

# VERT.X

Vert.X

high performance polyglot application toolkit

**Xavier MARIN**

@XavMarin

<https://github.com/Giwi>

Architecte chez @cmarkea

CTO chez @qaobee



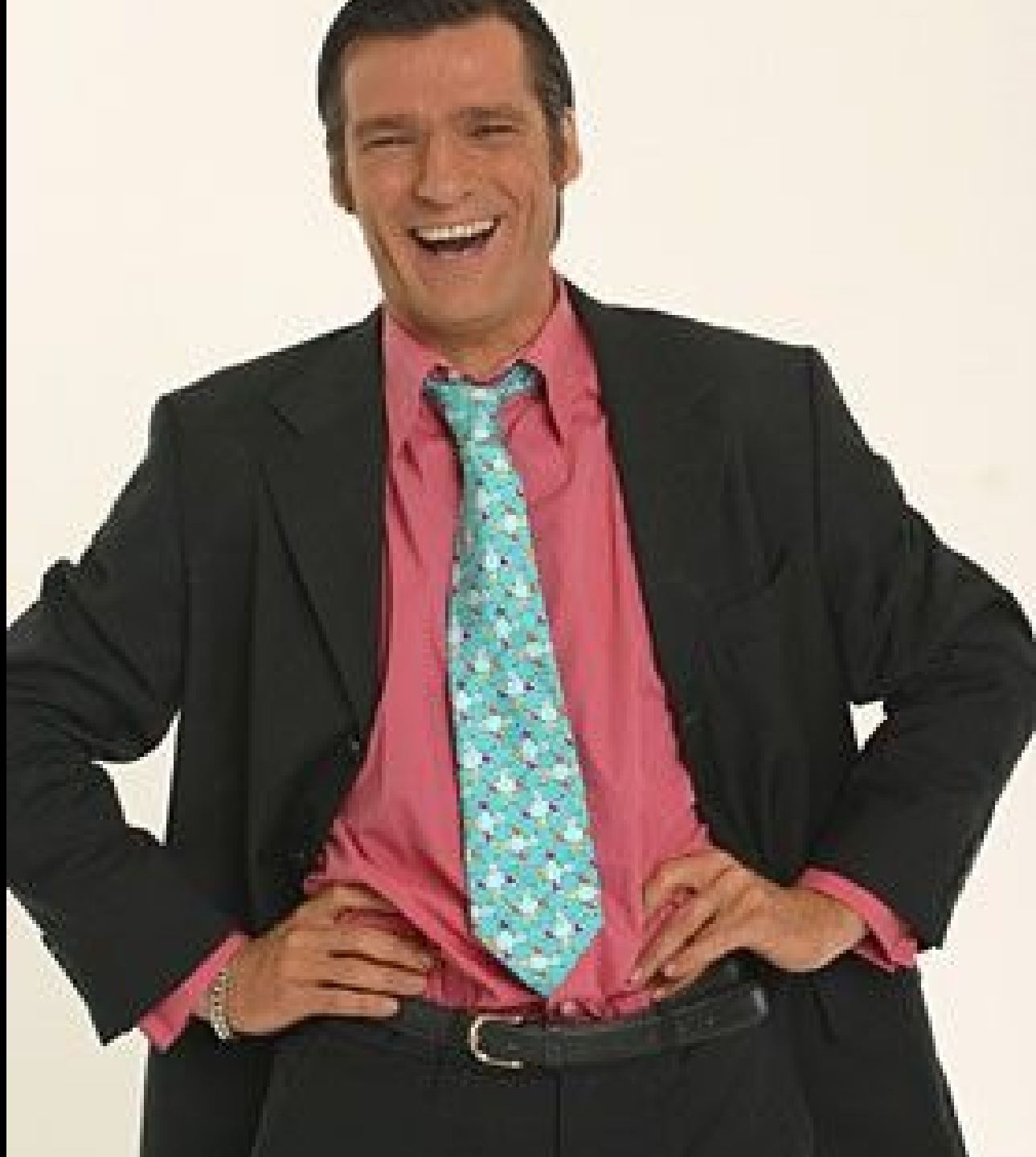
# Petite histoire

- Créé par Tim Fox @timfox
- en 2011 chez VMWare
- sous le nom de Node.x
- 2012 : devient Vert.X
- 2013 : passe dans la fondation Eclipse
- 24/06/2015 sortie de la v3
- environ 180 contributeurs

<https://www.parleys.com/search/vert.x/PRESENTATIONS>

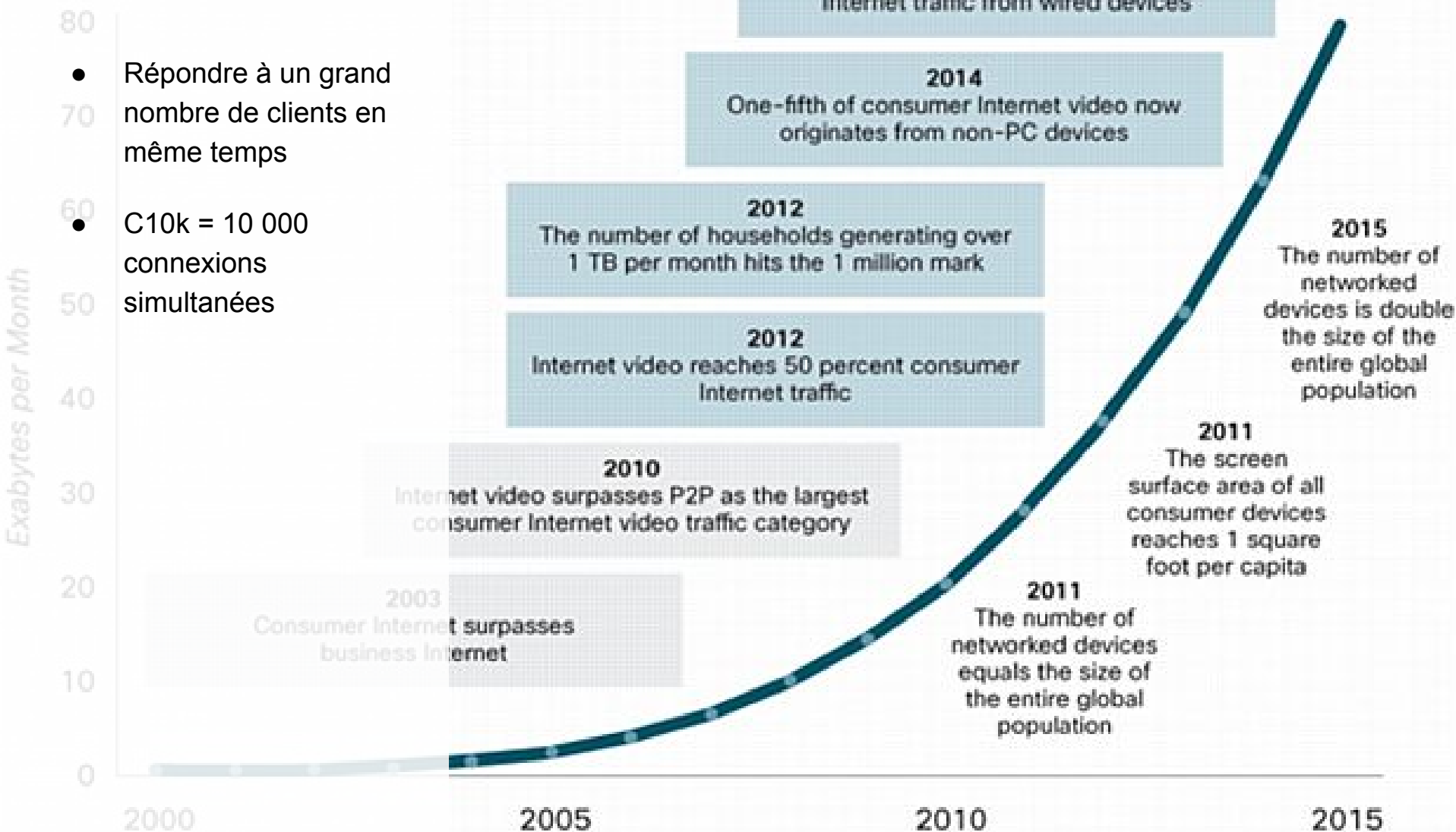
## Vocabulaire pour commerciaux

Simple threadSafe concurrent  
asynchronous eventDriven  
reactive eventBus over a  
scalable polyglot embeddable  
toolkit plateforme



## Le problème

- Répondre à un grand nombre de clients en même temps
- C10k = 10 000 connexions simultanées



