Erfolg innovativer Open-Source-Ansätze am Beispiel asynchroner Servertechnologien

Michael Hitz, Thomas Kessel
DHBW-Stuttgart
Paulinenstraße 50
70178 Stuttgart

Abstract: Ist Open Source innovativ oder doch nur eine ”copy cat”? Viele Projekte im Open Source (OS) Umfeld haben zum Ziel, für bestehende Ansätze oder offene Standards eine Lösung ”nachzubauen” und damit eine in der Anschaffung kostengünstige Alternative zu kommerziellen Produkten und durch Beteiligung der Community gereifte Lösung anzubieten. Open-Source-Projekte können aber durchaus auch neue, innovative Ansätze hervorbringen und damit zum Innovationstreiber werden. Im Rahmen dieses Papiers betrachten wir die Konzepte des Projekts **Vert.x** im Bereich asynchroner Servertechnologien genauer und untersuchen aus technischer Sicht an einer konkreten Problemstellung, ob die dort eingeführten Ansätze für Alltagsprobleme im Unternehmenskontext nutzbringend eingesetzt und damit als Innovationserfolg gewertet werden können.

1 Einleitung

Das Open-Source-Konzept [OSI14] hat sich in den meisten Anwendungsbereichen etabliert und bietet Lösungen an, die kommerziellen Produkten ebenbürtig sind. Das aktuelle Portfolio geht von Betriebssystemen (z.B. Linux, Android), Anwendungssoftware (z.B. Typo3, Activiti, mySQL), bis hin zu Werkzeugen (z.B. Eclipse, Mantis) und Frameworks für die Softwareentwicklung (z.B. Spring, Hibernate). Der aktuelle Stand der wissenschaftlichen Forschung zu Open Source ist in [CWHW12] dokumentiert.

Viele Projekte haben dabei zum Ziel, bekannte und etablierte Technologien, Spezifikationen oder Standards als Open-Source-Lösung zu implementieren (z.B. JBoss für JEE, VLC media player für Multimedia-Player, Libre Office für Bürosoftware) und damit eine kostengünstige Alternative zu kommerziellen Produkten anzubieten. In einigen Fällen (z.B. Open Office) haben die Lösungen sogar ihren Ursprung im kommerziellen Umfeld und wurden zur weiteren Entwicklung der Open-Source-Community übergeben. Im Gegensatz zu kommerzieller Software ist die eingebundene Entwickler-Community für die nachhaltige Entwicklung der Open-Source-Software entscheidend [BGH+12].

Insbesondere Open-Source-Projekte im Bereich der Softwareentwicklung besitzen häufig aber auch innovativen Charakter. Die Innovation reicht dabei von Verbesserungen beste-

hender Ansätze bis hin zu gänzlich neuen Konzepten, die in kommerziellen Produkten noch nicht verfügbar sind. Ein prominentes Beispiel sind die Entwicklungen im Bereich der NoSQL-Datenbanken, in welchem verschiedenste Ansätze in den letzten Jahren entstanden [BBLW13].

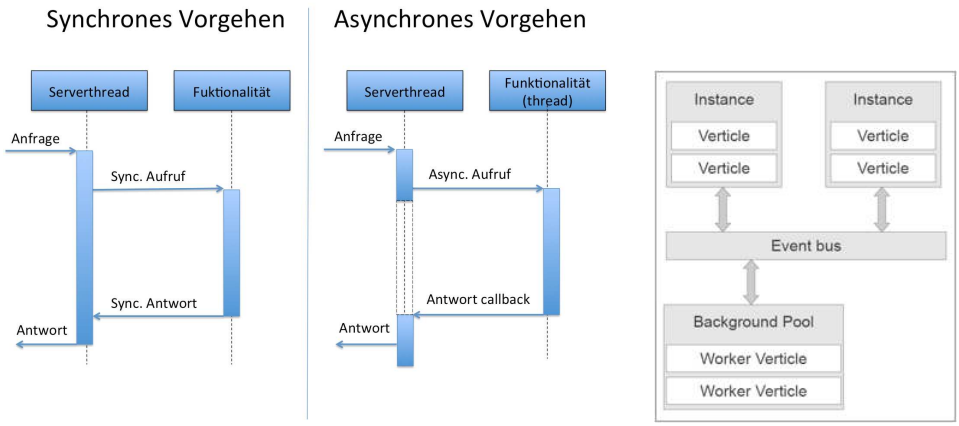
Während etablierte Open-Source-Produkte häufig durch Unternehmen oder Dachorganisationen wie Apache oder die Eclipse Foundation betreut werden und damit auf eine breite Community zählen können, beginnen innovative Projekte meist als kleine Projekte auf Open-Source-Portalen, z.B. http://sourceforge.net/ oder http://www.javaforge.com.

