Game of Verticles I: Introduction to Vert.x

Rafael Merino García

@imrafaelmerino

2021

Table of Contents

Goals

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?
    - Increasing number of cores?

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?
    - Increasing number of cores?
    - Increasing number of machines?

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?
    - Increasing number of cores?
    - Increasing number of machines?
- Understand better functional and reactive programming

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?
    - Increasing number of cores?
    - Increasing number of machines?
- Understand better functional and reactive programming
  - FP shines dealing with **effects**

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?
    - Increasing number of cores?
    - Increasing number of machines?
- Understand better functional and reactive programming
  - FP shines dealing with **effects**
  - Is your app really reactive?

Goals (I)

- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?
    - Increasing number of cores?
    - Increasing number of machines?
- Understand better functional and reactive programming
  - FP shines dealing with **effects**
  - Is your app really reactive?
    - What's your first integration test?

Goals (I)

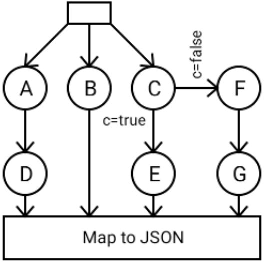
- How to use **Vertx** following the **Erlang** philosophy (**message passing**)
  - Why? **Low coupling** and
  - How do you get programs to run faster **without changing a single line?**
    - Increasing CPU frequency?
    - Increasing number of cores?
    - Increasing number of machines?
- Understand better functional and reactive programming
  - FP shines dealing with **effects**
  - Is your app really reactive?
    - What's your first integration test?
    - **Failures are just data**

Goals (II)

- Develop complex services in Vertx **and not die trying** (async programming is still hard nowadays)

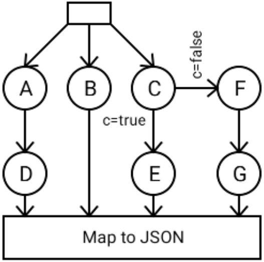
Goals (II)

- Develop complex services in Vertx **and not die trying** (async programming is still hard nowadays)
- A, B and C are HTTP requests or database calls executed **in parallel**



Goals (II)

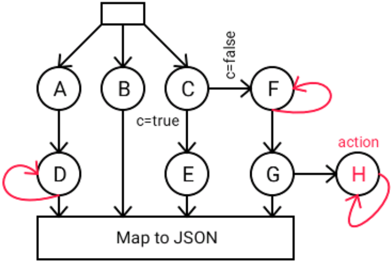
- Develop complex services in Vertx **and not die trying** (async programming is still hard nowadays)
- A, B and C are HTTP requests or database calls executed **in parallel**



- I didn't mention it, but I was thinking of **just one line of code**

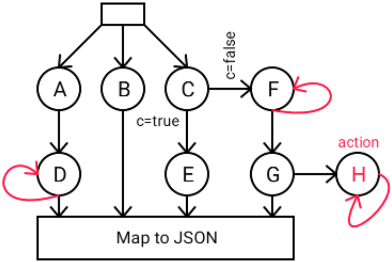
Goals (III)

- and if you retry some ops under certain errors (connection timeouts, network partitions, etc.) or want to add some action without waiting for the response, like sending an email (H)



Goals (III)

- and if you retry some ops under certain errors (connection timeouts, network partitions, etc.) or want to add some action without waiting for the response, like sending an email (H)



- I'm still thinking of just one line of code

History

Actor Model (1973)

Actor Model of Computation:
Scalable Robust Information Systems

Carl Hewitt

This article is dedicated to Alonzo Church and Dana Scott.

The Actor Model is a mathematical theory that treats "*Actors*" as the universal primitives of digital computation.

Hypothesis:[i] **All physically possible computation can be directly implemented using Actors.**

The model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent systems. The advent of massive concurrency through client-cloud computing and many-core computer architectures has galvanized interest in the Actor Model.

Message passing using types is the foundation of system communication:

- Messages are the unit of communication[1]

SmallTalk (1970-1980)

- **Alan Kay**, one of its creators, **coined the term Object- Oriented Programming** but...

SmallTalk (1970-1980)

- **Alan Kay**, one of its creators, **coined the term Object- Oriented Programming** but...
- “I’m sorry that I long ago coined the term “objects” for this topic because it gets many people to focus on the lesser idea. **The big idea is messaging**.”

