

# Large-scale Data Systems

Fall 2018

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)



# Logistics

This course is given by:

- Theory: Prof. Gilles Louppe ([g.louppe@uliege.be](mailto:g.louppe@uliege.be))
- Exercises and projects: Joeri Hermans ([joeri.hermans@doct.ulg.ac.be](mailto:joeri.hermans@doct.ulg.ac.be))

Feel free to contact any of us for help!

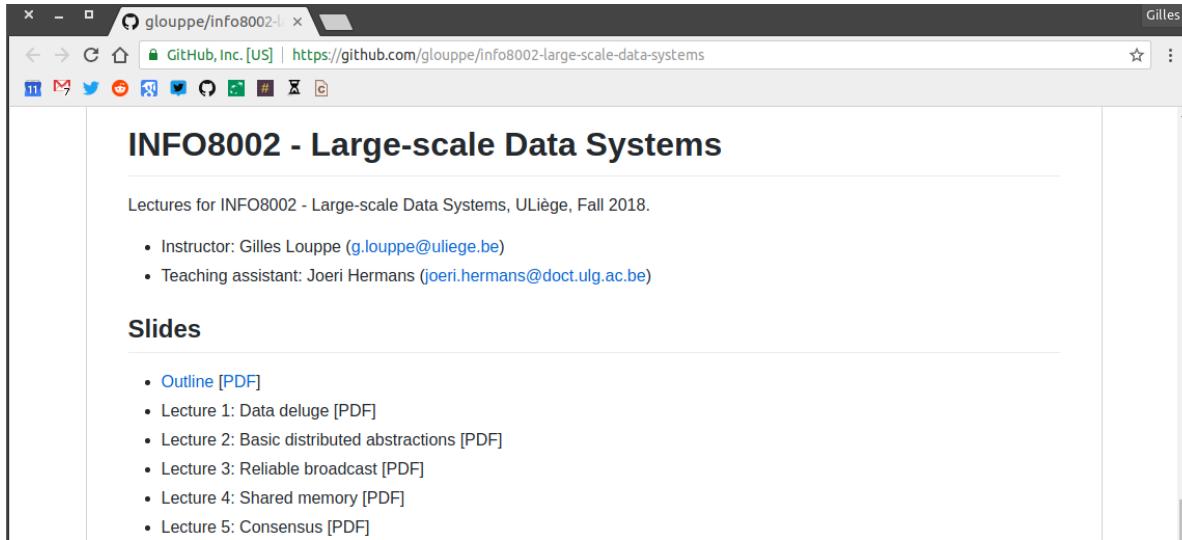


# Slides

Slides are available at [github.com/glouppe/info8002-large-scale-data-systems](https://github.com/glouppe/info8002-large-scale-data-systems).

- In HTML and in PDFs.
- Posted online the day before the lesson.
- Slightly different from previous years.

Slides are partially adapted from [CSE 486/585 Distributed systems](#) (University at Buffalo) and [CS425 Distributed systems](#) (University of Illinois UC).



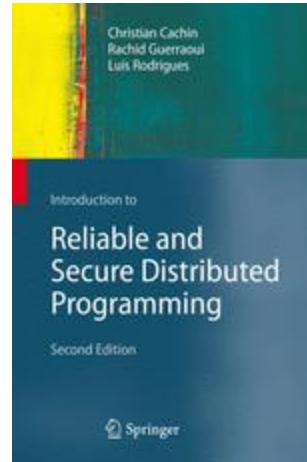
The screenshot shows a web browser window titled "gloppe/info8002-l". The address bar displays "GitHub, Inc. [US] | https://github.com/glouppe/info8002-large-scale-data-systems". The main content area is titled "INFO8002 - Large-scale Data Systems". Below the title, it says "Lectures for INFO8002 - Large-scale Data Systems, ULiège, Fall 2018." A list of contacts follows:

- Instructor: Gilles Louppe ([g.louppe@uliege.be](mailto:g.louppe@uliege.be))
- Teaching assistant: Joeri Hermans ([joeri.hermans@doct.ulg.ac.be](mailto:joeri.hermans@doct.ulg.ac.be))

A section titled "Slides" contains a list of lecture outlines:

- Outline [PDF]
- Lecture 1: Data deluge [PDF]
- Lecture 2: Basic distributed abstractions [PDF]
- Lecture 3: Reliable broadcast [PDF]
- Lecture 4: Shared memory [PDF]
- Lecture 5: Consensus [PDF]

# Textbook



The core content of this course (lectures 2 to 5) is based on the following textbook:

*Christian Cachin, Rachid Guerraoui, Luis Rodrigues, "Introduction to Reliable and Secure Distributed Programming", Springer.*

This textbook is **recommended**, although not required.

# Philosophy

## Solid ground

- Understand the **foundational principles** of distributed systems, on top of which distributed **databases** and **computing** systems are operating.

## Practical

- Exposition to **industrial software**.
- Fun and challenging course project.

## Critical thinking

- Assess the benefits and disadvantages of data systems.
- No hype!

# Lectures

- Theoretical lectures
- Exercise sessions

# Outline

- Lecture 1: Introduction
- Lecture 2: Basic distributed abstractions
- Lecture 3: Reliable broadcast
- Lecture 4: Shared memory
- Lecture 5: Consensus
- Lecture 6: Blockchain
- Lecture 7: Cloud computing
- Lecture 8: Distributed hash tables
- Lecture 9: Distributed file systems

# Projects

## Reading assignment

Read, summarize and criticize a major scientific paper in Large-Scale Data Systems.  
(Paper to be announced later.)

### MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

*Google, Inc.*

#### Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

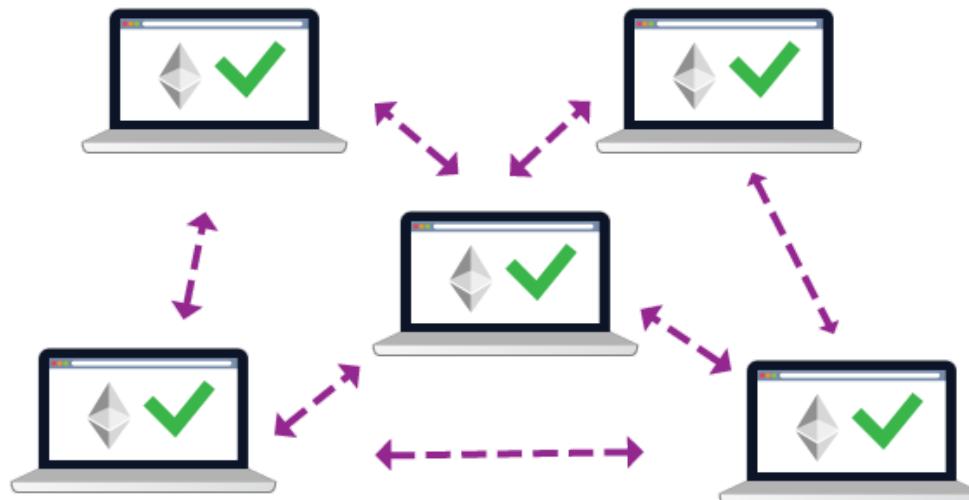
Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then

## Programming project

Implement a secure and distributed service using **Blockchain** technology.



# Evaluation

- Oral exam (60%)
- Reading assignment (10%)
- Programming project (30%)

Projects are **mandatory** for presenting the exam.



# Large-scale Data Systems

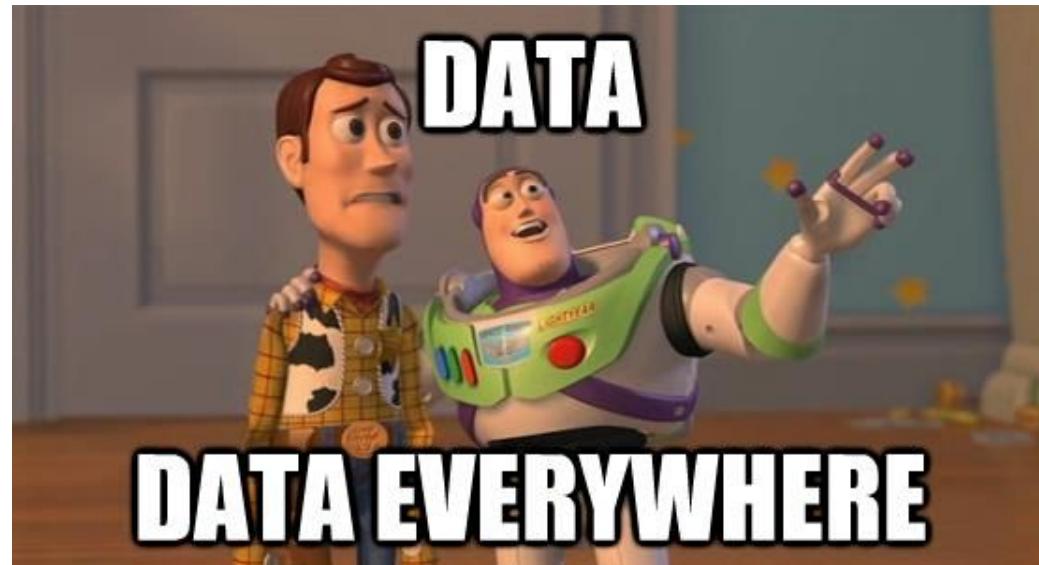
Lecture 1: Introduction

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)



# **The data science era**

Big data? Data science?



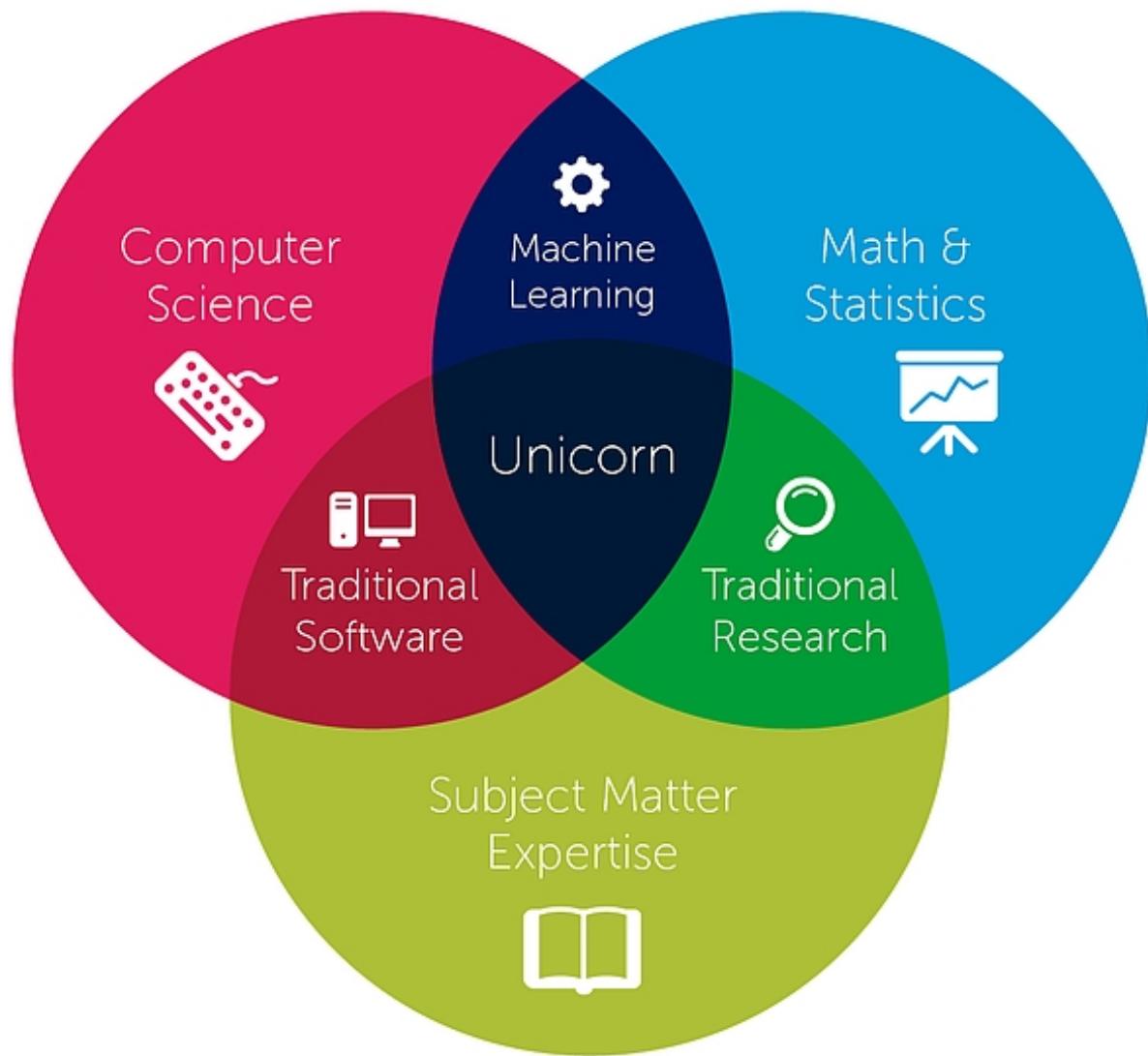
hype **vs.** business **vs.** science

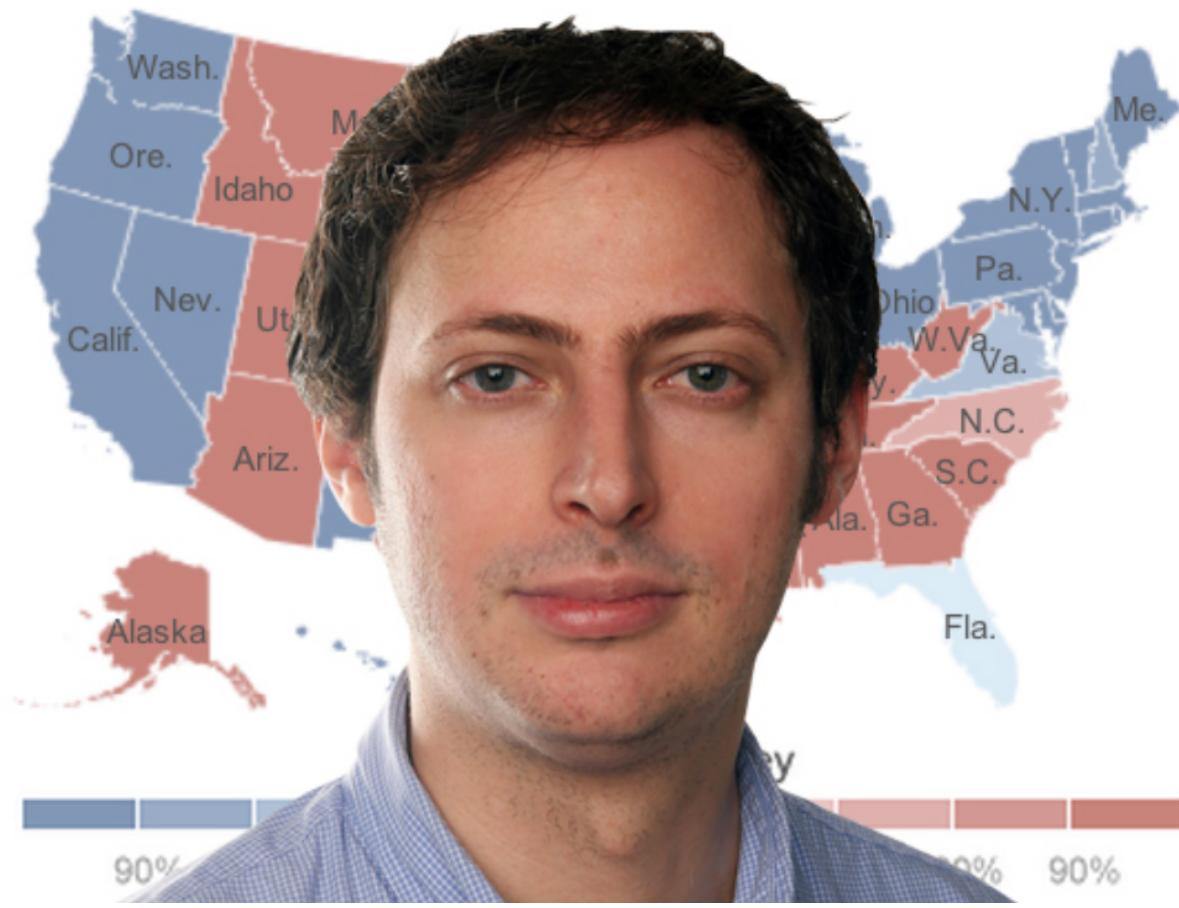
*"A data scientist is someone who knows more statistics than a computer scientist and more computer science than a statistician."*

Josh Blumenstock

*"Data scientist = statistician + programmer + coach + storyteller + artist"*

Shlomo Aragmon





*Nate Silver*

# FiveThirtyEight Forecast

Updated 12:27 AM ET on Oct. 1



Barack Obama

320.1

+10.7 since Sept. 23

President  
Now-cast

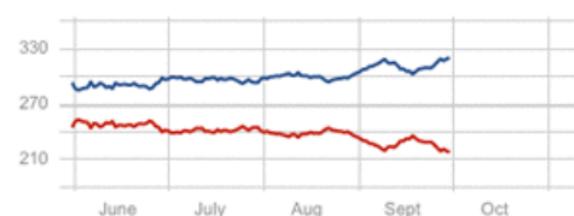
Senate  
Nov. 6 Forecast

Mitt Romney

217.9

-10.7 since Sept. 23

Electoral  
vote



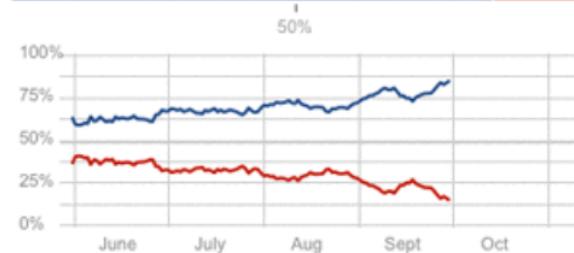
85.1%

+7.5 since Sept. 23

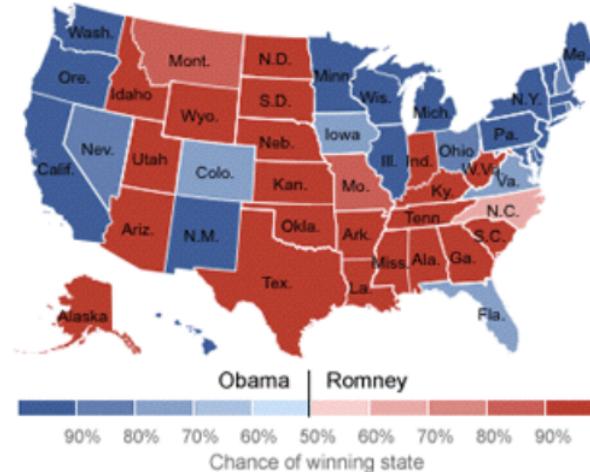
Chance of  
Winning

14.9%

-7.5 since Sept. 23

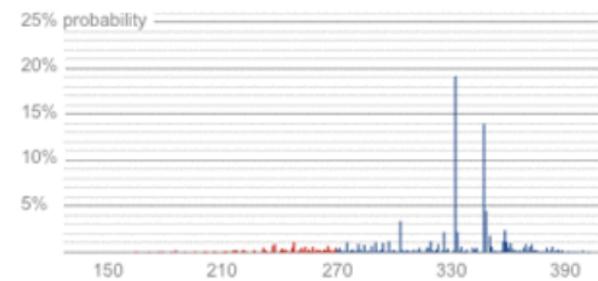


## State-by-State Probabilities

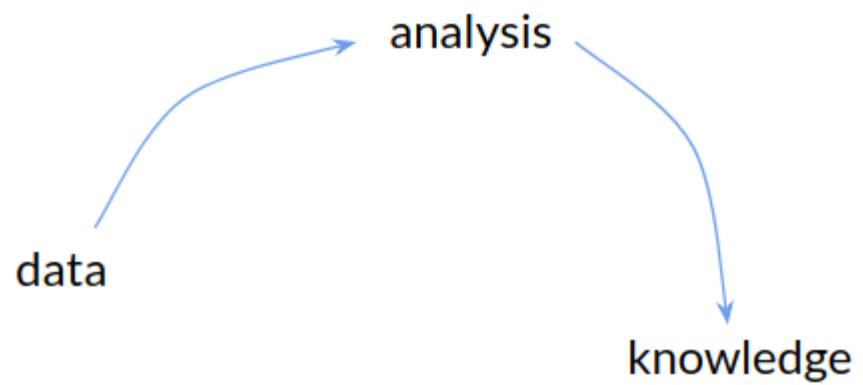


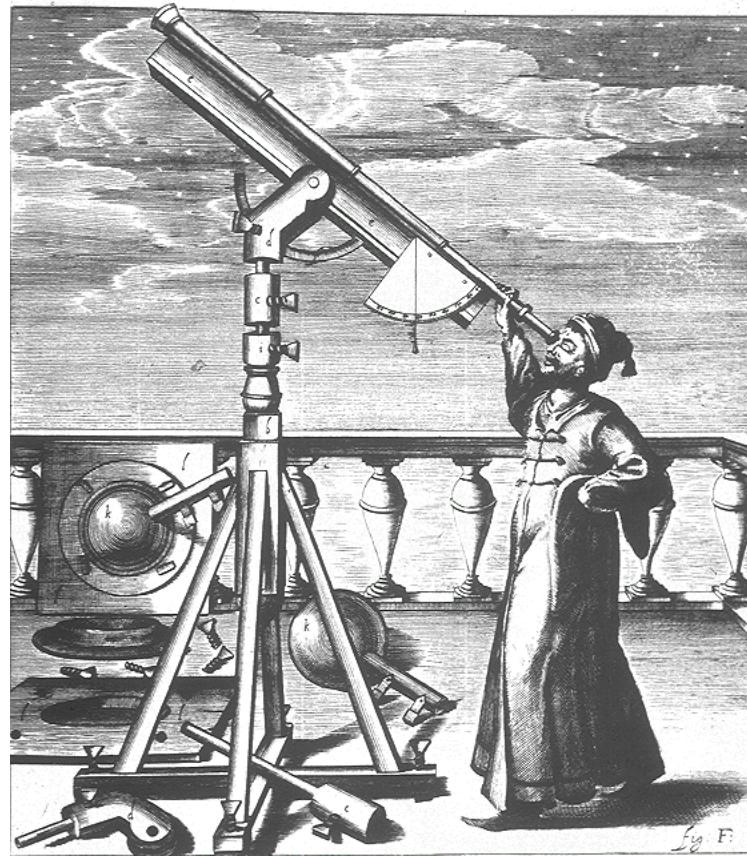
## Electoral Vote Distribution

The probability that President Obama receives a given number of Electoral College votes.



*"Nate Silver won the election" - Harvard Business Review*





*Haven't we be doing data analysis forever?*



*"Every two days now we create as much information as we did from the dawn of civilization up until 2003, according to Schmidt. That's something like five exabytes of data, he says.*

*Let me repeat that: we create as much information in two days now as we did from the dawn of man through 2003.*

Eric Schmidt, 2010.

# 1 Zettabyte (ZB) = 1 Trillion Gigabytes (GB)

We face an overwhelming amount of data in every industry

**>2.5 PB**

of customer data  
stored by Walmart  
**every hour.**

**292 exabytes**

of mobile traffic by  
2019, up from 30  
**exabytes** in 2014.

**1 TB**

of data produced  
by a cancer patient  
**every day.**

2010

2018

2025

Today

44 Zettabytes

Gap

Sensors & Devices

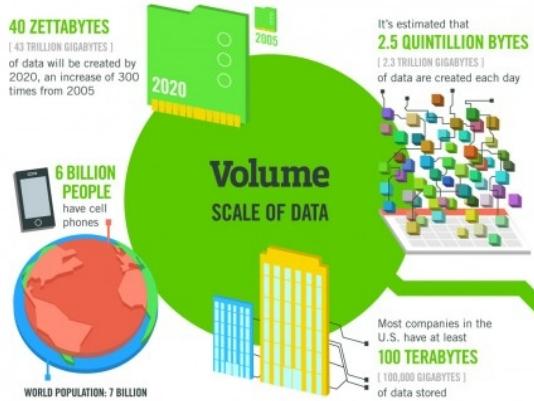
Images/Multimedia

Text

Enterprise Data

Traditional

Source: © 2018 Dvmobile Inc. All Rights Reserved



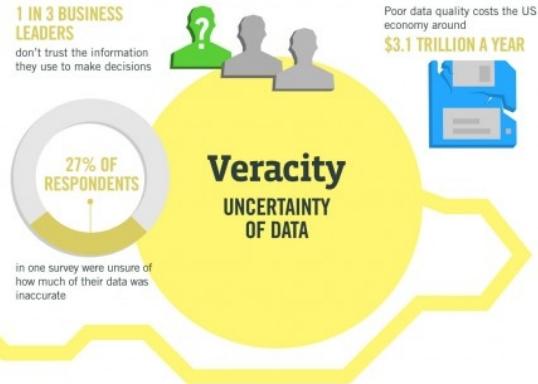
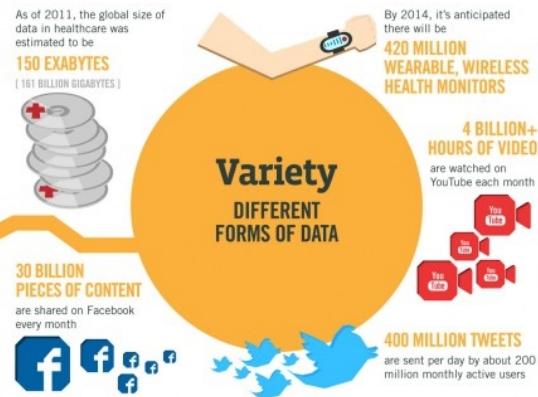
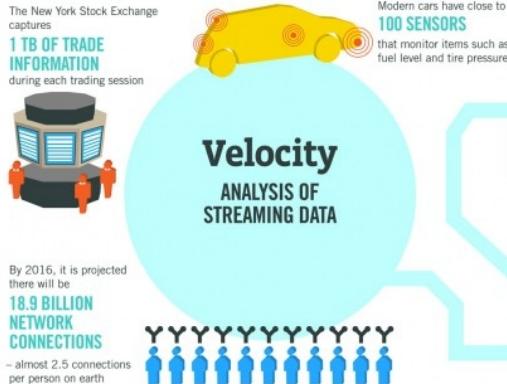
# The FOUR V's of Big Data

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety, and Veracity**.

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015 **4.4 MILLION IT JOBS** will be created globally to support big data, with 1.9 million in the United States.



Sources: McKinsey Global Institute, Twitter, Cisco, Gariner, EMC, SAS, IBM, MPTEC, QAS

IBM

Actually, **none of that is really new**... What is new is:

- our ability to **store machine generated data**, at unprecedented scale and rate.
- the broad understanding that **we cannot just manually get value out of data**.



The  
**F O U R T H**  
**P A R A D I G M**

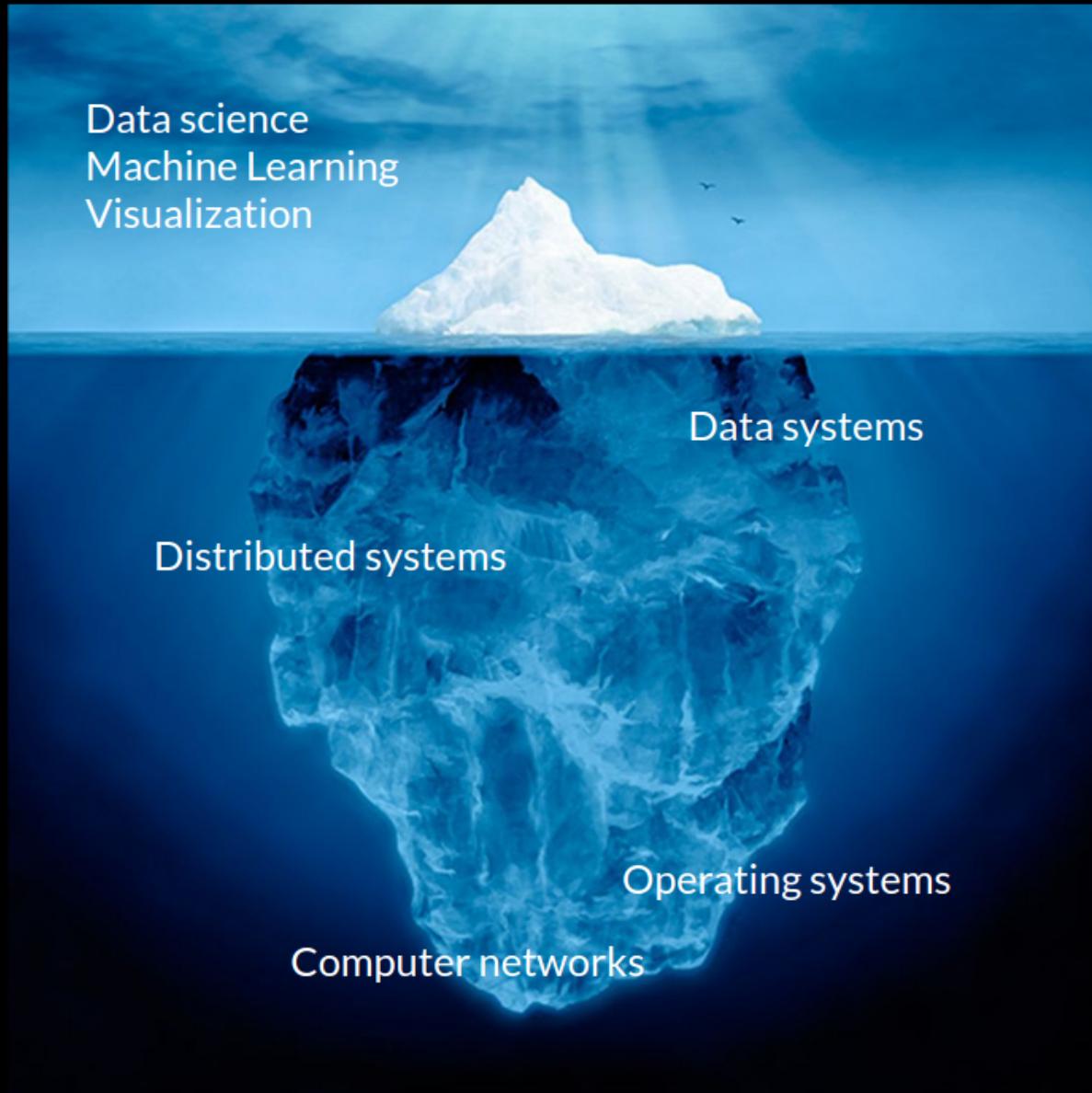
DATA-INTENSIVE SCIENTIFIC DISCOVERY

EDITED BY TONY HEY, STEWART TANSLEY, AND KRISTIN TOLLE

*"Increasingly, scientific breakthroughs will be powered by advanced computing capabilities that help researchers manipulate and explore massive datasets.*

*The speed at which any given scientific discipline advances will depend on how well its researchers collaborate with one another, and with technologists, in areas of eScience such as databases, workflow management, visualization, and cloud computing technologies."*

# Data systems



# Operating systems

Can you name examples of **operating systems**?

# Operating systems

Can you name examples of **operating systems**?

- Android
- Chrome OS
- FreeBSD
- iOS
- macOS
- OS/2
- RISC OS
- Solaris
- Windows
- ...

## **Definition**

The low-level software which handles the interface to peripheral hardware, schedules tasks, allocates storage, and presents a default interface to the user when no application program is running.

# Distributed systems

Can you name examples of **distributed systems**?

# Distributed systems

Can you name examples of **distributed systems**?

- A client/server system
- The web
- Wireless networks
- Telephone networks
- DNS
- Massively multiplayer online games
- Distributed databases
- BitTorrent (peer-to-peer overlays)
- A cloud, e.g. Amazon EC2/S3, Microsoft Azure
- A data center, e.g. a Google data center, AWS
- The bitcoin network

## Definition

A distributed system is a collection of entities with a common goal, each of which is **autonomous, programmable, asynchronous** and **failure-prone**, and which communicate through an **unreliable** communication medium.

- **Entity**: a process on a device.
- **Communication medium**: Wired or wireless network.

A distributed system appears to its users as a **single coherent** system.

# Internet



*What are the entities? What is the communication medium?*

## Data center



*What are the entities? What is the communication medium?*

# Why study distributed systems?

- Distributed systems are **everywhere**:
  - Internet
  - WWW
  - Mobile devices
  - Internet of Things
- **Technical** importance:
  - Improve **scalability**
    - Adding computational resources to a system is an easy way to scale its performance to many users.
  - Improve **reliability**
    - We want high availability and durability of the system.

- Distributed systems are **difficult** to build.
  - **Scale:** hundreds or thousands of machines.
    - Google: 4k-machine MapReduce cluster
    - Facebook: 60k machines providing the service
  - **Fault tolerance:** machines and networks do fail!
    - 50 machine failures out of 20k machine cluster per day (reported by Yahoo!)
    - 1 disk failure out of 16k disks every 6 hours (reported by Google)
  - **Concurrency:**
    - Nodes execute in parallel
    - Messages travel asynchronously
  - **Consistency:**
    - Distributed systems need to ensure user guarantees about the data they store.
    - E.g., all read operations return the same value, no matter where it is stored.
- But only a few **core problems** reoccur.

# Teaser: Two Generals' Problem

Two generals need to coordinate an attack.

- They must **agree** on time to attack.
- They will win only if they attack **simultaneously**.
- They communicate through **messengers**.
- Messengers may be **killed** on their way.



Let's try to solve the problem for generals  $g_1$  and  $g_2$ .

- $g_1$  sends time of attack to  $g_2$ .
- Problem: how to ensure  $g_2$  received the message?
- Solution: let  $g_2$  acknowledge receipt of message.
- Problem: how to ensure  $g_1$  received the acknowledgment?
- Solution: let  $g_1$  acknowledge receipt of acknowledgment.
- ...

This problem is **impossible** to solve!

(Unless we make additional assumptions)

- Applicability to distributed systems:
  - Two nodes need to **agree** on a **value**.
  - They communicate by **messages** using an **unreliable channel**.
- **Agreement** is one of the core problems of distributed systems.

# Data systems

Can you name examples of **data systems**?

# Data systems

Can you name examples of **data systems**?

- A database
- A file system
- A ledger
- Search engines
- Data flow frameworks
- Social networks
- ...

## **Definition**

In this course, data systems will broadly refer to any kind of computer systems, distributed or not, that can be used to store, retrieve, organize or process data.

Our main focus will be on data systems for data science purposes.

BIG DATA & AI LANDSCAPE 2018



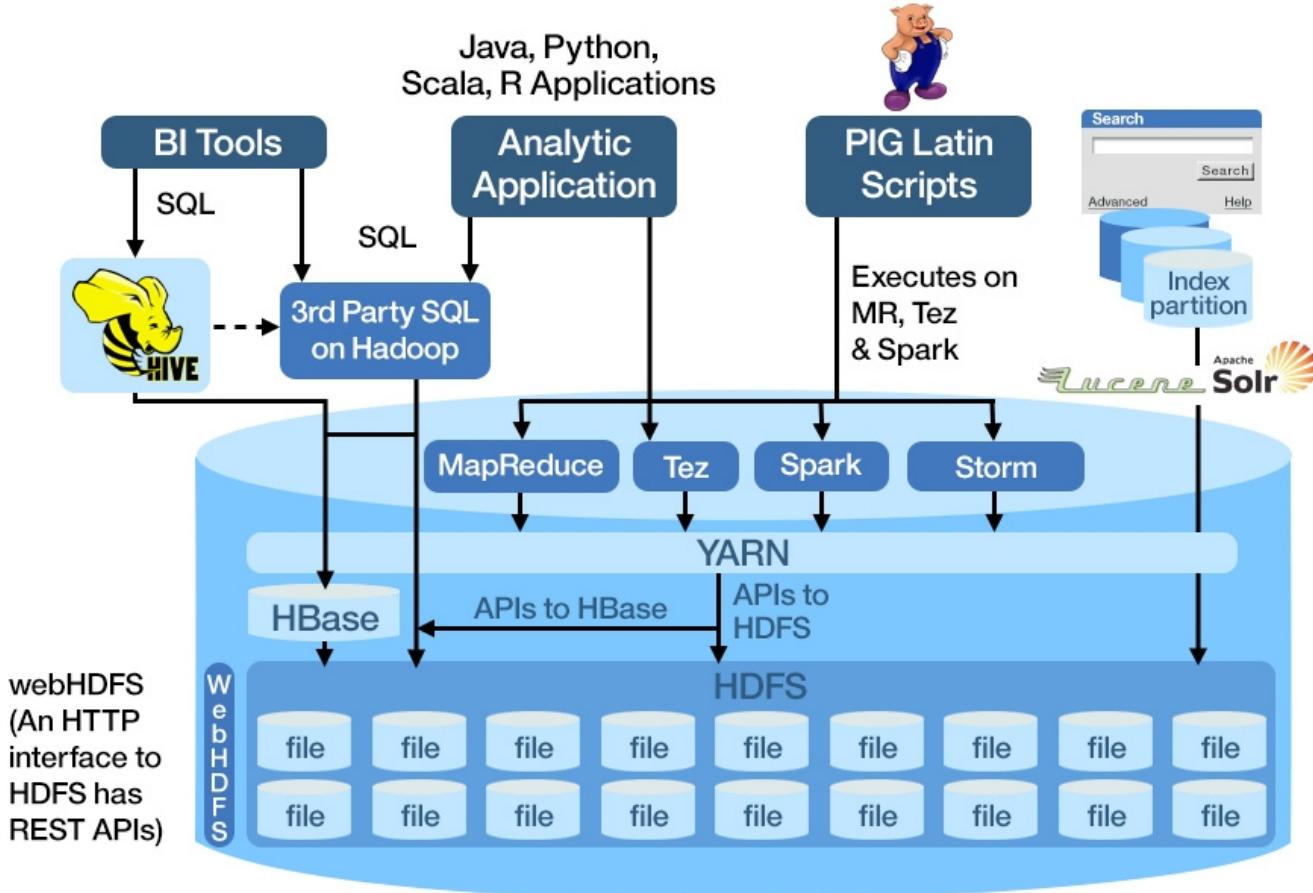
Final 2018 version, updated 07/15/2018

© Matt Turck (@mattturck), Demi Obavomi (@demi\_ obavomi), & FirstMark (@firstmarkcap)

[mattturck.com/bigdata2018](http://mattturck.com/bigdata2018)

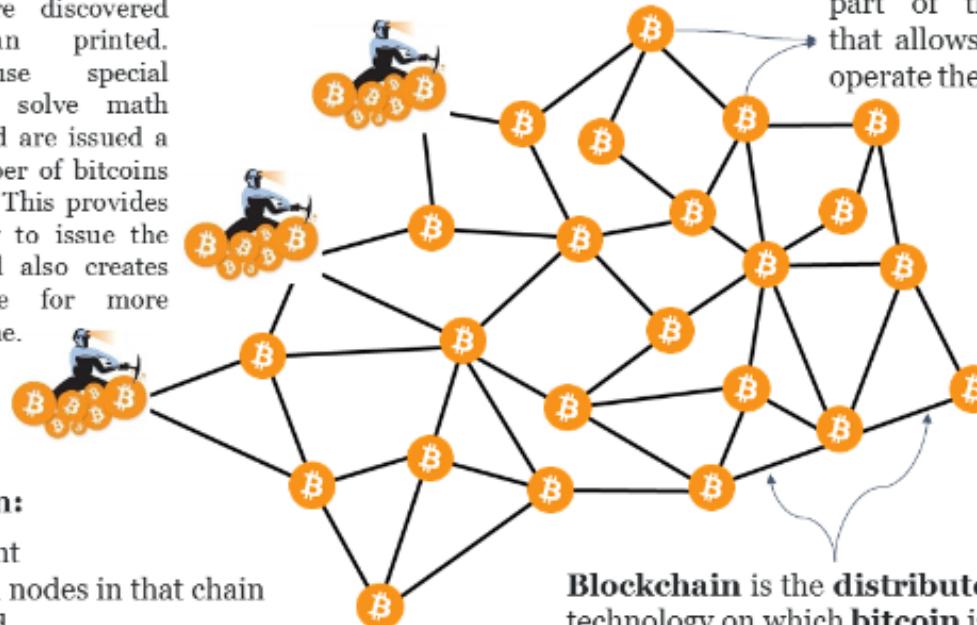
FIRSTMARK  
EARLY STAGE VENTURE CAPITAL

# The Hadoop ecosystem



## A distributed ledger

**Bitcoins** are discovered rather than printed. **Miners** use special software to solve math problems and are issued a certain number of bitcoins in exchange. This provides a smart way to issue the currency and also creates an incentive for more people to mine.



# Outline

# Fundamentals of distributed systems

Understand the **foundational principles** required for the **design, implementation** and **maintenance** of distributed systems.

- Communications
- Failures
- Consistency
- Concurrency
- Consensus

## Communications

- How do you talk to another machine?
  - Reliable networking.
- How do you talk to multiple machines at once, with ordering guarantees?
  - Multicast, Gossiping.

## Failures and consistency

- How do you know if a machine has failed?
  - Failure detection.
- How do you program your system to operate continually even under failure?
  - Gossiping, replication.
- What if some machines do not cooperate?
  - Byzantine fault tolerance.

## Concurrency

- How do you control access to shared resources?
  - Distributed mutual exclusion, distributed transactions, etc.

## Consensus

- How do multiple machines reach an agreement?
  - Time and synchronization, global states, leader election, Paxos, proof of work, blockchain.
- **Bad news:** it is impossible!
  - The impossibility of consensus for asynchronous systems.

# Case studies

From these building blocks, understand how to build and architecture data systems for large volumes or data or for data science purposes.

## Distributed storage

- How do you locate where things are and access them?
  - Distributed file systems
  - Key-value stores
- How do you record and share sensitive data?
  - Proof of work, blockchain

## Distributed computing for data science

- What are the distributed computing systems for data science?
  - Map Reduce (Hadoop)
  - Computational graph systems (Spark, Dask)
- Is distributed computing always necessary?



# References

- Hey, Tony, Stewart Tansley, and Kristin M. Tolle. The fourth paradigm: data-intensive scientific discovery. Vol. 1. Redmond, WA: Microsoft research, 2009.
- Kersten, Martin L., et al. "The researcher's guide to the data deluge: Querying a scientific database in just a few seconds." PVLDB Challenges and Visions 3.3 (2011).
- Silver, Nate. The signal and the noise: the art and science of prediction. Penguin UK, 2012.

# Large-scale Data Systems

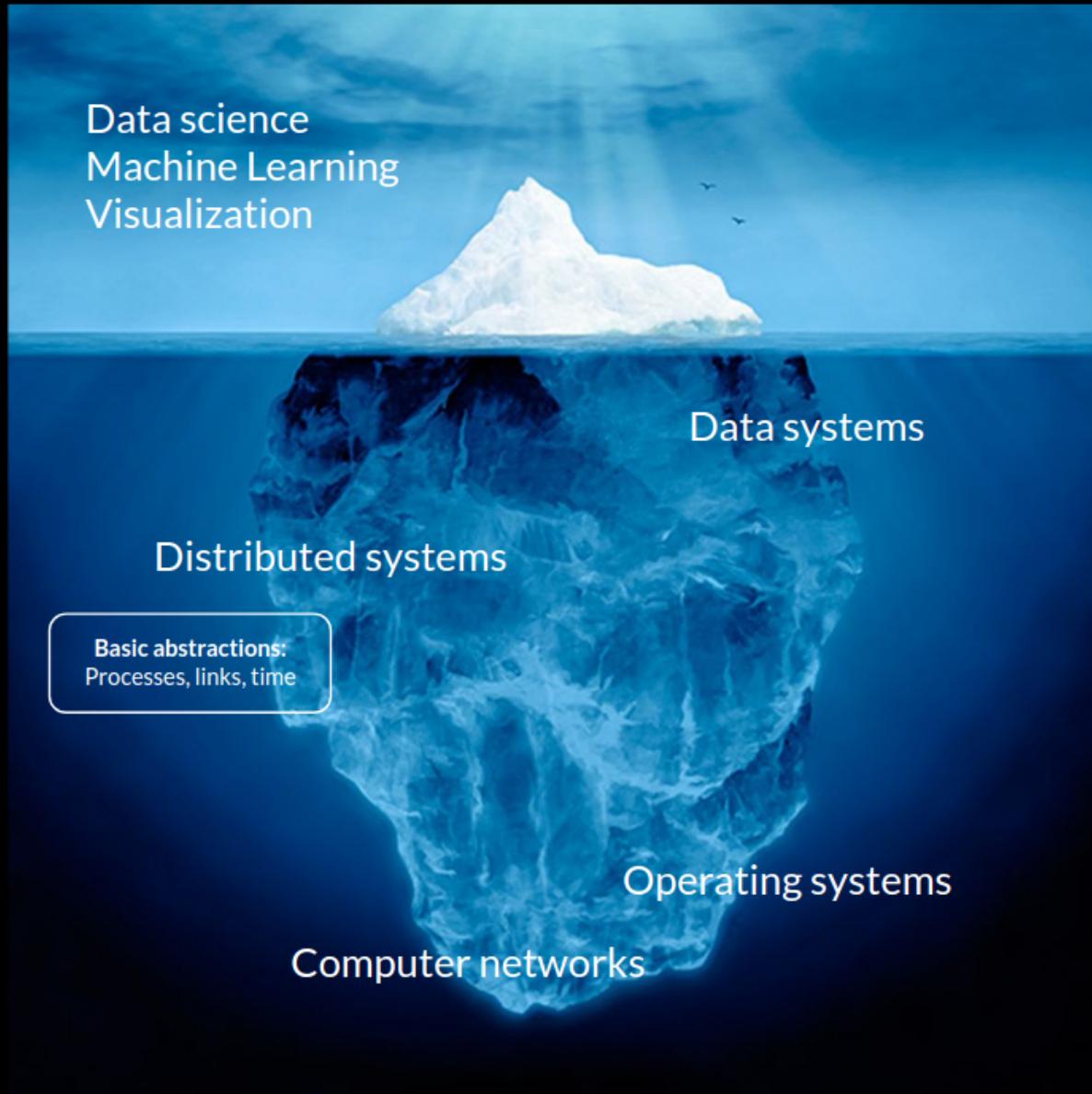
Lecture 2: Basic distributed abstractions

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)



# Today

- Define **basic abstractions** that capture the fundamental characteristics of distributed systems:
  - **Process** abstractions
  - **Link** abstractions
  - **Timing** abstractions
- A **distributed system model** = a combination of the three categories of abstractions.



# Why distributed abstractions?

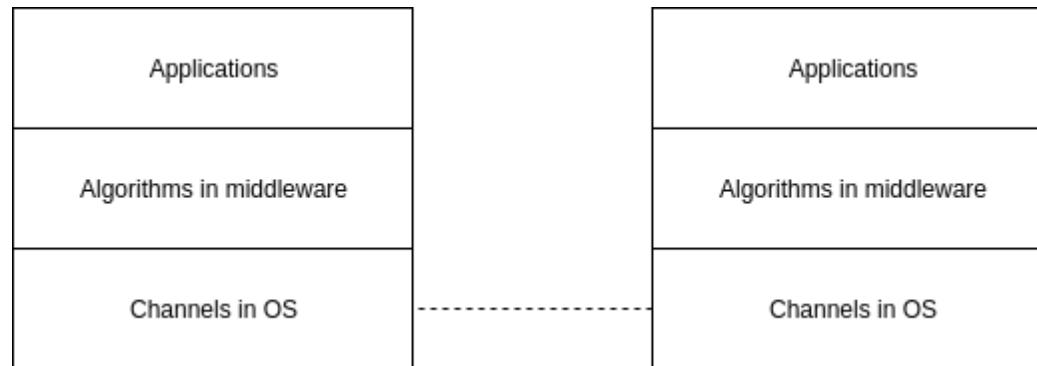
Reliable distributed applications need underlying services stronger than transport protocols (e.g., TCP or UDP).



*"All problems in computer science can be solved by another level of indirection"* - David Wheeler.

## Distributed abstractions

- Core of any distributed system is a **set of distributed algorithms**.
- Implemented as a middleware between network (OS) and the application.



# Network protocols are not enough

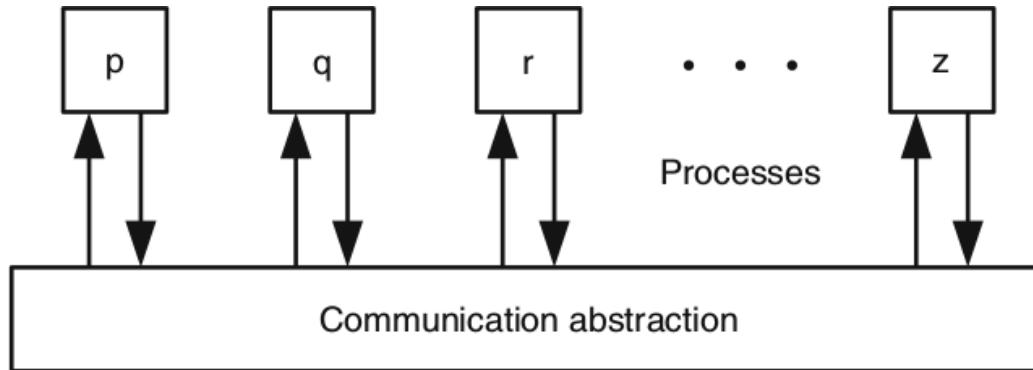
- Communication
  - Reliability guarantees (e.g. with TCP) are only offered for **one-to-one** communication (client-server).
  - How to do **group communication**?
- High-level services
  - Sometimes one-to-many communication is not enough.
  - Need reliable **higher-level services**.
- Strategy: build complex distributed systems in a **bottom-up** fashion, from simpler ones.

**High level services:**  
shared memory  
consensus  
atomic commit  
group membership

**Group communication:**  
reliable broadcast  
causal order broadcast  
total order broadcast  
terminating reliable broadcast

# Distributed computation

# Distributed algorithms



- A **distributed algorithm** is a distributed collection  $\Pi = \{p, q, r, \dots\}$  of  $N$  processes implemented by **identical** automata.
- The automaton at a process regulates the way the process executes its computation steps.
- Processes jointly implement the application.
  - Need for **coordination**.

# Event-driven programming

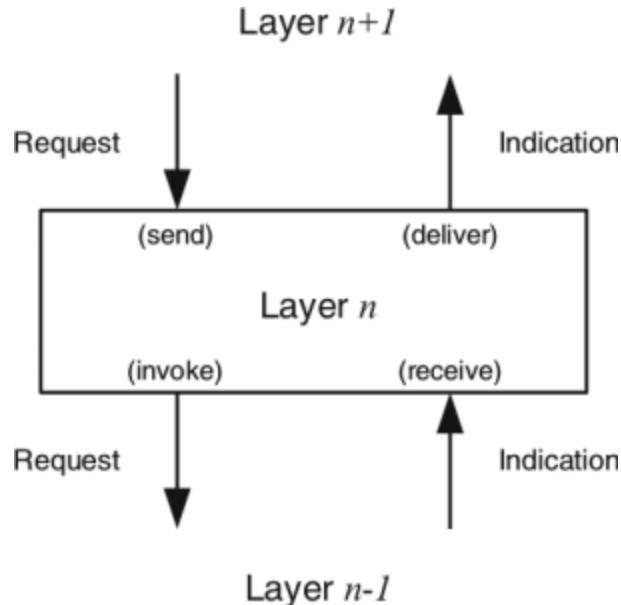
- Every process consists of **modules** or **components**.
  - Modules may exist in multiple instances.
  - Every instance has a unique identifier and is characterized by a set of properties.
- Asynchronous **events** represent **communication** or **control flow** between components.
  - Each component is constructed as a state-machine whose transitions are triggered by the reception of events.
  - Events carry information (sender, message, etc)
- Reactive programming model:

```
upon event < co1, Event1 | att11, att12, ... > do
    do something;
    trigger < co2, Event2 | att21, att22, ... >;           // send some event

upon event < co1, Event3 | att31, att32, ... > do
    do something else;
    trigger < co2, Event4 | att41, att42, ... >;           // send some other event
```

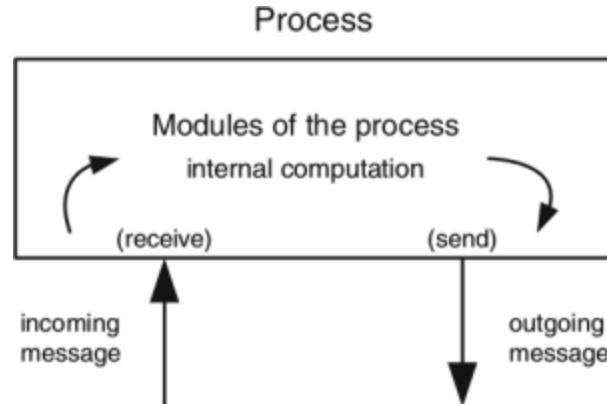
- Effectively, a distributed algorithm is described by a set of event handlers.

## Layered modular architecture



- Components can be composed locally to build software stacks.
  - The top of the stack is the **application layer**.
  - The bottom of the stack the **transport** or **network** layer.
- Distributed programming abstraction layers are typically in the middle.
- We assume that every process executes the code triggered by events in a mutually exclusive way, without concurrently processing  $\geq 2$  events.

# Execution



- The **execution** of a distributed algorithm is a **sequence of steps** executed by its processes.
- A **process step** consists in
  - **receiving** a message from another process,
  - **executing** a local computation,
  - **sending** a message to some process.
- Local messages between components are treated as local computation.
- We assume **deterministic** process steps (with respect to the message received and the local state prior to executing a step).

# Example: Job handler

---

## Module 1.1: Interface and properties of a job handler

---

**Module:**

**Name:** JobHandler, instance  $jh$ .

**Events:**

**Request:**  $\langle jh, Submit \mid job \rangle$ : Requests a job to be processed.

**Indication:**  $\langle jh, Confirm \mid job \rangle$ : Confirms that the given job has been (or will be) processed.

**Properties:**

**JH1:** *Guaranteed response:* Every submitted job is eventually confirmed.

---

**Algorithm 1.1:** Synchronous Job Handler

---

**Implements:**

JobHandler, **instance** *jh*.

**upon event**  $\langle jh, Submit \mid job \rangle$  **do**  
    process(*job*);  
    **trigger**  $\langle jh, Confirm \mid job \rangle$ ;

---

**Algorithm 1.2:** Asynchronous Job Handler

---

**Implements:**

JobHandler, instance  $jh$ .

```
upon event ⟨  $jh$ , Init ⟩ do
     $buffer := \emptyset$ ;  
  
upon event ⟨  $jh$ , Submit |  $job$  ⟩ do
     $buffer := buffer \cup \{job\}$ ;
    trigger ⟨  $jh$ , Confirm |  $job$  ⟩;  
  
upon  $buffer \neq \emptyset$  do
     $job := selectjob(buffer)$ ;
    process( $job$ );
     $buffer := buffer \setminus \{job\}$ ;
```

---

**Module 1.2:** Interface and properties of a job transformation and processing abstraction

---

**Module:**

**Name:** TransformationHandler, instance  $th$ .

**Events:**

**Request:**  $\langle th, Submit \mid job \rangle$ : Submits a job for transformation and for processing.

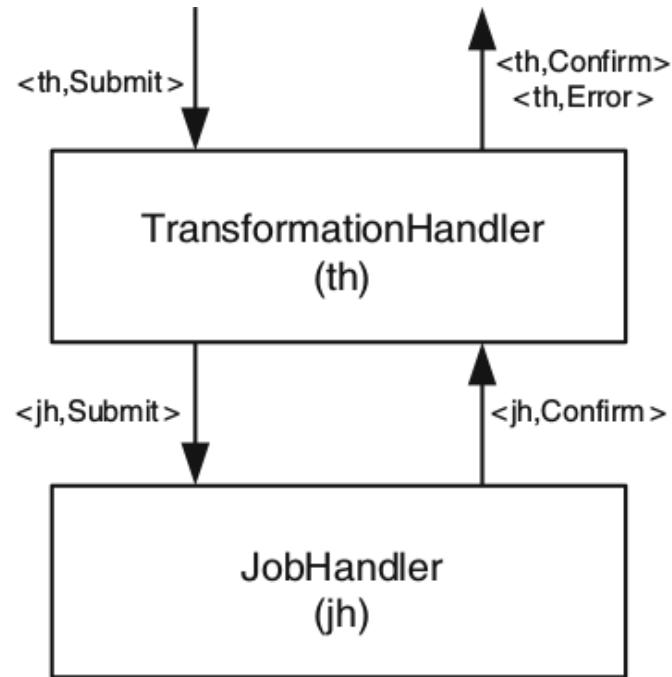
**Indication:**  $\langle th, Confirm \mid job \rangle$ : Confirms that the given job has been (or will be) transformed and processed.

**Indication:**  $\langle th, Error \mid job \rangle$ : Indicates that the transformation of the given job failed.

**Properties:**

**TH1:** *Guaranteed response:* Every submitted job is eventually confirmed or its transformation fails.

**TH2:** *Soundness:* A submitted job whose transformation fails is not processed.



**Figure 1.3:** A stack of job-transformation and job-handler modules

---

**Algorithm 1.3:** Job-Transformation by Buffering

---

**Implements:**

TransformationHandler, instance *th*.

**Uses:**

JobHandler, instance *jh*.

```
upon event < th, Init > do
    top := 1;
    bottom := 1;
    handling := FALSE;
    buffer := [⊥]M;

upon event < th, Submit | job > do
    if bottom + M = top then
        trigger < th, Error | job >;
    else
        buffer[top mod M + 1] := job;
        top := top + 1;
        trigger < th, Confirm | job >;

upon bottom < top ∧ handling = FALSE do
    job := buffer[bottom mod M + 1];
    bottom := bottom + 1;
    handling := TRUE;
    trigger < jh, Submit | job >

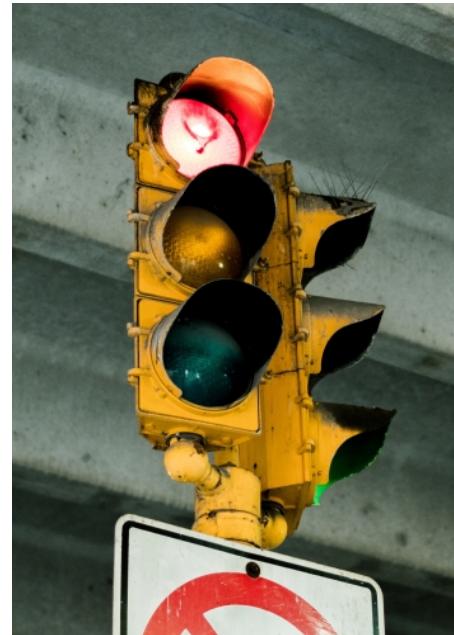
upon event < jh, Confirm | job > do
    handling := FALSE;
```

# Liveness and safety

- Implementing a distributed programming abstraction requires satisfying its **correctness** in all possible executions of the algorithm.
  - i.e., in all possible interleaving of steps.
- Correctness of an abstraction is expressed in terms of **liveness** and **safety** properties.
  - **Safety:** properties that state that nothing bad ever happens.
    - A safety property is a property such that, whenever it is violated in some execution  $E$  of an algorithm, there is a prefix  $E'$  of  $E$  such that the property will be violated in any extension of  $E'$ .
  - **Liveness:** properties that state something good eventually happens.
    - A liveness property is a property such that for any prefix  $E'$  of  $E$ , there exists an extension of  $E'$  for which the property is satisfied.
- Any property can be expressed as the conjunction of safety property and a liveness property.

## Example 1: Traffic lights at an intersection

- Safety: only one direction should have a green light.
- Liveness: every direction should eventually get a green light.



## **Example 2: TCP**

- Safety: messages are not duplicated and received in the order they were sent.
- Liveness: messages are not lost.
  - i.e., messages are eventually delivered.

# Assumptions

- In our abstraction of a distributed system, we need to specify the **assumptions** needed for the algorithm to be **correct**.
- A distributed system model includes assumptions on:
  - **failure** behavior of processes and channels
  - **timing** behavior of processes and channels

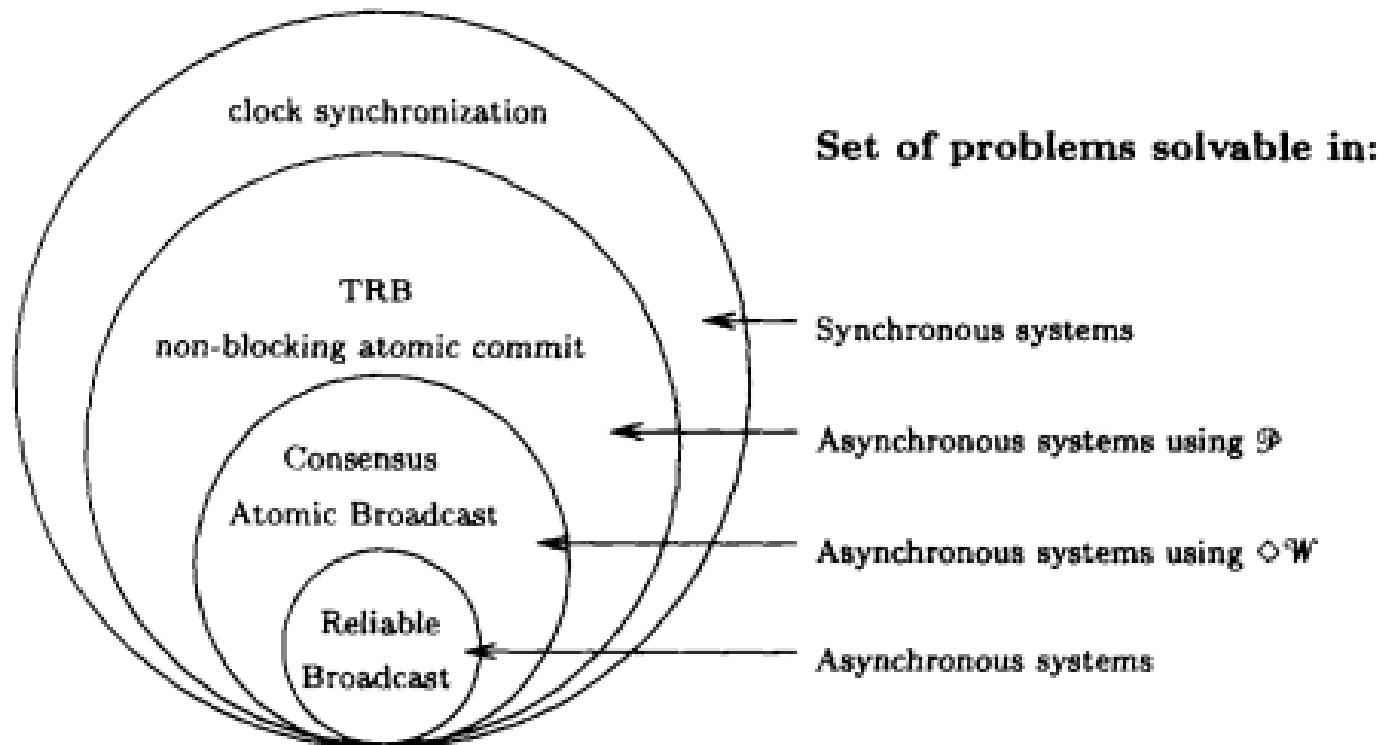


FIG. 9. Problem solvability in different distributed computing models.

*Together, these assumptions define sets of solvable problems.*

# Process abstractions

# Process failures

- Processes may **fail** in four different ways:
  - Crash-stop
  - Omissions
  - Crash-recovery
  - Byzantine / arbitrary
- Processes that do not fail in an execution are **correct**.

## Crash-stop failures

- A process **stops taking steps.**
  - Not sending messages.
  - Not receiving messages.
- We assume the crash-stop process abstraction **by default.**
  - Hence, do not recover.
  - [Q] Does this mean that processes are not allowed to recover?

## Omission failures

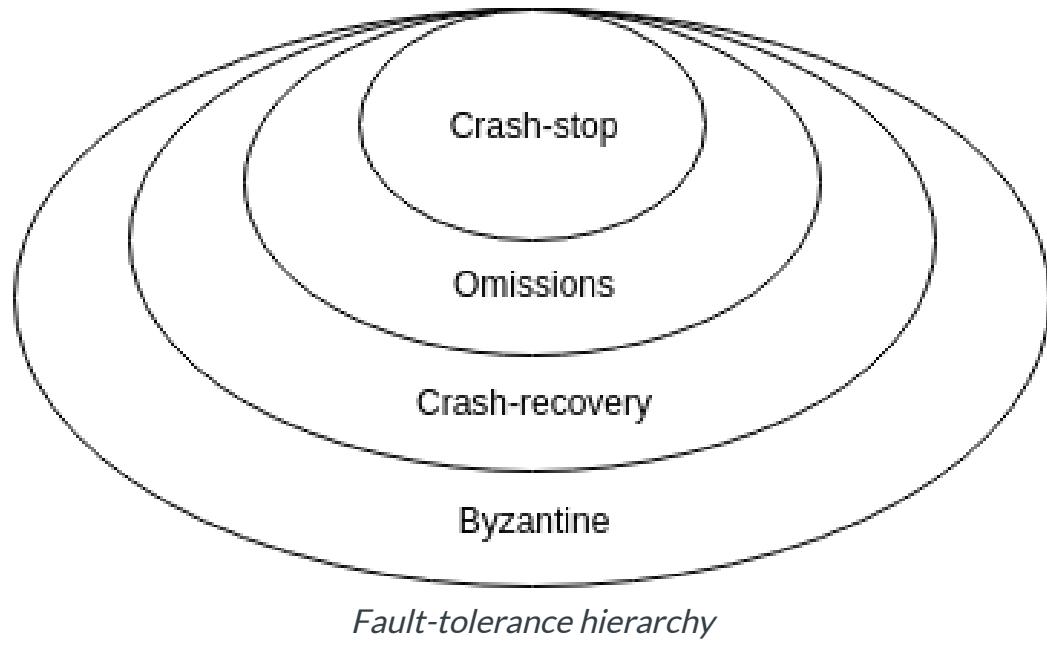
- Process **omits** sending or receiving messages.
  - **Send omission:** A process omits to send a message it has to send according to its algorithm.
  - **Receive omission:** A process fails to receive a message that was sent to it.
- Often, omission failures are due to **buffer overflows**.
- With omission failures, a process deviates from its algorithm by dropping messages that should have been exchanged with other processes.

## Crash-recovery failures

- A process **might crash**.
  - It stops taking steps, not receiving and sending messages.
- It may **recover** after crashing.
  - The process emits a <Recovery> event upon recovery.
- Access to **stable storage**:
  - May read/write (**expensive**) to permanent storage device.
  - Storage survives crashes.
  - E.g., save state to storage, crash, recover, read saved state, ...
- A failure is different in the crash-recovery abstraction:
  - A process is **faulty** in an execution if
    - It crashes and never recovers, or
    - It crashes and recovers infinitely often.
  - Hence, a **correct** process may crash and recover.

