Gut verknotet

Vert.x im Einsatz für hochskalierbare Architekturen

Rüdiger Grammes, Martin Lehmann, Kristine Schaal

Asynchrone, event-getriebene Architekturen haben in letzter Zeit stark an Bedeutung gewonnen. Treiber für diese Entwicklung ist das Web mit stetig steigenden Anforderungen an Performance, Skalierbarkeit und Ausfallsicherheit. Klassische Architekturen, die für jeden HTTP-Request einen Thread verwenden, stoßen schnell an ihre Grenzen. Das Vert.x-Framework kann mehr.

Vert.x im Überblick

In den letzten Jahren sind verschiedene Frameworks entstanden, die Requests asynchron mit einer geringen Anzahl Threads verarbeiten und dabei das Reactor-Pattern implementieren. Prominentester Vertreter ist Node.js, aber auch Akka, Rubys EventMachine sowie Netty und Reactor sind hier zu nennen. Deren grundlegende Philosophie beschreibt das [ReactiveManifesto].

Das Vert.x-Framework zieht aktuell viel Aufmerksamkeit auf sich. Als Open-Source-Projekt ist es mittlerweile bei der Eclipse Foundation (mit Apache License 2). Wir nutzen für unsere Beispiele, deren komplette Quellen Sie auf [SourcenGithub] finden, die Version 2.1 von Juni 2014 [Vertx].

Vert.x ist als asynchrones und event-getriebenes Framework auf Nebenläufigkeit, Performance und Skalierbarkeit optimiert. Das Programmiermodell basiert auf Callbacks, das Nebenläufigkeitsmodell ist an das Aktorenmodell [ActorModel] angelehnt. Nachrichten werden über einen Bus ausgetauscht, als Kommunikationsprotokolle stehen unter anderem TCP/SSL, HTTP/HTTPS, DatagramSockets und WebSockets zur Verfügung.

Vert.x erlaubt als polyglottes Framework den kombinierten Einsatz verschiedener JVM-Sprachen, neben Java auch JavaScript, Groovy, JRuby, Clojure und Scala.

Vert.x ist leichtgewichtig: Der Framework-Kern ist weniger als 1 MB groß, weitergehende Funktionalität kann über Module eingebunden werden. Web-Benchmarks bescheinigen Vert.x gute Performance-Werte im Bereich HTTP/Plain-Text und im Bereich JSON-Serialisierung [Techempower].

Ein erstes Beispiel: HTTP-Server

Starten wir mit einem Beispiel. Mit Maven generieren wir Projektstrukturen wie folgt:

```
mvn -e archetype:generate -Dfilter=io.vertx: -DgroupId=de.accso
  -DartifactId=simple-httpd -Dversion=0.1
```

Weitere Angaben zum Archetype geben wir interaktiv ein. Ergebnis ist ein komplett vorbereitetes Projekt, in dem die Abhängigkeiten zu Vert.x sowie andere nützliche Strukturen wie Vert.x-Maven-Plug-in, Testintegration usw. bereits voreingestellt sind.

Wir implementieren als erstes Beispiel einen einfachen HTTP-Server, siehe Listing 1:

```
 1: import org.vertx.java.platform.Verticle;
 2: import org.vertx.java.core.http.HttpServerRequest;
 3:
 4: public class HelloWorldHttpServer extends Verticle {
 5:   @Override
 6:   public void start() {
 7:     vertx.createHttpServer().requestHandler(
 8:         (HttpServerRequest req) -> {
 9:       logInfoString(" received request " + req.remoteAddress());
10:       req.response().putHeader("content-type", "text/plain")
11:          .end("Hello World!");
12:     }).listen(8123);
13:
14:     logInfoString("registered on port 8123");
15:   }
...
// logInfoString (loggt String und Thread-Name) nicht weiter ausgefuehrt
...
```

