Functional Reactive Programming in the Netflix API

QCon London – March 6 2013

Ben Christensen

Software Engineer – API Platform at Netflix

@benjchristensen

http://www.linkedin.com/in/benjchristensen

http://techblog.netflix.com/

NETFLIX

Questions? Call 0800 096 6379 – 24 hours a day

Member Sign In

1 MONTH FREE TRIAL

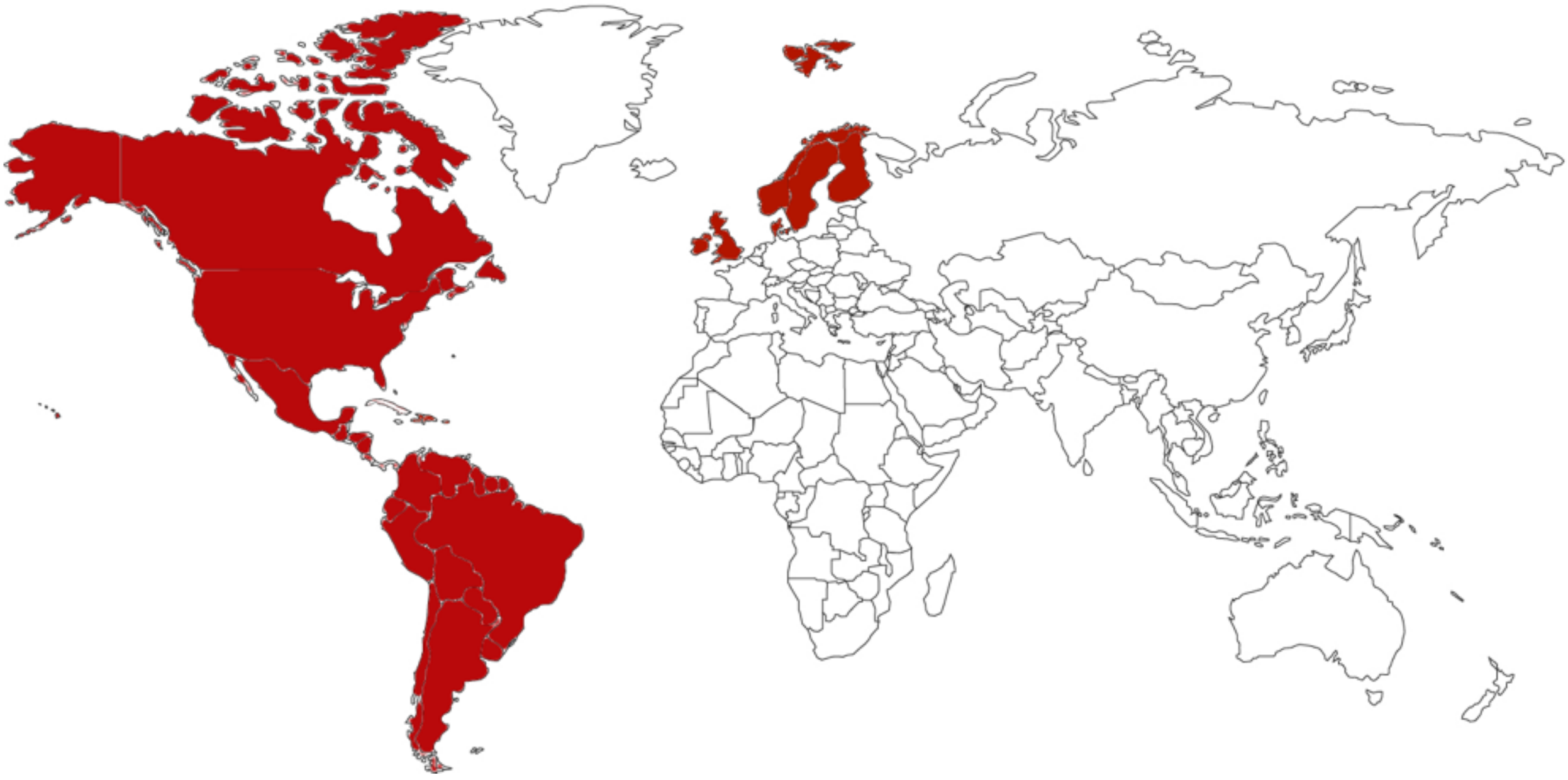
Watch TV programmes & films anytime, anywhere.

Only £5.99 a month.

Start Your Free Month

- Watch on your **PS3**, **Wii**, **Xbox**, **PC**, **Mac**, **Mobile**, **Tablet** and more.
- TV programmes & films streamed instantly over the internet
- No commitments, **cancel online at any time**

More than 33 million Subscribers
in more than 50 Countries and Territories



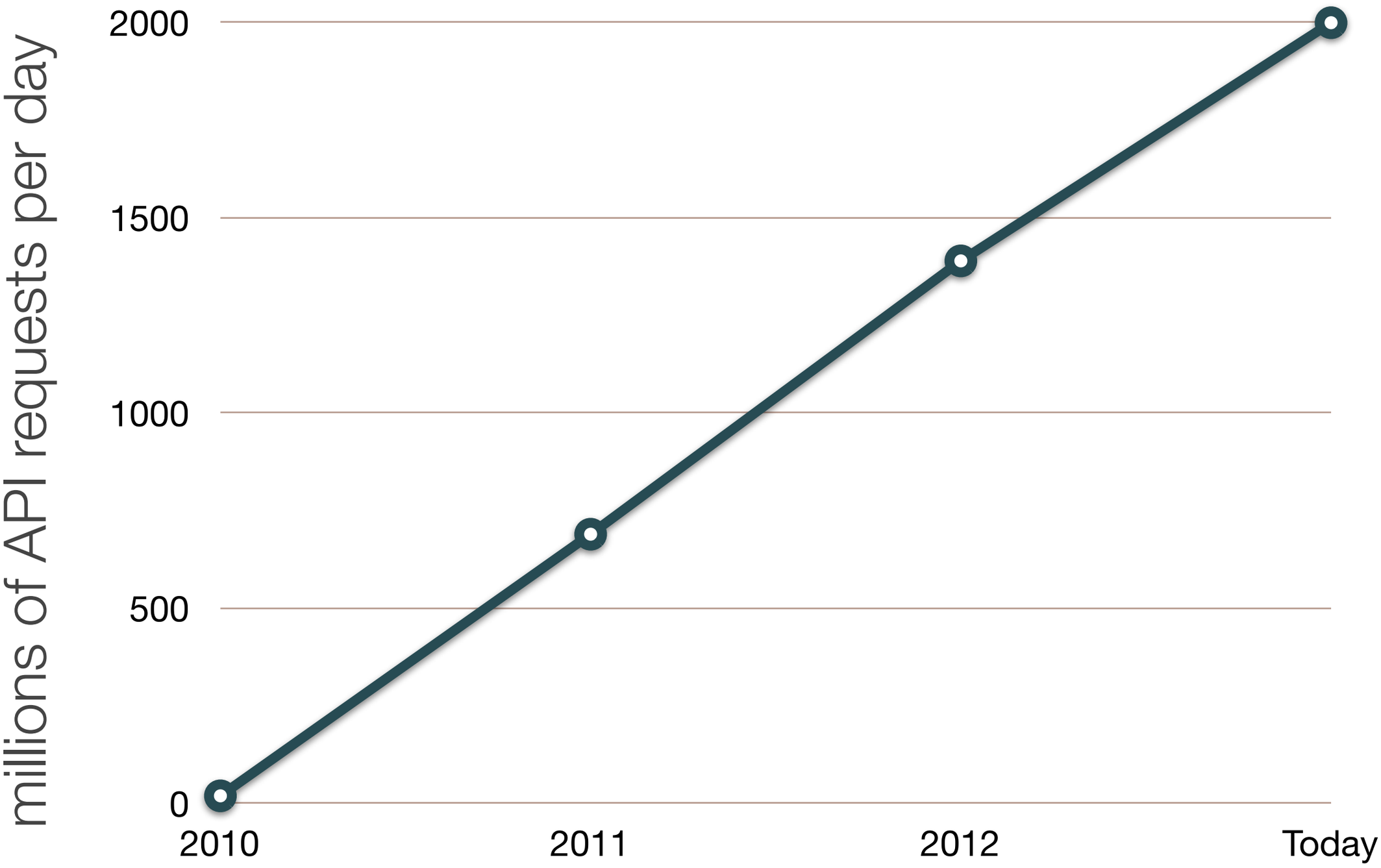
Netflix accounts for 33% of Peak Downstream Internet Traffic in North America

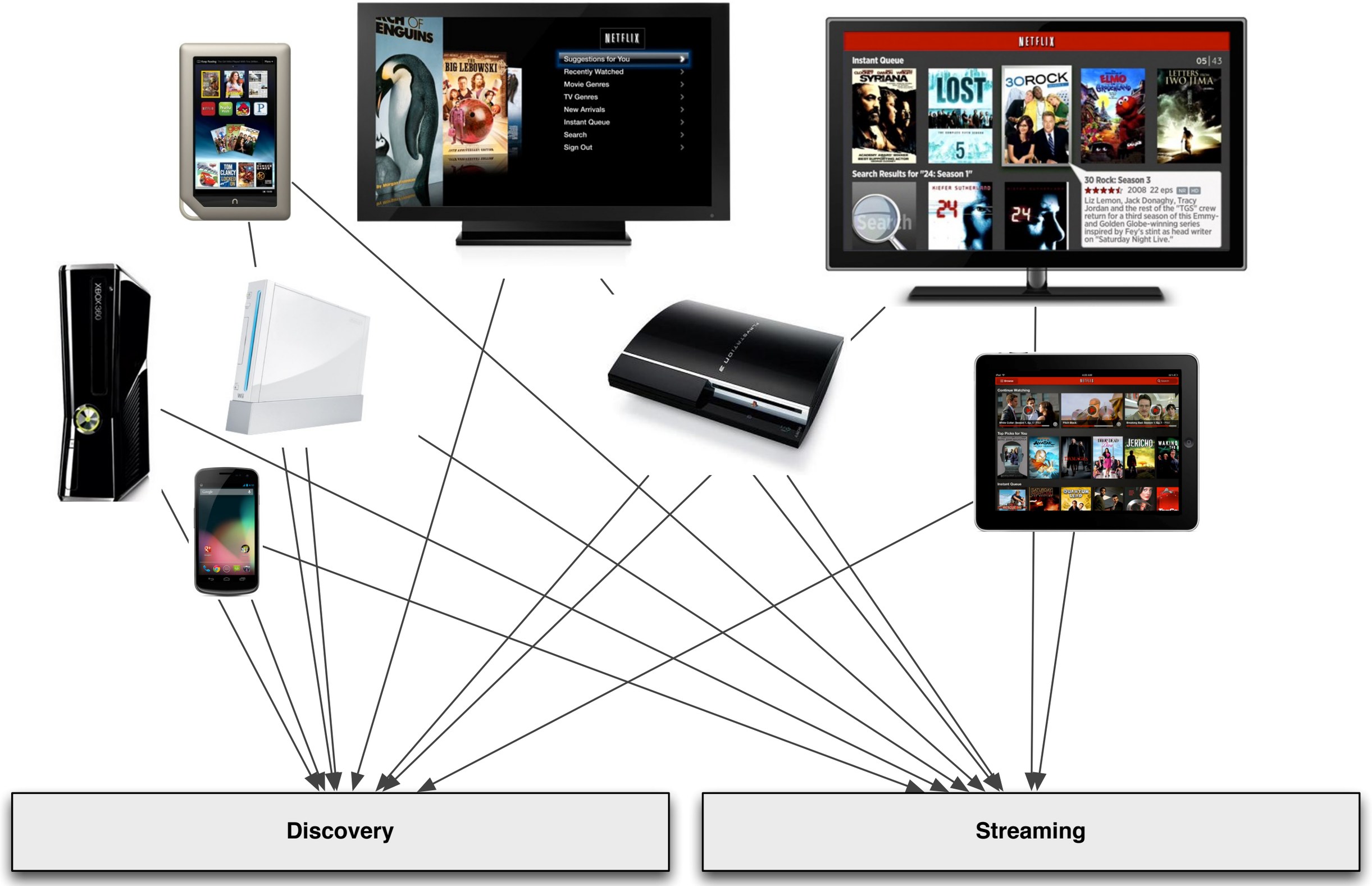
	Upstream		Downstream		Aggregate	
Rank	Application	Share	Application	Share	Application	Share
1	BitTorrent	36.8%	Netflix	33.0%	Netflix	28.8%
2	HTTP	9.83%	YouTube	14.8%	YouTube	13.1%
3	Skype	4.76%	HTTP	12.0%	HTTP	11.7%
4	Netflix	4.51%	BitTorrent	5.89%	BitTorrent	10.3%
5	SSL	3.73%	iTunes	3.92%	iTunes	3.43%
6	YouTube	2.70%	MPEG	2.22%	SSL	2.23%
7	PPStream	1.65%	Flash Video	2.21%	MPEG	2.05%
8	Facebook	1.62%	SSL	1.97%	Flash Video	2.01%
9	Apple PhotoStream	1.46%	Amazon Video	1.75%	Facebook	1.50%
10	Dropbox	1.17%	Facebook	1.48%	RTMP	1.41%
	Top 10	**68.24%**	**Top 10**	**79.01%**	**Top 10**	**76.54%**

