

---

# Fault Tolerance and Resilience, Raft, Cyber 3

26 January 2025  
Lecture 12

Slide Credits: Maarten van Steen

# Topics for Today

---

- Failures and Resilience
- Raft and Consensus
- Cyber 3: Consensus and Failure
  
- Source: TvS 8

# Dependability

---

- A **component** provides **services** to **clients**. To provide services, the component may require the services from other components → a component may depend on some other component.
- A component  $C$  depends on  $C^*$  if the **correctness** of  $C$ 's behavior depends on the correctness of  $C^*$ 's behavior.
- **Note**: in the context of distributed systems, components are generally **processes** or **channels**.

<b>Availability</b>	Readiness for usage
<b>Reliability</b>	Continuity of service delivery
<b>Safety</b>	Very low probability of catastrophes
<b>Maintainability</b>	How easily can a failed system be repaired

# Reliability versus Availability

- **Reliability  $R(t)$** : probability that a component has been **up and running continuously** in the time interval  $[0, t)$ :
- **Traditional metrics**:

Mean Time to Failure  
( $MTTF$ ): Average time  
until a component fails

Mean Time to Repair  
( $MTTR$ ): Average time it  
takes to repair a failed  
component.

Mean Time Between Failures  
( $MTBF$ ):  $MTTF + MTTR$

# Reliability versus Availability

---

**Availability  $A(t)$ :** Average fraction of time that a component has been up and running in the interval  $[0, t)$ :

- (Long term) availability  $A$ :  $A(\infty)$

**Note:**

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$$

---

**Observation**

Reliability and availability make sense only if we have an accurate notion of what a failure is

# Terminology

---

Term	Description	Example
Failure	May occur when a component is not living up to its specification	A crashed program
Error	Part of a component that may lead to a failure	A programming bug
Fault	The cause of an error	A sloppy programmer

# Handling Faults

Term	Description	Example
Fault Prevention	Prevent the occurrence of a fault	Don't hire sloppy programmers
Fault Tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault Removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault Forecasting	Estimate current presence, future incidence, and consequences of bugs	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

# Handling Faults

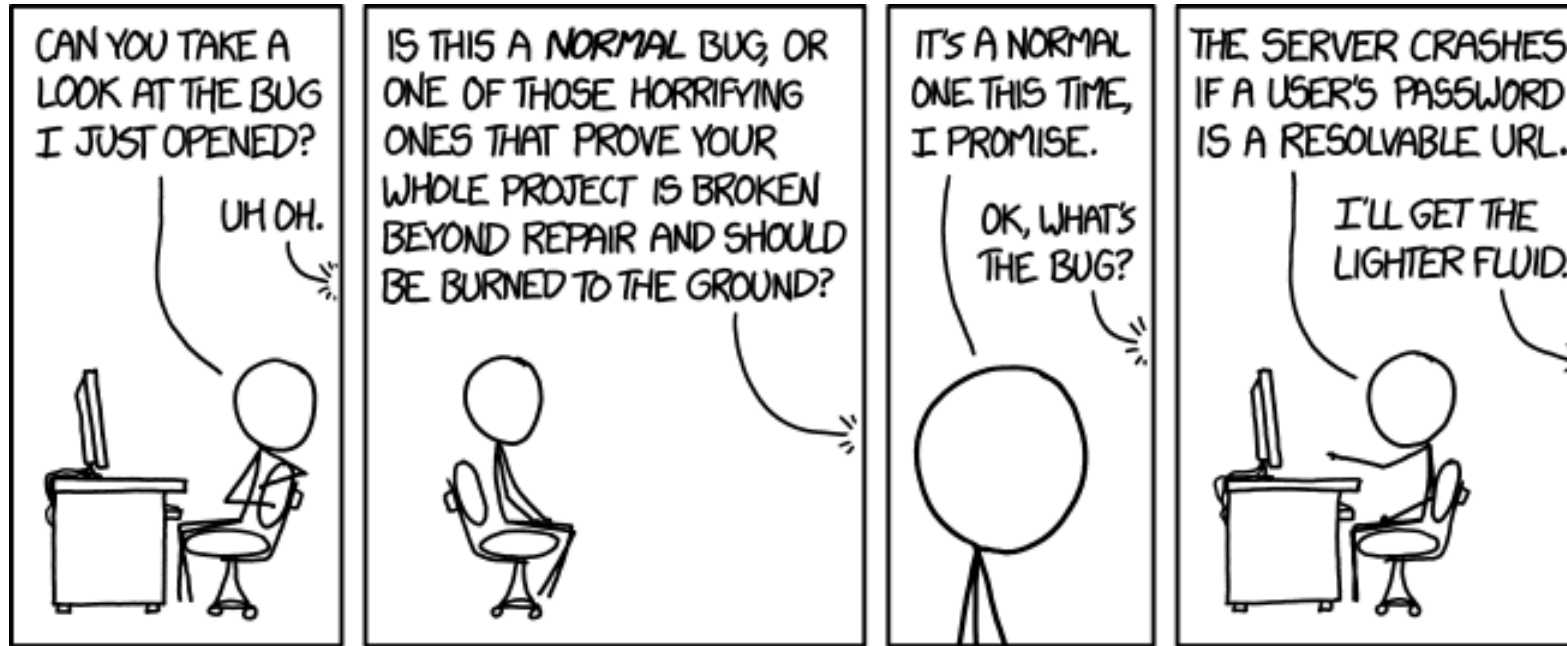


Image: <https://xkcd.com/1700/>



# Failure models

---

## Crash failures:

- Halt, but correct behavior until halting

## General omission failures:

- Failure in sending or receiving messages
- Receiving omissions: Sent messages are not received
- Send omissions: Messages are not sent that should have

## Timing failures:

- Correct output but provided outside a specified time interval.

# Failure models

---

## Response failures:

- Response is incorrect
- **Value failures**: Wong output values
- **State transition failures**: Deviation from correct flow of control

## Arbitrary failures:

- May produce arbitrary responses at arbitrary times

# Dependability versus Security

---

- **Omission failure:** A component fails to take an action that it should have taken
- **Commission failure:** A component takes an action that it should not have taken

## Observations

---

Deliberate failures, be they omission or commission failures, stretch out to the field of security

There is a thin line between dependability and security

# Halting Failures

---

**Scenario:**  $\mathcal{C}$  no longer perceives any activity from  $\mathcal{C}^*$   $\rightarrow$  a halting failure? Distinguishing between a crash or omission/timing failure may be impossible:

## Asynchronous system

- No assumptions about process execution speeds or message delivery times
- $\rightarrow$  cannot reliably detect crash failures.

## Synchronous system

- Process execution speeds and message delivery times are bounded
- $\rightarrow$  we can reliably detect omission and timing failures.

