Using Publish-Subscribe Infrastructures

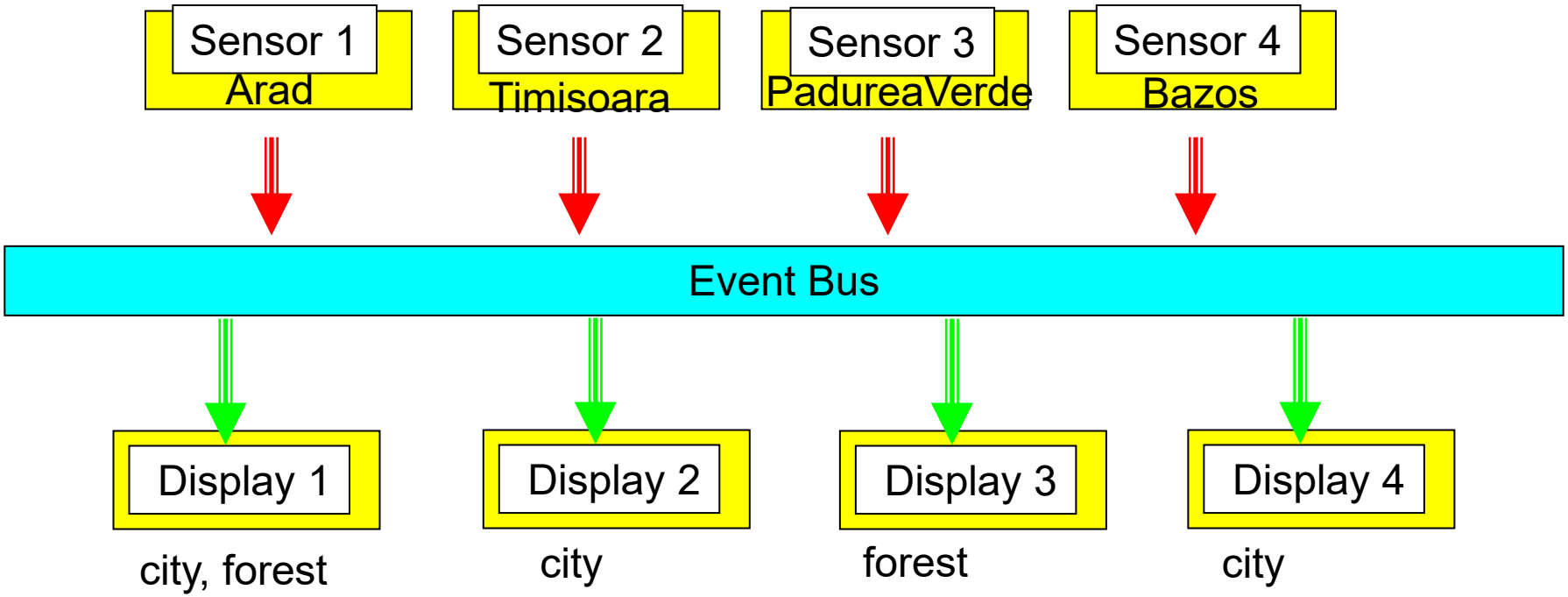
Case Study: RabbitMQ

Event Service Infrastructures

- **Infrastructures** that provide a Publish-Subscribe API to be used by applications; Examples:
- **In-process EventBus**:
  - Greenrobot eventbus
  - Akka event bus
  - Alternative: ReactiveX - RXJava
- **Inter-process, distributed Publish-Subscribe** infrastructure: part of middleware for distributed computing
  - Message Broker vs. EventBus
  - RabbitMQ, ActiveMQ, Kafka
  - JMS

Motivating Example: A Scenario with Sensors and Displays

- A variable number of Sensors periodically report measured temperature values
- A variable number of displays show values from sensors
- Sensors may be in a city or in a forest. Each display should be configurable to follow only city temps, only forest temps, or both.



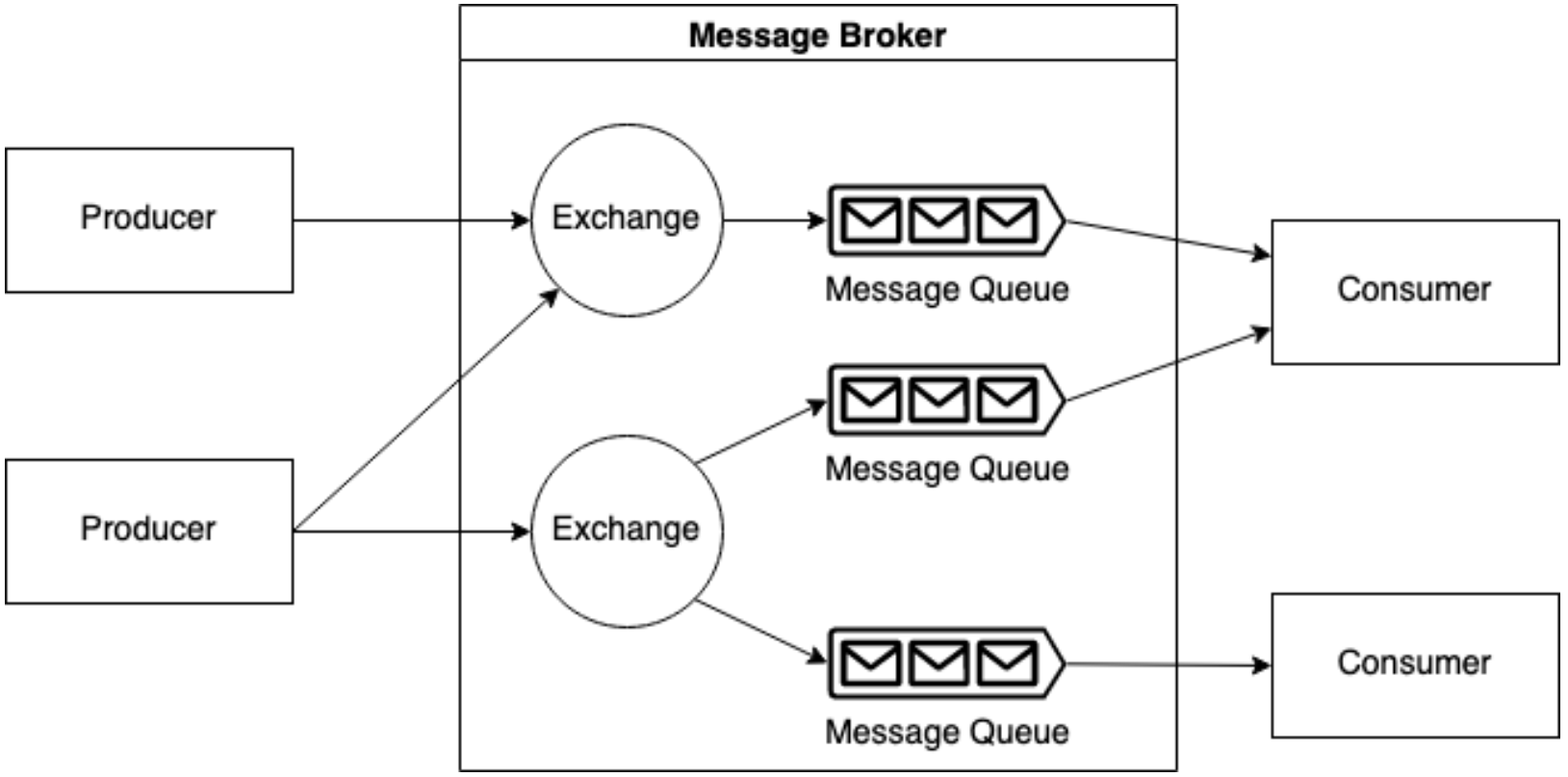
Example: Distributed Publish-Subscribe with RabbitMQ

