redhat

# REACTIVE POLYGLOT MICROSERVICES WITH OPENSHIFT AND VERT.X

**CLEMENT ESCOFFIER**

Vert.x Core Developer, Red Hat

redhat

# VERT.X

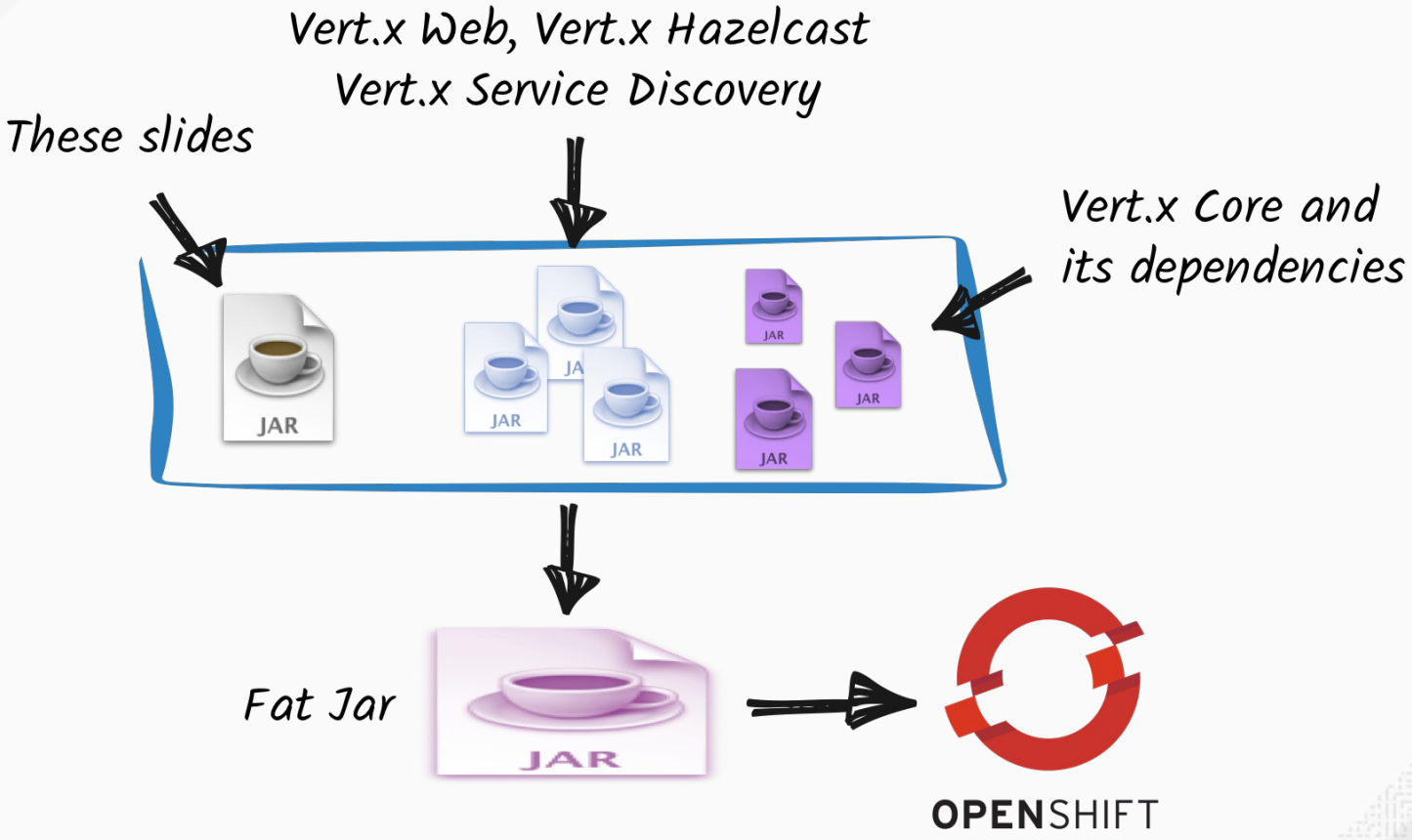
## IN 10 SLIDES...

redhat.

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

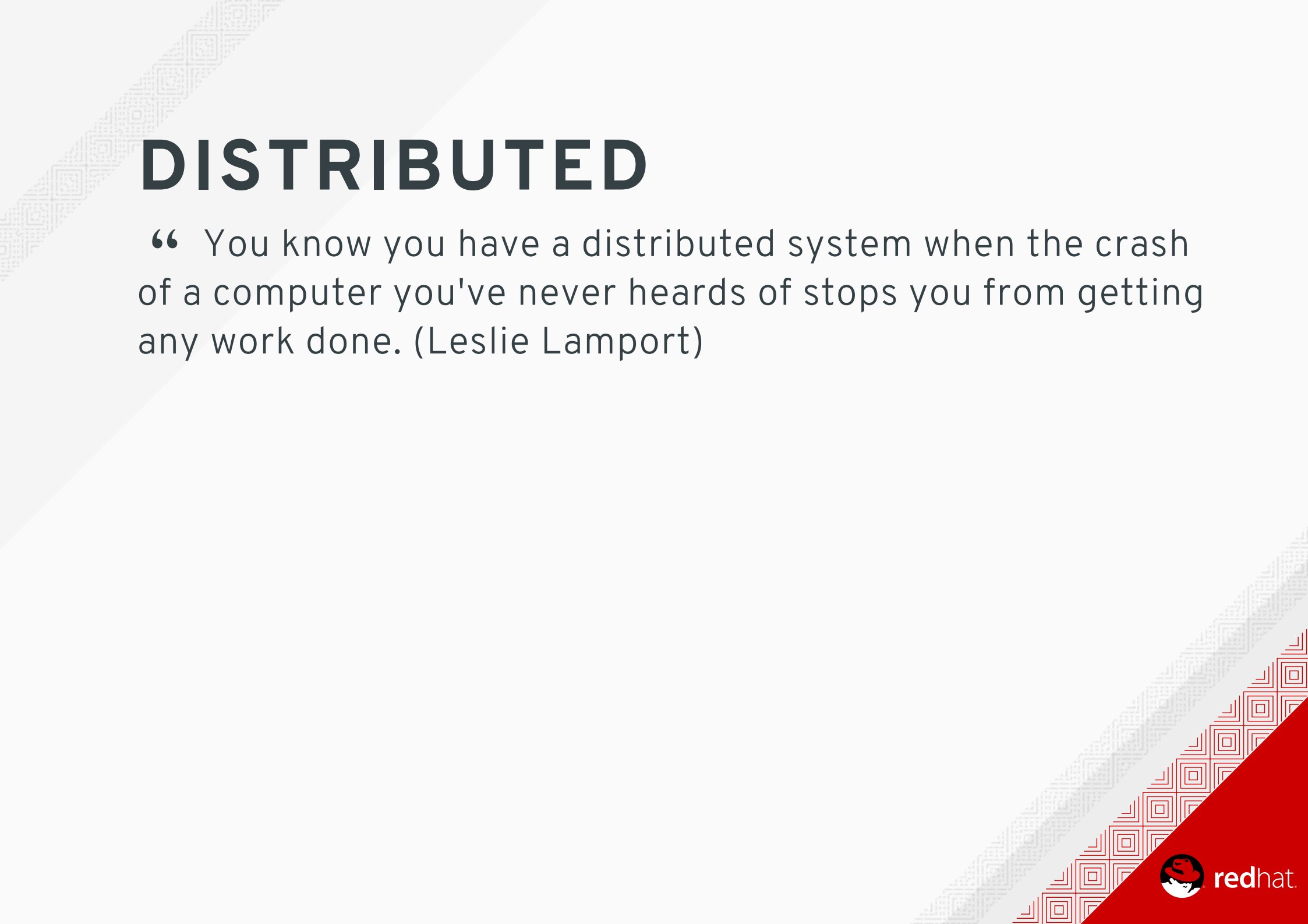
# TOOLKIT

- Vert.x is a plain boring **jar**
- 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

Reactive systems are

- **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/

# POLYGLOT

```
vertx.createHttpServer()
  .requestHandler(req -> req.response().end("Hello Java"))
  .listen(8080);
```

```
vertx.createHttpServer()
    .requestHandler({ req -> req.response().end("Hello Groovy") })
    .listen(8080)
```

```
vertx.createHttpServer()
    .requestHandler(function (req) {
       req.response().end("Hello JavaScript")
    })
    .listen(8080);
```

```
vertx.createHttpServer()
     .requestHandler(req => req.response().end("Hello Scala"))
     .listen(8080)
```

# MICROSERVICES

## REBRANDING DISTRIBUTED APPLICATIONS

# MICROSERVICES

> The microservice **architectural style** is an approach to developing a single application as **a suite of small services**, each running in its own process and communicating with **lightweight mechanisms**, often an HTTP resource API. These services are built around business capabilities and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies. (Martin Fowler)

redhat.

