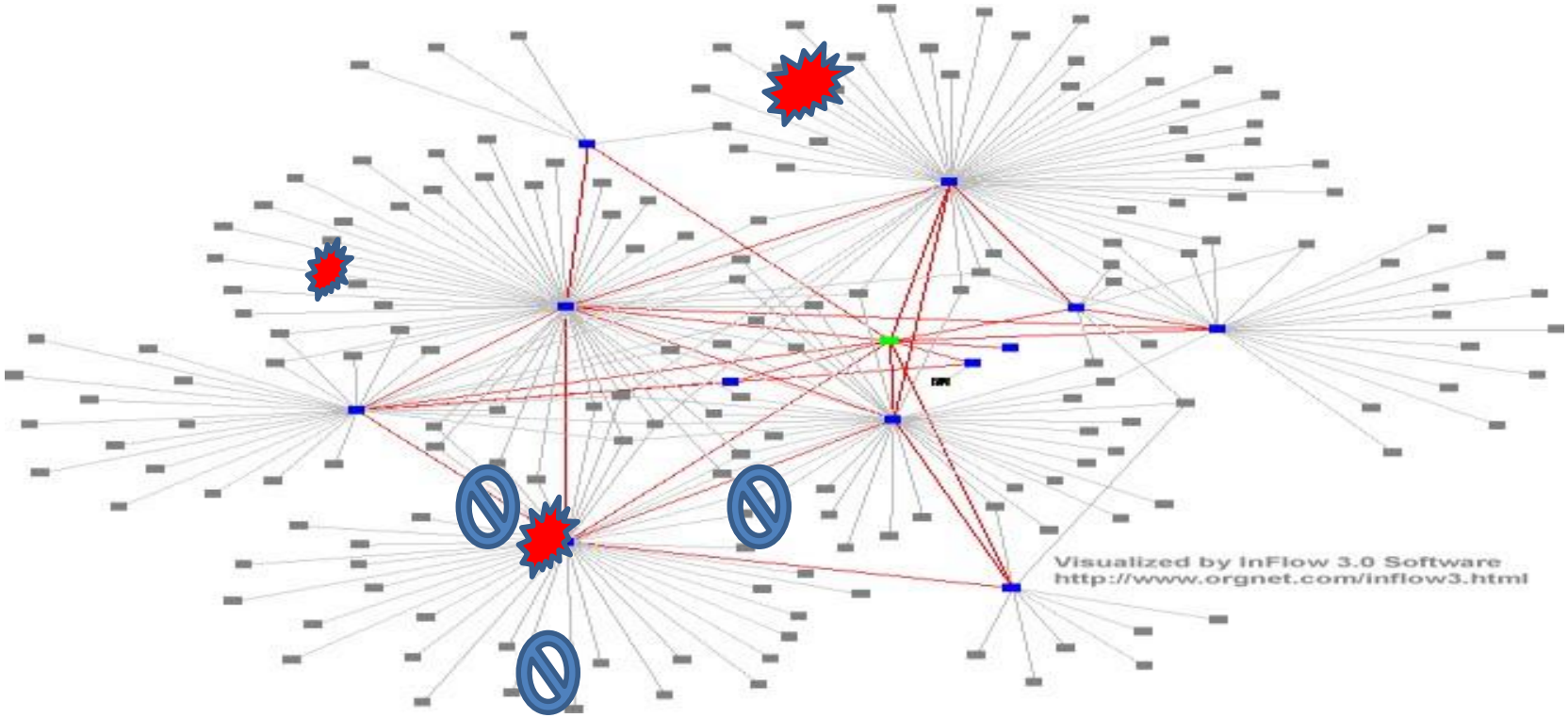
DCS Fault Tolerance Issues Part-1



Dr. Sunny Jeong. spjeong@uic.edu.cn

With Thanks to Prof. B Parhami, Prof. A.S. Tanenbaum,
Prof. J. Chen and Prof. W. Natta

Overview

- What is Fault Tolerance
- Key Terms
- Properly designed fault-tolerant system
- Dependable Systems
- Software Dependability

Introduction

- The failure rates of any system(or computer) must be extraordinarily small.

(*Software Engineering Concepts for DCS*)

  - Such computers must therefore be **fault-tolerant** : be able to continue operating despite the failure of a limited subset of their hardware or software

  - They must also be **gracefully-degradable** : as the size of the faulty set increases, the system must not suddenly collapse but continue executing part of its workload.

  - HW fault or SW fault, which is more urgent?

NYSE outage and market swings blamed on human error impacting disaster recovery system

But the real error is with the human that designed the New York Stock Exchange's process

January 26, 2023 By: Sebastian Moss Have your say

An outage and IT error on Tuesday, January 24, at the New York Stock Exchange caused thousands of trades to be canceled and huge swings in blue-chip stocks.

The problem was blamed on human error, with an NYSE employee failing to properly shut down a disaster recovery system.

The issue led to 84 major stocks seeing their valuation drastically drop or grow, until they hit limits designed to stop securities from trading at extreme prices.

– Sebastian Moss

81 stocks had short-selling restrictions applied due to the glitch, with Morgan Stanley and Snap impacted.

The day after the incident, the exchange said it would have to cancel more than 4,300 trades in 251 different stocks.

The NYSE blamed the issue on a manual error with its disaster recovery configuration.

The Business Times reports that the system connecting to the Chicago-based backup data center is meant to be manually turned on and off when the market closes.

But an employee failed to turn it off properly, leaving the backup system running. At 9:30am, the system skipped the day's opening auctions that set prices and caused turmoil.

The Securities and Exchange Commission is looking into the incident. It introduced rules in 2014 that gives it the power to punish exchanges for technological mishaps.

Four years later, the NYSE became the first exchange to be fined under the new rules.

Fault Tolerance Basic Concepts

Being fault tolerant is strongly related to what are called **dependable systems**

"Dependability".
In Distributed Systems, this is characterized under a number of headings:

Availability – the system is ready to be used immediately.

Reliability – the system can run continuously without failure.

Safety – if a system fails, nothing catastrophic will happen.

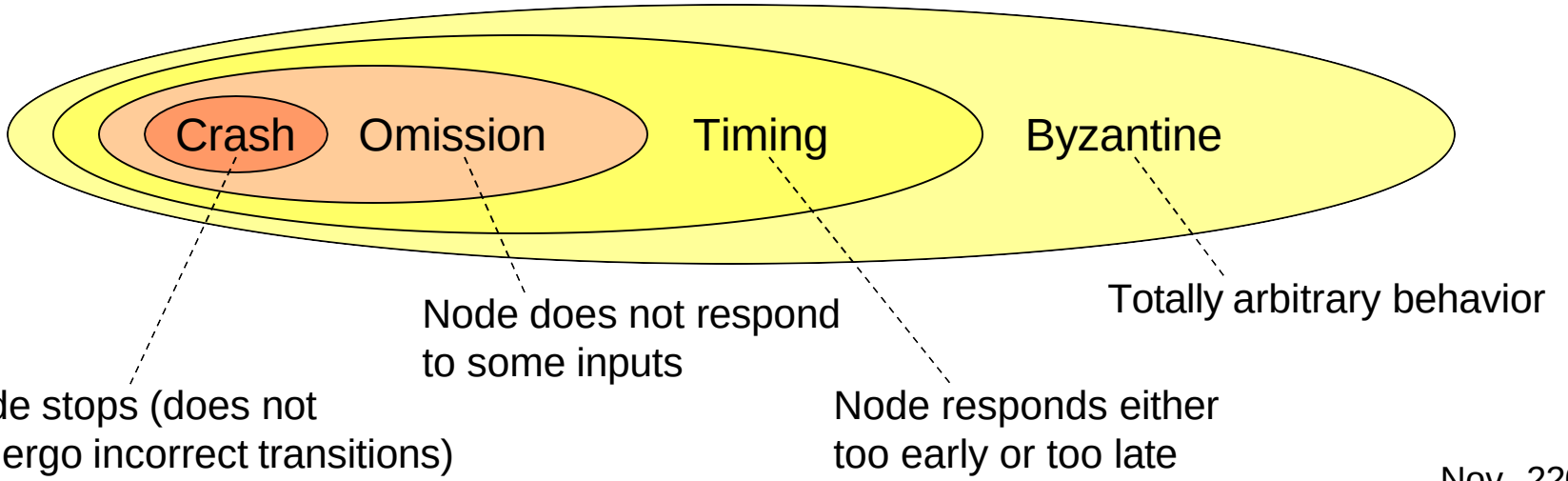
Maintainability – when a system fails, it can be repaired easily and quickly (and, sometimes, without its users noticing the failure).

