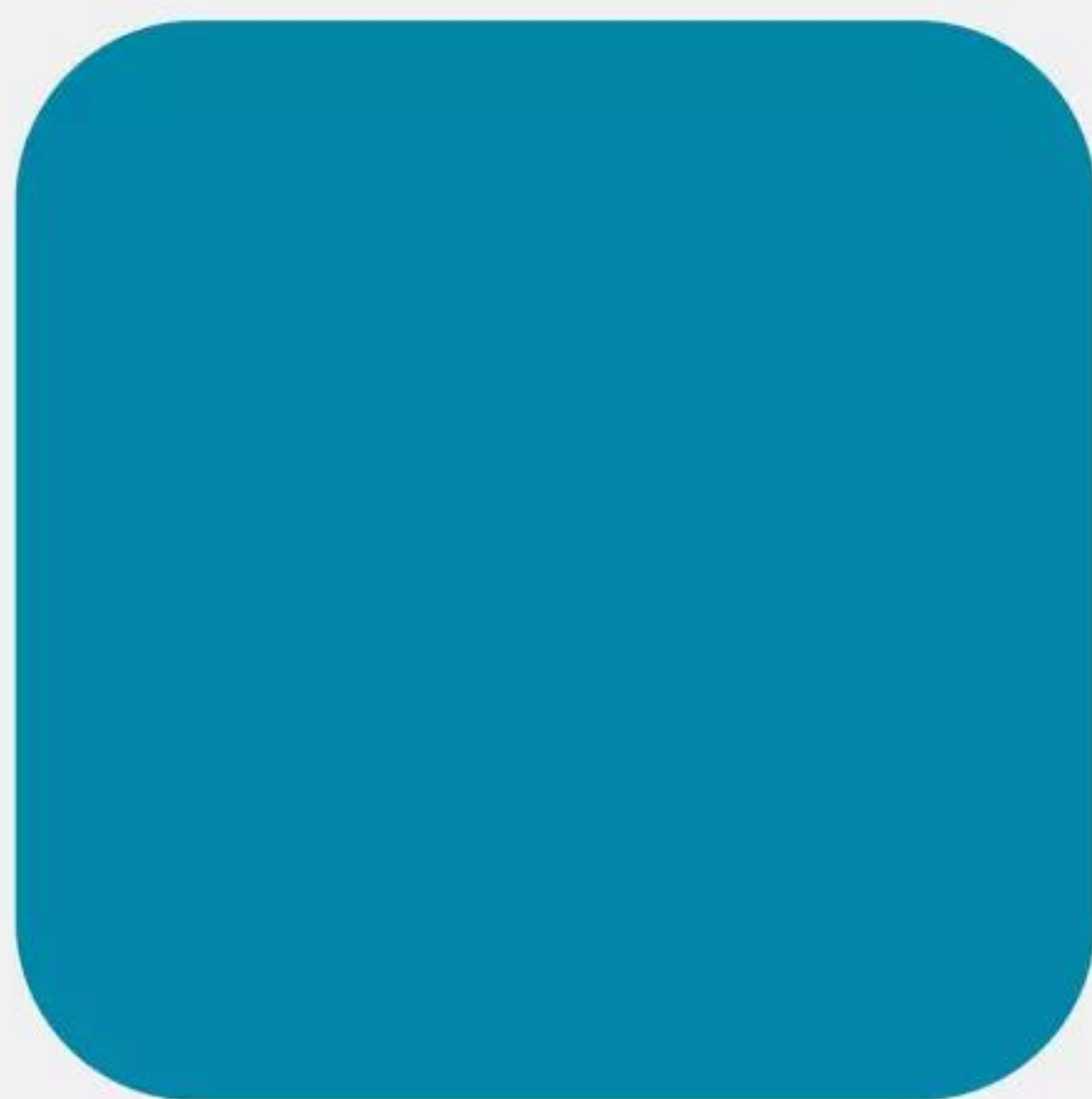


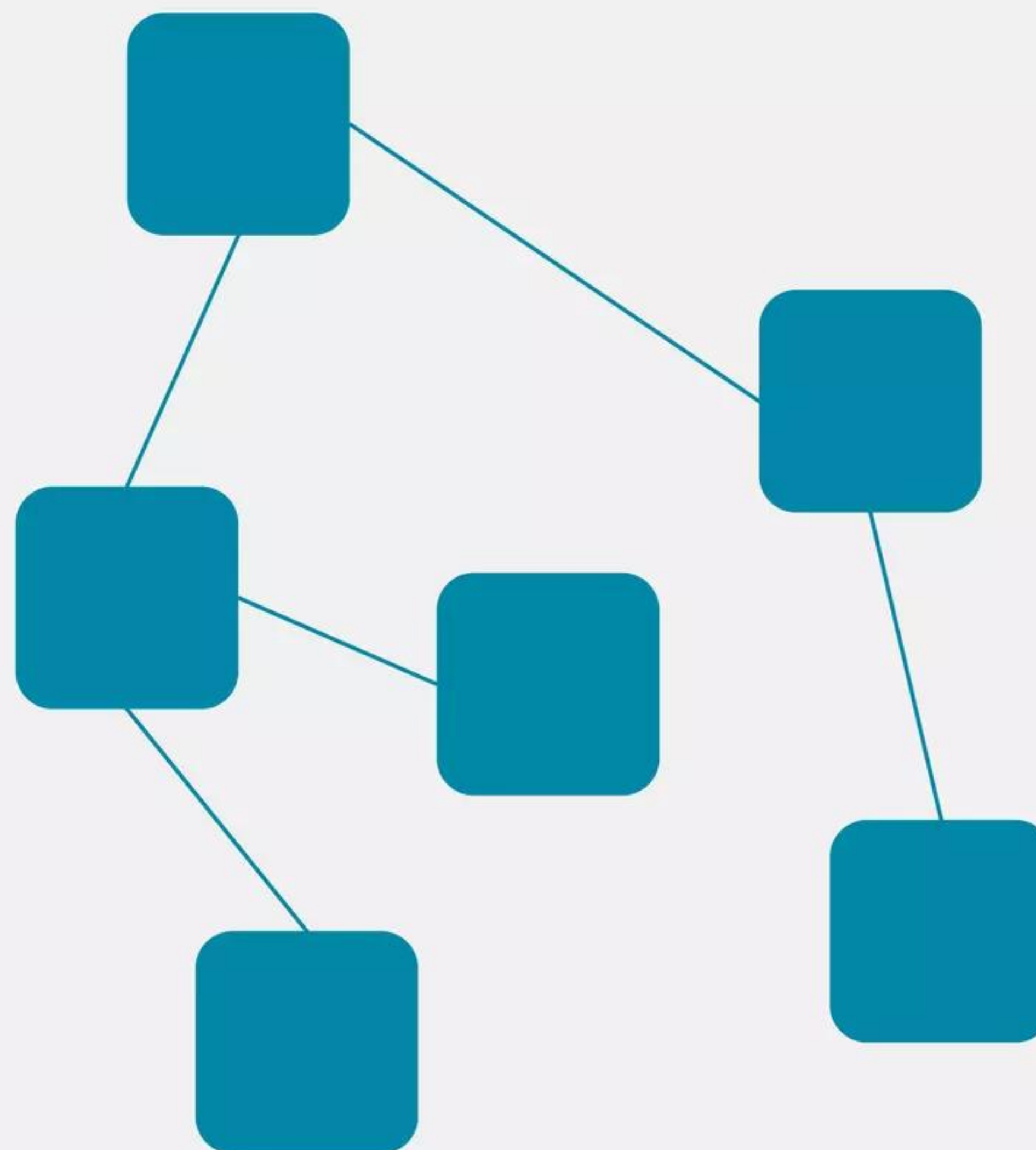
Why Actor-Based Systems Are The Best For Microservices

