*Massimiliano Dessì*

# VERT.X

*@desmax74*

GDG Sardegna

Java User Group Sardegna onlus

# Speaker

@desmax74

Massimiliano Dessì has more than 13 years of experience in programming.
He's a proud father of three, Manager of Google Developer Group Sardegna, Founder of SpringFramework IT, co-founder of JugSardegna.
Author of Spring 2.5 AOP.
He works and lives in Cagliari, Italy.

# Vert.x

Vert.x is a

lightweight (IoT)

polyglot

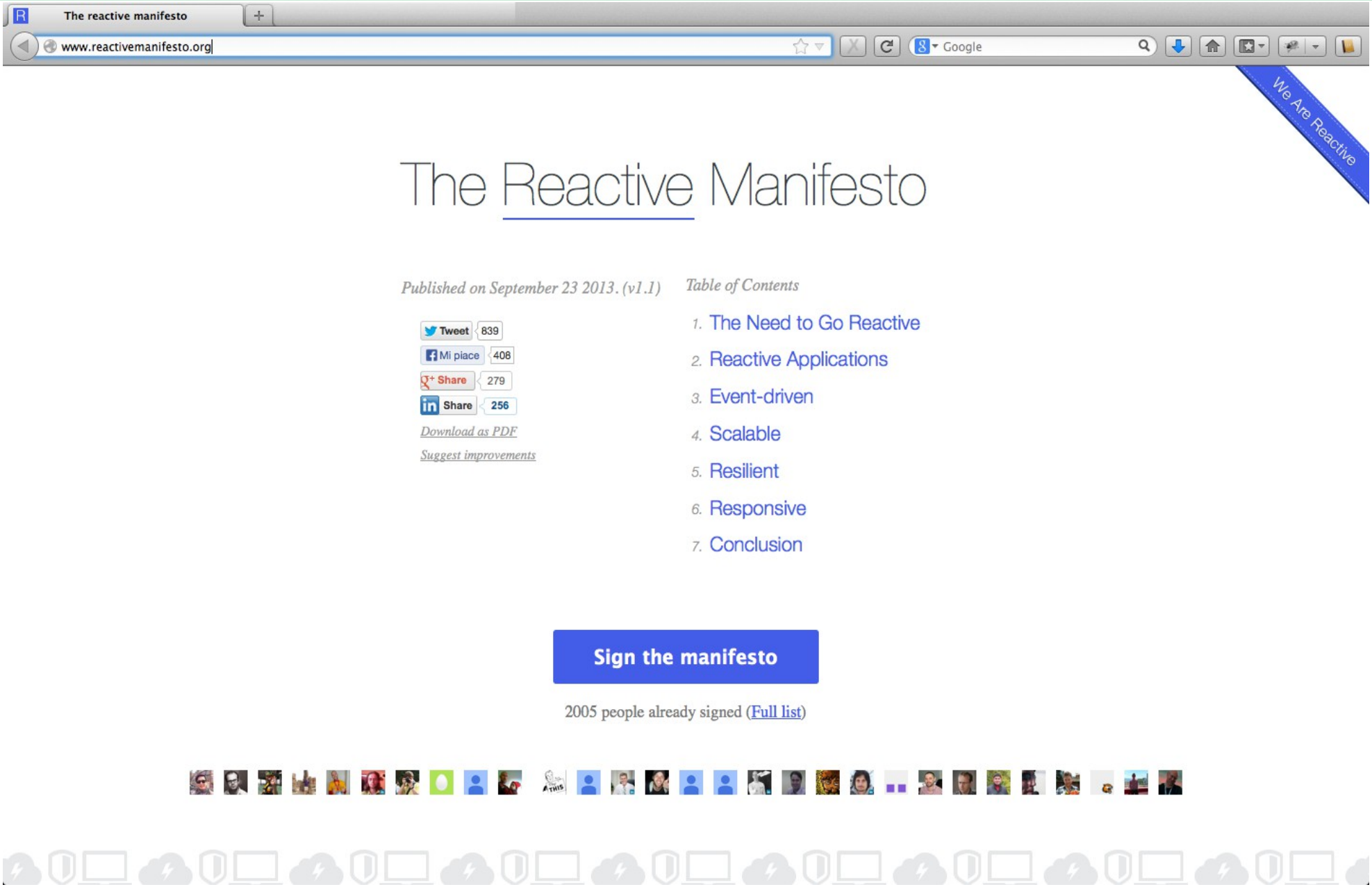
application development framework

for the JVM

enabling you

to build

high performance/reactive applications


The reactive manifesto
www.reactivemanifesto.org
Google
We Are Reactive
The Reactive Manifesto
Published on September 23 2013. (v1.1)
Tweet 839
Mi piace 408
Share 279
Share 256
Download as PDF
Suggest improvements
Table of Contents
1. The Need to Go Reactive
2. Reactive Applications
3. Event-driven
4. Scalable
5. Resilient
6. Responsive
7. Conclusion
Sign the manifesto
2005 people already signed (Full list)

# Software requirements nowadays

Highly Responsive

Real Time

Scalable

Resilient

Petabytes

# New Problems

We need different weapons (architectures)


http://iamdany.com/Famous-Weapons

# Reactive

"readily responsive to a stimulus"