Dependable Collaboration

Distributed systems, built from COTS nodes (processors plus memory) and interconnects, have redundancy and allow software-based malfunction tolerance implementation

Interconnect malfunctions are dealt with by synthesizing reliable communication primitives (point-to-point, broadcast, multicast)

Node malfunctions are modeled differently, with the more general models requiring greater redundancy to deal with



Failure Models

- Different failure models of A Server system

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*	A 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

Faulty Systems

- Possible cases:

1. Synchronous versus asynchronous systems.
2. Communication delay is bounded or not.
3. Message delivery is ordered or not.
4. Message transmission is done through unicasting or multicasting.

What Is “Failure”?

Definition:

A system is said to “fail” when it *cannot meet* its promises.

A failure is brought about by the *existence* of “errors” in the system.

The *cause* of an error is called a “fault”.

Types of Fault

There are three main types of ‘fault’:

Transient Fault – appears once, then disappears.

Intermittent Fault – occurs, vanishes, reappears; but: follows no real pattern (worst kind).

Permanent Fault – once it occurs, only the replacement/repair of a faulty component will allow the DS to function normally.

Failure Masking by Redundancy

Strategy: hide the occurrence of failure from other processes using *redundancy*. Three main types:

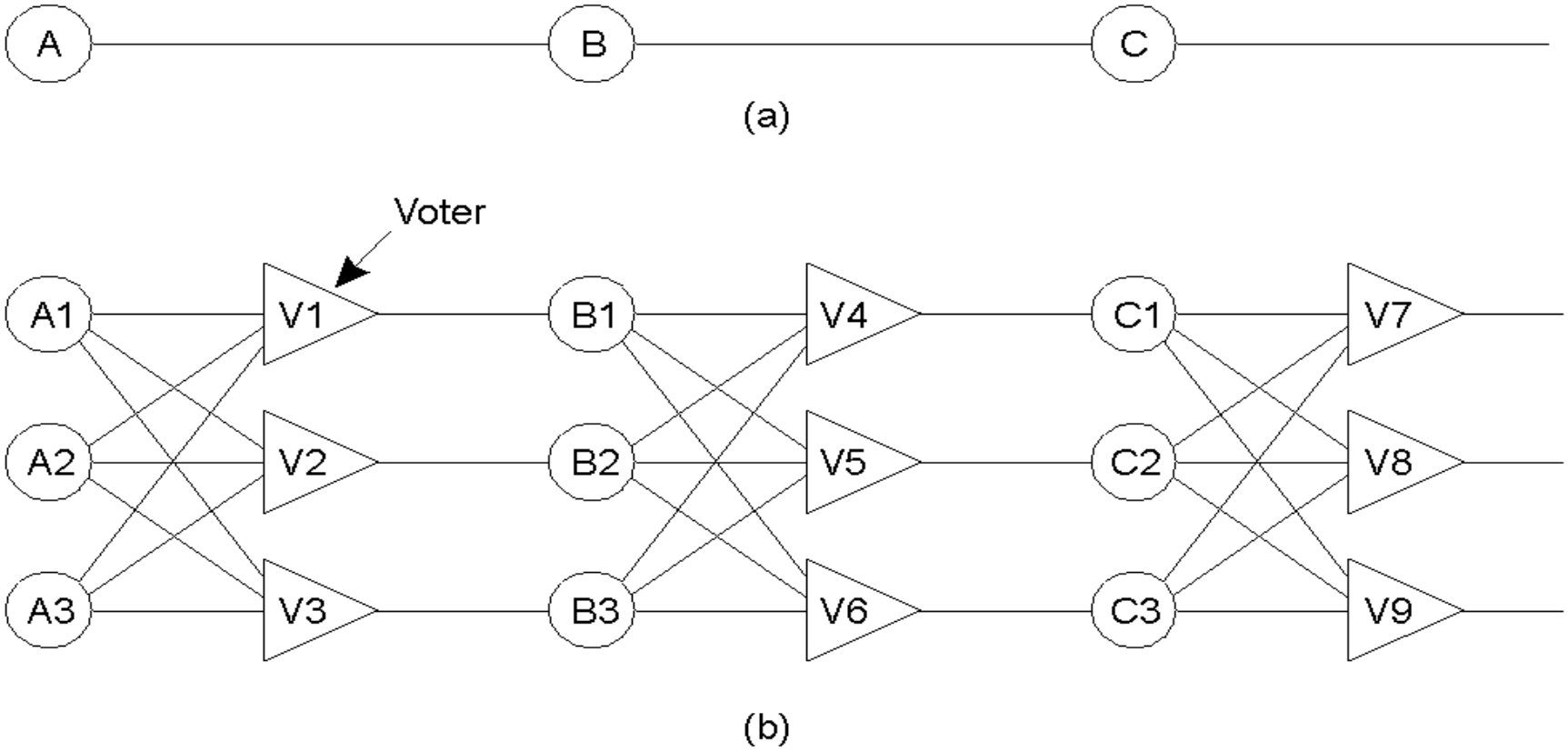
Information Redundancy – add extra bits to allow for error detection/recovery (e.g., Hamming codes and the like).

Time Redundancy – perform operation and, if needs be, perform it again. Think about how transactions work (BEGIN/END/COMMIT/ABORT).

Physical Redundancy – add extra (duplicate) hardware and/or software to the system.

Failure Masking by Redundancy

Triple modular redundancy. (Physical Redundancy)



DS Fault Tolerance Issues

1. Process Resilience
2. Reliable Client/Server Communications
3. Reliable Group Communication
4. Distributed COMMIT
5. Recovery Strategies

