Exam and Term Project

- Exam
  - Exam, Monday, April 8, 2024
  - Time: 3:05pm – 5:05pm; Room XXxxx
  - Close book, Close note, no phone, calculator, or computer or any internet access

- Term Project
  - Presentation, Monday and Tuesday, April 22
  - Term project report, April 25, 2024
  - Progress report March 20

Guest Lecture After Spring Break

- Wednesday
  - Dr. Bogdan Nicolae
  - AI, Deep Learning, and HPC

Chapter 8: Fault Tolerance

- Basic concepts in fault tolerance
- Masking failure by redundancy
- Process resilience
- Reliable communication
  - One-one communication
  - One-many communication
- Distributed commit
  - Two phase commit
  - Three phase commit
- Failure recovery
  - Checkpointing
  - Message logging

Motivation

- Single machine systems
  - Failures are all or nothing
    - OS crash, disk failures
- Distributed systems: multiple independent nodes
  - Partial failures are also possible (some nodes fail)
- *Question:* Can we automatically recover from partial failures?
  - Important issue since probability of failure grows with number of independent components (nodes) in the systems
  - Prob(failure) = Prob(Any one component fails)=1-P(no failure)

Reliability Concerns

- Systems are getting bigger
  - O(10,000)~ O(100,000) processor systems are being designed/deployed
- Even highly reliable HW can become an issue at scale
  - 1 node fails every 10,000 hours
  - 6,000 nodes fail every 1.6 hours
  - 64,000 nodes fail every 5 minutes

 A needs for new fault management!

Simple checkpointing is not good enough!

A Perspective

- Computing systems are not very reliable
  - OS crashes frequently (Windows), buggy software, unreliable hardware, software/hardware incompatibilities
  - Until recently: computer users were “tech savvy”
    - Could depend on users to reboot, troubleshoot problems
  - Growing popularity of Internet/World Wide Web
    - “Novice” users
    - Need to build more reliable/dependable systems
  - Example: what is your TV (or car) broke down every day?
    - Users don’t want to “restart” TV or fix it (by opening it up)
    - The cost of reparie
- Need to make computing systems more reliable

Basic Concepts

- Need to build *dependable* systems
- Requirements for dependable systems
  - *Availability*: ready to be used
  - *Reliability:* run continuously without failure
  - *Safety:* no catastrophic, even at the fail time
  - *Maintainability: easy to repair*

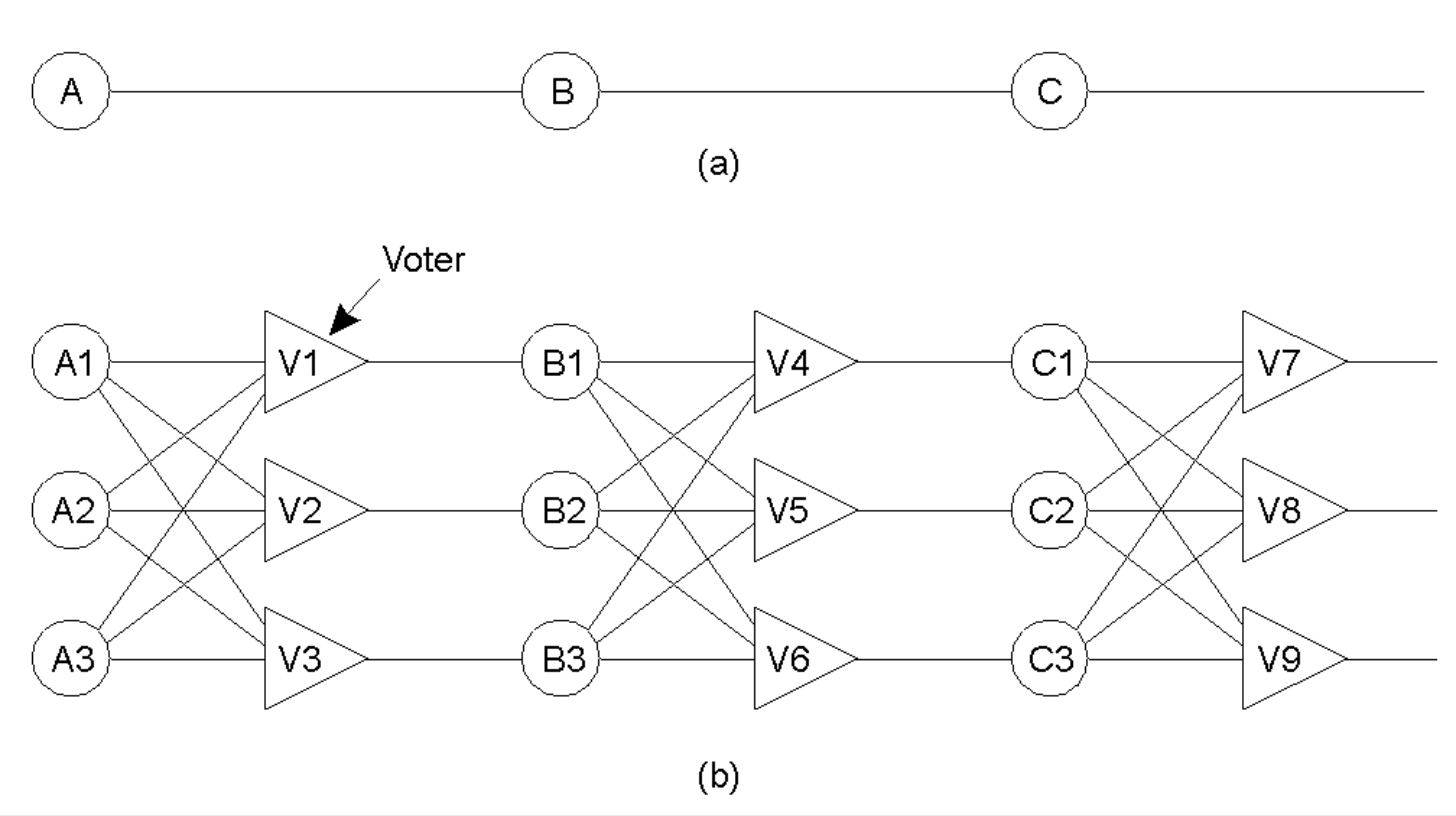
Basic Concepts (cont.)

- Fault: the cause of an error which makes the system cannot meet its promise
- Fault tolerance: system should provide services despite faults
- Three types:
  - Transient faults: not appear again
  - Intermittent faults: may appear again
  - Permanent faults: always there

Failure Models

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Failure Masking by Redundancy



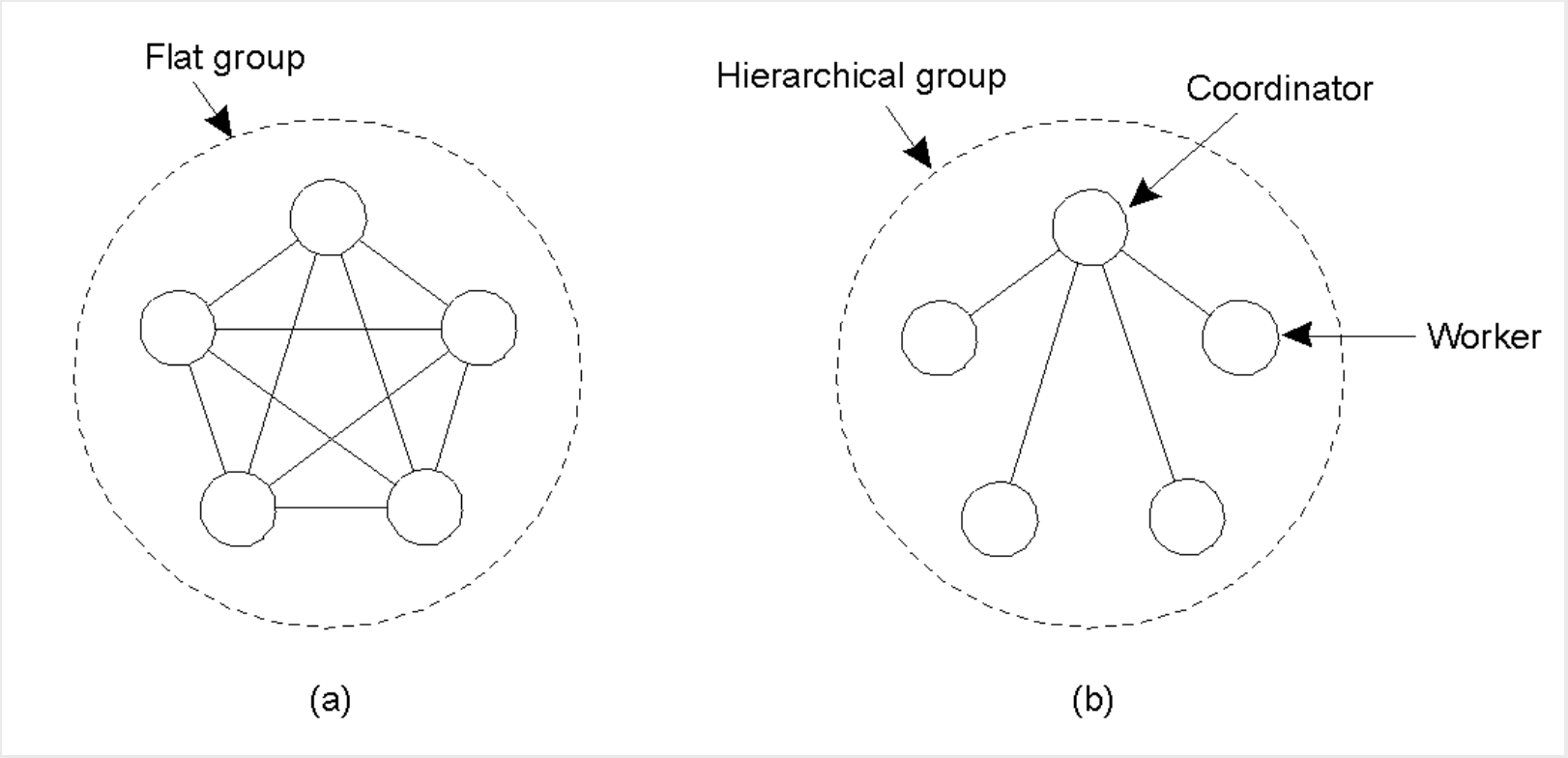
Triple modular redundancy

- Information redundancy, time redundancy, physical redundancy

Process Resilience

- Handling faulty processes: organize several processes into a group
  - All processes perform the same computation
  - All messages are sent to all members of the group
  - Majority need to agree on results of a computation
  - Ideally want multiple, independent implementations of the application (to prevent identical bugs)
- Use *process groups* to organize such processes