sandvine

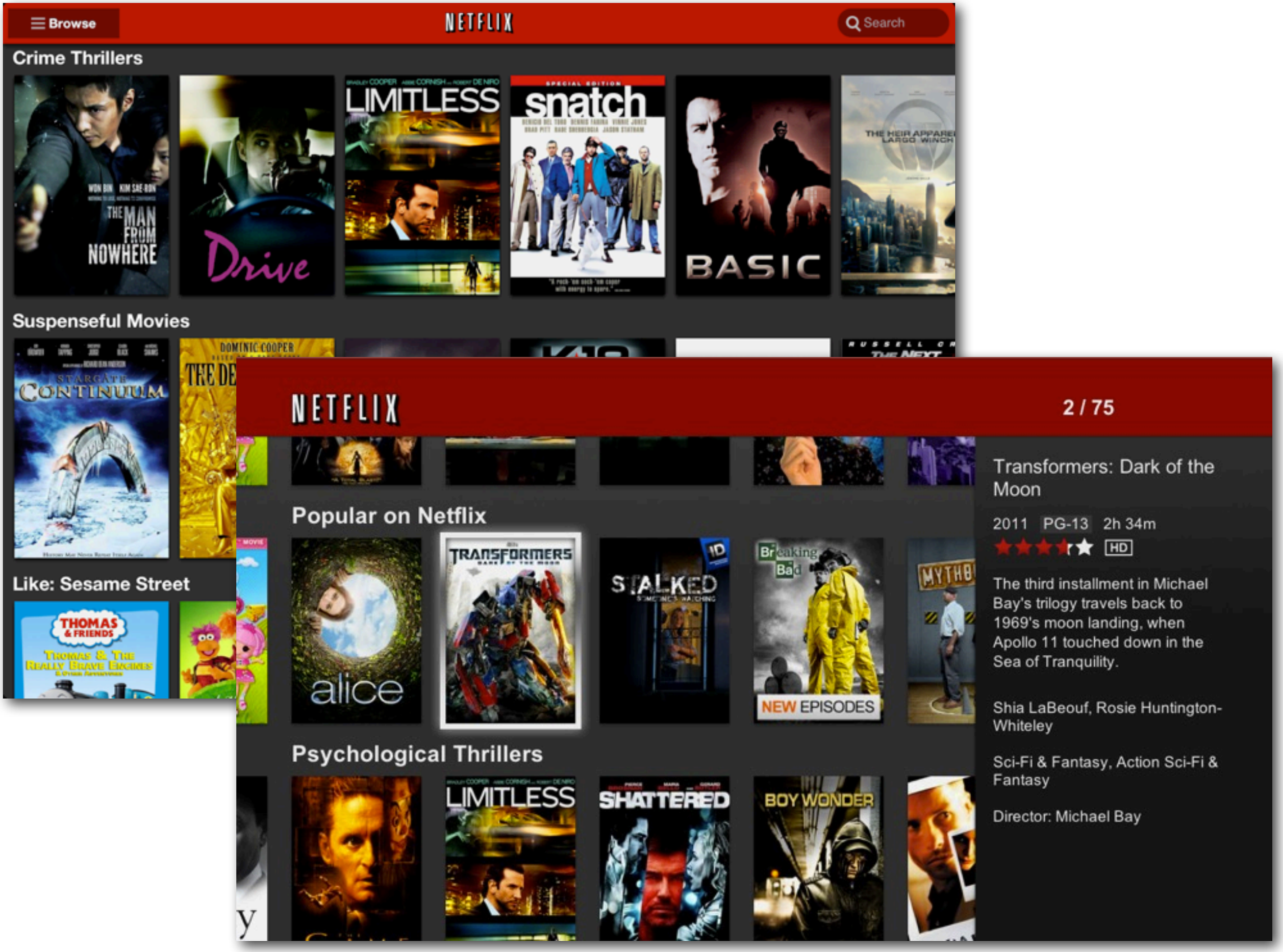
Netflix subscribers are watching more than 1 billion hours a month

API traffic has grown from ~20 million/day in 2010 to >2 billion/day

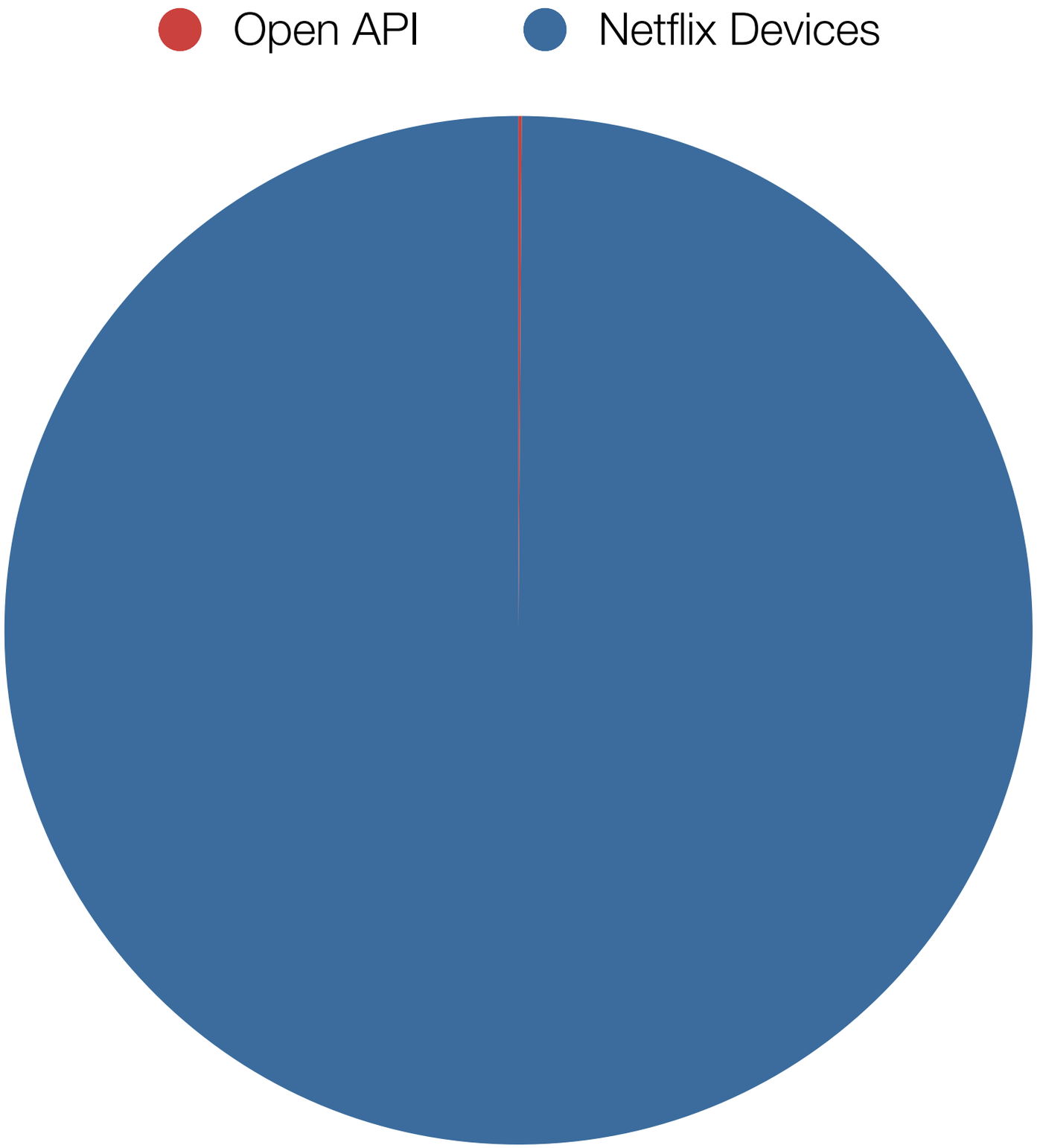


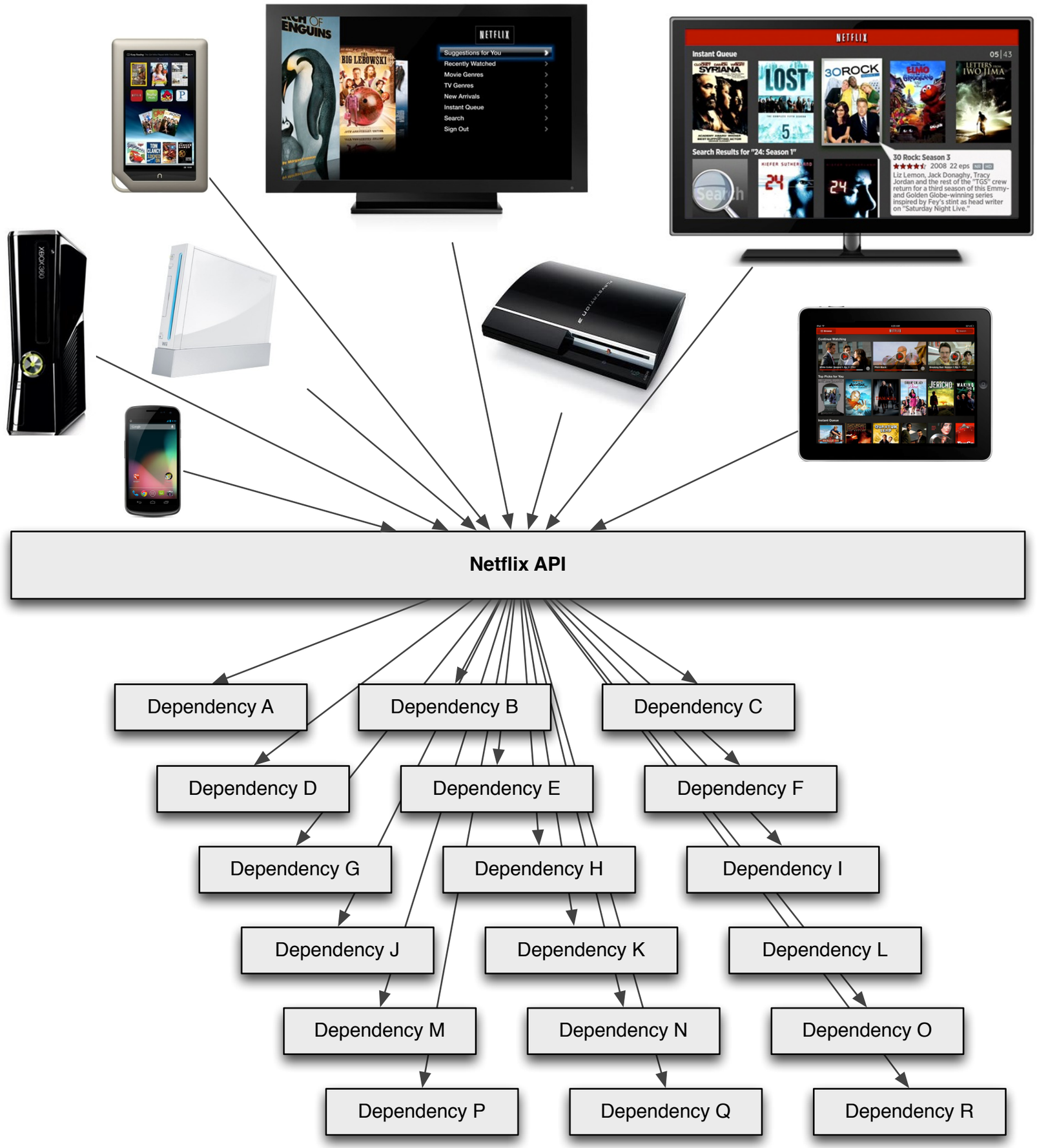

NETFLIX
Suggestions for You
Recently Watched
Movie Genres
TV Genres
New Arrivals
Instant Queue
Search
Sign Out
NETFLIX
Instant Queue
05|43
Search Results for "24: Season 1"
30 Rock: Season 3
2008 22 eps
Liz Lemon, Jack Donaghy, Tracy Jordan and the rest of the "TGS" crew return for a third season of this Emmy- and Golden Globe-winning series inspired by Fey's stint as head writer on "Saturday Night Live."
Discovery
Streaming


