QoS-Aware Configuration of Auto-Scalers

Praktikum: Werkzeuge für Agile Modellierung

Niko Benkler

16. 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.

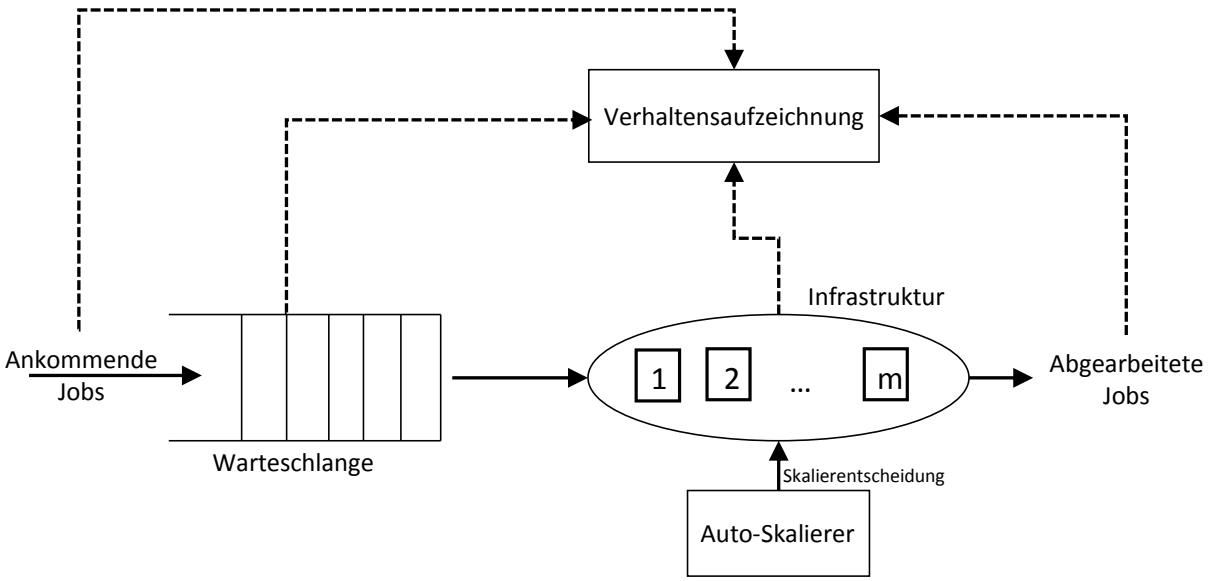

Abbildung 1.1.: Grundlegende Idee der Testbench

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/

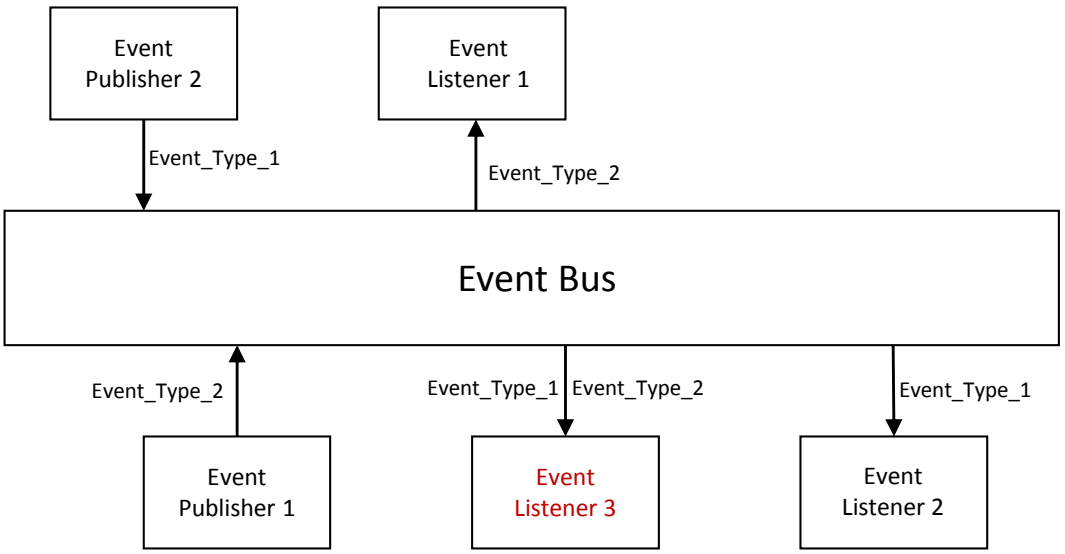

Abbildung 2.1.: Grundlegende Idee der Testbench

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 und Event-Publisher der Komponenten nicht Teil des Diagramms.

