An introduction to reactive applications, Reactive Streams, and options for the JVM

Steve Pember
CTO, ThirdChannel

Software Architecture Con, 2016
“Reactive Streams”, “Reactive Extensions”, or “Rx”
Agenda

• The Problem
• What are Reactive Streams?
• Rx in depth
• An Overview of JVM options
• Demo Time!
The Problem: The Need to go Reactive
Really, it’s Two problems
1) Performance Demands Are Always Increasing
We Use Technology from the Beginning of Web Development
Things Slow Down
Users get angry quickly.
<table>
<thead>
<tr>
<th>Delay</th>
<th>User reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 100 ms</td>
<td>Instant</td>
</tr>
<tr>
<td>100 - 300 ms</td>
<td>Slight perceptible delay</td>
</tr>
<tr>
<td>300 - 1000 ms</td>
<td>Task focus, perceptible delay</td>
</tr>
<tr>
<td>1 s+</td>
<td>Mental context switch</td>
</tr>
<tr>
<td>10 s+</td>
<td>I'll come back later...</td>
</tr>
</tbody>
</table>

1000 ms time to glass challenge

- Simple user-input must be acknowledged within ~100 milliseconds.
- To keep the user engaged, the task must complete within 1000 milliseconds.

_Ergo, our pages should render within 1000 milliseconds._

*Speed, performance and human perception*
Let’s Keep Our Users Happy And Engaged
2) The Rise of Microservices
Multiple Integration Points
It’s Not Only Users That Use Up Resources
So what to do?
Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today’s demands are simply not met by yesterday’s software architectures.
Reactive Applications

- Responsive
Reactive Applications

- Responsive
- Resilient
Embrace Failure
Independent Things Fail Independently
Reactive Applications

• Responsive
• Resilient
• Elastic (Scalable)
Reactive Applications

- Responsive
- Resilient
- Elastic (Scalable)
- Asynchronous / Message Driven
Free up resources with Async Operations & Non-Blocking I/O
Async is Hard for Humans
One Excellent Tool is (are?) Reactive Streams
Agenda

- The Problem
- What are Reactive Streams?
Collections + Time
Zip

combine the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function

1  2  3  4  5
A  B  C  D
1A 2B 3C 4D
Single abstraction over data from many sources
Observer Pattern

Push (not Pull) based Iterators
Stream-Based Functional Programming
Imperative vs Stream

1 // Iterative
2 List numbers = 1..100
3 int max = numbers.size()
4 Map result = [count:0, sum: 0]
5 for (int i = 0; i < max; i++) {
6     // only work with even numbers
7     if (numbers[i] % 2 == 0) {
8         result.count++
9         result.sum += numbers[i]
10     }
11 }
12 println "Results were: \{result\}"

13 // Reactive Stream
14 rx.Observable.from(1..100)
15 .filter({int num -> num % 2 == 0}) // filter out odd
16 .reduce([count:0, sum: 0], {Map data, int num->
17         data.count++
18         data.sum += num
19         data
20 })
21 .subscribe({println "Results were \{it\}"})
Streams with Extensions for Reactive Programming
Rx makes Async behavior easy!
(Reactive Pull) Backpressure
What is Rx?

- Collections + Time
- A Single Abstraction over data from different sources
- Observer Pattern with Push-based iterators
- Stream Based Functional Programming
- ... with Extensions for Reactive Programming
- Async is easy
- Backpressure
Rx Simplifies Complex Work
...Once you understand, of course...
Agenda

• The Problem
• What are Reactive Streams?
• Rx in depth
Key Terms:
An Observable is like Promise ++
An Observable pushes items to Subscribers
Subscribers receive and operate on emitted data
Observables and Subscribers operate on a Scheduler
The following examples use rxJava
But, try out rxGroovy
Groovy

