Robustheit und Antifragilität: Eignen sich Microservices für die Systeme der Zukunft?

Solarkraftwerke sind ausfallsicher und skalieren perfekt. Beim Vergleich mit Microservice-orientierten Architekturen zeigen sich Ähnlichkeiten, aber auch Unterschiede, welche die Unvollkommenheit moderner Softwaresysteme erklären. Wir beschreiben die Herausforderungen beim Entwurf und Betrieb Microservice-orientierter Systeme mit dem Fazit, dass noch viel zu tun ist, bis Software so robust sein wird wie Solarkraftwerke. Aber das eigentliche Ziel sind antifragile Systeme, und hier stehen wir noch ganz am Anfang.

Antifragile Systeme [Tal12] werden gebaut in der Annahme, dass zu jedem Zeitpunkt alles kaputtgehen kann (*everything fails all the time*, Werner Vogels, CTO von Amazon). Der Ausfall einer oder mehrerer Komponenten ist kein beklagenswerter Einzelfall, sondern die Standardsituation. Antifragile Systeme überstehen Störungen nicht nur unversehrt, sondern im besten Fall gestärkt. Insofern ist Antifragilität wesentlich mehr als nur Ausfallsicherheit.

Das wichtigste Beispiel für Antifragilität ist die Evolution: Sie funktioniert seit vielen Milliarden Jahren mit bemerkenswerten Ergebnissen und hat zahllose Katastrophen gestärkt überdauert. Ein wesentlich schlichteres Beispiel sind Solarkraftwerke. Sie bestehen aus vielen Tausend gleichartiger Panels, die in Gruppen zusammengeschaltet sind und je nach Sonnenschein mehr oder weniger Strom produzieren. Ihre simple Architektur hat viele Vorteile:

(1) Solarkraftwerke baut man Stück für Stück, ein Panel nach dem anderen. Schon nach kurzer Zeit kann man mit der ersten verfügbaren Fläche Strom produzieren; Erweiterungen sind jederzeit möglich.
(2) Solarkraftwerke skalieren im Rahmen des verfügbaren Grunds beliebig. Wenn man die Fläche verdoppelt, erhöht sich die Komplexität nur wenig.
(3) Solarkraftwerke sind robust: Bei 10 Prozent defekten Panels reduziert sich die Gesamtleistung um 10 Prozent. Der Austausch defekter Panels ist unkritisch und passiert ständig.
(4) Viele billige Panels sind besser als wenige teure. Die maximale Energieausbeute ist nicht so wichtig, denn dort, wo Solarkraftwerke stehen, zum Beispiel in der Wüste, ist der Grund praktisch kostenlos.
(5) Ein Solarkraftwerk kann ohne Weiteres Panels von verschiedenen Herstellern in verschiedenen Versionen beherbergen, solange die Anschlüsse passen.
(6) Solarkraftwerke sind einfach zu planen und in Betrieb zu nehmen. Sollte man sie eines Tages nicht mehr brauchen, entsteht Müll in Form verbrauchter Panels, die man wie Altglas entsorgt.

Solarkraftwerke sind robust, aber noch nicht antifragil. Kernkraftwerke dagegen sind riesige, nicht erweiterbare Monolithen mit endlosen Planungs- und Stilllegungszeiten. Die geringste Panne verursacht den Stopp des Gesamtsystems. Kernkraftwerke sind fragil.

Robustheit in Hard- und Software

Die großen Internet-Anbieter haben das Prinzip der Solarkraftwerke auf Hardware übertragen: Moderne Rechenzentren bestehen aus vielen Racks mit jeweils mehreren Hundert Blades. Ständig sind einige Blades kaputt, ohne das Gesamtsystem nennenswert zu beeinträchtigen. Die Rechner-Monolithen der Vergangenheit haben ausgedient, denn eine Farm mit 100.000 einfachen Rechnern ist billiger und mit weniger Stress zu betreiben. Voraussetzung ist ein Betriebssystem wie Mesos [Hin11], das die Rechner-Farm verwaltet und als anonyme, beliebig abrufbare CPU-Power bereitstellt.

