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, wodurch sichergestellt wird, 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

Verschiedene Strategien und Konzepte, wie Redundanz, Partitionierung und Skalierbarkeit, bieten Ansätze, um Systeme robust und flexibel zu gestalten.

2.1 Redundanz

Redundanz bezeichnet in technischen Systemen die bewusste Vervielfältigung kritischer Komponenten oder Funktionen, um die Zuverlässigkeit und Verfügbarkeit des Gesamtsystems zu erhöhen. Durch diese Mehrfachauslegung kann das System trotz des Ausfalls einzelner Elemente weiterhin ordnungsgemäß funktionieren. Dieses Prinzip findet insbesondere in sicherheitskritischen Bereichen Anwendung, um die Ausfallsicherheit zu gewährleisten. Man unterscheidet dabei zwischen den Konzepten der aktiven und passiven Redundanz, die als Architekturdesigns betrachtet werden können, sowie zwischen verschiedenen technischen Implementierungsebenen der Redundanz wie Software-, Hardware-, Daten-, Netzwerk- und geografischer Redundanz.

Aktive Redundanz. In dieser Art von Redundanz arbeiten mehrere Komponenten gleichzeitig und parallel, um dieselbe Funktion auszuführen. Falls eine der Komponenten ausfällt, übernehmen die verbleibenden Einheiten ihre Aufgaben nahtlos, ohne Unterbrechungen oder Leistungseinbußen.

Passive Redundanz. Hier bleibt die redundante Komponente in einem Standby-Zustand und wird erst aktiviert, wenn die primäre Komponente ausfällt. Im Gegensatz zur aktiven Redundanz benötigt dieser Ansatz eine kurze Umschaltzeit, um die redundante Komponente in Betrieb zu nehmen.

Während aktive und passive Redundanzen die Vorgehensweise festlegen, wie Redundanz in einem System genutzt wird, konzentrieren sich die technischen Implementierungsebenen darauf, welche

*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.

Elemente redundant sind und auf welcher Ebene die Redundanz umgesetzt wird.

Software-Redundanz. bezieht sich auf die Implementierung mehrerer gleichwertiger oder ergänzender Softwarekomponenten, die dieselbe Funktionalität erfüllen können. Dies wird häufig genutzt, um Softwarefehler oder Programmabstürze abzufangen. Ein Beispiel ist das Design redundanter Algorithmen oder die Verwendung von „N-Version Programming“, bei dem verschiedene Softwareversionen unabhängig voneinander entwickelt und parallel ausgeführt werden.

Hardware-Redundanz. beschreibt die Mehrfachvorhandenheit physischer Komponenten in einem System, um dessen Zuverlässigkeit zu erhöhen. Typische Beispiele sind Redundanz in Form von doppelten Netzteilen, CPUs oder Festplatten. RAID-Systeme (Redundant Array of Independent Disks) verwenden z.B. Hardware-Redundanz, um Datenverluste bei einem Festplattenausfall zu verhindern.

Daten-Redundanz. bezieht sich auf die mehrfach vorhandene Speicherung oder Repräsentation derselben Daten. Sie wird häufig in Datenbanksystemen oder Cloud-Umgebungen eingesetzt, um sicherzustellen, dass Daten bei einem Systemausfall oder einer Beschädigung verfügbar bleiben. Ein Beispiel ist die Verwendung von Datenreplikation oder Backup-Strategien, die die Daten an mehreren Standorten oder in mehreren Versionen speichern.

Netzwerk-Redundanz. bezeichnet die Implementierung alternativer Datenübertragungswege und -ressourcen, um sicherzustellen, dass Kommunikationsnetzwerke auch bei einem Ausfall einer Verbindung oder eines Knotens funktionsfähig bleiben. Typische Methoden umfassen redundante Switches, Router oder Glasfaserverbindungen.

Geografische Redundanz. beschreibt die Verteilung von Systemressourcen und Daten an mehreren geografisch getrennten Standorten. Dieses Konzept wird verwendet, um die Auswirkungen von Naturkatastrophen, Stromausfällen oder anderen großflächigen Störungen zu minimieren [GeeksforGeeks 2024c].

2.2 Partitionierung

Partitionierung bezeichnet den Prozess der physischen Unterteilung von Daten in kleinere, logisch zusammenhängende Einheiten (Partitionen), die unabhängig voneinander verwaltet und abgerufen werden können. Dieses Konzept wird verwendet, um die Skalierbarkeit zu verbessern, die Leistung zu optimieren, Konflikte bei Datenzugriffen zu reduzieren und mehr Flexibilität im Umgang mit Daten zu schaffen. Partitionierung ist insbesondere in umfangreichen Datenlösungen von Bedeutung, da sie es ermöglicht, Daten effizient zu organisieren und auf spezifische Nutzungsmuster abzustimmen.