RabbitMQ

- RabbitMQ is a **message broker** or MOM (Message Oriented Middleware)
- The difference between message-oriented and event-oriented: a message has a specific addressable recipient while an event just happen for others to observe
- RabbitMQ allows applications to define and use *message queues.*
- RabbitMQ **can be used in different modes**:
  - Direct messaging
  - Publish-Subscribe
  - RPC (Remote Procedure Call)

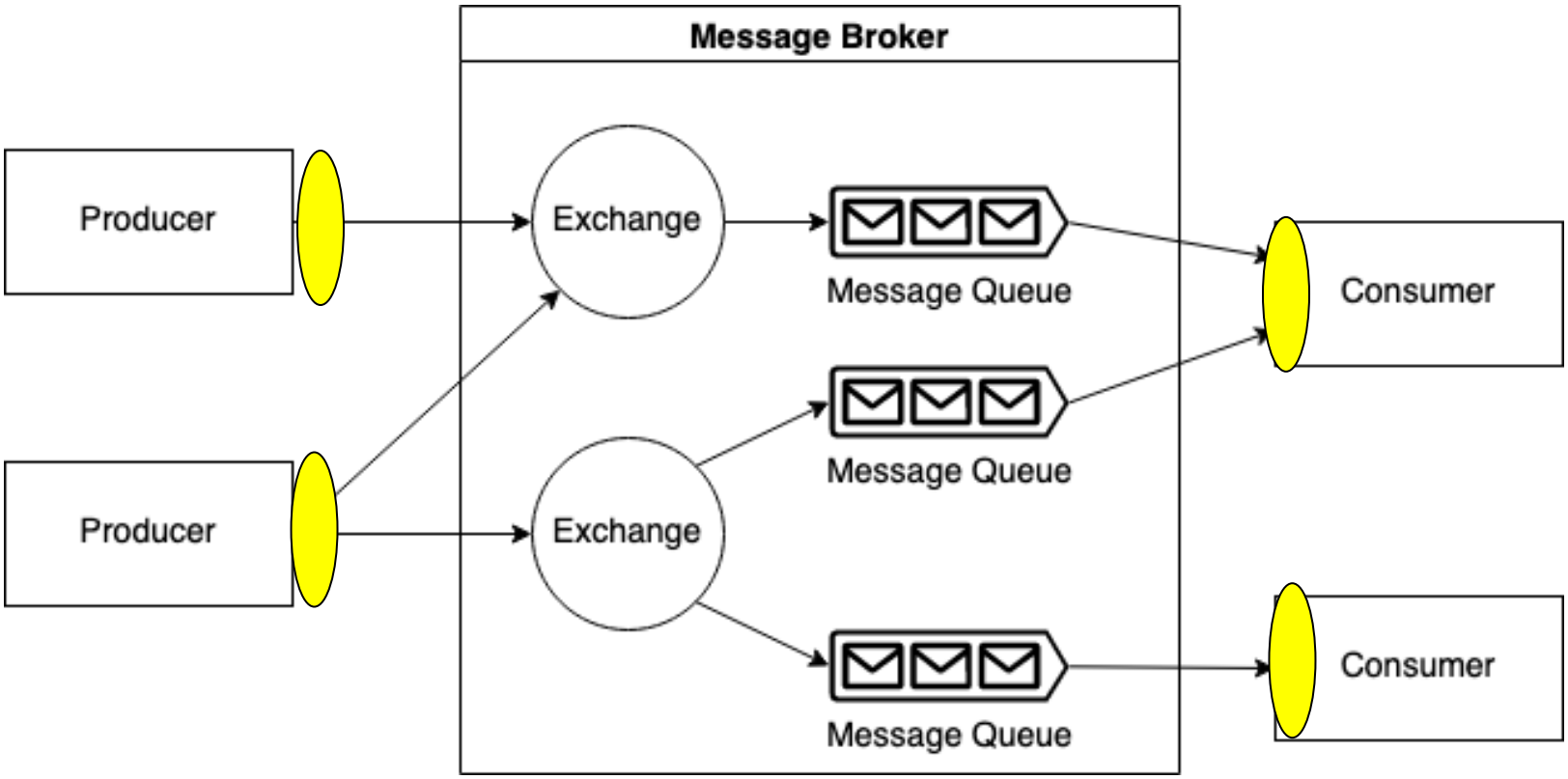
RabbitMQ architecture

- Message Broker = a running RabbitMQ instance (a server)



Client libraries

- Producers and Consumers need a **client library** that understands the same protocol in order to communicate with RabbitMQ.
- There are many options for AMQP client libraries in many different languages.

