3 Event bus: the backbone of a Vert.x application

📌 A Vert.x application is made of one or more verticles, and each verticle forms a unit for processing asynchronous events

📌 It is common to specialize verticles by functional and technical concerns, such as having one verticle for exposing an HTTP API and another for dealing with a data store

📌 This design also encourages the deployment of several instances of a given verticle for scalability purposes.

❓how verticles can communicate with each other?

For example, an HTTP API verticle needs to talk to the data store verticle if the larger Vert.x application is to do anything useful.

Connecting verticles and making sure they can cooperate is the role of the **event bus**

This is important when building reactive applications—the event bus offers a way to transparently distribute event-processing work both inside a process and across several nodes over the network.

What is the event bus?

📌 The **event bus** is a means for sending and receiving messages in an **asynchronous** fashion.

Messages are sent to and retrieved from **destinations**.

A destination is simply a free-form string, such as **incoming.purchase.orders** or **incoming-purchase-orders,** although the former format with dots is preferred.

Message, can have

- body
- optional headers
- optional expiration delay

📌 Message bodies are commonly encoded using the Vert.x JSON representation

📌 The event bus allows for decoupling between verticles

📌 **Vert.x is polyglot,** the event bus allows verticles written in different languages to communicate with each other, whether for communications inside the same JVM process or across the network.

Communications over the event bus follow three patterns:

1. Point-to-point messaging
2. Request-reply messaging
3. Publish/subscribe messaging

❓ Is the event bus just another message broker?

The short answer is that no, the Vert.x event bus is not an alternative to Apache ActiveMQ, RabbitMQ, ZeroMQ, or Apache Kafka.

The longer explanation is that it is an event bus for verticle-to-verticle communications inside an application, not a message bus for application-to-application communications.

Vert.x integrates with message brokers, but the event bus is no replacement for this type of middleware.

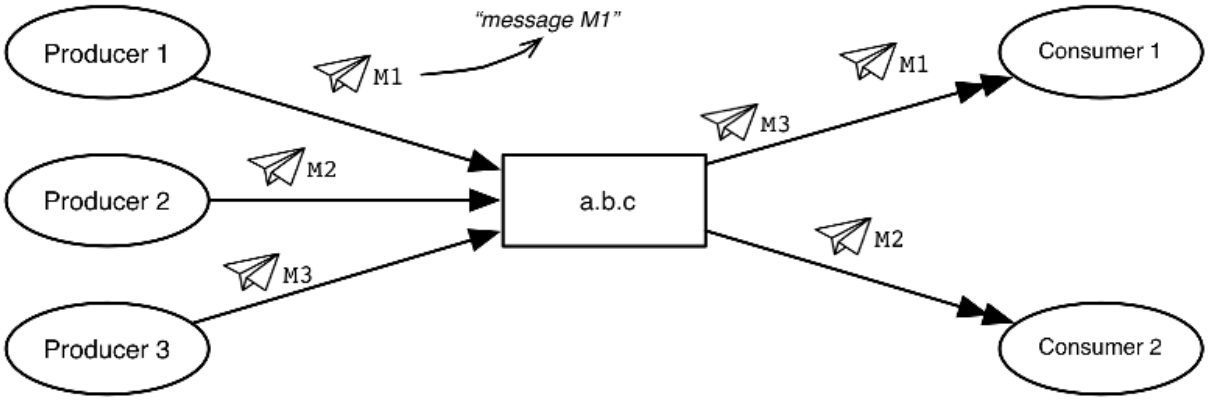
Specifically, the event bus does not do the following:

- Support message acknowledgments

- Support message priorities
- Support message durability to recover from crashes
- Provide routing rules
- Provide transformation rules (schema adaptation, scatter/gather, etc.)

