2.– 5. September 2013
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Wir bauen uns ein fehlertolerantes System

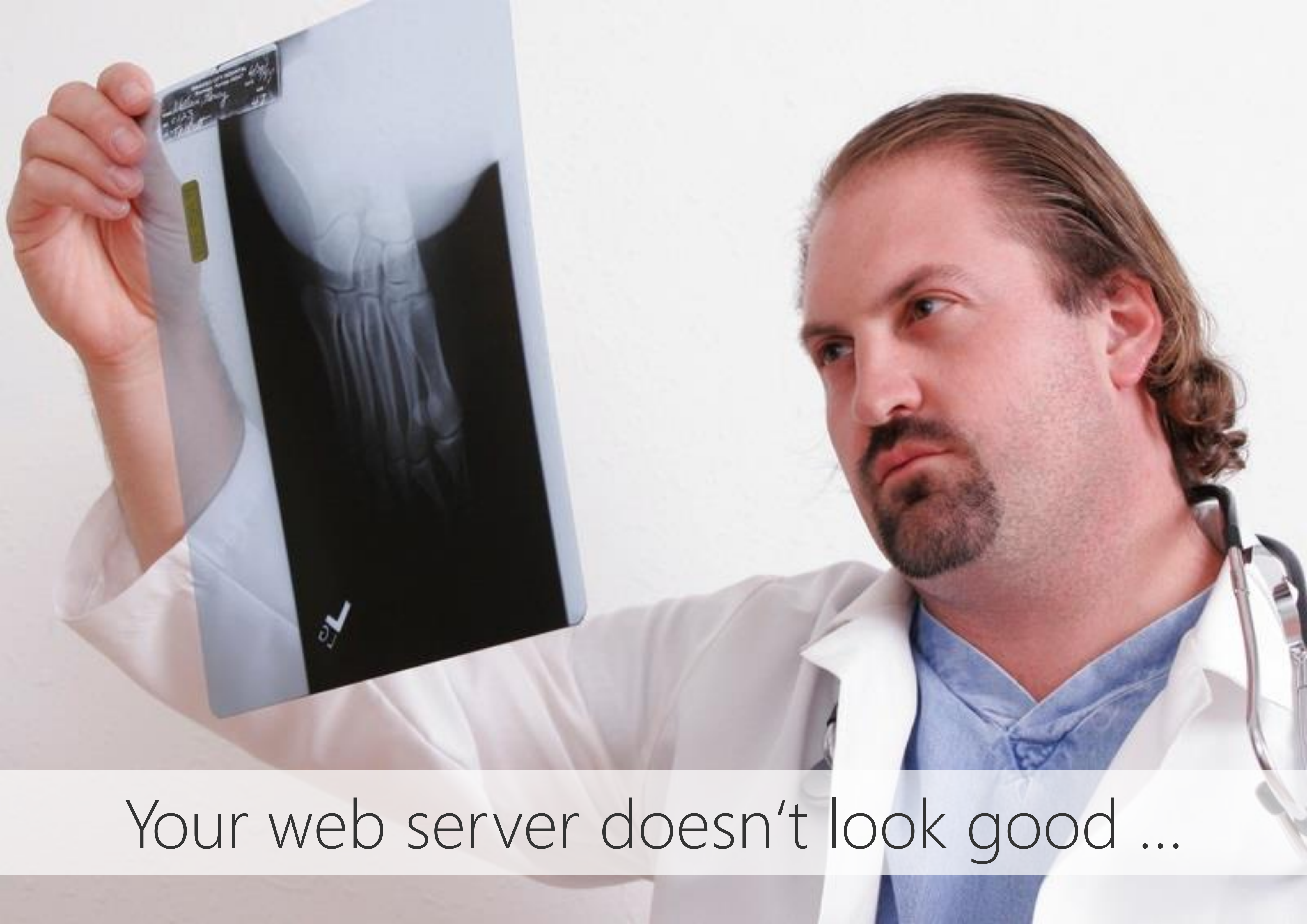
Muster für Fehlertoleranz einfach umgesetzt

Uwe Friedrichsen

codecentric AG

Uwe Friedrichsen

@ufried


Your web server doesn‘t look good …

The dreaded SiteTooSuccessfulException ...

I can hardly hear you ...

It’s all about production!

Production

⬇

Availability

⬇

Resilience

⬇

Fault Tolerance

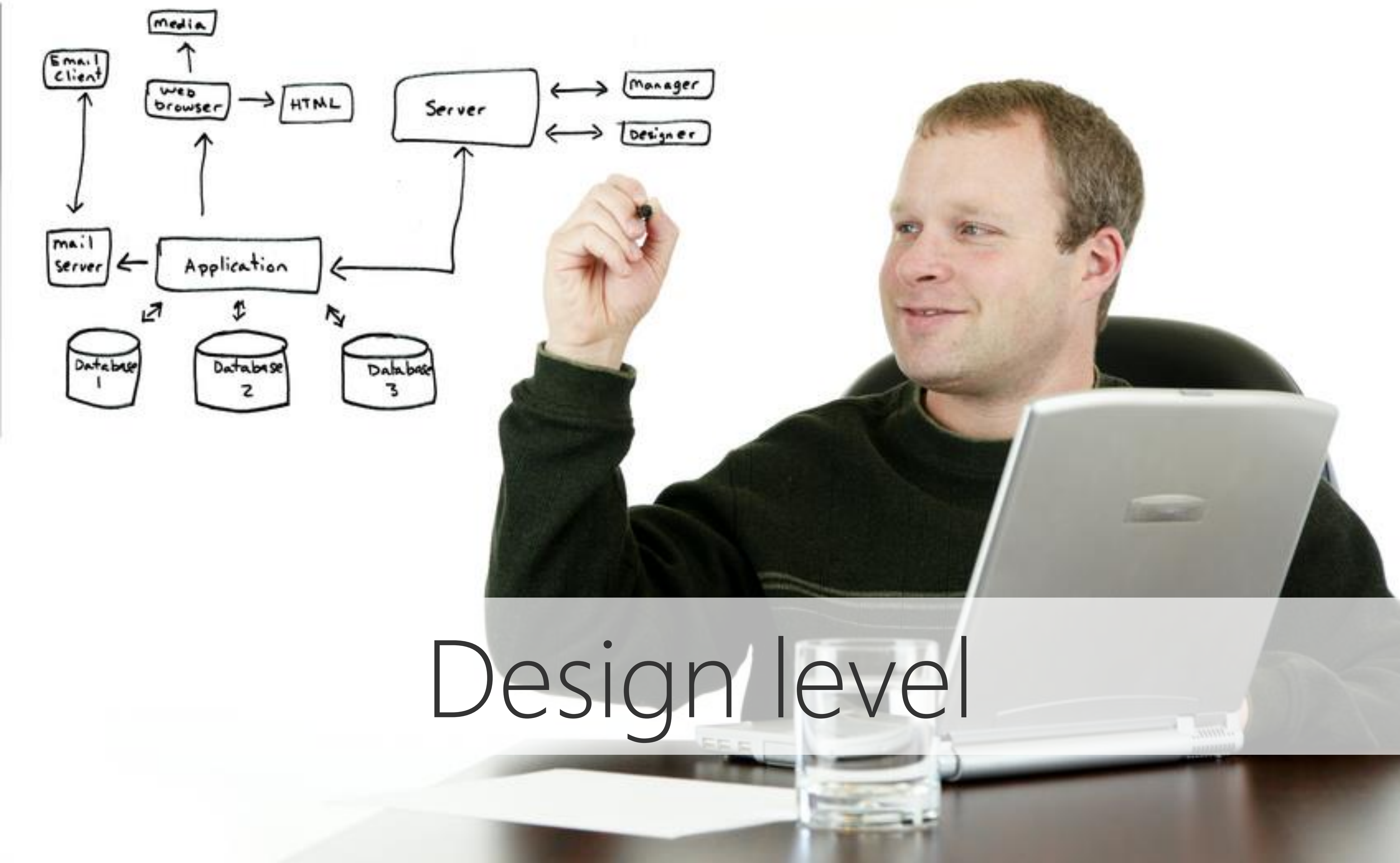

It's also about scale out!