Listing 1: Einfacher HTTP-Server als Verticle

- Unser **`HelloWorldHttpServer`** leitet von der Framework-Klasse **`Verticle`** ab (Zeile 4). Verticles sind die Basiskomponenten von Vert.x, somit sowohl Strukturvorgabe für den technischen Komponentenschnitt als auch ausführbare Laufzeit-Einheiten.
- Die Lifecycle-Methode **`start`** (Zeile 6) wird durch die Laufzeitumgebung automatisch beim Deployment aufgerufen.
- **`HelloWorldHttpServer`** hat über die vererbte Variable **`vertx`** Zugriff auf das Vert.x-API, mit der wir einen HTTP-Server auf Port 8123 erzeugen (Zeile 12).
- HTTP-Requests beantwortet unser Verticle mit **`Hello World!`**. Wir übergeben dazu einen Callback-Handler (Zeile 8), den die Laufzeitumgebung für eingehende Requests aufruft. Für die vielen Callbacks bietet sich ab Java 8 die Verwendung von Lambda-Ausdrücken [GüLe13] an, da andernfalls Programme schnell unübersichtlich werden.
- **`logInfoString`** nutzt den geerbten **`container`** als Referenz auf die Laufzeitumgebung und schreibt dabei auch den Namen des Threads, in dem das Verticle abläuft.

Verticles können über API-Methoden eingebettet gestartet werden, alternativ direkt in der Konsole über **`vertx run HelloWorldHttpServer.java`**. Vert.x übernimmt dann sogar das Kompilieren des Quellcodes!

Statische Sicht: Verticles, Instanzen, Cluster

Abbildung 1 zeigt die statische Komponentensicht von Vert.x:

- Verticles sind die Laufzeitkomponenten einer Anwendung in einer Vert.x-Instanz.
- Verticles können einzeln oder zu Modulen gruppiert ablaufen.

- Es gibt zur Laufzeit eine implizite Hierarchie durch das Deployment: Wird ein Verticle undeployt, werden alle von diesem gestarteten Verticles ebenfalls undeployt.
- Verticles teilen sich innerhalb einer Instanz einen Shared-Data-Bereich.
- Instanzen können einen Cluster über Prozess- beziehungsweise Maschinengrenzen bilden.

Dynamische Sicht: Verticles laufen auf Event-Loop-Threads

Beim Start einer Instanz erzeugt Vert.x einen Thread-Pool – dessen Größe ist die Anzahl der Maschinen-CPUs/Cores. Auf dessen Threads, den sogenannten Event-Loop-Threads, werden die Verticles ausgeführt. Dabei garantiert die Laufzeitumgebung zwei wichtige Eigenschaften:

- Jedes Verticle erhält einen eigenen Classloader, sodass kein globaler Zustand (selbst nicht in static-Variablen) zwischen Verticles geshared werden kann.
- Jedes Verticle wird immer im selben Event-Loop-Thread ausgeführt, womit Cache-Zugriffe optimiert werden. Obige `logInfoString`-Methode gibt also immer den gleichen Thread-Namen aus.

Ein Verticle kann folglich so implementiert werden, als ob es single-threaded ausgeführt würde. Die sonst mit Nebenläufigkeit verbundene Komplexität – Synchronisierung und Sichtbarkeit von State, dadurch Gefahr von Deadlocks usw. sowie Performance-Auswirkungen (vgl. [Disruptor], [Mader]) – wird durch dieses Programmiermodell komplett vor dem Anwendungsentwickler verborgen.

Ein Event-Loop-Thread führt mehrere Verticles aus und ruft bei eingehenden Events das entsprechende Verticle beziehungsweise dessen Event-Handler auf. Abbildung 2 zeigt das am Beispiel von vier Event-Loop-Threads mit „ihren“ Verticles.

Diese erweiterte Implementierung des Reactor-Patterns wird in Vert.x „Multi-Reactor“-Pattern genannt und erlaubt es, sehr viele Tausend parallel eingehende Requests effizient abzuarbeiten. Mit einer „klassischen“ Thread-per-Request-Architektur stieße man aufgrund der Anzahl maximal möglicher Threads und deren Memory-Footprints schnell an technische Grenzen.

Ein Verticle wird nicht unterbrochen, muss also möglichst kurz ablaufen, um „seinen“ Event-Loop-Thread schnell wieder freizugeben! Blockierende Aktionen müssen unbedingt vermieden werden, da diese den Event-Loop-Thread belegen, was den Durchsatz des Systems massiv beeinträchtigt. Zu vermeiden sind beispielsweise `Thread.sleep`, lange Berechnungen und blockierende Zugriffe durch Dritt-Bibliotheken zum Beispiel über JDBC.

