The Road to Reactive with RxJava

Or: How to use non blocking I/O without wanting to kill yourself
Legacy

“An accomplishment that remains relevant long after a person has died”
Software is not so lucky
Netscape® Navigator 4.04

Copyright © 1994-1997 Netscape Communications Corporation, All rights reserved.

This software is subject to the license agreement set forth in the license. Please read and agree to all terms before using this software.

Netscape and Netscape Navigator are registered trademarks of Netscape Communications Corporation in the United States and other countries. Netscape's logos and Netscape product and service names are also trademarks of Netscape Communications Corporation, which may be registered in other countries. Other product and brand names are trademarks of their respective owners.
Why don’t we love old code?
Is this comparison fair?
Get to the point, please?
Throwing away old code is so tempting…
Replacing legacy systems is really hard
Love your code like your home town
Frank Lyaruu

CTO at Dexels, building enterprise software for sports federations.
Sportlink 2016
Blocking code
A servlet story
Blocking code

String response = callMethod();
doSomethingElse(response);
...

WELL, LIKE DUH.
Blocking code

- Easy to compose using the ‘;’ operator ;-) 
- Easy to reason about 
- Pleasant to debug 
- Maps nicely to low level instructions 
- It’s efficient* 
- It’s *extremely* mature 
- It’s familiar
Blocking code looks pretty good

Why mess with it?
Blocking code

- It’s only efficient when it is *doing* something
- Not when it is waiting for something
- At scale, threads are expensive
public Result getSomeRemoteData(String argument) {
    return HttpClient.callRemoteURL(argument);
}
<table>
<thead>
<tr>
<th>Blocking: kayak</th>
<th>Blocking: mouse, beer, gnome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time: 619.2ms</td>
<td>Total Time: 1523.7ms</td>
</tr>
<tr>
<td>Thread held (red): 619.2ms</td>
<td>Thread held (red): 1523.7ms</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Asynchronous: kayak</th>
<th>Asynchronous: mouse, beer, gnome</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time: 471.0ms</td>
<td>Total Time: 536.0ms</td>
</tr>
<tr>
<td>Thread held (red): 0.6ms</td>
<td>Thread held (red): 0.9ms</td>
</tr>
<tr>
<td>(0.3 initial + 0.3 generate)</td>
<td>(0.6 initial + 0.3 generate)</td>
</tr>
<tr>
<td>Async wait (green): 470.5ms</td>
<td>Async wait (green): 535.2ms</td>
</tr>
</tbody>
</table>

See: https://webtide.com/async-rest/
Microservices

- Remote calls to other parts of the application are *everywhere*
- A remote call might be calling *another* remote call
- One operation might be blocking more than *one* thread
Blocking code: Stability

- Slow network connections are very expensive
- Performance problems cause an explosion of threads….
- … which cause performance problems
- Blocking systems don’t degrade gracefully
Blocking code

- Expensive at scale
- A liability for distributed systems
Non blocking code

A non-blocking method will return before it is truly ‘finished’

So when it returns without an error, that means *nothing*
Non blocking code

• When is it finished?

• How do I know if it succeeded?

• If it did, where is my result?

• How often can I call it again?
Callbacks

```java
public void performAsyncMethod(String argument, Callback callback);
performAsyncMethod("arg", result->{... do stuff ...});
```
public void executeAsync() {
    performAsyncMethod("arg", result->
        performOperation(result, r->
            performAnotherOperation(r, res->{})
        )
    );
}
Servlet 3.1 Non-blocking I/O
public class BlockingServlet extends HttpServlet {
    private static final int BUFFER_SIZE = 1024;

