hochschule mannheim

Virtualisierte Arbeitsumgebung für den Test verteilter Systeme

Jan-Philipp Willem

Studienarbeit
Studiengang Informatik

Fakultät für Informatik
Hochschule Mannheim

23.06.2017

Betreuer: Prof. Dr. Sandro Leuchter
Zweitkorrektor: Jens Kohler

Zusammenfassung

Ein verteiltes Softwaresystem zu entwickeln und dessen Auswirkungen zu verstehen, kann eine Hürde für Studierende im Grundstudium der Informatik darstellen. Oft ist es schon anspruchsvoll genug, in Übungen eine geforderte Transferleistung für Grundlagen umzusetzen. Um dies zu erleichtern, soll eine Umgebung geschaffen werden, welche es ermöglicht, eine verteilte Anwendung unabhängig des eigenen Computers zu testen.

Eine Programmieraufgabe soll von einem Dozenten als Experiment definiert werden können und aus einer gegebenen Anzahl an Instanzen eines ebenso definierbaren Serversystems bestehen. Anschließend soll das Experiment von Studierenden durchgeführt und die eigenen Lösungen hochgeladen werden können. Ferner soll die Möglichkeit gegeben sein, die aus voneinander getrennten Rechner-Instanzen resultierenden Ausgaben darzustellen und die Ergebnisse zu analysieren.

Abstract

As a computer science undergraduate it might be challenging to develop a distributed server-system and yet fully understand its ramifications. Frequently it is demanding enough to apply new principles in tutorial class. To facilitate this, an environment should be build, whereby it is possible to test a distributed program independently of one's own computer.

A lecturer should be able to define an experiment as a programming exercise, which consists of a given amount of instances of a definable server system. Following this, a student is supposed to conduct this exercise and upload his/her own solutions, whereas resulting outputs by independent computer instances are displayed, which should be analyzed.

Inhaltsverzeichnis

1 Problemstellung **1**
1.1 Bisheriger Ablauf 1
1.2 Anforderungen 2

2 Grundlagen **3**
2.1 Eingesetzte Webtechnologien 3
2.1.1 Single-Page-Applications 3
2.1.2 Websockets 4
2.2 Funktionale Programmierung 4
2.3 Verwendete Programmiersprachen 6
2.3.1 Clojure 6
2.3.2 Elm 6
2.4 Container-Virtualisierung mit Docker 8

3 VAR-Tool **10**
3.1 UI-Mockup 11
3.2 Architektur 14
3.2.1 Geschäftsprozesse 14
3.2.2 Netzwerkgrenzen 16
3.2.3 Deployment 17
3.3 Umsetzung 19
3.3.1 Kommunikation über Nachrichten 19
3.3.2 Server 21
3.3.3 Client 24
3.4 Erwartungen an die Performance 30
3.5 Installation 30

4 Fazit **31**

Quellenverzeichnis **v**

Abkürzungsverzeichnis **vi**

Abbildungsverzeichnis

2.1 Schichten einer Virtualisierung mit Docker 8

3.1 Container-Übersicht 10
3.2 UI-Mockup: Übersicht der Experimente 12
3.3 UI-Mockup: Experimentenansicht mit verschiedenen Zuständen pro Instanz 13
3.4 Sequenzdiagramm 15
3.5 Netzwerkdiagramm 16
3.6 Deploymentdiagramm 18
3.7 Umsetzung des Clients: Übersichtsseite 28
3.8 Umsetzung des Clients: Experimentenansicht 29

Kapitel 1

Problemstellung

Diese Arbeit unterteilt sich in vier Kapitel. Das erste Kapitel soll eine Einleitung in die Problematik geben, welche erforscht werden soll. Als nächstes folgen Grundlagen, welche die Techniken und Technologien des VAR-Tool's erklären. Im dritten Kapitel wird der Planungsprozess und mit dessen Hilfe der praktische Teil der Studienarbeit erläutert. Weiterhin folgt ein Fazit der geleisteten Vorgänge und ein Ausblick, wie das Projekt weitergeführt werden könnte.

1.1 Bisheriger Ablauf

Der typische Ablauf bei Programmieraufgaben in einer Laborübung von Verteilte Architekturen (VAR) ist folgendermaßen: Ein Student soll verteilte Aufgaben programmieren, welche es nötig machen, zunächst eine passende lokale Entwicklungsumgebung zu installieren. Dazu wird die Nutzung von schwergewichtigen Integrated-Development-Environments (IDE's) wie Eclipse oder Netbeans vorgeschlagen. Ein Grund dafür sind die schon vorhandenen IDE-Integrationen von Servern und Technologien, welche eingesetzt werden sollen. Dennoch führt schon dieser Prozess bei einigen Studenten zu Problemen.

Gelingt es die Integrationen einzurichten, so kann die Funktionsweise der eigenen Programme getestet werden. Allerdings führt diese Herangehensweise zu keiner echten Verteilung, da alle Services auf der gleichen Maschine ausgeführt werden. Das Ergebnis kann sich bei einer Trennung der Instanzen auf Netzwerkebene erheblich unterscheiden, oder ist unter Umständen nicht lauffähig.

1.2 Anforderungen

Es soll ein Prototyp entwickelt werden, mit dessen Hilfe ein Dozent so genannte Experimente im Kontext von verteilten Systemen definieren kann. Diese sollen jeweils eine gegebene Anzahl an Instanzen besitzen können, auf denen Programmieraufgaben eines Studierenden und die dafür benötigten Technologien ausgeführt werden sollen. Eine jeweilige Instanz wird in einer Konfigurationsdatei durch die Angabe eines Namens, dessen geöffneten Netzwerk-Ports und eine Argumentenliste beschrieben. Das Deployment und die Konfiguration der Systeme soll mithilfe von Docker realisiert werden. Auf Netzwerkebene sollen sich die Instanzen gegenseitig erreichen können, jedoch soll der Zugriff auf Instanzen anderer Nutzer und dem Internet unterbunden werden.

Die Durchführung eines Experiments geschieht durch die Lösung von gewissen Aufgaben innerhalb des Instanzkontextes. Dies geschieht durch einzelnes Hochladen der Programmpakete (Jar, War,..) pro Instanz und der anschließenden Angabe einer Main-Class und einer Argumentenliste. Anschließend kann eine Instanz gestartet und dessen Ausgaben beobachtet werden.

Der Grund für das getrennte Hochladen ist in der Hoffnung begründet, dass die echte Verteilung der Instanzen so von den Studenten besser verstanden werden kann.

Kapitel 2

Grundlagen

Dieses Kapitel möchte einige Grundlagen definieren, welche im Kapitel 3 vorausgesetzt werden. Es handelt von der Definition des funktionalen Programmier-Paradigma, welches in den Sprachen Clojure und Elm zum Einsatz kommt. Weiterhin werden die Webtechnologien Single-Page-Applications (SPA's) und Websockets erklärt, welche das Frontend des VAR-Tool's nutzt. Die Architektur der ganzen Applikation wird maßgebend durch die Trennung in unabhängige Microservices beeinflusst, welche mit Docker realisiert sind.

2.1 Eingesetzte Webtechnologien

2.1.1 Single-Page-Applications

Einzelseiten-Webanwendungen (engl. Single-Page-Applications) stellen eine spezielle Art einer Webanwendung dar. Sie unterscheiden sich zu einer klassischen Webseite dahingehend, dass sie nur aus einem einzelnen Html-Dokument bestehen und ihren Inhalt dynamisch nachladen. Innerhalb dieser Einzelseite wird ebenso Java-Script-Code geladen, welcher die eigentliche Funktionalität enthält. Oft weisen SPA's außerdem Merkmale von Rich-Internet-Applications (RIA's) auf, da ihre Oberflächen typischerweise eher einer Desktop-Anwendung gleichen, als einer statischen Webseite. In diesem Kontext spricht man deswegen auch von so genannten Web-Apps.

Da eine Verlagerung von sonst in einem Server enthaltene Präsentations-Logik stattfindet, resultiert diese Web-Architektur in einem Fat-Client. Es wird dadurch die Serverlast reduziert und gleichzeitig die Grundlage für einen flüssigeren Übergang der Darstellung der Oberflächen geschaffen. Alternativ dazu endet bei einer normalen Webseite der Präsentationsvorgang bei einem neuen Abfragen einer weiteren Unterseite.

Um die zu nutzenden Daten zur Anzeige der Applikation zu erhalten, bedarf es einer gewissen Client-Architektur, welche einzelne AJAX-Requests an eine (REST-)Schnittstelle schickt, oder Nachrichten mit Hilfe einer beid-

seitigen Verbindung über Websockets sendet. Seit in der Java-Script-Welt versucht wurde, das Konzept einer SPA umzusetzen, war es schwierig, ein vernünftig wartbares System zu entwickeln. Bekannte Vertreter von Libraries und Frameworks, welche die dabei anfallenden Probleme zu lösen versuchen, sind Google Angular und Angular 2, beziehungsweise (bzw.) Facebook React oder gänzlich neue Sprachen wie Elm oder Clojure-Script.

2.1.2 Websockets

Websocket-Verbindungen beruhen auf einem gleichnamigen Netzwerkprotokoll, welche eine beidseitige Kommunikation von Client und Server über TCP ermöglichen. Sie haben jedoch, anders wie ihr Name es vermuten lässt, wenig mit klassischen Unix-Sockets gemeinsam.