Sind solche blockierende Aktionen unvermeidbar, so müssen sie stattdessen in Worker-Verticles ablaufen. Deren Programmiermodell gleicht dem normaler Verticles. Ihre Ablaufumgebung ist aber ein eigener Worker-Thread-Pool, der von den Event-Loop-Threads getrennt ist. Auch Worker-Verticles werden nie durch mehr als einen Thread parallel ausgeführt, die Ausführung durch immer denselben Thread ist aber nicht garantiert.

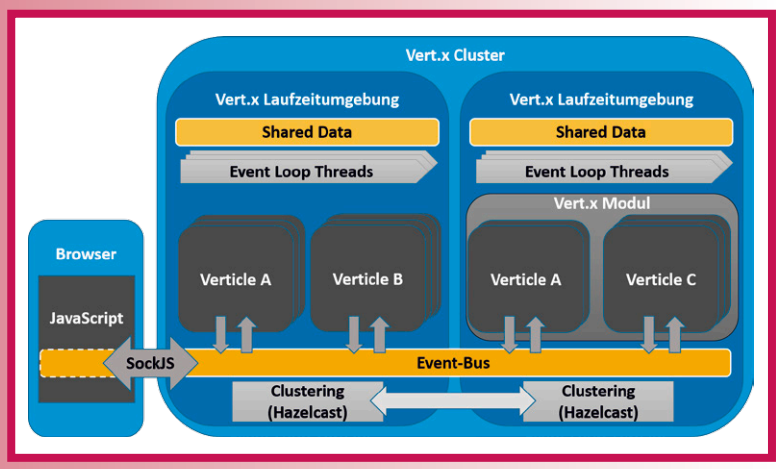

Abb. 1: Statische Sicht: Verticles, Instanzen, Cluster

Abb. 2: Dynamische Sicht: Verticles laufen auf Event-Loop-Threads

Messaging über den Event-Bus

Verticles laufen zur Laufzeit komplett getrennt. Die Kommunikation zwischen Verticles erfolgt ausschließlich durch Nachrichten über den Event-Bus, dem „zentralen Nervensystem“ von Vert.x.

Ein Verticle kann sowohl eine Nachricht unter Angabe einer Adresse erstellen als auch Nachrichten für eine Adresse empfangen. Adressen sind einfache Strings ohne weitere Strukturvorgabe.

Eine Nachricht wird an alle Verticles, die sich unter dieser Adresse registriert haben, adressiert – unabhängig davon, ob der Empfänger in der gleichen Instanz oder im Cluster läuft. Der Nachrichtenaustausch erfolgt asynchron. Die Zustellung einer Nachricht wird nicht garantiert.

Vert.x bietet diese Kommunikat ionsformen:

- Publish-Subscribe: Eine mit `publish` verschickte Nachricht wird allen Subscribern zugestellt (d. h. allen für diese Adresse registrierten Verticles).
- Punkt-zu-Punkt: Eine mit `send` verschickte Nachricht wird nur einem Empfänger aus der Subscriber-Liste zugestellt. Die Auswahl erfolgt per Round-Robin. Ein gezielter Nachrichtenversand an ein bestimmtes Verticle wird nicht unterstützt.
- Request-Reply: Auf eine Punkt-zu-Punkt-Nachricht kann dem Sender mit `reply` geantwortet werden.

Ping-Pong über den Bus

Der Maven-Archetype generiert neben den Projektstrukturen auch gleich ein ausführbares Ping-Beispiel für Java sowie für Groovy, JavaScript und Python samt Tests. Dieses Beispiel greifen wir nun auf.

```
 1: public class PongVerticle extends Verticle {
 2:   @Override
 3:   public void start() {
 4:     // Handler fuer "Ping"-Nachrichten
 5:     Handler<Message<String>> handler =
 6:           new Handler<Message<String>>() {
 7:       public void handle(Message<String> message) {
 8:         logInfoString(" receives message " + message.body());
 9:         vertx.eventBus().send("pong-address", "pong");
10:       }
11:     };
12:     // Handler wird fuer "Ping"-Nachrichten registriert
13:     vertx.eventBus().registerHandler("ping-address", handler);
14:   }
...
```

Listing 2: PongVerticle

Listing 2 zeigt unser **PongVerticle**:

- Es registriert einen Nachrichten-Handler für die Adresse **ping-address** und empfängt dahin geschickte Ping-Nachrichten (Zeilen 5-7). Will man sichergehen, dass die Registrierung in allen Instanzen des Clusters bekannt geworden ist, kann man **registerHandler** dafür einen weiteren Handler mitgeben.
- Unser Verticle schreibt jede eintreffende Nachricht ins Log (Zeile 8: Zugriff über **message.body**) und schickt danach eine Nachricht nach **pong-address** (Zeile 9).

