Schriftliche Ausarbeitung

-

Labor Verteilte Systeme

FileNest

Erstellt am: 11.09.2024

Florian Pallmann
Timo Heckmann

Contents

1 Anforderungen

Für den Filesharing-Server FileNest wurden im Rahmen einer Anforderungsanalyse folgende funktionale und nicht-funktionale Anforderungen für das fertige Produkt festgelegt. Das Hauptziel des Projekts liegt in der Umsetzung gelernter Techniken und Praktiken für die Gestaltung, sowie dem erlernen eines korrekten Umgangs mit verteilten Systemen.

1.1 Funktionale Anforderungen

Anforderung	**Beschreibung**
Datei-Upload	Benutzer sollen Dateien auf den Server hochladen können.
Datei-Download	Benutzer sollen Dateien vom Server herunterladen können.
Datei-Löschen	Benutzer sollen eigens hochgeladene Dateien vom Server löschen können.
Benutzerregistrierung	Der Server muss eine sichere Registrierung und Authentifizierung von Benutzern ermöglichen.
Suche und Filterung	Benutzer sollen Dateien anhand von Namen, Dateitypen und anderen Metadaten suchen und filtern können.

1.2 Nicht-funktionale Anforderungen

Anforderung	**Beschreibung**
Leistung	Der Server muss in der Lage sein, eine hohe Anzahl gleichzeitiger Benutzeranfragen zu verarbeiten, ohne die Leistung zu beeinträchtigen.
Ausfallsicherheit	Das gesamte System muss im Falle des ausfalls eines oder mehrerer Komponenten weiterhin größtmöglich seine Funktionalität aufrecherhalten können
Skalierbarkeit	Das System sollte skalierbar sein, um steigende Benutzerzahlen und Datenmengen handhaben zu können.
Benutzerfreundlichkeit	Die Benutzeroberfläche soll intuitiv und einfach zu bedienen sein, um eine schnelle Einarbeitung zu ermöglichen.

2 Allgemeiner Projektaufbau

2.1 Beschreibung der Architektur

Die Architektur des verteilten Filesharing-Systems (1) besteht aus fünf Hauptkomponenten: einem Broker, einem MetaDatenServer mit Datenbank, einem FileServer, einem Cluster von Datei-Datenbanken und einem Admin-Dienst. Der Broker fungiert als Vermittler zwischen den verschiedenen Diensten, um die Kommunikation und Koordination zu ermöglichen. Er fungiert zeitgleich als Publisher, um die Daten an alle relevanten Dienste gleichzeitig zu verteilen. Der MetaDatenServer verwaltet alle Dateimetadaten und Zugriffsrechte, während der FileServer die eigentlichen Dateien entgegennimmt. Diese FileServer sind eng verbunden mit dem skalierbaren und hochverfügbaren Cluster, in welchem die Dateien persistent und verteilt gespeichert werden können. Der Admin übernimmt hier die Kontrollaufgaben, indem er in zyklischen Abfragen die Schnittstellen der Cluster-Nodes der Dateispeicherung und dem dazugehörigen Cluster auf Verfügbarkeit und Speicherplatz überprüft und bei Bedarf dies in den Metadaten festhält. Diese Architektur fördert Skalierbarkeit, Ausfallsicherheit und effizientes Dateimanagement. Es wird unterstützt durch verteilte Speicherlösungen und eine zentrale Verwaltung der Metadaten. Für die Interaktion mit dem System wird eine Weboberfläche in einem Client bereitgestellt. Dieser kommuniziert mit dem Backendsystem über einen Loadbalancer mit Reverse Proxy, um Anfragen zu verteilen und das System zu kapseln.

2.2 Kommunikation zwischen den Komponenten

In dieser Architektur erfolgt die Kommunikation zwischen den meisten Komponenten über REST, während die Verbindungen zum Broker und zwischen den einzelnen Brokernodes auf Sockets basieren. Der Broker bietet allerdings auch REST-Endpunkte an, welche für die initiale Kopplung zwischen den

