Resilience-Patterns in Cloud-Applications

Kristian Köhler

sourcefellows

> Software design as taught today is terribly incomplete. It talks only about what systems should do. It doesn't address the converse - things systems should not do. Michael T. Nygard

The question is not IF an error occurs, It's about WHEN an error occurs.

With 40000 requests, there are at least 40000 possible errors.

The concept of resilience in the software industry

- **Origins in materials science**
  - Return to original shape after „deformation“, impuls, stress
- **Transactions can be processed despite errors or stress**
  - Short-term failures, load peaks, etc
  - Focus not purely on the stability of the system
  - Goal: users can still get their work done - 'unit of work'
- **The ability of a system to react to unexpected errors**
  - Without the user noticing
  - Possible shutdown/degrade of a service

Integration points are the number-one killer of systems.

Michael T. Nygard

Preventing the spread - Cascading Failures

- **System failures start with a small crack**
  - „System X does not respond fast enough“
  - „Database Y is down“
  - „Message processing runs into an error“
- **„Cracks propagate“**
- **Stop cracks and prevent the spread**
  - Stop cracks from „jumping the gap“
  - Introduce predetermined breaking points
  - „Crackstoppers“ - James R. Chiles

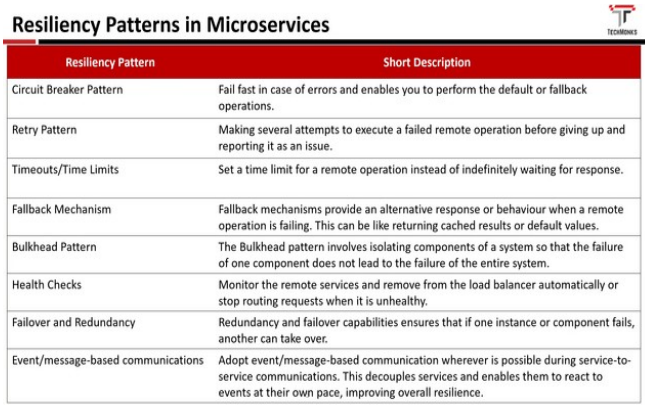
Pattern catalogues and languages

- **Release It! - Design and Deploy Production-Ready Software**

  Michael T. Nygard
- **Microsoft Azure – Microservice Patterns**

  https://learn.microsoft.com/en-us/azure/architecture/patterns/
- **Well architectured Frameworks**

  AWS, Google, Microsoft



Resiliency Patterns in Microservices

Resiliency Pattern	Short Description
Circuit Breaker Pattern	Fail fast in case of errors and enables you to perform the default or fallback operations.
Retry Pattern	Making several attempts to execute a failed remote operation before giving up and reporting it as an issue.
Timeouts/Time Limits	Set a time limit for a remote operation instead of indefinitely waiting for response.
Fallback Mechanism	Fallback mechanisms provide an alternative response or behaviour when a remote operation is failing. This can be like returning cached results or default values.
Bulkhead Pattern	The Bulkhead pattern involves isolating components of a system so that the failure of one component does not lead to the failure of the entire system.
Health Checks	Monitor the remote services and remove from the load balancer automatically or stop routing requests when it is unhealthy.
Failover and Redundancy	Redundancy and failover capabilities ensures that if one instance or component fails, another can take over.
Event/message-based communications	Adopt event/message-based communication wherever is possible during service-to-service communications. This decouples services and enables them to react to events at their own pace, improving overall resilience.

Quelle: https://anjireddy-kata.medium.com/architecture-and-design-101-resiliency-patterns-in-microservices-71029bbb92b7

Who am i?

Kristian Köhler

Source Fellows GmbH
https://www.source-fellows.com
https://www.linkedin.com/in/kristian-köhler/

25+ years in software engineering

Java Enterprise background
Javascript, Python, C#, etc etc

Timeouts

Timeouts

- **Timeout controls cancellation of processing**
  - Blocked threads can make a system hang
    - Deadlocks
  - No more response is expected (server and client side)
- **Timeouts provide isolation from failures**
  - External error does not affect own system
  - External systems are connected via the network
    - Networks can fail (Router, Switch, Firewall, Cable …)
    - External systems themselves can be unstable
      - Events possible at any time
  - Resource Pools can be exhausted

> The Timeouts pattern is useful when you need to protect your system from someone else's Failure.
>
> Michael T. Nygard

Timeouts – Integration Points – What to do?

