MASTERARBEIT

ZUR ERLANGUNG DES AKADEMISCHEN GRADES MASTER OF SCIENCE

Realisierung konsistenter Microservice-Systeme unter Verwendung des Saga-Patterns

HOCHSCHULE FÜR TECHNIK, WIRTSCHAFT UND KULTUR LEIPZIG

Studiengang
Informatik

Fakultät
Informatik und Medien

Vorgelegt von
Richard Werner (B.Sc.)

Matrikelnummer
77353

Erstprüfer
Prof. Dr. rer. nat.
Thomas Riechert

Zweitprüfer
Johannes Elsmann
(M.Sc.)

February 2, 2024

I-ITWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Abstract

Atomare Transaktionen sind ein essentieller Bestandteil vieler moderner Anwendungen. In Architekturstilen wie Microservices besteht das Gesamtsystem aus mehreren in Kommunikation stehenden Komponenten. Dies stellt Systeme vor die Herausforderung, Transaktionen über die Grenzen einzelner Komponenten heraus abzubilden. Commitprotokolle wie der Zwei-Phasen-Commit (2PC) stellen zwar eine etablierte Möglichkeit dar, solche verteilten Transaktionen abzubilden, bringen jedoch auch Nachteile mit sich. Ein Nachteil ist das blockierende Verhalten während der Ausführung der Transaktion. Das von Garcia-Molina and Salem beschriebene Muster von Sagas bietet eine Möglichkeit, atomare Operationen in sequentieller Ausführung nachzubilden. Im Kontext verteilter Systeme wird dieses Muster verwendet, um Transaktionen sequentiell auszuführen und somit Atomarität und schließlich Konsistenz zu gewährleisten. In dieser Arbeit wird untersucht, ob das Konzept der Fehlerbehandlung in Sagas verwendet werden kann, um auf Netzwerkfehler während der Ausführung von verteilten Transaktionen zu reagieren. Es wird das Verhalten verschiedener Recoverymechanismen des Saga-Patterns unter Auftreten von Netzwerkpartitionen untersucht. Die Untersuchung findet im Rahmen eines praktischen Versuchs statt. Im Versuch wird ein exemplarischer Geschäftsprozess unter Verwendung des Saga-Patterns implementiert und iterativ optimiert. Jede Implementierung wird hinsichtlich Konsistenz und Durchsatz untersucht und bewertet. Schlussendlich sind Aussagen zu treffen, welche Voraussetzung eine Transaktion und deren Transaktionsteilnehmer erfüllen müssen, um mittels Saga-Pattern umgesetzt werden zu können.

Inhaltsverzeichnis

1 Einleitung

Moderne Anwendungen bestehen häufig aus mehreren individuellen Komponenten, die an der Lösung einer gemeinsamen Aufgabe beteiligt sind. Dabei besteht die Aufgabe einer Komponente oft darin, eine Anfrage entgegenzunehmen, bestimmte Bedingungen zu prüfen, die Daten zu transformieren, in einer Datenbank zu speichern und auf den Aufrufer mit einem Ergebnis zu antworten. Solche modularen Komponenten tauchen besonders im Architekturstil der Microservices auf. Ein Ziel der Modularisierung ist eine möglichst geringe Bindung der Services, damit die Komplexität innerhalb einer Komponente gering gehalten wird und falls notwendig durch eine andere Komponente ausgetauscht werden kann. Deshalb arbeiten Microservices oft mit einer eigenen Datenbank, die nur von diesem Service erreichbar ist.
Aufgrund der Verteilung der Daten über mehrere Datenbanken entstehen neue Herausforderungen, die das System bewältigen muss. Besonders die Verwendung von transaktionellen Operationen stellt die Herausforderung der verteilten Transaktionen. Die Mechanismen des Transaktionsinterfaces von ACID-Transaktionen einer relationalen Datenbank funktionieren nur dann, wenn alle auszuführenden Operationen in einem Transaktionskontextes ausgeführt werden. Prozesse, die atomare Veränderungen in mehreren Datenbanken bewirken sollen, werden als verteilte Transaktionen bezeichnet.
Commitprotokolle wie etwa der Zwei-Phasen-Commit (2PC) stellen eine mögliche Implementierung verteilter Transaktionen, die ACID-Eigenschaften gewährleisten können. Dabei werden jedoch die betroffenen Ressourcen blockiert, was zu einem geringen Durchsatz führen kann. In Fällen, die unter keinen Umständen Inkonsistenzen gewährleisten können, ist dies eine praktikable Lösung. In Systemen, die für einen Zeitraum einen inkonsistenten Systemzustand annehmen dürfen, können die Änderungen sequentiell prozessiert werden. Dies verspricht eine höhere Verfügbarkeit und höheren Durchsatz. Der Begriff der schlussendlichen Konsistenz (*Eventual Consistency*) beschreibt dieses Verhalten.

1.1 Fragestellung

Diese Arbeit verwendet das von Garcia-Molina und Salem vorgeschlagene Muster des Saga-Patterns im Kontext eines verteilten Systems. Das Muster verspricht atomares Verhalten mehrere Operationen in einer sequentiellen Ausführung. Dies erlaubt die Verwendung des Musters im Kontext verteilter Transaktionen.
Der zentrale Punkt des Musters befasst sich mit der Behandlung von Fehlern während der Transaktion. Es soll untersucht werden, ob Netzwerkausfälle in diese Fehlerbehandlung integrierbar sind.

1.2 Aufbau dieser Arbeit

Kapitel 2 befasst sich mit dem Begriff der Transaktion. Dabei werden damit verbundene Konzepte in zentralisierten und verteilten Systemen vorgestellt.
In Kapitel 3 wird Garcia-Molina und Salems Arbeit “Sagas” referenziert. Außerdem wird das darin definierte Entwicklungsmuster in den Kontext verteilter Systeme eingeordnet. Zusätzlich wird eine Formalisierung vorgenommen, die die Beschreibung von mittels Saga-Pattern implementierter Transaktionen erlaubt.
Die Vermutung, dass die Fehlerbehandlung des Saga-Patterns Netzwerkausfälle integrieren kann, soll in einem Versuch überprüft werden. Kapitel 4 bereitet den durchzuführenden Versuch vor. In diesem Kapitel wird das zu entwerfende System definiert indem ein imaginärer Geschäftsprozess definiert wird, der mittels Saga-Pattern umgesetzt werden soll. Es werden Metriken definiert, die der Beantwortung der These dienen sollen. Die Durchführung des Versuchs wird in Kapitel 5 erläutert. Die im Versuch gesammelten Messwerte, Ergebnisse und Zusammenhänge werden in Kapitel 6 dargestellt. Die These und die dazugehörigen Leitfragen werden in diesem Kapitel beantwortet.

2 Grundlagen

2.1 Datenbank

Eine Datenbank (DB) wird in der Literatur beschrieben als eine strukturierte Sammlung von Daten. Mit den Datenkann über ein Datenbank-Management-System (DBMS) interagiert werden[2], [3].

2.2 Datenbank-Management-System

Das Datenbank-Management-System (DBMS) ist eine STandardsoftware und ermöglicht Nutzern das Erstellen und Betreiben einer Datenbank. Die Nutzung eines DBMS verspricht folgende Vorteile[2]:

- Unabhängigkeit der Daten
- Effizienten Datenzugriff
- Datenintegrität und Datensicherheit
- Administration
- Mehrbenutzerbetrieb und Wiederherstellung nach einem Absturz
- Verringerte Entwicklungszeit für Anwendungen

2.3 Transaktionen

Eine *Transaktion* ist eine Abstraktion des DBMS, um mehrere Operationen als eine Einheit auszuführen. Die Verwendung dieser Abstrahierungsschicht vereinfacht den Mehrbenutzerbetrieb. Jede Operation kann entweder ein *read* oder *write* sein. Anwendungen haben die Möglichkeit, eine Transaktion zu beginnen und mehrere *reads* oder *writes* auf den Daten auszuführen. Schlussendlich kann die Anwendung alle durchgeführten Änderungen bestätigen (*commit*) oder zurückrollen (*abort*)[4].

2.4 Transaktionseigenschaften

2.4.1 ACID

ACID ist ein Akronym, welches häufig verwendet wird, um Eigenschaften auszudrücken, die eine Transaktion auf Datenbankebene aufweisen sollte. Die Buchstaben stehen für **A**tomicity (Atomarität), **C**onsistency (Konsistenz), **I**solation (Isolation) und **D**urability (Dauerhaftigkeit)[4].

Atomarität

Eine Transaktion, also eine Folge von Datenbankoperationen, muss entweder alle Operationen ausführen oder in einem Fehlerfall alle Änderungen auf den ursprünglichen Zustand zurückrollen.

Konsistenz

Alle Änderungen der Datenbank überführen die Daten von einem konsistenten Zustand in einen ebenfalls konsistenten Folgezustand.

Isolation

Isolation stellt sicher, dass sich parallel in Ausführung befindliche Transaktionen nicht gegenseitig beeinflussen können. Das bedeutet, dass eine Transaktion Teile der Datenbasis sperren können muss. Eine solche Sperre führt jedoch zu blockierendem Verhalten und schränkt somit die nebenläufige Benutzung der Datenbank ein.

Dauerhaftigkeit

Dauerhaftigkeit von Transaktionen muss gewährleistet sein, damit bei einem Ausfall Transaktionsdaten nicht verloren gehen. Um Dauerhaftigkeit zu erreichen, wird ein Transaktionslog gefährt, welches im Ausfall verwendet werden kann, um die verlorenen Transaktionen nachzuspielen.

2.4.2 BASE

Als Alternative zum Konsistenzmodell ACID soll nun BASE (**B**asically **A**vailable, **S**oft State, **E**ventual Consistency) vorgestellt werden. BASE ist schwächer als ACID, verfolgt jedoch ebenfalls die Wahrung der Konsistenz in einem System. Es kommt am häufigsten in NoSql-Datenbank zum Einsatz, die die Daten hauptsächlich in Aggregaten zusammenfassen. Innerhalb eines Aggregats können Daten transaktionell verändert werden. Transaktionen über die Daten mehrerer Aggregate werden jedoch nicht atomar sondern sequentiell ausgeführt. Dabei kann das System die Ausführung der Commits der Teilschritte unter hoher Last aufschieben und zu einem späteren Zeitpunkt durchführen. Dadurch erreichen Datenbanksysteme mit BASE-Eigenschaften eine höhere Erreichbarkeit. Diese hohe Erreichbarkeit wird eingetauscht für Atomarität und strenge Konsistenz[5].

Grundsätzlich verfügbar (BA)
Das System ist hochverfügbar. Trotz hoher Last beantwortet das System Anfragen. Die Antworten können unter Umständen einen intermediären Zustand abbilden, der inkonsistent ist.

Weicher Zustand (S) Bevor alle Operationen einer Transaktion ausgeführt wurden, befindet sich das System in einem Übergangszustand. Dieser Übergangszustand kann Konsistenzfehler aufweisen. In Bezug auf ACID verletzt dieser *Soft State* die Atomarität und die Isolation.

Schlussendliche Konsistenz (E) Als schlussendlich konsistent wird das System angesehen, wenn alle Teiloperationen aller Transaktionen abgeschlossen sind. Der Zustandsübergang beginnt in einem konsistenten Zustand, geht in einen inkonsistenten *Soft State* über und befindet sich nach Abschluss aller Teiloperationen in einem konsistenten Endzustand.

2.4.3 CAP-Theorem

Das CAP-Theorem wurde in einem Vortrag auf dem PODC Symposium von Brewer vorgestellt. Darin behauptete Brewer folgende Aussage:

Theorem 1 (CAP-Theorem)**.** *Für einen Webservice ist es unmöglich, folgende drei Eigenschaften zu garantieren:*

- *Konsistenz (Consistency)*
- *Verfügbarkeit (Availability)*
- *Partitionstoleranz (Partition-tolerance)*

Alle drei Eigenschaften sind erwünschenswert und werden von realen Webservices erwartet[6].

Der Beweis für dieses Theorem wurde 2002 von Gilbert und Lynch bewiesen. Das CAP-Theorem ist in Abbildung 2.1 dargestellt. Von den drei Möglichkeiten sind nur zwei realisierbar. In einem verteilten System ist die Partitionstoleranz bereits inbegriffen, weshalb die Wahl zwischen Konsistenz und Verfügbarkeit besteht.

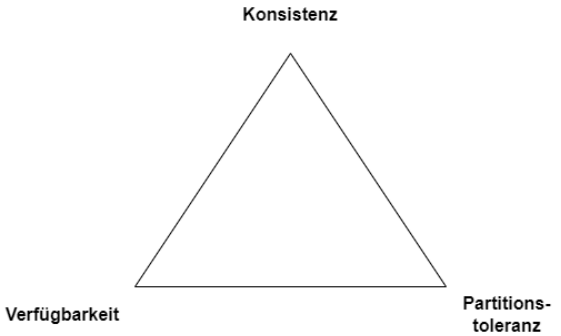

Abbildung 2.1: Visuelle Darstellung des CAP-Theorems

Das Theorem kann dazu verwendet werden, ein verteiltes System in verschiedene Kategorien zu unterteilen.

AP-System

AP-Systeme setzen auf Verfügbarkeit und Partitionstoleranz. Ein Beispiel dafür ist ein soziales Netzwerk. Es ist annehmbar, wenn neue Beiträge nicht live sondern leicht verzögert angezeigt werden. Besteht eine Netzwerkpartition, ist das System trotzdem funktional, arbeitet unter Umständen auf inkonsistenten Daten. Das bedeutet, dass in der einen Partition ein neuer Beitrag angezeigt wird, in der anderen Partition ist dieser Beitrag noch nicht synchronisiert und kann demnach nicht angezeigt werden. Das System ist inkonsistent. Unverfügbarkeit ist für ein soziales Netzwerk jedoch wesentlich schlimmer als ein inkonsistentes System, weshalb solche Anwendung oft als AP-System realisiert werden.

CP-System

Transaktionskritische Systeme, die unter keinen Umständen einen inkonsistenten Zustand erreichen dürfen, sind als CP-System umgesetzt. Dabei wird die Verfügbarkeit im Falle von Netzwerkpartitionen nicht gewährleistet.

2.5 Isolationsanomalien

Werden Systeme durch mehrere Benutzer und somit durch mehrere Transaktionen gleichzeitig verändert, können verschiedene Anomalien auftreten. Diese Anomalien könnnen inkonsistente Datenbankzustände verursachen. Die Anomalien treten dann auf, wenn im Mehrbenutzerbetriebs mehrere gleichzeitig in Ausführung befindliche Transaktionen auf dieselben Ressourcen zugreifen.

2.5.1 Anomalien

Dirty Reads

Dirty Reads treten auf, wenn eine Transaktion einen Datensatz modifiziert und eine zweite Transaktion diese liest, bevor die Änderungen committet wurden. Rollt die erste Transaktion die Änderungen zurück, hat der zweite Prozess nicht existierende Daten gelesen[8].

Lost Updates

Lost Updates treten auf, wenn zwei Transaktionen gleichzeitig den selben Datensatz updaten[8].

Non Repeatable Read

Nonrepeatable Reads treten auf, wenn wiederholte Lesevorgänge einer Transaktionen unterschiedliche Ergebnisse liefern. Dies tritt auf, wenn die gelesenen Datensätze von einer anderen Transaktion verändert wurden[8], [9].

Phantom Read

Phandomreads treten auf, wenn eine Transaktion eine Menge an Datensätzen mit

einer bestimmten Suchbedingung mehrfach abfragt. Die Abfrageergebnisse verändern sich, wenn die Menge der Suchbedingung erfüllenden Datensätze durch eine andere Transaktion verändert wurde[9].

2.5.2 Isolationsstufen

Implementierungen des DBMS lösen dieses Problem häufig über die Isolationsstufen *READ_UNCOMMITTED*, *READ_COMMITTET*, *REPEATABLE_READ* und *SERIALIZABLE*. *READ_UNCOMMITTED* ist die niedrigste und *SERIALIZABLE* die höchste Isolationsstufe. Wählt eine Transaktion eine Isolationsstufe aus, werden für die Dauer der Transaktion Ressourcen gesperrt. Je höher die Stufe, desto mehr Ressourcen werden gesperrt. Deshalb führt die Verwendung einer höheren Isolationsstufe zu einem geringeren Performance, da Transaktionen öfter abgelehnt werden und zurückgerollt werden müssen[10].

2.6 Transaktionsinterface

Anwendungen interagieren über das DBMS mit der DB. Damit die Grenzen der Transaktion eindeutig sind, signalisiert die Anwendung den Beginn einer Transaktion (*Begin*). Nach der Ausführung der gewünschten Operationen muss dem DBMS der erfolgreiche oder erfolglose Abschluss (*Commit* oder *Rollback*) signalisiert werden[4]. Das DBMS bietet dem Entwickler der Anwendung diese drei Kommandos an.

2.7 Verteilte Transaktionen

In verteilten Systemen kann die Anforderung bestehen, dass eine Aktion mehrere Datenquellen verändern muss. Von dieser Aktion wird oft erwartet, dass sie ACID-Eigenschaften gewährleistet. Besonders die Atomarität einer solchen Transaktion stellt eine große Herausforderung in verteilten Systemen dar.
Die Komplexität verteilter Transaktionen entsteht vordergründig durch das potentielle Auftreten von Netzwerkfehlern. Ist ein Transaktionsteilnehmer vorübergehend nicht erreichbar, ist der Zustand der LLT ungewiss. Es ist nicht eindeutig, ob die im nicht erreichbaren Teilnehmer stattfindende Operation ausgeführt wurde oder nicht. Deshalb ist auch die Entscheidung spekulativ, ob die erfolgreich absolvierten Teiloperationen zurückgerollt werden müssen oder nicht. In jedem Bereich der verteilten Transaktion können Netzwerkfehler auftreten.
Systemabstürze stellen ebenfalls eine Herausforderung für verteilte Transaktionen dar. Es ist unklar, ob ein Teilnehmerservice abgestürzt ist bevor oder nachdem die Änderungen committet wurden.
Als Lösung für diese Herausforderungen wurden Protokolle für verteilte Transaktionen entwickelt.

2.8 Zwei-Phasencommit

Der Zwei-Phasen-Commit (2PC) ist ein solches Protokoll. Es besteht aus 2 Phasen: der *Prepare-* und der *Commit-Phase*. Die Transaktion wird von einem Koordinator gesteuert.

Prepare-Phase

Der Koordinator signalisiert allen Transaktionsteilnehmer den Beginn der Transaktion. Die Transaktionsteilnehmer antworten entweder mit einem *ready* oder *failed*. Im Falle eines *ready* muss der Transaktionsteilnehmer garantieren, dass er seine Operation fehlerfrei committen kann. Dabei werden die zu verändernden Ressourcen gesperrt.

Commit-Phase

Der Koordinator wartet bis alle Teilnehmer geantwortet haben. Signalisieren alle Antworten einen Erfolg, fordert der Koordinator alle Teilnehmer zu einem Commit auf. Der Commit wird von allen Teilnehmern bestätigt.
Erhält der Koordinator in der *Prepare-Phase* mindestens ein *failed*, werden im Anschluss alle Teilnehmer zu einem Rollback aufgefordert. Auch diesen müssen wieder alle Teilnehmer bestätigen[11].

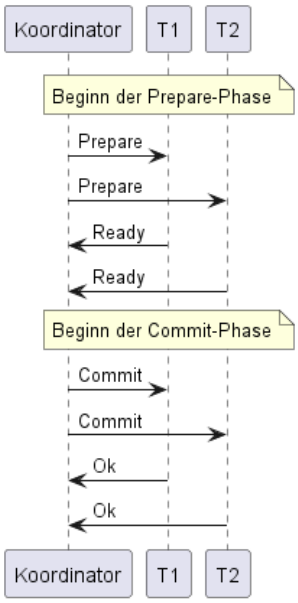

Abbildung 2.2: Sequenzdiagramm für erfolgreichen 2PC

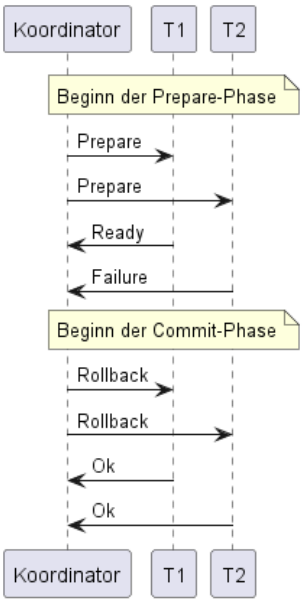

Abbildung 2.3: Sequenzdiagramm für erfolglosen 2PC

Nachteile des 2PC

- **Blockierendes Verhalten**: Da in der *Prepare-Phase* das Versprechen der Teilnehmer besteht, eine Transaktion committen zu können, müssen die zu verändernden Ressourcen gesperrt werden. Dies stellt sicher, dass die Transaktion von anderen Transaktionen isoliert ist.

- **Blockierendes Verhalten bei Ausbleiben einer Antwort**: Treten Netzwerkfehler auf, dann bleiben die Locks so lange bestehen, bis die Netzwerkpartition wieder aufgelöst ist. Auch im 2PC hat der Koordinator keine Kenntnis über den Ausführungsstand im Teilnehmer falls eine Antwort ausbleibt.
- **Starke Bindung**: Der Koordinator ist sehr eng mit den Teilnehmern gekoppelt.
- **Single Point of Failure**: Der Koordinator stellt eine kritische Komponente dar. Fällt der Koordinator aus, bleiben die Locks der Transaktionsteilnehmer bestehen.

3 Das Saga Pattern

3.1 Herkunft des Saga-Patterns

Das Implementierungsmuster einer Saga stammt aus dem 1987 veröffentlichten Paper "Sagas". Darin adressieren Garcia-Molina und Salem die blockierende Eigenschaft von Long Lived Transaction (LLT) und führen das Sagamuster als Problemlösung ein.

3.1.1 Langlebige Transaktionen

Damit die transaktionelle Implementierung eines Prozesses die ACID-Eigenschaften gewährleisten kann, werden Datenbanksperren verwendet. Ein solcher Lock sperrt ausgewählte Ressourcen für die Dauer der Transaktion. Eine LLT repräsentiert einen solchen transaktionellen Prozess, der eine sehr hohe Ausführungsdauer hat. Als Ursache für diese lange Dauer nennen Garcia-Molina und Salem viele langwierige Berechnungsschritte und die Pausierung zum Warten auf Benutzerinput [1].

3.1.2 Auftrennung einer LLT in Teiltransaktionen

Als Lösung für die blockierende Eigenschaft von LLT formulieren Garcia-Molina und Salem das Saga-Pattern, welches die Atomarität eines transaktionellen Prozesses auflöst und die LLT per Teiltransaktionen realisiert. Die LLT wird in kleinere Teiltransaktionen aufgebrochen und sequentiell durchgeführt. Jedes T wird dabei mit einer zugehörigen kompensierenden Transaktion ergänzt, die die Nebenwirkung des Ts neutralisiert. Die Transaktion, die den Rollback durchführt wird dabei als Kompensierung (C) bezeichnet. In Abbildung 3.1 und Abbildung 3.2 sind die Scopes einer LLT bestehend aus 3 Schritten in einem Komponentendiagramm dargestellt. Es ist zu sehen, dass die in einer Transaktion implementierte LLT für sich die ACID-Eigenschaften gewährleistet. Die mittels Saga implementierte LLT verwendet für jeden Operationsschritt eine Transaktion.

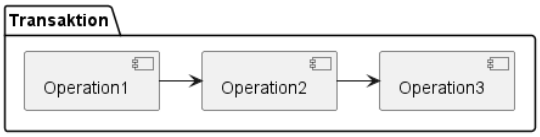

Abbildung 3.1: Komponentendiagramm Scope einer normalen Transaktion mit 3 Operationen

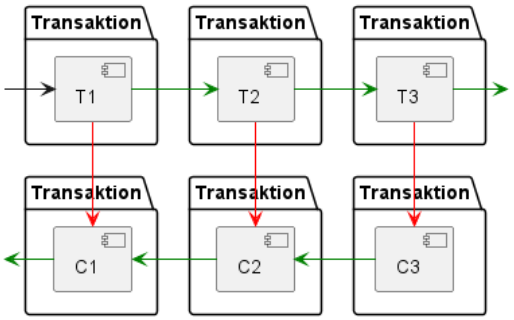

Abbildung 3.2: Komponentendiagramm Scope einer Saga mit 3 Operationen

Die Blockierung einer Ressource besteht nur solange, wie sich eine dieser Teiltransaktionen in Ausführung befindet. Nach Beendigung wird die die Teiltransaktion committet. Die Atomarität geht hier also teilweise verloren. Das Gesamtsystem befindet sich zwischen Abschluss von T_1 und Abschluss von T_3 oder C_1 in einem intermediären Zustand, der andere Transaktionen beeinflussen kann. Somit wird auch die Isolation aufgelöst. Dadurch ist es möglich, dass ein inkonsistenter Zustand in das System eingeführt wird. Es werden alle ACID-Eigenschaften außer der Dauerhaftigkeit aufgebrochen.

3.2 Übertragung des Saga-Patterns auf verteilte Transaktionen

Das von Garcia-Molina und Salem vorgeschlagene Implementierungsmuster baut auf ACID-Transaktionen auf. Es wird davon ausgegangen, dass eine ACID-Transaktion sequentiell kleinere ACID-Transaktion aufgeteilt werden kann. Der Rollback-Mechanismus dieser Teiltransaktionen muss mittels zugehöriger Kompensierungen per Implementierung nachgebildet werden. In Abschnitt 3.1 wurde erläutert, dass das Saga-Muster durch die Auflösung der Atomarität und Isolation und somit auch der Konsistenz keine strenge ACID-Sicherheit gewährleisten kann.

3.2.1 Saga-Patterns und BASE-Eigenschaft

Da nun etabliert wurde, dass das Saga-Pattern grundsätzlich keine strenge ACID-Eigenschaften unterstützt, soll es mit dem alternativen Datenkonsistenzmodell BASE vergleichen werden, da sie sich sehr ähneln. Die Rollback-Logik des Saga-Patterns ist eine Möglichkeit mit *Soft State* umzugehen. Das Aufbrechen der LLT in Teiloperationen ist im BASE-Modell ebenfalls vorhanden, da Veränderungen über mehrere Aggregate nicht atomar sind. Auch das Interesse des Saga-Patterns an Hochverfügbarkeit ist im BASE-Modell als *Basically Available* zu finden.
Das Saga-Pattern kann also als eine Implementierung des BASE-Modells aufgefasst werden.

