Illinois Java Users Group 2016

Reactive Architecture Patterns

Using Java and Messaging

Mark Richards

Independent Consultant

Hands-on Software Architect

Published Author / Conference Speaker

www.wmrichards.com

Author of *Software Architecture Fundamentals Video Series* (O'Reilly)
Author of Microservices vs. Service-Oriented Architecture (O'Reilly)
Author of Microservices AntiPatterns and Pitfalls (O'Reilly)
Author of *Enterprise Messaging Video Series* (O'Reilly)
Author of *Java Message Service 2nd Edition* (O'Reilly)

agenda

reactive architecture overview

channel monitor pattern

consumer supervisor pattern

producer control flow pattern

thread delegate pattern

workflow event pattern

combining patterns

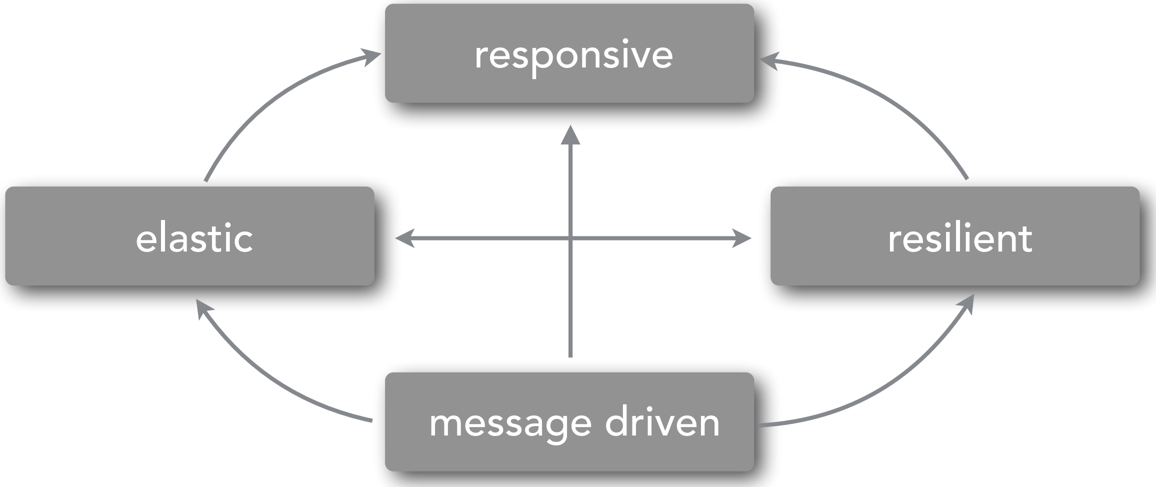
source code

https://github.com/wmr513/reactive

Reactive Architecture Overview

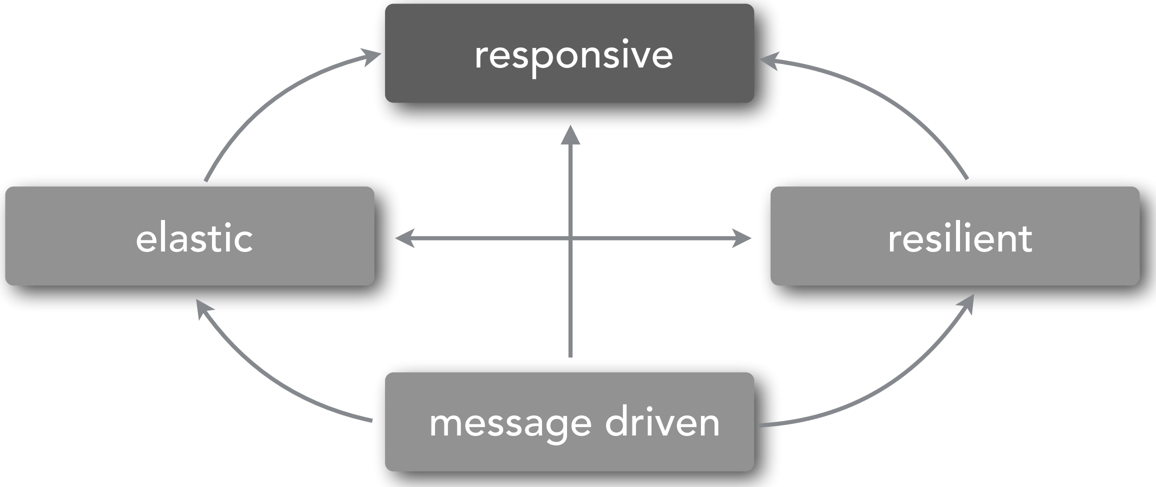
reactive architecture

reactive manifesto



reactive architecture

reactive manifesto

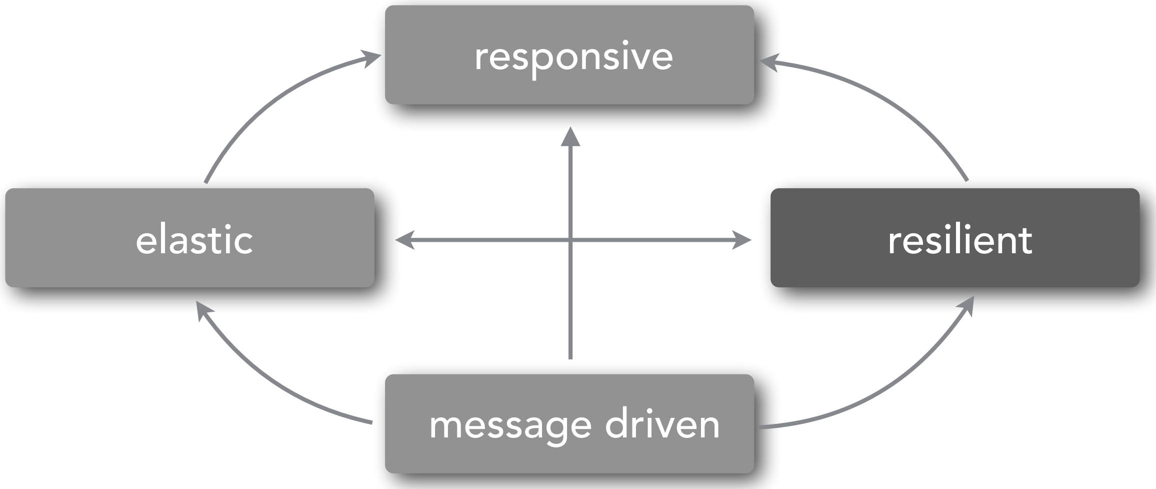


the system responds in a consistent, rapid, and timely manner whenever possible

how the system reacts to users

reactive architecture

reactive manifesto

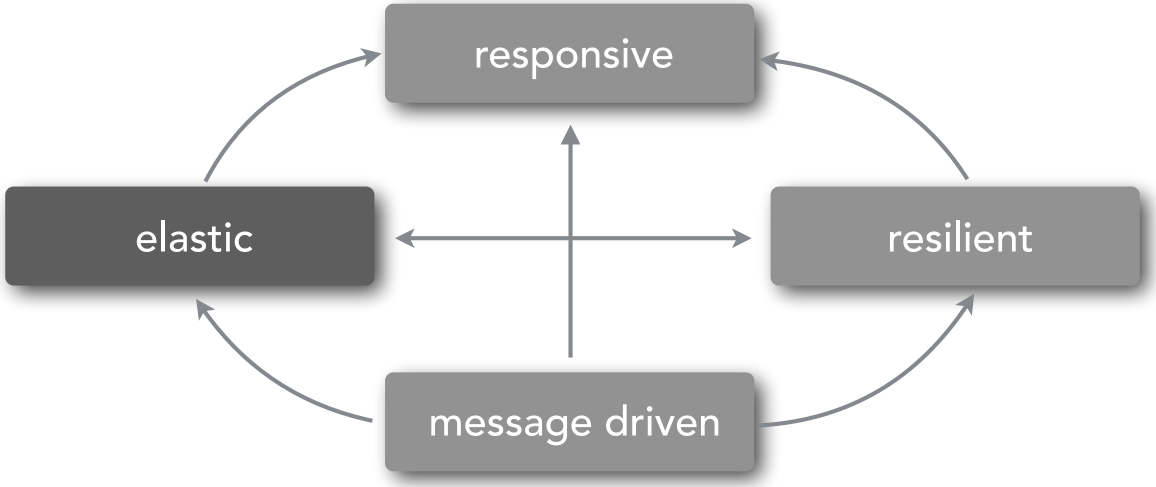


the system stays responsive after a failure through replication, containment, isolation, and delegation

how the system reacts to failures

reactive architecture

reactive manifesto

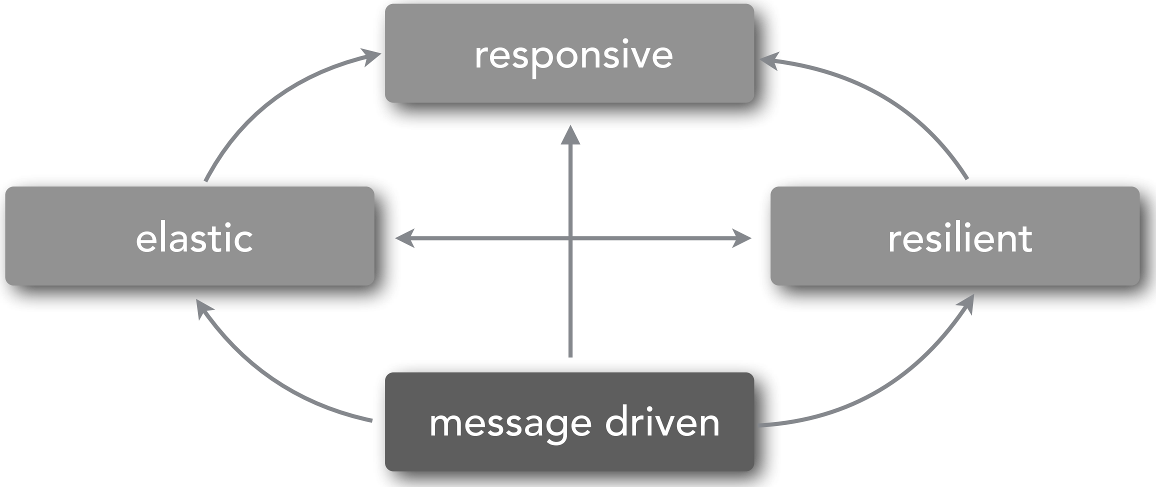


the system stays responsive under varying workload

how the system reacts to load

reactive architecture

reactive manifesto



the system relies on asynchronous messaging to ensure loose coupling, isolation, location transparency, and error delegation

how the system reacts to events

reactive architecture

reactive architecture

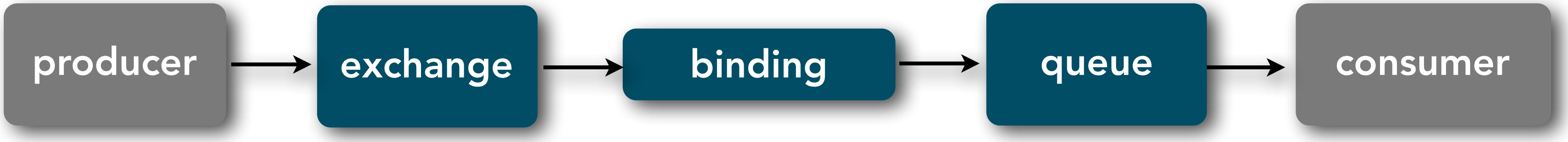
vs.

reactive programming

advanced message queuing protocol

open specification that defines an industry standard wire-level messaging protocol used to send and receive messages across all platforms.

