Resilient software design in a nutshell
A compact introduction for the curious software engineer

Uwe Friedrichsen (codecentric AG) – Software Architecture Conference – London, 18. October 2017
@ufried
Agenda

• The why and what of resilience
• Design for isolation
• Communication paradigms
• Basic resilience patterns
• The full 9 yards
• The resilience design cycle
• Introducing resilience @ work

Breaks based on wall clock
What’s that “resilience” thing?
resilience (ri'zil yəns) also resilience, n.

1. the power or ability to return to the original form, position, etc., after being bent, compressed, or stretched; elasticity.
2. ability to recover readily from illness, depression, adversity, or the like; buoyancy.

Again, what’s that “resilience” thing?
What’s it all about?
It’s all about production!
Business → Production → Availability
Availability := \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}

MTTF: Mean Time To Failure
MTTR: Mean Time To Recovery
A word (or two) about failures ...
Fault  –  Error  –  Failure
Failure types

- Crash failure
- Omission failure
- Timing failure
- Response failure
- Byzantine failure
How can I maximize availability?
Traditional stability approach

Availability := \[ \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \]

Maximize MTTF
Stability approach

- Minimize error likelihood
- Special high availability hardware
- Server clusters
- Multiple network connections
- ...

→ Solved upfront using infrastructure means
What’s the problem?
(Almost) every system is a distributed system

-- Chas Emerick

http://www.infoq.com/presentations/problems-distributed-systems
The Eight Fallacies of Distributed Computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

-- Peter Deutsch

“A 2011 study of several Microsoft datacenters observed over 13,300 network failures with end-user impact, with an estimated median 59,000 packets lost per failure. The study found a mean of 40.8 network link failures per day (95th percentile: 136), with a median time to repair of around five minutes (and up to one week).”

Source: Peter Bailis et al., "Highly Available Transactions: Virtues and Limitations", VLDB 2014
world-networks/ guarantees serializability, but, as we see below, their benefits are of-
scribe varying restrictions on the space of schedules that are allow-
than serializability: the host of so-called 
mance degradation, scalability limitations, and, often, aborts due 
in the form of decreased concurrency (and, subsequently, perfor-
ies associated with serializability can be severe and are manifested 
however, even in a single-node database, the coordination penal-
designs face a choice between availability and strong semantics. 
meaning that, in environments like those in Section 2, database 
that serializability is not achievable in a highly available system [28], 
databases? Database researchers and designers have long realized 
address partitions and latency: what does this mean for distributed 
ning communication delays is also an active area of research in the 
station of latencies varies between links, but the trend is clear: remote 
singapore RTT is 106.7ms, ping packets incur an average 362.8ms 
light: for example, while a speed-of-light RTT from S 
ble 1c). The cost of wide-area communication exceeds the speed of 
study in Table 1. On average, intra-datacenter communication (Ta-
and within a single "availability zone" (datacenter), at a granular-
trip times, or RTTs) between all seven EC2 geographic "regions," 
illustrate the difference between intra-datacenter, inter-datacenter, 
(c) Cross-region (CA: California, OR: Oregon, VA: Virginia, TO: Tokyo, 
IR: Ireland, SY: Sydney, SP: São Paulo, SI: Singapore) 
Table 1: Mean RTT times on EC2 (min and max highlighted) 

Source: Peter Bailis et al., "Highly Available Transactions: Virtues and Limitations", VLDB 2014
... and that’s only the network.

We haven’t talked about failing nodes, faulty processes, etc. yet, that also materialize as failures in distributed systems.
A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

-- Leslie Lamport
Failures in today's complex, distributed and interconnected systems are not the exception.

- They are the normal case
- They are not predictable
... and it’s getting “worse”

- Cloud-based systems
- Microservices
- Zero Downtime
- Mobile & IoT
- Social Web

→ *Ever-increasing complexity and connectivity*
Everything fails, all the time.

-- Werner Vogels
Do not try to avoid failures. Embrace them.
Resilience approach

Availability := \frac{MTTF}{MTTF + MTTR}

Minimize MTTR
resilience (IT)

the ability of a system to handle unexpected situations

- without the user noticing it (best case)
- with a graceful degradation of service (worst case)
Graceful degradation behavior is a business decision

Make sure to press for the decision
Beware of the “100% available” trap!
What is the business case of resilience?
Measure your lost revenue due to technical failures and do not forget the delayed revenue losses
Designing for resilience
The basics
A little taxonomy on resilience ...
Isolation

• System must not fail as a whole
• Split system in parts and isolate parts against each other
• Avoid cascading failures
• Foundation of resilient software design
Core

Detect → Prevent → Mitigate → Recover → Treat → Complement

Isolation

Bulkhead
Bulkheads

- Core isolation pattern (a.k.a. “failure units” or “units of mitigation”)
- Diverse implementation choices available, e.g., µservice, actor, scs, ...
- Implementation choice impacts system and resilience design a lot
- Shaping good bulkheads is extremely hard (pure design issue)
Sounds easy. Where is the problem?
Due to functional design, Service A always needs backing from Service B to be able to answer a client request, i.e. the isolation is broken by design.