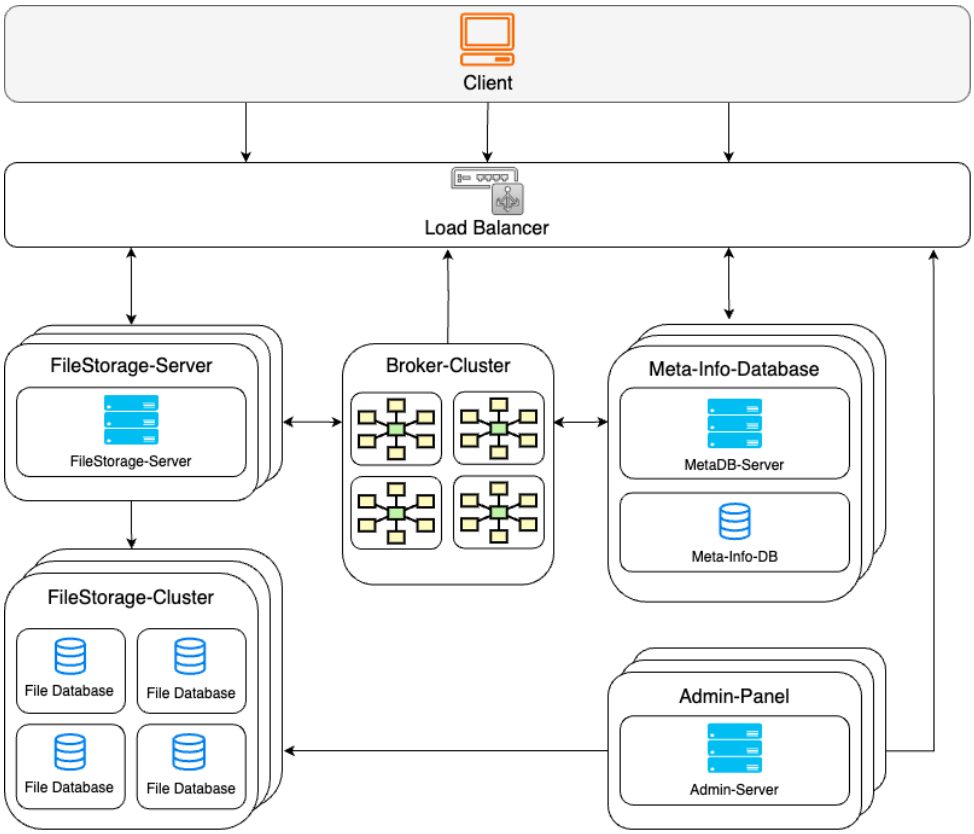

Figure 1: Allgemeine Architektur

Komponenten sind. Auch das Hinzufügen der Broker-Queue passiert per REST. Der Austausch des Brokers basiert auf Socket-Verbindungen, was eine schnelle, bidirektionale und persistente Verbindung ermöglicht, ideal für Echtzeitinteraktionen. Der Broker verwaltet die Vermittlung von Anfragen an den MetaDatenServer und den FileServer ebenfalls über Sockets, um effiziente und dauerhafte Verbindungen zwischen diesen Komponenten sicherzustellen. Die übrigen Interaktionen zwischen MetaDatenServer, FileServer und anderen Diensten erfolgen über REST, was einfache, standardisierte HTTP-basierte Anfragen und Antworten ermöglicht und die Modularität und Interoperabilität des Systems unterstützt.

2.3 Architekturentscheidungen

2.3.1 Kommunikation

Die Entscheidung, Sockets für die Kommunikation zwischen dem Broker und den Clients sowie zwischen dem Broker und den anderen Systemkomponenten (MetaDatenServer und FileServer) zu verwenden, wurde getroffen, um Echtzeitkommunikation und geringe Latenzen zu gewährleisten. Gleichzeitig wurde für die Interaktionen zwischen den anderen Systemkomponenten und externen Diensten REST gewählt, um Standardisierung, Modularität und die Nutzung bestehender HTTP-Infrastrukturen zu ermöglichen.

2.3.2 Datenspeicherung

Die Wahl eines Clusters für die Dateispeicherung bietet hohe Skalierbarkeit und Ausfallsicherheit, auch wenn einzelne Nodes ausfallen, während die Datenbank des MetaDatenServers eine effiziente Verwaltung und schnelle Abfragen der Dateimetadaten ermöglicht.

2.3.3 Broker

Der Broker als zentrale Einheit für den Datenaustausch wurde eingeführt, um die Aufgabe der Verteilung in ein eigenes System zu kapseln. Dadurch kann sich ein System komplett um eine performante Verteilung und den Umgang mit der Erreichbarkeit eines Systems kümmern. Dass der Broker als Cluster umgesetzt wird, resultiert daraus, dass durch eine Aufteilung von Verbindungen die Performance weiter gesteigert

werden kann und durch das Synchronisieren einer gemeinsamen Warteschlange ist das Cluster sicher vor möglichen Ausfällen einzelner Nodes.

2.3.4 Adminservice

Die Umsetzung der administrativen Aufgaben in einem eigenen Teil hat den Hintergrund, dass hier auch wieder eine Fokussierung auf einen speziellen Aufgabenbereich passieren kann. Dieser ist rein auf Überwachung ausgelegt und seine Aufgaben blockieren oder verlangsamen somit nicht Komponenten des restlichen produktiven Systems. Außerdem benötigt diese Instanz eine Verbindung zu dem Datei-Cluster und der Datenbank mit den Metainformationen, da bei einem Vorfall der entsprechende Clusterzugang in den Metainformationen markiert werden kann. Durch diese Auslagerung kann die Verbindung zwischen den Komponenten der Dateien und den Metainformationen weiterhin getrennt werden.

2.3.5 Loadbalancer

Ein Loadbalancer ist in diesem System notwendig, da es auf Skalierung ausgelegt ist und hiermit die Anfragen der Nutzer performant auf alle Komponenten verteilt werden können. Auch bei einem Ausfall von einer einzelnen Komponente kann hiermit das System weiter aufrechterhalten werden. Deswegen befindet sich dieser direkt zwischen Client und dem restlichen System.

3 Wahl und Begründung der Systemkomponenten

3.1 Broker-Cluster

In diesem Projekt basiert die Kommunikation zwischen den Brokern auf einer Socket-Verbindung, insbesondere im Kontext des Austauschs von Queues. Dies kann durch mehrere technische und praktische Vorteile untermauert werden, die sich aus den Benchmarks, siehe Quelle Luecke 2018, ergeben:

3.1.1 Echtzeit-Kommunikation und Latenzzeiten

Eine der Stärken von Socket-Verbindungen, wie WebSockets, ist die Fähigkeit zur bidirektionalen Echtzeit-Kommunikation. WebSockets weisen eine deutlich niedrigere Latenzzeit auf als traditionelle HTTP-Anfragen. Im Vergleich dazu beträgt die durchschnittliche Zeit für eine HTTP-Anfrage etwa 107 Millisekunden, während eine Socket-Anfrage lediglich 83 Millisekunden benötigt. Dieser Unterschied wird besonders signifikant, wenn es um den Austausch von Nachrichten in Echtzeit geht, wie es bei der Synchronisation von Queues zwischen Brokern erforderlich ist. Die Echtzeitfähigkeit von WebSockets ermöglicht es den Brokern, sofort auf eingehende Nachrichten zu reagieren und die Queues zeitnah zu aktualisieren, was bei HTTP-Verbindungen nicht in der gleichen Geschwindigkeit möglich ist. Dies ist in diesem Projekt sehr relevant, da aufgrund der gewählten Architektur zur Beschleunigung der Anwendung, alle Clients des gleichen Typs mit verschiedenen Brokern verbunden sein können. Die Kommunikation zwischen den Brokern ist somit ausschlaggebend für die Zeit des Hinzufügens bis zum Erreichen aller gewünschten Clients, um einen konsistenten Zustand möglichst schnell wieder herzustellen.