# A SUITE OF INDEPENDENT SERVICES:

Each service runs in its own process

- So they are **distributed applications**

Lightweight interactions - Loose-coupling

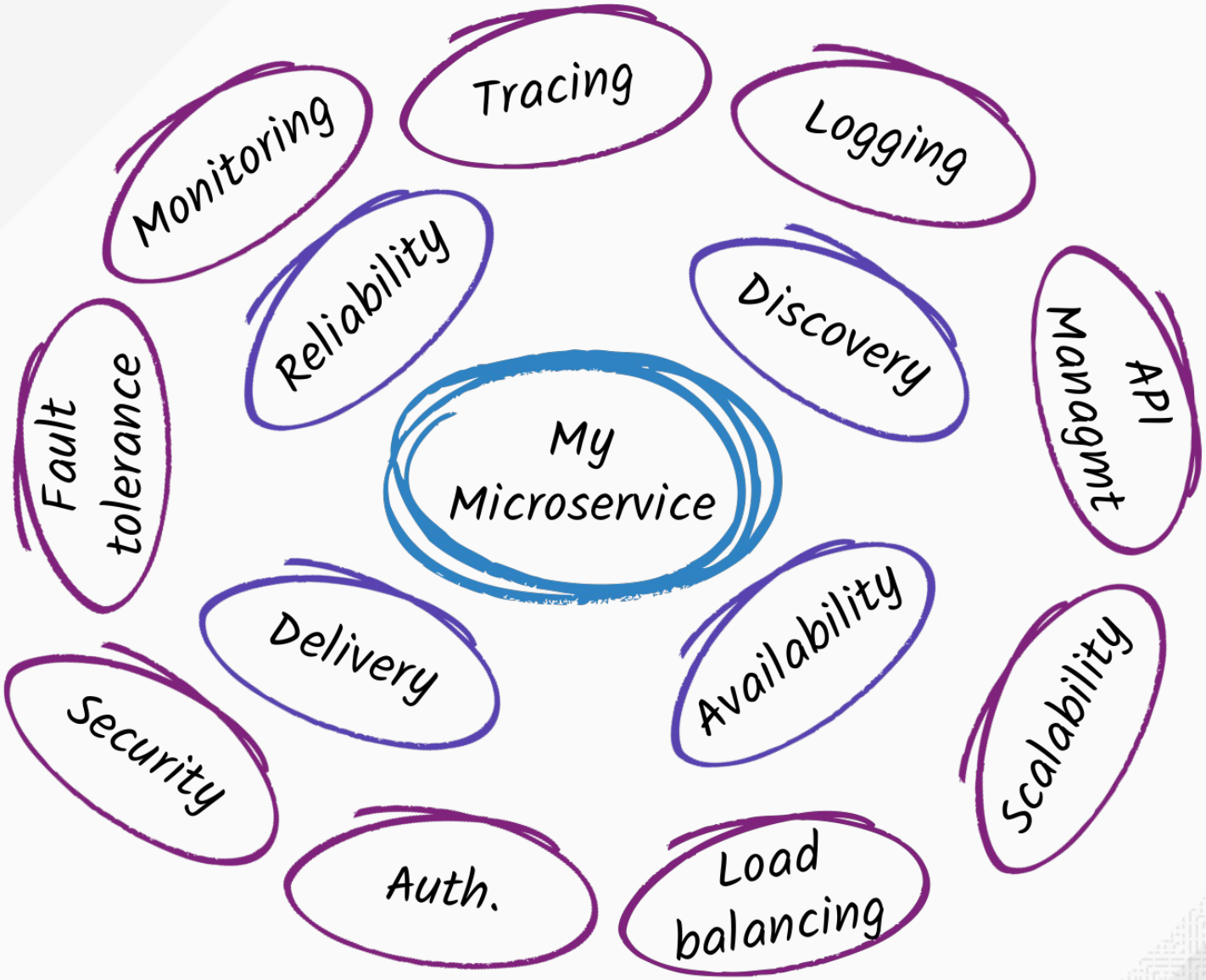
- **Not only HTTP**
- Messaging
- Streams
- (async) RPC

# A SUITE OF INDEPENDENT SERVICES:

Independently developed, tested and deployed

- Automated process
- (Liskov) substitutability
- It's all about **agility**

# NO FREE LUNCH




redhat.

# VERT.X & MICROSERVICES

We wont' build *regular* microservices, but **reactive** 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

# PICK YOUR OWN DEVELOPMENT MODEL

```
vertx.createHttpServer()
   .requestHandler(request -> {
     doA("Austin", ar -> {
       doB("Reactive Summit", ar2 -> {
         request.response().end(ar.result() + "\n" + ar2.result());
       });
     });
   }).listen(8080);
```

# PICK YOUR OWN DEVELOPMENT MODEL

```
vertx.createHttpServer()
   .requestHandler(request -> {
     Future<String> a = doA("Austin");
     Future<String> b = doB("Reactive Summit");
     CompositeFuture.all(a, b).setHandler(ar -> {
       request.response().end(a.result() + "\n" + b.result());
     });
   }).listen(8080);
```

# PICK YOUR OWN DEVELOPMENT MODEL

```
vertx.createHttpServer()
   .requestHandler(request -> {
     CompletableFuture<String> a = doA("Austin");
     CompletableFuture<String> b = doB("Reactive Summit");
     CompletableFuture.allOf(a, b).thenRun(() -> {
       request.response().end(a.join() + "\n" + b.join());
     });
   }).listen(8080);
```

# PICK YOUR OWN DEVELOPMENT MODEL

```
HttpServer server = vertx.createHttpServer();
server.requestStream().toObservable()
    .subscribe(req -> {
      Observable<String> a = doA("Austin");
      Observable<String> b = doB("Reactive Summit");
      Observable.zip(a, b, (s1, s2) -> s1 + "\n" + s2)
          .subscribe(s -> req.response().end(s));
    });
server.listen(8080);
```

# WHAT DOES VERT.X PROVIDE TO BUILD MICROSERVICES?

- TCP, UDP, HTTP 1 & 2 servers and clients
- (non-blocking) DNS client
- Event bus (messaging)
- Distributed data structures
- Load-balancing
- Fail-over
- Service discovery
- Failure management, Circuit-breaker
- Shell, Metrics, Deployment support