Das offene Web wurde anhand des HTTP-Standards nach und nach entwickelt. Dieser wurde entworfen um den damaligen Anforderungen gerecht zu werden. So erhält man für jede Anfrage an einem Webserver eine passende Antwort desselbigen. Allerdings wurde es in den letzten Jahren immer wichtiger, Webseiten noch dynamischer zu gestalten. Bei Applikationen welche Informationen in Echtzeit darstellen wollen, kam die Anforderung auf, eine kontinuierlich bestehende Verbindung nutzen zu können.

Es gab Ansätze, um dies mit HTTP-Requests zu erreichen. Die so genannten langen Abfragen (engl. long polling) waren eines dieser. Es wird periodisch eine Anfrage an einen Server geschickt und asynchron auf seine Antwort gewartet. Sobald eine Antwort erhalten wurde, sendet man eine neue Anfrage. Man erhält damit ein Server-Push-Feature.

Websockets stellen eine Alternative zu den Ansätzen auf HTTP-Basis dar. Um solch eine Verbindung aufzubauen, wird ein besonderer Request gesendet, welcher den Browser dazu veranlässt, das Übertragungsprotokoll auf WS bzw. WSS umzustellen. Eine einmal aufgebaute Verbindung bleibt in beiden Richtungen zunächst dauerhaft offen.

Die Ursprünge lagen in einer Diskussion in 2008 und zwölf Versionen später in 2011 wurden die Websockets in allen gängigen Browsern ausgeliefert.

2.2 Funktionale Programmierung

Die funktionale Programmierung gehört zu den Deklarativen Programmierparadigmen und stellt neben der imperativen und objektorientierten Programmierung eines der am Häufigsten genutzten Paradigmen dar.

Anders als bei imperativen Sprachen, unterscheiden sich funktionale Sprachen vor Allem an ihrer Unveränderbarkeit (engl. Immutability) von Variablen und ihrer Datenstrukturen. Es wird dabei angestrebt, bei einer Veränderung des Zustandes (engl. State) einer Variablen nicht den Inhalt ihrer Referenz zu verändern, sondern stattdessen eine Kopie der Daten zurückzuliefern.

Wie der Name schon vermuten lässt basiert die funktionale Programmierung zu größten Teilen aus Funktionen. Diese sind meist als Module gekapselt, welche nach einer bestimmten Aufgabe gruppiert sind. Anders als bei Objekten möchte man offen mit den an die Funktionen übergebenen Daten umgehen und sie nicht in der Implementierung verstecken (engl. Information-Hiding). Dabei sollte eine Funktion nur diese eine Aufgabe erledigen, welche mithilfe ihres Namens beschrieben wird. Um dieses deterministische Verhalten zu erreichen, werden oft sehr granulare Funktionen erstellt, welche anschließend durch eine Komposition miteinander vereint werden[1]. Jegliche Seiteneffekte sollten vermieden werden, was man auch als reine (engl. pure) Funktionen bezeichnet. So sollten Funktionen als Daten-Transformationen verstanden werden: Man erhält Daten, verarbeitet diese und gibt sie anschließend transformiert zurück, welches als Eingabe, Verarbeitung, Ausgabe (EVA)-Prinzip bekannt ist.

Da Funktionen selbst auch Typen darstellen, können sie ebenso als Parameter anderer Funktionen dienen, welche man dann auch als Funktionen Höherer Ordnung (engl. Higher Order Funktions) bezeichnet. Mit dieser Voraussetzung können Verhalten aus einer aufzurufenden Funktion herausgelöst werden, womit das jeweilige Verhalten dynamisch bestimmt werden kann. Diese Art von funktionen werden auch als Callbacks bezeichnet. Hilfreich sind bei dieser Vorgehensweise auch so genannte Lamdas oder Closures, welche Funktionen darstellen, die inline definiert werden können und somit anonym bzw. Unbenannt sind.

Grundsätzlich werden funktionale Sprachen nochmals zwischen solchen Sprachen unterschieden, welche neben anderen Paradigmen auch das Funktionale Paradigma unterstützen und solchen, welche ausschließlich auf den Prinzipien der funktionalen Programmierung beruhen. Die sogenannten reinen funktionalen Sprachen haben nicht einmal beispielsweise (bspw.) die Möglichkeit Seiteneffekte auszuführen und sind sehr oft strikt typisiert. In Multiparadigmensprachen beruhen viele funktionale Prinzipien und deren Erfolg, auf deren Einsatz und der generellen Bedachtheit des jeweiligen Programmierers bzw. des einsetzenden Teams. Man mag diese Prinzipien auch (Programming-)Patterns nennen, welche es zu befolgen, oder missverstehen gilt.

[1] `f(x) = g(x) • h(x) <=> f(x) = h(g(x))`

2.3 Verwendete Programmiersprachen

Das Projekt verwendet für den Server die Sprache Clojure, wohingegen der Client mithilfe von Elm umgesetzt ist.

2.3.1 Clojure

Die Sprache Clojure stellt einen modernen Lisp-Dialekt dar. Sie ist eine Multiparadigmen-Sprache, möchte jedoch den Fokus auf funktionale Programmierung legen. Ihre Quelldateien werden zu Java-Bytecode kompiliert und auf der Java-VM ausgeführt. Clojure ist dynamisch typisiert, aber es existiert eine Bibliothek um Gradual-Typing zu ermöglichen. Die bestehenden Datenstrukturen sind aufgrund ihrer Unveränderbarkeit (engl. Immutability) gerade auch für Einsatzzwecke geeignet, welche eine Nebenläufigkeit bzw. eine asynchrone Verarbeitung erfordern. Durch die Interoperabilität zu Java kann auf deren umfassend bestehenden Bibliotheken innerhalb von Clojure-Code zugegriffen werden. Ihre Bekanntheit hat Clojure unter Anderem (u.A.) durch ihren speziellen Umgang von Zuständen erlangt („Separating Identity from State“).

Ein weiteres bekanntes Merkmal ist das Konzept der Reducer (eager) bzw. Transducer (lazy). Diese ermöglichen es, mehrere Transformationen an einer beliebigen Collection mit bspw. `map, filter, reduce` zusammenzufassen (Komposition). Dadurch kann die benötigte Zeit um die Transformationen auszuführen erheblich minimiert werden, weil nicht jede Teil-Transformation einzeln pro Element stattfindet, sondern gemeinsam. Es existieren in einigen anderen Programmiersprachen[2] Module, welche das Konzept der Transducer in ihrem Kontext umgesetzt haben.

2.3.2 Elm

Elm[3] ist eine rein funktionale Programmiersprache, welche zu Java-Script kompiliert wird. Die Syntax[4] ist an ML-Artige Sprachen orientiert. Sie wurde konzipiert um eine deklarativen Entwicklung von browserbasierten User-Interfaces zu ermöglichen. Um Dinge wie HTML oder SVG abzubilden, existieren Funktionen, welche eine Liste an Eigenschaften eines Elements, sowie eine Liste seine Kinder erwarten. Die Manipulationen an dem DOM des Browserfensters werden durch eine optimierte Vorgehensweise berechnet (Virtual DOM).

Ihr Compiler nutzt den statisch-typisierten Quellcode um eine möglichst hohe Developer-Experience (DX) zu gestalten. So sind bspw. seine Fehlermeldungen dazu gedacht, von Menschen gelesen zu werden. Es ist eine In-

[2] bspw. in Elm: http://package.elm-lang.org/packages/avh4/elm-transducers/latest
[3] https://guide.elm-lang.org/
[4] http://elm-lang.org/docs/syntax

teroperabilität zu Java-Script über so genannte Ports möglich, welche die zugesicherte Typensicherheit gewährleistet. Im Vergleich zu einfachem JS-Code, verspricht Elm keine Laufzeitfehler zu erzeugen, weil der Compiler, mithilfe des Typensystems, sehr erfolgreich dabei ist Fehler zu finden.

Elm hat einige Gemeinsamkeiten bei ihrem Schriftbild mit Java-Script[5]. Allerdings unterscheidet sich Elm dahingehend, dass es zu keinem Zeitpunkt möglich ist, einen `undefined`- oder `null`-Wert abzubilden. Für optionale Werte, oder solche aus Berechnungen, welche auch fehlschlagen können, werden die Monaden Maybe und Result in der Core-Api von Elm verwendet. Maybe stellt einen abstrakten Datentyp dar, welcher einen tatsächlichen Wert enthält (engl. Just) oder ein explizites Nicht-Vorhandensein eines Zustandes (engl. Nothing) in sich trägt. Ein Result trägt entweder den Typ eines `Ok` mit einem Payload oder einen expliziten `Err` mit einem Fehler-Payload. Elm hat eine kluge Records-Definition[6], welche ein einfaches und dennoch direktes aktualisieren von Feldern in typisierten Records zu ermöglichen.

Es wird vorgeschlagen, seinen Code in Modulen zu kapseln, welche einer gewissen Struktur folgen, der sogenannten "'The-Elm-Architecture" (TEA). Diese setzt u.A. das Vorhandensein jeweils einer `init-, update-` und `view-`Funktion und der Deklaration eines `Model`-Typs pro Modul voraus. Eine `update-`Funktion verarbeitet Zustandsänderungen anhand dem Typen seiner eintreffenden `Message`. Elm-`Messages` können bspw. durch ein Klicken auf einen Button ausgelöst werden und können einen Payload enthalten. Die einzelnen Typen von `Messages` können mithilfe eines Verbund-Datenyps (engl. Union-Type) beschrieben werden. Es gibt einige Implementierungen von TEA in anderen Programmiersprachen. Eines davon nennt sich *Redux*, welches im Kontext von Java-Script View-Frameworks (bspw. React) das State-Handling durch eine funktionale Herangehensweise strukturiert. Eine ähnliche Architektur wurde von Facebook's Flux als „One-Way Data-Flow" vorgeschlagen.