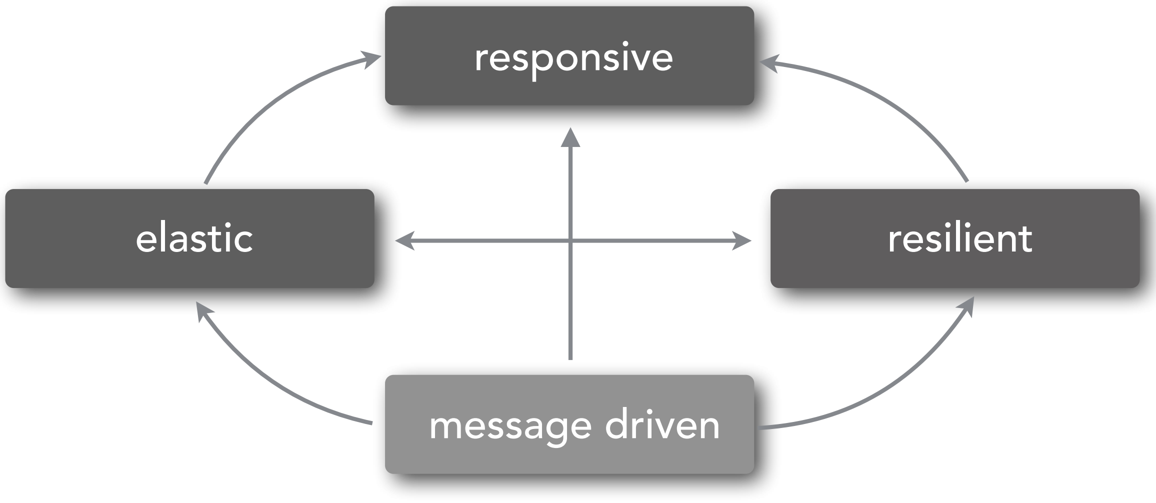
RabbitMQ™



Channel Monitor Pattern

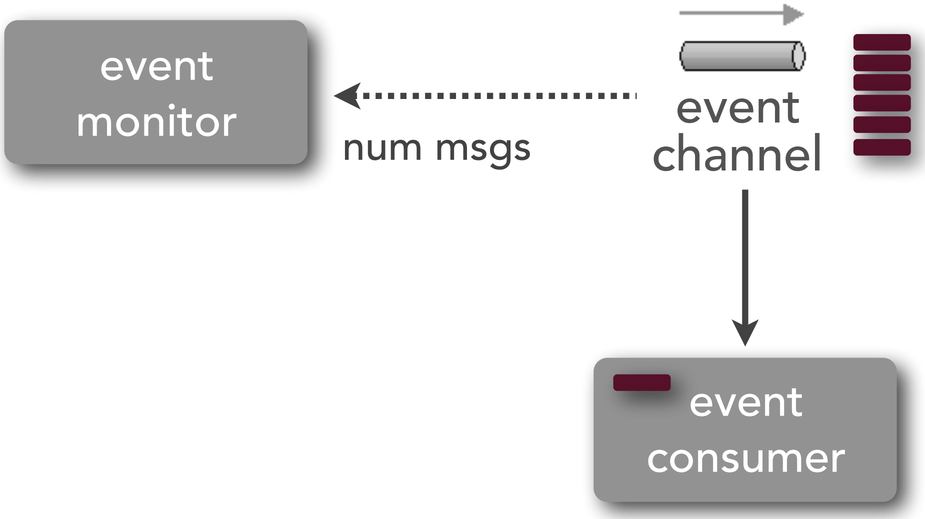
channel monitor pattern

how can you determine the current load on an event channel without consuming events?

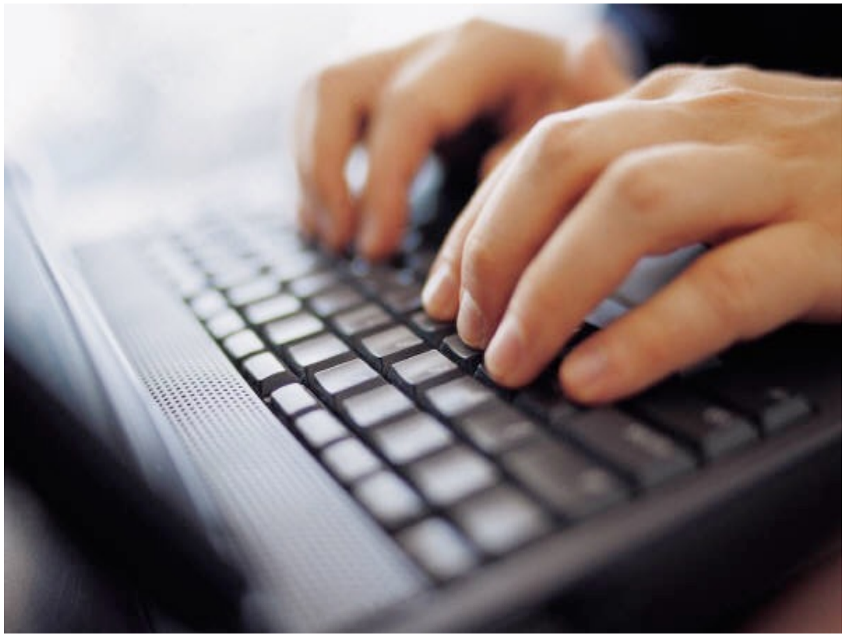


channel monitor pattern

how can you determine the current load on an event channel without consuming events?



channel monitor pattern



let’s see the basic setup and issue…

channel monitor pattern

Monitor.java

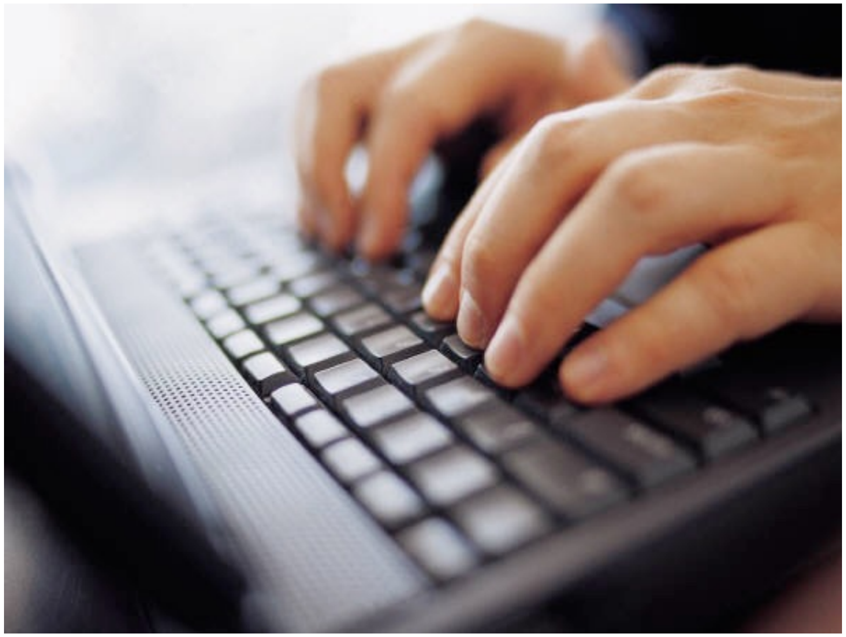
```
Channel channel = AMQPCommon.connect();
long consumers = channel.consumerCount("trade.eq.q");
long queueDepth = channel.messageCount("trade.eq.q");


DeclareOk queue = channel.queueDeclare("trade.eq.q",...);
long consumers = queue.getConsumerCount();
long queueDepth = queue.getMessageCount();
```

channel monitor pattern

Consumer.java

```
Channel channel = AMQPCommon.connect();
channel.basicQos(1);
channel.basicConsume("trade.eq.q", false, consumer);
QueueingConsumer.Delivery msg = consumer.nextDelivery();
channel.basicAck(msg.getEnvelope().getDeliveryTag(), false);
```

channel monitor pattern

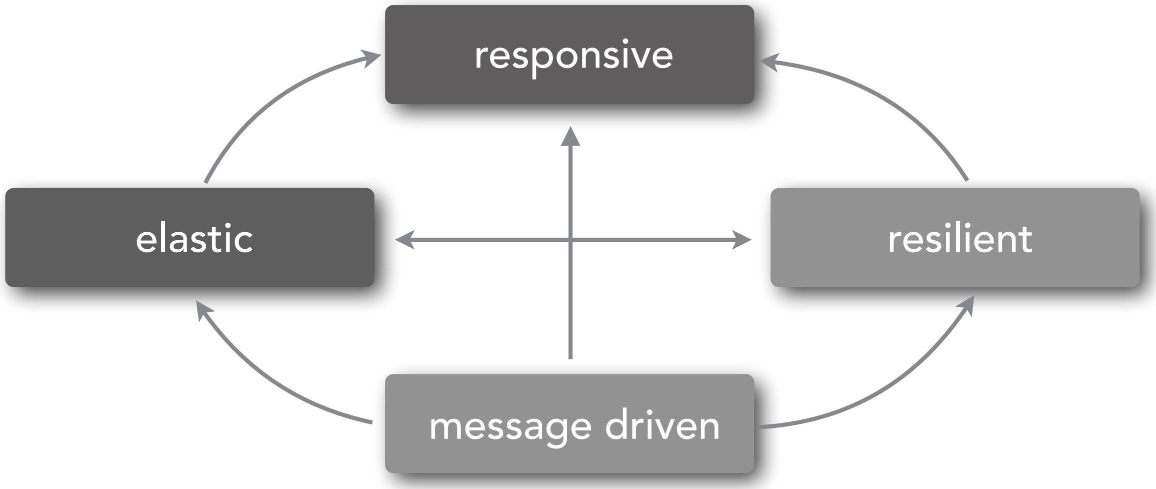


let’s see the result…

Consumer Supervisor Pattern

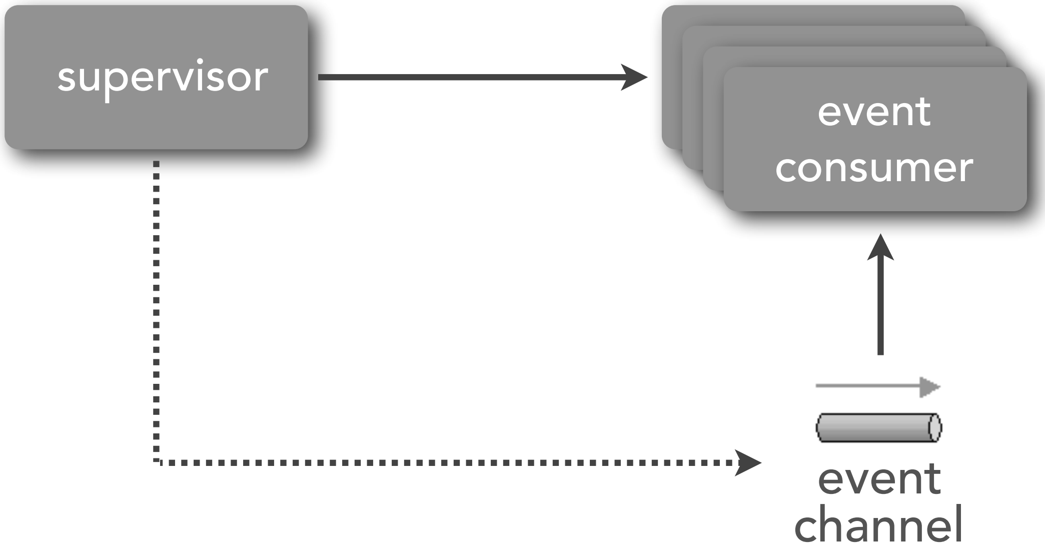
consumer supervisor pattern

how can you react to varying changes in load to event consumers to ensure consistent response time?

