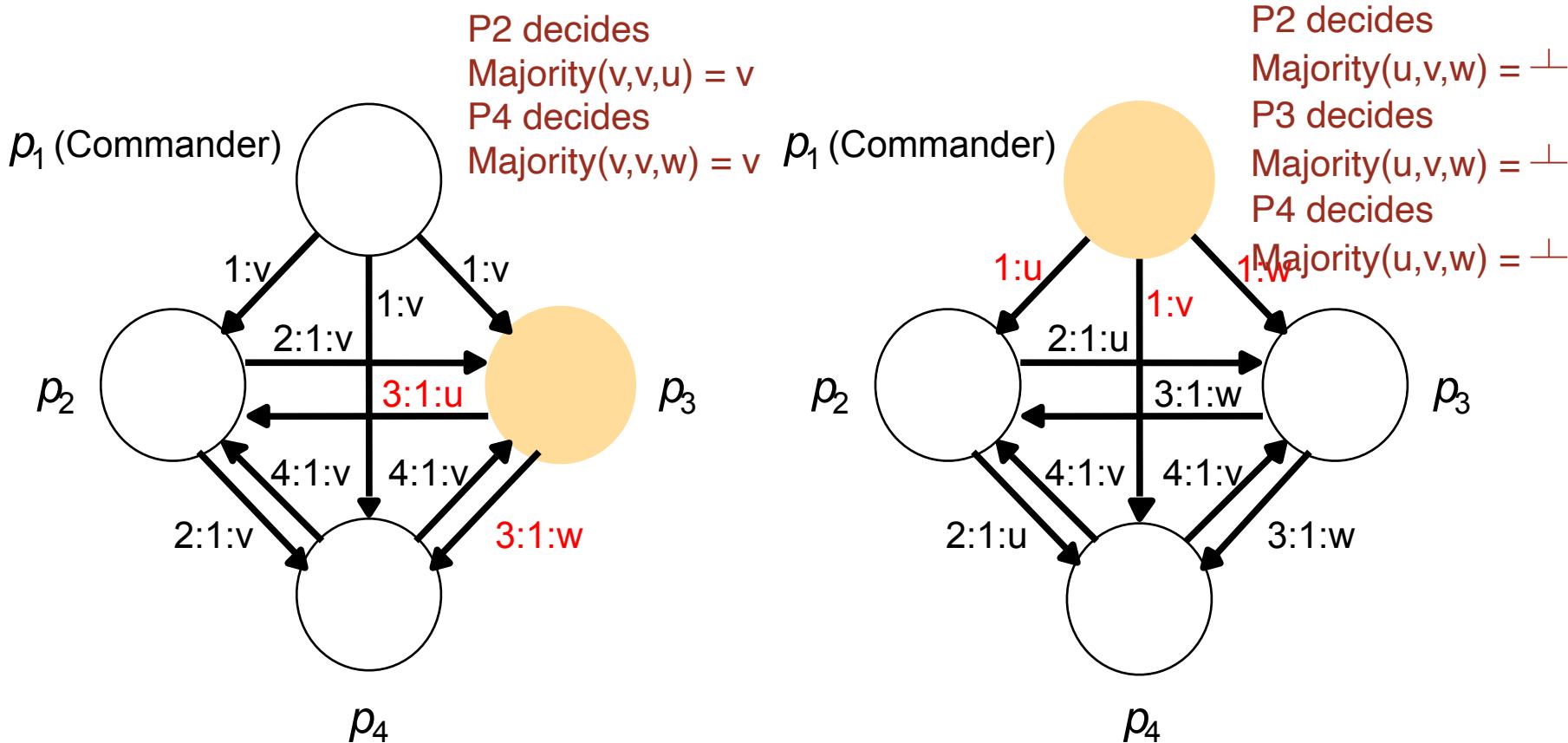
Origin: Byzantine Generals

- A distinguished process (the commander) proposes initial value (eg. "attack", "retreat")
- Other processes, the lieutenants, communicate the commander's value
- Malicious processes can lie about the value (i.e., are faulty)
- Correct processes report truth (i.e., are correct)
- Commander or lieutenants may be faulty
- Consensus means
 - If the commander is correct, then correct processes should agree on his proposed value
 - If the commander is faulty, then all correct processes agree on a value (any value, could be the faulty commander's value!)

Byzantine generals: $3f + 1$

- Lieutenants are not voting for their own preference. They are reporting what they heard from the commander.
- Lieutenants decide based on the majority of reports received.
- For f faulty processes, $N \geq 3f + 1$ required to reach consensus
- [Proof: Reaching Agreement in the Presence of Faults](#)

3f+1 Condition (1 failure, 4 nodes)

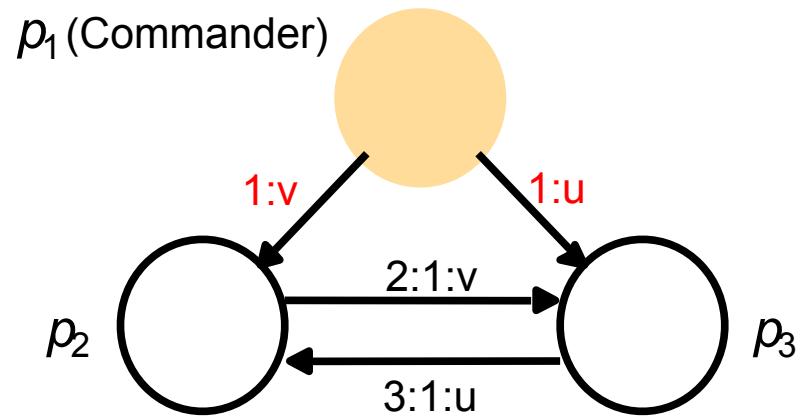
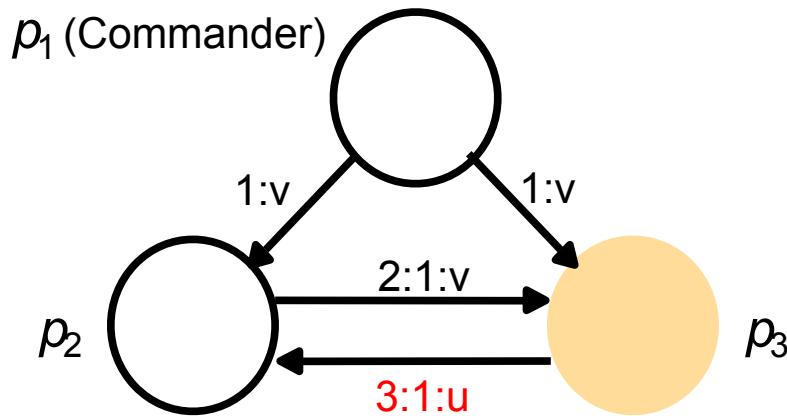


Source: Tanenbaum, Steen.

Faulty processes are shown coloured

Counter-Example

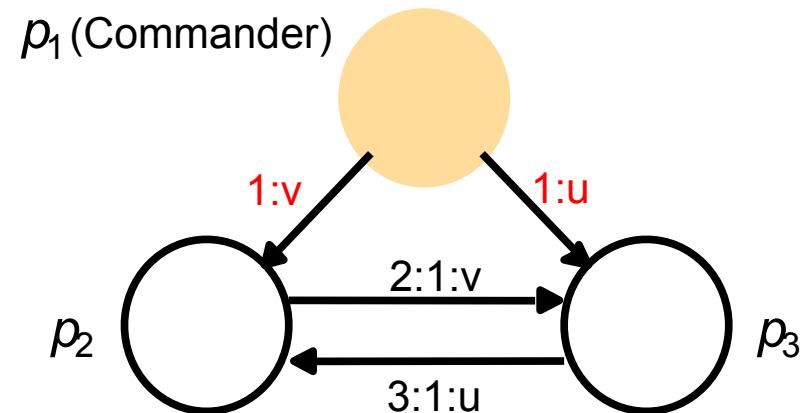
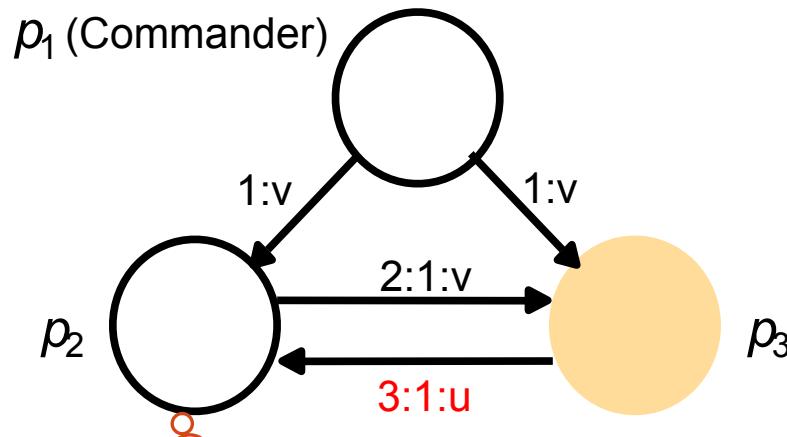
(1 failure, 3 nodes)



Faulty processes are shown coloured

Counter-Example

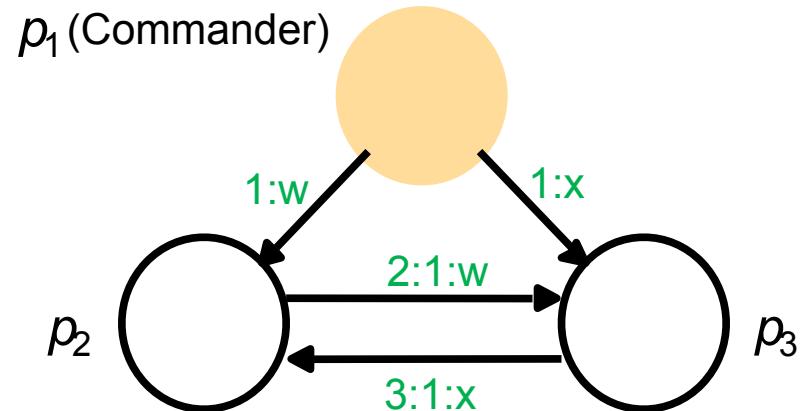
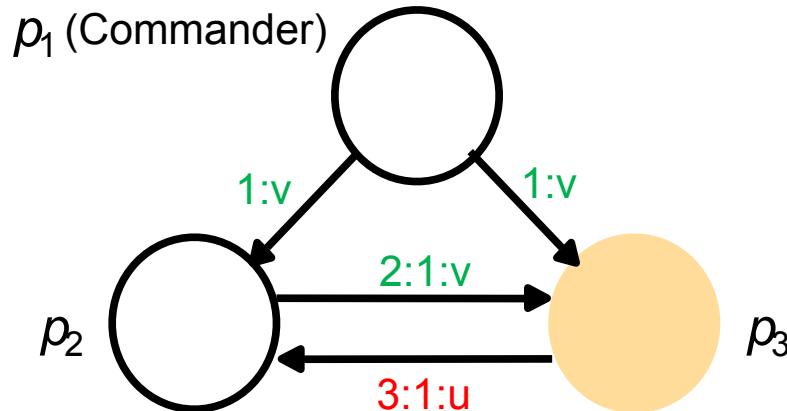
(1 failure, 3 nodes)



Faulty processes are shown coloured

Trust commander or p_3 ?
 Can't tell the difference
 between both
 situations!

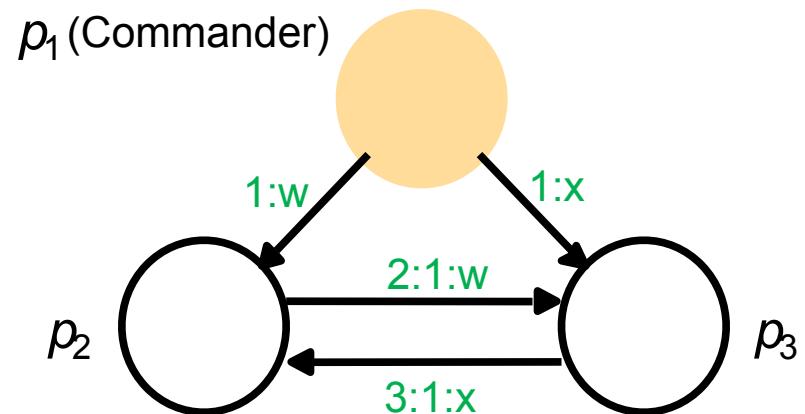
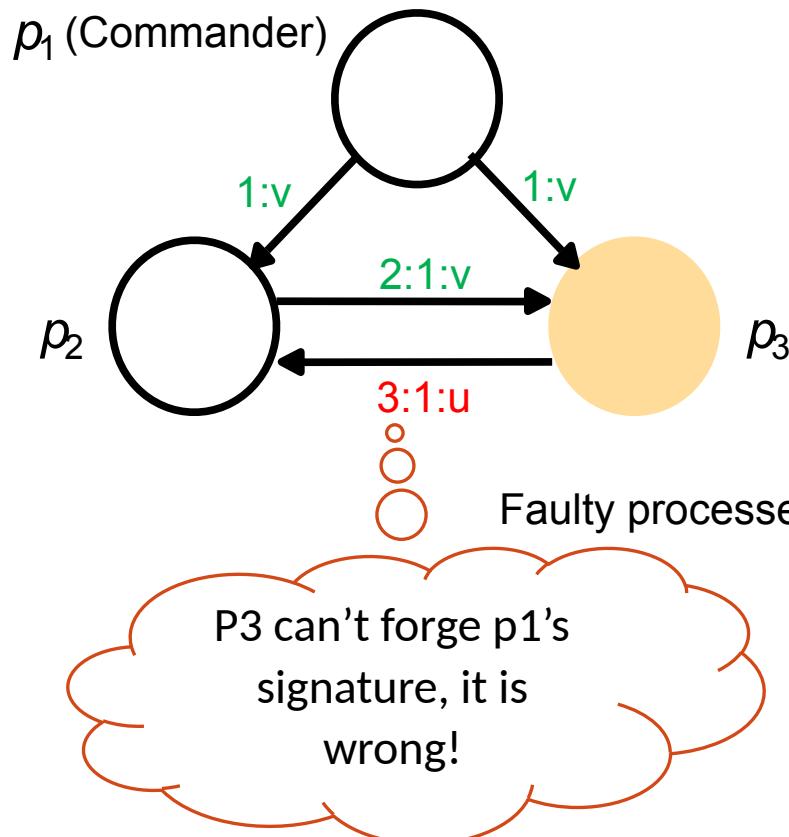
With Signed Messages (Public-key)



Faulty processes are shown coloured

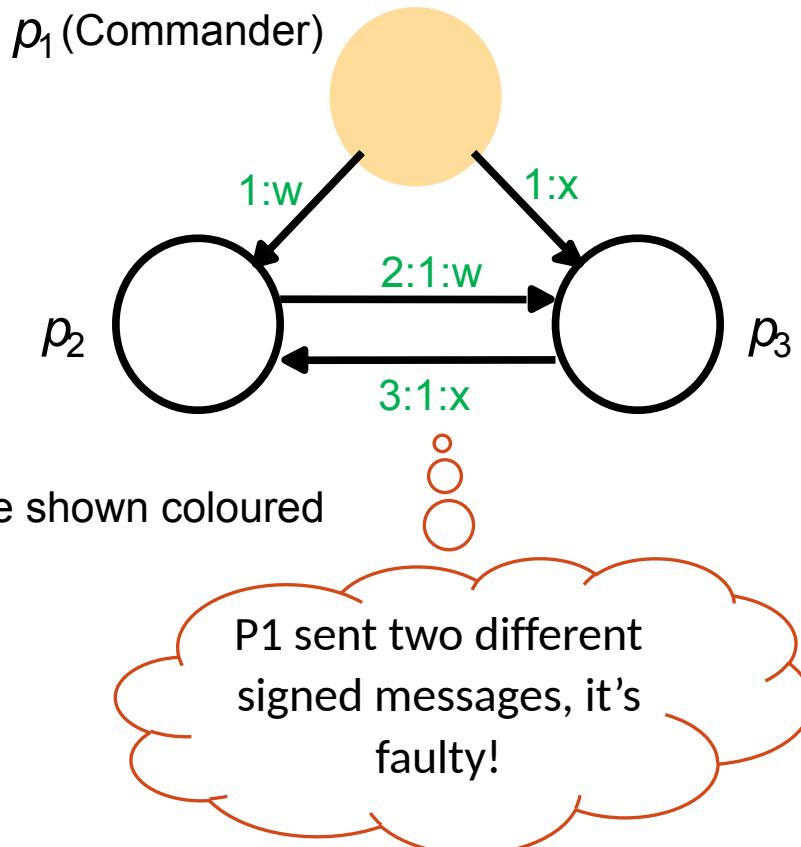
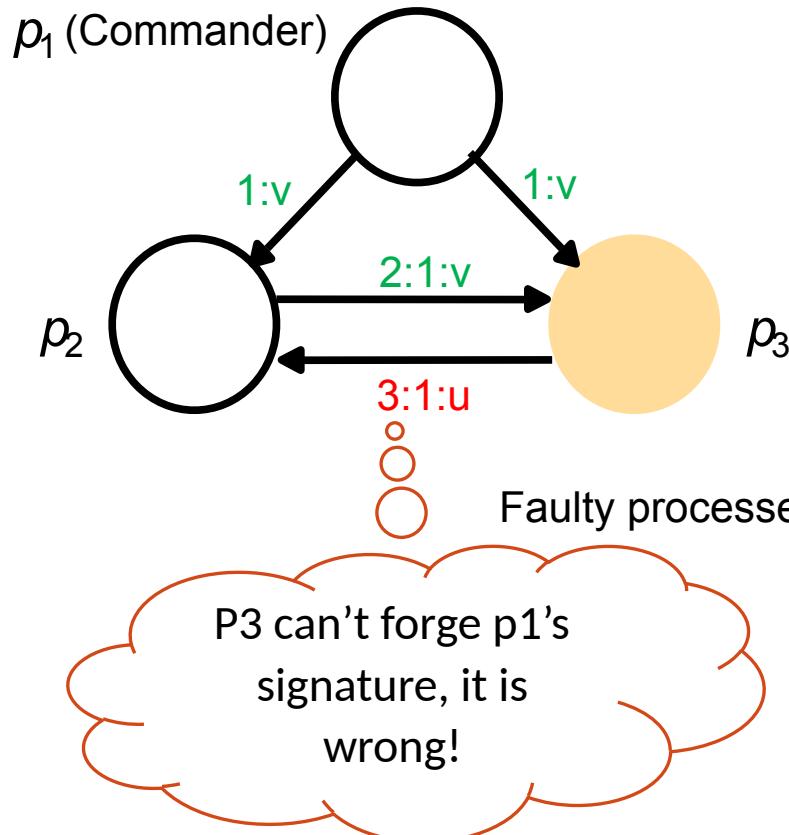
Different model: each message is signed and verified by the public key of sender.

With Signed Messages (Public-key)



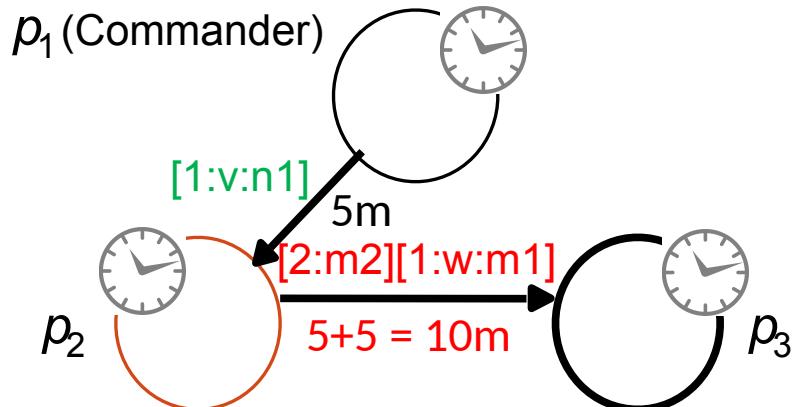
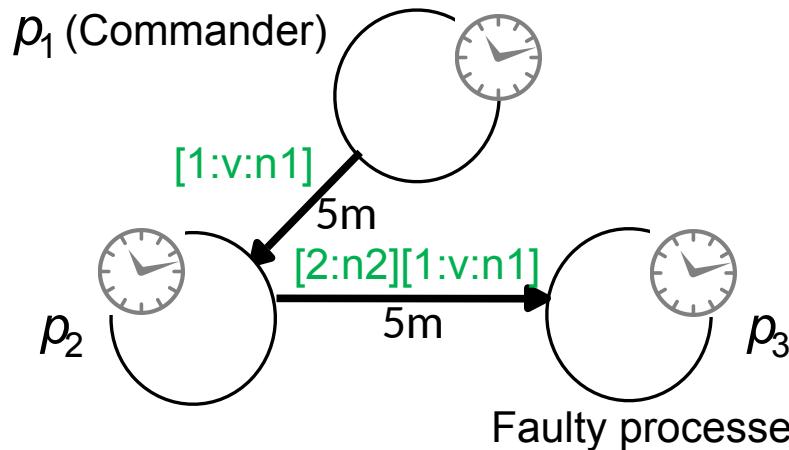
Different model: each message is signed and verified by the public key of sender.

With Signed Messages (Public-key)



Different model: each message is signed and verified by the public key of sender.

With Blockchains (Proof-of-Work)



Idea #1:

Each message takes exactly 5 minutes to create by any process. ("Magic Block")

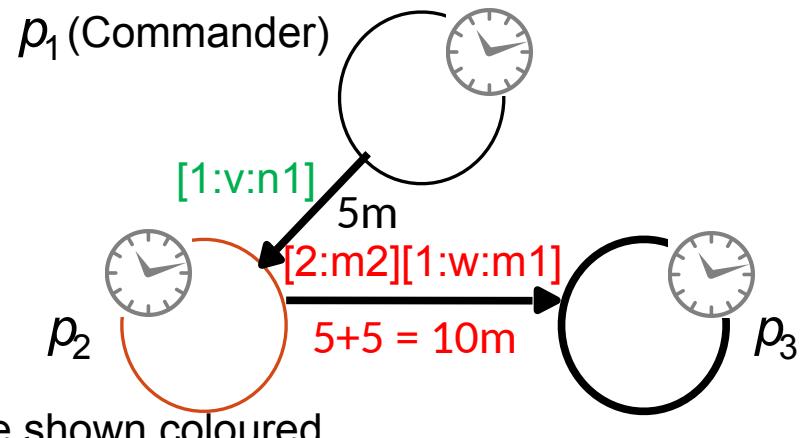
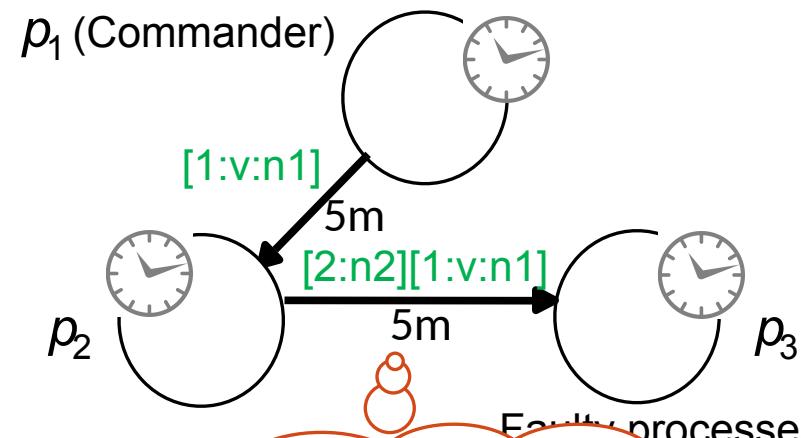


Idea #2:

Each process can accurately measure the amount of time taken by a process to create a message ("Magic Watch")



With Blockchains (Proof-of-Work)



Faulty processes are shown coloured

P3 verifies each block and receives them in time.

Each message takes exactly 5 minutes to create by any process. ("Magic Block")

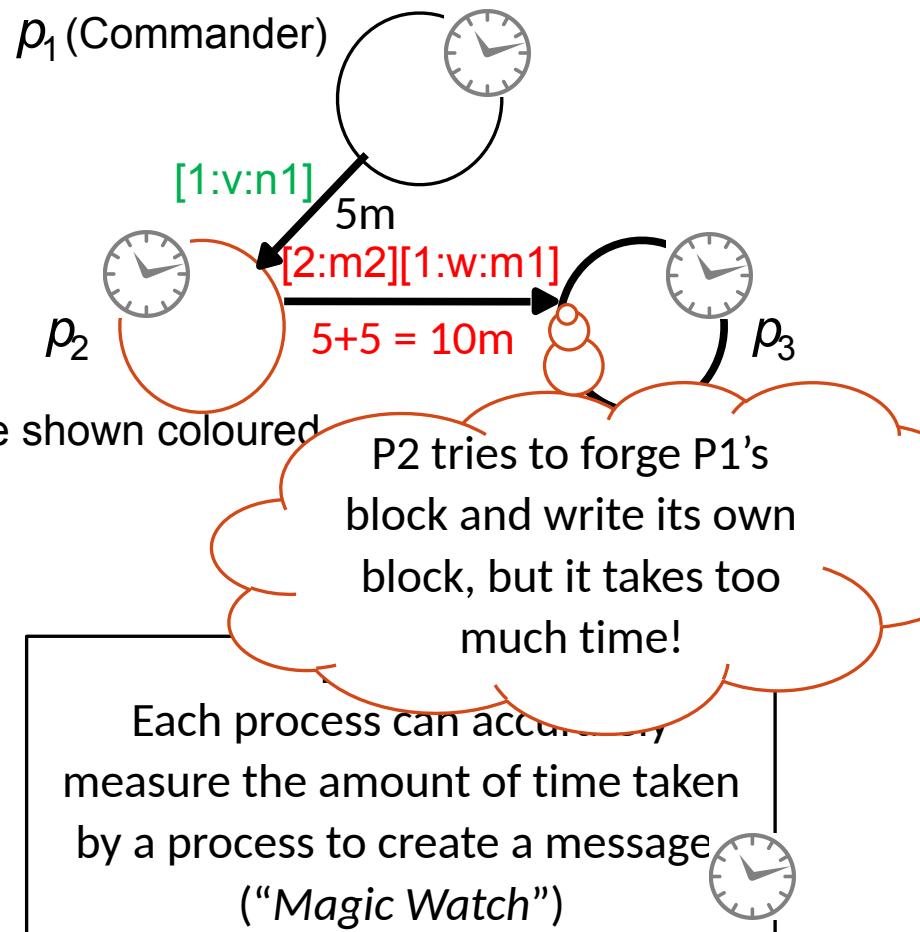
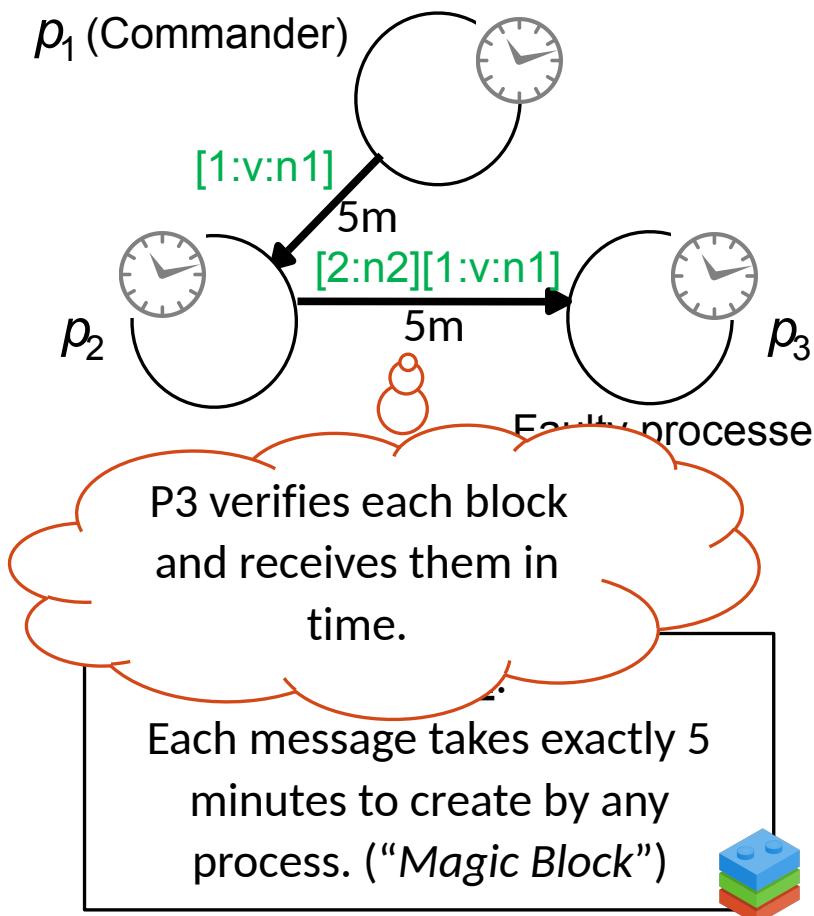


Idea #2:

Each process can accurately measure the amount of time taken by a process to create a message ("Magic Watch")



With Blockchains (Proof-of-Work)



Blockchain “Puzzles”

verify(nonce, data) meets some “requirements”

Use of “trapdoor functions” (hash functions)

- Cannot reverse the function to find the input
- Therefore, keep trying random values (called nonce) until you find a solution
- Like trying random combinations to a lock...
- The more computing power you have,
the faster you can solve the puzzle.
- “Magic blocks” are blocks with puzzles,
where everyone has the same power.



Proof-of-Work Example

E.g., the challenge is:

- sha256sum("data:nonce") starts with an "0"
- Normally more complicated than that! (e.g., 18 zeroes)

➤ P1 wants to send "1:v" to P2

```
kzhang@grey:~$ echo "1:v:118" | sha256sum
9479038ca7543ece09f48e8c77fce147d7561cac14058199afea18c2f323b8b
kzhang@grey:~$ echo "1:v:119" | sha256sum
79ae2bbac929112a349c2fe7f50210355f4a24683b2dd1ea8f059c9beeed7fd6
kzhang@grey:~$ echo "1:v:120" | sha256sum
002ce3a3b7092d960abf1795a89f70eb0f9ef960036e7d4620cbd3d26d34ffc8
```

➤ Send "1:v:120" to p2

Proof-of-Work Example

- P2 verifies “1:v:120” is correct (very quick!)
- P2 wants to send “2:1:v:120” to P3

```
kzhang@grey:~$ echo "2:1:v:120:119" | sha256sum  
911ab1edf1f331ff423a45fe4c382db30a3f1cf802bb2211df53c80d5798c7ba  
kzhang@grey:~$ echo "2:1:v:120:120" | sha256sum  
5344a3561673b1481b9cf69493368ca408b1edef67e3f96819c5d1b36cea53ce  
kzhang@grey:~$ echo "2:1:v:120:121" | sha256sum  
0a908c651e9ec5374976dc8f49a3342a4a789660011551da8871a6cc123c5b57
```

- P2 sends “2:1:v:120:121”
- P3 verifies “1:v:120” AND “2:1:v:120:121” are correct
- If P2 wants to send “2:1:w” and fool P3, it needs to find n1 for “1:w:n1” & n2 for “2:1:w:n1:n2”
- If P3 has a way to detect that P2 is *doing too much work*, it can detect fraud.



Bitcoin

- Introduced in 2008 by “Satoshi Nakamoto”.
- Decentralized cryptocurrency called BTC not backed by any entity (**not fiat currency**)
- Allows users to perform transactions **securely** without a middleman (e.g., bank, notary) and without trusting other users
- Miners form a P2P network managing a blockchain which includes validated transactions
- 21 million bitcoins limit reached in 2140

Bitcoin vs. Blockchain

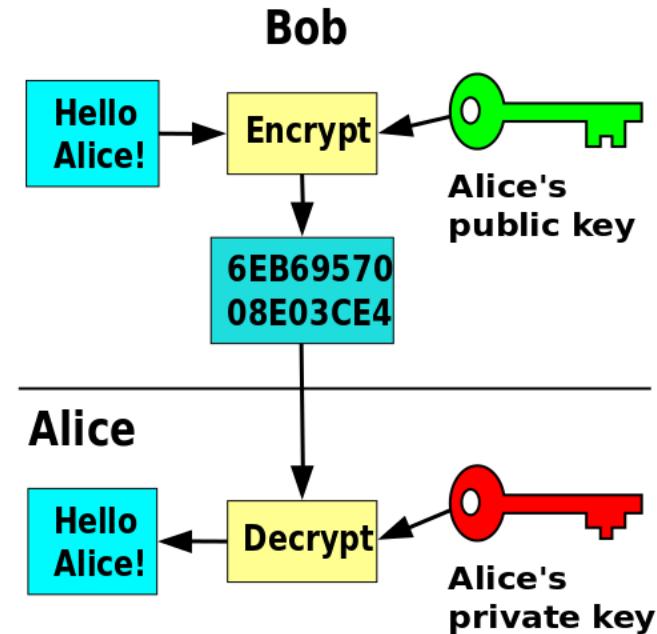
- Bitcoin is a specific system
 - Open-source implementation
 - There are alternative cryptocurrency systems , but they are not Bitcoin
- Blockchain is ambiguous: can be the data structure used in Bitcoin or a separate concept
 - A guiding design principle/paradigm
 - Not even a standard
 - Generalization of Bitcoin
 - Hundreds of implementations
 - Ethereum alone has hundreds of deployments in addition to the main public deployment

Wallets and Addresses

- Users require a Bitcoin wallet to store money
- Wallet is authenticated by private/public-key
 - Generated using ECDSA (Elliptic curve cryptography)
- Money is sent to the wallet by addressing its public-key
- The owner proves it owns the money by encrypting transactions using the private key
- If you lose the private key, the money in the wallet disappears from the world!
- But also don't share it!

Public Key Cryptography (Asymmetrical Cryptography)

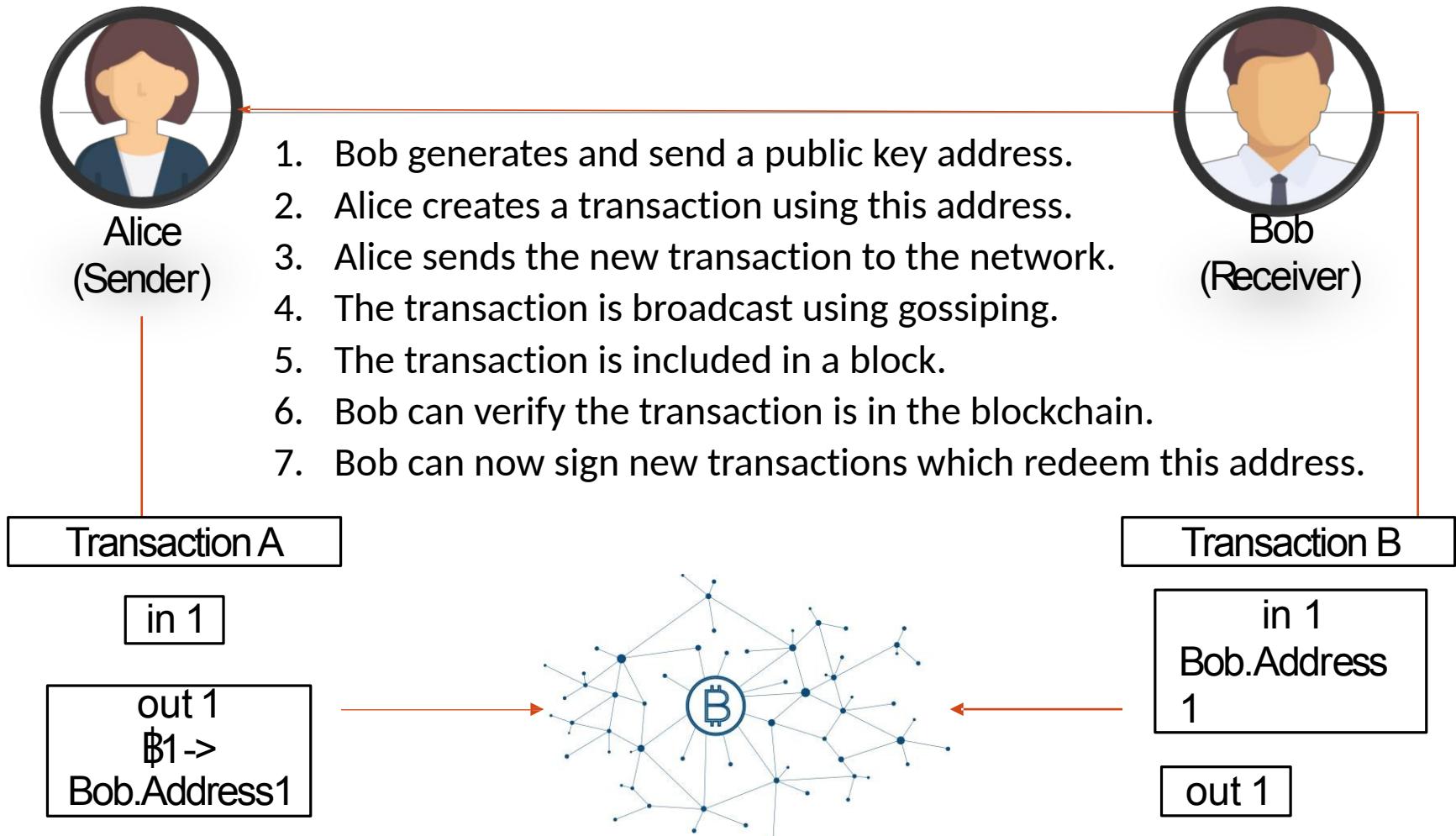
- Recipient's public key is used to encrypt the plaintext to ciphertext
- Recipient's private key to decrypt the ciphertext to original plaintext
- No one can use the public key to decrypt the ciphertext to plaintext



Communication in Bitcoin

- Broadcast to all the network:
 - Users broadcast their transactions
 - Miners broadcasts updates to the blockchain (new blocks)
- Implemented via **gossiping protocol** in a P2P network
 - Not terribly efficient but has not been a bottleneck so far
- Works because financial transactions are very short and their rate in Bitcoin is far below that of credit cards
- Needs to be fairly reliable for the system to work but 100 percent reliability in message delivery is not required
 - Users and miners need to detect message loss and retransmit messages if needed
- Message propagation should be reasonably fast
 - Slower network quantifiably increases the risk of attacks

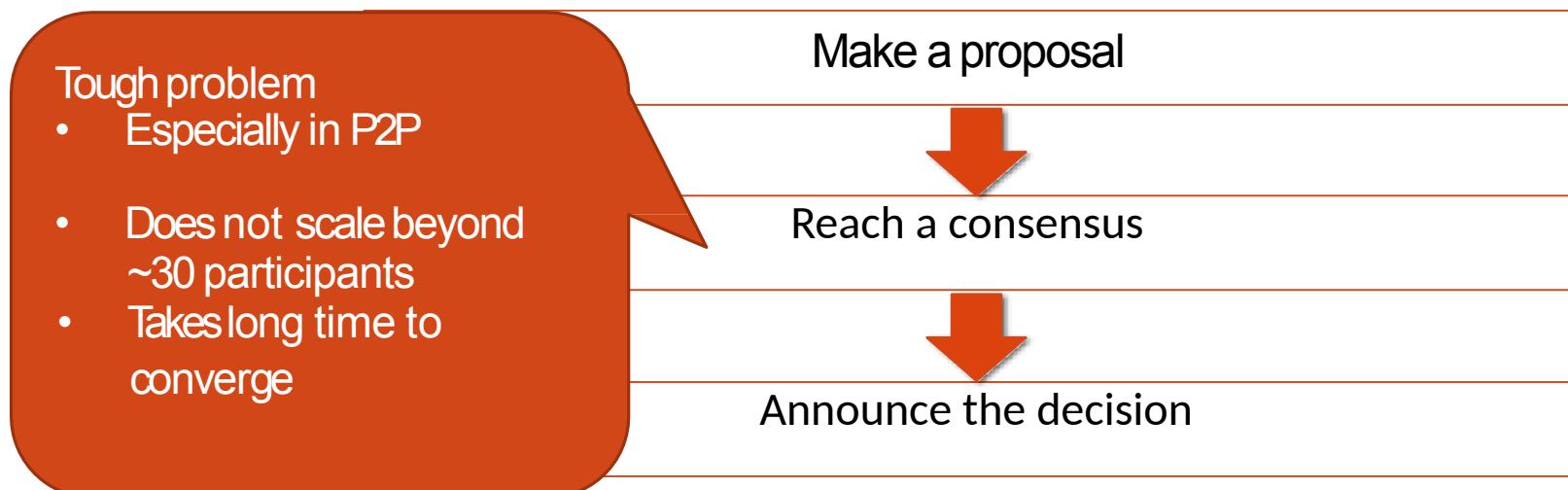
Transaction Flow



Consensus in Bitcoin

The network needs to agree on

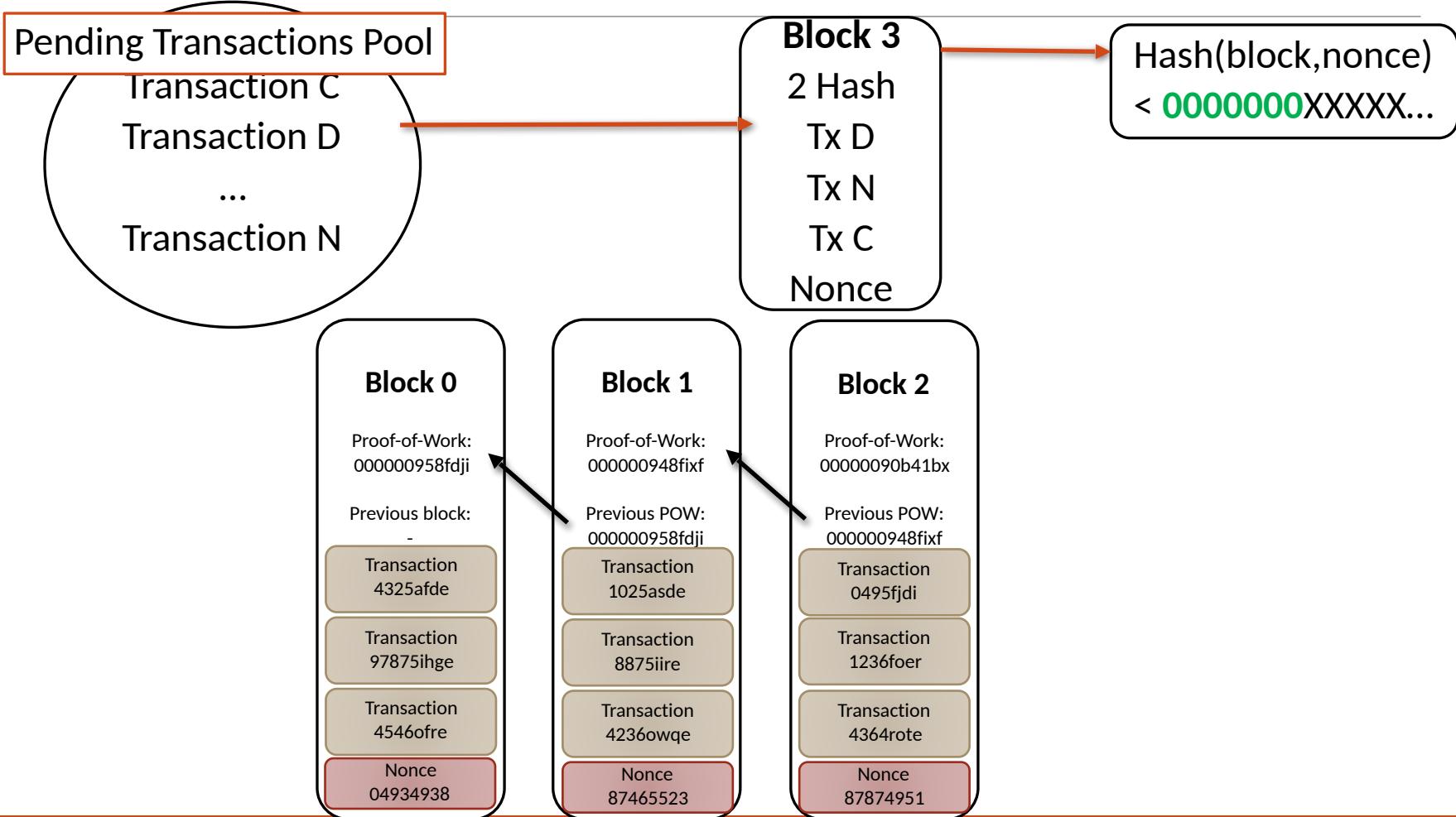
- Which recently broadcast transactions go into the blockchain
- In what order



Challenge 1: who proposes and when?