Component active and ready to respond to event

Event Driven

# Reactive

React to events → Event Driven

React to failure → Resilient

React through a UI → Interactive

React to load → Scalable

# React to event - Event driven

Asyncronous and loosely coupled
+
Non blocking
=

lower latency and higher

throughput

# Wear your Seatbelt

# Higher throughput



http://vertxproject.wordpress.com/2012/05/09/vert-x-vs-node-js-simple-http-benchmarks/

http://www.techempower.com/benchmarks/

http://www.techempower.com/blog/2013/04/05/frameworks-round-2/

# Old blocking model

# Blocking apps

The “traditional” applications/containers

reserve

one thread

for each I/O resource,

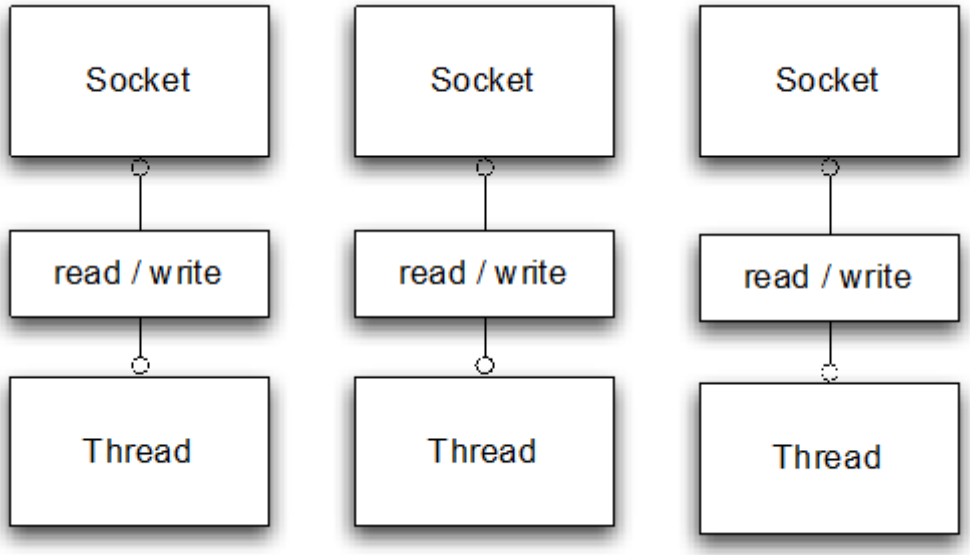
this mean

one thread per connection,

this is a blocking architecture

because rest of incoming connections must await

# Old blocking model



# New Challenge

http://en.wikipedia.org/wiki/C10k_problem

"The C10k problem is the problem of optimising network sockets to handle a large number of clients at the same time"

# C10k solutions on jvm

- No shared mutable state (all solutions derived from this) -

Functional approach [Scala, JDK8]

Actors [Akka]

Reactor/EventLoop [Vertx]

