Praktikumsbericht: eine Docker-basierte Xodx-Simulationsinfrastruktur

im Rahmen des '*Distributed Semantic Social Networks Praktikums*' (WiSe 2014)

Markus Ackermann

Institut für Informatik

Universität Leipzig

14. April 2014

Einleitung

Diese Ausarbeitung dokumentiert Anstrengungen im Rahmen eines kooperativen Studentenprojektes zur Umsetzung einer auf Virtualisierung beruhenden Simulationsumgebung für Distributed Semantic Social Networks (DSSNs).

Zuerst wird die Arbeit an DSSNs kurz motiviert und es wird allgemeine Zielstellung des Studentenprojektes dargestellt. Nachdem kurz Anforderungen identifiziert und die Verantwortlichkeiten des Autors innerhalb der Praktikumsbemühungen umrissen wurden, folgen kurze Einführungen in für das Praktikum relevante Projekte und Technologien. Abschließend wird ein Infrastuktur-Entwurf für die Umsetzung der geplanten Simulationen vorgestellt und diskutiert und abschließend wird über die Tätigkeiten zur Beginn der Umsetzung diser Infrastruktur und deren Ergebnisse berichtet.

1 Überblick über den Praktikumsgegenstand und Motivation der Vorhaben

Die Gesamtzahl aktiver Nutzer Sozialer Netzwerke als auch die kummulative Nutzeraktivität nimmt nun schon viele Jahre stetig zu und die größten und erfolgreichsten Sozialen Netzwerke erweitern regelmäßig ihren Funktionsumfang und durchdringen weitere Aspekte des privaten und öffentlichen Lebens ihrer Nutzer. Bedenklich ist dabei, dass sich die über-

wiegende Mehrheit der Nutzungsaktitivät von Internetdiensten für *Social Networking* auf nur drei große Anbieter entfällt. Einerseits gewinnen diese Anbieter dadurch sehr großen Einfluss auf Medien- und Kommunikationsmärkte in weiten Teilen der Welt. Zudem läuft diese Entwicklung entgegen einem der grundlegendsten Idee des *World Wide Web*, der Pflege einer stark dezentralen Netzwerktologie ([1, S. 7]).

Verteilte Soziale Netzwerke bieten gegenüber den etablierten zentralistischen Sozialen Netzwerken folgende Vorteile (nach [6]):

verbesserte Privatsphäre für Nutzer, da diese beim Betreiben eines eigenens Knotens selbst vollständig über den Umgang mit ihren Daten bestimmen können und andernfalls zumindest eine breitere Auswahl der Datenschutzrichtlinien bei verschiedenen Knotenbetreibern ihres Vertrauens erhalten

verbesserter Datenschutz , da Aufgrund der verteilten Architektur das Stehlen umfangreicher schützenswerter persönlicher Daten aufwendiger wird

erhöhter Schutz der freien Meinungsäußerung , da Zensur von auf einem auf viele Knoten verteiltem sozialen Netzwerk geteilen Inhalten sehr viel schwieriger ist

erhöhte Ausfallsicherheit gegen typische gegen Web-Dienste gerichtete Angriffsmethoden, wie denial-of-service Attacken

Darüber hinaus kann die stark von *Linked Data*-Grundsätzen inspirierte Architektur für ein DSSN gemäß [6] noch weitere wünschenwerte Merkmale unterstützen:

- Nutzer behalten vollständig die Kontrolle über Art und Umfang der Veröffentlichung ihrer persönlichen Informationen durch das DSSN
- der Funktionsumfang des DSSN ist vergleichsweise einfach durch von verschiedenen Interessensgruppen erweiterbar, da die vom DSSN geteilten Resrourcen keinem spezifischen Schema unterliegen und kein zentraler Anbieter existiert, der möglicherweise derartige Erweiterungen aus politischen oder wirtschaftlichen Motivationen heraus einschränken oder unterbinden würde

Xodx ([1]), ist eine Implementierung eines Knotens für ein DSSN gemäß [6]. Das allgemeine Vorhaben des kooperativen Studentenpraktikums war der Test dieser Knoten-Implementation durch die Simulation von Interaktion vieler Knoten-Instanzen untereinander, wodurch folgende Ziele verfolgt wurden:

/SZ1/ ein allgemeiner Stabilitäts- und Lasttest für Xodx

/SZ2/ Untersuchungen zur technische Machbarkeit des Aufbaus und Betriebs eines größeren DSSNs mit Xodx-Knoten bei Nutzerverhalten ähnlich wie bei einem großen, etablierten zentralistischen *Social Networking*-Dienst

Hierfür ergaben sich u.A. folgende Anforderungen an die Umsetzung der Simulation:

/SA1/ die simulierten Nutzeraktivitäten sollen das tatsächliche Nutzerverhalten einer möglichst repräsentativen Stichprobe einer *Social Network Community* möglichst genau nachbilden

