ABSTURZ UNERWÜNSCHT

Muster für fehlertolerante Systeme

Uwe Friedrichsen – Berlin Expert Days – 5. April 2013

ABOUT ME …

Name: Uwe Friedrichsen

Professional experience: Several years …

Focus areas:

- Making teams, projects and systems successful – with a special focus on architecture and agility
- Holistic thinking, connect ideas and concepts, make people think
- New technologies & concepts

Position: CTO at codecentric AG

AGENDA


Why should I care about fault tolerance at all?
It didn't affect me all the years before – so why should it affect me now?


In Scale up fault tolerance was often built into your infrastructure, but with
Scale out
you have to deal with fault tolerance yourself in your applications.

AGENDA

Motivation

Terms and definitions

Fault tolerant mindset

Design for fault tolerance

More stuff …

Summary

FAULT, ERROR & FAILURE

Fault

Defect that can cause an error

Can be caused by incorrect specifications, designs, coding, …

Error

Incorrect behaviour that can cause failures

Can be detected (from inside the system) before it becomes a failure

Manifestation of faults

Failure

Behaviour that does not conform to the specification

Observable from outside the system (i.e. by users)

Caused by errors

- Fault → Error → Failure
- Error detection and handling is a core effort in fault tolerant system desgin

FAILURE CLASSES

Crash failure

System crashes, but worked correctly up to then

Typical failure situation in a scale out scenario

Omission failure

System does not respond to a request

Also typical failure – often caused by infrastructure failures

Timing failure

System does not respond within a specified time frame

Often caused by system overload – also important for scale out

Response failure

Response is wrong

Harder to handle automatically – often requires code changes

Byzantine (arbitrary) failure

System creates arbitrary reponses (at arbitrary points in time)

Very hard to handle – requires complex handling procedures

MTTF, MTTR & MTBF

MTTF

Mean Time To Failure

Average time from start of operation until first failure

MTTR

Mean Time To Repair/Recovery

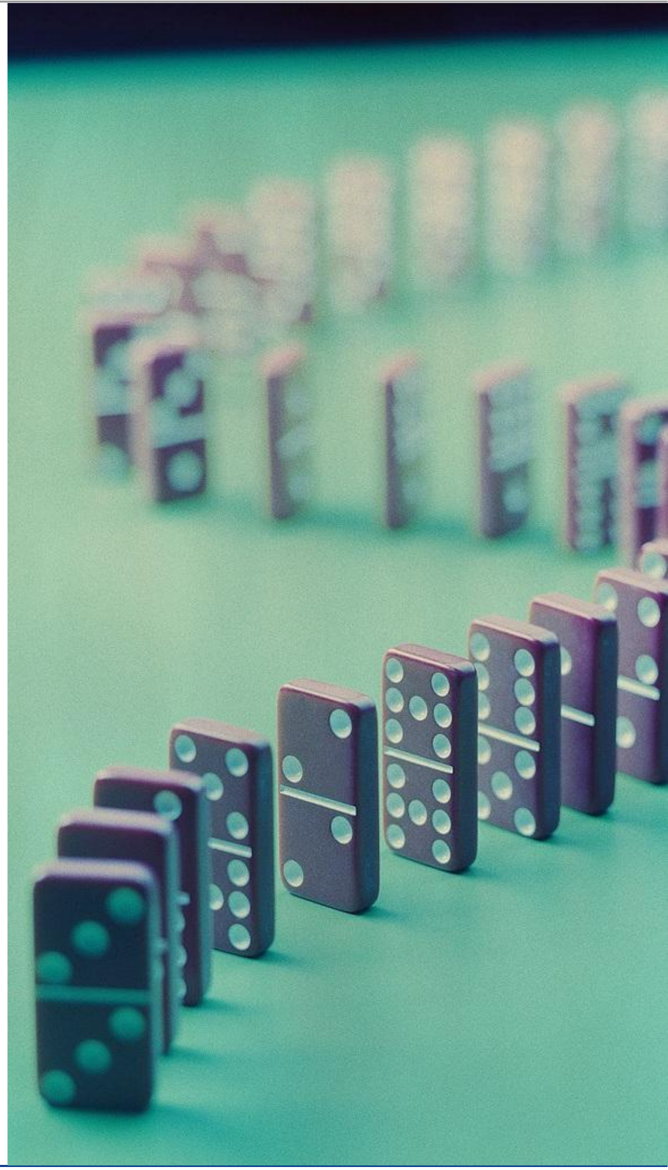
Average time to restore a failing component to operation

MTBF

Mean Time Between Failure

MTBF = MTTF + MTTR

- MTTF usually cannot be influenced (for one node)
- Yet, availability must not be compromised
- Thus, MTTR usually is the important factor



AGENDA

Motivation

Terms and definitions

Fault tolerant mindset

Design for fault tolerance

More stuff …

Summary

Strive for Simplicity

The system should be made as simple as possible (- but no simpler)