# C10k

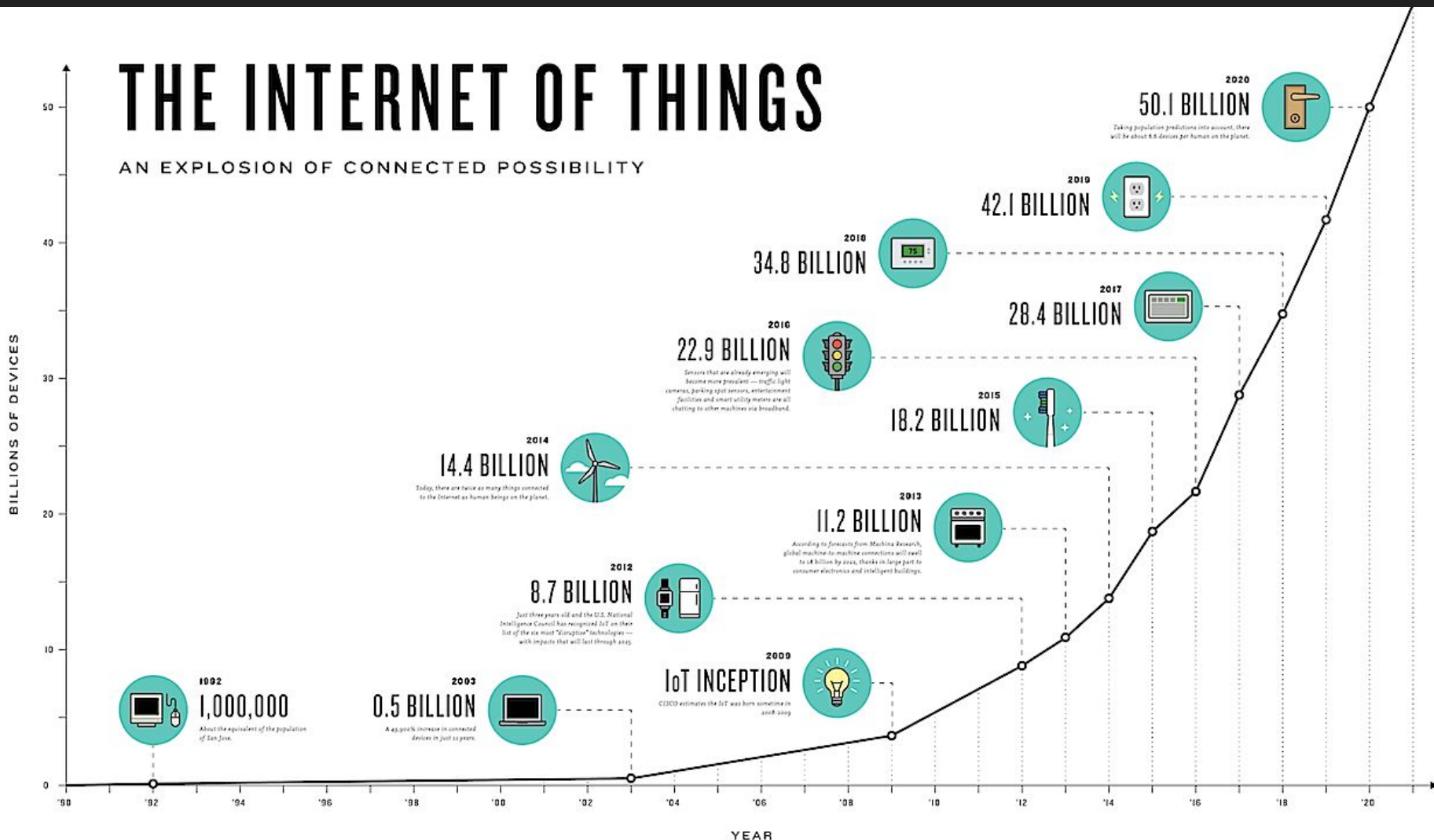
Global Mobile Traffic as % of Total Internet Traffic, 12/08 – 5/13  
(with Trendline Projection to 5/15E)



# C10k

## THE INTERNET OF THINGS

AN EXPLOSION OF CONNECTED POSSIBILITY



# Multi Thread vs Mono Thread + boucle d'événement





# Multi Thread vs Mono Thread + boucle d'événement

J2EE					Mono Thread + boucle d'événement			
Statique	EJB	JSP	Servlet			Traitement	TPL	Statique
x	x	x	x	JBoss WebSphere Tomee				

# Multi Thread vs Mono Thread + boucle d'événement

J2EE					Mono Thread + boucle d'événement			
Statique	EJB	JSP	Servlet			Traitement	TPL	Statique
x	x	x	x	JBoss WebSphere Tomee				
x		x	x	Tomcat				

# Multi Thread vs Mono Thread + boucle d'événement

J2EE					Mono Thread + boucle d'événement			
Statique	EJB	JSP	Servlet			Traitement	TPL	Statique
x	x	x	x	JBoss WebSphere Tomee				
x		x	x	Tomcat				
x			x	Jetty				

# Multi Thread vs Mono Thread + boucle d'événement

J2EE					Mono Thread + boucle d'événement			
Statique	EJB	JSP	Servlet			Traitement	TPL	Statique
x	x	x	x	JBoss WebSphere Tomee				
x		x	x	Tomcat				
x			x	Jetty				
Serveur HTTP								
x				Apache HTTPD				

# Multi Thread vs Mono Thread + boucle d'événement

J2EE					Mono Thread + boucle d'événement			
Statique	EJB	JSP	Servlet			Traitement	TPL	Statique
x	x	x	x	JBoss WebSphere Tomee	NodeJS	x	x	x
x		x	x	Tomcat	Vert.X	x	x	x
x			x	Jetty	Lagom	x	x	x
Serveur HTTP								
x				Apache HTTPD	NginX			x

# Système réactif vs programmation réactive

## The Reactive Manifesto

*Published on September 16 2014. (v2.0)*

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and [scalable](#). This makes them easier to develop and amenable to change. They are significantly more tolerant of failure and when [failure](#) does occur they meet it with elegance rather than disaster. Reactive Systems are highly responsive, giving [users](#) effective interactive feedback.

Reactive Systems are:

**Responsive:** The [system](#) responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

**Resilient:** The system stays responsive in the face of [failure](#). This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by [replication](#), containment, [isolation](#) and [delegation](#). Failures are contained within each [component](#), isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

**Elastic:** The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the [resources](#) allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve [elasticity](#) in a cost-effective way on commodity hardware and software platforms.

**Message Driven:** Reactive Systems rely on [asynchronous message-passing](#) to establish a boundary between components that ensures loose coupling, isolation, [location transparency](#), and provides the means to delegate [errors](#) as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying [back-pressure](#) when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. [Non-blocking](#) communication allows recipients to only consume [resources](#) while active, leading to less system overhead.

<http://www.reactivemanifesto.org>

Les applications modernes font face à de nouveaux défis, bla bla bla

Responsive : répond sur un laps de temps

Résilient : répond même en cas d'échec

Elastique : répond quel que soit la charge

Message-driven : repose sur la communication événementielle asynchrone

# Caractéristiques

## Réactif

- Responsive
  - Tient la charge (scalable)
  - Robuste (resilience)
- Message-driven
  - événementiel asynchrone

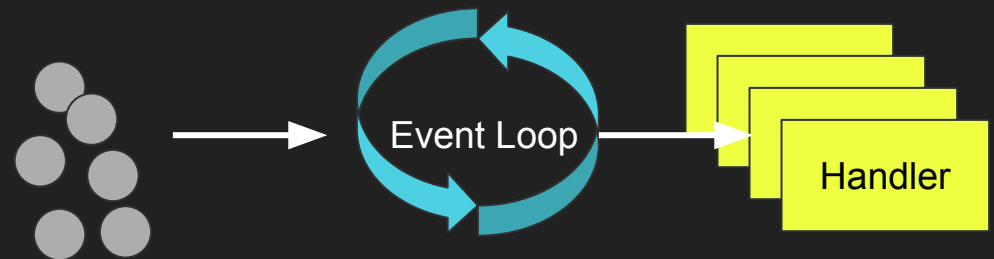
## Polyglotte

- Pas que en Java
  - Groovy, Ruby, Javascript
- S'exécute sur la VM

message-driven



reactor pattern



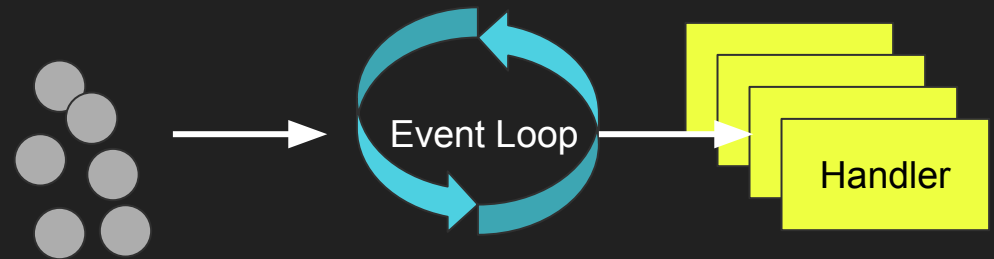
Les handlers sont toujours appelés dans le même thread (event loop)

Les handlers ne bloquent jamais le thread qui les appelle

# Qu'est-ce que ● ?

- Un message
- Une notification
- Une requête HTTP
- Une commande, une instruction
- Un fichier
- Un résultat, un rapport, une erreur

```
void operation(param1, param2, Handler< ● > {  
    // ...  
    handler.handle(●);  
    // ...  
}
```



```
void handle(●) {  
    // faire un truc avec ●  
}
```




# Modèle de développement asynchrone

```
vertx.createHttpServer()  
    .requestHandler(  
        req -> req.response().end("Hello there!")  
    )  
    .listen(8080,  
        ar -> {  
            if (ar.failed()) {  
                System.out.println("The server failed to start");  
            } else {  
                System.out.println("Server started on 8080");  
            }  
        }  
    );
```

**ar : AsyncResult**

- status : success ou failed
- result : si success
- cause : si failed

A collection of vintage tools is arranged on a dark wooden surface. The tools include a hammer with a wooden handle, an axe, a saw, a pair of gloves, a metal mug, and various other hand tools. The scene is lit with warm, natural light, creating a rustic and workshop-like atmosphere.