it's

huge

let's

focus



Design level

Fault

Error

Failure

Crash failure

Omission failure

Timing failure

Response failure

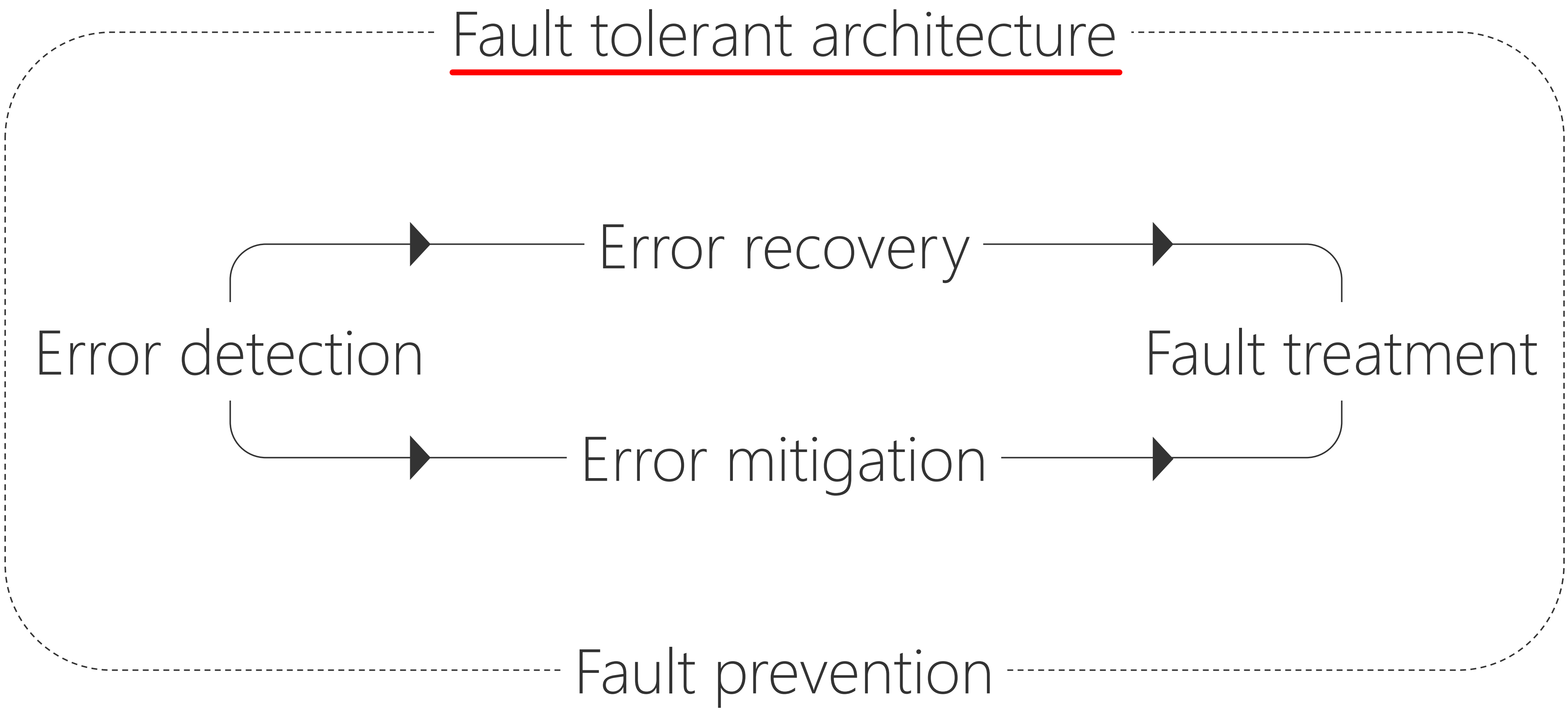
Byzantine failure

MTTF

MTTR

MTBF

Pattern taxonomy



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

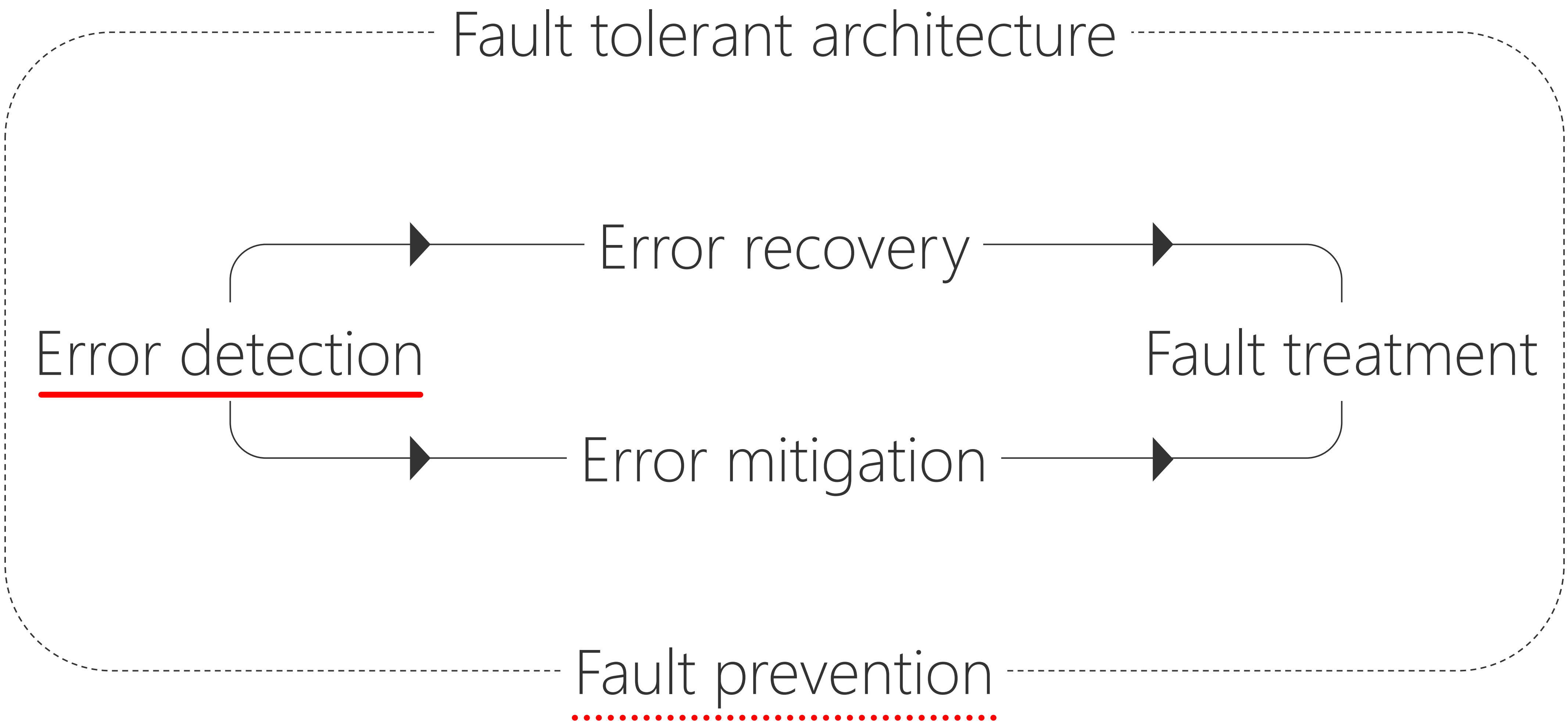
Let it crash, limit retries, rollback, failover, reset, …

