Resilienz und Fehlertoleranz in verteilten Systemen

MAKSYM DERHACHOV*, FLORIAN SCHMIDT*, and LUKAS WESTHOLT*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

Diese Arbeit untersucht Strategien und Muster zur Steigerung der Resilienz moderner verteilter Anwendungen, darunter Circuit Breaker, Retry-Muster und Fallback-Strategien. Ziel ist es, Systeme gegen Ausfälle und Spitzenbelastungen abzusichern, betriebliche Kosten zu reduzieren und Nutzerzufriedenheit zu gewährleisten. Die Analyse umfasst allgemeine Ansätze wie Redundanz und Skalierung sowie konkrete Implementierungen in Python und Beispiele aus der Industrie. Abschließend werden zentrale Erkenntnisse zusammengefasst und Handlungsempfehlungen abgeleitet.

1 EINLEITUNG UND MOTIVATION

Die rasante technologische Entwicklung im Bereich verteilter Systeme und Cloud-basierter Anwendungen stellt die Softwarearchitektur vor neue Herausforderungen. Eine zentrale Motivation für die vorliegende Arbeit ist die Notwendigkeit, Anwendungen effizient gegen Ausfälle und Störungen abzusichern, um langfristig betriebliche Kosten zu reduzieren und die Nutzerzufriedenheit zu gewährleisten. Gerade in geschäftskritischen Anwendungen können Systemausfälle erheblichen wirtschaftlichen Verlust und Vertrauensverlust bei den Nutzerinnen und Nutzern verursachen. Darüber hinaus sind in regulierten Branchen, wie der Finanz- und Gesundheitsindustrie, hohe Anforderungen an Verfügbarkeit und Stabilität zu erfüllen. Diese regulatorischen Vorgaben machen es notwendig, Resilienzstrategien nicht nur als Zusatz, sondern als integralen Bestandteil der Softwareentwicklung zu betrachten.

Ein weiteres Schlüsselmotiv ist die Weiterentwicklung technologischer Ansätze zur Resilienzsteigerung. Mit der zunehmenden Verbreitung von Microservices, Containerisierung und cloud-nativen Architekturen entstehen neue Potenziale, aber auch Herausforderungen, die innovative Muster und Techniken erfordern. Techniken wie Circuit Breaker, Retry-Muster und Fallback-Strategien sind hier entscheidend, um Fehler zu isolieren, die Stabilität des Gesamtsystems zu erhalten und negative Auswirkungen auf Benutzer zu minimieren. Denn Anwendungen müssen nicht nur unter normalen Betriebsbedingungen eine hohe Leistung erbringen, sondern auch unter Spitzenbelastungen und bei unerwarteten Systemausfällen robuste Funktionalität gewährleisten.

Die iterative Integration solcher Resilienzstrategien in den Entwicklungsprozess wird durch agile Methoden begünstigt. Agile Ansätze ermöglichen eine schrittweise Identifikation, Implementierung und Evaluierung von Resilienzanforderungen. Dadurch wird sichergestellt, dass Systeme nicht nur initial robust gestaltet werden, sondern sich kontinuierlich an neue Anforderungen und Bedrohungen anpassen können.

Die vorliegende Arbeit untersucht Strategien und Muster zur Steigerung der Resilienz und Fehlertoleranz moderner webbasierter Anwendungen. Ziel ist es, aktuelle Architekturansätze und Mechanismen systematisch einzuordnen und ihre Effektivität sowie Einsatzpotenziale zu bewerten. Dazu werden zunächst allgemeine Resilienzstrategien wie Redundanz, Partitionierung und Skalierung vorgestellt (Kapitel 2). Im weiteren Verlauf werden spezifische Patterns detailliert analysiert und hinsichtlich ihrer Vor- und Nachteile sowie Erfolgsfaktoren untersucht (Kapitel 3).