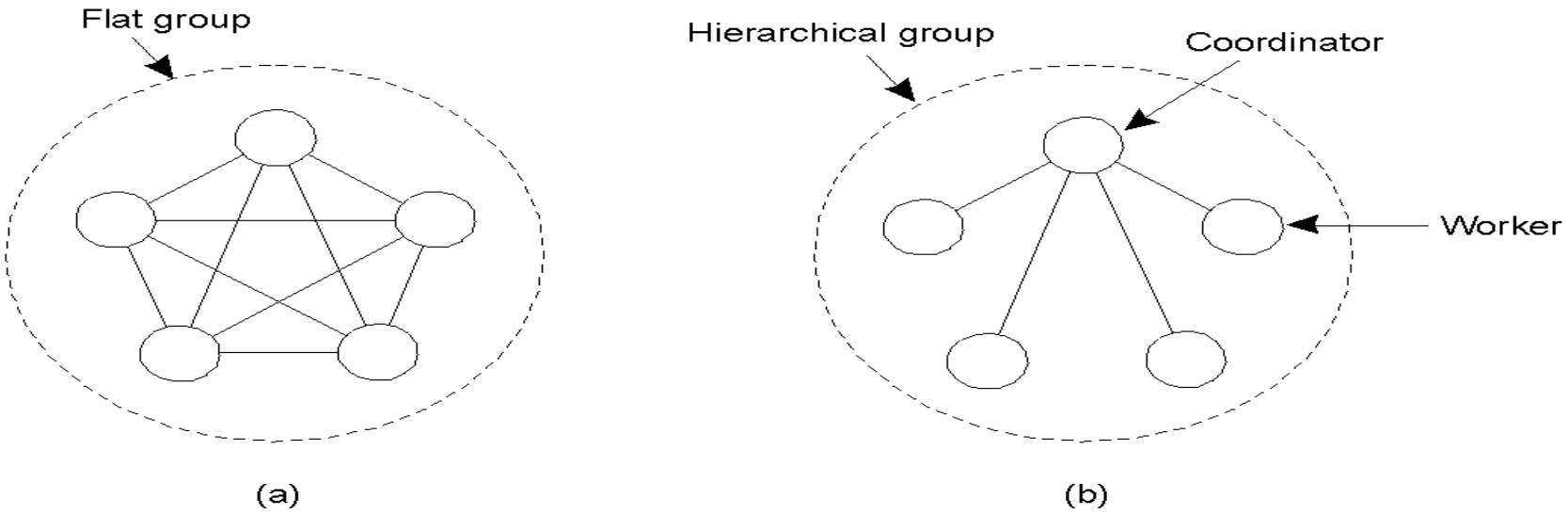
1. Process Resilience

Processes can be made fault tolerant by arranging to have a group of processes, with each member of the group being *identical*.

A message sent to the group is delivered to all of the “copies” of the process (the group members), and then *only one* of them performs the required service.

If one of to function (and service any pending request or operation). the processes fail, it is assumed that one of the others will still be able

Flat vs. Hierarchical Groups



a) **Communication in a flat group** – all the processes are equal, decisions are made collectively. **Note**: no single point-of-failure, however: decision making is complicated as consensus is required.

b) **Communication in a simple hierarchical group** – one of the processes is elected to be the coordinator, which selects another process (a worker) to perform the operation. **Note**: single point-of-failure, however: decisions are easily and quickly made by the coordinator without first having to get consensus.

Failure Masking and Replication

By organizing a *fault tolerant group of processes*, we can protect a single vulnerable process.

There are two approaches to arranging the replication of the group:

1. Primary (backup) Protocols.
2. Replicated-Write Protocols.

The Goal of Agreement Algorithms

"To have all *non-faulty* processed reach consensus on some issue (quickly)."

The **two-army problem**.

Even with non-faulty processes, agreement between even two processes is not possible in the face of unreliable communication.

History Lesson: The Byzantine Empire

Time: 330-1453 AD.

Place: Balkans and Modern Turkey.

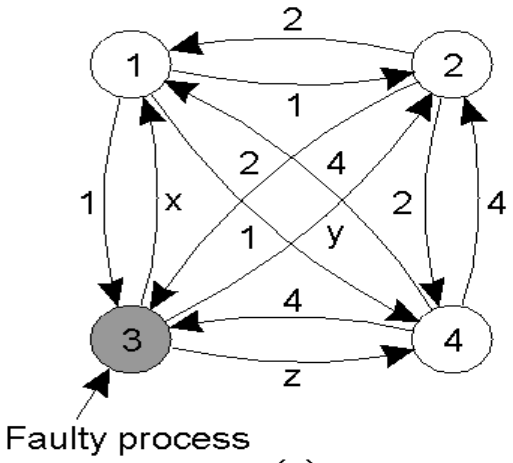
Endless conspiracies, intrigue, and untruthfullness were alleged to be common practice in the ruling circles of the day

That is: it was typical for intentionally wrong and malicious activity to occur among the ruling group. A similar occurrence can surface in a DS, and is known as ‘Byzantine failure’.

Question: how do we deal with such malicious group members within a distributed system?

Agreement in Faulty Systems (1)

How does a process group deal with a faulty member?



(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)

The "Byzantine Generals Problem" for 3 loyal generals and 1 traitor.

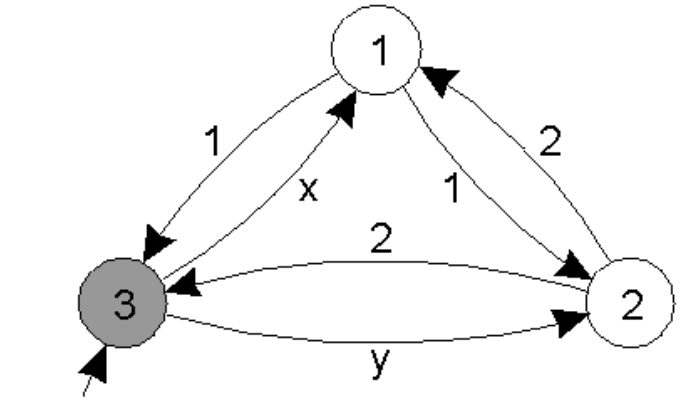
a) The generals announce their troop strengths (in units of 1 kilosoldiers) to the other members of the group by sending a message.

b) The vectors that each general assembles based on (a), each general knows their own strength. They then send their vectors to all the other generals.

c) The vectors that each general receives in step 3. It is clear to all that General 3 is the traitor. In each 'column', the majority value is assumed to be correct.

Agreement in Faulty Systems (2)

Warning: the algorithm does not always work!



(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 algorithm as in previous slide, except now with 2 loyal generals and 1 traitor. Note: It is no longer possible to determine the majority value in each column, and the algorithm has failed to produce agreement.

It has been shown that for the algorithm to work properly, *more* than two-thirds of the processes have to be working correctly. That is: if there are F faulty processes, we need $2F + 1$ functioning processes to reach agreement.

2 .Reliable Client/Server Comms.

In addition to process failures, a communication channel may exhibit crash, omission, timing, and/or arbitrary failures.

In practice, the focus is on masking *crash* and *omission* failures.

For example: the point-to-point TCP masks omission failures by guarding against lost messages using ACKs and retransmissions. However, it performs poorly when a crash occurs (although a DS may try to mask a TCP crash by automatically re-establishing the lost connection).

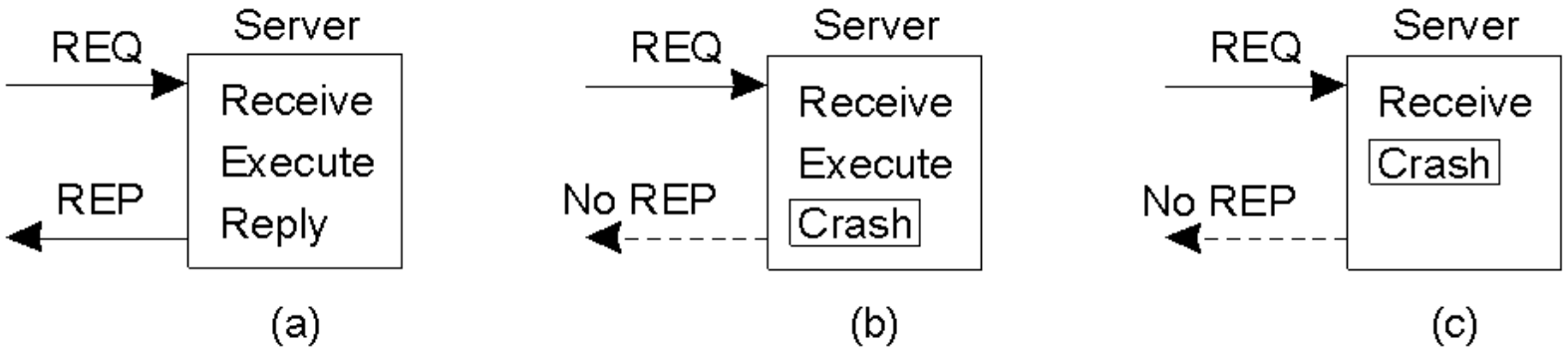
Example: RPC Semantics and Failures

The RPC mechanism works well as long as both the client and server function perfectly. (the higher level)

Five classes of RPC failure can be identified:

1. *The client cannot locate the server*, so no request can be sent.
2. *The client's request to the server is lost*, so no response is returned by the server to the waiting client.
3. *The server crashes after receiving the request*, and the service request is left acknowledged, but undone.
4. *The server's reply is lost on its way to the client*, the service has completed, but the results never arrive at the client
5. *The client crashes after sending its request*, and the server sends a reply to a newly-restarted client that may not be expecting it.