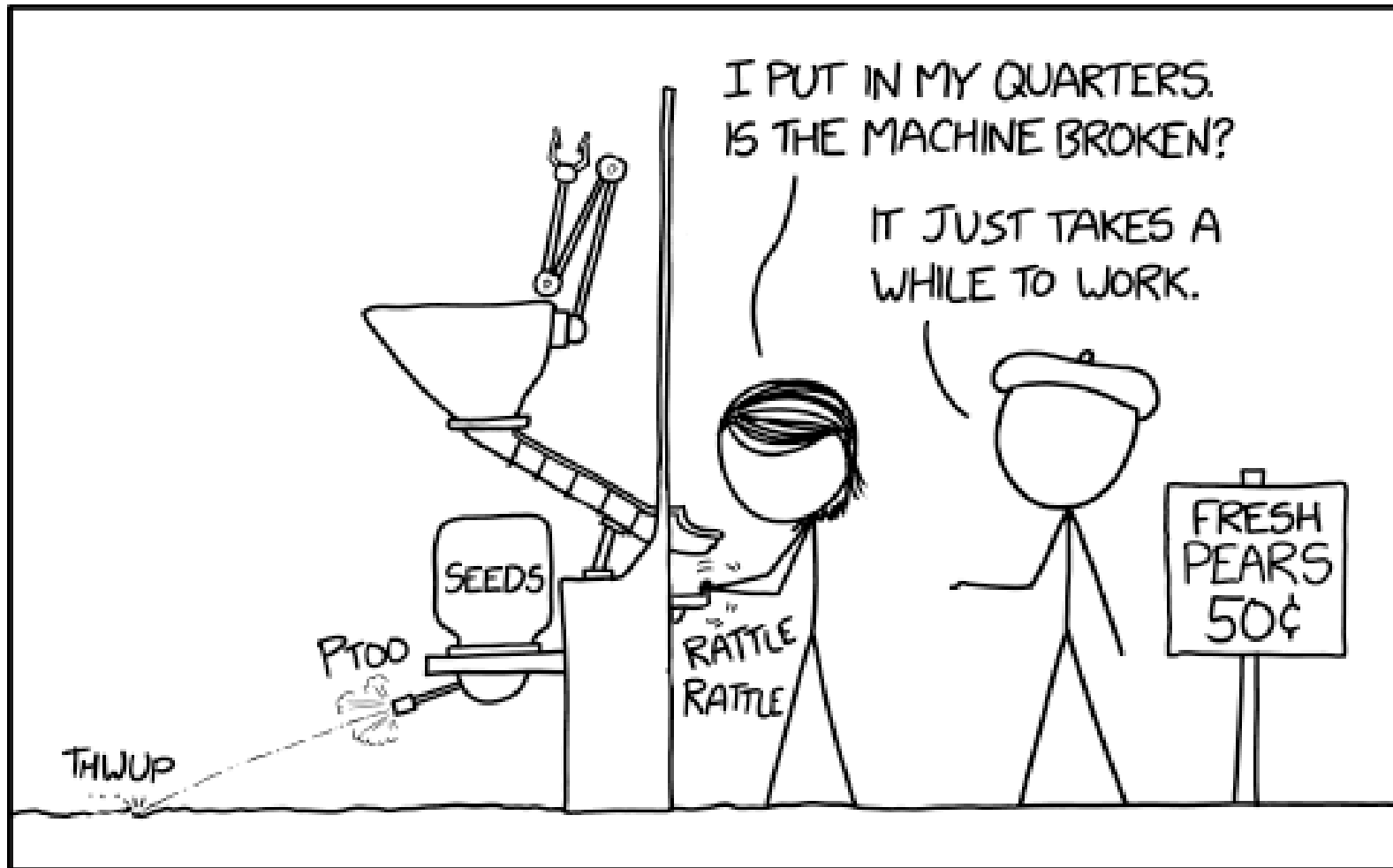
# Halting Failures

---

**Scenario:**  $\mathcal{C}$  no longer perceives any activity from  $\mathcal{C}^*$   $\rightarrow$  a halting failure? Distinguishing between a crash or omission/timing failure may be impossible:

## Partially synchronous systems

- Most of the time, we can assume the system to be synchronous
- Yet there is no bound on the time that a system is asynchronous
- $\rightarrow$  can normally reliably detect crash failures



# Halting Failures Assumptions

---

**Fail-stop:** Crash failures, but reliably detectable



**Fail-noisy:** Crash failures, eventually reliably detectable



**Fail-silent:** Omission or crash failures: clients cannot tell what went wrong.



**Fail-safe:** Arbitrary, yet benign failures (can't do any harm).



**Fail-arbitrary:** Arbitrary, with malicious failures



# Ways to Mask Failure

---

## Information redundancy

- Add extra bits to data units so that errors can be recovered when bits are garbled.



## Time redundancy

- Design a system so actions can be performed again if something is wrong.
- Used when faults are transient or intermittent.



## Physical redundancy

- Add equipment or processes to allow failure of one or more components
- Extensively used in distributed systems.