Verschiedene Typen sind für den Nachrichten-Payload nutzbar, darunter String, Long, Integer, Double, Number sowie JSON. Empfohlen ist JSON, da dies durch alle unterstützten Sprachen am einfachsten verarbeitbar ist (in Java mit [Jackson]).

```
 1: public class PingVerticle extends Verticle {
 2:   @Override
 3:   public void start() {
 4:     // Handler fuer "Pong"-Nachrichen
 5:     Handler<Message<String>> handler =
 6:       new Handler<Message<String>>() {
 7:       public void handle(Message<String> message) {
 8:         logInfoString("receives message " + message.body());
 9:         Handler<Message<String>> replyHandler = this;
10:
11:         // erneuten Ping nach 1s rausschicken
12:         vertx.setTimer(1000, new Handler<Long>() {
13:           public void handle(Long event) {
14:             vertx.eventBus().send("ping-address",
15:                   "ping", replyHandler);
14:           }
15:         });
16:       }
17:     };
18:     // Handler wird fuer "Pong"-Nachrichten registriert
19:     vertx.eventBus().registerHandler("pong-address", handler);
20:
21:     // Initiale Nachricht an alle PongVerticles
22:     vertx.eventBus().send("ping-address", "ping", handler);
23:   }
...
```

Listing 3: PingVerticle

Unser **PingVerticle** zeigt Listing 3:

- Nach Start des **PingVerticle** wird die initiale Ping-Nachricht verschickt (Zeile 22). Passiert dies, bevor ein **PongVerticle** seinen Handler auf **ping-address** registrieren konnte, so geht die Nachricht verloren und ein Nachrichtenaustausch kommt nicht zustande.
- Der in Zeile 5 definierte Nachrichten-Handler reagiert auf die Pong-Nachrichten des **PongVerticle**. Dieser Handler ist gleichzeitig ein Reply-Handler, der Antworten auf ein Ping (wie vom später folgenden JavaScript-Beispiel) verarbeiten kann, und wird bei **send** mitgegeben (Zeilen 9, 15, 22).
- Auch unser **PingVerticle** loggt Nachrichten, bevor es einen Vert.x-Timer startet (Zeile 12), der nach einer Sekunde eine weitere Ping-Nachricht auf den Bus schickt (Zeilen 14-15). Hätten wir anstelle des Timers den Thread mit **Thread.sleep(1000)** schlafen gelegt, so hätte dies den Event-Loop-Thread des **PingVerticle** blockiert!

Cluster mit Hazelcast

Wir starten zwei **PongVerticle** in derselben Vert.x-Instanz, danach ein **PingVerticle** in einer zweiten Instanz. Damit sich diese beiden Instanzen finden, müssen sie als Cluster gestartet werden:

```
vertx run PongVerticle.java –cluster –instances 2
vertx run PingVerticle.java –cluster
```

Ping-Nachrichten werden über Round-Robin an die beiden **PongVerticle** verteilt, was die Log-Ausgaben der ersten Instanz gut zeigen.

Vert.x setzt als Cluster-Technologie mit Hazelcast eine hochskalierbare Plattform ein [Hazelcast]: Für die Datenverteilung bietet Hazelcast unter anderem Maps, Sets, Lists und Queues an, deren Inhalte in Cluster-Instanzen verteilt genutzt werden können. Diese Datenstrukturen kann man so verwenden, wie man das von „normalen“ Java-Datenstrukturen gewohnt ist. Bei Start oder Stopp von Hazelcast-Knoten zur Laufzeit sorgt Hazelcast automatisch dafür, dass alle Knoten dies erfahren.

Vert.x im Cluster-Modus benutzt Hazelcast unter der Haube:

- Jede Vert.x-Instanz entspricht einem Hazelcast-Knoten.
- Auch die High-Availability-Funktionalität von Vert.x basiert auf Hazelcast.
- Vert.x verwendet eine Hazelcast-MultiMap (Map mit mehreren Values pro Key): Dort ist für die Instanzen sichtbar, welcher Handler auf welchem Host und Port erreichbar ist.