Weiterhin werden hilfreiche Abstraktionen wie von Websockets oder HTTP-Requests bereitgestellt, welche ebenso Elm-`Messages` auslösen. In diesem Kontext sind auch die Elm-`Subscriptions` hilfreich, welche bspw. ein kontinuierliches Reagieren auf Websocket-Ereignisse ermöglichen.

[5] http://elm-lang.org/docs/from-javascript
[6] http://elm-lang.org/docs/records

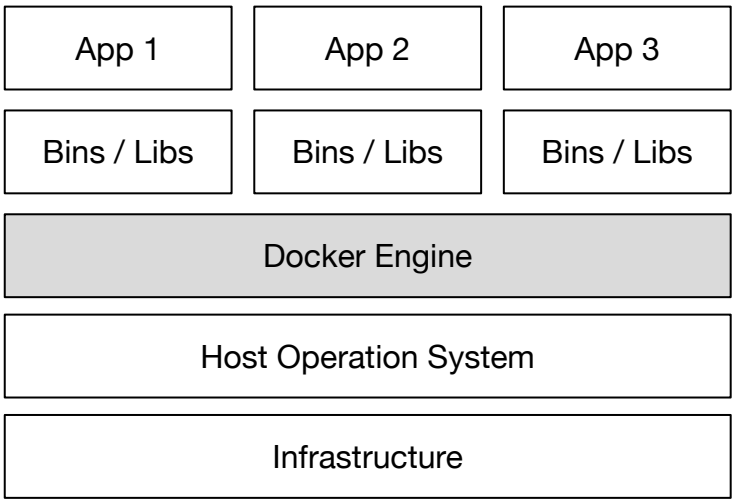

Abbildung 2.1: Schichten einer Virtualisierung mit Docker

2.4 Container-Virtualisierung mit Docker

Docker[7] ist eine Software zum Deployment von Applikationen innerhalb von Containern. Im Vergleich zu Virtuelle Maschinen (VM's) ähnelt sich deren Vorgehensweise, jedoch laufen die Prozesse der Container direkt auf dem Host-Betriebssystem. Grundlage dafür sind Bibliotheken, welche im Kernel oder sehr nah an diesem arbeiten, wohingegen klassische VM's durch ihre potenziell abstrakteren Zwischenschicht, weniger performant arbeiten können. Trotz dieser Tatsache sind die Prozesse mithilfe von u.A. Control-Groups und Kernel Namespaces voneinander isoliert. Weiterhin hat man die Möglichkeit mit gängigen Mandatory-Access-Control-Frameworks wie SE-Linux oder App-Armor die Rechte innerhalb der Container zu beschränken. Als Anforderung besteht keine spezifische Hardware-Infrastruktur wie bei beispielsweise VMWare ESXi. Ebenso ist Docker mittlerweile auf allen gängigen Betriebssystemen lauffähig, wobei Linux-Derivate die beste Unterstützung erhalten. Dies ist damit begründet, dass die Docker-Engine (Abb. 2.1) auf den einzelnen Betriebssystemen verschieden aussieht.

Ein Container wird mithilfe eines Dockerfiles beschrieben. Darin werden deskriptive Instruktionen definiert um Abhängigkeiten zu installieren, Konfigurationen vorzunehmen und andere Build-Steps auszuführen. Ein spezifischer Container wird als Image bezeichnet und setzt sich aus granularen Sub-Images zusammen. Beim Erzeugen von Images eigener Dockerfiles müssen im Vergleich zu VM's keine großen Dateien transferiert werden, da sie aus ihrem Rezept reproduzierbar sind. Es besteht auch die Möglichkeit, von anderen Dockerfiles zu erben und diese über das Netzwerk verteilt bereit-

[7] `https://docs.docker.com/`

zustellen. Falls die über eine Docker-Registry angebotenen Schichten eines Containers noch nicht auf dem eigenen Rechner vorhanden sind, so werden diese heruntergeladen.

Weiterhin gibt es einen entscheidenden Vorteil gegenüber einer traditionellen VM: Ein Dockerfile trägt implizit zur Dokumentation bei, da jede Änderung an einem Container in seinem Rezept ergänzt werden muss, um diese bei einem erneuten Start bzw. erneuten Build zu erhalten.

Wird eine Applikation in einzelnen Services aufgeteilt, so bietet sich das Tool **Docker-Compose**[8] an. Es ermöglicht in einer einzelnen Datei (`docker-compose.yml`) alle Konfigurations-Parameter der einzelnen Container zu definieren. Das können bspw. Volume-Mounts, Ports, Environment-Variables oder Netzwerke sein, welche sonst bei jedem `docker exec` Command angegeben werden müssten. Alternativ dazu stünden eigene und oft fehlerintensive Shell-Scripts um ähnliches zu erreichen. Es bietet sich auch an, getrennte Environments für Development, Test und Production als zentrale Dokumentation der Applikation zu nutzen (`docker-compose.[env].yml`).

[8] `https://docs.docker.com/compose/overview/`

Kapitel 3

VAR-Tool

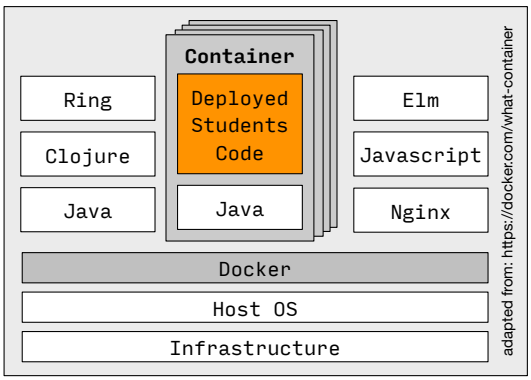

Abbildung 3.1: Container-Übersicht

Diese Kapitel beschreibt die Vorgehensweise und Umsetzung des VAR-Tools. Es wird zunächst ein Mockup des Clients vorgestellt und dann auf die weitere Architektur des Systems eigegangen. Abbildung 3.1 zeigt eine Übersicht der geplanten Applikation. Wir haben auf der linken Seite einen Server-Container, welcher auf Java bzw. Clojure aufgebaut ist. Daneben befinden sich mehrere User-Container in denen anschließend Programmpakete eines Studierenden ausgeliefert werden sollen. Auf der rechten Seite ist ein Client-Container sichtbar, welcher aus einem Webserver (Nginx) besteht. Dieser liefert eine statische HTML-Seite aus, in welcher eine Elm-Applikation als Java-Script integriert ist. Die Container laufen auf einem Rechner, auf dem Docker installiert ist und aus einer beliebigen Infrastruktur besteht.

3.1 UI-Mockup

Im Vorfeld wurde überlegt, welche Features ein Client zur Darstellung von Instanzlogs besitzen soll. Dazu wurden Skizzen auf dem Papier entwickelt, bevor es dazu kommen sollte, ein einfach gehaltenes Design einer Oberfläche zu gestalten.

Abbildung 3.2 zeigt eine Übersichtsseite, welche die verfügbaren Experimente als anklickbare Quadrate mit einem beschreibenden Titel darstellt. Auf der linken Seite könnte eine Menüleiste enstehen, welche die benötigten Aktionen bereitstellt. So gibt es darin ein Link zu der Experimentenübersicht und eine Verlinkung zu einer Hilfe bzw. dem Autor des Tools.

Bei einem Öffnen eines Experiments sollen wie in Abbildung 3.3 eine gewisse Anzahl an zugehörigen Instanzen dargestellt werden. Diese Instanzen können verschiedene Zustände aufweisen.

Der erste Zustand wird in der unteren Ecke skizziert und erlaubt eine Dateiauswahl um ein Programmpaket hochzuladen. Bei einem Drag-and-Drop einer Datei auf der Instanz soll es ebenso möglich sein, diese zum Hochladen anzunehmen.

Die rechte untere Ecke zeigt eine Instanz, welche auf weitere Ereignisse von dem Server wartet. Es wird ein beschreibender Text und ein drehendes Warte-Icon dargestellt. Dieser Zustand soll bei einem Hochladen und auch bei einem Instanzstart angezeigt werden.

Der nächste Zustand (oben links) ermöglicht es, der zu startenden Instanz, eine Hauptklasse innerhalb des Programmpakets und eine Argumentenliste zu übergeben. Ebenso wird in einem Schaubild visualisiert, welche Ports an der Instanz geöffnet sind. Diese sind nicht zu bearbeiten, sollen aber nochmals auf die Aufgabenbeschreibung erinnern. Die hochgeladene Datei wird mit ihrem Namen dargestellt und man kann diese auch wieder mit dem Mülleimer-Icon löschen. Daraufhin wird wieder der Upload-Zustand gezeigt.

Die rechte obere Ecke zeigt eine Instanz, welche gestartet wurde und stellt darin geschehene Ausgaben dar. Man kann die Instanz mit einem Stop-Icon anhalten.