- Dynamic Language for the JVM
- Less Verbose (Reduce Java Boilerplate)
- Ruby/Python-esque Collections
- Run Time Meta Programming
- Optionally Typed
- AST Transformations
- Powerful Annotations
- Multi-Inheritance via Traits and @DelegatesTo
rx.Observable.from(1..1000)
    .filter(new Func1<Integer, Boolean>(){
        @Override
        Boolean call(Integer integer) {
            return integer % 2 == 0
        }
    })
    .map(new Func1<Integer, Integer>() {
        @Override
        Integer call(Integer integer) {
            integer * integer
        }
    })
    .subscribe(new Action1<Integer>(){
        @Override
        void call(Integer integer) {
            System.out.println("Have even square: "+ integer)
        }
    });
rx.Observable.from(1..1000)
  .filter({Integer integer -> integer % 2 == 0})
  .map({Integer integer -> integer * integer})
  .subscribe({
    println ("Have even square: " + it)
  })
Basic Usage
Observable<Integer> numbers = Observable.create({subscriber ->
    try {
        List numbers = 1..100
        numbers.each {Integer num->
            subscriber.onNext(num)
            // once subscriber has finished with one number,
            // it asks for more
        }
        // after sending all values we complete the sequence
        if (!subscriber.unsubscribed) {
            subscriber.onCompleted()
        }
    }
    catch(Exception e) { //catch all. Gross, I know
        if (!subscriber.isUnsubscribed()){
            subscriber.onError(e)
        }
    }
} as Observable.OnSubscribe) // so IntelliJ is happy
```java
new rx.Subscriber<Integer>() {
    @Override
    void onComplete() {
        log.error("All done!")
    }

    @Override
    void onError(Thrower e) {
        log.error("Ooops!", e)
    }

    @Override
    void onNext(Integer integer) {
        // do some work on the incoming Integer
        log.info("Received integer 
            // after work is complete, call:
            request(1)
            // this value doesn't have to be 1. I can ask for varying
            // number items based on current resources
    }

    @Override
    void onStart() {
        // ask the Observer to send 1 item to begin with
        request(1)
    }
}
```
Thankfully, there are shortcuts
```java
rx.Observable.from(1..10)
.subscribe { Integer i ->
    // this closure is the 'onNext' action
    println i*i
}
```
```java
rx.Observable.from(1..10)
.subscribe({ Integer i ->
    // first argument is the 'onNext' action
    println i*i
},
    { // 2nd argument is the 'onError' action, receives Throwable as argument
        log.error("An error occurred: ", it)
},
    {
        // 3rd argument is the 'onCompleted' action
        println "All done!"
    }
)
Streams are Composable
```python
def stream = rx.Observable.from(1..10)

def mapAndFilter = stream
  .map({ it * it })
  .filter({it % 2 == 0})

def andReduce = mapAndFilter
  .reduce(0, {int sum, int item ->
    sum += item
  })