Ein innovativer Ansatz, der seinen Ursprung im Open-Source-Umfeld hat, sind asynchrone Frameworks zur Anwendungserstellung für webbasierte Anwendungen und Servertechnologien, die ebenfalls auf dem synchronen, event-getriebenen Vorgehen aufbauen. Die bekanntesten Produkte, die mittlerweile einen Einzug in die Praxis gefunden haben ([MHB14], [Wol13],[Gro13]), sind *Node.js* [Nod14] und *Vert.x* [Ver14b]. Der asynchrone Ansatz zur Abarbeitung von Anfragen verspricht eine optimalere Ausnutzung von Serverressourcen und kann so durch eine optimierte Nutzung der Serverressourcen zu Kosteneinsparungen beim Betrieb von Webanwendungen führen.

Im Rahmen des Projekts "Kompetenzzentrum Open Source" (KOS) an der DHBW Stuttgart wird für konkrete Open-Source-Produkte untersucht, inwiefern sie einen Nutzen im Unternehmensumfeld erbringen können. In diesem Kontext wurde **Vert.x** näher analysiert ([BBSW14]). Die Implementierung Vert.x wurde ursprünglich von Entwicklern der Firma VMWare initiiert und im August 2013 an die Eclipse Foundation übergeben. Obwohl der Verbreitungsgrad des Produkts noch nicht so hoch wie der von Node.js ist, verspricht es aber durch seine größere Sprachunabhängigkeit einen flexibleren Einsatz als Node.js, das auf JavaScript beschränkt ist.

In diesem Papier wird Vert.x als Vertreter einer innovativen Technologie technisch genauer betrachtet, für die es – nach unserem Kenntnisstand - keine vergleichbaren kommerziellen Produkte gibt. Es werden Optimierungsmöglichkeiten aufgezeigt, die mit dem neuen Ansatz erreicht werden können. Insbesondere steht die in der Praxis häufig auftretende Problemstellung einer zeitweiligen Blockierung bzw. hohen Auslastung von Webservern durch langlaufende Operationen im Vordergrund - die Lösung dieser Problemstellung stellt einen hohen praktischen Nutzen dar.

Der Einsatz innovativer Open-Source-Software in einem Unternehmen birgt eine Reihe von Risiken [HK13] - insbesondere in Hinsicht auf den Investitionsschutz. Die Kosten für die Einführung eines Open-Source-Produkts mit hohem Innovationsgrad erscheinen auf den ersten Blick gering, jedoch können sich die Folgekosten bei der Integration in die eigene Produktentwicklung als hoch erweisen, wenn man auf auf "das falsche Pferd" setzt und die Entwicklung des Open-Source-Produkts eingestellt wird. Mittels Open Source Maturity Models versucht man solche Risiken einzuschätzen. Fokus des vorliegenden Papiers sind jedoch die technischen Aspekte. Eine Betrachtung der Reife und Nachhaltigkeit kann anhand [GHJ12], [GKPP12] vorgenommen werden.



(a) Synchrones vs. asynchrones Vorgehen

(b) Kernelemente von Vert.x

Abbildung 1: Synchron vs. asynchron und Vert.x Kernelemente

2 Asynchrone Servertechnologien

Synchron vs. Asynchron

Die gängigsten, heute eingesetzten Web-/Applicationserver arbeiten nach einem synchronen Prinzip: wenn ein Request an den Server eingeht, wird dieser an einen Thread gebunden, der für die Abarbeitung zuständig ist. Während der gesamten Lebensdauer der Anfrage, ist dieser Thread somit belegt. Sobald die Anfrage bearbeitet ist, wird der Thread wieder freigegeben und kann weitere Anfragen bearbeiten.

Asynchrone Ansätze haben zum Ziel, diese Kopplung von Request und Thread, also deren Bindung bis zur vollständigen Abarbeitung auch aufwändiger Requests, aufzulösen. Dies wird durch eine ereignisgesteuerte Programmierung erreicht: es existiert ein Hauptthread, der Anfragen entgegennimmt. Operationen, die ausgeführt werden sollen, werden asynchron bearbeitet: das bedeutet, ihre Ausführung wird in eigenen Threads angestoßen, es wird aber nicht im Hauptthread auf ein Ergebnis gewartet. Es wird lediglich eine *callback*-Methode angegeben, die bei Beendigung der Operation gerufen wird. Dieses Ereignis kann dann die Weiterbearbeitung steuern (z.B. die Aufbereitung der Antwort an den Rufer). Indem auf diese Weise ein- oder ausgabeintensive Operationen von sog. Eventhandlern in nebenläufigen Threads bearbeitet werden, wird die Eventschleife nicht blockiert. Sie kann also weiterhin ihre Hauptfunktionen erfüllen: die Abfrage neuer Events und Aufruf des passenden Eventhandlers.