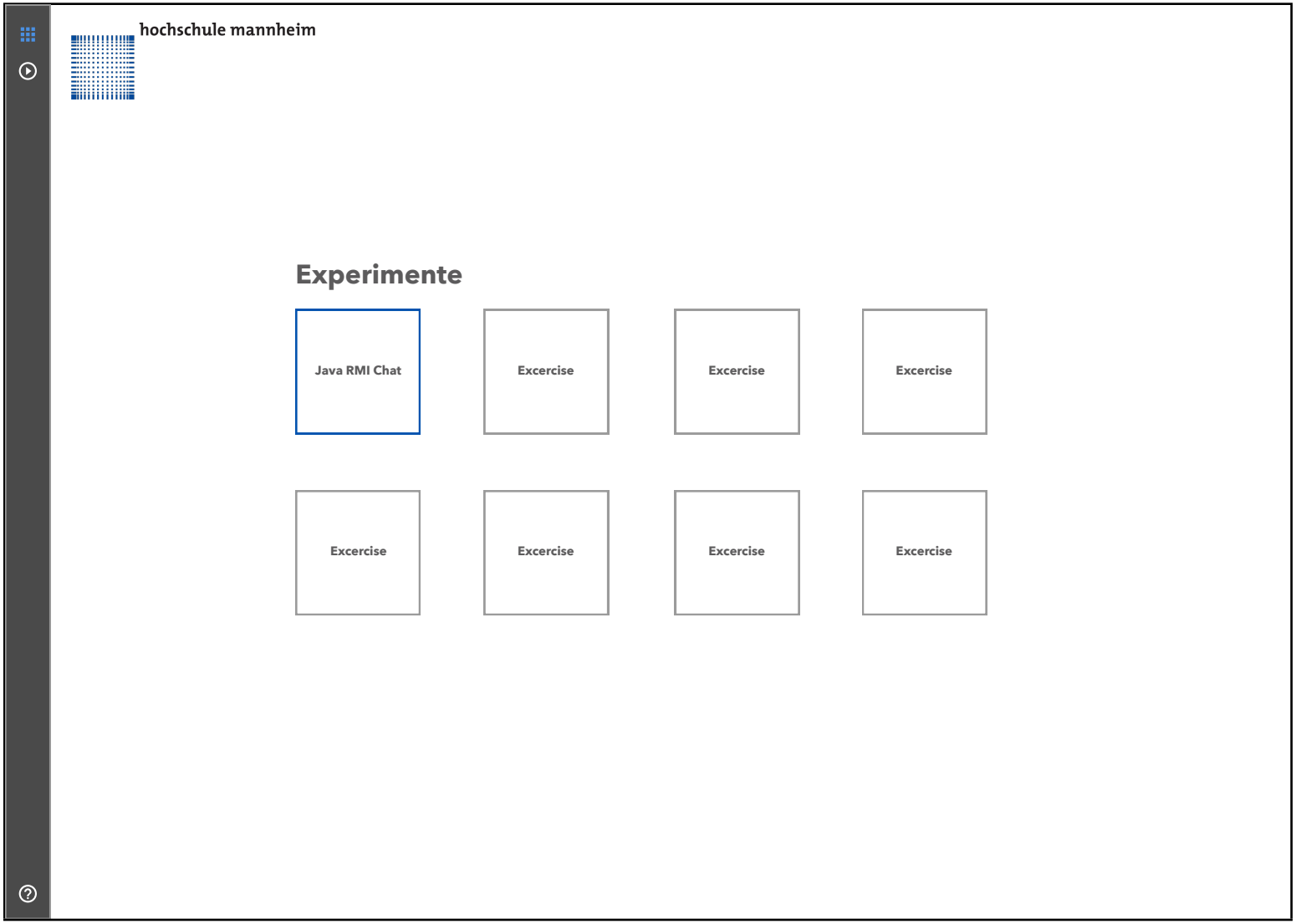

Abbildung 3.2: UI-Mockup: Übersicht der Experimente

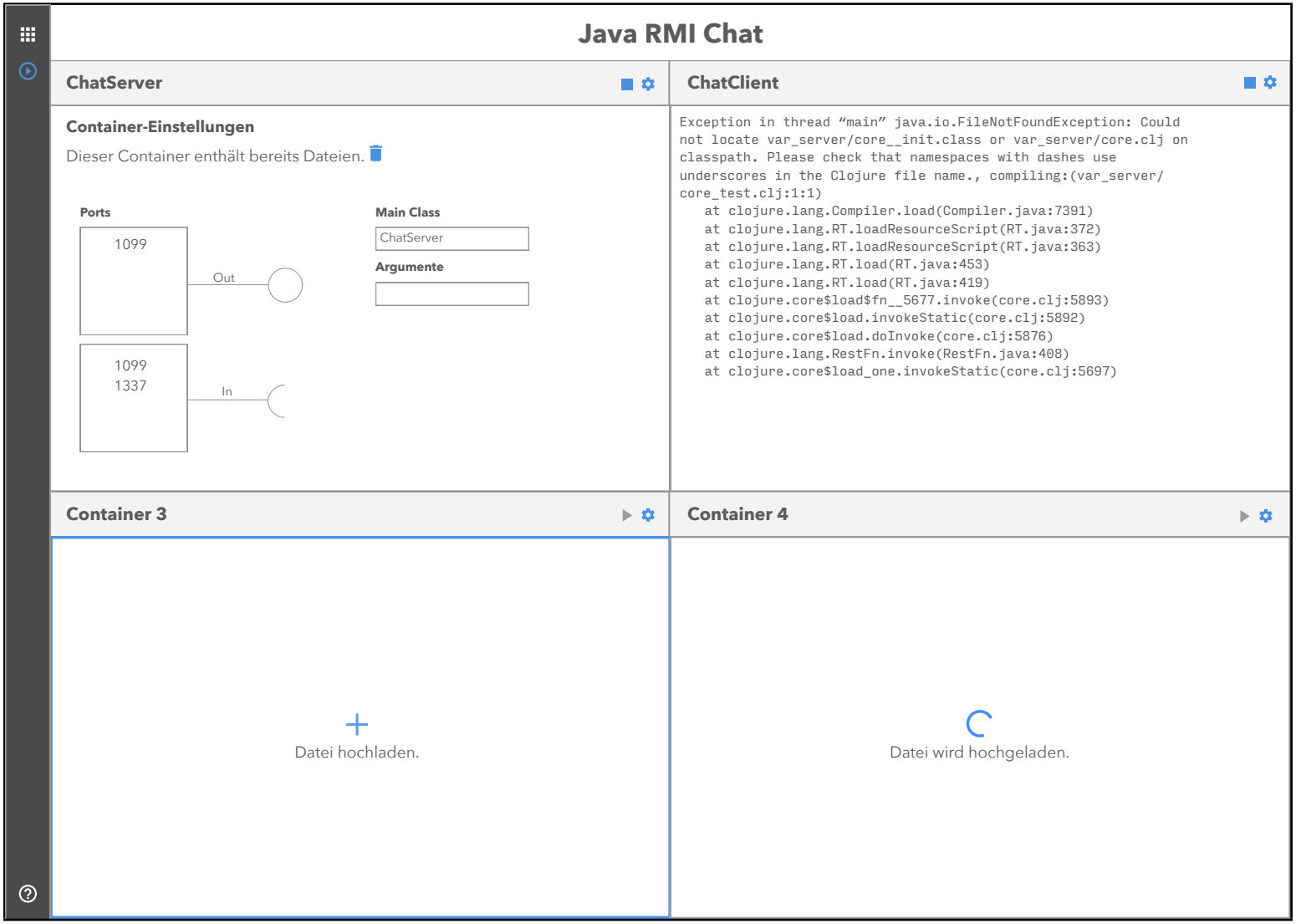

Abbildung 3.3: UI-Mockup: Experimentenansicht mit verschiedenen Zuständen pro Instanz