Die horizontale Partitionierung. unterteilt Datensätze einer Tabelle in mehrere Partitionen, wobei jede Partition eine Untermenge der Gesamtdaten darstellt. Dies geschieht auf der Grundlage eines Partitionsschlüssels, der dazu dient, die Daten gleichmäßig zu verteilen. Jede Partition kann unabhängig gespeichert und verwaltet werden, wodurch eine hohe Skalierbarkeit und Lastverteilung gewährleistet wird. Diese Methode reduziert Konflikte bei gleichzeitigen Datenzugriffen und ist besonders nützlich in groß angelegten, verteilten Systemen.

Die vertikale Partitionierung. teilt eine Tabelle in mehrere Partitionen, indem sie Spalten oder Attribute gruppiert. Häufig verwendete Attribute werden in einer Partition gespeichert, während weniger relevante oder selten genutzte Attribute in separaten Partitionen abgelegt werden. Diese Strategie optimiert die Leistung, weil Abfragen nur auf die relevanten Daten zugreifen müssen.

Die funktionale Partitionierung. organisiert Daten basierend auf ihrer Funktion oder ihrem Zweck innerhalb eines Systems. Unterschiedliche logische Datensätze werden in separaten Partitionen abgelegt, die unabhängig voneinander verwaltet werden können.

Die RANGE-Partitionierung. teilt Daten anhand von Wertebereichen auf. Diese können auf numerischen, zeitlichen oder anderen skalierbaren Kriterien basieren. Diese Methode erleichtert die Organisation und Verwaltung der Daten erheblich. Besonders bei regelmäßigen Archivierungs- oder Löschvorgängen ist die Verwaltung von Bereichspartitionen vorteilhaft, da Partitionen eigenständig bearbeitet werden können.

Die HASH-basierte Partitionierung. verwendet eine Hash-Funktion, um Daten basierend auf einem bestimmten Feld auf Partitionen zu verteilen. Diese Methode sorgt für eine gleichmäßige Verteilung der Daten und verhindert die Entstehung von Hotspots. Sie eignet sich besonders für Systeme, die eine gleichmäßige Workload-Verteilung und optimale Ressourcennutzung erfordern.

Die Round-Robin Partitioning. verteilt Daten in zyklischer Reihenfolge gleichmäßig auf alle verfügbaren Partitionen. Sie ist eine einfache und leicht implementierbare Methode, um eine gleichmäßige Verteilung der Daten zu gewährleisten. Diese Strategie ist jedoch weniger effizient bei datenabhängigen Abfragen, da Daten nicht nach logischen Gruppen organisiert sind [GeeksforGeeks 2024a].

2.3 Skalierbarkeit

Skalierbarkeit bezeichnet die Fähigkeit eines Systems, seine Ressourcen und Kapazitäten flexibel an veränderte Anforderungen anzupassen, wie etwa wachsende Nutzerzahlen, größere Datenmengen oder eine erhöhte Verarbeitungslast. Dieses Prinzip ist entscheidend, um Systeme zukunftssicher zu gestalten und sowohl kurzfristige Spitzenbelastungen als auch langfristiges Wachstum effizient zu bewältigen. Skalierbarkeit wird dabei in zwei grundlegende Ansätze unterteilt:

Vertikale Skalierung (Scale Up). umfasst die Erweiterung der Ressourcen eines einzelnen Systems, um dessen Kapazität zu steigern. Dabei werden die Hardware-Komponenten, wie Prozessoren, Arbeitsspeicher oder Speicherplatz, aufgerüstet, um eine höhere Leistung zu erzielen. Vertikale Skalierung bezieht sich ausschließlich auf die Verbesserung eines einzigen Servers oder einer einzelnen Instanz, ohne zusätzliche Systeme hinzuzufügen. Diese Methode basiert auf der Optimierung bestehender Ressourcen innerhalb einer zentralen Einheit.

Horizontale Skalierung (Scale Out). erweitert die Kapazität eines Systems durch das Hinzufügen zusätzlicher Server, Instanzen oder Knoten, die parallel arbeiten. Sie basiert auf der Schaffung eines verteilten Systems, in dem Arbeitslasten und Daten gleichmäßig über mehrere Einheiten verteilt werden. Bei dieser Methode bleibt die Leistung nicht auf ein einzelnes System beschränkt, da die Arbeitslast von mehreren Ressourcen gleichzeitig bearbeitet wird. Die horizontale Skalierung setzt eine verteilte Architektur voraus, bei der mehrere Systeme miteinander verbunden sind, um die Gesamtleistung zu erhöhen [Henderson 2023].