3.2.2 Anwendung des Patterns auf verteilte Transaktionen

Das Paper "Sagas" befasst sich mit dem Saga-Pattern als Implementierungswerkzeug innerhalb eines zentralen Systems. Die ACID-Eigenschaft wurde lediglich aufgelöst, um das blockierende Verhalten zu umgehen und Hochverfügbarkeit zu erreichen.
Ein verteiltes System stellt sich dem selben Problem. Die Daten eines verteilten Systems sind über mehrere Datenbanken verteilt. Der Zugriff auf diese Daten ist häufig nur über einen dazugehörigen Microservice möglich. Muss also eine LLT Operationen in mehreren Datenbanken durchführen, dann kann das Saga-Pattern auf diese LLT angewendet werden. Somit wird zumindest ein BASE-Datenkonsistenzlevel erreicht.

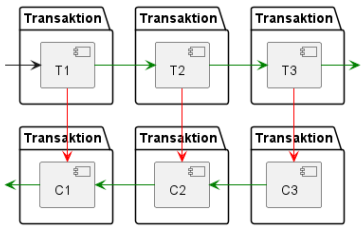

Abbildung 3.3: Komponentendiagramm Scope einer zentralisierten Saga mit 3 Operationen

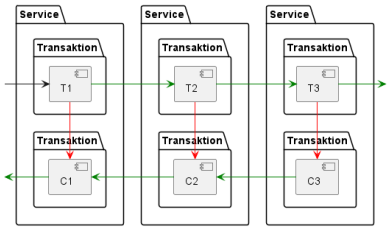

Abbildung 3.4: Komponentendiagramm Scope einer verteilten Transaktion mit 3 Operationen

3.3 Bestandteile des Musters

3.3.1 Funktionsweise

Eine Operation ist im Saga-Pattern eine lokale Transaktion, die in sich geschlossen ist und die ACID-Eigenschaft erfüllen muss. Für eine solche Operation wird gleichzeitig eine Schnittstelle angeboten, die die Veränderungen rückgängig macht. Somit besteht die Möglichkeit eine Operation zu neutralisieren. Es wird also die Anforderung an den Entwickler gestellt, für jede angebotene Operation eine Umkehroperation bereitzustellen, die selbst eine lokale Transaktion darstellt. Im Paper werden lokale Transaktionen, die eine Operation der Transaktion darstellen, als Ts bezeichnet. Die dazugehörigen lokalen Transaktionen werden als Cs bezeichnet.
Zur Ausführung der LLT werden die Ts sequentiell aufgerufen (im Gegensatz zum gleichzeitig ausgelösten Transaktionsstart im 2PC). Tritt bei der Ausführung eines Ts ein Fehler auf, können alle bereits ausgeführten Operationen in ihren Ursprungszustand zurückgesetzt werden, indem in der umgekehrten Reihenfolge die notwendigen Cs aufgerufen werden. Im Fehlerfall wird der Ausgangszustand in allen Services wiederhergestellt und die Atomarität der Transaktion ist gewährleistet. Sind alle Operationen erfolgreich, wird nach Ausführung aller Ts der Endzustand erreicht und die Transaktion hat einen neuen Zustand hergestellt. Sowohl im erfolgreichen als auch im kompensierten Endzustand ist die Konsistenz gewahrt.

3.3.2 Transaktionsteilnehmer

Die verteilte Saga setzt voraus, dass die Operation nicht zentralisiert innerhalb eines geschlossenen Systems umgesetzt werden kann. Die Teiloperationen sind über mehrere Services verteilt. Diese Services werden als Teilnehmerservice bezeichnet. Es liegt in der Verantwortung der Entwickler des entsprechenden Teilnehmerservices, die Teiloperation über eine Schnittstelle anzubieten und korrekt zu implementieren. Darunter fällt auch die Implementierung der dazugehörigen Kompensierung. Dabei verwaltet jeder Service seine eigenen Daten in einer eigenen Datenbank. Dieses Muster nennt sich Database-per-Service-Pattern und ist sehr verbreitet in der Entwicklung von Microservices [12].

3.3.3 Formulieren der Kompensierungstransaktionen

Die einzelnen Operationen, die ausgeführt werden müssen, müssen kompensiert werden können. Eine solches T wird als lokale Transaktion betrachtet, die in einem anderen Service stattfindet. Der Nebeneffekt der Transaktion ist häufig eine Veränderung in der Datenbank. Es wird nun betrachtet, welche verschiedenen Effekte T in der Datenbank haben kann und welchen Effekt das entsprechende C haben muss.

Insert

Äußert sich der Effekt von T in einem Insert, dann ist der kompensierende Effekt von C ein Delete. Alternativ kann ein Soft-Delete implementiert werden, der den Datensatz als ungültig markiert.

Update

Beim Kompensieren von Updates wird zwischen idempotenten und nicht-idempotenten Updates unterschieden. Zur Veranschaulichung sollen die in Codebeispiel 3.1 und Codebeispiel 3.2 dargestellten SQL-Statements betrachtet werden.

```
update stock
set amount = 10
where id = 42 and ...
```

Codebeispiel 3.1: SQL Skript für ein idempotentes Update

```
update stock
set amount = amount - 1
where id = 42 and ...
```

Codebeispiel 3.2: SQL Skript für ein nicht-idempotentes Update

In Codebeispiel 3.1 wird ein Update-Statement ausgeführt, welches den entsprechenden Wert *amount* auf 10 setzt. Dabei geht die Information des vorherigen Zustands verloren. Eine Kompensierung ist hier nur möglich, indem eine zusätzliche History-Tabelle verwendet wird.
In Codebeispiel 3.2 wird ein Update-Statement ausgeführt, welches den entprechenden Wert *amount* um 1 verringert. Um dieses Update zu kompensieren, muss der alte Wert nicht bekannt sein, wenn die LLT Kenntnis von dem Wert der Änderung hat. Es kann eine Kompensierung in Form einer Addition um den selben Wert durchgeführt werden.

Delete

Das Löschen eines Datensätzes kann nur kompensiert werden, wenn der gesamte Eintrag vor dem Löschen gespeichert wurde (History-Tabelle). Falls T nur ein Soft-Delete ausgelöst hat, kann die Markierung der Ungültigkeit zur Kompensierung aufgehoben werden.

Ts ohne Nebeneffekte

Es ist möglich, dass ein T eine Operation durchführt, die zu keiner Nebenwirkung führt. Beispielsweise kann dies der Aufruf einer Schnittstelle sein, um einen Wert zu validieren. In diesem Fall muss die Kompensierung nicht implementiert werden.

Es ist hervorzuheben, dass der Effekt eines Ts neben Änderungen in der Datenbank oder Aufrufe von anderen Schnittstellen auch reale Geschäftsprozesse auslösen können. Ein solcher Prozess kann unter Umständen nicht kompensierbar sein. Hier kann auch weiter differenziert werden.

Nicht kompensierbare Ts

Ist der Effekt von T die Versendung eines Briefs, so kann diese Versendung nicht kompensiert werden. Ein Folgebrief kann jedoch als Kompensierung angesehen werden, die den ausgelösten Effekt neutralisiert. Im Folgebrief können beispielsweise Anweisungen stehen, die den Empfänger informieren, dass der vorherige Brief als ungültig angesehen werden kann. Der Effekt von T kann hier als kompensiert angesehen werden.
Es gibt jedoch auch Effekte, die nicht kompensierbar sind und im Scheitern einer Saga resultieren. In solchen Fällen kann das System in einen inkonsistenten Zustand überführt werden. Dieses Verhalten tritt immer dann auf, wenn der Effekt einer Transaktion in einer endgültigen Aktion resultiert. Garcia-Molina und Salem nennen als Beispiel für ein nicht kompensierbares T das Starten einer Rakete [1, p.257].

3.3.4 Komponenten des Saga-Patterns

Garcia-Molina und Salem nennt folgende Komponenten als Grundbausteine des Saga-Patterns:

- Transaktionen und Kompensierungen
- Transaction Execution Component
- Saga Execution Component

Transaktionen und Kompensierungen

Das Aufteilen einer Transaktion in die entsprechenden Teiltransaktionen sowie das Verhalten im Falle eines Fehlers wurden bereits in erläutert. Nach Übertragung des Musters auf ein verteiltes System finden diese Ts und Cs innerhalb eines Teilnehmerservices statt.

Transaction Execution Component

Die Transaction Execution Component (TEC) wird beschrieben als eine Komponente, die die Ausführung der zu einer Teiltransaktion gehörenden Aktionen ausführt. Eine an die TEC gestellte Anfrage für die Ausführung einer Teiltransaktion wird atomar verarbeitet. In der verteilten Implementierung einer Saga befindet sich die TEC innerhalb eines jeden Teilnehmerservices. Auf Anfrage wird ein T oder ein C ausgeführt. Alle dafür erforderlichen Datenbankoperationen bestehen innerhalb eines herkömmlichen ACID-Transaktionskontexts.

Saga Execution Component

Die Saga Execution Component (SEC) ist die äußerste Schicht aller Sagakomponenten. Sie ist zuständig für die Ausführung der Saga. Dazu gehört das sequentielle Aufrufen

aller Teiltransaktionen und die Reaktion mit Kompensierungstransaktionen im Falle von Fehlschlägen. Eine weitere Aufgabe der SEC ist das Neustarten der Saga im Falle eines Absturzes.
Die SEC verwendet ein Transaktionslog, um den Prozessablauf zu steuern. Dieses Log enthält für jede Saga die Liste an ausgeführten Schritten. Eine Liste der vergangenen Werte kann auch Bestandteil des Logs sein, damit im Falle von Kompensierungen der ursprüngliche Stand gesetzt werden kann.

3.4 Formalisierung von Sagas

Die SEC wurde als die Komponente definiert, die für die Ausführung und Steuerung des Saga-Prozesses zuständig ist. Die Steuerung des Ablaufs auf Basis eines dazugehörigen Transaktionslogs ist sehr ähnlich zur Funktionsweise eines Deterministischer endlicher Automat (DEA). Deshalb soll nun eine Formalisierung von Sagas als DEA vorgenommen werden.

3.4.1 Darstellung eines DEA

Ein DEA wird dargestellt als Tupel mit folgenden Elementen:

- Q: Zustandsmenge
- Σ: endliches Eingabealphabet
- $\delta : Q \times \sum \rightarrow Q$: Übergangsrelation
- $q_0 \in Q$: Startzustand
- $F \subseteq Q$: Menge an akzeptierenden Zuständen

3.4.2 Saga als formale Sprache

Im vorherigen Abschnitt wurde die Saga Execution Component definiert als ein Tupel aus Transaktionslog und Zustandsautomat. Nun soll dieses Tupel in einen DEA überführt werden. Ein solcher DEA A_{Saga} akzeptiert die Sprache L_{Saga}, die alle gültigen Wörter enthält, die eine Saga darstellen.
Das Eintabealphabet Σ ist die Menge aller Elemente, die im Transaktionslog auftauchen können. Somit kann jedes Transaktionslog als Eingabewort aufgefasst werden.
Somit ist die von A_{Saga} akzeptierte Sprache $L_{Saga} = L(A_{Saga})$:

$$\forall w \in \Sigma^\star : w \in L(A_{Saga}) \iff w \in L_{Saga}$$

3.4.3 Überführung einer Saga in einen DEA

Um eine Saga in einen DEA überführen zu können, müssen zuerst einige Definitionen vorgenommen werden. Die Unterscheidung zwischen Ts und Cs wird im Modell eines Zustandsautomaten per Zustand ausgedrückt. Es muss also eine Abstrahierung vorgenommen werden, die Ts und Cs vereinigt. Diese Abstrahierung wird im Folgenden als *Aktion A* bezeichnet. Eine solche Aktion a_n wird immer im entsprechenden Zustsand $q_n \in Q$ ausgeführt. In der folgenden Erläuterung kann die Zustandsmenge Q mit der Menge $T \cup C$ gleichgesetzt werden.
Das Eingabealphabet Σ drückt aus, welche möglichen Ergebnisse eine Aktion haben kann. Eine Aktion kann einerseits ein Aufruf einer externen Schnittstelle sein. Die Antwort dieser Schnittstelle kann das Ergebnis in unterschiedlichen Formen ausdrücken. Das können beispielsweise folgende Ausdrucksformen sein:

- Http-Statuscode
- Custom Http-Responsebody

Diese sind üblicherweise in einer Schnittstellendefinition aufgelistet. Im Folgenden wird davon ausgegangen, dass alle möglichen Antworten einer Http-Schnittstelle per Http-Statuscode ausgedrückt werden. Es wird ein Typ definiert, der für jede Aktion alle möglichen Http-Statuscodes enthält:

$$API - Ergebnis\ AE \in \{tn_{200}, tn_{201}, tn_{400}, tn + 1_{200}, tn + 1_{400}, tn + 1_{409}, ...\}.$$

Eine Aktion kann neben dem Aufruf einer Schnittstelle eine interne Verarbeitung sein. Das könnte beispielsweise eine Prüfung auf Vorhandensein eines Feldes in einer vorangegangenen Schnittstellenantwort sein. Eine solche Aktion wird definiert:

$$Internes\ Prozessergebnis\ IPE \in \{tn_{Success}, tn_{Failure}, tn + 1_{Success}, tn + 1_{Failure}, ...\}.$$

Ein Ergebnis einer Aktion wird also definiert als:

$$Ergebnis\ E = AE \cup IPE.$$

Das Eingabealphabet beinhaltet Elemente aus dem Ergebnistyp:

$$\Sigma = Ergebnis.$$

Ein Übergang von einem Zustand in den Folgezustand drückt somit aus, dass die Saga eine Aktion ausgeführt hat und dem Ergebnis entsprechend einen Zustandswechsel durchgeführt hat.
Der Startzustand q_0 ist die erste auszuführende Transaktion.
Ein Endzustand q_{f1} wird erreicht, nachdem die letzte auszuführende Transaktion erfolgreich beendet wurde. Ein weiterer Endzustand q_{f2} wird erreicht, nachdem die letzte Kompensierung erfolgreich beendet wurde. Der letzte Endzustand q_{f3} wird erreicht, nachdem die erste Kompensierung erfolglos beendet wurde.

Konfiguration
Die Ausführung eines DEA kann mittels Konfigurationen dargestellt werden. Eine Konfiguration K ist definiert als:

$$K = (q, w) \in Q \times \Sigma^\star$$

Der Automat wechselt in einen Folgezustand, indem er ein Element aus dem Eingabewort abarbeitet und eine passende Übergangsrelation in δ findet. Somit gilt:

$$q1, q2 \in Q \land u \in \Sigma \land v \in \Sigma^\star : (q_1, u \circ v) \vdash (q_2, v) \implies \delta(q_1, u) = q_2$$

Außerdem können mehrere Konfigurationsübergänge mittels $\vdash^\star$ dargestellt werden:

$$K_1 \vdash^\star K_2 \implies K_1 = K_2 \lor \exists K : K_1 \vdash K \land K \vdash^\star K_2$$

3.4.4 Betrachtung des Zustands nach Erfolg/Misserfolg

Der Zustand des Systems soll nun in folgenden Fällen betrachtet werden:

1. Erfolgreicher Ablauf einer Saga
2. Scheitern der Saga nach n Schritten
3. Scheitern der Saga nach n Schritten und Scheitern der Kompensierung nach m Schritten

Die Ausführung der Saga als DEA soll an folgendem Beispiel illustriert werden:
$Saga = (Q, \Sigma, \delta, q_0, F)$ mit

$$\begin{aligned}
Q &= \{q_{t1}, q_{t2}, q_{t3}, q_{c1}, q_{c2}, q_{c3}, q_{f1}, q_{f2}, q_{f3}\} \\
\Sigma &= \{t1_{200}, t1_{400}, t2_{Success}, t2_{Failure}, t3_{200}, t3_{400}, c1_{200}, c1_{400}, c2_{Sucess}, c2_{Failure}, c3_{200}, c3_{400}\} \\
\delta &= \{((q_{t1}, t1_{200}), q_{t2}), ((q_{t2}, t2_{Sucess}), q_{t3}), ((q_{t3}, t3_{200}), q_{f1}), ((q_{t1}, t1_{400}), q_{f2}), \\
&\quad ((q_{t2}, t2_{Failure}), q_{c1}), ((q_{t3}, t3_{400}), q_{c2}), ((q_{c1}, c1_{200}), q_{f2}), ((q_{c2}, c2_{Sucess}), q_{c1}), \\
&\quad ((q_{c3}, c3_{200}), q_{c2}), ((q_{c1}, c1_{400}), q_{f3}), ((q_{c2}, c2_{Failure}), q_{f3}), ((q_{c3}, c3_{Failure}), q_{f3})\} \\
q_0 &= q_{t1} \\
F &= \{q_{f1}, q_{f2}, q_{f3}\}
\end{aligned}$$

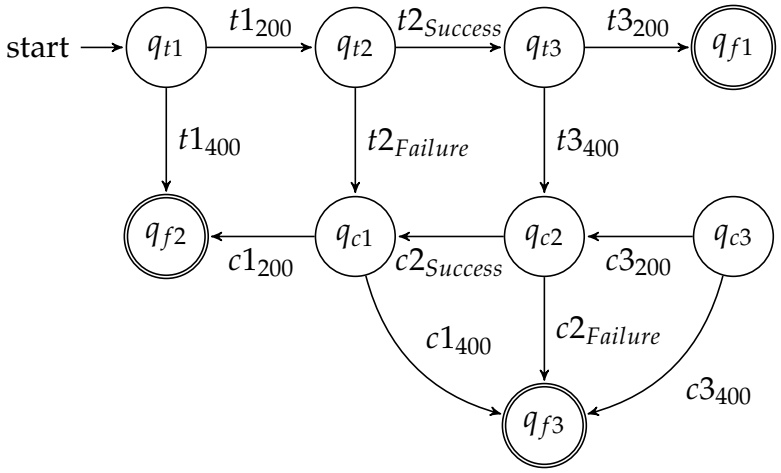

Abbildung 3.5: Darstellung einer Saga als Deterministischen endlichen Automaten

Endzustand q_{f1}
Im Folgenden wird davon ausgegangen, dass die Aktionen der Zustände q_{t1}, q_{t2} und q_{t3} in einem erfolgreichen Ergebnis resultieren. Somit wird am Ende der Endzustand q_{f1} erreicht. Dieser Zustand drückt einen erfolgreichen Durchlauf einer Saga aus. Als Darstellung für das leere Element wird # verwendet.
Das Eingabewort $e_1 \in \Sigma^\star$ ist $t1_{200} \circ t2_{Success} \circ t3_{200} \circ \#$. Die Konfigurationsübergänge für e_1 sind:

$$\begin{aligned}(q_{t1}, t1_{200} \circ t2_{Success} \circ t3_{200} \circ \#) \\ \vdash (q_{t2}, t2_{Success} \circ t3_{200} \circ \#) \\ \vdash (q_{t3}, t3_{200} \circ \#) \\ \vdash (q_{f1}, \#)\end{aligned}$$

Endzustand q_{f2}
Es wird nun davon ausgegangen, dass bei der Aktion im Zustand q_{t3} ein Ergebnis $t3_{400}$ erfolgt. Ein solches Ergebnis führt dazu, dass der Zustand q_{c2} erreicht wird. Hier wird davon ausgegangen, dass die Aktionen q_{c2} und q_{c1} erfolgreiche Ergebnisse haben. Das Eingabewort $e_2 \in \Sigma^\star$ ist $t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200}$.
Die Konfigurationsübergänge für e_2 sind:

$$\begin{aligned}(q_{t1}, t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#) \\ \vdash (q_{t2}, t2_{Success} \circ t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#)) \\ \vdash (q_{t3}, t3_{400} \circ c2_{Success} \circ c1_{200} \circ \#)) \\ \vdash (q_{c2}, c2_{Success} \circ c1_{200} \circ \#)) \\ \vdash (q_{c1}, c1_{200} \circ \#)) \\ \vdash (q_{f2}, \#))\end{aligned}$$

Endzustand q_{f3}
Zuletzt soll der Zustand q_{f3} betrachtet werden. Dafür soll die Aktion in q_{t3} das Ergebnis $t3_{400}$ haben. Danach schlägt die Aktion q_{c2} fehl und liefert das Ergebnis $c2_{Failure}$. Das Eingabewort $e_3 \in \Sigma^\star$ ist $t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Failure} \circ \#$.
Die Konfigurationsübergänge für e_3 sind:

$$\begin{aligned}
&(q_{t1}, t1_{200} \circ t2_{Success} \circ t3_{400} \circ c2_{Failure} \circ \#) \\
\vdash\ &(q_{t2}, t2_{Success} \circ t3_{400} \circ c2_{Failure} \circ \#) \\
\vdash\ &(q_{t3}, t3_{400} \circ c2_{Failure} \circ \#) \\
\vdash\ &(q_{c2}, c2_{Failure} \circ \#) \\
\vdash\ &(q_{f3}, \#)
\end{aligned}$$

3.4.5 Recovery-Mechanismen

Eine Saga, die in der Ausführung einer Transaktion fehlschlägt, wechselt nach der Definition in die entsprechende Kompensierung und versucht, alle bis dahin ausgeführten Transaktionen zu kompensieren. Somit wird der Anfangszustand des Systems wiederhergestellt. Dieses Verhalten wird als Backward-Recovery bezeichnet.
Neben der Backward Recovery wird ein weiteres Verhalten vorgeschlagen, welches Forward-Recovery genannt wird. Das Ziel der Forward Recovery ist es, seltener in einem erfolglosen Endzustand zu gelangen. Im Modell der hier aufgestellten DEA-Saga sind das Zustände q_{f2} und q_{f3}. Um dies zu erreichen, werden Save-Points definiert. Ein Save-Point stellt einen Zustand dar, von dem bei einem Systemabsturz oder einem erfolglosen Ergebnis die Ausführung weitergeführt werden kann. Es wird im Fehlerfall Backward-Recovery bis zum nächsten Save-Point ausgeführt. Wird dieser erreicht, werden alle noch fehlenden Ts ausgeführt, um zum erfolgreichen Endzustand zu gelangen. Das bedeutet, dass von der Kompensierungskette zurück auf die Transaktionskette gesprungen wird.

Backward Recovery
Der DEA einer Saga, die Backward-Recovery implementiert, ist im vorherigen Abschnitt beschrieben.

Forward Recovery
Forward-Recovery ist auf verschiedenen Wegen erreichbar. Der erste Ansatz beinhaltet die Verwendung eines Save-Points. Der DEA aus Abschnitt soll um einen Checkpoint und Forward Recovery ergänzt werden. Es wird ein weiterer Zustand eingeführt, der nach erfolgreichem Ergebnis von q_{t1} erreicht wird. Der Checkpoint wird hier dargestellt als ein interner Prozessschritt q_{sp1} und hat somit die möglichen Ergebnisse $\in \{sp1_{Success}, sp1_{Failure}\}$. Es ist zu sehen, dass dieser DEA eine mögliche Endlosschleift zulässt. Wenn q_{sp1} erreicht wird und in q_{t2} oder q_{t3} immer ein erfolgloses Ergebnis auftritt, darf im Zustand $qsp1$ nur endlich oft der Übergang $sp1_{Success}$ gewählt werden.

Die Funktion f die in q_{sp1} ein internes Prozessergebnis IPE berechnet, sieht so aus:

$f : \mathbb{N} \rightarrow IE$
$maxSavepointExecutionCount \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{sp1}\ während$
$der\ Ausführung\ der\ Saga$

$$f(x) = \begin{cases} IE_{Success}, x < maxSavepointExecutionCount \\ IE_{Failure}, else \end{cases}$$

Die Anzahl an Ausführungen beginnend bei q_{t2} ist begrenzt. Es wird also solange Forward Recovery versucht, bis die Saga erfolgreich ist oder das Oberlimit $maxSavepointExecutionCount$ erreicht wird. Wenn dieses Oberlimit erreicht ist, wird die Forward Recovery aufgegeben und in den Zustand q_{c1} gewechselt.

