Software Systems Architecture

FEUP-M.EIC-ASSO-2023-2024

Ademar Aguiar, Neil Harrison

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Dependability

"*Reliability* is the probability that the system will perform its intended functionality under specified design limits, without failure, over a specified time period."

"*Availability* is the probability that the system is operational at a given time."

"A software system is *robust* if it is able to respond adequately to unanticipated run time conditions."

"A software system is *fault-tolerant* if it is able to respond gracefully to failures at run time."

"*Survivability* is a software system's ability to resist, recognize, recover from, and adapt to mission-compromising threats."

"*Safety* denotes the ability of a software system to avoid failures that will result in loss of life, injury, significant damage to property, or destruction of property."

Bugs!

What is the difference between a fault and a failure?

A fault is a defect in the system

A failure is incorrect behavior of the system that is visible (to users, for example)

Faults can cause failures

A failure can be caused by different faults

- Of by a combination of faults

A failure may be caused by external events too

Which do we care about?

Both, of course

Faults (bugs in the code)

Part of our job is to minimize the faults in the code

How?

Failures

The bottom line

(actually, if a fault NEVER causes a failure, we don't care.)

But we can't guarantee that we eliminated all the faults

- Or eliminated all the faults that will cause failures

And external events can also cause failures

So we have to design our systems to:

- Prevent faults from causing failures
- Prevent external events from causing failures
- Mitigate the effects of failures (that we can't prevent)

Dependability and Architectural Configurations

Avoid single points of failure

Provide back-ups of critical functionality and data

Support nonintrusive system health monitoring

Support dynamic adaptation

- (maybe)

Reliability and Patterns

The architecture patterns selected can have a significant impact on the implementation of reliability

Depends on the tactics selected

- Tactics are approaches to implementing aspects of quality attributes

Details covered in chapter 9 discussion, implementation

Patterns and Reliability

Peer to Peer: good for survivability

Broker: Can help with availability (But the broker is often a single point of failure)

Client-Server: you have to replicate the server (um, and then you might as well use a broker…)

Layers: might support some fault tolerance actions

- Not particularly strong for availability (doesn't hurt, doesn't help)

Pipes and Filters: not strong

- (but maybe you can have alternate filter paths when one dies)
- For availability of a P&F sequence, you need an independent monitor
- Fault correction can be difficult

State: Often central to high availability systems; compatible with it.

Patterns for Fault Tolerant Software

These are kind of architectural

- But not the structural architecture patterns we have seen
- They can be used with them

Some are overall design approaches

Others are more like tactics

Units of Mitigation

How can you keep the whole system from being unavailable when an error occurs?

Divide the system into parts that will contain both any errors and the error recovery. Choose the divisions that make sense for your system. Design the rest of the system around these parts that represent the basic units of error mitigation.

- Units perform self-checks
- Units perform their own recovery, where possible
- Units are barriers to errors – prevent errors from propagating to other units

Units of Mitigation -- examples

General:

- Repositories
- Broker pattern: each server

Banking:

- Loans and Accounts should be separate
- Each account might be its own unit (sub-architectural)

Flight Control:

- Fuel system, each control surface

Correcting Audits

Faulty data causes errors

Detect and correct data errors as soon as possible. Check related date for errors, correct and record the occurrence of the error.

- (“as soon as possible” means before the error causes a failure, of course)

Example:

- Telephone switching systems: endpoint state audits

Fail to a Stable State

Audits

Some ways data can be checked

Check structural properties

- linked lists are correctly linked
- Pointers into lists, etc., are within bounds

Known correlations

- Cross linkages between different data structures are correct
- Software representing hardware state matches real state

Sanity checks

- Values are in range of expected values
- Checksums correct
- Heap data not corrupted

Direct comparison

- Duplicated data are checks to each other

Redundancy

How can we reduce the amount of time between error detection and the resumption of normal operation after error recovery?

Provide redundant capabilities that support quick activation to enable error processing to continue in parallel with normal execution.

Note: there are specific tactics for this

Replication: System Availability

Duplication strategies

- Full duplication
- N + M sparing (Often: N + 1 sparing)

Readiness

- Hot Standby
- Warm Standby
- Cold Standby

Replication: Data Integrity

Data backup

Automated vs. manual

Location: local vs. remote

Recovery Blocks

How can we make sure that processing results in an error-free value, when executing the same code repeatedly will produce the same error?