Design for Failure

Whatever can go wrong will go wrong

Design incrementally

AGENDA

Motivation

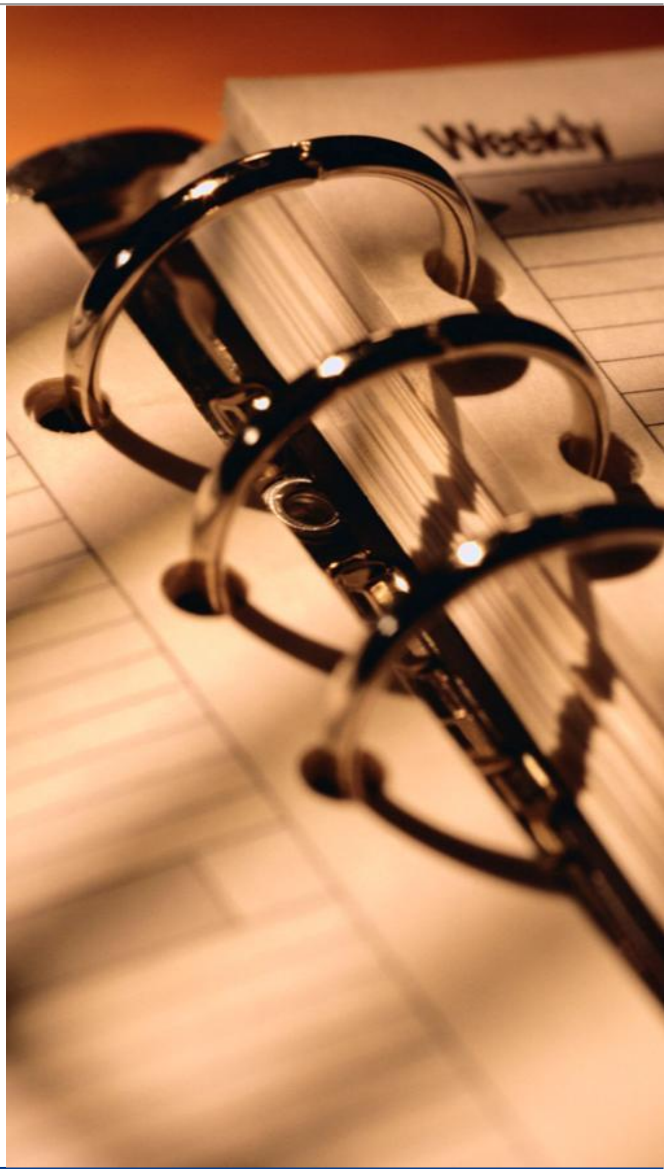
Terms and definitions

Fault tolerant mindset

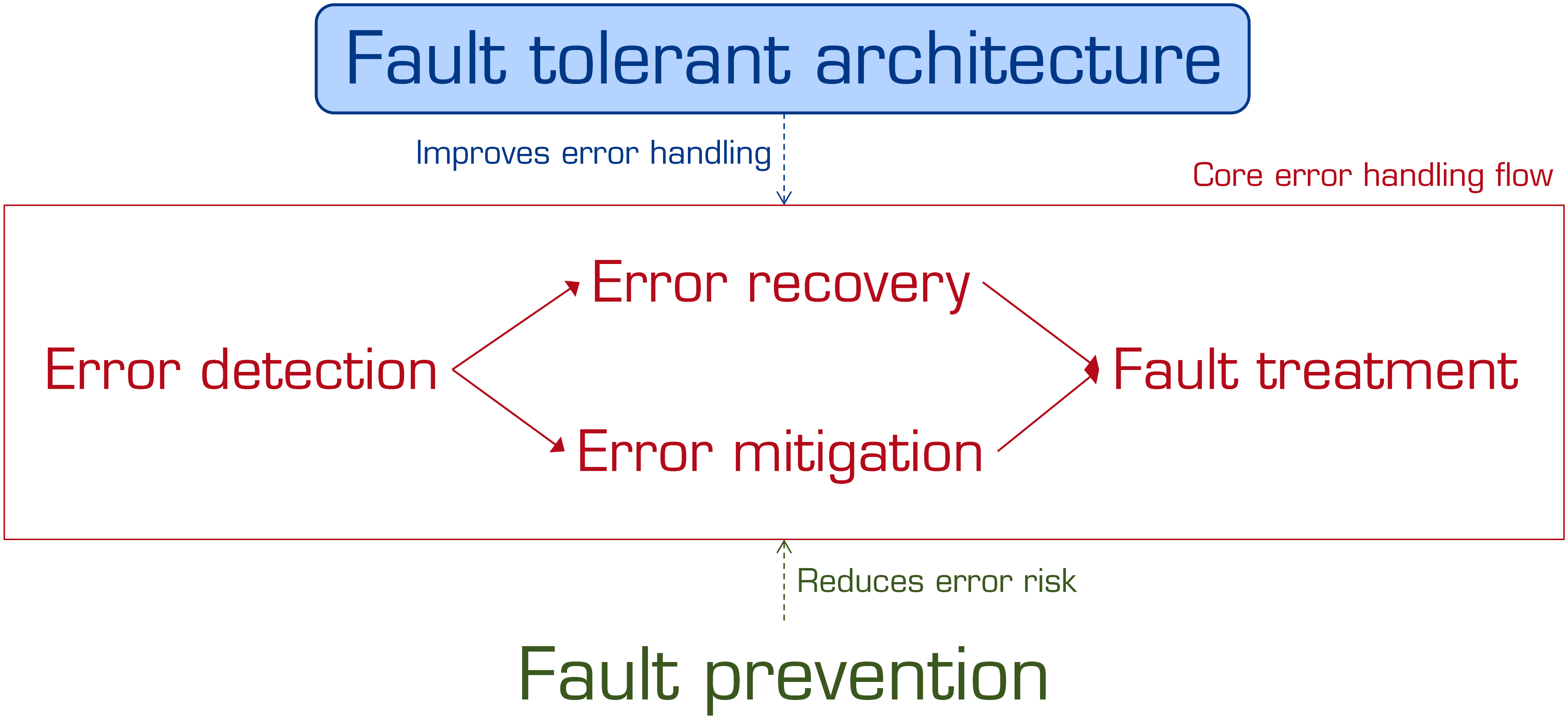
Design for fault tolerance

More stuff …

Summary



A SIMPLE TAXONOMY FOR FAULT TOLERANT DESIGN



UNITS OF MITIGATION

Domain

Architectural pattern

When to use

To prevent the system to fail as a whole

Whenever possible

How to implement

Decouple units/components as much as possible

Implement error checks and barriers at unit boundaries

Let units fail silently if an error is detected

Related Concepts

Redundancy, failover, error handler, …

Tradeoffs

Finding of good units is a non-trivial design task

Balance between added value and added complexity needs to be kept

REDUNDANCY

Domain

Architectural pattern

When to use

The system must not become unavailable