3.1.2 Skalierbarkeit und Performance

Die Benchmark-Tests verdeutlichen auch, dass WebSockets bei der Verarbeitung einer hohen Anzahl paralleler Anfragen signifikant leistungsfähiger sind als HTTP. Während HTTP-Anfragen bei 50 parallelen Anfragen etwa 5 Sekunden benötigten, um abgeschlossen zu werden, konnten Sockets dieselbe Anzahl von Anfragen in nur etwa 180 Millisekunden abwickeln. Diese Leistungssteigerung zeigt sich auch bei höheren Belastungen: Sockets können bis zu 4000 Anfragen pro Sekunde verarbeiten, während HTTP nur etwa 10 Anfragen pro Sekunde bewältigt. Da die Broker in dieser Anwendung der Zentrale Kommunikationspunkt zwischen Fileserver und Metadatenserver ist, laufen alle relevanten Anfragen aller Nutzer über diese Broker. Dies kann bei einer skalierten Anwendung zu enormen Datenmengen führen. Für Broker, die mit großen Datenmengen und schnellen Nachrichtenwechseln umgehen müssen, stellt dies einen entscheidenden Vorteil dar, da die Skalierbarkeit und Geschwindigkeit der Kommunikation durch Socket-Verbindungen erheblich verbessert werden.

3.1.3 Reduzierter Datenaufwand

Ein weiterer Vorteil der Nutzung von Socket-Verbindungen ist der geringere Datenaufwand im Vergleich zu HTTP. Die Datenübertragung über WebSockets erfordert wesentlich weniger Overhead, da keine zusätzlichen HTTP-Header gesendet werden müssen. Die in der Quelle angegebenen Datenübertragungsraten zeigen, dass eine HTTP-Anfrage und -Antwort zusammen etwa 282 Bytes benötigen, während die Äquivalente über WebSocket nur 54 Bytes umfassen. Da diese Anwendung das Potenzial hat enorm skaliert zu werden, muss darauf geachtet werden, so wenig Datentraffic wie möglich zu erzeugen. Diese Effizienz reduziert den Gesamtaufwand für die Datenübertragung, was in einer Steigerung der Performance des Systems widerspiegelt, was für die effiziente Verwaltung und Synchronisation von Queues zwischen Brokern von großer Bedeutung ist.

3.1.4 Flexibilität und Robustheit

WebSockets bieten eine kontinuierliche Verbindung, die eine flexible und reaktionsschnelle Kommunikation ermöglicht. WebSockets sind in der Lage, eine große Anzahl von Nachrichten über eine einzige Verbindung zu senden, ohne dass eine neue Verbindung für jede Nachricht aufgebaut werden muss. Dies reduziert die Notwendigkeit für ständige Verbindungsaufbau- und Abbauzeiten und ermöglicht eine robuste und kontinuierliche Datenübertragung. Die Broker in diesem Projekt bilden zusammen ein Cluster, wobei eine ständige Kommunikation notwendig ist, um Verbindungen und Nachrichten auszutauschen. Somit werden diese Verbindungen nicht unnötigerweise aufrecht gehalten.

3.1.5 Belastbarkeit und Fehlertoleranz

Die Robustheit von Socket-Verbindungen ist ein weiterer entscheidender Faktor. WebSocket-Verbindungen haben im Vergleich zu HTTP-Anfragen eine schnellere Wiederherstellung im Falle eines Verbindungsabbruchs. Dies erhöht die Fehlertoleranz und gewährleistet eine zuverlässige Kommunikation zwischen Brokern und erhöht die Ausfallsicherheit. In einem System, das auf die ständige Verfügbarkeit und Konsistenz von Queues angewiesen ist, trägt diese erhöhte Fehlertoleranz zur Stabilität und Zuverlässigkeit des gesamten Kommunikationssystems bei.

3.2 Nginx

In diesem Projekt wurde Nginx als Komponente für die Lastverteilung und das Reverse-Proxy-Management integriert. Die Wahl von Nginx bringt dabei mehrere Vorteile mit sich, die sich besonders in der effizienten Verwaltung von Lasten und der Sicherstellung hoher Verfügbarkeit widerspiegeln. Die folgenden Punkte beziehen sich auf Ma and Chi 2022

3.2.1 Lastverteilung

Nginx unterstützt eine Vielzahl von Lastverteilungsalgorithmen wie Round Robin, Weighted Round Robin (WRR) und Least Connection, die eine optimale Verteilung der Anfragen an den Broker, den Metadaten Server und den Servern für das MinIOCluster ermöglichen. In einem System wie diesem, in dem mehrere FileServer-Anfragen gleichzeitig verarbeitet werden müssen, sorgt die Lastverteilung von Nginx dafür, dass keine Überlastung einzelner Server auftritt und die Systemressourcen effizient genutzt werden.

3.2.2 Reverse Proxy und Sicherheitsvorteile

Durch die Verwendung von Nginx als Reverse Proxy bleiben die tatsächlichen IP-Adressen der internen Server (wie dem Metadaten Server und dem MinIOCluster) vor den Clients verborgen. Dies trägt zu einer erhöhten Sicherheit bei und schützt die Backend-Komponenten vor direktem Zugriff, was die Gesamtsicherheit des Systems verbessert.