3.1. Komponenten

test

3.1.1. Clock

test

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

3.1.2. Infrastructure Model

3.1.2.1. VM Booting Queue

3.1.3. Infrastructure State

3.1.4. Virtual Machine

3.1.5. Auto Scaler

3.1.5.1. Scaling Controller

3.1.5.2. Metric Source

3.1.6. Queue Model

3.1.7. Workload Handler

3.1.7.1. Workload Info

3.1.8. Tracker

3.1.8.1. Queue Utilization Tracker

3.1.8.2. Queue Discarded Jobs Tracker

3.1.8.3. Infrastructure Utilization Tracker

3.1.9. Application Start Up Runner

3.1.10. JSON Loader

3.2. Events

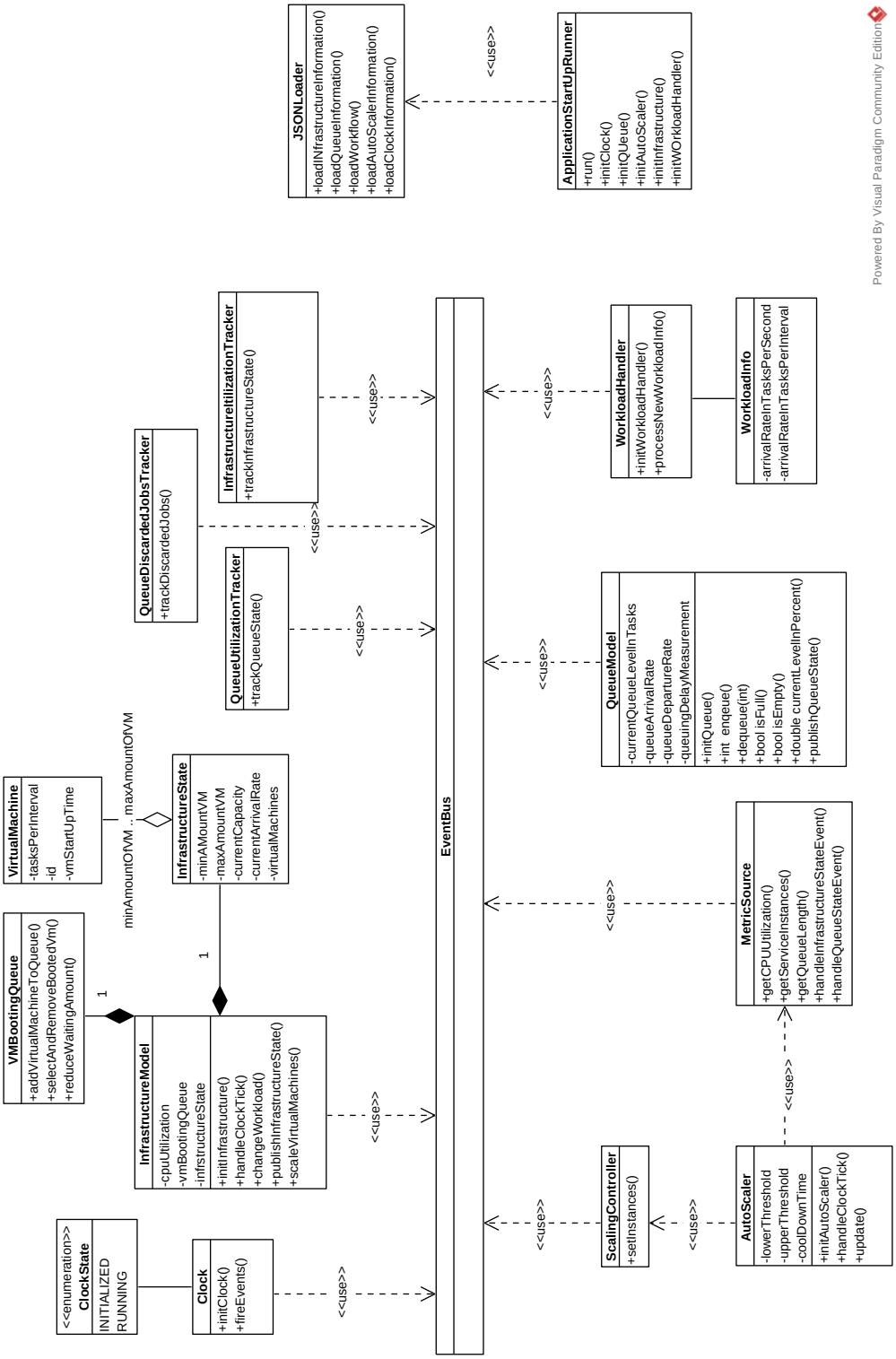

Abbildung 3.1.: Aufbau des Auto-Skalierers

4. Konfiguration

5. Evaluation

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