VERT.X HTTP SERVER WHITE PAPER

Name Hariprasad

Date September, 2019

Contents

1. Introduction

Vert.x is an open source, reactive, polyglot toolkit or platform running on the Java Virtual Machine. We can think of Vert.x as an alternative to the Java Enterprise Edition but with a different approach to solving the same problem - developing networked, highly concurrent applications. Vert.x provides one of the feature i.e Vert.x Http Server using these server we can develop highly concurrent application. We can see in these white paper few basics of vert.x , how to create an HTTP server and how to handle requests.

2. Vert.x Verticles

Vert.x can deploy and execute components called Verticles. We can think of verticles as being similar to servlets or message driven EJBs in the Java Enterprise Edition model. Here we can see Vert.x platform with 4 verticles running:

2.1 Verticle Types

There are three different types of verticles:

2.1.1 Standard Verticles

Standard verticles are assigned an event loop thread when they are created and the start method is called with that event loop. When you call any other methods that takes a handler on a core API from an event loop then Vert.x will guarantee that those handlers, when called, will be executed on the same event loop.

This means we can guarantee that all the code in your verticle instance is always executed on the same event loop (as long as you don’t create your own threads and call it!).

This means we can write all the code in your application as single threaded and let Vert.x worry about the threading and scaling. No more worrying about synchronized and volatile any more, and we also avoid many other cases of race conditions and deadlock so prevalent when doing hand-rolled 'traditional' multi-threaded application development.

2.1.2 Worker Verticles

A worker verticle is just like a standard verticle but it’s executed using a thread from the Vert.x worker thread pool, rather than using an event loop.

Worker verticles are designed for calling blocking code, as they won’t block any event loops.

If we don’t want to use a worker verticle to run blocking code, we can also run inline blocking code directly while on an event loop.

Worker verticle instances are never executed concurrently by Vert.x by more than one thread, but can executed by different threads at different times.

2.1.3 Multi-threaded worker verticles

A multi-threaded worker verticle is just like a normal worker verticle but it can executed concurrently by different threads.

Because of the concurrency in these verticles you have to be very careful to keep the verticle in a consistent state using standard Java techniques for multi-threaded programming .

Multi-threaded worker verticles were designed and are intended for the sole use of consuming simultaneously EventBus messages in a blocking fashion.

Multi-threaded worker verticles are an advanced feature and most applications will have no need for them.

Vert.x server (HTTP) cannot be created in a multi-threaded worker verticle. Should we incidentally try, an exception will be thrown.

2.2 Implementing a Verticle

We can implement a verticle by creating a class that extends `io.vertx.core.AbstractVerticle`. Here we can see verticle class:

```
import io.vertx.core.AbstractVerticle;

public class BasicVerticle extends AbstractVerticle {

}
```

The Class BasicVerticle extends AbstractVerticle but it doesn't have any functionality.

2.2.1 start()

The AbstractVerticle class contains a start() method which we can override in our verticle class. The start() method is called by Vert.x when the verticle is deployed and ready to start. Here we can see how we can implementing the start() method.

```
public class BasicVerticle extends AbstractVerticle {
    public void start() throws Exception {
        System.out.println("BasicVerticle started");
    }
}
```

The start ( ) method is where we initialize our verticle. Inside the start ( ) method we will normally create e.g. HTTP server, register event handlers on the event bus, deploy other verticles, or whatever else our verticle needs to do its work.

2.2.2 stop()

The AbstractVerticle class also contains a stop() method we can override. The stop() method is called when Vert.x shuts down and our verticle needs to stop. Here we can see overriding the stop() method in our own verticle.

```
public class BasicVerticle extends AbstractVerticle {

    public void start() throws Exception {
        System.out.println("BasicVerticle started");
    }

    public void stop() throws Exception {
        System.out.println("BasicVerticle stopped");
    }
}
```

2.3 Deploying a Verticle

Once we have created a Verticle we need to deploy it inside Vert.x in order to execute it. Here is how we can deploy a verticle.

```
public class VertxVerticleMain {

    public static void main(String[] args) throws InterruptedException {
        Vertx vertx = Vertx.vertx();

        vertx.deployVerticle(new BasicVerticle());
    }
}
```

First a Vertx instance is created. Second, the deployVerticle() method is called on the Vertx instance, with an instance of our verticle (BasicVerticle is our verticle) as parameter.

Vert.x will now deploy the verticle internally. Once Vert.x deploys the verticle, the verticle's start() method is called.