Minimizing MTTR (from an external perspective) is important

How to implement

Provide the component/unit of mitigation several times

Align your solution to the required level of availability

Use infrastructure means if available and suitable

Related Concepts

Failover, recovery blocks, routine excercise, …

Tradeoffs

Balance costs and level of availability carefully

Pure software redundancy needs extra implementation effort

ESCALATION

Domain

Architectural pattern

When to use

Error processing or mitigation important for system to work

Error cannot be treated successfully on local level

How to implement

Design different levels of error handling, each with a more complete view of the system

Plan for more drastic measures to handle error at each level

Use infrastructure built-in propagation techniques if available

Related Concepts

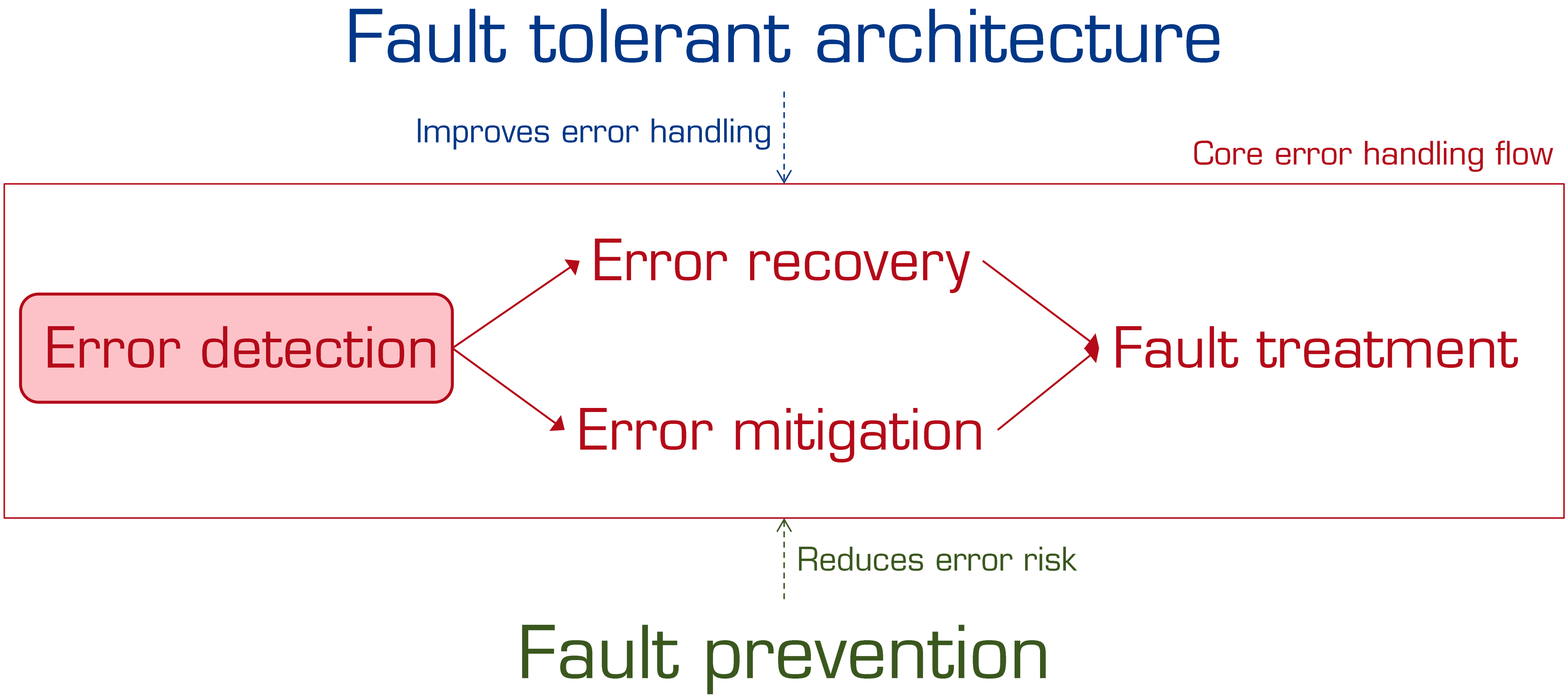
Let it crash, limit retries, rollback, failover, reset, …

