Reactive Architecture Patterns 2

Mark Richards
Independent Consultant
Hands-on Software Architect / Published Author
Founder, DeveloperToArchitect.com
www.wmrichards.com

Author of *Software Architecture Fundamentals Video Series* (O’Reilly)
Author of Microservices Pitfalls and AntiPatterns (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

thread delegate pattern

producer control flow pattern
source code

https://github.com/wmr513/reactive
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
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

event delegate

event delegate

A-M

symbol striping

N-Z
thread delegate pattern

event dispatcher models

- Event producer
- Event dispatcher
- Thread delegate

- Single-use thread allocation
- Managed queued thread pool
thread delegate pattern

single-use thread allocation

event dispatcher

thread delegate
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
//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>
```

does this make it single-threaded to preserve message order
//hold 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 instance, count>
private Map<Object, Long> threadCount =
    new HashMap<Object, Long>();
//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...
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);
}
incrementThreadCount(thread);
thread.execute(new TradeProcessorThread(this, msg));
public void requestComplete(Object thread, String symbol) {
    long count = decrementThreadCount(thread);
    if (count == 0) {
        allocationMap.remove(symbol);
    }
}
thread delegate pattern

managed queued thread pool
thread delegate 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

Concurrent Consumers (zoomed in)
(500 Messages / 100ms Processing Time)
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?

slow down!
producer control flow pattern

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

stop (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

- event producer
- event channel
- event consumer
- 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);
    }
}
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

event producer

event channel

flow monitor

event consumer
producer control flow pattern
public void startListener() {
    new Thread(()->{
        //connect to message broker and create consumer
        while (true) {
            msg = getNextMessageFromQueue();
            long delayValue = msg.getDelayValue();
            synchronized(delay) { delay = delayValue; }
        }
    }).start();
}

private void placeTrade() {
    Thread.sleep(delay);
    //send trade to processing queue...
}
producer control flow pattern
producer control flow pattern

let’s see the result…
Reactive Architecture Patterns 2

Mark Richards
Independent Consultant
Hands-on Software Architect / Published Author
Founder, DeveloperToArchitect.com
www.wmrichards.com

Author of *Software Architecture Fundamentals Video Series* (O’Reilly)
Author of Microservices Pitfalls and AntiPatterns (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)
GREAT INDIAN DEVELOPER SUMMIT 2019
Conference: April 23-26, Bangalore

Register early and get the best discounts!

www.developersummit.com @greatindiandev bit.ly/gidslinkedin