SmallTalk (1970-1980)

- **Alan Kay**, one of its creators, **coined the term Object- Oriented Programming** but...
- “I’m sorry that I long ago coined the term “objects” for this topic because it gets many people to focus on the lesser idea. **The big idea is messaging**.”
- "I made up the term ‘object-oriented’, and I can tell you **I didn’t have C++ in mind**.”

SmallTalk (1970-1980)

- **Alan Kay**, one of its creators, **coined the term Object- Oriented Programming** but...
- "I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. **The big idea is messaging**."
- "I made up the term 'object-oriented', and I can tell you **I didn't have C++ in mind**."
- Download Pharo and be blown away!

Erlang (1986)

- Open Source in 1998

Erlang (1986)

- Open Source in 1998
- Functional programming language created **to build fault-tolerant and scalable distributed systems in Ericsson**

Erlang (1986)

- Open Source in 1998
- Functional programming language created **to build fault-tolerant and scalable distributed systems in Ericsson**
- Isolated processes that **interact sending messages** (10 servers handling 1 req instead of 1 server handling 10 req)

Erlang (1986)

- Open Source in 1998
- Functional programming language created **to build fault-tolerant and scalable distributed systems in Ericsson**
- Isolated processes that **interact sending messages** (10 servers handling 1 req instead of 1 server handling 10 req)
- Keep calm and **let it crash** (avoid defensive programming and don't make things worse)

Erlang (1986)

- Open Source in 1998
- Functional programming language created **to build fault-tolerant and scalable distributed systems in Ericsson**
- Isolated processes that **interact sending messages** (10 servers handling 1 req instead of 1 server handling 10 req)
- Keep calm and **let it crash** (avoid defensive programming and don't make things worse)
- WhatsApp and WeChat are implemented in Erlang!

Influential people

Vertx

Recomendations (my own)

- Understand what message passing means. **Adoption of a new paradigm is a complex and slow process that has to hurt**.

Recomendations (my own)

- Understand what message passing means. **Adoption of a new paradigm is a complex and slow process that has to hurt**.
- EOOP just doesn't fit (**don't use Spring or try to avoid it**)

Recomendations (my own)

- Understand what message passing means. **Adoption of a new paradigm is a complex and slow process that has to hurt**.
- EOOP just doesn't fit (**don't use Spring or try to avoid it**)
- Dependency injection == Coupling

Recomendations (my own)

- Understand what message passing means. **Adoption of a new paradigm is a complex and slow process that has to hurt**.
- EOOP just doesn't fit (**don't use Spring or try to avoid it**)
- Dependency injection == Coupling
- **Have a plan to handle complexity**.

Verticles

- **Everything** is a Verticle (minimum unit of computation)

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication
- Can create and kill other verticles

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication
- Can create and kill other verticles
- Verticles are strongly **isolated** (share no resources)

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication
- Can create and kill other verticles
- Verticles are strongly **isolated** (share no resources)
- Verticles creation and destruction is a lightweight operation

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication
- Can create and kill other verticles
- Verticles are strongly **isolated** (share no resources)
- Verticles creation and destruction is a lightweight operation
- Verticles have **unique** addresses

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication
- Can create and kill other verticles
- Verticles are strongly **isolated** (share no resources)
- Verticles creation and destruction is a lightweight operation
- Verticles have **unique** addresses
- If you know the address of a verticle, you can send it a message (location transparency)

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication
- Can create and kill other verticles
- Verticles are strongly **isolated** (share no resources)
- Verticles creation and destruction is a lightweight operation
- Verticles have **unique** addresses
- If you know the address of a verticle, you can send it a message (location transparency)
- Message passing is **the only way** for verticles to interact

Verticles

- **Everything** is a Verticle (minimum unit of computation)
- computation | storage | communication
- Can create and kill other verticles
- Verticles are strongly **isolated** (share no resources)
- Verticles creation and destruction is a lightweight operation
- Verticles have **unique** addresses
- If you know the address of a verticle, you can send it a message (location transparency)
- Message passing is **the only way** for verticles to interact
- **The more verticles, the better**

Message passing

- Message passing is assumed to be **atomic** which means that a message is either delivered in its entirety or not at all