3.2 Architektur

3.2.1 Geschäftsprozesse

State-Charts

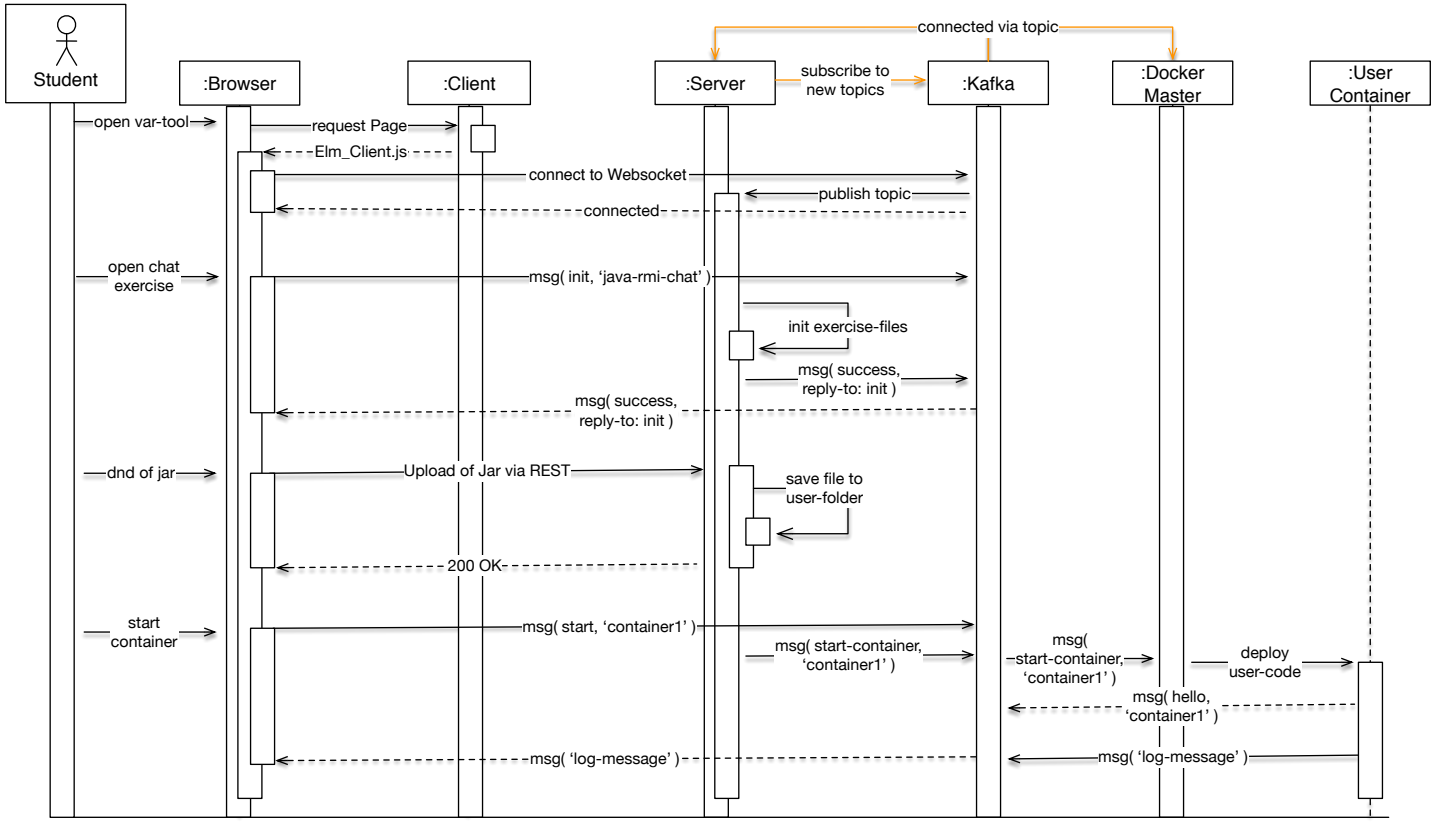

Abbildung 3.4: Sequenzdiagramm

3.2.2 Netzwerkgrenzen

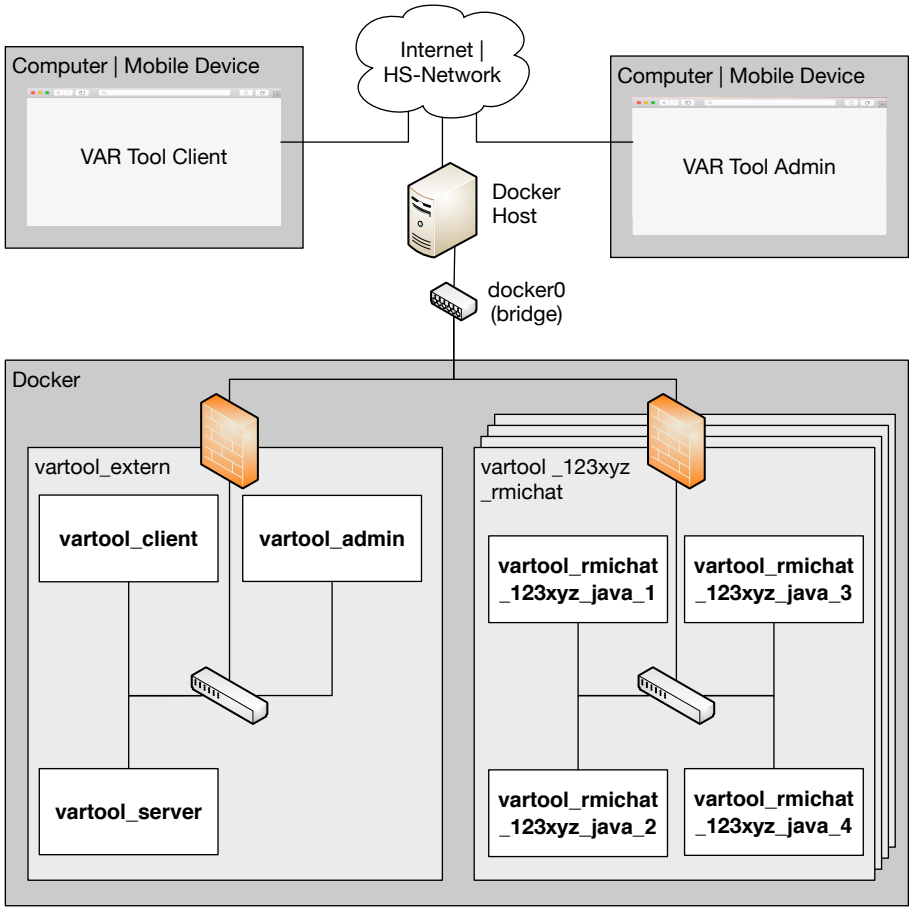

Abbildung 3.5: Netzwerkdiagramm

Es wird ein Rechner benötigt, welcher aus dem Internet oder hochschulintern erreichbar ist. Auf diesem läuft Docker und stellt die Applikation des Projekts bereit. Die Zahl an Benutzern des Clients und Admin-Clients ist beliebig, da das Frontend als Web-Applikation realisiert wird.

Innerhalb von Docker existieren mehrere gekapselte Netzwerke, welche an einem Bridge-Interface des Docker-Hosts angeschlossen sind. Es werden nur solche Ports an den Netzwerken geöffnet, welche in einem Container freigegeben werden. Eines dieser Netzwerke stellt die eigentliche Applikation selbst dar und trägt den Namen *vartool_extern*. Ruft ein Studierender das VAR-Tool auf und startet darin eine Instanz eines Experiments, so wird ein neues Netzwerk angelegt. Dieses Netzwerk ist unabhängig von anderen Experimenten des gleichen und verschiedenen Studierenden, mit dem Namen $vartool\_\{sessionToken\}\_\{experimentId\}$.

3.2.3 Deployment

Abbildung 3.6 zeigt die Verteilung der einzelnen Bestandteile der Applikation. Die Container Server, User und die Clients werden mithilfe von Docker auf einem Linux-Rechner deployed.

Der Client und Admin-Client bestehen jeweils aus dem Webserver *nginx*, welcher ein statisches HTML-Dokument und ein Java-Script Bundle ausliefert, welches aus einer kompilierten Elm-Applikation resultiert. Wird die URL eines Clients von einem beliebigen Rechner mit aktuellem Browser aufgerufen, so werden die benötigten Dateien übertragen und die Elm-App gestartet. Während der Laufzeit des Clients verbindet sich der jeweilige Browser mit dem Server über einen Websocket. Programmpakete welche hochgeladen werden sollen, werden über REST übertragen. Es können dort anschließend verschiedene Experimente angezeigt und bei dessen Ausführung seine Ausgaben dargestellt werden. Im Admin-Client sollen Statistiken dargestellt werden, welche u.A. die aktuelle Anzahl an experimentierten Studierenden umfasst.

Der Server besteht aus einer Java-Umgebung, welche ein Jar mit Clojure-Code ausführt. Er hat die Aufgabe, einen Websocket-Endpoint anzubieten, welcher die Clients zur Kommunikation nutzen. Ebenso soll dieser ein Endpoint anbieten, welcher ein Upload von Programmpaketen der Studierenden per POST-Request annimmt. Jeder Benutzer bekommt einen temporären Ordner zugewiesen, in welchem seine Dateien gespeichert werden sollen. Dieser User-Ordner werden anschließend für die zugewiesenen User-Container eines jeweiligen Studierenden verfügbar sein. Mit diesem Seiteneffekt, über das File-System des Docker-Hosts, kann ein User-Container auf die jeweiligen Programmpakete zugreifen und diese ausführen.

Ein solcher Container kann von dem Server gestartet werden, da der Server auf den Docker-Socket des Hosts zugreifen darf. Sowohl die Eingaben wie auch resultierende Ausgaben sollen zunächst auch über das File-System zwischen Server und einzelnem User-Container geteilt werden. Der Server reagiert auf die Änderung der Datei und schickt die Ausgaben an den Client.