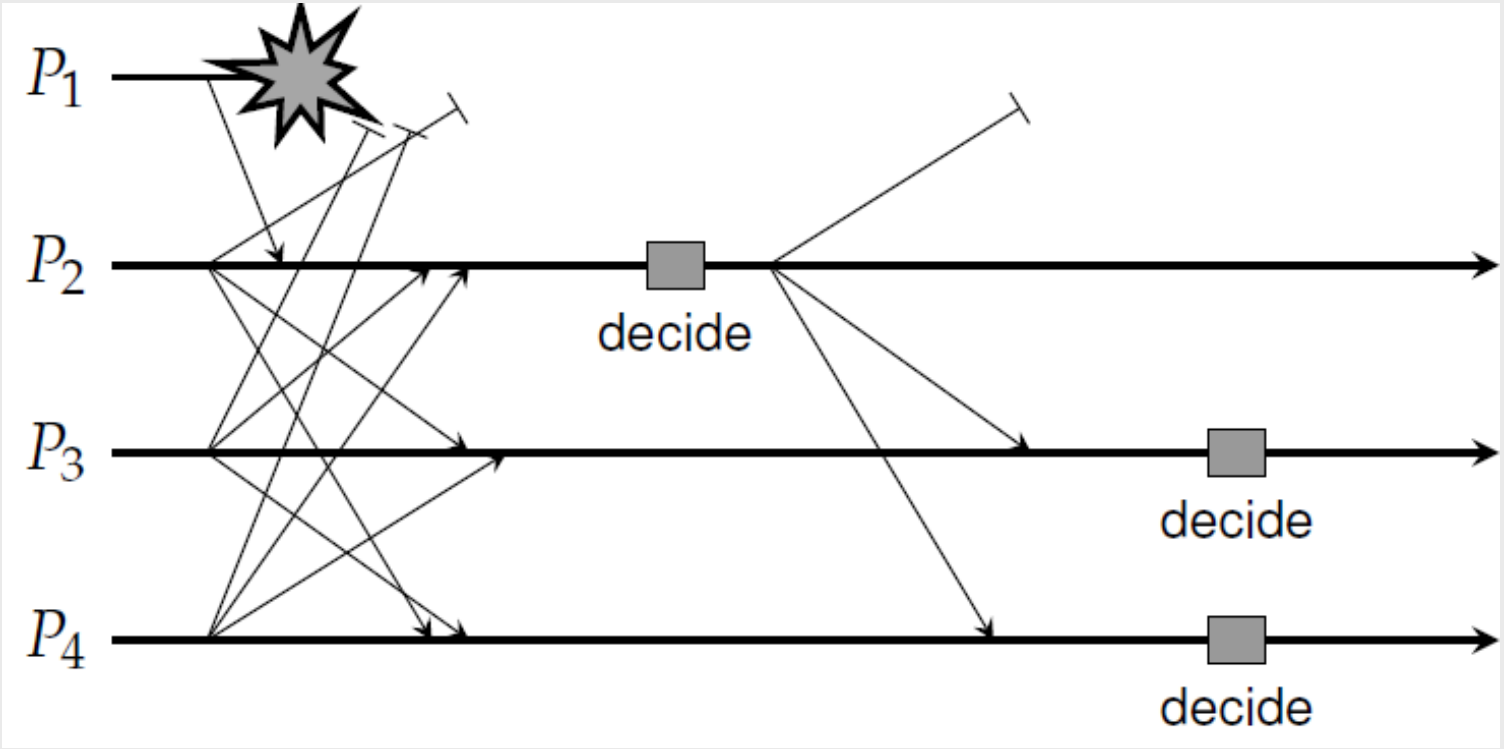
Flat Groups versus Hierarchical Groups



Advantages and disadvantages?

Membership: group server, multicast announcement

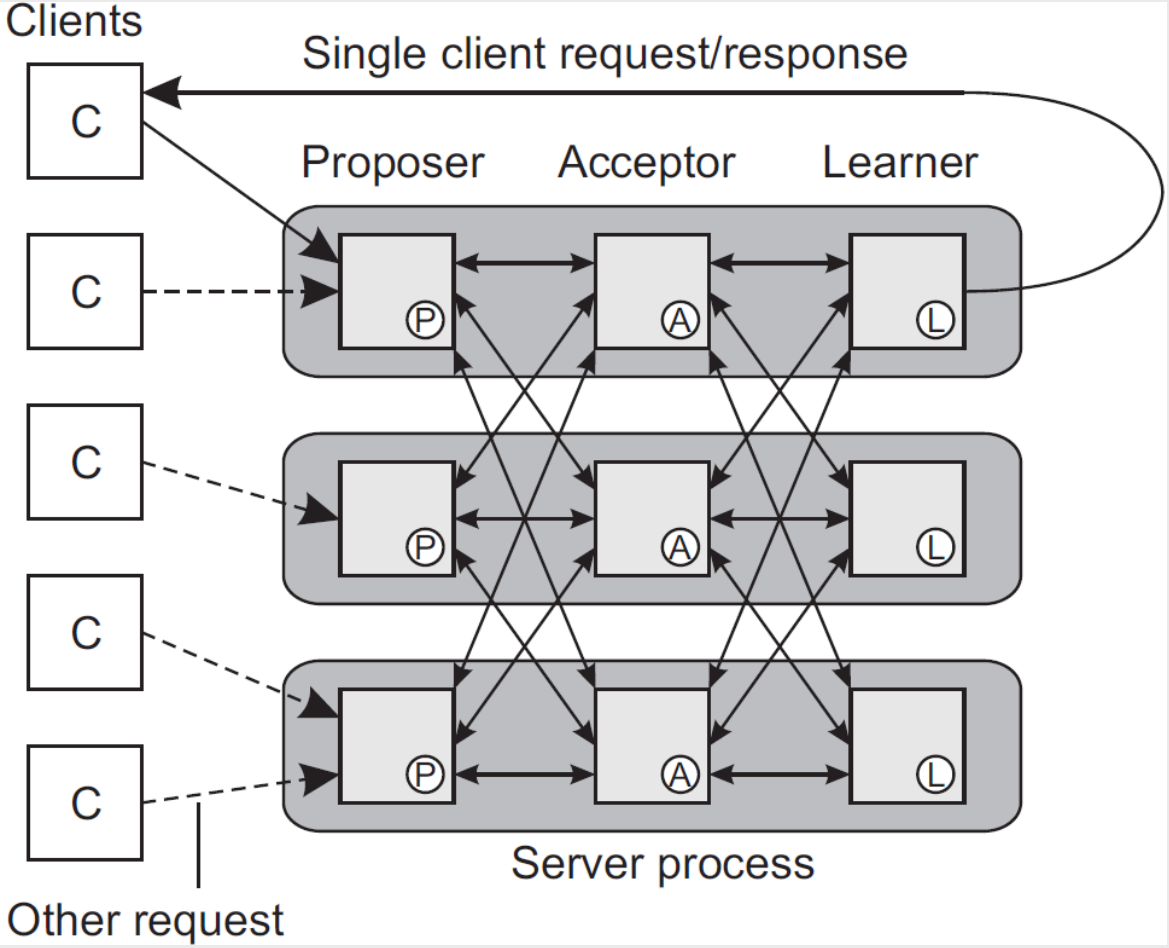
Reaching consensus with crash failure



Flooding-based Consensus

Consensus: everyone do the same things, in the same order
P2 broadcast its decision in the second round

Three logical Processes of Paxos



A two step approach

Need a consensus algorithm/system

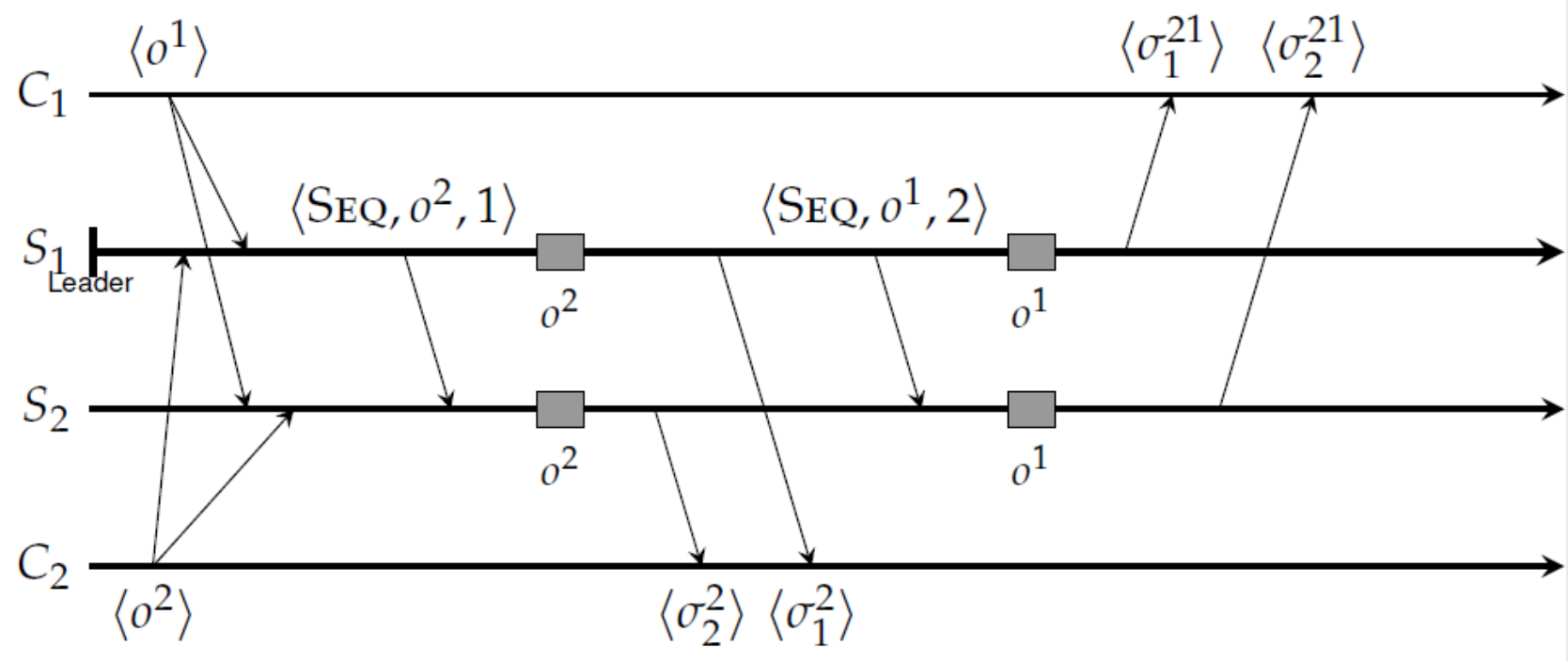
Paxos

Using a leading proposer to drive the acceptors toward the execution of the same operation