andReduce.subscribe({println it})
```
You can get much power from 5 functions

- filter
- map
- reduce
- groupBy
- flatMap
rx.Observable.from(1..100)
// filter requires some sort of truth statement
.filter({Integer num -> num % 2 == 0})
.subscribe({
    println "My next even number is ${it}"
})
rx.Observable.from(1..100)
// map allows you to perform some
// transformational operation. You
// can even change types
.map({Integer num ->
        "My next square is ${num * num}"
    })
.subscribe({ String square->
            println square
    })
rx.Observable.from(1..100)
// note: you can reduce into anything you want
// number, list, map, custom object, etc.
.reduce(0, {int sum, int num -> sum += num})
.subscribe {println it}
class Book {
    int yearPublished
    String genre
    String title
}

rx.Observable.from(bookList)
  .groupBy({it.yearPublished})
  // at this point we actually have a
  // stream of Observables
  .subscribe({GroupedObservable go -> println go})
First Mental Leap: An Observable of Observables
Observable start

stream actions (e.g. map, filter)

groupBy

actions on new stream

actions on new stream

actions on new stream

actions on new stream

Merge
```java
rxjs.Observable.from(bookList)
  .groupBy({it.yearPublished})
  .flatMap({yearGroup ->
    // sum up all books in this year stream
    // create String of details
    yearGroup
    .reduce(0, {sum, book -> sum += 1})
    .map({Integer sum -> "The number of books" +
          "from ${yearGroup.key} was ${sum}"})
  })
  .subscribe({println it })
```
```typescript
rxjs.Observable.from(bookList) // there's only 5
  .groupBy({it.yearPublished}) // group by year
  .flatMap({yearGroup ->
    yearGroup
      .groupBy({it.genre}) // then group by genre
      .flatMap({genreGroup ->
        genreGroup
          .reduce(0, {sum, book -> sum += 1}) // then count
          .map({Integer sum -> [(genreGroup.key):sum] })
      }) // now we a stream of [genre:sum]
      .reduce([], {Map genres, Map row -> genres += row})
      .map({[(yearGroup.key): it]})
  }) // now we have a stream of year Breakdowns
  .reduce([], {Map years, Map row -> years += row})
//output ex: [2016:[Sci-Fi:1, Drama:2], 2015:[Horror:1, Mystery:1]]
 .subscribe({println it })
```
Hot vs Cold
Cold Observable: finite data, on demand

Hot Observable: infinite data, as it’s ready
Cold Observable: only starts emitting data on `.subscribe()`

Hot Observable: emits data whenever it’s ready
Asynchronous Streams
rx.Observable.from(1..100)
.reduce(0, {int sum, int num -> sum += num})
// 'observeOn' controls how your
// Observer / Subscriber executes
.observeOn(Schedulers.computation())
.subscribe {println it}
rx.Observable.from(bookList)
// 'subscribeOn' controls how your
// Observable executes.
// Seems Backward, I know
.subscribeOn(Schedulers.io())
.subscribe({println "Received \${it}"})
rxjs.Observable.from(bookList)
.subscribeOn(Schedulers.io())
.map({"I have book: ${it.title}"})
.observeOn(Schedulers.computation())
.subscribe({println "Received ${it}"})
BackPressure
Observable

Give me 10
You sure? Here's 10
Give me 1
Ok, 1
Give me 5
Oh, there's only 3 left
onComplete()

Subscriber
Can only Mitigate Hot Streams

- throttle
- sample
- window
- buffer
- drop
Stream Interaction
Don’t Unsubscribe from Observables

Programmatically complete them when another Observable fires
AutoComplete Requirements

• Wait 250 ms between keypresses before querying
• If no keys are pressed, no query
• Successful queries should render movies
• Any new queries should kill in-flight queries
var searchResultsSets = keyPresses
  .throttle(250)
  .map(key =>
      getJSON("/searchResults?q=" + input.value)
        .retry(3)
        .takeUntil(keyPresses)
  )
  .concatAll(); // flatten or flatmap in other implementations

searchResultsSets.forEach(
    resultSet => updateSearchResults(resultSet),
    error => showMessage("The server does not know about your inane movie choice")
)
Pretty
Great,
But what’s going on?
The keyPress stream is reacting to itself
Questions?
Agenda

• The Problem
• What are Reactive Streams?
• Rx In Depth
• An Overview of JVM options
Story Time
Story Time

Microsoft → GitHub

@svpember
Story Time

Microsoft

GitHub

@svpember
Story Time

Microsoft

2012 - MS Open Source’s RX!

GitHub

@svpember
Story Time

2012 - MS Open Source’s RX!
Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols.

The Problem

Handling streams of data—especially “live” data whose volume is not predetermined—requires special care in an asynchronous system. The most prominent issue is that resource consumption needs to be controlled such that a fast data source does not overwhelm the stream destination. Asynchrony is needed in order to enable the parallel use of computing resources, on collaborating network hosts or multiple CPU cores within a single machine.

The main goal of Reactive Streams is to govern the exchange of stream data across an asynchronous boundary—think passing elements on to another thread or thread-pool—while ensuring that the receiving side is not forced to buffer arbitrary
Implementations

We are also proud to let Reactive Streams 1.0.0 be announced to the world accompanied with a multitude of compliant implementations verified by the TCK for 1.0.0, listed below in alphabetical order:

- **Akka Streams (version 1.0-RC2)**
  - See this [Activator template](#) and the [documentation](#).
- **MongoDB (version 1.0.0)**
  - For the documentation see [here](#).
- **Ratpack (version 0.9.16)**
  - See the “Streams” chapter of the manual.
- **Reactive Rabbit (version 1.0.0)**
  - Driver for RabbitMQ/AMQP, see [here](#).
- **Reactor (version 2.0.1.RELEASE)**
