REACTIVE DISTRIBUTED SYSTEMS WITH VERT.X

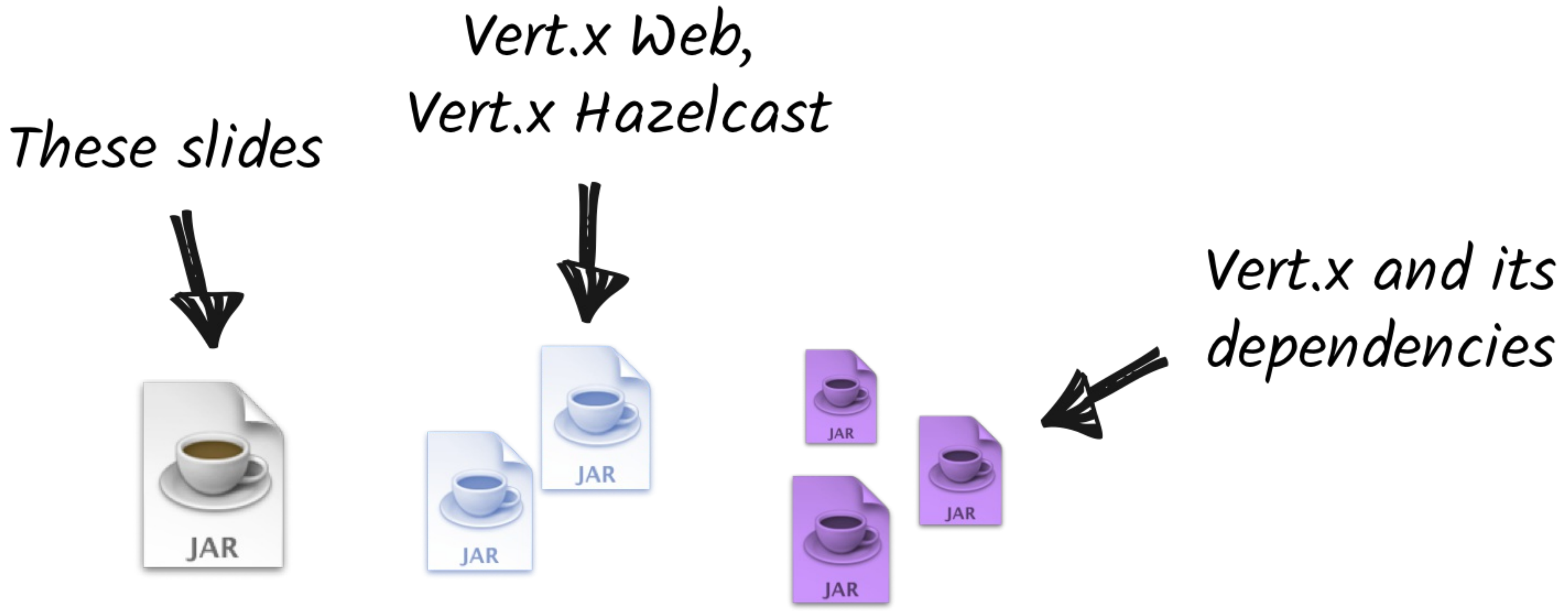
CLEMENT ESCOFFIER

RED HAT

VERT.X IS A TOOLKIT TO BUILD DISTRIBUTED AND REACTIVE SYSTEMS ON TOP OF THE JVM USING AN ASYNCHRONOUS NON-BLOCKING DEVELOPMENT MODEL.

TOOLKIT

- Vert.x is a plain boring **jar**
- Vert.x components are plain boring jars
- Your application depends on this set of jars (classpath, fat-jar, ...)



DISTRIBUTED

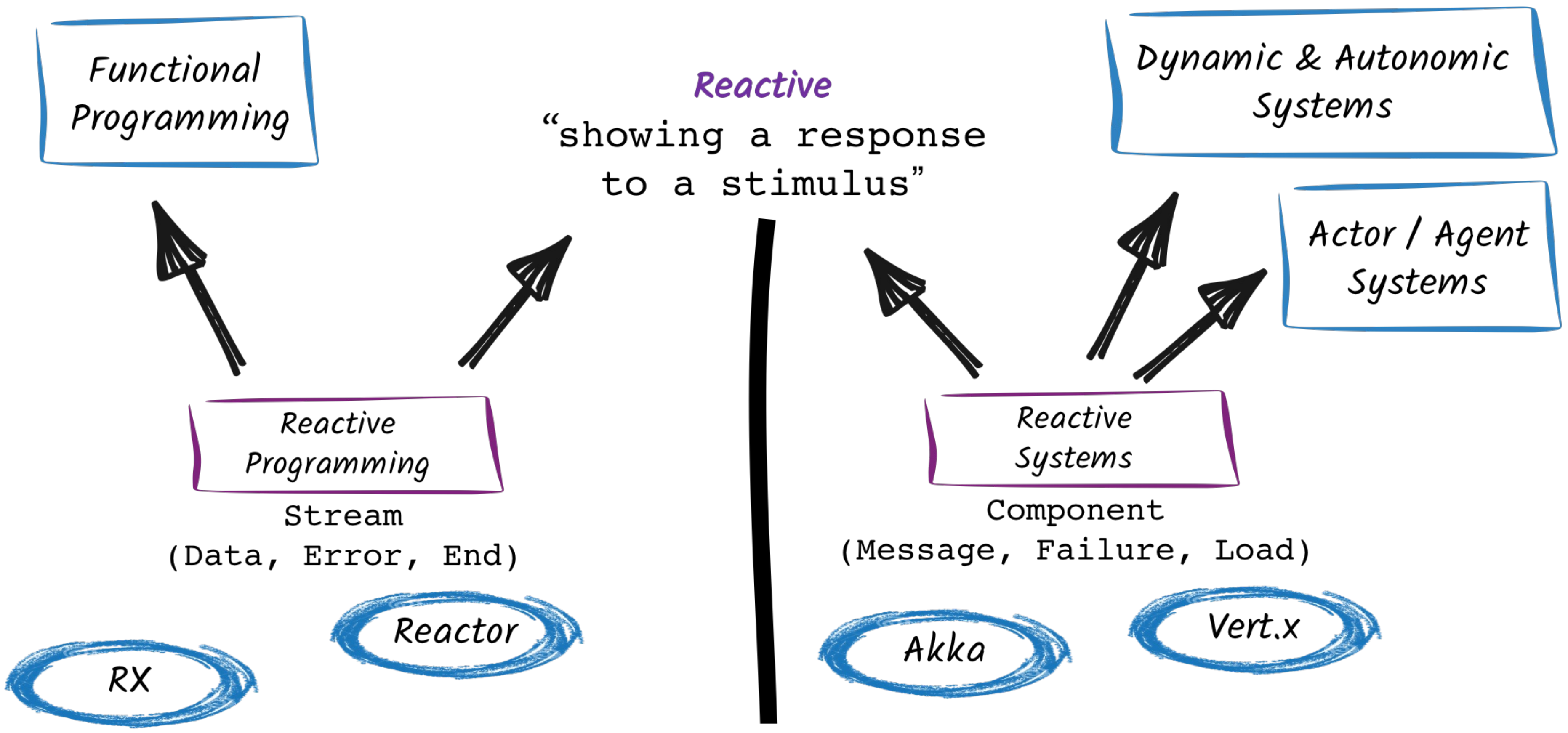
“You know you have a distributed system when the crash of a computer you’ve never heards of stops you from getting any work done.”
(Leslie Lamport)