# JVM polyglotte

Dois-je coder en java pour exécuter  
un programme sur la JVM?

Oui, mais pas que, fais-toi plaisir!

# JVM polyglotte

On peut programmer en plus de 200 langages

Java, JavaFX, Ceylon, Gosu, Groovy, Clojure, Rhinoceros, Nashorn, Mirah, Scala, JRuby, Xtend, JPython, Fantom, Oxygene, Kotlin, ...

L'intérêt : à chaque problème sa solution

**“Quand on n’a qu’un marteau dans la main, les vis ressemblent à des clous”**

# Polyglotte

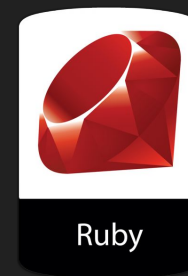


```
var vertx = require('vertx-js/vertx');

var server = vertx.createHttpServer();
server.requestHandler(function (request) {
  var response = request.response();
  response.putHeader('content-type', 'text/plain');
  response.end('Hello World!');
});

server.listen(8080);
```

# Polyglotte



```
server = vertx.create_http_server()

server.request_handler() { |request|
  response = request.response()
  response.put_header("content-type", "text/plain")
  response.end("Hello World!")
}

server.listen(8080)
```

# Polyglotte



```
def server = vertx.createHttpServer()

server.requestHandler({ request ->
    def response = request.response()
    response.putHeader("content-type", "text/plain")
    response.end("Hello World!")
})

server.listen(8080)
```

# Polyglotte



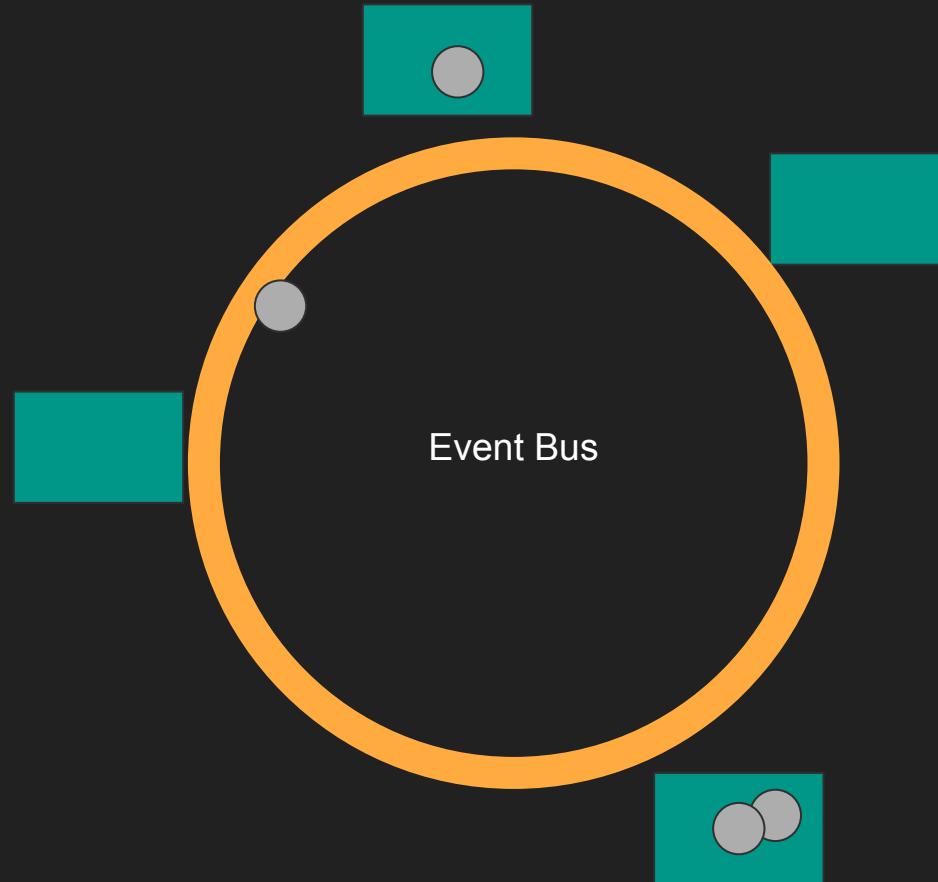
```
public class Main {  
  
    public static void main(String[] args) {  
  
        HttpServer server = vertx.createHttpServer();  
        server.requestHandler(request -> {  
            HttpServerResponse response = request.response();  
  
            response.putHeader("content-type", "text/plain");  
            response.end("Hello World!");  
        });  
        server.listen(8080);  
    }  
  
}
```

# L'event bus

- Point à point
- Publish / Subscribe
- Requête / réponse

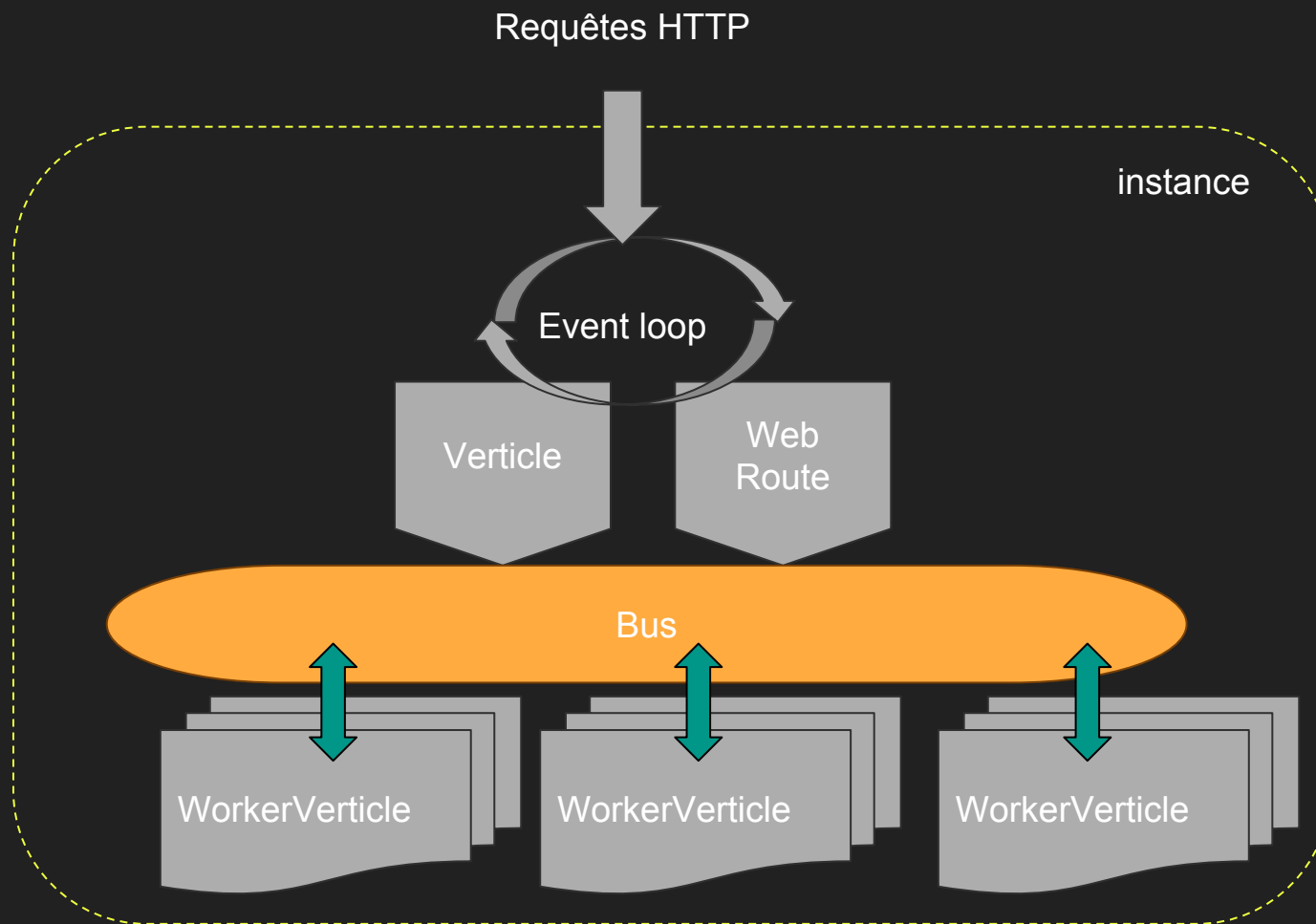
Les messages sont reçus par des handlers et déposés en utilisant l'eventLoop