3.2.3 Performance und Latenzzeiten

Nginx ermöglicht, hoch optimierte Lastverteilungsalgorithmen wie IP_HASH zu verwenden, die dafür sorgen, dass Anfragen effizient an die richtigen Server weitergeleitet werden. Diese Fähigkeit ist besonders wichtig für den Broker, der die Datenströme zwischen Clients und dem Filesharing-System verwaltet. Durch die Lastverteilung und die schnelle Weiterleitung von Anfragen sorgt Nginx für eine geringe Latenz und schnelle Reaktionszeiten.

3.2.4 Verfügbarkeit und Fehlertoleranz

Mit der Integration von Nginx wird im System eine High-Availability-Architektur geschaffen, die sicherstellt, dass auch im Falle eines Ausfalls eines Servers oder einer Komponente ein Backup-Server automatisch einspringt. Dies ist besonders für die Aufrechterhaltung des Zugriffs auf den MetaDatenServer und das MinIOCluster wichtig, um eine hohe Verfügbarkeit des Systems zu gewährleisten.

3.3 MinIO

Für dieses Projekt ist es sinnvoll, MinIO als zentrale Lösung zur Speicherung der Dateien zu verwenden. MinIO bietet mehrere Vorteile, die sich besonders gut für ein hochverfügbares, skalierbares und leistungsstarkes System eignen. Die folgenden Daten könne MinIO 2024 entnommen werden.

3.3.1 Leistung

MinIO ist bekannt als das schnellste Object Storage System weltweit, mit Lesegeschwindigkeiten von bis zu 325 GB/s und Schreibgeschwindigkeiten von 171 GB/s auf 32 Knoten mit Standardhardware. Diese Geschwindigkeit ist entscheidend, um sicherzustellen, dass das Filesharing-System auch unter hoher Last schnell und effizient arbeitet. Besonders bei gleichzeitigen Dateioperationen in einem verteilten System sorgt MinIO dafür, dass Lese- und Schreibvorgänge performant ablaufen und somit keine Engpässe entstehen.

3.3.2 Skalierbarkeit

MinIO ist darauf ausgelegt, nahtlos zu skalieren. Es kann mit einem einzigen Cluster beginnen und auf jede Größe erweitert werden, ohne dass dabei ein Rebalancing erforderlich ist. Dies ist ein großer Vorteil für ein verteiltes System, in dem die Anforderungen an die Speicherkapazität mit wachsender Nutzerbasis steigen können. MinIO unterstützt die Nutzung unterschiedlicher Hardware und Speichermedien, was das System flexibel und zukunftssicher macht.

3.3.3 Verwaltung

Die einfache Konfiguration und Verwaltung von MinIO reduziert die Fehleranfälligkeit und minimiert den Administrationsaufwand. Mit nur einem Befehl lassen sich Updates ohne Ausfallzeiten durchführen, was die Betriebszeit maximiert. In einem verteilten Filesharing-System, in dem ständige Verfügbarkeit der Schlüssel zum Erfolg ist, bietet MinIO den Vorteil einer fast wartungsfreien Verwaltung.

3.3.4 Sicherheit und Datenintegrität

MinIO bietet umfangreiche Sicherheitsfunktionen, wie Object Locking, das sicherstellt, dass Dateien nicht ungewollt geändert oder gelöscht werden können. Zudem ermöglicht der MinIO Enterprise Key Management Server (KMS) die sichere Verwaltung von Verschlüsselungsschlüsseln, was besonders für den Schutz sensibler Daten in einem Filesharing-System relevant ist. Durch die Multi-Tenant-Funktion des KMS können verschiedene Nutzergruppen sicher und isoliert auf Daten zugreifen.

3.3.5 Multi-Cloud-Kompatibilität

MinIO ist das einzige Object Storage System, das Multi-Cloud-fähig ist. Dies bedeutet, dass das Filesharing-System sowohl in privaten als auch in öffentlichen Cloud-Umgebungen betrieben werden kann, ohne dass Anpassungen erforderlich sind. Diese Flexibilität ermöglicht es, die Vorteile unterschiedlicher Cloud-Anbieter zu nutzen und die Verfügbarkeit und Leistung weiter zu optimieren.

3.3.6 Verfügbarkeit und Ausfallsicherheit

Dank der Active-Active Replikation kann MinIO Dateien auf verschiedene geografisch verteilte Datenzentren replizieren. Diese Funktion stellt sicher, dass Dateien auch im Falle eines Server- oder Standortausfalls weiterhin verfügbar bleiben, was die Ausfallsicherheit des Systems erheblich erhöht. Für ein verteiltes Filesharing-System, das eine hohe Verfügbarkeit benötigt, ist dies ein zentraler Vorteil.

3.4 Fastify

Basierend auf den Anforderungen des verteilten Filesharing-Systems gibt es mehrere Gründe, warum die Verwendung von Fastify für die Server-Komponenten sinnvoll ist (Demashov and Gosudarev n.d.):

3.4.1 Leistung

Fastify hat in dieser Performance-Untersuchung die besten Ergebnisse unter den getesteten Node.js-Frameworks erreicht. Bei 300 Verbindungen konnte Fastify 9198 Anfragen pro Sekunde verarbeiten, mit einer Latenz von nur 32,03 ms. Dies ist besonders wichtig für ein verteiltes System, das möglicherweise eine hohe Anzahl gleichzeitiger Verbindungen und Anfragen bewältigen muss. Insbesondere Express konnte hier nur mit 5021 Anfragen und 59,27 ms abschneiden.

3.4.2 Effizienz bei API-Entwicklung

Der Artikel empfiehlt Fastify speziell für den Aufbau von APIs, die "Fastify which provides tools for the fast endpoints dynamic deployment which service clients' demands in calculations which do not depend on the file system and external resources". Dies passt gut zu den Anforderungen eines Brokers und den Servern in diesem System.