Der praktische Teil der Arbeit umfasst neben der Diskussion von Anwendungsbeispielen aus der Industrie auch Beispielimplementierungen in der Scriptsprache Python (Kapitel 4). Als Fallstudie dient Netflix, ein Vorreiter in der Entwicklung und Implementierung resilienter Architekturen. Abschließend werden die zentralen Erkenntnisse zusammengefasst (Kapitel 5). Dabei ist es vorrangig, übertragbare Handlungsempfehlungen und Entscheidungshilfen für die Auswahl geeigneter Resilienzmaßnahmen abzuleiten.

Durch diese Arbeit wird ein Beitrag zur Weiterentwicklung robuster, skalierbarer und fehlertoleranter Systeme geleistet, die den hohen Anforderungen moderner Webanwendungen gerecht werden.

2 RESILIENZ- UND FEHLERTOLERANZSTRATEGIEN

Resilienz ist definiert als die Fähigkeit, Teilausfälle zu bewältigen und die Ausführung ohne Absturz fortzusetzen.

(Begriffe und Definitionen)

3 PATTERN UND KONZEPTE FÜR RESILIENZ IN VERTEILTEN SYSTEMEN

(der Hauptteil umfasst typischerweise ca. 2/3 bis 3/4 des Texts der Arbeit.)

3.1 Circuit-Breaker

Circuit-Breaker sind Entwurfsmuster, die dazu dienen, Fehler in verteilten Systemen zu isolieren, indem sie den Zugriff auf fehlerhafte Dienste vorübergehend blockieren, um eine Überlastung zu verhindern und die Systemstabilität zu gewährleisten.

Montesi and Weber [2016] heben die Rolle von Circuit-Breaker in Microservices-Architekturen hervor, um kaskadierende Fehler zu vermeiden. Microservices sind autonome Dienste, die über Message Passing kommunizieren, während bei *Serviceorientierter Architektur* (SOA) die Komponenten einer Anwendung Teil eines einzigen ausführbaren Artefakts, eines Monolithen, sind.

Die Hauptvorteile der *Microservices-Architektur* (MSA) bestehen darin, dass Komponenten unabhängig voneinander bereitgestellt und verwaltet werden können, dass neue Versionen schrittweise eingeführt werden können, dass Komponenten mithilfe verschiedener Technologien spezialisiert werden können und dass die Skalierung effizienter durchgeführt werden kann. Der hohe Verteilungsgrad der MSA bringt jedoch auch einige Herausforderungen mit sich, wie z.B. Kommunikationsausfälle, die Überlastung von Diensten und die Notwendigkeit, Änderungen an Dienst-APIs im Laufe der Zeit zu bewältigen.

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

Ein Ausfall in einer MSA gilt als unvermeidlich und kann sich auf andere Dienste auswirken, die von dem ausgefallenen Dienst abhängig sind [Haley 2018; Montesi and Weber 2016]. Es entsteht das Konzept des kaskadierenden Ausfalls, bei dem der Ausfall eines Dienstes zu einem Dominoeffekt führen kann, der andere miteinander verbundene Dienste ebenfalls in Mitleidenschaft zieht. Um dieses Problem zu entschärfen, wird das Circuit-Breaker-Muster als Präventivmaßnahme eingeführt, um Ausfälle innerhalb einer einzelnen Komponente einzudämmen und systemweite Ausfälle zu verhindern. Der Grundgedanke hinter dem Circuit-Breaker-Pattern ist „Fail Fast“, d.h. sobald ein Dienst Anzeichen von Unreaktivität zeigt, sollten die Anrufer sofort aufhören, auf ihn zu warten, und die Situation in der Annahme behandeln, dass der Dienst möglicherweise nicht verfügbar ist.

