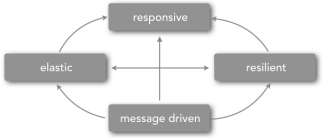
Reactive Architecture Patterns Parts 1 and 2

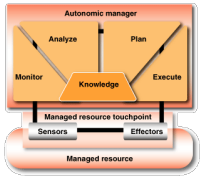
Mark Richards
Independent Consultant
Hands-on Software Architect
Published Author / Conference Speaker
www.wmrichards.com

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

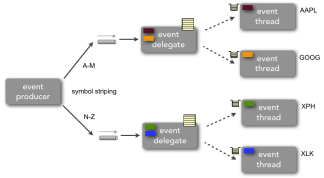
reactive architecture agenda



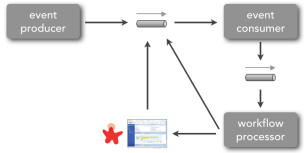
reactive architecture overview



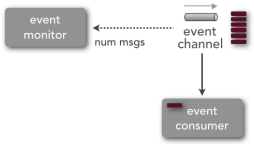
history of autonomic systems



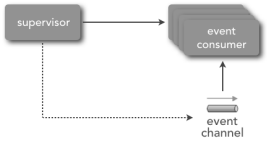
thread delegate pattern



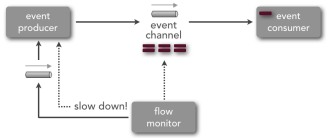
workflow event pattern



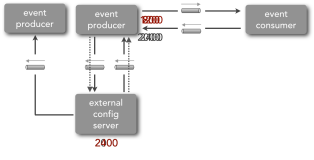
channel monitor pattern



consumer supervisor pattern



producer control flow pattern



threshold adjust pattern

source code

https://github.com/wmr513/reactive

source code

```
3 import com.rabbitmq.client.Channel;
6
7 public class AMQPCommon {
8
9     public static Channel connect() throws Exception {
10        ConnectionFactory factory = new ConnectionFactory();
11        factory.setHost("127.0.0.1");
12        factory.setPort(32768);
13        Connection conn = factory.newConnection();
14        return conn.createChannel();
15    }
16
17    public static void close(Channel channel) throws Exception {
18        channel.close();
19        channel.getConnection().close();
20    }
21
22 }
```

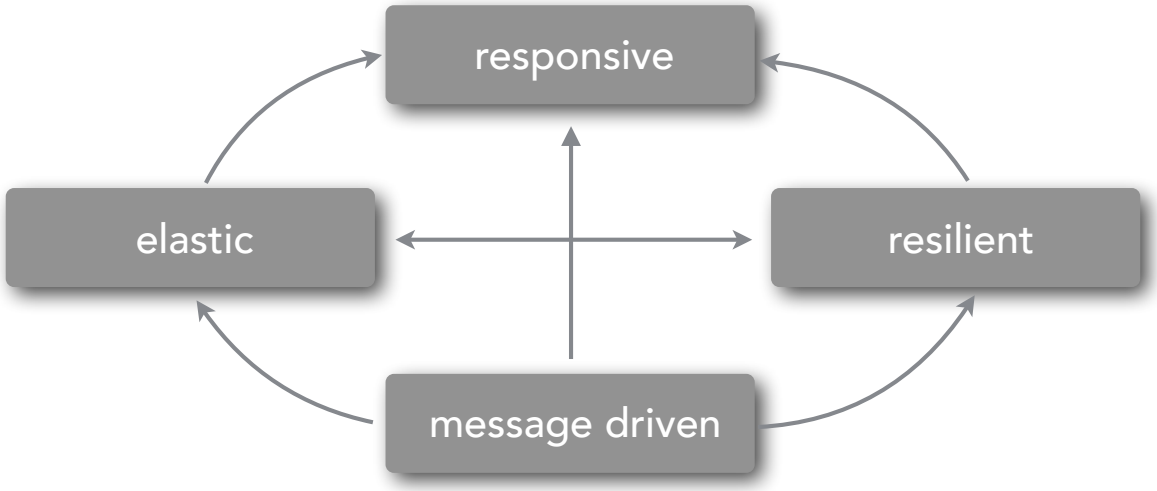
source code

```
public class AMQPInitialize {

    public static void main(String[] args) throws Exception {
        Channel channel = AMQPCommon.connect();

        //create the durable exchanges
        channel.exchangeDeclare("flow.fx", "fanout", true);
        channel.exchangeDeclare("orders.dx", "direct", true);
        System.out.println("exchanges created.");

        //create the durable queues
        channel.queueDeclare("trade.request.q", true, false, false, null);
        channel.queueDeclare("trade.response.q", true, false, false, null);
        channel.queueDeclare("config.q", true, false, false, null);
        channel.queueDeclare("flow.q", true, false, false, null);
        channel.queueDeclare("trade.eq.q", true, false, false, null);
        channel.queueDeclare("trade.1.q", true, false, false, null);
        channel.queueDeclare("trade.2.q", true, false, false, null);
        channel.queueDeclare("workflow.q", true, false, false, null);
```

Reactive Architecture Overview

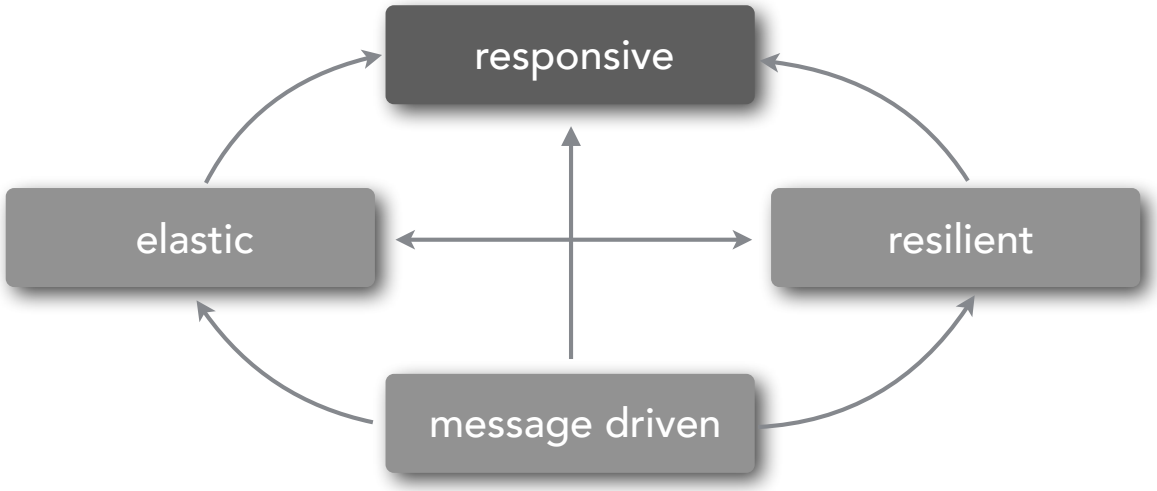
reactive architecture

reactive manifesto



reactive architecture

reactive manifesto

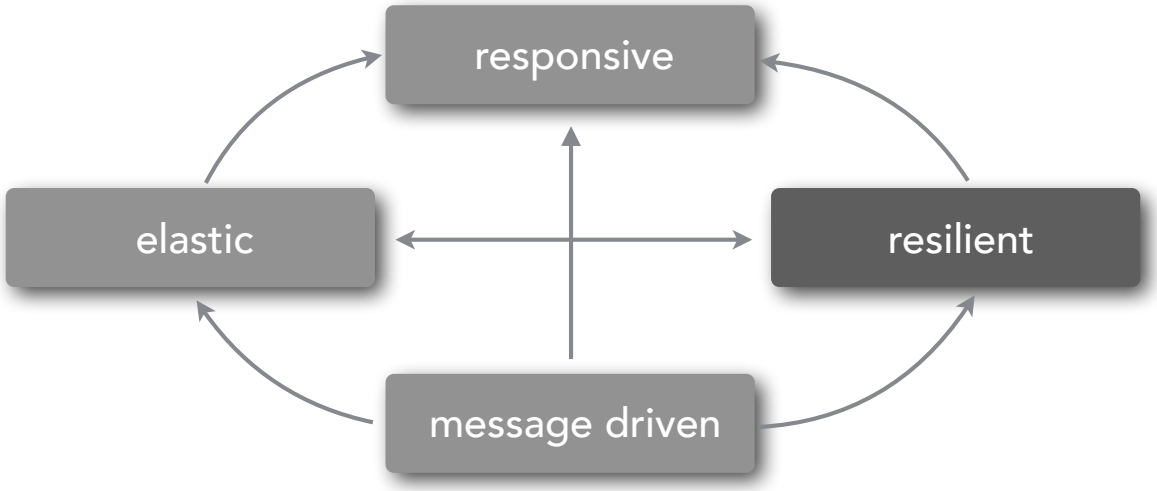


the system responds in a consistent, rapid, and timely manner whenever possible

how the system reacts to users

reactive architecture

reactive manifesto

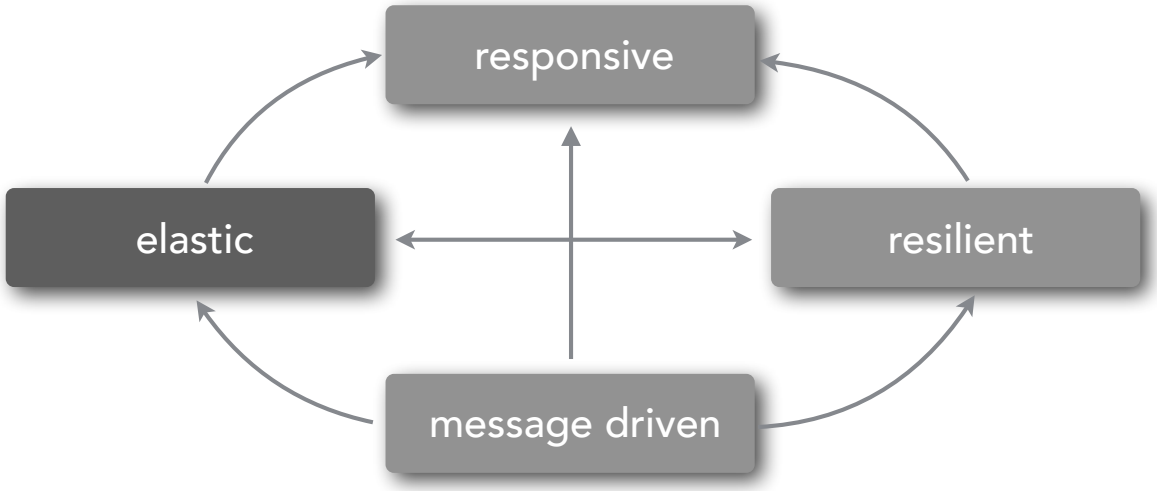


the system stays responsive after a failure through replication, containment, isolation, and delegation

how the system reacts to failures

reactive architecture

reactive manifesto

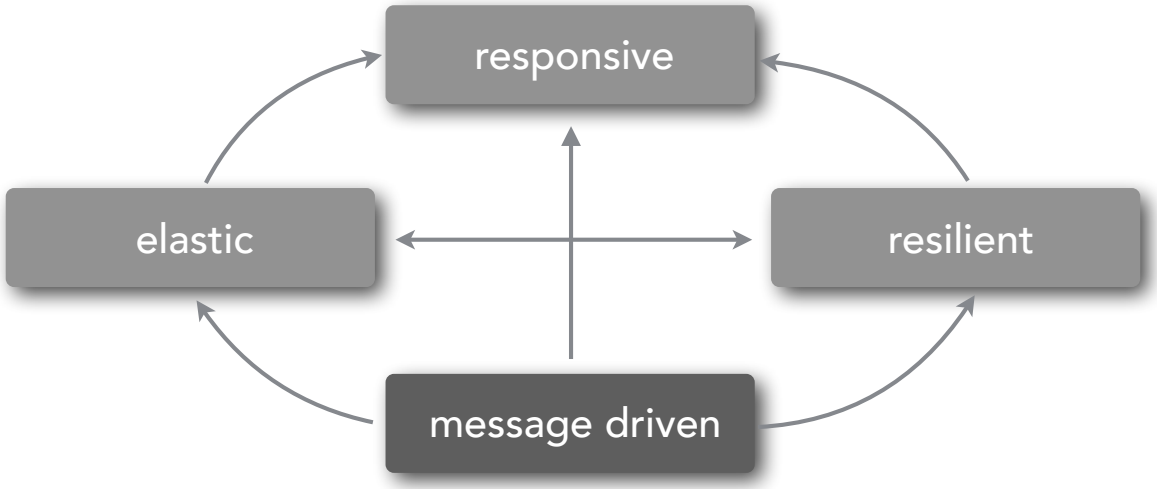


the system stays responsive under varying workload

how the system reacts to load

reactive architecture

reactive manifesto

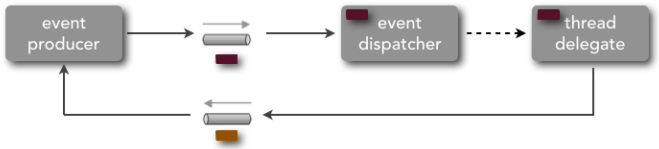


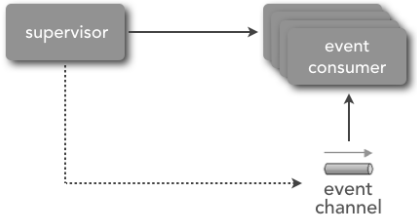
the system relies on asynchronous messaging to ensure loose coupling, isolation, location transparency, and error delegation

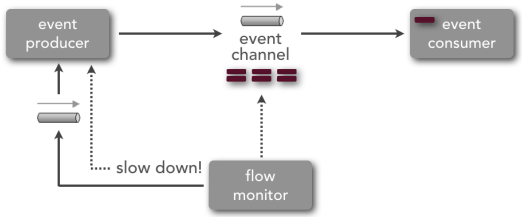
how the system reacts to events

reactive architecture

reactive architecture vs. reactive programming



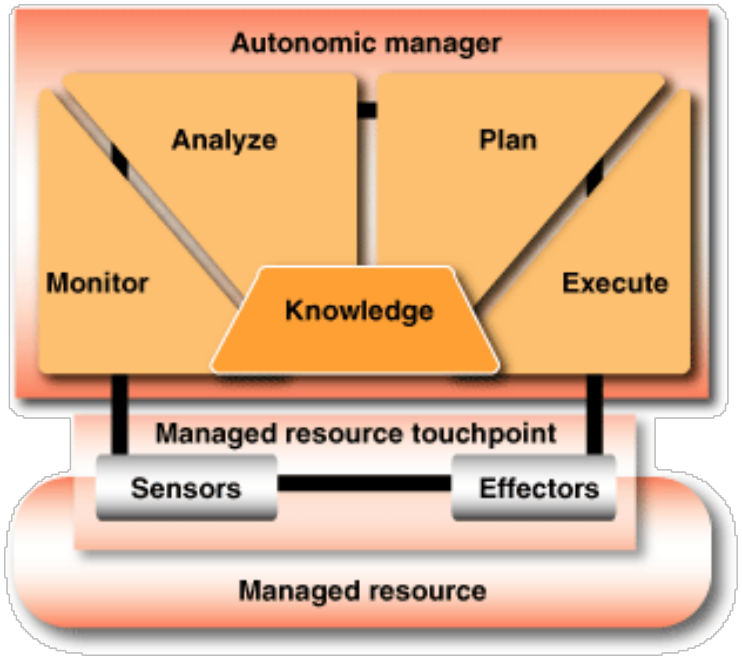




History of Autonomic Systems

autonomic systems

self-managing systems that can adapt to unpredictable changes while hiding complexity from operators or users




2001: A SPACE ODYSSEY

autonomic systems

self-managing systems that can adapt to unpredictable changes while hiding complexity from operators or users

self-configuring: automatic configuration of systems

self-healing: automatic discovery and correction of faults

self-optimization: automatic monitoring to ensure optimal functioning

autonomic systems

self-managing systems that can adapt to unpredictable changes while hiding complexity from operators or users

self-configuring: automatic configuration of systems

self-healing: automatic discovery and correction of faults

self-optimization: automatic monitoring to ensure optimal functioning

self-protection: proactive protection from arbitrary attacks

autonomic systems

self-managing systems that can adapt to unpredictable changes while hiding complexity from operators or users

self-configuring: automatic configuration of systems

self-healing: automatic discovery and correction of faults

self-optimization: automatic monitoring to ensure optimal functioning

self-protection: proactive protection from arbitrary attacks

autonomic systems

at a minimum every autonomic system should be able to exhibit the following set of properties

automatic

ability to self-control its internal functions and operations

adaptive

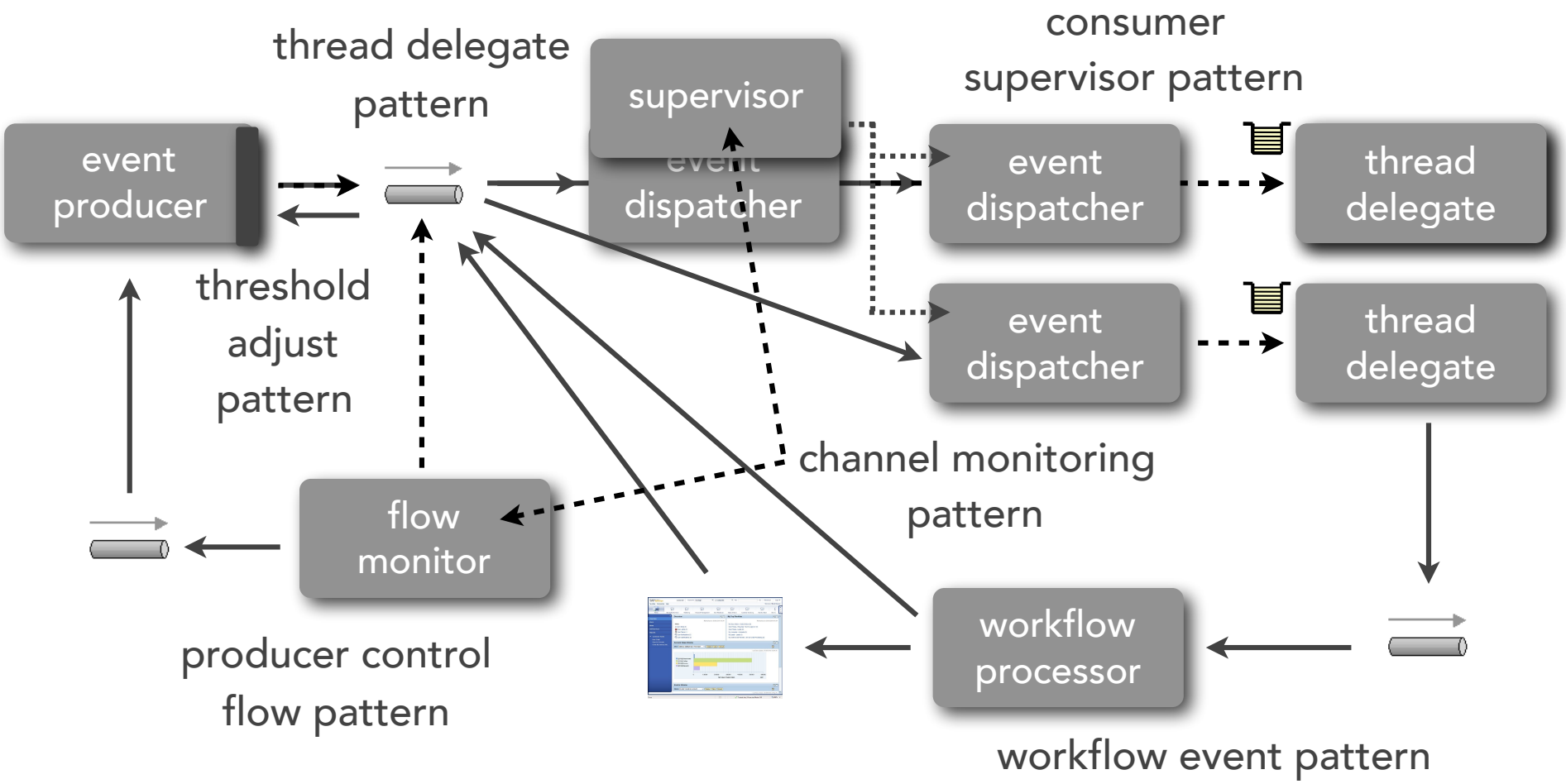
ability to change its operation and configuration

aware

ability to self-monitor and self-assess its state

reactive architecture

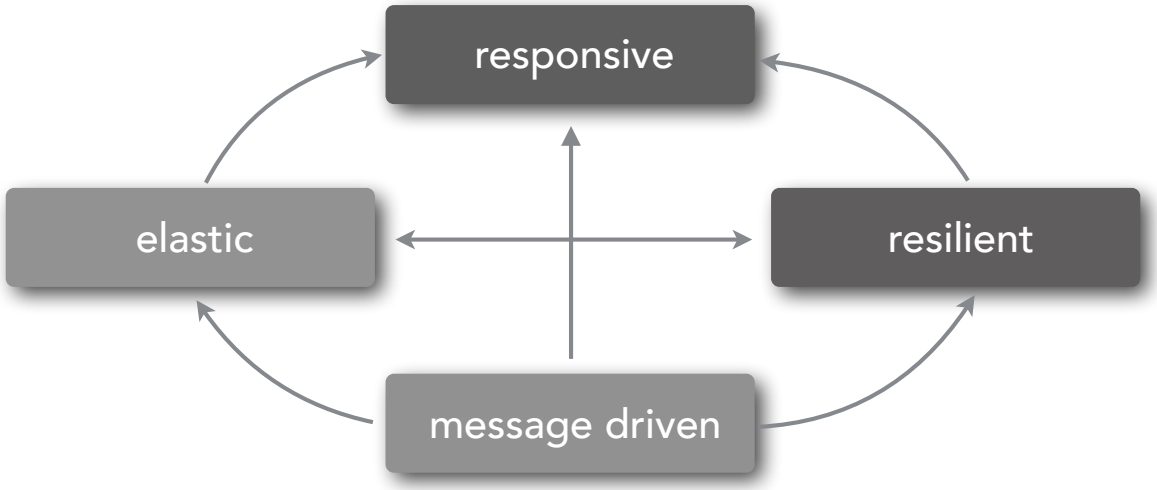
self-healing and self-monitoring systems that can automatically configure and repair themselves