Yaroslav Tkachenko
Senior Software Engineer, Lead at Mobify
@sap1ens





Monolith



Microservices



“Easy” way – HTTP / RPC API

- **Destination** – where to send request?
 - Service discovery
 - Tight coupling
- **Time** – expect reply right away?
- **Failure** – always expect success?
 - Retries and failure handling
 - Back-pressure / circuit breakers

But wait...

How about Netflix, Twitter, etc.?

Yes, **it's possible** to build reliable systems
with HTTP/RPC APIs! But:

- Are you ready to invest a lot of resources
- Do you really need it?

Resilient Protocols

DEPEND ON

ASYNCHRONOUS COMMUNICATION

EVENTUAL CONSISTENCY

- **ARE TOLERANT TO**
 - MESSAGE LOSS
 - MESSAGE REORDERING
 - MESSAGE DUPLICATION
- **EMBRACE ACID 2.0**
 - ASSOCIATIVE
 - COMMUTATIVE
 - IDEMPOTENT
 - DISTRIBUTED

Reactive Revealed: Resiliency, Failures vs Errors, Isolation, Delegation and Replication by **Jonas Boner**

Pattern Matching

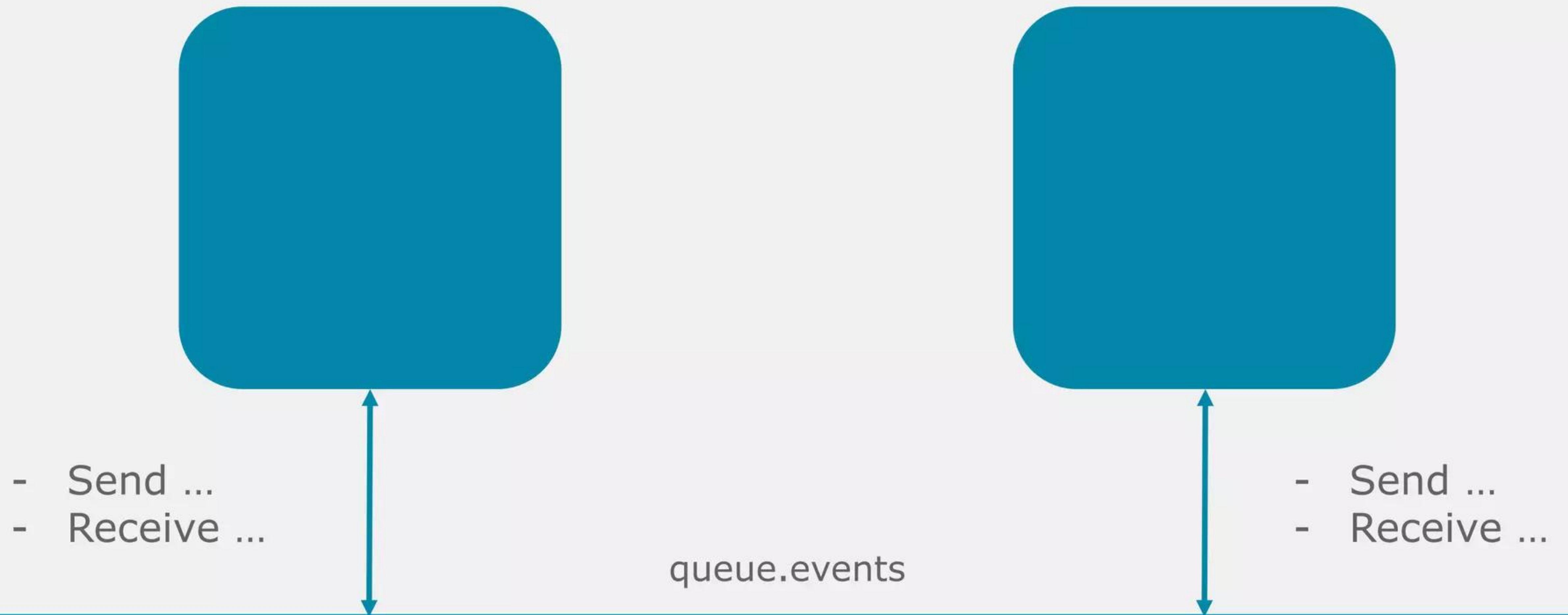
Service discovery is an anti-pattern.
Instead, make messages first-class citizens.
Use message data to define patterns,
and these patterns define a language!

