Evaluation of Asynchronous Server Technologies

Can Paul Bineytioglu, Joe André Boden, Rocco Schulz,

Max Vökler Robert Wawrzyniak

January 19, 2013

Corporate State University
Baden-Wuerttemberg - Stuttgart

Abstract: This paper evaluates asynchronous server technologies using the more recent frameworks Node.js and Vert.x as references. Strenghts and weaknesses of asynchronous programming models are elaborated to identify areas where these technologies should be chosen over the common programming frameworks. A proof of concept based on Node.js and Vert.x is used to evaluate non-functional attributes such as maintainability and integration into an existing application landscape.

Contents

List of Tables

List of Figures

[1]**g_trends**.

List of Listings

1 Introduction

In traditional web application development data is transmitted synchronously, i.e. upon a GET/-POST request the result can be displayed only after transmission and processing are finished, as highlighted in Fig. 1[2]. While maintaining simplicity and predictability this can cause serious latency when uploading large pieces of data most commonly complex forms for registration. Naturally rich content such as images and videos causes even more waiting.

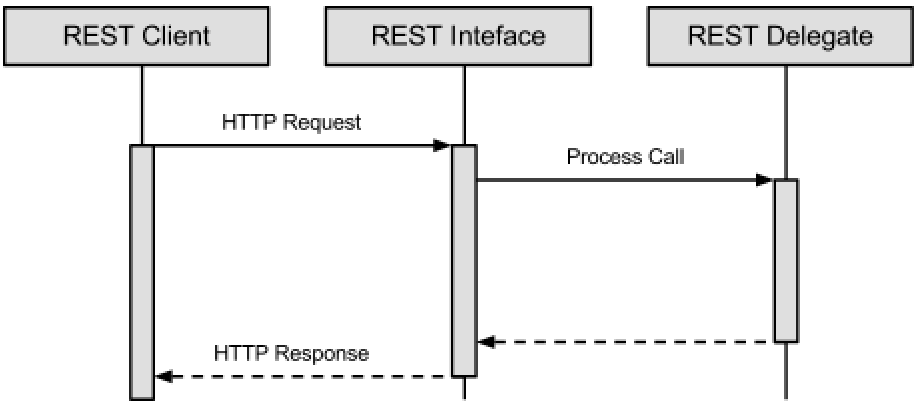

Figure 1: REST request-response sequence diagram

As demands around collaborative access and media richness evolved, this became a serious bottleneck, essentially preventing these types of applications. On the client-, i.e. browser-, side developers were able to work around the issue of synchronous transmission using the XmlHttpRequest object which allows to request resources programmatically (using JS) while deferring handling of the response to a callback (see figure 2[3]) thus enabling much more responsive software.

Although this addressed the issue on the client-side, server-side request were still handled very much in a synchronous fashion. For example the popular Apache web server forks a new process for each incoming request[4]. As popular applications have to cope with unprecedented amounts of concurrent users in conjunctions with massive request counts, this obviously causes performance issues.

1.1 Objectives

The main goal of this paper is to identify specific use cases for asynchronous server technologies. Focus is espacially laid on an application in a business environment. The authors' intention is to build a working prototype to present the advantages and disadvantages of this rising technologie.

[2]**req_res**.
[3]**img_ajax**.
[4]TODO: find source

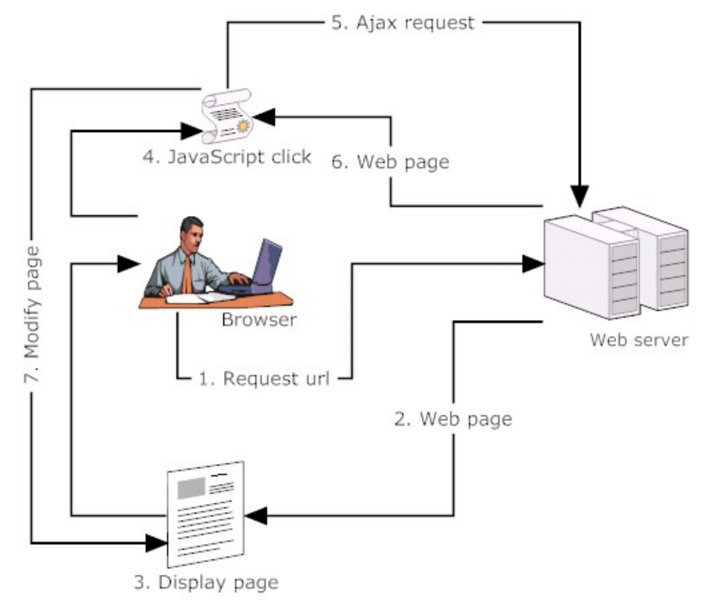

Figure 2: AJAX Diagram

1.2 Composition of this paper

The paper is devided in four parts. The first one explains the scientific fundamentals of the asynchronous server technology. Afterwards the asynchronous programming frameworks Node.js and verte.x are used as a practical implementation. In the following chapter the authors conceive a proof of concept, that guides through a step by step construction of a server- and client side based validation web application with non-blocking IO. The next part uses the POC to benchmark vertex and Node.js against common server technologies. The last chapter combines the findings into a final conclusion.