NETFLIX
Suggestions for You
Recently Watched
Movie Genres
TV Genres
New Arrivals
Instant Queue
Search
Sign Out
Instant Queue
05|43
Search Results for "24: Season 1"
30 Rock: Season 3
2008 22 eps
Liz Lemon, Jack Donaghy, Tracy Jordan and the rest of the "TGS" crew return for a third season of this Emmy- and Golden Globe-winning series inspired by Fey's stint as head writer on "Saturday Night Live."
Netflix API
Streaming


Browse
NETFLIX
Search
Crime Thrillers
THE MAN FROM NOWHERE
Drive
LIMITLESS
snatch
BASIC
Suspenseful Movies
STARGATE CONTINUUM
Like: Sesame Street
THOMAS & FRIENDS
Popular on Netflix
alice
TRANSFORMERS
STALKED
Breaking Bad
NEW EPISODES
Psychological Thrillers
LIMITLESS
SHATTERED
BOY WONDER
2 / 75
Transformers: Dark of the Moon
2011 PG-13 2h 34m
HD
The third installment in Michael Bay's trilogy travels back to 1969's moon landing, when Apollo 11 touched down in the Sea of Tranquility.
Shia LaBeouf, Rosie Huntington-Whiteley
Sci-Fi & Fantasy, Action Sci-Fi & Fantasy
Director: Michael Bay

API Request Volume by Audience




NETFLIX
Suggestions for You
Recently Watched
Movie Genres
TV Genres
New Arrivals
Instant Queue
Search
Sign Out
Instant Queue
Search Results for "24: Season 1"
30 Rock: Season 3
Liz Lemon, Jack Donaghy, Tracy Jordan and the rest of the "TGS" crew return for a third season of this Emmy- and Golden Globe-winning series inspired by Fey's stint as head writer on "Saturday Night Live."
XBOX360
Netflix API
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R

/ps3/home

/android/home

/tv/home

Functional Reactive Dynamic Endpoints

Asynchronous Java API

Dependency A
10 Threads

Dependency B
8 Threads

Dependency C
10 Threads

Dependency D
15 Threads

Dependency E
5 Threads

Dependency F
10 Threads

Dependency G
10 Threads

Dependency H
10 Threads

Dependency I
5 Threads

Dependency J
8 Threads

Dependency K
15 Threads

Dependency L
4 Threads

Dependency M
5 Threads

Dependency N
10 Threads

Dependency O
10 Threads

Dependency P
10 Threads

Dependency Q
8 Threads

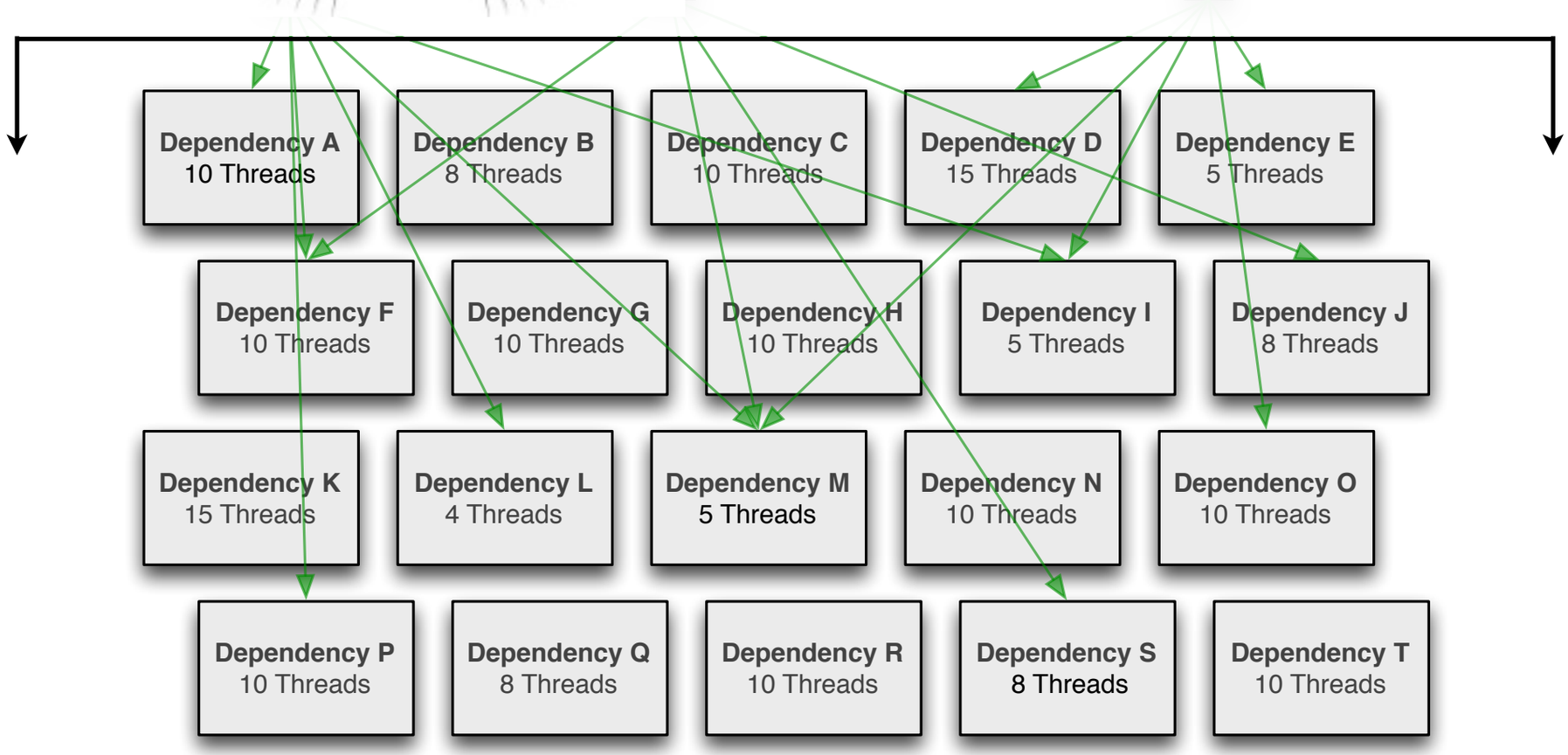
Dependency R
10 Threads

Dependency S
8 Threads

Dependency T
10 Threads

Hystrix

fault-isolation layer



/ps3/home

/android/home

/tv/home

Functional Reactive Dynamic Endpoints

Asynchronous Java API

Dependency A 10 Threads	**Dependency B** 8 Threads	**Dependency C** 10 Threads	**Dependency D** 15 Threads	**Dependency E** 5 Threads
Dependency F 10 Threads	**Dependency G** 10 Threads	**Dependency H** 10 Threads	**Dependency I** 5 Threads	**Dependency J** 8 Threads
Dependency K 15 Threads	**Dependency L** 4 Threads	**Dependency M** 5 Threads	**Dependency N** 10 Threads	**Dependency O** 10 Threads
Dependency P 10 Threads	**Dependency Q** 8 Threads	**Dependency R** 10 Threads	**Dependency S** 8 Threads	**Dependency T** 10 Threads

RxJava

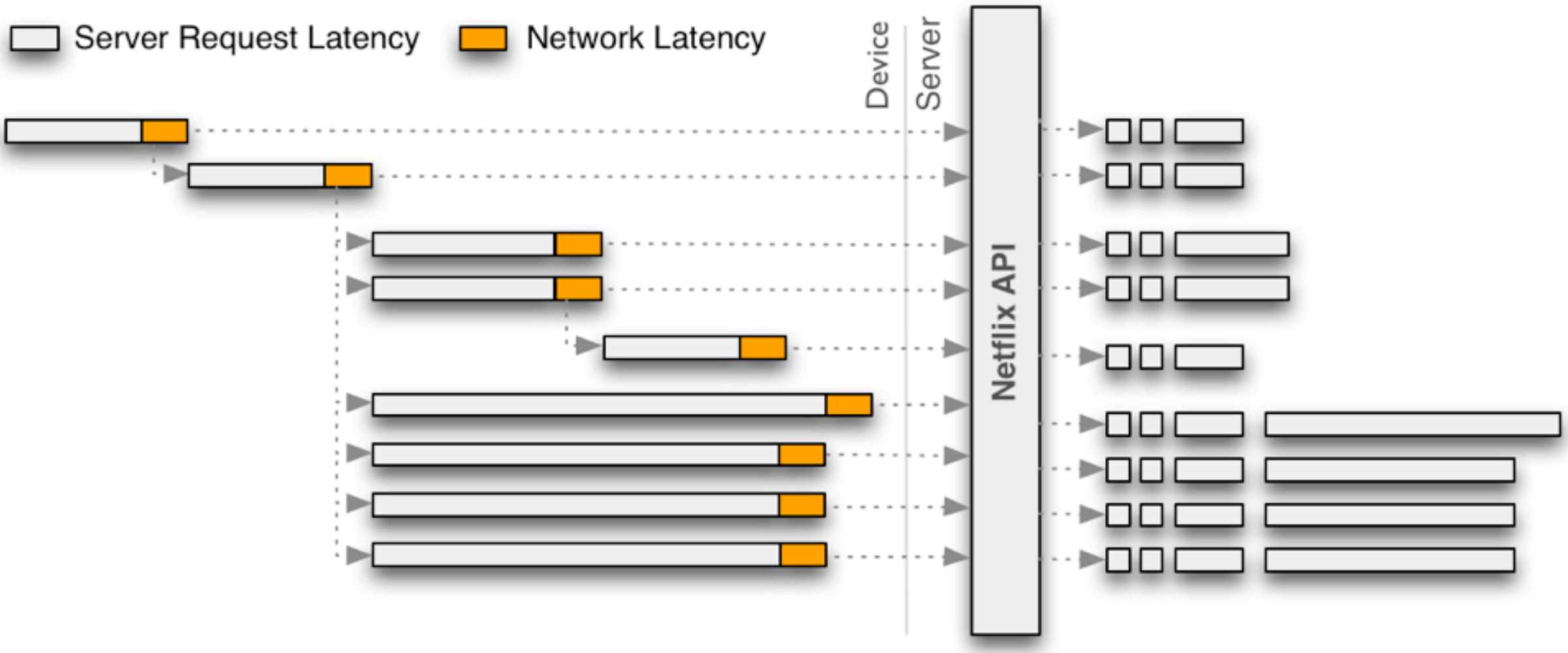
"a library for composing asynchronous and event-based programs using observable sequences for the Java VM"

A Java port of Rx (Reactive Extensions)