Thread Delegate Pattern

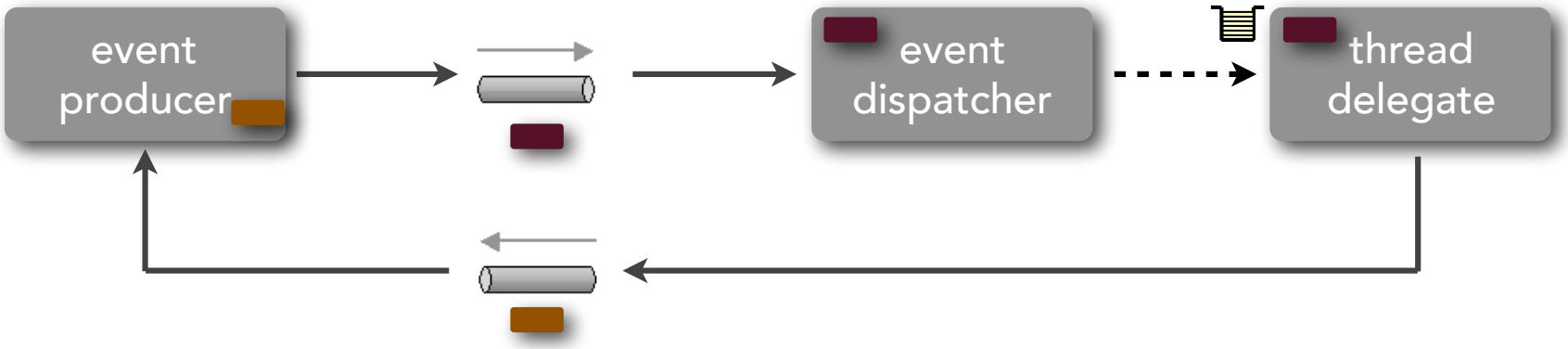
thread delegate pattern

how can you ensure timely and consistent response time as your system grows?

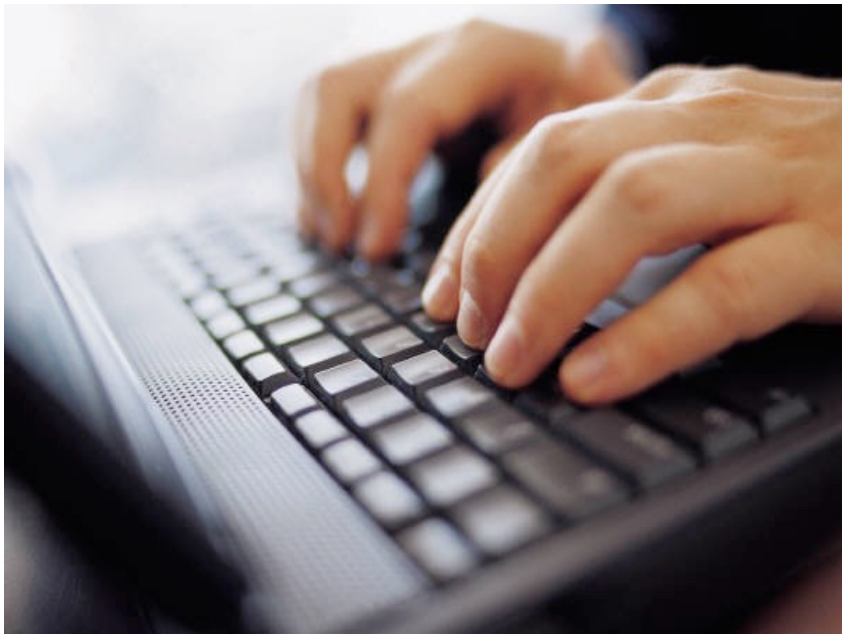


thread delegate pattern

how can you ensure timely and consistent response time as your system grows?



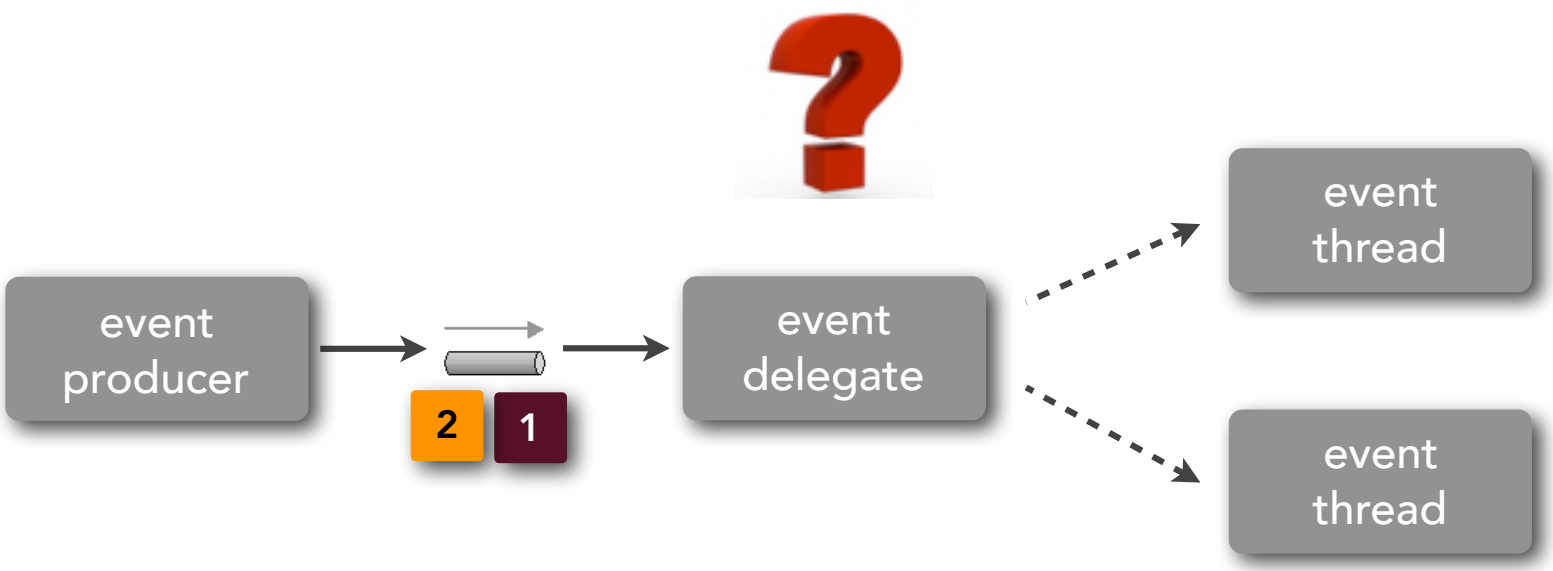
thread delegate pattern



let's see the issue…

thread delegate pattern

preserving message order



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
3. REBOOK AAPL A-136 1,800,000.00
```

`1, 2, 3`

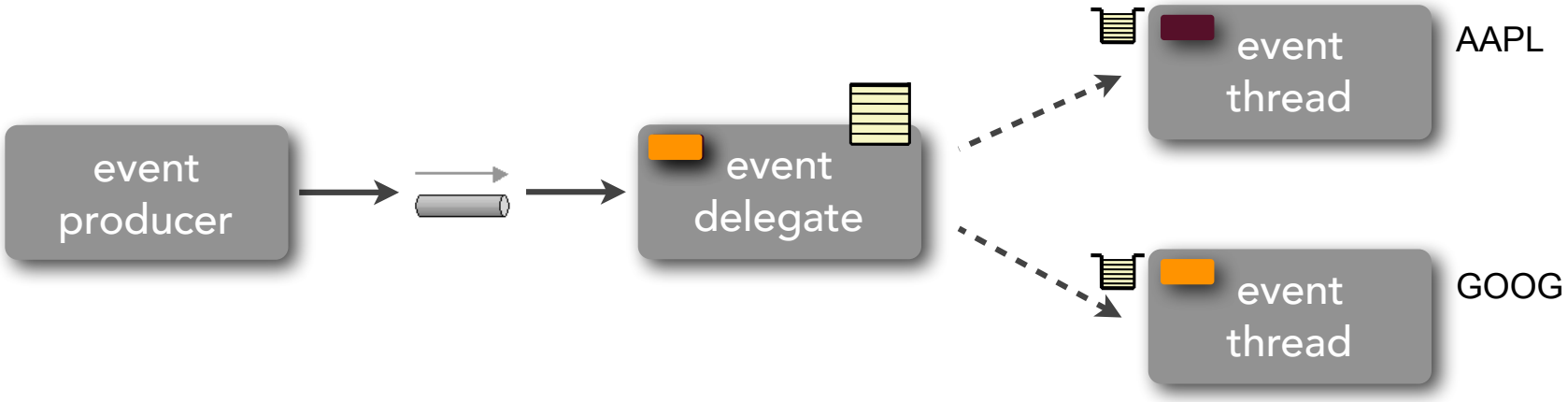
```
1. PLACE  AAPL A-136 2,000,000.00
2. PLACE  GOOG V-976   650,000.00
3. CANCEL GOOG V-976   650,000.00
4. CANCEL AAPL A-136 2,000,000.00
5. REBOOK AAPL A-136 1,800,000.00
6. REBOOK GOOG V-976   600,000.00
```

`1, 4, 5`

`2, 3, 6`

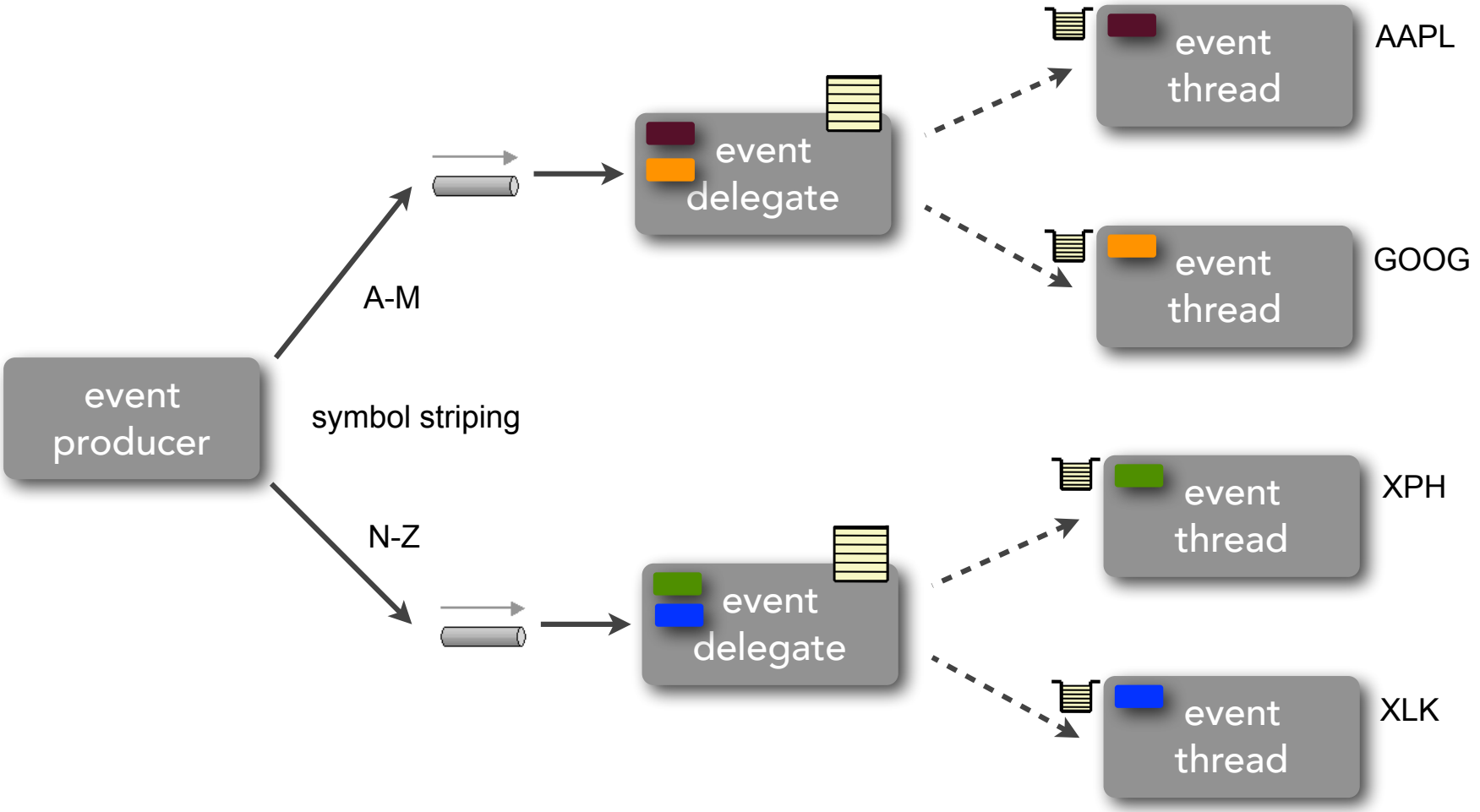
thread delegate pattern

preserving message order



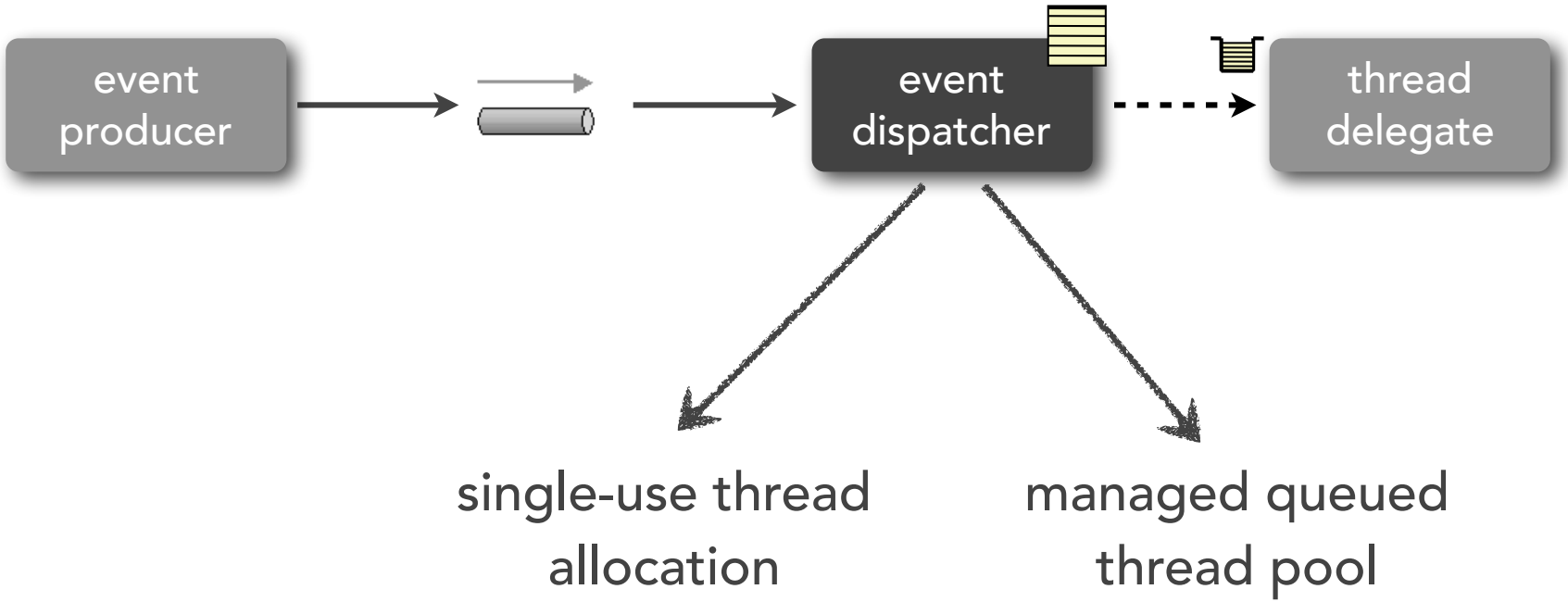
thread delegate pattern

preserving message order



thread delegate pattern

event dispatcher models

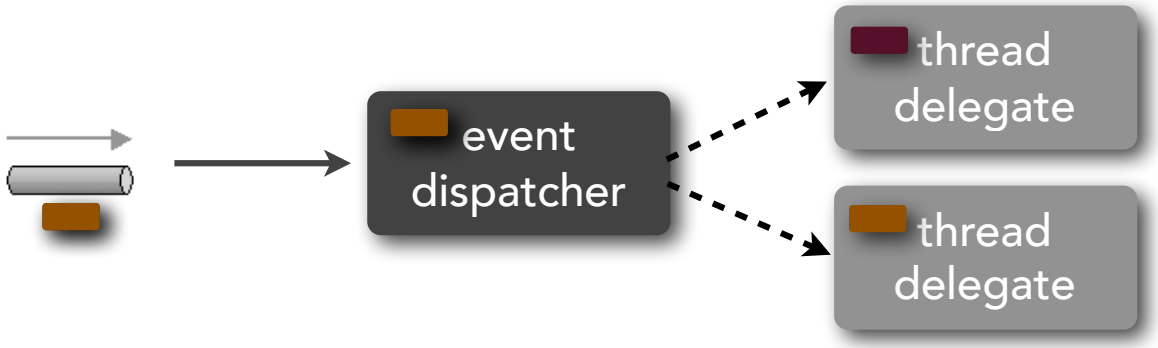


thread delegate pattern

single-use thread allocation

thread delegate pattern

single-use thread allocation



thread delegate pattern

single-use thread allocation

does not preserve message order

risk of running out of threads

risk of backing up primary delegate queue

thread delegate pattern

event dispatcher

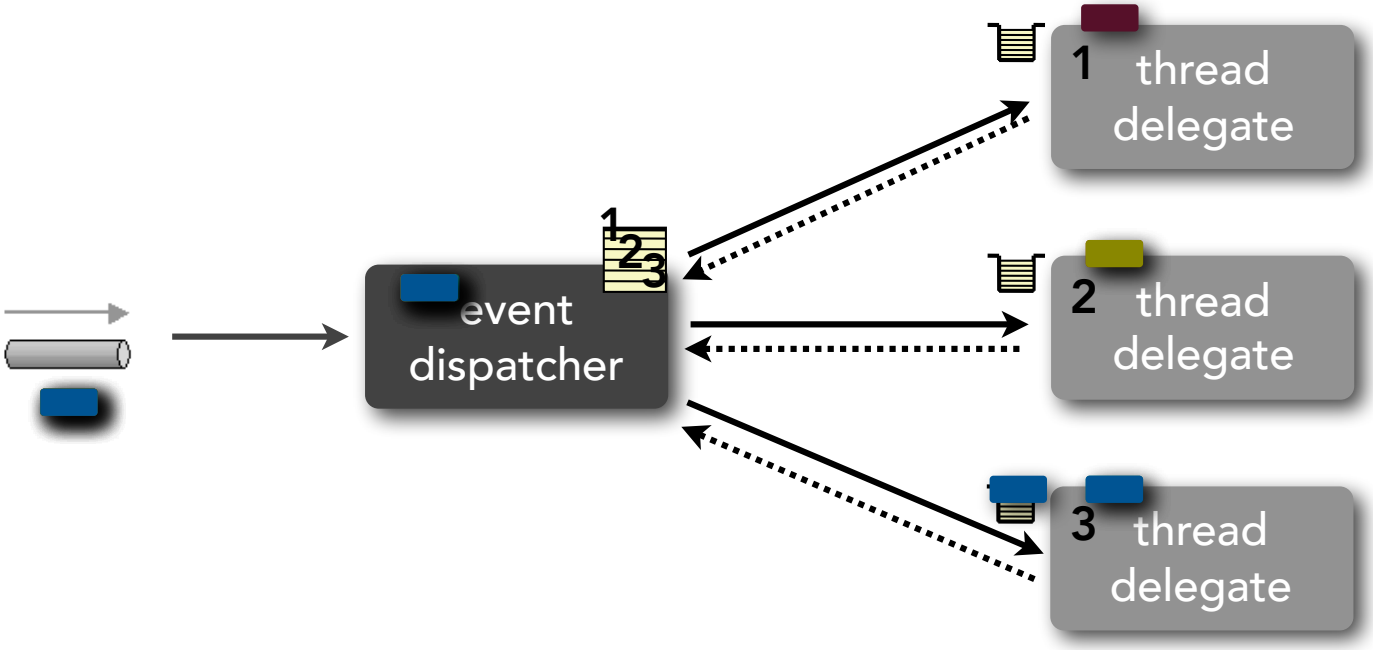
```
//connect to message broker and create consumer
while (true) {
   msg = getNextMessageFromQueue();
   new Thread(
      new ThreadProcessor(msg.getBody())).start();
}
```

thread delegate pattern

single-use thread allocation

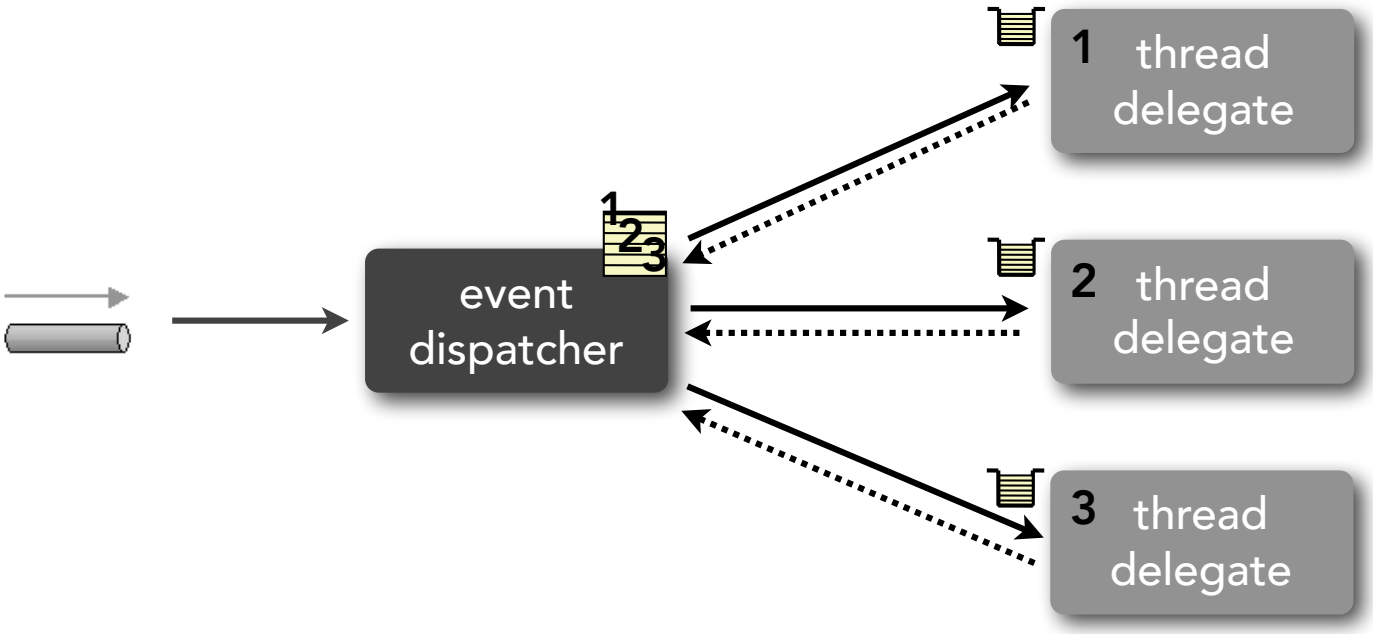
thread delegate pattern

managed queued thread pool



thread delegate pattern

managed queued thread pool



thread delegate pattern

event dispatcher