/SA2/ alle Knoten der Simulation werden in einer isolierten Systemumgebung ausgeführt und Interagieren untereinander lediglich über die für den tatsächlichen Betrieb vorgesehenen Kommunikationskanäle und Schnittstellen

/SA3/ während der Durchführung der Simulation sind begleitend Kennwerte über interne Zustände, Auslastung und Verhalten der Knoten im simulierten Netzwerk zu erfassen, um anschließend statistische Auswertungen des Netzwerk-Verhalten allgemein und Untersuchung von Unregelmäßigkeiten im Verhalten einzelner Knoten zu ermöglichen

Die im Rahmen dieser Ausarbeitung vorgestellten Beiträge zum Praktikum beziehen sich hauptsächlich auf Simulationsanforderung /**SA2**/. Darüber hinaus wurde ein umfassenderer Vorschlag für die Gestaltung der Simulationsinfrastruktur entwickelt, bei dem /**SA1**/ und /**SA3**/ ebenso beachtet wurden. Bevor dieser schlüssig dargestellt werden kann, sollte in einige relevante Projekte und Technologien eingeführt werden.

2 relevante Projekte und Technologien

2.1 Distributed Semantic Social Networks als Architekturidee

In [6] werden vier Schichten zur Aufteilung der Komponenten und ihrer Verantwortlichkeiten in DSSN Implementierungen eingeführt:

die Daten-Schicht verwaltet zum Einen die Persistenz in sich geschlossener *Resourcen* wie *WebIDs* und Medien-Artefakte und stellt zudem die Schnittstelle für *Feeds*, also Ansammlung von Repräsentationen von im DSSN eingetretenen Ergeignissen und Nutzeraktiviäten, dar

die Protokoll-Schicht ist einerseits mit der Umsetzung des WebID-Protokolls ([5]) betraut und setzt zudem *Semantic Pingback* als Mechanismus zum Herstellen des ersten Kontakt zwischen zwei im DSSN agierenden Agenten um. *Semantic Pingback* wird zudem genutzt um den Eigentümer einer *Resource* über Aktivitäten zu informieren, die sich auf diese *Resource* Bezug nehmen

die Service-Schicht stellt vier grundlegende Infrastruktur-Dienste bereit, die im Rahmen von typischen Anwendungszenarien und DSSN-Applikationen immer wieder benötigt werden: einen *Ping Service*, einen *Push Service*, einen *Update Service* und einen Suchdienst

die Applikationsschicht greift auf Funktionalitäten die Service-Schicht und teilweise auch tiefer liegende Schichten zurück, um letzlich die typischen Anwendungszenarien

für Nutzer sozialer Netzwerke, beispielsweise Blogging, Bekanntmachen des derzeigen Status und das Teilen von Fotos, umzusetzen

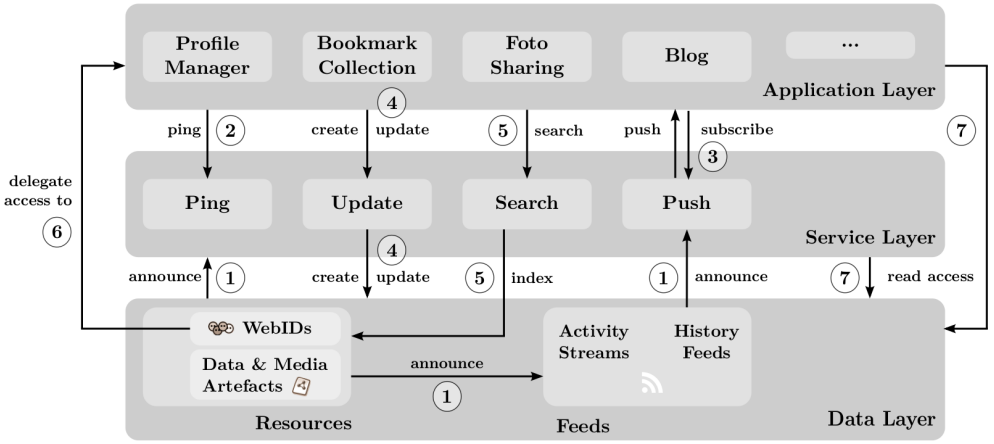

Abbildung 1: mögliche Interaktionen zwischen der Daten-Schicht, der Service-Schicht und der Applikations-Schicht der DSSN-Architektur (aus [6, S. 4])

Ein wesentlicher Gewinn der beschriebnen Unterteilung für den Nutzer ist ein Zugewinn an Transparenz, da diese so ihr persönlichen Daten, über die sie völlig Kontrolle besitzen sollten, sehr viel klarer von stärker fremd-bestimmten Implementierungen der DSSN-Appplikationen trennen können. Zusätzlich zum *Service Decoupling* durch die Schicht-Architektur sind die Umsetzung vom *Linked Data Paradigma* ([3]) bei der Veröffentlichung und Integration von Daten im DSSN und Minimalismus bei der Spezifikation von Protokollen weitere Design-Prinzipien, die beim Entwickeln der DSSN-Architektur umgesetzt wurden.