The Five Classes of Failure (1)



A server in client-server communication.

a) The normal case.
b) Crash *after* service execution.
c) Crash *before* service execution.

The Five Classes of Failure (2)

An appropriate exception handling mechanism can deal with a missing server. However, such technologies tend to be very language-specific, and they also tend to be non-transparent.

Dealing with lost request messages can be dealt with easily using timeouts. If no ACK arrives in time, the message is resent. Of course, the server needs to be able to deal with the possibility of duplicate requests.

The Five Classes of Failure (3)

Server crashes are dealt with by implementing one of three possible implementation philosophies:

At least once semantics: a guarantee is given that the RPC occurred at least once, but (also) possibly more that once. (double withdrawals possible)
At most once semantics: a guarantee is given that the RPC occurred at most once, but possibly not at all. (nothing happened)
No semantics: nothing is guaranteed, and client and servers take their chances! (upd comm.)

It has proved difficult to provide ***exactly once semantics***.

The Five Classes of Failure (4)

Lost replies are difficult to deal with.
Why was there no reply? Is the server *dead*, *slow*, or did the reply just go *missing*? Need to wait for a while?

A request that can be repeated any number of times without any nasty side-effects is said to be *idempotent*. (For example: a read of a static web-page is said to be idempotent).

Nonidempotent requests (for example, the electronic transfer of funds) are a little harder to deal with. A common solution is to employ *unique sequence numbers*. Another technique is the inclusion of additional bits in a retransmission to identify it as such to the server.

The Five Classes of Failure (5)

When a client crashes, and when an 'old' reply arrives, such a reply is known as an *orphan*.

Four orphan solutions have been proposed:

extermination (the orphan is simply killed-off),
reincarnation (each client session has an *epoch* associated with it, making orphans easy to spot),
gentle reincarnation (when a new epoch is identified, an attempt is made to locate a requests owner, otherwise the orphan is killed), and,
expiration (if the RPC cannot be completed within a standard amount of time, it is assumed to have expired).

In practice, however, none of these methods are desirable for dealing with orphans. Research continues …

3. Reliable Group Communication

Reliable multicast services guarantee that all messages are delivered to all members of a process group.

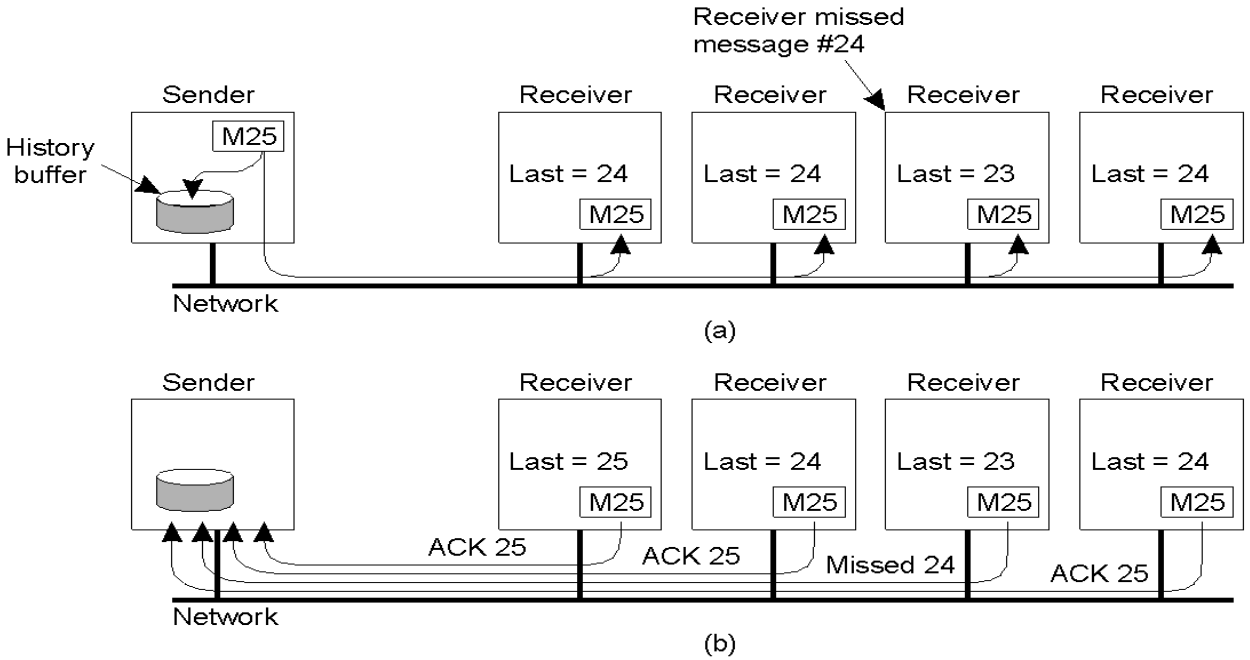
Sounds simple, but is surprisingly *tricky* (as multicasting services tend to be *inherently* unreliable).

For a small group, multiple, reliable point-to-point channels will do the job, however, such a solution *scales poorly* as the group membership grows. Also:

What happens if a process *joins* the group during communication?

Worse: what happens if the sender of the multiple, reliable point-to-point channels *crashes* half way through sending the messages?

Basic Reliable-Multicasting Schemes



This is a simple solution to reliable multicasting when *all receivers are known* and are assumed *not to fail*. The sending process assigns a sequence number to outgoing messages (making it easy to spot when a message is missing).

a) Message transmission – note that the third receiver is expecting 24.

b) Reporting feedback – the third receiver informs the sender.

c) But, how long does the sender keep its *history-buffer* populated?

d) Also, such schemes **perform poorly** as the group grows … there are *too many* ACKs.

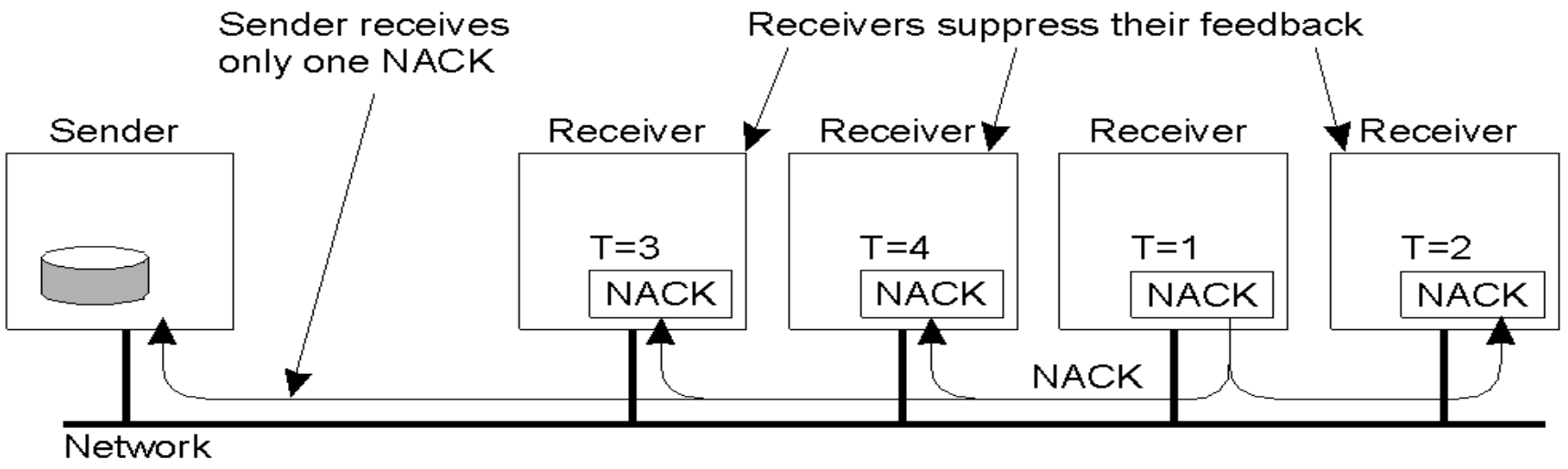
SRM: Scalable Reliable Multicasting

Receivers *never* acknowledge successful delivery.
Only missing messages are reported.
NACKs are multicast to all group members.

