QoS-Aware Configuration of Auto-Scalers

Praktikum: Werkzeuge für Agile Modellierung

Niko Benkler

18. Juli 2019

an der Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Betreuender Mitarbeiter: M.Sc. Manuel Gotin

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einführung

Der Aufstieg des Cloud Computing ermöglicht Kunden ihre Applikationen dynamisch zu skalieren. Dies steigert nicht nur die Verfügbarkeit für den Endnutzer, sondern hat auch ökologische und ökonomische Vorteile. Investitionen in Hardware und Software können reduziert werden, da die Ressourcennutzung durch das dynamische Skalieren optimiert wird. Gleichzeitig kann der Klimafußabdruck reduziert werden, da Ressourcen nicht unnötig brach liegen [1] [2].
Trotz dieser Vorteile ist es eine nicht triviale Aufgabe, die korrekte Allokation von Ressourcen zu planen, vorherzusagen und auszuführen. Gründe dafür sind die verschiedenen Charakteristiken der Workloads im Cloud Computing, wie beispielsweise Varianz, Periodizität (Muster, die sich täglich, wöchentlich etc. wiederholen), Art der benötigten Ressource oder Änderungsrate [3].
Diese Aufgabe wird mittels Auto-Skalierer durchgeführt, die basierend auf der Systemlast, diversen Metriken und Strategien dynamisch Ressourcen hinzufügen oder wieder wegnehmen. Zwischen dem Cloud Service Anbieter und dem Kunden existiert dabei ein Rahmenvertrag (Service Level Agreements oder kurz, *SLAs*), der die Dienstgüte des Cloud Service beschreibt [1]. Dieser Vertrag beschreibt Qualitätsparameter wie Verfügbarkeit, Reaktionszeit des Anbieters, Durchsatz oder die Ausfallrate der Infrastruktur. Der Auto-Skalierer sollte dabei so arbeiten, dass er diese Parameter erfüllt und dabei die Gesamtkosten gering hält, sodass der Anbieter mit seiner angebotenen Leistung Profit erzielen kann.
In den letzten Jahren wurden deswegen verschiedenste Strategien für Auto-Skalierer entwickelt, die diese Aufgabe effizient lösen sollen [2]. Um konkrete Implementierungen dieser Auto-Skalierer zu evaluieren, kann man diese auf realer Infrastruktur und unter einer gegebenen Last testen. Dies jedoch erfordert hohe Kosten, da eine Infrastruktur bereitgestellt und eine Last erzeugt werden muss, um so das Verhalten des Auto-Skalieres zu beobachten. Bei einer Simulation hingegen können diese Kosten größtenteils reduziert werden, da nur Ressourcen für die Durchführung der Simulation benötigt werden.
Um eine solche Simulation durchzuführen muss eine Umgebung geschaffen werden, in der die Infrastruktur eines Cloud-Service Anbieters modelliert wird, um so das Verhalten eines gegebenen Auto-Skalierers zu beobachten. Diese Entwicklung einer solchen Umgebung, auch Testbench für Auto-Skalierer genannt, ist der Bestandteil dieses Praktikums und wird in der folgenden Dokumentation beschrieben.

1.1. Motivation

Bei einer zeit-diskrete Simulationen ist es möglich, das Verhalten eines Auto-Skalierers anhand einer gegebenen diskreten Last auf einer simulierten Infrastruktur zu testen. Dabei kann die diskretisierte Last sowohl auf einer realen Last basieren, die über einen sehr langen Zeitraum gemessen wurde, als auch auf einer generierten Last, die verschiedene, der in der Einführung

beschriebenen Charakteristika aufweist. Es ist daher möglich, den Auto-Skalierer ohne größere Anstrengungen unter verschiedenen Lastbedingungen zu testen. Außerdem soll eine Testbench konfigurierbar sein, sodass auch verschiedene Konfigurationen von Infrastrukturen im Zusammenspiel mit verschiedenen Auto-Skalierern getestet werden können.
Die variable Granularität der diskreten Zeitintervalle ermöglichst er weiterhin, grobgranulare Langzeittests, sowie feingranulare Analysen des Verhaltens durchzuführen. Die dynamisch konfigurierbare Last, Infrastruktur und das Testen beliebiger Auto-Skalierer bekräftigt somit die Implementierung einer solchen Testbench.
Im folgenden Abschnitt wird die grundlegende Idee der Testbench beschrieben.

1.2. Grundlegende Idee

Die Testbench für einen beliebigen Auto-Skalierer ähnelt einer stark vereinfachte Form einer typischen Cloud-Infrastruktur. Schaubild 1.1 skizziert diese. Wie man sehen kann, besteht die Testbench aus drei Hauptkomponenten: Die Warteschlange, die Infrastruktur und eine Möglichkeit zum Aufzeichnen des Verhaltens der jeweiligen Komponenten. Der Auto-Skalierer ist zwar auch Teil der Abbildung, sollte aber austauschbar an die Testbench angeschlossen werden können.

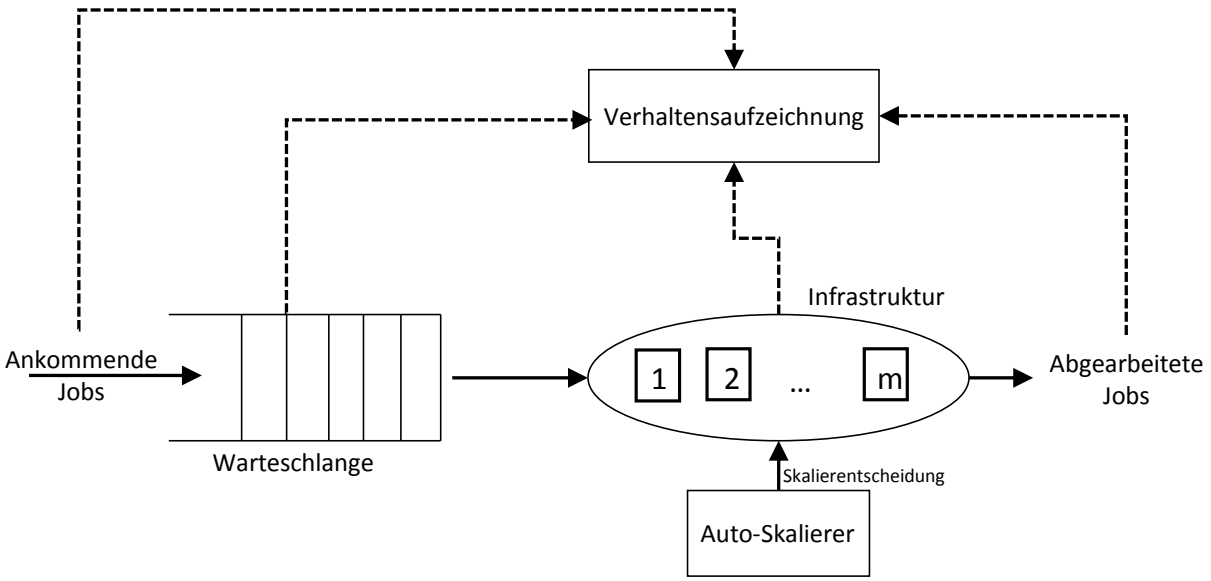

Abbildung 1.1.: Grundlegende Idee der Testbench

Zu einem Zeitpunkt t erreicht eine (durch die gegebene Workload spezifizierte) Menge an Jobs das System und wird in der Warteschlange, die als Puffer fungiert, eingereiht. Die Infrastruktur wird durch eine Menge an Virtuellen Maschinen (*VMs*) repräsentiert. Jede VM ist in der Lage in einer Zeiteinheit eine gewisse Menge an Jobs abzuarbeiten. Die Infrastruktur kann also pro Zeiteinheit eine durch die VMs definierte Menge an Jobs aus der Warteschlange nehmen und verarbeiten. Falls die Menge an ankommenden Jobs größer ist, als die Kapazität der Infrastruktur, so wird die Menge an wartenden Jobs in der Warteschlange größer. Wird die Kapazität