2.2 Xodx als Implementierung eines DSSN-Netzwerk-Knotens

Im Rahmen des Xodx-Projekts entstand eine Umsetzung eines DSSN-Knotens gemäß der Architektur aus [6] als PHP-Webapplikation. Zusätzlich zu *Zend* wurde hierfür auch auf das *Erfurt-RDF-Framework* und *lib-dssn-php* eingesetzt (beide *spin-offs* des *OntoWiki*-Projekts ([2])), so dass man für die Persistenz der RDF-Daten die Wahl zwischen dem *Virtuoso Universal Server* und einer *MySQL*-basierten Speicherlösung hat. Zur Verringerung der Anfrage-Last für Xodx-Knoten durch, regelmäßiges Anfragen nach Aktualisierungen von im Knoten gehosteten Resourcen (*polling*) wird das *PubSubHubbub*-Kommunikationsmodell ([4]) genutzt. Dadurch können Knoten von Interessenten beinahe augenblicklich über Aktualisierungen benachrichtigt werden, ohne dass hierfür der Knoten, dem die entsprechende Resource gehört, hierfür selbst Verbindung zu allen ‚interessierten' Knoten aufnehmen muss.

2.3 Das PubSubHubbub (PuSH) Protokoll

PuSH folgt dem *publish-subscribe* Kommunikationsmuster. Der Herausgeber (*publisher*) eines per *PuSH* veröffentlichten Artefakts (häufig Atom- oder RSS-Feeds, das *topic*) pflegt in das topic einen Verweise auf einen *PuSH*-Dienst ein. Ein Web-Agent, der an Aktualisierungen für das *topic* Interesse hat, kann andschließend bei diesem Dienst eine *callback*-`URI` hinterlegen. Jedes mal, wenn der *publisher* das *topic* aktualisiert hat, sendet er eine entsprechende Nachricht an den *PuSH*-Dienst, der wiederum für jeden registrierten *subscriber* an die jeweils hinterlegte *callback*-`URI` die Informationen über die neueste Aktualisierung weiterleitet.

2.4 Docker

Docker[1] ist ein Open Source Projekt zur leichtgewichtigen Erzeugen von isolierten und auf verschiedensten Plattformen identisch reproduzierbaren Linux-Ausführungsumgebungen für das Testen und Deployment von Programmen und Diensten. Folgende Entitäten sind dabei von besonderer Bedeutung:

das Host System ist ein vollwertigex Linux-System, auf dem der *docker deamon* Dienst verfügbar ist. Dieses System stellt Speicherplatz, Arbeitsspeicher und CPU-Zeit für auszuführende *Container* bereit

ein Image ist das Abbild eines Linux-Dateisystems, kombiniert mit Festlegungen für eine Umgebungskonfiguration (bspw. bezüglich zu simulierender Netzwerk-Adapter). Images definieren den gewünschten Startzustand des zu virtualisierenden Dienstes

ein Container ist eine isolierte, simulierte Systemumgebung für einen oder meherere darhin ausgeführte Prozesse. Alle Änderungen am simulierten Dateisystem werden vom Docker-System persistent versioniert. Die so gespeicherten Modifikationen des ursprünglichen Dateisystems im Container können auch als Vorlage zur Erstellung weiterer *Images* genutzt werden.

Docker bietet einserseits eine vereifachende Nutzerschnittstelle zu den *Linux Containers* (*LXC*) und ergänzt diese um *copy-on-write*-Versionierung für die in Containern simulierten Dateisysteme (das *unified filesystem*).

2.5 Docker Compose

Docker Compose (ehemals *Fig*) ist ein Python-Kommandozeilenwerkzeug das die Orchestrierung von einzelnen Docker Containern zu größeren logischen Services vereinfacht. Hierzu wird in einer `YAML`-Konfigurationsdatei angegeben, aus welchen Images Container, wel-

[1] https://www.docker.com/

che zusätzlich Parameter, wie Einschränkungen des Speicherverbrauch, auf den Container anzuwenden sind und wie Netwerk-Ports einzelner Container mit einander zu verknüpfen sind. *Docker Compose* nutzt dann die `REST`-API des Docker Dienstes, um die gewünschte Container-Topologie aufzubauen und ermöglichst u.A. das Stoppen und (Neu-)Starten aller zusammengehöriger Container durch einen einzigen Befehl, sowie eine zusammengeführte Ansicht der Konsolenausgaben aller zu einem Service aggregierten Container zugleich.