Relatively young frameworks such as Node.js and Vert.x try to address this issue by providing a completely asynchronous programming model which allows associating multiple simultanious connections with a single thread by using an event-driven approach.

This paper elaborates the concepts behind these young frameworks and analyses their technical strengths and weaknesses. Furthermore non-functional attributes will be evaluated based on two sample implementations in Node.js and Vert.x.

2 Setting the context

2.1 Comparison between Asynchronous and Synchronous Processing

A common case in programming is access to I/O. In synchronous processing a running thread needs to wait for the completion of the I/O operation before it can continue. The thread is in an idle state while it is waiting which allows another process or thread to occupy the CPU in the meanwhile.

In multithreaded applications several threads can run simultaniously within one process. Several threads can access shared memory concurrently which can cause inconcistent states. This can be avoided by synchronizing threads - e.g. with locks. This means that programmers need to take into account every possible execution order to effectively avoid program defects such as data races and deadlocks.[5] This can be time consuming and potentially results in error-prone code.

A typical synchronous call is provided in listing 1. The contents of a file are read and displayed afterwards. The programm is blocked until the read operation has finished.

```
reader = new FileReader();
content = reader.readAsText("input.txt");
printContent(content);
```

Listing 1: Pseudocode: Synchronously reading and displaying a file's contents

Asynchronous programming style uses a different concept. The flow of an application is determined by events, which is why this style is also called event-driven programming.[6] In listing 2 the call to the *readAsText* function is done asynchronously. There is no return value - instead an event handler is provided as a second argument. This function is also referred to as a callback function. It is called as soon as the read operation has completed.

```
read_completed = function(content) {
  printContent(content);
}

reader = new FileReader();
reader.readAsText("input.txt", read_completed);
```

Listing 2: Pseudocode: Asynchronously reading and displaying a file's contents

This concept is coupled with an event loop, which is a single thread that is running inside the main process. The loop constantly checks for new events. When an event is detected, the loop invokes the corresponding callback function. The callback is processed in the same thread which means that there is at most one callback running at a time. The event loop continues when the callback has completed. As a result the developer does not need to take care of concurrency issues during development. But the developer's task is to write light event handlers that can be

[5] **Breshears_2009**.
[6] **teixeira_2012**.

processed quickly as every callback is an interruption of the event processing in the event loop.[7] Memory or processor intense callbacks can lead to growing queues of unserved events which eventuelly results in a slow application or service[8].

2.2 Existing Asynchronous Frameworks

This sub-section will introduce two upcoming frameworks for server-side asynchronous development: Node.js and Vert.x followed by a brief outline of their potential market success.

2.2.1 Node.js

To put it in a nutshell one can say, that Node.js is JavaScript on a server.
Besides, Node.js is a young platform with a lot of buzz around it. Due to the rising of the Web 2.0 and widely accessible internet through smartphones, demanding users expect more complex and more interactive forms of application usage. The challenge even gets harder considering the steep number of devices that are interacting with online services.To overcome those problems Node.js lays it's foundations on an event driven computing architecture for web servers.
Node.js doesn't try to make you perform undoable things. It rather lets events drive the action, so that it is single-threaded and only one thing happens at once. This is why an event loop is a fundamental part of Node.js. It includes the concept of nonblocking I/O activities. A result is that actions that cause the program to wait like database requests and file I/O do not halt execution until they return data. In contrast they process independently and raise an event when the data is accessible. It is therefore necessary to use callbacks for dealing with different kinds of I/O.
An exemplary code for a basic HTTP server in Node.js is shown in listing 15 to deepen the understandin gof the event loop and callback in Node.js.

```
var http = require("http");

http.createServer(function(request, response) {
    response.writeHead(200, {"Content-Type": "text/plain"});
    response.write("Hello World");
    response.end();
}).listen(8888);
```

Listing 3: The simplest way to programm a server in Node.js

The code uses a factory method to create a new HTTP server and attaches the argument of the createServer function as a callback to the request event. The first run of this code is also called setup. When a HTTP request arrives the anonymous callback function is processed and "Hello World" appears on the browser.
That the basic code above isn't the most sophisticated way to write Node.js code explains the

[7]**Croucher_2010**.
[8]**teixeira_2012**.

following thought experiment: Assuming that the Hello World page would be popular and had a lot of requests from different devices to handle and in addition the callback processing would take one second, it is obvious that the second request would already have to wait for one second until it gets served. This is far away from the near-real-time requirement Node.js is confronted with.
Two programming rules in Node.js can be inferred from the basic server and it's event loop blocking problem, which is described in section 2. First, once the setup is in place all actions should be programmed event-driven. Second, if a workload requires Node.js to process something for a long time, it should be outsourced to web workers.[9]

Node.js large performance benefits are caused through the use of Google's new JavaScript engine V8, that ships naturally with the web browser Chrome. Due to the fact that V8 compiles JavaScript into real machine code before executing it, Node.js has an advantage over traditional techniques such as executing bytecode or interpreting it. [10]

2.2.2 Vert.x

Vertx is a polyglot that runs on the JVM (Java Virtual Machine). It is hence possible to scale over available cores without manually forking multiple servers.
The application API is exposed in multiple programming lanugages (see table 1).

In Vert.x the smallest available deployment unit is a verticle, which runs inside an instance. Each instance runs singlethreaded. Multiple verticles can be run inside one instance as depicted in figure 3.