durch eine Skalierentscheidung des Auto-Skalierers erhöht oder sinkt die anliegende Last am System, so reduziert sich diese Menge wieder.
Allgemein ist die Testbench parametrisierbar. Beispielsweise soll die Warteschlangengröße und die Menge an maximal hinzuzufügenden VMs begrenzt sein. Weiterhin ist die Zeit zwischen Ankunft eines Jobs und dessen Abarbeitung in der Realität nicht gleich null. Diese und weitere Parameter können dynamisch konfiguriert werden, sodass nicht nur der Auto-Skalierer unter verschiedenen Lastbedingungen getestet werden kann, sondern auch auf unterschiedlich konfigurierten Infrastrukturen.
Um eine Skalier-Entscheidung treffen zu können, muss der Auto-Skalierer (je nach Typ) diverse Metriken der Infrastruktur oder der Warteschlange erheben, wie beispielsweise Länge der Warteschlange oder Auslastung der VMs. Dafür ist es notwendig, dass alle wichtigen Komponenten diese Metriken zu Verfügung stellen.
Um im Anschluss an eine Simulation den Auto-Skalierer bewerten zu können, wird der Zustand sämtlicher Komponenten zeitdiskret aufgezeichnet. Dabei werden Informationen wie Auslastung des Systems, Füllstand der Warteschlange, Durchsatz oder Wartezeiten im Puffer aufgezeichnet. Mittels dieser Daten kann dann das Verhalten des Auto-Skalierers, und damit seine Güte bestimmt werden.

1.3. Übersicht

Im Kapitel... wird bla beschrieben

2. Architektur und Technologien

Dieses Kapitel beschreibt die verwendeten Technologien und die Architektur der Testbench. Aufgabe ist es ein Kommandozeilen-basiertes Programm zu entwickeln, das basierend auf gegebenen Konfigurations-Dateien die Testbench startet und die Simulation durchführt. Die beobachteten Ergebnisse sollen anschließend wiederum in Dateien geschrieben werden

2.1. Technologien

Die vorgegebene, zu verwendende Programmiersprache ist Java. Weiterhin wird das Java-basierte Spring-Framework[1] verwendet, da es einige nützliche Features wie *Dependency Injection* oder Event-basierte Kommunikation bietet. Als Applikationsserver wird durch Spring implizit eine leichtgewichtige Variante von Tomcat benutzt. Für den Augenblick besitzt der Auto-Skalierer keine Benutzeroberfläche und läuft auf einem lokalen Rechner. Mit Hinblick auf die Zukunft, könnte die Testbench entweder als Web-Applikation gestaltet werden, oder aber eine Benutzeroberfläche erhalten und als Desktop-Anwendung laufen. In beiden Fällen unterstützt das Spring-Framework diesen Evolutionsschritt enorm, sodass beide Varianten mit geringem Aufwand umgesetzt werden können.
Die Konfigurationen für die Testbench liegen im neutralen JSON-Format[2] vor, da dieses sehr einfach eingelesen werden kann. Die Eingabe der Workload erfolgt ebenfalls im JSON-Format. Die Ausgabe-Informationen sind zeit-diskrete, strukturierte Werte weshalb das Tabellen-Format *CSV* zur Speicherung verwendet wird.

2.2. Architektur

Wie in Sektion 1.1 beschrieben besteht die Testbench aus verschiedenen Modulen, die miteinander kommunizieren müssen. Gleichzeitig ist es wünschenswert, die Module lose zu koppeln und Austauschbar zu gestalten. Auch kann eine Informationsquelle bei der Testbench mehrere Informationssenken haben: Der Zustand der Infrastruktur muss periodisch an verschiedenste Komponenten gesendet werden wie zum Beispiel an die Module zur Aufzeichnung der diversen Metriken oder an den Auto-Skalierer. Weitere Informationssenken, die diese Information verarbeiten wollen, sollten dabei ohne größere Veränderung hinzugefügt werden können. Außerdem erzwingt die zeit-diskrete Simulation einen Taktgeber, der periodisch alle Komponenten über den nächsten Taktzyklus informiert. Direkte Kommunikation zwischen den jeweiligen Komponenten hätte dadurch einen nennenswerten Nachteil: Jede Komponente muss alle Komponenten kennen, mit denen sie kommunizieren muss. Im Falle des Taktgebers wären das sogar alle.

[1] https://spring.io/
[2] https://www.json.org/

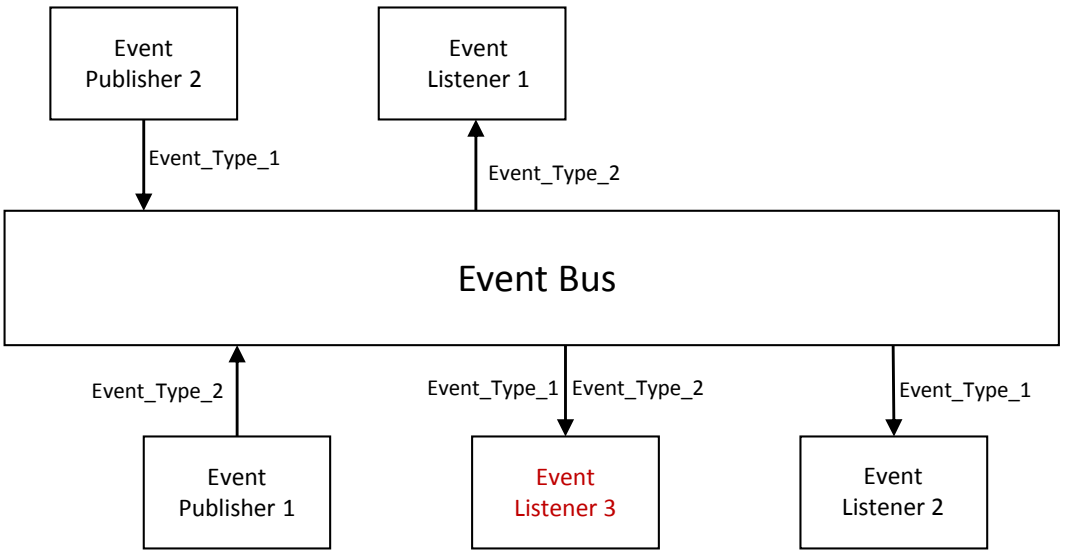

Abbildung 2.1.: Grundlegende Idee der Testbench

Event-basierte Architektur bietet einen Mechanismus, der das Versenden und Empfangen von Nachrichten entkoppelt. Damit ist es möglich die Module selbst zu entkoppeln und Nachrichtenaustausch über einen Event-Bus zu realisieren. Abbildung 2.1 skizziert einen solchen Event Bus. Jede Komponente die eine Nachricht an andere Komponenten versenden möchte benötigt einen *Event Publisher*, der eine (oder mehrere) Nachrichten versendet. Eine Nachricht hat dabei immer einen vordefinierten Typ. Jede Komponente die an dieser Nachricht interessiert ist, muss lediglich einen *Event Listener* implementieren, der auf Nachrichten dieses Event-Typs hört. Falls eine weiteres Modul entwickelt wird, dass an bereits vorhandenen Nachrichtentypen interessiert ist (wie im Schaubild *Event Listener 3*), so muss lediglich ein geeigneter Listener implementiert werden. Die anderen Module, vor allem die bereits implementierte Informationsquelle, muss fabei nicht verändert werden.

3. Aufbau

Dieses Kapitel beschreibt den konkreten Aufbau der Testbench. Wie in Sektion 2.2 beschrieben, ist die verwendete Architektur Event-basiert. Abbildung 3.1 skizziert eine vereinfachte Form der Testbench in einer UML-Klassendiagramm ähnlichen Form. Aus Übersichtsgründen sind Utility-Klassen, Transfer-Objekte, POJO's[1] und die Event-Listener/Event-Publisher der Komponenten nicht Teil des Diagramms. Für jede Komponente, die eine Nachricht in Form eines Events an andere schicken oder von anderen empfangen möchte, existiert ein Listener beziehungsweise ein Publisher. Methoden des Listeners delegieren nach Empfang von Events die erhaltenen Nachrichten (Methodenaufrufe) an ihre nachgeschaltete Komponente. Komponenten die Nachrichten senden wollen, überreichen diese an ihren jeweiligen Publisher, der diese in Form eines Events auf den Event-Bus legt. Näheres ist in Sektion 3.2 erklärt.
Im Folgenden werden alle wichtigen Systemkomponenten erläutert. Anschließend wird ein Überblick über die verschiedenen Event-Typen gegeben. Danach wir der zeitliche, sich periodisch wiederholende Ablauf eines Intervalls in der Testbench vorgestellt.

