entwickler.press
shortcuts

# Resilience

## Wie Netflix sein System schützt

Uwe Friedrichsen, Stefan Toth,
Eberhard Wolff

*Uwe Friedrichsen, Stefan Toth, Eberhard Wolff*

# Resilience

## Wie Netflix sein System schützt

ISBN: 978-3-86802-561-3



Ein Imprint der Software & Support Media GmbH

# Inhaltsverzeichnis

# 1 Eine kurze Einführung in Resilient Software Design

In letzter Zeit hört man immer häufiger den Begriff „Resilience“ und manchmal auch etwas vollständiger „Resilient Software Design“. Irgendwie scheint es mit dem Umgang mit Fehlern zur Laufzeit zusammenzuhängen. Aber was daran ist so neu und anders, dass man dafür einen neuen Begriff prägen muss? Oder hat man wieder nur alten Wein in neue Schläuche gefüllt? Zeit für eine kurze Einführung: Worum geht es? Was ist anders? Und wie fühlt es sich an?

Wo fängt man am besten an, wenn man eine kurze Einführung in Resilient Software Design schreiben will? Wahrscheinlich am besten beim Wert von Software: Der primäre Zweck aller Geschäftsprozesse und der sie implementierenden und unterstützenden IT-Systeme ist es, Geld zu erwirtschaften bzw. Kundenbedürfnisse zu befriedigen – am besten beides. Das funktioniert aber nur, solange die IT-Systeme zuverlässig und – von außen betrachtet – fehlerfrei laufen.

Sind die Systeme nicht verfügbar oder fehlerhaft, sind die Kunden unzufrieden, und man verdient kein Geld mit ihnen – kurzum: Sie sind wertlos. Nur zuverlässig laufende Systeme haben Wert. Die Verfügbarkeit von Systemen in Produktion ist also essenziell für den Wert der Software. (Bei manchen Systemen wie z. B. eingebetteten Systemen geht es häufig sogar um viel mehr als nur einen monetären Wert, aber das wollen wir an dieser Stelle nicht näher betrachten.)

## Verfügbarkeit und Fehlertypen

Was aber ist *Verfügbarkeit* genau? Die Verfügbarkeit *A* (für *Availability*, den englischen Begriff für Verfügbarkeit) ist definiert als [1] $A := MTTF / (MTTF + MTTR)$. Darin bedeuten:

- *MTTF (Mean Time To Failure):* die durchschnittliche Zeit vom Beginn des ordnungsgemäßen Betriebs eines Systems bis zum Auftreten eines Fehlers
- *MTTR (Mean Time To Recovery):* die durchschnittliche Zeit vom Auftreten eines Fehlers bis zur Wiederherstellung des ordnungsgemäßen Betriebs des Systems

Während der Nenner also die gesamte Zeit beschreibt, beschreibt der Zähler den Teil der Zeit, in dem das System ordnungsgemäß funktioniert. Damit kann die Verfügbarkeit Werte zwischen 0 für „gar nicht verfügbar“ und 1 für „immer verfügbar“ annehmen. Wenn man den Wert mit 100 multipliziert, erhält man die vertrautere Darstellung als Prozentwert.

Bevor wir uns der Frage zuwenden, wie man die Verfügbarkeit maximieren kann, ist es sinnvoll, einen kurzen Blick auf die möglichen Arten von Fehlern zu werfen, die die Verfügbarkeit kompromittieren können. Die eingängige Literatur (siehe z. B. [2]) unterscheidet fünf Fehlertypen:

1. *Crash Failure* (Absturzfehler): Ein System antwortet permanent nicht mehr, hat bis zum Zeitpunkt des Ausfalls aber korrekt gearbeitet.
2. *Omission Failure* (Auslassungsfehler): Ein System reagiert auf (einzelne) Anfragen nicht, sei es, dass es die Anfragen nicht erhält oder keine Antwort sendet.
3. *Timing Failure* (Antwortzeitfehler): Die Antwortzeit eines Systems liegt außerhalb eines festgelegten Zeitintervalls.
4. *Response Failure* (Antwortfehler): Die Antwort, die ein System gibt, ist falsch.
5. *Byzantine Failure* (byzantinischer/zufälliger Fehler): Ein System gibt zu zufälligen Zeiten zufällige Antworten („es läuft Amok“).

Wenn man von Fehlern im Kontext von Verfügbarkeit spricht, denken die meisten Personen nur an Absturzfehler. Es ist aber wichtig zu berücksichtigen, dass alle Fehlerklassen in die Verfügbarkeit einfließen. Es geht also nicht nur darum, die relativ einfach zu handhabenden Absturzfehler zu behandeln, sondern alle Fehlertypen, also z. B. auch zu langsame oder falsche Antworten, zu erkennen und damit umzugehen.

## Traditionelle Stabilitätsansätze

Aber wie maximiert man jetzt die Verfügbarkeit? Schaut man sich die zuvor beschriebene Formel für Verfügbarkeit an, gibt es dafür zwei Möglichkeiten: Entweder man versucht, den Wert für *MTTF* zu maximieren, oder man versucht, den Wert für *MTTR* zu minimieren. In beiden Fällen entwickelt sich der Wert von *A*, d. h. die Verfügbarkeit gegen 1, was der gewünschte Effekt ist.

Der traditionelle Ansatz ist, das Eintreten eines Fehlers möglichst lange hinauszuzögern, d. h., den Wert für *MTTF* so groß zu machen, dass der Wert für *MTTR* unbedeutend wird (**Abb. 1.1**). Bei diesen traditionellen Stabilitätsansätzen treibt man dafür meist einen großen Aufwand auf Infrastrukturebene: Es wird redundante Hardware eingesetzt, man betreibt HA-Cluster (High-Availability), es werden mehrfache Netzwerkverbindungen (hoffentlich über verschiedene Switches) verwendet usw.

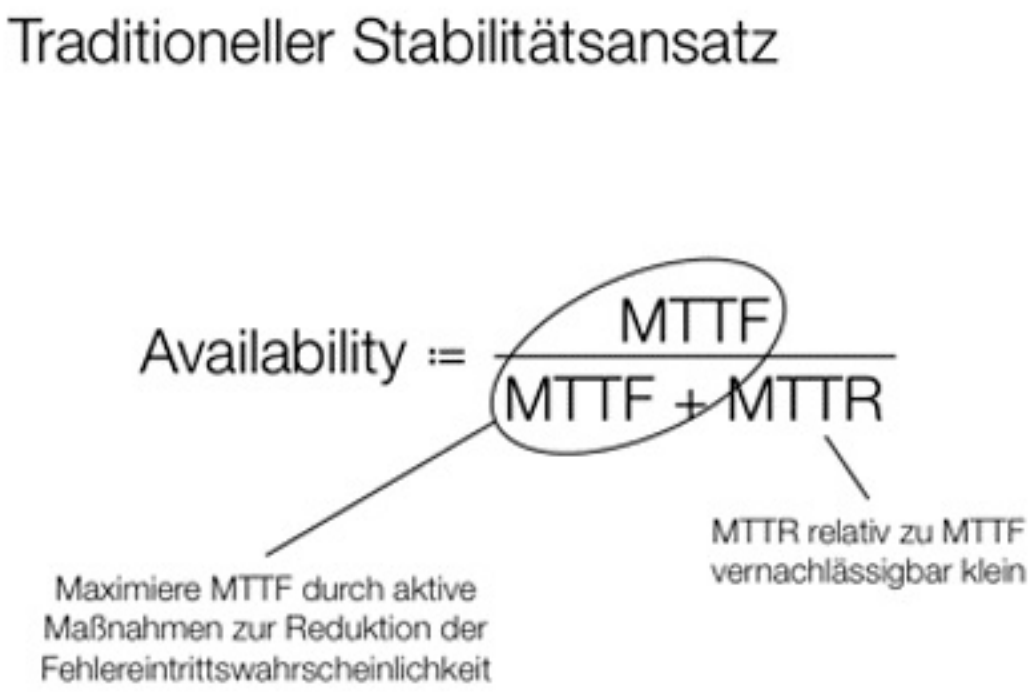


**Abbildung 1.1:** Der traditionelle Stabilitätsansatz

Dazu kommen aufwändige Verfahren zur Vermeidung von Fehlern auf der Softwareebene. Kurzum: Man versucht, die möglichen Fehlerquellen zu antizipieren und ergreift im Vorfeld Maßnahmen, um die Wahrscheinlichkeit des Eintretens solcher Fehler zu minimieren.

Das ist ein durchaus valider Ansatz, solange das betrachtete System relativ isoliert läuft und es nur wenige Faktoren gibt, die die Verfügbarkeit des Systems beeinflussen. Dieser Ansatz spiegelt sich auch in den bekannten Softwarequalitätsstandards wider. So findet sich z. B. im Softwarequalitätsstandard ISO/IEC 25010:2011(en) [3] unter *Reliability* (Zuverlässigkeit) der Elterncharakteristik zu *Availability* (Verfügbarkeit) die folgende Definition: „Reliability: degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.“

Wie man in der zweiten Hälfte der Definition lesen kann, ist die zugrunde liegende Annahme, dass es möglich ist, die Rahmenbedingungen für den Betrieb eines Systems deterministisch festzulegen.

## Das Ende des Determinismus

Wo ist das Problem bei dieser Annahme und den darauf basierenden Stabilitätsansätzen? Das Problem liegt darin, dass heute so gut wie jedes System (in einem Unternehmen) ein verteiltes System ist. Selbst eine einfache Webanwendung besteht in der Regel aus einem Webserver, einem Application Server und einer Datenbank (Firewalls, Reverse Proxies, Load Balancer, eventuelle Cacheserver, Router und Switches nicht mitgezählt).

Und schaut man sich z. B. ein CRM-System bei einem größeren Unternehmen an, dann ist dieses System in der Regel mit mehreren Dutzend anderen Systemen verbunden – vielfach online. Wir haben es heute also mit hochgradig komplexen, verteilten Systemlandschaften zu tun.

Zu verteilten Systemen hat Leslie Lamport, einer der führenden Köpfe auf diesem Gebiet, einmal etwas salopp, aber treffend gesagt: „A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Was er damit ausdrücken will, ist, dass es so viele potenzielle Fehlerquellen in verteilten Systemen gibt, die das korrekte Funktionieren eines Systems kompromittieren können, dass es unmöglich ist, diese alle zu antizipieren und zu vermeiden. Anders formuliert: Fehler sind in verteilten Systemen der Normalfall, nicht die Ausnahme, und es ist nicht möglich, sie vorherzusagen.

Und es wird noch „schlimmer": Entwicklungen wie Cloud Computing, immer höhere Verfügbarkeitsanforderungen, Mobile und Internet of Things sowie Social Media sorgen für immer komplexere, immer höher vernetzte Systemlandschaften mit immer mehr beteiligten Systemen und immer weniger vorhersehbaren Zugriffs- und Lastmustern.

Der angenommene Determinismus der Stabilitätsansätze lässt sich also nicht mehr aufrechterhalten. Es gibt keine deterministisch definierbaren Rahmenbedingungen in heutigen Systemlandschaften mehr, wie sie in den Softwarequalitätsstandards gefordert werden. Wir haben es mit – teilweise unbekannten – Wahrscheinlichkeiten zu tun. Der Determinismus ist einem Probabilismus gewichen.

## Der Resilience-Ansatz

Das führt zum Mantra von Resilient Software Design: *Versuche nicht, Fehler zu vermeiden. Akzeptiere, dass sie geschehen und gehe damit um* (im Englischen etwas kompakter: *Do not try to avoid failures. Embrace them.*).

Folgerichtig versucht Resilient Software Design nicht, *MTTF* zu maximieren, da die Grundannahme ist, dass Fehler im Allgemeinen weder vermeidbar noch vorhersehbar sind, sondern einfach passieren – sprich: *MTTF* wird als nicht beeinflussbar hingenommen.

Um die Verfügbarkeit trotzdem zu maximieren, muss also versucht werden, *MTTR* zu minimieren, d. h. die Zeit vom Auftreten eines Fehlers bis zu seiner Behebung (oder akzeptablen Eindämmung) zu minimieren (**Abb. 1.2**).