REACTIVE SYSTEMS

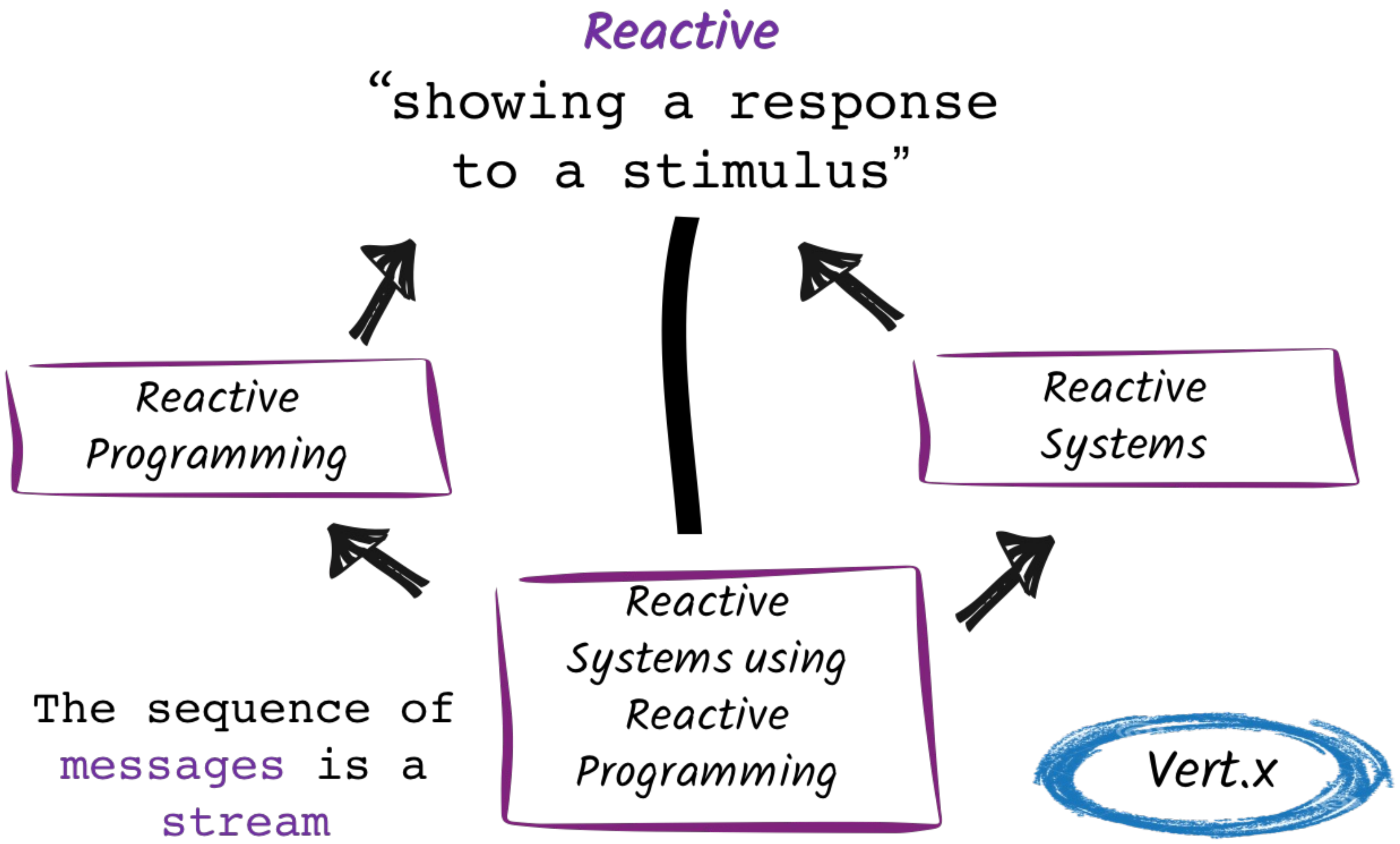
- **Responsive** - they respond in an acceptable time
- **Elastic** - they scale up and down
- **Resilient** - they are designed to handle failures gracefully
- **Asynchronous** - they interact using async messages

http://www.reactivemanifesto.org/

REACTIVE SYSTEMS != REACTIVE PROGRAMMING



REACTIVE SYSTEMS + REACTIVE PROGRAMMING


Reactive
“showing a response to a stimulus”
Reactive Programming
Reactive Systems
Reactive Systems using Reactive Programming
The sequence of messages is a stream
Vert.x

POLYGLOT

Vert.x applications can be developed using

- Java
- Groovy
- Ruby (JRuby)
- JavaScript (Nashorn)
- Ceylon
- Scala
- Kotlin

VERT.X

A toolkit to build distributed systems

VERT.X

Build **distributed** systems:

- Do not hide the **complexity**
- **Failure** as first-class citizen
- Provide the building blocks, not an all-in-one solution

WHAT DOES VERT.X PROVIDE ?

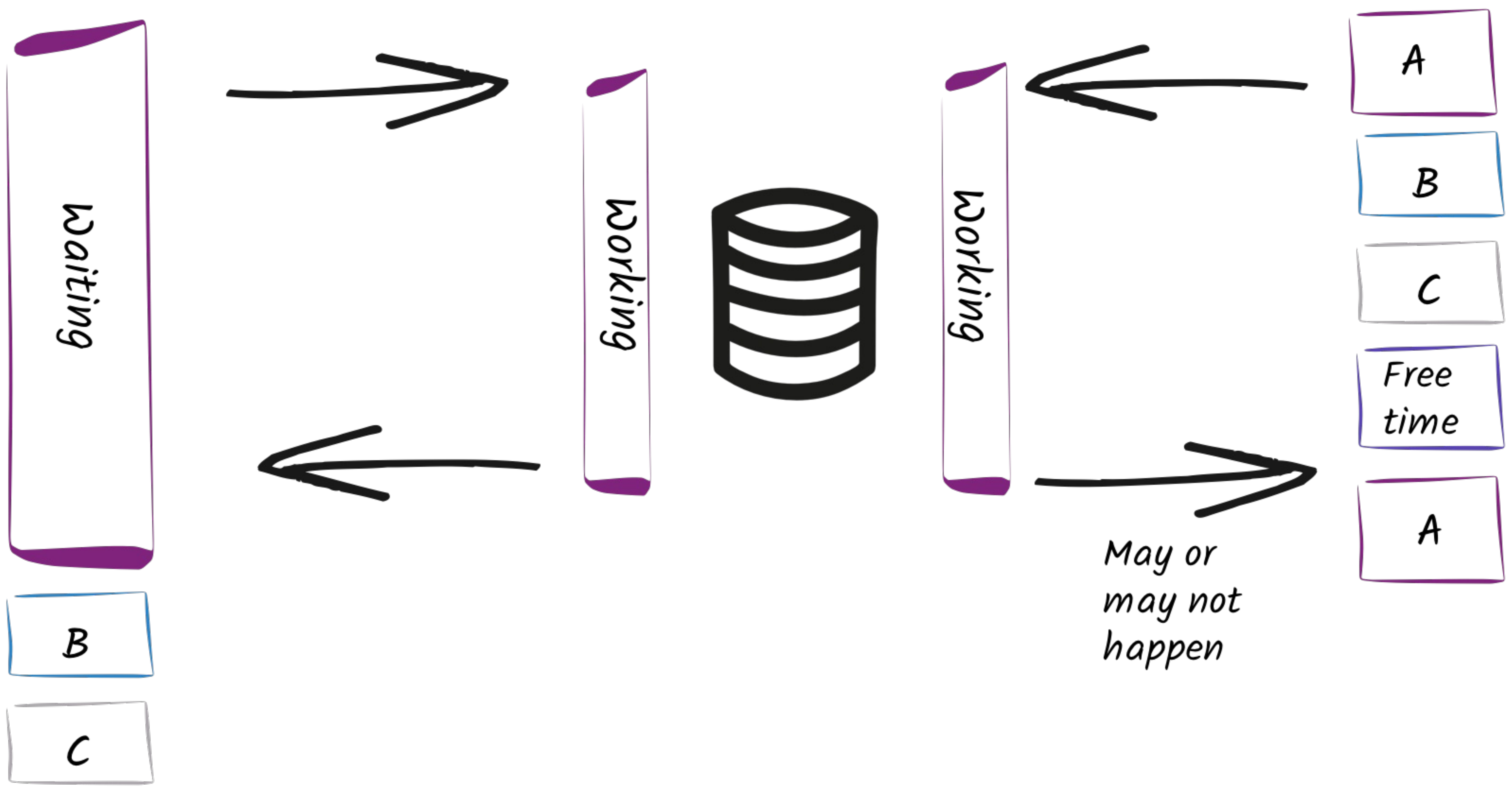
- TCP, UDP, HTTP 1 & 2 servers and clients
- (non-blocking) DNS client
- Clustering
- Event bus (messaging)
- Distributed data structures
- (built-in) Load-balancing
- (built-in) Fail-over
- Pluggable service discovery, circuit-breaker
- Metrics, Shell

REACTIVE

Build **reactive distributed** systems:

- **Responsive** - fast, is able to handle a large number of events / connections
- **Elastic** - scale up and down by just starting and stopping nodes, round-robin
- **Resilient** - failure as first-class citizen, fail-over
- **Asynchronous message-passing** - asynchronous and non-blocking development model

ASYNCHRONOUS & NON-BLOCKING



ASYNCHRONOUS & NON-BLOCKING

```
// Synchronous development model
X x = doSomething(a, b);

// Asynchronous development model - callback variant
doSomething(a, b, // Params
    ar -> {       // Last param is a Handler<AsyncResult<X>>
       // Result handler
    });

// Asynchronous development model - future variant
Future<X> future = doSomething(a, b);
future.setHandler(
    ar -> {  /* Completion handler */ });
```

REQUEST - REPLY INTERACTIONS

HTTP, TCP, RPC...