Circuit-Breaker spielen eine entscheidende Rolle bei der Verbesserung der Stabilität und Belastbarkeit von Diensten innerhalb einer MSA. Clients sind in der Lage, die Verschwendung von Ressourcen für nicht reagierende Dienste zu vermeiden, indem sie Fehler schnell erkennen und ihre Aktionen entsprechend anpassen. Gleichzeitig wird überlasteten Diensten die Möglichkeit gegeben, sich zu erholen, indem sie laufende Aufgaben abschließen, ohne mit weiteren Anfragen bombardiert zu werden. Die praktische Umsetzung eines Circuit Breakers beinhaltet die Überwachung der Ausfallraten von Anrufen, die an einen bestimmten Dienst gerichtet sind. Wenn der Dienst Leistungsprobleme wie langsame Antworten oder häufige Fehler aufweist, wird der Circuit-Breaker-Mechanismus ausgelöst, sodass zukünftige Aufrufe sofort eine Fehlerantwort zurückgeben.

Das Circuit-Breaker-Muster lässt sich als endliche Maschine mit verschiedenen Zuständen und Übergängen darstellen, die durch eine Reihe von Parametern in einer Steuertabelle gesteuert werden. Durch die Nutzung des Leistungsschaltermusters werden die Zuverlässigkeit der Dienste und die Ausfallsicherheit des Systems in einer MSA-Umgebung erheblich verbessert. Dieser proaktive Ansatz schützt nicht nur vor kaskadierenden Ausfällen, sondern gewährleistet auch eine effiziente Ressourcennutzung und fördert den allgemeinen Zustand der miteinander verbundenen Dienste.

Deployment. In diesem Paper werden drei verschiedene Ansätze für die Implementierung von Circuit-Breaker in einer MSA diskutiert.

Der Circuit-Breaker wird in der Regel innerhalb der Clients eingesetzt, wo er Aufrufe an externe Dienste abfängt. Diese Strategie verhindert, dass Nachrichten den Zieldienst erreichen, wenn der Leistungsschalter geöffnet ist, wodurch die Notwendigkeit ähnlicher Schutzmechanismen im Dienst entfällt. Allerdings beruht dieser Ansatz auf der Annahme, dass Clients zur Verwendung der Circuit Breaker gezwungen werden können und dass sie nicht böswillig sind. Der Nachteil besteht darin, dass das Verfügbarkeitswissen eines Dienstes auf den Client beschränkt ist, sodass regelmäßige Pings erforderlich sind, um den Dienstintegritätsstatus abzufragen.

Eine alternative Implementierungsstrategie besteht darin, Circuit-Breaker auf der Seite der Dienste oder in Proxys zwischen Kunden und Diensten einzuführen. Dieser Ansatz hat seine eigenen Vor- und Nachteile. Wenn der Circuit-Breaker auf der Seite des Dienstes platziert ist, kann er den Dienst davor schützen, durch fehlgeschlagene Anfragen von mehreren Clients überfordert zu werden. Außerdem hat der dienstseitige Circuit-Breaker einen globalen Überblick über die Verfügbarkeit des Dienstes, im Gegensatz zu den client-seitigen Circuit-Breaker, die einen lokalen Überblick auf der Grundlage der Interaktionen einzelner Clients haben.

Die Platzierung von Circuit-Breaker in Proxys zwischen Clients und Diensten kann einen Mittelweg darstellen, bei dem der Proxy die Kommunikation zwischen Clients und Diensten überwachen und steuern kann. Dieser Ansatz kann von Vorteil sein, wenn die Clients nicht so verändert werden können, dass sie Circuit-Breaker enthalten, oder wenn die Dienste nicht unter der Kontrolle der Anwendungsentwickler stehen.

Montesi and Weber [2016] schlagen vor, dass praktische Anwendungen diese verschiedenen Einsatzstrategien kombinieren sollten, um die besten Ergebnisse zu erzielen. Durch die Verwendung einer Kombination aus clientseitigen, dienstseitigen und proxy-basierten Circuit-Breaker kann die Anwendung von den Vorteilen jedes Ansatzes profitieren und ihre jeweiligen Nachteile abmildern. Diese Flexibilität bei der Bereitstellung ermöglicht eine robustere und effektivere Implementierung von Circuit-Breaker, die die allgemeine Ausfallsicherheit und Verfügbarkeit des Systems verbessern kann.