Abbildung 1(a) zeigt die unterschiedlichen Herangehensweisen in einem Ablaufdiagramm: während beim synchronen Vorgehen der Thread während der Abarbeitung des Requests ständig aktiv ist und auf die Antwort einer synchron gerufenen Funktionalität wartet (also faktisch blockiert ist), ist der Serverthread im Fall des asynchronen Aufrufs der Funktiona-

lität (und dessen Abarbeitung in einem eigenen Thread) frei für die Abarbeitung anderer Aufgaben und somit nicht blockiert. Erst wenn die gerufene Funktionalität abgearbeitet ist, wird der Serverthread wieder beansprucht.

Vert.x als konkrete Implementierung

Vert.x ist eine Implementierung des obigen asynchronen Konzepts, und basiert auf Java-Technologien. Es wird in einer Java Virtual Machine (JVM) ausgeführt und ein Vert.x Webserver kann damit mehrere Prozessorkerne nutzen, ohne dass der Benutzer sich um die Prozessverwaltung oder die Kommunikation zwischen den Prozessen kümmern muss [Ver14a]. Vert.x bezeichnet sich selbst als *polyglott*. Dies bedeutet, dass Anwendungen in mehreren Sprachen erstellt und auf dem Server ausgeführt werden können, sofern diese auf einer JVM umgesetzt werden können (z. Zt. sind dies Ruby, Groovy, Python, Java, JavaScript, Clojure) [Ver14b]. Diese größere Sprachenunabhängigkeit war entscheidend für die Auswahl von Vert.x für das Papier [BBSW14], auf dem die folgenden Untersuchungen basieren.

Abbildung 1(b) (entnommen aus [BBS+13]) illustriert das Kernkonzept von Vert.x: die Kernelemente von Vert.x sind sog. *Verticles*, die zu Modulen gepackt werden können und welche die Anwendungslogik implementieren. Mehrere Verticles können innerhalb einer *Vert.x-Instanz* ausgeführt werden, die wiederum in einer eigenen JVM-Instanz bzw. einem Prozessor ausgeführt wird. Vert.x ordnet so jedes Verticle genau einer Eventschleife zu und verhindert damit, dass eine Verticle-Instanz zur gleichen Zeit in mehr als einem Thread ausgeführt wird.

Für den Fall, dass ein Verticle ein- oder ausgabeintensiven (d.h. blockierenden) Code enthält, kann es als sogenanntes "Worker Verticle" deklariert werden. Es ist dann nicht der Vert.x-Eventschleife zugeordnet, sondern wird beim Aufruf in einem speziellen *Workerthread* abgearbeitet. Dadurch wird eine Blockade der Eventschleife verhindert [Ver14a].

Anwendungsbereiche asynchroner Ansätze

Die Implementierungen von Node.js und Vert.x basieren darauf, dass es nur *einen* Serverthread gibt, der alle eingehenden Anfragen beantwortet. Dieser erhält die gesamte Rechenleistung des Prozessors. Dies beschränkt zwar auf den ersten Blick die Nutzbarkeit des Ansatzes für Unternehmensanwendungen, die z.T. rechenintensive Operationen ausführen müssen - zumindest, wenn diese im Hauptthread ausgeführt werden müssten.

Aber die Voraussetzung für das asynchrone Vorgehen ist, dass die im Hauptthread gerufenen Operationen asynchron arbeiten können. Hierzu bieten die asynchronen Serverimplementierungen, insbesondere für die - aus der Laufzeitsicht - teuren Ein-/Ausgabeoperationen, bereits asynchrone Implementierungen an. Die Anwendungsbereiche, die für den Einsatz asynchroner Server darum häufig als optimal angegeben werden, beziehen sich meist auf die von den Implementierungen gelieferten Funktionalitäten: Ein-/Ausgabeoperationen, kurzlaufende Requests von Webanwendungen, Entgegennahme und Bereit-

stellung von großen Datenmengen (z.B. für Mediastreaming, Lieferung von Daten bzw. Dateien aus Objekt-Datenbanken) [Cap13], [BBS+13].

Im Unternehmensumfeld geht es allerdings häufig darum, bestehende Funktionalitäten aufzurufen, die nur auf den Backend-IT-Systemen des Unternehmens verfügbar sind, und diese Operationen sind z.T. (sehr) zeitaufwändig und somit langlaufend. In Artikeln und Literatur (z.B. [Rod12], [Cap13]) wird deshalb häufig das asynchrone Konzept als nicht geeignet für solche Anwendungsfälle bezeichnet (z.B. rechenintensive Anwendungen auf dem Server, Operationen auf relationalen Datenbanken). Dies gilt dann, wenn diese Aktionen innerhalb des Serverthreads ausgeführt werden sollen.