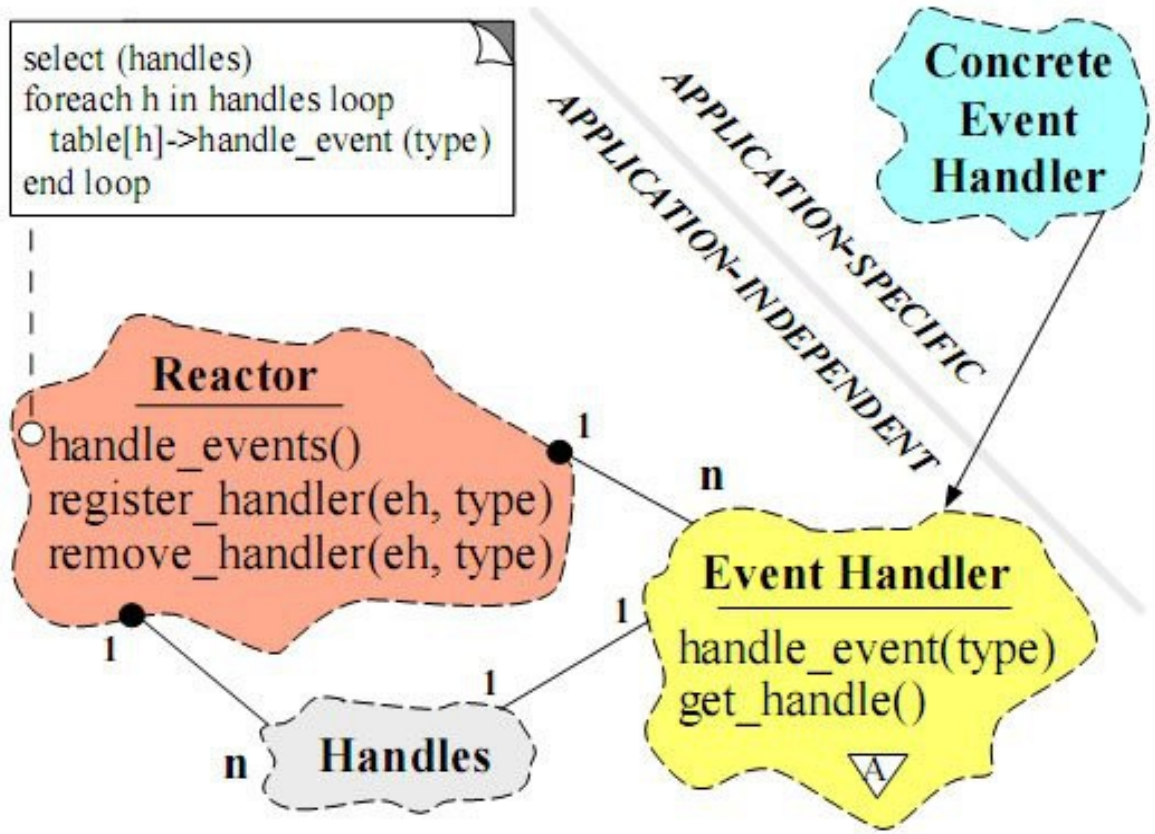
Project Reactor

Jetty

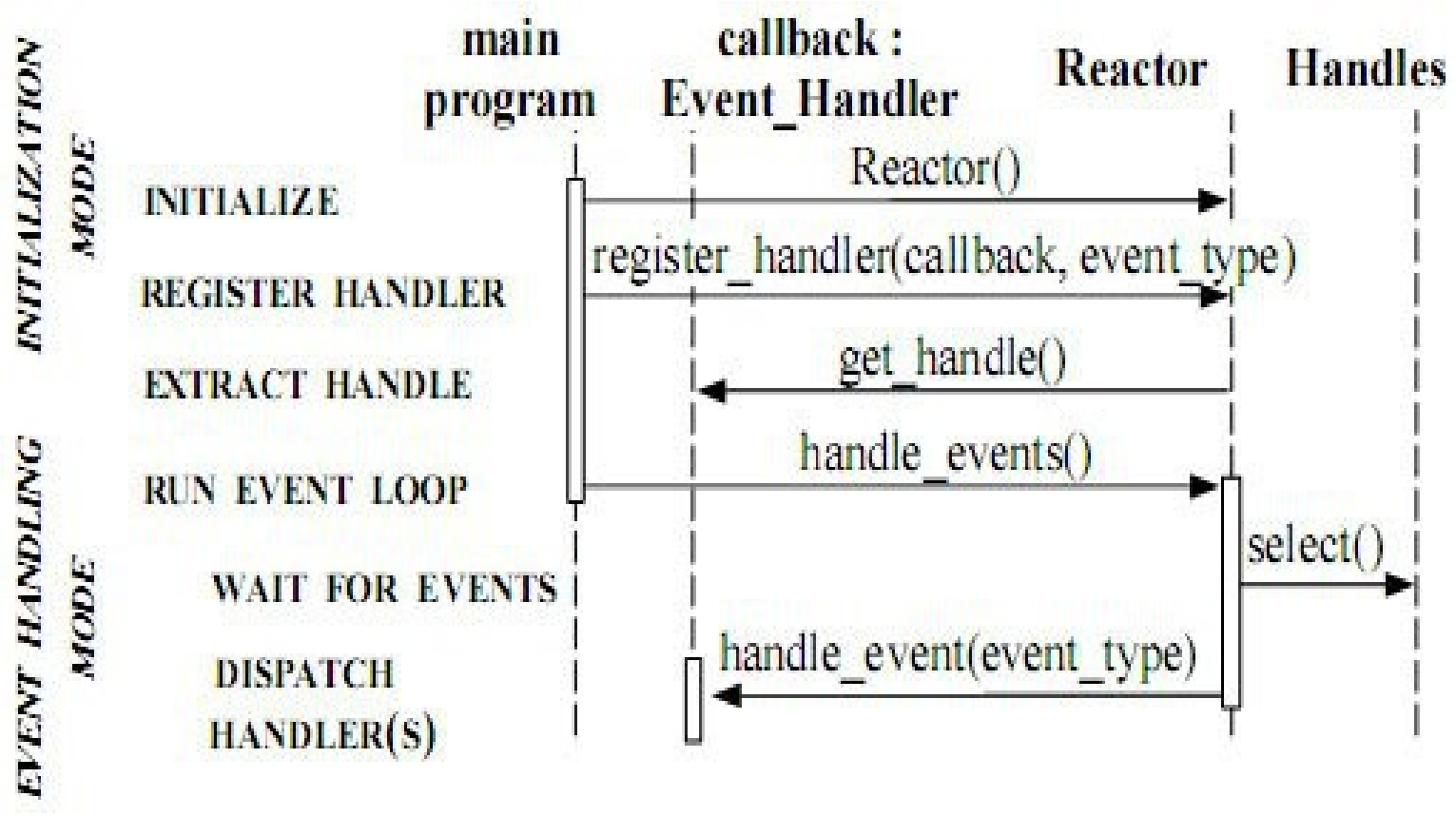
Disruptor

...