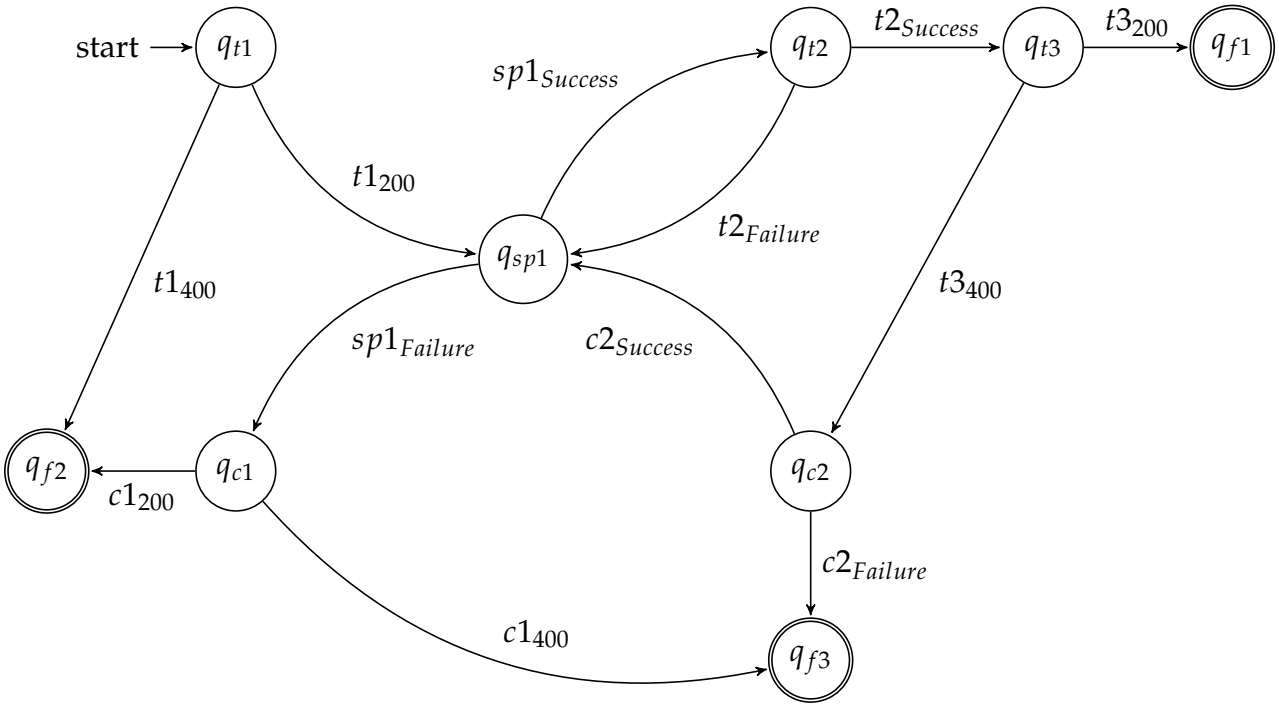

Abbildung 3.6: Forwardrecovery in einem DEA

Forward-Recovery kann alternativ auch als Retry interpretiert und somit ohne Save-Points realisiert werden. Einen solcher Retry kann sehr einfach in jedem Zustand ergänzt werden. Dazu wird eine Kante hinzugefügt, die im gleichen Zustand bleibt. Die Kante, die zuvor ein erfolgloses Ergebnis ausgedrückt hat, drückt nun ein Scheitern oberhalb des Retrylimits aus.
Der Typ Ergebnis wird dafür definiert als:

$$Ergebnis\ E = \{t1_{Success}, t1_{Failure}, t1_{FinalFailure}, ...\}$$

Die Funktion fn_{AE}, die in dem jeweiligen Zustand q_{tn} das entsprechende API-Ergebnis AE berechnet, ist:
$fn_{AE} : \mathbb{N} \times AE \rightarrow E$
$maxSavepointExecutionCount_n \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{tn}$

während der Ausführung der Saga

$$fn_{AE}(x,y) = \begin{cases} En_{Success}, y = tn_{200} \\ En_{Failure}, y \neq tn_{200} \wedge x < maxSavepointExecutionCount_n \\ En_{FinalFailure}, else \end{cases}$$

Die Funktion fn_{IE}, die in dem jeweiligen Zustand q_{tn} das entsprechende interne Prozessergebnis IPE berechnet, ist:
$fn_{AE} : \mathbb{N} \times IPE \rightarrow Ergebnis$
$maxSavepointExecutionCount_n \in \mathbb{N} : Anzahl\ des\ Erreichens\ von\ q_{tn}$
während der Ausführung der Saga

$$fn_{AR}(x,y) = \begin{cases} En_{Success}, y = tn_{Success} \\ En_{Failure}, y \neq tn_{Failure} \wedge x < maxSavepointExecutionCount_n \\ En_{FinalFailure}, else \end{cases}$$

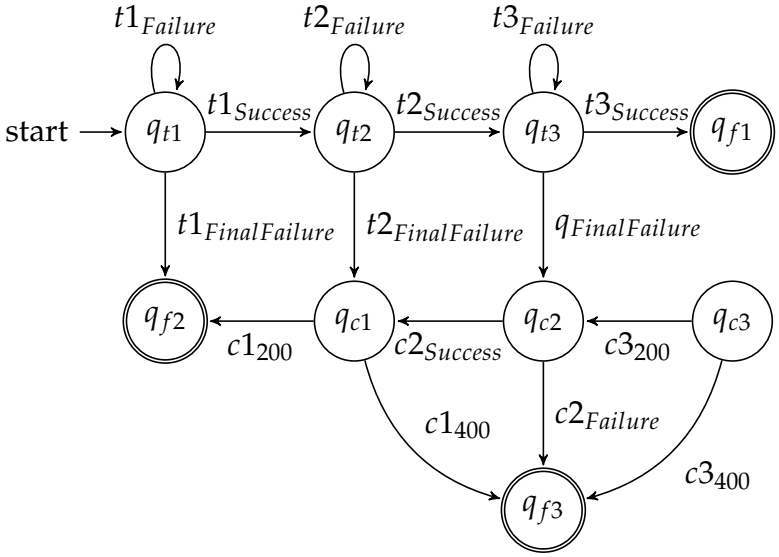

Abbildung 3.7: Backwardrecovery in einem DEA

Es kann außerdem verboten werden, dass in einer Implementierung von Forward-Recovery der Fall verboten wird, der zu einer Backward-Recovery führt. Dabei wird erreicht, dass es nur einen gültigen Endzustand gibt. Dieser Endzustand drückt einen erfolgreichen Abschluss der Saga aus. Dabei ist zu beachten, dass das wiederholte Ausführen einer Aktion schlussendlich zu einem erfolgreichen Ergebnis führen muss. Der DEA für dieses Verhalten sieht so aus:

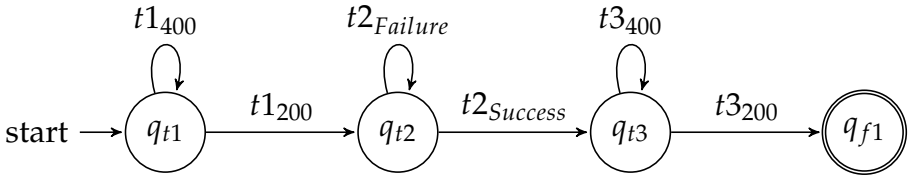

Abbildung 3.8: Erzwungene Forwardrecovery in einem DEA

Es ist zu sehen, dass in diesem DEA keine Zustände enthalten sind, die eine Kompensierungsaktion ausdrücken. Somit geht in dieser Implementierung der Gedanke der Kompensierung verloren, der eine zentrale Rolle im Saga-Pattern innehat. Bei wiederholtem Auftreten eines erfolglosen Ergebnisses endet die Saga nie.

Voraussetzung für Forward-Recovery

Damit eine Forward-Recovery sinnvoll ist, muss die Möglichkeit bestehen, dass ein gescheitertes T bei erneutem Ausführen ein erfolgreiches Ergebnis liefert. Das ist abhängig von der Semantik des Ergebnisses. Ist ein T beispielsweise ein Aufruf einer Schnittstelle zum Buchen eines Hotels, so könnten erfolglose Ergebnisse beispielsweise folgende Bedeutungen haben:

1. Hotel ist im angefragten Zeitraum ausgebucht
2. Hotel ist im angefragten Zeitraum im Betriebsurlaub

Im ersten Fall ist eine Forward Recovery möglich. Wenn ein andere Kunde seine Reservierung storniert, ist es es möglich, dass bei erneutem Anfragen eine Reservierung zustande kommt, die vorher abgelehnt wurde.
Im zweiten Fall ist Forward-Recovery ohne Effekt. Wenn eine Hotelbuchung für einen Zeitraum angefragt wird, in dem das Hotel im Betriebsurlaub ist, wird auch bei wiederholter Anfrage keine Buchung zustande kommen.

3.4.6 Implementierungsformen des Patterns

Um eine Saga als Microservice-System zu implementieren, gibt es zwei verschiedene Herangehensweisen. Die zwei Formen der Implementierung werden als Orchestrierung und als Choreografie bezeichnet. Beide Ausprägungen des Saga-Patterns verfolgen denselben Zweck: den Gedanken, eine globale verteilte Transaktion in einem verteilten System in lokale Teiltransaktionen aufzuteilen, die mittels passender Kompensierung zurückgerollt werden können.
Die zwei Ausprägungen unterscheiden sich hauptsächlich in der Softwarearchitektur. Es ist zu beachten, dass beide Implementierungen denselben Geschäftsprozess abbilden können und somit als äquivalent angesehen werden können.
Im Folgenden sollen die beiden Implementierungsansätze vorgestellt werden. Um die Unterschiede zu verdeutlichen, soll in den nachfolgenden Erläuterungen von einem Geschäftsprozess ausgegangen werden, der Ts enthält, die Teil einer verteilten, globalen Transaktion sind. Jedes T soll eine andere Schnittstelle aufrufen. Jedes T hat ein entsprechendes C zugeordnet.

Orchestration

Die Orchestrierung zentralisiert die Logik für eine Saga in einem einzigen Service. Dieser Service wird als Koordinator oder Orchestrator bezeichnet. Der Koordinator ist verantwortlich für die Einhaltung der Transaktionsanforderungen. Er ruft aktiv die restlichen teilhabenden Services auf und muss die Ergebnisse der Aufrufe auswerten. Die teilhabenden Services haben nur Verantwortung für die Korrektheit der Prozessierung ihre eigenen Servicegrenzen. Ein solcher vom Koordinator aufgerufener Service hat keine Kenntnis vom ablaufenden Geschäftsprozess.

Der Orchestrator stellt einen Prozessmanager dar. Als solcher muss dieser Service garantieren, dass eine gestartete Saga nicht abbricht. Damit ein Absturz des Orchestrators dies gewährleisten kann, muss der Zustand der gestarteten Saga persistiert werden. Häufig wird das Transaktionslog in einer Datenbank gespeichert und erlaubt damit die Weiterführung der Saga auch nach Absturz der Anwendung.

Choreografie

Bei der Choreographie gibt es keinen koordinierenden Service. Alle teilhabenden Services kennen den Ablauf des Geschäftsprozesses. Die Logik ist über alle Services verteilt. Ein Service ist auch hier für die Korrektheit der Prozessierung innerhalb der eigenen Servicegrenzen verantwortlich. Zusätzlich muss jeder Service nach der Prozessierung den Prozess weiterführen. Dazu gehören sowohl mögliche weitere Transaktionen als auch mögliche Kompensierungsaufrufe.

Kommunikationsstrategien

Die Orchestration unterstützt sowohl synchrone als auch asynchrone Kommunikation mit den teilhabenden Services.
Bietet ein an der globalen Transaktion teilhabender Service eine synchrone Schnittstelle zur Verfügung, muss der Koordinator warten, bis der aufgerufene Service eine Antwort liefert und ist solange blockiert. Bei einem Ausfall des aufgerufenen Services hat der Koordinator keine Möglichkeit, die Transaktion fortzufahren. Die Verfügbarkeit aller Services zum Aufrufzeitpunkt ist Voraussetzung für den erfolgreichen Abschluss einer orchestrierten Saga. Dafür ist dem Koordinator in einem solchen Fall die Unerreichbarkeit des Services bekannt und kann entsprechend reagieren.
Des weiteren kann ein Service eine asynchrone Schnittstelle zur Verfügung stellen. Diese Schnittstelle kann eine Implementierung des asynchronen Request-Response Musters sein (Polling Pattern, Callback Pattern). Um eine asynchrone Request-Response Schnittstelle zu verwenden muss der Orchestrator das entsprechende Protokoll des Musters einhalten. Der Vorteil in der Verwendung asynchroner Kommunikation liegt darin, dass der Orchestrator nicht blockiert. In der Zeit zwischen der Platzierung der Anfrage und dem Erhalt der Antwort kann der Orchestrator die Prozessierung der aktuellen Saga pausieren und mit der Verarbeitung anderer Anfragen fortfahren. Der Vorteil dieser Implementierungen ist die Entkopplung von Request und Response. Das zahlt sich in Fällen aus, in denen die Verarbeitung der Anfrage einen längeren Zeitraum in Anspruch nimmt.
Die Implementierung eines asynchronen Request-Response Musters ist wesentlich komplizierter als die Entwicklung einer synchronen Schnittstelle. Deshalbt sollte dies als Implementierung einer lokalen Transaktion unter Verwendung einer Orchestrierung nur in Szenarien gewählt werden, die die Entkopplung von Anfrage und Antwort voraussetzen.
Schlussendlich bietet die Orchestrierung die Möglichkeit, asynchrone Messaging Komponenten zu verwenden. Anstatt direkt miteinander zu kommunizieren platziert der Koordinator die Anfrage als Event in einer Messaging-Middleware und kann mit der Prozessierung der Saga pausieren. Der angefragte Service erhält dieses Event und kann

eine beliebig lang andauernde Verarbeitung ausführen. Nachdem die Verarbeitung abgeschlossen ist, kann die Antwort wiederum als Event in der Middleware platziert werden. Der Koordinator erhält dieses Event und kann darin das Ergebnis ablesen.
Um eine Saga mittels Choreographie zu implementieren, sollte asynchrones Messaging verwendet werden. Da die Geschäftslogik über alle Komponenten verteilt ist, ist selten ein Service am Ergebnis des nächsten Transaktionsschrittes interessiert. Ein Service S_1 verarbeitet seinen Teil der Transaktion und informiert den nächsten Service S_2 über den Erfolg der Berechnung. S_2 ist so implementiert, dass er die Logik für seine eigenen Berechnungen kennt. Somit muss S_1 nicht über den Erfolg informiert werden. Ein Erfolg von der in S_2 ablaufenden Transaktion endet in einem Event für einen nachfolgenden Service S_3. Die Kommunikation ist hier nicht auf ein Request-Response Muster ausgelegt, es werden Einweg-Nachrichten genutzt. Die Ausnahme ist ein erfolgloses Ergebnis in S_2. In diesem Fall wird S_3 nicht per Event informiert. Es wird lediglich S_1 mit einem erfolglosen Ergebnis benachrichtigt. Als Reaktion auf dieses Event kann S_1 mit Forward- oder Backward-Recovery reagieren.
Die Implementierung einer Choreographie per Request-Response Muster ist nicht unmöglich. S_1 ruft S_2 per synchroner oder asynchroner Request-Response Schnittstelle auf. Daraufhin erhält S_1 eine Antwort mit dem Ergebnis von der Berechnung von S_2. Bei einem Erfolg findet in S_1 jedoch keine Reaktion statt. Lediglich bei einem Misserfolg muss S_1 Kenntnis vom Ergebnis der Transaktion in S_2 haben. Somit hat die Verwendung einer Response nur einen Nutzen, falls ein Misserfolg vorliegt.
Des Weiteren hat die Verwendung einer synchronen Kommunikation in der Implementierung der Choreographie den Nachteil, dass es zu Blockierungen aller teilhabenden Services führt. Auch S_2 ruft S_3 synchron auf. Somit muss S_2 warten, bis die Response in S_3 erfolgt. Erst danach kann S_2 die Response für S_1 absenden. Dieses Verhalten wird als Aufrufkaskadierung bezeichnet und wirkt sich sowohl auf den Fall eines Erfolgs als auch den eines Misserfolgs aus.
Aus den genannten Gründen ist es zu empfehlen, bei der Implementierung einer Saga per Choreographie eine eventbasierte Architektur mit asynchronen Messaging-Komponenten zu verwenden.

4 Versuchsvorbereitung

4.1 Problemstellung

In **??** wurde das Saga-Pattern als ein Implementierungsmuster vorgestellt, welches ermöglichen soll, die ACID-Anforderungen in einem verteilten System nachzubilden. Ein verteiltes System steht immer vor der Herausforderung von Netzwerkfehlern. Verwenden Transaktionsteilnehmer eines Systems die Request-Response-Kommunikation, so besteht immer die Möglichkeit, dass einzelne Nachrichten ihr Ziel nicht erreichen. Wenn eine solche Kommunikation deterministischer Natur ist, kann der Sender seinen Request ohne Gefahr wiederholen. Die im Saga-Pattern verwendeten lokalen Transaktion stellen jedoch keine deterministische Abfrage dar, sondern verfolgen das Ziel eines Zustandswechsels des Empfängers. Wiederholt der Sender seine Requests, führt dies zu ungewünschten Nebenwirkungen.
Es wird folgende These formuliert:

These. *Mittels Saga-Pattern implementierte langlebige Transaktionen (LLT) ermöglichen ausfallsichere Konsistenz in Microservicesystemen, die per Request-Response Pattern kommunizieren.*

4.2 Zielstellung

Die folgenden Kapitel dienen dem Zweck, das Saga-Pattern hinsichtlich Systemkonsistenz zu untersuchen. Dabei wird davon ausgegangen, dass jegliche Kommunikation per Request-Response-Protokolle abläuft.

1 Es ist die Frage zu beantworten, unter welchen Bedingungen ein Microservice-System, welches mittels Saga-Pattern implementiert wurde, eine LLT abbilden kann.

2 Es sind Fehlerquellen und Fehlertypen zu identifizieren, die einen inkonsistenten Systemzustand verursachen können. Es sollen Lösungen im Rahmen des Saga-Patterns formuliert, implementiert und evaluiert werden.

3 Es soll eine Antwort darauf gefunden werden, welche Kriterien eine Schnittstelle erfüllen muss, um an einer LLT teilnehmen zu können.

4 Es soll beantwortet werden, ob aufgrund Netzwerkpartitionen auftretende Fehler in die Fehlerbehandlung des Saga-Patterns integrierbar sind.

4.3 Ausgangspunkt

Es soll ein Geschäftsvorgang mittels Saga-Pattern in einem Microservicesystem entworfen, implementiert und bewertet werden. Die gewählte Geschäftsvorgang soll als LLT aufgefasst werden und eine verteilte Transaktion abbilden.
Es soll eine Implementierungsform des Saga-Patterns gewählt werden. In der Zielstellung wird ein System gefordert, welches Request-Response-Kommunikation verwendet. Somit kann die Verwendung von Messagingkomponenten ausgeschlossen werden. Die auf diese Komponenten ausgerichtete Implementierung per Choreografie wird deshalb nicht gewählt.
Als Implementierungsform des Saga-Patterns wird für diesen Versuch die Orchestrierung gewählt.

4.4 Methodik

In diesem Abschnitt soll das Vorgehen bei der Bearbeitung des Problems erläutert werden.

4.4.1 Entwurf eines Bestell- und Lieferprozesses

Zur Untersuchung der Probleme soll ein Geschäftsprozess entworfen und als Saga-System implementiert werden. Der Durchlauf durch diesen Geschäftsprozess soll als LLT aufgefasst werden.
Es besteht die Anforderung an den zu entwerfenden Prozess, dass dieser durch eine Menge von lokalen Transaktionen abbildbar sein muss. Außerdem muss es möglich sein, Kompensierungen für jede dieser Transaktionen zu formulieren.
Als abzubildender Geschäftsprozess soll ein Bestell- und Liefervorgang eines Online-Shops dienen. Der Bestellvorgang soll durch das Platzierung einer Bestellung ausgelöst werden. Die Benutzeroberfläche gehört nicht zum Scope des umzusetzenden Systems.
Als Ausgangspunkt soll folgender Geschäftsprozess dienen:

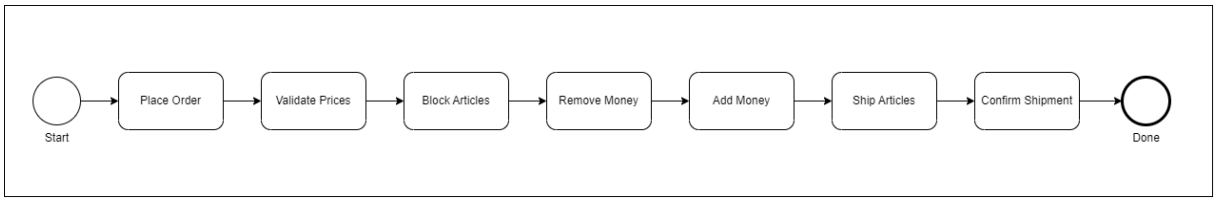

Abbildung 4.1: Vereinfachte Darstellung eines Bestell- und Lieferprozesses

Die zum Prozess gehörenden Schritte sind folgende:

Entgegennehmen der Bestellung

Die Bestellung wird über ein imaginäres Frontend entgegengenommen. Dieses Frontend baut einen Request auf und sendet diesen per Http-Schnittstelle an das Backend. Dort wird der Request entgegengenommen und muss alle für die Abwicklung der Bestellung

erforderlichen Daten enthalten. Dazu gehören der bestellende Nutzer, die geforderten Artikel und die Zahlungsinformationen. Beim Entgegennehmen wird die Bestellung initialisiert.

Validierung des Preises

Der Bestellungsrequest enthält eine Liste von den gewünschten Produkten und dem bekannten Preis pro Produkt. Um zu überprüfen, ob der dem Nutzer (dem Frontend) bekannte Preis mit dem aktuellen Preis übereinstimmt, muss dieser validiert werden.

Blockieren der Artikel

Die geforderten Artikel sollten für diese Bestellung reserviert werden, bis der Bestellvorgang abgeschlossen ist. In einem Online-Shop wird angezeigt, wieviele Artikel auf Lager vorrätig sind. Beim Blockieren der Artikel wird dieser Betrag verändert. Somit sehen andere Nutzer nach Ausführung dieses Schrittes den aktuellen Wert der vorrätigen Artikel.

Zahlungsabwicklung

Der berechnete Preis der Bestellung muss vom Konto des Kunden abgebucht werden. Das Konto des Händlers erhält denselben Betrag gutgeschrieben. Die Konten des Kunden und des Online-Shop-Besitzers müssen nicht bei derselben Bank liegen.

Auslösen der Lieferung

Die blockierten Artikel werden versendet. Dieser Prozess dauert einen längeren Zeitraum an.

Abschluss der Lieferung

Der Lieferant bestätigt die Übergabe der Waren an den Kunden.

4.4.2 Implementierung des Geschäftsprozesses

Nachdem der Geschäftsprozess definiert wurde, soll das Saga-System implementiert werden. In 3.4 wurde erläutert, wie die im Koordinator laufende SEC beschrieben werden kann. Die Implementierung soll durch diese Darstellungsform beschrieben werden können.
Für die Transaktionsteilnehmer gilt zu Beginn der Implementierung lediglich die Anforderung, dass die Schnittstellen per Request-Response-Muster aufgerufen werden.

4.4.3 Messung der verschiedenen Implementierungen

Die Bewertung der Implementierungen soll auf Grundlage von Messdaten erfolgen. Es soll nun beschrieben werden, wie die Erfassung dieser Daten erfolgen soll.

Systemtest

Die Messdaten erfolgen in einer produktionsähnlichen Umgebung im Rahmen von Systemtests. Das erwartete Ergebnis ist ein konkreter Endzustand, der erreicht werden soll. Dieser Endzustand kann mit dem erreichten Endzustand verglichen werden.

Testkonfiguration

Ein solcher Systemtest wird unter einer bestimmten Konfiguration durchgeführt. Die Konfiguration setzt sich zusammen aus einem Testcase und einem Netzwerkszenario.

Testcase

Ein Testcase stellt eine konkrete Interaktion mit dem System dar. Die Testcases übernehmen die Aufgabe, die verschiedenen Fälle der Geschäftslogik auf Korrektheit zu überprüfen.

Netzwerkszenarien

Ein Netzwerkszenario ist ebenfalls Teil der Testkonfiguration. Wird ein Systemtest durchgeführt, so beschreibt das Netzwerkszenario das konkrete Netzwerkverhalten.

Messgegenstand

Das Ziel der Messung ist, Aussagen über die Konsistenz des Systems zu treffen.
Die verwendete Implementierung per Orchestrierung setzt das Vorhandensein eines Koordinators voraus. Die darin befindliche Steuerung der LLT durch die SEC gibt Auskunft über die ausgeführten lokalen Transaktionen einer Saga. Es kann für jede lokale Transaktion gemessen werden, wie oft die SEC davon ausgeht, dass eine Transaktion ausgeführt wurde. Analog dazu kann die tatsächlich ausgeführte Anzahl an Transaktionen bestimmt werden, indem die Sicht der Transaktionsteilnehmer verwendet wird. Stimmen die korrespondierenden Werte aller lokalen Transaktionen in beiden Sichten überein, kann davon ausgegangen werden, dass die LLT keine Inkonsistenzen in das System eingeführt hat.
Ein weiterer Anhaltspunkt für Konsistenzanomalien ist der erreichte Endzustand einer Transaktion. In 3.4 wurde ein Endzustand für einen DEA definiert, der erreicht wird, nachdem eine Saga in einer kompensierenden Aktion eine Fehlerantwort erhält. In solchen Fällen ist der Koordinator nicht in der Lage die LLT abzuschließen und endet in einem weder erfolgreichen noch kompensierten Zustand.
Da dieser Zustand einen Fall aufzeigt, in dem die Atomarität der LLT verletzt wird, ist das Auftreten solcher Endzustände ein unmittelbares Zeichen für Inkonsistenz.

4.4.4 Analyse der Messdaten

Die Ergebnisse aus der Messung stellen die Grundlage für eine Analyse des implementierten Systems dar. Die Messwerte geben Auskunft über Konsistenz, Laufzeit und Erfolgsrate. Die Werte sollen interpretiert werden. Es sind Ursachen für Konsistenzanomalien, übermäßig lange oder unbeendete Ausführungen sowie erfolglose LLTs zu identifizieren.

4.4.5 Iteratives Vorgehen

Auf Basis der Messwerte und deren Analyse sind Verbesserungsvorschläge zu formulieren. Im Rahmen einer iterativen Vorgehens fließen diese Ergebnisse in die Entwicklung einer neuen Implementierung ein.

5 Versuchsdurchführung

In diesem Kapitel soll die der in Kapitel 4 geforderte Versuch durchgeführt werden. Zunächst wird der initiale Systementwurf beschrieben. Dazu gehört die Modellierung der lokalen Transaktionen, die Implementierung der dafür vorgesehenen Services und die Implementierung des Koordinators.
Es wird im Anschluss ein initialer DEA verwendet und iterativ angepasst. Jede Anpassung resultiert aus Messergebnissen und daraus ableitbaren Schlüssen.

