Welche Methoden sind zum Testen von Software-Resilienz verwendbar?

Böhm Tobias
Technische Hochschule Ingolstadt

***Zusammenfassung*—Dieses Dokument gibt einen Überblick über das Themengebiet der Software-Resilienz und bespricht anschließend Möglichkeiten, diese in der Theorie und Praxis auf ihre korrekte Implementierung und Funktionalität zu testen. Ferner wird beispielhaft auf eine Richtlinie des Hardware- und Softwareausstatters IBM und eine Musterimplementierung des Streaminganbieters Netflix eingegangen.**

I. Einleitung

„Resilient Software Design“ ist der Name eines noch jungen Fachgebietes in der Informatik, das sich mit dem Entwurf von hochverfügbaren Anwendungen beschäftigt. Resilienz ordnet sich in der Klassifizierung der Fehlertoleranz wie folgt ein:

- Wenn das Programm nach Auftreten eines Fehlers in einem sicheren Zustand stoppt, wird es als „Fail-stop“ oder „Fail-safe“ bezeichnet.
- Wenn es auf permanente Fehler im System reagieren kann, handelt es sich um ein System, das „Fault-tolerant“ ist.
- Kann es die Ausführung mit reduzierter Funktionalität fortsetzen, beherrscht es zudem „Graceful degradation“.
- Wenn es gezielt auf hohe Verfügbarkeit, Zuverlässigkeit und gute Wartbarkeit hin entworfen wurde, spricht man von einem „Dependable System“.
- Nur wenn es fähig ist, auf interne und externe Störfaktoren zu reagieren, Probleme entsprechend zu beheben und den Regelbetrieb danach fortzusetzen, handelt es sich um ein „Resilientes“ System.

(vgl. [1]) IBM definiert Software-Resilienz als Fähigkeit, die Auswirkungen eines Problems auf einen oder mehrere Teile des System zu absorbieren und dabei ein akzeptables Service Level zu garantieren. (vgl [2]) Um zu verstehen, wie sich dieser Ansatz von den bisherigen Methodiken unterscheidet, um Anwendungen widerstandsfähig und robust zu machen, muss man die Definition von Verfügbarkeit (engl. Availability) genauer betrachten:

$$Verfügbarkeit = \frac{MTTF}{MTTF + MTTR} \tag{1}$$

wobei gilt:
$MTTF$ = Mean Time To Failure
(mittlere Zeit bis zum Auftreten eines Fehlers)
$MTTR$ = Mean Time To Recovery
(mittlere Zeit bis zur Rückkehr zum Normalbetrieb)
(vgl. [3]) Für gewöhnlich wird Software unter der optimistischen Annahme entwickelt, dass eine Maximierung des MTTF-Faktors ausreichend ist, um höchste Verfügbarkeit gewährleisten zu können, was die Mean Time To Recovery zweitrangig macht. Resiliente Softwareentwicklung dagegen rechnet mit dem Versagen einer Hardware- oder Softwarekomponente und versucht durch die Optimierung des MTTR-Faktors die Konsequenzen eines Fehlverhaltens zu minimieren. Ein Fehlerfall sollte möglichst frühzeitig erkannt und entsprechende Maßnahmen zum Abschwächen der Auswirkungen getroffen werden. Das Ziel ist eine garantierte Rückkehr zum Regelbetrieb in möglichst kurzer Zeit. Notwendig wird dieser Ansatz durch den gestiegenen Grad der Komplexität und Verteilung moderner Systeme: Netzwerke werden redundant angelegt, Rechenleistung wird bei externen Dienstleistern eingekauft, Daten liegen in Clouds und bei externen Content Delivery Networks. Die Idee an sich ist dabei nicht grundlegend neu. Bereits 1987 erwähnte der amerikanische Informatiker und Mathematiker Leslie Lamport in einer E-Mail: „A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.“ [4] Auch Werner Vogels, seines Zeichens Chief Technology Officer und Vizepräsident bei Amazon, erwähnte einst in einem Interview „everything fails all the time“ [5] Doch warum sind nun besonders verteilte Systeme gefährdet? Einige Gründe dafür lassen sich in Rotem-Gal-Oz's Werk „Fallacies of Distributed Computing Explained“ in dem er insgesamt acht häufige Irrtümer von verteilten Systemen beschreibt, finden:

1) Das Netzwerk ist verlässlich.
2) Es existiert keine Latenz.
3) Die Bandbreite ist unbegrenzt.
4) Das Netzwerk ist sicher.
5) Die Topologie wird sich niemals ändern.
6) Es gibt nur einen Administrator.
7) Es entstehen keine Transportkosten.
8) Das Netzwerk ist homogen.

(s.a. [6]) Diese Irrtümer können insbesondere in heutigen Produktivsystemen kritische Fehlerzustände begünstigen. Fehler breiten sich schnell intern aus und verursachen hohe Ausfallszeiten und damit auch Folgekosten. Nur mit einem funktionstüchtigen System kann Geld verdient werden. Stillstand oder eine Störung der direkten Kundenkommunikation gefährden die Zufriedenheit und Reputation eines Unternehmens. Dabei spielt es auch keine Rolle, ob der potentielle Störfaktor aus dem eigenen Netz kommt oder durch einen externen Angreifer gegeben ist. Über allem steht das große Ziel, dass ein internes Problem nach außen hin für den Nutzer nicht spürbar ist. Software-Resilienz ist damit die Antwort auf die gestiegenen Anforderungen des heutigen Internets.