The verticle will be deployed asynchronously, so the verticle may not be deployed by the time the deployVerticle() method returns.

2.4 Deploying a Verticle From Another Verticle

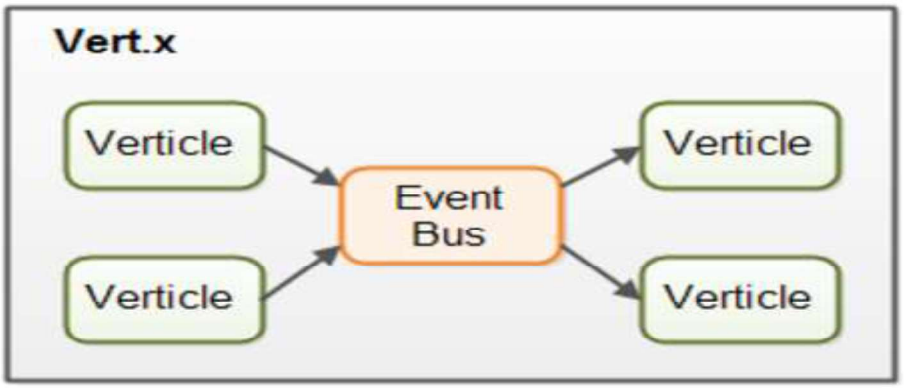
It is possible to deploy one verticle from inside another verticle. Here we can see

```
public class BasicVerticle extends AbstractVerticle {
    public void start() throws Exception {
        System.out.println("BasicVerticle started");

        vertx.deployVerticle(new SecondVerticle());
    }
    public void stop() throws Exception {
        System.out.println("BasicVerticle stopped");
    }
}
```

3. Vert.x Event Bus

Verticles are event driven, meaning they do not run unless they receive a message. Until then they remain dormant. Verticles can communicate with each other via the Vert.x event bus. Here we can see how the verticles communicate via the Vert.x event bus:



Messages can be simple objects (e.g. Java objects), strings, CSV, JSON, binary data or whatever else you need.

Verticles can send and listen to *addresses*. When a message is sent to a given address, all verticles that listen on that address receive the message. Verticles can subscribe and unsubscribe to addresses without the senders knowing. This results in a very loose coupling between message senders and message receivers.

All message handling is asynchronous. If a verticle sends a message to another verticle, that message is first put on the event bus, and control returned to the sending verticle.

Later, the message is dequeued and given to the verticles listening on the address the message was sent to.

3.1 Using The Event Bus

It is very common for verticles to either listen for incoming messages from the event bus, or to write messages to other verticles via the event bus.

3.1.1 Listening for Messages

When a verticle wants to listen for messages from the event bus, it listens on a certain address. An address is just a name (a String) which we can choose freely.

Multiple verticles can listen for messages on the same address. This means that an address is not unique to a single verticle. An address is thus more like the name of a channel we can communicate via. Multiple verticles can listen for messages on an address, and multiple verticles can send messages to an address.

A verticle can obtain a reference to the event bus via the vertx instance inherited from AbstractVerticle

Here is how listening for messages on a given address:

```
public class EventBusReceiverVerticle extends AbstractVerticle {

    public void start() {

        vertx.eventBus().consumer("anAddress", message -> {
            System.out.println("One received message.body() = "+ message.body());
        });
    }
}
```

A verticle that registers a *consumer* (listener) of messages on the Vert.x event bus. The consumer is registered with the address anAddress, meaning it consumes messages sent to this address via the event bus.

The consumer is a handler object which contains a single method. That is why it is implemented above using a lambda expression.

3.1.2 Sending Messages

Sending messages via the event bus can be done via either the send() or publish() method on the event bus.

The publish method sends the message to all verticles listening on a given address.

The send() method sends the message to just one of the listening verticles. Which verticle receives the message is decided by Vert.x.

