Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Michael Müller

Steigerung der Fehlertoleranz bei Ausfall der Simulationsausführung in der MARS Microservice-Umgebung

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Michael Müller

Steigerung der Fehlertoleranz bei Ausfall der Simulationsausführung in der MARS Microservice-Umgebung

Bachelorarbeit eingereicht im Rahmen der Bachelor Thesis

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas Clemen

Eingereicht am: 15. Januar 2019

Michael Müller

Thema der Arbeit
Steigerung der Fehlertoleranz bei Ausfall der Simulationsausführung in der MARS Microservice-Umgebung

Stichworte
Microservices, Multi Agenten Simulation, Crash, Recovery, Verteilte Systeme

Kurzzusammenfassung
Um die Fehlertoleranz zu steigern, werden in dieser Arbeit Ausfälle der MARS Simulationsausführung analysiert und behandelt. Die MARS Forschungsgruppe der HAW Hamburg untersucht Multi Agenten Simulationen und verwendet ein verteiltes System mit Microservices. Auf Basis von Heartbeats zur Ausfallerkennung und Checkpoints für die Zustandswiederherstellung wird ein Microservice erstellt, der die Simulationsausführungen überwacht und ggf. neu startet.

Title of the paper
Increase of fault tolerance during crash of simulation execution in the MARS microservice environment

Keywords
Microservices, Multi Agent Simulation, Crash, Recovery, Distributed Systems

Abstract
To increase the fault tolerance of MARS simulation executions, error and crashes need be analyzed and handled. The HAW Hamburg MARS research group uses a distributed system with microservices. A microservice will be designed to detect crashes with heartbeats and recover from simulation execution crashes with checkpoints.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

In den letzten Jahren gewannen Microservice Architekturen, aufgrund ihrer erwarteten besseren Wartbarkeit, der gezielteren Skalierung und der erhöhten Ausfallsicherheit, an Popularität (Newman (2015), Düllmann und van Hoorn (2017), Gil und Díaz-Heredero (2018), Li u. a. (2018), Hasselbring (2016), Fowler (2014)). Wie die Ausfallsicherheit für MARS Simulationsausführungen in einem Kubernetes Cluster erhöht werden kann, behandelt diese Arbeit.

1.1 Motivation

Ein Ausfall von zustandsbehafteten Microservices, wie zum Beispiel einer MARS Simulationsausführung, ist in Kubernetes ein Problem, da Kubernetes nicht genau erkennt, warum ein Microservice ausgefallen ist [Kubernetes (2018d)] und dessen Zustand nur indirekt wiederherstellen kann [Kubernetes (2018h)].

In Grošelj (1991) wurde unter Verwendung von Checkpoints gezeigt, wie eine verteilte Multi Agenten Simulation fehlertoleranter gestaltet werden kann. In verteilten Systemen ist die Verwendung von Checkpoints zur Zustandswiederherstellung (Haerder und Reuter (1983), Lin und Dunham (1997)) und die Ausfallerkennung anhand von Heartbeats (Burrows (2006), Ghemawat u. a. (2003), Isard u. a. (2007)) schon intensiv studiert worden. Die Kombination aus Heartbeats und Checkpoints wird allerdings noch nicht in der MARS Kubernetes Umgebung eingesetzt.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist ein Entwurf und eine prototypische Implementation. Der Entwurf beschreibt wie die Ausfallsicherheit für zustandsbehaftete Simulationsausführungen im MARS Kubernetes Cluster mithilfe von Checkpoints und Heartbeats erhöht werden kann, sodass bei einem Ausfall kein Simulationsfortschritt verloren geht. Um im Rahmen dieser Bachelorarbeit zu bleiben, umschließt die prototypische Implementierung neben der kompletten Ausfallerkennung, vereinfachte Checkpoints und keine Zustandswiederherstellung.

Anhand verschiedener Experimente wird überprüft, ob die Ausfälle richtig erkannt und behandelt werden.

1.3 Struktur

Die Bachelorarbeit ist nach dieser Einleitung wie folgt gegliedert: In Kapitel 2, werden verwandte Arbeiten behandelt, um ähnliche Problematiken zu erläutern und benötigte Lösungsansätze vorzustellen. Im 3. Kapitel geht es um die Klärung weiterer Begrifflichkeiten und Einführungen in das Themengebiet um die Grundlage für die darauffolgenden Kapitel zu legen. Eine tief gehende Analyse des Problems wird in Kapitel 4 erstellt, darauf aufbauend stellt Kapitel 5 den Microservice vor, der das Problem lösen soll. Kapitel 6 wird auf Details der Implementierung eingehen und im 7. Kapitel wird die Forschungsfrage beantwortet. Das letzte Kapitel schließt die Bachelorarbeit mit einer Zusammenfassung ab und gibt einen Ausblick zukünftige Arbeiten.

2 Verwandte Arbeiten

In diesem Kapitel wird auf verwandte Arbeiten eingegangen und die eigene Arbeit eingeordnet.

2.1 Fehlererkennung

Die folgenden Alternativen beschreiben, wie ein Ausfall eines Prozesses (Fehler) erkannt werden kann (Fehlererkennung).

2.1.1 Hardware Aktivität

Tan u. a. (2012) beschreibt eine Fehlererkennung, die den zu überwachenden Prozess in zwei Phasen aufteilt, Berechnungsphase und Kommunikationsphase. Während in der Berechnungsphase v.a. die CPU-Last steigt, steigt in der Kommunikationsphase die Last für die Festplatten, das Netzwerkes oder den CPU Kernel Speicher. Anhand dieser Lasten wird erkannt, ob der Prozess noch aktiv ist und in welcher Phase er sich befindet.

2.1.2 Heartbeats

Heartbeats werden zur Fehlererkennung genutzt, dabei wird regelmäßig eine Nachricht von dem zu überwachenden Prozess an den Fehlerdetektor gesendet [Aguilera u. a. (1997)]. Die Heartbeats werden in Kombination mit Timeouts verwendet, damit ein Ausfall erkannt wird, wenn z. B. mehrere Heartbeats fehlen.

Diese Fehlererkennung wird zum Beispiel in Burrows (2006), Ghemawat u. a. (2003) und Isard u. a. (2007) verwendet. Alle drei Systeme sind für “coarse-grained” Systeme, also für parallele Systeme mit großen Prozessen gedacht. Prozesse werden als Groß eingestuft, wenn sie länger als 20 Sekunden für Ihre Berechnung brauchen. Eine MARS Simulation dauert in der Regel Minuten, Stunden oder Tage, weshalb diese zu den größeren Prozessen gehören und somit auch “coarse-grained” sind.

2.1.3 Fehlerdetektoren ohne Timeouts

Die meisten Fehlerdetektoren arbeiten mit Timeouts, um Ausfälle zu erkennen [Cachin u. a. (2011)], jedoch gibt es gute Gründe dies nicht zu tun. Zwar nennen Aguilera u. a. (1997) und Leners u. a. (2011) Fehlerdetektoren die Timeouts nutzen, um zu bestimmen, ob ein Prozess ausgefallen ist, verwenden sie selbst allerdings nicht. Leners u. a. (2011) geht sehr stark darauf ein, dass ein Timeout nicht richtig gesetzt werden kann. Entweder der Timeout ist zu groß und es entsteht Wartezeit, bis der Ausfall erkannt wird, oder der Timeout ist zu klein und es werden "false positive" Ausfälle erkannt.

Das FALCON System von Leners u. a. (2011) benutzt keine Timeouts für die Ausfallerkennung, ist aber dafür wesentlich komplizierter. Das Falcon System bedingt, dass auf jedem Rechner ein kleines Überwachungsprogramm vorhanden ist das als "Spy" bezeichnet wird. Mithilfe dieser kleinen Überwachungsprogramme kann der Status des zu beobachtenden Prozesses auf verschiedenen Ebenen abgefragt werden.

2.2 Wiederherstellung

Bei zustandsbehafteten Prozessen, muss nach Ausfall, der Zustand vor dem Ausfall wiederhergestellt werden. Wie eine solche Wiederherstellung umgesetzt werden kann, behandeln die folgenden Arbeiten.

2.2.1 Protokollierung

Bei der Protokollierung wird vorausgesetzt, dass das Protokollierte erneut durchgeführt werden kann, dies sind zum Beispiel Transaktionen. Sobald eine Transaktion empfangen und abgeschlossen wurde, wird sie protokolliert. Fällt der protokollierte Prozess aus, so werden nach Neustart alle Transaktionen in derselben Reihenfolge erneut durchgeführt. Sind alle Transaktionen abgearbeitet, wurde der Zustand vor dem Ausfall wiederhergestellt (Tanenbaum und Steen (2008), Lin und Dunham (1997)).

2.2.2 Checkpoints

Checkpoints sind persistierte Zustände eines Programms oder Daten zu einem bestimmten Zeitpunkt. Nach Haerder und Reuter (1983) gibt es mehrere Checkpointarten, die darin unterscheiden wie fein die Elemente der Checkpoints sind und wann ein Checkpoint eines Elements erstellt wird.

Eine Checkpointart ist der “Transaction Consistent Checkpoint (TCC)”. Die Definition besagt, dass zum Zeitpunkt der Erstellung keine Transaktion bearbeitet werden darf. Jede begonnene Transaktion wird abgeschlossen und alle neuen müssen warten, bis der Checkpoint erstellt wurde. Eine Alternative Checkpointart sind “Fuzzy Checkpoints”, diese sind darauf ausgelegt schnell einen Checkpoint zu erstellen, ohne Transaktionen aufzuhalten. Die Checkpoints werden Fuzzy genannt, weil sie in der Regel keine Konsistenz garantieren, da mehrere Objekte zu unterschiedlichen Zeitpunkten gespeichert wurden. Der TCC ist zur Wiederherstellung besser geeignet, da er einen kompletten konsistenten Zustand hält. Ein Fuzzy Checkpoint dagegen braucht für die Wiederherstellung weitaus länger, denn hier werden gegebenenfalls noch weitere Protokoll Daten gebraucht um aus dem inkonsistenten Zustand in einen konsistenten zu gelangen.

Bei der Wiederherstellung verwenden Checkpoints in der Regel zusätzlich ein Protokoll der durchgeführten Transaktionen die zwischen der Checkpoint Erstellung und des Ausfalls bearbeitet wurden. In Kombination mit dem Protokoll gehen weniger bereits bearbeitete Transaktionen verloren (Haerder und Reuter (1983), Lin und Dunham (1997)).

2.3 Ausfallsichere Simulationen

In Grošelj (1991) wird eine ausfallsichere verteilte Simulation vorgestellt. Wie in den MARS Simulationen, ist eine Simulation in Zeitschritte eingeteilt. Der Autor erkennt Deadlocks und Prozessorausfälle als Probleme für konservative Algorithmen von verteilten Simulationen.

Beide Probleme werden durch das Erstellen von globalen Snapshots (Checkpoints) gelöst. Die globalen Checkpoints bestehen aus der Summe der lokalen Checkpoints und werden alle nach einem Zeitschritt erstellt. Auf Basis des aktuellsten globalen Checkpoints wird analysiert, ob ein Deadlock vorliegt. Die lokalen Checkpoints werden dazu genutzt, um die ausgefallenen Prozesse wieder in den richtigen Zustand zu versetzen. Einen Ausfall muss der Prozessor selbst erkennen und sich daraufhin wiederherstellen.

Der Autor erkennt nach den Benchmarks, dass die sichere Simulation (mit den Checkpoints) nur 10% langsamer ist als die unsichere Simulation (ohne Checkpoints). Zudem konnten alle Deadlocks vermieden werden.

Der größte Unterschied zu den MARS Simulationen findet sich darin, dass die in dem Paper vorgestellten Simulationen selbst einen Ausfall erkennen müssen. MARS Simulationsausführungen können dies nicht, weshalb die Wiederherstellungs- und Deadlock Mechanismen nicht übertragen werden können.

2.4 Einordnung

Diese Arbeit ordnet sich in den Bereich Fehlertoleranz, mit Bezug zu Simulationen in einem Kubernetes Microservice Cluster, ein. Es werden die Fehlererkennung durch Heartbeats und die Wiederherstellung durch Checkpoints mit einbezogen.

Nicht Teil dieser Arbeit ist dagegen die Wiederherstellung durch Protokollierung der Aktionen innerhalb einer Simulation oder die Ausfallsicherheit von verteilten Simulationen.

3 Grundlagen

In diesem Kapitel werden die Grundlagen für die Bachelorarbeit erläutert.

3.1 Ausfälle in verteilten Systemen

Einen Ausfall bezeichnen Tanenbaum und Steen (2008) als das nicht Erfüllen von Zusagen. Ein Fehler ist Teil des Systemzustands, der zu einem Ausfall führen kann.

Ein Fehler (auch Störung genannt) kann vorübergehend, wiederkehrend oder permanent sein. Eine vorübergehende Störung wird gelöst, indem der Vorgang wiederholt wird. Die vorübergehenden Fehler treten einmalig auf. Eine wiederkehrende Störung allerdings wird öfter erscheinen und verschwinden, zum Beispiel ein Wackelkontakt in einem elektronischen Schaltkreis. Wiederkehrende Störungen sind schwer zu diagnostizieren [Tanenbaum und Steen (2008)]. Die permanenten Störungen gleichen Totalausfällen, diese können nur behoben werden, indem die betroffenen Teile auswechselt werden. Zum Beispiel durch das Auswechseln beschädigter Hardware.

Für Prozessausfälle gibt es unter anderem zwei Abstraktionen, “crash-stop” und “crash-recover”. Bei “crash-stop” wird davon ausgegangen, dass ein ausgefallener Prozess frühestens dann wiederhergestellt ist, wenn das Ziel durch die restlichen Prozesse schon erreicht wurde, zum Beispiel in einem verteilten Algorithmus. Mit eingeschlossen ist auch keine oder eine nicht erfolgreiche Wiederherstellung. Dagegen ist bei der “crash-recover” Abstraktion vorgesehen, dass ein Prozess ausfällt und daraufhin sofort wiederhergestellt wird [Cachin u. a. (2011)].

3.1.1 Fehlermodelle

Fehler und Ausfälle wurden von Tanenbaum und Steen (2008), Avizienis u. a. (2004) und Cachin u. a. (2011) wie folgt klassifiziert:

Crash Failure Ein Server arbeitete bis zu seinem Ausfall (Crash) richtig. Durch diesen Crash stürzt der Server zum Beispiel ab, die Störung ist permanent. Erst ein Neustart kann diese Störung beheben.

Omission Failure Der Server antwortet nicht mehr auf eingehende Nachrichten oder sendet selbst keine Nachrichten. Ein möglicher Grund hierfür kann zum Beispiel eine Endlosschleife sein, in der sich der Server gerade befindet. Eine solche Störung ist nicht auf einen Störungstypen beschränkt.

Timing Failure Die Antwortzeit eines Servers liegt außerhalb des festgelegten Zeitintervalls. Eine Lösung ist etwa die Entlastung des Servers durch horizontale oder vertikale Skalierung um eine schnellere Antwortzeit zu schaffen. Ein Timing Failure ist wiederkehrend, bis die Last für den Server und Netzwerk Komponenten gesunken sind oder die Clients mit längeren Antwortzeiten rechnen.

Response Failure Die Antwort eines Servers ist falsch formatiert oder der Wert ist inkorrekt. Die Lösung ist meistens das Debugging des Server Codes, da entweder die Anforderung falsch verstanden wurde, oder sie wurde richtig erkannt, aber es wurde die falsche Antwort gesendet. Bis zur Behebung des Problems ist diese Störung permanent oder wiederkehrend.

Byzantine Failure Byzantinische Fehler sind die kompliziertesten Fehler in Bezug auf die Lösung, da deren Fehlerquelle, -ursache und -art stark variieren. Bei einem solchen Fehler erstellt ein Server zum Beispiel zufällige Antworten zu zufälligen Zeiten oder zufälligen Inhalt, diese Fehlerart ist nicht beschränkt auf ein Störungstyp oder einen Lösungsweg.

Eavesdropping Faults Diese permanente Fehlerart konzentriert sich mehr auf die IT Sicherheit als die Arten zuvor, denn die Geheimhaltung von Nachrichten kann gefährdet sein [1]. In einer unsicheren Umgebung können Angreifer die gesendeten Daten der verteilten Prozesse abhören und lesen. Eine Lösung hierfür ist unter anderem die Verschlüsselung der Nachrichten.

3.1.2 Fehlertoleranz

Tanenbaum und Steen (2008) beschreiben Fehlertoleranz als das Tolerieren von Ausfallfehlern, dies enthält das Erkennen von Fehlern und die adäquate Reaktion darauf. Avizienis u. a. (2004) geht darüber hinaus und bezeichnet die Fehlertoleranz als die Vertrauensbasis gegenüber einem Programm.

Selbst wenn Komponenten oder extern Systeme nicht mehr korrekt arbeiten oder ausfallen, muss das Programm den eigenen Dienst weiter korrekt fortführen. Die Fehlertoleranz

[1] mehr dazu siehe Andress (2014)

hängt sehr eng mit verlässlichen Systemen zusammen, dessen Eigenschaften sind nachfolgend genannt und beschrieben:

Verfügbarkeit Diese Eigenschaft gibt an, wie wahrscheinlich es ist, dass das System korrekt funktioniert und ein Benutzer es nutzen kann. Bei einem höchst verfügbaren System ist demnach die Chance sehr hoch, dass es sich zu jedem Zeitpunkt wie erwartet verhält.

Zuverlässigkeit Die Zuverlässigkeit ähnelt der Verfügbarkeit sehr stark. Der Unterschied, der beiden ist, dass die Verfügbarkeit auf Zeitpunkte und die Zuverlässigkeit auf längere Zeitspannen konzentriert ist. Ein höchst zuverlässiges System ist eines, dass längere Zeit ohne Unterbrechung aktiv ist.

Funktionssicherheit Die Funktionssicherheit gibt an, dass trotz Systemausfall, noch gewisse Aufgaben erfüllt werden müssen. So führt Tanenbaum und Steen (2008) das Beispiel von Leitsystemen in Kernkraftwerken an. Wenn diese ausfallen, darf dies zu keiner Katastrophe führen.

Wartbarkeit Die letzte Eigenschaft bezieht sich auf die Behebung von Bugs oder die Reparatur ausgefallener Systeme, dies beinhaltet auch das Wiederherstellen von Zuständen nach einem Ausfall. Geschieht diese Wiederherstellung automatisch, so hat dies einen deutlich positiven und direkten Effekt auf die Verfügbarkeit und Zuverlässigkeit.

3.2 Microservices

Microservices sind autonome Services die zusammen arbeiten [Newman (2015)]. Sie haben eine hohe Kohäsion und sind gegenüber anderen Microservices gering gekoppelt [Mazzara und Meyer (2017)]. Die Microservices kommunizieren in der Regel über das Netzwerk, da sie in der Regel auf unterschiedlichen Computern ausgeführt werden. Microservices werden auch als Verfeinerung der “Service Orientierten Architektur” angesehen und sind auch ein Architektur Pattern (Fowler (2014), Heinrich u. a. (2017)). Ein monolithisches Programm kann in kleinere Microservices aufgespalten werden, dies erhöht die Performanz des Gesamtsystems. In der Performanz sieht Newman (2015) das Hauptziel von Microservices.

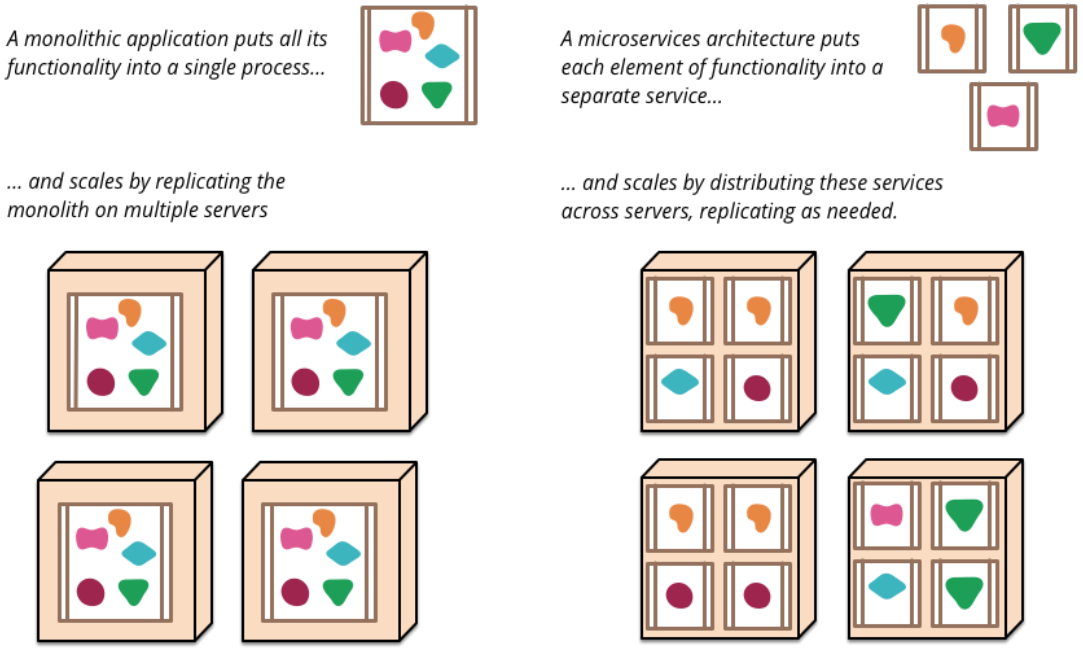

Abbildung 3.1: Beispielhafte Aufteilung eines Monolithen in Microservices nach Fowler (2014)

Die Autonomie wird weiterhin mit der Nutzung von Containern erhöht. Container sind leichtgewichtige Virtualisierungen [Augsten (2017)] und es gibt mehrere Anbieter dieser Technologie. Im MARS Umfeld werden Microservices in Docker Containern ausgeführt. Durch die erhöhte Autonomie lässt sich die Performanz von Microservices steigern, in dem für jeden Microservice die Technologien verwendet werden die für den Use Case am besten geeignet sind.

Der Ausfall eines Containers betrifft andere Container nicht, dagegen wird bei einem Monolithen das gesamte Programm ausfallen [Hasselbring (2016)]. Die Kommunikation über das Netzwerk zwischen den Microservices fördert zwar die Autonomie der einzelnen Microservices, führt jedoch gleichzeitig auch zu erhöhter Komplexität, was sich vor allem während des Debuggings äußert [Newman (2015)].

Microservices können in zustandsbehaftete und zustandslose Microservices unterschieden werden. Der Unterschied von zustandsbehafteten Microservices zu zustandslosen Microservices ist der, dass zustandslose Microservices bei selben Input immer denselben Output erzeugen.