II. Hauptteil

Die Entwicklung resilienter Anwendungen ist ein kontinuierlicher Prozess, der sich von der Planung über die Entwicklung bis hin zur Wartung erstreckt und konsequentes Testen erfordert. Unerlässlich hierfür ist ein grundlegendes Verständnis davon, wie sich Fehlerzustände konkret äußern können. Ein wichtiges Maß dabei sind:

A. Die fünf Fehlertypen nach Andrew S. Tanenbaum

Die Klassifizierung ist ein Standard in der Systemarchitektur und unterscheidet Fehler wie folgt:

1) Absturzfehler (engl. Crash Failure) - Das System antwortet permanent nicht mehr.
2) Auslassungsfehler (engl. Omission Failure) - Das System antwortet auf einzelne Anfragen nicht mehr; dabei bleibt vorerst ungeklärt ob es die Anfragen nicht bekommen hat oder aus einem anderen Grund nicht antwortet.
3) Antwortzeitfehler (engl. Timing Failure) - Die Antwortzeit des Systems liegt außerhalb eines festgelegten Zeitintervalls.
4) Antwortfehler (engl. Response Failure) - Die Antwort des Systems ist fehlerhaft.
5) Byzantinischer / zufälliger Fehler (engl. Byzantine Failure) - Das System reagiert nichtdeterministisch, es gibt zu zufälligen Zeiten zufällige Antworten.

[7] Es ist daher nicht ausreichend, ein System ausschließlich gegen Absturzfehler abzusichern. Resiliente Anwendungen zeichnen sich durch Ausfallsicherheit gegenüber jeglichen Fehlertypen aus.

B. Entwicklung von resilienter Software

Resilienz ist ein kritisches Qualitätsziel, das von Anfang an verfolgt und konsequent auf Einhaltung geprüft werden muss. Die Grundlagen liegen bereits in der Entwurfsphase der Softwareentwicklung. Dabei gibt es keine allgemeingültigen Lösungen, die exakten Resilienzmechanismen müssen für jeden Fall einzeln konkret ausgearbeitet werden. Zwei wichtige Prinzipien dabei, Bulkhead- und Resilience-Muster, werden im Folgenden näher erläutert.

1) Das Bulkhead-Muster: „Bulkhead“ (aus dem Englischen für die Schottwand bei Schiffen) beschreibt ein Vorgehen beim Software Design, bei dem einzelne Module logisch voneinander getrennt („abgeschottet“) werden. Ähnlich dem „Separation of concerns“ als Anliegen einer Komponentenarchitektur beim Software-Engineering, geht es darum, eine Abstraktion durchzuführen und Funktionen einer Software zu entkoppeln. Damit ist sichergestellt, dass die Komponenten selbstständig agieren können und voneinander isoliert sind. Eine einzelne Komponente ist zudem weniger komplex und damit leichter wart- und austauschbar. Die Einhaltung dieses Schemas sorgt dafür, dass ein Fehler sich im Ernstfall nicht unkontrolliert auf andere Module der Software fortpflanzen kann und sich damit nicht zu einem kaskadierenden Fehler weiterentwickelt. Damit wird sie zu einer essentiellen Grundlage von resilienter Software. E. Evans beschreibt in seinem Buch „Domain-Driven Design“ einige Vorgehensweisen und konkrete Muster, gibt allerdings auch zu bedenken, dass das reine Kennen eines Musters noch lange nicht garantiere, dass man es auch gut umsetzen könne. (vgl. [8])

2) Resilience-Muster: Neben der Isolation der einzelnen Module voneinander, müssen Resilience-Muster als spezifische Reaktionspläne auf einen Fehlerzustand ausgearbeitet werden. Hilfreich kann dabei eine Muster-Taxonomie sein, wie sie im Folgenden gezeigt wird. (siehe 1)

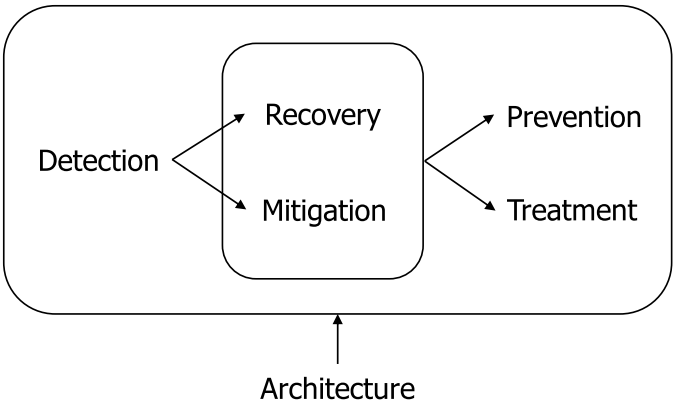

Abbildung 1. Eine einfache Taxonomie für Resilience-Muster nach U. Friedrichsen [9]

- Error Detection - Initiale Erkennung eines internen Fehlers, bevor er nach außen sichtbar wird.
- Error Recovery - Vollständige Behebung eines Fehlers (falls genug Zeit und Wissen vorhanden sind).
- Error Mitigation - Eindämmen bzw. Abschwächen der Konsequenzen des Fehlers, ohne ihn zu beheben.
- Error Prevention - Die Wahrscheinlichkeit eines Fehlers reduzieren.
- Error Treatment - Die Ursache eines entstandenen Fehlers beheben.
- Architecture - Ergänzende strukturelle und verhaltensorientierte Muster, die die Umsetzung von Mustern aus den zuvor genannten Domänen ermöglichen bzw. unterstützen.

(vgl. [9]) Anhand dieser Taxonomie können Resilience-Muster erstellt und auf Vollständigkeit geprüft werden. Jeder Reaktionsplan enthält dabei die initiale Error Detection, entscheidet danach, ob man den Fehler vollständig beheben kann (Recovery) oder nur abmildern soll (Mitigation) und schließlich, ob Gegenmaßnahmen getroffen werden, die ein zukünftiges Eintreten des gleichen Fehlers verhindern (Prevention). Die Anzahl der angewandten Resilience-Muster ist dabei stark vom jeweiligen Einsatzzweck abhängig; wichtig ist, das korrekte Maß unter Berücksichtigung der Use-Cases und des Wissens über potentielle und häufige Fehlerquellen zu finden. Je komplexer die gewählte Strategie ausfällt, desto komplexer wird auch die Anwendung selbst; damit steigen auch die Wartungskosten und die Wahrscheinlichkeit von weiteren Schwachstellen.

C. Testen von Software-Resilienz in der Theorie

Will man nun eine Software bezüglich ihrer Resilienzfähigkeit auf die Probe stellen, so gibt es viele mögliche Vorgehensweisen, die sich je nach Anwendungsfall mehr oder weniger anbieten. Handelt es sich um eine Software im kleineren Maßstab mit wenig Abhängigkeiten von Ressourcen, so bieten sich Tests auf Hardware-Integrität an. Geht es dagegen um eine ganze Systemlandschaft mit hoher Ressourcenabhängigkeit, wird auch der Testentwurf komplexer.