# Reactor pattern



# Reactor pattern



# Event Loop

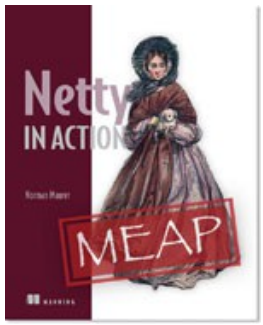
The Reactor pattern

implementation in Vertx

is based on

Netty

EventLoop

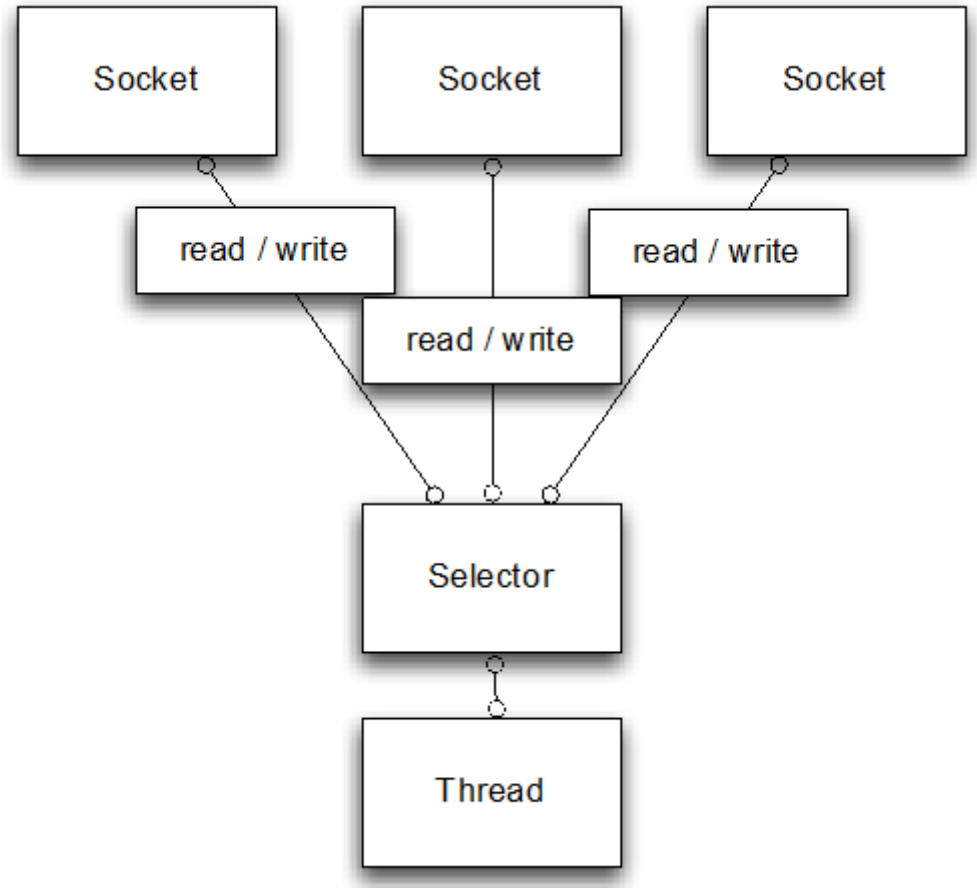


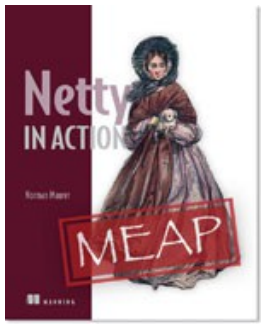
# Event Loop



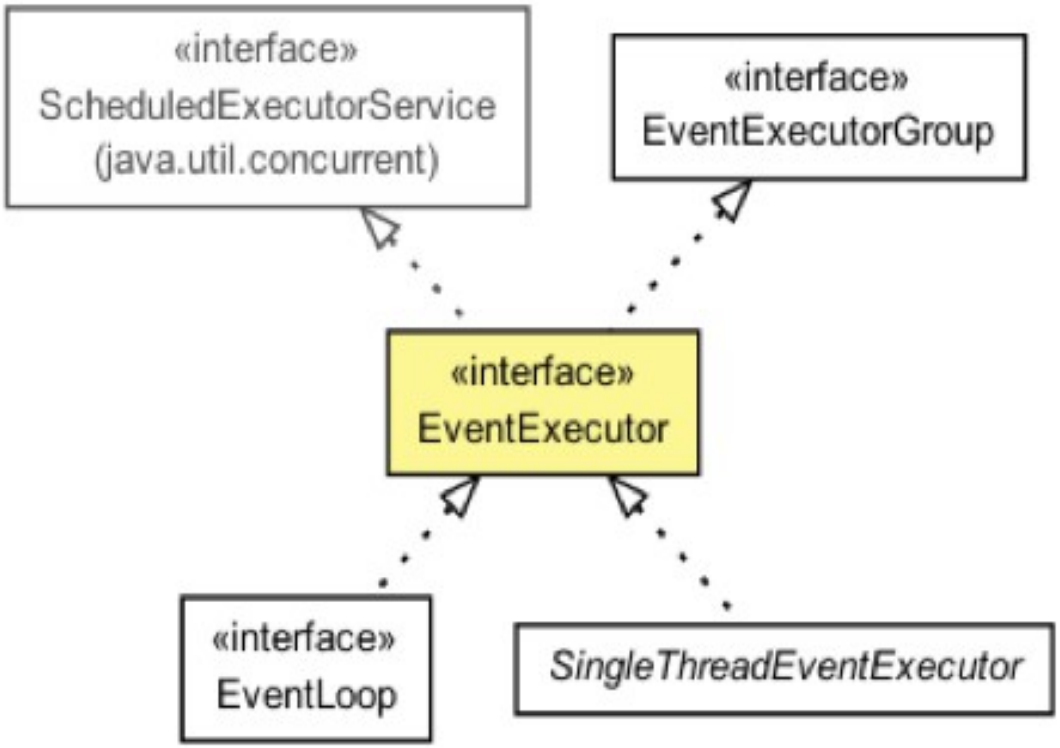
http://500px.com/photo/40357406

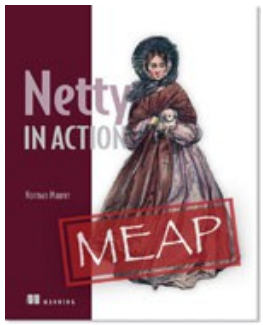
# Non blocking – Netty approach



# Non blocking – Netty approach



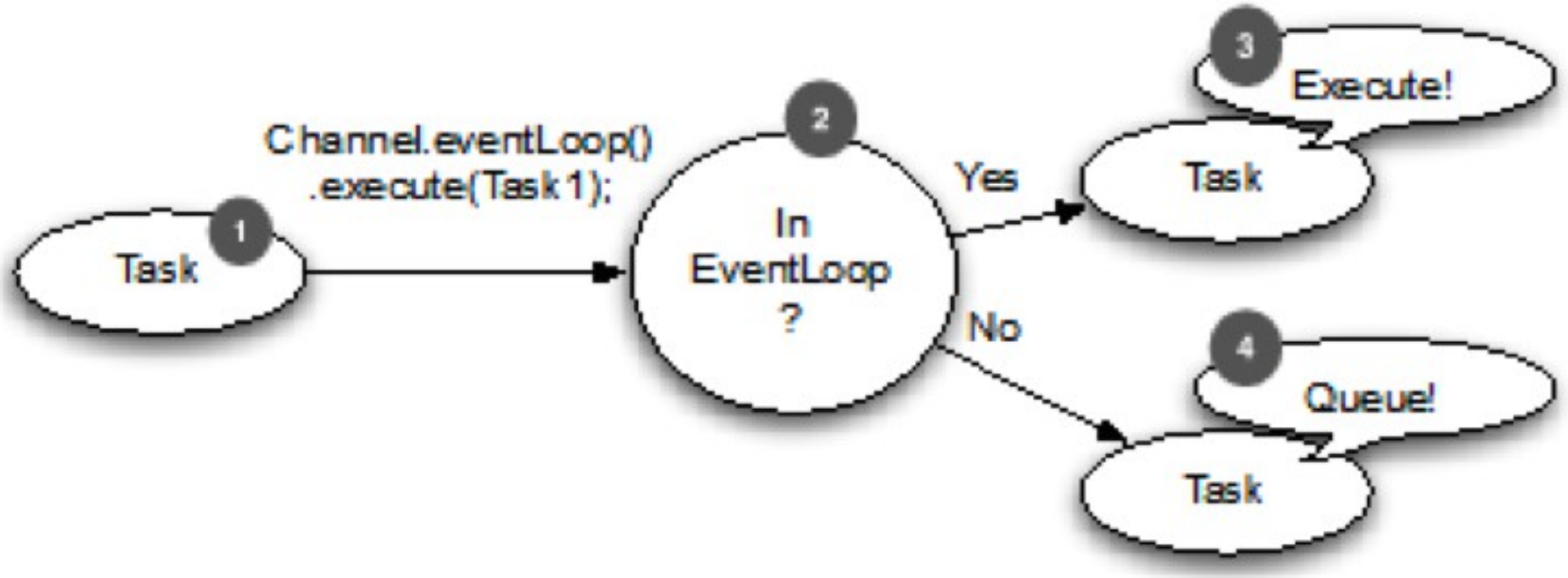



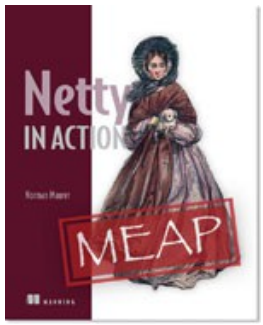

An eventLoop is powered by exactly one Thread

that never change.