Tradeoffs

Finding and implementing a good escalation strategy is complex

Decision when to escalate is often hard

A SIMPLE TAXONOMY FOR FAULT TOLERANT DESIGN



MONITOR

Domain

Error detection

When to use

Continuous availability is important

Failures and crashes need to be detected quickly

How to implement

Create an independent monitor component

Let the monitor share as few resources as possible with the monitored components

Check if out-of-the-box solutions are sufficient, use if applicable

Related Concepts

Acknowledgement, heartbeat, watchdog, supervisor-worker, …

Tradeoffs

Complexity and load of monitored component usually raised

Finding good metrics and escalation thresholds is often hard

DATA VERSIONING

Domain

Error detection

When to use

Always in a scale-out environment

How to implement

Add a version indicator to each single entity

When accessing related entities always check if the versions match

Update the elder entity on the fly to match the newer entity if possible, accept inconsistency otherwise

Related Concepts

Vector clocks, BASE, replication, quorum, routine maintenance

Tradeoffs

Must be implemented explicitly (which is a lot of work)

Sometimes hard to figure out how to repair the outdated entity

ROUTINE MAINTENANCE

Domain

Fault prevention/Error detection

When to use

System needs to run failure-free for long periods

Availabilty is very important

How to implement

Create background jobs that check components and data

Start jobs automatically if possible, otherwise by an operator

Combine findings incrementally with (correcting) fault handlers

Related Concepts

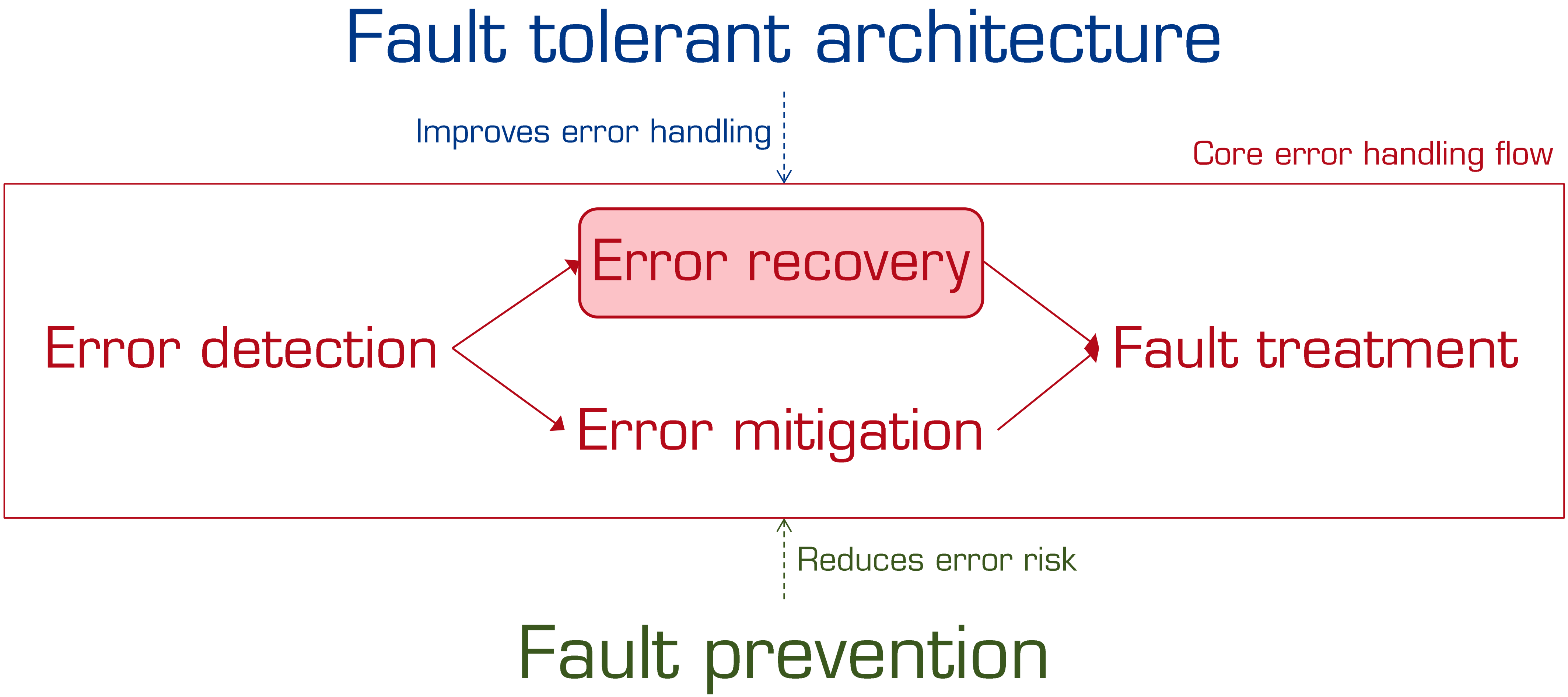
Automation, routine audits, routine exercise, …