Die im vorangegangenen Abschnitt beschriebenen *Worker-Verticles*, die in Vert.x vorgesehen sind, gestatten es jedoch, langlaufende Operationen in externe Threads auszulagern und so diesen scheinbaren Nachteil auszugleichen.

So kann eine konkrete Problemstellung untersucht werden, die in der Praxis in Unternehmensportalen häufig auftritt - die temporäre Blockade von Webservern durch langlaufende Anfragen.

3 Nutzen in der Praxis

Auf Webservern, die Anfragen in großen Unternehmensportalen bearbeiten, treffen zahlreiche Requests zur gleichen Zeit ein. Üblicherweise sind die meisten gerufenen Aktionen dabei kurzlaufend (z.B. die Auslieferung einer angeforderten Seite oder das Absenden eines Mailformulars). Es gibt jedoch auch eine Reihe von Anfragen, bei denen die Antwort auf den Request des Nutzers einige Zeit in Anspruch nimmt. Meist sind hier Zugriffe auf das Backend der IT-Infrastruktur involviert, die eine längere Wartezeit erfordern.

Die aktuell verbreitet eingesetzten Webservertechnologien (z.B. Tomcat, JBoss, Apache Webserver) bearbeiten einen eingehenden Request synchron - in einem eigenen Thread. Hierzu steht ein Thread-Pool zur Verfügung, aus der - bei einer Anfrage - ein Thread für die Abarbeitung entnommen und danach wieder für andere Anfragen freigegeben wird.

Bei langlaufenden Anfragen kann diese Vorgehensweise dazu führen, dass ein Server weitgehend blockiert wird durch die vielen langlaufende Anfragen, denn der Server ist dann vorwiegend damit beschäftigt, auf die Antworten des Backends der IT-Infrastruktur zu warten. In dieser Zeit können somit keine weiteren Anfragen beantworten werden. Dies führt in ungünstigen Last-Situationen dazu, dass der Server alle weiteren Anfragen ablehnt und die Website nicht mehr erreicht werden kann, bis wieder neue Threads freigegeben werden können.

Das betrachtete Szenario sollen Self-Service-Anwendungen sein, wie sie in Versicherungs- oder Banking-Portalen häufig vorkommen. Hier existieren sowohl lang- als auch kurzlau-

fende Operationen (z.B. Informationen zu Produkten, Kontoübersicht, Durchführung einer Überweisung, Adressänderung). Folgende Rahmenbedingungen wurden hierfür aufgrund der Praxiserfahrungen zugrunde gelegt:

- Es existieren viele kurzlaufende Anfragen (z.B. die Anzeige von Produktinformation, die Adressänderung)
- Die Zahl der langlaufenden Anfragen ist gering, aber die Abarbeitungsdauer ist hoch, da zeitaufwändige Backendaufrufe notwendig sind (z.B. Durchführen einer Überweisung und gleichzeitige Vertragsanzeige)
- Langlaufende Operationen dürfen nicht zur Blockade des Servers führen.
- kurze Operationen haben Priorität. Bei Häufung langlaufender Operationen dürfen diese länger dauern. Kurze Operationen sollen aber nicht blockiert werden.

Für diese Konstellation wurde ein Lösungsansatz entworfen, der die asynchronen Features von Vert.x ausnutzt.

Lösungsansatz

Das in unseren Untersuchungen ([BBSW14]) zugrunde gelegte Prinzip nutzt die asynchrone Herangehensweise aus und vermeidet die Probleme, die zu einer Blockierung der Hauptschleife führen können.

Die Operationen wurden als Verticles in Java implementiert und als REST-ful Serviceaufrufe vom Server zur Verfügung gestellt.

Kurzlaufende Operationen wurden auf dem Server als *reguläre Verticles* konfiguriert und somit werden die von Vert.x bereitgestellten Mittel genutzt (z.B. optimierte I/O für statische Inhalte), jedoch nicht in Worker-Verticles ausgelagert. Zeitintensive Operationen werden als *Worker-Verticles* konfiguriert, die bei einem Aufruf in einem gesonderten Thread ausgeführt werden. Sie belasten so nicht den Serverthread, sondern sie werden zur Abarbeitung für einen eigenen Workerthread vorgemerkt, der dem Threadpool entnommen wird.

Diese Anordnung verspricht theoretisch, dass auf diese Weise in der Gesamtsicht keine Blockade des Hauptthreads auftreten kann (oder erst dann, wenn die Last auf den Server extrem hoch wird). Die kurzlaufenden Operationen werden unmittelbar ausgeführt, die Abarbeitung der langlaufenden Operationen wird jedoch nach außen verlagert.