How do we avoid this ...
Due to functional design we need to call a lot of services to be able to answer a client request, i.e. availability is broken by design.

... and this ...
By trying to avoid the aforementioned issues we ended up with cramming all required functionality in one big service

_ i.e. the isolation is broken by design

... without ending up with this?
Let’s use the well-known best practices

• Divide & conquer a.k.a. functional decomposition
• DRY (Don’t Repeat Yourself)
• Design for reusability
• Layered architecture
• ...

Un fortunately, ...
Due to functional design, Service A always needs backing from Service B to be able to answer a client request, i.e. the isolation is broken by design.

... this usually leads to this ...
Due to functional design we need to call a lot of services to be able to answer a client request, *i.e. availability is broken by design*

... and this ...
By trying to avoid the aforementioned issues we ended up with cramming all required functionality in one big service.

\textit{i.e. the isolation is broken by design}

... and in the end also often to this.
Welcome to distributed hell!
Caches to the rescue!
Due to functional design, Service A always needs backing from Service B to be able to answer a client request, i.e. the isolation is broken by design.

Break tight service coupling by caching data/responses of downstream service.
Caches to the rescue?
Do you really think that copying stale data all over your system is a suitable measure to fix an inherently broken design?
We have to re-learn design for distributed systems!
No silver bullet
Yet, a few guiding thoughts about bulkhead design ...
Foundations of design

• High cohesion, low coupling
• Separation of concerns
• Crucial across process boundaries
• Still poorly understood issue
• Start with
  • Understanding organizational boundaries
  • Understanding use cases and flows
  • Identifying functional domains (→ DDD)
  • Finding areas that change independently
  • Do not start with a data model!
Short activation paths

- Long activation paths affect availability
- Increase likelihood of failures
- Minimize remote calls per request
- Need to balance opposing forces
  - Avoid monolith \(\rightarrow\) clear separation of concerns
  - Minimize requests \(\rightarrow\) cluster functionality & data
  - Caches sometimes help, but stale data as trade-off
Dismiss reusability

- Reusability increases coupling
- Reusability leads to bad service design
- Reusability compromises availability
- Reusability rarely pays
- Do not strive for reuse
- Strive for replaceability instead
- Try to tackle reusability issues with libraries
Broadening the options ...
Core

Detect

Prevent

Mitigate

Recover

Treat

Complement

Isolation

Communication paradigm

Bulkhead
Communication paradigm

- Request-response <-> messaging <-> events <-> ...
- Heavily influences resilience patterns to be used
- Also heavily influences functional bulkhead design
- Very fundamental decision which is often underestimated
Request/Response: Horizontal slicing

Event-driven: Vertical slicing

Flow / Process
Synchronous R/R vs. asynchronous events

• Decomposition
  • Vertically divide-and-conquer vs. horizontally go-with-the-flow

• Coordination
  • Coordination logic/services and orchestration vs. event chains and choreography

• Transactions
  • Built-in transaction handling vs. external supervision

• Error handling
  • Built into service vs. escalation/supervision strategy

• Separation of concerns
  • Multiple responsibilities service vs. single responsibility services

• Encapsulation
  • Domain logic distributed across services vs. domain logic in one place
  • Reusability vs. Replaceability

• Complexity
  • A draw ...
The communication paradigm influences the functional service design a lot and also the resilience patterns to be used
Example: order fulfillment

- Simple order, credit card, non-digital items
- Add coupons incl. validation
- Add promotions incl. usage notification
- Add bonus card incl. purchase notification
- Customer accounts as payment type
- PayPal as payment type
- Integrate digital music library
- Integrate digital video library
- Integrate e-book library
Design exercise – Part 1

Create a bulkhead design for the case study

- Use one communication paradigm
  - Synchronous request/response (e.g., REST)
  - Asynchronous messaging (e.g., Akka)
  - Asynchronous events (e.g., Pub/Sub)

- Split up in groups
  - Each group using a different paradigm

- Take a few notes
Customer pressed “Buy now”