Message passing

- Message passing is assumed to be **atomic** which means that a message is either delivered in its entirety or not at all
- Message passing between a pair of verticles is assumed to be ordered: **the messages will be received in the same order they were sent**

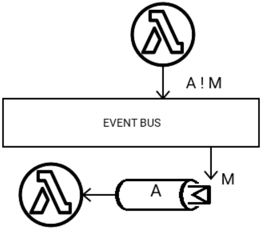
Message passing

- Message passing is assumed to be **atomic** which means that a message is either delivered in its entirety or not at all
- Message passing between a pair of verticles is assumed to be ordered: **the messages will be received in the same order they were sent**
- Messages **should not contain references to data structures contained within verticles**—they should only contain constants and/or address

Message passing

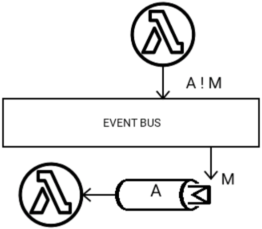
- Message passing is assumed to be **atomic** which means that a message is either delivered in its entirety or not at all
- Message passing between a pair of verticles is assumed to be ordered: **the messages will be received in the same order they were sent**
- Messages **should not contain references to data structures contained within verticles**—they should only contain constants and/or address
- **send and pray** semantics. We send the message and pray that it arrives

Vertx model (I)



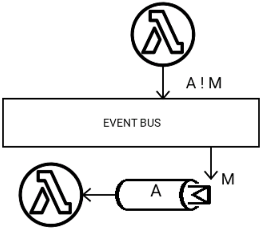
- The event bus is the nervous system of Vertx

Vertx model (I)



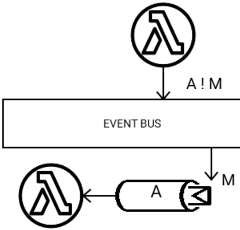
- The event bus is the nervous system of Vertx
- Distributed event bus: verticles **from different machines** can communicate with each other by sending messages. Scale-out

Vertx model (I)



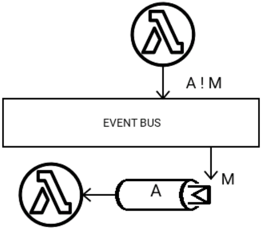
- The event bus is the nervous system of Vertx
- Distributed event bus: verticles **from different machines** can communicate with each other by sending messages. Scale-out
- It provides an API to:

Vertx model (I)



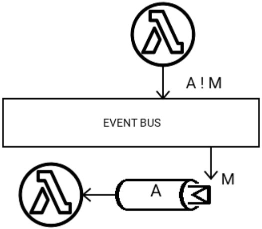
- The event bus is the nervous system of Vertx
- Distributed event bus: verticles **from different machines** can communicate with each other by sending messages. Scale-out
- It provides an API to:
  - deploy verticles listening on addresses

Vertx model (I)



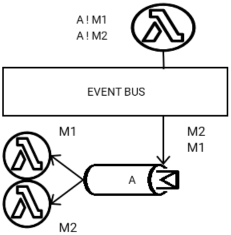
- The event bus is the nervous system of Vertx
- Distributed event bus: verticles **from different machines** can communicate with each other by sending messages. Scale-out
- It provides an API to:
  - deploy verticles listening on addresses
  - **send and publish** messages to addresses: point-to-point and pub/sub

Vertx model (I)



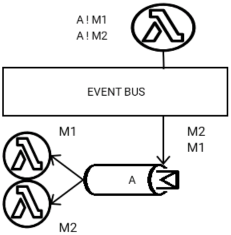
- The event bus is the nervous system of Vertx
- Distributed event bus: verticles **from different machines** can communicate with each other by sending messages. Scale-out
- It provides an API to:
  - deploy verticles listening on addresses
  - **send and publish** messages to addresses: point-to-point and pub/sub
- **Verticles process ONE message at a time**. Synchronization is implemented this way

Vertx model (II)



- Scale-up: deploying multiple instances of a verticle listening on the same address

Vertx model (II)