    protected void service(HttpServletRequest request, HttpServletResponse response) {
        byte[] buffer = new byte[BUFFER_SIZE];
        while (true) {
            int read = request.getInputStream().read(buffer, 0, BUFFER_SIZE);
            if (read < 0)
                break;
            response.getOutputStream().write(buffer, 0, read);
        }
    }
}
A non blocking Echo Servlet

```java
public class NonBlockingServlet extends HttpServlet {
    private static final int BUFFER_SIZE = 1024;

    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        AsyncContext asyncContext = request.startAsync(request, response);
        asyncContext.setTimeout(0);
        Echoer echoer = new Echoer(asyncContext);
        request.getInputStream().setReadListener(echoer);
        response.getOutputStream().setWriteListener(echoer);
    }

    private class Echoer implements ReadListener, WriteListener {
        private final byte[] buffer = new byte[BUFFER_SIZE];
        private final AsyncContext asyncContext;
        private final ServletInputStream input;
        private final ServletOutputStream output;
        private boolean complete;

        private Echoer(AsyncContext asyncContext) throws IOException {
            this.asyncContext = asyncContext;
            this.input = asyncContext.getRequest().getInputStream();
            this.output = asyncContext.getResponse().getOutputStream();
        }

        @Override
        public void onDataAvailable() throws IOException {
            while (input.isReady()) {
                int read = input.read(buffer);
                output.write(buffer, 0, read);
                if (!output.isReady())
                    return;
            }
            if (input.isFinished())
                complete = true;
            asyncContext.complete();
        }

        @Override
        public void onAllDataRead() throws IOException {
```

```java
```
```java
    @Override
    public void onWritePossible() throws IOException {
        if (input.isFinished())
            if (!complete)
                asyncContext.complete();
        else
            onDataAvailable();
    }

    @Override
    public void onError(Throwable failure) {
        failure.printStackTrace();
    }
}
```
A simpler example

protected void doGet(HttpServletRequest req, HttpServletResponse resp) {
    AsyncContext asyncContext = request.startAsync(request, response);
    final ServletOutputStream out = resp.getOutputStream();
    final byte[] buf = new byte[1024];
    final FileInputStream file = …;
    out.setWriteListener(new WriteListener() {
        @Override
        public void onWritePossible() throws IOException {
            while(out.isReady()) {
                int len = file.read(buf);
                if(len<0) {
                    return;
                }
                out.write(buf, 0, len);
            }
        }
    });
}
THIS CAN'T BE RIGHT
Blocking code & I/O
don’t play nice
Why?

Threads don’t react, they DO

I/O is reactive by nature
Node.js

```javascript
var server = http.createServer(function (req, res) {
    req.pipe(res);
});
```
Reactive Programming

“Programming pipes”
ReactiveX

Polyglot API for reactive streams

RxJava RxJS rx.Net … RxPHP RxSwift
The world view of RxJava

Source: Observable

Destination: Subscriber
RxJava at High speed

```java
Observable.<String>just("Ouagadougou","Dakar","Accra","Rabat")
    .subscribe(System.err::println);
```

Ouagadougou
Dakar
Accra
Rabat
RxJava at High speed

```java
Observable.range(0, 1000)
    .subscribe(System.err::println);
```

0
1
2
3
4
5
6
...
998
999
RxJava at High speed

```java
Observable.range(0, 1000)
    .skip(10)
    .take(10)
    .subscribe(System.err::println);
```
RxJava at High speed

```java
Observable.interval(1, TimeUnit.SECONDS)
    .take(5)
    .subscribe(System.err::println);
```

0 (with one second delay each)
1
2
3
4
RxJava at High speed

Bytes.fromClassPath("citiesafrica.xml")
    .lift(XML.parse())
    .subscribe(System.err::println);

<cities>
    <africa>
        <city name="Lusaka"/>
        <city name="Harare"/>
        <city name="Kigali"/>
    </africa>
</cities>

START_DOCUMENT
START_ELEMENT cities
START_ELEMENT africa
START_ELEMENT city {name=Lusaka}
END_ELEMENT city
START_ELEMENT city {name=Harare}
END_ELEMENT city
START_ELEMENT city {name=Kigali}
END_ELEMENT city
END_ELEMENT africa
END_ELEMENT cities
END_DOCUMENT

XML parsing example: https://github.com/flyaruu/xml-rx
RxJava at High speed

Bytes.fromClassPath("citiesafrica.xml")
  .lift(XML.parse())
  .filter(e->e.getType()==XmlEventTypes.START_ELEMENT)
  .map(e->e.getAttributes().get("name"))
  .subscribe(System.err::println);

<cities>
  <africa>
    <city name="Lusaka"/>  Lusaka
    <city name="Harare"/>  Harare
    <city name="Kigali"/>  Kigali
  </africa>
</cities>
RxJava at High speed

```java
Database.from("jdbc:...
    .select("select name from cities where continent = ? order by name")
    .parameter("africa")
    .getAs(String.class)
    .subscribe(System.err::println);
```

