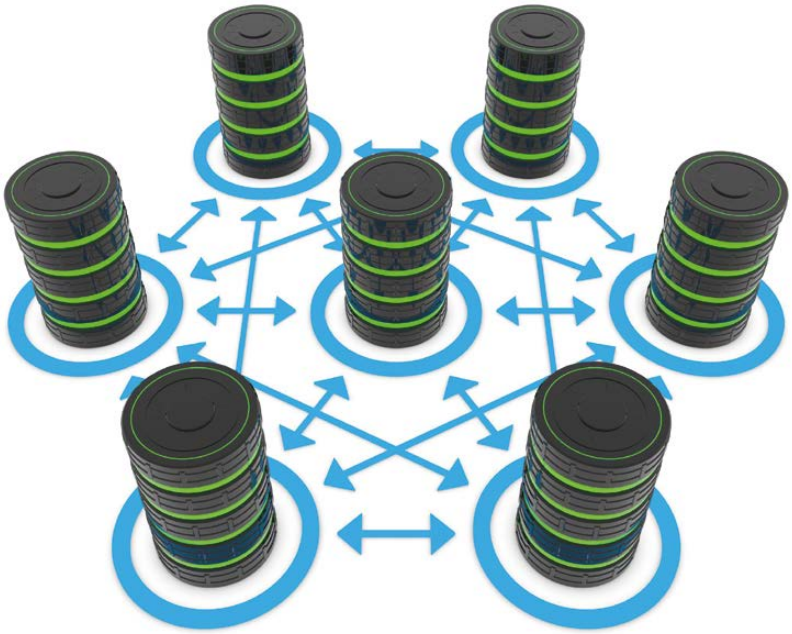
Fazit

Dieser Artikel wird noch keinen Entwickler zum Grafikdesigner machen. Aber der Autor hofft, er konnte hier einen ersten Eindruck und einen Einstieg geben. Wenn das ein oder andere Missverständnis in der Kommunikation mit Grafikdesignern in Zukunft leichter auszuräumen ist oder es gar nicht erst auftritt, sind wir einen großen Schritt weiter, um gemeinsam eine bessere Anwendung zu schaffen. Vielleicht möchte sich ja auch der eine oder andere tiefer mit diesem Thema beschäftigen.

Alexander (Sascha) Klein
alexander.klein@codecentric.de

Alexander (Sascha) Klein ist Niederlassungsleiter der codecentric AG am Standort Stuttgart. Er ist seit etwa zwanzig Jahren im Java-Umfeld als Entwickler, Architekt und Coach tätig. Seine Interessengebiete sind UI-Entwicklung, Ergonomie, DSLs und Produktivität in der Software-Entwicklung. Er ist bekennender Groovy-Jünger und Committer beim Open-Source-Projekt Griffon.



Resilient Software Design Patterns

Thorsten Maier, Orientation in Objects GmbH

Dieser Artikel beschreibt ausgewählte Entwurfsmuster zur Erstellung einer Software, die möglichst widerstandsfähig (resilient) gegenüber dem Ausfall einzelner Teilsysteme ist. Die vorgestellten Muster sind anhand einer kurzen Beispiel-Implementierung konkretisiert.

Software-Systeme werden immer komplexer. Eine klare Strukturierung und damit einhergehend eine Modularisierung ist die wichtigste Grundvoraussetzung, um komplexe Systeme auch langfristig warten zu können. Mit steigender Komplexität steigt auch die Zahl der einzelnen Teile, und der Ausfall eines einzelnen Subsystems wird immer wahrscheinlicher. Ein Ausfall kann beispielsweise durch einen Programmierfehler bei einem Versionsupdate, die Störung des Netzwerks oder aber auch durch die Überlastung der Datenbank entstehen.