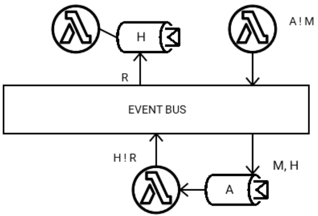


- Scale-up: deploying multiple instances of a verticle listening on the same address
- The instance that receives the message is chosen using a non-strict round-robin algorithm

Vertx model (III)

Imagine that a verticle **sends a message** to another verticle and has to **process the response**

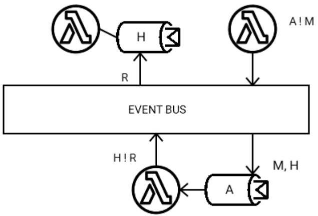
- Programmatically, it's just a handler



Vertx model (III)

Imagine that a verticle **sends a message** to another verticle and has to **process the response**

- Programmatically, it's just a handler
- but in practice, it's just another verticle listening on a random address



Verticles life cycle

- Verticles that are created during app bootstrap and never dye

Verticles life cycle

- Verticles that are created during app bootstrap and never dye
- Verticles that are associated with the life cycle of an entity (user login and logout)

Verticles life cycle

- Verticles that are created during app bootstrap and never dye
- Verticles that are associated with the life cycle of an entity (user login and logout)
- Verticles that are created to do computation and die after that

Messages

- String, null, bytes, Numbers, Boolean and Json (from Jackson)

Messages

- String, null, bytes, Numbers, Boolean and Json (from Jackson)
- All supported messages have a MessageCodec associated

Messages

- String, null, bytes, Numbers, Boolean and Json (from Jackson)
- All supported messages have a MessageCodec associated
- You can send your own messages if you create and register their codecs

Messages

- String, null, bytes, Numbers, Boolean and Json (from Jackson)
- All supported messages have a MessageCodec associated
- You can send your own messages if you create and register their codecs
- The Json from Jackson is mutable. ¿Why is this a problem?

Messages

- String, null, bytes, Numbers, Boolean and Json (from Jackson)
- All supported messages have a MessageCodec associated
- You can send your own messages if you create and register their codecs
- The Json from Jackson is mutable. ¿Why is this a problem?
  - Verticles have to be isolated

```java
@Override
public JsonObject transform(JsonObject jsonObject) {
  return jsonObject.copy();
}
```

Messages

- String, null, bytes, Numbers, Boolean and Json (from Jackson)
- All supported messages have a MessageCodec associated
- You can send your own messages if you create and register their codecs
- The Json from Jackson is mutable. ¿Why is this a problem?
    - Verticles have to be isolated

```java
@Override
public JsonObject transform(JsonObject jsonObject) {
  return jsonObject.copy();
}
```

- imrafaelmerino/json-values is a better alternative: it's persistent and provides a better api

```java
@Override
public JsObj transform(final JsObj obj) {
  return obj;
}
```

Threading model

- Two types of threads: **event loops and workers**

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.
- computationally intensive or blocking tasks $\Rightarrow$ worker

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.
- computationally intensive or blocking tasks $\Rightarrow$ worker
- never block an event loop

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.
- computationally intensive or blocking tasks $\Rightarrow$ worker
- never block an event loop
- Size of pools by default

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.
- computationally intensive or blocking tasks $\Rightarrow$ worker
- never block an event loop
- Size of pools by default
    - event loops: 2 x processors

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.
- computationally intensive or blocking tasks $\Rightarrow$ worker
- never block an event loop
- Size of pools by default
  - event loops: 2 x processors
  - workers: 20

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.
- computationally intensive or blocking tasks $\Rightarrow$ worker
- never block an event loop
- Size of pools by default
  - event loops: 2 x processors
  - workers: 20
  - What's the right size?

Threading model

- Two types of threads: **event loops and workers**
- When deploying a verticle, you have to specify if a worker or an event loop will execute it.
- computationally intensive or blocking tasks $\Rightarrow$ worker
- never block an event loop
- Size of pools by default
  - event loops: 2 x processors
  - workers: 20
  - What's the right size?
- What about green threads and project Loom?

