VERT.X

A TOOLKIT TO BUILD DISTRIBUTED REACTIVE SYSTEMS

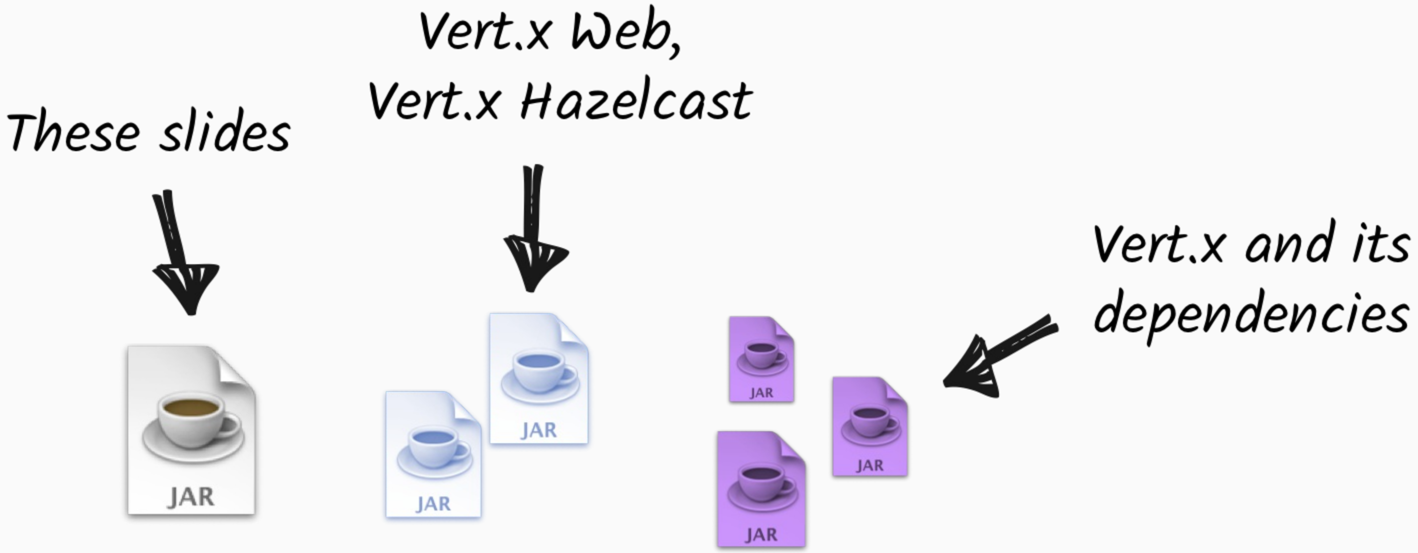
CLEMENT ESCOFFIER

Vert.x Core Developer, Red Hat

VERT.X IS A **TOOLKIT** TO BUILD **DISTRIBUTED** AND **REACTIVE** APPLICATIONS 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*, ...)



These slides

Vert.x Web,
Vert.x Hazelcast

Vert.x and its
dependencies

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)

DISTRIBUTED

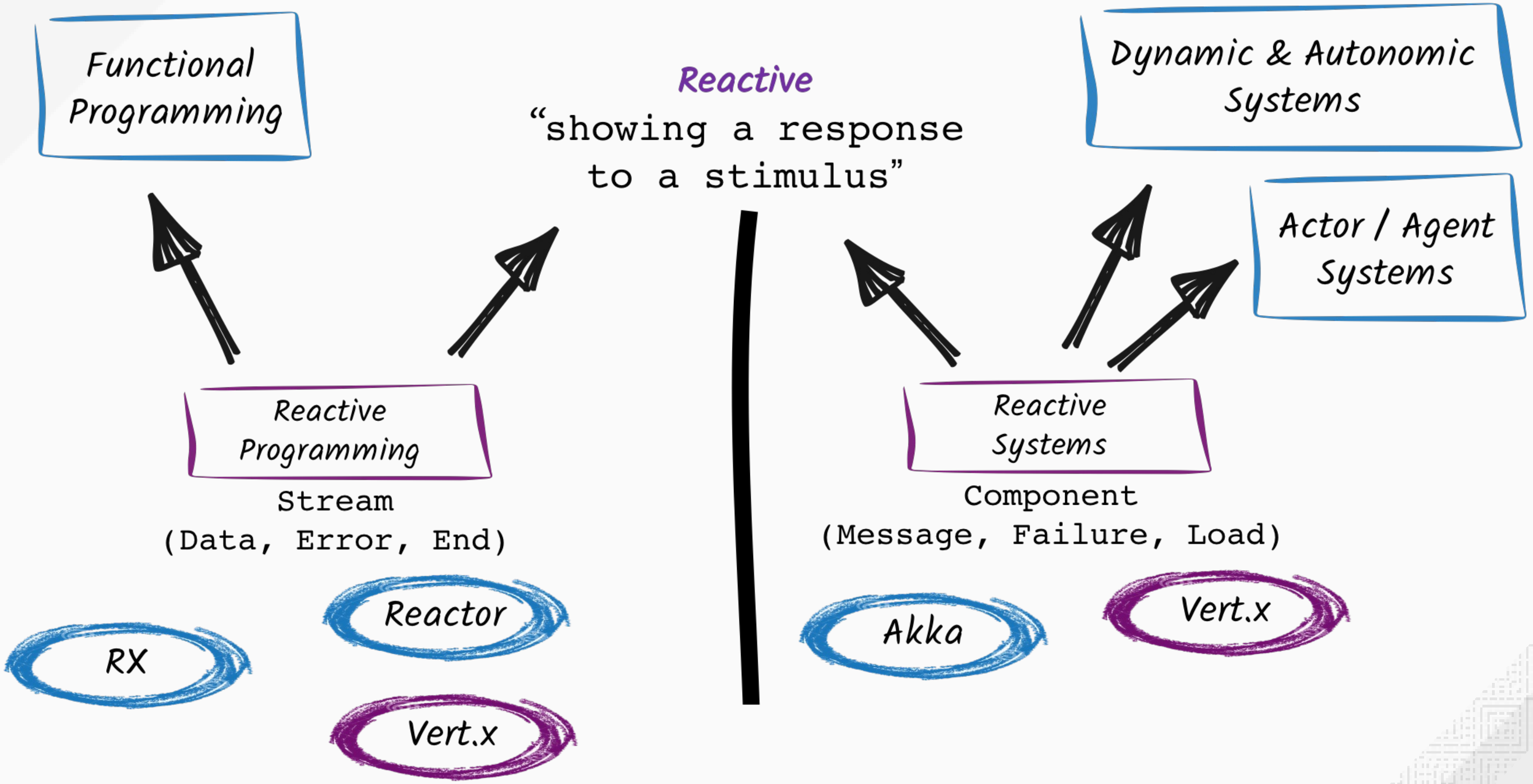
“ You know you have a distributed system when the crash of a **microservice** you've never heards of stops you from getting any work done.”
(Everyone having developed microservices)

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



Functional Programming

Reactive

"showing a response to a stimulus"

Dynamic & Autonomic Systems

Actor / Agent Systems

Reactive Programming

Stream
(Data, Error, End)