3.4.3 Modularität

Fastify wird im Artikel als modulares Framework beschrieben. Diese Eigenschaft kann bei der Entwicklung und Wartung komplexer Systeme wie dem verteilten Filesharing-System hilfreich sein, da sie eine bessere Strukturierung und einfachere Erweiterbarkeit des Codes ermöglicht.

3.4.4 Latenz

Die niedrige Latenz von Fastify ist besonders wichtig für die Komponenten des Systems, die schnelle Antwortzeiten erfordern, wie z.B. der Broker.

3.4.5 Moderne Architektur

Fastify ist ein relativ neues Framework, das moderne Best Practices und Architekturprinzipien berücksichtigt. Dies kann die Entwicklung eines robusten und zukunftssicheren Systems unterstützen. Zudem ist es ein sehr leichtgewichtiges Framework, was in einer kleineren Imagegröße für Docker-Container resultiert.

4 Komponenteninteraktion

Innerhalb der Filesharing-Anwendung muss die Interaktion der verschiedenen Komponenten so gestaltet sein, dass eine gute Performance, sowie eine maximale Ausfallsicherheit gewährleistet sind. Die Interaktionen der einzelnen Komponenten miteinander lässt sich bereits im Architekturdiagramm einsehen. Dabei stehen zur Kommunikation der Microservices untereinander meist die Kommunikation über **Nginx** und über den **Message-Broker** zur Verfügung. Die Wahl des Kommunikationsmittels hängt dabei immer von der konkreten Operation des Systems ab.

4.1 Client-Microservice-Kommuniktaion

Für die Verarbeitung von **Client**-Anfragen fungiert **Nginx** als zentrale Schnittstelle, die alle eingehenden Anfragen annimmt und an den entsprechenden Microservice weiterleitet. Dabei dient **Nginx** als Loadbalancer unter den verschiedenen im System laufenden Instanzen eines Services. Die Verteilung erfolgt mittels Round Robin. Des weiteren übernimmt Nginx bereits erste für die Stabilität unserer Anwendung erforderlichen Überprüfungen und lässt beispielsweise nur den Upload von Dateien, deren größe innerhalb der festgelegten Regularien liegen zu (In unserem Fall 10MB).

4.2 Micrservice-Microservice-Kommunikation

Das lesen und schreiben von Dateiinformationen erfolgt innerhalb der Anwendung rein aus unserem **Fastify**-Backend-Server heraus, der die Implementationen der entsprechenden REST-Routen enthält. Für den Zugriff auf Informationen zu Speicherort oder Metainformationen greift der **Fastify**-Server auf den **Datenbank-Service** zu.

Lesende Anfragen an den **MetaDB-Server**, wie beim erfragen eines Dateispeicherortes für den Datei-Download werden über **Nginx** durchgeführt um eine Lastverteilung des Datentransfers im System zu gewährleisten. Somit wird die Performance im System möglichst hoch und sichergestellt, dass die Anfrage an eine funktionsfähige Instanz des **Datenbank-Services** weitergeleitet wird.

Schreibende Operationen, wie beim Hochladen oder Löschen von Dateien, erfolgen über den **Message-Broker**. Der Broker koordiniert alle Schreibvorgänge auf die **MetaDB**, indem er sie an den **MetaDB-Server** weiterleitet. Dadurch kann sichergestellt werden, dass alle laufenden **Datenbank-instanzen** die Informationen erhalten und eine Kostistenz zwischen den Datenbanken gewährleistet ist.

5 Alternativen

5.1 Speicherung der Metainformationen

Anstelle eines Multi-Master-Datenbanksystems, bei dem alle Knoten sowohl Schreib- als auch Lesezugriffe handhaben können, könnte ein Master-Slave-Modell implementiert werden. In dieser Architektur übernimmt der Master die Schreibvorgänge, während die Slaves für das Lesen der Daten verantwortlich sind. Diese Trennung der Aufgaben verhindert Schreibkonflikte, reduziert die Komplexität der Replikation und bietet eine bessere Konsistenz. Obwohl dies möglicherweise weniger skalierbar ist als ein Multi-Master-Setup, bietet das Master-Slave-Modell eine einfachere Verwaltung und eignet sich gut für Szenarien, in denen der Schreibdurchsatz geringer ist als der Lesezugriff. Dieses Projekt implementiert das Multimasterprinzip, da ein hoher Schreibaufwand anfallen kann und so diese Last auch verteilt werden kann.

5.2 Speicherung der Dateien

Eine Alternative zum MinIO-Cluster, der eine lokale, verteilte Objektspeicherung bereitstellt, ist die Verwendung von cloudbasierten Speicherlösungen wie Amazon S3 oder Google Cloud Storage. Cloud-Speicher bieten eine elastische und kosteneffiziente Möglichkeit, Daten zu speichern und gleichzeitig hohe Verfügbarkeit und Redundanz zu gewährleisten. Der Vorteil von Cloud-Speichern liegt darin, dass sie den Wartungsaufwand für Infrastruktur reduzieren und eine nahtlose Skalierung ermöglichen. Für Anwendungen, die auf hohe Datenmengen und variable Lasten ausgelegt sind, bietet die Cloud-Infrastruktur zudem eine zuverlässige und weltweit verfügbare Speicherlösung, ohne dass lokale Hardware-Investitionen erforderlich sind. Hier wird dennoch MinIO eingesetzt, um unabhängig von diesen anderen Anbietern zu sein und das komplette System selbst hosten zu können.