Alternatives

- Erlang

Alternatives

- Erlang
- Scala Akka

Alternatives

- Erlang
- Scala Akka
- Pony (compiled programming language)

vertx-effect: where Vertx meet FP

Vertx future

- Vertx 4 use futures to represent asynchronous results, but

Vertx future

- Vertx 4 use futures to represent asynchronous results, but
- Vertx future API **is not rich enough** to develop complex verticles:

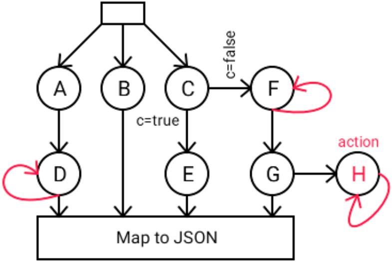
Vertx future

- Vertx 4 use futures to represent asynchronous results, but
- Vertx future API **is not rich enough** to develop complex verticles:
- **Only three methods to coordinate**: join, all and any

Vertx future

- Vertx 4 use futures to represent asynchronous results, but
- Vertx future API **is not rich enough** to develop complex verticles:
- **Only three methods to coordinate**: join, all and any
- Since **it's not lazy**, key reactive operations like *retry*, *recoverWith* and *fallbackTo* are missing

Impossible to address the following flow reasonably:



Val and λ to the rescue

```
import java.util.function.Supplier;
import java.util.function.Function;
import io.vertx.core.Future;

public interface Val<O> extends Supplier<Future<O>> {...}

public interface λ<I,O> extends Function<I, Val<O>> {...}
```

- Val is **lazy**. It **describes** an asyncronous effect

Val and λ to the rescue

```
import java.util.function.Supplier;
import java.util.function.Function;
import io.vertx.core.Future;

public interface Val<O> extends Supplier<Future<O>> {...}

public interface λ<I,O> extends Function<I, Val<O>> {...}
```

- Val is **lazy**. It **describes** an asyncronous effect
- The types **I** and **O** represent messages sent to the Event Bus

Val and λ to the rescue

```
import java.util.function.Supplier;
import java.util.function.Function;
import io.vertx.core.Future;

public interface Val<O> extends Supplier<Future<O>> {...}

public interface λ<I,O> extends Function<I, Val<O>> {...}
```

- Val is **lazy**. It **describes** an asyncronous effect
- The types **I** and **O** represent messages sent to the Event Bus
- If they are not supported by Vertx

Val and λ to the rescue

```
import java.util.function.Supplier;
import java.util.function.Function;
import io.vertx.core.Future;

public interface Val<O> extends Supplier<Future<O>> {...}

public interface λ<I,O> extends Function<I, Val<O>> {...}
```

- Val is **lazy**. It **describes** an asyncronous effect
- The types **I** and **O** represent messages sent to the Event Bus
- If they are not supported by Vertx
    - Implement and register a **MessageCodec** for them

Practice makes perfect

Hands-on

What we will do in the next session:

- Playing around with values

Hands-on

What we will do in the next session:

- Playing around with values
- Expressions

Hands-on

What we will do in the next session:

- Playing around with values
- Expressions
- Lambdas

Hands-on

What we will do in the next session:

- Playing around with values
- Expressions
- Lambdas
- Deploying verticles and modules

Hands-on

What we will do in the next session:

- Playing around with values
- Expressions
- Lambdas
- Deploying verticles and modules
- Reactive http client

Hands-on

What we will do in the next session:

- Playing around with values
- Expressions
- Lambdas
- Deploying verticles and modules
- Reactive http client
- Wrapping existing libraries with lambdas: vertx-mongodb-effect

Hands-on

What we will do in the next session:

- Playing around with values
- Expressions
- Lambdas
- Deploying verticles and modules
- Reactive http client
- Wrapping existing libraries with lambdas: vertx-mongodb-effect
- Implementing complex flows

References I

- Carl Hewitt: Actor Model of Computation
  https://arxiv.org/vc/arxiv/papers/1008/1008.1459v8.pdf
- Alan C. Kay: The early history of Smalltalk.
  http://www.metaobject.com/papers/Smallhistory.pdf
- Alan C. Kay: Email explaining that the big idea is "messages" and not objects. http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html
- Dr. Alan Kay on the Meaning of "Object-Oriented Programming".
  http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

References II

- Joe Armstrong explaining that they weren't interested in, or worried about, the actor model while creating Erlang.
  http://erlang.org/pipermail/erlang-questions/2014-June/079891.html
- Joe Armstrong. Programming Erlang: Software for a Concurrent World