Provide a diversity of redundancy implementations, either different designs or different coding. Execute them within a framework that checks for acceptable results from the execution of one and try the next secondary block, if the results were unacceptable.

Recovery Block, Structure

```
Ensure:     Successful Execution
By:         executing primary block
Else by:    executing secondary block #1
Else by:    executing secondary block #2
…
Else by:    executing secondary block #n
Else:       trigger exception
```

The details of each step are design time, not architecture

Architecture:

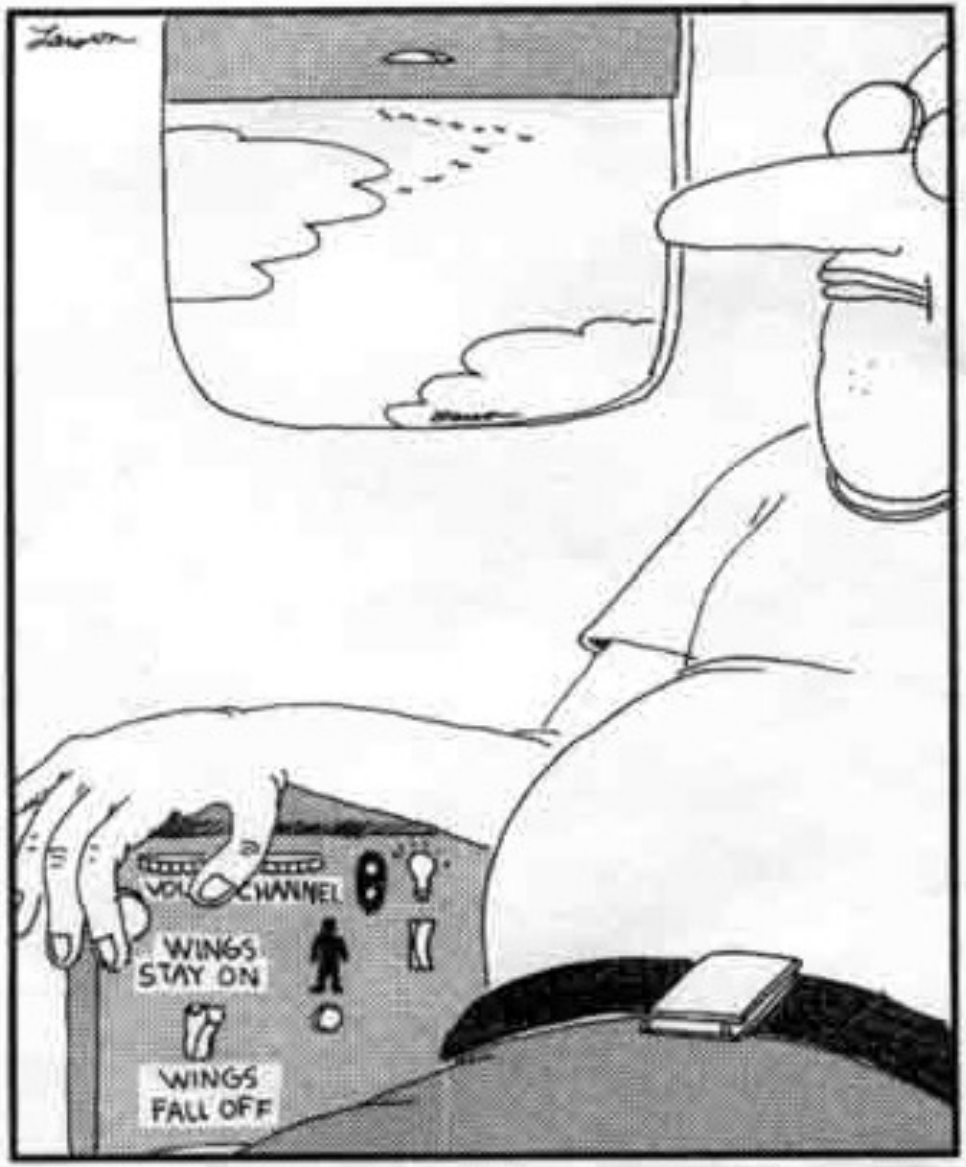
- Decide to use it
- Decide what partitions use it (see also: Units of Mitigation)

Minimize Human Intervention

How can we prevent people from doing the wrong things and causing errors?