1) Gewährleistung der Hardware-Integrität: Software-Resilienz beginnt bei der korrekten Funktionsweise der Hardware. Nur wenn man garantieren kann, dass die Integrität und Konsistenz des Host-Systems gewährleistet ist, kann man auch sicher sein, dass die Anwendung an sich und alle weiteren Resilienzverfahren gemäß ihrer Implementierung korrekt ausgeführt werden. Ziel ist es, temporäre oder permanente Probleme in der Hardware zu erkennen, die unsere Ausführung beeinflussen könnten. Persistente Fehler sind besonders kritisch, da ihr Auftreten oftmals subtil ist und über einen längeren Zeitraum verborgen bleiben kann. Initial sollte ein Hardware-Test T durchgeführt werden, damit sichergestellt ist, dass das Host-System von Anfang an voll funktionsfähig gewesen ist. Das Ergebnis wird zum späteren Abgleich abgespeichert. Nach dem Programmablauf P wird der Hardware-Test ein zweites Mal durchgeführt und die Ergebnisse miteinander verglichen. Stimmen sie überein, kann davon ausgegangen werden, dass sich kein permanenter Fehler in der Hardware entwickelt hat. Eine reguläre Ausführungssequenz nach diesem Schema ist in Abbildung 2 dargestellt. Ein weiterer

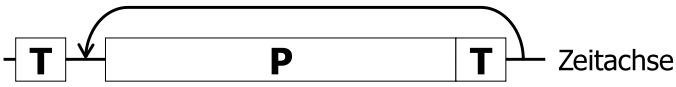

Abbildung 2. Abgleich zweier Hardware-Tests, um persistente Fehler zu entdecken

Fehlerfall kann darüber hinaus vorliegen, wenn ein Fehler nur kurzzeitig während der Programmlaufzeit aufgetreten ist. Diese Verletzung der Hardware-Integrität würde damit von dem zweiten Hardware-Test nicht mehr abgedeckt werden und somit unentdeckt bleiben. Damit temporäre Fehler innerhalb der Ausführungssequenz erkannt werden können, muss der Programmablauf ein weiteres Mal durchgeführt und die Ergebnisse miteinander verglichen werden, wie in Abbildung 3 dargestellt ist. Die doppelte Ausführung der Prozedur ist sehr

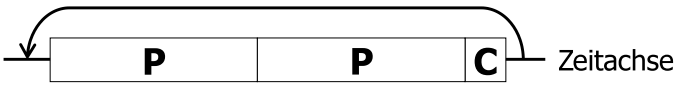

Abbildung 3. Abgleich des Programmablaufs, um temporäre Fehler zu erkennen

kostspielig und erhöht den Anwendungs-Overhead enorm. Ein solcher Test ist daher nicht für jeden Einsatzzweck geeignet. Seine Notwendigkeit muss abgewogen werden. Abbildung 4 kombiniert die Vorzüge beider Schemata zu einem Modell, das sowohl die Erkennung temporärer als auch persistenter Hardware-Integritätsverletzungen abdeckt. (vgl. [1]) Mit

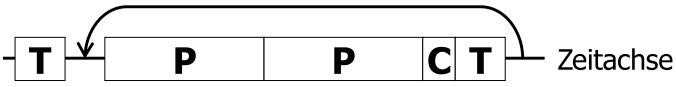

Abbildung 4. Kombinierter Test auf permanente und temporäre Hardware Fehler

diesem Modell lassen sich insbesondere kleinere Anwendungen sehr einfach auf ihre korrekte Funktionsweise überprüfen. Komplizierter wird es bei größeren Unternehmen mit einem höheren Verteilungsgrad und mit einer unverzichtbaren Kundenkommunikation. Wenn aus einem einzelnen System eine ganze Systemlandschaft wird, wachsen auch die Mittel, die nötig sind, um die Resilienz vollständig durch Tests abzudecken.

2) Empfehlungen der Amazon Web Services: Eine weitere Möglichkeit, die Voraussetzungen für Software-Resilienz zu optimieren, ist ein teilweiser oder kompletter Umzug der Anwendung zu einem externen Cloud Computing Dienst. Je nach benötigtem Leistungsvolumen steigert das die monatlichen Ausgaben, die zum Betrieb notwendig sind; es entfallen allerdings Betriebs-, Wartungs- und Erweiterungskosten der eigenen Hardware. Insbesondere bei kleineren und mittleren Unternehmen mit nur eingeschränkten Mitteln, die einer umfangreich angelegten Redundanz der Hardware im Weg stehen könnten, kann sich dieser Schritt als sehr sinnvoll erweisen. Zudem lässt sich der gebuchte Umfang leicht ausweiten oder nach Bedarf um zusätzliche Dienste erweitern. Das Unternehmen bleibt so möglichst flexibel, was die Hardware betrifft, und muss sich nur um die Software kümmern. Für Kunden ihrer Web Services bietet Amazon in Form des AWS Well-Architected Frameworks eine Orientierungshilfe zum Aufsetzen des eigenen Webdienstes an. Teil davon sind auch die folgenden Empfehlungen, wie Tests auf Zuverlässigkeit durchgeführt werden sollten.

- Anlegen und Nutzen von Playbooks, um Fehler zu analysieren:
  Playbooks dienen als Anleitung, um Probleme anhand ihres Fehlerbildes aufzuspüren. In wenigen Schritten kann so ein Fehler in dokumentierte Teilfaktoren zerlegt werden, bis die Ursache gefunden und behoben wurde. Playbooks sollten regelmäßig erweitert und so gesichert werden, dass sie insbesondere im Falle eines Ausfalls verfügbar sind, notfalls in gedruckter Form.
- Ursachenanalysen durchführen:
  Im Falle eines Ausfalls, besonders einem, der nach außen hin sichtbar wurde und Auswirkungen auf die Qualität der Anwendung hatte, sollte dieser im Nachhinein analysiert werden. Für die Betrachtung wichtig sind dabei vor allem die direkten Ursachen, begünstigenden Faktoren und getroffenen Gegenmaßnahmen. Diese Informationen helfen dabei, nötige Änderungen an der Anwendung vorzunehmen, damit ein gleiches Fehlerbild nicht wiederholt auftreten kann.