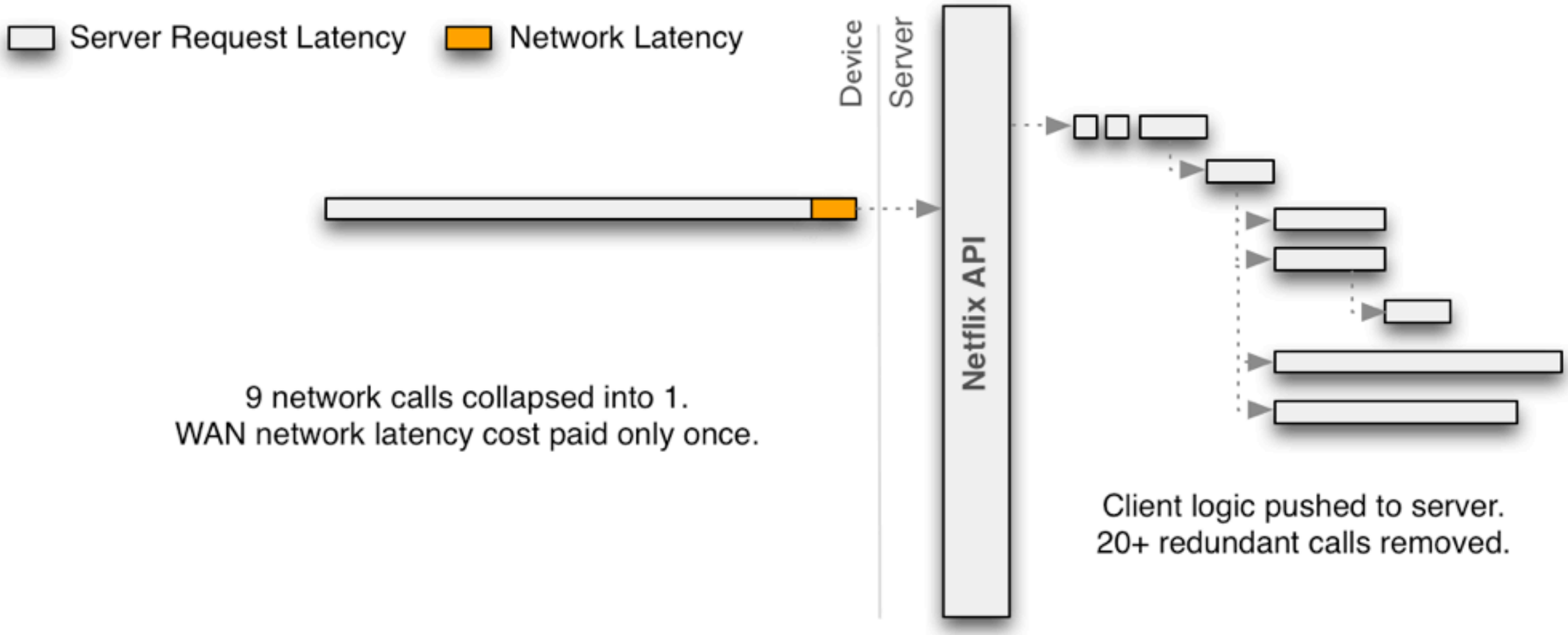
https://rx.codeplex.com (.Net and Javascript by Microsoft)

Do we really need another way of “managing” concurrency?

Discovery of Rx began with a re-architecture ...



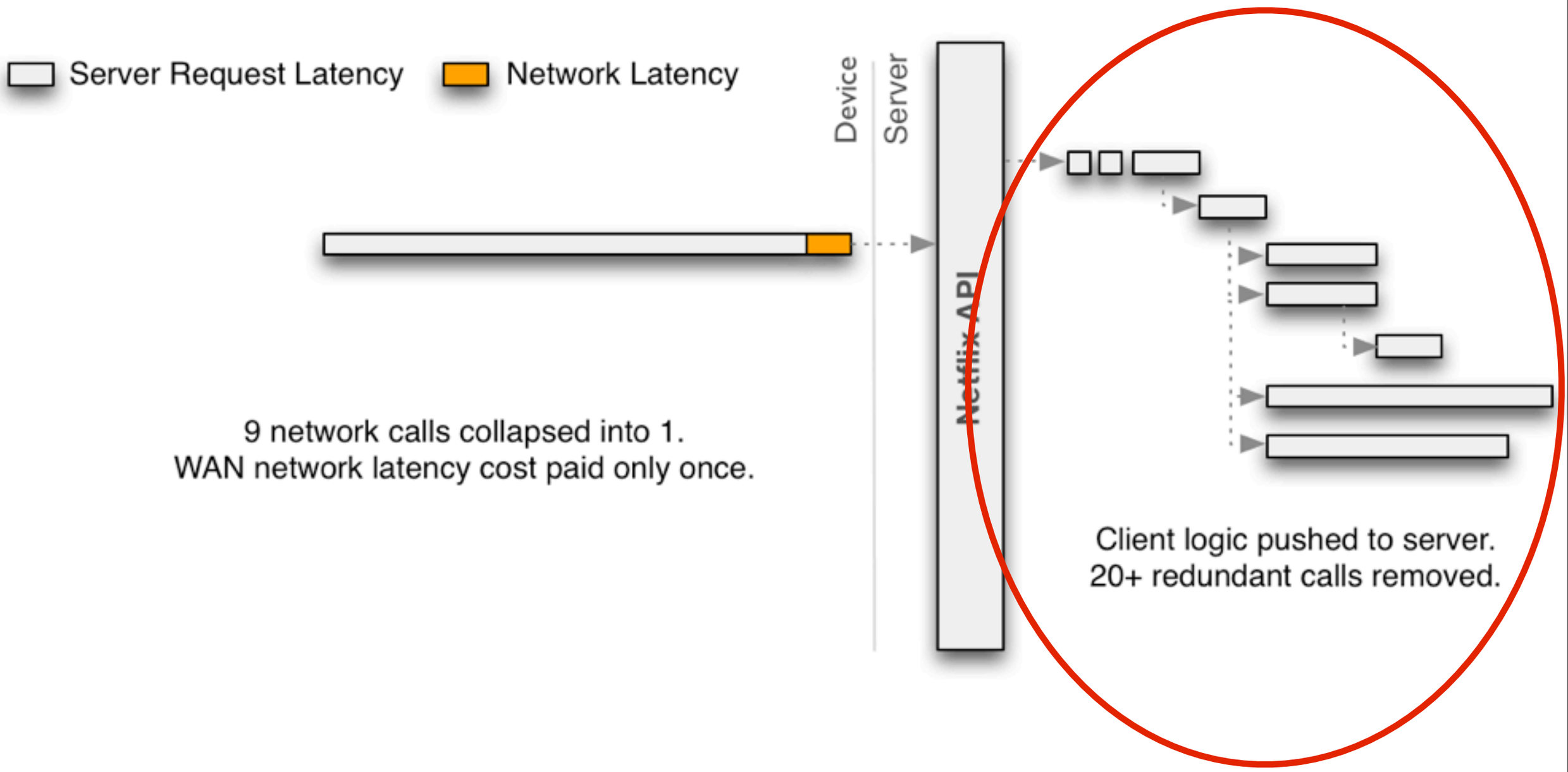
... that collapsed network traffic into coarse API calls ...



9 network calls collapsed into 1.
WAN network latency cost paid only once.

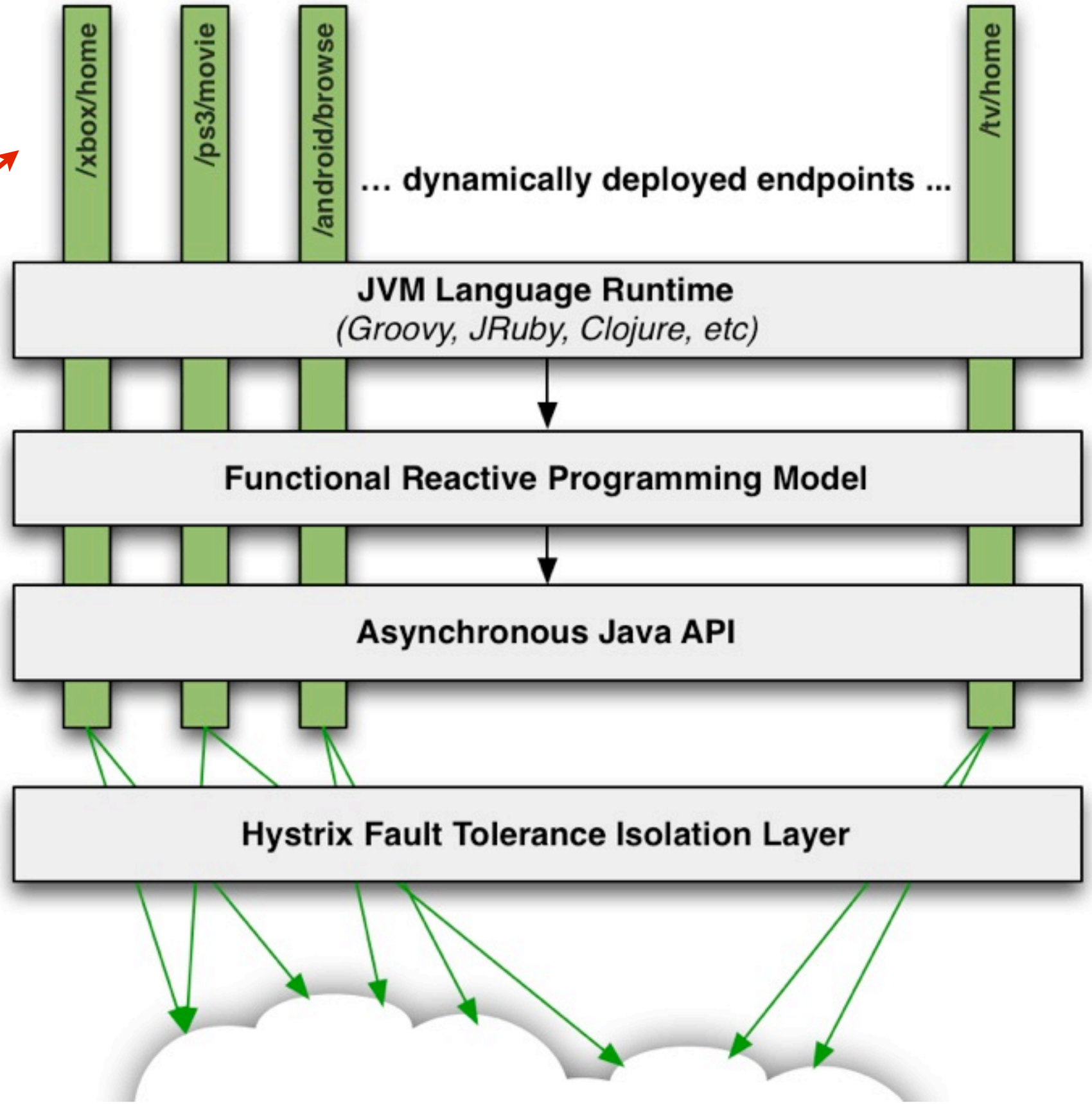
Client logic pushed to server.
20+ redundant calls removed.

... that collapsed network traffic into coarse API calls ...



Nested, conditional, parallel execution

... and we wanted to allow anybody to create endpoints, not just the "API Team"




Java


Clojure


Scala

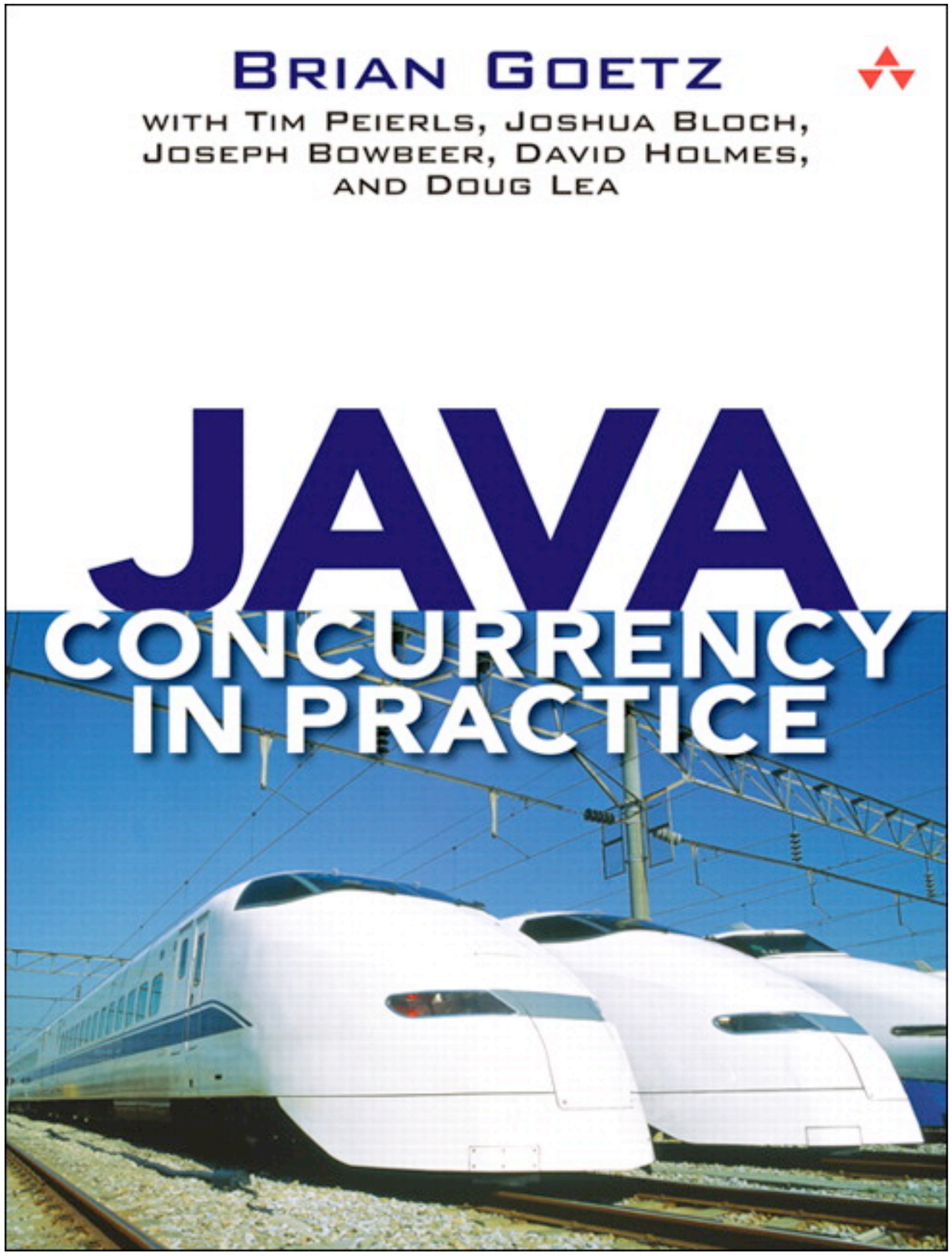

JavaScript


Groovy


JRuby

Concurrency without each engineer reading and re-reading this ->

(awesome book ... everybody isn't going to - or should have to - read it though, that's the point)