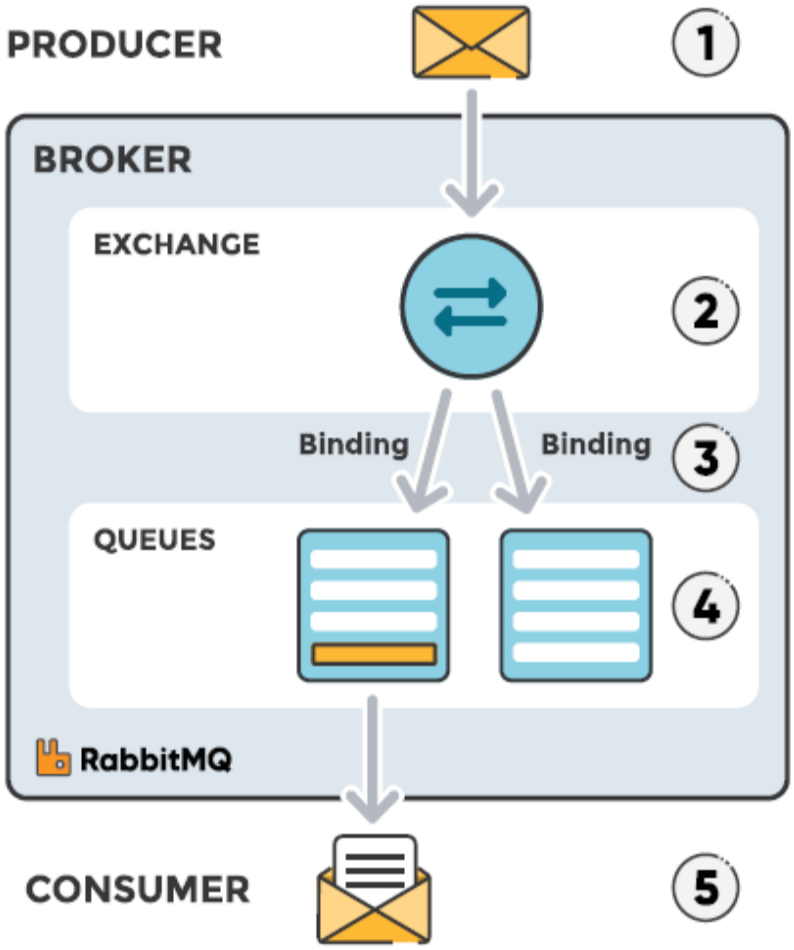


RabbitMQ Concepts

- **Producer:** Application that sends the messages.
- **Consumer:** Application that receives the messages.
- **Message:** Data sent from producer to consumer through RabbitMQ
- **Queue:** Buffer within the broker that stores messages.
- **Connection:** A link (TCP connection) between the application (producer or consumer) and the broker. performs underlying networking tasks including initial authentication, IP resolution, and networking.
- **Channel:** A virtual connection inside a connection. Channels can multiplex over a single TCP connection When publishing or consuming messages from a queue - it's all done over a channel.
- **Users:** It is possible to connect to RabbitMQ with a given username and password. Every user can be assigned permissions such as rights to read, write and configure privileges within the instance or for specific virtual hosts.
- **Vhost:** Provides a way to segregate applications using the same RabbitMQ instance. Users can have different permissions to different vhost and queues and exchanges can be created, so they only exist in one vhost.
- **AMQP (**Advanced Message Queuing Protocol): protocol used.