5.1 Services

Es sollen zuerst die Teilnehmerservices identifiziert werden. Die Schnittstelle zwischen dem Backend-System und dem Benutzer stellt ein Frontend dar. in diesem Frontend werdem dem Nutzer die Produkte dargestellt. Außerdem übernimmt das Frontend die Aufgabe der Verwaltung eines Warenkorbsystems. Der Nutzer kann Produkte zu seinem Warenkorb hinzufügen oder entfernen. Die Bestellung kann nun ausgelöst werden und an das Backend übermittelt werden.
Der Einstiegspunkt für die Abwicklung einer Bestellung soll der OrderService sein. Neben der Initialisierung dieser Bestellung und der Aktualisierung des Bestellungsstatus fallen aus Sicht der Prozessdefinition keine weiteren Aufgaben in den Bereich dieses Services. Die Aktualisierung des Bestellungsstatus ist sehr eng mit der Verwaltung des Prozesses aus Sicht des Koordinators verbunden. Deshalb kann der OrderService die Rolle des Koordinators übernehmen.
Die Artikeldaten sollen dem Nutzer und dem Geschäftsprozess in einer gemeinsamen Schnittstelle bekannt gemacht werden. Diese Aufgabe übernimmt ein eigener ArticleService.
Eine Aufgabengebiet ist die Verwaltung der Lagerbestände. Die Verwaltung des Lagers und den vorrätigen und reservierten Artikeln sowie die Verwaltung von Lieferungen übernimmt der StockService.
Zuletzt muss ein Service die Kontostände der Nutzerkonten und des Händlerkontos verwalten. Dies wird von einem BankService übernommen. Im Versuch sollen zwei Instanzen des BankServices verwendet werden, die zwei verschiedene BankServiceProvider darstellen sollen.

Name des Services	Aufgabe
Frontend	Nutzerschnittstelle (GUI), Anzeige der Produkte, Platzieren der Bestellung
OrderService	Entgegennehmen der Bestellung, Koordinierung des Bestellprozesses
ArticleService	API für die angebotenen Produkte und Preise
StockService	Verwaltung des Lagerbestands, Verwaltung des Lieferprozesses
BankingServices	Verwaltung von Kontonutzern und Kontoständen

Entwicklungsscope

Für die Betrachtung des Bestell- und Lieferprozesses als LLT wird das Frontend außen vorgelassen. Die LLT beginnt mit der Entgegennahme eines Order-Requests im OrderService. Es werden lediglich die Backend-Services entwickelt, die Teil der LLT sind.

5.2 Lokale Transaktionen

Die einzelnen lokalen Transaktionen sollen nun identifiziert werden. Im Folgenden wird beschrieben, durch welchen Teilnehmer die lokale Transaktion jeweils ausgeführt wird.

Initialisierung

Die Bestellung wird im OrderService entgegengenommen. Die Bestellung wird initialisiert. Der Saga-Prozess wird erstellt.

Validierung der Produktdaten

Die in der Bestellung enthaltenen Produktdaten müssen validiert werden. Dies geschieht in zwei Schritten:

1. Abfrage aller in der Bestellung enthaltenen Produktdaten am ArticleService
2. Validierung

Blockierug der Artikel

Die Ware wird im StockService für andere Bestellungen blockiert.

Geldabbuchung

Das Käuferkonto muss belastet werden. Dabei wird der Geldbetrag des Käuferkontos in der entsprechenden Instanz des BankService verringert.

Geldzubuchung

Das Händlerkonto bekommt den selben Geldbetrag gutgeschrieben. Dies geschieht ebenfalls in der entsprechenden Instanz des BankService.

Auslösung der Lieferung

Die blockierten Artikel der Lieferung werden aus dem Lager entfernt und die Lieferung wird ausgelöst. Dies geschieht im StockService.

Lieferabschluss

Die Ware trifft beim Kunden ein. Die Lieferung wird durch den Lieferanten bestätigt. Diese Information erreicht den StockService.

5.3 Kompensierungen

Jede an der LLT teilhabende lokale Transaktion muss kompensierbar sein, damit das Saga-Pattern funktionieren kann. Führt eine lokale Transaktion zu keinem Wechsel des Systemzustands, muss keine Kompensierung stattfinden.

Transaktion	Kompensierung
Initialisierung	
Validierung der Produktdaten	
Blockierug der Artikel	StockService - Freigeben der blockierten Artikel
Geldabbuchung	Erhöhen des Geldbetrages des Kundenkontos
Geldzubuchung	Verringern des Geldbetrages des Händlerkontos
Auslösung der Lieferung	Zurückrufen der Lieferung
Lieferabschluss	

5.4 Fachliche Kontextabgrenzung

Für die Realisierung des Microservicesystems im Rahmen dieser Arbeit wurde die Orchestrierung gewählt. Die Rolle des Koordinators übernimmt der OrderService. Der OrderService übernimmt die Annahme des Bestellprozesses und löst somit die Saga aus.

5.4.1 Frontend

In einem Online-Shop interagiert der Kunde per Frontend mit der Anwendung. Das Frontend soll übernimmt die grafische Schnittstelle zwischen Backend und dem Nutzer. Dazu gehört vor Allem die Darstellung der Artikel in einer Katalogansicht. Die darzustellenden Daten für eine solche Liste müssen zumindest Artikelbezeichnung und Artikelpreis enthalten. Diese Daten sollten aus einer API für Artikeldaten stammen. Darüber hinaus muss das Frontend einen Prozess unterstützen, in dem der Kunde ein Formular ausfüllt, welches die erforderlichen Daten für das Platzieren einer Bestellung enthält. Dazu gehört ein Warenkorbsystem sowie eine Authorisierung und Authentifizierung der Zahlungsidentität des Kunden. Die Bestellung kann also als Objekt mit folgenden Feldern zusammengefasst werden:

- Zahlungsinformationen des Kunden: BankId, UserId

- Liste der zu bestellenden Artikel, mit Artikel: ArticleId, ArticlePrice, Amount

Dieses Objekt kann an das Backend gesendet werden.

5.4.2 ArticleService

Dieser Service ist ein Service zum reinen Lesen der Produktdaten. Er soll eine Schnittstelle zur Verfügung stellen, die dem Frontend ermöglicht, den Produktkatalog abzufragen und darzustellen. Das Backend muss außerdem die Möglichkeit haben, die im Request enthaltenen Artikelpreise zu validieren. Dazu benötigt der ArticleService eine Produktdatenbank. Da dieser Service ausschließlich die Produktdaten als Ressource behandelt, kann er RESTful implementiert werden.

5.4.3 StockService

Der Service soll den aktuellen Bestand an vorrätigen Artikeln abbilden. Es soll möglich sein, eine Menge an Artikeln für eine konkrete Bestellung zu reservieren. Eine Reservierung von einer Menge von Artikeln wartet auf die Auslösen der Lieferung.
Um dies zu erlauben, muss der aktuelle Lagerstand in einer Tabelle hinterlegt sein. Die Tabelle muss den aktuell verfügbaren Bestand pro Artikel ausdrücken.
Um eine Reservierung zu ermöglichen, muss es eine weitere Tabelle geben, die eine Menge von blockierten Artikeln für einen bestimmten Bestellprozess enthält. Beim Reservieren verringert sich der Bestand in der Bestandstabelle und erhöht sich in der Reservierungstabelle. Um die Konsistenz zu gewährleisten, müssen beide Operationen in einer lokalen Transaktion ausgeführt werden.
Um das Auslösen und Abschließen einer Lieferung zu ermöglichen, muss es eine Tabelle geben, die den Inhalt einer Lieferung und einen Status enthält. Wenn eine Lieferung ausgelöst wird, werden die für diesen Vorgang reservierten Artikel aus der Reservierungstabelle entfernt und in der Lieferungstabelle eingefügt. Diese Transaktion soll den physischen Prozess abbilden, die bestellten und für diese Bestellung blockierten Artikel aus dem Lager in das Transportfahrzeug und schließlich zum Kunden zu transferieren.
Die Übergabe der Ware an den Kunden stellt den finalen Schritt des Prozesses dar. Ist dies geschehen, gibt der Lieferant dem StockService die Bestätigung für die gelieferte Bestellung.
Die Blockierung eines Artikels muss kompensiert werden können, da sonst der reservierte Artikel nach Abbruch einer Bestellung nicht wieder freigegeben würde. Deshalt muss diese Kompensierung die Einträge aus der Blockierungstabelle entfernen und die Anzahl auf den Lagerbestand addiert werden. Dies soll ebenfalls in einer lokalen Transaktion ablaufen, um Konsistenz zu wahren.
Die Auslösung der Lieferung ist nur bedingt kompensierbar. Nachdem das Transportfahrzeug mit der Ware losgefahren ist und die Ware noch nicht übergeben hat, kann die Lieferung noch abgebrochen und somit kompensiert werden. Die Kompensierung muss also den Lieferant benachrichtigen und die Ankunft der Waren bestätigen. Nach der Saga-Definition soll die zu kompensierende Transaktion zurückgerollt werden. Deshalb werden die Waren aus der Lieferungstabelle zurück in die Reservierungstabelle geschrieben.

Der Abschluss einer Lieferung bildet die physische Warenübergabe an den Kunden ab. Eine Kompensierung ist hier nicht möglich. Da diese lokale Transaktion die letzte Transaktion ist und einen erfolgreichen Abschluss der Saga zur Folge hat, muss hier keine Kompensierung angeboten werden.

5.4.4 BankingServices

Im Geschäftsprozess wurde definiert, dass die Transaktion den Geldbetrag des Kundenkontos und des Händlerkontos in zwei separaten lokalen Transaktionen abwickeln können soll. Somit muss der BankingService jeweils eine Transaktion zum Erhöhen und zum Verringern des Geldbetrages anbieten. Der BankingService soll am Ende in zwei Instanzen laufen, die zwei verschiedene Banken darstellen sollen. Kunden- und Käuferkonto können, müssen aber nicht bei derselben Bank liegen.
Um dies zu ermöglichen benötigt der BankingService eine Tabelle, die seine Nutzer enthält. Zusätzlich benötigt der Service eine Tabelle, die den aktuellen Geldbetrag jedes Nutzers enthält. Außerdem sollten die einzelnen Transaktionen jedes Nutzers in einer separaten Tabelle gesichert werden. Für die reine Implementierung dieser Anwendung wäre dies nicht notwendig. Für den Nutzer eines BankingServices ist neben dem Kontostand auch die Liste an getätigten Transaktionen interessant, um die Ausgaben und Einnahmen zuordnen zu können. Im Rahmen dieser Implementierung wird die Tabelle zusätzlich für Analysezwecke verwentet werden.
Bei einer Anfrage, den Geldbetrag eines konkreten Nutzers zu erhöhen, wird in einer lokalen Transaktion der Betrag des Kontos in der UserCredit-Tabelle erhöht und die Differenz in der Transaktion-Tabelle eingetragen.
Der Service muss Anfragen zum Geldabbuchung ablehnen, wenn die Verringerung den Kontostand in den negativen Bereich fallen lassen würde. In diesem Fall wird die Transaktion abgebrochen.
Beide angebotenen Operationen benötigen eine zugehörige Kompensation, da sie den Datenbestand verändern. Die Verwendung des jeweils anderen Endpunktes ist semantisch bereits korrekt. Der Klarheit halber sollen zwei weitere Entpunkte eingeführt werden, die nur für die Kompensation verwendet werden sollen.

5.4.5 OrderService

Der OrderService übernimmt die Rolle des Koordinators im Orchestrator-Saga-Patterns. Die Bestellung wird entgegengenommen und vom OrderService initialisiert. Zur Initialisierung gehört die Generierung einer Vorgangsnummer sowie das Abspeichern der Bestellung in einer separaten Tabelle. Anhand dieser Tabelle wird persistiert, an welcher Stelle der Ausführung die Saga sich befindet, und in welchem Status die Bestellung ist. Die etwaigen ausgeführten Kompensationsschritte sind in ihrer eigenen Tabelle und werden der Vorgansnummer zugeordnet. Die gewünschten Artikel einer Bestellung sind in eine separate Tabelle ausgelagert und verweisen auf die Saga-Tabelle.
Als Koordinator hat dieser Service die Verantwortung, die an der Saga beteiligten Services korrekt aufzurufen. Die Reihenfolge und die getroffenen Entscheidungen repräsentieren die Geschäftslogik.

Nach jedem Schritt persistiert der OrderService den Erfolg oder Misserfolg. Die Tabelle, in der die Schritte gespeichert werden, stellt das Saga Execution Log dar. Außerdem ruft der Service nach Feststellung eines Misserfolgs die Backward-Recovery auf.
Neben der Schnittstelle zum Platzieren der Bestellung soll dem Nutzer ermöglicht werden, die Bestellung zu stornieren. Dieser zusätzliche Endpunkt nimmt die Vorgangsnummer der zuvor ausgelösten Bestellung entgegen. Falls die Ware noch nicht beim Kunden eingetroffen ist, kann hier die Bestellung abgebrochen werden. Eine solche Stornierung löst ebenfalls Backward-Recovery aus und soll den Initialzustand wiederherstellen. Diese Aktion stellt keine lokale Transaktion der globalen Transaktion dar; es ist eine Aktion, die von außen in den Bestellprozess eingreift. Damit ist die Stornierung kein T und hat somit kein zugehöriges C.

5.5 Technische Kontextabgrenzung

Bei der Modellierung der Datenbankschicht wurde das Database-per-Service Muster verwendet. Dieses Muster erlaubt eine sehr lose Bindung der Services und lässt die Datenbasis in der Verantwortung eines Services. Es ist möglich, ein Microservice-System zu implementieren, welches das Muster einer gemeinsamen Datenbank verwendet. Dies würde die Implementierung atomaren und konsistenten Transaktionen vereinfachen. Die Voraussetzungen für eine solche Implementierungen sind jedoch nicht immer gegeben, da an der LLT teilhabende Services unter Umständen nicht in der Verantwortung des Entwicklerteams liegen könnten, welches die LLT zu implementieren hat. Aus diesem Grund wird im Rahmen der für diese Implementierung vorzunehmende Implementierung jeder Service als getrennte Komponente aufgefasst, die als alleinige Instanz auf ihre eigenen Daten zugreift und zugreifen kann.

5.5.1 ArticleService

Datenbankschicht Der ArticleService soll lediglich Daten aus der Datenbanksicht zurückliefern. Deshalb wird für diesen Service lediglich eine Tabelle benötigt. Die Tabelle *article* enthält für jeden anzubietenden Artikel den zugehörigen Namen und Preis.

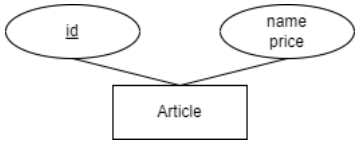

Abbildung 5.1: ER-Diagramm ArticleService

Schnittstellen

Die Produktdaten sollen von dem Frontend abgefragt und dargestellt werden. Dabei werden alle vorhandenen Produktdaten selektiert und zurückgegeben.
Zusätzlich soll der Koordinator eine gezielte Abfrage durchführen können, welche den Preis eines Artikels für eine gegebene ProduktId liefert. Beide Schnittstellen sind RESTful, da jeder Artikel als Ressource angesehen wird.

Da eine Bestellung mehrere unterschiedliche Artikel enthalten darf, wird zusätzlich eine Schnittstelle angeboten, die dem Aufrufer ermöglicht, mehrere Artikel per Id aufzulösen. Die angefragten ArtikelIds werden als Queryparameter übergeben.

Endpunkt	Http-Methode	Argumente
/api/articles	GET	optionale Einschränkung der Ids per Queryparameter
/api/articles/{id}	GET	ArticleId

5.5.2 StockService

Datenbankschicht

Der StockService verwaltet vier Tabellen. Die zentrale Tabelle ist *articlestock* und stellt für jeden bekannten Artikel den auf Lager befindlichen Vorrat dar. Eine Reservierung für eine gegebene Vorgangsnummer wird in der Tabelle *blockedarticles* durch alle Tupel, bei denen diese Nummer auftritt. Die Lieferungen werden in der Tabelle *shipments* dargestellt. Auch hier gehört eine Vorgangsnummer zu den Spalten. Zusätzlich gibt es das Attribut *hasarrived*, was Auskunft über den Status der Lieferung gibt. Die zu einer Lieferung gehörenden Artikel und deren Menge sind in der Tabelle *shippedarticles* enthalten.

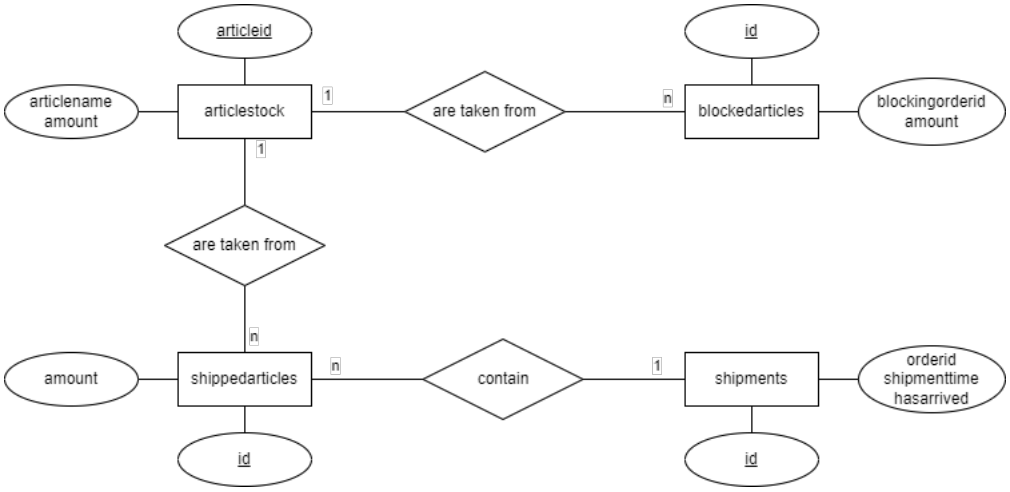

Abbildung 5.2: ER-Diagramm StockService

Schnittstellen

Der Endpunkt */api/blocked-articles* überträgt frei verfügbare Artikel aus der Tabelle *articlestock* in die Tabelle *blockedarticles*. Als Argumente werden die zu blockierenden Artikel-Ids und Artikel-Mengen sowie die Vorgangsnummer benötigt. Teil der lokalen Transaktion sind folgende Schritte:

1. Reduzieren jedes Artikelvorrats in der Tabelle *articlestock*

2. Einfügen eines Elements in der Tabelle *blockedarticles* für jede zu blockierende Artikel-Id

Der Endpunkt */api/blocked-articles-compensation* stellt die Kompensierung für die Schnittstelle */api/blocked-articles* dar. Als Argument wird hier lediglich die Vorgangsnummer benötigt. Notwendige Schritte der auszuführenden lokalen Transaktion sind:

1. Selektieren aller Elemente aus *blocked-articles* mit der angefragten Vorgangsnummer
2. Löschen dieser Elemente
3. Erhöhen der Vorratsmengen in *articlestock* für jede Artikelblockierunge

Der Endpunkt */api/start-shipment* erwartet die Vorgangsnummer als Argument. Die reservierten Artikel werden in eine Lieferung umgewandelt und versendet. Die Schritte der ablaufenden lokalen Transaktion sind:

1. Initialisieren eines Elements in *shipments* mit dem Status *hasarrived=0*
2. Selektieren und Löschen aller Elemente in *blockedarticles*, die die OrderId enthalten
3. Einfügen der Selektierungen in der Tabelle *shippedarticles*

Der Endpunkt */api/finish-shipment* wird ausschließlich vom Lieferanten verwendet und dient zur Bestätigung der Lieferung. Es wird lediglich der Status der entsprechenden ShipmentId in der Tabelle *shipments* auf 1 gesetzt.
Der Endpunkt */api/shipments* liefert lediglich den Status der angeforderten ShipmentId. Der Koordinator hat mit dem Aufruf von */api/start-shipments* die Lieferung ausgelöst. Der Koordinator hat mit der Schnittstelle */api/shipments* die Möglichkeit, den Status der Lieferung solange abzufragen, bis der Lieferant per Aufruf von */api/finish-shipment* den Lieferabschluss bestätigt. Die Kombination von */api/start-shipment*, */api/finish-shipment* und */api/shipments* können als Polling-Implementierung eines asynchronen Request-Response Musters aufgefasst werden.
Der letzte Endpunkt des StockServices ist */api/cancel-shipment* und bietet dem Koordinator die Möglichkeit, auf eine Stornierung zu reagieren. Bis die Lieferung abgegeben und bestätigt wurde, kann diese Schnittstelle verwendet werden, um die Lieferung abzubrechen. Als Argument wird die ShipmentId benötigt.

Endpunkt	Http-Methode	Argumente
/api/blocked-articles	POST	ArticleId, Amount, OrderId
/api/blocked-articles-compensation	POST	OrderId
/api/start-shipment	POST	OrderId
/api/finish-shipment	POST	ShipmentId
/api/shipments	GET	ShipmentId
/api/cancel-shipment	POST	ShipmentId

5.5.3 BankingService

Datenbankschicht

Zur Verwaltung des BankingServices gehören drei Tabellen. Die Tabelle *bankuser* enthält alle Nutzer, die Tabelle *bankusercredit* ordnet jedem Nutzer einen Kontostand zu und die Tabelle *bankusertransaction* stellt Veränderung des Kontostands der Tabelle *bankcredit* in der Tabelle *bankusertransaction* als Historie dar.

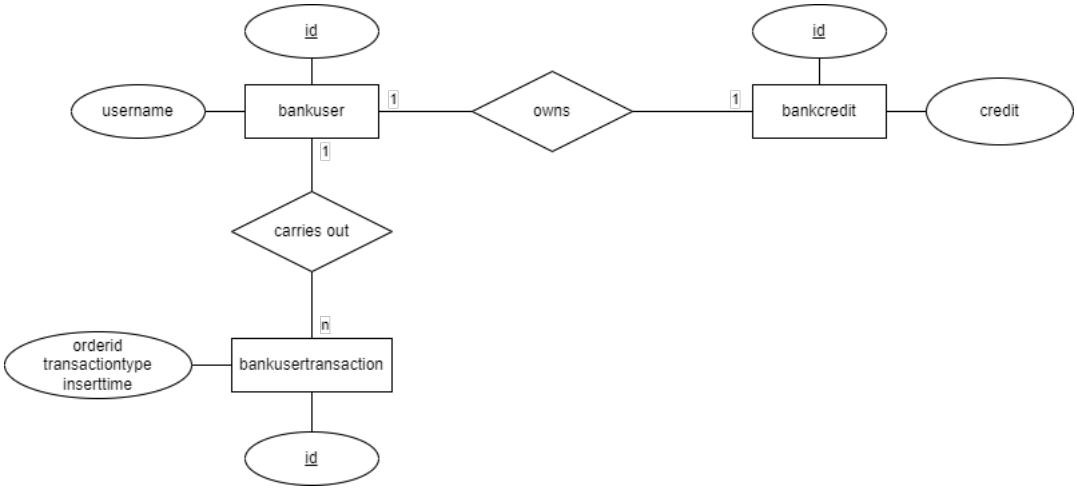

Abbildung 5.3: ER-Diagramm BankingService

Schnittstellen

Der BankingService bietet jeweils eine Schnittstelle zum Erhöhen und zum Verringern des Kontostandes an. Dabei wird lediglich in der Tabelle *bankcredit* das Credit-Attribut erhöht oder verringert. Die Spalte ist im Datenbankmanagementsystem mit einer Einschränkung versehen, die verhindert, dass der Wert der Spalte unter 0 fällt.
Die Veränderung des Kontostandes ist der erste Schritt der lokalen Transaktion. Der zweite Schritt ist das Einfügen eines neuen Elementes in der Tabelle *bankusertransaction*. In dieser Tabelle wird der Transaktionstyp festgehalten.

Endpunkt	Http-Methode	Argumente
/api/add-money	POST	UserId, Amount, OrderId
/api/add-money-compensation	POST	UserId, Amount, OrderId
/api/remove-money	POST	UserId, Amount, OrderId
/api/remove-money-compensation	POST	UserId, Amount, OrderId

5.6 Ergebnisse

Der entworfene Prozess soll nun als DEA dargestellt werden. Dafür sind alle möglichen Ergebnisse zu erfassen, die jede lokale Transaktion aus Sicht des Koordinators liefern kann.

5.6.1 Ergebnisse aller Transaktionen

Transaktion	Ergebnis	Bedeutung
Initialize Saga	Success	Bestellung ist initialisiert
Get Article Data	200	Artikeldaten wurden vom ArticleService empfangen
	404	Artikel wurde nicht gefunden
Validate Price	Success	Preis aus Bestellung und aus dem System stimmen überein
	Failure	Preise aus Bestellung und aus dem System stimmen nicht überein
Block Articles	200	Produkte wurden reserviert
	409	Conflict (Lagervorrat ist erschöpft)
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Remove Money	200	Geldbetrag auf dem Konto wurde verringert
	409	Lokale Transaktion ist fehlgeschlagen (Konto ist nicht gedeckt)
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Add Money	200	Geldbetrag auf dem Konto wurde erhöht
	409	Lokale Transaktion ist fehlgeschlagen
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Start Shipment	200	Lieferung wurde ausgelöst
	409	Lokale Transaktion ist fehlgeschlagen
	429	Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Check No Cancellation	Success	Es wurde keine Stornierung festgestellt
	Failure	Es wurde eine Stornierung festgestellt
Get Shipment Status	200	Status wurde vom StockService empfangen
	404	Lieferung existiert nicht
Check Shipment Status	Success	Lieferstatus signalisiert abgeschlossene Lieferung
	Failure	Lieferstatus signalisiert noch nicht abgeschlossene Lieferung

5.6.2 Ergebnisse aller Kompensierungen

Transaktion	Ergebnis	Bedeutung
Initialize Saga Compensation	Success	

Get Article Data Compensation	Success	
Validate Price Compensation	Success	
Block Articles Compensation	200 404 409 429	Reservierung wurde aufgehoben Reservierung wurde nicht gefunden Lokale Transaktion ist fehlgeschlagen Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Remove Money Compensation	200 409 429	Geldbetrag auf dem Konto wurde erhöht Lokale Transaktion ist fehlgeschlagen Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Add Money Compensation	200 208 404 409 429	Geldbetrag auf dem Konto wurde erhöht Transaktion wurde auf vorherige Anfrage bereits kompensiert keine zugehörige kompensierbare Transaktion gefunden Lokale Transaktion ist fehlgeschlagen Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl
Start Shipment Compensation	200 404 409 410 429	Lieferung wurde abgebrochen Lieferung existiert nicht Lokale Transaktion ist fehlgeschlagen Lieferung wurde nicht abgebrochen, da sie bereits abgeschlossen ist Mehrere Transaktionen behindern sich und lokale Transaktion schlägt fehl

5.7 Saga Execution Component

Es wurde festgelegt, dass der OrderService die Rolle des Koordinators übernimmt. Das bedeutet, dass der OrderService die Saga Execution Component enthält. Als solche hat der OrderService die Verantwortung, die an der LLT teilhabenden Services aufzurufen und auf die Ergebnisse zu reagieren. In diesem Abschnitt werden die dafür notwendigen Implementierungsdetails dargestellt.