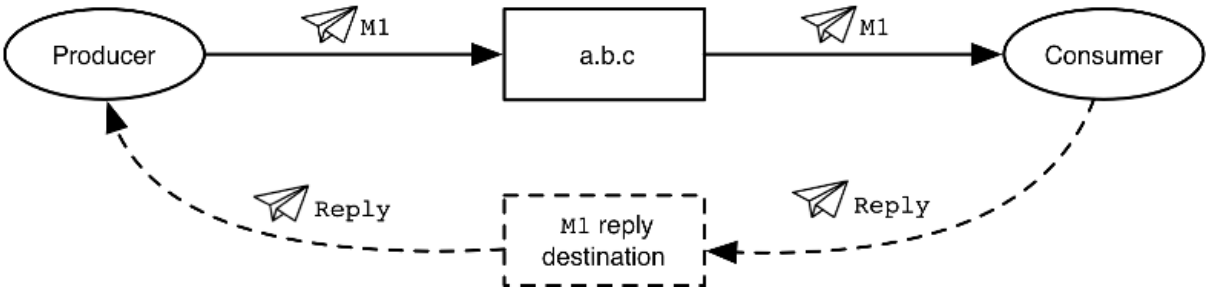
Point-to-point messaging

Point-to-point messaging over the event bus



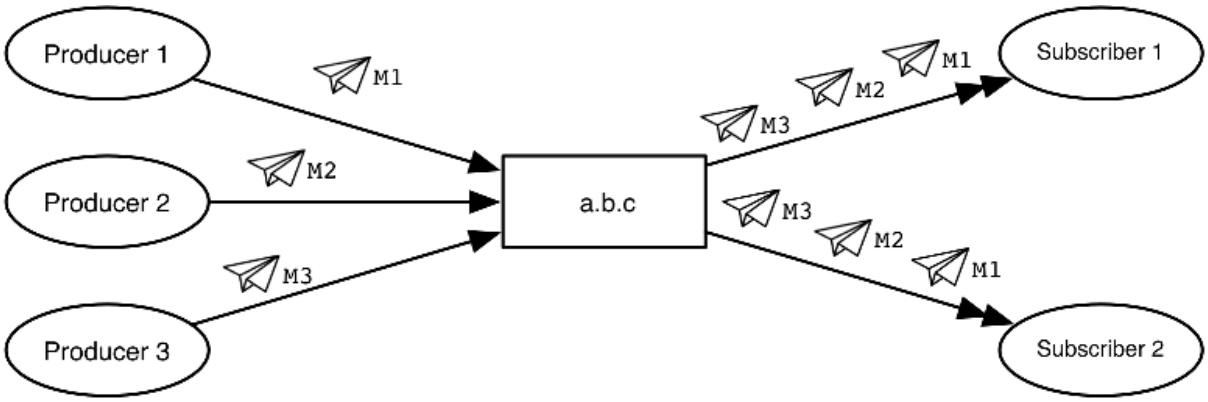
Request-reply messaging

Request-reply messaging over the event bus



Publish/subscribe messaging

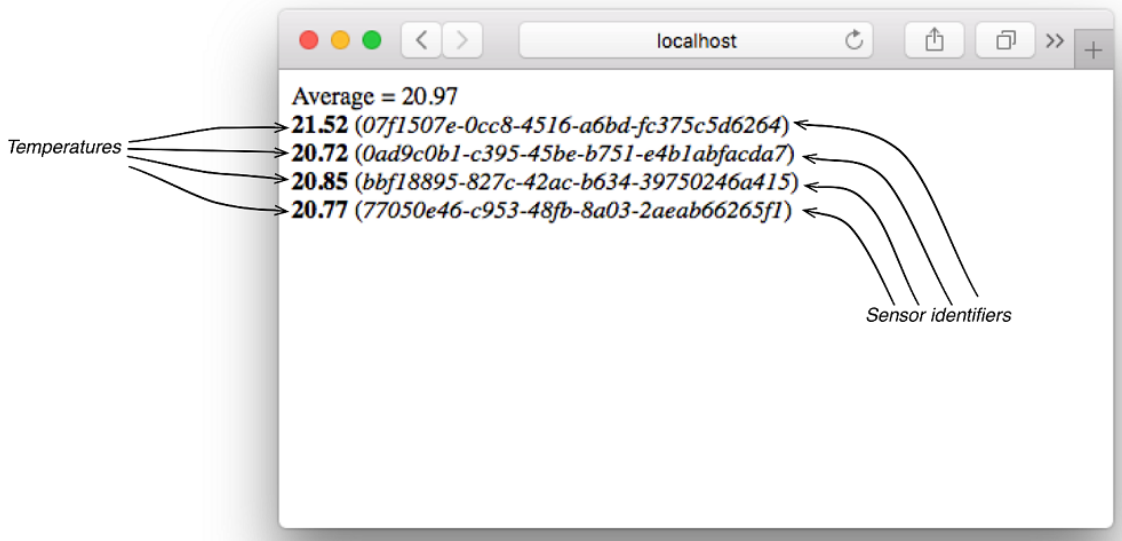
Publish/subscribe messaging over the event bus