RabbitMQ Concepts

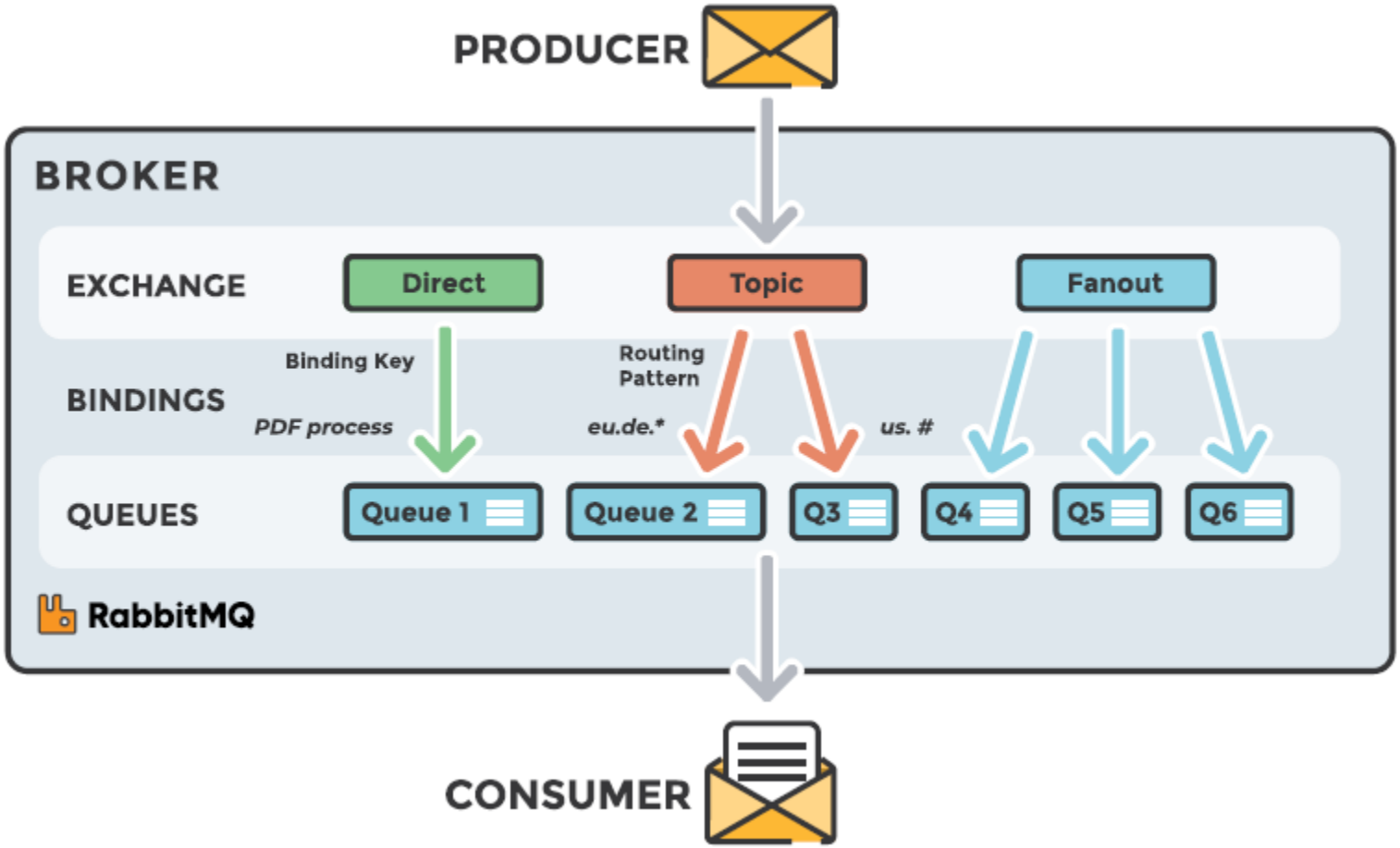
- **Queue**: stores messages until they are consumed by the consumer
- **Exchange**: serves as intermediary between producers and queues, enabling efficient ***message routing.***
- **Bindings:** link the queue to an exchange. It describes which queue is interested in messages from a given exchange.
- **Routing keys:** act as an address for the message.
- Queues, Exchanges and Bindings are server-side objects residing in the RabbitMQ server



https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html

RabbitMQ Concepts

- Types of Exchanges: Direct, Topic, Fanout. The type of Exchange determines the policy used for message routing