## Byzantine failures

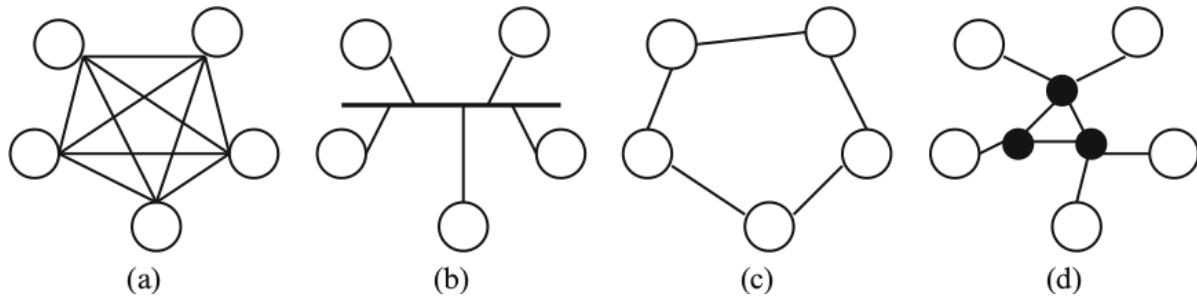
- A process may **behave arbitrarily**.
  - Sending messages not specified by its algorithm.
  - Updating its state as not specified by its algorithm.
- Might behave **maliciously**, attacking the system.
  - Several malicious nodes might collude.



# Communication abstractions

# Links

- Every process may **logically** communicate with every other process (a).
- The physical implementation may **differ** (b-d).



# Link failures

- Fair-loss links
  - Channel delivers any message sent, with non-zero probability.
- Stubborn links
  - Channel delivers any message sent infinitely many times.
  - Can be implemented using fair-loss links.
- Perfect links (reliable)
  - Channel delivers any message sent exactly once.
  - Can be implemented using stubborn links.
  - By default, we assume the perfect links abstraction.
- [Q] What abstraction do UDP and TCP implement?

## Stubborn links (*sl*)

**Module:**

**Name:** StubbornPointToPointLinks, **instance** *sl*.

**Events:**

**Request:**  $\langle sl, Send \mid q, m \rangle$ : Requests to send message *m* to process *q*.

**Indication:**  $\langle sl, Deliver \mid p, m \rangle$ : Delivers message *m* sent by process *p*.

**Properties:**

**SL1: Stubborn delivery:** If a correct process *p* sends a message *m* once to a correct process *q*, then *q* delivers *m* an infinite number of times.

**SL2: No creation:** If some process *q* delivers a message *m* with sender *p*, then *m* was previously sent to *q* by process *p*.

[Q] Which property is safety/liveness/neither?

## Perfect links (*pl*)

**Module:**

**Name:** PerfectPointToPointLinks, instance *pl*.

**Events:**

**Request:**  $\langle pl, Send \mid q, m \rangle$ : Requests to send message *m* to process *q*.

**Indication:**  $\langle pl, Deliver \mid p, m \rangle$ : Delivers message *m* sent by process *p*.

**Properties:**

**PL1: Reliable delivery:** If a correct process *p* sends a message *m* to a correct process *q*, then *q* eventually delivers *m*.

**PL2: No duplication:** No message is delivered by a process more than once.

**PL3: No creation:** If some process *q* delivers a message *m* with sender *p*, then *m* was previously sent to *q* by process *p*.

[Q] Which property is safety/liveness/neither?

**Implements:**

PerfectPointToPointLinks, **instance** *pl*.

**Uses:**

StubbornPointToPointLinks, **instance** *sl*.

**upon event**  $\langle pl, \text{Init} \rangle$  **do**  
*delivered* :=  $\emptyset$ ;

**upon event**  $\langle pl, \text{Send} \mid q, m \rangle$  **do**  
    **trigger**  $\langle sl, \text{Send} \mid q, m \rangle$ ;

**upon event**  $\langle sl, \text{Deliver} \mid p, m \rangle$  **do**  
    **if** *m*  $\notin$  *delivered* **then**  
        *delivered* := *delivered*  $\cup \{m\}$ ;  
        **trigger**  $\langle pl, \text{Deliver} \mid p, m \rangle$ ;

[Q] How does TCP efficiently maintain its delivered log?

## Correctness of *pl*

- PL1. Reliable delivery
  - Guaranteed by the Stubborn link abstraction. (The Stubborn link will deliver the message an infinite number of times.)
- PL2. No duplication
  - Guaranteed by the log mechanism.
- PL3. No creation
  - Guaranteed by the Stubborn link abstraction.

# Timing abstractions

# Timing assumptions

- Timing assumptions correspond to the **behavior** of processes and links **with respect to the passage of time**. They relate to
  - different processing speeds of processes;
  - different speeds of messages (channels).
- Three basic types of system:
  - Asynchronous system
  - Synchronous system
  - Partially synchronous system

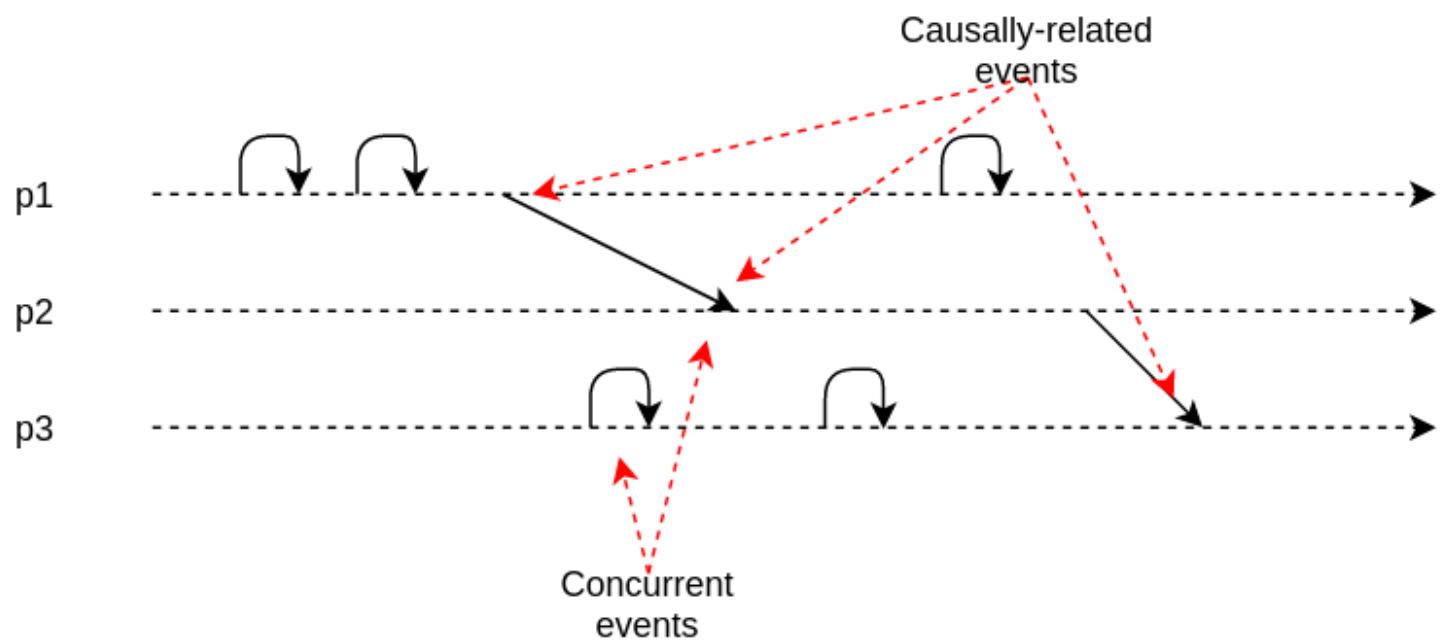
# Asynchronous systems

- No timing assumptions on processes and links.
  - Processes do not have access to any sort of physical clock.
  - Processing time may vary arbitrarily.
  - No bound on transmission time.
- But causality between events can still be determined.
  - How?

## Causal order

The **happened-before** relation  $e_1 \rightarrow e_2$  denotes that  $e_1$  may have caused  $e_2$ . It is true in the following cases:

- **FIFO order:**  $e_1$  and  $e_2$  occurred at the same process  $p$  and  $e_1$  occurred before  $e_2$ ;
- **Network order:**  $e_1$  corresponds to the transmission of  $m$  at a process  $p$  and  $e_2$  corresponds to its reception at a process  $q$ ;
- **Transitivity:** if  $e_1 \rightarrow e'$  and  $e' \rightarrow e_2$ , then  $e_1 \rightarrow e_2$ .



## Similarity of executions

- The **view** of  $p$  in  $E$ , denoted  $E|p$  is the subsequence of process steps in  $E$  restricted to those of  $p$
- Two executions  $E$  and  $F$  are **similar w.r.t. to  $p$**  if  $E|p = F|p$ .
- Two executions  $E$  and  $F$  are **similar** if  $E|p = F|p$  for all processes  $p$ .

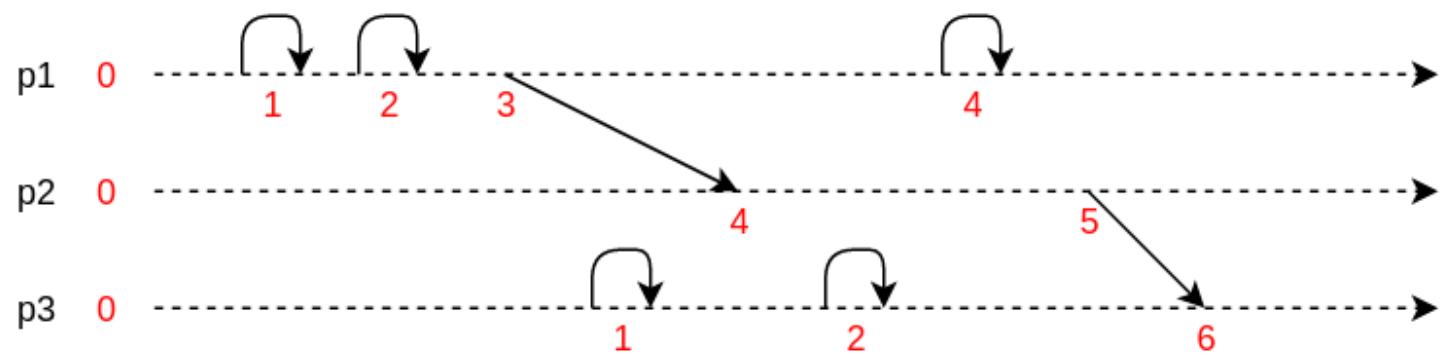
## Computation theorem

If two executions  $E$  and  $F$  have the same collection of events and their **causal order** is preserved, then  $E$  and  $F$  are similar executions.

## Logical clocks

In an asynchronous distributed system, the passage of time can be measured with **logical clocks**:

- Each process has a local logical clock  $l_p$ , initially set a  $0$ .
- Whenever an event occurs locally at  $p$  or when a process sends a message,  $p$  increments its logical clock.
  - $l_p := l_p + 1$
- When  $p$  sends a message event  $m$ , it timestamps the message with its current logical time,  $t(m) := l_p$ .
- When  $p$  receives a message event  $m$  with timestamp  $t(m)$ ,  $p$  updates its logical clock.
  - $l_p := \max(l_p, t(m)) + 1$



## Clock consistency condition

Logical clocks capture **cause-effect relations**:

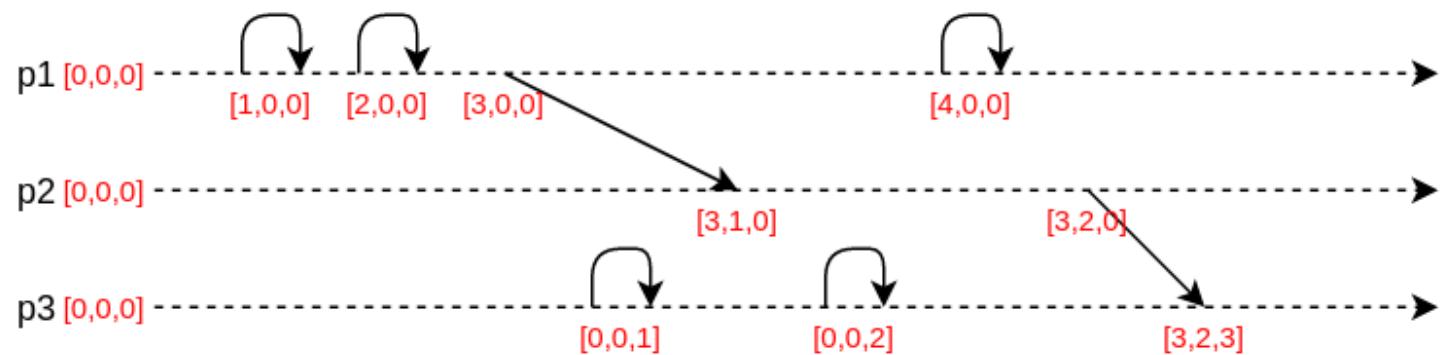
$$e_1 \rightarrow e_2 \Rightarrow t(e_1) < t(e_2)$$

- If  $e_1$  is the cause of  $e_2$ , then  $t(e_1) < t(e_2)$ .
  - Can you prove it?
- But not necessarily the opposite:
  - $t(e_1) < t(e_2)$  does not imply  $e_1 \rightarrow e_2$ .
  - $e_1$  and  $e_2$  may be logically **concurrent**.

## Vector clocks

Vector clocks fix this issue by making it possible to tell when two events cannot be causally related, i.e. when they are concurrent.

- Each process  $p$  maintains a vector  $V_p$  of  $N$  clocks, initially set at  $V_p[i] = 0 \forall i$ .
- Whenever an event occurs locally at  $p$  or when a process sends a message,  $p$  increments the  $p$ -th element of its vector clock.
  - $V_p[p] := V_p[p] + 1$
- When  $p$  sends a message event  $m$ , it piggybacks its vector clock as  $V_m := V_p$ .
- When  $p$  receives a message event  $m$  with the vector clock  $V_m$ ,  $p$  updates its vector clock.
  - $V_p[p] := V_p[p] + 1$
  - $V_p[i] := \max(V_p[i], V_m[i])$ , for  $i \neq p$ .



## Comparing vector clocks

- $V_p = V_q$ 
  - iff  $\forall i V_p[i] = V_q[i]$ .
- $V_p \leq V_q$ 
  - iff  $\forall i V_p[i] \leq V_q[i]$ .
- $V_p < V_q$ 
  - iff  $V_p \leq V_q$  AND  $\exists j V_p[j] < V_q[j]$
- $V_p$  and  $V_q$  are logically concurrent.
  - iff NOT  $V_p \leq V_q$  AND NOT  $V_q \leq V_p$

# Synchronous systems

Assumption of three properties:

- **Synchronous computation**
  - Known upper bound on the process computation delay.
- **Synchronous communication**
  - Known upper bound on message transmission delay.
- **Synchronous physical clocks**
  - Processes have access to a local physical clock;
  - Known upper bound on clock drift and clock skew.

[Q] Why studying synchronous systems? What services can be provided?

# Partially synchronous systems

A partially synchronous system is a system that is synchronous **most of the time**.

- There are periods where the timing assumptions of a synchronous system do not hold.
- But the distributed algorithm will have a long enough time window where everything behaves nicely, so that it can achieve its goal.

[Q] Are there such systems?

# Failure detection

- It is **tedious** to model (partial) synchrony.
- Timing assumptions are mostly needed to detect failures.
  - Heartbeats, timeouts, etc.
- We define **failure detector** abstractions to **encapsulate timing assumptions**:
  - Black box giving suspicions regarding node failures;
  - Accuracy of suspicions depends on model strength.

## Implementation of failure detectors

A typical implementation is the following:

- Periodically exchange **heartbeat** messages;
- **Timeout** based on **worst case** message round trip;
- If timeout, then **suspect** node;
- If reception of a message from a suspected node, **revise suspicion** and increase timeout.

# Perfect detector ( $\mathcal{P}$ )

Assuming a crash-stop process abstraction, the **perfect detector** encapsulates the timing assumptions of a **synchronous system**.

**Module:**

**Name:** PerfectFailureDetector, instance  $\mathcal{P}$ .

**Events:**

**Indication:**  $\langle \mathcal{P}, \text{Crash} \mid p \rangle$ : Detects that process  $p$  has crashed.

**Properties:**

**PFD1: Strong completeness:** Eventually, every process that crashes is permanently detected by every correct process.

**PFD2: Strong accuracy:** If a process  $p$  is detected by any process, then  $p$  has crashed.

[Q] Which property is safety/liveness/neither?

**Implements:**

PerfectFailureDetector, instance  $\mathcal{P}$ .

**Uses:**

PerfectPointToPointLinks, instance  $pl$ .

**upon event**  $\langle \mathcal{P}, \text{Init} \rangle$  **do**

$alive := \Pi;$

$detected := \emptyset;$

    starttimer( $\Delta$ );

**upon event**  $\langle \text{Timeout} \rangle$  **do**

**forall**  $p \in \Pi$  **do**

**if**  $(p \notin alive) \wedge (p \notin detected)$  **then**

$detected := detected \cup \{p\};$

            trigger  $\langle \mathcal{P}, \text{Crash} \mid p \rangle;$

        trigger  $\langle pl, \text{Send} \mid p, [\text{HEARTBEATREQUEST}] \rangle;$

$alive := \emptyset;$

        starttimer( $\Delta$ );

**upon event**  $\langle pl, \text{Deliver} \mid q, [\text{HEARTBEATREQUEST}] \rangle$  **do**

    trigger  $\langle pl, \text{Send} \mid q, [\text{HEARTBEATREPLY}] \rangle;$

**upon event**  $\langle pl, \text{Deliver} \mid p, [\text{HEARTBEATREPLY}] \rangle$  **do**

$alive := alive \cup \{p\};$

## Correctness

We assume a synchronous system:

- The transmission delay is bounded by some known constant.
- Local processing is negligible.
- The timeout delay  $\Delta$  is chosen to be large enough such that
  - every process has enough time to send a heartbeat message to all,
  - every heartbeat message has enough time to be delivered,
  - the correct destination processes have enough time to process the heartbeat and to send a reply,
  - the replies have enough time to reach the original sender and to be processed.

- PFD1. Strong completeness
  - A crashed process  $p$  stops replying to heartbeat messages, and no process will deliver its messages. Every correct process will thus eventually detect the crash of  $p$ .
- PFD2. Strong accuracy
  - The crash of  $p$  is detected by some other process  $q$  only if  $q$  does not deliver a message from  $p$  before the timeout period.
  - This happens only if  $p$  has indeed crashed, because the algorithm makes sure  $p$  must have sent a message otherwise and the synchrony assumptions imply that the message should have been delivered before the timeout period.

# Eventually perfect detector ( $\diamond \mathcal{P}$ )

The **eventually perfect detector** encapsulates the timing assumptions of a **partially synchronous system**.

**Module:**

**Name:** EventuallyPerfectFailureDetector, **instance**  $\diamond \mathcal{P}$ .

**Events:**

**Indication:**  $\langle \diamond \mathcal{P}, \text{Suspect} \mid p \rangle$ : Notifies that process  $p$  is suspected to have crashed.

**Indication:**  $\langle \diamond \mathcal{P}, \text{Restore} \mid p \rangle$ : Notifies that process  $p$  is not suspected anymore.

**Properties:**

**EPFD1:** *Strong completeness*: Eventually, every process that crashes is permanently suspected by every correct process.

**EPFD2:** *Eventual strong accuracy*: Eventually, no correct process is suspected by any correct process.

**Implements:**

EventuallyPerfectFailureDetector, instance  $\diamond\mathcal{P}$ .

**Uses:**

PerfectPointToPointLinks, instance  $pl$ .

**upon event**  $\langle \diamond\mathcal{P}, Init \rangle$  **do**

$alive := \Pi;$   
 $suspected := \emptyset;$   
 $delay := \Delta;$   
 $starttimer(delay);$

**upon event**  $\langle \text{Timeout} \rangle$  **do**

**if**  $alive \cap suspected \neq \emptyset$  **then**  
     $delay := delay + \Delta;$   
**forall**  $p \in \Pi$  **do**  
    **if**  $(p \notin alive) \wedge (p \notin suspected)$  **then**  
         $suspected := suspected \cup \{p\};$   
        **trigger**  $\langle \diamond\mathcal{P}, Suspect \mid p \rangle;$   
    **else if**  $(p \in alive) \wedge (p \in suspected)$  **then**  
         $suspected := suspected \setminus \{p\};$   
        **trigger**  $\langle \diamond\mathcal{P}, Restore \mid p \rangle;$   
    **trigger**  $\langle pl, Send \mid p, [\text{HEARTBEATREQUEST}] \rangle;$   
 $alive := \emptyset;$   
 $starttimer(delay);$

**upon event**  $\langle pl, Deliver \mid q, [\text{HEARTBEATREQUEST}] \rangle$  **do**  
    **trigger**  $\langle pl, Send \mid q, [\text{HEARTBEATREPLY}] \rangle;$

**upon event**  $\langle pl, Deliver \mid p, [\text{HEARTBEATREPLY}] \rangle$  **do**  
     $alive := alive \cup \{p\};$

[Q] Show that this implementation is correct.

# Leader election (*le*)

- Failure detection captures failure behavior.
  - Detects **failed** processes.
- Leader election is an abstraction that also captures failure behavior.
  - Detects **correct** nodes.
  - But a single and same for all, called the **leader**.
- If the current leader crashes, a new leader should be elected.

**Module:**

**Name:** LeaderElection, instance  $le$ .

**Events:**

**Indication:**  $\langle le, Leader \mid p \rangle$ : Indicates that process  $p$  is elected as leader.

**Properties:**

**LE1:** *Eventual detection*: Either there is no correct process, or some correct process is eventually elected as the leader.

**LE2:** *Accuracy*: If a process is leader, then all previously elected leaders have crashed.

**Implements:**  
LeaderElection, instance  $le$ .

**Uses:**  
PerfectFailureDetector, instance  $\mathcal{P}$ .

```
upon event < le, Init > do
    suspected := ∅;
    leader := ⊥;

upon event < P, Crash | p > do
    suspected := suspected ∪ {p};

upon leader ≠ maxrank(Π \ suspected) do
    leader := maxrank(Π \ suspected);
    trigger < le, Leader | leader >;
```

[Q] Show that this implementation is correct.

[Q] Is LE a failure detector?

# **Distributed system models**

# Distributed system models

We define a **distributed system model** as the combination of (i) a process abstraction, (ii) a link abstraction, and (iii) a failure detector abstraction.

- **Fail-stop** (synchronous)
  - Crash-stop process abstraction
  - Perfect links
  - Perfect failure detector
- **Fail-silent** (asynchronous)
  - Crash-stop process abstraction
  - Perfect links

- **Fail-noisy** (partially synchronous)

- Crash-stop process abstraction
- Perfect links
- Eventually perfect failure detector

- **Fail-recovery**

- Crash-stop process abstraction
- Stubborn links

The fail-stop distributed system model substantially simplifies the design of distributed algorithms.



# References

- Alpern, Bowen, and Fred B. Schneider. "Recognizing safety and liveness." *Distributed computing* 2.3 (1987): 117-126.
- Lamport, Leslie. "Time, clocks, and the ordering of events in a distributed system." *Communications of the ACM* 21.7 (1978): 558-565.
- Fidge, Colin J. "Timestamps in message-passing systems that preserve the partial ordering." (1987): 56-66.

# Large-scale Data Systems

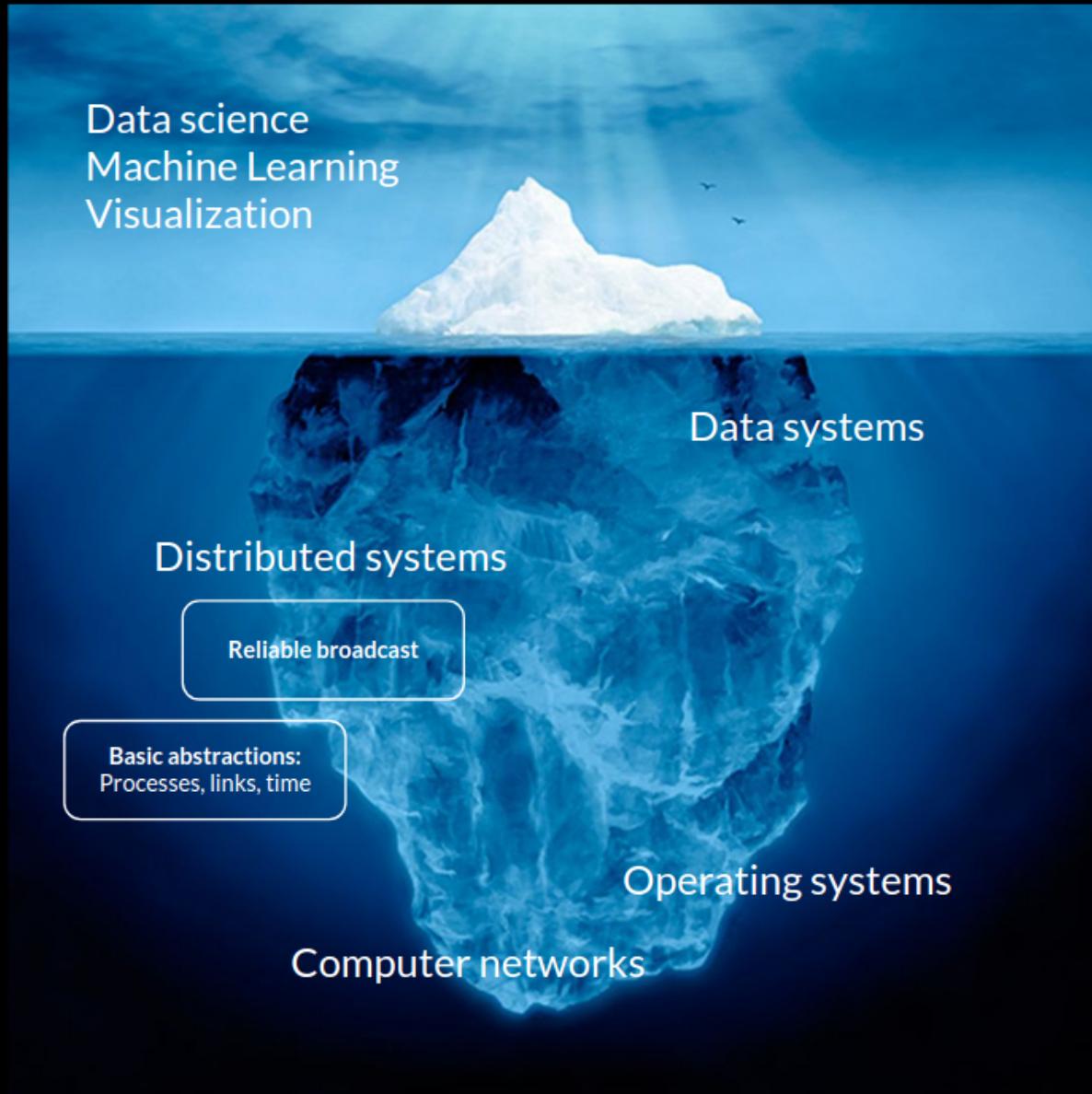
Lecture 3: Reliable broadcast

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)

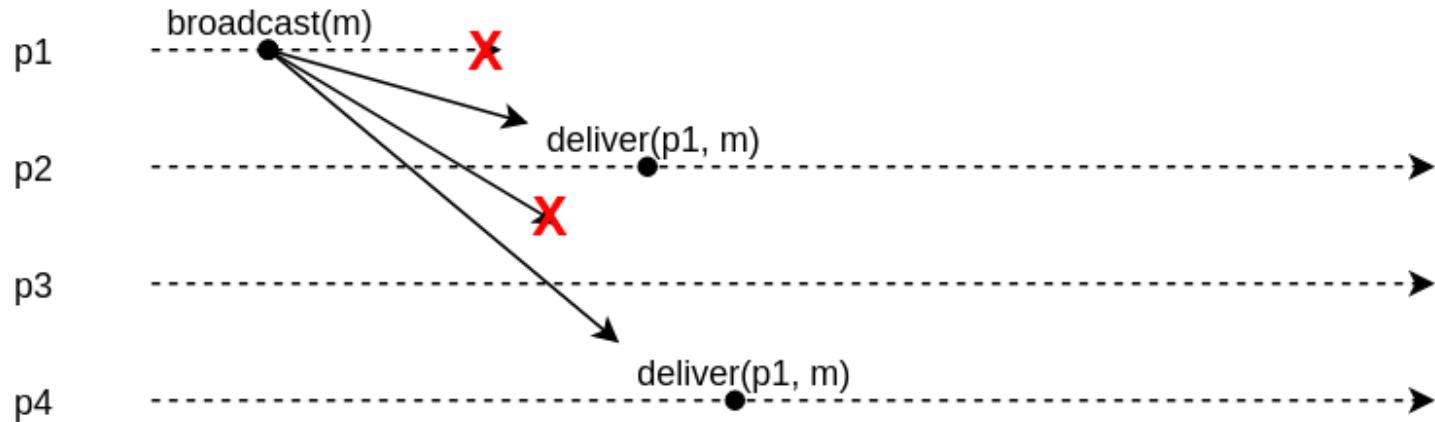


# Today

- How do you talk to **multiple machines** at once?
- What if some of them **fail**?
- Can we guarantee that correct nodes all receive the same messages?
- What about **ordering**?
- What about **performance**?



# Unreliable broadcast



Constraints:

- The sender may fail.
- Recipients may fail.
- Packets might get lost.
- Packets may take long to travel.

How do we define a **reliable** broadcast service?

# **Reliable broadcast abstractions**

# Reliable broadcast abstractions

- Best-effort broadcast
  - Guarantees reliability only if sender is correct.
- Reliable broadcast
  - Guarantees reliability independent of whether sender is correct.
- Uniform reliable broadcast
  - Also considers the behavior of failed nodes.
- Causal reliable broadcast
  - Reliable broadcast with causal delivery order.

# Best-effort broadcast (*beb*)

**Module:**

**Name:** BestEffortBroadcast, **instance** *beb*.

**Events:**

**Request:**  $\langle \text{beb}, \text{Broadcast} \mid m \rangle$ : Broadcasts a message *m* to all processes.

**Indication:**  $\langle \text{beb}, \text{Deliver} \mid p, m \rangle$ : Delivers a message *m* broadcast by process *p*.

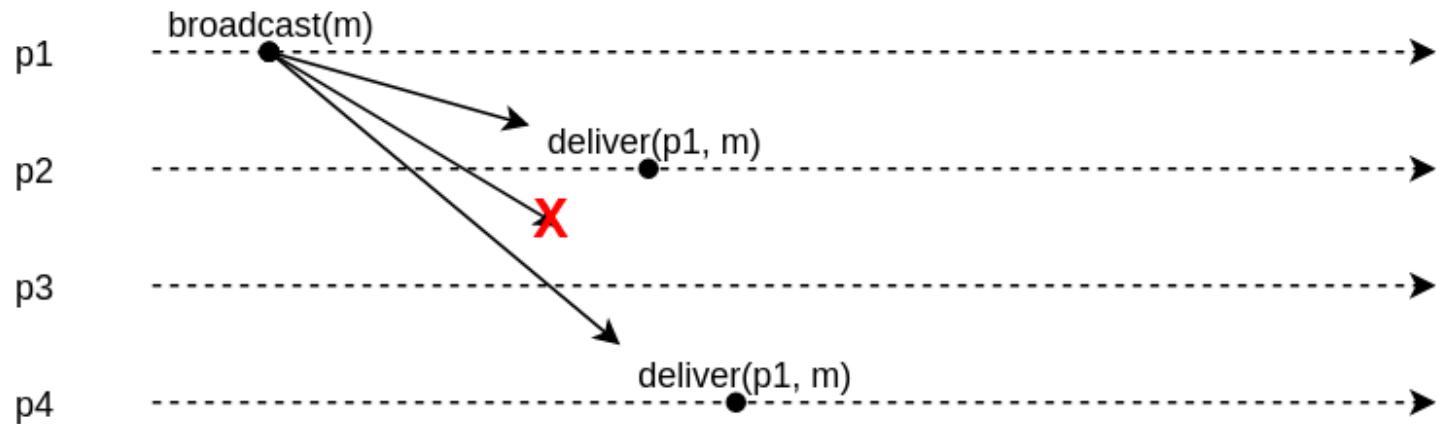
**Properties:**

**BEB1:** *Validity*: If a correct process broadcasts a message *m*, then every correct process eventually delivers *m*.

**BEB2:** *No duplication*: No message is delivered more than once.

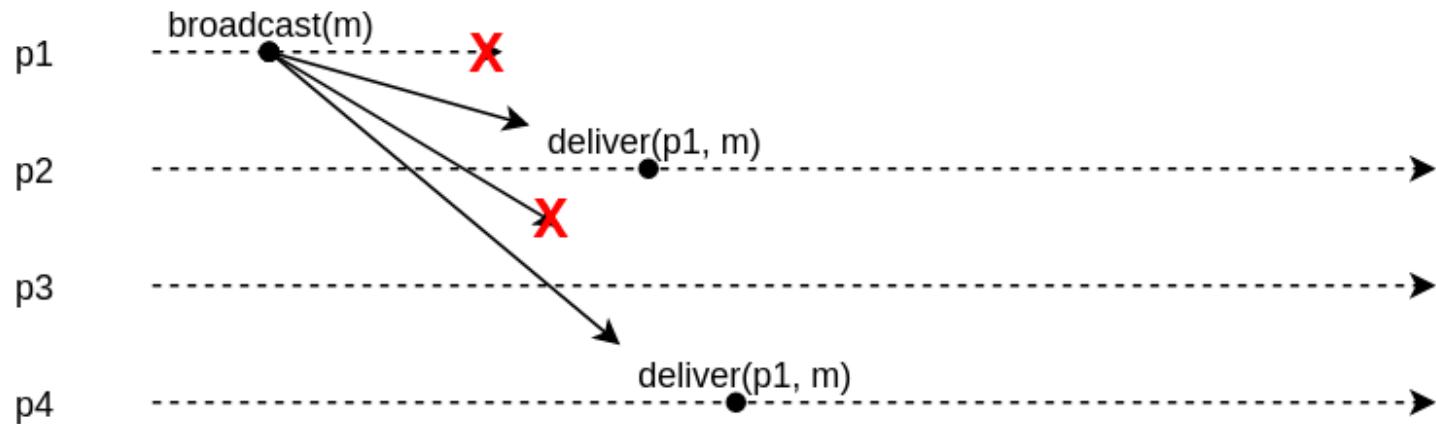
**BEB3:** *No creation*: If a process delivers a message *m* with sender *s*, then *m* was previously broadcast by process *s*.

## *beb* example (1)



[Q] Is this allowed?

## *beb* example (2)



[Q] Is this allowed?

# Reliable broadcast ( $rb$ )

- Best-effort broadcast gives no guarantees if **sender crashes**.
- **Reliable broadcast**:
  - Same as best-effort broadcast +
  - If sender crashes, ensure **all or none** of the correct node deliver the message.

**Module:**

**Name:** ReliableBroadcast, instance  $rb$ .

**Events:**

**Request:**  $\langle rb, Broadcast \mid m \rangle$ : Broadcasts a message  $m$  to all processes.

**Indication:**  $\langle rb, Deliver \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

**Properties:**

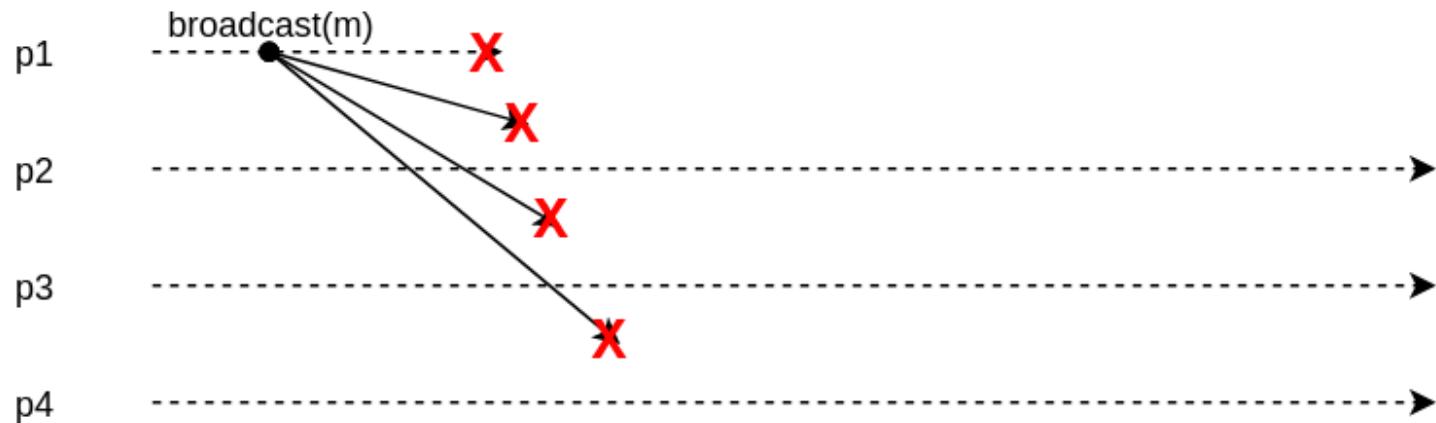
**RB1: Validity:** If a correct process  $p$  broadcasts a message  $m$ , then  $p$  eventually delivers  $m$ .

**RB2: No duplication:** No message is delivered more than once.

**RB3: No creation:** If a process delivers a message  $m$  with sender  $s$ , then  $m$  was previously broadcast by process  $s$ .

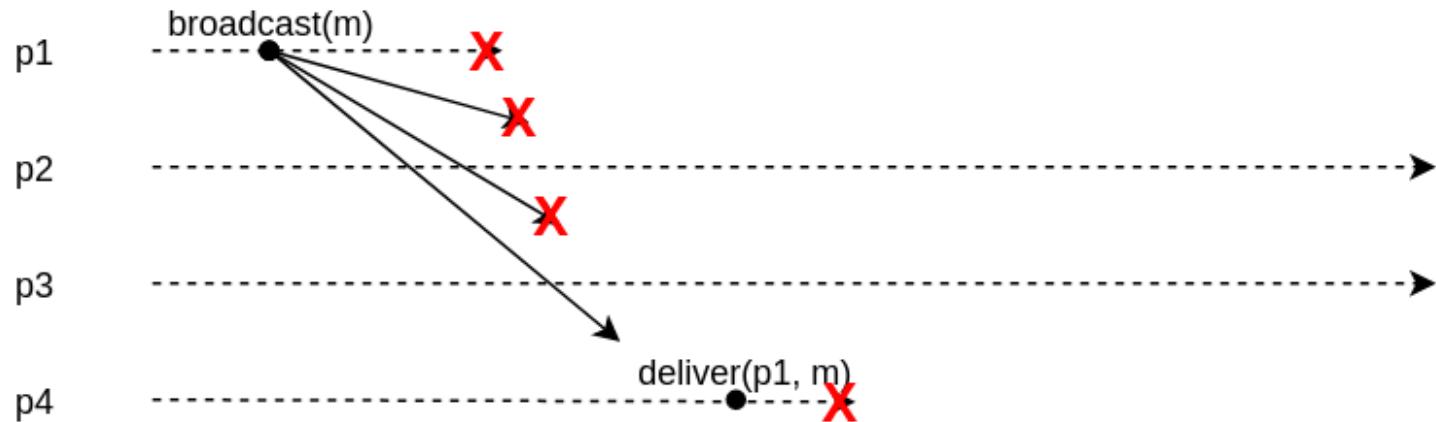
**RB4: Agreement:** If a message  $m$  is delivered by some correct process, then  $m$  is eventually delivered by every correct process.

## *rb* example (1)



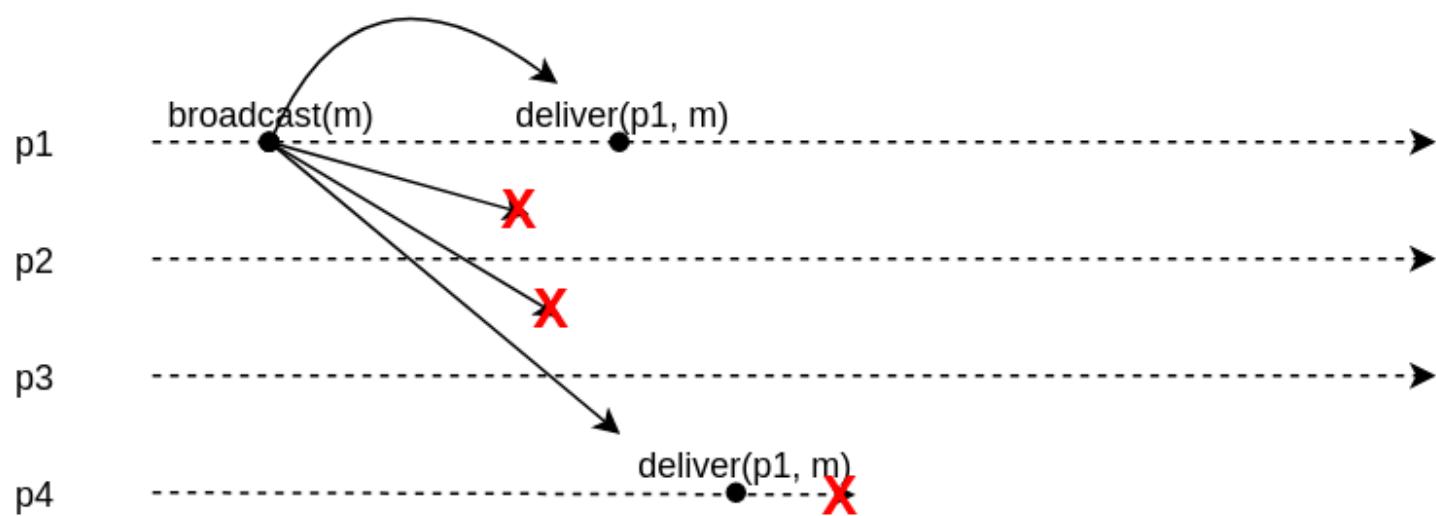
[Q] Is this allowed?

## *rb* example (2)



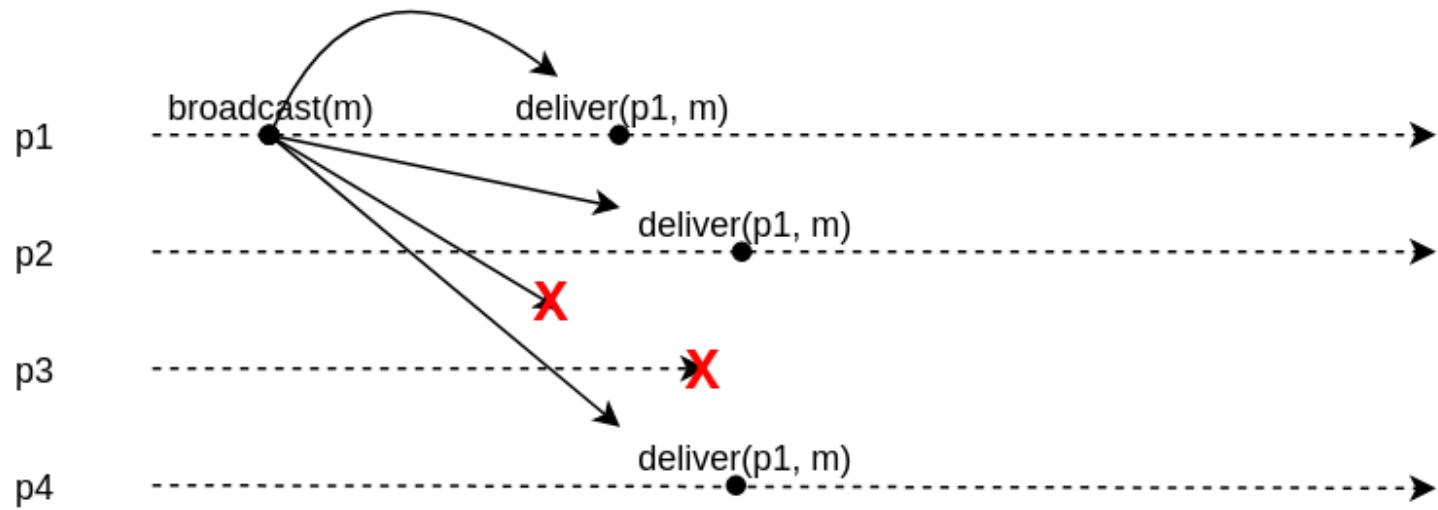
[Q] Is this allowed?

## *rb* example (3)



[Q] Is this allowed?

## *rb* example (4)



[Q] Is this allowed?

# Uniform reliable broadcast (*urb*)

- Assume sender broadcasts a message
  - Sender fails
  - No correct node delivers the message
  - Failed nodes deliver the message
- Is this OK?
  - A process that delivers a message and later crashes may bring the application into a inconsistent state.
- **Uniform** reliable broadcast ensures that if a message is delivered, by a correct or a **faulty** process, then all correct processes deliver.

**Module:**

**Name:** UniformReliableBroadcast, **instance**  $urb$ .

**Events:**

**Request:**  $\langle urb, Broadcast \mid m \rangle$ : Broadcasts a message  $m$  to all processes.

**Indication:**  $\langle urb, Deliver \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

**Properties:**

**URB1–URB3:** Same as properties RB1–RB3 in (regular) reliable broadcast (Module 3.2).

**URB4:** *Uniform agreement*: If a message  $m$  is delivered by some process (whether correct or faulty), then  $m$  is eventually delivered by every correct process.

# Implementations

# Basic broadcast

**Implements:**

BestEffortBroadcast, **instance** *beb*.

**Uses:**

PerfectPointToPointLinks, **instance** *pl*.

**upon event**  $\langle \text{beb}, \text{Broadcast} \mid m \rangle$  **do**

**forall**  $q \in \Pi$  **do**

**trigger**  $\langle \text{pl}, \text{Send} \mid q, m \rangle$ ;

**upon event**  $\langle \text{pl}, \text{Deliver} \mid p, m \rangle$  **do**

**trigger**  $\langle \text{beb}, \text{Deliver} \mid p, m \rangle$ ;

Correctness:

- **BEB1. Validity:** If a correct process  $p$  broadcasts  $m$ , then every correct process eventually delivers  $m$ .
  - If sender does not crash, every other correct node receives message by perfect channels.
- **BEB2+3. No duplication + no creation**
  - Guaranteed by perfect channels.

# Lazy reliable broadcast

- Assume a fail-stop distributed system model.
  - i.e., crash-stop processes, perfect links and a perfect failure detector.
- To broadcast  $m$ :
  - best-effort broadcast  $m$
  - Upon `bebDeliver`:
    - Save message
    - `rbDeliver` the message
- If sender  $s$  crashes, detect and relay messages from  $s$  to all.
  - case 1: get  $m$  from  $s$ , detect crash of  $s$ , redistribute  $m$
  - case 2: detect crash of  $s$ , get  $m$  from  $s$ , redistribute  $m$ .
- Filter duplicate messages.

**Implements:**

ReliableBroadcast, **instance**  $rb$ .

**Uses:**

BestEffortBroadcast, **instance**  $beb$ ;  
PerfectFailureDetector, **instance**  $\mathcal{P}$ .

**upon event**  $\langle rb, \text{Init} \rangle$  **do**

$correct := \Pi$ ;  
     $from[p] := [\emptyset]^N$ ;

**upon event**  $\langle rb, \text{Broadcast} \mid m \rangle$  **do**

**trigger**  $\langle beb, \text{Broadcast} \mid [\text{DATA}, \text{self}, m] \rangle$ ;

**upon event**  $\langle beb, \text{Deliver} \mid p, [\text{DATA}, s, m] \rangle$  **do**

**if**  $m \notin from[s]$  **then**

**trigger**  $\langle rb, \text{Deliver} \mid s, m \rangle$ ;

$from[s] := from[s] \cup \{m\}$ ;

**if**  $s \notin correct$  **then**

**trigger**  $\langle beb, \text{Broadcast} \mid [\text{DATA}, s, m] \rangle$ ;

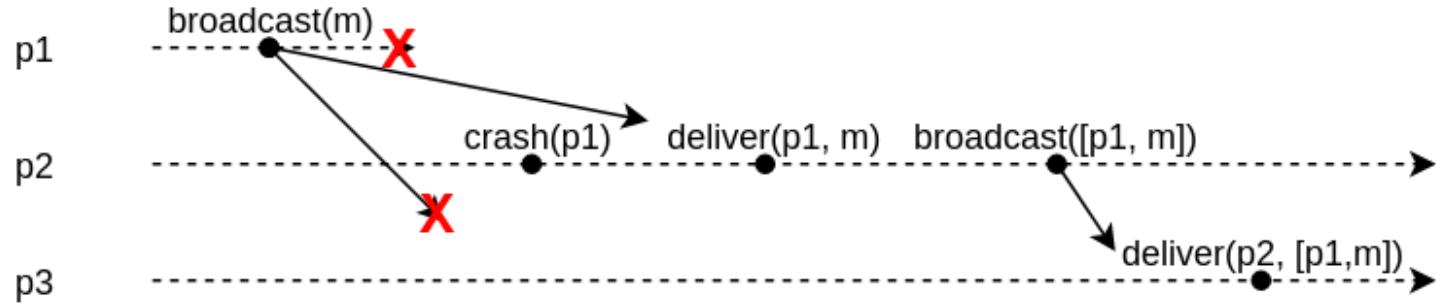
**upon event**  $\langle \mathcal{P}, \text{Crash} \mid p \rangle$  **do**

$correct := correct \setminus \{p\}$ ;

**forall**  $m \in from[p]$  **do**

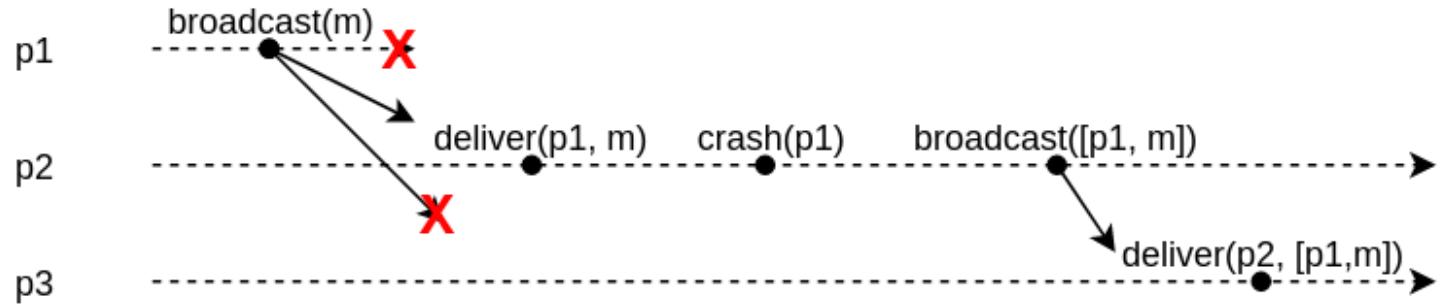
**trigger**  $\langle beb, \text{Broadcast} \mid [\text{DATA}, p, m] \rangle$ ;

## Lazy reliable broadcast example (1)



[Q] Which case?

## Lazy reliable broadcast example (2)



[Q] Which case?

## Correctness of lazy reliable broadcast

- RB1-RB3
  - Satisfied with best-effort broadcast.
- RB4. **Agreement:** If a message  $m$  is delivered by some correct process, then  $m$  is eventually delivered by every correct process.
  - When correct  $p_j$  delivers  $m$  broadcast by  $p_i$ 
    - if  $p_i$  is correct, BEB ensures correct delivery
    - if  $p_i$  crashes,
      - $p_j$  detects this (because of completeness of the PFD)
      - $p_j$  uses BEB to ensure (BEB1) every correct node gets  $m$ .

# Eager reliable broadcast

- What happens if we use instead an **eventually** perfect failure detector?
  - Only affects performance, not correctness.
- Can we modify Lazy RB to not use a perfect failure detector?
  - Assume all nodes have failed.
  - BEB broadcast all received messages.

**Implements:**

ReliableBroadcast, **instance** *rb*.

**Uses:**

BestEffortBroadcast, **instance** *beb*.

**upon event**  $\langle rb, \text{Init} \rangle$  **do**  
*delivered* :=  $\emptyset$ ;

**upon event**  $\langle rb, \text{Broadcast} \mid m \rangle$  **do**  
    **trigger**  $\langle beb, \text{Broadcast} \mid [\text{DATA}, self, m] \rangle$ ;

**upon event**  $\langle beb, \text{Deliver} \mid p, [\text{DATA}, s, m] \rangle$  **do**  
    **if** *m*  $\notin$  *delivered* **then**  
        *delivered* := *delivered*  $\cup \{m\}$ ;  
        **trigger**  $\langle rb, \text{Deliver} \mid s, m \rangle$ ;  
        **trigger**  $\langle beb, \text{Broadcast} \mid [\text{DATA}, s, m] \rangle$ ;

[Q] Show that eager reliable broadcast is correct.

# Uniformity

Neither Lazy reliable broadcast nor Eager reliable broadcast ensure **uniform** agreement.

E.g., sender  $p$  immediately RB delivers and crashes. Only  $p$  delivered the message.

## Strategy for uniform agreement

- Before delivering a message, we need to ensure all correct nodes have received it.
- Messages are **pending** until all correct nodes get it.
  - Collect acknowledgements from nodes that got the message.
- Deliver once all correct nodes acked.

# All-ack uniform reliable broadcast

**Implements:**

UniformReliableBroadcast, instance  $urb$ .

**Uses:**

BestEffortBroadcast, instance  $beb$ .

PerfectFailureDetector, instance  $\mathcal{P}$ .

```
upon event < urb, Init > do
    delivered := ∅;
    pending := ∅;
    correct := Π;
    forall m do ack[m] := ∅;

upon event < urb, Broadcast | m > do
    pending := pending ∪ {(self, m)};
    trigger < beb, Broadcast | [DATA, self, m] >;

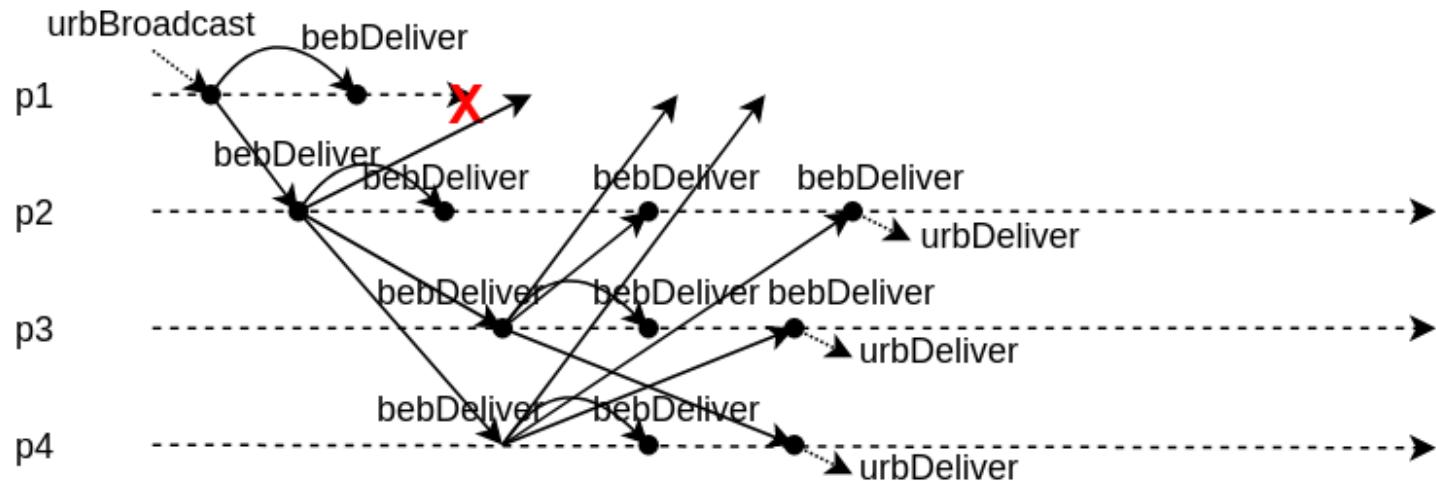
upon event < beb, Deliver | p, [DATA, s, m] > do
    ack[m] := ack[m] ∪ {p};
    if (s, m) ∉ pending then
        pending := pending ∪ {(s, m)};
        trigger < beb, Broadcast | [DATA, s, m] >;

upon event < P, Crash | p > do
    correct := correct \ {p};

function candeliver(m) returns Boolean is
    return (correct ⊆ ack[m]);

upon exists (s, m) ∈ pending such that candeliver(m) ∧ m ∉ delivered do
    delivered := delivered ∪ {m};
    trigger < urb, Deliver | s, m >;
```

## Example



## Correctness of All-ack URB

**Lemma.** If a correct node  $p$  BEB delivers  $m$ , then  $p$  eventually URB delivers  $m$ .

Proof:

- A correct node  $p$  BEB broadcasts  $m$  as soon as it gets  $m$ .
- By BEB1, every correct node gets  $m$  and BEB broadcasts  $m$ .
- Therefore  $p$  BEB delivers from every correct node by BEB1.
- By completeness of the perfect failure detector,  $p$  will not wait for dead nodes forever.
  - `canDeliver` becomes true and  $p$  URB delivers  $m$ .

- **URB1. Validity:** If a correct process  $p$  broadcasts  $m$ , then  $p$  delivers  $m$ 
  - If sender is correct, it will BEB delivers  $m$  by validity (BEB1)
  - By the lemma, it will therefore eventually URB delivers  $m$ .
- **URB2. No duplication**
  - Guaranteed because of the delivered set.
- **URB3. No creation**
  - Ensured from best-effort broadcast.
- **URB4. Uniform agreement:** If a message  $m$  is delivered by some process (correct or faulty), then  $m$  is eventually delivered by every correct process
  - Assume some node (possibly failed) URB delivers  $m$ .
    - Then `canDeliver` was true, and by accuracy of the failure detector, every correct node has BEB delivered  $m$ .
  - By the lemma, each of the nodes that BEB delivered  $m$  will URB deliver  $m$ .

# *urb* for fail-silent

- All-ack URB requires a perfect failure detector (fail-stop).
- Can we implement URB in **fail-silent**, without a perfect failure detector?
- **Yes**, provided a majority of nodes are correct.

**Implements:**

UniformReliableBroadcast, instance *urb*.

**Uses:**

BestEffortBroadcast, instance *beb*.

// Except for the function *candeliver*( $\cdot$ ) below and for the absence of  $\langle$  Crash  $\rangle$  events  
// triggered by the perfect failure detector, it is the same as Algorithm 3.4.

```
function candeliver(m) returns Boolean is
    return #(ack[m]) > N/2;
```

[Q] Show that this variant is correct.

# Causal reliable broadcast



**Mathias Verraes**

@mathiasverraes

Follow



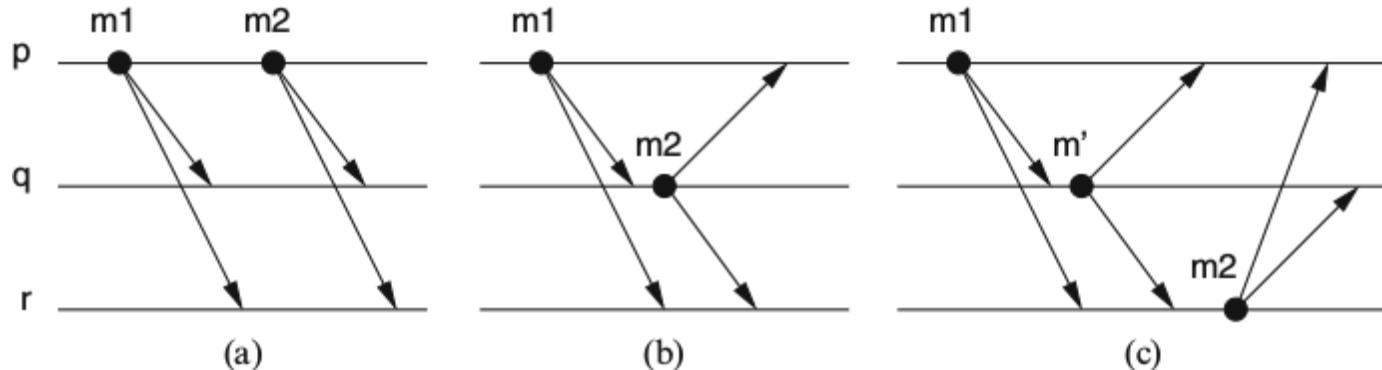
There are only two hard problems in distributed systems:  
1. Guaranteed order of messages  
2. Exactly-once delivery

Reliable broadcast:

- Exactly-once delivery: **guaranteed** by the properties of RB.
- Order of message? **Not guaranteed!**

[Q] Does uniform reliable broadcast remedy this?

# Causal order of messages



A message  $m_1$  may have caused another message  $m_2$ , denoted  $m_1 \rightarrow m_2$  if any of the following relations apply:

- (a) some process  $p$  broadcasts  $m_1$  before it broadcasts  $m_2$ ;
- (b) some process  $p$  delivers  $m_1$  and subsequently broadcasts  $m_2$ ; or
- (c) there exists some message  $m'$  such that  $m_1 \rightarrow m'$  and  $m' \rightarrow m_2$ .

# Causal broadcast (*crb*)

**Module:**

**Name:** CausalOrderReliableBroadcast, instance *crb*.

**Events:**

**Request:**  $\langle \text{crb}, \text{Broadcast} \mid m \rangle$ : Broadcasts a message  $m$  to all processes.

**Indication:**  $\langle \text{crb}, \text{Deliver} \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

**Properties:**

**CRB1–CRB4:** Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

**CRB5:** *Causal delivery*: For any message  $m_1$  that potentially caused a message  $m_2$ , i.e.,  $m_1 \rightarrow m_2$ , no process delivers  $m_2$  unless it has already delivered  $m_1$ .

# No-waiting causal broadcast

**Implements:**

CausalOrderReliableBroadcast, **instance** *crb*.

**Uses:**

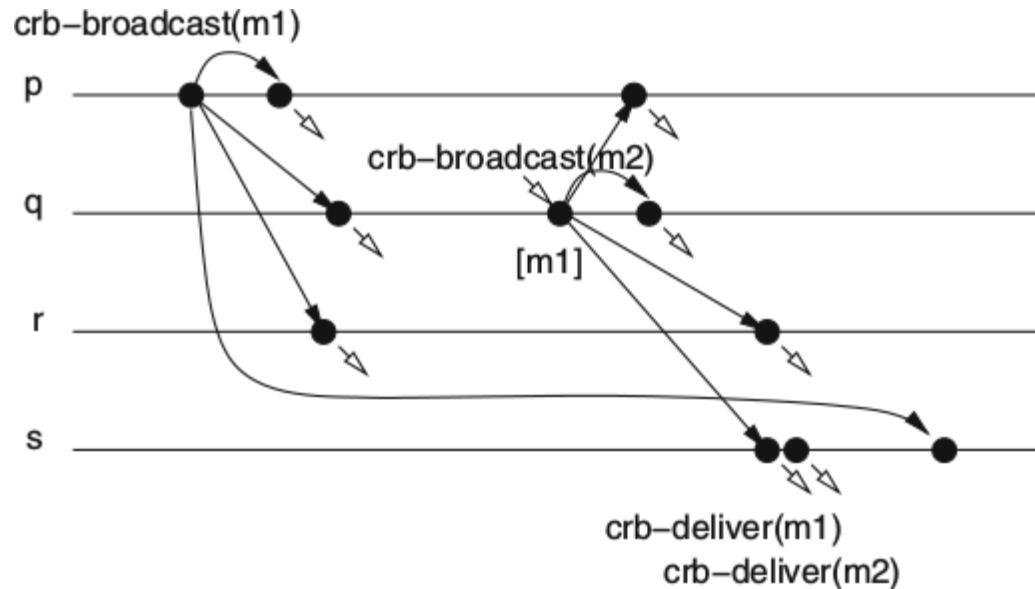
ReliableBroadcast, **instance** *rb*.

```
upon event < crb, Init > do
    delivered := ∅;
    past := [];

upon event < crb, Broadcast | m > do
    trigger < rb, Broadcast | [DATA, past, m] >;
    append(past, (self, m));

upon event < rb, Deliver | p, [DATA, mpast, m] > do
    if m ∉ delivered then
        forall (s, n) ∈ mpast do
            if n ∉ delivered then
                trigger < crb, Deliver | s, n >;
                delivered := delivered ∪ {n};
                if (s, n) ∉ past then
                    append(past, (s, n));
            trigger < crb, Deliver | p, m >;
            delivered := delivered ∪ {m};
            if (p, m) ∉ past then
                append(past, (p, m));
```

## No-waiting CB example



- The size of the message **grows with time**, as messages include their list of causally preceding messages  $m_{\text{past}}$ .
- Solution 1: **Garbage collect** old messages by sending acknowledgements of delivery to all nodes and purging messages that have been acknowledged from all.
- Solution 2: History is a **vector timestamp**!

# Waiting causal broadcast

**Implements:**

CausalOrderReliableBroadcast, **instance**  $crb$ .

**Uses:**

ReliableBroadcast, **instance**  $rb$ .

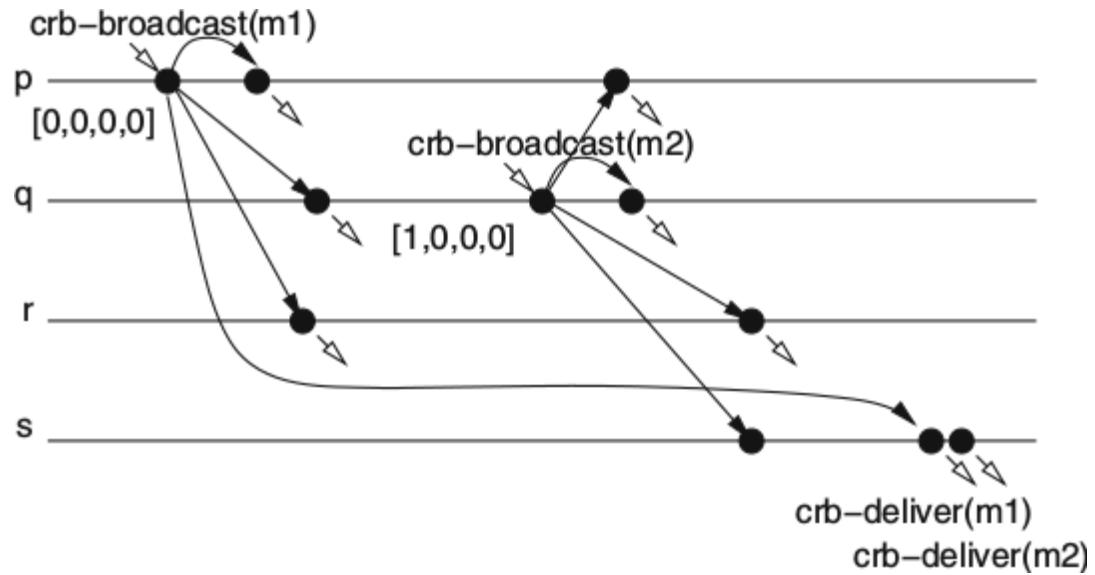
```
upon event <  $crb$ , Init > do
     $V := [0]^N$ ;
     $lsn := 0$ ;
     $pending := \emptyset$ ;

upon event <  $crb$ , Broadcast |  $m$  > do
     $W := V$ ;
     $W[\text{rank}(\text{self})] := lsn$ ;
     $lsn := lsn + 1$ ;
    trigger <  $rb$ , Broadcast | [DATA,  $W$ ,  $m$ ] >;

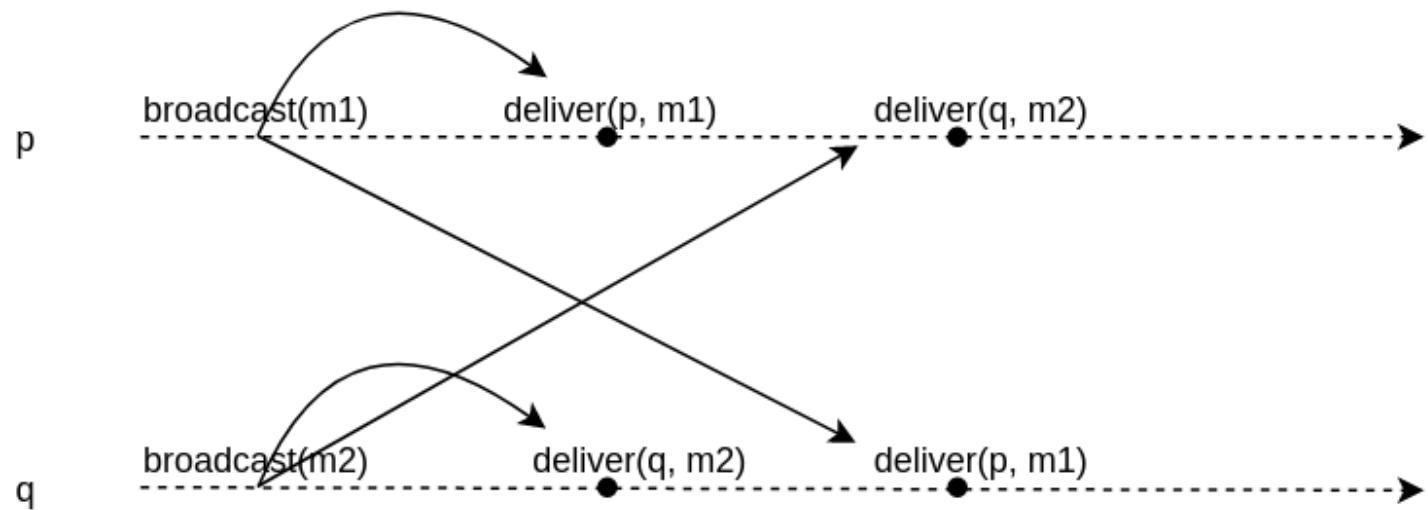
upon event <  $rb$ , Deliver |  $p$ , [DATA,  $W$ ,  $m$ ] > do
     $pending := pending \cup \{(p, W, m)\}$ ;
    while exists  $(p', W', m') \in pending$  such that  $W' \leq V$  do
         $pending := pending \setminus \{(p', W', m')\}$ ;
         $V[\text{rank}(p')] := V[\text{rank}(p')] + 1$ ;
        trigger <  $crb$ , Deliver |  $p'$ ,  $m'$  >;
```

[Q] Show the correctness of the algorithm.

