The University for
world-class professionals

Manchester Metropolitan University

Advanced Operating Systems
Lecture 6: Fault Tolerance

Soufiene Djahel
Office: John Dalton E114

Email: s.djahel@mmu.ac.uk

Telephone: 0161 247 1522

Office hours: Monday 13 -15, Tuesday 11-12

Credits: Some slides adapted from material by
Anthony D. Joseph (UC Berkley)
Jalal Y. Kawash (U. Calgary)
Daniel M. Zimmerman (CALTECH)

Recap from last week

- Virtualisation technology
- Why use Virtualisation?
- Virtual Machine Monitor (VMM)
- Different approaches to virtualisation
- Performance issues

Today’s objectives

- Definitions for Fault Tolerance
- Causes of system failures
- Fault Tolerance approaches
  - HW- and SW-based Fault Tolerance, Datacenters, Cloud, Geographic diversity

Fault-tolerant computing system

- A fault-tolerant computing system is a system which has the built-in capability (without external assistance) to preserve the continued correct execution of its programs and I/O functions in the presence of a certain set of operational faults.

(Algirdas Avizienis, Fault-Tolerant Systems)

Fault tolerance in distributed systems

- A distributed system should be fault-tolerant
  - Should be able to continue functioning in the presence of faults

- Fault tolerance is related to **dependability**

Dependability attributes

Dependability is a measure of a system's

- Availability
- Reliability
- Safety
- Maintainability

Dependability

- **Availability:** A measurement of whether a system is *ready to be used immediately*
  - System is up and running at any given moment

- **Reliability:** A measurement of whether a system can *run continuously without failure*
  - System continues to function for a long period of time

Dependability

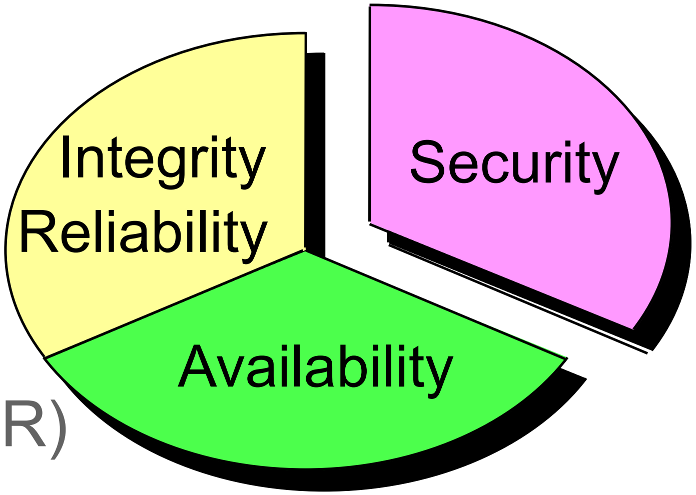
- A system goes down 1ms/hr has an availability of more than 99.99%, but is unreliable

- A system that never crashes but is shut down for a week once every year is 100% reliable but only 98% available

Dependability

- **Safety:** A measurement of *how safe failures are*
  - System fails, nothing serious happens
  - For instance, high degree of safety is required for systems controlling nuclear power plants
- **Maintainability:** A measurement of *how easy it is to repair a system*
  - A highly maintainable system may also show a high degree of availability
  - Failures can be detected and repaired automatically? Self-healing systems?

Dependability (an alternative view)



- **Reliability / Integrity:**
  does the right thing.

  (Need large MTBF)

- **Availability:** does it now.

  (Need small (MTTR/(MTBF+MTTR)

- **System Availability:**
  if 90% of terminals up & 99% of DB up?

  (=> 89% of transactions are serviced on time)

MTBF or MTTF = Mean Time Between (To) Failure
MTTR = Mean Time To Repair

Mean Time to Recovery

- Critical time as further failures can occur during recovery

- Total Outage duration (MTTR) =

Time to Detect	(need good monitoring)
+ Time to Diagnose	(need good docs/ops, best practices)
+ Time to Decide	(need good org/leader, best practices)
+ Time to Act	(need good execution!)