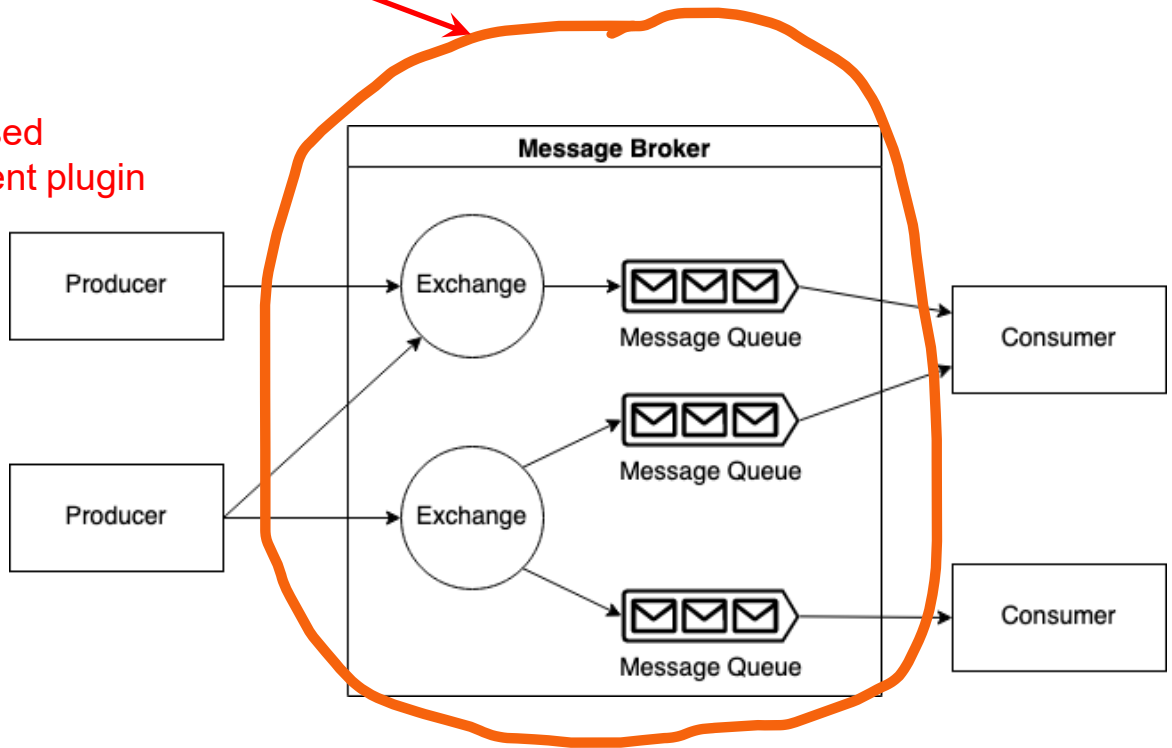


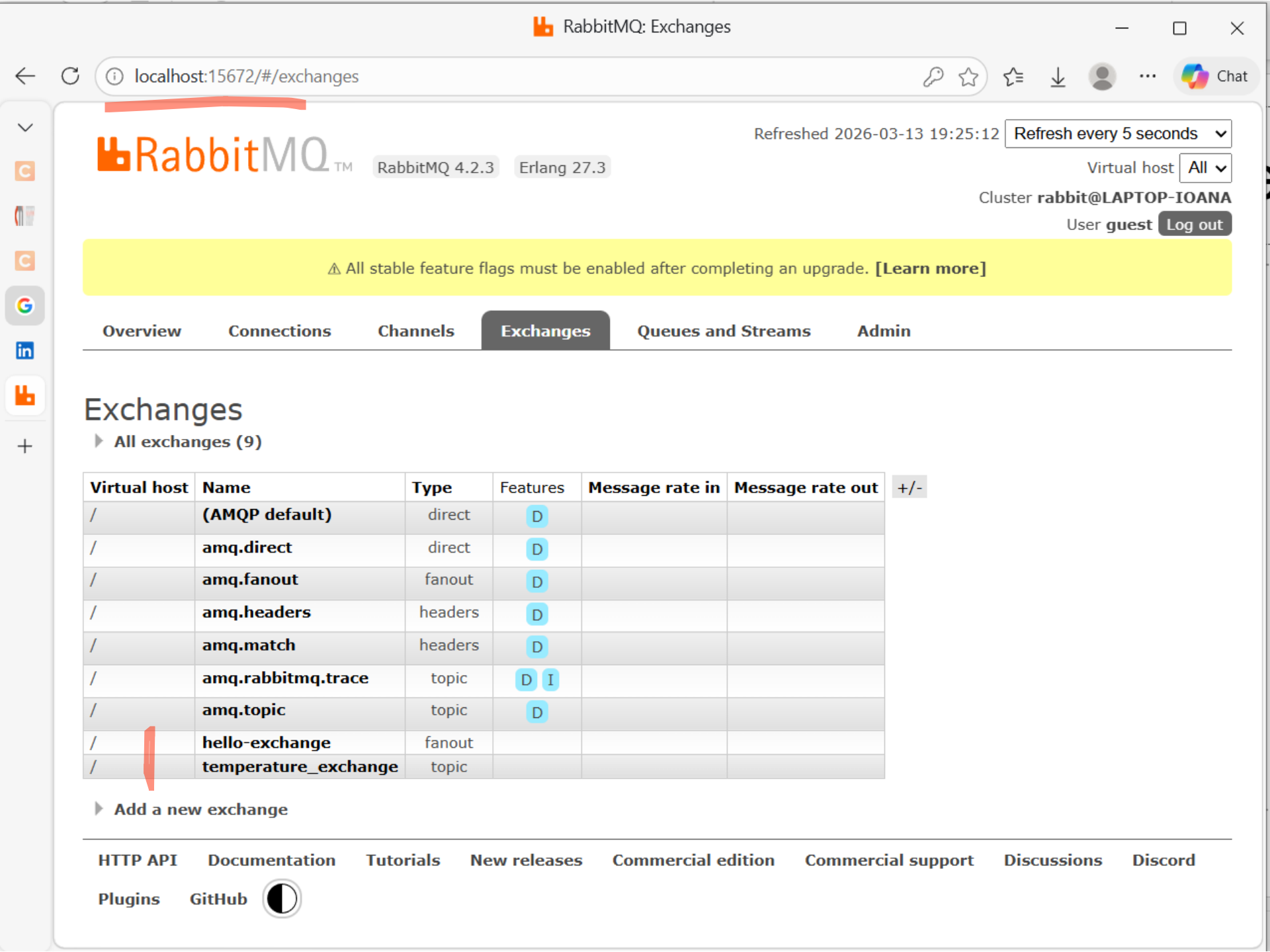
Server Management

Server management operations:

- Users and permissions
- Monitor exchanges, queues, connection

- Options:
  - CLI tools
  - HTTP-based management plugin




RabbitMQ: Exchanges
localhost:15672/#/exchanges
Chat
RabbitMQ
RabbitMQ 4.2.3
Erlang 27.3
Refreshed 2026-03-13 19:25:12
Refresh every 5 seconds
Virtual host All
Cluster rabbit@LAPTOP-IOANA
User guest
Log out
⚠ All stable feature flags must be enabled after completing an upgrade. [Learn more]
Overview
Connections
Channels
Exchanges
Queues and Streams
Admin
Exchanges
All exchanges (9)
Virtual host
Name
Type
Features
Message rate in
Message rate out
+/-
/ (AMQP default) direct D
/ amq.direct direct D
/ amq.fanout fanout D
/ amq.headers headers D
/ amq.match headers D
/ amq.rabbitmq.trace topic D I
/ amq.topic topic D
/ hello-exchange fanout
/ temperature_exchange topic
Add a new exchange
HTTP API
Documentation
Tutorials
New releases
Commercial edition
Commercial support
Discussions
Discord
Plugins
GitHub

Typical RabbitMQ

Producer

- Connect to RabbitMQ
- Obtain a channel
- Declare an exchange
- Create a message
- Publish the message
- Close the channel
- Close the connection