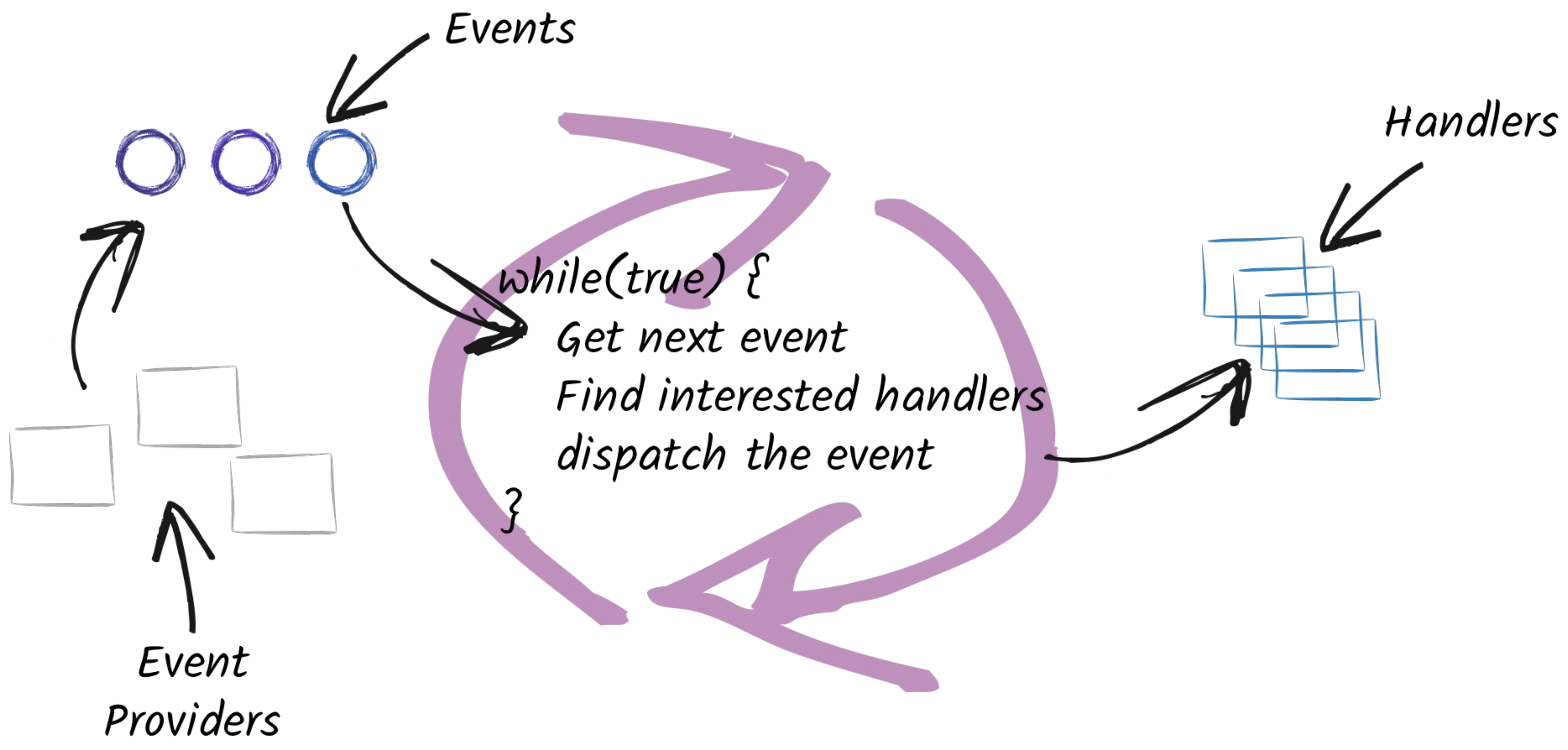
VERT.X HELLO WORLD

```java
Vertx vertx = Vertx.vertx();
vertx.createHttpServer()
  .requestHandler(request -> {
    // Handler receiving requests
    request.response().end("World !");
  })
  .listen(8080, ar -> {
    // Handler receiving start sequence completion (AsyncResult)
    if (ar.succeeded()) {
      System.out.println("Server started on port "
        + ar.result().actualPort());
    } else {
      ar.cause().printStackTrace();
    }
  });
```

VERT.X HELLO WORLD

Invoke

EVENT LOOPS

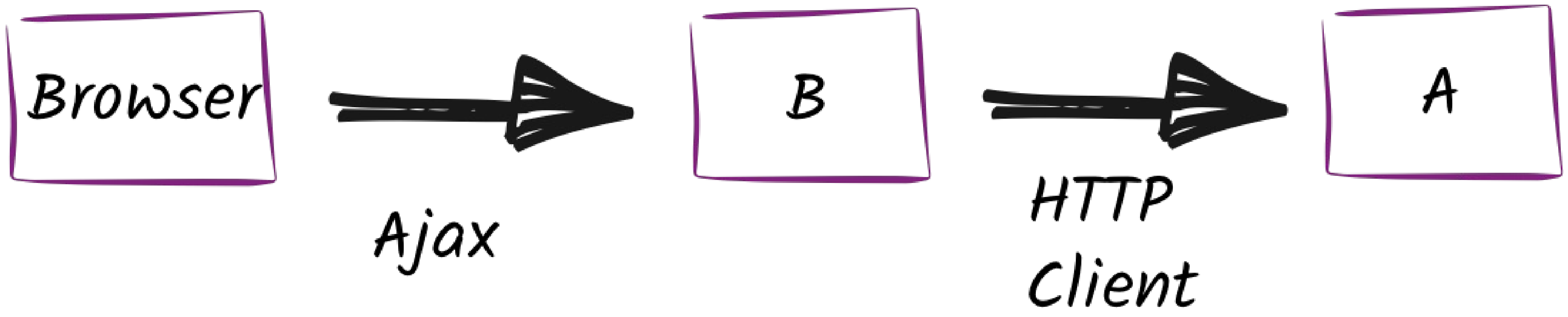


```
while(true) {
    Get next event
    Find interested handlers
    dispatch the event
}
```

VERT.X ASYNC HTTP CLIENT

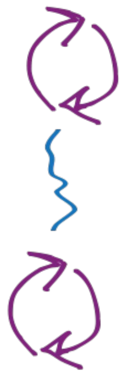
```
HttpClient client = vertx.createHttpClient(
        new HttpClientOptions()
                    .setDefaultHost("localhost")
                    .setDefaultPort(8081));

client.getNow("/", response -> {
  // Handler receiving the response

  // Get the content
  response.bodyHandler(buffer -> {
    // Handler to read the content
  });
});
```

CHAINED HTTP REQUESTS



Invoke

INTERACTING WITH BLOCKING SYSTEMS



```
vertx.executeBlocking(
  future -> {
     // Executed using a worker thread
  },
  asyncResult -> {
    // Executed in the event loop thread
    if (asyncResult.failed()) {
     // ...
    } else {
     // ...
    }
}
);
```

MESSAGING

The eventbus - the spine of Vert.x applications...

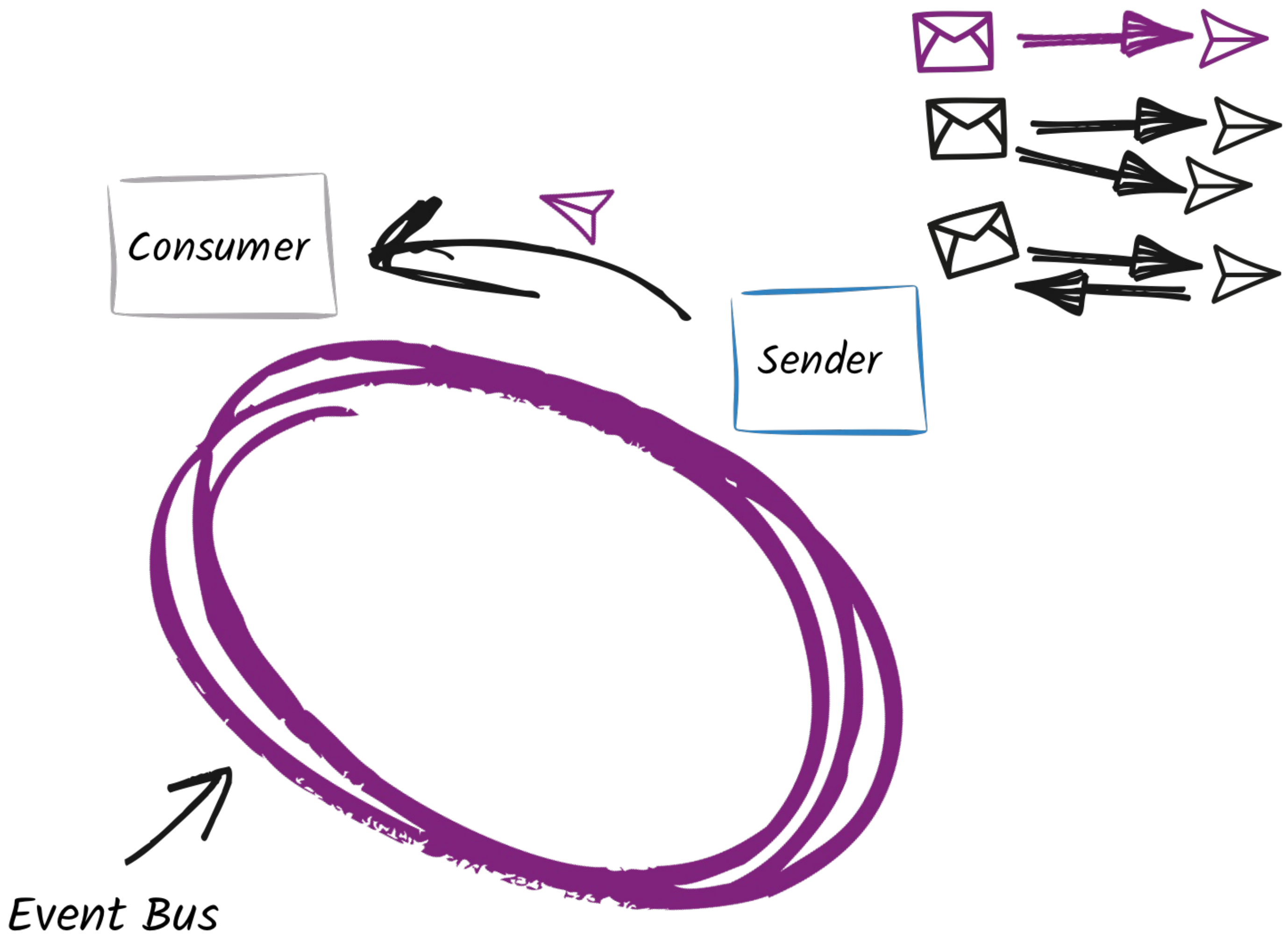
THE EVENT BUS

The event bus is the **nervous system** of vert.x:

- Allows different components to communicate regardless
  - the implementation language and their location
  - whether they run on vert.x or not (using bridges)

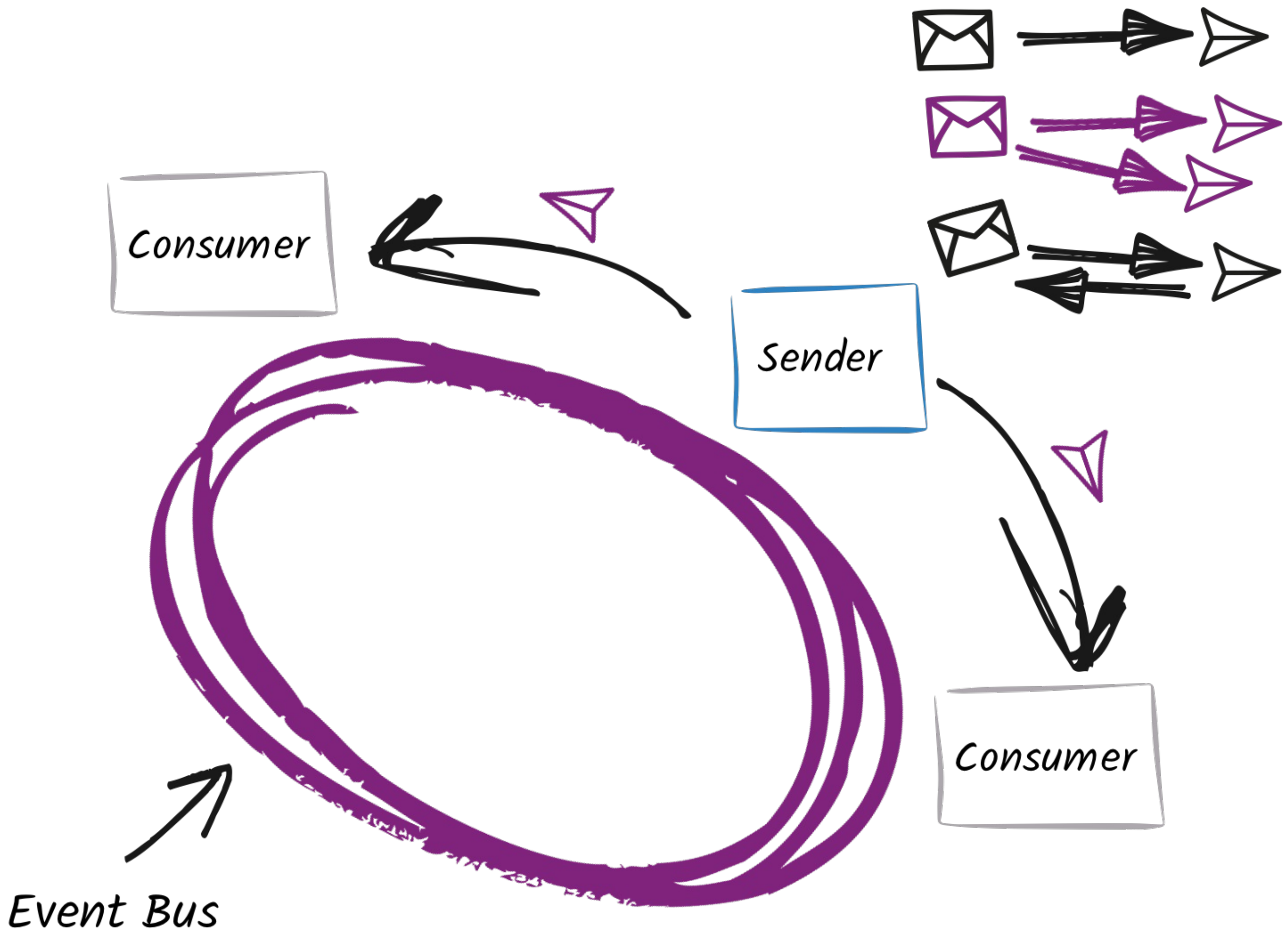
- **Address**: Messages are sent to an address
- **Handler**: Messages are received by `Handler`s.

POINT TO POINT



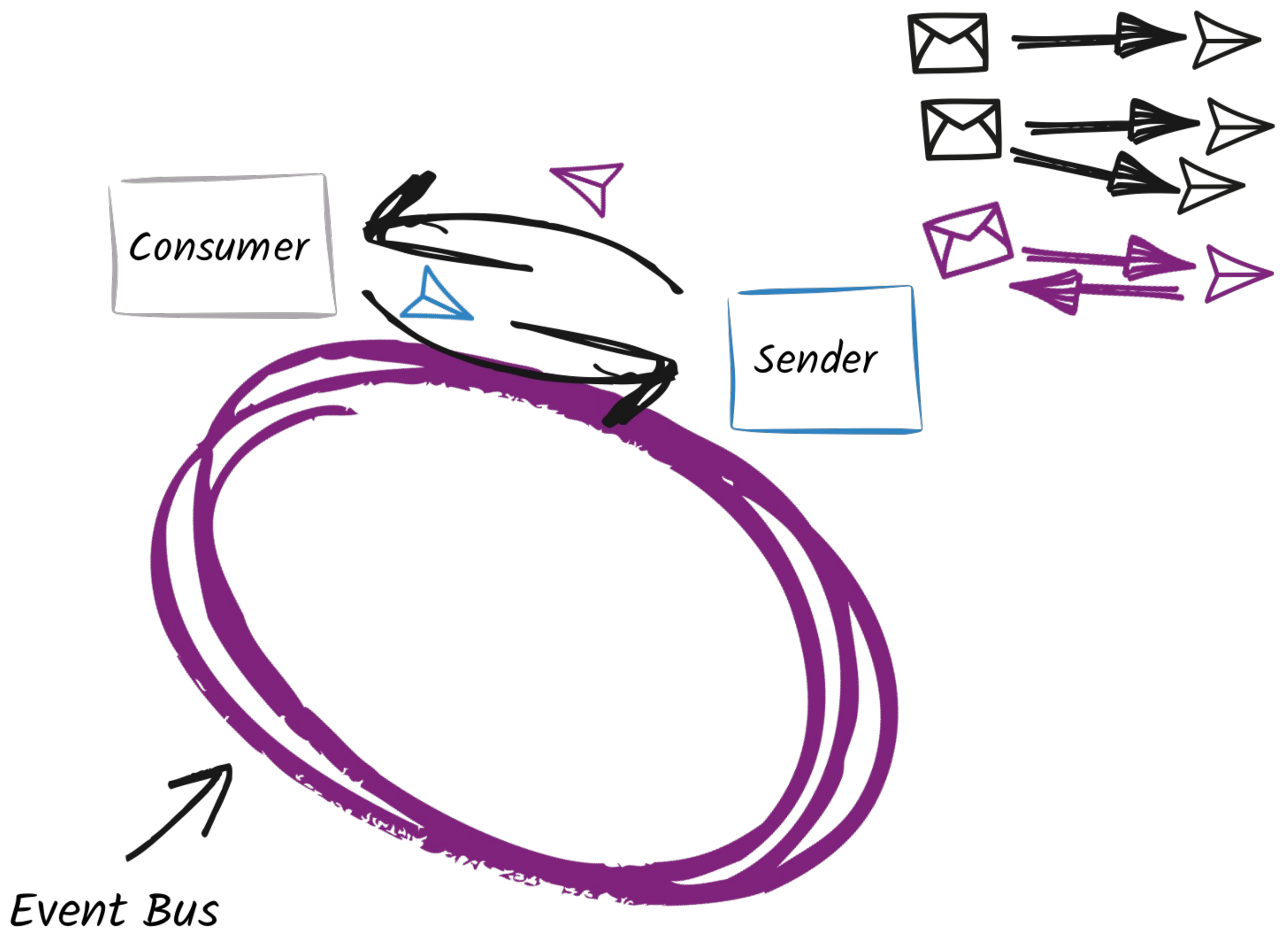
```
vertx.eventBus().send("address", "message");
vertx.eventBus().consumer("address", message -> {});
```

PUBLISH / SUBSCRIBE



```
vertx.eventBus().publish("address", "message");
vertx.eventBus().consumer("address", message -> {});
```

REQUEST / RESPONSE



```
vertx.eventBus().send("address", "message", reply -> {});
vertx.eventBus().consumer("address",
    message -> { message.reply("response"); });
```