Fault Tolerance vs. Disaster Tolerance

- **Fault-Tolerance:** mask local faults
  - Redundant HW or SW
  - RAID (Redundant Array of Independent Disks) disks
  - Uninterruptible Power Supplies
  - Cluster Failover
- **Disaster Tolerance:** masks site failures
  - Protects against fire, flood, sabotage,..
  - Redundant system and service at remote site(s)
  - Use design diversity

High Availability System Classes

Manchester Metropolitan University

Availability %	Downtime per year	Downtime per month	Downtime per week
90% ("one nine")	36.5 days	72 hours	16.8 hours
99% ("two nines")	3.65 days	7.20 hours	1.68 hours
99.9% ("three nines")	8.76 hours	43.2 minutes	10.1 minutes
99.99% ("four nines")	52.56 minutes	4.32 minutes	1.01 minutes
99.999% ("five nines")	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% ("six nines")	31.5 seconds	2.59 seconds	0.605 seconds

GOAL: Class 6
2010: Gmail (99.984), Exchange (>99.9)
Unavailability ~ MTTR/MTBF
    Can cut it by reducing MTTR *or* increasing MTBF

Causal Factors for Unavailability

Lack of best practices for:

- Change control
- Monitoring of the relevant components
- Requirements and procurement
- Operations
- Avoidance of network failures, internal application failures, and external services that fail
- Physical environment, and network redundancy
- Technical solution of backup, and process solution of backup
- Physical location, infrastructure redundancy
- Storage architecture redundancy

Source: Ulrik Franke et al: Availability of enterprise IT systems - an expert-based Bayesian model

Faults

- A system **fails** when it cannot meet its promises (specifications)
- An **error** is part of a system state that may lead to a failure
- A **fault** is the cause of the error

- **Fault-Tolerance**: the system can provide services even in the presence of faults
- Faults can be:
  - Transient (appear once and disappear)
  - Intermittent (appear-disappear-reappear behavior)
    - A loose contact on a connector → intermittent fault
  - Permanent (appear and persist until repaired)

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 (Byzantine failure)	A server may produce arbitrary responses at arbitrary times

Cloud Computing Outages 2011

Vendor	When	Duration	What Happened & Why
Apple iPhone 4S Siri	November 2011	1 Day	Siri loses even the most basic functionality when Apples servers are down. Because Siri depends on servers to do the heavy computing required for voice recognition, the service is useless without that connection. Network outages caused the disruption according to Apple.
Blackberry outage	October 2011	3 Days	Outage was caused by a hardware failure (core switch failure) that prompted a "ripple effect" in RIM's systems. Users in Europe, Middle East, Africa, India, Brazil, China and Argentina initially experienced email and message delays and complete outages and later the outages spread to North America too. Main problem is message backlogs and the downtime produced a huge queue of undelivered messages causing delays and traffic jams.
Google Docs	September 2011	1 Hour	Google Docs word collaboration application cramp, shutting out millions of users from their document lists, documents, drawings and Apps Scripts. Outage was caused by a memory management bug software engineers triggered in a change designed to "improve real time collaboration within the document list.
Windows Live services - Hotmail & SkyDrive	September 2011	3 Hours	Users did not have any data loss during the outage and the interruption was due to an issue in Domain Name Service (DNS). Network traffic balancing tool had an update and the update did not work properly which caused the issue.
Amazon's EC2 cloud &	August 2011	1-2 days	Transformer exploded and caught fire near datacenter that resulted in power outage due to generator failure. Power back up systems at both the data centers failed causing power outages. Transformer explosion was caused by lightening strike but disputed by local utility provider.
Microsoft's BPOS	August 2011	1-2 days	Transformer exploded and caught fire near datacenter that resulted in power outage due to generator failure. Power back up systems at both the data centers failed causing power outages. Transformer explosion was caused by lightening strike but disputed by local utility provider.