Tradeoffs

Implementing a good escalation strategy is complex

Decision when to escalate is often hard

Pattern taxonomy



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

Availability 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

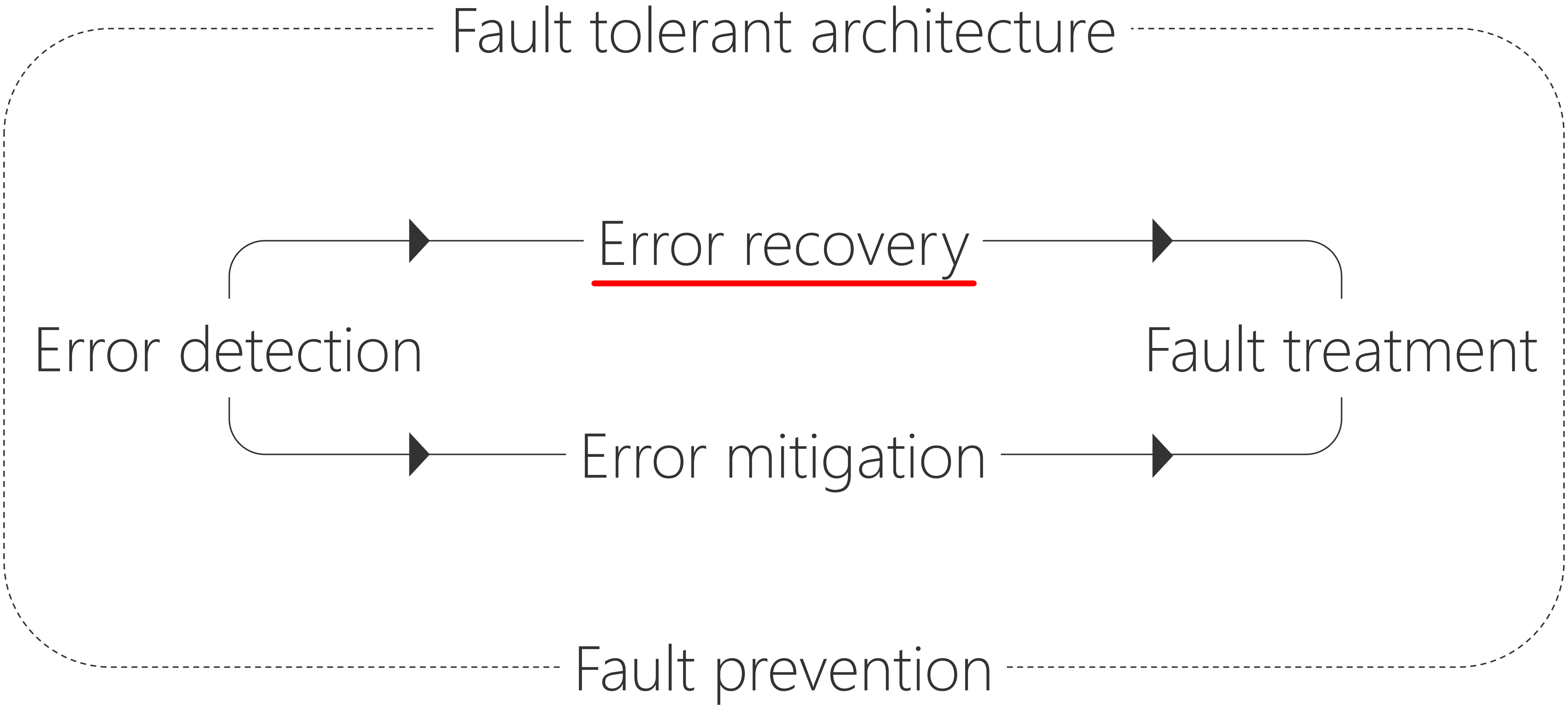
Automation, routine audits, routine exercise, …

Tradeoffs

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

Cost/benefit analysis is usually needed

Pattern taxonomy



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

Recovery strategy

Domain

Error recovery

When to use

An error has occurred and the system needs to recover

Select strategy depending on the severity of the error and data

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

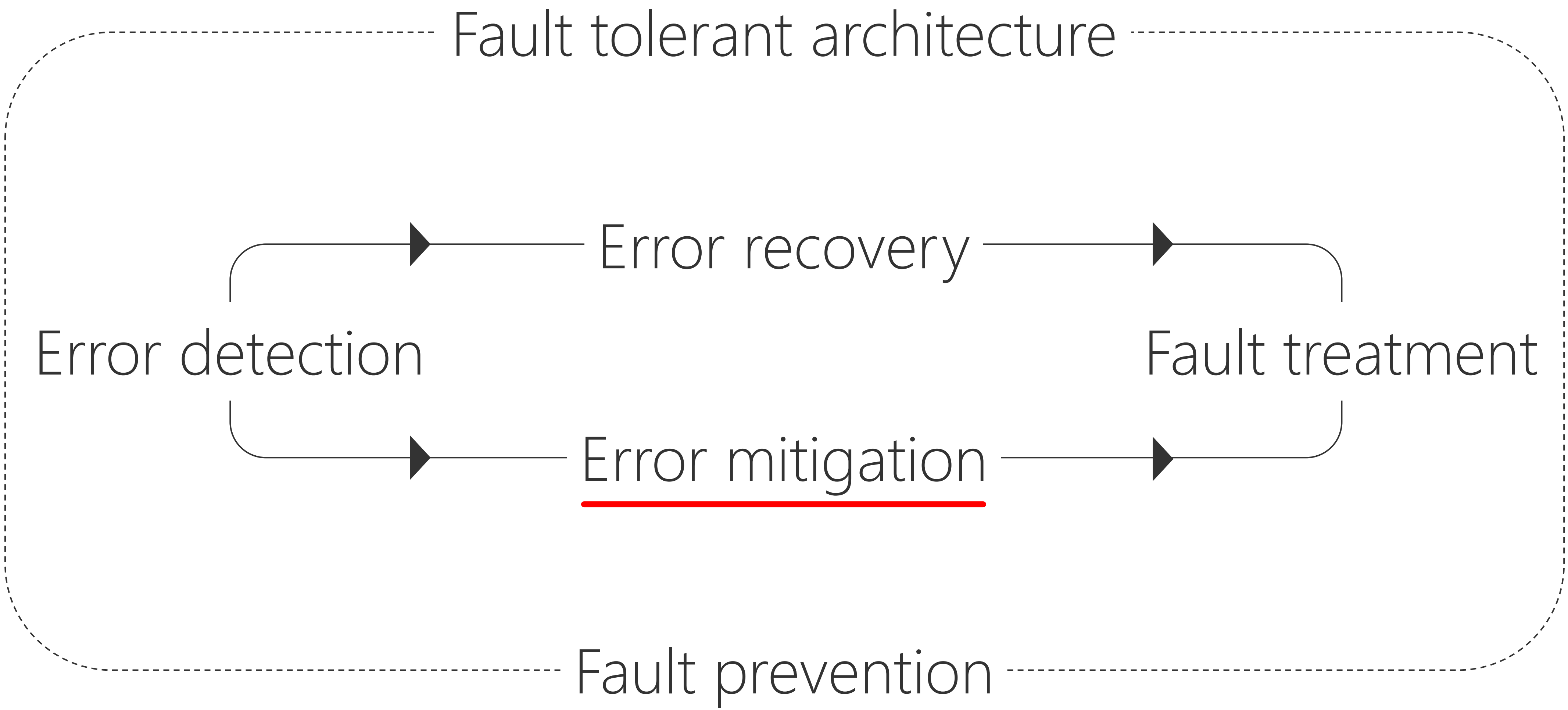
Related Concepts

Redundancy, escalation, restart, …

Tradeoffs

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

Pattern taxonomy



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

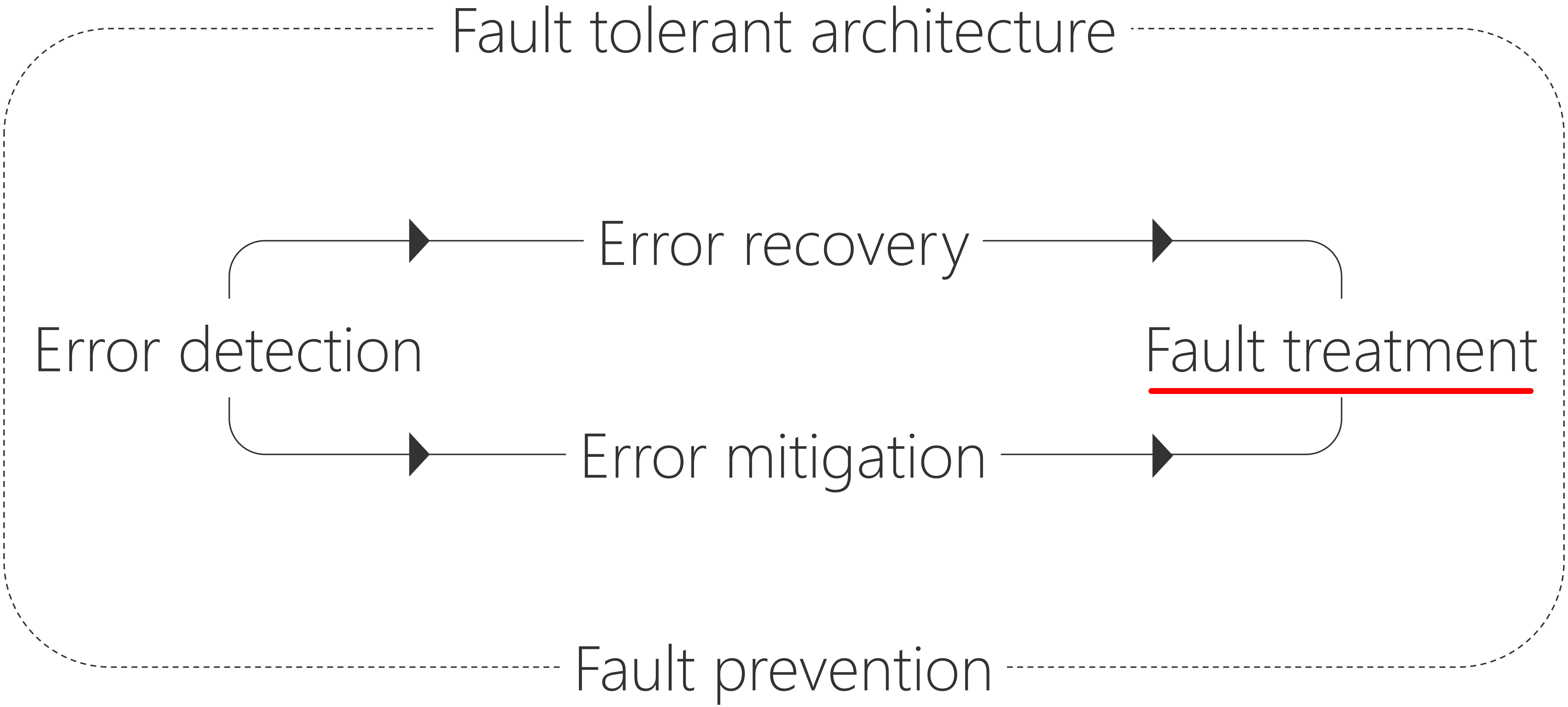
Routine audits, error correcting codes, …

