Some Basic Concepts on the Design of Fault-Tolerant Distributed Systems

Julián Proenza
Systems, Robotics and Vision Group. UIB. SPAIN
The aim of this presentation

• Discuss some basic concepts related to Dependability and Fault Tolerance in Distributed Systems

• We will focus on those specifically required for a better understanding of the work by our group in the design of dependable distributed embedded systems based on the Controller Area Network (CAN) field bus.
Presentation Outline

1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism
Presentation Outline

1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism
Basic Concepts and Terminology

• A definition for dependability:

“Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers” [Car82]
Basic Concepts
The Dependability Tree (Laprie et al.)

- Concepts organized in 3 classes of notions [Lap92]

- Dependability
  - Attributes
    - Reliability
    - Availability
    - Safety
    - Security
    - Performability
    - Maintainability
    - Testability
  - Impairments
    - Faults
    - Errors
    - Failures
  - Means
    - Procurement
    - Fault Prevention
    - Fault Tolerance
    - Validation
    - Fault Removal
    - Fault Forecasting
Basic Concepts
The Dependability Tree (Laprie et al.)

- First class is the attributes of dependability [Lap92]
Basic Concepts
The Dependability Tree (Laprie et al.)

• The **attributes of dependability** express different properties that can be expected from a dependable system.

• For different system applications, some aspects of dependability are more important than the others.

• The **most relevant attributes** are:
  - Reliability
  - Availability
  - Safety
  - Security
Basic Concepts

The Dependability Tree (Laprie et al.)

- For a system to be **reliable**, it has to exhibit a high probability to provide continuous correct service.

- For a system to be **available**, it has to exhibit a high probability to be ready to provide service (even after failures).

- For a system to be **safe**, it has to be likely to avoid catastrophic consequences on the environment.

- For a system to be **secure**, it has to prevent unauthorized access and/or handling of information.
Basic Concepts
The Dependability Tree (Laprie et al.)

• Second class is the impairments of dependability [Lap92]
Basic Concepts
The Dependability Tree (Laprie et al.)

• The **impairments of dependability** are the *undesired circumstances causing or resulting from un-dependability* [Lap92].

• A system **failure** is a deviation from the service that the system is supposed to provide.

• An **error** is an incorrect value of the system state which can provoke a failure.

• Finally a **fault** is a defect in the design or in the operation of the system that may cause an error.
Basic Concepts
The Dependability Tree (Laprie et al.)

- Example in a Controller Area Network (CAN) fieldbus

**Fault:** “weak” transceiver

**Error:** wrong bit in a received frame [RVA+98]

**Failure:** Inconsistent reception

Inconsistent Message Omission that may cause a general system failure
Basic Concepts
The Dependability Tree (Laprie et al.)

- Third class is the means for dependability [Lap92]
Basic Concepts
The Dependability Tree (Laprie et al.)

• The **means for dependability** are the methods and techniques that can be used to ensure that the system delivers a service on which reliance can be placed.

• In fact, most of times the development of a dependable computing system is more likely to succeed by using a combination of these methods.

• Four classes of such methods can be identified:
Basic Concepts
The Dependability Tree (Laprie et al.)

• **fault prevention:** methods intended to *prevent* the occurrence or introduction of faults;

• **fault tolerance:** methods devised to *design* the system in such a way that it provides a service complying with its specification even in the presence of faults;

• **fault removal:** methods aimed at *reducing* the number and seriousness of faults;

• **fault forecasting:** methods devoted to *estimate* the present number, the future incidence, and the consequences of faults.
1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism
# Faults [Lap92]

<table>
<thead>
<tr>
<th>USUAL LABELLING OF TYPICAL FAULTS</th>
<th>NATURE</th>
<th>PHENOMENOLOGICAL CAUSE</th>
<th>ORIGIN</th>
<th>PERSISTENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Accidental faults</td>
<td></td>
<td>Permanent faults</td>
</tr>
<tr>
<td></td>
<td>Intentional faults</td>
<td>Physical faults</td>
<td></td>
<td>Temporary faults</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Human-made faults</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Physical faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transient faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intermittent faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Design faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Interaction faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Malicious faults</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intrusions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
## Faults [Lap92]