Online Shop Checkout

Credit Card Provider
PayPal
Accounts Receivables
Coupon Management
Campaign Management
Loyalty Management
E-Mail Server
Warehouse System
Music Library
E-Book Library
Video Library
Services are responsible to eventually succeed or fail for good, usually incorporating a supervision/escalation hierarchy for that.
Do not limit your design options upfront without an important reason
Timeouts

- Preserve responsiveness independent of downstream latency
- Crucial if using synchronous communication
- Also needed if using asynchronous request/response style
- Good library support in most programming languages
Basic timeout support (Java)

// 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 with standard library means

// Wrap blocking action in a Callable
Callable<MyActionResult> myAction = <My Blocking Action>

// Use a simple ExecutorService to run the action in its own thread
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<MyActionResult> future = executor.submit(myAction);
MyActionResult result = null;

// Use Future.get() method to limit time to wait for completion
try {
    result = future.get(TIMEOUT, TIMEUNIT);
    // Action completed in a timely manner – process results
} catch (TimeoutException e) {
    // Handle timeout (e.g., schedule retry, escalate, alternate action, …)
} catch (...) {
    // Handle other exceptions that can be thrown by Future.get()
} finally {
    // Make sure the callable is stopped even in case of a timeout
    future.cancel(true);
}
Timeouts using the Guava library

// 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) {
    // Handle timeout (e.g., schedule retry, escalate, alternate action, ...)
} catch (...)
    ...
}
Circuit breaker

- Probably most often cited resilience pattern
- Extension of the timeout pattern
- Takes downstream unit offline if calls fail multiple times
- Can be used for most failure types (not only for timing failures)
Circuit breaker – Basic flow

Client  →  Request  →  Circuit Breaker  →  Resource available  →  Resource

- Resource unavailable
- Closed
- Open
- Half-Open

Lifecycle
Circuit breaker – State model

Closed
- on call / pass through
- call succeeds / reset count
- call fails / count failure
- threshold reached / trip breaker

Half-Open
- on call / pass through
- call succeeds / reset
- call fails / trip breaker

Open
- on call / fail
- on timeout / attempt reset

Source: M. Nygard, „Release It!“
What is Hystrix?

In a distributed environment, failure of any given service is inevitable. Hystrix is a library designed to control the interactions between these distributed services providing greater latency and fault tolerance. Hystrix does this by isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve the system's overall resiliency.

Hystrix evolved out of resilience engineering work that the Netflix API team began in 2011. Over the course of 2012, Hystrix continued to evolve and mature, eventually leading to adoption
public class HelloCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "Hello"; // Not important here
    private final String name;

    // Request parameters are passed in as constructor parameters
    public HelloCommand(String name) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.name = name;
    }

    @Override
    protected String run() throws Exception {
        // Usually here would be the resource call that needs to be guarded
        return "Hello, " + name;
    }
}

// Usage of a Hystrix command - synchronous variant
@Test
public void shouldGreetWorld() {
    String result = new HelloCommand("World").execute();
    assertEquals("Hello, World", result);
}
Acknowledgement

- Simpler variant of heartbeat without dedicated monitor
- Can also be used to track completion of asynchronous flows
- Use acknowledgement carefully with asynchronous communication
- Balance timeouts and lost acknowledges to avoid false positives
Routine checks

• Periodically check availability of nodes
• Can also be used to check operability of standby components
• Health check usually used for checking technical availability
• Synthetic transaction usually used for checking functional availability
Monitor

- Never, ever run a distributed system without thorough monitoring!
- Automatically respond to detected failures ("Self-healing")
- Alarms for human action request are only sent if self-healing fails
- Try to correlate errors to avoid excessive notifications
Retry

• Very basic recovery pattern
• Recover from omission or other transient errors
• Limit retries to minimize extra load on an already loaded resource
• Limit retries to avoid recurring errors
// doAction returns true if successful, false otherwise
boolean doAction(...) {
    ...
}

// General pattern
boolean success = false
int tries = 0;
while (!success && (tries < MAX_TRIES)) {
    // Should add a pause after first retry
    success = doAction(...);
    tries++;
}

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

- Roll back state and/or execution path to a defined safe state
- Recover from internal errors caused by external failures
- Use checkpoints and safe points to provide safe rollback points
- Limit retries to avoid recurring errors
Roll-forward

- Advance execution past the point of error
- Often used as escalation if retry or rollback do not succeed
- Use checkpoints and safe points to provide safe roll-forward points
- Not applicable (without extra measures) if skipped activity is essential
Reset