Tradeoffs

Ignoring marked data is a lot of manual implementation effort

Hard to implement à posteriori into an existing system

Pattern taxonomy



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

And a lot more stuff …

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


Implementation level

Pattern #1

Timeouts

Timeouts (1)

```
// Basics
myObject.wait();  // Do not use this by default
myObject.wait(TIMEOUT);  // Better use this

// Some more basics
myThread.join();  // Do not use this by default
myThread.join(TIMEOUT);  // Better use this
```

Timeouts (2)

```
// Using the Java concurrent library
Callable<MyActionResult> myAction = <My Blocking Action>

ExecutorService executor = Executors.newSingleThreadExecutor();
Future<MyActionResult> future = executor.submit(myAction);
MyActionResult result = null;

try {
    result = future.get();  // Do not use this by default
    result = future.get(TIMEOUT, TIMEUNIT);  // Better use this
} catch (TimeoutException e) {  // Only thrown if timeouts are used
    ...
} catch (...) {
    ...
}
```

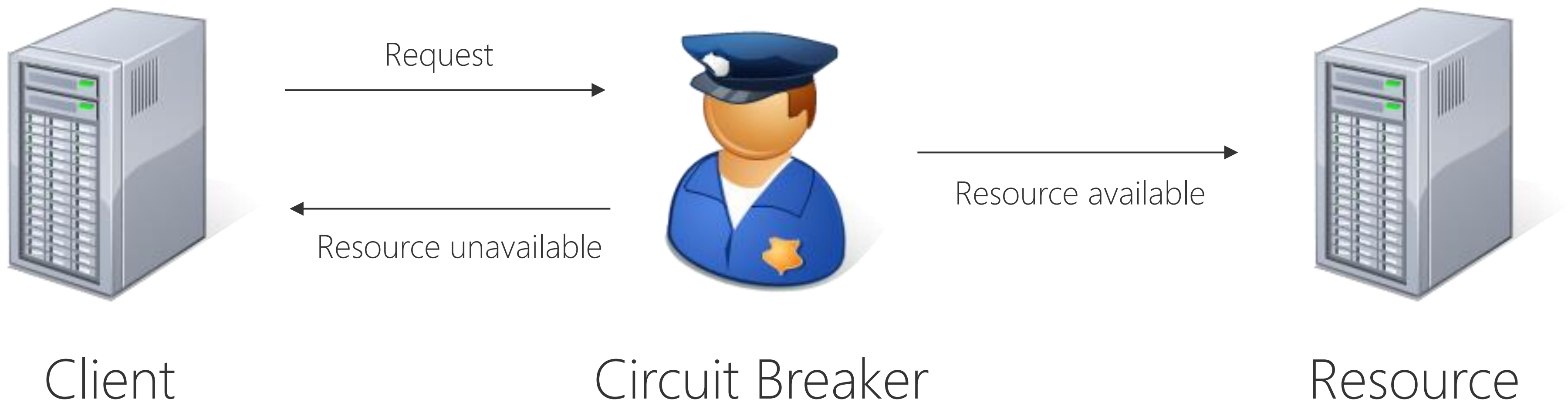
Timeouts (3)

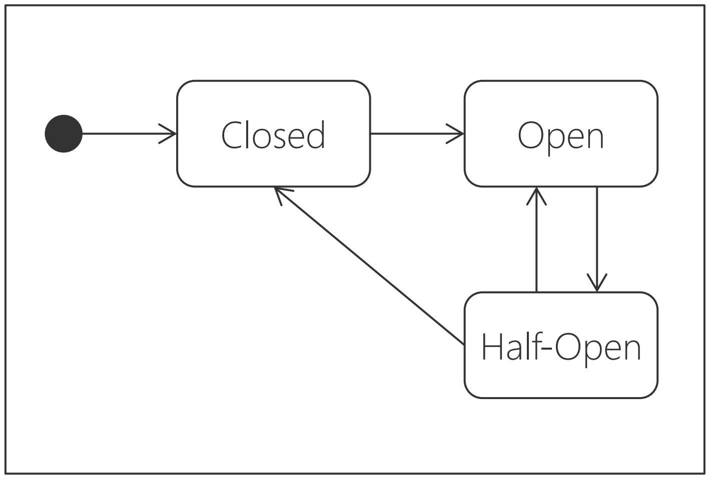
```
// Using Guava SimpleTimeLimiter
Callable<MyActionResult> myAction = <My Blocking Action>

SimpleTimeLimiter limiter = new SimpleTimeLimiter();
MyActionResult result = null;

try {
    result =
        limiter.callWithTimeout(myAction, TIMEOUT, TIMEUNIT, false);
} catch (UncheckedTimeoutException e) {
    ...
} catch (...) {
    ...
}
```

Pattern #2

Circuit Breaker

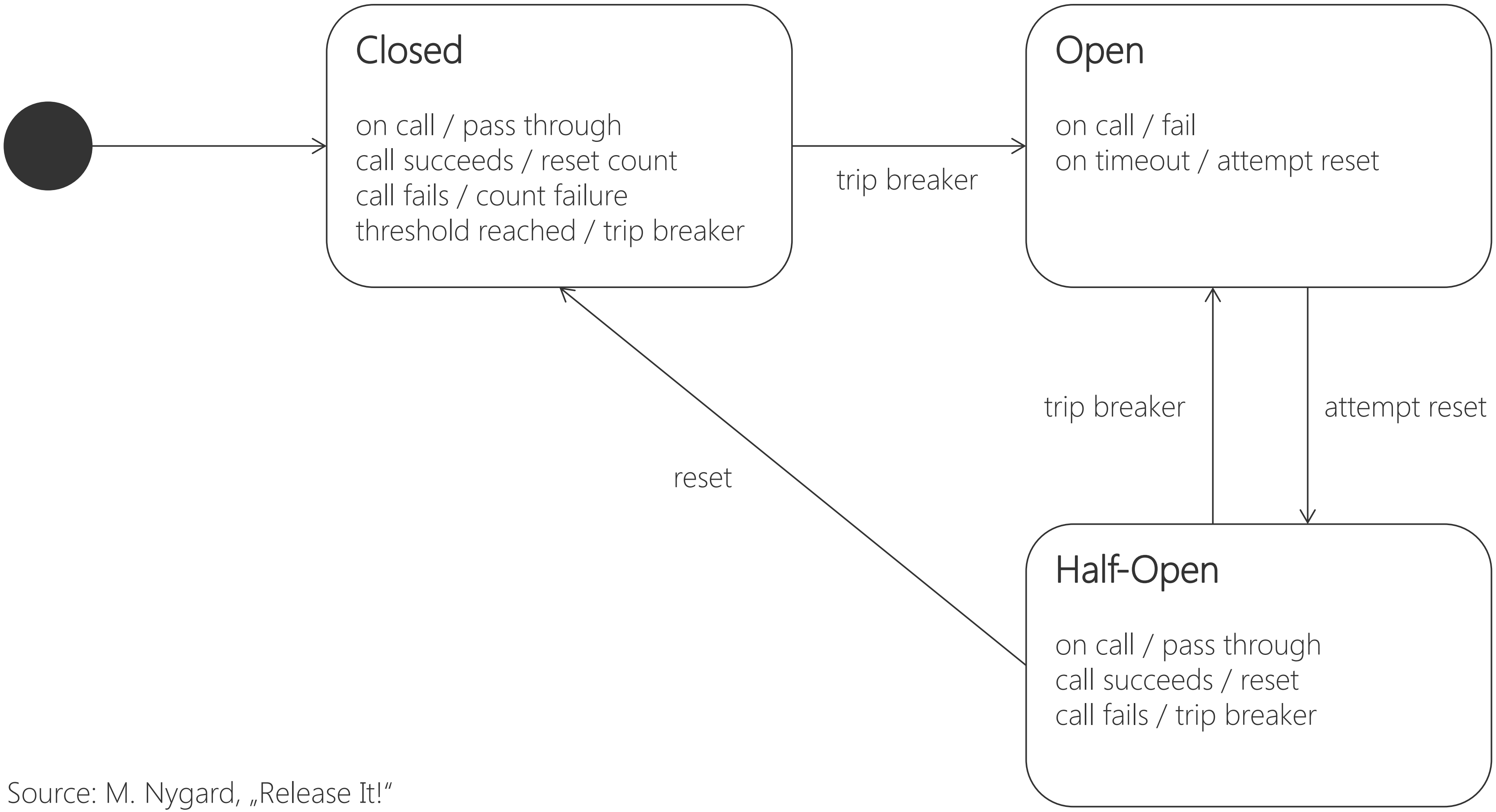
Circuit Breaker (1)