This allows other members to suppress their feedback, if necessary.

To avoid "retransmission clashes," each member is required to wait a random delay prior to NACKing.

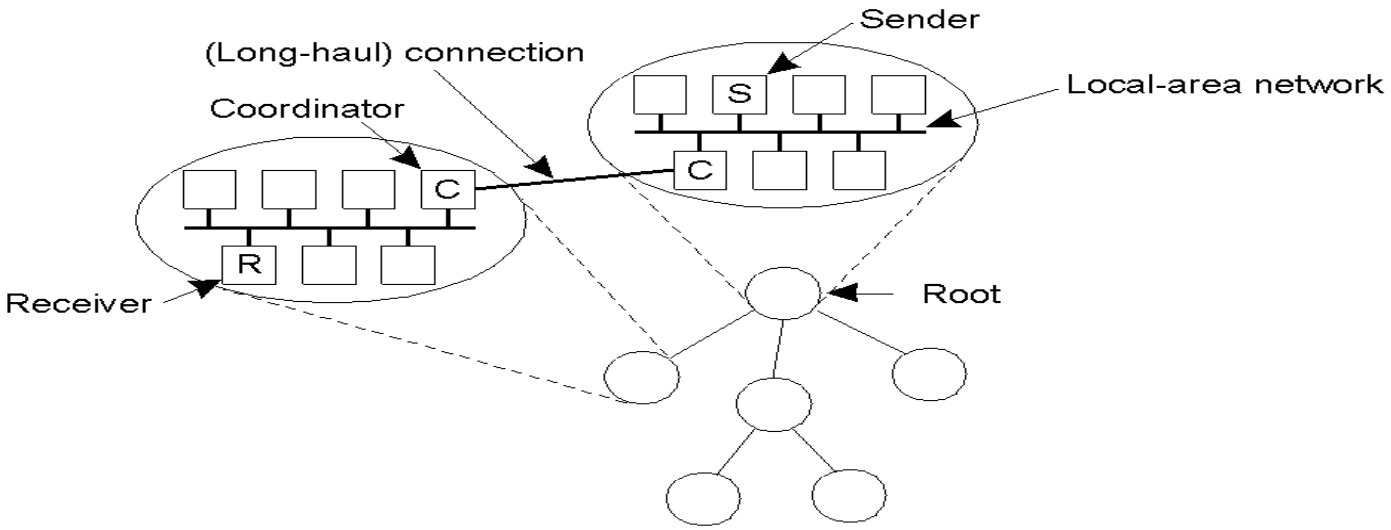
Nonhierarchical Feedback Control



Feedback Suppression – reducing the number of feedback messages to the sender (as implemented in the *Scalable Reliable Multicasting Protocol*).

Successful delivery is never acknowledged, only missing messages are reported (NACK), which are multicast to all group members. If another process is about to NACK, this feedback is suppressed as a result of the first multicast NACK. In this way, only a **single** NACK is delivered to the sender.

Hierarchical Feedback Control



Hierarchical reliable multicasting is another solution, the main characteristic being that it supports the creation of **very large groups**.

a) Sub-groups within the entire group are created, with each *local coordinator* forwarding messages to its children.

b) A local coordinator handles retransmission requests *locally*, using any appropriate multicasting method for small groups.

Atomic Multicasting

There often exists a requirement where the system needs to ensure that all processes get the message, or that none of them get it.

An additional requirement is that all messages arrive at all processes in sequential order.

This is known as the “atomic multicast problem”.

4. Distributed COMMIT

General Goal:

We want an operation to be performed by all group members, or none at all.

[In the case of atomic multicasting, the operation is the delivery of the message.]

There are three types of “commit protocol”: single-phase, two-phase and three-phase commit.

Commit Protocols

One-Phase Commit Protocol:

An elected co-ordinator tells all the other processes to perform the operation in question.

But, what if a process cannot perform the operation? There's no way to tell the coordinator! Whoops …

The solutions:

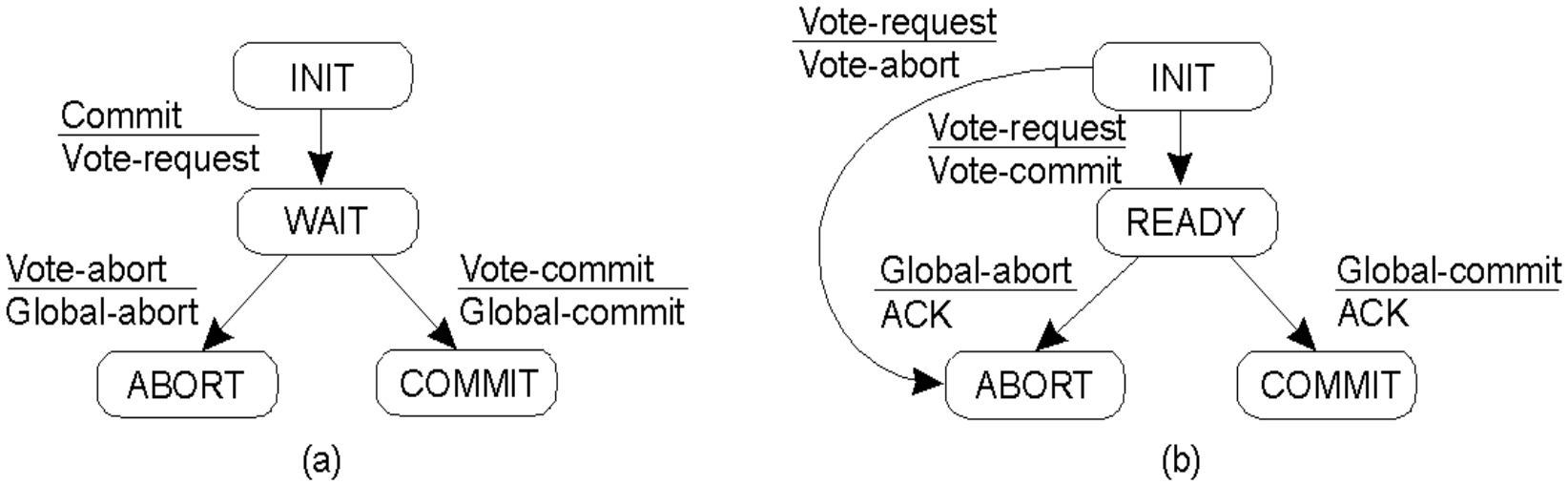
The *Two-Phase* and *Three-Phase Commit Protocols*.

The Two-Phase Commit Protocol

First developed in 1978!!!
Summarized: GET READY, OK, GO AHEAD.