- The network cannot sustain each and every user or peer making a proposal whenever she wishes
- Need to moderate the number of proposers and rate of concurrent proposals
 - While keeping them sufficiently high
- Several principal solutions
 - Proof-of-work: need to do heavy computation and show the proof of it
 - Proof-of-stake: need to possess a sufficient amount of coins
- Important optimization: propose new transactions in batches
 - A block in Bitcoin is structured as a tree of proposed transactions

Proof-of-Work in Bitcoin



Pending transactions are propagated to the peers (miners)

Proof-of-Work in Bitcoin

Pending Transactions Pool

Transaction C

Transaction D

...

Transaction N

Block 3

2 Hash

Tx D

Tx N

Tx C

Nonce

Hash(block,nonce)

< 0000000XXXXX...

Block 0

Proof-of-Work:
000000958fdji

Previous block:

Transaction
4325afde

Transaction
97875ihge

Transaction
4546ofre

Nonce
04934938

Block 1

Proof-of-Work:
000000948fixf

Previous POW:
000000958fdji

Transaction
1025asde

Transaction
8875iire

Transaction
4236owqe

Nonce
87465523

Block 2

Proof-of-Work:
00000090b41bx

Previous POW:
000000948fixf

Transaction
0495fjdi

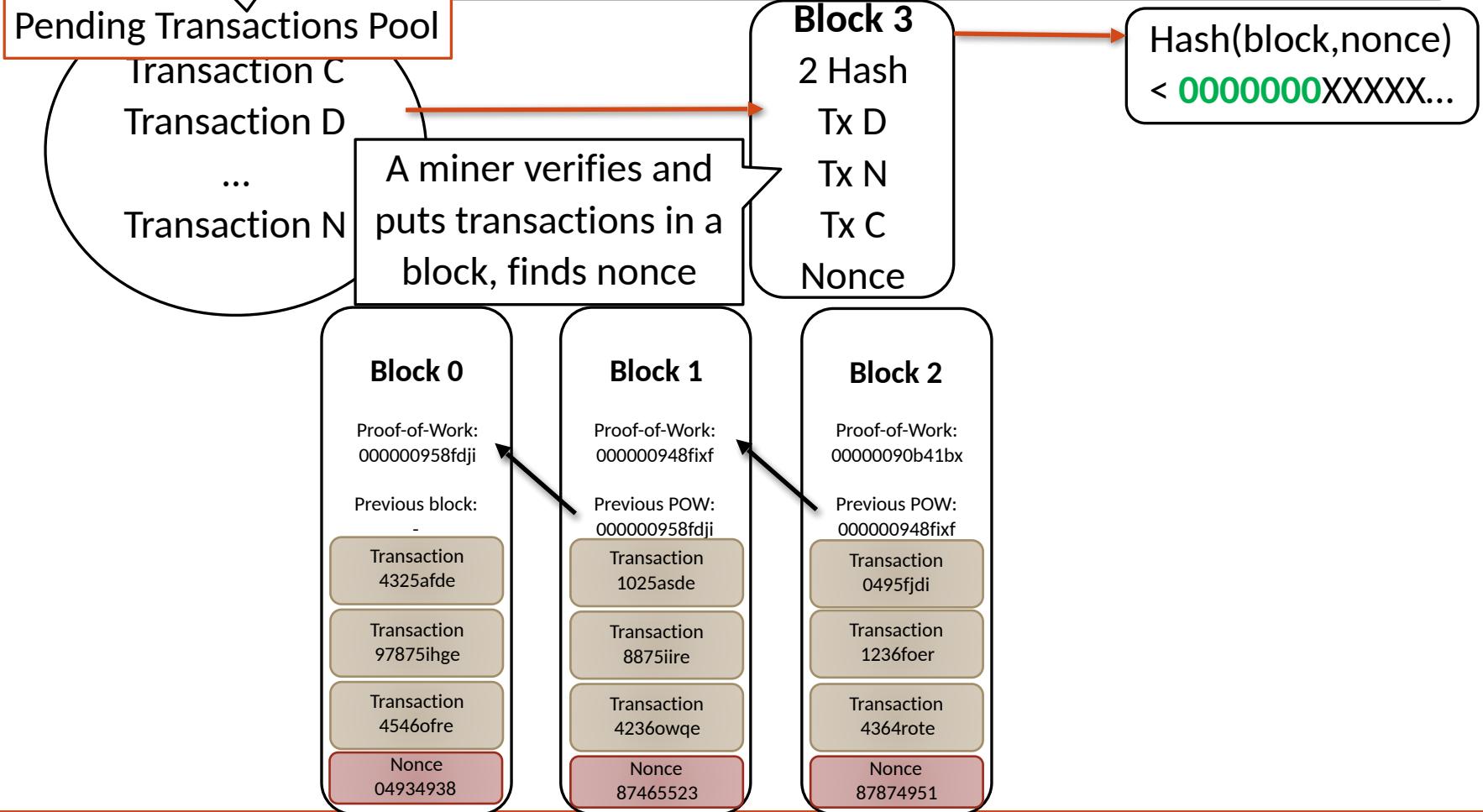
Transaction
1236foer

Transaction
4364rote

Nonce
87874951

Pending transactions are propagated to the peers (miners)

Proof-of-Work in Bitcoin



Pending transactions are propagated to the peers (miners)

Proof-of-Work in Bitcoin

Pending Transactions Pool

Transaction C

Transaction D

...

Transaction N

A miner verifies and puts transactions in a block, finds nonce

Block 3

2 Hash

Tx D

Tx N

Tx C

Nonce

Hash(block,nonce)

< 0000000XXXXX...

Number of leading zeroes (difficulty) depend on the global **hash-rate**, s.t. one block is solved **per 10 minutes**

Block 0

Proof-of-Work:
000000958fdji

Previous block:

Transaction
4325afde

Transaction
97875ihge

Transaction
4546ofre

Nonce
04934938

Block 1

Proof-of-Work:
000000948fixf

Previous POW:

Transaction
1025asde

Transaction
8875iire

Transaction
4236owqe

Nonce
87465523

Block 2

Proof-of-Work:
00000090b41bx

Previous POW:

Transaction
0495fjdi

Transaction
1236foer

Transaction
4364rote

Nonce
87874951

Pending transactions are propagated to the peers (miners)

Proof-of-Work in Bitcoin

Pending Transactions Pool

Transaction C

Transaction D

...

Transaction N

A miner verifies and puts transactions in a block, finds nonce

Block 3

2 Hash

Tx D

Tx N

Tx C

Nonce

Hash(block,nonce)

< 0000000XXXXX...

Number of leading zeroes (difficulty) depend on the global **hash-rate**, s.t. one block is solved **per 10 minutes**

Block 0

Proof-of-Work:
000000958fdji

Previous block:

Transaction
4325afde

Transaction
97875ihge

Transaction
4546ofre

Nonce
04934938

Block 1

Proof-of-Work:
000000948fixf

Previous POW:

Transaction
1025asde

Transaction
8875iire

Transaction
4236owqe

Nonce
87465523

Block 2

Proof-of-Work:
00000090b41bx

Previous POW:

Transaction
0495fjdi

Transaction
1236foer

Transaction
4364rote

Nonce
87874951

Block 3

Proof-of-Work:
000000r9d8fjj

Previous block:

Transaction
D

Transaction
N

Transaction
C

Nonce
79146512

The miner attaches the solved block to the chain, or stops solving if someone else finds a valid block.

Pending transactions are propagated to the peers (miners)

Proof-of-Work in Bitcoin

Pending Transactions Pool

Transaction C

Transaction D

...

Transaction N

A miner verifies and puts transactions in a block, finds nonce

Block 3

2 Hash

Tx D

Tx N

Tx C

Nonce

Hash(block,nonce)

< 0000000XXXXX...

Number of leading zeroes (difficulty) depend on the global **hash-rate**, s.t. one block is solved **per 10 minutes**

The more **confirmations** a transaction receive, the less likely it is to disappear.

Block 0

Proof-of-Work:
000000958fdji

Previous block:

Transaction
4325afde

Transaction
97875ihge

Transaction
4546ofre

Nonce
04934938

Block 1

Proof-of-Work:
000000948fixf

Previous POW:
000000958fdji

Transaction
1025asde

Transaction
8875iire

Transaction
4236owqe

Nonce
87465523

Block 2

Proof-of-Work:
00000090b41bx

Previous POW:
000000948fixf

Transaction
0495fjdi

Transaction
1236foer

Transaction
4364rote

Nonce
87874951

Block 3

Proof-of-Work:
000000r9d8fjj

Previous block:
00000090b41bx

Transaction
D

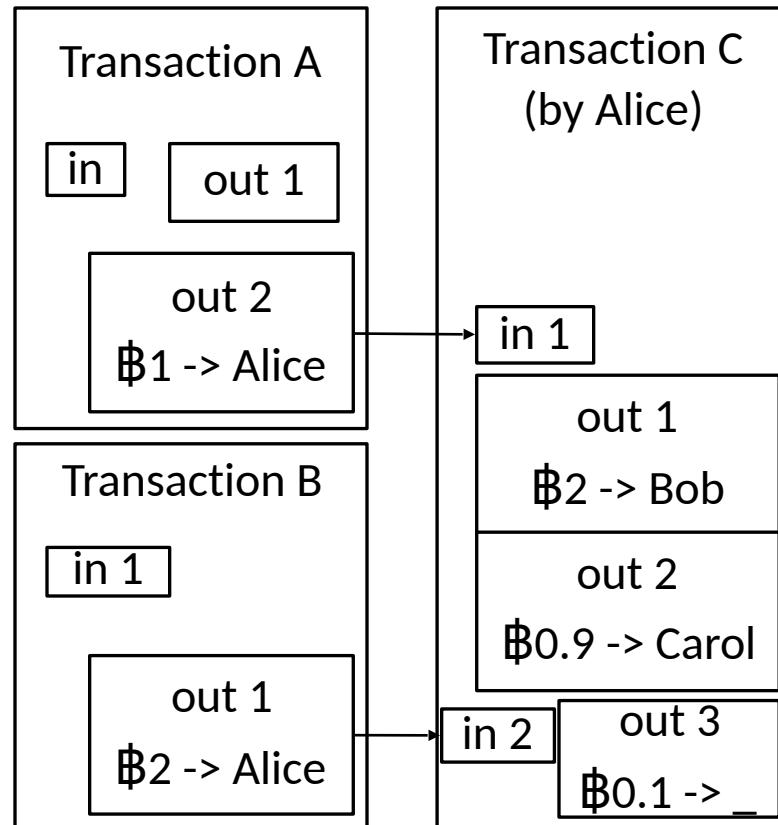
Transaction
N

Transaction
C

Nonce
79146512

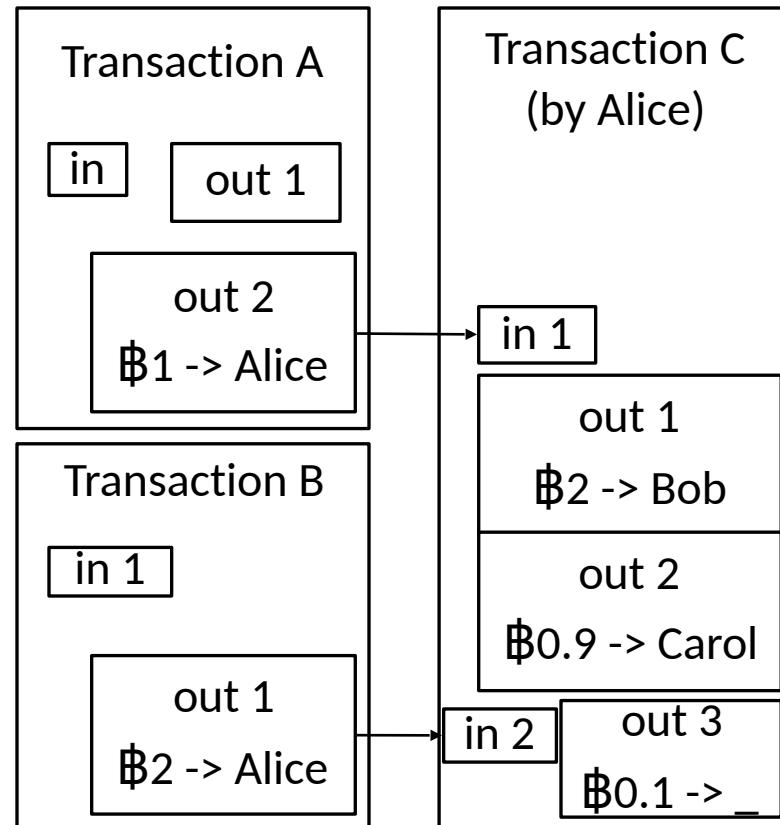
The miner attaches the solved block to the chain, or stops solving if someone else finds a valid block.

Bitcoin Transactions



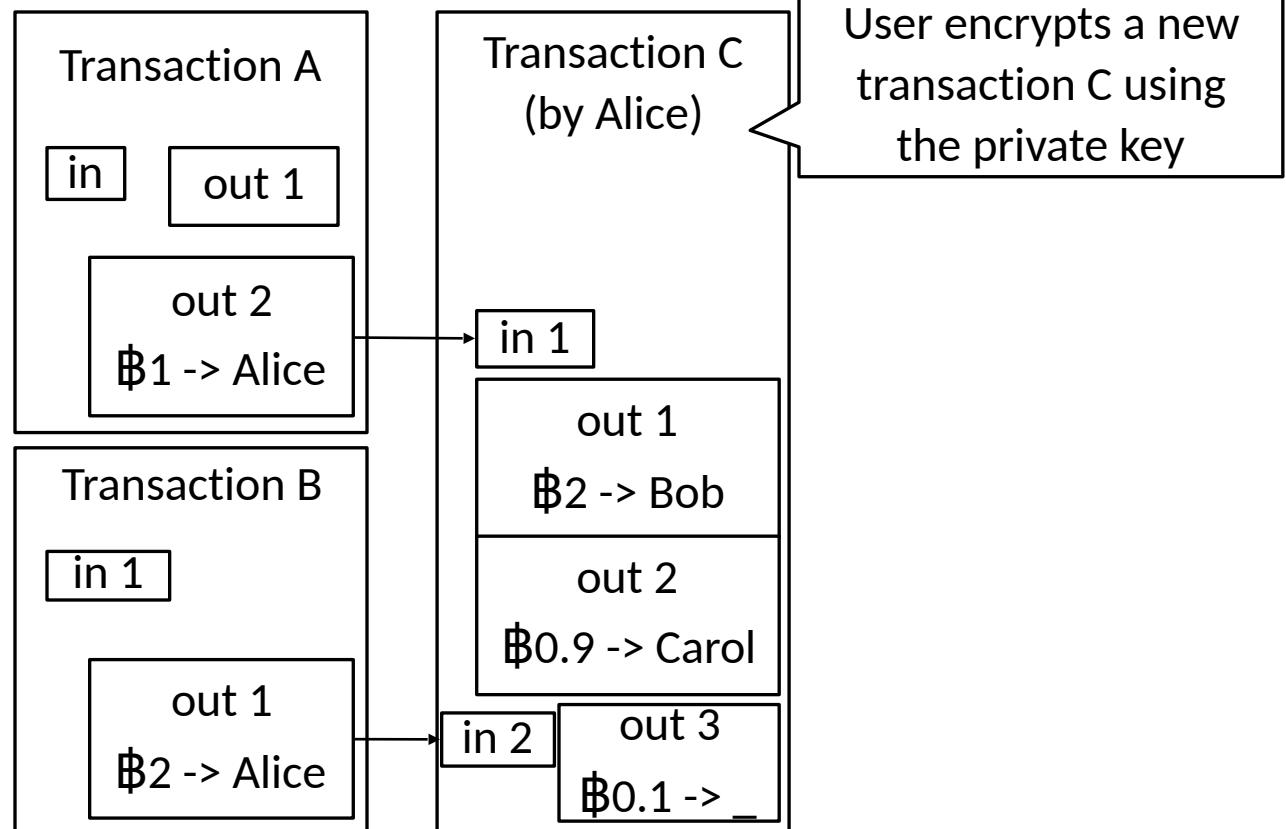
Bitcoin Transactions

Each user possesses a wallet identified by *public/private key pairs*



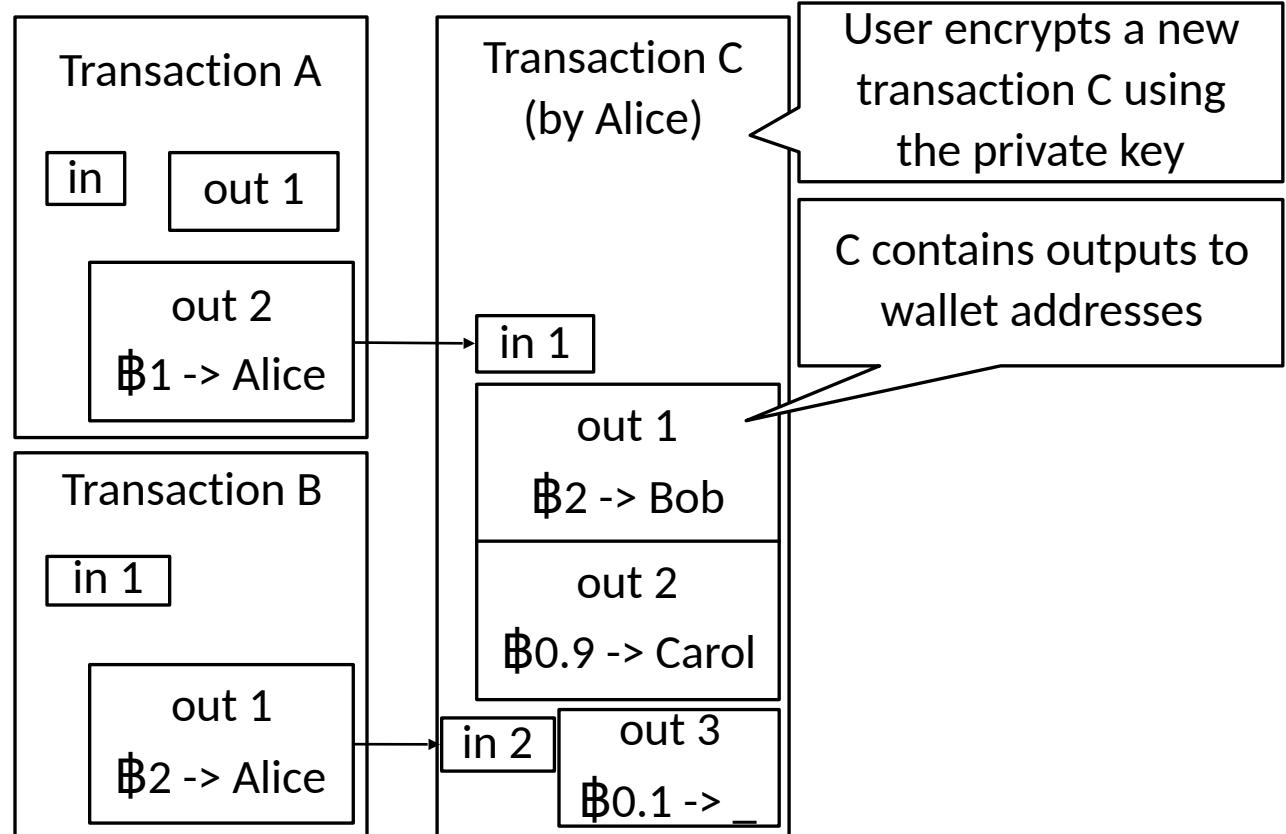
Bitcoin Transactions

Each user possesses a wallet identified by *public/private key pairs*



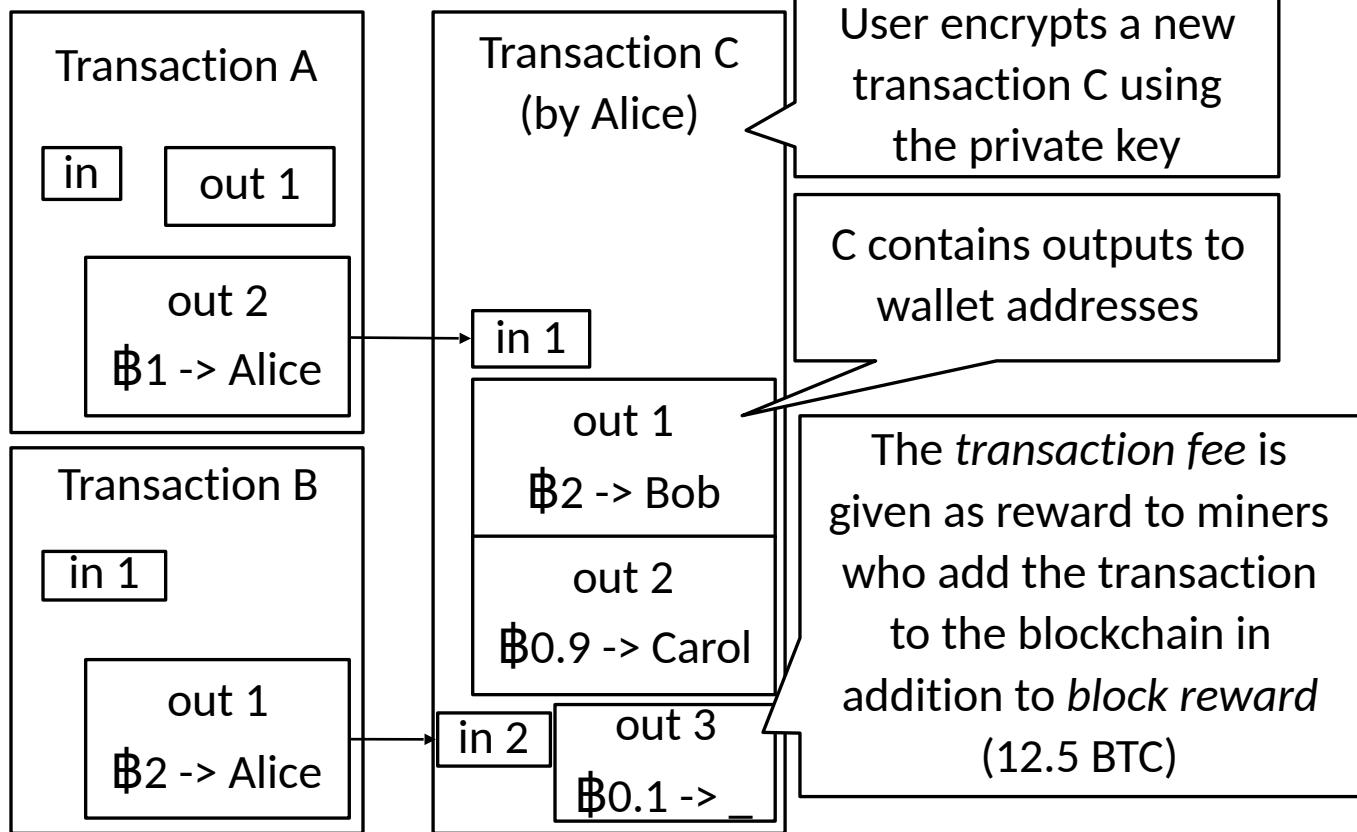
Bitcoin Transactions

Each user possesses a wallet identified by *public/private key pairs*



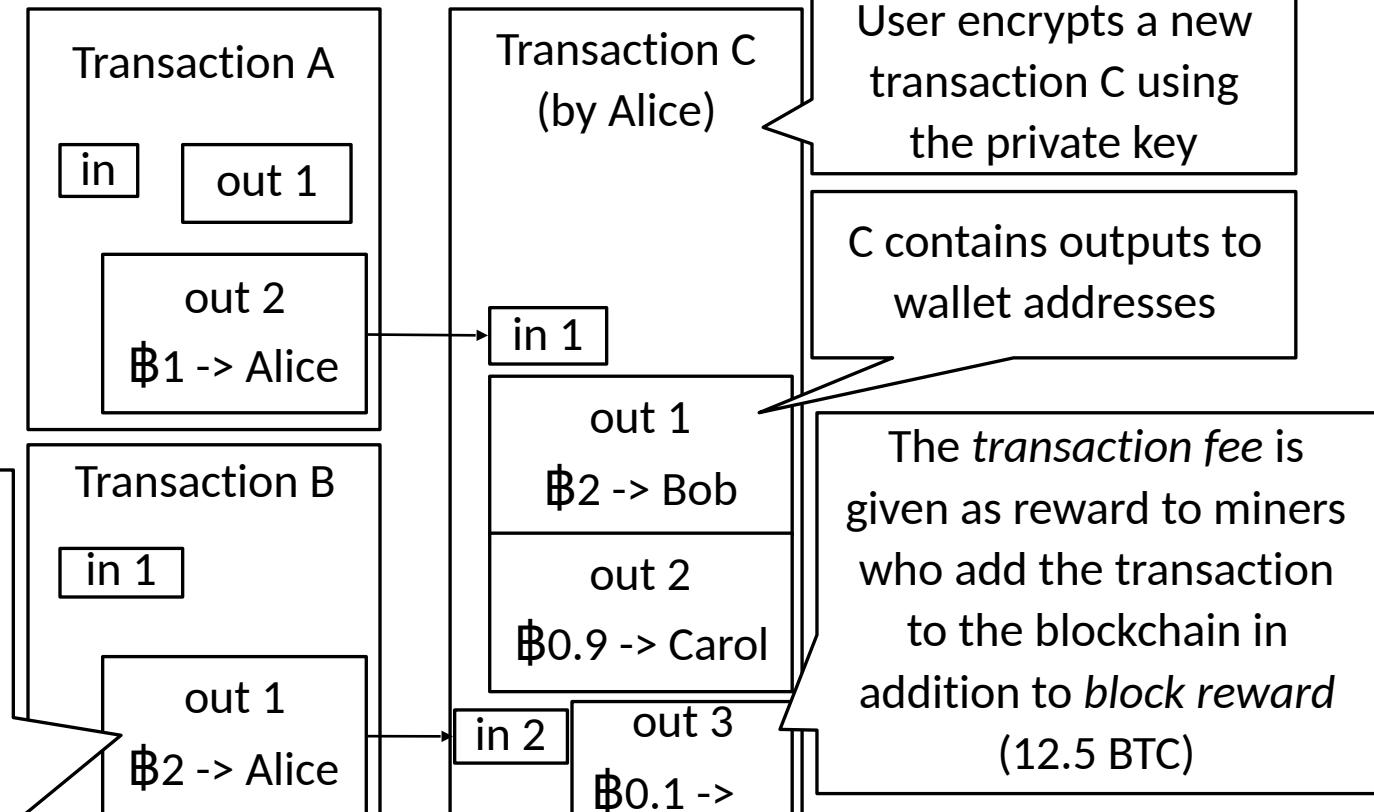
Bitcoin Transactions

Each user possesses a wallet identified by *public/private key pairs*



Bitcoin Transactions

Each user possesses a wallet identified by *public/private key pairs*



To do so, tx C must reference *unspent transactions outputs* (UTXOs) from previously committed blocks equal to the total output of tx C (3 BTC)

User encrypts a new transaction C using the private key

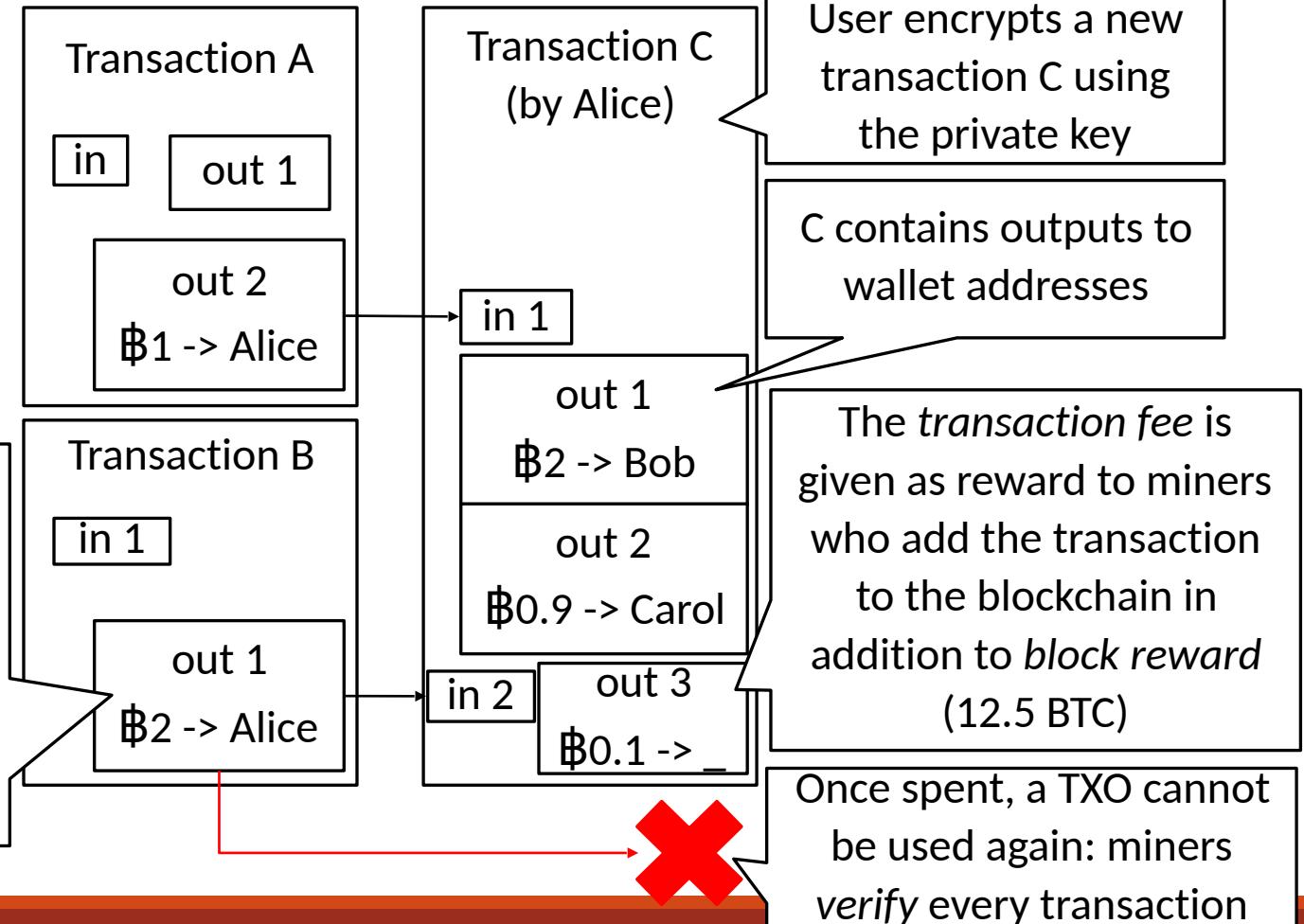
C contains outputs to wallet addresses

The *transaction fee* is given as reward to miners who add the transaction to the blockchain in addition to *block reward* (12.5 BTC)

Bitcoin Transactions

Each user possesses a wallet identified by *public/private key pairs*

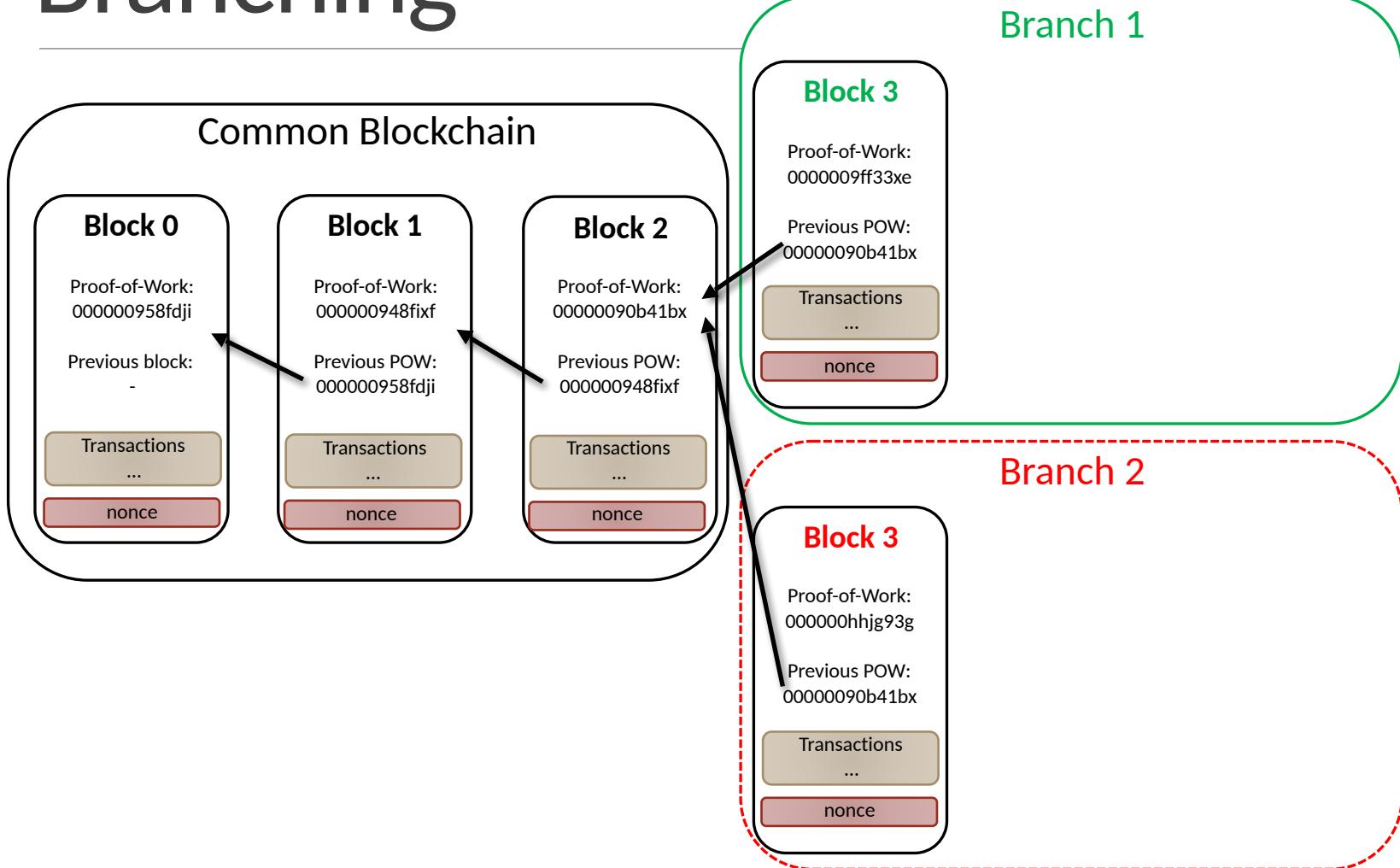
To do so, tx C must reference *unspent transactions outputs* (UTXOs) from previously committed blocks equal to the total output of tx C (3 BTC)