Microservices [New15] spielen in der Softwarearchitektur eine ähnliche Rolle wie Panels im Solarkraftwerk oder Frames in der Hardware-Farm. Systeme mit Microservice-Architektur nennt man *Mode 2*, herkömmliche Architekturen *Mode 1*, und gemischte Formen heißen *bimodal*.

Mode 2-Systeme haben durchschlagenden Erfolg: Die großen Internet-Anwendungen von Google, Twitter oder Netflix sind unglaublich schnell, unbegrenzt skalierbar, völlig ausfallsicher und offenbar mit vertretbarem Aufwand zu implementieren, zu pflegen und zu betreiben. Neben diesen Internet-Größen gibt es aber so gut wie keine Anwender, deren Microservices das Experimentierstadium verlassen haben. Der erste Grund dafür liegt in der Unreife der heutigen Produkte und im Fehlen von Standards. Darüber sprechen wir im nächsten Abschnitt.

Der zweite Grund liegt tiefer: Die Analogie von Solarpanels und Microservices ist bestechend, aber leider nur teilweise richtig, denn:

- Solarpanels sind weitgehend identisch und austauschbar. Microservices sind aber alle verschieden; Identität und Austauschbarkeit gilt nur für parallele Instanzen eines einzelnen Microservice.
- Solarpanels kommunizieren untereinander nicht, Microservices tun das dauernd. Der passende Schnitt von Microservices in der richtigen Größe, dem richtigen Maß an innerer Kohäsion und äußerer Koppelung ist in der Mode 2-Welt schwerer zu finden als in Mode 1. Grund dafür ist unter anderem der hohe Grad an Parallelität und Asynchronität in Mode 2-Systemen.
- Solarpanels haben kein Gedächtnis. Einige wenige Microservices sind tatsächlich gedächtnislos, viele aber nicht. Deren Gedächtnis liegt manchmal im Hauptspeicher, oft in der Datenbank und ist bei Ausfall und Wiederanlauf immer ein Problem.
- Der Zustand eines Solarkraftwerks ist im Wesentlichen gegeben durch den Zustand der einzelnen Panels. Der Zustand eines Mode 2-Systems ist ungleich komplizierter, bedingt durch die

zahlreichen Kommunikationsprozesse, die zu jedem Zeitpunkt laufen und überwacht werden müssen.

Daher sind Mode 2-Architekturen bisher nicht besonders robust, geschweige denn antifragil. Wir haben weiter oben sechs wichtige Eigenschaften von antifragilen Systemen genannt. Davon sind nur zwei in vollem Umfang gültig, die anderen vier aber nur mit erheblichen Einschränkungen:

(1) Mode 2-Systeme entstehen Service für Service. Schon nach kurzer Zeit kann man funktionsfähige Teilsysteme in Betrieb nehmen: *Teilweise richtig*, und zwar unter der Voraussetzung einer klaren, tragfähigen Architektur. Auch und gerade bei Microservices droht der Worst Case mit $O(n^2)$ Punkt-zu-Punkt-Verbindungen.
(2) Mode 2-Systeme skalieren beliebig: Jeder einzelne Microservice läuft in beliebig vielen Instanzen. *Richtig:* Den einzelnen Microservice kann man vervielfachen, ohne die Komplexität nennenswert zu erhöhen. Die Infrastruktur rund um Microservices sorgt dafür, dass horizontale Skalierung wenige Seiteneffekte produziert.
(3) Mode 2-Systeme sind robust. Microservices sind perfekt gegeneinander abgeschottet. *Teilweise richtig:* der Ausfall einer Instanz eines Microservice ist unschädlich, solange noch mindestens eine parallele Instanz überlebt. Aber wenn die letzte Instanz auch noch stirbt, droht der Ausfall des Gesamtsystems.
(4) Der einzelne Microservice ist klein und überschaubar. Reparatur oder Austausch gegen eine alternative Implementierung geht schnell. *Teilweise richtig:* Der Austausch eines Microservice ist technisch leicht. Test und Integration sind lösbar, aber alles andere als offensichtlich.
(5) Ein einzelnes Mode 2-System kann ohne Weiteres Microservices aus verschiedenen Quellen oder unterschiedlicher Hersteller unter einem Dach verbinden, solange gewisse Standards eingehalten sind. *Richtig.*
(6) Mode 2-Systeme sind einfach zu planen und in Betrieb zu nehmen: Man entwirft eine Microservice-Architektur nach der anderen und unabhängig voneinander. *Teilweise richtig:* Auch eine Microservice-Architektur erfordert eine sorgfältige Planung, bevor man den ersten Microservice aufsetzt. Ferner ist die Integration eines Systems von Microservices ungleich komplexer. Trotzdem skaliert die Entwicklung gut, weil hinter jedem Microservice ein dediziertes Team steckt, und diese Teams können weitgehend unabhängig voneinander arbeiten.