Transport Independence

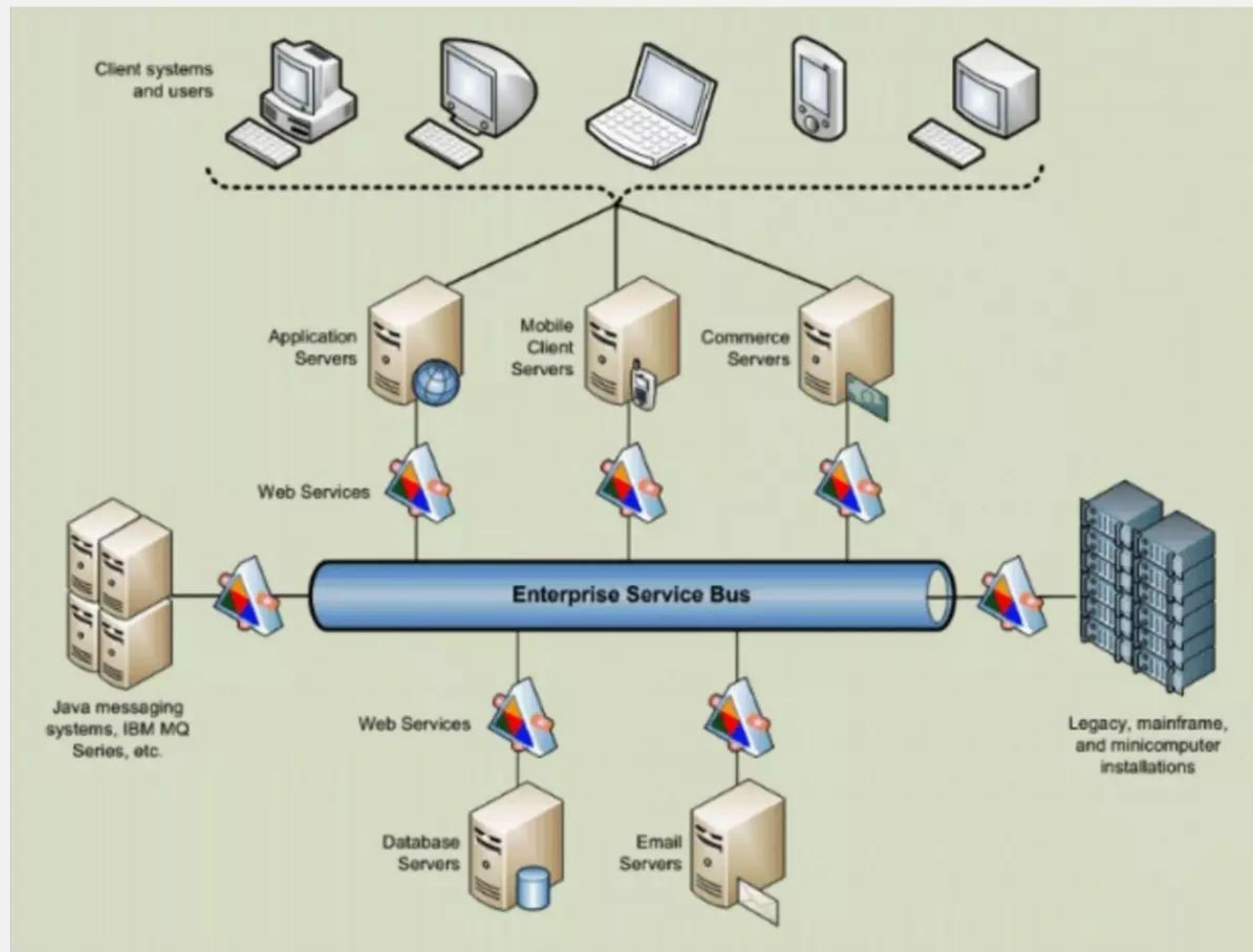
Services should not know about each other,
or how to send messages.
Services are fully defined by:
message patterns that they recognise, and
message patterns that they emit.

So, let's try to:

- Use **message queues** for communication
- Use **actors** as publishers and subscribers

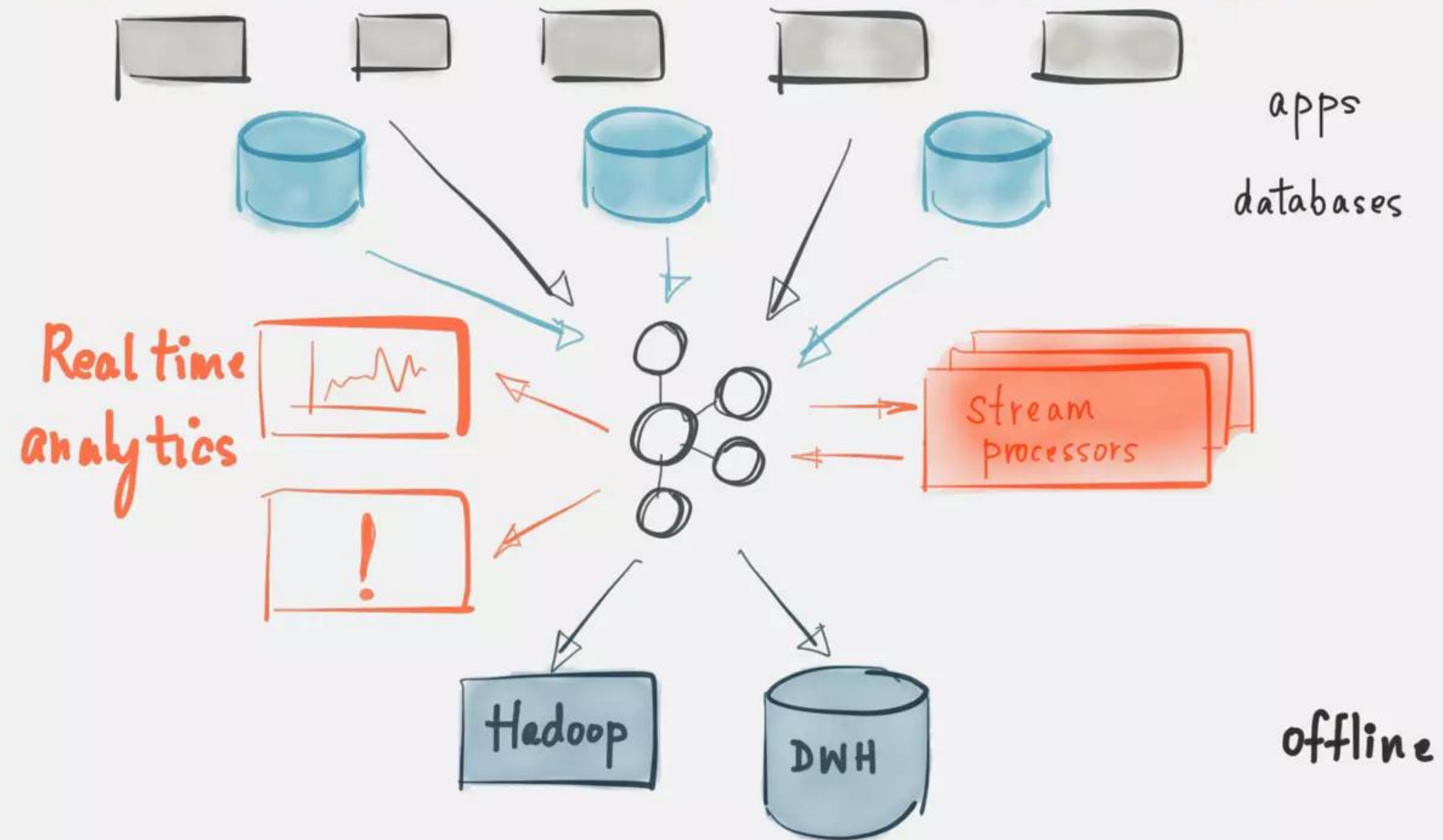


Another way - Messaging



Enterprise Service Bus?