3.1. Komponenten

Abbildung 3.1 beschreibt den Zusammenhang aller Kern-Komponenten des Auto-Skalierers. Der Event-Bus selbst wird vom Spring-Framework bereitgestellt und ist nicht selbst implementiert. Die *«use»* Bezeichnung soll lediglich verdeutlichen, dass die Komponenten Listener und/oder Publisher vorgeschaltet haben, die mit dem Event-Bus interagieren. Aus Gründen der Übersicht existieren keine Assoziations-Pfeile zwischen dem *ApplicationStartUpRunner* und den Komponenten, die er initialisiert. Die Methodennamen lassen aber darauf schließen, mit welcher Komponente noch eine Assoziation existiert.

3.1.1. Clock

Die *Clock* erfüllt die Aufgabe eines Taktgebers und ist somit das Herz der Testbench. In einer zeit-diskreten Simulation ist es solcher Taktgeber notwendig, da jede Komponenten über den Beginn eines neuen Intervalls informiert werden muss. Dabei gilt: Ein Intervall gilt erst dann als beendet, sobald jede Komponente ihre Aufgaben innerhalb dieses Intervalls erledigt hat. Nach Initialisierung der Komponenten arbeitet die Clock für eine in der Konfiguration definierte Anzahl an Intervallen (Simulationszeit). Dabei wird in jedem Schritt zuerst der Workload, falls notwendig, geändert. Danach versendet die Clock ein Event, dass den Beginn eines neuen Intervalls bekannt gibt. Alle Komponenten werden benachrichtigt und erledigen daraufhin ihre Arbeit, wie Verarbeitung der aktuell anliegenden Workload oder Ausführen von Skalier-Entscheidungen. Abschließend werden *InfrastructureModel* und *QueueModel* durch die Clock

[1] Plain Old Java Objects: Werden bspw. für die JSON-Deserialisierung benutzt

aufgefordert ihren aktuellen Zustand zu publizieren. Basierend darauf führen die Komponenten im nächsten Intervall ihre Aufgaben durch.

3.1.2. Infrastructure Model

Das *InfrastructureModel* bündelt die Haupt-Komponenten der Testbench. Es hat eine Warteschlange für ankommende und wartende Jobs (vgl.3.1.6), eine Komponente die sich um das Hochfahren der Virtuellen Maschinen kümmert (vgl.3.1.2.1) sowie einen gekapselten Zustand (vgl.3.1.3).
Die Information, wie viel Last in welchem Zeitintervall anliegt, wird im *InfrastructureModel* gespeichert und ggf. durch Erhalt einer Workload-Änderung (*changeWorkload()*) geändert. Basierend darauf, ist sie verantwortlich bei Erhalt eines jeden Clock-Ticks (*handleClockTick()*) die erforderliche Menge an Jobs in der Warteschlange (vgl.3.1.6) einzureihen. Danach berechnet das *InfrastructureModel* als Abstraktion der Infrastruktur die vorhandene Kapazität an Jobs, die in diesem Intervall abgearbeitet werden können (abhängig von Anzahl der vorhandenen Virtuellem Maschinen) und nimmt diese Anzahl aus der Warteschlange und verarbeitet sie.
Unregelmäßig muss das Model mit Skalier-Entscheidungen des Auto-Skalieres umgehen (*scaleVirtualMachines()*). Dabei werden hochfahrende Virtuelle Maschinen in der *VMBootingQueue* eingereiht und nach abgelaufener Boot-Zeit mit in die Kapazitäts-Berechnung eingenommen. Das herunterfahren ist einfachheitshalber instantan.
Zuletzt kann der Zustand *(vgl.3.1.3)* über die Methode *publishInfrastructureState()* publiziert werden. Dieser Vorgang wird wie in Sektion3.1.1 beschrieben, von *Clock* angestoßen.

3.1.2.1. VM Booting Queue

Diese Komponente ist eine Warteschlange für hochzufahrende VMs. Nach dem Erhalt einer Skalier-Entscheidung (Hinzufügen einer VM), darf das *InfrastructureModel* diese nicht direkt umsetzen, da eine VM eine gewisse Zeit braucht, um hochzufahren bevor sie aktiv mit in die Kapazitätsberechnung. Deswegen wird eine VM mittels *addVirtualMachineToQueue()* hinzugefügt. In jedem Zeitintervall wird die zu wartende Zeit reduziert und überprüft, ob eine VM bereit ist und zur Menge der aktiv arbeitenden hinzugefügt werden kann *(selectAndRemoveBootedVM())*.

3.1.3. Infrastructure State

Der Zustand der Infrastruktur ist von der Funktionalität abgekapselt. Dieser speichert die aktuell anliegende Workload*(currentArrivalRate)* sowie die aktiven VMs und damit die Kapazität. Der Zustand kann in ein *TransferObject* verpackt und versendet werden. Dieses Objekt beinhaltet zusätzlich Informationen übe die CPU-Auslastung (Diskrepanz zwischen Kapazität und Anzahl der Tasks, die in einem Intervall das System verlassen).

