redhat

VERT.X

MICROSERVICES WERE NEVER SO EASY

CLEMENT ESCOFFIER

Vert.x Core Developer, Red Hat

redhat

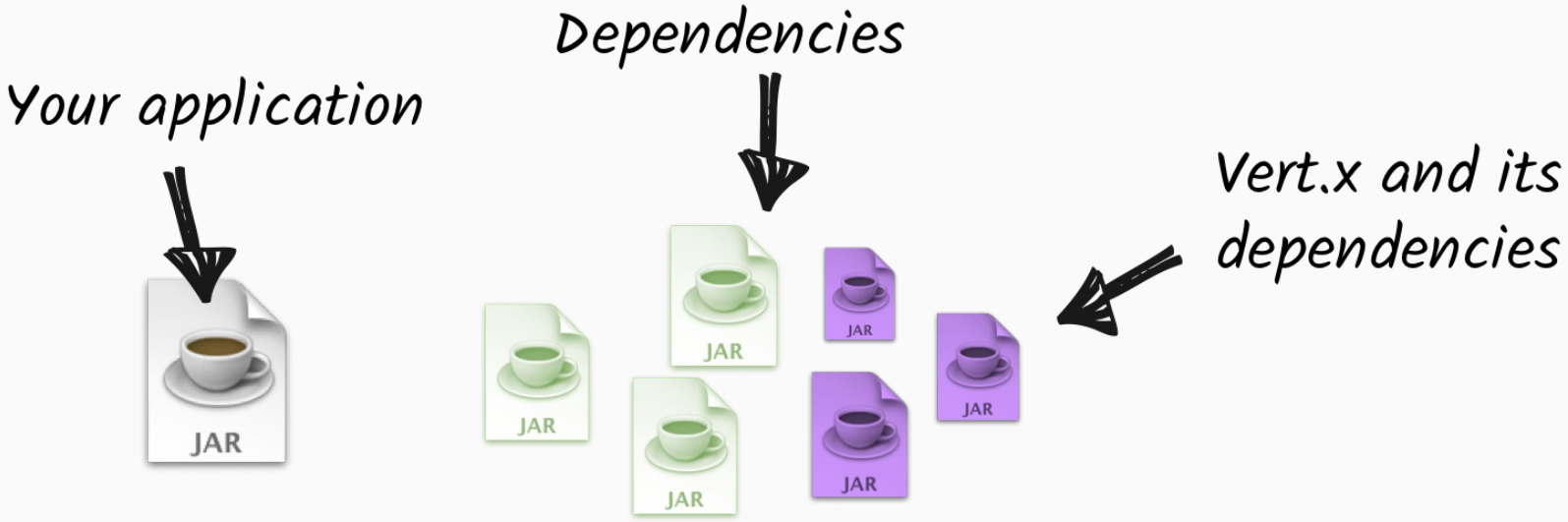
VERT.X

IN 10 SLIDES...

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*, ...)



TOOLKIT

These slides are served by a **vert.x** application:

- executed on Openshift
- packaged as a *fat jar*

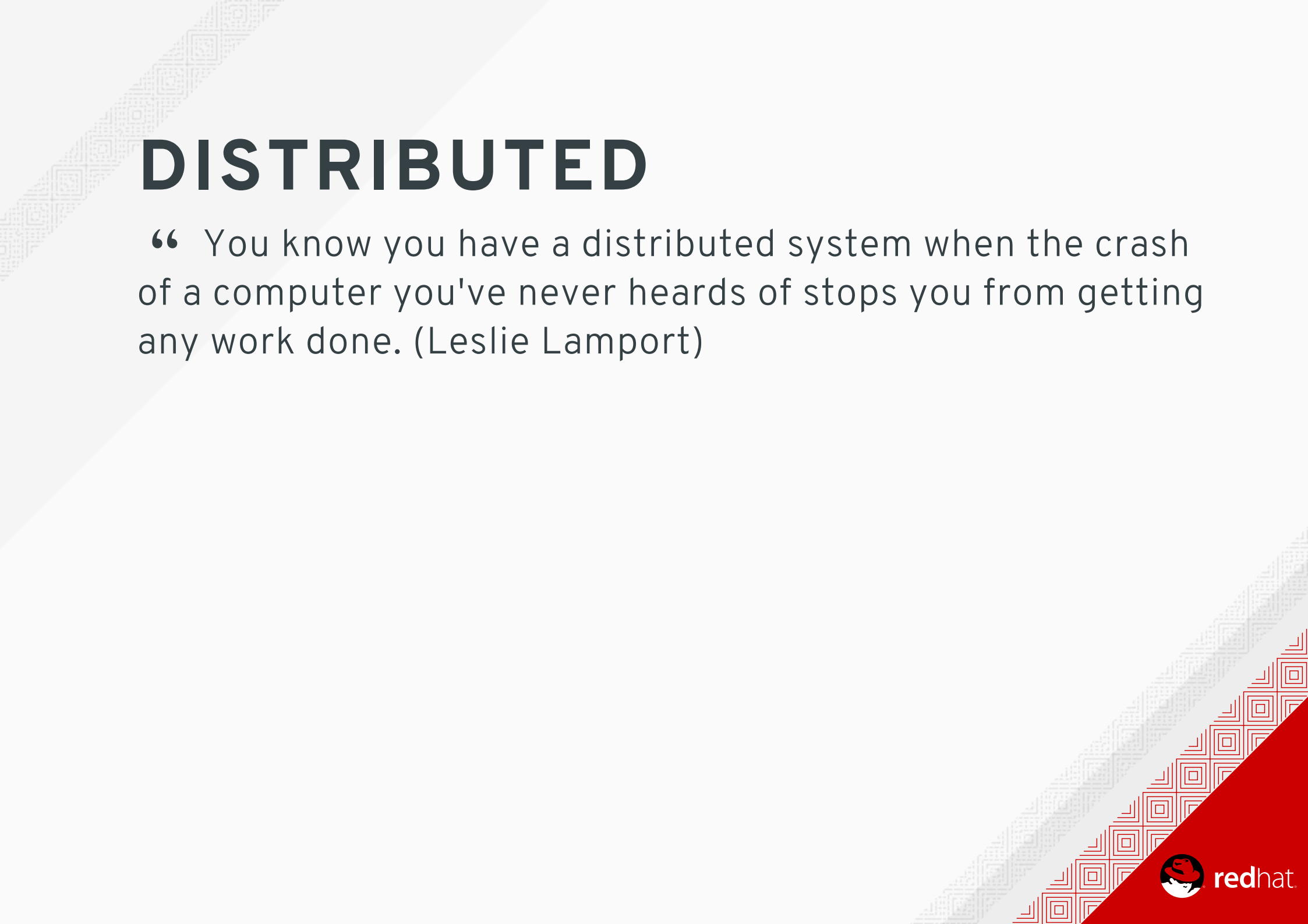
- **vertx-core**: the main vert.x component
- **vertx-web**: a component to build modern web applications
- **vertx-hazelcast**: an implementation of the vert.x cluster manager
- **vertx-service-discovery-bridge-kubernetes**: an extension to interact with Kubernetes

DISTRIBUTED

redhat

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/

REACTIVE

> "Moore's law" is the observation that, over the history of computing hardware, the number of transistors in a dense integrated circuit has doubled approximately every 2 years.

Unfortunately, no free lunch anymore...

REACTIVE

CPU manufacturers cannot get a single-core CPU to go any faster

- Multi-core Processors
- Require parallel execution
- Require a different way to design, develop and execute software
- Reactive systems is one solution to this issue

POLYGLOT

Vert.x applications can be developed using

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

MICROSERVICES

REBRANDING DISTRIBUTED APPLICATIONS

redhat.