RX

Reactor

Vert.x

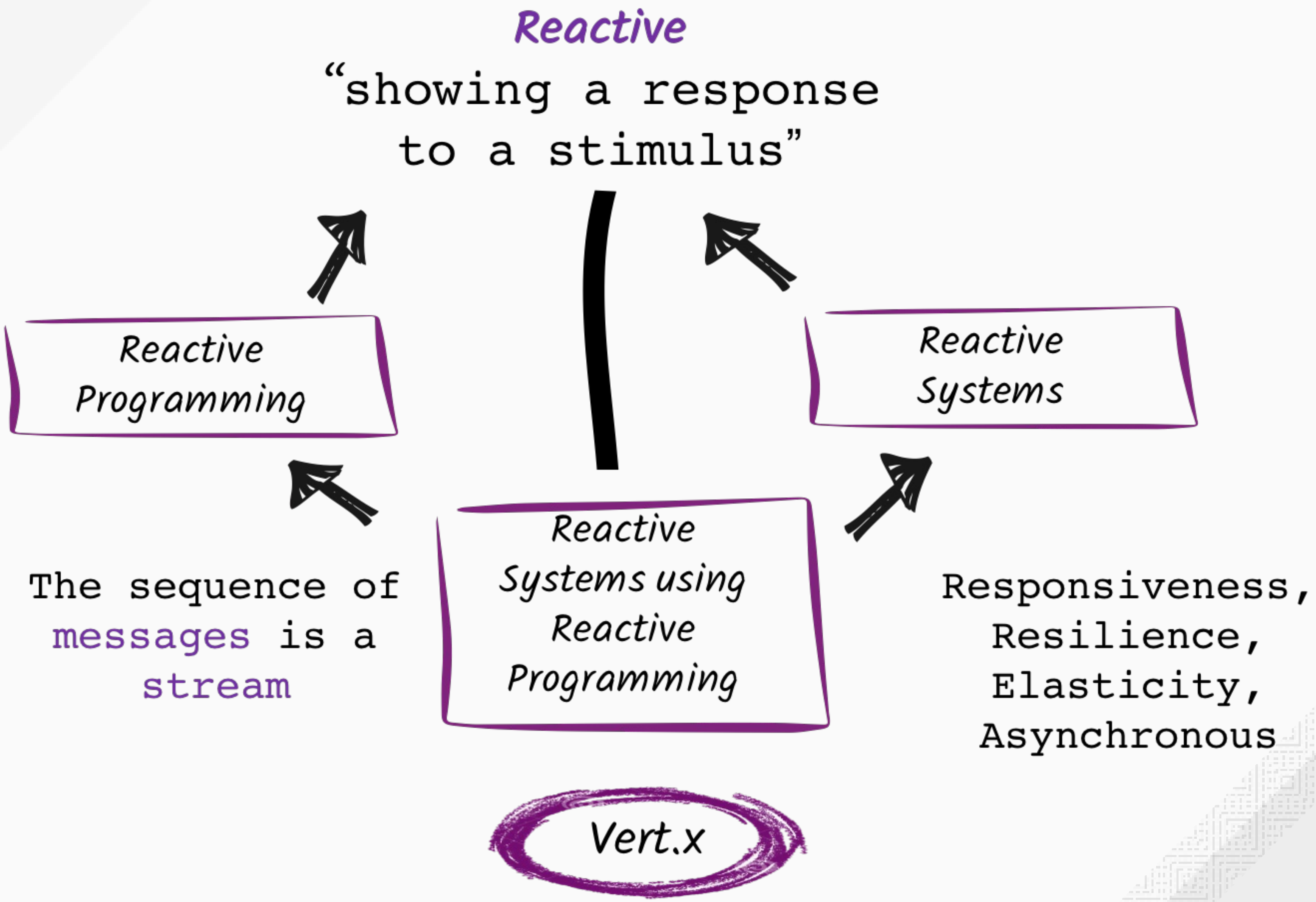
Reactive Systems

Component
(Message, Failure, Load)

Akka

Vert.x

REACTIVE SYSTEMS + REACTIVE PROGRAMMING



Reactive

“showing a response to a stimulus”

Reactive Programming

Reactive Systems

The sequence of messages is a stream

Reactive Systems using Reactive Programming

Responsiveness, Resilience, Elasticity, Asynchronous

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 reactive distributed systems & microservices

A TOOLKIT TO

Build **distributed** systems:

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

Build **microservice** systems:

- Asynchronous
- Location transparency
- Resilience patterns
- Simple deployment & management

WHAT DOES VERT.X PROVIDE ?

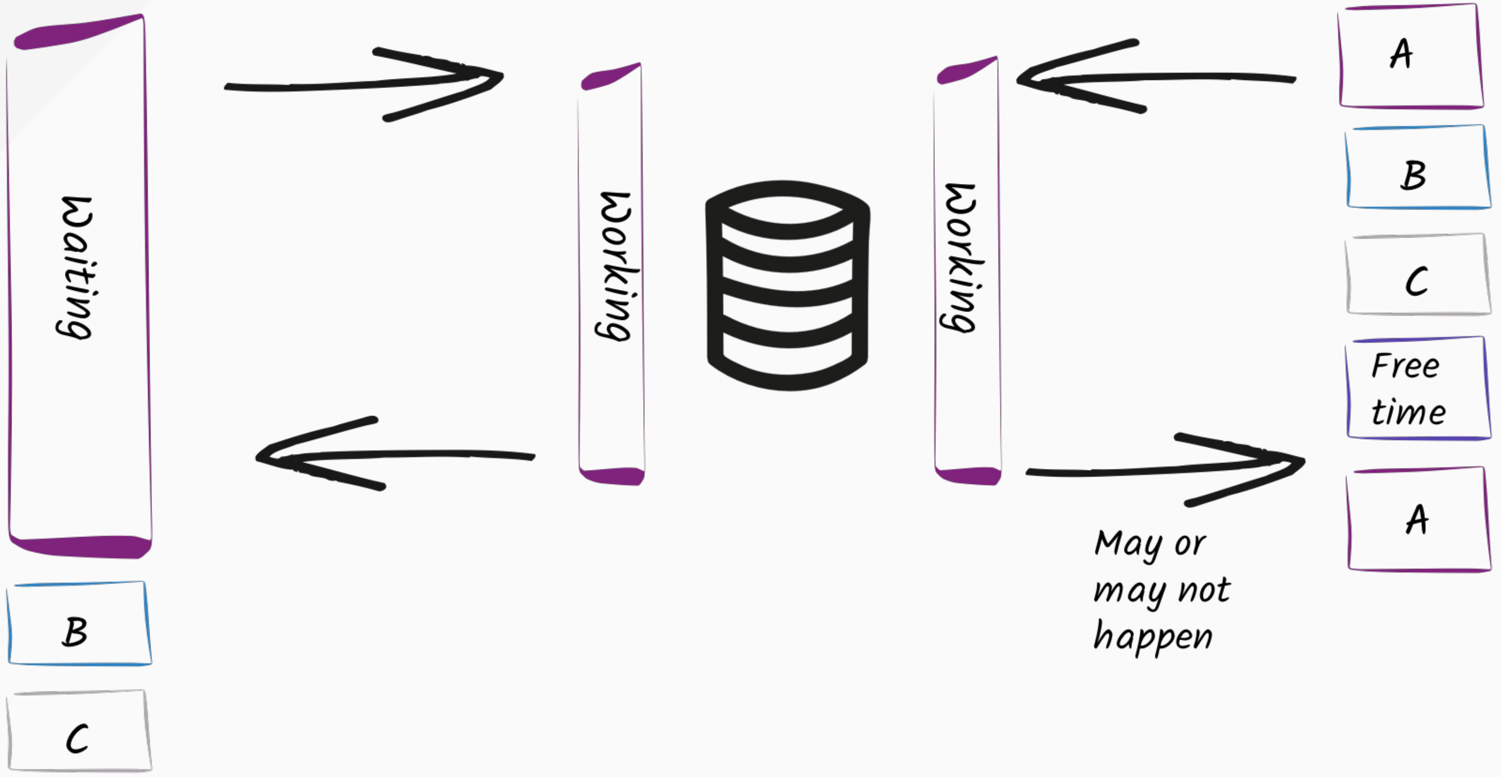
- TCP, UDP, HTTP 1 & 2 servers and clients
- (non-blocking) DNS client
- Clustering
- Event bus (messaging)
- Distributed data structures
- (built-in) Load-balancing
- (built-in) Fail-over
- Pluggable service discovery, circuit-breaker
- Metrics, Shell

REACTIVE

Build **reactive distributed** systems / microservices:

- **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 - RX variant
Single<X> single = rxDoSomething(a, b);
single.subscribe(
   r -> {  /* Completion handler */ });
```

REQUEST - REPLY INTERACTIONS

HTTP, TCP, RPC...

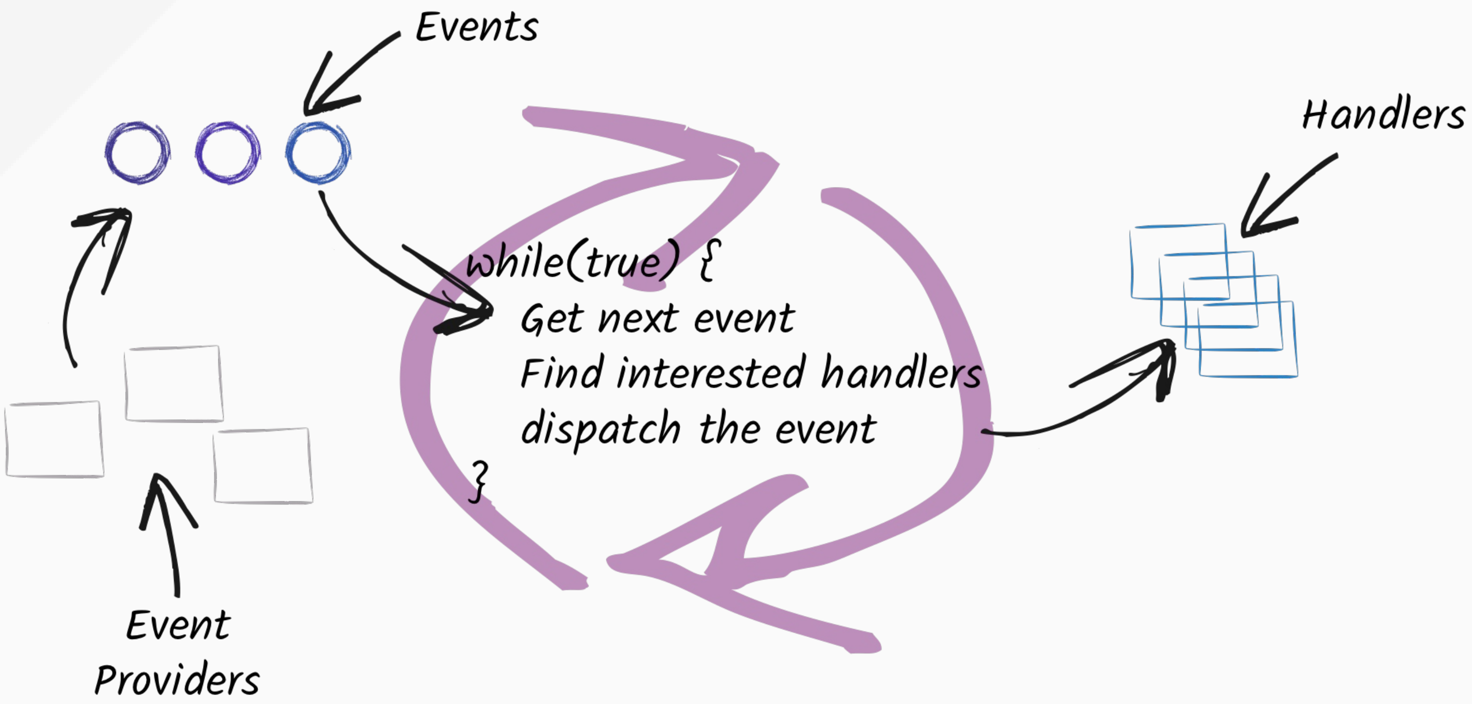
VERT.X HELLO WORLD