Bei einem zustandsbehafteten Microservices hängt der Output vom dem Zustand ab, in dem der Microservices sich aktuell befindet [Sandoval (2017)].

Bei der Wiederherstellung, nach einem Ausfall, muss diese Eigenschaft eines Microservices berücksichtigt werden. Während ein zustandsloser Microservice ohne weiteres Neu gestartet werden kann, muss bei einem zustandsbehafteten Microservice der Zustand vor dem Ausfall wiederhergestellt werden. Wie in 2.2 erwähnt, ist dies zum Beispiel mit Checkpoints möglich.

3.3 Kubernetes

Luksa (2018) beschreibt Kubernetes als “Cluster Management Tool” und Hilfsmittel, das einzelnen Ressourcen (Speicher, Computer) in ein Cluster zusammenfasst. Kubernetes übernimmt die Verwaltung von ausgeführten Programmen und lastet alle Computer (Nodes) gleichmäßig aus.

Kubernetes besteht aus Pods und Nodes. Ein Node ist zum Beispiel ein Computer auf dem Pods aktiv sein können. Innerhalb der Pods können wiederum Container gestartet werden. Kubernetes Services laufen in einem oder mehreren Kubernetes Pods, diese Pods sind die kleinste Einheit die zur Lastverteilung herangezogen werden kann [Kubernetes (2018e)]. Die Entitäten sind in Abbildung 3.2 wiederzufinden.

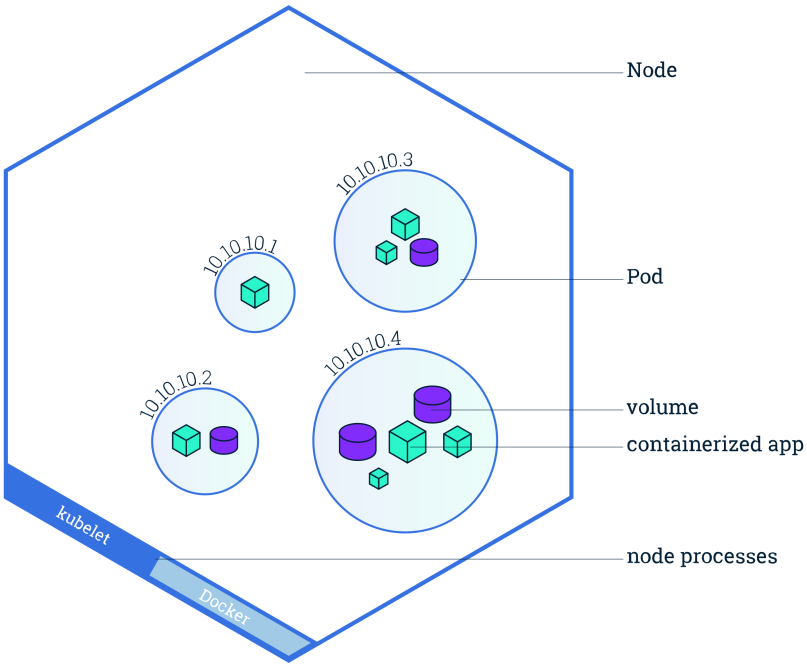

Abbildung 3.2: Kubernetes Komponenten nach Kubernetes (2018i)

3.4 MARS

Das “Multi-Agent Research and Simulation system” (MARS) ist ein Simulationsframework. Es handelt sich dabei zudem um ein “Modeling and Simulation as a Service” (MSaaS) System, dass in der HAW Hamburg entwickelt, stetig erweitert und von der MARS Forschungsgruppe gepflegt wird [Hüning u. a. (2016)]. Die Multi Agenten Simulationen werden in einem Kubernetes Cluster der HAW Hamburg mit Microservices ausgeführt.

Die Simulationen mit teils mehreren Millionen Agenten sind in sogenannte “Ticks” eingeteilt. Jeder Tick hat eine definierte Zeitlänge. Die Simulation beginnt bei Tick 0 und arbeitet alle Ticks ab, diese Simulationen benötigen unter Umständen sehr viel Rechenleistung, weshalb es ratsam ist größere Simulationen nicht lokal zu simulieren, sondern in einem verteilten Cluster rechnen zu lassen.

Der Abhängigkeitsgraph in Abbildung 3.3 zeigt einen Ausschnitt der an einer Simulation beteiligten Ressourcen. Hierbei ist zu erkennen, dass die Simulationsausführungen abhängig von einem “Simulation Plan” sind. In den “Simulation Plans” werden unter anderem die vorhin

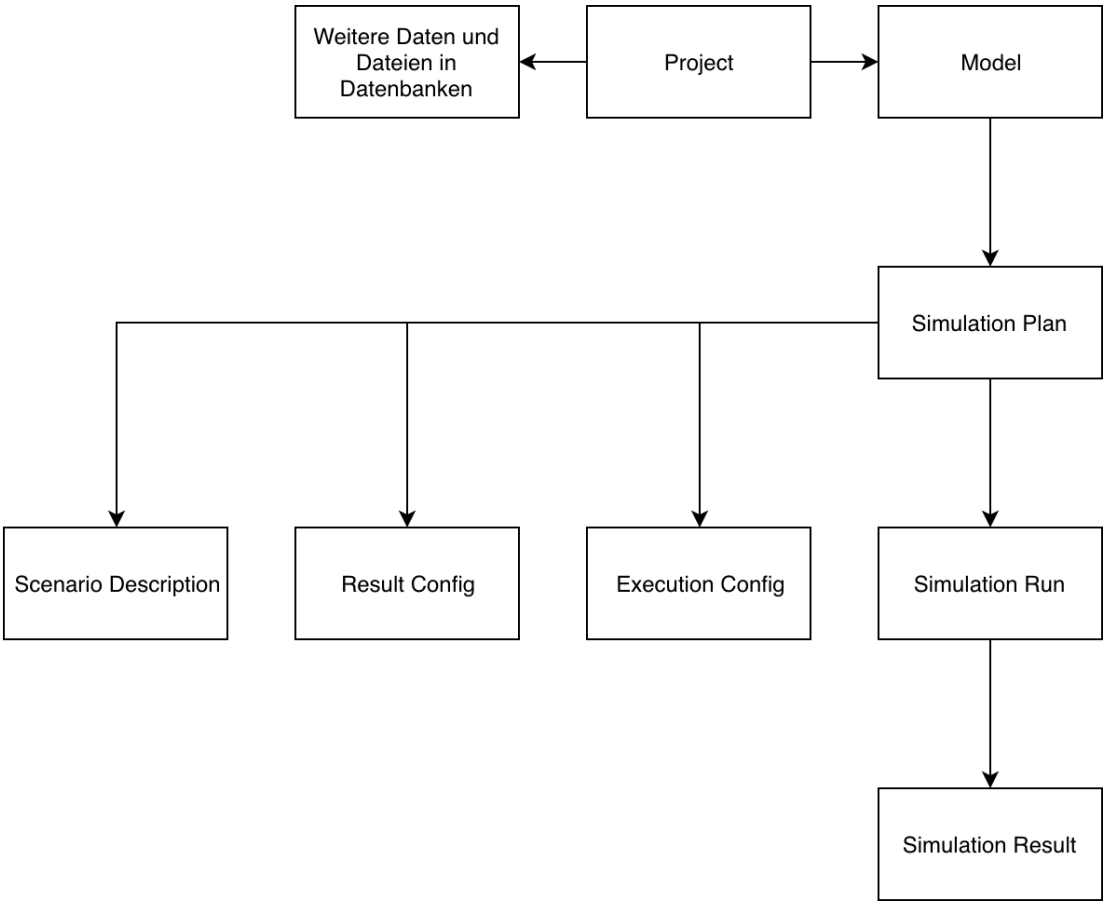

Abbildung 3.3: Abhängigkeiten von MARS Ressourcen

erwähnten Tick Zeitlängen und weitere Konfigurationen definiert. Die Ausführung eines "Simulation Plans" erzeugt Simulationsausführungen, auch "Simulation Runs" genannt.

4 Problem- und Anforderungsanalyse

In diesem Kapitel werden alle Problemfälle beschrieben und darauf aufbauend die Anforderungen für das neue System erstellt.

4.1 Ausfälle

4.1.1 Kubernetes Pod Ausfallerkennung

Kubernetes erkennt den Ausfall eines Docker Containers anhand seines Exit Codes oder durch Abfragen einer Rest Schnittstelle [Kubernetes (2018a)]. Ist der Exit Code eines Containers nicht 0 wird der Pod als Ausfall deklariert [Kubernetes (2018d)]. Eine native Alternative besteht darin, eine REST Schnittstelle für Kubernetes anzubieten. Sobald diese Schnittstelle einen HTTP Code zurückgibt der nicht mehr zwischen 200 und 300 liegt, oder gar keine Antwort empfangen wird, erkennt Kubernetes den Container als ausgefallen.

Beide Alternativen decken nicht den Fall ab, dass der Container der Simulationsausführung durch einen Laufzeitfehler beendet wurde. In diesem Fall ist der Exit Code nicht 0, da der Container aufgrund eines Fehlers terminierte und die REST Schnittstelle kann nicht mehr antworten, weil der Container schon terminiert ist. Ein Laufzeitfehler kann also in einem Kubernetes Cluster dazuführen, dass ein Container stetig terminiert und neu gestartet wird. Der Neustart durch Kubernetes lässt sich deaktivieren, dies muss allerdings explizit in den Kubernetes Deployments definiert werden.

4.1.2 Ausfall einer Simulationsausführung

Eine Simulationsausführung kann zu jeder Zeit ausfallen. Es wird in folgende Zustände unterschieden:

Zustand 1 Ausfall der Simulation vor Anmeldung

Zustand 2 Ausfall der Simulation nach Anmeldung

Zustand 3 Ausfall während der Berechnungsphase und dem sequentiellen berechnen der Ticks

Zustand 4 Ausfall der Simulation vor Abmeldung

Zustand 5 Ausfall der Simulation nach Abmeldung

Fällt die Simulationsausführung in Zustand 1 aus, wird der Snapshot Manager dies nicht bemerken, da die Simulationsausführung noch nicht angemeldet war. Ein Neustart muss von Hand eingeleitet werden. Ähnlich verhält es sich in den Zuständen 4 & 5, dann muss allerdings der zuständige Entwickler selbst prüfen, ob ein Neustart wirklich relevant ist, da unter Umständen alle Ticks bereits berechnet wurden.

Im 2. und 3. Zustand muss die Simulationsausführung vom Snapshot Manager neu gestartet werden. Es wird erwartet, dass im 3. Zustand die meisten Ausfälle passieren, da diese Phase am längsten anhält. In dieser Phase werden Snapshots erstellt und bei Ausfall von dem letzten Snapshot gemäß Kapitel 4.2.4 fortgefahren.

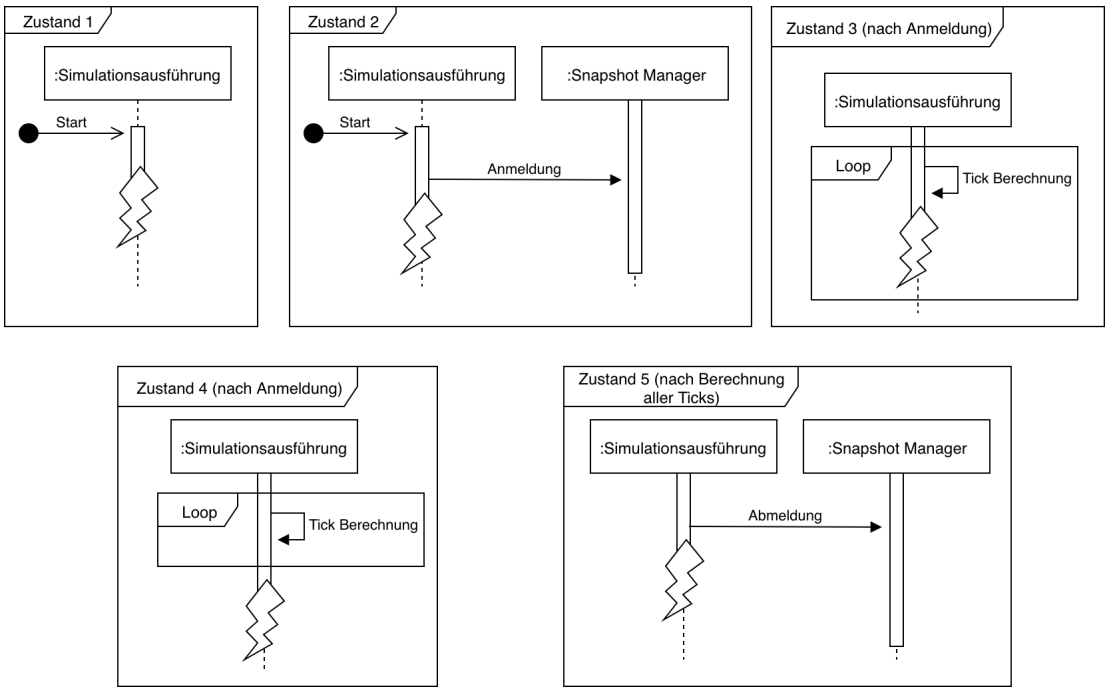

Abbildung 4.1: Die Ausfälle der Simulationsausführung

4.1.3 Snapshot Manager Ausfall

Auch einer der Snapshot Manager (siehe Use Cases) kann ausfallen. Es gibt folgende Zustände, in denen ein Snapshot Manager ausfallen kann:

Zustand 1 Der Snapshot Manager bearbeitet keine Anfrage und prüft nichts.

Zustand 2 Der Snapshot Manager bearbeitet gerade eine Anfrage oder prüft.

Der 1. Zustand ist hierbei der leichteste, da der Snapshot Manager in dem Moment nichts tut, kann dieser Zustand sehr leicht wiederhergestellt werden. Für die Wiederherstellung muss nur ein neuer Snapshot Manager gestartet werden.

Bei Zustand 2 muss berücksichtigt werden, dass die Aktion (Bearbeitung einer Anfrage, Prüfung) erneut durchgeführt werden muss. Die Daten die während der fehlgeschlagenen Aktion manipuliert wurde, müssen wieder in den Ursprungszustand zurückgesetzt werden.

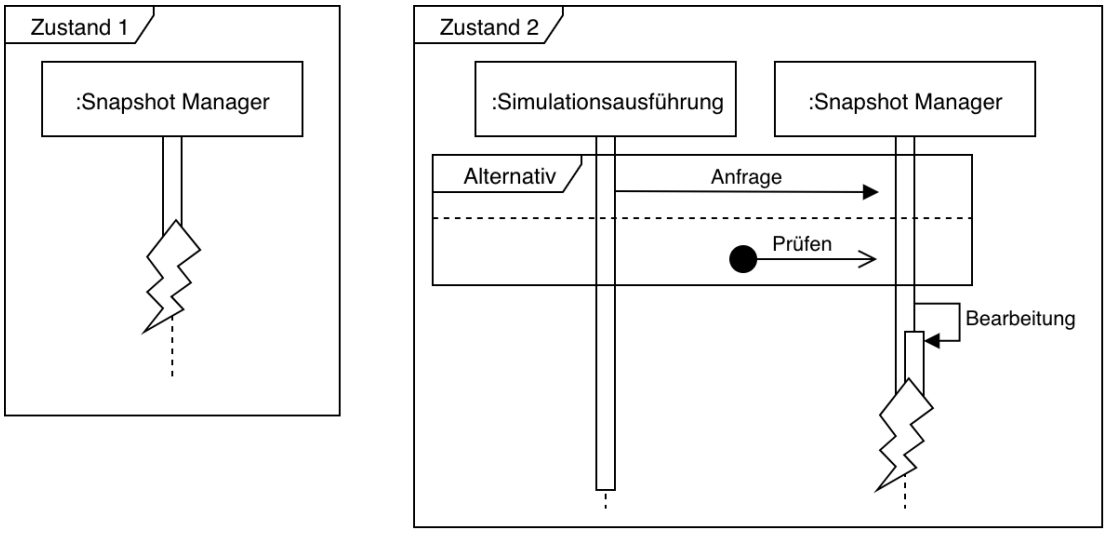

Abbildung 4.2: Die Ausfälle des Snapshot Managers

4.1.4 Verlorene Nachrichten

Wie in anderen verteilten Systemen, können hier auch Nachrichten verloren gehen. So werden hierbei folgende Fehlerfälle beachtet:

Ausfall 1 Die Nachricht von der Simulationsausführung zum Snapshot Manager kam nicht an.

Ausfall 2 Die Antwort des Snapshot Managers kam nicht bei der Simulationsausführung an.

Mit Ausfall 1 muss so umgegangen werden, dass man nach endlicher Zeit die Nachricht erneut schickt oder ohne erneutes Senden fortfährt, sonst entsteht ein Deadlock.

Der 2. Ausfall ist dem ersten aus Sicht der Simulationsausführung sehr ähnlich, denn sie bekommt keine Antwort. Allerdings nicht aus Sicht des Snapshot Managers, denn dieser kann aufgrund der Nachricht schon Daten manipuliert haben. Der Ausfall wird toleriert, in dem die Simulationsausführung wie in Ausfall 1 reagiert. Ein erneutes Senden der Nachricht, darf keine weitere Manipulation mit sich ziehen.

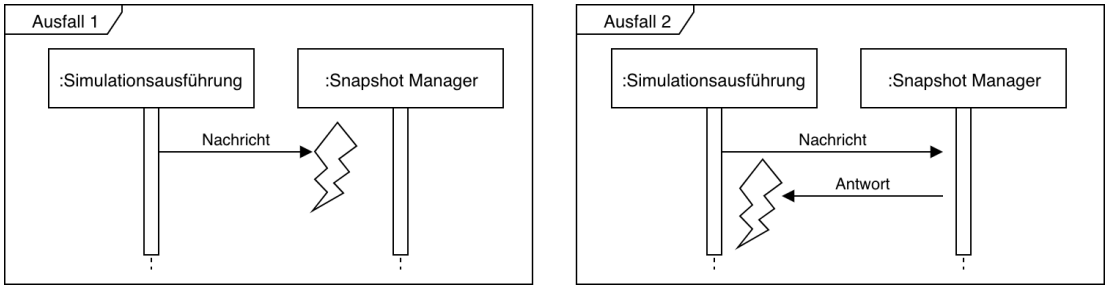

Abbildung 4.3: Fehler beim Senden von Nachrichten

4.1.5 Netzwerk Ausfall

Hin zu den bereits besprochenen Fehlern kann auch das Netzwerk instabil werden. Hierfür berücksichtigte ich die folgenden Fehler:

Ausfall 1 Die Simulationsausführung erreicht keinen Snapshot Manager.

Ausfall 2 Die Simulationsausführung ist vom Cluster abgeschnitten.

Ausfall 3 Der Snapshot Manager ist vom Cluster abgeschnitten.

Ausfall 4 Der Snapshot Manager und die Simulationsausführung sind vom Cluster getrennt.

Ausfall 1 führt dazu, dass sich die Simulationsausführung nicht An- oder Abmelden kann und dem Snapshot Manager keine neuen Snapshot IDs zu senden kann. Der Snapshot Manager wiederum muss richtig erkennen, warum die Heartbeats ausbleiben und darf keinen Neustart anstoßen. Siehe auch Kapitel 4.2.2, 4.2.3 und 4.2.4.

Ausfall 2 stellt den Totalausfall für die Simulationsausführung dar, da hier gegenüber Ausfall 1 keine Ergebnisse gespeichert werden können. In diesem Fall muss von außen manuell eingegriffen werden. Analog zu Ausfall 2 ist Ausfall 3 der Totalausfall aus Sicht des Snapshot Managers. Ausfall 4 ist ähnlich, zwar hat die Simulationsausführung noch Kontakt zu Snapshot Manager, kann aber keine Ergebnisse speichern.

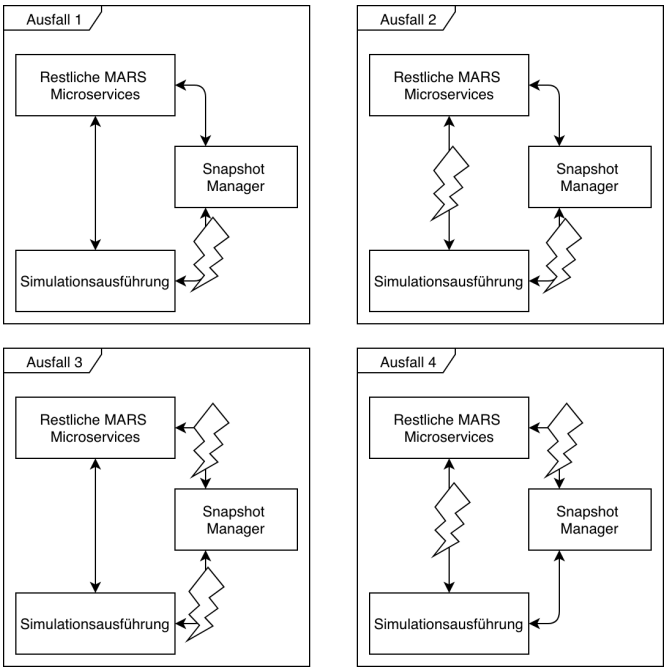

Abbildung 4.4: Fehler durch instabiles Netzwerk

4.2 Use Cases

Die erforderte Funktionalität kann man in fünf Use Cases einteilen. Im Zuge, dessen wird der Snapshot Manager eingeführt, seine Aufgaben sind die Überwachung und Wiederherstellung der Simulationsausführungen. Der Snapshot Manager ist ein separater Service im Kubernetes Cluster.

UC01	Anmeldung der Simulationsausführung
UC02	Erstellung der Snapshots
UC03	Senden von Heartbeats
UC04	Prüfen der Simulationsausführungsdaten und ggf. Neustarten des ausgefallenen Simulationsausführungen
UC05	Abmeldung der Simulationsausführung

Tabelle 4.1: Die festgestellten Use Cases

Folgend werden alle Use Cases einzeln näher erläutert.

4.2.1 UC01 Anmeldung

Damit der Snapshot Manager weiß, welche Simulationsausführungen ausgefallen sind, ist es wichtig zu wissen, welche Simulationsausführungen gestartet wurden. Aus diesem Grund meldet sich die Simulationsausführung bei dem Snapshot Manager, dies gilt auch bei Neustart nach einem Ausfall.

