Evaluation of Asynchronous Server Technologies

Rocco Schulz, Max Vökler, Joe Boden,

Robert Wawrzyniak, Can Paul Bineytioglu

Corporate State University

Baden-Wuerttemberg - Stuttgart

January 13, 2013

This paper evaluates asynchronous server technologies. Strenghts and weaknesses of asynchronous programming models are elaborated and a proof of concept based on node.js and vert.x is used to evaluate non-functional attributes such as maintainability. . . .

1 Introduction

In traditional Web applications a connection's lifespan starts with a request and ends with the servers response. Each connection is assigned a thread which processes the request and terminates afterwards. However in order to allow real-time[1] communication between client and server it is necessary to extend the connection's lifespan to the duration of the whole session which results in many simultaniously open connections. Assigning one thread for each open connection would cause a large overhead for the server even though most threads would be inactive. This is due to the data associated with each thread and the process of context switching between all threads.
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.

[1]proper real time, as opposed to ancient definitions. TODO: reference

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.[2] 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.[3] 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 processed quickly as every callback is

[2] **Breshears_2009**.
[3] **teixeira_2012**.

an interruption of the event processing in the event loop.[4] Memory or processor intense callbacks can lead to growing queues of unserved events which eventuelly results in a slow application or service[5].

2.2 Existing Asynchronous Frameworks

2.2.1 Overview

Name	**Language/s**	**Description**
Twisted	Python	"Twisted is an event-driven networking engine written in Python and licensed under the open source MIT license"[6]. Twisted is a mature framework with a large number of supported networking protocols. Its development is backed by an active community.[7]
EventMachine	Ruby	"EventMachine is a library for Ruby, C++, and Java programs. It provides event-driven I/O using the Reactor pattern."[8]
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.[9]
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.[10]

Table 1: Existing asynchronous programming frameworks

[4]**Croucher_2010**.
[5]**teixeira_2012**.
[6]**Twisted_2012**.
[7]**fettig_2005**.
[8]**eventmachine_2012**.
[9]**Mashtable_2011**.
[10]**vertx_2012**.

2.2.2 Node.js

2.2.3 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 1.

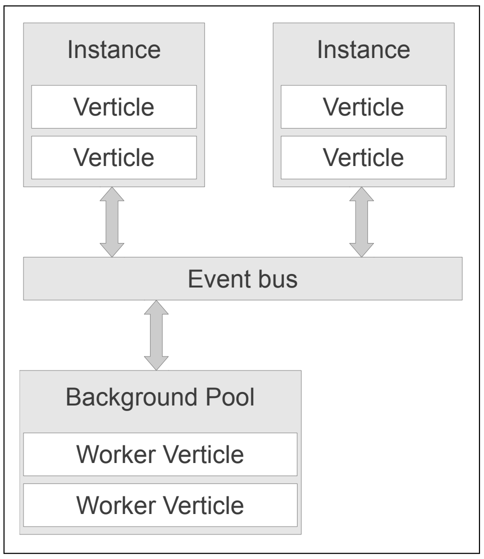

Figure 1: Abstracted deployment units of Vert.x

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 IO 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]

[11]**vertx_2012**.

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.

3 Areas of Application

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 analytics / trackers
- web servers
- proxies
- email and messaging systems
- authorization processors

Should not be used for:

- systems that do a lot of computing / IO access
- ...

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.

4.1 Software Description

Brief description of the insurance fee calculator

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

4.2 Software Design

High level design
Interface description, and differences between Node.js and Vert.x

4.3 Software Implementation

Complications or any other notes on the implementation process that might be of importance for the evaluation.

5 Evaluation of Non-functional Attributes

5.1 Maintainability

Language: JavaScript is wide spread, same for Java. However JavaScript offers better flexibility and native constructs for these types of applications. (Java 8 might improve things a bit). Node's API is undergoing backwards incompatible changes from time to time. It is desired to updated some components of a Node based application from time to time.

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 it might be desired to reuse existing software that is not fully event driven. Dont make use of blocking apis. Consider connecting that software with a message bus. Separate IO intense tasks into "web workers" or similar constructs.

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.

6 Conclusion

7 Lists of References