Tradeoffs

Can create a lot of information that is hard to handle manually

Cost/benefit analysis is usually needed

A SIMPLE TAXONOMY FOR FAULT TOLERANT DESIGN



ERROR HANDLER

Domain

Error recovery

When to use

An error has been detected and needs to be handled

The system should stay as simple and maintainable as possible

How to implement

Delegate work to a dedicated error handler if an error occurs

Encapsulate all error recovery related code in the error handler

Shift the error handler to a different system part if suitable

Related Concepts

Fault observer, restart, rollback, roll-forward, final handling, …

Tradeoffs

Needs explicit design upfront

Just using catch-blocks or other programming-language-provided constructs is tempting

RETRY, ROLLBACK, ROLL-FORWARD, RESTART, …

Domain

Error recovery

When to use

An error has occured and the system needs to recover

Depending on the severity of the error and data different strategies can be applied

How to implement

Retry if it seems to be a transient error (but limit retries)

Rollback to a checkpoint if you have the data available

Roll-Forward to a reference point if you don‘t have the data, the time or the error is sticky

Use restart if nothing else helps (the error is really hard)

Related Concepts

Escalation, checkpoint, reference point, limit retries, …

Tradeoffs

Escalation strategy needs to be balanced

FAILOVER

Domain

Error recovery

When to use

An error has occured and the system needs to recover quickly

Fault handling will take too long and compromise availability

How to implement

Provide component redundant

Switch to spare component in case of error

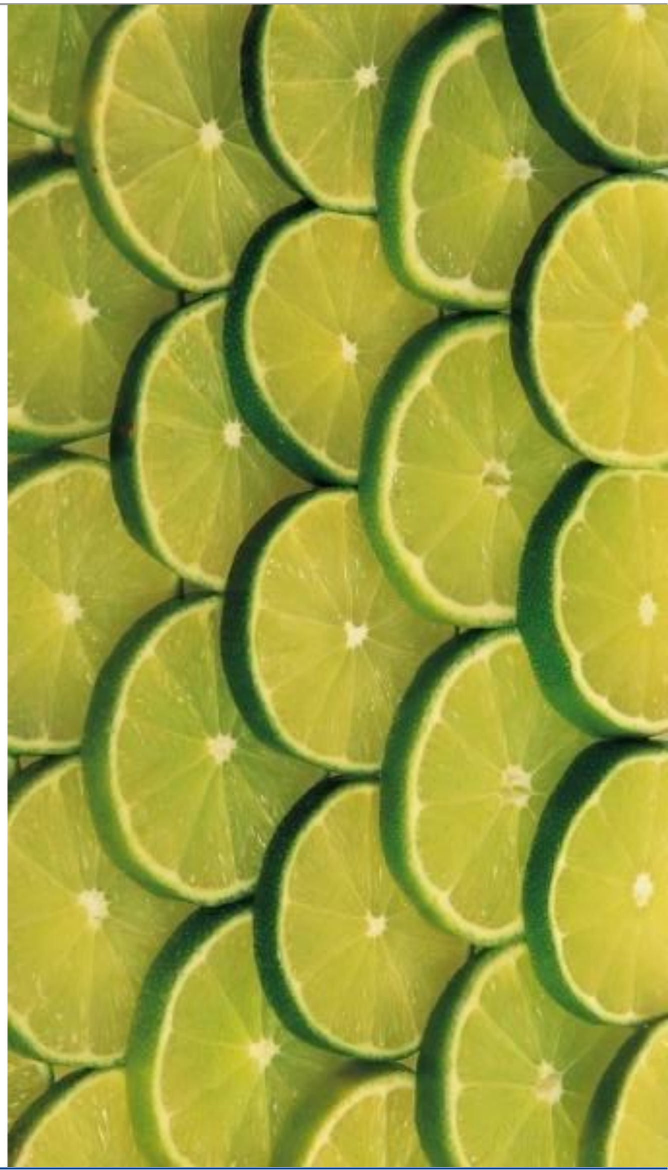
Use infrastructure solutions if suitable