The Events and task are executed in a FIFO order

# Netty thread model internals

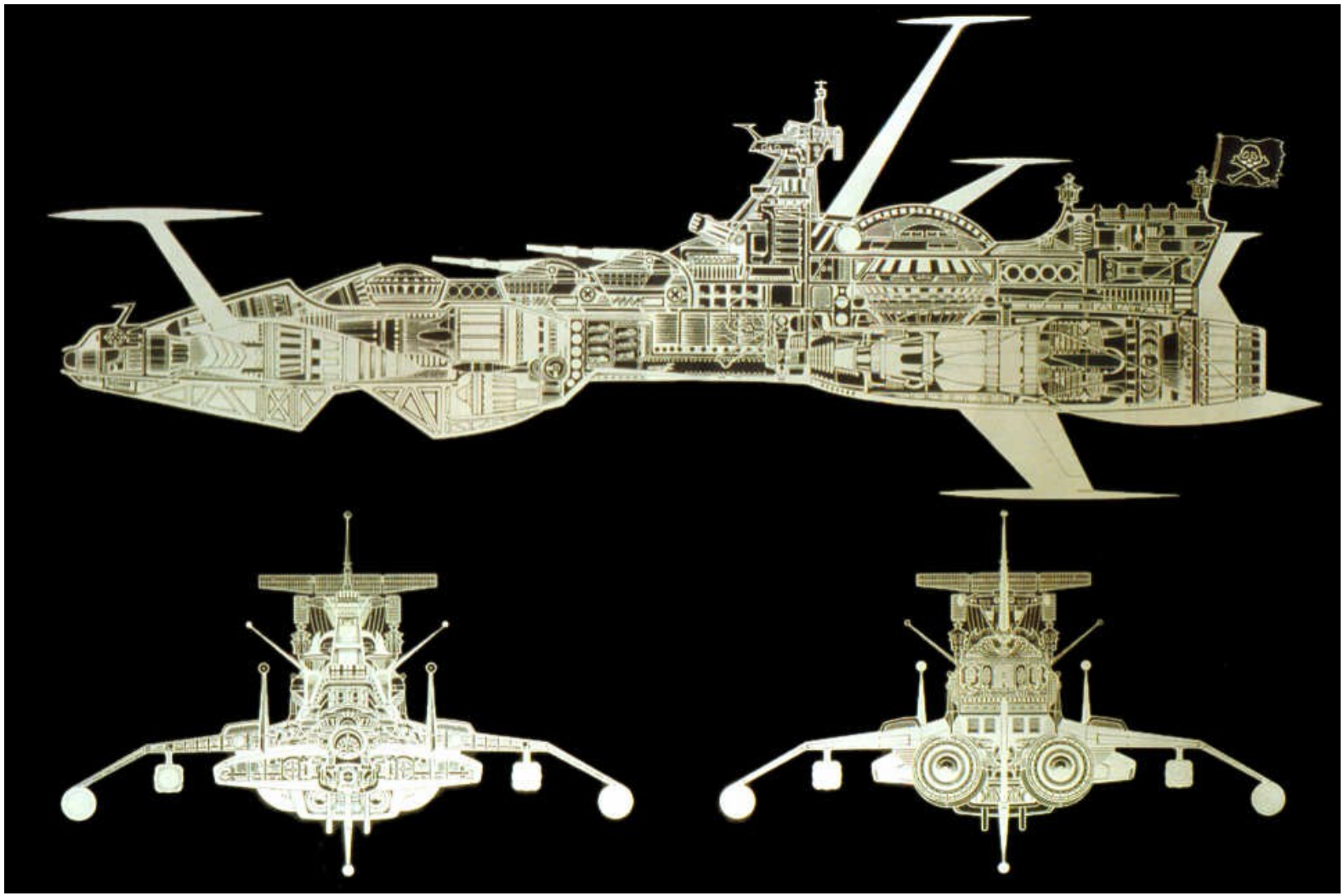


# EventLoop Vertx through Netty

```
public class EventLoopContext extends DefaultContext {

….
  public void execute(Runnable task) {
    getEventLoop().execute(wrapTask(task));
  }

  public boolean isOnCorrectWorker(EventLoop worker) {
    return getEventLoop() == worker;
  }

}
```

# EventLoop Internals



```
protected Runnable wrapTask(final Runnable task) {
    return new Runnable() {
      public void run() {
        Thread currentThread = Thread.currentThread();
        String threadName = currentThread.getName();
        try {
          vertx.setContext(DefaultContext.this);
          task.run();
        } catch (Throwable t) {
          reportException(t);
        } finally {
          if (!threadName.equals(currentThread.getName())) {
            currentThread.setName(threadName);
          }
        }
        if (closed) {
          unsetContext();
        }
      }
```

# EventLoop Vertx through Netty

The benefit of executing

the task in the event loop is

that you don't need to worry

about any **synchronization** or **concurrency problems**.

The runnable will get executed

in the same thread as all

other events that are related to the channel.

# Event Loops

When the data arrives from the outside or from inside,

the event loop thread wakes up,

executes any callback function  registered for the

specific event type,

and returns to its wait state until a new event occurs

Vertx

creates as many event loop threads as the number of CPU cores

# Event Loops

When the data arrives from the outside or from inside,

the event loop thread wakes up,

executes any callback function  registered for the

specific event type,

and returns to its wait state until a new event occurs

Vertx

creates as many event loop threads as the number of CPU cores

# Event Loops

The unit of execution is a Verticle

which reacts to event messages,

and communicates sending event messages.