Mode 2-Systeme sind somit ein guter Ansatz, aber bestimmt kein Silver Bullet für robuste, geschweige denn antifragile Anwendungen. Wir beschreiben die wichtigsten Entwurfsprinzipien bei der Entwicklung von Mode 2-Systemen.

Wie entwirft man Mode 2-Systeme?

Microservices sind komplexer als Solarpanels: Sie sind alle verschieden, sie kommunizieren miteinander, und viele haben ein Gedächtnis. Es besteht die Gefahr, dass die Komplexität der verschiedenen Tools, der zahlreichen Microservices und deren Kommunikation die angestrebte Robustheit zunichtemacht. Deshalb ist die Architektur von Mode 2-Systemen eine besondere Herausforderung. Folgend befassen wir uns mit den wichtigsten Entwurfsprinzipien.

Von Komponenten zu Microservices

Der Komponentenbegriff, der sich seit 40 Jahren über zahllose Publikationen hinweg gefestigt hat, ist unabhängig von der vorliegenden Technik. Das gilt auch für Mode 2: Komponenten sind das, was sie schon immer waren. Microservices sind eine neue, attraktive Art der Verpackung von Komponenten, eine neue Laufzeitumgebung. Sie sind ein technischer Fortschritt, aber keine Alternative oder gar Konkurrenz zu Komponenten. Andere Komponenten-Verpackungen sind zum Beispiel COM+, Spring, OSGi oder Jigsaw, und auch ein Applikationsserver wie Tomcat oder Glassfish ist nichts anderes als eine Laufzeitumgebung für Komponenten.
Wir sollten aus den Fehlschlägen lernen: Die große CORBA-Revolution blieb aus, da die OO-Schnittstellen viel zu kleinteilig waren. Auch DCOM war ein Schritt in Richtung Microservices, der sich nicht richtig durchgesetzt hat.
Die Kernfrage lautet: Wie bilde ich meine Komponenten, die ich technikunabhängig entworfen habe, auf Microservices ab? Implementiere ich Schnittstellen mit SOAP, REST oder einem binären Format? Es wäre ein Fehler, mit Microservices anzufangen und das ganze System um diese Idee herum zu bauen. Stattdessen beginnt man mit einer technikunabhängigen Architektur und entscheidet spät und revidierbar, am besten über Konfigurationsparameter beim Deployment, welche Microservices in welcher Form tatsächlich instanziiert werden.
Microservices stehen am Ende des Entwurfsprozesses, nicht am Anfang. Regeln der Bauart: „Ein Microservice hat mindestens (oder meinetwegen höchstens) 1.000 LOC“ sind nicht hilfreich. Auf der Basis einer soliden, tragfähigen Architektur gerät der Entwurf von Microservices zu einer angenehmen, stressfreien Betätigung, die unter Beachtung einiger offensichtlicher Regeln fast schematisch abläuft: Intensiv kommunizierende Komponenten wird man in einem Microservice zusammenfassen, Komponenten mit unterschiedlichen Release-Zyklen wird man trennen, und bei Konflikten muss man sich etwas einfallen lassen.
Eine besonders wichtige Option heißt *Monolith first* [Fow15] und bedeutet, dass das ganze System bei Bedarf als Monolith deploybar ist – eine nicht zu unterschätzende Hilfe bei Test und Diagnose. Erst in der Produktion sind fachliche Komponenten eigenständige Deployment-Einheiten. Ein anderer Trick besteht in der Entkoppelung von Protokollen: Man beginnt mit einem Textprotokoll und wechselt später, zum Beispiel aus Performanzgründen, praktisch ohne Extra-Aufwand zu einem Binärprotokoll.

