



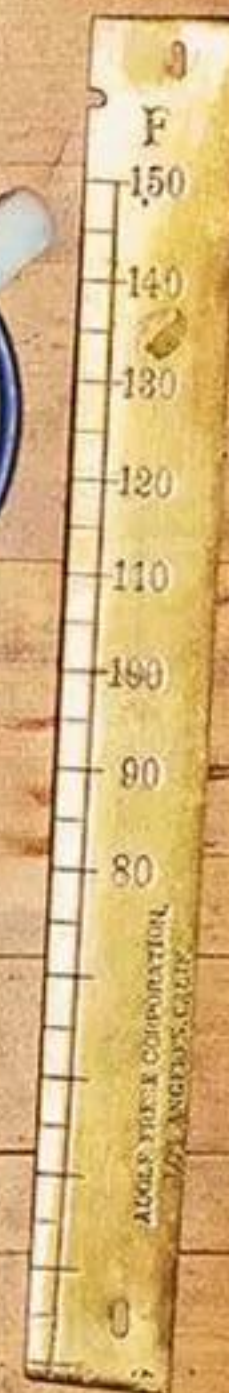
Why actor-based systems are the best for microservices



Yaroslav Tkachenko
@sap1ens

Director of Engineering, Platform at Bench Accounting

Bench^π





Agenda

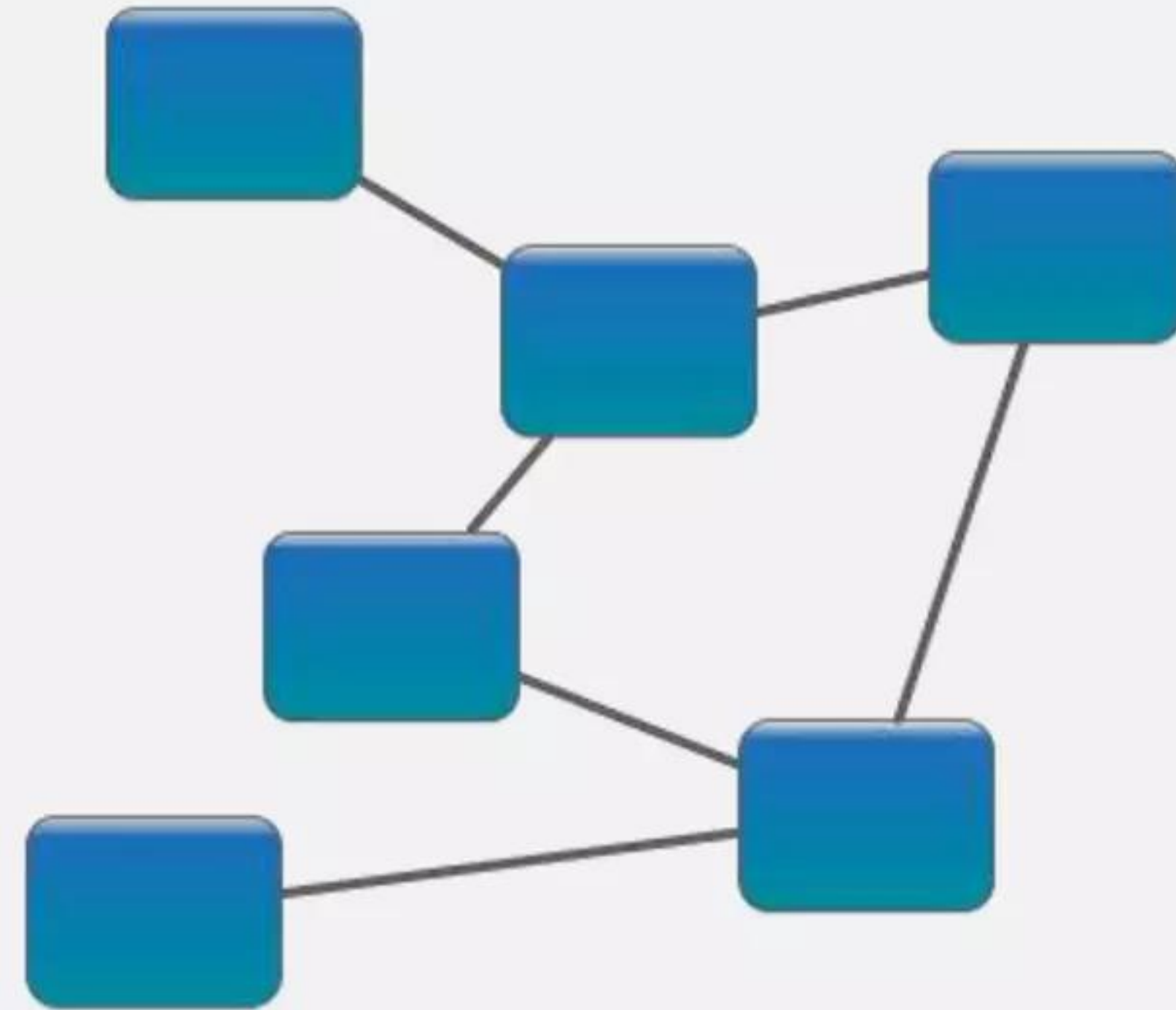
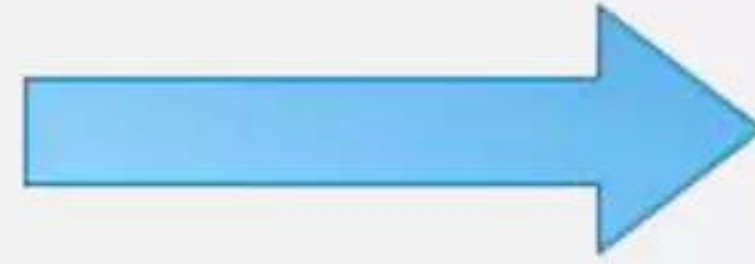
- Microservices
- Messaging for microservices
- EIP and messaging
- Actors and messages
- All combined... and more