Owner of API should retain control of concurrency behavior.

Owner of API should retain control of concurrency behavior.

```
public Data getData();
```

What if the implementation needs to change from synchronous to asynchronous?

How should the client execute that method without blocking? spawn a thread?

```
public void getData(Callback<T> c);

public Future<T> getData();

public Future<List<Future<T>>> getData();
```

What about ... ?

Iterable	**Observable**
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()

Iterable	**Observable**
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()

```
// Iterable<String>
// that contains 75 Strings
getDataFromLocalMemory()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
 .forEach(
    { println "next => " + it})
```

```
// Observable<String>
// that emits 75 Strings
getDataFromNetwork()
 .skip(10)
 .take(5)
 .map({ s ->
  return s + "_transformed"})
 .subscribe(
    { println "onNext => " + it})
```

Iterable	**Observable**
pull	*push*
T next()	onNext(T)
throws Exception	onError(Exception)
returns;	onCompleted()

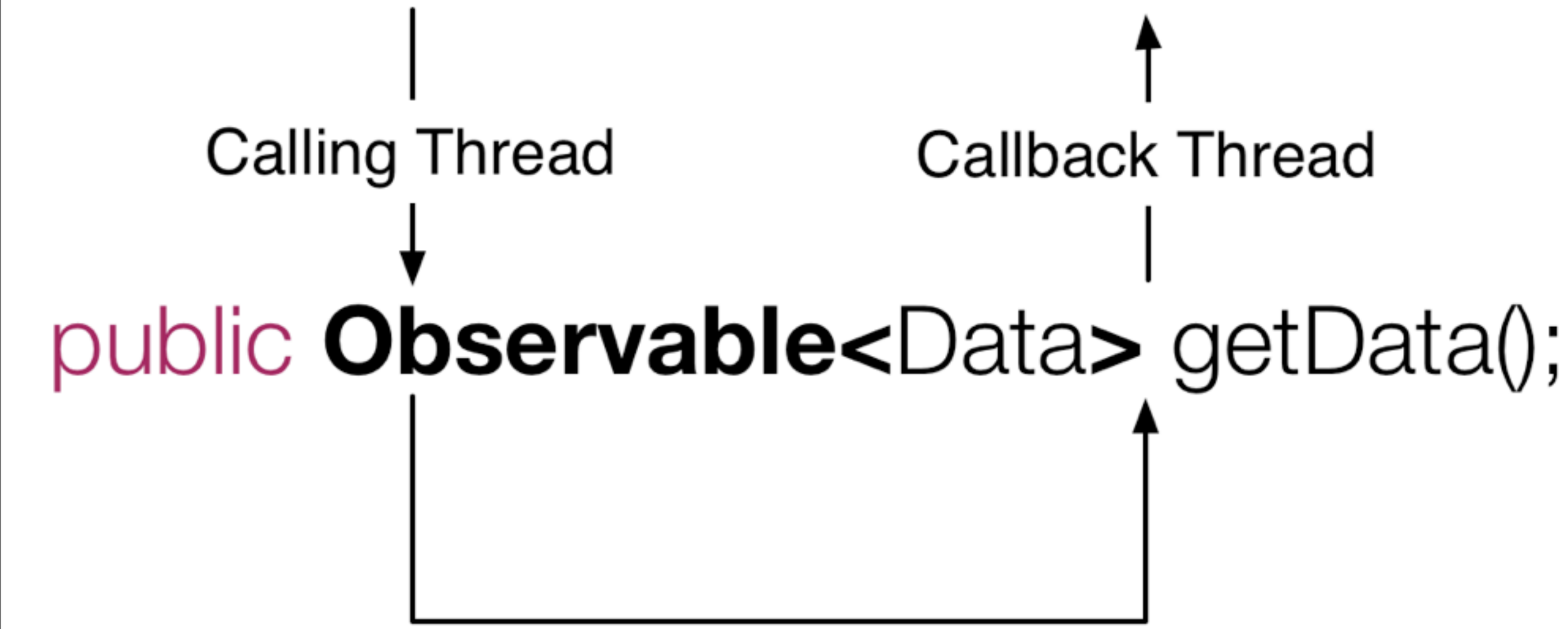
```
// ~~Iterable~~<String>                    // Observable<String>
// that contains 75 Strings                // that emits 75 Strings
getDataFromLocalMemory()                   getDataFromNetwork()
 .skip(10)                                  .skip(10)
 .take(5)                                   .take(5)
 .map({ s ->                                .map({ s ->
  return s + "_transformed"})                return s + "_transformed"})
 .~~forEach~~(                              .subscribe(
    { println "onNext => " + it})              { println "onNext => " + it})
```

Instead of blocking APIs ...

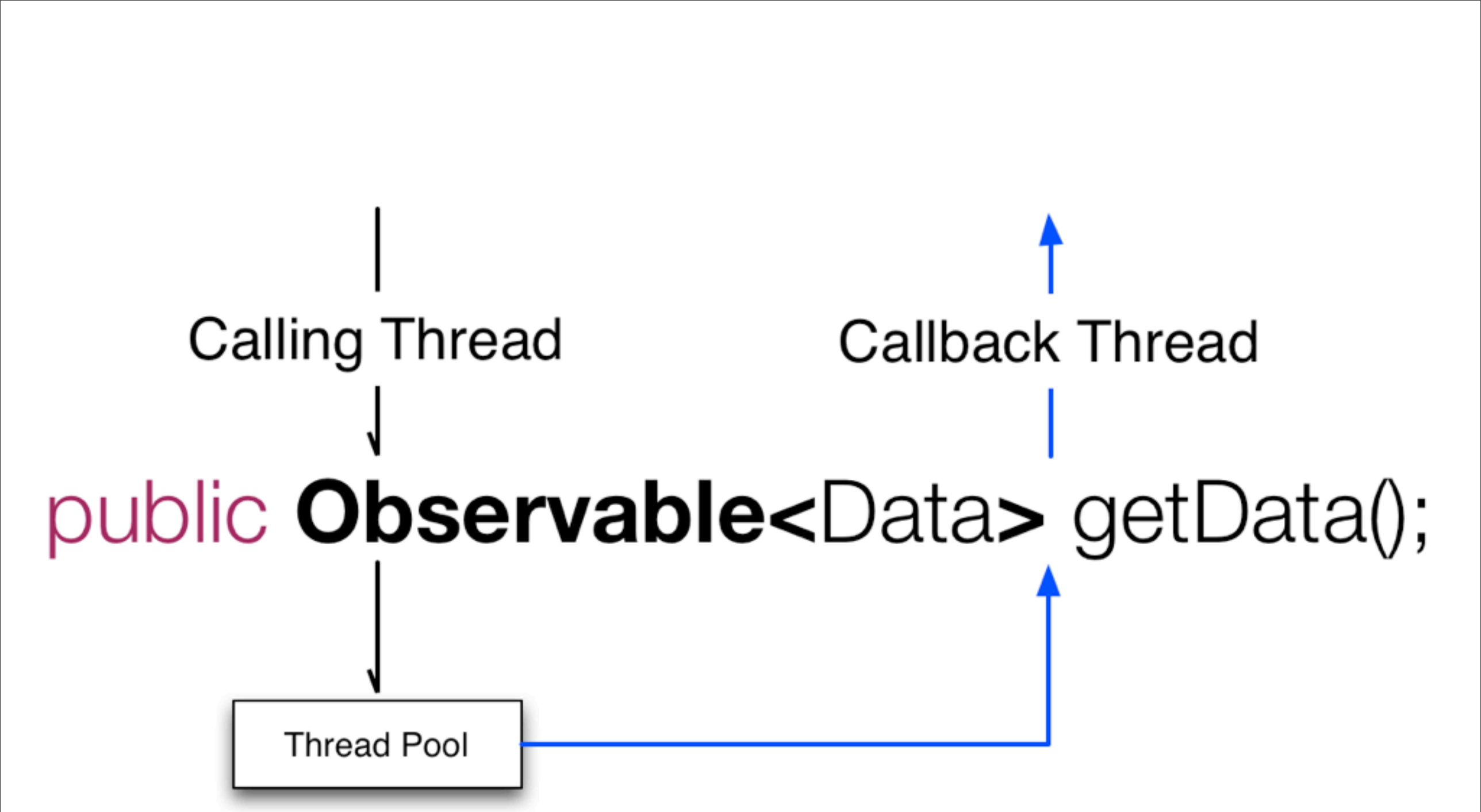
```
class VideoService {
   def VideoList getPersonalizedListOfMovies(userId);
   def VideoBookmark getBookmark(userId, videoId);
   def VideoRating getRating(userId, videoId);
   def VideoMetadata getMetadata(videoId);
}
```

... create Observable APIs:

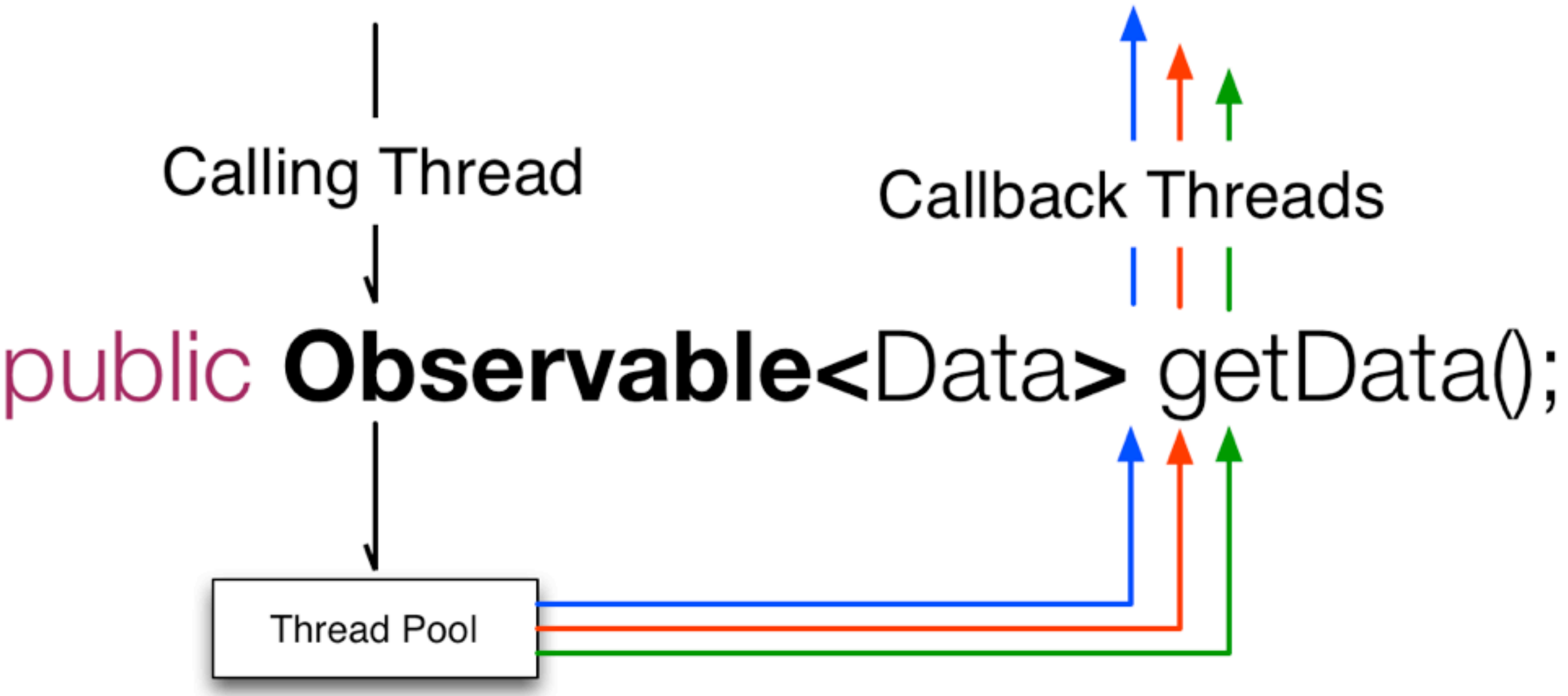
```
class VideoService {
   def Observable<VideoList> getPersonalizedListOfMovies(userId);
   def Observable<VideoBookmark> getBookmark(userId, videoId);
   def Observable<VideoRating> getRating(userId, videoId);
   def Observable<VideoMetadata> getMetadata(videoId);
}
```