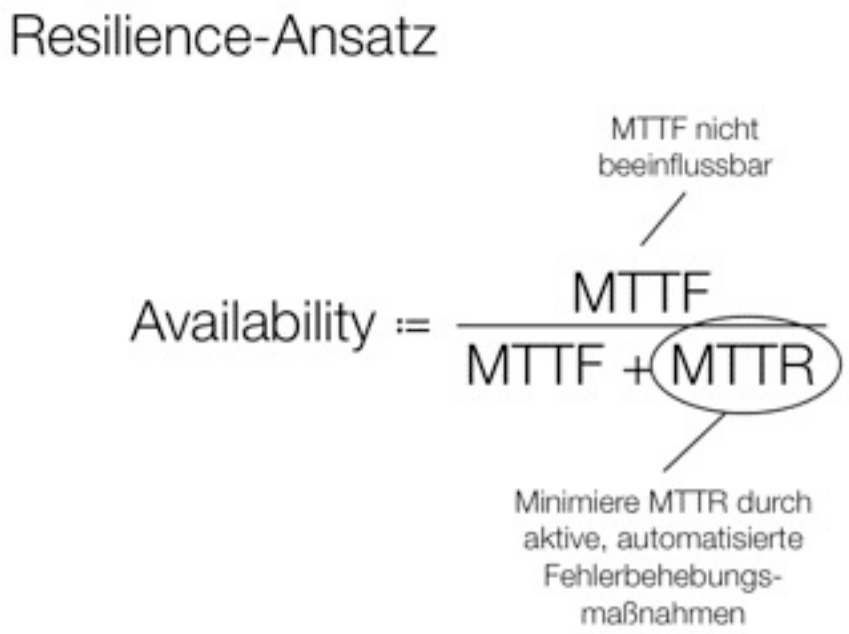


**Abbildung 1.2:** Der Resilience-Ansatz

Damit kann man Resilience folgendermaßen definieren: *Resilience: Die Fähigkeit eines Systems, mit unerwarteten (Fehler-)Situationen umzugehen, ohne dass der Anwender davon etwas mitbekommt (bester Fall) oder mit einer definierten Herabsetzung der Servicequalität („graceful degradation of service“ im Englischen, schlechtester Fall).*

So weit, so gut. Aber wie designt man eine Anwendung denn konkret, dass sie „resilient“ ist? Das ist ein weites Gebiet, das den Rahmen dieses shortcuts bei Weitem sprengen würde. Deshalb möchte ich an dieser Stelle nur einige wenige Resilience-Prinzipien (**Abb. 1.3**) kurz vorstellen und beschreiben, welche Fragen man sich dafür beim Design stellen muss bzw. welche Aspekte man berücksichtigen sollte. Da es sich um Prinzipien handelt, mit denen man üblicherweise seine Überlegungen zu Resilient Software Design beginnt, hat auch dieser kleine Einblick durchaus einen praktischen Wert.


**Abbildung 1.3:** Elementare Grundprinzipien im Resilient Software Design

Codebeispiele habe ich – wenngleich eher ungewöhnlich für dieses Magazin – aus zwei Gründen weggelassen: Zum einen ist bei Resilience das Erarbeiten eines angemessenen Designs die eigentliche Herausforderung. Zum anderen ist es sehr schwer, Codebeispiele zu zeigen, die nicht nur einen kleinen Spezialfall abdecken. So müsste man für jedes der folgenden Prinzipien eigentlich Unmengen an Codebeispielen zeigen, was dann wieder den Rahmen des shortcuts sprengen würde. Also habe ich mich dazu entschlossen, dann lieber gar keine Codebeispiele zu machen und etwas genauer auf die Fragen einzugehen, die man sich beim Design stellen sollte.

## Isolation

Das erste Grundprinzip von Resilience ist *Isolation*. Auf diesem Prinzip bauen fast alle anderen Resilience-Muster auf. Die Idee hinter Isolation ist, dass ein System niemals als Ganzes kaputtgehen darf. Um das zu vermeiden, teilt man das System in möglichst unabhängige Einheiten auf und isoliert sie gegeneinander. Diese unabhängigen Einheiten werden *Bulkheads* (als Metapher aus dem Schiffsbau übernommen), *Failure Units* oder *Units of Mitigation* genannt. Die Einheiten isolieren sich upstream und downstream gegen Fehler anderer Einheiten, um kaskadierende, d. h. sich über mehrere Einheiten fortpflanzende Fehler zu vermeiden. Dafür stehen eine Menge Muster und Prinzipien zur Verfügung, z. B. lose Kopplung, Latenzüberwachung, Request-Limitierung oder die vollständige Validierung aller Aufrufparameter und Rückgabewerte. Einige der genannten Prinzipien werde ich im weiteren Verlauf dieses Kapitels noch vorstellen.

Es ist allerdings nicht hinreichend, nur die Failure Units gegen Fehler zu isolieren. Man benötigt auch eine Fallback-Strategie für den Fall, dass man einen Fehler bemerkt. Auch darauf werde ich im weiteren Verlauf dieses Kapitels noch eingehen.

Eine letzte Bemerkung zu Failure Units: Dies ist ein reines Designthema. Es gibt keine fertigen Bibliotheken oder Frameworks, noch irgendwelche vorgefertigten Best Practices, die man einfach verwenden könnte. Man muss sich explizit darüber Gedanken machen, wie man das Gesamtsystem in verschiedene Failure Units aufteilt und das dann entsprechend umsetzen. Hier geht es also, salopp ausgedrückt, um Hirnschmalz und nicht um Tools.

## Redundanz

*Redundanz* ist ein weiteres zentrales Resilience-Muster. Typischerweise sind es Failure Units, die redundant ausgelegt werden. Redundanz ist geeignet, um mit allen Arten von Fehlern umzugehen, nicht nur mit Absturzfehlern, an die die meisten Personen im Zusammenhang mit Redundanz nur denken. Entsprechend muss man sich zunächst einmal Gedanken über das Szenario machen, das man mit Redundanz adressieren möchte:

- Möchte ich *Failover* implementieren, d. h. beim Ausfall einer Einheit idealerweise komplett transparent auf eine andere Einheit umschalten?
- Oder möchte ich die Latenz gering halten, d. h. redundante Einheiten nutzen, um die Wahrscheinlichkeit einer zu langsamen Antwort zu reduzieren?
- Oder möchte ich Antwortfehler erkennen, d. h. die Antworten mehrerer redundanter (und im Extremfall auch auf unterschiedlicher Hardware laufender und unabhängig entwickelter) Einheiten auswerten, um die Wahrscheinlichkeit fehlerhafter Antworten zu reduzieren?
- Oder möchte ich Lastverteilung implementieren, d. h. die – für eine Einheit zu zahlreichen – Anfragen auf mehrere Einheiten verteilen?
- …

Abhängig vom gewählten Szenario sind dann weitere Aspekte zu berücksichtigen:

- Welche Routingstrategie passt zu meinem Szenario, wenn ich Load Balancer einsetze? Reicht einfaches Round Robin oder benötige ich eine komplexere Strategie, und wird diese von meinem Load Balancer unterstützt?
- Stellt mein System einen automatischen Masterwechsel sicher, wenn ich einen Master-Slave-Ansatz für Failover verwende und der Master ausfällt?
- Wenn ich die Latenz gering halten will, kann ich einen *Fan out & quickest one wins*-Ansatz verwenden, d. h., ich sende meine Anfrage an mehrere redundante Einheiten

und nutze die schnellste Antwort. Wie stelle ich bei so einem Ansatz sicher, dass die Einheiten nicht von alten, nicht mehr benötigten Anfragen so belastet werden, dass sie die neuen Anfragen nicht mehr zeitnah bearbeiten können?

- Welche Aspekte automatisiere ich und welche gestalte ich manuell? Eine allgemeine Resilience-Empfehlung besagt, dass man Fehlerbehebung möglichst vollständig automatisieren sollte, da im Falle eines Fehlers im Produktivsystem eine Stresssituation vorliegt und Menschen unter Stress typischerweise Fehler machen. Die Chance, dass während der Fehlerbehebung neue Fehler entstehen, ist viel zu hoch. Trotzdem muss es für einen Administrator möglich sein, in den Fehlerbehebungsprozess einzugreifen, wenn er es als notwendig erachtet.
- Wie stelle ich sicher, dass der Administrator stets den aktuellen Zustand des Systems sieht? Welche Einheit ist Master, welche sind Slave? Wie viele Einheiten laufen? Wo laufen sie? Usw.

Redundanz ist also ein mächtiges Muster mit vielfältigen Einsatzmöglichkeiten, das für eine saubere Implementierung aber auch eine Menge konzeptioneller Vorarbeit benötigt.

## Lose Kopplung

*Lose Kopplung* ist ein Grundprinzip zur Vermeidung kaskadierender Fehler und komplementiert so die zuvor beschriebenen Failure Units.

Ein gutes Muster zur Umsetzung loser Kopplung ist die Verwendung asynchroner Event- oder nachrichtenbasierter Kommunikation. Dadurch wird eine maximale Entkopplung der einzelnen Einheiten erreicht, und die Wahrscheinlichkeit sich fortpflanzender Latenzfehler wird minimiert.

Nun gibt es häufig Bedenken gegen den Einsatz asynchroner Kommunikation, weil es so viel komplizierter als synchrone Kommunikation sei. Das stimmt allerdings nur teilweise. Tatsächlich kann das menschliche Gehirn besser mit synchronen Call-Stack-orientierten Aufrufstrukturen umgehen als mit asynchronen Nachrichtennetzwerken. Außerdem benötigt man zusätzliche Visualisierungs- und Überwachungsmöglichkeiten für asynchrone Nachrichtennetzwerke, um den Überblick zur Laufzeit nicht zu verlieren.

Auf der anderen Seite muss man bei synchroner Kommunikation aber Time-out- und Latenzüberwachung ergänzen, um Antwortzeitfehler erkennen und behandeln zu können. Hierbei muss mit dem Fall umgegangen werden, dass man eine Antwort nicht rechtzeitig bekommen hat. Dieser Fall teilt sich in die Frage auf, was ich meinem Aufrufer zurückmelde, sowie die Frage, wie ich (bei verändernden Aufrufen) sicherstelle, dass die Information bei der aufgerufenen Einheit angekommen ist. Die Behandlung dieser Aspekte sorgt dafür, dass synchrone Kommunikation in letzter Instanz ähnlich kompliziert

ist wie asynchrone Kommunikation.

Setzt man dennoch synchrone Kommunikation ein (wofür es jenseits von Resilience durchaus gute Gründe geben kann), dann sollte man auf jeden Fall Muster wie *Timeout* und *Circuit Breaker* einsetzen (für eine gute Beschreibung dieser Muster siehe z. B. [4]), um eine gute Latenzüberwachung zu implementieren. Bibliotheken wie z. B. Hystrix [5] von Netflix können einem bei der Umsetzung gute Dienste leisten).

Ein weiteres hilfreiches Muster in diesem Kontext, das hier kurz erwähnt werden soll, ist *Idempotenz*. Ein (verändernder) Aufruf ist idempotent, wenn wiederholte Aufrufe keine zusätzlichen Seiteneffekte haben. Ein ganz einfaches Beispiel: Ein Aufruf „Addiere 1 auf Wert“ ist nicht idempotent, weil sich der Wert, auf den dieser Aufruf angewendet wird, als Seiteneffekt mit jedem Aufruf verändert. Ein Aufruf „Setze Wert auf 5“ hingegen ist idempotent. Egal, wie oft ich diesen Aufruf auf einen Wert anwende, das Ergebnis wird immer gleich sein, d. h., es entstehen keine zusätzlichen Seiteneffekte.

Im Falle synchroner Aufrufe mit Latenzüberwachung steht man vor dem Problem, dass man im Falle einer zu langsamen Antwort in der Regel nicht unterscheiden kann, ob der Aufruf gar nicht erst beim Empfänger angekommen ist oder ob der Empfänger nur zu langsam geantwortet hat. Es ist also unklar, ob die Anfrage verarbeitet wurde oder nicht. Verwendet man jetzt nicht idempotente Aufrufe, steht man bei zu langsamen Antworten vor dem Problem, dass man nicht oder nur mit sehr großem Aufwand entscheiden kann, ob man den Aufruf noch einmal senden darf oder nicht.

Verwendet man hingegen idempotente Aufrufe, stellt sich dieses Problem nicht. Man ruft seinen Empfänger einfach so oft auf (natürlich mit sinnvollen Pausen und Fehlerbehebungsmaßnahmen zwischen den Aufrufen, siehe auch *Fallback* im nächsten Abschnitt), bis man die Rückmeldung erhält, dass der Aufruf erfolgreich verarbeitet worden ist.

Idempotente Aufrufe ermöglichen also, von einer *Exactly Once*-Kommunikation auf eine *At Least Once*-Kommunikation zu wechseln, die wesentlich leichter umsetzbar ist und deutlich geringere Anforderungen an die Kommunikationsinfrastruktur stellt.