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**

- Agility of the composition
- You can replace any microservice
- You can decide who uses who

SORRY... NO FREE LUNCH

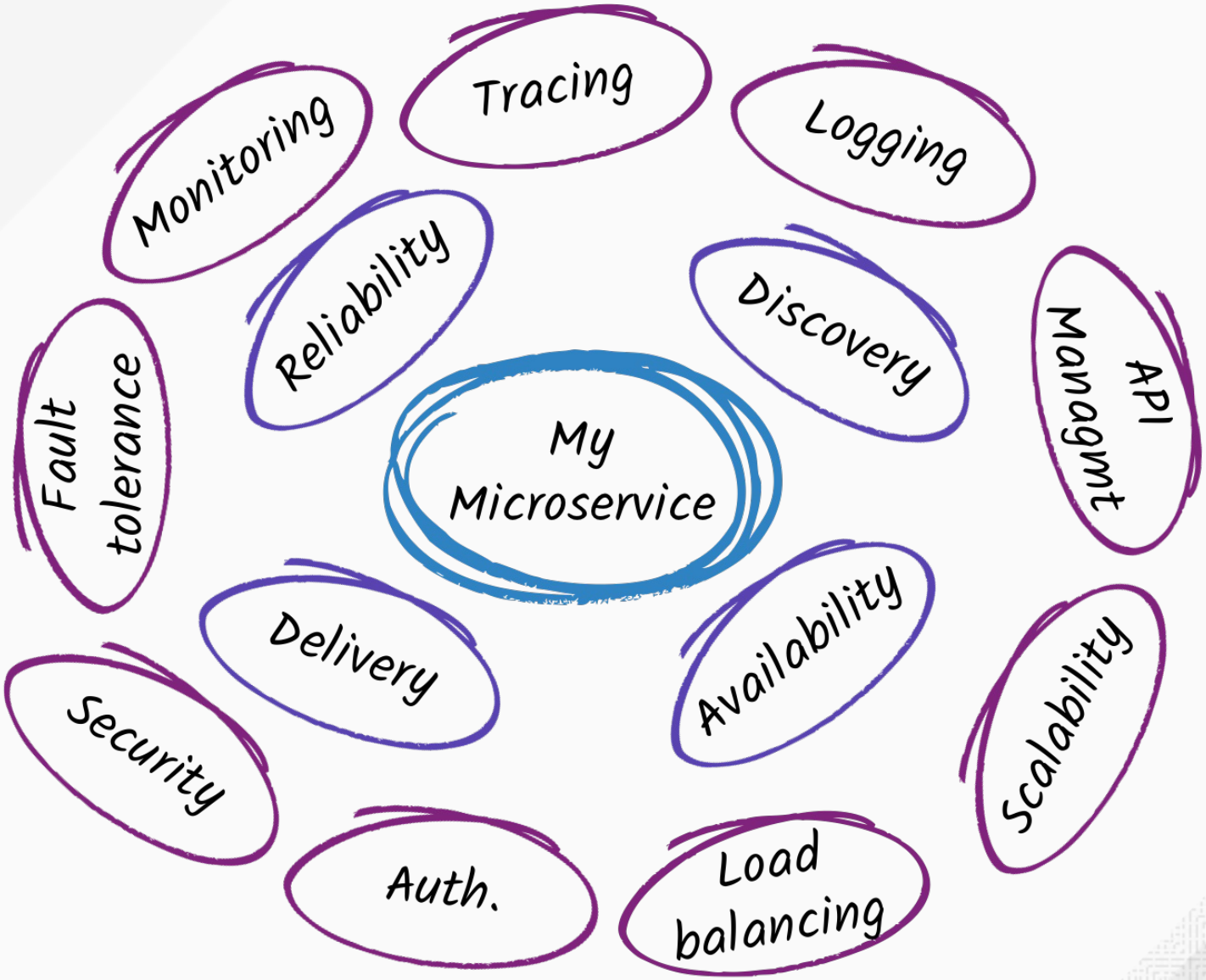
Distributed applications are hard

- **fail** (fail-stop, byzantine fault)
- Discoverability issue
- Reliability issue
- Availability issue

How to keep *things* on track

- Monitoring
- Health
- Tracing
- ...

COMPLEXITY GROWTH



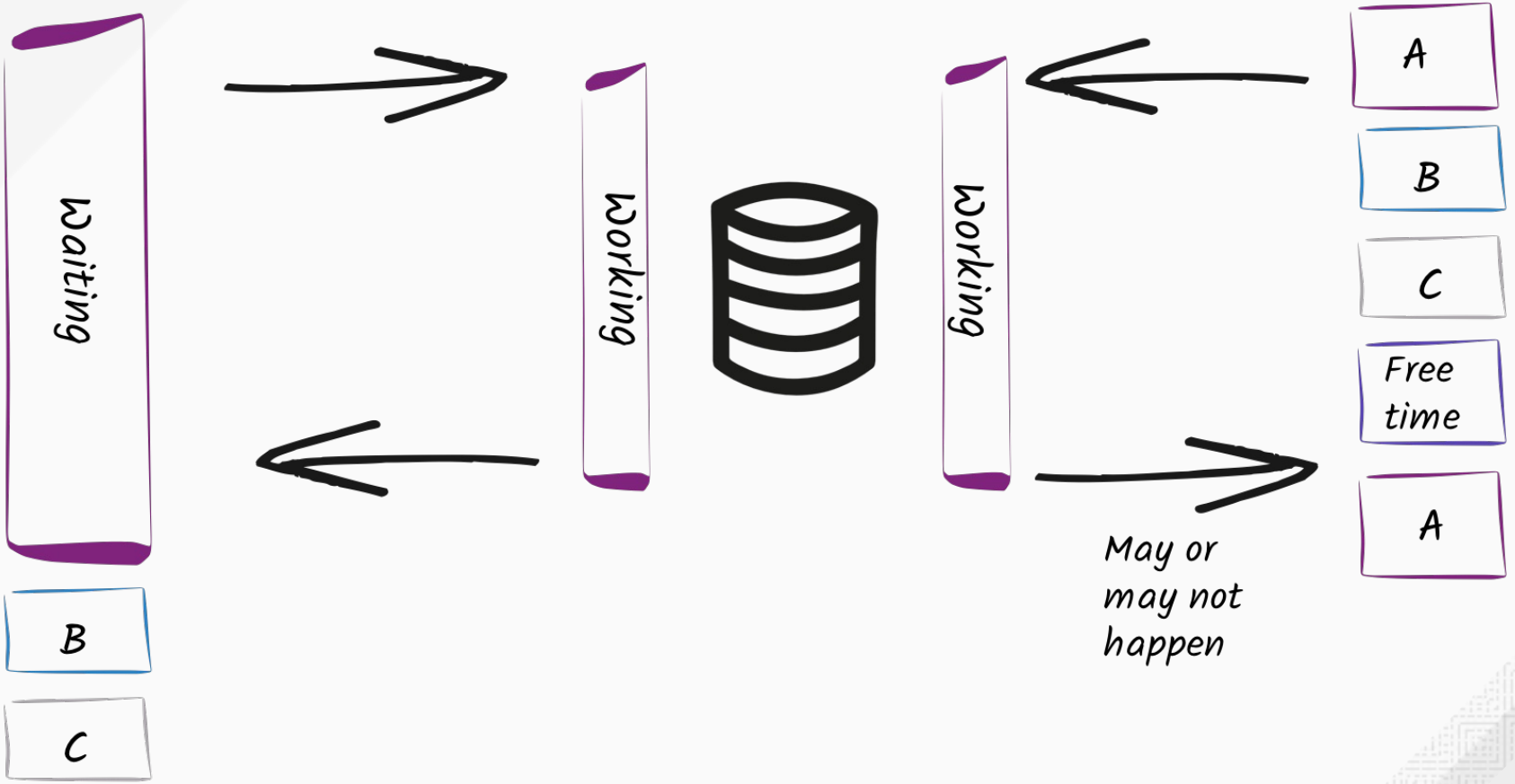
redhat.