# HTTP & REST

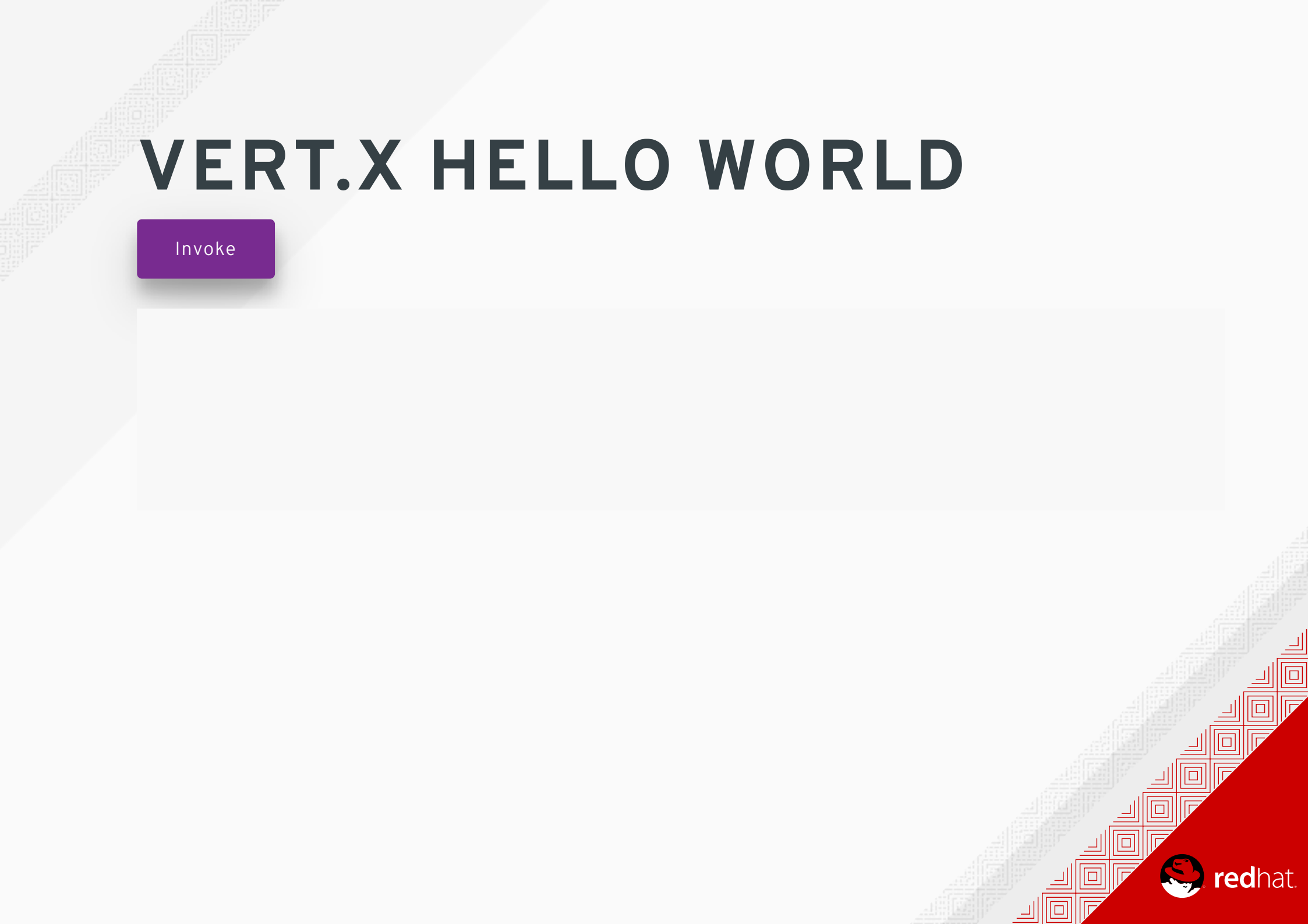
## BECAUSE IT ALWAYS BEGIN WITH A REST

# VERT.X HELLO WORLD

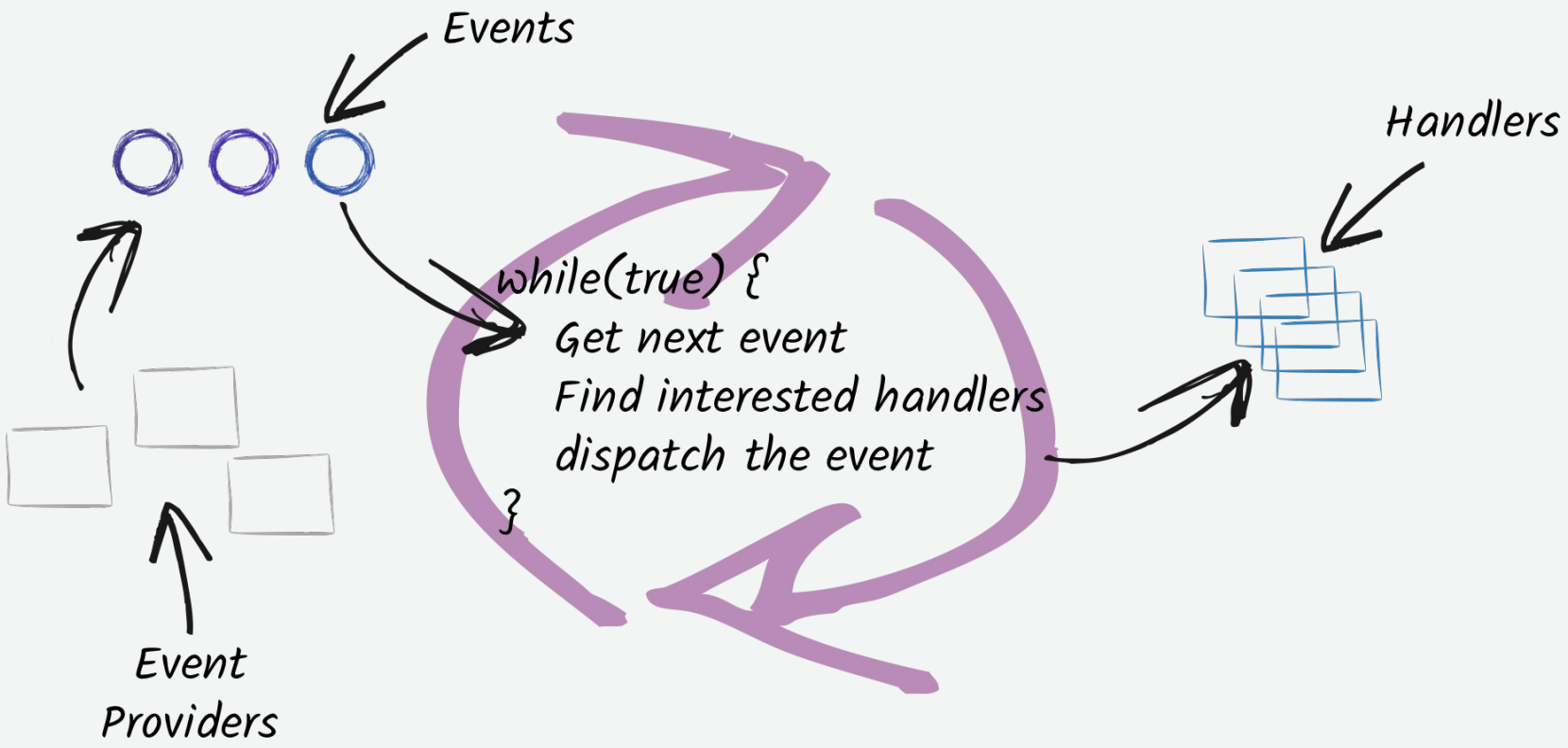
```
vertx.createHttpServer()
  .requestHandler(request -> {
    // Handler receiving requests
    request.response().end("World !");
  })
  .listen(8080, ar -> {
    // Handler receiving start sequence completion (AsyncResult)
    if (ar.succeeded()) {
      System.out.println("Server started on port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

# VERT.X HELLO WORLD

Invoke



# EVENT LOOPS



# VERT.X ASYNC HTTP CLIENT

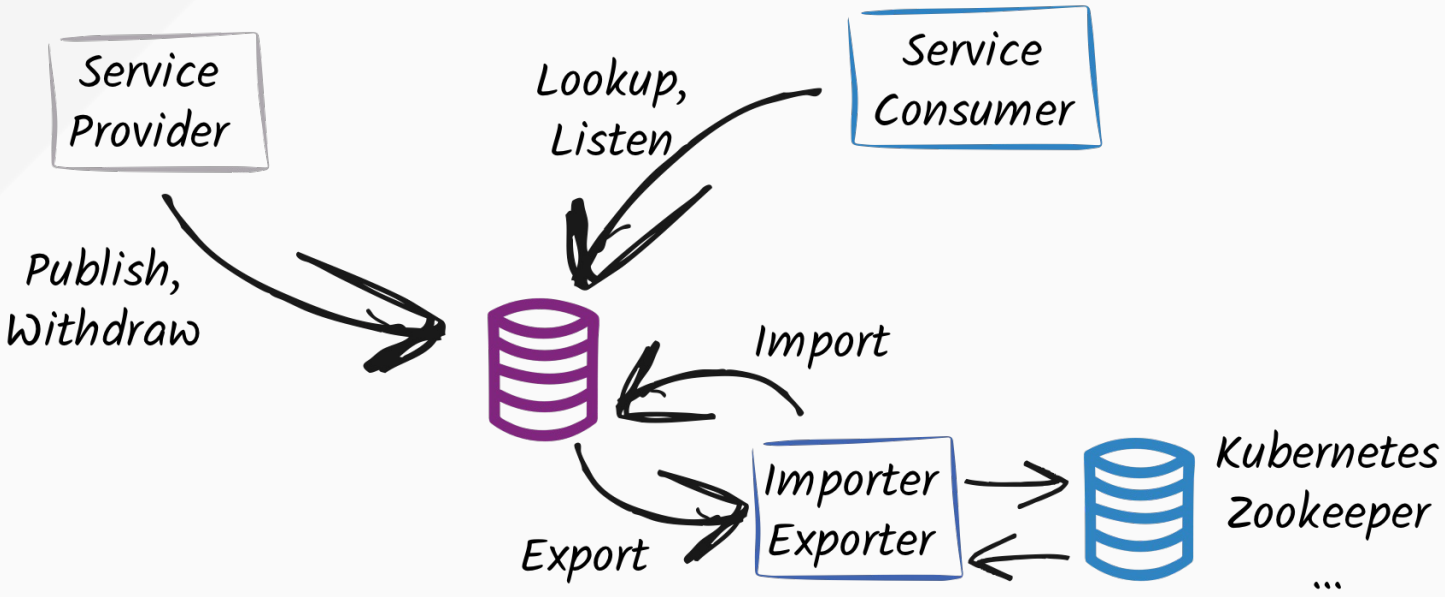
```
HttpClient client = vertx.createHttpClient(
        new HttpClientOptions()
                .setDefaultHost(host)
                .setDefaultPort(port));

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

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

# SERVICE DISCOVERY

Locate the services, environment-agnostic

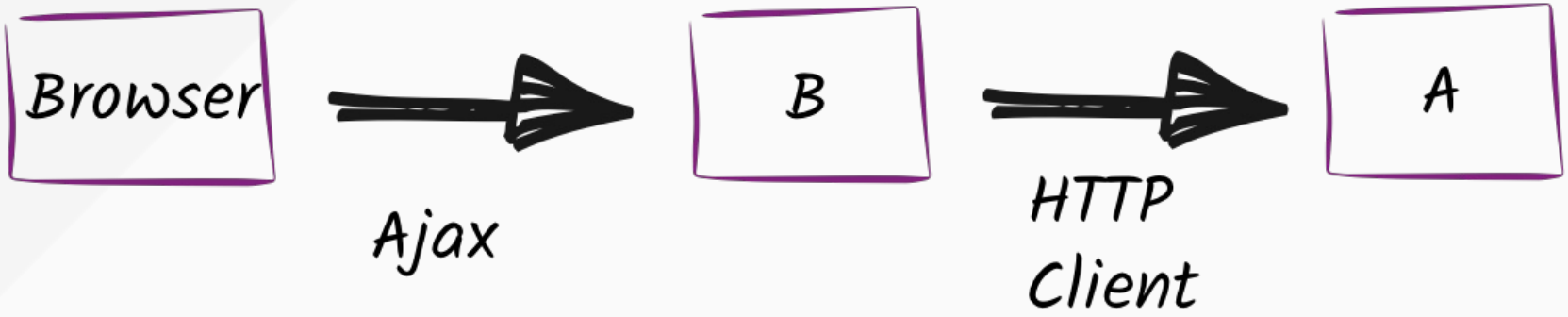


# SERVICE DISCOVERY - KUBERNETES

No need for publication - Import Kubernetes services

```
HttpEndpoint.getClient(discovery,
    new JsonObject().put("name", "vertx-http-server"),
    result -> {
      if (result.failed()) {
        rc.response().end("D'oh no matching service");
        return;
      }
      HttpClient client = result.result();
      client.getNow("/", response -> {
        response.bodyHandler(buffer -> {
          rc.response().end("Hello " + buffer.toString());
        });
      });
    });
```

# CHAINED HTTP REQUESTS




Invoke

# MESSAGING WITH THE EVENT BUS

## THE SPINE OF VERT.X APPLICATIONS

redhat.