The event bus in an example

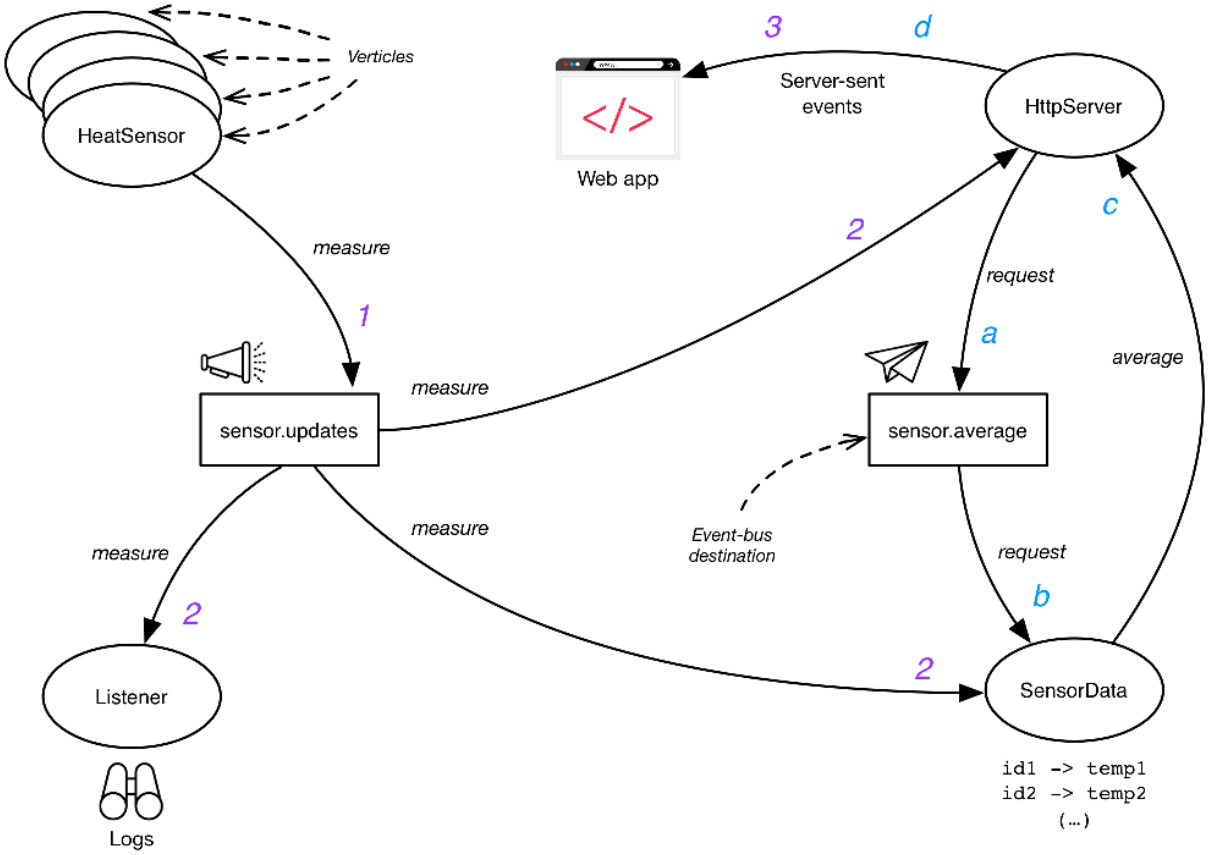
Let’s put the event bus to use and see how we can communicate between independent verticles.