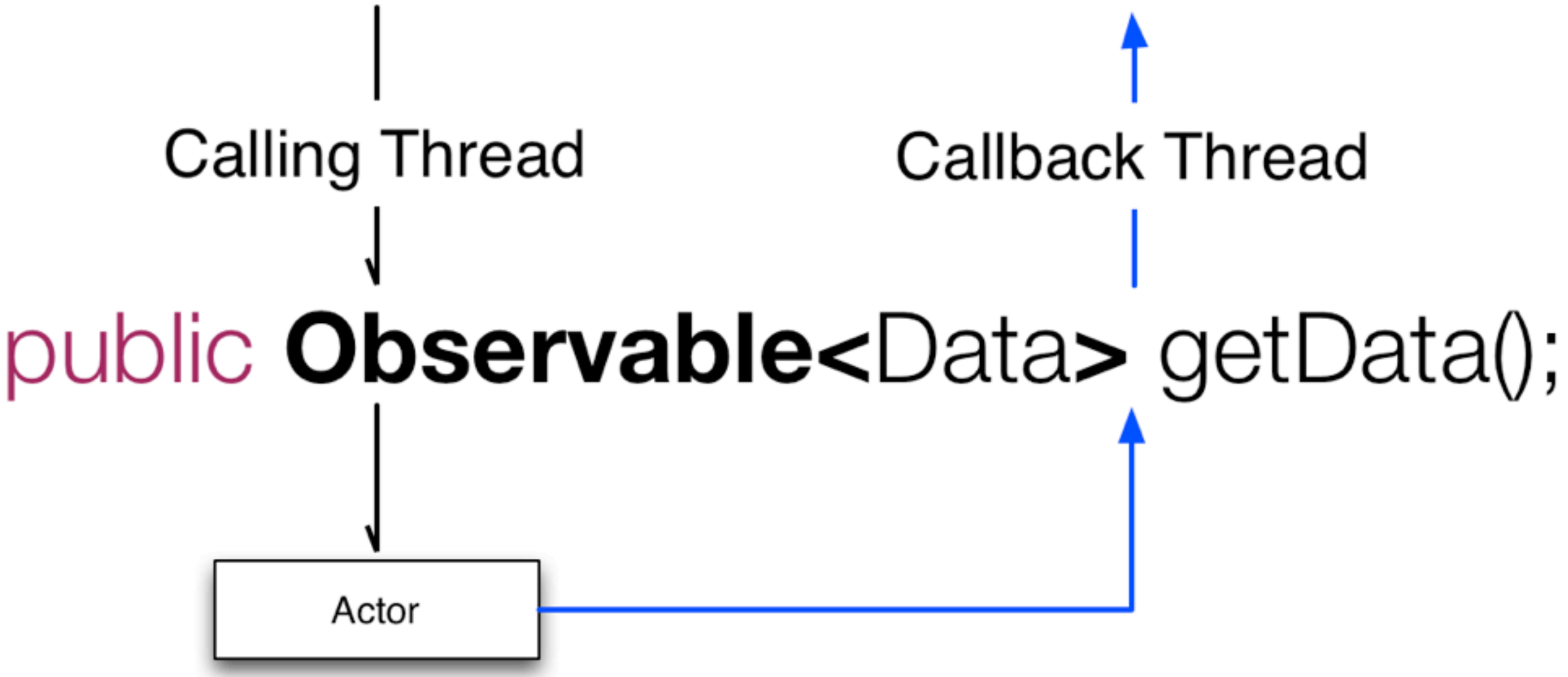
Do work synchronously on calling thread.



Do work asynchronously on a separate thread.



Do work asynchronously on a multiple threads.

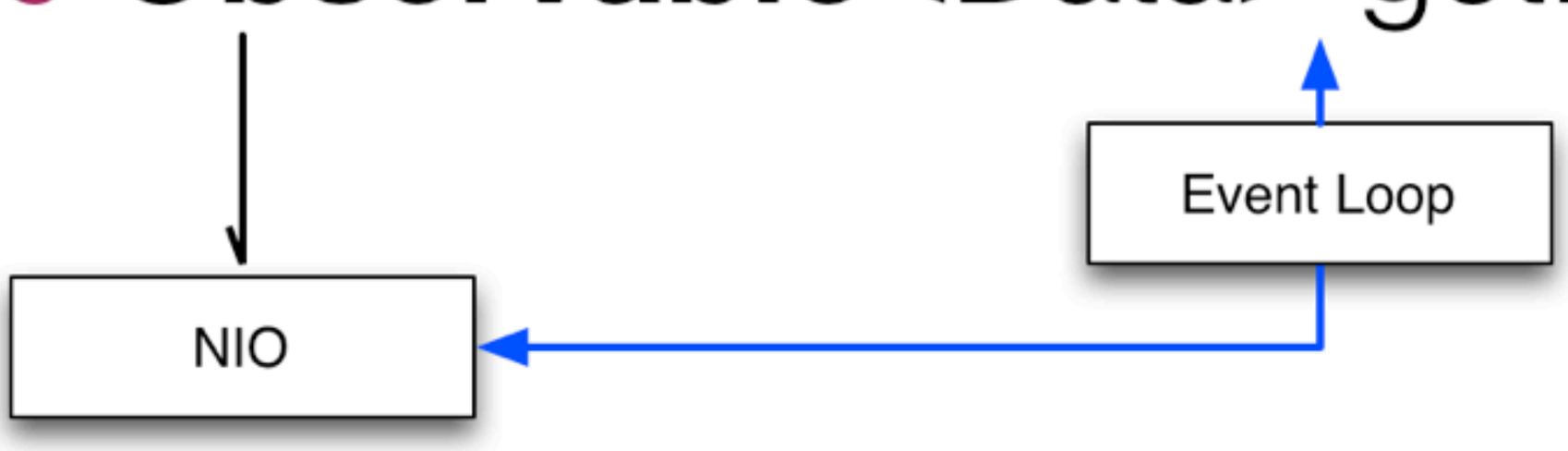


Do work asynchronously on an actor
(or multiple actors).

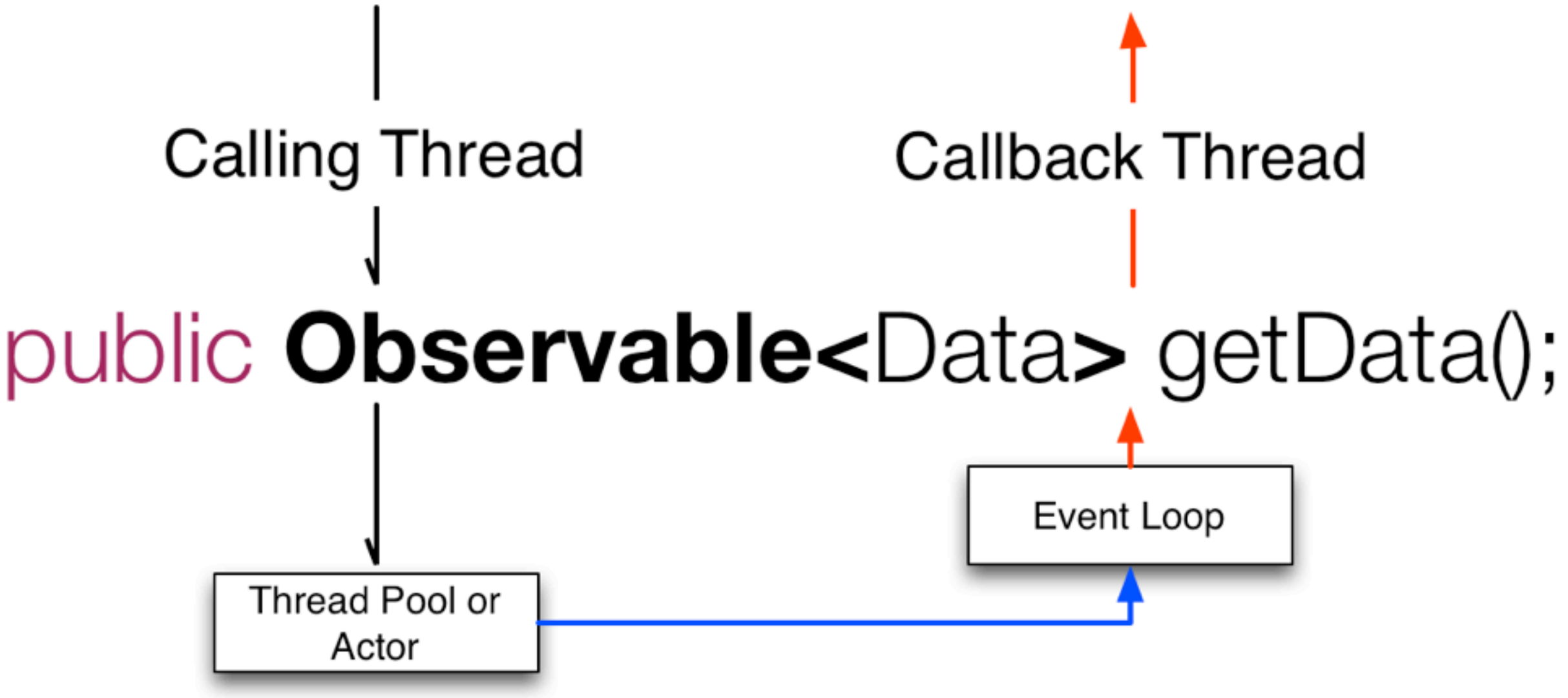
Calling Thread

Callback Thread

```
public Observable<Data> getData();
```



Do network access asynchronously using NIO and perform callback on Event Loop



Do work asynchronously and perform callback via a single or multi-threaded event loop.