Für die langlaufenden Operationen tritt dennoch dasselbe Problem auf, wie bei der Abarbeitung durch einen synchronen Server mit einer beschränkten Threadpool-Größe: bei Erreichung der Threadpool-Grenze können keine weiteren Anfragen abgearbeitet werden. Sie müssen warten, bis wieder ein Thread frei wird. Der Unterschied zum synchronen Verhalten liegt aber darin, dass der Server weiterhin kurzlaufende Anfragen entgegennehmen kann - aus Nutzersicht – ist er also nicht blockiert und somit weiterhin verfügbar.

Dies ist aus Sicht des Betriebs von webbasierten Anwendungen ein großer Vorteil, der ohne diesen Ansatz nicht erreicht werden könnte.

Erfolgsmessung: Vergleich des asynchronen und synchronen Ansatzes

In diesem Abschnitt werden die Ergebnisse der Untersuchungen aus [BBSW14] zusammengefasst. Eine detaillierte Betrachtung findet sich dort.

Zur Messung des Verhaltens von Webservern wurden für die Untersuchung zwei vergleichbare Infrastrukturen aufgebaut. Die synchrone Infrastruktur bestand aus einem Apache Tomcat-Server, welcher die Funktionalität in Form von REST-ful Services zur Verfügung stellte. Die asynchrone Implementierung wurde mit Vert.x umgesetzt, welche dieselbe Schnittstelle anbot. Um eine Vergleichbarkeit hinsichtlich der Threadzahl zu erreichen, wurde dem Apache Tomcat-Server eine feste Threadzahl zugewiesen. Dieselbe Anzahl an Threads wurde für den Threadpool für die langlaufenden Worker-Verticles verwendet.

Die Untersuchungen wurde auf einer relativ schwachen Hardwareausstattung durchgeführt; dies entspricht zwar nicht der Realität von Rechenzentren eines Unternehmens, sie zeigt aber deutlich die Skalierungseffekte, da der Sättigungsbereich des Servers bei Anfragen schneller erreicht wird. Um den Versuchsaufbau möglichst einfach zu halten, wurde aus diesem Grund zudem den Serverinstanzen nur jeweils ein einziger Prozessorkern zugewiesen. Da sowohl Tomcat als auch Vert.x mehrere Prozessorkerne unterstützen können, wird hiermit keiner Technologie ein potentieller Vorteil gegeben.

Die Beispielfunktionen wurden aus dem Bankenbereich gewählt. Abbildung 3 illustriert die gewählten UseCases. Für die Aktionen wurden feste Laufzeiten gewählt und so ein Satz von kurz- bzw. langlaufenden Aktionen bestimmt. Es wurde zudem ein Test-Client (s. Abbildung 2) entwickelt, mit dem die Anzahl der Requests und der Anteil der jeweiligen Operationen bestimmt und auf den Server geschickt werden können. So können bei den Simulationen Änderungen hinsichtlich des Verhältnisses von kurz- zu langlaufenden Operationen durchgeführt werden.

Die Messungen wurden für verschiedene Konfigurationen der Vert.x-Verticles vorgenommen (Verteilung von Verticles auf die Workerthreads) um die Effekte nachweisen zu können. Wir beschränken uns hier nur auf einen Teil der Szenarien aus [BBSW14].

Gemessen wurde die Durchlaufzeit der einzelnen Requestarten und damit das Antwortverhalten des Gesamtsystems für die einzelnen UseCases.

- **Konfiguration 1:** Alle Requestarten werden von Worker-Verticles verarbeitet. Die Zuteilung der Threads erfolgt nach Eingang der Anfragen.
- **Konfiguration 2:** Alle Requestarten werden von Worker-Verticles bearbeitet. Die Zuteilung der Threads erfolgt jedoch explizit. Hierbei wird jedem Verticle eine Anzahl von Threads aus dem Pool zur Verfügung gestellt. Langlaufende Operationen erhalten eine höhere Zahl von Threads. Die Tabelle in Abbildung 4 zeigt eine solche