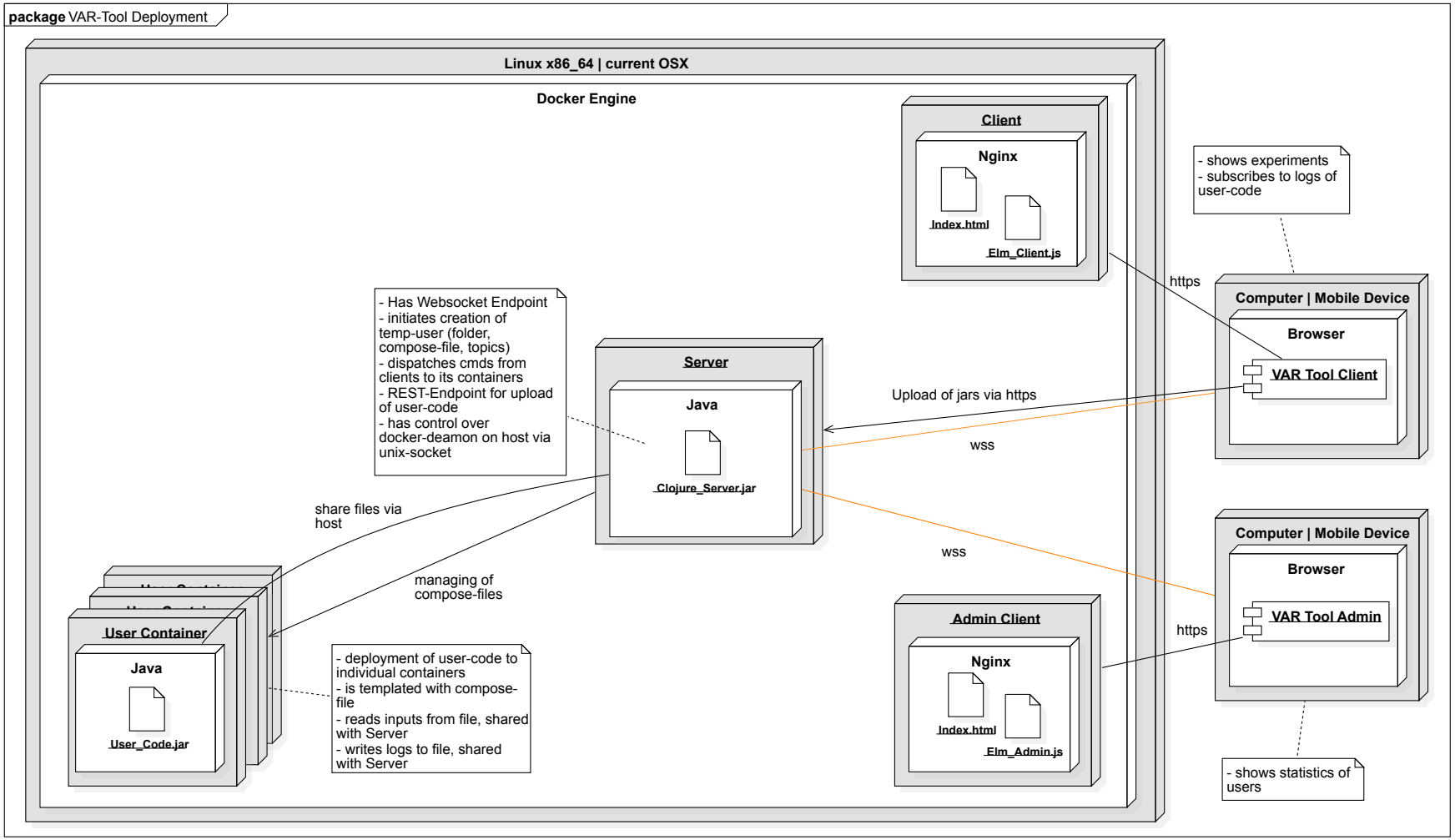

Abbildung 3.6: Deploymentdiagramm

3.3 Umsetzung

Im Folgenden werden Code-Beispiele gezeigt, welche in ihrem vollem Umfang im Repository des Projekts[1] zu finden sind.

3.3.1 Kommunikation über Nachrichten

Im Vorfeld wurde ein Nachrichten-Protokoll definiert, welches die Kommunikation zwischen Server und Client standardisiert. Dabei heißen Client-Nachrichten *Commands* und Server-Nachrichten *Messages*. So können beide Nachrichten-Typen durch ein triviales Merkmal unterschieden werden. Jeweilig unterscheiden sich diese weiter in Kinder-Typen und deren *Payloads*. Dabei werden die Daten über eine Websocket-Verbindung in einem JSON-Envelope als String übertragen.

Commands:

- request-experiments:
  ```
  { kind: command,
  subkind: request-experiments }
  ```
- add-input:
  ```
  { kind: command,
  subkind: add-input,
  payload: { experimentId: rmichat,
           instanceId: 1,
           input: Hello }}
  ```
- start-instance:
  ```
  { kind: command,
  subkind: start-instance,
  payload: { experimentId: rmichat,
           instanceId: 1,
           mainClass: var.rmi.chat.ChatClient,
           arguments: Anton }}
  ```
- stop-instance:
  ```
  { kind: command,
  subkind: stop-instance,
  payload: { experimentId: rmichat, instanceId: 1 }}
  ```

Messages:

- log:
  ```
  { kind: message,
  ```

[1] `https://github.com/jwillem/var-tool`

```
subkind: log,
payload: { experimentId: rmichat,
           instanceId: 1,
           log: Hello}}
```

- reply:

```
{ kind: message,
subkind: reply,
payload: {
    to: request-experiments,
    success: true,
    data: {
        rmichat: {
            id: rmichat,
            name: RMI Chat,
            lecturer: Sandro Leuchter,
            class: VAR,
            numberOfInstances: 4,
            instances: {
                1: {
                    mainClass: var.rmi.chat.ChatClient,
                    arguments: ''
                }, ..
            }
        }, ..
    }
}
```

3.3.2 Server

Der Server besteht im Wesentlichen aus einem Webserver basierend auf *http-kit* und *ring* in der Programmiersprache Clojure. Um die Entwicklung zu vereinfachen, wurde als erster Schritt ein Development-Container in Docker erstellt, welcher sich bei einer Änderung einer Quellendatei selbst aktualisiert, ohne die Java-VM neu zu starten.

Es gibt dort drei Endpoints:

- GET: `server:8080/hello`
  Ankündigen des Clients am Server, Empfangen eines Responses mit `Set-Cockie`-Header
- `ws://server:8080/ws`
  Anmeldung von Client an Websocket, Senden und Empfangen von Nachrichten
- POST: `server:8080/experiment/{experimentId}/instance/{instanceId}`
  Hochladen der Programmpakete eines Studierenden; Werden in temporären User-Ordner in `data/submissions/{session-token}/{experimentId}/{instanceId}/{fileName}` gespeichert

Um auf eintreffende Nachrichten zu reagieren (`on-receive`), wurde eine `match`-Funktion innerhalb des Websocket-Handler genutzt (Listing 3.1). Zunächst werden die Nachrichten von ihrer Json-Repräsentation in ein für Clojure passendes Keyword decodiert. Weiterhin wird auf eine passende verarbeitende Funktion verwiesen, oder einen Fehler mithilfe von `handle-command-error` an den Websocket-Channel zurückgegeben.

Bei einem Eintreffen eines `start-instance`-Commands wird ein Docker-Compose-Template eines Experiments mit den übergebenen Daten (Main-Class & Argumentenliste) befüllt. Dies geschieht mit dem Unix-Tool `envsubst` aus der `gettext`-Suite, welches Umgebungsvariablen in dem Docker-Compose-Template ersetzt. Dabei wird die Datei als `docker-compose.yml` in dem Datenverzeichnis `data/submissions/{session-token}/{experimentId}/` des Projekts gespeichert und ausgeführt (`docker-compose run user_{instanceId}`). Es wird zum Ausführen der jeweiligen Instanz das vorher hochgeladene Programmpaket in einem Unterordner mit dem Namen der Instanz-Id verwendet.

Um ein Experiment zu definieren wurde folgendes Konzept erarbeitet:

- `data/experiments/{experimentId}/`
  dient als ein Entry-Point um portable „self-contained“ Experimente zu enthalten.

- `data/experiments/{experimentId}/experiment.yml`
  enthält einige beschreibende Eigenschaften eines Experiments. Diese können später im Client genutzt werden.

- `data/experiments/{experimentId}/docker-compose.yml.template`
  beschreibt ein Experiment über enthaltene Services mit seinen spezifizierten Eigenschaften der `user_{instanceId}`-Containern. Es können dort auch andere für die Lösung der Aufgabe nötigen Dienste aufgeführt werden.

- `data/experiments/{experimentId}/user/Dockerfile`
  kann optional in Docker-Compose-Templates genutzt werden, um besondere Konfigurationen an einem Containersystem vornehmen zu können. Das Beispiel Template nutzt zurzeit das Docker-Image `openjdk:8`, das beschriebene Dockerfile erbt ebenfalls von diesem.

Das Parsen eines `experiment.yml` erfolgt im Moment noch nicht, ist aber bei geringem Aufwand realisierbar. Die im Client angezeigten Experimente sind in der Datei `server/src/var_tool/server/fixtures.clj` definiert. Natürlich sei es dahingestellt, ob das Austauschformat der Experimente als Yaml oder als Clojure-Keyword sinnvoller erscheint.

```
(defn websocket-handler
  ""
  [request]
  (with-channel request channel
    (on-close
      channel
      (fn [status] (println "channel closed: " status)))
    (on-receive
      channel
      (fn [data]
        (let [session-token (get-in request [:cookies "ring-session" :value])
              ;; TODO error if session-token empty
              ;; TODO return error on java.lang.Exception: JSON error
              command-keyword (json/read-str data :key-fn keyword)
              _ (println "new command: " command-keyword)
              {:keys [kind subkind payload]} command-keyword]
          (match [kind subkind]
                 ["command" "request-experiments"]
                   (handle-request-experiments channel)
                 ["command" "add-input"]
                   (handle-add-input channel payload session-token)
                 ["command" "start-instance"]
                   (handle-start-instance channel payload session-token)
                 ["command" "stop-instance"]
                   (handle-stop-instance channel payload session-token)
                 :else (handle-command-error channel)))))))
```

Listing 3.1: Websocket-Handler im Server (`handler.clj`)

3.3.3 Client

Der Client wurde in der Programmiersprache Elm erstellt und nutzt *elm-mdl* um die Darstellung aus Elementen im Stile des Material-Design Frameworks von Google aufzubauen. Auch für den Client wurde zunächst ein Development-Container definiert, welcher automatisch neue Quelldateien kompiliert und den Browser dazu auffordert, sich zu aktualisieren.

Die Entwicklung seiner Funktionen wurde stark durch das Typensystem und den Compiler von Elm geprägt, welche sich als sehr hilfreich herausgestellt haben. Dazu wurden die Typen in einer Datei `Types.elm` definiert welche in anderen Elm-Modulen importiert wird. Bei der Größe des Projektes ist es noch in Ordnung alle Typendefinitionen in einer Datei zu haben, jedoch sollte man diese in Zukunft bedachtsam in Unter-Module aufteilen.