- Phase 1: taking which proposal first
  - Phase 1a (prepare)
  - Phase 1b (promise)
- Phase 2 execute which proposal now
  - Phase 2a (accept)
  - Phase 2b (learn)

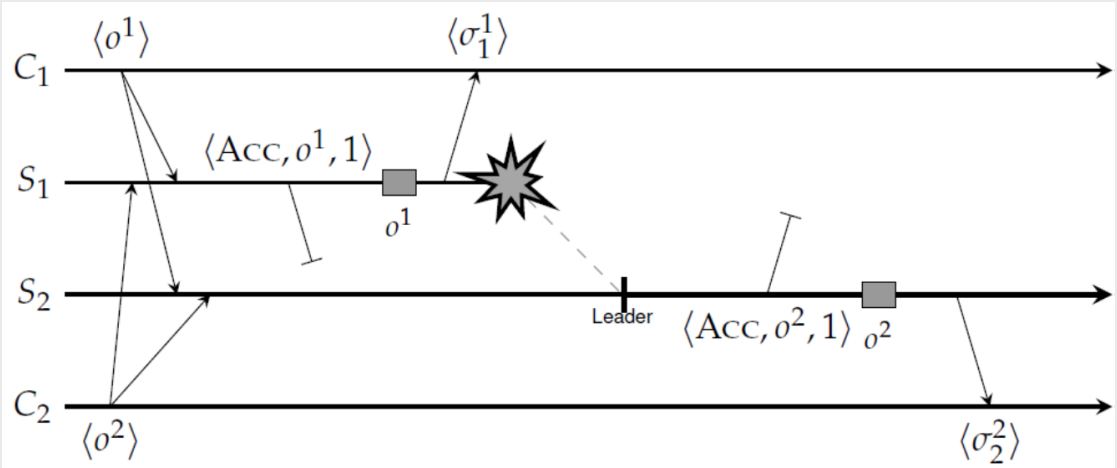
Understanding Paxos

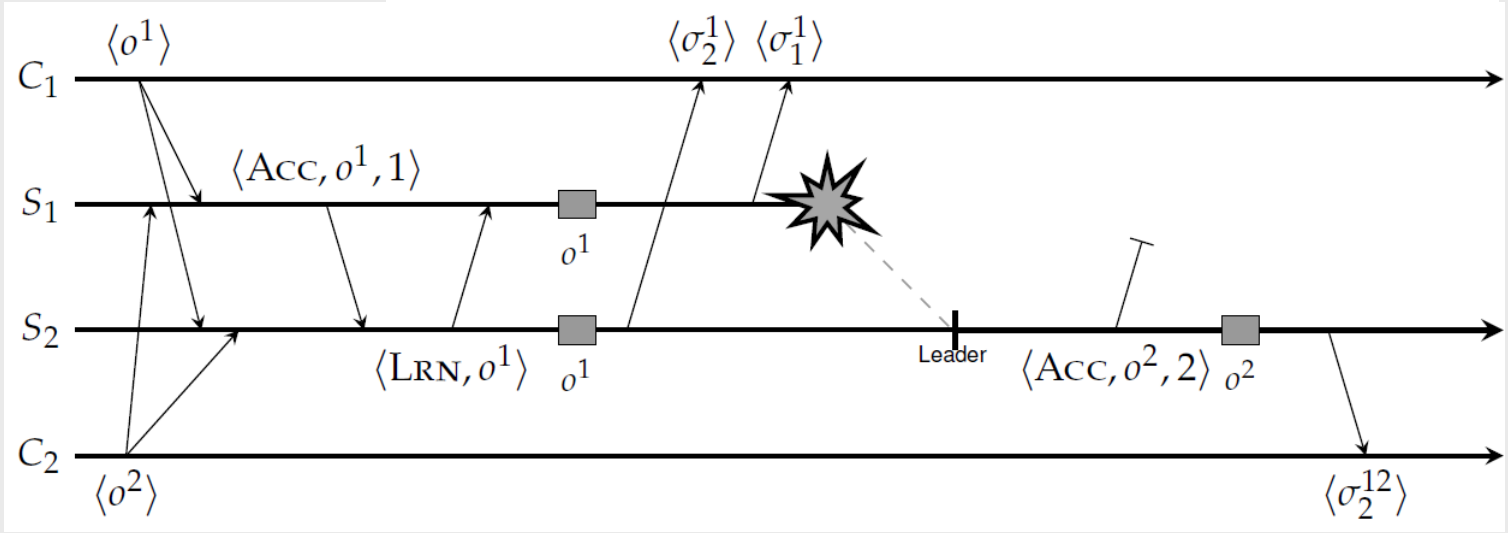
Leader gives the order: with no fault it is easy



Two Clients Communicate with a 2-server process group

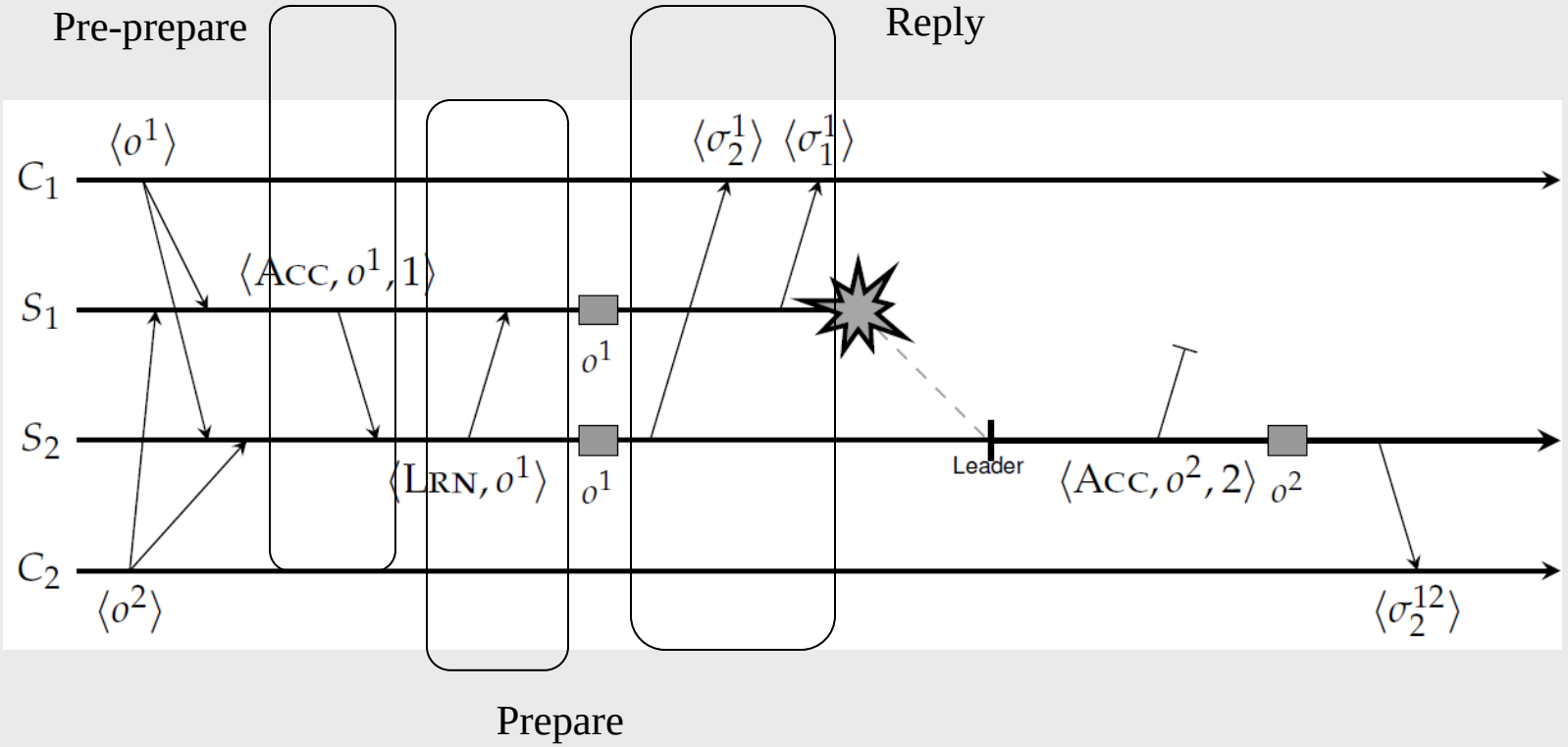
When the Leader Crash: Drop the faulty server then Commit