Decoupling communication

with event handlers and messages

enables location transparency

# Golden Rule

Never block the Event Loop

Never block the Event Loop

Never block the Event Loop

If you need a blocking or long time computation code

use a separate thread for this

# Verticle

Vertx provide an abstraction in which write code like a single-thread, this abstraction is called Verticle.

In classic framework we have to write Controller or Service Object, with Vertx all communications are async with events through Verticles.

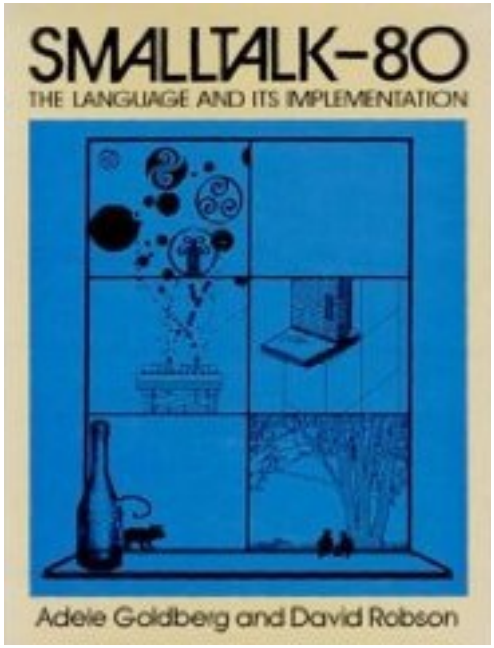
Direct calls does not exist in Vertx,

all calls are messages on Event Bus

# Message to object (roots)

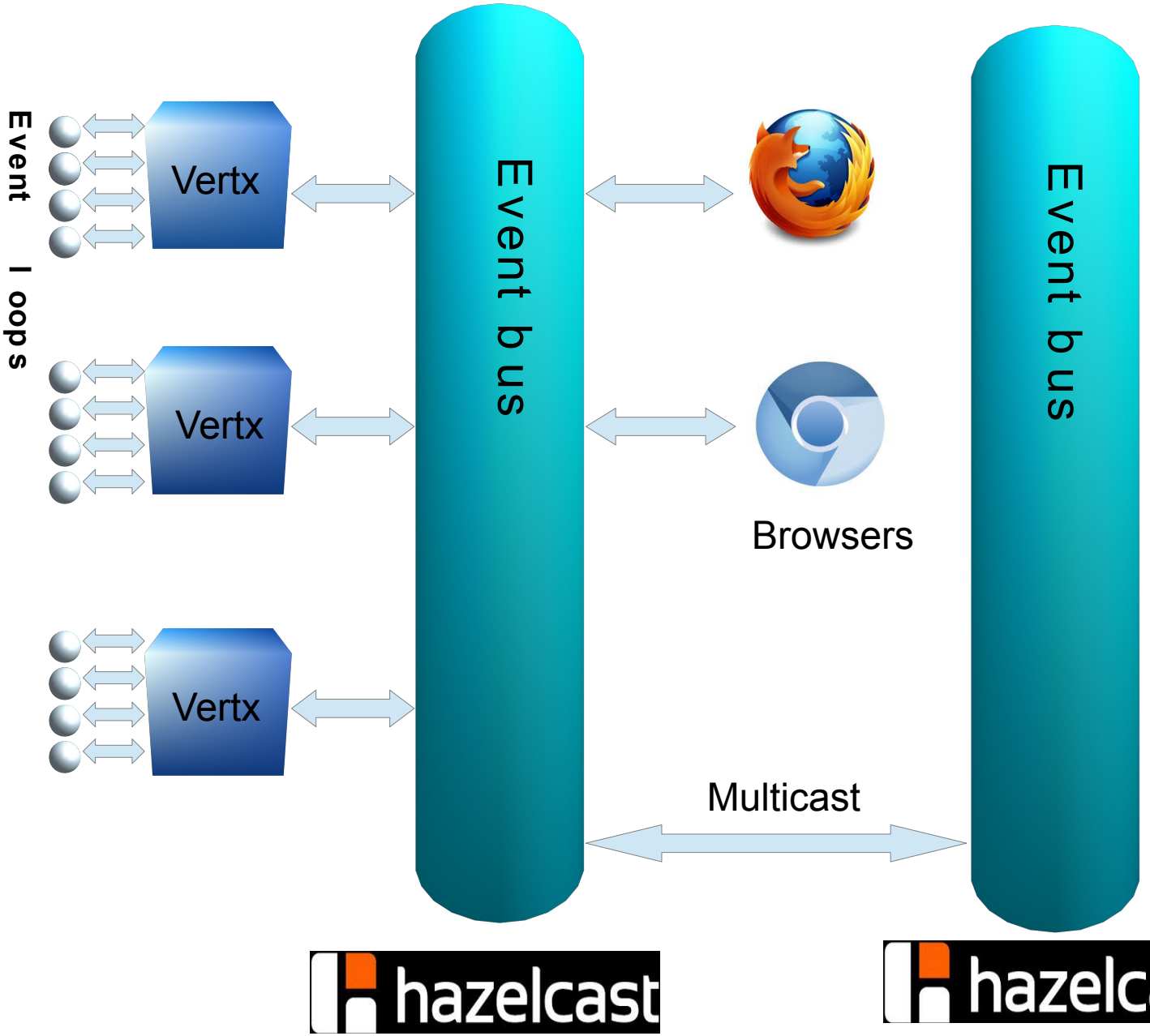
Sending Messages to Objects

all Smalltalk processing is accomplished by sending messages to objects. An initial problem solving approach in Smalltalk is to try to reuse the existing objects and message




http://en.wikipedia.org/wiki/Smalltalk

# Event Bus distributed




Location transparency

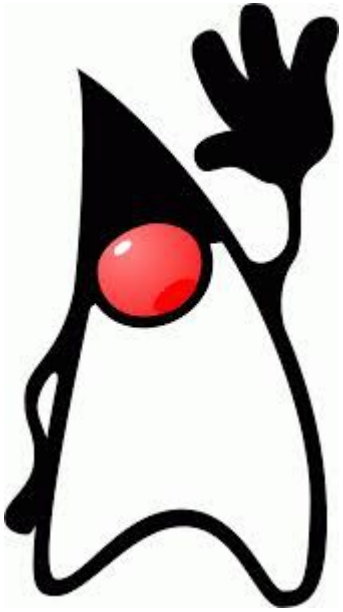
# Verticle

- Each Verticle instance is executed from only one thread
- Each Verticle instance assigned to thread/EventLoop
- Separate classloader for each Verticle
- Polyglot Verticles
- React to event with event handlers