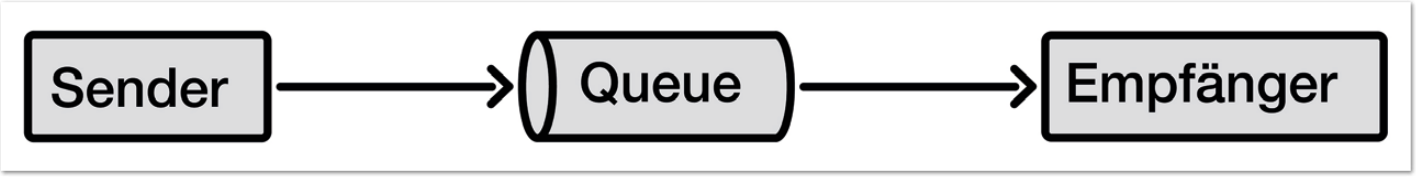


Abbildung 1: Asynchrone Kommunikation

Traditionell haben wir versucht, die Verfügbarkeit einer Software-Lösung durch eine möglichst hohe Verfügbarkeit der einzelnen Teile zu maximieren. Ab einer gewissen Anzahl von Einzelteilen ist dieses Vorgehen allerdings extrem aufwendig, da sich die Wahrscheinlichkeit für einen Ausfall mit jedem Teil erhöht.

Warum entwickeln wir unsere Software zukünftig nicht unter der Annahme, dass der Ausfall eines Teilsystems nicht die Ausnahme, sondern der Normalfall ist? Statt wie bisher zu versuchen, einen Fehler um jeden Preis zu verhindern, versuchen wir stattdessen, möglichst gut auf die auftretenden Fehler und Ausfälle zu reagieren. Genau das ist die Grundidee des Resilience Software Design. Wir entwerfen ein Software-System, das möglichst widerstandsfähig gegenüber Fehlern ist. Dabei helfen uns verschiedene wiederkehrende Muster.

Grundprinzipien

Betrachten wir zunächst das grundsätzliche Vorgehen für den Entwurf eines widerstandsfähigen Systems. Grundlage sind dabei die vier wesentlichen Prinzipien des Resilience Software Designs:

- Isolation
- Lose Kopplung
- Redundanz
- Fallback

Eine widerstandsfähige Anwendung kann nur entstehen, falls wir möglichst isolierte Fehlereinheiten entwerfen, die auch in einer instabilen Umgebung ihren Dienst möglichst lange aufrechterhalten. In jeder Fehlereinheit gelten alle umliegenden Teile per Definition als instabil; daher sollte die Kommunikation nach außen auf ein Mindestmaß reduziert werden. Ist eine Kommunikation notwendig, so sollte jede Einheit jederzeit mit einem Kommunikationsfehler rechnen und eine geeignete Reaktion vorsehen. Die angenommene Instabilität der Subsysteme bedeutet außerdem, dass bei einem Aufruf von außen alle Aufrufparameter validiert werden müssen. Daraus potenziell entstehende Fehler lassen sich so möglichst früh unterbinden.

Damit das Gesamtsystem trotz der Trennung in abgeschottete Einheiten seine Aufgabe erfüllen kann, sind definierte Schnittstellen zwischen den Teilen von entscheidender Bedeutung. Das Ziel der Schnittstellen ist die lose Kopplung der Teile, um deren Unabhängigkeit zu bewahren. Sie sollten möglichst freundlich mit ihren Aufrufern umgehen, sodass das Auftreten von Fehlern im Client möglichst unwahrscheinlich wird. Falls beispielsweise eine grafische Benutzeroberfläche mehrere Millionen Datensätze aus einer Datenbank-Tabelle anfordert, sollte die Menge der ausgelieferten Datensätze entgegen dem Willen des Aufrufers auf einen sinnvollen Wert beschränkt werden. Nur so kann die Reaktionsfähigkeit des Gesamtsystems gewährleistet werden, da der Aufrufer mit hoher Wahrscheinlichkeit ohnehin nicht alle Datensätze in einer sinnvollen Zeit verarbeiten kann.

```
JmsTemplate jmsTemplate = context.getBean(JmsTemplate.
class);
jmsTemplate.convertAndSend("coffeeOrderQueue", new
CoffeeOrder("horsten", "Cappuccino"));
```

Listing 1

```
@JmsListener(destination = "coffeeOrderQueue")
public void receiveMessage(CoffeeOrder coffeeOrder) {
  System.out.println("Received <" + coffeeOrder + ">");
}
```