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

ASYNCHRONOUS & NON-BLOCKING



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

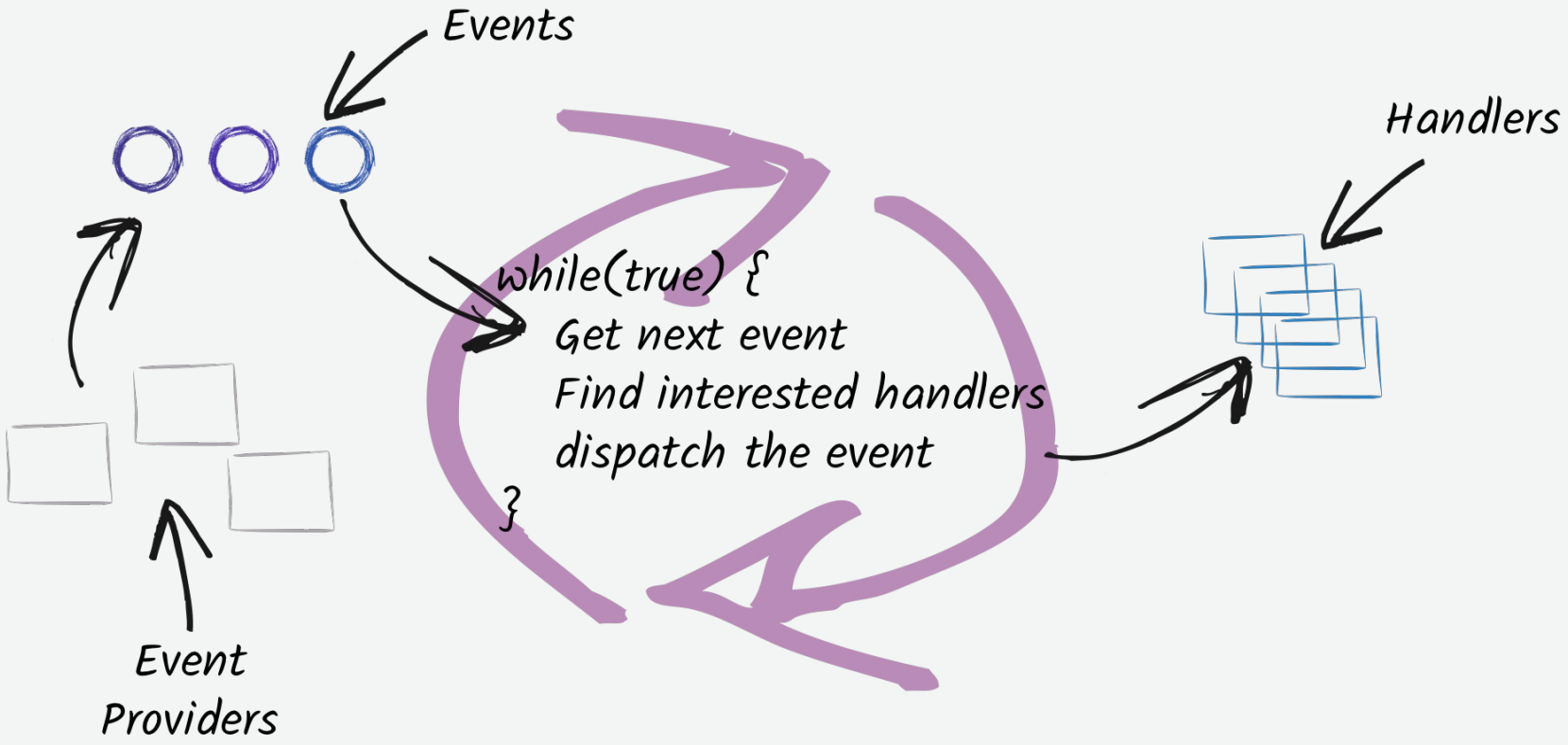
VERT.X HELLO WORLD

```
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

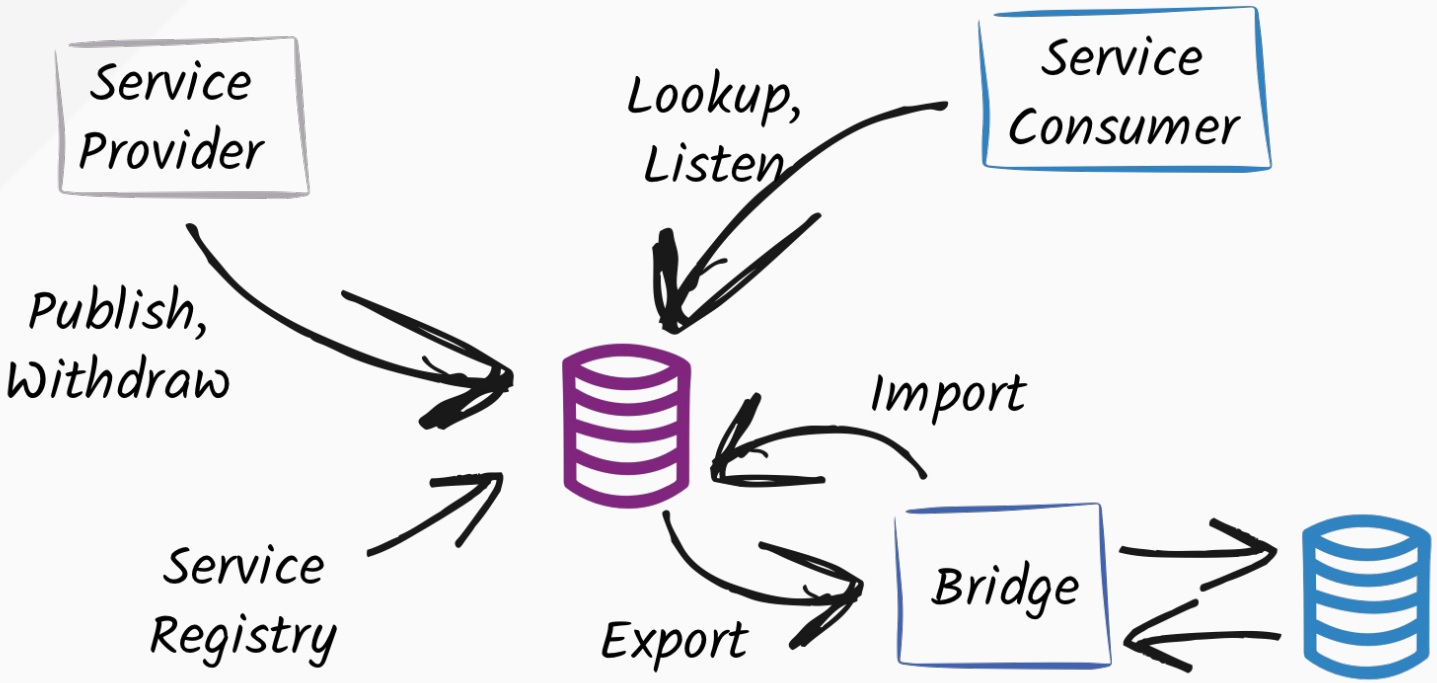
```
HttpClient client = vertx.createHttpClient(
        new HttpClientOptions()
                .setDefaultHost(host)
                .setDefaultPort(port));

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

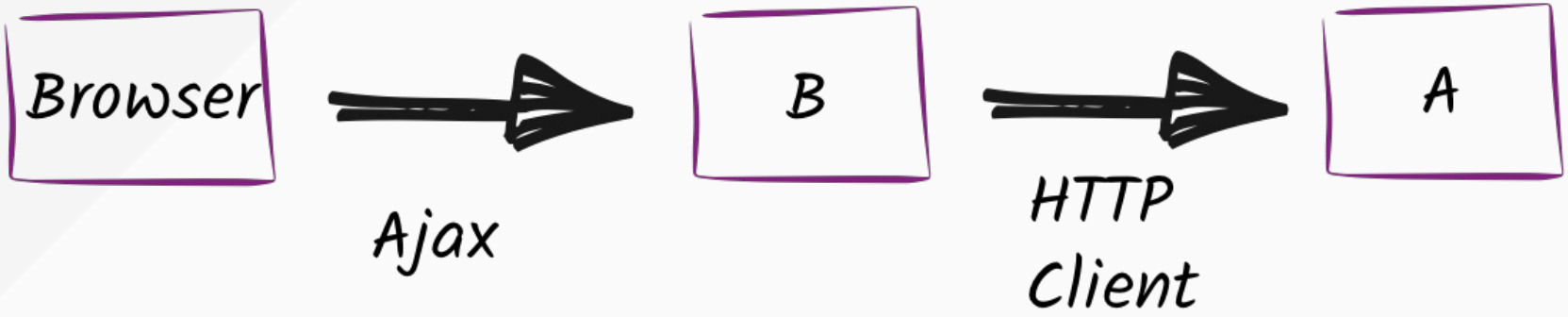


SERVICE DISCOVERY - KUBERNETES

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

INTERACTING WITH BLOCKING SYSTEMS

```
vertx.executeBlocking(
  future -> {
     // Executed using a worker thread
  },
  res (O) -> {
    // Executed in the event loop thread
 }
);
```

MESSAGING WITH THE EVENT BUS

THE SPINE OF VERT.X APPLICATIONS

redhat.