Vert.x‘ Bus-Kommunikation basiert dagegen nicht auf Hazelcast, ebenso wenig das Shared-Data-Feature: Daten in Shared-Data-Maps oder -Sets können nur innerhalb einer Vert.x-Instanz geshared werden (nicht im Cluster).

Die Log-Ausgaben unserer beiden Instanzen zeigen nicht nur das Ping-Pong-Messaging, sondern – bei entsprechender Log-Konfiguration – auch Hazelcasts Log-Ausgaben unseres Clusters:

- Der erste Hazelcast-Knoten startet auf Port 5701 die Vert.x-Instanz mit den beiden **PongVerticle** (Log-Ausgabe „**Members [1]**“).
- Später kommt mit der Vert.x-Instanz mit dem **PingVerticle** ein zweiter Hazelcast-Knoten auf Port 5702 hinzu (Ausgabe „**Members [2]**“). Immer wenn ein Hazelcast-Knoten hinzukommt, verteilt Hazelcast die von ihm verwalteten Daten neu auf alle Knoten (Log-Ausgabe „**Re-partitioning cluster-data**“).
- Die Log-Ausgabe zeigt außerdem, dass Hazelcast über die Default-Einstellung Multicast kommuniziert („**Creating MulticastJoiner**“). Alternativ sind TCP/IP oder EC2 Auto Discovery nutzbar.

Viel hilft viel?

Wir ergänzen unser Ping-Pong-Beispiel um weitere Aspekte, um die technischen Grenzen des Nachrichtenversands auszuloten. Wir zeigen nur das erweiterte **PingVerticle** (s. Listing 4, das **PongVerticle** ist fast unverändert):

- Änderung des Nachrichtentyps: Unsere Nachrichten sind nun keine Strings mehr, sondern enthalten als **JsonObject** ID und Zeitstempel.
- Nutzung von Shared-Data: Wir zählen im **PingVerticle** mit, wie viele Antwort-Nachrichten für eine ID schon eingetroffen sind, und schreiben die Information in eine Shared-Data-

```
1: // Shared-Data-Map, die zu jeder Pong-Id die Messages mitzaehlt
2: final ConcurrentSharedMap<Integer, Long> messageCountingMap =
3:   vertx.sharedData().getMap(COUNTING_MAP);
4:
5: Handler<Message<JsonObject>> handler=
6:   new Handler<Message<JsonObject>>(){
7:   public void handle(Message<JsonObject> message) {
8:     final Integer id = message.body().getInteger(JSON_ATTR_ID);
9:
10:    // Zaehler fuer Id erhoehen
11:    Long idCounter = messageCountingMap.get(id);
12:    if (idCounter==null) idCounter = new Long(0);
13:    messageCountingMap.put(id, idCounter+1);
...
```

Listing 4: Shared-Data-Nutzung im PingVerticle

Map (Zeilen 2-3, 11-13). Ein weiteres Verticle in der gleichen Vert.x-Instanz kann für Monitoring-Zwecke diese Zähler auslesen und z. B. als Webseite darstellen. Um bei Shared-Data Threading-Probleme zu verhindern, erlaubt Vert.x nur die Verwendung von Immutable-Datentypen.

Den Kommunikationstyp ändern wir auf Publish-Subscribe: Unser **`PingVerticle`** schickt nun keine Nachrichten per **`send`**, sondern per **`publish`** an alle **`PongVerticle`** (im Listing nicht weiter ausgeführt):

- Wie gehabt wird initial eine Ping-Nachricht mit ID 1 verschickt. Darauf antwortet nun jedes **`PongVerticle`** (zuvor nur eines, da die Pings per Round-Robin verteilt wurden).
- Das **`PingVerticle`** empfängt die Antworten und schreibt für jede Antwort nach einer Sekunde Wartezeit erneut ein Publish mit ID+1 auf den Bus.
- Starten wir zwei **`PongVerticle`**, so verschickt unser **`PingVerticle`** also erst eine Nachricht und erhält dafür zwei Antworten. Für beide Antworten erfolgt wieder ein Publish mit vier Antworten, danach vier Nachrichten mit acht Antworten usw. Wir erzeugen so eine exponentiell wachsende Zahl von Bus-Nachrichten. Ein solches Nachrichtenwachstum ist sicherlich ein künstliches Beispiel, soll Vert.x aber gezielt an seine technischen Grenzen bringen.