Listing 2

Um die Verfügbarkeit des Gesamtsystems weiter zu erhöhen, können die lose gekoppelten Teile zudem redundant ausgelegt sein. Da in einer widerstandsfähigen Anwendung ohnehin alle Prozesse mit dem Ausfall der anderen Prozesse und somit der kurzzeitigen Nichtverfügbarkeit rechnen, ist die lastverteilte Umschaltung zwischen mehreren redundanten Kopien der logische nächste Schritt. Zu guter Letzt ist mit einem Fallback-Verhalten auf der Seite des Aufrufers sichergestellt, dass zumindest ein wesentlicher Teil der Funktionalität auch dann noch aufrechterhalten werden kann, wenn trotz Redundanz Teilsysteme vollständig ausfallen.

Design-Muster

Nachdem wir uns die abstrakte Vorgehensweise für die Erstellung einer widerstandsfähigen Anwendung angeschaut haben, folgt eine Auswahl der wichtigsten Muster zur Umsetzung dieses Konzepts. Diese lassen sich grob in zwei Kommunikationsarten einteilen: Wir unterscheiden zwischen asynchroner und synchroner Kommunikation. Asynchrone Aufrufe führen an sich bereits zu einer widerstandsfähigen Anwendung. Bei synchroner Kommunikation kann man die Widerstandsfähigkeit mit Mustern wie Verzeichnisdienst, Circuit Breaker oder clientseitigem Load-Balancing erhöhen.

Alle Konzepte dieses Artikels sind mit einem kurzen Code-Beispiel verdeutlicht. Für die technische Umsetzung bietet sich Spring *[1]* an, da in diesem Framework sowie den zahlreichen Erweiterungen bereits all die vorgestellten Konzepte integriert wurden.

Asynchrone Kommunikation

Vergleichbar mit den Schotten im Schiffsbau soll die Trennung zwischen den isolierten Teilsystemen dafür sorgen, dass niemals das System als Ganzes, sondern höchstens einzelne Teile funktionsunfähig werden. Diese geforderte Isolation kann im einfachsten Fall erreicht werden, indem Sender und Empfänger lediglich indirekt über Nachrichten kommunizieren *(siehe Abbildung 1)*.