FROM LOCAL TO CLUSTERED

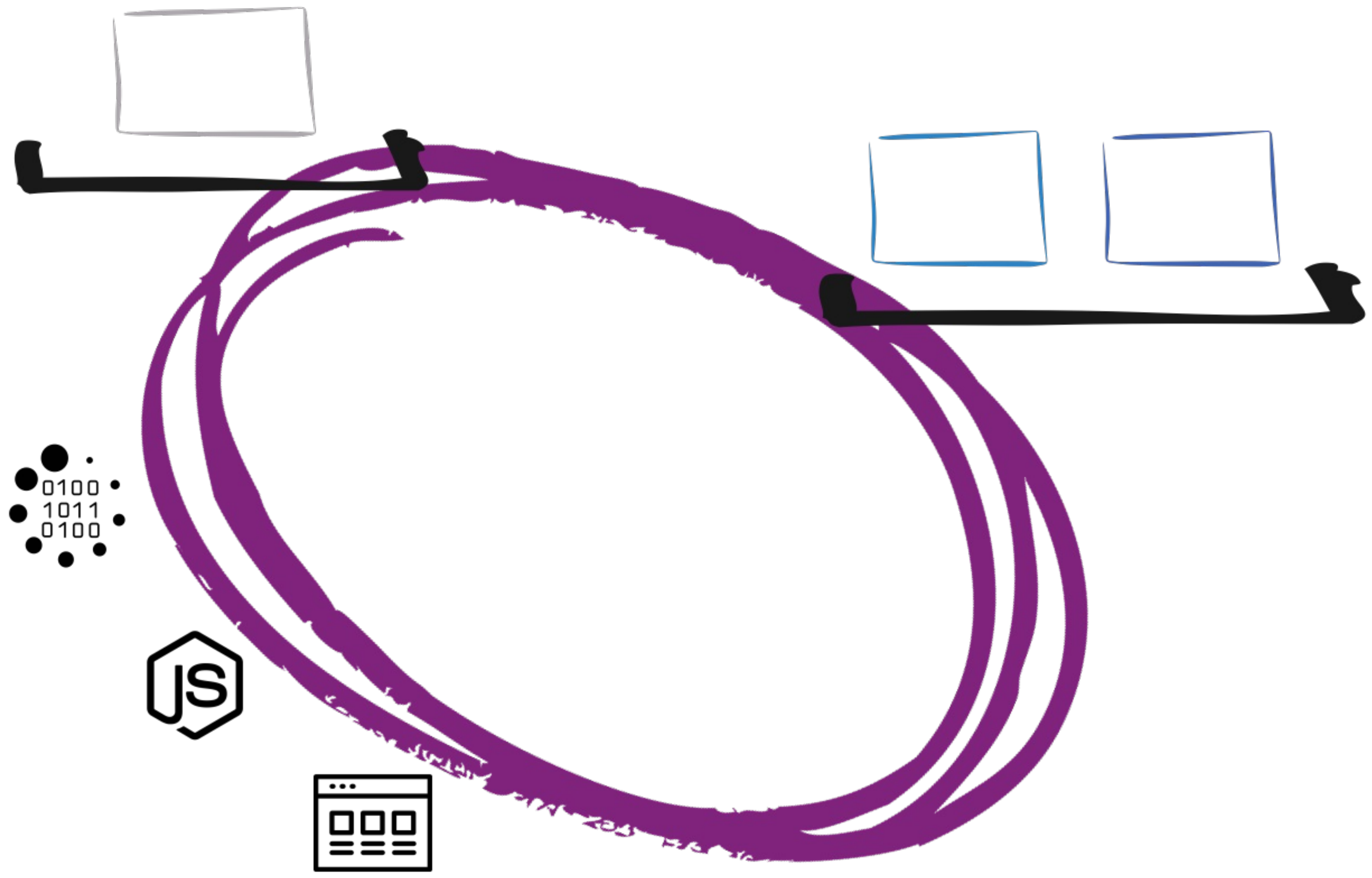
Vert.x instances form a **cluster**

```
Vertx.clusteredVertx(new VertxOptions(), result -> {
  if (result.failed()) {
    System.err.println("Cannot create a clustered vert.x : "
                        + result.cause());
  } else {
    Vertx vertx = result.result();
    // ...
  }
});
```

The event bus is distributed on all the cluster members

DISTRIBUTED EVENT BUS

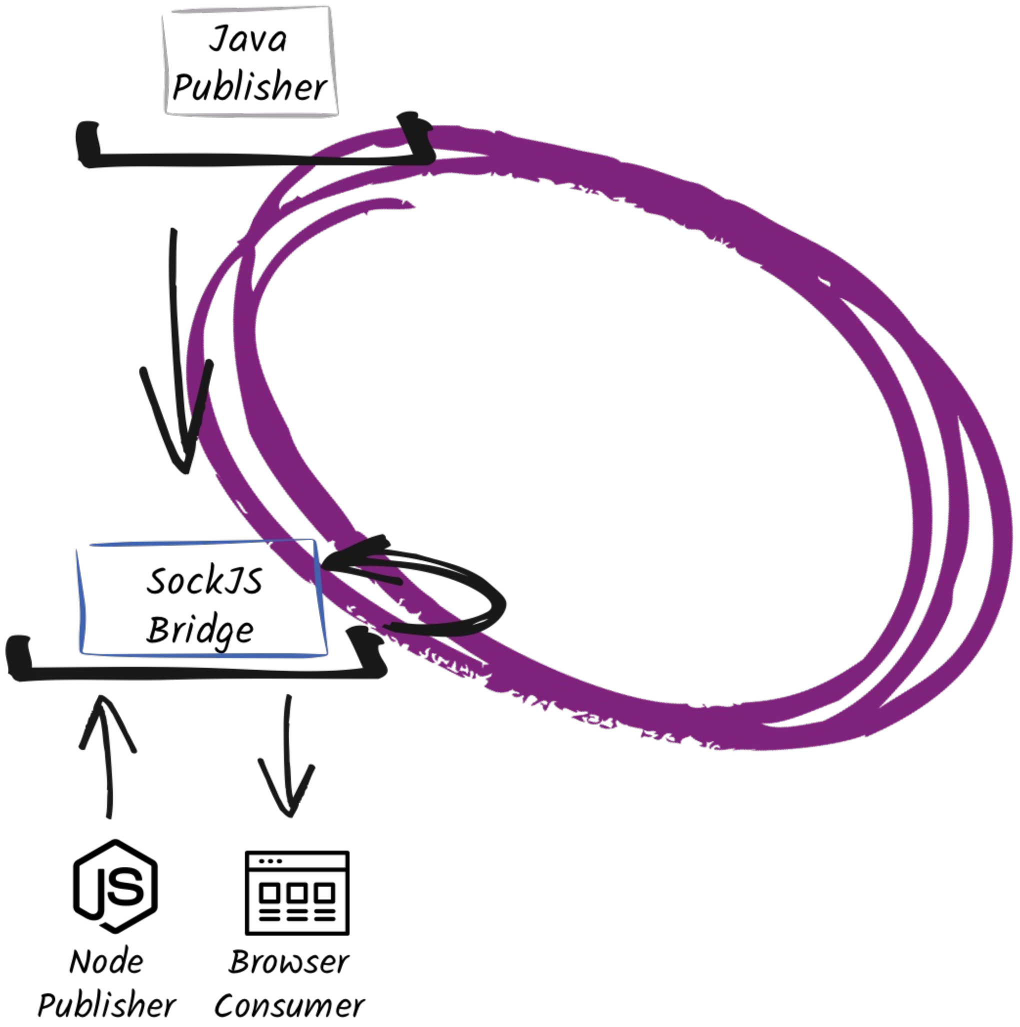
Almost anything can send and receive messages



DISTRIBUTED EVENT BUS

Let's have a java (Vert.x) app, and a node app sending data just here:

DISTRIBUTED EVENT BUS



EVENTBUS CLIENTS AND BRIDGES

Bridges

- SockJS: browser, node.js
- TCP: languages / systems able to open a TCP socket
- Stomp
- AMQP
- Camel

Clients:

- Go, C#, C, Python...

RELIABILITY PATTERNS

Don't be fool, be prepared to fail

RELIABILITY

It's not about being bug-free or bullet proof,
we a **humans**.

It's about being prepared to **fail**,
and handling these **failures**.

MANAGING FAILURES

Distributed communication may fail

`AsyncResult` lets us manage these failures:

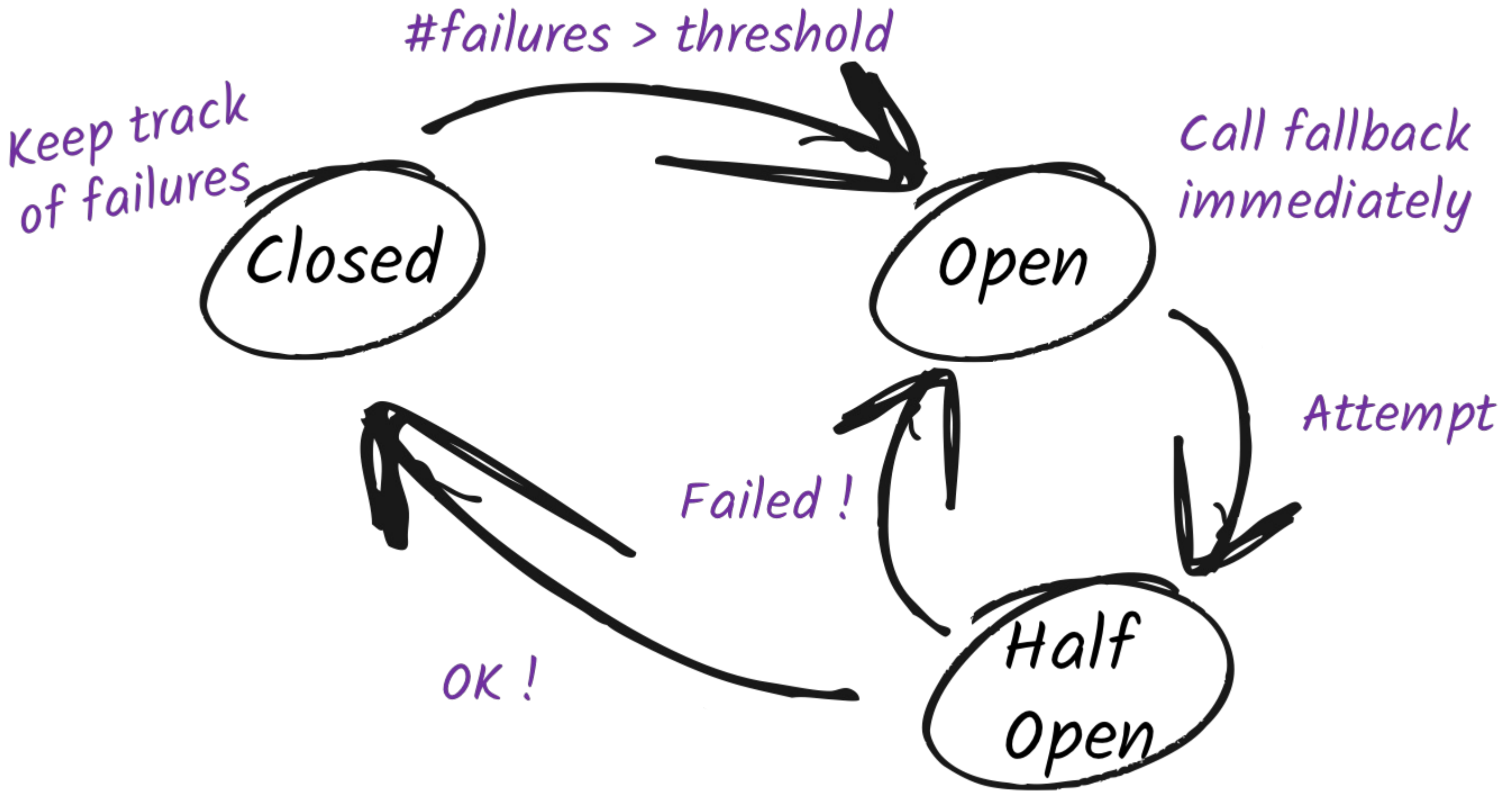
```java
doSomethingAsync(param1, param2,
  ar -> {
    if (ar.failed()) {
      System.out.println("D'oh, it has failed !");
    } else {
      System.out.println("Everything fine ! ");
    }
});
```