# Verticle

```
public class MyVerticle extends Verticle {

    @Override
    public void start() {
     // register handlers
    }

    @Override
    public void stop() {
      ...
    }
}
```

# Polyglot Verticles







# Verticles packaging (Module)

A module is a collection of verticles and other code and files that is packaged as a unit,

and then referenced from other Vert.x modules or applications.

The module descriptor is a JSON file called mod.json

```
{
  "main": "MyPersistor.java"
  "worker": true                  //worker verticle
  "preserve-cwd":true
   ...
}
```

# Deploy module runtime

```
var container = require('vertx/container');



var config = {
  "web_root": ".",
  "port": 8080
};


//downloaded form vertx repository
container.deployModule("io.vertx~mod-web-server~2.0.0-final", config);
```

# Worker Verticle

Vertx provide Worker verticles

that run on a separate thread

to perform blocking operations

without block the Eventloop

# Handlers

```
EventBus bus = vertx.eventBus();
     Handler<Message> handler = new Handler<Message>() {
          @Override
          public void handle(Message message) {
               //doSomething
          }
     }

bus.registerHandler("com.codemotion.firsthandler", handler);
```

# Pub Sub

```
EventBus bus = vertx.eventBus();
bus.publish("com.codemotion.firsthandler", "Hello world");
```

publish mean broadcast to all subscribers

# P2P

```
EventBus bus = vertx.eventBus();
bus.send("39.216667.Nord-9.116667.Est", "Hello world");
```

send mean point to point, only one subscriber

# Sender

```
bus.send("39.216667.Nord-9.116667.Est", "This is a message !",
         new Handler<Message<String>>() {

    @Override
    public void handle(Message<String> message) {
        String received = message.body();
    }
});
```

# Receiver

```
Handler<Message> handler = new Handler<Message<String>>() {
      @Override
      public void handle(Message<String message) {
            String received = message.body();
           message.reply("This is a reply");
      }
 }

bus.registerHandler("39.216667.Nord-9.116667.Est", handler);
```

# Message from the UI

```
<script src="http://cdn.sockjs.org/sockjs-0.3.4.min.js"></script>
<script src='vertxbus.js'></script>

<script>

    var eb = new vertx.EventBus('http://localhost:8080/eventbus');

    eb.onopen = function() {

      eb.registerHandler('some-address', function(message) {

        console.log('received a message: ' + JSON.stringify(message);

      });

      eb.send('some-address', {name: 'tim', age: 587});

    }

</script>
```

# Goodies

- HTTP/HTTPS servers/clients
- WebSockets support
- SockJS support
- Timers
- Buffers
- Streams and Pumps
- Routing
- Async File I/O
- Shared Data
- Embeddable

- Module Repo

  (http://modulereg.vertx.io/)

  - WebServer
  - SessionManager
  - Auth manager
  - Persistors (Mongo, JDBC)
  - Spring
  - RxJava
  - Many others

- Compile on the fly
  (deploy .java verticle)

# Notification of reply failure

```
Logger logger = container.logger();
getVertx().eventBus().sendWithTimeout("test.address", "This is a
message", 1000, new Handler<AsyncResult<Message<String>>>() {
    public void handle(AsyncResult<Message<String>> result) {
        if (result.succeeded()) {
            Logger.info("I received a reply " + message.body);
        } else {
            ReplyException ex = (ReplyException)result.cause();
            logger.error("Failure type: " + ex.failureType());
            logger.error("Failure code: " + ex.failureCode());
            logger.error("Failure message: " + ex.message());
            // restart dead verticle
        }
    }
});
```

# Timers

```
long timerID = vertx.setPeriodic(1000, new
Handler<Long>() {
    public void handle(Long timerID) {
        log.info("And every second this is printed");
    }
});

log.info("First this is printed");
```

# RxJava

Futures are Expensive to Compose

“Futures are straight-forward to use for a single level of asynchronous execution but they start to add non-trivial complexity when they're nested. “



https://github.com/Netflix/RxJava/wiki

# RxJava

Reactive

Functional reactive offers efficient execution and composition by providing a collection of operators capable of filtering, selecting, transforming, combining and composing Observable's.



https://github.com/Netflix/RxJava/wiki

# RxJava

“The Observable data type can be thought of as a "push" equivalent to Iterable which is "pull". With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive.

By contrast with the Observable type, the producer pushes values to the consumer whenever values are available. This approach is more flexible, because values can arrive synchronously or asynchronously.”



https://github.com/Netflix/RxJava/wiki

# RxJava

```
def simpleComposition() {

   customObservableNonBlocking()
   .skip(10)// skip the first 10
   .take(5)// take the next 5
   .map({ stringValue -> return stringValue+ "transf"})
   .subscribe({ println "onNext => " + it})
}
```

https://github.com/Netflix/RxJava/wiki

# RxJava

"The Observable type adds two missing semantics to the Gang of Four's Observer pattern, which are available in the Iterable type:

1) The ability for the producer to signal to the consumer that there is no more data available.

2)The ability for the producer to signal to the consumer that an error has occurred."

https://github.com/Netflix/RxJava/wiki

# RxJava

RxJava a library (Java, Groovy, Scala, Clojure, JRuby) for composing asynchronous and event-based programs by using observable sequences .