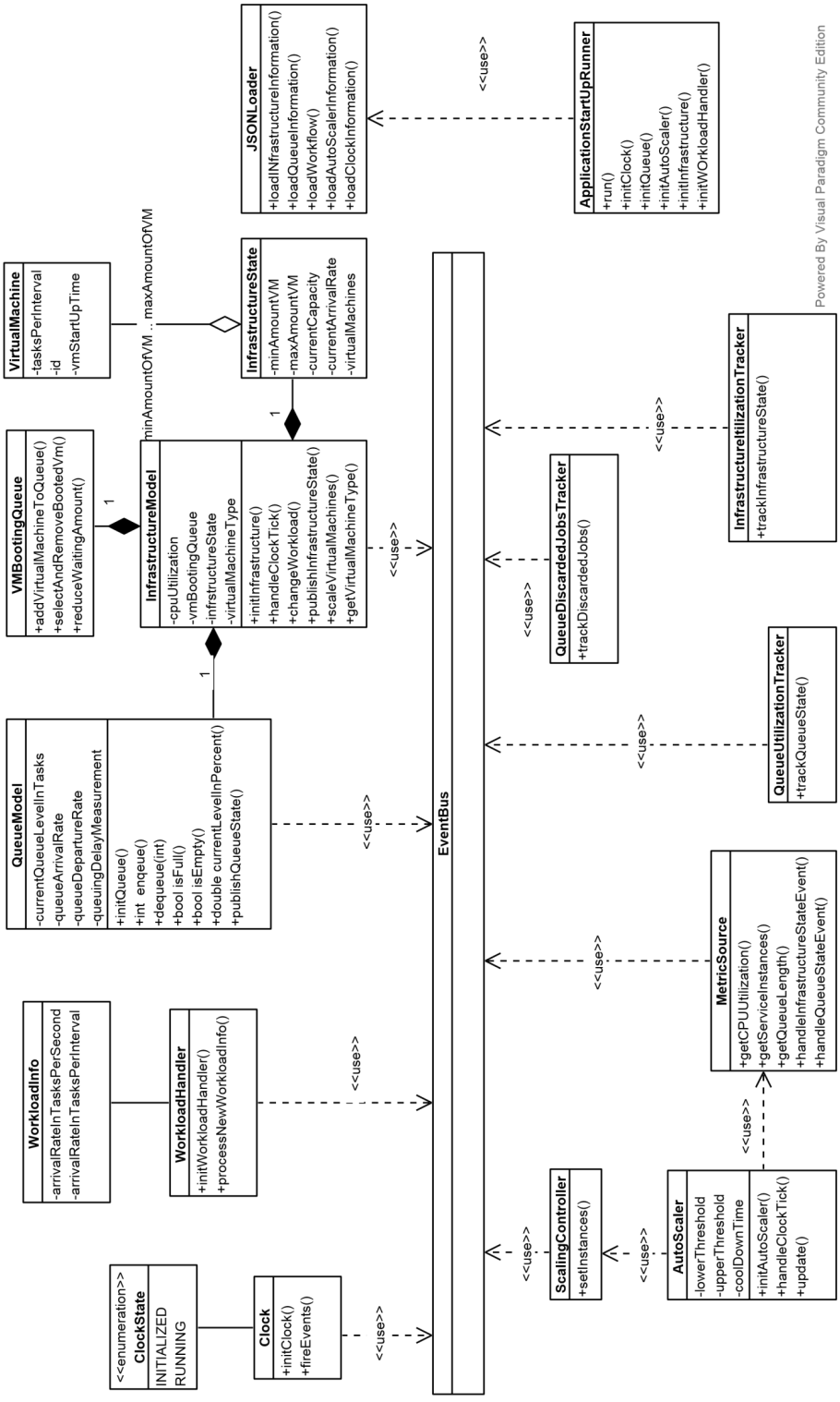

Abbildung 3.1.: Aufbau des Auto-Skalierers

3.1.4. Virtual Machine

Eine Virtuelle Maschine ist eindeutig durch ihre Id identifizierbar. Jede VM benötigt eine vordefinierte Zeit zum hochfahren *vmStartUpTime* und kann pro Zeitintervall eine gewisse Anzahl an Jobs verarbeiten*tasksPerInterval*. Die beiden letzten Werte sind bei dieser Testbench für alle Maschinen pro Simulation gleich, werden also einmalig bei Simulationsstart als Konfigurationsparameter mitgegeben.

3.1.5. Auto Scaler

Der *AutoScaler* als solcher ist kein Teil der Testbench, da er ja das Testobjekt ist. In diesem Fall ist er lediglich hinzugefügt, um die Funktionalität der Testbench zu testen. Ein Auto-Scaler wird angebunden, indem er System-Metriken wie Auslastung oder Warteschlangenlänge über die Komponente *MetricSource* bezieht und Skalierentscheidungen an die Komponente *ScalingController* weiterreicht. Die Interfaces zur Anbindung eines Auto-Skalierers sind in Sektion4.3 beschrieben.

3.1.5.1. Scaling Controller

Der *ScalingController* bietet lediglich eine Schnittstelle für einen Auto-Skalierer, um Skalier-Entscheidungen an die Infrastruktur zu propagieren. Dabei wird der *ScalingMode* zusammen mit den VMs, die entweder hoch- oder heruntergefahren werden sollen, übergeben.

3.1.5.2. Metric Source

Die *MetricSource* bietet eine Schnittstelle für den Auto-Skalierer, um Metriken der Infrastruktur, wie bspw. die CPU-Auslastung oder die Warteschlangenlänge abzugreifen. Diese Metriken liegen immer als moving average vor, wobei das Fenster in den Konfiguration festgelegt werden kann.

3.1.6. Queue Model

Die Warteschlange definiert zwei grundlegende Aufgabe: Zuerst werden alle ankommenden Jobs in der Warteschlange eingereiht. Auch dann, wenn die Infrastruktur nicht völlig ausgelastet ist. Dies liegt an der Natur einer Cloud-Infrastruktur: Ein Job kann nicht in null Zeit ankommen und verarbeitet werden. Diese Verarbeitungszeit wird mit einer Warteschlange simuliert, die als FIFO-Liste aufgebaut ist. Ankommende Jobs können die Warteschlange erst verlassen, wenn sie mindestens für die Warteschlangenverzögerung *(queuing delay)*.
Des Weiteren verhält sich die Warteschlange als Pufferspeicher: Falls die anliegende Last auf dem System größer als dessen Kapazität ist füllt sie sich. Ist es umgekehrt, so leert sie sich wieder. Da Puffer-Speicher in der Realität nicht unendlich groß sind, hat die Warteschlange eine Maximalkapazität. Ist diese erreicht, so werden weitere Jobs verworfen.
Wie auch das *InfrastructureModel* muss die Warteschlange nach einer vordefinierten Zeit ihren Zustand publizieren, sodass *Tracker* und die *MetricSource* diesen auslesen können.

3.1.7. Workload Handler

Der *WorkloadHandler* verarbeitet die zu simulierende Last auf dem System. Diese ist eine Eingabe als Liste von Werten. Der *WorkloadHandler* verarbeitet nach einer frei konfigurierbaren Zeit den nächsten Eintrag und informiert das *InfrastructureModel*, dass sie die Last geändert hat. Damit wird umgangen, dass tatsächlich Last generiert werden muss. Die Infrastruktur bekommt lediglich Informationen darüber, zu welchem Zeitpunkt wie viel Last anliegt.

3.1.7.1. Workload Info

WorkloadInfo ist eine Klasse, die eine Last zu einem bestimmten Zeitpunkt beschreibt.

3.1.8. Tracker

Die verschiedenen *Tracker* zeichnen die Simulation zeit-diskret auf. Die Informationen können anschließend benutzt werden, um den Auto-Skalierer zu beurteilen. Alle aufgezeichneten Parameter werden durch Aufzeichnung der Zustände der jeweiligen Komponenten erhoben. Dabei wird ein gleitender Mittelwert bestimmt, um glattere Ergebnisse zu bekommen.

3.1.8.1. Queue Utilization Tracker

Dieser *Tracker* zeichnet folgende Parameter auf:

- Anzahl an wartenden Jobs
- Füllstand in Prozent, gemessen an der maximalen Länge
- Ankunftsrate
- Verarbeitungsrate (Menge an Jobs, die die Warteschlange verlassen)
- Durchschnittliche Wartezeit in der Warteschlange

3.1.8.2. Queue Discarded Jobs Tracker

Dieser *Tracker* zeichnet folgende Parameter auf:

- Anzahl an verworfenen Jobs Pro Intervall

3.1.8.3. Infrastructure Utilization Tracker

- Ankunftsrate
- Verarbeitungsrate (Menge an Jobs, die das System verlassen)
- Kapazität (Menge an Jobs, die alle VMs verarbeiten können)
- Anzahl der Virtuellen Maschinen
- CPU Auslastung

3.1.9. Application Start Up Runner

Diese Komponente liest die Konfigurationen via *JSONLoader* und initialisiert alle anderen Komponenten. Dies beinhaltet die Umrechnung der externen Einheit (Parameter gegeben in Millisekunden) in die interne Einheit (gegeben in Clock-Ticks). Anschließend startet sie die Simulation.

3.1.10. JSON Loader

Der *JSONLoader* ist eine Klasse zum laden und serialisieren der Konfigurationen im JSON-Format. Jede Konfigurations-Datei wird dabei mittels *Object-Mapper* auf eine Klasse abgebildet.

3.2. Events

Für die Kommunikation zwischen den Komponenten, beziehungsweise um das Auslösen einer Handlung (Methodenaufruf) einer speziellen Komponente anzustoßen, werden Events benutzt. Abbildung 3.2 zeigt, dass alle Events von einem *AbstractEvent* erben, in dem der aktuelle Clock-Zähler und die feste Intervall-Dauer gegeben ist. Diese Informationen sind somit in jedem Event verfügbar, sodass man das Event einem diskreten Zeitintervall zuordnen kann. Im folgenden wird darauf eingegangen, was ein Event absetzt und für was es bestimmt ist.
In dieser Applikation wird mit synchronen Events gearbeitet, das heißt, dass nach Erhalt eines Events durch einen Listener zuerst der gesamte Code auszuführen ist, bevor es an der Stelle weitergeht, an der das Event geworfen wurde. Falls mehr als ein Listener auf ein geworfenes Event reagiert, wird der Code aller Listener-Methoden nacheinander abgearbeitet. Jedoch ist nicht spezifiziert, in welcher Reihenfolge das geschieht, da dies vom Spring-Framework verwaltet wird. Falls also eine Ordnung der Ausführung erzwungen werden soll, müssen zwei verschiedene Event-Typen definiert werden. Diese können dann in der gewünschte Reihenfolge auf den Event-Bus gelegt werden. Die synchrone Abarbeitung garantiert, dass der Code des Listeners, der auf das erste Event hört zuerst vollständig bearbeitet wird.