<table>
<thead>
<tr>
<th>USUAL LABELLING OF TYPICAL FAULTS</th>
<th>NATURE</th>
<th>PHENOMENOLOGICAL CAUSE</th>
<th>ORIGIN</th>
<th>PERSISTENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Accidental faults</td>
<td>Physical faults</td>
<td>Human-made faults</td>
</tr>
<tr>
<td>Physical faults</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Transient faults</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intermittent faults</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Design faults</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Interaction faults</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Malicious faults</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intrusions</td>
<td>✓ ✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Note:** The table above outlines the usual labelling of typical faults, their phenomenological cause, and their origin in the context of system boundaries and phases of creation. The persistence of these faults is also indicated.
Presentation Outline

1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism
Failures

• The components of a system are other systems or subsystems. Therefore, a failure can be experienced either by a system or by any of its components.

• The different ways in which a generic system can fail are called failure modes.

• These modes can be characterized according to three viewpoints [Lap92]:
  – consequences on the environment
  – domain
  – perception by the system users
Failures
Classification of failure modes

• Consequences on the environment. In some systems it is possible to clearly differentiate:

  – **benign failures**, which are those whose consequences have a **cost** of the same order of magnitude as the benefit obtained from the system when it works properly;

  – **catastrophic failures**, which are those whose consequences have a **cost incommensurably greater** than the benefit obtained from the system when it works properly.
Failures
Classification of failure modes

• Domain

  - **value failures**, which are the failures that happen when the **value** of the delivered service does not comply with the specification;

  - **timing failures**, which are the failures that happen when the **timing** of the service delivery does not comply with the specification (particularly relevant in Real-Time Systems).
Failures
Classification of failure modes

• **Perception by the system users** (When a system has several users):
  
  – **consistent failures**, those in which all users have the *same perception* of the service provided by the faulty system;
  
  – **inconsistent failures**, those in which the users may have *different perceptions* of the service provided by the faulty system. Because of the way this kind of failures are presented in [LSP82], they are also called *Byzantine failures*. 
A more *practical* classification for the design and analysis of (distributed) systems is a hierarchical one [Pol96h]. Each failure mode is included in the one above it.

- Byzantine or arbitrary failures
- Authentication detectable byzantine failures
- Incorrect computation failures
- Performance failures
- Omission failures
- Crash failures
- Stopping (Fail-stop) failures
• **Byzantine or arbitrary failures.** Lack of any assumption or restriction on the ways the affected system can behave (the user may perceive any kind of effects).
  – AKA *fail-uncontrolled*.
  – Sometimes called *malicious* (equivalent to byzantine) because it includes, for instance,
    • “two-faced” behaviour, which means that the affected system can send a message “fact X is true” to one user and a message “fact X is false” to another user.
    • **forging of messages** of other systems.
Failures
Hierarchical Classification

• **Authentication detectable byzantine failures.** The system may show arbitrary and byzantine behaviour with a **single restriction**: it is **not able to forge messages** of other systems.
  – This means that the faulty system cannot lie about facts that are sent by other systems.
  – This is naturally achieved when authenticated messages are used.
Failures
Hierarchical Classification

• **Authentication detectable byzantine failures.** The system may show arbitrary and byzantine behaviour with a single restriction: it is not able to forge messages of other systems.
  – This means that the faulty system cannot lie about facts that are sent by other systems.
  – This is naturally achieved when authenticated messages are used.

• **Incorrect computation failures:** The system fails to deliver correct results either in the time domain or in the value domain [LMJ91]
  – Without malicious or inconsistent behaviour.
Failures
Hierarchical Classification

• **Performance failures:** The system delivers correct results in the value domain, but it is faulty in the time domain
  – Results may be delivered *early* or *late*.
  – These failures are sometimes called *timing failures*
Failures
Hierarchical Classification

• **Performance failures:** The system delivers correct results in the value domain, but it is faulty in the time domain
  – Results may be delivered *early* or *late*.
  – These failures are sometimes called *timing failures*

• **Omission failures:** The system delivers wrong results in the time domain in such a way that the results are *late with an infinite delay*.
  – Note that subsequent service requests may be properly served
Failures
Hierarchical Classification

• **Crash failures**: The system omits the delivery of results for a service request and for all the subsequent ones.
  – Then it is said that the system has *crashed*.
Failures
Hierarchical Classification

• **Crash failures**: The system omits the delivery of results for a service request and for all the subsequent ones.
  – Then it is said that the system has *crashed*.