- **Check each Integration Point separatly**
  - Slow responses can also lead to problems
  - Avoid blocking threads! Everywhere!
- **Think about possible retries**
  - Fast retries are very likely to fail again
    - Use something like „exponential backoffs“
  - Maybe: place request in queue and execute later

Timeouts in libraries

- **Default values in libraries usually suboptimal**
  - Often no timeout configured blocking thread
- **Libraries usually offer good configuration options**
  - Check which timeout values can be set
  - Use suitable values for the usecase (Example: HTTP-Streaming)
- **Each access to a resource should be configured with pooling**
  - Don't wait forever! blocking thread

Timeouts in Go

Go Context - API

- **For deadline or cancellation**
- **Also for call-dependent values**
  - Request-Scoped Values
  - “ThreadLocal”
- **Context should be included with every call**
  - First parameter named „ctx“
  - Propagation through application
  - Implementation in standard library

Go Context – you should know…

- **Context objects are immutable - „Immutable Objects“**
  - Can be passed as parameter to Go-Routinen without problems
  - No synchronisation necessary
- **Context objects form a hierarchie**
  - Propagation of status through hierarchy (Timeout, ...)
- **Information about cancellation of a context via channel**

```
ctx := context.Background()
ctx, cancel := context.WithTimeout(ctx, 2*time.Second)
defer cancel()

<- ctx.Done()
```

Examples in Go APIs

- **Standardlibrary**
  - Network connections (Example net.Dial)
    - Integration in a lot of libraries possible
  - HTTP-Client and server
  - SQL/DB Package
- **NoSQL- Database**
  - MongoDB driver
- **Messaging**
  - Nats.io, Kafka

```
http.NewRequestWithContext(...)
```

```
ctx:= context.Background()
for {
    msg, err :=
            reader.ReadMessage(ctx)
    if err != nil {
        break
    }
    ...
}
```

Sample

Use `context.Context` whenever possible!

HTTP-Client in Go

Go HTTP-Client – With and without context.Context

```go
response, err := http.Get("http://source-fellows.com")
if err != nil {
	log.Fatal(err)
}
defer response.Body.Close()
```

```go
ctx:= context.Background()
req, err := http.NewRequestWithContext(ctx, http.MethodGet, "url", nil)
response, err := http.DefaultClient.Do(req)
if err != nil {
	return
}
defer response.Body.Close()
```

HTTP-Client Timeouts

- **Timeout values can be configured for „HTTP-client“-objects**

  Global configuration of the „DefaultClient“

```
http.DefaultClient.Timeout = 3*time.Second
```

  Use of a separate client for each backend

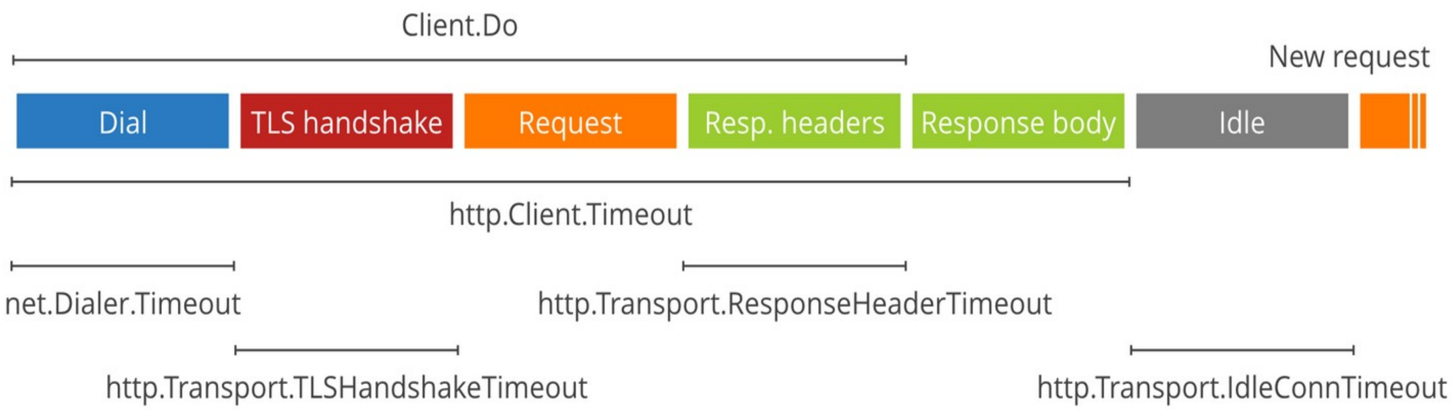
```
client := http.Client{Timeout: 3*time.Second}
client.Do(...)
```

- **Default timeout values not suitable for production**

  Timeout-Value of 0 infinite!