5.7.1 Rahmenbedingung für die Versuchsdurchführung

Die SEC übernimmt die Aufgabe, einen DEA auszuführen. Im Rahmen des Versuchs sollen mehrere DEAs konstruiert und durch die selbe SEC ausgeführt werden. Die SEC muss also flexibel genug implementiert werden, damit eine Parametrisierung der Ausführung mit einem DEA als Argument möglich ist.

5.7.2 Ausführung eines DEAs

Zunächste soll der Ablauf eines Durchlaufs eines DEAs innerhalb der SEC dargestellt werden. Es soll nun davon ausgegangen werden, dass ein DEA definiert wurde.

1. Nach Eingang einer Bestellung wird der Initialzustand gewählt, um die Ausführung des DEAs zu starten.
2. Jeder Zustand korrespondiert mit einer auszuführenden Aktion. Diese Aktion wird ausgeführt.
3. Eine Aktion resultiert in einem Ergebnis. Dieses Ergebnis stellt ein Element aus der Menge des Eingabealphabets dar. Dieses Element wird zusammen mit dem Zustand in der Datenbank gespeichert.
4. Die Ausführung des Zustands wird beendet.
5. Die SEC bestimmt den nächsten Zustand. Und führt die korrespondierende Aktion aus.
6. Die Aktion resultiert in einem Ergebnis. Das Ergebnis wird in der Datenbank gespeichert.
7. Die Ausführung des Zustands wird beendet.
8. Die SEC wiederholt diesen Prozess solange bis ein Endzustand erreicht wird.

5.7.3 Modellierung eines DEAs

Die Modellierung eines DEAs ist in Codebeispiel 5.1 abgebildet.

```
1  public class SimpleStateMachine
2  {
3    public List<string> States;
4    public string InitialState;
5    public List<string> EndStates;
6    public List<string> Sigma;
7    public List<Tuple<Tuple<string, string>, string>> Relations;
8
9
10   public SimpleStateMachine(List<string> states,
11     string initialState,
12     List<string> endStates,
13     List<string> sigma,
```

```
14      List<Tuple<Tuple<string, string>, string>> relations)
15    {
16      States = states;
17      InitialState = initialState;
18      EndStates = endStates;
19      Sigma = sigma;
20      Relations = relations;
21    }
22  }
```

Codebeispiel 5.1: Modellierung eines DEA in C#

Relations

Das Feld *Relations* stellt Liste von möglichen Zustandsübergängen dar. Die SEC kann aus einem Zustand und einem Element aus Sigma den entsprechenden Zustandsübergang berechnen. Damit diese Berechnung deterministisch ist, darf für einen gegebenen Zustand und einen gegebenes Element aus Sigma nur ein Element existieren.

5.7.4 Konstruktion eines DEAs

Nun soll ein solcher DEA initialisiert werden. Dazu wird der Konstruktor dieser Klasse aufgerufen. Es soll folgender Automat konstruiert werden:

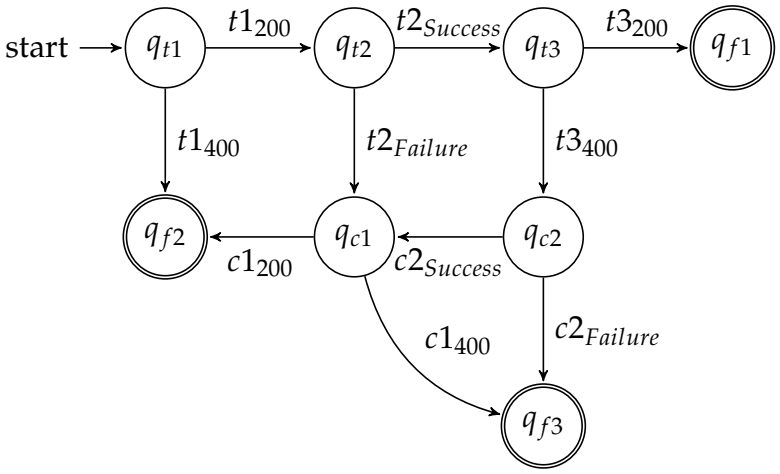

Abbildung 5.4: Zu konstruierender DEA

Die Initialisierung ist in Codebeispiel 5.2 abgebildet:

```
1  public class ExampleDea
2  {
3    // representation of states
4    public static string Q_T1 = "qt1";
5    public static string Q_T2 = "qt2";
6    public static string Q_T3 = "qt3";
7    public static string Q_C1 = "qc1";
8    public static string Q_C2 = "qc2";
9    public static string Q_F1 = "qf1";
```

```
10    public static string Q_F2 = "qf2";
11    public static string Q_F3 = "qf3";
12
13    // representation of sigma
14    public static string T1_200 = "t1_200";
15    public static string T1_400 = "t1_400";
16    public static string T2_Success = "t2_Success";
17    public static string T2_Failure = "t2_Failure";
18    public static string T3_200 = "t3_200";
19    public static string T3_400 = "t3_400";
20    public static string C1_200 = "c1_200";
21    public static string C1_400 = "c1_400";
22    public static string C2_200 = "c2_200";
23    public static string C2_400 = "c2_400";
24
25    public SimpleStateMachine ConstructExampleDea()
26    {
27      var states = new List<string>
28      {
29        Q_T1, Q_T2, Q_T3, Q_C1, Q_C2, Q_F1, Q_F2, Q_F3,
30      };
31
32      var initialState = qt1;
33
34      var sigmaElements = new List<string>
35      {
36        T1_200, T1_400,
37        T2_Success, T2_Failure,
38        T3_200, T3_400,
39        C1_200, C1_400,
40        C2_200, C2_400
41      };
42
43      var endStates = new List<string>
44      {
45        Q_F1, Q_F2, Q_F3
46      };
47
48      var relations = new List<
49        Tuple<Tuple<string, string>, string>>
50      {
51        BuildRelation(Q_T1, T1_200, Q_T2),
52        BuildRelation(Q_T1, T1_400, Q_F2),
53        BuildRelation(Q_T2, T2_Success, Q_T3),
54        BuildRelation(Q_T2, T2_Failure, Q_C1),
55        BuildRelation(Q_T3, T3_200, Q_F1),
56        BuildRelation(Q_T3, T3_400, Q_C2),
57        BuildRelation(Q_C1, C1_200, Q_F2),
58        BuildRelation(Q_C1, C1_400, Q_F3),
```

```
59          BuildRelation(Q_C2, C2_200, Q_C1),
60          BuildRelation(Q_C2, C2_400, Q_F3)
61      };
62
63      return new SimpleStateMachine(states, initialState, sigmaElements
            , endStates, relations);
64    }
65
66    private Tuple<Tuple<string, string>, string> BuildRelation(
67      string inputState,
68      string inputSigmaElement,
69      string newState)
70    {
71      return new Tuple<Tuple<string, string>, string>(
72        new Tuple<string, string>(inputState, inputSigmaElement),
73      newState);
74    }
75  }
```

Codebeispiel 5.2: Modellierung eines Beispiel DEA in C#

Verwaltung der DEAs

Nach Vorbild des Beispiels in Codebeispiel 5.2 werden alle ausführbaren DEAs definiert. Diese Initialisierung geschieht innerhalb der in Codebeispiel 5.3 definierten Klasse.

```
1   public interface IStateMachineMapper
2   {
3     StateMachine MapToStateMachine(int smValue);
4   }
5
6   public class StateMachineMapper : IStateMachineMapper
7   {
8     private readonly StateMachine Sm1;
9     private readonly StateMachine Sm2;
10    // more StateMachineDefinitions
11
12    public StateMachine MapToStateMachine(int smValue) {
13      switch (smValue)
14      {
15        case 1:
16          return Sm1;
17        case 2:
18          return Sm2;
19        // ...
20      }
21
22          throw new UnknownStateMachineException($"Unknown StateMachine
                for value [{smValue}].");
23    }
```

```
24
25 }
```

Codebeispiel 5.3: Modellierung eines StateMachineMappers in C#

Per Dependency Injection kann die SEC diese Klasse verwenden. Per Strategie-Pattern wird in einer Funktion der gewünschte DEA ermittelt werden.

Kontrollfluss der SEC

Die SEC verwendet den StateMachineMapper, um den gewünschten DEA auszuführen. Die tatsächliche Ausführung findet in einer separaten Komponente, dem StateMachineExecutor statt. Diese Klasse enthält eine Funktion *ExecuteStateMachine*, die für einen DEA und ein Eingabewort den aktuellen Zustand berechnet. Dazu wird jedes Element des Eingabeworts abgearbeitet. Für jedes Element und den aktuellen Zustand wird in der Funktion *FindRelation* die zugeordnete Relation berechnet. Der in der Relation enthaltene Folgezustand wird gesetzt und das abgearbeitete Element des Eingabeworts wird entfernt.
Das Ergebnis der Berechnung ist eine Liste von Konfigurationsübergängen. Eine Konfiguration wird durch die in Codebeispiel 5.4 dargestellten Klasse modelliert.

```
1  public class StateMachineConfiguration
2  {
3    public List<string> Word { get; set; } = new();
4    public string State { get; set; } = string.Empty;
5
6    // Clone Function
7  }
```

Codebeispiel 5.4: Modellierung einer StateMachineConfiguration in C#

Ein Konfigurationsübergang stellt den Wechsel von einer Konfiguration in eine Folgekonfiguration dar. Dabei wird die Länge des Eingabeworts stets verringert.

```
1  public class StateMachineConfigurationTransition
2  {
3    public StateMachineConfiguration C1 { get; set; }
4    public StateMachineConfiguration C2 { get; set; }
5    public Tuple<Tuple<string, Status>, string> Relation { get; set; }
6
7    public StateMachineConfigurationTransition(
8      StateMachineConfiguration c1,
9      StateMachineConfiguration c2,
10     Tuple<Tuple<string, string>, string> relation)
11   {
12     C1 = c1;
13     C2 = c2;
14     Relation = relation;
15   }
16 }
```

Codebeispiel 5.5: Modellierung einer StateMachineConfigurationTransition in C#

Solange das Eingabewort noch nicht vollständig abgearbeitet wurde, werden Konfigurationsübergänge berechnet. Die resultierende Liste von Konfigurationsübergängen beschreibt eindeutig den gewählten Graphen.

```
1  public interface IStateMachineExecutor
2  {
3    List<StateMachineConfigurationTransition> ExecuteStateMachine(
       StateMachine sm, List<Status> word);
4  }
5
6  public class StateMachineExecutor : IStateMachineExecutor
7  {
8    public List<StateMachineConfigurationTransition>
       ExecuteStateMachine(StateMachine sm, List<string> word)
9    {
10     List<StateMachineConfigurationTransition> result = new();
11
12     StateMachineConfiguration current = new StateMachineConfiguration
13     {
14       Word = word,
15       State = sm.InitialState
16     };
17
18     while (current.Word.Count > 0)
19     {
20       StateMachineConfiguration c1 = StateMachineConfiguration.Clone(
           current);
21
22       var relation = FindRelation(sm.Relations, current);
23
24       string nextState = relation.Item2;
25
26       current.State = nextState!;
27       current.Word.RemoveAt(0);
28
29       StateMachineConfigurationTransition transition = new()
30       {
31         C1 = c1,
32         C2 = new StateMachineConfiguration
33         {
34           State = nextState,
35           Word = current.Word
36         },
37         Relation = relation
38       };
39
40       result.Add(transition);
41     }
42
```

```
43      return result;
44    }
45
46    private Tuple<Tuple<string, string>, string> FindRelation(
47      List<Tuple<Tuple<string, string>, string>> relations,
48      StateMachineConfiguration currentConfiguration)
49    {
50      foreach (var relation in relations)
51      {
52        var relStartState = relation.Item1.Item1;
53        var relWord = relation.Item1.Item2;
54
55        if (currentConfiguration.State.Equals(relStartState) &&
56          currentConfiguration.Word.First() == relWord)
57        {
58          return relation;
59        }
60      }
61
62      throw new NoRelationException($"No Relation found for State = [{
          currentConfiguration.State}] and Element = [{
          currentConfiguration.Word.First()}]");
63    }
64  }
```

Codebeispiel 5.6: Modellierung einer StateMachineExecutors in C#

Das Ergebnis der Berechnung ist die Liste von Konfigurationsübergängen. Der letzte Konfigurationsübergang enthält den Zielzustand, der noch auszuführen ist. Über den ActionExecutor wird die mit diesem Zustand korrespondierende Aktion ausgeführt. Die Kommunikation der von der SEC verwendeten Klassen ist in Abbildung 5.5 abgebildet. Die Funktion *GetTransactionsForOrderSaga* greift auf die Datenbanktabelle *ordersagatransactions* zu und liefert das Transaktionslog. Das Transaktionslog wird lediglich von den mit einem Zustand korrespondierenden Aktionen ergänzt. Die SEC selbst greift nur lesend auf dieses Log zu.

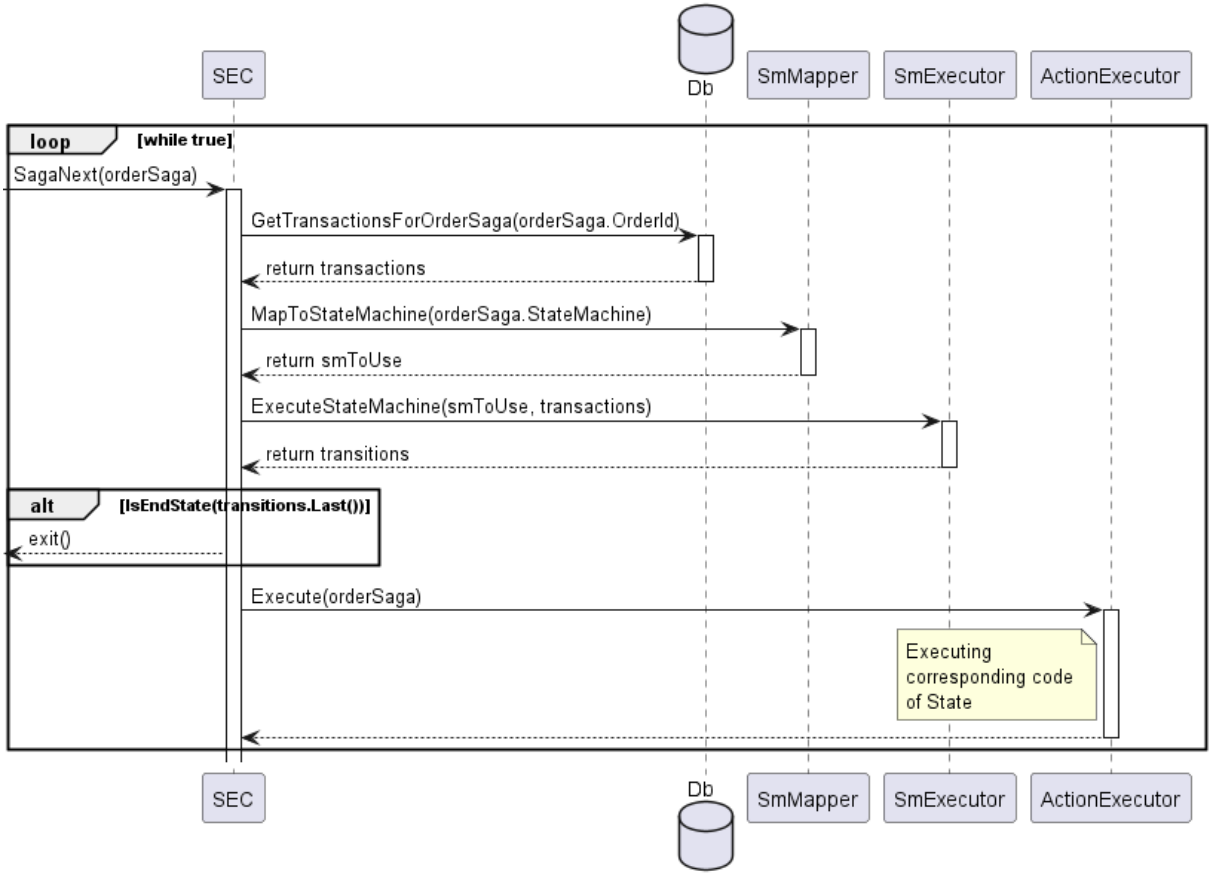

Abbildung 5.5: Kontrollfluss der SEC

5.8 Planung der Datenerfassung

Auf Grundlage des nun bekannten Systems sollen die Testfälle und Testszenarien definiert werden.

5.8.1 Testszenarien

Ein Testszenario soll das Verhalten des Netztwerks beschreiben. Ziel der in diesem Kapitel stattfindenden Untersuchung ist eine robuste Saga-Implementierung, die in allen möglichen Testszenarien ein konsistentes Verhalten des Systems gewährleistet.

5.8.2 Fehlerquellen

Zuerst ist zu identifizieren, an welchen Stellen eine Kommunikation in einem verteilten System mittels Request-Response-Muster fehlschlagen kann. Im folgenden Beispiel wird dies anhand einer simplen Kommunikation illustriert, die aus einem Request und einer Response besteht. Es wird davon ausgegangen, dass die Verarbeitung des Requests zu einer Änderung des Systemzustands des Empfängers führt.

Szenario 1
Im ersten Szenario entstehen keine Netzwerkfehler. Dieses Szenario ist der Ausgangspunkt. Die Korrektheit der Saga in diesem Fall ist eine Voraussetzung für die Formulierung und Analyse in folgenden Szenarien.

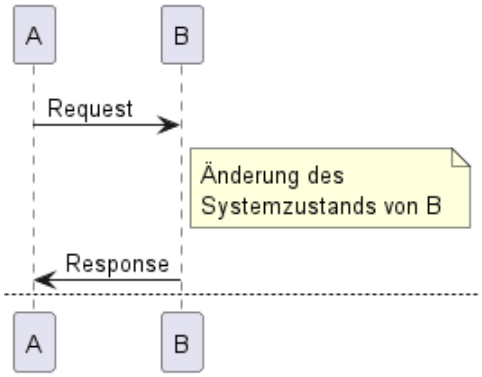

Abbildung 5.6: Sequenzdiagramm für Szenario 1

Szenario 2
Es können Netzwerkfehler auftreten, die verhindern, dass der Request den Empfänger erreicht. Dabei findet keine Verarbeitung der Nachricht im Empfängerservice statt. Es findet weder eine Veränderung des Systemzustands des Empfängers noch eine Versendung einer Response statt.

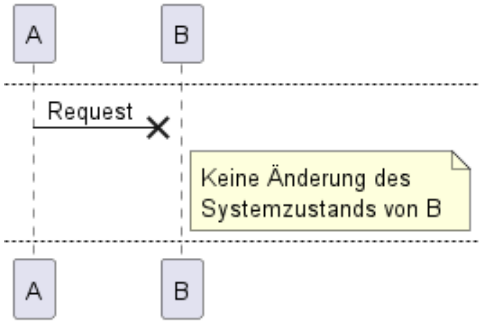

Abbildung 5.7: Sequenzdiagramm für Szenario 2

Im Rahmen dieses Versuchs wird das Auftreten solcher Fehler in Testszenario 2 simuliert.

Szenario 3
Findet ein Netzwerkfehler nach der Verarbeitung des Requests im Empfängerservice statt, erreicht die Response den Sender nicht. Die abgeschlossene Verarbeitung des Requests hat zu einer Veränderung des Systemzustands im Empfängerservice geführt. Dies führt dazu, dass der Sender keine Kenntnis über den Erfolg oder Misserfolg des ursprünglichen Requests hat.

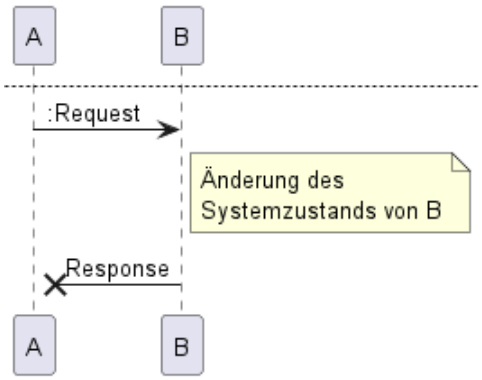

Abbildung 5.8: Sequenzdiagramm für Szenario 3

Im Rahmen dieses Versuchs wird das Auftreten solcher Fehler in Testszenario 3 simuliert. Zusätzlich treten in Testszenario 3 die Netzwerkfehler des Testszenario 2 auf.
Das Testszenario 3 simuliert somit ein Netzwerkverhalten, welches verschiedene Arten von Netzwerkfehlern abdeckt. Wenn eine LLT unter den im Testszenario 3 geschaffenen Bedingungen Konsistenz gewährt, kann die These angenommen werden.

5.8.3 Simulation der Testfälle

Die zu simulierenden Testfälle werden durch einen separaten Service durchgeführt, die TestApi. Die Schnittstellen, die direkt mit der Durchführung der LLT interagieren sind:

Service	Schnittstelle	Beschreibung	Akteur
OrderService	*POST /api/orders*	Platzieren einer Bestellung	Frontend (Kunde)
StockService	*POST /api/finish-shipment*	Lieferbestätigung	Lieferant
OrderService	*POST /api/cancel-order*	Stornierung	Frontend (Kunde)

FinishOrders

Der erste Testfall simuliert eine erfolgreiche Verarbeitung des Bestell- und Lieferprozesses. Die TestApi generiert Bestellungen und ruft den entsprechenden Endpunkt zum Platzieren im OrderService auf. Für jede der Bestellungen wird in der TestApi gewartet, bis die Lieferung ausgelöst wurde. Dann wird die Lieferbestätigung simuliert, indem der entsprechende Endpunkt im StockService aufgerufen wird.
Der Koordinator erhält die Möglichkeit, den erfolgreichen Endzustand zu erreichen. Dies ist der erwartete Endzustand. Dieser Testfall wird als *FinishOrders* bezeichnet.

CancelOrders

Der zweite Testfall simuliert eine Stornierung der Bestellung. Die TestApi generiert Bestellungen und platziert diese. Nachdem die Lieferung ausgelöst wurde, platziert die TestApi die Stornierung.

Der Koordinator erkennt die Stornierung und hat die Verantwortung, die Lieferung zu stoppen und im Anschluss alle lokalen Transaktionen zu kompensieren. In diesem Testfall werden alle Kompensierungen aufgerufen. Dieser Testfall wird mit *CancelOrders* bezeichnet.
Der erwartete Endzustand dieser Bestellung ist *FailedWithCompensation*.

5.8.4 Datengenerierung

Um den Ablauf der Services zu simulieren, müssen Daten in jedem Teilsystem generiert werden. Dabei wird in dieser Versuchsdurchführung zwischen zwei Arten der Datengenerierung unterschieden: statische und dynamische Datengenerierung.

Statisch generierte Daten

Zu den statisch generierten Daten gehören die Daten, die manuell einmalig generiert und anschließend in die Datenbank eingefügt werden.
Im ArticleService werden die Daten der Produkttabelle generiert. Dazu gehören die verschiedenen Produktbezeichnungen und Preise.
In den zwei BankServices werden die Daten der Nutzertabellen generiert. Für jeden BankService werden 100 verschiedene Nutzer erstellt. Initial erhält jeder dieser Nutzer einen ausreichend großen Geldbetrag als Startguthaben.

Dynamisch generierte Daten

Die in der TestApi generierten Bestellungen werden dynamisch erzeugt. Jede Bestellung wählt einen zufälligen BankService-Nutzer und eine Menge von zwischen 1 und 10 verschiedenen Artikeln mit einer zufälligen Menge zwischen 1 und 4.
Eine solche zufällig generierte Bestellung ist in Codebeispiel 5.7 abgebildet.

```
{
  "consument": {
    "userId": "Xena",
    "bankId": "Bank1"
  },
  "requestedArticles": [
  {
    "articleId": 42,
    "articlePrice": 99.0,
    "amount": 2
  },
  {
    "articleId": 16,
    "articlePrice": 49.99,
    "amount": 3
  }
}
```

Codebeispiel 5.7: Json für dynamisch generierte Bestellung

5.8.5 Messwerte

Um die verschiedenen Messungen vergleichen zu können, werden die Messungen eines DEAs in jedem Testfall und in jedem Testszenario durchgeführt. Nach der Messung sollen daraus Aussagen über die Konsistenz und über die Korrektheit der modellierten LLT getroffen werden. Dazu werden folgende Messwerte erhoben:

StateAnalysisResult

Dieser Teil des Messergebnisses verwendet die im Koordinator enthaltenen Zustände. Als Scope wird ein konkreter Testfall, ein konkretes Testszenario und ein DEA festgelegt.

Messwert	Beschreibung
totalCount	Anzahl der Sagas
successfullCount	Anzahl der Sagas mit Endzustand $q_{Success}$
finishedCount	Anzahl der Sagas mit Endzustand $\neq q_{Pending}$
pendingCount	Anzahl der Sagas mit Endzustand $q_{Pending}$
failedWithCompensation-Count	Anzahl der Sagas mit Endzustand $q_{failedWithCompensation}$
failedWithoutCompensation-Count	Anzahl der Sagas mit Endzustand $q_{failedWithoutCompensationCount}$
hasCorrectEndstateCount	Anzahl der Sagas mit dem erwarteten Endzustand des Testfalls
containsAllExpectedLogs-Count	Anzahl der Sagas, deren Transaktionslog alle erwarteten Logs des Testfalls aufweist
isSuccessfullTestInstance-Count	Anzahl der Sagas mit dem erwarteten Endzustand und erwarteten Transaktionslogs des Testfalls

Für jeden dieser Werte gibt wir einen normalisierter Wert berechnet, der ins Verhältnis zum Messwert *totalCount* gesetzt wird. Da diese Werte ausschließlich aus der Sicht des Koordinators gemessen werden, geben diese Werte lediglich Auskunft über den erreichten Endzustand einer Saga. Eine Saga mit dem Endzustand $q_{Success}$ ist nicht gleichbedeutend mit einer konsistenten LLT.