Infrastruktur

Microservices laufen nicht im luftleeren Raum, sie benötigen eine Infrastruktur (**siehe Abbildung 1**) um sie herum, die sie bestmöglich vor der Komplexität schützt, die mit verteilten Systemen oft einher kommt.
Microservices benötigen typischerweise die folgende Infrastruktur um sie herum:

- *Konfiguration und Koordination (Service Configuration & Coordination):* Dieser Dienst stellt Konsens über alle Knoten in einem verteilten System her. Dies kann Konsens über den Wert eines Konfigurationsparameters, den Zustand einer Sperre oder auch den für eine bestimmte Aufgabe führenden Knoten sein. Beispiele hierfür sind Consul, etcd und ZooKeeper.
- *Service Discovery:* An diesem Dienst kann ein Microservice seine Services registrieren. Dies erfolgt über eine Service-Id, den Endpunkt und beliebige

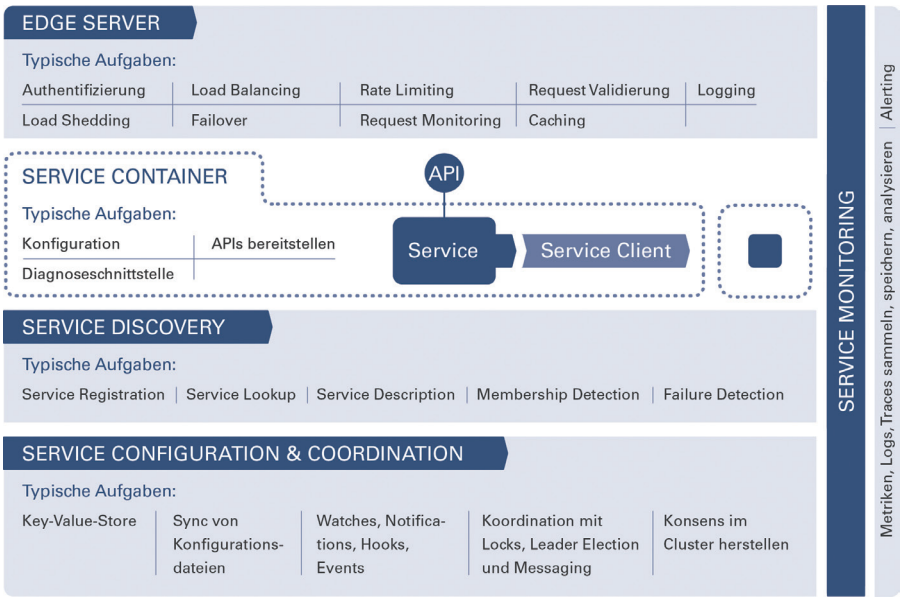

Abb. 1: Microservice-Plattform als Infrastruktur.

Metadaten. Andere Microservices können dann anhand einer Service-Id oder Metadaten auf die verfügbaren Endpunkte zugreifen. Beispiele hierfür sind Consul und Eureka.

- *Edge Server:* Dieser bildet das Tor zur Außenwelt, sprich zum Internet. Er nimmt alle Anfragen von außen entgegen und verteilt sie mit Hilfe der Service Discovery an die Microservices. Beispiele hierfür sind Zuul, traefik und fabio.

Die Microservices mitsamt der Infrastruktur drum herum benötigen eine Ausführungsumgebung, die sie zuverlässig verteilt und laufen lässt. Eine solche Ausführungsumgebung wird als *Cluster-Orchestrierer* bezeichnet. Die Idee ist, alle Ausführungseinheiten wie Microservices und Infrastruktur-Bausteine in Container zu verpacken und diese Container dann über einen speziellen Scheduling-Mechanismus im Cluster zu verteilen – ähnlich wie es der Betriebssystem-Scheduler für einen Rechner macht, hier eben aber für viele Rechner gleichzeitig. Beispiele für Cluster-Orchestrierer sind Kubernetes, DC/OS Marathon oder Docker Compose.