Related Concepts

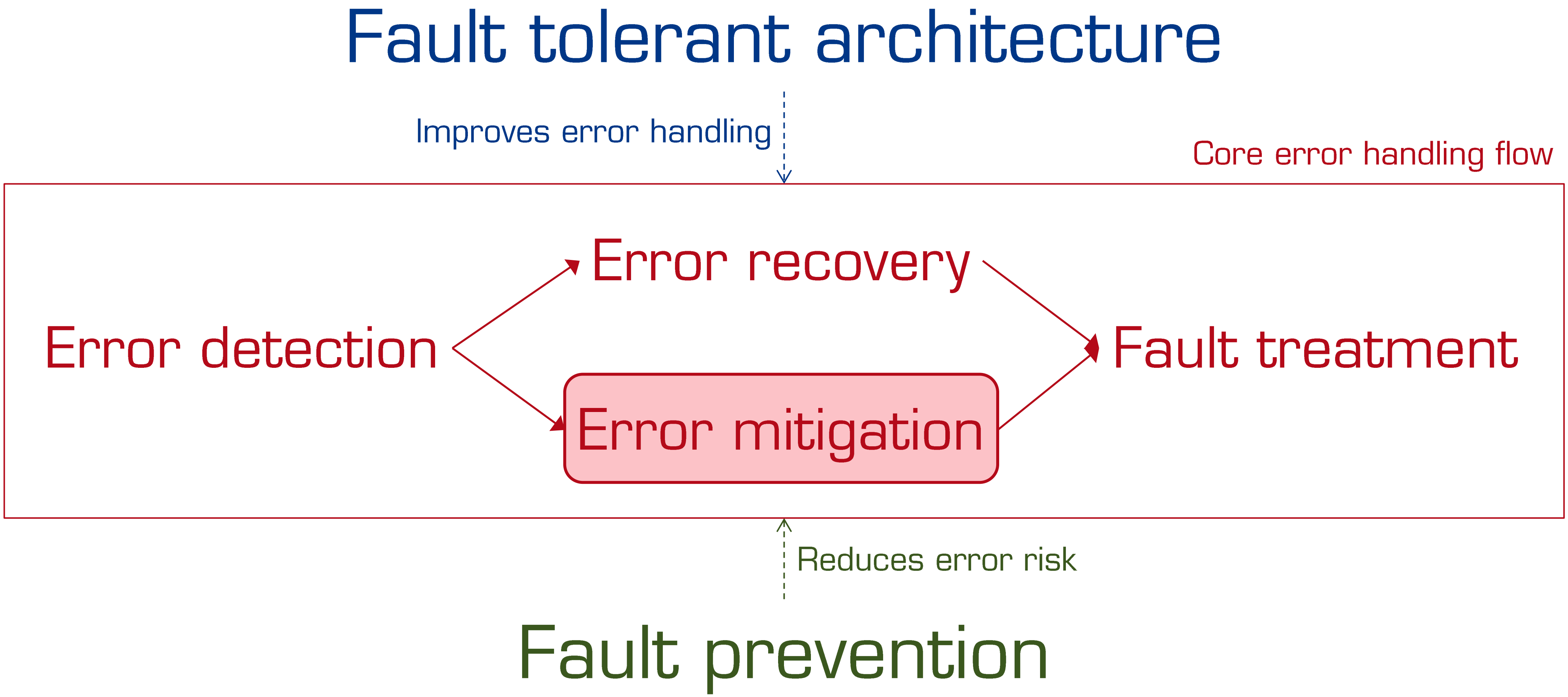
Redundancy, escalation, restart, …

Tradeoffs

Different failover strategies (hot standby, cold standby, …) affect costs and effort – cost/benefit analysis usually required



A SIMPLE TAXONOMY FOR FAULT TOLERANT DESIGN



SHED LOAD

Domain

Error mitigation

When to use

System must keep up service even under high load

Long response times are worse than rejecting a request upfront

How to implement

Monitor system load and response times

Implement gatekeeper at system entry

Let gatekeeper reject requests if monitored response times and load increase

Related Concepts

Share load, finish work in progress, fresh work before stale, …

Tradeoffs

Consequences of dropping requests need to be considered well

MARKED DATA

Domain

Error mitigation

When to use

System must work reliable even in presence of corrupted data

Corrupted data cannot be fixed when detected

How to implement

Flag data to mark it as faulty

Make sure flagged data is not used by rest of the system

Use common markers if suitable (NaN, null, …)

Related Concepts

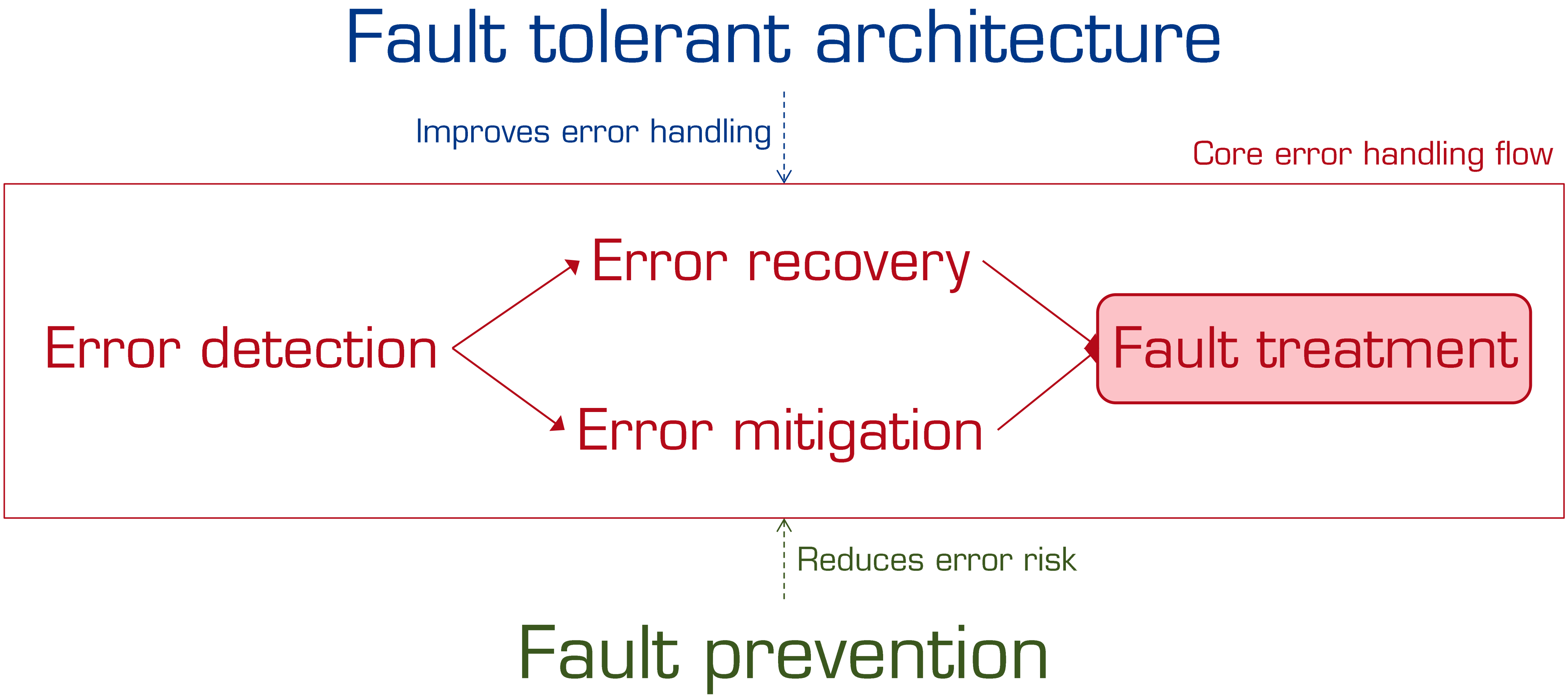
Routine audits, error correcting codes, …