Auslöser	**Die Simulationsausführung meldet sich beim Snapshot Manager an.**
Preconditions	-
Postconditions	- Die Simulationsausführung erhielt eine positive Antwort. - Der Snapshot Manager konnte die Anmeldung erfolgreich bearbeiten.
Erfolgreiche Sequenz	**1.** Die Simulationsausführung meldet sich direkt nach Start und vor dem ersten Tick bei dem Snapshot Manager an. **2.** Snapshot Manager empfängt die Anmeldung und bearbeitet diese. **3.** War die Bearbeitung erfolgreich, wird der Simulationsausführung positiv geantwortet.
Erweiterungen	**2a.** War die Bearbeitung nicht erfolgreich, so wird der Simulationsausführung negativ geantwortet. Die Simulationsausführung wird die Anmeldung stetig anstoßen, bis sie eine positive Antwort erhält.

4.2.2 UC02 Snapshot Erstellung

Ist die Simulationsausführung dabei die Ticks zu berechnen, wird nach einer spezifizierten Zeit ein Snapshot erstellt. Theoretisch sollte ein Snapshot so viele Daten wie nötig umfassen, damit bei Neustart der exakt selbe Zustand wiederhergestellt werden kann, da dies den Rahmen dieser Bachelorarbeit überschreiten würde sind die Snapshots hier die Ergebnisdaten aus den Tick Berechnungen. Nichtsdestotrotz, wird ein Snapshot so behandelt als wäre er ein vollständiger Weltzustand.

Die Erstellung der Snapshots muss sequentiell zwischen den Tick Berechnungen geschehen. Bei einer vorhandenen Parallelität wird sonst die Berechnung des nächsten Ticks schon Agenten und Layer Objekte (siehe Hüning u. a. (2016)) verändern, die noch nicht im Snapshot gespeichert werden konnten, dies wird zu einer Inkonsistenz der Daten führen, da manche Objekte noch aus Tick n und andere aus Tick n+1 wären.

Die Layer Daten werden ohne inhaltliche Transformation in den Snapshot gespeichert. Die Simulationsausführung wird sonst zu lange für seine eigentliche Arbeit, die Simulation, brauchen. Ein weiterer Aspekt ist, dass ein Ausfall und somit eine Transformation der Layer Daten nicht zwingend passiert. Die Snapshot Erstellung dagegen mehrmals pro Simulationsausführung durchgeführt werden muss.

Auslöser	**Die Simulationsausführung sendet dem Snapshot Manager eine Snapshot ID.**
Postconditions	- Die Simulationsausführung hat den aktuellen Tick fertig simuliert und dessen Ergebnisse in die Result Datenbank geschrieben. - Die Simulationsausführung ist angemeldet.
Postconditions	- Die Simulationsausführung konnte den aktuellen Snapshot vollständig abspeichern. - Der Snapshot Manager konnte die von der Simulationsausführung erstellte Snapshot ID zu den bestehenden Simulationsausführungsdaten hinzufügen.

Erfolgreiche Sequenz	**1.** Die Simulationsausführung hat einen Tick vollständig simuliert und die Ergebnisse vollständig in Result DB geschrieben. **2.** Danach prüft die Simulationsausführung, ob ein Snapshot erstellt werden muss (gemäß spezifizierten Tickintervall). Ist diese erfolgreich, wird hier fortgefahren. **3.** Die Simulationsausführung erstellt ein neues Snapshot Objekt und speichert dies in die Snapshot Datenbank. Daraufhin teilt er dem Snapshot Manager mit, welche Snapshot ID neu erstellt wurde. **4.** Erst nach dem Empfang der positiven Antwort des Snapshot Managers befüllt die Simulationsausführung nun den Snapshot mit Daten.
Erweiterungen	**2a.** Wenn die Prüfung Negativ ist, wird mit der Berechnung des nächsten Ticks fortgefahren. **4a.** Wurde keine positive Antwort oder der Anfrage Timeout lief aus, wird die Nachricht eine bestimmte Zahl oft wiederholt. Ist nach dieser Anzahl keine positive Antwort gekommen, so wird ohne Snapshot fortgefahren.

4.2.3 UC03 Heartbeats

Durch die Anmeldung ist dem Snapshot Manager bekannt, welche Simulationsausführungen aktiv sein sollten. Um festzustellen, ob eine Simulationsausführung noch aktiv ist, werden Heartbeats verwendet (siehe 2.1).

Die Heartbeats werden hierbei in einem spezifizierten Rhythmus von den Simulationsausführungen an den Snapshot Manager geschickt. Der erste Heartbeat wird vor dem ersten Tick gesendet. Der letzte Heartbeat wird nach dem letzten Tick gesendet.

Auslöser	**Die Simulationsausführung sendet einen Heartbeat an den Snapshot Manager.**
Preconditions	- Die Simulationsausführung ist angemeldet.
Postconditions	- Der Snapshot Manager konnte die Heartbeat Daten persistent speichern. - Die Simulationsausführung erhielt eine positive Antwort.
Erfolgreiche Sequenz	**1.** Die Simulationsausführung schickt einen Heartbeat an den Snapshot Manager. **2.** Der Snapshot Manager empfängt diesen Heartbeat, erstellt einen Zeitstempel und speichert ihn in den Simulationsausführungsdaten. **3.** War die Bearbeitung erfolgreich, wird dies der Simulationsausführung geantwortet.
Erweiterungen	**3a.** Auch eine Nicht erfolgreiche Bearbeitung wird der Simulationsausführung geantwortet. Die Simulationsausführung achtet bei der Antwort nur darauf, ob diese positiv oder negativ ist.

4.2.4 UC04 Prüfung & Neustart

Regelmäßig überprüft der Snapshot Manager die Snapshot & Heartbeat Daten die er über die Simulationsausführungen gespeichert hat. Es wird geprüft, ob die gespeicherten Snapshots bestimmte Kriterien erfüllen, wenn nicht werden sie gelöscht. Die Heartbeat Daten werden separat anhand eigener Kriterien auf ihre Aktualität geprüft.

Sollte ein Ausfall erkannt worden sein, so wird die Simulationsausführung neu gestartet. Hierfür werden zwei Datensätze aktualisiert, die Daten für die Agenteninitialisierung und die Startzeit des ersten Ticks. Beide Datensätze können aus dem aktuellsten Snapshot generiert werden. Zudem muss die Result Datenbank auf den Neustart vorbereitet werden, dies bedeutet die Löschung bestimmter Ergebnisse.

Auslöser	**Der Prüfungstimer im Snapshot Manager läuft ab.**
Preconditions	-
Postconditions	- Der Snapshot Manager hat die Simulationsausführungsdaten und die Snapshot Datenbank geprüft und dabei ggf. Ausfall festgestellt und einen Neustart eingeleitet. - Die Ergebnisse der Simulationsausführung sind auf den Tick des Snapshots zurückgesetzt.

Erfolgreiche Sequenz	**1.** Der Snapshot Manager iteriert über alle angemeldeten Simulationsausführungen. **2.** Zunächst prüft er anhand spezifizierter Kriterien die Snapshots der Simulationsausführung. Entsprechen die Snapshots nicht den Kriterien werden sie gelöscht. **3.** Danach stellt der Snapshot Manager fest, ob ein Ausfall vorliegt. **3.1.** Dafür analysiert er die vorhandenen Heartbeat Daten. Ist der aktuellste Heartbeat zu alt, so wird vorerst von einem Ausfall ausgegangen. **3.2.** Um letztendlich von einem Ausfall wirklich ausgehen zu können, muss das Netzwerk getestet werden, siehe Kapitel 4.1.5. Ist das Netzwerk nicht ausgefallen, wurde ein Ausfall festgestellt. **4.** Nun muss die ausgefallene Simulationsausführung neu gestartet werden. **4.1.** Der aktuellste Snapshot (anhand der Ticknummer) wird aus der Datenbank geladen und in die benötigte Form transformiert. Zudem wird berechnet, wie viele Ticks noch simuliert werden müssen. **4.2.** Anhand der gespeicherten Ticknummer in dem geladenen Snapshot, werden alle Ergebnisse von größeren Ticknummern der Simulationsausführung aus der Result Datenbank gelöscht. **4.3.** Zuletzt wird eine neue Simulationsausführung gestartet. Über die Parameter bekommt er die Daten für die Agenteninitialisierung und die Startzeit des ersten Ticks. Damit ist der Neustart vollzogen. Die Simulationsausführung wird sich nun wie gewohnt bei dem Snapshot Manager Anmelden.

Erweiterungen	**3.2a.** Es kann sein, dass ausschließlich die Simulationsausführung vom restlichen Netzwerk abgeschnitten ist. So wird eine Simulationsausführung als ausgefallen erkannt, obwohl sie noch aktiv ist. Der beschriebene Sonderfall lässt sich nicht lösen, da keinerlei Kontakt zu der Simulationsausführung besteht. **3.3a.** Sind andere Teile des Netzwerks nicht erreichbar, so wird von einem Netzwerk Ausfall und nicht von einem Simulationsausführungsausfall ausgegangen. In diesem Fall wird kein Neustart eingeleitet werden.

4.2.5 UC05 Abmeldung

Hat die Simulationsausführung alle Ticks berechnet und die letzten Ergebnisse abgespeichert, meldet sie sich vom Snapshot Manager ab. Der Snapshot Manager wird mit Ausnahme der Ergebnisse alle Daten der Simulationsausführung löschen und im Zuge dessen auch nicht mehr Überprüfen.

Auslöser	**Die Simulationsausführung meldet sich bei dem Snapshot Manager ab.**
Preconditions	- Die Simulationsausführung erkennt das Ende der Tick Berechnung (zum Beispiel nach Berechnung des letzten Ticks oder durch Laufzeitfehler). - Die Simulationsausführung beendete den Heartbeat Thread. - Die Simulationsausführung ist angemeldet.

Postconditions	- Die Simulationsausführung ist nicht mehr aktiv. - Der Snapshot Manager löschte die Snapshots und gespeicherten Daten des Simulationsausführungen.
Erfolgreiche Sequenz	**1.** Die Simulationsausführung erkennt, dass alle Ticks simuliert wurden und alle Ergebnisse persistent gespeichert sind. Daraufhin beendet er das Senden von Heartbeats und meldet sich bei dem Snapshot Manager ab. **2.** Der Snapshot Manager empfängt und bearbeitet die Abmeldung. Im Zuge, dessen löscht er alle vorhandenen Snapshots. **3.** War die Löschung der Snapshots erfolgreich, werden die Simulationsausführungsdaten gelöscht. **4.** Waren die beiden Löschungen erfolgreich so ist es auch die Abmeldung und der Simulationsausführung wird dementsprechend geantwortet.
Erweiterungen	**1a.** Die Simulationsausführung hatte einen Laufzeitfehler und meldet sich deswegen ab. **3a.** Sollten Fehler auftreten, bekommt die Simulationsausführung eine negative Antwort und wiederholt die Abmeldung, bis er eine positive Antwort erhalten hat.

4.3 Fachliches Datenmodell

In Bezug auf die spezifizierten Use Cases, wird hier nun ein fachliches Datenmodell abgeleitet.

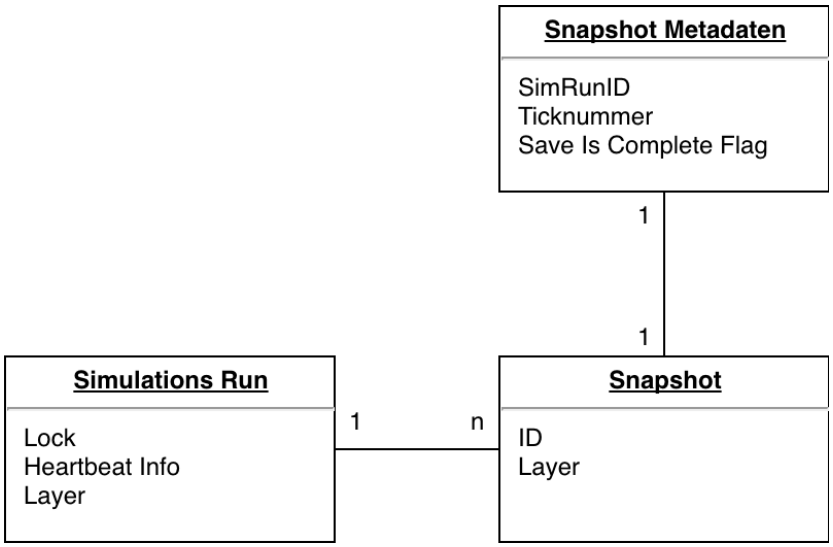

Abbildung 4.5: Das fachliche Datenmodell

4.4 Konstante Werte

Vorbereitend für die (nicht) funktionellen Anforderungen werden Konstanten mit Begründung und Quelle(n) definiert.

4.4.1 Intervalle und Timeouts

Snapshot Intervall

Der Snapshot Intervall gibt an, wie oft ein Snapshot erstellt wird. Hierbei muss zwischen höherer Ausfallsicherheit, durch aktuellere Snapshots, oder schnellerer Tick Berechnung, durch weniger Snapshots, abgewägt werden.

Oracle allein nennt in der Dokumentation zu ihrer TimesTen Datenbank [Oracle (2017)] eine Default Checkpoint Frequenz von 600 Sekunden (= 10 Minuten), diesen Wert übernehme ich, da keine Alternative gefunden werden konnte.

Heartbeat Intervall

Mit dem Heartbeat Intervall wird ausgedrückt, wie oft ein Heartbeat gesendet wird. Wie in anderen Arbeiten auch (siehe Kapitel 2), werden die Heartbeats in einem festen zeitlichen Abstand gesendet. Hinzu kommt, dass damit gerechnet werden muss, dass Heartbeats verloren

gehen (siehe Kapitel 4.1.4), somit dürfen 1 bis 2 fehlende Heartbeats nicht direkt zu einem falschen Neustart führen.

Auf diese Feststellungen aufbauend muss nun abgewägt werden, ob man die Heartbeats schneller schickt und somit einen Ausfall schneller erkennen kann oder, ob man die Heartbeats langsamer schickt und das Netzwerk nicht so stark belastet. Kubernetes definiert für ihre "healthz" Abfragen einen Standardwert von 10 Sekunden (siehe "periodSecond" in Kubernetes (2018a)). Der Falcon Fehlerdetektor verwendet eine Zeit von 5 Sekunden [Leners u. a. (2011)].

Durch den Umstand, dass MARS kein System ist, dass in Echtzeit auf Kundenabfragen reagieren muss, bestehen keine harten Zeitkriterien. Um die Last auf das Netzwerk niedrig zu halten, entscheide ich mich deswegen für den höheren Wert von 10 Sekunden den auch Kubernetes verwendet.

Anfragen Retry Anzahl & Timeout

Die Retry Anzahl in der Simulationsausführung gibt an, wie oft eine Nachricht zum Snapshot Manager erneut gesendet wird, zum Beispiel nach einer negativen Antwort. Der Timeout Wert gibt an, wie lange die Simulationsausführung auf eine Antwort wartet, bis er die Nachricht erneut schickt.

Um Lastspitzen zu berücksichtigen, muss dem Snapshot Manager und dem Netzwerk genug Zeit gegeben werden. Für TCP wurden hierzu 3 Sekunden spezifiziert [Braden (1989)], diese werden übernommen. Die Nachrichten der Simulationsausführung an den Snapshot Manager werden maximal 5 Mal wiederholt, da sonst die Tick Berechnung zu sehr verlangsamt wird. Einzige Ausnahmen sind Heartbeats, diese werden zweimal wiederholt, da nach weiteren 10 Sekunden ohnehin der nächste Heartbeat gesendet wird (siehe 4.4.1).

Prüfungsintervall

Bei dem Prüfungsintervall geht es darum, wie oft die Daten im Snapshot Manager geprüft werden. Kubernetes prüft alle 10 Sekunden [Kubernetes (2018a)], ob die Pods noch aktiv sind. Vorerst ist dieser Wert in Ordnung für MARS, denn in der Regel sind nicht mehr als 10 Simulationsausführungen aktiv. Sollte diese Zahl in Zukunft steigen, muss dieses Intervall erhöht werden, da die Prüfung zu weiterer Last für CPU und Netzwerk führt.

Heartbeat Timeout

Der Heartbeat Timeout legt fest, wie viel Zeit vergehen muss, bevor der Snapshot Manager einen Simulationsausführungsausfall vermutet. Kubernetes verwendet einen Standard Timeout von

einer Sekunde (“timeoutSeconds” in Kubernetes (2018a)), dies berücksichtigt keine verlorenen Nachrichten.

Aus diesem Grund wird erst nach drei fehlenden Nachrichten ein Ausfall vermutet. Der Heartbeat Intervall beträgt 10 Sekunden, also wird nach 30 Sekunden von einem Ausfall ausgegangen, dies ist im Rahmen dessen, was auch andere Systeme verwenden (12 bis 60 Sekunden, Leners u. a. (2011)).

Lock Timeout

Die parallele Ausführung mehrerer Snapshot Manager und die gemeinsame Datenbasis setzen voraus, dass gesteuert wird, welcher Prozess mit welchen Daten arbeiten darf. In diesem Fall wird ein einfacher Lock verwendet. Bei Setzen eines Locks wird ein Locktext und der aktuelle Zeitstempel gesetzt. Nur der Snapshot Manager, der den Lock gesetzt hat, darf ihn wieder entfernen. Sollte dieser Snapshot Manager jedoch zwischen Setzen und Löschen ausfallen, wird ein Timeout benötigt, um einen Deadlock zu vermeiden. Der Timeout gibt an, wie viel Zeit verstreichen muss, bis der gesetzte Lock ignoriert wird.

Hierzu gibt es wenige Defaultwerte. Microsoft (2017) hat zum Beispiel einen unendlichen Default Wert. Ein Gegenteil ist EssBase Datenbank von Oracle, die einen Standard Timeout von 3600 Sekunden hat [Oracle (2018)].

3600 Sekunden, sind allerdings zu hoch für MARS. Wird ein Snapshot Manager einen Datensatz locken und dann abstürzen, kann dieser Datensatz für 60 Minuten nicht geändert werden. In dieser Zeit sind manche kleine Simulationen schon komplett fertig, deswegen braucht es einen wesentlich geringeren Wert.

Es wird der Heartbeat Timeout als Grundlage verwendet, denn dieser wird in einer ähnlichen Situation benutzt. Heartbeat und Lock Timeout sind insofern unterschiedlich, dass auf Basis des Heartbeat Timeouts ein Ausfall einer Simulationsausführung und auf Basis des Lock Timeouts ein Ausfall eines Snapshot Manager erkannt werden kann.

4.4.2 Sonstige

Snapshot Datenbank Kriterien

Wie in Use Case 4 definiert (4.2.4), werden die Snapshots regelmäßig geprüft. Dafür wird definiert anhand welcher Kriterien sie geprüft werden. Es gelten folgende Kriterien:

Anzahl der vollständigen Snapshots beträgt maximal 2 zu jeder Zeit.

Unvollständige Snapshots werden gelöscht.

Anzahl von Snapshot Managern

Um zu jeder Zeit mindestens einen aktiven Snapshot Manager im Kubernetes Cluster zu haben, braucht es mindestens drei Snapshot Manager, diese Zahl erklärt sich wie folgt:

Kubernetes fährt unter Umständen Pods eines Kubernetes Services herunter (Kapitel 3.3). Zur selben Zeit werden maximal 25% aller Repliken, beziehungsweise mindestens ein Pod (Default Wert siehe Kubernetes (2018b)) heruntergefahren. So wären bei 3 Repliken zu einem Zeitpunkt des Neustarts noch 2 Snapshot Manager aktiv. Sollte nun noch ein anderer Host mitsamt Pods ausfallen, ist immer noch einer aktiv.

Zuzüglich wird diese Replika Anzahl in den diesbezüglichen Dokumentationen als Standardwert genannt (zum Beispiel Kubernetes (2018b), Kubernetes (2018f), Kubernetes (2018g)). Aus diesen beiden Gründen verwende ich diesen Wert.

4.5 Funktionale Anforderungen

Mit Bezug auf die vorher gehenden Probleme und Use Cases, wird hier herausgearbeitet, welche Anforderungen an das System bestehen. Zudem ist vermerkt, worauf sich die Anforderung stützt.

	Beschreibung	**Referenz**
F01	Der Kubernetes Cluster darf die Pods der Simulationsausführungen nicht selbst Neustarten.	4.1.1, 4.2.4
F02	Es darf zu keinen Problemen führen, wenn mehrere Nachrichten einer Simulationsausführung an unterschiedliche Snapshot Manager gehen. Alle müssen auf denselben Daten arbeiten. Es darf auch kein Problem sein, wenn verschiedene Snapshot Manager gerade mit Daten der selben Simulationsausführung arbeiten.	3.3, 4.2.2, 4.2.3, 4.2.4
F03	Zu jeder Zeit muss mit einem Ausfall eines Snapshot Manager gerechnet werden. Ein Ausfall darf zu keinem Verlust von Daten oder Anfragen führen. Nicht vollständig manipulierte Daten müssen wieder in den Ursprungszustand zurückgesetzt werden.	3.3
F04	Sollte ein Netzwerk Ausfall bestehen, so muss dies der Snapshot Manager erkennen und dies bei der Prüfung die Simulationsausführung Heartbeats berücksichtigen, um nicht auf falsche Ergebnisse zu kommen.	4.1.5, 4.2.4

F05	Der Snapshot Manager weiß erst mit Anmeldung der Simulationsausführung, welche aktiv sind.	4.2.1, 4.2.4
F06	Die Simulationsausführung teilt dem Snapshot Manager mit, welchen Snapshot sie gerade erstellt hat.	4.2.2
F07	Sind seit dem letzten Snapshot, oder dem Systemstart, 600 Sekunden vergangen. So wird ein neuer Snapshot erstellt.	4.1.2, 4.2.2, 4.4.1
F08	Der Snapshot Manager vermutet einen Ausfall einer Simulationsausführung, wenn sein letzter Heartbeat mehr als 30 Sekunden her ist.	4.1.2, 4.2.3, 4.2.4, 4.4.1
F09	Alle 10 Sekunden schickt die Simulationsausführung dem Snapshot Manager einen Heartbeat.	4.1.2, 4.2.3, 4.4.1
F10	Es darf kein Problem darstellen, wenn Anfragen bei dem Snapshot Manager nicht erfolgreich bearbeitet werden konnten oder wenn Nachrichten verloren gehen. Ein Retry findet nach 3 Sekunden statt. Die Anfragen werden 5 Mal wiederholt, außer bei Heartbeats, diese werden 2 Mal wiederholt.	4.2.1, 4.2.2, 4.2.3, 4.2.5, 4.1.4, 4.4.1
F11	Für die Snapshot Prüfung gelten folgende Kriterien. Alle unvollständigen müssen gelöscht werden. Es dürfen nur maximal 2 der aktuellsten Snapshots gespeichert werden.	4.2.4, 4.4.2
F12	Der Snapshot Manager prüft, angelehnt an den Heartbeat Timeout, alle 30 Sekunden, ob eine Simulationsausführung ausgefallen ist und ob Snapshots gelöscht werden müssen.	4.2.4, 4.4.1
F13	Der Snapshot Manager muss in der Lage sein, neue Simulationsausführungen starten zu können.	4.2.4
F14	Der Snapshot Manager weiß erst mit Abmeldung der Simulationsausführung, dass dieser nicht mehr neu gestartet werden muss.	4.1.2, 4.2.4, 4.2.5
F15	Sollte die Simulationsausführung keinen Kontakt mehr zu einem Manager, jedoch noch Verbindung zu der Result Datenbank haben, so muss die Simulationsausführung mit der Tick Berechnung fortfahren.	4.1.5
F16	Ein gesetzter Lock eines Simulationsausführungsdatensatzes wird nach 30 Sekunden ignoriert.	4.4.1

Tabelle 4.7: Übersicht über alle funktionalen Anforderungen

4.6 Nicht funktionale Anforderungen

Auf den funktionalen Anforderungen aufbauend, werden hier noch weitere Anforderungen festgelegt.