3 Entwicklung einer Simulationsinfrastruktur

3.1 Ausgangssituation und Rahmenbedingungen

Eine Vorgabe bereits zu Prakikumsbeginn war die Nutzung von *Docker* und es lag bereits ein Github-Projekt vor, in dem deklarative Definitionen für den Aufbau eines Docker-Images (ein *Dockerfile*) vorlag. Dieses Image kapselte einen Virtuoso Open Source Server (VOS) Prozess in der Standard-Konfiguration, kombiniert mit allen weiteren Komponenten, die für den Betrieb eines Xodx-Knotens erforderlich sind:

- eine Kopie des Xodx-Quellcodes selbst, sowie benötige PHP-Bibliotheken
- einen HTTP-Server mit (hier *nginx*) und eine passende *FastCGI*-Umgebung für PHP
- die Datenbankkonnektivitätslösung UnixODBC samt der ODBC-Treiberdateien zum Verbindungsaufbau mit VOS
- angepasste Konfigurationen für nginx und ODBC

Als Docker-Host war zu Praktikumsbeginn ein dedizierter Server mit Ubuntu und 32 GB Arbeitsspeicher vorgesehen. Während der Speicherverbrauch von Xodx-Containern moderat war, zeichnete sich die Menge an Speicher, die für VOS-Container gemäß des o.g. Dockerfiles als stark beschränkender Faktor für die Simulation zahlreicher Knoten ab.

Zur Gewährleistung von /**SA1**/ wurde das Sampling passender Folgen von Nutzeraktivitäten des Microblogging Dienstes *Twitter* beschlossen. Jeder Knoten der Simulationsumgebung repräsentiert einen der im Twitter-Sample vorkommenen Nutzer und erhält eine Auflistung der Ereignisse mit Zeistempeln, die diese Nutzer im Sample durchgeführt hat.

Aus diesen und weiteren Rahmenbedingungen leiteten sich zusätzliche Anforderungen an die Simulationsinfrastruktur ab:

/IA1/ über alle Xodx-Knoten in einem simulierten Netzwerk hinweg ist das voranschreiten der Simulationszeit einheitlich zu koordinieren

/IA2/ die in Docker-Containern simulierten Komponenten für ein Experiment müssen derart parametrisierbar sein, dass ihr maximales Speicherverbauch vorhersagbar und möglichst moderat ist

/IA3/ jede Xodx-Instanz in einen Docker-Container muss durch eine URI erreichbar sein, die über die gesamte Simulations-Infrastruktur hinweg eindeutig ist

3.2 Architekturüberblick

Ein bei der Nutzung von Docker empfohlenes Prinzip, lautet, dass jeder Container nur lediglich einen einzigen primären Prozess beherbergen sollte. Dieses Prinzip ist dem *single responsibility principle* des Objektorientierten Designs ähnlich erhöht die Wartbarkeit und Wiederverwendbarkeit der entwickelten Docker Images. Dementsprechend wurde auch die Aufteilung vom VOS Triple Store und den Webapp Komponenten für Xodx Instanzen in verschiedene Container übernommen und es sind ebenfalls seperate Container für lokale PuSH-Hubs auf dem Host-Systems vorgesehen. Im Vorfeld des Aufbaus einer Simulationsanordnung werden die zu simulierenden Nutzerereignisse aus dem Sample des Twitter-Ereignisstromes (siehe 3.1) nach Twitter User IDs gruppiert und chronologisch sortiert. Anschließend wird eine Bijektion $tw2int$ zwischen den N_U vorkommenden Twitter User IDs und den Zahlen $1 \ldots N_U$ festgelegt (etwa durch lexikografisches Sortieren der User IDs). Für jeden Nutzer u mit $k = tw2int(u)$ wird ein Docker Data Volume[2] angelegt, in dem die u im Twitter Sample ausgeführten Aktionen in einer geeigneten Serialisationsform abgelegt werden. Für jedes k wird beim Aufbau der Simulationsanordnung ein VOS Container und ein Container mit einer Xodx Instanz erstellt, wobei der Port für die Virtuoso ODBC-Anbindung direkt durch einen Docker Link im Xodx Container erreichbar gemacht wird.

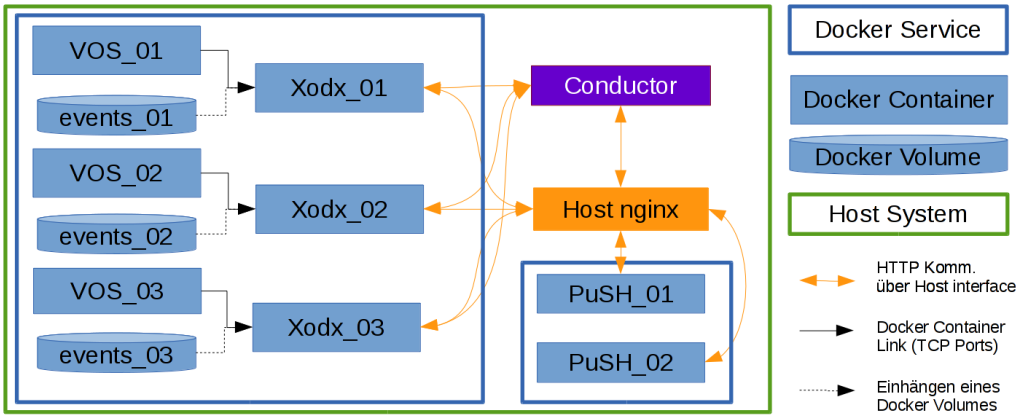

Abbildung 2: schematische Darstellung der Simulationsarchitektur (Node Driver sind nicht explizit dargestellt, da sie in die Container für Xodx Instanzen integriert werden)