## Fallback

Das letzte Muster, das an dieser Stelle beschrieben werden soll, ist *Fallback*. Wie bereits bei Isolation erwähnt, ist es nicht hinreichend, Fehler zu erkennen. Man benötigt auch eine klare Strategie, was man machen will, wenn ein Fehler erkannt worden ist. Diese Strategie sollte sowohl die Bedürfnisse der Nutzer als auch die der Administratoren so gut wie möglich unterstützen.

Die Benutzer sollten im Idealfall überhaupt nicht bemerken, dass ein Fehler aufgetreten

ist. Die Administratoren sollten möglichst wenig Stress beim Auftreten eines Fehlers haben, denn Stress während der Fehlerbehebung führt – wie bereits unter Redundanz beschrieben – in der Regel zu neuen Fehlern. Aus diesem Grund sollte die Fehlerbehandlung möglichst vollständig automatisiert erfolgen – einfach per Nagios eine E-Mail an den Administrator zu schicken, ist nicht (mehr) das Mittel der Wahl.

Die Kernfrage, die man sich im Zusammenhang mit Fallbacks stellen muss, ist: Wie soll das System reagieren, wenn ein Fehler entdeckt wird?

Nehmen wir die zuvor beschriebene einfache Webanwendung: Ein Anwender stellt eine Anfrage an die Anwendung. Dies führt zu einer Anfrage an die Datenbank. Die Anfrage an die Datenbank läuft nach – sagen wir – einer Sekunde in einen Time-out, der erkannt wird. Eine Anfragewiederholung (als Maßnahme zum Umgang mit Auslassungsfehlern) läuft ebenfalls in einen Time-out. Jetzt stellt sich die Frage, wie die Anwendung in dieser Situation reagieren soll:

- Soll dem Anwender eine Fehlerseite mit dem Hinweis gezeigt werden, dass gerade ein Problem besteht und er es später noch einmal versuchen soll?
- Oder halten wir möglicherweise die Nutzerdaten in Caches und können Leseanfragen weiterhin aus den Caches bedienen und müssen nur Schreibanfragen mit einer „Versuchen Sie es später noch einmal“-Seite quittieren?
- Oder schreiben wir in dem Fall Schreibanfragen in eine Queue, die von einem asynchronen Job abgearbeitet wird, sobald die Datenbank wieder verfügbar ist und antworten dem Anwender, dass sein Auftrag angenommen wurde und später verarbeitet wird (und sehen ggf. ergänzend dazu für den Anwender eine Möglichkeit vor, den Verarbeitungsstatus seiner Anfrage(n) einzusehen)?
- Oder … ?

Das sind alles Varianten einer *Graceful Degradation Of Service*, der kontrollierten Herabsetzung der Servicequalität. Jede dieser Optionen kann je nach Anwendungsfall valide sein. Nur eine Variante ist nicht valide, nämlich dass der Anwender die Sanduhr sieht, bis sein Browser nach fünf Minuten einen Netzwerk-Time-out meldet.

Wichtig ist hierbei, dass das keine Entscheidungen sind, die ein Entwickler „on the fly“ treffen kann, während er gerade die entsprechende Fehlerbehandlung implementiert, also z. B. im Fall von Java oder C# den Code für den zugehörigen Ausnahmebehandlungsblock schreibt.

Das sind letztlich Fachanforderungen, nämlich die Frage, welche Reaktion der Anwender im Falle eines technischen Fehlers zu Gesicht bekommen soll. Diese Fragen müssen wie die User Stories und die Basisarchitektur geklärt sein, **bevor** ein Entwickler den

zugehörigen Code schreibt. Das sind Fragen, die beispielsweise im agilen Umfeld ein Product Owner beantworten muss.

Hat man die grundsätzliche Fallback-Strategie festgelegt, kann man die Strategie bei der Umsetzung noch durch Resilience-Muster wie *Escalation Strategy* oder *Error Handler* unterstützen, die das Implementieren einer automatischen Fehlerbehebung unterstützen (für eine detaillierte Beschreibung dieser Muster siehe z. B. [1]).

## Zusammenfassung

Das war ein kurzer Ausflug in das Design von Resilience anhand einiger elementarer Prinzipien. Natürlich gäbe es noch sehr viel mehr zu schreiben, da Resilience eine reichhaltige und umfangreiche Mustersprache bietet, aber das würde den Rahmen dieses shortcuts bei Weitem sprengen.

Was bleibt unterm Strich? Bei Resilient Software Design geht es darum, die Verfügbarkeit von Systemen und Systemlandschaften zu maximieren. Im Gegensatz zu traditionellen Stabilitätsansätzen versucht Resilience nicht, die Eintrittswahrscheinlichkeit von Fehlern zu reduzieren, sondern nimmt Fehler als unvermeidbar und unvorhersehbar hin und versucht stattdessen, möglichst schnell auf auftretende Fehler zu reagieren.

Dieser Ansatz passt sehr gut zu den heutigen komplexen, verteilten und hochgradig vernetzten Systemlandschaften, in denen es unmöglich ist, die vielfältigen möglichen Fehlerquellen zu antizipieren und durch vorbeugende Maßnahmen auszuschließen – und der immer höher werdende Grad an Verteilung und Vernetzung, sei es wegen Cloud, Mobile oder Internet of Things sowie die immer schwerer vorhersagbaren Lastmuster, ebenfalls wegen Mobile und Internet of Things, aber auch wegen Social Media, werden dafür sorgen, dass Resilient Software Design in Zukunft noch viel wichtiger wird, als es jetzt bereits ist.

## Links & Literatur

[1] Hanmer, Robert S.: „Patterns for Fault Tolerant Software“, Wiley, 2007

[2] Tanenbaum, Andrew; van Steen Marten: „Distributed Systems – Principles and Paradigms“, Prentice Hall, 2nd Edition, 2006

[3] Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, ISO/IEC 25010:2011(en): https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

[4] Nygard, Michael T.: „Release It!“, Pragmatic Bookshelf, 2007

[5] Hystrix: https://github.com/Netflix/Hystrix

# 2 Netflix: Resilience konsequent zu Ende gedacht

Netflix gilt als Vorreiter, wenn es darum geht, Fehler nicht nur zu vermeiden, sondern sie als normalen Teil des (System)lebens zu akzeptieren und effektiv zu überstehen. Mit einem Konzert von Tools und Bibliotheken rund um Hystrix, Asgard, Eureka und die Cloud-Infrastruktur von Amazon isoliert Netflix seine Nutzer von Fehlern und bleibt als System stabil. Dabei sind sich die Herren und Damen so sicher, dass sie selbst Hand anlegen und eine Vielzahl unterschiedlicher Fehler im Produktivsystem verursachen – täglich –, um es zu beweisen und im Ernstfall zu können. Dazu gehört Courage und Selbstbewusstsein. Haben sie die?

Fehler sind unvermeidbar. Sie testen Ihre Software auf unterschiedlichen Ebenen, unterziehen sie Reviews, Sie lassen Bug Finder und Metriken auf Ihre Applikationen los und trotzdem: Völlige Bugfreiheit ist illusorisch. Manche Firmen entwickeln mit unterschiedlichen Teams zwei Mal das gleiche System, um dieses Problem zu umgehen, setzen auf unterschiedliche Frameworks und Bibliotheken. Selbst dann bleibt noch die Hardwareseite der Medaille, um die sie sich kümmern müssen. Festplatten, Platinen, Sensoren etc. – alles Fehlerquellen. Müssen Sie Fremdsysteme anbinden oder haben Sie Verteilungsgrenzen in Ihrem System? Glückwunsch! Das Netzwerk hatten wir bisher noch gar nicht auf der Rechnung. Es ist jedoch nicht alleine die schiere Menge an Fehlerquellen, die problematisch ist, sondern auch deren zufällige und nicht vorhersagbare Verteilung.

Es gibt mehrere Strategien, wie Sie trotzdem zuverlässige und verfügbare Systeme bauen können. Eine davon ist besonders effektiv und bietet sozusagen ein zweites Fangnetz: Resilient Design. Wenn Fehler nicht vermieden werden können (oder deren Vermeidung sehr teuer wäre), sollte das System gut damit umgehen können und nicht als Ganzes gefährdet sein. *Resilience* ist also eine Eigenschaft, die es Systemen erlaubt, Defekte und Fehler zu überstehen, sie zu isolieren, ihre Auswirkungen gering zu halten und sie idealerweise zu korrigieren. Für manche Systeme ist Resilience der einzige Weg zur Ausfallssicherheit, für andere einfach die billigere Alternative zu teuren Simulationen, Analysen und Testumgebungen. Netflix ist ein Pionier des Resilient Designs und setzt neue Maßstäbe, was die Entwicklung für Fehlertoleranz angeht. Sehen wir, warum das so ist, was Netflix genau macht und wie Sie davon profitieren können.

## Warum Netflix?

Netflix ist der größte Online-Streaming-Anbieter der Welt. Über zwei Milliarden Stunden Filme und Serien sind für Mitglieder verfügbar (nicht in jedem Land, aber dennoch). Warum wurde Netflix so innovativ, was die Verfügbarkeit angeht? Man bekommt eine erste Idee, wenn man einen Blick auf die Webseite von Netflix wirft: „Netflix-Mitglieder können Serien und Filme schauen – so viele sie wollen, jederzeit, überall, auf fast jedem internetverbundenen Bildschirm“. Eine zentrale Ansage, ein Versprechen und eine Architekturherausforderung zugleich.

Um zu verstehen, wie groß die Herausforderung eigentlich ist, zeigt **Abbildung 2.1** eine topologische Sicht auf Netflix. Die grünen Bereiche zeigen Services von Netflix. Insgesamt sind es über sechshundert verschiedene Services, die jeweils drei bis mehrere hundert Mal installiert sind. Der Übersicht halber zeigt **Abbildung 2.1** diese Redundanz nicht. Die blauen Linien zeigen Abhängigkeiten bzw. Aufrufwege. Greift man auf die Webseite zu, trifft man etwa zwanzig bis dreißig Prozesse im Front Tier, dahinter wird es komplizierter. Insgesamt arbeiten etwa fünfzig bis sechzig Teams an diesem System.

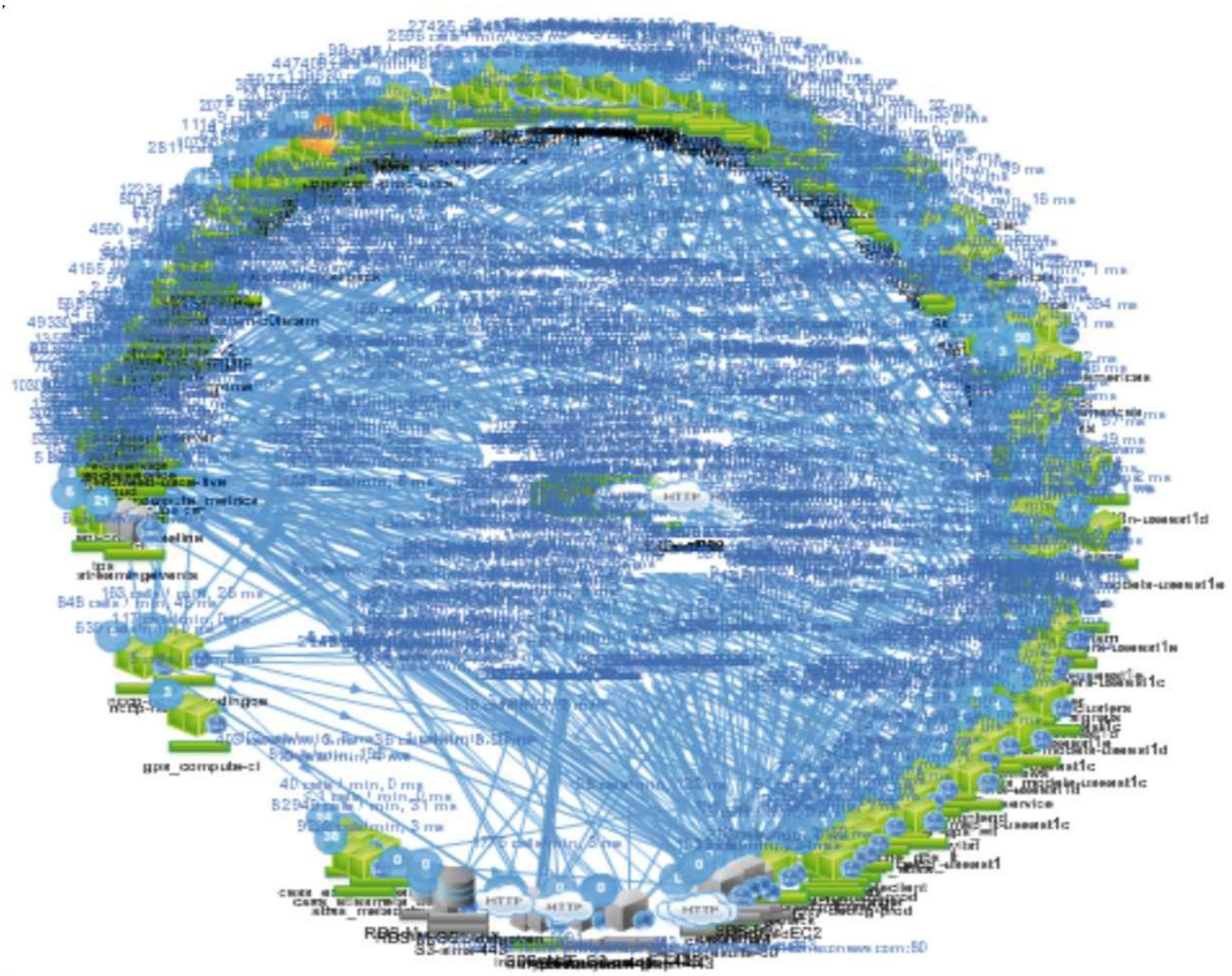

**Abbildung 2.1:** Netflix: Topologiesicht; Screenshot aus AppDynamics [1]

Es ist eine gewisse Hürde, ein solches System in einer realistischen Testumgebung