	Beschreibung	**Referenz**
NF01	Es werden 3 Repliken des Snapshot Managers gestartet, damit zu jeder Zeit mindestens einer aktiv ist.	3.3, 4.4.2
NF02	Die vom Snapshot Manager angebotene Schnittstelle ist bei jedem Aufruf Idempotent.	4.1.4

Tabelle 4.8: Übersicht über alle Nicht funktionalen Anforderungen

5 Entwurf

In dem Entwurfskapitel werden folgend die Prozess-, Kontext-, Baustein-, Laufzeit- und Verteilungssicht beschrieben, um den neuen Microservice aus verschiedenen Blickwinkel zu betrachten.

5.1 Fehlererkennung und Zustandswiederherstellung

Die Ausfälle von Simulationsausführungen werden mit Heartbeats in Verbindung mit Timeouts erkannt. Die Nachteile sind wie in Kapitel 2.1 beschrieben, eine ungenaue Erkennung von Ausfällen mit einer ggf. langen Erkennungszeit. Die Vorteile dagegen sind eine sehr generelle Verwendung, von einem zum anderen System müssen nur die Timeout Werte angepasst werden. Ferner ist dieser Mechanismus leicht zu verstehen und leicht zu implementieren. Die Vorteile überwiegen in dieser Arbeit, da eine genaue Erkennung mit kurzen Fehlerzeiten nicht vonnöten sind.

Um Zustände vor einem Ausfall wiederherzustellen werden Snapshots (Checkpoints) erstellt. Der Nachteil ist, dass jedes Mal ein vollständiger Snapshot erstellt wird und dies viel Speicherplatz und CPU Zeit benötigt. Der Vorteil ist, dass ein Snapshot ohne weitere Snapshots direkt genutzt werden kann und sich dieses Verfahren leicht verständlich ist. Die Snapshots werden erstellt und genutzt, aufgrund ihrer Verständlichkeit und ihrer Vollständigkeit. Die Vorteile gegenüber den Einbußen in der Performanz.

5.2 Prozesssicht

Hierfür werden als Grundlage die spezifizierten Use Cases (Kapitel 4.2) verwendet. Das folgende Aktivitätsdiagramm stellt den gesamten Ablauf des Systems abstrakt dar.

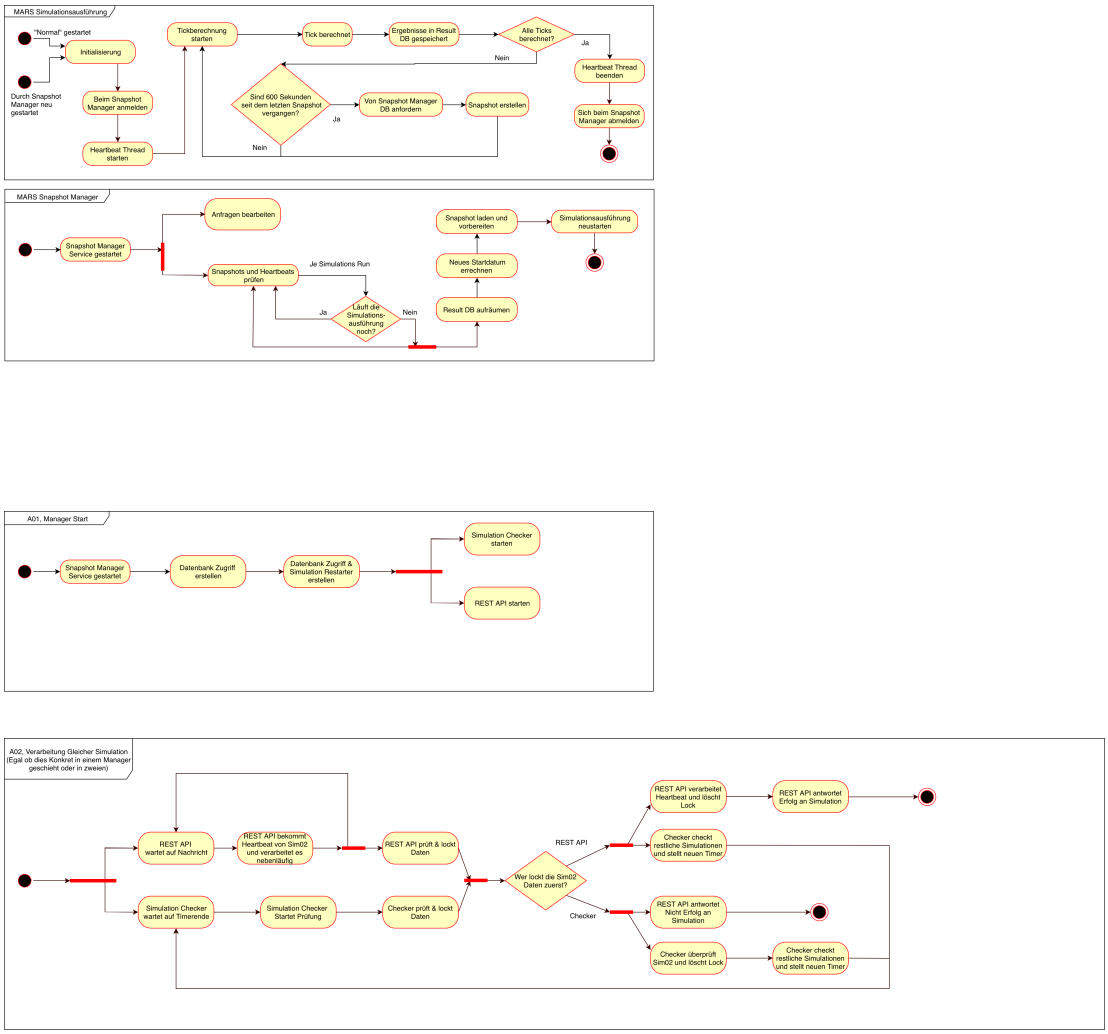

Abbildung 5.1: Grober Ablauf der Simulationsausführung und des Snapshot Managers

In der Simulationsausführung, werden wie gefordert die Tick Berechnung, Ergebnis Speicherung und Snapshot Erstellung sequentiell berechnet (F09). Neben läufig ist der Heartbeat Prozess aktiv, dieser sendet stetig Heartbeats an den Snapshot Manager. Der Heartbeat Prozess startet nach der Anmeldung bei dem Snapshot Manager und vor der Berechnung des ersten Ticks. Er terminiert nach der Berechnung des letzten Ticks und vor der Abmeldung bei dem Snapshot Manager.

Im Snapshot Manager laufen die Bearbeitung von Anfragen und die Prüfung der gespeicherten Daten parallel. Wurde ein Ausfall einer Simulationsausführung festgestellt, wird diese neben läufig neu gestartet. Der Snapshot Manager terminiert nicht.

5.3 Kontextsicht

Hier geht es um die Einbettung des neuen Snapshot Managers in das MARS Cluster. Der Snapshot Manager kommt mit dem “Sim-Runner-Service”, der Simulationsausführung (auch “Simulation Run” genannt) und der Mongo DB in Kontakt. Die Mongo DB wird in diesem Fall für die Speicherung von Snapshots und der Simulationsausführungsdaten genutzt.

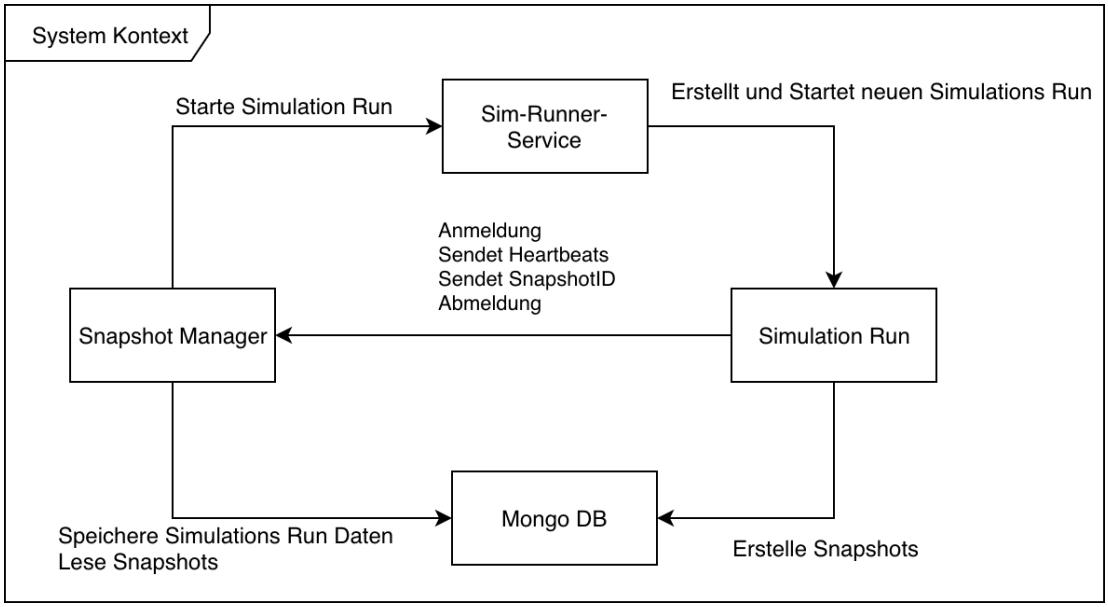

Abbildung 5.2: Kontextsicht der Simulationsausführung und des Snapshot Managers

5.4 Bausteinsicht

Der neue Snapshot Manager wird in 4 Komponenten unterteilt, die nach Verantwortlichkeiten getrennt sind.

Name	Verantwortlichkeit(en)

Simulations Run Checker	Die Simulations Run Checker Komponente enthält Funktionen, um zu überprüfen, ob Snapshots gelöscht werden müssen und ob die Simulationsausführungen noch aktiv sind. Die Kriterien hierfür sind in Kapitel 4.5 Anforderungen beschrieben. Ist die Prüfung der Heartbeats positiv (Ausfall), so wird der Neustart der Simulation mithilfe der Simulations Run Restarter Komponente vorbereitet und angestoßen.
Simulations Run Restarter	Die Komponente bereitet die Daten vor (transformiert den aktuellsten Snapshot in eine .csv Datei) und stößt den Neustart der Simulationsausführung an (F13). Wichtig zu beachten ist, dass diese Komponente die Kubernetes Funktion ersetzt die Simulationsausführungen neu zu starten, weswegen diese Kubernetes Funktion explizit deaktiviert wurde (F01).
Datenbankzugriff	Hier werden CRUD Funktionen angeboten um den Zugriff auf Datenbanken (Snapshot & Result & Simulationsausführung Datenbank) zu vereinfachen. So werden die direkten Zugriffe gekapselt und nur diese CRUD Methoden implementiert die auch benötigt werden, dies hilft dabei dem Entwickler ein klares Bild zu geben, wie er mit den Klassen arbeiten soll.
REST API	Die REST API dient der Kommunikation mit der Simulationsausführung und mit Kubernetes (siehe Kubernetes (2018c), Kapitel 4.2.3). Mit der Wahl von REST API hat man eine Programmiersprachen unabhängige Schnittstelle, hinzukommt eine offensichtliche und verständliche Versionierung. Die Nennung der Api Version in den Ressourcenpfaden (“/v1/...”), macht dem Nutzer schnell klar, mit welcher Version er arbeitet. Die Pfade sind alle Idempotent, gemäß NF02. Die verwendeten Ressourcen und deren Verhalten leiten sich direkt aus den Use Cases und den Anforderungen ab (4.2, 4.5, 4.6). An dieser Stelle wird folgend auf den abstrakten Entwurf der Schnittstellen eingegangen. Der konkretere Entwurf (u. a. mit HTTP Codes nach Fielding u. a. (1999)) steht in einer separaten Swagger Datei.

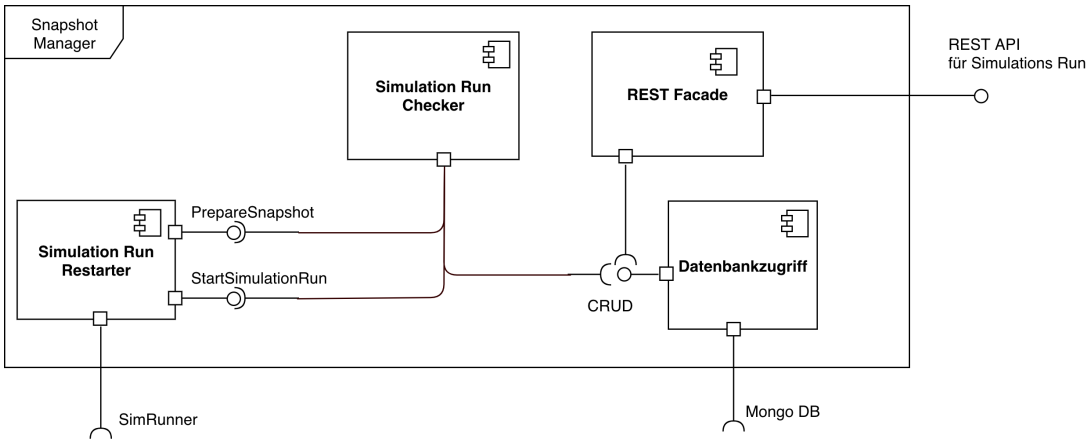

Abbildung 5.3: Bausteinsicht des Snapshot Managers

5.4.1 REST

Folgende Behandlung von Fehlerfällen haben alle REST Pfade gemeinsam:

Sollte keine Verbindung zu der nötigen Datenbank vorhanden sein oder sollten die benötigten Daten nicht lesbar (aber vorhanden) sein. Zum Beispiel, weil sie gerade durch einen anderen Prozess gelockt wurden (F02), gilt die Anfrage als nicht erfolgreich bearbeitet.

Konnte der Snapshot Manager die Anfrage nicht erfolgreich bearbeiten, antwortet er der Simulationsausführung mit dem Error Text und einem negativen HTTP Code.

Anmeldung

Zu aller erst braucht es einen REST Pfad der die Anmeldung der Simulationsausführung bei dem Snapshot Manager ermöglicht (F05). Die Anfrage muss folgende Werte enthalten:

- ID der Simulationsausführung

- Gesamte Tick Anzahl

War die Anmeldung erfolgreich, bekommt die Simulationsausführung eine Antwort mit positivem HTTP Code. Der Pfad ist Idempotent, da er die vorhandenen Daten stets mit den neuen überschreibt. Ein erneutes Senden der gleichen Anmeldung wird zur Speicherung der exakt selben Daten führen.

Speichern der aktuellen Snapshot ID

Mithilfe dieser REST Schnittstelle kann die Simulationsausführung dem Snapshot Manager die ID des neuen Snapshots mitteilen. Über diesen Weg kann der Manager zuordnen, welche Snapshot IDs zu welcher Simulationsausführung gehören (F06).

War die Bearbeitung der Anfrage erfolgreich, bekommt die Simulationsausführung eine positive Antwort. Die Schnittstelle ist Idempotent, da der Snapshot Manager keine Snapshot ID einer Simulationsausführung zweimal speichert.

Heartbeats

Die Simulationsausführung sendet dem Snapshot Manager eine Nachricht mit ID der Simulationsausführung. Der Snapshot Manager wird draufhin einen aktuellen Zeitstempel erstellen und in dem Datensatz der Simulationsausführung speichern.

War die Bearbeitung der Anfrage erfolgreich, bekommt die Simulationsausführung einen positiven HTTP Code. Wie bei den beiden Pfand zuvor, überschreiben die neuen Daten die alten, damit besteht die Idempotenz.

Abmeldung

Zuletzt muss sich die Simulationsausführung bei dem Snapshot Manager abmelden können (F14). Hierzu sendet sie dem Snapshot Manager nur ihre ID mit.

War die Abmeldung erfolgreich, so antwortet der Snapshot Manager mit einem positiven HTTP Code. Bei der Abmeldung werden die Daten gelöscht, deswegen sind mehrere Abmeldungen einer Simulationsausführung kein Problem, da die Daten stets gelöscht bleiben.

5.5 Laufzeitsicht

In diesem Kapitel werden Details der Use Cases und der Parallelität der Prozesse näher beschrieben. Die auf den in Kapitel 4 beschriebenen Use Cases basierenden Sequenzdiagramme sind wegen ihrer Größe im Anhang zu finden.

5.5.1 Einbettung der Ausfallsicherheit

Zur Steigerung der Fehlertoleranz werden verschiedene Fehler des Nachrichtenverkehrs und innerhalb des Snapshot Managers berücksichtigt.

Für den Nachrichtenverkehr bedeutet dies, dass zunächst für jede Nachricht ein Timer gestellt wird. Verstreicht dieser Timer, bevor eine Antwort des Snapshot Managers kam, wird

die Nachricht erneut gesendet. Ist eine Antwort empfangen worden, wird der Timer beendet und ihr HTTP Code geprüft. Erst danach wird die Antwort gegebenenfalls bearbeitet. In den Sequenzdiagrammen ist dies wie in Abbildung 5.4 dargestellt.

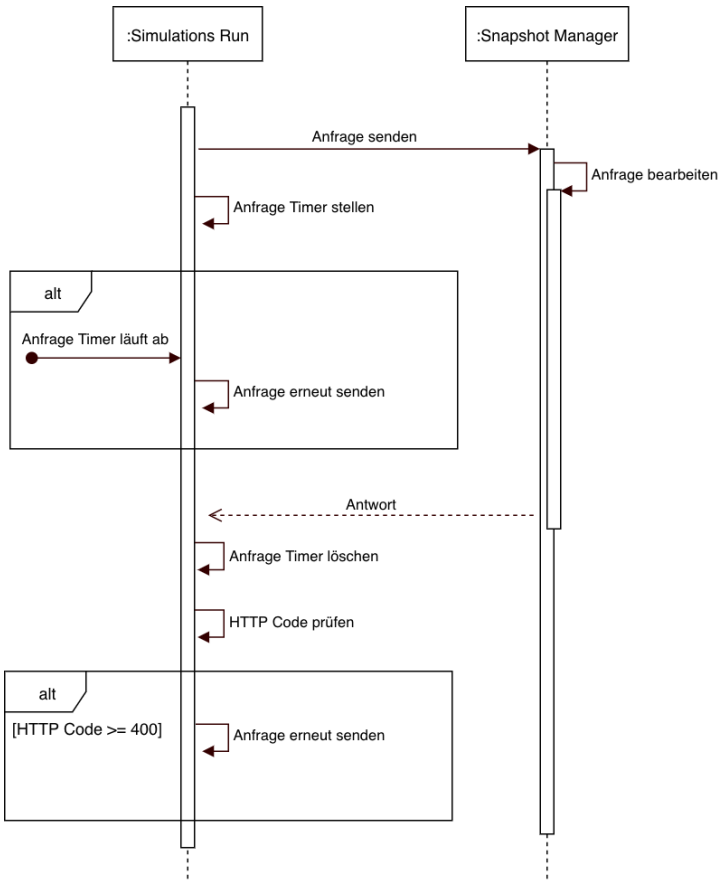

Abbildung 5.4: Beispielhafter und verkürzter Nachrichtenverkehr

Innerhalb des Snapshot Managers verhindern gelockte Daten oder eine fehlende Verbindung zur Datenbank, eine erfolgreiche Bearbeitung von Anfragen. In den Sequenzdiagrammen ist dies wie in Abbildung 5.5 dargestellt.

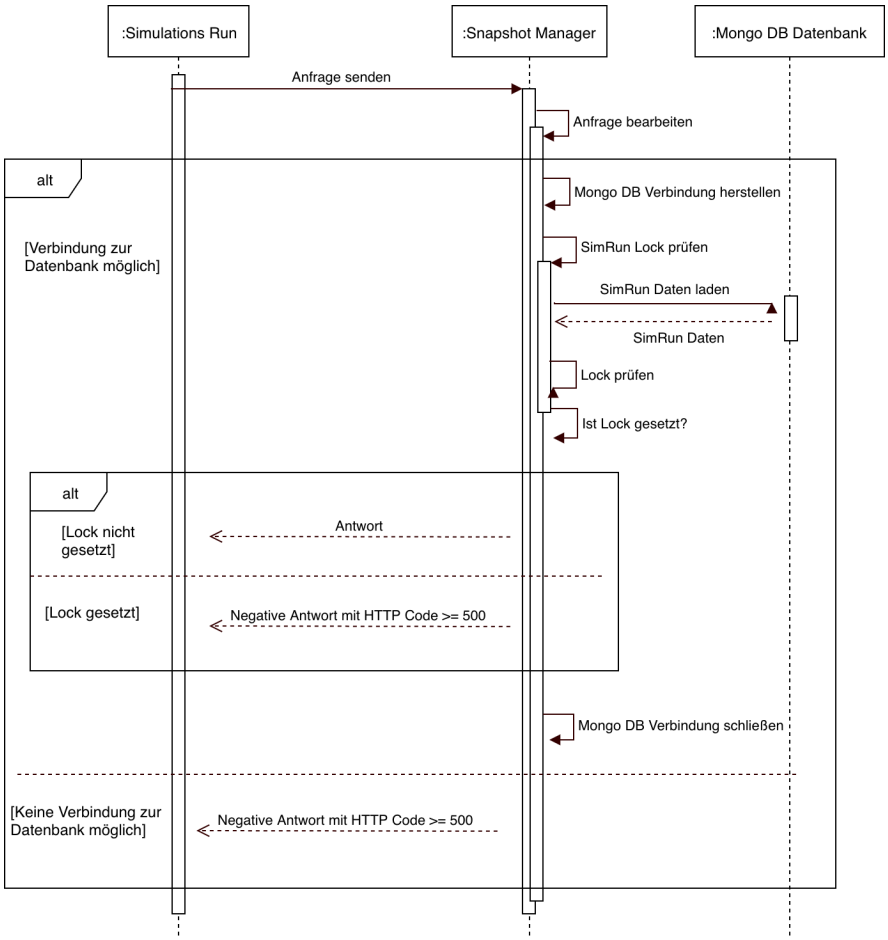

Abbildung 5.5: Beispiel für Probleme im Snapshot Manager

5.5.2 Aspekte der Parallelität von Snapshot Managern

Das folgende Beispiel zeigt, wie Use Case 3 und Use Case 4.1 parallel ablaufen können. In diesem Beispiel sendet eine Simulationsausführung (“S1”) einen Heartbeat (Use Case 3), dieser wird von Snapshot Manager “M1” empfangen während parallel Snapshot Manager “M2” mit der Prüfung beginnt (Use Case 4.1). Der Prozess, der den Heartbeat bearbeiten will lockt in diesem Fall die Daten zuerst, dies hat zur Folge, dass andere Prozesse diese Daten aktuell nicht verwenden können. Der prüfende Snapshot Manager “M2” überspringt deswegen die Daten und fährt mit den restlichen Daten fort.