Traditional fault tolerance techniques

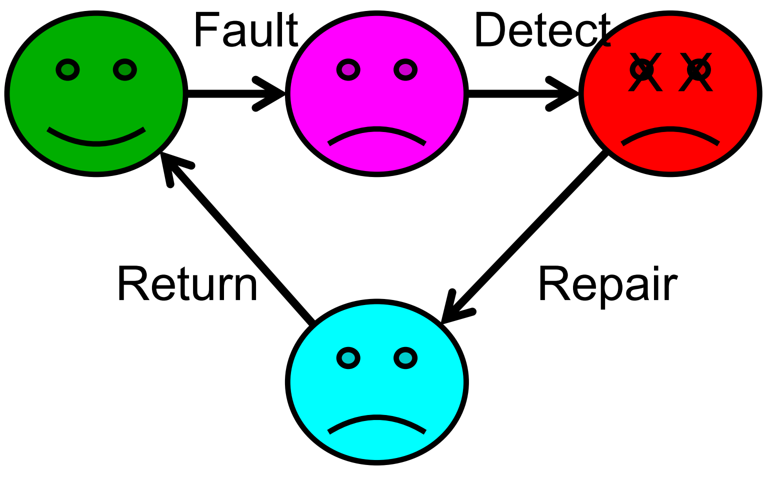
- Fail fast modules: work or stop
- Spare modules: yield instant repair time
- Process/Server pairs: Mask HW and SW faults
- Transactions: yields ACID (Atomicity, Consistency, Isolation, Durability) semantics (simple fault model)

Fail-Fast is Good; Repair is Needed

Lifecycle of a module
- fail-fast gives
- short fault latency

High Availability
- is low UN-Availability

Unavailability ~ MTTR/MTBF



Improving either MTTR or MTBF gives benefit

Software Techniques: Learning from Hardware

- Fault avoidance starts with a *good and correct design*

- After that – Software Fault Tolerance Techniques:
  Modularity (isolation, fault containment)
  Programming for Failures: Programming paradigms that assume failures are common and hide them
  Defensive Programming: Check parameters and data
  N-Version Programming: N-different implementations
  Auditors: Check data structures in background

Try & Catch Alone isn’t Fault Tolerance!

```
String filename = "/nosuchdir/myfilename";

try {
    // Create the file
    new File(filename).createNewFile();
}
catch (IOException e) {
    // Print out the exception that occurred
    System.out.println("Unable to create file
("+filename+”): "+e.getMessage());
}
```

- Fail-Fast, but is this the desired behavior?
- Alternative behavior: (re)-create missing directory?!

Recovery

- We’ve talked a lot about fault tolerance, but not about what happens after a fault has occurred
- A process that exhibits a failure has to be able to recover to a correct state
- There are two basic types of recovery:
  - Backward Recovery
  - Forward Recovery

Backward Recovery

- The goal of *backward recovery* is to bring the system from an erroneous state back to a prior correct state
- The state of the system must be recorded - *checkpointed* - from time to time, and then restored when things go wrong
  - Example
    - Reliable communication through packet retransmission

Forward Recovery

- The goal of *forward recovery* is to bring a system from an erroneous state to a correct new state (not a previous state)
- Example:
  - Reliable communication via erasure correction, such as an (*n*, *k*) block erasure code

More on Backward Recovery

- Backward recovery is far more widely applied
- The goal of *backward recovery* is to bring the system from an erroneous state back to a prior correct state
- But, how to get a prior correct state?
  - Checkpointing
  - Checkpointing is costly, so it is often combined with *message logging*

Stable Storage

- In order to store checkpoints and logs, information needs to be stored safely - not just able to survive crashes, but also able to survive hardware faults

- RAID is the typical example of stable storage

Determining Global States

- **The global state of a distributed computation is**
  - *the set of local states of all individual processes involved in the computation*

  +

  - *the states of the communication channels*

- **How?**

Obvious First Solution…

- Synchronize clocks of all processes and ask all processes to record their states at known time t
- Problems?
  - Time synchronization possible only approximately
    - distributed banking applications: no approximations!
  - Does not record the state of messages in the channels