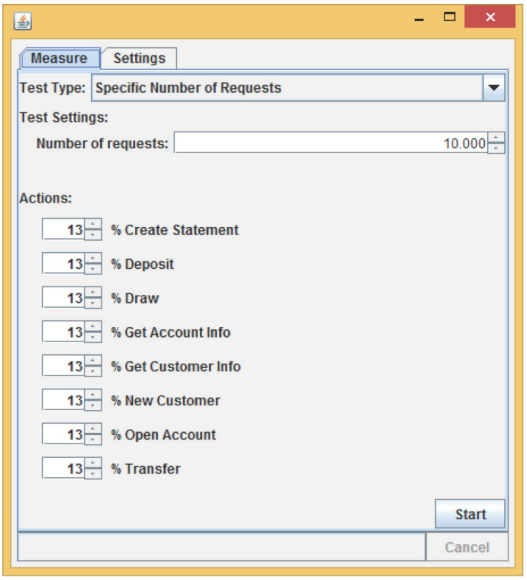


Abbildung 2: Benchmarktool zur Simulation von Anfragen

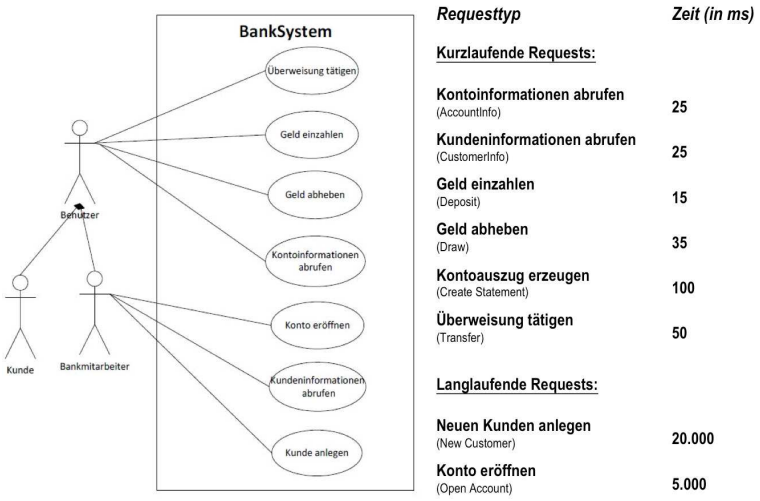


Abbildung 3: Gewählte UseCases und Laufzeiten in der Benchmarkanwendung

Zuordnung beispielhaft. Hier wird aufgrund der Abarbeitungszeit und des prozentualen Anteils an der Summe der Aufrufe eine Gewichtung vergeben, die zu einer Zuordnung von Threads für die Requestart führt.

Berechnung der Zuweisung der Workerthreads							
Requestart	**Ausarbeitungszeit**	**Anteile**	**"Gewicht"**	**Anteil am Gesamtgewicht**	**1 Thread je kurzem Typ**	**Aufteilung des Rests auf die langen Typen**	**Verteilung**
DepositWorker	15	15	225	0,03	1		**1**
AccountWorker	25	15	375	0,06	1		**1**
CustomerWorker	25	15	375	0,06	1		**1**
DrawWorker	35	15	525	0,08	1		**1**
TransferWorker	50	15	750	0,12	1		**1**
StatementWorker	100	15	1500	0,23	1		**1**
OpenAccountWorker	5000	5	25000	3,88	14	2,8	**3**
NewCustomerWorker	20000	5	100000	15,53		11,2	**11**
			128750				

Abbildung 4: Explizite Zuordnung von Workerthreads zu Transaktionen

- **Konfiguration 3:** Abarbeitung einiger kurzlaufenden Requestarten im Serverthread. Die verbleibenden Requests sind analog zu Szenario 2. Hierbei soll jedoch der Effekt untersucht werden, wenn der Hauptthread Aufgaben übernimmt, ohne dass hierzu eine Threadverwaltung eingeschaltet werden muss. Die könnte den Threadpool für sehr kurze Operationen entlasten.

Die Konfiguration 1 entspricht im Grundsatz der synchronen Vorgehensweise. Wie erwartet wurden hier keine Vorteile bei der Abarbeitung erzielt, da das Erreichen der Threadpoolgröße zum Warten auf einen freiwerdenden Thread für alle Requests führt. Allerdings begann der synchrone Server frühzeitig mit der Ablehnung von Anfragen und blockierte damit.

Die Ergebnisse der Konfiguration 2 zeigten signifikante Veränderungen in den Antwortzeiten, im Vergleich zu den Ergebnissen des synchronen Servers. Abbildung 5 stellt die Differenz der Antwortzeiten der asynchronen und synchronen Lösung für unterschiedliche Requestarten dar. Die *Deposit*-Operation ist explizit als ein Repräsentant für eine kurzlaufende Operation aufgeführt. Bei einer steigenden Anzahl von Anfragen verbesserte sich die Antwortzeit für kurzlaufenden Anfragen signifikant. Die langlaufenden Anfragen hingegen wurden langsamer abgearbeitet. Dieser Effekt kann gedämpft werden, wenn den langlaufenden Operationen anteilig mehr Threads zugewiesen werden. Auch hier begann der synchrone Server mit der (frühzeitigen) Ablehnung von Anfragen.