- *FinishSimulationEvent*
  - **Absender:** *Clock*
  - **Empfänger:** Alle Tracker
  - **Grund:** Erstellen der Ausgabedateien, Schließen der File-Writer
  - **Parameter:** Skalierfaktor, zurückrechnen in Ausgabeformat
- *StartSimulationEvent*
  - **Absender:** *Clock*
  - **Empfänger:** Alle Tracker
  - **Grund:** Beenden des Schreibens in die jeweiligen Dateien

- *ClockEvent*
  - **Absender:** *Clock*
  - **Empfänger:** *InfrastructureModel*, *AutoScaler*
  - **Grund:** Beginn neues Intervall
- *TriggerPublishInfrastructureStateEvent*
  - **Absender:** *Clock*
  - **Empfänger:** *InfrastructureModel*
  - **Grund:** Infrastruktur soll Zustand publizieren
- *TriggerPublishQueueStateEvent*
  - **Absender:** *Clock*
  - **Empfänger:** *QueueModel*
  - **Grund:** Warteschlange soll Zustand publizieren
- *TriggerWorkloadHandlerEvent*
  - **Absender:** *Clock*
  - **Empfänger:** *WorkloadHandler*
  - **Grund:** Neue Workload publizieren
- *ScalingEvent*
  - **Absender:** *ScalingController*
  - **Empfänger:** *InfrastructureModel*
  - **Grund:** Neue Skalier-Entscheidung getroffen
  - **Parameter:** Skalier-Modus und jeweilige VM-Instanzen
- *QueueStateEvent*
  - **Absender:** *QueueModel*
  - **Empfänger:** *QueueTracker*
  - **Grund:** Zustand der Warteschlange soll geschrieben werden
  - **Parameter:** Enthält Infos über Zustand, die in Datei geschrieben werden sollen
- *WorkloadChangedEvent*
  - **Absender:** *WorkloadHandler*
  - **Empfänger:** *InfrastructureModel*
  - **Grund:** Veränderung der Workload
  - **Parameter:** Neue Workload

- *DiscardedJobsEvent*
  - **Absender:** *QueueModel*
  - **Empfänger:** *QueueTracker*
  - **Grund:** Jobs wurden verworfen, da Warteschlange übergelaufen ist
  - **Parameter:** Anzahl der verworfenen Jobs im Zeitintervall
- *InfrastructureStateEvent*
  - **Absender:** *InfrastructureModel*
  - **Empfänger:** *InfrastructureTracker*
  - **Grund:** Zustand der Infrastruktur soll geschrieben werden
  - **Parameter:** Enthält Infos über Zustand, die in Datei geschrieben werden sollen