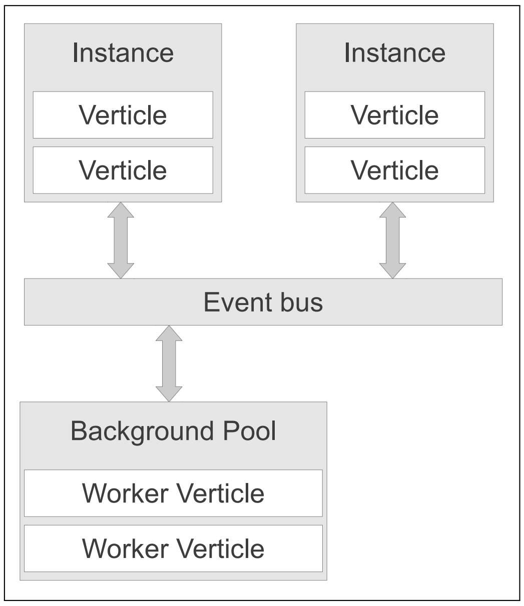

Figure 3: Abstracted deployment units of Vert.x

[9]**Croucher_2012**.
[10]See `https://developers.google.com/v8/intro`

When multiple instances are run on one machine, Vert.x automatically distributes incoming requests among all running instances in a round-robin fashion, so that each vert.x verticle instance remains single threaded.

Vert.x also includes a distributed event bus, which enables verticles to communicate with each other, either within the same instance or across different instances. The event bus allows direct communication with in-browser JavaScript as well.
Vert.x allows to run I/O heavy tasks in separate worker threads that reside in a so-called background pool as these tasks would otherwise block the event loop as described in section 2.1.

The core functionality of Vert.x covers basic networking tasks and protocols, web servers, clients, access to the file system, shared maps and the event bus. The core libraries of Vert.x can be embedded in any JVM program for reuse in larger projects.
As opposed to Node.js, the core functionality and API can be considered quite static as changes need to be done in all supported languages.[11]
The core can be exctended with additional features that are provided by optional modules that can be obtained over a public git-based module repository.[12]. The repository currently contains 16 distinct modules in different versions.[13]

An extensive online documentation is available for all supported languages. Additionally, code examples for most features are available for all supported languages in a public repository[14].
Vert.x is open source and licensed under the Apache Software License 2.0[15], so that commecial redistribution in closed source projects should not be an issue.

2.2.3 Market Overview

Table 1 shows that the previously introduced frameworks face stiff competition from established communities (namely the Ruby and Python ones). An important aspect in determining the potential success of a new technology is measuring the interest of software developers in the newcomer. Google-developed Google Trends visualizes the 'search interest', i.e. the amount of queries of a particular subject and related ones[16].

Looking at the charts in Fig. 4, it can be concluded that interest in Vert.x is low at the moment in absolute terms as well as in relative terms when comparing it to the Node at the same point in its lifecycle.

The gap in interest could be related to many reasons, two of which will be discussed briefly thus concluding this sub-section.

[11]**vertx_2012**.
[12]**vertx_mod_2012**.
[13]**Vertx_repository_2012**.
[14]**Fox_2013**.
[15]See `http://www.apache.org/licenses/LICENSE-2.0.html`
[16]**g_trends**.

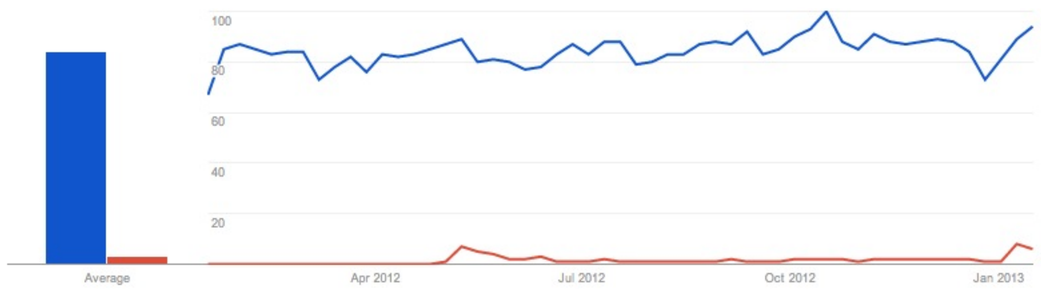

Figure 4: Google search volume of node.js and Vert.x compared[17]

Theory (1) would be that by the time Vert.x became complete enough to develop productively (at least as productive as was possible with Node at the same point), Node already established itself as the dominant technology especially if mindshare is concerned, which is supported by the data from Google. Theory (2) is based around technical considerations. Because of the dogmatic nature of asynchronicity in the Node.js framework, every package available on NPM is built with asynchronous calling and non-blocking I/O in mind. Vert.x on the other hand is in a curious position here, Vert.x itself is architecturally similar to Node but its ecosystem of established Java libraries is not developed with these principles in mind. This poses the danger of executing synchronous/blocking code, negating all of Vert.x' advantages. Obviously this can limit interest from the outset.