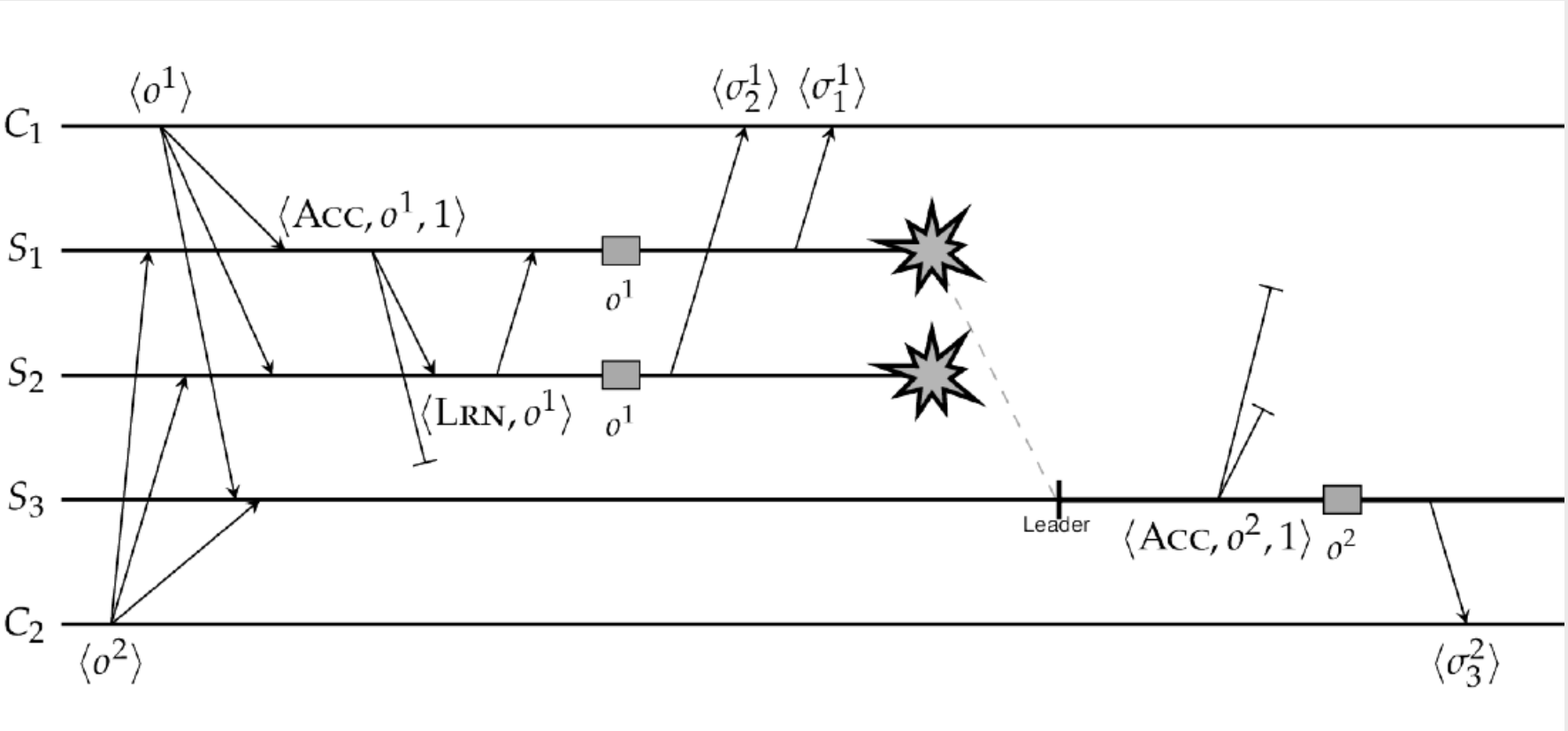
In *Paxos*, a server *S* cannot execute an operation *o* until it has received a *Learn(o)* from all the other nonfaulty servers

Synchronization: Commitment



The two step procedure is correct, and the result looks correct, but in the second part S2 works alone, with no majority vote, could be problematic.

Paxos requires at least three replicated service for correctness



More Complex Situation: with three Servers

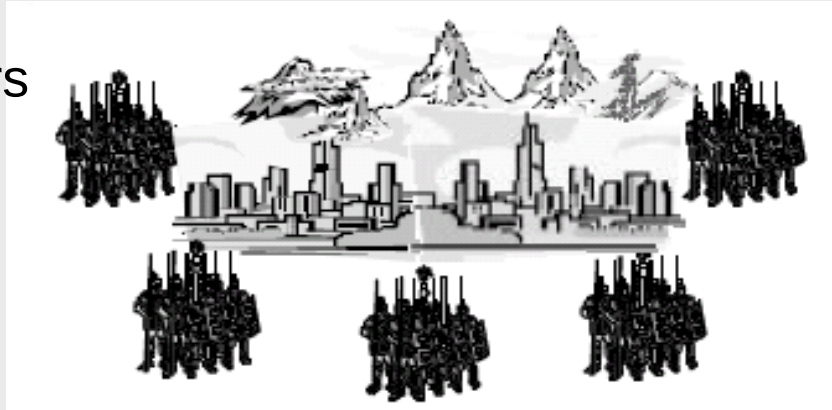
Fig. 8.9 a server S cannot execute until it received a LEARN(0) from all nonfault servers

Agreement in Faulty Systems

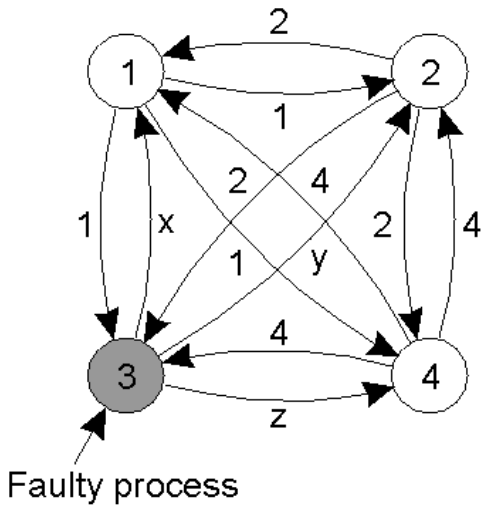
- How should processes agree on results of a computation?
- *K-fault tolerant*: *?*
  - Survive *k* components fail silently, with k+1 replica
  - Survive *k* components faults, with 2k+1 replica
- *Byzantine failures*: *?*
  - Arbitrary failures, intentionally provide wrong information, where the final results need global information, 3k+1 replica

Byzantine Generals Problem

- N divisions of Byzantine army surround city
  - Each division commanded by a general
  - Some of the N generals are traitors
- Generals communicate via messages
  - Traitors can send different values to different generals
- Requirements:
  - All loyal generals decide upon same plan of action
  - A “small” number of traitors cannot cause loyal generals to adopt a bad plan
  - NOT required to identify traitors



Byzantine Generals Problem



(a)

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

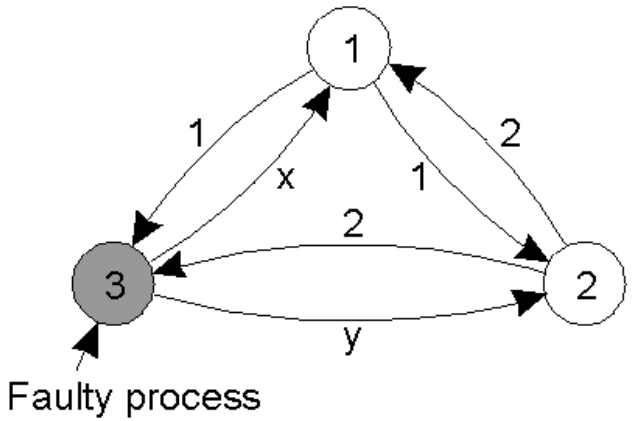
(b)

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

(c)

Two broadcast, traitor general 3

Byzantine Generals Problem Example



(a)

1 Got(1, 2, x)
2 Got(1, 2, y)
3 Got(1, 2, 3)

(b)

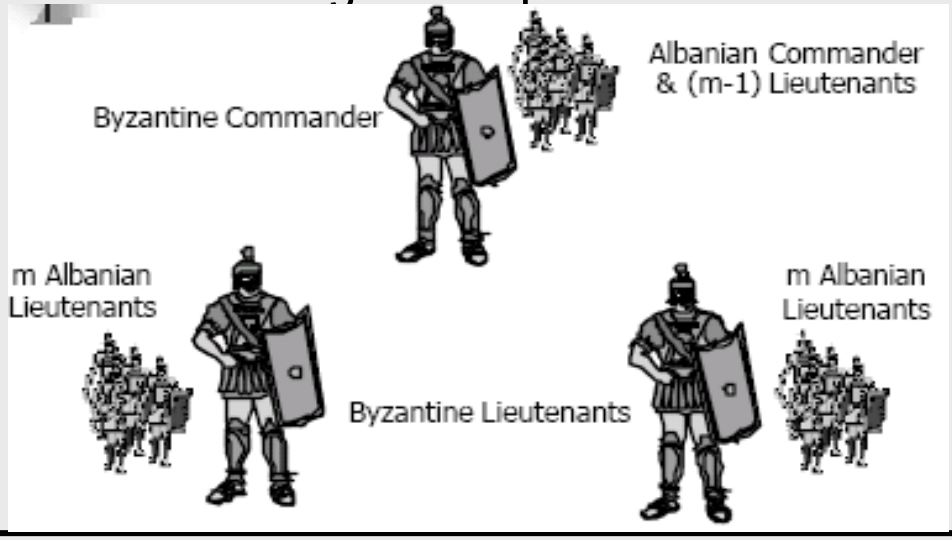
1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

(c)

- The same as in previous slide, except now with 2 loyal generals and one traitor.
- Conclusion: ?
- Theoretical result: a system with m faulty component needs 2m+1 correct components to reach agreement, 3m+1