Tradeoffs

Ignoring marked data is a lot of manual implementation effort

Hard to implement à posteriori into an existing system

A SIMPLE TAXONOMY FOR FAULT TOLERANT DESIGN



SMALL PATCHES

Domain

Fault treatment

When to use

Fault correction needs a system update (i.e. software patch)

Risk of introducing new faults by the update should be as small as possible

How to implement

Deliver as small patches as possible

Use continuous delivery techniques

Automate your delivery chain to keep update effort low

Related Concepts

Continuous delivery, let sleeping dogs lie, root cause analysis, …

Tradeoffs

Without a solid delivery chain automation small patches will be extremely expensive and error prone

AGENDA

Motivation

Terms and definitions

Fault tolerant mindset

Design for fault tolerance

More stuff …

Summary

WHAT I DIDN'T TALK ABOUT …

Lots of patterns

Maintenance interface, someone in charge, fault correlation, voting, checksums, leaky bucket container, quarantine, data reset, overload toolboxes, queue for resources, slow it down, fresh work before stale, add jitter, …

Recovery oriented computing

Microreboot

Undo/Redo

Crash-only software

Highly scalable systems

Many complementary patterns and priciples

And many more …

Fault tolerance in other areas (real-time, extreme conditions)

Detection of and recovery from byzantine errors

Theoretical foundations, advanced techniques and algorithms

MORE TO READ …

[1] Robert S. Hanmer,
Patterns for fault tolerant software,
Wiley, 2007

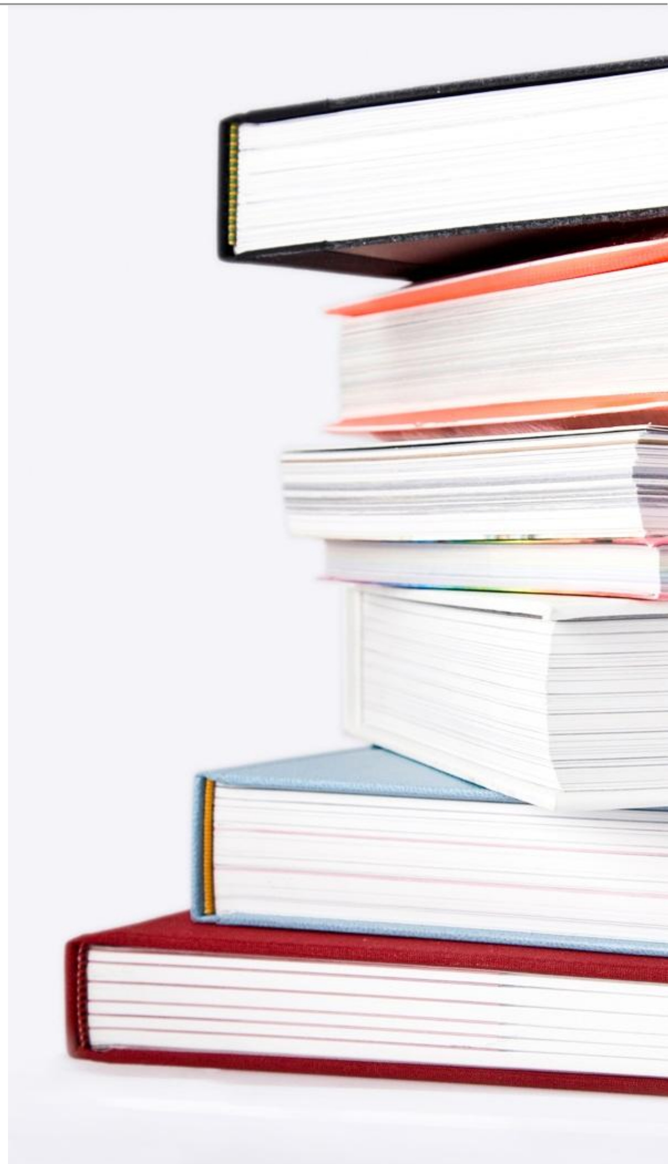
[2] The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project, http://roc.cs.berkeley.edu/