Abbildung 5.6 dient der Veranschaulichung der Parallelität und stellt deshalb keine Details dar.

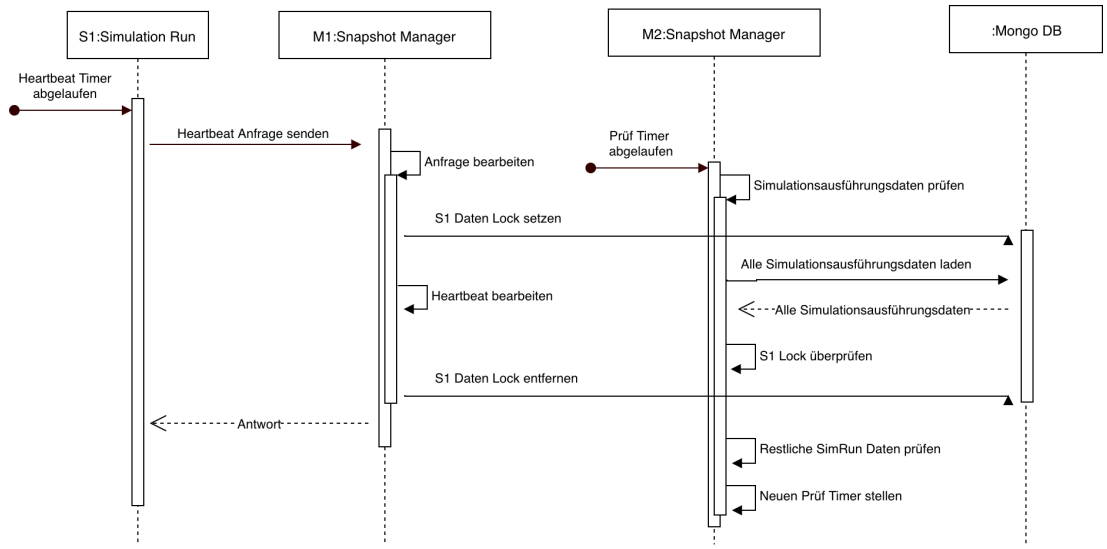

Abbildung 5.6: Laufzeitsicht von UC03 in Zusammenspiel mit UC04.1

6 Realisierung

In diesem Kapitel wird beschrieben, wie der Entwurf realisiert wurde. Dabei gehe ich auf die Quality Assurance, Integration der neuen Funktionalitäten in die MARS Umgebung, Designentscheidungen und Entwurfsmuster ein.

6.1 Quality Assurance

Um die zukünftige Wartbarkeit zu erhöhen, wird der Simulation Manager mithilfe von Clean Code [Martin (2013)] implementiert. Die Hauptgründe hierfür sind, dass auch andere MARS Mitarbeiter diesen Microservice Warten müssen. Der Microservice wird in den nächsten Jahren noch innerhalb der MARS Gruppe eingesetzt, besonders dann muss der Code für jeden anderen Entwickler lesbar und verständlich sein.

Clean Code wurde vor allem bei der Berücksichtigung von Datei-, Klassen-, Funktion-, Konstanten- und Variablennamen berücksichtigt. Ferner wurde darauf geachtet, dass jede Funktion eine Aufgabe erledigt. Des Weiteren wird in Martin (2013) definiert, dass eine Funktion 10 bis 15 Zeilen umspannt. Ein positives Beispiel hierfür ist die “StartContinousChecking” Methode der “SnapRunChecker” Klasse, zu sehen in Listing 6.1.

```
1  func (checker *SnapRunChecker) StartContinousChecking() {
2  var allSnapSimRuns []interfaces.ISnapshotSimRun
3  var getAllErr error
4
5  for true {
6  time.Sleep(interfaces.CheckingIntervallInSeconds)
7  log.Println("Start Checking")
8
9  allSnapSimRuns, getAllErr = checker.snapshotSimRunController.
       GetAllSnapSimRuns()
10
11 if getAllErr == nil {
12 checker.checkSnapSimRuns(allSnapSimRuns)
13 }
```

```
14 }
15 }
```

Listing 6.1: Beispiel für kleine Funktionen

In der "SnapRunChecker" Klasse ist zudem die durchschnittliche Länge der Methoden 16,33 Zeilen, inklusive Funktionsname und Zeilen die nur eine geschlossene geschweifte Klammer enthalten.

Sollte eine Funktion doch zwei Aufgaben zugleich erledigen, wird dies im Namen explizit klar gemacht und in der Funktion werden die zwei Aufgaben von je einer Teilfunktion übernommen. In Listing 6.2 ist dafür ein Beispiel aus der "SnapLock" Klasse zu finden. Durch das "And" im Namen soll klar gemacht werden, dass die Methode zwei Aufgaben hat. In der "CheckAndIfOKSetNewLock" Methode werden zwei Aufgaben zusammen gelegt, die zuerst unabhängig voneinander waren. Als aufgefallen ist, dass die beiden Methoden stets nacheinander ausgeführt werden, wurden sie zusammengelegt. Die Methode könnte allerdings noch vereinfacht werden, wenn die "if" Bedingung klarer formuliert oder in eine extra Funktion ausgelagert wird.

```
1 func (snaplock *SnapLock) CheckAndIfOKSetNewLock(newLockText string)
      bool {
2 var newLockWasSet bool
3
4 if !snaplock.checkIfLockIsStillValid() || snaplock.
     checkIfLockIsFreeForNewLock(newLockText) {
5 snaplock.setNewLockTextAndTimestamp(newLockText, time.Now())
6 newLockWasSet = true
7 }
8
9 return newLockWasSet
10 }
```

Listing 6.2: Beispiel Methode für Funktion mit zwei Aufgaben

Für jedes Objekt wird eine Schnittstelle (auch "Interface" genannt) definiert um die tatsächliche Verwendung von der Implementation zu trennen. Erst nach definieren und erstellen einer Schnittstelle wird das entsprechende Objekt implementiert. Die Implementation ist "Package Private" und nur die Factory Funktionen und Interfaces sind nach Außen sichtbar. Alle Interfaces sind in einem Interface Package um die Zahl der Komponenten Abhängigkeiten untereinander zu verringern.

Ferner werden komplexere Klassen “Test Driven” implementiert um Fehler früher zu finden und um eine höhere Softwarequalität zu erreichen. Dabei wurde die Implementation und die Tests parallel geschrieben. Der Mehraufwand lohnte sich insofern, dass ich bemerkte, dass nach Vervollständigung wesentlich weniger Fehler in diesen Komponenten enthalten waren, als in den Komponenten die nicht “Test Driven” implementiert wurden.

6.2 Integration der neuen Funktionalitäten in den “Simulation Run”

In den “Simulation Run” sind die Anmeldung (Use Case 1), Abmeldung (Use Case 5), das Senden von Snapshots (Use Case 2) und Heartbeats (Use Case 3) zu implementieren. Dafür werden drei Klassen zusätzlich erstellt. Eine Kommunikationsklasse die, die Kommunikation mit dem Manager kapselt, eine Client Klasse die als Funktionalität die Use Cases bietet und eine Klasse die, die nötigen Daten (ID und Gesamte Tick Anzahl) einer Simulationsausführung hält.

In der “SimulationStarter” Klasse wird vor der Vorbereitung des ersten Ticks das Senden der Anmeldung und der Start des Heartbeat Threads implementiert. In selbiger Klasse wird nach Berechnung des letzten Ticks die Abmeldung gesendet. In Listing 6.3 ist ein Auszug aus der “SimulationStarter” Klasse zu sehen. Die Erstellung von Snapshots inklusive Senden der Snapshot Id wird in die “StepExecutionUseCase” Klasse implementiert, direkt nach dem Speichern der Tick Ergebnisse.

```
1  //Anmeldung beim Simulation Manager
2  var url = "sim-manager-svc";
3  var client = new Core.SimSnapClient(url);
4
5  var simrunID = configuration.SimulationId.ToString();
6  var totalTickCount = simulationConfig.Globals.Ticks;
7  client.RegisterAtManager(simrunID, totalTickCount);
8
9  //Start Paralleles Senden von Heartbeats
10 var heartbeatThread = client.StartConcurrentHeartbeatCalls(simrunID);
11
12 //Start der Simulationsausfuehrung
13 [...]
14
15 //Beende senden von Heartbeats und sende Abmeldung
16 heartbeatThread.Interrupt();
```

```
17 client.UnregisterAtManager(simrunID);
```

Listing 6.3: Auszug aus der Implementation der “SimulationStarter” Klasse

6.3 Integration des Snapshot Managers in die MARS Microservice Umgebung

Zurzeit wird die MARS Microservice Umgebung aufgeräumt, da durch studentische Arbeiten viele sehr spezielle Microservices vorhanden sind, wurde das Kubernetes Cluster unübersichtlich. Um dieses Problem zu beseitigen, werden mehrere Services zusammengefasst oder entfernt. Im selben Zuge wurde auch überlegt, wie man den Snapshot Manager in das Umfeld implementieren kann.

Der “Sim-Runner-Service” hat die Aufgaben eine Simulationsausführung zu erstellen, starten, stoppen oder Auskunft über “Simulation Plans” und Simulationsausführungen zu erteilen. Mit einer Zusammenlegung von “Sim-Runner-Service” und dem neuen Snapshot Manager ist die Verwaltung der Simulationsausführungen in einem Microservice. Der neue Microservice heißt “Sim-Manager-Service” und ersetzt den bestehenden “Sim-Runner-Service” restlos.

Somit ist die Funktionalität eine Simulationsausführung zu starten schon vorhanden (siehe F13), jedoch wird der Neustart nach Ausfallerkennung im Prototyp noch nicht umgesetzt.

6.3.1 Refaktorisierung

Des Weiteren hat die Zusammenführung den Nebeneffekt, dass der alte Code des “Sim-Runner-Service” noch einmal überprüft wird. Im Zuge, dessen wird der Code refaktorisiert, denn er stammt aus einer frühen Go Version, ist schlecht lesbar und es fehlen Tests.

Ordnerstruktur

In Abbildung 6.3.1 ist die Ordnerstruktur vorher und nachher zu sehen. Vorher der Refaktorisierung waren alle Ordner und Dateien im Hauptordner des Git Repositories. Auch die “Main” Klasse war in diesem Hauptordner. Nach der Refaktorisierung wurden die verschiedenen Ordner besser getrennt. Zunächst ist die Implementation in einem extra Ordner (“sim-manager-svc”) und die Implementation (“pkg”) ist von den Startdateien (“cmd”) getrennt [1]. Des Weiteren sind die Clients für andere MARS Services in einem extra Ordner (“serviceclients”).

[1] siehe https://talks.golang.org/2014/organizeio.slide#9

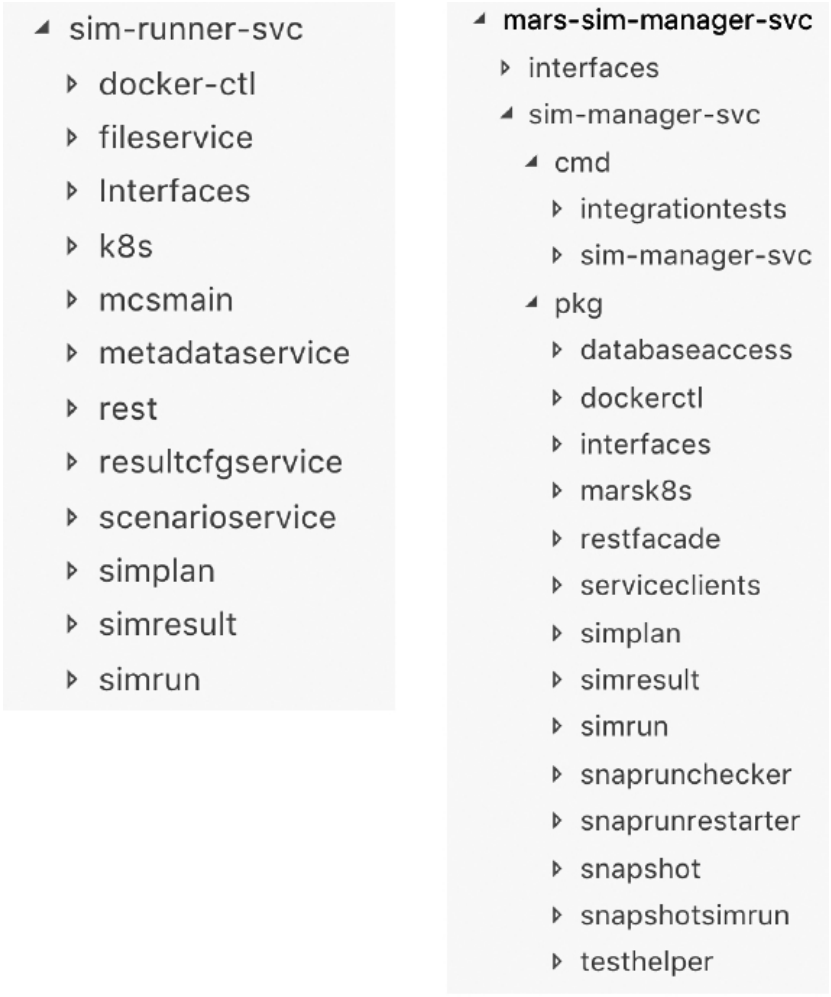

Abbildung 6.1: Vergleich der Ordnerstrukturen vor und nach Refaktorisierung

Zwei Representationen der Simulationsausführung

Wie in der Ordnerstruktur des refaktorisierten "Sim-Manager-Service" zu sehen, gibt es die Ordner "simrun" und "snapshotsimrun". Sowohl der "Sim-Runner-Service" als auch der Snapshot Manager verwenden beide eine Entität "Simulation Run" die, die Simulationsausführungen repräsentiert. Die Unterschiede liegen in den verwendeten Instanzvariablen.

Es gibt die Möglichkeiten entweder die bestehende Entität zu verändern oder die neue Entität einen anderen Namen zugeben. Ich entscheide mich für letzteres, da die Änderung einer Entität eine Überarbeitung der REST Schnittstelle zur Folge hätte. Im MARS Team wurden früher leichtfertig Schnittstellen geändert, die oft zu einer starken Behinderung des Systems führte. Um nicht leichtfertig zu handeln, müssten zuerst einige Tests geschrieben werden, um sicher zu gehen, dass die Änderungen der Entität keine Änderung des Verhaltens der REST Schnittstelle nach sich zieht. Die Zeit meiner Bachelorarbeit will ich jedoch lieber für den Snapshot Manager verwenden. Als Name wird "snapshotsimrun" verwendet, da während der Bachelorarbeit der vorgestellte Mechanismus intern "Snapshoting" heißt und es klar sein soll, dass die Entität Daten für das "Snapshoting" speichert.

REST Schnittstelle

Der "Sim-Runner-Service" besitzt eine REST Schnittstelle. Bevor die Schnittstelle um die Funktionalität des Snapshot Managers erweitert wird, wird auch diese refaktorisiert. Nötig ist dies vor allem mit Blick auf die Länge der Funktionen. "HandleSimulationPlan" ist 183 Zeilen und "HandleSimulationRun" ist 280 Zeilen lang. Der Hintergrund ist, dass diese Funktionen alle HTTP Verben eines Pfades abdecken.

Statt in Pfade wird die REST Schnittstelle zukünftig in "REST Routen" unterteilt, eine "REST Route" besteht aus einem Namen (zur Identifikation), einer "Methode" (das HTTP Verb), einem "Pattern" (dem Pfand) und dem "Handler" (der sich um die Anfrage kümmert). Somit entstehen kleinere Funktionen, da nicht mehr eine Funktion alle ausgewählten HTTP Verben eines Pfades bedienen muss, sondern sich auf ein HTTP Verb konzentriert. Der "Handler" nutzt für die Bearbeitung der Anfrage, stets "Controller" Klassen der benötigten Komponente, zu sehen in Abbildung 6.2 (siehe auch Facade Pattern).

Die Funktion "HandleSimulationRun" wird somit in sechs Funktionen geteilt, diese sind durchschnittlich 58 Zeilen lang was eine deutlich bessere Übersicht über die Funktionen ermöglicht. Zusätzlich sind je Komponente eigene Dateien für die benötigten REST Routen vorhanden, dies führt eine deutliche Trennung der Routen von dem Server nach sich. Um "REST Routen"

wie beschrieben verwenden zu können wird die externe Bibliothek “github.com/gorilla/mux” verwendet.

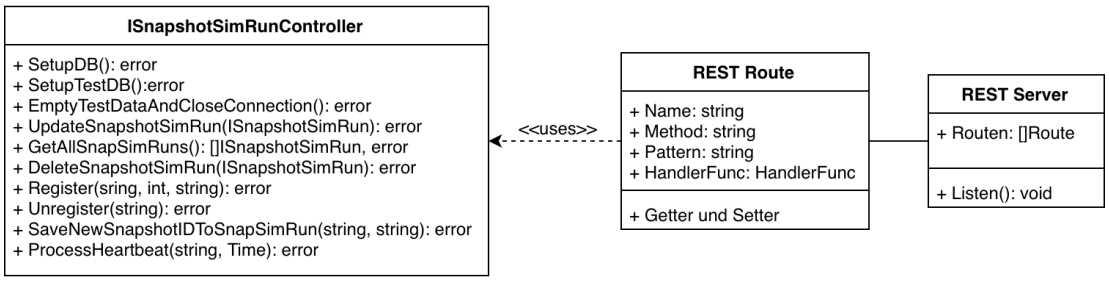

Abbildung 6.2: Auszug aus Abbildung 6.5 für REST Beispiel

6.4 Gewählte Programmiersprache

Das MARS Team möchte neben der Anzahl von Microservices auch die Anzahl der verwendeten Programmiersprachen verringern. So bestehen als Möglichkeiten aktuell Go und C#. Wie erwähnt ist der bestehende Code in Go implementiert. Um großen Mehraufwand durch eine Re-Implementierung in eine neue Sprache zu meiden wird auch der Snapshot Manager in Go implementiert. Aktuell ist Go in Version 1.11.

Go besitzt die Möglichkeit funktional programmiert zu werden. Aus diesem Grund werden alle Klassen die keine Instanz benötigen (zum Beispiel Factory Klassen) funktional programmiert und sind deswegen in den folgenden Klassendiagrammen nicht enthalten.

6.5 Gewählte Datenbank

Für die Datenbank wurde von dem MARS Team MongoDB vorgegeben. Die MongoDB ist ein eigener Microservice innerhalb der MARS Umgebung. Zu gunsten der Performanz wurde die MongoDB fragmentiert (“Sharding”).

Eine MongoDB ist in “Databases”, “Collections” und “Documents” strukturiert. Wie in der Refaktorisierung erwähnt, wird der Mechanismus Team intern “Snapshoting” genannt, aus diesem Grund heißt die von “Sim-Manager-Service” und Simulationsausführung verwendete “Database” “Snapshoting”. Die einzelnen Entitäten werden in verschiedene “Collections” gespeichert, zum Beispiel werden “SnapshotSimRuns” in der “SimRunData” “Collection” gespeichert.

Die einzelnen Datensätze sind dann in den jeweiligen “Documents” und sind anhand ihres “_id” Attributes eindeutig identifizierbar.

Hierbei sind die genanten Konstanten pro Entität beliebig anpassbar solang eine MongoDB verwendet wird. Ein Beispiel ist in Listing 6.4 anhand der “SnapshotSimRunDatabaseAccess” Klasse zu sehen.

```
1 const snapSimRunMongoDatabaseURL = "localhost"
2 const snapSimRunMongoDatabasePort = 27017
3 const snapSimRunMongoDatabaseName = "Snapshoting"
4 const snapSimRunMongoDatabaseCollectionName = "SimRunData"
```

Listing 6.4: Beispiel für die Individualisierung der MongoDB anhand von “SnapshotSimRunDatabaseAccess”

6.6 Konstanten

Die in Kapitel 4 spezifizierten Konstanten werden “public” zu den jeweiligen Schnittstellen geschrieben. Die Konstanten sind “public” und statisch, so dass jeder darauf zugreifen könnte. Implementiert sind die Konstanten bei dem Interface der Klasse die, die Konstanten am meisten benötigt. Zum Beispiel anhand der “SnapRunChecker” Klasse die zur Prüfung der Daten sowohl den Heartbeat Timeout, das Prüfungsintervall als auch die maximale Anzahl erlaubter Snapshots braucht. Dabei basiert der Heartbeat Timeout auf dem Heartbeat Intervall, weshalb dieser dort auch spezifiziert wird.

```
1 type ISnapRunChecker interface {
2         StartContinousChecking()
3 }
4
5 const CheckingIntervallInSeconds = time.Duration(10 * time.Second)
6 const HeartbeatIntervallInSeconds = time.Second * 10
7 const HeartbeatTimeoutInSeconds = HeartbeatIntervallInSeconds * 3
8 const MaximalSnapshotCount = 2
```

Listing 6.5: Konstanten der “SnapRunChecker” Schnittstelle

6.7 Entwurfsmuster

Die folgenden Entwurfsmuster (auch Pattern genannt) werden eingesetzt:

6.7.1 Factory Pattern

Für die Erzeugung von Entitäten wird das Factory Pattern eingesetzt. Mit dem Factory Pattern wird die Konstruktion eines Objektes abstrahiert. Wird ein neues Objekt benötigt, wird somit die Funktion der Factory Klasse statt dem Objektkonstruktor direkt aufgerufen. In Verbindung mit den erwähnten Interfaces, wird somit stets die eigentliche Implementation abstrahiert. In Abbildung 6.5 sind die Factories nicht enthalten, da sie keine Klassen sind (siehe 6.4).