Chaque instance de Vert.X a accès à l'eventBus



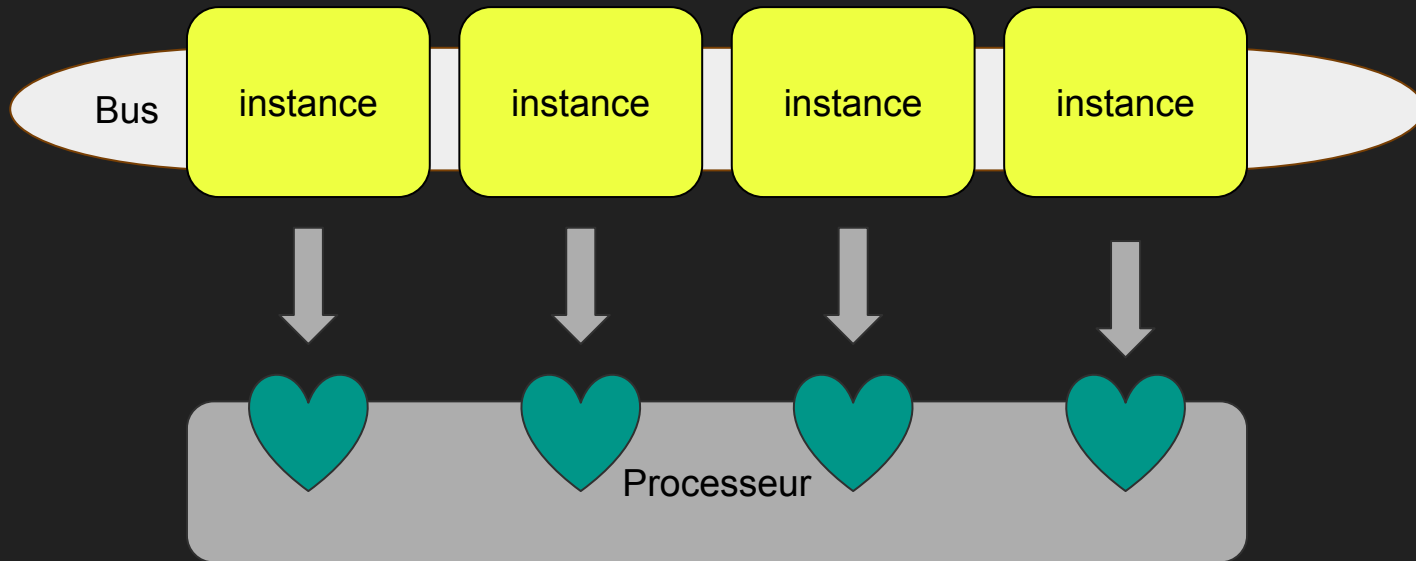


# Architecture



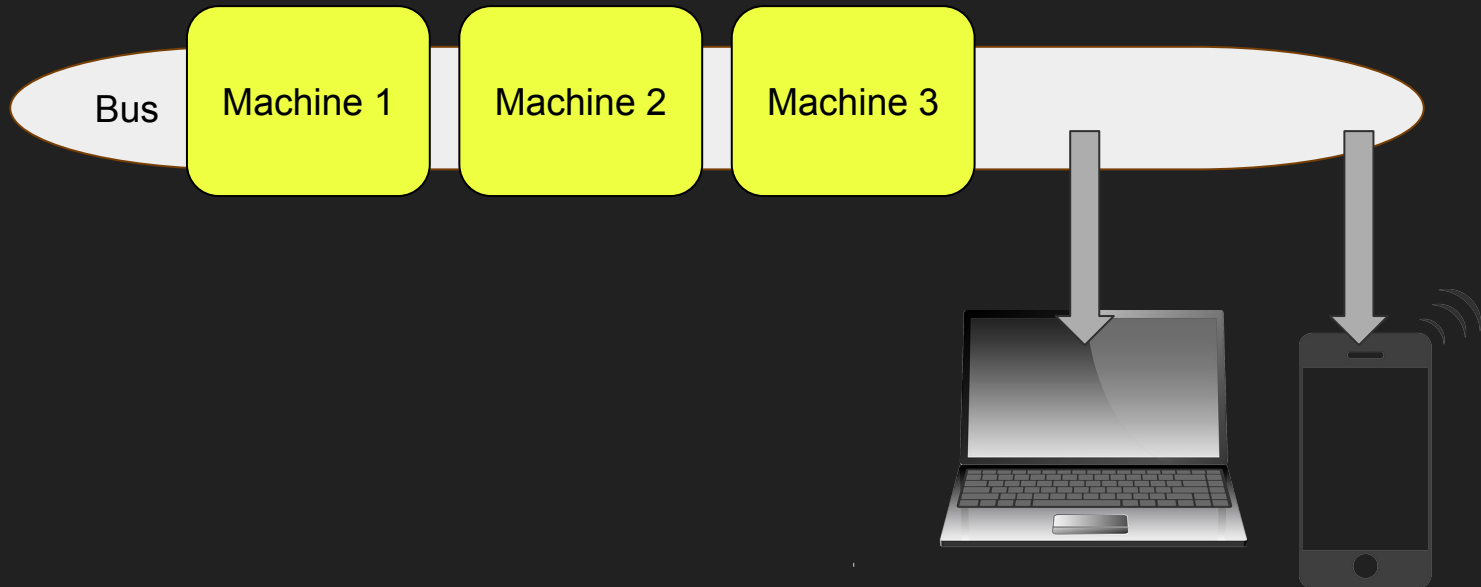
# Architecture

Multi-instances



# Architecture

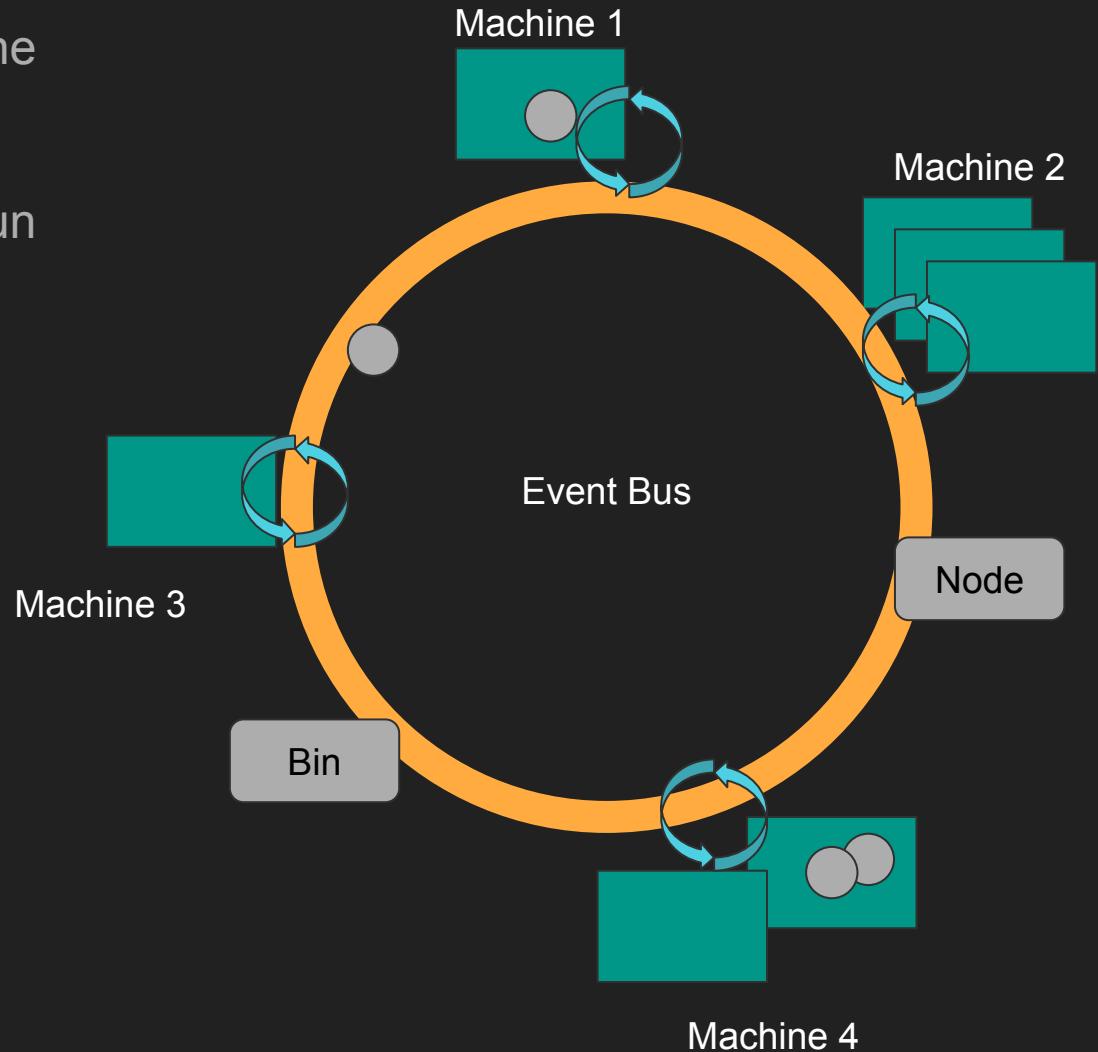
Cluster



# L'event bus

L'eventBus permet une communication distribuée.

Presque tout peut publier un message sur le bus.