  Context object helps if timeout is specified there

Timeout configuration of the Golang HTTP-Client



Client.Do

New request

Dial

TLS handshake

Request

Resp. headers

Response body

Idle

http.Client.Timeout

net.Dialer.Timeout

http.Transport.ResponseHeaderTimeout

http.Transport.TLSHandshakeTimeout

http.Transport.IdleConnTimeout

Quelle: https://blog.cloudflare.com/the-complete-guide-to-golang-net-http-timeouts/

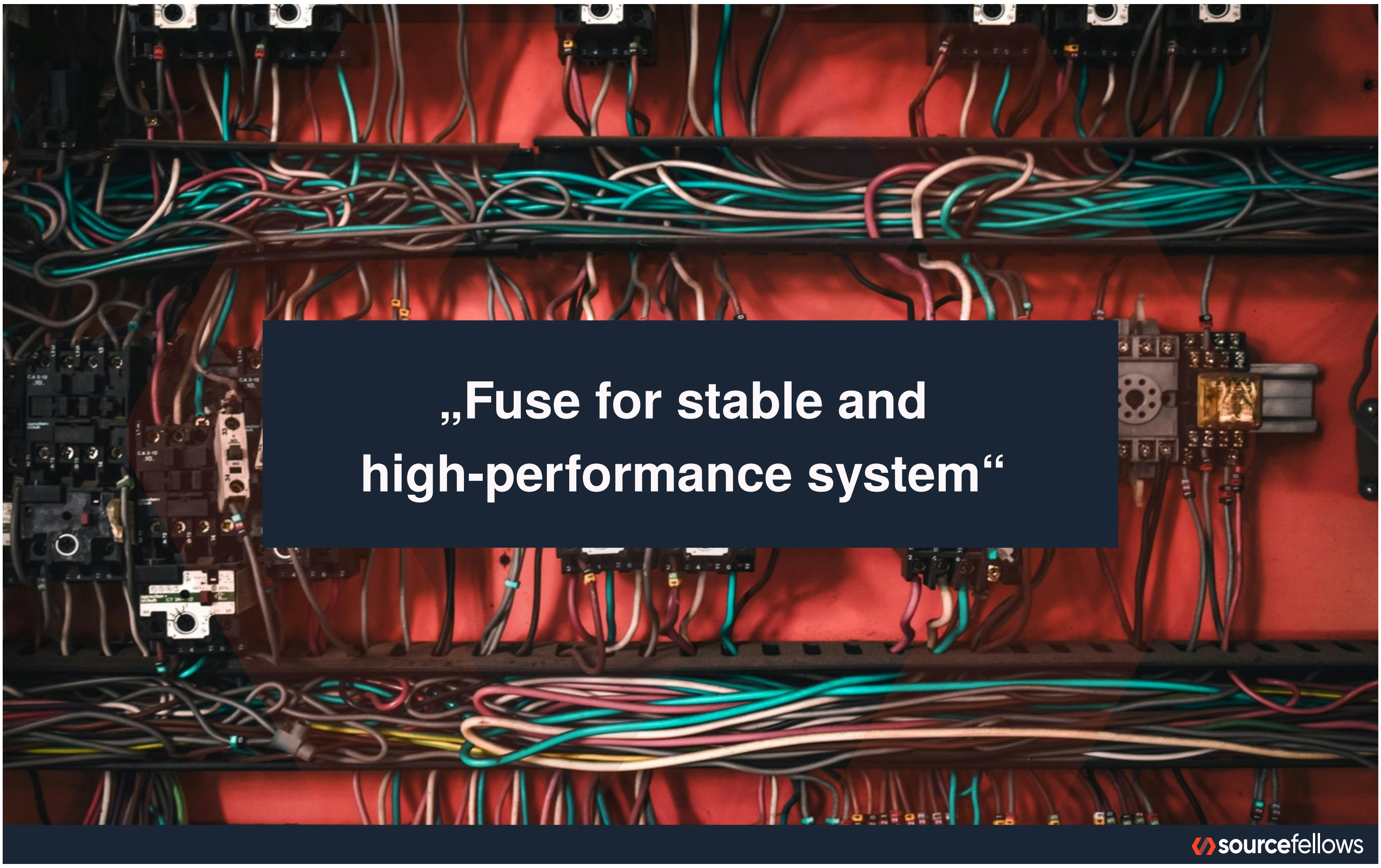
ToxiProxy – Test Harness

- **„Toxiproxy is a framework for simulating network conditions“**
  - A TCP proxy written in Go
  - Manipulate the health via HTTP
- **Created at Shopify**
  - OpenSource - MIT License
  - https://github.com/Shopify/toxiproxy
- **Go and other language client libraries available**

Sample

Always think about Timeouts and configure them accordingly.