6 Lösungen/Probleme bei der Projektdurchführung

6.1 Probleme eines Verteilten Systems

Ein verteiltes System hat neben seinen zahlreichen Vorteilen auch Nachteile. Darunter fallen unter anderem die Latenz bei der Kommunikation zwischen den Komponenten auf verschiedenen Servern oder sogar zwischen verschiedenen Rechenzentren. Eine Annahme und Simulation solcher Latenzen sind schwer anzunehmen und nachzuahmen. Außerdem wird dieses Projekt im Rahmen eines Studierendenprojektes umgesetzt, wobei technische und finanzielle Mittel fehlen, die Anwendung geografisch oder serverübergreifend zu testen. Für dieses Projekt ist eine ausreichende Lösung, indem man dieses Problem wahrnimmt und bei der Entwicklung durch Timeouts und Warten minimiert. Letztlich kann dies trotzdem nicht vollständig getestet und garantiert werden.

6.2 Ausfallsicherheit

Um dieses System ausfallsicher zu gestalten, muss dies bei der Architektur und der Umsetzung bedacht werden. Da es nicht genügt, die einzelnen Komponenten mehrmals bereitzustellen, da hier die Aufrechterhaltung der Konsistenz nicht möglich ist, müssen dafür zusätzliche Abläufe umgesetzt werden. In diesem Projekt ist dies nicht allgemein möglich, sondern muss auf Komponentenebene passieren. Bei den Datenbanken für die Metainformationen wird ein Multimaster Prinzip umgesetzt, welche durch den Broker auf einem einheitlichen Stand gehalten werden. Dieser sorgt durch eine Queue für eine Speicherung der Daten, welche brokerübergreifend gehalten wird. Bei der Speicherung der Dateien wird hier auf externe Mittel zurückgegriffen und ein Cluster von MinIO aufgesetzt. Dieses besteht aus mehreren Nodes, welches auch beim Ausfall einzelner Nodes die Verfügbarkeit aller Dateien verspricht. Der Broker verwendet eine voll vernetzte Topologie, welche dafür sorgt, dass neue Daten, die bei einem Broker ankommen, an alle anderen verteilt werden und infolgedessen beim Ausfall eines Einzelnen weiterhin verfügbar sind. Die restlichen Komponenten verarbeiten nur Daten und sind nicht auf Persistenz oder Synchronisation angewiesen und können einfach repliziert werden.

6.3 Konsistenz

Durch die Verwendung einer Multimasterdatenbank für die Metainformationen ist es in diesem System möglich, dass die Einhaltung von Konsistenz nicht immer garantiert ist. In diesem Projekt wurde sich bewusst für eine Architektur entschieden, die dem CAP-Theorem folgt, welches besagt, dass ein verteiltes System nicht gleichzeitig Konsistenz, Verfügbarkeit und Partitionstoleranz gewährleisten kann. Der Fokus liegt auf Verfügbarkeit und Partitionstoleranz, was bedeutet, dass die Systemleistung unter hoher Last maximal ist. Dies bedeutet, dass in bestimmten Situationen Kompromisse bei der Konsistenz der Daten eingegangen werden müssen. Durch diese Entscheidung kann eine hohe Reaktionsfähigkeit und eine bessere Nutzererfahrung sichergestellt werden, auch wenn dies bedeutet, dass die Daten möglicherweise nicht immer sofort konsistent sind. Die Lösung für dieses System ist, dass die Daten in der Queue der Broker gespeichert werden, bis sie versendet werden können. Wenn eine Komponente allerdings die Verbindung zum Broker nicht halten kann, aber weiterhin Anfragen vom Client erhält, können diese Daten einen abweichenden Stand haben. Dieser Kompromiss wird aufgrund des CAP-Theorems eingegangen.

6.4 Was nicht umgesetzt wurde

In diesem Projekt wurde die automatische Skalierung nicht umgesetzt, da sich bei der Zuweisung von MariaDB zu einem Datenbank-Server mit einer 1:1-Verbindung erhebliche Schwierigkeiten ergaben. MariaDB benötigt eine feste Zuordnung zu einem Server, um eine stabile und direkte Verbindung zu gewährleisten, was durch die Skalierung einzelner Container erschwert wird, da trotzdem garantiert werden muss, dass jeder Server eine eigene DB zugewiesen bekommt.

Ähnlich problematisch gestaltete sich die automatische Skalierung der MinIO-Container. In unserem System erfordert die Registrierung neuer Cluster einen API-Call, um die Instanzen ordnungsgemäß einzubinden. Allerdings bietet das Docker-Image des MinIO-Containers keine Möglichkeit, diesen API-Call automatisch durchzuführen, was die Implementierung einer nahtlosen Skalierung erschwert. Diese Einschränkung machte es notwendig, die Registrierung der Cluster manuell durchzuführen.

Beide Probleme können durch verschiedene Wege, wie beispielsweise weitere Routinen des Admincontainers, umgesetzt werden. Dieser könnte die Rechte bekommen, neue Container hochzufahren und diese dann zu registrieren und zuzuweisen, allerdings ist dies aus zeitlichen Gründen in diesem Projekt nicht umgesetzt worden. Bei der Architektur, Verwendung und Kommunikation im kompletten System ist dies allerdings bedacht worden und eine manuelle Erweiterung der Docker-Compose Datei und ggf. des initialen Scripts der DB kann dies bereits umgesetzt werden.