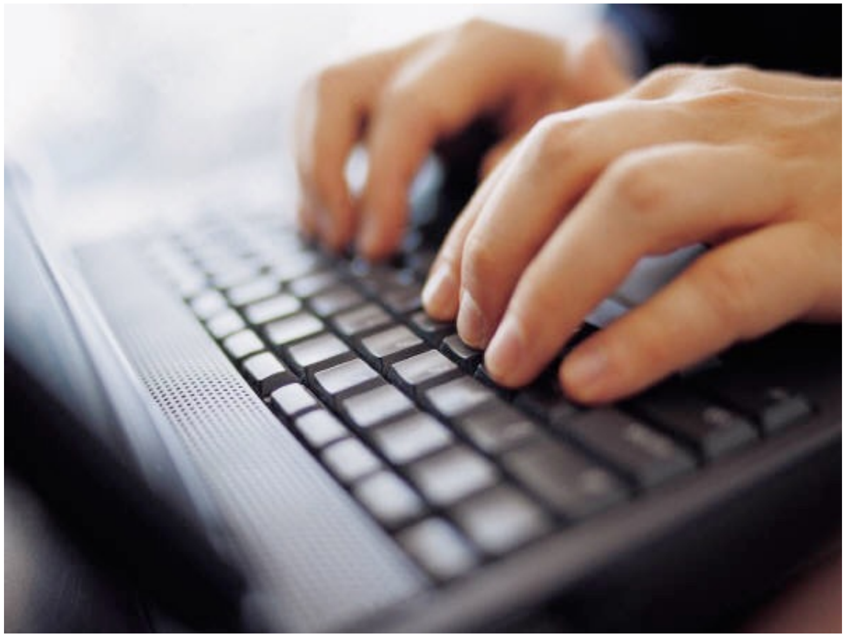


consumer supervisor pattern

how can you react to varying changes in load to event consumers to ensure consistent response time?



consumer supervisor pattern



let’s see the issue….

consumer supervisor pattern

Supervisor.java

```java
private List<AMQPConsumer> consumers =
   new ArrayList<AMQPConsumer>();
Connection connection;

private void startConsumer() {
   AMQPConsumer consumer = new AMQPConsumer();
   consumers.add(consumer);
   new Thread() {
      public void run() {
        consumer.startup(connection);
   }}.start();
}
```

consumer supervisor pattern

Supervisor.java

```
private void stopConsumer() {
  if (consumers.size() > 1) {
    AMQPConsumer consumer = consumers.get(0);
    consumer.shutdown();
    consumers.remove(consumer);
   }
 }
```

consumer supervisor pattern

Supervisor.java

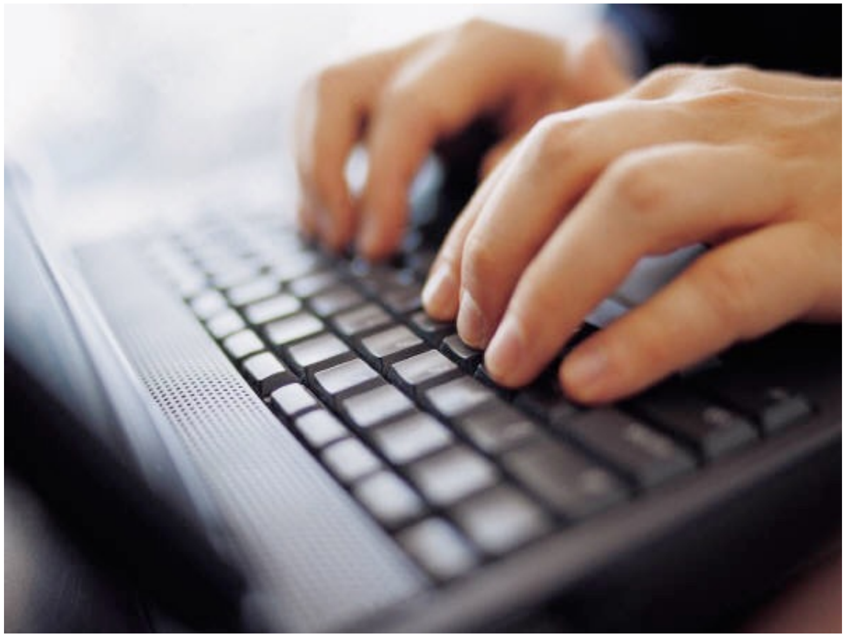
```java
public void execute() throws Exception {
   Channel channel = AMQPCommon.connect();
   connection = channel.getConnection();
   startConsumer();
   while (true) {
      long queueDepth = channel.messageCount("trade.eq.q");
      long consumersNeeded = queueDepth/2;
      long diff = Math.abs(consumersNeeded - consumers.size());
      for (int i=0;i<diff;i++) {
         if (consumersNeeded > consumers.size())
            startConsumer();
         else
            stopConsumer();
      }
      Thread.sleep(1000);
   }
 }
```

consumer supervisor pattern

Consumer.java

```
private Boolean active = true;

public void startup(Connection connection) {
  Channel channel = connection.createChannel();
   QueueingConsumer consumer = new QueueingConsumer(channel);
   ...
   while (active) {
      QueueingConsumer.Delivery msg = consumer.nextDelivery();
      ...
   }
   channel.close();
}


public void shutdown() {
   synchronized(active) { active = false; }
}
```

consumer supervisor pattern

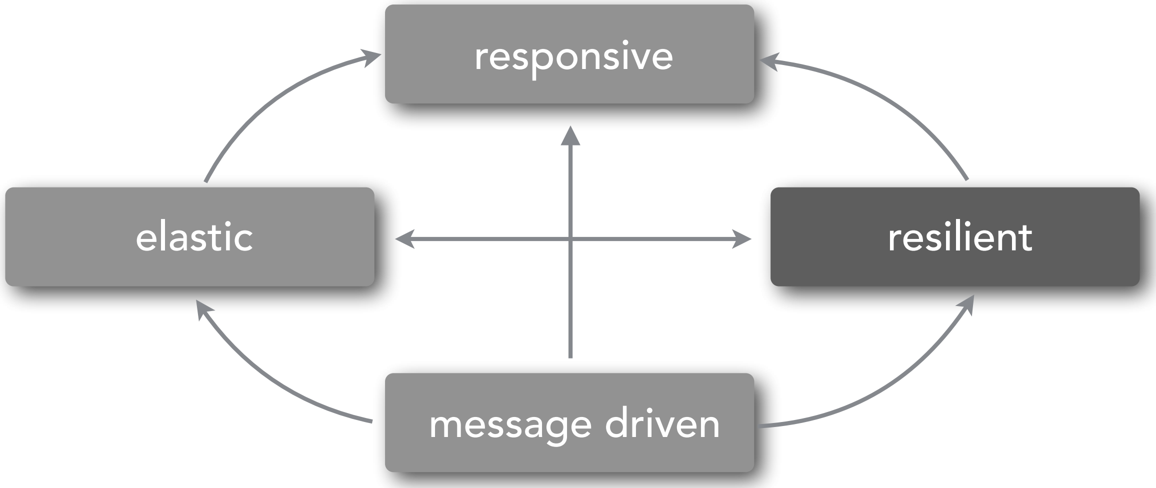


let’s see the result…

Producer Control Flow Pattern

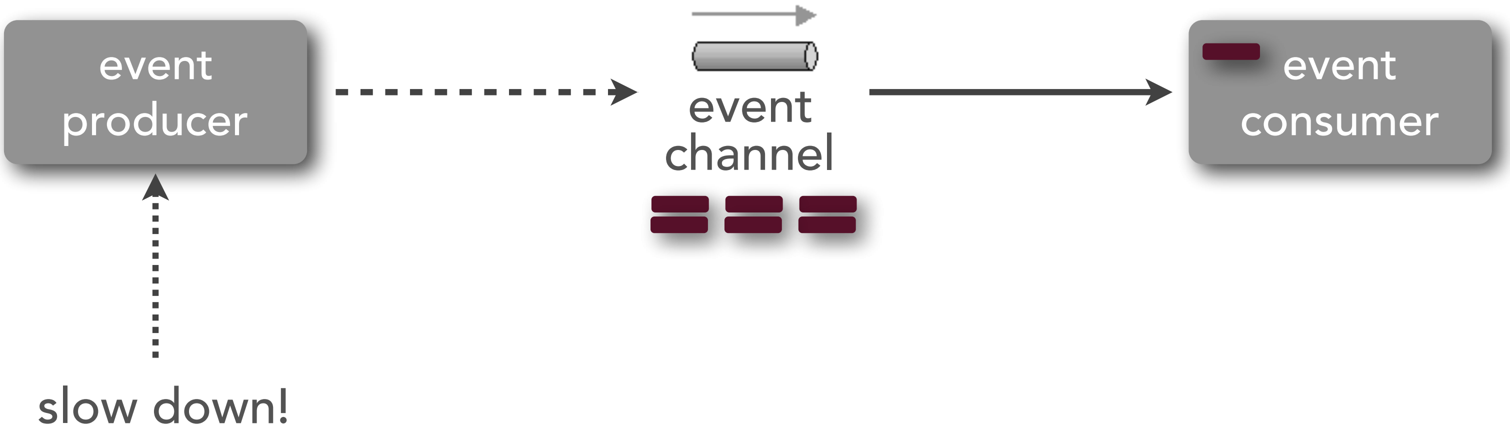
producer control flow pattern

how can you slow down message producers when the messaging system becomes overwhelmed?



producer control flow pattern

how can you slow down message producers when the messaging system becomes overwhelmed?



producer control flow pattern

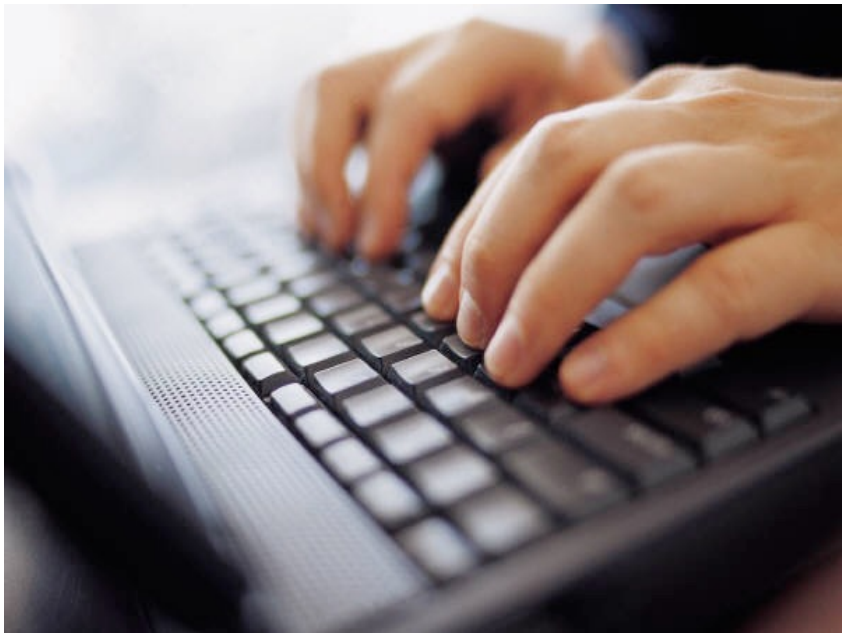
how can you slow down message producers when the messaging system becomes overwhelmed?

RabbitMQ™

ActiveMQ

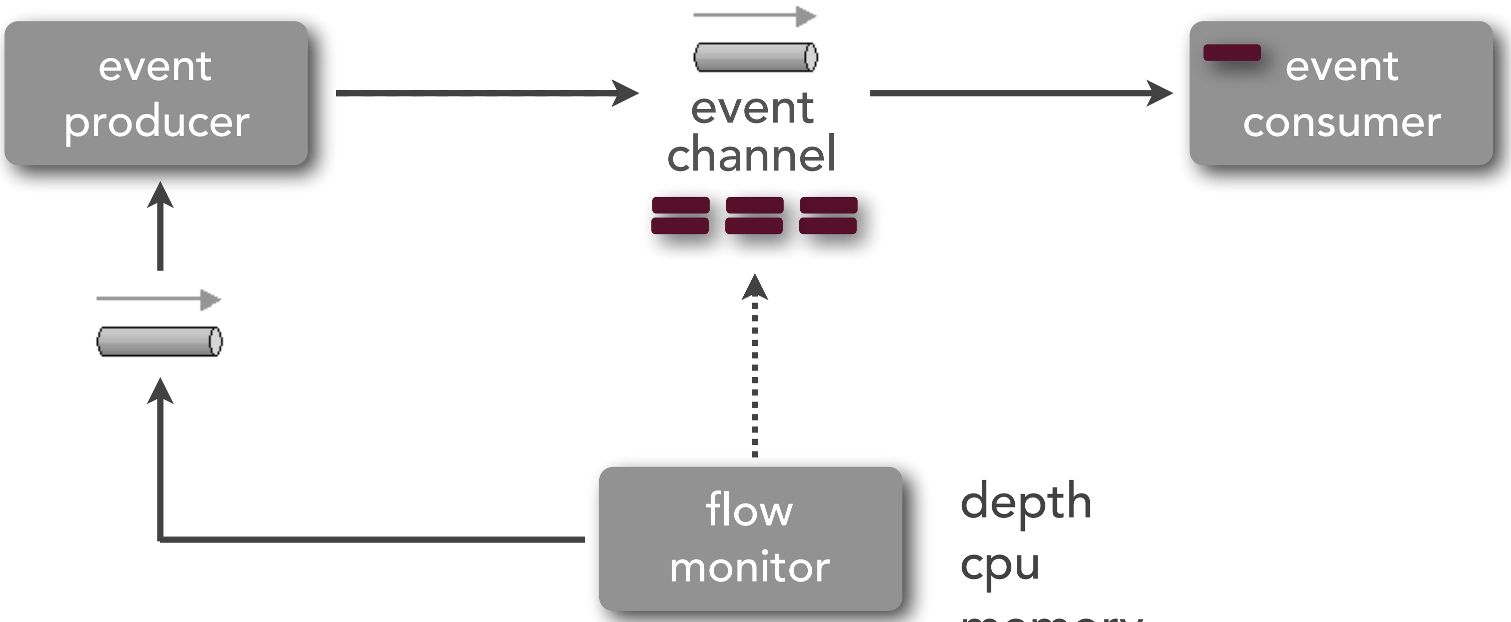
shutdown (broker) vs. slowdown (pattern)

producer control flow pattern



let’s see the issue….