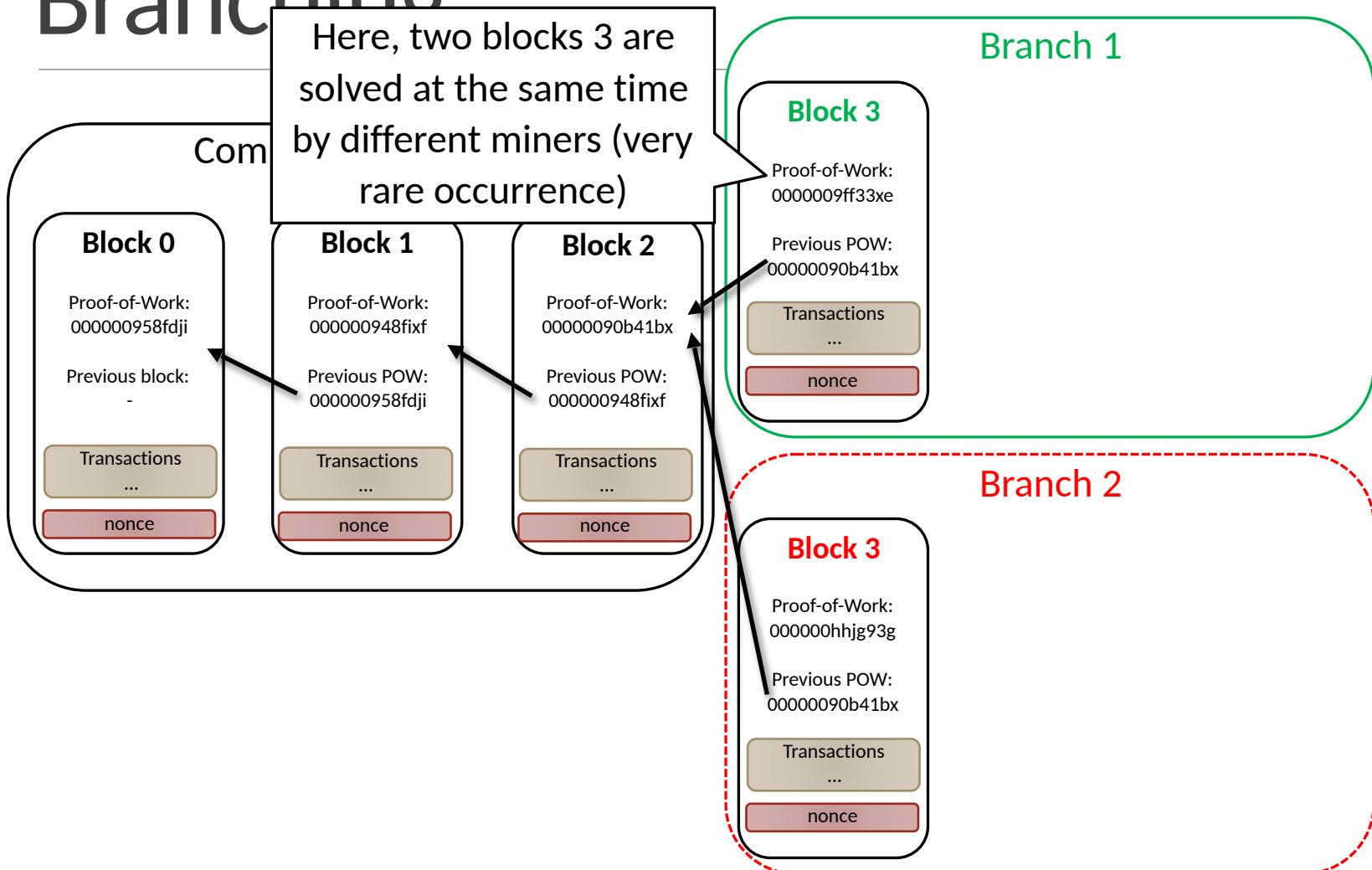
Reaching Consensus in Bitcoin

- A miner broadcasts the proposed block
- When a peer receives a proposed block
 - Check that the proof of cryptopuzzle solution is valid
 - Check that each transaction is valid (business logic)
 - If the hash pointer is valid, append the new block to the local copy of the blockchain
 - Conflict resolution: if the proposed chain is longer than the current local copy, replace the local copy
- Local copies may diverge!
 - Lost messages and concurrent blocks arriving in reverse order
- Probabilistic convergence over time is proven when using the longest chain for conflict resolution
 - The probability of a block being non-final decreases exponentially with the number of later blocks stored in the chain
 - The standard client sends a confirmation after six later blocks stored in the chain
 - Takes an order of one hour in practice

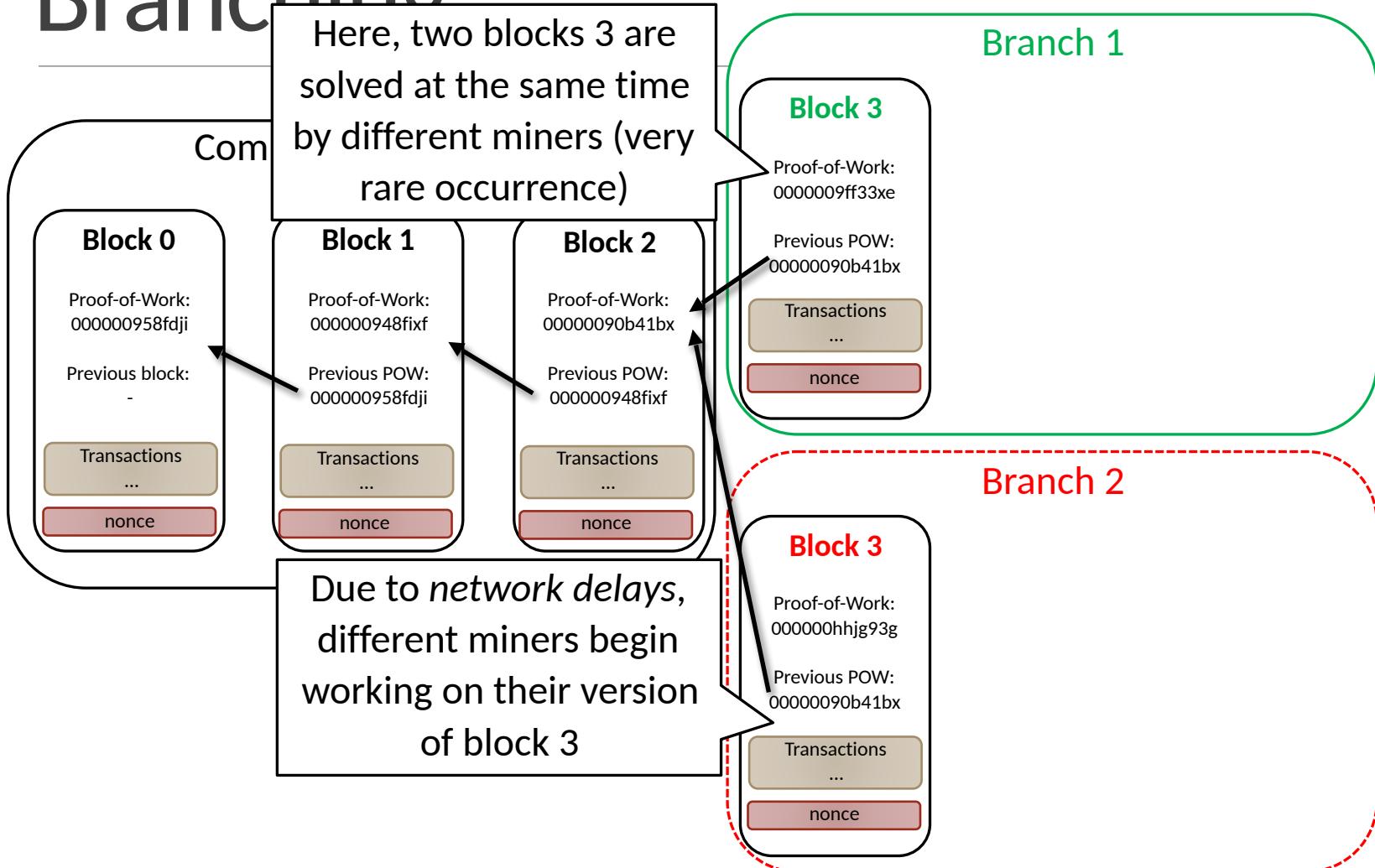
Branching



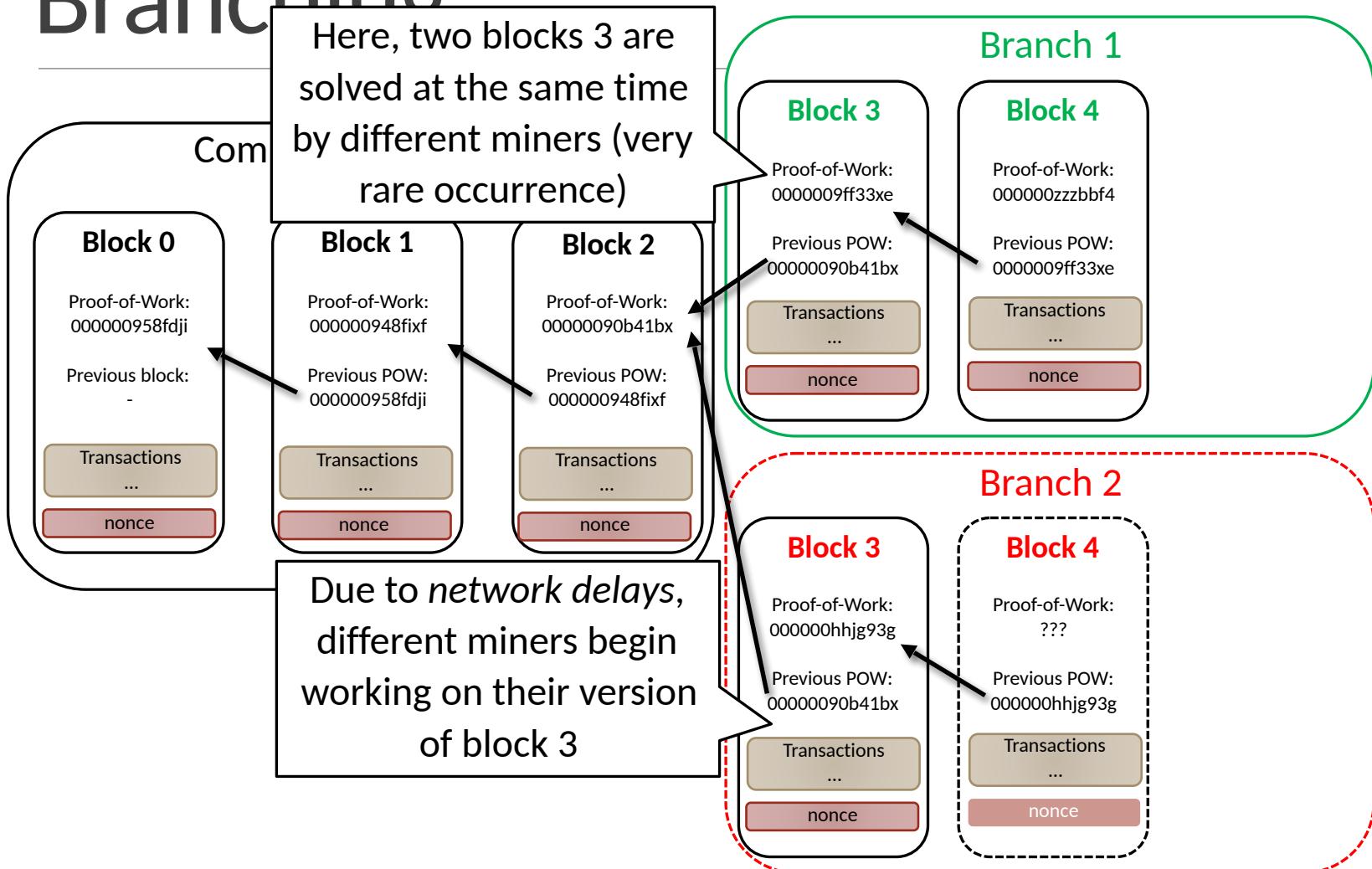
Branching



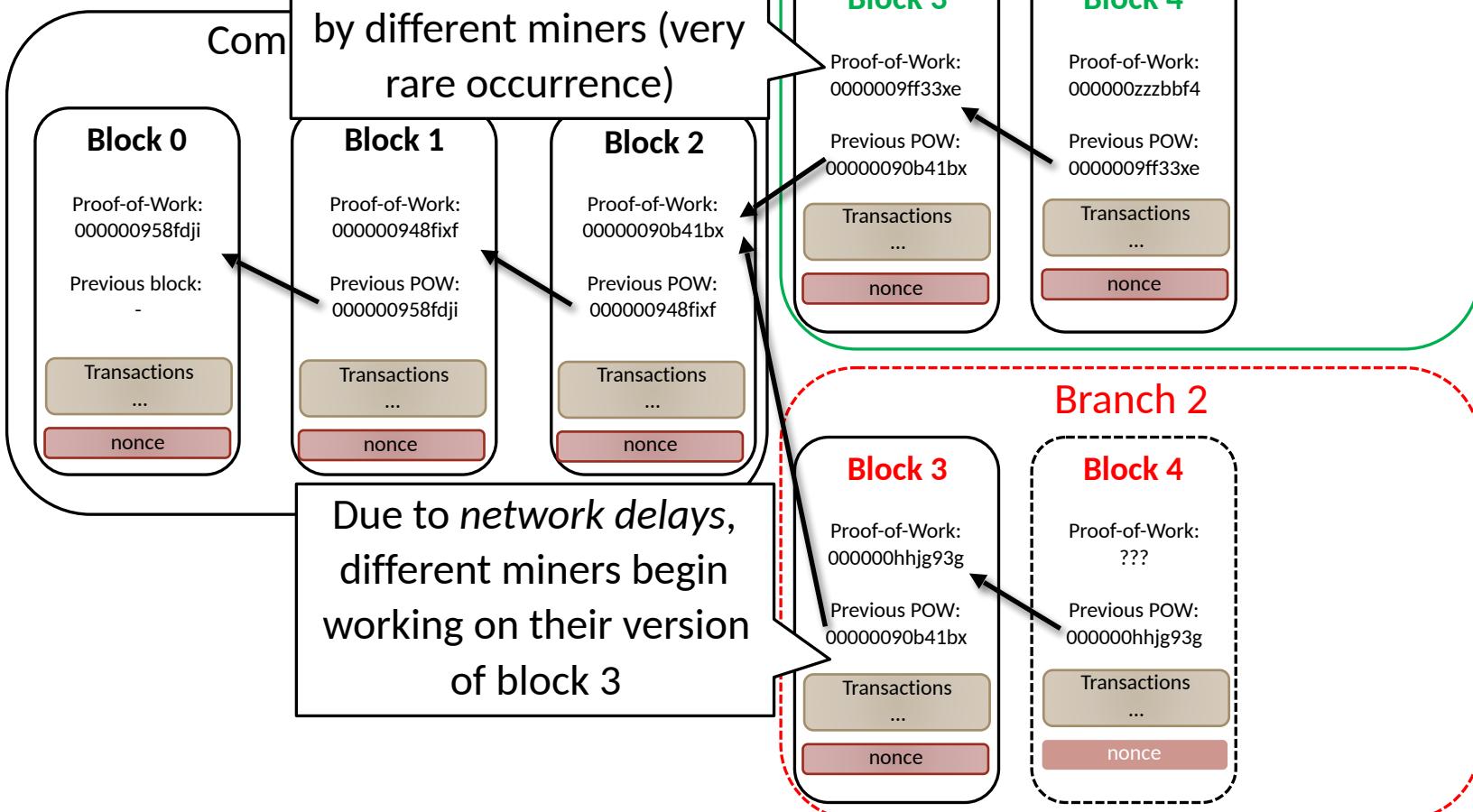
Branching



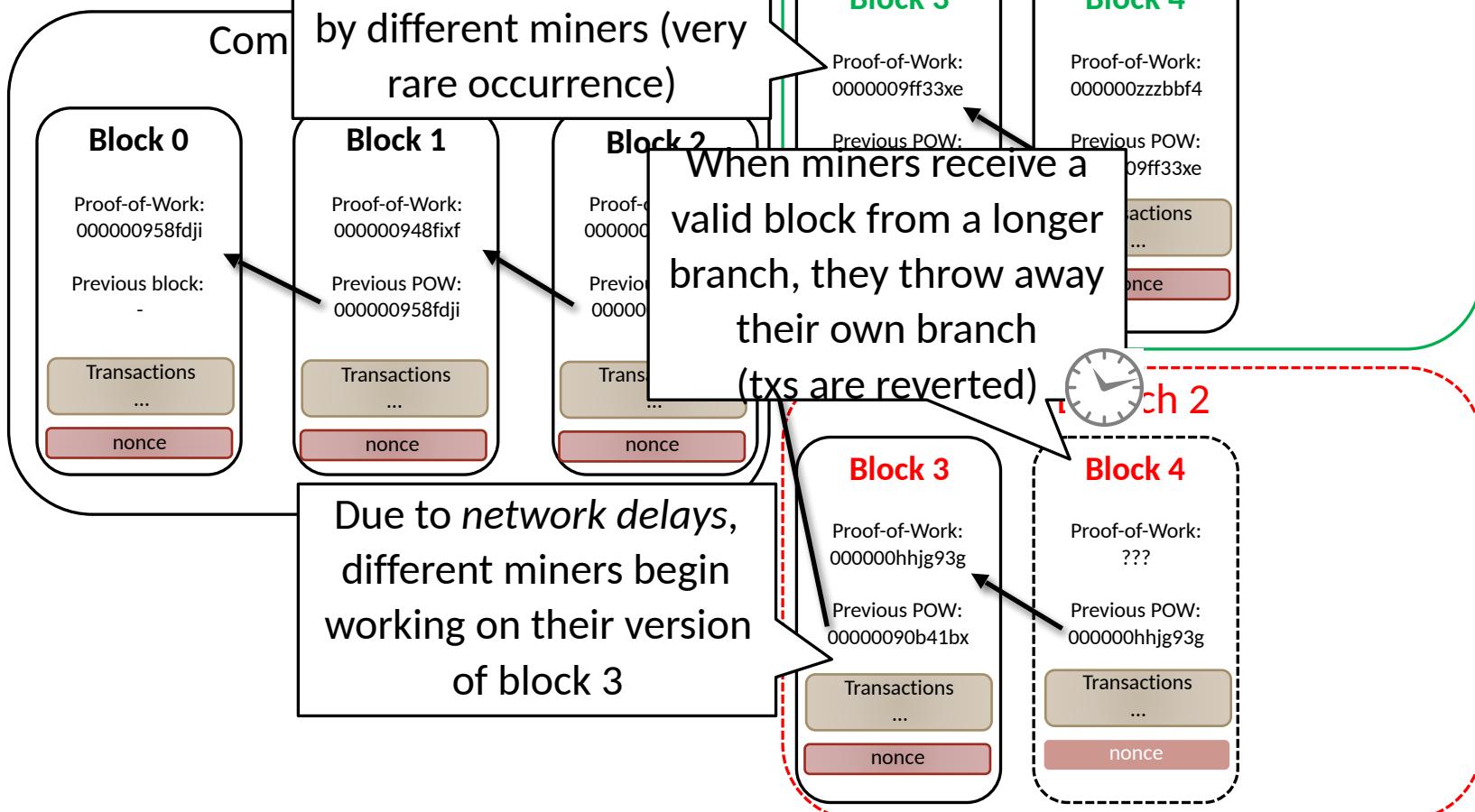
Branching



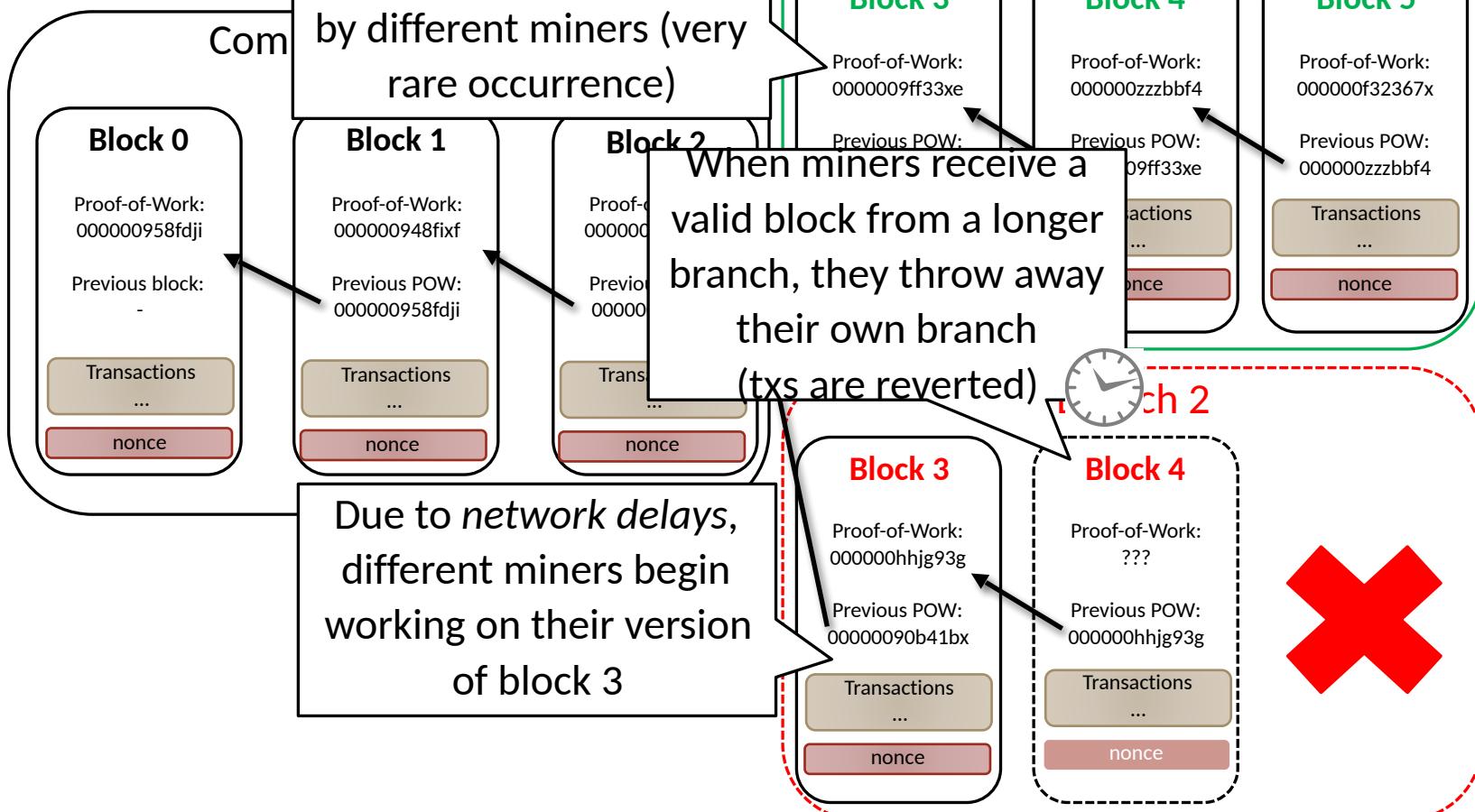
Branching



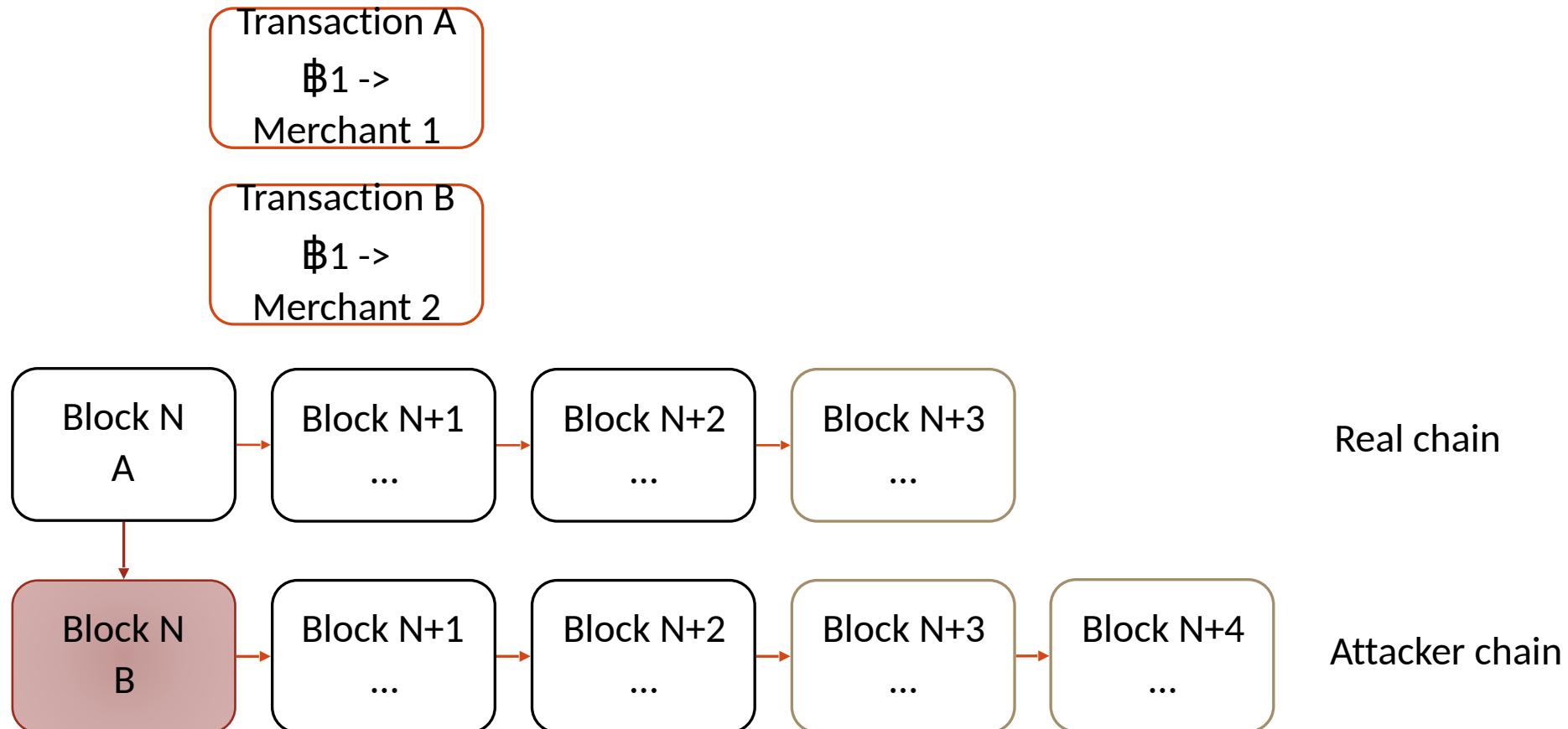
Branching



Branching



Preventing Double Spending: 51% Attack



Preventing Double Spending: 51% Attack

Transaction A

฿1 ->

Merchant 1

A malicious attacker creates two transactions using the same money (*double-spending*)

Transaction B

฿1 ->

Merchant 2

Block N

A

Block N+1

...

Block N+2

...

Block N+3

...

Real chain

Block N

B

Block N+1

...

Block N+2

...

Block N+3

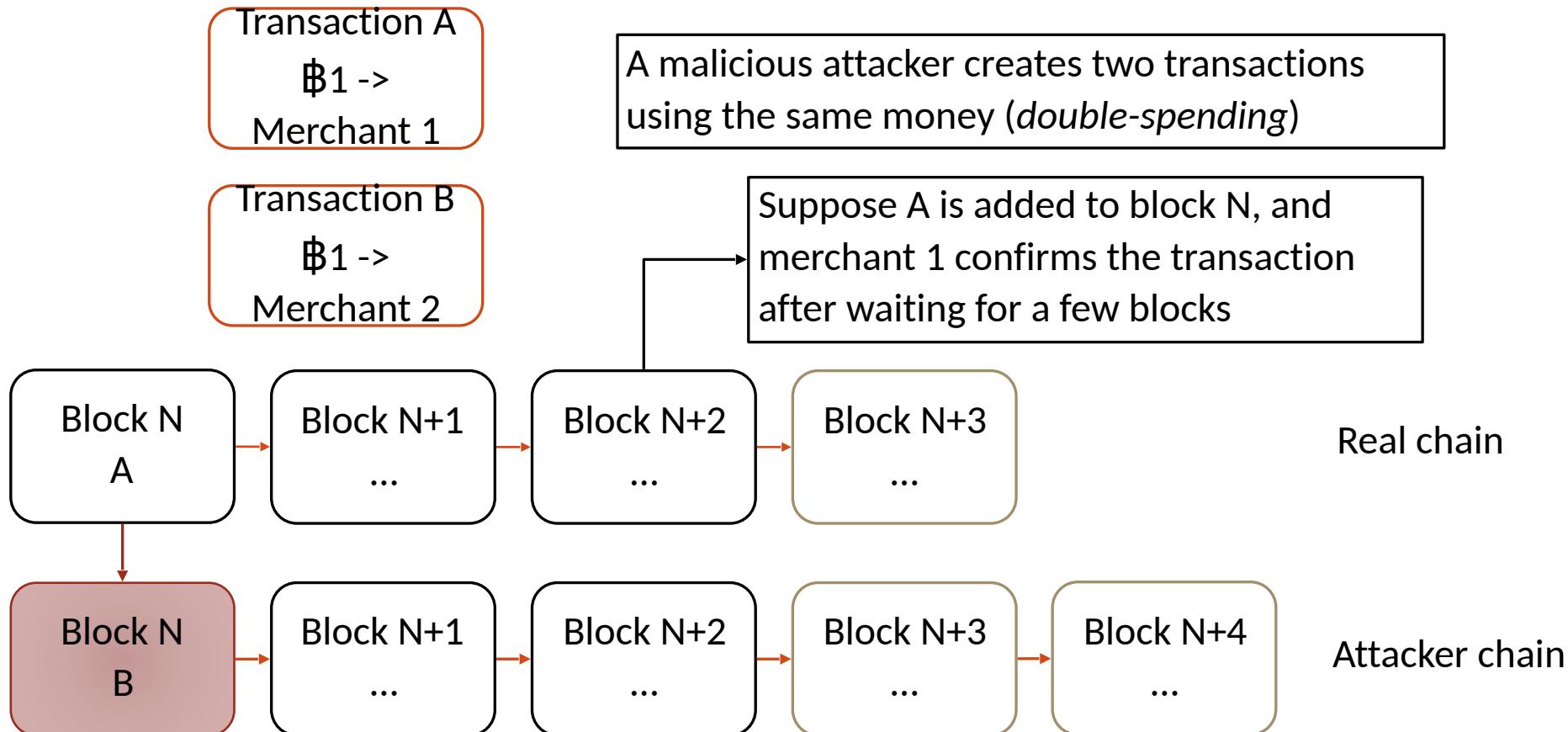
...

Block N+4

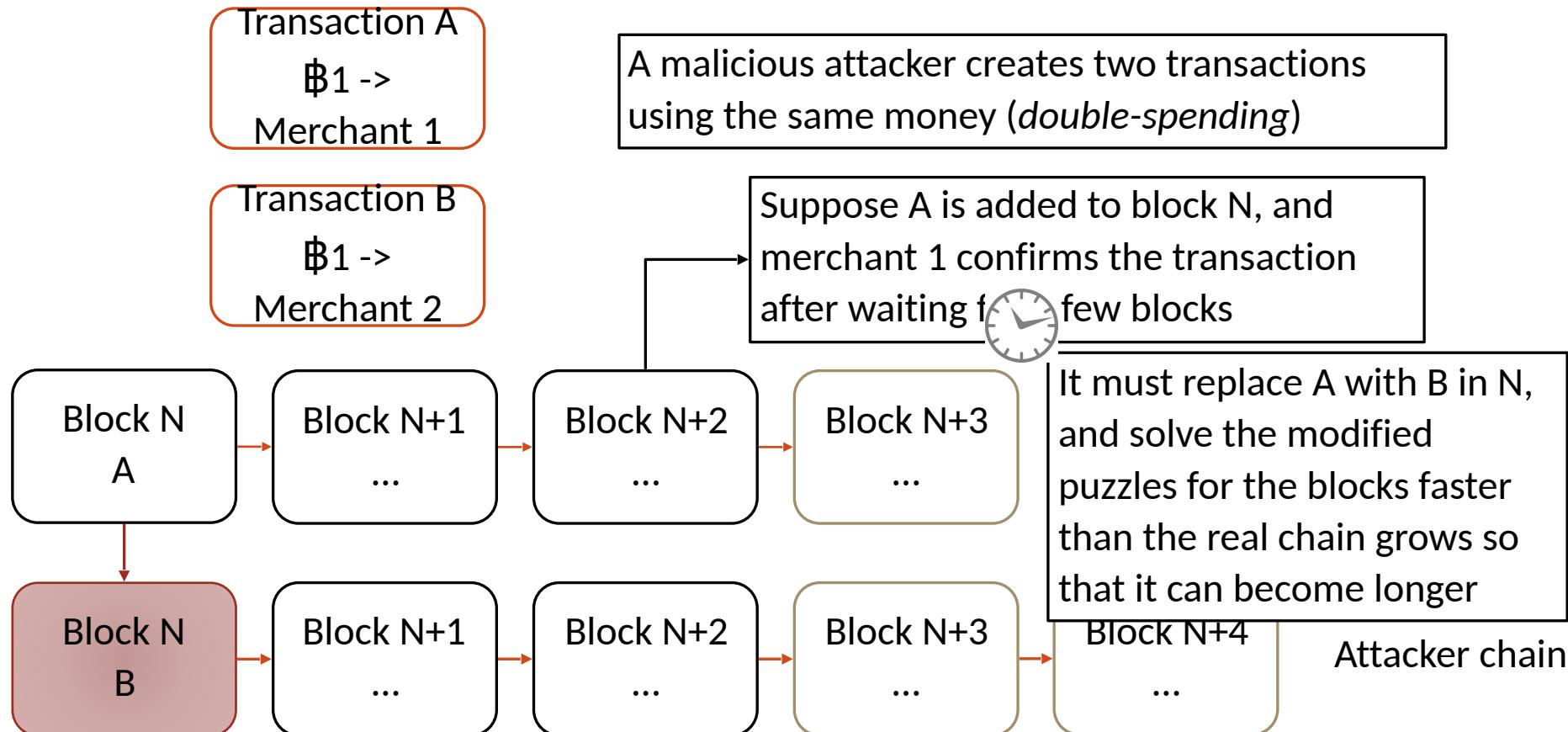
...

Attacker chain

Preventing Double Spending: 51% Attack



Preventing Double Spending: 51% Attack

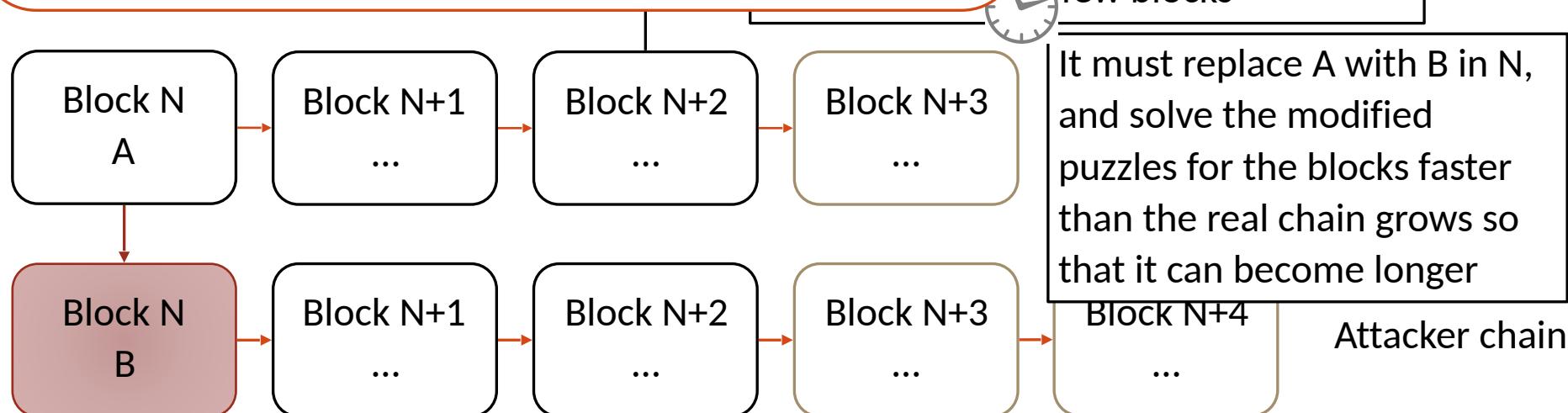


Preventing Double Spending: 51% Attack

- The “Magic Watch” is the *continuous generation* of blocks in the main chain which *limits the amount of time* an attacker has to create its own chain.
- If the attacker owns *>51% of the power* in the network, the “Magic Watch” gives *enough time* to the attacker to *tamper the data!*

ates two transactions
double-spending)

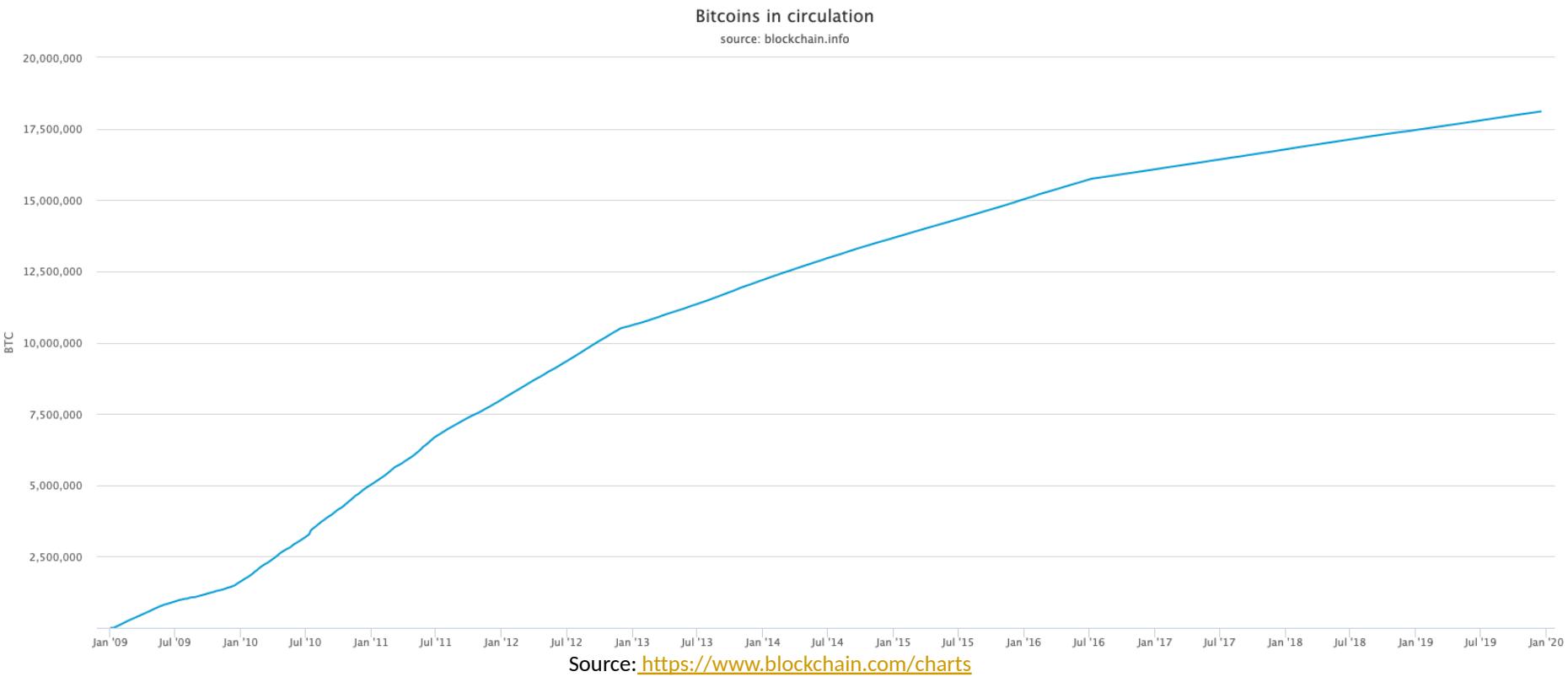
ded to block N, and
irms the transaction
few blocks



Challenge 2: Why propose non-empty blocks? Bitcoin Rewards

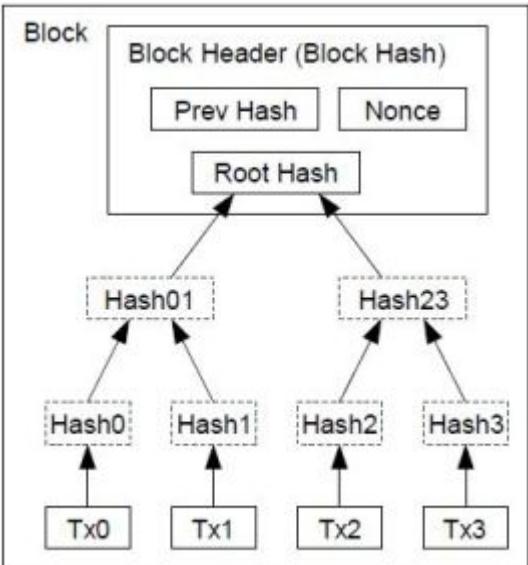
- When a miner successfully solves a block, it is allowed to include a transaction which sends a **reward to its own wallet**.
- In addition, it can collect **all fees** attached to the transactions included in the block.
- Rewards are halved every 210k blocks (4 years).
- Started at 50 BTC!
- Now 12.5 BTC, eventually it will hit 0, miners are then only incentivized by transaction fees.