## Waiting CB example



## Possible execution?



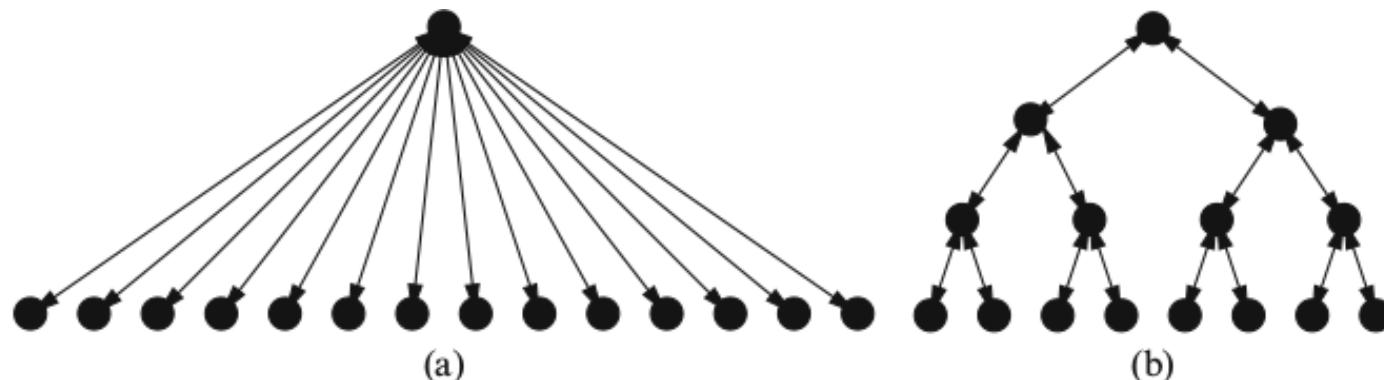
[Q] Is this a valid execution? the order of delivery is not the same.

# Probabilistic broadcast

(a.k.a. epidemic broadcast or gossiping.)

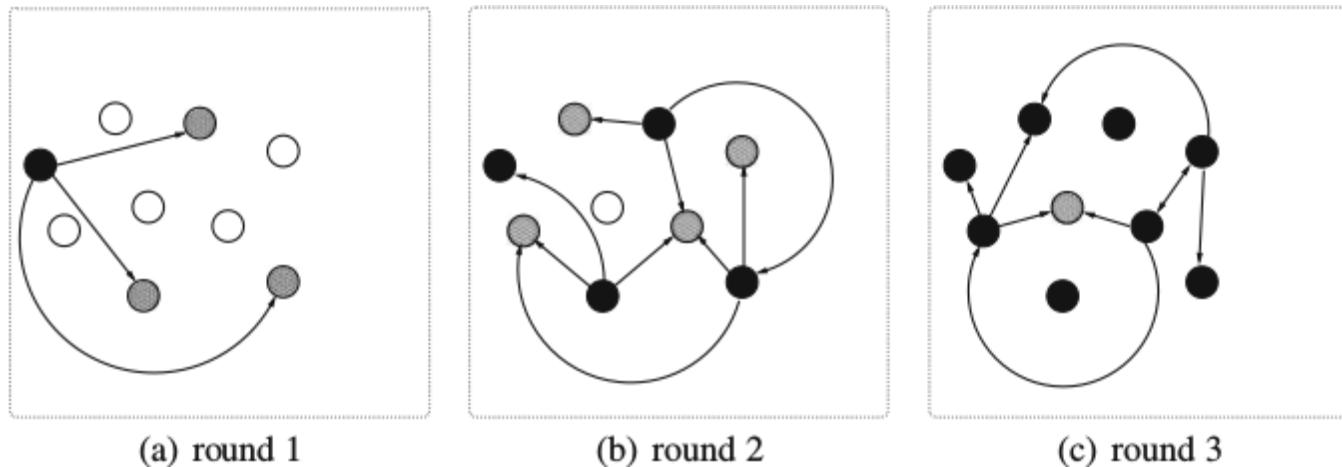
# Scalability of reliable broadcast

- In order to broadcast a message, the sender needs
  - to send messages to all other processes,
  - to collect some form of acknowledgement.
  - $O(N^2)$  are exchanged in total.
    - If  $N$  is large, this can become overwhelming for the system.
- Bandwidth, memory or processing resources may limit the number of messages/acknowledgements that may be sent/collected.
- Hierarchical schemes reduce the total number of messages.
  - This reduces the load of each process.
  - But increases the latency and fragility of the system.



# Epidemic dissemination

- Nodes infect each other through messages sent in **rounds**.
  - The **fanout  $k$**  determines the number of messages sent by each node.
  - Recipients are drawn **at random** (e.g., uniformly).
  - The **number of rounds** is limited to  $R$ .
- Total number of messages is usually less than  $O(N^2)$ .
- No node is overloaded.



(a) round 1

(b) round 2

(c) round 3

# Probabilistic broadcast ( $pb$ )

**Module:**

**Name:** ProbabilisticBroadcast, instance  $pb$ .

**Events:**

**Request:**  $\langle pb, Broadcast \mid m \rangle$ : Broadcasts a message  $m$  to all processes.

**Indication:**  $\langle pb, Deliver \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

**Properties:**

**PB1: Probabilistic validity:** There is a positive value  $\varepsilon$  such that when a correct process broadcasts a message  $m$ , the probability that every correct process eventually delivers  $m$  is at least  $1 - \varepsilon$ .

**PB2: No duplication:** No message is delivered more than once.

**PB3: No creation:** If a process delivers a message  $m$  with sender  $s$ , then  $m$  was previously broadcast by process  $s$ .

# Eager probabilistic broadcast

**Implements:**

ProbabilisticBroadcast, **instance**  $pb$ .

**Uses:**

FairLossPointToPointLinks, **instance**  $fl$ .

```
upon event ⟨ pb, Init ⟩ do
    delivered := ∅;

procedure gossip(msg) is
    forall t ∈ picktargets(k) do trigger ⟨ fl, Send | t, msg ⟩;

upon event ⟨ pb, Broadcast | m ⟩ do
    delivered := delivered ∪ {m};
    trigger ⟨ pb, Deliver | self, m ⟩;
    gossip([GOSSIP, self, m, R]);

upon event ⟨ fl, Deliver | p, [GOSSIP, s, m, r] ⟩ do
    if m ∉ delivered then
        delivered := delivered ∪ {m};
        trigger ⟨ pb, Deliver | s, m ⟩;
    if r > 1 then gossip([GOSSIP, s, m, r - 1]);
```

# The mathematics of epidemics

Assume a virus using a distributed system to propagate, with human hosts as nodes.

## Setup

- Initial population of  $N$  individuals.
- At any time  $t$ ,
  - $S(t)$  = the number of **susceptible** individuals,
  - $I(t)$  = the number of **infected** individuals.
- $I(0) = 1$
- $S(0) = N - 1$
- $S(t) + I(t) = N$  for all  $t$ .

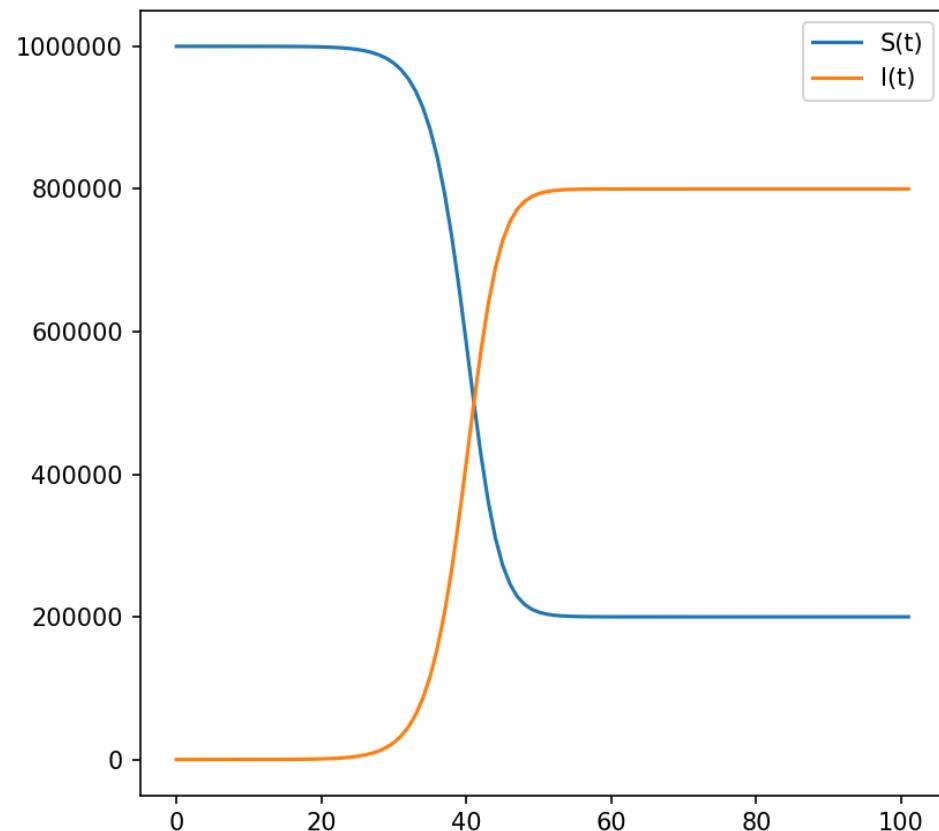
The expected dynamics of the SIS model is given as follows:

$$S(t+1) = S(t) - \frac{\alpha\Delta t}{N} S(t)I(t) + \gamma\Delta t I(t)$$

$$I(t+1) = I(t) + \frac{\alpha\Delta t}{N} S(t)I(t) - \gamma\Delta t I(t)$$

where

- $\alpha$  is the contact rate with whom infected individuals make contact per unit of time.
- $\frac{S(t)}{N}$  is the proportion of contacts with susceptible individuals for each infected individual.
- $\gamma$  is the probability for an infected individual to recover and switch to the pool of susceptibles.



$$N = 1000000, \alpha = 5, \gamma = 0.5, \Delta t = 0.1$$

In eager reliable broadcast,

- $\alpha = k$ 
  - An infected node selects  $k$  nodes among  $N$  to send its messages.
- $\gamma = 1$ 
  - An infected node immediately recovers.

# Probabilistic validity

At time  $t$ , the probability of not receiving a message is

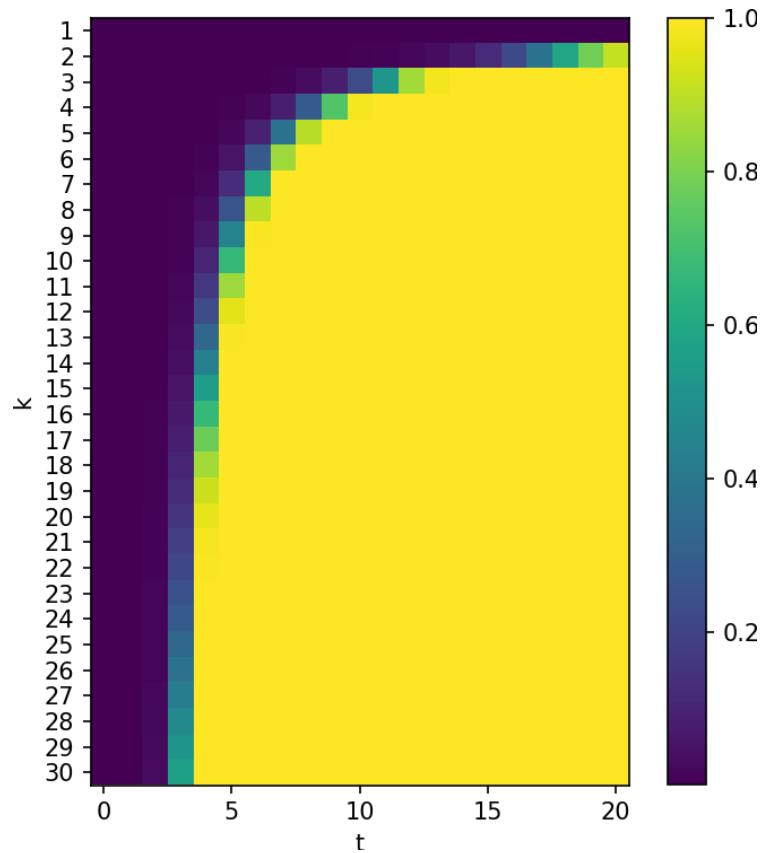
$$(1 - \frac{k}{N})^{i(t)}$$

Therefore the probability of having received of one or more gossip messages up to time  $t$ , that is to have PB-delivered, is

$$p(\text{delivery}) = 1 - (1 - \frac{k}{N})^{\sum_{t_i=0}^t i(t_i)}$$

[Q] What if nodes fail? if packets are loss?

$$p(\text{delivery}|k, t)$$



$$N = 1000000, \gamma = 1.0$$

From this plot, we observe that:

- Within only a few rounds (**low latency**), a large fraction of nodes receive the message (**reliability**)
- Each node has transmitted no more than  $kR$  messages (**lightweight**).

# Lazy Probabilistic broadcast

- Eager probabilistic broadcast consumes **considerable resources** and causes many **redundant transmissions**.
  - in particular as  $r$  gets larger and almost all nodes have received the message once.
- Assume **a stream of messages** to be broadcast.
- Broadcast messages in **two phases**:
  - **Phase 1 (data dissemination)**: run probabilistic broadcast with a large probability  $\epsilon$  that reliable delivery fails. That is, assume a constant fraction of nodes obtain the message (e.g.,  $\frac{1}{2}$ ).
  - **Phase 2 (recovery)**: upon delivery, detect omissions through sequence numbers and initiate retransmissions with gossip.

# Phase 1: data dissemination

Implements:

ProbabilisticBroadcast, instance  $pb$ .

Uses:

FairLossPointToPointLinks, instance  $fl$ ;

ProbabilisticBroadcast, instance  $upb$ .

// an unreliable implementation

```
upon event < pb, Init > do
    next := [1]N;
    lsn := 0;
    pending := ∅; stored := ∅;

procedure gossip(msg) is
    forall t ∈ picktargets(k) do trigger < fl, Send | t, msg >;
    
upon event < pb, Broadcast | m > do
    lsn := lsn + 1;
    trigger < upb, Broadcast | [DATA, self, m, lsn] >;
    
upon event < upb, Deliver | p, [DATA, s, m, sn] > do
    if random([0, 1]) > α then
        stored := stored ∪ {[DATA, s, m, sn]};
    if sn = next[s] then
        next[s] := next[s] + 1;
        trigger < pb, Deliver | s, m >;
    else if sn > next[s] then
        pending := pending ∪ {[DATA, s, m, sn]};
        forall missing ∈ [next[s], . . . , sn − 1] do
            if no  $m'$  exists such that [DATA, s,  $m'$ , missing] ∈ pending then
                gossip([REQUEST, self, s, missing, R − 1]);
            starttimer( $\Delta$ , s, sn);
```

## Phase 2: recovery

```
upon event <fl, Deliver | p, [REQUEST, q, s, sn, r] > do
    if exists m such that [DATA, s, m, sn] ∈ stored then
        trigger <fl, Send | q, [DATA, s, m, sn] >;
    else if r > 0 then
        gossip([REQUEST, q, s, sn, r - 1]);
    end

upon event <fl, Deliver | p, [DATA, s, m, sn] > do
    pending := pending ∪ {[DATA, s, m, sn]};
    next[s] := next[s] + 1;
    pending := pending \ {[DATA, s, x, sn]};
    trigger <pb, Deliver | s, x >;
    end

upon exists [DATA, s, x, sn] ∈ pending such that sn = next[s] do
    next[s] := next[s] + 1;
    pending := pending \ {[DATA, s, x, sn]};
    trigger <pb, Deliver | s, x >;
    end

upon event < Timeout | s, sn > do
    if sn > next[s] then
        next[s] := sn + 1;
    end
```

# Summary

- Reliable multicast enable group communication, while ensuring **validity** and (uniform) **agreement**.
- Causal broadcast extends reliable broadcast with **causal ordering** guarantees.
- **Probabilistic broadcast** enable low-latency, reliable and lightweight group communication.



# References

- Allen, Linda JS. "Some discrete-time SI, SIR, and SIS epidemic models." Mathematical biosciences 124.1 (1994): 83-105.

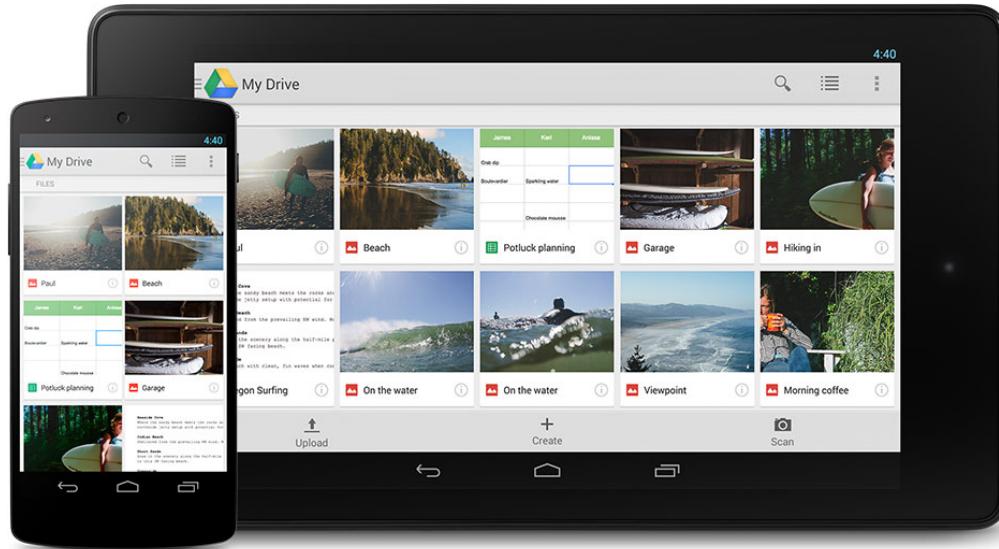
# Large-scale Data Systems

Lecture 4: Shared memory

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)

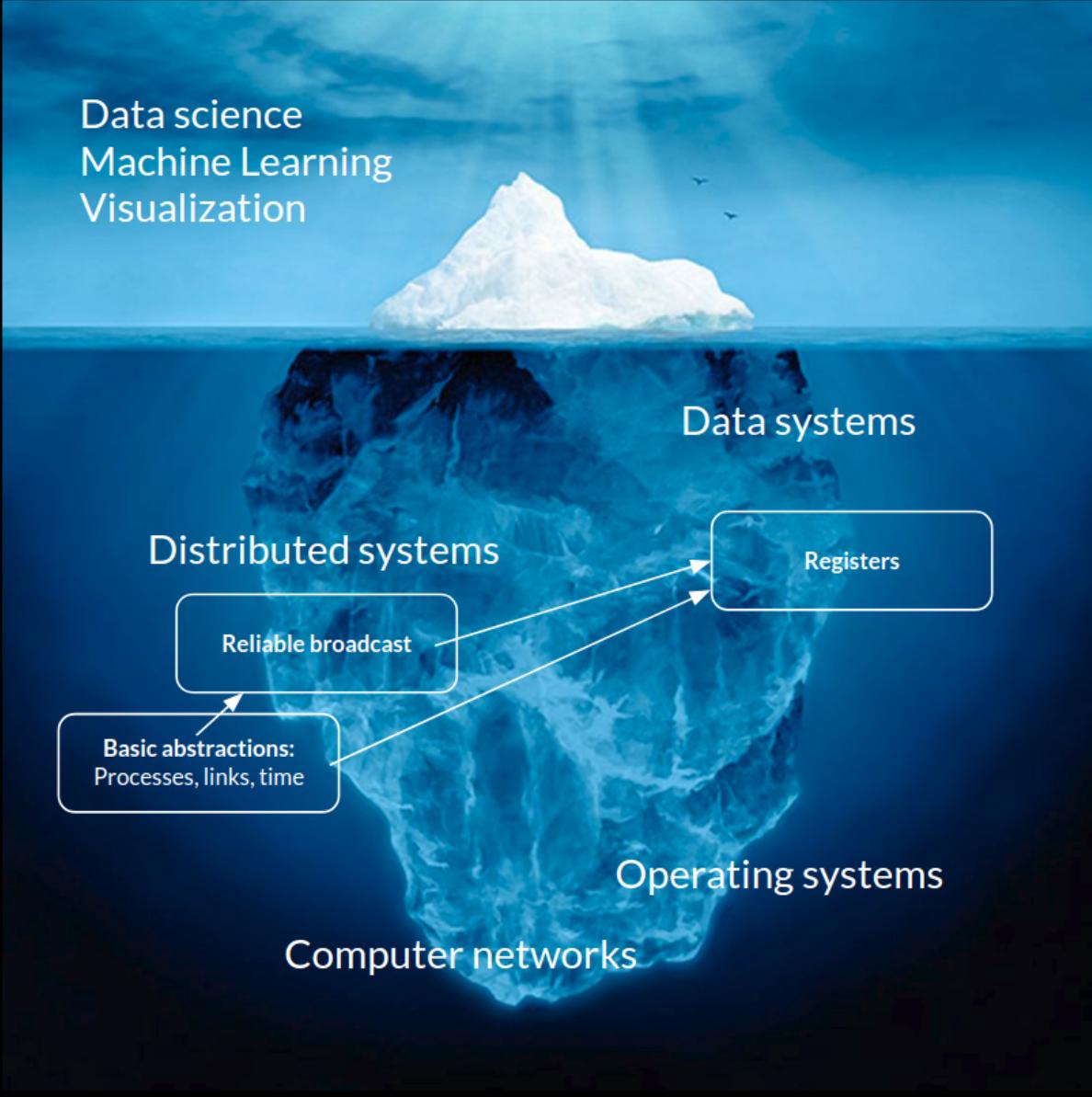


# Today



*Towards a distributed file system.*

- How do you **share resources** between processes?
- Can we build the **illusion of single storage**?
  - While replicating data for **fault-tolerance** and **scalability**?
  - While maintaining **consistency**?



Data science  
Machine Learning  
Visualization

Data systems

Distributed systems

Registers

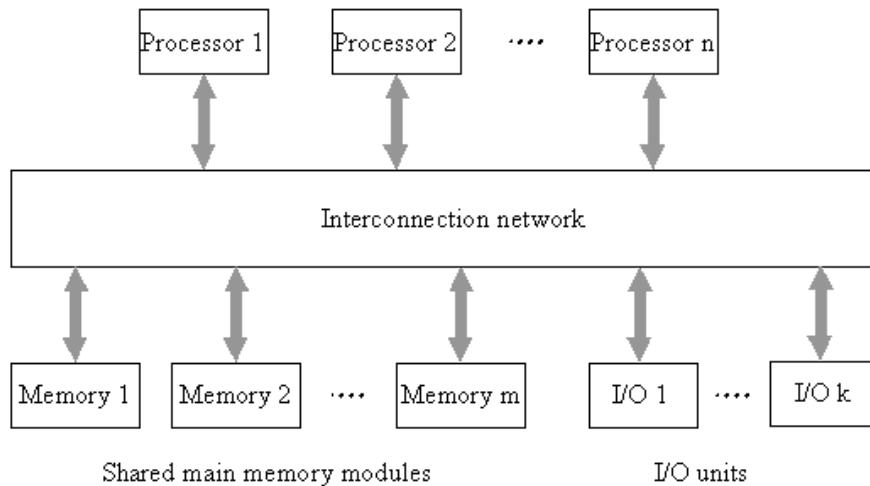
Reliable broadcast

Basic abstractions:  
Processes, links, time

Operating systems

Computer networks

# Shared memory



In a multiprocessor machine, processors typically communicate through **shared memory** provided at the hardware level (e.g., shared blocks of RAM that can be accessed by distinct CPUs).

- Shared memory can be viewed as an **array of registers** to which processors can **read** or **write**.
- Shared memory systems are **easy to program** since all processors share a single view of the data.

## Shared memory emulation

We want to **simulate** a **shared memory abstraction** in a distributed system, on top of message passing communication.

- Enable shared memory algorithms without being aware that processes are actually communicating by exchanging messages.
  - This is often much easier to program.
- Equivalent to **consistent data replication** across nodes.

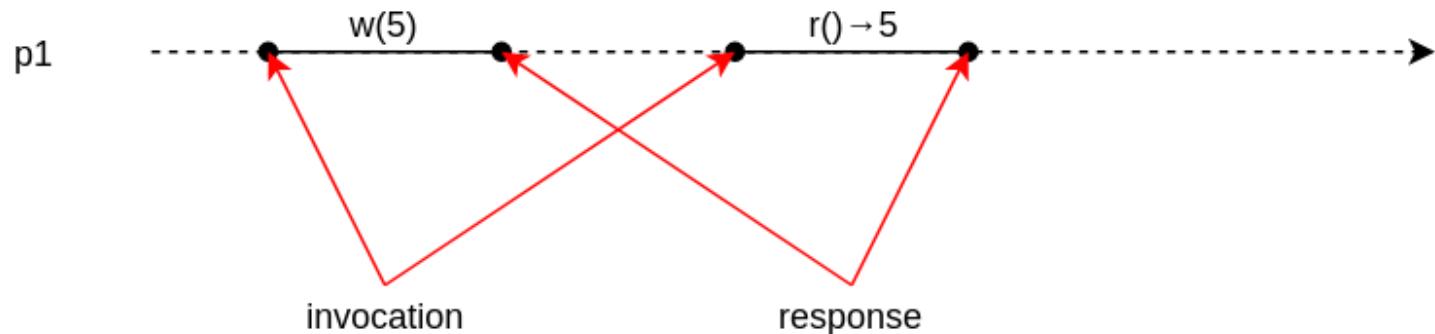
## Data replication

- Shared data allows to:
  - Reduce network traffic
  - Promote increased parallelism
  - Be robust against failures
  - Result in fewer page faults
- Applications:
  - distributed databases
  - distributed file systems
  - distributed cache
- Challenges:
  - Consistency in presence of **failures**.
  - Consistency in presence of **concurrency**.

# Regular registers

# Read/Write registers

- A **register** represents each memory location.
- A register contains only positive integers and is initialized to **0**.
- Registers have two **operations**:
  - **read()**: return the current value of the register.
  - **write( $v$ )**: update the register to value  $v$ .
- An operation is **not instantaneous**:
  - It is first **invoked** by the calling process.
  - It computes for some time.
  - It returns a **response** upon completion.



## Definitions

- In an execution, an operation is
  - completed if both invocation and response occurred.
  - failed if invoked but no response was received.
- Operation  $o_1$  precedes  $o_2$  if response of  $o_1$  precedes the invocation of  $o_2$ .
- Operations  $o_1$  and  $o_2$  are concurrent if neither precedes the other.
- $(1, N)$ -register: 1 designated writer,  $N$  readers.
- $(M, N)$ -register:  $M$  writers,  $N$  readers.

# Regular registers (*onrr*)

**Module:**

**Name:**  $(1, N)$ -RegularRegister, instance *onrr*.

**Events:**

**Request:**  $\langle \text{onrr}, \text{Read} \rangle$ : Invokes a read operation on the register.

**Request:**  $\langle \text{onrr}, \text{Write} \mid v \rangle$ : Invokes a write operation with value  $v$  on the register.

**Indication:**  $\langle \text{onrr}, \text{ReadReturn} \mid v \rangle$ : Completes a read operation on the register with return value  $v$ .

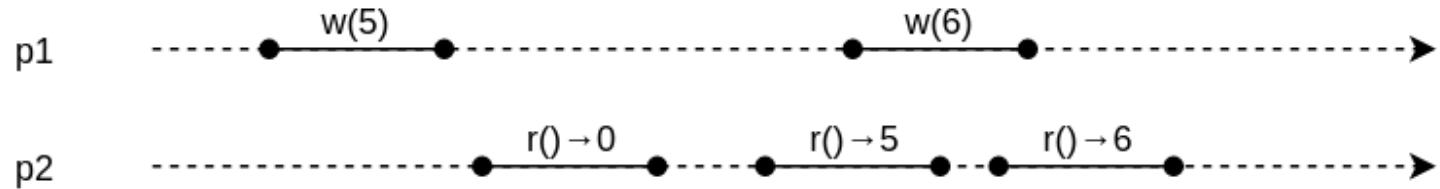
**Indication:**  $\langle \text{onrr}, \text{WriteReturn} \rangle$ : Completes a write operation on the register.

**Properties:**

**ONRR1: Termination:** If a correct process invokes an operation, then the operation eventually completes.

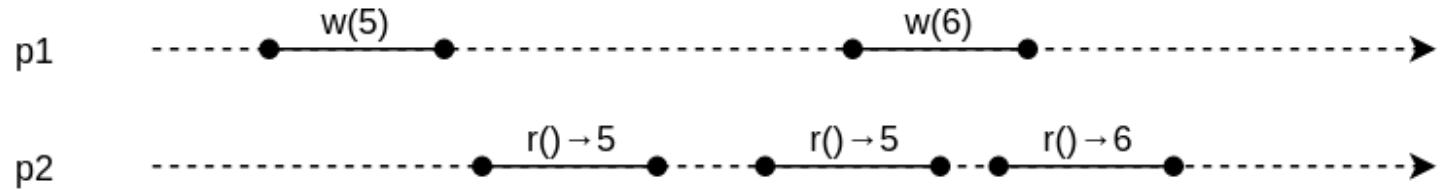
**ONRR2: Validity:** A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.

## Regular register example (1)



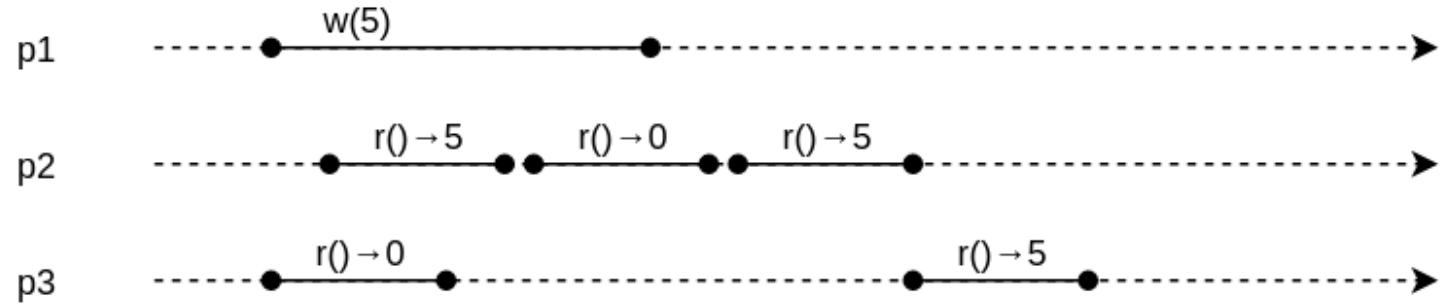
[Q] Regular or non-regular?

## Regular register example (2)



[Q] Regular or non-regular?

### Regular register example (3)



[Q] Regular or non-regular?

# Centralized algorithm

- Designates one process as the **leader**.
  - E.g., using the leader election abstraction (see Lecture 2).
- To **read()**:
  - Ask the leader for latest value.
- To **write( $v$ )**:
  - Update leader's value to  $v$ .

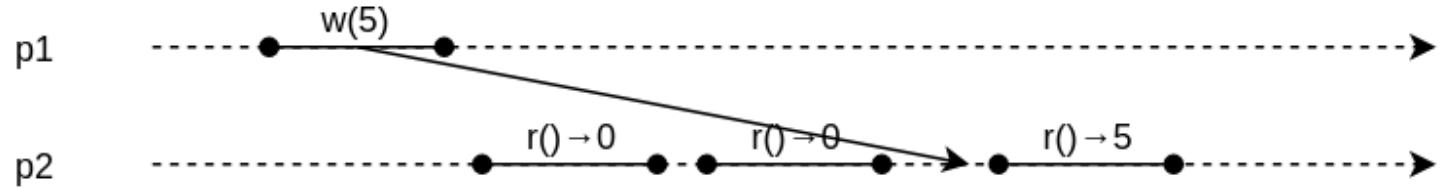
[Q] Problem? **Does not work if leader crashes!**

# Decentralized algorithm (bogus)

- Intuitively, make an algorithm in which
  - A `read()` reads the local value.
  - A `write( $v$ )` writes to all nodes.
- To `read()`:
  - Return local value.
- To `write( $v$ )`:
  - Update local value to  $v$ .
  - Broadcast  $v$  to all (each node then locally updates).
  - Return.

[Q] Problem?

## Decentralized algorithm (bogus) example



Validity is violated!

# Read-one Write-all algorithm

- Bogus algorithm modified.
- To `read()`:
  - Return local value.
- To `write( $v$ )`:
  - Update local value to  $v$ .
  - Broadcast  $v$  to all (each node locally updates).
  - Wait for acknowledgement from all correct nodes.
    - Require a perfect failure detector (fail-stop).
  - Return.

**Implements:**

(1,  $N$ )-RegularRegister, **instance**  $onrr$ .

**Uses:**

BestEffortBroadcast, **instance**  $beb$ ;  
PerfectPointToPointLinks, **instance**  $pl$ ;  
PerfectFailureDetector, **instance**  $\mathcal{P}$ .

```
upon event < onrr, Init > do
    val := ⊥;
    correct := Π;
    writeset := ∅;

upon event < P, Crash | p > do
    correct := correct \ {p};

upon event < onrr, Read > do
    trigger < onrr, ReadReturn | val >;

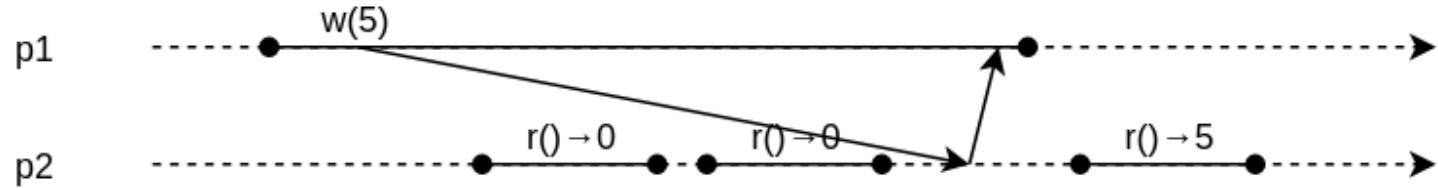
upon event < onrr, Write | v > do
    trigger < beb, Broadcast | [WRITE, v] >;

upon event < beb, Deliver | q, [WRITE, v] > do
    val := v;
    trigger < pl, Send | q, ACK >;

upon event < pl, Deliver | p, ACK > do
    writeset := writeset ∪ {p};

upon correct ⊆ writeset do
    writeset := ∅;
    trigger < onrr, WriteReturn >;
```

## Read-one Write-all example



Validity is no longer violated because the write response has been postponed.

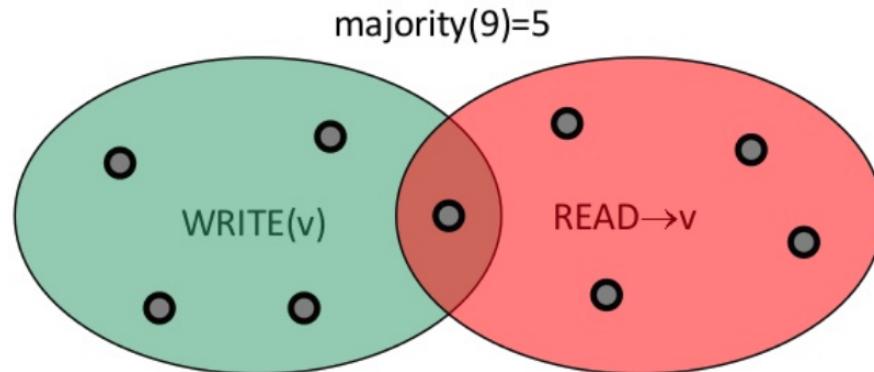
# Quorum principle

Can we implement a regular register in fail-silent? (without a failure detector)

# Quorum principle

Can we implement a regular register in fail-silent? (without a failure detector)

- Assume a majority of correct nodes.
- Divide the system into two overlapping **majority quorums**.
  - i.e., each quorum counts at least  $\lfloor \frac{N}{2} \rfloor + 1$  nodes.
- Always write to and read from a majority of nodes.
- At least one node must know the most recent value.



# Majority voting algorithm

**Implements:**

(1,  $N$ )-RegularRegister, **instance**  $onrr$ .

**Uses:**

BestEffortBroadcast, **instance**  $beb$ ;

PerfectPointToPointLinks, **instance**  $pl$ .

```
upon event < onrr, Init > do
     $(ts, val) := (0, \perp)$ ;
     $wts := 0$ ;
     $acks := 0$ ;
     $rid := 0$ ;
     $readlist := [\perp]^N$ ;

upon event < onrr, Write |  $v$  > do
     $wts := wts + 1$ ;
     $acks := 0$ ;
    trigger < beb, Broadcast | [WRITE,  $wts, v$ ] >

upon event < beb, Deliver |  $p$ , [WRITE,  $ts', v'$ ] > do
    if  $ts' > ts$  then
         $(ts, val) := (ts', v')$ ;
        trigger < pl, Send |  $p$ , [ACK,  $ts'$ ] >

upon event < pl, Deliver |  $q$ , [ACK,  $ts'$ ] > such that  $ts' = wts$  do
     $acks := acks + 1$ ;
    if  $acks > N/2$  then
         $acks := 0$ ;
        trigger < onrr, WriteReturn >
```

```

upon event < onrr, Read > do
  rid := rid + 1;
  readlist := [⊥]N;
  trigger < beb, Broadcast | [READ, rid] >;

upon event < beb, Deliver | p, [READ, r] > do
  trigger < pl, Send | p, [VALUE, r, ts, val] >;

upon event < pl, Deliver | q, [VALUE, r, ts', v'] > such that r = rid do
  readlist[q] := (ts', v');
  if #(readlist) > N/2 then
    v := highestval(readlist);
    readlist := [⊥]N;
    trigger < onrr, ReadReturn | v >;

```

[Q] Why resetting acks and readlist right after having received back just more than  $N/2$  messages?

# Atomic registers

Towards single storage illusion.

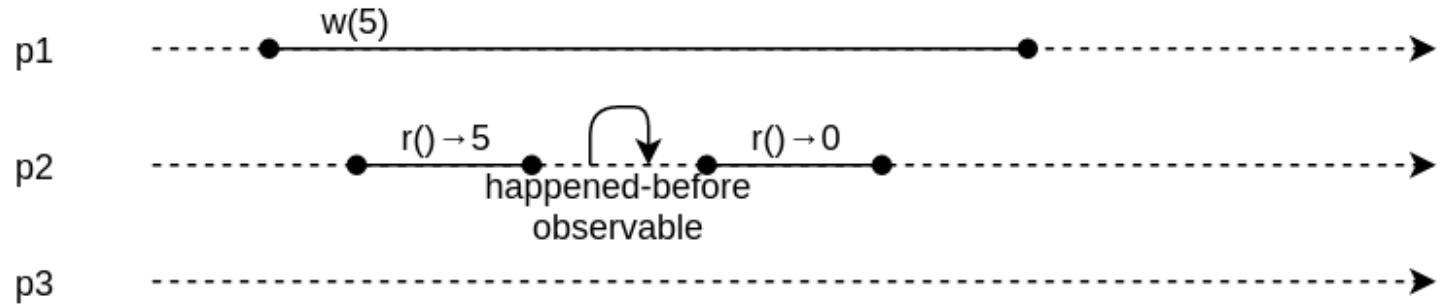
# Sequential consistency

An operation  $o_1$  locally precedes  $o_2$  in  $E$  if  $o_1$  and  $o_2$  occur at the same node and  $o_1$  precedes  $o_2$  in  $E$ .

An execution  $E$  is **sequentially consistent** if an execution  $F$  exists such that:

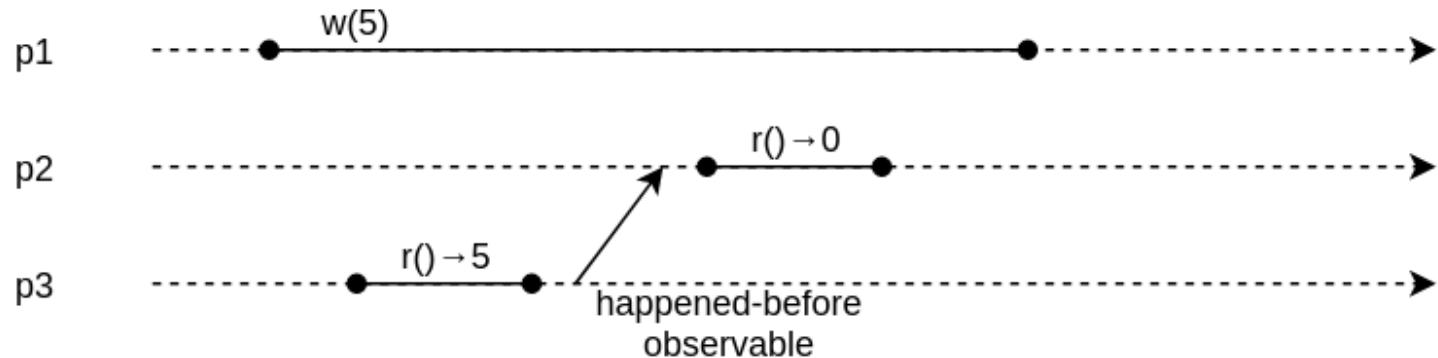
- $E$  and  $F$  contain the same events;
- $F$  is sequential;
- Read responses have value of the preceding write invocation in  $F$ ;
- If  $o_1$  locally precedes  $o_2$  in  $E$ , then  $o_1$  locally precedes  $o_2$  in  $F$ .

## Example (1)



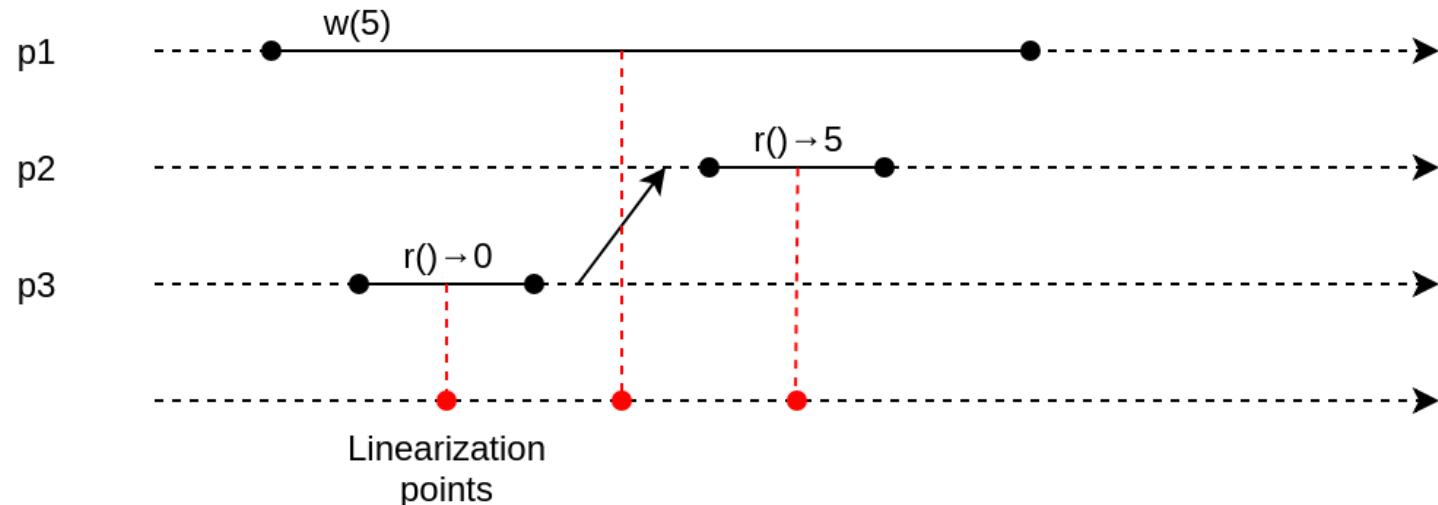
Sequential consistency **disallows** such execution.

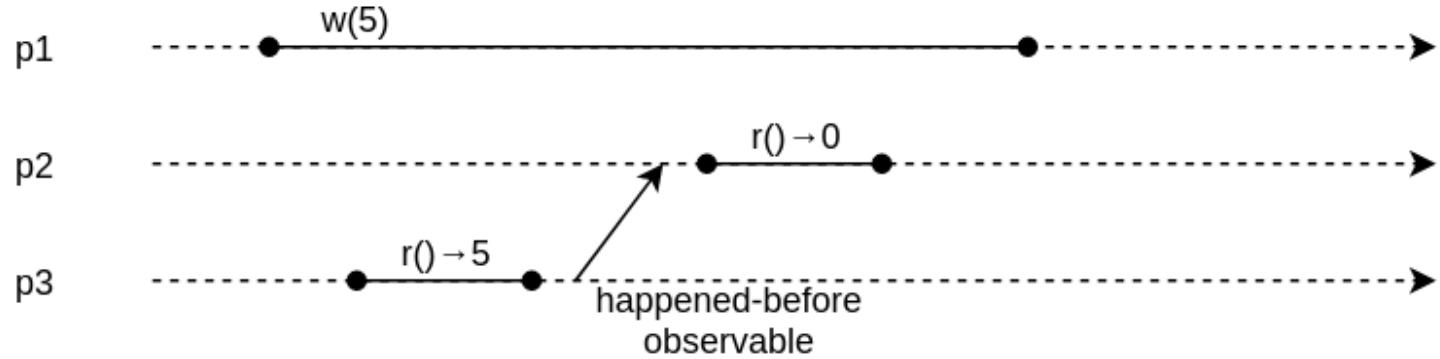
## Example (2)



Sequential consistency **allows** such execution.

# Linearization





This execution is sequentially consistent but is not linearizable!

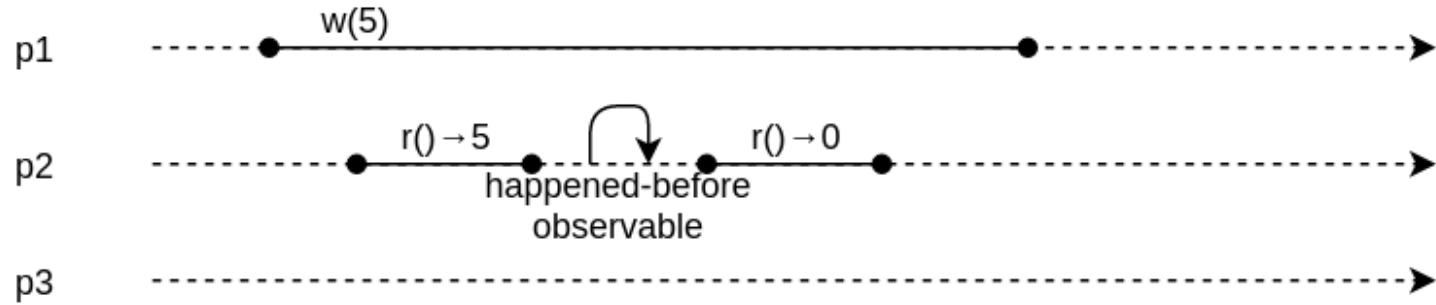
- **Linearizability:**

- Read operations appear as if **immediately** happened at all nodes at time between invocation and response.
- Write operations appear as if **immediately** happened at all nodes at time between invocation and response.
- Failed operations appear as
  - completed at every node, XOR
  - never occurred at any node.
- The hypothetical serial execution is called a **linearization** of the actual execution.

- **Termination:**

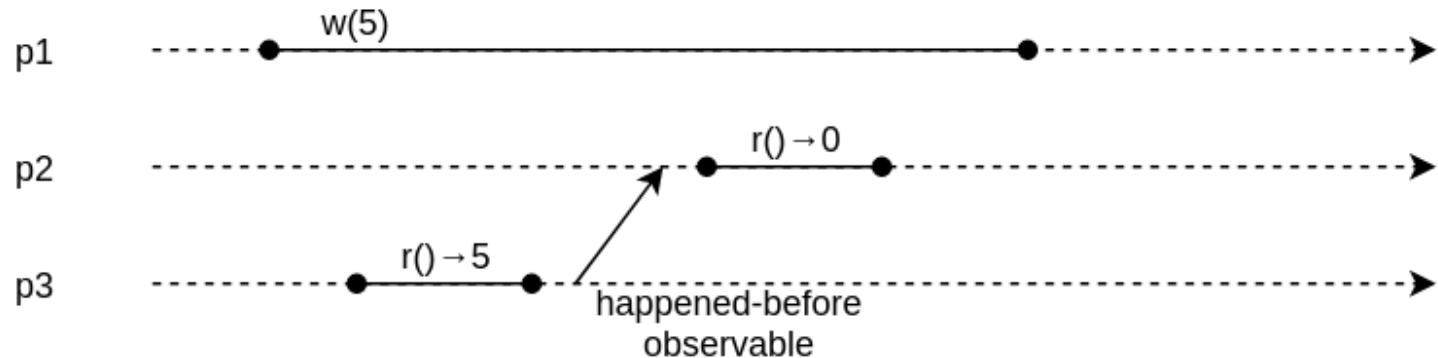
- If node is correct, each read and write operation eventually completes.

## Example (1)



Linearizability **disallows** such execution.

## Example (2)



Linearizability **disallows** such execution.

# $(1, N)$ atomic registers

**Module:**

**Name:**  $(1, N)$ -AtomicRegister, instance  $onar$ .

**Events:**

**Request:**  $\langle onar, Read \rangle$ : Invokes a read operation on the register.

**Request:**  $\langle onar, Write \mid v \rangle$ : Invokes a write operation with value  $v$  on the register.

**Indication:**  $\langle onar, ReadReturn \mid v \rangle$ : Completes a read operation on the register with return value  $v$ .

**Indication:**  $\langle onar, WriteReturn \rangle$ : Completes a write operation on the register.

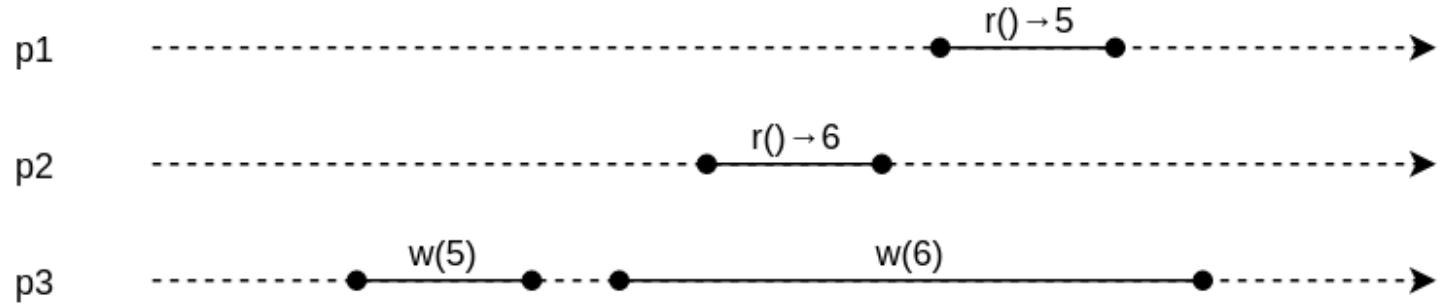
**Properties:**

**ONAR1–ONAR2:** Same as properties ONRR1–ONRR2 of a  $(1, N)$  regular register (Module 4.1).

**ONAR3:** *Ordering:* If a read returns a value  $v$  and a subsequent read returns a value  $w$ , then the write of  $w$  does not precede the write of  $v$ .

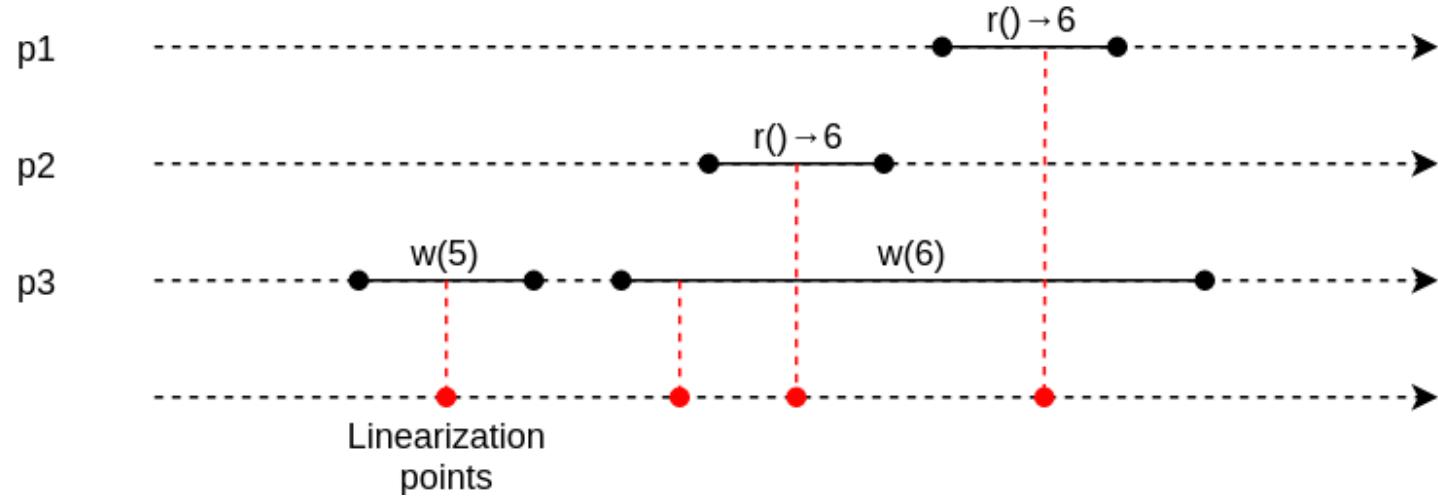
[Q] Show that linearizability is equivalent to validity + ordering.

## Atomic register example (1)



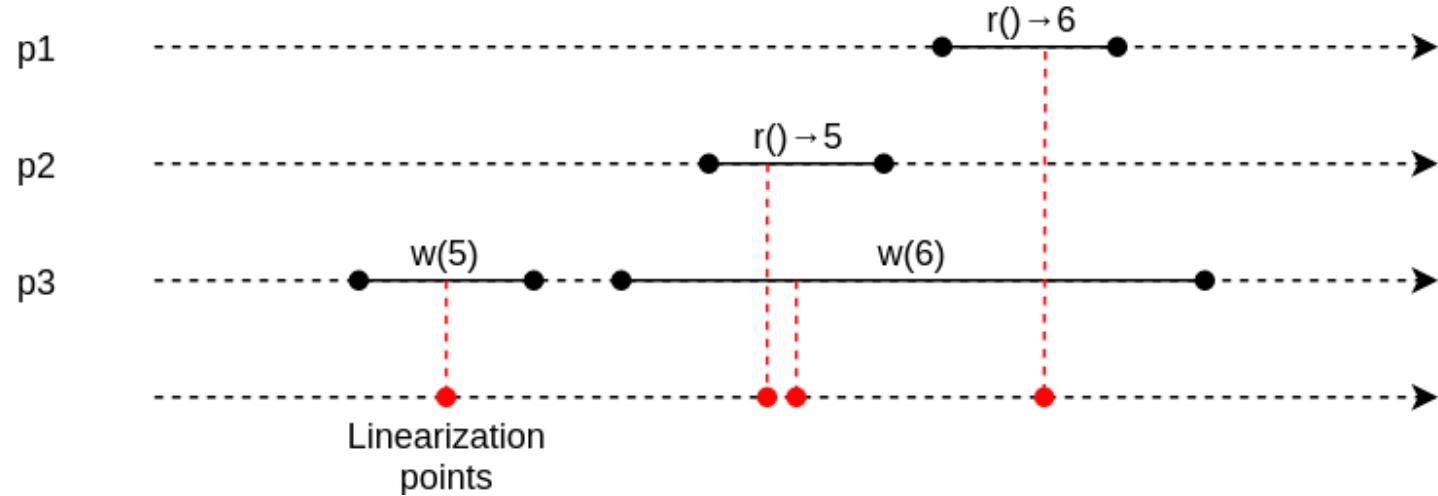
[Q] Atomic? **No**, not possible to find linearization points.

## Atomic register example (2)



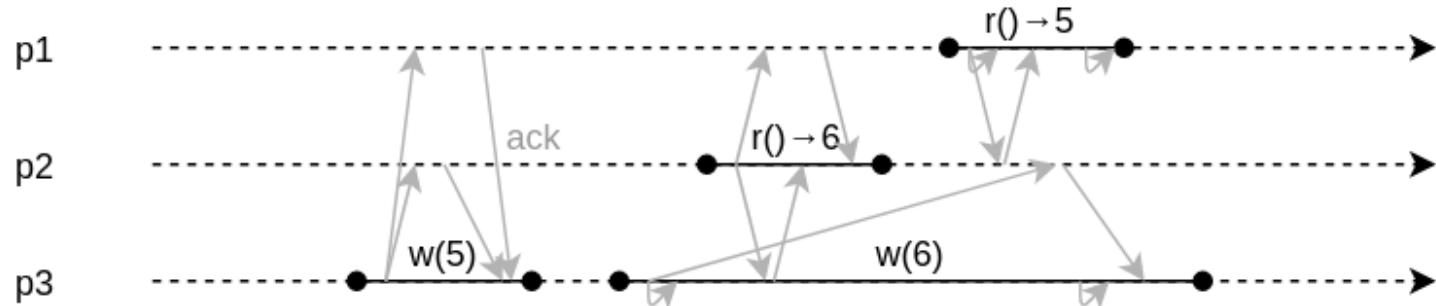
[Q] Atomic? Yes

## Atomic register example (3)



[Q] Atomic? Yes

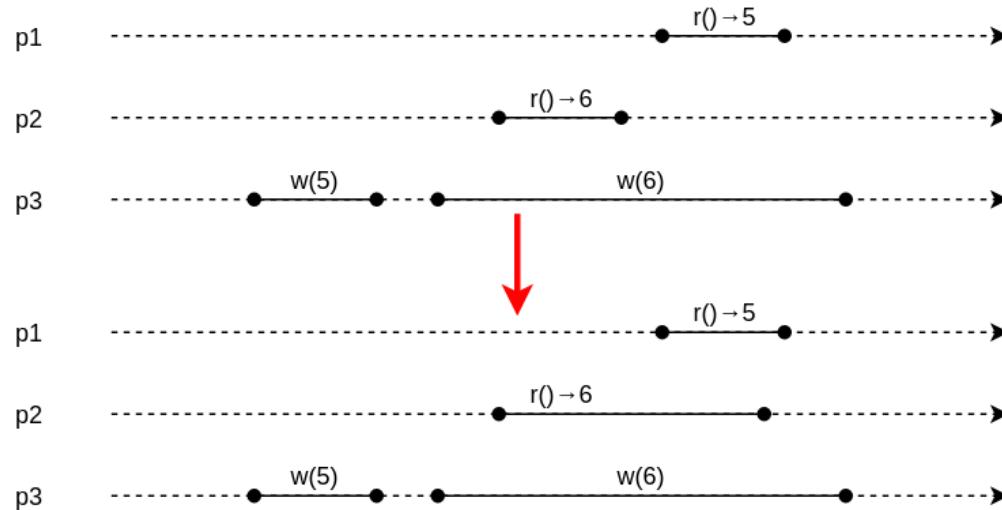
## Regular but not atomic



[Q] Atomic? **No**. Regular? **Yes**, using majority voting.

# Implementation of $(1, N)$ atomic registers

- When reading, write back the value that is about to be returned.
- Maintain a local timestamp  $ts$  and its associated value  $val$ .
- Overwrite the local pair only upon a write operation of a more recent value.



# Read-Impose Write-all algorithm

**Implements:**

(1,  $N$ )-AtomicRegister, **instance**  $onar$ .

**Uses:**

BestEffortBroadcast, **instance**  $beb$ ;  
PerfectPointToPointLinks, **instance**  $pl$ ;  
PerfectFailureDetector, **instance**  $\mathcal{P}$ .

**upon event**  $\langle onar, Init \rangle$  **do**

$(ts, val) := (0, \perp)$ ;

$correct := \Pi$ ;

$writeset := \emptyset$ ;

$readval := \perp$ ;

$reading := \text{FALSE}$ ;

**upon event**  $\langle \mathcal{P}, Crash \mid p \rangle$  **do**

$correct := correct \setminus \{p\}$ ;

**upon event**  $\langle onar, Read \rangle$  **do**

$reading := \text{TRUE}$ ;

$readval := val$ ;

**trigger**  $\langle beb, Broadcast \mid [\text{WRITE}, ts, val] \rangle$ ;

```

upon event < onar, Write | v > do
  trigger < beb, Broadcast | [WRITE, ts + 1, v] >;
upon event < beb, Deliver | p, [WRITE, ts', v'] > do
  if ts' > ts then
    (ts, val) := (ts', v');
  trigger < pl, Send | p, [ACK] >;
upon event < pl, Deliver | p, [ACK] > then
  writeset := writeset ∪ {p};
upon correct ⊆ writeset do
  writeset := ∅;
  if reading = TRUE then
    reading := FALSE;
    trigger < onar, ReadReturn | readval >;
  else
    trigger < onar, WriteReturn >;

```

[Q] How to adapt to fail-silent? **Read-Impose Write-Majority**

# Correctness

- **Ordering:** if a read returns  $v$  and a subsequent read returns  $w$ , then the write of  $w$  does not precede the write of  $v$ .
  - $p$  writes  $v$  with timestamp  $ts_v$ .
  - $p$  writes  $w$  with timestamp  $ts_w > ts_v$ .
  - $q$  reads the values the  $w$ .
  - some time later,  $r$  invokes a read operation.
  - when  $q$  completes its read, all correct processes have a timestamp  $ts \geq ts_w$ .
  - there is no way for  $r$  to change its value back to  $v$  after this because  $ts_v < ts_w$ .

[Q] Show that the termination and validity properties are satisfied.

# $(N, N)$ atomic registers

**Module:**

**Name:**  $(N, N)$ -AtomicRegister, instance  $nmar$ .

**Events:**

**Request:**  $\langle nmar, \text{Read} \rangle$ : Invokes a read operation on the register.

**Request:**  $\langle nmar, \text{Write} \mid v \rangle$ : Invokes a write operation with value  $v$  on the register.

**Indication:**  $\langle nmar, \text{ReadReturn} \mid v \rangle$ : Completes a read operation on the register with return value  $v$ .

**Indication:**  $\langle nmar, \text{WriteReturn} \rangle$ : Completes a write operation on the register.

**Properties:**

**NNAR1:** *Termination*: Same as property ONAR1 of a  $(1, N)$  atomic register (Module 4.2).

**NNAR2:** *Atomicity*: Every read operation returns the value that was written most recently in a hypothetical execution, where every failed operation appears to be complete or does not appear to have been invoked at all, and every complete operation appears to have been executed at some instant between its invocation and its completion.

- How do we handle **multiple writers**?
- Read-Impose Write-all does not support multiple writers:
  - Assume  $p$  and  $q$  both store the same timestamp  $ts$  (e.g., because of a preceding completed operation).
  - When  $p$  and  $q$  proceed to write, different values would become associated with the same timestamp.
- Fix:
  - Together with the timestamp, pass and store the identity  $pid$  of the process that writes a value  $v$ .
  - Determine which message is the latest
    - by comparing timestamps,
    - by breaking ties using the process IDs.

[Q] How many messages are exchanged per read and write operations?

[Q] Can we similarly fix Read-Impose Write-Majority?

# Simulating message passing?

- As we saw, we can simulate shared with message passing.
  - A majority of correct nodes is all that is needed.
- Can we **simulate message passing** in shared memory?
  - Yes: use one register  $pq$  for every channel.
    - Modeling a directed channel from  $p$  to  $q$ .
  - Send messages by appending to the right channel.
  - Receive messages by busy-polling incoming "channels".
- Shared memory and message passing are **equivalent**.