```
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

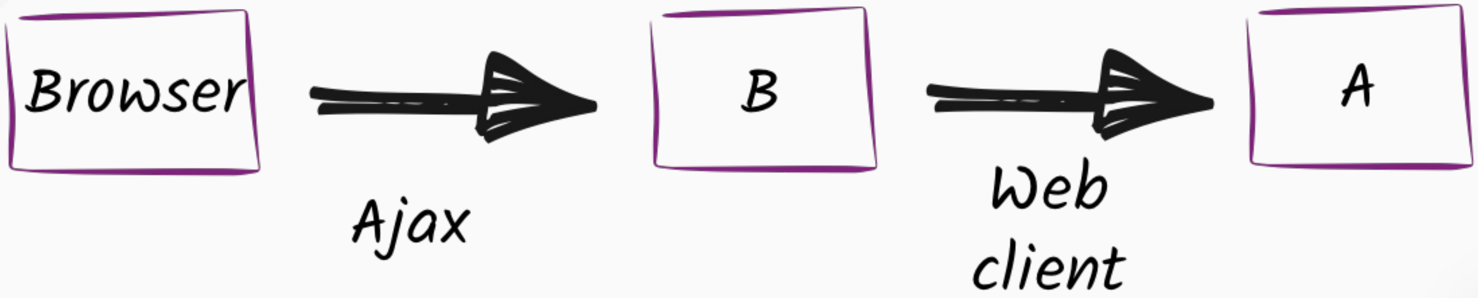


```
while(true) {
    Get next event
    Find interested handlers
    dispatch the event
}
```

VERT.X ASYNC WEB CLIENT

```
client.get(SERVICE_PORT, SERVICE_HOST, "/")
  .send(ar -> {
   if (ar.failed()) {
    // Something bad happened
   } else {
    String body = ar.result().bodyAsString();
   }
  });
```

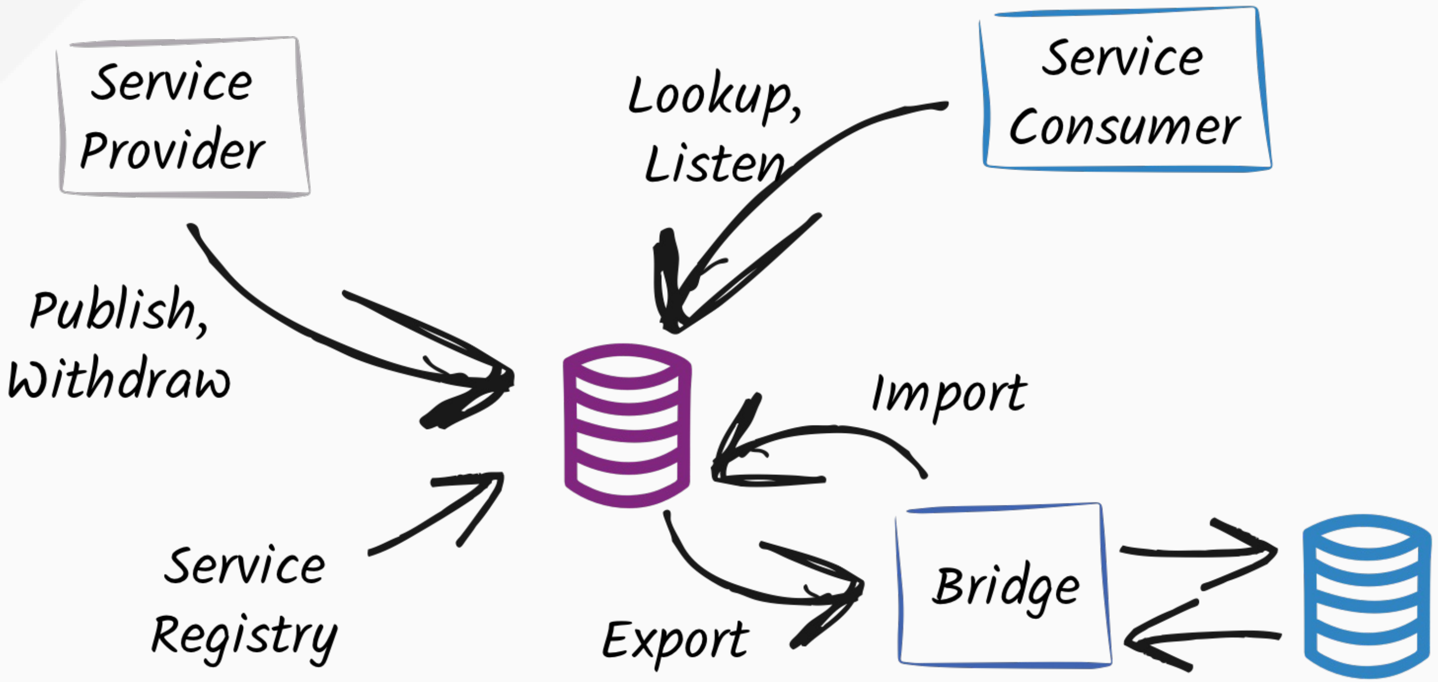
CHAINED HTTP REQUESTS



Invoke

SERVICE DISCOVERY

Locate the services, environment-agnostic



SERVICE DISCOVERY

```
HttpEndpoint.rxGetWebClient(discovery,
    svc ->  svc.equals("vertx-http-server"))
    .subscribe( client -> {
        client.get("/").send(ar -> {
                String body = ar.result().bodyAsString();
        });
      });
```