Bitcoins in Circulation



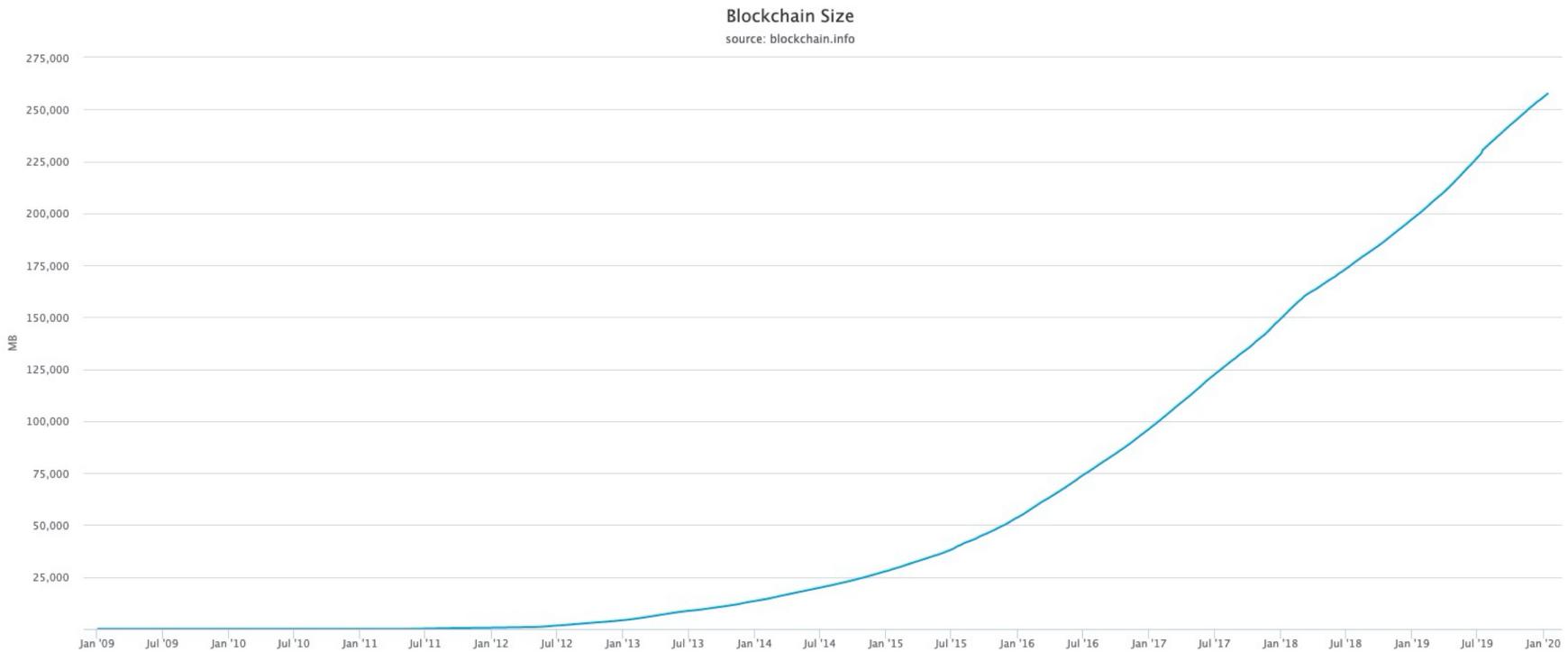
Data Structure within a Block

Merkle Tree



- ❑ To avoid hashing the entire block data when computing PoW, only the root hash of the Merkle tree is included.
- ❑ Spent transactions can be pruned, leaving only the necessary intermediate nodes to save space.
- ❑ For users without a full copy of the blockchain, simple payment verification (SPV) is used to verify if a specific transaction exists.
- ❑ A Merkle proof only requires block headers and related branches, e.g., Hash01, Hash2 (for Tx3).

Size of the Ledger



Source: <https://www.blockchain.com/charts>

Limitations of Bitcoin

Limited expressiveness

- Cryptocurrency only
- Each app requires new platform (e.g. NameCoin, PrimeCoin, CureCoin)

Slow block time (10 mins)

- Also slow confirmation time (1+ hour for 6 confirmations)

Hard/Soft forks

- Updates to the code cause forks
- Hard forks are not compatible
- Duplicated money
- 5 Bitcoin forks (e.g., Bitcoin Classic)

Slow transaction rate

- 7 transactions/second
- VISA Network: 2000 tps (average)
- Limited block size (1MB -> 2MB)

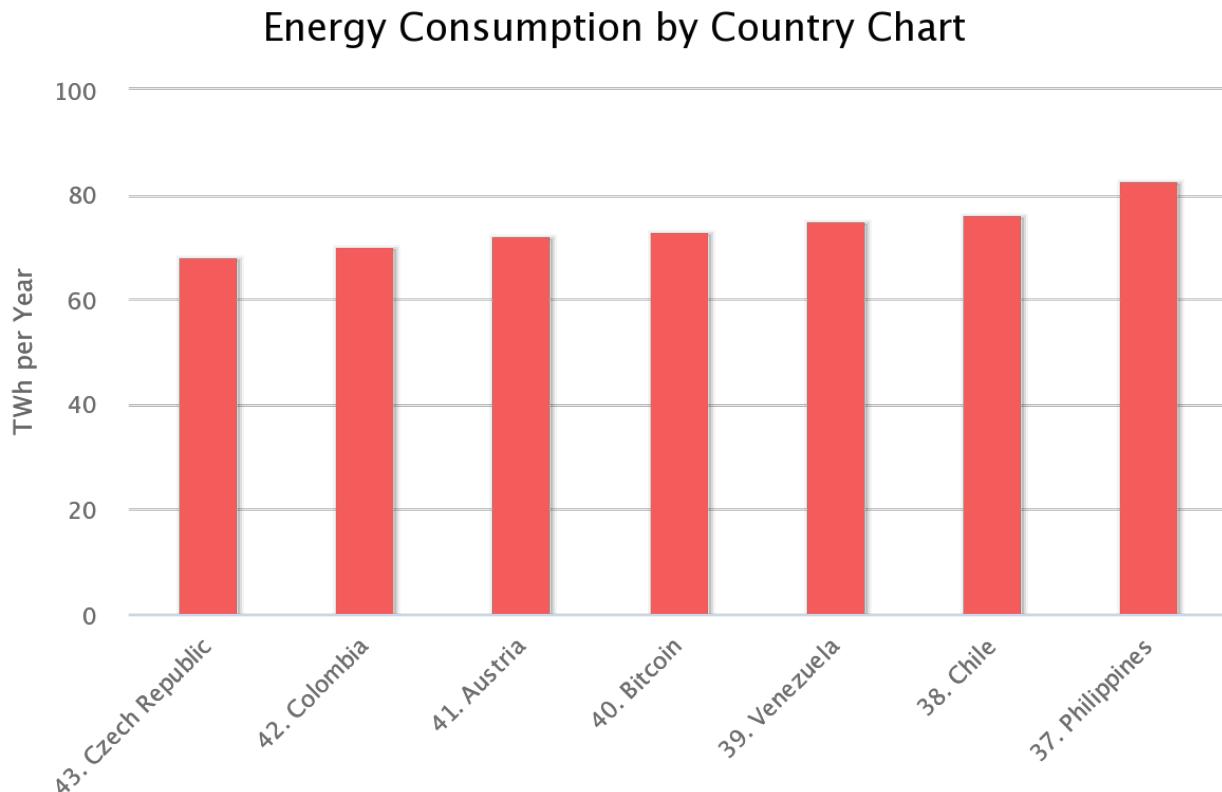
Weaknesses of proof-of-work

- Environmental impact: ~1000x more energy than credit card
- Ahead of 159 countries for energy consumption (e.g. austria)

Long bootstrap time for a miner

- Full ledger: 250 GB (2020/01)
- CPU/IO cost to verify each transaction/block
- Takes hours/days

Energy Consumption of Bitcoin



BitcoinEnergyConsumption.com

Energy Consumption of Bitcoin

Annualized Total Footprints

Carbon Footprint	Electrical Energy	Electronic Waste
34.73 Mt CO₂ 	73.12 TWh 	10.89 kt 

Comparable to the carbon footprint of Denmark.
Comparable to the power consumption of Austria.
Comparable to the e-waste generation of Luxembourg.

Single Transaction Footprints

Carbon Footprint	Electrical Energy	Electronic Waste
313.17 kgCO₂ 	659.30 kWh 	98.21 grams 

Equivalent to the carbon footprint of 782,922 VISA transactions or 52,195 hours of watching YouTube.
Equivalent to the power consumption of an average U.S. household over 22.28 days.
Equivalent to the weight of 1.51 'C'-size batteries or 2.14 golf balls. (Find more info on e-waste [here](#).)

Source: <https://digiconomist.net/bitcoin-energy-consumption>

Blockchain Platforms

ETHEREUM

HYPERLEDGER



ETHEREUM

Managing entity: Ethereum Foundation

- Major players: Deloitte, Toyota, Microsoft, ...

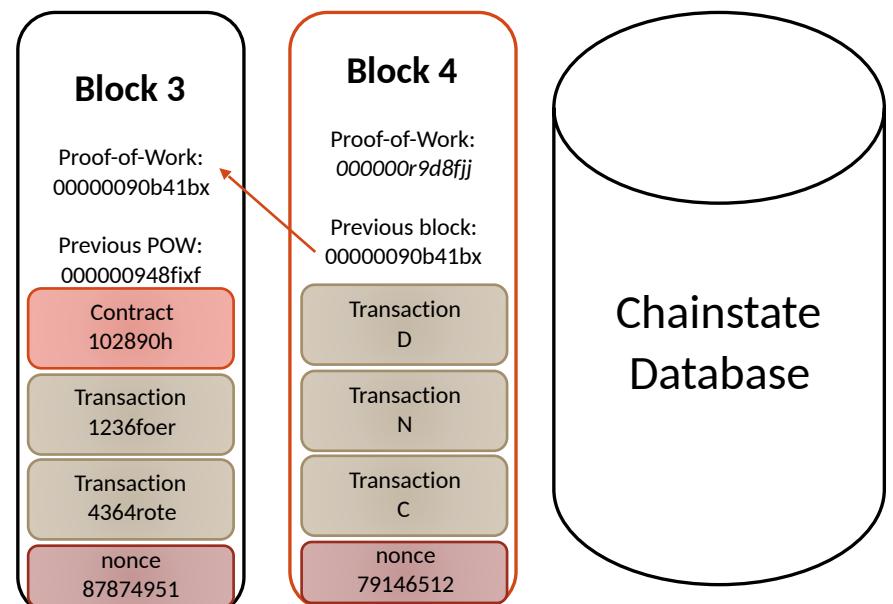
Focus: Open-source, flexible, platform

- Cryptocurrency: 1 Ether = 1e18 Wei
- Smart contracts: Solidity, Remix (Web IDE), Truffle (Dev./Test), *Viper*
- Ethereum Virtual Machine (EVM), Ethereum Web Assembly (eWASM)
- Permissionless (public) ledger: Proof-of-Work, *Proof-of-Stake (Casper)*

Evolution in Business Logic

- Proliferation of Bitcoin spawn-offs
 - Digital currency is not the only electronic object of value
 - Documents: authorizations, legal, diploma, design, various deliverables
 - Support for extended financial applications such as crowdfunding
 - Support for multi-party escrow transactions
 - Ethereum envisioned that a single platform supporting the above is better than hundreds of specialized systems
 - Provided a verifiable Turing-complete script language
 - With script templates
 - Scripts can be stateful, with a state stored on the chain

Smart Contracts

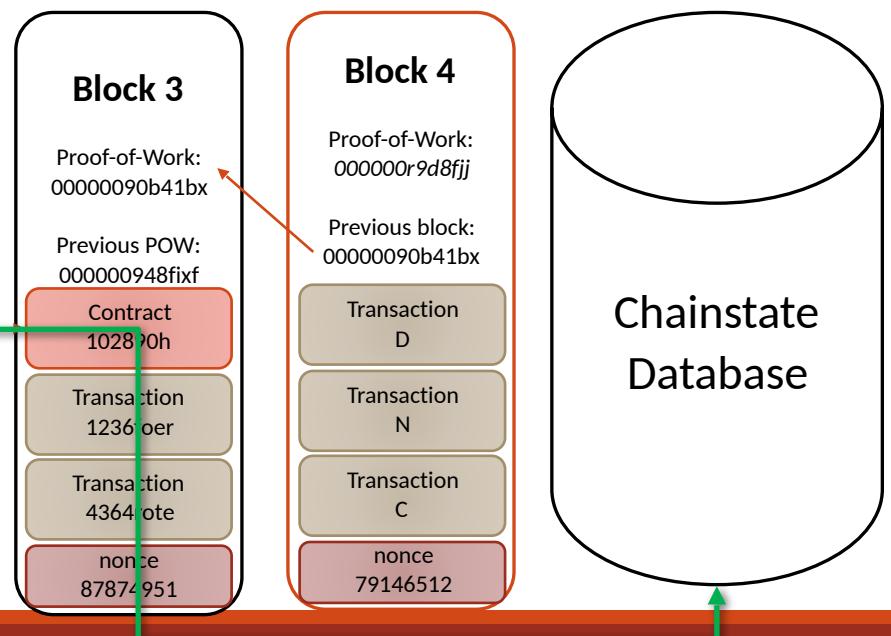


Smart Contracts

- Contracts contain *executable bytecode*
- Created with a blockchain tx
- Contracts have internal storage

```

1 • <contract>
2   └──
3     └──
4       └──
5         └──
6           └──
7             └──
8               └──
9                 └──
10                └──
11                  └──
12                    └──
13                      └──
14                        └──
15                          └──
16                            └──
17                              └──
18                                └──
19
20 • </contract>
```



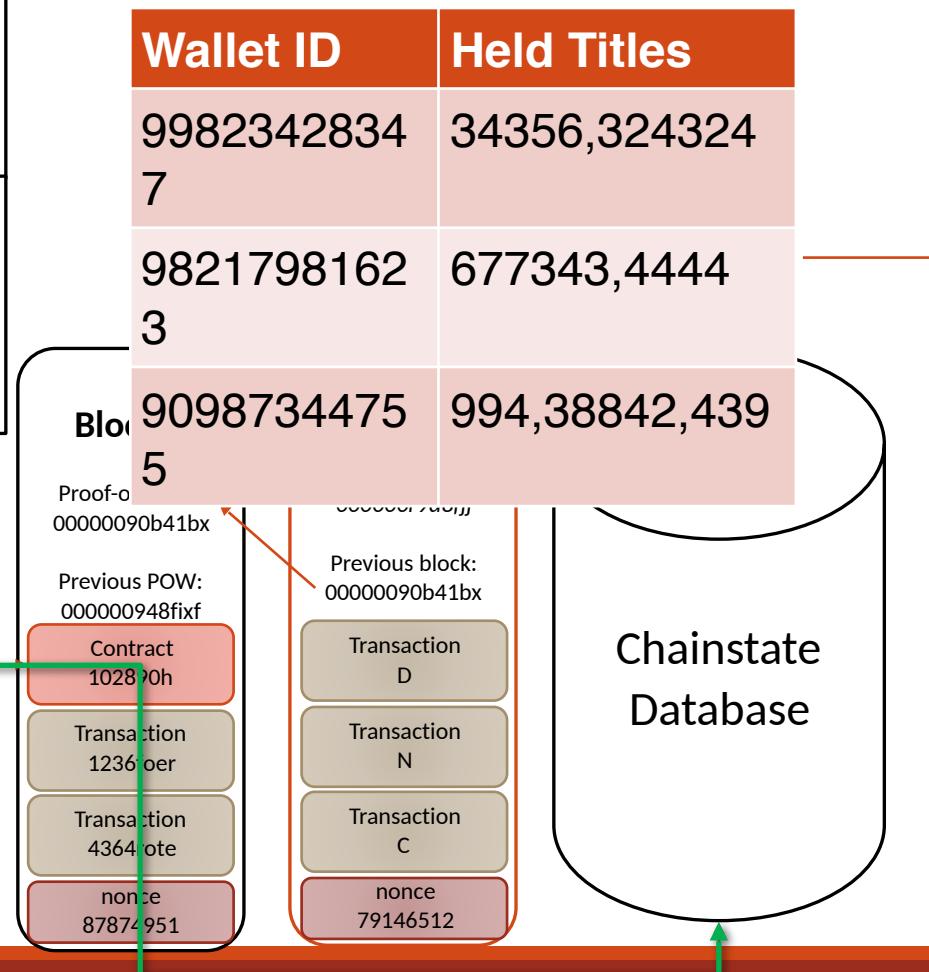
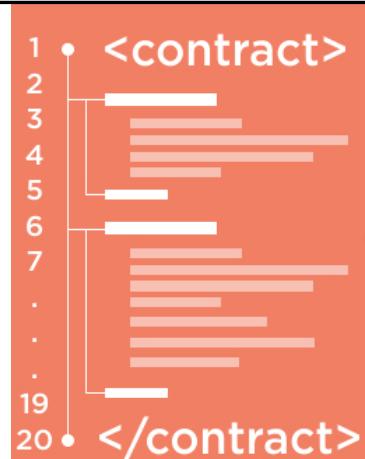
Smart Contracts

- Contracts contain *executable bytecode*
- Created with a blockchain tx
- Contracts have internal storage

Contracts execute when triggered by a transaction (or by another contract)

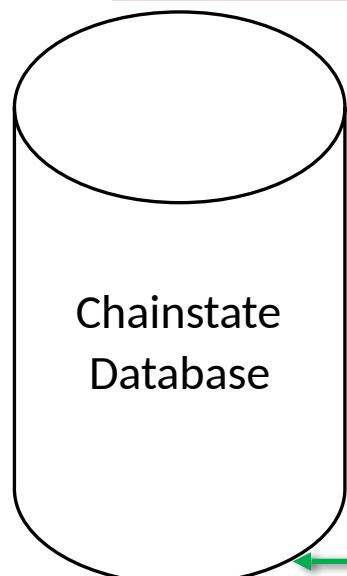
Execution time is limited by *gas*

Example: Land registry



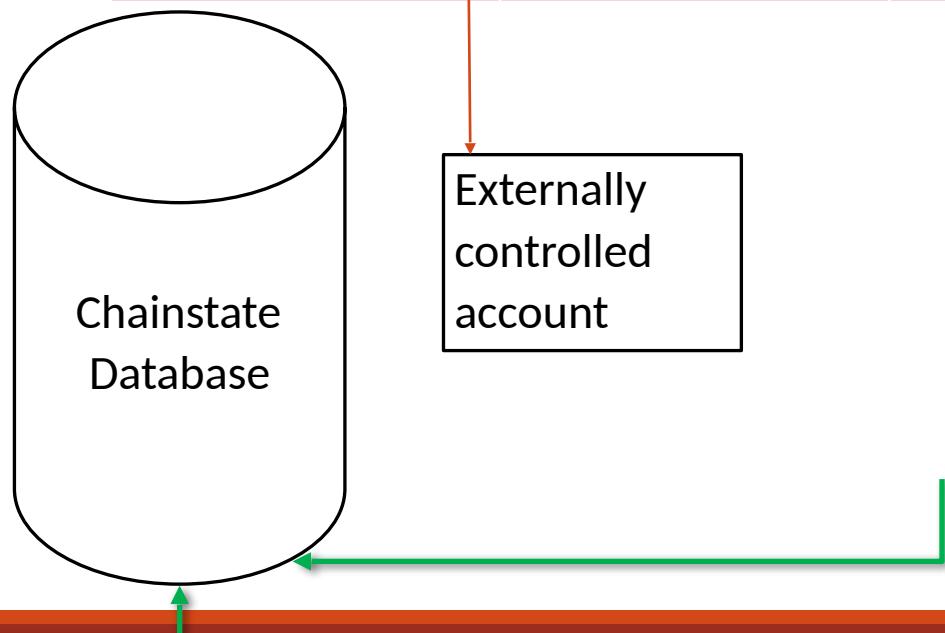
Account State (“World State”)

Wallet ID	Balance	Code Hash	Internal State
99823428347	45.12	-	99554HGJ
98217981623	1123.332	9ERU12T4	3453ADFG
90987344755	9.3444	0490CNDJ	132GJR4



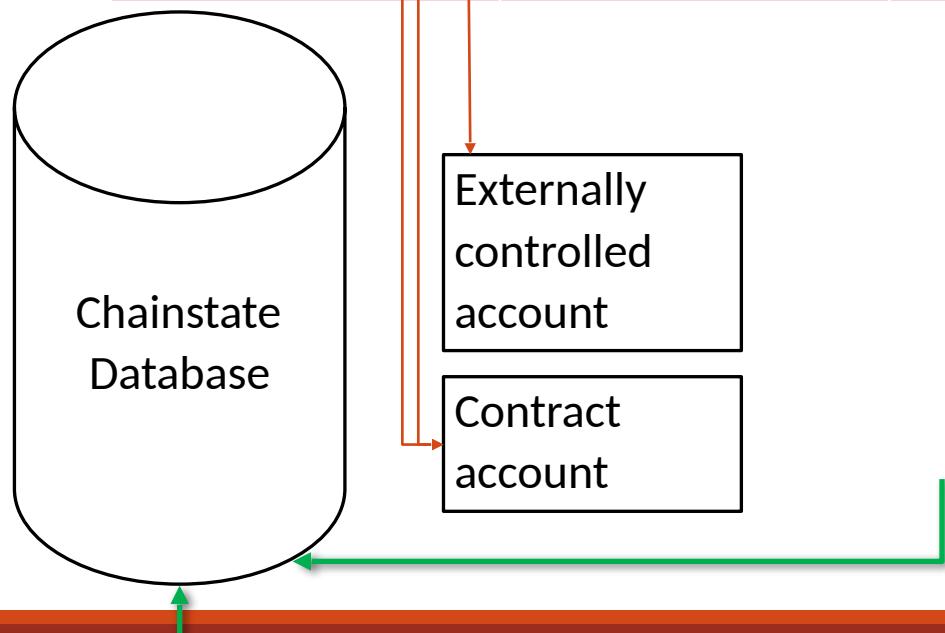
Account State (“World State”)

Wallet ID	Balance	Code Hash	Internal State
99823428347	45.12	-	99554HGJ
98217981623	1123.332	9ERU12T4	3453ADFG
90987344755	9.3444	0490CNDJ	132GJR4



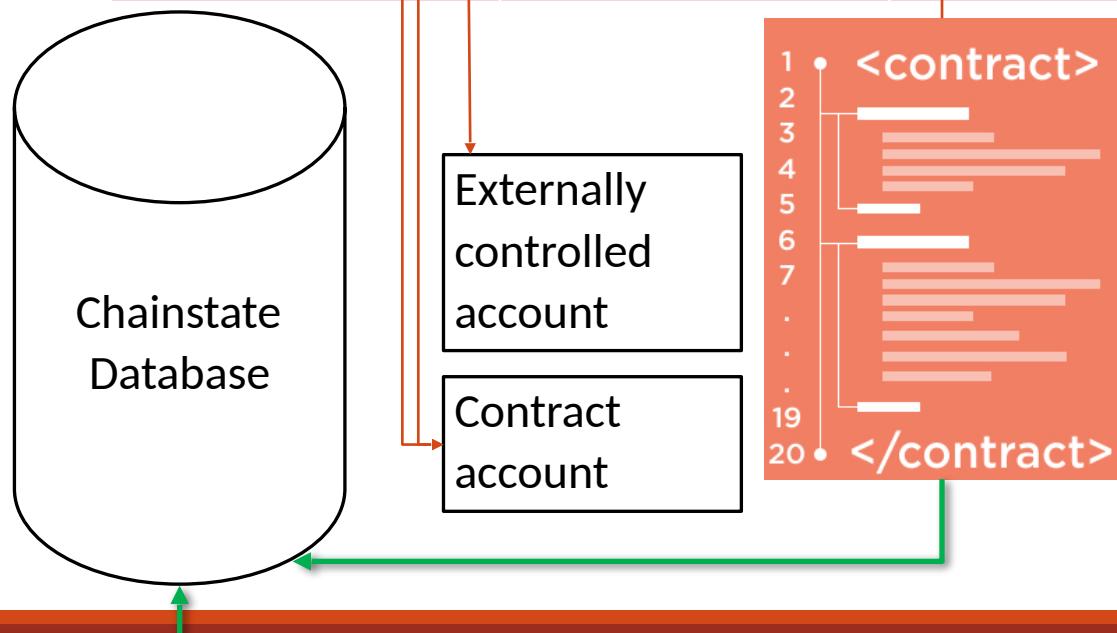
Account State (“World State”)

Wallet ID	Balance	Code Hash	Internal State
99823428347	45.12	-	99554HGJ
98217981623	1123.332	9ERU12T4	3453ADFG
90987344755	9.3444	0490CNDJ	132GJR4



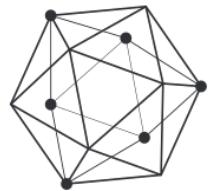
Account State (“World State”)

Wallet ID	Balance	Code Hash	Internal State
99823428347	45.12	-	99554HGJ
98217981623	1123.332	9ERU12T4	3453ADFG
90987344755	9.3444	0490CNDJ	132GJR4



Comparison with Bitcoin

	Bitcoin	Ethereum
Transactions	Transfer of bitcoins	<i>Contract creation, transfer of ether, contract calls, internal transactions</i>
Accounts	User wallets	Externally owned accounts, <i>contract accounts</i>
Transaction fees	Amount specified by sender	Gas calculated using sender's values
Block content	Transactions trie	Transactions, <i>State Root Hash, Receipts Root Hash</i>
Chainstate Database	World state: UTXOs for wallets	World state, <i>receipts, bytecodes for contracts</i>
Querying	Simple Payment Verification	Merkle proofs for <i>events, transactions, balance, etc.</i>



HYPERLEDGER

Managing entity: Hyperledger Consortium

- Major players: IBM, NEC, Intel, R3, ...

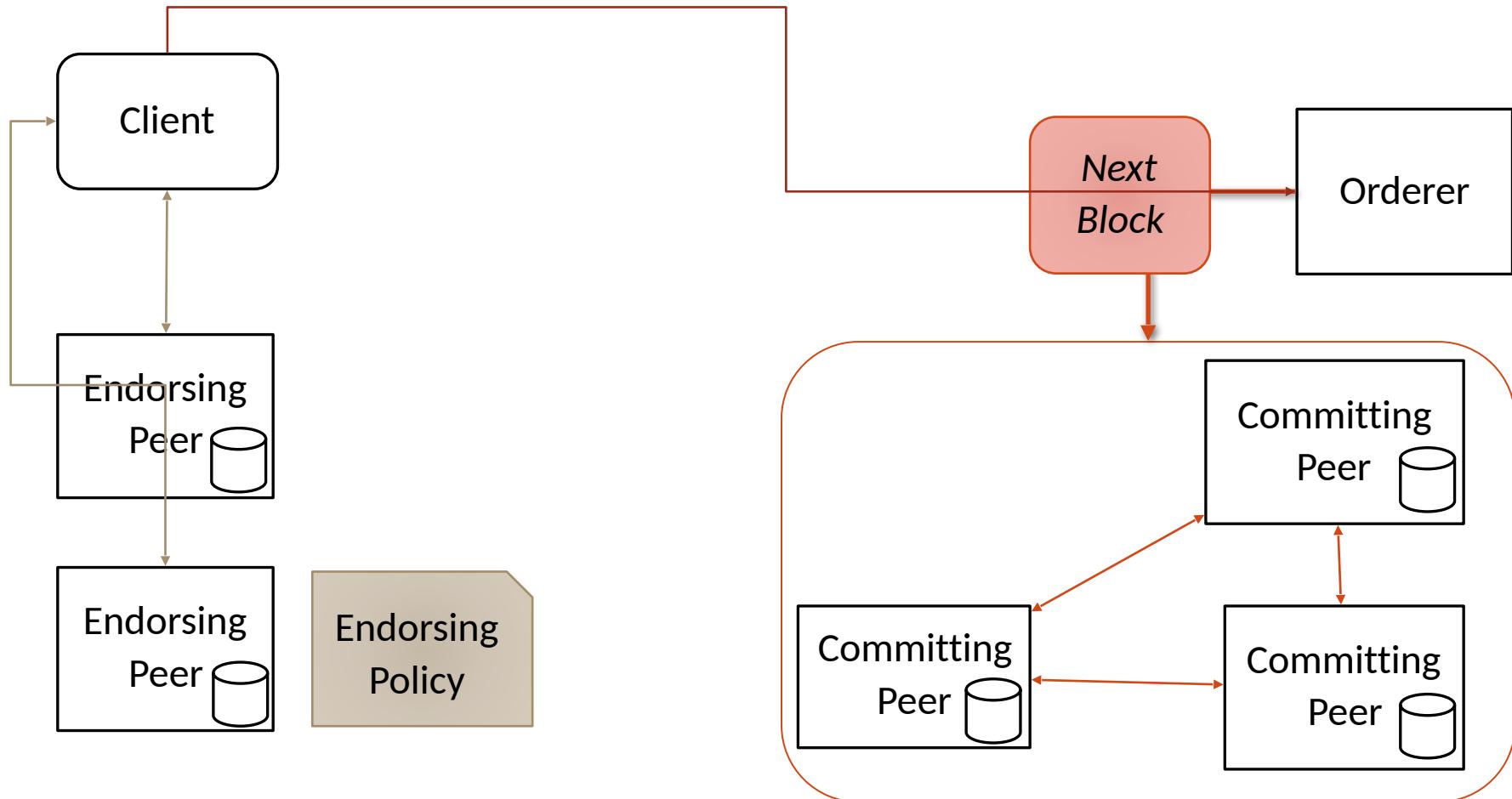
Focus: Enterprise blockchains

- Permissioned ledger (private/consortium network)
- Smart contracts, aka Chaincode
- Open-source
- World state on CouchDB, event listener

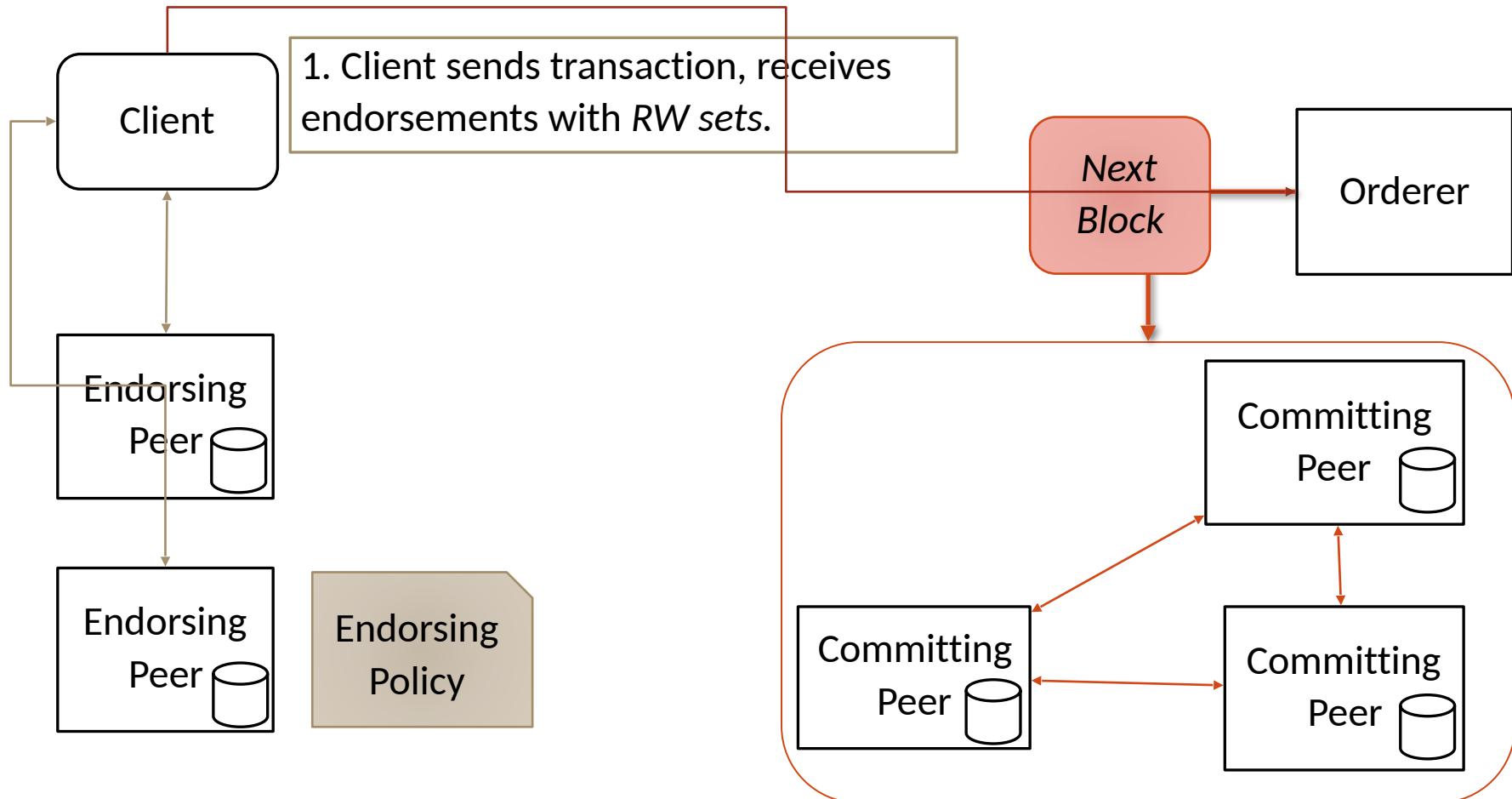
Projects

- Fabric: Execute-Order-Validate transaction processing
- Sawtooth: Proof-of-Elapsed-Time (using Intel SGX)
- Cello: Blockchain-as-a-Service framework