producer control flow pattern



depth
cpu
memory
threads
throughput

producer control flow pattern

FlowMonitor.java

```java
public void execute() throws Exception {
   Channel channel = AMQPCommon.connect();
   long threshold = 10;
   boolean controlFlow = false;
   while (true) {
      long queueDepth = channel.messageCount("trade.eq.q");
      if (queueDepth > threshold && !controlFlow) {
         controlFlow = enableControlFlow(channel);
      } else if (queueDepth <= (threshold/2) && controlFlow) {
         controlFlow = disableControlFlow(channel);
      }
      Thread.sleep(3000);
   }
}
```

producer control flow pattern

FlowMonitor.java

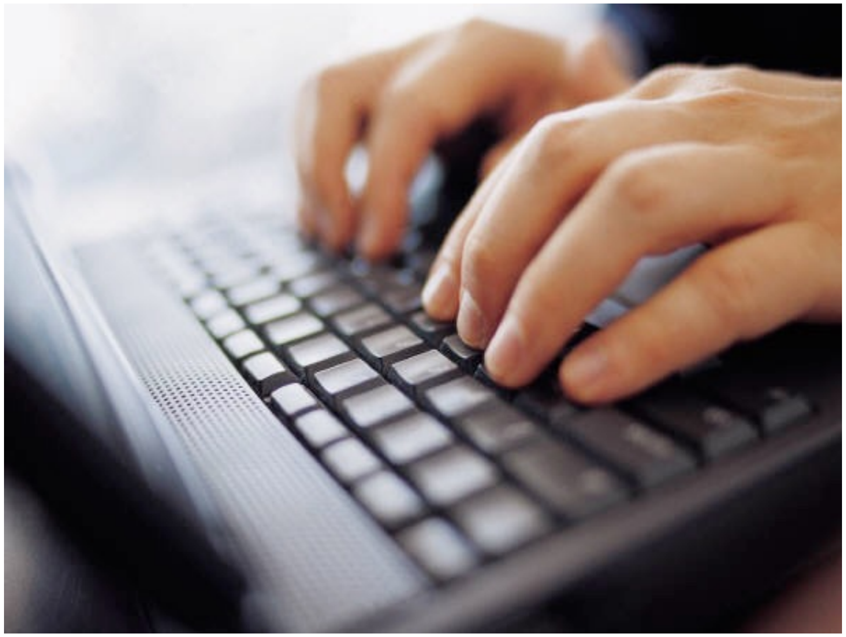
```
private boolean enableControlFlow(Channel channel) {
   byte[] msg = String.valueOf(true).getBytes();
   channel.basicPublish("flow.fx", "", null, msg);
   return true;
}


private boolean disableControlFlow(Channel channel) {
   byte[] msg = String.valueOf(false).getBytes();
   channel.basicPublish("flow.fx", "", null, msg);
   return false;
}
```

producer control flow pattern

Producer.java

```
public void startListener() {
   new Thread() {
     public void run() {
      //basic rabbitmq consumer setup...
      while (true) {
        QueueingConsumer.Delivery msg = consumer.nextDelivery();
        boolean controlFlow =
          new Boolean(new String(msg.getBody())).booleanValue();
        synchronized(delay) { delay = controlFlow ? 3000 : 0; }
      }
   }}.start();
}

private void produceMessages() {
   //send trade to queue...
   Thread.sleep(delay);
}
```

producer control flow pattern

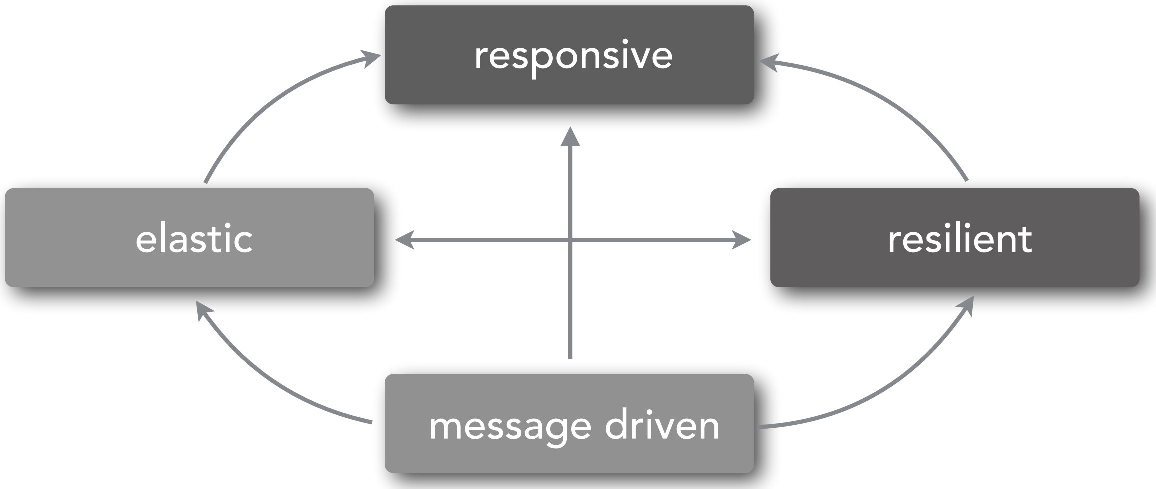


let’s see the result…

Thread Delegate Pattern

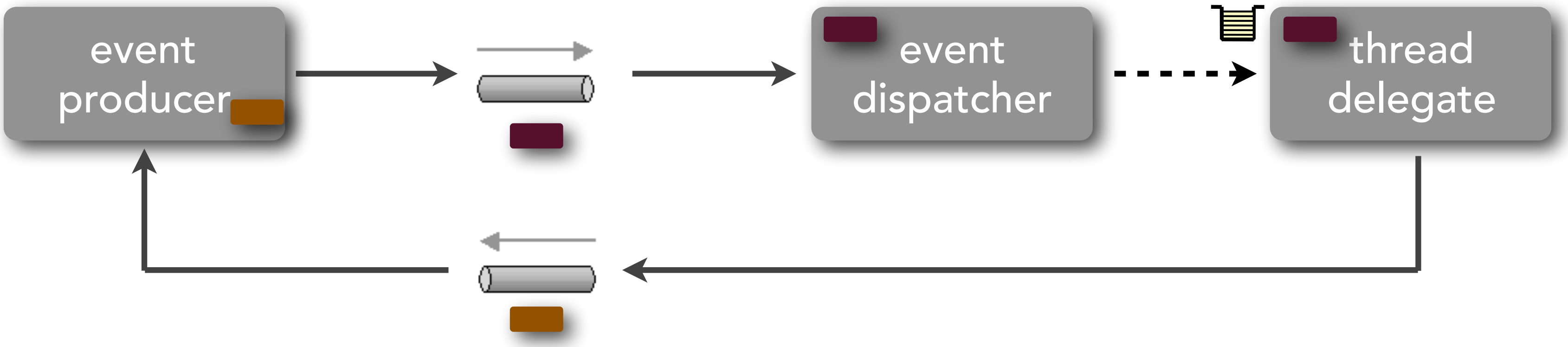
thread delegate pattern

how can you consume messages faster than they are being produced?

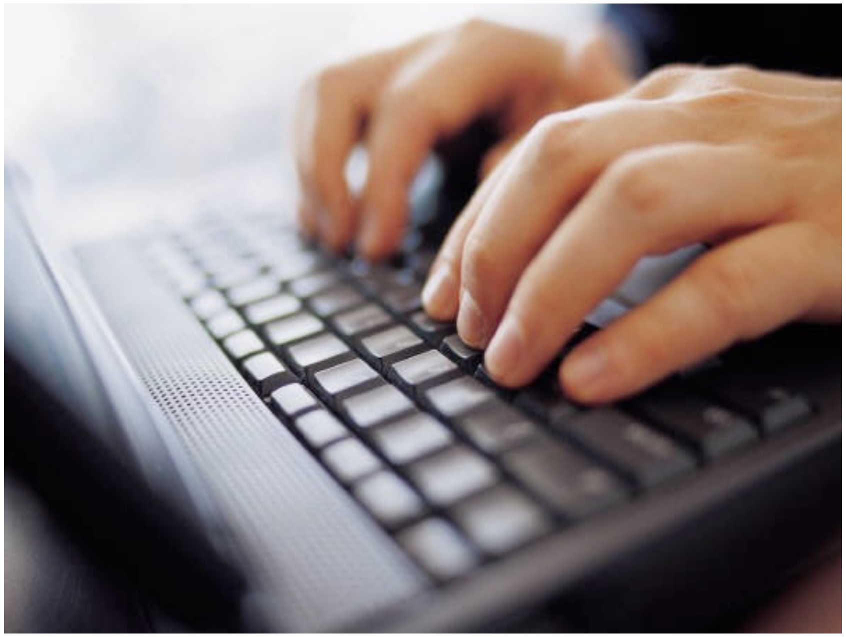


thread delegate pattern

how can you consume messages faster than they are being produced?



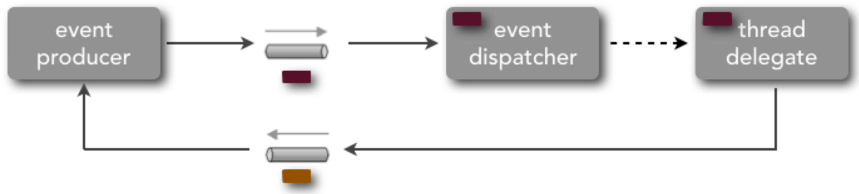
thread delegate pattern

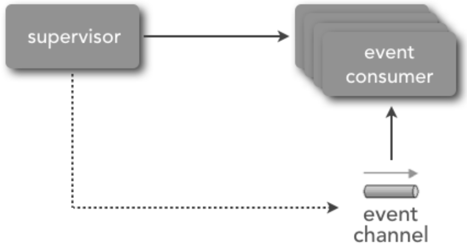


let’s see the issue…

thread delegate pattern

thread delegate vs. consumer supervisor

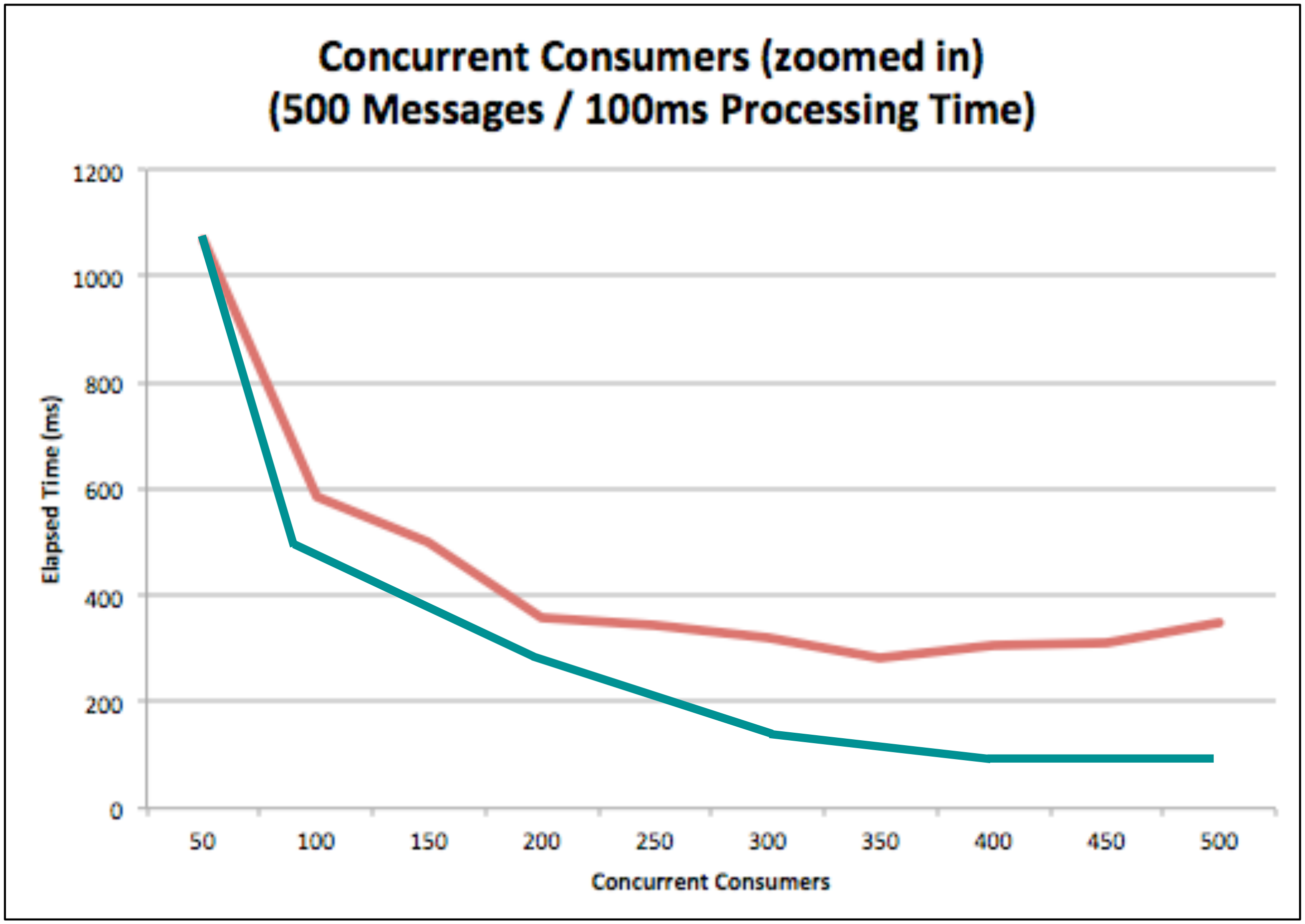




scalability	elasticity
consistent consumers	variable consumers
decoupled event processors	coupled event processors
near-linear performance	diminishing performance