MESSAGING

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

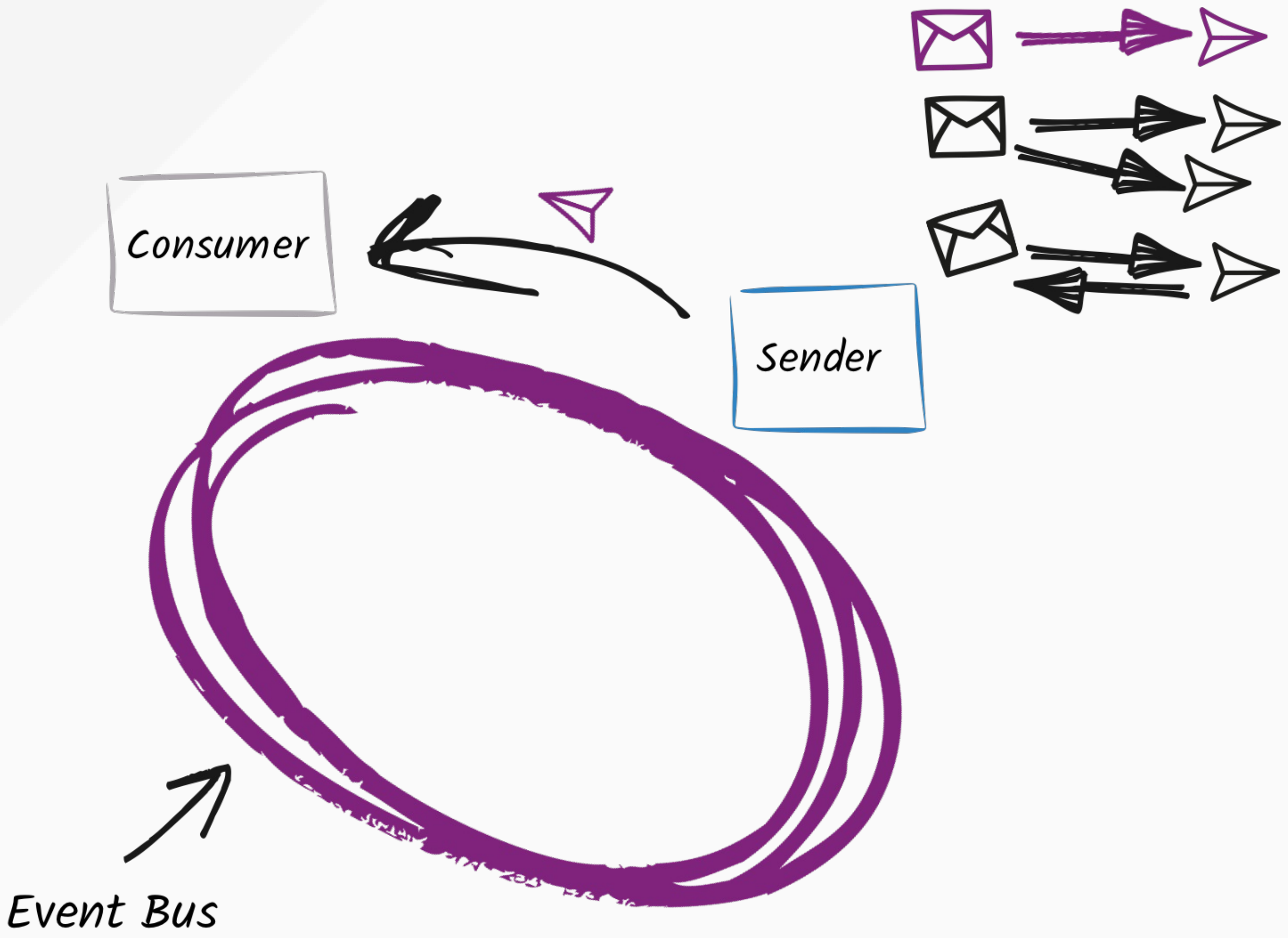
THE EVENT BUS

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)

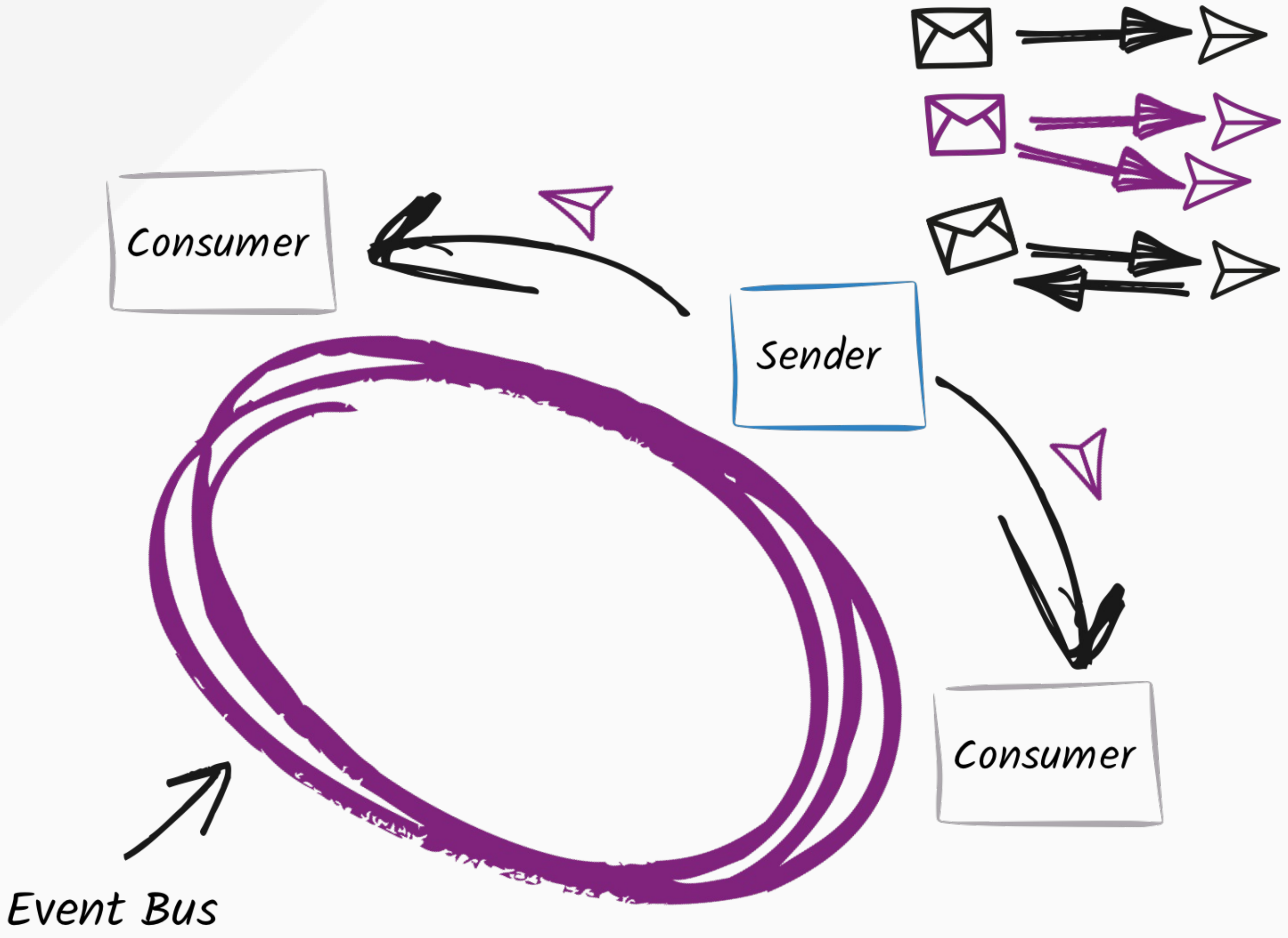
- **Address**: Messages are sent to an address
- **Handler**: Messages are received by Handlers.

POINT TO POINT



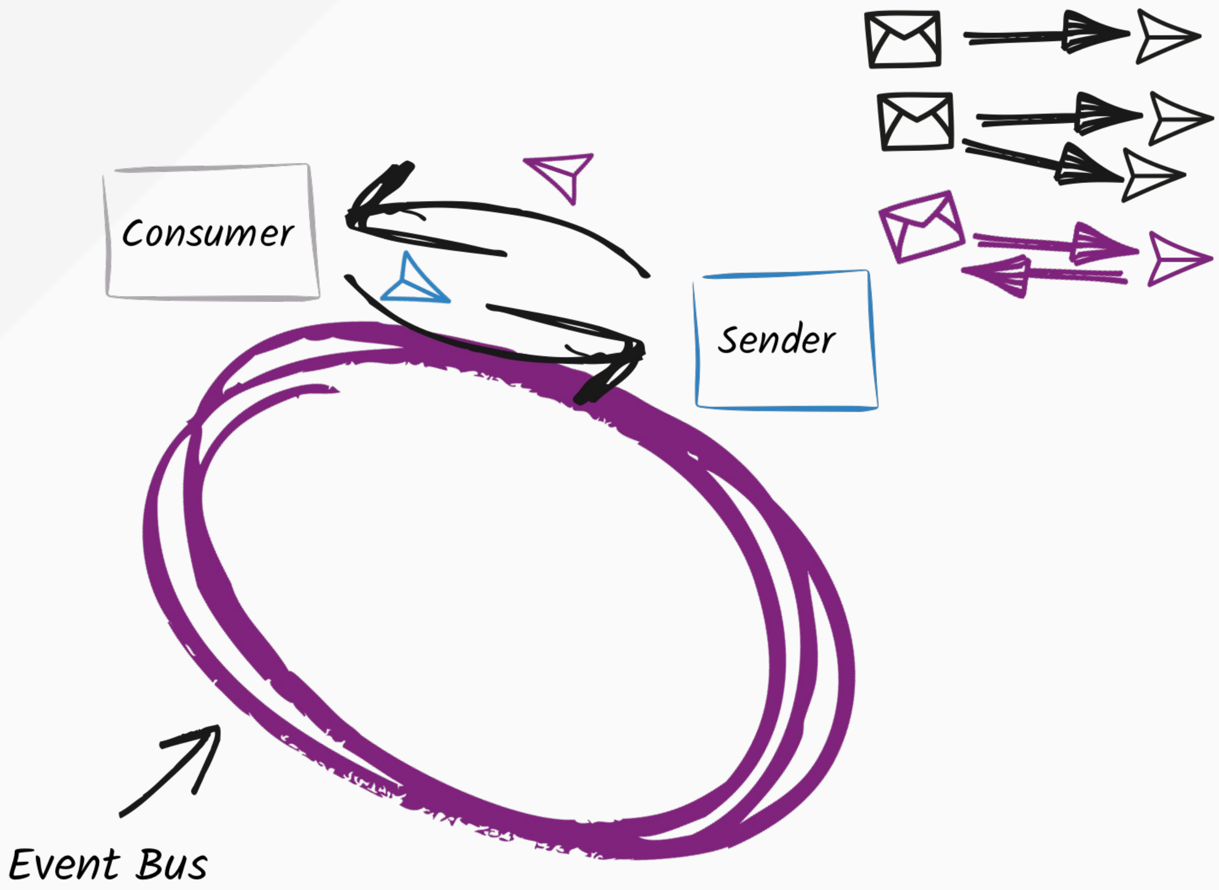
```
vertx.eventBus().send("address", "message");
vertx.eventBus().consumer("address", message -> {});
```