• **Stopping (Fail-stop) failures**: The system presents a crash failure and also delivers a constant value service.
  – The constant value delivered may be, e.g., the last correct value, some predetermined value, etc.
  – Very interesting since it permits all users to know that the system has failed.
Failures
Hierarchical Classification

• As indicated, each class comprises those listed below it.

  – Byzantine or arbitrary failures
  – Authentication detectable byzantine failures
  – Incorrect computation failures
  – Performance failures
  – Omission failures
  – Crash failures
  – Stopping (Fail-stop) failures
Failures
Hierarchical Classification

- As indicated, each class comprises those listed below it.

  - Byzantine or arbitrary failures
  - Authentication detectable byzantine failures
  - Incorrect computation failures
  - Performance failures
  - Omission failures
  - Crash failures
  - Stopping (Fail-stop) failures

More restricted
More benevolent
(easier to cope with)
Failures
Failure Semantics

• The failure behaviour of a system is called the system’s failure semantics [Cri89].

• The following definition can be found in [Pol96h].

  – **Failure semantics**: A system exhibits a given failure semantics if the probability of failure modes which are not covered by the failure semantics is sufficiently low.
Failures
Importance of Failure Semantics

• Moreover, assuming a failure semantics is particularly useful when applied to the different subsystems constituting a more complex dependable system.

• This simplifies the system integration and the fault tolerance techniques to be used at the system level.

• Examples can be found in the area of dependable distributed systems:
  – First, assuming byzantine failure semantics for the nodes of a distributed system, the protocol necessary to reach an agreement on a local value is much more complicated than the one necessary to achieve the same under more benevolent failure semantics.
Failures
Importance of Failure Semantics

• Moreover, assuming a failure semantics is particularly useful when applied to the different subsystems constituting a more complex dependable system.

• This simplifies the system integration and the fault tolerance techniques to be used at the system level.

• Examples can be found in the area of dependable distributed systems:
  – Second, when there is a central server that is the only responsible for taking important decisions, its failure semantics should be benevolent in order to facilitate the detection of its failures by the other servers.
Actually, distributed f-t systems are designed assuming a given failure semantics for the constituting servers.

Thus, if a server fails violating its supposed failure sem., then the whole system may fail since the f-t mechanisms are not able to cope with this specific server failure.

This has leaded to propose the concept of assumption coverage [Pow92] which has been defined as follows.

- **Assumption coverage**: is the probability that the possible failure modes defined by the failure semantics of a server proves to be true in real conditions on the fact that the server has failed.
Failures
Importance of the Assumption Coverage

• This **coverage turns out to be critical** in the design of distributed fault-tolerant systems.

• If the failure semantics is assumed to be very restricted, the design of the system is significantly simplified, but the assumption coverage can be too low to be acceptable.

• Due to this problem it is necessary to try to find the right balance between assumption coverage and design complexity each time a new system is designed.
Failures
Increasing the Assumption Coverage

• In practice, the failure semantics is not a simple assumption

• It is enforced through specific mechanisms added at the design phase. The goal: to reach high values for the assumption coverage.

• Some examples:
  – Enforcing authentification detectable byzantine failure semantics for all the servers of a distributed system, is as simple as using authenticated messages for all the communications among them.
  – Crash failure semantics can be enforced by providing some mechanism to detect the server’s errors and another mechanism to disconnect the server upon error detection.
Presentation Outline

1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism
Fault Tolerance Fundamentals

- Another concept from the Dependability Tree that we are going to discuss in more detail is **fault tolerance**
  - **fault tolerance**: methods devised to design the system in such a way that it provides a service complying with its specification even in the presence of faults.

- Special attention to eliminate any **single point of failure**, (any single component within a system whose failure leads to a failure of the whole system).
  - However, it may be **acceptable** for a f-t system to **present a single point of failure** as long as the corresponding component has a sufficiently low probability of failure.
• **Fault tolerance actions** are usually performed in the form of the so-called *error processing* and *fault treatment* [AL81].

  – *Error processing* is aimed at eliminating errors from the computational state before the errors cause the failure of the system.