6.7.2 Dependency Injection

Dependency Injection wird eingesetzt, wenn ein Objekt zur Erstellung ein Objekt einer anderen Komponente benötigt. In Verbindung mit Interfaces sind somit keine Abhängigkeiten zu Implementierungen anderer Komponenten nötig. Hinzu kommt, dass in den Komponenten kein Wissen vorhanden sein muss, wie Objekte der anderen Komponente erstellt werden (siehe auch Factory Pattern). Hierbei handelt es sich um "Pure" Dependency Injection, da keine Dependency Injection Container verwendet werden.

Aufgrund der Möglichkeit für Tests benötigte Objekte kontrolliert zu erstellen und mitzugeben, erleichtert Dependency Injection "Test Driven Development".

6.7.3 Facade Pattern

Die erwähnte REST Schnittstelle arbeitet nicht direkt mit den Entitäten, sondern mit einer Fassade (auch Facade Pattern genannt). Die Fassade kapselt alle Funktionalitäten der unterliegenden Komponente, um den Umgang mit diesen zu vereinfachen. In den Abbildungen 6.3 und 6.5 haben die Fassaden Klassen "Controller" im Namen. Die REST Schnittstelle an sich ist streng genommen auch eine Fassade nach Außen, da diese das System nach außen hin kapselt.

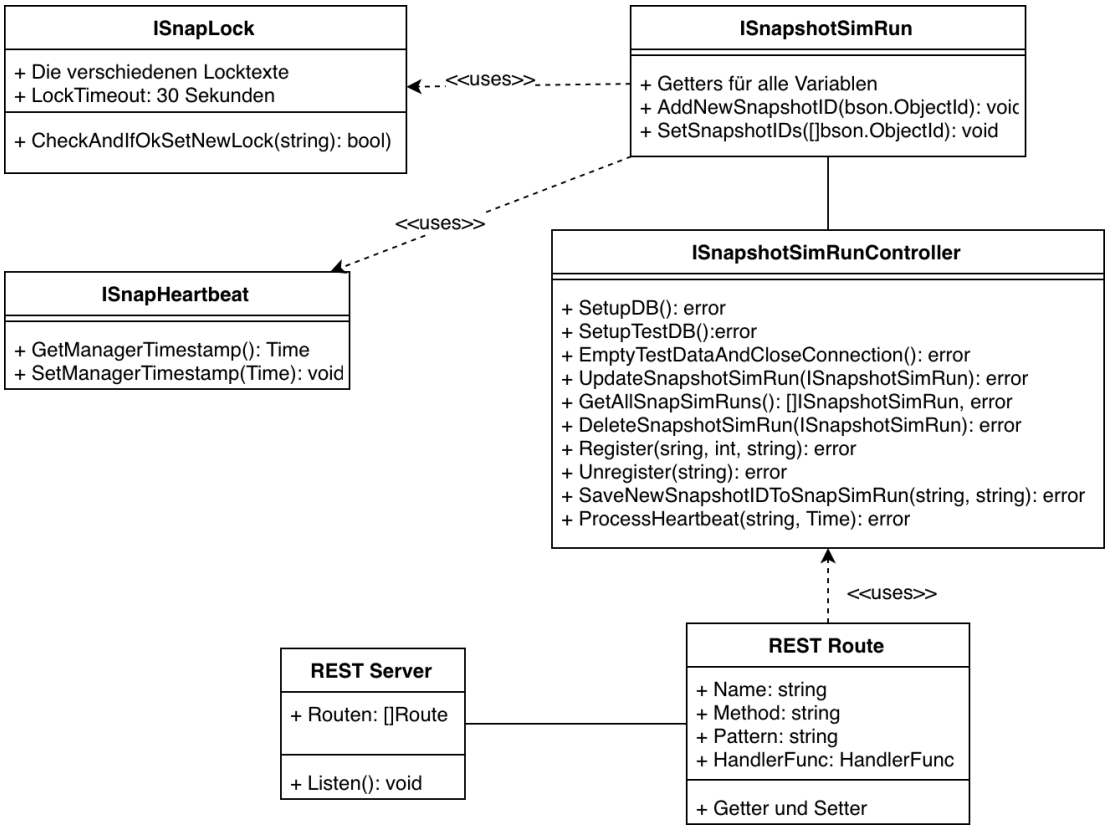

Abbildung 6.3: Auszug aus Abbildung 6.5 für Facade Pattern Beispiel anhand von "ISnapshot-SimRunController"

6.7.4 Composite Root Pattern

Dependency Injection wird zu dem in Verbindung mit dem Composite Root Pattern verwendet, um einmalig im Einstiegspunkt des Programms den Abhängigkeitsgraphen zu erzeugen. Für größere Projekte ist es ratsam ein Framework für die Erstellung des Abhängigkeitsgraphen und für die Dependency Injection zu haben. In dieser Bachelorarbeit sind die Abhängigkeiten noch überschaubar und somit ist die zusätzliche Abhängigkeit und Komplexität eines solchen Frameworks nicht vonnöten [2].

In Listing 6.6 ist zu sehen, wie die jeweiligen "Controller" aller Komponenten und der größte Teil des Abhängigkeitsgraphen erzeugt werden. Die erstellten "Controller" werden danach, wenn benötigt an den REST Server und den "SnapRunChecker" weitergegeben.

[2] siehe "You aren't gonna need it" und "Keep it simple, stupid" [Cachin u. a. (2011)]

```
        newSimRunController := simrun.CreateSimRunController()
newSimPlanController := simplan.CreateSimPlanController(
    newSimRunController)
newSimResultController := simresult.CreateSimResultController()
newSnapshotController := snapshot.CreateSnapshotController()
newSnapRunController := snapshotsimrun.
    CreateSnapshotSimRunController(newSnapshotController)
```

Listing 6.6: Beispiel für die Implementierung des Composite Root Pattern in Verbindung mit Dependency Injection

6.7.5 Adapter Pattern

In jeder Komponente gibt es eine "DatabaseAccess" Klasse, die die Zugriffe auf die Daten kapselt.Die Funktionalität ist bei jeder dieselbe: Lesen, Speichern, Updaten und Löschen von Daten. Aus diesem Grund werden die Funktionalitäten außerhalb der jeweiligen Komponenten mit anonymen Datentypen in einer "GeneralDatabaseAccess" Klasse implementiert. Jede Komponente nutzt ihren eigenen Adapter (Adapter Pattern) für diese Klasse und diese beinhaltet nur die benötigten Funktionalitäten der "GeneralDatabaseAccess" Klasse. Durch diese explizite Limitierung soll deutlich gemacht werden wie die Komponente mit der Datenbank arbeitet, dies soll zukünftigen Entwicklern eine Weiterentwicklung und Wartung erleichtern. In diesem Adapter wird auch von dem anonymen Datentyp in den benötigten Datentyp konvertiert. Auf Basis der Adapter werden dann die "Controller" implementiert. So ist der "Single Point of Control" bezüglich Datenbanken in der "GeneralDatabaseAccess" Klasse und die Adapter haben Komponenten spezifische Spezialisierungen.

In den Abbildungen 6.4 und 6.5 kann man sehen, wie unter anderem die "SnapshotDatabase-Access" Klasse die Methoden der "GeneralDatabaseAccess" nutzt und für die Anforderungen der Snapshot Komponente anpasst.

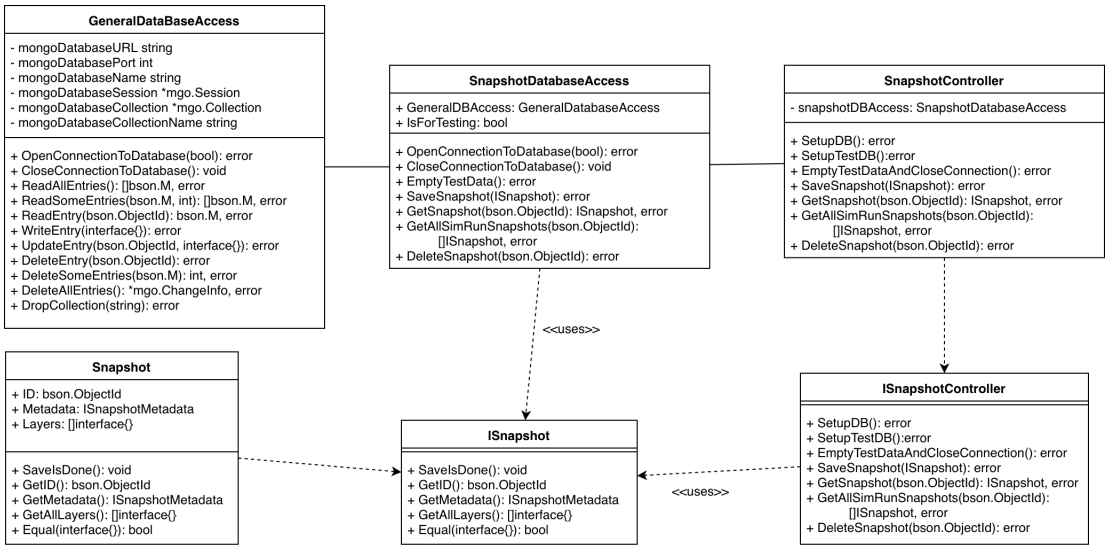

Abbildung 6.4: Auszug aus Abbildung 6.5 für Adapter Pattern Beispiel anhand von “Snapshot-DatabaseAccess”

6.8 Klassendiagramm

Abbildung 6.5 zeigt die Klassen des Snapshot Managers im “Sim-Manager-Service”. Wie in Kapitel **??** erwähnt, ist jede Komponente gleich aufgebaut. Die Schnittstellen nach Außen sind die “REST Routen”. Die Zugriffe auf die Datenbank sind jeweils durch eine “DatabaseAccess” Klasse gekapselt. Ferner sind in den Komponente Factories und Entitäten enthalten.

Farblich getrennt sind die einzelnen Komponenten in Abbildung 6.5 wie folgt:

Farbe	**Komponente**
Blau	“SnapshotSimRun”
Grün	“Snapshot”
Rot	“SnapRunChecker”
Flieder	Datenbankzugriff
Orange	REST Server

Tabelle 6.1: Übersicht über die farbliche Einteilung der Komponenten in Abbildung 6.5

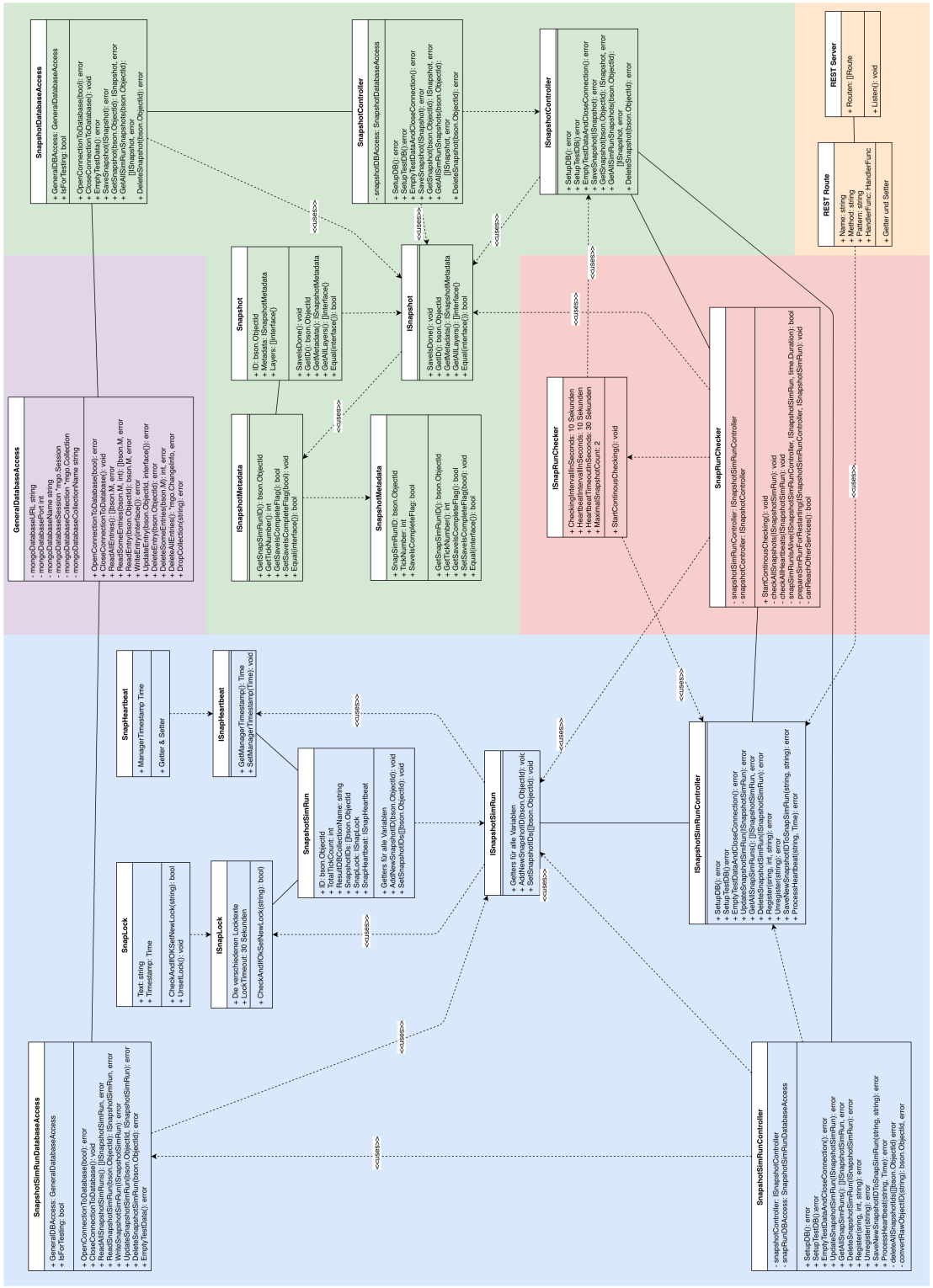

Abbildung 6.5: Technische Klassendiagramm des Snapshot Managers im “Sim-Manager-Service”

7 Evaluation

In diesem Kapitel wird nach Vorstellung des Testkonzepts und der zusätzlichen Experimente geklärt, ob der erstellte Entwurf und der implementierte Code den Anforderungen (Kapitel 4.5 und 4.6) entspricht.

Das zugrunde liegende Testsystem ist ein MacBook Pro (Mitte 2015, macOS Mojave V10.14.2) mit einem 2,2 GHz i7 Prozessor und 16 GB Hauptspeicher. Die Experimente testen nur Funktionalitäten und Code des Snapshot Managers im "Sim-Manager-Service", weswegen hier weiterhin vom Snapshot Manager die Rede ist.

7.1 Testkonzept

Der Code wird mit Unit Tests getestet, hilfreich war hierbei vor allem die Implementierung mit Dependency Injection. Die Unit Tests werden bei jedem Push in das Git Repository in der Gitlab Pipeline ausgeführt. Über diesen Weg werden spätestens nach jedem Commit alle Tests ausgeführt und Fehler fallen direkt auf. Einige Komponenten wurden zudem Test Driven implementiert um noch früher auf Fehler zu stoßen.

Die Testabdeckung des Snapshot Managers Code beträgt 72,18%, die des restlichen Microservices beträgt 23,06% und die des gesamten Microservice beträgt 40,60%. Vor der Zusammenführung (siehe Kapitel 6.3) war die gesamte Testabdeckung bei lediglich 17,04%.

7.2 Experimente

Aus organisatorischen Gründen konnten die Experimente nicht im MARS Kubernetes Cluster ausgeführt werden, weshalb auf eine lokale Ausführung ausgewichen wird.

Lokal kann entweder die komplette MARS Microservice Umgebung kopiert oder nur der wesentliche Teil der Simulationsausführungen nachgebildet werden. Die kommenden Experimente werden auf Basis der Nachbildung (auch "Mock" genannt) ausgeführt, da dies wesentlich schneller zu einem Ergebnis führt. Hinzu kommt, dass die Experimente den neuen, unabhängigen Teil des Simulationsausführungs Codes testet und somit das gesamte Cluster nicht

zwingend notwendig ist. Die lokale Ausführung des kompletten Clusters würde somit viele Ressourcen fruchtlos blockieren und Zeit bis zu den Ergebnissen erheblich vergrößern.

Die Nachbildung verhält sich wie eine Simulationsausführung ohne Tick Berechnung. Das Fehlen der Tick Berechnung ist nicht relevant, da in den Experimenten vor allem die Ausfallerkennung getestet wird, die unabhängig von der Tick Berechnung ist. Die Nachbildung wie auch die richtige Simulationsausführungn sind in C# implementiert und kommunizieren mit dem Snapshot Manager wie definiert über das REST Protokoll.

Jedes Experiment ist in ein Ziel, eine Umsetzung und ein Ergebnis eingeteilt. Das Ergebnis enthält, ob das gesetzte Ziel erfolgreich erreicht werden konnte beziehungsweise warum es nicht erfolgreich war.

7.2.1 Experiment 1 - Intervalle und Timeouts

Ziel

Es soll getestet werden, dass die in Kapitel 4 definierten Intervalle (Heartbeat und Prüfung) und Timeouts (Heartbeats) gesetzt sind. Ferner soll geprüft werden, ob es Probleme gibt, wenn mehrere Snapshot Manager gleichzeitig aktiv sind.

Umsetzung

Es werden drei Snapshot Manager und eine Simulationsausführung gestartet. Die Simulationsausführung meldet sich an und sendet drei Heartbeats. Die Snapshot Manager bearbeiten die Anfragen. 30 Sekunden nach dem Empfang des letzten Heartbeats stellt ein Snapshot Manager den Ausfall fest.

Ergebnis

Das gesetzte Ziel konnte erreicht werden. In dem Listing 7.1 kann gesehen werden ab wann die Simulationsausführung als ausgefallen erkennt werden soll. In Listing 7.2 ist zu sehen, dass der Snapshot Manager der auf Pot 8080 hört, den Ausfall um 09:53 Uhr und 38 Sekunden erkennt. Die optimale Zeit zwischen Ausfall und Erkennung beträgt 30 Sekunden (siehe Heartbeat Timeout), in diesem Fall sind es 9 Sekunden mehr. Die Diskrepanz ist damit zu erklären, dass die Snapshot Manager alle 10 Sekunden prüfen. In diesem Fall wurde eine Sekunde vor dem Ausfall geprüft, somit wird erst wieder 9 Sekunden nach dem Ausfall geprüft. Die Snapshot Manager wurden leicht versetzt gestartet, in einem Produktionsumfeld ist dies durch gegebene Neustarts nicht der Fall, somit würden die Prüfungen über die 10 Sekunden besser verteilt sein und ein Ausfall wird noch eher erkannt.

```
[...]
08:52:59 Sending to URI: http://127.0.0.1:8082/v1/
   snapsimrunheartbeats?simrunid=5c0d098c6905c205395348ce
[...]
08:53:29 SimRunID 5c0d098c6905c205395348ce should be seen dead after
    next check.
[...]
```

Listing 7.1: Auszug aus dem Client Log während Experiment 1

```
[...]
2019/01/15 09:53:38 SnapSimRun with ID ObjectIdHex("5
   c0d098c6905c205395348ce") is pressumbly dead and will be
   restarted
[...]
```

Listing 7.2: Auszug aus einem Server Log während Experiment 1

7.2.2 Experiment 2 - Korrekte Ausfallerkennung

Ziel

Das zweite Experiment hat ein sehr ähnliches Ziel wie Experiment 1, jedoch mit stärkerem Fokus auf den Heartbeat Timeout während der Prüfung. Es soll getestet werden, dass trotz zwei fehlender Heartbeats der Simulationsausführung nicht als Ausfall erkannt wird.

Umsetzung

Der Aufbau ist der gleiche wie bei Experiment 1, im Gegensatz dazu werden allerdings nach dem Senden von drei Heartbeats 20 Sekunden nichts gesendet. Nach diesen 20 Sekunden werden drei weitere Heartbeats gesendet bevor sich die Simulationsausführung abmeldet. Keiner der Snapshot Manager darf die Simulationsausführung als ausgefallen erkannt haben, da erst nach 30 Sekunden ohne Heartbeats ein Ausfall erkannt wird.

Ergebnis

Das gesetzte Ziel konnte erreicht werden. In Listing 7.3 ist zu sehen, dass zwischen zwei Heartbeats 20 Sekunden vergehen. In den Snapshot Managern wurden die fehlenden Heartbeats nicht direkt als Ausfall erkannt, wie in Listing 7.4 zu sehen ist.

```
[...]
09:06:43 Sending to URI: http://127.0.0.1:8080/v1/
   snapsimrunheartbeats?simrunid=5c0d098c6905c205395348ce
09:07:03 SimRunID 5c0d098c6905c205395348ce should not be seen dead
   after next check.
09:07:03 Sending to URI: http://127.0.0.1:8080/v1/
   snapsimrunheartbeats?simrunid=5c0d098c6905c205395348ce
[...]
```

Listing 7.3: Auszug aus dem Client Log während Experiment 2

```
[...]
2019/01/15 10:06:38 Start Checking
2019/01/15 10:06:48 Start Checking
2019/01/15 10:06:58 Start Checking
2019/01/15 10:07:08 Start Checking
[...]
```

Listing 7.4: Auszug aus einem Server Log während Experiment 2

7.2.3 Experiment 3 - Snapshot Manager Ausfall ohne Lock

Ziel

Es soll getestet werden, dass das Gesamtsystem mit dem Ausfall eines Snapshot Managers umgehen kann. In diesem Fall fällt der Snapshot Manager aus, bevor er einen Lock gesetzt hat. Der Client muss die Anfrage erneut senden und es muss ein neuer Snapshot Manager gestartet werden.

Umsetzung

Es starten 3 Snapshot Manager und eine Simulationsausführung. Die Simulationsausführung meldet sich an, sendet 7 Heartbeats und meldet sich ab. Nachdem ein bestimmter Snapshot Manager eine Heartbeat Anfrage bekommen hat, schläft der Thread, bevor der Lock zur Bearbeitung des Heartbeats gesetzt wird und der Prozess dieses Snapshot Managers wird in dieser Zeit manuell beendet. Anschließend wird ein Neuer Snapshot Manager gestartet. Die Datenbank wird manuell ausgelesen, um zu prüfen, ob kein Lock gesetzt wurde. Anhand des Protokolls der Simulationsausführungen ist zu erkennen, ob die Nachricht erneut gesendet wurde. Der Snapshot Manager der auf Port 8082 hört, wird nach Empfang der Heartbeats

und vor setzten des Locks 6 Sekunden, in dieser Zeit wird der Snapshot Manager manuell “interrupted”.

Ergebnis

Das gesetzte Ziel konnte erreicht werden.