Microservices



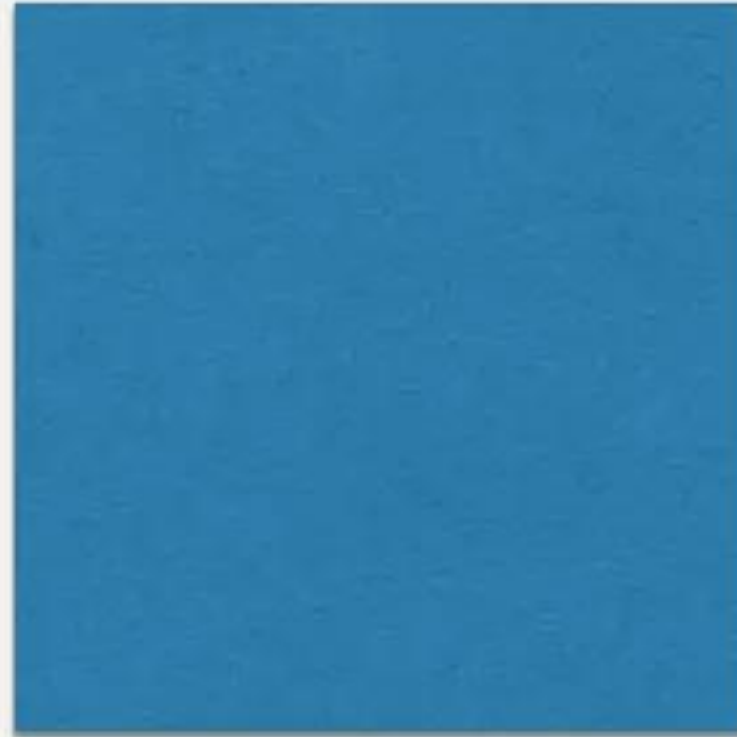
Monolith



Microservices



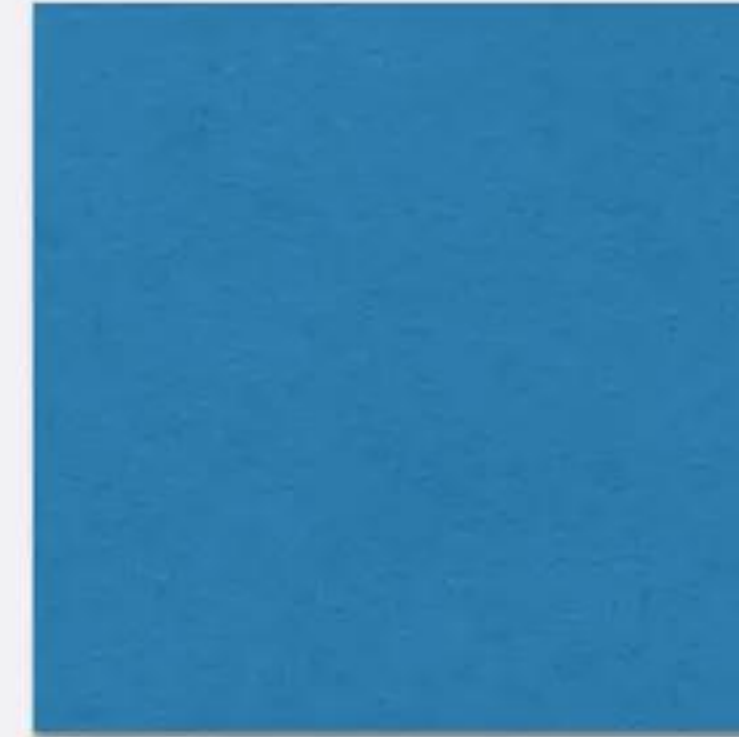
Microservices



Service A



HTTP API / RPC



Service B

“Easy” way



Microservices

Challenges with HTTP API / RPC:

- Service discovery
- Retries for failures
- Routing and load balancing
- Webhooks
- Tracing
- Versions
- ...



Microservices

But...



Messaging for microservices