Circuit Breaker

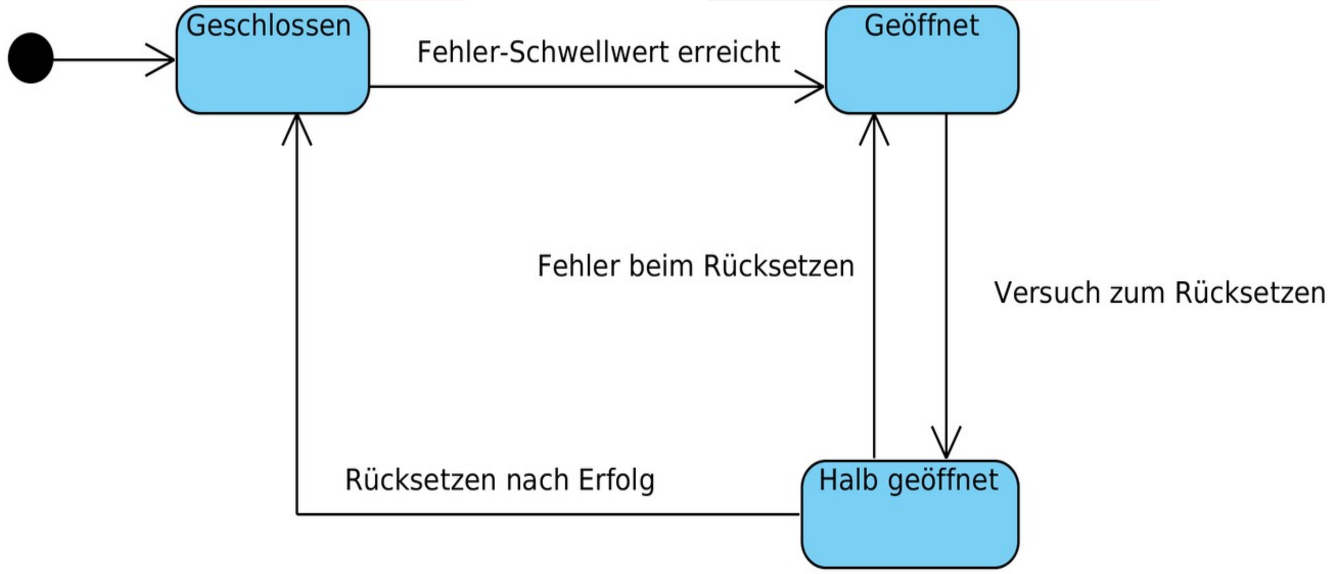


„Fuse for stable and high-performance system“

Circuit Breaker – The fuse for backends

- **When there’s a difficulty with an integration point, stop calling it**
  - Too many or certain errors
- **Use together with useful Timeouts**
  - A timeout indicates that there is a problem with an integration point
  - Blocking calls are not seen as errors without a timeout
- **Expose, track and report status changes on the Circuit Breaker**
  - Indicator for serious problems