```
<bean id="thread1"
   class="org.springframework.scheduling.concurrent.
   ThreadPoolTaskExecutor">
     <property name="corePoolSize" value="1" />
     <property name="maxPoolSize" value="1" />
     <property name="queueCapacity" value="100" />
  </bean>
```

this makes it single-threaded
to preserve message order

thread delegate pattern

event dispatcher

```
//holds threads created by dispatcher
private List<TaskExecutor> threads =
   new ArrayList<TaskExecutor>();
private int index = 0;

//<symbol, thread instance>
private Map<String, Object> allocationMap =
  new HashMap<String, Object>();
```

thread delegate pattern

event dispatcher

```
//STARTUP LOGIC

threads.add((TaskExecutor)ctx.getBean("thread1"));
threads.add((TaskExecutor)ctx.getBean("thread2"));
threads.add((TaskExecutor)ctx.getBean("thread3"));

//connect to message broker and create consumer
//wait for messages...
```

thread delegate pattern

event dispatcher

```
//DEQUEUE AND MESSAGE ASSIGNMENT LOGIC

msg = getNextMessageFromQueue();
String symbol = //get symbol from message properties
TaskExecutor thread = null;
if (allocationMap.containsKey(symbol)) {
   thread = (TaskExecutor)allocationMap.get(symbol);
} else {
   index = (index == threads.size()-1) ? 0 : index+1;
   thread = threads.get(index);
   allocationMap.put(symbol, thread);
}
thread.execute(new TradeProcessorThread(this, msg));
```

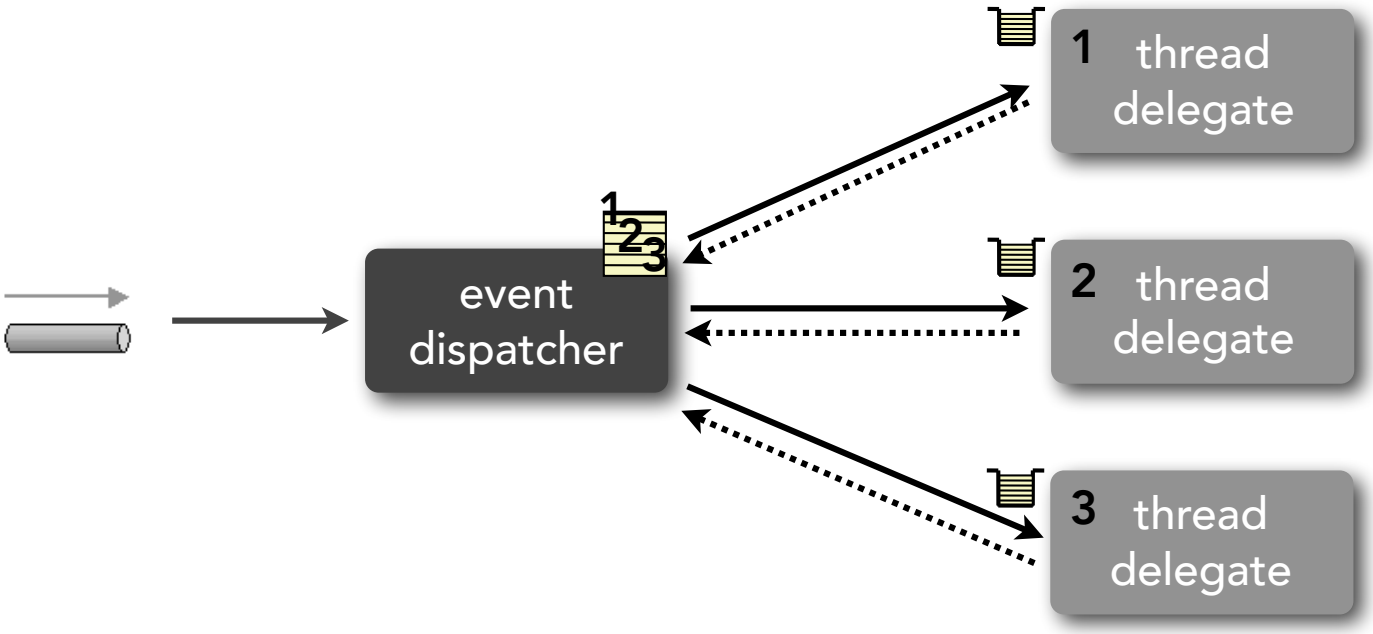
thread delegate pattern

event dispatcher

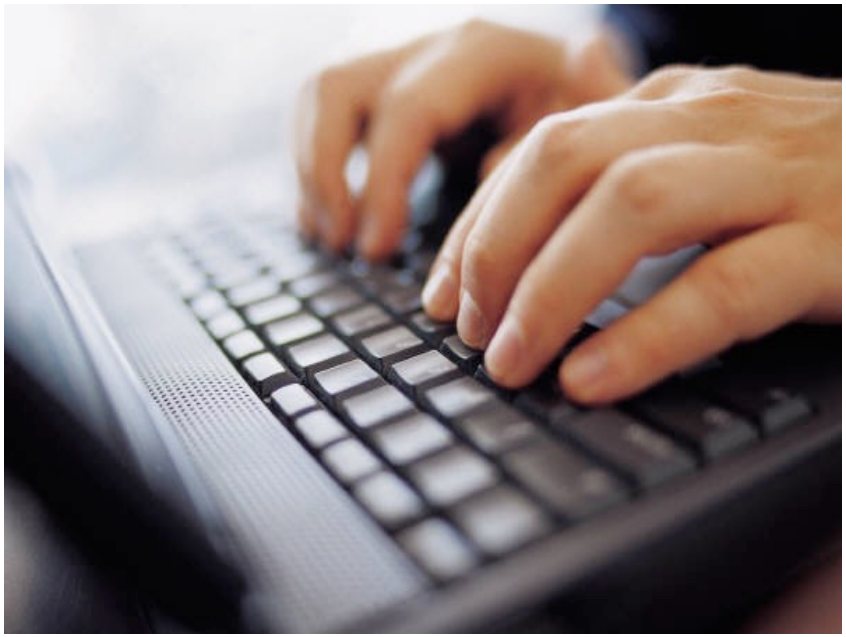
```
//THREAD CALLBACK AND ALLOCATION MAP CLEANUP LOGIC

public void markRequestComplete(String symbol) {
   //set timer on allocation map to un-assign thread after
   //a period of inactivity for that symbol

   //if a symbol comes in during the timer period,
   //cancel the timer

   allocationMap.remove(symbol);
 }
```

thread delegate pattern

managed queued thread pool



thread delegate pattern

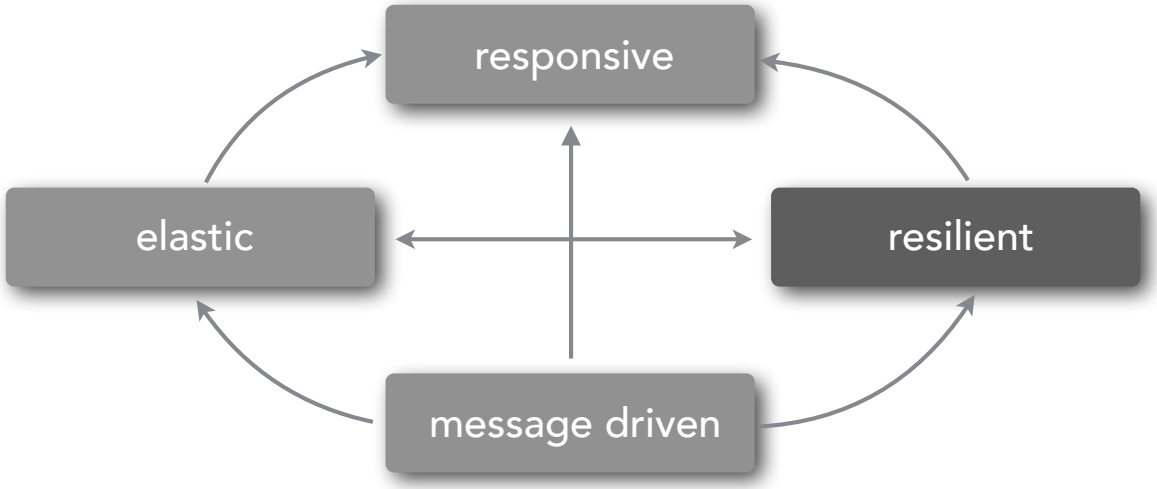


let’s see the result…

Workflow Event Pattern

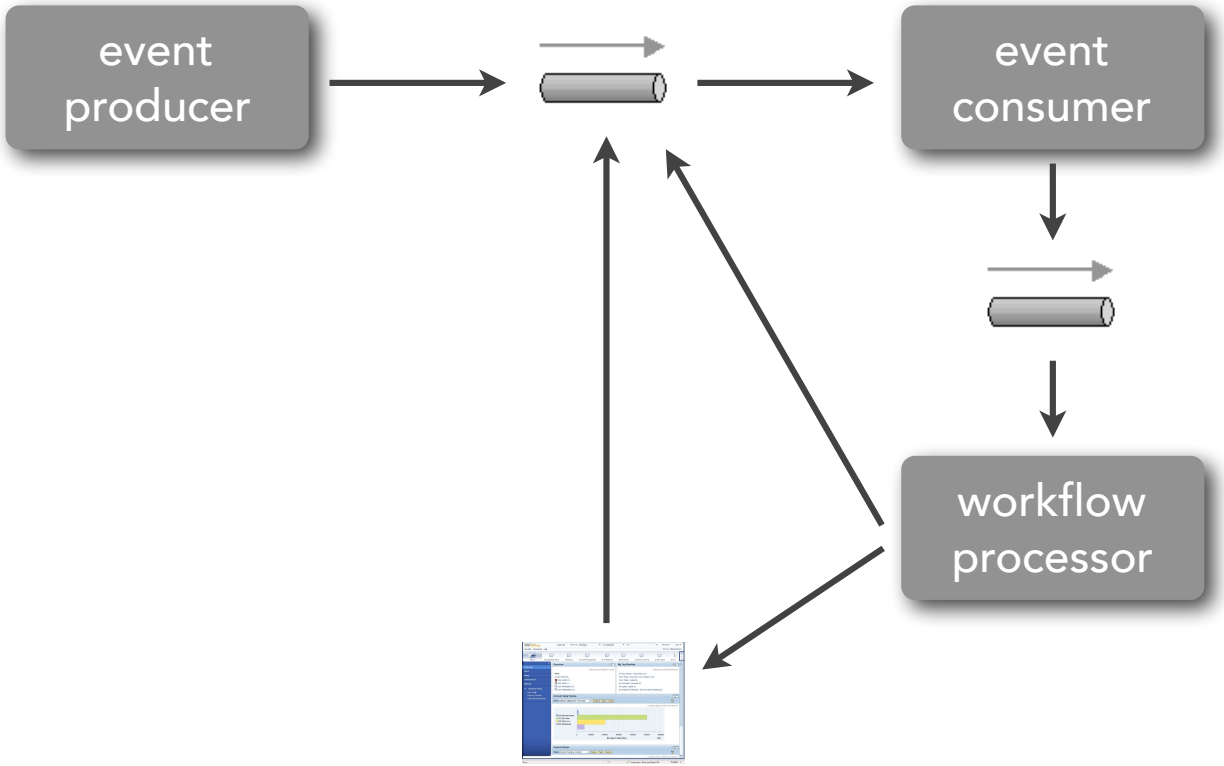
workflow event pattern

how can you handle error conditions without failing the transaction?



workflow event pattern

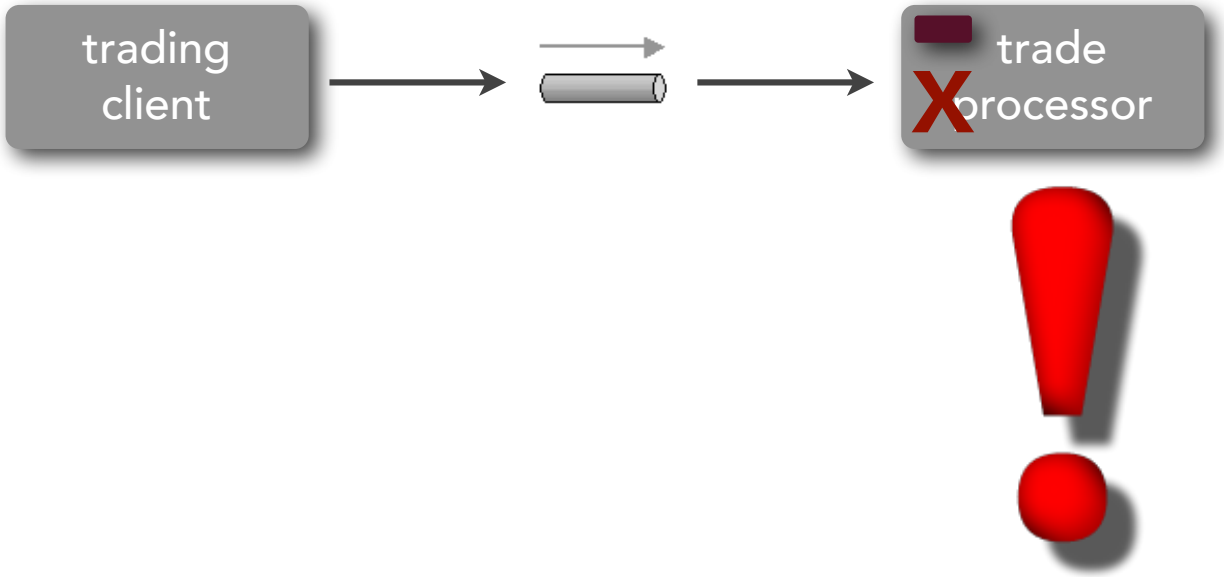
how can you handle error conditions without failing the transaction?



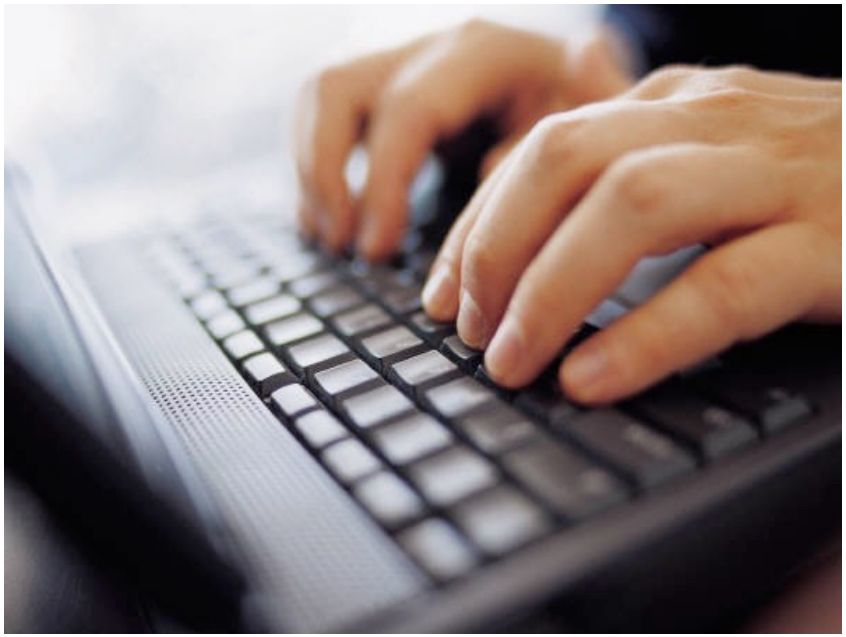
workflow event pattern

example

while asynchronously processing trades an error occurs with one of the trade orders



workflow event pattern

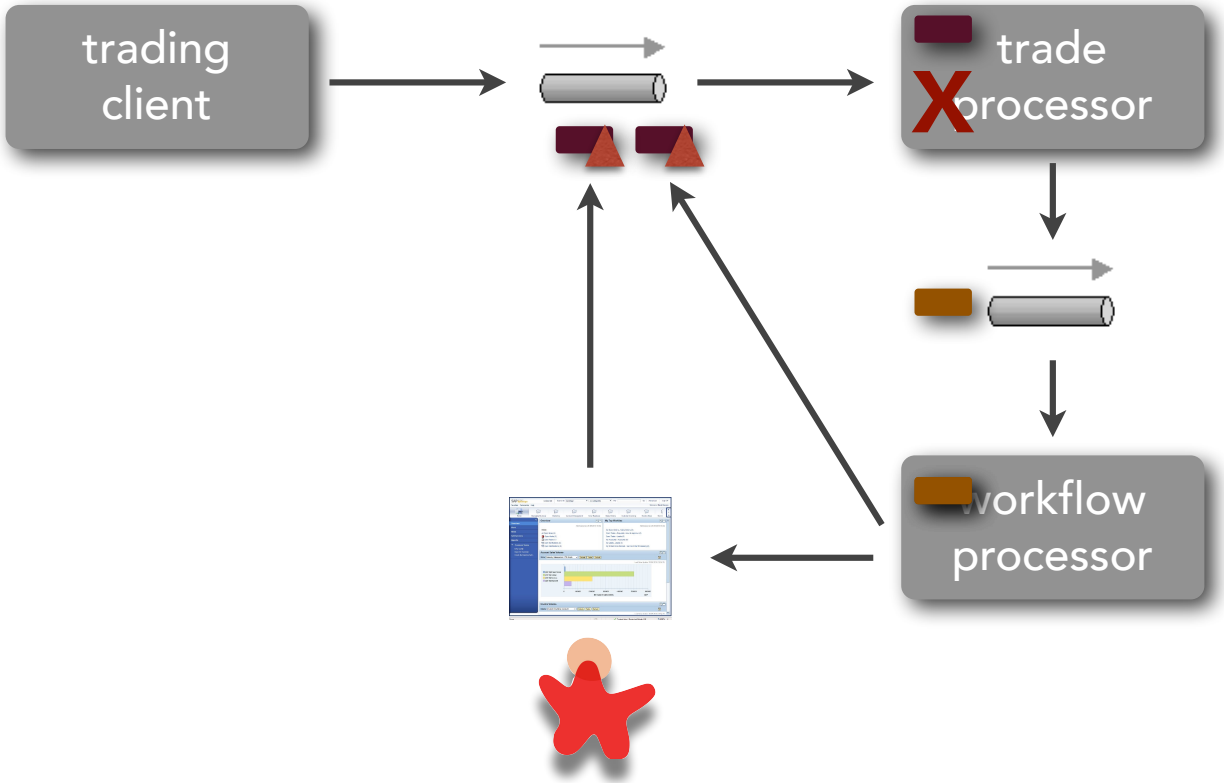


let's see the issue…

workflow event pattern

example

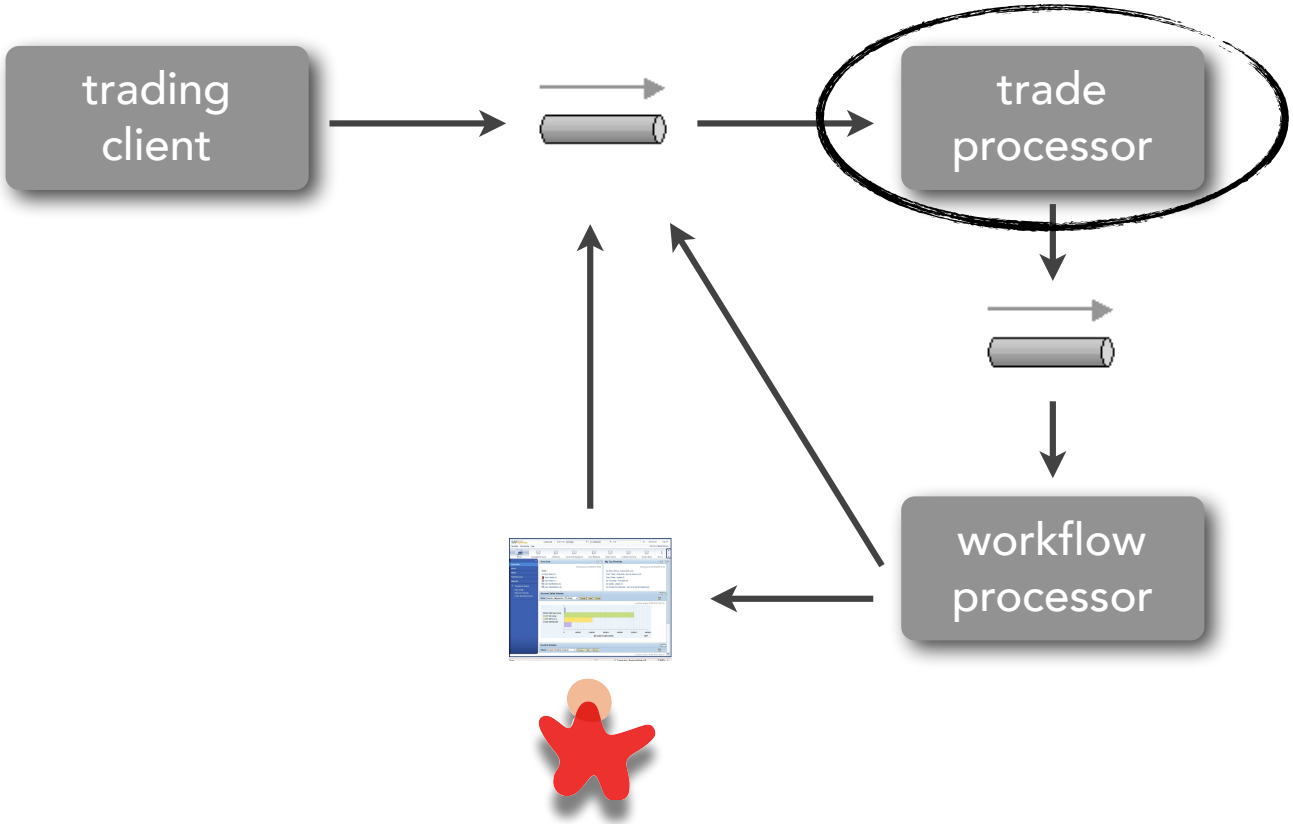
while asynchronously processing trades an error occurs with one of the trade orders



workflow event pattern

example

while asynchronously processing trades an error occurs with one of the trade orders



workflow event pattern

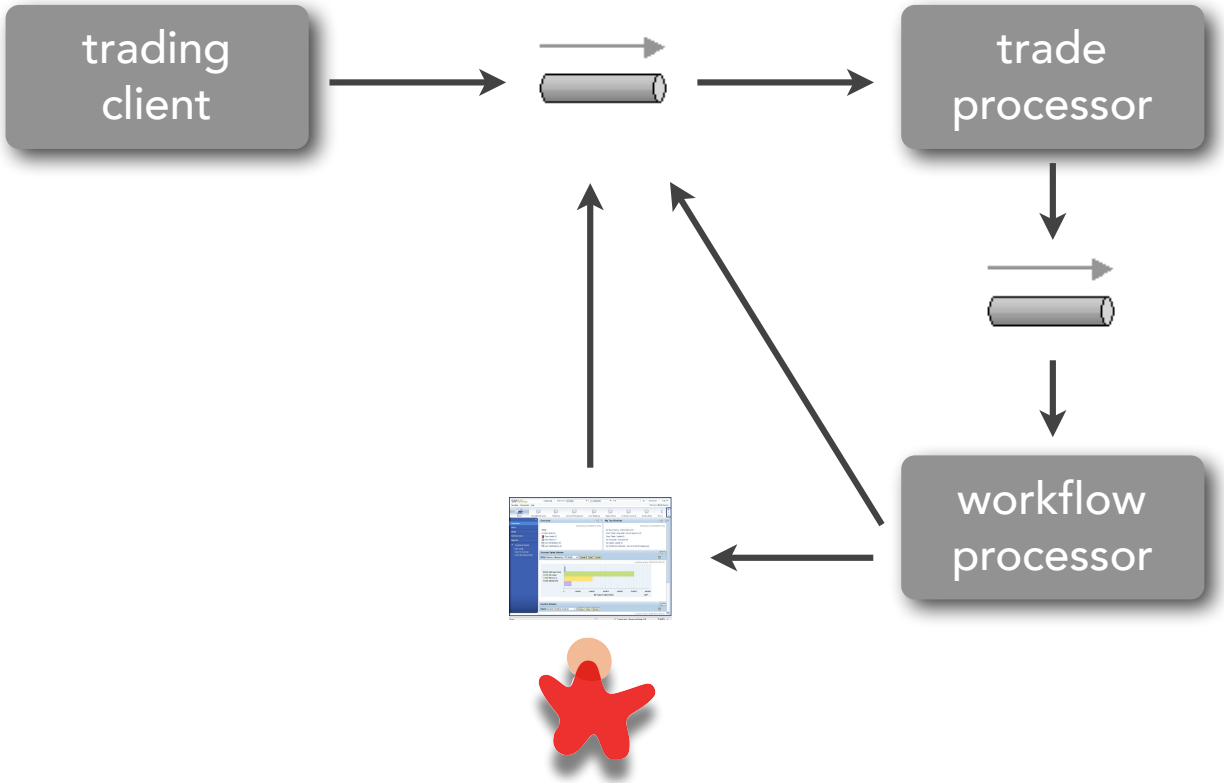
trade processor