- Funktionale Anforderungen überprüfen:
  Diese können über Unit Tests und Integration auf die korrekte und benötigte Funktionsweise getestet werden.
- Belastungs- und Performancetests:
  In einem dem Produktivsystem ähnlichen Umfeld sollten Belastungs- und Performancetests durchgeführt werden. Dadurch ist erkennbar, ob die geforderten Belastungsgrenzen eingehalten werden, und wie sich ein durch Überlastung hervorgerufenes Fehlverhalten äußert. Außerdem sollte geprüft werden, ob Antwortzeiten und benötigte Rechenleistung den Anforderungen entsprechen.
- Bewusstes Erzeugen von Fehlern:
  In Vorproduktions- und Produktionsumgebungen sollten regelmäßig bewusst Fehler erzeugt werden. Diese Methodik nennt sich „Chaos Engineering“ und wird in II-C5 weiter ausgeführt.
- Regelmäßig Gamedays veranstalten:
  Mitarbeiter, die in für die Resilienz zuständigen Bereichen angestellt sind, sollten regelmäßig an Gamedays teilnehmen. Der genaue Ablauf wird im folgenden Kapitel II-C3 näher besprochen. Gamedays können zudem genutzt werden, um an Play- und Runbooks zu arbeiten.

(vgl. [10])

3) Gamedays: Die Veranstaltung sogenannter „Chaos Gamedays“ ist eine teambildende Methodik, die es erlaubt, Entwickler auf spielerische Art und Weise für das Thema Resilienz zu sensibilisieren und mit dem System vertraut zu machen. Diese Maßnahme vermittelt eine neue Sichtweise auf die zu Grunde liegenden Mechanismen und Datenflüsse. Sind Fehler normalerweise ein Übel, das gerne außen vor gelassen wird und dessen Auftreten für Panik und Überforderung sorgen kann, so reagiert geschultes Personal entspannter und kann im Ernstfall die richtigen Schlüsse ziehen. Die Durchführung läuft wie folgt ab:
Ein „Master of Disaster“ bereitet einen Fehlerfall vor. Das Team weiß nicht, **was**, nur, **dass** etwas passieren wird. Ein Mitglied des Teams übernimmt die Rolle des Ersthelfers. Seine Aufgabe ist es, andere Teammitglieder zu alarmieren und in die Fehlersuche miteinzubeziehen. Die Herausforderung gilt als bestanden, wenn der Fehler in weniger als 75% der veranschlagten Zeit gefunden und behoben wurde. Ist das Team nicht in der Lage, den Fehler zu lokalisieren, kann der Master of Disaster die Problemsituation bis zum Totalausfall immer weiter zuspitzen, damit ein Fehlverhalten nach außen hin sichtbarer wird. In diesem Fall kann der Test weitere Anhaltspunkte über Mängel im Monitoring geben. Nachdem der Test beendet und alle Probleme vom Master of Disaster rückgängig gemacht wurden, soll vom Team ein „Post Mortem“ durchgeführt werden. In dieser finalen Schadensanalyse werden die Ereignisse zusammengefasst: Was passiert ist, wie reagiert wurde und inwieweit die tatsächlichen Beobachtungen von den Erwartungen abgewichen sind. Insbesondere Lücken im Monitoring und Ideen, wie das System gegen das Fehlerszenario gehärtet werden kann, sind wichtige Schlüsselaspekte. Wurden die gefundenen Mängel beseitigt, so kann die korrekte Funktionsweise über eine Wiederholung des Fehlerszenarios validiert werden. Gamedays sollten regelmäßig mit steigendem Komplexitätsgrad abgehalten werden. Erzielbare Langzeiteffekte auf das Team, die sich dabei einstellen, sind ein wichtiger Faktor. Entwickler erkennen wiederkehrende Muster und beginnen resilient zu denken. Eine grundlegende Erhöhung der Robustheit neuer Anwendungskomponenten ist die Folge. Resilienz und Monitoring werden von Anfang an Bestandteil des Designs. (vgl. [11])

4) DevOps & Continuous Integration: Die Wortneuschöpfung „DevOps“ setzt sich aus den Wörtern „Development“ und „IT Operations“ zusammen. Im Wesentlichen beschreibt sie einen Prozessverbesserungsansatz, bei dem die Bereiche Entwicklung, Betrieb und Qualitätssicherung zusammenwachsen. Das ermöglicht eine effizientere und effektivere Zusammenarbeit zwischen sonst getrennten Teams und baut Kommunikationsbarrieren ab. (vgl. [12]) Insbesondere für das Ziel eines resilienten Systems ist dieser Ansatz sinnvoll, da Entwickler häufig isoliert arbeiten und nicht mit den Auswirkungen ihrer Arbeit konfrontiert werden. Durch das direkte Feedback aus dem Betrieb können sie so eine völlig neue Sichtweise auf die Anwendung bekommen. Hier zeigt sich auch ein weiterer Aspekt des DevOps Modells: Es ist eng mit den Prinzipien der agilen Softwareentwicklung verknüpft. Engere Kommunikation mit dem Betrieb und den Kunden bedeutet schnellere Reaktion auf Wünsche und mögliche Probleme. (vgl. [13])
Ein weiteres Verfahren, das sich häufig im Kontext von DevOps finden lässt, ist die sogenannte „Continuous Integration“. Entwickler sollen Codeänderungen und neue Funktionen regelmäßig in einem zentralen Repository zusammenführen, sie „kontinuierlich integrieren“. Es ist ein Schritt weg von seltenen und festen Releases hin zu einer rollenden Auslieferung der aktuellsten Änderungen (auch „Continuous Deployment“ genannt). Das Vorgehen hat dabei viele Vorteile, besonders für die Entwicklung resilienter Anwendungen. Dadurch, dass Entwickler dazu angehalten sind, ihre Änderungen häufig zusammenzuführen, können sich weniger langfristige Bugs ohne Korrekturen ansammeln. Der Prozess des Zusammenführens wird ebenfalls erleichtert, da Entwickler nicht mehr isoliert arbeiten und über die Änderungen der Kollegen Bescheid wissen. So können Kollisionen sehr simpel vermieden werden, und es kommt zu keinen Fehlern durch Inkompatibilitäten zwischen den einzelnen Versionen. Einer der größten Vorteile von Continuous Integration ist die Möglichkeit, mit jeder neuen Änderung die komplette Anwendung auf Fehler und Einhaltung der Anforderungen testen zu können. So fallen Probleme direkt auf und lassen sich leicht innerhalb der Änderung nachvollziehen. Es kann entschieden werden, ob der Fehler direkt mit einem Patch behoben wird oder ob die Änderung vorerst zurückgezogen und überarbeitet werden soll. (vgl. [14])