Der erste Ansatz ist der dienstseitige Circuit-Breaker, bei dem der Circuit-Breaker im Dienst selbst implementiert ist. Dieser Ansatz hat den Vorteil, dass keine Annahmen über das Verhalten des Clients getroffen werden müssen, da der Dienst die volle Kontrolle über den Circuit-Breaker hat. Er erfordert jedoch eine Änderung des Quellcodes des Dienstes und verbraucht die eigenen Ressourcen des Dienstes, um den Circuit Breaker auszuführen. Ein Vorteil dieses Ansatzes ist, dass der Dienst auf aggregierte Informationen über seine eigene Reaktionsfähigkeit über alle Clients hinweg zugreifen kann.

Der zweite Ansatz ist der Proxy-Circuit-Breaker, bei dem der Circuit-Breaker in einem Proxy-Dienst eingesetzt wird, der sich zwischen den Clients und den Zieldiensten befindet. Dieser Proxy enthält für jedes Client-Dienst-Paar einen separaten Circuit-Breaker, der Anfragen nur dann zulässt, wenn sowohl der Client- als auch der Dienst-Circuit-Breaker geschlossen sind. Dieser Ansatz hat den Vorteil, dass keine Änderungen am Client- oder Dienstcode erforderlich sind, da der Proxy unabhängig konfiguriert werden kann. Außerdem schützt er sowohl Clients als auch Dienste, indem er Dienste vor zu aggressiven Clients und Clients vor fehlerhaften Diensten abschirmt. Allerdings führt der Proxy zu einem potenziellen Engpass im Netzwerk, der möglicherweise durch den Einsatz mehrerer Proxys behoben werden muss.

Insgesamt bieten die drei vorgestellten Ansätze unterschiedliche Kompromisse in Bezug auf die Komplexität der Implementierung, die Ressourcennutzung und den Grad des Schutzes, der den Clients und Diensten geboten wird.

Implementation. Eine der bekanntesten Implementierungen von Circuit-Breaker wird von der Hystrix-Bibliothek inspiriert, während die Python-Bibliothek *pybreaker* eine ähnliche Funktionalität bietet. *Pybreaker* ist eine flexible und leichtgewichtige Bibliothek, die es ermöglicht, Python-Code in eine Prozedur zu verpacken, die von einem Circuit-Breaker gesteuert wird. Diese Bibliothek hilft, den Ausfall von abhängigen Diensten oder Komponenten durch Überlastung oder Fehler zu vermeiden und unterstützt Funktionen

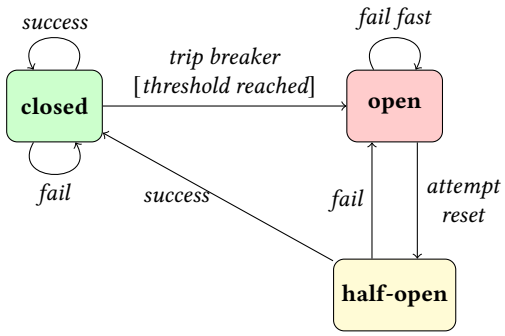

Abb. 1. Circuit Breaker Zustandsdiagramm nach [Montesi and Weber 2016].

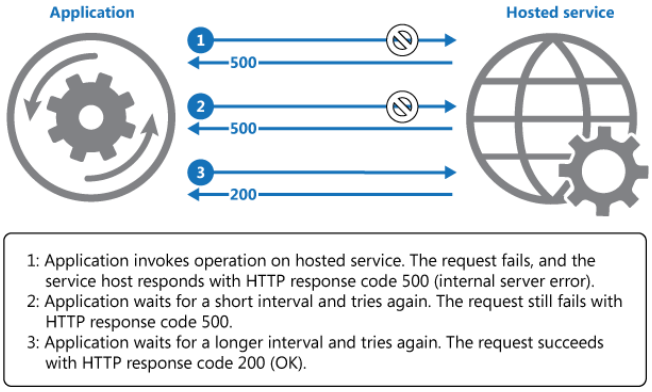

Abb. 2. Aufrufen eines Vorgangs in einem gehosteten Dienst unter Verwendung von *Retry* [Microsoft [n. d.]]

wie Fehlerzählung, Zustandsüberwachung und benutzerdefinierte Rückfallstrategien. *Pybreaker* unterstützt sowohl Client- als auch dienstseitige Circuit-Breaker.