# Event bus

```
EventBus eb = vertx.eventBus();
MessageConsumer<String> consumer = eb.consumer("news.uk.sport", message -> {
    System.out.println("I have received a message: " + message.body());
    message.reply("how interesting!");
}).completionHandler(res -> {
    if (res.succeeded()) {
        System.out.println("Registration has reached all nodes");
    } else {
        System.out.println("Registration failed!");
    }
});

consumer.unregister(res -> {
    if (res.succeeded()) {
        System.out.println("Un-registration has reached all nodes");
    } else {
        System.out.println("Un-registration failed!");
    }
});
```

# Event bus

## Publish / Subscribe

```
eventBus.publish("news.uk.sport", "You kicked a ball");
```

## Point à point

```
eventBus.send("news.uk.sport", "You kicked a ball", ar -> {  
    if (ar.succeeded()) {  
        System.out.println("reply: " + ar.result().body());  
    }  
});
```

## Ajout d'en-tête

```
DeliveryOptions options = new DeliveryOptions();  
options.addHeader("some-header", "some-value");  
eventBus.send("news.uk.sport", "You kicked a ball", options);
```

# Vertices : un modèle “agent” ou presque

Les vertices sont des bouts de code qui sont déployés et exécutés par Vert.X peu importe le langage.

```
vertx.deployVerticle("mon.verticle");
```

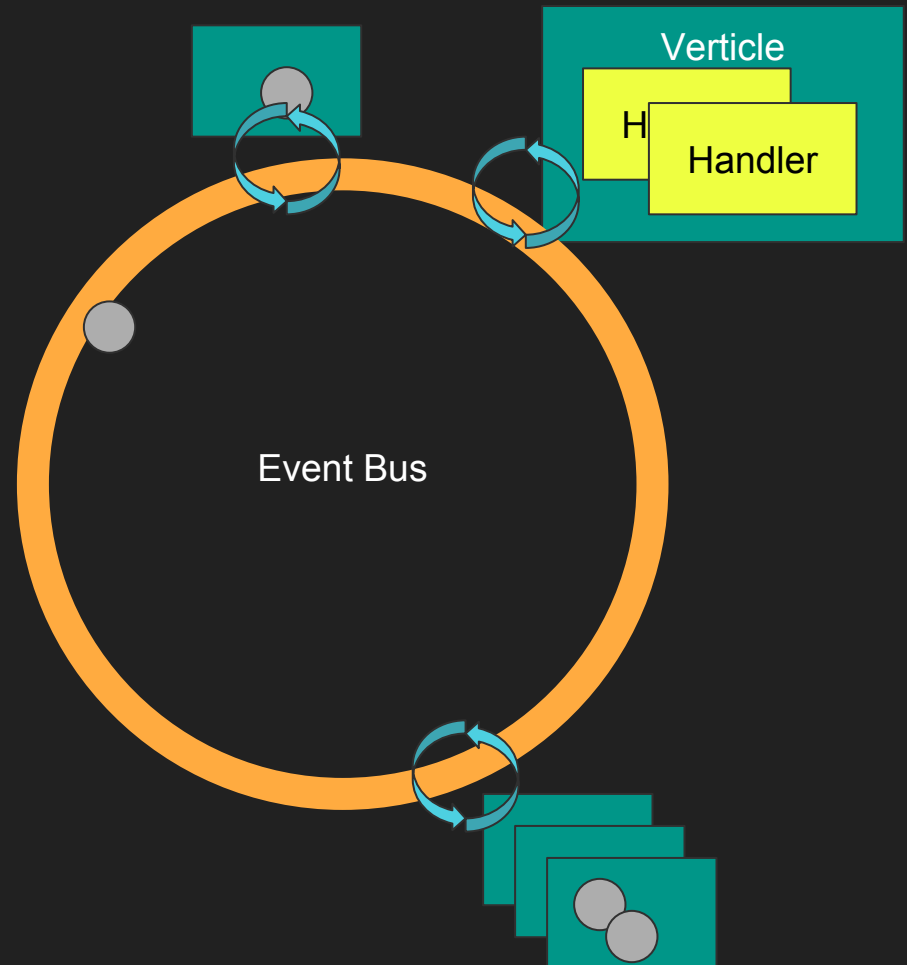
Ils ont un cycle de vie : start et stop.

Ils interagissent en utilisant des événements ou des messages.

Ils peuvent être instanciés plusieurs fois et se respawn en cas d'échec.

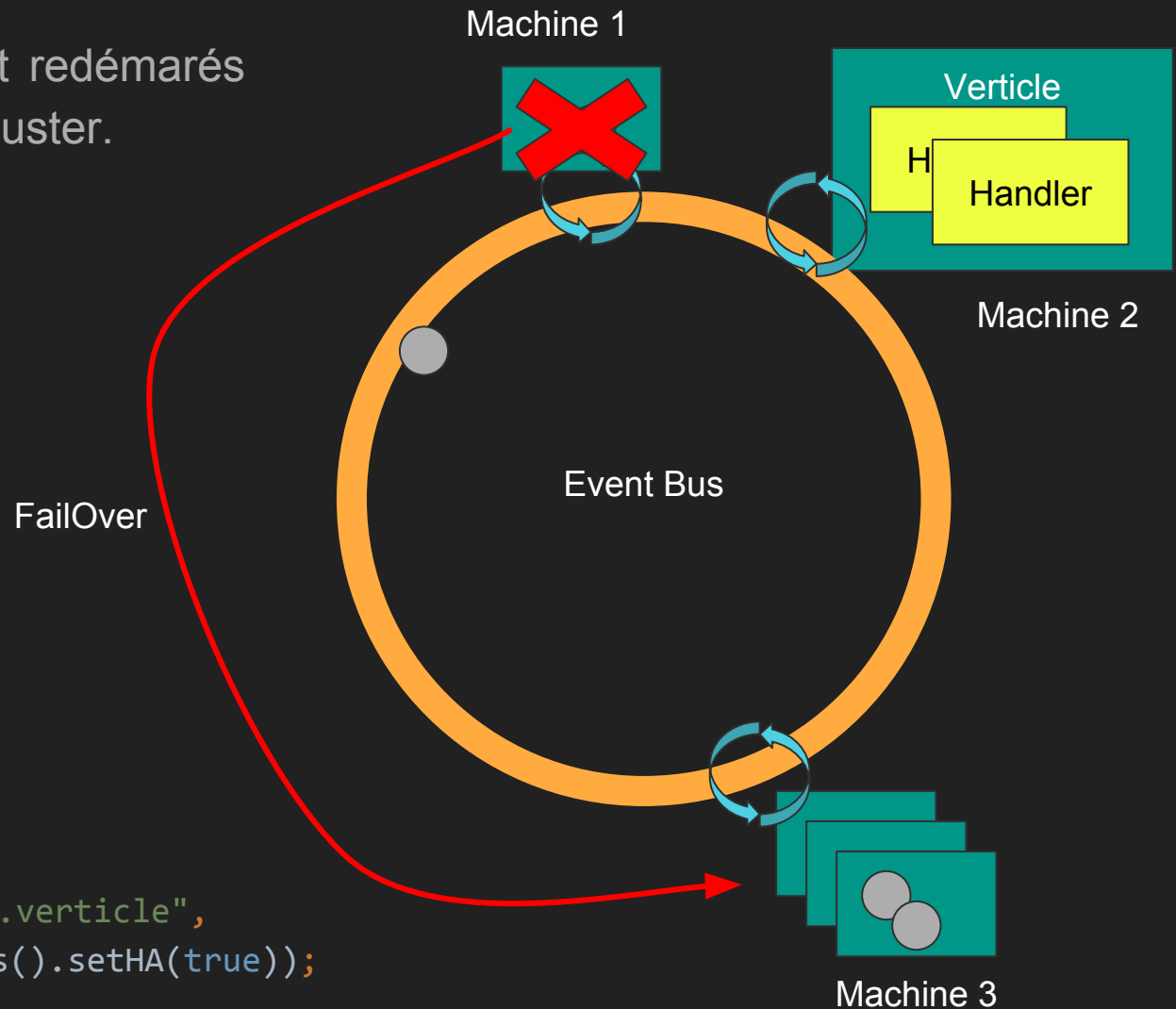
Les workerVertices traitent le bloquant

```
vertx.deployVerticle("mon.verticle",  
    new DeploymentOptions().setInstances(3));
```