Another important metric to assess the potential impact of a framework is the technical background and the availability of skilled developers. Redmonk investigated in three consecutive years the popularity of programming languages by analyzing the usage on GitHub[18] and StackOverflow[19]. As of September 2012, JavaScript was ranked first before Java, PHP, Python and Ruby[20] – all of which have shown a strong performance over the past three years. This is an affirmative aspect with regard to the potential target audience of developers.

[18]See `http://github.com`
[19]See `http://stackoverflow.com`
[20]**Redmonk_2012**.
[21]**Twisted_2012**.
[22]**fettig_2005**.
[23]**eventmachine_2012**.
[24]**Mashtable_2011**.
[25]**vertx_2012**.

Name	**Language/s**	**Description**
Twisted	Python	"Twisted is an event-driven networking engine written in Python and licensed under the open source MIT license"[21]. Twisted is a mature framework with a large number of supported networking protocols. Its development is backed by an active community.[22]
EventMachine	Ruby	"EventMachine is a library for Ruby, C++, and Java programs. It provides event-driven I/O using the Reactor pattern."[23]
Node.js	JavaScript	Node is based on Chrome's JavaScript runtime *V8*. The platform is fully event-driven and offers core functionalities for network programming. Its functionality can be extended with a large number of modules using an integrated package management system. Node.js started development in 2009 and the community is growing since that.[24]
Vert.x	JavaScript, Java, Python, Groovy, Ruby, Coffeescript	A JVM based platform for network applications which is inspired by Node.js. Vert.x comes with its own event bus system that allows distributing applications among multiple network nodes. Support for the languages Scala and Clojure is scheduled for future releases.[25]

Table 1: Existing asynchronous programming frameworks

3 Areas of Application

3.1 Use Cases

The non-blocking nature of asynchronous calls is important in all types of applications that need to handle a large number of requests in real time.
Some success stories are available on `http://nodejs.org/`.

Could be used for:

- networked applications that tend to keep many inactive connections
- web trackers
- web servers
- lightweight json APIs (non-blocking I/O model combined with JavaScript make it a great choice for wrapping other data sources such as databases or web services and exposing them via a JSON interface)
- proxies
- email and messaging systems
- authorization processors
- Streaming data (e.g. file uploads in real time)

3.2 Don't use cases

As with all technologies one has to carefully evaluate whether or not it suits the requirements of a project. Besides those use cases listed in the previous section there are also a few usage scenarios where one should not use asynchronous frameworks like Node.js or Vert.x.
In general it can be said that these frameworks are not suited for tasks that require a lot of computation and IO access. These heavy tasks would have to be moved to worker threads. However having most of the computation logic in constructs that are not of the event loop contradicts the point of these frameworks. Running worker threads does partially introduce concurrency issues that were avoided with the event loop again as multiple threads could concurrently access IO.
Another thing to note is that one should not choose these frameworks when there are other concepts or frameworks that might better fulfill the projects requirements. A selection of such cases is shown in the list below.

Criticism on these concepts:
`http://xquerywebappdev.wordpress.com/2011/11/18/node-js-is-good-for-solving-problems-i-`
`http://www.theserverside.com/discussions/thread.tss?thread_id=61693`
`http://static.usenix.org/events/hotos03/tech/full_papers/vonbehren/vonbehren_html/`

`index.html`
and some more. Why not use message queues that make use of a thread pool?

Node.js and Vert.x do not offer any possibility to speed up a single request. These requests will aways run at the same speed. What it does is to maximise the number of possible requests while keeping the speed steady. This is why it scales so well. The only additional delay between request and response is the time that a request waits in the event loop until it gets processed. In many cases however it is desirable to minimize the computation time itself for a single request. Once this is achieved it becomes a goal to keep that speed for a larger amount of requests.

Datawarehousing with analytics Doing analytical and complex computations on large data sets at runtime usually requires a lot of computing power. It can be expected that each request will require quite some time to be processed. This computation time cannot be shortened much when run on a single thread but could potentially be speed up significantly when run on multiple cores in parallel.

CRUD / HTML applications This refers to classical websites that basically only serve as an interface for an object relational model. At present Node.js and Vert.x do not provide additional benefits to scalability for these types of web applications. Unless the page has to deal with a large number of concurrent requests it should be considered to use a more powerfull frameworks like Ruby On Rails, Grails or Django. These are currently better suited for quickly developing such an application. Providing a site that is suited for millions of requests does not automatically increase the number of users.

4 Exemplary Implementations

A simple web form application has been implemented in Node.js and Vert.x to further analyze non-functional requirements and collect practical experience with these frameworks. These implementations are thoroughly described in the following sub-sections.

4.1 Software Description

The exemplary implementation consists of a web service that can be used to calculate the expected fee for an insurance. However this application should only serve as a demonstration of the used asynchronous frameworks and is hence very simplified.