1. The coordinator sends a *VOTE_REQUEST* message to all group members.
2. The group member returns *VOTE_COMMIT* if it can commit locally, otherwise *VOTE_ABORT*.
3. All votes are collected by the coordinator. A *GLOBAL_COMMIT* is sent if all the group members voted to commit. If one group member voted to abort, a *GLOBAL_ABORT* is sent.
4. The group members then **COMMIT** or **ABORT** based on the last message received from the coordinator.

Two-Phase Commit Finite State Machines



a) The finite state machine for the coordinator.
b) The finite state machine for a participant (group member).

Big Problem with Two-Phase Commit

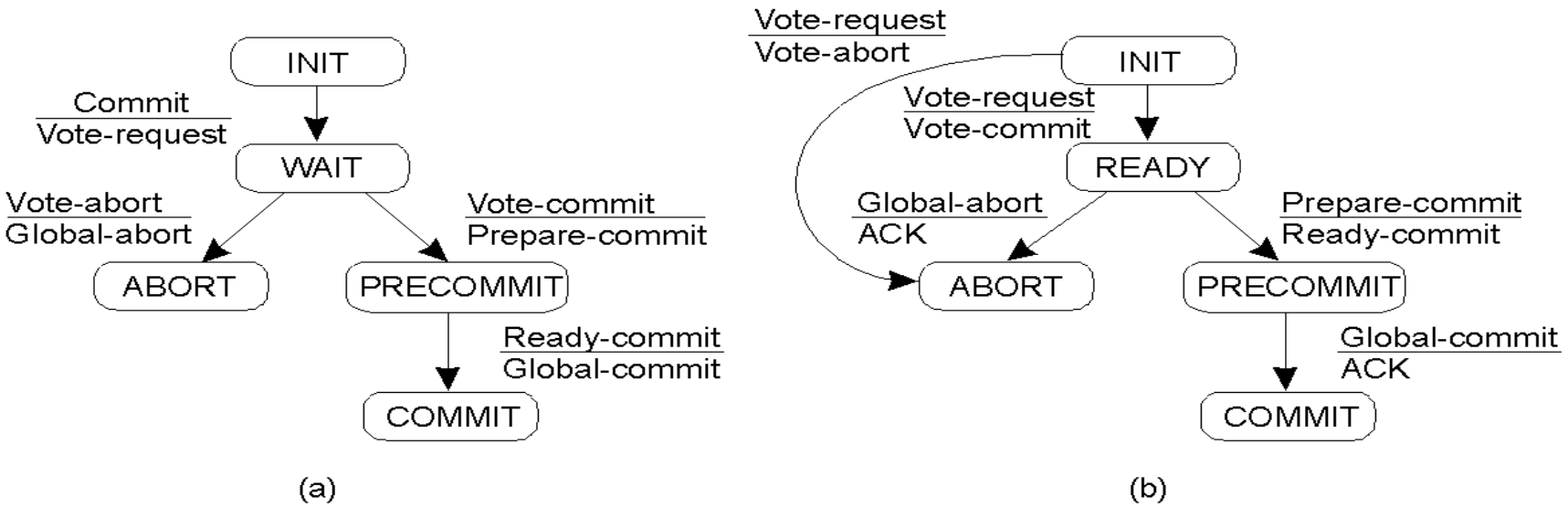
It can lead to both the coordinator and the group members **blocking**, which may lead to the dreaded *deadlock*.

If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers …*

Two-Phase Commit is known as a **blocking-commit protocol** for this reason.

The solution? *The Three-Phase Commit Protocol.*

Three-Phase Commit



a) Finite state machine for the coordinator.
b) Finite state machine for a group member.
c) **Main point**: although 3PC is generally regarded as *better* than 2PC,
it is *not applied often in practice*, as the conditions under which 2PC blocks rarely occur.
a) Refer to the textbook for details on how this works.

5. Recovery Strategies

Once a failure has occurred, it is essential that the process where the failure happened *recovers* to a correct state. Recovery from an error is *fundamental* to fault tolerance.

Two main forms of recovery:

1. **Backward Recovery**: return the system to some previous correct state (using *checkpoints*), then continue executing.
2. **Forward Recovery**: bring the system into a correct state, from which it can then continue to execute.

Forward and Backward Recovery

Disadvantage of Backward Recovery:

Check pointing can be very expensive (especially when errors are very rare).

[Despite the cost, backward recovery is implemented more often. The "logging" of information can be thought of as a type of check pointing.].

Disadvantage of Forward Recovery:

In order to work, all potential errors need to be accounted for *up-front*.

When an error occurs, the recovery mechanism then knows what to do to bring the system *forward* to a correct state.

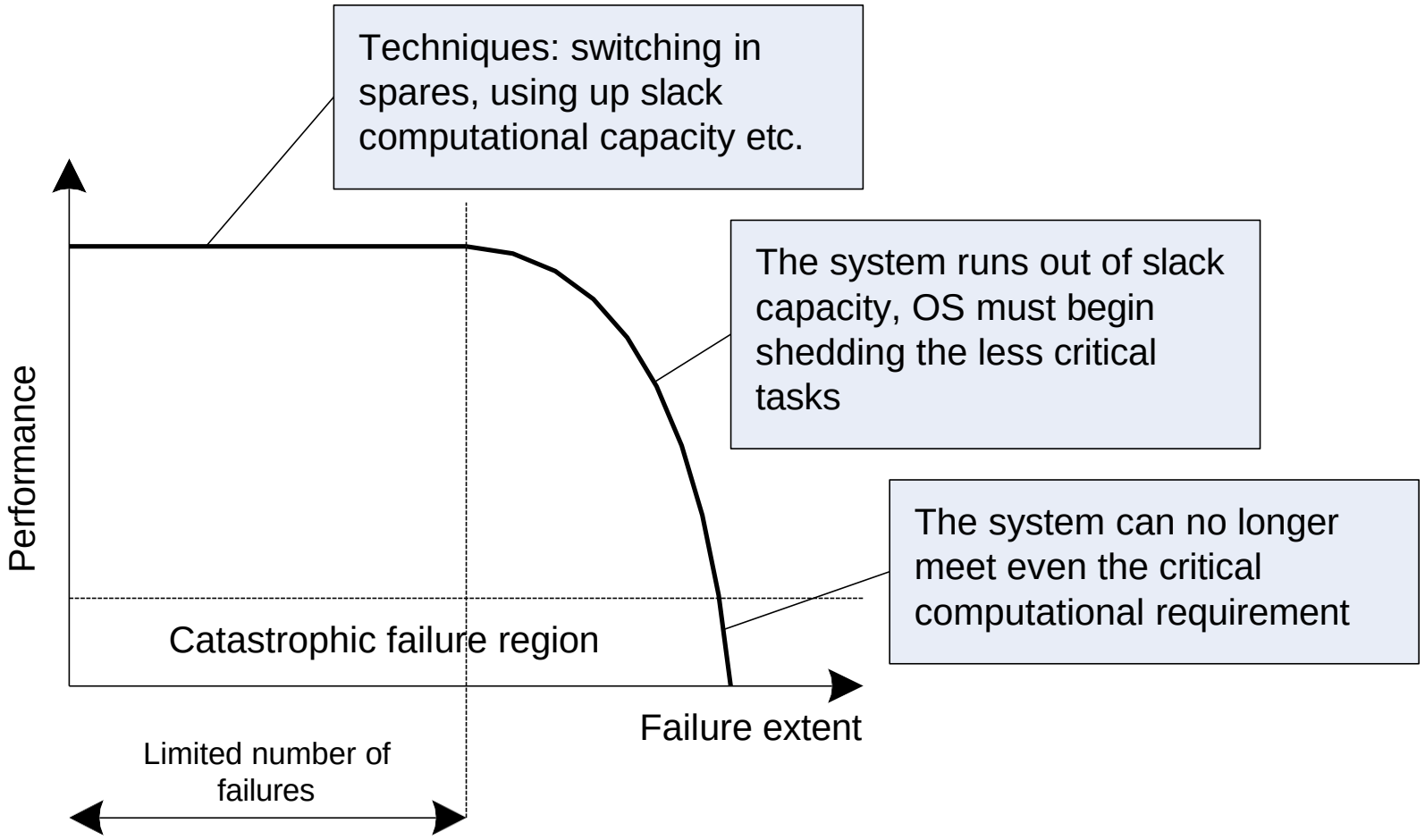
Recovery Example

Consider as an example: Reliable Communications.