# Verticles : un modèle “agent” ou presque

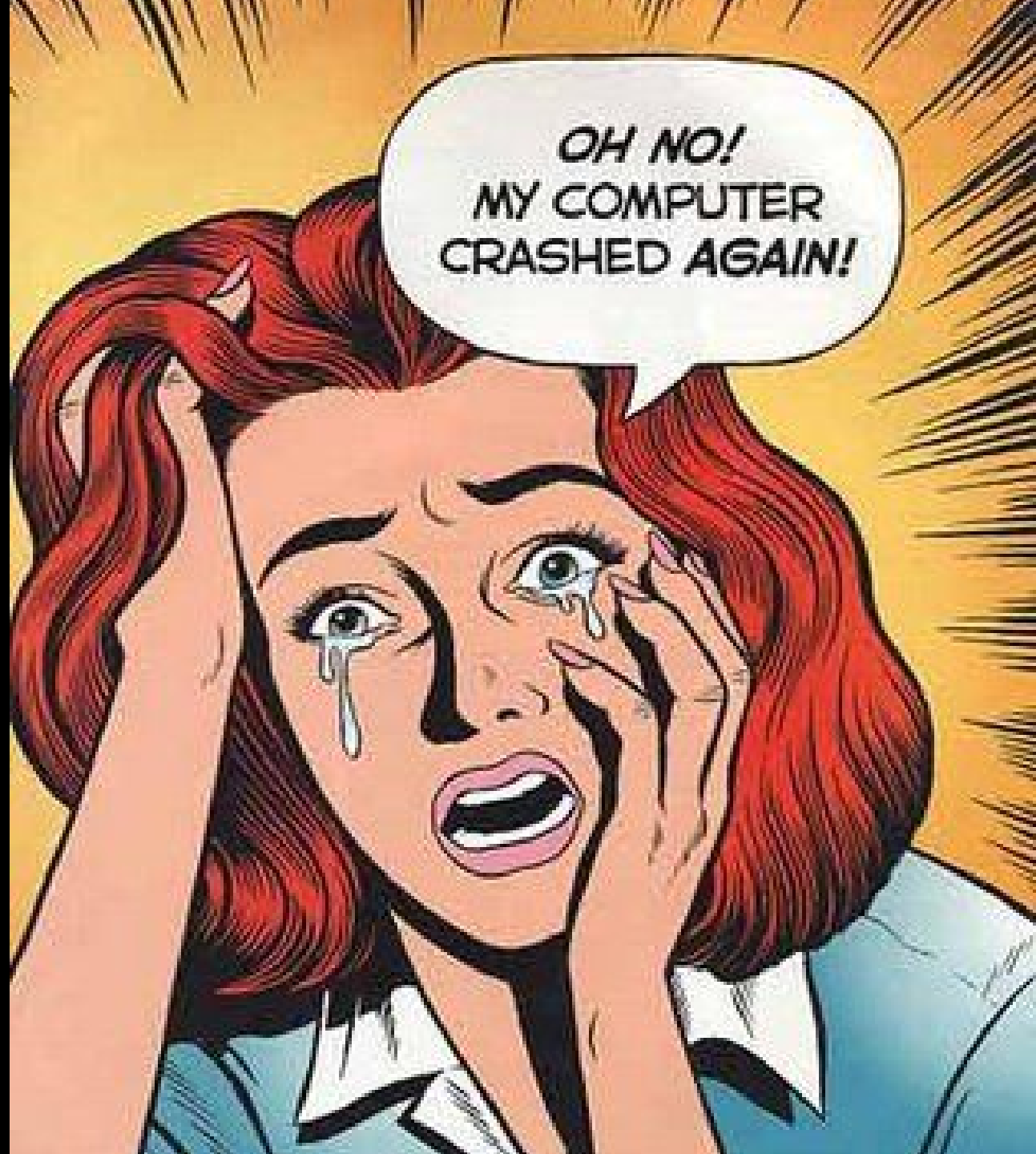
Les verticles morts sont redémarrés sur un autre noeud du cluster.



```
vertx.deployVerticle("mon.verticle",  
    new DeploymentOptions().setHA(true));
```

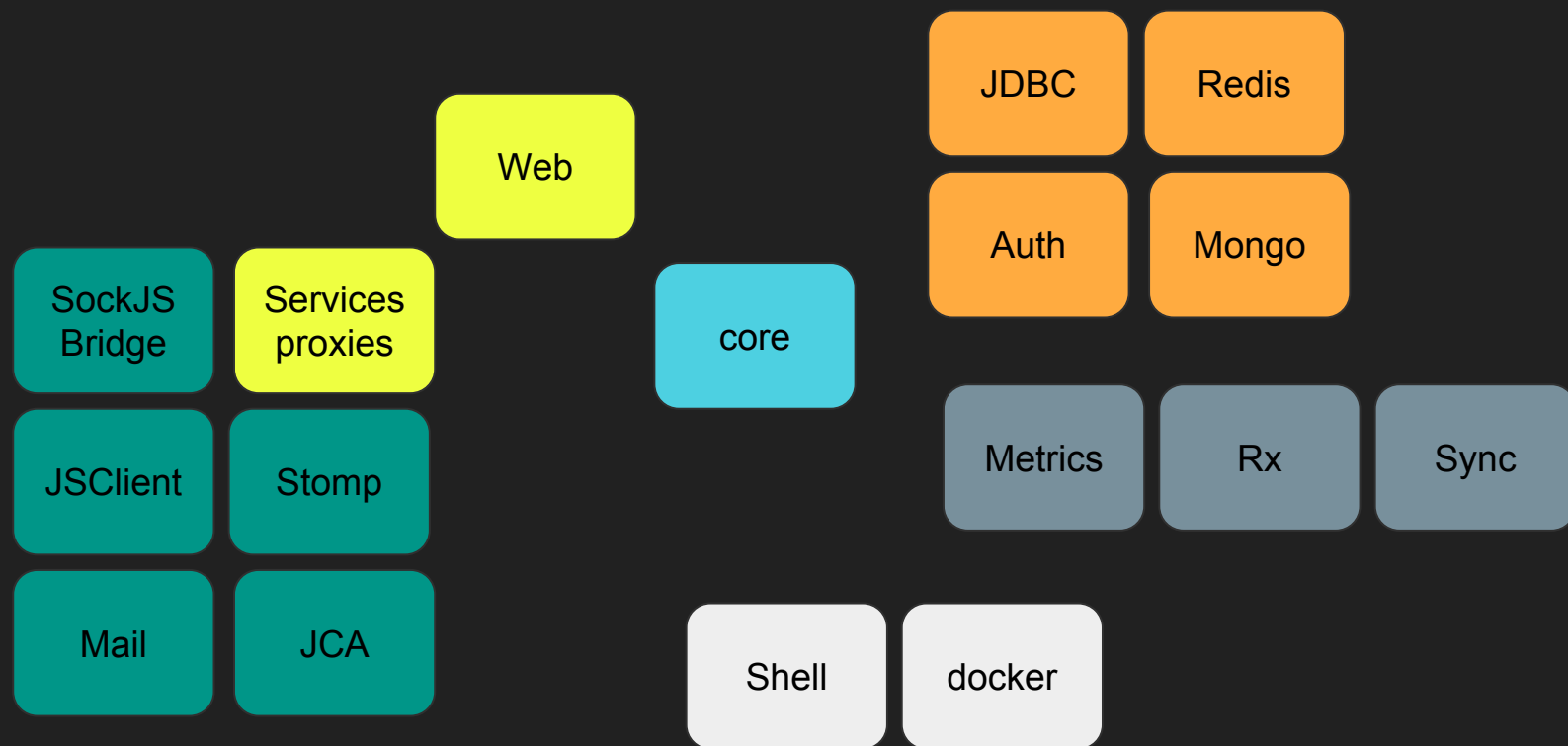


## Fail-over demo



# Stack Vert.X

<http://vertx.io/docs/>



# Stack Vert.X

- Vert.x core
- Vert.x Web
  - Routages et sous-routages
  - Sessions
  - Cookies
  - Sécurité (basic auth, shiro, JWT, ...)
  - Templates (Handlebars, Jade, MVEL, Thymeleaf)
  - SockJS
  - Static files
  - ...
- Accès aux données
  - MongoDB, JDBC, Redis, SQL Common
- Intégration
  - Mail et JCA
- Sécurité
  - basic auth, shiro, JWT, JDBC Auth
- Reactive
  - Vert.X Rx, Reactive streams
- Metrics
- Vert.X unit

# Accès aux données de manière asynchrone

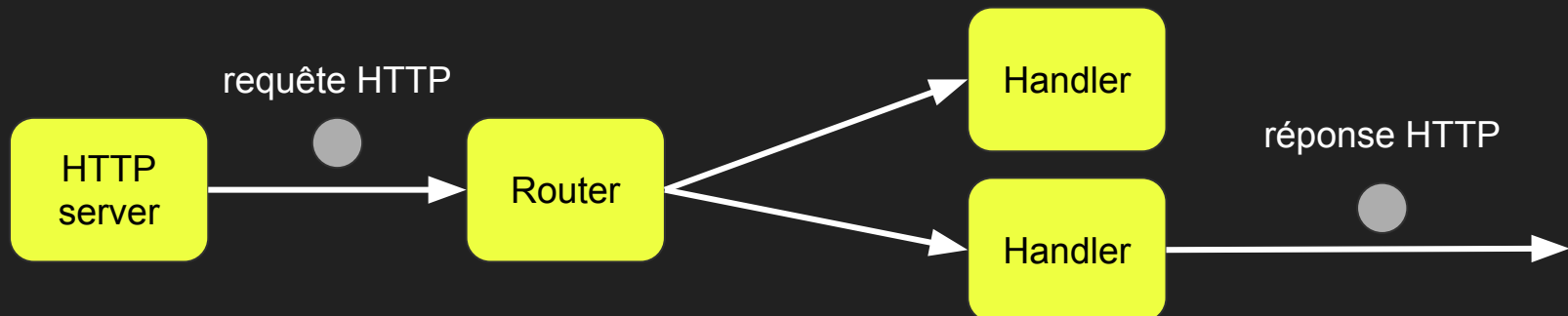
```
jdbc.getConnection(ar -> {  
    SqlConnection connection = ar.result();  
    connection.query("SELECT * FROM Beer", resp -> {  
        if (!resp.failed()) {  
            List<Beer> beverages = resp.result()  
                .getRows()  
                .stream()  
                .map(Beer::new)  
                .collect(Collectors.toList());  
        }  
        //...  
    });  
});
```