```
try {
   msg = getNextMessageFromQueue();
   String message = new String(msg.getBody());
   String[] parts = message.split(",");
   long shares = new Long(parts[2]).longValue();
   ...
} catch (Exception e) {
   System.out.println("sending to workflow");
   //send original message to workflow queue with error
}
```

workflow event pattern

example

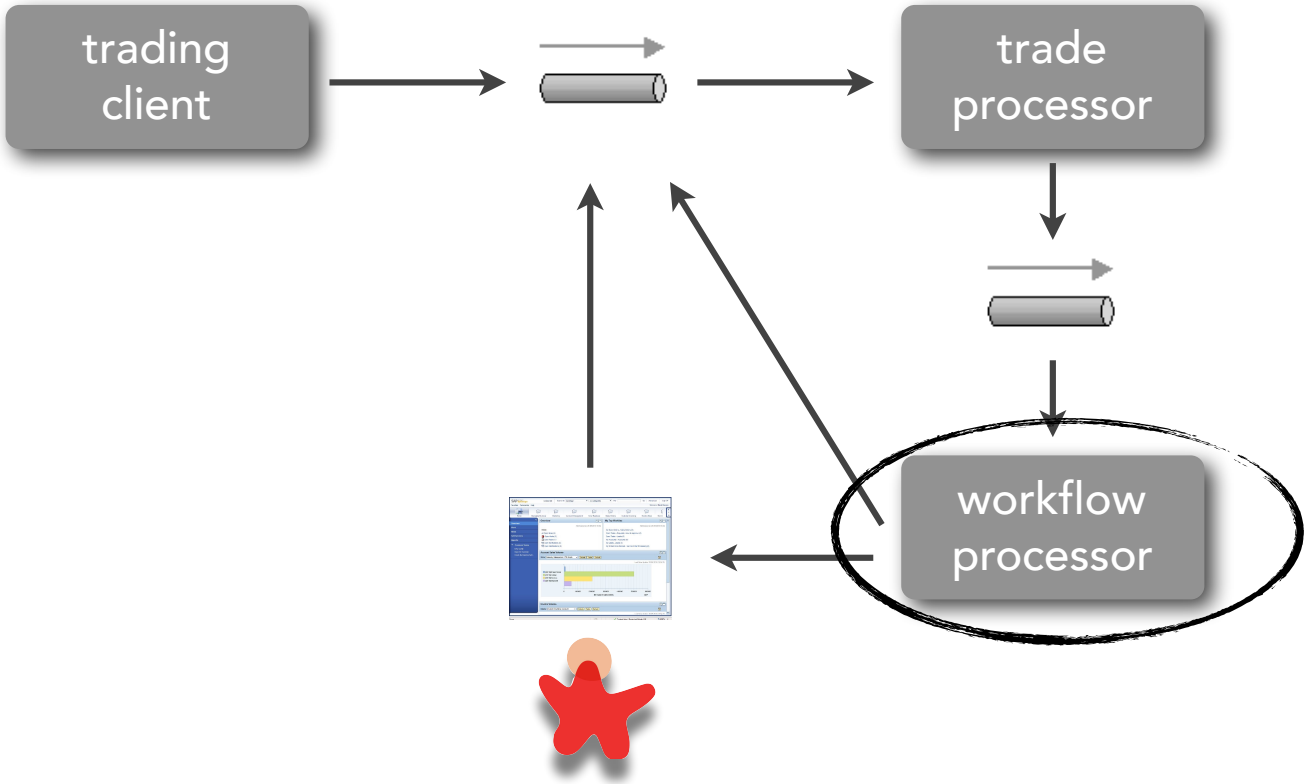
while asynchronously processing trades an error occurs with one of the trade orders



workflow event pattern

example

while asynchronously processing trades an error occurs with one of the trade orders



workflow event pattern

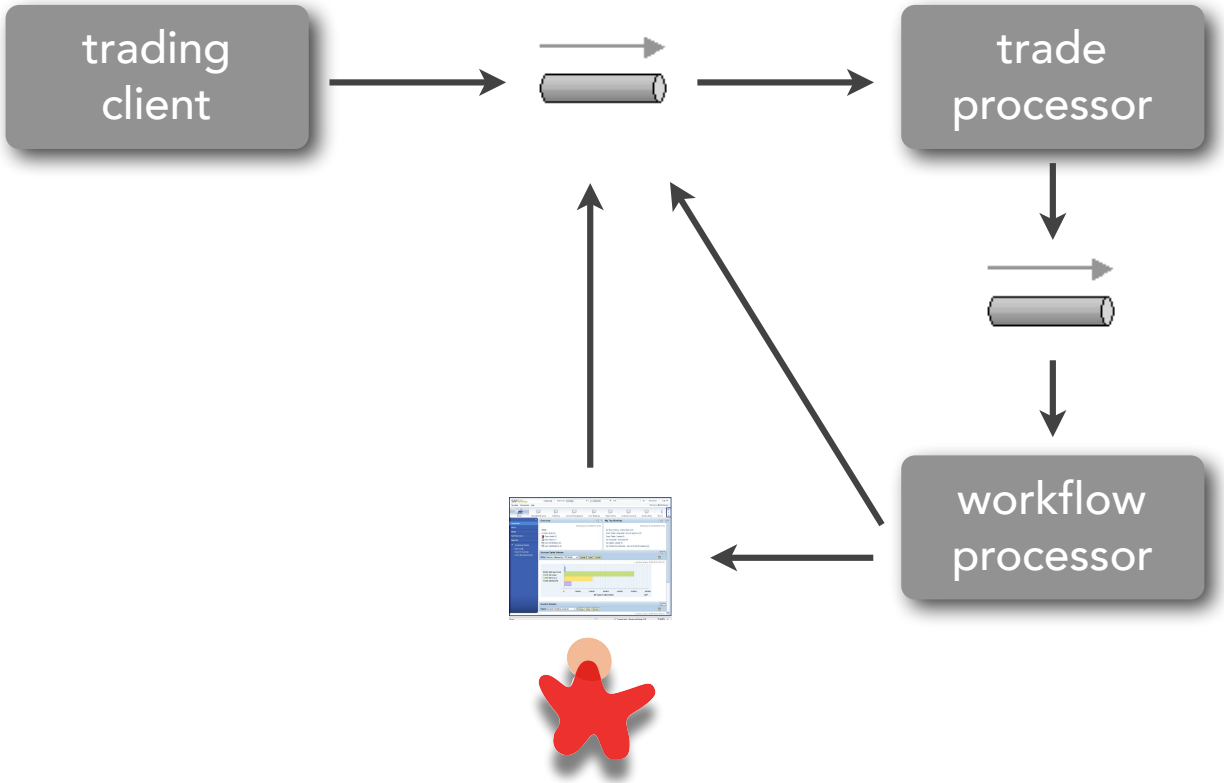
workflow processor