Global State

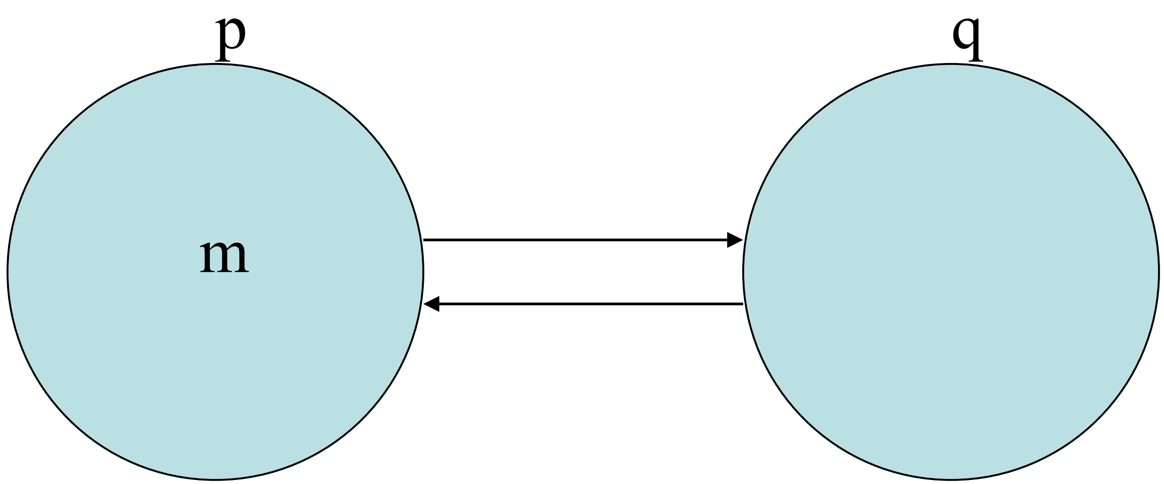
- We cannot determine the exact global state of the system, but we can record a snapshot of it
- **Distributed Snapshot**: a state the system might have been in [Chandy and Lamport]

A naïve snapshot algorithm

- Processes record their states at *any* arbitrary points
- A designated process collects these states

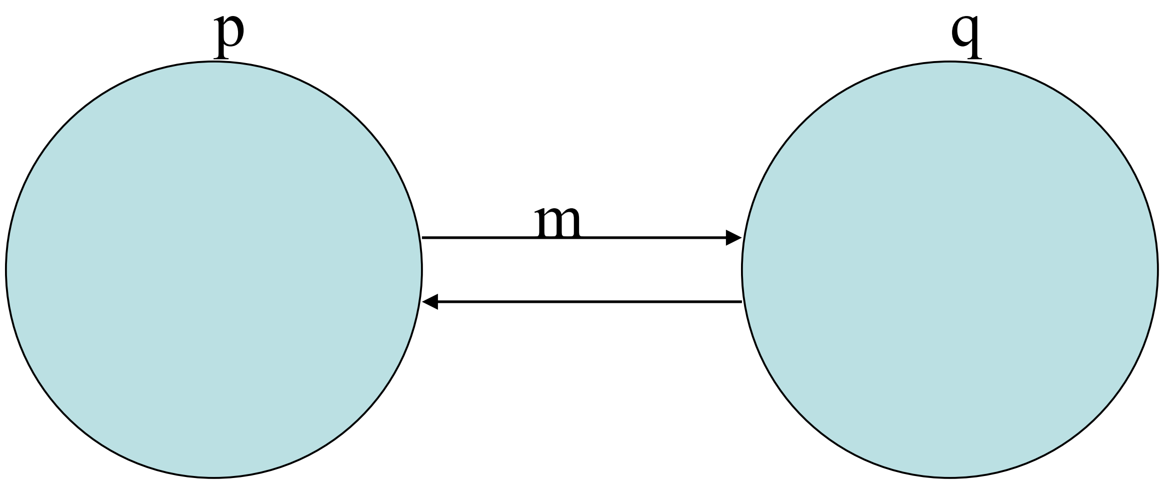
- \+ So simple!!
- \- Correct??