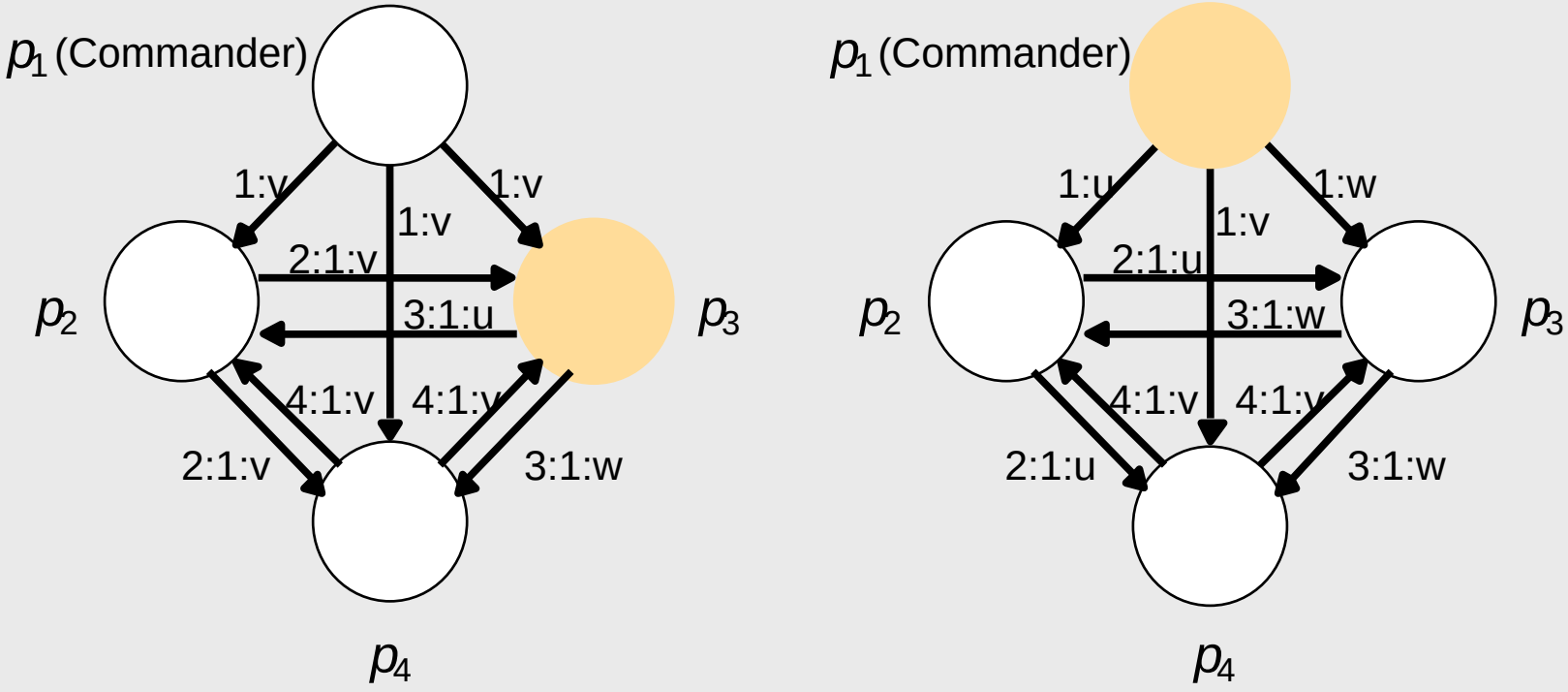
General Impossibility Result

- In general, no solutions with fewer than 3m+1 generals if there are m traitors
- Proof by contradiction:
  - Assume there is a solution for 3m Albanians, including m traitors
  - Let a Byzantine general simulate m Albanian generals
  - The problem is then reduced to 3-general problem



Solution Example

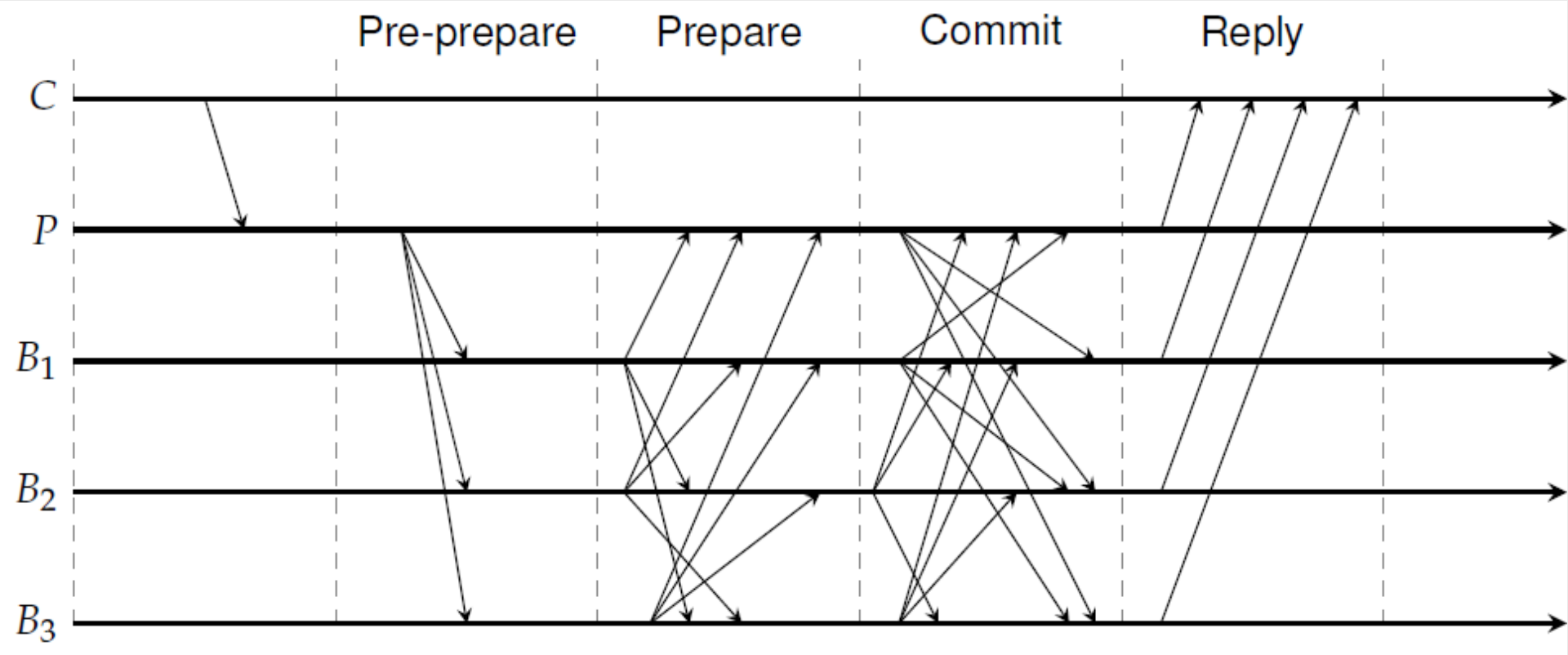
- With one faulty process: f=1, N=4
- 1st round: the cmd sends a value to each Lt
- 2nd round: each Lt copies the value to all other Lts



Faulty processes are shown coloured

Practical Byzantine Fault Tolerance

- 2K+1 majority, 3K+1 vote
- It is good, as long as P does not fail

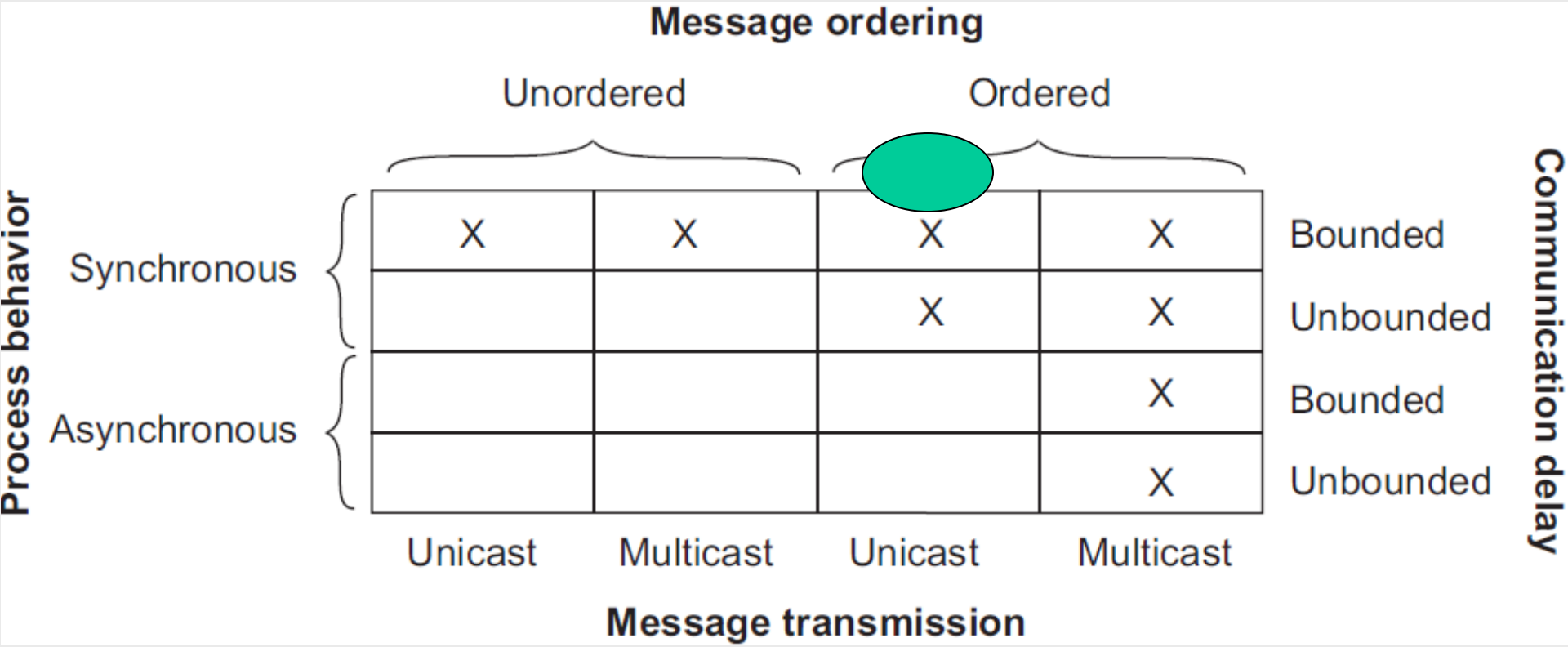


On Reaching Consensus

- Distributed consensus
- Synchronous and asynchronous system
- Communication delay is bounded or not
- Message delivery is order or not
- Unicast of multicast
- Possible have solution:
  - X

Circumstances & Distributed Consensus Possible

- Cost is application/circumstance dependent
- When fault may occur, it may not be able to reach consensus at all



Consistency, Availability, & Partitioning

CAP Theorem: Any networked system providing shared data can provide only two of the following three properties:

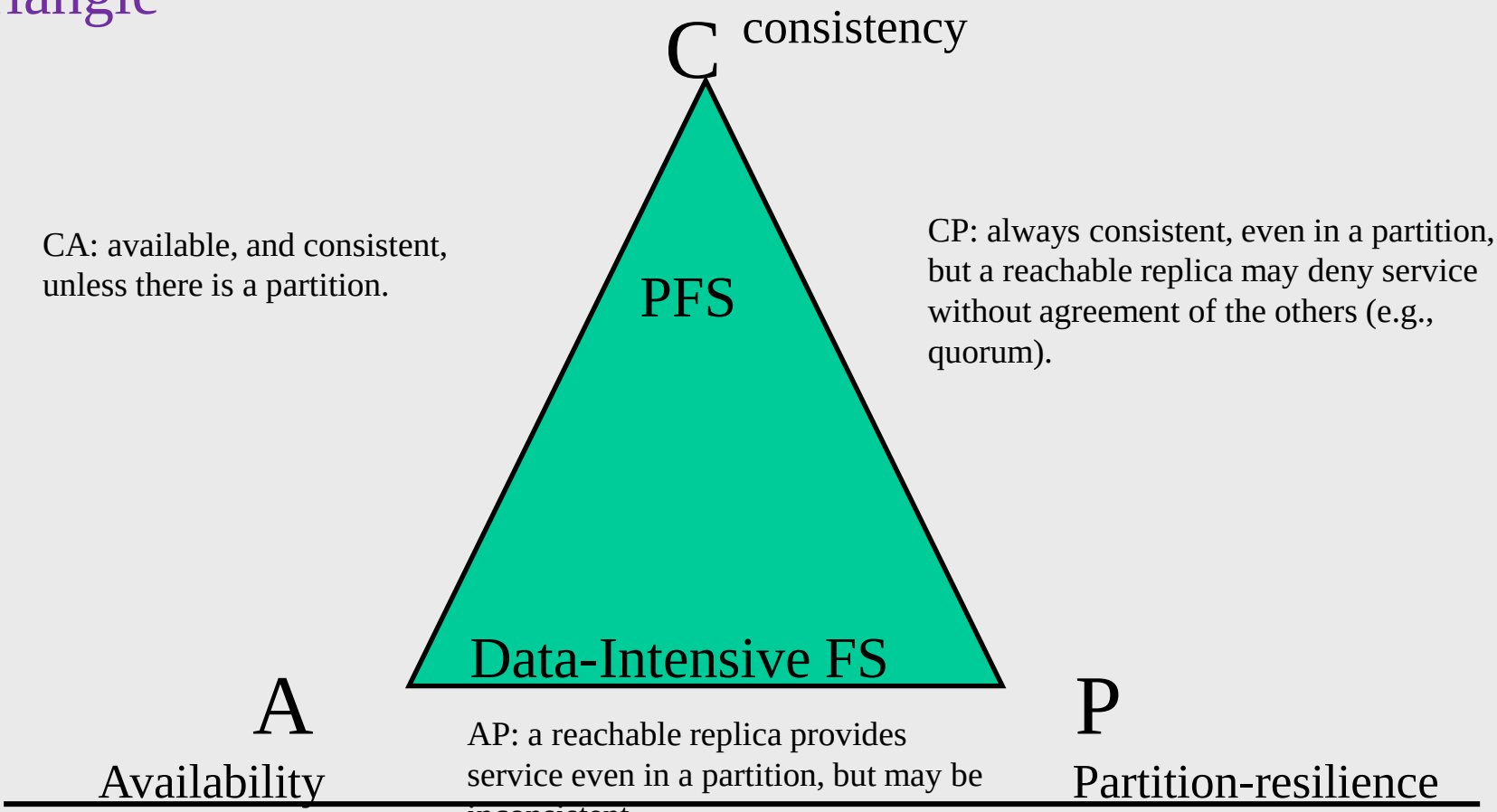
- **C**: consistency, by which a shared and replicated data item appears as a single, up-to-date copy
- **A**: availability, by which updates will always be eventually executed
- **P**: Tolerant to the partitioning of process group (e.g. because of a failing network)

• In other words, in a network subject to communication failures, it is impossible to realize an atomic read/write shared memory that guarantees a response to every request

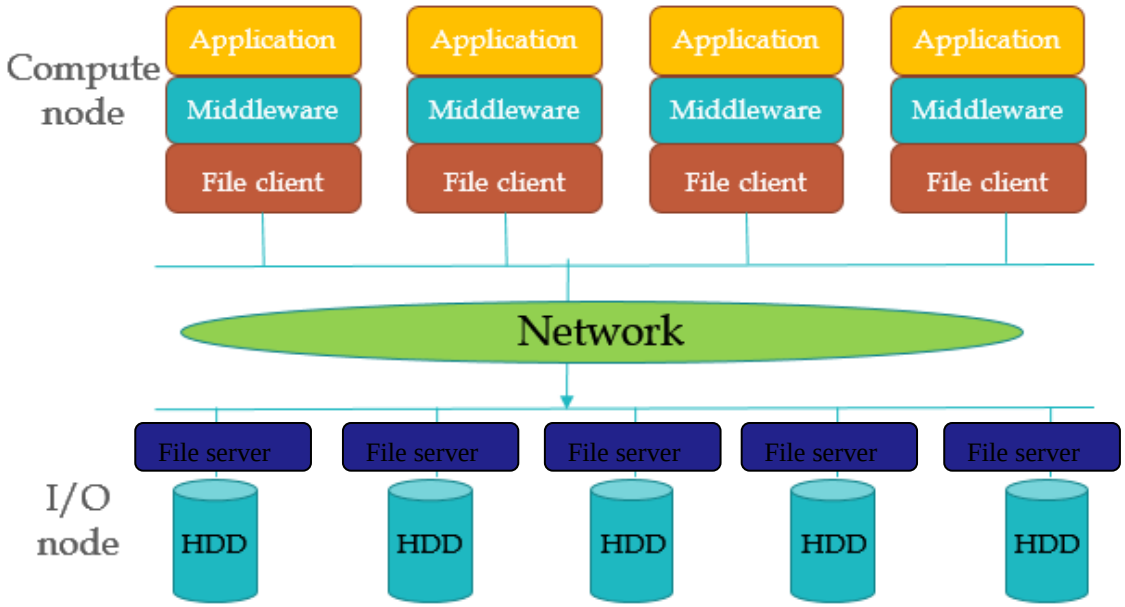
Eric Brewer 2000, Gilbert and Lynch 2002