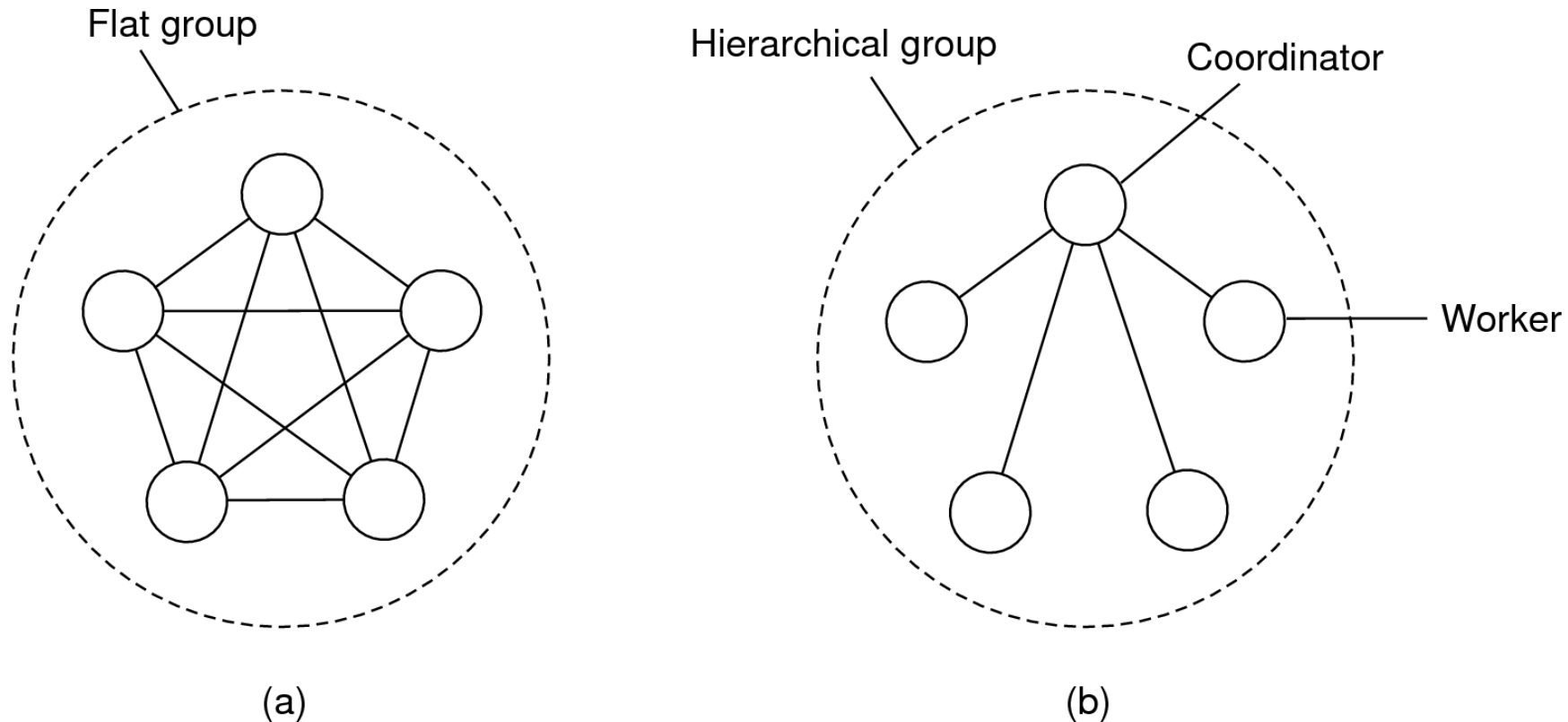
# So Far

---

- Failures and Resilience
- Raft and Consensus
- Cyber 3: Consensus and Failure

# Process Resilience

Basic idea: protect yourself against faulty processes through **process replication**:



# Groups and Failure Masking

---

## $k$ -Fault-tolerant group

- When a group can mask any  $k$  concurrent member failures
- $k$  is **degree of fault tolerance**

## Group size

### Halting failures:

- $k + 1$  members: no member will produce an incorrect result, so one good member suffices

### Arbitrary failures

- $2k + 1$  members: Correct result can be obtained only through a majority vote.

# Groups and Failure Masking

---

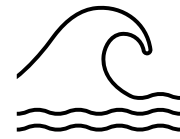
- **Important:**
  - All members are identical
  - All members process commands in the same order
- **Result:**
  - Only then do we know that all processes are programmed to do exactly the same thing

---

## Observation

The processes need to have **consensus** on which command to execute next

# Flood-based Consensus



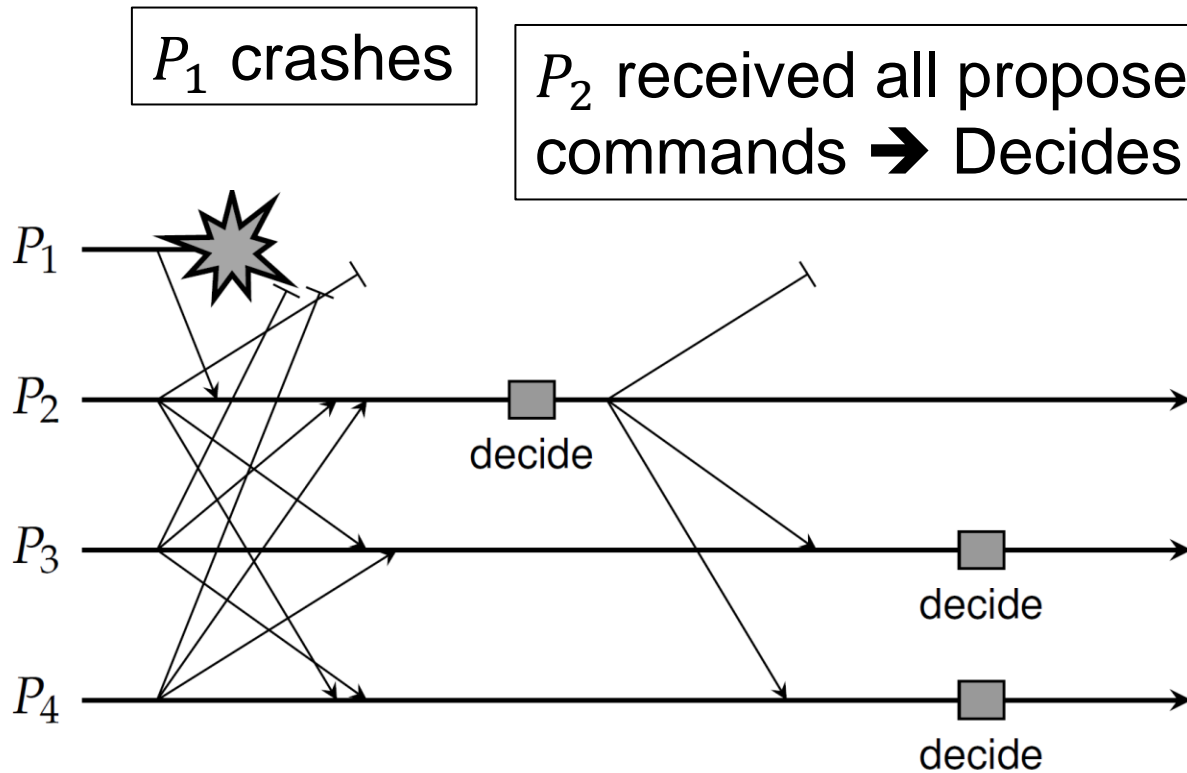
## System Model

- Process group  $P = \{P_1, P_2, \dots, P_n\}$
- **Fail-stop** semantics
- **Reliable failure** detection
- Client contacts  $P_i$  requesting it to execute a command
- Every  $P_i$  maintains a list of proposed commands

## Basic idea (Rounds)

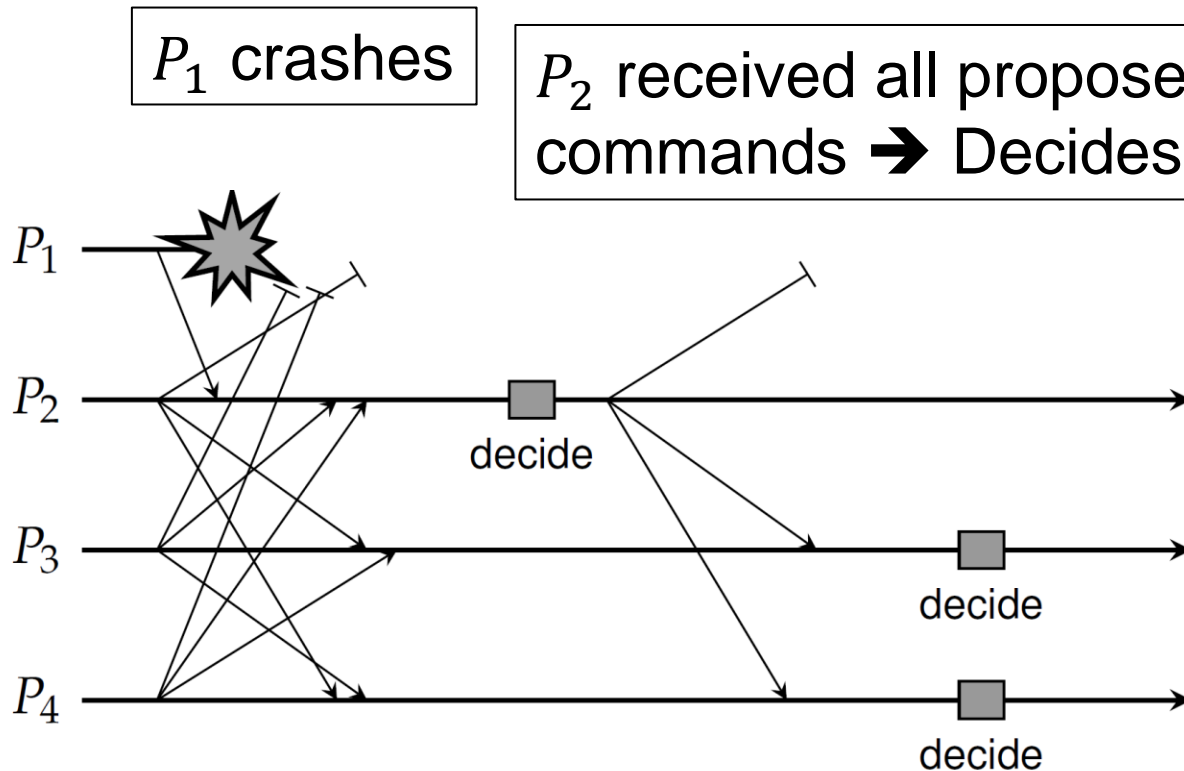
- **Round**  $r$ ,  $P_i$  multicasts its known set of commands  $C_i^r$  to all others
- At the end of  $r$ , each  $P_i$  **merges** all received commands into new  $C_i^{r+1}$
- Next command  $cmd_i$  **selected** through globally shared deterministic function  $cmd_i \leftarrow select(C_i^{r+1})$