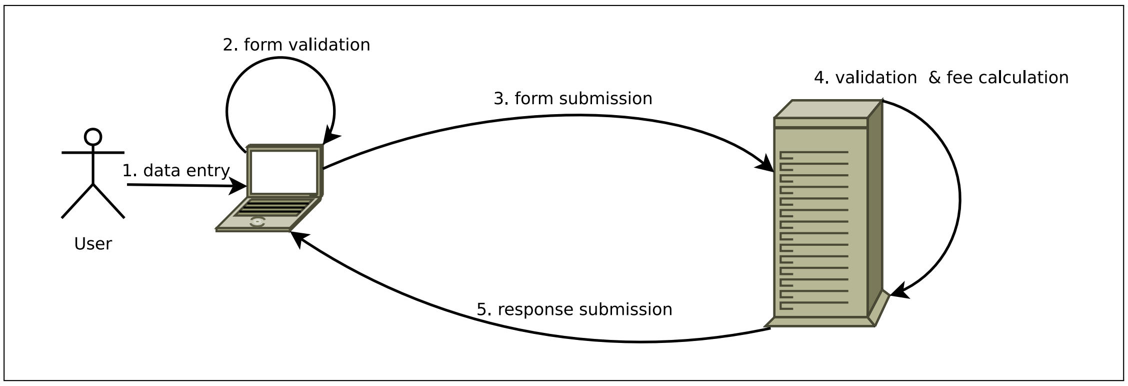

Figure 5: Usage workflow of the exemplary application

The basic usage workflow is shown in figure 5. In a first step the user opens the website in a browser and is shown a form that consists of two parts - the first part being a section for personal details and the second part for parameters that will be used to determine the fee for the insurance.
The user then fills in all necessary form fields and instantly gets feedback on the vailidy of the entered values. Once all fields are filled correctly the form can be submitted to the application via HTTP or HTTPS. The data received by the server then gets validated again to avoid processing of manipulated requests. Eventually the valid data is passed into the calculation routine which returns its result to the user by triggering the according callback.

4.2 Software Design

This simple usage scenario leads to a few requirements and design decisions. The user interface has to consist of HTML, CSS and JavaScript files as it will be displayed in a web browser. In general these files are not initially available on the clients machine and need to be delivered via HTTP/HTTPS by the server. These file requests are usually done via GET.

In addition to the static files, the server will also need to process requests that are sent via POST in order to receive the form data for the fee calculation.

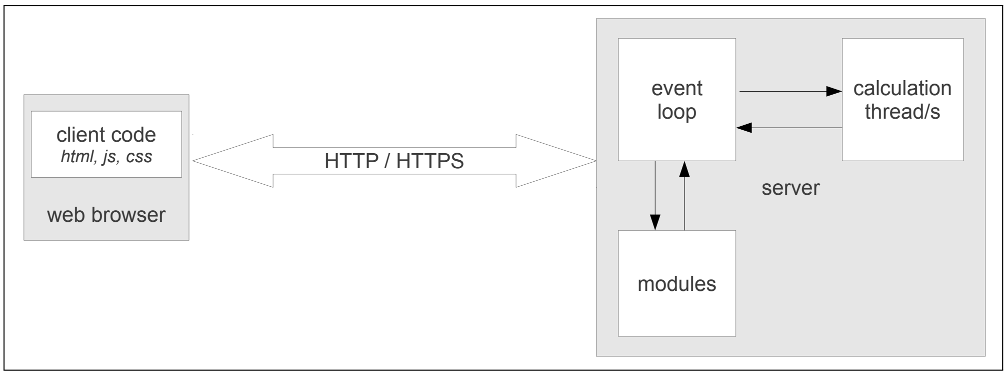

Figure 6: High level architecture of the application

The body size of these requests is fairly small as the submitted values consist of plain text only. Therefore it is safe to assume that it won't be necessary to buffer each part of the requests body but start calculating once the data submission has completed.
In reality these calculations are much complexer than in this simple scenario, so that they have to be done outside of the event loop in a separate thread. In the implementation these long running calculations are simulated artificially for demonstration purposes.
Additional features in the backend can be integrated via existing modules from the repositories of Vert.x or Node.js.

To create a secure application, HTTPS is required. To demonstrate an HTTP and HTTPS server likewise, the application includes two webservers, both of which serve the same functionality. It is necessary to generate a pair consisting of a private and a public key, because the handshake prior to an HTTPS session follows a specific procedure[26] : (1) Client contacts the server using a secured URL. (2) The server responds sending the digital certificate to the browser. (3) The client certifies that the certificate is valid (e.g. by trusted authorities). (4) Once the certificate is validated, the one-time session key generated by the client will be used to encrypt all communication with the server. (6) The client encrypts the session key with the public key of the server. Only the server can decipher the data using the private key.
The generation of private and public keys can be done be using the following commands, leveraging OpenSSL, an Open Source toolkit to implementing secure protocols[27] :

```
openssl genrsa -out privatekey.pem 1024
openssl req -new -key privatekey.pem -out certrequest.csr
openssl x509 -req -in certrequest.csr -signkey privatekey.pem -out certificate.pem
```

Listing 4: Generating a new pair of public/private keys

[26]**Nemati_2011**.
[27]See `http://www.openssl.org/`

4.3 Software Implementation

4.3.1 Vert.x

The vertx application is written in Java entirely, but it is also possible to write each verticle in a different language if desired. This could be done to maximize code reuse between frontend and backend.

The business logic of the backend consists of three verticles and additional utility classes. One verticle serves as a web server that accepts HTTP and HTTPS requests. GET requests are interpreted as file requests and are answered directly from within the verticle. POST requests are interpreted as form submissions. Whenever a POST requests arrives the verticle transforms the request body to a Json object and submits it via Vert.x' event bus.