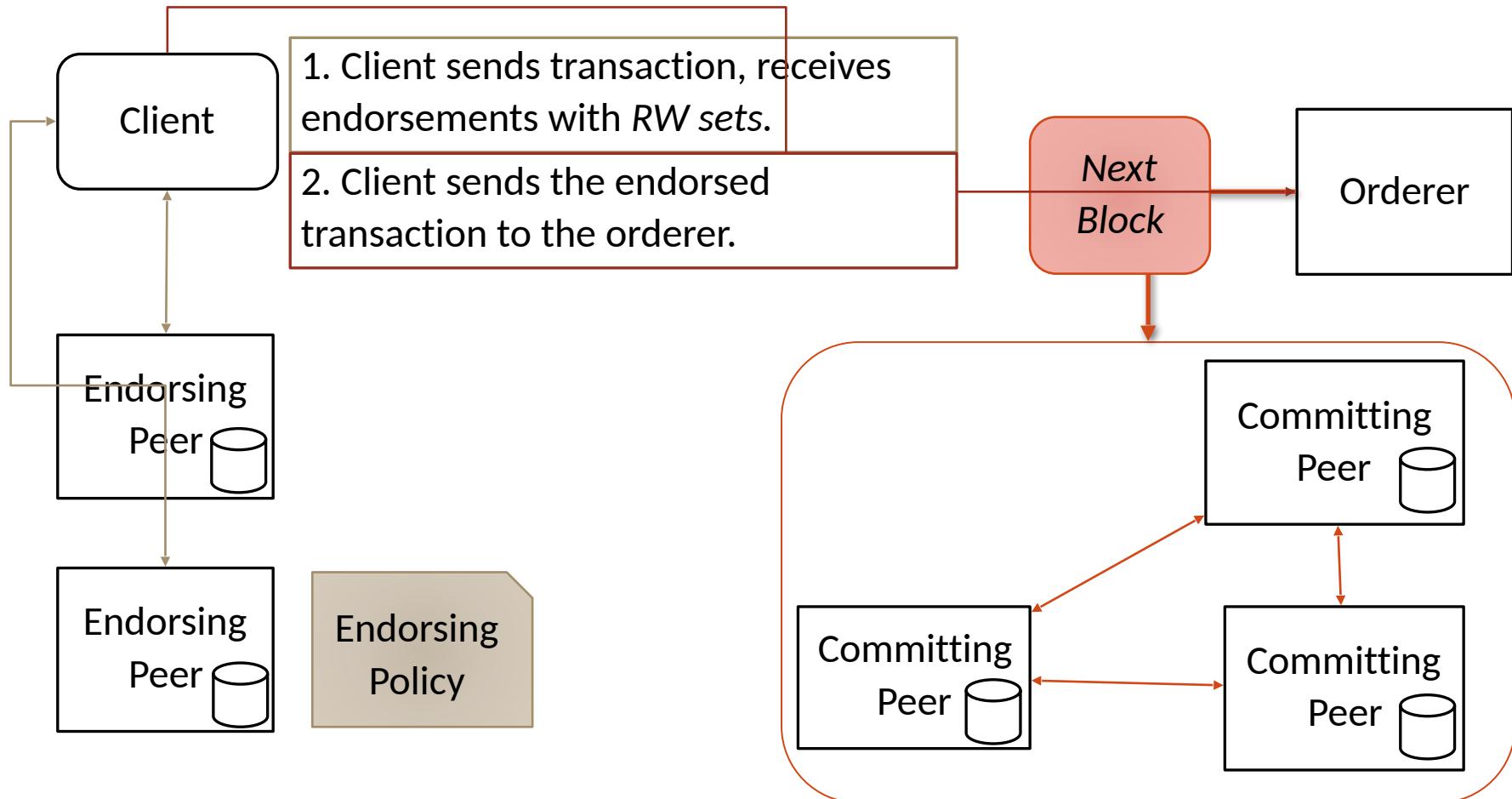
Transaction Processing Flow



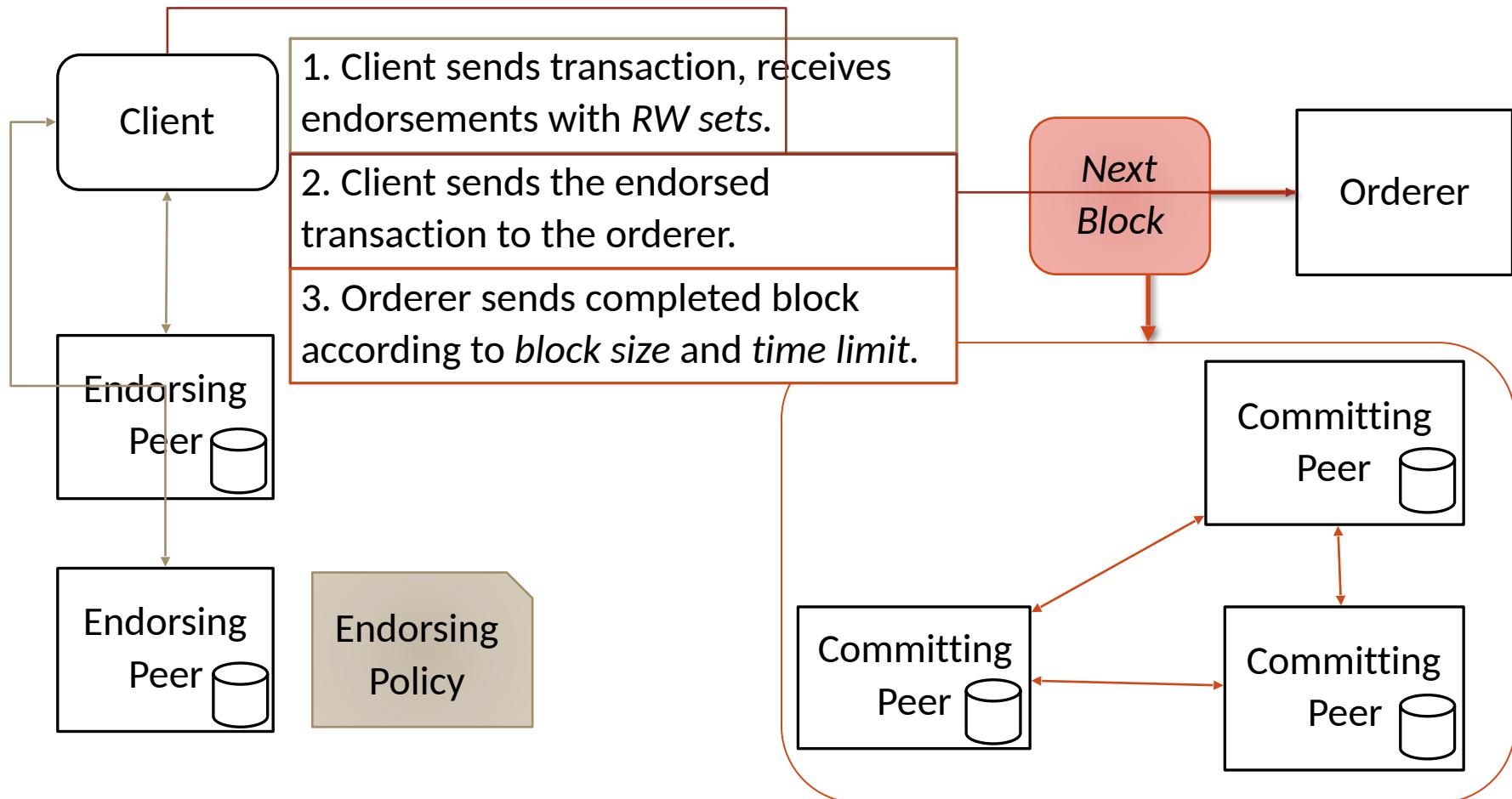
Transaction Processing Flow



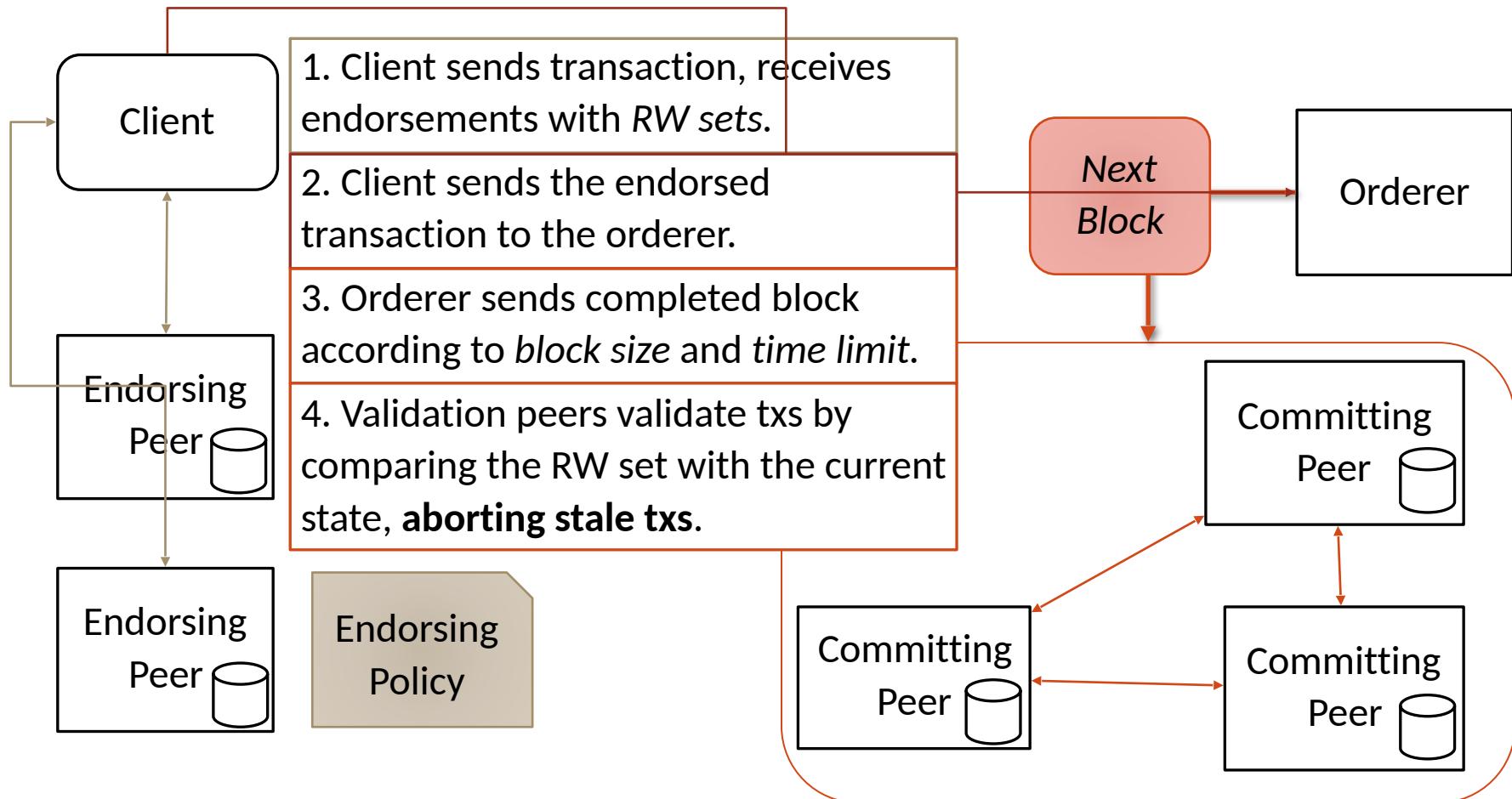
Transaction Processing Flow



Transaction Processing Flow



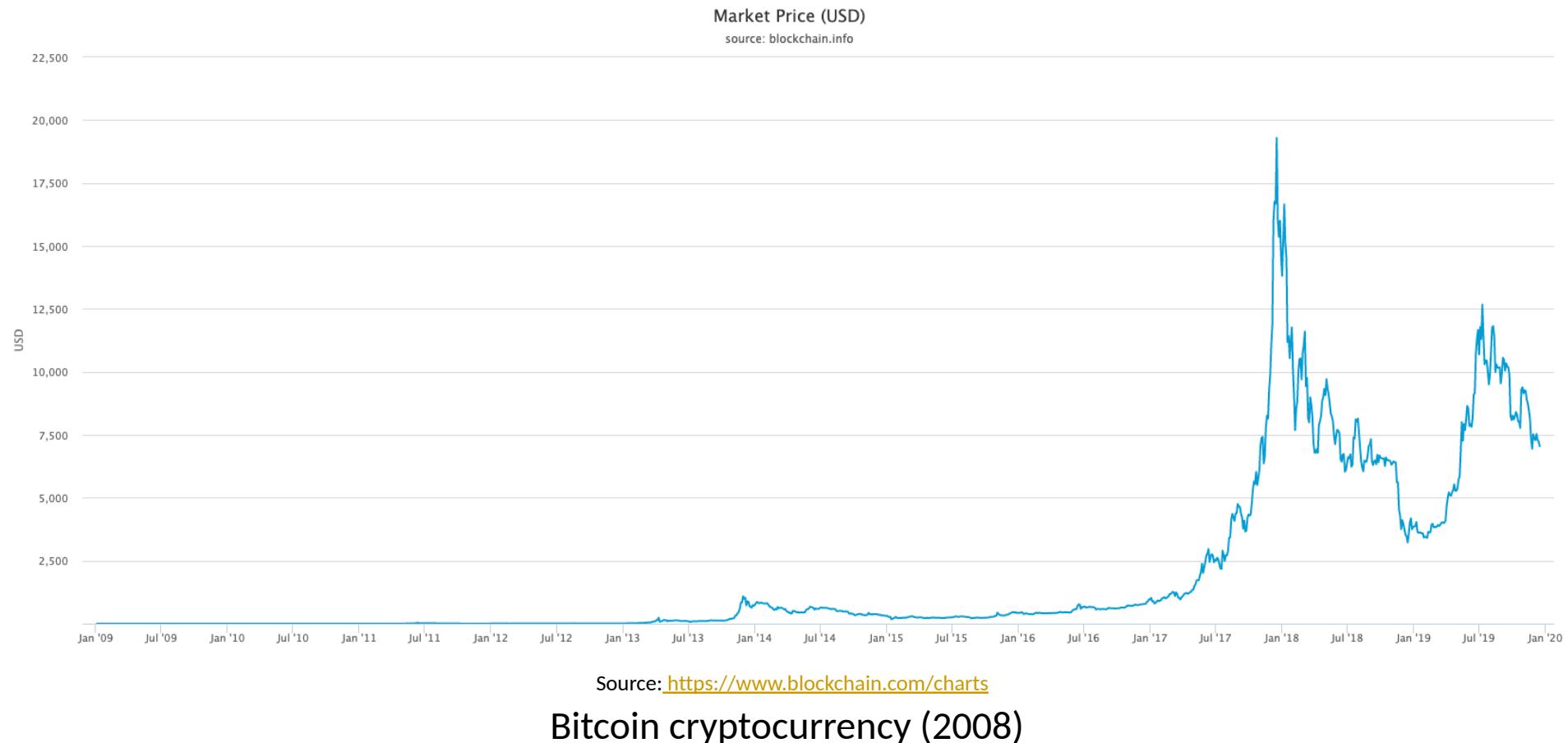
Transaction Processing Flow



Blockchain Applications

1.0, 2.0, 3.0 GENERATIONS
IMPACT

Blockchain 1.0: Currency



Blockchain 2.0: Decentralized Apps (DApps)



DApps are applications built on blockchain platforms using smart contracts (e.g. Ethereum)

Blockchain 2.0: Decentralized Apps (DApps)



DApps are applications built on blockchain platforms using smart contracts (e.g. Ethereum)



Blockchain 2.0: Decentralized Apps (DApps)



DApps are applications built on blockchain platforms using smart contracts (e.g. Ethereum)



Blockchain 2.0: Decentralized Apps (DApps)



DApps are applications built on blockchain platforms using smart contracts (e.g. Ethereum)



Blockchain 2.0: Decentralized Apps (DApps)

Blockchain 2.0: Decentralized Apps (DApps)



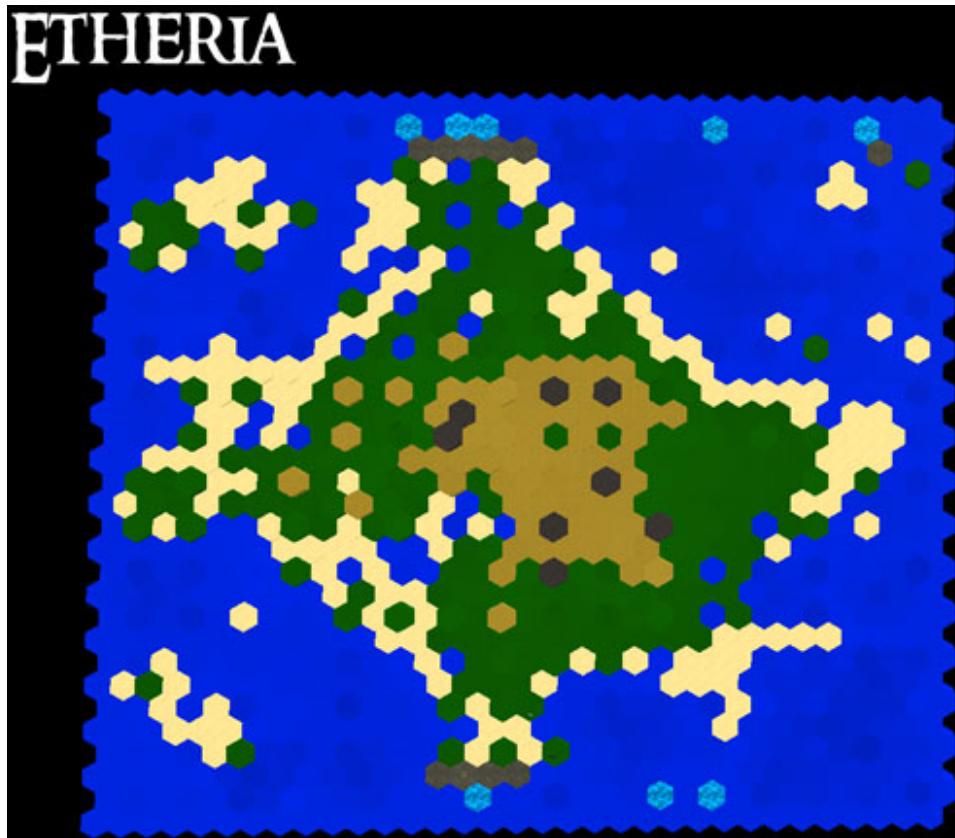
GNOSIS

Forecast market (e.g. betting, insurance)

Blockchain 2.0: Decentralized Apps (DApps)



Forecast market (e.g. betting, insurance)



Blockchain 3.0: Pervasive Apps

Applications involve entire industries, **public sector**, and IoT.

Blockchain 3.0: Pervasive Apps



everledger

Diamonds Provenance

Applications involve entire industries, **public sector**, and IoT.

Blockchain 3.0: Pervasive Apps



everledger

Diamonds Provenance



FACTOM

Land Registry in Honduras

Applications involve entire industries, **public sector**, and IoT.

Blockchain 3.0: Pervasive Apps



everledger

Diamonds Provenance

Applications involve entire industries, **public sector**, and IoT.



FACTOM

Land Registry in Honduras



BlockchainHealth

Electronic Health Records

Blockchain 3.0: Pervasive Apps



everledger

Diamonds Provenance



FACTOM

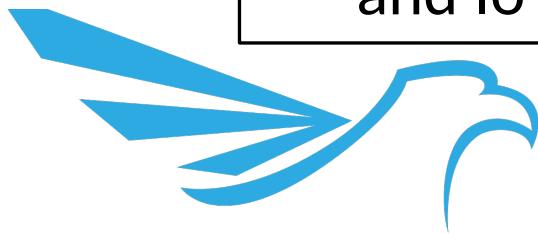
Land Registry in Honduras



BlockchainHealth

Electronic Health Records

Applications involve entire industries, **public sector**, and IoT.



VOTEWATCHER

Transparent Voting System

Proof-of-Stake

PROOF-OF-WORK



THE PROBABILITY OF MINING A BLOCK IS DEPENDENT ON HOW MUCH WORK IS DONE BY THE MINER



PAYOUTS BECOMES SMALLER AND SMALLER FOR BITCOIN MINERS. THERE IS LESS INCENTIVE TO AVOID A 51% ATTACK



POW SYSTEMS HAVE POWERFUL MINING COMMUNITIES - BUT TEND TO BECOME CENTRALIZED OVER TIME

OR

PROOF-OF-STAKE



PERSON CAN "MINE" DEPENDING ON HOW MANY COINS THEY HOLD



THE POS SYSTEMS MAKES ANY 51% ATTACK MORE EXPENSIVE



POS SYSTEMS ARE MORE DECENTRALIZED - BUT MUST WORK HARD TO BUILD COMMUNITIES AROUND THEIR COINS

Downsides to Proof-of-Work

- Requires enormous time and computational effort

- Tragedy of commons
 - As the reward decreases, the number of miners go down (less profitable)
 - This reduces the difficulty of the puzzle
 - Makes it easier to inject a large amount of compute power and execute the 51% attack

Proof-of-Stake

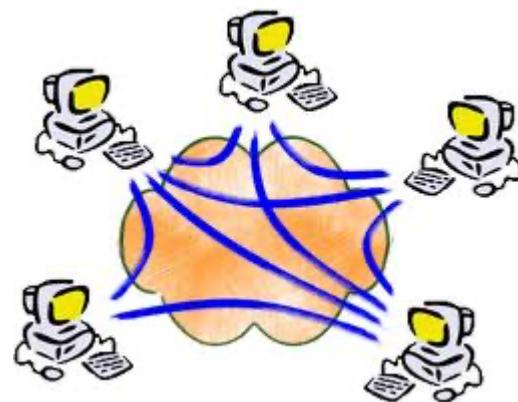
- Instead of mining with computing power, mine with virtual currency
- Essentially, each coin a miner has is a lottery ticket.
- A random generator draws the winning coin; the probability of a winner is proportional to the number of coins it has.
- 51% attacks are only possible if a miner owns 51% of the total currency in the system
- Can be used to reduce the difficulty of the work in PoW
- First system: PeerCoin (2012)
- Ethereum to adopt later

Virtualization, Cloud and Serverless Computing

Pezhman Nasirifard

Hans-Arno Jacobsen

Application & Middleware Systems Research Group



Outline

- Virtualization
 - Hardware Virtualization
 - Containers
- Cloud Computing
 - PaaS, IaaS
- Serverless Computing

Final Exam Review (Feb 5th):
<https://forms.gle/JbyYk4S7XfUz2vT99>

VIRTUALIZATION

Physical Servers

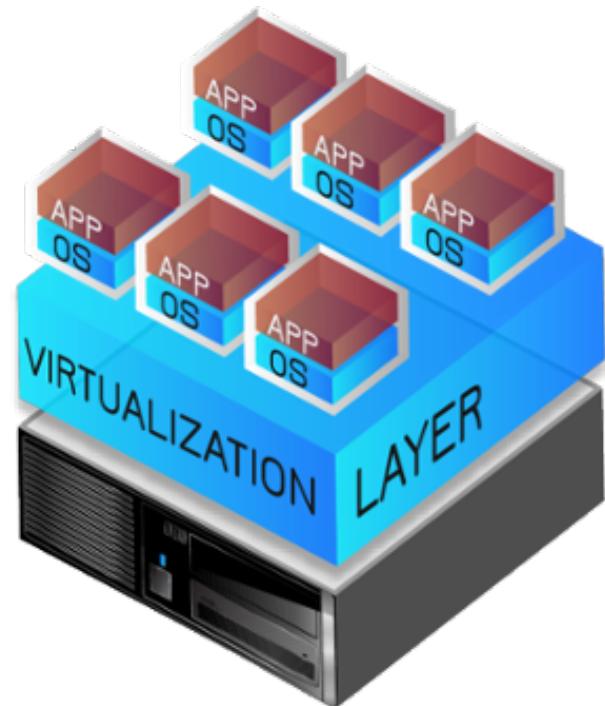
- One OS/Application per server
 - High up-front costs
 - High ongoing costs
 - Most of the server resources are wasted



Source: Wikipedia

What is Virtualization?

- The OS is abstracted away from hardware
- OS no longer has to be bound to the Server or PC that runs on
- Multiple OS and Applications on one Server



Source: stepupitserices

Server Virtualization

- Most common form of virtualization
- Multiple OS runs on the virtual layer, a.k.a Hypervisor

Virtual Machine
(Each guest OS running on the host)

Virtual Host
(Physical Server with virtualization layer)

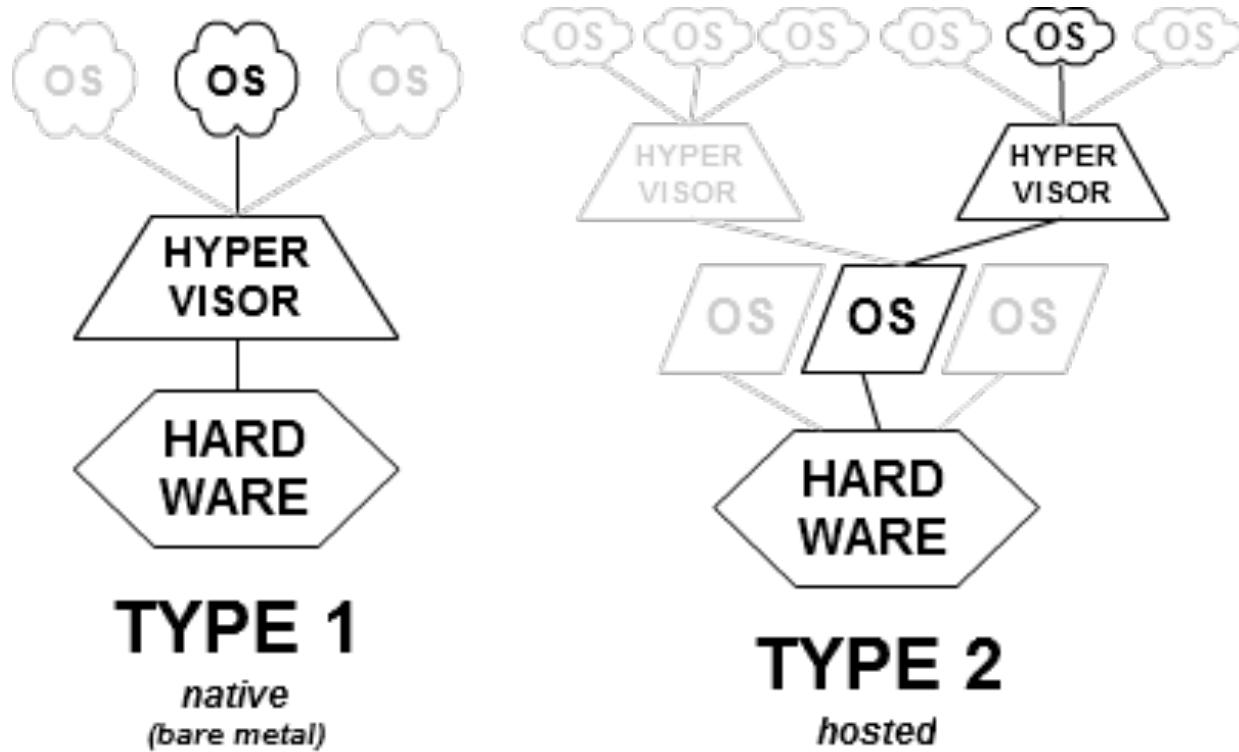
Hypervisor

- Hypervisor
 - Creates the virtualization layer that makes server virtualization possible
 - Contains the Virtual Machine Monitor (VMM)
- Example of Hypervisors
 - KVM
 - Oracle VirtualBox

Type 1 vs. Type 2 Hypervisor

- Type 1 Hypervisor is loaded directly on the hardware (a.k.a native or bare-metal)
 - Microsoft Hyper-V
 - VMware ESX/ESXi
 - KVM
- Type 2 is loaded on an OS running on the hardware (a.k.a hosted hypervisor)
 - Oracle VirtualBox
 - VMware Workstation

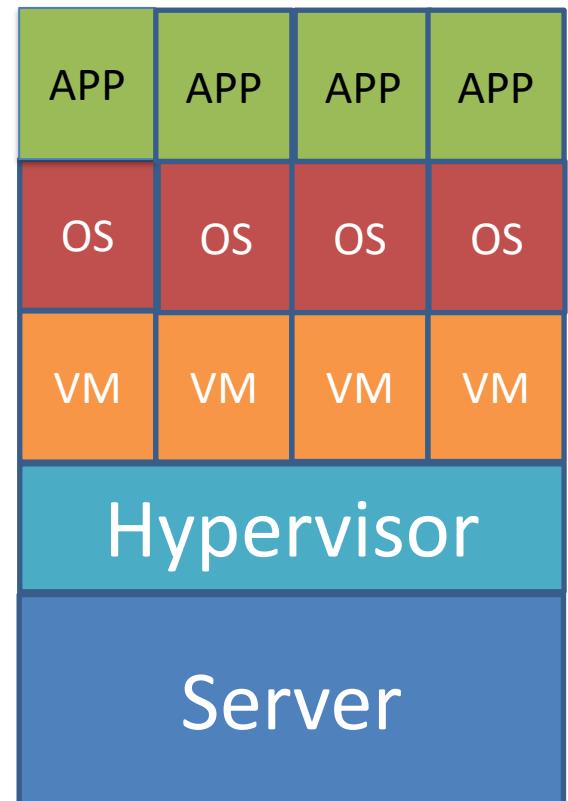
Type 1 vs. Type 2 Hypervisor



Source: Wikipedia

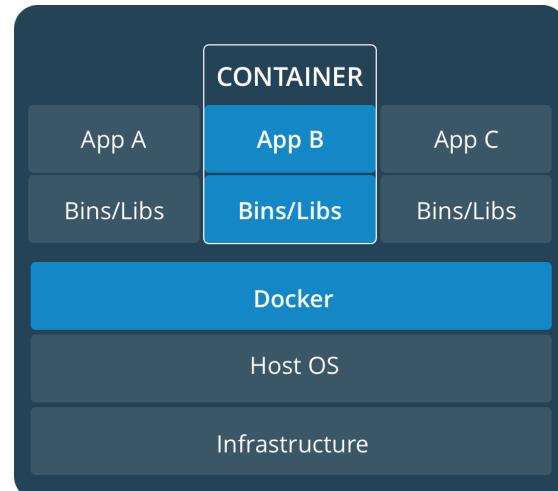
Disadvantages of Hypervisors

- The Server's resources are shared among VMs
- Execution of OS consumes resources (CPU/RAM/Disk) which is independent of the app
- OS can be expensive and requires administration



Containers

- Containers help with execution of apps
- Containers provide isolated area (processes) on top of OS areas where apps can run
 - Docker, most well-known Container Management
- OS does not provide much isolation for apps by default, which is not secure enough and leads conflicts
 - Virtual Memory Isolation
 - Privileges of the process owner
- Less overhead than VMs



Source: Docker

Isolation Evolution (cont.)

- 1979 – Change root
- 1999
 - Linux Kernel 2.2 Adds Capabilities
 - Security-Enhanced Linux
 - AppArmor
- 2000 – FreeBSD Jails
- 2001 – Linux Vserver
- 2002 – Mount Namespaces on Kernel 2.4.19

Namespaces

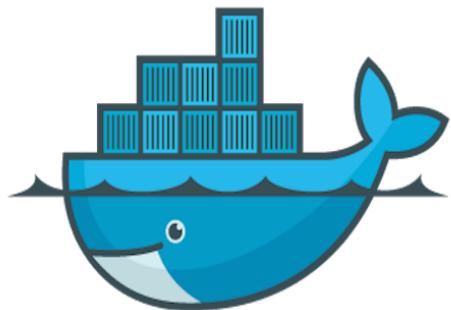
- A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of global resources.
- Changes to the resources are hidden from processes in other namespaces

Namespaces

- Current Linux Namespaces
 - Cgroup – isolate resource usage for a group of processes
 - IPC – isolate process communication
 - Network – isolate network resources
 - Mount – isolate mount points
 - PID – isolated list of process Ids
 - User – isolating the list of users
 - UTS – isolate hostname of computer
- Namespaces trick a process into seeing an entirely separated machine

Managing Containers

- Containers management tools simplify working with namespaces



docker



Isolation Evolution (cont.)

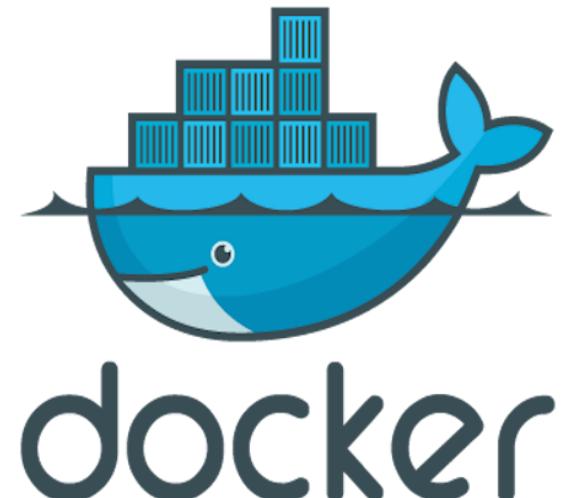
- 2004 – Solaris Containers
- 2005 – OpenVZ
- 2006
 - Process Containers (cgroups)
 - UTC & IPC Namespaces

Isolation Evolution

- 2008
 - PID, Network & User Namespaces
 - LXC
- 2013 – Docker
- 2014 – Rocket
- 2015 – OCI – The Open Container Initiative
- 2016 – Windows Containers

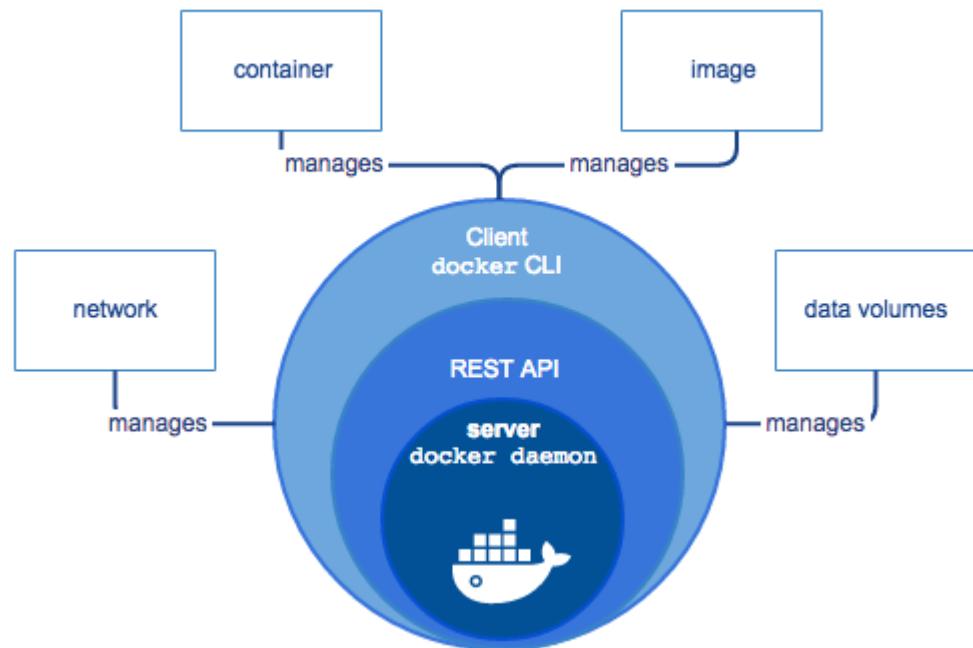
What is Docker?

- Docker Inc. the main force behind Docker project
- Docker provides a set of open source tools to build, distribute and deploy applications
- Docker Engine the primary tool for managing the containers
- Docker Hub the public official containers registry



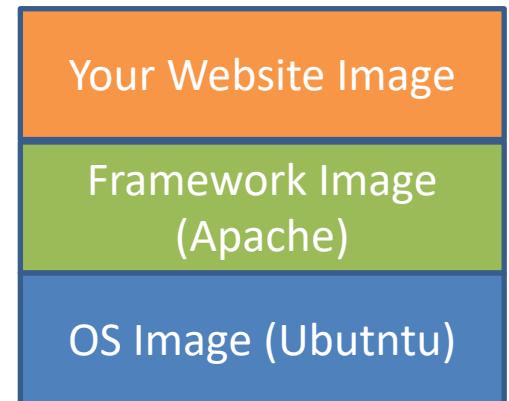
Docker Engine

- Docker engine is a client-server application for managing and executing Docker objects
 - The Docker Daemon
 - The Docker Client



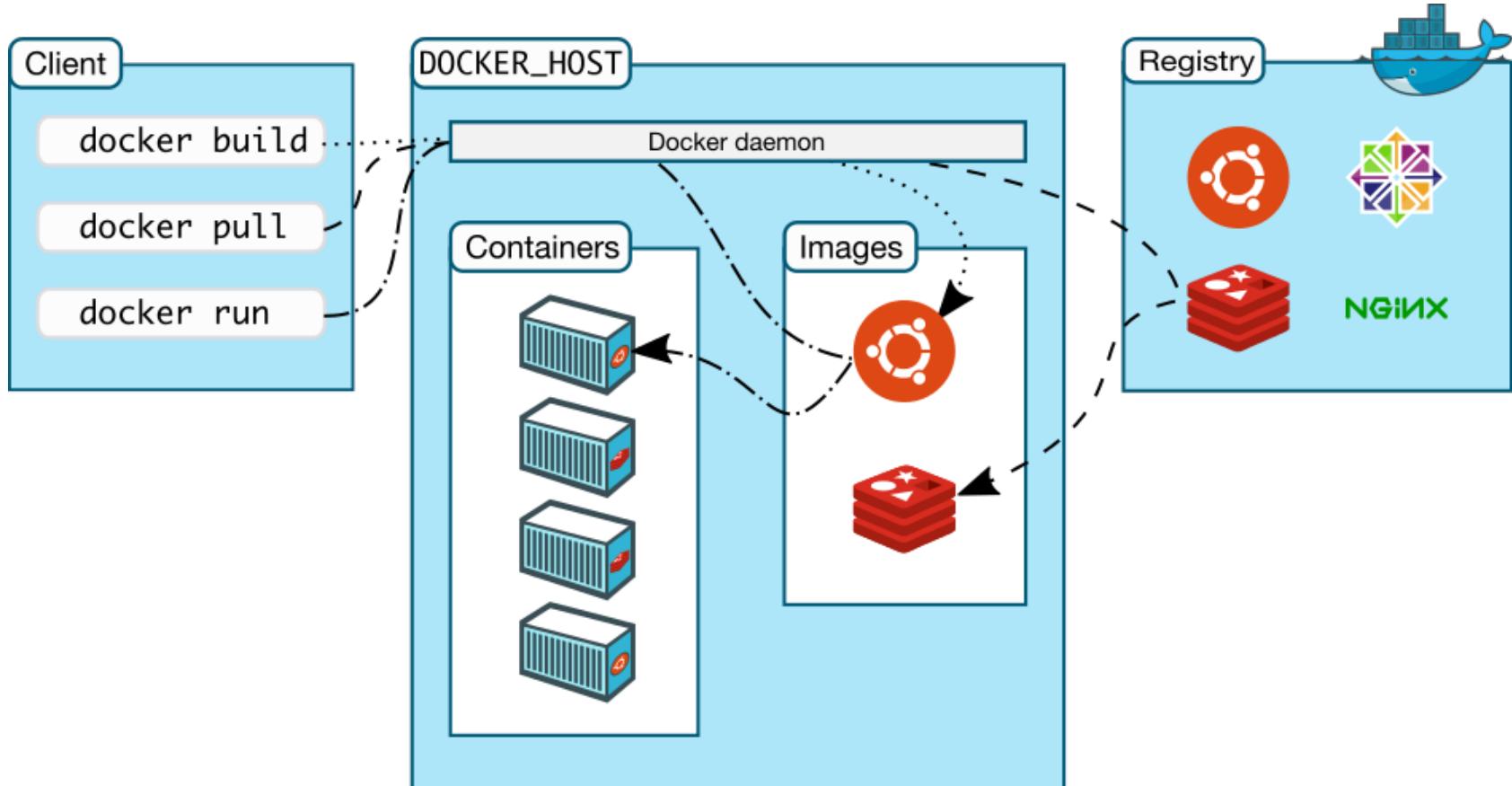
Docker Images

- An image is a read-only template with instructions for creating a Docker container
- An Image has a layered structure based on Union File Systems
- Each layer depends on the underlying layers
- Use Dockerfile to create images
- Public images are on Docker Hub



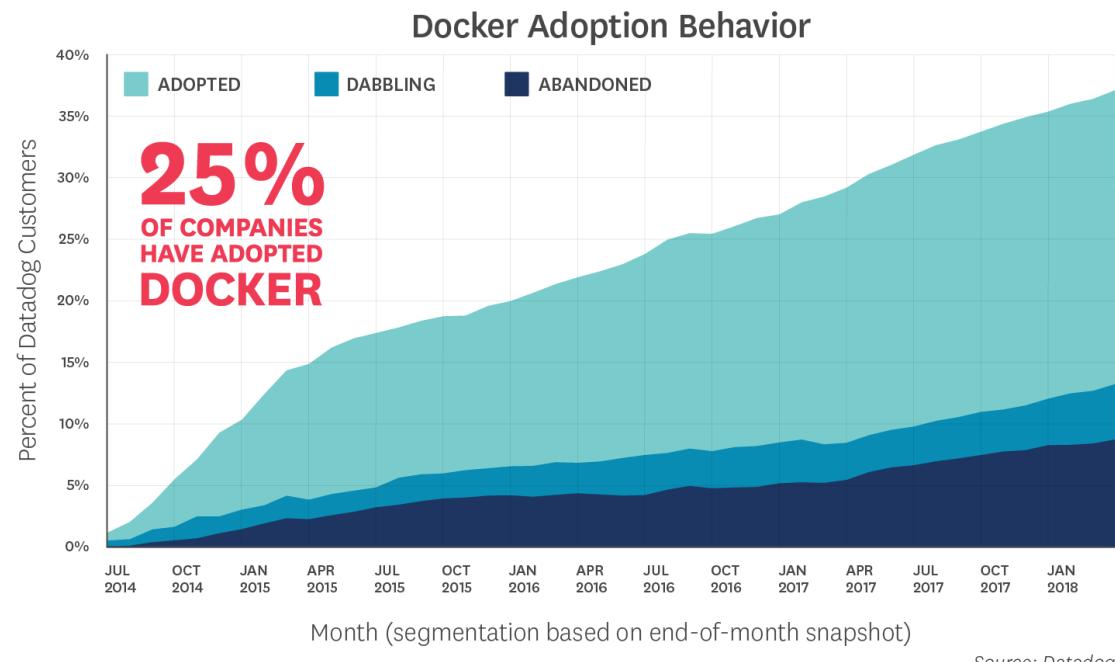
Running Containers

- A container is a runnable instance of an image



Why use Docker?

- Fast, simplified, scalable application development and shipment
- Lightweight footprint and minimal overhead
- Cross-platform portability



Overview Virtualization

- Server Virtualization most popular form of virtualization
- Type 1 and Type 2 Hypervisors
- Containers provide isolated area on top of OS for running applications
- Docker is the primary player in containers world

CLOUD COMPUTING

What is Cloud?

- A computing service you traditionally did local, now the service is performed remotely (off-premises)
- Cloud computing is an approach to computing that leverages the efficient pooling of an on-demand, self-managed, virtual infrastructure



Advantages of Cloud Computing

- Fast and efficient access to resources that clients actually need
- Pay for what you use
- No initial capital expenditure
- Less need for a big administrator team
- Self-maintenance and fault tolerance resources

Disadvantages of Cloud Computing

- Clients need to trust the cloud providers
- Possible limited access to your data
- Possible conflicts with government regulations and restrictions
- Potential data loss
- Locked in within the cloud provider's specification
- You may encounter unknown costs (check SLA)
- Know who you are dealing with!

Evolution of Cloud Computing

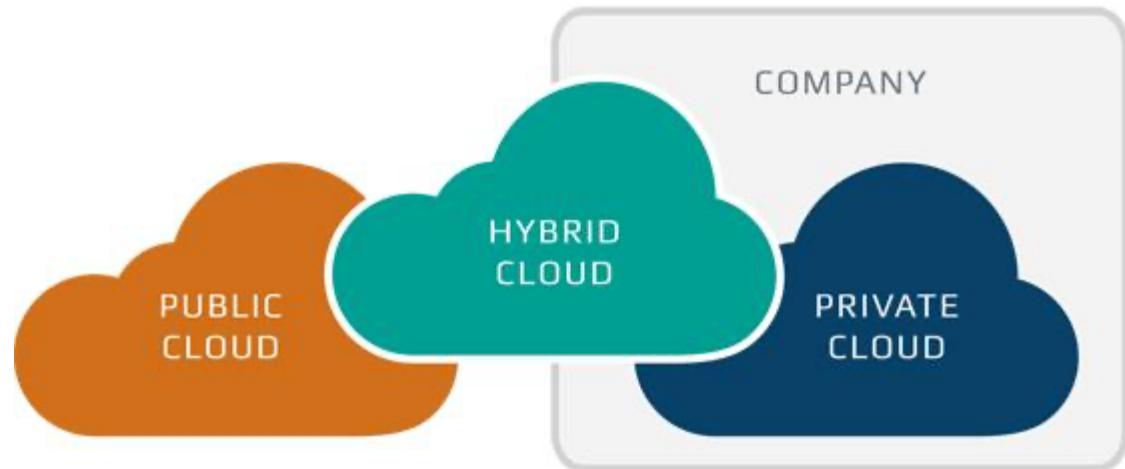
- The 1950s – Mainframe Computing
- 1960 – 1990
 - Internet, VMs, VPNs
- 1999 – Salesforce, the first SaaS
- 2006 – Amazon Web Services
- 2008 – Microsoft Azure
- 2013 – Google Compute Engine
- 2014 – IBM Bluemix

Cloud Comes in Many Shapes!