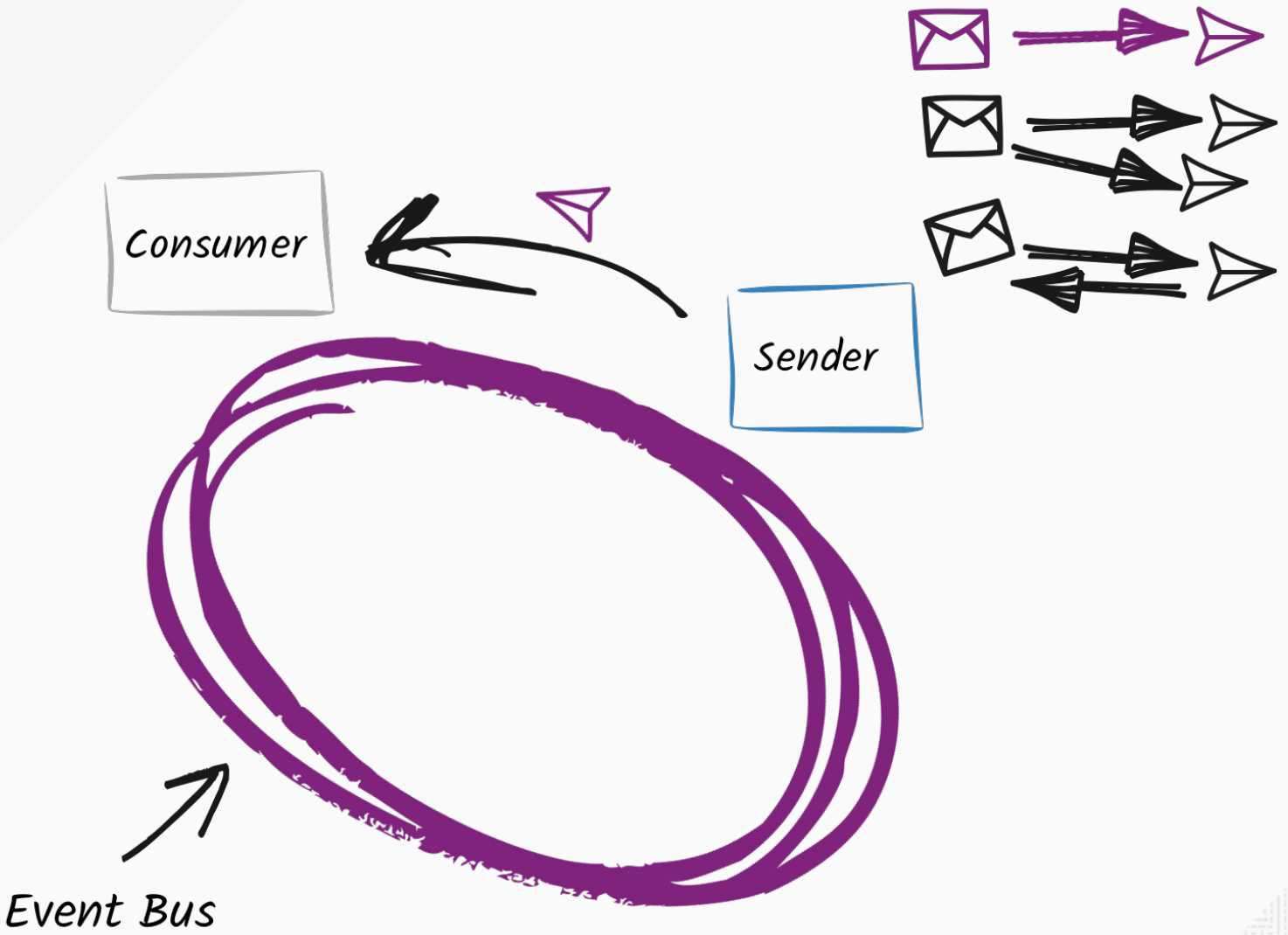
THE EVENT BUS

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

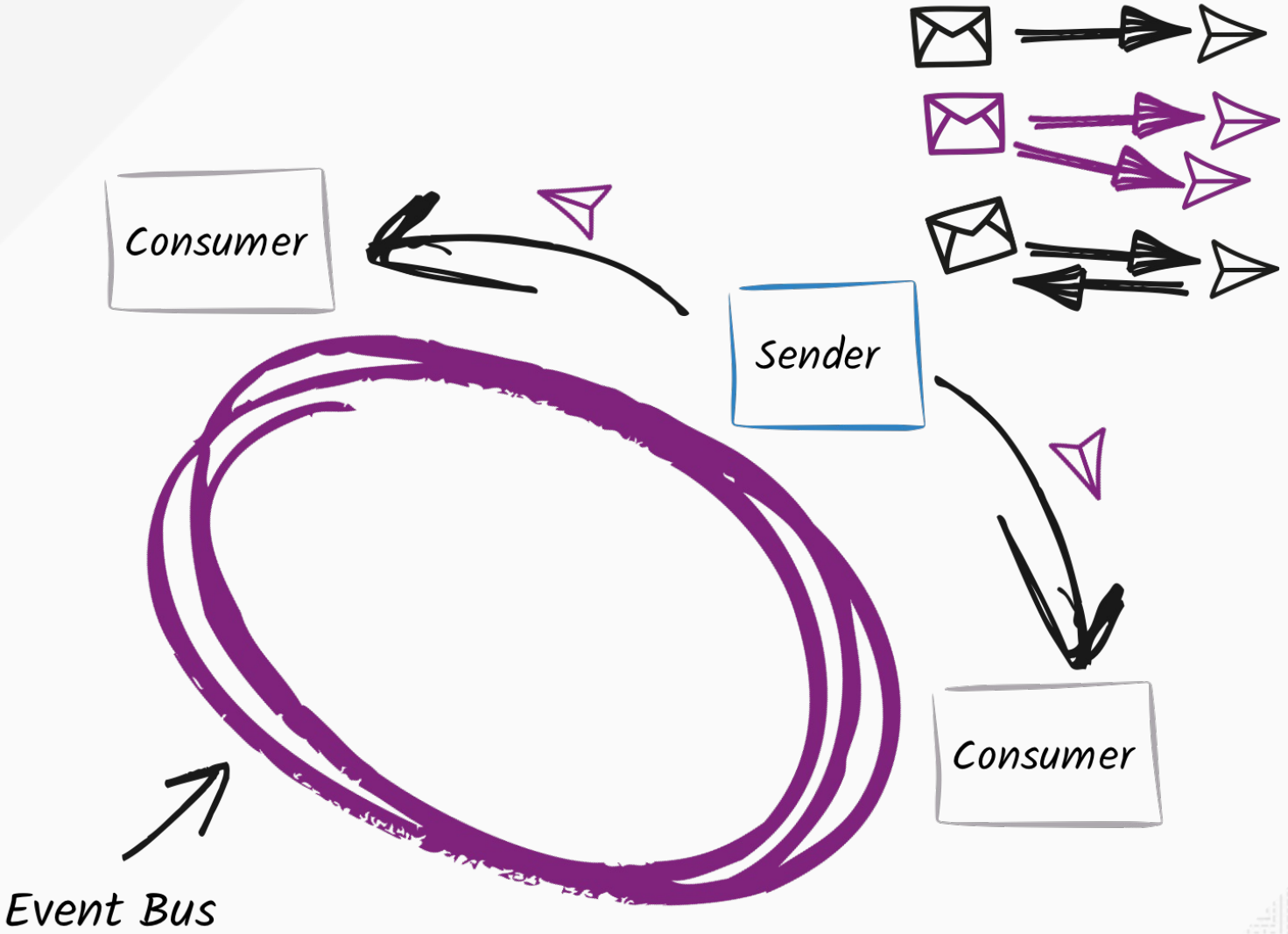
- Allows different components to communicate regardless
  - the implementation language and their location
  - whether they run on vert.x or not (using bridges)