Zustände

Zustandslos im engeren Sinn sind reine Berechnungsmodule, wie eine Bibliothek mit mathematischen Funktionen. Aber in Wirklichkeit haben fast alle Microservices einen Zustand, den man oft in die Datenbank auslagert. Der Service selbst ist damit aus technischer Sicht zustandslos, hängt aber ab von der Verfügbarkeit der Datenbank.

Eine große Hilfe bei der Verwaltung von systemweiten Zuständen sind Tools wie Apache Ignite, Geode oder Hazelcast. Diese stellen einen Cache zur Verfügung, der eine systemweite, knotenübergreifende Sicht auf langlebige Daten ermöglicht. Damit können Daten nah an der Anwendung gehalten werden. Die genannten Caches bieten einen komfortablen Zugriff mit allen Java-Sprachmitteln inklusive Streams und Lambda-Funktionen, und sie implementieren intelligente Strategien zur Datenlokalität: Jeder Service findet die Daten, auf die er zugreift, soweit wie möglich lokal.

Sie alle unterliegen jedoch dem CAP-Theorem [Gil02], einer Unmöglichkeitsaussage für verteilte Systeme: Von den drei Eigenschaften *Konsistenz* (C wie Consistency), *Verfügbarkeit* (A wie Availability) und *Partitionstoleranz* (P wie Partition Tolerance) kann ein verteiltes System nur zwei gleichzeitig realisieren (**siehe Abbildung 2**).

Der übliche Ausweg besteht darin, dass man Verfügbarkeit und Partitionstoleranz behält, die Forderung nach *sofortiger Konsistenz* aber abschwächt auf *verzögerte Konsistenz* (*finally consistent*) ohne Zeitgarantie. Microservices auf verschiedenen Knoten sehen also möglicherweise unterschiedliche Stände, was eine besondere, sehr defensive Art der Programmierung erfordert.

Weil Microservice-Architekturen ohnehin im Wesentlichen asynchron arbeiten, hätte man das Problem der Inkonsistenz auch ohne CAP. CAP macht also die Welt nicht viel komplizierter, als sie ist.

Kommunikation

Lokale Kommunikation innerhalb eines Microservice ist schnell und meistens synchron. Kommunikation zwischen verschiedenen Microservices ist um den Faktor 100 bis 1000 langsamer und allein deswegen meistens asynchron.

Daraus ergeben sich harte Restriktionen für den Schnitt von Microservices. Jede Kommunikation schafft Abhängigkeiten: Manche sind unvermeidbar, manche überflüssig und einige tödlich. Zyklenfreiheit

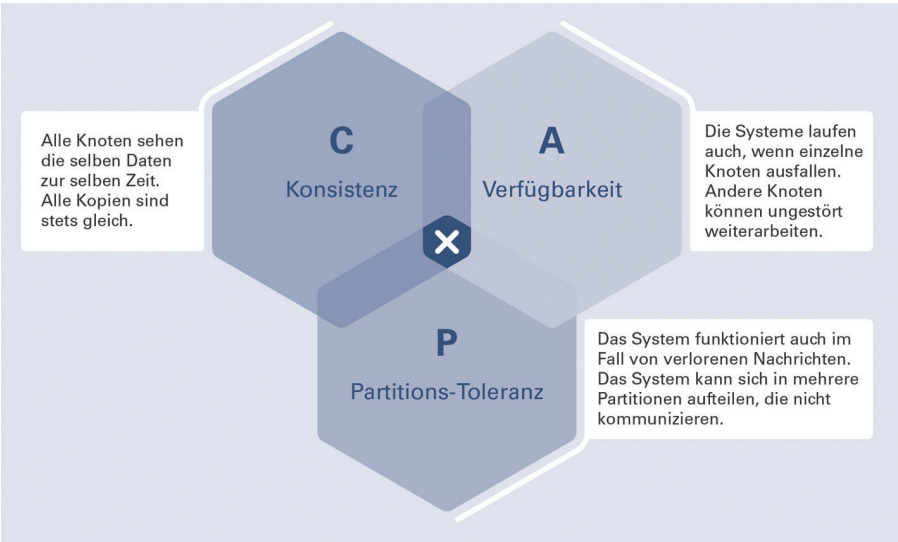

Abb. 2: Das CAP-Theorem.