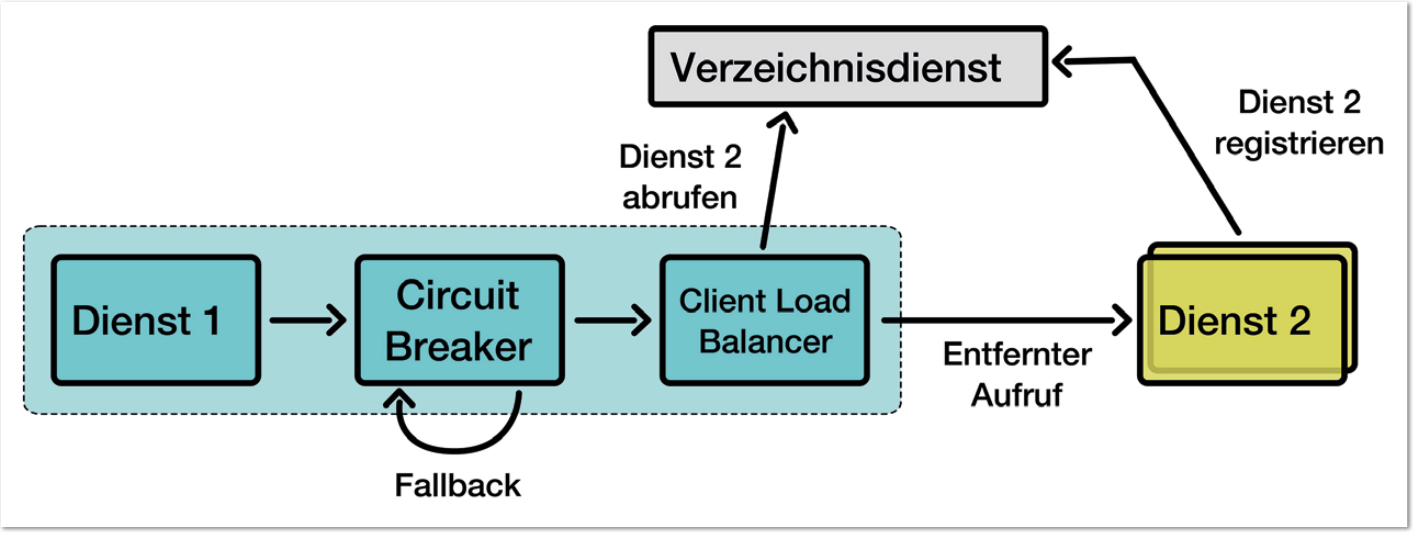


Abbildung 2: Widerstandsfähige synchrone Kommunikation

Die Nachrichten werden in der Queue zwischengespeichert und somit kann der Sender unmittelbar nach dem Versand einer Nachricht mit seiner Arbeit fortfahren; er muss nicht auf die Verfügbarkeit eines Empfängers warten. Gleichzeitig ist auch der Empfänger nicht auf die Verfügbarkeit des Senders angewiesen und kann die Nachricht zu einem für ihn passenden Zeitpunkt verarbeiten.

Kommen wir zur konkreten Implementierung. Der Versand einer asynchronen Nachricht in eine Queue wird bei Spring durch die Klasse „JmsTemplate" unterstützt *(siehe Listing 1)*. Als Empfänger für die Nachrichten einer Queue kann man sich mit der Annotation „@JmsListener" registrieren *(siehe Listing 2)*.

Obwohl die Vorteile der asynchronen Kommunikation über Queues den meisten bekannt sein dürften, wird diese Art der Kommunikation dennoch in traditionellen Systemen verhältnismäßig selten eingesetzt. Ein Grund dafür ist die unter Software-Entwicklern weitverbreitete „transaktionale Denkweise". Damit ist gemeint, dass wir darauf bedacht sind, unseren Datenbestand zu jedem beliebigen Zeitpunkt konsistent zu halten, und dass wir dies über Transaktionen sicherstellen müssen. Genau dies lässt sich mit asynchroner Kommunikation über mehrere Prozesse hinweg allerdings nur schwer erreichen, denn wir müssten unsere Transaktionsgrenze über die beteiligten Prozesse hinweg aufspannen, was technisch zwar nicht unmöglich, aber durchaus anspruchsvoll ist.

```
@SpringBootApplication
@EnableEurekaServer
public class Application {
  public static void main(String[] args) {
     SpringApplication.run(Application.class, args);
  }
}
```