nachzubauen, Daten(-verteilungen) und Benutzerverhalten realistisch zu simulieren und dieses System analytisch zu betrachten. Netflix verzichtet trotz allem nicht gänzlich darauf, betrachtet diese Techniken jedoch als Ergänzung. Resilience ist aufgrund der Komplexität ein Erfordernis und in vielen Fällen billiger als Fehlervermeidungsstrategien vor der Auslieferung. Hinzu kommt noch der Zeitaspekt: Netflix möchte innovativ bleiben, neue Features schnell ausliefern und damit den Vorsprung gegenüber der Konkurrenz zumindest halten. Manager sagen dazu gerne Time to Market – bei Netflix heißt es, der schnellste Weg, Code auszuliefern, sei es, gelegentlich Bugs auszuliefern.

## Die zentralen Herausforderungen

Verteilte Systeme wie Netflix sind von Fehlerquellen durchsetzt. Festplattenfehler, Stromausfälle, Backup-Generatorausfälle, Netzwerkfehler, Softwarebugs, menschliches Versagen und einiges mehr machen hohe Verfügbarkeit schwierig.

Recht gut verstanden und einfach zu reparieren sind „saubere“ Ausfälle von Instanzen. Gehen etwa durch einen Stromausfall oder einen anderen Fehler einzelne Instanzen verloren, ist der lokale Status zwar ebenfalls verloren, der Fehler ist allerdings schnell entdeckt, und das System kann entsprechend reagieren, etwa, indem der Traffic umgeleitet und eine Ersatzinstanz bereitgestellt wird. Sind Services zustandslos oder wird der Zustand über Caches oder eine Datenbank „geshared“, wird das Problem noch kleiner.

Problematischer sind andere Ausfälle. Wird etwa fehlerhafter Code ausgeliefert, ist ein Service mit all seinen Instanzen betroffen. Die fehlerhafte Funktionalität muss schnell aus dem Verkehr gezogen werden, steht dann dem Benutzer aber nicht zur Verfügung. Schnelle Erkennung von solchen Fehlern und noch schnelleres Rollback sind wichtig, aber schwierig.

Netzwerkfehler isolieren Instanzen vom Rest des Systems, sie sind aber oft weiterhin ansprechbar – zumindest bei partitionstoleranten Systemen, wie es hoch skalierbare Webanwendungen meist sind. Dadurch kann der Status inkonsistent werden, und man erhält komplexere Symptome, die eine Wiederherstellung erschweren.

Sich fortpflanzende Fehler stellen ein weiteres, größeres Problem dar. AWS-(Amazon-Web-Services-)Fehler können beispielsweise dazu führen, dass keine neuen Instanzen mehr gestartet werden können. Die Threadpools können nicht mehr aus dem Vollen schöpfen, das Load Balancing wird beeinflusst und reagiert vielleicht fehlerhaft. Kommt dann noch etwas menschliches Versagen ins Spiel, kann auch eine ganze Zone oder gar Region Schaden nehmen. Sich fortpflanzende Fehler zeigen oft verwirrende Symptome und so genannte „byzantinische“ Seiteneffekte.

Bleiben noch Latenzprobleme, die ebenfalls zur herausfordernden Kategorie gehören.

Nachdem Services bei diesen Problemen noch antworten, ist die Erkennung und Behandlung schwieriger. Dabei kann eine einzelne latent antwortende Funktionalität das gesamte System gefährden. Bei Netflix ist etwa das API stark bedroht. Es spricht alle anderen (Mid-Tier-)Services an. Dauert eine Operation normalerweise 20 Millisekunden und hat plötzliche Spikes im Sekundenbereich, wird diese Funktionalität schnell alle Ressourcen, Queues und Threads in der API-Schicht binden. Bei mehr als 50 Requests pro Sekunde können alle Request-Threads innerhalb weniger Sekunden blockieren, und das System kollabiert.

## Strategien und Techniken

Diesen Problemen setzt Netflix ein ganzes Set an Strategien und Techniken entgegen. Die wichtigsten davon sind die folgenden, die ich im weiteren Verlauf dieses Kapitels noch genauer besprechen werde:

1. **Redundanz**: Netflix' Regel: „Everything is built for three".
2. **Logging und Monitoring**: Die Erkennung und Analyse von aufgetretenen oder bevorstehenden Fehlern.
3. **Schnelles Fallback/Rollback/Failover**: Erkannte Fehler müssen schnell zu Reaktionen führen.
4. **Fehlerisolierung**: Einzelne Fehler oder Latenzen sollen nicht „durchschlagen".
5. **Empirische Überprüfung der Resilience**: Zielgerichtete Einsteuerung von Fehlern in die Produktivumgebung als Test und Übung.

Netflix hat neben den Infrastrukturelementen der Amazon-Cloud, die bereits einige dieser Punkte behandeln, einige Werkzeuge und Frameworks erarbeitet, die mittlerweile zum Großteil Open Source zur Verfügung gestellt wurden. Die berühmtesten Projekte für die genannten Strategien sind Hystrix, Asgard und die Simian Army.

## Redundanz

Netflix verteilt alle Applikationen und Infrastrukturkomponenten über mehrere „Zonen", um bei Ausfällen „Redundanz für alles" zu haben. Netflix unterteilt seine Datencenter in drei große AWS-Regionen: EU, US-East und US-West. Theoretisch kann der Ausfall einer ganzen Region verkraftet werden, wenn die anderen beiden Regionen entsprechend elastisch hochskaliert werden. Innerhalb der Regionen sind so genannte „Availability Zones" die nächstkleinere Einheit. Dabei handelt es sich um Datencenter, in denen unabhängige Application-Cluster wohnen („Auto Scaling Groups", kurz ASGs). Die ASGs sind wiederum aus bis zu 100 Instanzen pro Zone aufgebaut, die über eine Service Registry und Software-Load-Balancing miteinander kommunizieren. Jeder Request eines

Nutzers hat eine Präferenz für eine Zone (innerhalb einer Region), der nächste Request ist aber völlig ungebunden und kann die Zone theoretisch wechseln.

Die einzelnen Instanzen sind weitestgehend zustandslos, der eventuell vorhandene Zustand wird jedoch in einer Netflix-Erweiterung von Memcached (EVCache) gehalten und über mehrere Zonen verteilt, natürlich über mindestens drei. Auch die Daten, die Netflix hauptsächlich in Cassandra-Clustern ablegt, sind redundant über mindestens drei Zonen gestreut – für global relevante Daten auch über Zonen unterschiedlicher Regionen. Über Cassandra-Mechanismen wie Local Write Quorums und Hinted Handoffs ist die sichere Ablage von Daten gelöst. Nächtliche Vergleichs- und Reparaturarbeiten sorgen für Konsistenz.

Der Ausfall einer einzelnen Instanz wird nun bereits durch die ASGs, also die automatisch skalierenden Application-Cluster, abgefangen und ausgeglichen. Als fehlerhaft erkannte Instanzen werden von der Service Discovery ausgeschlossen. Bis dahin überspringen Software-Load-Balancer die fehlerhafte Instanz. Auto Scaling sorgt nach dem Trafficstop für neue Instanzen, um das System stabil zu halten.

Der Ausfall ganzer Zonen ist ähnlich verkraftbar, weil alle Daten und Funktionen über mindestens drei Zonen gestreut sind. Komplizierter wird es trotzdem. Kurzfristig ist eine Zone immer mit etwa ein Drittel Overhead ausgelegt, kann also einen Zonenausfall gemeinsam mit einer anderen Zone abfangen. Muss mehr Traffic abgefangen werden, setzt ein Throttling-Mechanismus ein, der den Zugriff auf die Zone begrenzt und sie so schützt. Nach dem Auto Scale der enthaltenen Application-Cluster wird wieder der gesamte Traffic angenommen. Dieses Vorgehen verhindert einen Totalausfall und sorgt dafür, dass einzelne Services lediglich sehr kurz für einige wenige Benutzer nicht verfügbar sind. Daneben muss das Routing sauber sein, um nicht zu viel Traffic zwischen den Zonen zu verursachen und in die fehlerhafte Zone tatsächlich keinen Traffic mehr zu leiten. Netflix beobachtet solche Ausfälle in Regionen und reagiert proaktiv. Sind Zonenausfälle in US-East zu beklagen, wird vorsichtshalber in US-West hochskaliert, um im Ernstfall schnell reagieren zu können.

## Logging und Monitoring

Für die eben besprochene proaktive Reaktion müssen Fehler erst einmal bekannt sein. Für die spätere Analyse des Fehlergrunds und die daraus abgeleiteten Maßnahmen zur Verbesserung müssen die Zustände und Aktionen nachvollziehbar und analysierbar sein. Auch Resilience-Werkzeuge wie die weiter unten besprochene Simian Army müssen über den Systemzustand Bescheid wissen. Absichtlich ins System eingeschleuste Fehler sollten nicht sowieso schon kränkelnde Systemteile destabilisieren. Im Resilient Design sind Logging und Monitoring also zentral – Netflix bezeichnet sich manchmal als Logging-

Service, der zufällig auch Filme abspielen kann.

Die drei wichtigsten Tools, die Netflix hierfür einsetzt, sind Atlas, Chronos und Hystrix. Altlas ist ein Cloud-weites Monitoringframework mit Millionen von Metriken rund um Performance, Fehler- und Time-out-Raten. Es operiert unterhalb der Hystrix-Ebene auf Services, Instanzen, der JVM und Tomcat. Aus Atlas bedienen sich weitere Monitoringtools wie Mogul und Vector.

Chronos ist ein Netflix-Projekt, das noch nicht Open Source bereitgestellt wurde. Es zeichnet alle Änderungen an der Systemumgebung oder -konfiguration auf und soll formale Change-Prozesse so weit wie möglich obsolet machen. Chronos nimmt über eine REST-Schnittstelle Events an und erlaubt es, Menschen und Maschinen nach den Aktivitäten der letzten Stunde, nach speziellen Deployments oder sonstigen (Zustands-)Änderungen zu fragen. Über die Integration zu anderen Netflix-Werkzeugen werden die meisten Änderungen an der Umgebung automatisch an Chronos reportet. Prinzipiell soll jedes Event, das den Zustand des Systems ändert, aufgezeichnet werden. Dazu zählen z. B. Deployments, Änderungen in der Runtime Configuration, AB-Tests, Securityevents und andere automatisierte Aktionen.

Hystrix ist eine Bibliothek für Latenz- und Fehlertoleranz. Der „Fault Tolerance Layer“ von Netflix besteht hauptsächlich aus Hystrix – es ist *das* Tool für Resilience. Hystrix isoliert Servicezugriffe voneinander, stoppt sich fortpflanzende Fehler, sorgt auch bei Latenz für schnelle Fehler und koordiniert Fallbacks (mehr dazu in den folgenden beiden Abschnitten 3 und 4). Hierfür wrappt Hystrix die Aufrufe fremder Services und verfügt über alle wichtigen Informationen für ein feingranulares Monitoring. Die Daten werden wie beim grundlegenderen Atlas auf einem Dashboard gesammelt und mit geringer Latenz live aktualisiert. **Abbildung 2.2** zeigt ein annotiertes Beispiel-Dashboard für eine ASG.