```
msg = getNextMessageFromQueue();
//detect the error type and if it can be repaired
if (SHARES_ERROR) {
   String msg = new String(message.getBody());
   String newMsg = msg.substring(0, msg.indexOf(" shares"));
   System.out.println("Trade fixed: " + newMsg);
   //send new message back to original request queue
} else {
  //send message to dashboard for manual fixing
}
```

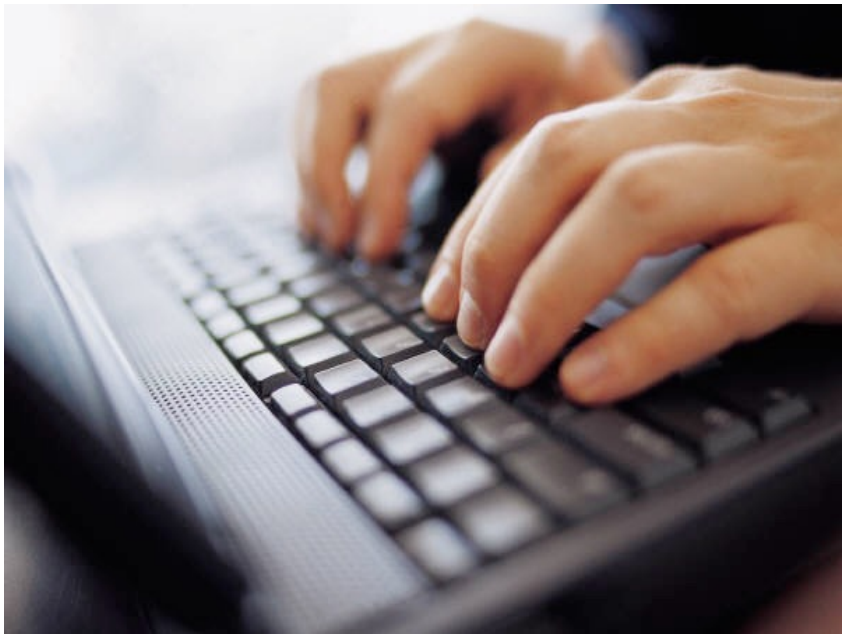
workflow event pattern

example

while asynchronously processing trades an error occurs with one of the trade orders



workflow event pattern

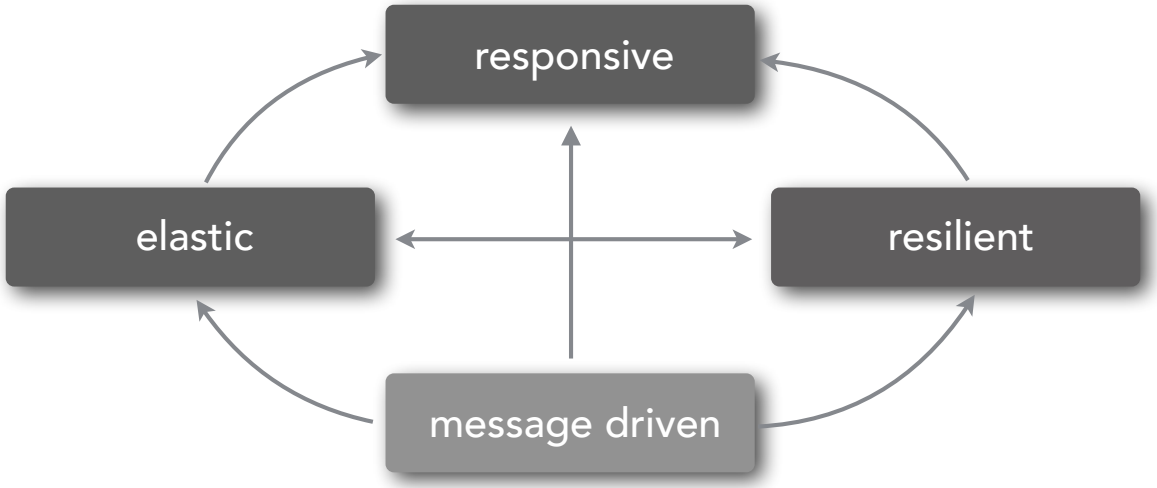


let’s see the result…

Channel Monitor Pattern

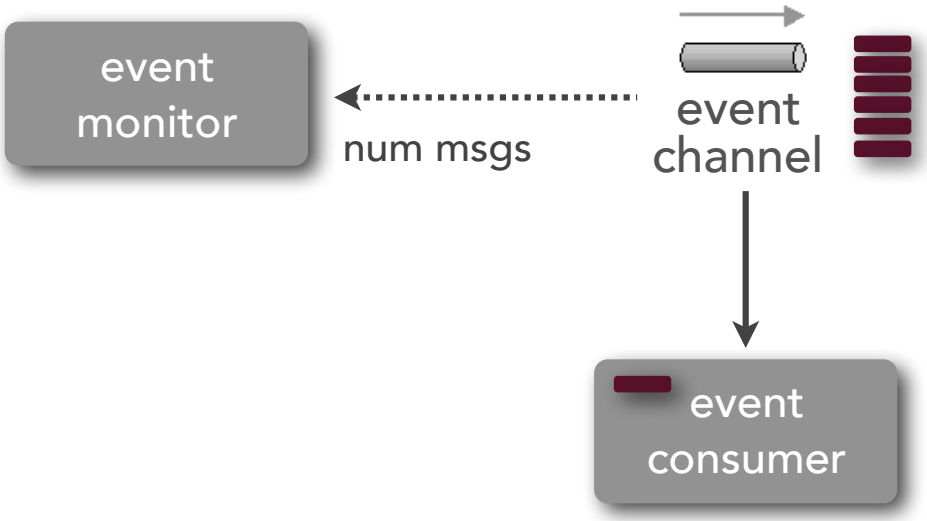
channel monitor pattern

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

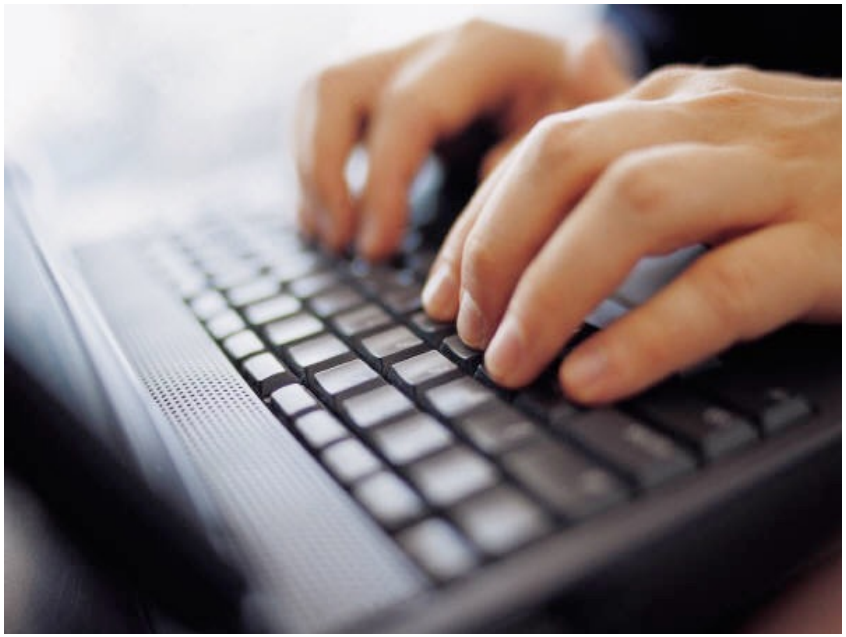


channel monitor pattern

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



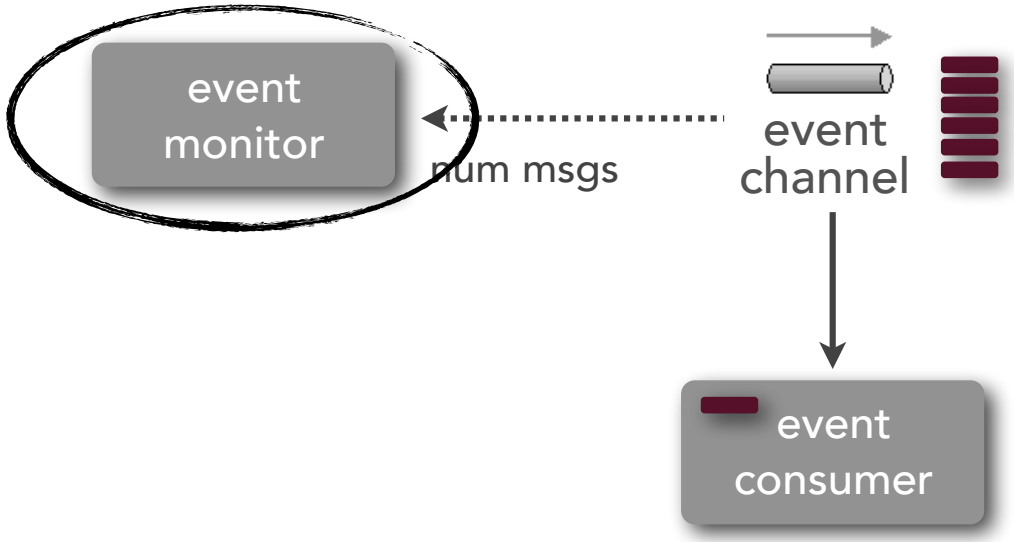
channel monitor pattern



let’s see the issue…

channel monitor pattern

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



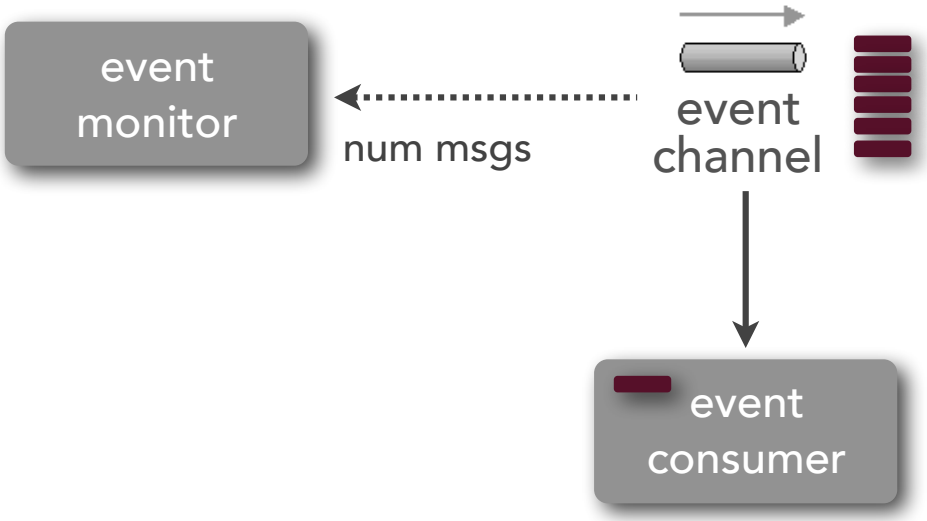
channel monitor pattern

event monitor

```
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();
```

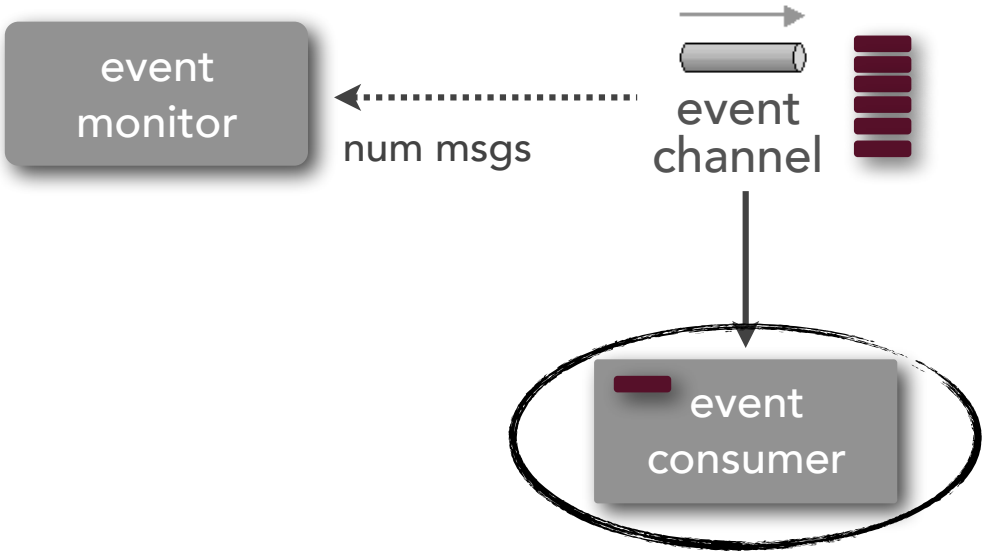
channel monitor pattern

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?



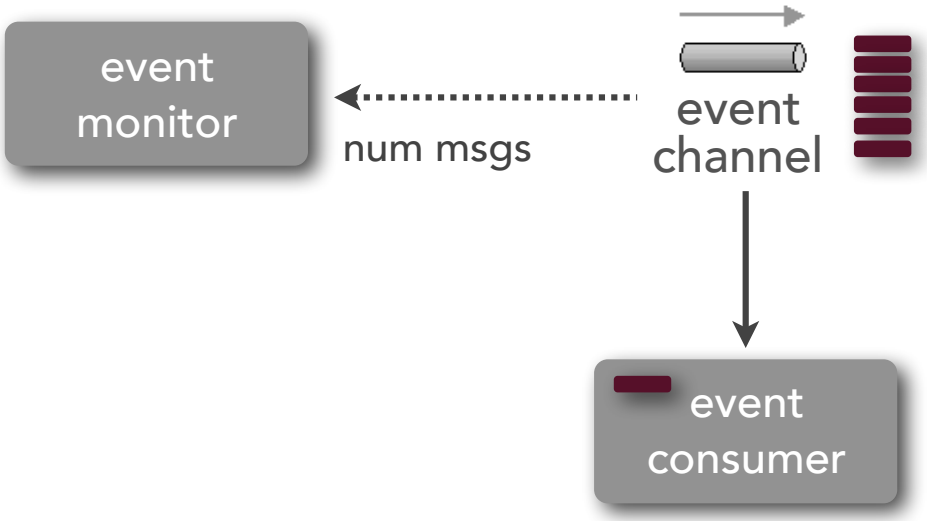
channel monitor pattern

event consumer

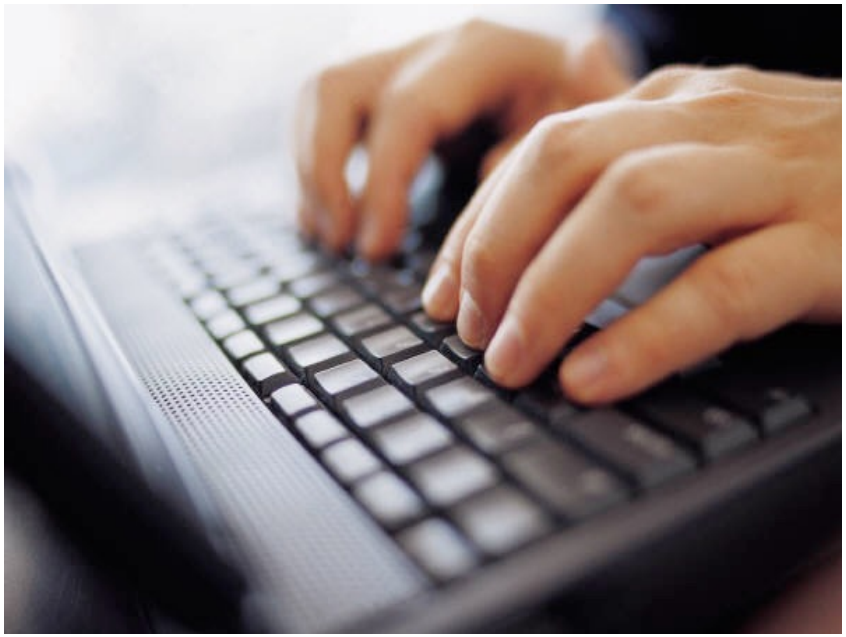
```
Channel channel = AMQPCommon.connect();
channel.basicQos(1);
channel.basicConsume("trade.eq.q", false, consumer);
QueueingConsumer.Delivery msg = consumer.nextDelivery();
channel.basicAck(msg.getEnvelope().getDeliveryTag(), false);
```

channel monitor pattern

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



channel monitor pattern

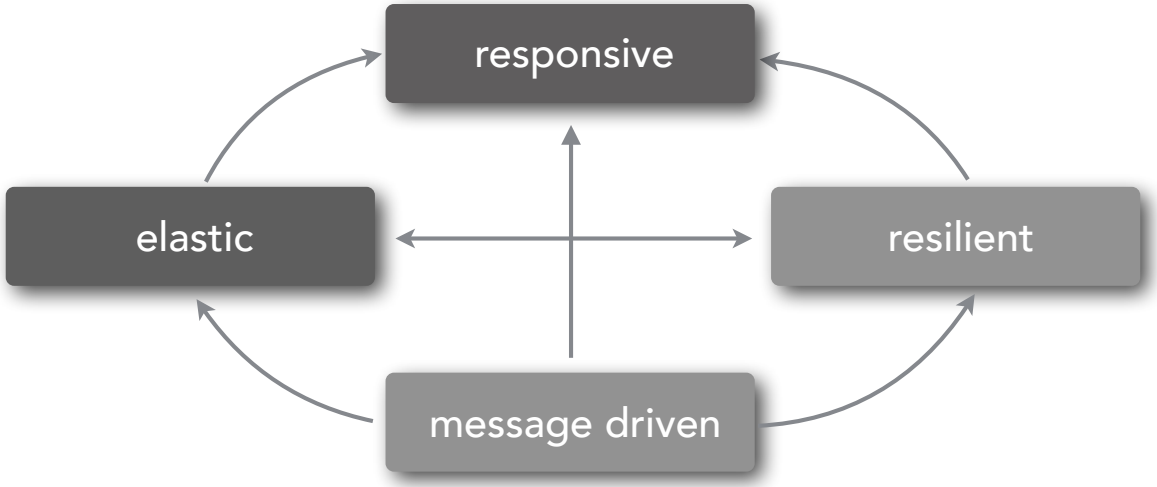


let’s see the result…

Consumer Supervisor Pattern

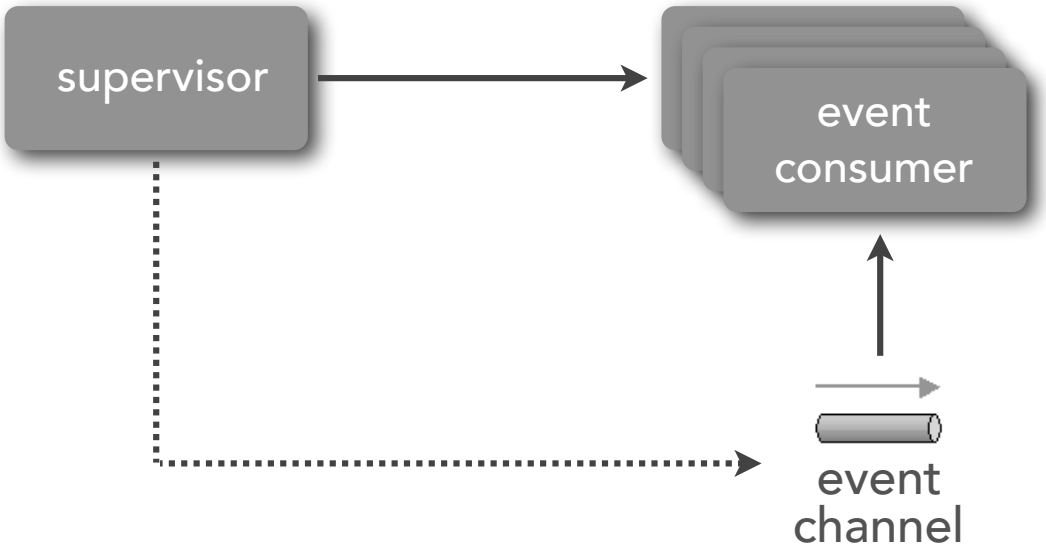
consumer supervisor pattern

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

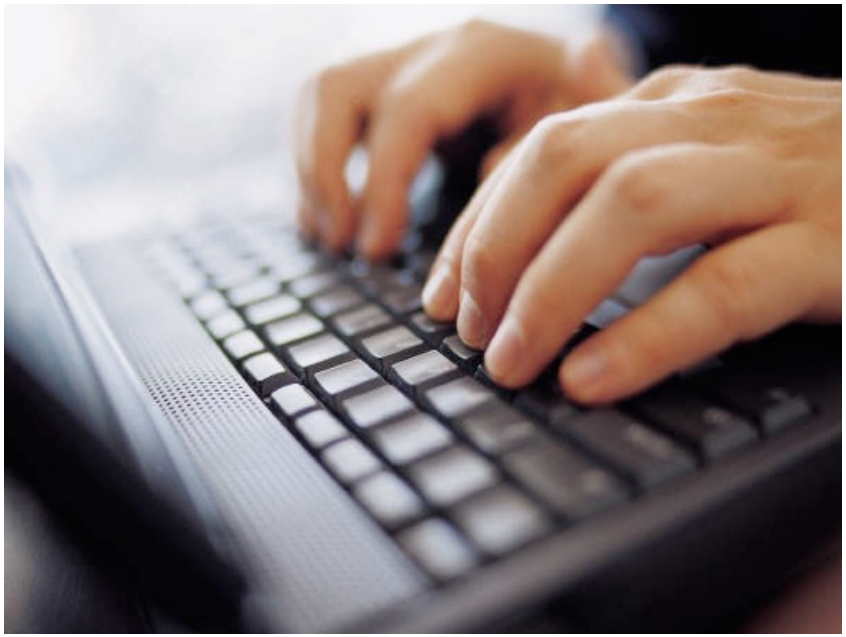


consumer supervisor pattern

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

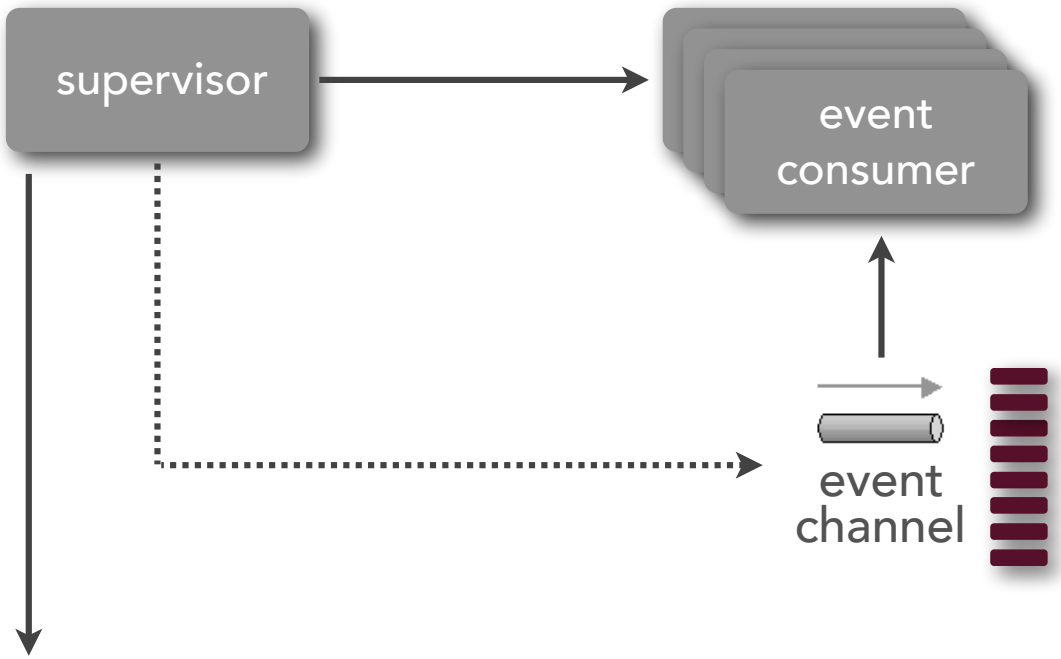


consumer supervisor pattern



let’s see the issue….

consumer supervisor pattern



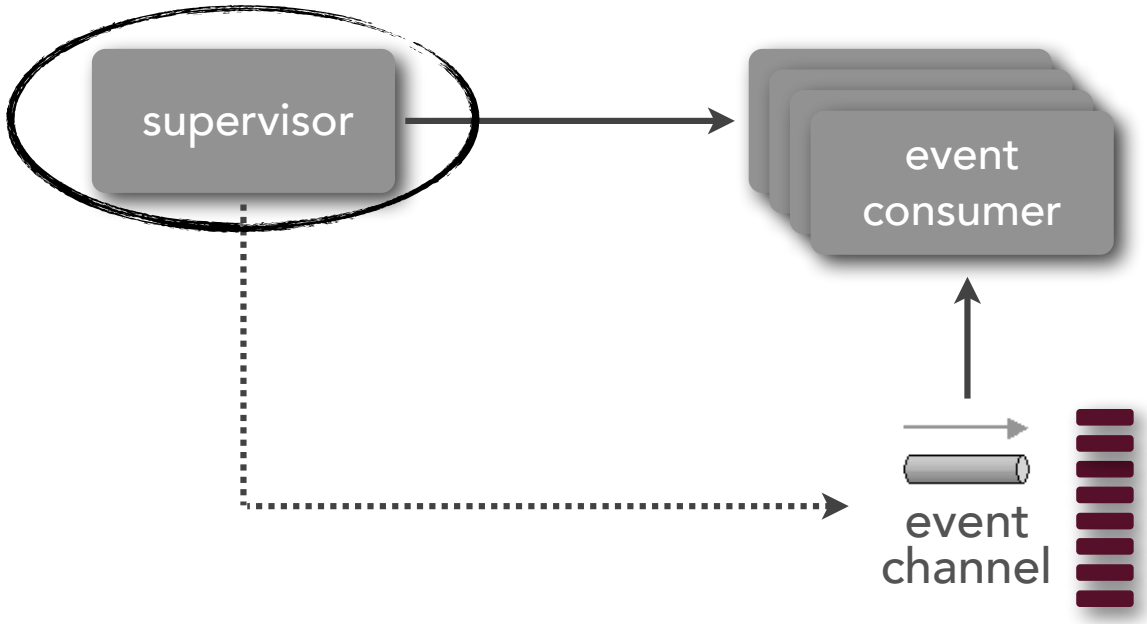
continually monitor queue depth (e.g., 1000ms)

determine consumers needed (e.g., depth/2)

apply max threshold (e.g., 1000 msgs)

add or remove consumers as needed

consumer supervisor pattern



consumer supervisor pattern

supervisor

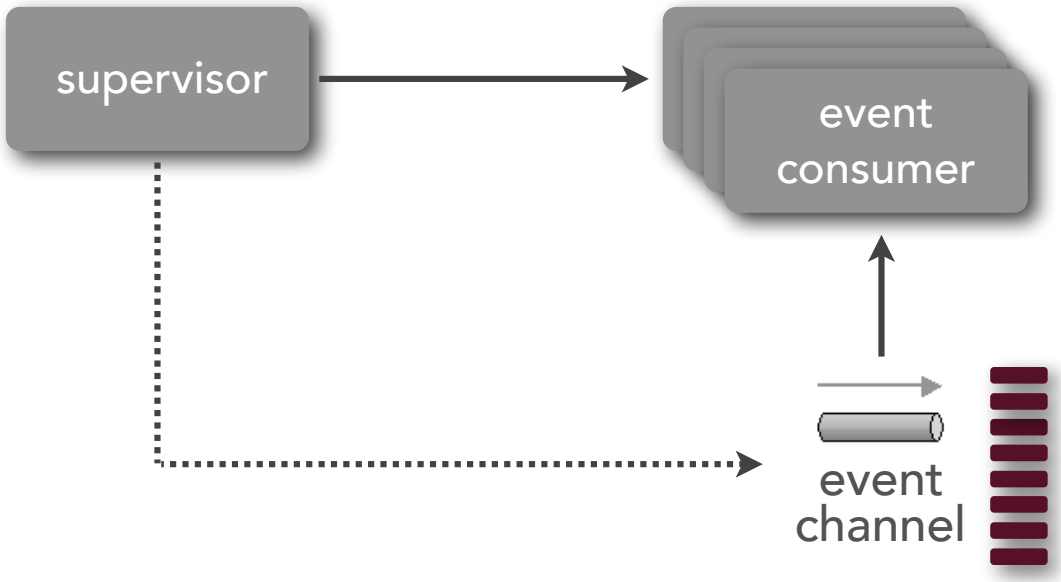
```
private List<MyConsumer> consumers =
    new ArrayList<MyConsumer>();