7.2.4 Experiment 4 - Snapshot Manager Ausfall mit Lock

Ziel

Wie bei Experiment 3 wird, geprüft, ob das Gesamtsystem den Ausfall eines Snapshot Managers toleriert. In diesem Experiment liegt der Fokus auf dem Timeout für Locks. Ziel ist, dass der gesetzte Lock nach 30 Sekunden ignoriert wird.

Umsetzung

Der Ablauf ist wie bei Experiment 3, jedoch wird der Snapshot Manager Prozess manuell beendet, nachdem dieser den Heartbeat Lock gesetzt hat. Manuell wird, dann in der Datenbank nachgesehen, ob der Lock gesetzt wurde. Mit der ersten Prüfung eines Snapshot Managers nach 30 Sekunden wird dieser Lock ignoriert und ein neuer Lock gesetzt. Der Snapshot Manager, der auf Port hört wird nach empfang des Heartbeats und dem dementsprechenden Locken 10 Sekunden warten. In dieser Zeit kann der Snapshot Manager manuell “interrupted” werden.

Ergebnis

Das gesetzte Ziel konnte erreicht werden. In Listing 7.5 wird beispielhaft der Output der manuellen Datenbankabfrage gezeigt. Darin ist zu erkennen, dass der Datensatz um 09:35 Uhr und 13 Sekunden den “UpdatingHeartbeat” Locktext hält. Während der Lock gesetzt ist können die Snapshot Manager keine weiteren Anfragen bearbeiten, zum Beispiel Heartbeats wie zu sehen in Listing 7.6. 35 Sekunden nach dem setzten des Locks um 09:35 Uhr und 48 Sekunden wurde die Datenbank erneut abgefragt und es ist kein Lock mehr zu sehen (“Alive” Lock entspricht einem freien Lock). Somit ist erfolgreich getestet, dass ein gesetzter Lock nach 30 Sekunden ignoriert wird, in diesem Beispiel entspricht die Zeit zwischen setzen des Locks und ignorieren 35 Sekunden, diese Ungenauigkeit kann zum einen davon kommen, dass nicht exakt nach 30 Sekunden mit den Daten gearbeitet und somit der Lock entfernt wurde, oder durch die leicht zu späte Abfrage der Datenbank.

```
Michaels-MacBook-Pro:mars-sim-manager-svc hapemac$ date && py bh.py
    6 0
Di 15 Jan 2019 09:35:13 CET
{'_id': ObjectId('5c0d098c6905c205395348ce'),'totaltickcount': 100,
    'snapshotids': [], 'snaplock': {'text': 'UpdatingHeartbeat', '
    timestamp': datetime.datetime(2019, 1, 15, 8, 35, 11, 779000)}, '
    snapheartbeat': {'managertimestamp': datetime.datetime(2019, 1,
    15, 8, 34, 58, 741000)}}

[...]

Michaels-MacBook-Pro:mars-sim-manager-svc hapemac$ date && py bh.py
    6 0
Di 15 Jan 2019 09:35:48 CET
{'_id': ObjectId('5c0d098c6905c205395348ce'), 'totaltickcount': 100,
     'snapshotids': [], 'snaplock': {'text': 'Alive', 'timestamp':
    datetime.datetime(1, 1, 1, 0, 0)}, 'snapheartbeat': {'
    managertimestamp': datetime.datetime(2019, 1, 15,8, 35, 44,
    792000)}}
```

Listing 7.5: Manuelle Datenbankabfrage während Experiment 4

```
2019/01/15 09:35:34 Add new Heartbeat
2019/01/15 09:35:34 Error occurred! Err was: Lock could not be set
```

Listing 7.6: Auszug aus einem Server Log während Experiment 4

7.3 Erfüllte Anforderungen

Anhand der Implementation, der Unit Tests und den oben genannten Experimenten wird anschließend geklärt, ob die erstellten Anforderungen aus Kapitel 4.5 und 4.6 erfüllt wurden.

	Beschreibung
F01	Bei dem Start einer Simulationsausführung ist nun festgelegt, dass dieser von Kubernetes nicht neu gestartet werden darf.
F02	Alle Snapshot Manager arbeiten auf derselben Datenbank und halten selbst keine zusätzlichen Daten. Für die synchrone Bearbeitung von Daten wurden Locks eingeführt.

F03	Der Ausfall eines Snapshot Managers führt zu keinem Zeitpunkt zu Datenverlusten, da die Daten erst in die Datenbank geschrieben werden, wenn sie fertig bearbeitet wurden. Anschließend wird der Lock auf die Daten entfernt. Sollte der Snapshot Manager zwischen Speichern und Löschen des Locks ausfallen, ist hierfür der Lock Timeout die Lösung. Bekommt die Simulationsausführung keine oder eine negative Antwort wird die Anfrage erneut gesendet. Über diesen Weg gehen keine Anfragen verloren.
F04	Nach positivem Prüfen eines Heartbeats wird wie spezifiziert geprüft, ob die meisten anderen Services erreicht werden können.
F05	Wie definiert wartet der Snapshot Manager auf eingehende Anmeldungen die Simulationsausführungen, um zu wissen, welche Simulationsausführungen aktiv sind.
F06	Die Simulationsausführung teilt nach Erstellung eines Snapshots dem Snapshot Manager die ID des neuen Snapshots mit.
F07	Anhand des Zeitstempels, welcher speichert, wann der letzte Snapshot erstellt wurde, erkennt die Simulationsausführung, ob nach einem Tick wieder ein Snapshot erstellt werden soll oder nicht.
F08	Der Snapshot Manager erkennt den Ausfall eines Clients 30 Sekunden nach seinem letzten Heartbeat.
F09	Der Heartbeat Intervall von 10 Sekunden wurde im Client implementiert und ist somit erfüllt.
F10	Unter Berücksichtigung der entsprechenden Werte (siehe Kapitel 4.4) sendet die Simulationsausführung eine Anfrage erneut an den Snapshot Manager, falls keine oder eine negative Antwort erhalten wurde.
F11	Bei der Prüfung der Simulationsausführungsdaten werden unvollständige Snapshots gelöscht. Sollten danach noch mehr als 2 Snapshots übrig bleiben, werden bis auf die 2 aktuellsten Snapshot alle gelöscht.
F12	Der Snapshot Manager prüft, alle 30 Sekunden die Simulationsausführungsdaten. In dieser Prüfung wird festgestellt, ob eine Simulationsausführung ausgefallen ist und ob Snapshots gelöscht werden müssen.
F13	Durch die Zusammenführung des Snapshot Managers und des “Sim-Runner-Services”, besteht die Möglichkeit Simulationsausführungen neu zu starten.
F14	Der Snapshot Manager berücksichtigt bei der Überprüfung nur angemeldete Simulationsausführungen.

F15	Kann die Simulationsausführung beim Senden der neuen Snapshot ID auch nach dem 5. Mal keinen der Snapshot Manager erreichen, hat aber noch eine Verbindung zur Result Datebank, fährt die Simulationsausführung mit der Berechnung der Ticks fort.
F16	Ist ein gesetzter Lock vor genau oder mehr als 30 Sekunden gesetzt worden, wird dieser ignoriert.
NF01	Im Kubernetes Deployment des Snapshot Manager ist festgesetzt, dass 3 Repliken gestartet werden sollen.
NF02	Alle Schnittstellen des Snapshot Managers wurden idempotent entworfen und erstellt.

Tabelle 7.1: Übersicht über alle Anforderungen

8 Zusammenfassung

In diesem letzten Kapitel wird die Bachelorarbeit abgerundet, mit einem Fazit über das Zurückliegende und dem Ausblick auf das Zukünftige.

8.1 Fazit

Zu Beginn dieser Bachelorarbeit wurde die Frage gestellt, wie die Ausfallsicherheit für zustandsbehaftete Simulationsausführungen im MARS Kubernetes Cluster mithilfe von Checkpoints und Heartbeats erhöht werden kann, sodass bei einem Ausfall kein Simulationsfortschritt verloren geht. Mithilfe von Unit Tests und Experimenten konnte eine valide Lösung für diese Fragestellung gefunden werden.

Aufgrund fehlender Funktionalität der Kubernetes eigenen Ausfallerkennung, musste ein neuer Microservice erstellt werden. Es wurden Use Cases erstellt, die beschreiben, welche neuen Anforderungen an den Snapshot Manager und die Simulationsausführung bestehen. Eine Simulationsausführung meldet sich nach dem Start nun bei dem Snapshot Manager an und sendet danach stetig Heartbeats. Während der Tick Berechnung erstellt die Simulationsausführung nach jedem Tick Ergebnisse und regelmäßig Snapshots. Sind alle Ticks berechnet, meldet sich die Simulationsausführung beim Snapshot Manager ab.

Parallel prüft der Snapshot Manager die angemeldeten Simulationsausführungen auf ihre Heartbeats und löscht alte und unvollständige Snapshots. Aufgrund der parallelen Ausführung mehrerer Snapshot Manager und der gemeinsamen Datenbank, mussten Locks für die Datensätze eingeführt werden. Die Locks haben den Zweck zu steuern, welche Snapshot Manager mit welchen Daten arbeiten dürfen.

8.2 Ausblick

Auf Basis dieser Bachelorarbeit kann in Zukunft auf die folgenden Probleme eingegangen werden:

8.2.1 Neustart und Weltzustände

Die prototypische Implementation ist ohne Neustart und Zustandswiederherstellung (siehe Kapitel 1.2). In zukünftigen Arbeiten könnte dies implementiert werden. Hier zu muss nach einer positiven Ausfallerkennung ein Neustart durchgeführt werden, des Weiteren muss der Snapshot den Weltzustand der Simulationsausführung beinhalten aus dem der Zustand nach Neustart wiederhergestellt werden kann.

8.2.2 Snapshot Manager Uhren

Bei Empfang eines Heartbeats erstellt der Snapshot Manager einen Zeitstempel. Anhand dieses Zeitstempels prüfen die Manager den Zustand der Simulationsausführungen. Gegebenenfalls ist der Ersteller und der Prüfer des Heartbeat Zeitstempels nicht derselbe Snapshot Manager. Es ist davon auszugehen, dass zwei Snapshot Manager unterschiedliche Uhren haben die für eine möglichst exakte Prüfung zuerst synchronisiert werden müssten [Tanenbaum und Steen (2008)].

8.2.3 Verbesserung der Ausfallerkennung

In zukünftigen Versionen des Services könnte die Ausfallerkennung deutlich erhöht werden in dem die Idee des FALCON Systems eingesetzt wird (siehe Kapitel 2.1). Eine Überwachung mehrerer Ebenen (Applikation, Container, Host, Netzwerk) ermöglicht eine genauere Fehlerkennung und genaueren Umgang mit den Fehlern.

8.2.4 Testen der spezifizierten Konstanten

Die festgestellten Werte der Konstanen im Kapitel 4.4 könnten geprüft werden, ob ein höherer oder niedriger Wert besser ist. Die Überprüfung könnte so gestaltet sein, dass man die Werte verändert und dann verschiedene Szenarien durchgeht. Nach einem Szenario wird geprüft, wie viele der Ausfälle richtig und wie viele falsch erkannt wurden. Anhand dieser Zahlen kann dann die Veränderung bewertet werden. Das Ziel ist es möglichst optimale Werte je Szenario zu finden.

Literaturverzeichnis

[Aguilera u. a. 1997] Aguilera, Marcos K. ; Chen, Wei ; Kawazoe, Marcos ; Wei, Aguilera ; Toueg, Sam: *Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication.* 1997

[Andress 2014] Andress, Jason: *The Basics of Information Security: Understanding the Fundamentals of InfoSec in Theory and Practice.* 2nd. Syngress Publishing, 2014. – ISBN 9780128008126

[Augsten 2017] Augsten, Stephan: *Was sind Docker-Container?* April 2017. – URL https://www.dev-insider.de/was-sind-docker-container-a-597762. – Zugriffsdatum: 04.11.2018

[Avizienis u. a. 2004] Avizienis, Algirdas ; Laprie, Jean-Claude ; Randell, Brian ; Landwehr, Carl: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE Trans. Dependable Secur. Comput.* 1 (2004), Januar, Nr. 1, S. 11–33. – URL http://dx.doi.org/10.1109/TDSC.2004.2. – ISSN 1545-5971

[Braden 1989] Braden, R.: *Requirements for Internet Hosts - Communication Layers.* Internet Requests for Comments. Oktober 1989. – URL https://www.ietf.org/rfc/rfc1122.txt. – Zugriffsdatum: 15.10.2018

[Burrows 2006] Burrows, Mike: The Chubby Lock Service for Loosely-coupled Distributed Systems. In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation.* Berkeley, CA, USA : USENIX Association, 2006 (OSDI '06), S. 335–350. – URL http://dl.acm.org/citation.cfm?id=1298455.1298487. – ISBN 1-931971-47-1

[Cachin u. a. 2011] Cachin, Christian ; Guerraoui, Rachid ; Rodrigues, Luis: *Introduction to Reliable and Secure Distributed Programming.* Springer, 2011. – URL http://infoscience.epfl.ch/record/208942

[Düllmann und van Hoorn 2017] Düllmann, Thomas F. ; Hoorn, André van: Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. New York, NY, USA : ACM, 2017 (ICPE '17 Companion), S. 171–172. – URL http://doi.acm.org/10.1145/3053600.3053627. – ISBN 978-1-4503-4899-7

[Fielding u. a. 1999] Fielding, Roy T. ; Gettys, James ; Mogul, Jeffrey C. ; Nielsen, Henrik F. ; Masinter, Larry ; Leach, Paul J. ; Berners-Lee, Tim: *Hypertext Transfer Protocol – HTTP/1.1*. Internet Requests for Comments. Juni 1999. – URL https://www.ietf.org/rfc/rfc2616.txt. – Zugriffsdatum: 09.10.2018

[Fowler 2014] Fowler, Martin: *Microservices - a definition of this new architectural term*. März 2014. – URL https://martinfowler.com/articles/microservices.html. – Zugriffsdatum: 27.11.2018

[Ghemawat u. a. 2003] Ghemawat, Sanjay ; Gobioff, Howard ; Leung, Shun-Tak: The Google File System. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2003 (SOSP '03), S. 29–43. – URL http://doi.acm.org/10.1145/945445.945450. – ISBN 1-58113-757-5

[Gil und Díaz-Heredero 2018] Gil, David G. ; Díaz-Heredero, Rubén A.: A Microservices Experience in the Banking Industry. In: *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*. New York, NY, USA : ACM, 2018 (ECSA '18), S. 13:1–13:2. – URL http://doi.acm.org/10.1145/3241403.3241418. – ISBN 978-1-4503-6483-6

[Grošelj 1991] Grošelj, Bojan: Fault-tolerant Distributed Simulation. In: *Proceedings of the 23rd Conference on Winter Simulation*. Washington, DC, USA : IEEE Computer Society, 1991 (WSC '91), S. 637–641. – URL http://dl.acm.org/citation.cfm?id=304238.304332. – ISBN 0-7803-0181-1

[Haerder und Reuter 1983] Haerder, Theo ; Reuter, Andreas: Principles of Transaction-oriented Database Recovery. In: *ACM Comput. Surv.* 15 (1983), Dezember, Nr. 4, S. 287–317. – URL http://doi.acm.org/10.1145/289.291. – ISSN 0360-0300

[Hasselbring 2016] Hasselbring, Wilhelm: Microservices for Scalability: Keynote Talk Abstract. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance*

Engineering. New York, NY, USA : ACM, 2016 (ICPE '16), S. 133–134. – URL http://doi.acm.org/10.1145/2851553.2858659. – ISBN 978-1-4503-4080-9

[Heinrich u. a. 2017] Heinrich, Robert ; Hoorn, André van ; Knoche, Holger ; Li, Fei ; Lwakatare, Lucy E. ; Pahl, Claus ; Schulte, Stefan ; Wettinger, Johannes: Performance Engineering for Microservices: Research Challenges and Directions. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ACM, 2017 (ICPE '17 Companion), S. 223–226. – URL http://doi.acm.org/10.1145/3053600.3053653. – ISBN 978-1-4503-4899-7

[Hüning u. a. 2016] Hüning, Christian ; Adebahr, Mitja ; Thiel-Clemen, Thomas ; Dalski, Jan ; Lenfers, Ulfia ; Grundmann, Lukas: Modeling & Simulation As a Service with the Massive Multi-agent System MARS. In: *Proceedings of the Agent-Directed Simulation Symposium*. San Diego, CA, USA : Society for Computer Simulation International, 2016 (ADS '16), S. 1:1–1:8. – URL http://dl.acm.org/citation.cfm?id=2972193.2972194. – ISBN 978-1-5108-2315-0

[Isard u. a. 2007] Isard, Michael ; Budiu, Mihai ; Yu, Yuan ; Birrell, Andrew ; Fetterly, Dennis: Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA : ACM, 2007 (EuroSys '07), S. 59–72. – URL http://doi.acm.org/10.1145/1272996.1273005. – ISBN 978-1-59593-636-3

[Kubernetes 2018a] Kubernetes: *Configure Liveness and Readiness Probes*. Juli 2018. – URL https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes. – Zugriffsdatum: 09.10.2018

[Kubernetes 2018b] Kubernetes: *Deployments*. Oktober 2018. – URL https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. – Zugriffsdatum: 15.10.2018

[Kubernetes 2018c] Kubernetes: *Kubernetes API Overview*. Mai 2018. – URL https://kubernetes.io/docs/reference/using-api/. – Zugriffsdatum: 19.10.2018

[Kubernetes 2018d] Kubernetes: *Pod Lifecycle*. December 2018. – URL https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. – Zugriffsdatum: 12.12.2018

[Kubernetes 2018e] Kubernetes: *Pod Overview*. November 2018. – URL https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. – Zugriffsdatum: 23.12.2018

[Kubernetes 2018f] Kubernetes: *ReplicaSet*. August 2018. – URL https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/. – Zugriffsdatum: 15.10.2018

[Kubernetes 2018g] Kubernetes: *ReplicationController*. September 2018. – URL https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/. – Zugriffsdatum: 15.10.2018

[Kubernetes 2018h] Kubernetes: *StatefulSets*. November 2018. – URL https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. – Zugriffsdatum: 12.12.2018

[Kubernetes 2018i] Kubernetes: *Viewing Pods and Nodes*. November 2018. – URL https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/. – Zugriffsdatum: 23.12.2018

[Leners u. a. 2011] Leners, Joshua B. ; Wu, Hao ; Hung, Wei-Lun ; Aguilera, Marcos K. ; Walfish, Michael: Detecting Failures in Distributed Systems with the Falcon Spy Network. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2011 (SOSP '11), S. 279–294. – URL http://doi.acm.org/10.1145/2043556.2043583. – ISBN 978-1-4503-0977-6

[Li u. a. 2018] Li, Qiankun ; Yin, Gang ; Wang, Tao ; Yu, Yue: Building a Cloud-Ready Program: A Highly Scalable Implementation Based on Kubernetes. In: *Proceedings of the 2Nd International Conference on Advances in Image Processing*. New York, NY, USA : ACM, 2018 (ICAIP '18), S. 159–164. – URL http://doi.acm.org/10.1145/3239576.3239605. – ISBN 978-1-4503-6460-7

[Lin und Dunham 1997] Lin, Jun-Lin ; Dunham, Margaret H.: A Survey of Distributed Database Checkpointing. In: *Distrib. Parallel Databases* 5 (1997), Juli, Nr. 3, S. 289–319. – URL http://dx.doi.org/10.1023/A:1008689312900. – ISSN 0926-8782

[Luksa 2018] Luksa, Marko: *Kubernetes in Action*. Hanser Verlag, 2018. – ISBN 3446455108

[Martin 2013] Martin, Robert C.: *Clean Code - Refactoring, Patterns, Testen und Techniken für sauberen Code - Deutsche Ausgabe.* Heidelberg : MITP-Verlag, 2013. – ISBN 978-3-826-69638-1

[Mazzara und Meyer 2017] Mazzara, Manuel ; Meyer, Bertrand: *Present and Ulterior Software Engineering.* 1st. Springer Publishing Company, 2017. – ISBN 3319674242, 9783319674247

[Microsoft 2017] Microsoft: *Set Lock Timeout.* September 2017. – URL `https://docs.microsoft.com/de-de/sql/t-sql/statements/set-lock-timeout-transact-sql?view=sql-server-2017`. – Zugriffsdatum: 15.10.2018

[Newman 2015] Newman, Sam: *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, 2015. – ISBN 1491950358

[Oracle 2017] Oracle: *Oracle TimesTen In-Memory Database API Reference Guide Release 7.0.* September 2017. – URL `http://download.oracle.com/otn_hosted_doc/timesten/703/TimesTen-Documentation/tt_ref.pdf`

[Oracle 2018] Oracle: *Technical Reference - LOCKTIMEOUT.* 2018. – URL `https://docs.oracle.com/cd/E57185_01/ESBTR/locktimeout.html`. – Zugriffsdatum: 15.10.2018

[Sandoval 2017] Sandoval, Kristopher: *Defining Stateful vs Stateless Web Services.* Mai 2017. – URL `https://nordicapis.com/defining-stateful-vs-stateless-web-services/`. – Zugriffsdatum: 11.12.2018

[Tan u. a. 2012] Tan, Jiaqi ; Kavulya, Soila ; Gandhi, Rajeev ; Narasimhan, Priya: Lightweight Black-box Failure Detection for Distributed Systems. In: *Proceedings of the 2012 Workshop on Management of Big Data Systems.* New York, NY, USA : ACM, 2012 (MBDS '12), S. 13–18. – URL `http://doi.acm.org/10.1145/2378356.2378360`. – ISBN 978-1-4503-1752-8

[Tanenbaum und Steen 2008] Tanenbaum, Andrew S. ; Steen, Maarten v.: *Verteilte Systeme - Prinzipien und Paradigmen.* Pearson Studium, 2008. – ISBN 978-3-827-37293-2

Glossar

Interface: Synonym und englische Übersetzung von einer Schnittstelle die die eigentliche Implementation versteckt, Seite 43

Mock: Nachbildung eines bestimmten Objektes oder Verhalten., Seite 56

Package: Ein Package ist ein eigener Namensbereich und enthält Code, in Objektorientierten Programmiersprachen meistens Klassen, Seite 43

Pattern: Synonym und englische Übersetzung von Entwurfsmuster, Seite 49

Simulation Run: Entität innerhalb von MARS, Synonym und englische Übersetzung von Simulationsausführungen, Seite 13