- Infrastructure-as-a-Service (IaaS)
- Platform-as-a-Service (PaaS)
- Software-as-a-Service (SaaS)
- Function-as-a-Service (FaaS)
- Database-as-a-Service (DBaaS)
- Everything-as-a-Service (*aaS, XaaS)

Infrastructure as a Service (IaaS)

- VMs/Containers offered as a service
 - Private Clouds run on your premises (e.g. OpenStack)
 - Public Clouds runs over the internet (e.g. AWS)
 - Hybrid Cloud is not a real cloud. It is the cooperation between public and private cloud



Virtualization vs. Private Cloud

- Virtualization is required for cloud computing
- Virtualization provides scalability, fault-tolerance, high availability and load balancing
- Private clouds are built on top of virtualizations
- Private clouds provide abstraction of resources, secure multi-tenancy, better separation of concerns

Platform-as-a-Service (PaaS)

- Clients do not need to manage storage, network, OS, database, etc.
- Cloud provider install all the dependencies client need
 - Cloud Foundry
 - Google App Engine
 - AWS Elastic Beanstalk
 - MS Azure App Service
 - Heroku

IaaS vs PaaS

- IaaS is simply a VM with some OS installed. Clients are required to install all the packages they need
- PaaS builds upon IaaS. Clients have access to the previously installed and configured resources
- PaaS is suitable solution for fast and easy application design and development

Storage-as-a-Service (SaaS)

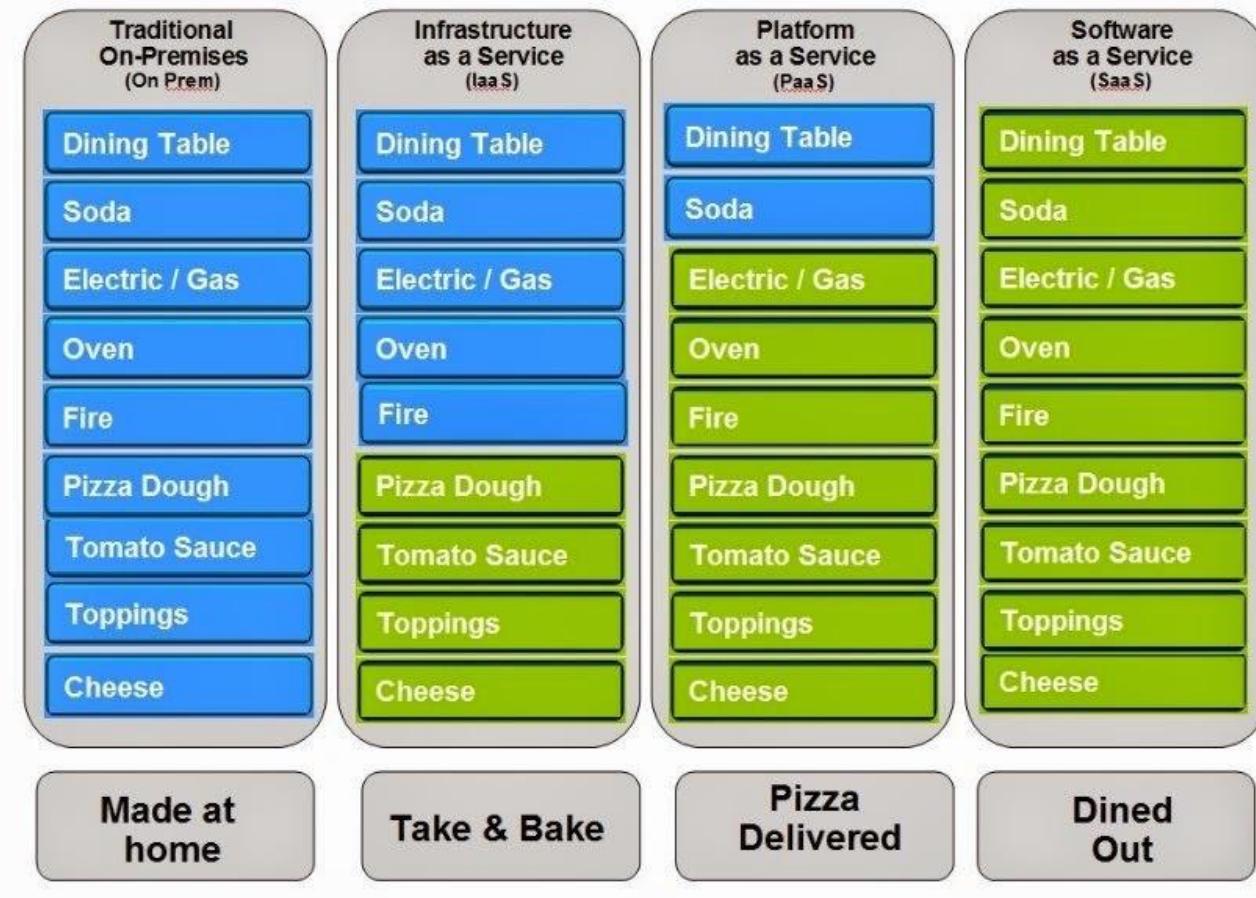
- Storing files and objects in the cloud
 - Auto-replication and backup
 - Universal availability
 - Dropbox, Microsoft OneDrive, Google Drive
 - AWS Simple Storage Service (S3)
 - Microsoft Azure Blob Storage

Software-as-a-Service (SaaS)

- No hardware or software that clients need to install, configure and maintain
- In many cases more affordable than similar options
- Clients have fast access to most recent patches and features
 - Google Docs
 - Salesforce
 - Gmail

Pizza-as-a-Service

Pizza as a Service



Source: Albert Barron

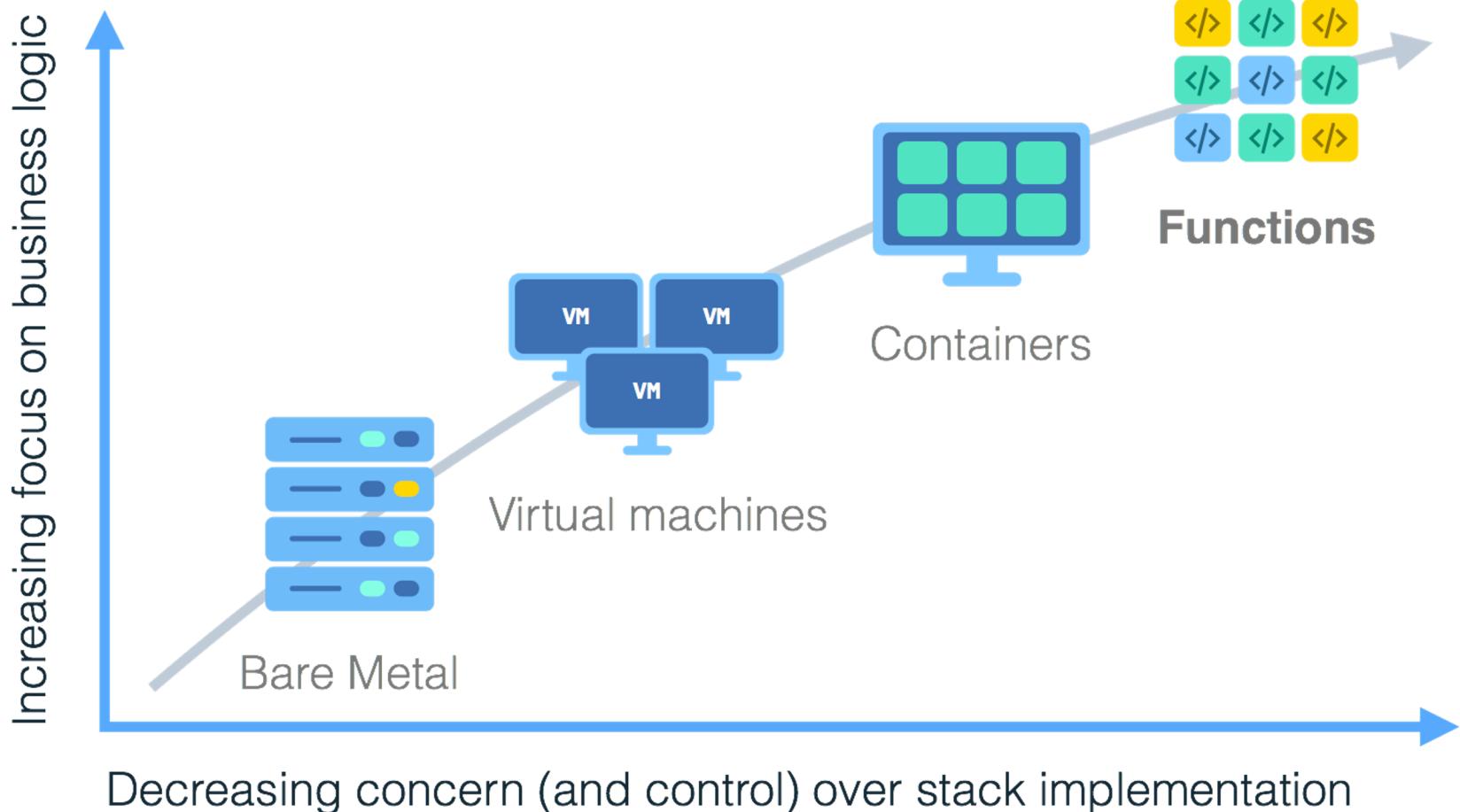
Overview Cloud Computing

- Cloud computing is using the managed remote computational resources over the network
- IaaS, PaaS, SaaS, XaaS

Thanks to Paul Castro, Vatche Ishakian, Vinod Muthusamy and
Aleksander Slominski @IBM Research

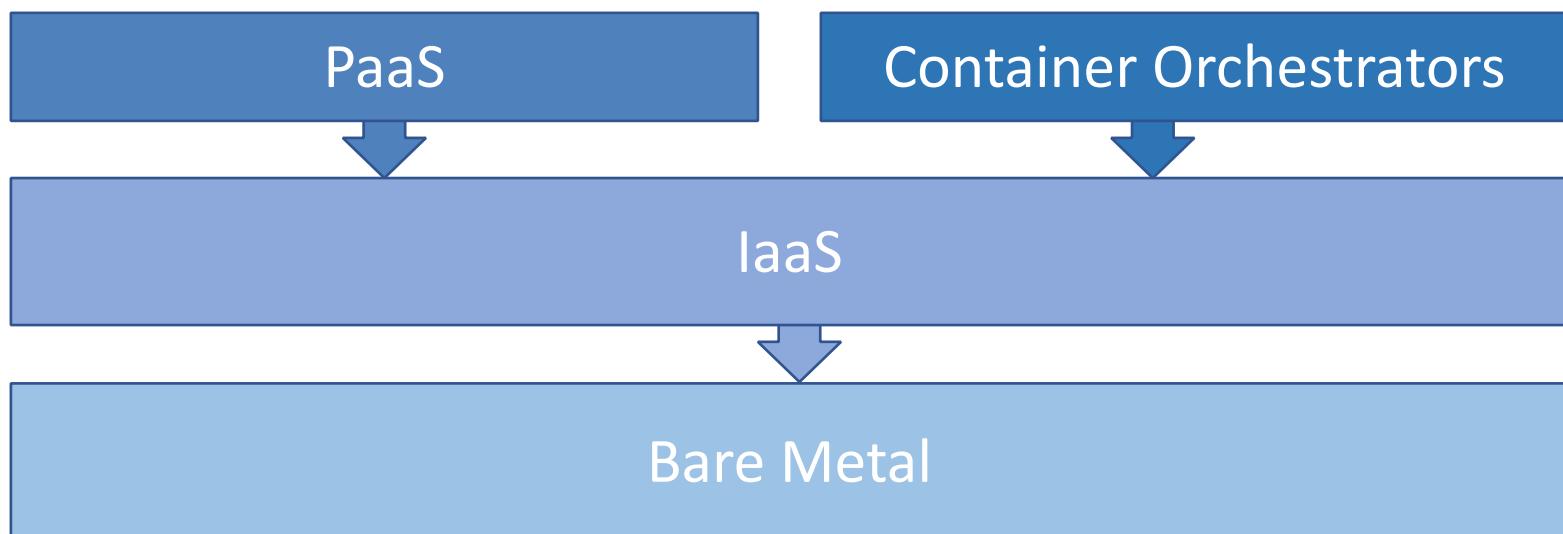
SERVERLESS COMPUTING

Cloud Computing Evolution

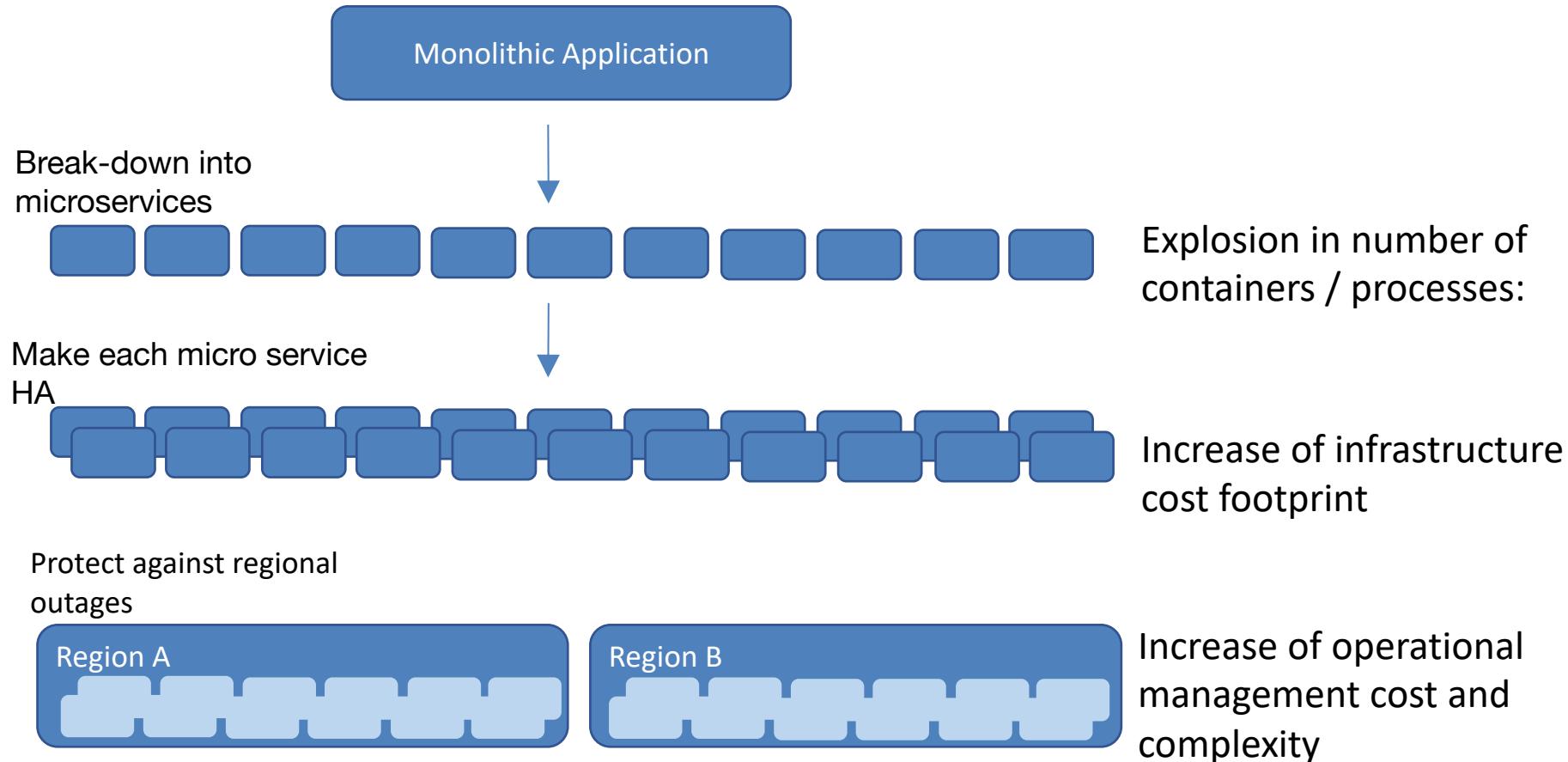


Evolution of Serverless

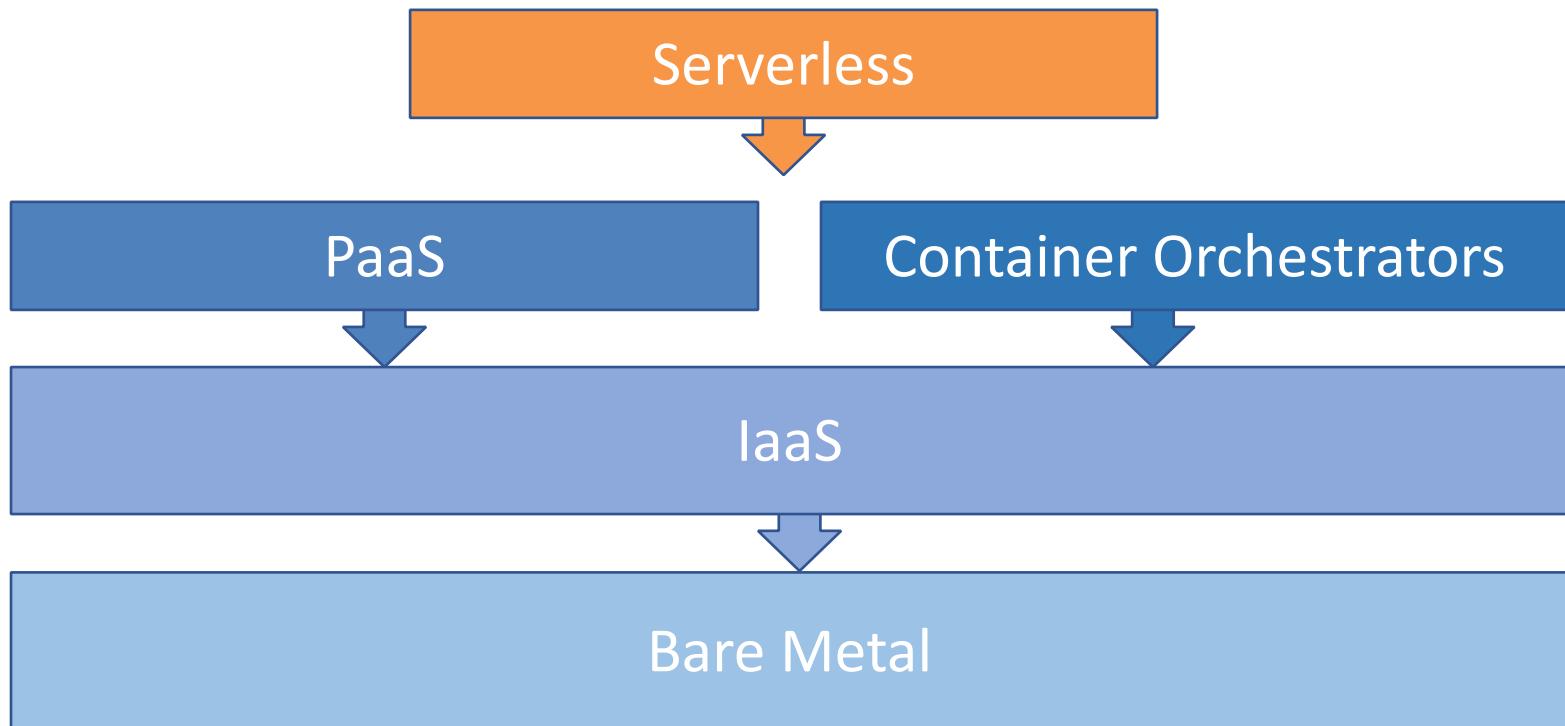
- We have the cloud, Why Serverless computing?



Evolution of Serverless



Serverless Position



What is Serverless?

- A cloud-native platform for short-running, stateless computation, and event-driven applications which scales up and down instantly and automatically and charges for actual usage at millisecond granularity
 - Auto-scalability and maintenance
 - Pay for what you use
- Serverless does not mean no servers, means worry-less servers
- Also known as Function-as-a-Service (FaaS)

Why is Serverless?

- Making application development and execution faster, cheaper and easier

	On-prem	VMs	Containers	Serverless
Time to provision	Weeks-months	Minutes	Seconds-Minutes	Milliseconds
Utilization	Low	High	Higher	Highest
Charging granularity	CapEx	Hours	Minutes	Blocks of milliseconds

What is Serverless Good For?

- Serverless is a good solution for short-running stateless event-driven operations
 - Microservice
 - Mobile Backends
 - Bots, ML Inferencing
 - IoT
 - Modest Stream Processing
 - Service Integration

What is Serverless NOT Good For?

- Serverless is not good for long-running stateful computationally heavy operations
 - Databases
 - Deep Learning Training
 - Heavy-Duty Stream Analytics
 - Spark/Hadoop Analytics
 - Video Streaming
 - Numerical Simulations

Serverless Platforms

- All major cloud providers offer serverless platforms



OpenLambda



Azure
Functions



Red-
Hat



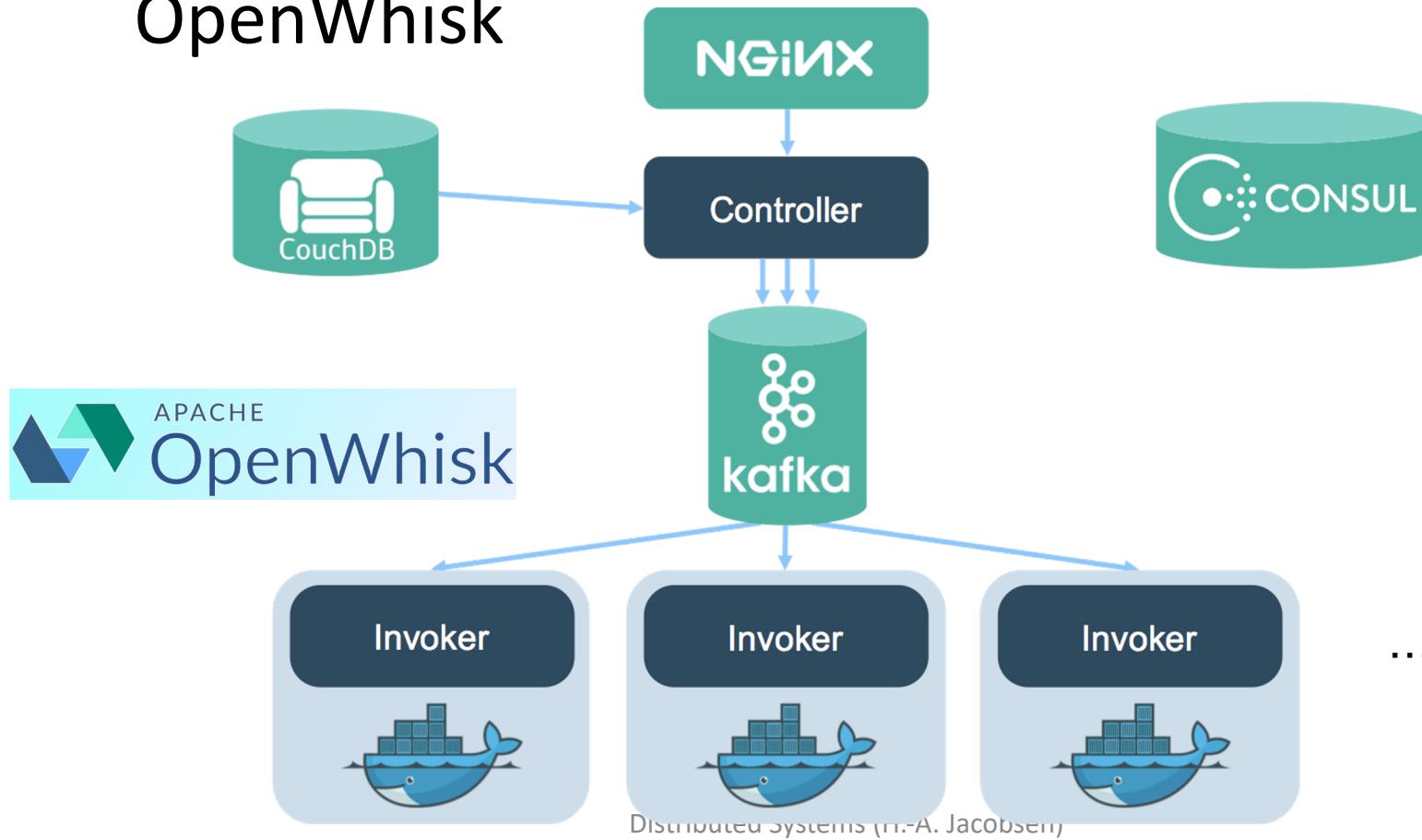
Google
Functions



Kubernetes

Apache OpenWhisk Serverless Architecture

- IBM Cloud Functions is implemented based on OpenWhisk



OpenWhisk Programming Model

all constructs first-class
— powerful extensible
language

language support to
encapsulate, share, extend code

Package

Trigger

Action

Rule

first-class
event-driven
programming
constructs

first-class functions
compose via sequences

docker
containers as
actions

Serverless Actions

- Action is a stateless function that is executed in response to an event

```
function main(params) {  
    console.log("Hello " + params.name);  
    return { msg: "Goodbye " + params.name };  
}
```



```
def lambda_handler(event, context):  
    print("hello world")
```



Triggers, Rules and Sequences

- An event triggers the execution of an action or a sequence of actions
- A Rule maps a trigger to an action



AWS Lambda Trigger Sources

DATA STORES



Amazon S3



Amazon
DynamoDB



Amazon
Kinesis



Amazon
Cognito

ENDPOINTS



Amazon
Alexa



Amazon
API Gateway



AWS IoT

CONFIGURATION REPOSITORIES



AWS
CloudFormatio
n



AWS
CloudTrail



AWS
CodeCommit



Amazon
CloudWatch

EVENT/MESSAGE SERVICES



Amazon
SES



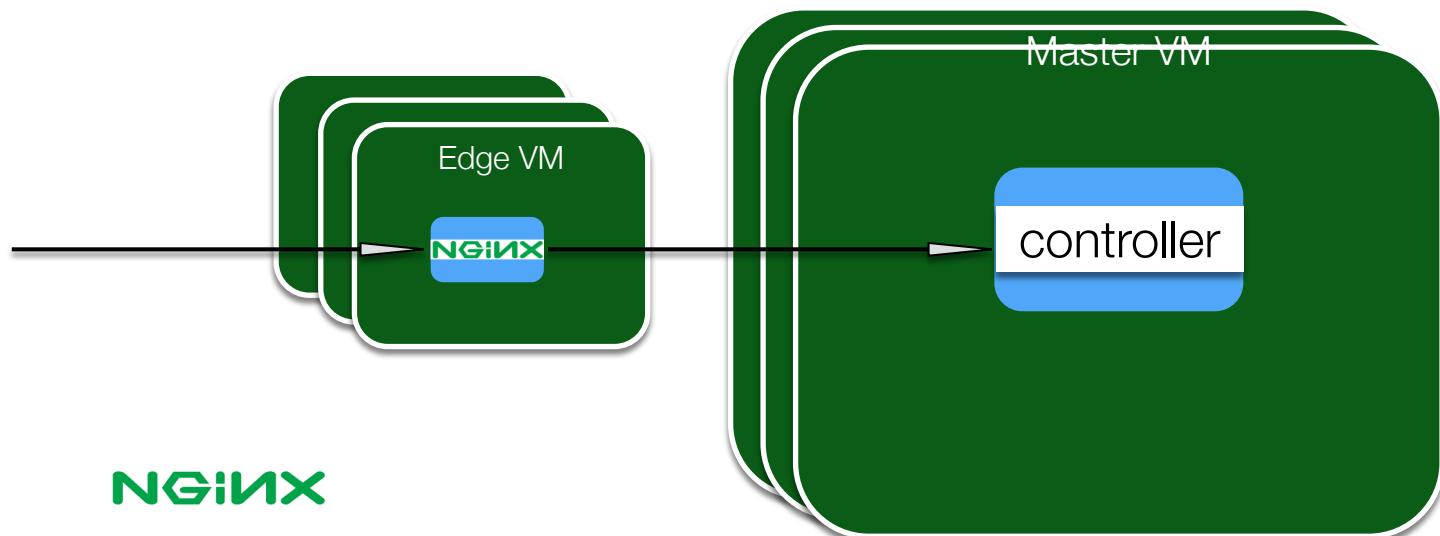
Amazon
SNS



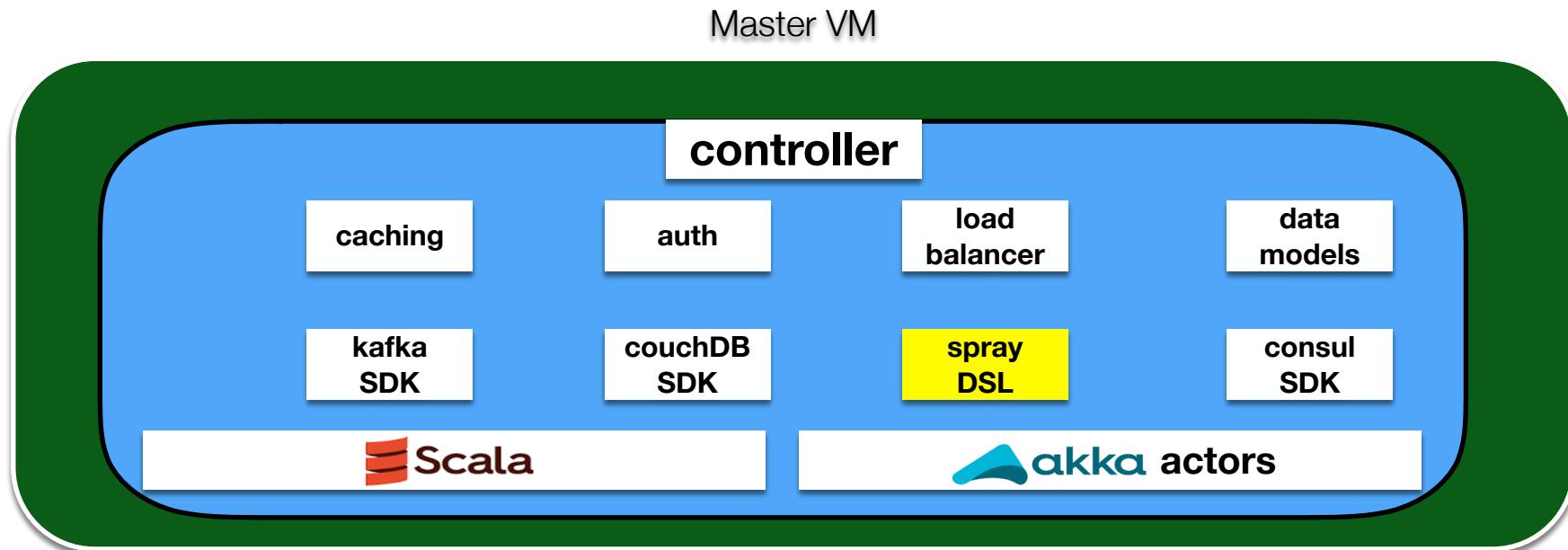
Cron events

OpenWhisk: Step 1. Entering the system

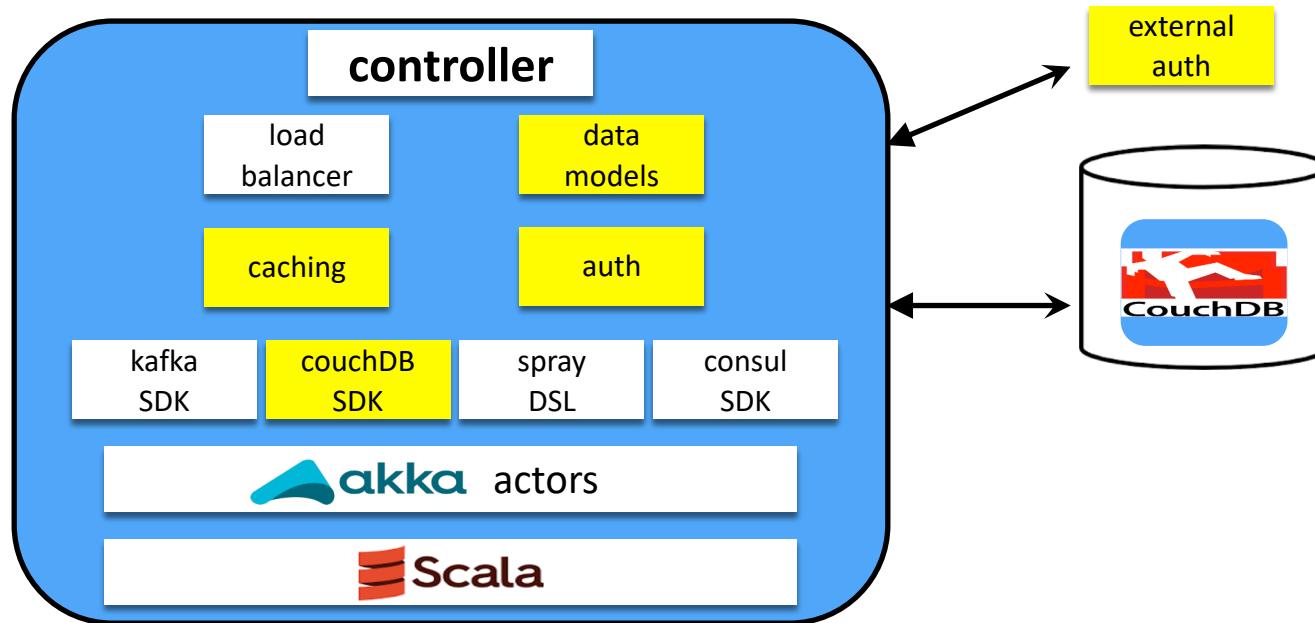
```
POST /api/v1/namespaces/myNamespace/actions/myAction
```



OpenWhisk: Step 2. Handle the request

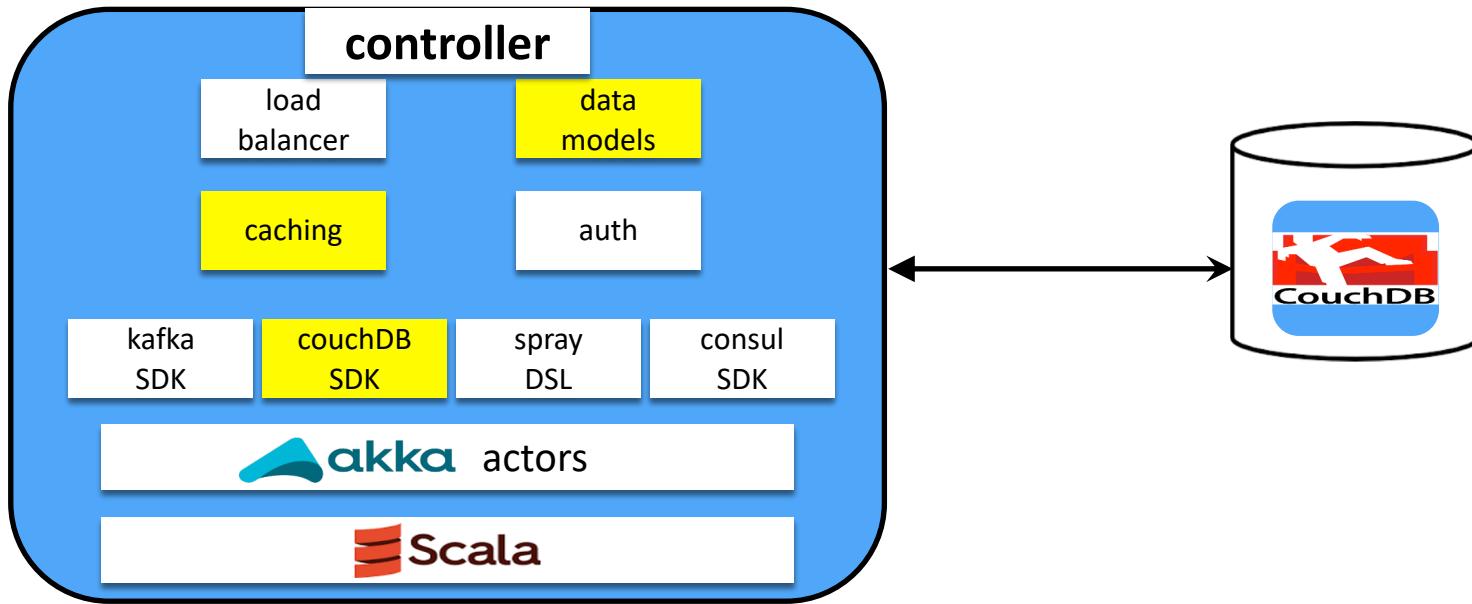


OpenWhisk: Step 3. Authentication + Authorization



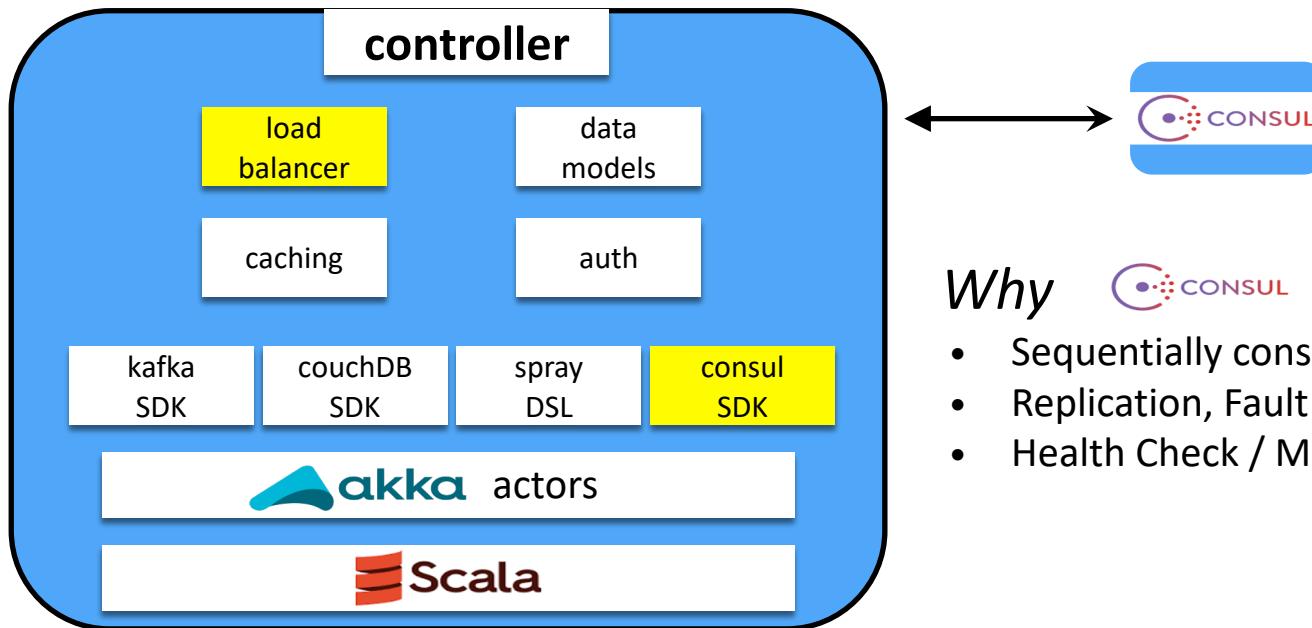
- Cloudant: hosted CouchDB
- plug-in structure for custom authentication module

OpenWhisk: Step 4. Get the action



- check resource limits
- actions stored as documents in CouchDB
 - binaries as objects (attachments)

OpenWhisk: Step 5. Looking for a home

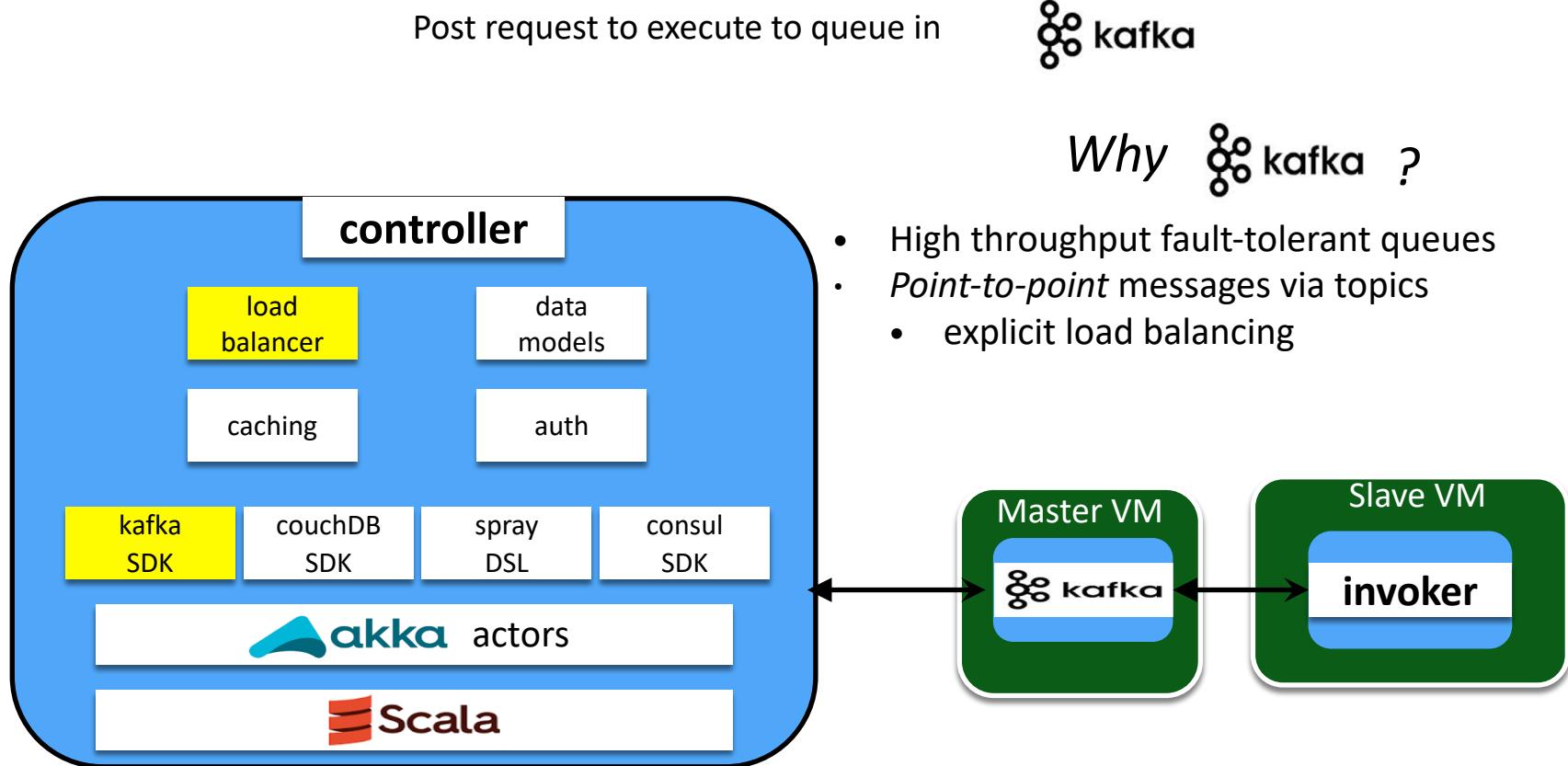


Why ?