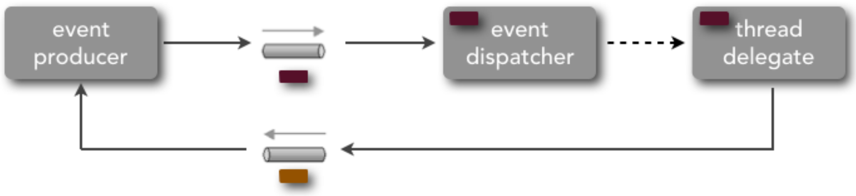
thread delegate pattern

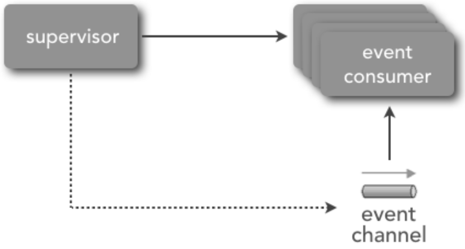
thread delegate vs. consumer supervisor



thread delegate pattern

thread delegate vs. consumer supervisor



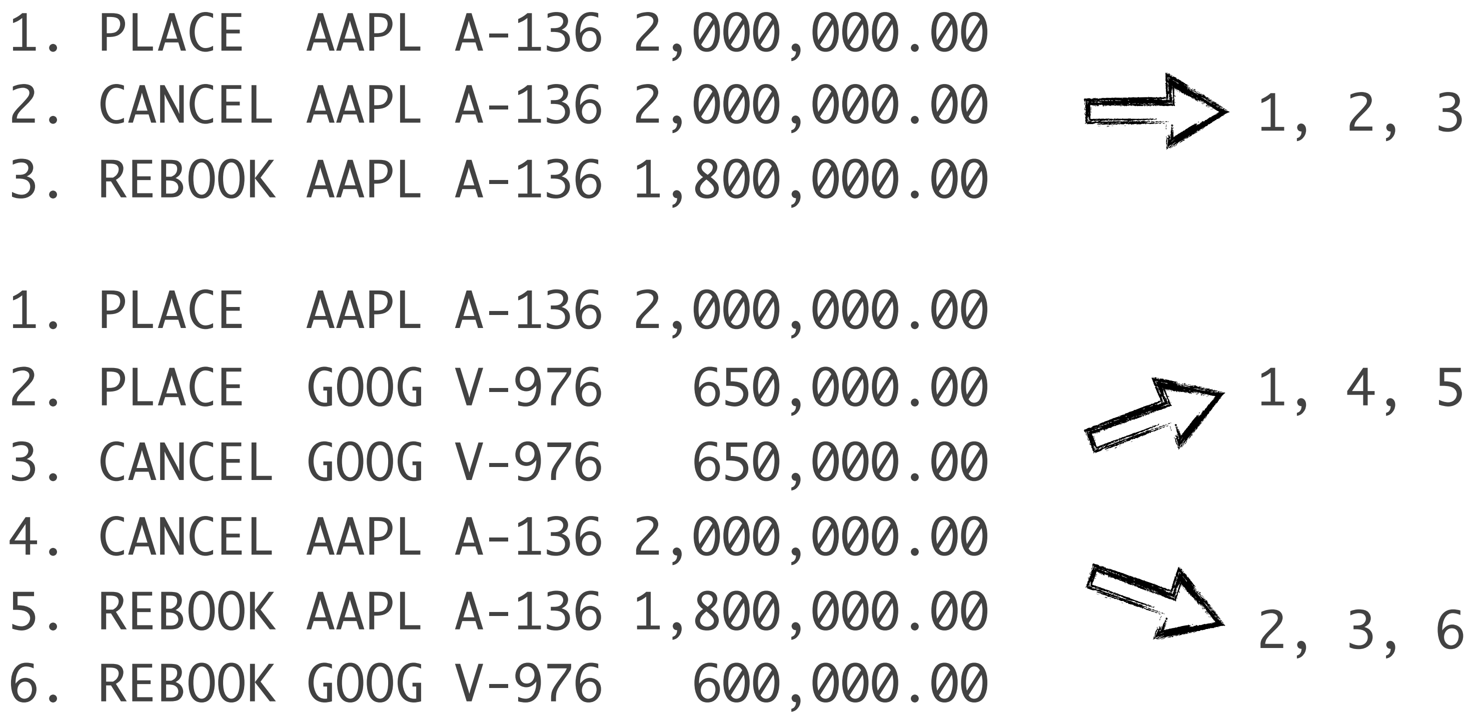


scalability	elasticity
consistent consumers	variable consumers
decoupled event processors	coupled event processors
near-linear performance	diminishing performance
can preserve message order	message order not preserved

thread delegate pattern

preserving message order

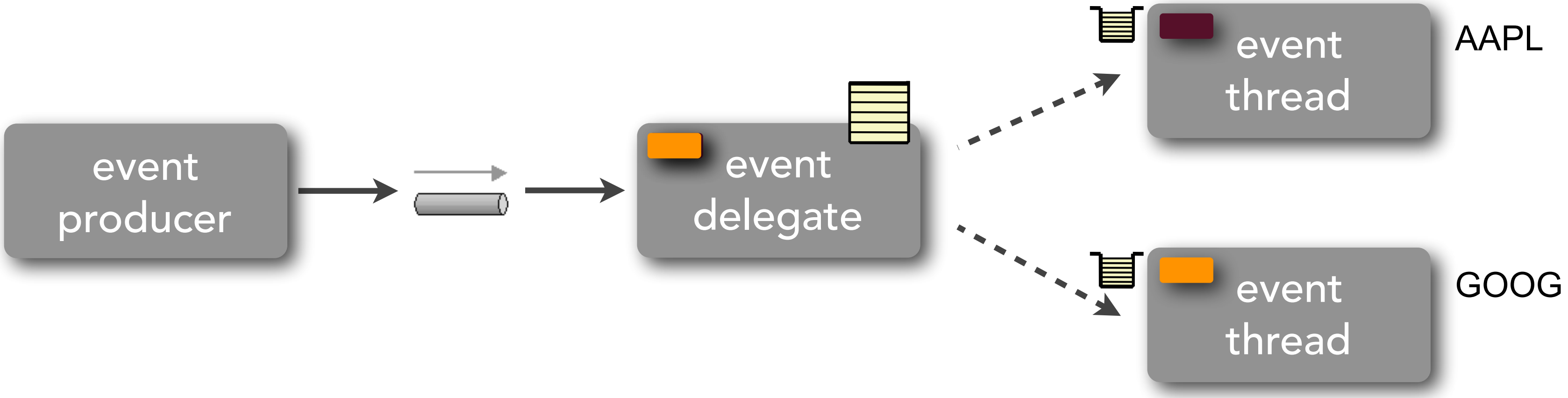
premise: not every message must be ordered, but rather messages *within a context* must be ordered



```
1. PLACE  AAPL A-136 2,000,000.00
2. CANCEL AAPL A-136 2,000,000.00      →  1, 2, 3
3. REBOOK AAPL A-136 1,800,000.00

1. PLACE  AAPL A-136 2,000,000.00
2. PLACE  GOOG V-976   650,000.00      ↗  1, 4, 5
3. CANCEL GOOG V-976   650,000.00
4. CANCEL AAPL A-136 2,000,000.00
5. REBOOK AAPL A-136 1,800,000.00      ↘  2, 3, 6
6. REBOOK GOOG V-976   600,000.00
```