# Summary

- Shared memory registers form a **shared memory abstraction** with read and write operations.
  - Consistency of the data is guaranteed, even in the presence of failures and concurrency.
- Regular registers:
  - Bogus algorithm (does not work)
  - Centralized algorithm (if no failures)
  - Read-One Write-All algorithm (fail-stop)
  - Majority voting (fail-silent)
- Atomic registers:
  - Single writers
  - Multiple writers



# Large-scale Data Systems

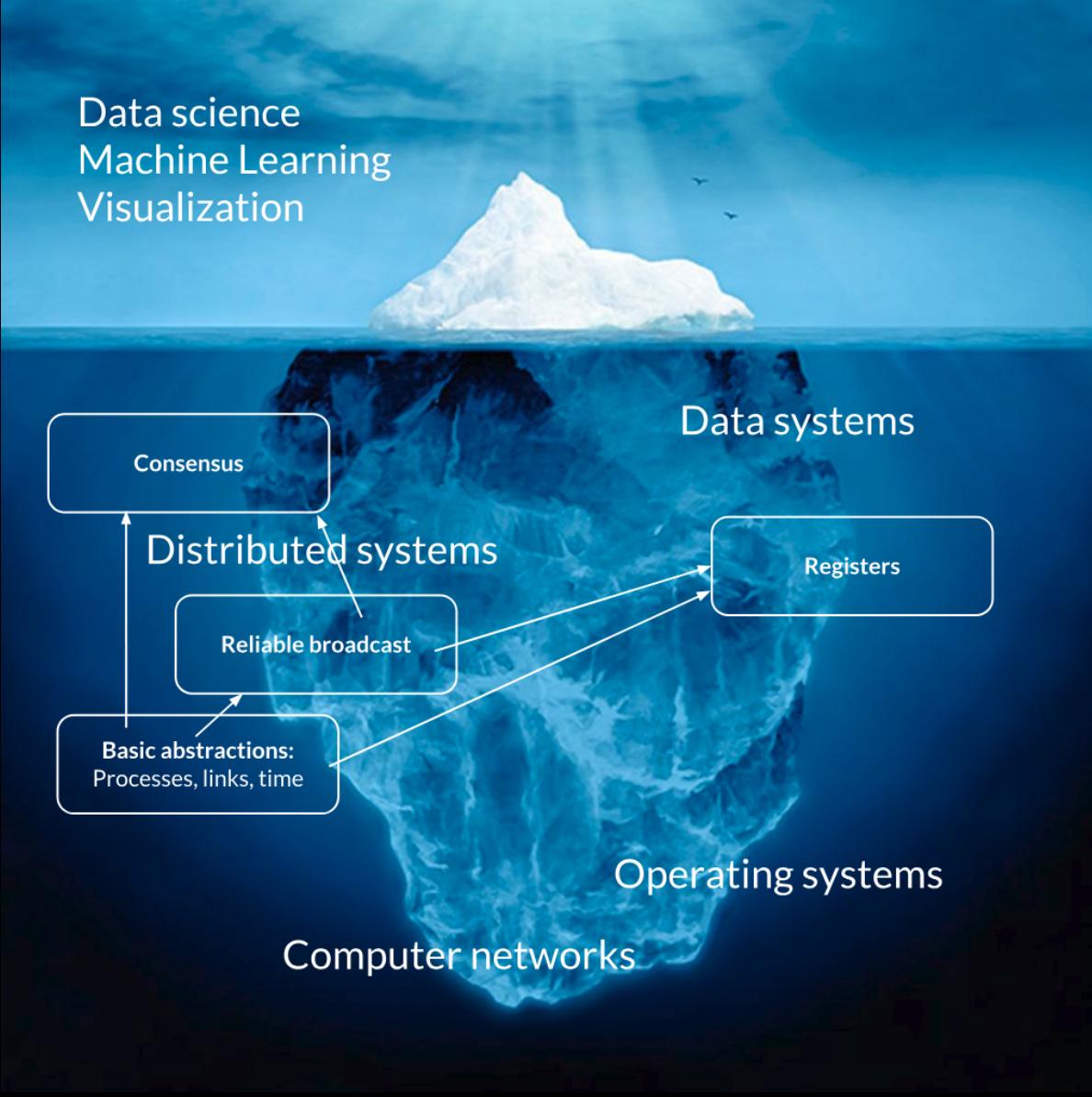
Lecture 5: Consensus

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)



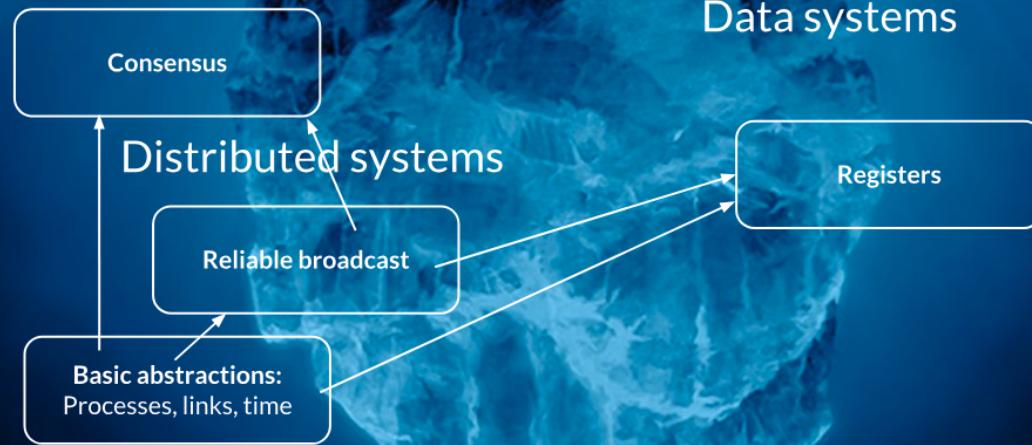
# Today

- Most important abstraction in distributed systems: consensus.
- Builds upon broadcast and failure detectors.
- From consensus, we will build:
  - total order broadcast
  - replicated state machines
  - ... and almost all higher level distributed fault-tolerant applications!



Data science  
Machine Learning  
Visualization

Data systems

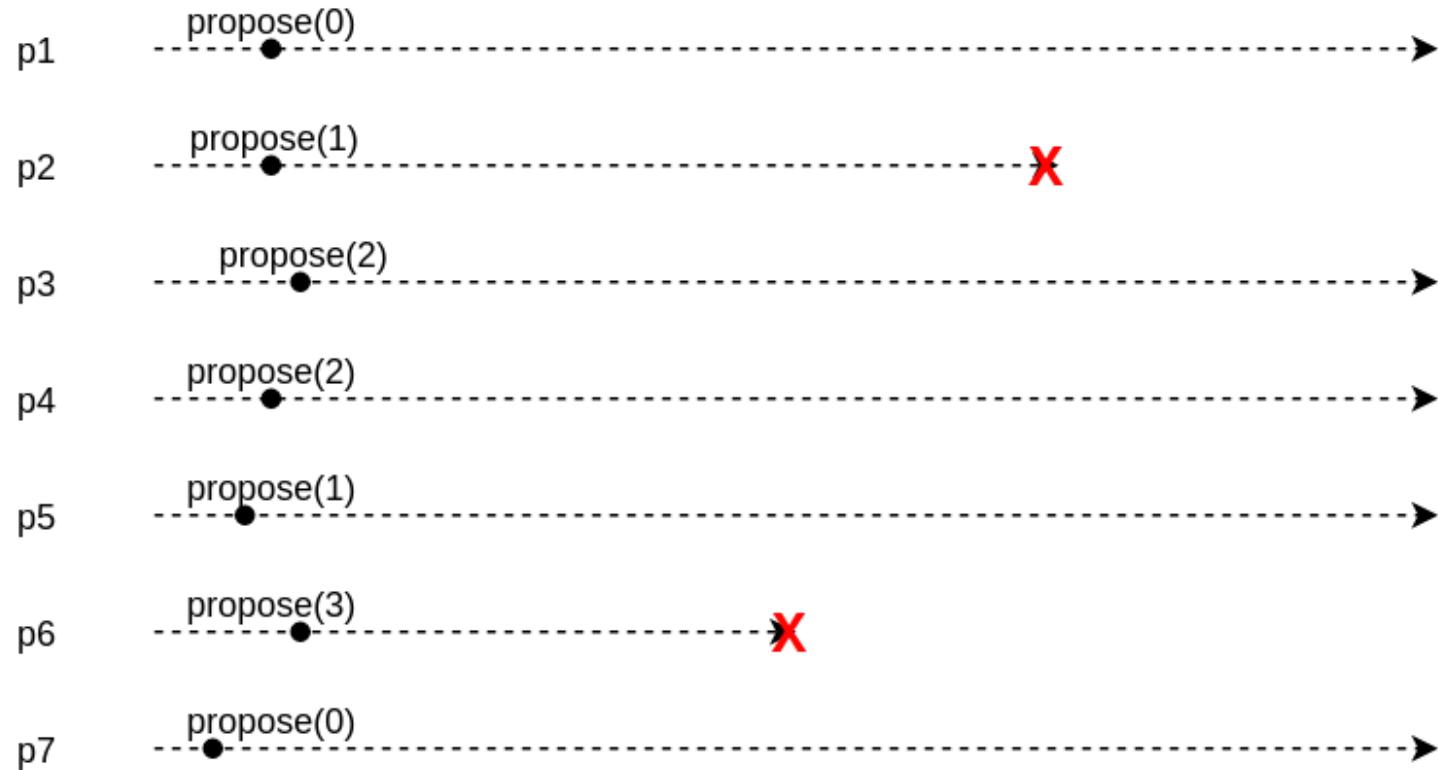


Operating systems

Computer networks

# Consensus

**Consensus** is the problem of making processes all **agree** on one of the values they propose.



*How do we reach an agreement?*

## Motivation

- Solving consensus is **key** to solving many problems in distributed computing:
  - synchronizing replicated state machines;
  - electing a leader;
  - managing group membership;
  - deciding to commit or abort distributed transactions.
- Any algorithm that helps multiple processes **maintain common state** or to **decide on a future action**, in a model where processes may fail, involves **solving a consensus problem**.

# Consensus

**Module:**

**Name:** Consensus, instance  $c$ .

**Events:**

**Request:**  $\langle c, \text{Propose} \mid v \rangle$ : Proposes value  $v$  for consensus.

**Indication:**  $\langle c, \text{Decide} \mid v \rangle$ : Outputs a decided value  $v$  of consensus.

**Properties:**

**C1: Termination:** Every correct process eventually decides some value.

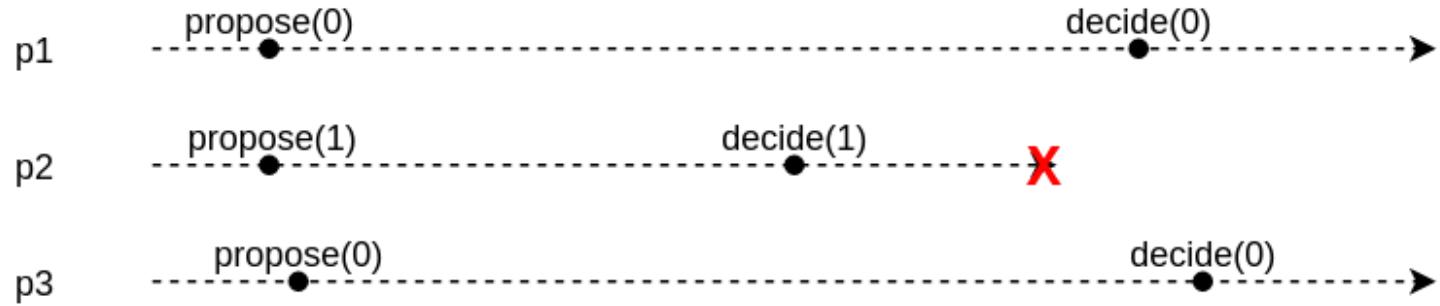
**C2: Validity:** If a process decides  $v$ , then  $v$  was proposed by some process.

**C3: Integrity:** No process decides twice.

**C4: Agreement:** No two correct processes decide differently.

[Q] Which is safety, which is liveness?

## Sample execution



[Q] Does this satisfy consensus?

# Uniform consensus

**Module:**

**Name:** UniformConsensus, instance  $uc$ .

**Events:**

**Request:**  $\langle uc, Propose \mid v \rangle$ : Proposes value  $v$  for consensus.

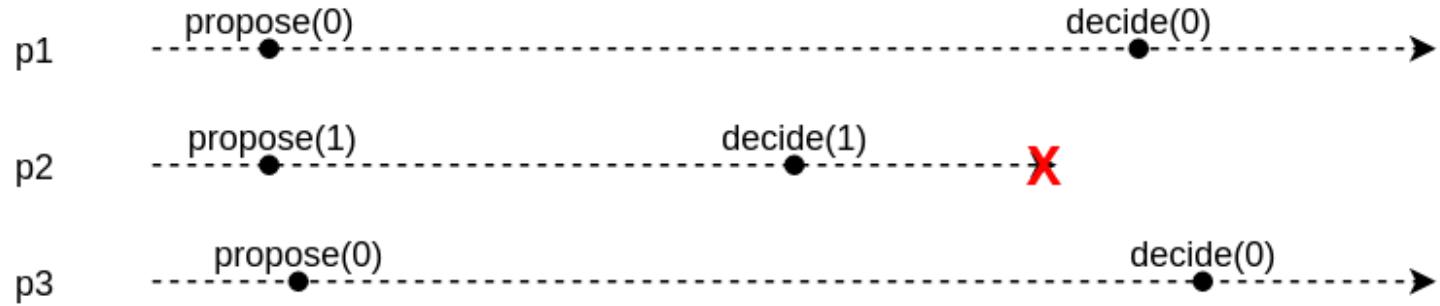
**Indication:**  $\langle uc, Decide \mid v \rangle$ : Outputs a decided value  $v$  of consensus.

**Properties:**

**UC1–UC3:** Same as properties C1–C3 in (regular) consensus (Module [5.1](#)).

**UC4:** *Uniform agreement:* No two processes decide differently.

## Sample execution



[Q] Does this satisfy uniform consensus?

# **Impossibility of Distributed Consensus with One Faulty Process**

**MICHAEL J. FISCHER**

*Yale University, New Haven, Connecticut*

**NANCY A. LYNCH**

*Massachusetts Institute of Technology, Cambridge, Massachusetts*

**AND**

**MICHAEL S. PATERSON**

*University of Warwick, Coventry, England*

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

So, are we done? **No!**

- The FLP impossibility result holds for **asynchronous systems** only.
- Consensus can be implemented in **synchronous** and **partially synchronous** systems. (We will prove it!)
- The result only states that termination cannot be guaranteed. Can we have other guarantees while maintaining a high probability of termination?

# **Consensus in fail-stop**

# Hierarchical consensus

## Asumptions

- Assume a **perfect failure detector** (synchronous system).
- Assume processes  $1, \dots, N$  form an ordered **hierarchy** as given by a  $\text{rank}(p)$  function.
  - $\text{rank}(p)$  is a **unique** number between  $1$  and  $N$  (e.g., the pid).

## Algorithm

- Hierarchical consensus ensures that the correct process with **the lowest rank imposes its value** on all the other processes.
  - If  $p$  is correct and rank 1, it imposes its values on all other processes by broadcasting its proposal.
  - If  $p$  crashes immediately and  $q$  is correct and rank 2, then it ensures that  $q$ 's proposal is decided.
  - The core of the algorithm addresses the case where  $p$  is faulty but crashes after sending some of its proposal messages and  $q$  is correct.
- Hierarchical consensus works **in rounds**, with a rotating **leader**.
  - At round  $i$ , process  $p$  with rank  $i$  is the leader.
  - It decides its proposal and broadcasts it to all processes.
  - All other processes that reach round  $i$  wait before taking any actions, until they deliver this message or until they detect the crash of  $p$ .
    - upon which processes move to the next round.

**Implements:**

Consensus, instance  $c$ .

**Uses:**

BestEffortBroadcast, instance  $beb$ ;  
PerfectFailureDetector, instance  $\mathcal{P}$ .

```
upon event <  $c$ , Init > do
    detectedranks :=  $\emptyset$ ;
    round := 1;
    proposal :=  $\perp$ ; proposer := 0;
    delivered := [FALSE] $^N$ ;
    broadcast := FALSE;

upon event <  $\mathcal{P}$ , Crash |  $p$  > do
    detectedranks := detectedranks  $\cup$  {rank( $p$ )};

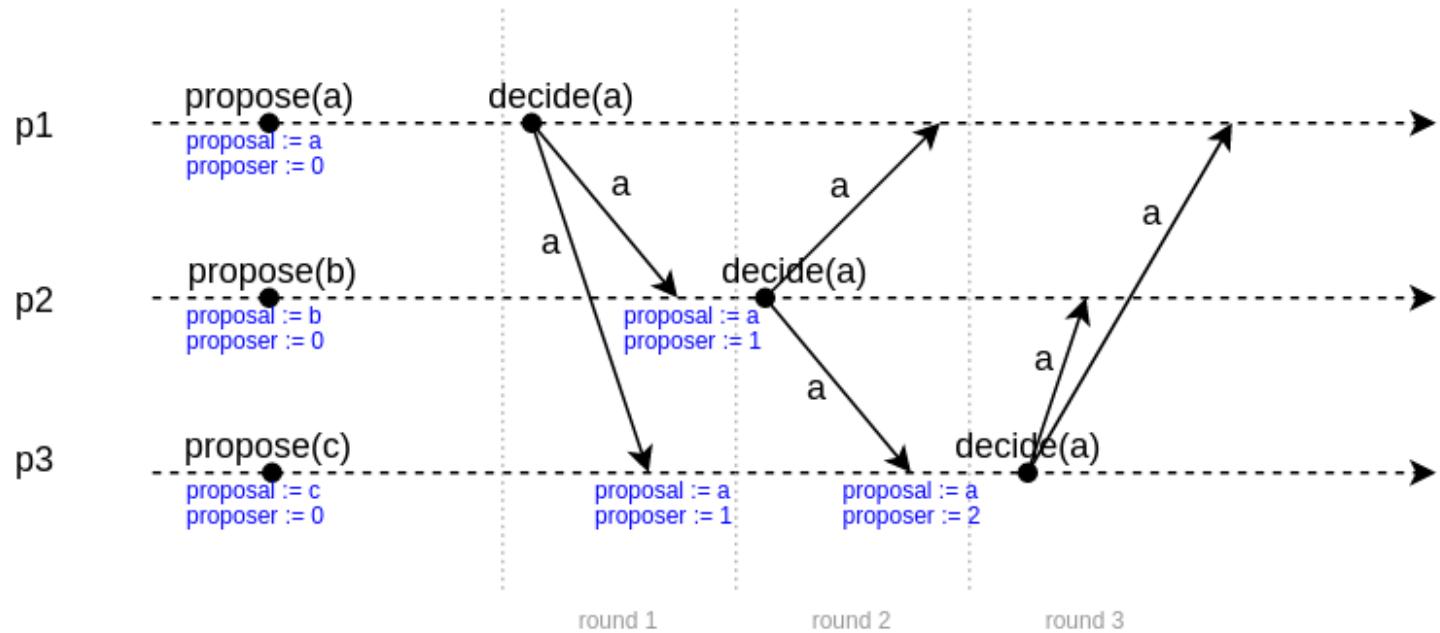
upon event <  $c$ , Propose |  $v$  > such that proposal =  $\perp$  do
    proposal :=  $v$ ;

upon round = rank(self)  $\wedge$  proposal  $\neq \perp$   $\wedge$  broadcast = FALSE do
    broadcast := TRUE;
    trigger <  $beb$ , Broadcast | [DECIDED, proposal] >;
    trigger <  $c$ , Decide | proposal >;

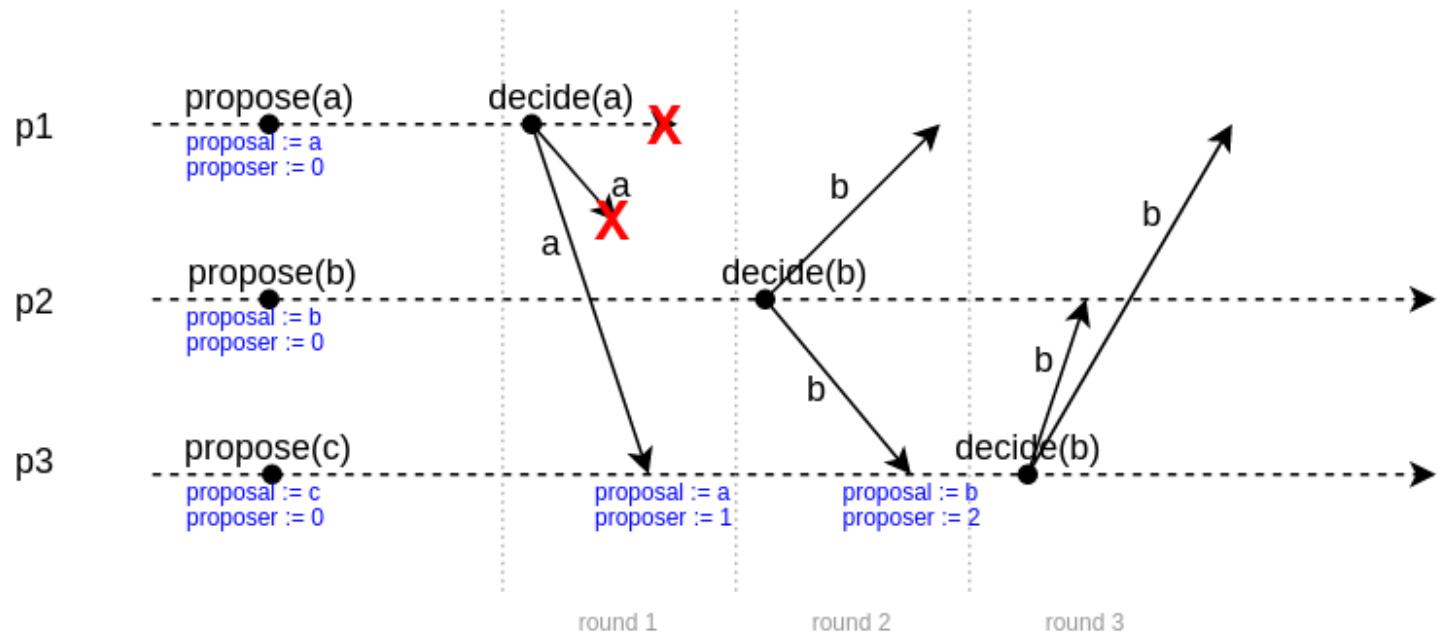
upon round  $\in$  detectedranks  $\vee$  delivered[round] = TRUE do
    round := round + 1;

upon event <  $beb$ , Deliver |  $p$ , [DECIDED,  $v$ ] > do
     $r$  := rank( $p$ );
    if  $r <$  rank(self)  $\wedge r >$  proposer then
        proposal :=  $v$ ;
        proposer :=  $r$ ;
        delivered[ $r$ ] := TRUE;
```

## Execution without failure



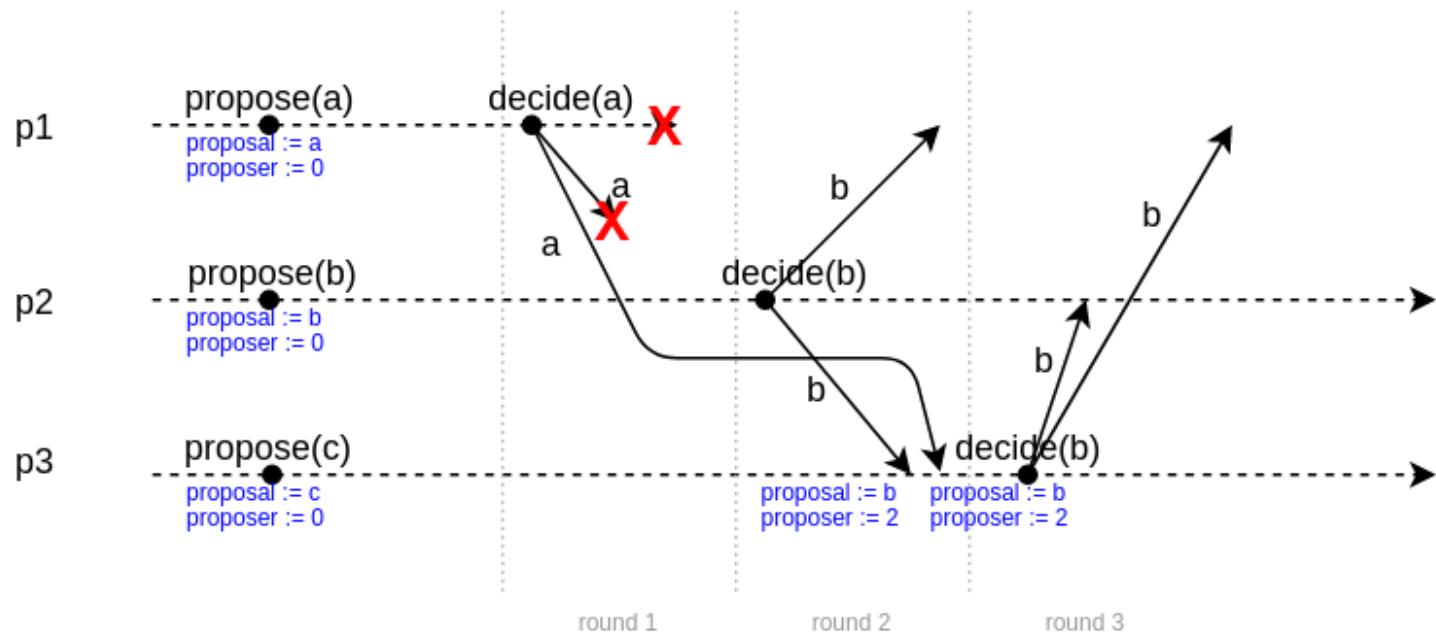
## Execution with failure (1)



[Q] Uniform consensus?

[Q] How many failures can be tolerated?

## Execution with failure (2)



## Correctness

- **Termination:** Every correct process eventually decides some value.
  - Every correct node makes it to the round it is leader in.
    - If some leader fails, completeness of the FD ensures progress.
    - If leader correct, validity of BEB ensures delivery.
- **Validity:** If a process decides  $v$ , then  $v$  was proposed by some process.
  - Always decide own proposal or adopted value.
- **Integrity:** No process decides twice.
  - Rounds increase monotonically.
  - A node only decides once in the round it is leader.
- **Agreement:** No two correct processes decide differently.
  - Take correct leader with minimum rank  $i$ .
    - By termination, it will decide  $v$ .
    - It will BEB  $v$ :
      - Every correct node gets  $v$  and adopts it.
      - No older proposals can override the adoption.
      - All future proposals and decisions will be  $v$ .

# Hierarchical uniform consensus

- Same as [hierarchical consensus](#).
- A round consists of **two communication steps**:
  - The leader BEB broadcasts its proposal
  - The leader collects acknowledgements
- Upon reception of all acknowledgements, **RB** broadcast the decision and decide at delivery.
  - This ensures that if a decision is made (at a faulty or correct process), then this decision will be made at all correct processes.
  - Processes proceed to the next round only if the current leader fails.

**Implements:**

UniformConsensus, **instance**  $uc$ .

**Uses:**

PerfectPointToPointLinks, **instance**  $pl$ ;  
BestEffortBroadcast, **instance**  $beb$ ;  
ReliableBroadcast, **instance**  $rb$ ;  
PerfectFailureDetector, **instance**  $\mathcal{P}$ .

```
upon event <  $uc$ , Init > do
    detectedranks :=  $\emptyset$ ;
    ackranks :=  $\emptyset$ ;
    round := 1;
    proposal :=  $\perp$ ; decision :=  $\perp$ ;
    proposed :=  $[\perp]^N$ ;

upon event <  $\mathcal{P}$ , Crash |  $p$  > do
    detectedranks := detectedranks  $\cup \{rank(p)\}$ ;

upon event <  $uc$ , Propose |  $v$  > such that  $proposal = \perp$  do
    proposal :=  $v$ ;

upon round = rank(self)  $\wedge$  proposal  $\neq \perp$   $\wedge$  decision =  $\perp$  do
    trigger <  $beb$ , Broadcast | [PROPOSAL, proposal] >

upon event <  $beb$ , Deliver |  $p$ , [PROPOSAL,  $v$ ] > do
    proposed[rank( $p$ )] :=  $v$ ;
    if rank( $p$ )  $\geq$  round then
        trigger <  $pl$ , Send |  $p$ , [ACK] >;
```

```

upon round  $\in$  detectedranks do
  if proposed[round]  $\neq \perp$  then
    proposal := proposed[round];
    round := round + 1;

upon event  $\langle pl, Deliver \mid q, [ACK] \rangle$  do
  ackranks := ackranks  $\cup \{rank(q)\}$ ;

upon detectedranks  $\cup$  ackranks =  $\{1, \dots, N\}$  do
  trigger  $\langle rb, Broadcast \mid [DECIDED, proposal] \rangle$ ;

upon event  $\langle rb, Deliver \mid p, [DECIDED, v] \rangle$  such that decision =  $\perp$  do
  decision := v;
  trigger  $\langle uc, Decide \mid decision \rangle$ ;

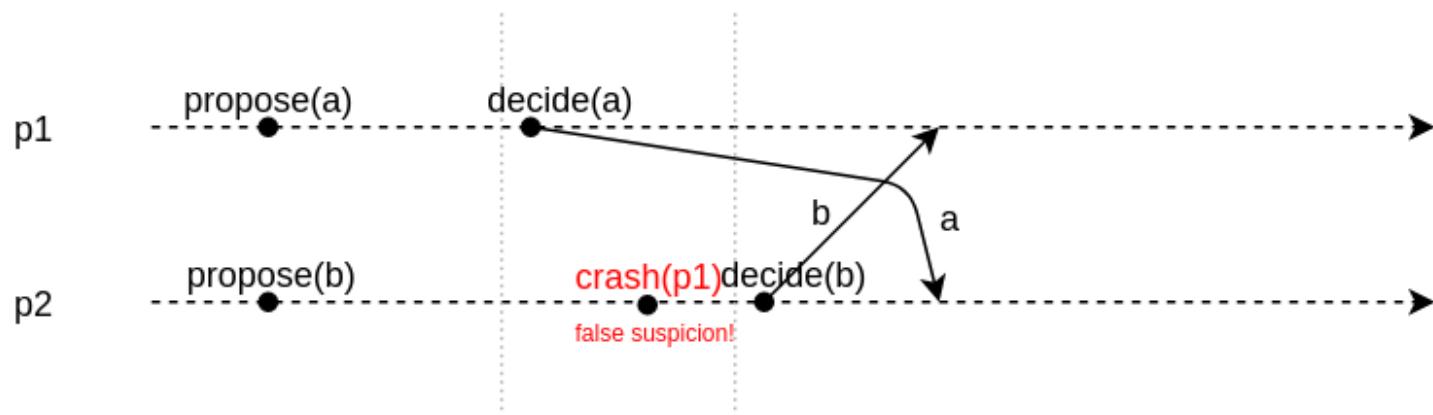
```

[Q] Why is reliable non-uniform broadcast sufficient to have uniform consensus?

# Consensus in fail-noisy

Would hierarchical consensus work with an [eventually perfect failure detector](#)?

- A false suspicion (i.e., a violation of strong accuracy) might lead to the **violation of agreement**.
- Not suspecting a crashed process (i.e., a violation of strong completeness) might lead to the **violation of termination**.



## Towards consensus...

We will build a consensus component in **fail-noisy** by combining three abstractions:

- an eventual leader detector
- an epoch-change abstraction
- an epoch consensus abstraction

# Eventual leader detector ( $\Omega$ )

**Module:**

**Name:** EventualLeaderDetector, **instance**  $\Omega$ .

**Events:**

**Indication:**  $\langle \Omega, Trust \mid p \rangle$ : Indicates that process  $p$  is trusted to be leader.

**Properties:**

**ELD1:** *Eventual accuracy*: There is a time after which every correct process trusts some correct process.

**ELD2:** *Eventual agreement*: There is a time after which no two correct processes trust different correct processes.

[Q] This abstraction can be implemented from an eventually perfect failure detector. How?

# Epoch-Change (*ec*)

- Let us define an **Epoch-Change** abstraction, whose purpose it is to signal a change of epoch corresponding to the election of a leader.
- An indication event **StartEpoch** contains:
  - an epoch timestamp *ts*
  - a leader process *l*.

**Module:**

**Name:** EpochChange, instance  $ec$ .

**Events:**

**Indication:**  $\langle ec, StartEpoch \mid ts, \ell \rangle$ : Starts the epoch identified by timestamp  $ts$  with leader  $\ell$ .

**Properties:**

**EC1: Monotonicity:** If a correct process starts an epoch  $(ts, \ell)$  and later starts an epoch  $(ts', \ell')$ , then  $ts' > ts$ .

**EC2: Consistency:** If a correct process starts an epoch  $(ts, \ell)$  and another correct process starts an epoch  $(ts', \ell')$  with  $ts = ts'$ , then  $\ell = \ell'$ .

**EC3: Eventual leadership:** There is a time after which every correct process has started some epoch and starts no further epoch, such that the last epoch started at every correct process is epoch  $(ts, \ell)$  and process  $\ell$  is correct.

# Leader-based Epoch-Change

- Every process  $p$  maintains two timestamps:
  - a timestamp  $lastts$  of the last epoch that it locally started;
  - a timestamp  $ts$  of the last epoch it attempted to start as a leader.
- When the leader detector makes  $p$  trust itself,  $p$  adds  $N$  to  $ts$  and broadcasts a NewEpoch message with  $ts$ .
- When  $p$  receives a NewEpoch message with parameter  $newts > lastts$  from  $l$  and  $p$  most recently trusted  $l$ , then  $p$  triggers a StartEpoch message.
- Otherwise,  $p$  informs the aspiring leader  $l$  with a NACK that a new epoch could not be started.
- When  $l$  receives a NACK and still trusts itself, it increments  $ts$  by  $N$  and tries again to start a new epoch.

**Implements:**

EpochChange, **instance**  $ec$ .

**Uses:**

PerfectPointToPointLinks, **instance**  $pl$ ;  
BestEffortBroadcast, **instance**  $beb$ ;  
EventualLeaderDetector, **instance**  $\Omega$ .

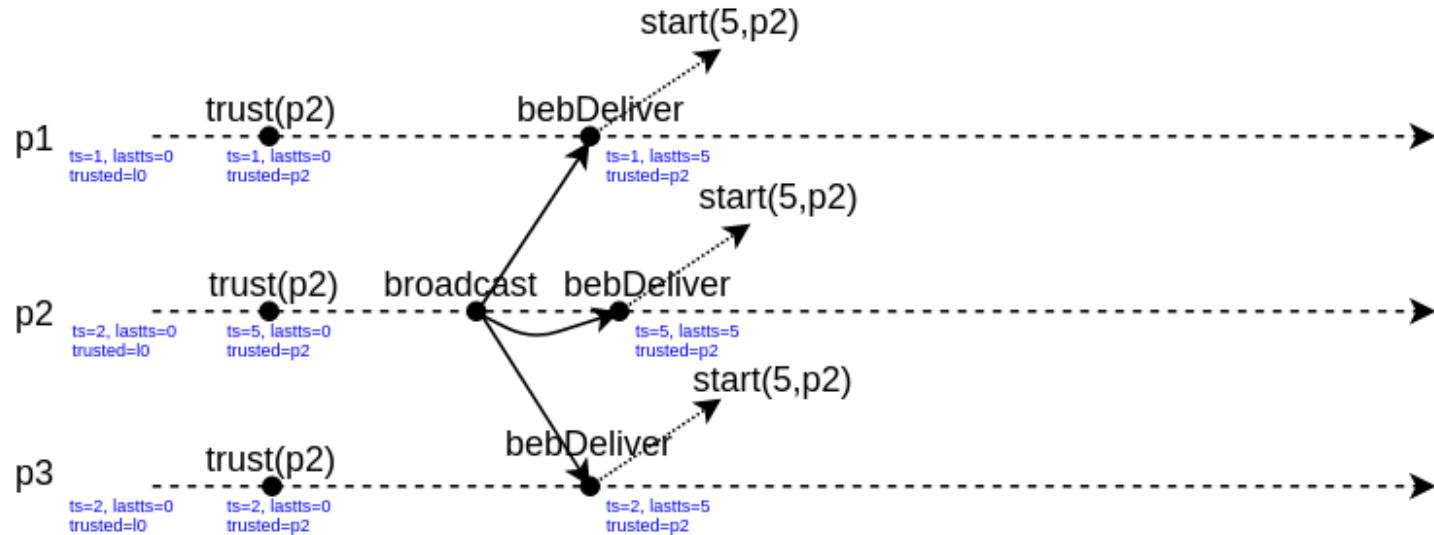
```
upon event <  $ec$ , Init > do
    trusted :=  $\ell_0$ ;
    lastts := 0;
    ts := rank(self);

upon event <  $\Omega$ , Trust |  $p$  > do
    trusted :=  $p$ ;
    if  $p = self$  then
        ts := ts + N;
        trigger <  $beb$ , Broadcast | [NEWEPOCH, ts] >;

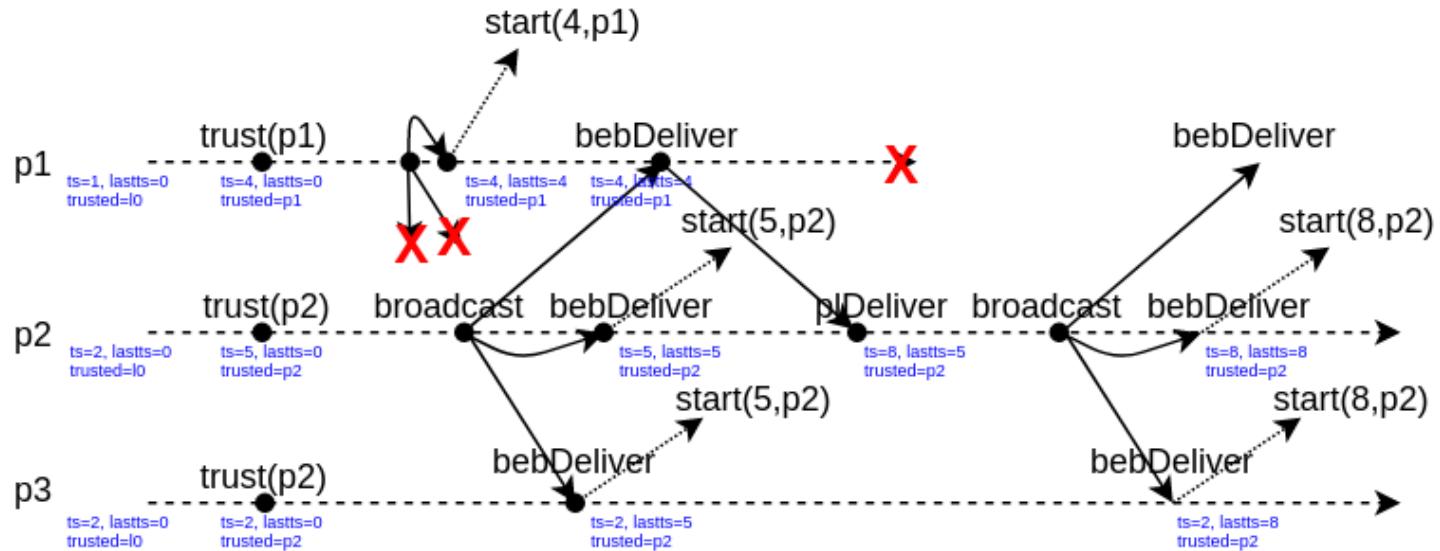
upon event <  $beb$ , Deliver |  $\ell$ , [NEWEPOCH, newts] > do
    if  $\ell = trusted \wedge newts > lastts$  then
        lastts := newts;
        trigger <  $ec$ , StartEpoch | newts,  $\ell$  >;
    else
        trigger <  $pl$ , Send |  $\ell$ , [NACK] >

upon event <  $pl$ , Deliver |  $p$ , [NACK] > do
    if trusted = self then
        ts := ts + N;
        trigger <  $beb$ , Broadcast | [NEWEPOCH, ts] >;
```

## Sample execution (1)



## Sample execution (2)



[Q] What if  $p_1$  fails only later, some time after the second bebDeliver event?

[Q] What if instead of crashing,  $p_1$  eventually trusts  $p_2$ ?

[Q] Could  $p_1$  and  $p_2$  keep bouncing NACKs to each other?

# Epoch consensus (*ep*)

- Let us define an **epoch consensus** abstraction, whose purpose is similar to **consensus**, but with the following simplifications:
  - Epoch consensus represents an **attempt** to reach consensus.
    - The procedure can be aborted when it does not decide or when the next epoch should already be started by the higher-level algorithm.
  - Every epoch consensus instance is identified by an **epoch timestamp *ts*** and a **designated leader *l***.
  - **Only the leader** proposes a value. Epoch consensus is required to decide **only when the leader is correct**.
- An instance **must terminate** when the application locally triggers an **Abort** event.
- The state of the component is initialized
  - with a higher timestamp than that of all instances it initialized previously;
  - with the state of the most recently locally aborted epoch consensus instance.

**Module:**

**Name:** EpochConsensus, **instance**  $ep$ , with timestamp  $ts$  and leader process  $\ell$ .

**Events:**

**Request:**  $\langle ep, \text{Propose} \mid v \rangle$ : Proposes value  $v$  for epoch consensus. Executed only by the leader  $\ell$ .

**Request:**  $\langle ep, \text{Abort} \rangle$ : Aborts epoch consensus.

**Indication:**  $\langle ep, \text{Decide} \mid v \rangle$ : Outputs a decided value  $v$  of epoch consensus.

**Indication:**  $\langle ep, \text{Aborted} \mid state \rangle$ : Signals that epoch consensus has completed the abort and outputs internal state  $state$ .

**Properties:**

**EP1: Validity:** If a correct process  $ep$ -decides  $v$ , then  $v$  was  $ep$ -proposed by the leader  $\ell'$  of some epoch consensus with timestamp  $ts' \leq ts$  and leader  $\ell'$ .

**EP2: Uniform agreement:** No two processes  $ep$ -decide differently.

**EP3: Integrity:** Every correct process  $ep$ -decides at most once.

**EP4: Lock-in:** If a correct process has  $ep$ -decided  $v$  in an epoch consensus with timestamp  $ts' < ts$ , then no correct process  $ep$ -decides a value different from  $v$ .

**EP5: Termination:** If the leader  $\ell$  is correct, has  $ep$ -proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually  $ep$ -decides some value.

**EP6: Abort behavior:** When a correct process aborts an epoch consensus, it eventually will have completed the abort; moreover, a correct process completes an abort only if the epoch consensus has been aborted by some correct process.

# Read/Write Epoch consensus

- Let us initialize the Read/Write Epoch consensus algorithm with the state of the most recently aborted epoch consensus instance.
  - The state contains a proposal *val* and its associated timestamp *valts*.
  - Passing the state to the next epoch consensus serves the **validity** and **lock-in** properties.
- The algorithm involves two rounds of messages from the leader to all processes.
  - The leader writes its proposal value to all processes, who store the epoch timestamp and the value in their state, and acknowledge this to the leader.
  - When the leader receives enough acknowledgements, it decides this value.
  - However, if the leader of some previous epoch already decided some value *val*, then no other value should be decided (to not violate lock-in).
  - To prevent this, the leader first reads the state of the processes, which return State messages.
  - The leader receives a quorum of State messages and chooses the value that comes with the highest timestamp, if one exists.
  - The leader decides and broadcasts its decision to all processes, which then decide too.

**Implements:**

EpochConsensus, **instance**  $ep$ , with timestamp  $ets$  and leader  $\ell$ .

**Uses:**

PerfectPointToPointLinks, **instance**  $pl$ ;  
BestEffortBroadcast, **instance**  $beb$ .

```
upon event < ep, Init | state > do
    (valts, val) := state;
    tmpval := ⊥;
    states := [⊥]N;
    accepted := 0;

upon event < ep, Propose | v > do                                // only leader ℓ
    tmpval := v;
    trigger < beb, Broadcast | [READ] >;

upon event < beb, Deliver | ℓ, [READ] > do
    trigger < pl, Send | ℓ, [STATE, valts, val] >;

upon event < pl, Deliver | q, [STATE, ts, v] > do                // only leader ℓ
    states[q] := (ts, v);
```

```

upon #(states) > N/2 do // only leader ℓ
  (ts, v) := highest(states);
  if v ≠ ⊥ then
    tmpval := v;
  states := [⊥]N;
  trigger ⟨ beb, Broadcast | [WRITE, tmpval] ⟩;

upon event ⟨ beb, Deliver | ℓ, [WRITE, v] ⟩ do
  (valts, val) := (ets, v);
  trigger ⟨ pl, Send | ℓ, [ACCEPT] ⟩;

upon event ⟨ pl, Deliver | q, [ACCEPT] ⟩ do // only leader ℓ
  accepted := accepted + 1;

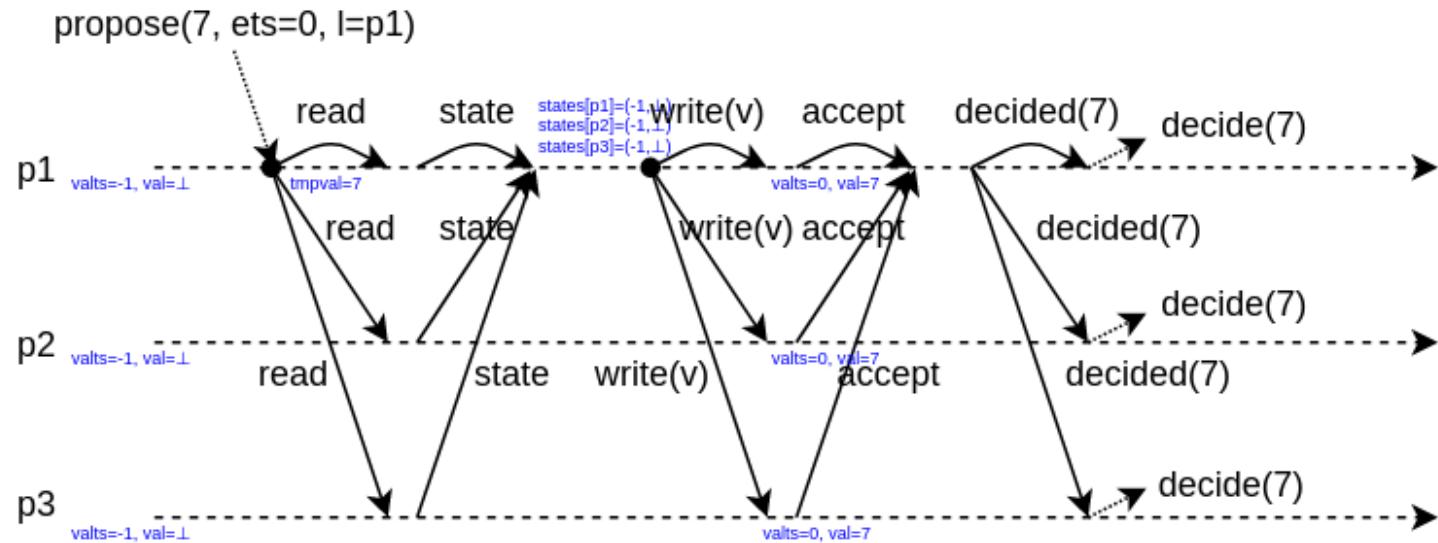
upon accepted > N/2 do // only leader ℓ
  accepted := 0;
  trigger ⟨ beb, Broadcast | [DECIDED, tmpval] ⟩;

upon event ⟨ beb, Deliver | ℓ, [DECIDED, v] ⟩ do
  trigger ⟨ ep, Decide | v ⟩;

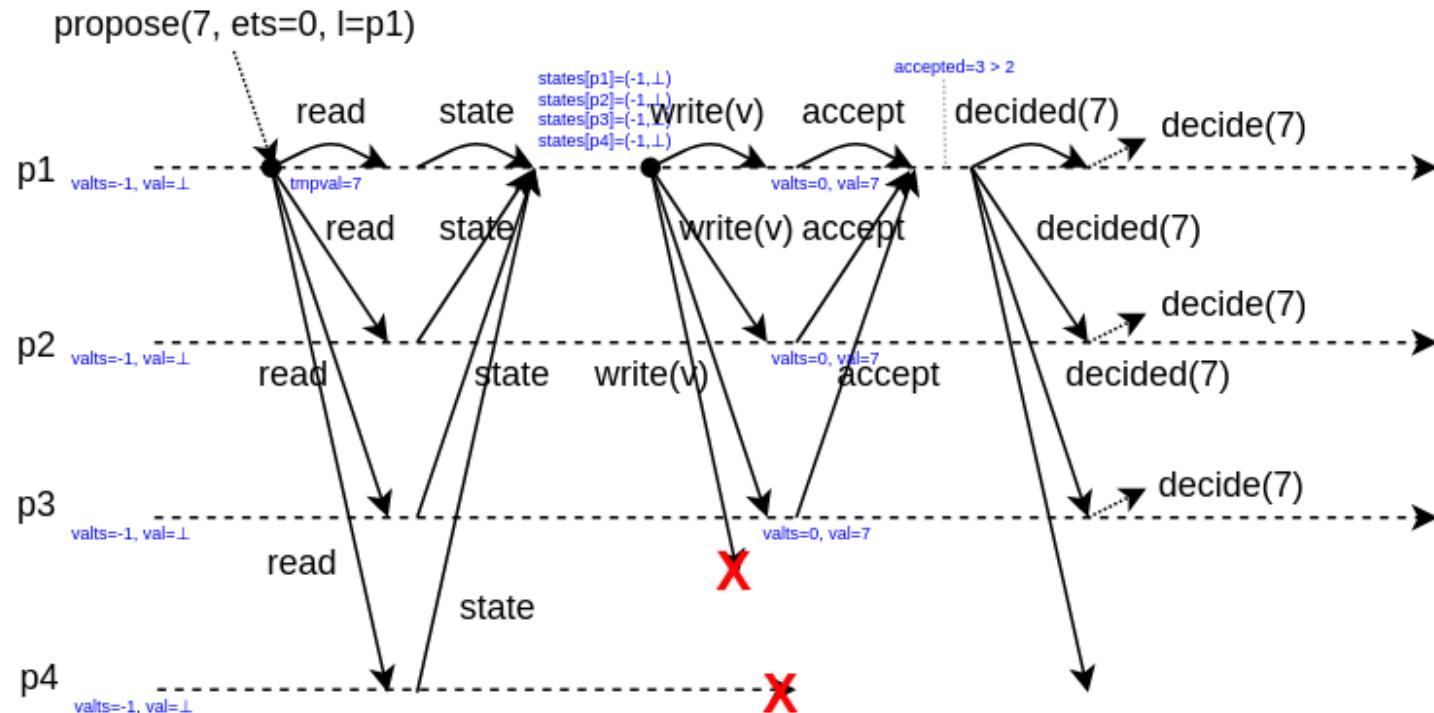
upon event ⟨ ep, Abort ⟩ do
  trigger ⟨ ep, Aborted | (valts, val) ⟩;
  halt; // stop operating when aborted

```

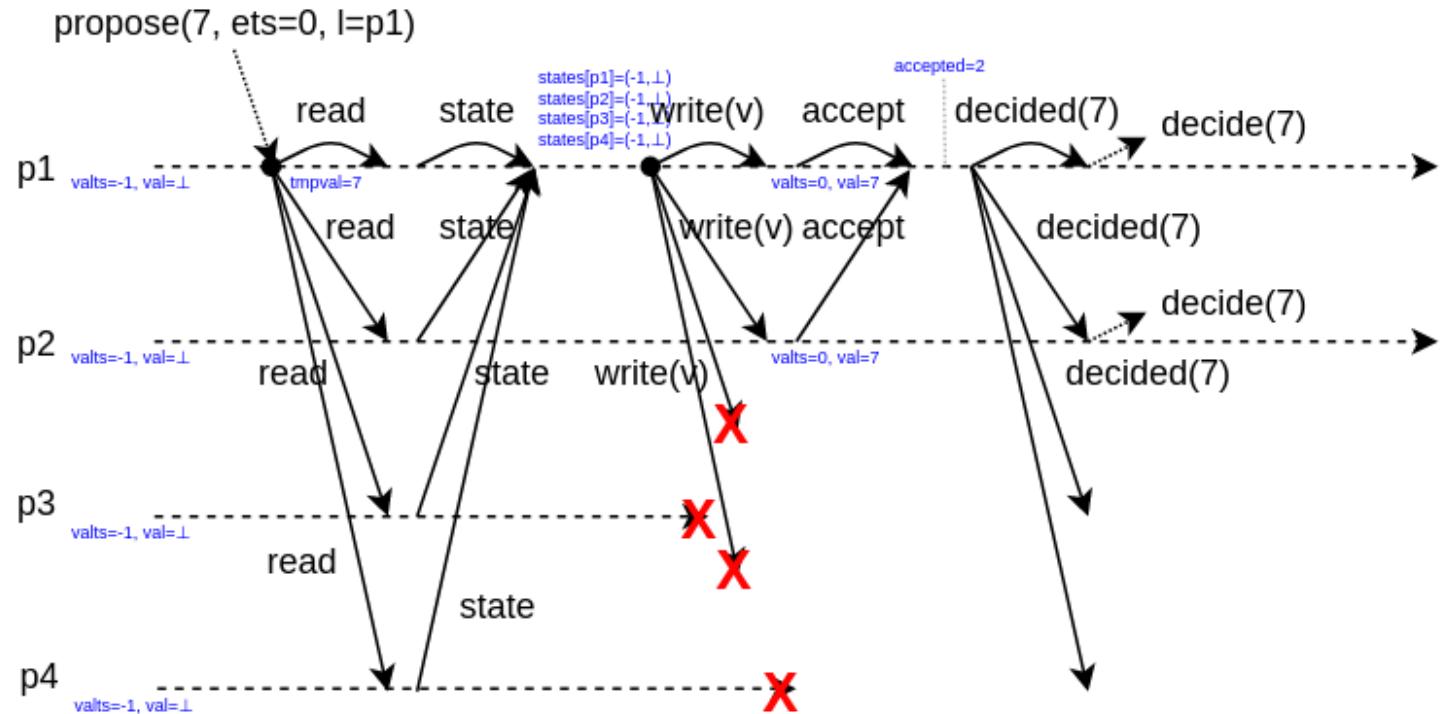
## Sample execution (1)



## Sample execution (2)

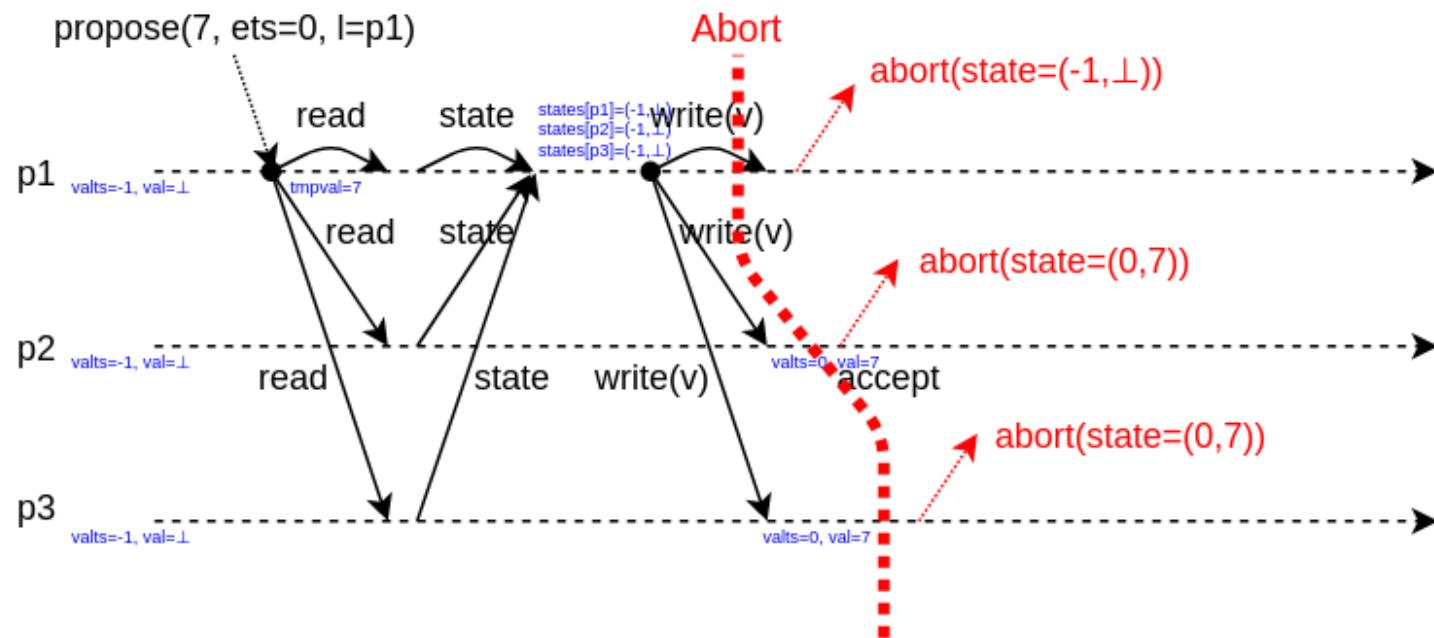


## Sample execution (3)



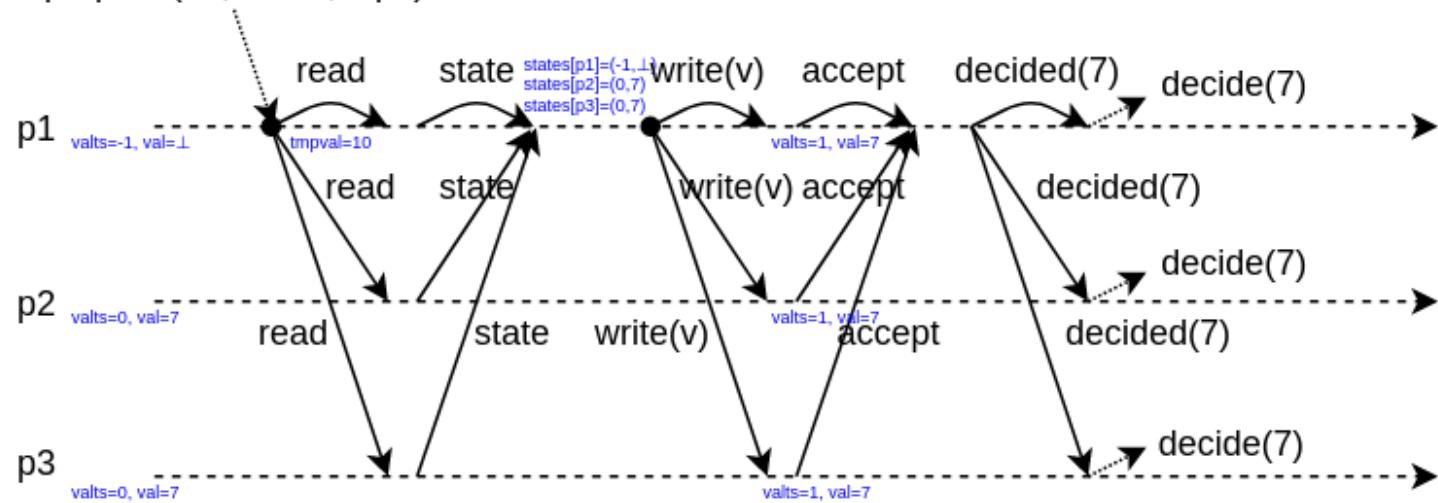
[Q] What is wrong in this execution?

## Sample execution (4a)



## Sample execution (4b)

propose(10, ets=1, l=p1)



## Correctness

Assume a **majority of correct processes**, i.e.  $N > 2f$ , where  $f$  is the number of crash faults.

- **Lock-in:** If a correct process has ep-decided  $v$  in an epoch consensus with timestamp  $ts' \leq ts$ , then no correct process ep-decides a value different from  $v$ .
  - If some process ep-decided  $v$  at  $ts' < ts$ , then it decided after receiving a Decided message with  $v$  from leader  $l'$  of epoch  $ts'$ .
  - Before sending this message,  $l'$  had broadcast a Write containing  $v$  and collected Accept messages.
  - These responding processes set their variables  $val$  to  $v$  and  $valts$  to  $ts'$ .
  - At the next epoch, the leader sent a Write message and collected Accept messages with the previous  $(ts', v)$  pair.
  - This pair has the highest timestamp with a non-null value.
  - This implies that the leader of this epoch can only ep-decides  $v$ .
  - This argument can be continued until  $ts$ , establishing lock-in.

- **Validity:** If a correct process ep-decides  $v$ , then  $v$  was ep-proposed by the leader  $l'$  of some epoch consensus with timestamp  $ts' \leq ts$  and leader  $l'$ .
  - If some process ep-decides  $v$ , it is because this value was delivered from a Decided message.
  - Furthermore, every process stores in  $val$  only the value received in a Write message from the leader.
  - In both cases, this value comes from  $tmpval$  of the leader.
  - In any epoch, the leader sets  $tmpval$  only to the value it ep-proposed or to some value it received in a State message from another process.
  - By backward induction,  $v$  was ep-proposed by the leader in some epoch  $ts' \leq ts$ .
- **Uniform agreement + integrity:** No two processes ep-decide differently + Every correct process ep-decides at most once.
  - $l$  sends the same value to all processes in the Decided message.
- **Termination:** If the leader  $l$  is correct, has ep-proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually ep-decides some value.
  - When  $l$  is correct and no process aborts the epoch, then every process eventually receives a Decide message and ep-decides.

[Q] What may go wrong if we do not assume a majority of correct processes?

# Leader-Driven consensus

- Let us now combine the epoch-change and the epoch consensus abstractions to form the **leader-driven consensus** algorithm.
- We will write the **glue** to repeatedly run epoch consensus until epoch changes stabilize and all decisions are taken.
- The algorithm provides **uniform consensus** in **fail-noisy**.

Leader-driven consensus runs through a **sequence of epochs**, triggered by StartEpoch events from the epoch-change primitive:

- The current epoch timestamp is  $\textit{ets}$  and the associated leader is  $\textit{l}$ .
- The StartEpoch events determine the timestamp  $\textit{newts}$  and the leader  $\textit{newl}$  of the next epoch consensus instance to start.
- To switch from one epoch consensus to the next, the algorithm aborts the running epoch consensus instance, obtains its state and initializes the next epoch consensus instance with it.
- As soon as a process has obtained a proposal value  $v$  for consensus and is the leader of the current epoch, it ep-proposes this value for epoch consensus.
- When the current epoch ep-decides a value, the process also decides this value for consensus.
- The process continues to participate in the consensus to help other processes decide.

**Implements:**

UniformConsensus, **instance**  $uc$ .

**Uses:**

EpochChange, **instance**  $ec$ ;

EpochConsensus (multiple instances).

**upon event**  $\langle uc, Init \rangle$  **do**

$val := \perp$ ;

$proposed := \text{FALSE}$ ;  $decided := \text{FALSE}$ ;

Obtain the leader  $\ell_0$  of the initial epoch with timestamp 0 from epoch-change inst.  $ec$ ;

Initialize a new instance  $ep.0$  of epoch consensus with timestamp 0,

leader  $\ell_0$ , and state  $(0, \perp)$ ;

$(ets, \ell) := (0, \ell_0)$ ;

$(newts, new\ell) := (0, \perp)$ ;

**upon event**  $\langle uc, Propose \mid v \rangle$  **do**

$val := v$ ;

```

upon event ⟨ ec, StartEpoch | newts', newℓ' ⟩ do
  (newts, newℓ) := (newts', newℓ');
  trigger ⟨ ep.ets, Abort ⟩;

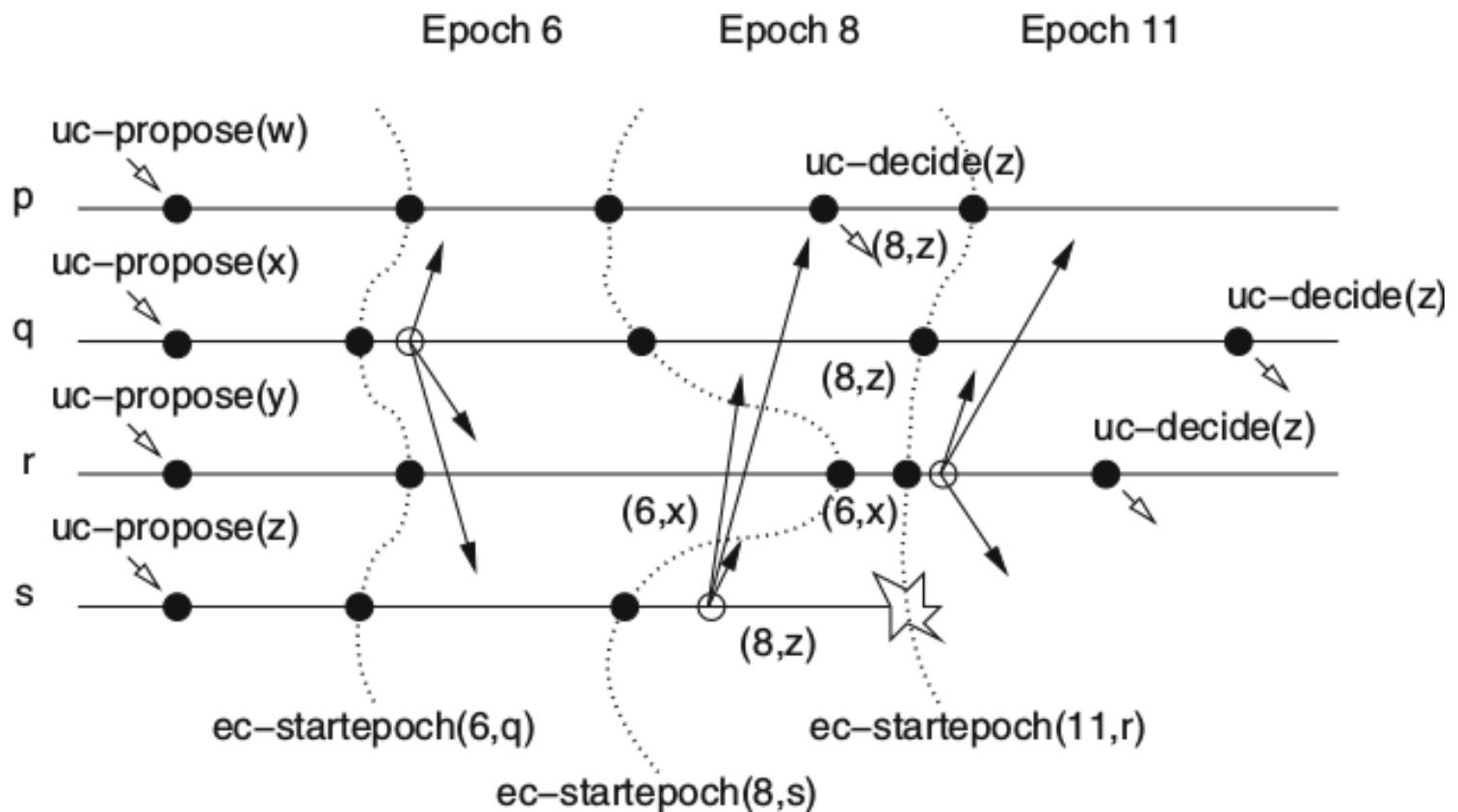
upon event ⟨ ep.ts, Aborted | state ⟩ such that ts = ets do
  (ets, ℓ) := (newts, newℓ);
  proposed := FALSE;
  Initialize a new instance ep.ets of epoch consensus with timestamp ets,
  leader ℓ, and state state;

upon ℓ = self  $\wedge$  val  $\neq \perp$   $\wedge$  proposed = FALSE do
  proposed := TRUE;
  trigger ⟨ ep.ets, Propose | val ⟩;

upon event ⟨ ep.ts, Decide | v ⟩ such that ts = ets do
  if decided = FALSE then
    decided := TRUE;
  trigger ⟨ uc, Decide | v ⟩;

```

## Sample execution



## Correctness

- **Validity:** If a process decides  $v$ , then  $v$  was proposed by some process.
  - A process uc-decides  $v$  only when it has ep-decided  $v$  in the current epoch consensus.
  - Every decision can be attributed to a unique epoch and to a unique instance of epoch consensus.
  - Let  $ts^*$  be the smallest timestamp of an epoch consensus in which some process decides  $v$ .
  - According to the validity property of epoch consensus, this means  $v$  was ep-proposed by the leader of some epoch whose timestamp is at most  $ts^*$ .
  - Since a process only ep-proposes  $val$  when  $val$  has been uc-proposed for consensus, the **validity** property follows for processes that uc-decide in epoch  $ts^*$ .
  - The argument extends to  $ts > ts^*$  because the lock-in property of epoch consensus forces processes to ep-decide  $v$  only, which in turn make them uc-decide.