Zusätzlich zur Xodx Anwendung enthalten Xodx-Container ein weiteres leichtgewichti-

[2]Docker Data Volumes sind Teilbäume des Dateisystems in einem Docker Container, die von der Versionierung von Dateisystemänderungen für den Container ausgenommen sind. Statt dessen werden die Inhalte von Data Volumes vom Docker Dienst unabhängig vom nutzenden Container verwaltet und persistiert.

ges Programm mit einer simplen REST-Schnittstelle, den *Node Driver*. Der *Node Driver* liest auf Anfrage des Conductors alle serialisierten Nutzeraktivitäten aus den einhängten Events-Volume, übersetzt diese in entsprechene HTTP-Anfragen, um in der Xodx Instanz desselben Containers die entsprechenden Aktivitäten auszulösen, sendet diese an die Xodx Anwendung und speichert alles HTTP-Responses von Xodx, die Fehler oder Unregelmäßigkeiten signalisieren. Wenn alle Aktivitäten zur angefragten Zeitscheibe ausgeführt wurden oder ein Timeout verstrichen ist, wird ein Report über die Anzahl erfolgreicher, nicht-erfolgreicher und ggf. ausstehender simulierter Nutzeraktivitäten an den Conductor versandt.

Wie bereits angedeutet, initiert der Conductor schrittweise das Arbarbeiten zu simulierender Nutzeraktivitäten (vgl. /**IA1**/). Zudem können hier Informationen über Probleme beim Nachbilden von Nutzeraktivitäten aggregiert werden und der Conductor kann ggf. die Fortführung der Simulation abbrechen, sobald eine Obergrenze der Anzahl fehlerhafter oder ausbleibender Nutzeraktivitäten erreicht wurde.

Zur Erfüllung von /**IA3**/ wird für den Xodx-Container zu k der HTTP Port innerhalb des Containers durch Docker auf einem virtualisierten Netzwerkadapter auf den Port $8000 + k$ [3] freigegeben. Dadurch kann jede Xodx-Instanz zwar bereits durch URIs der Form `http://localhost:8001` erreichen. Abschnitt ??? behandelt allerdings, warum diese Kommunikationskanäle unzureichend sind und wie damit verbundene Probleme umgangen wurde.

Die Gesamtheit aller Xodx Container zusammen mit den für sie eingerichteten VOS Containern und den Event Volumes bilden zusammen eine größere logische Einheit, die als ein aggregierter *docker compose* Service gruppiert wird. Da eine Menge von simulierten Xodx Containers prinzipiell ebensogut ohne im Host-System bereitgestellte PuSH-Hubs operieren könnten und diese auch deutlich eigene, abweichende Verantwortlichkeiten haben, werden diese in einem eigenständigen docker compose Service verwaltet, ebenfalls für die in Abschnitt ??? besprochenen Ansätze für die Xodx-Instanzen erreichbar gemacht.

3.3 Diskussion möglicher alternativer Entscheidungen zur Simulationsarchitektur

Ein weiterer, weniger aufwendiger Ansatz für ein koordiniertes Voranschreiten in der Simulationszeit für die Node Driver ließe diese schlichtweg die Systemzeit nutzen. Es könnte etwa einfach ein Zeitpunkt eine Datei ins Dateisystem eines jeden Xodx Containers geschrieben werden, der als t_0, die Simulationsstartzeit aufzufassen ist. Die Node Driver würden dann regelmäßig die Systemzeit abfragen und könnten so für jede im Event Volume vorgeschriebene Aktivität mit relativen Zeitpunkt t' diese ausführen lassen, sobald die Systemzeit

[3] Die Wahl des Startwertes 8000 ist arbiträr und es wäre jeder andere größere Bereich unbelegter non-priveleged Ports ebenso geeignet.

$t_0 + t'$ erreicht hat. Obgleich dieser Ansatz den großen Vorteil hat, dass die Node Driver ohne weitere Kommunikation mit einer zentralen Steuerstelle implementiert werden können, hat er auch eine schwerwiegende Einschränkung: Sobald das Ausführen einzelner Aktionen für Xodx Instanz X_A um ein vielfaches länger dauert, gefährdet dies die Integrität des gesamten Simulationszenarios, da bereits der Node Driver für eine weitere Instanz X_B bereits Aktionen abrufen könnte, die in den ursprünglichen Twitter Aktivitätsstrom eine Reaktion auf eine verzögerte Aktion in X_A war und somit noch gar nicht gültig ausgeführt werden kann. (Beispielsweise könnte auf X_B bereits versucht werden, eine Statusmeldung zu kommentieren, die auf X_A noch gar nicht erfolgreich erstellt wurde.) Derartige Phantom-Reaktionen können kaskadieren, da reagierende Aktionen auf die so in X_B ausgebliebene Aktion gegenfalls ebenfalls nicht ausführbar sind usw. Verschiedene Ansätze zur Überwachung von Fehler-Logs für autonome, systemzeitgesteuerte Node Driver wären denkbar, um derartige Fehlermuster zu erkennen, aber es ist fraglich, ob dies weniger komplex in der Implementierung und resourcenschonender für das Host-System wäre als die vorgeschlagene Lösung über ein schlankes HTTP-Protokoll mit geringen Nachrichtengrößen.