# Flood-based Consensus



- $P_3$  may detect  $P_1$  crashed, but does not know if  $P_2$  received anything
- $P_3$  cannot know if it has same information as  $P_2$  → cannot make decision

# Flood-based Consensus



- $P_3$  and  $P_4$  can't decide yet – they know  $P_1$  crashed
- $P_2$  tells the others what it decided
- $P_3$  and  $P_4$  then follows  $P_2$ 's choice

# Raft



Image source and © <https://raft.github.io/>

- Developed for understandability

## Straightforward leader-election algorithm

- Current leader operates during the current term.

## Log

- Every server (5) keeps a log of operations, some of which have been committed.
- Backup will not vote for a new leader if its own log is more up to date.

All committed operations have the same position in the log of every server.

## Leader decides which pending operation is to be committed next

- $\Rightarrow$  a primary-backup approach.





# Leader election in Raft

Relatively small group of servers

## States

- Follower
- Candidate
- Leader

Each server starts in the follower state.

Protocol works in terms, starting with term 0

A leader regularly broadcasts messages

- Updates or
- A simple heartbeat

# Selecting a new Leader



Follower **A** hasn't received anything from leader **L** for some time

**A** broadcasts that it volunteers to be the next leader, increasing the term by 1.

**A** enters the **candidate** state.



# Selecting a new Leader 1

Leader **L** receives the election message  
Responds by acknowledging that it is still the leader.

**A** returns to the **follower** state.

**End**



# Selecting a new Leader 2

Another follower  
**B** gets the  
election message  
from **A**

**B** hasn't heard  
other candidates  
in the current  
round  
**B** votes for **A**.

**A** collects a  
majority of votes  
**A** is leader of new  
term

**End**



# Things could go wrong

## Multiple Candidates, no Majority

- Two candidates started
  - Both received 2 votes
- Election times out, try again
- **Solution:** Slightly differ the timeout values per follower for deciding when to start an election
- Avoids concurrent elections

## Network Partition

- Network divided into parts
- Part with majority elects leader
- Part with minority doesn't elect leader
- **Solution:** Minority part doesn't commit anything
- When network heals, minority part re-syncs



# Things could go wrong

---

## Bad Candidate

- Candidate has less up to date log than recipient
- Might lead to leader who is ignorant
- Solution: Node will not vote for candidate who is less up to date
- Eventually most up to date node will candidate

## Animation Project

- <http://thesecretlivesofdata.com/raft/>



# Submitting an operation

Client  
submits a  
request for  
operation  $o$ .

Leader  
appends the  
request  
 $\langle o, t, k \rangle$  to its  
own log  
Current term  $t$   
 $k$  location in  
leader's log

Log is  
(conceptually)  
broadcast to  
the other  
servers.

Others  
(conceptually)  
copy the log  
Acknowledge  
receipt.