  – *Fault treatment* is aimed at preventing faults from producing errors again.

```plaintext
FAULT  →  ERROR  →  FAILURE
```

2nd: Fault Treatment  
1rst: Error Processing
Two approaches for performing error processing:

- **Error recovery**, which consists in first detecting the error and second replacing the erroneous state by an error-free state. This replacement may be executed in two different ways [AL81]:
  - Backward recovery
  - Forward recovery

- **Error compensation**, in which the system is designed to perform the computation in a redundant manner and thereby it can produce error-free results even if the redundant internal state is erroneous.
• Discussing the two error recovery ways:
  
  – **Error recovery**, which consists in first detecting the error and second replacing the erroneous state by an error-free state. This replacement may be executed in two different ways [AL81]:
    
    • **Backward recovery**: the erroneous state is replaced by a past state (occupied before error occurrence). In order to be able to do that, the system has to store the correct state at some successive points in time which are called recovery points
    
    • **Forward recovery**: the erroneous state is replaced by a new state that allows the resumption of the system operation, although it may be in a degraded mode
An example of error compensation (the system is designed to perform the computation in a redundant manner and thereby it can produce error-free results even if the redundant internal state is erroneous) is TMR:

- Input 1 → Module 1
- Input 2 → Module 2
- Input 3 → Module 3
- Voter → Output
An **example** of **error compensation** (the system is designed to perform the computation in a redundant manner and thereby it can produce error-free results even if the redundant internal state is erroneous) is **TMR**:

- Note that the “voter” is a **single point of failure**. So if too unreliable it may be necessary to use **redundant voters** (recursive fault tolerance).
**Error Processing. Error Compensation**

- Error detection is **not necessary** for error compensation.
- However, if error detection is not used, faulty components are not going to be detected and available redundancy is going to decrease (called *redundancy attrition*) without being noticed (uncertain current ft capacity of the system).
Error detection is not necessary for error compensation.

However, if error detection is not used, faulty components are not going to be detected and available redundancy is going to decrease (called redundancy attrition) without being noticed (uncertain current fault capacity of the system).

Therefore, it is common to use error detection for the various redundant components after errors are compensated (e.g. in TMR the voter can detect errors).
Fault Tolerance Actions: Fault Treatment

• **Fault tolerance actions** are usually performed in the form of the so-called *error processing* and *fault treatment* [AL81].

  – **Error processing** is aimed at eliminating errors from the computational state before the errors cause the failure of the system.

  – **Fault treatment** is aimed at preventing faults from producing errors again.

```
FAULT → ERROR → FAILURE
```

1rst: Error Processing 2nd: Fault Treatment
The two typical steps of fault treatment:

- **Fault diagnosis**, which is aimed at finding out the cause of each error (both its location and its nature).

- **Fault passivation**, which is aimed at preventing the faults from being activated again. This passivation is done by preventing the faulty components from participating in any other computation of the system.

  • If this action makes the system unable of providing its intended service, then a system reconfiguration may be carried out.
There are two possible approaches to fault tolerance

- **Application-specific fault tolerance**: methods which use knowledge about the specific application.
  - *Reasonableness checks* to detect faults (error detection) and *state estimations* for continued operation (recovery) despite of faults.

- **Systematic fault tolerance** is based on *replication* of components, in which divergence among replicas is used as a criterion for fault detection. Redundant components are used for continued service.
Systematic Fault Tolerance is based on the use of redundancy that can be of four different types:

- Hardware redundancy
- Software redundancy
- Information redundancy
- Time redundancy
FT Fundamentals
Systematic FT and Redundancy

- **Systematic Fault Tolerance** is based on the use of **redundancy** that can be of **four different types**:
  - **Hardware redundancy**: addition of extra hardware, usually for the purpose of either **detecting** errors or **tolerating** faults.
  - **Software redundancy**
  - **Information redundancy**
  - **Time redundancy**
• **Systematic Fault Tolerance** is based on the use of **redundancy** that can be of **four different types**:

  - **Hardware redundancy**
  - **Software redundancy**: addition of extra software, beyond what is needed to perform a given function, to detect and possibly tolerate faults.
  - **Information redundancy**: addition of extra information, beyond that required to implement a given function; for example, error detecting codes.
  - **Time redundancy**: use of extra time to perform the functions of a system such that error detection and often fault tolerance can be achieved.
F T Fundamentals
Independence of Failures Requirement

• Redundancy is always added to cope with specific faults.

• Thus it is vital to ensure that the redundant components are independent with respect to the process of creation and activation of that kind of faults. Examples:

• In a system designed to tolerate physical faults, the redundant modules may be identical given that it is assumed that hardware components fail independently.