Je nach Zustand (geschlossen, offen oder halboffen) werden unterschiedliche Aktionen ausgeführt, z.B. das Weiterleiten von Nachrichten, die Behandlung von Fehlern oder das Blockieren von Anfragen, siehe Abbildung 1.

3.2 Retry-Muster

Retry ist ein Architekturmuster, welches vorübergehende Fehler behandelt, indem es fehlgeschlagene Operationen automatisch wiederholt, oft mit exponentiellen Backoff-Strategien. Es eine bewährte Methode zur Verbesserung der Resilienz von Anwendungen, die mit entfernten Diensten oder Netzressourcen kommunizieren [Meheden et al. 2021]. Wiederholungen sind ein entscheidender Aspekt für die Ausfallsicherheit von Anwendungen, die in Containern oder in der Cloud ausgeführt werden [Haley 2018].

Retry ist eine effektive Technik zur Bewältigung vorübergehender Ausfälle in der komponentenübergreifenden Kommunikation innerhalb eines Systems. Azure und andere Cloud-Dienste und Client-SDKs bieten Wiederholungsfunktionalitäten, wie z.B. die Verwendung von EnableRetryOnFailure von Entity Framework Core, um eine Wiederholungsstrategie für Datenbankaufrufe einzurichten [Haley 2018].

Bedingungen. Für die Nutzung wird angenommen, dass die Fehlfunktion nur temporär ist. Das Muster eignet sich besonders für Szenarien, in denen Fehler aufgrund von vorübergehender Nichtverfügbarkeit oder Netzwerkkonnektivitätsproblemen auftreten [Meheden et al. 2021].

Das Beispiel in Abbildung 2 zeigt, dass bei einem fehlerhaften Aufruf einer Dienstoperation (z.B. Code 500) die Anwendung zunächst wartet und es erneut versucht. Sollte der Fehler weiterhin bestehen, wird die Wartezeit verlängert, und schließlich kann die Anfrage erfolgreich abgeschlossen werden (z.B. Code 200).

Die Wiederholungsstrategie muss jedoch an die spezifischen Geschäftsanforderungen angepasst werden. So sollten weniger kritische Anfragen eher schnell scheitern, um die Benutzererfahrung nicht zu beeinträchtigen, während bei größeren Verzögerungen die Anzahl der Wiederholungen begrenzt werden sollte. Für erfolgreiche Implementierungen müssen Fehler gründlich protokolliert und getestet werden, um potenzielle Probleme frühzeitig zu erkennen und zu beheben [Meheden et al. 2021].

Insgesamt trägt das Wiederholungsmuster dazu bei, die Fehlertoleranz und Stabilität von Anwendungen zu erhöhen, indem es vorübergehende Fehler effektiv adressiert und die Auswirkungen auf die Geschäftsprozesse minimiert.

Die Kombination von Retry mit dem Circuit Breaker-Pattern kann die Ausfallsicherheit weiter erhöhen, indem die Wiederholungsversuche nach einer bestimmten Anzahl von Fehlern gestoppt werden, sodass das System sich erholen und anders reagieren kann, z.B. auf einen zwischengespeicherten Wert zurückgreifen kann.

Vorteile.

3.3 Fallback-Strategien

3.4 Load Balancing

Load Balancing ist eine wichtige Strategie, um die Leistung und Resilienz verteilter Systeme erheblich zu steigern. Hierfür wird die anstehende Arbeitslast dynamisch auf verfügbare Server oder Ressourcen aufgeteilt.