Automatische Skalierung. ist der Prozess, bei dem Ressourcen dynamisch und flexibel an die aktuellen Leistungsanforderungen einer Anwendung angepasst werden. Automatische Skalierung wird häufig in Cloud-Umgebungen eingesetzt, bei denen Ressourcen automatisch hinzugefügt oder entfernt werden, abhängig von der aktuellen Last oder dem Bedarf. Dieses Konzept minimiert den manuellen Verwaltungsaufwand und stellt sicher, dass Systeme jederzeit optimal konfiguriert sind [Microsoft 2025].

3 PATTERN UND KONZEPTE FÜR RESILIENZ IN VERTEILTEN SYSTEMEN

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.

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 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

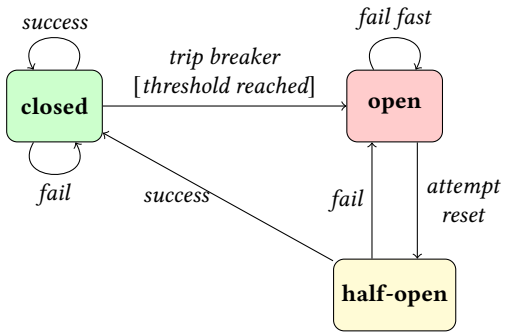

Abb. 1. Circuit Breaker Zustandsdiagramm nach [Montesi and Weber 2016].

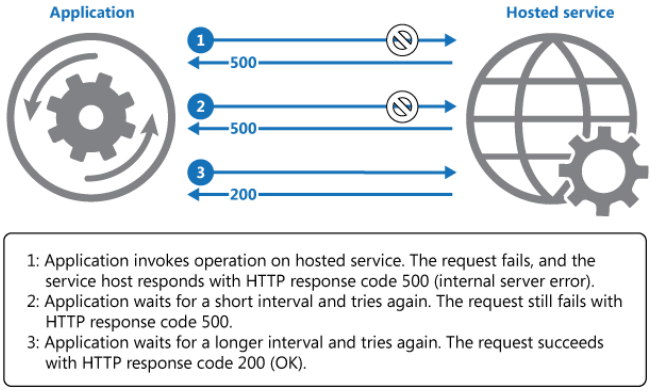

Abb. 2. Aufrufen eines Vorgangs in einem gehosteten Dienst unter Verwendung von *Retry* [Microsoft [n. d.]]

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.

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.

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 ist die Hystrix-Bibliothek für Java, 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 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.

3.3 Fallback-Strategien

In komplexen Systemen, insbesondere in verteilten Umgebungen, stellt die ständige Verfügbarkeit eine der zentralen Anforderungen dar. Dennoch sind solche Systeme anfällig für unvorhergesehene Fehler oder Ausfälle in einzelnen Komponenten, die die Funktionalität des gesamten Systems beeinträchtigen können. Diese Störungen können vielfältiger Natur sein – von Netzwerkproblemen über Ressourcenüberlastungen bis hin zu Hardwarefehlern. Solche Vorfälle bergen das Risiko, dass Benutzer den Zugriff auf kritische Dienste verlieren oder die Stabilität des Systems nachhaltig gefährdet wird. Ohne geeignete Mechanismen zur Fehlerbewältigung ist es schwierig, die Integrität und Verfügbarkeit des Systems sicherzustellen.

Eine effektive Methode, um diese Herausforderungen zu bewältigen, ist die Fallback-Strategie. Sie bietet einen strukturierten Ansatz, um die Auswirkungen von Fehlern oder Ausfällen zu minimieren und die Stabilität sowie Verfügbarkeit des Systems auch unter schwierigen Bedingungen sicherzustellen. Die Fallback-Strategie beschreibt Mechanismen, die alternative Lösungen aktivieren, um die Funktionalität eines Systems auch bei Fehlern oder Ausfällen aufrechtzuerhalten. Sie basiert auf einer Kombination aus frühzeitiger Fehlererkennung, der Aktivierung vorab definierter Alternativen und der Fähigkeit, das System weiterhin stabil zu halten. Diese Alternativen können in Form von redundanten Ressourcen, alternativen Prozessen oder einer reduzierten Funktionalität auftreten, die den Kernbetrieb absichern. Dadurch wird sichergestellt, dass das System wesentliche Aufgaben weiterhin erfüllt, während das ursprüngliche Problem behoben wird [Shekhar 2024].