5) Chaos Engineering: Als Chaos Engineering wird das Experimentieren innerhalb eines Systems bezeichnet, um dessen Widerstandsfähigkeit in anspruchsvollen Produktivsituationen auf die Probe zu stellen. Selbst in bekannten Anwendungen und Systemen kann sich potenziell unberechenbares Verhalten verbergen, das sich nur in sehr seltenen Grenzfällen

äußert. Dieses Verhalten soll gezielt provoziert werden, um so aus dem entstehenden Chaos lernen zu können.

1) Einen Grundzustand definieren, in dem das System sich normal verhält und keine Fehler aufweist.
2) Zwei vergleichbare Gruppen erzeugen: Die Kontrollgruppe und die Testgruppe. Beide befinden sich im Grundzustand.
3) Fehler innerhalb der Testgruppe streuen. Es sollte sich um realistische Ereignisse handeln wie z.B. Verbindungsprobleme, Server Crashes oder Festplattenausfälle. Insbesondere, wenn das Eintreten häufig ist oder die erwarteten Konsequenzen schwerwiegend sind, sollte der Testlauf hohe Priorität haben.
4) Kontroll- und Testgruppe vergleichen. Sollten Unterschiede in der Ausführung erkennbar sein, wurde eine Schwachstelle erkannt, die weiter untersucht werden muss.

Chaos Engineering interessiert sich dabei hauptsächlich dafür, **ob** das System funktioniert, nicht **wie**. Komplexes Monitoring ist in dieser Hinsicht nicht nötig, solange garantiert werden kann, dass messbare Ausgaben des Systems nicht durch die Experimente vom Normalzustand abweichen. Um die Korrektheit der Ergebnisse garantieren zu können, sollten die Testläufe immer in der realen Produktivumgebung durchgeführt werden. Nur so ist sichergestellt, dass das Verhalten des Systems authentisch ist und nicht durch potenzielle Ungenauigkeiten einer Simulation abweicht. Der Chaos Engineer muss allerdings in jedem Fall dafür Sorge tragen, dass mögliche Folgen eines Experiments die Qualität des Dienstes höchstens kurzzeitig mindern. Wurden Testläufe entwickelt, sollten diese kontinuierlich und automatisiert durchgeführt werden. Eine praktische Anwendung von Chaos Engineering wird in II-D1 betrachtet. (vgl. [15])

6) Entwicklung von Resilienztests nach dem Leitfaden von IBM: Deutlich strukturierter und geplanter verfährt dagegen IBM bei seinen Resilienztests: Als eines der führenden Unternehmen im Bereich der professionellen Hardware- und Softwarelösungen haben sie einen Leitfaden herausgegeben. Über sieben Schritte verteilt kann damit ein ausführlicher Plan zum Testen der Resilienz und Einspeisen der nötigen Änderungen für den eigenen Anwendungsfall erzeugt werden. Personal und Kunden bekommen damit ein klar definiertes Vorgehen vorgesetzt, mit dem sich im Zusammenspiel mit der benötigten Testsoftware leicht ein geeigneter persönlicher Ablauf ausarbeiten lässt. Grundvoraussetzung für die Resilienz des gesamten Systems ist dabei, dass jede einzelne Unterkomponente wie Firewalls, Switches, Betriebssysteme und Software in sich selbst bereits ebenfalls resilient sind.

Schritt 1: Testfälle entwickeln
Ziel ist es, herauszufinden, wie ein einzeln betrachteter Teilausschnitt des Systems auf einen auftretenden Fehler, insbesondere auf ein Problem mit einer benötigten Ressource, reagiert. Für jede identifizierte Ressource sollten ein oder mehrere Testfälle ausgearbeitet werden. Solch ein Testfall könnte zum Beispiel der Ausfall einer Datenbank sein. Es ist wichtig, dass der ausführende Resilienz-Beauftragte ein grundlegendes Verständnis der Komponenten und deren interner Interaktionen hat, damit die Vollständigkeit der Betrachtung garantiert ist. Sequenz- und Flussdiagramme können dazu ebenfalls hilfreich sein. Insbesondere nicht-funktionale Anforderungen wie Antwortzeit, Durchsatz und Verfügbarkeit sollten dabei berücksichtigt werden, da diese ein kritisches Maß für eine akzeptable Herabsetzung des Servicelevels im Rahmen der Graceful Degradation darstellen. Als Nächstes sollten anhand der erkannten funktionalen und nicht-funktionalen Anforderungen Testfälle ausgearbeitet werden. Zentrale Fragen könnten dabei sein, wie sich der komplette Ausfall oder ein Performance-Problem einer einzelnen Ressource auf das System auswirken könnte, und ob es die nicht-funktionalen Anforderungen beeinflusst. Ist die Verfügbarkeit einer bestimmten Ressource kritisch für das Gesamtergebnis? Wie würde sich eine beobachtete Komponente im Falle einer verzögerten Bereitstellung der Ressource verhalten? Die Verantwortlichen der jeweiligen System-Teilbereiche sollten ebenfalls zu Rat gezogen werden. Damit Unterschiede korrekt messbar sind, muss vor der Durchführung eines Testlaufs sichergestellt werden, dass der Umfang des Monitorings ausreichend ist. Die Ergebnisse sollten wie folgt festgehalten werden:

1. Kritikalität
2. Problem
   a) Auswirkungen
      1 Direkt betroffene Anfragen
         · Nicht-funktionale Anforderungen
      2. Indirekt betroffene Anfragen
         · Nicht-funktionale Anforderungen
   b) Beschreibung des Resilienz-Tests
      1. Beobachtetes Verhalten
      2. (Resultierende Probleme)
         · Nötige Änderungen zur Behebung des Problems

Schritt 2: Testfälle ausführen
Damit eine Resilienzanpassung durchgeführt werden kann, ist es zwingend notwendig, den zuvor definierten Test ablaufen zu lassen. Mehrmaliges Wiederholen dieses Schrittes erhöht die Zuversicht, dass es sich um ein konstantes Verhalten handelt.
Schritt 3: Anwendungsverhalten überwachen
Um sicherzustellen, dass Anpassungen an unserem Betrachtungsbereich keine nicht akzeptablen Auswirkungen auf andere Bereiche und nicht-funktionale Anforderungen hat, ist es notwendig, detailliertes Monitoring durchzuführen. Die Resultate können als Entscheidungshilfe verwendet werden, ob der gewählte Ansatz zur Behebung des Resilienzproblems den Anforderungen entspricht. Falls nicht, kann mit den Daten weitergehend eine Ursachenanalyse durchgeführt werden.
Schritt 4: Zusätzliche Resilienzprobleme identifizieren
Während des Testdurchlaufs können sich weitere Probleme aus der Manipulation einer Ressource ergeben, die ebenfalls problematisch werden könnten. Insbesondere, wenn diese Probleme auch, nachdem die betroffene Ressource in den

ursprünglichen Zustand wiederhergestellt wurde, weiterhin bestehen, ist Handlungsbedarf nötig. Häufig ergeben sich diese Folgefehler aus Defekten einer verwendeten Software oder aus fehlerhaften Konfigurationen. Es ist daher wichtig, dass sich die verwendete Software auf dem aktuellsten Stand befindet und nicht mit einer Standardkonfiguration betrieben wird.
Schritt 5: Resilienzparameter identifizieren
Die Wahrscheinlichkeit, bei der Ausführung der Resilienztests auf Veränderungen im Bereich der nicht-funktionalen Anforderungen zu treffen, ist hoch. Selbst nachdem ein Problem behoben ist, kann es beispielsweise weiterhin zu Verzögerungen kommen. Etwaige Varianzen sollen als Resilienzparameter identifiziert und dokumentiert werden.
Schritt 6: Resilienzparameter anwenden
Bei der Einspeisung der Änderungen sollten die generellen Richtlinien nicht außer Acht gelassen werden. Wichtig ist, die Konfiguration weiterhin so eingerichtet zu erhalten, dass die Anzahl der gleichzeitigen Anfragen an das Zielsystem minimal bleibt.
Schritt 7: Den gesamten Prozess wiederholen
Nachdem die Resilienzparameter identifiziert und die nötigen Anpassungen in das System eingespeist wurden, ist eine erneute Durchführung der Resilienztests notwendig, da sonst die Bestätigung der korrekten Funktionsweise fehlt. Erst wenn dabei keine Probleme mehr auftauchen und die Feinjustierung erfolgt ist, ist dieser iterative Prozess (siehe 5) abgeschlossen und das System erfolgreich gehärtet.

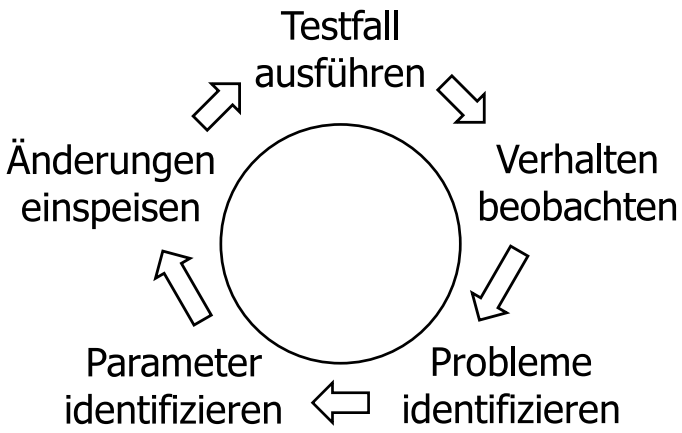

Abbildung 5. Iterativer Prozess der Feinjustierung der Resilienz

(vgl. [2])

D. Testen von Software-Resilienz in der Praxis

1) Die Netflix Simian Army: Netflix ist mit 182,26 Millionen zahlenden Kunden (Stand 1. Quartal 2020 [16]) der weltweit größte Anbieter von bezahltem Videostreaming und macht damit einen beträchtlichen Teil des weltweiten Datenverkehrs aus. Um die Kunden mit den regional annähernd 6000 Filmen und Serien erreichen zu können, muss die Verfügbarkeit trotz der hohen Last zu jedem Zeitpunkt gewährleistet sein. Damit dieses Ziel erreicht werden kann, setzt Netflix neben Redundanz in der Hardware durch die Aufteilung auf regionale Rechenzentren auf ein Konzept, das sie selbst als ihre „Simian Army“ bezeichnen. Digitale Störprozesse wüten auf den Servern und verursachen regelmäßig bewusst herbeigeführte Zwischenfälle, ähnlich dem Verhalten einer tobenden Affenhorde. Das erste Tool solcher Art war der „Chaos Monkey“, ein Programm, das geschrieben wurde, um in einem bewachten Umfeld zufällig Prozesse zu beenden und die Auswirkungen auf das gesamte System zu beobachten. Die daraus resultierenden Ergebnisse wurden als Grundlage für die automatische Generierung von Resilienzmustern genutzt. Auf Basis dieses Erfolgs wurden noch einige weitere Störprozesse entwickelt:

- Der „Latency Monkey“ erzeugt künstliche Auszeiten in der Client-Server Kommunikation. Indem sehr große Auszeiten simuliert werden, kann zudem der Ausfall von einzelnen oder mehreren Komponenten getestet werden, während sie für den Rest des Systems verfügbar bleibt. Das ist besonders interessant, um das Verhalten eines neuen Services zu beobachten, dem plötzlich die Abhängigkeiten fehlen.
- Der „Conformity Monkey“ findet nicht den Anforderungen konforme Instanzen, trennt diese vom System und benachrichtigt eine zuständige Person.
- Der „Doctor Monkey“ überwacht Zustandsparameter einer Distanz. Dies können zum Beispiel die CPU-Last oder die S.M.A.R.T.-Werte einer verbauten Festplatte sein. Sollten sich Auffälligkeiten ergeben, die auf einen schlechten Zustand hinweisen, wird die Instanz vom Netz getrennt, bis das Problem behoben wurde.
- Der „Janitor Monkey“ übernimmt die Rolle der Garbage Collection - unbenutzte Resourcen werden gesucht und freigegeben.
- Der „Security Monkey“ agiert als eine Erweiterung des Conformity Monkeys und sucht nach potenziellen Schwachstellen und Verwundbarkeiten. Dazu können sowohl falsche Konfigurationen als auch ungültige SSL- und DRM-Zertifikate zählen.
- Der „10-18 Monkey“ (kurz für „Localization-Internationalization“, oder „l10n-i18n“) erkennt Konfigurations- und Laufzeitzprobleme in Instanzen und kommt mit unterschiedlichen Sprachen und Zeichensätzen für die jeweiligen geographischen Regionen zurecht.
- Der „Chaos Gorilla“. Ähnlich dem ursprünglichen Chaos Monkey, ist seine Aufgabe, den Ausfall einer kompletten Amazon-Verfügbarkeitszone zu simulieren. Ziel ist die dynamische Verlagerung auf andere funktionale Zonen, ohne dass Nutzer etwas davon mitbekommen und ein manuelles Eingreifen durch Techniker nötig wird.

(vgl. [17]) In der schnell wachsenden Industrie der Cloud-Service-Anbieter, bei denen Langzeiterfahrungen und bewährte Strukturen noch kaum vorhanden sind, bedarf es kreativer Ansätze wie dieser, um eine hohe Güte der Resilienz garantieren zu können.

III. Ausblick

Die kontinuierliche Digitalisierung und Autonomisierung unseres Alltags ist ein unaufhaltsamer Prozess, der sich seinen Weg immer weiter in unser Leben bahnt und dabei fortlaufend komplett neue Technologiebereiche erschließt. Jüngst wurde mit dem Ausbau des Mobilfunknetzes auf 5G der Grundpfeiler für zukünftige Kommunikation gelegt. Ein Netz, das eine immer weiter steigende Anzahl an Geräten des Internets der Dinge (IOT) beherbergen wird. Kleine Helfer sorgen im Rahmen des Smarthomes dafür, dass der Haushalt durch die Automatisierung alltäglicher Aufgaben intelligent wird. Er wird somit direkt mit den neuesten Inhalten der stark expandierenden Unterhaltungsindustrie verbunden. Netflix, Amazon und Google sind drei Unternehmen, die bereits heute den stark umkämpften Markt des Medienstreamings dominieren und für einen beträchtlichen Anteil des weltweiten Datendurchsatzes sorgen; Daten, die mittels leistungsfähiger Unterseeleitungen Ländergrenzen binnen Millisekunden überwinden können und per Glasfaser bis an die Haustür transportiert werden. Übertragungen von Datenmengen im Bereich von mehreren Terabyte, die heute noch viele Tage dauern können, werden so in Zukunft innerhalb weniger Stunden abgeschlossen sein. Auch sonst wird die zunehmende Autonomisierung nicht nur das Leben unterstützen und komfortabler machen, sondern grundlegend verändern. Maschinen und Software werden überall dort eingesetzt, wo der Mensch an seine Grenzen stößt. Wissen, Präzision, Gesundheit und Ausdauer sind nur einige der Faktoren, die das menschlich Mögliche stark einschränken. Technik wird uns dabei unterstützen, diese Hürden zu überwinden und in neue Bereiche der Forschung vorzudringen. Die Raumfahrt wird mit Hilfe autonom agierender Sonden in neue Tiefen des Universums eindringen können. Der Bereich der medizinischen Informatik wird neue Meilensteine in der Entwicklung von Impfstoffen oder Behandlungsmöglichkeiten und im Ablauf von Operationen setzen. Autonome Mobilität wird das Verkehrsgeschehen der Zukunft revolutionieren und dabei Entwickler vor völlig neue Herausforderungen stellen. Damit ein Verbindungsabriss bei 200km/h keine tödlichen Folgen hat, müssen zuerst neue Zuverlässigkeitskonzepte entwickelt werden, um diese Art der Fortbewegung zu ermöglichen. Höchste Verfügbarkeit ist bei all diesen Anwendungsgebieten nicht nur wünschenswert; es ist eine absolute Grundvoraussetzung. Insbesondere in solchen Fällen, in denen nicht nur Geld, sondern auch Menschenleben und Gesundheit auf dem Spiel stehen, wird Resilienz zu einem unverzichtbaren Qualitätsmerkmal von Anwendungen. Doch auch weniger risikobehaftete Anwendungsfälle wie Social Media bedürfen ausgefeilter Pläne, um den Kontakt zum Kunden nicht zu verlieren. Ausfälle bedeuten, nicht geschäftsfähig zu sein, verursachen hohe Verluste bei den Einnahmen und gefährden die Kundenzufriedenheit. Dies alles sind Punkte, die für Wirtschaftsunternehmen besonders kritisch sind und keineswegs eintreten dürfen. Unsere Kommunikation ist global, digital und findet rund um die Uhr statt. Damit dies gewährleistet werden kann, muss Resilienz als höchstes Ziel angestrebt werden und kontinuierlich auf die korrekte Funktionsweise getestet werden. Auch die Art und Weise selbst, wie resiliente Anwendungen entwickelt und getestet werden, wird sich maßgeblich wandeln: Fortschritte im Bereich der Künstlichen Intelligenz und bei der Mustererkennung werden dabei helfen, Anwendungen besonders zuverlässig zu machen. Systeme, die sich immer weiter selbstständig auf fehlerhafte Konfigurationen untersuchen können und drohendes Fehlverhalten erkennen, noch bevor es eintritt. Aber auch Systeme, die es schaffen, sich im Ernstfall dynamisch umzubauen und durch intelligente Umverteilung der Last die Funktionsfähigkeit zu erhalten. Systeme, die aus Fehlverhalten lernen und auf dieses Wissen fortan zurückgreifen können.