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

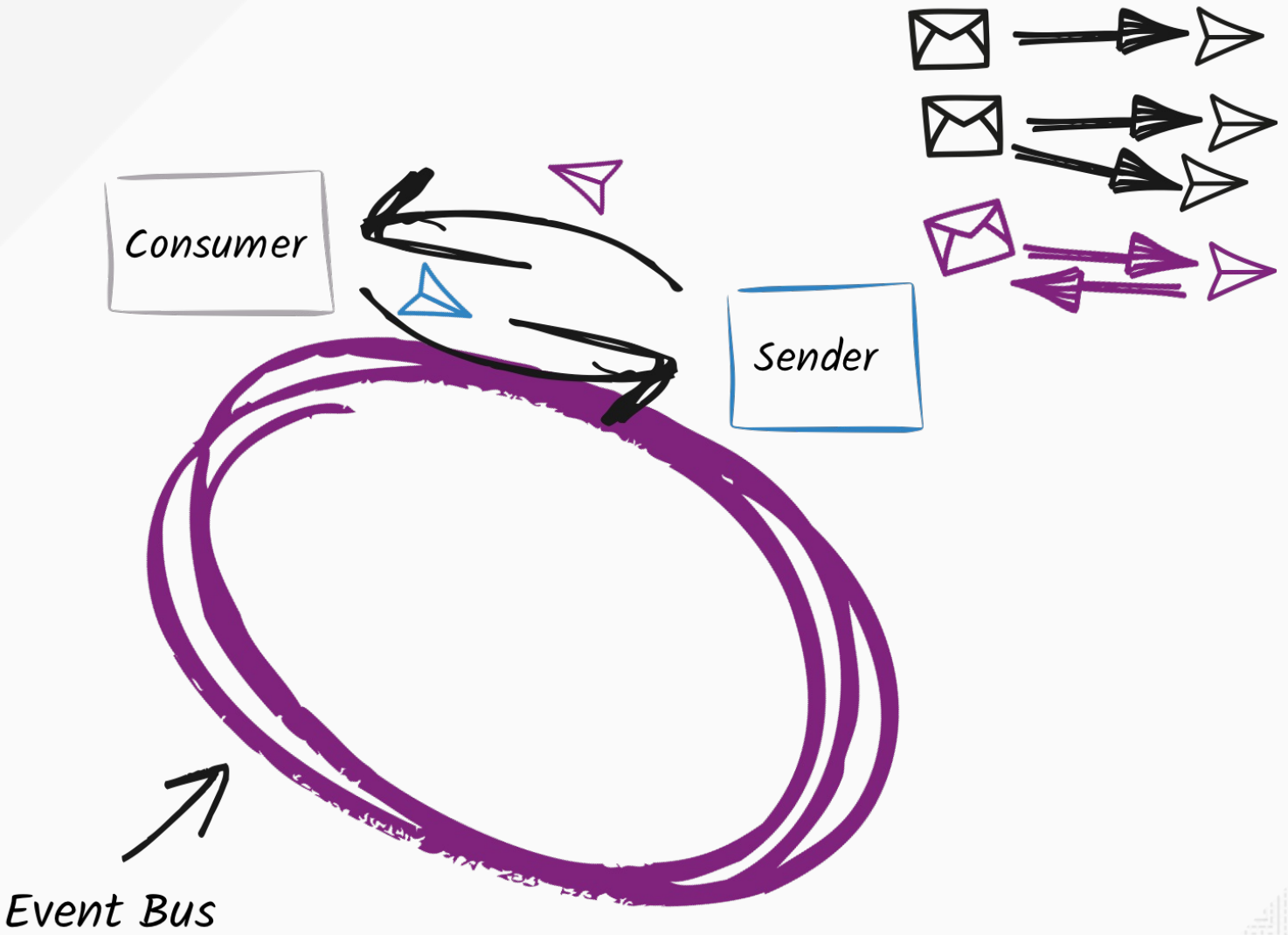
POINT TO POINT



PUBLISH / SUBSCRIBE

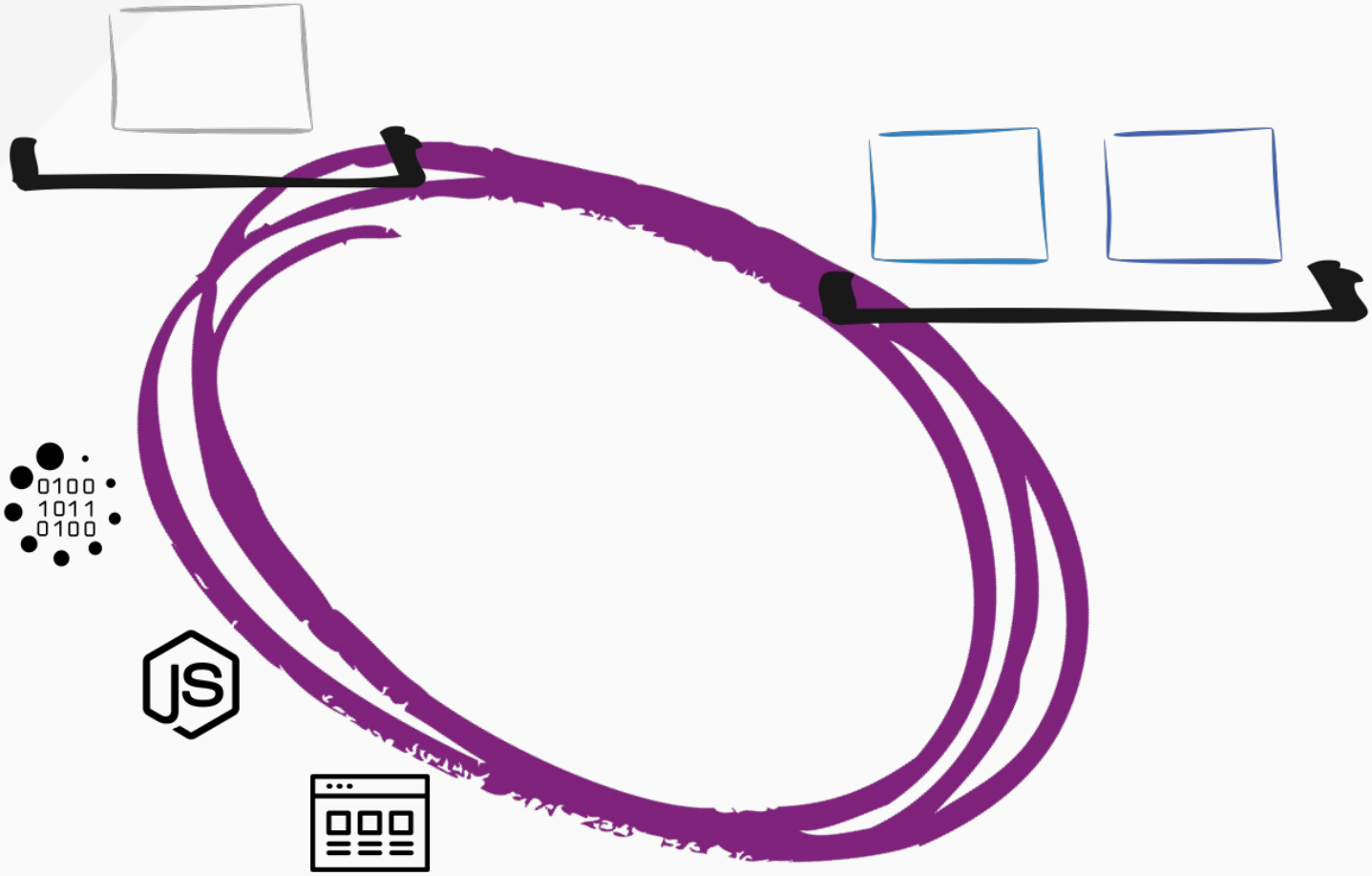


REQUEST / RESPONSE



DISTRIBUTED EVENT BUS

Almost anything can send and receive messages



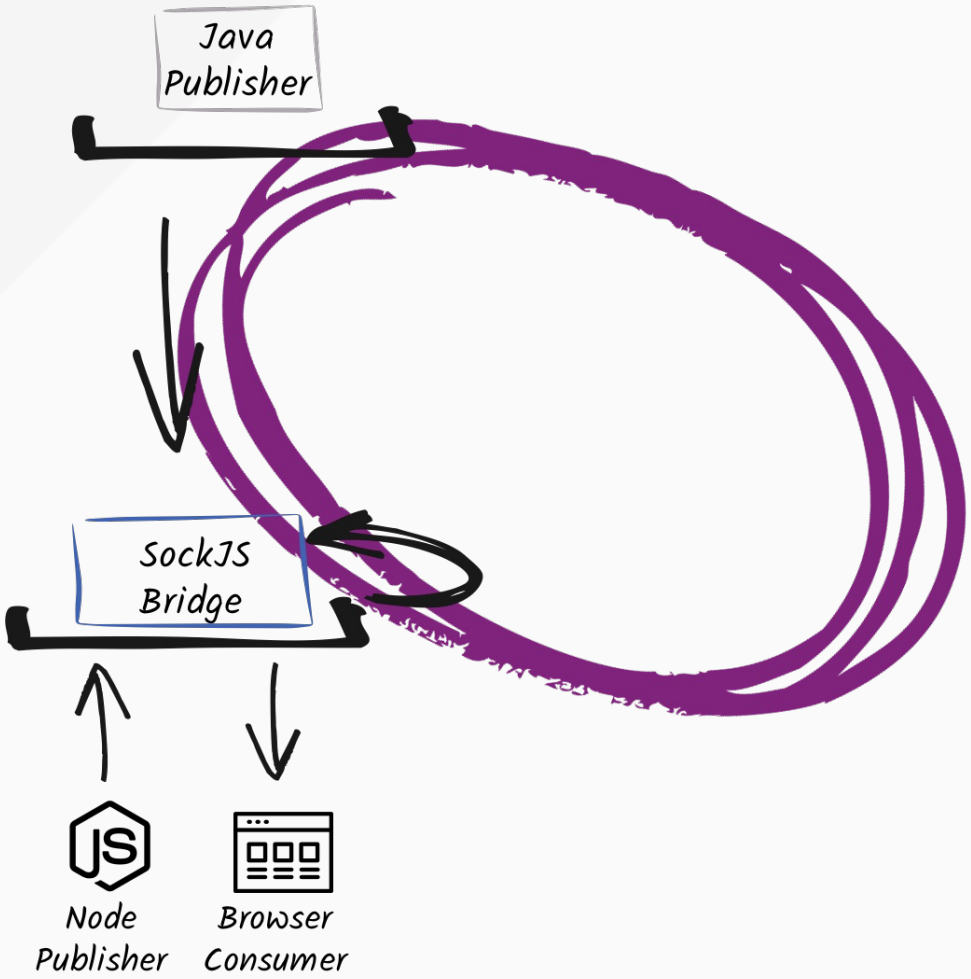
redhat.