Anhang

1 Use Case Laufzeitdiagramme

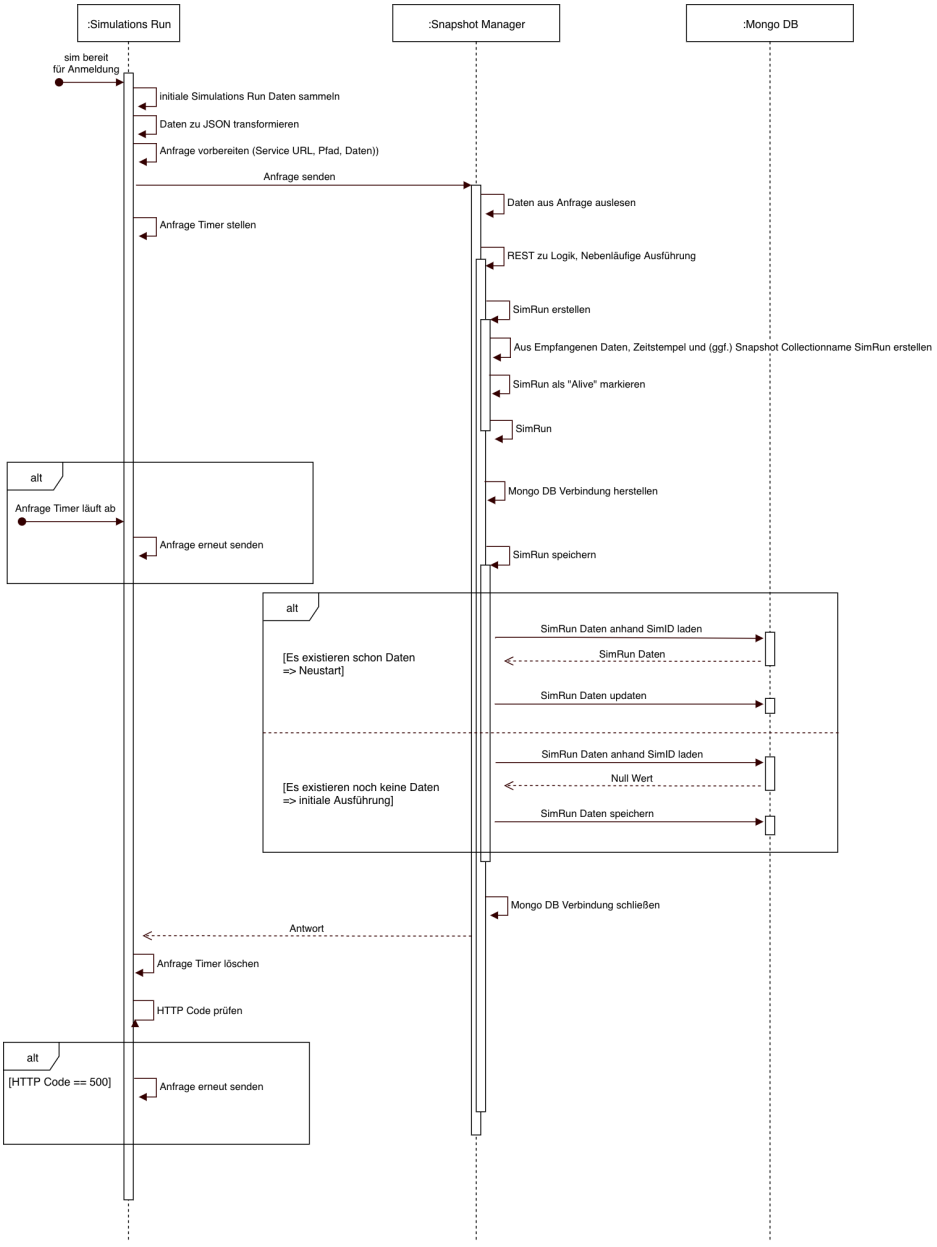

Abbildung 1: Laufzeitsicht von UC01

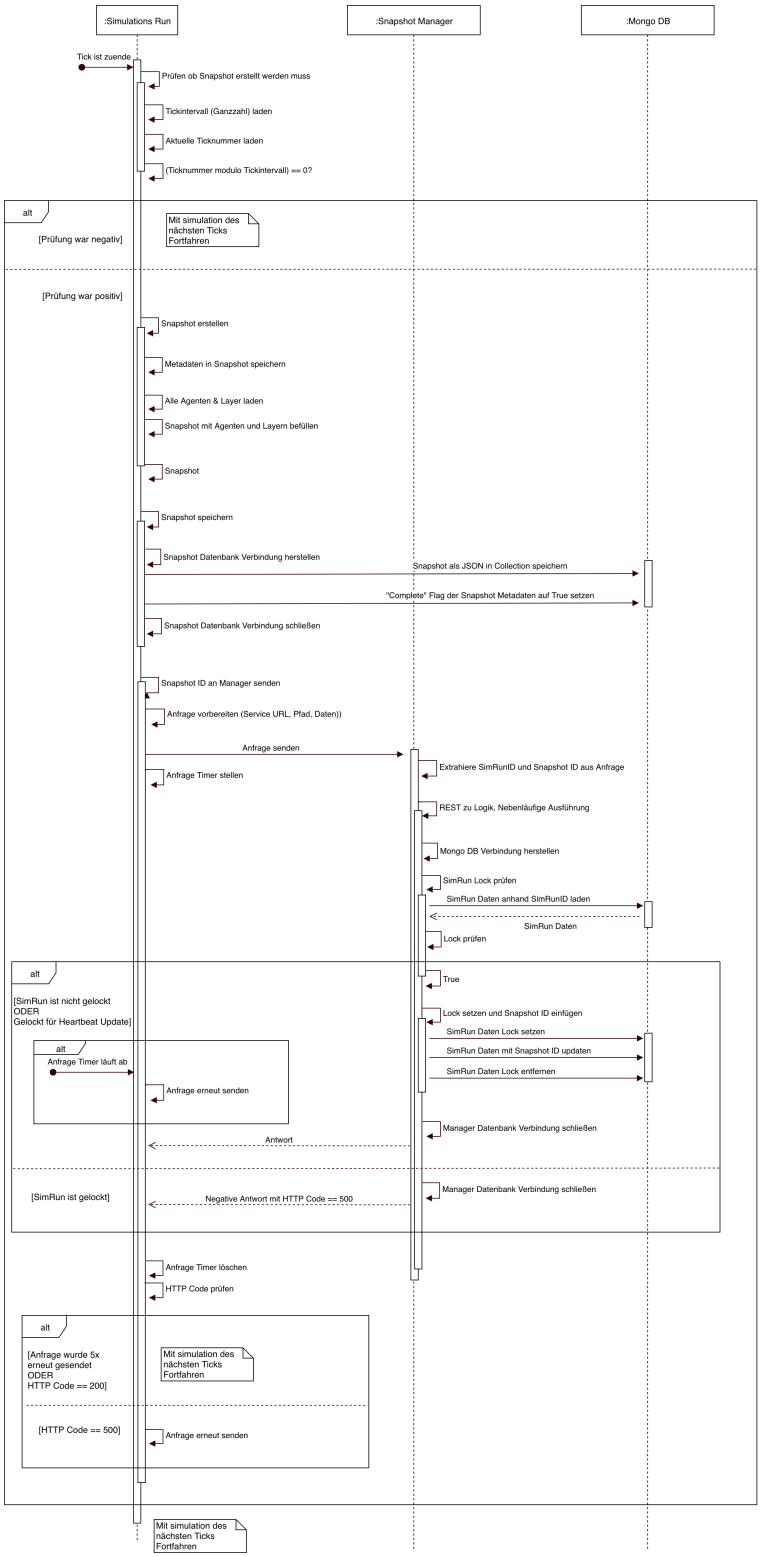

Abbildung 2: Laufzeitsicht von UC02

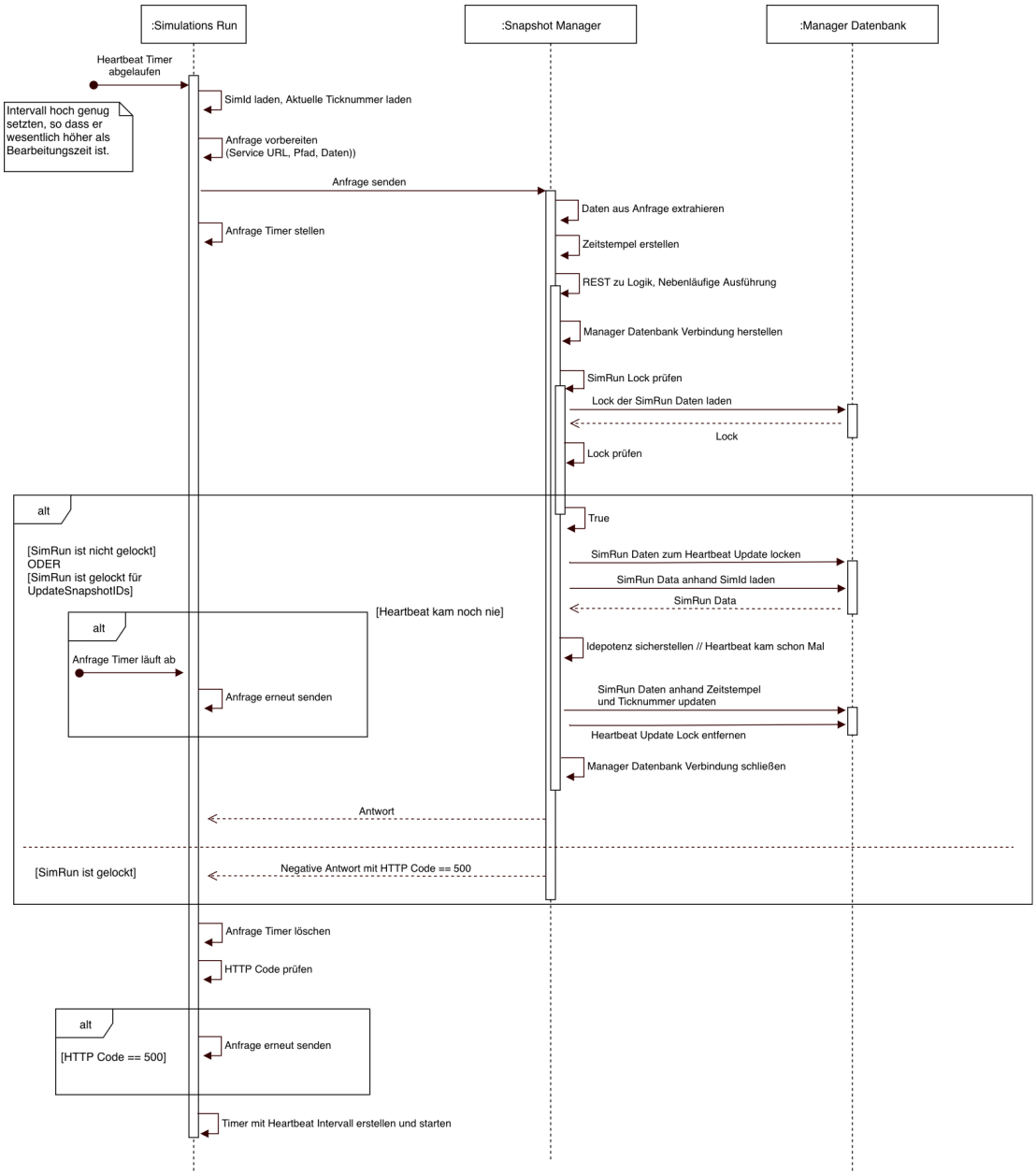

Abbildung 3: Laufzeitsicht von UC03

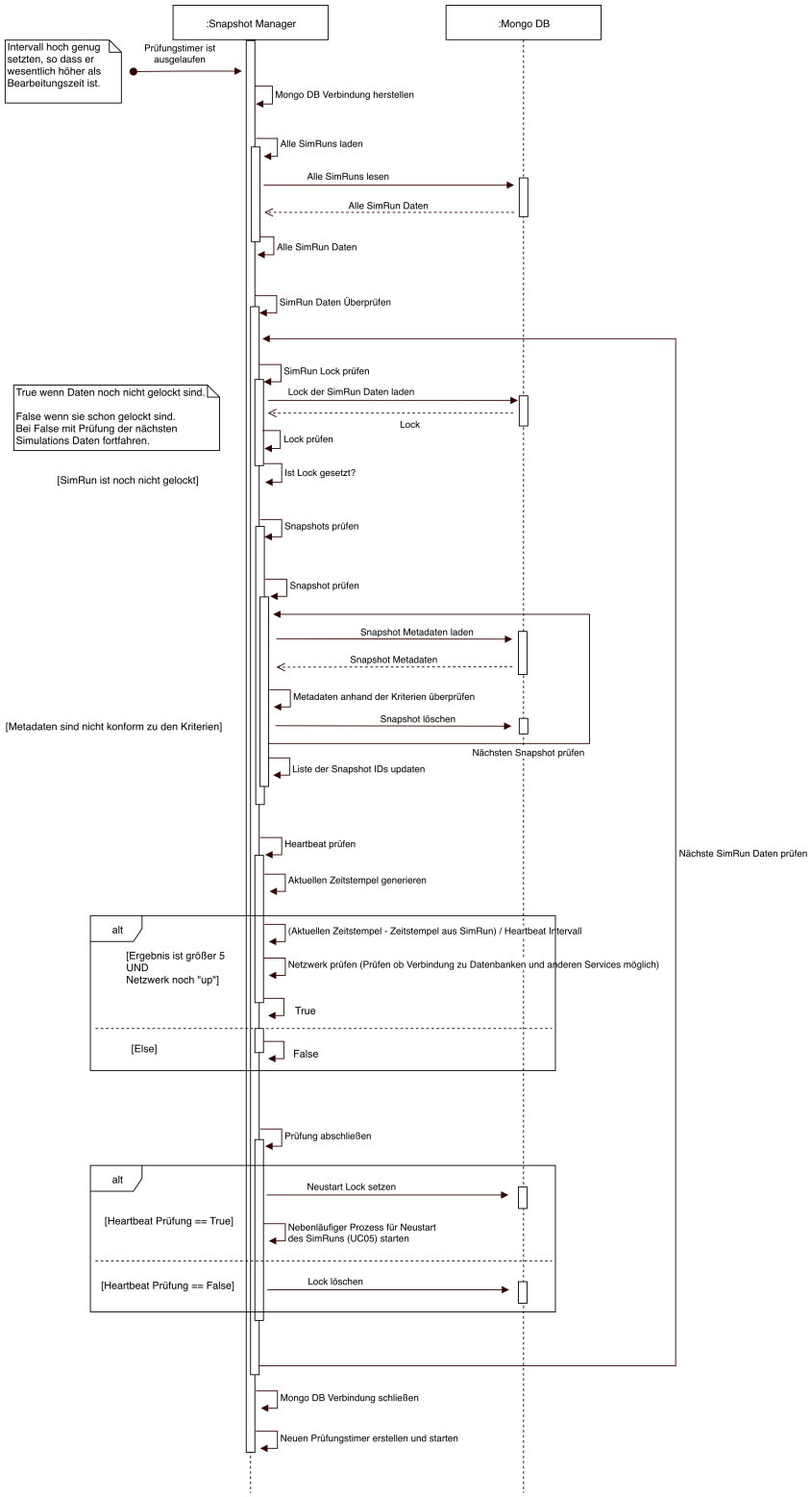

Abbildung 4: Laufzeitsicht von UC04.1

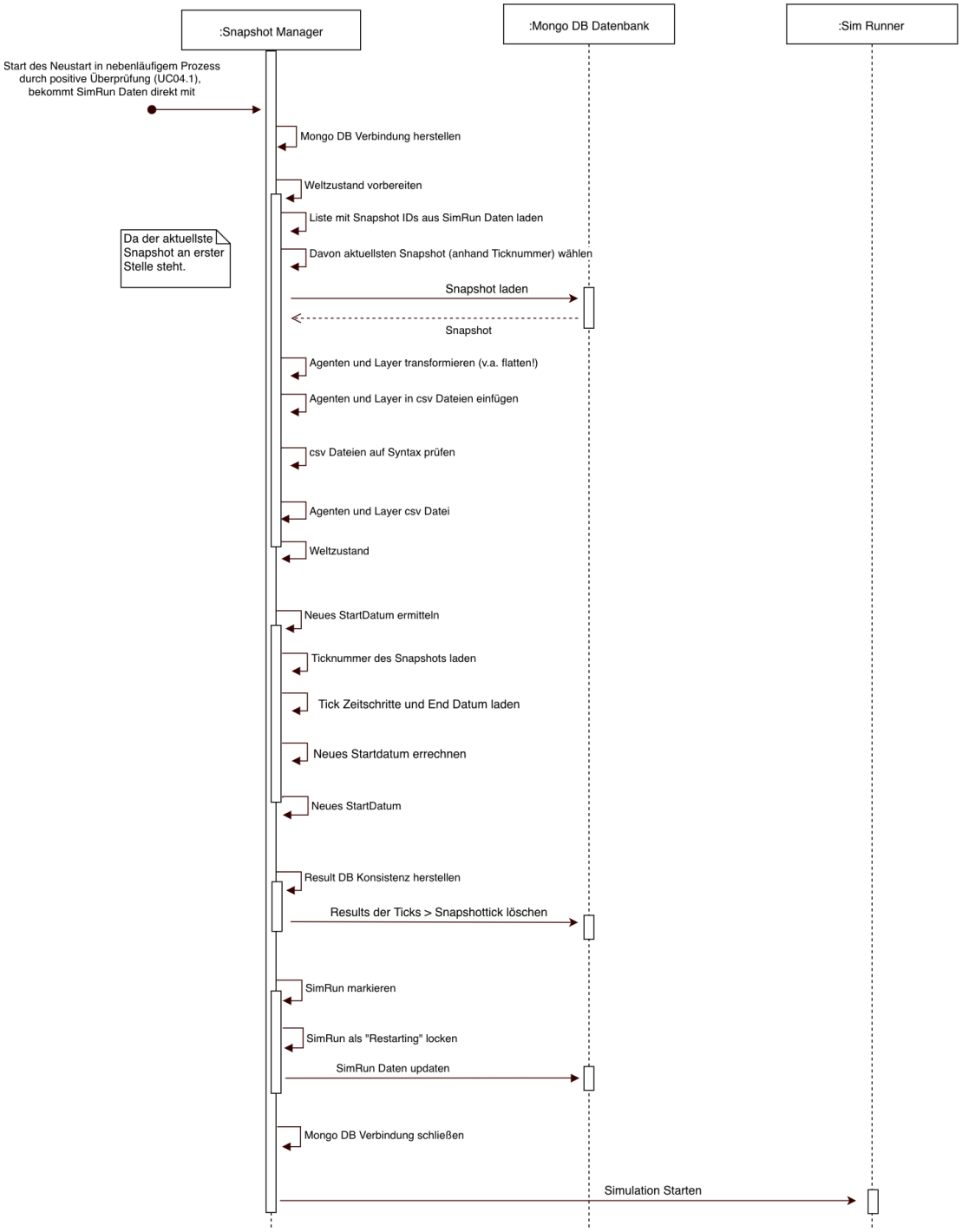

Abbildung 5: Laufzeitsicht von UC04.2

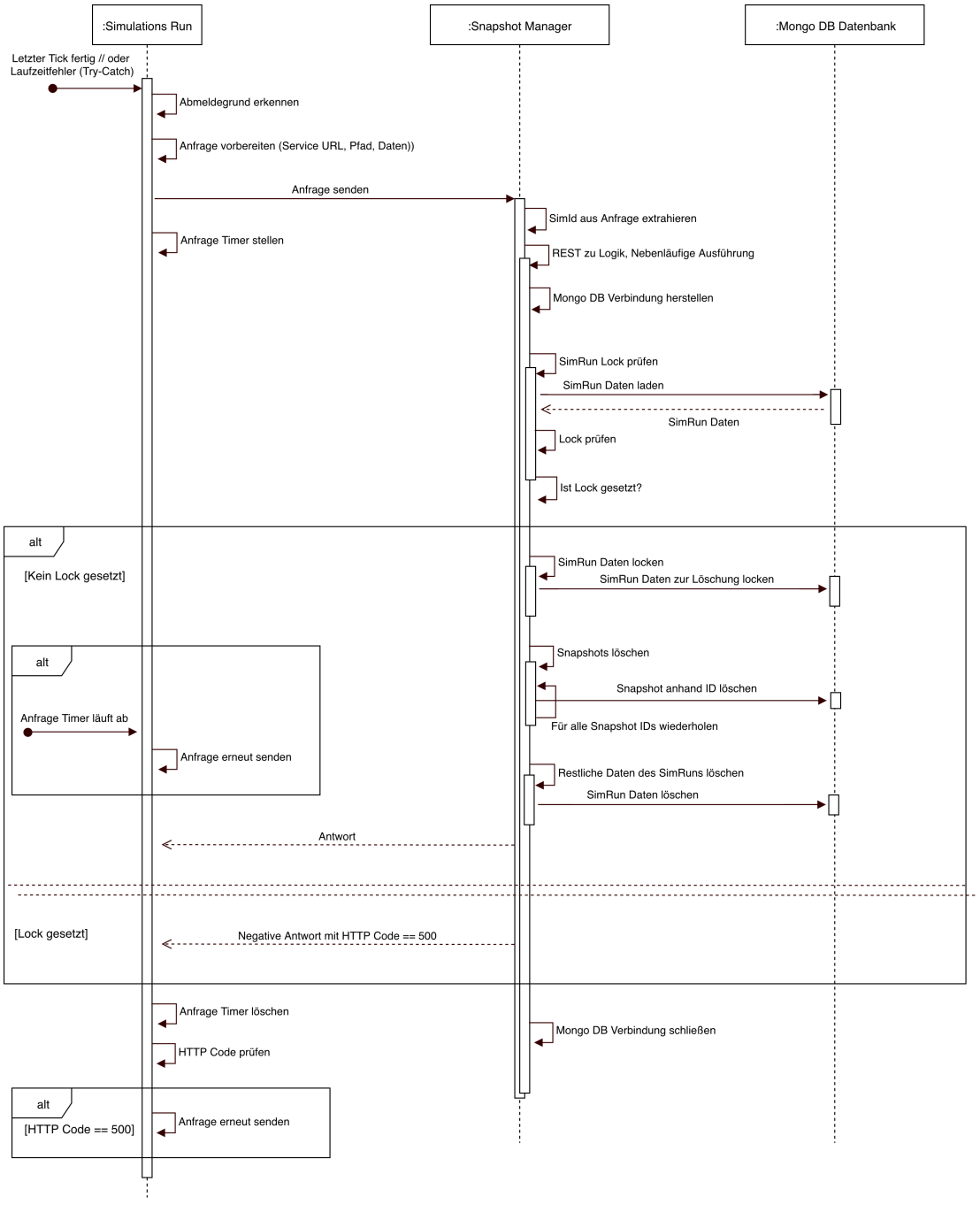

Abbildung 6: Laufzeitsicht von UC05

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15. Januar 2019 Michael Müller