Circuit Breaker Pattern



Geschlossen

Fehler-Schwellwert erreicht

Geöffnet

Fehler beim Rücksetzen

Versuch zum Rücksetzen

Rücksetzen nach Erfolg

Halb geöffnet

Circuit Breaker in Go

- **GoBreaker – OpenSource library**

  Circuit Breaker implemented in Go - MIT Lizenz

  https://github.com/sony/gobreaker

- **Wrapping for methods or functions**

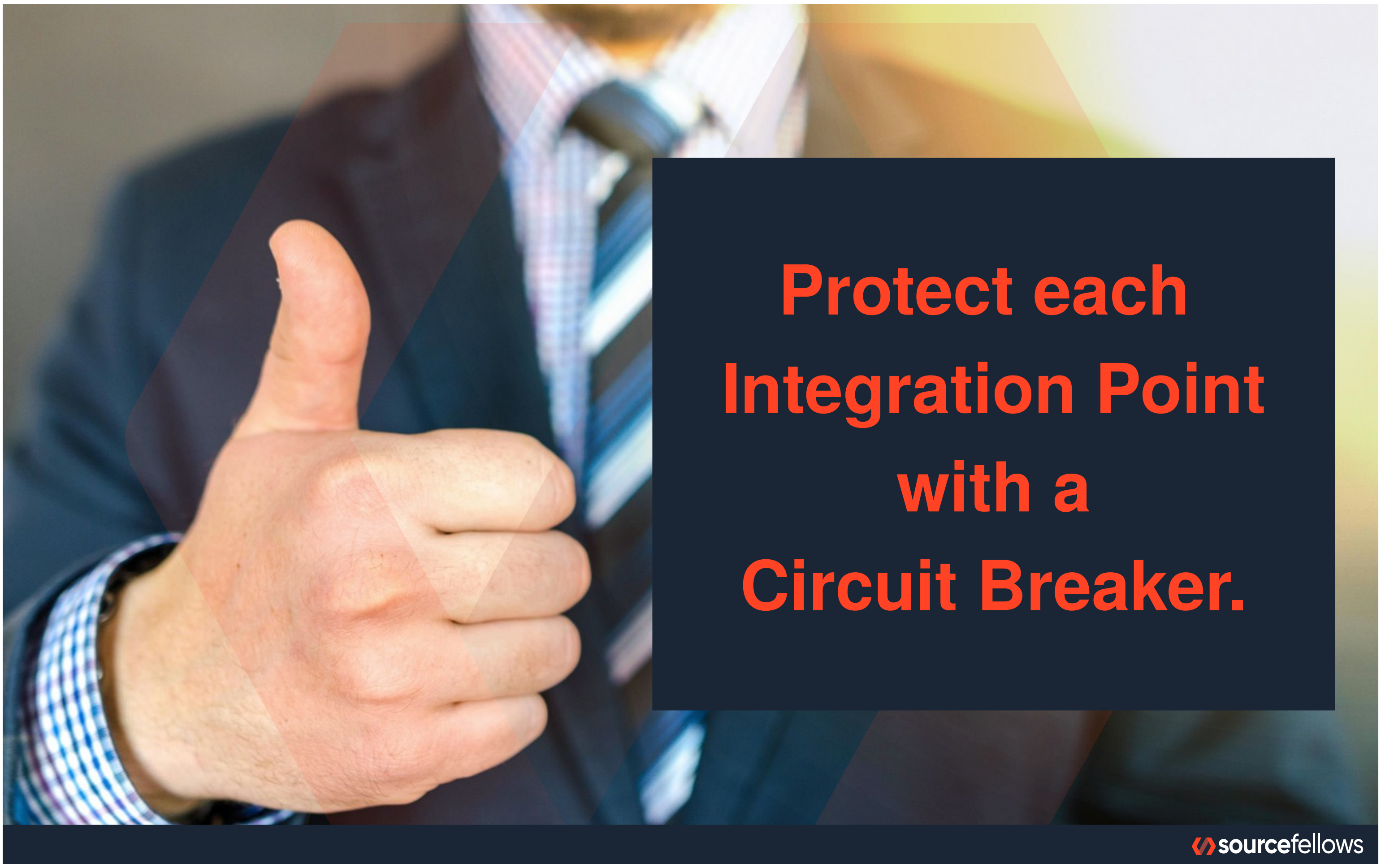
  Any errors that occur are used for status determination

```
func (cb *CircuitBreaker[T]) Execute(req func() (T, error)) (T, error)
```

Sample

Circuit Breaker – Use with caution...

- **Pay attention dependencies**
  - What does the failure mean for other components?
  - Are other components prepared for faults?
- **Think of possible chain reaction**
  - Backend-call always returns an error
    - What impact does this have (for other systems)?
    - Error may occur faster...
  - Possibly stop entire components as a reaction
    - Use context.Context to stop things and possible whole components



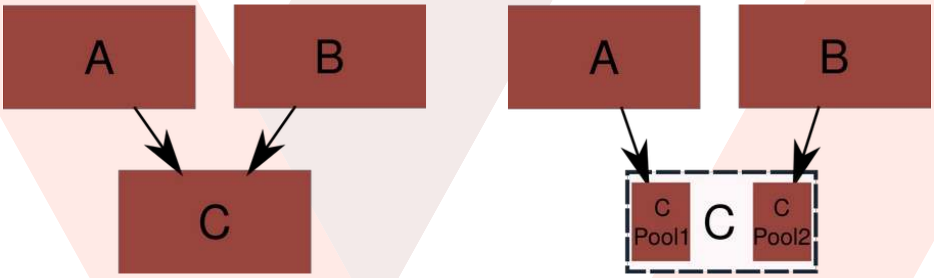
Protect each Integration Point with a Circuit Breaker.

Bulkhead

As in shipping, ‘bulkheads’ are designed to prevent the failure of one component from affecting the entire system.

Bulkhead

- **Partitioning the system**
  - Redundancy of systems is the simplest option
  - Example: Assign servers to specific tasks
  - Separation within applications
- **Dependencies between applications via third-party applications**
  - Separation within application necessary



Bulkhead-Variants

- **Thread Pool Bulkhead**
  - Pools for different tasks or operations (e. g. Frontend / Backend Pools)
- **Service Bulkhead**
  - Separate individual services from each other (e. g. Microservices)
  - „Safety“ through resource requirements
- **Database Bulkhead**
  - Separate connection pools, partitions (e. g. separate read and write operations)
- **Infrastructure Bulkheads**
  - Network, Process, Resource Bulkheads
    - Separation of the connections (e. g. Management-Network)
    - Separation of individual processes on different physical machines
    - Splitting ressources (e. g. CPU, memory, etc)