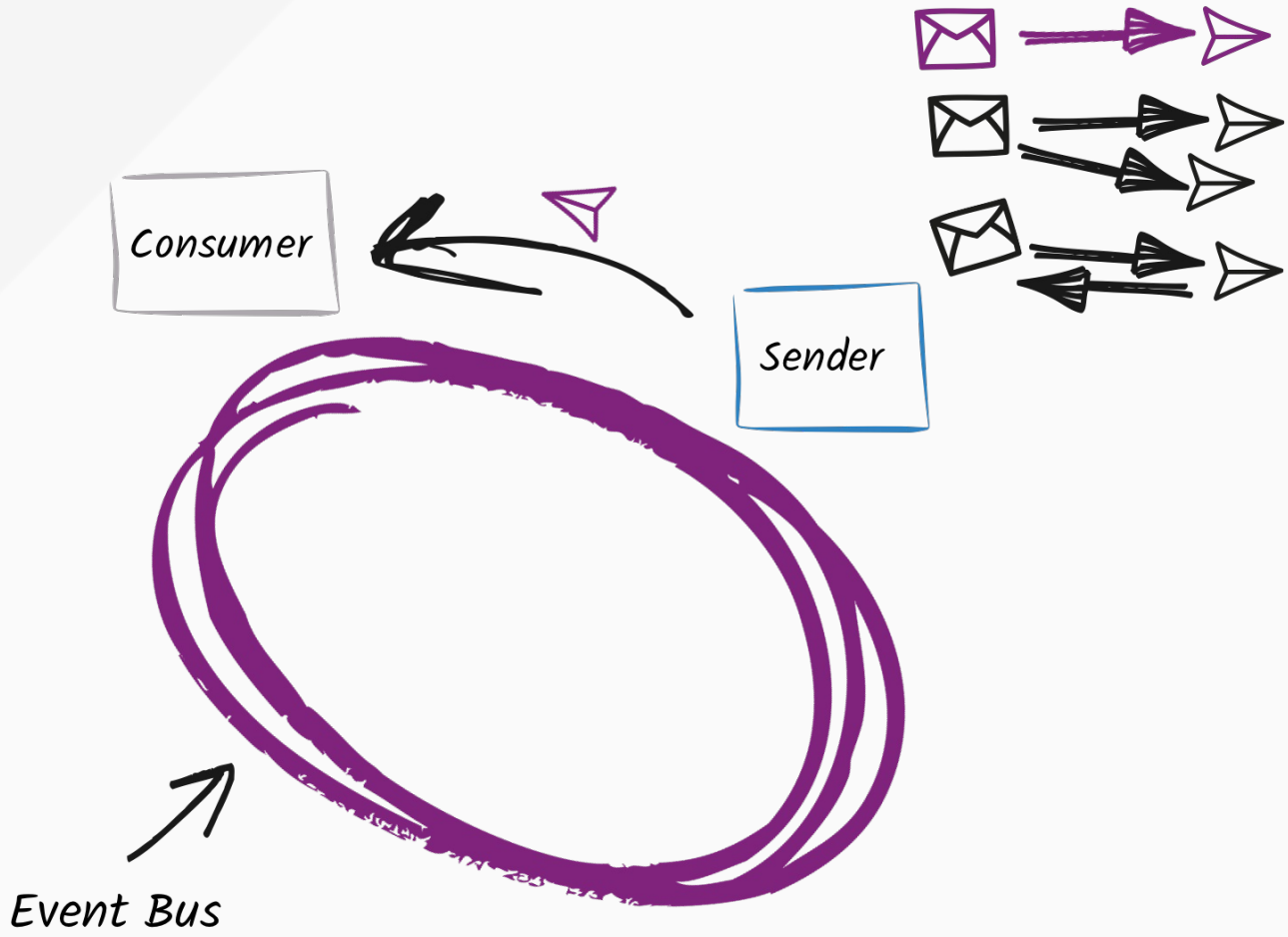
# THE EVENT BUS

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)

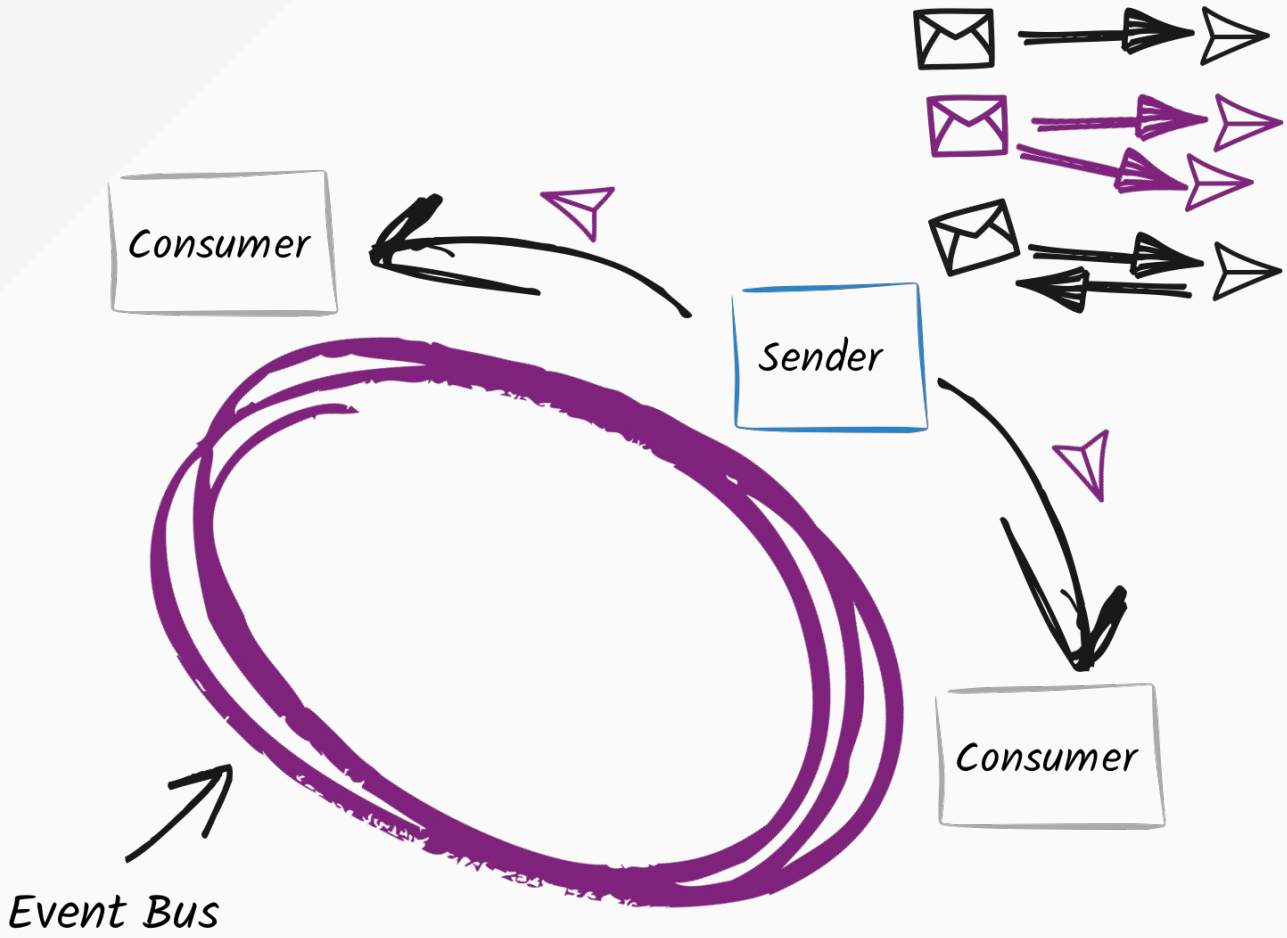
- **Address**: Messages are sent to an address
- **Handler**: Messages are received in handlers.

# POINT TO POINT



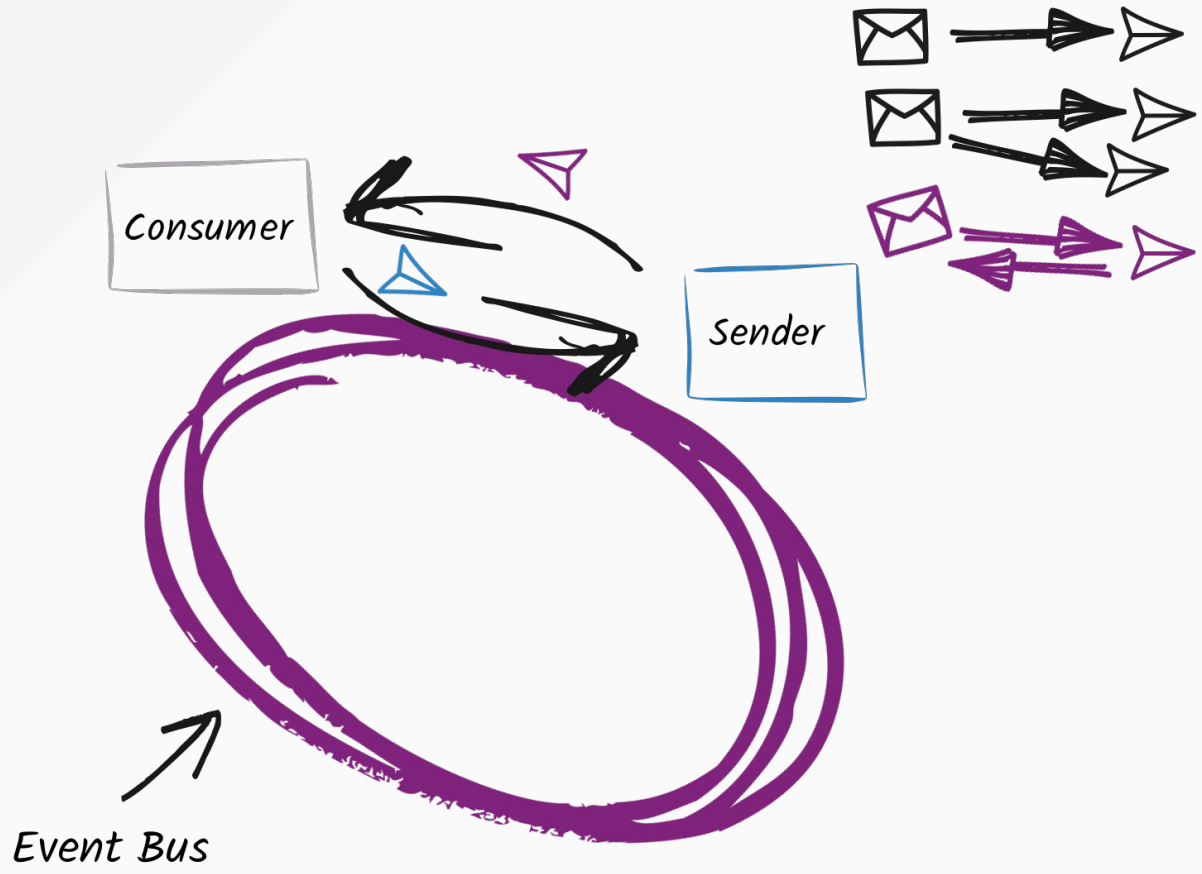

```
vertx.eventBus().send("address", "message");
vertx.eventBus().consumer("address", message -> {});
```