• In contrast, a system designed to tolerate design faults cannot use identical modules since a design fault would cause errors (even identical ones) in all of them. So, modules must be designed diversely (design diversity).
• Unfortunately, even if the redundant elements initially have this independency, they can “lose it” during operation.

• The reason is that errors from a faulty element may propagate to (cause the failure of) other elements if measures are not taken.

• So, it is common (specially in distributed systems) to define error containment regions (include in parts of the system mechanisms to prevent the propagation of their errors).
• Even if they are properly designed, none of the mentioned techniques and mechanisms works in all cases.

• They have a **limited** effectiveness due to multiple causes:
  – Faults out of the fault assumptions actually happen
  – Design faults affect the ft mechanisms

• The **coverage** (e.g. error detection coverage, error recovery coverage) is the measure of this effectiveness

• The coverage is estimated through **fault injection** in a real implementation of the system

• The final dependability **dramatically depends** on these coverages
1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism
Techs to Tolerate Software Faults

- **Traditional techniques** for FT were oriented to improve the dependability of **hardware circuits**.

- At the middle 70s the FT community started to pay **attention** to the important contribution of **software faults** to the final dependability of computing systems.

- Software is increasingly complex and therefore unreliable
  - In fact, it is widely accepted as one of the most important sources of unreliability in computer systems

- In critical applications, it is interesting to **tolerate** soft faults
Tolerating Software Faults

Types of Techniques

• Two types of techniques:
  – **Single version** software techniques consist in adding error detection, containment and handling mechanism into the design of a single piece of software in order to improve its tolerance to its own design faults [TP00].
  – **Multi-version** software techniques use multiple versions (or variants) of a piece of software in a structured way to ensure that design faults in one version do not cause system failures [TP00]. They use software *design diversity*, to ensure *fault independence*.

• They are **usually** applied **only** for the system’s components which are considered to be **most prone** to present design faults due to their **higher complexity** [Lyu95].
Multi-version software techniques are based on the assumption that versions which have been built in different manners —i.e. by different designers, using different algorithms and different design tools, etcetera— should fail differently [AC77]

Two basic techniques:
- Recovery Blocks (U. Newcastle Upon Tyne, UK)
- N-Version Programming (NVP) (UCLA, USA)

Other techniques developed later on are essentially variations and combinations of Recovery Blocks and NVP
Tolerating Software Faults
Recovery Blocks

• The f-t program is divided in **blocks** executed sequentially.
• For each block there is a **primary** version and several **alternate** versions that are designed **independently**.
Tolerating Software Faults
Recovery Blocks

- The f-t program is divided in **blocks** executed sequentially
- For each block there is a **primary** version and several **alternate** versions that are designed **independently**.

**Main drawback:**
- An AT must be designed for each block
- The AT must be much simpler (more reliable and faster) than its block
Tolerating Software Faults
N-Version Programming (NVP)

- The program is divided in **segments** executed sequentially.
- For each segment there are **N versions** that are developed **independently**.

```
VERSION 1
  SEGMENT 1
    cc-vector

VERSION 2
  SEGMENT 1
    cc-vector

... 

VERSION N
  SEGMENT 1
    cc-vector

DECISION ALGORITHM (VOTING)

  consensus cc-vector

  SEGMENT 2
  SEGMENT 2
  ... 
  SEGMENT 2
```
Tolerating Software Faults
NVP Aspects

- Advantages over RBs: Unlike AT, **voting is not segment-specific** but quite general & tolerates (masks) faults faster

- A majority of versions must be OK for error compensation

- Not only error compensation **but also recovery** of errors (prevent a fast attrition of redundancy due to transients)
  - **Transitorily faulty versions** use consensus cc-vector as input for the next computation allowing recovery. This may work even for (not too) **late versions**.
  - **Recovery points** are used when a version is lost.

- The N-Version Executive (**NVX**) is the environment controlling the execution of an **NVS**
Tolerating Software Faults
NVP Problems

• **Costly** (as any other multi-version technique)

• Based on the assumption that **different versions fail independently**. Controversy:
  – **Common mode failures.** Due to similar programming mistakes or specification errors (quite frequent)
  – **Consistent comparison problem** (more on this later)

• However, it is accepted nowadays [LPS00] that using **NVP** is “on average” **more reliable** than using a single software version

• Moreover, there are **guidelines and methodologies** aimed at achieving a **desired level of diversity**
Tolerating Software Faults
Decision Algorithm in NVP

• As said, the decision algorithm is not segment-specific. It only depends on the type of data included in the cc-vectors. In the initial developments at UCLA, 3 standard decision functions were distinguished:
  – **Exact match** when bit-by-bit identical data is expected
  – **Numeric match** when small differences are acceptable
  – **Cosmetic match**, only applicable to strings, in which spacing, character substitution, or misspelling errors can be corrected.