- **Uniform agreement:** No two processes decide differently.
  - Every decision attributed to an ep-decision of some epoch consensus instance.
  - If two correct processes decide when they are in the same epoch, then the uniform agreement of epoch consensus ensures the decisions are the same.
  - If they decide in different epoch, the lock-in property establishes uniform agreement.

- **Integrity:** No process decides twice.
  - The decided flag in the algorithm prevents multiple decisions.
- **Termination:** Every correct process eventually decides some value.
  - Because of **eventual leadership** of the epoch-change primitive, there is some epoch with timestamp  $ts$  and leader  $l$  such that no further epoch starts and  $l$  is correct.
  - From that instant, no further abortions are triggered.
  - The **termination** property of epoch consensus ensures that every correct process eventually ep-decides, and therefore uc-decides.

# Paxos

Leader-driven consensus is a modular formulation of the **Paxos** consensus algorithm by Leslie Lamport.



## The Part-Time Parliament

LESLIE LAMPORT

Digital Equipment Corporation

---

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

Categories and Subject Descriptors: C2.4 [Computer-Communications Networks]: Distributed Systems—*Network operating systems*; D4.5 [Operating Systems]: Reliability—*Fault-tolerance*; J.1 [Administrative Data Processing]: Government

General Terms: Design, Reliability

Additional Key Words and Phrases: State machines, three-phase commit, voting

---

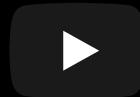
This submission was recently discovered behind a filing cabinet in the *TOCS* editorial office. Despite its age, the editor-in-chief felt that it was worth publishing. Because the author is currently doing field work in the Greek isles and cannot be reached, I was asked to prepare it for publication.

The author appears to be an archeologist with only a passing interest in computer science. This is unfortunate; even though the obscure ancient Paxon civilization he describes is of little interest to most computer scientists, its legislative system is an excellent model for how to implement a distributed computer system in an asynchronous environment. Indeed, some of the refinements the Paxons made to their protocol appear to be unknown in the systems literature.

The author does give a brief discussion of the Paxon Parliament's relevance to distributed computing in Section 4. Computer scientists will probably want to read that section first. Even before that, they might want to read the explanation of the algorithm for computer scientists by Lampson [1996]. The algorithm is also described more formally by De Prisco et al. [1997]. I have added further comments on the relation between the ancient protocols and more recent work at the end of Section 4.

Keith Marzullo  
University of California, San Diego

...  
...  
**<<C>**  
...  
...



# Total order broadcast

# Total order broadcast (*tob*)

- The **total-order (reliable) broadcast** (also known as **atomic broadcast**) abstraction ensures that all processes deliver the same messages in a **common global order**.
- Total-order broadcast is the key abstraction for maintaining consistency among multiple replicas that implement one logical service.

**Module:**

**Name:** TotalOrderBroadcast, **instance**  $tob$ .

**Events:**

**Request:**  $\langle tob, Broadcast \mid m \rangle$ : Broadcasts a message  $m$  to all processes.

**Indication:**  $\langle tob, Deliver \mid p, m \rangle$ : Delivers a message  $m$  broadcast by process  $p$ .

**Properties:**

**TOB1: Validity:** If a correct process  $p$  broadcasts a message  $m$ , then  $p$  eventually delivers  $m$ .

**TOB2: No duplication:** No message is delivered more than once.

**TOB3: No creation:** If a process delivers a message  $m$  with sender  $s$ , then  $m$  was previously broadcast by process  $s$ .

**TOB4: Agreement:** If a message  $m$  is delivered by some correct process, then  $m$  is eventually delivered by every correct process.

**TOB5: Total order:** Let  $m_1$  and  $m_2$  be any two messages and suppose  $p$  and  $q$  are any two correct processes that deliver  $m_1$  and  $m_2$ . If  $p$  delivers  $m_1$  before  $m_2$ , then  $q$  delivers  $m_1$  before  $m_2$ .

# Consensus-based TOB

- Messages are first disseminated using a **reliable broadcast instance**.
  - No particular order is imposed on the messages.
  - At any point in time, it may be that no two processes have the same sets of unordered messages.
- The processes use **consensus** to decide on one set of messages to be delivered, order the messages in this set, and finally deliver them.

**Implements:**

TotalOrderBroadcast, instance *tob*.

**Uses:**

ReliableBroadcast, instance *rb*;  
Consensus (multiple instances).

**upon event**  $\langle \text{tob}, \text{Init} \rangle$  **do**

*unordered* :=  $\emptyset$ ;  
    *delivered* :=  $\emptyset$ ;  
    *round* := 1;  
    *wait* := FALSE;

**upon event**  $\langle \text{tob}, \text{Broadcast} \mid m \rangle$  **do**

**trigger**  $\langle \text{rb}, \text{Broadcast} \mid m \rangle$ ;

**upon event**  $\langle \text{rb}, \text{Deliver} \mid p, m \rangle$  **do**

**if**  $m \notin \text{delivered}$  **then**  
        *unordered* := *unordered*  $\cup \{(p, m)\}$ ;

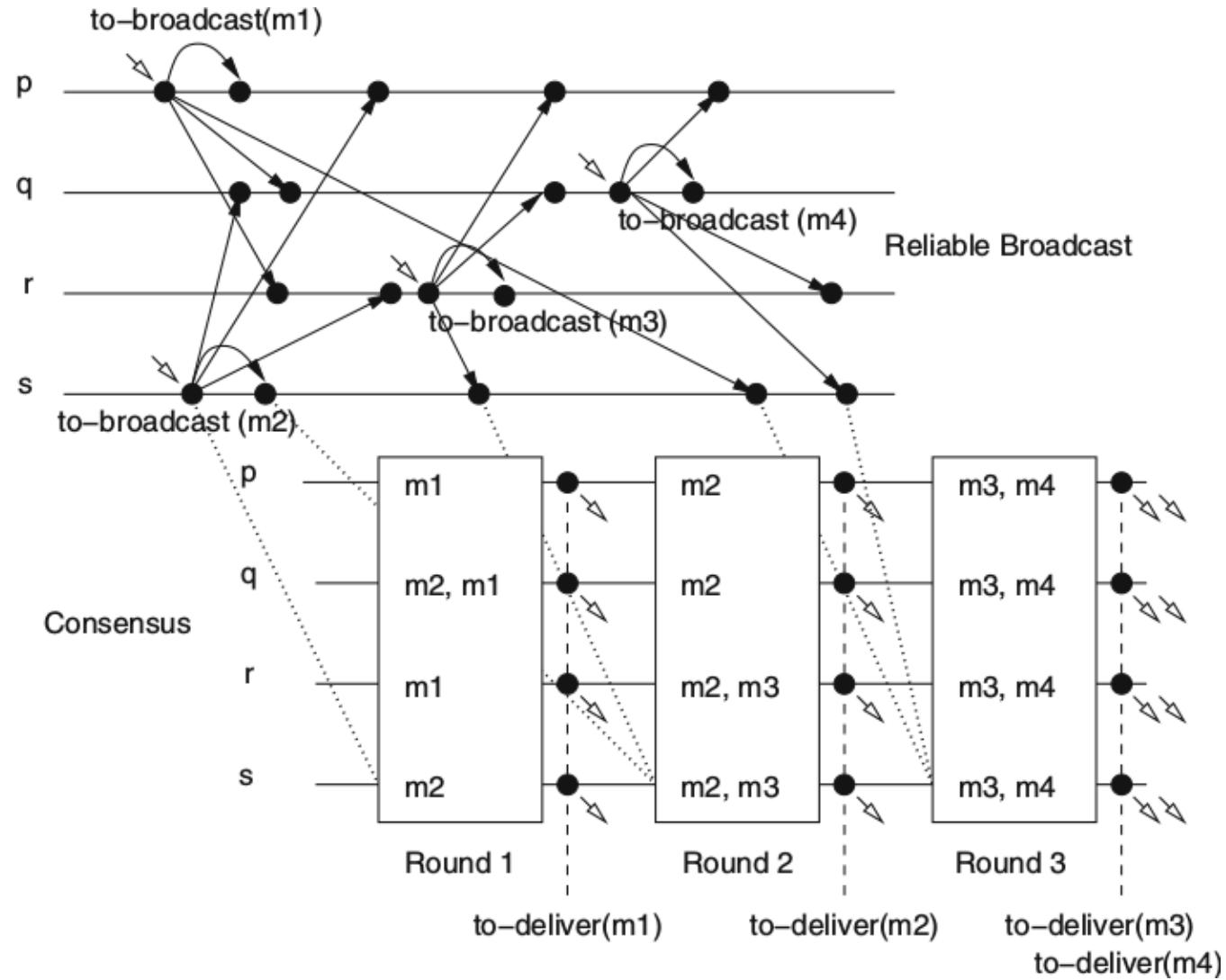
**upon**  $\text{unordered} \neq \emptyset \wedge \text{wait} = \text{FALSE}$  **do**

*wait* := TRUE;  
    Initialize a new instance *c*.*round* of consensus;  
    **trigger**  $\langle c.\text{round}, \text{Propose} \mid \text{unordered} \rangle$ ;

**upon event**  $\langle c.r, \text{Decide} \mid \text{decided} \rangle$  **such that**  $r = \text{round}$  **do**

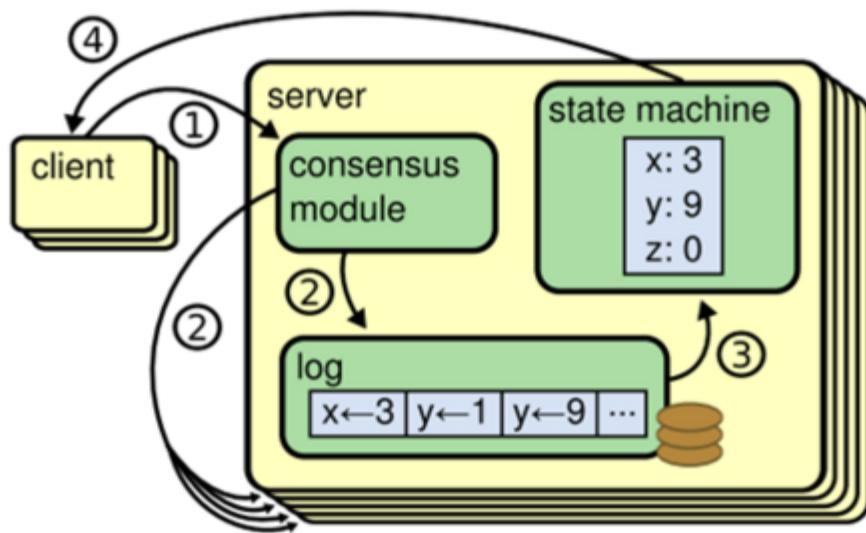
**forall**  $(s, m) \in \text{sort}(\text{decided})$  **do** // by the order in the resulting sorted list  
        **trigger**  $\langle \text{tob}, \text{Deliver} \mid s, m \rangle$ ;  
        *delivered* := *delivered*  $\cup \text{decided}$ ;  
        *unordered* := *unordered*  $\setminus \text{decided}$ ;  
        *round* := *round* + 1;  
        *wait* := FALSE;

## Sample execution



# Replicated state machines (*rsm*)

- A **state machine** consists of variables and commands that transform its state and produce some output.
- Commands are **deterministic** programs, such that the outputs are solely determined by the initial state and the sequence of commands.
- A state machine can be made **fault-tolerant** by replicating it on different processes.
- This can now be easily implemented simply by disseminating all commands to execute using a uniform total-order broadcast primitive.
- This gives a **generic recipe** to make any deterministic program distributed, consistent and fault-tolerant!



**Module:**

**Name:** ReplicatedStateMachine, **instance** *rsm*.

**Events:**

**Request:**  $\langle rsm, Execute \mid command \rangle$ : Requests that the state machine executes the command given in *command*.

**Indication:**  $\langle rsm, Output \mid response \rangle$ : Indicates that the state machine has executed a command with output *response*.

**Properties:**

**RSM1:** *Agreement*: All correct processes obtain the same sequence of outputs.

**RSM2:** *Termination*: If a correct process executes a command, then the command eventually produces an output.

# TOB-based Replicated state machines

**Implements:**

ReplicatedStateMachine, **instance** *rsm*.

**Uses:**

UniformTotalOrderBroadcast, **instance** *utob*;

**upon event** *< rsm, Init >* **do**

*state* := initial state;

**upon event** *< rsm, Execute | command >* **do**

**trigger** *< utob, Broadcast | command >*;

**upon event** *< utob, Deliver | p, command >* **do**

*(response, newstate)* := execute(*command, state*);

*state* := *newstate*;

**trigger** *< rsm, Output | response >*;

# Summary

- Consensus is the problem of making processes all agree on one of the values they propose.
- The FLP impossibility result states that no consensus protocol can be proven to always terminate in an asynchronous system.
- In fail-silent, Hierarchical Consensus provides an implementation based on broadcast and failure detection.
- In fail-noisy, Leader-Driven Consensus achieves consensus by repeatedly running epoch consensus until all decisions are taken.
- The consensus primitive greatly simplifies the implementation of any fault-tolerant consistent distributed system.
  - Total-order broadcast
  - Replicated state machines



# References

- Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." *Journal of the ACM (JACM)* 32.2 (1985): 374-382.
- Lamport, Leslie. "The part-time parliament." *ACM Transactions on Computer Systems (TOCS)* 16.2 (1998): 133-169.
- Lamport, Leslie. "Paxos made simple." *ACM Sigact News* 32.4 (2001): 18-25.

# Large-scale Data Systems

Lecture 6: Blockchain

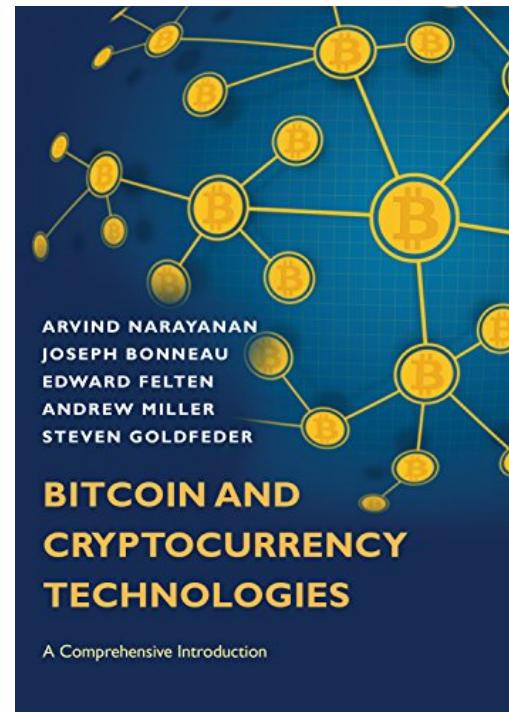
Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)



# Today

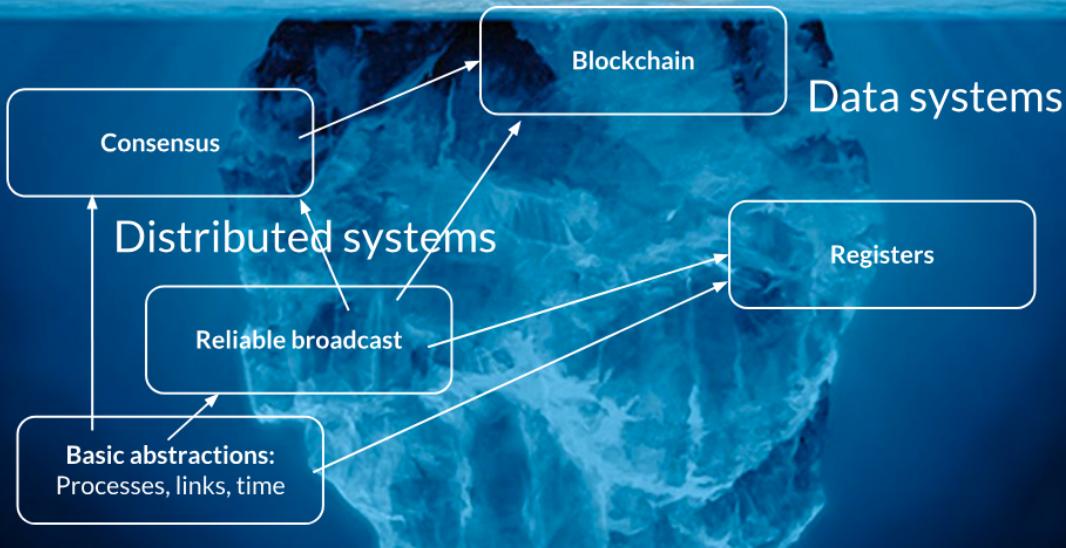
Blockchain:

- Hash functions and data structures
- Digital signatures
- Simple (fictitious) cryptocurrencies
- Consensus in the blockchain
- Bitcoin and friends



Most of today's lecture is based on "Bitcoin and cryptocurrency technologies: A comprehensive introduction" by Narayanan et al.

# Data science Machine Learning Visualization



# **Hash functions and data structures**

# Hash functions

A **hash function** is a mathematical function  $H$  with the following properties:

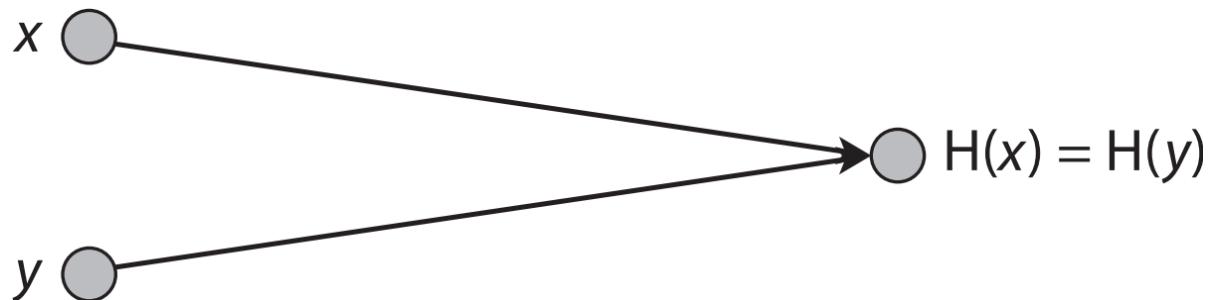
- Its input can be any string of any size.
- It produces a fixed-size output (e.g. 256 bits).
- It is efficiently computable.
  - E.g., computing the hash of an  $n$ -bit string should be  $O(n)$ .

## Security properties

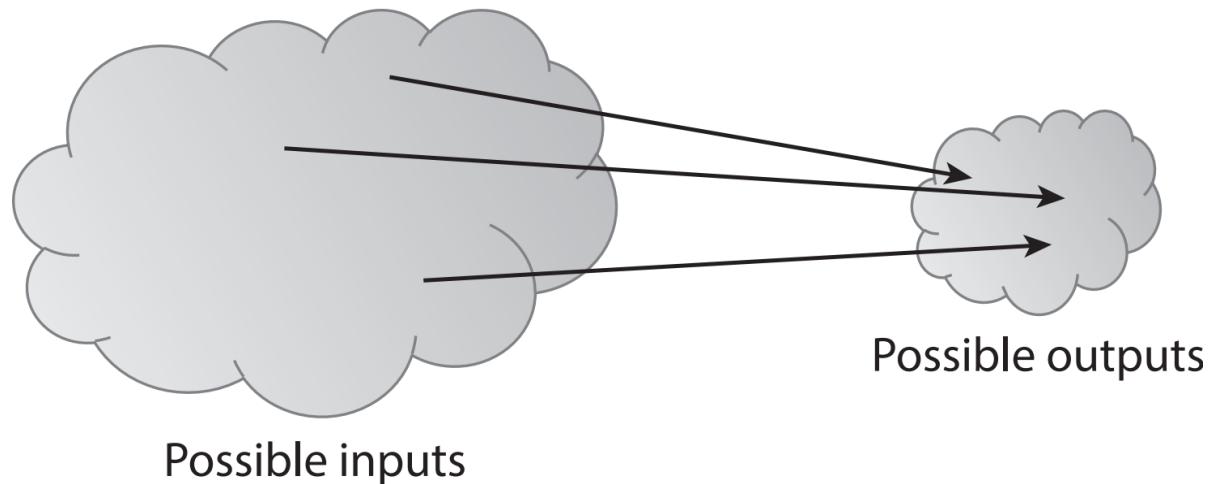
- Collision resistance
- Hiding
- Puzzle-friendliness

# Collision resistance

A hash function  $H$  is said to be **collision resistant** if it is infeasible/difficult to find two values  $x$  and  $y$  such that  $x \neq y$  yet  $H(x) = H(y)$ .



Collisions do exist. But can anyone find them?



How to find a collision:

- Pick  $2^{256} + 1$  distinct values and compute the hashes of each of them.
- At least one pair of inputs must collide!

This works no matter the hash function  $H$ . But it takes too long to matter!

# Hiding

A hash function  $H$  is said to be **hiding** if when a secret value  $r$  is chosen from a probability distribution that has high entropy, then given  $H(r||x)$ , it is infeasible to find  $x$ .

## **Application: Commitments**

A commitment is the digital analog of taking a value, sealing it in an envelope, and putting that envelope out on the table where everyone can see it.

- Commit to value (the content of the envelope).
- Reveal it later (open the envelope).

## Commitment scheme

- $\text{commit}(\text{msg}, \text{key}) := (H(\text{msg}||\text{key}), H(\text{key}))$ :  
The commit function takes a message and a (secret) key as input and returns a commitment pair  $\text{com}$ .
- $\text{verify}(\text{com}, \text{key}, \text{msg})$ : The verify function takes a commitment, a key and a message as inputs. It returns true iff  $\text{com} = \text{commit}(\text{msg}, \text{key})$ .

## Security properties

- Hiding: given  $\text{com} := (H(\text{msg}||\text{key}), H(\text{key}))$ , it is infeasible to find  $\text{msg}$ .
- Binding: Infeasible to find  $\text{msg} \neq \text{msg}'$  such that  $H(\text{msg}||\text{key}) = H(\text{msg}'||\text{key})$  is true, nor to change the key.

# Puzzle-friendliness

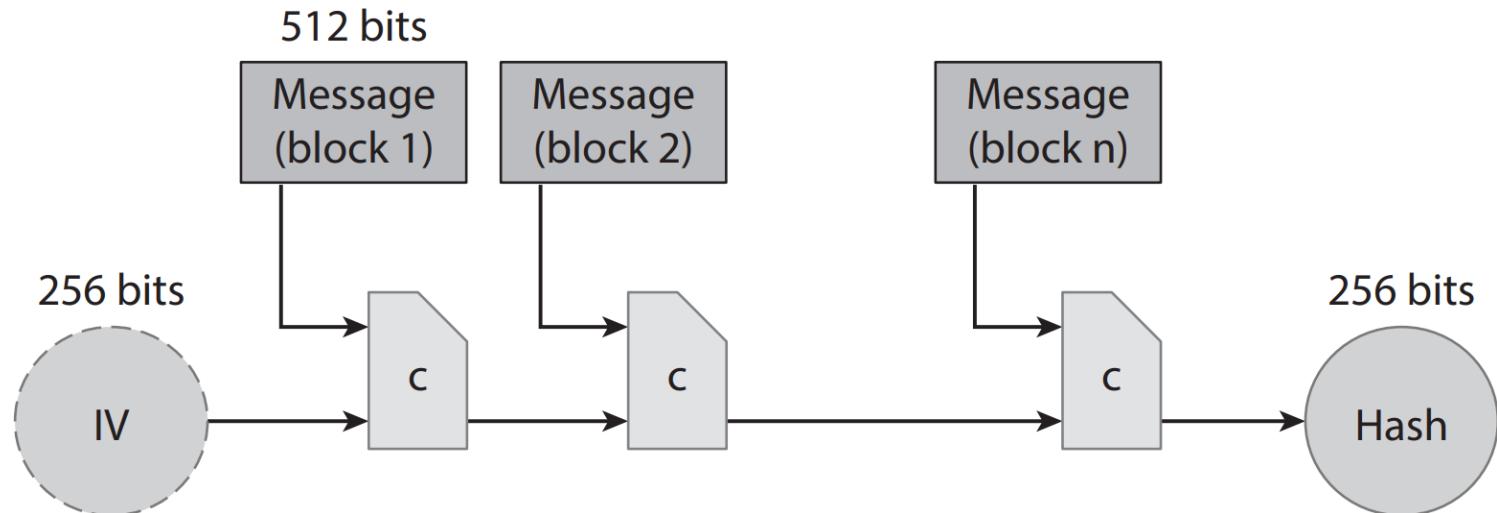
A hash function  $H$  is said to be **puzzle friendly** if for every possible  $n$ -bit output value  $y$ , if  $k$  is chosen from a distribution with high entropy and made public, then it remains infeasible to find  $x$  such that  $y = H(k||x)$  in time significantly less than  $2^n$ .

## Application: Search puzzle

Given a puzzle ID  $i$  and a target set  $Y$ , try to find a solution  $x$  such that  $H(i||x) \in Y$ .

- Puzzle-friendliness implies that no solving strategy is much better than trying random values of  $x$ .
- Later, we will see that mining is a computational puzzle.

# SHA-256 hash function

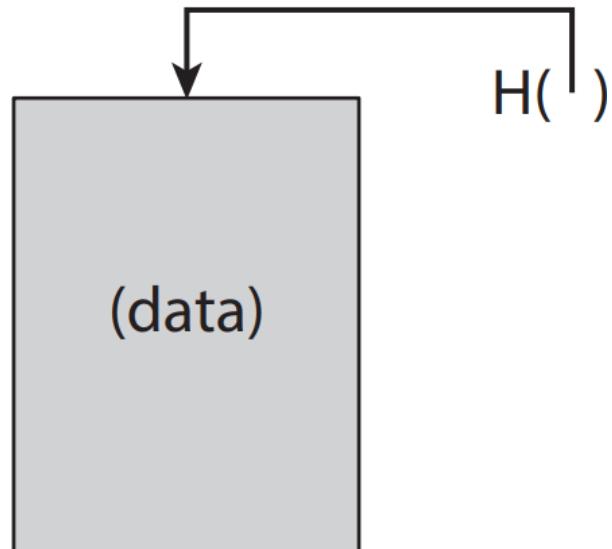


SHA-256 uses the Merkle–Damgård construction to turn a fixed-length collision resistant compression function  $c$  into a hash function that accepts arbitrary-length inputs.

# Hash pointers

A **hash pointer** is a pointer to where some information is stored, together with a cryptographic hash of this information. A hash pointer can be used for:

- retrieving the information associated to the pointer,
- verifying that the information has not changed.

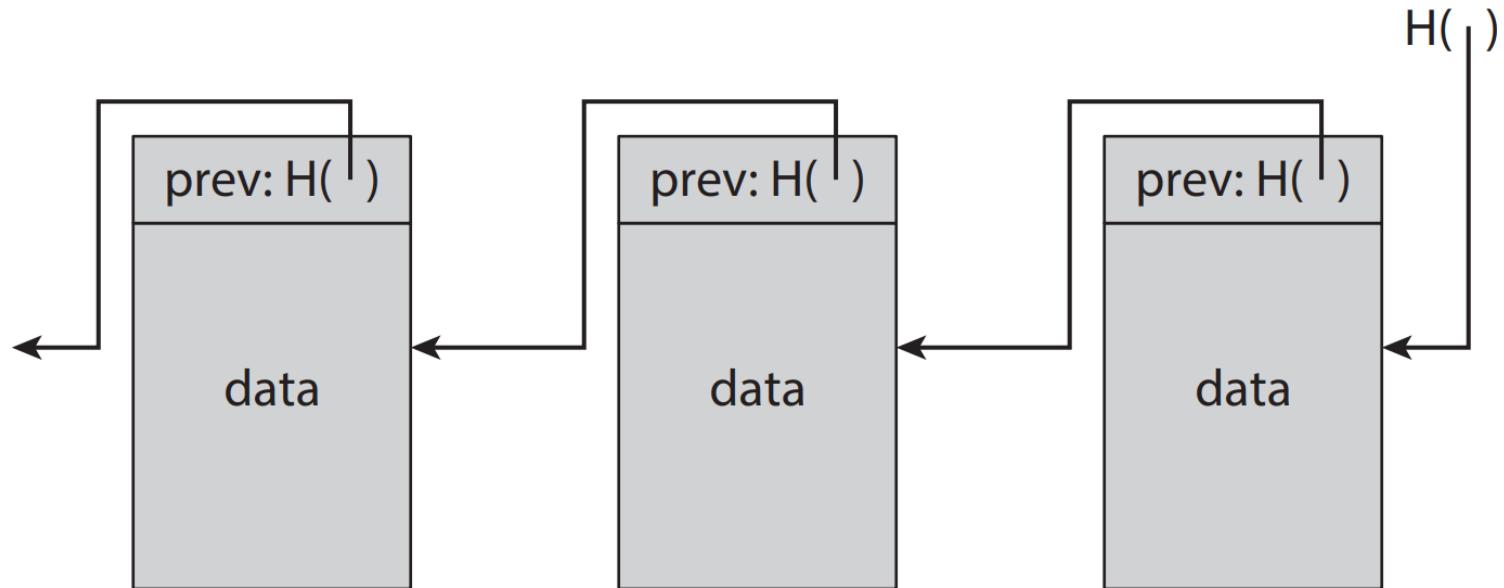


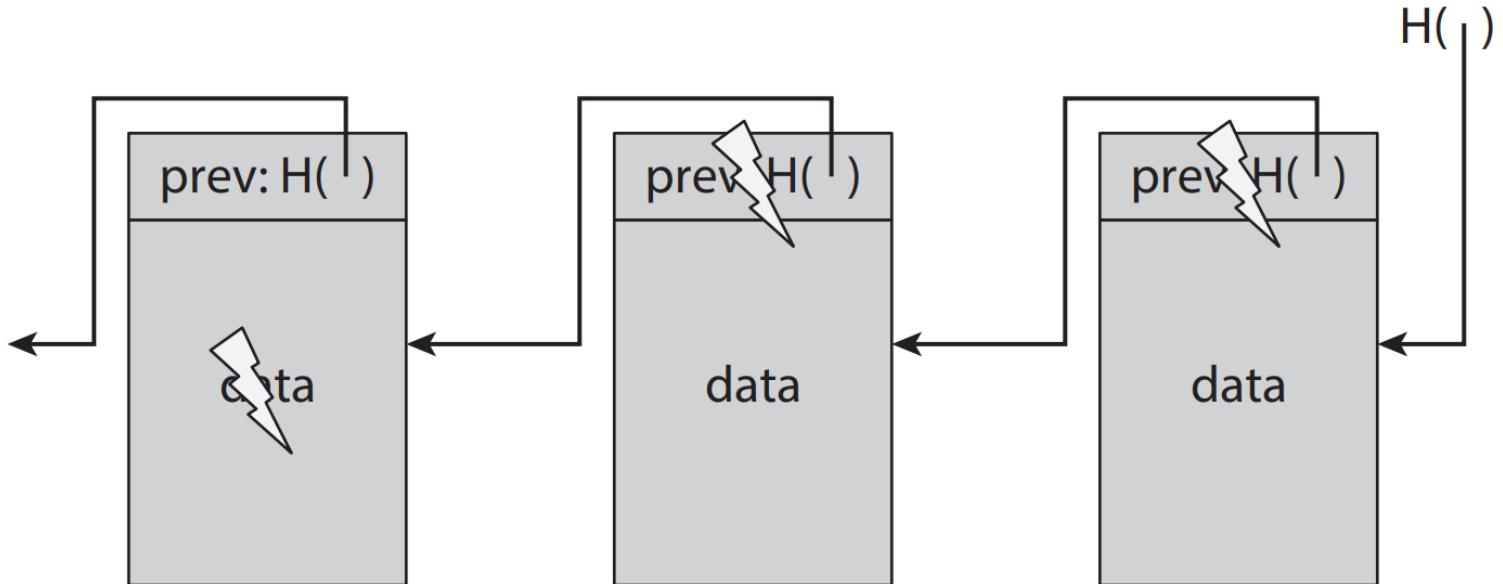
**Key idea:** build data structures with hash pointers

# Blockchain

A **blockchain** is a linked list that makes use of hash pointers.

- In a regular linked list, each block has data as well as a pointer to the previous block in the list.
- In a blockchain, the previous-block pointers are replaced by hash pointers.



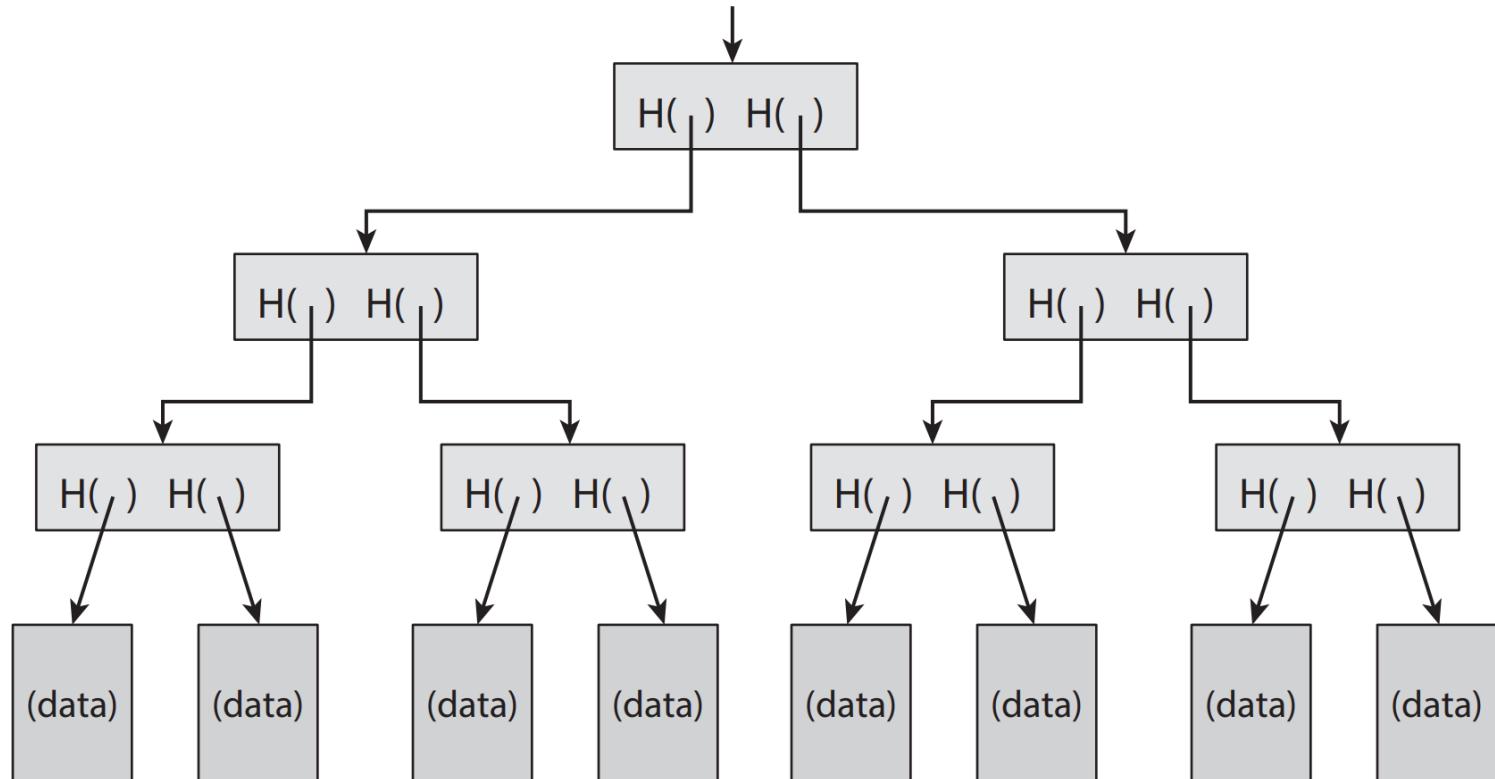


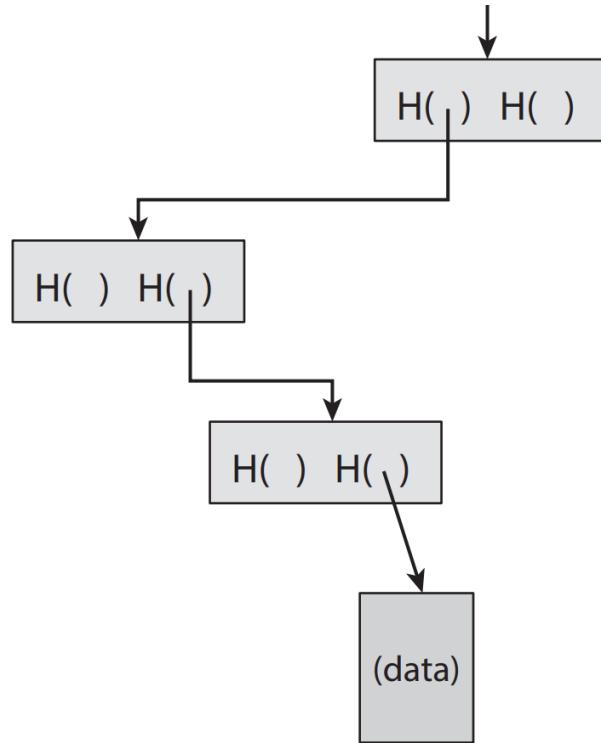
## Tamper-evident log

- If an adversary modifies data anywhere in the blockchain, it will result in the hash pointer in the following block being incorrect.
- If we stored the head of the list, then even if an adversary modifies all pointers to be consistent with modified data, the head pointer will be incorrect and the change would be detected.

# Merkle tree

A **merkle tree** is a binary tree that makes use of hash pointers.





## Proof of membership

Proving that a data block is included in the tree only requires showing the blocks in the path from that data block to the root. Hence  $O(\log N)$ .

# Digital signatures

# Digital signatures

Digital signatures is the second cryptographic primitive we will need for implementing cryptocurrencies.

A **digital signature** is the digital analog to a handwritten signature on paper:

- Only you can make your signature, but anyone can verify that it is valid.
- Signatures are tied to a particular document. They cannot be cut-and-pasted to another document to indicate your agreement or endorsement.

Bitcoin makes use of the [Elliptic Curve Digital Signature Algorithm \(ECDSA\)](#) for digital signatures.

## API for digital signatures

- $(\text{sk}, \text{pk}) := \text{generateKeys}(\text{keysize})$ 
  - $\text{sk}$ : secret signing key
  - $\text{pk}$ : public verification key
- $\text{sig} := \text{sign}(\text{sk}, \text{msg})$
- $\text{isValid} := \text{verify}(\text{pk}, \text{msg}, \text{sig})$

# Requirements

- Valid signatures must verify.
- Signatures are existentially unforgeable.

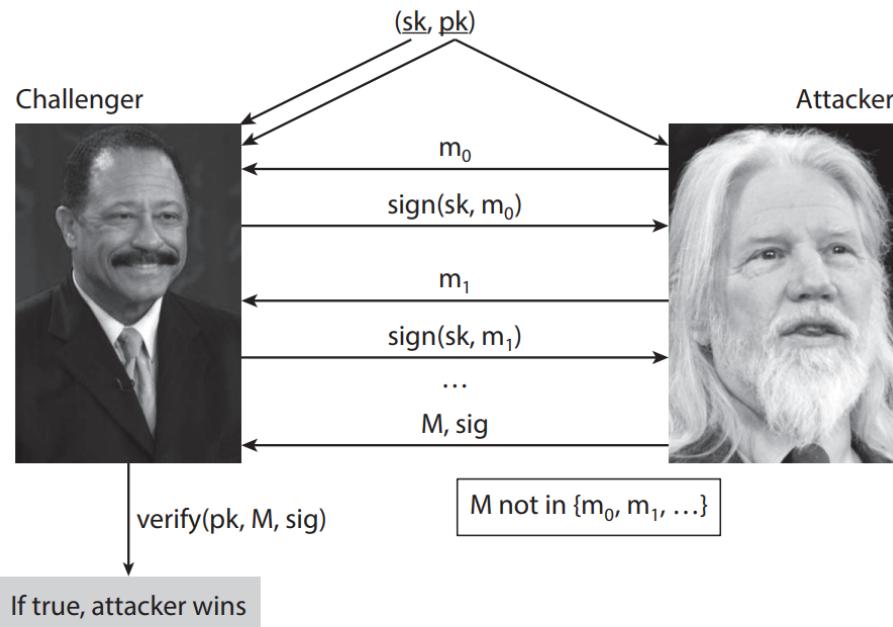


FIGURE 1.9. Unforgeability game. The attacker and the challenger play the unforgeability game. If the attacker is able to successfully output a signature on a message that he has not previously seen, he wins. If he is unable to do so, the challenger wins, and the digital signature scheme is unforgeable.

# Simple (fictitious) cryptocurrencies

# Goofycoin

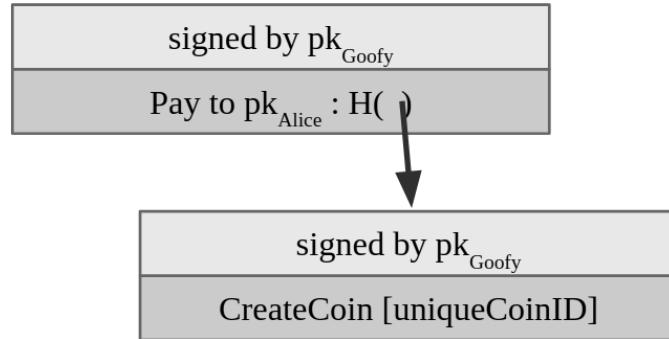
## Rule 1: a designated entity (Goofy) can create new coins

To create a coin, Goofy generates a unique coin ID along with the statement "CreateCoin [uniqueCoinID]".

- Goofy computes the digital signature of this string with his secret signing key.
- The string, together with Goofy's signature, is a coin.
- Anyone can verify that the coin contains Goofy's valid signature of the CreateCoin statement and is therefore a valid coin.

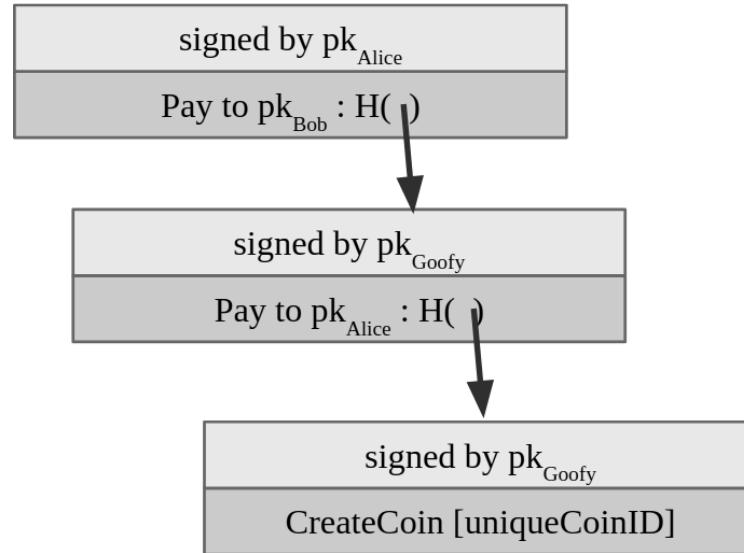
signed by $pk_{\text{Goofy}}$
CreateCoin [uniqueCoinID]



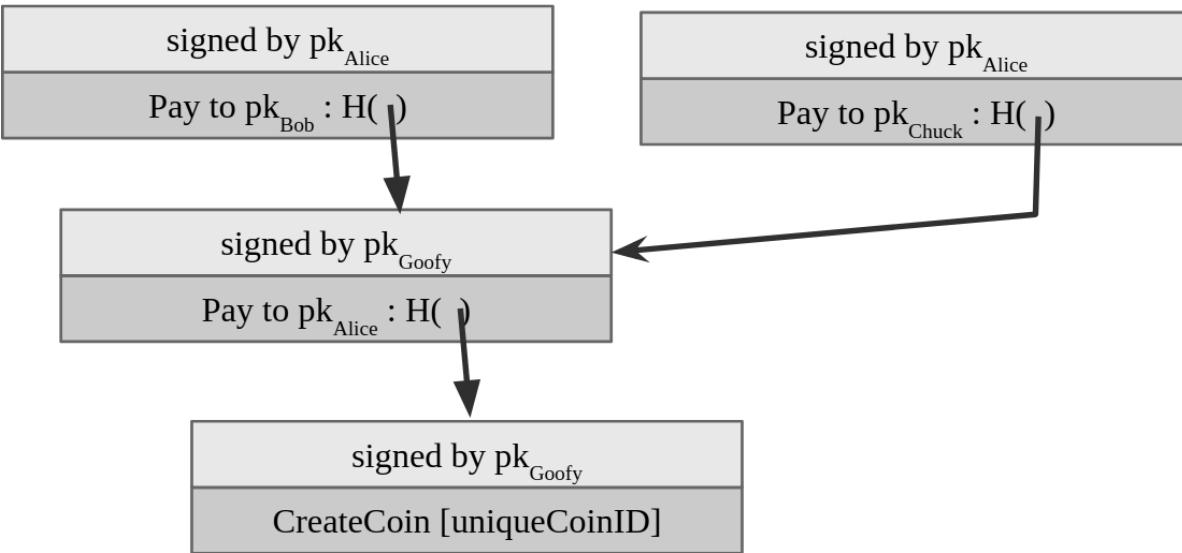


## Rule 2: Whoever owns a coin can transfer it to someone.

- Let's say Goofy wants to transfer a coin he created to Alice.
- To do this, Goofy creates a new statement "Pay this to Alice", where
  - "this" is a hash pointer that references the coin in question,
  - Alice's identity is defined by her public signing key.
- Alice can prove to anyone that she owns the coin because she can present the data structure with Goofy's valid signature.



The recipient can pass on the coin again.



## Double-spending attack

- Let's say Alice passed her coin on to Bob by sending her signed statement, but didn't tell anyone else.
- She could create another signed statement that pays the same coin to Chuck.
- Both Bob and Chuck would have valid-looking claims to be the owner of this coin.

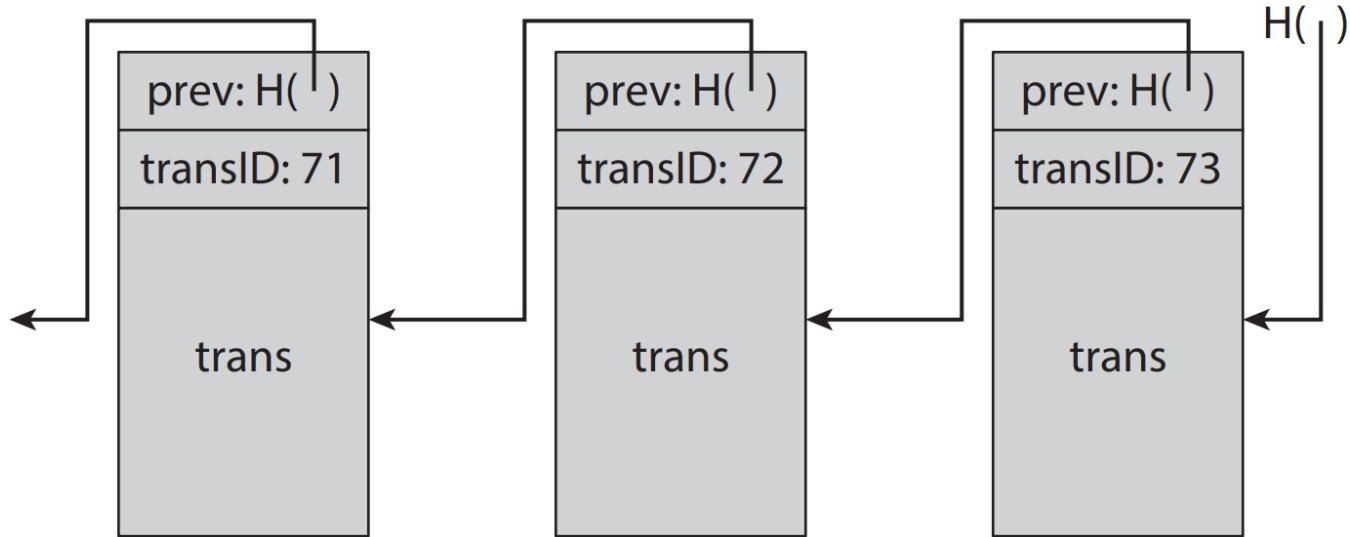
Goofycoin does not solve the **double-spending attack** problem. For this reason, it is **not secure**.

# Scroogecoin

A [designated and trusted entity](#) (Scrooge McDuck) publishes an [append-only ledger](#) containing the history of all transactions.

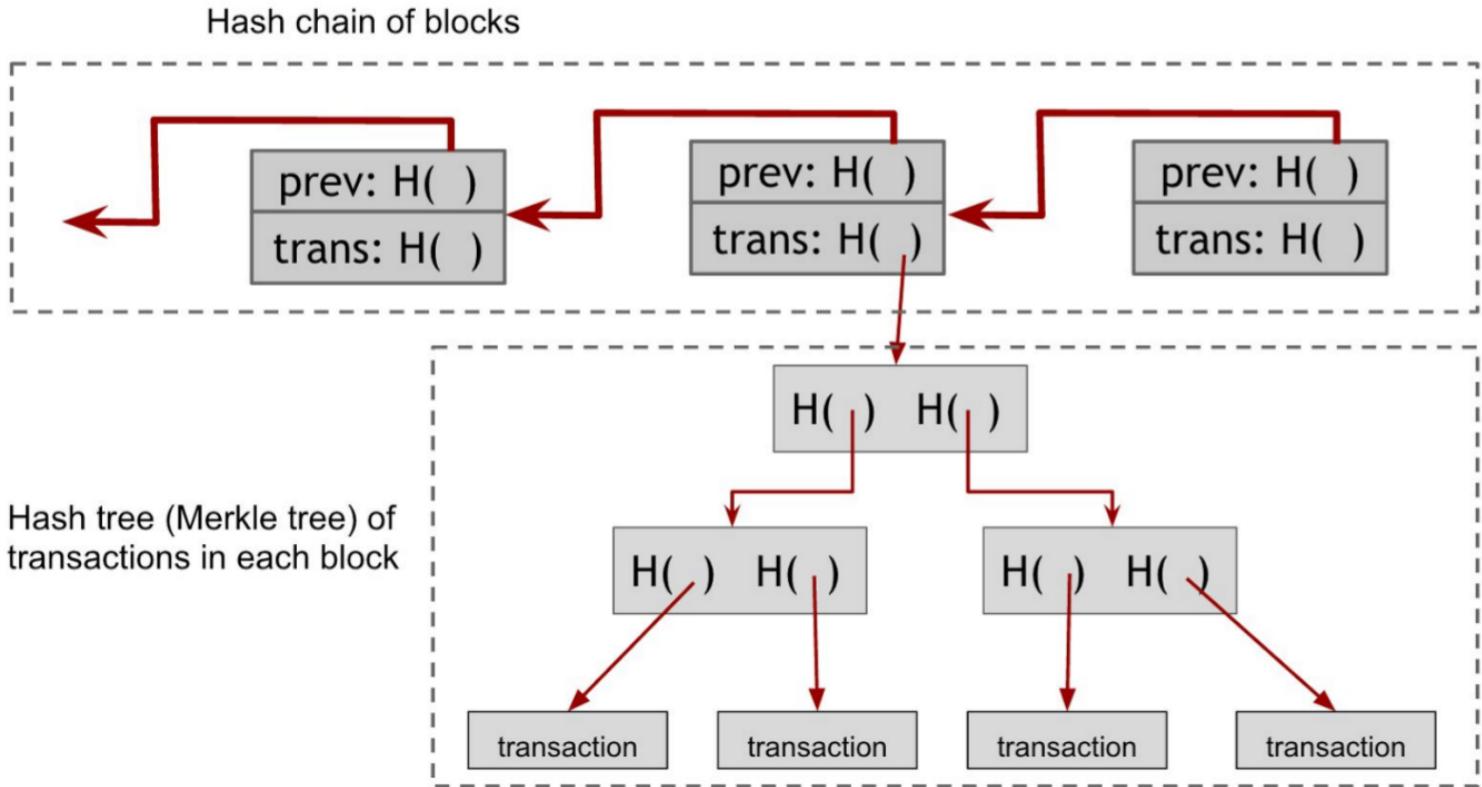
- Append-only ensures that any data written to this ledger will remain forever in the ledger.
- Therefore, this can be used to prevent double spending by requiring that all transactions are written in the ledger before they are accepted.





To implement the append-only ledger, Scrooge makes use of a **blockchain**, which he will digitally sign.

- The blockchain is a series of data blocks, each with one or more transaction(s) in it.
- Each block has the IDs of the transactions, the transaction's contents, and a hash pointer to the previous block.
- Scrooge digitally signs the final hash pointer, which binds all the data in this entire structure, and he publishes the signature along with the blockchain.



In Bitcoin, the blockchain contains two different hash structures.

- The first is a **hash chain of blocks** that links the different blocks to one another.
- The second is internal to each block and is a **Merkle tree of transactions** within the blocks.

A transaction only counts if it is in the block chain **signed by Scrooge**.

- Anybody can verify a transaction was endorsed by Scrooge by checking Scrooge's signature on the block that records the transaction.
- Scrooge makes sure that he does not endorse a transaction that attempts to double spend an already spent coin.
- If Scrooge tries to add or remove a transaction, or to change an existing transaction, it will affect all following blocks published by Scrooge.
  - As long as the latest hash pointer published by Scrooge is monitored, the change will be obvious and easy to catch.

## Coin creation

- Same as for Goofycoin, but we extend the semantics to allow for multiple coins to be created per transaction.
- Coins are referred to by a transaction ID and a coin's serial number in that transaction.

transID: 73	type:CreateCoins	
coins created		
<i>num</i>	<i>value</i>	<i>recipient</i>
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

← coinID 73(0)  
← coinID 73(1)  
← coinID 73(2)

## Payments

A transaction consumes (and destroys) some coins and creates new coins of the same total value.

A transaction is **valid** if:

- The consumed coins are valid.
- The consumed coins have not already been consumed.
- The total value out in the transaction is to equal to the total value in.
- The transaction is validly signed by all the owners of the consumed coins in the transaction.

transID: 73	type:PayCoins			
consumed coinIDs: 68(1), 42(0), 72(3)				
coins created				
num	value	recipient		
0	3.2	0x...		
1	1.4	0x...		
2	7.1	0x...		
signatures				

## **Immutable coins**

Coin cannot be transferred, subdivided or combined.

But we can obtain the same effect by using transactions. E.g., to subdivide:

- create a new transaction;
- consume your coins;
- pay out two new coins (of half the value of the original coin) to yourself.

Scoorge cannot create fake transactions, because he cannot forge other people's signatures.

However,

- he could stop endorsing transactions from some users, denying them service and making their coins unspendable;
- he could refuse to publish transactions unless they transfer some mandated transaction fee to him.
- he can create as many coins for himself as he wants.

Can the system operate **without any central, trusted party?**



*Do not worry.  
I am honest.*

# **Consensus in the blockchain**

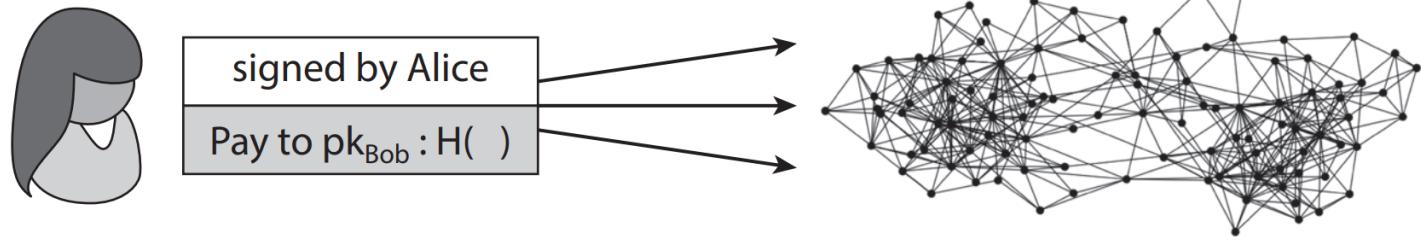
# Decentralization

We want a **decentralized** cryptocurrency system without any central (supposedly) trusted party.

## Solution

- Implement the currency protocol on top of a **peer-to-peer** network of nodes.
- Each node maintains its own copy of the ledger.





When Alice wants to pay Bob, she **broadcasts** the transaction to all nodes in the network. Each node updates its ledger accordingly.

Note that Bob's computer is not (necessarily) in the picture.

- Who maintains the ledger?
- Who has authority over which transactions are valid?
- Who creates new coins?
- Who determines how the rules of the system change?
- How do coins acquire exchange values?

# Consensus

For this peer-to-peer system to work:

- All nodes must have the exact same copy of the ledger.
- Therefore, they must agree on the transactions that are added in the ledger, and in which order.

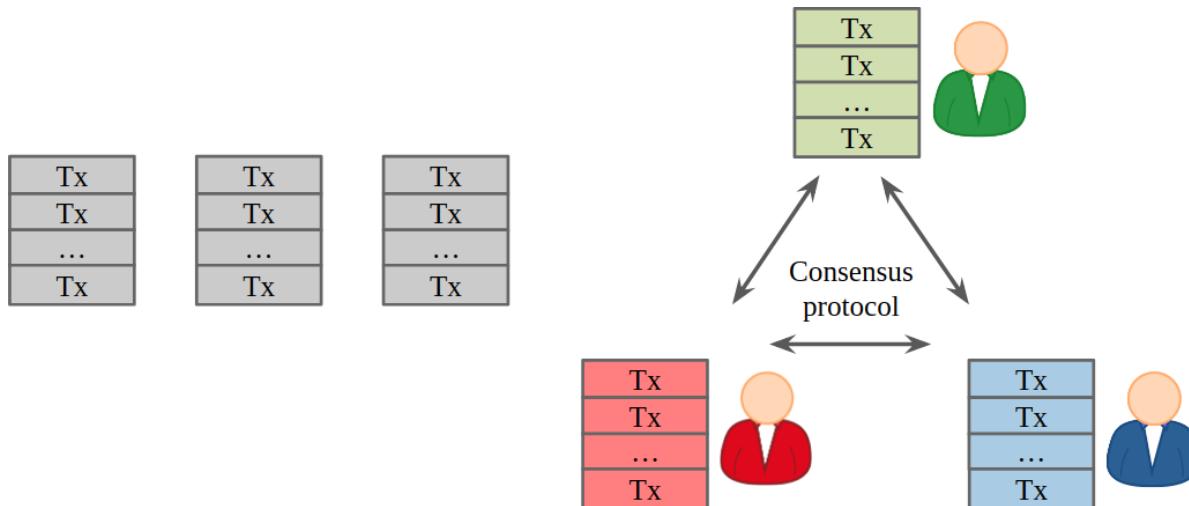
⇒ They must reach **consensus**.

## How consensus could work

At any given time:

- All nodes have a blockchain consisting of a sequence of blocks, each containing a list of transactions they have reached consensus on.
- Each node has a set of outstanding transactions it has heard about.

At regular intervals, every node in the system proposes its own outstanding transaction pool to be included in the next block, using some consensus protocol.



## **Consensus is hard**

- Nodes may crash
- Nodes may be malicious
- Network is highly imperfect
  - Not all pairs of nodes connected
  - Faults in the network
  - Latency (no notion of global time)

Why not simply use a **Byzantine fault-tolerant** variant of **Paxos**?

- It would never produce inconsistent results.
- However
  - there are certain (rare) conditions in which the protocol may fail to make any progress,
  - no solution exists if less than  $\frac{2}{3}$  of the nodes are honest.

## Additional constraints

- **Pseudonymity**: we do not want nodes to have an identity.
- **Sybil attacks**: we do not want an adversary to be able to spawn many nodes (e.g., a majority) and take control of the system.

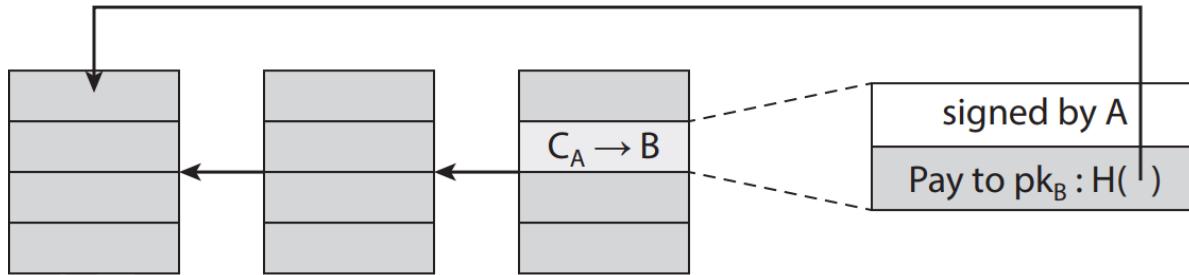
# Implicit consensus

Assume the ability to select a random node in manner that is not vulnerable to Sybil attacks, such that at least 50% of the time an honest node is picked.

## Consensus algorithm

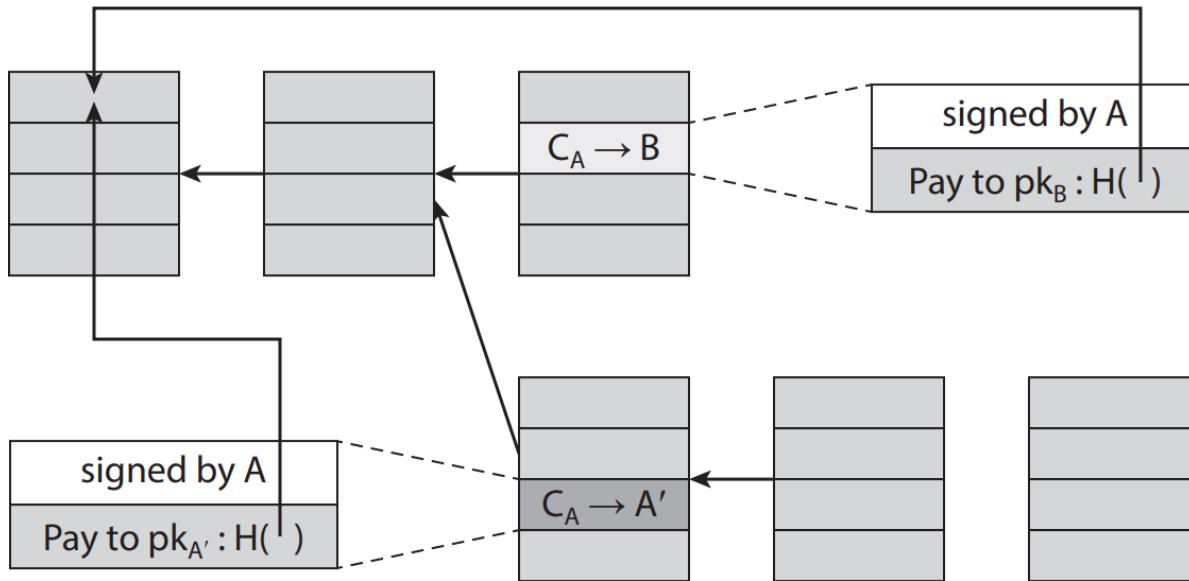
1. New transactions are **broadcast** to all nodes.
2. Each node collects new transactions into a block.
3. In each round, a **random node** gets to broadcast its block.
4. Other nodes accept the block only if all transactions in it are valid (unspent, valid signatures).
5. Nodes express their acceptance of the block by including its hash in the next block they create.

[Q] To what basic abstraction does random selection is an implementation of?



Alice adds an item to her shopping cart on Bob's website. The server requests payment.

- Alice creates a transaction from her address to Bob's and broadcast it to the network.
- An honest node creates the next block and includes this transaction in that block.
- On seeing the transaction included in the blockchain, Bob concludes that Alice has paid him and send the purchased item to Alice.



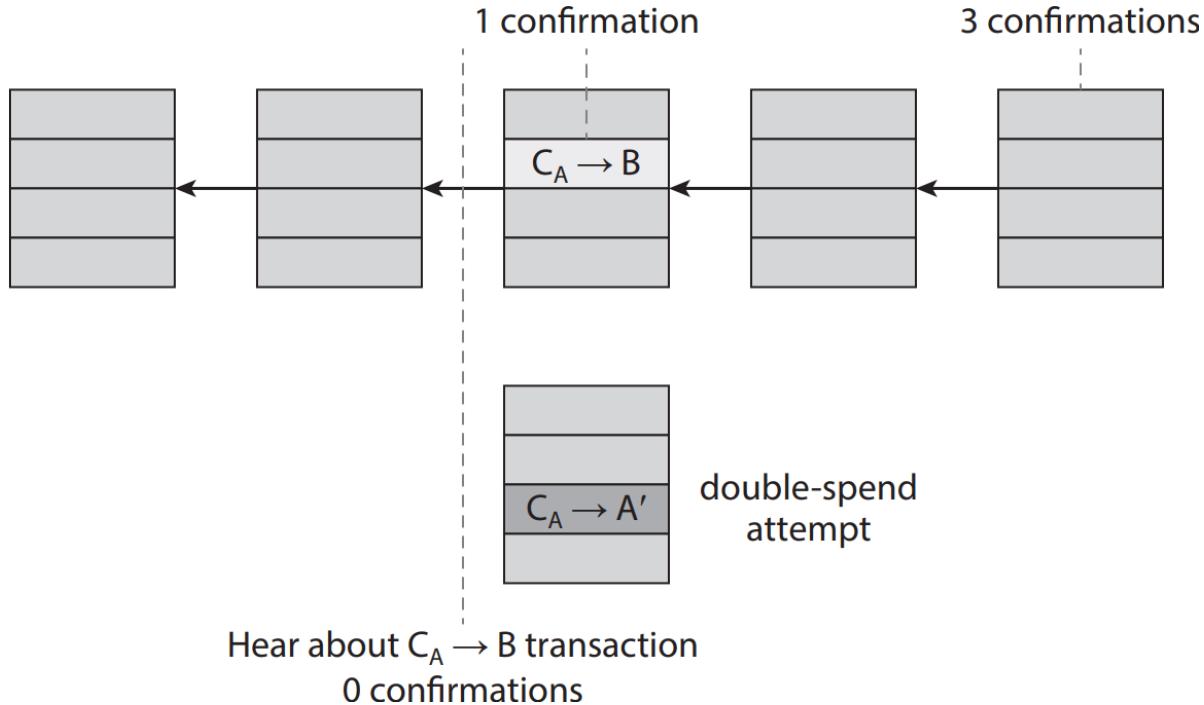
## Double-spend attack

- Assume the next random node happens to be controlled by Alice.
- Since Alice gets to propose the block, she could propose one that ignores the block that contains the payment to Bob.
- Worse, she could make a transaction that transfers the same coin to an address she controls.
- Since the two transactions spend the same coins, only one of them will be included in the blockchain.

The double-spend attack success will depend on which block will ultimately end up on the long-term consensus chain.

## Policy upon forks

- Honest nodes follow the policy that **extends the longest valid branch**.
- In step 4 of implicit consensus, if an honest node discovers that the new block belongs to a longer branch than what it thought was part of the longest branch, then the node locally **reorganizes** its chain.