Example: Producer Consumer problem

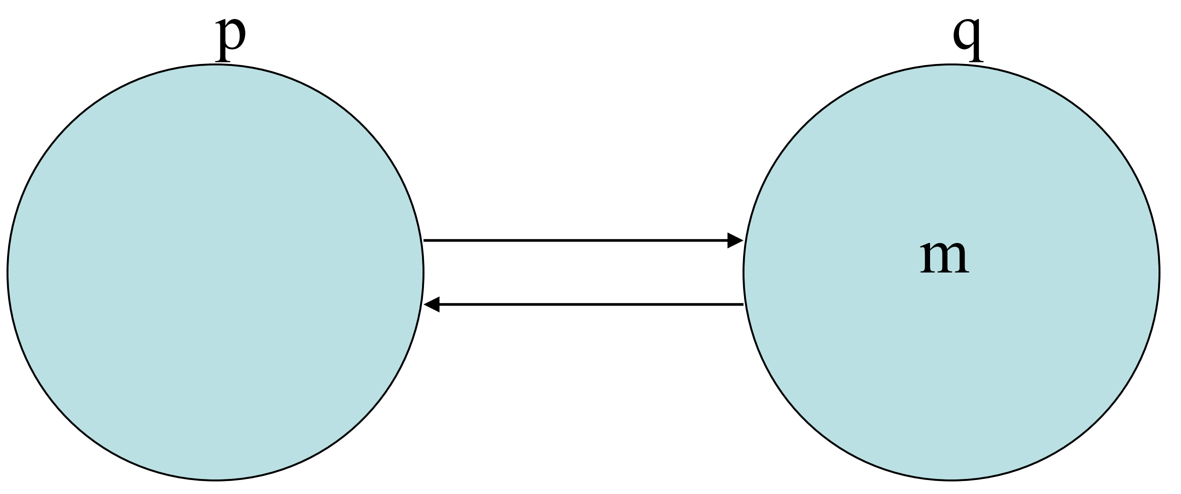


- p records its state

Example

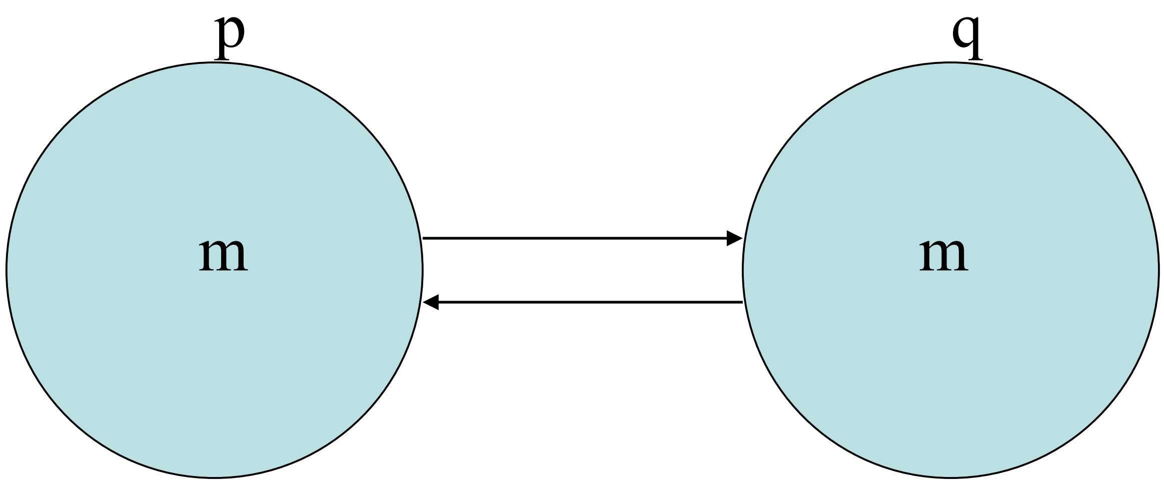


Example



- q records its state