public void execute() throws Exception {
    //connect to message broker
    startConsumer();
    while (true) {
        long consumersNeeded = getMsgCount("trade.eq.q")/2;
        if (consumersNeeded > 1000) consumersNeeded = 1000;
        long diff = Math.abs(consumersNeeded - consumers.size());
        if (consumersNeeded > consumers.size()) {
            for (int i=0;i<diff;i++) { startConsumer(); }
        } else {
            for (int i=0;i<diff;i++) { stopConsumer(); }
        }
        Thread.sleep(1000);
    }}
```

consumer supervisor pattern

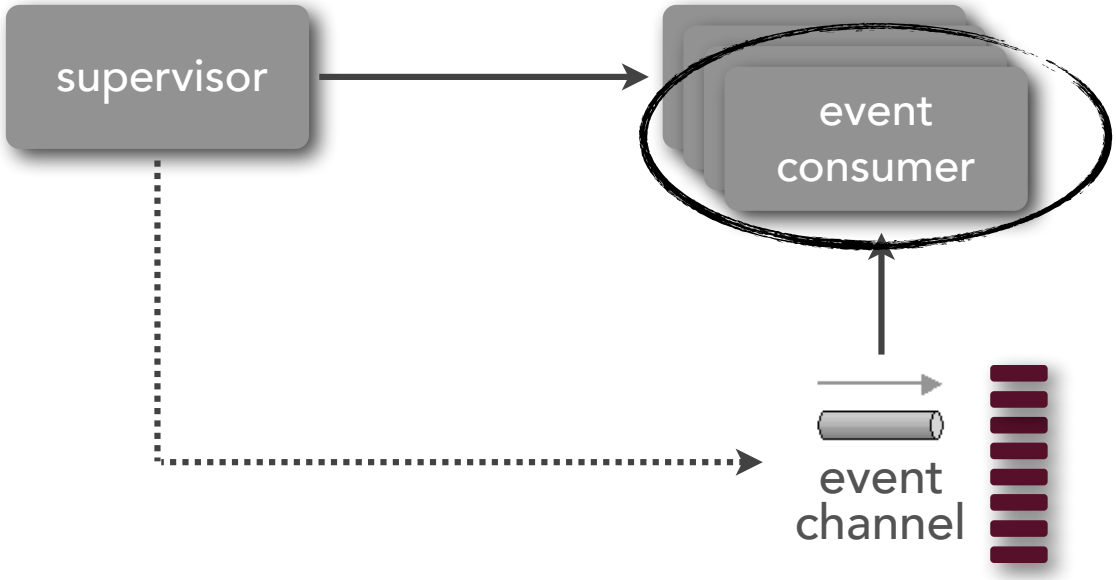
supervisor

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

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

consumer supervisor pattern



consumer supervisor pattern

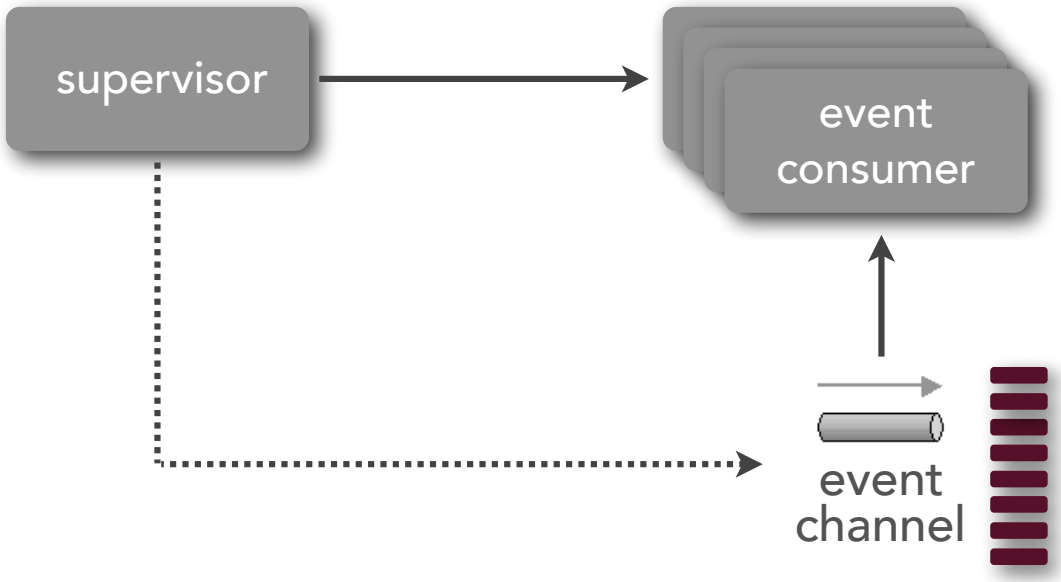


consumer supervisor pattern

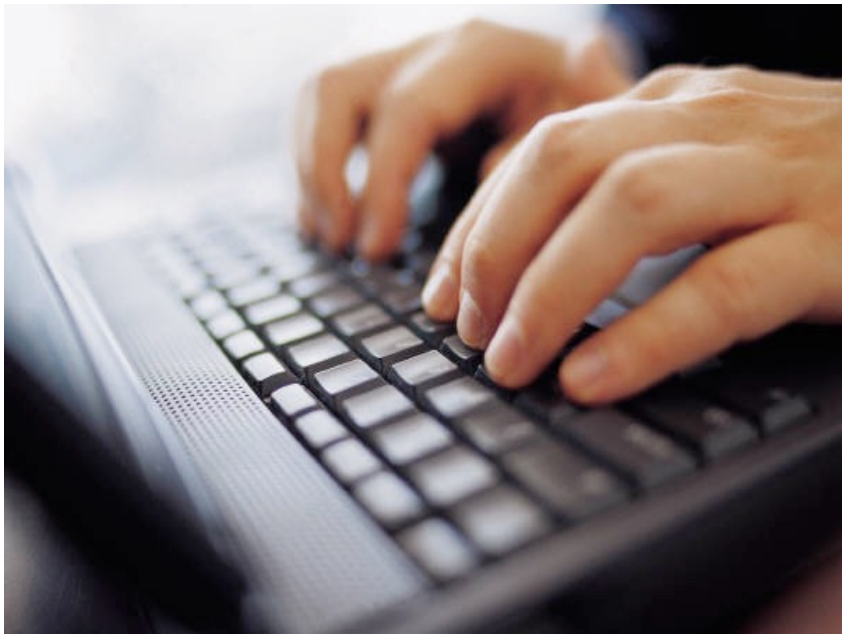
event consumer

```
private Boolean active = true;

public void startup(Connection connection) {
   //get broker session or channel and create consumer
   while (active) {
      msg = getNextMessageFromQueue(10000);
      if (msg != null)
        //process message
   }
   //close broker session or channel
}

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

consumer supervisor pattern



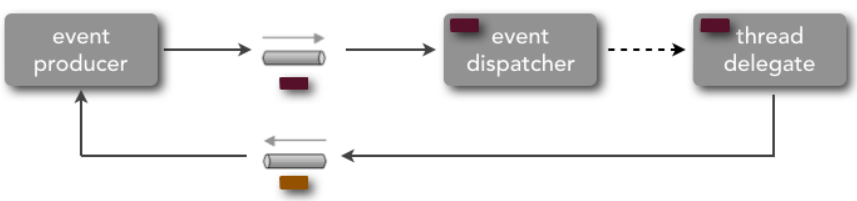
consumer supervisor pattern

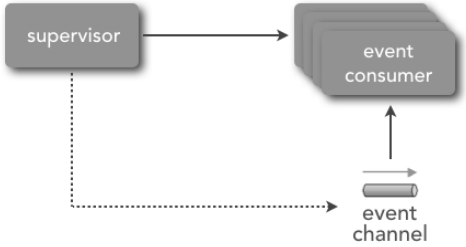


let’s see the result…

thread delegate pattern

thread delegate vs. consumer supervisor





scalability

consistent consumers

decoupled event processors

near-linear performance

elasticity

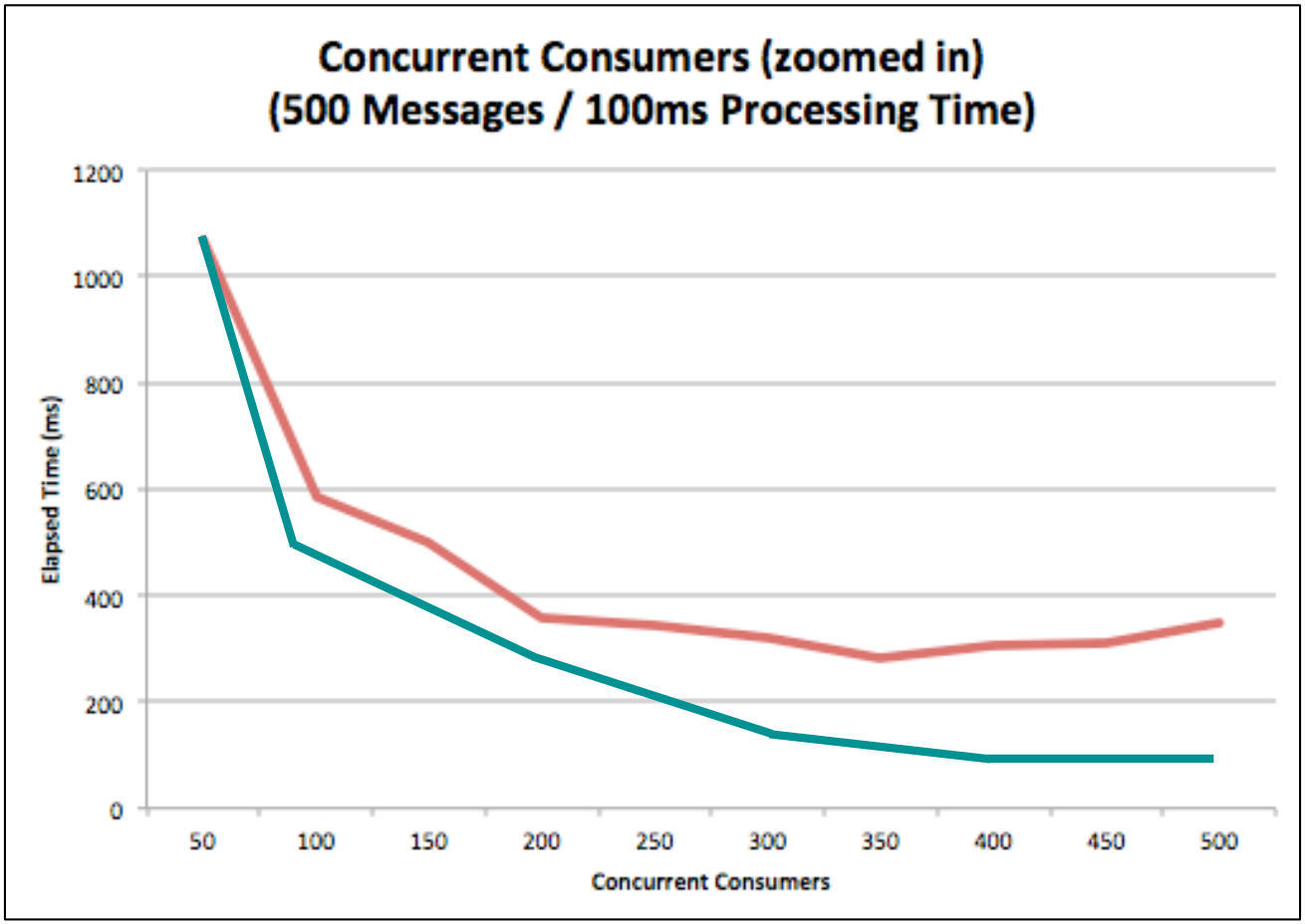
variable consumers

coupled event processors

diminishing performance

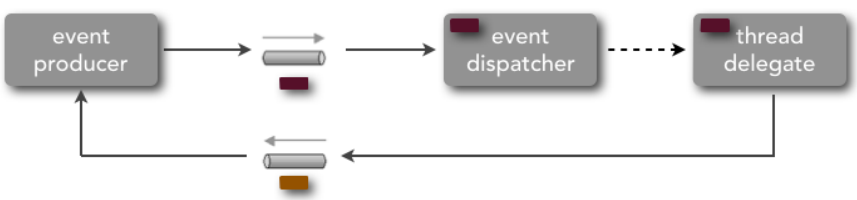
thread delegate pattern

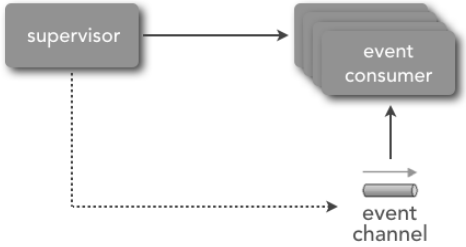
thread delegate vs. consumer supervisor



thread delegate pattern

thread delegate vs. consumer supervisor





scalability

consistent consumers

decoupled event processors

near-linear performance

can preserve message order

elasticity

variable consumers

coupled event processors

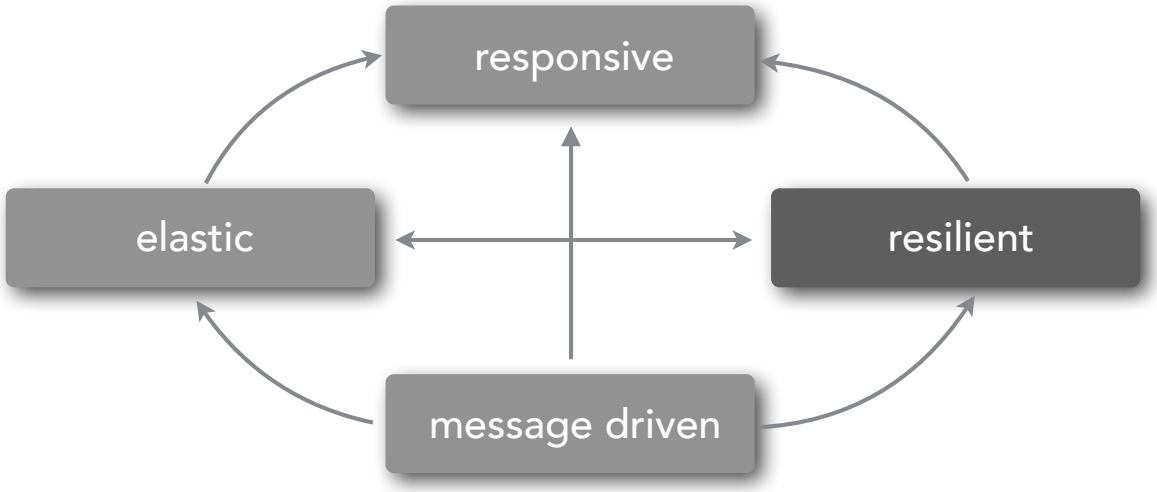
diminishing performance

message order not preserved

Producer Control Flow Pattern

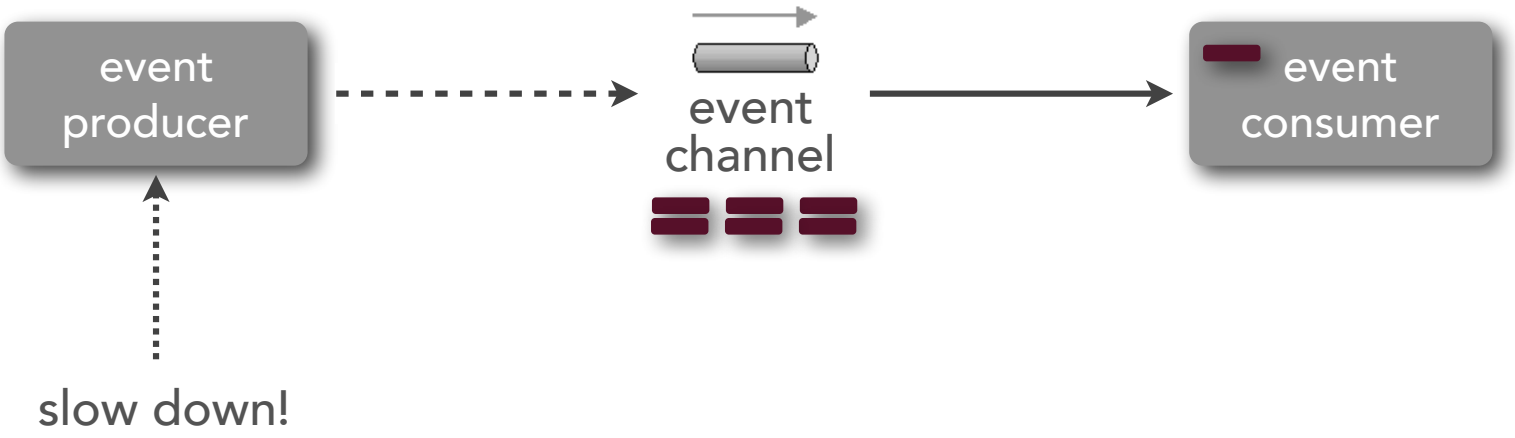
producer control flow pattern

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?