- Often used as radical escalation if all other measures failed
- Restart service – do not forget to provide a consistent startup state
- Reset data to a guaranteed consistent state if nothing else helps
- Sometimes simply trying to reconnect helps (often forgotten)
Failover

- Used as escalation if other measures failed or would take too long
- Requires redundancy – trades resources for availability
- Many implementation variants available, incl. out-of-the-box solutions
- Usually implemented as a monitor-dynamic router combination
Fallback

• Execute an alternative action if the original action fails
• Basis for most mitigation patterns
• Fail silently – silently ignore the error and continue processing
• Default value – return a predefined default value if an error occurs
public class DefaultValueCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";
    private final boolean preCondition;

    public DefaultValueCommand(boolean preCondition) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.preCondition = preCondition;
    }

    @Override
    protected String run() throws Exception {
        if (!preCondition)
            throw new RuntimeException("Action failed");
        return "Smart result";
    }

    @Override
    protected String getFallback() {
        return "*Default value*";  // Return default value if action fails
    }
}
@Test
public void shouldSucceed() {
    DefaultValueCommand command = new DefaultValueCommand(true);
    String s = command.execute();

    assertEquals("Smart result", s);
}

@Test
public void shouldProvideDefaultValue() {
    DefaultValueCommand command = new DefaultValueCommand(false);
    String s = null;
    try {
        s = command.execute();
    } catch (Exception e) {
        fail("Did not return default value");
    }
    assertEquals("*Default value*", s);
}
Queues for resources

- Protect resource from temporary overload situations
- Avoid losing requests by queuing them in front of resource
- Unlimited queues can create excessive latency
- Often complemented with backpressure or bounded queue
Backpressure

- Mitigate risk of unlimited input queue growth
- Notify callers to throttle request production
- Requires notification backchannel to callers
- Often used to implement flow control
Bounded queue

- Avoid risk of unlimited input queue growth
- Limit queue size to limit latency at longer-lasting overload
- Finish work in progress – Create pushback or discard new entries
- Fresh work before stale – Discard old entries
Bounded queue example

// Executor service runs with up to 6 worker threads simultaneously
// When thread pool is exhausted, up to 4 tasks will be queued -
// additional tasks are rejected triggering the PushbackHandler
final int POOL_SIZE = 6;
final int QUEUE_SIZE = 4;

// Set up a thread pool executor with a bounded queue and a PushbackHandler
ExecutorService executor =
    new ThreadPoolExecutor(POOL_SIZE, POOL_SIZE, // Core pool size, max pool size
    0, TimeUnit.SECONDS, // Timeout for unused threads
    new ArrayBlockingQueue(QUEUE_SIZE),
    new PushbackHandler);

// PushbackHandler - implements the desired pushback behavior
public class PushbackHandler implements RejectedExecutionHandler {
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        // Implement your pushback behavior here
    }
}
Share load

- Use if additional resources for load sharing are available
- Share load between (added) resources to keep throughput good
- Can be implemented statically or dynamically
- Minimize amount of synchronization needed between resources
Routine maintenance

• Reduce system entropy – keep preventable errors from occurring
• Especially important if errors were only mitigated, not corrected
• Check system periodically and fix detected faults and errors
• Balance benefits, costs and additional system load
Redundancy

- Core resilience concept
- Basis for many recovery and mitigation patterns
- Not only for crash failures, but applicable to all failure types
- Often different variants implemented in a system
Idempotency

- Non-idempotency is complicated to handle in distributed systems
- (Usually) increases coupling between participating parties
- Use idempotent actions to reduce coupling between nodes
- Very fundamental resilience and scalability pattern
Unique request token (sketch)

// Client/Sender part

// Create request with unique request token (e.g., via UUID)
token = createUniqueToken()
request = createRequest(token, payload)

// Send request until successful
while (!successful)
    send(request, timeout) // Do not forget failure handling

// Server/Receiver part

// Receive request
request = receive()

// Process request only if token is unknown
if (!lookup(request.token)) // needs to implemented in a CAS way to be safe
    process(request)
store(token) // Store token for lookup (can be garbage collected eventually)
Stateless

• State is hard to replicate and creates coupling between replicas
• Compromises consistency or availability, failover, relocation, etc.
• Try to move state to the client or the data store
• Very fundamental resilience and scalability pattern
Escalation

• Units often don’t have enough time or information to handle errors
• Escalation peer with more time and information needed
• Often multi-level hierarchies
• Separate error handling flow from processing flow
Escalation implementation using Worker/Supervisor
About using resilience patterns ...
Using resilience patterns