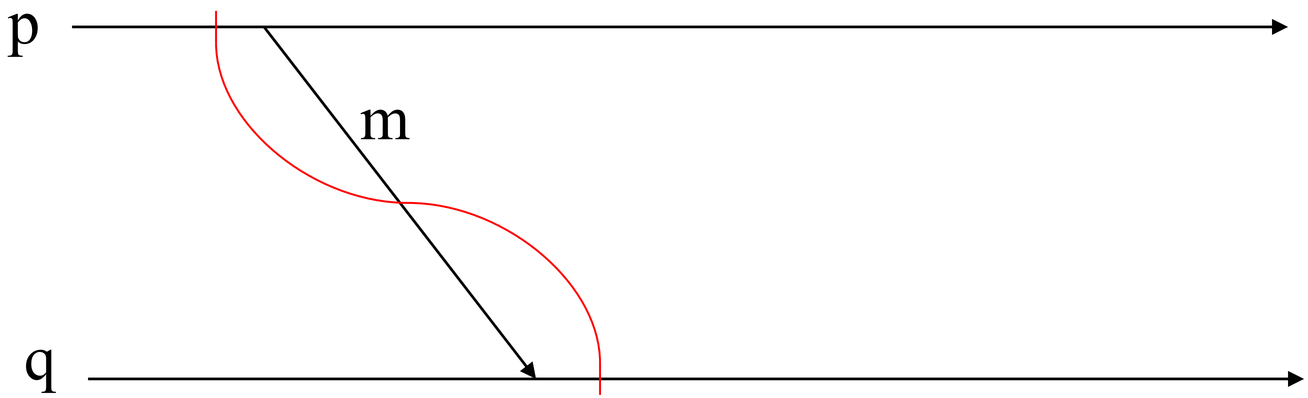
Example: The recorded state



The sender has *no record* of the sending

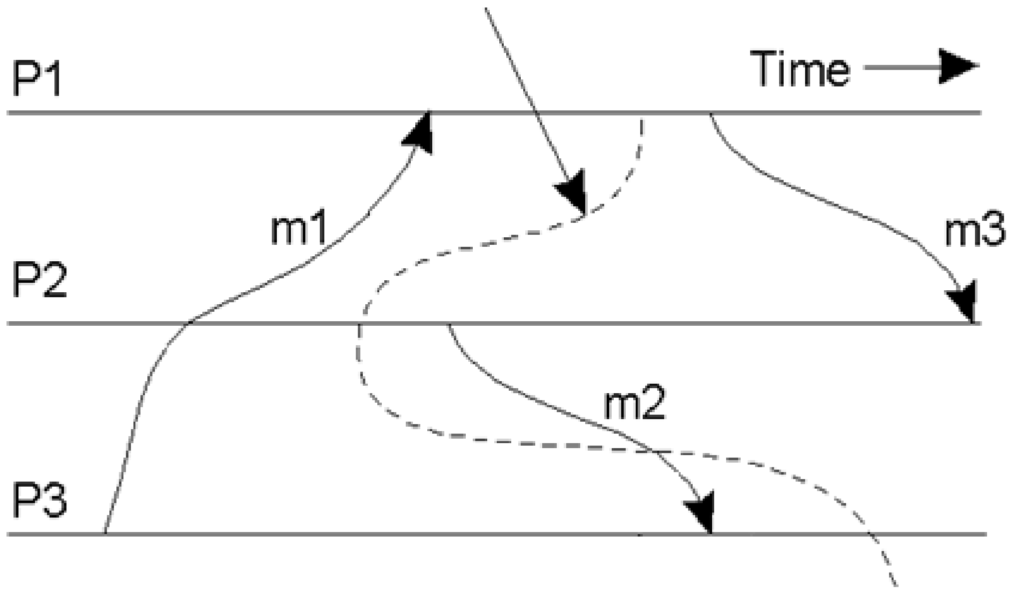
The receiver *has the record* of the receipt

What’s Wrong?