The second verticle is a worker verticle which does the CPU intense calculation tasks.This verticle listens on the event bus for incoming messages. Whenever such a message arrives it transforms the received Json object into a form instance for validation and fee calculation.The results are then sent back via the event bus of this operation has completed, so that the event loop can return the result to the client in its next iteration.

The third verticle serves as a starter script which programmatically starts the previously mentioned verticles and terminates afterwards as it does not define any callbacks itself (see listing 4).

```
1  package main;
2  import org.vertx.java.core.SimpleHandler;
3  import org.vertx.java.core.json.JsonObject;
4  import org.vertx.java.deploy.Verticle;
5
6  public class AppStarter extends Verticle {
7    private static final String HTTP_CONF = "http_conf";
8    private static final int NUM_PROCESSORS = Runtime.getRuntime().
         availableProcessors();
9
10   @Override
11   public void start() throws Exception {
12     JsonObject httpConfig = appConfig.getObject(HTTP_CONF);
13     container.deployVerticle("main.Server", httpConfig);
14     container.deployWorkerVerticle("form.Calculator", NUM_PROCESSORS - 1);
15   }
16 }
```

Listing 5: Verticle that starts two other verticles

The actual deployment of the other verticles happen in lines 13 and 14. An important detail is the 2nd parameter in the `deployWorkerVerticle` method. Vert.x allows deploying multiple instances of a verticle programmatically which eases scaling an application to the number of available processors of a machine. In this case the main loop is single-threaded and for each remaining processor there is one worker verticle. Vert.x automatically distributes requests among these worker verticles (as described in section 2.2.2).

A global configuration file is read in line 12. These configuration file is formatted as Json string and can include settings for modules or verticles. The path to the configuration file has to be provided to the `vertx` command upon server start to allow access to it from within the verticles. For this implementation the command that has to be invoked is shown in listing 5.

```
vertx run main.AppStarter -conf app_conf.json -cp bin
```

Listing 6: Command for starting the Vert.x application

Vert.x allows either running verticles by providing the compiled Java class or by providing the source file which is then compiled automatically by Vert.x. However in this setup we experienced some issues with the Vert.x run command when we tried to use it with the source files. The issues observed where partially related to Vert.x bug 444[28]. Due to incomplete dependency resolution, not all classes were compiled so that Vert.x could not find all required classes in the path at runtime. These startup issues can easily be avoided by using compiled classes instead of the source files when multiple verticles need to be started programmatically. The path to the compiled classes has to be provided with the `cp` switch as shown in listing 5.

```
 1  private Handler<HttpServerRequest> formHandler = new Handler<HttpServerRequest
        >() {
 2    public void handle(final HttpServerRequest req) {
 3      req.bodyHandler(new Handler<Buffer>() {
 4
 5        @Override
 6        public void handle(Buffer buff) {
 7          JsonArray form = new JsonArray(buff.toString());
 8
 9          eBus.send("form.calculate", form,
10              new Handler<Message<JsonArray>>() {
11                public void handle(Message<JsonArray> message) {
12                  req.response.end(message.body.toString());
13                }
14              });
15        }
16      });
17    }
18  };
```

Listing 7: Command for starting the Vert.x application

The implementation of the communication between the web server and the worker verticle was rather easy by using the event bus. The event bus has a very simple adressing mechanism where the address consists of a normal string. Listing 6 shows the form handler which is defined in the web server verticle and gets called whenever a form submission arrives. The handler registers another handler that is called as soon as the entire request body has arrived (line 3). The data that arrives as a buffer is than turned into a JsonArray (line 7) and submitted to the event bus (line 9). As soon as a reply is returned over the event bus it is forwarded to the client (line 12).

[28]see `https://github.com/vert-x/vert.x/issues/444`

4.3.2 Node.js

The node.js implementation starts with four modules that are required to run the application:

```
var http = require('http');
var https = require('https');
var fs = require('fs');
var express = require('express');
```

Listing 8: HTTP and HTTPS in Node.js

The HTTP and HTTPS modules are necessary to set up the webservers accordingly. In addition, the "fs" (FileSystem[29]) module is necessary to read the encryption keys for HTTPS. The web framework "Express" offers complementary functionality for a webserver, like routing, and will therefore be used for convenience purposes, as recommended by Roden, G. (2012). Express can be installed using the following command-line command:

```
npm install express
```

Listing 9: Installing Express via command-line

In order to enable HTTPS by leveraging the node.js class https.Server, the privatekey.pem and certificate.pem files need to be read and their content can be written into the httpsOptions array by using the FileSystem module and its function readFileSync():

```
var httpsOptions = {
    key: fs.readFileSync(__dirname + '/privatekey.pem')
    , cert: fs.readFileSync(__dirname + '/certificate.pem')
}
```

Listing 10: FileSystem module

The files in this example are located in the same location as the application file. To run the HTTP/HTTPS servers, the following code is used:

```
http.createServer(app).listen(8888);
https.createServer(httpsOptions, app).listen(4431);
```

Listing 11: Running the server

The HTTP server can be accessed using http://localhost:8888/ as opposed to the HTTPS server that is available at https://localhost:4431/.