In Tests mit drei getrennten Instanzen (eine Maschine, Kommunikation also lokal) konnten wir so ca. 1 Mio. Nachrichten pro Minute erzeugen, bevor nach ca. 10 Minuten die Prozesse derart überlastet waren, dass sie „zusammenbrachen“ und von Hazelcast und Vert.x für „tot“ erklärt wurden.

Gegen derartige Ausfälle von Vert.x-Instanzen schützt bei Bedarf die High-Availability-Funktionalität: Dafür müssen mehrere Vert.x-Instanzen mit der Option **`-ha`** gestartet werden und die Anwendung muss als Modul aus einem Repository deployt werden. Erkennt Hazelcasts Cluster-Manager den Ausfall einer Vert.x-Instanz, so werden dessen Module automatisch auf einer noch laufenden Instanz neu gestartet. Daten der ausgefallenen Instanz gehen verloren. In unseren Tests führten Abbruch und Neustart eines **`PongVerticle`** zu einem Verlust von ca. 10.000 Nachrichten.

Esperanto auf der JVM: Vert.x ist polyglott

Verticles können auch in anderen Programmiersprachen als Java implementiert werden: Vert.x 2.1 unterstützt JavaScript und CoffeeScript mit Rhino sowie JRuby, Groovy, Jython, Scala und Clojure. Experimentell sind PHP sowie JavaScript mit Nashorn (ab Java 8) und DynJS.

Ein **`PongVerticle`** in JavaScript zeigt Listing 5. Es beantwortet eintreffende Ping-Nachrichten mit **`reply`**.

```
1: var vertx = require('vertx')
2: var console = require('vertx/console')
3: vertx.eventBus.registerHandler('ping-address',
4:   function(message, replier) {
5:     replier('pong-js!');
6: });
```

Listing 5: PongVerticle in JavaScript

Gruppierung von Verticles zu Modulen

Bisher haben wir Verticles nur einzeln deployt. Verticles können in Module zu größeren, versionierbaren Einheiten gruppiert werden. Die nötigen Artefakte werden durch Maven-Archetype generiert, einschließlich des Moduldeskriptors als JSON-Datei.

Der Maven-Build erstellt daraus ein ausführbares Modul in Form einer Zip-Datei, startbar mit: **`vertx runzip simple-ping-pong-0.1-mod.zip -cluster`**. Dies startet das im Deskriptor definierte „Main“-Verticle, löst beschriebene Abhängigkeiten zu anderen Modulen auf und beachtet weitere Konfigurationseinstellungen.

Bus-Nachrichten mit WebSockets

Bus-Nachrichten können auch im Browser in JavaScript über WebSockets [GrGüLe13] verarbeitet werden. Basis ist SockJS, das ein full-duplex WebSocket-Protokoll auf HTML5-Basis implementiert.

Dazu muss zunächst ein Verticle als Bridge zwischen Event-Bus und SockJS gestartet werden:

- Als Bridge nutzen wir das bestehende Modul **`mod-web-server`** [VertxModWebserver], das wir aus einem JavaScript-Verticle heraus starten (Listing 6, Zeile 2).
- Über die JSON-Konfiguration aktivieren wir die Bridge und schalten nötige Adressen explizit frei (Zeilen 3-5). Andernfalls verweigert Vert.x aus Sicherheitsgründen den Zugriff auf die entsprechenden Bus-Adressen.
- JavaScript im Browser kann sich mit der Bridge verbinden (Listing 7, Zeile 4) und dann Bus-Nachrichten empfangen und versenden (Zeilen 6-7).