MANAGING FAILURES

Adding timeouts

```
vertx.eventbus().send(..., ...,
  new DeliveryOptions().setSendTimeout(1000),
  reply -> {
    if (reply.failed()) {
      System.out.println("D'oh, he did not reply to me !");
    } else {
      System.out.println("Got a mail " + reply.result().body());
    }
});
```

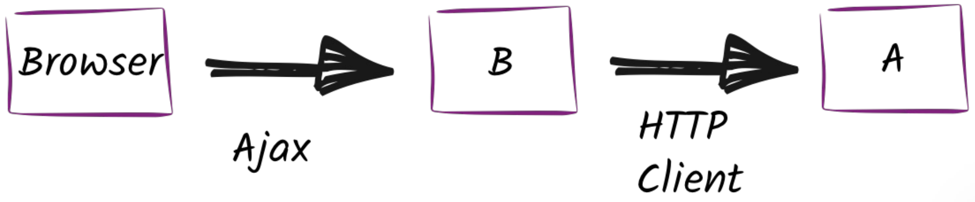
CIRCUIT BREAKER



CIRCUIT BREAKER

```
cb.executeWithFallback(future -> {
  // Async operation
  client.get("/", response -> {
    response.bodyHandler(buffer -> {
      future.complete("Ola " + buffer.toString());
    });
  })
    .exceptionHandler(future::fail)
    .end();
  },

  // Fallback
  t -> "Sorry... " + t.getMessage() + " (" + cb.state() + ")"
)
  // Handler called when the operation has completed
  .setHandler(content -> /* ... */);
```

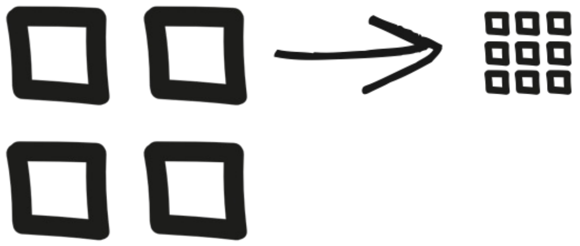
CIRCUIT BREAKER



Invoke

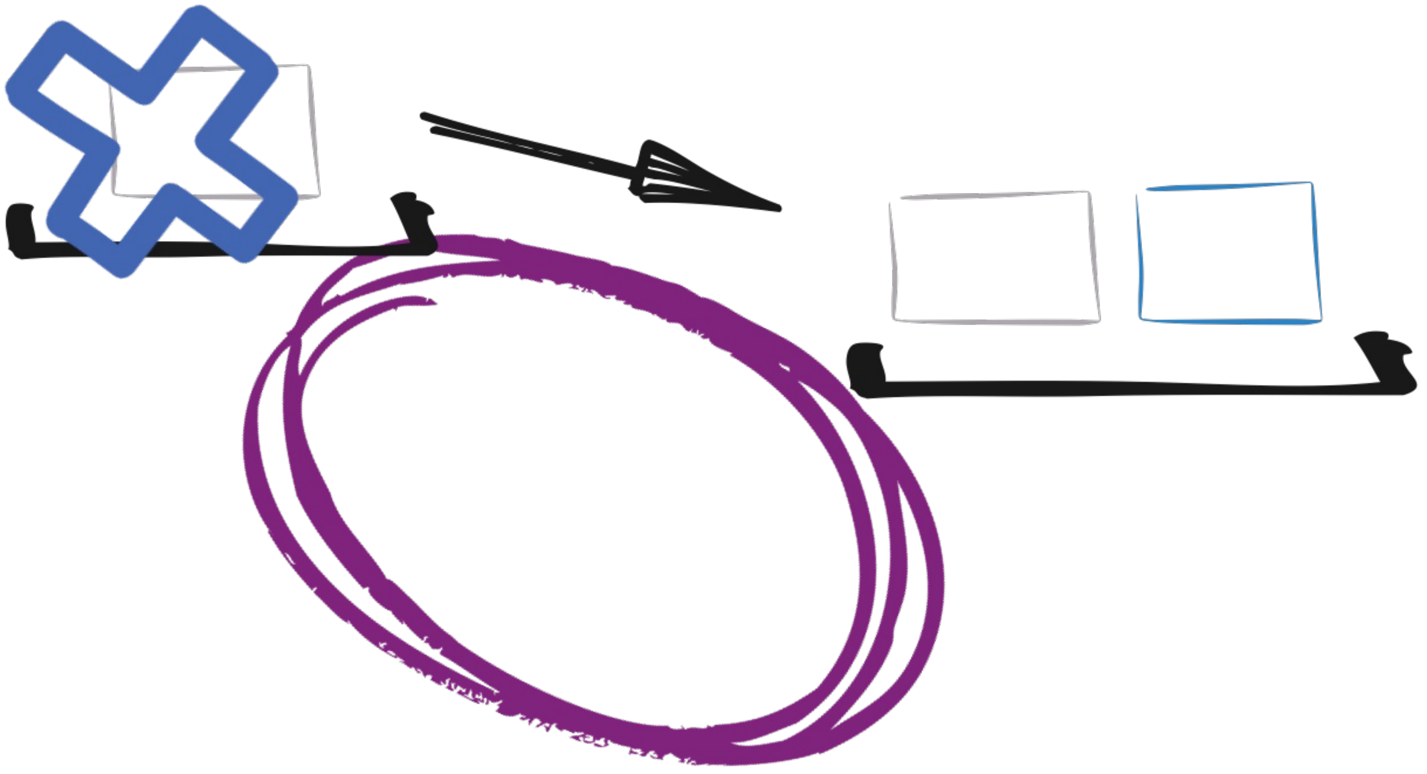
VERTICLE FAIL-OVER

- Verticles are chunk of code that get deployed and run by Vert.x
- Verticles can deploy other verticles
- Verticles can be written in Java, Groovy, JavaScript, Ruby, Ceylon...



VERTICLE FAIL-OVER

In **High-Availability** mode, verticles deployed on a node that **crashes** are redeployed on a sane node of the cluster.