Die Vorteile der Fallback-Strategie liegen vor allem in ihrer Fähigkeit, die Zuverlässigkeit eines Systems signifikant zu verbessern. Durch die Möglichkeit, den Betrieb auch bei Teilstörungen aufrechtzuerhalten, wird die Verfügbarkeit des Systems deutlich erhöht. Gleichzeitig werden Ausfallzeiten minimiert, da kritische Prozesse durch alternative Mechanismen nahtlos weitergeführt werden können. Dies reduziert die Auswirkungen von Fehlern oder Ausfällen auf die Endnutzer und schützt die Benutzererfahrung. Zudem trägt die Fallback-Strategie zur Fehlertoleranz bei, indem sie es einem System ermöglicht, Störungen abzufedern und die Stabilität des Gesamtsystems auch unter schwierigen Bedingungen zu gewährleisten. Indem essenzielle Funktionen weiterhin verfügbar bleiben, wird die Resilienz des Systems gestärkt und das Risiko schwerwiegender Beeinträchtigungen minimiert [Gabrielson 2019].

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 dem Load Balancer gestellt, welcher diese dann mithilfe einer geeigneten Strategie möglichst gerecht auf die Service-Instanzen aufteilt.

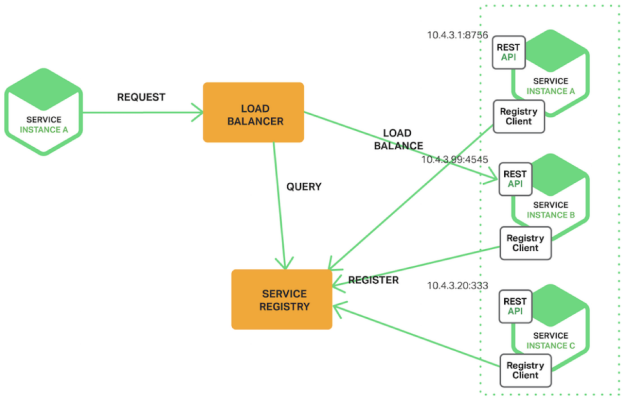

Abb. 3. Load Balancer mit drei Serviceinstanzen [Schöner et al. 2017]

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, 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 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 Wegen 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 2024b].

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

- 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 DNS Round Robin

Im vorherigen Abschnitt wurde eine Methode erläutert, mit der serverseitig mehrere Dienstinstanzen verwaltet und gebündelt und dann nach außen als eine Einheit kommuniziert werden. Dadurch steigt sowohl die Verfügbarkeit des Systems als auch die Kapazität für Anfragen, da mehrere Instanzen die gleiche Arbeit verrichten können und einzelne Ausfälle nicht zum Totalausfall führen.

Dennoch ist auch ein System mit Lastverteilung nicht ausfallsicher: Der Lastverteiler selbst oder der gesamte Servercluster könnte ausfallen, woraufhin die Dienstleistung nicht mehr erreichbar wäre. Eine Strategie, um diesem Fall entgegenzusteuern, ist DNS Round Robin.

Das Domain Name System, kurz DNS, wird genutzt um eine Assoziation zwischen einer Domain und einer IP-Adresse zu hinterlegen. Beispielsweise könnte ein Record hinterlegt werden, dass die Domain `server.de` auf die Serveradresse des ersten Lastverteilers zeigt. Wenn ein Client nun anstelle einer IP-Adresse die zugehörige Domain benutzt, wird zuerst der lokale DNS (Cache) befragt, ob der Name zu einer IP aufgelöst werden kann. Ansonsten wird der externe autoritative Namensserver nach der IP befragt, und diese dann lokal zwischengespeichert und genutzt [Kopparapu 2002].

Falls die IP des Lastverteilers nicht erreichbar ist, würde die Anfrage also fehlschlagen. Es ist jedoch möglich, für eine Domain mehrere IP-Adressen zu hinterlegen. Diese würden dann zu verschiedenen Servercomputern, ggf. mit jeweils eigenen internen Lastverteilern führen.

Wenn ein Client nun zur Namensauflösung den autoritativen DNS befragt, gibt dieser immer alle Records zurück. Die Reihenfolge der IP-Adressen wird jedoch im Round Robin Verfahren zyklisch rotiert: Wenn die erste Anfrage die drei Adressen `A-B-C` erhält, würde die zweite Anfrage die Antwort `B-C-A` erhalten.