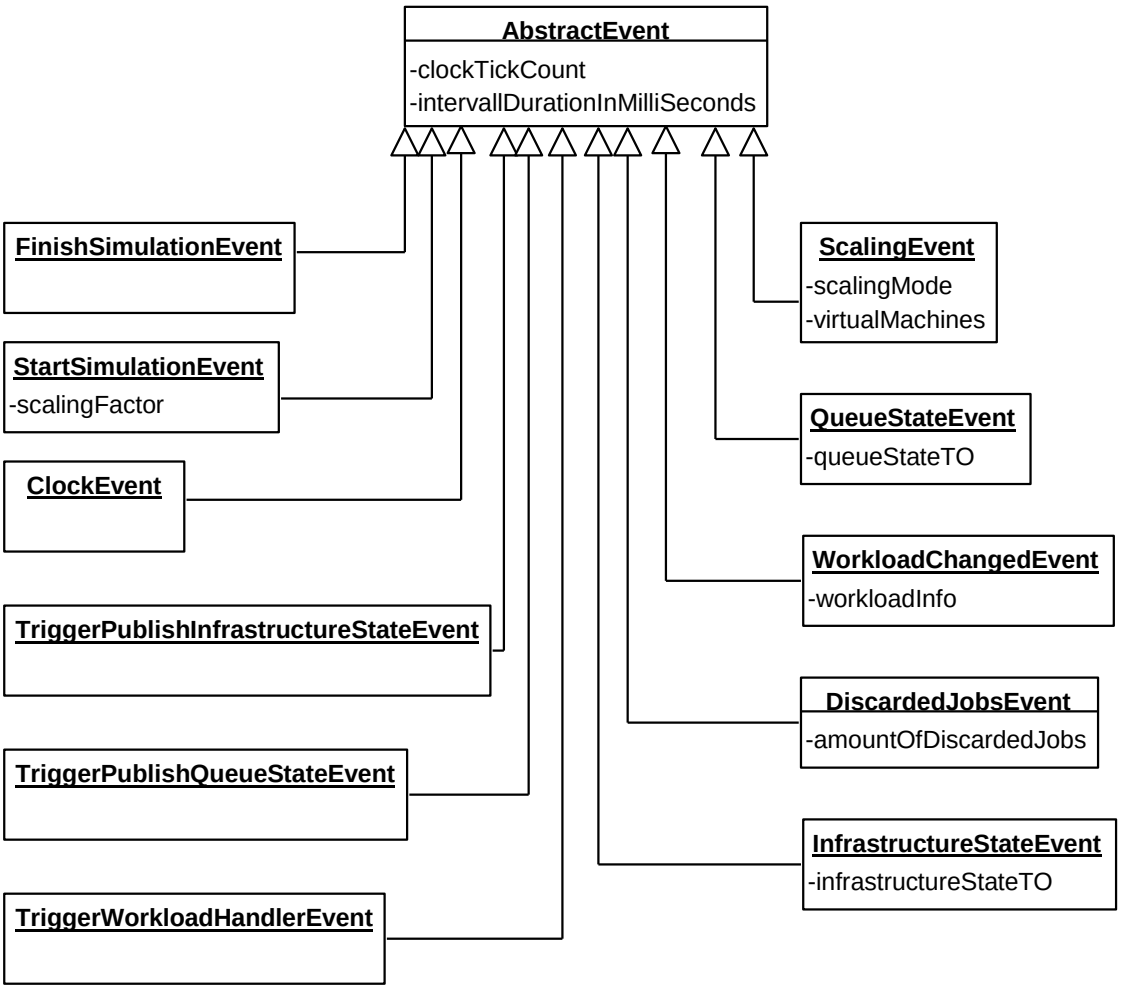

Abbildung 3.2.: Übersicht der Events

3.3. Ablauf

Das Sequenzdiagramm in Abbildung 3.3 verdeutlicht den Ablauf eines Simulationszyklus. Wie in Sektion 3.2 beschrieben, werden die Event synchron empfangen. Falls in der Abbildung ausgehend von einer Komponente zwei mal das Selbe Event direkt hintereinander gesendet wird (bspw. *startSimulation()*), so ist dessen Ausführungsreihenfolge nicht definiert. In der Applikation wird dieses Event nur ein mal auf den Event-Bus gelegt, allerdings von zwei Listener entnommen.
Nachdem die Konfigurationen gelesen und die Komponenten initialisiert wurden, wird die *Clock* gestartet. Zuerst werden die *Tracker* initialisiert, da diese je einen *File-Writer* erstellen, mit denen später die Zustandsinformationen geschrieben werden können. Daraufhin startet die eigentliche Simulation. Die Schleife wird sooft durchlaufen, bis die vorkonfigurierte Simulationszeit verstrichen ist.
Im gesamten Sequenzdiagramm befinden sich mehrere Ausführungen, die nur unter gewissen Bedingung ausgeführt werden. Im folgenden wird der Zyklus ein mal komplett durchlaufen, mit der Prämisse, dass alle Bedingungen erfüllt wären.
Der *WorkloadHandler* wird periodisch, nach einer vorgegebenen Anzahl Zyklen angestoßen. Der neue Workload wird geladen und an das *InfrastructureModel* weitergegeben. Dort wird die alte Workload mit der neuen ersetzt, was bedeutet, dass entweder mehr oder weniger Jobs pro Zeitintervall ankommen (bis sich die Workload wieder ändert).
Auch der *AutoScaler* wird periodisch, nach einer vorgegebenen Anzahl Zyklen angestoßen. Falls er nicht in der CoolDown-Phase ist (Zeit nach einer vergangenen Skalier-Entscheidung, in der er nichts machen soll), holt er sich die benötigten Metriken von der *MetricSource*, wie bspw. CPU-Auslastung. Skalier-Entscheidungen werden dann an die Schnittstelle (*ScalingController*) gegeben, die die Informationen über den Event-Bus an das *InfrastructureModel* sendet. Dort angekommen wird die Skalier-Entscheidung ausgeführt. Im Falle einer Hinzunahme weiterer VMs, werden diese in die *VMBootingQueue* eingereiht. Das Abschalten einer VM funktioniert ohne weiteres Warten.
Sobald das *InfrastructureModel* angestoßen wird, wird überprüft ob neue VMs hochgefahren sind. Ist dies der Fall, so werden diese zur Infrastruktur hinzugefügt und damit die Kapazität erhöht. Anschließend werden, basierend auf der aktuellen Workload, eine Anzahl an Jobs in die Warteschlange eingereiht. Danach werden, basierend auf der gegenwärtigen Kapazität, eine Anzahl an Jobs der Warteschlange entnommen. Sowohl bei dem *QueueModel* als auch bei dem *InfrastructureModel* werden dabei diverse Metriken erhoben und als gleitender Mittelwert gespeichert. Zuletzt wird die Wartezeit noch nicht hochgefahrener VMs dekrementiert, sodass diese ggf. im nächsten Iterationsschritt hinzugefügt werden können.
Im letzten Schritt einer Iterationszyklus, werden *InfrastructureModel* und *QueueModel* aufgefordert, ihren Zustand zu verbreiten. Die Informationen gehen im ersten Fall an den *InfrastructureTracker* und im zweiten Fall an den *QueueTracker*. Beides wird ebenfalls von der *MetricSource* benutzt, um für diverse Metriken, die aus den Zustandsinformationen erhoben werden können, je einen gleitenden Mittelwert zu bilden.
Dieser Zyklus wiederholt sich, bis die Anzahl der durchlaufenen Zyklen der Simulationszeit entspricht. Zuletzt werden noch die *File-Writer* in den *Tracker* geschlossen und die Simulation ist beendet.

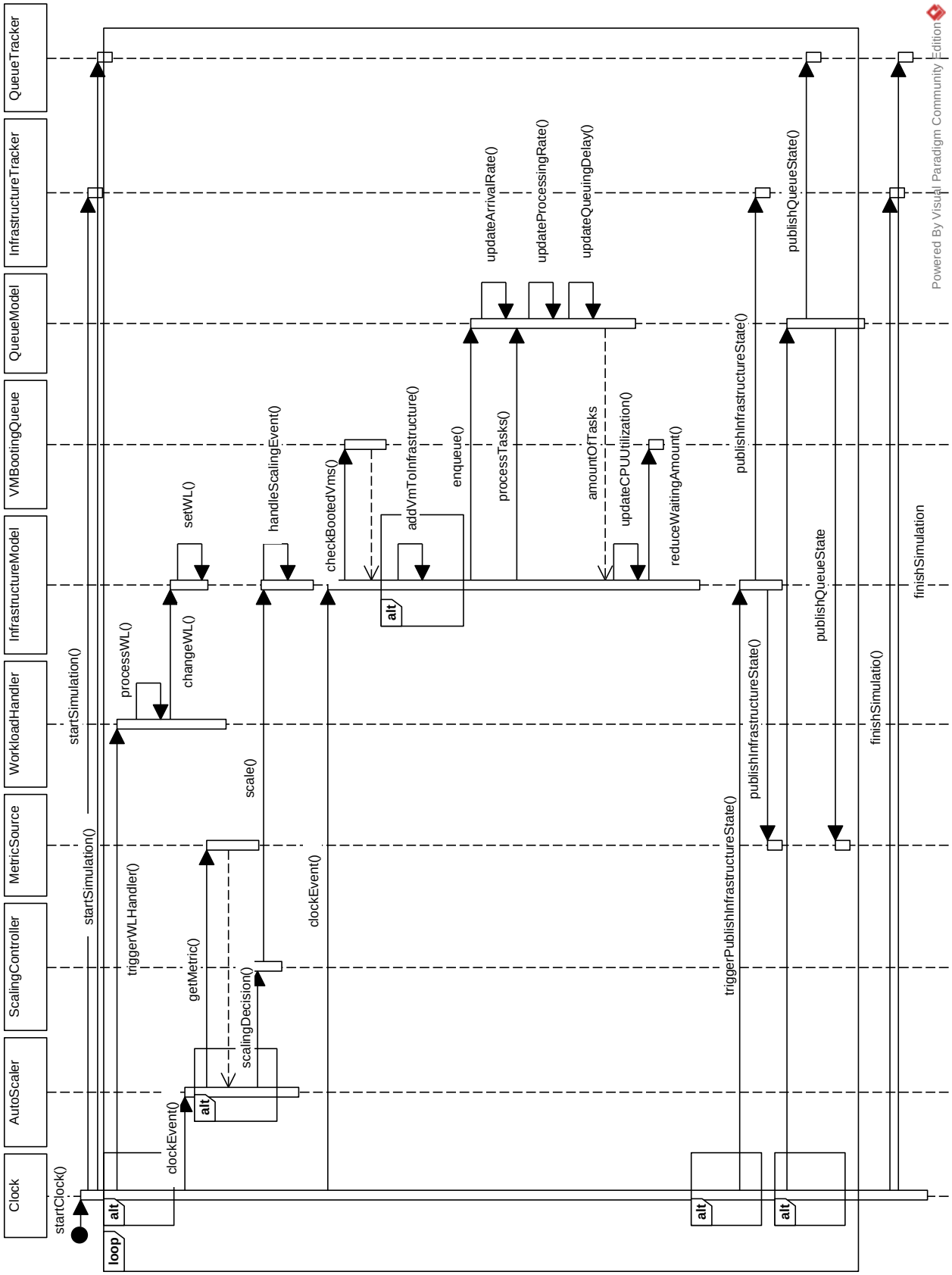

Abbildung 3.3.: Ablauf der zeit-diskreten Simulation

4. Konfiguration

Während Kapitel 3 den Aufbau der Testbench beschreibt, befasst sich dieses mit deren Konfiguration. Da sie so flexibel wie möglich gehalten ist, muss der Benutzer Komponenten wie die Infrastruktur, die Warteschlange, Virtuelle Maschinen und die Auflösung der zeit-diskreten Simulation parametrisieren. Die verwendeten Parameter sollten möglichst nahe der Realität entsprechen, sodass die Simulationsergebnisse korrekt sind. Im folgenden werden die einzelnen Konfigurationsdateien beschreiben, sowie die Anbindung eines individuellen Auto-Skalierers und ein weiterer Tracker.

4.1. Einheiten

Die externe Einheit der zeit-diskreten Simulation wird in Millisekunden angegeben. Intern werden diese in Clock-Ticks umgerechnet. So wird etwa die benötigte Zeit zum Hochfahren einer Virtuellen Maschine (in [ms]) umgerechnet in die Anzahl an Clock-Ticks. Dies ist notwendig, da der Benutzer die Konfigurationsparameter in Millisekunden (oder einer Potenz davon) vorliegen hat. Abhängig von der Auflösung eine Zeitintervalls, also Abstand zwischen zwei Clock-Ticks, wird dieser Wert in Clock-Ticks umgerechnet.

4.2. Konfigurationsparameter

Fünf verschiedene Konfigurationsdateien liegen vor. Diese befinden sich im Verzeichnis *src/main/data*. Für die Zukunft ist geplant, den Pfad dieser Dateien über die Kommandozeile beim Ausführen der Applikation zu übergeben.