Lifecycle

Circuit Breaker (2)



Source: M. Nygard, „Release It!“

Circuit Breaker (3)

```
public class CircuitBreaker implements MyResource {
    public enum State { CLOSED, OPEN, HALF_OPEN }

    final MyResource resource;
    State state;
    int counter;
    long tripTime;

    public CircuitBreaker(MyResource r) {
        resource = r;
        state = CLOSED;
        counter = 0;
        tripTime = 0L;
    }

    ...
```

Circuit Breaker (4)

```
...
public Result access(...) {  // resource access
    Result r = null;

    if (state == OPEN) {
        checkTimeout();
        throw new ResourceUnavailableException();
    }

    try {
        r = r.access(...);  // should use timeout
    } catch (Exception e) {
        fail();
        throw e;
    }
    success();
    return r;
}
...
```

Circuit Breaker (5)

```
...

private void success() {
    reset();
}

private void fail() {
    counter++;
    if (counter > THRESHOLD) {
        tripBreaker();
    }
}

private void reset() {
    state = CLOSED;
    counter = 0;
}

...
```

Circuit Breaker (6)

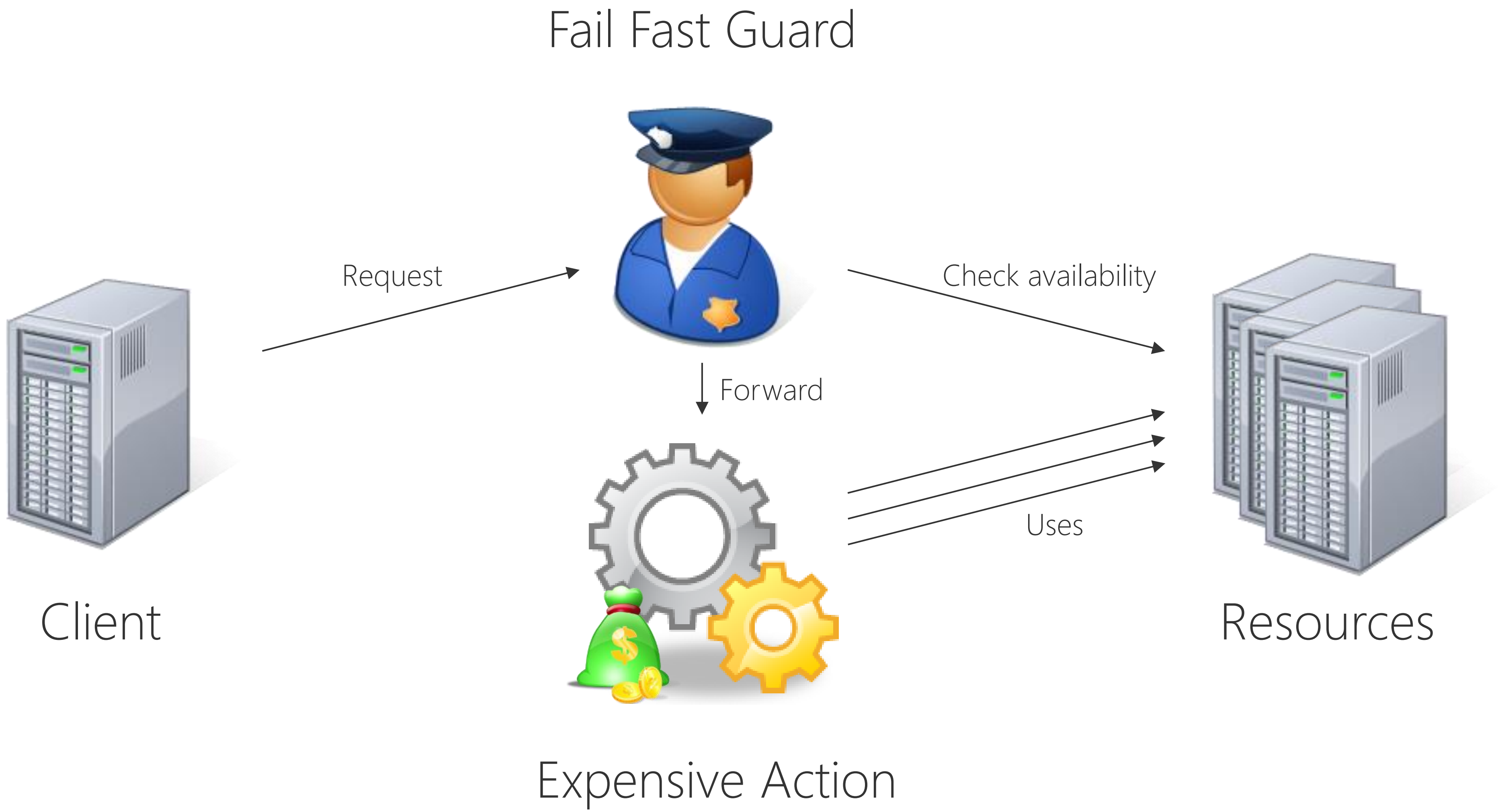
```
    ...

    private void tripBreaker() {
        state = OPEN;
        tripTime = System.currentTimeMillis();
    }

    private void checkTimeout() {
        if ((System.currentTimeMillis - tripTime) > TIMEOUT) {
            state = HALF_OPEN;
            counter = THRESHOLD;
        }
    }

    public State getState()
        return state;
    }
}
```

Pattern #3

Fail Fast

Fail Fast (1)

Fail Fast (2)


Fail Fast Guard
Request
Check availability
Forward
Uses
Client
Resources
Expensive Action

Fail Fast (3)

```
public class FailFastGuard {
    private FailFastGuard() {}

    public static void checkResources(Set<CircuitBreaker> resources) {
        for (CircuitBreaker r : resources) {
            if (r.getState() != CircuitBreaker.CLOSED) {
                throw new ResourceUnavailableException(r);
            }
        }
    }
}
```

Fail Fast (4)