STREAM DATA PLATFORM



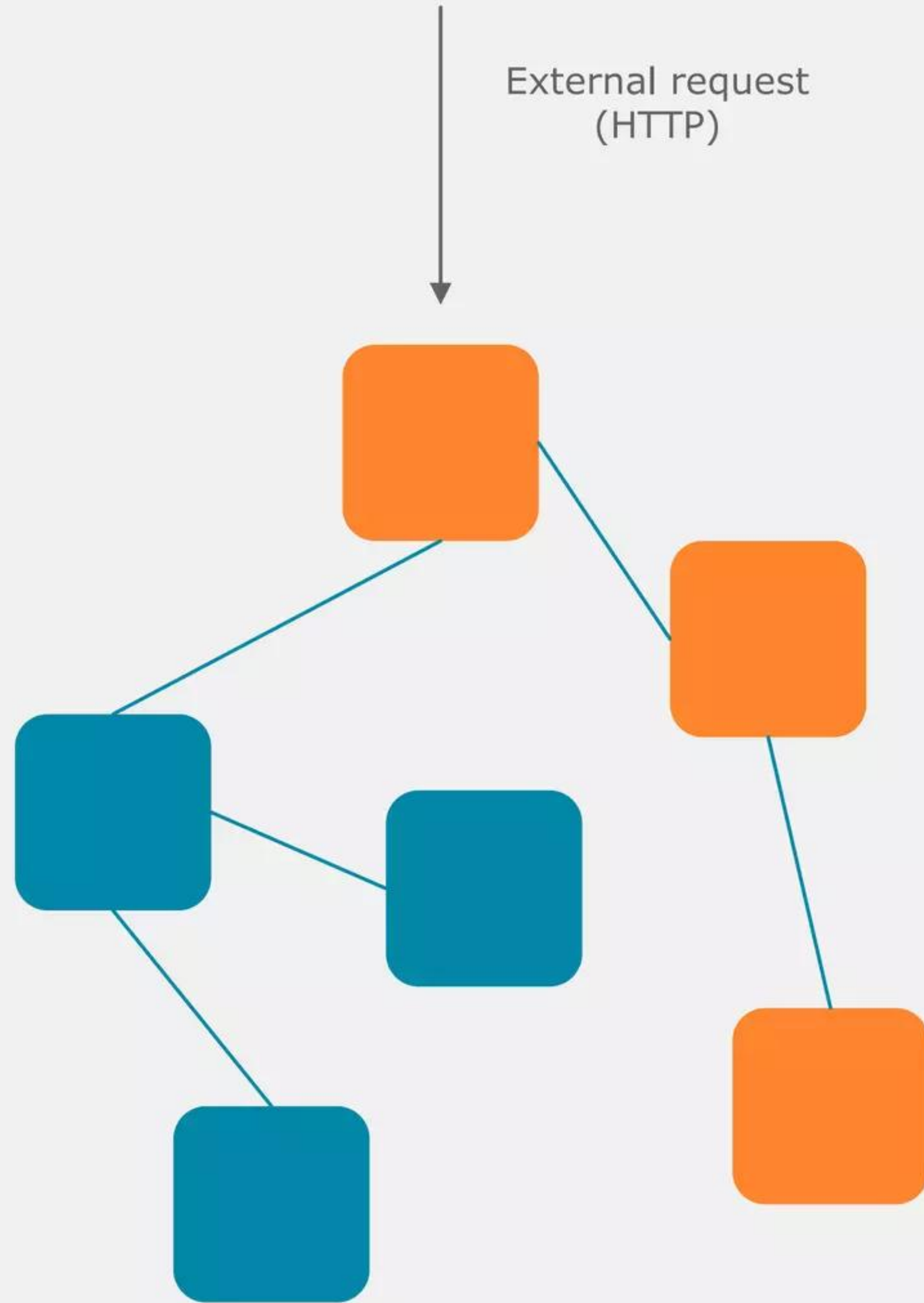
Kafka Connect

So, why message queues now?

- **Not ESBs!**
- Lightweight pub/sub instead
- Low coupling
- Promotes eventual consistency
- Highly scalable and available
- Built-in discovery, retries, tracing, etc.
- Serverless!

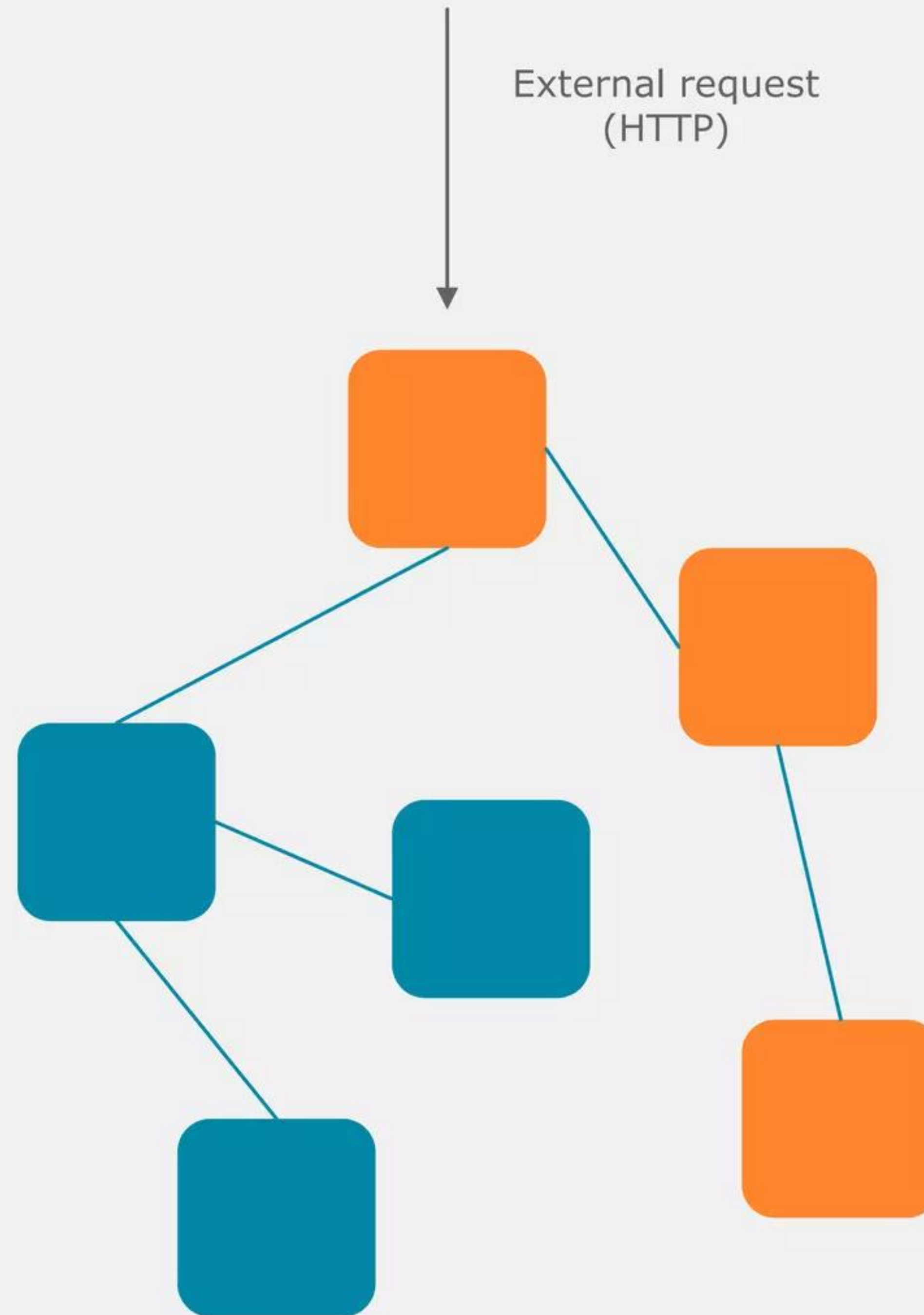
By the way, I don't think that ALL your communication should happen through the message queues. But it's something that you should strive for!