- Sequentially consistent KV store
- Replication, Fault Tolerance
- Health Check / Monitoring utilities

Load balancer: find a slave to execute
Slave health, load stored in Consul

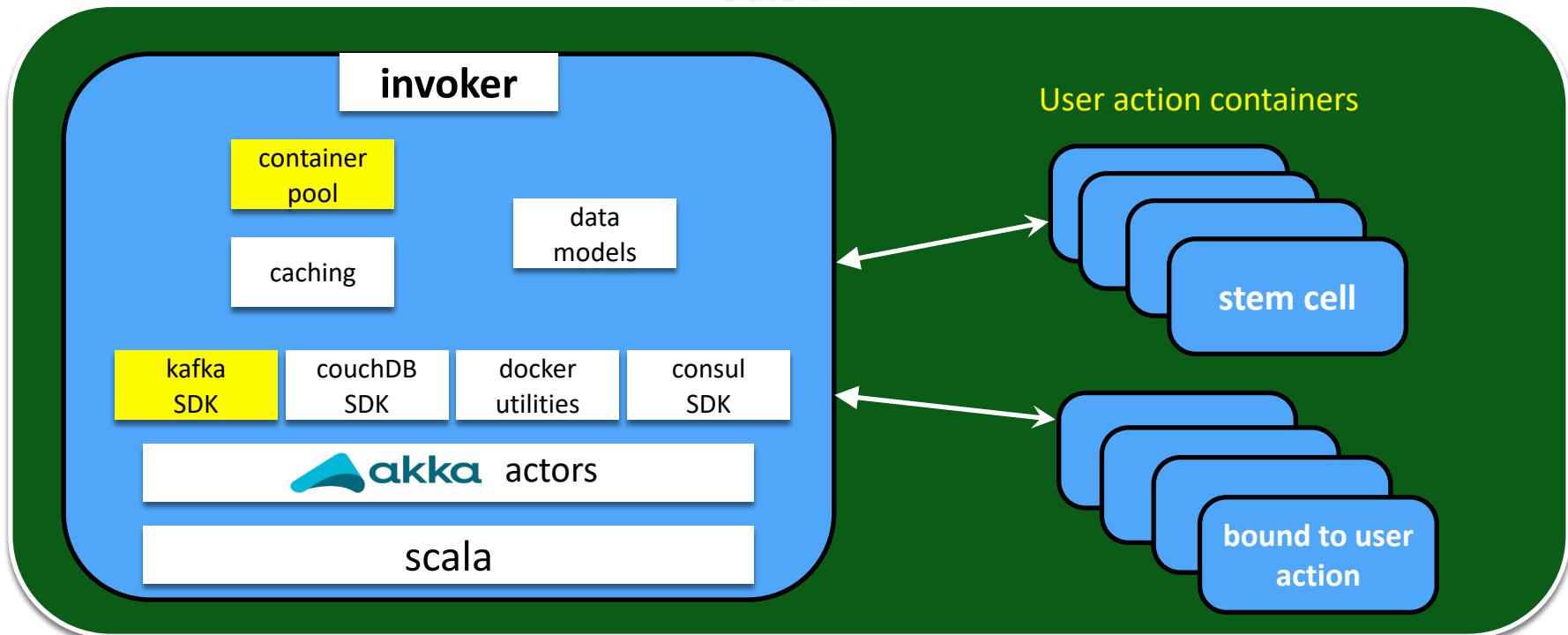
OpenWhisk: Step 6. Get in line!



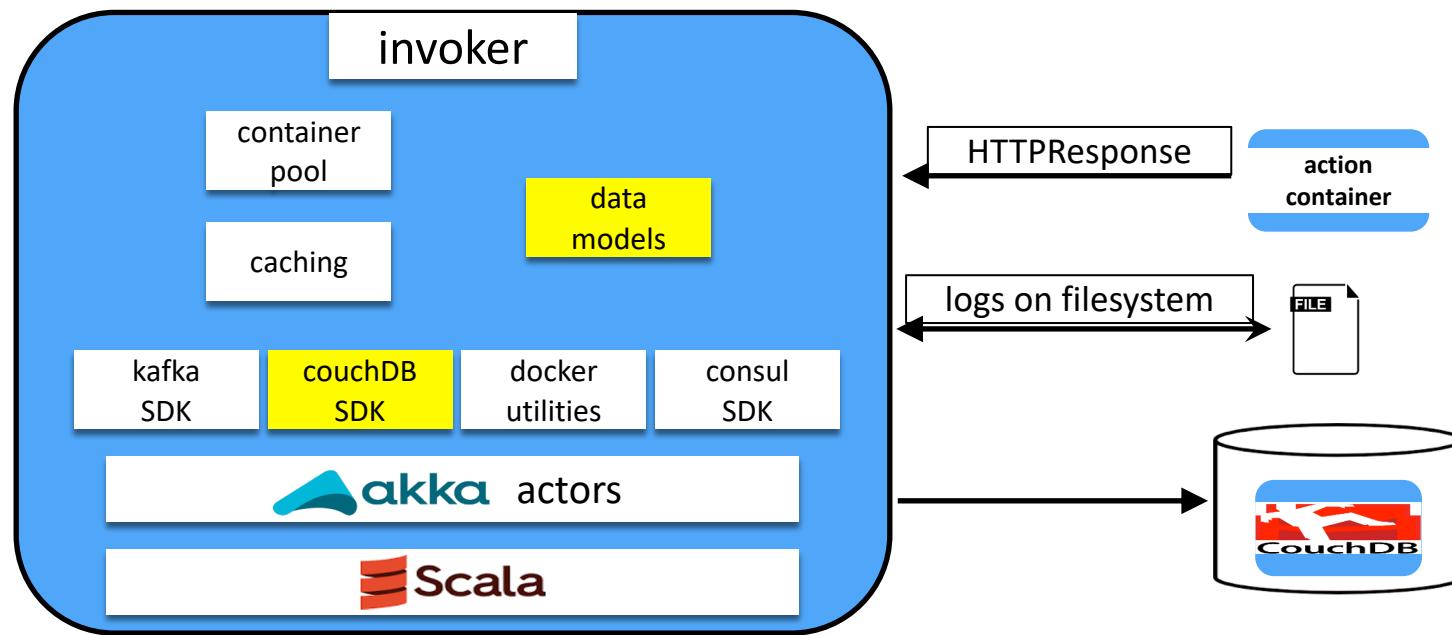
OpenWhisk: Step 7. Get to work!

- each user action gets its own container (isolation)
- containers may be reused
- container pool allocates and garbage collects containers

Slave VM



OpenWhisk: Step 8. Store the results.



Overview Serverless Computing

- An auto-scalable cloud resource for hosting and executing stateless functions
- Makes development more cost and time efficient
- A good fit for short-lived stateless microservice
- All primary cloud providers offer serverless platforms

Bonus DS Topics

Pezhman Nasirifard

Hans-Arno Jacobsen

Application & Middleware Systems Research
Group

Outline

- Structured Peer-to-Peer Systems
 - Pastry
- Anonymity Networks
 - Onion Routing
 - Tor Network
- Conflict Free Replicated Data Types

Final Exam Review (Feb 5th):
<https://forms.gle/JbyYk4S7XfUz2vT99>

STRUCTURED PEER-TO-PEER SYSTEMS

Pastry (2001)

- We have a ring of size 2^m where every peer is mapped to a node by a uniform hash function, e.g. SHA1.
- Each peer discovers and exchanges state information: List of leaf nodes, neighborhood list, routing table
- **Leaf node list** are $L/2$ closest peers by node ID in each direction around the circle
 - Lookups first search the leaf node list
- **Neighborhood list** are M closest peers in terms of routing metric (e.g. ping delay)
 - Good candidates for routing table

Pastry

- Every node ID is expressed in the base- B numeral system for some fixed B .
- Usually, in practice, $B = 16$ but for simplicity we will use $B = 4$.
- Hence, every node is expressed as a string belonging to $\{0, 1, 2, 3\}^{m/2}$.
- Furthermore, we let $m = 8$. Therefore, we have string IDs of length 4.

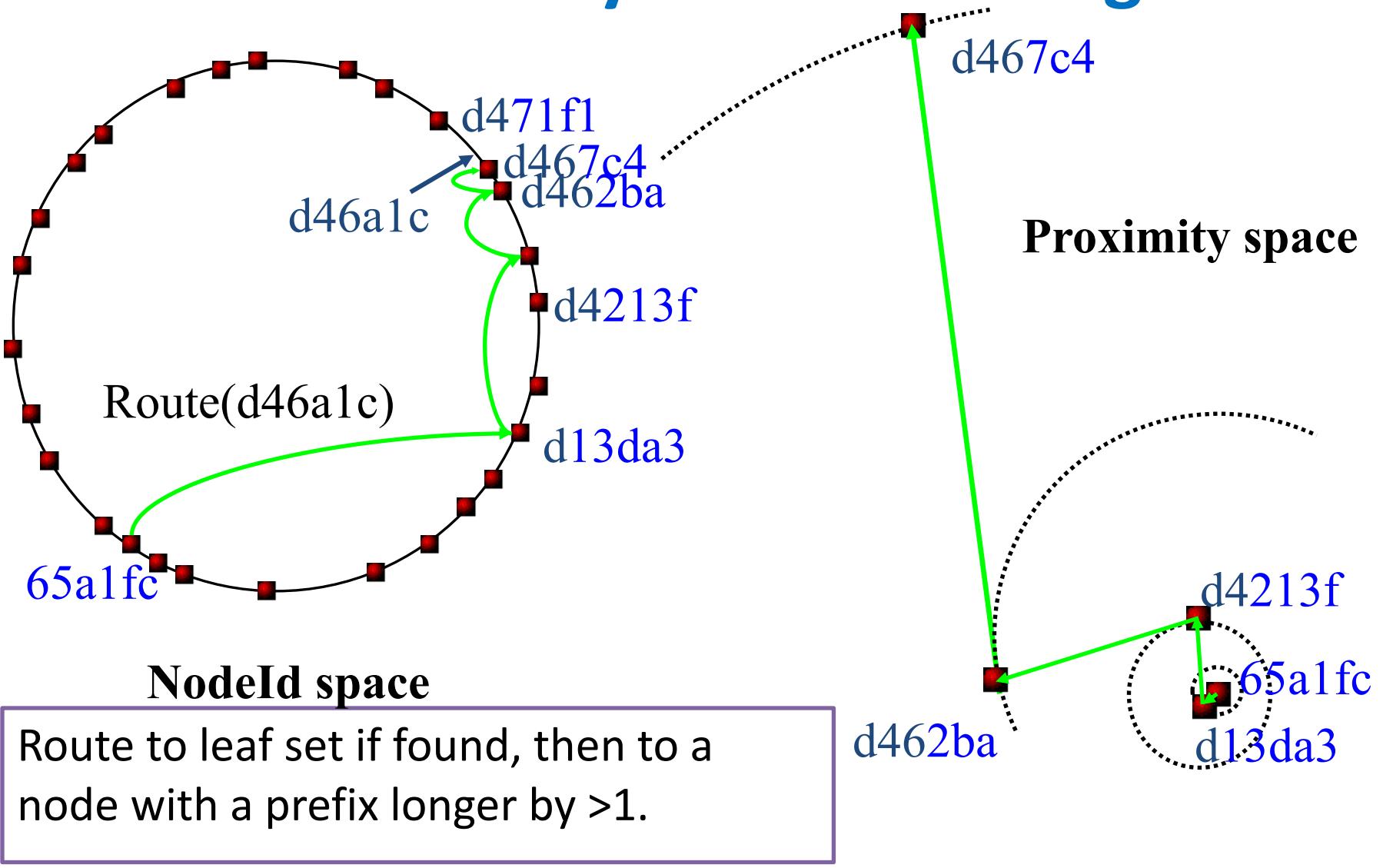
Pastry

- Every node v stores a leaf set L that contains:
 - $|L|/2$ nodes preceding v in the ring, and
 - $|L|/2$ nodes succeeding v in the ring.
- Moreover, every node stores a routing table RT , which contains $\log_B(n)$ rows and B columns where n is the max number of peer nodes in the ring.
- There is a column for each number in the set $\{0, \dots, B-1\}$, which stands for a digit of the base- B numeral system.
- Every row $i \in \{1, \dots, \log_B(n)\}$ corresponds to the digit we are currently looking at.

Routing Table

- 6 digits, base 4: table of 6x4
- Row i contains nodes which share $i\text{-}1\text{-}th$ long prefix
- Populate cells with neighbors if possible
- Column indicates $i\text{-}th$ digit
- Lookup in RT finds a node with a longer prefix

Proximity-based Routing



Tutorial Example - Pastry

- Consider the Pastry network containing the following nodes:
 $\{0023, 0113, 0133, 0322, 1002, 1010, 1132, 1223, 2000, 2112, 2210, 2231\}$
- $B = 4$.
- Every node ID $\in \{0, 1, 2, 3\}^4$.
- $|L| = 4$.

Tutorial Example - Pastry

{**0023**, 0113, 0133, 0322, 1002, 1010, **1132**, 1223, 2000, 2112, **2210**, 2231}

0023's routing table

0	1	2	3
0023	1010	2112	-
00	01	02	03
0023	0133	-	0322
000	001	002	003
-	-	0023	-

$$L = \{0113, 0133, 2210, 2231\}$$

Tutorial Example - Pastry

{**0023**, 0113, 0133, 0322, 1002, 1010, **1132**, 1223, 2000, 2112, **2210**, 2231}

1132's routing table

0	1	2	3
0113	1132	2000	-
10	11	12	13
1002	1132	1223	-
110	111	112	113
-	-	-	1132

$$L = \{1002, 1010, 1223, 2000\}$$

Tutorial Example - Pastry

{**0023**, 0113, 0133, 0322, 1002, 1010, **1132**, 1223, 2000, 2112, **2210**, 2231}

2210's routing table

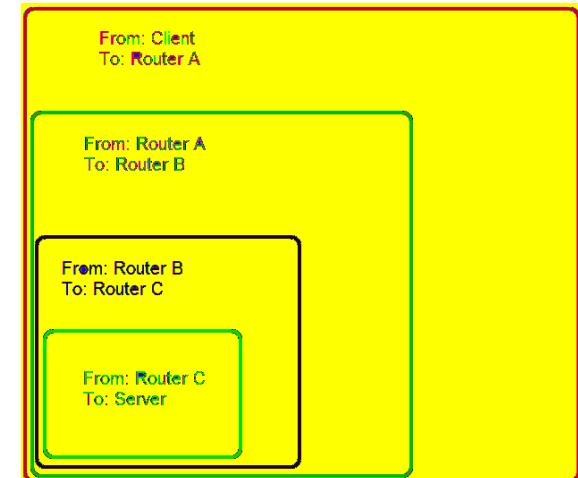
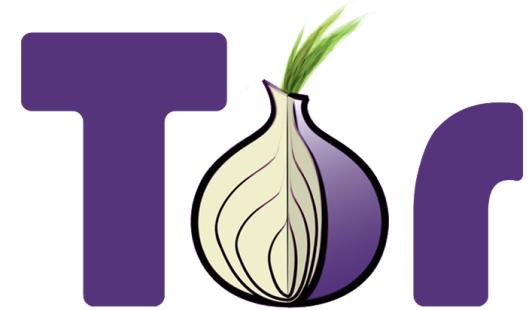
0	1	2	3
0322	1132	2210	-
20	21	22	23
2000	2112	2210	-
220	221	222	223
-	2210	-	-

$$L = \{2000, 2112, 2231, 0023\}$$

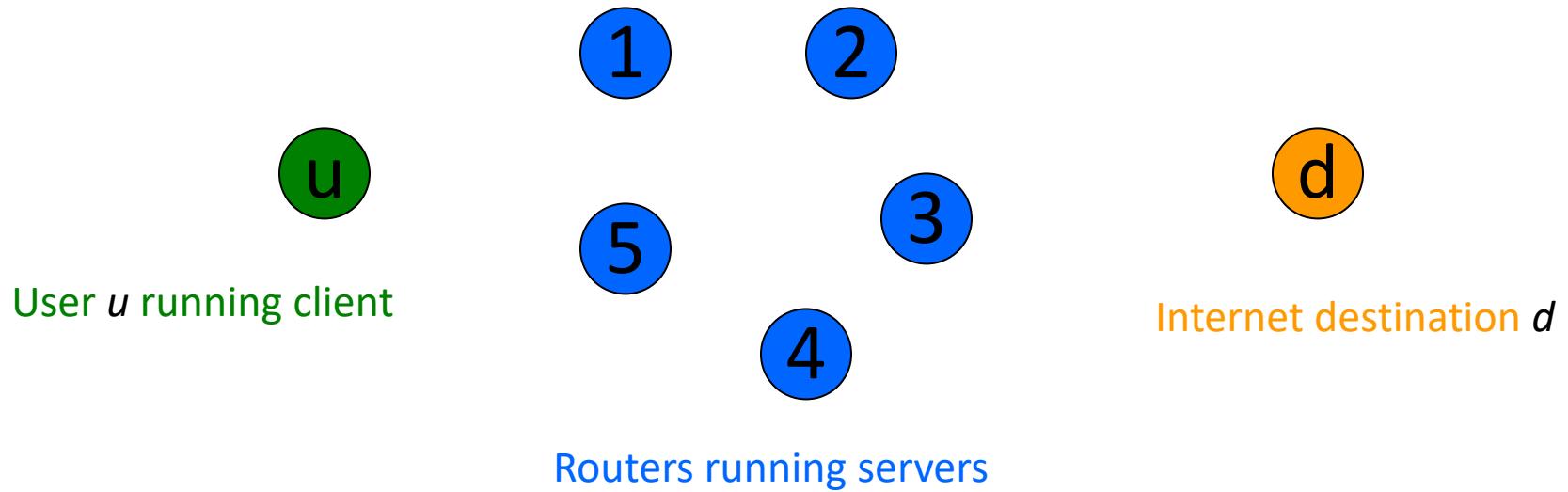
ANONYMITY NETWORKS

Onion Routing (mid-90's)

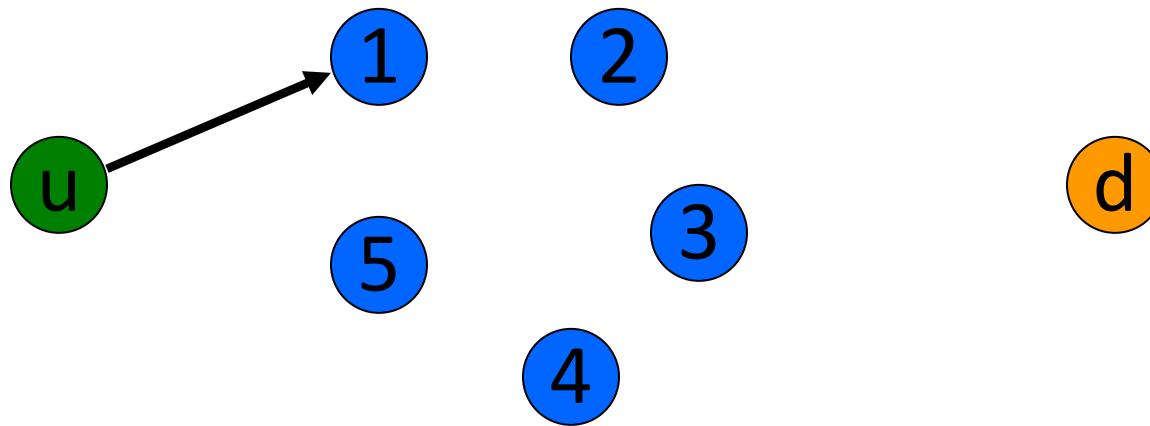
- Circuit routing
- Use a path of intermediaries to circulate message
- Message is an “onion”
- Encrypted with different keys for each router
- Preserve anonymity of the entire path
- Implementation: The Onion Router (TOR)



How Onion Routing Works

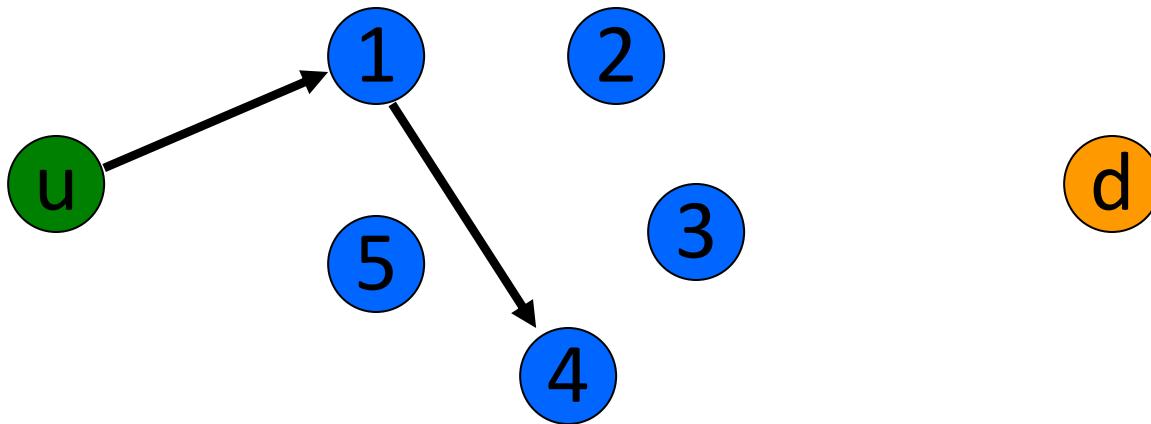


How Onion Routing Works



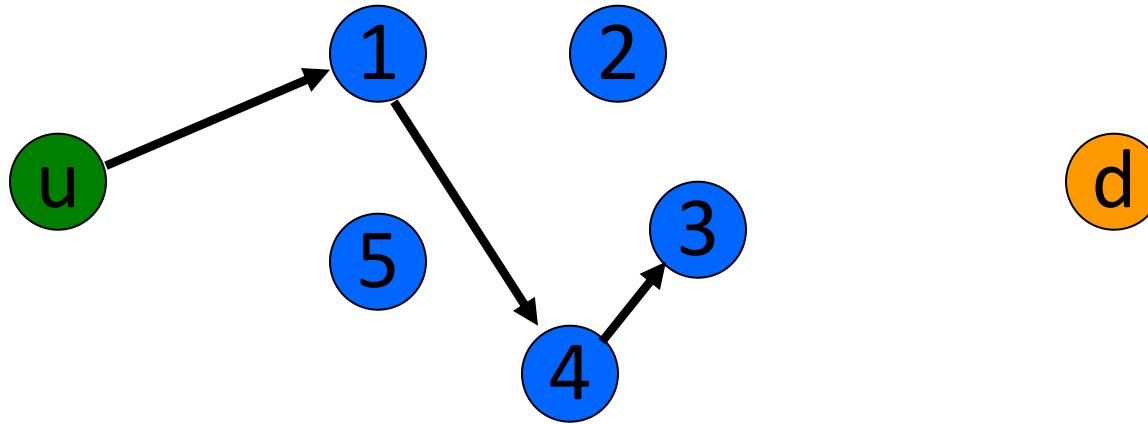
1. u creates l -hop circuit through routers

How Onion Routing Works



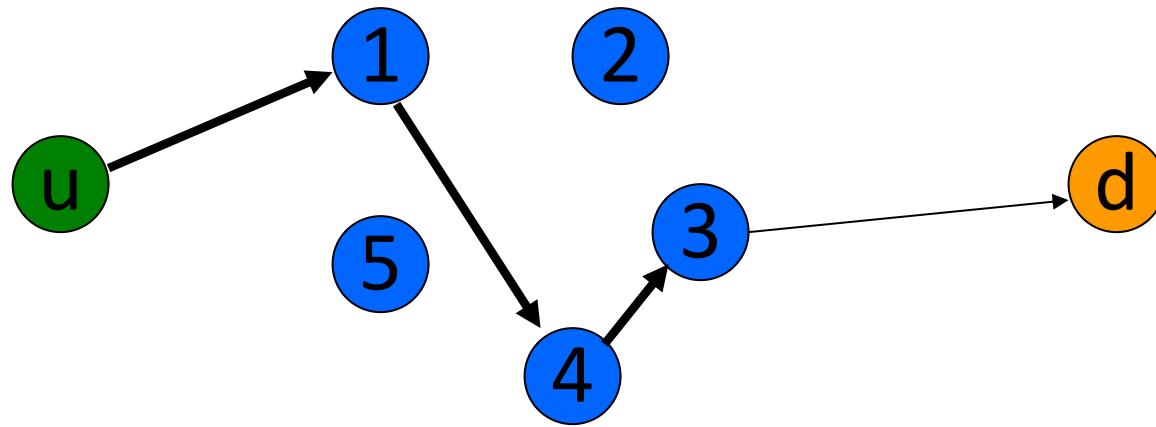
1. u creates l -hop circuit through routers

How Onion Routing Works



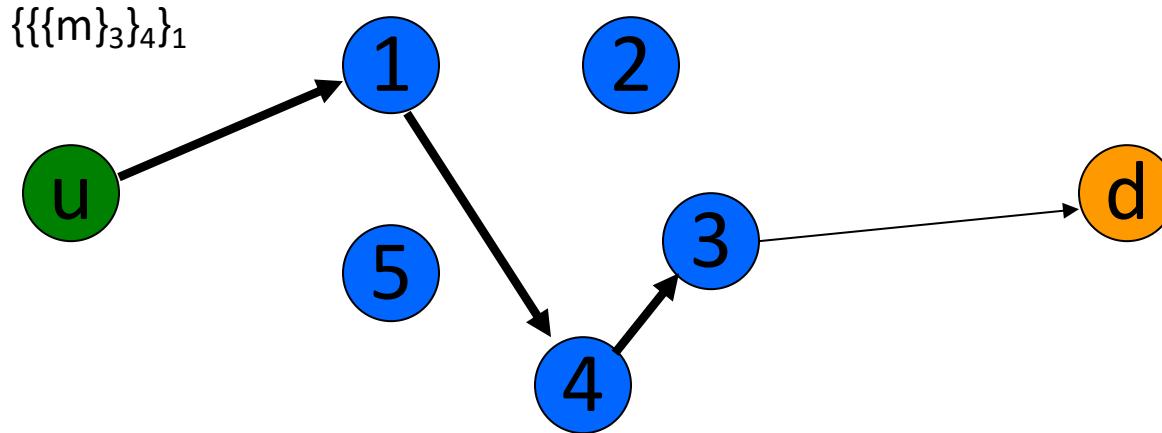
1. u creates l -hop circuit through routers

How Onion Routing Works



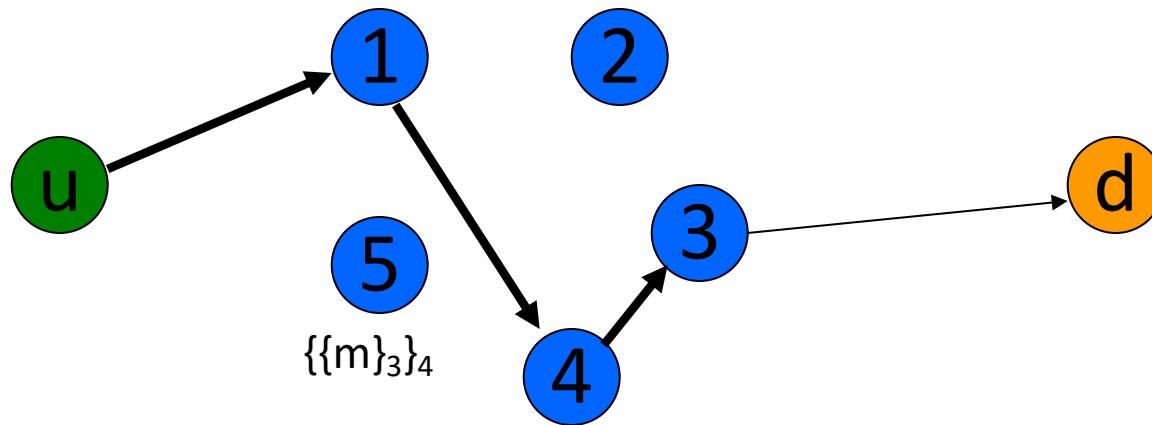
1. u creates l -hop circuit through routers
2. u opens a stream in the circuit to d

How Onion Routing Works



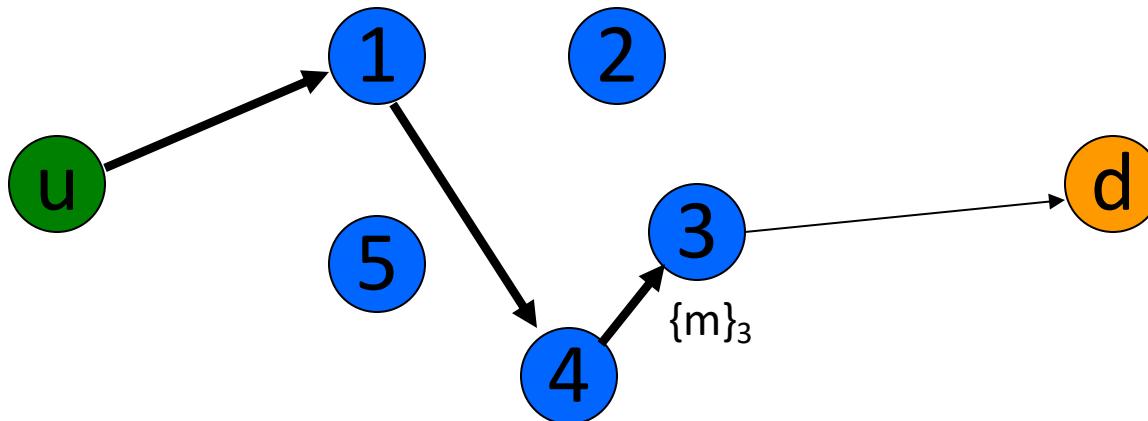
1. u creates l -hop circuit through routers
2. u opens a stream in the circuit to d
3. Message m is encrypted using intermediate public keys by the chosen circuit

How Onion Routing Works



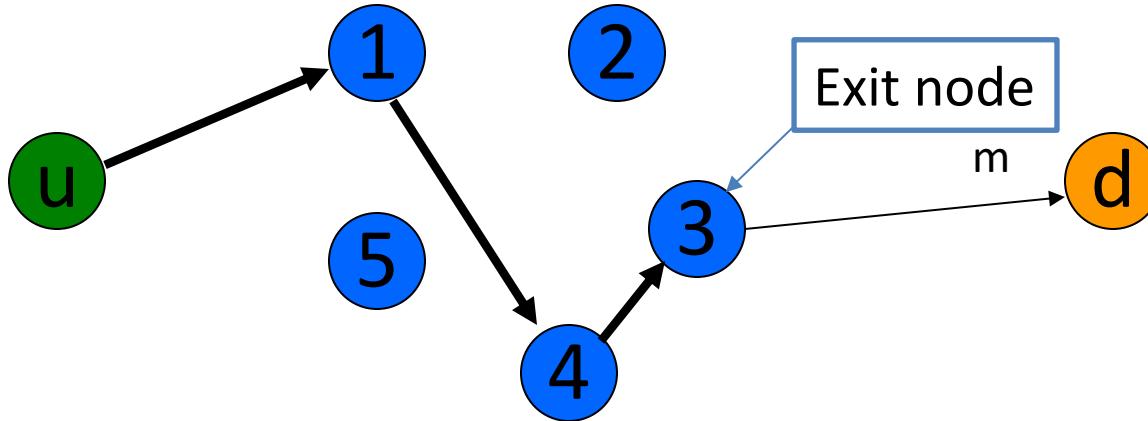
1. u creates l -hop circuit through routers
2. u opens a stream in the circuit to d
3. Message m is encrypted using intermediate public keys by the chosen circuit
4. Each router decrypts part of the message

How Onion Routing Works



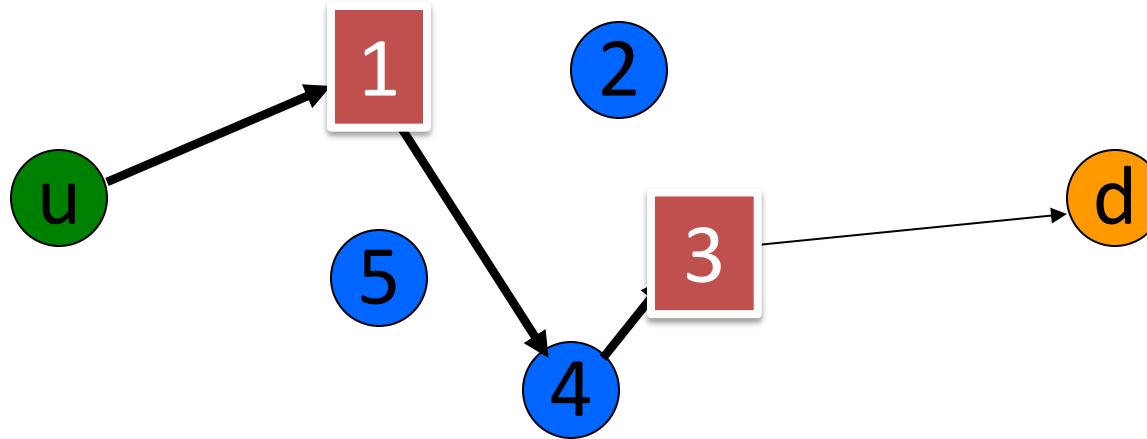
1. u creates l -hop circuit through routers
2. u opens a stream in the circuit to d
3. Message m is encrypted using intermediate public keys by the chosen circuit
4. Each router decrypts part of the message

How Onion Routing Works



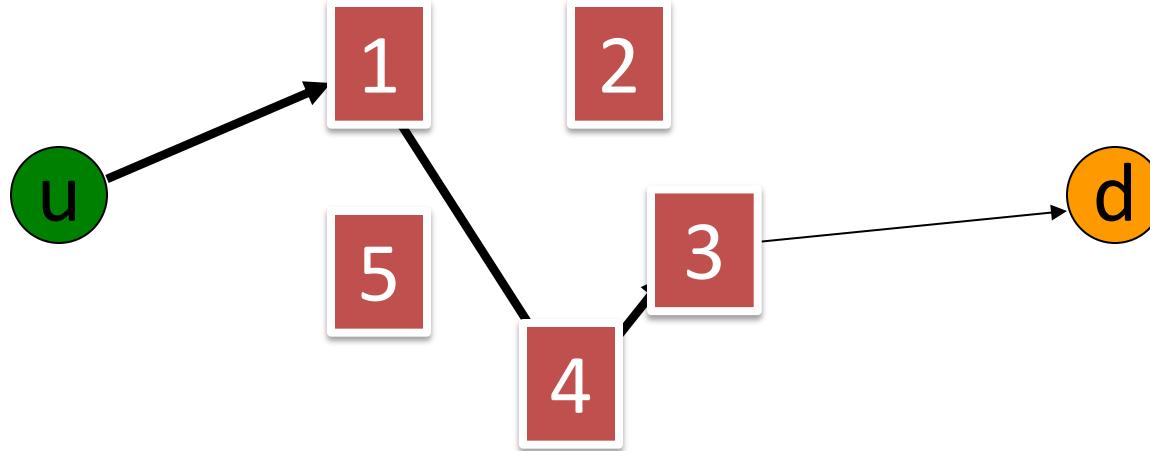
1. u creates l -hop circuit through routers
2. u opens a stream in the circuit to d
3. Message m is encrypted using intermediate public keys by the chosen circuit
4. Each router decrypts part of the message
5. Exit node sends unencrypted message to d

Onion Routing Security



1. If entry node is compromised, it can only determine that u is talking
2. If exit node is compromised, it can only determine that d is listening
3. Circuit is changed often to reduce the risk of timing analysis or long-term correlation

Onion Routing Weaknesses



1. Onion routing does not work if the whole/most network is compromised!
2. The whole path is then visible to the attacker
3. Exit node vulnerability: can see unencrypted content

Freenetproject.org, since 2000

- Goals by founder: “*Providing freedom of speech with strong anonymity protection.*”
- Protects **anonymity** of participants
- Platform for **censorship-resistant** communication
 - Publish sensitive material **anonymously**
- Decentralized, highly survivable, distributed **cache** (blogs, pages, files, etc.)
 - Using P2P for redundant storage

Freenet

- Pure unstructured network: fully peer-to-peer, no dedicated clients or servers
- Only enables access to information previously inserted (it is not a Web proxy)
- Every node contributes a configurable amount of storage

Anonymity requirement & implications

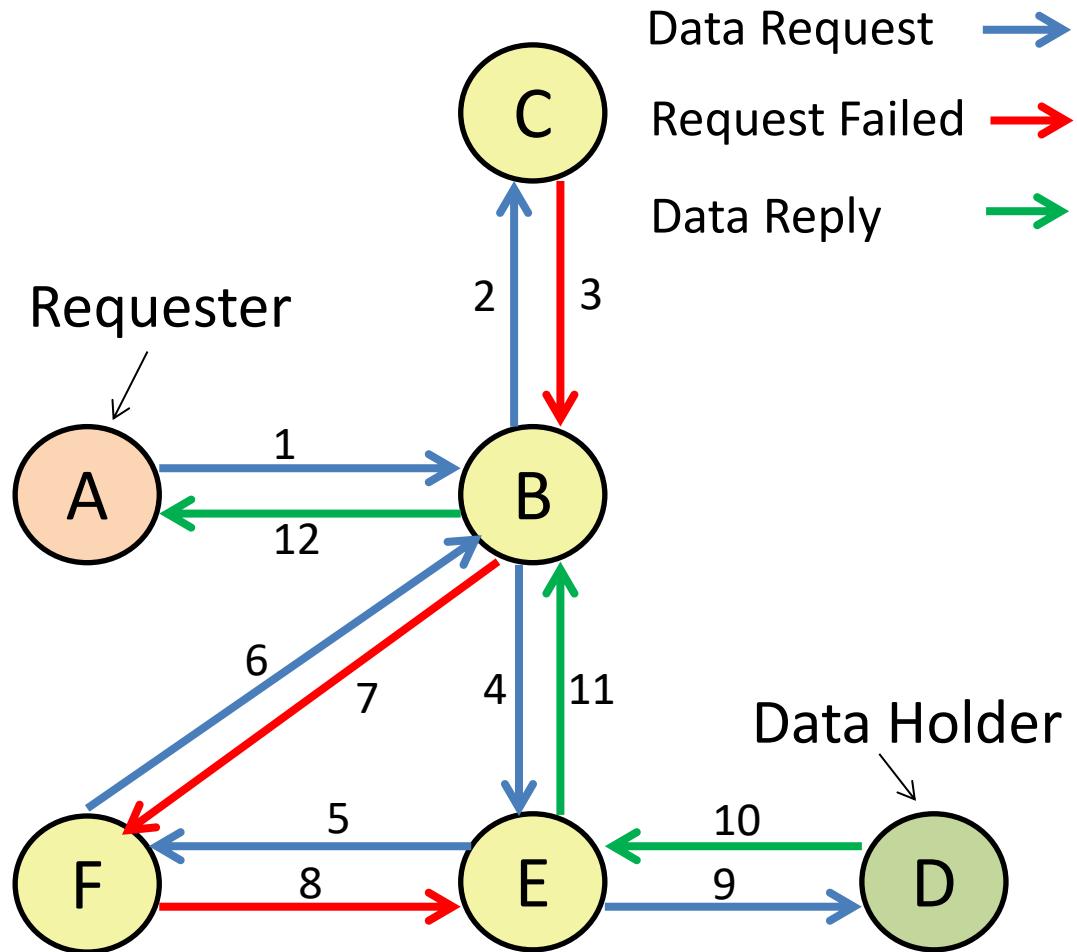
- Anonymity for information **upload & download**
- No need for source to stay after upload
- Files are broken into **encrypted blocks** and are **redundantly stored** across network
 - Encrypted traffic: No external evidence of peer activities
- For download, **blocks** are **found** and **reassembled**
- Node requesting a datum does not connect directly to node that has datum
- Datum **routed across intermediaries**, none of which know request originator or location
- **Higher bandwidth** use required, **slower transfers**

Key disadvantage of storage model

- No one node is responsible for any block of data
- If data is not retrieved for some time, old data might be dropped, if space is exceeded by newly arriving data
- Therefore, Freenet tends to 'forget' data, not retrieved regularly
- There is no way to delete data (unless it is "forgotten")

Request processing (simplified)

- Queries
 - Requests get routed to correct peer by incremental discovery
- Data flows in reverse path of query
 - Impossible to know if a user is **initiating** or **forwarding** a query
 - Impossible to know if a user is **consuming** or **forwarding** data



* Figure from <http://en.wikipedia.org/wiki/Freenet>

CONFLICT FREE REPLICATED DATA TYPES

Consistency & Transactions

Agenda

- Quick recap eventual consistency
- Replicated state-based object
- Conflict-free replicated data types

Eventual Consistency

- Eventual consistency states all replicas **eventually** converge when write operations stop
 - e.g., lazy replication using gossiping (cf. replication)
- Weak form of consistency with no time bound, but **highly-available** (i.e., always return a value, but value could be stale)

Eventual Consistency

- Works fine if a client always reads from the **same** replica...
- ...but gives weird results when the client reads from **multiple replicas**
 - Mobile scenario
 - Replica failure
- Client-centric consistency models describe what must happen when a **single** client reads from **multiple replicas**