3.4.1 Übersicht. Drei Komponenten spielen hier eine wichtige Rolle. Im Mittelpunkt dieses Verfahrens steht der Load Balancer, eine Software- oder Hardwarekomponente, welche serverseitig über mehrere Dienste verwaltet. Die zweite Komponente ist eine Gruppe von gleichartigen Service-Instanzen, welche die eigentliche Geschäftslogik beinhalten. Eingehende Anfragen werden zuerst den Load Balancer gestellt, welcher diese dann mithilfe einer geeigneten Strategie möglichst gerecht auf die Service-Instanzen aufteilt.

Zuletzt ist eine Komponente von Nöten, welche verfolgt, wie viele Service-Instanzen in der Konstellation vorhanden sind und in welchem Zustand sich diese befinden, also Service Discovery bewerkstelligt. Dies erfolgt hier Server-Side: Ein Client hat keine Kenntnis über den Aufbau, also Topologie, dieses Systems [Schöner et al. 2017]. Ob Load Balancing überhaupt Anwendung findet oder wie viele Ressourcen zur Verfügung stehen, wird nicht kommuniziert. Stattdessen geschieht jede externe Kommunikation nur mit dem Lastverteiler, welcher intern mithilfe der Registry ermittelt,

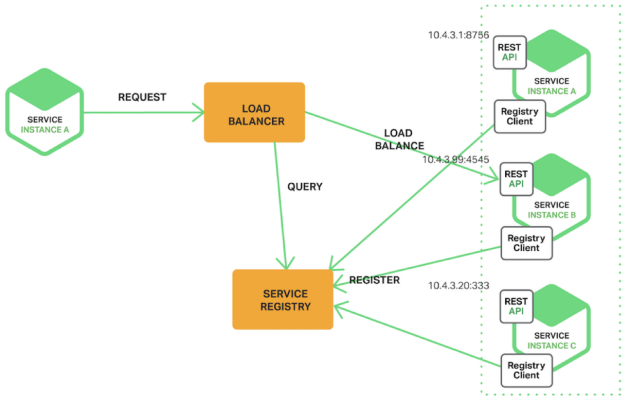

Abb. 3. Load Balancer mit drei Serviceinstanzen [Schöner et al. 2017]

welche Dienstknoten geeignet und bereit sind, die Anfrage entgegenzunehmen. Der Lastverteiler vermittelt nach Auswahl des Knotens schließlich zwischen dem Client und dem gewählten Service.

Load Balancing ist ebenso mit clientseitiger Service Discovery möglich. Hier existiert also kein zentraler Load Balancer, stattdessen ist der Client *Registry-aware* [Schöner et al. 2017] und wählt selbstständig einen passenden Serviceknoten aus, er verhält sich also selbst wie ein Load Balancer. Da die eigentliche Lastverteilungsmethodik dennoch ähnlich zur serverseitigen Discovery erfolgt und diese Methode speziell angepasste Clientsoftware benötigt, wird sie im weiteren Verlauf nicht näher erläutert.

In Abb. 3 werden die drei Komponenten mit dem typischen Datenfluss schematisch dargestellt: die Dienstinstanzen registrieren sich bei ihrem Start in der Service Registry. Erfolgt nun eine externe Anfrage an den Load Balancer, fragt dieser wiederum die Registry an, welche Dienste angemeldet sind. Mit dieser Information kann ein Dienst ausgewählt werden, und die eigentliche Anfrageverarbeitung kann stattfinden. Die Anfrage (Abb. 3, links) erfolgt nie direkt an eine der Dienstinstanzen.

3.4.2 Zustandsverfolgung der Dienste. Im vorherigen Beispiel verfolgt die Service Registry zuerst nur, ob ein Service gestartet wurde. Dafür registriert bzw. deregistriert sich der Service zum Start oder planmäßigen Ende seiner Ausführung.