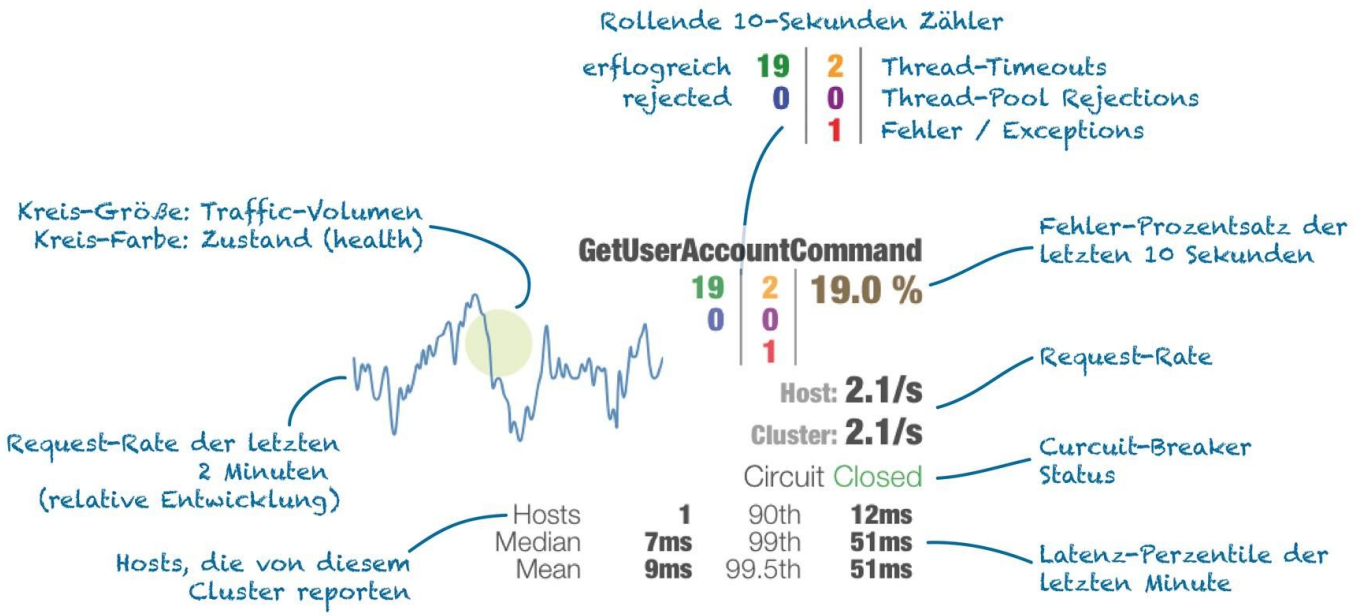

**Abbildung 2.2:** Beispiel eines Hystrix Dashboards [2]

Mit Turbine, einem weiteren Netflix-Projekt, lassen sich die Daten mehrerer ASGs aggregieren. So können mehrere hundert Server live getrackt werden. Die Anzeigelatenz steigt dabei von Sekundenbruchteilen für einzelne Server auf etwa 1–2 Sekunden. Netflix weiß also recht schnell, wenn es brennt und wo es brennt.

## Schnelles Fallback/Rollback/Failover

Ebenso wichtig wie die Erkennung von Fehlern und ungewolltem Verhalten ist die Reaktion darauf. Bei Ausfällen muss ein automatischer Failover erfolgen, Zonen und Regionen müssen sich gegenseitig auffangen können. Hier hilft die Redundanz und die Infrastruktur von Amazon (siehe Punkt 1 – Redundanz). Bei Ausfällen ist der Failover schnell, bei Problemen mit der Latenz hilft wiederum Hystrix: Netzwerkzugriffe werden mit aggressiven Time-outs versehen, und das Circuit Breaker Pattern [3] sorgt dafür, dass langsam oder fehlerhaft reagierende Services beim nächsten Mal gar nicht erst aufgerufen werden, sondern über den „Short Circuit“ sofort in einen Fallback münden. Das spart Zeit.

Um auch nach einem Failover sofort wieder schnelle Antwortzeiten zu garantieren, ist ein aufgewärmter Ersatzcache nötig. Den hält Netflix mit EVCache vor – einer Erweiterung von memcached und spymemcached, die mit dem Netflix-Ökosystem und speziell mit der Service-Registry von Netflix (Eureka) zusammenarbeitet. EVCache steht für Ephemeral Volatile Cache, beschreibt also einen kurzlebigen Cache, dessen Daten jederzeit verschwinden können. Es handelt sich um einen verteilten Key-Value-Store, der über mehrere AWS-Availability-Zonen operieren kann. Beim Start-up registrieren sich EVCache-Instanzen bei Eureka. Clients verbinden sich dann per Namen mit den registrierten Cacheinstanzen. Im Falle eines Multi-Zone-Deployments erfolgen Lesezugriffe immer in einer Zone, Schreib- oder Löschzugriffe immer in allen Zonen. Kommt es zum Failover, findet der Client in einer „fremden“ Zone einen warmen Cache vor. Auch wenn der zonenübergreifende Zugriff etwas Latenz verursacht, ist das meist um Größenordnungen schneller als ein Zugriff auf Cassandra, S3 oder SimpleDB selbst. EVCache unterstützt also tatkräftig beim schnellen Failover bzw. der Verschleierung des Fehlers gegenüber dem Benutzer.

Das Wegbrechen einzelner Instanzen oder Zonen ist mit den genannten Techniken schnell auszugleichen. Problematisch wird es jedoch, wenn neue oder veränderte Funktionalität in Produktion gebracht wird und dort für Ausfälle oder fehlerhafte Antworten sorgt. Die Funktionalität muss schnell von allen Instanzen verschwinden und durch eine nicht schadhafte Version ersetzt werden, am schnellsten natürlich in einem Rollback zur Vorversion. Die Deployment-Kette um Tools wie Asgard hilft Netflix hier (**Abb. 2.3**) und orientiert sich am Ansatz der Blue-Green Deployments [4].

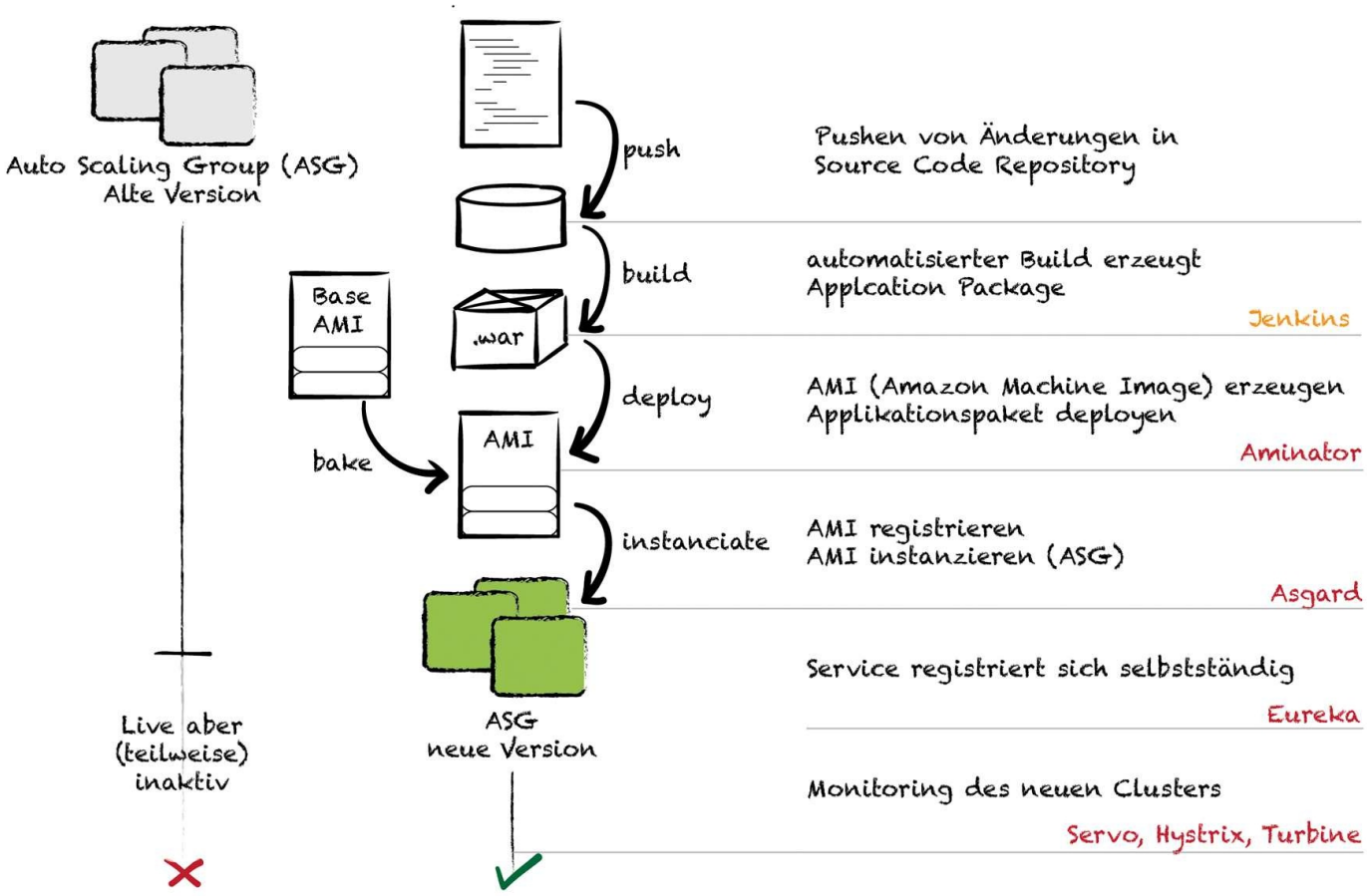


**Abbildung 2.3:** Build und Deployment bei Netflix

Der Build- und Deploymentprozess von Netflix wird in der Mitte gezeigt, seine Beschreibung rechts. Neben Standardwerkzeugen wie Jenkins setzt Netflix auf eine eigene Image-Bakery (Aminator). Dort werden die Applikationen auf Maschinen-Images für die Amazon-Cloud installiert, bevor sie deployt werden. Das Deployment selbst wird durch Asgard gesteuert und kann so in einem einzelnen Schritt erfolgen. Auf der linken Seite von **Abbildung 2.3** sehen Sie die Lebenslinie der alten Version der Funktionalität. Sie lebt nicht nur bis zum Deployment der neuen Version, sondern ist auch danach noch aktiv. In einer Cloud-Umgebung können Sie unbegrenzt Maschinen erzeugen. Netflix instanziiert die neue Serviceversion also *zusätzlich* und leitet lediglich den Traffic teilweise um. Oft werden nur einzelne Knoten mit der neuen Version live geschaltet – so genannte „Canaries“. Mit Werkzeugen wie Hystrix oder Servo überwacht Netflix nun beide produktive Versionen und protokolliert unterschiedliches Verhalten oder Fehler. Bei Fehlern oder stark verschlechterten Performanz- und Stabilitätseigenschaften wird der neue Service automatisch aus dem Verkehr gezogen – und zwar genau so wie bei einem Failover. Der Traffic wird einfach zu den alten Instanzen zurück geleitet, die Instanz(en) der neuen Serviceversion aus der Service-Discovery ausgeschlossen. Dieses Vorgehen ist zwar ressourcenintensiver, aber sehr effizient im Sinne der Resilience. Nutzer sind sehr schnell von fehlerhaften Services abgeschnitten und können das System wie gewohnt benutzen.

## Fehlerisolierung

Abhängigkeiten zwischen Services machen Systeme kompliziert. Wie man in **Abbildung 2.1** sehen kann, ist das bei Netflix auch nicht anders. Im Sinne der Resilience sollten Fehler in einem Service nicht zu anderen Services durchschlagen bzw. sollten sich Abhängigkeiten zu unterschiedlichen Services nicht gegenseitig beeinflussen. Es ist deshalb wichtig, Aufrufe unterschiedlicher Services in unterschiedliche Threads auszulagern. Aufgrund der schwer durchschaubaren Komplexität beim Schachteln von Threads und Responses hat Netflix RxJava entwickelt: eine Java-Portierung von ReactiveX aus dem Microsoft-Umfeld [5]. Grob gesprochen handelt es sich dabei um ein asynchrones Kommunikationsmodell, in dem über Events kommuniziert wird. Beobachtete Event- und Datenströme können dabei explizit geschlossen werden, Eventempfänger wissen also, wann sich weiteres Warten auf Events nicht mehr lohnt. RxJava erlaubt Netflix eine sehr lose, aber effiziente Kopplung von Services.