DISTRIBUTED EVENT BUS

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

```
hello from java (8321)
hello from java (8320)
hello from java (8319)
hello from java (8318)
hello from java (8317)
```

DISTRIBUTED EVENT BUS



EVENTBUS CLIENTS AND BRIDGES

Clients:

- SockJS: browser, node.js
- TCP: every system able to open a TCP socket
- Go: for system develop in Go

Bridges:

- Stomp
- AMQP
- Camel

RELIABILITY PATTERNS

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

RELIABILITY

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

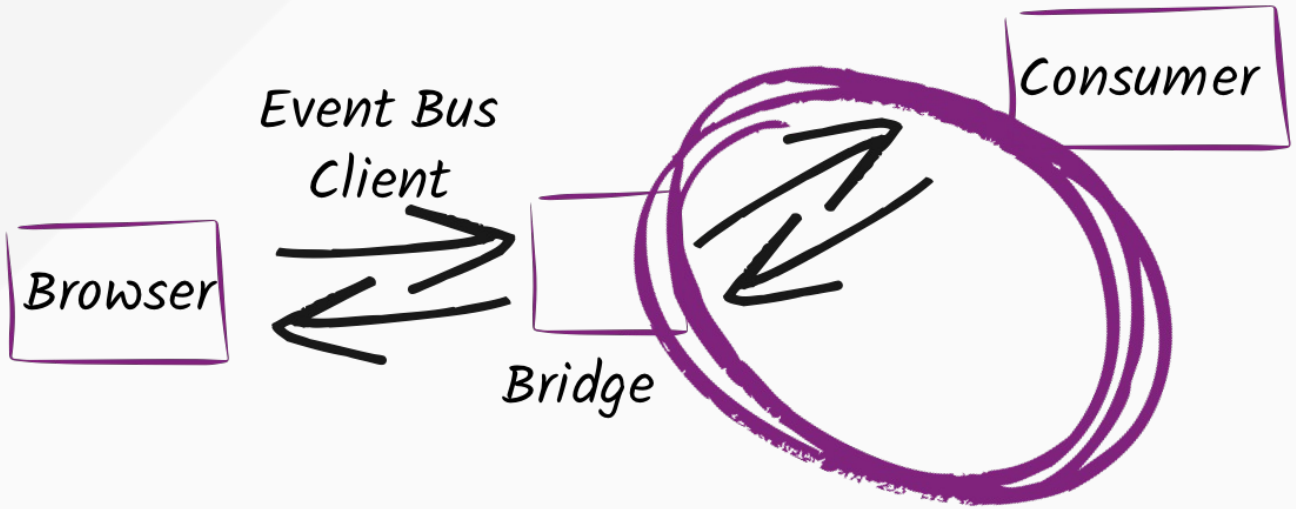
It's about being prepared to fail,
and handling these **failures**.

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

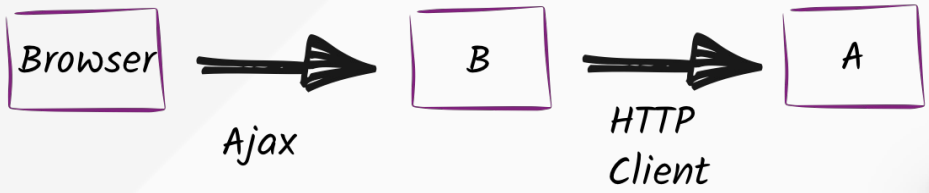


Invoke

MANAGING FAILURE

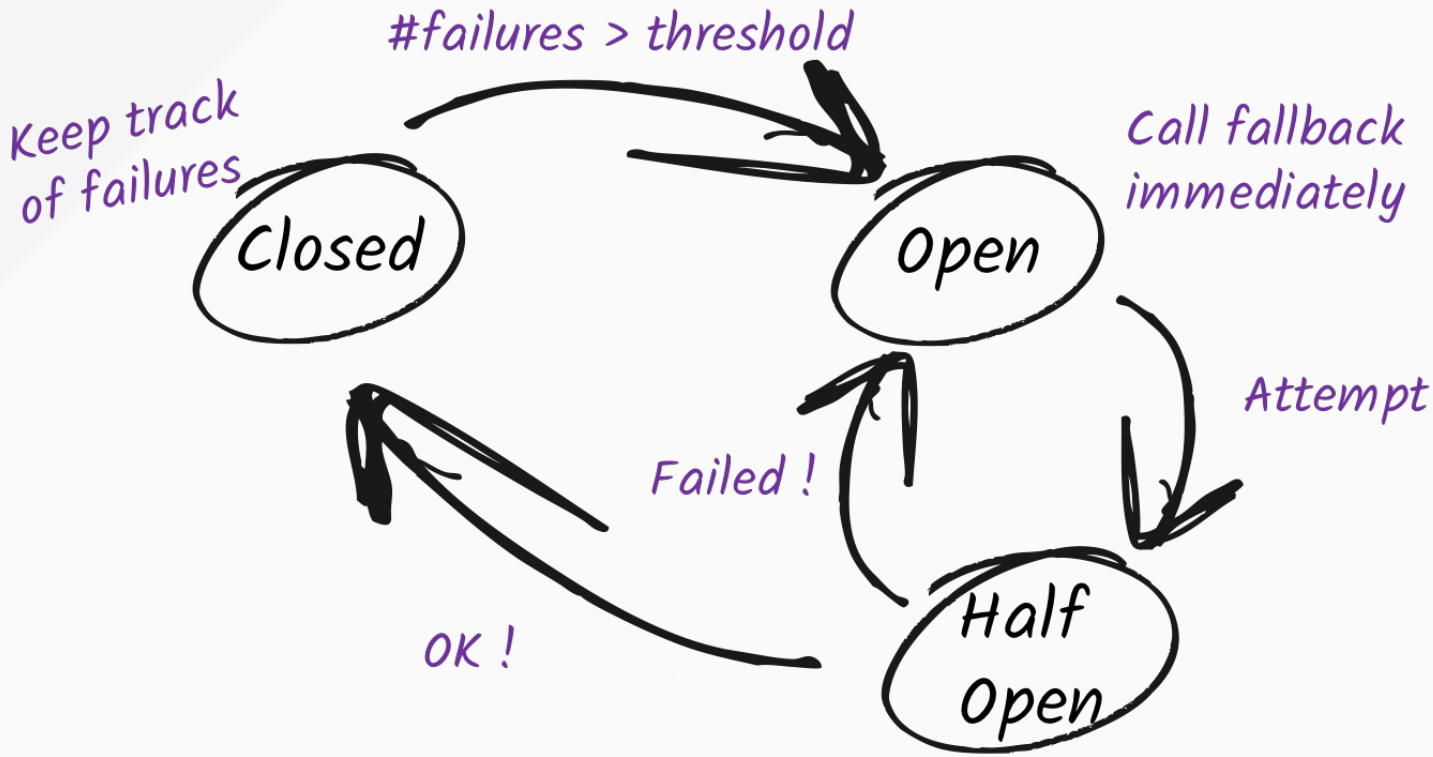
```
client.get("/", response -> {
  response
    .exceptionHandler(t -> {
      // If the content cannot be read
      rc.response().end("Sorry... " + t.getMessage());
    })
    .bodyHandler(buffer -> {
      rc.response().end("Ola " + buffer.toString());
    });
  })
    .setTimeout(3000)
    .exceptionHandler(t -> {
      // If the connection cannot be established
      rc.response().end("Sorry... " + t.getMessage());
    })
    .end();
```