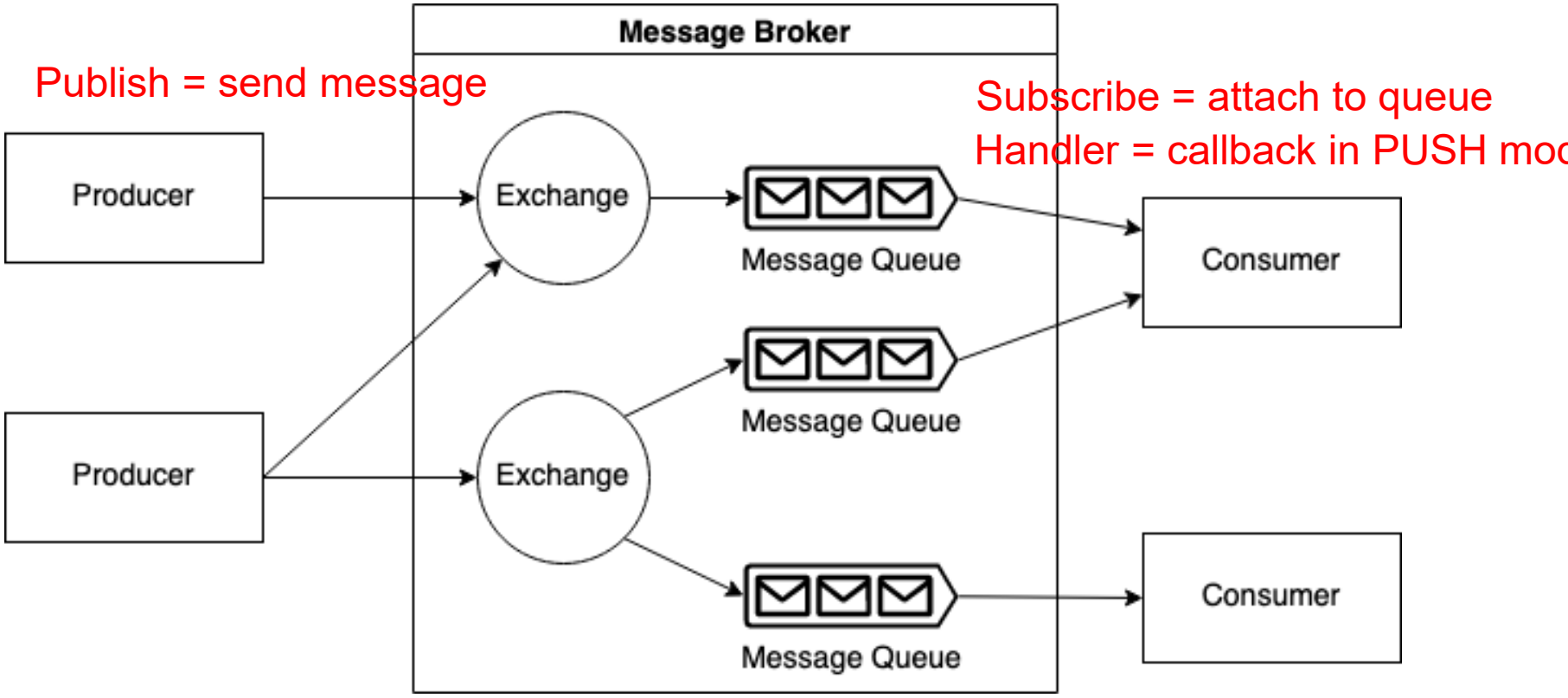
Consumer

- Connect to RabbitMQ
- Obtain a channel
- Declare an exchange
- Declare a queue
- Bind the queue with the exchange
- Consume the messages
- Close the channel
- Close the connection

RabbitMQ Consumer Modes

- A Consumer can receive messages from a queue in two ways:
  - **Callback / Push Model**: The broker pushes messages from the queue to the consumer, and the application code handles them in a callback handler
    - `channel.basicConsume`
  - **Polling / Pull Model**: the application can actively request messages from the queue
    - `channel.basicGet`
- When a queue has multiple consumers, messages received by the queue are served in a round-robin fashion to these consumers. Each message is sent to only one consumer subscribed to the queue

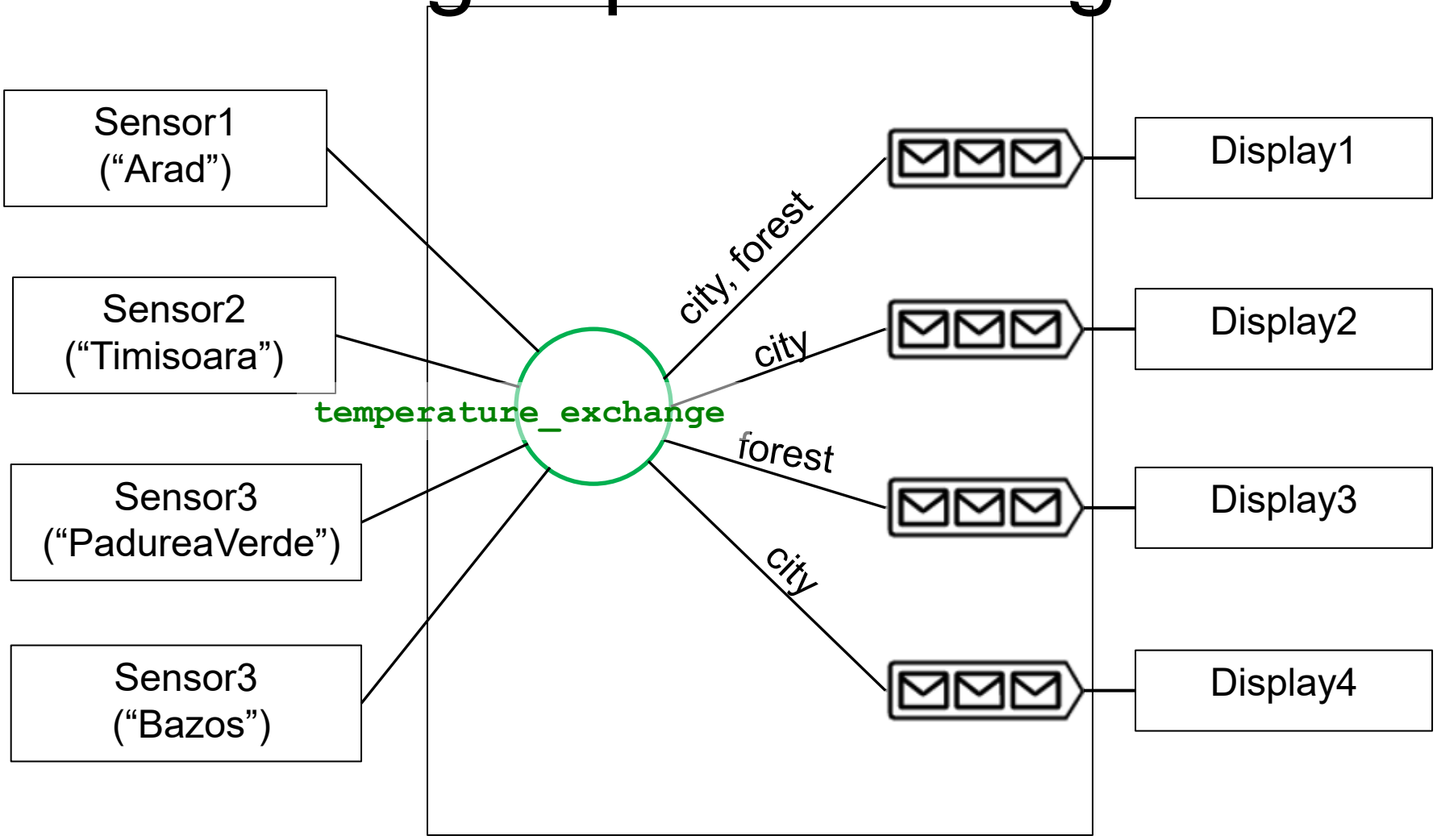
Publish-Subscribe (EventBus) with RabbitMQ