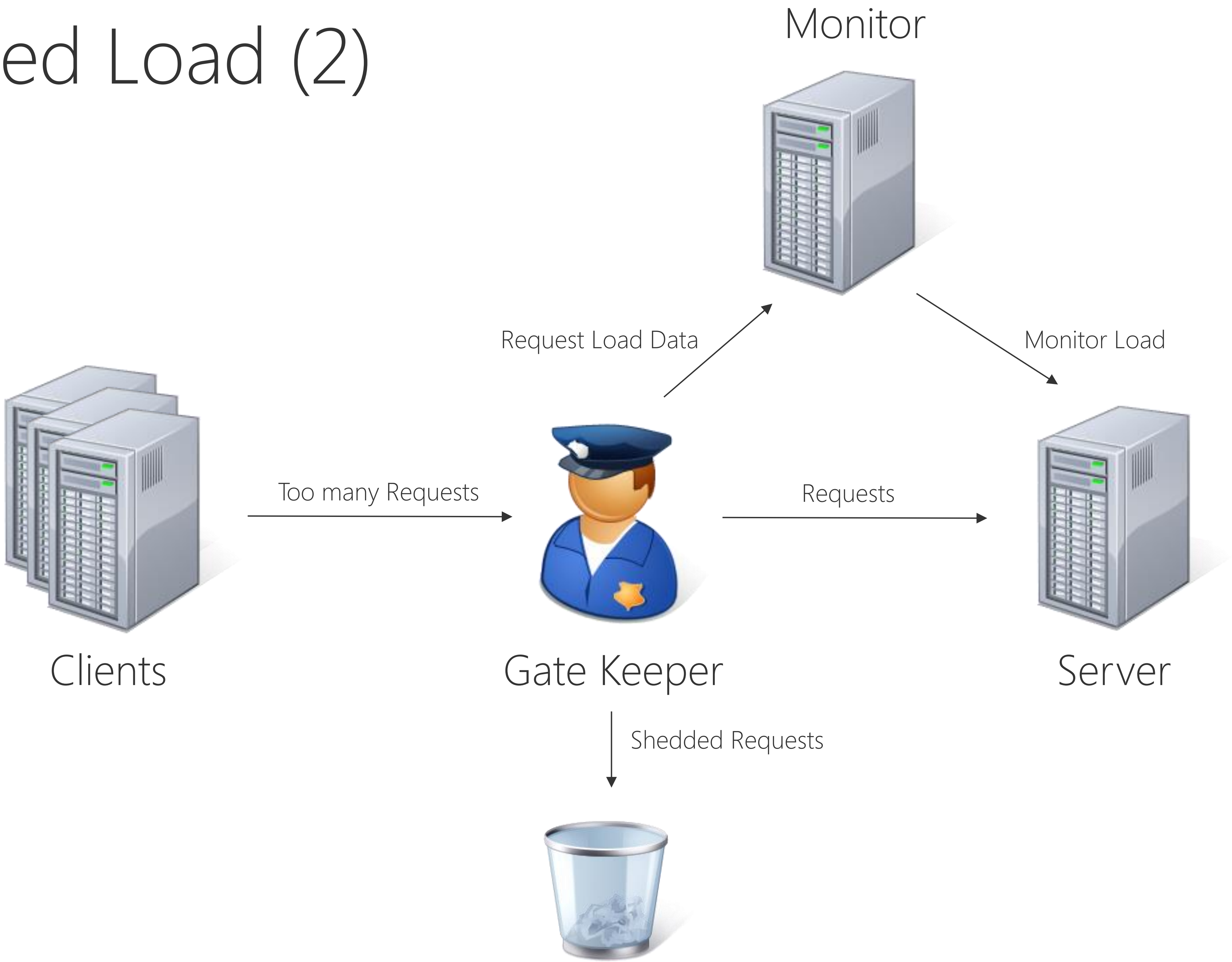
```
public class MyService {
    Set<CircuitBreaker> requiredResources;

    // Initialize resources
    ...

    public Result myExpensiveAction(...) {
        FailFastGuard.checkResources(requiredResources);

        // Execute core action
        ...
    }
}
```

Pattern #4

Shed Load

Shed Load (1)

Shed Load (2)


Monitor
Request Load Data
Monitor Load
Too many Requests
Requests
Clients
Gate Keeper
Server
Shedded Requests

Shed Load (3)

```
public class ShedLoadFilter implements Filter {
    Random random;

    public void init(FilterConfig fc) throws ServletException {
        random = new Random(System.currentTimeMillis());
    }

    public void destroy() {
        random = null;
    }

    ...
```

Shed Load (4)

```
...

public void doFilter(ServletRequest request,
                     ServletResponse response,
                     FilterChain chain)
                     throws java.io.IOException, ServletException {
    int load = getLoad();
    if (shouldShed(load)) {
        HttpServletResponse res = (HttpServletResponse)response;
        res.setIntHeader("Retry-After", RECOMMENDATION);
        res.sendError(HttpServletResponse.SC_SERVICE_UNAVAILABLE);
        return;
    }
    chain.doFilter(request, response);
}

...
```

Shed Load (5)

```
    ...

    private boolean shouldShed(int load) {  // Example implementation
        if (load < THRESHOLD) {
            return false;
        }
        double shedBoundary =
            ((double)(load - THRESHOLD))/
            ((double)(MAX_LOAD - THRESHOLD));
        return random.nextDouble() < shedBoundary;
    }
}
```

Shed Load (6)


www.catonmat.net/http-proxy-in-nodejs/
Programming
55 Comments
April 28, 2010
A HTTP Proxy Server in 20 Lines of node.js Code
Microsoft Cloud Computing
Virtualisieren Sie jetzt: Mit der Microsoft Private Cloud! Infos hier
microsoft.com/virtualisierung
AdChoices
var http = require('http');
http.createServer(function(request, response) {
var proxy = http.createClient(80, request.headers['host'])
var proxy_request = proxy.request(request.method, request.url, request.h
proxy_request.addListener('response', function (proxy_response) {
proxy_response.addListener('data', function(chunk) {
response.write(chunk, 'binary');
});
proxy_response.addListener('end', function() {
response.end();
});
response.writeHead(proxy_response.statusCode, proxy_response.headers);
});
request.addListener('data', function(chunk) {
proxy_request.write(chunk, 'binary');
});
request.addListener('end', function() {
proxy_request.end();
});
}).listen(8080);
This is just amazing. In 20 lines of node.js code and 10 minutes of time I was able to write a HTTP proxy. And it scales well, too. It's not a blocking HTTP proxy, it's event driven and asynchronous, meaning hundreds of people can use simultaneously and it will work well.
Peteris Krumins' blog about programming, hacking, software reuse, software ideas, computer security, google and technology.
Reach me at:
peter@catonmat.net
Or meet me on:
Twitter Facebook Plurk ↓ more
Subscribe to my posts:
Subscribe through an RSS feed:
15383 readers
BY FEEDBURNER
(what is rss?)
Subscribe through email:
Enter your email address:
Subscribe
Delivered by FeedBurner
Server Sponsors
I am being sponsored by Syntress since 2007! They bought me an amazing dedicated server to run catonmat on. If you're looking web services in Chicago

Shed Load (7)

nginx.org/en/docs/http/ngx_http_limit_conn_module.html

Module ngx_http_limit_conn_module

The `ngx_http_limit_conn_module` module allows to limit the number of connections per defined key, in particular, the number of connections from a single IP address.

Not all connections are counted; only those that have requests currently being processed by the server, in which request header has been fully read.

Example Configuration

```
http {
    limit_conn_zone $binary_remote_addr zone=addr:10m;

    ...

    server {

        ...

        location /download/ {
            limit_conn addr 1;
        }
```