# Le Web

```
HttpServer server = vertx.createHttpServer();
Router router = Router.router(vertx);
Route route = router.route(HttpMethod.PUT, "myapi/orders")
    .consumes("application/json")
    .produces("application/json");

route.handler(routingContext -> {
    // This would be match for any PUT method to paths starting with
    // "myapi/orders" with a content-type of "application/json"
    // and an accept header matching "application/json"
});

server.requestHandler(router::accept).listen(8080);
```



# Le Web

```
Router restAPI = Router.router(vertex);
restAPI.get("/products/:productID").handler(rc -> {
    // TODO Handle the lookup of the product....
    rc.response().write(productJSON);
});

restAPI.put("/products/:productID").handler(rc -> {
    // TODO Add a new product...
    rc.response().end();
});

Router mainRouter = Router.router(vertex);
// Handle static resources
mainRouter.route("/static/*").handler(StaticHandler.create());
mainRouter.route(".*\\.templ").handler(myTemplateHandler);
mainRouter.mountSubRouter("/productsAPI", restAPI);

$curl http://localhost:8080/productsAPI/products/1234
```

# Le Web

```
router.route().path("/*")
    .consumes("application/json")
    .handler(routingContext -> {
        HttpServerResponse response = routingContext.response();
        response.putHeader("content-type", "application/json");
        routingContext.next();
    });

router.get("/beers/").handler(routingContext -> {
    JsonObject query = new JsonObject();
    mongoClient.find("beers", query, res -> {
        if (res.succeeded()) {
            JsonArray jar = new JsonArray();
            res.result().forEach(jar::add);
            routingContext.response().end(jar.encodePrettily());
        } else {
            res.cause().printStackTrace();
        }
    });
});
```

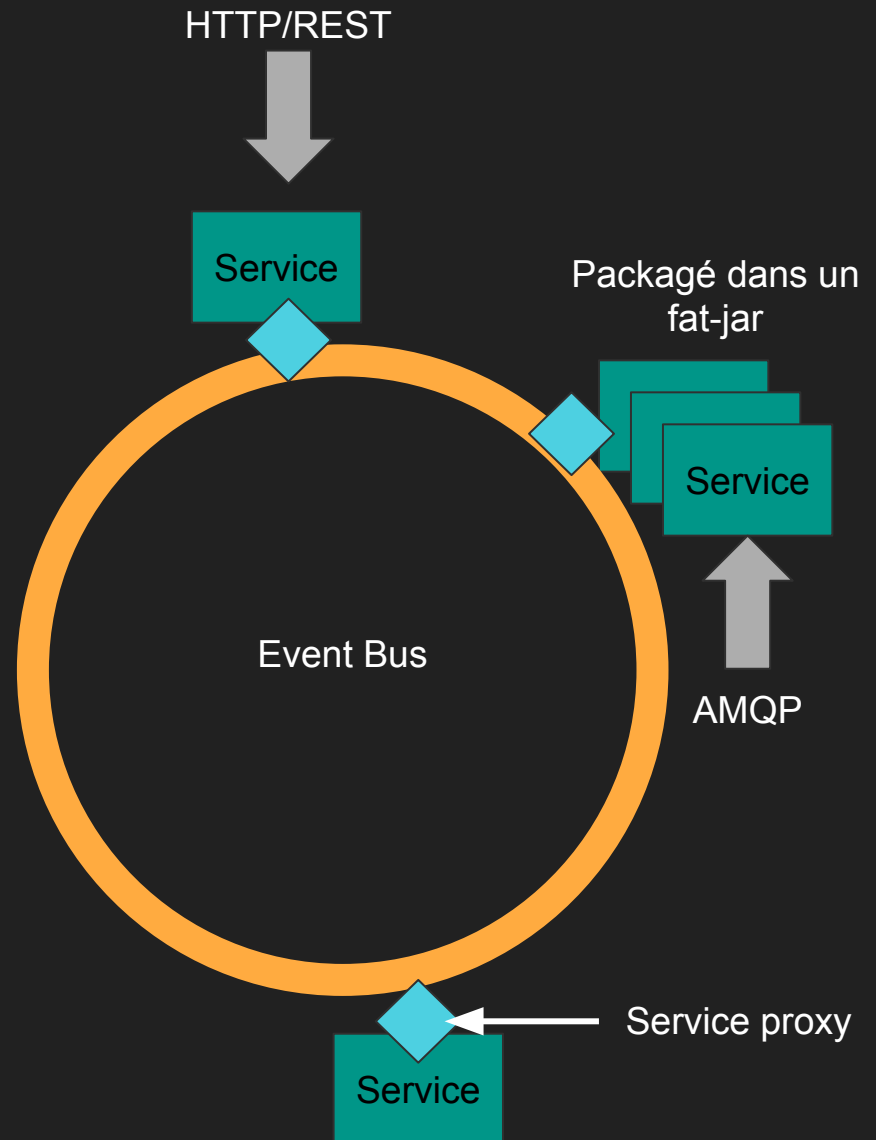
# Micro services

Les micro-services interagissent en utilisant l'eventBus ou en utilisant un autre protocole.

Des service-proxies sont générés pour l'eventBus.

On peut même consommer les services depuis NodeJS, le navigateur, iOS ou Android via l'eventBus.

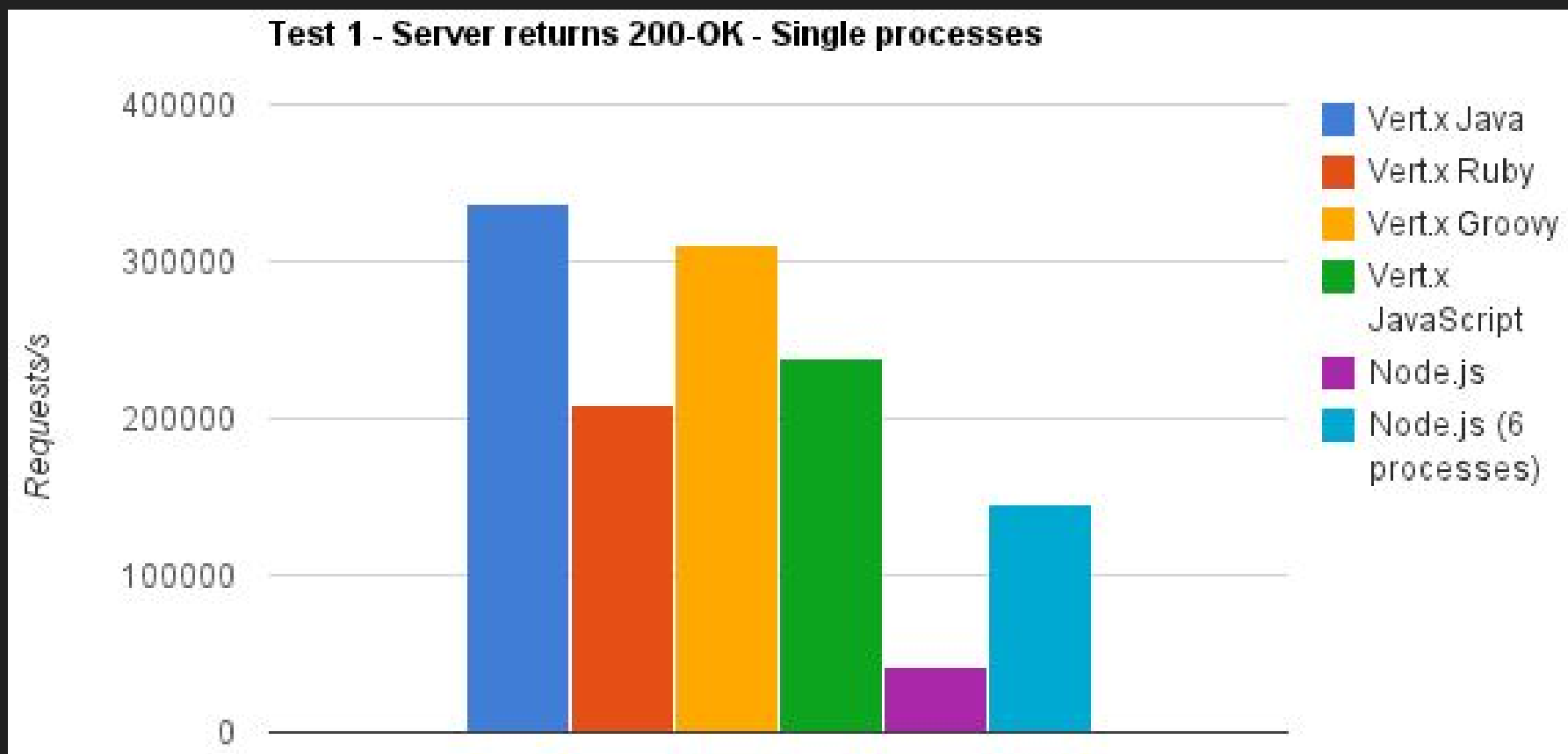
Chaque service peut se mettre à l'échelle en fonction de la charge et supporte le fail-over.