The CAP Theory: only Two/Three

Claim: Every distributed system is on one side of the triangle

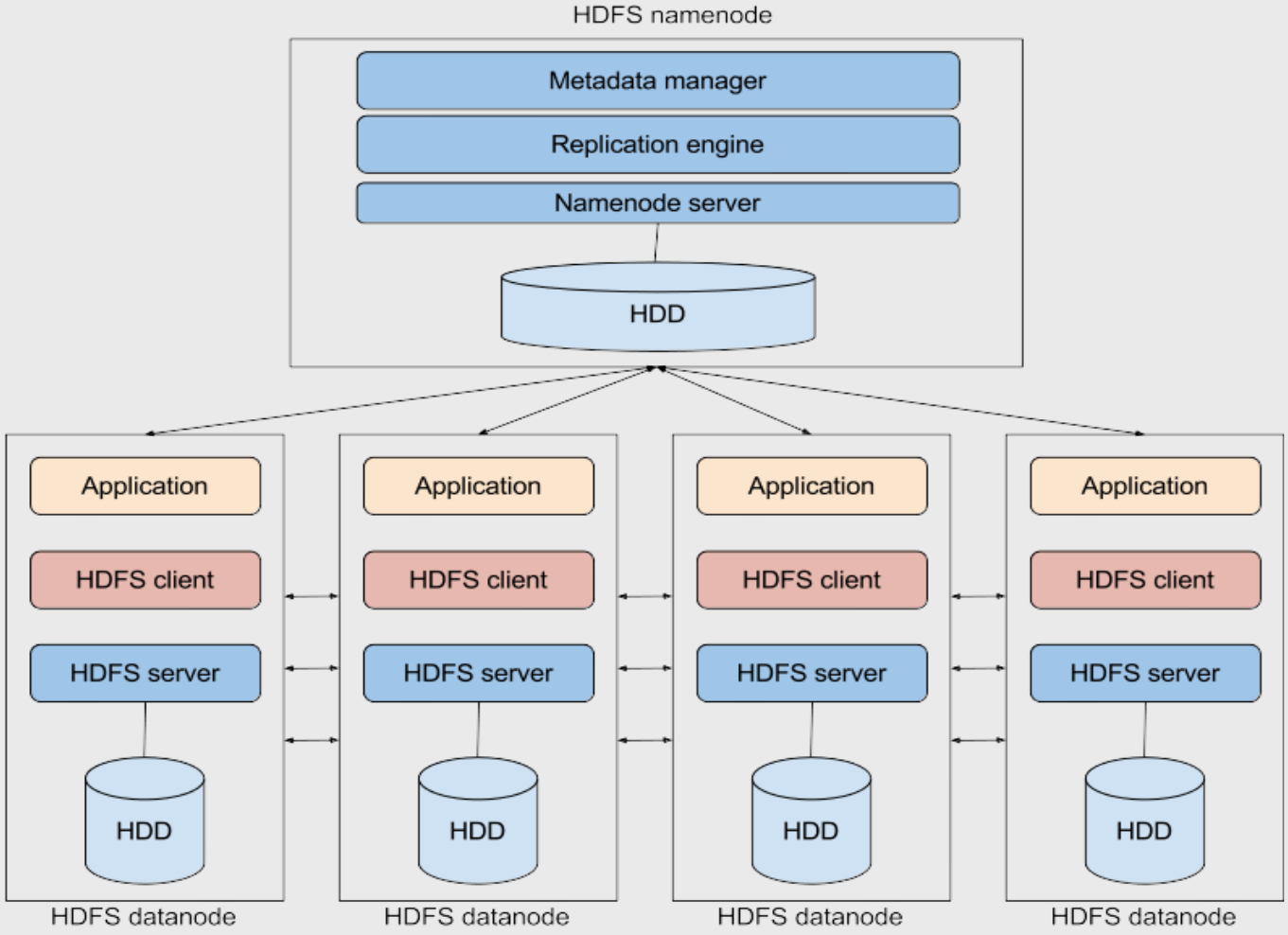


Traditional Solution： Parallel I/O Systems



Architecture of a typical parallel I/O system

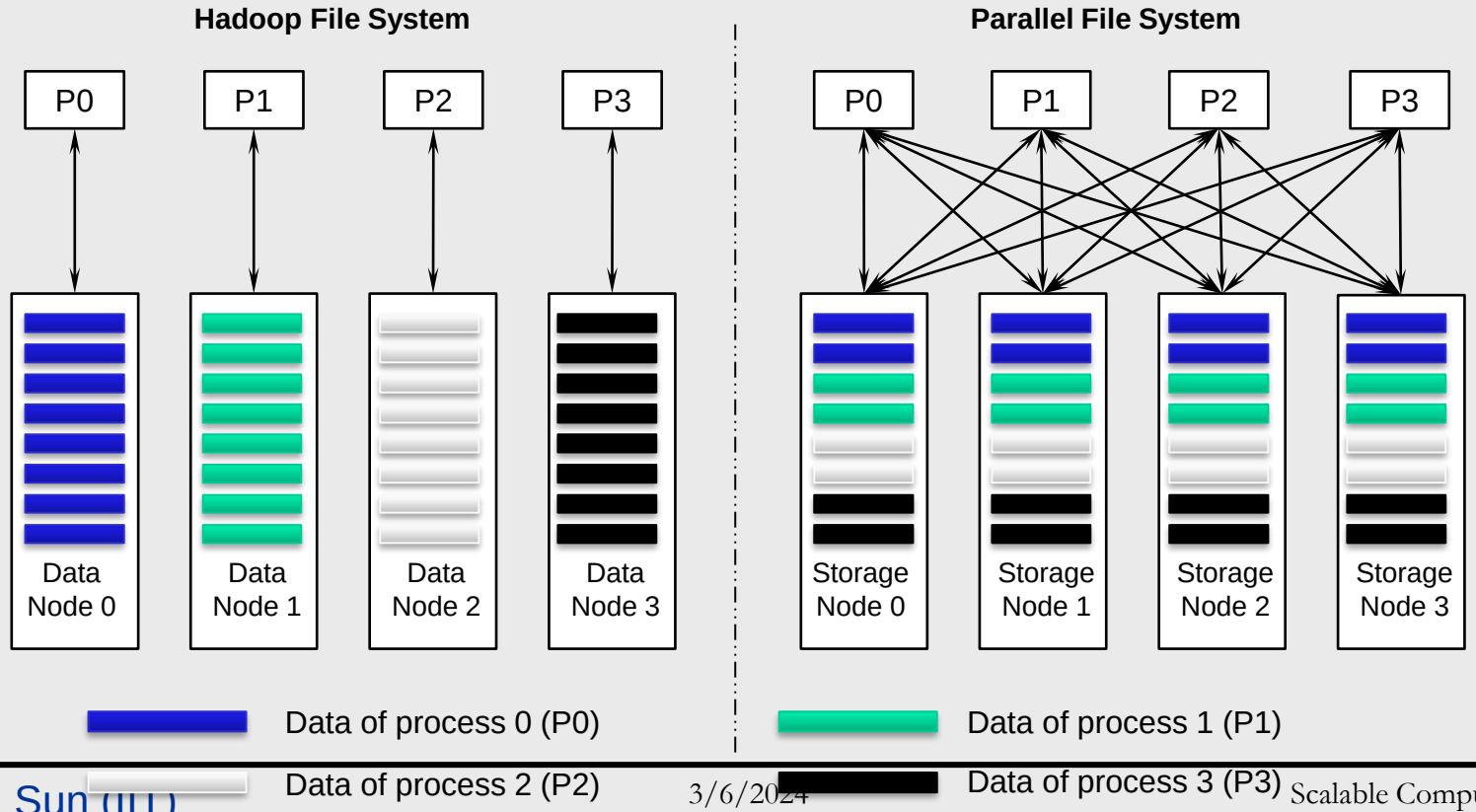
Big Data Industrial Solution:
Distributed I/O Systems



Architecture of a typical HDFS system

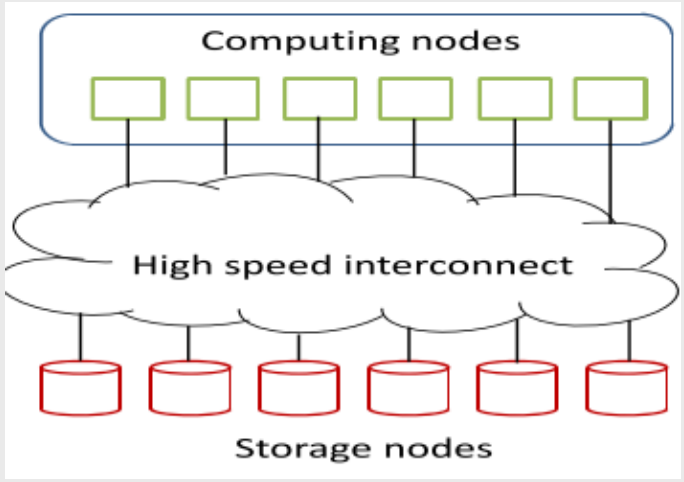
Two Camps of High Performance File Systems

- Google Hadoop File System (HFS): Perfect parallel
- IBM Parallel File System (PFS): Consistent
- The two do not talk to each other



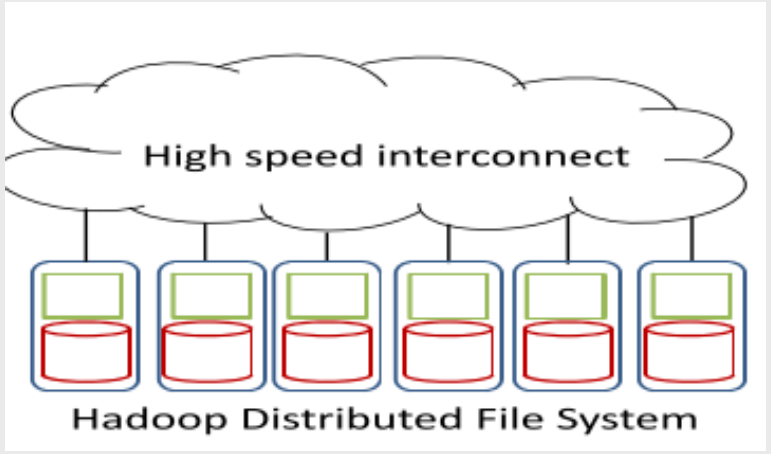
Integrate Two Camps.

- High-Performance Computing



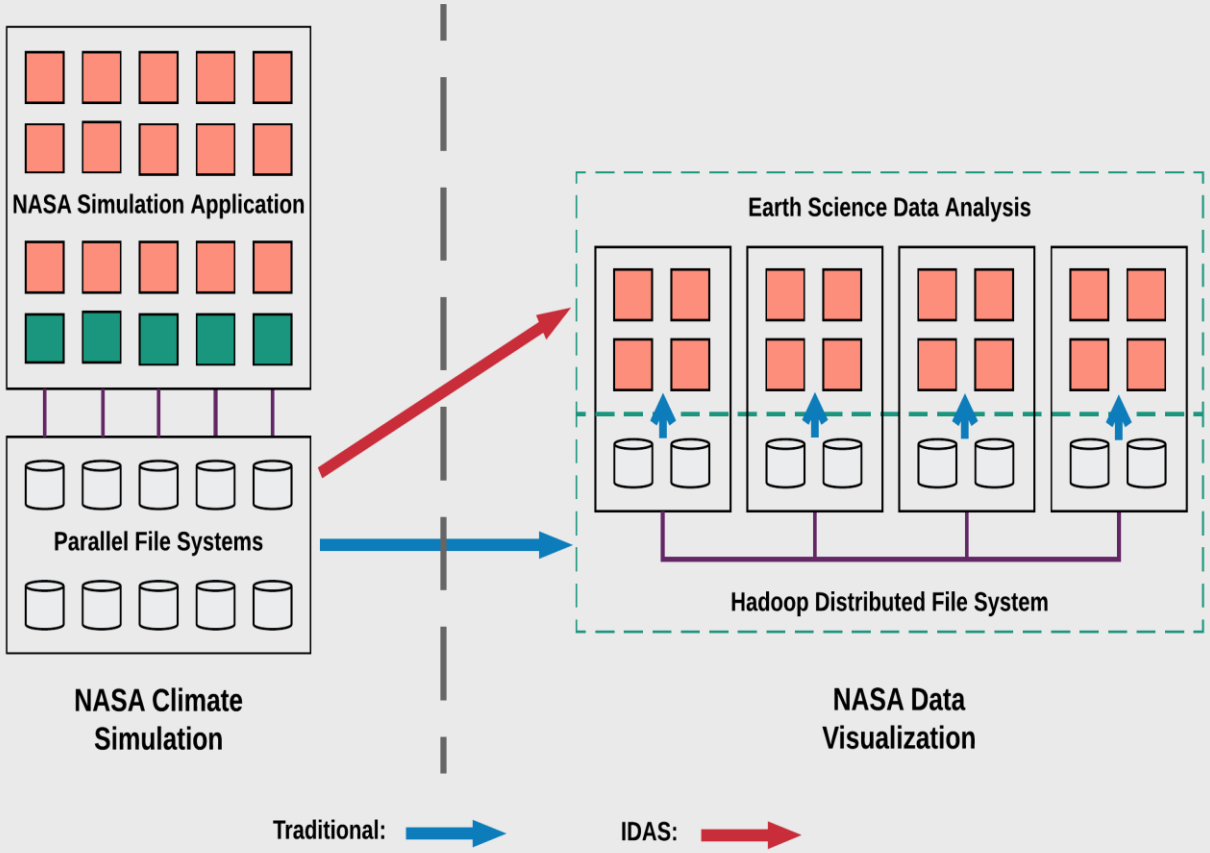
A typical HPC Cluster

- Cloud Computing (MapReduce)