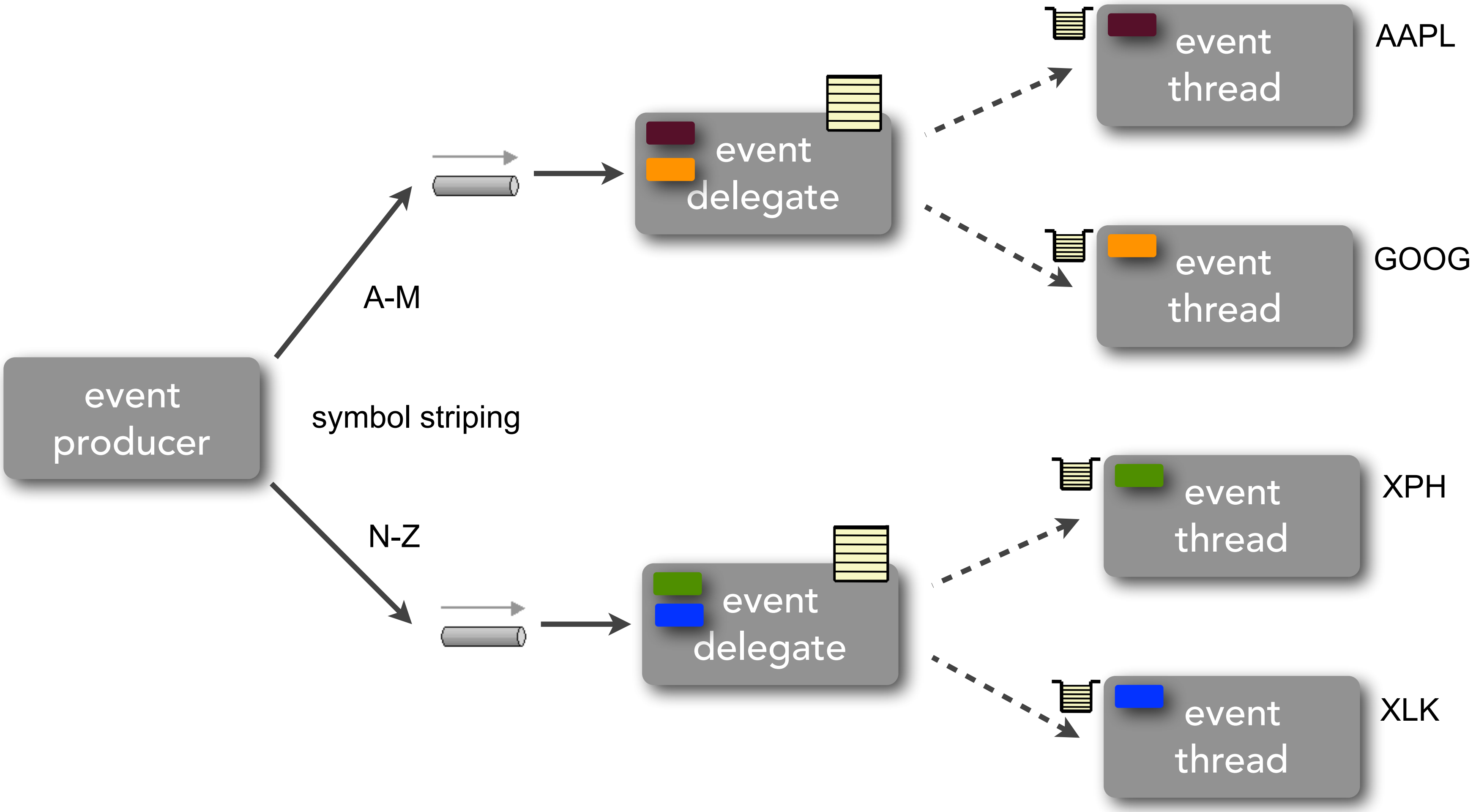
thread delegate pattern

preserving message order



thread delegate pattern

preserving message order

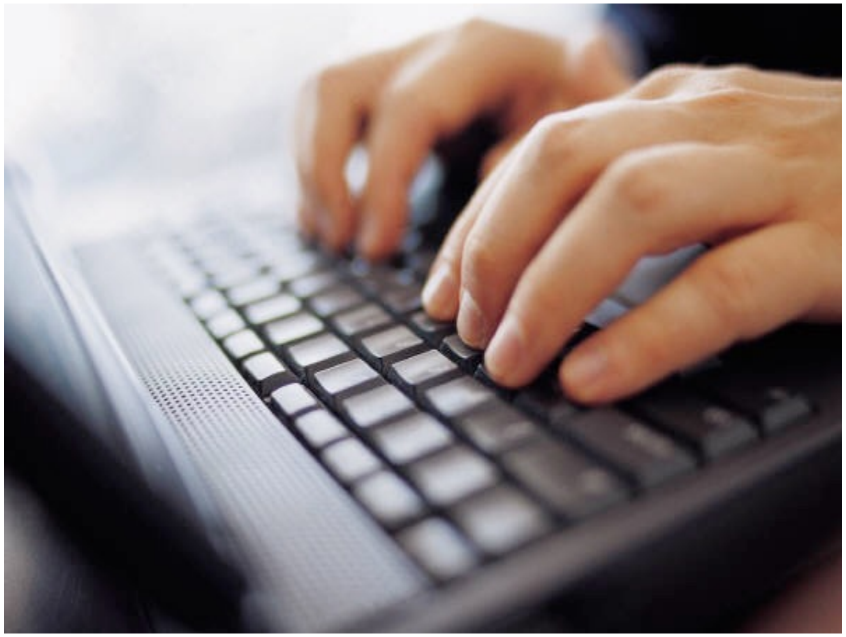


thread delegate pattern

Dispatcher.java

```
Channel channel = AMQPCommon.connect();
QueueingConsumer consumer = new QueueingConsumer(channel);
channel.basicConsume("trade.eq.q", true, consumer);

while (true) {
   QueueingConsumer.Delivery msg = consumer.nextDelivery();
   new Thread(new POJOThreadProcessor(
      new String(msg.getBody()))).start();
}
```

thread delegate pattern

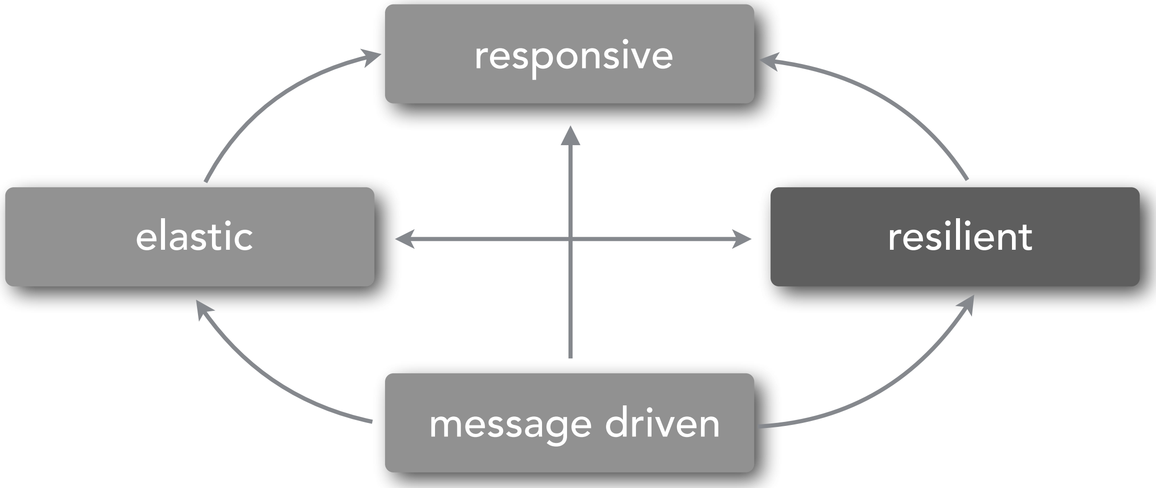


let’s see the result…

Workflow Event Pattern

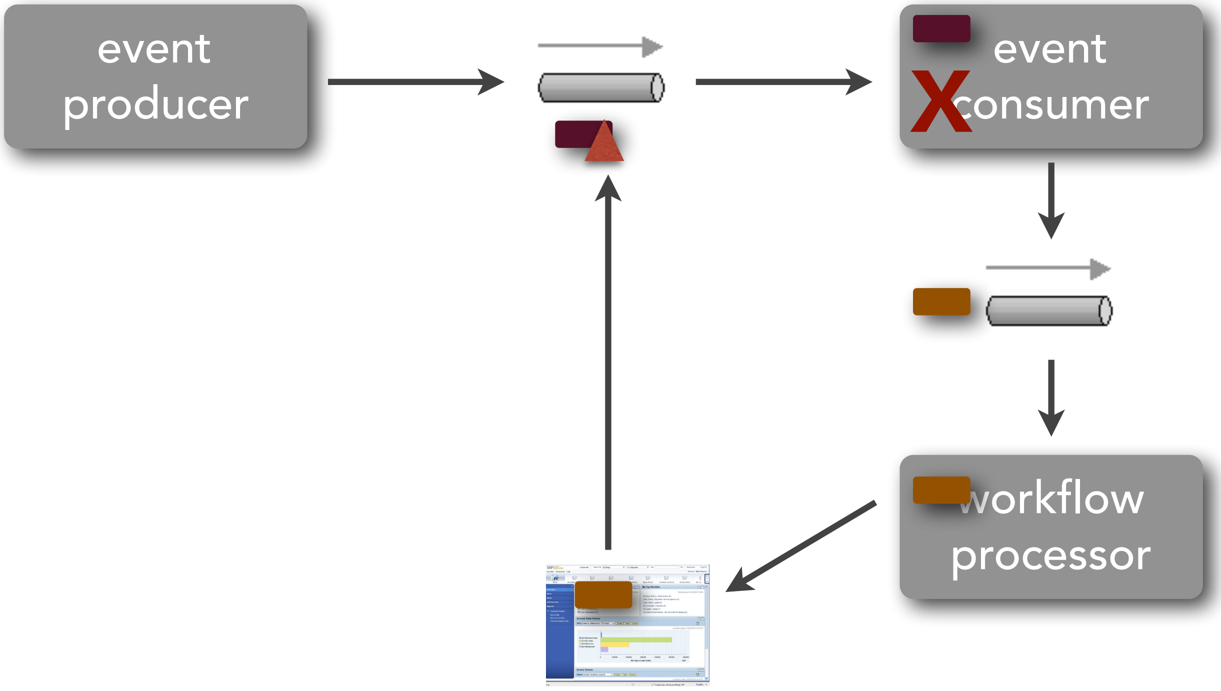
workflow event pattern

how can you handle error conditions without failing the transaction?



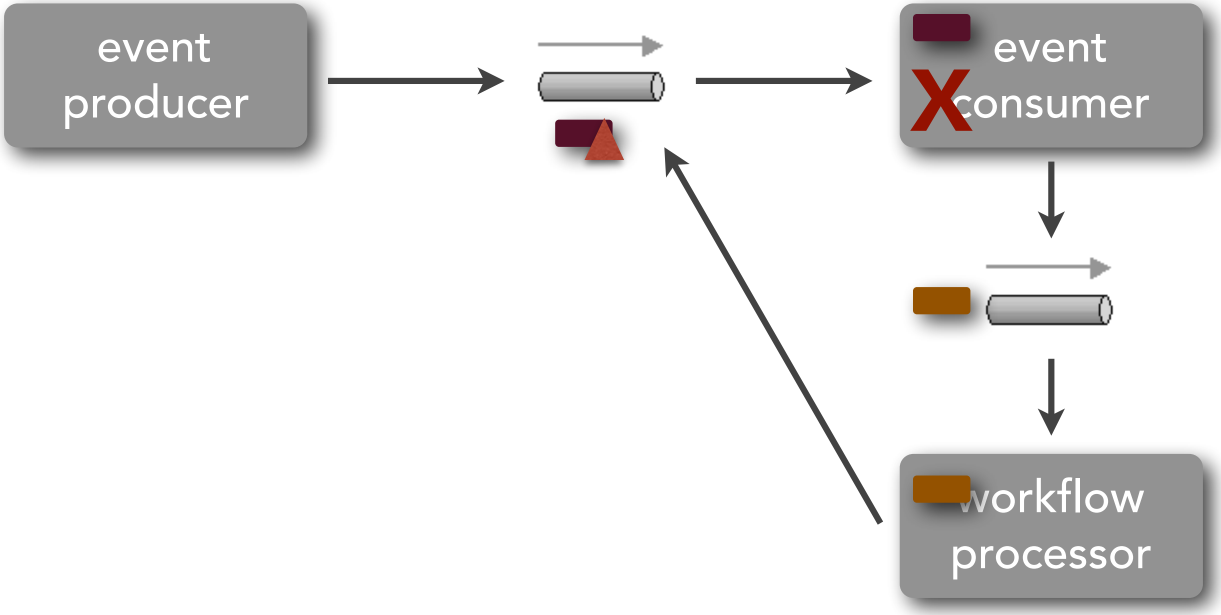
workflow event pattern

how can you handle error conditions without failing the transaction?



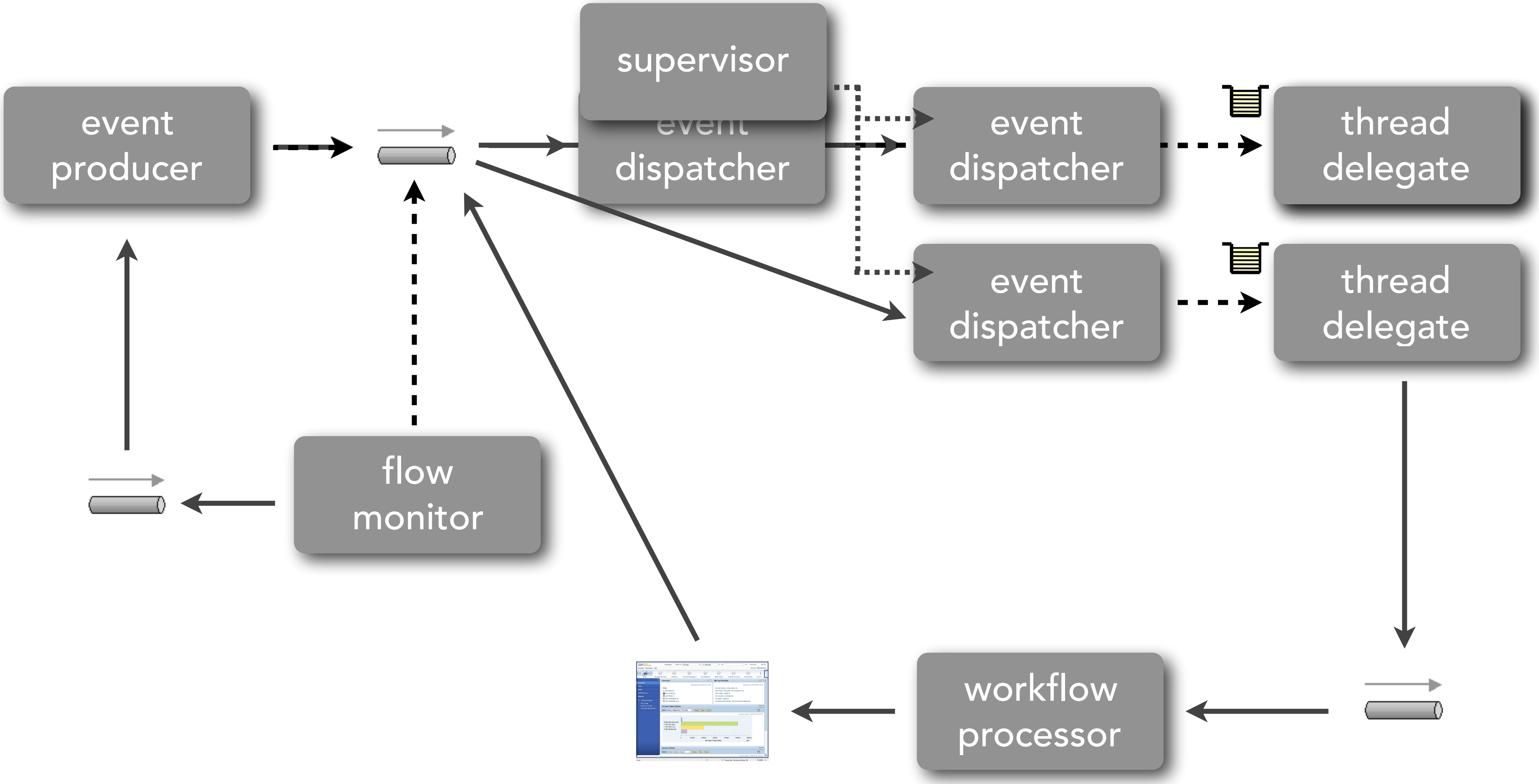
workflow event pattern

how can you handle error conditions without failing the transaction?