Eine konsequentere Umsetzung von Docker Best Pratices würde je nach Sichtweise das kapseln von Node Drivern in eigene Docker Container implizieren. Als Gegenargument ist die mögliche Interpretation anzuführen, dass die Aktivität des Node Drivers den eigentlichen primären Prozess im Xodx-Container während der Ausführung der Simulation darstellt und die Xodx-Instanz nur ein Dienst für den Node Driver ist.

4 Implementierungsarbeiten für die Simulationsinfrastruktur

4.1 Konsolidierung der Docker Images for VOS und Xodx

Zunächst wurde das in Abschnitt 3.1 erwähnte bereits vorhandene *Dockerfile* in die Beschreibung für zwei separate Images aufgespalten. Ein Dockerfile-Template ist nun für das Erzeugen von Images für VOS Instanzen zuständig und ein weiteres Dockerfile erzeugt ein die Umgebung für eine Xodx-Instanz in die ein VOS-Container an den Hostnamen `vos` zu verlinken ist. Dabei wurde auch die Startskripte nachgebessert, die die *Entrypoints* der Images sind zuerst die erforderlichen Dienste starten, um anschließend fortlaufend die Änderungen an relevanten Log-Dateien anzuzeigen. Während zuvor die Container vom Docker Dienst nur durch `SIGKILL` zum Stillstand gebracht werden konnten, findet nun bei beiden Containern ein geordnetes Beenden statt. Desweiteren wurden die in Abschnitt ??? dargestellten Anpassungen an Xodx in den Kontext für den Aufbau des zugehörigen Images eingepflegt.

4.2 Parametrisierbare Docker Images

Die Anforderung /**IA2**/ in Abschnitt 3.1 war hauptsächlich durch die Beobachtung motiviert, dass eine Instanz des Virtuoso Open Source Servers gemäß der Konfiguration entsprechend der Debian Pakete bereits mehr als 100 MiB Arbeitsspeicher benötigt, obgleich in diesem Fall nur eine sehr geringe Menge an Arbeitspeicher für *disk page caching*[4] vorgesehen ist. Dieser recht hohe Sockerbetrag des Speicherverbrauchs ist auf den großen Umfang von Funktionalitäten eines einen Triple Stores und RDBMS hinaus zurückzuführen, der durch VOS abgedeckt werden soll. Die Entwickler von VOS haben allerdings Probleme mit derart hohen Speicheranforderungen für bestimmte Einsatzszenarien erkannt und bieten daher *LiteMode* an, in dem nur Module für Kernfunktionalitäten geladen werden. Dadurch sinkt der minimale Speicherbedarf auf ca. 40 MiB. Wünschenswert war darüber hinaus das automatisierte Generieren von VOS Docker Images für Simulationsanordnungen, für die eine Obergrenze Speicherverbrauch vorgegeben wird, diese Obergrenze nicht überschritten wird, aber zugleich *disk page caching* in einem Umfang stattfindet, dass der überwiegende Teil des zugesprochenen Speichers auch für entsprechenden Performance-Gewinne durch *caching* ausgenutzt wird.

Entsprechendes Tooling wurde in *Python* implementiert. Zur Erstellung eines VOS Images laufen im Wesentlichen die folgenden Arbeitsschritte ab:

1. Bestimmen der passenden Konfigurationsparameter, in diesem Fall, ob der *LiteMode* genutzt werden soll und die Größe des *caches* als Anzahl von *disk pages* gemäß des vorgebenen gewünschten Speicherverbrauchs in MiB (Klasse `VirtuosoConfiguration`)

2. Generieren des Textinhalts for die VOS Konfigurationsdatei aus einem *mustache*-Template durch einsetzten der Konfigurationsparameter (Methoden aus dem `TemplatingMixin`)

3. Kopieren des anzupassenden *Dockerfile*-Kontexts in ein temporäres Verzeichnis, anschließend Modifizieren dieses Kontextverzeichnisses. In Fall der VOS Images ist die einzige Modifikation das Schreiben des VOS Konfigurationtexts in die Datei `virtuoso.ini` (Klasse `VirtuosoImage`)

4. der Docker Dämon erhält über die Docker `REST`-API den Befehl ein neues Image aus dem temporären Dockerfile-Kontext zu erstellen (hierfür wird auf die Programmbibliothek *docker-py* zurückgegriffen). Der Name des neuen Images wird dabei automatisch aus den Parametern gebildet, mit denen der Image-Erstellungs-Prozess gestartet wurde.