PUBLISH / SUBSCRIBE



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

REQUEST / RESPONSE



Consumer

Sender

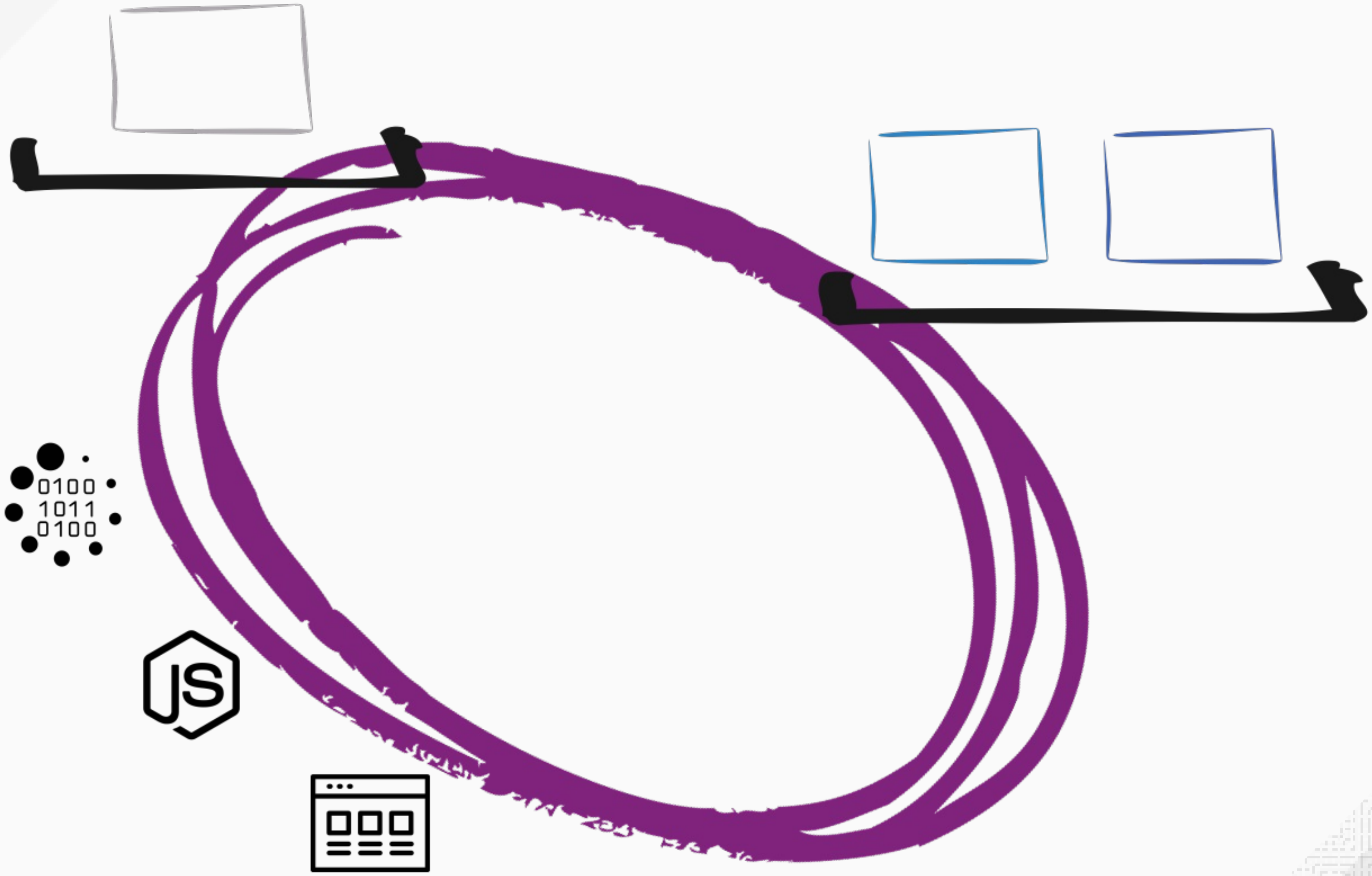
Event Bus

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

DISTRIBUTED EVENT BUS

The event bus is distributed on all the cluster members

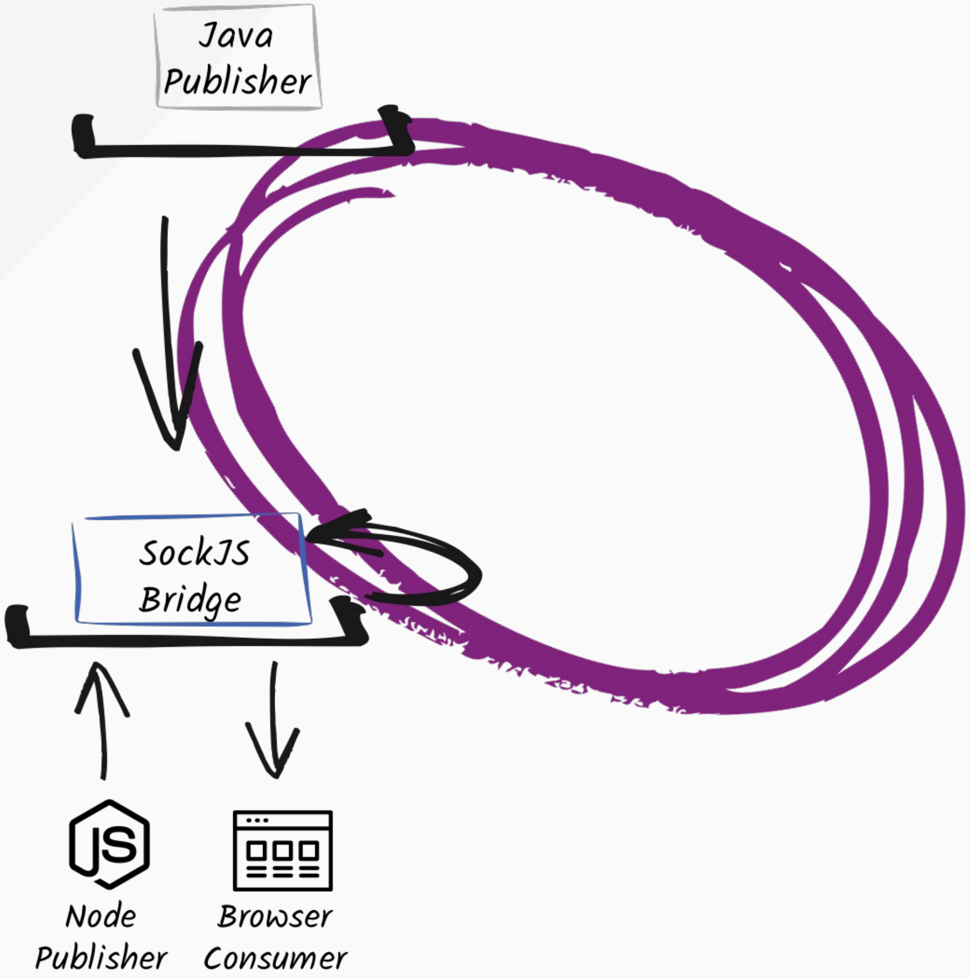
Almost anything can send and receive messages



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
- Apache Camel

Clients:

- Go, C#, C, Python, Swift…

RELIABILITY PATTERNS

Don't be fool, be prepared to fail

MANAGING FAILURES

Distributed communication may fail

AsyncResult lets us manage these failures:

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

MANAGING FAILURES

Distributed communication may fail

Single / Observable let us manage these failures:

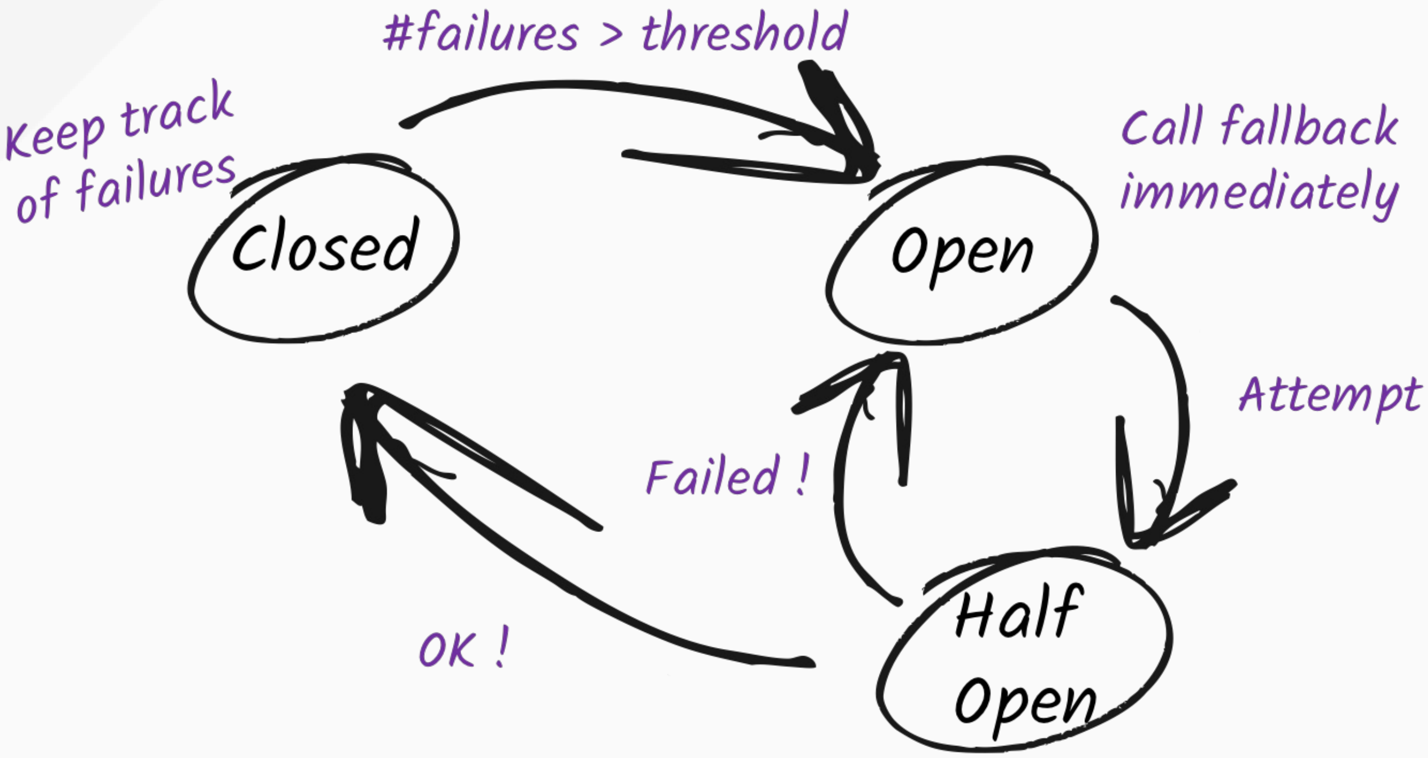
```
doSomethingAsync(param1, param2)
  .subscribe(
    r -> System.out.println("Everything fine ! "),
    e -> System.out.println("D'oh, it has failed !")
);
```

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());
    }
});
```

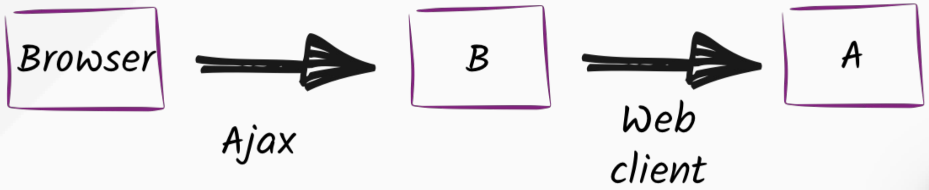
CIRCUIT BREAKER



CIRCUIT BREAKER