External request
(HTTP)



Write

Always
asynchronous
using messages

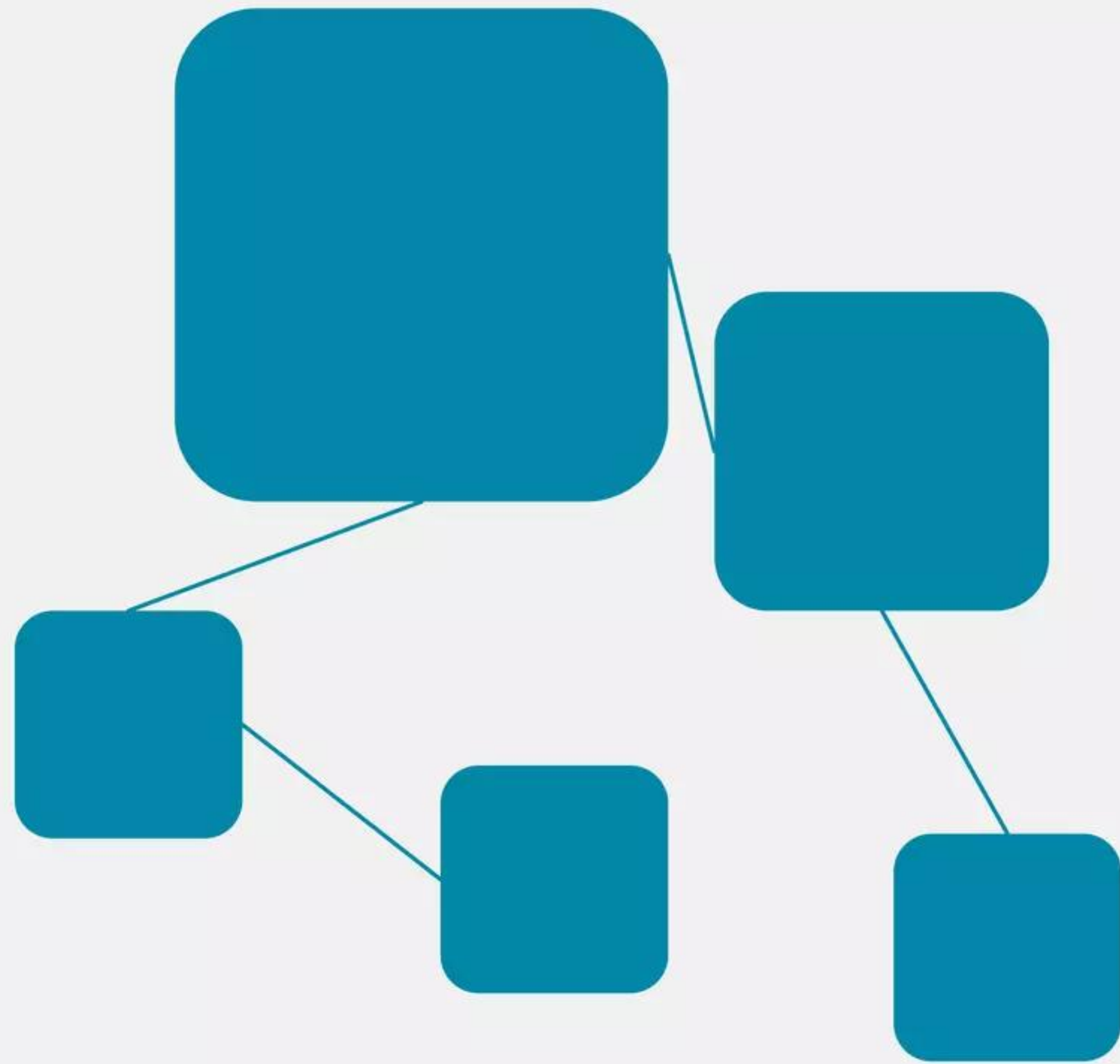


Read

Alternatives to
blocking call:

- Caching
- Denormalization
- Different routing

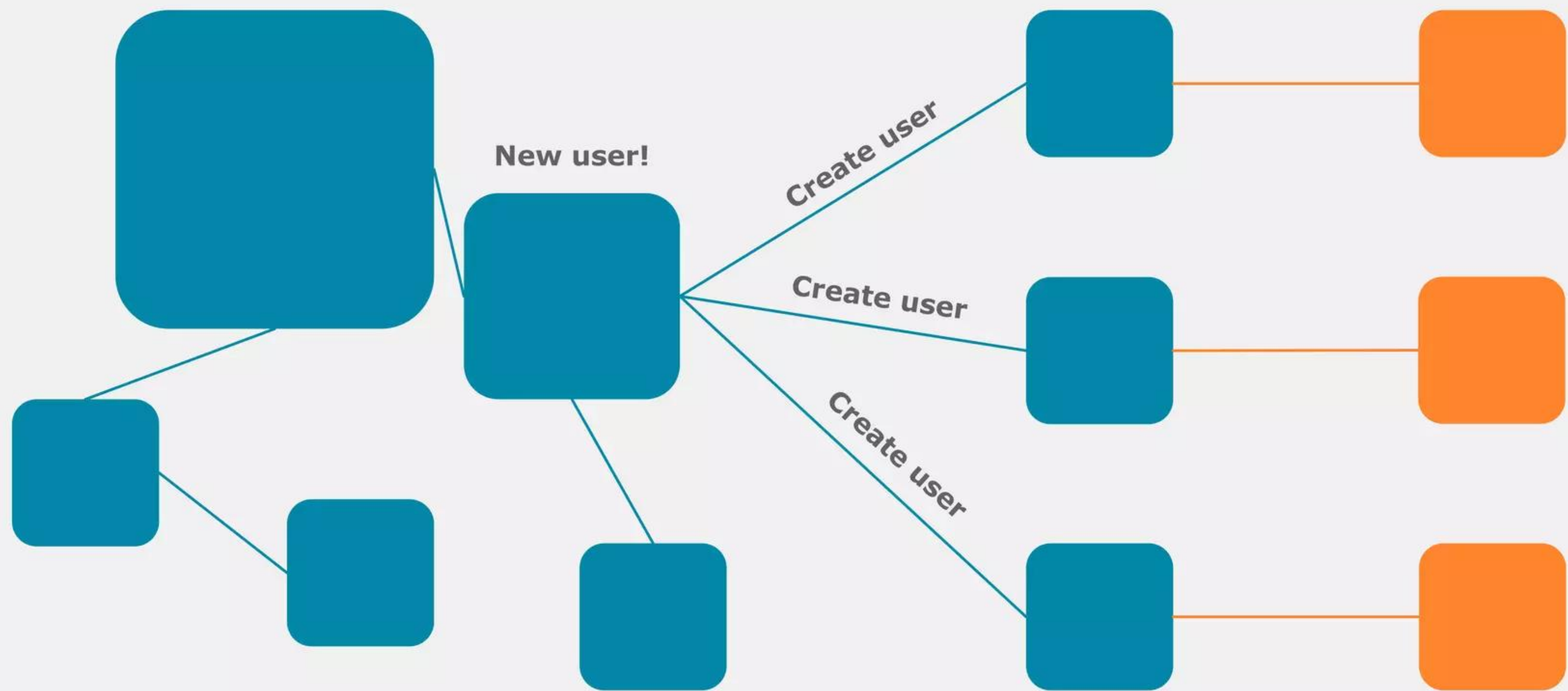
Example: integrations



"Macroservices"