Die bevorzugt über RxJava abgebildeten Serviceabhängigkeiten werden durch Hystrix gewrappt. Hystrix zeichnet nicht nur Informationen für das Monitoring auf und sorgt für (schnelle) Fallbacks im Fehlerfall, es sorgt vor allem auch für die Isolierung von Abhängigkeiten. Aufrufe werden in Threads oder tryable Semaphores ausgelagert, und es werden so genannte „Bulkheads“ eingezogen. Um zu verhindern, dass fehlerhaft oder langsam antwortende Services binnen Sekunden alle Request-Threads auf Clientseite binden, werden Requests einem eigenen Threadpool je Serviceabhängigkeit zugeordnet. Ist dieser gesättigt, wird eine Exception geworfen und ein Fallback ausgelöst, andere Threadpools (bzw. Serviceaufrufe) sind davon nicht betroffen. Die genaue Funktionsweise inkl. Flow-Chart finden Sie unter [6].

## Empirische Überprüfung der Resilience

Die bisher besprochenen Techniken helfen beim Resilient Design. Wie wissen Sie aber, dass Sie mit Ihren Bemühungen erfolgreich sind? Wie im Abschnitt „Warum Netflix?“ beschrieben, ist Testen für Netflix impraktikabel, was ROI und Time to Market neuer Features betrifft. Stattdessen hat man sich entschieden, Resilience am echten System zu beweisen – dort, wo es darauf ankommt. Dazu gehört eine ordentliche Portion Selbstbewusstsein, die sich Netflix Stück für Stück erarbeitet hat. Mittlerweile gibt es eine ganze Abteilung, die sich mit dem „Chaos-Engineering“ beschäftigt, also dem kontrollierten Einschleusen von Fehlern und Latenzen. Zyniker würden Netflix vielleicht raten, einfach auf eine unzuverlässigere Infrastruktur umzuziehen, um sich diese Chaosabteilung zu sparen. Das, was Netflix salopp als „Chaos“ bezeichnet, hat jedoch durchaus Ordnung und Methode.

Die fehlerverursachenden Applikationen sind allesamt „Monkeys“ und gehören der

Simian Army an – dem wohl berühmtesten Ausschnitt des Netflix-Ökosystems. Tabelle 2.1 zeigt einen Überblick über einige bei Netflix eingesetzten Monkeys.

| Monkey | Zweck |
|---|---|
| Chaos Monkey | Terminiert produktive Instanzen nach dem Zufallsprinzip |
| Latency Monkey | Macht Aufrufe auf Client- oder Serverseite langsamer bzw. fehlerhaft |
| Chaos Gorilla | Simuliert den Ausfall ganzer AWS-Availability-Zonen (Datencenter) |
| Chaos Kong | Simuliert den Ausfall einer der drei AWS-Regionen |
| Janitor Monkey | Markiert ungebrauchte Ressourcen, Instanzen etc. und räumt sie ab |
| Conformity Monkey | Prüft Services auf die Einhaltung von Best Practices und Standards. Zieht nicht konforme Services aus dem Verkehr |
| Doctor Monkey | Führt Health-Checks durch und analysiert beobachtbare Serviceeigenschaften. Terminiert „kranke“ Knoten nach einer Analysefrist |

**Tabelle 2.1:** Die wichtigsten Affen der Simian Army

Die unteren drei Monkeys aus Tabelle 2.1 informieren die Serviceeigentümer per Mail über ihre Erkenntnisse. Teams werden so für nicht konforme oder überlastete Services sowie unsauberes Ressourcenmanagement in die Verantwortung gezogen und können aus diesen Problemen lernen. Chaos Monkey, Gorilla, Kong und Latency Monkey intervenieren unabhängig von Serviceeigenschaften. Sie sind das unberechenbare Element, um für den Ernstfall zu üben.

Chaos Monkey war der erste Vertreter seiner Gattung und simuliert den wohl häufigsten Fehler in der typischen Cloud-Umgebung: den Ausfall einer virtuellen Instanz. Über Asgard und Edda findet Chaos Monkey die momentan laufenden Serviceinstanzen. Jede Stunde wählt er einige davon zufällig aus und terminiert sie über AWS-APIs. In der Chaos-Monkey-Konfiguration können Teams die Wahrscheinlichkeiten einstellen, mit der ein bestimmter Service getroffen wird – auch auf 0 Prozent, falls nötig (Opt-out).

Chaos Gorilla funktioniert ähnlich. Um Zonenfehler zu simulieren, werden zwei Modes angeboten. Neben dem totalen Zonenausfall können auch Netzwerkpartitionen entstehen, die ein Grüppchen von Instanzen weiterhin funktionsfähig, aber vom Restsystem isoliert belassen. Der Einsatz von Chaos Gorilla ist eine ungleich höhere Herausforderung und sorgt für massive Probleme bei der Lastverteilung und Recovery. Die Ausführung ist deshalb nicht automatisiert, sondern erfolgt manuell.

Chaos Kong kümmert sich um den Ausfall einer ganzen AWS-Region. Ein entsprechender Ausfall ist allein schon eine Herausforderung in der Simulation, und selbst Netflix lässt dieses Biest nicht allzu oft aus dem Käfig. Alle paar Monate ist es aber so weit. Generell

werden die „Ausfalls“-Monkeys in der Reihenfolge ausgeführt, wie sie auch hier beschrieben sind. Zur Einführung dieser Monkeys konnten sich Teams zu Beginn freiwillig für das Chaos melden (Opt-in). Später wurde dann ein Blacklisting erstellt, wie beim Chaos Monkey beschrieben.

Der Latency Monkey stellt schlussendlich sicher, dass Netflix Services mit Problemen umgehen können, die keinen klaren Ausfall darstellen. Dafür wird REST-Calls eine zufällige Latenz zugeschlagen, oder es werden Fehler zurückgegeben (Status 500 beispielsweise). Werden sehr lange Latenzzeiten zugeschlagen, kann auch der Ausfall eines Service simuliert werden, ohne die entsprechenden Instanzen tatsächlich terminieren zu müssen. Der Service kann so auch nur für bestimmte Clients nicht verfügbar gemacht werden, während er für andere Clients weiterhin normal ansprechbar bleibt – schön vor allem beim Test der Robustheit von neu deployten Services. Durch den Einsatz dieses Monkeys hat Netflix vor allem in den Bereichen Start-up Resilience, Bootstraping, Cache-Warm-up und der Koordinierung von Abhängigkeiten gelernt.

Insgesamt hilft die Simian Army Netflix dabei, die Probleme rund um Resilience zu verstehen und Teams schrittweise zu einem entsprechenden Design zu führen. Die Monkeys werden prinzipiell zwischen 9:00 und 15:00 Uhr (Serviceeignerzeit) ausgeführt und rigorosem Monitoring unterzogen. So werden Fehler zu Zeiten provoziert, in denen viele Entwickler reagieren und gegensteuern können, die Lernerfahrung wird ins System zurückgetragen, und man ist auf echte Ausfälle zu eventuell ungünstigeren Zeiten besser vorbereitet. Dass diese Ausfälle tatsächlich passieren, zeigt die Geschichte. Netflix musste am 21. April 2011 mit großen Ausfällen in der AWS-Region US-East umgehen, zu Weihnachten 2012 fiel das Elastic Load Balancing von Amazon aus, und letztes Jahr (2014) mussten wegen Wartungsarbeiten 218 Cassandra-Knoten heruntergefahren werden.

## Sie wollen das auch?

Netflix baut mit der eigenen Software auf der Cloud-Infrastruktur von Amazon auf. Viele der entwickelten Frameworkkomponenten entlässt Netflix als Open-Source-Software (OSS) in die Freiheit [7]. So können Projekte auf viele Errungenschaften zurückgreifen (**Abb. 2.4**).

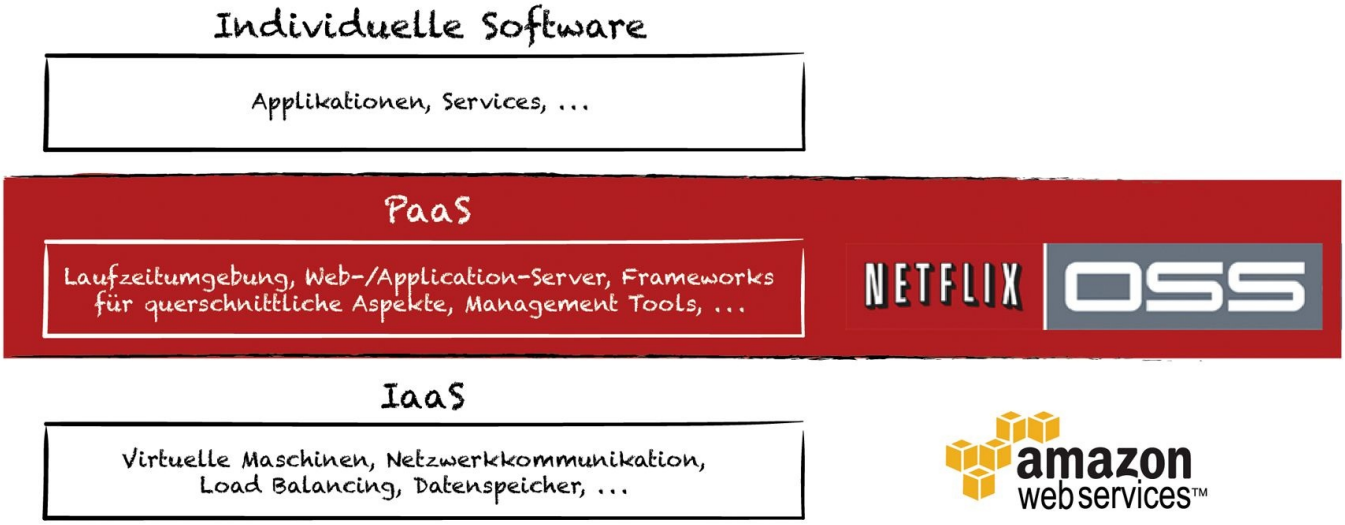


**Abbildung 2.4:** Netflix OSS als PaaS

Die in diesem Kapitel erwähnten Projekte Chaos Monkey, Janitor Monkey, Conformity Monkey, Hystrix, Turbine, Asgard, EVCache, Edda, Eureka, Aminator und RxJava sind alle bereits verfügbar, weitere Bibliotheken und Werkzeuge sollen folgen.

Selbst wenn Sie nicht auf Basis von AWS arbeiten, bleiben einige Möglichkeiten. Spring Cloud Netflix ermöglicht die Netflix-OSS-Integration über Spring Boot. Einige Firmen lassen Netflix OSS in Docker laufen [9]. Mittlerweile gibt es mit ZeroToDocker auch eine offizielle Netflix-Variante, um Teile des OSS-Stacks in Docker laufen zu lassen und so zumindest auszuprobieren [10]. Momentan findet man leider nur Asgard, Edda und Eureka von den hier erwähnten Tools, es sollen jedoch weitere Projekte folgen.

Sie können also einiges vom Klassenbesten abschreiben und die Ideen in Ihre eigene Lösung einfließen lassen. Denken Sie nur daran, dass technologische Lösungen alleine wenig helfen. Teamverantwortung bis ins Deployment und eine Kultur, in der Lernen wichtiger als Schuldzuweisung ist, sind wichtige Säulen von Resilience, die in diesem Kapitel nicht beleuchtet werden konnten.

## Links & Literatur

[1] Tseitlin, A.: „Resiliency through failure – Netflix's Approach to Extreme Availability in the Cloud", QCon NY, 2013

[2] Hystrix bei GitHub: https://github.com/Netflix/Hystrix

[3] Nygard, M. T.: „Release It! – Design and Deploy Production-Ready Software", O'Reilly, 2007

[4] Fowler, M.: „BlueGreenDeployment": http://martinfowler.com/bliki/BlueGreenDeployment.html

[5] ReactiveX-Website: http://reactivex.io

[6] Hystrix – How it Works: https://github.com/Netflix/Hystrix/wiki/How-it-Works

[7] Netflix OSS Repository: http://netflix.github.io/#repo

[8] Spring Cloud Netflix – Reference: http://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix.html

[9] R. Patel: „Netflix OSS, Meet Docker!“, 2013: http://nirmata.com/2013/10/netflix-oss-meet-docker

[10] ZeroToDocker: An easy way to evaluate NetflixOSS through runtime packaging, 2014: http://techblog.netflix.com/2014/11/zerotodocker-easy-way-to-evaluate.html

[11] Sussna, J.: „Are We Sure Netflix is Just an Edge Case?“: http://blog.ingineering.it/post/62910634258/are-we-sure-netflix-is-just-an-edge-case

# 3 Integration von Netflix Hystrix und Turbine

Microservices machen aus einem System ein verteiltes System. Da Netzwerk und Server ausfallen können, sind auch Microservices nicht immer verfügbar. Hystrix und Turbine von Netflix sind die Technologien, die Spring Cloud für dieses Problem wappnen.