  - For the documentation see [here](#).
- **RxJava (version 1.0.0)**
  - See [github.com/ReactiveX/RxJavaReactiveStreams](#).
- **Slick (version 3.0.0)**
  - See the “Streaming” section of the manual.
- **Vert.x 3.0 (version milestone-5a)**
  - Vert.x 3.0 is currently in beta. The Reactive Streams implementation can be found [here](#).
We should probably start with RxJava
Brings Reactive Streams to the JVM
But Don’t Forget rxGroovy
The **Join** operator combines the items emitted by two Observables, and selects which items to combine based on duration-windows that you define on a per-item basis. You implement these windows as Observables whose lifespans begin with each item emitted by either Observable. When such a window-defining Observable either emits an item or completes, the window for the item it is associated with closes. So long as an item’s window is open, it will combine with any item emitted by the other Observable. You define the function by which the items combine.
Take a Look at Ratpack

- High performance web framework
- Non-opinionated
- Non-Blocking Network Stack
- Built on Reactive Streams, Netty, Java 8, Guice
- Fully embodies reactive
- Light-weight, self-contained deployables
Ratpack Processing Model

- How Ratpack works (4 CPU example)

One thread per CPU core

Return thread to request-taking pool while we wait

Waiting for something? (db call, call to remote)

Includes rxRatpack module, but we’ll talk about that later
Akka & Akka Streams

- Library
- Definition of Reactive System
- Created by LightBend
- Actor-Based Concurrency
- Implemented Streams on Top of Actor Model
@Slf4j
@CompileStatic
class Drinker extends UntypedActor {
    @Override
    void onReceive(Object o) throws Exception {
        if (o instanceof FullPint) {
            onReceivePint((FullPint)o)
        } else {
            log.info("What is this ${o.class} you're trying to give me?")
        }
    }

    void onReceivePint(FullPint pint) {
        log.info("${self.path().name()}: Received pint ${pint.number}. Drinking it down")
        // drinking time
        Thread.sleep(1000)
        log.info("${self.path().name()}: That was delicious! Sending empty pint ${pint.getNumber()} back")
        sender().tell(new EmptyPint(pint.getNumber()), self)
    }
}
The message flow travels across actor systems on different servers.

An external event enters an actor system.

Some actors are involved in handling the event, based on how the event should be handled and what the behavior is of the actors. In this case actors the chain of a, b, c, d, z, y, d, q, x, p, s creates a result.

Yes, it can get complex quickly!

Some result refers to the event cause.
• Library
• Reactive Streams
• Reactor Pattern
• Built on LMAX Ring Buffer / Disrupter
• Multiple libraries to extend Ring Buffer in multiple ways
receiver is currently writing to slot 31. It keeps advancing forward, just checking it doesn't wrap past the business logic consumer.

journaler and replicator are both reading slot 24. They can keep reading forwards up to slot 30.

un-marshaller went off for a tea break, so is back to 15. It may go as far as 30.

business logic consumer cannot go past any of the other three, so must wait at slot 14 until the un-marshaller gets back from its tea break.
This is the timeline of the Mono. Time flows from left to right.

This is the eventual item emitted by the Mono.

This vertical line indicates that the Mono has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Mono. The text inside the box shows the nature of the transformation.

This Mono is the result of the transformation.

If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X.
These are items emitted by the Flux.

This vertical line indicates that the Flux has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Flux. The text inside the box shows the nature of the transformation.

This Flux is the result of the transformation.

If for some reason the Flux terminates abnormally, with an error, the vertical line is replaced by an X.
Many Tools Seem to be Moving Towards Reactive Streams
Demo Time
Any Questions?
Thank You!

@svpember
spember@gmail.com
More Information

- Reactive Groovy & Ratpack Demo: https://github.com/spember/reactive-movie-demo
- Jafar Husain: RxJS: https://www.youtube.com/watch?v=XRYN2xt11Ek
- Reactive Streams Spec: http://www.reactive-streams.org/
- Reactive Manifesto: http://www.reactivemanifesto.org/
- Akka: http://akka.io/
- rxJava / ReactiveX libraries: https://github.com/ReactiveX
- Ratpack: http://ratpack.io/
- Reactor: https://github.com/reactor/reactor
- The Introduction to Reactive Programming you've been missing: https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- Martin Fowler: Stream / Pipeline programming: http://martinfowler.com/articles/refactoring-pipelines.html
- Or Just on Groovy (Groovy the Awesome Parts): http://www.slideshare.net/SpringCentral/groovy-the-awesome-parts
- Advanced RxJava Blog: http://akarnokd.blogspot.com/
Images

• Empty Pool: http://www.wtok.com/home/headlines/Water-Problems-205987121.html
• Juggling: https://en.wikipedia.org/wiki/Juggling
• Directing Traffic: https://www.flickr.com/photos/tracilawson/3474012583L
• LMAX Disrupter: http://martinfowler.com/articles/lmax.html
• Mailman: thebrandtstandard.com/2013/02/09/u-s-post-office-to-end-saturday-letter-delivery-this-summer/
• Actors Diagram: https://blog.codecentric.de/en/2015/08/introduction-to-akka-actors/
• Cheetah: www.livescience.com/21944-usain-bolt-vs-cheetah-animal-olympics.html
• Dominoes: https://www.flickr.com/photos/louish/5611657857/sizes/l/in/photostream/
• Spartans: www.300themovie.com/
• Stampeding Buffalo: news.sd.gov/newsitem.aspx?id=15164