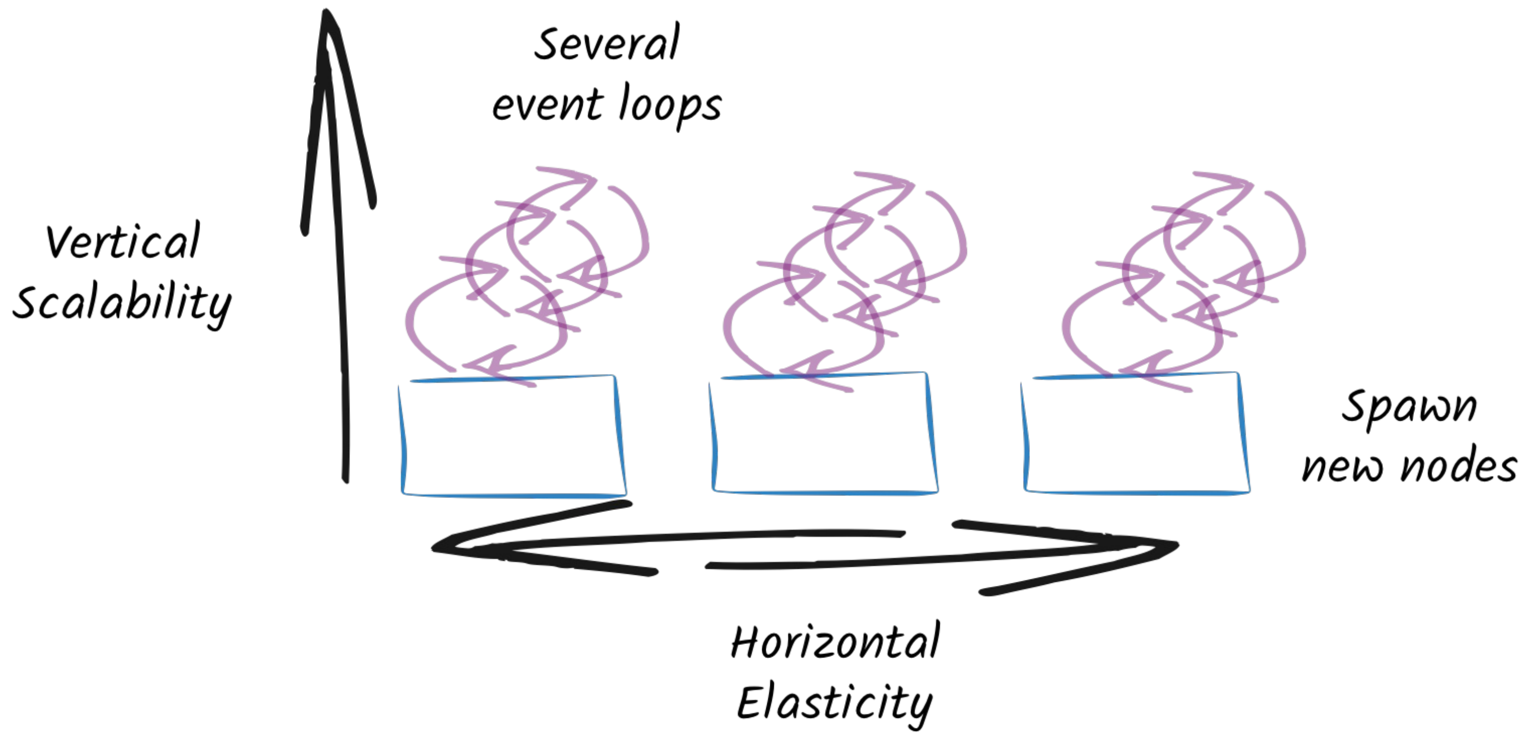
VERTICLE FAIL-OVER

Invoke

ELASTICITY PATTERNS

Be prepared to be famous

ELASTICITY PATTERNS

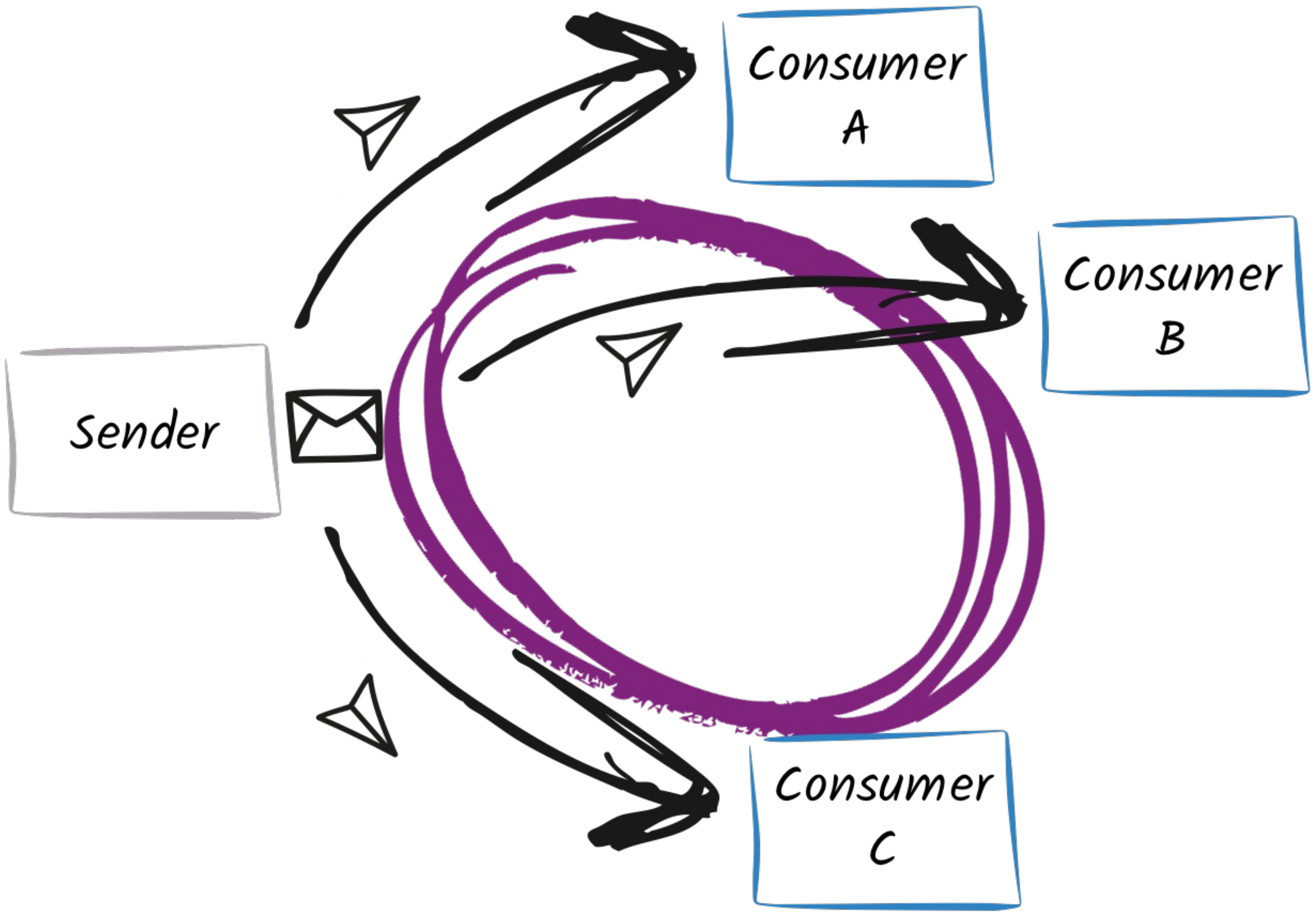


BALANCING THE LOAD

When several consumers listen to the same address, Vert.x dispatches the sent messages using a **round robin**.

So, to improve the scalability, just spawn a new node!