So ist bereits für eine rudimentäre Lastverteilung gesorgt, da bei jeder Auflösung eine andere Adressreihenfolge herausgegeben und genutzt wird. Auch die Verfügbarkeit ist erhöht, da ein Client bei Nichterreichbarkeit einer Adresse zur nächsten zurückgreifen kann.

3.5.1 Vorteile. DNS Round Robin hat zwei große Vorteile:

- Es ist simpel zu implementieren, da keine spezielle Software auf dem Server- oder Clientsystem notwendig ist. Die Lastverteilung und Redundanz wird hier allein durch den Namensserver ermöglicht.
- Es schützt vor kompletter Nichterreichbarkeit. Ein Lastverteiler auf dem Server hilft nicht mehr wenn er selbst ausgefallen ist, oder wenn durch andere Probleme wie Geoblocks oder Firewalls gewisse IP-Ranges für den Client nicht erreichbar sind.

3.5.2 Nachteile. Auf den ersten Blick scheint dieses Verfahren ähnlich effektiv zu sein wie die statische Round Robin Lastverteilung aus Abschnitt 3.4.3: Anfragen werden ohne Einsicht in die Ressourcenverfügbarkeit möglichst fair auf Kapazitäten aufgeteilt. Dies funktioniert in der Praxis jedoch nur eingeschränkt, da DNS-Einträge zwischengespeichert werden. Die Lastverteilung erfolgt also nur in seltenen Abständen. Besonders wenn regulär nur eine kleine Anzahl von Clients mit jeweils hohen Rechenanforderungen auf das System zugreifen, kann schnell durch Zufall eine ungleiche Auslastung entstehen (eine große Nutzerzahl würde erhebliche Schwankungen eindämmen).

Dieses Problem kommt erneut ins Spiel, wenn einer Überlastung entgegengesteuert werden soll. Nur neue Nutzer würden einen neu hinzugefügten DNS-Eintrag unmittelbar erhalten, wodurch eine erhebliche Verzögerung bis zur gleichverteilten Benutzung aller DNS-Einträge erfolgt.

Zuletzt ist auch das Verhalten im Fehlerfall nachteilhaft. Zwei bestehende Probleme haben den Ursprung darin, dass der Client die Verantwortung für das Failover-Verhalten übernimmt:

- Der Rückfall auf die weiteren IP-Adressen kann nur erfolgen, wenn der Client einen Fehler bei dem Abruf der Primäradresse erkennt. Dies funktioniert wenn der Server nicht erreichbar ist. Wenn aber stattdessen beispielsweise Fehlercodes anstatt von Inhalt zurückgegeben werden, ist die Kommunikation mit dem Host aus Sicht des Clients erfolgt; es erfolgt kein Failover.
- Der Client kann erst nach einer gewissen Wartezeit davon ausgehen, dass eine Adresse nicht erreichbar ist. So kann eine erhebliche Wartezeit entstehen, bis das Failover-Verhalten ausgelöst wird [StackOverflow [n. d.]].

3.6 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 ZUSAMMENFASSUNG UND AUSBLICK

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

Jacob Gabrielson. 2019. Vermeiden von Fallback in verteilten Systemen. *Amazon Builders' Library* (2019).

GeeksforGeeks. 07.12.2024c. Redundancy in System Design. https://www.geeksforgeeks.org/redundancy-system-design

GeeksforGeeks. 11.11.2024b. *Load Balancing Algorithms.* https://www.geeksforgeeks.org/load-balancing-algorithms/#1-static-load-balancing-algorithms

GeeksforGeeks. 2024a. Data Partitioning Techniques in System Design. https://www.geeksforgeeks.org/data-partitioning-techniques

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/

Dina Henderson. 2023. Cloud scalability: Scale-up vs. scale-out. *IBM Blog* (2023). https://www.ibm.com/think/topics/scale-up-vs-scale-out

Chandra Kopparapu. 2002. *Load balancing servers, firewalls, and caches.* Wiley, New York, NY and Weinheim.

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

Microsoft. 2025. Automatische Skalierung. https://learn.microsoft.com/de-de/azure/architecture/best-practices/auto-scaling

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).

Gaurav Shekhar. 2024. Microservices Design Patterns for Cloud Architecture. *IEEE Chicago Section* (2024).

StackOverflow. [n. d.]. *DNS Round Robin very slow requests in failover scenario.* https://stackoverflow.com/questions/4489897/dns-round-robin-very-slow-requests-in-failover-scenario