```
cb.executeWithFallback(future -> {
  // Async operation
  client.get("/").send(response -> {
      if (response.failed()) {
        future.fail(response.cause());
      } else {
        future.complete("Hello " + response.getResult().bodyAsString());
      }
  }),
  // Fallback
  t -> "Sorry... " + t.getMessage() + " (" + cb.state() + ")"
)
  // Handler called when the operation has completed
  .setHandler(content -> /* ... */);
```

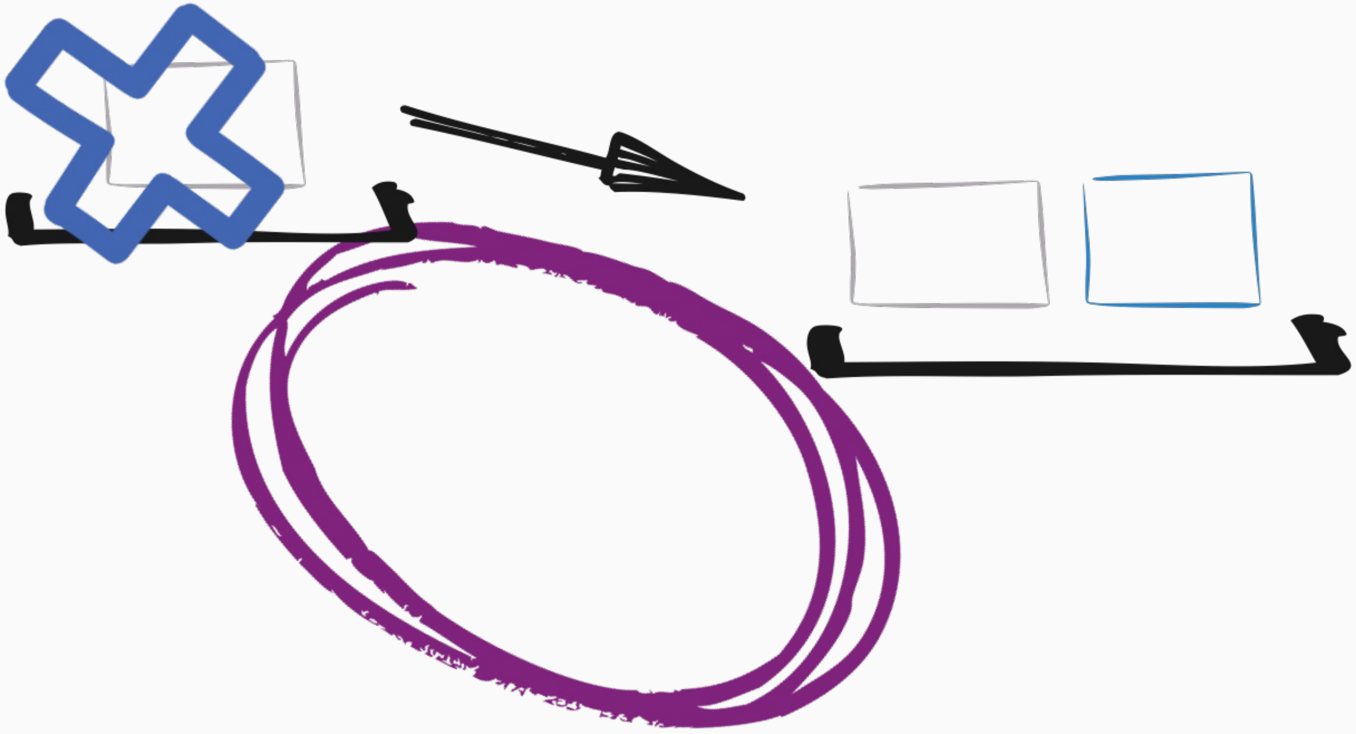
CIRCUIT BREAKER



Invoke

VERTICLE FAIL-OVER

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



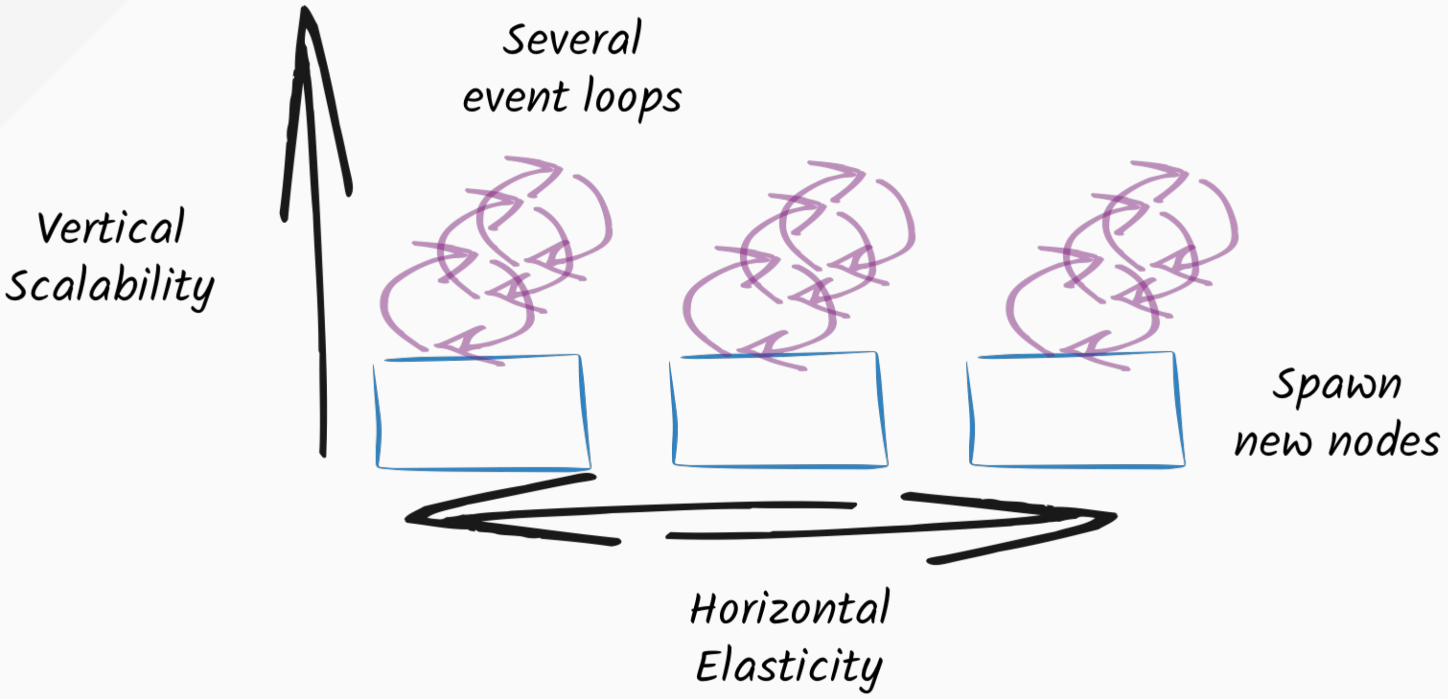
VERTICLE FAIL-OVER

Invoke