# PUBLISH / SUBSCRIBE




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

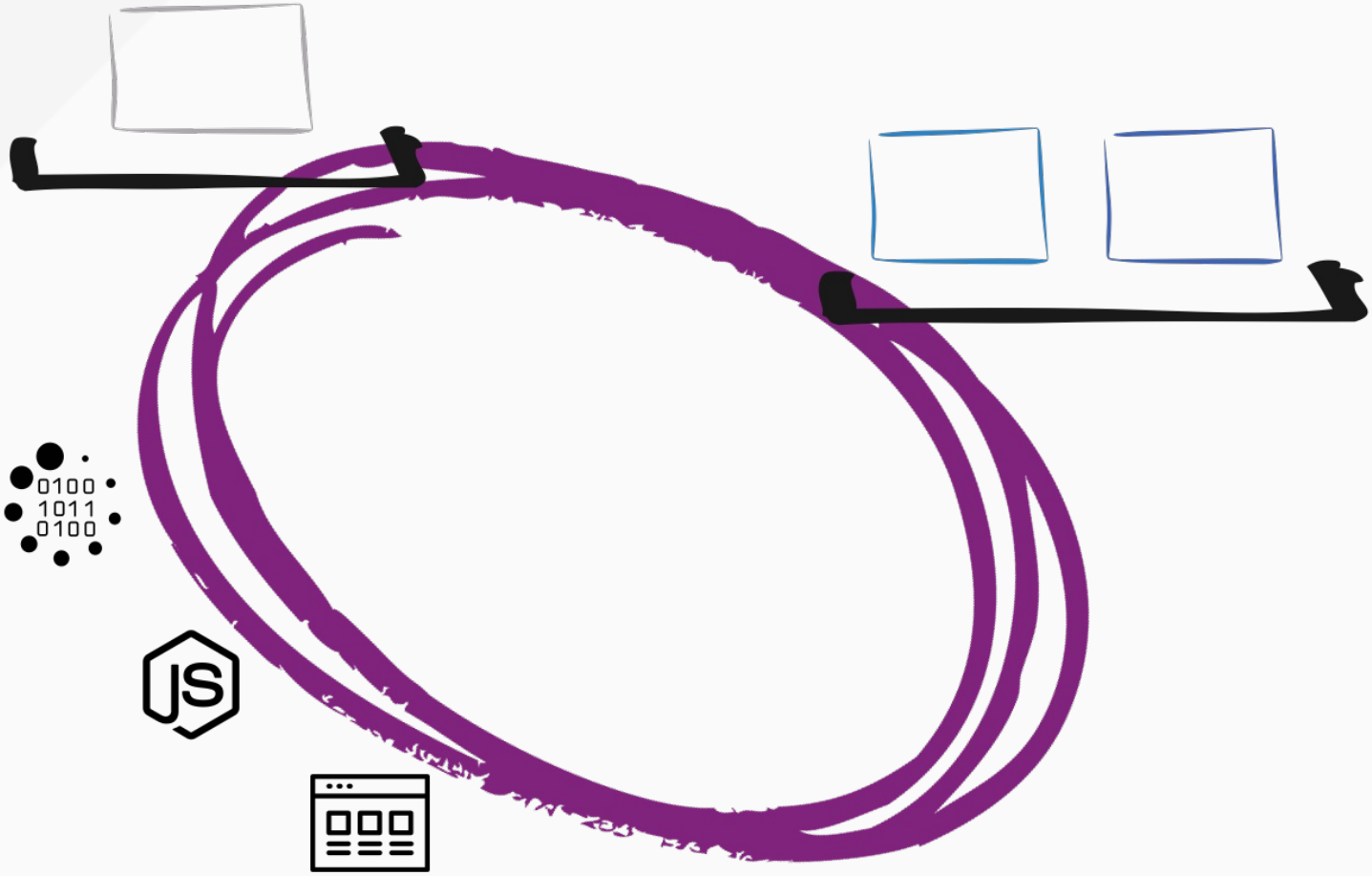
# REQUEST / RESPONSE




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

# DISTRIBUTED EVENT BUS

Almost anything can send and receive messages



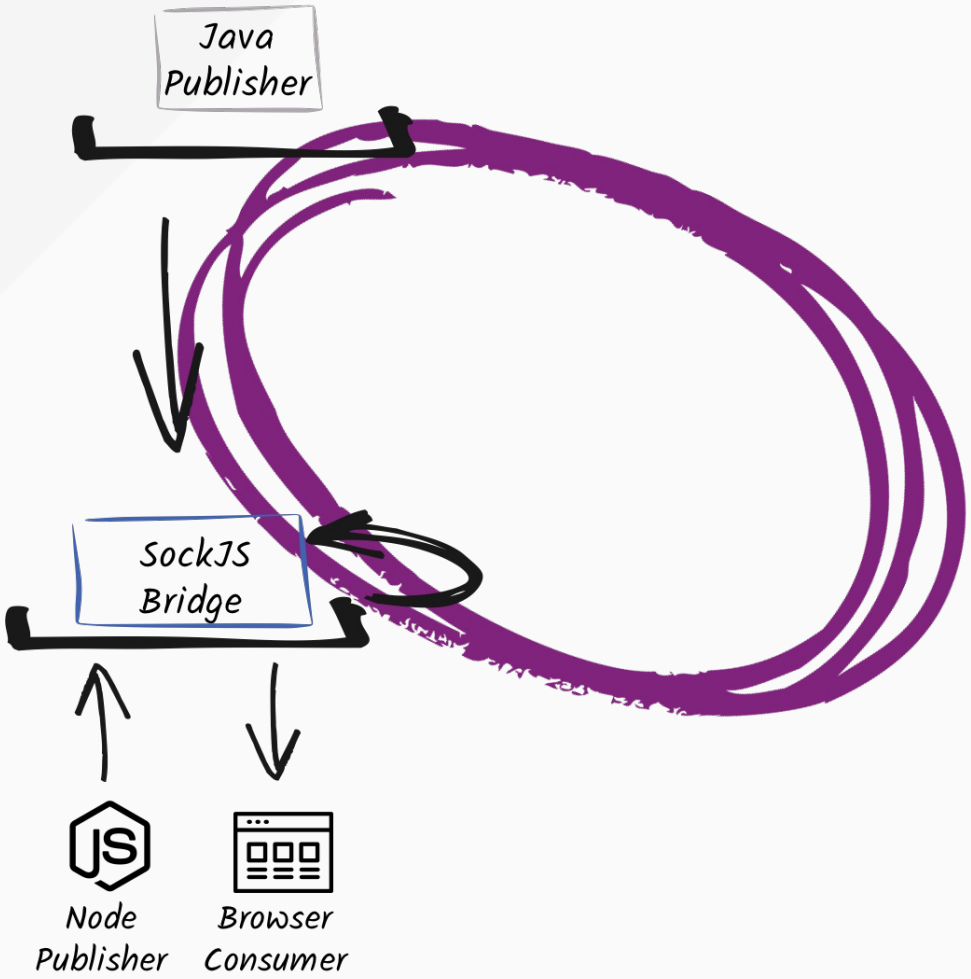

redhat.

# DISTRIBUTED EVENT BUS

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

```
bonjour from node (248)
hello from java (286)
bonjour from node (247)
hello from java (285)
bonjour from node (246)
```

# 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, Swift...

# RELIABILITY PATTERNS

## DON'T BE FOOL, BE PREPARED TO **FAIL**

# RELIABILITY

It's not about being bug-free or bullet proof,
it's **impossible**.

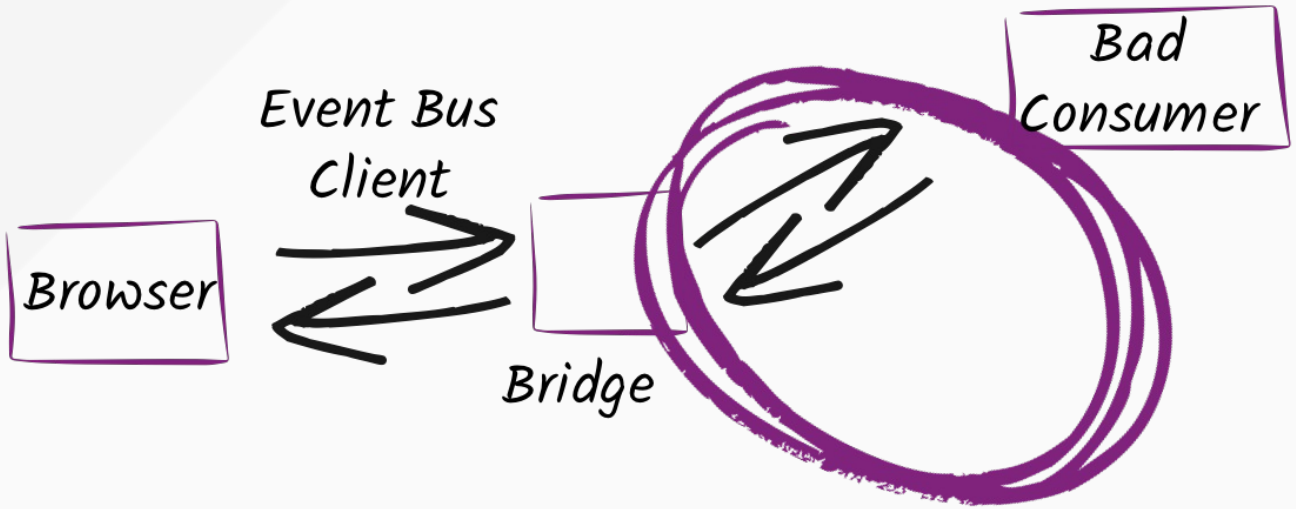
It's about being prepared to fail,
and handling these **failures**.