We should use **only** messaging for communication between microservices



Messaging for microservices

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 Repli



Messaging for microservices

Pattern Matching

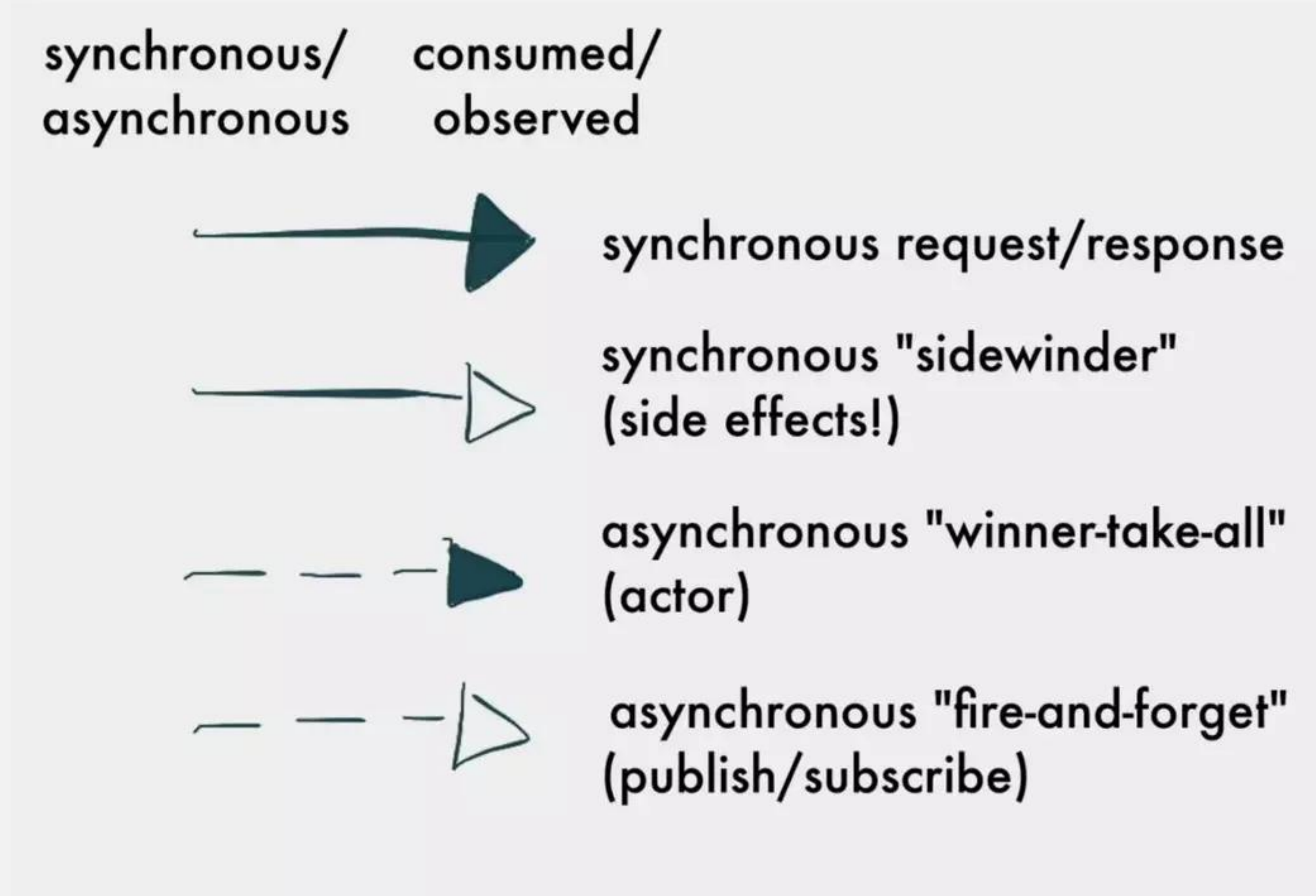
Service discovery is an anti-pattern.
Instead, make messages first-class citizens.
Use message data to define patterns,
and these patterns define a language!