Listing 3

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
```

Listing 4

Bei genauerer Analyse der Anwendungsfälle einer Anwendung sollte man diese transaktionale Denkweise allerdings hinterfragen. Oft ist es ausreichend, wenn die Konsistenz der Daten nicht sofort, sondern erst zu einem späteren Zeitpunkt hergestellt werden kann. Im Gegensatz zu „Strict Consistency", bei der die Konsistenz am Ende jeder Aktion sichergestellt wird, reicht es bei „Eventual Consistency", dass die Daten schlussendlich in naher Zukunft konsistent sind.

„Eventual Consistency" lässt sich mithilfe asynchroner Kommunikation sicherstellen, während für „Strict Consistency" auf synchrone Kommunikation gesetzt werden muss. Im Blog-Artikel „Starbucks Does Not Use Two-Phase Commit" *[2]* ist diese Thematik anhand einer Kaffeebestellung sehr anschaulich erläutert. Zusammenfassend ist beschrieben, dass aus Gründen der Durchsatzoptimierung und der Fehlertoleranz ein asynchroner Kommunikationsansatz mit einer Becherwarteschlange beim Bestellvorgang eines Kaffees deutliche Vorteile bietet.

Was können wir daraus lernen? Wir sollten für jeden Anwendungsfall analysieren, ob „Eventual Consistency" gegebenenfalls ausreichend ist. Dann können wir durch den Einsatz von asynchroner Kommunikation ein deutlich widerstandsfähigeres Gesamtsystem entwerfen.

Widerstandsfähige synchrone Kommunikation

Selbstverständlich ist es nicht immer möglich, auf asynchrone Kommunikation zu setzen. Daher müssen wir in diesen Fällen auf synchrone Aufrufe zurückgreifen. Der entscheidende Nachteil im Sinne einer widerstandsfähigen Anwendung ist dabei, dass für eine erfolgreiche Kommunikation beide Partner gleichzeitig kommunikationsbereit sein müssen. Dies wiederum ist in einem System, in dem wir alle Kommunikationspartner per Definition als unzuverlässig einstufen, nicht immer gegeben. Wir benötigen daher Mechanismen, mit denen wir auch diese Art der Kommunikation widerstandsfähig gegen Ausfälle machen können.

Abbildung 2 zeigt den schematischen Aufbau einer fehlertoleranten synchronen Kommunikation. Dienst 1 möchte mit Dienst 2 kommu-

nizieren. Die Widerstandsfähigkeit wird hierbei durch den Einsatz eines Verzeichnisdienstes, eines Circuit Breakers und eines clientseitigen Load-Balancer erreicht.

Verzeichnisdienst

Der Verzeichnisdienst sorgt dafür, dass Dienst 1 nicht wissen muss, wo Dienst 2 gerade ausgeführt wird. Soll eine Kommunikation aufgebaut werden, wendet sich Dienst 1 über den clientseitigen Load-Balancer – der im Anschluss noch vorgestellt wird – an den Verzeichnisdienst und ruft von dort eine oder auch mehrere Instanzen von Dienst 2 ab. Auch wenn Dienst 2 seit der letzten Kommunikation umgezogen ist oder durch eine andere Instanz ersetzt wurde, kann der anschließende, entfernte Aufruf trotzdem fehlerfrei ausgeführt werden.

Mit Eureka *[3]* können wir auf einen fertigen Verzeichnisdienst zurückgreifen, der bei Netflix zum Einsatz kommt und somit bereits gezeigt hat, dass er auch für sehr große Systeme bestens geeignet ist. Mit Spring Cloud Netflix können wir Eureka auf einfache Art und Weise in unsere bestehende Infrastruktur einbinden. *Listing 3* zeigt die Erstellung eines Verzeichnisdienst-Servers als Spring-Anwendung.

Die einzelnen Dienste registrieren sich anschließend automatisch während des Startvorgangs beim Eureka-Verzeichnisdienst. Damit dies funktioniert, muss lediglich die passende Eureka-Client-Implementierung im Klassenpfad des Dienstes vorhanden sein. *Listing 4* zeigt die entsprechende Maven-Dependency.

Circuit Breaker

Durch den Einsatz eines Verzeichnisdienstes erreichen wir eine Ortstransparenz für entfernte Dienste. Problematisch ist dies allerdings, wenn ein solcher Dienst unmittelbar nach dem Auffinden ausfällt oder umzieht. Für diesen Fall benötigen wir einen zweiten Sicherungsmechanismus. Zwischen den beiden Diensten kann zusätzlich ein sogenannter „Circuit Breaker" *[4]* eingesetzt werden. Dieser überwacht die Kommunikation zwischen beiden Kommunikationsseiten und entbindet den Aufrufer somit von der Fehlerbehandlung. Falls der entfernte Dienst zu spät oder überhaupt nicht

```
@HystrixCommand(fallbackMethod = "fallback")
public String readString() {
  return remoteServiceCall();
}