7 Reflektion

Die Architektur demonstriert ein tiefes Verständnis für die Komplexität verteilter Systeme und berücksichtigt wichtige Aspekte wie Skalierbarkeit, Leistung und Ausfallsicherheit. Eine besondere Stärke des Designs liegt in seiner modularen Struktur. Die Aufteilung in separate Komponenten wie Broker, MetaDatenServer, FileServer und Admin-Dienst ermöglicht eine klare Trennung der Zuständigkeiten. Dies fördert nicht nur die Wartbarkeit des Systems, sondern erleichtert auch zukünftige Erweiterungen und Optimierungen einzelner Komponenten.
Die Wahl moderner Technologien wie Fastify für die Server-Komponenten und MinIO für die Datenspeicherung zeigt einen starken Fokus auf Leistungsoptimierung. Insbesondere die Verwendung von MinIO als verteiltes Objektspeichersystem verspricht hohe Skalierbarkeit und Effizienz bei der Datenverwaltung. Die Integration von Nginx als Lastverteiler und Reverse Proxy trägt zusätzlich zur Verbesserung der Gesamtleistung und Sicherheit bei.
Ein weiterer positiver Aspekt ist die Berücksichtigung der Ausfallsicherheit. Die Implementierung eines Broker-Clusters mit voll vernetzter Topologie und die Verwendung von Multi-Master-Datenbanken für Metainformationen zeigen das Bestreben, Single Points of Failure zu vermeiden und die Systemverfügbarkeit zu maximieren.
Trotz dieser Stärken gibt es einige Aspekte, die kritisch betrachtet werden sollten. Die Komplexität des Systems, hauptsächlich durch die Vielzahl interagierender Komponenten, könnte die Fehlersuche und das Debugging erschweren. Hier wäre es sinnvoll, robuste Logging- und Monitoring-Lösungen zu integrieren, um die Systemüberwachung und Fehlerbehebung zu erleichtern.
Ein weiterer Punkt, der mehr Aufmerksamkeit verdient, ist die Datenkonsistenz in einem verteilten Umfeld. Während das Multi-Master-Datenbankmodell Vorteile in Bezug auf Verfügbarkeit und Partitionstoleranz bietet, könnten Konflikte bei gleichzeitigen Schreibzugriffen zu Inkonsistenzen führen. Eine detailliertere Strategie zur Konfliktlösung und Konsistenzsicherung wäre hier wünschenswert.
Abschließend ist anzumerken, dass die praktische Umsetzung und das Testen eines solch komplexen verteilten Systems eine erhebliche Herausforderung darstellen. Die erwähnten Einschränkungen bezüglich der Simulation realistischer Netzwerkbedingungen im Rahmen eines Studierendenprojekts sind ein wichtiger Punkt. Für eine vollständige Validierung des Designs wären umfangreiche Tests unter realen Bedingungen mit geografisch verteilten Komponenten erforderlich.
Allgemein kann festgehalten werden, dass dieses System mit seiner Architektur für ein verteiltes File-Sharing-System mit dem Fokus auf Performance erfolgreich und funktionierend umgesetzt wurde, allerdings noch mehr Beachtung für Test-Szenarien unter realen Bedingungen erfordert.

8 Anhang

Aufgrund der Größe der Diagramme sind enthaltene Schriftzüge und Details innerhalb der Dokumentation nur schwer zu erkennen. Daher lassen sich alle Diagramme nochmals als SVG im Projekt-Repository im Ordner ‘documentation‘ auf `https://github.com/TH-Projects/FileNest` einsehen.