4.2.1. Clock

Die Datei *clock.json* beschreibt die grundlegenden Parameter für die Auflösung der Simulation und die Zeitintervalle, in denen diverse Komponenten angestoßen werden.

Listing 4.1: clock.json

```
{
  "millisecondsTillPublishInfrastructureState": 500,
  "millisecondsTillPublishQueueState": 500,
  "intervalDurationInMilliSeconds": 100,
  "millisecondsTillWorkloadChange": 1000,
  "experimentDurationInMinutes": 7
}
```

- **Zeile 2:** Alle 500ms soll die Infrastruktur ihren Zustand publizieren
- **Zeile 3:** Alle 500ms soll die Warteschlange ihren Zustand publizieren
- **Zeile 4:** Intervallbreite, Die Zeit zwischen zwei Clock-Ticks soll 100ms betragen
- **Zeile 5:** Jede Sekunde soll sich die Workload ändern
- **Zeile 6:** Die Simulationszeit soll sieben Minuten betragen

Es ist zu beachten, dass alle Werte (außer die Simulationsdauer) ein Vielfaches der Intervallbreite sind. Ist dies nicht der Fall, kommt es ggf. zu Rundungsfehler, die die Simulationsergebnisse verfälschen.
Beispiel: Intervallbreite = 100ms; Workload-Änderung alle 1000ms. Daraus folgt, dass die Workload alle 10 Clock-Ticks geändert wird. Wird beispielsweise die Workload-Änderung mit 1049ms angegeben, so wird sie immer noch mit 10 Clock-Ticks umgerechnet (durch interne Rundung).

4.2.2. Infrastruktur

Die Datei *infrastructure.json* parametrisiert das *InfrastructureModel*. Die Infrastruktur soll immer nur einen VM-Typ zulassen.

Listing 4.2: infrastructure.json

```
1 {
2   "virtualMachineType":
3     {
4      "millisecondsPerTask": 500,
5      "vmStartUpTimeInMilliSeconds": 3000
6     },
7   "amountOfVmsAtSimulationStart": 1,
8   "cpuUitilizationWindow" :20
9 }
```

- **Zeile 2-6:** Beschreibung des VM-Typs; jede VM benötigt 500ms um einen Job abzuarbeiten. Die Zeit zum hochfahren beträgt 3000ms
- **Zeile 7:** Zu Beginn soll eine VM gestartet sein
- **Zeile 8:** Anzahl der Intervalle, über die ein gleitender Mittelwert der CPU-Auslastung gemessen wird

Durch einen internen Skalierfaktor ist es möglich, dass die Abarbeitung eines Tasks über mehrere Intervallgrenzen hinausgeht. Das CPU-Fenster sollte in Relation zu der Intervallbreite und dem Abstand, mit dem der Zustand der Infrastruktur publiziert wird gesetzt werden (beide Werte in *clock.json* definiert).
Beispiel: Aus *clock.json* geht hervor, dass alle 10 Clock-Ticks der Zustand der Infrastruktur zu publizieren ist. Deshalb sollte der angegebene Wert größer als 10 sein, damit alle Clock-Ticks seit der letzten Publizierung in die Berechnung mit einfließen. Ein Wert von 20 bedeutet somit, dass der Mittelwert der letzten Zustandspublikationen noch mit einfließt.

4.2.3. AutoScaler

Die Datei *autoscaler.json* beschreibt das Verhalten des implementierten Auto-Skalierers. Falls ein anderer implementiert und angebunden wird, so muss auch diese Datei angepasst werden. Eventuell kommen hier weitere Felder hinzu, da ein anderer Scaler andere Konfigurationsparameter erwartet. Diese hier gelten nur für den verwendeten Prototyp.

Listing 4.3: autoscaler.json

```
1  {
2    "lowerThreshold": 0.25,
3    "upperThreshold": 0.75,
4    "vmMax": 20,
5    "vmMin": 1,
6    "timeInMsTillNextScalingDecision": 1000,
7    "cpuUtilWindow" :10,
8    "queueLengthWindow": 10,
9    "coolDownTimeInMilliSeconds": 100000
10 }
```

- **Zeile 2:** Auslastung, ab die der Scaler runter skalieren soll (25%)
- **Zeile 3:** Auslastung, ab die der Scaler hoch skalieren soll (75%)
- **Zeile 4:** Maximale Anzahl Virtueller Maschinen; Ist diese erreicht, so wird auch bei einer Auslastung >75% nicht weiter hoch skaliert
- **Zeile 5:** Minimale Anzahl Virtueller Maschinen; verhält sich invers zu Zeile 4
- **Zeile 6:** Zeit zwischen zwei aufeinanderfolgenden Skalier-Entscheidungen
- **Zeile 7:** Anzahl der zu betrachteten Status-Updates der CPU-Auslastung für einen gleitenden Mittelwert
- **Zeile 8:** Wie Z.7, nur für die Queue-Länge
- **Zeile 9:** Zeit nach einer ausgeführten Skalier-Entscheidung, in der der Scaler nichts ausführen soll

Zeile 7 und Zeile 8 sind Parameter, die für die *MetricSource* zu Verfügung gestellt werden müssen. Diese müssen also auch bei einem anderen Scaler in der Konfigurationsdatei definiert sein. Auch hier ist zu beachten, dass der Parameter in Zeile 6 ein Vielfaches der Intervallbreite aus *clock.json* ist.

4.2.4. Queue

Die Datei *queue.json* beschreibt die Konfigurationsparameter für die Warteschlange.

Listing 4.4: queue.json

```
1 {
2   "queueLengthMax": 80000,
3   "windowSize" :20,
4   "queuingDelayInMilliSeconds": 100
5 }
```

- **Zeile 2:** Maximale Anzahl an Jobs, die in die Warteschlange passen. Ist diese erreicht, so werden weitere Jobs verworfen.
- **Zeile 3:** Anzahl der Intervalle, über die ein gleitender Mittelwert der Warteschlangen-Länge gemessen wird
- **Zeile 4:** Warteschlangenverzögerung, Zeit die ein Jobs braucht zwischen Ankunft im System und verlassen der Warteschlange, bzw. Abarbeitung im System

Die Fenstergröße sollte analog wie die Fenstergröße in Sektion 4.2.2 gewählt werden. Außerdem ist zu beachten, dass die Verzögerungszeit ebenfalls als Vielfaches der Auflösung zu wählen ist.

4.2.5. Workflow

Die Datei *workflow.json* beschreibt den Workflow, also die Last die am System zu einem Zeitpunkt anliegen soll. Dabei entspricht jeder Wert der Anzahl an Jobs, die zwischen zwei Workload-Änderungen abzuarbeiten ist. Intern wird dieser Wert also wie folgt umgerechnet:
Gegeben sei der erste Wert (=7) aus der Liste. Aus *clock.json* geht hervor, dass eine Workload-Änderung alle 1000ms passiert. In dieser Zeit sollen also sieben Jobs abgearbeitet werden. Da die Intervallbreite 100ms beträgt, errechnet sich der Wert $\frac{7 Jobs * 100ms}{1000ms * Interval} = \frac{0.7 Jobs}{Interval}$.

Aus diesem Grund wird intern skaliert, sodass auch mit solch "kleinen"Werten umgegangen werden kann.

Listing 4.5: workflow.json

```
{
  "workflow": [
    7.0,
    5.0,
    8.0,
    6.0,
   ]
}
```

4.3. Anbindung eines Auto-Skalierers

Um einen eigenen Auto-Skalierer zu implementieren, muss eine Klasse erstellt werden, die das Interface *IAutoScaler* in Abbildung 4.7 implementiert. Ein Beispiel dafür ist die Abbildung 4.6. Zuerst muss die bestehende Implementierung entweder gelöscht werden, oder deren Bezeichner im *@Component-Tag* geändert werden. Das Framework benutzt immer den Skalierer mit dem eindeutig vergebenen Namen *@Component("activeScaler")*.
Falls andere Konfigurationsparameter benutzt werden müssen, wie die die in *initAutoScaler()* definiert sind, so muss eine gröberer Änderung vorgenommen werden, die hier nur oberflächlich skizziert wird:

- Ergänze/Lösche Parameter in *autoscaler.json*
- Analog dazu Felder+Getter*Setter in *AutoScalerPOJO.java*
- Adaptiere *initAutoScaler()* in *IAutoScaler.java* und der eigenen Implementierung
- Instanziiere der AutoScaler mit den korrekten Attributen in *ApplicationStartUpRunner.java*, Methode *initAutoScalerAndMetricSource()*.