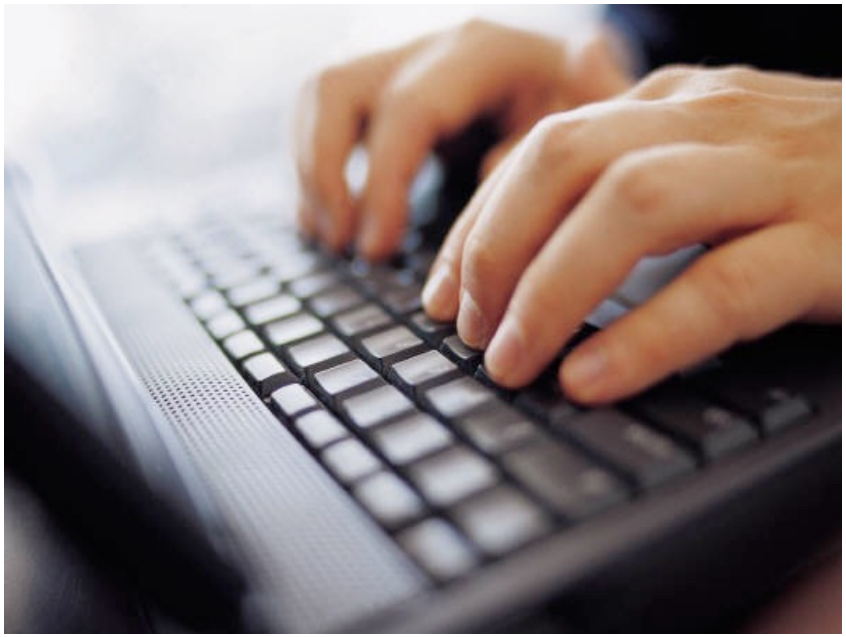


producer control flow pattern

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

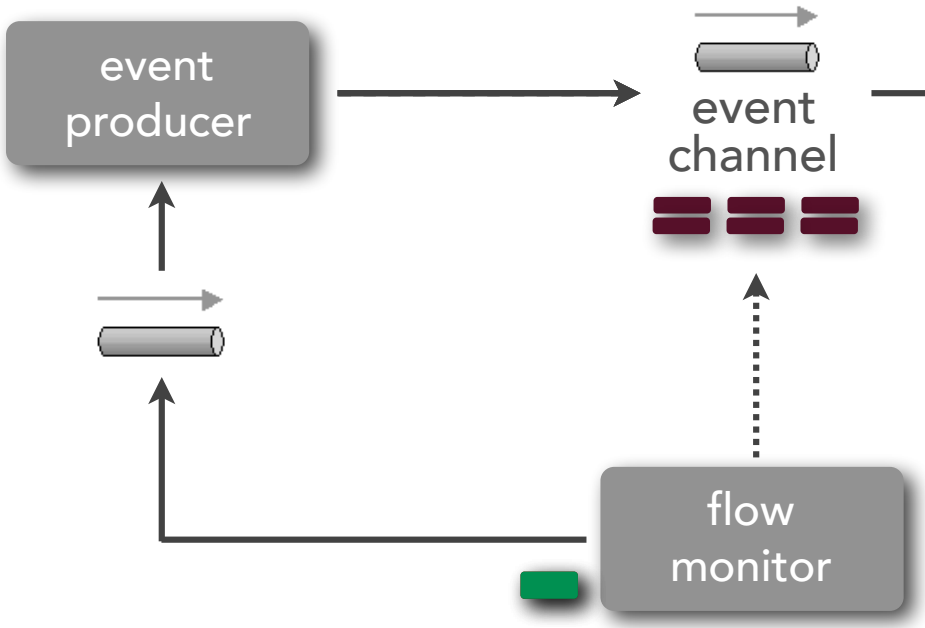
shutdown (broker) vs. slowdown (pattern)

producer control flow pattern



let’s see the issue….

producer control flow pattern



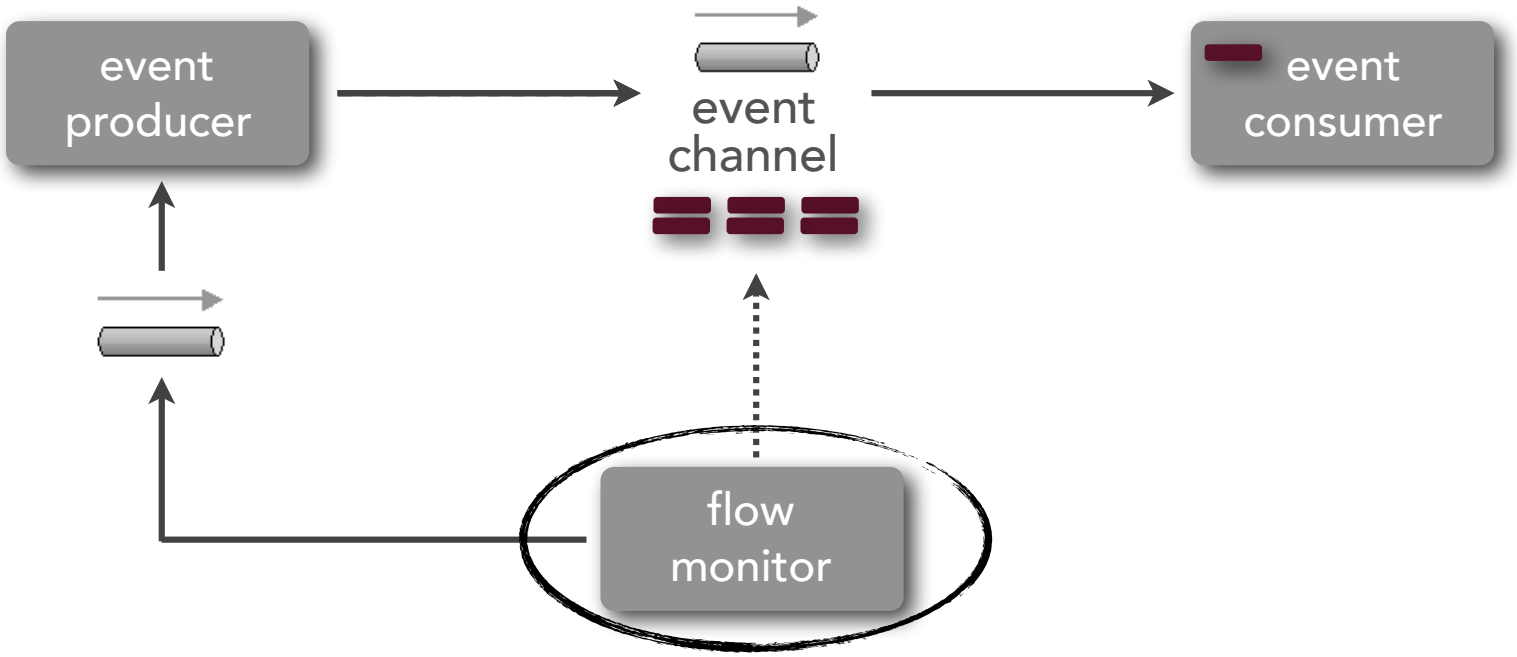
wait for upper threshold

tell producers to slow down

wait for lower threshold

tell producers to resume

producer control flow pattern



producer control flow pattern

flow monitor

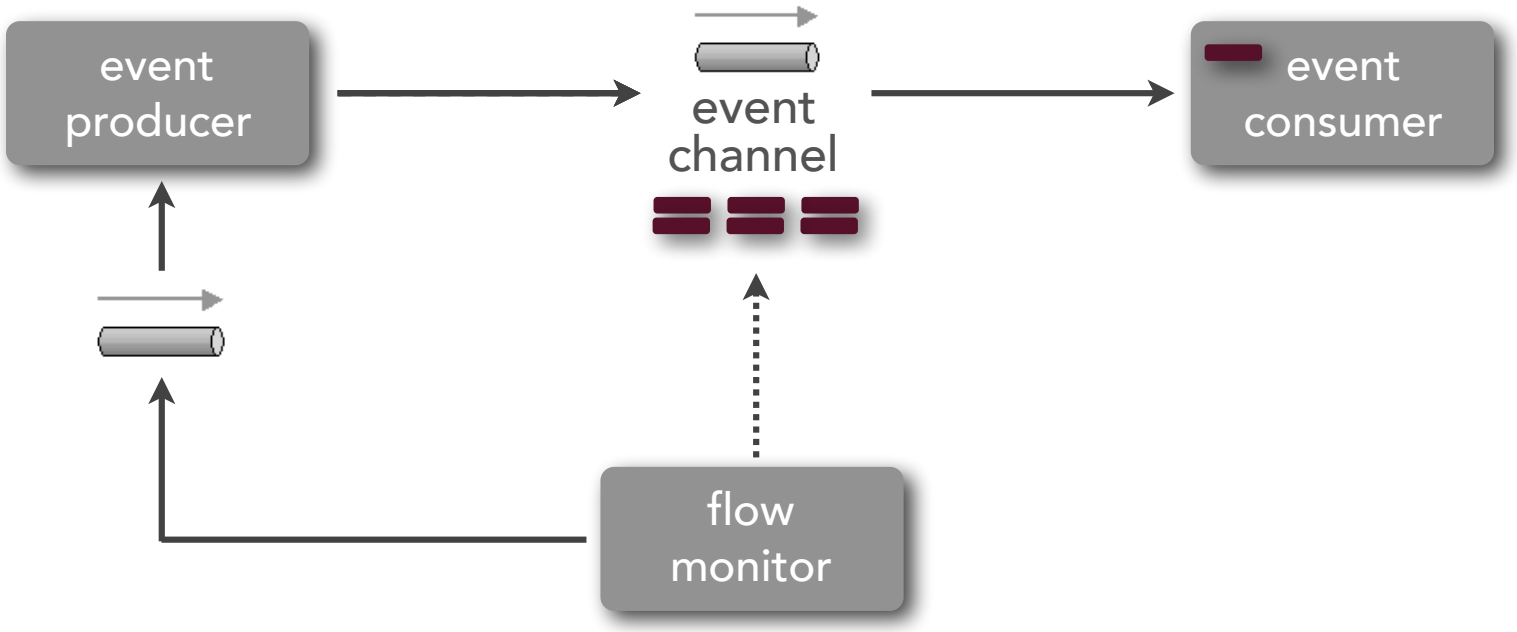
```
public void execute() throws Exception {
   //connect to message broker
   long threshold = 10;
   boolean controlFlow = false;
   while (true) {
      long queueDepth = getMessageCount("trade.eq.q");
      if (queueDepth > threshold && !controlFlow) {
         controlFlow = enableControlFlow(channel);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

producer control flow pattern

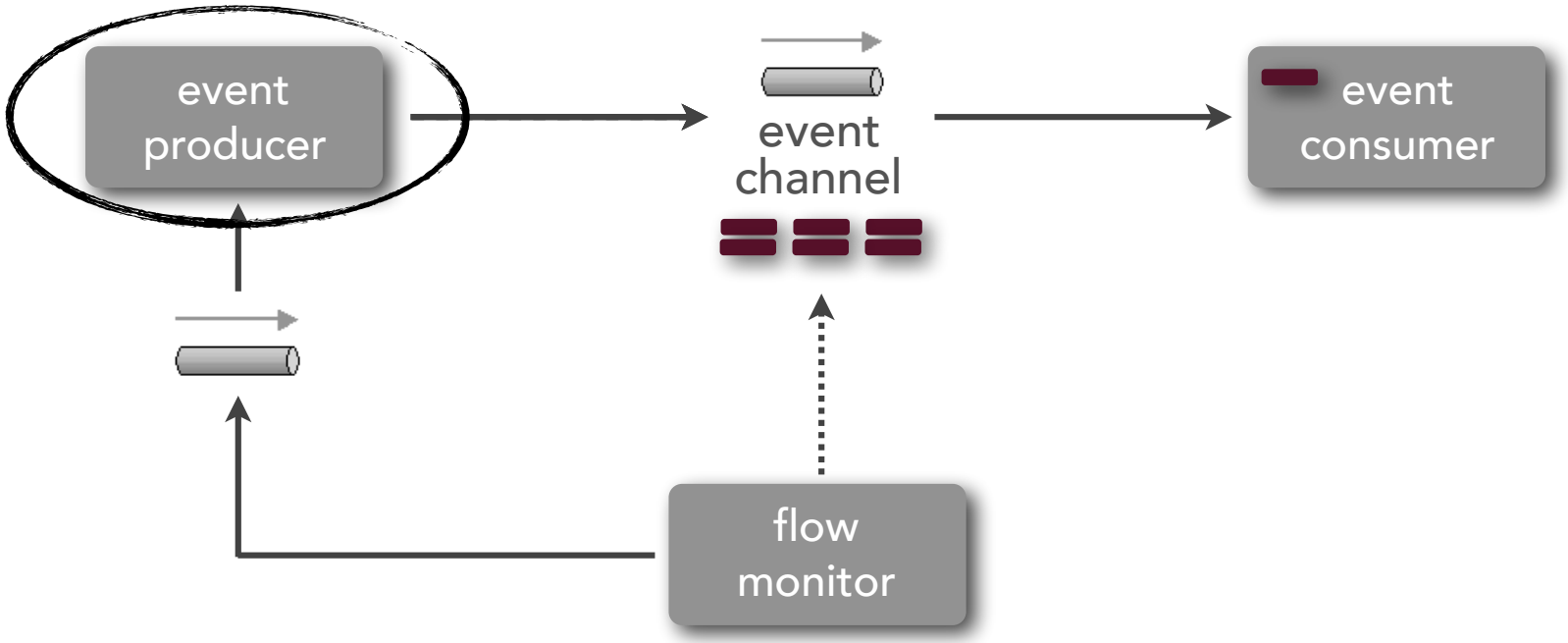
flow monitor

```
private boolean enableControlFlow(Channel channel) {
   Message msg = createMessage(delay, 3000);
   //send message to producer flow queue;
   return true;
}


private boolean disableControlFlow(Channel channel) {
   Message msg = createMessage(delay, 0);
   //send message to producer flow queue;
   return false;
}
```

producer control flow pattern



producer control flow pattern

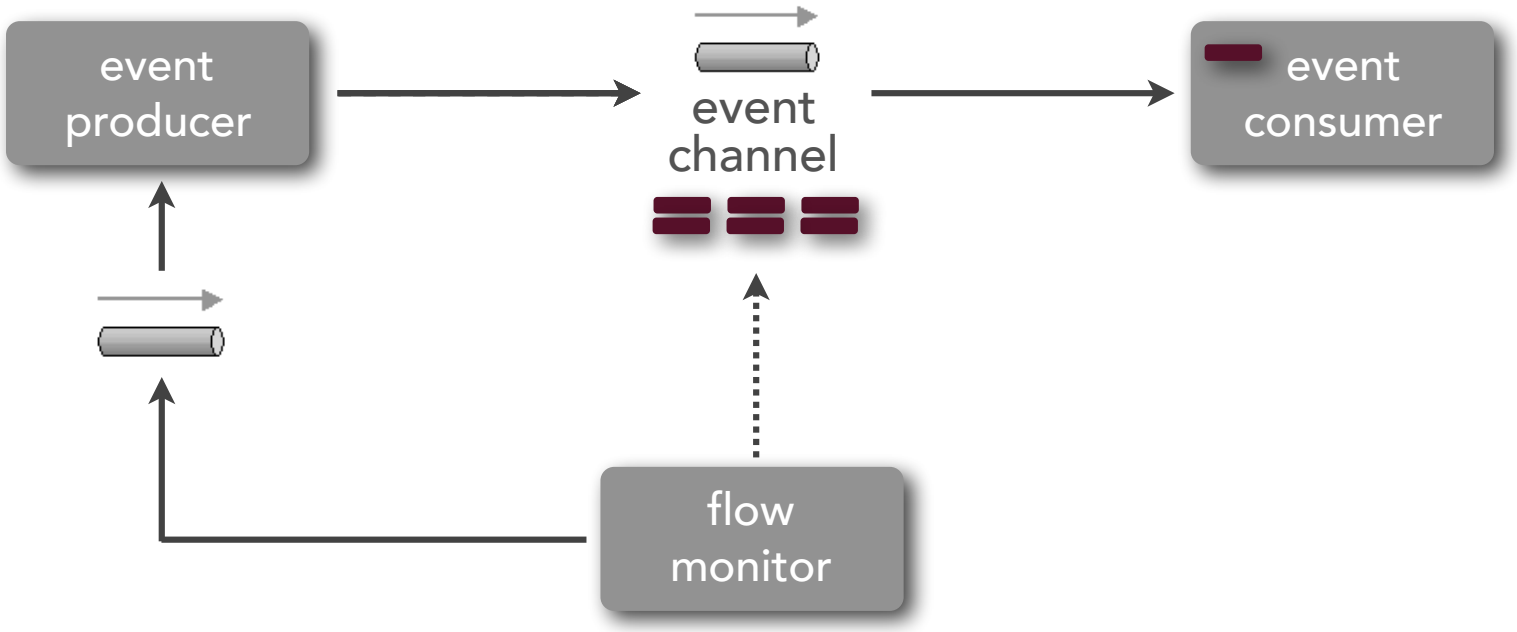


producer control flow pattern

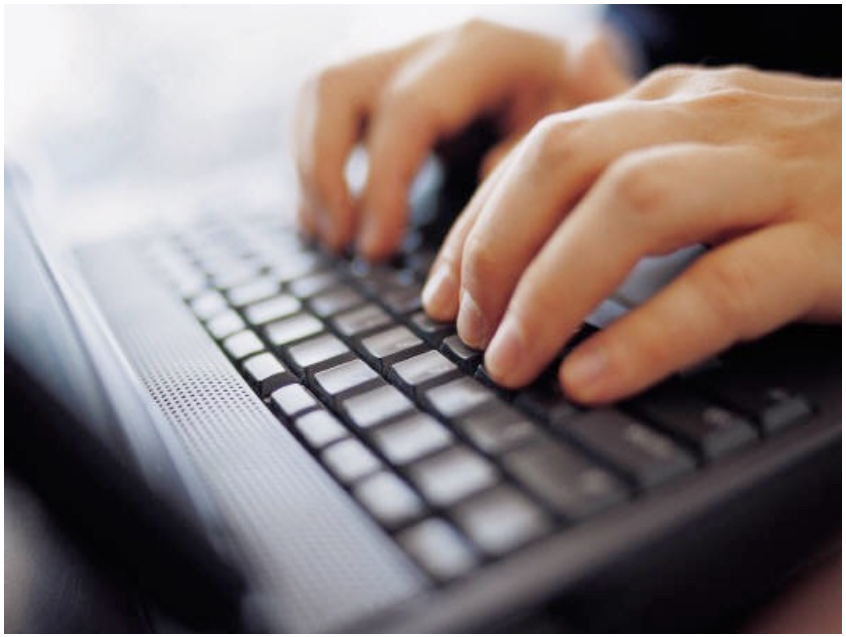
event producer

```
public void startListener() {
   new Thread() { public void run() {
      //connect to message broker and create consumer
      while (true) {
        msg = getNextMessageFromQueue();
        long delayValue = msg.getDelayValue();
        synchronized(delay) { delay = delayValue; }
      }
   }}.start();
}