Neben **`mod-web-server`** existieren weitere vorgefertigte Module, zum Beispiel für Datenbank-Anbindungen und für Protokolle wie Mail oder FTP. Zentrale Modul-Registry ist **`http://modulereg.vertx.io/`**. Für die Modul-Ablage sind Maven-Repositories sowie Bintray eingebunden, alternativ auch lokale oder firmenweite Maven-Repositories.

```
1: var container = require('vertx/container');
2: container.deployModule("io.vertx~mod-web-server~2.0.0-final", {
3:     port: 8645, bridge: true,
4:     inbound_permitted: [ { address: 'pong-address' } ],
5:    outbound_permitted: [ { address: 'ping-address' } ]
6: });
```

Listing 6: Bridge zwischen Bus und SockJS

```
1: <script src='http://cdn.sockjs.org/sockjs-0.3.4.min.js'></script>
2: <script src='vertxbus-2.1.js'></script>
3: <script>
4: var bus = new vertx.EventBus('http://localhost:8645/eventbus');
5: bus.onopen = function() {
6:   bus.registerHandler('ping-address',function(message, replier) {
7:     bus.send('pong-address','pong-browser-js!');
8:   });
9: }
10: </script>
```

Listing 7: Verticle im Browser

Fazit und Ausblick

Vert.x ist bereits ein sehr mächtiges Framework und wurde daher unter anderem mit dem JAX Innovation Award 2014 für die innovativste Java-Technologie ausgezeichnet. Vert.x 3.0 als das nächste große Meilenstein-Release befindet sich aktuell noch in einem frühen Stadium. Neben internen Umstrukturierungen und Verbesserungen sind diese Erweiterungen absehbar:

- Clusterübergreifendes Shared-Data, Clusterfähigkeit über LAN-Grenzen hinweg,
- Umbau und Vereinfachung des Modulsystems,
- Monitoring-Support für den Event-Bus zum Beispiel über JMX,
- Verschlüsselung und Flusskontrolle für den Event-Bus.

Eine vollständige Liste geplanter Änderungen findet sich in [Vertx3].

Literatur und Links

[ActorModel] `https://en.wikipedia.org/wiki/Actor_model`

[Disruptor] M. Thompson u. a., Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads, Mai 2011, `https://lmax-exchange.github.io/disruptor/`

[GrGüLe13] R. Grammes, M. Günther, M. Lehmann, Keine Einbahnstraße: WebSockets in Java EE 7, in: JavaSPEKTRUM, 6/2013

[GüLe13] M. Günther, M. Lehmann, Ausdrucksstark: Lambda-Ausdrücke in Java 8, in: JavaSPEKTRUM, 3/2013

[Hazelcast] Cluster-Bibliothek Hazelcast – Projekt-Homepage, `http://hazelcast.com/`

[Jackson] Json-Bibliothek Jackson – Projekt-Homepage auf Github, `https://github.com/FasterXML/jackson`

[Mader] J. Mader, Lockperformance – Project to reproduce the numbers presented in [Disruptor], `https://github.com/codepitbull/lockperformance`

[ReactiveManifesto] `http://www.reactivemanifesto.org/`

[SourcenGithub] Alle Quellcode-Beispiele auf Github `https://github.com/accso/vertx-examples` oder `http://www.sigs-datacom.de/nc/fachzeitschriften/javaspektrum/archiv.html`

[Techempower] Performance-Benchmarks für Web-Frameworks, `http://www.techempower.com/benchmarks/`

[Vertx] Vert.x Homepage, `http://vertx.io/`

[Vertx3] Geplante Features für Vert.x 3.0, `https://github.com/eclipse/vert.x/wiki/Vert.x-3.0-plan`

[VertxModWebserver] Webserver-Modul für Vert.x, `https://github.com/vert-x/mod-web-server`

Rüdiger Grammes ist seit November 2011 als Managing Consultant bei der Accso – Accelerated Solutions GmbH und dort als Softwarearchitekt in verschiedenen Projekten unterwegs.
E-Mail: grammes@accso.de

Martin Lehmann ist Diplom-Informatiker und als Cheftechnologe und Softwarearchitekt bei der Accso – Accelerated Solutions GmbH tätig. Seit Ende der 90er-Jahre arbeitet er in der Softwareentwicklung in diversen Projekten der Individualentwicklung für Kunden verschiedener Branchen.
E-Mail: lehmann@accso.de

Kristine Schaal ist als Softwarearchitektin bei der Accso – Accelerated Solutions GmbH tätig. Sie arbeitet seit fast 20 Jahren in der Softwareentwicklung und ist in Projekten der Individualentwicklung für Kunden verschiedener Branchen unterwegs.
E-Mail: schaal@accso.de