Weiterhin wurden die Json-Encoder definiert, welche ausgehende Nachrichten in eine Json-Repräsentation bringt. Ebenso war es wichtig eingehende Nachrichten mittels mehrerer Json-Decoder in Elm-Typen zu konvertieren. Man erlangt so eine strikte Typisierung bei sowohl eingehendenwie auch ausgehenden Daten. Innerhalb dieser, konnte ebenso eine `match`-Funktion zum Einsatz gebracht werden (Listing 3.2). Dabei können in einem `case`-Statement beliebige Elm-Typen auf einen Match überprüft werden. Im Payload-Decoder wird ein Record vom Typ Message-Base mit dem `kind` von 'message' und den `subkind`s von 'log' oder 'reply' erwartet. Im Code-Beispiel ist ebenso der Log-Payload-Decoder aufgeführt, welcher schließlich eine Log-Message zurückgibt.

Der vorbereitete Decoder kommt anschließend innerhalb der `update`-Funktion von `Update.elm` zum Einsatz (Listing 3.3). Es wird zunächst überprüft, ob die Dekodierung erfolgreich war und anschließend den beiden Message-Typen zugeordnet, welche jeweilig anders mit den übergebenen Daten umgehen.

Die Umsetzung der Mockups aus Abbildung 3.2 und 3.3 sind in den Abbildungen 3.7 und 3.8 zu erkennen. Dabei wurde versucht möglichst alle initialen Ideen zu erhalten und einige Verbesserungen integriert. So war in den Mockups vorherig kein Eingabefeld vorhanden, um Eingaben an die laufenden Programme zu senden. Weiterhin wurden die Zustände der Instanzansichten vereinfacht. Es sind nun folgende Zustände möglich:

- Empty
  Zeigt den File-Uploader (Abb. 3.8 links oben)
- Uploading
  Zeigt eine Warteinformation beim Hochladen der Programmpakete (Abb. 3.8 rechts oben)
- Settings
  Zeigt die Start-Parameter der Instanz (Abb. 3.8 links unten)

- Running
  Zeigt die Logs einer laufenden Instanz (Abb. 3.8 rechts unten)

Die Darstellung der Instanzen aus Abbildung 3.8 ist dynamisch, d.h. die Höhe der angezeigten Instanzen passt sich je nach Instanzanzahl eines Experiments an. Somit wird die verfügbare Fläche des Browserfensters optimal genutzt.

Die Dateiauswahl innerhalb einer Instanz (Instanz hat den Zustand `Empty`) könnte in Zukunft bei geringem Aufwand einen Drag-and-Drop-Container erhalten, welcher das Hochladen vereinfachen würde.

Die angezeigten Experimente in Abbildung 3.7 werden bei einem Neustart der Applikation über die bestehende Websocket-Verbindung abgefragt (request-experiments). Die dabei erhaltenen Daten werden im Browser-Cache gespeichert und können nach einem Auswählen eines Experiments wiederverwendet werden.

Der geplante Admin-Client konnte aus zeitlichen Gründen nicht mehr entwickelt werden. Die Übermittlung der Daten kann analog mit der oben entwickelten Methode über ein Command im Client bzw. einem Reply im Server geschehen. Die im Admin-Interface anzuzeigenden Daten müssten ebenso noch im Server gesammelt werden.

```
messageDecoder : Decoder Message
messageDecoder =
    map2 MessageBase
        (field "kind" string)
        (field "subkind" string)
        |> andThen payloadDecoder

payloadDecoder : MessageBase -> Decoder Message
payloadDecoder { kind, subkind } =
    case kind of
        "message" ->
            case subkind of
                "log" ->
                    field "payload" logPayloadDecoder

                "reply" ->
                    field "payload" replyPayloadDecoder

                _ ->
                    fail "Subkind is unknown!"

        _ ->
            fail "Kind is unknown!"


logPayloadDecoder : Decoder Message
logPayloadDecoder =
    map3 LogPayload
        (field "experimentId" string)
        (field "instanceId" string)
        (field "log" string)
        |> andThen logMessageDecoder


logMessageDecoder : LogPayload -> Decoder Message
logMessageDecoder payload =
    succeed (LogMessage payload)
```

Listing 3.2: Json-Decoder der Log-Messages im Client (`Decoders.elm`)

```
handleNewMessage : Model -> String -> ( Model, Cmd Msg )
handleNewMessage model message =
    let
        decodedMessage =
            Decoders.decodeMessage message
    in
        case decodedMessage of
            Ok message ->
                case message of
                    LogMessage payload ->
                        handleLogMessage model payload

                    DataMessage payload ->
                        handleDataMessage model payload

            Err error ->
                let
                    _ =
                        Debug.log "Decoder" error
                in
                    ( model, Cmd.none )
```

Listing 3.3: Verarbeiten von neuen Nachrichten im Client (`Update.elm`)