• Numeric match **for floating-point numbers**, since in the absence of errors numbers of this type might still be slightly different. It basically performs an average.
Presentation Outline

1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism
Strategies for F-T Systems Design

• **Designing a complete f-t system** using the previous concepts is a **complex task**
  – Intrinsic complexity of designing a system considering “all” its possible faults
  – Many tradeoffs related to costs, available technology, temporal constraints, etcetera

• Advisable to **find systematic approaches** (avoid pitfalls).

• One of the most complete ones is the **Design Paradigm by Prof. Avizienis** [Avi95a], one of the founders of FT
F-T Systems Design
Some general recommendations

• **Before** describing this Paradigm let us mention some **general recommendations** he made establishing some **biological analogies** (dependability in living organisms).

• **First observation**: the **immediate defense mechanisms** of a living body (e.g. immune system, pain sensors, etc.) are **autonomous** (i.e. in “hardware” or “firmware”) and do not require cognitive function (“software”) support.
  – However, higher-level protection (e.g. medication) can be invoked by the brain’s decision.
  – Nevertheless, most computing subsystems built today (e.g. microprocessors or disk drives) lack a sufficient set of local error detection and recovery attributes, and software must be involved in managing their fault conditions.
• **Second observation**: the immediate defense mechanisms discussed above are *distributed* and their generic services are shared by the specialized subsystems of the body.
  – The analogous generic defense mechanisms are error detecting or correcting codes, totally self-checking logic, comparisons, local rollback or voting, etc.

• **Third observation**: *diversity is the key* attribute of a species that *protects* it *against extinction* due to genetic defects in the individual members, while the diversity among species has assured the continuity of live on earth.
  – The self-evident analogy here is the use of hardware and software design diversity in fault-tolerant systems.
F-T Systems Design
Some general recommendations

• **Fourth observation**: it may be useful to view the dependability aspects of networks of computing systems as similar to those of social structures, in which maintaining a consistent view of common reality and avoiding the distribution of (accidentally or deliberately) contaminated information are major objectives.
  – The analogous techniques are the protocols for fault-tolerant consensus and robust data structures for data integrity.
  – In fact, consensus can be considered as a part of a **more general problem** appearing in fault-tolerant distributed systems: the replica non-determinism problem.
  – We will discuss this problem in more detail later on.
Going back to the Avizienis Design Paradigm we must say it differentiates three activities in the design process:

- Specification
- Design
- Evaluation

For each activity, it identifies several steps
F-T Systems Design
The Avizienis’ Design Paradigm

SPECIFICATION
  1. DEPENDABILITY OF SERVICE
  2. CLASSES OF EXPECTED FAULTS
  3. EVALUATION METHODS

DESIGN
  1. SYSTEM PARTITIONING
    - CHOOSE CONTAINMENT BOUNDARIES
    - ALLOCATE REDUNDANT RESOURCES
    - DECIDE DESIGN DIVERSITY
    - ASSIGN GLOBAL F.T. FUNCTIONS
    - SET SUBSYSTEM GOALS
  2. SUBSYSTEM DESIGN
    - ERROR DETECTION
    - FAULT DIAGNOSIS
    - ERROR RECOVERY
    - FAULT REMOVAL
    - RECOVERY VALIDATION
    - EVALUATION
  3. SYSTEMWIDE INTEGRATION

EVALUATION
  1. QUALITATIVE EVALUATION
  2. SIMULATIONS
  3. PROTOTYPING
  4. EXPERIMENTATION
  5. QUANTITATIVE EVALUATION
F-T Systems Design
The Avizienis’ Design Paradigm