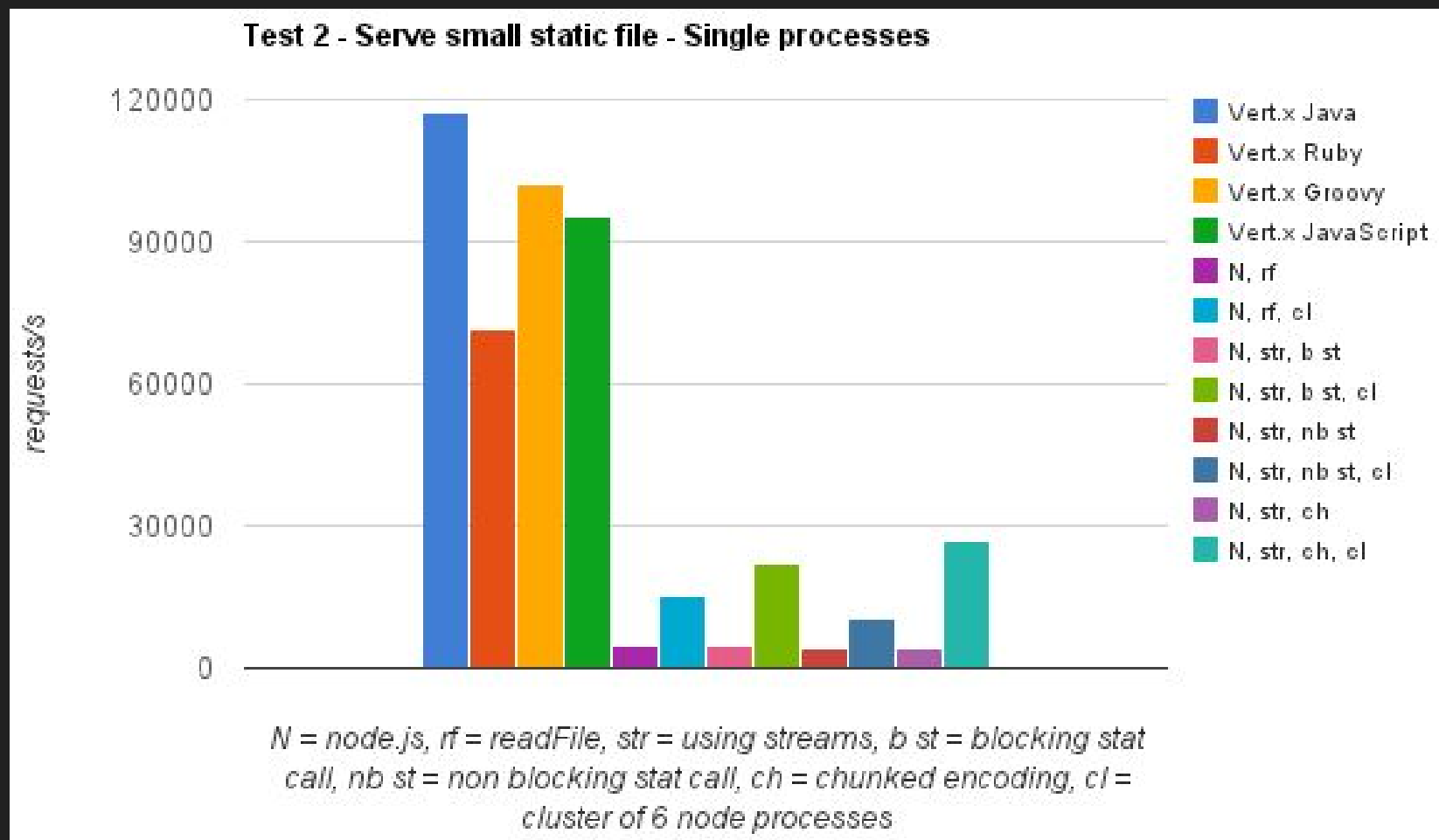
# Performances

Source : <http://www.cubrid.org>



# Performances

Source : <http://www.cubrid.org>

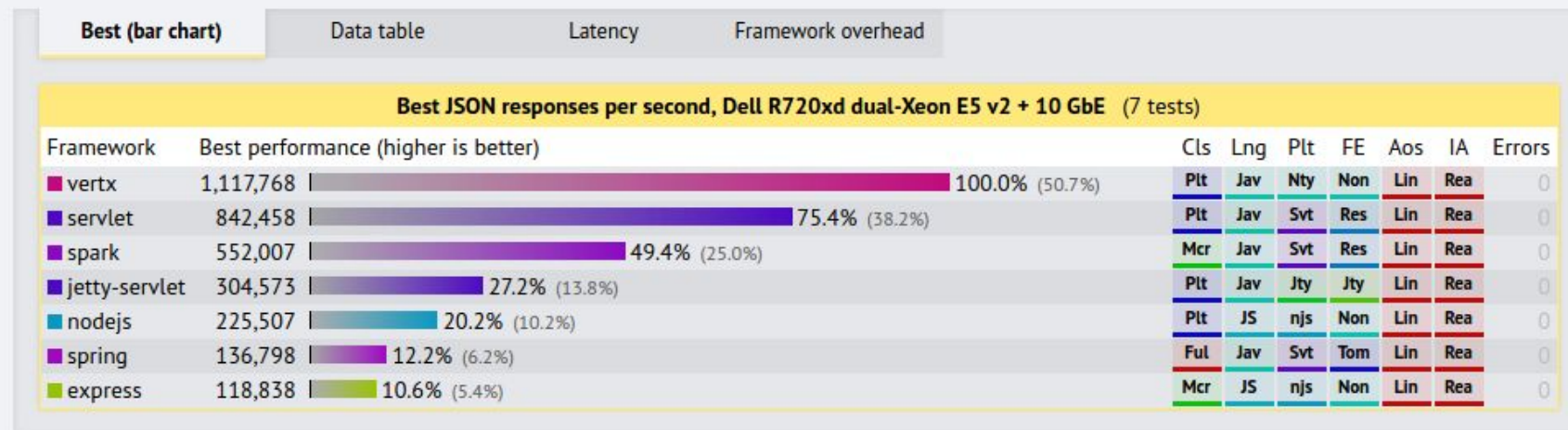


# Performances

Source : <https://www.techempower.com>

## JSON serialization

### Results

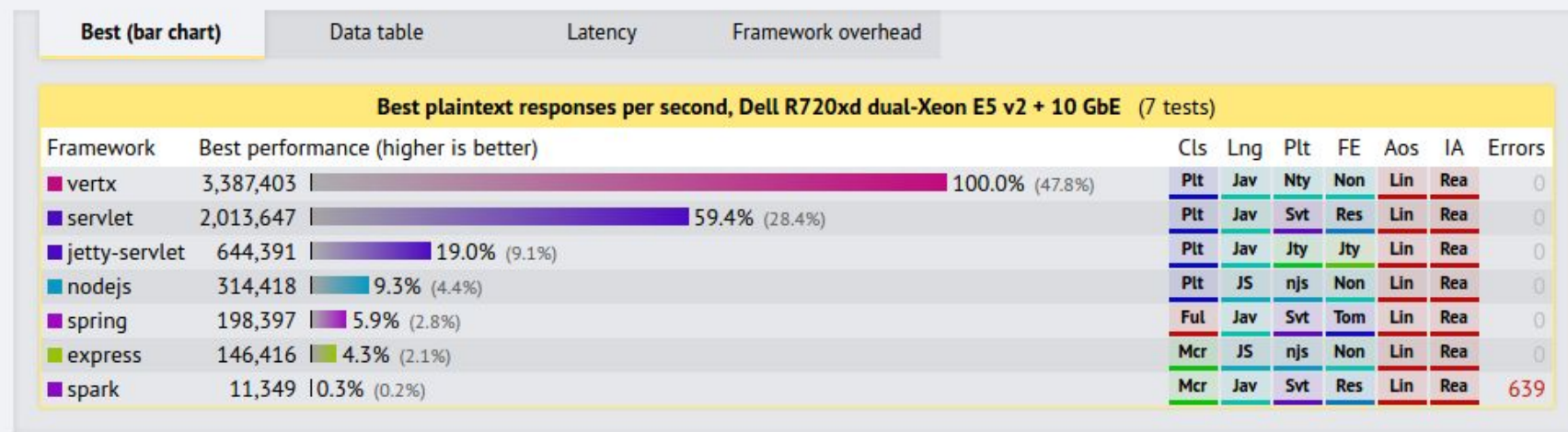


# Performances

Source : <https://www.techempower.com>

## Plaintext

### Results

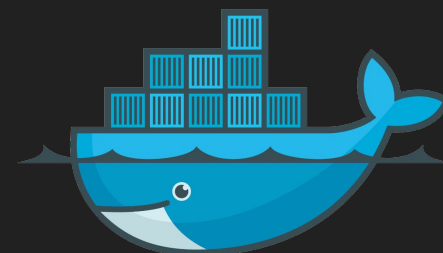


# Cloud ready

- Vert.x OpenShift Cartridge
  - <https://github.com/vert-x3/vertx-openshift-cartridge>
- Vert.x OpenShift Using DIY Cartridge
  - <https://github.com/vert-x3/vertx-openshift-diy-quickstart>
- Vert.x Docker Images
  - <https://github.com/vert-x3/vertx-stack/tree/master/stack-docker>



OPENSHIFT



docker

Production ready



CYANOGEN



hulu



*ticketmaster*



jClarity



# The end ...

« Si vous avez compris ce que je viens de vous dire, c'est que je me suis probablement mal exprimé »

A. Greenspan

<https://github.com/Giwi/vertx3-angular-beers>