# MANAGING FAILURES

## Distributed communication may fail

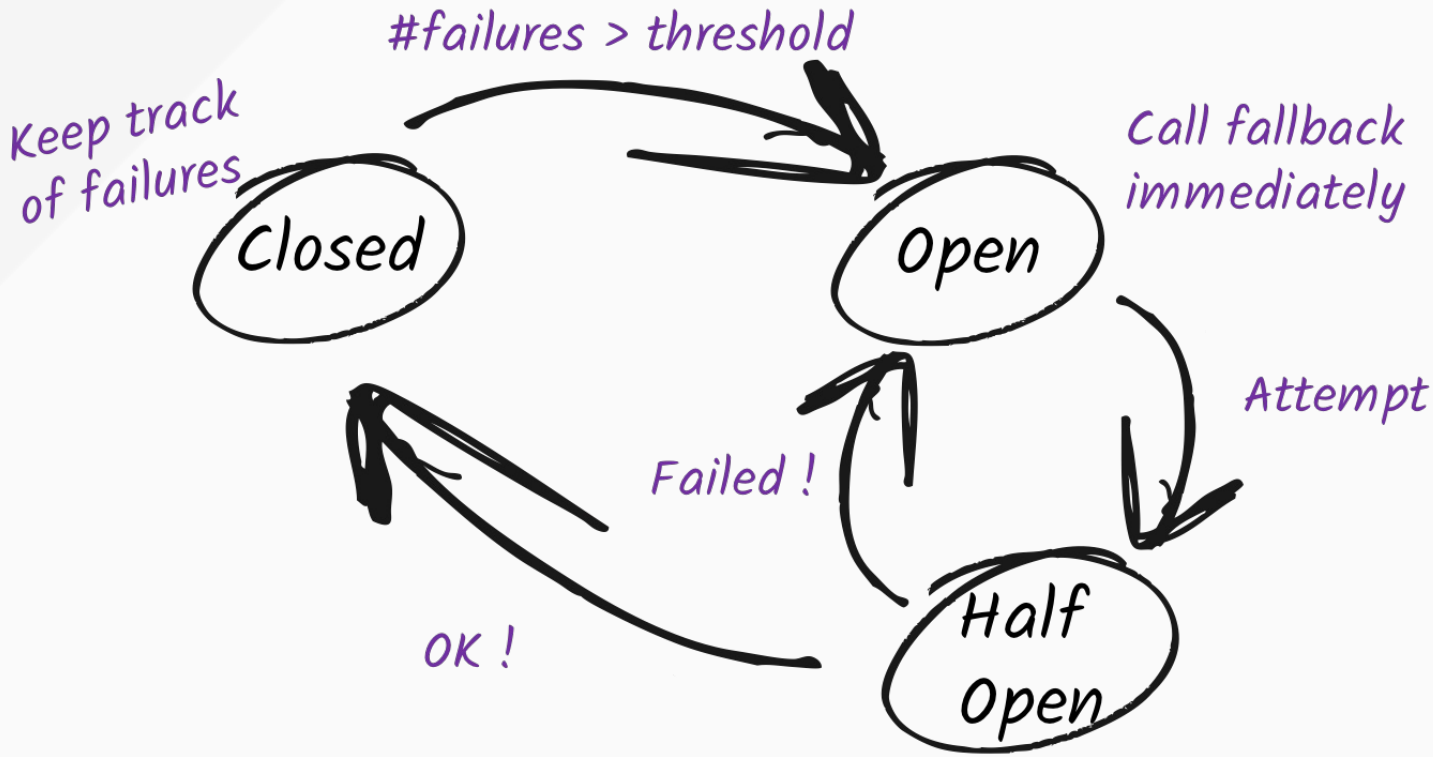
```
vertx.eventbus().send(..., ...,
  new DeliveryOptions().setSendTimeout(1000),
  reply -> {
    if (reply.failed()) {
      System.out.println("D'oh, he did not reply to me !");
    } else {
      System.out.println("Got a mail " + reply.result().body());
    }
});
```