IV. Conclusion

In der Welt von heute gibt es immer mehr Digitalisierung, immer mehr Autonomisierung, immer mehr internetfähige Geräte. Mit wachsenden Datendurchflussraten und steigendem Grad der Verteilung unserer Netzwerke wachsen auch die Herausforderungen an die Angestellten und die zu Grunde liegenden Systeme und Anwendungen. Resilienz hat sich dabei als wichtiges Qualitätsmerkmal einer guten Software- und Serverarchitektur herausgebildet. Sie beginnt bei der Hardware und erstreckt sich über die Software bis zu den Entwicklern selbst. Nur, wenn alle Faktoren zusammenspielen und Resilienz von Anfang an bewusst angestrebt wird, kann ein System von höchster Resilienzgüte erreicht werden. Hohe Verlässlichkeit und gute Wartbarkeit sind dabei zwei bedeutende Kernelemente von resilienten Systemen. Es bedarf fortlaufender Tests und Überwachung, um potentielle Probleme zu erkennen, bevor sie zur Bedrohung werden. Das Personal muss entsprechend geschult werden und in teambildenden Methodiken lernen, resilient zu denken und im Ernstfall die richtigen Entscheidungen zu treffen. Nur wer die Hintergründe und Ausprägungen kennt und auf Fehler nicht mit Ablehnung und Panik reagiert, kann eine unkontrollierte Ausbreitung und Chaos verhindern. Auch der Entwicklungsprozess an sich lässt sich resilienzfördernd optimieren. Das DevOps-Prinzip legt im Rahmen der agilen Softwareentwicklung die dazu nötigen Grundsteine. Kommunikationsbarrieren werden abgebaut und Entwickler individuell gefördert. Continuous Integration ermöglicht eine stetige Validierung der Anwendung und eine direkte Reaktion auf auftretende Probleme. Die möglichen Stör- und Fehlerquellen sind hierbei so vielseitig wie die Anwendungszwecke heutiger Softwarelösungen selbst. Modelle wie das des Unternehmens IBM können dabei helfen, ein System vollständig auf dessen Resilienzfähigkeiten zu testen. Allgemeingültige Komplettlösungen existieren in dieser neuen digitalen Welt, die sich täglich weiterentwickelt und expandiert, allerdings nicht. Es liegt an den eigenen Entwicklern, mit Kreativität und Einsatz ein geeignetes Resilienzmodell zu entwerfen und durch kontinuierliches Testen dessen korrekte Funktionsweise sicherzustellen.

LITERATUR

[1] Igor Schagaev, Eugene Zouev und Kaegi Thomas. *Software Design for Resilient Computer Systems*. 2nd ed. 2020. 2020. ISBN: 978-3-030-21244-5.

[2] Samir Nasser. *Software solution resiliency guidelines can help prevent bad behavior: Best practices for identifying and tuning solution resiliency parameters: When bad things happen to good systems*. IBM Developer Works, 2014. URL: https://www.ibm.com/developerworks/websphere/techjournal/1407_col_nasser/1407_col_nasser.html.

[3] Robert Hanmer. *Patterns for Fault Tolerant Software*. 1. Aufl. Wiley, 2013. ISBN: 978-0-470-31979-6.

[4] Leslie Lamport. *Distribution*. Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87. 1987. URL: https://www.microsoft.com/en-us/research/publication/distribution/.

[5] Anne Helmond. *Werner Vogels: "Everything fails all the time"*. Hrsg. von The Next Web. 2008. URL: https://thenextweb.com/2008/04/04/werner-vogels-everything-fails-all-the-time.

[6] Arnon Rotem-Gal-Oz. „Fallacies of Distributed Computing Explained". In: *Doctor Dobbs Journal* (2008). URL: https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained.

[7] Andrew S. Tanenbaum und Maarten van Steen. *Verteilte Systeme: Prinzipien und Paradigmen*. 2., aktualisierte Aufl. Pearson Studium, 2008. ISBN: 978-3-8273-7293-2.

[8] Eric Evans. *Domain-Driven design: Tackling complexity in the heart of software*. Addison-Wesley, 2011. ISBN: 978-0-321-12521-7.

[9] Uwe Friedrichsen. *Resilient Software Design: Robuste Software entwickeln*. Hrsg. von Informatik Aktuell. 2016. URL: https://www.informatik-aktuell.de/entwicklung/methoden/resilient-software-design-robuste-software-entwickeln.html.

[10] Amazon Web Services. *How do you test reliability*. 2020. URL: https://wa.aws.amazon.com/wat.question.REL_12.en.html.

[11] James Burns. *How to Get Started with Chaos: A Step-by-Step Guide to Gamedays*. Hrsg. von Lightstep. 2019. URL: https://lightstep.com/blog/get-started-with-chaos-guide-to-gamedays/.

[12] Stephan Augsten. *Was ist DevOps?* Dev Insider, 2017. URL: https://www.dev-insider.de/was-ist-devops-a-570286/.

[13] Amazon Web Services. *Was ist DevOps?* 2020. URL: https://aws.amazon.com/de/devops/what-is-devops/.

[14] Amazon Web Services. *Was ist Continuous Integration?* 2020. URL: https://aws.amazon.com/de/devops/continuous-integration/.

[15] Chaos Community. *Principles of Chaos Engineering*. 2018. URL: https://principlesofchaos.org/.

[16] Amy Watson. *Netflix: Statistics & Facts*. Hrsg. von Statista. 2020. URL: https://www.statista.com/topics/842/netflix/.

[17] Netflix. *The Netflix Simian Army*. The Netflix Tech Blog, 2011. URL: https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116?gi=47a81a28076f.