Directives

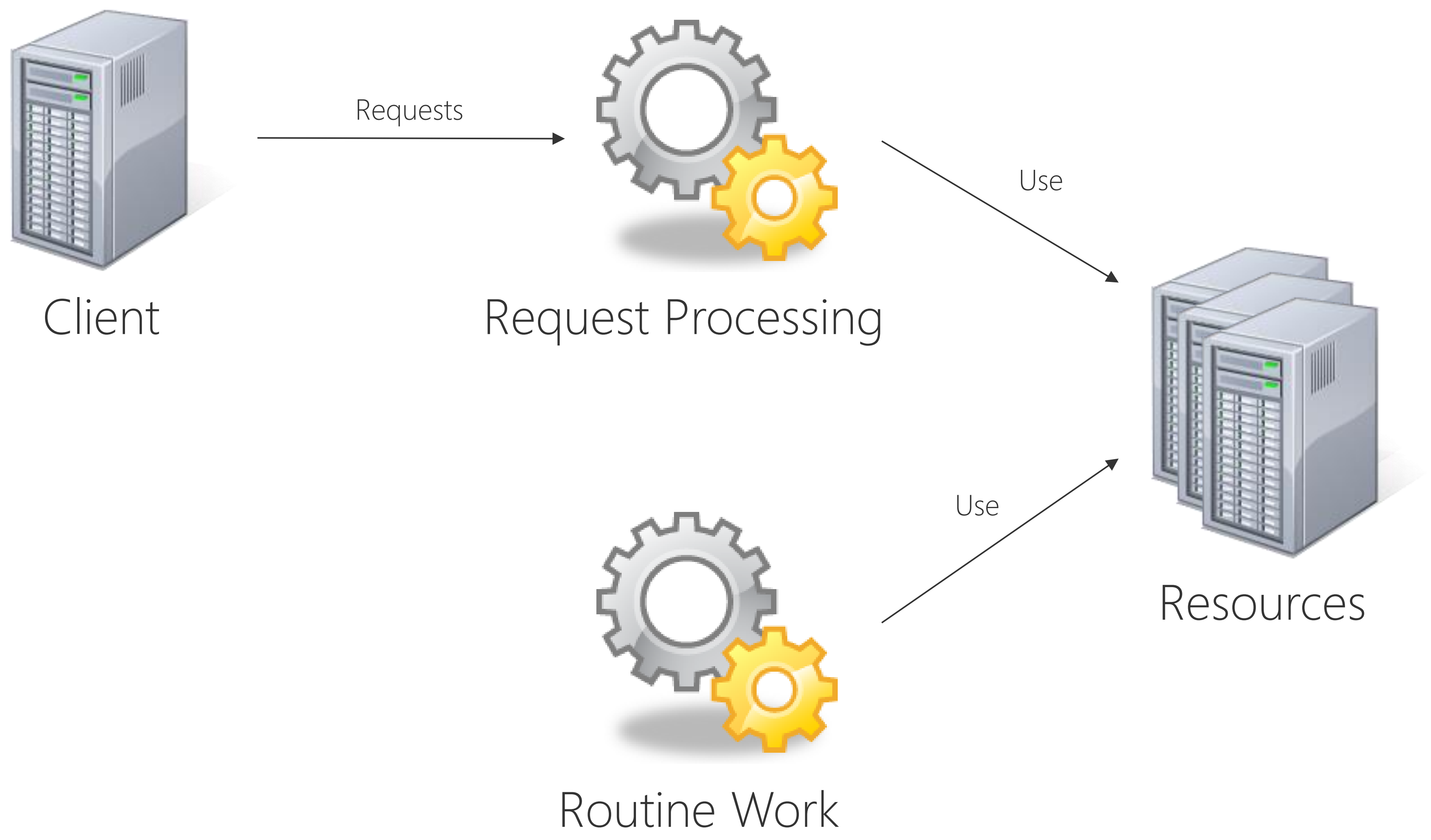
syntax: `limit_conn zone number;`
default: —
context: `http`, `server`, `location`

Sets a shared memory zone and the maximum allowed number of connections for a given key value. When this limit is exceeded, the server will return error 503 (Service Temporarily Unavailable) in reply to a request. For example, the directives

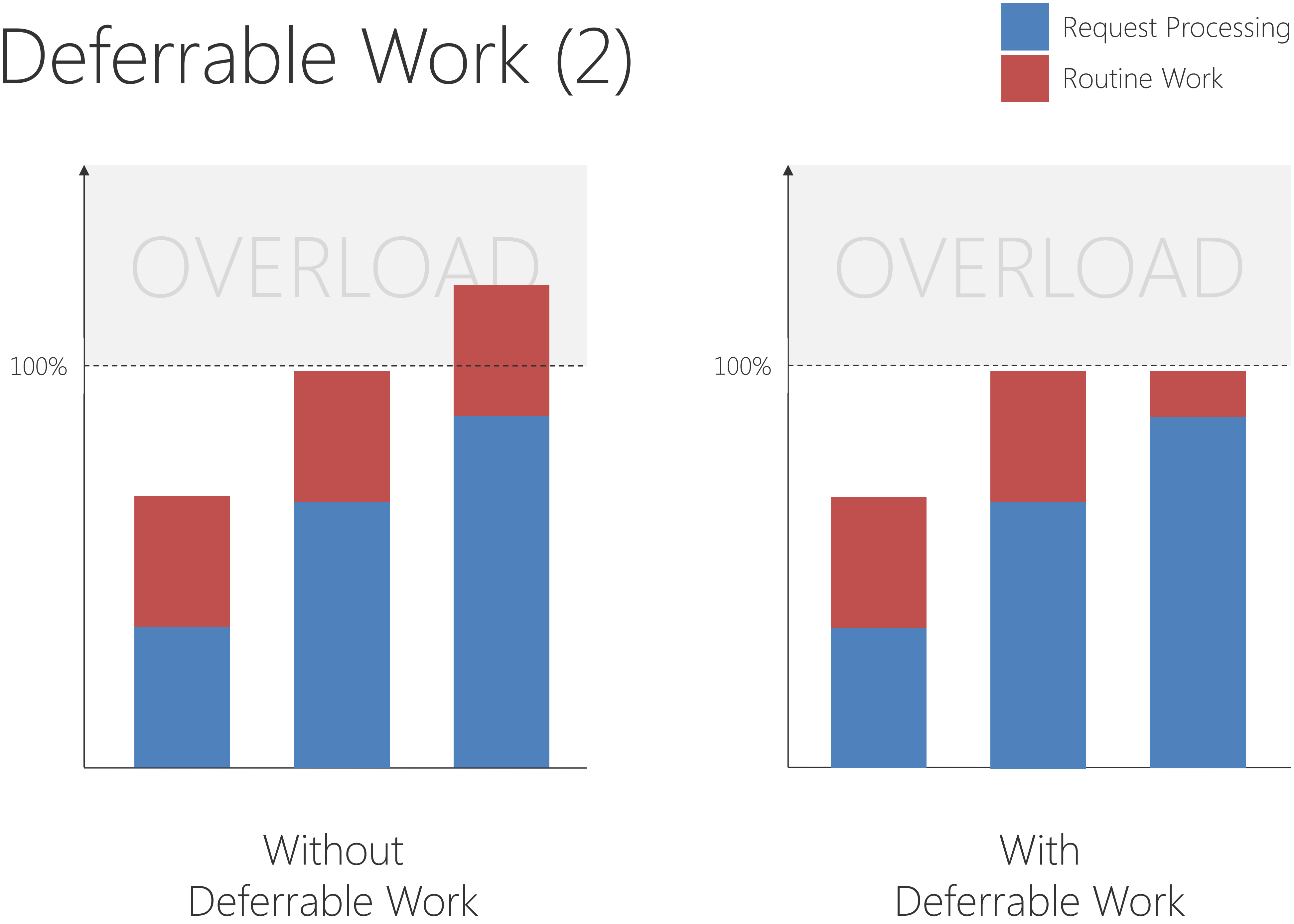
Pattern #5

Deferrable Work

Deferrable Work (1)



Deferrable Work (2)



Deferrable Work (3)

```
// Do or wait variant
ProcessingState state = initBatch();
while(!state.done()) {
    int load = getLoad();
    if (load > THRESHOLD) {
        waitFixedDuration();
    } else {
        state = processNext(state);
    }
}

void waitFixedDuration() {
    Thread.sleep(DELAY);  // try-catch left out for better readability
}
```