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.



Messaging for microservices



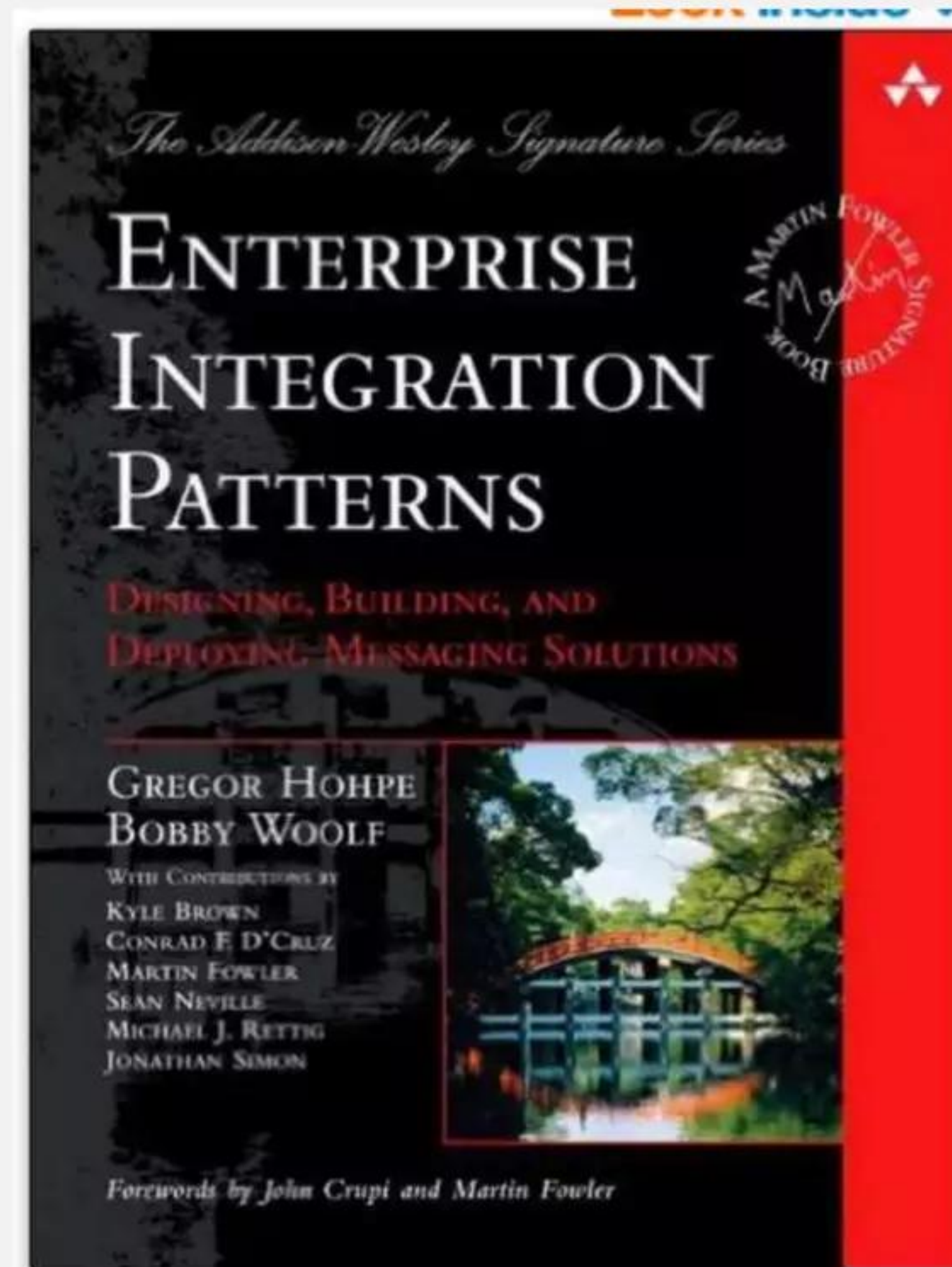
Surviving Micro-services by **Richard Rodger**