im Abhängigkeitsgraphen ist ein ehernes Gesetz, das im Mode 2 noch an Bedeutung gewinnt. Ein Brei von Microservices mit $O(n^2)$ Spaghetti-artigen Punkt-zu-Punkt-Beziehungen wäre das Ticket zur Hölle. Die in Mode 2-Systemen unvermeidliche Asynchronität verzögert die Konsistenz der Daten und macht herkömmliche Transaktionslogik schwer bis unmöglich.
Die gute Nachricht ist, dass uns das CAP-Theorem nur wegnimmt, was wir sowieso nicht haben, nämlich sofortige Konsistenz.

Datenbanken
Die meisten Systeme verwenden eine oder mehrere Datenbanken. Ohne Datenbank geht es nicht. Wir sehen zwei typische Wege der Integration von Datenbanken in Mode 2:

- Man betrachtet die Datenbank als externes System in Mode 1. Der Zugriff darauf erfolgt über einen der genannten Caches. Dies funktioniert gut; allerdings profitiert die Datenbank nicht von den Segnungen der Microservice-Architektur. Sie skaliert in Mode 1 so gut oder schlecht wie vorher.
- Man zerlegt die Datenbank in mehrere Shards und ordnet diese unter dem Gesichtspunkt der Datenlokalität den Microservices zu. So könnte zum Beispiel ein Kundenservice in zehn parallelen Instanzen laufen, wobei jede Instanz einen Nummernkreis bedient. Sharding wird von zahlreichen NoSQL-Datenbanken unterstützt, wie Solr und Cassandra. Bekannte Strategien zur Bildung von Shards sind Replikation, vertikale Zerlegung (z.B. nach Nummernkreisen), horizontale Zerlegung (nach Attributen oder Tabellen) oder eine Kombination dieser Strategien. All das ist noch wenig erprobt; Datenmanagement auf zwei Ebenen (Cache und Shards), mit überlappenden Funktionen für Verteilung und Konsistenz, ist eine echte Herausforderung.

Transaktionen
Im Mode 2 gelten sinngemäß dieselben Regeln für den Umgang mit Transaktionen wie in Mode 1: Jeder Microservice hat seine Datenhoheit; technische Transaktionen operieren immer nur lokal im Rahmen eines Microservice. Für übergreifende oder hierarchische Transaktionen braucht man immer fachliche Logik, zum Beispiel einen fachlichen Rollback. Die Koppelung verschiedener Microservices durch einen 2-Phasen-Commit würde genau die Abhängigkeiten erzeugen, die man vermeiden will.

User Interface
Die bekannten Architekturprinzipien gelten weiterhin. Microservices können ihre eigene Benutzeroberfläche (UI) mitbringen, die dann zum Beispiel über ein Portal erreichbar ist. Das nennt man dann auch Self-Contained System [SCS16].
Aber viele Microservices eigenen sich überhaupt nicht für den entfernten Aufruf vom Client, wo Java-Script im Browser seinen Dienst tut: Manche sind zu kleinteilig, andere erzeugen Race Conditions zwischen Client und Server. In diesem Fall baut man UI-Adapter zur Aufbereitung vorhandener Microservices. Der Verzicht auf derartige Adapter rächt sich durch endlose Antwortzeiten und Unbenutzbarkeit. Wir empfehlen dringend das Single-Page-Prinzip: Die UI-Logik läuft komplett im Browser (implementiert in einer Java-Script-Version) und sieht den Server über eine textorientierte Schnittstelle.

Wie gut sind Mode 2-Systeme wirklich?

Wir haben bereits gesehen, dass Mode 2-Systeme noch viel zu wünschen übrig lassen. Es besteht die Gefahr, dass die Komplexität der Infrastruktur das zweifellos vorhandene Potenzial zunichtemacht.