TransactionAnalysisResult

Um Aussagen über die Konsistenz einer LLT zu treffen, werden die durchgeführten Transaktionen, die im Transaktionslog des Koordinators festgehalten sind, mit den durchgeführten Transaktionen der Teilnehmerservices verglichen. Für jede Saga kann der Unterschied zwischen Transaktionsanzahl aus Koordinatorsicht und aus Teilnehmerservicesicht berechnet werden. Enthält eine Saga mindestens eine Transaktion, bei der dieser Unterschied auftritt, wird sie als inkonsistent bezeichnet. Die Summe der Unterschiede pro Transaktionstyp kann ebenfalls gebildet werden. Es ergibt sich folgende Tabelle:

Messwert	Beschreibung
diffRemoveMoney-Transaction	Summe der Unterschiede über alle Sagas für Transaktionstyp RemoveMoney
diffAddMoneyTransaction	Summe der Unterschiede über alle Sagas für Transaktionstyp AddMoney
diffRemoveMoney-CompensationTransaction	Summe der Unterschiede über alle Sagas für Transaktionstyp RemoveMoneyCompensation
diffAddMoney-CompensationTransaction	Summe der Unterschiede über alle Sagas für Transaktionstyp AddMoneyCompensation
diffBlockArticles-Transaction	Summe der Unterschiede über alle Sagas für Transaktionstyp BlockArticles
diffStartShipment-Transaction	Summe der Unterschiede über alle Sagas für Transaktionstyp StartShipment
diffBlockArticles-CompensationTransaction	Summe der Unterschiede über alle Sagas für Transaktionstyp BlockArticlesCompensation
diffStartShipment-CompensationTransaction	Summe der Unterschiede über alle Sagas für Transaktionstyp StartShipmentCompensation
consistentSagas	Anzahl der konsistenten Sagas

ExecutionTimeAnalysisResult

Schlussendlich wird ein weiteres Messergebnis definiert. Dieses Ergebnis dient dazu, die verschiedenen DEAs hinsichtlich ihrer Laufzeit zu untersuchen. Dabei wird die Differenz der ersten und letzten Transaktion im Log des Koordinators gebildet.

Messwert	Beschreibung
minRuntime	minimale Laufzeit aller Sagas [s]
maxRuntime	maximale Laufzeit aller Sagas [s]
avgRuntime	durchschnittliche Laufzeit aller Sagas [s]
medianRuntime	50% Quantil der Laufzeit aller Sagas [s]
upperQuartileRuntime	75% Quantil aller Sagas [s]
lowerQuartileRuntime	25% Quantil aller Sagas [s]

Die Analyse der Laufzeiten trägt keine Bedeutung für Aussagen bezüglich der Konsistenz oder Korrektheit einer LLT. Ein wesentliches Werkzeug für die Definition der verschiedenen DEAs sind Retries. Die ExecutionTimeAnalysisResults geben Auskunft, inwiefern die Laufzeit einer Saga beeinflusst wird.

5.9 Vermeidung von Konsistenzanomalien

In Abschnitt 2.5.1 wurden Isolationsanomalien vorgestellt, die in Transaktionen auftreten können, wenn die Isolation verletzt wird. Im Saga-Pattern ist die Isolation grundlegend nicht gewährleistet, da die von einer in Ausführung befindlichen Transaktion veränderten Daten für andere Transaktionen sichtbar sind. Es soll nun evaluiert werden, wie zu implementierende LLT mit diesen Anomalien umgeht. Dabei können die Anomalien jeweils innerhalb der lokalen Transaktionen im Teilnehmerservice als auch im Kontext der orchestrierten LLT auftreten. Diese Betrachtung soll durchgeführt werde, da das System in der Lage sein soll, mehrere parallele LLTs durchzuführen. Dies bedeutet, dass ein Mehrbenutzerbetrieb des Systems beachtet werden muss.

5.9.1 Anomalien innerhalb der lokalen Transaktionen

Jede lokale Transaktion läuft innerhalb eines Teilnehmerservices statt. Dort besteht immer die Möglichkeit, eine ACID-Transaktion zu verwenden, da eine lokale Transaktion immer nur die Daten in einer Datenquelle verändert.
Für die Identifizierung potentieller Isolationsanomalien sind alle Transaktionen zu ermitteln, die in paralleler Ausführung möglicherweise auf dieselben Ressourcen zugreifen können. Da die Implementierung der lokalen Transaktionen in den meisten Fällen die Id der LLT als Kontext verwendet, ist die Anzahl der Transaktionen, die die Isolation verletzen, sehr gering.

Isolationsanomalie RemoveMoney

Ein Beispiel dafür sind die Transaktionen im BankService, etwa *RemoveMoney*. Diese greifen auf die Tabelle *bank1usercredit* zu, welche sich nicht auf die LLT sondern auf den Nutzer der Bank bezieht. So können mehrere LLTs, die Spalte *credit* des selben Nutzers verändern. In Codebeispiel 5.8 ist eine Transaktion dargestellt, die neben Lost Updates auch zu Dirty Reads führen kann. In Zeile 3 wird der Credit des entsprechenden Nutzers ausgelesen. Innerhalb des BankServices wird der neue Betrag ausgerechnet und anschließend in Zeile 8 gesetzt. Falls eine andere Transaktion den Wert in der Spalte *credit* verändert, ist der neue gesetzte Credit falsch.

```
1  begin transaction;
2
3    select * from bank1usercredit where userid = @userid;
4
5    // fieldmapping inside the service
6    // calculating @newcredit
7
8    update bank1usercredit
9    set credit = @newcredit
10   where id = @id;
11
12   insert into bank1usertransaction (orderid, transactiontype,
       inserttime, useridfk)
13   values(@orderid, @transactiontype, @inserttime, @useridfk);
```

```
14
15 commit;
```

Codebeispiel 5.8: Fehlerhafte Implementierung der RemoveMoney-Transaktion

Um die Dirty Reads zu vermeiden, kann der neue Betrag direkt innerhalb des Statements gesetzt werden. Codebeispiel 5.9 behebt das Problem der Dirty Reads und liest und setzt den neuen Wert in einem Statement. Dies löst die Isolationsanomalie ohne die Anpassung des Isolationslevels. Ein alternativer Lösungsansatz, der die originale Implementierung aus Codebeispiel 5.8 verwendet, ist das Anheben des Isolationslevels. Damit verbunden sind jedoch Einschränkungen in der Performance, weshalb die erste Lösung Teil der Umsetzung ist.

```
1  begin transaction;
2
3    update bank1usercredit
4    set credit = credit - @price
5    where userid = @userid;
6
7    insert into bank1usertransaction (orderid, transactiontype,
       inserttime, useridfk)
8    values(@orderid, @transactiontype, @inserttime, @useridfk);
9
10 commit;
```

Codebeispiel 5.9: Implementierung der RemoveMoney-Transaktion ohne Dirty Reads

Weitere Isolationsanomalien in den lokalen Transaktionen der LLT Die beschriebenen Eigenschaften für das Auftreten von Anomalien sind ebenfalls für alle anderen Transaktionen des BankServices sowie für die Transaktion *BlockArticles* inklusive Kompensierung *BlockArticlesCompensation*. Dort greifen Transaktionen auf die gemeinsame Ressource aus der Tabelle *articlestock* zu.
Die Lösung für diese Anomalien sind mit dem beschriebenen Vorgehen für *RemoveMoney* lösbar.

5.9.2 Anomalien innerhalb der LLT

Es wird nun davon ausgegangen, dass keine Anomalien innerhalb der Teilnehmerservices auftreten können. Die LLT hat jedoch im Gegensatz zu den lokalen Transaktionen keine Möglichkeit, Isolationslevel zu setzen, um Anomalien zu vermeiden.
Die Quellen für Isolationsanomalien erfüllen auch hier die Eigenschaft, dass die Ressourcen von mehreren Transaktionen manipuliert werden können. Diese wurden bereits identifiziert: *BlockArticles*, *AddMoney*, *RemoveMoney* und die korrespondierenden Kompensierungen.

Fehleranfällige Kommunikation

In Abbildung 5.9 ist eine alternative Implementierung der Kommunikation zwischen

dem SEC und BankService abgebildet, die zu Dirty Reads führt. Das Beispiel bildet das selbe Problem aus Codebeispiel 5.8 ab, welches auf Ebene des BankServices über Zusammenführung von Lese- und Schreibvorgang gelöst wurde.

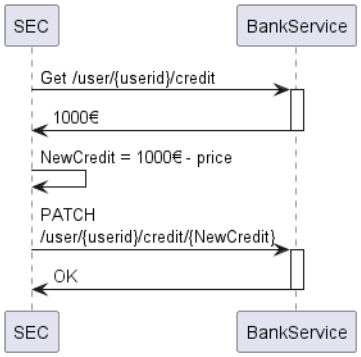

Abbildung 5.9: Sequenzdiagramm für eine Implementierung für *RemoveMoney*, die Dirty Reads verursacht

Veränderung von Daten außerhalb des Prozesses

Zusätzlich zu der Abbildung einer fehlerhaften Kommunikation zwischen SEC und Teilnehmerservice können weitere Anomalien auftreten, wenn weitere Prozesse Daten im System verändern. Das können Prozesses außerhalb der Saga sein. Ermöglicht der BankingService beispielsweise das Schließen eines Kundenkontos, treten ebenfalls Isolationsanomalien auf. Ist ein Kunde Bestandteil einer laufenden LLT und schließt während der Ausführung sein Konto, kann es unter Umständen dazu führen, dass die Konsistenz nicht gewährleistet werden kann. Ein weiteres Beispiel ist die Änderung des Preises eines Artikels in dem ArticleService nachdem eine LLT bereits den veralteten Preis validiert hat. Diese Anomalie stellt eine Instanz des Problems des Unrepeatable Reads dar.

Verhinderung der Anomalien innerhalb der LLT

Um die Isolationsanomalien auf Ebene der LLT zu vermeiden gibt es keine konkrete Lösung über eine Abstrahierung wie auf Ebene der lokalen Transaktion. Dirty Reads und Lost Updates können bedingt vermieden werden, indem die Kommunikation zwischen dem SEC und dem Teilnehmerservice möglichst wenige Schritte aufweist. Alle Veränderungen, die sich lediglich auf die Daten in einem Teilnehmerservice beziehen, sollten durch den Teilnehmerservice gesteuert werden. Dieser hat die Möglichkeit unter Verwendung von ACID-Transaktionen seine Daten konsistent zu halten, ohne die Isolation zu verletzen.
Isolationsanomalien, die durch andere Prozesse eingeführt werden, können nur verhindert werden, indem das System mit Einschränkungen versehen wird. Eine Einschränkung für das Schließen eines Kontos wäre die Bedingung, dass für den entsprechenden Nutzer keine laufende LLTs existieren. Diese Einschränkungen stellen eine Form der manuellen Implementierung einer Isolationsstufen dar.

5.10 Implementierung SmBasic

Der DEA der ersten Implementierung wird im Folgenden als SmBasic bezeichnet. Dieser DEA soll das grundlegende Prinzip des Saga-Patterns unter Verwendung der Backward-Recovery implementieren. Jeglicher Fehler führt dazu, dass die SEC eine Kompensierung der Saga verfolgt.

5.10.1 Strategie für die Konstruierung des DEAs SmBasic

Alle Erfolge eines Ts führen zum folgenden T. Alle anderen Ergebnisse führen für Ts zu einem Zustandswechsel zu dem nächstkleineren C. Erfolgreiche Cs führen zum nächstkleinerem C. Alle anderen Ergebnisse führen für Cs zu einem Zustandswechsel in den Zustand *FailedWithCompensation*. Das letzte T führt bei Erfolg zu einem Übergang in den Zustand *Done*. Ebenso führt C_1 bei Erfolg zu einem Übergang in den Zustand *FailedWithCompensation*.

Ergebnistyp	Zustand	Ergebnis e	Folgezustand
API-Ergebnis	T_n	$e \in \{Tn_{200}\}$	T_{n+1}
API-Ergebnis	T_n	$e \notin \{Tn_{200}\}$	C_{n-1}
IPE	T_n	$e \in \{Tn_{Success}\}$	T_{n+1}
IPE	T_n	$e \notin \{Tn_{Success}\}$	C_{n-1}
API-Ergebnis	C_n	$e \in \{Cn_{200}\}$	C_{n-1}
API-Ergebnis	C_n	$e \notin \{Cn_{200}\}$	*FailedWithoutCompensation*
IPE	C_n	$e \in \{Cn_{Success}\}$	C_{n-1}
IPE	C_n	$e \notin \{Cn_{Success}\}$	*FailedWithoutCompensation*

Aus diesem Regelwerk ergibt sich der in Abbildung 5.10 dargestellte DEA.

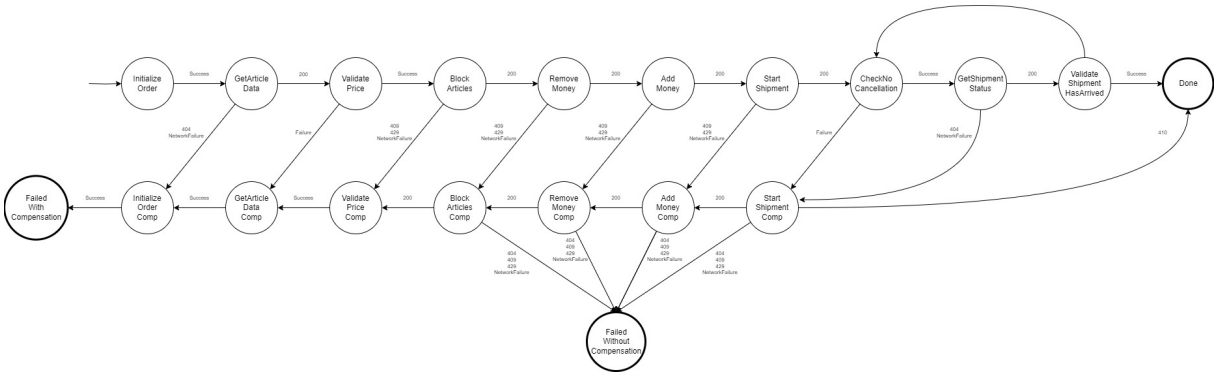

Abbildung 5.10: DEA für SmBasic

5.10.2 StateAnalysisResult

Testfall FinishOrders

Folgende Tabelle bildet das StateAnalysisResult der Messung des SmBasic in Testfall FinishOrders.

Messwert	S1	S2	S3
successfullPercentage	0.80	0.42	0.19
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	0.20	0.53	0.72
failedWithoutCompensation-Percentage	0.0	0.06	0.10
hasCorrectEndstatePercentage	0.80	0.42	0.19
containsAllExpectedLogs-Percentage	0.80	0.38	0.15
isSuccessfullTestInstance-Percentage	0.80	0.38	0.15

Die Messung zeigt, dass bereits in Testszenario 1 nur 80% der Sagas zum erwarteten Endzustand führen. Der Anteil der Sagas, die den Zustand *Success* erreichen, wird mit jedem Testszenario geringer: 80% in Testszenario S1, 42% in S2 und 19% in S3.
Das ist damit zu begründen, dass jeglicher Fehler in der Ausführung des SmBasic zur Kompensierung führt. Zu den Fehlern, die zur Kompensierung (für Ts) oder zum Zustand *FailedWithoutCompensation* (für Cs) führen, gehören neben den Fehlern, die einen kritischen Konflikt ausdrücken auch vorübergehende Fehler (API-Ergebnisse mit tn_{429}) und in S2 und S3 Netzwerkfehler.
Dies drückt sich außerdem in dem Messwert *failedWithoutCompensation* aus, der ebenfalls in jedem Testszenario größer wird.
In Tabelle 5.9 ist das Transaktionslog einer Saga abgebildet, die aufgrund eines vorübergehenden Fehlers abgebrochen wurde.

Zustand	Ergebnis
InitializeOrder	InitializeSagaSuccess
GetArticleData	GetProductData200
ValidatePrice	ValidatePriceSuccess
BlockArticles	BlockArticles429
ValidatePriceCompensation	ValidatePriceCompensationSuccess
GetArticleDataCompensation	GetProductDataCompensationSuccess
InitializeOrderCompensation	InitializeSagaCompensationSuccess

Tabelle 5.9: Transaktionslog für Saga im Testfall 1 und Testszenario 1

Testfall CancelOrders

In **??** ist das StateAnalysisResult für den Testfall CancelOrders dargestellt. Hier zeigt sich die Berechtigung des Messwertes *isSuccessfullTestInstancePercentage*. Der erwartete Endzustand *FailedWithCompensation* wird in diesem Testfall erwartet und in Testszenario 1 in 95% aller Sagas erreicht. Der Messwert *containsAllExpectedLogsPercentage* zeigt, dass in Testszenario 1 lediglich 87% die erwarteten Logs enthalten. Das bedeutet, dass ein

Anteil der Sagas mit Endzustand *FailedWithCompensation* aufgrund eines Fehlers zur Kompensierung gewechselt haben und nicht wie erwartet den Zustand *CheckNoCancellation* erreicht haben. Somit stellen diese Sagas keine erfolgreichen Testinstanzen mehr dar.

Messwert	S1	S2	S3
successfullPercentage	0.0	0.0	0.0
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	0.95	0.77	0.75
failedWithoutCompensation-Percentage	0.05	0.23	0.25
hasCorrectEndstatePercentage	0.95	0.77	0.75
containsAllExpectedLogs-Percentage	0.87	0.27	0.06
isSuccessfullTestInstance-Percentage	0.87	0.27	0.06

Abgebrochene Sagas

Der Messwert *failedWithoutCompensationPercentage* stellt den Anteil der Sagas dar, die aufgrund eines Fehlers während einer Kompensierung die Ausführung abbrechen. Im konstruierten SmBasic gibt es keine Retries. Das bedeutet, dass jede Saga, die zwei fehlerhafte Ergebnisse im Log enthält, in diesem Zustand landet. Ein Beispiel dafür ist im Transaktionslog in Tabelle 5.23 abgebildet.

Zustand	Ergebnis
InitializeOrder	InitializeSagaSuccess
GetArticleData	GetProductData200
ValidatePrice	ValidatePriceSuccess
BlockArticles	BlockArticles200
RemoveMoney	RemoveMoney200
AddMoney	AddMoney200
StartShipment	StartShipment200
CheckNoCancellation	CheckNoCancellationSuccess
GetShipmentStatus	GetShipmentStatus200
ValidateShipmentHasArrived	ShipmentHasArrivedFailure
CheckNoCancellation	CheckNoCancellationSuccess
GetShipmentStatus	GetShipmentStatusNetworkFailure
StartShipmentCompensation	StartShipmentCompensation200
AddMoneyCompensation	AddMoneyCompensationNetworkFailure

Tabelle 5.11: Transaktionslog für Saga im Testfall CancelOrders und Testszenario 2

5.10.3 TransactionAnalysisResult

In Tabelle 5.24 sind die TransactionAnalysisResults für beide Testfälle dargestellt.

	S1	S2	S3
FinishOrders	1	1	0.87
CancelOrders	1	1	0.81

In beiden Testfällen treten unter den Bedingungen von Testszenario 1 und 2 keine Inkonsistenzen auf. In Testszenario 3 führen 87% im Testfall *FinishOrders* und 81% im Testfall *CancelOrders* aller Sagas zu mindestens zu einem inkonsistenten Zustand innerhalb der Teilnehmerservices.
Neben den Sagas, die im Endzustand *FailedWithoutCompensation* landen, führen auch Sagas zu einem inkonsistenten Systemzustand, die nach einem Netzwerkfehler mit der Kompensierung fortfahren. Ein Beispiel dafür ist das Transaktionslog in Tabelle 5.13. Der BankService hat die lokale Transaktion *RemoveMoney* erfolgreich ausgeführt. Die Response ging verloren, weshalb der Koordinator mit der Kompensierung fortgefahren hat. Die lokale Transaktion *RemoveMoney* wurde nicht kompensiert. Fachlich stellt dieses Saga-Log einen Bestellprozess dar, der abgebrochen wurde. Das Geld des Kundenkontos wurde abgebucht und nicht rückerstattet. Der Saga-Prozess hat keine Kenntnis von diesem fachlichen Fehler.

Zustand	Ergebnis
InitializeOrder	InitializeSagaSuccess
GetArticleData	GetProductData200
ValidatePrice	ValidatePriceSuccess
BlockArticles	BlockArticles200
RemoveMoney	RemoveMoneyNetworkFailure
BlockArticlesCompensation	BlockArticlesCompensation200
ValidatePriceCompensation	ValidatePriceCompensationSuccess
GetArticleDataCompensation	GetProductDataCompensationSuccess
InitializeOrderCompensation	InitializeSagaCompensationSuccess

Tabelle 5.13: Transaktionslog einer Saga für *CancelOrders* im Testszenario 3

5.11 Implementierung SmBasicSafeRetries

Der nächste DEA verwendet den SmBasic als Grundlage und soll das Problem des vorzeitigen Abbruchs verhindern. Dabei sollen keine neuen Inkonsistenzquellen eingeführt werden.

5.11.1 Verhinderung eines Vorzeitigen Abbruchs

Netzwerkfehler

Für alle Transaktionen, die nicht zu einer Änderung des Systemzustands eines Teilnehmerservices führen, können Retries im Falle eines Netzwerkfehlers eingeführt werden. Dazu gehören *GetArticleData* und *GetShipmentStatus*.

Lastfehler

Behindern sich mehrere parallele lokale Transaktionen innerhalb eines Teilnehmerservices, kommt es zu einer Race Condition (Wettlaufsituation). Dabei gewinnt die erste lokale Transaktion T1 und kann wie gewohnt abgeschlossen werden. Alle lokalen Transaktionen, die innerhalb der Bearbeitungszeit von T1 auf die gesperrten Ressourcen zugreifen, schlagen fehl. Die aus diesem Fehler resultierende Response enthält den Http-Statuscode 429 und wird vom Koordinator auf ein entsprechendes Ergebnis gemappt. Dieses Verhalten ist für den StockService und die BankServices implementiert.
Antwortet ein Service mit einer solchen Response, ist der Fehler durch einen Retry auflösbar. Das bedeutet, dass für die lokalen Transaktionen *BlockArticles*, *RemoveMoney*, *AddMoney* und *StartShipment* ein Retry eingeführt werden kann. Ebenso kann diese Stategie auf alle zugehörigen Kompensierungen angewendet werden.

DEA SmBasicSafeRetries

Aus den beschriebenen Anpassungen ergibt sich der SmBasicSafeRetries in Abbildung 5.11.

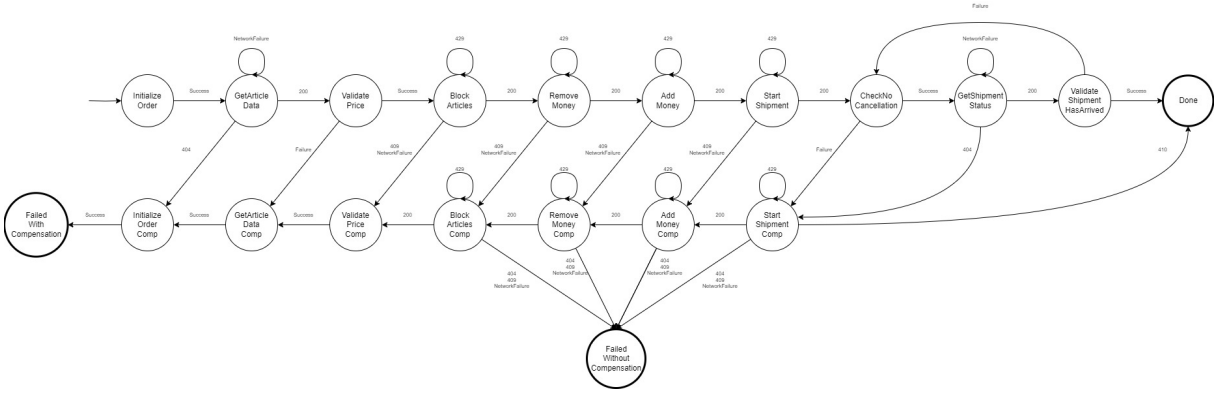

Abbildung 5.11: DEA für SmBasicSafeRetries

5.11.2 StateAnalysisResult

Das StateAnalysisResult von SmBasicSafeRetries und SmBasic sind wie erwartet sehr ähnlich. Im Testfall FinishOrders ist zu sehen, dass die Werte für *isSuccessFullTestInstancePercentage* deutlich höher sind. Die vorzeitigen Abbrüche aufgrund von Last- und Netzwerkfehlern an unkritischen nicht-transaktionellen Stellen wurden durch das Einführen von Retries eliminiert. In Testszenario 1 führt dies dazu, dass 100% der Sagas erfolgreich sind. Das gilt auch für den Testfall CancelOrders.

Testfall FinishOrders

Messwert	S1	S2	S3
successfullPercentage	1.0	0.65	0.39
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	0.0	0.31	0.47
failedWithoutCompensation-Percentage	0.0	0.05	0.15
hasCorrectEndstatePercentage	1.0	0.65	0.39
containsAllExpectedLogs-Percentage	1.0	0.65	0.39
isSuccessfullTestInstance-Percentage	1.0	0.65	0.39

Testfall CancelOrders

Messwert	S1	S2	S3
successfullPercentage	0.0	0.0	0.0
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	1.0	0.77	0.64
failedWithoutCompensation-Percentage	0.0	0.24	0.37
hasCorrectEndstatePercentage	1.0	0.77	0.64
containsAllExpectedLogs-Percentage	1.0	0.47	0.17
isSuccessfullTestInstance-Percentage	1.0	0.47	0.17

5.11.3 TransactionAnalysisResult

Das TransactionAnalysisResult zeigt, dass der Messwert *consistentSagasPercentage* im Vergleich zu SmBasic sinkt. Das ist damit zu erklären, dass die vorzeitigen Abbrüche im SmBasic das Ergebnis deutlich maskiert haben. Durch die vorzeitigen Abbrüche im SmBasic war die Gesamtzahl an Transaktionen niedriger. Das bedeutet, dass es weniger Möglichkeiten gab, einen Fehler zu verursachen, der zu Inkonsistenz führt.
Die Inkonsistenz ist bei dem Testfall für CancelOrders niedriger als bei FinishOrders. Das hat denselben Grund, der Testfall CancelOrders durchläuft zusätzlich zu den Ts immer die zugehörigen Cs und kann somit in häufigeren Zuständen Inkonsistenzen verursachen.