EIP and messaging



EIP and messaging





EIP and messaging

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



EIP and messaging

Message Channel
Message
Pipes and Filters
Message Router
Message Translator
Message Endpoint
Point-to-Point Channel
Publish-Subscribe Channel
Datatype Channel
Invalid Message Channel
Dead Letter Channel
Guaranteed Delivery
Channel Adapter
Messaging Bridge
Message Bus
Command Message
Document Message
Event Message
Request-Reply
Return Address
Correlation Identifier
Message Sequence
Message Expiration
Format Indicator
Introduction to Message Routing
Content-Based Router
Message Filter
Dynamic Router
Recipient List
Splitter
Aggregator

Resequencer
Composed Message Processor
Scatter-Gather
Routing Slip
Process Manager
Message Broker
Envelope Wrapper
Content Enricher
Content Filter
Claim Check
Normalizer
Canonical Data Model
Messaging Gateway
Messaging Mapper
Transactional Client
Polling Consumer
Event-Driven Consumer
Competing Consumers
Message Dispatcher
Selective Consumer
Durable Subscriber
Idempotent Receiver
Service Activator
Control Bus
Detour
Wire Tap
Message History
Message Store
Smart Proxy
Test Message
Channel Purger



Actors and messages



Actors and messages

So, I convinced you that messaging patterns are great. Right?



Actors and messages

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



Actors and messages

Sure! But let's compare



Actors and messages

```
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



Actors and messages

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

ActiveMQ message listener in Java (zoom-in)



Actors and messages

```
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



Actors and messages

Btw, some magic is provided by: **Apache Camel**

“The most unknown coolest library out there”