Combining Patterns

combining patterns



event producer

supervisor

event dispatcher

event dispatcher

thread delegate

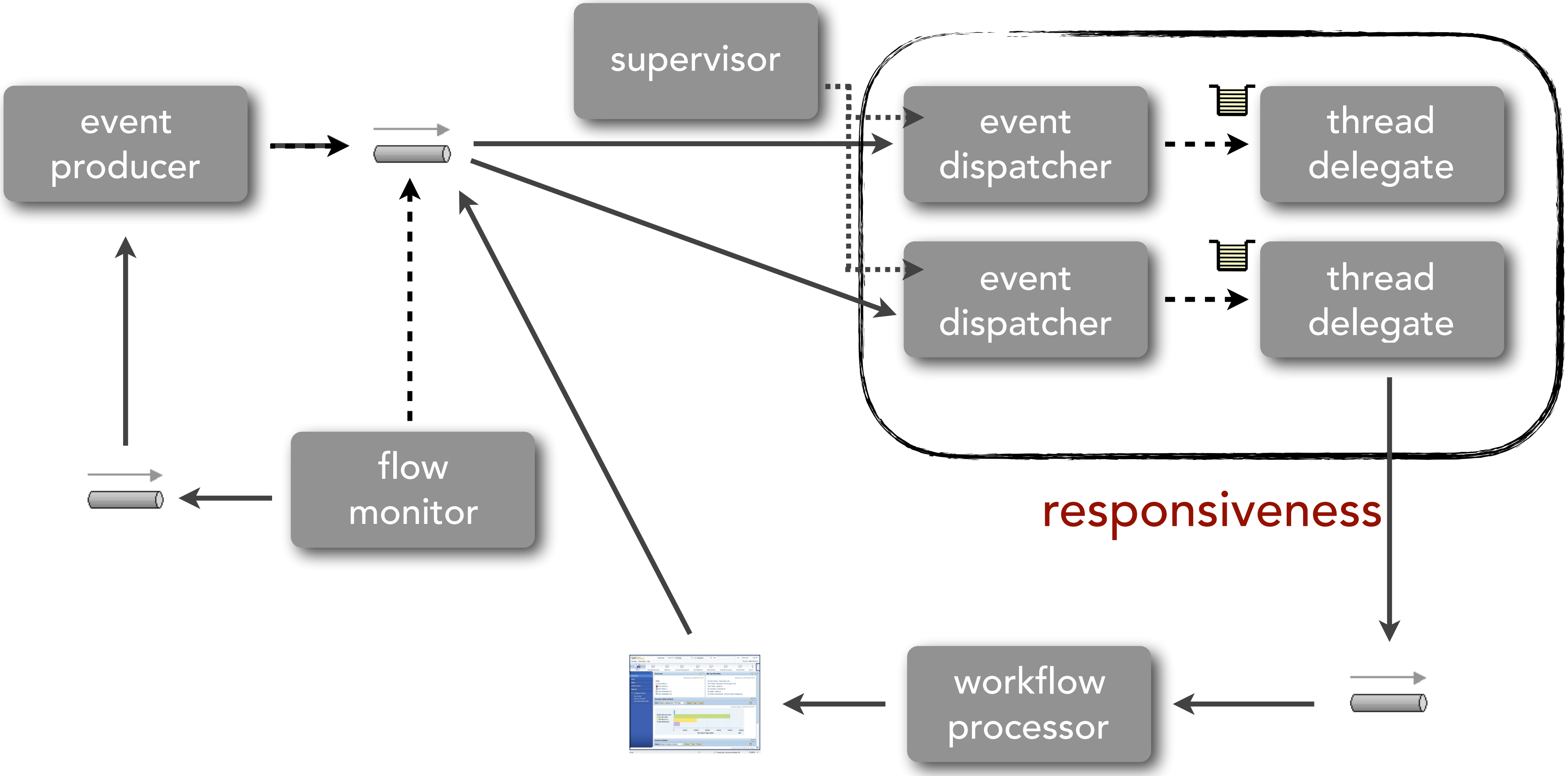
event dispatcher

thread delegate

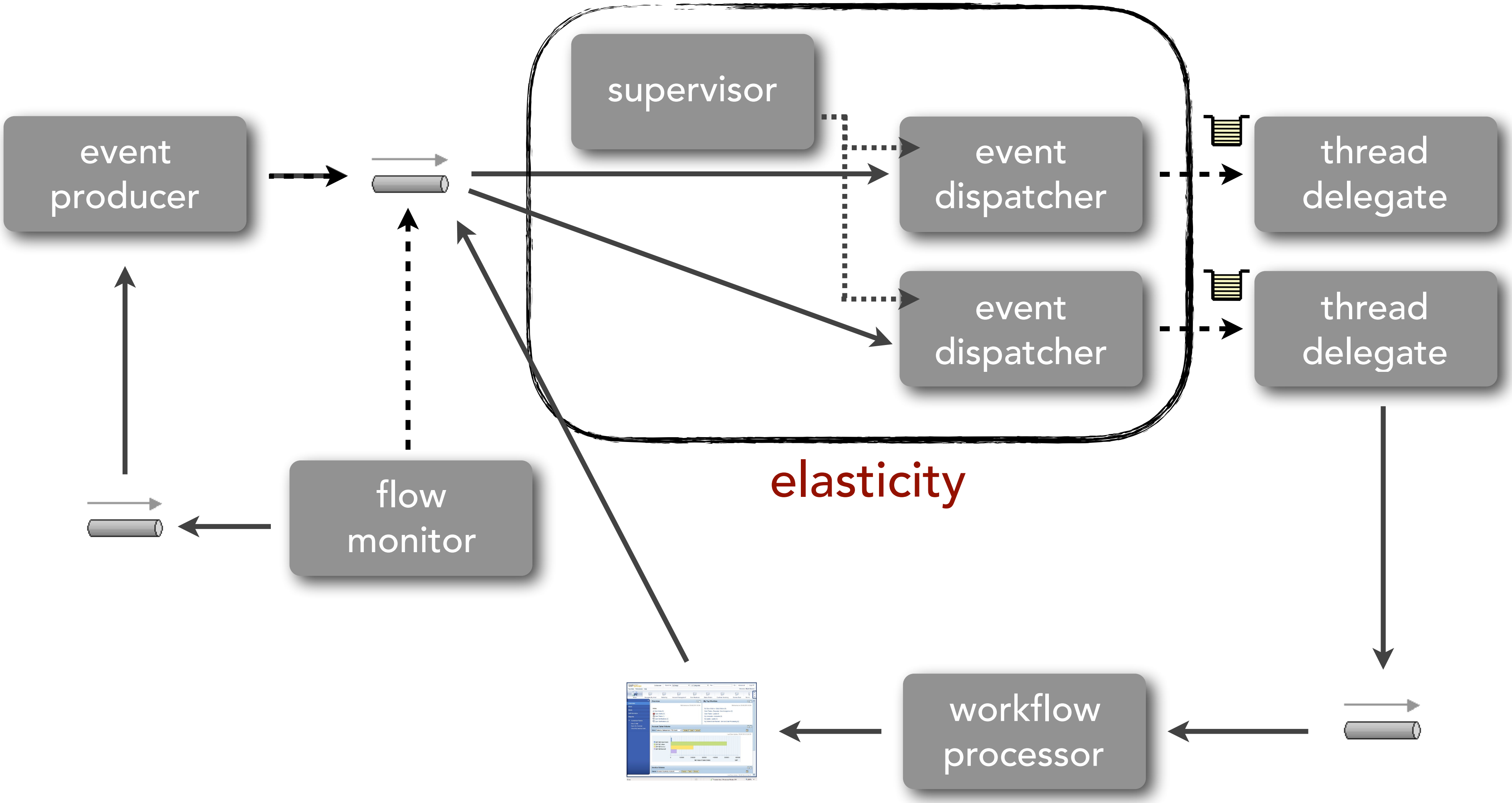
flow monitor

workflow processor

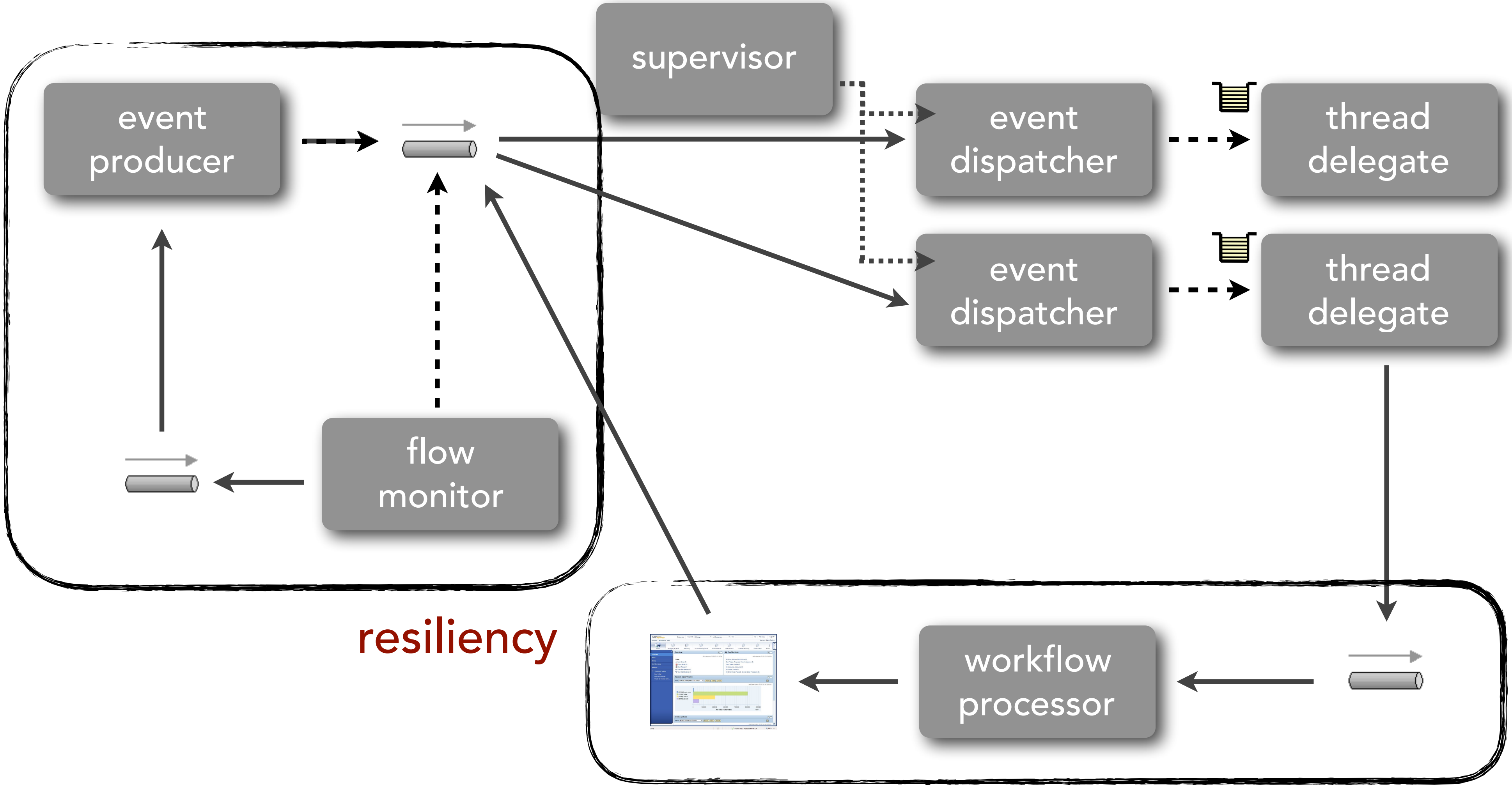
combining patterns



combining patterns



combining patterns



supervisor

event producer

event dispatcher

thread delegate

event dispatcher

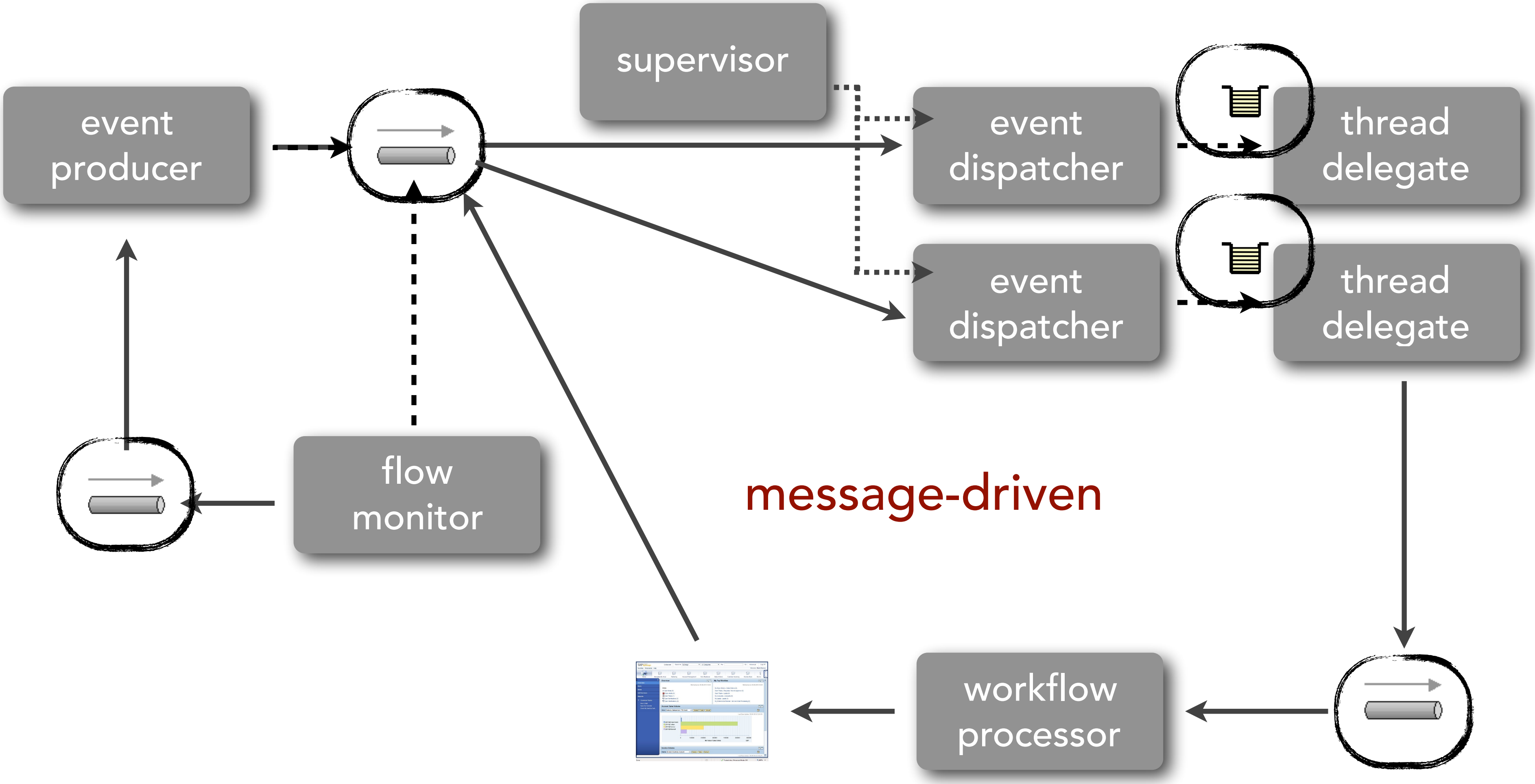
thread delegate

flow monitor

resiliency

workflow processor

combining patterns



event producer

supervisor

event dispatcher

thread delegate