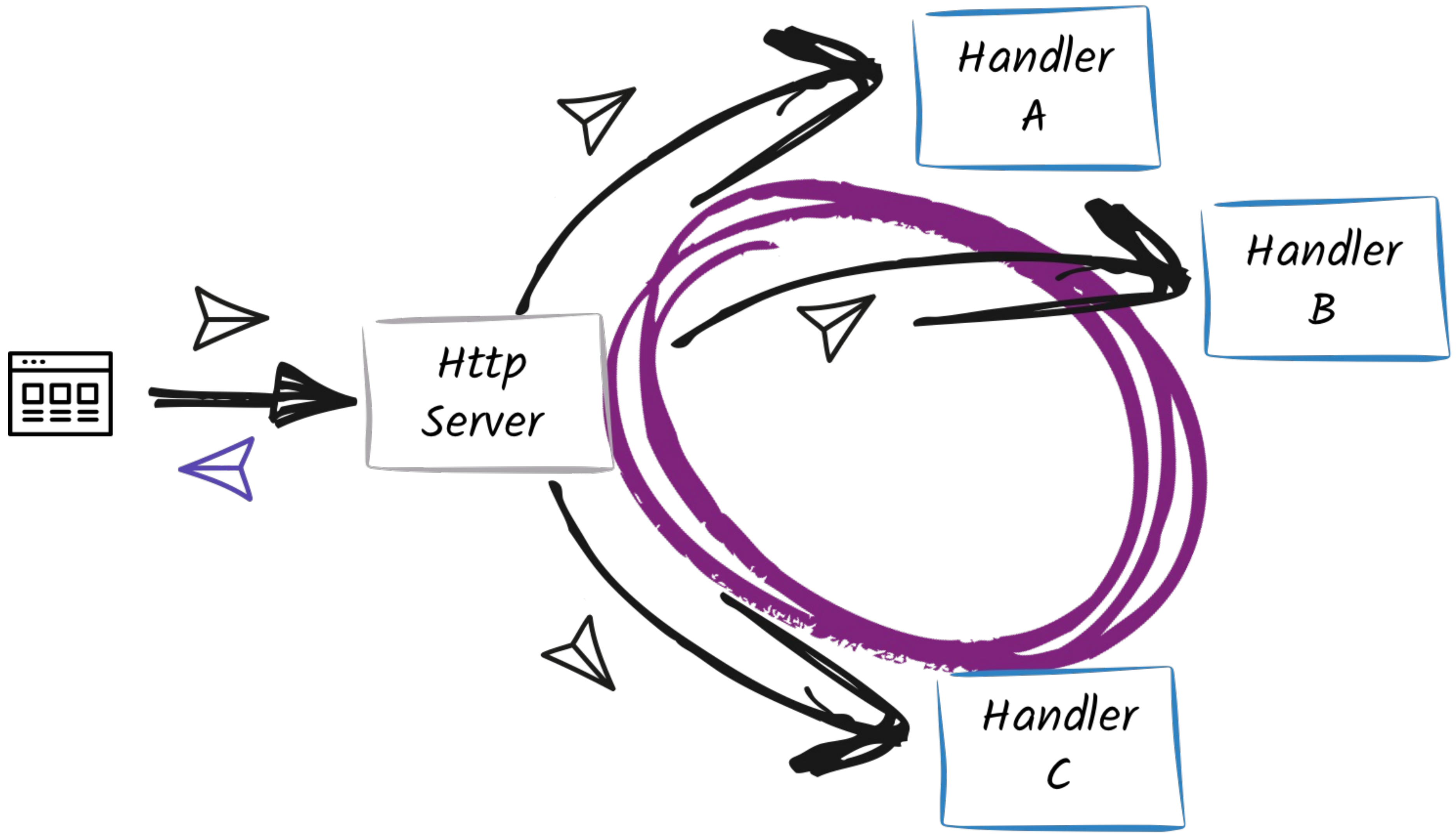
BALANCING THE LOAD



BALANCING THE LOAD

Invoke

SCALING HTTP

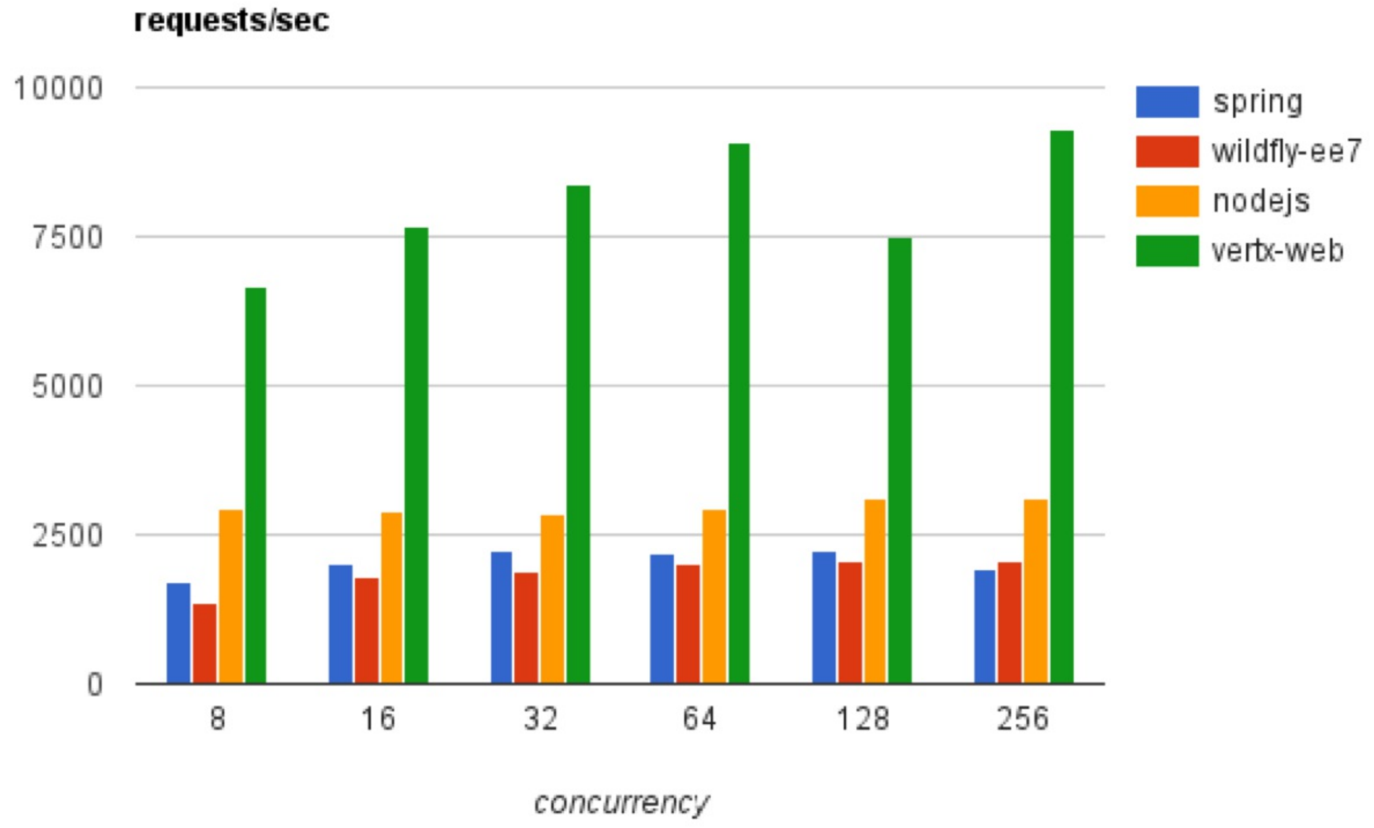


WHAT ABOUT PERFORMANCES ?

Because we do it well, and we do it fast

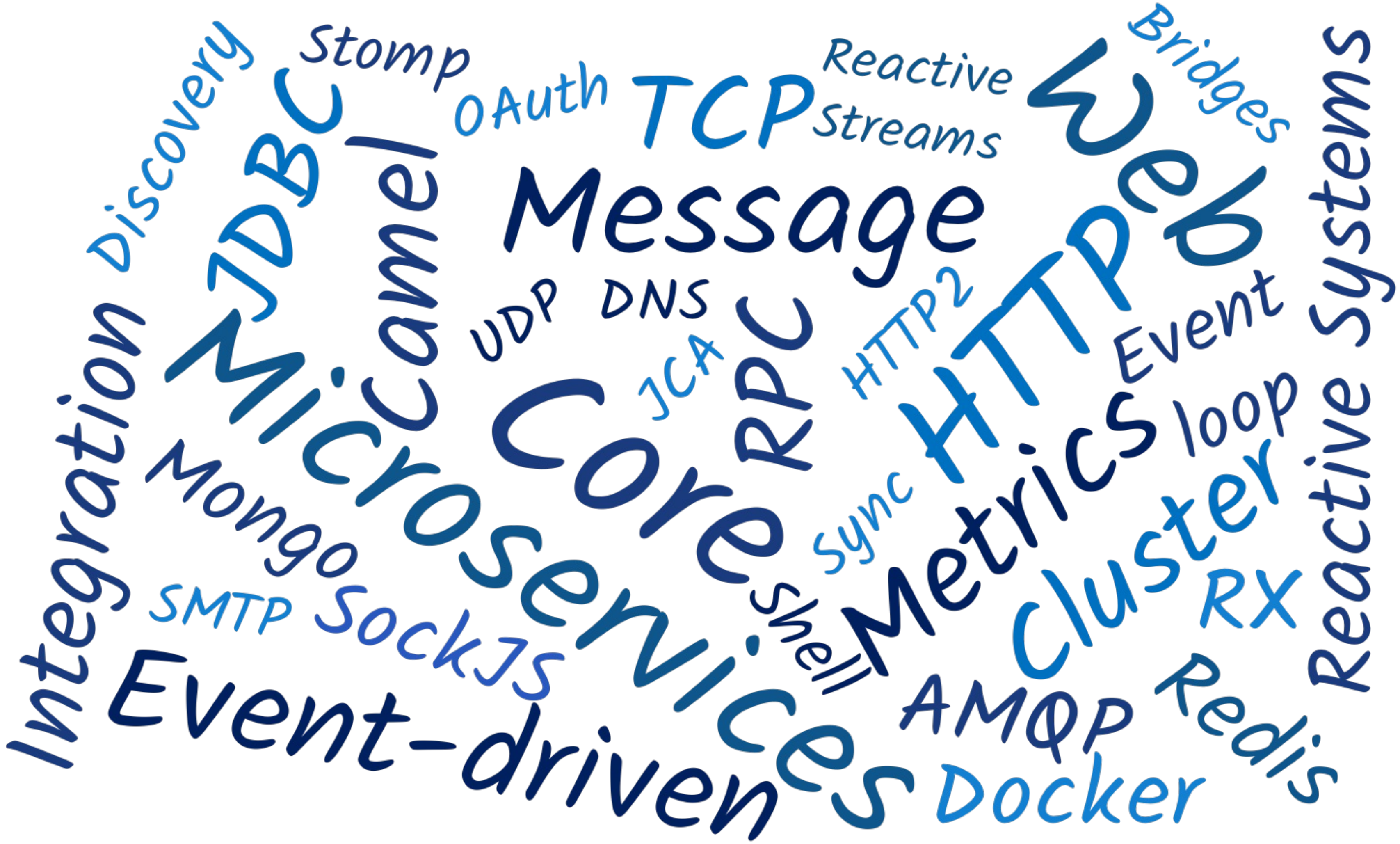
TECHEMPOWER - FORTUNE

Request -> JDBC (query) -> Template engine -> Response



THIS IS NOT THE END();

But the first step on the Vert.x path


Microservices
Core
Message
Web
HTTP
Metrics
Event-driven
Integration
Camel
JDBC
Cluster
Reactive Systems
Reactive
Discovery
Stomp
OAuth
TCP
Reactive
Streams
Bridges
UDP
DNS
RPC
HTTP2
JCA
Event
loop
Mongo
Sync
Shell
SMTP
SockJS
RX
AMQP
Redis
Docker

HOW TO START ?

- http://vertx.io
- http://vertx.io/blog/posts/introduction-to-vertx.html
- http://escoffier.me/vertx-hol/ (HOL3180)
- Reactive Microservices with Vert.x (CON5389)

Vote for us !

jax® INNOVATION AWARDS 2016

https://jaxlondon.com/jax-awards/