Retransmission of a lost/damaged packet is an example of a backward recovery technique.

When a lost/damaged packet can be reconstructed as a result of the receipt of other successfully delivered packets, then this is known as *Erasure Correction*. This is an example of a forward recovery technique.

Properly designed fault-tolerant system



How Is Software Different from Hardware?

Software unreliability is caused predominantly by design slips, not by operational deviations – we use *flaw* or *bug*, rather than *fault* or *error*

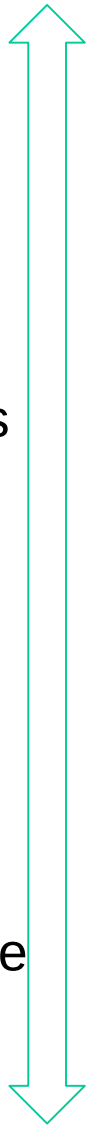
At the current levels of hardware complexity, latent design slips also exist in hardware, thus the two aren't totally dissimilar

Cf. Tolerable flaw or devasting errors

The curse of complexity

The 7-Eleven convenience store chain spent nearly $9M to make its point-of-sale software Y2K-compliant for its 5200 stores
The modified software was subjected to 10,000 tests (all successful)
The system worked with no problems throughout the year 2000
On January 1, 2001, however, the system began rejecting credit cards, because it "thought" the year was 1901 (bug was fixed within a day)

Software Development Life Cycle



Project initiation
Needs
Requirements
Specifications
Prototype design
Prototype test
Revision of specs
Final design
Coding
Unit test
Integration test
System test
Acceptance test
Field deployment
Field maintenance
System redesign
Software discard

Software flaws may arise at several points within these life-cycle phases

Evaluation by both the developer and customer

Implementation or programming
Separate testing of each major unit (module)
Test modules within pretested control structure

Customer or third-party conformance-to-specs test

New contract for changes and additional features
Obsolete software is discarded (perhaps replaced)

What Is Software Dependability?

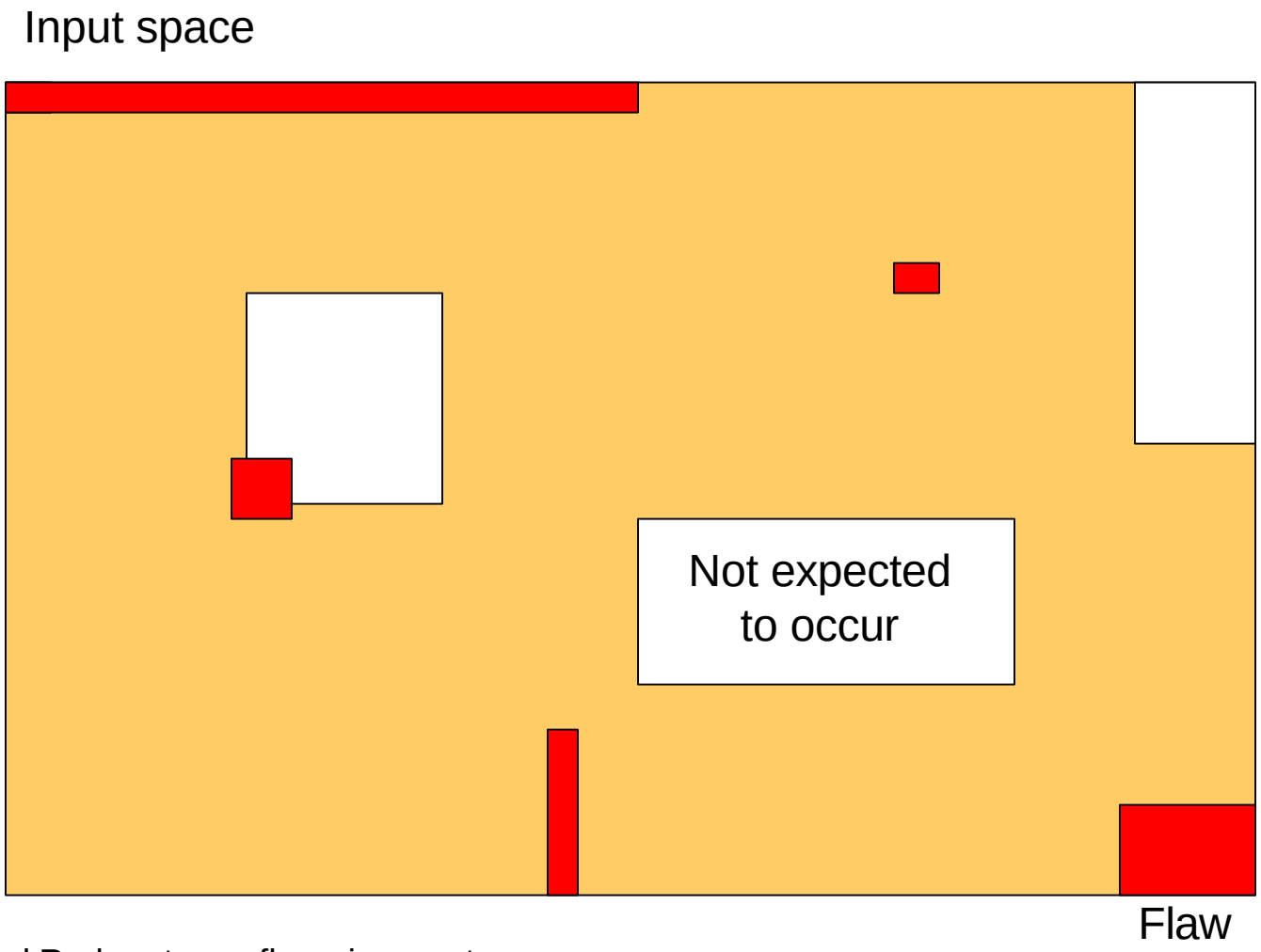
Major structural and logical problems are removed very early in the process of software testing

What remains after extensive verification and validation is a collection of tiny flaws which surface under rare conditions or particular combinations of circumstances, thus giving software failure a statistical nature

Software usually contains one or more flaws per thousand lines of code, with < 1 flaw considered good (linux has been estimated to have 0.1)

The only way to improve software reliability is to reduce the number of residual flaws through more rigorous verification and/or testing