To be able to use Express, a variable called "app" will be initialized:

```
var app = express();
```

Listing 12: App variable

The first Express functionality used is that static files can be served with node.js. Using the following statement, the directory "/UI" is made the root directory for a http://localhost:[port]/

```
app.use(express.static(__dirname + '/UI'));
```

Listing 13: Serving static assets with Express

[29]See `http://nodejs.org/api/fs.html`

Another Express feature is the “bodyParser”, which is used as middleware to parse request bodies, for this Proof of Concept especially JSON.

```
app.configure(function(){
    app.use(express.bodyParser());
});
```

Listing 14: Using the bodyParser

Moreover, Express is used to catch requests to specific URLs. As the AJAX call coming from the user interface is set up as POST-request, the following code shows how to catch this (AJAX-driven) POST-request in node.js on http://localhost:8888/insurances:

```
app.post('/insurances', function (req, res) {

// iterating through the array that contains JSON-objects
// write the value of the name attribute of each JSON-object into the console
    for(var i = 0; i < req.body.length; i++) {
        console.log(req.body[i].name);
}

    res.contentType("text/plain");
res.send(200, OK)

}
```

Listing 15: Iteration through an array consisting of JSON data

As the AJAX POST-request contains an array filled with JSON-objects, this example demonstrates how to access each item. Within the proof of concept, this for-loop is used to apply regular expressions in order to validate each form field.

The node.js servers can be started by using the command-line. After navigating into the project’s folder using cd, the only command needed is:

```
node [filename].js
```

Listing 16: Executing Node.js code

5 Evaluation of Non-functional Attributes

5.1 Maintainability

Maintainability is one of the most important issues in the enterprise for developers typically switch between projects (or participate only in a temporary manner). Thus the time needed to become familiar with a framework and its language directly impact the development cost either because of the time needed related to the learning curve or because of resources invested in maintenance and bug-fixing. The results of this sub-section are summarized in table 2.

Node.JS Node.js allows for leveraging the JS skills already in the market which significantly lowers the Time to Market (TTM) for its similarity to established libraries and conventions in web application development. Unfortunately TTM is not everything because equal attention needs to be given to Total Cost of Ownership (TCO), i.e. the total expenditure required including costs incurred post-deployment. Here Node presents a challenge in its reliance on callback leading to hard-to-follow and hard-to-fix code driving up TCO significantly because patches become more and more difficult to write with increasing $\Delta t_{Launch<->Present}$, especially if the code is not to be maintained by the original developers.

Furthermore, Node is still at an unstable stage of code maturity (at time of writing, the current version is 0.9.6 causing infrequent breaking changes to its APIs which usually is a no-go in the enterprise, this is reinforced by a community mindset of 'Upgrade Node, Update Code' based on a preference for new features. The unintended benefit of this is a lot of available material for Node programmers in blogs and on StackOverflow[30] leading to the conclusion that the current pace of progress does not facilitate enterprise adoption, however it is not a showstopper.

Vert.x Vert.x offers a similar picture to Node, however some of Node's shortcoming are more pronounced on the Vert.x side of things. Whereas Node can recruit from established web developers, is Vert.x suffering from a curious conundrum. It is based on the JVM, which has enormous amounts of talent (esp. in the enterprise) but because of its diametrical differences to the conventional libraries it does not seem to be able to recruit interest from this pool. Also its support for multiple programming languages (basically every JVM language) should lower the barrier for entry even more, as for example Groovy already features Vert.x-aligned language constructs. Unfortunately this means that new Vert.x programmers need to be trained increasing TTM significantly. Another important issue is the unstable state of the Vert.x codebase as essential features like code import do not work reliably or at all[31]. Section 2.2.3 already mentioned the

[30] `www.stackoverflow.com`
[31] TODO: INCLUDE REF TO BUG

ease of calling synchronous code with Vert.x though this can be worked around with by placing the code in new verticles, this still requires a lot of vigilance.

TCO is difficult to estimate for Vert.x for a lack of success/failure stories, it can be said though that because Vert.x requires Java hosting infrastructure ther emight be an associated licensing cost for application servers.

	Node.js	**Vert.x**
Skill Availability	Good	None
Skills Transferable?	Yes (from est. JS frameworks)	No (very different from conventional J2EE
Available Material	A lot (books, blogs, documentation)	Project documentation and users group
License Cost	None	None
TTM	Low (transferable skills + material)	High (learning curve)
TCO	Medium (low TTM but may be high maintenance)	Medium (takes longer to develop, but JVM maintenance is well known)

Table 2: Maintainability comparison of Node and Vert.x

5.2 Integration

Vert.x and Node.js are supposed to be used as fully event driven standalone applications that can be extended with event-driven modules. However when introducing a Node.js or Vert.x application into the current application landscape it might be desired to reuse or communicate with existing systems that are not fully event driven. Furthermore, integration of these technologies might not be limited to mere IPC but instead as a new middleware-layer beneath current and new applications.