When  
majority of  
acks arrive  
Leader  
commits  $o$

Leader  
responds to  
client



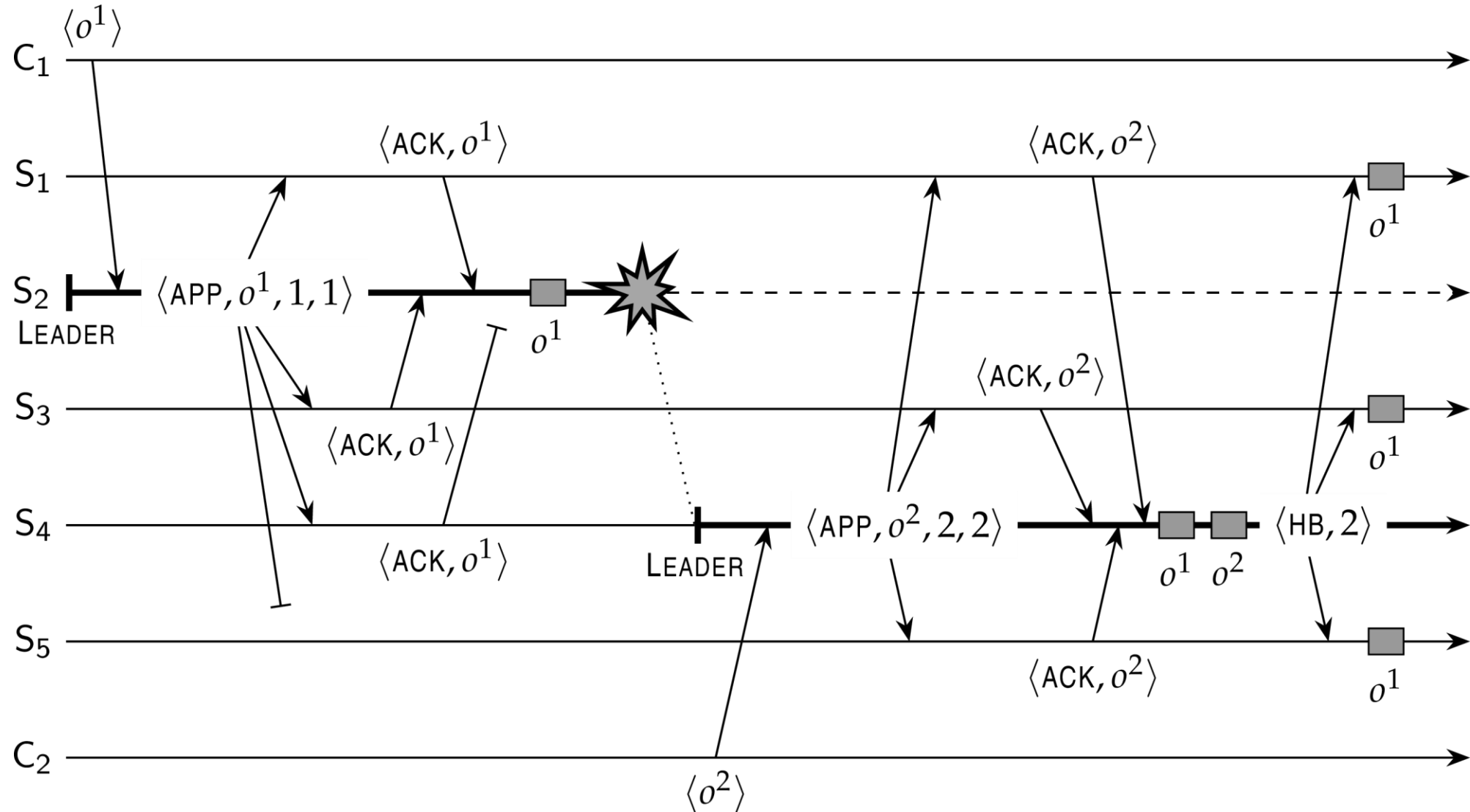
# Log replication in practice

---

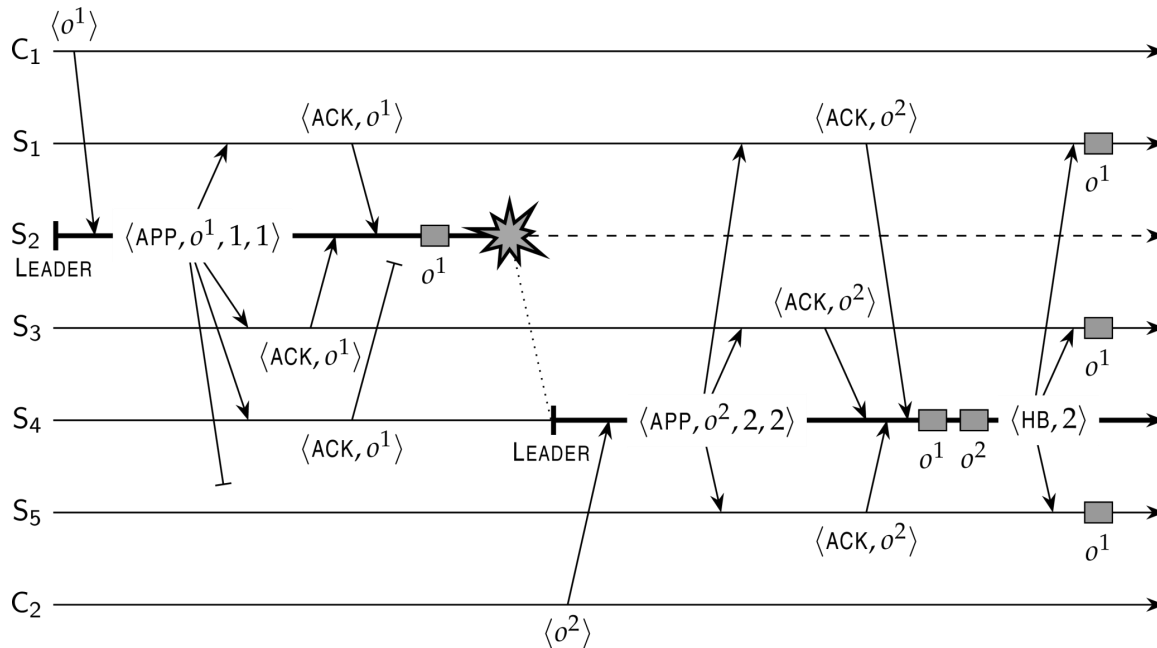
- Only updates are really broadcast
- At the end, every server has the same view and knows about the  $c$  committed operations.
- Effectively, any information at the backups is overwritten.
- If a follower doesn't ack an operation, leader will keep sending until it does
- Follower has pending operations list that can be overwritten if a new leader takes over
- If a new leader has a larger term or more operations in log, followed makes its log adapt



# Raft Crashing

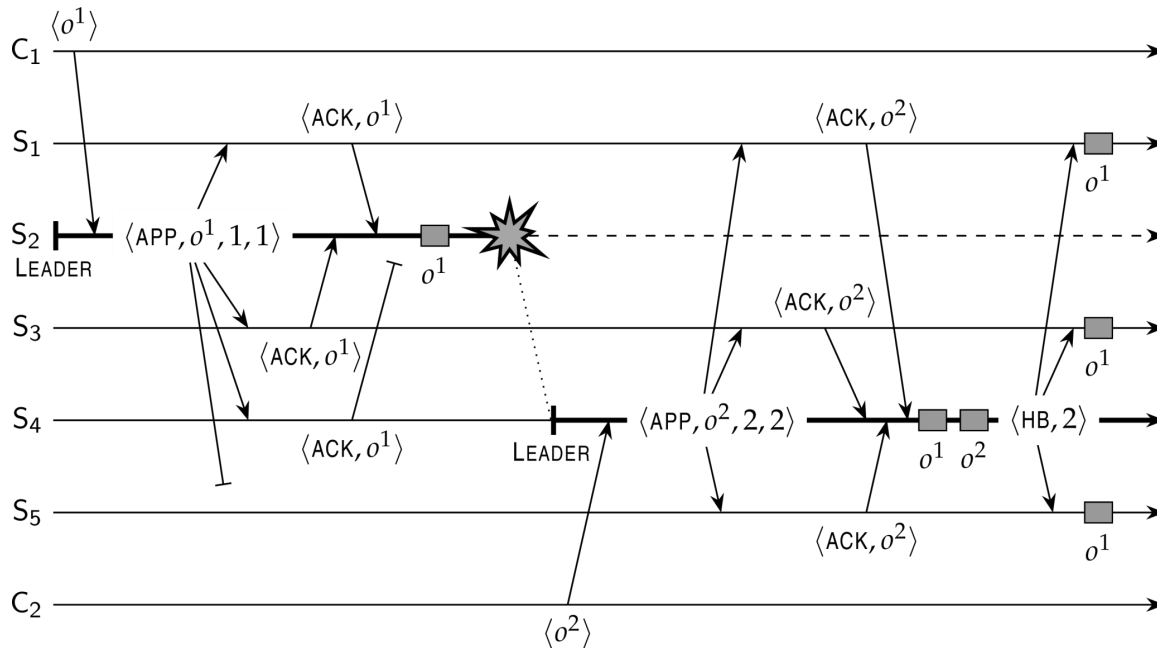


# Raft Crashing



1. Client  $C_1$  sends an operation  $o_1$  to the leader ( $S_2$ )
2.  $S_2$  records  $o_1$  as index 1 term 1
3.  $S_2$  sends Apply (APP) messages to others  $\langle APP, o_1, 1, 1 \rangle$
4. Majority acknowledge -  $S_1, S_3, S_4$
5.  $S_5$  does not receive APP
6.  $S_2$  commits  $o_1$
7.  $S_2$  crashes