Residual Software Flaws



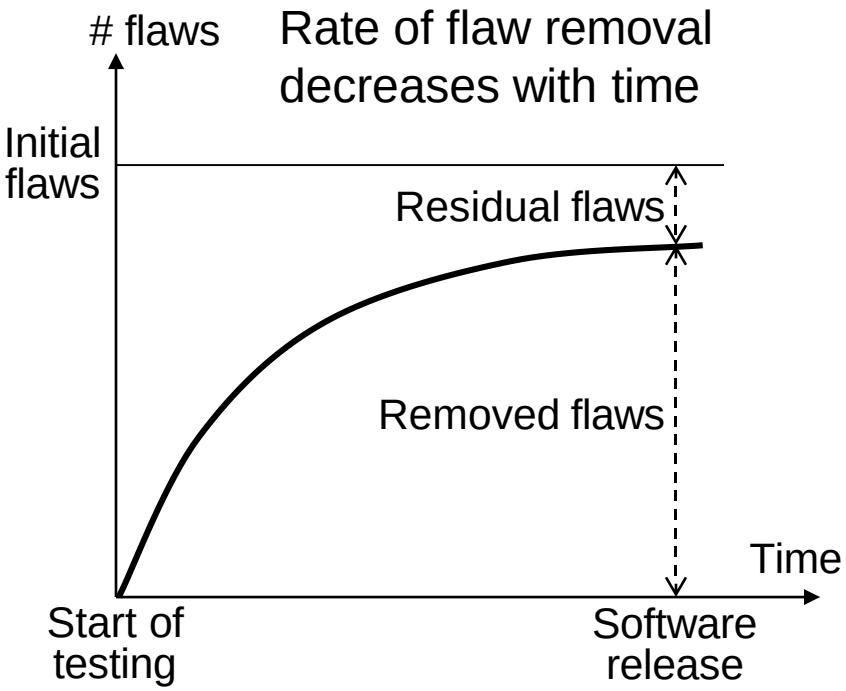
White and Red parts are flaws in a system

While Shooman, M.L., *Reliability of Computer Systems and Networks*,

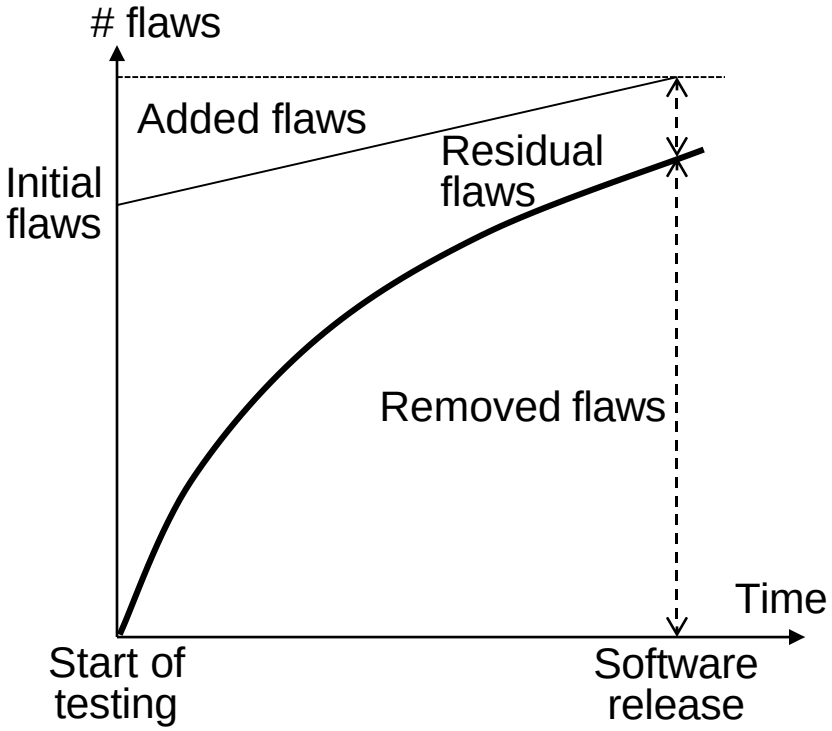
Software Malfunction Models

Software flaw/bug $\Rightarrow$ Operational error $\Rightarrow$ Software-induced failure

"Software failure" used informally to denote any software-related problem



Removing flaws, without generating new ones



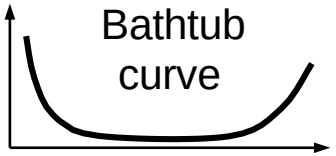
New flaws introduced are proportional to removal rate

The Phenomenon of Software Aging

Software does not wear out or age in the same sense as hardware

Yet, we do observe deterioration in software that has been running for a long time

So, the bathtub curve is also applicable to software



Reasons for and types of software aging:

- Accumulation of junk in the state part (reversible via restoration)
- Long-term cumulative effects of updates (patches and the like)

As the software's structure deviates from its original clean form, unexpected failures begin to occur

Eventually software becomes so mangled that it must be discarded and redeveloped from scratch

Design Checklist for Fault Tolerance

- Identify single points of failure (SPOFs).
- Add redundancy & diversity.
- Monitor health; automate failover.
- Implement graceful degradation policies.
- Test with failure injection/chaos engineering.

Case Study: Netflix Chaos Monkey

Chaos Monkey is a software tool Netflix engineers developed to test the resiliency and recoverability of its Amazon Web Services (AWS) infrastructure.

- Random instance termination in productions.
- Forces auto-healing architecture
- Higher availability during real outages
- Lesson: Be proactive against anticipating and non-anticipating faults.

https://github.com/Netflix/chaosmonkey

Case Study: Netflix Chaos Monkey

Netflix defines their fault-tolerance strategy as: '**The best way to avoid failure is to fail constantly.**' Many larger tech companies practice Chaos Engineering to understand their distributed systems and microservice architectures better.

More on Software Reliability Models

Linearly decreasing flaw removal rate isn't the only option in modeling

Constant flaw removal rate has also been considered, but it does not lead to a very realistic model

Exponentially decreasing flaw removal rate is more realistic than linearly decreasing, since flaw removal rate never really becomes 0

How does one go about estimating the model constants?

- Use handbook: public ones, or compiled from in-house data
- Match moments (mean, 2^{nd} moment, . . .) to flaw removal data
- Least-squares estimation, particularly with multiple data sets
- Maximum-likelihood estimation (a statistical method)

Checking Software Malfunction

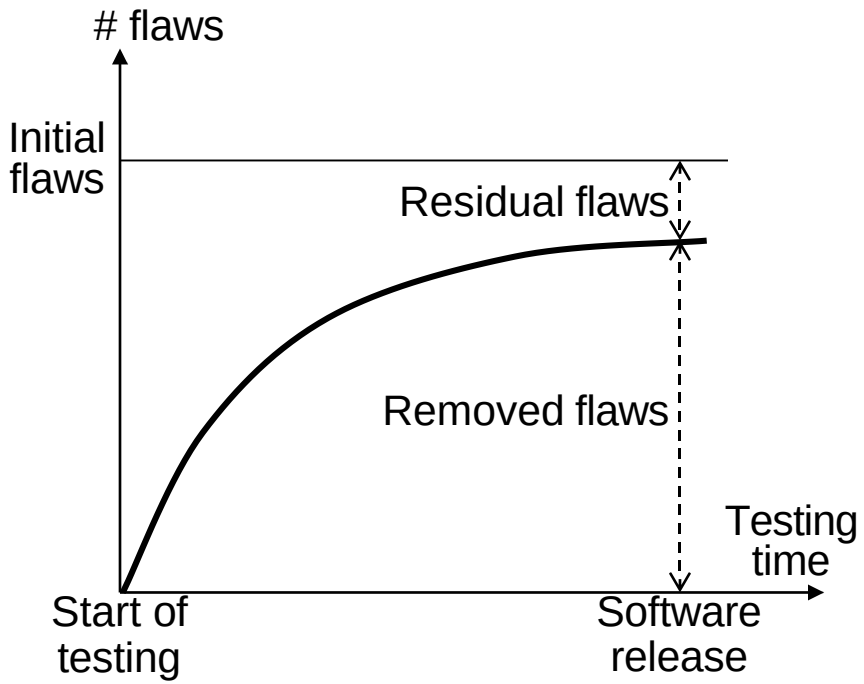
Simply we may calculate

1) Mean Time to Failure

$$MTTF = \frac{Total\ hours\ of\ operation}{Total\ number\ of\ assets\ in\ use}$$

2) Mean Time between Failure

$$MTBF = \frac{Total\ opreational\ time}{Total\ number\ of\ failure}$$



To be continued

- To be continued