• Patterns are options, not obligations
• Don’t pick too many patterns
• Each pattern increases complexity
• Complexity is the enemy of robustness
• Each pattern costs money in dev & ops
• You only have a certain resilience budget
• Look for complementary patterns
How did other people do it?
Erlang (Akka)

Core patterns
Netflix
Core patterns

- Circuit breaker
- Timeout
- Monitor
- Isolation
- (Micro)Service
- Communication paradigm
- Request/response
- Error injection
- Share load
- Bounded queue
- Redundancy
- Several variants
- Fallback
- Support patterns
- Retry
- Node level
- System level
- Either level
- Core
- Detect
- Prevent
- Mitigate
- Recover
- Complement
- Treat
- Zero downtime deployment
- Canary releases
Designing for resilience
The whole 9 yards
Backup request

- Send request to multiple workers (optionally a bit offset)
- Use quickest reply and discard all other responses
- Prevents latent responses (or at least reduces probability)
- Requires redundancy – trades resources for availability
Marked data

- Avoid repeated and/or spreading errors due to erroneous data
- Use if time or information to correct data immediately is missing
- Mark data as being erroneous – check flag before processing data
- Use routine maintenance job to correct data
Anti-fragility

• Avoid fragility caused by homogenization and standardization
• Protect against disastrous failures by using diverse solutions
• Protect against cumulating effects by introducing jitter
• Balance risks, benefits and added costs and efforts carefully
Error injection

• Make resilient software design sustainable
• Inject errors at runtime and observe how the system reacts
• Can also be used to detect yet unknown failure modes
• Make sure to inject errors of all types
• Chaos Monkey
• Chaos Gorilla
• Chaos Kong
• Latency Monkey
• Compliance Monkey
• Security Monkey
• Janitor Monkey
• Doctor Monkey

https://github.com/Netflix/SimianArmy
Zero downtime deployment

- Improve availability by avoiding deployment downtimes
- Makes bug fixing and continuous delivery in HA systems easier
- Use hot deployments if supported by runtime environment
- Use blue-green deployment, rolling or canary releases otherwise
Relaxed Temporal Constraints

• Strict consistency requires tight coupling of the involved nodes
• Any single failure immediately compromises availability
• Use a more relaxed consistency model to reduce coupling
• The real world is not ACID, it is BASE (at best)!
Towards a pattern language ...
Sorry, not yet ready ... :(
A little design cycle instead ...
1. Design bulkheads using DDD or alike

0. Make initial core architectural decisions

2. Select detection & recovery/mitigation pattern based on failure scenarios

2b. Optionally augment with prevention and complementing patterns

3. Implement

4. Deploy

5. Measure

6. Learn

7. Assess pattern selection value vs. costs

Repeat!

Revisit if needed
1. Design bulkheads using DDD or alike

0. Make initial core architectural decisions

2. Select detection & recovery/mitigation pattern based on failure scenarios

2b. Optionally augment with prevention and complementing patterns

3. Implement

4. Deploy

5. Measure

6. Learn

7. Assess pattern selection value vs. costs

Repeat!

Measure and learn! Don’t guess. Know!

Find the balance!
More is not always better. Keep the business case in mind!

Here be dragons!
Designing bulkheads is about 50% of resilience design and extremely hard!

Revisit if needed
How do I integrate resilience into my software development process?
Adopting resilient software design

- Building awareness
- Building capability
- Building sustainability
Building awareness

- Create awareness on the business level
  - Understand that failures are not avoidable
  - Understand the business case

- Create awareness on developer level
  - Create the required feedback loops
  - Go DevOps
  - Go Site Reliability Engineering
  - Do it your own way ...
  - ... but create the required feedback loops!
Building capability

- **Build knowledge**
  - Trainings
  - Coaching
  - Hackathons
  - ...

- **Build experience**
  - Just do it and learn!
Building sustainability

- **Error injection**
  - Automatically provoke failures in production
  - Observe if the system behaves as expected
  - Improve your resilience design based on lessons
  - Needs thorough monitoring
  - Should be combined with gradual rollouts or alike

- **Fire drills**
  - Failure scenarios that require human intervention
  - Only simulation leader knows script
Wrap-up

- Today’s systems are distributed
- Failures are not avoidable
- Failures are not predictable
- Resilient software design needed
- Start with careful functional design
- Augment with resilience patterns
- Repeat, learn and observe business value
- Establish feedback loop for awareness
Further reading


Do not avoid failures. Embrace them!

Beware of the “100% available” trap!
@ufried