Wenn ein Microservice ausfällt, kann das verheerende Folgen haben: Im schlimmsten Fall fällt jeder Microservice, der diesen Microservice nutzt, auch aus. Das pflanzt sich fort – und ehe man sich versieht, steht das gesamte System still. Ebenso können Systeme bei der Netzwerkkommunikation auf ungewöhnliche Probleme stoßen. Wenn beispielsweise eine Netzwerkverbindung vom Betriebssystem nicht erfolgreich aufgebaut wird, meldet es das Problem erst nach einem Time-out, das oft fünf Minuten oder länger ist. Wenn nun bei der Bearbeitung von Requests ein solches Problem auftritt, ist der Thread blockiert – und sehr schnell ist der Threadpool leer und nichts geht mehr.

Die Lösung liegt auf der Hand: Kommunikation mit anderen Microservices muss abgesichert werden. Beispielsweise muss ein Microservice in die Netzwerkkommunikation ein eigenes Time-out einbauen, um so schneller über Probleme informiert zu werden und das bereits erwähnte Leerlaufen des Threadpools auszuschließen. Eine weitere mögliche Maßnahme: Bei einem Ausfall eines Service tritt ein *Circuit Breaker* (engl. Sicherung) in Aktion: Die Sicherung springt beim ersten Fehler während eines Aufrufs heraus. Weitere Aufrufe gehen gar nicht mehr an den Service, sondern stoßen gleich auf einen Fehler. So wird das Time-out vollständig umgangen. Wenn der Service wieder zur Verfügung steht, kann die Sicherung schrittweise geschlossen werden, um eine Überlastung des Service zu vermeiden.

Außerdem kann statt des Aufrufs eines Service ein Default genutzt werden, wenn der Service nicht zur Verfügung steht. So kann das Gesamtsystem weiter funktionieren, auch wenn ein Service ausfällt. Dadurch entsteht ein Gesamtsystem, das eine sehr hohe Verfügbarkeit bieten kann. In einer klassischen monolithischen Architektur kann ein Fehler in einem Teil des Systems das gesamte System zum Ausfall bringen – beispielsweise kann ein Fehler in der Suche, der zu einem *OutOfMemoryError* führt, eine ganze Website zum Absturz bringen. Bei Microservices würde nur der Microservice für die Suche ausfallen, und die anderen Teile des Systems wären nicht beeinträchtigt, wenn entsprechende Vorkehrungen wie beispielsweise Default-Werte genutzt werden. So können Microservices trotz der Kommunikation über das Netzwerk eine höhere Verfügbarkeit bieten. Das Konzept, Fehler auf einen Service einzuschränken, geht auf das

Standardwerk „Release It!“ von Michael T. Nygard [1] zurück. Es ist fundamental für die Umsetzung wirklich stabiler Systeme.

## Hystrix

Eine bekannte Implementierung dieser Konzepte ist Hystrix [2], [3]. Es kommt aus dem Netflix-Stack und setzt Konzepte wie einen Circuit Breaker oder Time-outs für Aufrufe um. Basis sind Hystrix Commands, die den eigentlichen Methodenaufruf entsprechend dem Command-Pattern als eigenes Objekt modellieren und so einen Ansatzpunkt bieten, um die notwendigen Funktionalitäten einzubauen. Jeder Methodenaufruf kann mit Hystrix ergänzt werden, aber sinnvoll ist das vor allem bei der Schnittstelle zu anderen Systemen, bei denen eher solche Probleme auftauchen können.

Aber es wäre sicher wünschenswert, wenn diese zusätzlichen Funktionalitäten ohne die Implementierung eigener Klassen genutzt werden könnten. Schließlich kann bei Spring mit Annotationen wie *@Transactional* auch das Transaktionsverhalten gesteuert werden – und genau eine solche Steuerung mit Annotationen bietet das Projekt *hystrix-javanica*. Es ist ein Teil von hystrix-contrib [4]. Hinter den Kulissen nutzt *hystrix-javanica* Spring AOP, um die gewünschten Funktionalitäten zu den Spring Beans hinzuzufügen. Dieses Projekt liegt auch der Integration von Hystrix in Spring Cloud zugrunde.

```
public class OtherMicroService {

  @HystrixCommand
  public String simple() {...}

@HystrixCommand(commandKey = "circuitBreaker",
commandProperties = {
  @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",
value = "10")
 })
  public String circuitBreakerDemo() { ... }

@HystrixCommand(fallbackMethod = "fallback")
public String withFallback() { ... }

public String fallback() {...}

}
```

Listing 3.1

Listing 3.1 zeigt ein passendes Beispiel. Die Methode *simple()* trägt lediglich die Annotation *@HystrixCommand* und arbeitet dann mit den Default-Einstellungen. Hinter den Kulissen wird der Aufruf in ein Hystrix Command umgewandelt und durch einen anderen Thread ausgeführt, um so die Anwendung von einem möglichen Problem bei dem Aufruf weitgehend zu entkoppeln. So kann der Aufruf deswegen nach einer bestimmten Zeit abgebrochen werden, ohne dass dabei der aufrufende Thread beeinträchtigt wird.

Die Konfiguration des Aufrufs kann der Entwickler anpassen. So zeigt die Annotation der Methode *circuitBreakerDemo()*, wie der Threshold für den Circuit Breaker geändert

werden kann. Wenn zehn Methodenaufrufe innerhalb eines bestimmten Zeitfensters zu einer Exception führen, wird die Sicherung geöffnet. Weitere Aufrufe werden dann nicht an den Service weitergereicht, weil die Sicherung bzw. der Circuit Breaker das nun verhindert. Nach einer bestimmten Zeit werden Aufrufe wieder an den Service durchgelassen. Auch diese Einstellungen können alle angepasst werden. Natürlich können auch andere Werte angepasst werden. [5] enthält eine Auflistung. Die Möglichkeiten sind sehr umfassend und betreffen unter anderem die Ausführung des Commands, einen möglichen Fallback, den Circuit Breaker und auch Metriken.

Mit dem Wert *commandKey* aus der *HystrixCommand*-Annotation kann der Name des Commands festgelegt werden. Der Default ist der Name der Methode. Ebenso können mit *groupKey* mehrere Commands zu einer Gruppe zusammengefasst werden. Unter diesem Namen tauchen die Commands dann beispielsweise im Monitoring auf.

Die Methode *withFallback()* zeigt, wie bei einem Fehler eine andere Methode aufgerufen werden kann. Die Variable *fallbackMethod* nimmt den Namen der Methode auf, die bei einem Fehler aufgerufen werden soll. So kann also beim Ausfall eines Microservice ein Vorgabewert genutzt werden. Natürlich muss die Fallback-Methode dieselben Parameter übernehmen und denselben Rückgabewert haben.

Übrigens findet die Abarbeitung der Commands zwar in einem anderen Thread statt, aber sie ist trotzdem synchron. Wenn also eine der Methoden dieser Spring Bean aufgerufen wird, geht die Kontrolle erst dann an den Aufrufer zurück, wenn die Methode abgearbeitet ist. Der Thread dient also nur der Isolation. Wenn jedoch die Methode eine Instanz der Klasse *Future* zurückgibt wird, findet die Verarbeitung asynchron statt. Der Aufrufer übernimmt also sofort wieder die Kontrolle und kann später das Ergebnis auslesen. Ebenso ist es möglich, das Ergebnis als *Observable* zu erhalten. Das ist eine Klasse aus RxJava [6], einer reaktive Erweiterung für Java, die ebenfalls von Netflix kommt. Dann ist die Verarbeitung natürlich auch asynchron.

Um die Unterstützung für Hystrix zu aktivieren, muss in einer Spring-Boot-Anwendung an eine Configuration-Klasse die Annotation *@EnableHystrix* bzw. *@EnableCircuitBreaker* geschrieben werden. Die beiden Annotationen unterscheiden sich in der aktuellen Version von Spring Boot nicht voneinander, aber es wäre denkbar, dass in Zukunft zum Beispiel auch andere Implementierungen eines Circuit Breakers unterstützt werden. Als Abhängigkeit muss in das Projekt *spring-cloud-starter-hystrix* eingefügt werden.

Minimale Beispielanwendungen finden sich in dem neuen Testprojekt [7], beispielsweise zu Hystrix im Unterverzeichnis *hystrix*. Mit diesen Beispielen ist es sehr einfach, einen Eindruck von der Hystrix-Integration in Spring Cloud zu bekommen. Aber auch für die

anderen Features von Spring Cloud sind diese Beispielprojekte sehr hilfreich, da sie jeweils ein recht kleines Projekt enthalten, das jeweils das Feature demonstriert.

## Das Hystrix Dashboard

Ein weiteres Element von Hystrix ist das Dashboard. Es gibt mit einer Weboberfläche einen Eindruck davon, wie der Stand der Hystrix Commands gerade ist – ob also beispielsweise ein Circuit Breaker offen ist, wie oft entsprechende Commands abgesetzt worden sind und wie viele dieser Commands nicht erfolgreich waren. Dadurch kann das Dashboard zum Monitoring der Anwendung genutzt werden. **Abbildung 3.1** zeigt ein Beispiel: Offensichtlich ist der Command *circuitBreaker* gerade unter Last, und der Circuit Breaker für diesen Command ist auch gerade offen, weil zuvor eine entsprechende Anzahl Fehler beim Ausführen des Commands aufgetreten sind. In dem Dashboard wird außerdem der Stand der Thread-Pools gezeigt, die für die Ausführung der Commands genutzt werden.

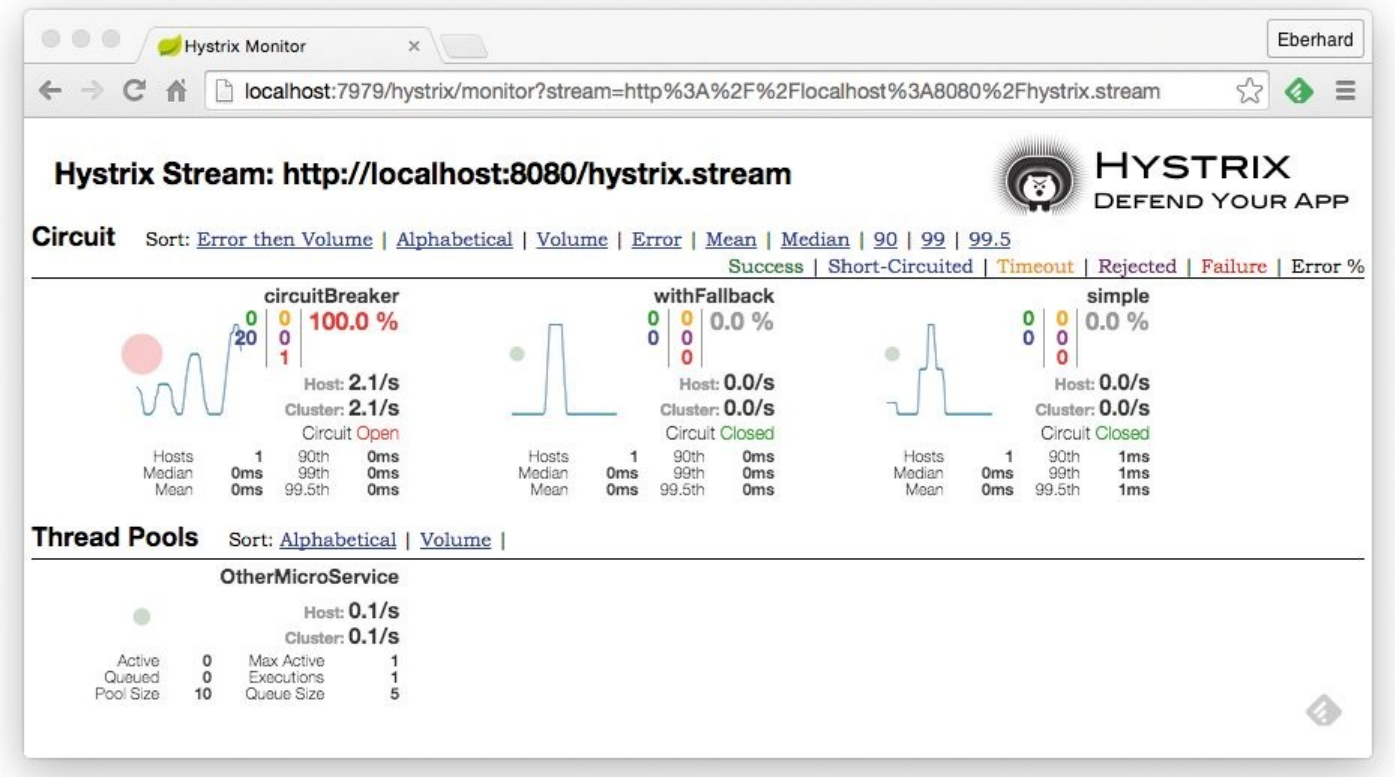


**Abbildung 3.1:** Screenshot der Hystrix-Monitor-Anwendung

Das Dashboard ist eine eigenständige Java-Webanwendung. Um das Dashboard zu nutzen, ist nur eine Spring-Boot-Anwendung mit der Annotation *@EnableHystrixDashboard* notwendig – beispielsweise enthält das Repository unter [8] ein solches Projekt. In dem Hystrix Dashboard kann eine beliebige Hystrix-Anwendung überwacht werden.