Design the system in a way that it is able to process and resolve errors automatically, before they become failures. This speeds error recovery and reduces the risk of procedural errors contributing to system unavailability.

Wrong way to do it



Fumbling for his recline button,
Ted unwittingly instigates a disaster.

Maximize Human Participation

Should the system ignore people totally? That will reduce procedural errors.

Know the users and their abilities. Design the system to enable knowledgeable operating personnel to participate in a positive way toward error detection and error processing. Provide appropriate Maintenance Interfaces and Fault Observer capabilities to give the operators the information that they need to be able to contribute constructively.

Minimize intervention/maximize participation balance:

Can be tricky

Case study: Boeing 737 Max

Ongoing case study: Autonomous driving systems

Maintenance Interface

Should maintenance and application requests be intermingled on the application input and output channels?

No (why not?)

Provide a separate interface to the system for the exclusive or almost exclusive use of maintenance interactions.

Someone in Charge

Anything can go wrong, even during error processing. When this happens the system might stop doing the error processing in addition to not doing the normal processing.

All fault tolerant related activities have some component of the system (“someone”) that is clearly in charge and that has the ability to determine correct completion and the responsibility to take action if it does not complete correctly. If a failure occurs, this component will be sure that the new failure doesn’t stop the system.

Someone In Charge

What are the architectural implications of this principle?

Someone In Charge

What are the architectural implications of this principle?

- Peer to Peer pattern is not a good fit
- Pipes and Filters:
  - Remember the example we showed?

Escalation

What does the system do when its attempt to process an error in an component is not achieving the correct effect?

When recovery or mitigation is failing, escalate the action to the next more drastic action.

Escalating restarts (example)

(Most severe): reboot

Reload: release all memory and reinitialize it.

Cold: release unprotected memory

Warm: preserve memory, but restart all child threads

(Utas, p. 165)

Fault Observer

The system does not stop when errors are detected, it automatically corrects them. How will people know what faults and errors have been detected and processed, both currently and in the past?

Report all errors to the Fault Observer. The fault observer will ensure that all interested parties receive information about the errors that are occurring.

Software Update

The system and its applications must not stop operating, not even to install new software.

Design the ability to change your software into its first release. Enhance this capability as necessary in every subsequent release. Do not assume that software will be an easy problem that you can solve after deployment.

The Big Picture

Dependability aspects require an entire systems approach:

- Hardware
- Software
- Even human behavior

Also may apply to other quality attributes

- For example, human behavior aspects of security

Case Study: MemPool

System crashed because of memory (heap) corruption

- Most likely used after freed
- Might also have been double deletes

Solution: Object Pools

- Didn't prevent the faults, but mitigated their effect

Tactics used

- Pools per Class (decrease chances of making data illegal)
- Free queue (if used after delete, likely shortly afterward)
- Pointers to free queue separate from the data itself
- Checksum of free objects
- Static pool size

Handling the Unusual Cases

Think Use Case Extensions

On Steroids. ☺

For inputs to the system:

- Consider the extreme cases
- Consider the cases that appear to be impossible (don't automatically dismiss them)

Examples of Extreme cases

Oakland earthquake, 1989

Testing our phones

Wall street phone outage

First landing on the moon

Buffer overflows (security issue)

#5ESS: what if a single bit in hardware goes bad?

Y2K Scare

Zug Island…

Beware of:

- “The users would NEVER do that!”
- That event (or combination of events) could NEVER happen!”


RIs
1
2ABC
3DEF
4GHI
5JKL
6MNO
7PQRS
8TUV
9WXYZ
*
0
#
<<<
>>>
Program
Message
Speed Call
Forward
Conference
Transfer
line 2
6031

Risk

Do we have to handle EVERY thing that might go wrong?

We have to balance:

- Probability of occurrence
- Frequency of occurrence
- Consequences when it happens
- Cost & time to implement countermeasures
- Ability to implement countermeasures

If time, we will discuss quality attribute analysis methods

Group Discussion

Patriot Missile Defense System

What things can go wrong?

- Which architectural approaches might be useful? (See summary list)

Notes:

- We won't go into specific tactics
- Notable failure in 1991; consider it, but think about the whole system needs

Summary architectural approaches for reliability

Units of Mitigation

Correcting Audits

Redundancy

Recovery Blocks

Minimize Human Intervention

Maximize Human Participation

Maintenance Interface

Someone in Charge

Escalation

Fault Observer

Software Update