# MANAGING FAILURES




Invoke

# CIRCUIT BREAKER

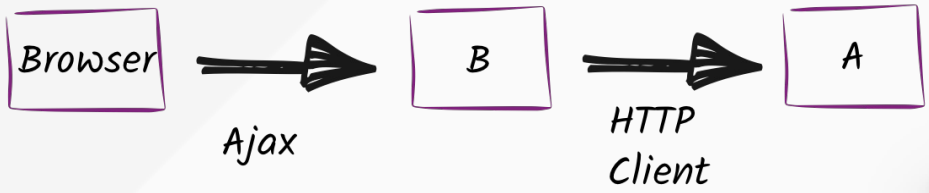


# CIRCUIT BREAKER

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

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

# CIRCUIT BREAKER

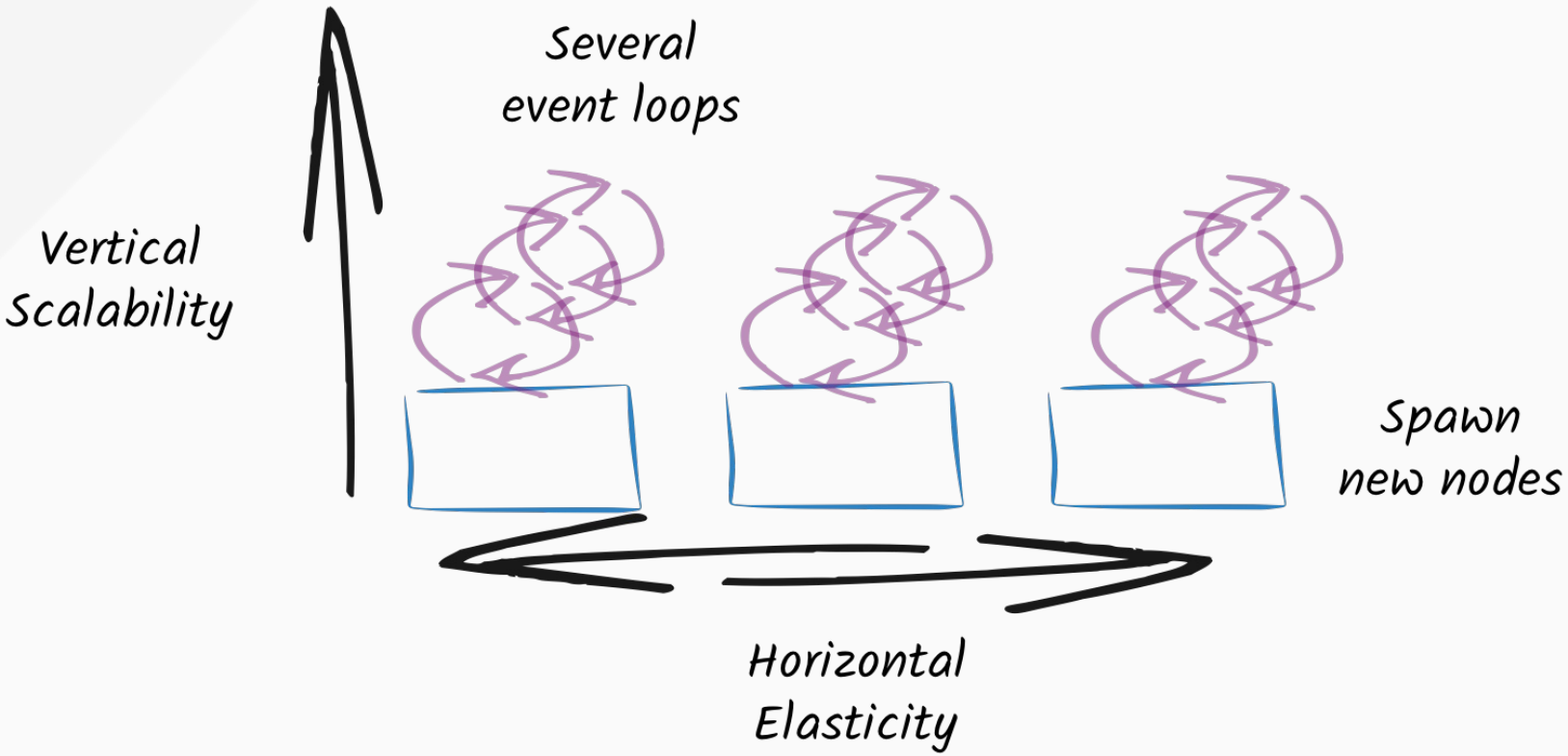



Invoke

# SCALABILITY PATTERNS

## BE PREPARED TO BE FAMOUS

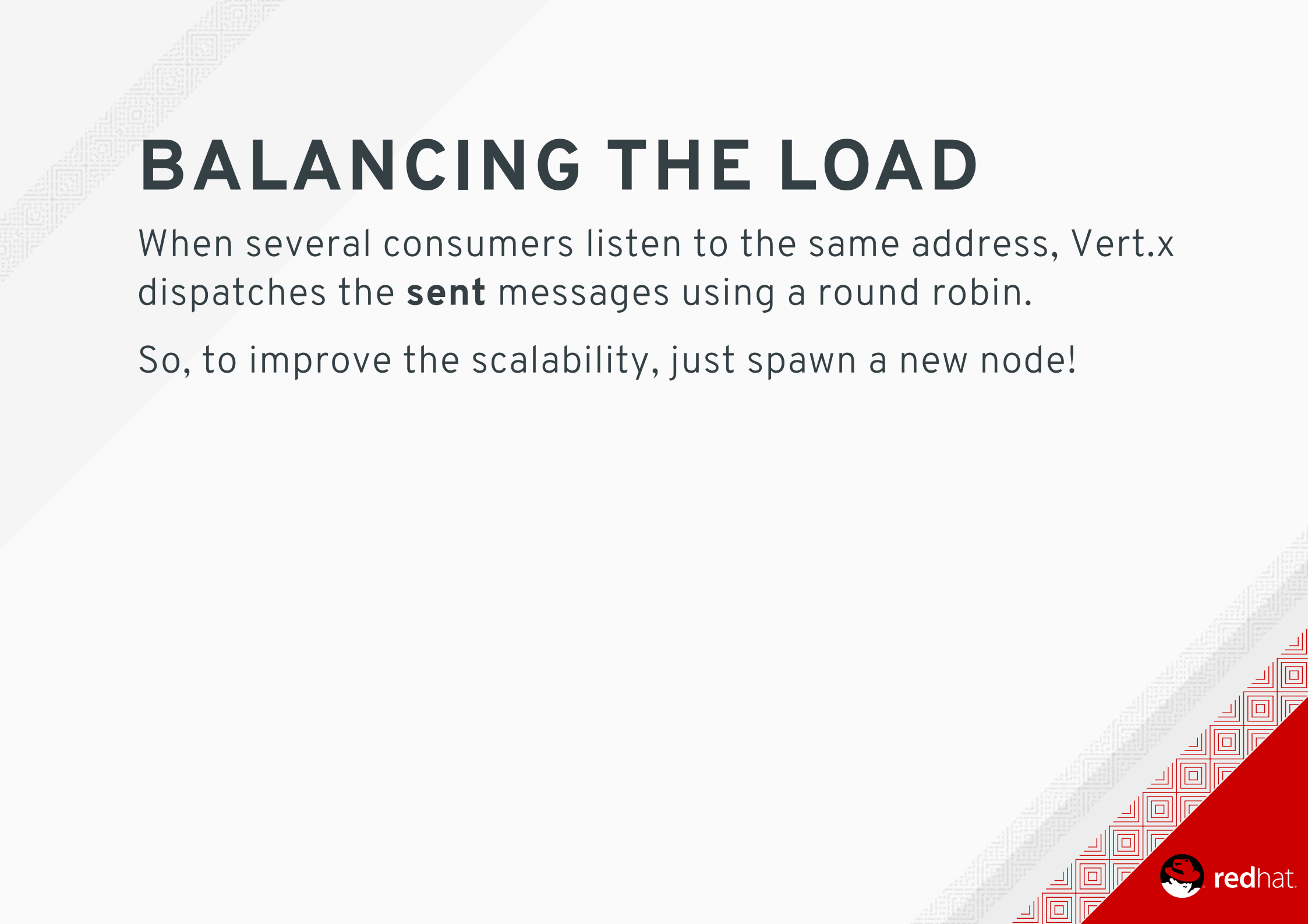
# ELASTICITY PATTERNS



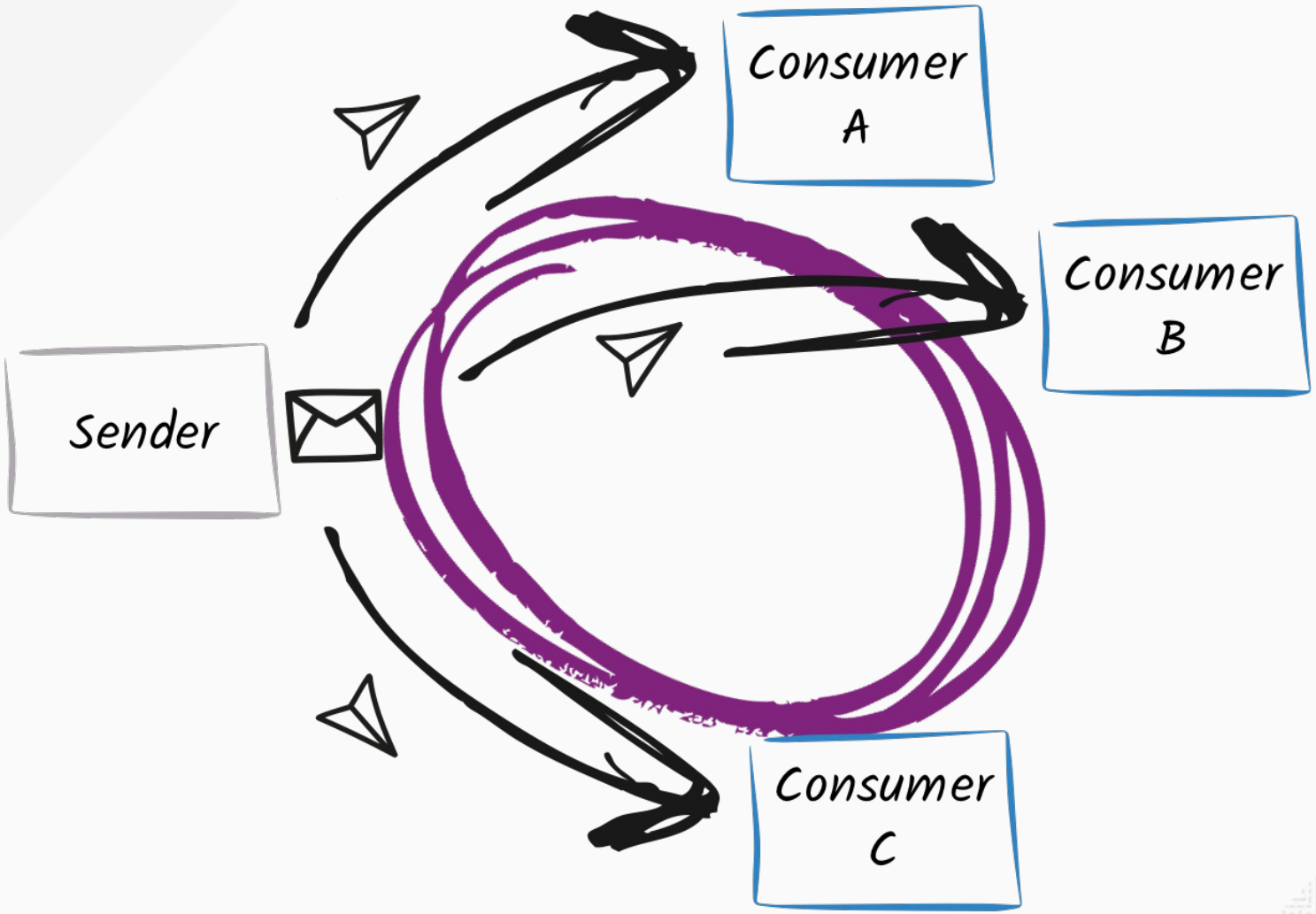
# BALANCING THE LOAD

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!