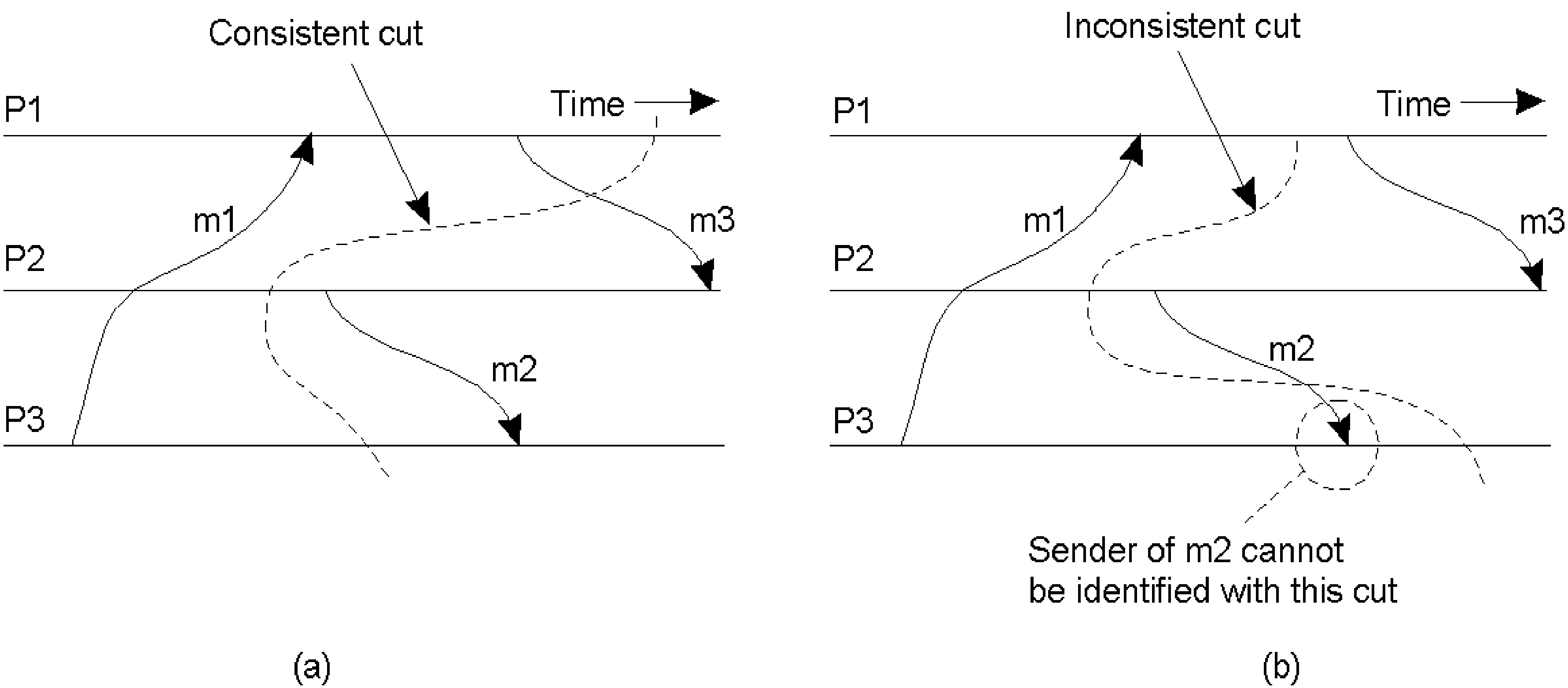


- Result:
  - Global state has record of the receive event but no send event violating the happens-before concept!!

Cut



A consistent cut (**meaningful global state**)?



a) A consistent cut (**meaningful global state**)

b) An inconsistent cut

There is a lot more to this story

- Coordinating distributed snapshots for error recovery is a non-trivial problem
- You need to be aware that it is possible, not the details of how it works!
- (You should have the ability to look it up and figure it out if you need it!)

Summary

- Focus on Reliability and Availability
- Use HW/SW FT to increase MTBF and reduce MTTR
  - Build reliable systems from unreliable components
  - Assume the unlikely is likely
- Make operations bulletproof: configuration changes, upgrades, new feature deployment, …
- Apply replication at all levels (including globally)

References

- Leslie Lamport: https://www.youtube.com/watch?v=zxeFdEUdba4
- Ulrik Franke, et al. Availability of enterprise IT systems: an expert-based Bayesian framework. Software Qual J (2012) 20:369–394
- K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (February 1985), 63-75. DOI=http://dx.doi.org/10.1145/214451.214456