Publish-Subscribe (EventBus) with RabbitMQ

- Subscriber: a consumer attached to a Queue, in Callback/Push Mode
- Producers publish messages to an Exchange and the Exchanges routes messages to Queues.
- Can use Fanout Exchange or Topic Exchange
- Fanout Exchange: all queues bound to the Exchange receive the message
  - In order to “subscribe” to an event type, Subscriber must bind its queue to the corresponding Exchange
  - For every event type there is another Fanout Exchange
- Topic Exchange: only the queues bound with matching topic receive the message
  - In order to “subscribe” to an event type, Subscriber must bind its queue to an Exchange and specify the routing key
  - A single Topic Exchange can be used to publish multiple event types, but using different routing keys

Example: Sensors-Displays using Topic Exchange



Example: Sensors-Displays

SensorProgram

- Args: sensorId location tag
- Connect to RabbitMQ (localhost)
- Obtain a channel
- Declare an exchange (“temperature_exchange”) of type *topic*
- Create a message on topic corresponding to its tag and a random temperature value
- Publish the message on the channel
- Close the channel
- Close the connection

DisplayProgram

- Args: displayName tag1 [tag2 ...]
- Connect to RabbitMQ (localhost)
- Obtain a channel
- Declare an exchange (“temperature_exchange”) of type *topic*
- Declare a queue (a temporary one, only for this connection)
- Bind the queue with the exchange, for all the subscribed topics(tags)
- Sets one DeliverCallback on channel
- Consume the messages
- Close the channel
- Close the connection

```java
package rabbitmq.sensors;

import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.Channel;

public class RabbitMQUtils {
    public static Connection getConnection() throws Exception {
        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost("localhost");
        return factory.newConnection();
    }

    public static final String EXCHANGE_NAME = "temperature_exchange";
}
```

```
public class SensorProgram {
   public static void main(String[] args) throws Exception

     //… some details omitted from slide

     try (Connection conn = RabbitMQUtils.getConnection();
             Channel channel = conn.createChannel()) {

           channel.exchangeDeclare(RabbitMQUtils.EXCHANGE_NAME, "topic");

           for (int i = 0; i < 10; i++) {
               // Generate random temp between 0 and 40
               float value = 0 + random.nextFloat() * 40;
               String message = sensorId + "," + location + "," + value;

               channel.basicPublish(RabbitMQUtils.EXCHANGE_NAME, tag,
                       null, message.getBytes("UTF-8"));

               System.out.println("Sensor " + sensorId + " sent: " +
                       message);

               Thread.sleep(500); // delay between temperature readings
           }
       }
    }
}
```

```
public class DisplayProgram {
    public static void main(String[] args) throws Exception {

//… some details omitted from slide

try (Connection conn = RabbitMQUtils.getConnection();
    Channel channel = conn.createChannel()) {
       channel.exchangeDeclare(RabbitMQUtils.EXCHANGE_NAME, "topic");
       String queueName = channel.queueDeclare().getQueue();

       for (String tag : tags) {
         channel.queueBind(queueName, RabbitMQUtils.EXCHANGE_NAME, tag);
         System.out.println(displayName + " listening for tag: " + tag);
       }

      DeliverCallback deliverCallback = (consumerTag, delivery) -> {
            String message = new String(delivery.getBody(), "UTF-8");
            System.out.println(displayName + " received: " + message);
            };

      channel.basicConsume(queueName, true, deliverCallback, consumerTag
-> { });

      System.out.println(displayName + " is running.");
      while (true) { Thread.sleep(1000);}
      }
    }
}
```

Using Lambda Expressions

```
// lambda expression as handle method of functional interface DeliverCallback

DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        String message = new String(delivery.getBody(), StandardCharsets.UTF_8);
        System.out.println(displayName + " received: " + message);
         };
```

```
// with anonymous inner class implementing the DeliverCallback interface

DeliverCallback deliverCallback = new DeliverCallback() {
    @Override
    public void handle(String consumerTag, Delivery delivery)  {
        String message = new String(delivery.getBody(), StandardCharsets.UTF_8);
        System.out.println(displayName + " received: " + message);
    }
};
```

Message Delivery in Callback Mode

- Message flow:
  - A message arrives in the queue.
  - The RabbitMQ broker sends the message to the client library
  - The client library delivers the message via a channel to **one** registered consumer callback.
- Parallelism rules:
  - Message deliveries on different channels can occur in parallel
  - Message deliveries to multiple consumers on the same channel are handled sequentially in a round-robin order.
  - Example:
    - Scenario1: Client connects to one queue, using 2 channels, one consumer (callback) on each channel: consumers work in parallel
    - Scenatio2: Client connects to one queue, using 1 channel, two consumers (callbacks) on same channel: consumers work sequentially, round robin