Bob the merchant's point of view:

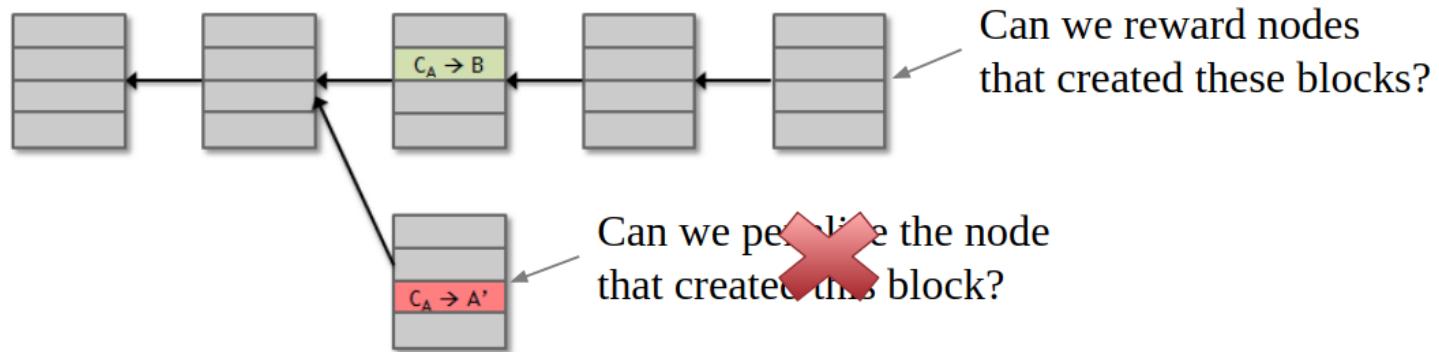
- Double-spend probability **decreases exponentially** with the number of confirmations.
- Common heuristic: wait for 6 confirmations before validating the transaction.

## Recap

- Protection against invalid transactions is cryptographic, but enforced by consensus.
- Protection against double-spending is purely by consensus.
- We are never 100% certain that a transaction is part of the consensus branch.  
**The guarantee is probabilistic.**
  - Even with 1% of the total hashing power, Alice would have a hard time cheating on Bob.
  - The probability of mining six blocks in a row is  $0.01^6 = 10^{-12}$ .

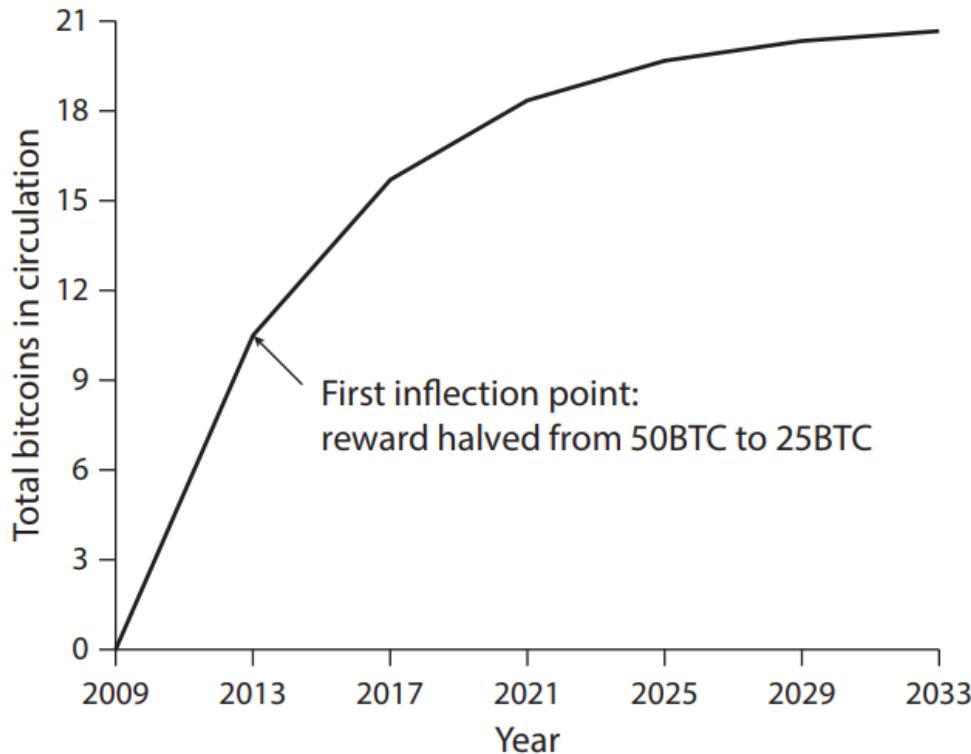
# Incentives

- Assuming node honesty is problematic.
- Instead, can we build **incentives** for nodes to behave honestly?



## Incentive 1: block reward

- The creator of a block gets to
  - include a special coin-creation transaction in the block
  - choose the recipient address of this transaction.
- The value is fixed: currently 12.5 coins, but halves every 210000 blocks (~ every 4 years).
- The block creator gets to collect the reward only if the blocks end up on the long-term consensus branch.



Total supply of coins with time. The block reward is cut in half every 4 years, limiting the total supply to 21 millions.

## **Incentive 2: transaction fees**

- The creator of a transaction can choose to make the output value less than input value.
- The remainder is a transaction fee and goes to the block creator.
- Purely voluntary.

# Proof of work

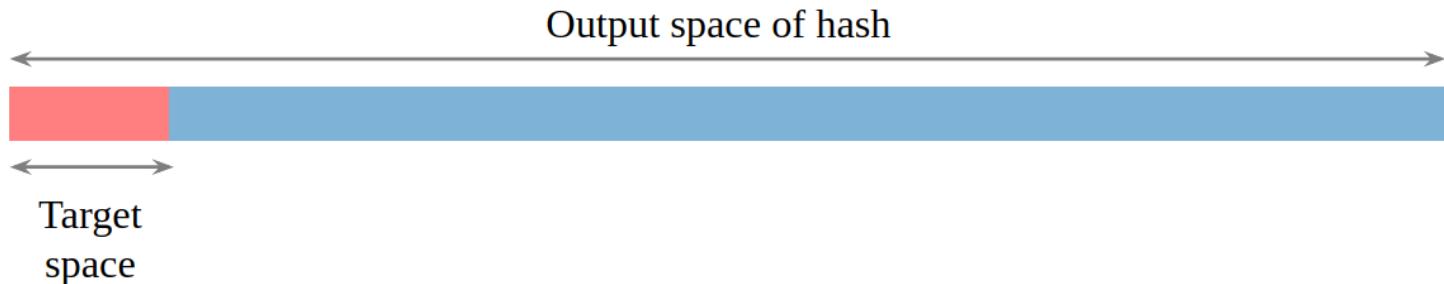
How does one select a node at random without being vulnerable to Sybil attacks?

- Approximate the selection of a random node by instead selecting nodes in proportion to a resource that (we hope) nobody can monopolize.
  - in proportion to computer power: **proof of work** (PoW)
  - in proportion to currency ownership: **proof of stake** (PoS)

How does one select nodes in proportion to their computing power?

- Allow nodes to compete with one another by using their computing power.
- This results in nodes being picked in proportion to that capacity.

## PoW with hash puzzles



To create a block, find a **nonce** such that

$$H(\text{nonce} \parallel \text{previous hash} \parallel \text{tx}_1 \parallel \text{tx}_2 \parallel \dots) < T$$

for some target  $T \ll 2^{256} \approx 10^{77}$ .

- If the hash function  $H$  is secure, the only way to succeed is to try enough nonces until getting lucky.
- Node creators are called **miners**.

## **Property 1: difficult to compute**

- As of 2018, the expected number of hashes to mine a block is  $3 \times 10^{22}$ .
- Only some nodes bother to compete.

## Property 2: parameterizable cost

- The target  $T$  is adjusted periodically as a function of how much hashing power has been deployed in the system by the miners.
- The goal is to maintain an average time of 10 minutes between blocks.

## Property 3: trivial to verify

- The **nonce** must be published as part of the block.
- Hence, anyone can verify that

$$H(\text{nonce} \parallel \text{previous hash} \parallel \text{tx}_1 \parallel \text{tx}_2 \parallel \dots) < T$$

# 51% attack

- The whole system relies on the assumption that a majority of miners, weighted by hash power, follow the protocol.
- Therefore, the protocol is vulnerable to attackers that would detain 51% or more of the total hashing power.

# **Bitcoin and friends**

# Bitcoin

- The cryptocurrency protocol presented so far corresponds to the general protocol used for **Bitcoin** (BTC).
- Bitcoin was invented by an unknown person (or group of people) using the name of Satoshi Nakamoto.
- It was released as an [open source software](#) in 2009.
- Its main goal is to establish a decentralized digital currency that is not tied to a bank or government.
- The estimated number of unique users is 3-6 million.

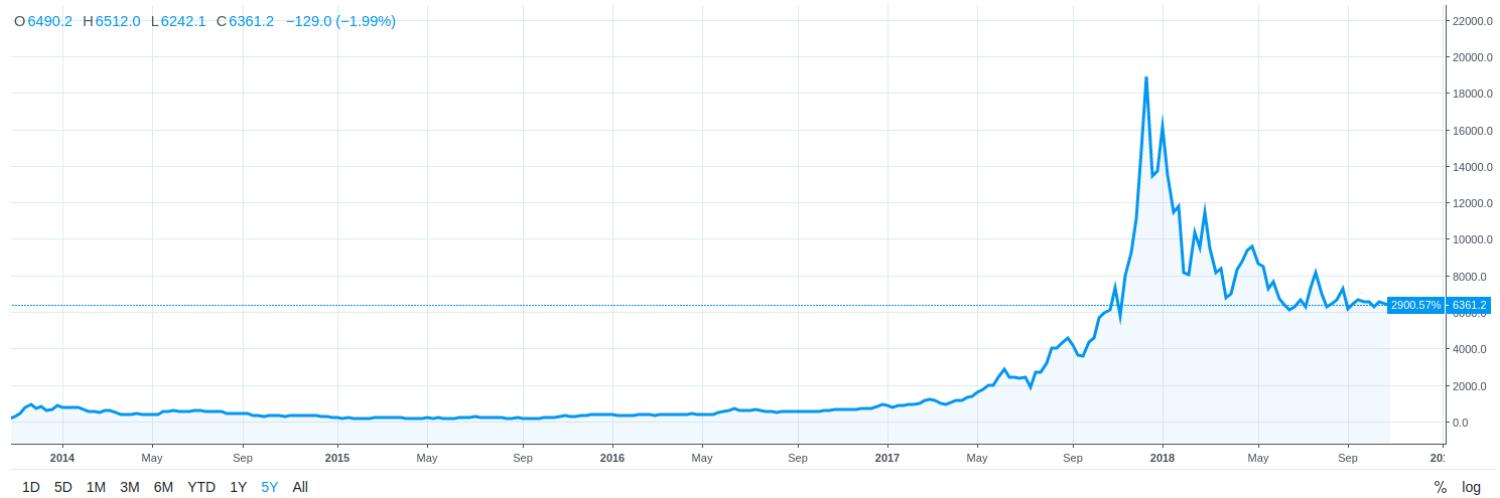


## Trading

- Bitcoin can be used to buy or sell goods.
- Bitcoin can be bought and sold like any other currency.
- Bitcoin ATMs even exist in some countries!



## Volatility



As any currency, BTC can be exchanged for other currencies (e.g. USD or EUR).

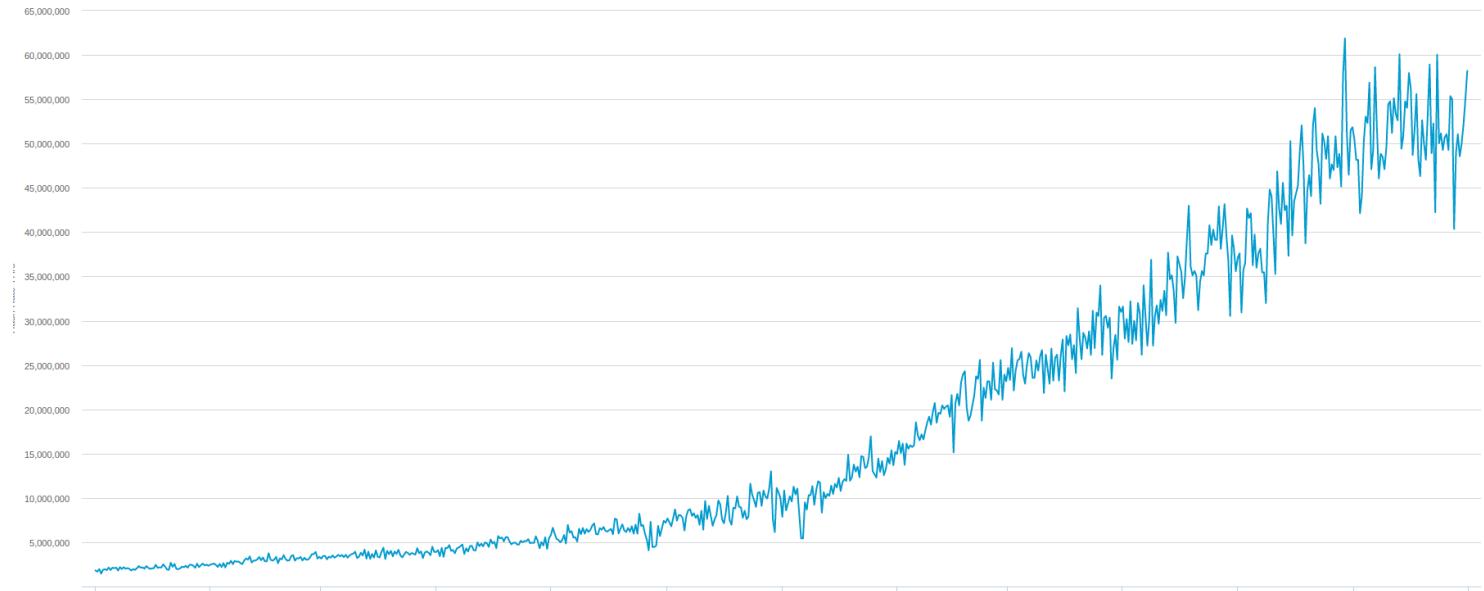
- The current exchange rate (November 1, 2018) is 1 BTC = 6358 USD.
- The price is highly volatile and subject to speculation.

## Mining as a business



*A mining farm.*

## Extreme competition



*Global hash rate over time.*

# Energy sinkhole

**ABC NEWS**

Just In Politics Australia World Business Sport Science Arts Analysis Fact Check More

Print Email

**ANALYSIS**

## Bitcoin mining likely uses more energy than it takes to keep New Zealand's lights on

The Conversation By John Quiggin, University of Queensland Updated 20 minutes ago

The recent upsurge in the price of Bitcoin seems to have finally awakened the world to the massively destructive environmental consequences of this bubble.

These consequences were pointed out as long ago

**BITCOIN EXPLAINED**



What the bitcoin bubble tells us about ourselves



**HOME > Business > Technology**

## Bitcoin miners' power needs could soar higher than global energy production by 2020

The electricity required to mine the cryptocurrency is now higher than Serbia's annual power consumption



**EDITOR'S PICKS**



## Mining economics

If

$$\text{mining reward} > \text{mining cost}$$

then the miner makes a profit

where

$$\text{mining reward} = \text{block reward} + \text{tx fees}$$

$$\text{mining cost} = \text{hardware cost} + \text{operating costs (electricity, cooling, etc.)}$$

See also the [Bitcoin Mining Profit Calculator](#).





# Other cryptocurrencies

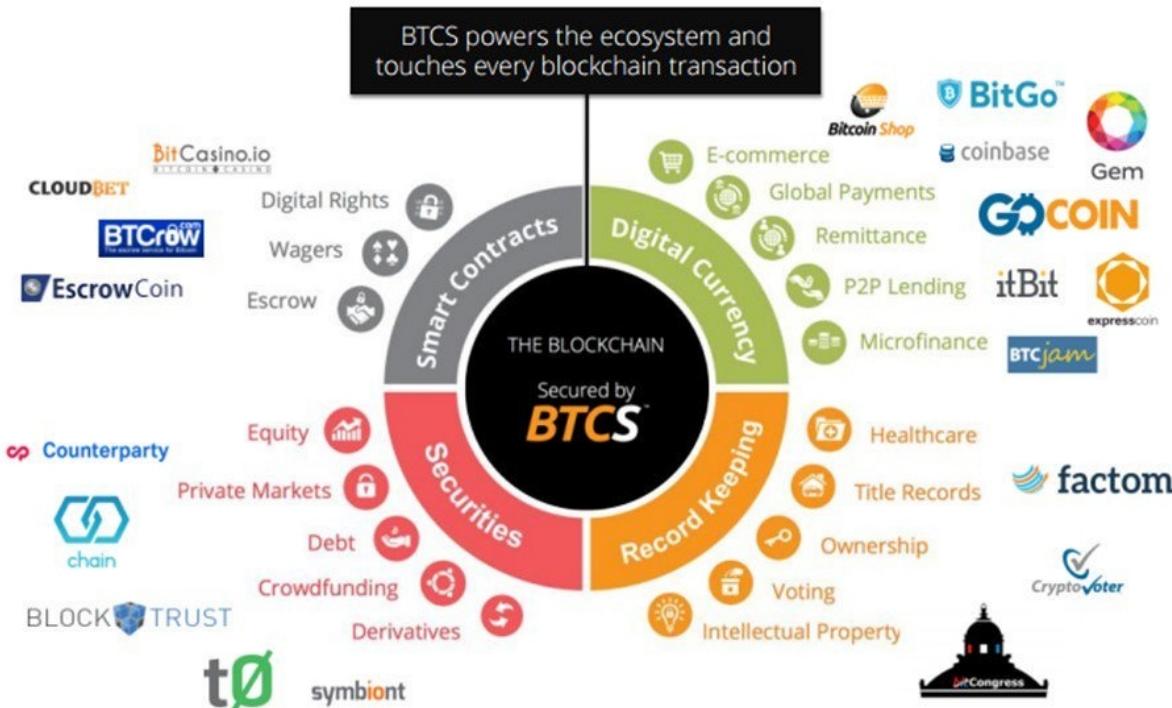
BTC is only one of many cryptocurrencies. Popular cryptocurrencies include:

- ETH
- XRP
- LTC

# Applications

A blockchain is nothing else than a continuously growing list of records.

- It is secure by design, with high Byzantine fault tolerance.
- Blockchains can therefore be used to store any kind sensitive information that should not be altered.



# Summary

- The **blockchain** is a linked list with hash pointers.
- It can be used for implementing **a ledger** that stores sensitive information that should not be tampered with.
- Decentralization requires **consensus**.
- In Bitcoin, consensus is achieved by **proof-of-work**, which provides high Byzantine fault tolerance.



# References

- Nakamoto, Satoshi. "Bitcoin: A peer-to-peer electronic cash system." (2008).
- Narayanan, Arvind, et al. Bitcoin and cryptocurrency technologies: a comprehensive introduction. Princeton University Press, 2016.

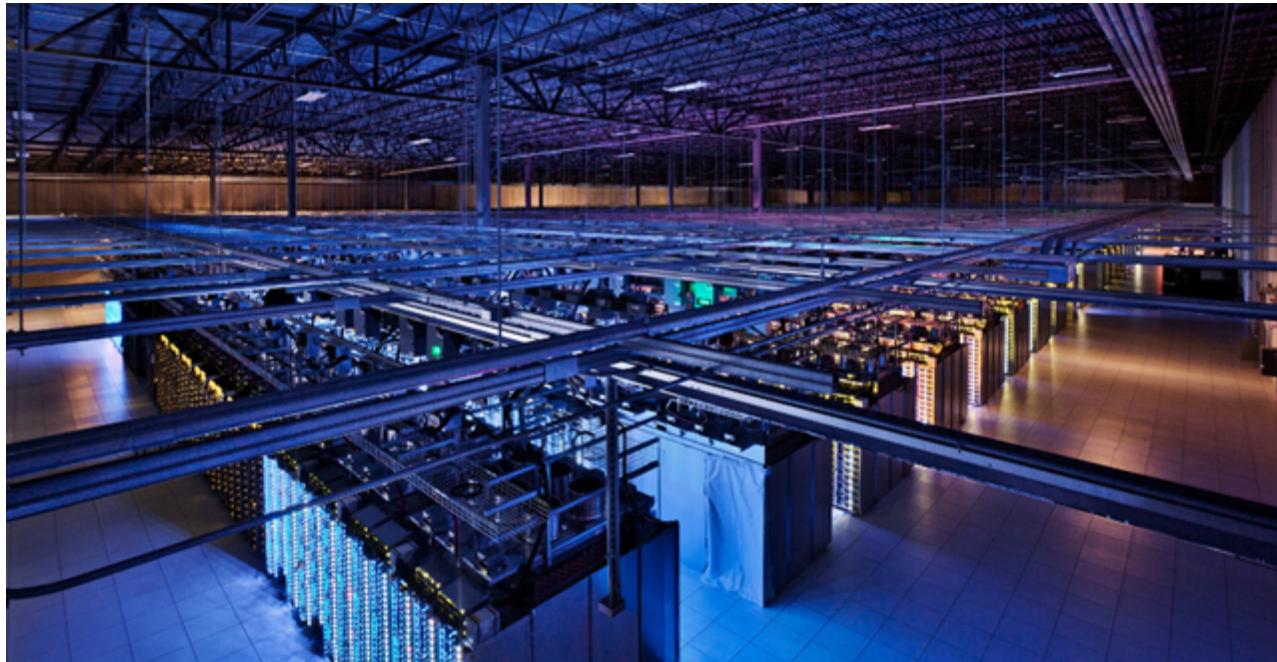
# Large-scale Data Systems

Lecture 7: Cloud computing

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)



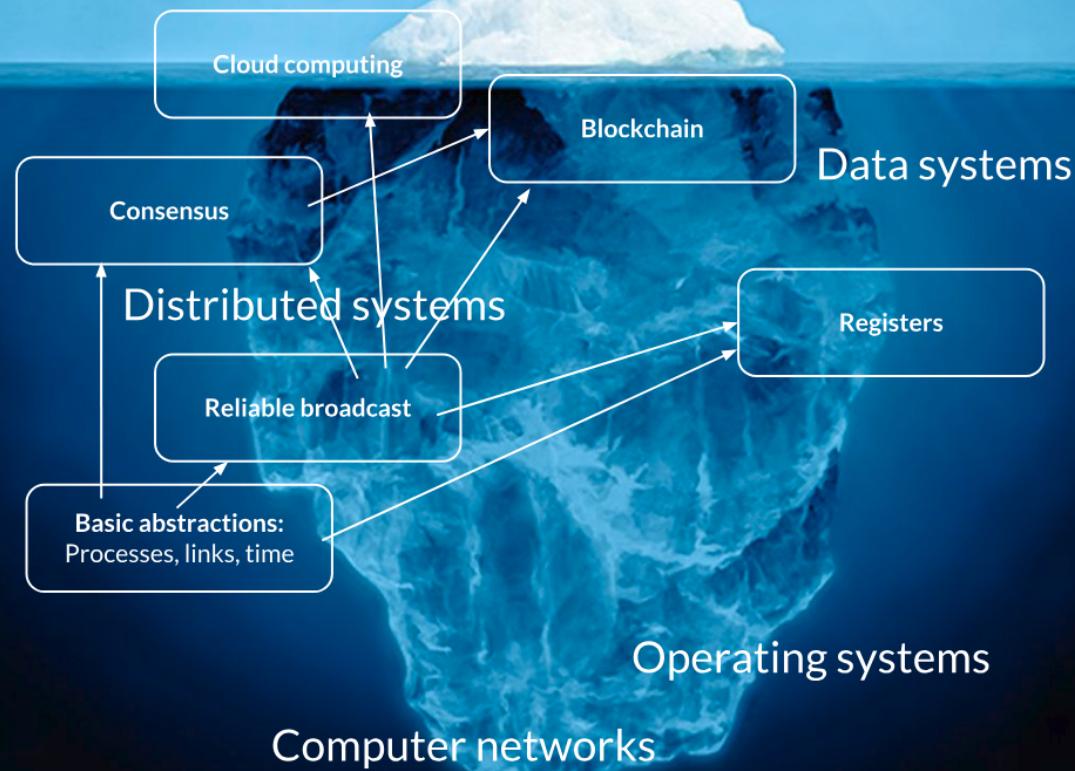
# Today



How do we program this thing?

- MapReduce
- Spark

# Data science Machine Learning Visualization



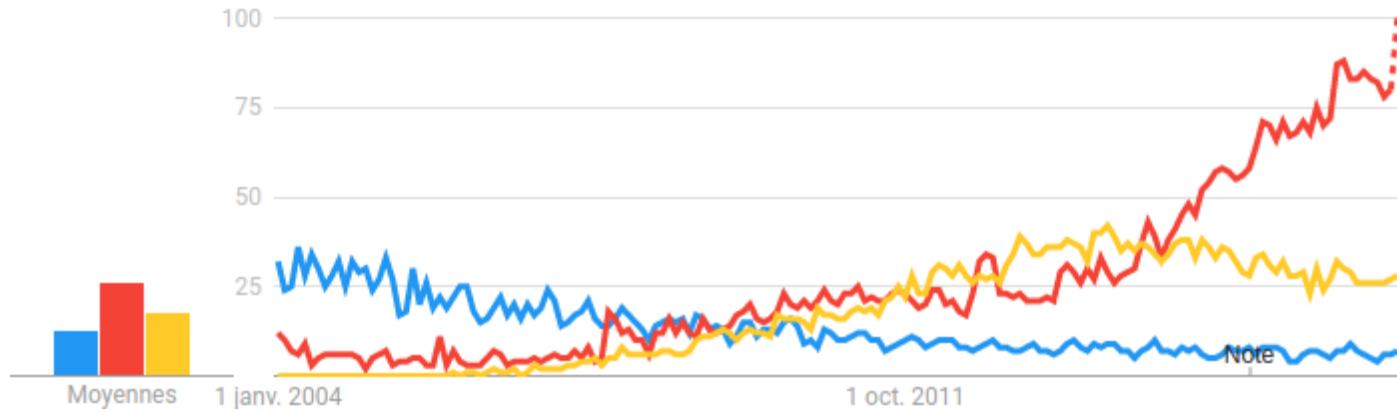
# Dealing with lots of data

- Example:
  - $130 + \text{trillion web pages} \times 50\text{KB} = 6.5 \text{ exabytes}$ .
  - $\sim 6500000$  hard drives ( $1\text{TB}$ ) just to store the web.
- Assuming a data transfer rate of  $200\text{MB/s}$ , it would require  $1000+$  years for a single computer to read the web!
  - And even more to make any useful usage of this data.
- Solution: **spread** the work over **many** machines.

# Traditional network programming

- Message-passing between nodes (MPI, RPC, etc).
- **Really hard to do at scale (for 1000s of nodes):**
  - How to **split** problem across nodes?
    - Important to consider network and data locality.
  - How to deal with **failures**?
    - a 10000-node clusters sees 10 faults/day.
  - Even without failure: **stragglers**.
    - Some nodes might be much slower than others.

● mpi ● spark ● hadoop



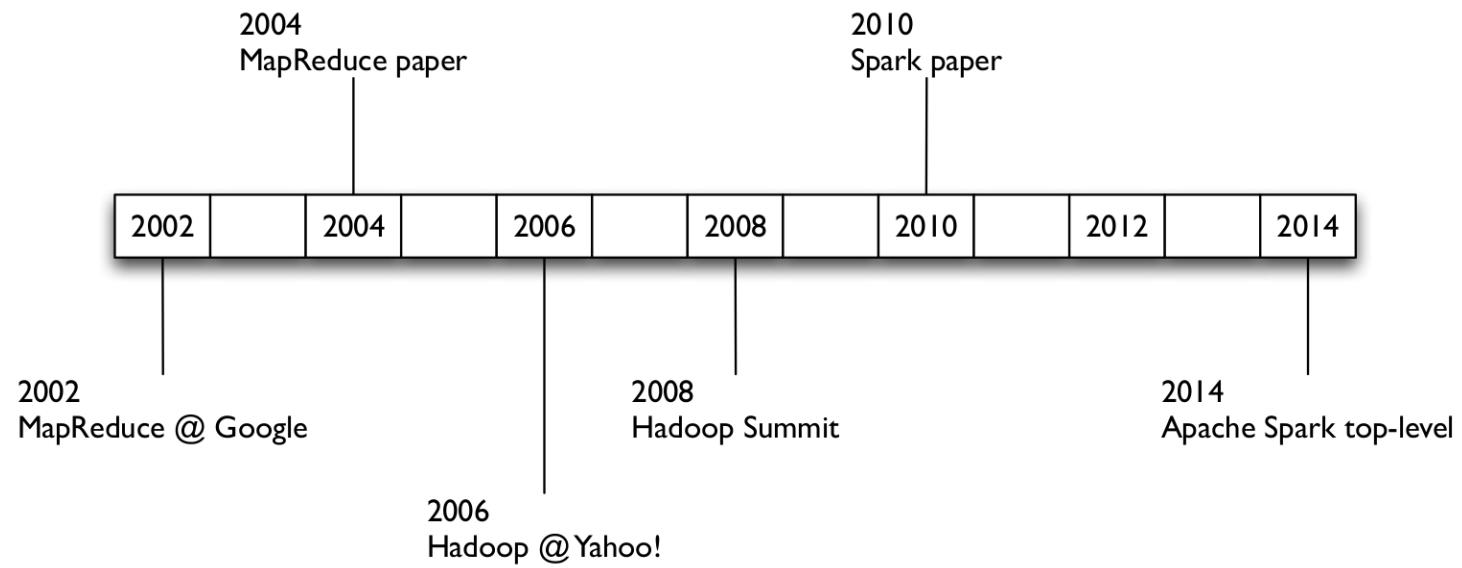
*Almost nobody does message-passing anymore!\**

\*: except in niches, like scientific computing.

# Data-parallel models

- Restrict and simplify the programming interface so that the system can do more automatically.
- "Here is an operation, run it on all of the data".
  - I do not care where it runs (you schedule that).
  - In fact, feel free to run it twice on different nodes if that can help.

# History



# MapReduce

# What is MapReduce?

MapReduce is a parallel programming model for processing distributed data on a cluster.

It comes with a simple high-level API limited two operations: map and reduce, as inspired by Lisp primitives:

- map: apply function to each value in a set.
  - (map 'length '(() (a) (a b) (a b c))) → (0 1 2 3)
- reduce: combines all the values using a binary function.
  - (reduce #'+ '(1 2 3 4 5)) → 15

- MapReduce is best suited for **embarrassingly parallel** tasks.
  - When processing can be broken into parts of equal size.
  - When processes can concurrently work on these parts.
- This abstraction makes it possible to not worry about handling
  - parallelization
  - data distribution
  - load balancing
  - fault tolerance

# Programming model

- **Map**: input key/value pairs → intermediate key/value pairs
  - User function gets called for each input key/value pair.
  - Produces a set of intermediate key/value pairs.
- **Reduce**: intermediate key/value pairs → result files
  - Combine all intermediate values for a particular key through a user-defined function.
  - Produces a set of merged output values.

# Under the hood

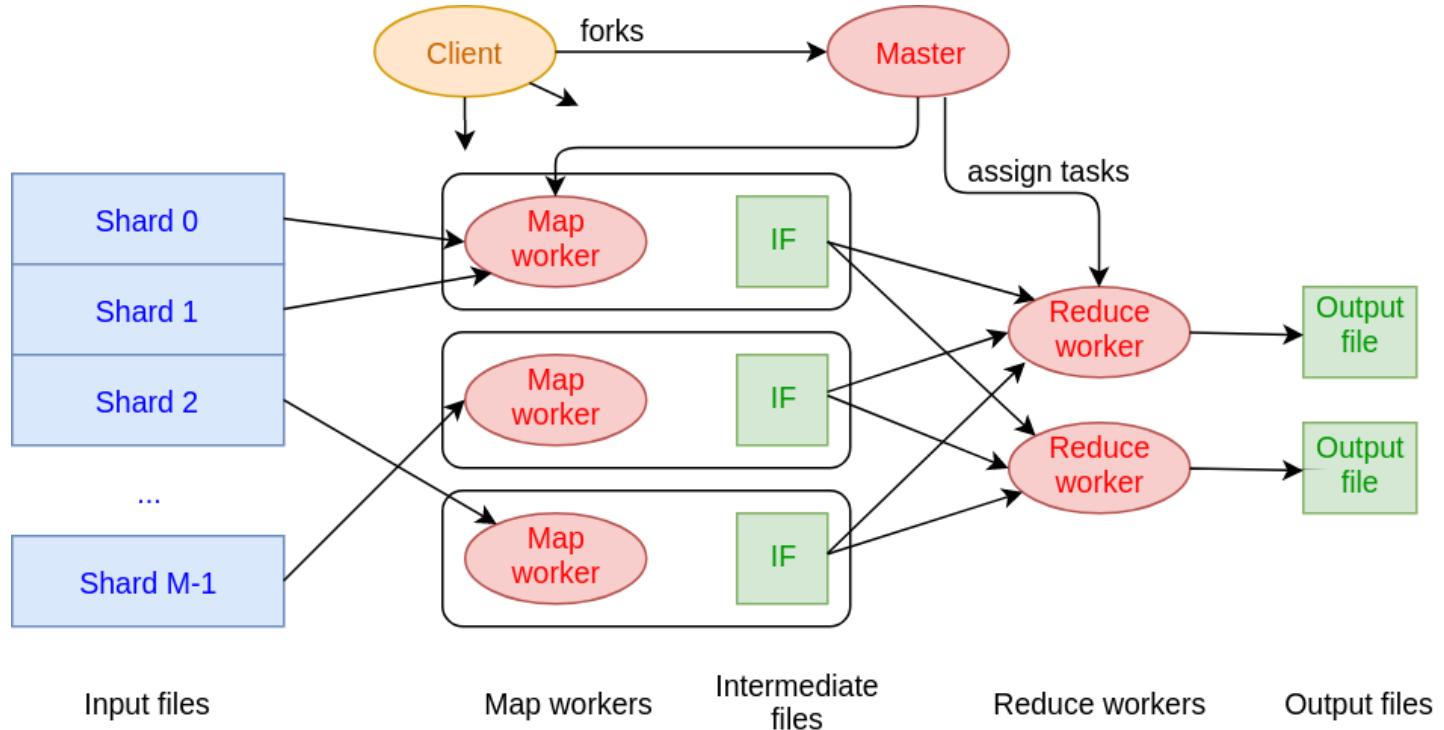
## Map worker

- Map:
  - Map calls are distributed across machines by automatically **partitioning** the input data into  $M$  **shards**.
  - Parse the input shards into input key/value pairs.
  - Process each input pair through a user-defined map function to produce a set of intermediate key/value pairs.
  - Write the result to an intermediate file.
- Partition:
  - Assign an intermediate result to one of  $R$  reduce tasks based on a partitioning function.
    - Both  $R$  and the partitioning function are user defined.

## Reduce worker

- Sort:
  - Fetch the relevant partition of the output from all mappers.
  - Sort by keys.
    - Different mappers may have output the same key.
- Reduce:
  - Accept an intermediate key and a set of values for the key.
  - For each unique key, combine all values through a user-defined reduce function to form a smaller set of values.

## Overview

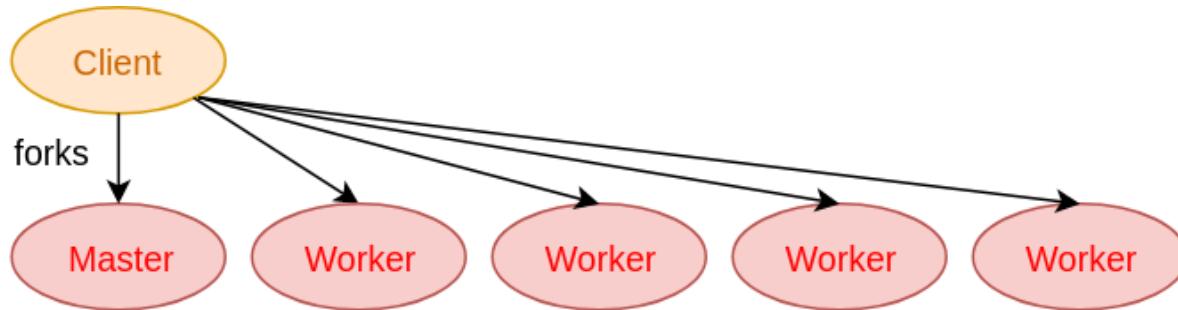


## Step 1: Split input files

Shard 0	Shard 1	Shard 2	...	Shard M-1
---------	---------	---------	-----	-----------

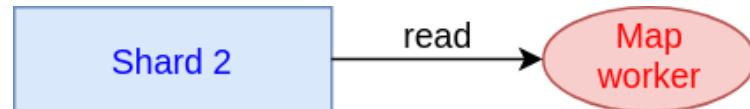
- Break up the input data into  $M$  shards (typically 64MB).

## Step 2: Fork processes



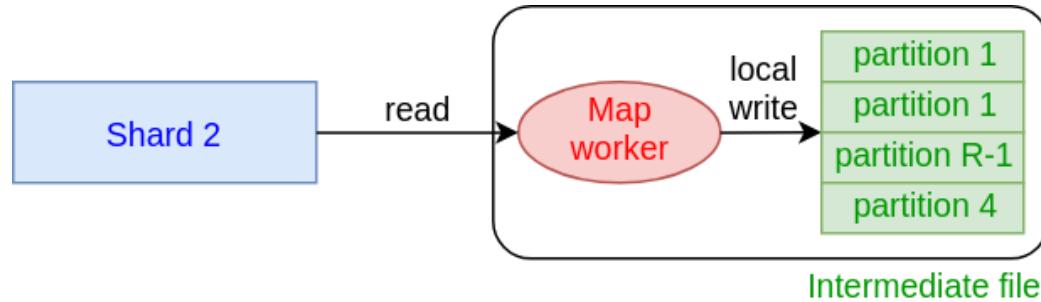
- Start up many copies of the program on a cluster of machines.
  - 1 master: scheduler and coordinator
  - Lots of workers
- Idle workers are assigned either:
  - map tasks
    - each works on a shard
    - there are  $M$  map tasks
  - reduce tasks
    - each works on intermediate files
    - there are  $R$  reduce tasks

## Step 3: Map task



- Read content of the input shard assigned to it.
- Parse key/value pairs  $(k, v)$  out of the input data.
- Pass each pair to a **user-defined** map function.
  - Produce (one or more) intermediate key/value pairs  $(k', v')$ .
  - These are buffered in memory.

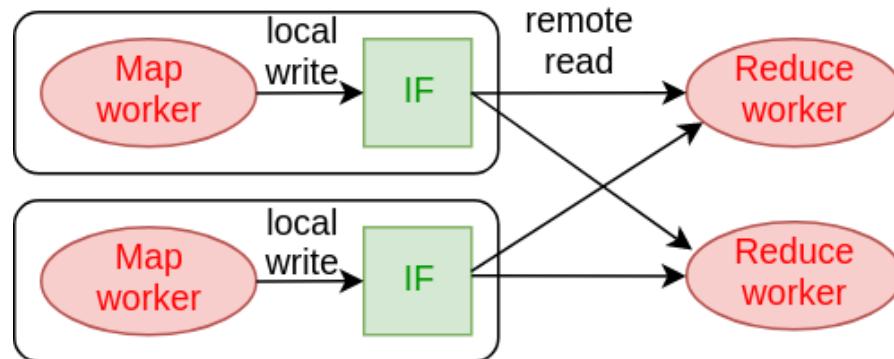
## Step 4: Create intermediate files



- Intermediate key/value pairs  $(k', v')$  produced by the user's map function are periodically written to **local** disk.
  - These files are partitioned into  $R$  regions by a partitioning function, one for each reduce task.
    - e.g.,  $\text{hash}(\text{key}) \bmod R$
- Notify master when complete.
  - Pass locations of intermediate data to the master.
  - Master forwards these locations to the reduce workers.

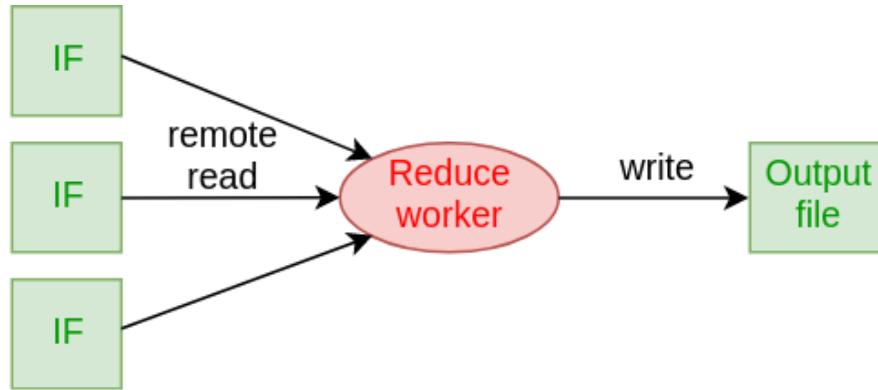
[Q] What is the purpose of the partitioning function?

## Step 5: Sorting/Shuffling



- Reduce worker get notified by master about the location of the intermediate files associated to their partition.
- RPC to read the data from the local disks for the map workers.
- When the reduce worker reads intermediate data for its partition:
  - it sorts the data by intermediate keys  $k'$ .
  - all occurrences  $v'_i$  associated to a same key are grouped together.

## Step 6: Reduce tasks

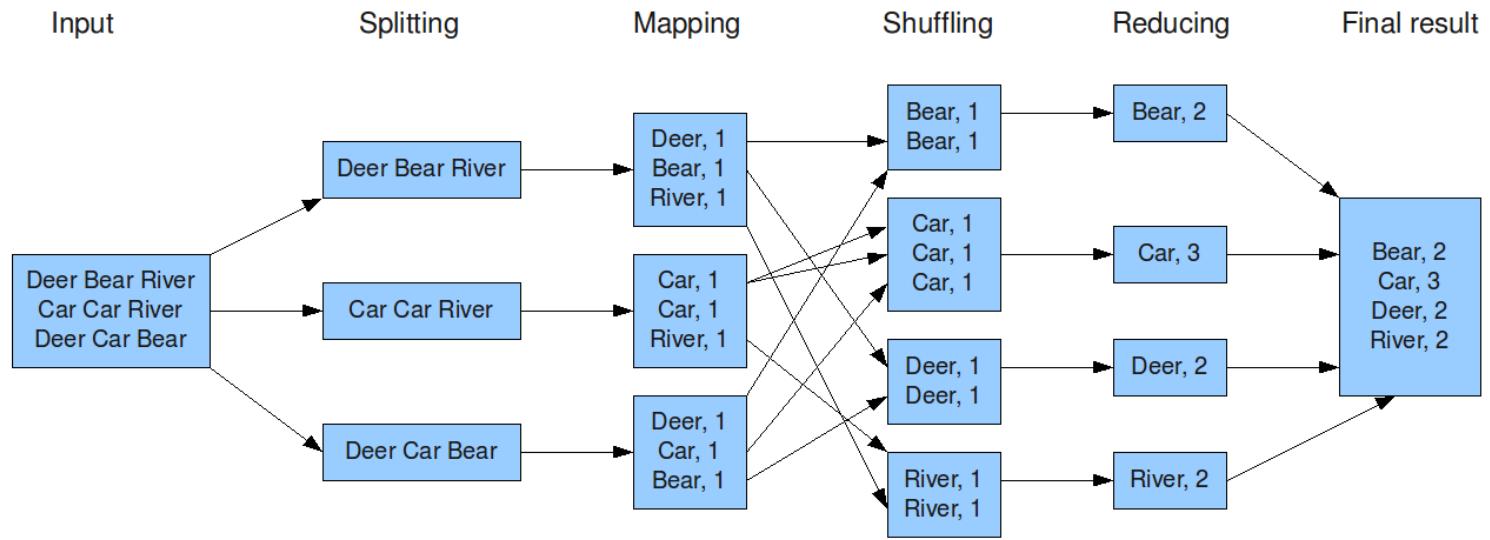


- The sorting phase grouped data sharing a unique intermediate key.
- The **user-defined** reduce function is given the key and the set of intermediate values for that key.
  - $(k', (v'_1, v'_2, v'_3, \dots))$
- The output of the reduce function is appended to an output file.

## Step 7: Return to user

- When all Map and Reduce tasks have completed, the master wakes up the user program.
- The MapReduce call in the user program returns and the program can resume execution.
  - The output of the operation is available in *R* output files.

## Example: Counting words



See also the [Hadoop tutorial](#).

## Other examples

- **Distributed grep**

- Search for words in lots of documents.
- Map: emit a line if it matches a given pattern. Produce  $(file, line)$  pairs.
- Reduce: copy the intermediate data to the output.

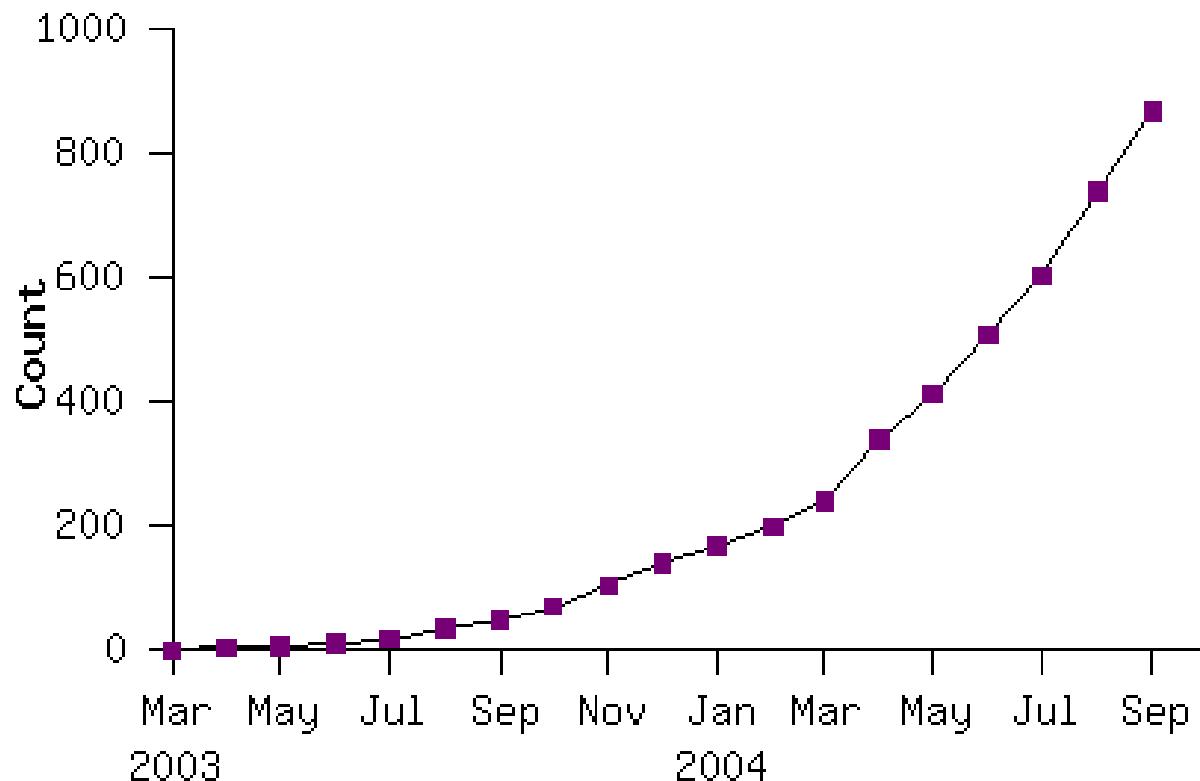
- **Count URL access frequency**

- Find the frequency of each URL in web logs.
- Map: process logs of web page access. Produce  $(url, 1)$  pairs.
- Reduce: add all values for the same URL.
  - Is this efficient?

- **Reverse web-link graph**

- Find where page links come from.
- Map: output  $(target, source)$  pairs for each link *target* in a web page *source*.
- Reduce: concatenate the list of all source URLs associated with a target.

# Wide applicability



Number of MapReduce programs in Google code source tree.

# Fault tolerance

- Master **pings** each worker periodically.
  - If no response is received within a certain delay, the worker is marked as **failed**.
  - Map or Reduce tasks given to this worker are reset back to the initial state and rescheduled for other workers.
  - Task completion is committed to master to keep track of history.

[Q] What abstraction does this use?

[Q] What if the master node fails? How would you fix that?

# Redundant execution

- Slow workers significantly **lengthen completion time**
  - Because of other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time

# Locality

- Input and output files are stored on a distributed file system.
  - e.g., GFS or HDFS.
- Master tries to schedule Map workers near the data they are assigned to.
  - e.g., on the same machine or in the same rack.
- This results in thousands of machines reading input at local disk speed.
  - Without this, rack switches limit read rate.

# MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

*Google, Inc.*

## Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

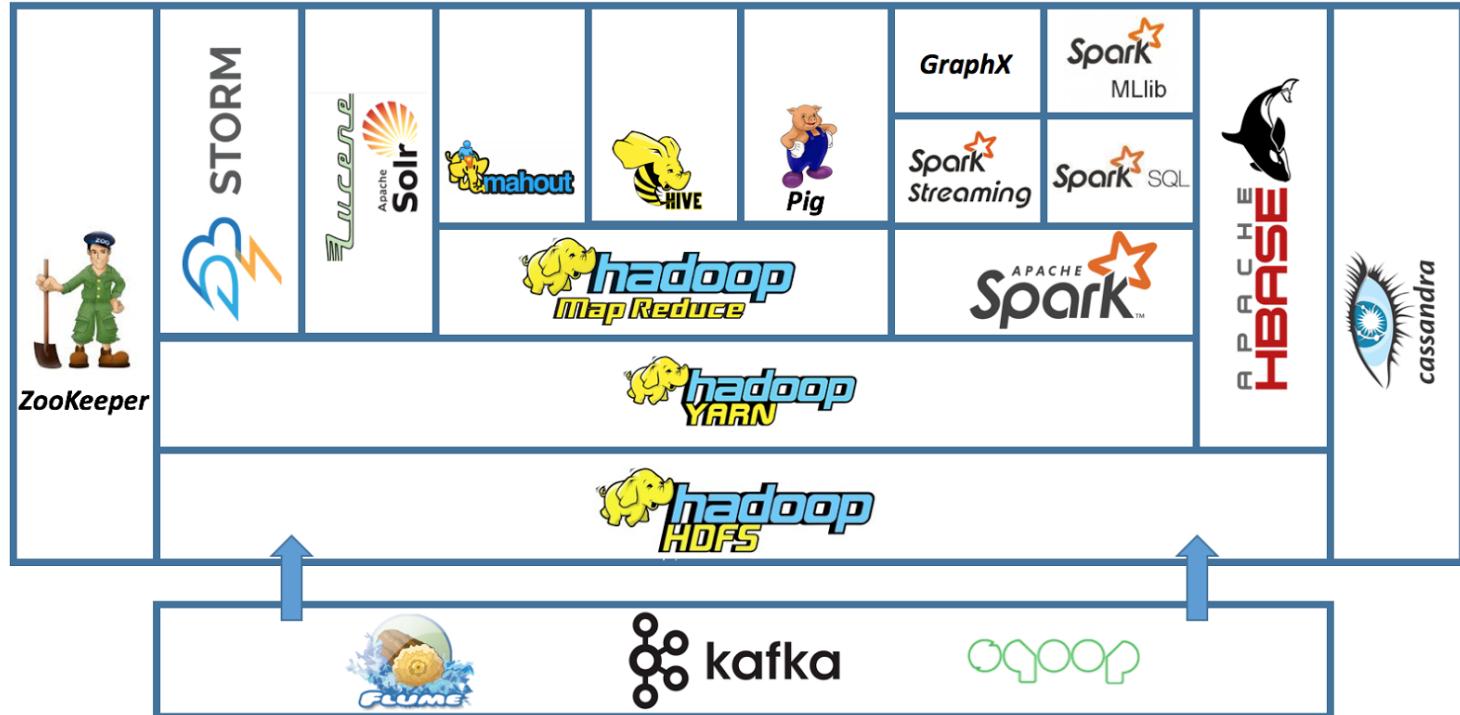
Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical “record” in our input in order to compute a set of intermediate key/value pairs, and then

*Google, 2004.*

# Hadoop Ecosystem



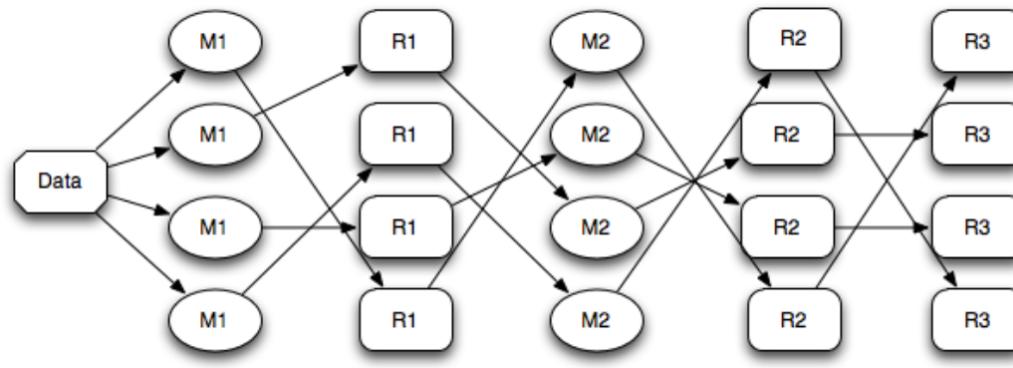
- **Hadoop HDFS**: A distributed file system for reliably storing huge amounts of unstructured, semi-structured and structured data in the form of files.
- **Hadoop MapReduce**: A distributed algorithm framework for the parallel processing of large datasets on **HDFS** filesystem. It runs on Hadoop cluster but also supports other database formats like **Cassandra** and **HBase**.
- **Cassandra**: A key-value pair NoSQL database, with column family data representation and asynchronous masterless replication.
  - Cassandra is built upon an architecture similar to a DHT.
- **HBase**: A key-value pair NoSQL database, with column family data representation, with master-slave replication. It uses HDFS as underlying storage.
- **Zookeeper**: A distributed coordination service for distributed applications.
  - It is based on a **Paxos algorithm** variant called Zab.

- **Pig**: Pig is a scripting interface over MapReduce for developers who prefer scripting interface over native Java MapReduce programming.
- **Hive**: Hive is a SQL interface over MapReduce for developers and analysts who prefer SQL interface over native Java MapReduce programming.
- **Mahout**: A library of machine learning algorithms, implemented on top of MapReduce, for finding meaningful patterns in HDFS datasets.
- **Yarn**: A system to schedule applications and services on an HDFS cluster and manage the cluster resources like memory and CPU.
- **Flume**: A tool to collect, aggregate, reliably move and ingest large amounts of data into HDFS.
- ... and many others!

# Spark

# MapReduce programmability

- Most applications require multiple MR steps.
  - Google indexing pipeline: 21 steps
  - Analytics queries (e.g., count clicks and top-K): 2-5 steps
  - Iterative algorithms (e.g., PageRank): 10s of steps
- Multi-step jobs create **spaghetti** code
  - 21 MR steps → 21 mapper + 21 reducer classes
  - Lots of boilerplate code per step



*Chaining MapReduce jobs.*

# Problems with MapReduce

- Over time, MapReduce use cases showed two major limitations:
  - not all algorithms are suited for MapReduce.
    - e.g., a **linear dataflow** is forced.
  - it is difficult to use for exploration and **interactive programming**.
    - e.g., inside a notebook.
  - there are significant performance bottlenecks in iterative algorithms that need to **reuse** intermediate results.
    - e.g., saving intermediate results to stable storage (HDFS) is **very costly**.
- That is, MapReduce does not compose so well for large applications.
- For this reason, dozens of high level frameworks and specialized systems were developed.
  - e.g., Pregel, Dremel, Fl, Drill, GraphLab, Storm, Impala, etc.

# Spark



- Like Hadoop MapReduce, **Spark** is a framework for performing distributed computations.
- Unlike various earlier specialized systems, the goal of Spark is to **generalize** MapReduce.
- Two small additions are enough to achieve that goal:
  - **fast data sharing**
  - general **direct acyclic graphs** (DAGs).
- Designed for data reuse and interactive programming.

# Programmability

```
1 public class WordCount {
2     public static class TokenizerMapper
3         extends Mapper<Object, Text, Text, IntWritable> {
4
5         private final static IntWritable one = new IntWritable(1);
6         private Text word = new Text();
7
8         public void map(Object key, Text value, Context context
9                         throws IOException, InterruptedException {
10             StringTokenizer itr = new StringTokenizer(value.toString());
11             while (itr.hasMoreTokens()) {
12                 word.set(itr.nextToken());
13                 context.write(word, one);
14             }
15         }
16     }
17
18     public static class IntSumReducer
19         extends Reducer<Text,IntWritable,Text,IntWritable> {
20         private IntWritable result = new IntWritable();
21
22         public void reduce(Text key, Iterable<IntWritable> values,
23                            Context context
24                            throws IOException, InterruptedException {
25             int sum = 0;
26             for (IntWritable val : values) {
27                 sum += val.get();
28             }
29             result.set(sum);
30             context.write(key, result);
31         }
32     }
33
34     public static void main(String[] args) throws Exception {
35         Configuration conf = new Configuration();
36         String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
37         if (otherArgs.length < 2) {
38             System.err.println("Usage: wordcount <in> [<out>]");
39             System.exit(2);
40         }
41         Job job = new Job(conf, "word count");
42         job.setJarByClass(WordCount.class);
43         job.setMapperClass(TokenizerMapper.class);
44         job.setCombinerClass(IntSumReducer.class);
45         job.setReducerClass(IntSumReducer.class);
46         job.setOutputKeyClass(Text.class);
47         job.setOutputValueClass(IntWritable.class);
48         for (int i = 0; i < otherArgs.length - 1; ++i) {
49             FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
50         }
51         FileOutputFormat.setOutputPath(job,
52             new Path(otherArgs[otherArgs.length - 1]));
53         System.exit(job.waitForCompletion(true) ? 0 : 1);
54     }
55 }
```

```
1 val f = sc.textFile(inputPath)
2 val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3 w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

WordCount in 50+ lines of Java MR

See also [Spark examples](#)

# Performance

Time for sorting **100TB** of data:

2013 Record:      2100 machines  
Hadoop



72 minutes



2014 Record:      207 machines  
Spark

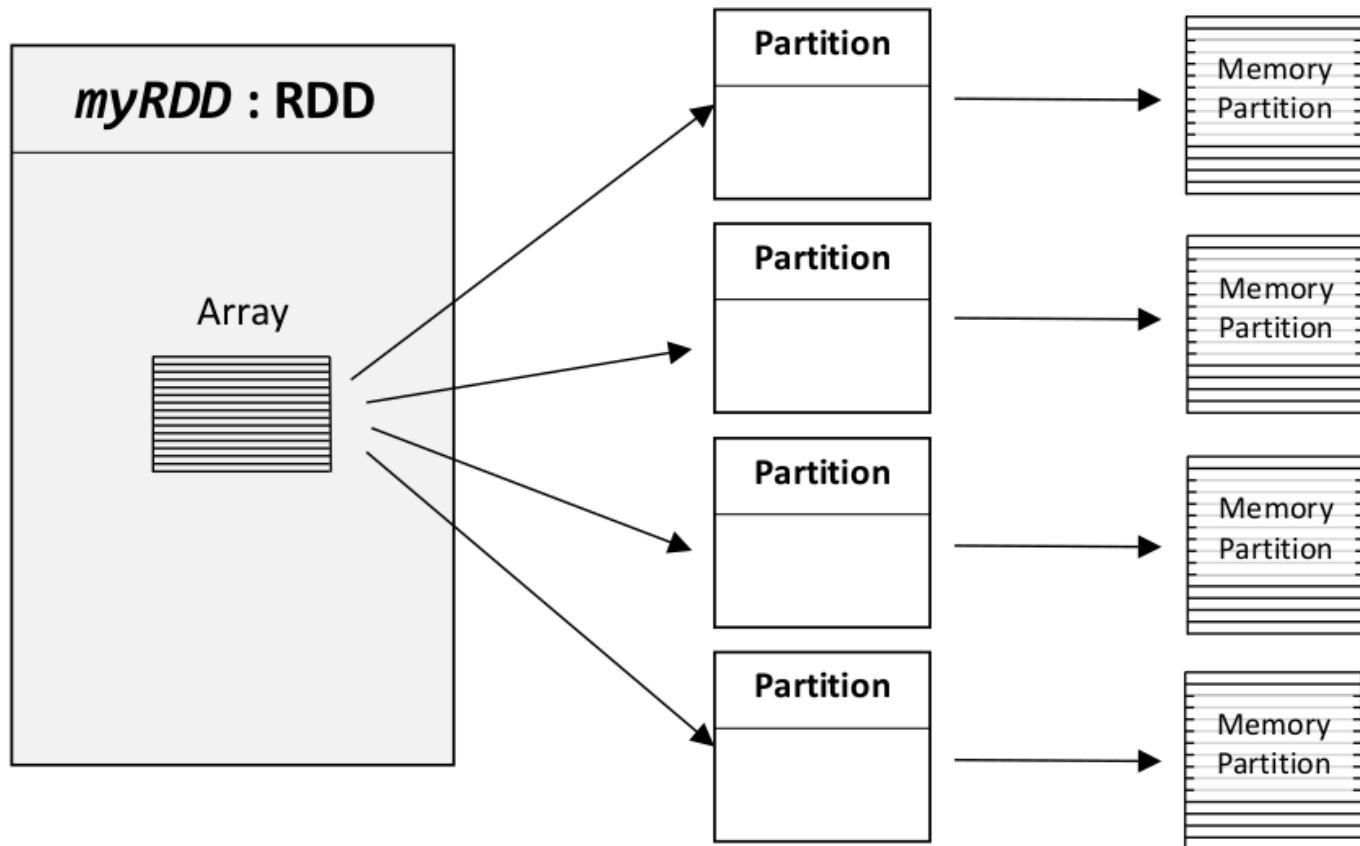


23 minutes



# RDD

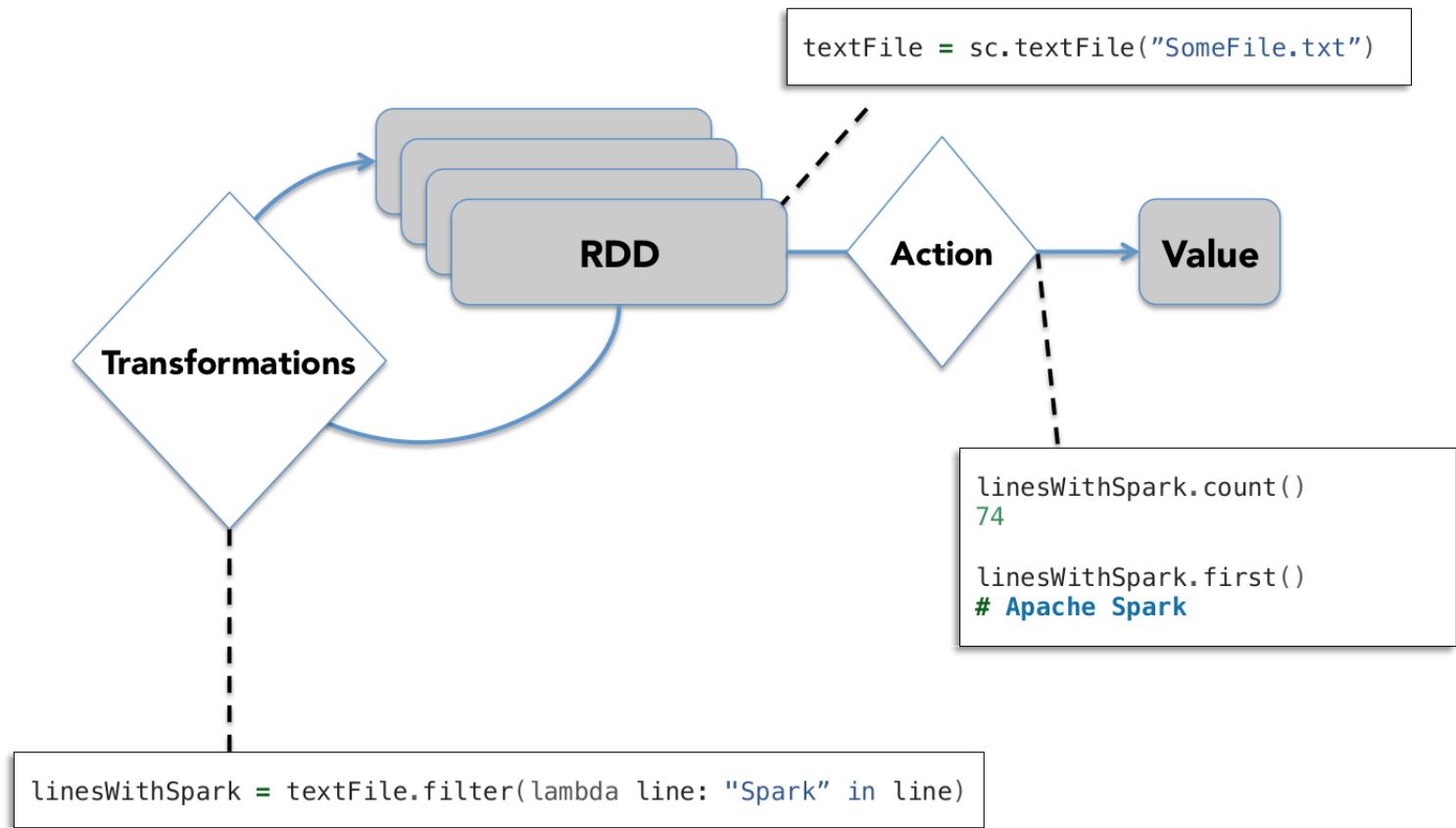
- Programs in Spark are written in terms of a Resilient Distributed Dataset (RDD) abstraction and operations on them.
- An RDD is a fault-tolerant read-only, partitioned collection of records.
  - Resilient: built for fault-tolerance (it can be recreated).
  - Distributed: content is divided into atomic partitions, usually stored in memory and across multiple nodes.
  - Dataset: collection of partitioned data with primitive values or values of values.
- RDDs can only be created through deterministic operations on either:
  - data in stable storage, or
  - other RDDs.



## Operations on RDDs

- **Transformations:**  $f(\text{RDD}) \rightarrow \text{RDD}'$ 
  - Coarse-grained operations only (à la pandas/numpy).
    - It is not possible to write to a single specific location in an RDD.
  - Lazy evaluation (not computed immediately).
  - e.g., `map` or `filter`.
- **Actions:**  $f(\text{RDD}) \rightarrow v$ 
  - Triggers computation.
  - e.g., `count`.
- The interface also offers explicit **persistence** mechanisms to indicate that an RDD will be reused in future operations.
  - This allows for significant internal optimizations.

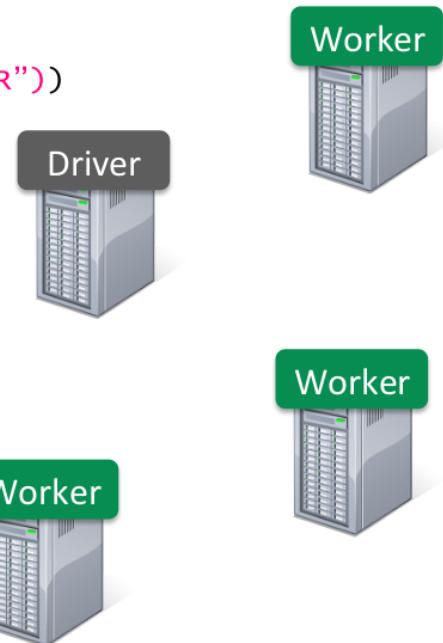
# Workflow



## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()  
  
messages.filter(lambda s: "mysql" in s).count()
```

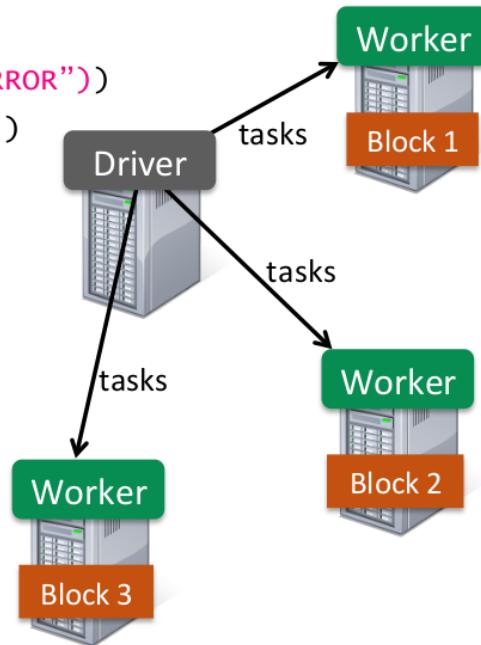


## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()
```

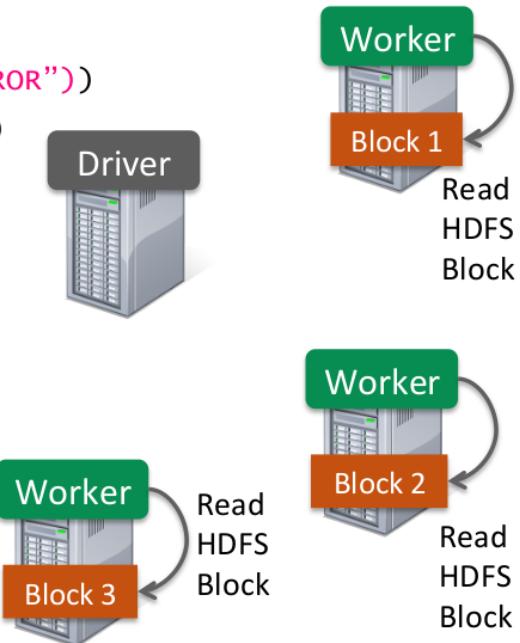
```
messages.filter(lambda s: "mysql" in s).count()
```



## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

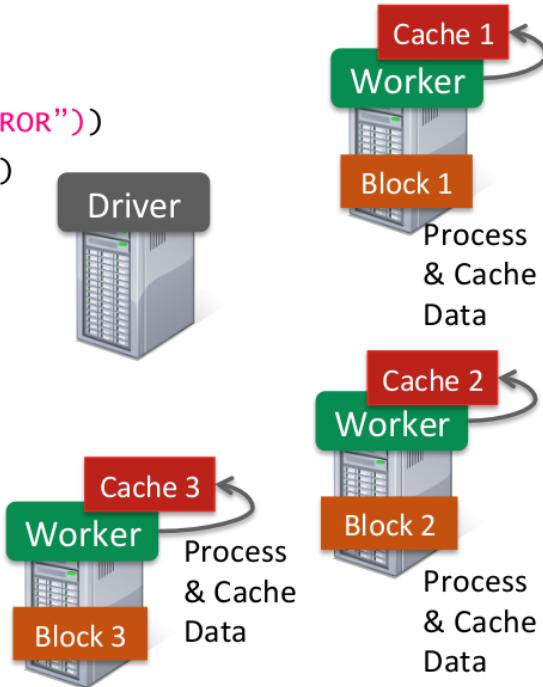
```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()  
  
messages.filter(lambda s: "mysql" in s).count()
```



## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()  
  
messages.filter(lambda s: "mysql" in s).count()
```

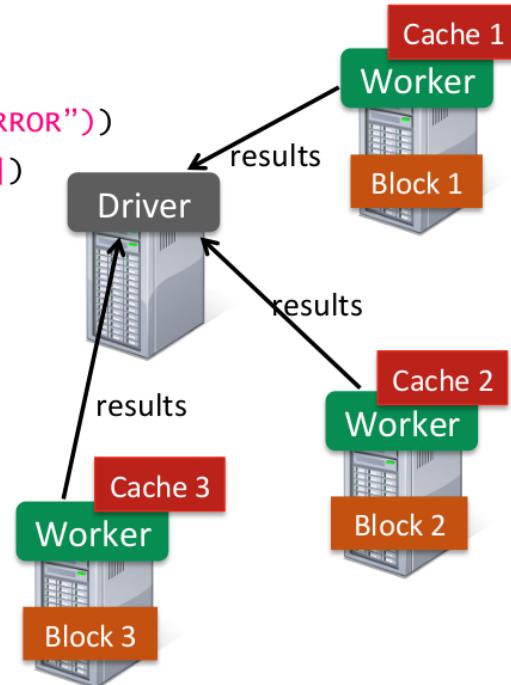


## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()
```

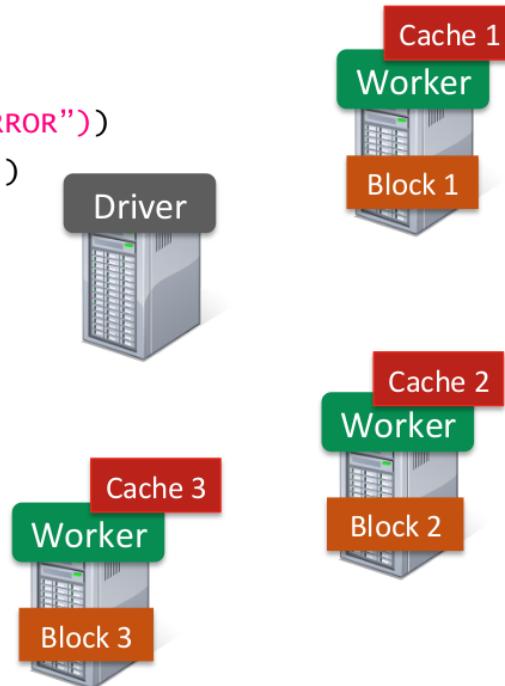


## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()
```

```
messages.filter(lambda s: "mysql" in s).count()  
messages.filter(lambda s: "php" in s).count()
```

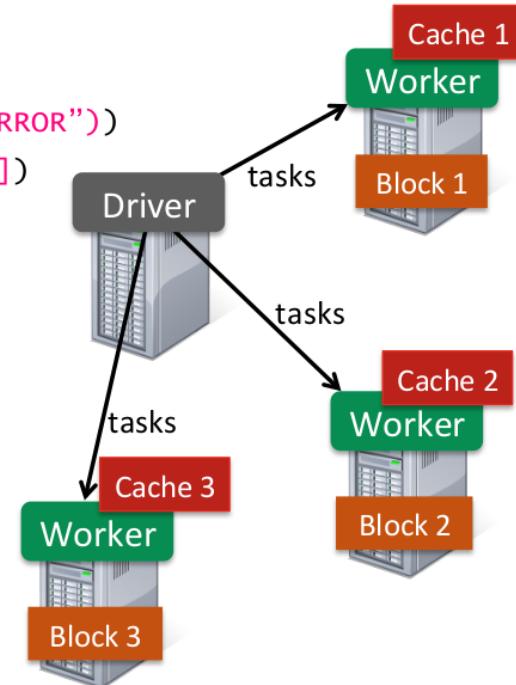


## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()
```

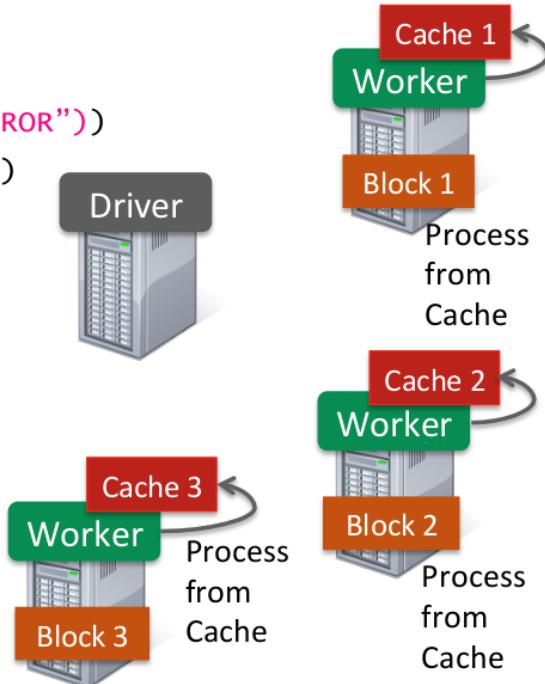
```
messages.filter(lambda s: "mysql" in s).count()  
messages.filter(lambda s: "php" in s).count()
```



## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

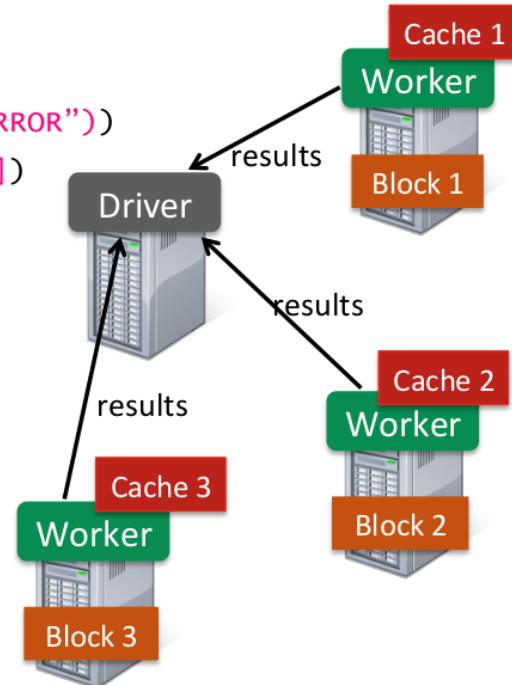
```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()  
  
messages.filter(lambda s: "mysql" in s).count()  
messages.filter(lambda s: "php" in s).count()
```



## Example: Log mining

Goal: Load error messages in memory, then interactively search for various patterns.