Um eine effektive Lastverteilung zu ermöglichen reicht diese Information jedoch nicht aus. Wenn das Balancing ausschließlich realisiert wird, indem alle Anfragen ‚blind' auf Serviceknoten verteilt werden, könnten zwei Szenarien eintreten:

- Obwohl die gleiche Anzahl an Anfragen fair an mehrere Knoten verteilt wird, könnte ein Knoten überlastet werden. Möglich wäre dies, wenn die entsprechenden Anfragen zufällig rechenintensiver sind als die Anfragen an andere Knoten. Ebenso denkbar ist, dass auf der Serverhardware andere Dienste oder (im Fall eines virtualisierten Computers) parallel ausgeführte virtuelle Maschinen mehr Rechenzeit verbrauchen (sogenannte „Noisy Neighbours"). Analog könnte eine langsamere Netzwerkanbindung zu unterschiedlichen Kapazitäten zwischen Knoten führen.
- Ein Serviceknoten könnte versagen. Beispielsweise könnte er ansprechbar sein, aber nur noch Fehlermeldungen zurückgeben. Alternativ antwortet er womöglich gar nicht mehr, oder benötigt derart viel Zeit, dass die Anfragen zum Zeitpunkt der Fertigstellung nicht mehr von Nutzen sind.

Es ist ersichtlich, dass ein pures Aufteilen der Anfragen auf die Serviceknoten ohne Einsicht in deren Zustände nicht sinnvoll ist. Aus diesem Grund wird die Service Registry erweitert um sogenannte *Health Checks*: Jede Serviceinstanz kommuniziert ihre Auslastung und etwaige Fehler, und der Lastverteiler kann daraufhin sinnvolle Entscheidungen treffen.

3.4.3 Strategien der Lastverteilung. Die Auswahl eines Serviceknoten bei der Lastverteilung kann prinzipiell auf zwei Wege erfolgen. Bei der statischen Lastverteilung wird die aktuelle, tatsächliche Auslastung eines Knoten nicht beachtet, die Last wird stattdessen ‚blind' verteilt, und lediglich das Bereitsein der Knoten wird berücksichtigt [Mishra et al. 2020]. Beispiele statischer Strategien sind Round Robin, bei welcher anstehende Aufgaben zyklisch auf die Instanzen verteilt werden, oder das Load Balancing mit Hashing der Anfragen-IP-Adresse, bei welcher die Serviceknoten praktisch zufällig gewählt werden während mehrere Anfragen der gleichen Quelle immer auf dem gleichen Serviceknoten verarbeitet werden [GeeksforGeeks [n. d.]].

Die dynamische Lastverteilung hingegen berücksichtigt die Momentanauslastung eines Serviceknotens. Gängige Strategien sind

im vergleich redundanz/partit./skal. bewerten

- Least Connection Load Balancing: Eine anstehende Aufgabe wird dem Dienst übergeben, welcher momentan die wenigsten aktiven Netzwerkverbindungen hat.
- Least Response Time Load Balancing: Die benötigte Zeit zur Vollendung der letzten Anfragen wird pro Serviceknoten protokolliert. Der Knoten, der historisch am schnellsten ist, erhält die nächste Aufgabe.
- Resource Based Load Balancing: Alle Serviceknoten berichten ihre aktuelle Auslastung anhand generischer Merkmale wie die CPU- oder Arbeitsspeicherauslastung. Dem Knoten mit der niedrigsten Auslastung wird die nächste Aufgabe zugewiesen.

3.5 Recap

Zusammenfassend lässt sich sagen, dass der alleinige Einsatz von Containern oder einer Cloud-Infrastruktur die Ausfallsicherheit von Anwendungen nicht garantiert, was die Notwendigkeit unterstreicht, eine Wiederholungslogik zu konfigurieren und Ausfallsicherheitsfunktionen zu implementieren. Die Nutzung von Bibliotheken und das Verständnis von Resilienzmustern wie Retry und Circuit Breaker sind entscheidend für die Aufrechterhaltung der Systemverfügbarkeit und die Minimierung von Ausfallzeiten im Falle von Fehlern.