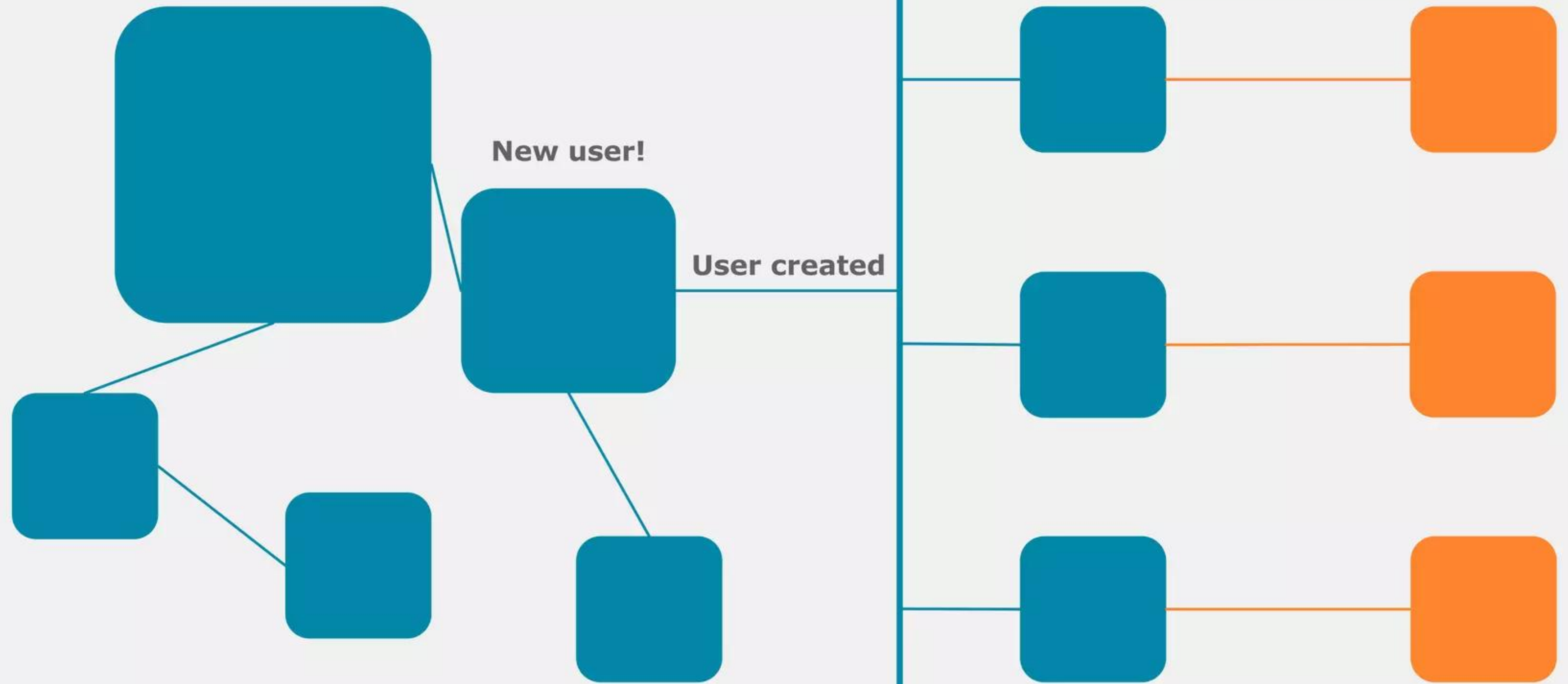


External APIs



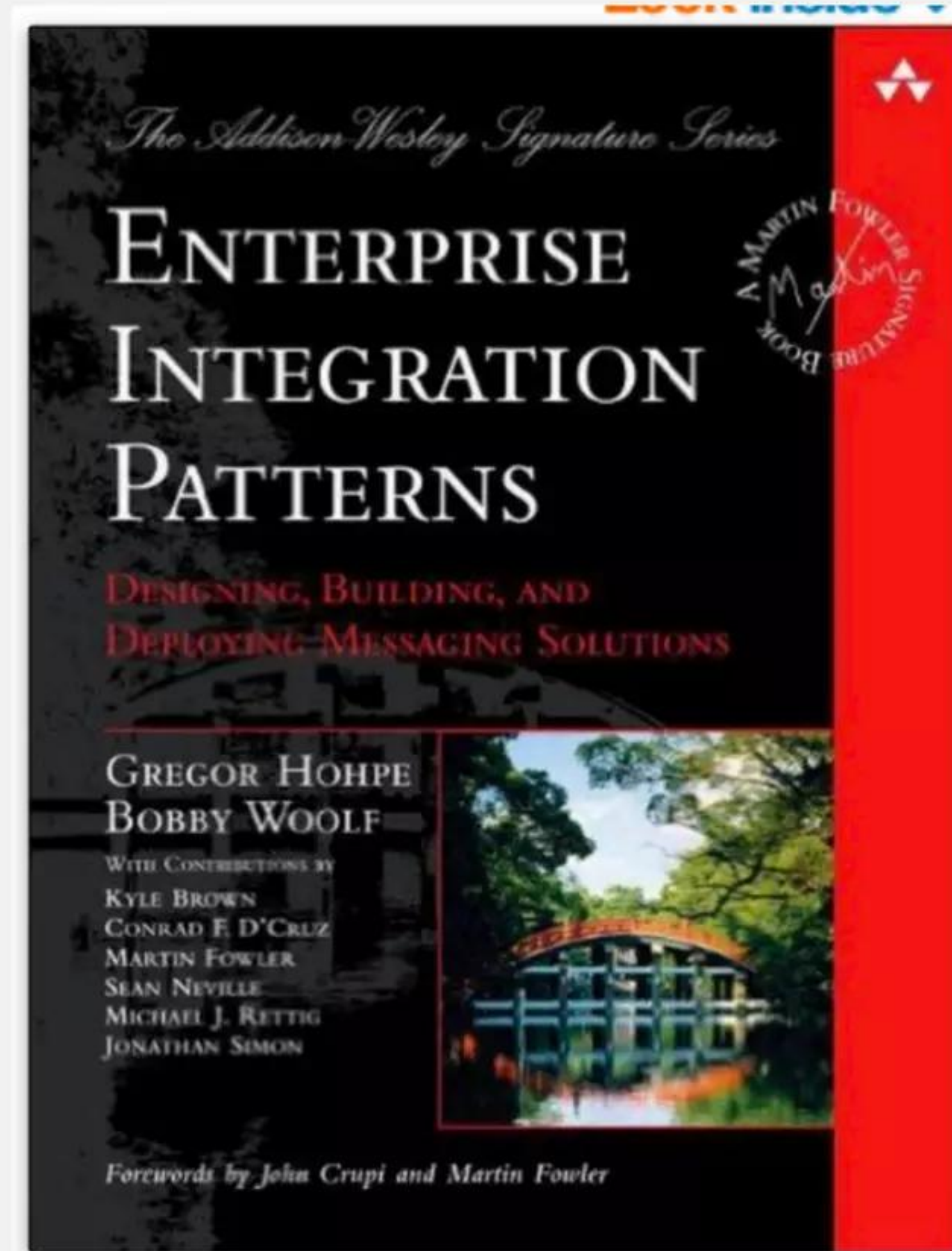
"Macroservices"

External APIs

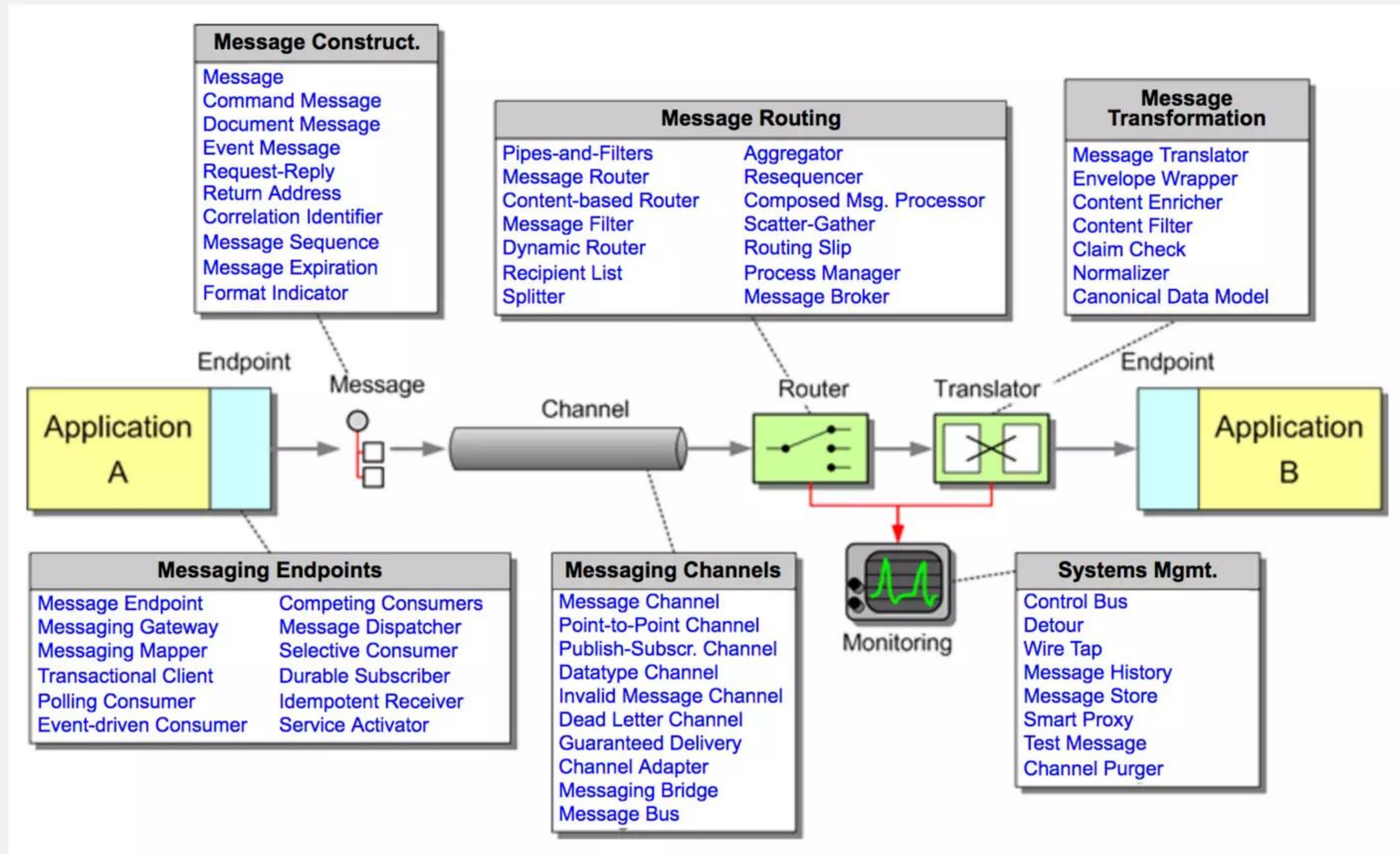


"Macroservices"

External APIs



EIP was published in **2003** and it contains 65 patterns



Enterprise Integration Patterns

Patterns from the example:

- Publish – Subscribe Channel (“Broadcast”)
- Durable Subscriber
- Idempotent Receiver

So, have I convinced you that messaging patterns are great?