# Raft Crashing



1.  $S_4$  notices crash, declares candidacy
2.  $S_4$  elected leader
3. Client  $C_2$  sends an operation  $o_2$  to leader ( $S_4$ )
4.  $S_4$  records  $o_2$  as index 2 term 2
5.  $S_4$  sends APP messages  $\langle APP, o_2, 2, 2 \rangle$
6. Majority acknowledge
7.  $S_4$  commits  $o_1, o_2$
8.  $S_4$  sends heartbeat message (HB) with current index (2)
  - Includes entire log
9.  $S_1, S_3, S_5$  commit  $o_1$

## Assumptions

- Partially synchronous (may even be asynchronous)
- Communication between processes may be unreliable
  - lost, duplicated, reordered
- Corrupted message can be detected (and ignored)
- All operations are deterministic
  - Once an execution is started, we know exactly what it will do
- Processes crash-fail, but not arbitrary fail
- Processes do not collude

## Essentials

- Assume a client-server configuration, with initially one primary server (the **leader**)
- For robustness, we add a backup server
- To ensure that all commands are executed in the same order at both servers, primary assigns unique sequence numbers to all commands
- Actual commands can always be restored (either from clients or servers), so we consider only control messages.

---

# For more info, read the paper

- Kirsch J. and Amir Y. Paxos for System Builders. Technical Report CNDP-2008-2, John Hopkins University, Mar. 2008 <http://www.cnds.jhu.edu/pub/papers/cnds-2008-2.pdf>

# Failure Detection

---

## Issue

How can we **reliably** detect that a process has **actually crashed**?

---

- **General model:**
  - Each process is equipped with a **failure detection module**
  - A process  $p$  **probes** another process  $q$  for a **reaction**
  - $q$  reacts  $\rightarrow q$  is **alive**
  - $q$  does not react within  $t$  time units  $\rightarrow q$  is **suspected** to have **crashed**
- **Note:** in a synchronous system:
  - A **suspected** crash is a **known** crash
  - Referred to as a **perfect failure detector**

# Failure Detection

---

- **Practice:** the eventually **perfect failure detector**
- Has two important properties:
  - **Strong completeness:** every **crashed process** is **eventually suspected** to have crashed by every correct process
  - **Eventual strong accuracy:** **eventually**, no **correct process** is **suspected** by any other correct process to have crashed
- **Implementation:**
  - If  $p$  did not receive heartbeat from  $q$  **within time**  $t \rightarrow p$  **suspects**  $q$
  - If  $q$  **later sends a message** (received by  $p$ ):
    - $p$  stops suspecting  $q$
    - $p$  **increases timeout** value  $t$
    - Note: If  $q$  does crash,  $p$  will keep suspecting  $q$

# Reliable Communication

---

**So far:** Concentrated on **process resilience** (by process groups). What about **reliable communication channels**?

## Error detection:

- Framing of packets to allow for bit error detection
- Use of frame numbering to detect packet loss

## Error correction:

- Add so much redundancy that corrupted packets can be automatically corrected.
- Request retransmission of lost, or last N packets



# Reliable RPC

---

RPC Communication: What can go wrong?

1: Client  
can't locate  
server

2: Client  
request is  
lost

3: Server  
crashes

4: Server  
response is  
lost

5: Client  
crashes

# Solutions: #1: Client can't locate server

---

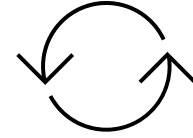
Relatively simple - just report back to client



# Solutions: #2: Client request is lost

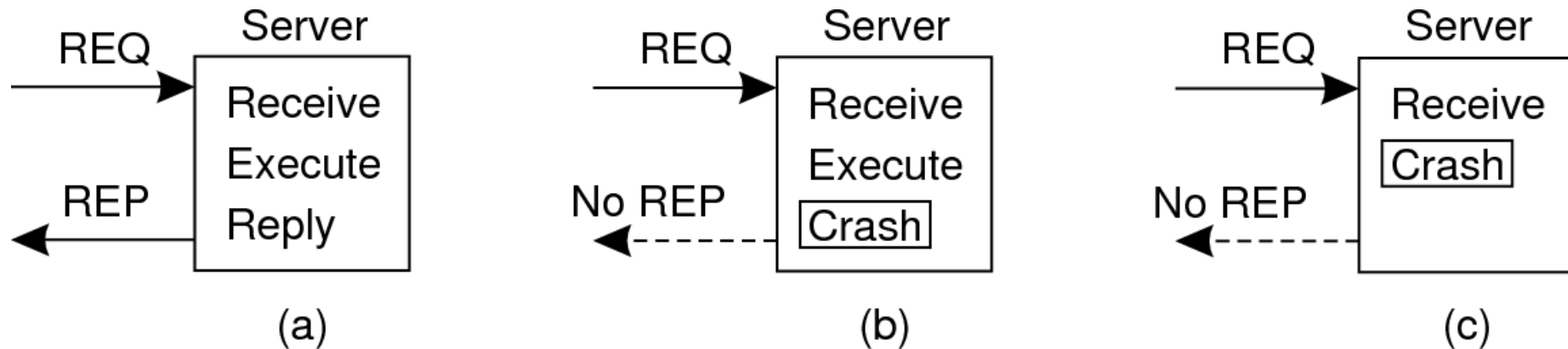
---

Just resend the request



# Solutions: #3 Server Crashes

Harder because you don't know what server did:



Need to decide on what we expect from the server:

- **At-least-once semantics:** The server guarantees it will carry out an operation at least once
- **At-most-once-semantics:** The server guarantees it will carry out an operation at most once.

# Solutions: #4 Server Response Lost

---

Detecting lost replies **can be hard**, because it can also be that the server had crashed. **You don't know** whether the server has carried out the operation.

- **Solution**: None, except that you can try to make your operations **idempotent**: repeatable without any harm done if it happened to be carried out before.

# Solutions: #5 Client crashes

---

**Problem:** The server is doing work and holding resources for nothing (called doing an **orphan** computation).

- Orphan is killed (or rolled back) by client when it reboots
- Broadcast new **epoch number** when recovering → servers kill orphans
- Requires computations to complete in  $T$  time units. Old ones are simply removed.

**Question:** What's the rolling back for?

# Groups and Failure Masking

---

- **$k$ -Fault-tolerant group**: When a group can mask any  $k$  concurrent member failures ( $k$  is called **degree of fault tolerance**).
- How **large** must a  $k$ -fault-tolerant group be:
  - With **halting failures** (crash/omission/timing failures): we need  **$k + 1$**  members: no member will produce an incorrect result, so the result of one member is good enough.
  - With **arbitrary failures**: we need  **$2k + 1$**  members: the correct result can be obtained only through a majority vote.