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

producer control flow pattern



producer control flow pattern

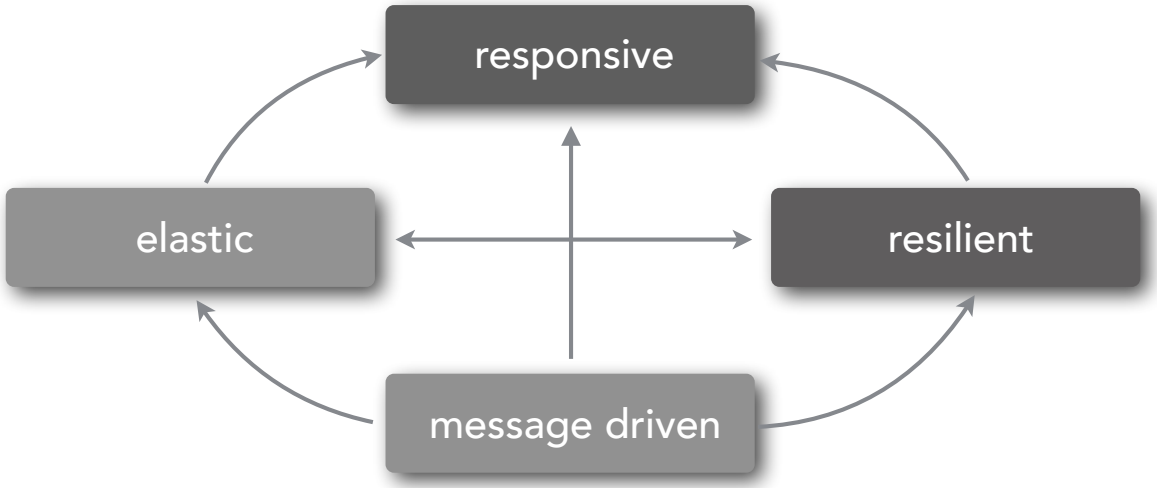


let’s see the result…

Threshold Adjust Pattern

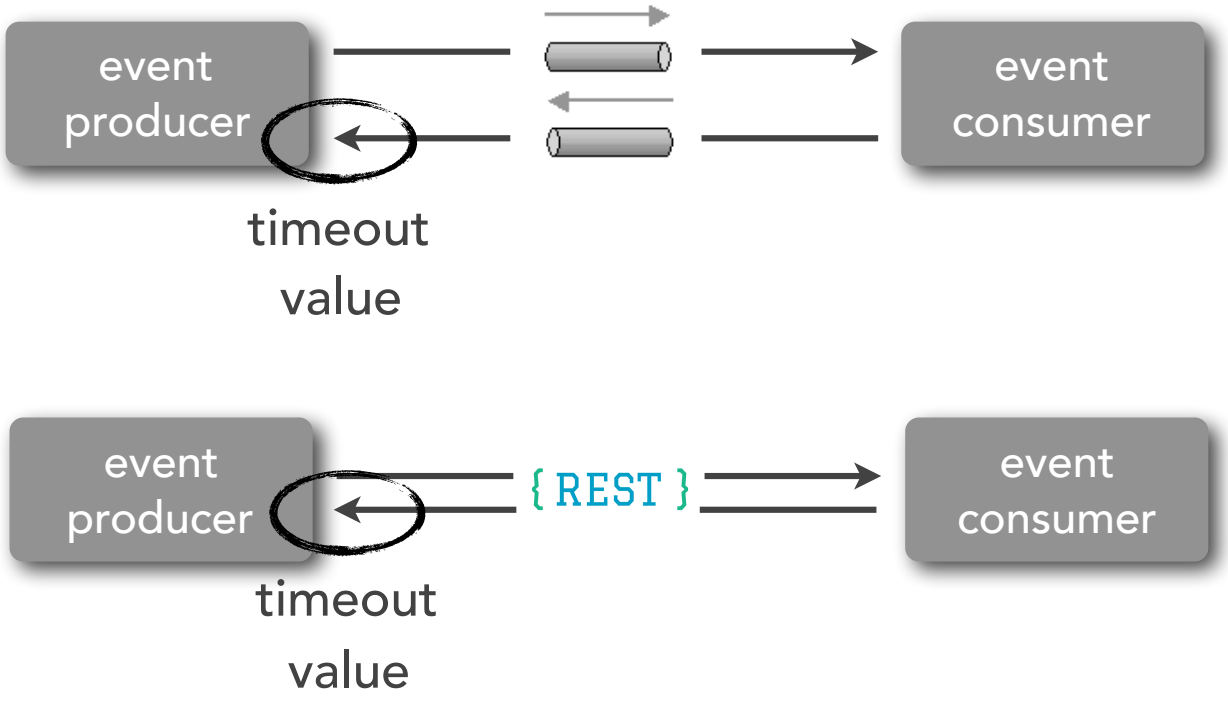
threshold adjust pattern

how can you automatically self-configure systems to manage changes in response times for remote calls?



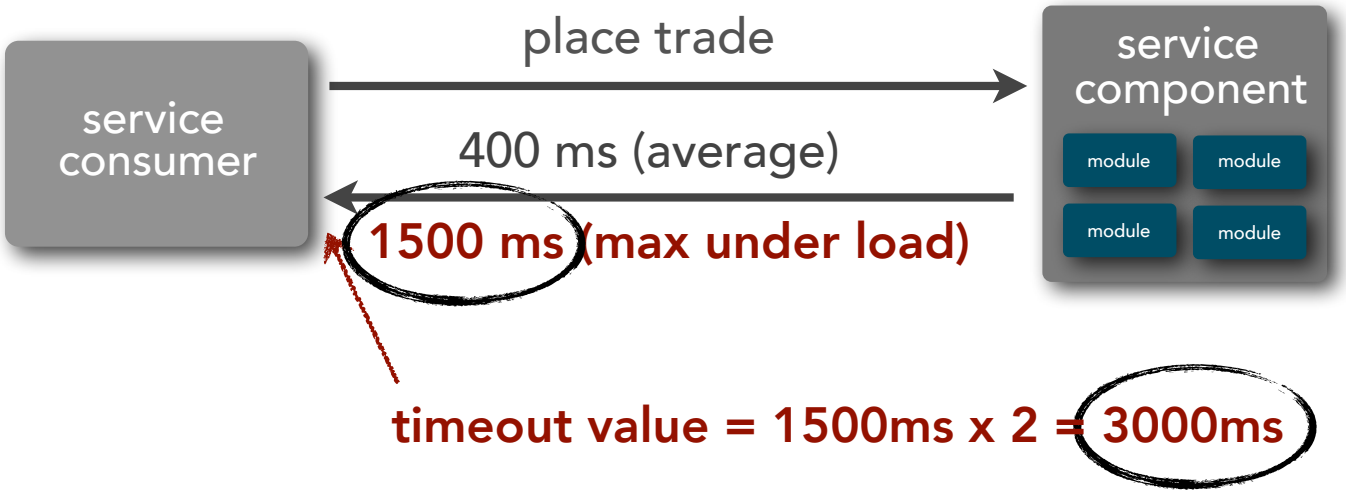
threshold adjust pattern

how can you automatically self-configure systems to manage changes in response times for remote calls?



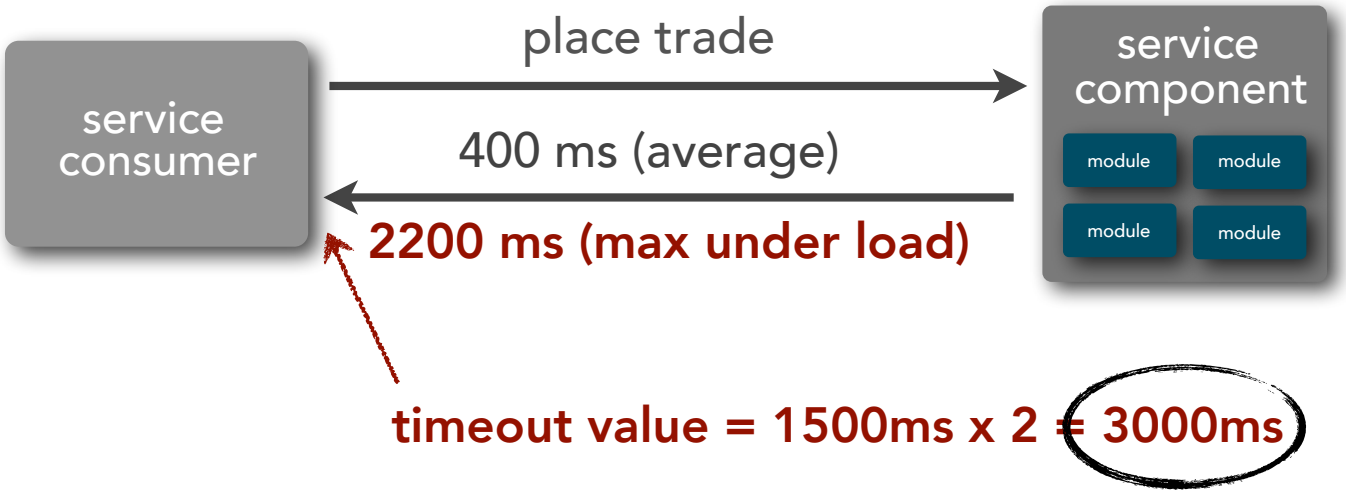
threshold adjust pattern

how can you automatically self-configure systems to manage changes in response times for remote calls?

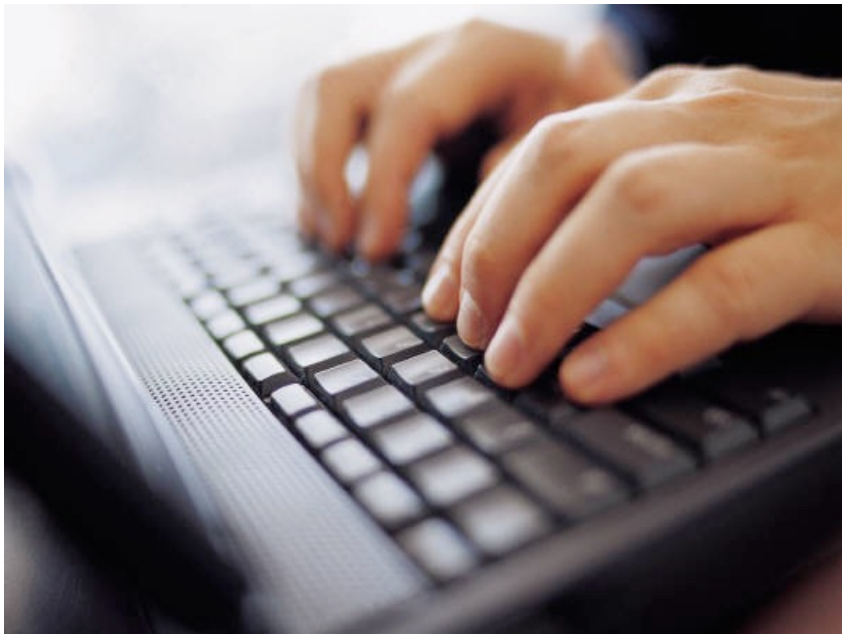


threshold adjust pattern

how can you automatically self-configure systems to manage changes in response times for remote calls?

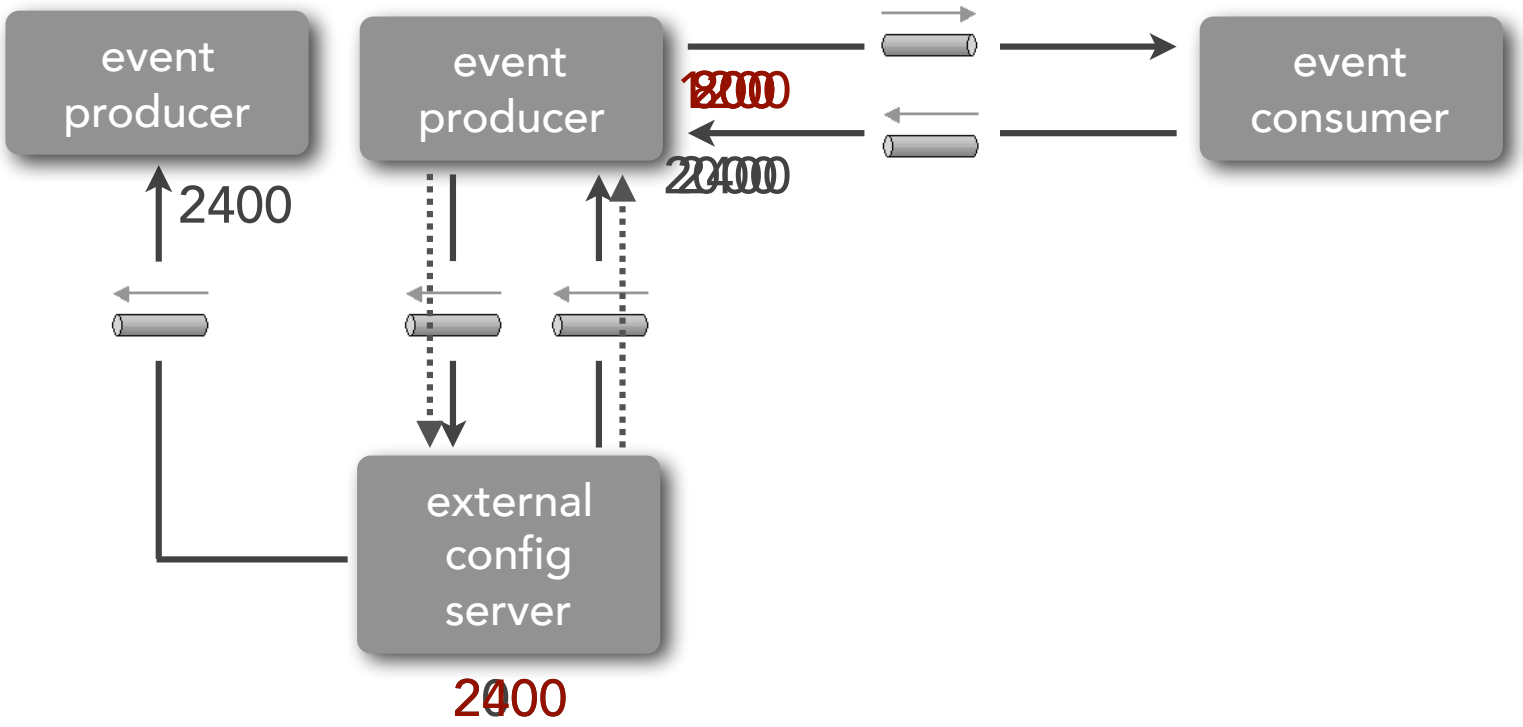


threshold adjust pattern

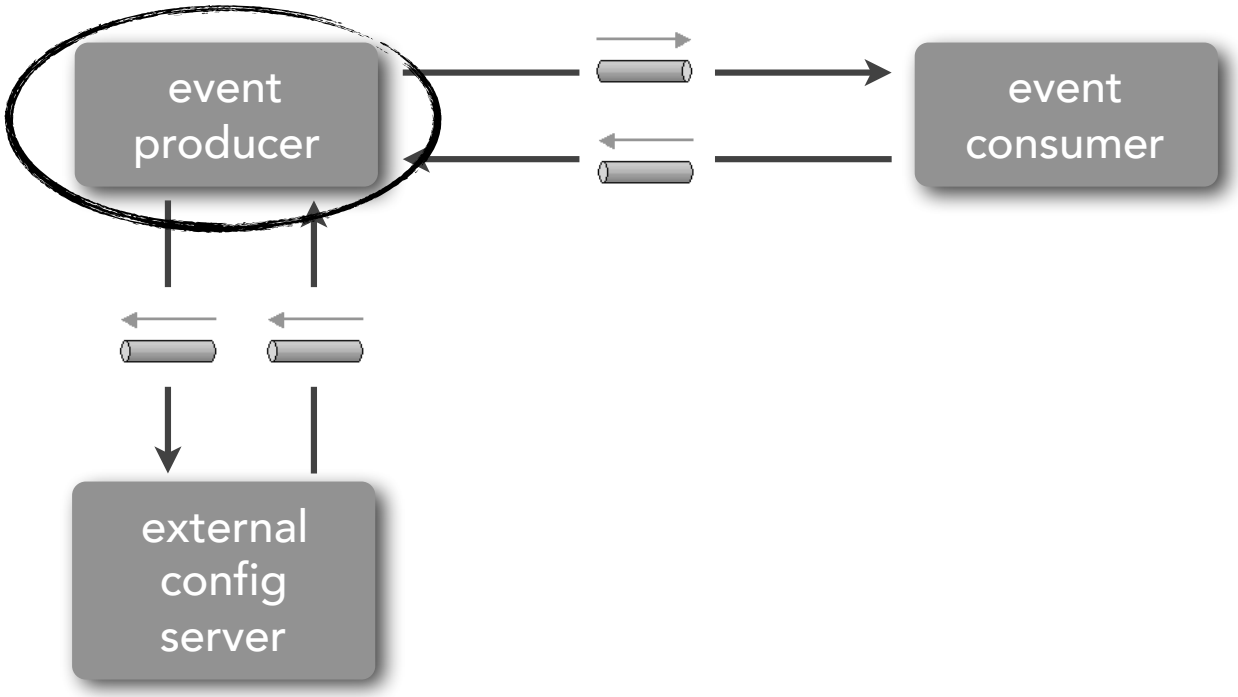


let’s see the issue…

threshold adjust pattern



threshold adjust pattern



threshold adjust pattern

event producer

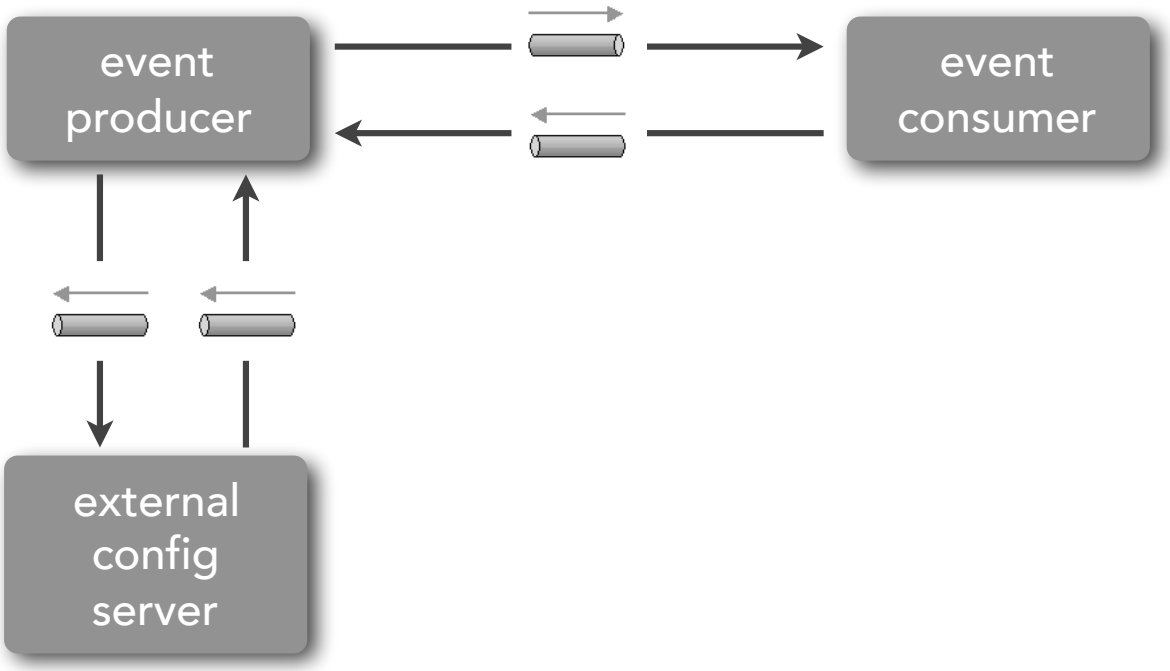
```
//STARTUP LOGIC
//connect to message broker
long threshold = getThresholdValue("place_trade");

//PROCESSING LOGIC
long start = System.currentTimeMillis();
String results = sendTradeRequest();
long end = System.currentTimeMillis();
long duration = end - start;
if ((duration*2) > threshold) {
   threshold = duration*2;
   updateThresholdValue("place_trade");
}
```

threshold adjust pattern

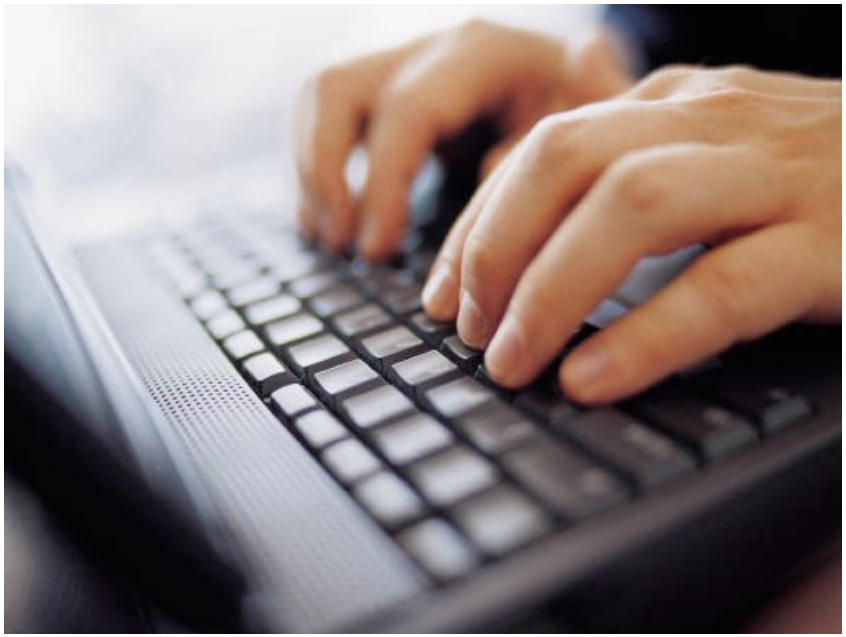
event producer

```
//SYNC LOGIC
//wait for notification message from config server
long threshold = getThresholdValue("place_trade");
```

threshold adjust pattern



threshold adjust pattern



let’s see the result…

Reactive Architecture Patterns

Mark Richards

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

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