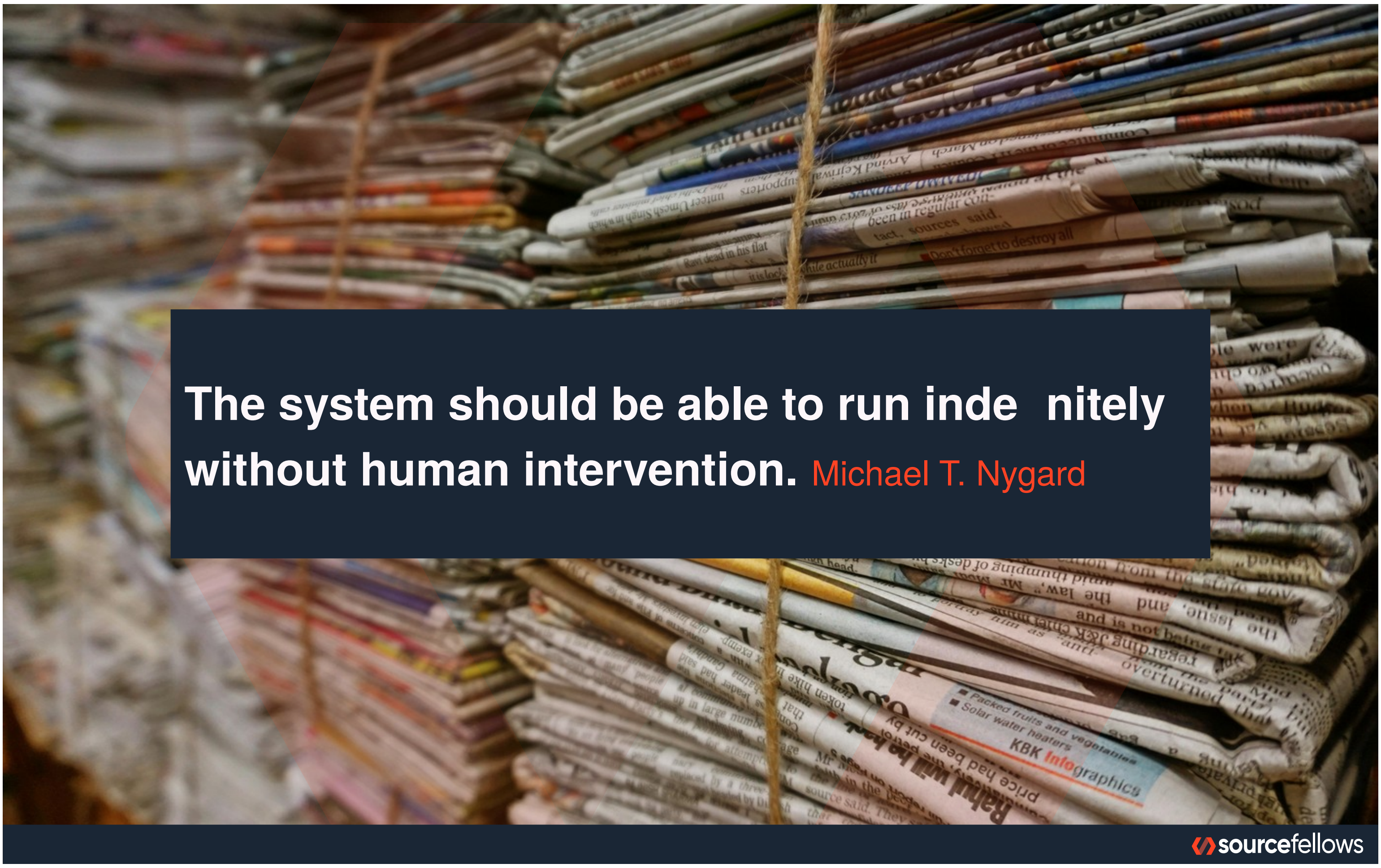
Bulkhead in Go

- **Use separate pools for integration points**
  - HTTP-Client has a pool for server connections
  - Configure own HTTP-Clients
- **Separation of incoming connections**
  - Different ports for different services or management
  - Use special Listener for connection pooling
    - Default implementation is limited through operating system
    - E. g. https://pkg.go.dev/golang.org/x/net/netutil#LimitListener
- **Enable server start even without backend**
  - Lazy-Init, retries, Circuit-Breaker

Sample

Define separate modules/components and make them independent from each other.

Steady State



The system should be able to run indefinitely without human intervention. Michael T. Nygard

Steady State

- **Systems collect data - sometimes without cleanup**

  Logs, Database, User-Uploads, ...
- **Cloud storage tempts you to keep your data - forever**
- **Set up a cleanup for each collection mechanism**

  Delete, compress, archive old data as soon as possible
- **Too much data can lead to instability**

  Long loading times, higher latency, higher load

  Memory runs out while loading...