Diagnostizierbarkeit
Mode 2-Systeme sind schwer zu diagnostizieren, denn vieles passiert gleichzeitig, zahllose Race Conditions sind möglich, Fehler lassen sich oft nur schwer reproduzieren. Daher braucht man noch mehr als in Mode 1 alle erdenklichen Arten von Logs, Traces und Laufzeitmetriken. Diese laufen automatisch mit und werden in einer eigenen, sehr großen Datenbank gesammelt und nicht nur im Fehlerfall, sondern laufend, auf Anomalien untersucht. Dafür gibt es Werkzeuge wie den ELK-Stack für Logs, ZipKin für Traces und Prometheus für Metriken. Man kann dieses Thema nicht überbetonen; klassische Logfiles, die man mit ein paar grep-Aufrufen untersucht, konnten noch nie befriedigen und sind heute völlig unzureichend.

Robustheit
Das Reactive Manifesto [RM] fordert für jeden Dienst Hochverfügbarkeit, Ausfallsicherheit, Performanz und Asynchronität. Die Fehlerbehandlung liegt im Wesentlichen bei den einzelnen Microservices, also in den Händen der Entwickler oder der Lieferanten. Die Formulierung und Durchsetzung der nötigen Regeln defensiver Programmierung ist eine enorme Herausforderung, aber das reicht noch nicht: Wir dürfen Fehler und Ausnahmen nicht mehr als unangenehme Sonderfälle betrachten, die selten oder nie auftreten. In Wirklichkeit gilt: *everything fails all the time*, und deswegen ist die Fehlerbehandlung integraler Bestandteil des Gesamtsystems und wird auch ständig durchlaufen. Der zugehörige Code ist daher genauso zu testen wie jeder andere, mit den offensichtlichen Konsequenzen für die Testabdeckung. Trotzdem gilt wie immer die Trennung der Zuständigkeiten. Ein Circuitbreaker wie Histrix oder ein Handler für asynchrone Kommunikation wie RxJava hat in der eigentlichen Anwendung nichts zu suchen.
Ausfallwahrscheinlichkeiten multiplizieren sich: Ein redundanzfreies System von zehn unabhängigen Services mit einer Ausfallwahrscheinlichkeit von jeweils 0,1 Prozent versagt insgesamt mit der Wahrscheinlichkeit[1]) von 1 Prozent. Deshalb baut man Systeme redundant, mit parallelen Instanzen einzelner Services. Aber dies erfordert zusätzliche Logik für den Fall, dass eine, zwei oder noch mehr dieser Instanzen ausfallen.

Skalierbarkeit
Skalierbarkeit ist für sich genommenen wohl der wichtigste Vorteil von Mode 2: Jeder Service läuft ohne Weiteres in beliebig vielen parallelen Instanzen. Aber auch hier gibt es kein *free lunch*:

- Parallele Instanzen müssen verwaltet werden: Man muss sie hochfahren, die Last gleichmäßig verteilen, überwachen und beim Ausfall einer Instanz reagieren.
- Man muss ihre Zustände verwalten, im einfachsten Fall in einer einzigen Datenbank, was die Skalierung begrenzt, oder in mehreren (unter Verwendung von Sharding und Replikation), was die Komplexität noch weiter erhöht.

Testbarkeit
Alle Regeln für Softwaretests gelten auch in Mode 2. Natürlich wird jeder Microservice für sich getestet, aber das reicht nicht. Auch

[1]) Die Rechnung dazu: $1 - 0{,}999^{10} \approx 0{,}01$

Microservices werden Schritt für Schritt über mehrere Ebenen integriert, wie bisher unter Verwendung verschiedener Umgebungen, bis hin zur Produktionsumgebung. Dafür braucht man ausgefeilte Verfahren, die komplexer und fehleranfälliger sind als in Mode 1. Es gibt aber mindestens drei wichtige Verbesserungen:

- Man kann neue Services in die Produktionsumgebung übertragen und zunächst nur einem reduzierten Benutzerkreis zur Verfügung stellen, sofern man die Abwesenheit von Seiteneffekten in einer Integrationsumgebung geprüft hat.
- Man kann verschiedene Versionen eines Service parallel einsetzen, solange man den Überblick behält.
- Komplette Deployments sind die Ausnahme. In der Regel werden immer nur einige wenige Services in die Produktion überstellt. Dies erleichtert und beschleunigt die Produktivsetzung.