public String fallback() {
  return "Fallback result";
}
```

Listing 5

reagiert, kann der Circuit Breaker diese Fehlersituation erkennen und eine Fallback-Antwort an den Aufrufer liefern. Für die praktische Umsetzung dieses Konzepts bedienen wir uns wieder einer von Netflix zur Verfügung gestellten Open-Source-Bibliothek namens Hystrix *[5]*. *Listing 5* zeigt, wie wir damit eine solche Sicherung per Annotation konfigurieren können.

Die Annotation „@HystrixCommand" erstellt automatisch einen Proxy, der zur Absicherung des entfernten Methodenaufrufs eingesetzt wird. Falls der Aufruf „remoteServiceCall()" nach einem konfigurierbaren Timeout (Default: 1 Sekunde) kein Ergebnis liefert, wird automatisch die Methode „fallback" aufgerufen und deren Rückgabewert an den Aufrufer geliefert. Die mit dieser Annotation konfigurierte Sicherung verwaltet einen internen Zustand und verfügt somit über eine gewisse Intelligenz. *Abbildung 3* zeigt die möglichen Zustände und die zugehörigen Zustandsübergänge.

Im Zustand „Geschlossen" werden alle Aufrufe an den entfernten Dienst weitergeleitet und dessen Reaktion abgewartet. Falls mehr als 50 Prozent der letzten 20 Aufrufe länger als eine Sekunde gedauert haben, springt die Sicherung auf und wechselt in den Zustand „Offen". Dieser Zustandsübergang findet bewusst nicht bei der ersten Überschreitung des Timeouts statt, damit einzelne Ausreißer keine Auswirkung auf das Verhalten des Systems haben.

Im Zustand „Offen" wird versucht, den entfernten Dienst nicht unnötig mit Aufrufen zu belasten; daher werden fünf Sekunden lang alle Anfrage mit dem Fallback-Wert beantwortet. Nach dieser War-

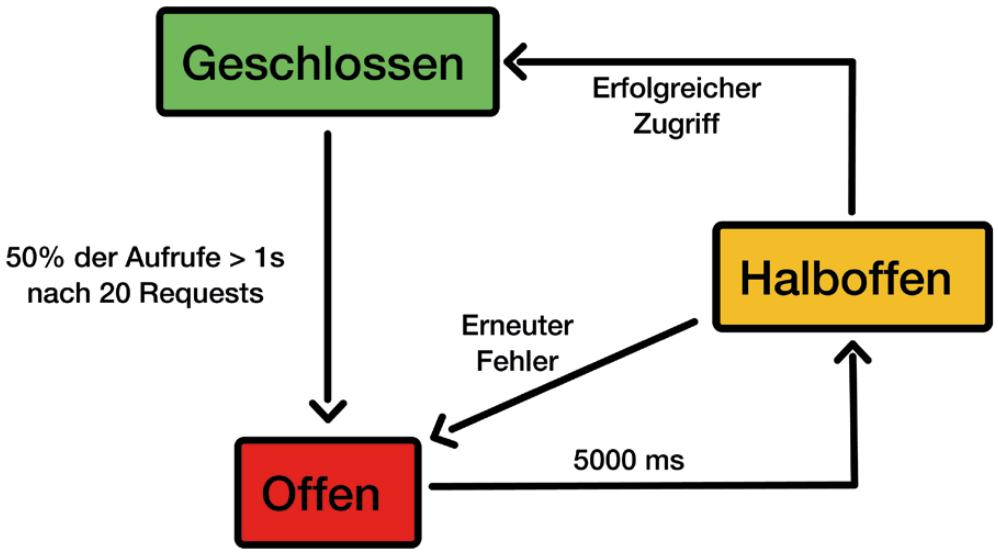


Abbildung 3: Hystrix-Zustände

tezeit findet ein automatischer Übergang in den Zustand „Halboffen" statt. Nun stellt ein erneuter Zugriff auf den entfernten Dienst fest, ob dieser wieder verfügbar ist. Ist dies der Fall, wird die Sicherung geschlossen; andernfalls erneut geöffnet und es findet nach weiteren fünf Sekunden Wartezeit der nächste Testaufruf statt. Die Default-Werte für die Timeouts sowie für den prozentualen Anteil der erlaubten Timeout-Überschreitungen lassen sich per Konfiguration anpassen.

Clientseitiges Load-Balancing

Durch den Einsatz des Verzeichnisdienstes und des Circuit Breaker konnten wir bereits eine gewisse Ausfallsicherheit erreichen. Allerdings können wir damit noch nicht auf unterschiedliche Lastanforderungen reagieren. Die Last kann auf mehrere redundante Instanzen des entfernten Dienstes verteilt werden. Ist er zustandslos implementiert, lässt er sich problemlos mehrfach starten. Klassischerweise hat man eine solche Lastverteilung durch die Zwischenschaltung eines getrennten Load-Balancer erreicht. Bei vielen einzelnen Diensten führt dies allerdings zu einem hohen Konfigurationsaufwand. Daher wollen wir uns hierzu eine leichtgewichtige und dennoch widerstandsfähige Alternative anschauen. Dabei verwenden wir einen im Aufrufer eingebauten, clientseitigen Load-Balancer. Dieser ruft sich vom Verzeichnisdienst alle zur Verfügung stehenden Instanzen des Dienstes ab und verteilt dann selbstständig die Aufrufe an die vorhandenen Instanzen. *Listing 6* zeigt den Einsatz dieses Konzepts bei einem weiteren Netflix-Projekt mit dem Namen Ribbon *[6]*.

„RestTemplate" ist eine Hilfsklasse von Spring, mit der per HTTP-Aufrufen ein Rest-Service abgefragt werden kann. Die beiden Annotationen „@LoadBalanced" und „@RibbonClient" sorgen nun dafür, dass alle Instanzen von „dienst2" über den Verzeichnisdienst abgerufen werden und der Aufruf der URL *„http://dienst2/"* automatisch auf diese Instanzen verteilt wird. Wann welche Instanz aufgerufen wird, lässt sich über den konfigurierbaren Load-Balancer-Algorithmus beeinflussen.

Die „RoundRobinRule" verteilt die Anfragen beispielsweise reihum; „RandomRule" wählt dagegen bei jedem Aufruf zufällig eine Instanz aus. Am interessantesten für den Einstieg ist allerdings die „WeightedResponseTimeRule". Dabei werden die Antwortzeiten der verschiedenen Instanzen analysiert und die Aufrufhäufigkeiten diesen Zeiten angepasst. Die schnellste Instanz bekommt verhältnismäßig die meisten Aufrufe. Aber auch langsame Instanzen werden hin und wieder aufgerufen, um die schnellen Instanzen nicht zu überlasten.