MANAGING FAILURE



Invoke

CIRCUIT BREAKER

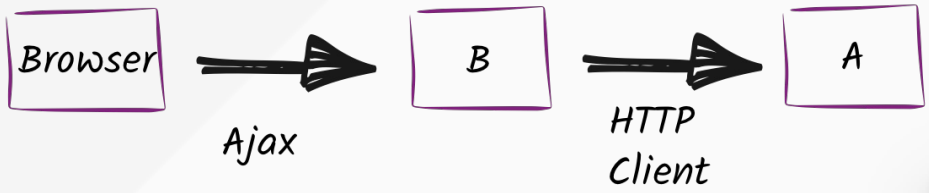


CIRCUIT BREAKER

```
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

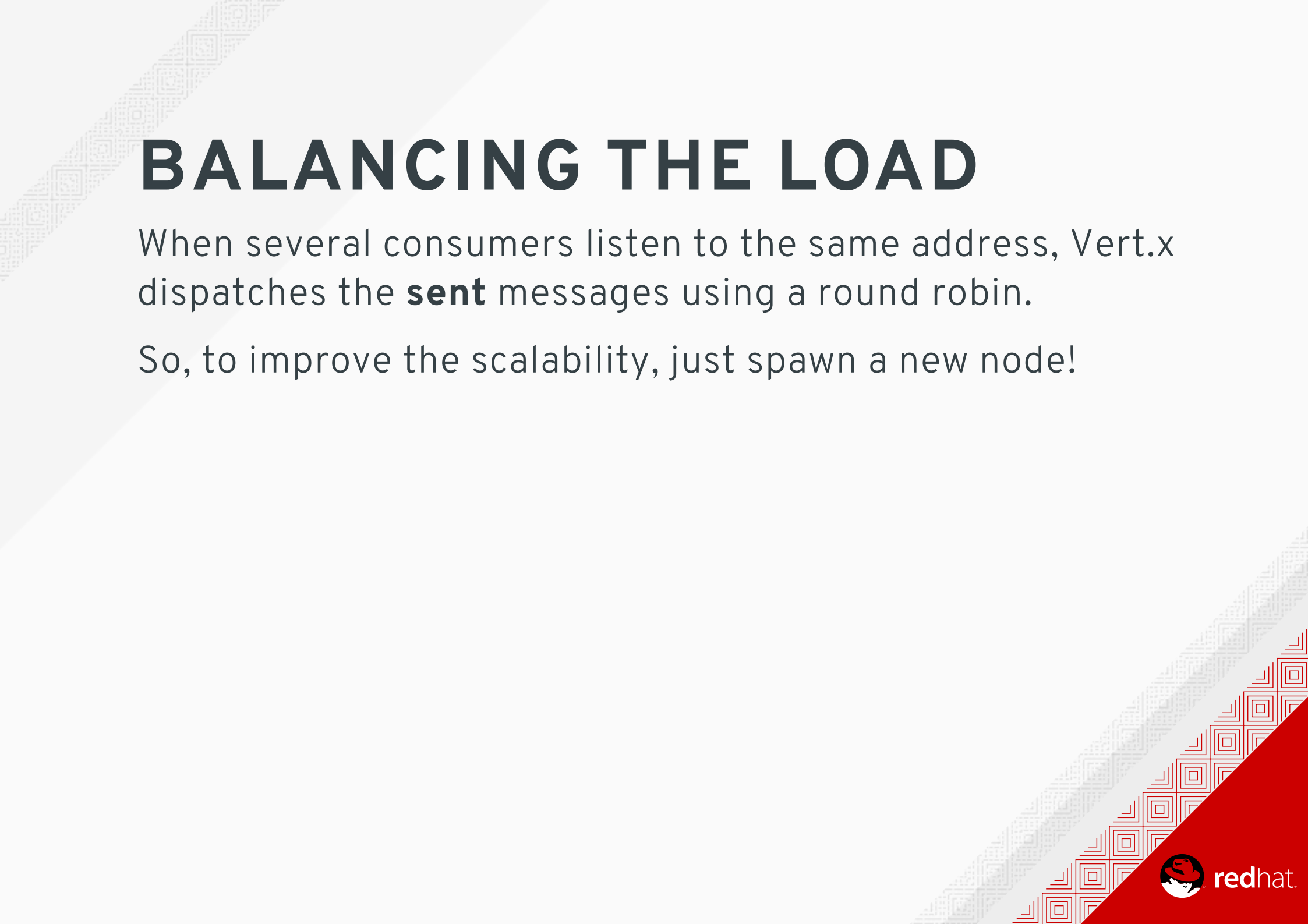
SCALABILITY PATTERNS

BE PREPARED TO BE FAMOUS

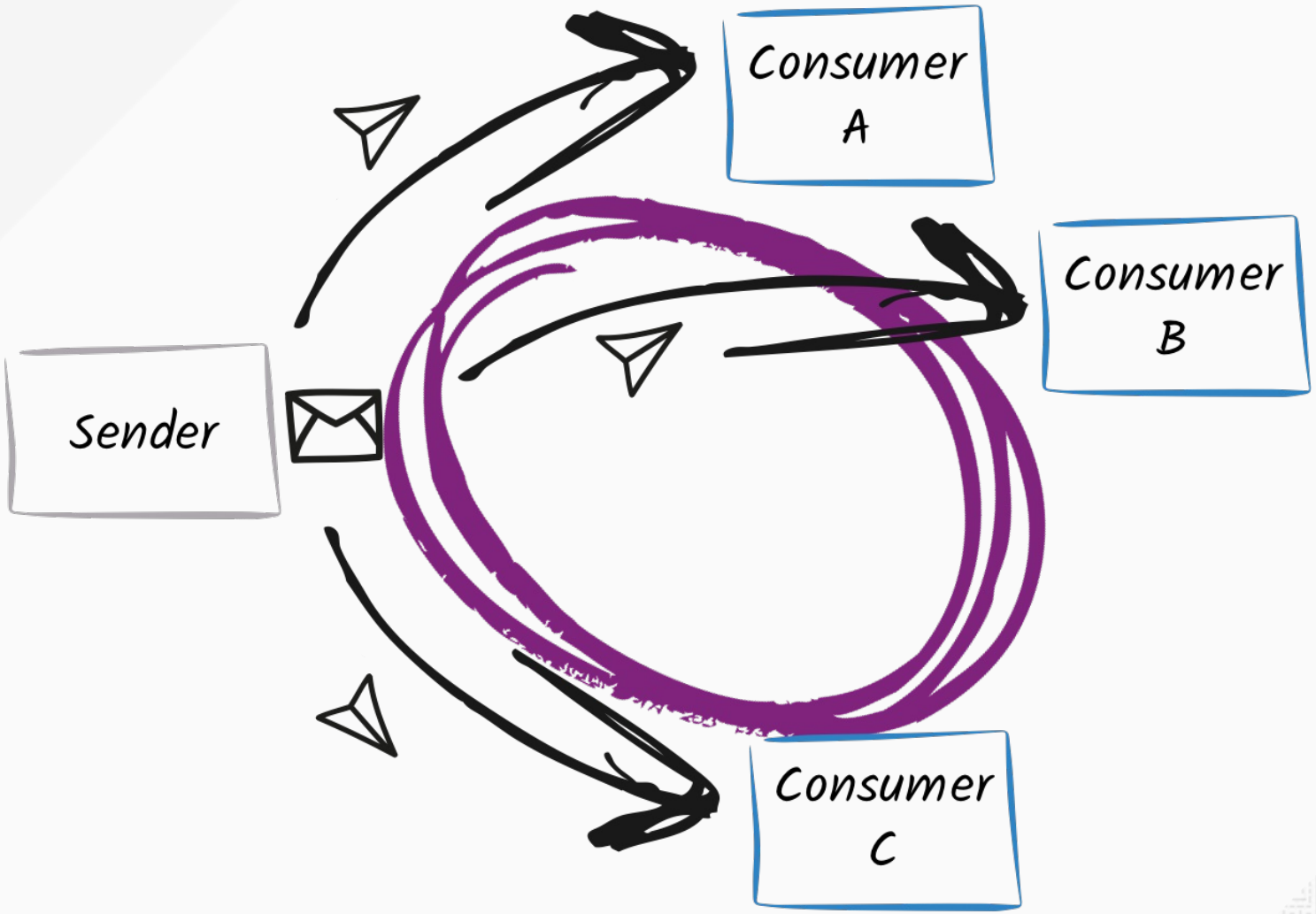
BALANCING THE LOAD

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!



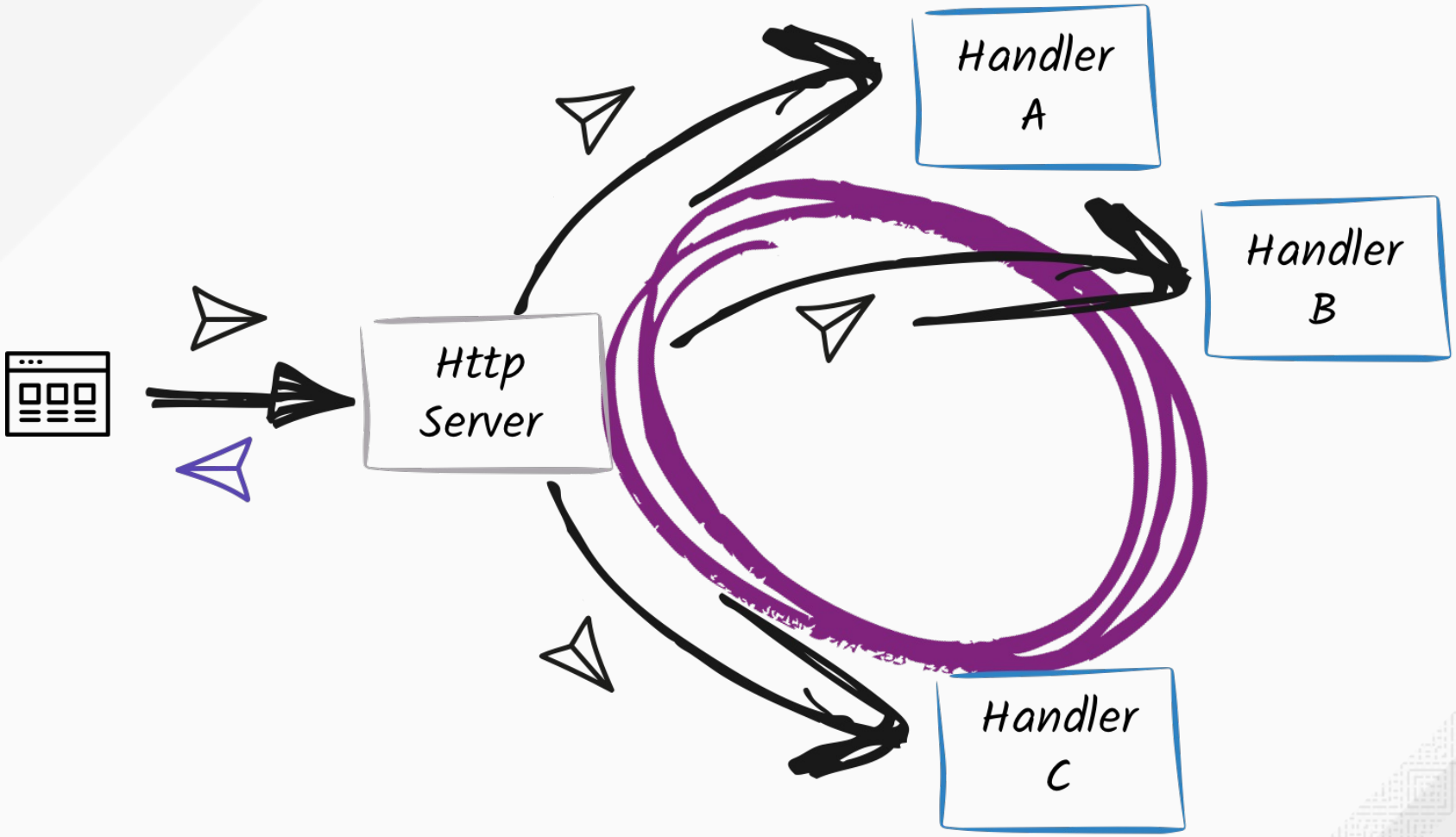
BALANCING THE LOAD



BALANCING THE LOAD