It sometimes seems that you'll be lucky if the system ever runs at all in the real world. The notion that it will run long enough to accumulate too much data to handle seems like a "high-class problem"—the kind of problem you'd love to have.

Do not allow any effect due to steady state!

- **Integrate Cleanup Jobs**

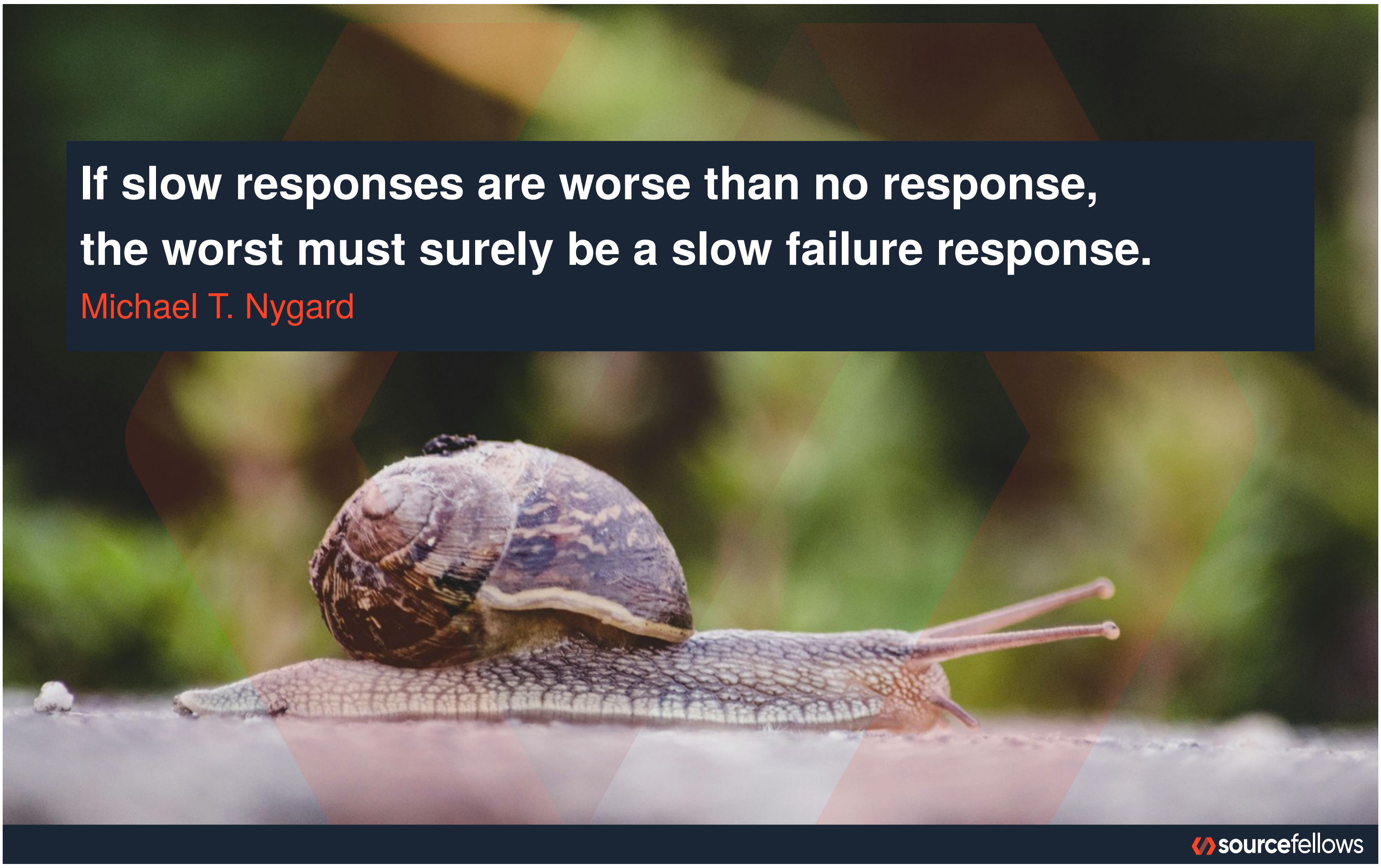
  Plan, implement and check at the beginning

  Determine sensible lifespan and data volume
- **Add Memory-Caches to your applications**

  Define limits for for quantity structure
- **Restrict queries or paging when loading**

  To ensure stability when more data is collected than expected

Fail Fast



> **If slow responses are worse than no response, the worst must surely be a slow failure response.**
>
> Michael T. Nygard

Fail Fast

- **Check whether resources are available before starting a transaction**

  In combination with Circuit Breaker Status

- **Fail as soon and quickly as possible !**

  Do not wait to see if the system reacts after all (E. g. Load-Balancer)