Clojure

```
(->
  (Observable/toObservable ["one" "two" "three"])
  (.take 2)
  (.subscribe (fn [arg] (println arg))))
```

Scala

```
Observable.toObservable("one", "two", "three")
  .take(2)
  .subscribe((arg: String) => {
      println(arg)
  })
```

Groovy

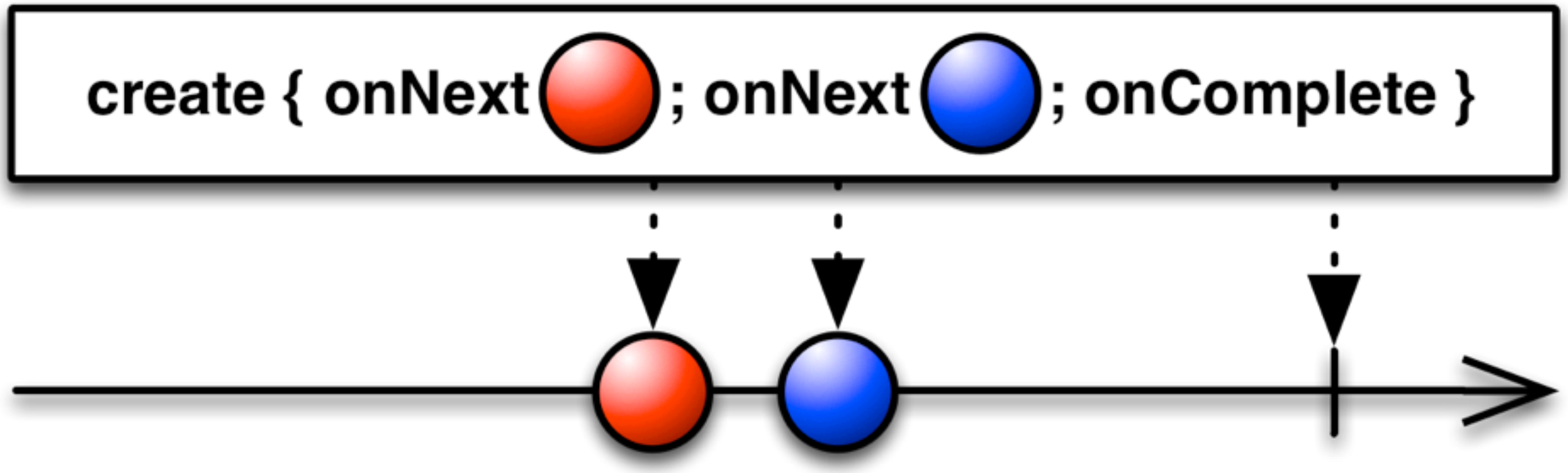
```
Observable.toObservable("one", "two", "three")
  .take(2)
  .subscribe({arg -> println(arg)})
```

JRuby

```
Observable.toObservable("one", "two", "three")
  .take(2)
  .subscribe(lambda { |arg| puts arg })
```

Java8

```
Observable.toObservable("one", "two", "three")
      .take(2)
      .subscribe((arg) -> {
            System.out.println(arg);
      });
```




```
Observable.create({ observer ->
  try {
    observer.onNext(new Video(id))
    observer.onCompleted();
  } catch(Exception e) {
    observer.onError(e);
  }
})
```

Asynchronous Observable with Single Value

```
def Observable<VideoRating> getRating(userId, videoId) {
    // fetch the VideoRating for this user asynchronously
    return Observable.create({ observer ->
        executor.execute(new Runnable() {
            def void run() {
              try {
                VideoRating rating = ... do network call ...
                observer.onNext(rating)
                observer.onCompleted();
              } catch(Exception e) {
                observer.onError(e);
              }
            }
        })
    })
}
```

Asynchronous Observable with Single Value

```
def Observable<VideoRating> getRating(userId, videoId) {
    // fetch the VideoRating for this user asynchronously
    return Observable.create({ observer ->
        executor.execute(new Runnable() {
            def void run() {
              try {
                VideoRating rating = ... do network call ...
                observer.onNext(rating)
                observer.onCompleted();
              } catch(Exception e) {
                observer.onError(e);
              }
            }
        })
    })
}
```

Synchronous Observable with Multiple Values

```
def Observable<Video> getVideos() {
    return Observable.create({ observer ->
        try {
            for(v in videos) {
                observer.onNext(v)
            }
            observer.onCompleted();
        } catch(Exception e) {
            observer.onError(e);
        }
    })
}
```

Caution: This is eager and will *always* emit all values regardless of subsequent operators such as take(10)

Synchronous Observable with Multiple Values

```
def Observable<Video> getVideos() {
    return Observable.create({ observer ->
        try {
            for(v in videos) {
                observer.onNext(v)
            }
            observer.onCompleted();
        } catch(Exception e) {
            observer.onError(e);
        }
    })
}
```

Caution: This is eager and will *always* emit all values regardless of subsequent operators such as take(10)

Asynchronous Observable with Multiple Values

```
def Observable<Video> getVideos() {
    return Observable.create({ observer ->
        executor.execute(new Runnable() {
            def void run() {
                try {
                    for(id in videoIds) {
                        Video v = ... do network call ...
                        observer.onNext(v)
                    }
                    observer.onCompleted();
                } catch(Exception e) {
                    observer.onError(e);
                }
            }
        })
    })
}
```

Asynchronous Observable with Multiple Values

```
def Observable<Video> getVideos() {
    return Observable.create({ observer ->
        executor.execute(new Runnable() {
            def void run() {
                try {
                    for(id in videoIds) {
                        Video v = ... do network call ...
                        observer.onNext(v)
                    }
                    observer.onCompleted();
                } catch(Exception e) {
                    observer.onError(e);
                }
            }
        })
    })
}
```

Combining via Merge

```
Observable<SomeData> a = getDataA();
Observable<SomeData> b = getDataB();
Observable<SomeData> c = getDataC();

Observable.merge(a, b, c)
    .subscribe(
        { element -> println("data: " + element)},
        { exception -> println("error occurred: "
                               + exception.getMessage())}
    )
```

Combining via Zip

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();
Observable<MoreData> c = getDataC();

Observable.zip(a, b, c, {x, y, z -> [x, y, z]})
    .subscribe(
        { triple -> println("a: " + triple[0]
                          + " b: " + triple[1]
                          + " c: " + triple[2])},
        { exception -> println("error occurred: "
                               + exception.getMessage())}
    )
```

Error Handling

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();
Observable<MoreData> c = getDataC();

Observable.zip(a, b, c, {x, y, z -> [x, y, z]})
    .subscribe(
        { triple -> println("a: " + triple[0]
                          + " b: " + triple[1]
                          + " c: " + triple[2])},
        { exception -> println("error occurred: "
                               + exception.getMessage())}
    )
```

Error Handling

```
Observable<SomeData> a = getDataA();
Observable<String> b = getDataB();
Observable<MoreData> c = getDataC()
    .onErrorResumeNext(getFallbackForDataC());

Observable.zip(a, b, c, {x, y, z -> [x, y, z]})
    .subscribe(
        { triple -> println("a: " + triple[0]
                          + " b: " + triple[1]
                          + " c: " + triple[2])},
        { exception -> println("error occurred: "
                               + exception.getMessage())}
    )
```

```
def Observable getVideos(userId) {
    return VideoService.getVideos(userId)

}
```

```
def Observable getVideos(userId) {
    return VideoService.getVideos(userId)

}
```

Asynchronous request that returns Observable<Video>

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
}
```

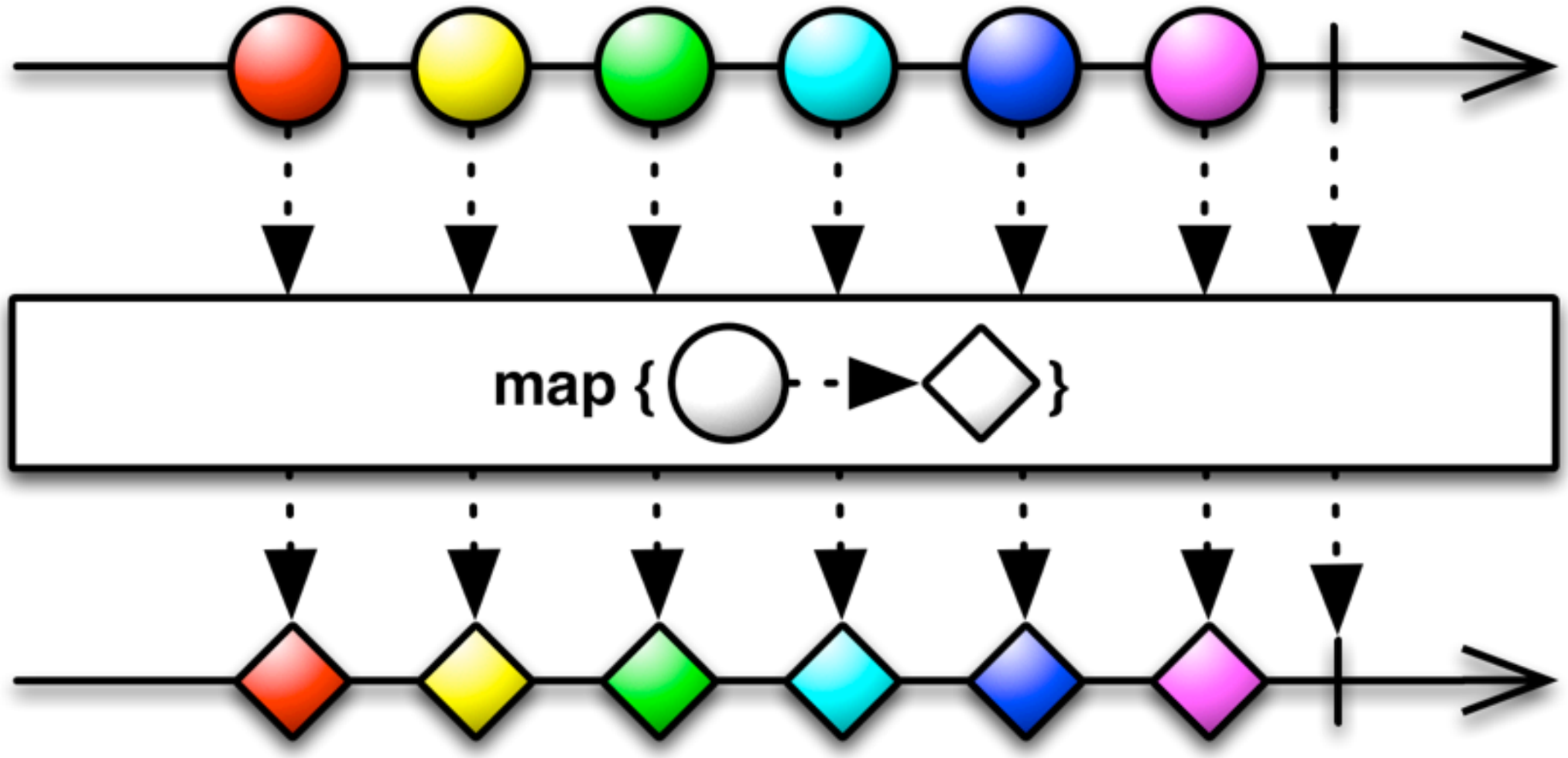
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
}
```

Reactive operator on the Observable that takes the first 10 Video objects then unsubscribes.