[4] VOS organisiert seine primären Datenbankdateien in Pages des Größe 8192 Byte. *disk page caching* bezeichnet hier die Strategie, häufig benötigte Pages im Arbeitsspeicher bereitzuhalten. Die Anzahl an Pages, die so schneller für Anfrageverarbeitungen lesebereit sind, hat den größten Einfluss aller VOS Konfigurationsoptionen auf die allgemeine Performance dieses Triple Stores.

4.3 einheitliche und eindeutige URIs für Xodx-Instanzen in einer Simulationsanordnung

Um Anforderung /**SA2**/ in Abschnitt 1 zu erfüllen musste sowohl für die Node Driver[5] als auch für Xodx-Instanzen einer Simulationsanordnung untereinander die Möglichkeit geschaffen werden, HTTP-Anfragen an alle weiteren Xodx-Instanzen und sich selbst absetzten zu können. Docker stellt zwar bereits HTTP-Zugriff auf die Xodx-Instanzen unter URIs wie `http://localhost:8001` bereits (vgl. Abschnitt 3.2), aber diese Lokatoren sind kontextabhängig und daher für die Simulationsanordnung problematisch: Wird vom Host-System aus `http://localhost:8001` abgerufen, erhält man zunächst eine erfolgreiche Antwort des nginx HTTP-Servers des Containers $Xodx_1$. Allerdings untersucht Xodx in der Initialisierungsphase der PHP-Skripte zur Beantwortung einer Anfrage die abgefrage URI, um den Ort zu bestimmen, unter dem die Xodx-Applikation erreichbar ist (die *App Base*. Im hier beschriebenen Fall wird dann als *App Base* `localhost:8001` genutzt und alle Verweise und Weiterleitungen beziehen sich auf diesen Lokator. Allerdings hat dann mittlerweile der Hostname `localhost` seine Bedeutung geändert, denn wir befinden uns nun logisch innerhalb eines Docker Containers und nicht mehr im Host-System und innerhalb des Containers ist kein Prozess and den Port 8001 gebunden, so dass alle Weiterleitungen und interne REST-Anfragen innerhalb der Xodx-Instanz fehlschlagen.

Um diese Probleme zu umgehen, musste für jede Xodx-Instanz in einer Simulationsanordnung eine für die Dauer der Simulation im Internet hinweg eindeutige, kontextunabhängige URI für deren *App Base* eingerichtet werden und sichergestellt sein, dass diese URI korrekt von der Xodx-Instanz erkannt und genutzt wird.

Zwei generelle Schemata[6] zur Bildung dieser URIs wurden erwägt:

Subdomain pro Instanz , so dass die die k-te Xodx-Instanz unter `http://xodx`k`.dssn-sim.net` erreichbar ist (d.h. `http://xodx1.dssn-sim.net` für die erste Instanz). Diese Methode wäre mit erfordert von Xodx nur das korrekte Übernehmen des Hostnamens, so dass keine Änderungen an der entsprechenden Logik in Xodx erforderlich gewesen wären. Allerdings ist hier das Einrichten einer großen Anzahl an Subdomains erforderlich, für die man gegebenfalls von der Vefügbarkeit und Zustimmung eines oder mehrerer Systemadministratoren abhängig ist. Werden auch die PuSH-Hubs der Simulationsanordnung auf demselben Host-System betrieben, genügt die Gültigkeit der Subdomänen für die Namensauflösung auf ebendiesen System. Dann kann auch der lokale Einsatz von *dnsmasq* oder eines anderen selbst konfigurierbaren DNS-Servers auf dem Host-Systems für das Einrichten der Subdomains in Betracht gezogen werden[7]

[5] siehe Abschnitt 3.2 für Erläuterungen zum Begriff, Konzeption und Verantwortlichkeiten

[6] Für weitere Erläuterungen zur den Schemata und ihrer Umsetzung wird beispielhaft der Domänenname `dssn-sim.net` genutzt.

[7] Dies erfordert allerdings einige Änderungen an der Systemkonfiguration mit *root*-Berechtigungen und