It extends the observer pattern to support sequences of data/events and adds operators that allow you to compose sequences together declaratively while abstracting away concerns about things like low-level threading, synchronization, thread-safety, concurrent data structures, and non-blocking I/O.

https://github.com/Netflix/RxJava/wiki

```
def Observable<T> getData(int id) {
    if(availableInMemory) {// sync
        return Observable.create({ observer ->
            observer.onNext(valueFromMemory);
            observer.onCompleted();
        })
    } else { //Async
        return Observable.create({ observer ->
        executor.submit({
        try {
            T value = getValueFromRemoteService(id);
            observer.onNext(value);
            observer.onCompleted();
        }catch(Exception e) {
            observer.onError(e);
        }
    })
    });
}}
```

# RxJava

**No differences from**

**the client perspective,**

**an Observable in both cases**

http://netflix.github.io/RxJava/javadoc/rx/package-tree.html

# Mod-rxvertx (RxJava in Vertx)

```
RxEventBus rxEventBus = new RxEventBus(vertx.eventBus());

rxEventBus.<String>registerHandler("foo").subscribe(
    new Action1<RxMessage<String>>() {
        public void call(RxMessage<String> message) {
            message.reply("pong!");// Send a single reply
        }
});

Observable<RxMessage<String>> obs = rxEventBus.send("foo", "ping!");

obs.subscribe(
  new Action1<RxMessage<String>>() {
    public void call(RxMessage<String> message) {
      // Handle response
    }
  },
  new Action1<Throwable>() {
    public void call(Throwable err) {
     // Handle error
    }
  }
);
```

# Automatic failover (resilient)

When a module is run with HA, if the Vert.x instance where it is running fails, it will be re-started automatically on another node of the cluster, this is module fail-over.

To run a module with HA, add the -ha switch when running it on the command line:

```
vertx runmod com.acme~my-mod~2.1 -ha
```



See details in `org.vertx.java.platform.impl.HAManager`

# HTTP (Java)

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

server.requestHandler(new Handler< HttpServerRequest >() {
    public void handle(HttpServerRequest request) {
        request.response().write("Hello world").end();
    }
});

server.listen(8080, "localhost");
```

# HTTP (JavaScript)

```
var vertx  = require('vertx');

var server = vertx.createHttpServer();


server.requestHandler(function(request) {
      request.response.write("Hello world").end();
});


server.listen(8080, "localhost");
```

# HTTP (Scala)

```
vertx.createHttpServer.requestHandler {
  req: HttpServerRequest => req.response.end("Hello World!")
}.listen(8080)
```

# HTTP (Clojure)

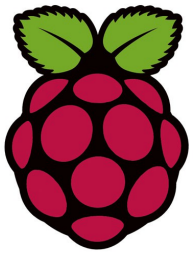
```
(ns demo.verticle
  (:require [vertx.http :as http]
            [vertx.stream :as stream]))

(defn req-handler [req]
  (-> (http/server-response req)
      (stream/write "Hello world !")
      (http/end)))

(-> (http/server)
    (http/on-request req-handler)
    (http/listen 8080))
```

# HTTP (JRuby)

```
require "vertx"

server = Vertx::HttpServer.new

server.request_handler do |request|
      request.response.write("Hello world").end;
end

server.listen(8080, "localhost")
```

# Vertx on RaspberryPi

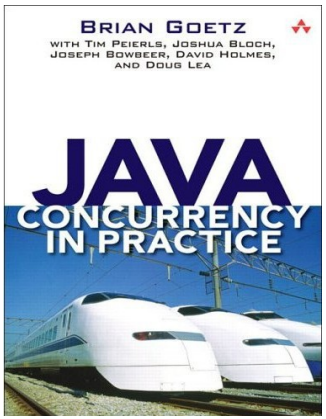
```
public class HardwareVerticle extends Verticle {

    public void start() {

        final GpioController gpio = GpioFactory.getInstance();

        System.out.println("GPIO LOADED");

        final GpioPinDigitalInput myButton =
gpio.provisionDigitalInputPin(RaspiPin.GPIO_06, PinPullResistance.PULL_DOWN);

   myButton.addListener(new GpioPinListenerDigital() {

     @Override

     public void handleGpioPinDigitalStateChangeEvent(GpioPinDigitalStateChangeEvent event){

          System.out.println(new Date() + "Button change");

          vertx.eventBus().publish("buttonbus",String.valueOf(event.getState().getValue()));

     }

  });

 }

}
```

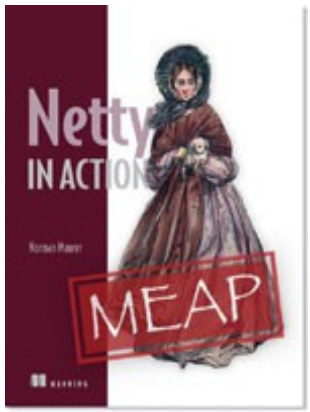


http://szimano.org/how-to-use-vert-x-to-make-your-raspberrypi-talk-to-your-browser/

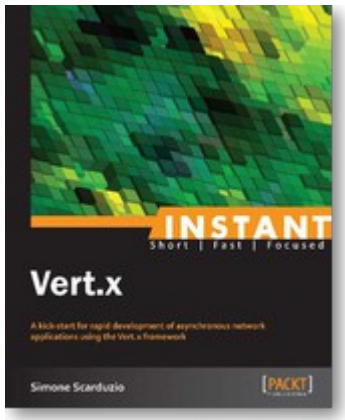
# References

- http://www.reactivemanifesto.org/
- http://vertx.io/
- http://netty.io/
- https://github.com/Netflix/RxJava
- http://lampwww.epfl.ch/~imaier/pub/DeprecatingObserversTR2010.pdf
- http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf
- http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf
- http://www.cs.bgu.ac.il/~spl051/Personal_material/Practical_sessions/Ps_12/ps12.html







# References speaker

https://twitter.com/desmax74

http://www.slideshare.net/desmax74

it.linkedin.com/in/desmax74

# Thanks for your attention !