But why actors? I can just use a message queue and a bunch of consumers/producers

It's all about **semantics**

An **actor** is a computational entity that, in response to a message it receives, can concurrently:

- send a finite number of messages to other actors
- create a finite number of new actors
- designate the behavior to be used for the next message it receives.

Actors are known for concurrency, but let's focus on message passing instead

```
class MyActor extends Actor with ActorLogging {  
  def receive = {  
    case msg => {  
      log.info(s"received $msg")  
  
      sender() ! "got it"  
    }  
  }  
}
```

```
loop() ->  
  receive  
    {From, Msg} ->  
      io:format("received ~p~n", [Msg]),  
  
      From ! "got it";  
  end.
```

Actors in Scala/Akka and Erlang



Actor



Message



Actor



Actor



Message queue



Actor

```

public class Server implements MessageListener {
    private static int ackMode;
    private static String messageQueueName;
    private static String messageBrokerUrl;

    private Session session;
    private boolean transacted = false;
    private MessageProducer replyProducer;
    private MessageProtocol messageProtocol;

    static {
        messageBrokerUrl = "tcp://localhost:61616";
        messageQueueName = "client.messages";
        ackMode = Session.AUTO_ACKNOWLEDGE;
    }

    public Server() {
        try {
            //This message broker is embedded
            BrokerService broker = new BrokerService();
            broker.setPersistent(false);
            broker.setUseJmx(false);
            broker.addConnector(messageBrokerUrl);
            broker.start();
        } catch (Exception e) {
            //Handle the exception appropriately
        }

        //Delegating the handling of messages to another class, instantiate it before setting up JMS so it
        //is ready to handle messages
        this.messageProtocol = new MessageProtocol();
        this.setupMessageQueueConsumer();
    }

    private void setupMessageQueueConsumer() {
        ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(messageBrokerUrl);
        Connection connection;
        try {
            connection = connectionFactory.createConnection();
            connection.start();
            this.session = connection.createSession(this.transacted, ackMode);
            Destination adminQueue = this.session.createQueue(messageQueueName);

            //Setup a message producer to respond to messages from clients, we will get the destination
            //to send to from the JMSReplyTo header field from a Message
            this.replyProducer = this.session.createProducer(null);
            this.replyProducer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

            //Set up a consumer to consume messages off of the admin queue
            MessageConsumer consumer = this.session.createConsumer(adminQueue);
            consumer.setMessageListener(this);
        } catch (JMSException e) {
            //Handle the exception appropriately
        }
    }

    public void onMessage(Message message) {
        try {
            TextMessage response = this.session.createTextMessage();
            if (message instanceof TextMessage) {
                TextMessage txtMsg = (TextMessage) message;
                String messageText = txtMsg.getText();
                response.setText(this.messageProtocol.handleProtocolMessage(messageText));
            }

            //Set the correlation ID from the received message to be the correlation id of the response message
            //this lets the client identify which message this is a response to if it has more than
            //one outstanding message to the server
            response.setJMSCorrelationID(message.getJMSCorrelationID());

            //Send the response to the Destination specified by the JMSReplyTo field of the received message,
            //this is presumably a temporary queue created by the client
            this.replyProducer.send(message.getJMSReplyTo(), response);
        } catch (JMSException e) {
            //Handle the exception appropriately
        }
    }

    public static void main(String[] args) {
        new Server();
    }
}

```

ActiveMQ message listener in Java (just kidding)

```
public class Server implements MessageListener {
    public void onMessage(Message message) {
        try {
            TextMessage response = this.session.createTextMessage();
            if (message instanceof TextMessage) {
                TextMessage txtMsg = (TextMessage) message;
                String messageText = txtMsg.getText();
                response.setText(
                    this.messageProtocol.handleProtocolMessage(messageText)
                );
            }

            this.replyProducer.send(message.getJMSReplyTo(), response);
        } catch (JMSException e) { // ... }
    }
}
```

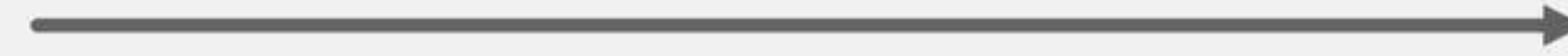
ActiveMQ message listener in Java (zoom-in)

```
class CustomerService extends Actor with ActorLogging with Consumer {  
  val camel = CamelExtension(system)  
  
  camel.context.addComponent("activemq", ActiveMQComponent.activeMQComponent(  
    "tcp://localhost:61616"  
  ))  
  
  def endpointUri = "activemq:topic:events"  
  
  def receive = {  
    case e: CamelMessage => {  
      sender() ! "SomeMessage"  
    }  
  }  
}
```

Actor-based (Akka) message listener in Scala + Camel

???

Receiver Handler



onMessage(Message)

VS

Sender Actor

Receiver Actor



Mailbox

Message

I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages...

OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.

Alan Kay, one of the fathers of the idea
of object-oriented programming

Actors over message handlers:

- Semantically natural message processing that **“feels right”**
- No difference between local and remote communication
- Straightforward state management
- Mailbox, routing and back-pressure
- Built-in failure handling (supervisors)

Actors in general:

- Simple and high-level abstraction for distribution, concurrency and parallelism
- Asynchronous, non-blocking and highly performant message-driven programming model
- Very lightweight message-driven processes

REACTIVE
MESSAGING
PATTERNS
with the
ACTOR MODEL

APPLICATIONS AND INTEGRATION
IN **SCALA** AND **AKKA**

V A U G H N V E R N O N

Foreword by Jonas Bonér, Founder of the Akka Project

Instead of a summary:

Everything is a trade-off!

Questions?

sap1ens.com

@sap1ens