Die Konfiguration 3 brachte auf den ersten Blick sehr verblüffende Ergebnisse (vgl. Laufzeitdifferenzen in Abbildung 6). In dieser Konfiguration wurde lediglich die *Deposit* Anfrage als normales Verticle festgelegt und damit nicht mehr durch einen Workerthread abgearbeitet. Die sich ergebenden Verbesserungen waren signifikant - insbesondere auch für die langlaufenden Operationen. Die *Deposit*-Operation wurde erheblich schneller (da das Threadmanagement entfiel). Da diese Operation 15% der Anfragen ausmachte (vgl. Abbildung 4), wurde demnach in gleichem Maße Pool-Kapazität frei - was der Abarbeitung der Langläufer zugute kam.

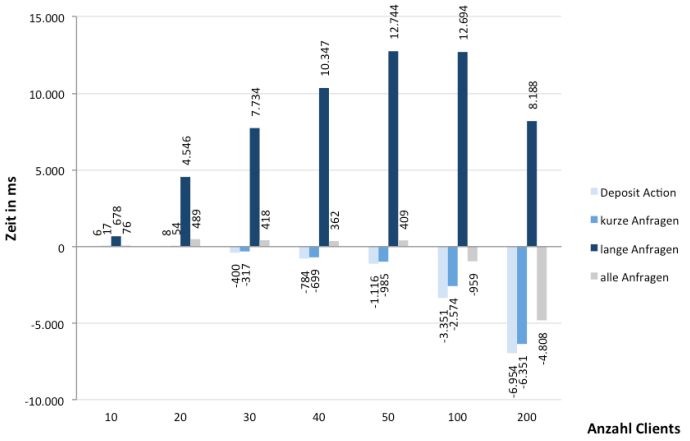


Abbildung 5: Vergleich Konfiguration 2 mit synchroner Verarbeitung

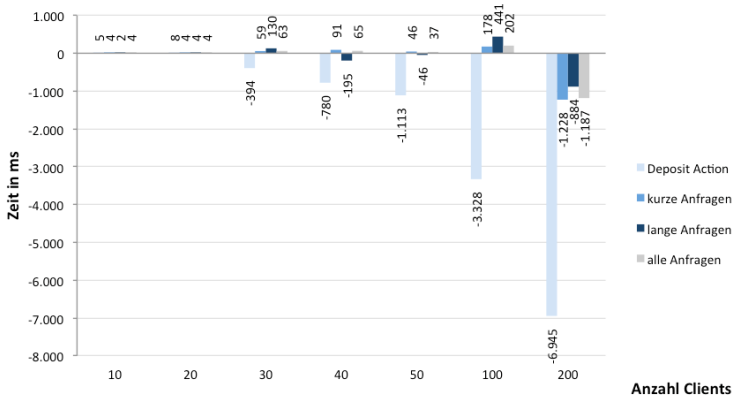


Abbildung 6: Vergleich Konfiguration 3 mit synchroner Verarbeitung

Die Ergebnisse unserer Untersuchungen geben Hinweise auf eine deutliche Verbesserung der Serverauslastung bei durchdachtem Einsatz von Vert.x. Durch die Konfiguration und die Verteilung von Aufgaben (in Form von Verticles) auf dedizierte Worker konnten schon Verbesserungen erreicht werden. Die deutlichste Verbesserung entstand aber in der Konfiguration, die kurzlaufende Operationen nicht dem Threadpool zuwies, da hier die Aufwände für die Threadverwaltung entfallen. Unsere formulierten Rahmenbedingungen konnten erfüllt werden und die Vermutung, dass sich durch den Einsatz eines asynchronen Vorgehens Vorteile ergeben, hat sich in der Versuchsreihe bestätigt. Um eine optimale Lösung für einen konkreten Anwendungsfall zu erreichen, bedarf es allerdings einer de-

taillierten Analyse der Gegebenheiten in einer konkreten Anwendung. Die Entscheidung, welche Aktivitäten in einen Threadpool ausgelagert werden sollen und wie viel Last der Serverthread verarbeiten kann, muss im Einzelfall ermittelt und eine entsprechende Konfiguration gefunden werden.

4 Zusammenfassung und Bewertung

Die Ergebnisse unserer technischen Untersuchungen geben einen deutlichen Hinweis, dass die im Open-Source-Umfeld entstandenen asynchronen Ansätze für die Umsetzung eines Servers eine Lösungsalternative zu den bestehenden synchronen Ansätzen im Webumfeld bieten. Auch wenn die Vorteile nur nach einer detaillierten Analyse des Anwendungsfalls gehoben werden können, so ermöglicht der Einsatz eines asynchronen Vorgehens eine erhebliche Verbesserung der Servernutzung und kann die Servicequalität eines Portals aus Nutzersicht erhöhen - der Ansatz stellt damit einen hohen Nutzen dar.