event dispatcher

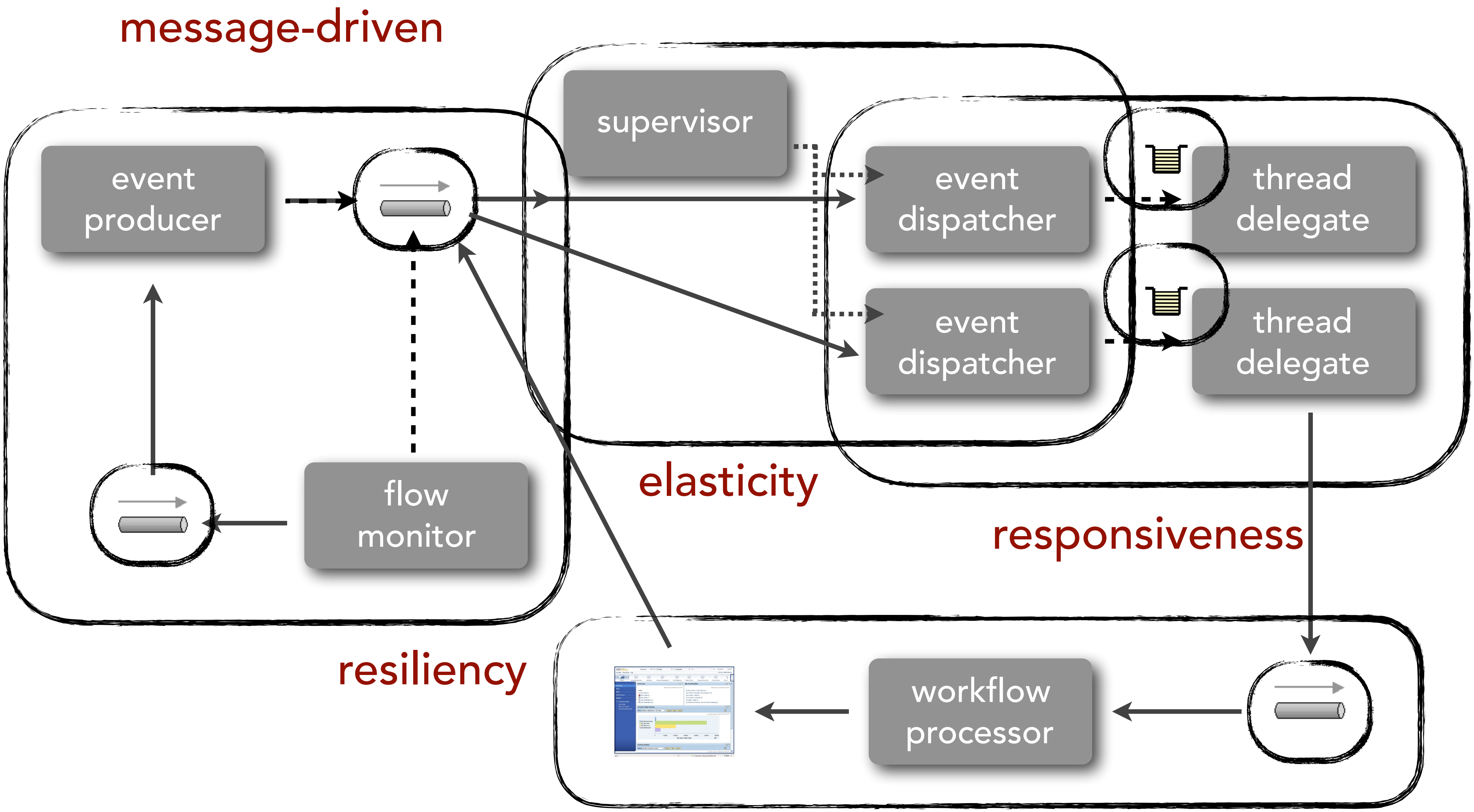
thread delegate

flow monitor

message-driven

workflow processor

combining patterns



message-driven

event producer

flow monitor

supervisor

event dispatcher

event dispatcher

thread delegate

thread delegate

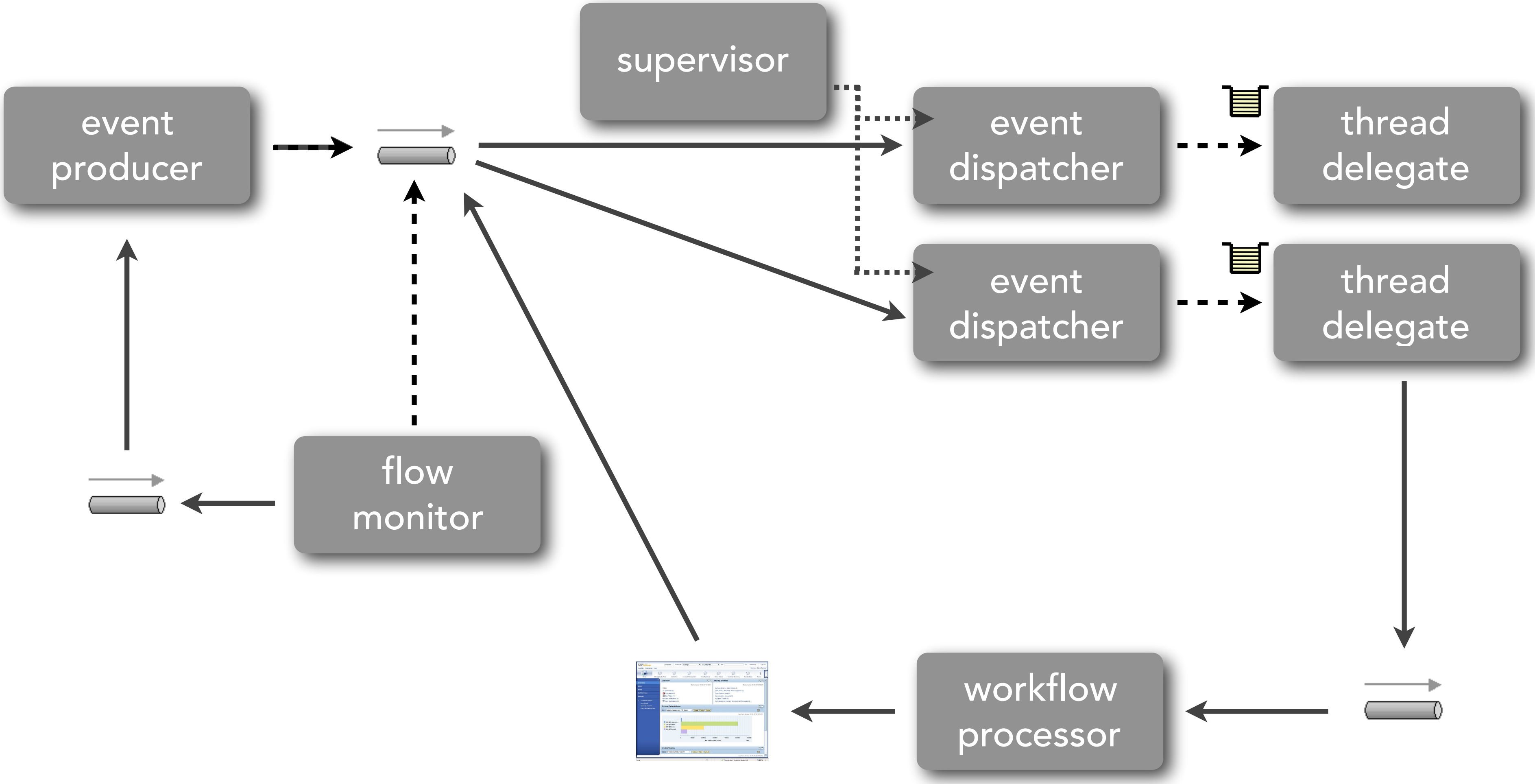
elasticity

responsiveness

resiliency

workflow processor

combining patterns



Reactive Architecture Patterns Using Java and Messaging

Mark Richards

Independent Consultant
Hands-on Software Architect
Published Author / Conference Speaker

http://www.wmrichards.com
https://www.linkedin.com/in/markrichards3
@markrichardssa