4 DISKUSSION

(Einordnung, Interpretation und Bewertung der Erkenntnisse – (nachvollziehbare, begründbare) Meinungen sind erlaubt)

5 ZUSAMMENFASSUNG UND AUSBLICK

(Überblick über die gesamte Arbeit, Rückführung auf Aussagen aus Kapitel 1 durchführen, offene Punkte als neue Forschungsfragen definieren)

Die wichtigste Erkenntnis ist, dass Entwickler aktiv Ausfallsicherheitsfunktionen in ihren Anwendungen entwerfen und implementieren müssen, selbst wenn diese in Containern oder in der Cloud ausgeführt werden. Die einfache Bereitstellung einer Anwendung in diesen Umgebungen macht sie nicht automatisch widerstandsfähig. Durch den Einsatz von Bibliotheken und Mustern wie Retry und Circuit Breaker können Entwickler zuverlässigere und fehlertolerantere Systeme erstellen [Haley 2018].

LITERATUR

GeeksforGeeks. [n. d.]. *Load Balancing Algorithms.* https://www.geeksforgeeks.org/load-balancing-algorithms/#1-static-load-balancing-algorithms

Jason Haley. 28.06.2018. Using the Retry pattern to make your cloud application more resilient. https://azure.microsoft.com/de-de/blog/using-the-retry-pattern-to-make-your-cloud-application-more-resilient/

Maria Meheden, Andrei Musat, Andrei Traciu, Andrei Viziteu, Adrian Onu, Constantin Filote, and Maria Simona Răboacă. 2021. Design Patterns and Electric Vehicle Charging Software. *Applied Sciences* 11, 1 (2021), 140. https://doi.org/10.3390/app11010140

Microsoft. [n. d.]. Retry Pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/retry

Sambit Kumar Mishra, Bibhudatta Sahoo, and Priti Paramita Parida. 2020. Load balancing in cloud computing: A big picture. *Journal of King Saud University - Computer and Information Sciences* 32, 2 (2020), 149–158. https://doi.org/10.1016/j.jksuci.2018.01.003

Fabrizio Montesi and Janine Weber. 19.09.2016. Circuit Breakers, Discovery, and API Gateways in Microservices. http://arxiv.org/pdf/1609.05830

Tom Schöner, Kai von Luck, Tim Tiedemann, Ulrike Steffens, and Stefan Sarstedt. 2017. Analyse abstrakter Architekturmodelle in verteilten Systemen. (2017).

A ANHANG 1

A.1 Übungsaufgaben

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi malesuada, quam in pulvinar varius, metus nunc fermentum urna, id sollicitudin purus odio sit amet enim. Aliquam ullamcorper eu ipsum vel mollis. Curabitur quis dictum nisl. Phasellus vel semper risus, et lacinia dolor. Integer ultricies commodo sem nec semper.

A.2 Part Two

Etiam commodo feugiat nisl pulvinar pellentesque. Etiam auctor sodales ligula, non varius nibh pulvinar semper. Suspendisse nec lectus non ipsum convallis congue hendrerit vitae sapien. Donec at laoreet eros. Vivamus non purus placerat, scelerisque diam eu, cursus ante. Etiam aliquam tortor auctor efficitur mattis.

B ANHANG 2

Nam id fermentum dui. Suspendisse sagittis tortor a nulla mollis, in pulvinar ex pretium. Sed interdum orci quis metus euismod, et sagittis enim maximus. Vestibulum gravida massa ut felis suscipit congue. Quisque mattis elit a risus ultrices commodo venenatis eget dui. Etiam sagittis eleifend elementum.

Nam interdum magna at lectus dignissim, ac dignissim lorem rhoncus. Maecenas eu arcu ac neque placerat aliquam. Nunc pulvinar massa et mattis lacinia.

C ANHANG 3

...