- **Early validation of user input or API parameters**

  Check values as soon as possible e. g. in HTTP-Handler

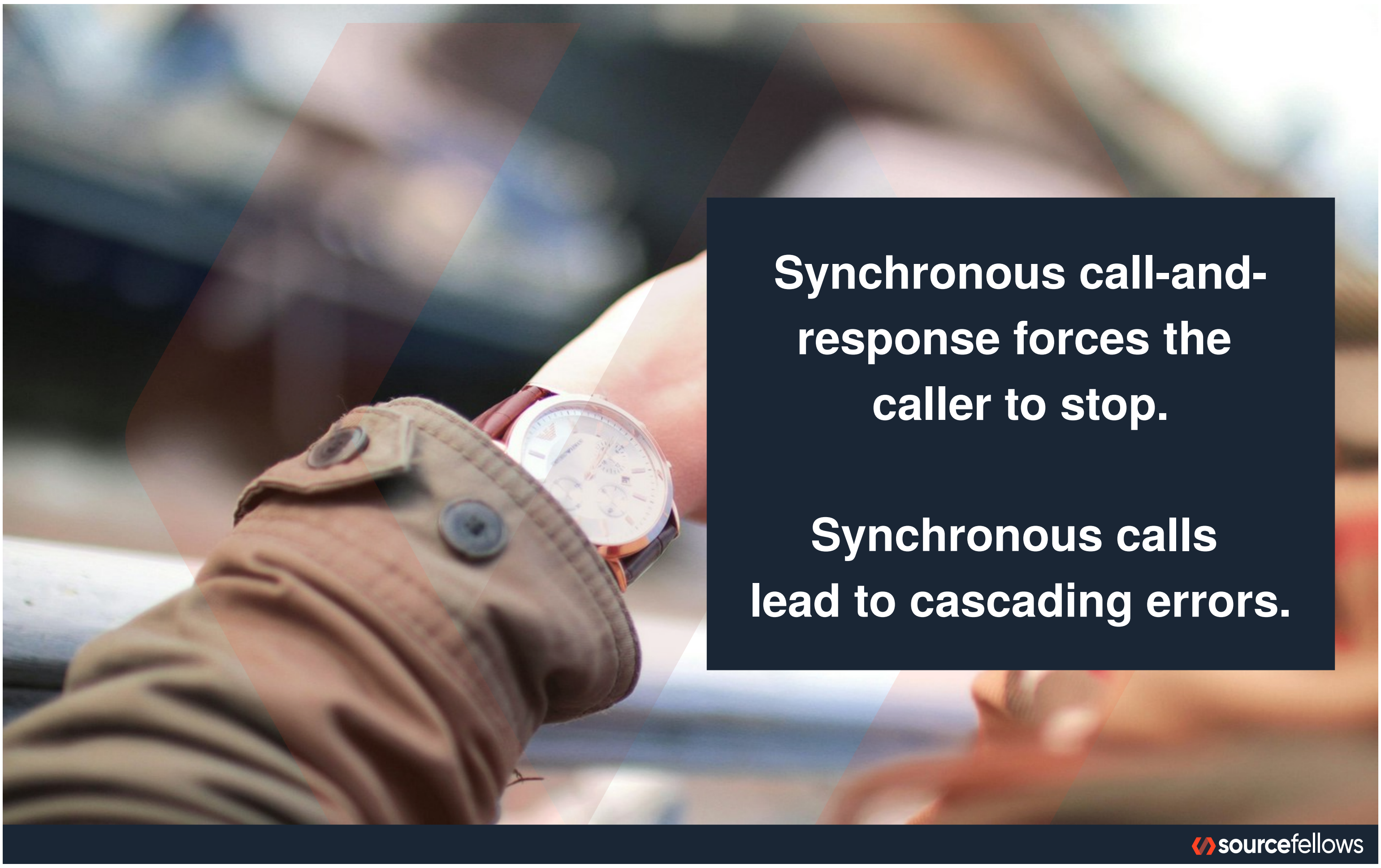
  Reject requests that could cause problems later on

- **Provide appropriate error messages**

  Differentiation between user and system errors

Sample

Decoupling



Synchronous call-and-response forces the caller to stop.

Synchronous calls lead to cascading errors.

Messaging can decouple systems

- **A Central broker takes care of the entire message management**
  - Sender and Consumer send or receive messages
  - Message persistence is possible and enables delayed delivery
- **Messaging leads to decoupled timelines of sender and receiver**
  - Sender does not have to wait for an answer
  - Avoid cascading errors
  - Splits a unit-of-work transaction into several technical transactions
- **Increases the complexity of the application**
  - Response (including errors) must be processed asynchronously
  - New infrastructure components

Sample

Loose coupling with messaging.

Thank you for your attention!