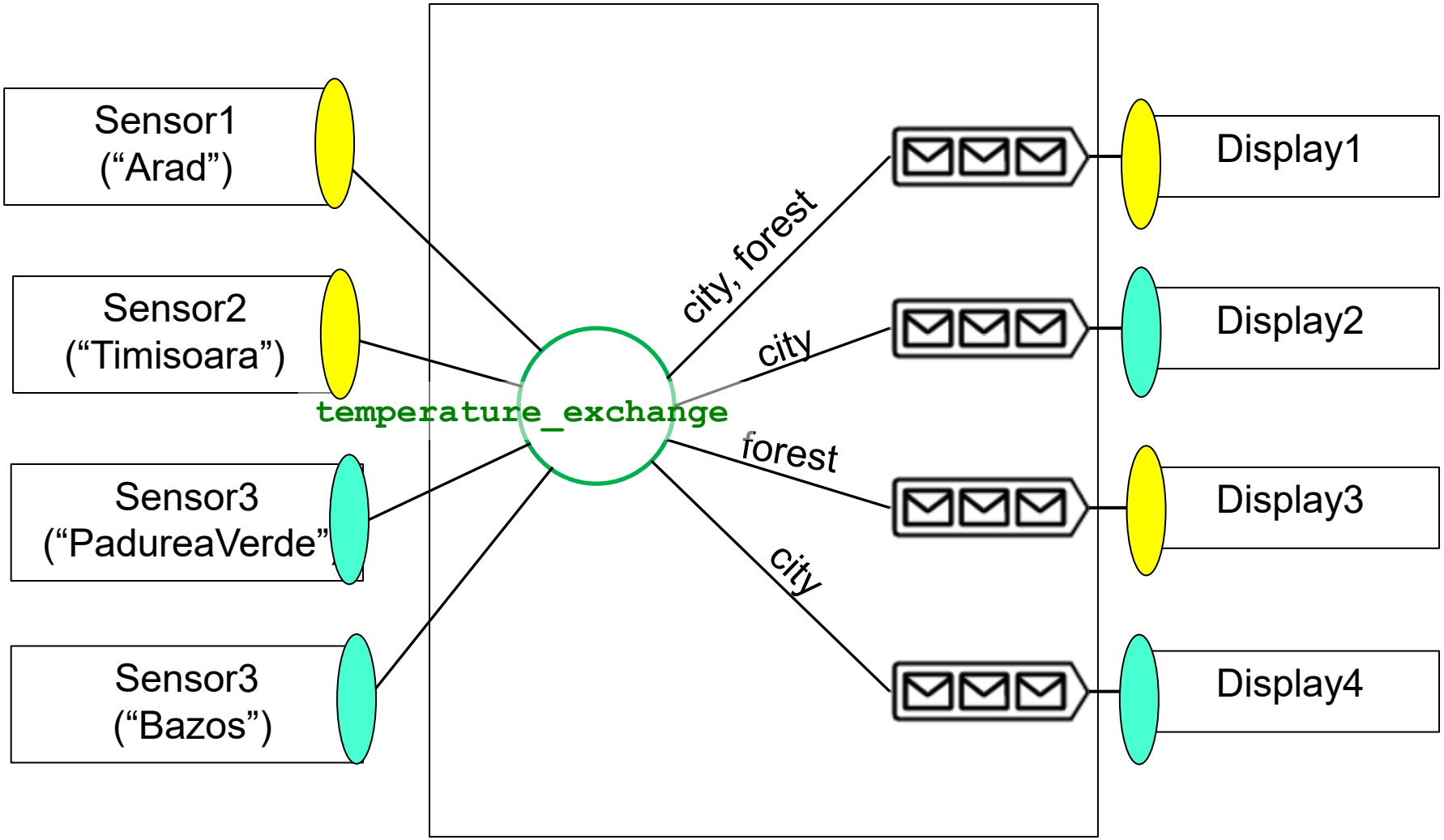
Channel and Consumer Concurrency Issues

- Consumer = one callback handler.
- Channel instances should not be shared between threads, because channels are not thread-safe.
- Callbacks for a single consumer are executed synchronously.
  - The next message for that consumer will not be processed until the current callback finishes. If manual acknowledgments are used, the message must also be acknowledged before the next one is processed.
  - Consequences:
    - Messages are effectively processed one at a time, in order, per consumer.
    - If message processing takes a long time: callback should delegate the long processing to an asynchronous task or worker thread and return from the callback as soon as possible

Using RabbitMQ cross-language

- RabbitMQ and AMPQ are language-independent
- Publishers and subscribers written in different languages can communicate if they use:
  - The same Broker instance
  - The same Exchange (name and type)
  - Same routing keys
- Example: some subscribers (displays) can be written in python, others in java, and all interact with the same publishers
  - the standard Python AMQP client is pika

Example: using different client libraries



```
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost')
)
channel = connection.channel()

EXCHANGE_NAME = "temperature_exchange"

channel.exchange_declare(exchange=EXCHANGE_NAME, exchange_type='topic')

result = channel.queue_declare(queue='', exclusive=True)
queue_name = result.method.queue

for tag in tags:
    channel.queue_bind(exchange=EXCHANGE_NAME,
                       queue=queue_name,
                       routing_key=tag)
    print(f"{display_name} listening for tag: {tag}")

print(f"{display_name} is running. Press Ctrl+C to exit.")

def callback(ch, method, properties, body):
    print(f"{display_name} received: {body.decode('utf-8')}")

channel.basic_consume(queue=queue_name,
                      on_message_callback=callback,
                      auto_ack=True)

channel.start_consuming()
```

Source Code of Examples

- Install RabbitMQ: link
- Source code: rabbitmq1.zip