```
@RibbonClient(name = "dienst2")
public class Application {
  @LoadBalanced
  @Bean
  RestTemplate restTemplate() {
     return new RestTemplate();
  }

  public String remoteServiceCall() {
     return restTemplate().getForObject("http://di-
enst2/", String.class);
  }
}
```

Listing 6

Durch die ständige Analyse der Antwortzeiten reguliert sich das System selbstständig und kann daher auf eine zentrale Steuerung verzichten.

Fazit

Eine Anwendung ist mithilfe der in diesem Artikel gezeigten Muster widerstandsfähig gegenüber Fehlern. Am einfachsten gelingt dies durch den Umstieg auf asynchrone Kommunikation, da hierdurch automatisch isolierte und nur lose miteinander gekoppelte Teilsysteme entstehen. Aber auch synchrone Kommunikation kann wie gezeigt unter Einsatz eines Verzeichnisdienstes, eines Circuit Breaker und durch clientseitiges Load-Balancing resilient gemacht werden.

Neben den hier beschriebenen wichtigsten Mustern gibt es viele weitere, die uns bei der Implementierung eines möglichst stabilen Systems helfen können. Dazu gehören beispielsweise ein Konfigurationsserver, ein API Gateway und ein externer HTTP-Session-Speicher. Zu bedenken ist allerdings wie immer, dass jedes neu verwendete Muster Einarbeitungs- und Implementierungsaufwände zur Folge hat und dass wir uns daher in jeder Anwendung auf ein Neues für die richtigen Muster entscheiden müssen.

Weitere Informationen

[1] http://projects.spring.io/spring-framework
[2] *http://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html*
[3] https://github.com/Netflix/eureka
[4] https://martinfowler.com/bliki/CircuitBreaker.html
[5] https://github.com/Netflix/Hystrix
[6] https://github.com/Netflix/ribbon

Thorsten Maier
Thorsten.Maier@oio.de

Thorsten Maier arbeitet bei OIO Orientation in Objects GmbH in Mannheim. Er erschließt kontinuierlich bessere Wege, Software zu entwickeln, indem er selbst als leidenschaftlicher Software-Entwickler mit Java und JavaScript unterwegs ist und anderen als Berater, Trainer und Autor dabei hilft. Trotz seiner Begeisterung für Neues sind ihm Menschen stets wichtiger als Technologien. Sein Hauptaugenmerk liegt daher auf der Frage, wie sich modernste Technologien in gewachsene Umgebungen einbinden lassen und wann man besser auf Bestehendes zurückgreifen sollte.