Screenshot of the web interface



The communications between the web interface and the server will happen using **server-sent events**, a simple yet effective protocol supported by most web browsers.

Overview of the example architecture



The application is structured around four verticles:

- `HeatSensor` generates temperature measures at non-fixed rates and publishes them to subscribers to the `sensor.updates` destination. Each verticle has a unique sensor identifier.
- `Listener` monitors new temperature measures and logs them using SLF4J.
- `SensorData` keeps a record of the latest observed values for each sensor. It also supports request-response communications: sending a message to `sensor.average` triggers a computation of the average based on the latest data, and the result is sent back as a response.
- `HttpServer` exposes the HTTP server and serves the web interface. It pushes new values to its clients whenever a new temperature measurement has

been observed, and it periodically asks for the current average and updates all the connected clients.

Heat sensor verticle

```
public class HeatSensor extends AbstractVerticle {
  private final Random random = new Random();
  private final String sensorId = UUID.randomUUID().toString();
  private double temperature = 21.0;

  @Override
  public void start() {
    scheduleNextUpdate();
  }

  private void scheduleNextUpdate() {
    vertx.setTimer(random.nextInt(5000) + 1000, this::update);
  }

  private void update(long timerId) {
    temperature = temperature + (delta() / 10);
    JsonObject payload = new JsonObject()
      .put("id", sensorId)
      .put("temp", temperature);
    vertx.eventBus().publish("sensor.updates", payload);
    scheduleNextUpdate();
  }

  private double delta() {
    if (random.nextInt() > 0) {
      return random.nextGaussian();
    } else {
      return -random.nextGaussian();
    }
  }
}
```

Listener verticle

```
public class Listener extends AbstractVerticle {
  private final Logger logger = LoggerFactory.getLogger(Listener.class);
```

```
  private final DecimalFormat format = new DecimalFormat("#.##");

  @Override
  public void start() {
    EventBus bus = vertx.eventBus();
    bus.<JsonObject>consumer("sensor.updates", msg -> {
      JsonObject body = msg.body();
      String id = body.getString("id");
      String temperature = format.format(body.getDouble("temp"));
      logger.info("{} reports a temperature ~{}C", id, temp);
    });
  }
}
```

Sensor data verticle

```
public class SensorData extends AbstractVerticle {
  private final HashMap<String, Double> lastValues = new HashMap<>();

  @Override
  public void start() {
    EventBus bus = vertx.eventBus();
    bus.consumer("sensor.updates", this::update);
    bus.consumer("sensor.average", this::average);
  }

  private void update(Message<JsonObject> message) {
    JsonObject json = message.body();
    lastValues.put(json.getString("id"), json.getDouble("temp"));
  }

  private void average(Message<JsonObject> message) {
    double avg = lastValues.values().stream()
      .collect(Collectors.averagingDouble(Double::doubleValue));
    JsonObject json = new JsonObject().put("average", avg);
    message.reply(json);
  }
}
```

HTTP server verticle

```
public class HttpServer extends AbstractVerticle {
  @Override
```

```
  public void start() {
    vertx.createHttpServer()
      .requestHandler(this::handler)
      .listen(config().getInteger("port", 8080));
  }

  private void handler(HttpServerRequest request) {
    if ("/".equals(request.path())) {
      request.response().sendFile("index.html");
    } else if ("/sse".equals(request.path())) {
      sse(request);
    } else {
      request.response().setStatusCode(404);
    }
  }
// (...)
}
```

The handler deals with three cases:

1. Serving the web application to browsers
2. Providing a resource for server-sent events
3. Responding with 404 errors for any other resource path