Die eigene Implementierung muss die Logik komplett selbst implementieren. Das bedeutet, *handleClockTick()* wird in jedem Intervall vom Taktgeber der Infrastruktur angestoßen. Der Auto-Skalierer ist selbst dafür verantwortlich, wann er welche Skalier-Entscheidung trifft, auf welchen Metriken er diese begründet, wie viele VMs hoch-oder heruntergefahren werden oder wie lange seine CoolDown-Phase ist. Dafür stehen die beiden Schnittstellen *IMetricSource* (Abbildung 4.8) und *IScalingController* (Abbildung 4.9) zu Verfügung.
Die *IMetricSource* stellt eine Schnittstelle zum Abrufen von Metriken da. Außerdem kann darüber abgerufen werden, welche VMs gerade aktiv sind. Dies ist notwendig, denn bei einer Skalier-Entscheidung (Modus *'SCALE_DOWN'* aus Abbildung 4.10), muss dem *IScalingController* (Abbildung 4.9)eine Liste der VMs übergeben werden, die herunterzufahren sind. Identifiziert werden diese über eine eindeutig vergebene ID. Sollten neue VMs hochgefahren werden, müssen diese mit den Parameter, die über die Methode *getVirtualMachineType()* ausgelesen werden können und einer eindeutigen ID (ggf. UUID), erzeugt werden.

Listing 4.6: CustomScaler.java

```java
@Component("activeScaler")
public class CustomScaler implements IAutoScaler{

 @Autowired
 private IScalingController scalingController;

 @Autowired
 private IMetricSource metricSource;

 @Override
 public void initAutoScaler(double lowerThreshold, double upperThreshold, int vmMin,
     int vmMax,
   int coolDownTimeInIntervallTicks, int clockTicksTillScalingDecision) {
   // implement logic here
 }

 @Override
 public void handleClockTick(ClockEvent event) {
   // implement logic here
 }
}
```

Listing 4.7: IAutoScaler.java

```java
public interface IAutoScaler {

 /**
 * AutoScaler is initialized with those parameters
 */
 void initAutoScaler(double lowerThreshold, double upperThreshold, int vmMin, int vmMax,
     int coolDownTimeInIntervallTicks, int clockTicksTillScalingDecision);

 /**
 * Implement Logic. Proactive scaling decision! Need to check each clockTick if
 * scaling decision needs to be executed.
 * This includes to administer coolDown counter.
 *
 * @param event
 */
 void handleClockTick(ClockEvent event);

}
```

Listing 4.8: IMetricSource.java

```
1  public interface IMetricSource {
2
3   /**
4   * Moving Average of CPU-Utilization
5   */
6   public double getCPUUtilization();
7
8   /**
9   * Moving Average of Queue-Length
10  */
11  public int getQueueLength();
12
13  /**
14  * Retrieve currently available List of active Virtual Machines
15  */
16  public List<VirtualMachine> getServiceInstances();
17
18  /**
19  * Return Infrastructure-State at a certain Clock-Interval
20  */
21  public InfrastructureStateTransferObject getInfrastructureState();
22
23
24  /**
25  * Information of currently used VM-Type! For a Scaling decision, it is needed
26  * to instantiate VMs with those parameters
27  */
28  public VirtualMachineType getVirtualMachineType();
29 }
```

Listing 4.9: IScalingController.java

```
1  public interface IScalingController {
2
3   /**
4   * Send Scaling decision to Infrastructure. Scaling mode determine whether
5   * Scaling Up or Down
6   */
7   void setInstances(List<VirtualMachine> updatedInstances, int clockTickCount, double
        intervallDuratioInMilliSeconds,
8         ScalingMode mode);
9  }
```

Listing 4.10: ScalingMode.java

```
public enum ScalingMode {
  /**
  * Boot VMs
  */
  SCALE_UP,

  /**
  * Shut down VMs
  */
  SCALE_DOWN;
}
```

4.4. Anbinden eines Trackers

Ein weiterer Tracker kann in diversen Varianten implementiert werden. So kann er beispielsweise die gesammelten Informationen in eine Datenbank schreiben, oder aber auch als Ausgabedatei in Form eines Bildes visualisieren. Wichtig ist nur, dass er eine Listener-Klasse implementiert, die die gewünschten Event verarbeitet. Beispielsweise kann ein Listener auf folgende Event hören:

- *StartSimulationEvent und FinishSimulationEvent*: Starten, beenden der Aufzeichnung.
- *QueueStateEvent*: Aufzeichnung diverser Metriken der Warteschlange
- *InfrastructureStateEvent*: Aufzeichnung diverser Metriken der Infrastruktur
- *WorkloadChangedEvent*: Aufzeichnung der Workload-Änderung
- *ScalingEvent*: Aufzeichnung der Skalier-Entscheidung

Ein solcher Listener ist in Abbildung 4.12 zu sehen. Zusätzlich muss ein Interface des eigentlichen Trackers und eine Implementierung bereitgestellt werden, die die Logik des Trackers umsetzt.
Zu beachten ist, dass absolute Werte noch skaliert werden müssen. Das liegt daran, dass das System intern die gegebene Workload und die Verarbeitungsrate der Virtuellen Maschinen mit einem Faktor skaliert, um genauere Ergebnisse zu Erhalten. Relative Werte (in Prozent) sind dadurch korrekt abgebildet, jedoch müssen absolute Werte mit der Methode *scaleValue(double value, double scalingFactor)* aus Abbildung 4.11 skaliert werden, wobei das Attribut *value* dem ausgelesenen Wert entspricht, und *scalingFactor* einem Wert, der über das *StartSimulationEvent* mitgegeben wird.

Listing 4.11: MathUtil.java

```
public final class MathUtil {
  ....

    public static double round(double value, int places) {
    if (places < 0)
     throw new IllegalArgumentException();
    BigDecimal bd = new BigDecimal(Double.toString(value));
    bd = bd.setScale(places, RoundingMode.HALF_UP);
    return bd.doubleValue();
  }

  public static double scaleValue(double value, double scalingFactor) {
    return MathUtil.round(value / scalingFactor, 4);
  }

  ....
}
```

Listing 4.12: CustomListener.java

```
public java CustomListener {

  @Autowired
  ICustomJobTracker customJobTracker;

  @EventListener
  public void listenStartSimulationEvent(StartSimulationEvent event){
    customJobTracker.start(event);
  }

  @EventListener
  public void listenFinishSimulationEvent(FinishSimulationEvent event){
    customJobTracker.finish(event);
  }

  @EventListener
  public void listenQueueStateEvent(QueueStateEvent event){
    customJobTracker.process(event);
  }
}
```

5. Evaluation

5.1. Approximation

Runden, EInheten hinundher rechnen, alle vms gleich, 0 zeit teilweise dazwischen, alle tasks gleich, keine ausfall der Infratruktur

nachteil dass alles vielfache sein muss

Literatur

[1] M. Jelassi, C. Ghazel und L. A. Saïdane. “A survey on quality of service in cloud computing”. In: *2017 3rd International Conference on Frontiers of Signal Processing (ICFSP)*. Sep. 2017, S. 63–67. DOI: 10.1109/ICFSP.2017.8097142.

[2] Tania Lorido-Botrán, Jose Miguel-Alonso und Jose Antonio Lozano. “A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments”. In: *Journal of Grid Computing* 12 (Dez. 2014). DOI: 10.1007/s10723-014-9314-7.

[3] Alessandro Papadopoulos u. a. “PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications”. In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 1 (Aug. 2016), S. 1–31. DOI: 10.1145/2930659.

A. Anhang

A.1. BPMN Models