Im Rahmen dieses Papiers wurde lediglich auf den Nutzen des asynchronen *Ansatzes* aus technischer Sicht eingegangen und keine Bewertung des *Projekts Vert.x* hinsichtlich seiner Beständigkeit vorgenommen. Um die Einsetzbarkeit des konkreten Produktes im Unternehmensumfeld zu bewerten, kann eine Nutzwertanalyse des Projekts und ein Vergleich der Ergebnisse mit anderen Projekten angewendet werden ([GHJ12], [GKPP12]).

Die asynchronen Konzepte haben ihr Fundament im Open-Source-Umfeld. Wenn auch teilweise von kommerziellen Unternehmen initiiert, wurden die wichtigen Schritte als Open Source Projekt durchgeführt - was sicherlich auch zur Verbreitung und damit Stabilisierung der Ansätze geführt hat. Der Nutzen des Ansatzes ist im Unternehmenskontext hoch - und kann damit als echter Innovationserfolg der Open-Source-Gemeinde gewertet werden.

Literatur

[BBLW13] Teresa Bogolowski, Tana Brunner, Sophie Lingelbach und Christin Wattler. NoSQL Datenbanken - Typisierung, Seminararbeit DHBW-Stuttgart, 2013. In *Projektergebnisse des Kompetenzzentrums Open Source*, 2013.

[BBS+13] Can Paul Bineytioglu, Joe André Boden, Rocco Schulz, Max Vökler und Robert Wawrzyniak. Evaluation of Asynchronous Server Technologies, Seminararbeit DHBW-Stuttgart, 2013. In *Projektergebnisse des Kompetenzzentrums Open Source*, 2013.

[BBSW14] Christian Brummer, Michael Busam, Thomas Scherer und Matthias Welz. Asynchrone Servertechnologien zur optimalen Auslastung von Webservern, Seminararbeit DHBW-

Stuttgart, 2014. In *Projektergebnisse des Kompetenzzentrums Open Source*. Duale Hochschule Baden-Württemberg, Stuttgart, 2014.

[BGH+12] Robert Bruchhardt, Anne Golembowska, Maximilian Heinemeyer, Felix Kugler, Stephen Said und Jennifer Zohar. Beständigkeit eines Open Source Projektes - Analyse von Anerkennungssystemen in Open Source Projekten, Semnararbeit DHBW-Stuttgart, 2012. In *Projektergebnisse des Kompetenzzentrums Open Source*, 2012.

[Cap13] Tomislav Capan. Why The Hell Would I Use Node.js? A Case-by-Case Introduction - http://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js, 2013.

[CWHW12] Kevin Crowston, Kangning Wei, James Howison und Andrea Wiggins. Free/Libre open-source software development. *ACM Computing Surveys*, 44(2):1–35, Februar 2012.

[GHJ12] Franziska Gorhan, Juliana Hettinger und Maren Wolter Juliane Schulz. Development of a Model Evaluating the Maturity of Open Source Software, Seminararbeit DHBW-Stuttgart, 2012. In *Projektergebnisse des Kompetenzzentrums Open Source*, 2012.

[GKPP12] Anne Golembowska, Madeline Klink, Jana Petrovic und Daniel Prescher. Entwicklung eines Modells zur Bewertung von Open Source Produkten hinsichtlich eines produktiven Einsatzes, Seminararbeit DHBW-Stuttgart, 2012. In *Projektergebnisse des Kompetenzzentrums Open Source*, 2012.

[Gro13] Christian Gross. Node.js - hinter dem Hype. Sollte man sich als Java-Entwickler mit Node.js beschäftigen? *JavaMagazin*, 13(4), 2013.

[HK13] Michael Hitz und Thomas Kessel. Einfluss der Nutzung und Auswahl von Open Source Software beim Entwurf einer Multikanal-Architektur. In *GI-Jahrestagung 2013*, Seiten 1324–1338, 2013.

[MHB14] Sascha Möllering, Mariam Hakobyan und Björn Stahl. Vert.x im Unternehmenseinsatz - Entwicklung und Betrieb von asynchronen Applikationen mit Vert.x in der Praxis. *JavaMagazin*, 14(4), 2014.

[Nod14] Nodejs. Node.js Website - http://nodejs.org, 2014.

[OSI14] OSI (Open Source Initiative). The Open Source Definition – http://www.opensource.org/osd, 2014.

[Rod12] Golo Roden. *Node.js & Co - Skalierbare, hochperformante und echtzeitfähige Webanwendungen professionell in JavaScript entwickeln*. dpunkt.verlag, 2012.

[Ver14a] Vert.x. Vert.x Manual - http://vertx.io/manual.html, 2014.

[Ver14b] Vert.x. Vert.x Website - http://vertx.io, 2014.

[Wol13] Eberhard Wolff. vert.x – alles wird alles: Polyglott – asynchron – modular. *JavaMagazin*, 13(4), 2013.