index.html

```
<div id="avg"></div>
<div id="main"></div>
<script language="JavaScript">
  const sse = new EventSource("/sse")
  const main = document.getElementById("main")
  const avg = document.getElementById("avg")

  sse.addEventListener("update", (evt) => {
    const data = JSON.parse(evt.data)
    let div = document.getElementById(data.id);
    if (div === null) {
      div = document.createElement("div")
      div.setAttribute("id", data.id)
      main.appendChild(div)
    }
    div.innerHTML = `<strong>${data.temp.toFixed(2)}</strong> (<em>${data.id}</em>)`
  })
```

```
  sse.addEventListener("average", (evt) => {
    const data = JSON.parse(evt.data)
    avg.innerText = `Average = ${data.average.toFixed(2)}`
  })
</script>
```

```
private void sse(HttpServerRequest request) {
  HttpServerResponse response = request.response();
  response
    .putHeader("Content-Type", "text/event-stream")
    .putHeader("Cache-Control", "no-cache")
    .setChunked(true);

  MessageConsumer<JsonObject> consumer = vertx.eventBus().consumer("sensor.updates");
  consumer.handler(msg -> {
    response.write("event: update\n");
    response.write("data: " + msg.body().encode() + "\n\n");
  });

  TimeoutStream ticks = vertx.periodicStream(1000);
  ticks.handler(id -> {
    vertx.eventBus().<JsonObject>request("sensor.average", "", reply -> {
      if (reply.succeeded()) {
        response.write("event: average\n");
        response.write("data: " + reply.result().body().encode() + "\n\n");
      }
    });
  });

  response.endHandler(v -> {
    consumer.unregister();
    ticks.cancel();
  });
}
```

Bootstrapping the application

```
public class Main {
  public static void main(String[] args) {
    Vertx vertx = Vertx.vertx();
```

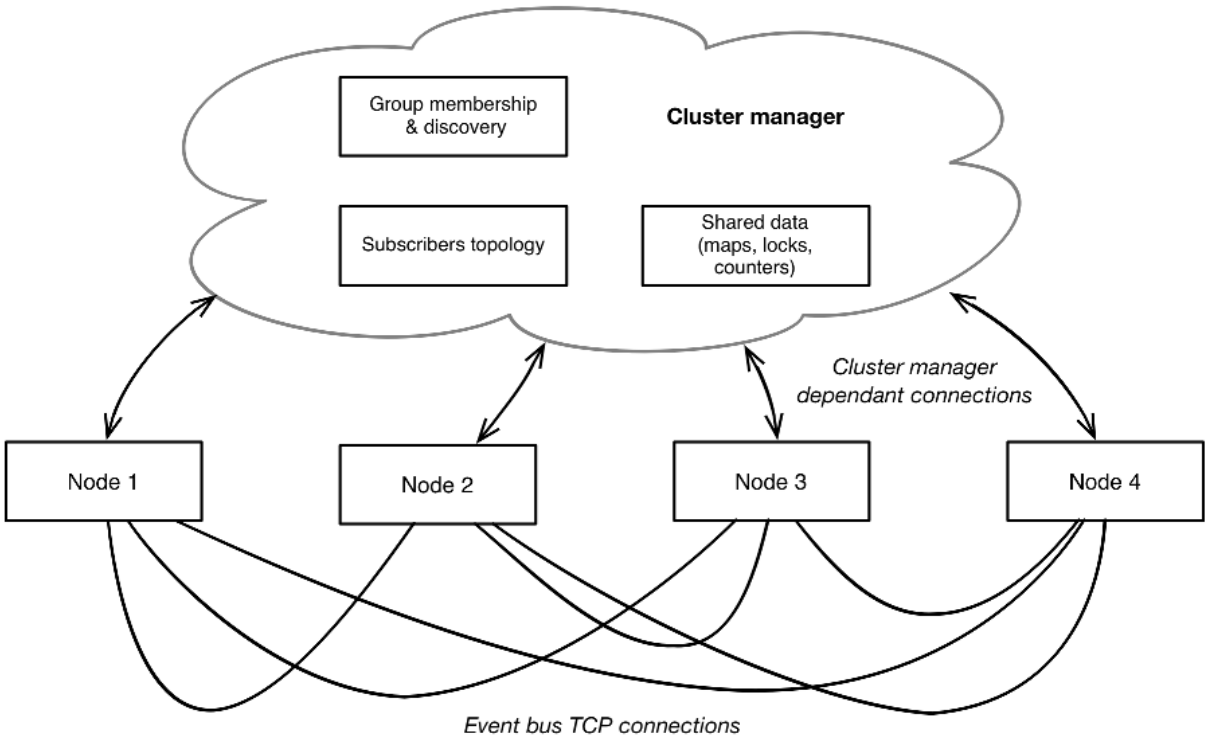
```
    vertx.deployVerticle("HeatSensor", new DeploymentOptions().setInstances(4));
    vertx.deployVerticle("Listener");
    vertx.deployVerticle("SensorData");
    vertx.deployVerticle("HttpServer");
  }
}
```

Clustering and the distributed event bus

use Vert.x clustering and benefit from a distributed event bus.