Die Datenübertragung erfolgt über den Hystrix-Stream-URL, beispielsweise *http://localhost:8080/hystrix.stream*. Unter diesem URL bietet die Anwendung die Daten über die verschiedenen Hystrix-Commands als JSON an. Dank HTTP Keep Alive schickt

die Anwendung ständig Updates. Damit eine Spring-Boot-Anwendung unter diesem URL auch tatsächlich Daten zur Verfügung stellt, muss *spring-boot-starter-actuator* als Abhängigkeit eingefügt werden. Dieses Projekt ist außerdem dafür verantwortlich, Monitoring-Daten für Spring Boot über eine HTTP-REST-Schnittstelle zur Verfügung zu stellen. Auch in diese Daten werden die Hystrix-Daten integriert. Wenn also auf Basis dieser Daten schon ein Monitoring umgesetzt ist, werden die Daten der Hystrix Commands dafür auch verfügbar.

## Größere Installationen mit Turbine

Das Monitoring eines einzigen Microservice ist natürlich nicht ausreichend. Typische Microservices-Installationen bestehen aus einer Vielzahl von Microservices. Und von jedem Microservice kann es mehrere Instanzen geben, um die Last auf mehrere Prozesse zu verteilen und auch beim Ausfall einer Instanz auch weiterhin den Dienst anbieten zu können. Also müssen die Daten der verschiedenen Microservices in einem Dashboard konsolidiert werden.

Netflix nutzt für die Konsolidierung der Daten aus den verschiedenen Hystrix-Instanzen ein eigenes Projekt: Turbine [9]. Letztendlich ist Turbine nicht sonderlich komplex: Es empfängt einfach die über HTTP von Hystrix zur Verfügung gestellten JSON-Daten. Diese Daten stehen typischerweise unter einem URL wie *http://<host:port>/hystrix.stream* zur Verfügung. Turbine fasst die Daten zusammen und stellt sie über HTTP als JSON unter einem URL wie *http://<host:port>/turbine.stream* zur Verfügung (**Abb. 3.2**).

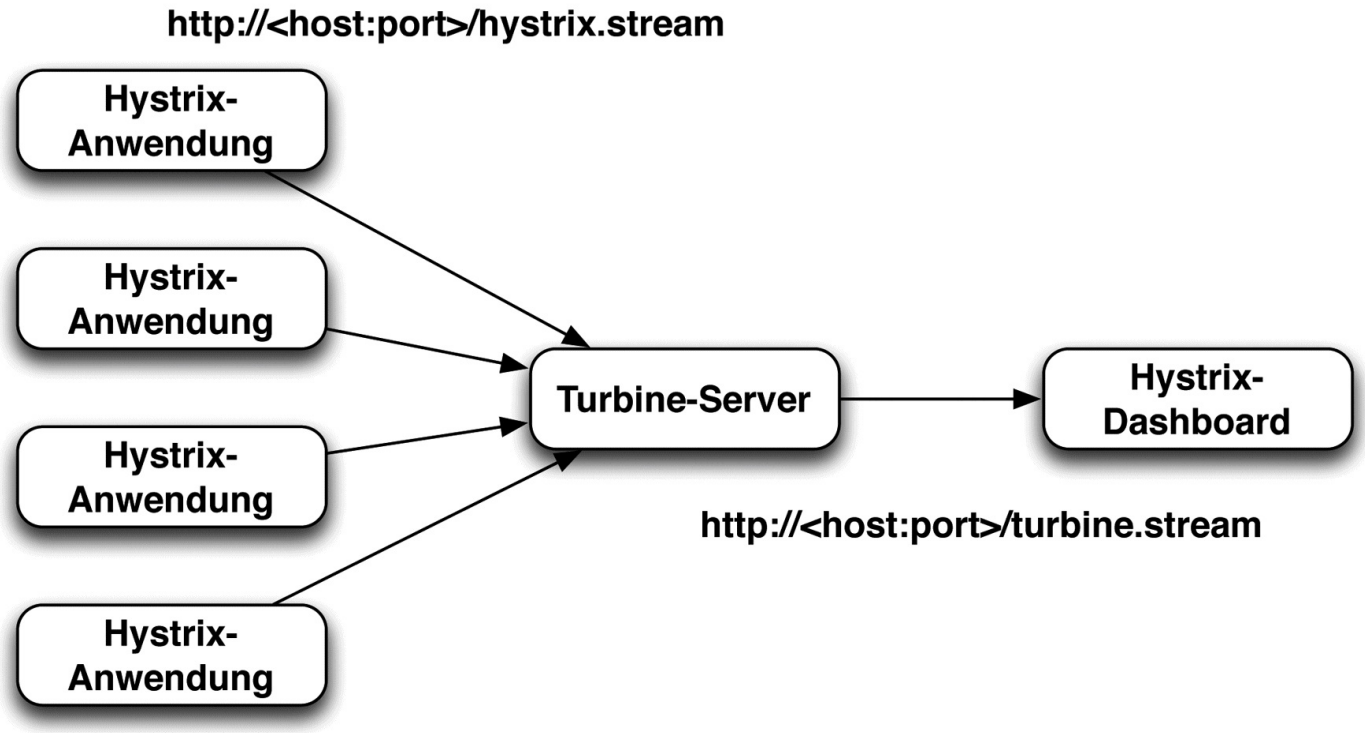


**Abbildung 3.2:** Konsolidiertes Hystrix-Monitorin mit Turbine

Die Datenformate sind identisch. So kann das Hystrix Dashboard sowohl mit dem Stream aus Hystrix arbeiten als auch mit einem Stream, den Turbine aus mehreren Quellen zusammengestellt hat. Der Nutzer muss nur den passenden URL beim Start des Hystrix Dashboards eingeben.

Die Frage ist nun natürlich, wie ein Turbine-Server die Anwendungen findet, deren Daten er zusammenfassen soll. Dazu nutzt Turbine Eureka, das wir schon im letzten Kapitel betrachtet haben. Eureka ist Netflix' Lösung, um Dienste im Netzwerk unter bestimmten Namen zu registrieren und zu finden. Also müssen die Anwendungen und Turbine so konfiguriert sein, dass sie denselben Eureka-Server nutzen und sich dort auch finden. Für Turbine muss dementsprechend ein Eureka-Server laufen, und außerdem müssen die Anwendungen so konfiguriert sein, dass sie sich bei Eureka registrieren. Dank Spring Cloud ist das aber nicht sehr aufwändig.

Der Turbine-Server selber ist mit Spring Boot und Spring Cloud wie schon die anderen Server sehr einfach umzusetzen: Es reicht die Annotation *@EnableTurbine* in der Konfigurationsklasse, um den entsprechenden Server zu starten. Ein einfaches Beispiel zum Experimentieren findet sich in dem schon erwähnten Projekt mit Beispielen für Spring Cloud [7] im Unterverzeichnis *turbine*. Es enthält neben dem Turbine-Server auch einen Eureka-Server und eine einfache Anwendung. So hat man recht schnell ein Set-up laufen, das als Basis für ein verteiltes System mit Hystrix und Turbine dienen kann – und es kann sogar direkt aus der IDE gestartet werden.

Dieses Set-up nutzt ein Pull-Modell: Der Turbine-Server holt per HTTP die Daten von den verschiedenen Hystrix-Anwendungen ab und stellt sie dann für das Dashboard zur Verfügung. Damit das funktioniert, muss eine HTTP-Verbindung möglich sein. Das ist aber nicht immer der Fall. Außerdem ist es oft wünschenswert, dass die Daten per Push übertragen werden. Eine Alternative für diesen Ansatz stellt AMQP dar [10]. Dieses Protokoll dient zur Übertragung von Nachrichten und wird von Messaging-Systemen wie RabbitMQ implementiert. Damit Turbine dieses Protokoll nutzt, muss der Turbine-Server mit *@EnableTurbineAmqp* annotiert werden und als Abhängigkeit *spring-cloud-starter-turbine-amqp* haben. Der Client muss mit einer Abhängigkeit zu *spring-cloud-netflix-hystrix-amqp* versehen werden, damit die Daten von Hystrix nicht wie sonst üblich über HTTP, sondern über AMQP übertragen werden.

## Fazit

Die Implementierung von Resilience mit Spring Cloud ist sehr einfach. Im Wesentlichen muss der Entwickler an den richtigen Stellen *@HystrixCommand*-Annotationen in den Code einfügen und gegebenenfalls passende Konfigurationen oder Fallbacks eintragen. Natürlich ist dann noch offen, was beim Ausfall eines Systems passieren soll, aber diese

Entscheidung ist nicht technisch, sondern oft fachlich. Soll beispielsweise beim Ausfall der Prüfung der Kreditwürdigkeit einfach die Bestellung doch angenommen werden, dann verzichtet man auf Sicherheit. Wenn beim Ausfall die Bestellungen nicht angenommen werden, verzichtet man auf Umsatz. Diese Entscheidung ist fachlich, kann aber mit Spring Cloud besonders einfach umgesetzt werden.

Die Implementierung basiert auf Hystrix, das sich gerade anschickt, in diesem Bereich zum Standard zu werden – schließlich ist es die Basis der Netflix-Architektur, die mit hoher Last umgehen muss und eine Art Vorbild für die Microservices-Architekturen darstellt. Diese Wahl ist also sicher richtig und sinnvoll.

Ein weiterer Vorteil von Hystrix ist das einfache Monitoring mit dem Hystrix Dashboard und das Konsolidieren verschiedener Messungen von Servern im Netz durch Turbine. Auch hier macht einem Spring Cloud das Leben einfacher. Alle Server sind mit einem einheitlichen Konzept betreibbar – letztendlich unterscheiden sie sich nur durch eine Annotation wie *@ EnableTurbine* oder *@EnableHystrix*. Leider erkennt man an dieser Stelle auch, dass die verschiedenen Netflix-Technologien alle recht eng zusammenhängen. Hystrix und Turbine nutzen Eureka, um die Server zu lokalisieren. Also zieht eine Entscheidung für Hystrix vermutlich auch gleich eine Entscheidung für Eureka nach sich.

## Links & Literatur


[1] Nygard, Michael T.: „Release It!: Design and Deploy Production-Ready Software“, Pragmatic Bookshelf, 2007, ISBN 978-0978739218

[2] Kraus, Holger; Landwehr, Arne: „Hystrix – Mehr Stabilität in verteilten Anwendungen“, in Java Magazin 8.2014

[3] https://github.com/Netflix/Hystrix

[4] https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib

[5] https://github.com/Netflix/Hystrix/wiki/Configuration

[6] https://github.com/ReactiveX/RxJava

[7] https://github.com/spring-cloud-samples/tests

[8] https://github.com/spring-cloud-samples/hystrix-dashboard

[9] https://github.com/Netflix/Turbine

[10] http://www.amqp.org

# Die Autoren

**Uwe Friedrichsen** ist ein langjähriger Reisender in der IT-Welt. Als Fellow der codecentric AG ist er stets auf der Suche nach innovativen Ideen und Konzepten. Seine aktuellen Schwerpunktthemen sind Skalierbarkeit, Resilience und die IT von (über)morgen. Er teilt und diskutiert seine Ideen regelmäßig auf Konferenzen, als Autor von Artikeln, Blog-Posts, Tweets und natürlich gerne auch im direkten Gespräch.

**Stefan Toth** ist Berater der embarc GmbH und unterstützt Unternehmen aus unterschiedlichen Branchen in Sachen Softwarearchitektur. Neben breitem technologischen Kontext ist methodische Erfahrung aus agilen Projekten, Architekturbewertungen und IT-Transformationen sein größtes Kapital.

Website: *st@embarc.de*

Twitter: *@st_toth*

**Eberhard Wolff** arbeitet als freiberuflicher Architekt und Berater. Außerdem ist er Java Champion. Sein technologischer Schwerpunkt liegt auf Spring, NoSQL und Cloud.

Website: *http://ewolff.com*

Twitter: *@ewolff*