Fazit

Microservices sind eine tolle, aber junge und unreife Idee. Sie sind ein wichtiger Schritt in Richtung Robustheit, aber der Weg dahin ist noch weit, und der zur Antifragilität noch weiter. Im ersten Schritt sehen wir zwei wesentliche Voraussetzungen für den Erfolg von Mode 2:

- Das Denken in Komponenten ist die Grundlage jeder tragfähigen Softwarearchitektur, egal welcher Couleur. Es wäre falsch, ein System um Microservices herum zu bauen, genauso wie es falsch war, Corba oder COM+ zum Zentrum der Architektur zu machen.
- Wir brauchen reife, zuverlässige Tools, die narrensicher und mit großer Selbstverständlichkeit funktionieren. Aufbau und Inbetriebnahme einer funktionsfähigen Cloud, Deployment und Start einer Microservice-Anwendung sind bisher eine Sache für leidensfähige Profis; Nicht-Profis sind hoffnungslos überfordert. Das kann so nicht bleiben.

Niemand weiß, ob sich die Erwartungen in Mode 2 erfüllen werden. Die Gefahr besteht, dass ungeschickte Architekturen und komplexe Tools die Fragilität unserer Systeme noch vergrößern. Und das eigentliche Ziel ist ja nicht nur Robustheit, sondern Antifragilität, also die Fähigkeit der Software, auf Störungen aktiv zu reagieren, aus Fehlern zu lernen und immer besser zu werden.

Die ersten Ansätze dazu gibt es schon: Software kann die eigenen Logs analysieren und daraus Konsequenzen ziehen, wie Zuschalten oder Abschalten von Servern, oder Neustart einer Datenbank. Das gesamte Arsenal an Machine-Learning-Techniken lässt sich auf die Behandlung von Fehlern und Ausnahmen anwenden, in der Hoffnung, dass immer mehr Sonderfälle einer automatischen Behandlung zugeführt werden können.

Die Zukunft ist also vielversprechend, und es gibt eigentlich nur eine Falle, die wir vermeiden müssen: Selbstzufriedenheit und das Gefühl, mit dem jeweils letzten Hype schon alles erreicht zu haben. ||

Literatur & Links

[Fow15] M. Fowler, MonolithFirst, 3.6.2015, siehe: `http://martinfowler.com/bliki/MonolithFirst.html`

[Gil02] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, in: ACM SIGACT, 2002

[Hin11] B. Hindman et al., Mesos: a platform for fine-grained resource sharing in the data center, in: Proc. NSDI'11, 2011

[New15] S. Newman, Building Microservices, O'Reilly, 2015

[RM] The Reactive Manifesto, Version 2.0, 16.9.2014, siehe: `http://www.reactivemanifesto.org/`

[SCS16] Self-Contained Systems, siehe: `http://scs-architecture.org`

[Tal12] N. N. Taleb, Antifragile. Things that Gain from Disorder, Penguin Books, 2012

Die Autoren

|| Dr. Josef Adersberger
(josef.adersberger@qaware.de)
ist Chefarchitekt, Geschäftsführer und Mitgründer von QAware. Hier ist er verantwortlich für Software-Engineering und Weiterbildung. Er lehrt und publiziert zu Themen des Software-Engineerings, aktuell insbesondere zu Cloud-Computing.

|| Prof. Dr. Johannes Siedersleben
(johannes.siedersleben@qaware.de)
war unter anderem Chefarchitekt bei T-Systems und entwickelte an der Hochschule Rosenheim den Studienschwerpunkt Software-Engineering. Er ist Autor diverser Publikationen zum Thema „Softwarearchitektur" und berät QAware in allen Gebieten des Software-Engineerings.

|| Johannes Weigend
(johannes.weigend@qaware.de)
ist Chefarchitekt, Geschäftsführer und Mitgründer von QAware, einem IT-Projekthaus mit Fokus auf Cloud-Native-Anwendungen und Softwaresanierung. Er verantwortet den Bereich Forschung und Entwicklung. Schwerpunkt seiner Publikationen ist unter anderem die Softwareanalyse.