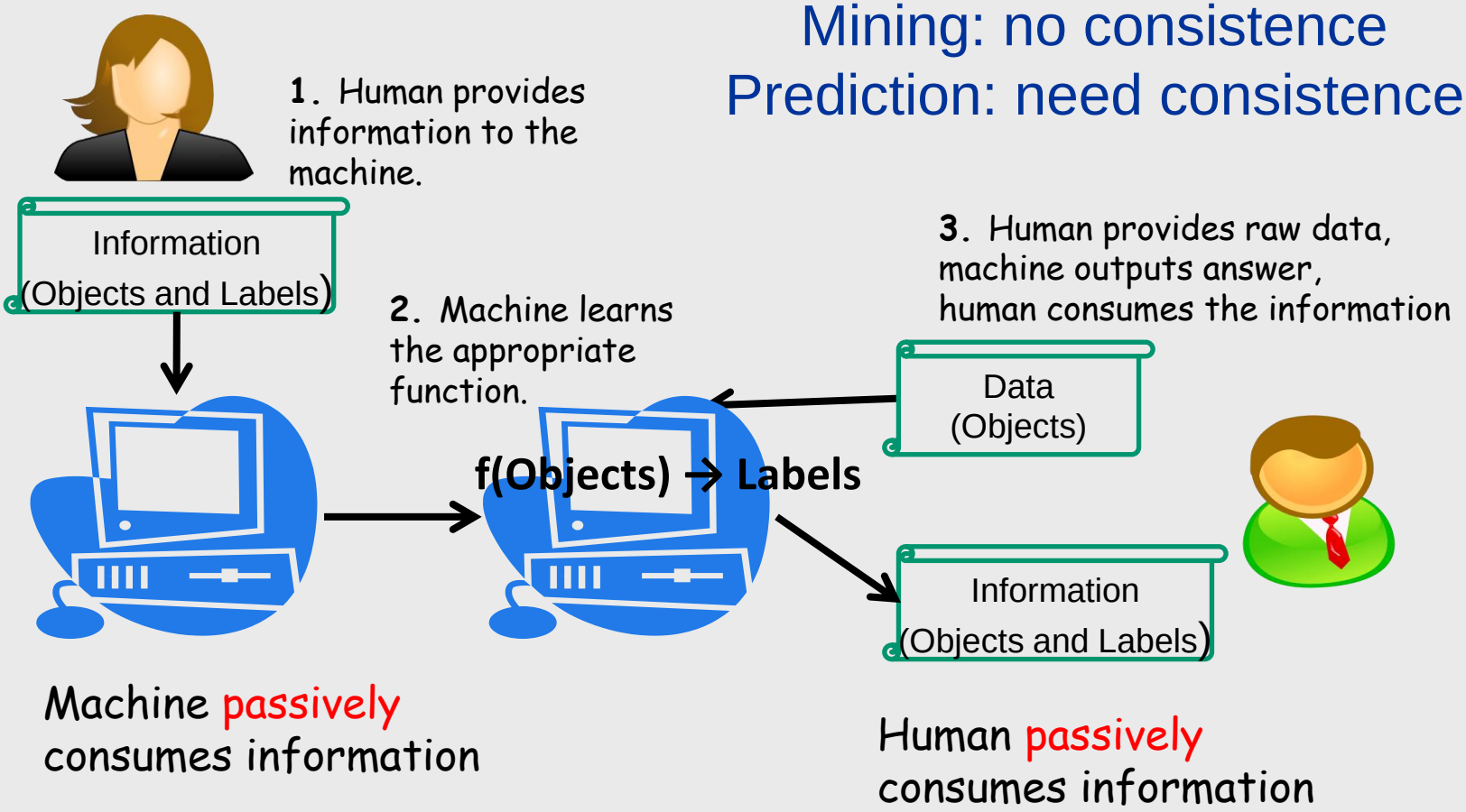
A Hadoop Cluster

Application -- NASA Earth Science Data Analysis (Super-Cloud Project)

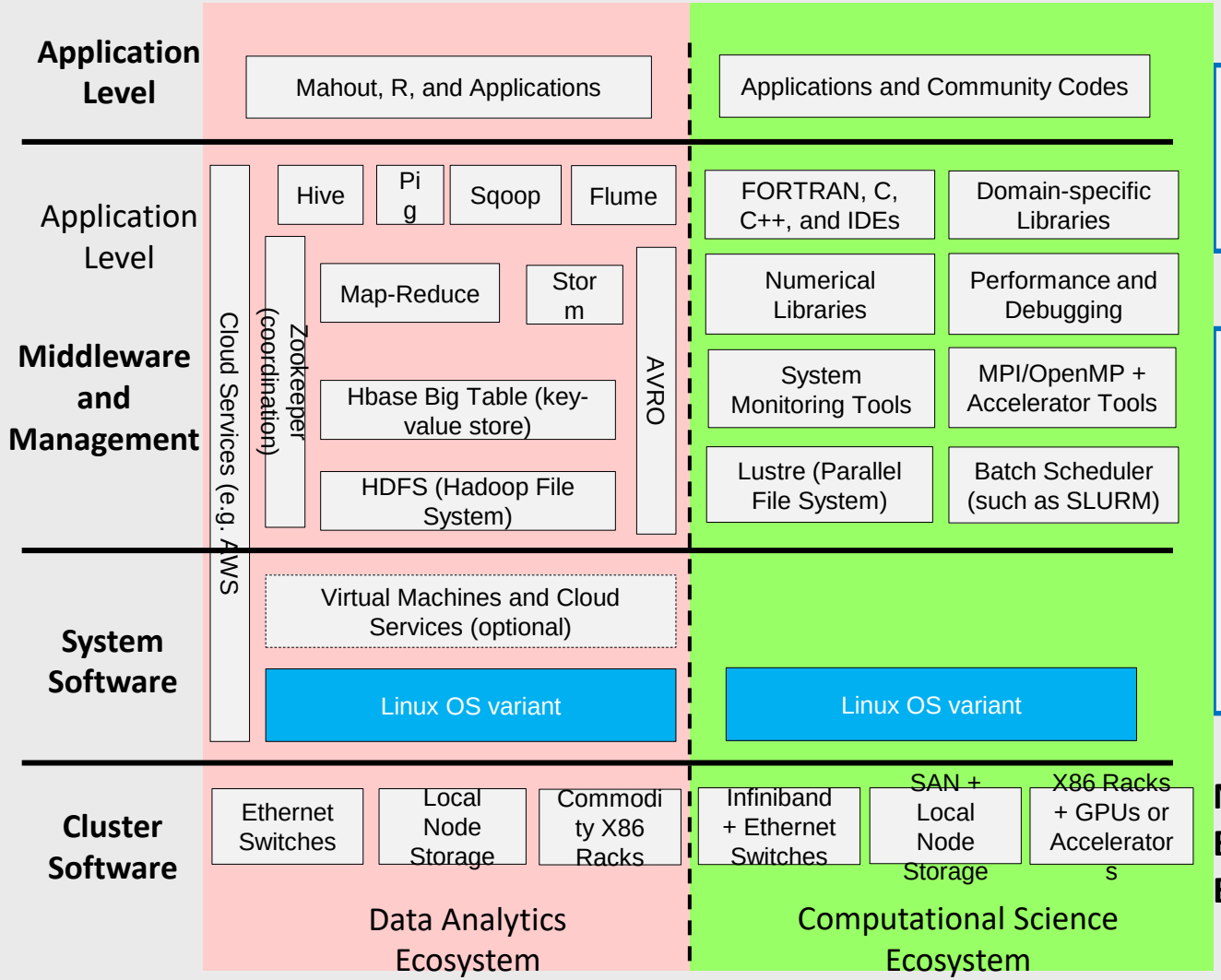


Simulation is done on MPI environment and Data Analysis is done on Hadoop environment

Big Data often is mixed with consistence and non-consistence requirement



The Data & Computing Ecosystems



Computational Science

FORTRAN,C,C++: languages
PAPI: performance and debugging tool
MPI/OpenMP: multi-core parallel model
SLURM: batch scheduler
Lustre: parallel file system

Data Analytic

Mahout: machine learning tool
Hive: data warehouse software
Pig: provide high level language for big data
Sqoop: exchange data with traditional database
Flume: log management
Zookeeper: maintaining consistency
Storm: real-time computation system.
Hbase: a distributed, scalable big data store.
AVRO: data serialization system.

NOTE: The Divergence of Big Data and HPC Eco-Systems!

Courtesy by "Exscale Computing and Big Data", DANIEL A. REED AND JACK DONGARRA , CACM July 2015

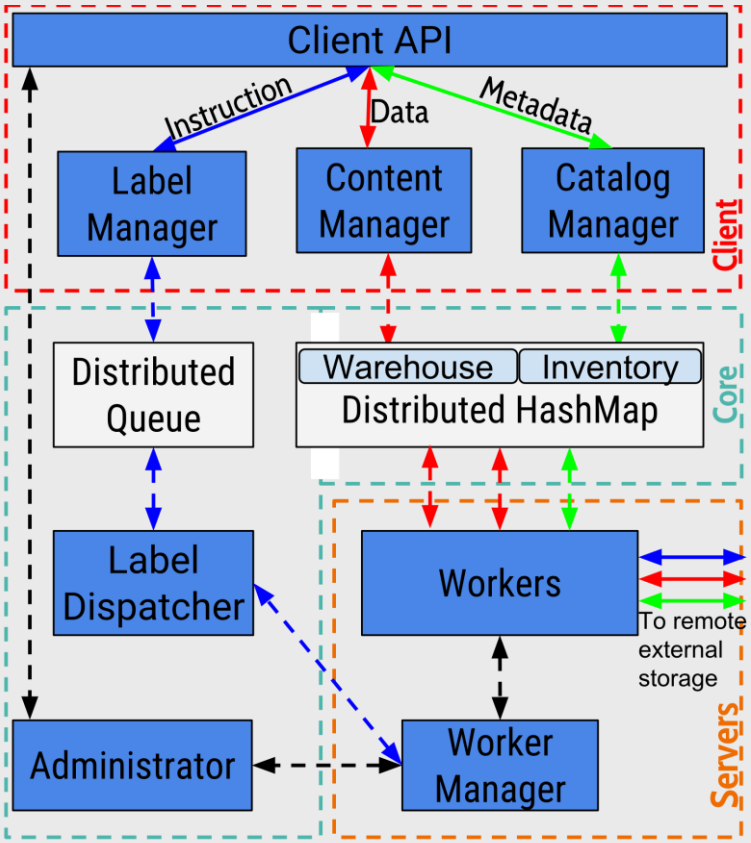
Recent Results in I/O

- **Hermes**: a heterogeneous-aware multi-tiered distributed I/O buffering system, *Proceedings of the 27th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2018, Kougkas, Anthony and Devarajan, Hariharan and Sun, Xian-He

- **LABIOS**: A Distributed Label-Based I/O System, *the 28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC'19)*, June 2019, A. Kougkas, H. Devarajan, J. Lofstead, X.-H. Sun

LABIOS: Label-Based I/O System

- Two main ideas:
  1. Split the data, metadata, and instruction paths.
  2. Decouple storage servers from the application.



High-level Architecture