Here we can see that deploys two event bus consumers (listeners), and one event bus sender. The sender sends two messages to a given address. The first message is sent via the publish() method, so both consumers receive the message. The second message is sent via the send() method, so only one of the consumers will receive the message.

```
Vertx vertx = Vertx.vertx();

vertx.deployVerticle(new EventBusReceiverVerticle("R1"));
vertx.deployVerticle(new EventBusReceiverVerticle("R2"));

Thread.sleep(3000);
vertx.deployVerticle(new EventBusSenderVerticle());
```

```
public class EventBusSenderVerticle extends AbstractVerticle {

    public void start() {
        vertx.eventBus().publish("anAddress", "message 2");
        vertx.eventBus().send   ("anAddress", "message 1");
    }
}
```

```
public class EventBusReceiverVerticle extends AbstractVerticle {
    private String name = null;
    public EventBusReceiverVerticle(String name) {
        this.name = name;
    }
    public void start() {
        vertx.eventBus().consumer("anAddress", message -> {
            System.out.println(this.name +  " received message: " + message.body());
        });
    }
}
```

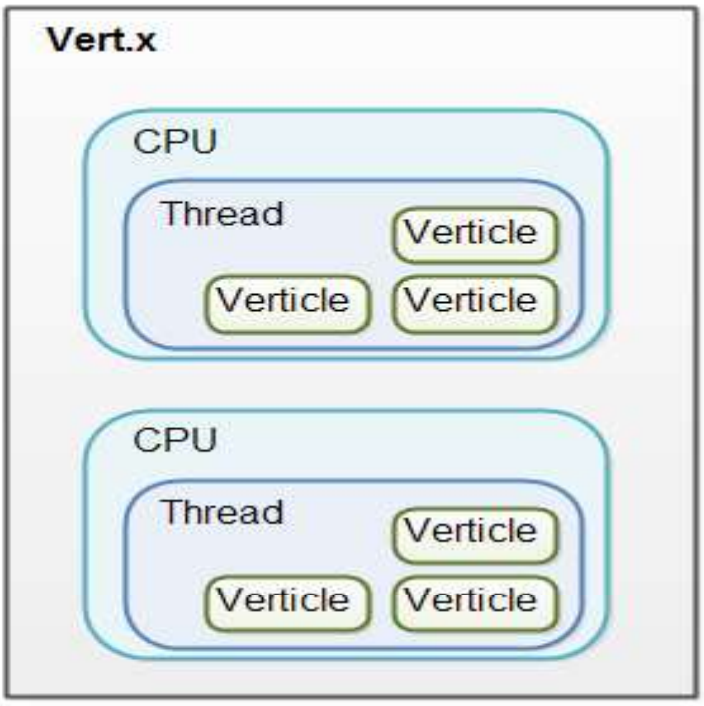
If you run this code you will see that the first message is received by both consumers (R1 + R2) whereas the second message is only received by one of the consumers.

4. The Vert.x Thread Model

Verticles run in single thread mode. That means, that a verticle is only ever executed by a single thread, and always by the same thread. That means that you will never have to think about multithreading inside your verticle (unless we start other threads which our verticle communicates with etc).

Vert.x is capable of using all the CPUs in your machine, or cores in your CPU. Vert.x does this by creating one thread per CPU. Each thread can send messages to multiple verticles. Remember, verticles are event driven and are only running when receiving messages, so a verticle does not need to have its own exclusive thread. A single thread can distribute messages to multiple verticles.

When a thread delivers a message to a verticle, the message handling code of that verticle is executed by the thread. The message delivery and message handling logic is executed by calling a method in a handler (listener object) registered by the verticle. Once the verticle's message handling logic finishes, the thread can deliver a message to another verticle.



4.1 Vert.x Installation

Vert.x is distributed in a zip file containing a bunch of JAR files. We can just unzip the zip file and add these JAR files to the classpath of our Java application, and you are good to go. Pretty simple!

Vert.x is not like a clunky Java application server which needs to be installed, have environment variables set up etc. We can just use Vert.x like any other API, from

inside our Java code. As long as the JAR files are available on the classpath of our application.

6. Vert.x HTTP Server

Vert.x makes it easy to create an HTTP server so our application can receive HTTP requests. We can create one or more HTTP servers, depending on our need. Here we can see how to create an HTTP server from inside a verticle and how to handle requests.

6.1 Creating an HTTP Server

Creating an HTTP server is done using the Vertx instance method createHttpServer(). Here we can how we can creating an HTTP server in Vert.x:

```
HttpServer httpServer = vertx.createHttpServer();
```

It is common to start an HTTP server from within an verticle. That way all handlers registered on the HTTP server will be executed by the same thread that started the verticle.

```
import io.vertx.core.AbstractVerticle;
import io.vertx.core.http.HttpServer;

public class VertxHttpServerVerticle extends AbstractVerticle {

private HttpServer httpServer = null;

  public void start() throws Exception {
     httpServer = vertx.createHttpServer()
   }

}
```

6.2 Starting The HTTP Server

Once we have created the HTTP server, we can start it using its listen() method. Here is how starting the HTTP server looks:

```
public class VertxHttpServerVerticle extends AbstractVerticle {

    private HttpServer httpServer = null;

    public void start() throws Exception {

        httpServer = vertx.createHttpServer();
        httpServer.listen(9999);

    }
}
```

The HttpServer class has more versions of the listen() method too, which gives different options for starting the HTTP server.