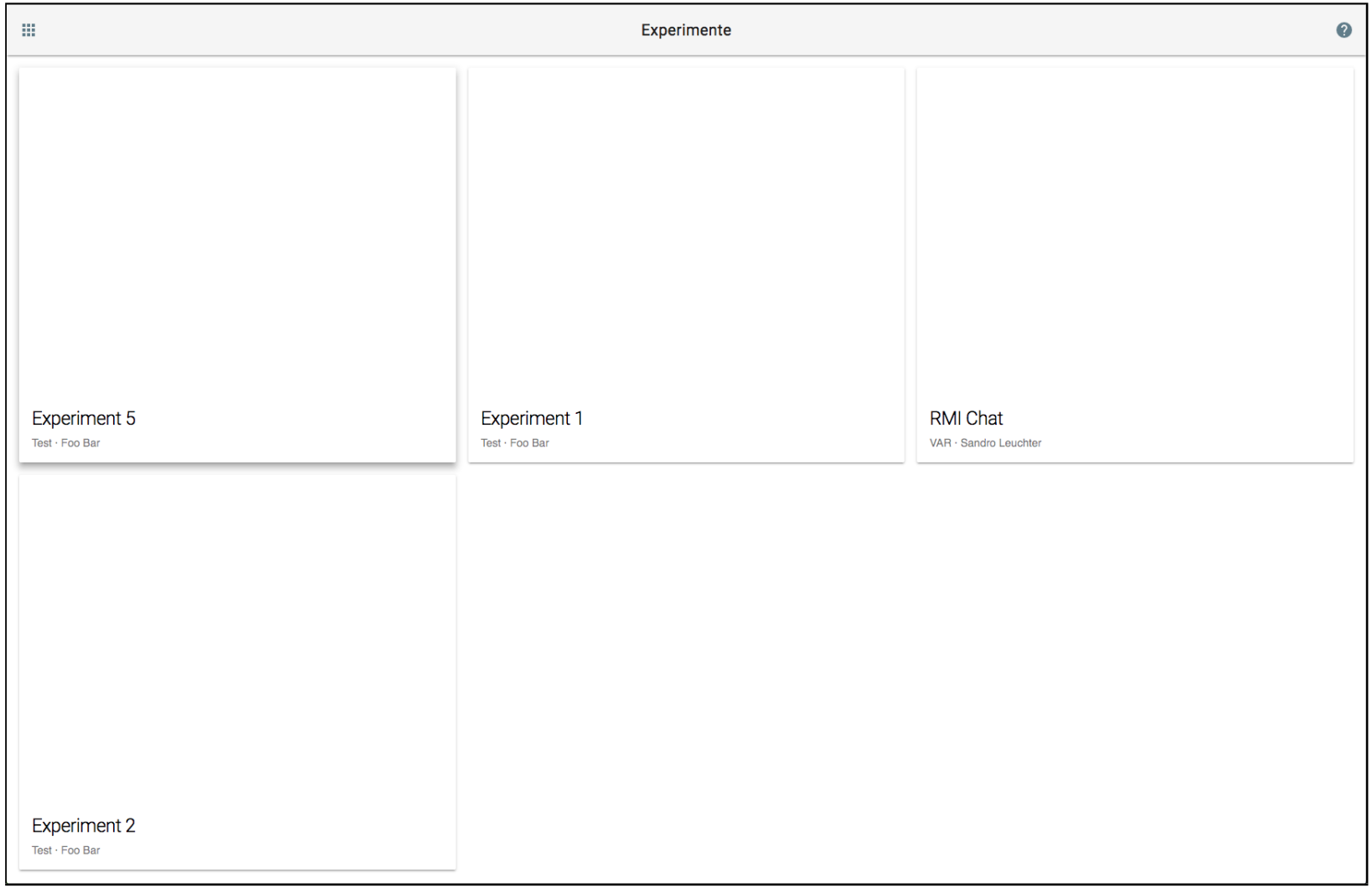

Abbildung 3.7: Umsetzung des Clients: Übersichtsseite

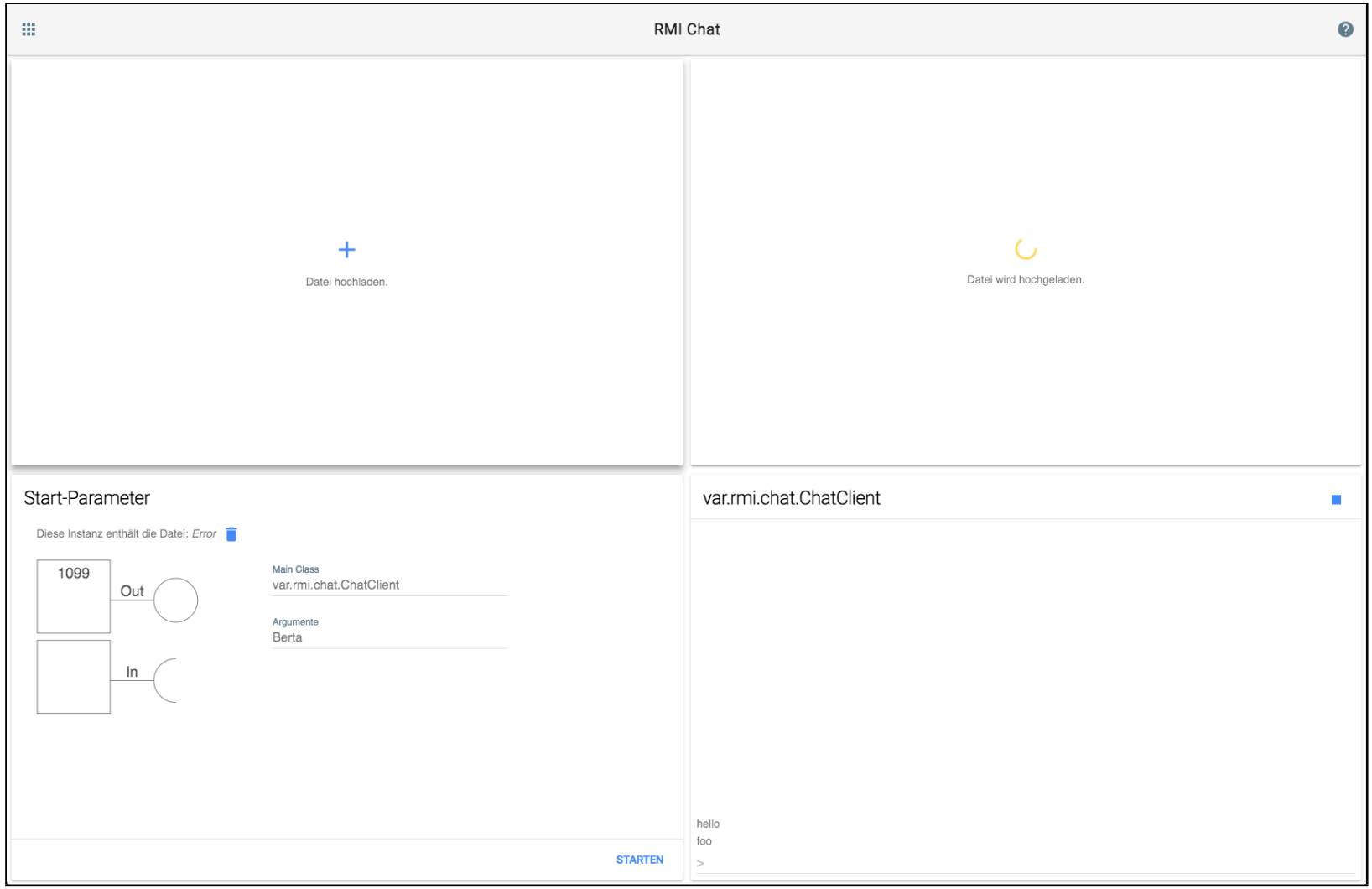

Abbildung 3.8: Umsetzung des Clients: Experimentenansicht

3.4 Erwartungen an die Performance

Der Client ist über einen Websocket mit dem Server verbunden. D.h. bei einer stabilen Internetverbindung sollten auftretende Logs innerhalb von wenigen Sekunden im Client sichtbar sein. Dies ist möglich, da der Client nicht in einem bestimmten Interval nach neuen Logs frägt, sondern aktiv davon benachrichtigt wird.
Auf Seiten der User-Container werden die Logs in eine Datei geschrieben und damit gepuffert. Der Server wartet auf Änderungen an dieser Datei und gibt die Inhalte an den Client weiter. Die Kopplung der User- und Server-Container über die Dateiebene sollte keine Performanceeinbußen bedeuten.

Es konnten von der Applikation leider noch keine Leistungstests gemacht werden. Gerade in Hinsicht auf einen Vergleich zu einem Deployment in einer herkömmlichen VM wäre dies jedoch interessant. Dies wäre bei einer Überführung des `docker-compose`-Files in ein *Ansible*[2]-Playbook möglich, da man über dieses Tool mehrere Deployment-Targets nutzen kann. Bei einem leistungsstarken Rechner sollten allerdings problemlos einige Studierende gleichzeitig mehrere Docker-Container starten können.

3.5 Installation

Als erstes wird das Projekt mithilfe von `git clone https://github.com/jwillem/var-tool.git` heruntergeladen. Um die Applikation zu starten, werden zunächst die Installationen von Docker[3] und Docker-Compose[4] benötigt. Nach einem Ausführen von `docker-compose build` im Projektverzeichnis kann die App mit `docker-compose up` gestartet werden.
Sollte eine weitere Entwicklung erfolgen, so können die vorbereiteten Development-Container genutzt werden. Dazu führt man `docker-compose -f docker-compose.dev.yml up` aus.

[2] `https://www.ansible.com/ansible-container`
[3] `https://www.docker.com/community-edition`
[4] `https://docs.docker.com/compose/install/`

Kapitel 4

Fazit

Während der Entwicklung hatte sich herausgestellt, dass die eigentliche Überlegung, Apache Kafka für die Integration der einzelnen Services der Applikation zu nutzen, erheblich mehr Aufwand bedeutet hätte. In der Vorbereitungsphase wurde eine Bibliothek Namens *kafka-websocket* ausgewählt, welche es versprechen sollte, die Clients über Websocket an Kafka zu verbinden. Bei näherer Betrachtung der Library wurde leider erkannt, dass das Format der einzelnen gesendeten Nachrichten an den sogenannten Kafka-Websocket-Proxy schwer zu verändern ist. Dies passte jedoch nicht zu dem vorherig überlegten Nachrichten-Schema zur Kommunikation zwischen Server und Client. Es wurde statt dem Einsatz von *kafka-websocket* ein eigener Websocket-Endpoint im Server entwickelt, welcher das geplante Nachrichten-Schema umsetzt. Die Übermittlung von Log-Daten aus den User-Containern in den Server wurde stattdessen über die Dateiebene des Docker-Hosts gestaltet. Die User-Container schreiben eine `stdout`-Datei und lesen eine `stdin`-Datei um Eingaben zu akzeptieren. Der Server liest die Datei `stdout` und gibt den Inhalt per Websocket an den Client weiter, bzw. schreibt in die Datei `stdin` um Eingaben vom Client an den Container weiterzugeben.

Die Grundüberlegung bzw. in der Architektur[1] der Applikation Kafka einzusetzen, könnte in einer weiteren Arbeit erforscht werden. Erzeugte Logs der User-Container könnten mit Kafka empfangen und auch persistent gespeichert werden. So wäre es denkbar, dass die einzelnen Ergebnisse der Durchführungen der Experimente auch von einem Dozenten im Nachhinein betrachtet werden könnten. Im Gesamten würde Kafka ebenso den Micro-Service Gedanke des Projekts weiter ausbauen.

Die Verbindung zwischen Client und Server sollte in Zukunft nach aktuellen Best-Practices vorsorglich verschlüsselt werden. Dazu kann für HTTPS ein SSL-Zertifikat von bspw. *let's encrypt*[2] genutzt werden. Ebenso sollten

[1] Hierzu kann im Repository des Projekts ein Diagramm mit dem Namen `deployment_kafka.pdf` gefunden werden.

[2] `https://letsencrypt.org/`

die Websocket-Verbindungen mit TLS über WSS eingerichtet werden.

Die geplante Abschottung der Container wurde bisher nur mit den Bordmitteln von Docker realisiert. Diese ersetzen jedoch nicht den Einsatz von einzelnen Firewall-Rules mithilfe von bspw. *iptables*. Bei einem tatsächlichen Deployment der Applikation könnte daher Server und Client über einen weiteren Proxy-Container (*nginx*) geleitet werden, der außerdem eine Firewall implementiert.

Quellenverzeichnis

@bookButcher:2014:SCM:2621977, author = Butcher, Paul, title = Seven Concurrency Models in Seven Weeks: When Threads Unravel, year = 2014, isbn = 1937785653, 9781937785659, edition = 1st, publisher = Pragmatic Bookshelf,
@booktate2014seven, title=Seven More Languages in Seven Weeks: Languages that are Shaping the Future, author=Tate, B. and Dees, I. and Daoud, F. and Moffitt, J., isbn=9781941222157, lccn=2015473679, series=Pragmatic programmers, url=https://books.google.de/books?id=lc-foAEACAAJ, year=2014, publisher=Pragmatic Bookshelf
@bookHigginbotham:2015:CBT:2836843, author = Higginbotham, Daniel, title = Clojure for the Brave and True: Learn the Ultimate Language and Become a Better Programmer, year = 2015, isbn = 1593275919, 9781593275914, edition = 1st, publisher = No Starch Press, address = San Francisco, CA, USA,

Abkürzungsverzeichnis

Abb. Abbildung

AJAX Asynchronous Java-Script and XML

bspw. beispielsweise

bzw. beziehungsweise

EVA Eingabe, Verarbeitung, Ausgabe

VAR Verteilte Architekturen

VM Virtuelle Maschine

IDE Integrated-Development-Environment

SPA Single-Page-Application

RIA Rich-Internet-Application

REST Representational state transfer

TCP Transmission Control Protocol

u.A. unter Anderem