# Groups and Failure Masking

---

- **Important:**
  - All members are identical
  - All members process commands in the same order
- **Result:**
  - Only then do we know that all processes are programmed to do exactly the same thing

---

## Observation

The processes need to have **consensus** on which command to execute next

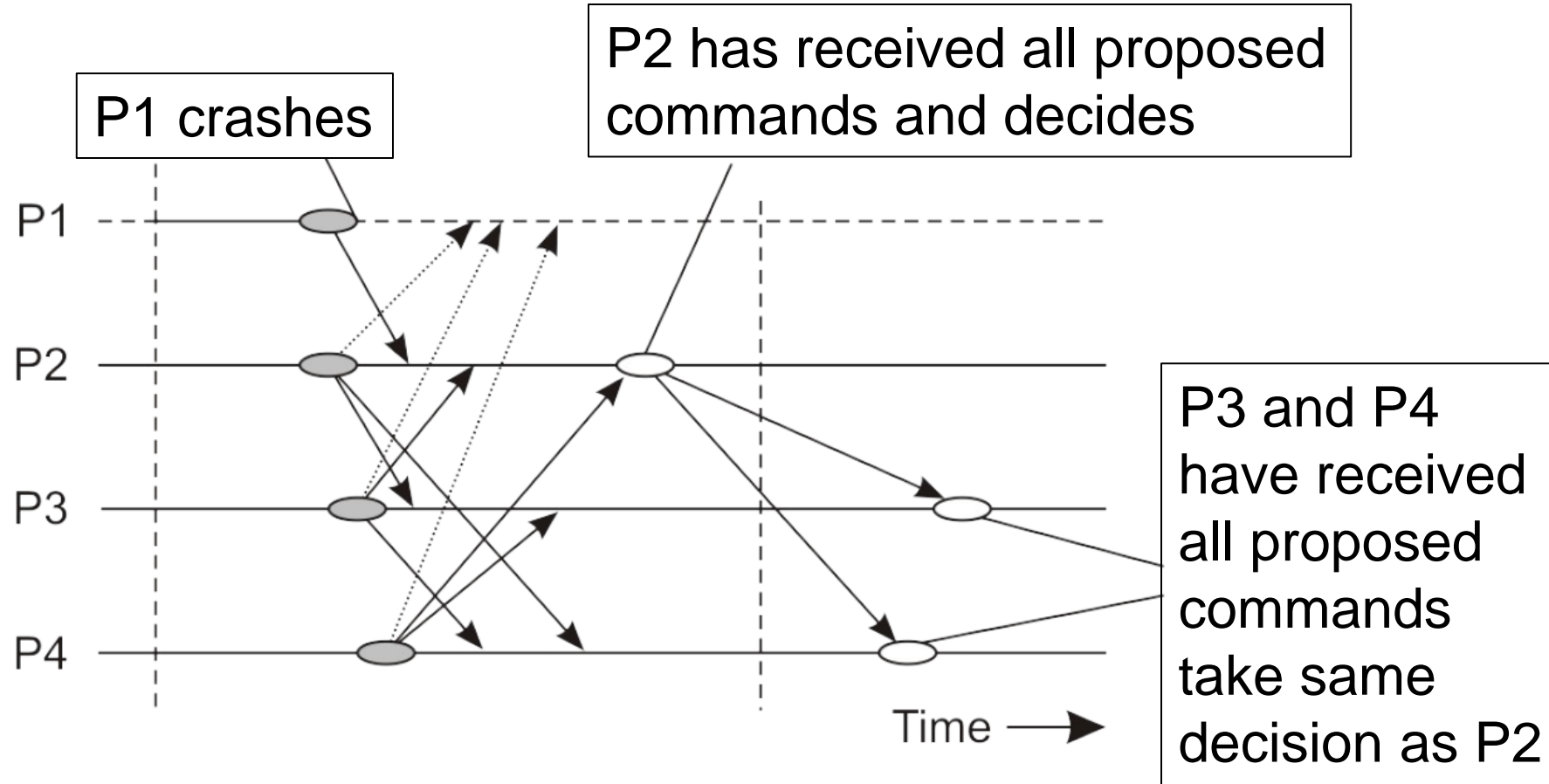


# Flood-based Consensus

---

- Assume:
  - Fail-crash semantics
  - Reliable failure detection
  - Unreliable communication
- Basic idea:
  - Processes multicast their proposed operations
  - All apply the same selection procedure → all processes will execute the same if no failures occur
- Problem:
  - Suppose a process crashes before completing its multicast

# Flood-based Consensus



# Failure Detection

---

## Issue

How can we **reliably** detect that a process has **actually crashed**?

---

- **General model:**
  - Each process is equipped with a **failure detection module**
  - A process  $p$  **probes** another process  $q$  for a **reaction**
  - $q$  reacts  $\rightarrow q$  is **alive**
  - $q$  does not react within  $t$  time units  $\rightarrow q$  is **suspected** to have **crashed**
- **Note:** in a synchronous system:
  - A **suspected** crash is a **known** crash
  - Referred to as a **perfect failure detector**

# Failure Detection

---

- **Practice:** the eventually **perfect failure detector**
- Has two important properties:
  - **Strong completeness:** every **crashed process** is **eventually suspected** to have crashed by every correct process
  - **Eventual strong accuracy:** **eventually**, no **correct process** is **suspected** by any other correct process to have crashed
- **Implementation:**
  - If  $p$  did not receive heartbeat from  $q$  **within time**  $t \rightarrow p$  **suspects**  $q$
  - If  $q$  **later sends a message** (received by  $p$ ):
    - $p$  stops suspecting  $q$
    - $p$  **increases timeout** value  $t$
    - Note: If  $q$  does crash,  $p$  will keep suspecting  $q$

# Reliable Communication

---

**So far:** Concentrated on **process resilience** (by process groups). What about **reliable communication channels**?

## Error detection:

- Framing of packets to allow for bit error detection
- Use of frame numbering to detect packet loss

## Error correction:

- Add so much redundancy that corrupted packets can be automatically corrected.
- Request retransmission of lost, or last N packets

# Reliable RPC

---

## RPC Communication: What can go wrong?

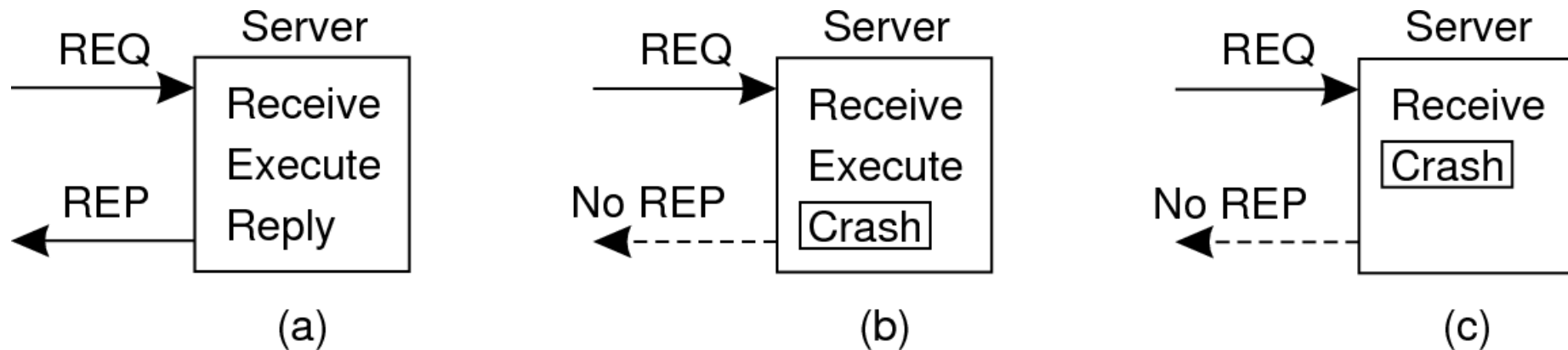
1. Client can't locate server
  2. Client request is lost
  3. Server crashes
  4. Server response is lost
  5. Client crashes
- 

## RPC Communication: Solutions

1. Relatively simple - just report back to client
2. Just resend message

# RPC: Solutions

3. Server crashes: Server crashes are harder as you don't know what it had already done:



Need to decide on what we expect from the server:

- **At-least-once semantics:** The server guarantees it will carry out an operation at least once
- **At-most-once-semantics:** The server guarantees it will carry out an operation at most once.