Figure 2: Sequenzdiagramm - Account Registrierung

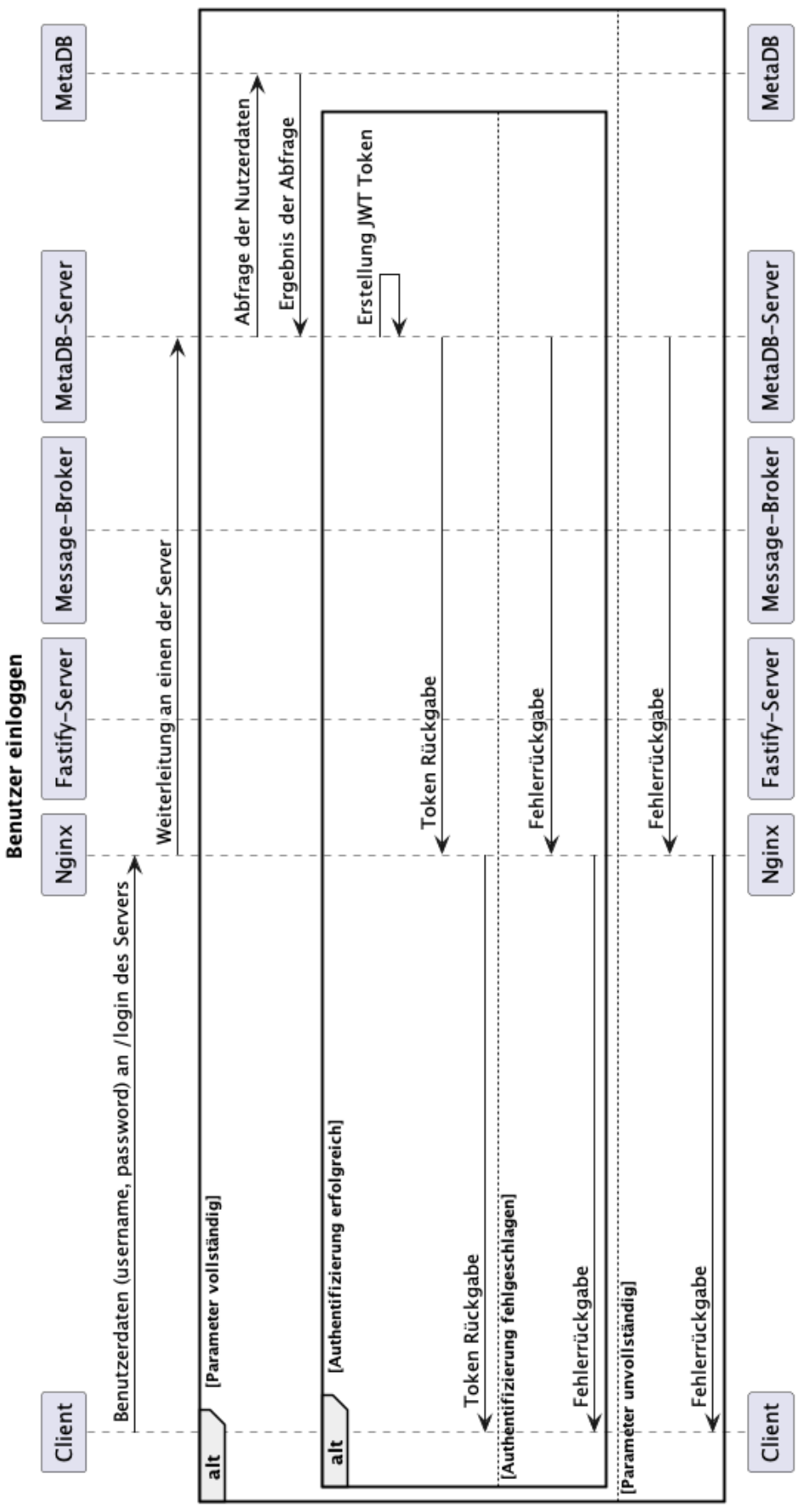

Figure 3: Sequenzdiagramm - Account Anmeldung

Figure 4: Sequenzdiagramm - Datei Upload

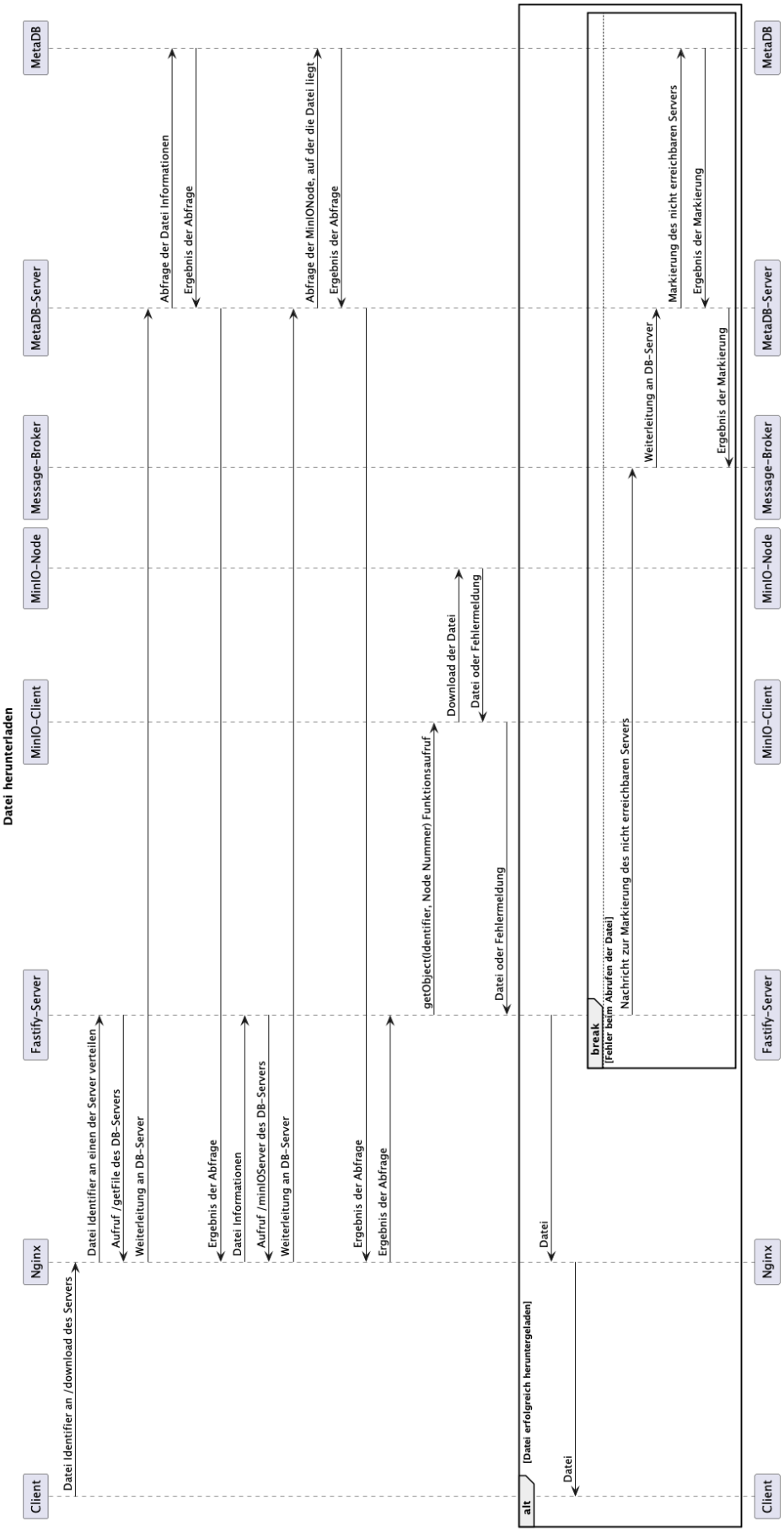

Figure 5: Sequenzdiagramm - Datei Download

Figure 6: Sequenzdiagramm - Datei löschen

References

Demashov, Danil and Ilya Gosudarev (n.d.). “Efficiency Evaluation of Node.js Web-Server Frameworks”. In: ().

Luecke, David (Jan. 27, 2018). *HTTP vs Websockets: A performance comparison.* Medium. URL: https://blog.feathersjs.com/http-vs-websockets-a-performance-comparison-da2533f13a77 (visited on 09/16/2024).

Ma, Chen and Yuhong Chi (2022). “Evaluation Test and Improvement of Load Balancing Algorithms of Nginx”. In: 10.

MinIO (2024). “Corporate Overview 2024: MinIO Enterprise Object Store”. In: Retrieved from MinIO Corporate Overview 2024.