[3] James Hamilton, On Designing and Deploying Internet-Scale Services, 21st LISA Conference 2007

[4] Zaipeng Xie, Hongyu Sun and Kewal Saluja,
A survey of software fault tolerance techniques,
http://www.pld.ttu.ee/IAF0030/Paper_4.pdf

[5] Michael R. Lyu (Ed.), Handbook of Software Reliability Engineering, McGraw-Hill 1996, http://www.freebookzone.com/goto.php?bkcls=se&bkidx=81&lkidx=1

[6] Andrew Tanenbaum, Marten van Steen,
Distributed Systems. Principles and Paradigms,
Prentice Hall, 2nd Edition, 2006



AGENDA

Motivation

Terms and definitions

Fault tolerant mindset

Design for fault tolerance

More stuff …

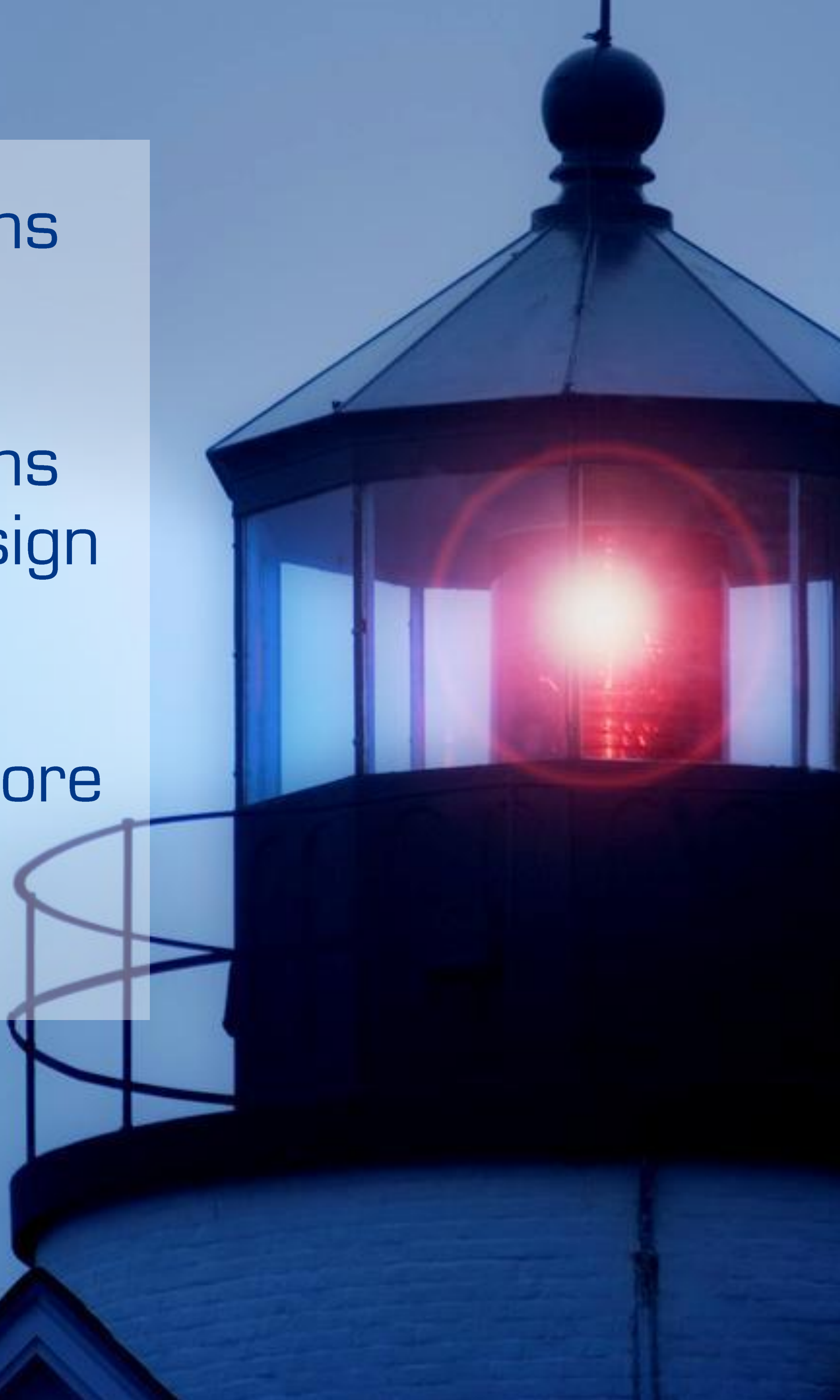
Summary

Scale out and distributed systems are becoming mainstream

Scale out and distributed systems require explicit fault tolerant design

Infrastructure provided fault tolerance does not suffice anymore

Right mindset is essential



THANK YOU FOR YOUR ATTENTION!

Uwe Friedrichsen
CTO

codecentric AG
Merscheider Straße 1
42699 Solingen

uwe.friedrichsen@codecentric.de
tel +49 (0) 212 . 23 36 28 10
fax +49 (0) 212 . 23 36 28 79
mobil +49 (0) 160 . 90 62 66 00

@ufried
www.codecentric.de
blog.codecentric.de
www.meettheexperts.de

QUESTIONS & DISCUSSIONS