Pfad-Präfix pro Instanz , so dass die k-te Xodx-Instanz unter `http://dssn-sim.net/xodx_`k erreichbar ist (d.h. `http://dssn-sim.net/xodx_1` für die erste Instanz). Hier genügt die bereits ein einzelner für das Host-System eingerichteter Domainname und die einige notwendige Änderung am der Systemkonfiguration, die *root*-Berechtigungen erfordert, ist gegebenfalls die *Site*-Konfiguration von *nginx*. Allerdings waren Anpassungen an Xodx und der nginx-Konfiguration der Xodx Docker Images erforderlich, damit auch bei diesem Schema das interne Routing korrekt funktioniert.

```
if (!empty($_SERVER['APP_BASE'])) {
  //if we receive an explicit setup for the app base, we use it
  $base_uri = $_SERVER['APP_BASE'];
} else {
  //otherwise attempt to construct it from default headers
  if (!empty($_SERVER['HTTPS']) && $_SERVER['HTTPS'] != 'off') {
    $protocol = 'https';
  } else {
    $protocol = 'http';
  }

  $app_path = $_SERVER['REQUEST_URI'];

  //trim trailing query string if present
  if(!empty($_SERVER['QUERY_STRING'])) {
    $qu_regex = '/'. preg_quote('?' . $_SERVER['QUERY_STRING'], '/') . '$/'
    $app_path = preg_replace($qu_regex, '', $app_path);
  }

  //remove PHP script name and remaining suffix after it, if present
  if(preg_match('/^.*?(\w+\.php.*)$/', $app_path, $script_match) === 1) {
    $app_path = str_replace($script_match[1], '', $app_path);
  }

  $base_uri = $protocol . "://" . $_SERVER['HTTP_HOST'] . $app_path;
}
```

Abbildung 3: überarbeitete PHP-Logik zur Festlegung der *App Base* in Xodx

Für das Testen bestehender Teile der Simulationsinfrastruktur wurde das Pfad-Präfix Schema gewählt. Es stellte sich allerdings heraus, dass die bestehende Initialisierungslogik von Xodx die *App Base* für diese URIs teilweise falsch erkannte. Im Speziellen war waren die Ergebnisse für die eigentlich funktioniell gleichwertigen HTTP GET Anfragen nach `http://dssn-sim.net/xodx_1/` und `http://dssn-sim.net/xodx_1/index.php` unterschiedlich. Das Vorgehen zur Erkennung der *App Base* in Xodx wurde zur Lösung dieses Problems überarbeitet und allgemein robuster für die Verarbeitung für Anfragen mit URIs unterschiedlicherer Gestalt gemacht. Im Rahmen dieser Überarbeitung wurde auch gleich eine alternative Möglichkeit für diese Routing-Problematik umgesetzt: Die Initialisierungslogik untersucht, ob ein spezieller HTTP-Header, `APP_BASE` gesetzt ist und übernimmt ggf. ohne das Anwenden von Heuristiken die dort angegebe URI als *App Base*. Dadurch

etwas vertieftes Verständnis über IP-Routing und DNS, da dies kein triviales Setup darstellt.

kann alternativ direkt in der Konfiguraion des HTTP-Servers die Information über den passenden Ort weitergegeben werden.

Für die Unterstützung des Pfad-Präfix Schema musste auch die *nginx*-Konfiguration für die Xodx-Container angepasst werden: Sofern eine Anfrage nicht von der eigentlich PHP-Applikation zu beantworten ist, sondern lediglich Resourcen (Bilddateien, CSS, ...) abruft, muss der Pfad-Präfix zur Identifikation der Instanz durch einen *rewrite* entfernt werden, da die Ordnerstruktur des Xodx-Projekts unabhängig von diesem Präfix bleiben soll.

```
set $index_redirect "0";
# check if the request is an exception and should not be handled by the index.php
if ($request_uri !~ (extensions|libraries.*)|(\.(js|ico|gif|jpg|png|css|swf|json))$) {
    set $index_redirect "1";
}
 if ($index_redirect = "0") {
    rewrite ^/xodx_[0-9]+(/.*)$ $1 last;
}
# rewrite all other URLs to index.php
if ($index_redirect) {
    rewrite ^.*$ /index.php last;
}
```

Abbildung 4: überarbeitete *site*-Konfiguration für den Xodx-Container, die für Anfragen nach Resourcen den '/xodx_*k*' Pfad-Präfix entfernt

Literatur

[1] Nathanael Arndt. „Xodx. Konzeption und Implementierung eines Distributed Semantic Social Network Knotens“. Masterarbeit. Universität Leipzig, 2013.

[2] Sören Auer, Sebastian Dietzold und Thomas Riechert. „OntoWiki–a tool for social, semantic collaboration“. In: *The Semantic Web-ISWC 2006*. Springer, 2006, S. 736–749.

[3] Tim Berners-Lee. „Linked data-design issues“. In: (2006). URL: `http://www.w3.org/DesignIssues/LinkedData.html`.

[4] B Fitzpatrick, B Slatkin und M Atkins. *PubSubHubbub protocol*. 2010. URL: `https://pubsubhubbub.googlecode.com/git/pubsubhubbub-core-0.4.html`.

[5] Manu Sporny u. a. „Webid 1.0: Web identification and discovery“. In: *Editor's draft, W3C* (2011). URL: `https://dvcs.w3.org/hg/WebID/raw-file/tip/spec/identity-respec.html`.

[6] Sebastian Tramp u. a. „An Architecture of a Distributed Semantic Social Network“. In: *Semantic Web Journal* Special Issue on The Personal and Social Semantic Web (2012).