JM



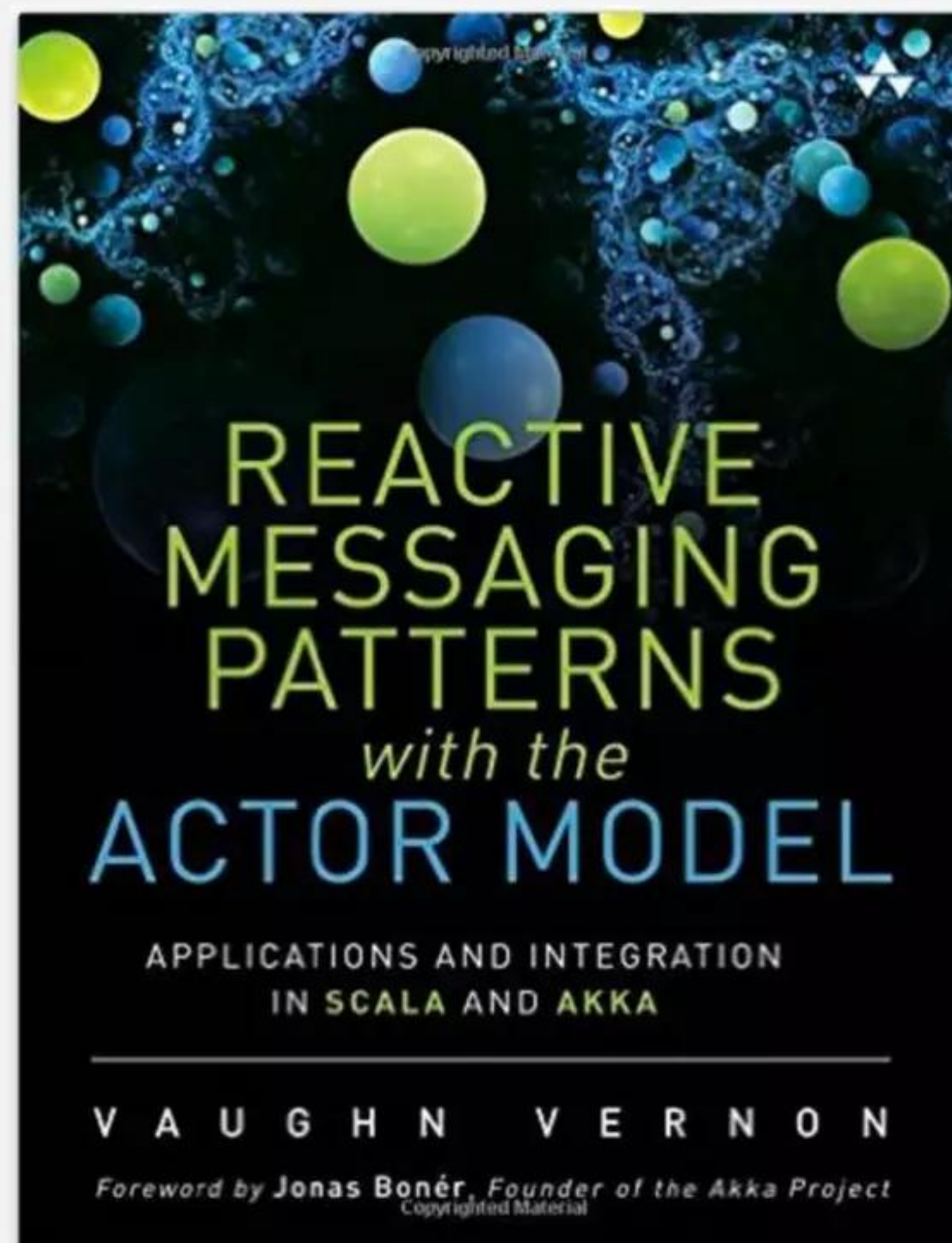
Actors and messages

Actors give you:

- Simple and high-level abstractions for distribution, concurrency and parallelism
- Asynchronous, non-blocking and highly performant message-driven programming model
- Very lightweight event-driven processes (several million actors per GB of heap memory)



Actors and messages





All combined... and more

Summary:

- It's easy to build microservices from scratch using messaging patterns
- All these patterns are well-known, tested and documented
- Actor model gives very simple abstractions to process and send messages
- Bonus: all kinds of routing models, back-pressure, reliable queues, broadcasts



Reading list

<http://microservices.io>

<http://www.enterpriseintegrationpatterns.com/patterns/messaging/>

<https://www.amazon.ca/Reactive-Messaging-Patterns-Actor-Model/dp/0133846830>

<https://www.youtube.com/watch?v=0pfghZxIFSg>

<https://www.youtube.com/watch?v=JvbUF33sKf8>

<http://akka.io>

<http://camel.apache.org>