SPECIFICATION
1. DEPENDABILITY OF SERVICE
2. CLASSES OF EXPECTED FAULTS
3. EVALUATION METHODS

DESIGN
1. SYSTEM PARTITIONING
   - CHOOSE CONTAINMENT BOUNDARIES
   - ALLOCATE REDUNDANT RESOURCES
   - DECIDE DESIGN DIVERSITY
   - ASSIGN GLOBAL F.T. FUNCTIONS
   - SET SUBSYSTEM GOALS
2. SUBSYSTEM DESIGN
   - ERROR DETECTION
   - FAULT DIAGNOSIS
   - ERROR RECOVERY
   - FAULT REMOVAL
   - RECOVERY VALIDATION
   - EVALUATION
3. SYSTEMWIDE INTEGRATION

EVALUATION
1. QUALITATIVE EVALUATION
2. SIMULATIONS
3. PROTOTYPING
4. EXPERIMENTATION
5. QUANTITATIVE EVALUATION
F-T Systems Design
The Avizienis’ Design Paradigm

SPECIFICATION

(1) DEPENDABILITY OF SERVICE

(2) CLASSES OF EXPECTED FAULTS

(3) EVALUATION METHODS
F-T Systems Design
The Avizienis’ Design Paradigm

SPECIFICATION
- (1) DEPENDABILITY OF SERVICE
- (2) CLASSES OF EXPECTED FAULTS
- (3) EVALUATION METHODS

DESIGN
- (1) SYSTEM PARTITIONING
  - CHOOSE CONTAINMENT BOUNDARIES
  - ALLOCATE REDUNDANT RESOURCES
  - DECIDE DESIGN DIVERSITY
  - ASSIGN GLOBAL F.T. FUNCTIONS
  - SET SUBSYSTEM GOALS

- (2) SUBSYSTEM DESIGN
  - ERROR DETECTION
  - FAULT DIAGNOSIS
  - ERROR RECOVERY
  - FAULT REMOVAL
  - RECOVERY VALIDATION
  - EVALUATION

- (3) SYSTEMWIDE INTEGRATION

EVALUATION
- (1) QUALITATIVE EVALUATION
- (2) SIMULATIONS
- (3) PROTOTYPING
- (4) EXPERIMENTATION
- (5) QUANTITATIVE EVALUATION
F-T Systems Design
The Avizienis’ Design Paradigm

DESIGN

(1) SYSTEM PARTITIONING
- CHOOSE CONTAINMENT BOUNDARIES
  - ALLOCATE REDUNDANT RESOURCES
  - DECIDE DESIGN DIVERSITY
  - ASSIGN GLOBAL F.T. FUNCTIONS
  - SET SUBSYSTEM GOALS

(2) SUBSYSTEM DESIGN
- ERROR DETECTION
  - FAULT DIAGNOSIS
  - ERROR RECOVERY
  - FAULT REMOVAL
  - RECOVERY VALIDATION
  - EVALUATION

(3) SYSTEMWIDE INTEGRATION
F-T Systems Design
The Avizienis’ Design Paradigm

**SPECIFICATION**
- (1) DEPENDABILITY OF SERVICE
- (2) CLASSES OF EXPECTED FAULTS
- (3) EVALUATION METHODS

**DESIGN**
- (1) SYSTEM PARTITIONING
  - CHOOSE CONTAINMENT BOUNDARIES
  - ALLOCATE REDUNDANT RESOURCES
  - DECIDE DESIGN DIVERSITY
  - ASSIGN GLOBAL F.T. FUNCTIONS
  - SET SUBSYSTEM GOALS
- (2) SUBSYSTEM DESIGN
  - ERROR DETECTION
  - FAULT DIAGNOSIS
  - ERROR RECOVERY
  - FAULT REMOVAL
  - RECOVERY VALIDATION
  - EVALUATION
- (3) SYSTEMWIDE INTEGRATION

**EVALUATION**
- (1) QUALITATIVE EVALUATION
- (2) SIMULATIONS
- (3) PROTOTYPING
- (4) EXPERIMENTATION
- (5) QUANTITATIVE EVALUATION
F-T Systems Design
The Avizienis’ Design Paradigm

EVALUATION

(1) QUALITATIVE EVALUATION
(2) SIMULATIONS
(3) PROTOTYPING
(4) EXPERIMENTATION
(5) QUANTITATIVE EVALUATION
1. Dependable Systems.
   1. Basic concepts and terminology
   2. Faults
   3. Failures
   4. Fault tolerance fundamentals
   5. Techniques to tolerate software faults
   6. Strategies for fault-tolerant system design

2. Replica Determinism in Distributed Systems
   1. The replica non-determinism problem
   2. Enforcing replica determinism