ELASTICITY PATTERNS

Be prepared to be famous

ELASTICITY PATTERNS

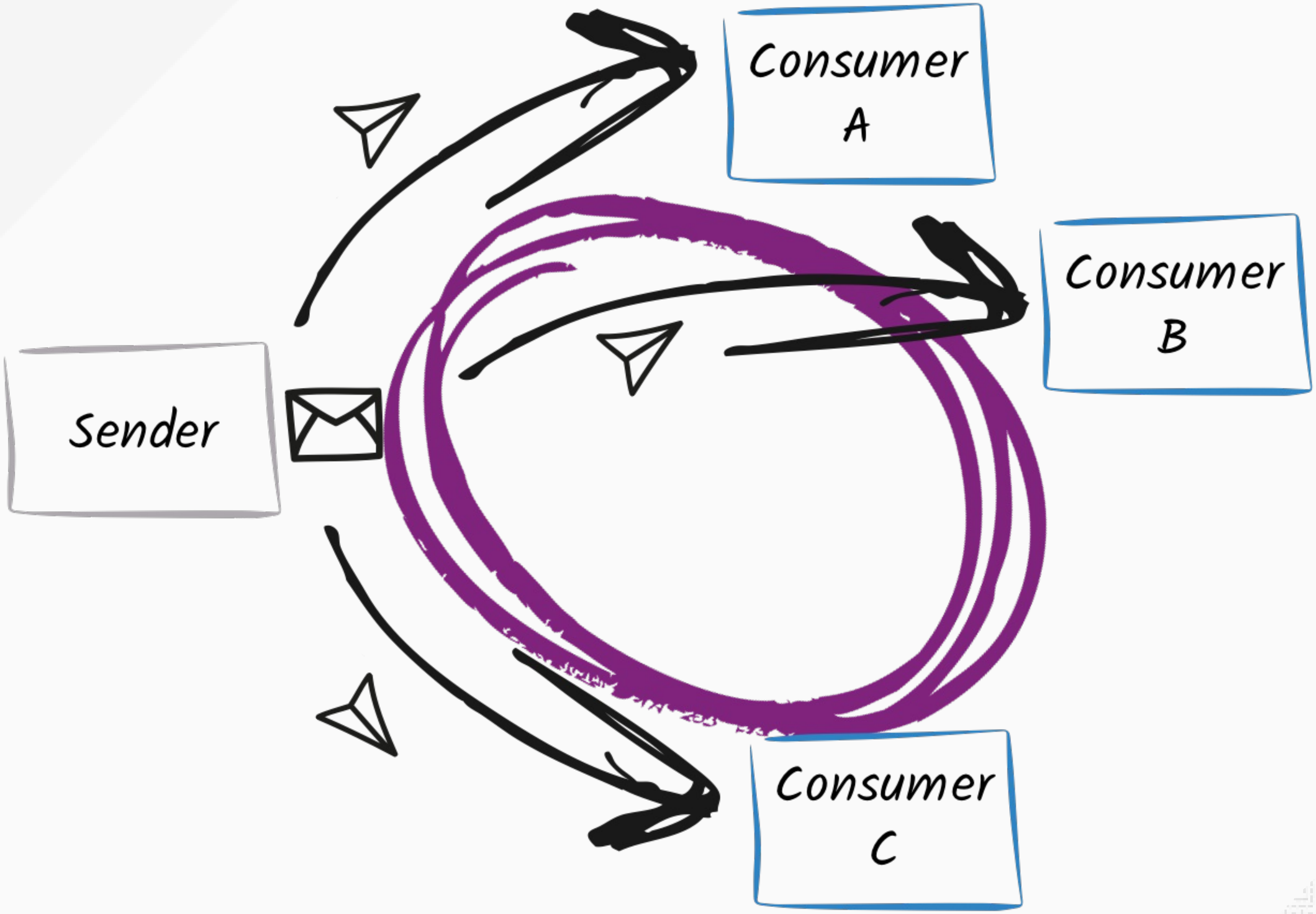


BALANCING THE LOAD

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!

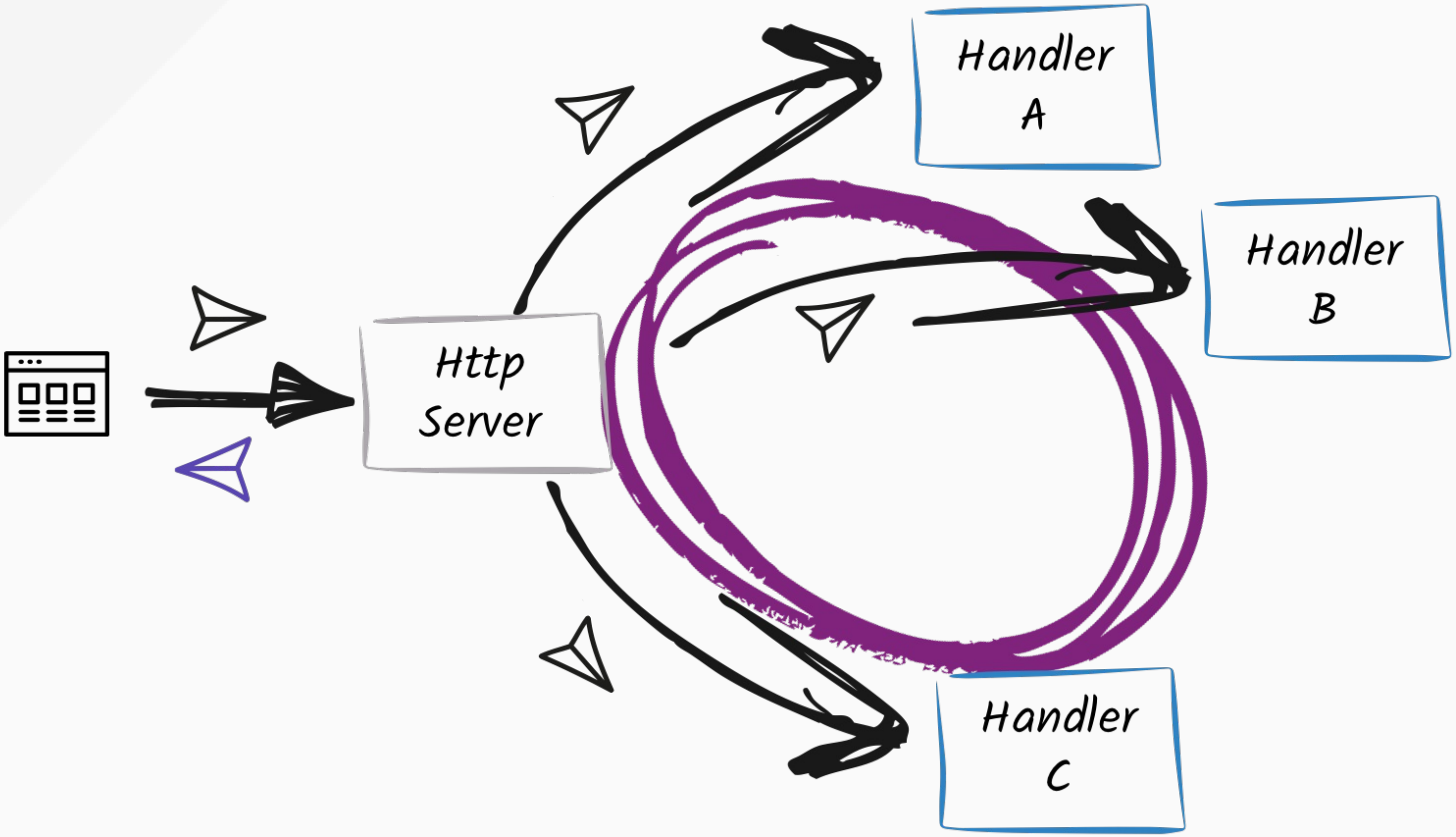
BALANCING THE LOAD



BALANCING THE LOAD

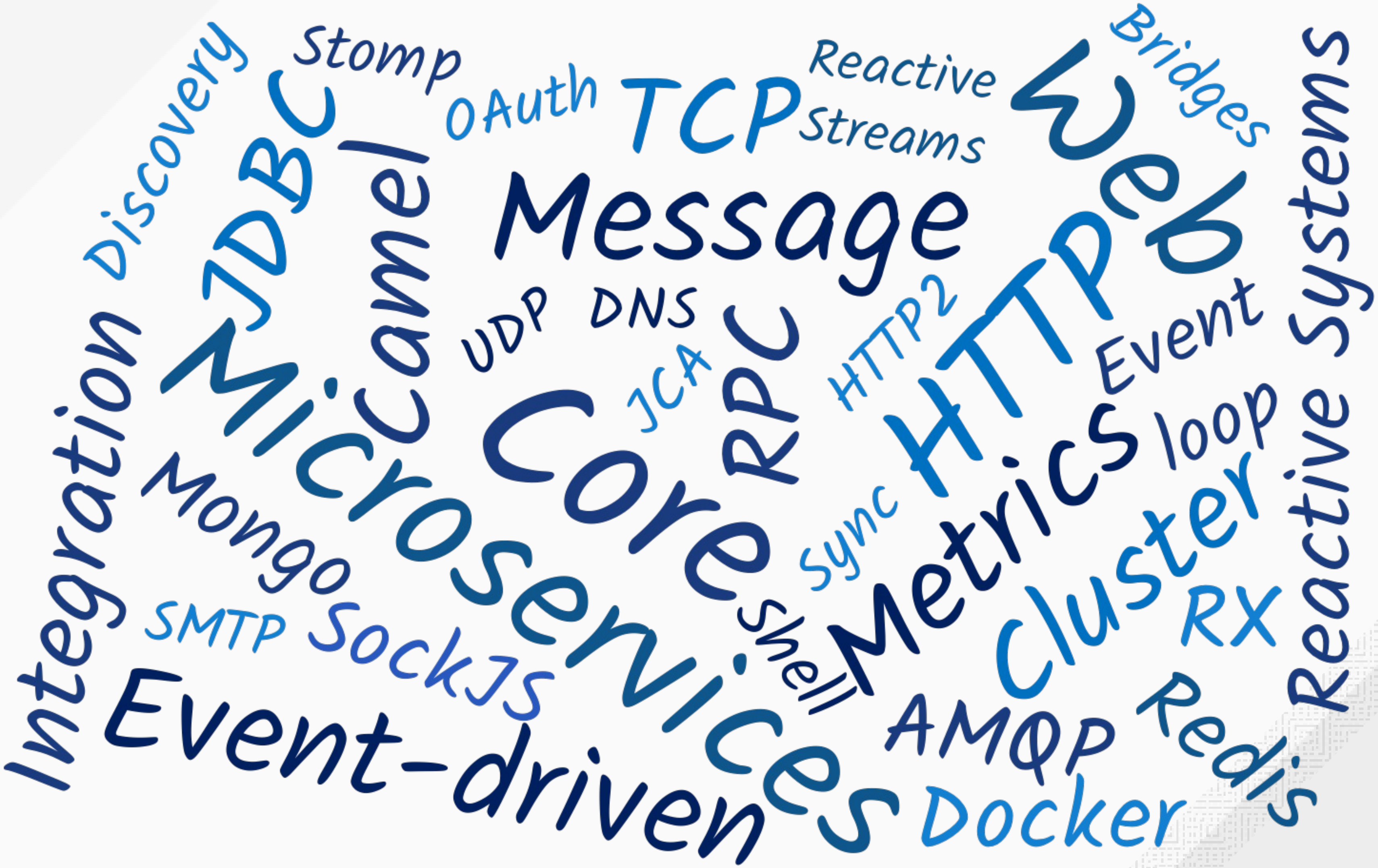
Invoke

SCALING HTTP

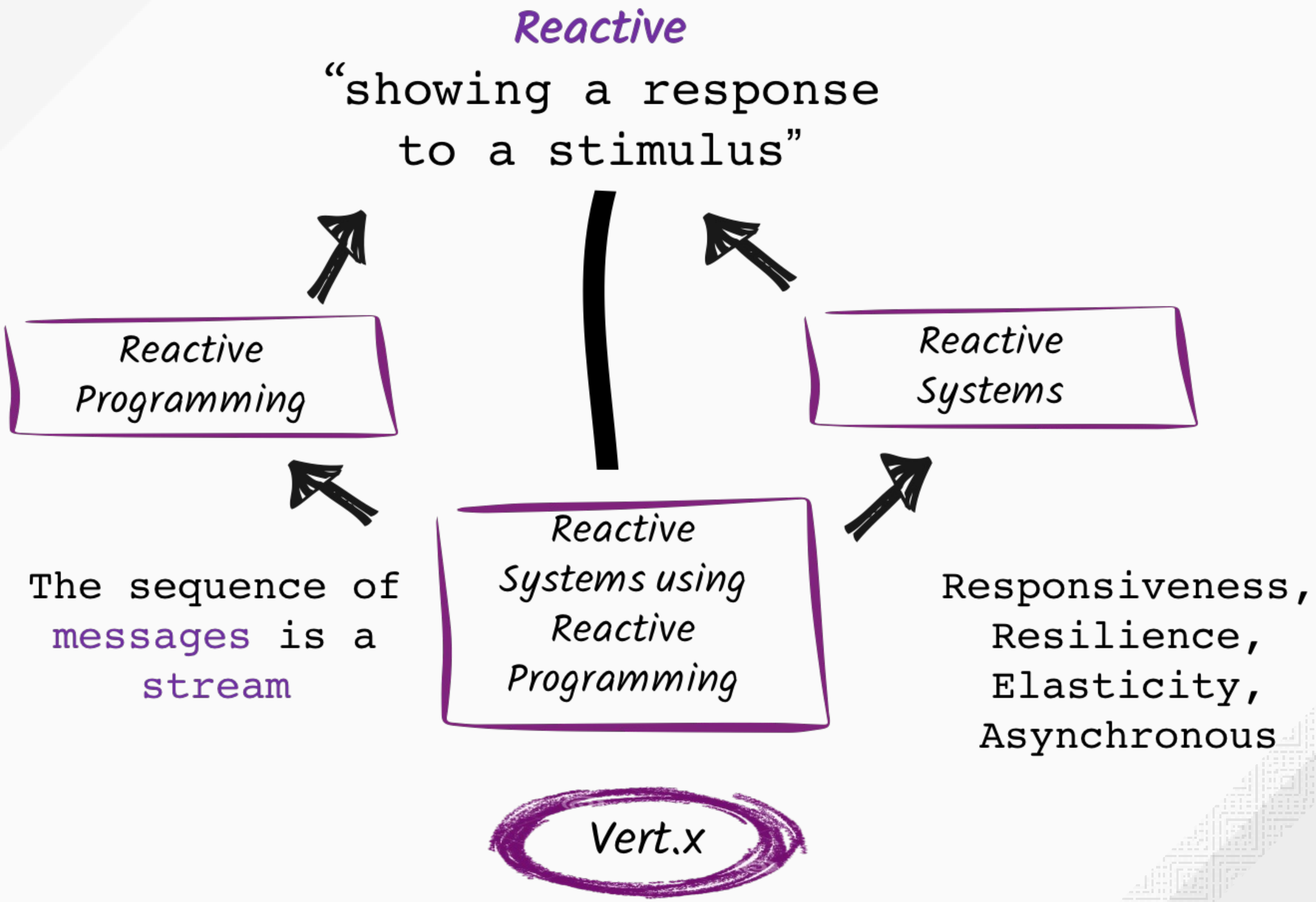


THIS IS NOT THE END();

But the first step on the Vert.x path


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

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

Responsiveness, Resilience, Elasticity, Asynchronous

Vert.x

redhat®

THANK YOU!

🐦 @clementplop

@vertx_project