	S1	S2	S3
FinishOrders	1	1	0.74
CancelOrders	1	1	0.54

5.12 Implementierung SmBasicNetworkFailureUnlimitedRetries

Der SmBasicNetworkFailureUnlimitedRetries baut auf das Konzept des vorherigen DEAs SmBasicSafeRetries auf. Es hat sich gezeigt, dass der Anteil der erfolgreichen Sagas erhöht werden kann, indem auf Netzwerkfehler per Forwardrecovery reagiert wird. Das Ziel dieses DEAs ist, den Anteil der vorzeitig abgebrochenen Sagas zu erhöhen. Dafür werden für die Ts *BlockArticles*, *RemoveMoney*, *AddMoney* und *StartShipment* sowie für die zugehörigen Cs ein Retry für Netzwerkfehler eingeführt. Es entsteht der DEA in Abbildung 5.12.

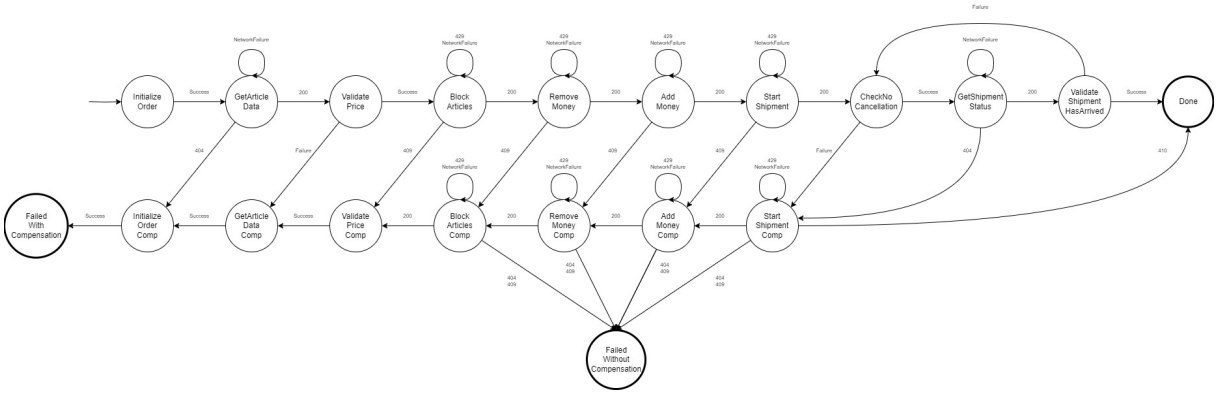

Abbildung 5.12: DEA für SmBasicNetworkfailureUnlimitedRetries

5.12.1 StateAnalysisResult

Das StateAnalysisResult zeigt, dass das Einführen von Retries für transaktionelle Zustände dazu führt, dass der gewünschte Endzustand öfter erreicht wird. Testszenario 3 enthält in Testfall FinishOrders 90% und in Testfall CancelOrders 74% erfolgreiche Testinstanzen. Aus Sicht des Koordinators kann auf Netzwerkfehler mit einem Retry erfolgreich reagiert werden, um zum nächsten Zustand zu gelangen.

Testfall FinishOrders

Messwert	S1	S2	S3
successfullPercentage	1.0	1.0	0.91
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0

Messwert	S1	S2	S3
failedWithCompensation-Percentage	0.0	0.0	0.0
failedWithoutCompensation-Percentage	0.0	0.0	0.1
hasCorrectEndstatePercentage	1.0	1.0	0.91
containsAllExpectedLogs-Percentage	1.0	1.0	0.91
isSuccessfullTestInstance-Percentage	1.0	1.0	0.91

Testfall CancelOrders

Messwert	S1	S2	S3
successfullPercentage	0.0	0.0	0.0
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	1.0	1.0	0.74
failedWithoutCompensation-Percentage	0.0	0.0	0.27
hasCorrectEndstatePercentage	1.0	1.0	0.74
containsAllExpectedLogs-Percentage	1.0	1.0	0.74
isSuccessfullTestInstance-Percentage	1.0	1.0	0.74

5.12.2 TransactionAnalysisResult

In Tabelle 5.24 sind die TransactionAnalysisResults für beide Testfälle dargestellt.

	S1	S2	S3
FinishOrders	1	1	0.67
CancelOrders	1	1	0.36

Hier zeigt sich, dass im Vergleich zum SmBasicSafeRetries die Kennzahl *consistentSagas-Percentage* deutlich kleiner wird. Konsistenz des Systems hat sich durch die Einführung der Änderungen verringert.
Aus Sicht des Koordinators ist der SmBasicNetworkFailureUnlimitedRetries erfolgreicher, der Vergleich von der Koordinator- und Teilnehmersicht zeigt jedoch, dass Transaktionen stattfinden, die der Koordinator nicht kennt.

Es zeigt sich außerdem, dass diese Fehler lediglich in Testszenario 3 auftreten. Die fehlerverursachenden Transaktionen sind genau die, bei denen die Response auf dem Rückweg verloren geht. Der Koordinator interpretiert dies im Kontext dieses Zustandsautomaten als wiederholbar und führt die Transaktion erneut aus. Dadurch wird im entsprechenden Teilnehmerservice die Transaktion mehr als einmal ausgeführt.

5.13 Wiederholbarkeit

In der Analyse des SmBasicNetworkFailureUnlimitedRetries hat sich gezeigt, dass die für die 4 Ts *BlockArticles*, *RemoveMoney*, *AddMoney* und *StartShipment* und ihre zugehörigen Cs eingeführten Retries zu inkonsistentem Verhalten in Testszenario 3 geführt haben. Das Problem besteht darin, dass lokale Transaktionen bei Wiederholung erneut ausgeführt werden. Dem Koordinator fehlt die Information, ob die lokale Transaktion im Fall eines Netzwerkfehlers bereits ausgeführt wurde oder nicht.

5.13.1 Implementierung von idempotentem Verhalten

Es wird ein idempotentes Verhalten benötigt, bei dem der Koordinator gefahrlos Aufrufe an den Teilnehmerservice wiederholen kann. Dazu muss jeder Teilnehmerservice die ausgeführten Transaktionen mit einer zugehörigen RequestId speichern. Taucht ein Request mit einer bereits verwendeten RequestId auf, wird der Request nicht erneut bearbeitet, sondern eine Response mit dem Statuscode 208 zurückgegeben. Damit erhält der Aufrufer die Information, dass der Request bereits erfolgreich verarbeitet wurde.
Um idempotentes Verhalten zu implementieren, muss ein Service jeden erfolgreichen Request mit der zugehörigen RequestId abspeichern. Diese Information werden in einer Tabelle persistiert. Die Prüfung, ob ein Request prozessiert werden kann, ist eine Fallunterscheidung. Wenn ein Eintrag in der Idempotenztabelle existiert, der in RequestId und Transaktion übereinstimmt, wird der Request mit 208 abgelehnt. Existiert kein solcher Datensatz, kann die Transaktion prozessiert werden.

5.13.2 Idempotente DEAs

Die drei DEAs SmBasic, SmBasicSafeRetries und SmBasicNetworkFailureUnlimitedRetries verwenden die nicht-idempotenten Implementierungen der Teilnehmerservices. Damit nicht-idempotente und idempotente DEAs verglichen werden können, sind alle Teilnehmerservices hinsichtlich Idempotenz konfigurierbar. Die Fallunterscheidung, die bereits ausgeführte Transaktionen ablehnt, wird bei einer nicht-idempotenten Konfiguration des Services übersprungen. Die folgenden zwei DEAs verwenden eine idempotente Konfiguration.

5.14 Implementierung SmIdempotencyBackwardRecovery

Der erste idempotente DEA wird als SmIdempotencyBackwardRecovery bezeichnet. Im Falle eines Netzwerkfehlers bei einem T wird ein Übergang zum entsprechenden C eingeführt; die Saga wird also versucht zurückzurollen, unabhängig ob das T ausgeführt

wurde oder nicht. In Abbildung 5.13 ist der Fall abgebildet, dass ein T einen Netzwerkfehler verursacht ohne eine Zustandsänderung im Teilnehmerservice zu bewirken. Abbildung 5.14 zeigt den Ablauf im Fall eines Netzwerkfehlers mit Zustandsänderung.

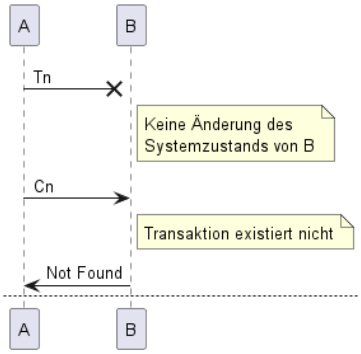

Abbildung 5.13: Sequenzdiagramm für Idempotentes Verhalten bei wiederholten Anfragen in Szenario 3

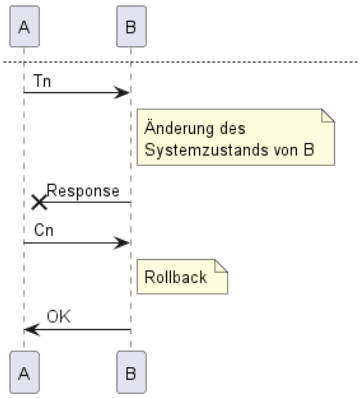

Abbildung 5.14: Sequenzdiagramm für Idempotentes Verhalten bei wiederholten Anfragen in Szenario 3

Solche Fehler treten jedoch auch bei den Cs auf. Deshalb müssen diese auch addressiert werden. Bei Fehlschlagen eines Cs könnte in den Zustand FailedWithoutCompensation gewechselt werden. Dies soll jedoch verhindert werden. Da die Teilnehmerservices idempotentes Verhalten unterstützen, können Cs wiederholt werden, bis ein eindeutiges Ergebnis vorliegt. In folgenden Fällen kann ein Übergang zum nächsten C eingeführt werden:

- 200: Die Transaktion wurde erfolgreich kompensiert
- 208: Die Transaktion wurde in einem vorherigen Schritt bereits erfolgreich kompensiert
- 404: Die Transaktion ist nicht bekannt und muss nicht kompensiert werden

Im Falle eines Konflikts (409) muss jedoch weiterhin auf den Zustand FailedWithoutCompensation gewechselt werden.
Der resultierende Zustandsautomat ist in Abbildung 5.15 abgebildet.

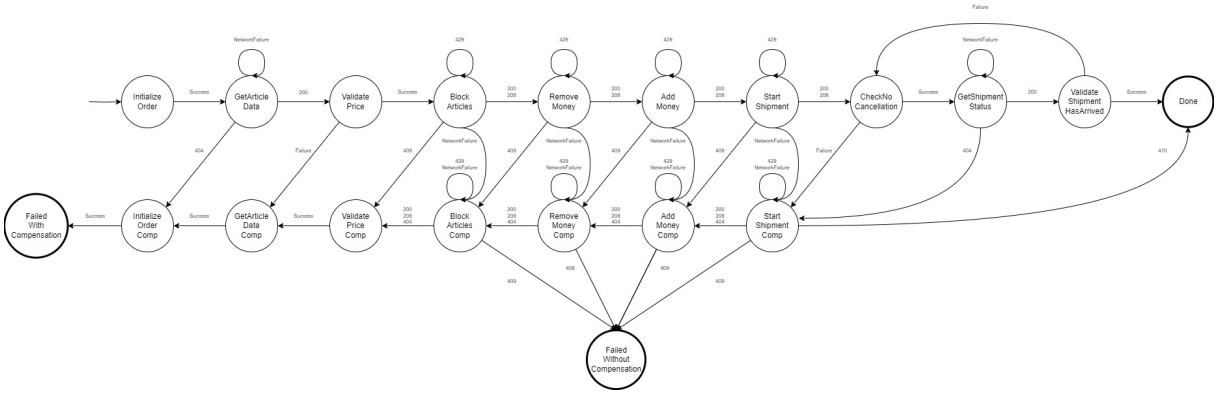

Abbildung 5.15: DEA für SmIdempotencyBackwardRecovery

5.14.1 StateAnalysisResult

Das StateAnalysisResult zeigt einen hohen Anteil an vorzeitig abgebrochenen Sagas. Das ist mit dem Konzept der Backwardrecovery begründet. Lediglich Testszenario 1 liefert in beiden Testfällen eine Erfolgsrate von 100%.

Testfall FinishOrders

Messwert	S1	S2	S3
successfullPercentage	1.0	0.63	0.44
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	0.0	0.38	0.57
failedWithoutCompensation-Percentage	0.0	0.0	0.0
hasCorrectEndstatePercentage	1.0	0.63	0.39
containsAllExpectedLogs-Percentage	1.0	0.63	0.44
isSuccessfullTestInstance-Percentage	1.0	0.63	0.44

Testfall CancelOrders

Messwert	S1	S2	S3
successfullPercentage	0.0	0.0	0.0
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	1.0	1.0	1.0

Messwert	S1	S2	S3
failedWithoutCompensation-Percentage	0.0	0.0	0.0
hasCorrectEndstatePercentage	1.0	1.0	1.0
containsAllExpectedLogs-Percentage	1.0	0.68	0.44
isSuccessfullTestInstance-Percentage	1.0	0.68	0.44

5.14.2 TransactionAnalysisResult

In Tabelle 5.24 sind die TransactionAnalysisResults für beide Testfälle dargestellt. Überraschend ist hier der niedrige Anteil an konsistenten Sagas in Testszenario 2 und 3.

	S1	S2	S3
FinishOrders	1	1	0.74
CancelOrders	1	1	0.74

Die Ursache dafür liegt in der Berechnung der Kennzahl sowie dem Aufbau des Zustandsautomaten. Für die Berechnung der Kennzahl werden die Differenzen der jeweiligen Transaktionen nach Koordinator- und Teilnehmersicht gebildet. Nur wenn diese für jedes T und jedes C übereinstimmen fließt das Ergebnis als erfolgreich in die Kennzahl ein.
Konkret soll nun diese Messungenauigkeit betrachtet werden. Wird die Anzahl einer Transaktion in Koordinatorsicht berechnet, wird das Transaktionslog verwendet. Dabei soll das in **??** dargestellte Transaktionslog verwendet werden.
Das Log zeigt eine Saga, die aufgrund eines Netzwerkfehlers im Zustand *StartShipment* zur Kompensierung wechselt. Zur Berechnung für die Kennzahl *consistentSagasPercentage* wird auch die Anzahl der Transaktionen mit dem Zustand *StartShipment* und dem Ergebnis *StartShipment200* oder *StartShipment208* berechnet. Für den SmIdempotencyBackwardRecovery bedeutet ein Netzwerkfehler in den kritischen Ts, dass das dazugehörige C die Änderungen kompensiert oder fortfährt, falls der Netzwerkfehler keine Zustandsänderungen bewirkt hat. Das Zählen der Transaktionen mit dem Zustand *StartShipment* und dem Ergebnis *StartShipment200* oder *StartShipment208* ist für diesen DEA also keine korrekte Berechnung, um Konsistenz zu messen.

Zustand	Ergebnis
InitializeOrder	InitializeSagaSuccess
GetArticleData	GetProductData200
ValidatePrice	ValidatePriceSuccess
BlockArticles	BlockArticles200
RemoveMoney	RemoveMoney200

Tabelle 5.23: Transaktionslog für Saga im Testfall CancelOrders und Testszenario 2

Zustand	Ergebnis
AddMoney	AddMoney200
StartShipment	StartShipmentNetworkFailure
StartShipmentCompensation	StartShipmentCompensationNetworkFailure
StartShipmentCompensation	StartShipmentCompensation208
AddMoneyCompensation	AddMoneyCompensation200
RemoveMoneyCompensation	RemoveMoneyCompensation200
BlockArticlesCompensation	BlockArticlesCompensation200
ValidatePriceCompensation	ValidatePriceCompensationSuccess
GetArticleDataCompensation	GetProductDataCompensationSuccess
InitializeOrderCompensation	InitializeSagaCompensationSuccess

Tabelle 5.23: Transaktionslog für Saga im Testfall CancelOrders und Testszenario 2

5.14.3 Alternative Sicherstellung der Konsistenz

Damit der Zustandsautomat trotz dieser verfälschten Messung bewertet werden kann, wird eine Alternative benötigt. Um eine Aussage über die Konsistenz zu treffen, kann der Inhalt der eigentlichen Tabellen der Services verwendet werden.

Alternative Konsistenzdefinition

Als Konsistenzkriterien dienen folgende Annahmen:

- Die Summe des Geldes im System bleibt gleich.
- Die Summer aller Artikel im System bleibt gleich.

Einordnung dieser Definition

Diese Vorgehensweise kann entscheiden, ob das System nach den definierten Regeln im konsistenten Zustand bleibt oder ob eine Transaktion das System in einen inkonsistenten Zustand überführt hat. Dies ist eine binäre Messung, über die Qualität der Konsistenz kann im vorliegenden System, unter Verwendung dieses Zustandsautomaten und unter Verwendung des aktuellen Versuchsaufbaus keine Aussage getroffen werden.
Nach den genannten Annahmen ist das System auch konsistent, wenn zwei parallele Sagas fehlschlagen und eine Saga *AddMoney* und die andere Saga *RemoveMoney* jeweils doppelt mit dem (zufällig) identischen Betrag ausführen. Die Menge des Geldes im System bleibt trotz Auftreten eines solchen hypothetischen Bugs konsistent.
Diese Erläuterung soll verdeutlichen, dass die sehr vage Definition von Konsistenz mit Vorsicht verwendet werden sollte. Für diesen Automaten soll dies jedoch ausreichen.

Messung

Die Messung erfolgt manuell. Die Summe des im System befindlichen Geldes wird vor Ausführung der Tests gemessen. Dabei ist jedes Konto mit einem Startcredit von 15000.00 initialisiert. Tritt ein Konsistenzfehler bei Verwendung der BankServices auf, so wird dies in diesem Statement deutlich, wenn sich die zwei Werte *expectedCreditSum* und *actualCreditSum* unterscheiden.

```
select case when
  15000.00 * cast(
    (select count(*) from bank1user) +
    (select count(*) from bank2user) as float)
  = cast(
    (select sum(credit) from bank1credit) +
    (select sum(credit) from bank2credit) as float)
  then 1 else 0 end as mightBeConsistent;
```

Die Anzahl der Artikel wird mit einem vergleichbaren Statement auf Konsistenz überprüft. Die Summe der Artikel wird über die drei Tabellen des StockServices gezählt. Ändert sich diese Zahl nach Ausführung der Tests, ist ein Konsistenzfehler aufgetreten.

```
select case when
  (select 15000 * count(*) from articlestock as expectedarticlesum)
  =
  (select sum(amount) as amount from (
    select articleid as articleid, amount as amount from articlestock
    union
    select articleid, sum(amount) as amount from blockedarticles
       group by articleid
    union
    select articleidfk as articleid, sum(amount) as amount from
       shippedarticles group by articleidfk) as actualarticlesum
  )
  then 1 else 0 end as mightBeConsistent;
```

Ändert sich in beiden Tests die Summe nicht, ist laut Abschnitt 5.14.3 die notwendige Bedingung für Konsistenz erreicht. Das Gesamtergebnis berechnet sich durch folgendes Statement:

```
with moneyMightBeConsistent as (
  select case when
    (select 15000 * count(*) from articlestock as expectedarticlesum)
    =
    (select sum(amount) as amount from (
    select articleid as articleid, amount as amount from articlestock
    union
    select articleid, sum(amount) as amount from blockedarticles
       group by articleid
    union
    select articleidfk as articleid, sum(amount) as amount from
       shippedarticles group by articleidfk) as actualarticlesum
  )
  then 1 else 0 end as mightBeConsistent),
articleMightBeConsistent as (
  select case when
    15000.00 * cast(
      (select count(*) from bank1user) +
      (select count(*) from bank2user) as float)
```

```
18      = cast(
19        (select sum(credit) from bank1credit) +
20        (select sum(credit) from bank2credit) as float)
21      then 1 else 0 end as mightBeConsistent)
22  select moneyMightBeConsistent.mightBeConsistent as
        moneyMightBeConsistent,
23    articleMightBeConsistent.mightBeConsistent as
          articleMightBeConsistent,
24    case when
25      moneyMightBeConsistent.mightBeConsistent
26      =
27      articleMightBeConsistent.mightBeConsistent
28    then 1 else 0 end as mightBeConsistent
29  from moneyMightBeConsistent, articleMightBeConsistent;
```

Ergebnis
Die Messung zeigt, dass zumindest die Menge der Produkte und Geldbeträge konsistent bleibt.

	S1	S2	S3
moneyMightBeConsistent	1	1	1
articleMightBeConsistent	1	1	1
mightBeConsistent	1	1	1

Tabelle 5.24: Messwert *altConsistentSagasPercentage* für SmIdempotencyBackwardRecovery

5.15 Implementierung SmIdempotencyForwardRecovery

Die Auswertung der Ergebnisse des SmIdempotencyBackwardRecovery hat gezeigt, dass die Strategie der BackwardRecovery Konsistenz gewährleisten kann. Dabei trat jedoch erneut das Problem des vorzeitigen Abbruchs auf. Das erste Vorkommen eines Netzwerkfehlers innerhalb eines Ts führte zur Kompensierung. Der nächste zu implementierende Zustandsautomat verfolgt zwei Ziele: eine möglichst hohe Erfolgsrate sowie eine möglichst hohe Konsistenz. Mittels ForwardRecovery kann eine Weiterführung der Saga erzwungen werden. Abgebrochen wird lediglich in Zuständen, bei denen ein Ergebnis auftritt, das nicht behebbar ist.
Für Ts ist lediglich ein Ergebnis wie 409 unbehebbar und es wird zum vorherigen C gewechselt. Für Netzwerkfehler werden Retries eingeführt. Aufgrund der Idempotenz der Teilnehmerservices ist dies problemlos möglich. Nur bei einem eindeutigen Ergebnis wird fortgefahren (200, 208) oder abgebrochen (409).
Der resultierende Zustandsautomat SmIdempotencyForwardRecovery ist in Abbildung 5.16 abgebildet.

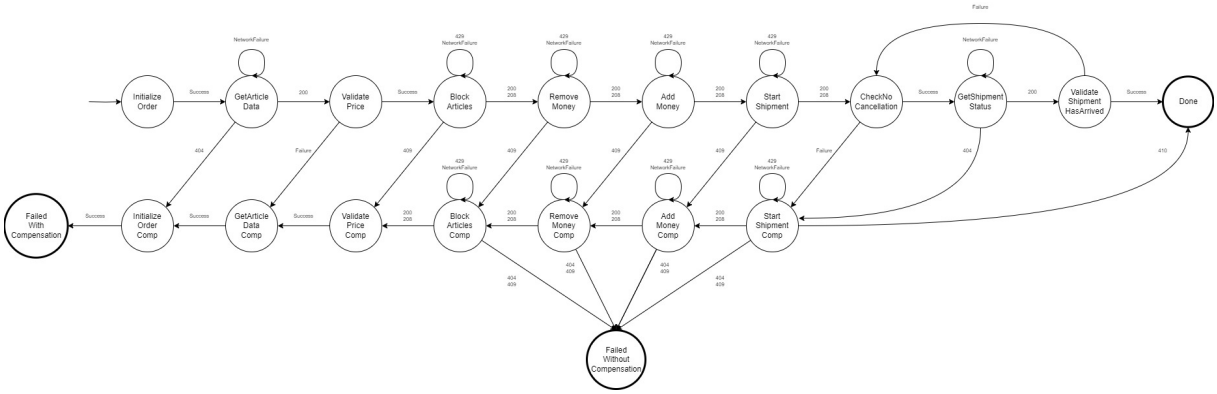

Abbildung 5.16: DEA für SmIdempotencyForwardRecovery

5.15.1 StateAnalysisResult

Das StateAnalysisResult zeigt, dass unabhängig vom Testfall und Testszenario alle Sagas im erwarteten Endzustand münden.

Testfall FinishOrders

Messwert	S1	S2	S3
successfullPercentage	1.0	1.0	0.91
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	0.0	0.0	0.0
failedWithoutCompensation-Percentage	0.0	0.0	0.1
hasCorrectEndstatePercentage	1.0	1.0	1.0
containsAllExpectedLogs-Percentage	1.0	1.0	1.0
isSuccessfullTestInstance-Percentage	1.0	1.0	1.0

Testfall CancelOrders

Messwert	S1	S2	S3
successfullPercentage	0.0	0.0	0.0
finishedPercentage	1.0	1.0	1.0
pendingPercentage	0.0	0.0	0.0
failedWithCompensation-Percentage	1.0	1.0	1.0
failedWithoutCompensation-Percentage	0.0	0.0	0.0

Messwert	S1	S2	S3
hasCorrectEndstatePercentage	1.0	1.0	1.0
containsAllExpectedLogs-Percentage	1.0	1.0	1.0
isSuccessfullTestInstance-Percentage	1.0	1.0	1.0

5.15.2 TransactionAnalysisResult

In **??** sind die TransactionAnalysisResults für beide Testfälle dargestellt. Es ist erkennbar, dass die Konsistenz in beiden Testfällen für alle Testszenarios eingehalten wird.