```
lines = spark.textFile("hdfs://...")  
errors = lines.filter(lambda s: s.startswith("ERROR"))  
messages = errors.map(lambda s: s.split("\t")[2])  
messages.cache()  
  
messages.filter(lambda s: "mysql" in s).count()  
messages.filter(lambda s: "php" in s).count()
```



# Rich, high-level API

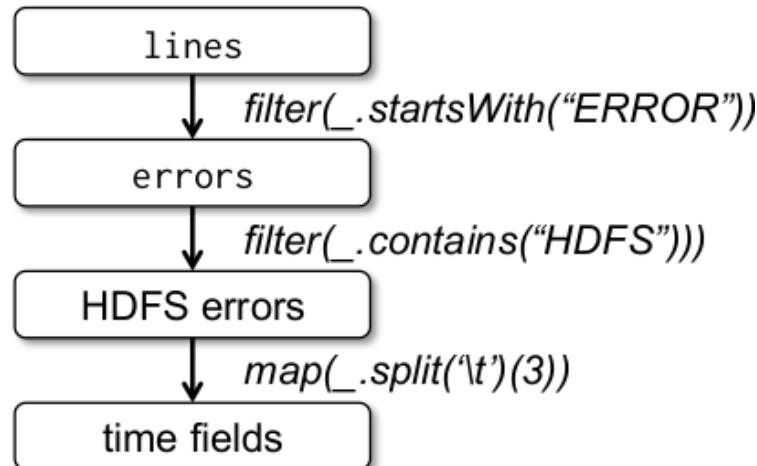
map  
filter  
sort  
groupBy  
union  
join  
...

reduce  
count  
fold  
reduceByKey  
groupByKey  
cogroup  
zip  
...

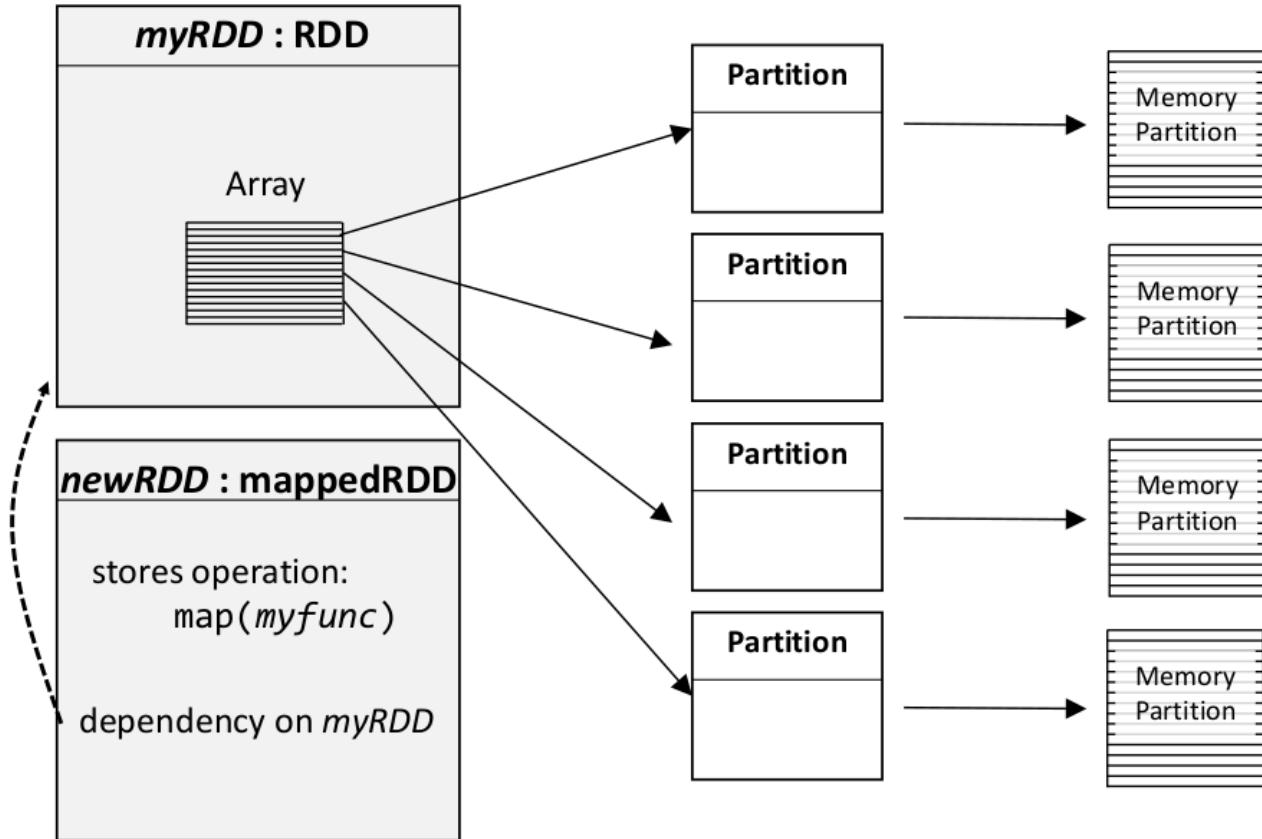
sample  
take  
first  
partitionBy  
mapWith  
pipe  
save  
...

# Lineage

- RDDs need not be materialized at all times.
- Instead, an RDD internally stores **how it was derived** from other datasets (its **lineage**) to compute its partitions from data in stable storage.
  - This derivation is expressed as coarse-grained transformations.
- Therefore, a program cannot reference an RDD that it cannot reconstruct after a failure.



```
newRDD = myRDD.map(myfunc)
```

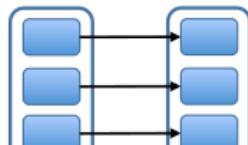


# Representing RDDs

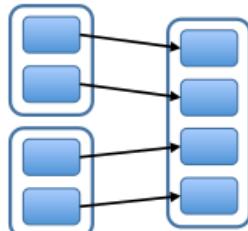
- RDDs are built around a **graph-based** representation (a DAG).
- RDDs share a common interface:
  - Lineage information:
    - Set of **partitions**.
    - List of **dependencies** on parents RDDs.
    - Function to **compute** a partition (as an iterator) given its parents.
  - Optimized execution (optional):
    - **Preferred locations** for each partition.
    - Partitioner (hash, range)

# Dependencies

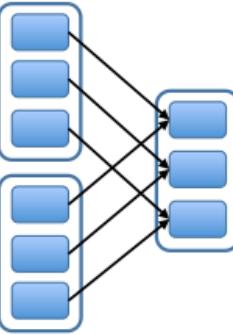
Narrow Dependencies:



map, filter

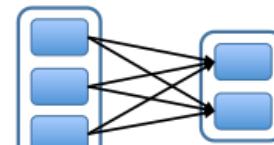


union

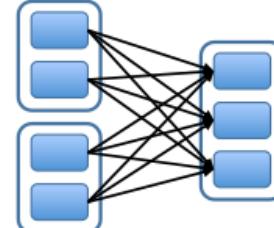


join with inputs  
co-partitioned

Wide Dependencies:



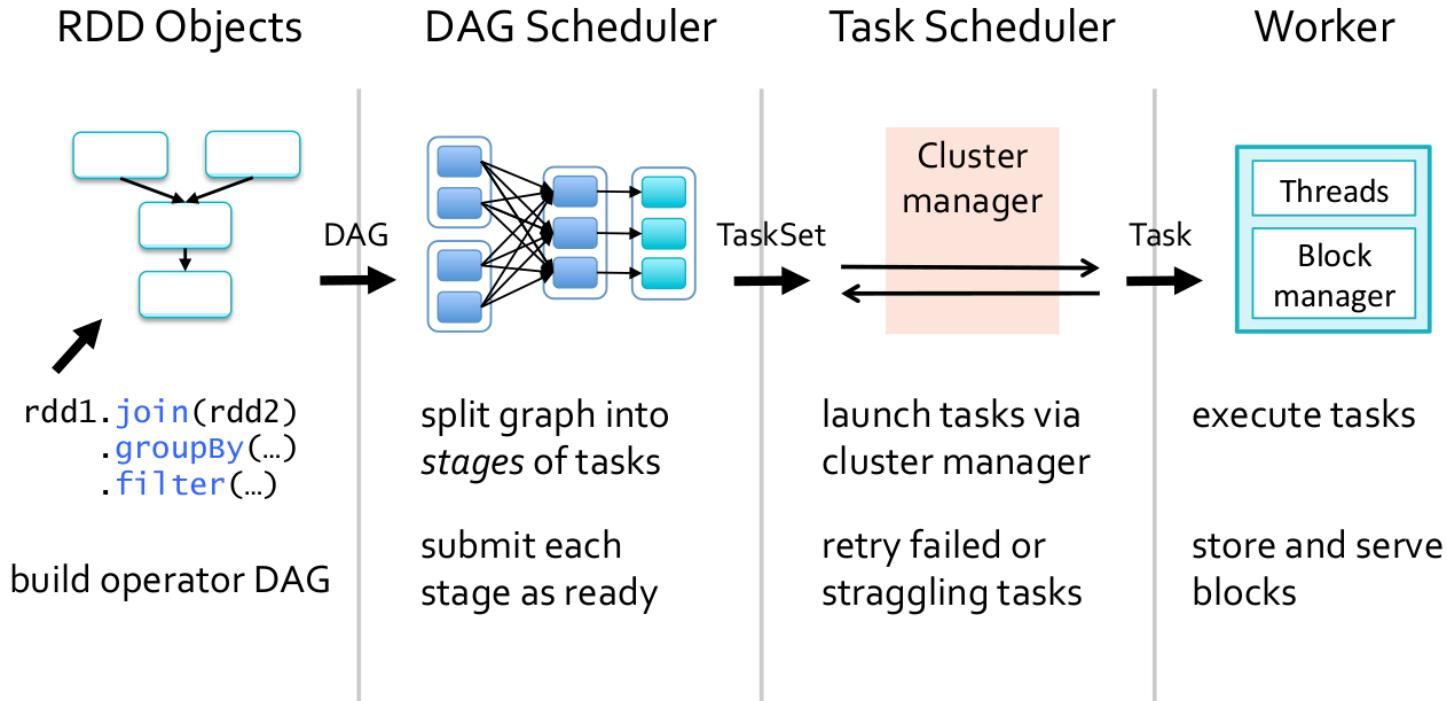
groupByKey



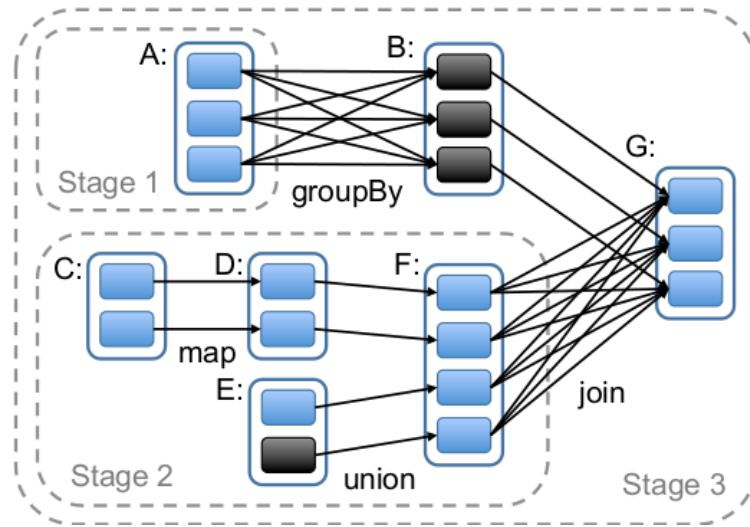
join with inputs not  
co-partitioned

- **Narrow dependencies:** each partition of the parent RDD is used by at most one partition of the child RDD.
  - Allow for pipelined execution on one node.
  - Recovery after failure is more efficient with a narrow dependency, as only the lost parents partitions need to be recomputed.
- **Wide dependencies:** multiple child partitions may depend on a parent partition.
  - A child partition requires data from all its parents to be recomputed.

# Execution process



# Job scheduler



- Whenever an **action** is called, the scheduler examines that RDD's lineage graph to build a **DAG of stages** to execute.
- Each **stage** contains as many pipeline transformations with narrow dependencies as possible.
- The boundaries of the stages are
  - the shuffle operations required for wide dependencies, or
  - already computed partitions that can short-circuit the computation of a parent RDD.

- The scheduler launches **tasks** to a lower-level scheduler to compute missing partitions from each stage until it has computed the target RDD.
  - One task per partition.
- Tasks are assigned to machines based on **data locality**.

# Fault tolerance

- If a task fails, it is rescheduled on another node, as long as its stage's parents are still available.
- If some stages have become unavailable, all corresponding tasks are resubmit to compute the missing partitions in parallel.

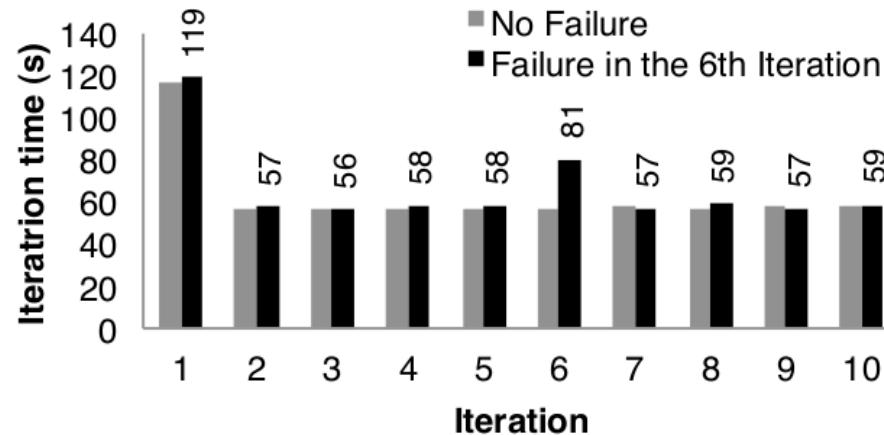


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

# Dataflow programming

- Spark builds upon the **dataflow programming** paradigm.
- Dataflow programming models a program as a **directed graph** of the data flowing between operations.
- An operation runs as soon as all of its inputs become valid.
- Dataflow languages are inherently parallel and work well in large, decentralized systems.
- Modern examples:
  - Scala
  - Spark
  - **Tensorflow**

# Summary

- High-level abstractions enable **cloud programming** over clusters.
  - Without having to handle parallelization, data distribution, load balancing, fault tolerance, ...
- **MapReduce** is a parallel programming model based on map and reduce operations.
  - Best suited for embarrassingly parallel and linear tasks.
  - Its simplicity is a disadvantage for complex iterative programs for interactive exploration.
- **Spark** generalizes MapReduce by making use of:
  - fast data sharing (data resides in memory)
  - general direct acyclic graphs of operations.



# References

- Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.
- Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012.
- Xin, Reynold. "[Stanford CS347 Guest Lecture: Apache Spark](#)". 2015.

# Large-scale Data Systems

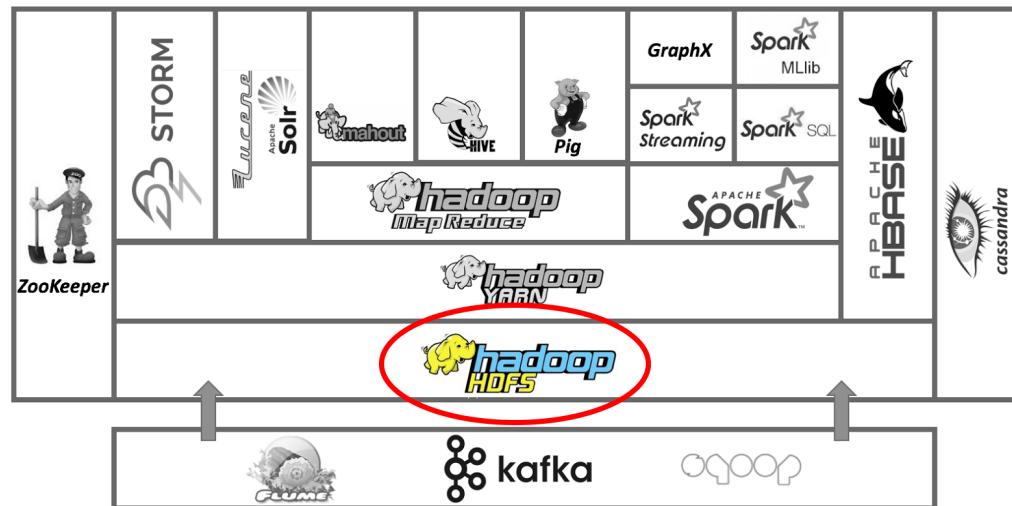
Lecture 8: Distributed file systems

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)

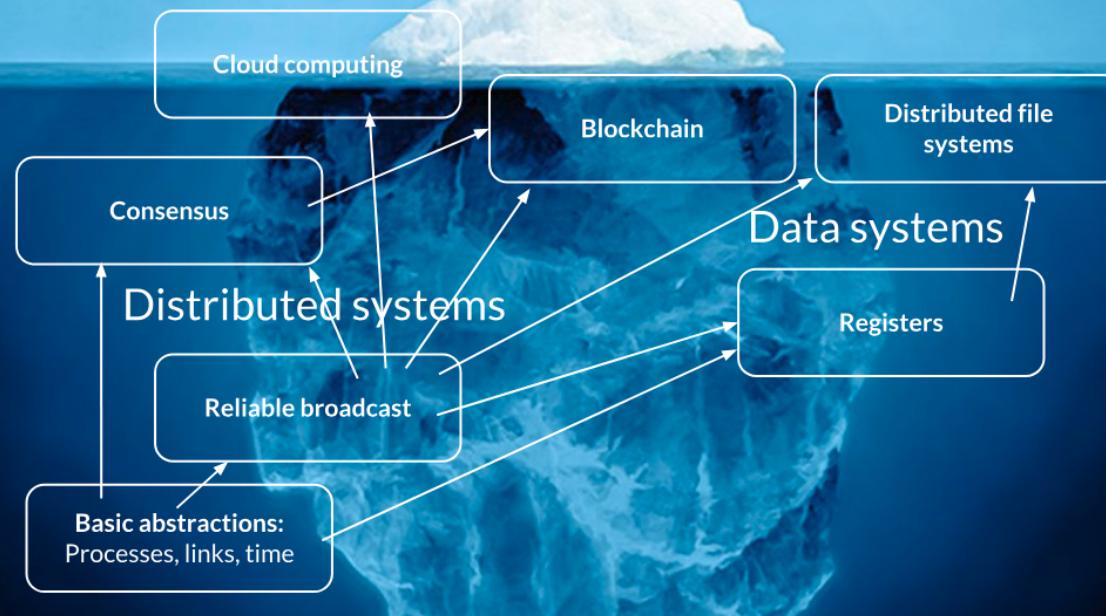


# Today

- Google File System (GFS)
  - Design considerations
  - Data replication
  - Reading and writing
  - Recovery from failure
- Hadoop Distributed File System (HDFS)



# Data science Machine Learning Visualization



Computer networks

# Google File System

# File systems

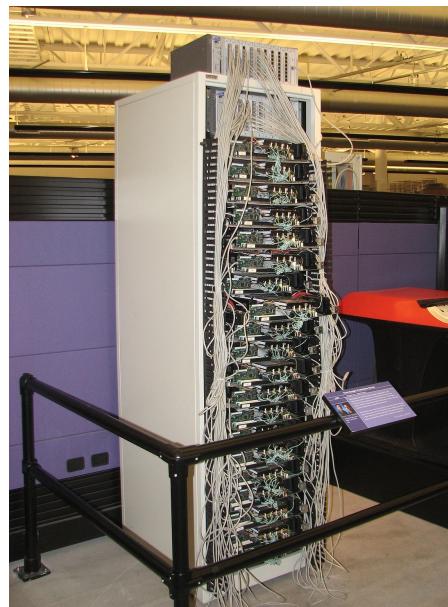
- File systems determine **how** data is stored and retrieved.
- **Distributed file systems** (DFS) manage the storage across a network of machines.
  - Goal: single-system **illusion** for users.
  - Added complexity due to the network.
- GFS and HDFS are examples of distributed file systems.
  - They represent **one** way (not the way) to design a distributed file system.

[Q] Which file systems do you know?

## How would you design a DFS?

We want **single system illusion** for data storage.

- Data is too large to be stored in a single system.
- Hardware **will** fail.



*Google first servers*

# History

GFS was developed at Google around 2003, jointly with MapReduce.

- Provide **efficient** and **reliable** access to data.
- Use large clusters of **commodity hardware**.
- Proprietary system, but detailed description.

## The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung  
Google\*

### ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

### 1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the

# Design aims

- Maintain data and system **availability**.
- Handle **failures** gracefully and transparently.
- **Low synchronization** overhead between entities.
- Exploit **parallelism** of numerous entities.
- Ensure **high sustained throughput** for individual reads/writes.

# Assumptions

- Hardware failures are common.
  - We want to use cheap commodity hardware.
- Files are large (multi-GB files are the norm) and their number is (relatively) limited (millions).
- Reads:
  - large streaming reads ( $\geq$  1MB in size), or
  - small random reads
- Writes:
  - Large sequential writes that append to files.
  - Concurrent appends by multiple clients.
  - Once written, files are seldom modified ( $\neq$  append) again.
    - Random modification in files is possible, but not efficient in GFS.
- High sustained bandwidth, but high latency.

## **Which of those fit the assumptions?**

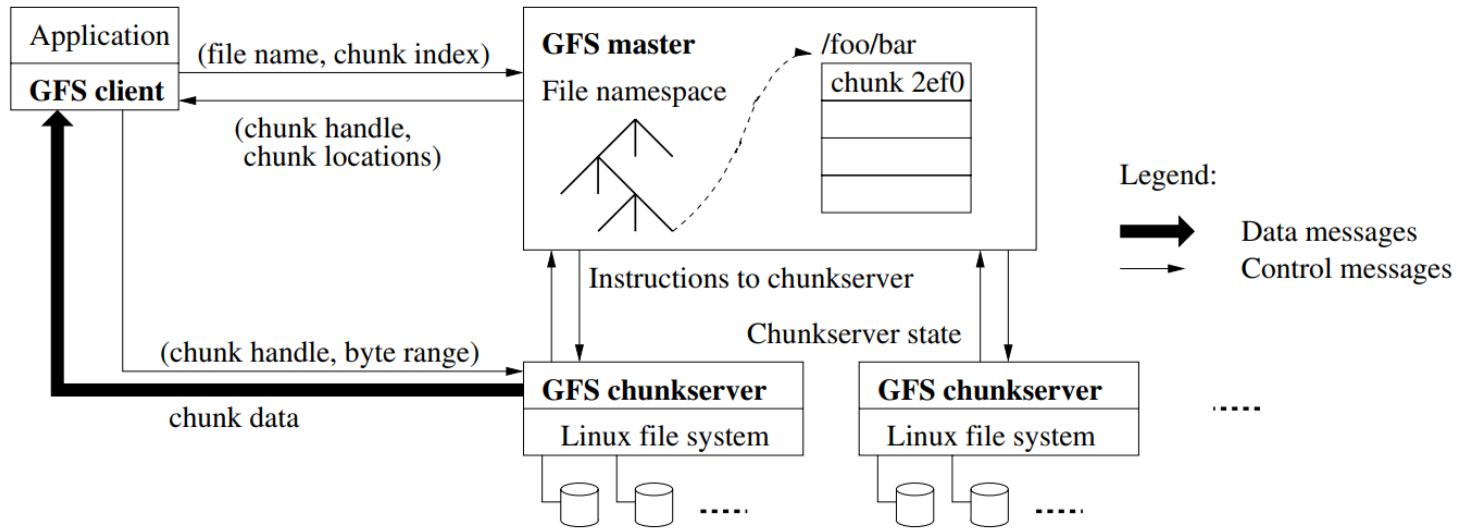
- Global company dealing with the data of its 100M employees.
  - Salary, bonuses, age, performance, etc.
- A search engine's query log.
- A hospital's medical imaging data generated from an MRI scan.
- Data sent by the Hubble telescope.
- A search engine's index (used to serve search results to users).

## **Disclaimer**

GFS (and HDFS) are not a good fit for:

- Low latency data access (in the ms range).
  - Solution: distributed databases, such as HBase.
- Many small files.
- Constantly changing data.

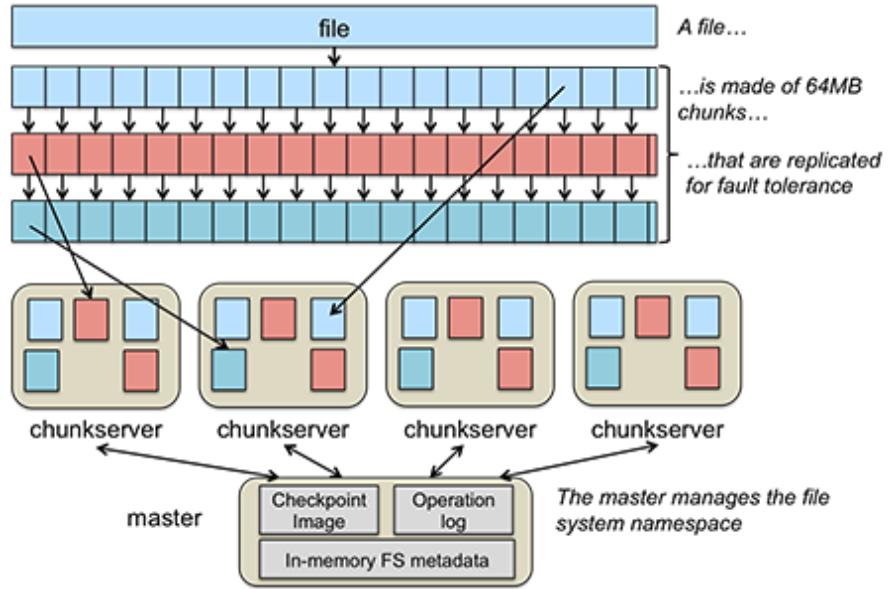
# Architecture



- A **single master** node.
- Many **chunkservers** (100s - 1000s) storing the data.
  - Physically spread in different racks.
- Many clients.

[Q] Why spreading across racks?

# Files



- A single **file** may contain several **objects** (e.g., images, web pages, etc).
- Files are divided into fixed-size **chunks**.
  - Each chunk is identified by a globally unique 64 bit **chunk handle**.
- Chunkservers store chunks on local disks as plain Linux files.
  - Read or write data specified by a pair (chunk handle, byte range).
  - By default **three replicas** of a chunk stored across chunkservers.

# Master

- The master node stores and maintains **all file system metadata**:
  - Three main types of metadata:
    - the file and chunk namespaces,
    - the mapping from files to chunks,
    - the locations of each chunk's replicas.
  - All metadata is kept in master's **memory** (fast random access).
    - Sets limits on the entire system's capacity.
- It controls and coordinates **system-wide activities**:
  - Chunk lease management
  - Garbage collection of orphaned chunks
  - Chunk migration between chunkservers
- **Heartbeat** messages between master and chunkservers.
  - To detect failures
  - To send instructions and collect state information
- An **operation log** persists master's state to permanent storage.
  - In case master crashes, its state can be recovered (more later).

## One node to rule them all

- Having a **single master** node vastly simplifies the system design.
- Enable master to make sophisticated chunk placement and replication decisions, using **global knowledge**.
- Its involvement in reads and writes should be minimized so to avoid that it becomes a bottleneck.
  - Clients never read and write file data through master.
  - Instead, clients ask the master which chunkservers it should contact.

[Q] As the cluster grows, can the master become a bottleneck?

# Chunks

- Default size = 64MB.
  - This a **key design parameter** in GFS!
- Advantages of large (but not too large) chunk size:
  - **Reduced need** for client/master interaction.
    - 1 request per chunk suits the target workloads.
    - Client can cache **all the locations** for a multi-TB working set.
  - **Reduced size** of metadata on master (kept in memory).
- Disadvantage:
  - A chunkserver can become a **hotspot** for popular files.

[Q] How to fix the hotspot problem?

[Q] What if a file is larger than the chunk size?

# Caching

Design decisions:

- Clients do **not** cache file data.
  - They do cache metadata.
- Chunckservers do **not** cache file data.
  - Responsibility of the underlying file system (e.g., Linux's buffer cache).
- Client caches offer **little benefit** because most applications
  - stream through huge files
    - disk seek time negligible compared to transfer time.
  - have working sets too large to be cached.
- Not having a caching system **simplifies the overall system** by eliminating cache coherence issues.

# Interface

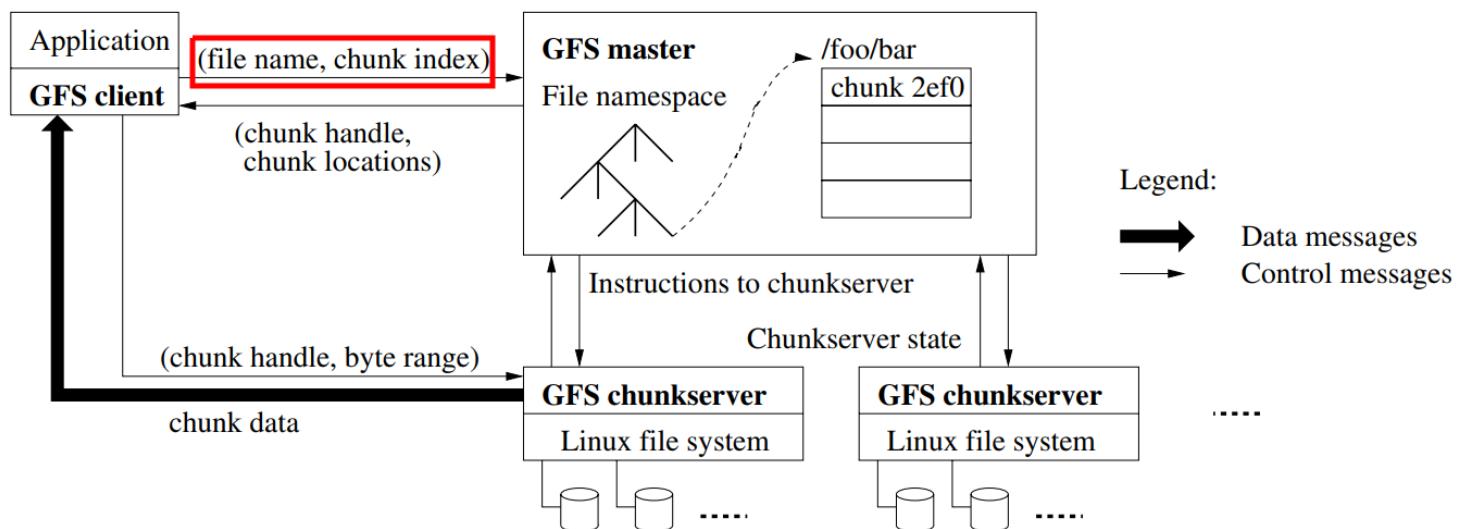
No file system interface at the operating-system level (e.g., under the VFS layer).

- User-level API is provided instead.
- Does not support all the features of POSIX file system access.
  - But looks similar (i.e., open, close, read, write, ...)

Two special operations are supported:

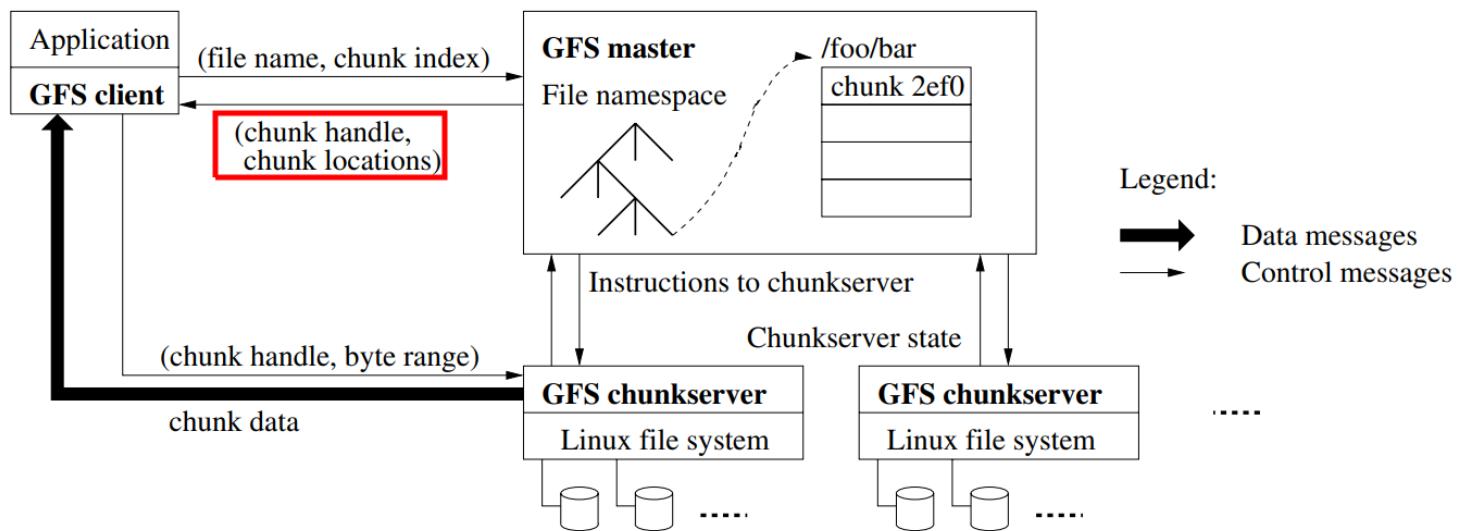
- **Snapshot**: efficient way of creating a copy of the current instance of a file or directory tree.
- **Append**: append data to a file as an **atomic operation**, without having to lock the file.
  - Multiple processes can append to the same file concurrently without overwriting one another's data.

## Reads (1)



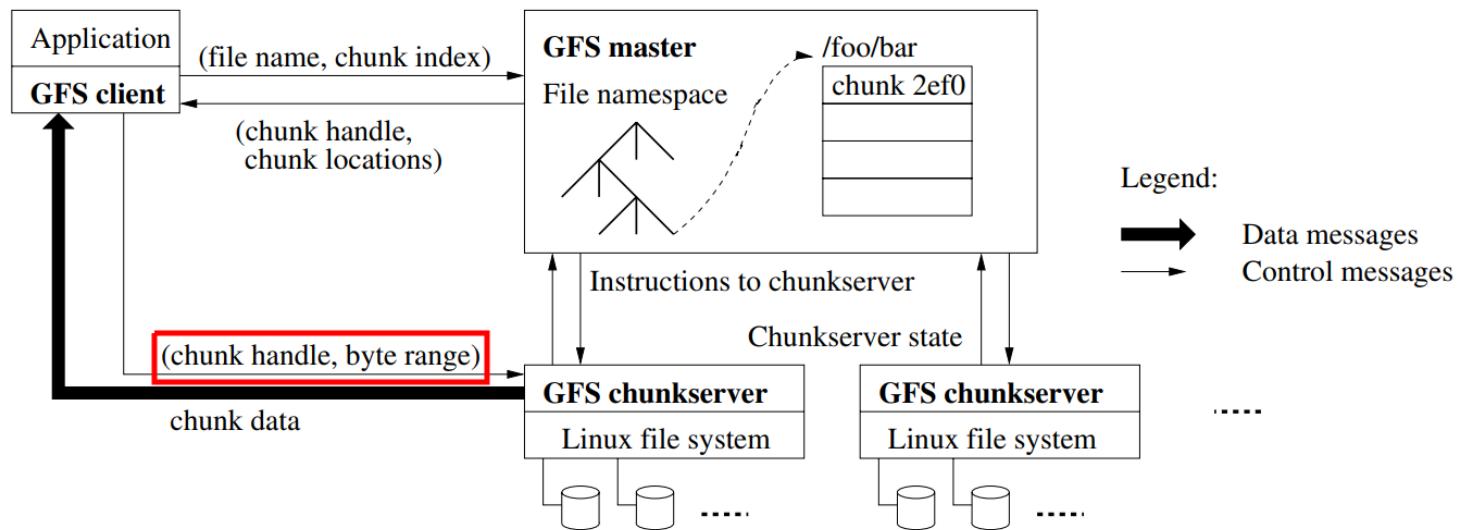
- 1) The GFS client translates filename and byte offset specified by the application into a **chunk index** within the file. A request is sent to master.

## Reads (2)



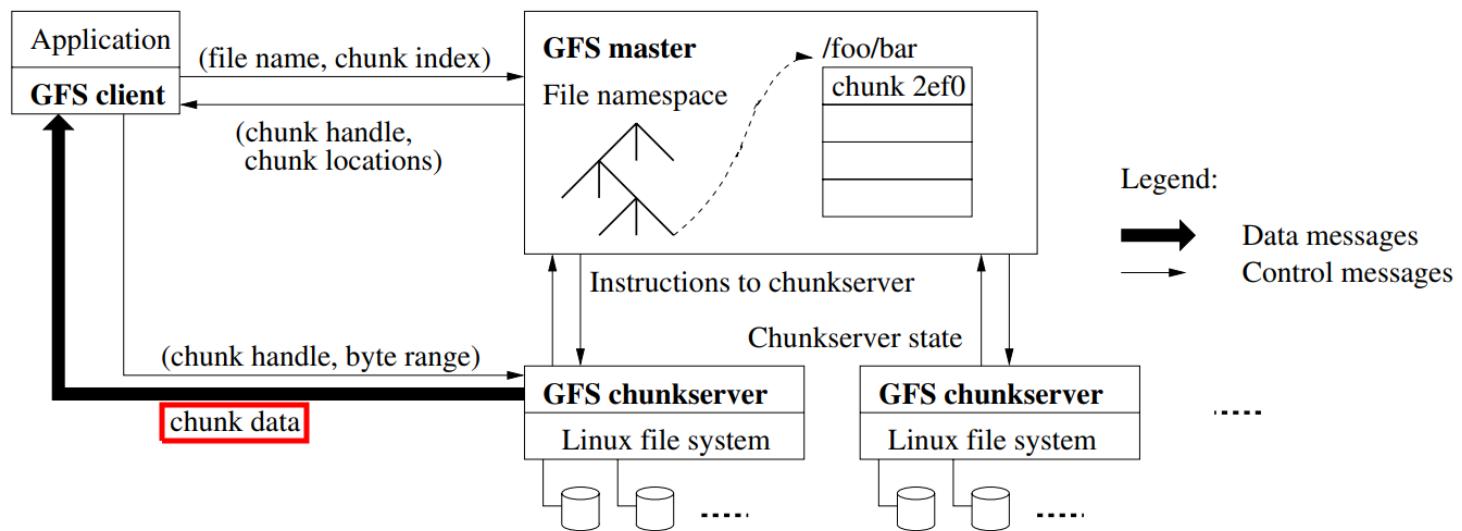
2) Master replies with chunk handle and locations of the replicas.

## Reads (3+4)



- 3) The client caches this information using the file name and chunk index as the key.
  - Further reads of the same chunk requires no more client-master interaction, until the cached information expires.
- 4) The client sends a request to one of the replicas, typically the closest.

## Reads (5)

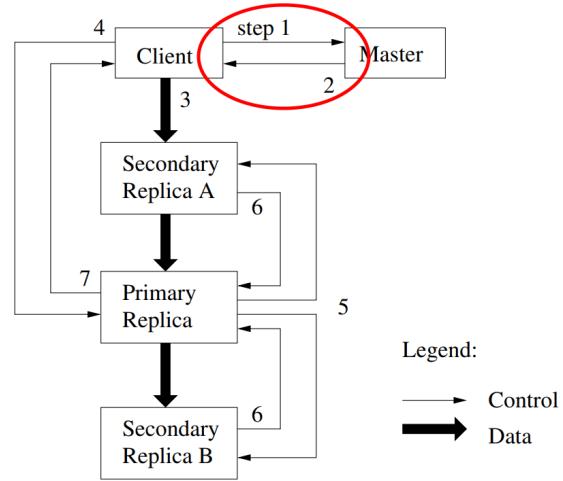


- 5) The contacted chunkserver replies with the data.

## Leases

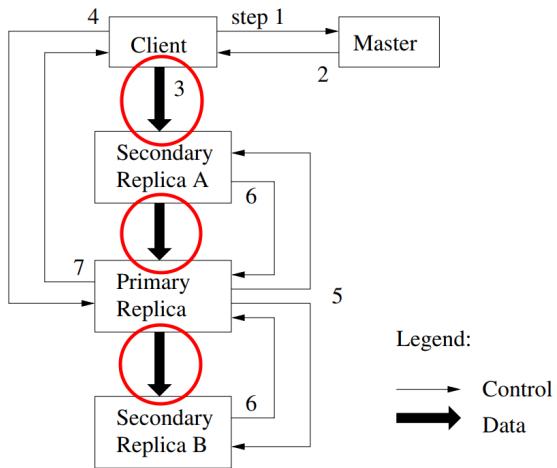
- A **mutation** is an operation that changes the content or metadata of a chunk (e.g., `write` and `append`).
- Each mutation is performed at all the chunk's replicas.
- **Leases** are used to maintain a consistent mutation order across replicas.
  - Master grants a chunk lease to one of the replicas, called the **primary**.
  - Leases are renewed using the periodic heartbeat messages between master and chunkservers.
- The primary picks a **serial order** for all mutations to the chunk.
  - All replicas follow this order when applying mutations.
- Leases and serial order at the primary define a **global ordering** of the operations on a chunk.

## Writes (1+2)



- 1) The GFS client asks master for the primary and the secondary replicas for each chunk.
- 2) Master replies with the locations of the primary and secondary replicas. This information is cached.

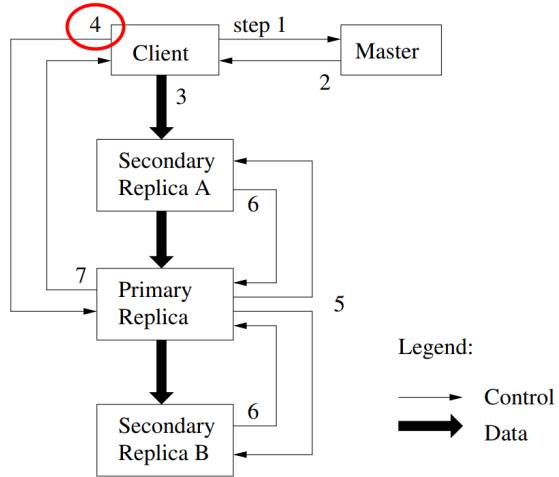
## Writes (3)



3) The client pushes the data to all replicas.

- Each chunkserver stores the data in an internal buffer.
- Each chunkserver sends back an acknowledgement to the client once the data is received.
- Data flow is decoupled control flow.

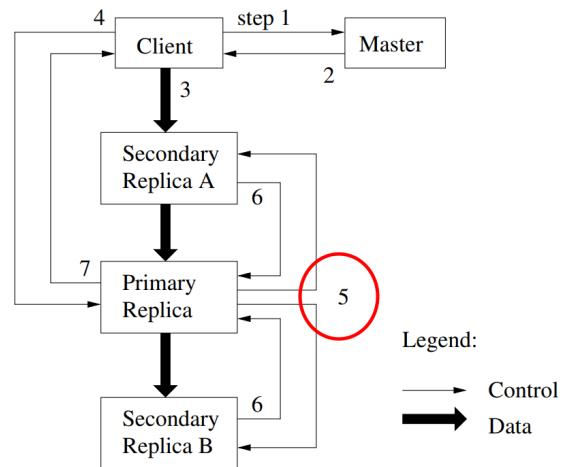
## Writes (4)



4) Once all replicas have acknowledged, a **write request** is sent to the primary.

- This request identifies the data pushed earlier.
- The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients.
  - This provides **ordering** and **consistency**.
- The primary applies the mutations, in the chosen order, to its local state.

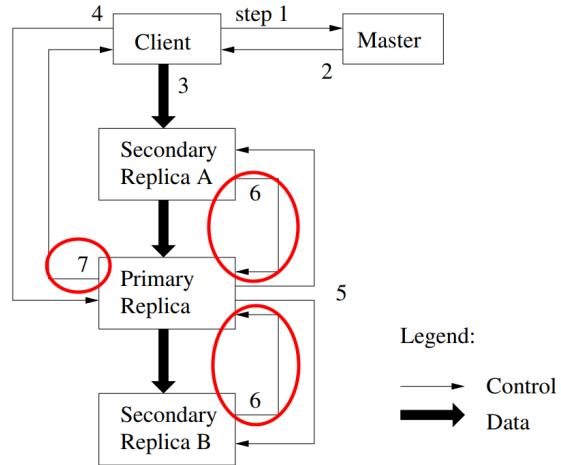
## Writes (5)



5) The primary forwards the write request to all secondary replicas.

- Mutations are applied locally in the serial order decided by the primary.

## Writes (6+7)



- 6) The secondaries all reply to the primary upon completion of the operation.
- 7) The primary replies to the client.

- Errors may be reported to the client.
  - Upon errors, the client request is considered to have failed.
  - The modified region is left in an **inconsistent state**.
  - The client handles errors by retrying the failed mutation.

## Appends

- Google uses large files as **queues** between multiple **producers** and **consumers**.
- Same control flow as for writes, except that:
  - Client pushes data to replicas of **last chunk** of file.
  - Client send an append request to the primary.
  - The request fits in current last chunk:
    - Primary appends data to own replica.
    - Primary tells secondaries to do same at same byte offset in theirs.
    - Primary replies with success to client, specifying the offset the data was written.
  - When the data does not fit in last chunk:
    - Primary fills current chunk with padding.
    - Primary tells secondaries to do the same.
    - Primary replies to client to **retry on next chunk**.
- If a record append fails at any replica, the client has to retry the operation.
  - Replicas of same chunk may not be bytewise identical!

# Consistency model

- Changes to metadata are always **atomic**.
  - Guaranteed by having a single master server.
- Mutations are **ordered** as chosen by a primary node.
  - All replicas will be **consistent** if they all successfully perform mutations in the same order.
  - Multiple writes from the same client may be interleaved or overwritten by concurrent operations from other clients.
    - i.e., a file region is **defined** only if client see mutations in entirety, it is **undefined** otherwise. However, the file region remains consistent.
- Record append completes **at least once**, at offset of GFS's choosing.
  - There might be duplicate entries.
- Failures can cause **inconsistency**.

	Write	Record Append
Serial success	<i>defined</i>	<i>defined</i> interspersed with <i>inconsistent</i>
Concurrent successes	<i>consistent</i> but <i>undefined</i>	
Failure		<i>inconsistent</i>

# Replica placement

- Policy is to maximize:
  - data **reliability** and **availability**,
  - network bandwidth utilization.
- Chunks are created initially empty.
  - Preferably create chunks at **under-utilized** chunkservers, spread across different racks.
  - Limit number of recent creations on each chunk server.
- Re-replication.
  - Started once the available replicas fall below a user-defined threshold.
  - Master instructs chunkserver to copy chunk data directly from existing valid replica.
  - Number of active clone operations/bandwidth is limited.
- Re-balancing
  - Changes in replica distribution for better load balancing.
  - New chunk servers are gradually filled.

# Garbage collection

How can a file be **deleted** from the cluster?

- Deletion is **logged** by master.
- The file is **renamed** to a hidden file and the deletion timestamp is kept.
- Periodic scan of the master's file system namespace.
  - Hidden files older than 3 days are deleted from master's memory.
  - I.e., there is no further connection between a file and its chunks.
- Periodic scan of the master's chunk namespace.
  - Orphaned chunks (not reachable from any file) are identified and their metadata is deleted.
- Heartbeat messages are used to synchronize deletion between master and chunkservers.

# Stale replica detection

Scenario: a chunkserver misses a mutation applied to a chunk (e.g., a chunk was appended).

- Master maintains a **chunk version number** to distinguish up-to-date and stale replicas.
- Before an operation on a chunk, master ensures that the version number advances.
  - Each time master grants new lease, the version is incremented and informed to all replicas.
- Stale replicas are removed in the regular garbage collection cycle.

# Operation log

- The **operation log** is a persistent historical record of critical changes on metadata.
- Critical to the **recovery** of the system, upon restart of master.
  - Master recovers its file system state by replaying the operation log.
  - Master periodically checkpoints its state to minimize startup time.
- Changes to metadata are only made visible to the clients **after** they have been written to the operation log.
- The operation log is **replicated** on multiple remote machines.
  - Before responding to a client operation, the log record must have been flushed locally and remotely.
- Serve as a **logical timeline** that defines the order of concurrent operations.

# Chunk locations

- Master does not keep a persistent record of chunk replica locations.
- Instead, it **polls** chunkservers about their chunks at startup.
- Master keeps up to date through **heartbeat** messages.
- A chunkserver has the **final word** over what chunks it stores.

[Q] What does this design decision simplify?

# What if master fails?

- ... and does not recover?
- This represents a **single point of failure** of system.
- Solution:
  - Maintain shadow **read-only** replicas of master.
  - Use these replicas in case master fails.
  - Eventually elect a new leader if master never recovers.

# What if a chunkserver fails?

- Master notices missing heartbeats.
- Master decrements count of replicas for all chunks on dead chunkserver.
- Master re-replicates chunks missing replicas.
  - Highest priority for chunks missing greatest number of replicas.

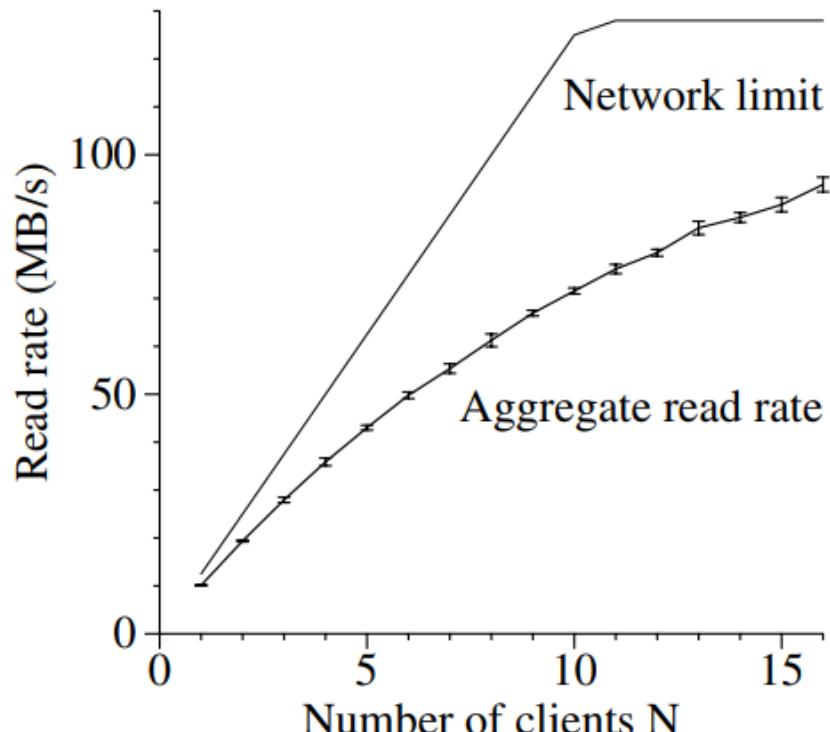
# Data corruption

- Data corruption or loss can occur at any time.
- Chunkservers use **checksums** to detect corruption of stored data.
  - Alternative: compare replicas across chunk servers.
- A chunk is broken into 64KB blocks, each has a 32bit checksum.
  - These are kept in memory and stored persistently.
- Read requests: the chunkserver **verifies the checksum** of the data blocks that overlap with the read range.
  - Corrupted data are not sent to the clients.

[Q] What if a read request fails because of corrupted data?

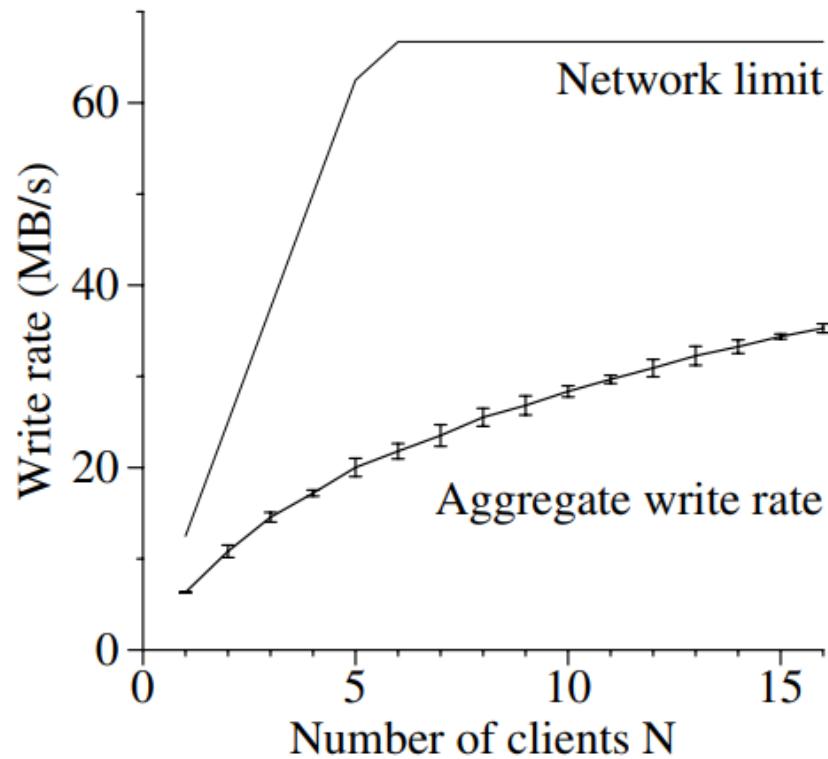
# Performance

## Reads



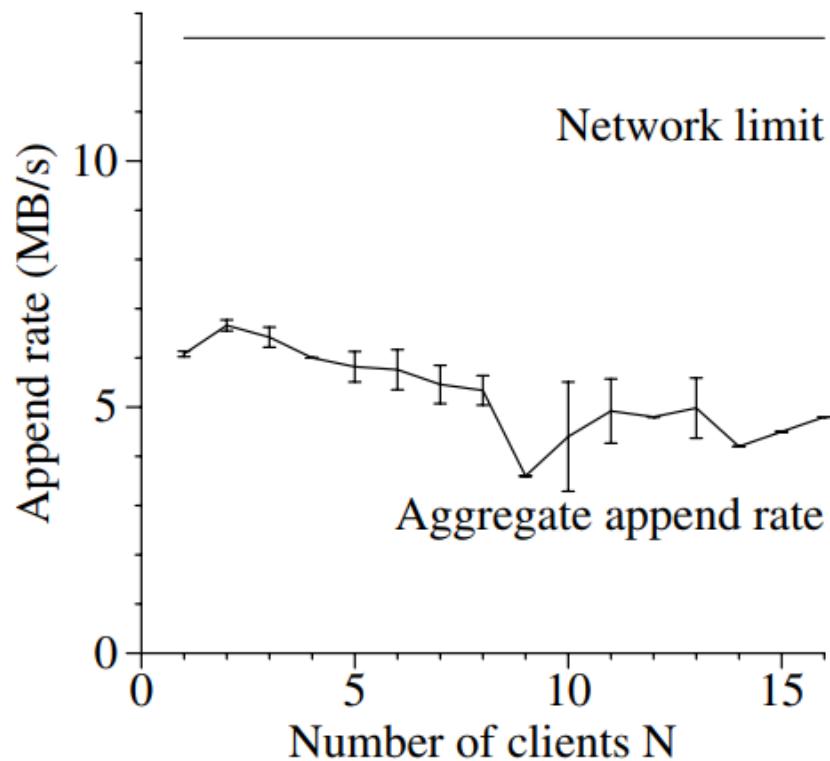
(a) Reads

## Writes



(b) Writes

## Append



(c) Record appends

# Summary

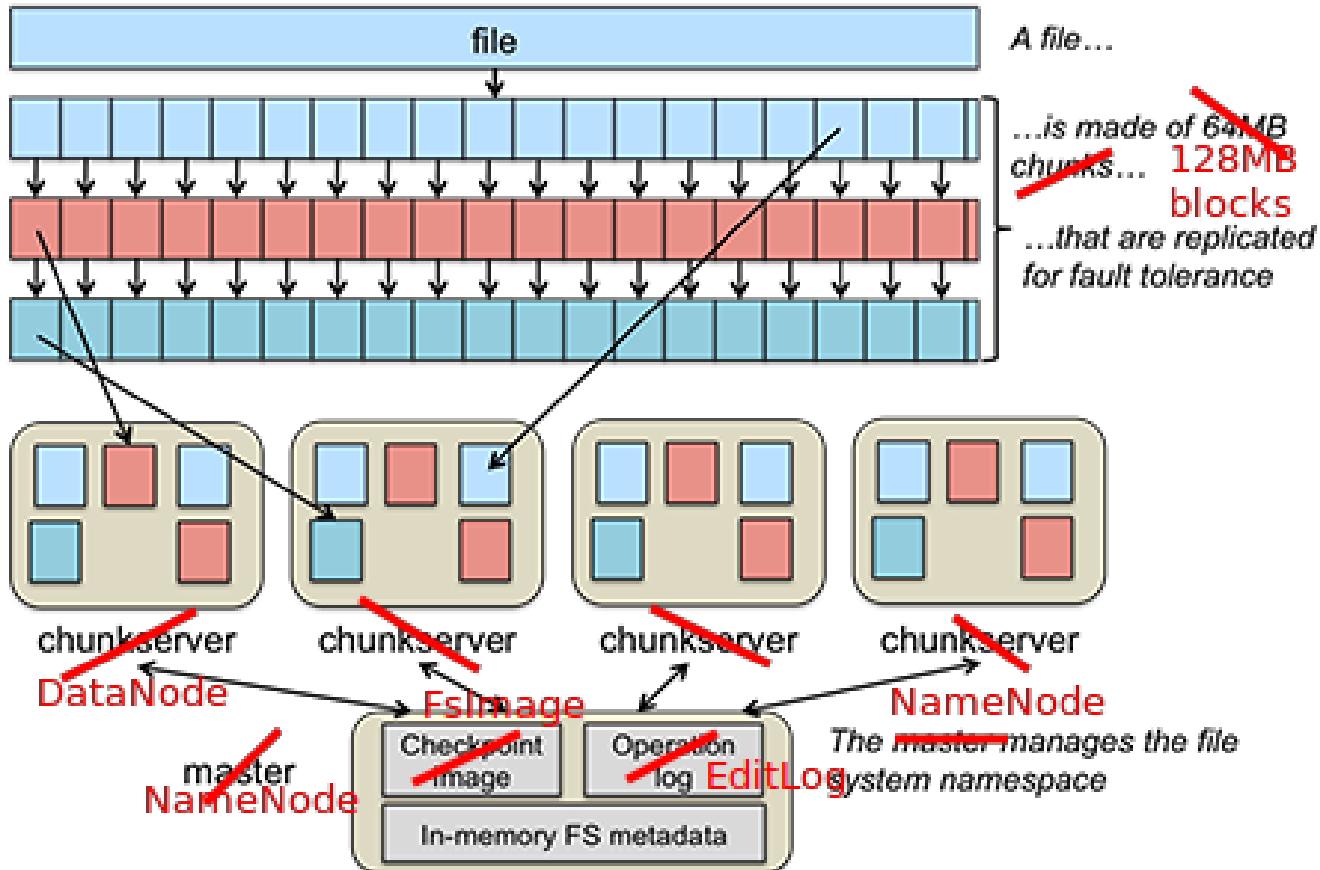
- GFS has been used actively by Google to support search service and other applications.
  - Availability and recoverability on cheap hardware.
  - High throughput by decoupling control and data.
  - Supports massive data sets and concurrent appends.
- Semantics not transparent to applications.
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics).
- Performance not good for all applications.
- Assumes read-once, write-once workload (no client caching!)
- Replaced in 2010 by Colossus
  - Eliminate master node as single point of failure
  - Targets latency problems due to more latency sensitive applications
  - Reduce block size to be between 1~8 MB
  - Few public details.

# HDFS

# HDFS

- Hadoop Distributed File System (HDFS) is an [open source](#) distributed file system.
- HDFS shares the same goals as GFS.
- HDFS's design is **strongly** inspired from GFS.
- HDFS uses a distributed cache.
- No leases (client decides where to write).
- Used by Facebook, Yahoo, IBM, etc.

## HDFS in one figure



## GFS

- Master
- Chunkserver
- Operation log
- Chunk
- Random file writes are possible
- Multiple writers, multiple readers model
- Default chunk size = 64MB



## HDFS

- NameNode
- DataNode
- Journal, edit log
- Block
- Only append is possible
- Single writer, multiple reader model
- Default block size = 128MB



# References

- Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS operating systems review. Vol. 37. No. 5. ACM, 2003.
- Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010.
- Claudia Hauff. "Big Data Processing, 2014/15. Lecture 5: GFS and HDFS".

# Large-Scale Data Systems

Lecture 9: Distributed Hash Tables

Prof. Gilles Louppe  
[g.louppe@uliege.be](mailto:g.louppe@uliege.be)

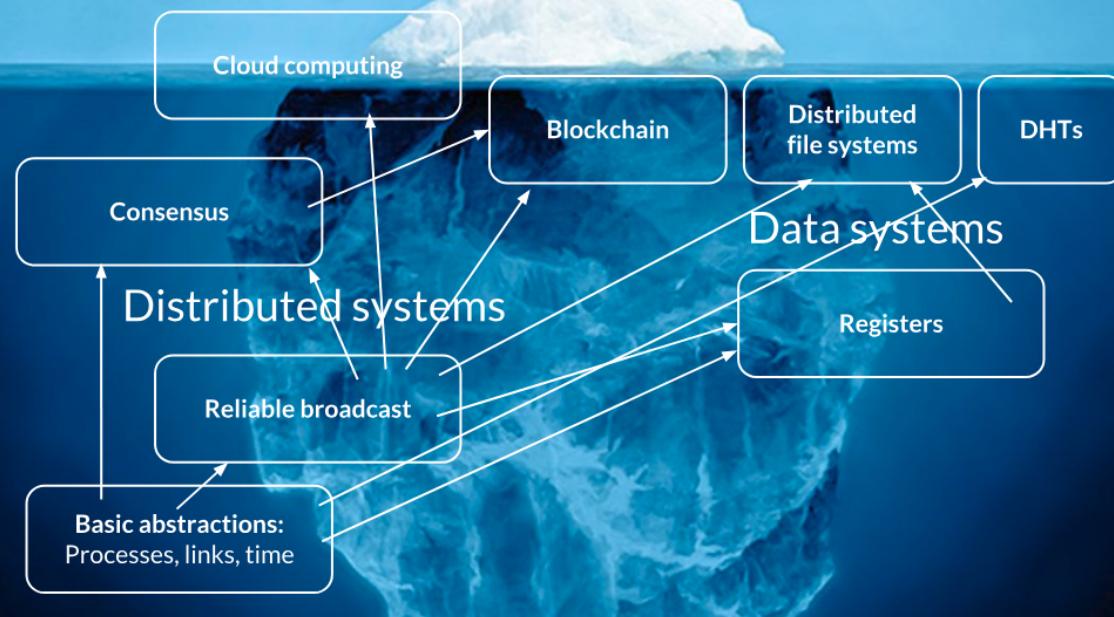


# Today

How to design a large-scale distributed system similar to a hash table?

- Chord
- Kademlia

# Data science Machine Learning Visualization

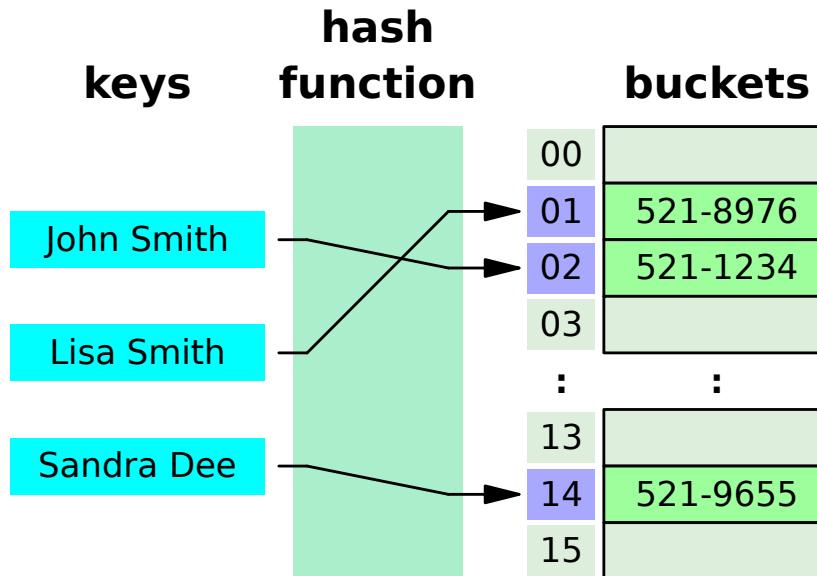


Computer networks

# Hash tables

A **hash table** is a data structure that implements an associative array abstract data type, i.e. a structure than can map keys to values.

- It uses a **hash function** to compute an index into array of buckets or slots, from which the desired value can be found.
- Efficient and scalable:  $\mathcal{O}(1)$  look-up and store operations (on a single machine).



# Distributed hash tables

A **distributed hash table** (DHT) is a class of decentralized distributed systems that provide a lookup service similar to a hash table.

- Extends upon multiple machines in the case when the data is so large we cannot store it on a single machine.
- Robust to **faults**.

## Interface

- $\text{put}(k, v)$
- $\text{get}(k)$

## Properties

- When  $\text{put}(k, v)$  is completed,  $k$  and  $v$  are reliably stored on the DHT.
- If  $k$  is stored on the DHT, a process will eventually find a node which stores  $k$ .

# Chord

# Chord

Chord is a protocol and algorithm for a peer-to-peer distributed hash table.

- It organizes the participating nodes in an **overlay network**, where each node is responsible for a set of keys.
- Keys are defined as  $m$ -bit identifiers, where  $m$  is a predefined system parameter.
- The overlay network is arranged in a **identifier circle** ranging from  $0$  to  $2^m - 1$ .
  - A **node identifier** is chosen by hashing the node IP address.
  - A **key identifier** is chosen by hashing the key.
- Based on **consistent hashing** with SHA-1 hash function.
- Supports a single operation: **lookup( $k$ )**.
  - Returns the host which holds the data associated with the key.

# Consistent hashing

## Traditional hashing

- Set of  $n$  bins.
- Key  $k$  is assigned to a particular bin.
- If  $n$  changes, all items need to be rehashed.
  - E.g. when `bin_id = hash(key) % num_bins.`

## Consistent hashing

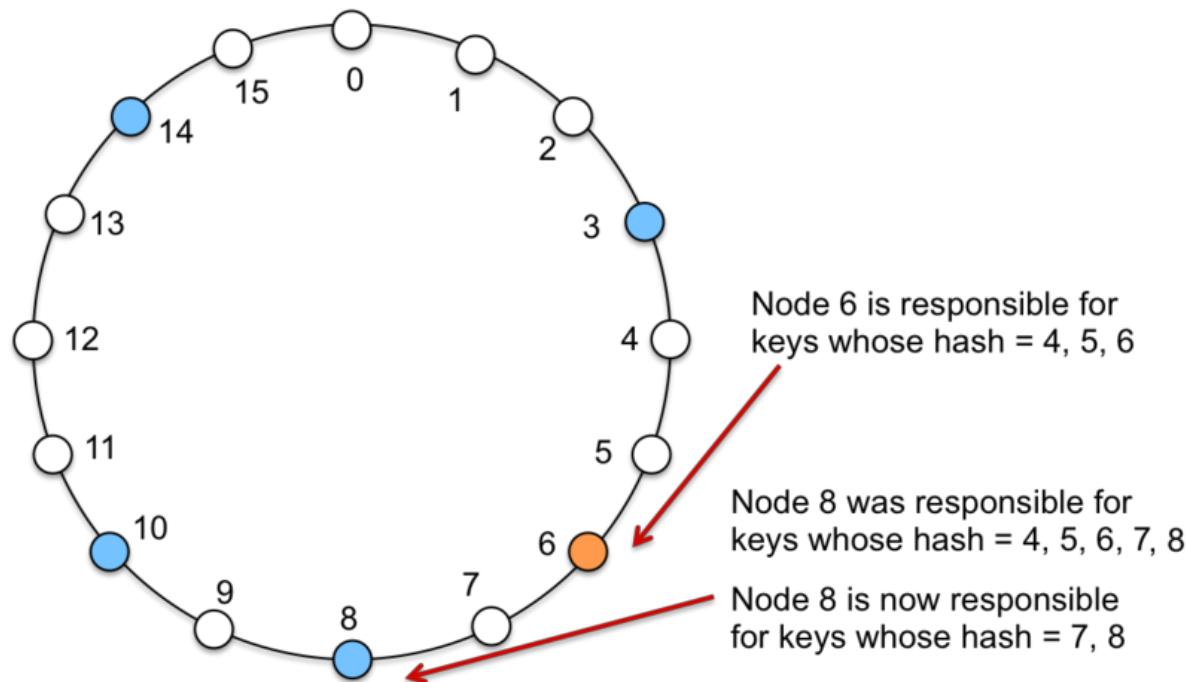
- Evenly distributes  $x$  objects over  $n$  bins.
- When  $n$  changes:
  - Only  $\mathcal{O}(\frac{x}{n})$  objects need to be rehashed.
  - Uses a deterministic hash function, independent of  $n$ .

Consistent hashing in Chord assigns keys to nodes as follows:

- Key  $k$  is assigned to the first node whose identifier is equal to or follows  $k$  in the identifier space.
  - i.e., the first node on the identifier ring starting from  $k$ .
- This node is called the **successor node** of  $k$ , denoted  $\text{successor}(k)$ .
- Enable **minimal disruption**.

To maintain the consistent (hashing) mapping, let us consider a node  $n$  which

- joins: some of the keys assigned to  $\text{successor}(n)$  are now assigned to  $n$ .
  - Which?  $\text{predecessor}(n) < k \leq n$
- leaves: All of  $n$ 's assigned keys are assigned to  $\text{successor}(n)$ .



# Routing

The core usage of the Chord protocol is to query a key from a client (generally a node as well), i.e. to find  $\text{successor}(k)$ .

## Basic query

- Any node  $n$  stores its immediate successor  $\text{successor}(n)$ , and no other information.
- If the key cannot be found locally, then the query is passed to the node's successor.
- Scalable, but  $\mathcal{O}(n)$  operations are required.
  - Unacceptable in large systems!

## Finger table

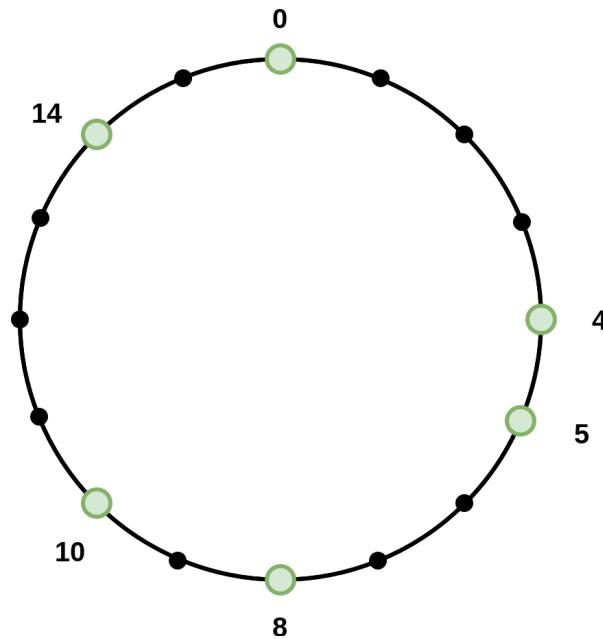
In Chord, in addition to **successor** and **predecessor** pointers, each node maintains a finger table to accelerate lookups.

- As before, let  $m$  be the number of bits in the identifier.
- Every node  $n$  maintains a routing (finger) table with at most  $m$  entries.
- Entry  $i$  in the finger table of node  $n$ :
  - First node  $s$  that succeeds  $n$  by at least  $2^{i-1}$  on the identifier circle.
  - Therefore,  $s = \text{successor}((n + 2^{i-1}) \bmod 2^m)$

Notation	Definition
$\text{finger}[k].start$	$(n + 2^{k-1}) \bmod 2^m, 1 \leq k \leq m$
$.interval$	$[\text{finger}[k].start, \text{finger}[k+1].start)$
$.node$	first node $\geq n.\text{finger}[k].start$
$\text{successor}$	the next node on the identifier circle; $\text{finger}[1].node$
$\text{predecessor}$	the previous node on the identifier circle

## Example

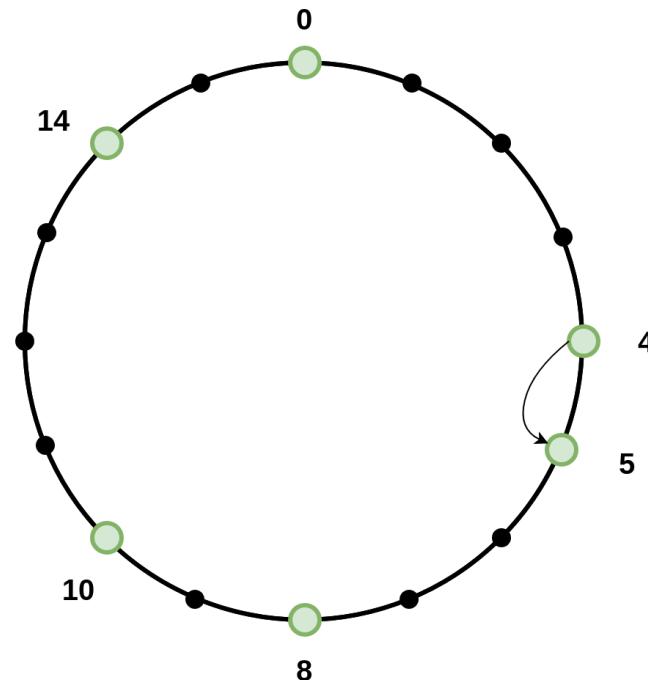
- $m = 4$  bits  $\rightarrow$  max 4 entries in the table.
- $i$ -th entry in finger table:  $s = \text{successor}((n + 2^{i-1}) \bmod 2^m)$



## Example: first entry

- $n = 4, i = 1$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(5) = 5$

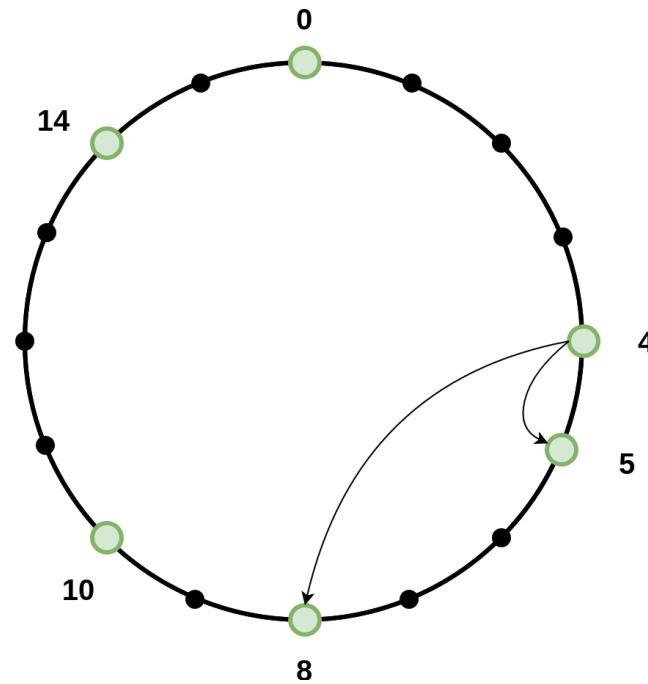
Entry	Interval	Successor
1	[5,6)	5



## Example: second entry

- $n = 4, i = 2$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(6) = 8$

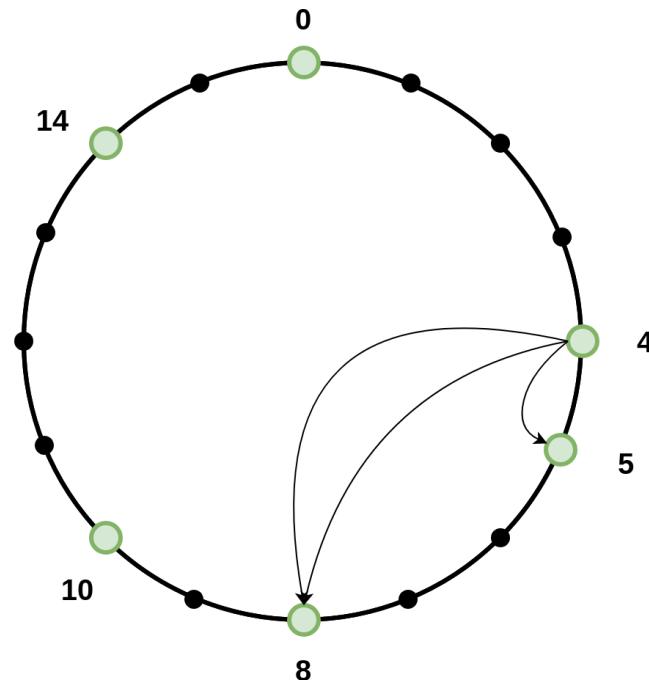
Entry	Interval	Successor
1	[5,6)	5
2	[6,8)	8



## Example: third entry

- $n = 4, i = 3$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(8) = 8$

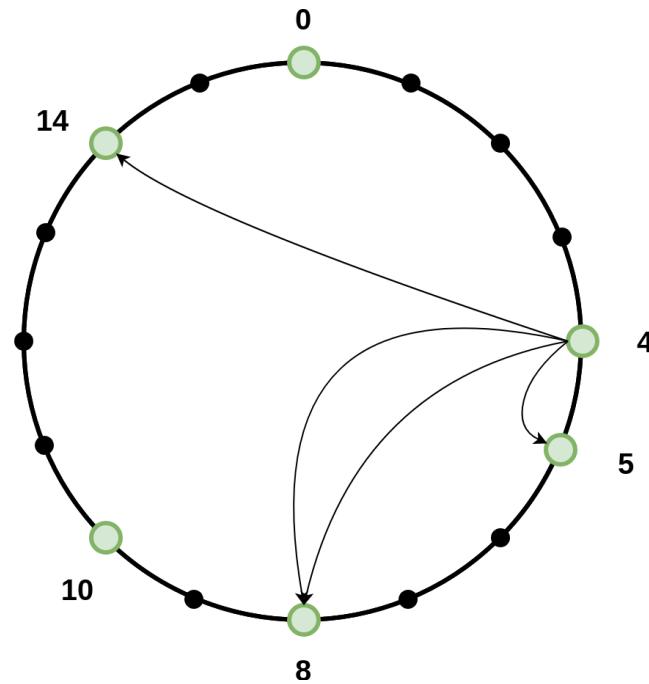
Entry	Interval	Successor
1	[5,6)	5
2	[6,8)	8
3	[8,12)	8



## Example: fourth entry

- $n = 4, i = 4$
- $s = \text{successor}((n + 2^{i-1}) \bmod 2^m) = \text{successor}(12) = 14$

Entry	Interval	Successor
1	[5,6)	5
2	[6,8)	8
3	[8,12)	8
4	[12, 4)	14



## Improved lookup

A lookup for  $\text{successor}(k)$  now works as follows:

- if  $k$  falls between  $n$  and  $\text{successor}(n)$ , return  $\text{successor}(n)$ .
- otherwise, the lookup is forwarded at  $n'$ , where  $n'$  is the node in the finger table that most immediately precedes  $k$ .
- Since each node has finger entries at power of two intervals around the identifier circle, each node can forward a query at least halfway along the remaining distance between the node and the target key.
- $\mathcal{O}(\log N)$  nodes need to be contacted.