6.3 Setting a Request Handler on the HTTP Server

In order to handle incoming HTTP requests we must set a request handler on the HTTP server. This is normally done before starting the server. Here is a Vert.x HTTP server request handler

```
public class VertxHttpServerVerticle extends AbstractVerticle {

    private HttpServer httpServer = null;

    public void start() throws Exception {
        httpServer = vertx.createHttpServer();

        httpServer.requestHandler(req -> {
            if(req instanceof HttpServerRequest){
                System.out.println("incoming request!");
            }
        });

        httpServer.listen(9999);
    }
}
```

Every time an HTTP request arrives at the HTTP server, the requestHandler() method of the HttpSever is called. We can execute the code needed Inside the HttpServerRequest if block to handle the HTTP request.

6.3.1 Request Headers and Parameters

We can access HTTP headers and parameters from the HttpServerRequest object passed as parameter to the handle() method. Here is showing how to access a few of the properties of a HTTP request:

```
httpServer.requestHandler(request-> {
    public void handle(HttpServerRequest request) {
        System.out.println("incoming request!");
        request.uri();
        request.path();
        request.getParam("id");
    }
});
```

6.3.2 Handling POST Requests

If the HTTP request is a HTTP POST request we need to handle it a bit differently. we need to attach a body handler to the HTTP request. The body handler is called whenever data from the request body arrives. Here we can see that

```
httpServer.requestHandler(request -> {
   public void handle(HttpServerRequest request) {
      System.out.println("incoming request !!!");
      if(request.method() == HttpMethod.POST){
         request.handler(data -> {
            System.out.println("incoming request is post request !!!");
         }
      }
   }
});
```

If we want to wait until the full HTTP POST body has arrived you can attach an end handler instead. The end handler is not called until the full HTTP POST body has been received. However, the end handler does not have direct access to the full HTTP POST body. We need to collect that in the request handler.

Here is a Vert.x HTTP request end handler which does all that:

```
httpServer.requestHandler(request -> {
    public void handle(HttpServerRequest request) {

        System.out.println("incoming request !!!");

        if(request.method() == HttpMethod.POST){

            request.endHandler(data ->  {
               System.out.println("Post request with full request body !!!");
            });

        }
    }
});
```

6.4 Sending Back an HTTP Response

We can send back an HTTP response for an incoming HTTP request. To do so we need to obtain the HttpServerResponse instance from the request object. This is how we obtain the HttpServerResponse object:

```
HttpServerResponse response = request.response();
```

Once we obtained a HttpServerResponse instance we can set the HTTP response status code and headers like this:

```
response.setStatusCode(200);
response.headers()
    .add("Content-Length", String.valueOf(57))
    .add("Content-Type", "text/html");
```

After writing the headers back we can write the response body back via the write() method.

```
response.write("Welcome to thbs !!!");
response.end();
```

We call the write() method also exists in a version that takes a Vert.x Buffer instance as parameter. This method will write the contents of the Buffer to the HTTP response.

The write() method is asynchronous and returns immediately after queuing up the string or buffer.

Once you have finished writing the HTTP response body you should end the HTTP response. This is done by calling the end() method.We can also write the HTTP response body and end the response in a single method call, like this:

```
response.end("Welcome to THBS !!!");
```

The end() method can take either a String or Buffer as parameter. The parameter will be written to the response body, and the response ended after that.

6.5 Closing the HTTP Server

To close an HTTP server we simply call its close() method like this:

```
httpServer.close();
```

The close() method executes asynchronously, so the HTTP server may not be fully closed by the time the close() method returns. We can pass a close handler as parameter to the close() method, to be notified when the HTTP server is fully closed.

7. Conclusion:

Vert.x offers a wide range of features for many types of complex scenarios. It is providing single threaded Http Server for handling concurrent http request and responses. For more advanced vertx features , I encourage you to check out the Vert.x documentation("https://vertx.io/docs/").