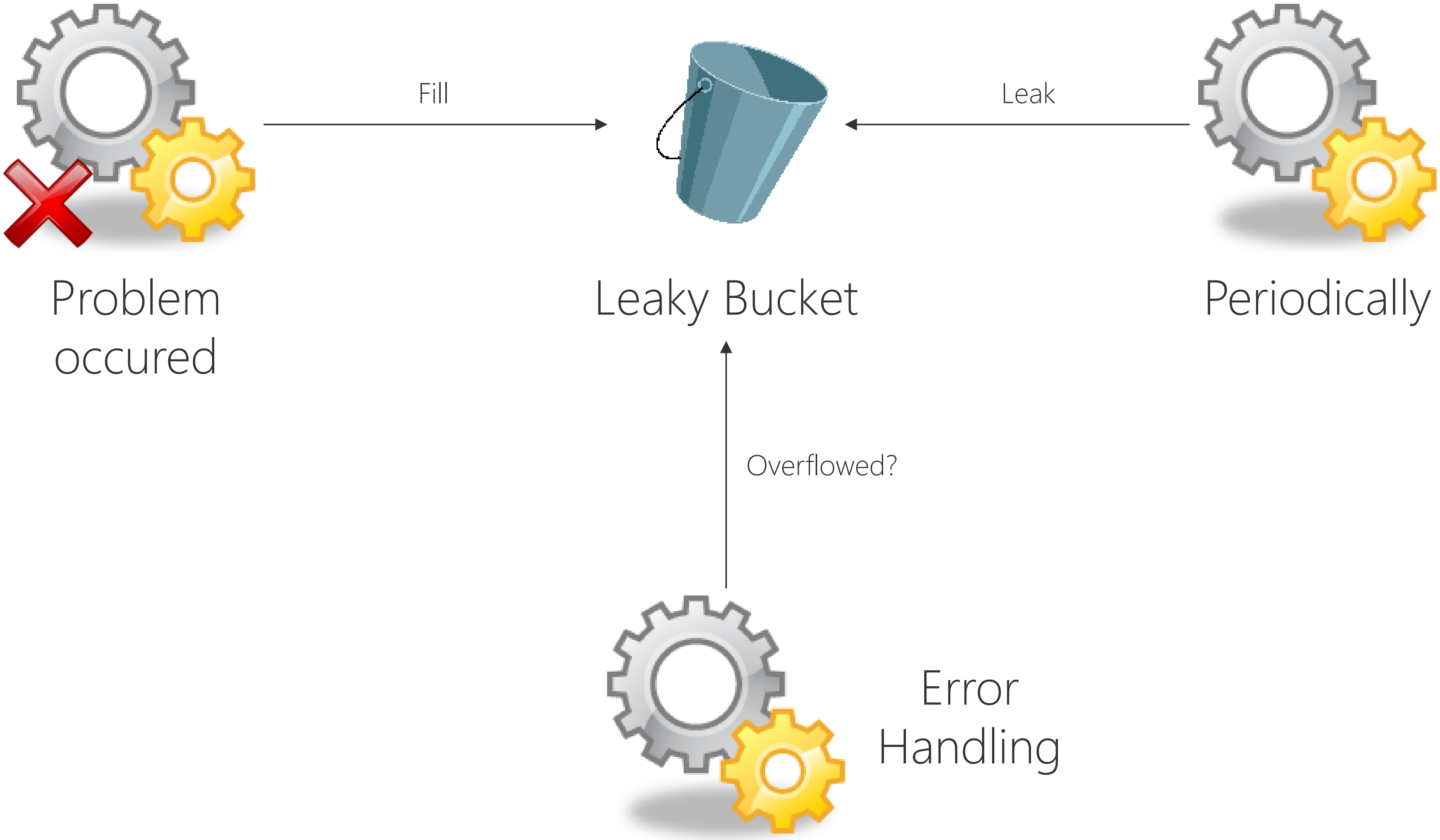
Deferrable Work (4)

```
// Adaptive load variant
ProcessingState state = initBatch();
while(!state.done()) {
    waitLoadBased();
    state = processNext(state);
}

void waitLoadBased() {
    int load = getLoad();
    long delay = calcDelay(load);
    Thread.sleep(delay);  // try-catch left out for better readability
}

long calcDelay(int load) {  // Simple example implementation
    if (load < THRESHOLD) {
        return 0L;
    }
    return (load - THRESHOLD) * DELAY_FACTOR;
}
```

Pattern #6

Leaky Bucket

Leaky Bucket (1)



Leaky Bucket (2)

```
public class LeakyBucket { // Very simple implementation
    final private int capacity;
    private int level;
    private boolean overflow;

    public LeakyBucket(int capacity) {
        this.capacity = capacity;
        drain();
    }

    public void drain () {
        this.level = 0;
        this.overflow = false;
    }

    ...
```

Leaky Bucket (3)

```
    ...

    public void fill() {
        level++;
        if (level > capacity) {
            overflow = true;
        }
    }

    public void leak() {
        level--;
        if (level < 0) {
            level = 0;
        }
    }

    public boolean overflowed() {
        return overflow;
    }
}
```

Pattern #7

Limited Retries

Limited Retries (1)

```
// doAction returns true if successful, false otherwise

// General pattern
boolean success = false
int tries = 0;
while (!success && (tries < MAX_TRIES)) {
    success = doAction(...);
    tries++;
}

// Alternative one-retry-only variant
success = doAction(...) || doAction(...);
```

More Patterns

- Complete Parameter Checking
- Marked Data
- Routine Audits

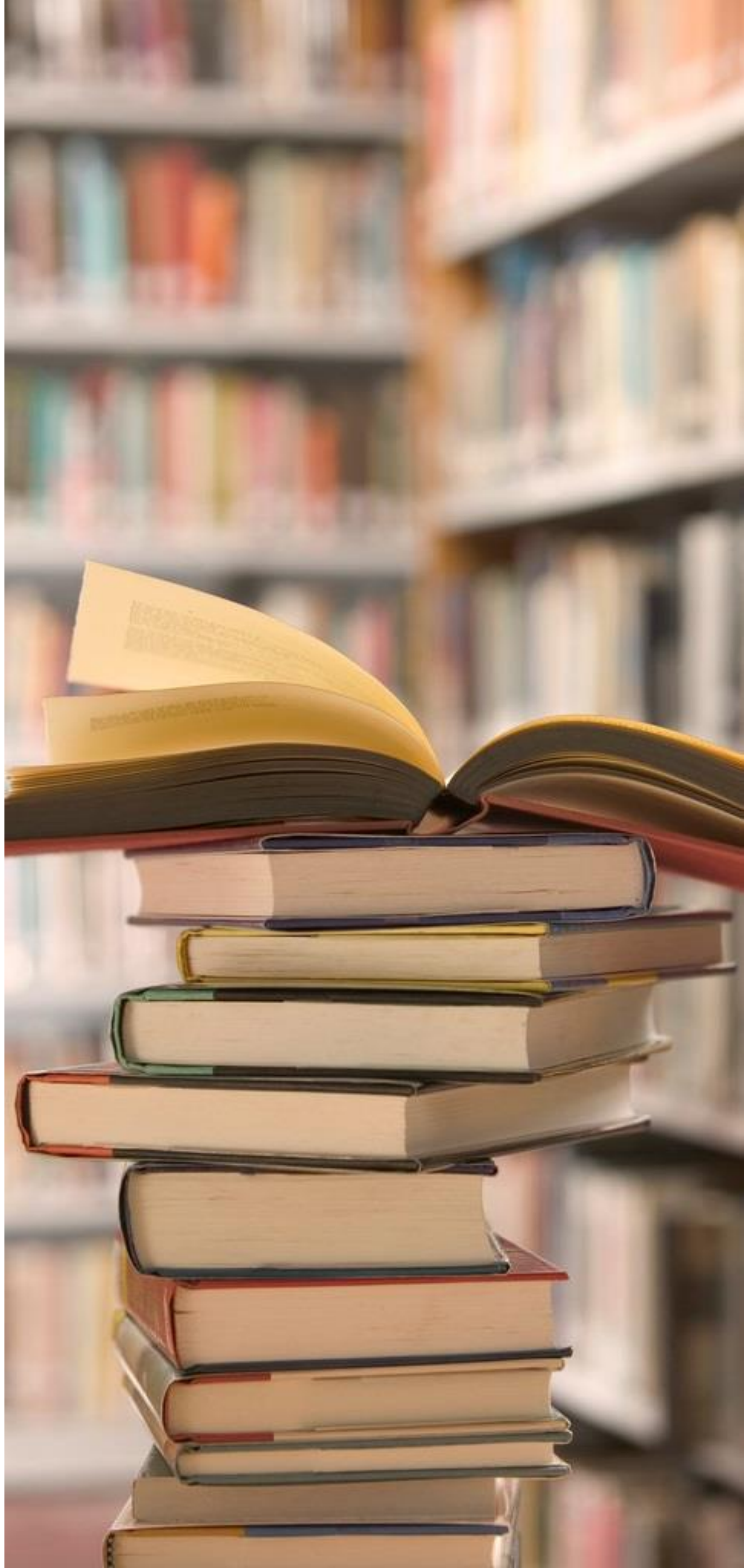
Further reading

1. Michael T. Nygard, Release It!, Pragmatic Bookshelf, 2007

2. Robert S. Hanmer, Patterns for Fault Tolerant Software, Wiley, 2007

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

4. Andrew Tanenbaum, Marten van Steen, Distributed Systems – Principles and Paradigms, Prentice Hall, 2nd Edition, 2006



It’s all about production!

Uwe Friedrichsen

@ufried

uwe.friedrichsen@codecentric.de

http://www.slideshare.net/ufried/
http://blog.codecentric.de/author/ufr