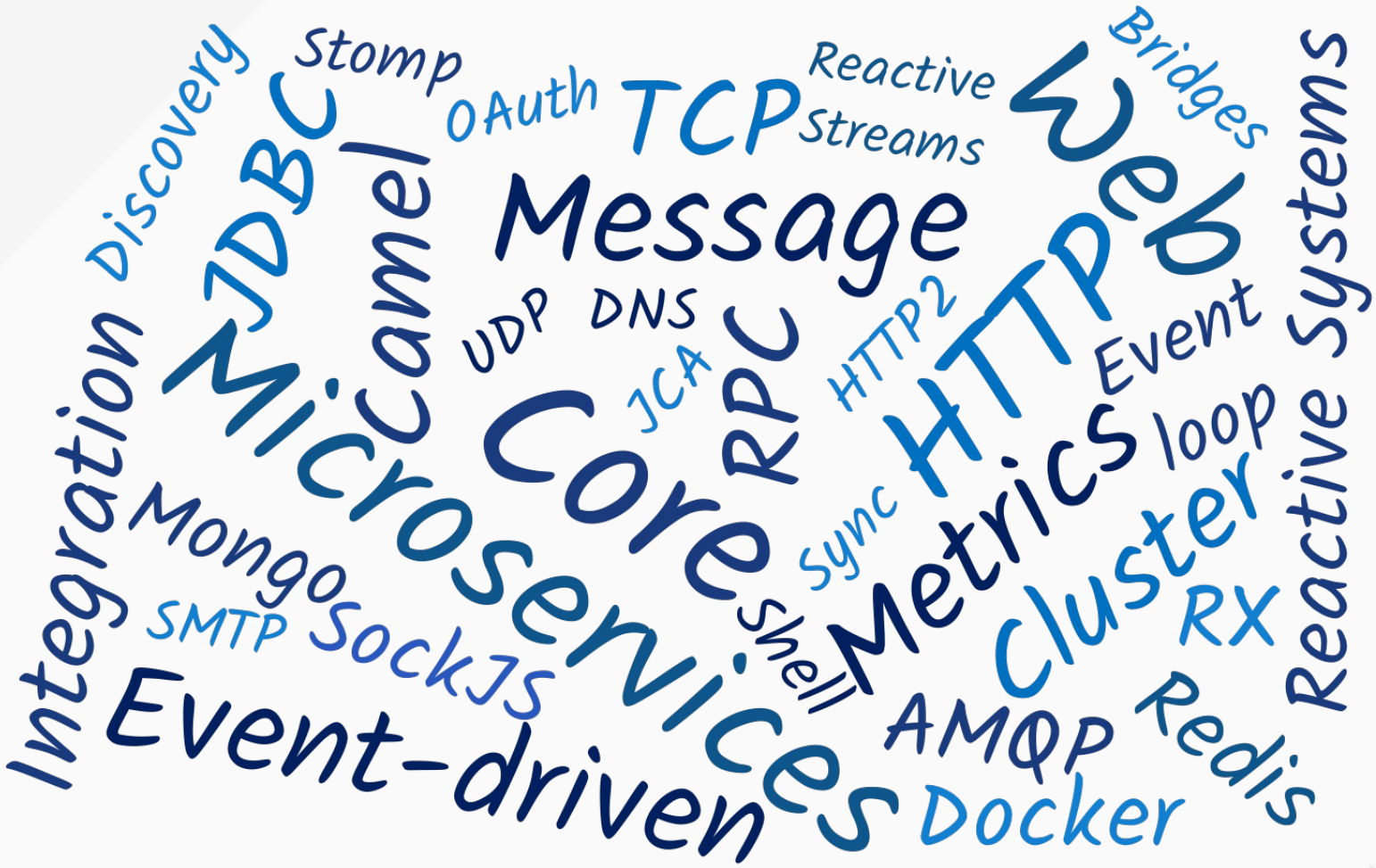
Invoke

SCALING HTTP



THIS IS NOT THE END()

BUT THE FIRST STEP ON THE REACTIVE MICROSERVICE ROAD


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 ?

- http://vertx.io
- http://vertx.io/blog/posts/introduction-to-vertx.html
- http://vertx-lab.dynamis-technologies.com
- https://github.com/vert-x3/vertx-examples

THANK YOU!


linkedin.com/company/red-hat


Red HatNews


facebook.com/redhatinc

youtube.com/Red Hat


redhat.