Clustering in Vert.x

Vert.x applications can run in clustering mode where a set of Vert.x application nodes can work together over the network



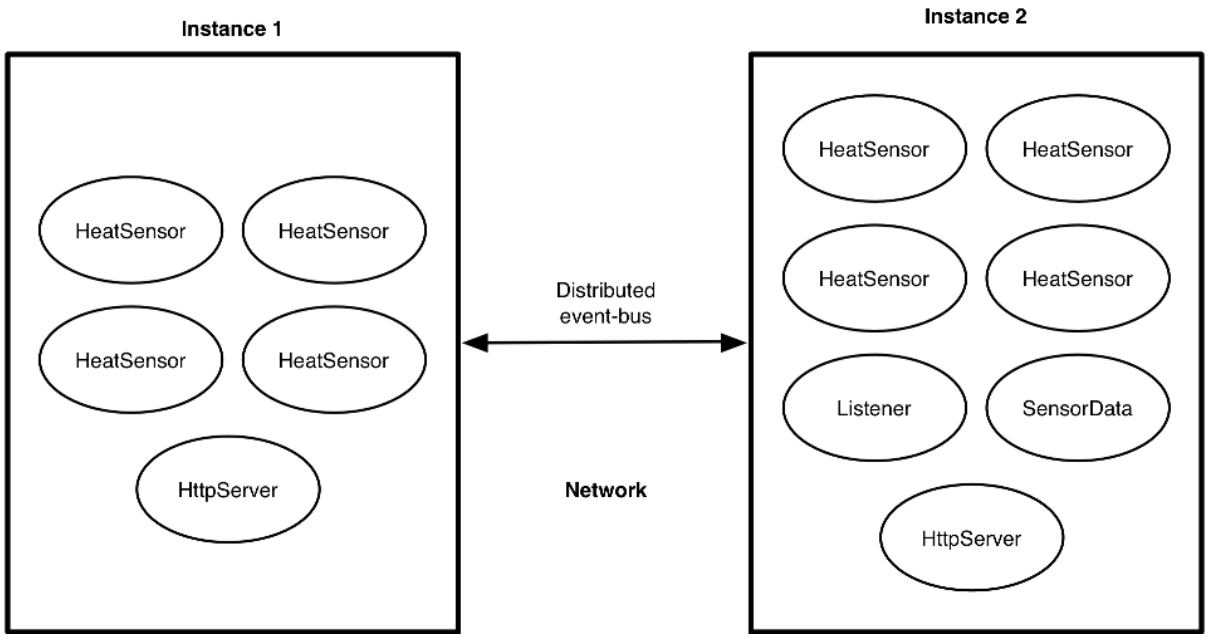
A cluster manager provides the following set of functionalities:

- Group membership and discovery allow discovering new nodes, maintaining the list of current nodes, and detecting when nodes disappear.
- Shared data allows maps and counters to be maintained cluster-wide, so that all nodes share the same values. Distributed locks are useful for some forms of coordination between nodes.
- Subscribers topology allows knowing what event-bus destinations each node has interest in.

There are several cluster manager implementations for Vert.x based on **Hazelcast**, **Infinispan**, **Apache Ignite**, and **Apache ZooKeeper.**

From event bus to distributed event bus

Clustered application overview



Summary

- The event bus is the preferred way for verticles to communicate, and it uses asynchronous message passing.
- The event bus implements both publish/subscribe (one-to-many) and point-to-point (many-to-one) communications.
- While it looks like a traditional message broker, the event bus does not provide durability guarantees, so it must only be used for transient data.
- Clustering allows networked instances to communicate over the distributed event bus in a transparent fashion, and to scale the workload across several

application instances.