5.2.1 Node.js Integration

Node.js is not designed to handle CPU-intensive tasks efficiently. However, there is a way a Node process can perform such tasks without impairing the application performance. Node.js uses so-called child processes for this (provided by the *child process* module). The module is basically a wrapper around the unix tool *popen* and provides access to a child process' communication streams (stdin, stdout, stderr).[32]

There are two cases child processes are used for:
First, CPU-intensive tasks can be performed outside Node by assigning them to a different process (which is then called a child process) in order not to block the event loop. Output data from the child process is then sent back to the parent process.[33] In this case, the child process is used to outsource a task that requires high computation work and would otherwise block the event loop. This approach however requires routines that should run within the child process to be written in JavaScript as well, so that it is not usable to properly connect an existing application with the Node application.
A second way of using the child process module is to actually run external commands, scripts, files and other utilities that cannot directly be executed inside Node.[34]

This characteristic lets external processes get well integrated with Node.js. A basic example of the child process module is shown in listing 18[35]. The child process instance that is created in lines 1 and 2 is an event emitter that allows registering callbacks for certain events (see lines 4,8 and 12).

```
 1  var spawn = require('child_process').spawn,
 2      ls    = spawn('ls', ['-lh', '/usr']);
 3
 4  ls.stdout.on('data', function (data) {
 5    console.log('stdout: ' + data);
 6  });
 7
 8  ls.stderr.on('data', function (data) {
 9    console.log('stderr: ' + data);
10  });
11
12  ls.on('exit', function (code) {
13    console.log('child process exited with code ' + code);
14  });
```

Listing 17: Example of running *ls -lh /usr* in Node.js, capturing stdout, stderr, and the exit code

Invoking processes on a different machine across the network still requires a remote API or a distributed message bus system.

5.2.2 Vert.x Integration

In Vert.x there are multiple ways to communicate with external programs. Depending on the used language there are already multiple libraries that can be used to invoke child processes. However these libraries usually only expose synchronous APIs, so that these calls need to be

[32]**node_child_process**.
[33]**teixeira_2012**.
[34]**teixeira_2012**.
[35]**node_child_process**.

done inside a worker verticle that doesn't block the event loop.

Listing 19 shows a minimal worker verticle written in Java that invokes *ls -lh /usr* via the *Apache Commons Exec* library.[36]

```java
import org.vertx.java.deploy.Verticle;

public class CommandWorker extends Verticle {

    public void start() {
      //TODO: listen for new tasks on the message bus
    }

    private void runCommand(String command, String workingDirectory, int
        timeoutInMs) {
      Executor executor = new org.apache.commons.exec.DefaultExecutor();
      //TODO: add code here :)
    }
```

Listing 18: Example of running *ls -lh /usr* in Vert.x from within a worker verticle

When writing verticles in Java it is also possible to make use of Javas Remote Method Invocation (RMI) mechanism to communicate with services that are on different machines in the network. The most natural way for inter-application communication is by either using the integrated message bus or by using a separate bus system. Vert.x can be embedded in any JVM based program by including the according libraries in the path, so that these applications can use the bus system as well.[37]

5.2.3 Middleware

"The term middleware refers to the software layer between the operating system—including the basic communication protocols—and the distributed applications that interact via the network. This software infrastructure facilitates the interaction among distributed software modules."[38] The JVM and its associated libraries building the foundation for Vert.x are a classic example of this concept. This section is to point out briefly from a theoretical and practical perspective some of the aspects worth noting when dealing with Node.js and Vert.x as middleware.

The git runderlying concepts of both frameworks outlined thus far in this paper present advantages to middleware/library development as it does for business logic implementation. This realization manifests itself not only in the number of packages available on *npm* (21,104 as of January 18, 2013)[39] but also in new projects built on top of Node.js (see below).

[36]See `http://commons.apache.org/exec/`
[37]**vertx_2012**.
[38]**Geihs_2001**.
[39]**node_packages**.

One issue mentioned when dealing with middleware is the format of messages that are exchanged among distributed systems.[40] As Node.js brings a typical client-side script language to the server-side, the developer can take advantage of that fact by using the universal format JavaScript Object Notation (JSON) and common exchange format by using application/json as content type.[41]

Meteor and Opa Meteor and Opa are two frameworks built on top of Node.js and MongoDB created around a radically new development workflow. Both eliminate the distinction between client-side and server-side code, instead the developer writes code in a very similar way to conventional desktop programming.[42] The difference is that the framework provides functionality to that enables the code to alter its behavior depending on where it runs.[43] An example of this methodology can be found in appendix TODO.

Meteor leverages node.js that way that it makes use of its JavaScript-based concept and also enables the developer to take advantage the variety of packages available for node.js. Special attention should be paid to the fact that Meteor server code runs in one thread per request as opposed to the usual asynchronous callback methodology that applies to Node.js itself normally.[44]

Etherpad Etherpad is an online collaborative text editor. Etherpad nicely illustrates the power of Node as middleware. By migrating to Node reductions of lines of code and memory usage[45] allow for easier embedding in web applications and also greater extensibility by offering a JS API.

5.3 Scalability

Support for single machine scaling using multiple threads exists. Deployment on distributed systems differs in node.js and vert.x. Vert.x. uses its own communication bus to share information between verticles. Vert.x uses Hazelcast's[46] clustering functionality internally to scale on multiple machines.

6 Conclusion

[40]**Tannenbaum_2007**.
[41]**rfc4627**.
[42]**meteor_docs**.
[43]**meteor_docs**.
[44]**meteor_docs**.
[45]**Weissschuh_2013**.
[46]See `http://www.hazelcast.com/`

Appendices

List of Appendices

Appendix 1: METEOR OR OPA EXAMPLE

CODE CODE CODE

Lists of References