Eventual Consistency

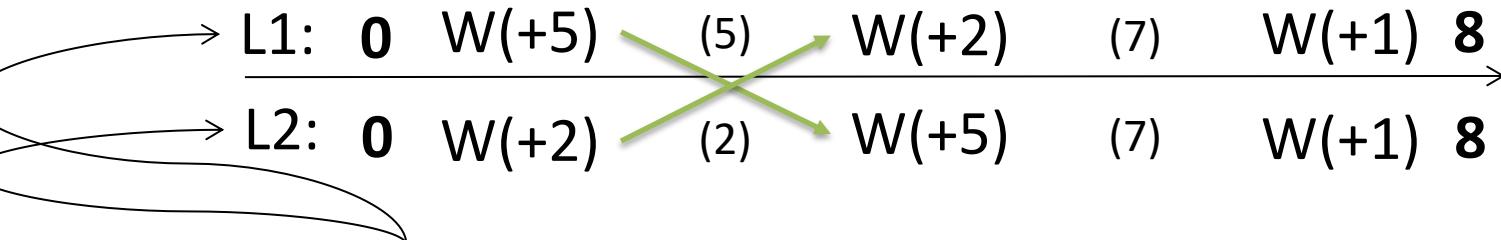
- Eventual consistency is desirable for **large-scale distributed systems** where **high availability** is important
- Tends to be cheap to implement (e.g., gossip)
- Constitutes a **challenge** for environments where **stronger consistency is important**

Handling Concurrent Writes

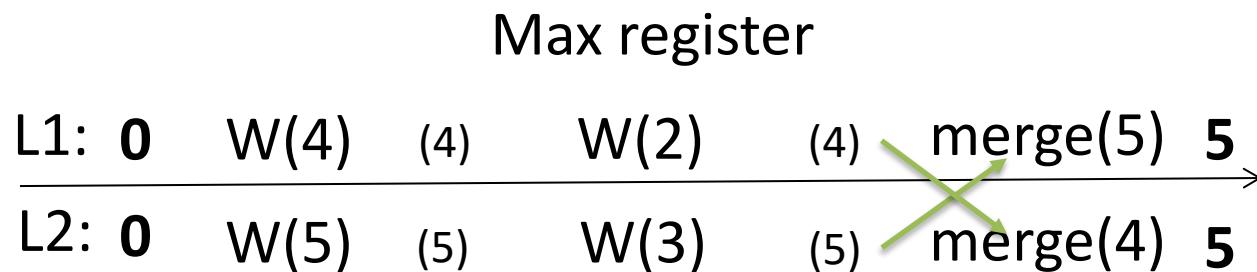
- Do need a mechanism to **handle concurrent writes**
- **If there were** a nice way to **handle concurrent writes**, we could support **eventual consistency** more broadly
- “Only” need to guarantee that after processing all writes for an item, all replicas converge, **no matter what order** the writes are processed

Example

Growth-only counter (G-counter)



Different locations (replicas)



System Model

- Replicated state-based objects
- Offer update and query requests to clients
- Objects perform merge requests amongst each other
- Clients send requests to given objects
- Objects periodically merge

State-based Object

- What we commonly know as object
- Comprised of
 - Internal state
 - One or more query methods
 - One or more update methods
 - A merge method

Class Average

```
class Avg(object):  
    def __init__(self):  
        self.sum = 0  
        self.cnt = 0  
  
    def query(self):  
        if self.cnt != 0:  
            return  
                self.sum /  
                self.cnt  
  
        else:  
            return 0
```

```
def update(self, x):  
    self.sum += x  
    self.cnt += 1  
  
def merge(self, avg):  
    self.sum += avg.sum  
    self.cnt += avg.cnt
```

Average

State-based object representing a running average

- Internal state
 - `self.sum` and `self.cnt`
- Query returns the average
- Update updates the average with a new value x
- Merge merges one Average instance into another one

Replicated State-based Object

- State-based object replicated across multiple nodes
- E.g., replicate Average across two nodes
- Both nodes have a copy of the state-based object
- Clients send query and update to a node
- Nodes periodically send their copy of the state-based object to other nodes for merging

Time-line

Node 1

State
 a_0



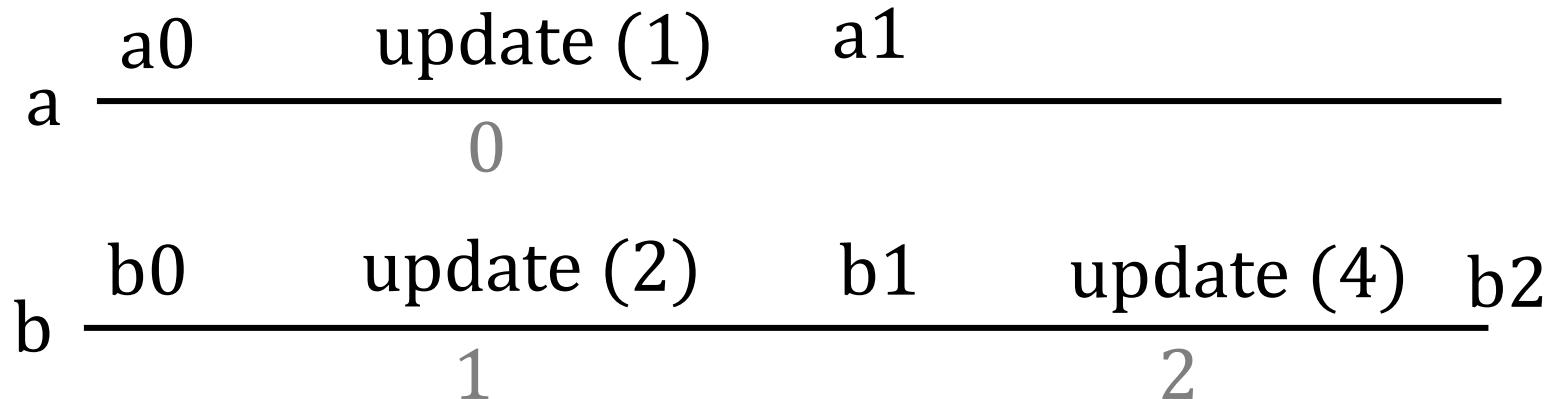
Unique operation identifier

Causal history based on operation identifiers

Each state represents a snapshot of the object in time that results from the updates applied.

	state	query ()	history
a0	sum:0, cnt:0	0	{}
a1	sum:1, cnt:1	1	{0}
a2	sum:4, cnt:2	2	{0,1}

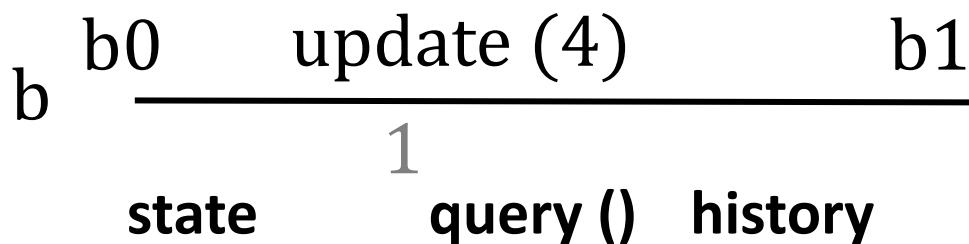
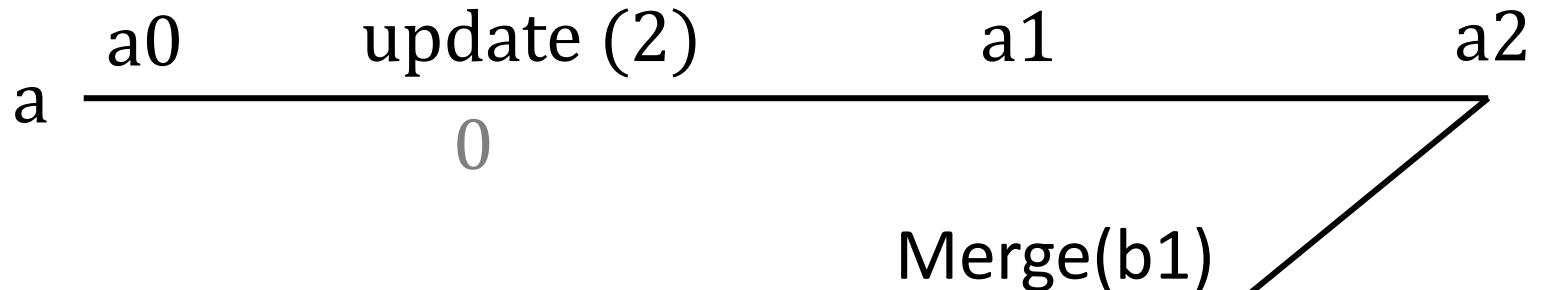
States and Causal Histories



If $y = x.\text{update}()$ where the update has identifier i , then the causal history of y is the causal history of x union $\{i\}$.

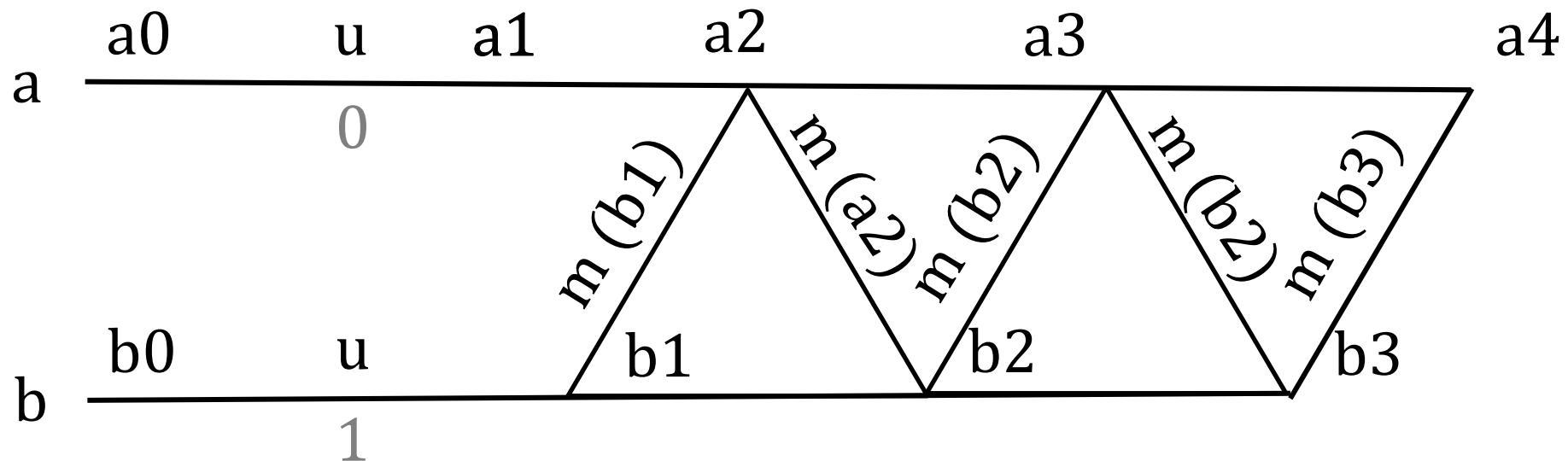
state	query ()	history
a0	sum:0, cnt:0	0
a1	sum:1, cnt:1	1
b0	sum:0, cnt:0	0
b1	sum:2, cnt:1	2
b2	sum:6, cnt:2	{1,2}

Merge



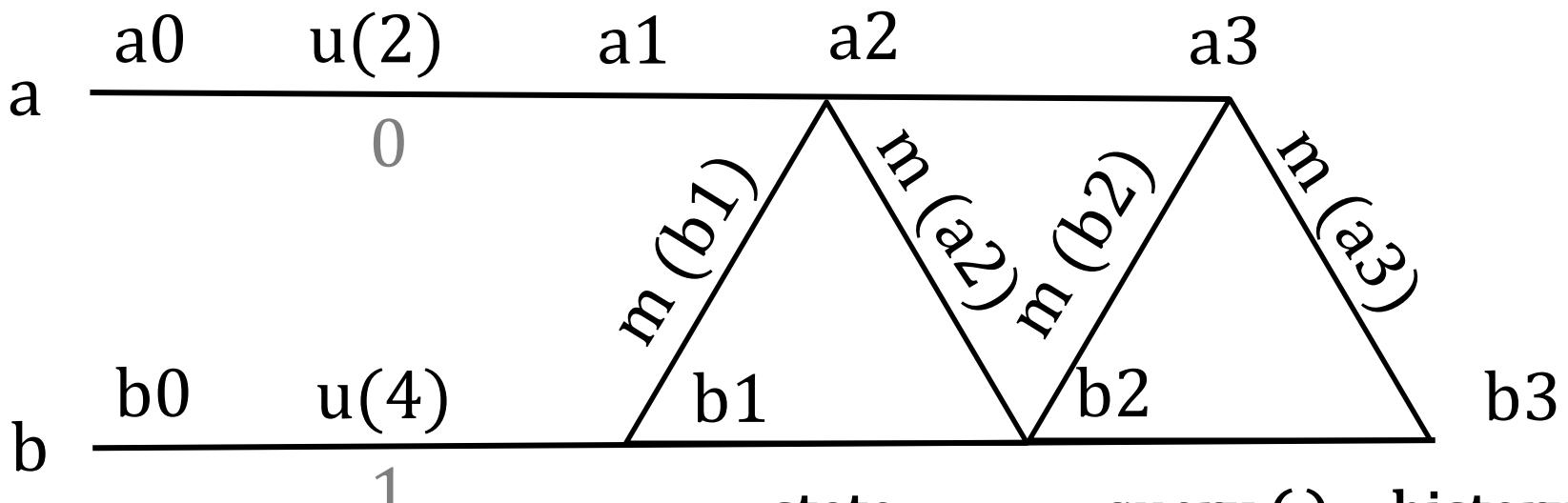
	state	query ()	history
a0	sum:0, cnt:0	0	{ }
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
b0	sum:0, cnt:0	0	{ }
b1	sum:4, cnt:1	4	{1}

Nodes Periodically Propagate Their State



Strong Eventual Consistency and Eventual Consistency

- A replicated state-based object is **eventually consistent** if whenever two replicas of the state-based object have the same causal history, they eventually (not necessarily immediately) converge to the same internal state.
- A replicated state-based object is **strongly eventually consistent** if whenever two snapshots of the state-based object have the same causal history, they (immediately) have the same internal state.
- Strong eventual consistency implies eventual consistency.



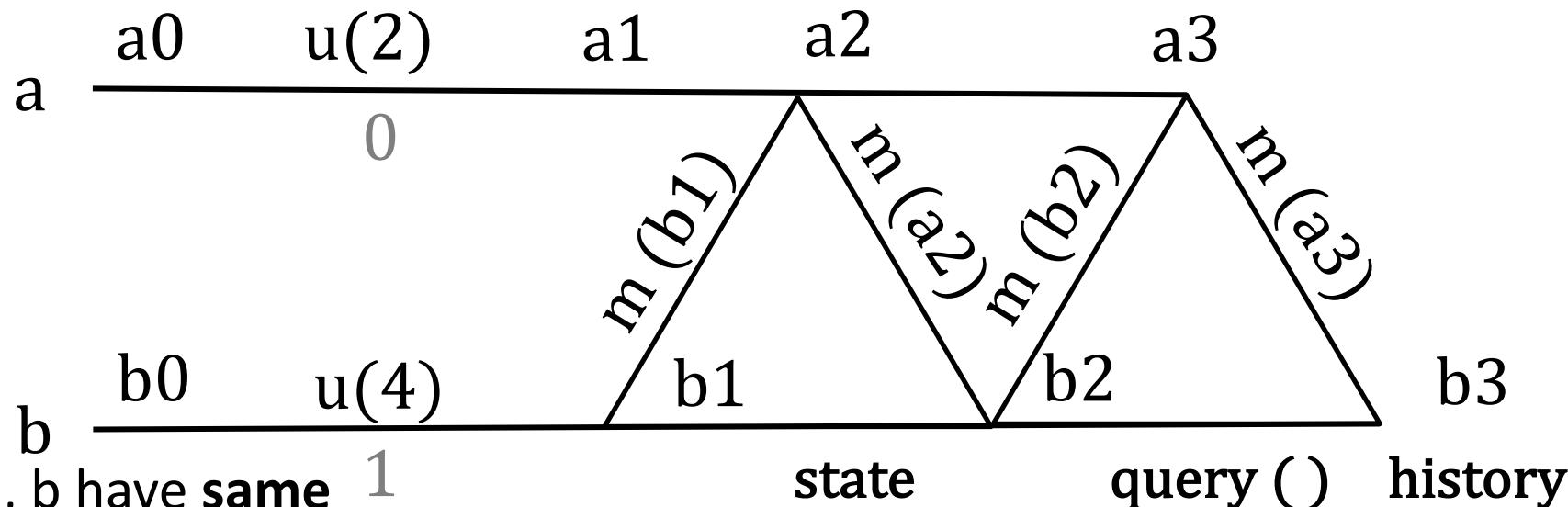
a, b attain the **same causal history** but **do not converge** to the **same internal state** – they do not converge at all

Neither eventually consistent, nor strongly eventually consistent

	state	query ()	history
a0	sum:0, cnt:0	0	{}
a1	sum:2, cnt:1	2	{0}
a2	sum:6, cnt:2	3	{0,1}
a3	sum:16, cnt:5	3.2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:10, cnt:3	3.3	{0,1}
b3	sum:26, cnt:8	3.25	{0,1}

NoMergeAverage State-based Object

- Object's merge does nothing
- All else is the same as for Average



a, b have same 1

causal history, both converge to a stable but *different internal state*.

Neither eventually consistent, nor

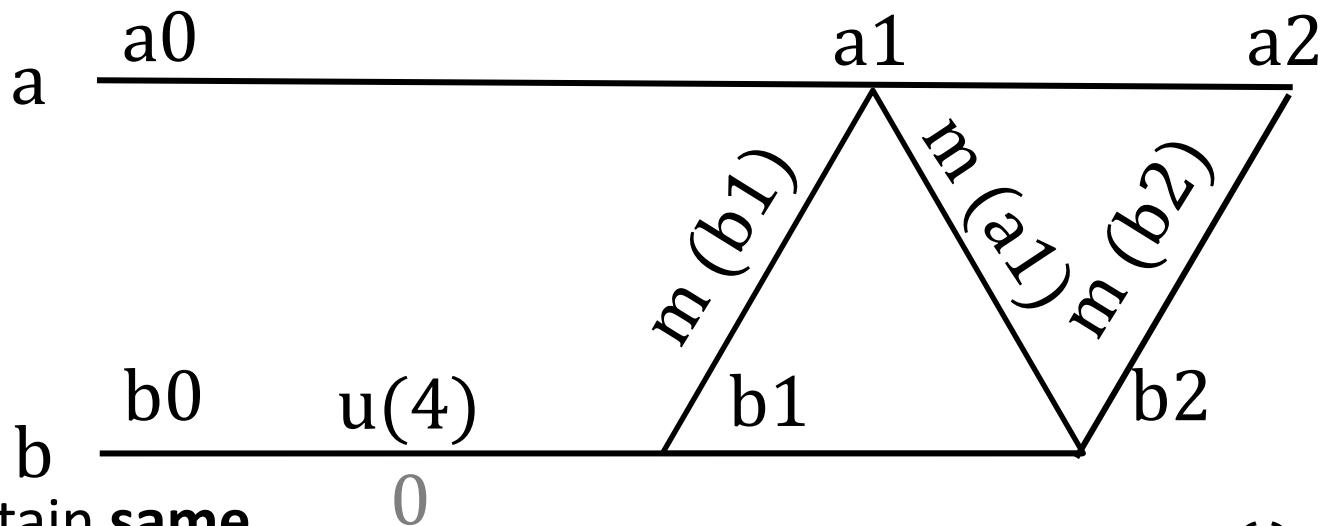
strongly eventually consistent



	state	query ()	history
a0	sum:0, cnt:0	0	{}
a1	sum:2, cnt:1	2	{0}
a2	sum:2, cnt:1	2	{0,1}
a3	sum:2, cnt:1	2	{0,1}
b0	sum:0, cnt:0	0	{}
b1	sum:4, cnt:1	4	{1}
b2	sum:4, cnt:1	4	{0,1}
b3	sum:4, cnt:1	4	{0,1}

BMergeAverage

- Object's merge
 - At b – overwrite state with state at a
 - At a – do nothing
- All else is the same as for Average



a, b attain **same causal history**, both eventually **converge** to the same **internal state – eventual consistent**.

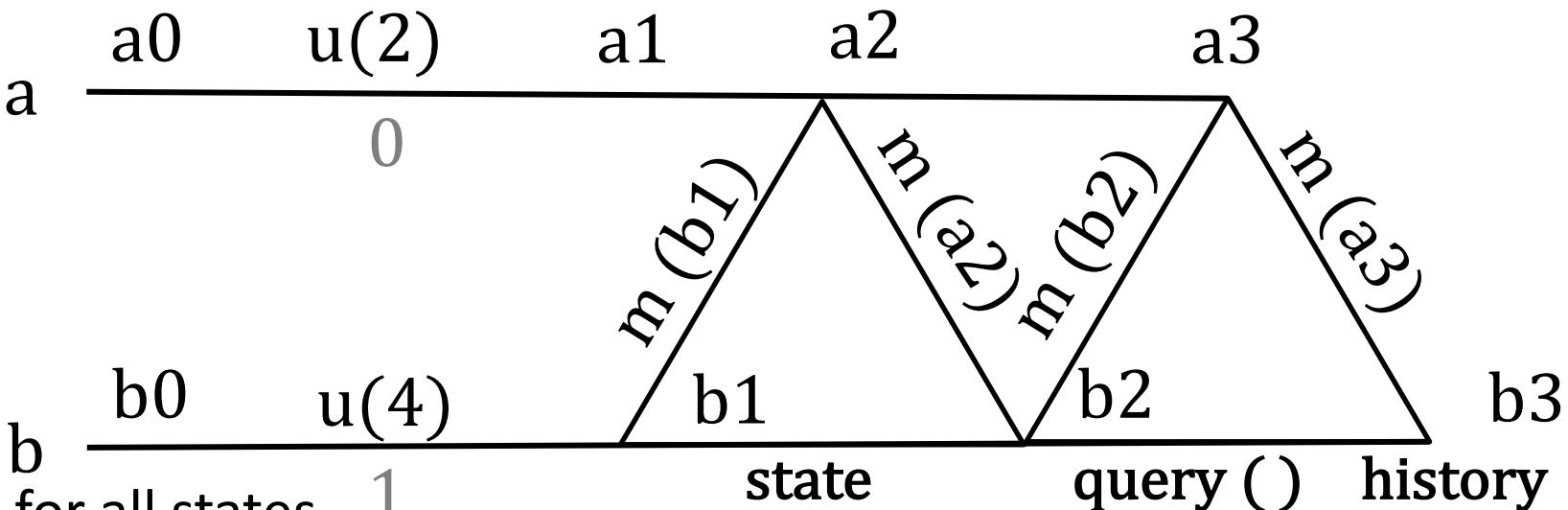
a1, b1 have same causal history but different internal state

– **not strongly eventually consistent**

state	query ()	histor y
a0	sum:0, cnt:0	0
a1	sum:0, cnt:0	{0}
a2	sum:0, cnt:0	{0}
b0	sum:0, cnt:0	{}
b1	sum:4, cnt:1	{0}
b2	sum:0, cnt:0	{0}

MaxAverage State-based Object

- Object's merge
 - Pair-wise max of sum and cnt
- All else is the same as for Average



At a, b for all states

with the **same causal history**, they have the **same internal state – strongly eventually consistent.**

Great!!! But, what does it actually compute? Here, update(2) overwritten by update(4)! ☺

state	query ()	history
a0	sum:0, cnt:0	0
a1	sum:2, cnt:1	{0}
a2	sum:4, cnt:1	{0,1}
a3	sum:4, cnt:1	{0,1}
b0	sum:0, cnt:0	0
b1	sum:4, cnt:1	{1}
b2	sum:4, cnt:1	{0,1}
b3	sum:4, cnt:1	{0,1}

Lessons Learned I

	C?	EC?	SEC?
Average	no	no	no
NoMergeAverage	yes	no	no
BMergeAverage	yes	yes	no
MaxAverage	yes	yes	yes

Designing a strongly eventually consistent state-based object with intuitive semantics is challenging

Lessons Learned II

- Replicated state-based object
- Convergence
- Eventual convergence in this model
- Strong eventual convergence in this model

Conflict-Free Replicated Data Types

- CRDT is a conflict-free replicated state-based object
- CRDT can handle concurrent writes
- **Solution:** do not allow writes with arbitrary values, limit to write operations which are **guaranteed not to conflict**
- CRDTs are data structures with **special** write operations; they guarantee **strong eventual converge** and are monotonic (no rollbacks)
- CRDTs are no panacea but a great solution when they apply!

Conflict-Free Replicated Data Types

- CRDTs can be **commutative / op-based (CmRDT)**:
 - **Example:** A growth-only counter, which can only process *increment* operations
 - Propagate operations among replicas (**duplicate-free messaging**)
- CRDTs can be **convergent / state-based (CvRDT)**:
 - **Example:** A max register, which stores the maximum value written
 - Propagate and merge states (**idempotent**)
- Therefore, the value of a CRDT depends on **multiple write operations or states**, not just the latest one

State-based CRDTs

- A CRDT is a replicated state-based object
- Supports
 - Query
 - Update
 - Merge

CRDT Properties

A CRDT is a replicated state-based object that satisfies

- Merge is **associative** (e.g., $(A + (B + C)) = ((A + B) + C)$)
 - For any three state-based objects x , y , and z ,
 $\text{merge}(\text{merge}(x, y), z)$ is equal to $\text{merge}(x, \text{merge}(y, z))$
- Merge is **commutative** (e.g., $A + B = B + A$)
 - For any two state-based objects, x and y , $\text{merge}(x, y)$ is equal to $\text{merge}(y, x)$
- Merge is **idempotent**
 - For any state-based object x , $\text{merge}(x, x)$ is equal to x
- Every **update is increasing**
 - Let x be an arbitrary state-based object and let $y = \text{update}(x, \dots)$ be the result of applying an arbitrary update to x
 - Then, update is increasing if $\text{merge}(x, y)$ is equal to y

Max Register is a CRDT

The state-based object IntMax is a CRDT

- IntMax wraps an integer
- Merge (a, b) is the max of a, b
- Update (x) adds x to the wrapped integer
- Prove that IntMax is associative, commutative, idempotent, increasing

```
class IntMax(object):  
    def __init__(self):  
        self.x = 0  
    def query(self):  
        return self.x  
    def update(self, x):  
        assert x >= 0  
        self.x += x  
    def merge(self, other):  
        self.x =  
            max(self.x,  
                other.x)
```

Establish Four Properties of CRDT

- **Associativity**

$$\begin{aligned} & \text{merge}(\text{merge}(a, b), c) \\ = & \max(\max(a.x, b.x), c.x) \\ = & \max(a.x, \max(b.x, c.x)) \\ = & \text{merge}(a, \text{merge}(b, c)) \end{aligned}$$

- **Impotence**

$$\begin{aligned} & \text{merge}(a, a) \\ = & \max(a.x, a.x) \\ = & a.x \\ = & a \end{aligned}$$

- **Commutativity**

$$\begin{aligned} & \text{merge}(a, b) \\ = & \max(a.x, b.x) \\ = & \max(b.x, a.x) \\ = & \text{merge}(b, a) \end{aligned}$$

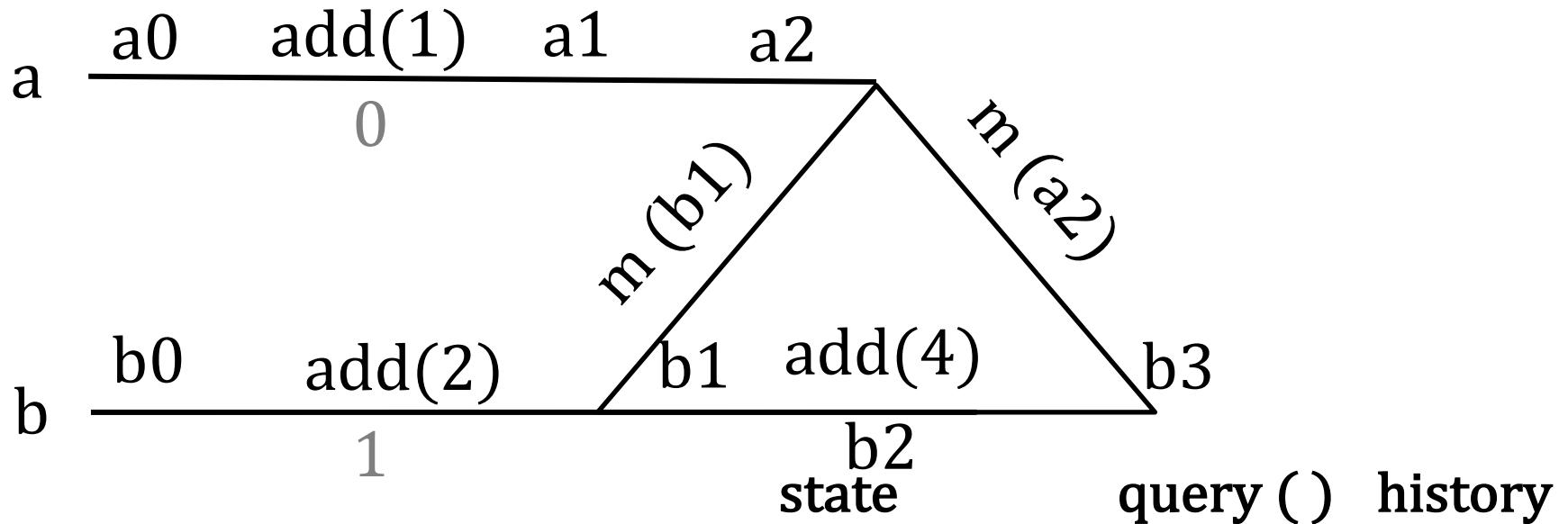
- **Update is increasing**

$$\begin{aligned} & \text{merge}(a, \text{update}(a, x)) \\ = & \max(a.x, a.x + x) \\ = & a.x + x \\ = & \text{update}(a, x) \end{aligned}$$

G-Counter CRDT

Replicated growth-only counter

- Internal state of a G-Counter replicated on n nodes is an n -length array of non-negative integers
- `query` returns sum of every element in n -length array
- `add (x)` when invoked on the i -th server, increments the i -th entry of the n -length array by x
 - E.g., Server 0 increments 0th entry, Server 1 increments 1st entry of array, and so on
- `merge` performs a pairwise maximum of the two arrays

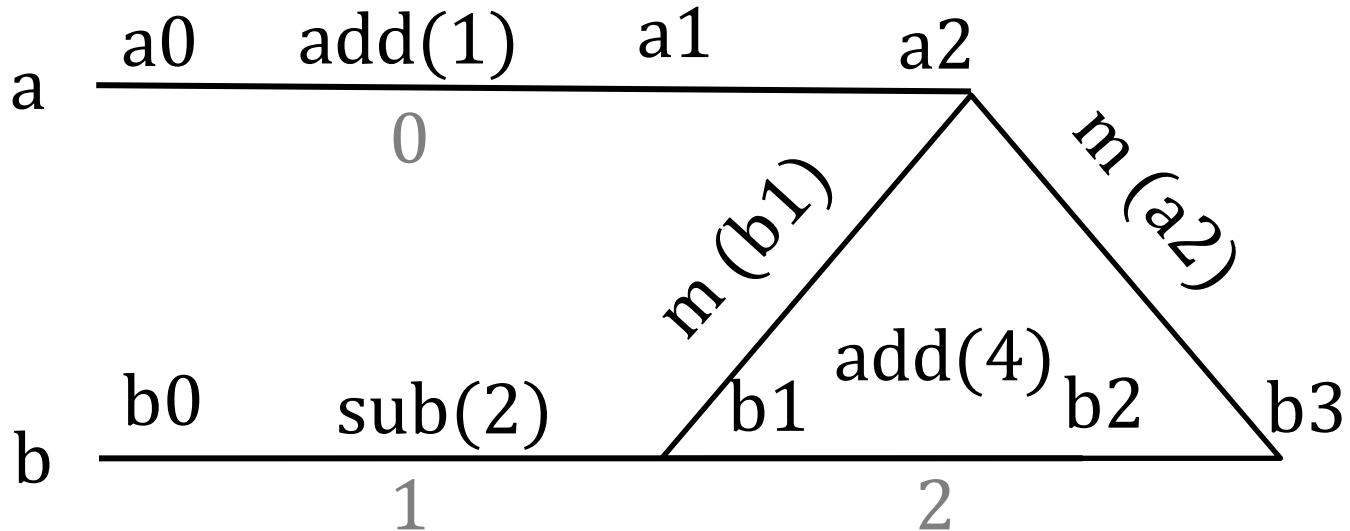


a0	i:0, n:2, xs :[0,0]	0	{}
a1	i:0, n:2, xs :[1,0]	1	{0}
a2	i:0, n:2, xs :[1,2]	3	{0,1}
b0	i:1, n:2, xs :[0,0]	0	{}
b1	i:1, n:2, xs :[0,2]	2	{1}
b2	i:1, n:2, xs :[0,6]	6	{1,2}
b3	i:1, n:2, xs :[1,6]	7	{0,1,2}

PN-Counter CRDT

Replicated counter supporting addition & subtraction

- Internal state of a PN-Counter
 - pair of two G-Counters named p and n .
 - p represents total value added to PN-Counter
 - n represents total value subtracted from PN-Counter.
- query method returns difference $p.\text{query}() - n.\text{query}()$
- add(x) –first of two updates– invokes $p.\text{add}(x)$
- sub(x) –second of updates– invokes $n.\text{add}(x)$
- merge performs a pairwise merge of p and n



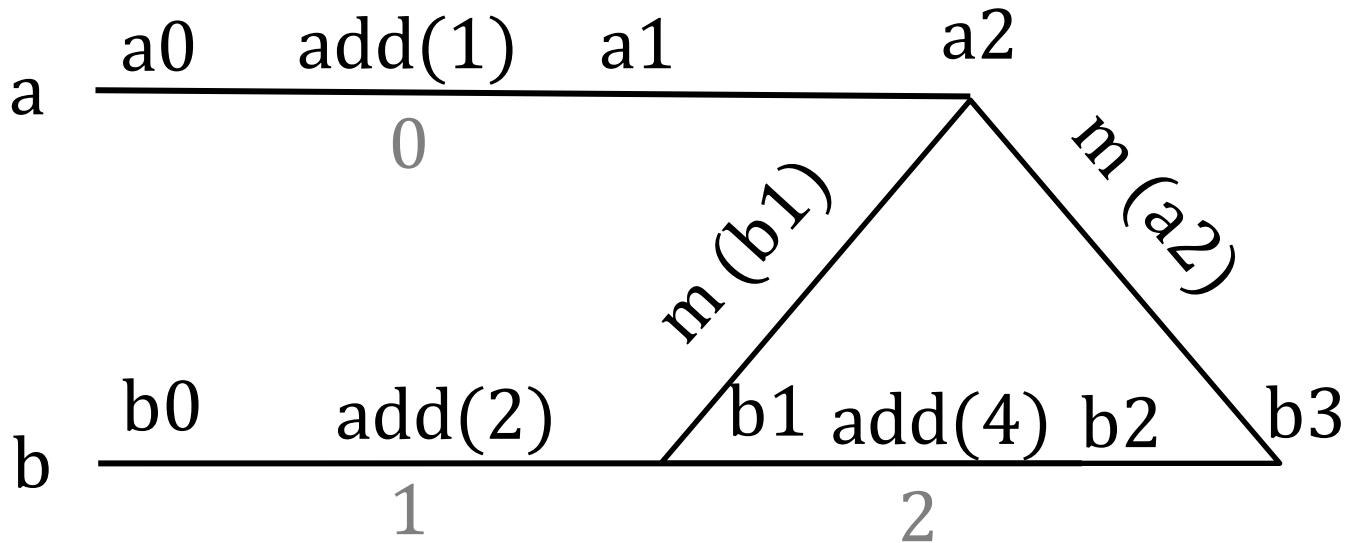
	state	query ()	history
a0	p.xs : 0, n.xs : 0	0	{ }
a1	p.xs : 1, n.xs : 0	1	{0}
a2	p.xs : 1, n.xs : 2	-1	{0,1}
b0	p.xs : 0, n.xs : 0	0	{ }
b1	p.xs : 0, n.xs : 2	-2	{1}
b2	p.xs : 4, n.xs : 2	2	{1,2}
b3	p.xs : 5, n.xs : 2	3	{0,1,2}

G-Set CRDT

Replicated growth-only set

A G-Set CRDT represents a replicated set which can be added to but not removed from

- Internal state of a G-Set is just a set
- `query` returns the set
- `add (x)` adds x to the set
- `merge` performs a set union

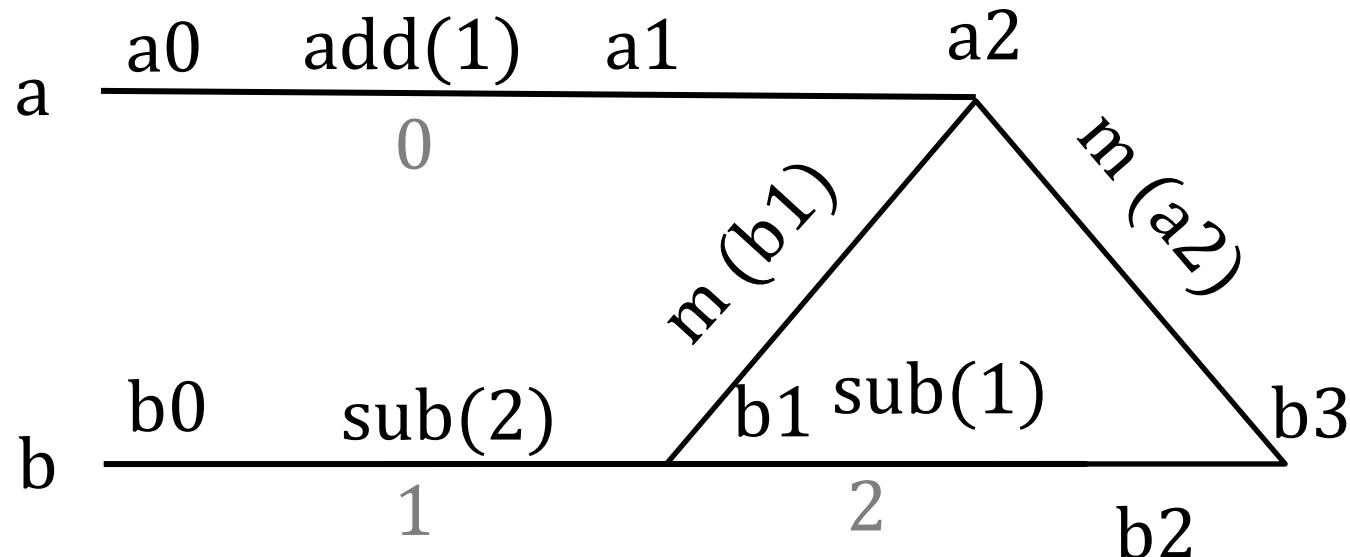


	state	query ()	history
a_0	$\{\}$	0	$\{\}$
a_1	$\{1\}$	$\{1\}$	$\{0\}$
a_2	$\{1, 2\}$	$\{1, 2\}$	$\{0, 1\}$
b_0	$\{\}$	$\{\}$	$\{\}$
b_1	$\{2\}$	$\{2\}$	$\{1\}$
b_2	$\{2, 4\}$	$\{2, 4\}$	$\{1, 2\}$
b_3	$\{1, 2, 4\}$	$\{1, 2, 4\}$	$\{0, 1, 2\}$

2P-Set CRDT

Replicated set supporting addition and subtraction

- Internal state of a 2P-Set is a
 - pair of two G-Sets named a and r
 - a represents set of values added to the 2P-Set
 - r represents set of values removed from the 2P-Set
- query method returns the set difference
 $a.\text{query}() - r.\text{query}()$
- add(x) –first of two updates–invokes $a.\text{add}(x)$.
- sub(x) –second of the two updates–invokes
 $r.\text{add}(x)$
- merge performs a pairwise merge of a and r



	state	query ()	history
a0	$a: \{\}, r:\{\}$	0	$\{\}$
a1	$a: \{1\}, r:\{\}$	$\{1\}$	$\{0\}$
a2	$a: \{1\}, r:\{2\}$	$\{1\}$	$\{0, 1\}$
b0	$a: \{\}, r:\{\}$	$\{\}$	$\{\}$
b1	$a: \{\}, r:\{2\}$	$\{\}$	$\{1\}$
b2	$a: \{\}, r:\{1,2\}$	$\{\}$	$\{1, 2\}$
b3	$a: \{1\}, r:\{1,2\}$	$\{\}$	$\{0, 1, 2\}$