- Harare
- Lusaka
- Kigali
RxJava at High speed

MQTT.onTopic("iot/temperature")
   .parseJSON()
   .get("temperature")
   .subscribe(System.err::println);

20.4
21.5
22.8
...

protected void doGet(HttpServletRequest req, final HttpServletResponse resp) {
    MongoDatabase database = mongoClient.getDatabase("mydb");
    MongoCollection<Document> collection = database.getCollection("test");
    collection
        .find()
        .writeAsJSON()
        .subscribe(createOutput(req));
}
Non Blocking Servlets

```java
static Observable<byte[]> createReadListener(HttpServletRequest req);
static Subscriber<byte[]> createOutput(HttpServletRequest req);

protected void doPost(HttpServletRequest req, final HttpServletResponse resp) {
    createReadListener(req)
        .subscribe(createOutput(req));
}
```
protected void doPost(HttpServletResponse resp) {
  String requestEncoding = (String) req.getHeader("Content-Encoding");
  Observable<byte[]> in =
    createReadListener(req)
    .lift(NavajoStreamOperators.decompress(requestEncoding))
    .lift(XML.parse())
    .lift(DomBuilder.parse())
    .lift(...)
    .subscribe(createOutput(req));
}
Observable API’s

Connecting hoses
public static Double temperatureInCity(String city) {
    return Double.parseDouble(HTTPClient.get("http://api.weather.org/weather?q=\"+city\"" +city)
            .toXml()
            .getContent("temperature");
}

For consumers:

Double temperature = temperatureInCity("Tripoli");
public Observable<Double> temperatureInCity(String city) {
    return Observable.just(
        Double.parseDouble(HTTPClient
            .get("http://api.weather.org/weather?q="+city)
            .toXml()
            .getContent("temperature")
        ));
}
SO YOU'RE TELLING ME

YOU CAN SIMPLY WRAP BLOCKING CODE INTO AN OBSERVABLE?
Well...

```java
Double temp = temperatureInCity("Cairo").toBlocking().first();
updateTemperatureUI(temp);
```

This code is still just as blocking

But now the consumer and producer are independent of threading
The API provider and consumer choose their own threading
Use a non blocking HTTP client

```java
public static Observable<Double> temperatureInCity(String city) {
    return HTTP.get("http://api.weather.org/weather?q="+city)
        .subscribeOn(Schedulers.io())
        .lift(XML.parse())
        .filter(e->e.getType()==XmlEventTypes.START_ELEMENT)
        .filter(e->e.getText().equals("temperature"))
        .first()
        .map(xml->Double.parseDouble(xml.getAttributes().get("value")));
}
```

And the consumer notices nothing...
Non blocking consumer

Blocking:

Double temp = temperatureInCity("Cairo").toBlocking().first();
updateTemperatureUI(temp);

Non blocking:

temperatureInCity("Nairobi")
  .observeOn(UISchedulers.uiThread())
  .subscribe(d->updateTemperatureUI(d));
public Observable<Double> temperatureInCity(String city) {
    Double temperature = temperatureCache.get(city);
    if (temperature != null) {
        return Observable.just(temperature);
    }
    return HTTP.get("http://api.weather.org/weather?q="+city)
        .subscribeOn(Schedulers.io())
        .lift(XML.parse())
        .filter(e->e.getType()==XmlEventTypes.START_ELEMENT)
        .filter(e->e.getText().equals("temperature"))
        .first()
        .map(xml->Double.parseDouble(xml.getAttributes().get("value")));
}
It’s just a library

• Not a new language
• Not a new framework
• Can be implemented incrementally
• Only needs a recent Java version and a recent Servlet Engine (if you use servlets)
Limitations
Non blocking is not faster

- Non blocking is about utilising threads better
- If thread utilisation is not an issue, it will perform about the same
Memory consumption

• Might be much better

• … or much worse
Backpressure
Backpressure

Our pipe:

```java
static Observable<byte[]> createReadListener(HttpServletRequest req);
static Subscriber<byte[]> createOutput(HttpServletRequest req);

protected void doPost(HttpServletRequest req, final HttpServletResponse resp) {
    createReadListener(req)
        .subscribe(createOutput(req));
}
Backpressure

- What if the request is HUGE
- and the request stream is fast
- … but the response stream is slow?
Things I skipped:

- Error handling
- Scheduling
- ByteBuffers and Bufferpools
- A lot of fancy operators
Conclusions
Thank you!