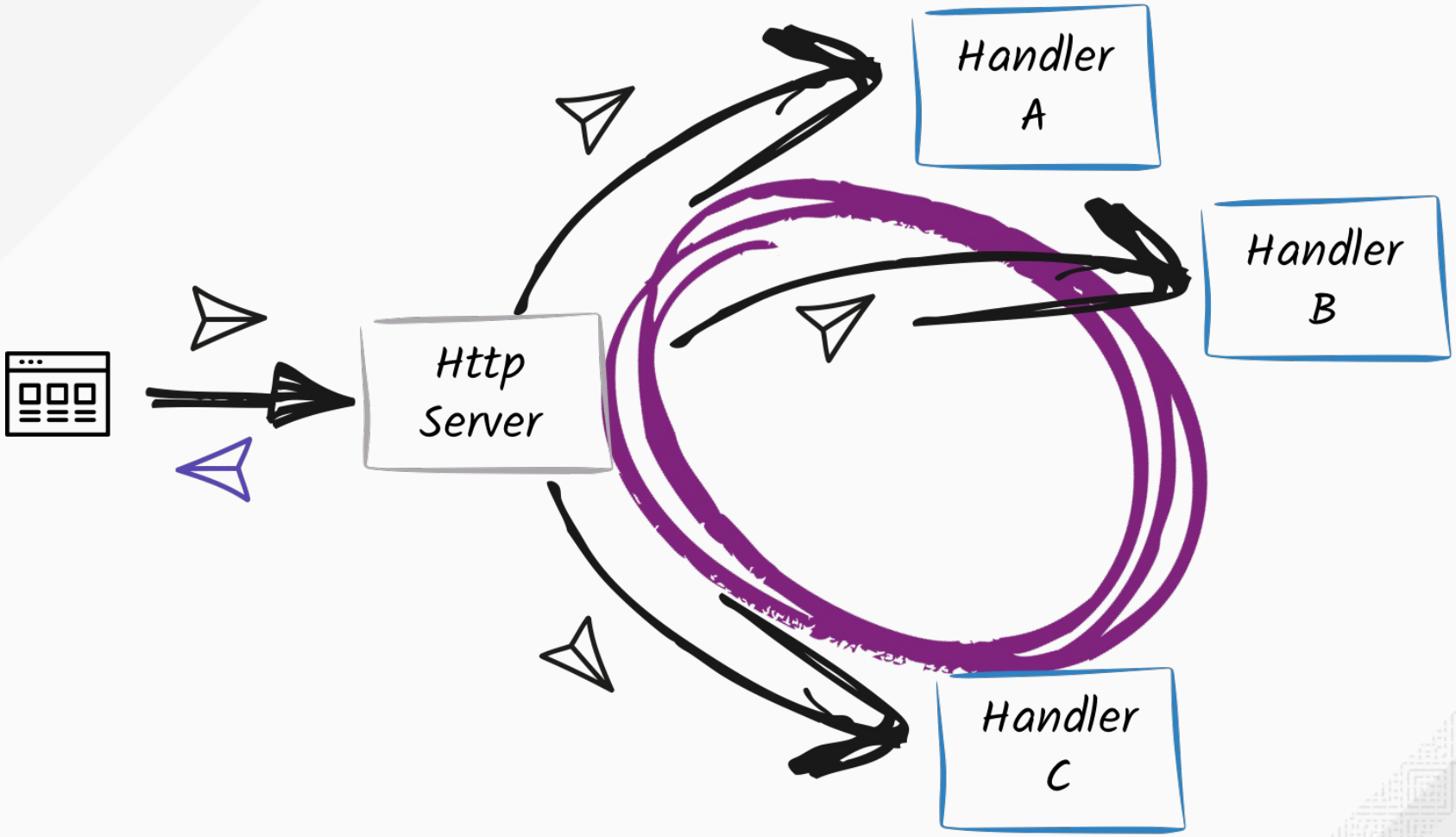
# BALANCING THE LOAD



# BALANCING THE LOAD

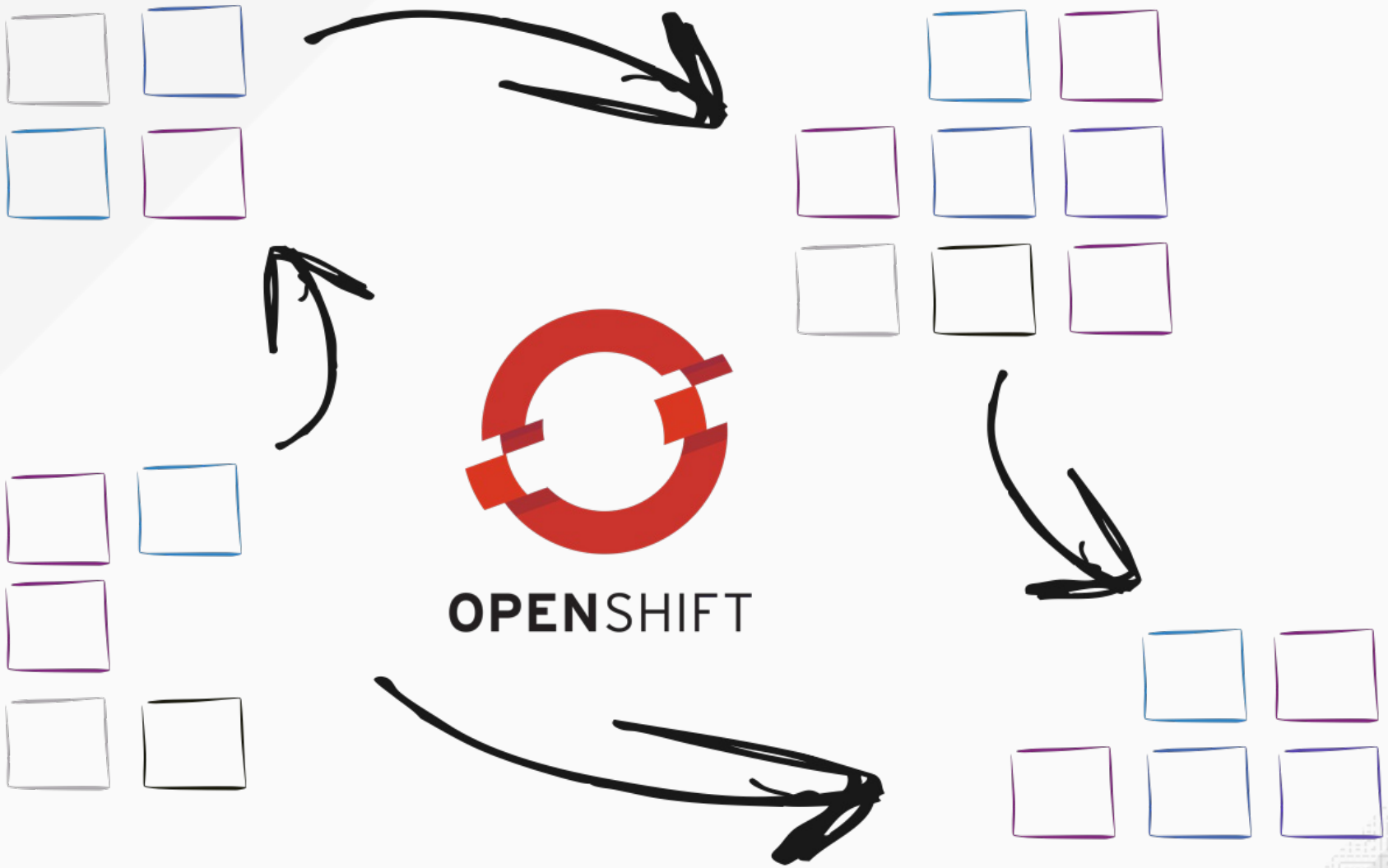
Invoke

# SCALING HTTP

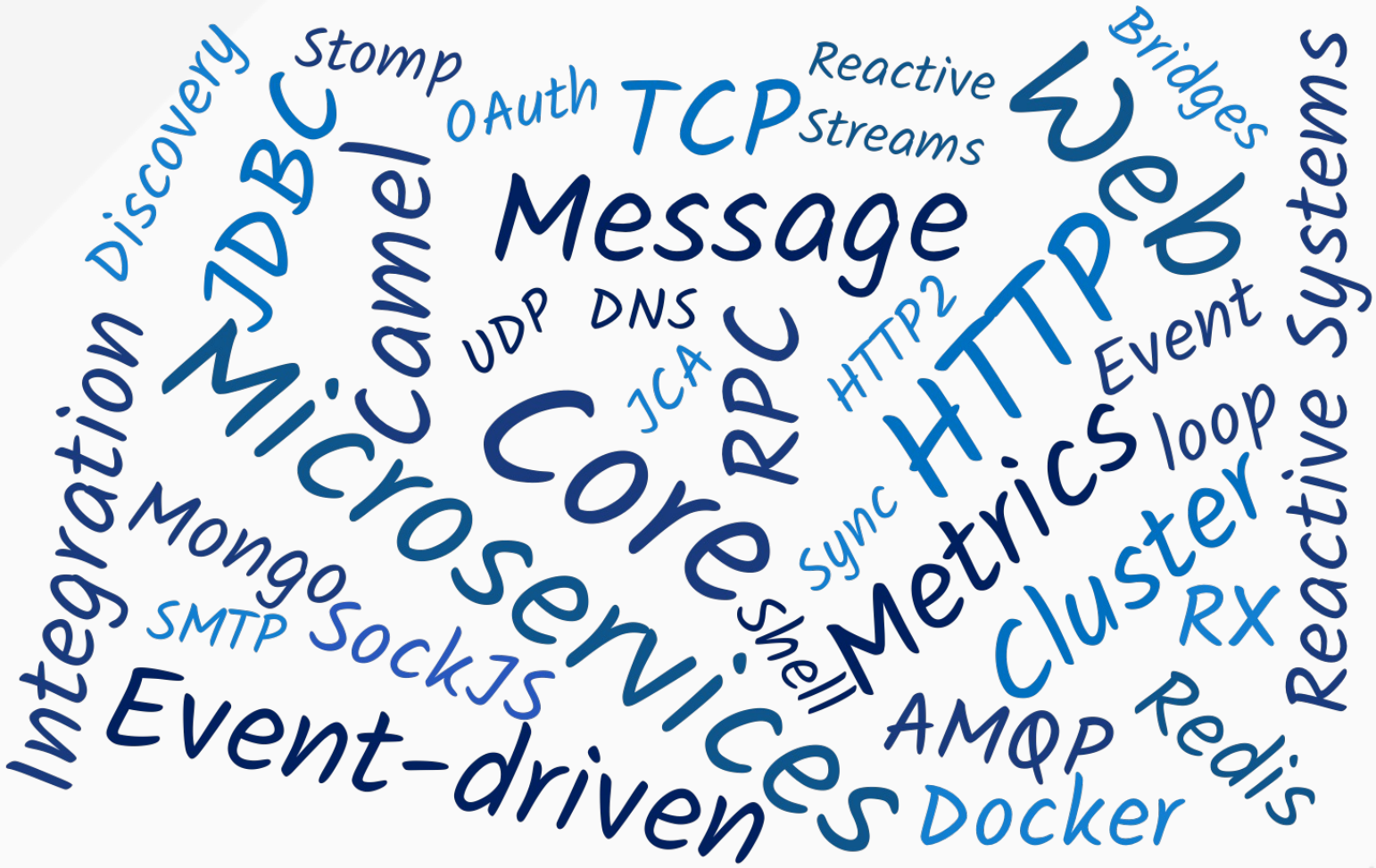


# THIS IS NOT THE END()

## BUT THE FIRST STEP ON THE REACTIVE MICROSERVICE ROAD




redhat.


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

# HOW TO START ?

- https://www.openshift.org

- http://vertx.io
- http://vertx.io/blog/posts/introduction-to-vertx.html
- http://escoffier.me/vertx-hol

- Secure Enclaves For Reactive Cloud Applications

Vote for us !


# THANK YOU!


clement.escoffier@redhat.com

@clementplop


redhat.