```
// ask node n to find the successor of id
n.find_successor(id)
  if (id ∈ (n, successor])
    return successor;
  else
    // forward the query around the circle
    n0 = successor.closest_preceding_node(id);
    return n0.find_successor(id);

// search the local table for the highest predecessor of id
n.closest_preceding_node(id)
  for i = m downto 1
    if (finger[i] ∈ (n,id))
      return finger[i];
  return n;
```

## Example: finding successor( $k = 3$ ) from node<sub>4</sub>

1. node<sub>4</sub> checks if  $k$  is in the interval (4, 5].
2. No, node<sub>4</sub> checks its finger table (starting from the last entry, i.e.,  $i = m$ ).
  1. Is node<sub>14</sub> in the interval (4, 4)? Yes!
3. node<sub>14</sub> checks if  $k$  is in the interval (14, 0].
4. No, node<sub>14</sub> checks its finger table for closest preceding node.
  1. Return node<sub>0</sub>.
5. node<sub>0</sub> checks if  $k$  is in the interval (0, 4]. Yes!

→ Node 0 is the preceding node of  $k = 4$ . Therefore  
successor( $k = 3$ ) = node<sub>0</sub>.successor = 4.

Of course, one could implement a mechanism that prevents node<sub>4</sub> from looking up its own preceding node in the network.

# Join

We must ensure the network remains consistent when a node  $n$  joins by connecting to a node  $n'$ . This is performed in three steps:

1. Initialize the successor of  $n$ .
2. Update the fingers and predecessors of existing nodes to reflect the addition of  $n$ .
3. Transfer the keys and their corresponding values to  $n$ .

## Initializing $n$ 's successor

$n$  learns its successor by asking  $n'$  to look them up.

```
// join a Chord ring containing node n'.  
n.join(n')  
predecessor = nil;  
successor = n'.find_successor(n);
```

## Periodic consistency check

```
// called periodically. n asks the successor
// about its predecessor, verifies if n's immediate
// successor is consistent, and tells the successor about n
n.stabilize()
    x = successor.predecessor;
    if (x∈(n, successor))
        successor = x;
    successor.notify(n);

// n' thinks it might be our predecessor.
n.notify(n')
    if (predecessor is nil or n'∈(predecessor, n))
        predecessor = n';

// called periodically. refreshes finger table entries.
// next stores the index of the finger to fix
n.fix_fingers()
    next = next + 1;
    if (next > m)
        next = 1;
    finger[next] = find_successor(n+2^{next-1});
```

## Transferring keys

- $n$  can become the successor only for keys that were previously the responsibility of the node immediately following  $n$ .
- $n$  only needs to contact **successor( $n$ )** to transfer responsibility of all relevant keys.

# Fault-tolerance

## Failures

- Since the successor (or predecessor) of a node may disappear from the network (because of failure or departure), each node records a whole segment of the circle adjacent to it, i.e., the  $r$  nodes following it.
- This successor list results in a high probability that a node is able to correctly locate its successor (or predecessor), even if the network in question suffers from a high failure rate.

## Replication

- Use the same successor-list to replicate the data on the segment!

# Summary

- Fast lookup  $\mathcal{O}(\log N)$ , small routing table  $\mathcal{O}(\log N)$ .
- Handling failures and addressing replication (load balance) using same mechanism (successor list).
- Relatively small join/leave cost.
- Iterative lookup process.
- Timeouts to detect failures.
- No guarantees (with high probability ...).
- Routing tables must be correct.

# Kademlia

# Kademlia

Kademlia is a peer-to-peer hash table with **provable** consistency and performance in a fault-prone environment.

- Configuration information spreads automatically as a side-effect of key look-ups (gossiping).
- Nodes have enough knowledge and flexibility to route queries through low-latency paths.
- Asynchronous queries to avoid timeout delays from failed nodes.
- Minimizes the number of configuration messages (guarantee).
- 160-bit identifiers (e.g., using SHA-1 or some other hash function, implementation specific).
- Key-Value pairs are stored on nodes based on **closeness** in the identifier space.
- Identifier based **routing** algorithm by imposing a **hierarchy** (virtual overlay network).

# System description

Nodes are structured in an overlay network where they correspond to the leaves of an (unbalanced) binary tree, with each node's position determined by the shortest unique prefix of its identifier.

- Node identifiers are chosen at random in the identifier space.
- Kamdelia ensures that every node knows at least one other node in every subtree. This guarantees that any node can locate any other node given its identifier.

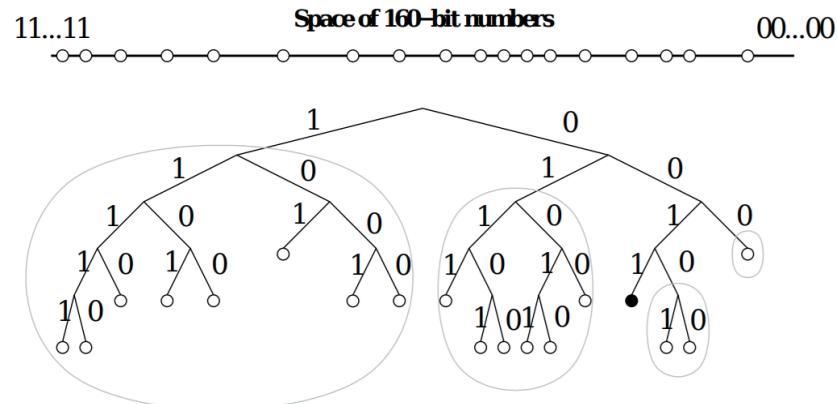


Fig. 1: Kademlia binary tree. The black dot shows the location of node  $0011\dots$  in the tree. Gray ovals show subtrees in which node  $0011\dots$  must have a contact.

## Node distance

The distance between two identifiers is defined as

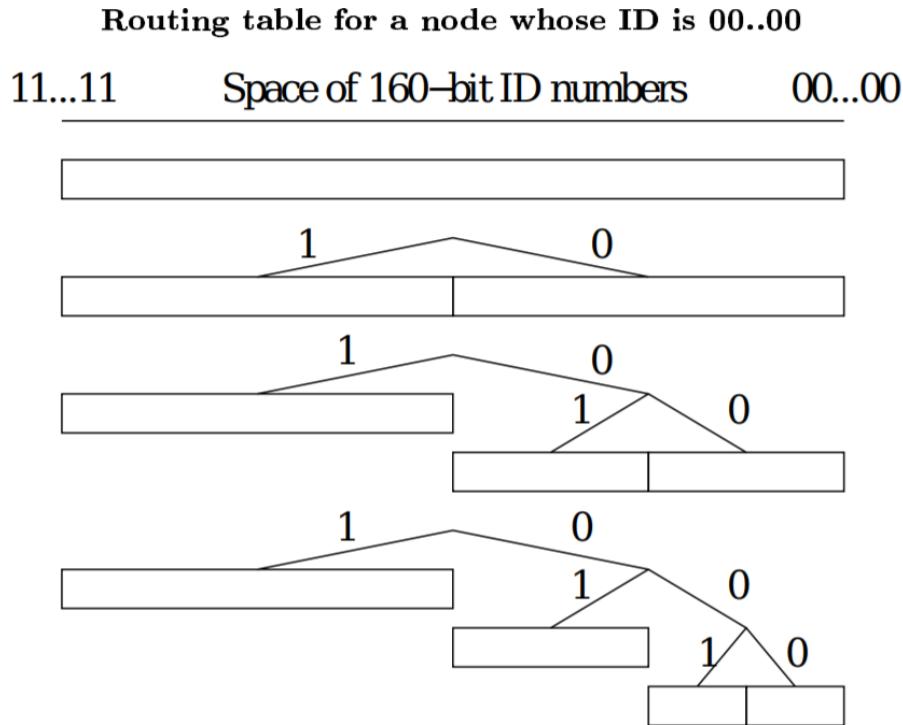
$$d(x, y) = x \oplus y.$$

- XOR is a valid, albeit non-Euclidean metric.
- XOR captures the notion of distance between two identifiers: in a fully-populated binary tree of 160-bit IDs, it is the height of the smallest subtree containing them both.
- XOR is symmetric.
- XOR is unidirectional.

## Node state

- For every prefix  $0 < i < 160$ , every node keeps a list, called a **k-bucket**, of (IP address, Port, ID) for nodes of distance between  $2^i$  and  $2^{i+1}$  of itself.
- Every k-bucket is sorted by time last seen (least recently seen first).
- When a node receives a message, it updates the corresponding k-bucket for the sender's identifier. If the sender already exists, it is moved to the tail of the list.
  - **Important:** If the k-bucket is full, the node pings the **least recently** seen node and checks if it is still available.
    - Only if the node is **not available** it will replace it.
    - If available, the node is pushed back at the end of the bucket.
  - Policy of replacement only when a nodes leaves the network → prevents Denial of Service (DoS) attacks (e.g., flushing routing tables).

## k-bucket



**Fig. 4:** Evolution of a routing table over time. Initially, a node has a single  $k$ -bucket, as shown in the top routing table. As the  $k$ -buckets fill, the bucket whose range covers the node's ID repeatedly splits into two  $k$  buckets.

# Interface

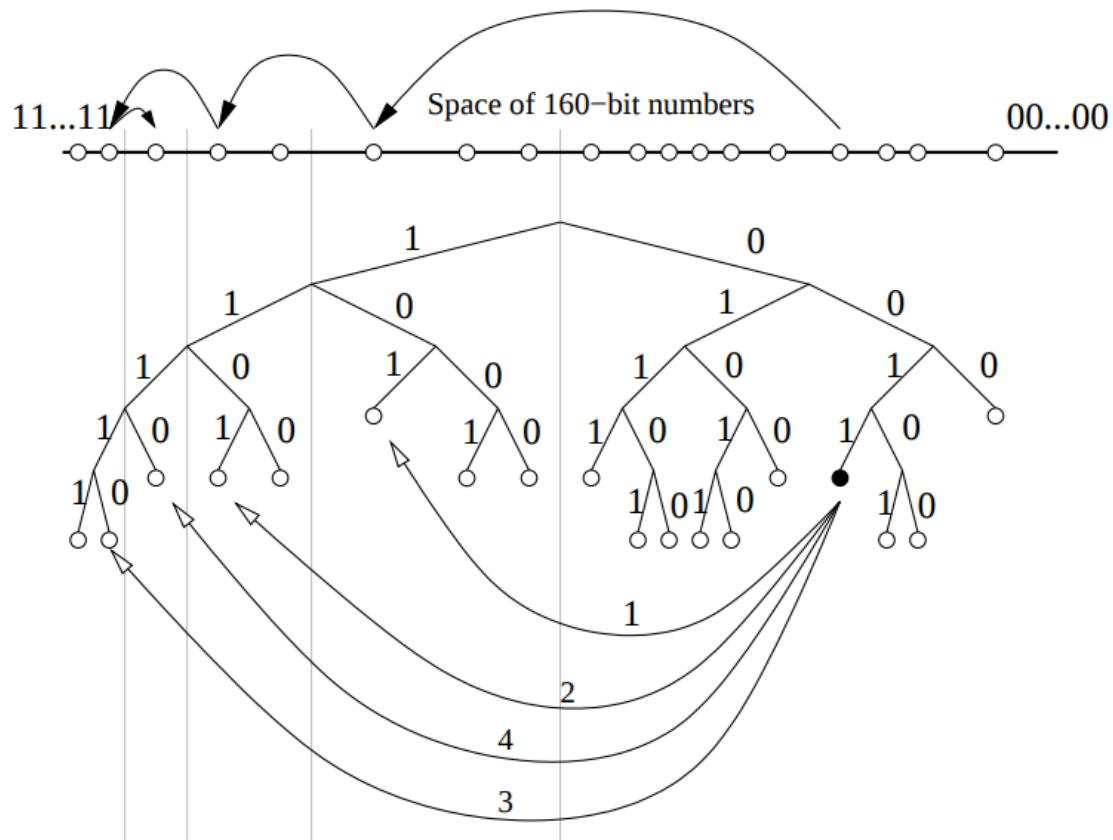
Kademlia provides four remote procedure calls (RPCs):

- **PING(*id*)** returns (IP, Port, ID)
  - Probes the node to check whether it is still online.
- **STORE(*key*, *value*)**
- **FIND\_NODE(*id*)** returns (IP, Port, ID) for the *k* nodes it knows about closest to ID.
- **FIND\_VALUE(*key*)** returns (IP, Port, ID) for the *k* nodes it knows about closest to ID, or the value if it maintains the key.

## Node lookup

The most important procedure a Kademlia participant must perform is locating the  $k$  closest nodes to some given identifier.

- Kademlia achieves this by performing a recursive lookup procedure.
- The initiator issues asynchronous FIND\_NODE requests to  $\alpha$  (system parameter) nodes from its closest non-empty k-bucket.
  - Parallel search with the cost of increased network traffic.
  - Nodes return the  $k$  closest nodes to the query ID.
  - Repeat and select the  $\alpha$  nodes from the new set of nodes.
  - Terminate when set doesn't change.
  - Possible optimization: choose  $\alpha$  nodes with lowest latency.



**Fig. 2: Locating a node by its ID.** Here the node with prefix 0011 finds the node with prefix 1110 by successively learning of and querying closer and closer nodes. The line segment on top represents the space of 160-bit IDs, and shows how the lookups converge to the target node. Below we illustrate RPC messages made by 1110. The first RPC is to node 101, already known to 1110. Subsequent RPCs are to nodes returned by the previous RPC.

## Storing data

Using the lookup procedure, **storing** and making data **persistent** is trivial.

→ Send a ST0RE RPC to the  $k$  closest nodes identified by the lookup procedure.

- To ensure persistence in the presence of **node failures**, every node periodically republishes the key-value pair to the  $k$  closest nodes.
- Updating scheme can be implemented. For example: delete data after 24 hours after publication to limit stale information.

## Retrieving data

1. Find  $k$  closest nodes of the specified identifier using `FIND_VALUE(id)`.
2. Halt procedure immediately whenever the set of closest nodes doesn't change or a value is returned.

→ For caching purposes, once a lookup succeeds, the requesting node stores the key-value pair at the **closest node to the key that did not return the value**.

Because of the **unidirectionality** of the topology (requests will usually follow the same path), future searches for the same key are likely to hit cached entries before querying the closest node.

→ Induces problem with popular nodes: **over-caching**.

**Solution:** Set expiration time **inversely proportional** to the distance between the true identifier and the current node identifier.

## Join

Straightforward approach compared to other implementations.

1. Node  $n$  initializes its k-bucket (empty).
2. A node  $n$  connects to an already participating node  $j$ .
3. Node  $n$  then performs a **node-lookup** for its own identifier.
  - o Yielding the  $k$  closest nodes.
  - o By doing so  $n$  inserts itself in other nodes  $k$ -buckets.

**Note:** The new node should store keys which are the closest to its own identifier by obtaining the  $k$ -closest nodes.

## Leave and failures

Leaving is very simple as well. Just disconnect.

- Failure handling is **implicit** in Kademlia due to **data persistence**.
- No special actions required by other nodes (failed node will just be removed from the k-bucket).

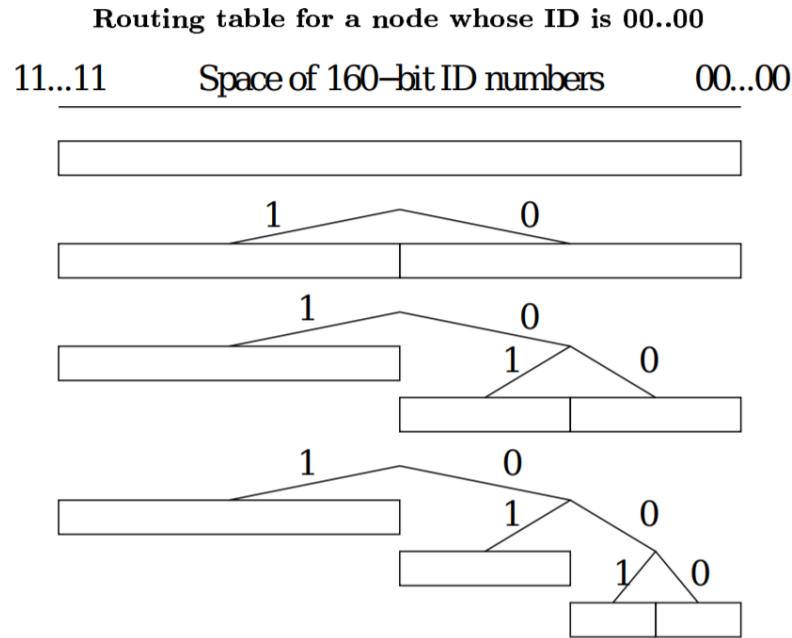
# Routing table

The routing table is an (unbalanced) binary tree whose leaves are  $k$ -buckets.

- Every  $k$ -bucket contains some nodes with a common prefix.
- The shared prefix is the  $k$ -buckets position in the binary tree.
- Thus, a  $k$ -buckets covers some range of the 160 bit identifier space.
- All  $k$ -buckets cover the **complete** identifier space with **no** overlap.

## Dynamic construction of the routing table

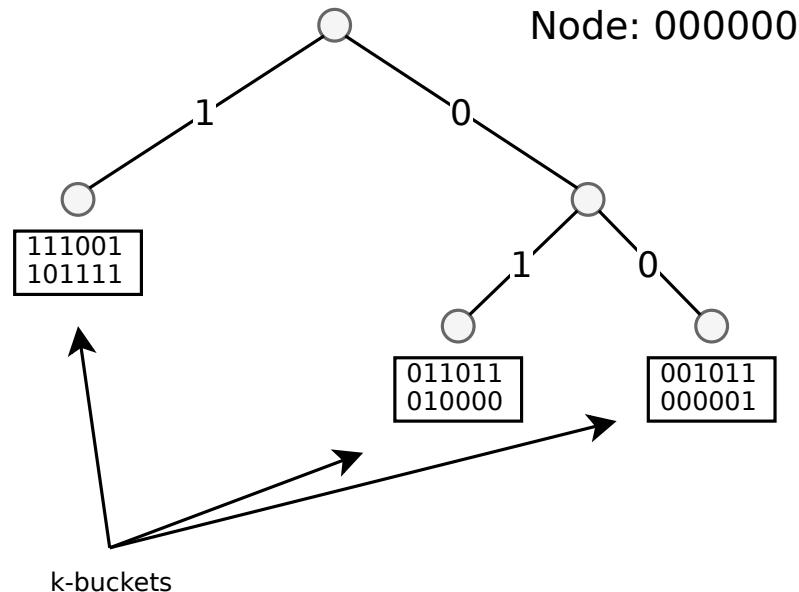
- Nodes in the routing table are allocated dynamically as needed.
- A bucket is split whenever the  $k$ -bucket is **full** and the range **includes** the node's own **identifier**.



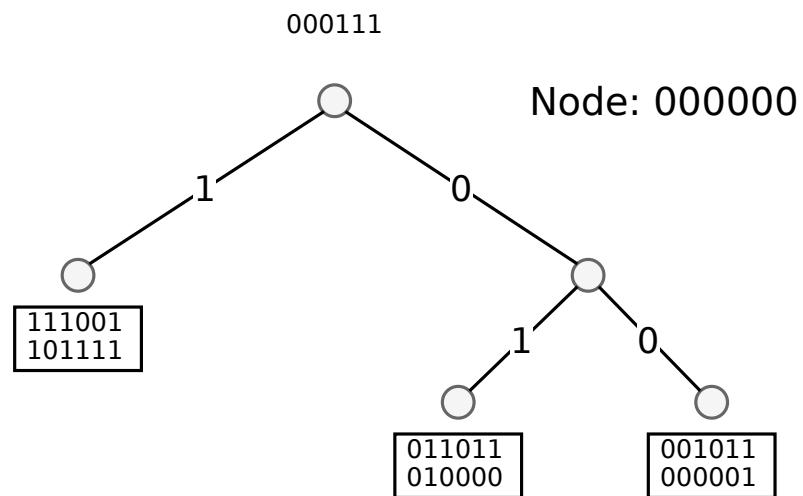
**Fig. 4:** Evolution of a routing table over time. Initially, a node has a single  $k$ -bucket, as shown in the top routing table. As the  $k$ -buckets fill, the bucket whose range covers the node's ID repeatedly splits into two  $k$  buckets.

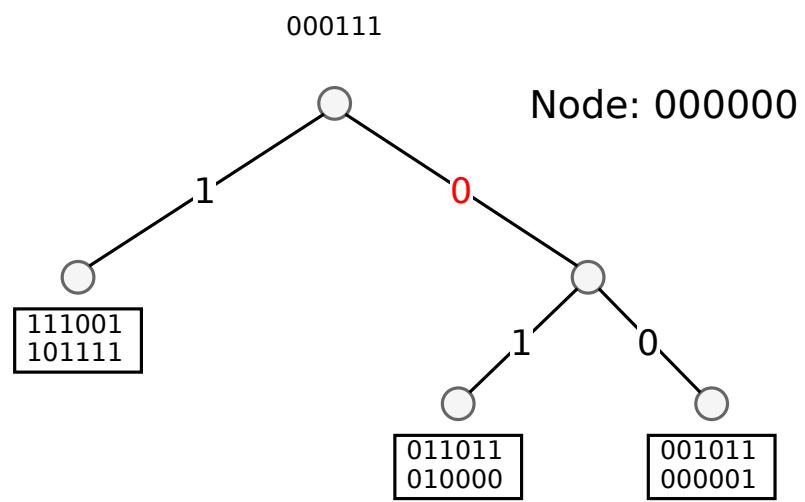
## Example

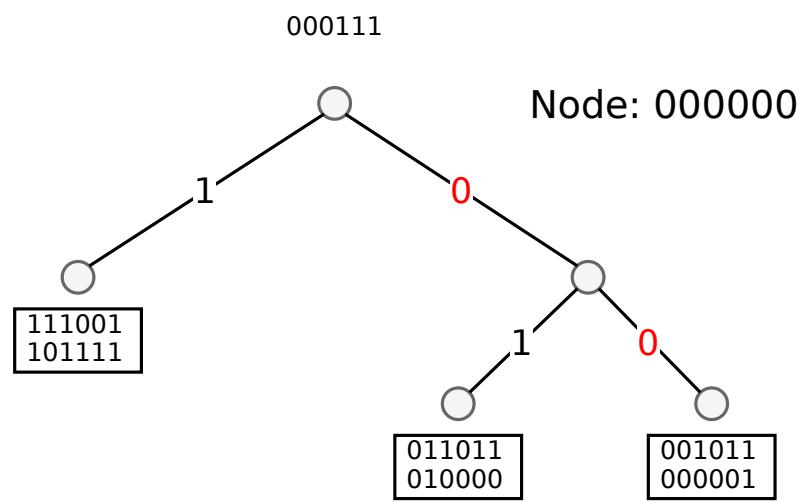
- $k = 2$
- $\alpha = 1$  (no asynchronous requests, also no asynchronous pings)
- Node identifier (000000) is **not** in the routing table

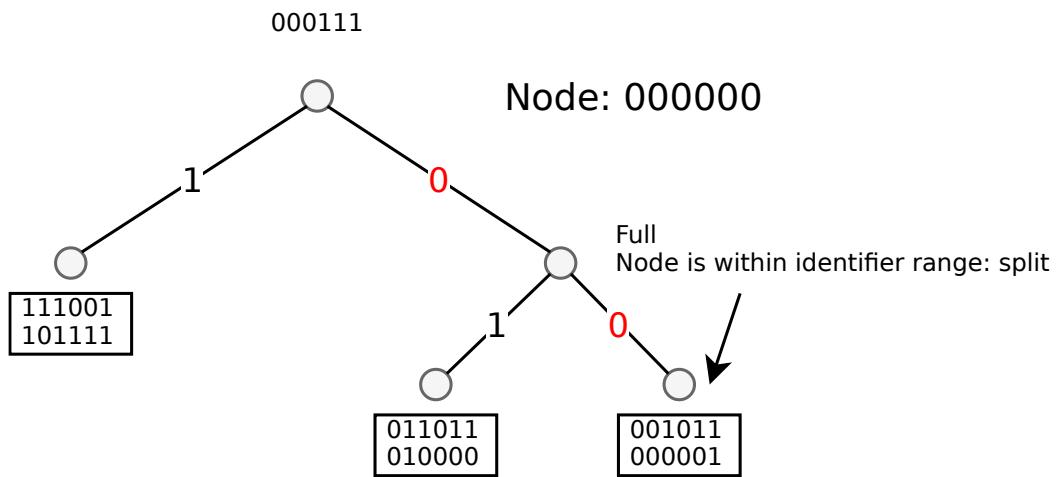


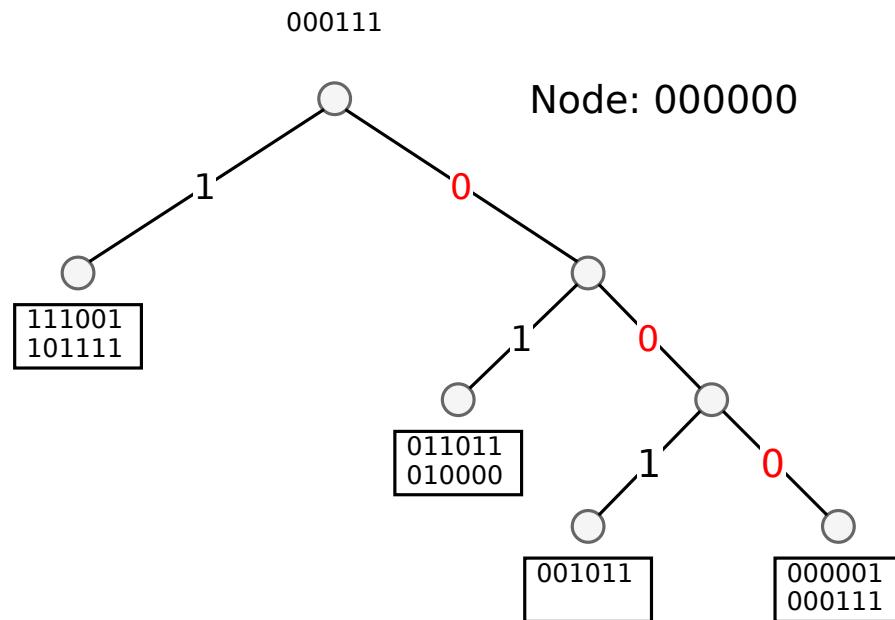
Node 000111 is involved with an RPC request, what happens?

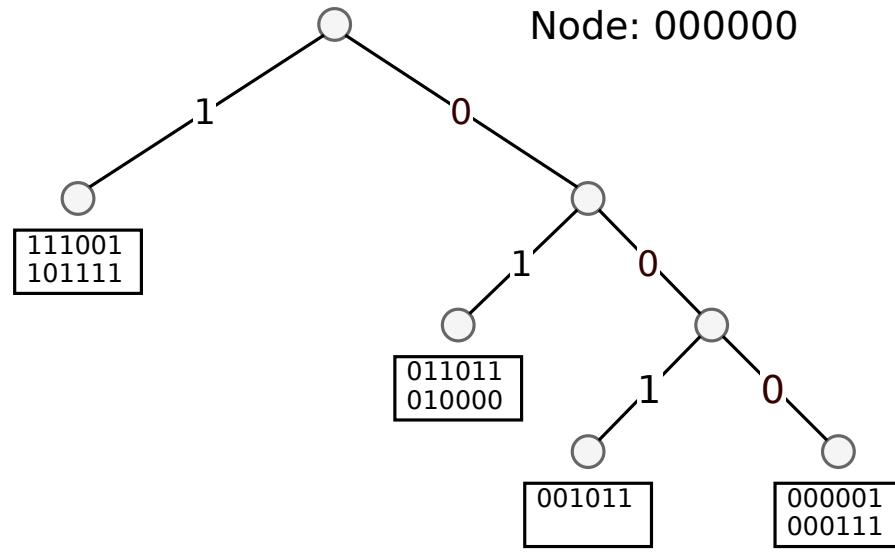




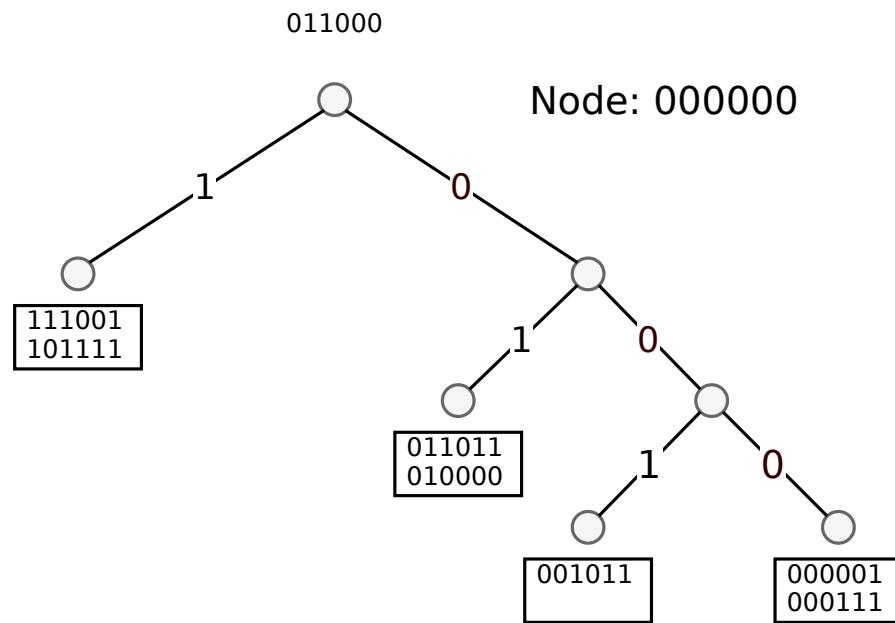


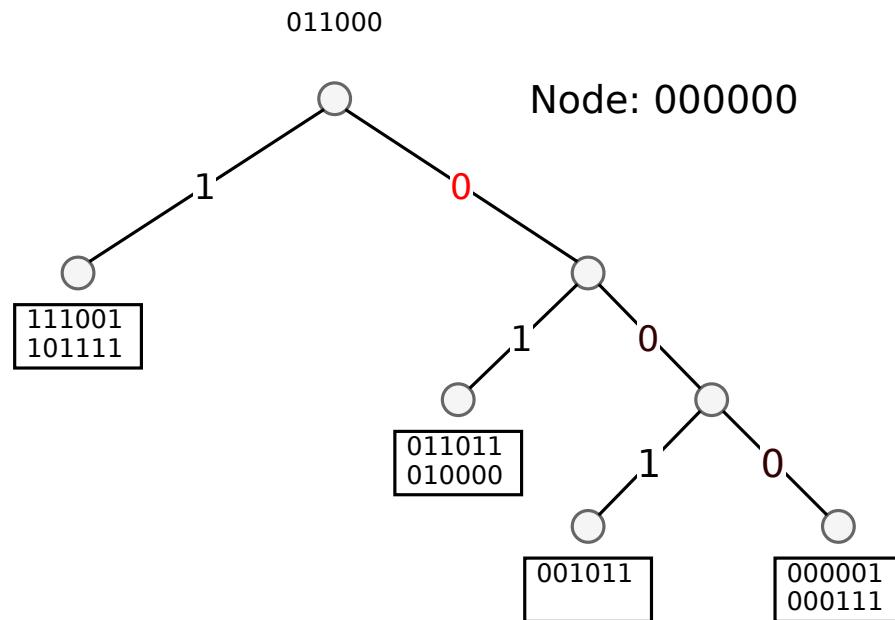


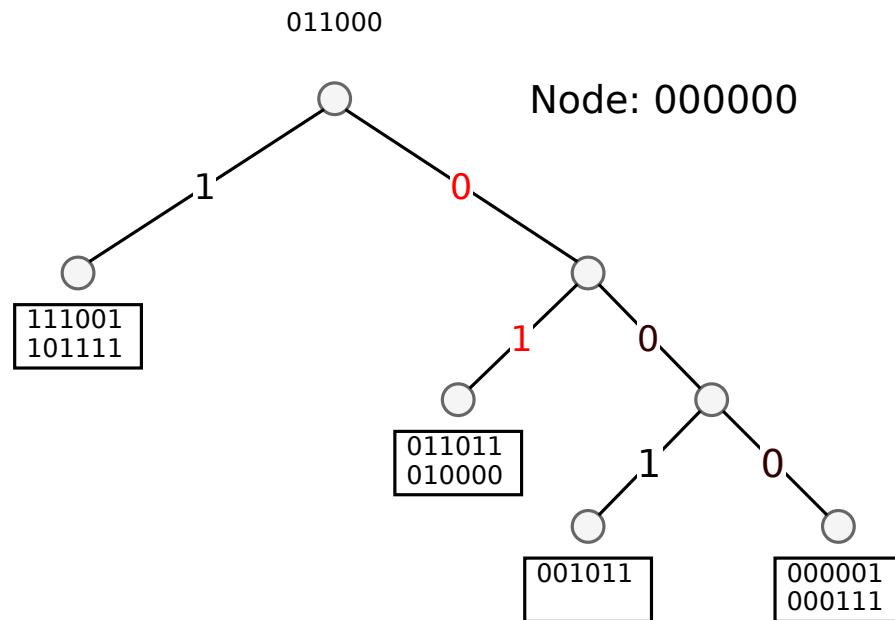


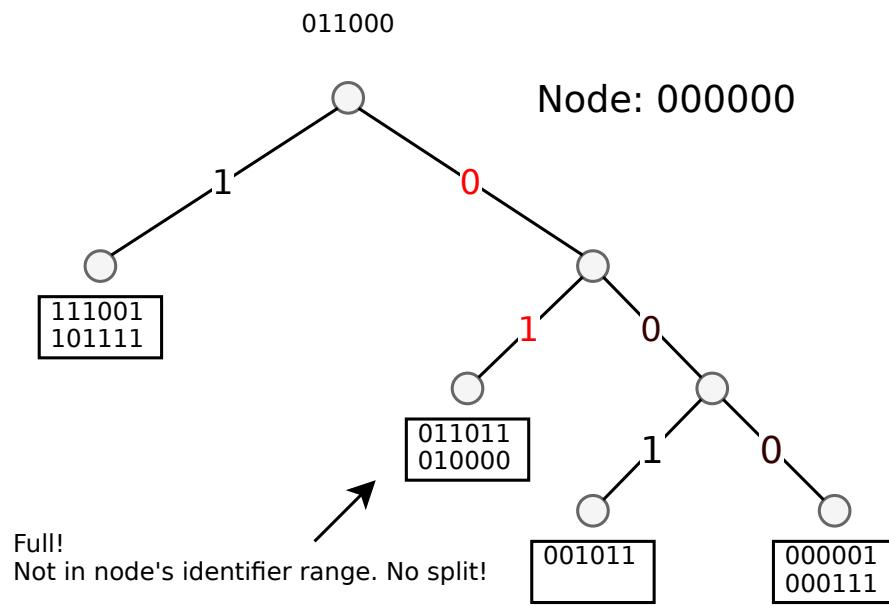


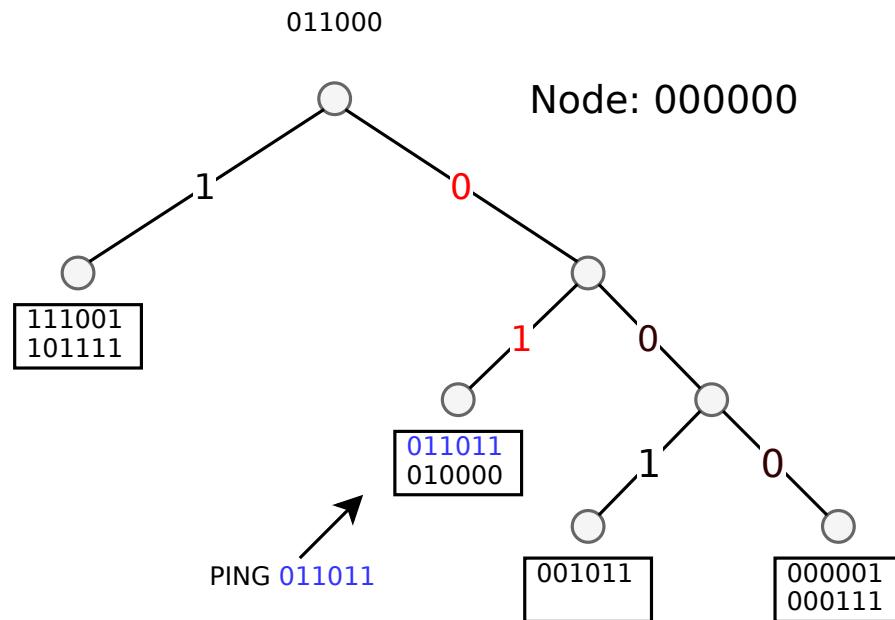
A new node 011000 is involved with a RPC message.

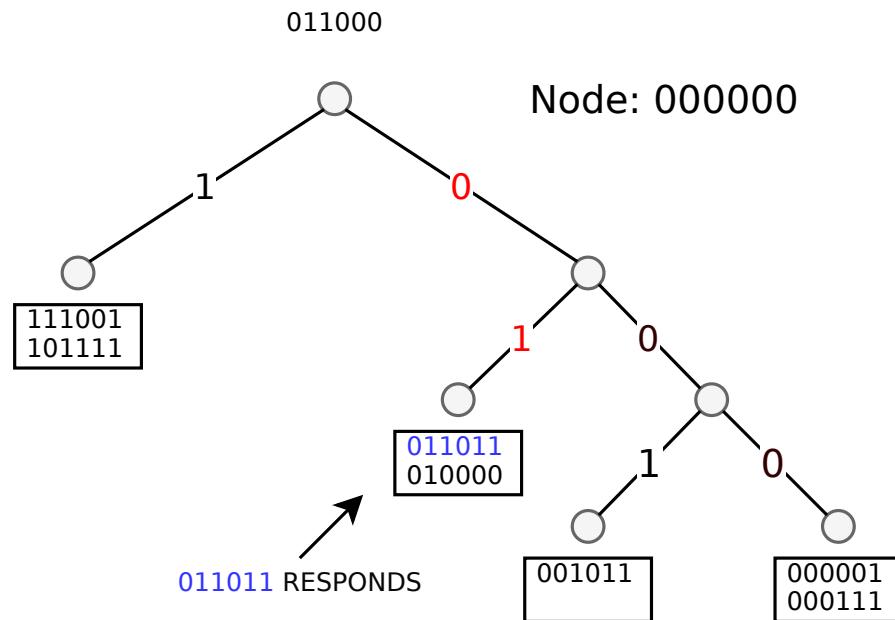


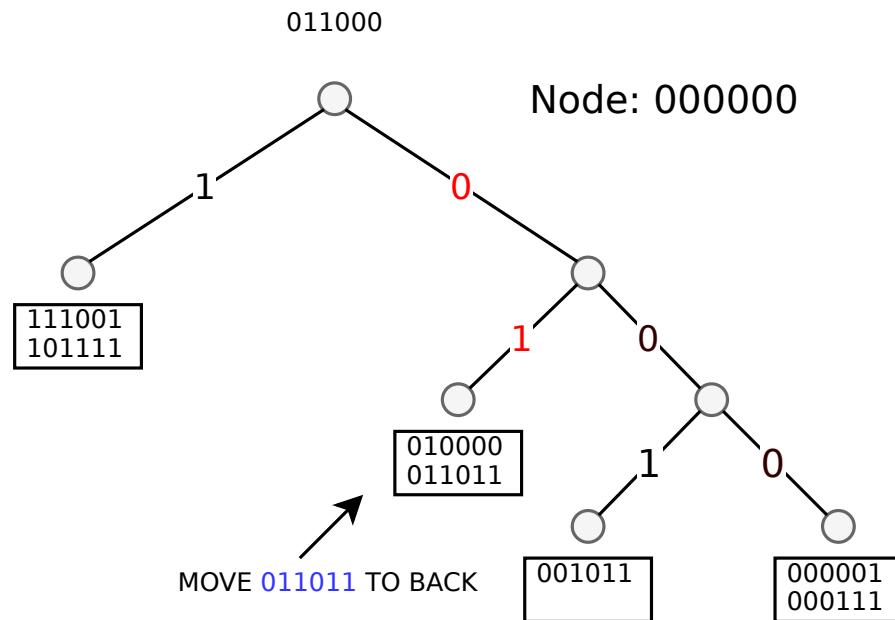


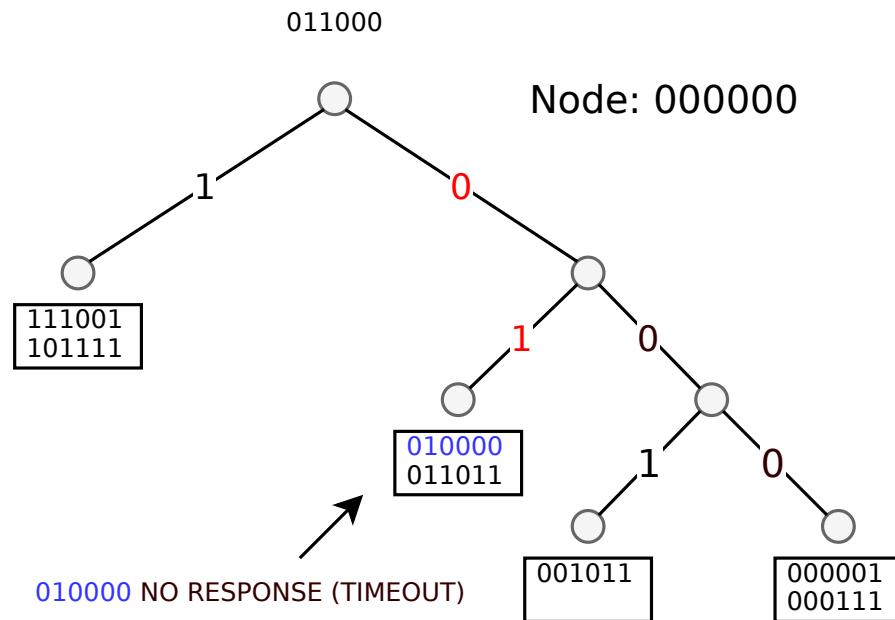


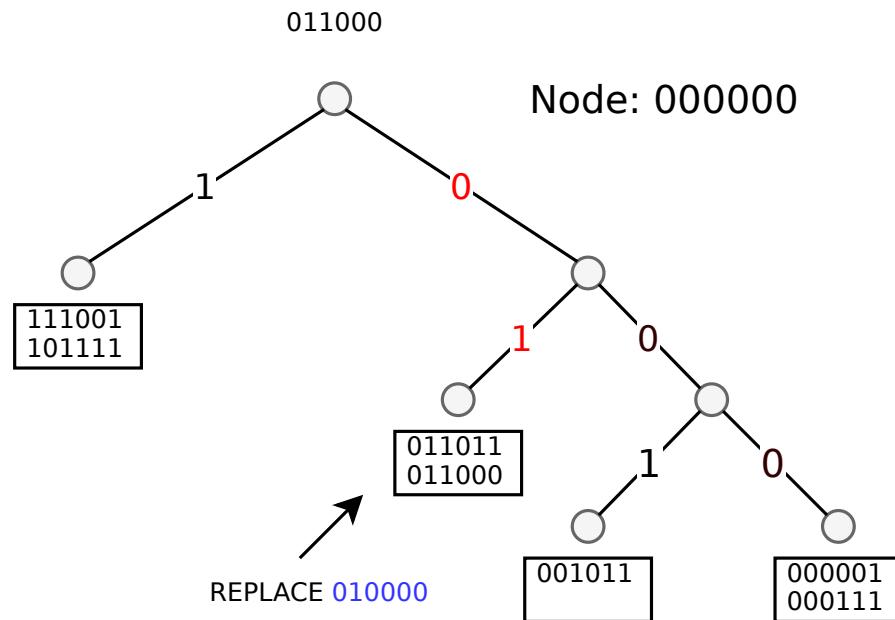


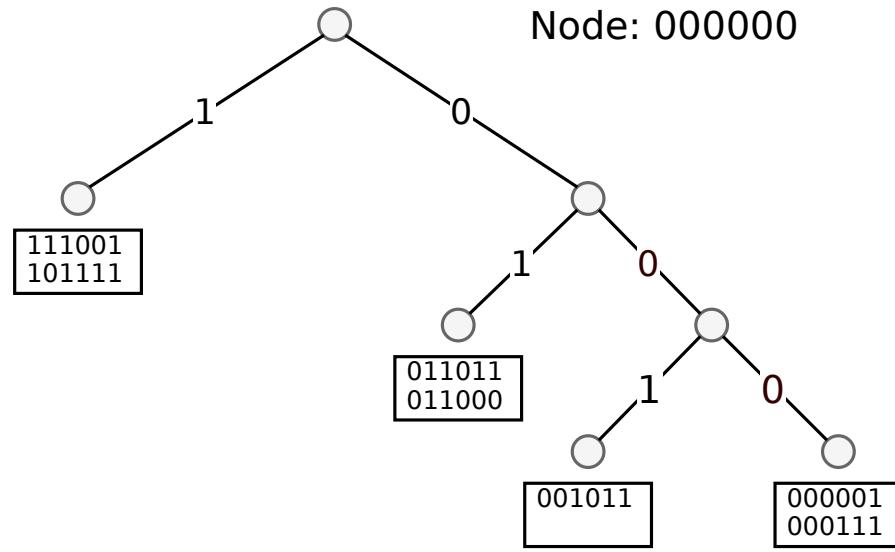












# Summary

- Efficient, guaranteed look-ups  $\mathcal{O}(\log N)$
- XOR-based metric topology (provable consistency and performance).
- Possibly latency minimizing (by always picking the lowest latency note when selecting  $\alpha$  nodes).
- Lookup is iterative, but concurrent ( $\alpha$ ).
- Kademlia protocol implicitly enables data persistence and recovery, no special failure mechanisms required.
- Flexible routing table robust against DoS (route table flushing).



# References

- Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review, 31(4), 149-160.
- Maymounkov, P., & Mazieres, D. (2002, March). Kademlia: A peer-to-peer information system based on the xor metric. In International Workshop on Peer-to-Peer Systems (pp. 53-65). Springer, Berlin, Heidelberg.