# Reliable RPC

---

Server response is lost: Detecting lost replies **can be hard**, because it can also be that the server had crashed. **You don't know** whether the server has carried out the operation.

**Solution:** None, except that you can try to make your operations **idempotent**: repeatable without any harm done if it happened to be carried out before.



# Reliable RPC: Client crashes

---

**Problem:** The server is doing work and holding resources for nothing (called doing an **orphan** computation).

- Orphan is killed (or rolled back) by client when it reboots
- Broadcast new **epoch number** when recovering → servers kill orphans
- Requires computations to complete in  $T$  time units. Old ones are simply removed.

**Question:** What's the rolling back for?

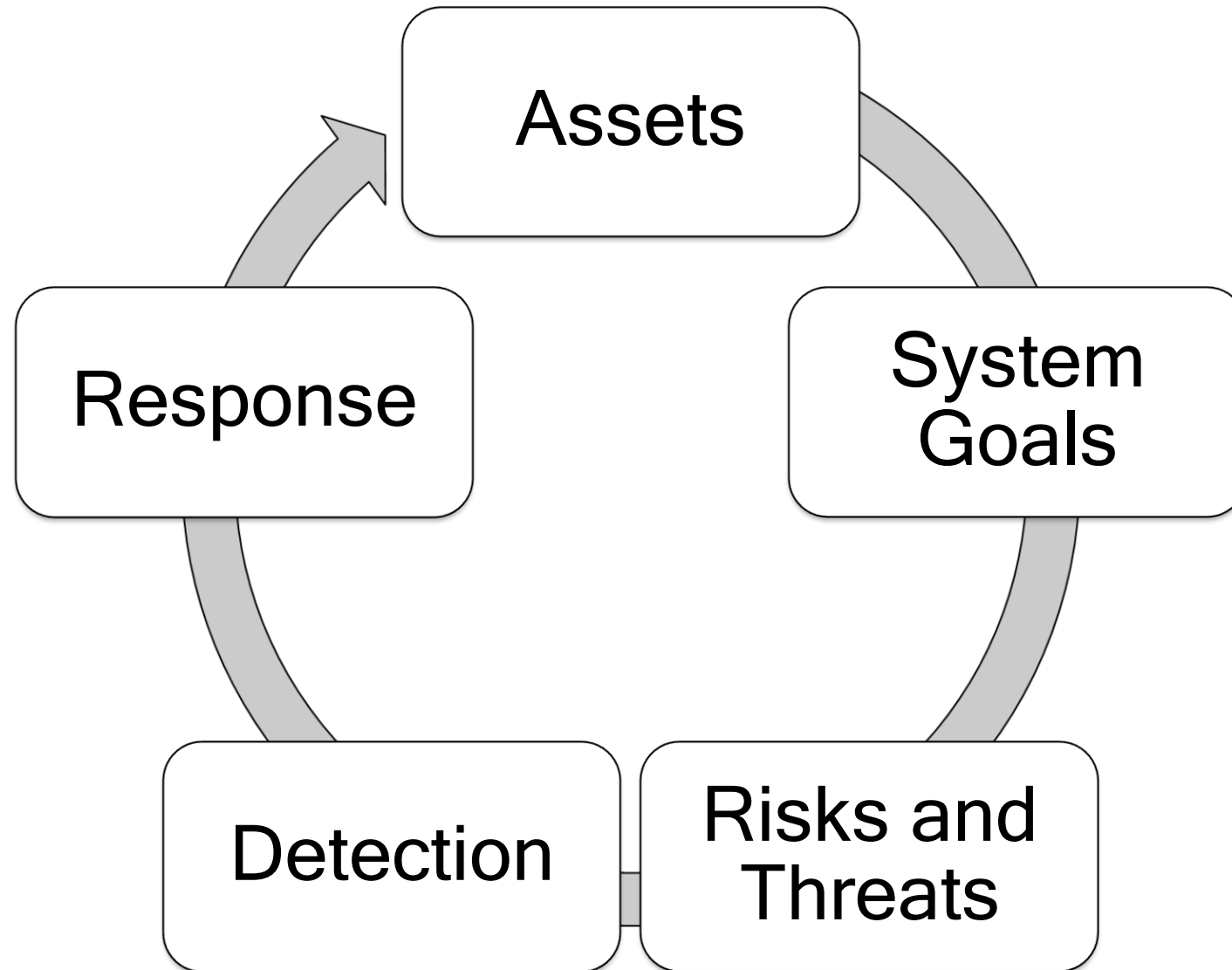
# So Far

---

- Failures and Resilience
- Raft and Consensus
- Cyber 3: Consensus and Failure

# Consistency and Consensus Cyber

---



# Assets - Consistency and Consensus

---

Databases

Data items

Transaction  
servers

Transaction  
logs

Communication  
links

Communication  
protocols

# System Goals - Consistency and Consensus

---

Provide  
consistent data

Allow timely  
writing

Keep nodes up  
to date

Don't lose data  
or transactions

Permit servers  
to come and  
leave

Tolerate failure  
and recover

Make it “just  
work” from user  
perspective

Identify  
problematic  
operations and  
undo/flag

# Risks and Threats - Consistency and Consensus

---

Malicious  
customers/users  
provide bad data

Network provider  
disrupts  
communication

Malicious server  
corrupts data

Misconfigured  
user/server causes  
system freeze

Misconfigured  
user/server causes  
system crash

Crash occurs, losing  
data or transactions

Network  
communications lag  
too much, breaking  
time assumptions

# Detection - Consistency and Consensus

---

Event monitoring

Dashboards for  
performance

Watch for  
network  
heartbeats and  
updates

Watch for client  
interaction  
delays

Test data for  
consistency and  
responsiveness

Monitor server  
communications  
and down time

Run test jobs to  
ensure they  
complete on time  
and consistently

# Response

---

Ignore/kick out  
malicious  
users/customers

Bring up additional  
servers when  
crashes occur

Watch for network  
partitions

Locate data near  
where it's needed  
to provide service in  
case of outage

Add servers or  
bandwidth  
connections if  
crashes occur often



# Conclusion

---

- Failures and Resilience