	S1	S2	S3
FinishOrders	1	1	1
CancelOrders	1	1	1

5.16 Laufzeitanalyse

Das primäre Werkzeug der Optimierung der Zustandsautomaten war die Einführung von Retries. Es ist zu erwarten, dass die Einführung von Retries die Laufzeit verlängert. Im Folgenden sind die Ergebnisse der Laufzeitanalyse per Boxplot dargestellt. Als Visualisierung wird ein modifizierter Boxplot verwendet. In jedem Diagramm sind das obere und untere Quartil, der Median, das Maximum und Minimum sowie Ausreißer ablesbar. Jeder Boxplot stellt die Werte eines DEAs dar. Aus Gründen der Übersicht wurden die langen Bezeichnungen der Zustandsautomaten in der Reihenfolge der Implementierung durch folgende Nummerierung ersetzt:

Zustandsautomat	Nummerierung
SmBasic	1
SmBasicSafeRetries	2
SmBasicNetworkFailureUnlimitedRetries	3
SmIdempotencyBackwardRecovery	4
SmIdempotencyForwardRecovers	5

5.16.1 Messungenauigkeiten

Die Durchführung des Versuchs ist ausgelegt auf die Vergleiche der unterschiedlichen DEAs hinsichtlich der Systemkonsistenz. Die Systemperformance war kein primäres Ziel des Versuchs. Die Messungen wurden zwar auf derselben Hardware ausgeführt, jedoch ist dies keineswegs mit einem Benchmark vergleichbar. Die Tätigkeit des Systems

wurde für die Durchführung des Versuchs nicht eingeschränkt und kann variieren. Deshalb dienen die folgenden Ergebnisse nur einem illustrativen Zweck. Die Einführung der Retries beeinflusst die Laufzeit des Systems. Die Bestimmung des exakten Einflusses ist hier nicht zu zeigen.

5.16.2 Testszenario 1

Im ersten Testszenario liegen die Laufzeiten alle sehr nah beieinander. Es sind lediglich kleine Unterschiede zu erkennen. Diese Unterschiede liegen in der Behandlung der Behandlung von Fehlern, die durch hohe Last verursacht werden (429). Der einzige DEA, bei dem diese Antworten zu vorzeitigen Abbrüchen führt, ist der SmBasic (1). Bei diesem DEA sind entsprechend viele Ausreißer nach unten zu erkennen. Dies gilt für beide Testfälle.
Die einzige weitere erkennbare Abweichung in Testszenario 1 ist die geringe Spanne von Laufzeitwerten in Testfall CancelOrders des SmIdempotencyBackwardRecovery. Die mittleren 50% der Ergebnisse liegen lediglich 13,38 Sekunden auseinander. Da in Testszenario 1 keine Netzwerkfehler auftreten, sind zumindest ähnliche Laufzeiten für DEAs SmIdempotencyBackwardRecovery und SmIdempotencyForwardRecovery zu erwarten. Die abweichenden Ergebnissen sind den in Abschnitt 5.16.1 beschriebenen Ursachen zuzuschreiben.

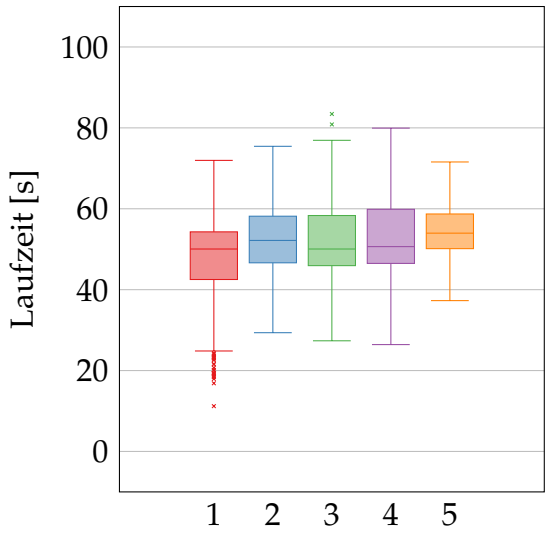

Abbildung 5.17: Boxplot FinishOrders in Szenario 1

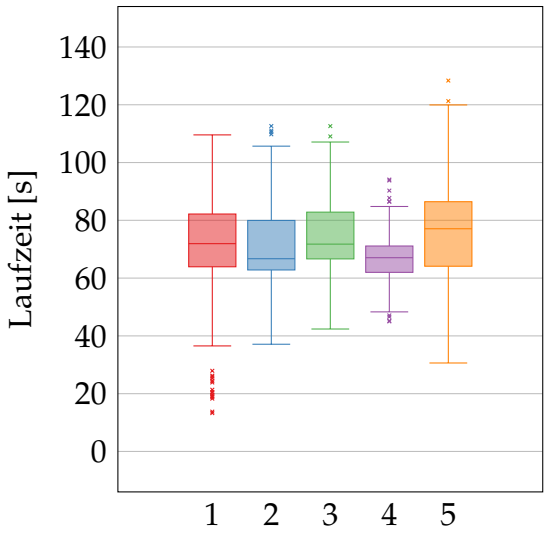

Abbildung 5.18: Boxplot CancelOrders in Szenario 1

5.16.3 Testszenario 2

Die Ergebnisse von Testszenario 2 zeigen wesentlich mehr ablesbare Unterschiede in der Laufzeit.

Steigende Laufzeit

Die Messwerte der ersten drei aufeinander basierenden Implementierungen (1, 2, 3) zeigen einen deutlichen Anstieg der Laufzeit. Der Median steigt von 22,41 s Sekunden auf 56,77 s.
Analog dazu steigt die Laufzeit von den zwei idempotenten DEAs (4, 5). Die niedrigere Laufzeit von SmIdempotencyBackwardRecovery ist ebenfalls mit vorzeitigen Abbrüchen zu erklären. Da diese auftreten, liegen die Werte für den vierten DEA in beiden Testfällen sehr weit auseinander.

Streuung der Werte

Die Streuung der Laufzeit ist als Abstand zwischen oberem und unterem Quartil ablesbar. In dem ersten DEA ist die Streuung sehr hoch. Die Messwerte für den zweiten DEA sind etwas weniger gestreut, da Retries für Lastfehler und nebenwirkungslose Aufrufe eingeführt wurden. Die Veränderung vom zweiten zum dritten DEA zeigt den selben Effekt.
Auch die zwei idempotenten DEAs bestätigen diese Beobachtung. Die Streuung für den mittels BackwardRecovery umgesetzten DEA ist deutlich größer als der per ForwardRecovery umgesetzte DEA.

Ausreißer

Besonders sind die Ausreißer des dritten und fünften DEAs. In beiden Fällen ist der Großteil der Werte sehr nah beieinander, jedoch sind eine Menge an Ausreißern über dem Boxplot erkennbar. Dies ist erneut mit der ForwardRecovery zu erklären. Da in den beiden DEAs die Netzwerkfehler zu einem unendlichen Retryverhalten führen, sind die Ausreißer genau die LLTs, die übermäßig viele Retries benötigt haben, jedoch trotzdem zum gewünschten Endzustand gefunden haben.

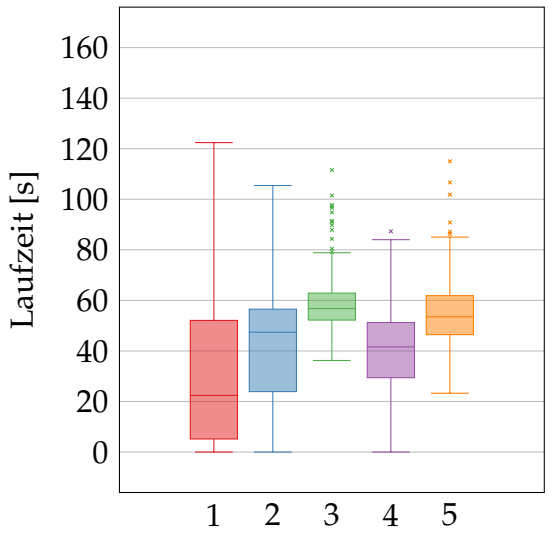

Abbildung 5.19: Boxplot FinishOrders in Szenario 2

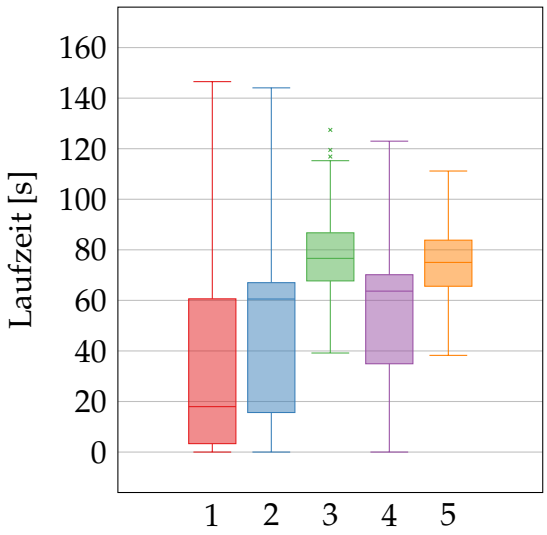

Abbildung 5.20: Boxplot CancelOrders in Szenario 2

5.16.4 Testszenario 3

Die in Testszenario 2 resultierenden Beobachtungen gelten ebenfalls für Testszenario 3. Die gemessenen Laufzeiten sind insgesamt höher. Die deutlich höheren Laufzeiten sind mit dem Auftreten von Timeouts zu erklären. Die auftretenden Ausreißer sind deshalb auch deutlich höher.

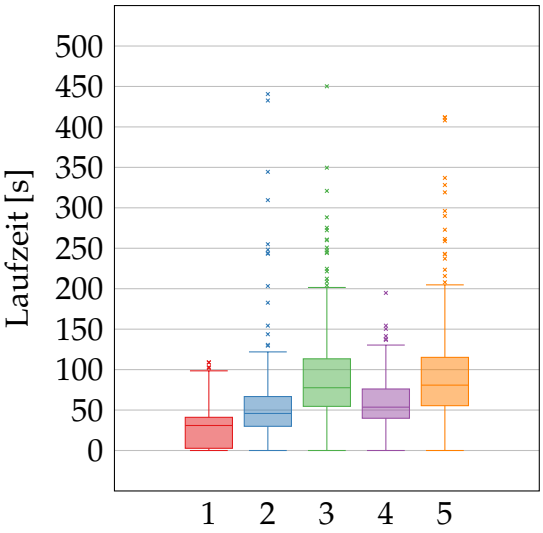

Abbildung 5.21: Boxplot FinishOrders in Szenario 3

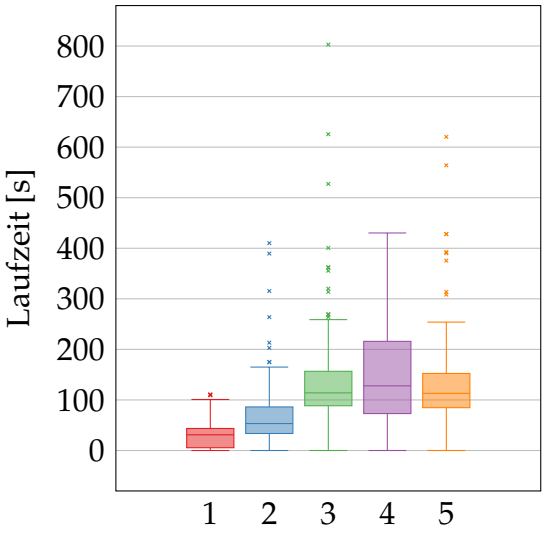

Abbildung 5.22: Boxplot CancelOrders in Szenario 3

6 Bewertung der Versuchsergebnisse

6.1 Versuchsergebnisse

Im Rahmen dieser Arbeit wurden folgende Schritte beschriebe:
?? wurde das von Garcia-Molina und Salem eingeführte Saga-Pattern und der Begriff der langlebigen Transaktionen vorgestellt. Die Möglichkeit, Transaktionen in Teiltransaktionen aufzuteilen und per neutralisierender Kompensierungen zurückzurollen, um die Atomaritätseigenschaft einer Transaktion nachzubilden wurde auf verteilte Transaktionen übertragen. Außerden wurde in Abschnitt 3.4 eine Formalisierung für Sagas vorgestellt, die eine LLT mit dem Modell der Zustandsautomaten verbindet.
In Abschnitt 4.1 wurde die These formulier, dass Netzwerkfehler in die Fehlerbehandlung des Saga-Patterns integrierbar sind, ohne Inkonsistenzen einzuführen. Anschließend wurde ein experimenteller Versuch geplant und durchgeführt. Ziel des Versuchs war die iterative Annäherung an ein verteiltes Saga-System, welches auch unter Auftreten von verschiedenen Netzwerkfehlern die LLT zum Abschluss kommen lässt, ohne das System in einen inkonsisten Zustand zu überführen.
Dazu wurde ein Bestellprozess eines E-Shops als abzubildende LLT gewählt. Die Implementierung dieses Systems geschah unter Verwendung einer Orchestrierung und ermöglichte die Simulation verschiedener Netzwerkfehler sowie verschiedenes Idempotenzverhalten der Teilnehmerservices.
In verschiedenen Testfällen und Testszenarios wurden Tests durchgeführt und gemessen. Die Ergebnisse dienten einer iterativen Verbesserung des ursprünglichen Systems.

Der Versuch adressierte folgende Herausforderungen:

- Vorzeitiger Abbruch der LLT
- Unbekannter Ausführungszustand der Teiltransaktion
- Gefahrlose Wiederholbarkeit der Teiltransaktionen
- Isolationsanomalien auf Ebene der Teilnehmerservices
- Isolationsanomalien auf Ebene der LLT

6.2 Zusammenhang zwischen CAP-Theorem und Saga-Pattern

In Abschnitt 2.4.3 wurde das CAP-Theorem vorgestellt. Es soll nun der Zusammenhang zwischen den Versuchsergebnissen und dem CAP-Theorem hergestellt werden.

Wenn die Kommunikation im Saga-Pattern zwischen dem Orchestrator und einem Teilnehmerservice ausfällt, dann entstehen zwei Partitionen.
Der dritte implementierte DEA SmBasicNetworkFailureUnlimitedRetries interpretierte die auftretenden Netzwerkfehler als Abbruchkriterium für die LLT. Dabei wurde sich für Verfügbarkeit gegenüber Konsistenz entschieden. Dieser DEA stellt also ein *AP-System* (Verfügbarkeit und Partitionstoleranz) dar.
Im Rahmen des Versuchsbeispiels war dies ein nicht annehmbares Ergebnis, da der inkonsistente Ablauf eines Bestellprozesses kein zufriedenstellendes Ergebnis ist. Im Anschluss wurden die zwei idempotenten DEAs implementiert. Beide DEAs interpretierten einen Netzwerkfehler als ein Ergebnis, welches den Ablauf der LLT zum Pausieren zwingt. Solange die Netzwerkpartition vorliegt, konnte der Prozess nicht zum erfolgreichen oder erfolglosen Abschluss gelangen. Diese Implementierung legte den Fokus auf Konsistenz. Somit stellen die zwei DEAs SmIdempotencyBackwardRecovery und SmIdempotencyForwardRecovery *CP-Systeme* (Konsistenz und Partitionstoleranz) dar.
Das CAP-Theorem bedingt somit das Design einer LLT mittels Saga-Pattern in einem verteilten System. Es müssen klare Entscheidungen für oder gegen Konsistenz und Verfügbarkeit getroffen werden. Diese Entscheidungen haben einen bedeutenden Einfluss auf das Design der verteilten Transaktion.

6.3 Beantwortung der Leitfragen

Unter Verwendung der im Versuch erbrachten Ergebnisse sollen nun die in Abschnitt 4.2 gestellten Leitfragen beantwortet werden.

6.3.1 Leitfrage 1

Es ist die Frage zu beantworten, unter welchen Bedingungen ein Microservice-System, welches mittels Saga-Pattern implementiert wurde, eine LLT abbilden kann.

Beantwortung

Damit eine LLT als Saga implementiert werden kann, muss die Transaktion in Teiloperationen aufteilbar sein. Das System, welche diese Teiloperationen ausführt, besteht aus verschiedenen Teilnehmerservices, die die Teiloperation ausführen, und einem Koordinator, der die LLT überwacht und den Prozessablauf steuert. Für auftretende Fehler kann der Koordinator eine Kompensierung aufrufen.

6.3.2 Leitfrage 2

Es sind Fehlerquellen und Fehlertypen zu identifizieren, die einen inkonsistenten Systemzustand verursachen können. Es sollen Lösungen im Rahmen des Saga-Patterns formuliert, implementiert und evaluiert werden.

Beantwortung
Die im Versuch aufgetretenen Fehlerquellen wurden als Isolationsanomalien und Netzwerkfehler identifiziert.
Die durch Verletzung der Isolationseigenschaft verursachten Fehler wurden auf Ebene der Teilnehmerservices mittels Anpassung der Isolationsstufe und auf Ebene der LLT mittels Einschränkung des Systems verhindert.
Die durch das Auftreten von Netzwerkpartitionen verursachten Fehler konnten durch die Verwendung idempotenter Schnittstellen und Recoverymechanismen behoben werden.

6.3.3 Leitfrage 3

Es soll eine Antwort darauf gefunden werden, welche Kriterien eine Schnittstelle erfüllen muss, um an einer LLT teilnehmen zu können.

Beantwortung
Damit ein Service zum Saga-Transaktionsteilnehmer werden kann, müssen die in der Schnittstelle stattfindenden Änderungen innerhalb einer ACID-Transaktion stattfinden. Dies verhindert die durch fehlende Isolation verursachten Anomalien. Außerdem besteht die Voraussetzung, dass Requests höchstens einmal erfolgreich verarbeitet werden. Dies kann realisiert werden, indem im Request eine Id erwartet wird, um die versehentliche Mehrfachverarbeitung auszuschließen.

6.3.4 Leitfrage 4

Es soll beantwortet werden, ob aufgrund Netzwerkpartitionen auftretende Fehler in die Fehlerbehandlung des Saga-Patterns integrierbar sind.

Beantwortung
Im Versuch wurden Netzwerkfehler in das Saga-Pattern integriert. Es hat sich gezeigt, dass unter Verwendung idempotenter Schnittstellen eine Retry-Logik implementiert werden kann, ohne ein inkonsistenten Systemzustand zu erreichen. Es hat sich außerdem gezeigt, dass im Falle eines Netzwerkfehlers nicht mit der Verarbeitung der LLT fortgefahren werden kann, da der Verarbeitungszustand des fehlgeschlagenen Schrittes unklar ist. Wird fortgefahren bevor ein konkreter Verarbeitungszustand klar ist, treten unweigerlich Inkonsistenzen auf.

6.4 Beantwortung der These

Zu Beginn dieser Arbeit wurde folgende These gestellt:

These. *Mittels Saga-Pattern implementierte langlebige Transaktionen (LLT) ermöglichen ausfallsichere Konsistenz in Microservicesystemen, die per Request-Response Pattern kommunizieren.*

Es wurde ein System entworfen, welches eine LLT abbildet. Dabei wurden alle Teiloperationen im Rahmen einer Teiltransaktion innerhalb des entsprechenden Teilnehmerservices durchführen kann. Nach Vorbild des Saga-Patterns wurden alle Schnittstellen mit jeweils einer Kompensierung ergänzt.
Der Koordinator war in der Lage den grundlegenden Ablauf der LLT ohne Auftreten von Netzwerkfehlern zu steuern. Unter Auftreten von Netzwerkfehlern führte die anfängliche Implementierung zu inkonsistenten Systemzuständen. Die Messung der Ergebnisse der ursprünglichen Implementierung zeigte vorzeitige Abbrüche und Inkonsistenzen auf.
Diese Lösung wurde iterativ erweitert. Es wurde der Zusammenhang des CAP-Theorems deutlich, dass eine verteilte Saga die Entscheidung treffen muss, ob es den Fokus auf Verfügbarkeit oder Konsistenz legt. Dies äußerte sich in der Implementierung darin, dass Netzwerkfehler sich nicht in das Fehlermanagement des Saga-Patterns integrieren ließen. Die einzige Implementierung, die die Konsistenz innerhalb des Systems wahrte, beinhaltete die Verwendung von idempotenten Schnittstellen in Kombination mit Retrymechanismen.
Es wurde demnach gezeigt, dass der Koordinator zwischen Netzwerkfehlern und Fehlern der lokalen Transaktionen unterscheiden muss. Die Fehlerbehandlung der Ergebnisse der lokalen Transaktionen führt zu einer Behandlung im Rahmen des Saga-Patterns (Forward- oder Backward-Recovery) während die Fehlerbehandlung für Netzwerkfehler den Prozess zum Pausieren zwingt, bis die Netzwerkpartition aufgelöst ist (Retries).

6.5 Realisierbarkeit von Sagas in eng gekoppelten Systemen

Die im Versuch auftretenden Isolationsanomalien konnten sehr einfach verhindert werden, indem die Grenzen der Teilnehmerservices entsprechend ihrer Aufgabe innerhalb der fachlichen Domäne gewählt wurden. Dadurch existierten sehr wenige Stellen, an denen ein Service auf Ressourcen zugreifen muss, die auch für andere Transaktionen relevant sind.
In einem realen Produktivsystem können die Verantwortlichkeiten der Services weniger deutlich geschnitten sein oder sehr eng an andere geschäftsrelevante Prozesse gebunden sein. In diesen Systemen sind die durch die Verwendung des Saga-Patterns auftretenden Isolationsanomalien schwieriger zu identifizieren und zu verhindern.
Die Lösung des Problems der Isolationsanomalien auf Ebene der LLT waren in diesem Versuch systemische Einschränkungen der Ressourcen. Beispielsweise war der BankingService offen für Isolationsanomalien, falls während einer laufenden LLT ein Kunde sein Konto schließt. Die zusätzliche Prüfung beim Schließen des Kontos auf laufende LLTs stellt eine solche systemische Einschränkung dar. Diese Lösung koppelt die Transaktionsteilnehmer jedoch sehr eng aneinander. Somit stellt diese Lösung in einem wachsenden System eine Gefahr für versteckte Fehler dar.

6.6 Bedeutung des Saga-Patterns für Entwickler

In verteilten Systemen sind Partitionen unvermeidbar und müssen somit bei der Planung von LLT beachtet werden. Das Saga-Pattern wurde als Alternative für den 2PC vorgestellt. Beide Muster sind wertvolle Werkzeuge für die Entwicklung verteilter Transaktionen. Der Entwickler muss dabei die Wichtigkeit von Konsistenz und Verfügbarkeit abwägen.
Ist Konsistenz das wichtigste Kriterium des Systems, so ist ein 2PC (oder davon ableitende Commitprotokolle) die beste Wahl.
Wenn das System mit Inkonsistenzen umgehen kann, dürfen Atomarität und Isolation von der LLT aufgelöst werden und per Saga-Pattern implementiert werden. Der Versuch hat gezeigt, dass eine Saga sowohl als *AP-System* mit hoher Verfügbarkeit sowie als *CP-System* mit hoher Konsistenz implementiert werden kann. Diese Tatsache bietet dem Entwickler eine Flexibilität während des Entwicklungsprozesses. Der Aufwand für das Abändern eines Saga-Systems von AP auf CP ist wesentlich geringer als der Aufwand für den Umbau eines per 2PC implementierten Systems auf Saga. Somit stellt dieses Muster eine komfortable Implementierung zur Verfügung, bei der die Bindung der Teilnehmer mit dem Koordinator möglichst gering gehalten werden kann bei vergleichsweise hoher Konsistenz.
In LLTs mit besonders geringer Abhängigkeit der einzelnen Schritte eignet sich das Saga-Pattern besonders gut. Es gibt Anwendungsfälle, in denen eine Reihe von Transaktionsteilnehmern integriert werden sollen und lediglich sichergestellt werden muss, dass der Aufruf atomar stattfindet. Aufgrund der Unabhängigkeit der Teilsysteme ist es außerdem sehr leicht, die fehlende Isolation zu vernachlässigen. In solchen Anwendungsfällen stellt das Saga-Pattern eine simple Möglichkeit der Integration dar.

6.7 Ausblick und weitere Forschung

Die

- Erkennung von Isolationsanomalien
- Recovery-Mechanismen

Abkürzungen

2PC Zwei-Phasen-Commit.

ACID Atomicity, Consistency, Isolation, Durability.

BASE Basically Available, Soft State, Eventual Consistency.

C Kompensierung.

DB Datenbank.

DBMS Datenbank-Management-System.

DEA Deterministischer endlicher Automat.

LLT Long Lived Transaction.

SEC Saga Execution Component.

TEC Transaction Execution Component.

Abbildungsverzeichnis

Literatur

[1] H. Garcia-Molina und K. Salem, "Sagas", *ACM SIGMOD Record*, 1987, ISSN: 0163-5808. DOI: 10.1145/38714.38742.

[2] DatabaseManagementSystems, *Database management systems*. 2003. Adresse: https://scholar.google.com/citations?user=udzsrkyaaaaj&hl=de&oi=sra.

[3] M. Jarke und J. Koch, "Query Optimization in Database Systems", *ACM Comput. Surv.*, Jg. 16, Nr. 2, S. 111–152, 1984, ISSN: 0360-0300. DOI: 10.1145/356924.356928.

[4] A. S. Dey, "Cherry Garcia: Transactions across Heterogeneous Data Stores: Cherry Garcia: Transactions across Heterogeneous Data Stores", Diss. Adresse: https://ses.library.usyd.edu.au/handle/2123/14212.

[5] Sadalage, Pramod J. and Fowler, Martin, "NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence", 2012.

[6] E. Brewer, *Towards robust distributed systems*, New York, NY, USA. Adresse: https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/brewer_podc_keynote_2000.pdf.

[7] S. Gilbert und N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services", *ACM SIGACT News*, Jg. 33, 2002. DOI: 10.1145/564585.564601.

[8] IBM Documentation, *IBM Documentation*, 2021. Adresse: https://www.ibm.com/docs/ko/iis/8.5?topic=levels-data-anomalies.

[9] PostgreSQL Documentation, *PostgreSQL*, 2023. Adresse: https://www.postgresql.org/docs/current/transaction-iso.html.

[10] A. Zelený, *Transaction_Isolation_Levels_and_anomalies_p2d2_2022: P2D2 2022*, Prague, Czech Republic, 2022-06-01. Adresse: https://p2d2.cz/files/isolation-levels-p2d2-zeleny.pdf.

[11] A. Gaitonde, "Distributed Transactions & Two-phase Commit - Geek Culture - Medium", *Geek Culture*, 24.03.2021. Adresse: https://medium.com/geekculture/distributed-transactions-two-phase-commit-c82752d69324.

[12] microservices.io, *Microservices Pattern: Database per service*, 12.01.2024. Adresse: https://microservices.io/patterns/data/database-per-service.html.