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .map({ Video video ->
           // transform video object
        })
}
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .map({ Video video ->
            // transform video object
        })
}
```

The ‘map’ operator allows transforming the input value into a different output.

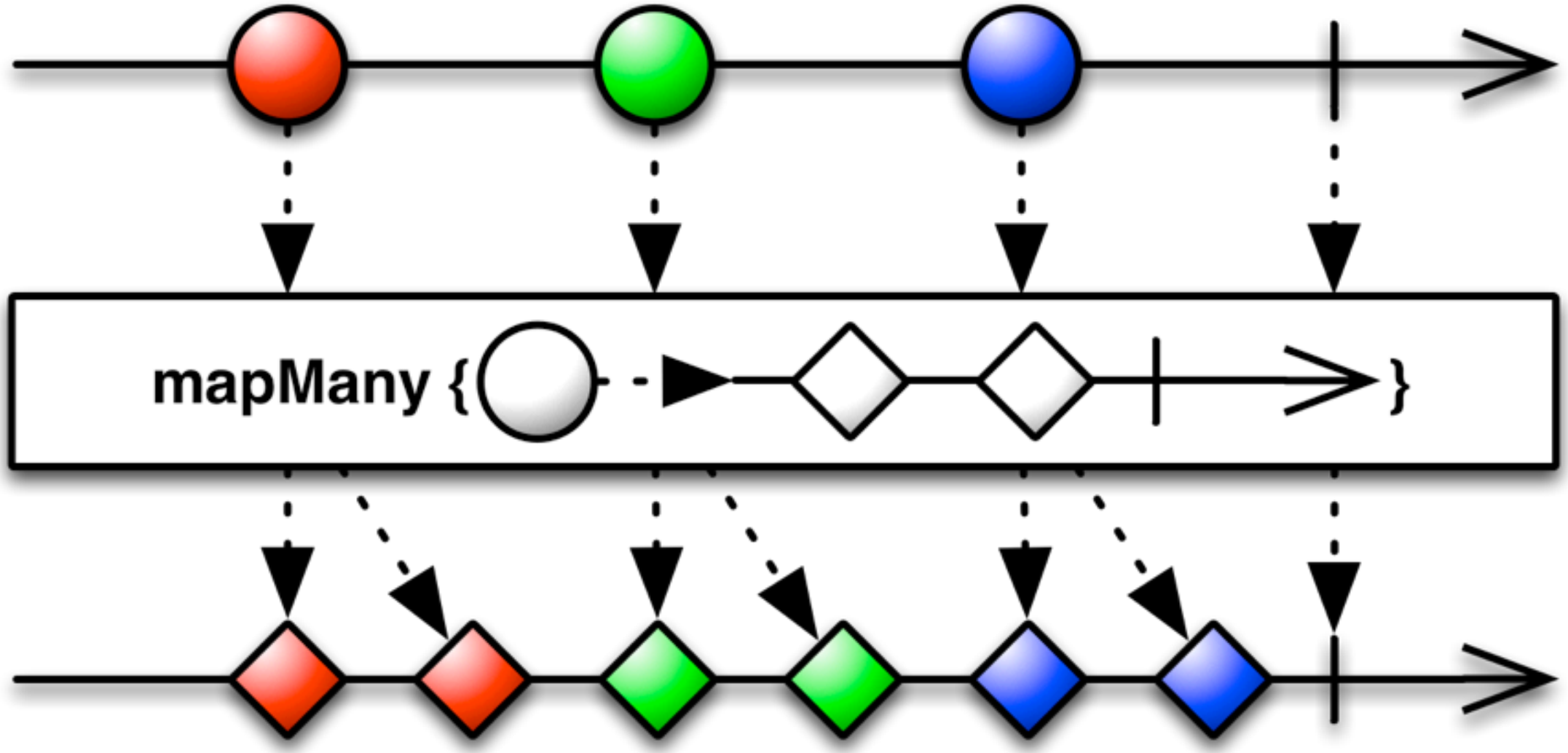



```
Observable<R> b = Observable<T>.map({ T t ->
  R r = ... transform t ...
  return r;
})
```

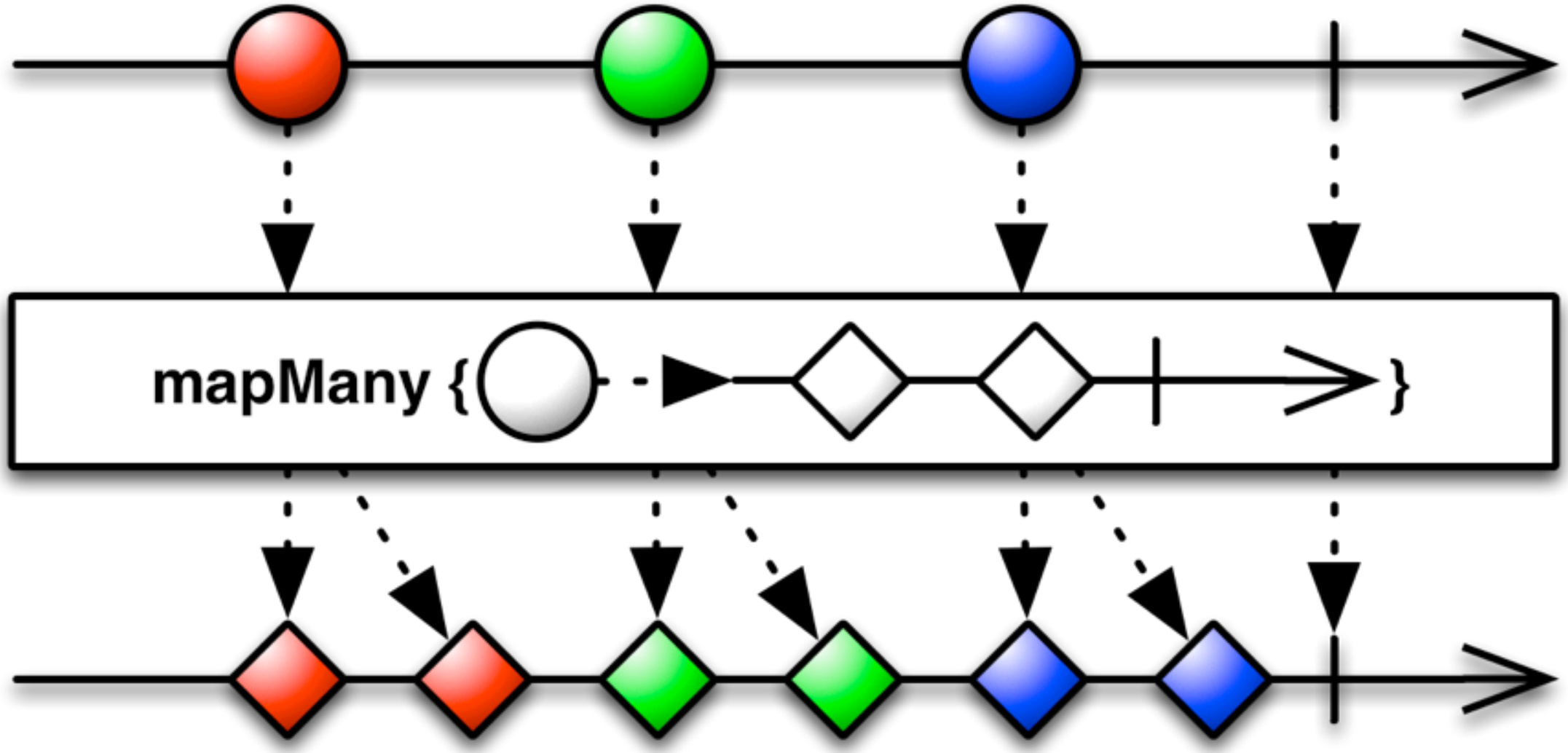
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
            // for each video we want to fetch metadata
            def m = video.getMetadata()
                .map({ Map<String, String> md ->
                // transform to the data and format we want
                    return [title: md.get("title"),
                            length: md.get("duration")]
                 })
            // and its rating and bookmark
            def b ...
            def r ...
        })
}
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
            // for each video we want to fetch metadata
            def m = video.getMetadata()
                .map({ Map<String, String> md ->
                // transform to the data and format we want
                  return [title: md.get("title"),
                          length: md.get("duration")]
                 })
                // and its rating and bookmark
                def b ...
                def r ...
        })
}
```

We change to ‘mapMany’ which is like merge(map()) since we will return an Observable<T> instead of T.




```
Observable<R> b = Observable<T>.mapMany({ T t ->
    Observable<R> r = ... transform t ...
    return r;
})
```




```
Observable<R> b = Observable<T>.mapMany({ T t ->
    Observable<R> r = ... transform t ...
    return r;
})
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
            // for each video we want to fetch metadata
            def m = video.getMetadata()
                .map({ Map<String, String> md ->
                    // transform to the data and format we want
                    return [title: md.get("title"),
                            length: md.get("duration")]
                })
            // and its rating and bookmark
            def b ...
            def r ...
        })
}
```

Nested asynchronous calls that return more Observables.

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
          // for each video we want to fetch metadata
          def m = video.getMetadata()
              .map({ Map<String, String> md ->
              // transform to the data and format we want
                return [title: md.get("title"),
                        length: md.get("duration")]
              })
          // and its rating and bookmark
          def b ...
          def r ...
        })
}
```

Observable<VideoMetadata>

Observable<VideoBookmark>

Observable<VideoRating>

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
            // for each video we want to fetch metadata
            def m = video.getMetadata()
                .map({ Map<String, String> md ->
                // transform to the data and format we want
                  return [title: md.get("title"),
                          length: md.get("duration")]
                 })
            // and its rating and bookmark
            def b ...
            def r ...
        })
}
```

Each Observable transforms its data using ‘map’

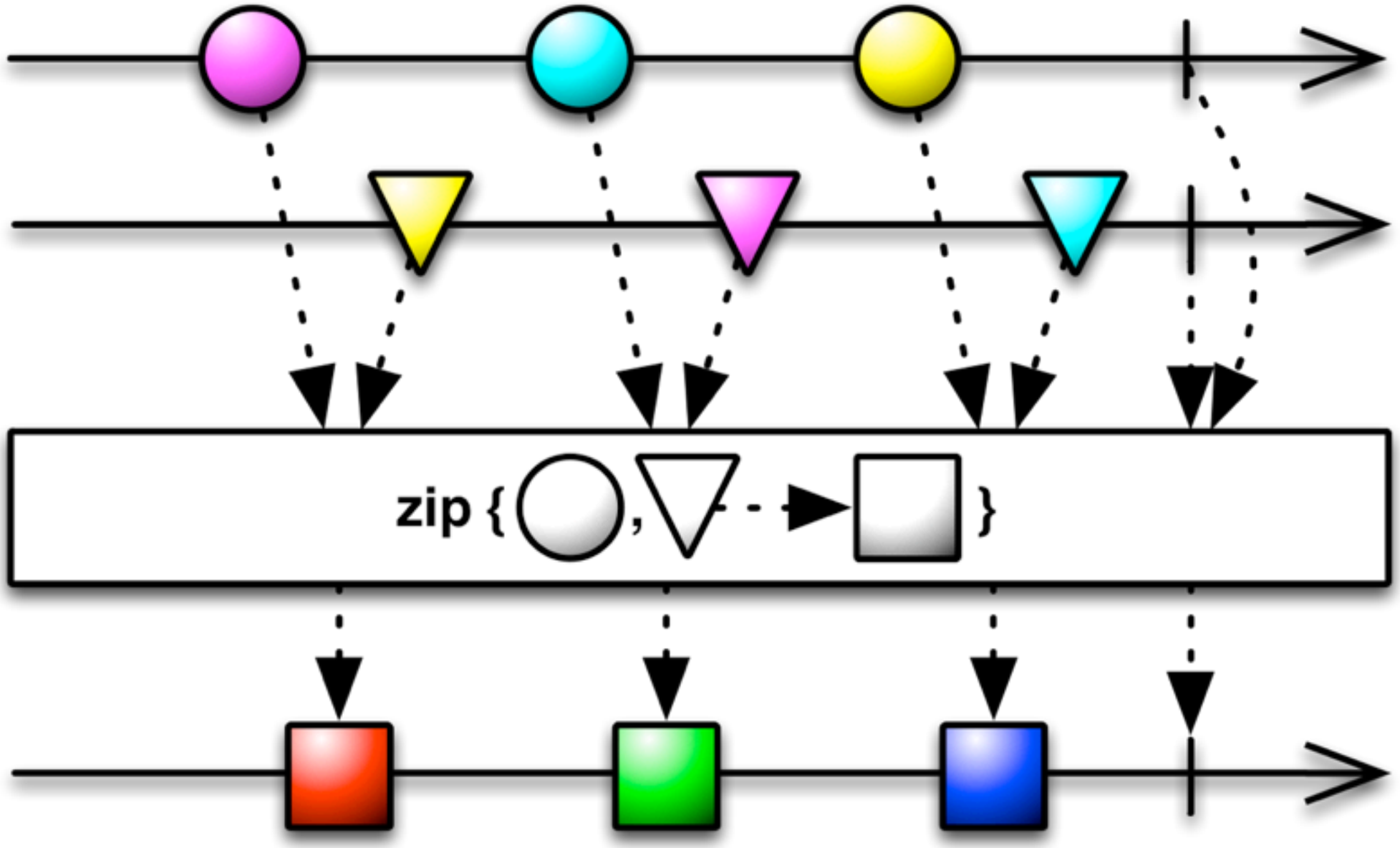
```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
          // for each video we want to fetch metadata
          def m = video.getMetadata()
              .map({ Map<String, String> md ->
              // transform to the data and format we want
                return [title: md.get("title"),
                        length: md.get("duration")]
              })
          // and its rating and bookmark
          def b ...
          def r ...
          // compose these together
        })
}
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
    })
}
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return Observable.zip(m, b, r, {
                        metadata, bookmark, rating ->
          // now transform to complete dictionary
          // of data we want for each Video
              return [id: video.videoId]
                            << metadata << bookmark << rating
          })
    })
}
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return Observable.zip(m, b, r, {
                        metadata, bookmark, rating ->
        // now transform to complete dictionary
        // of data we want for each Video
            return [id: video.videoId]
                        << metadata << bookmark << rating
        })
    })
}
```

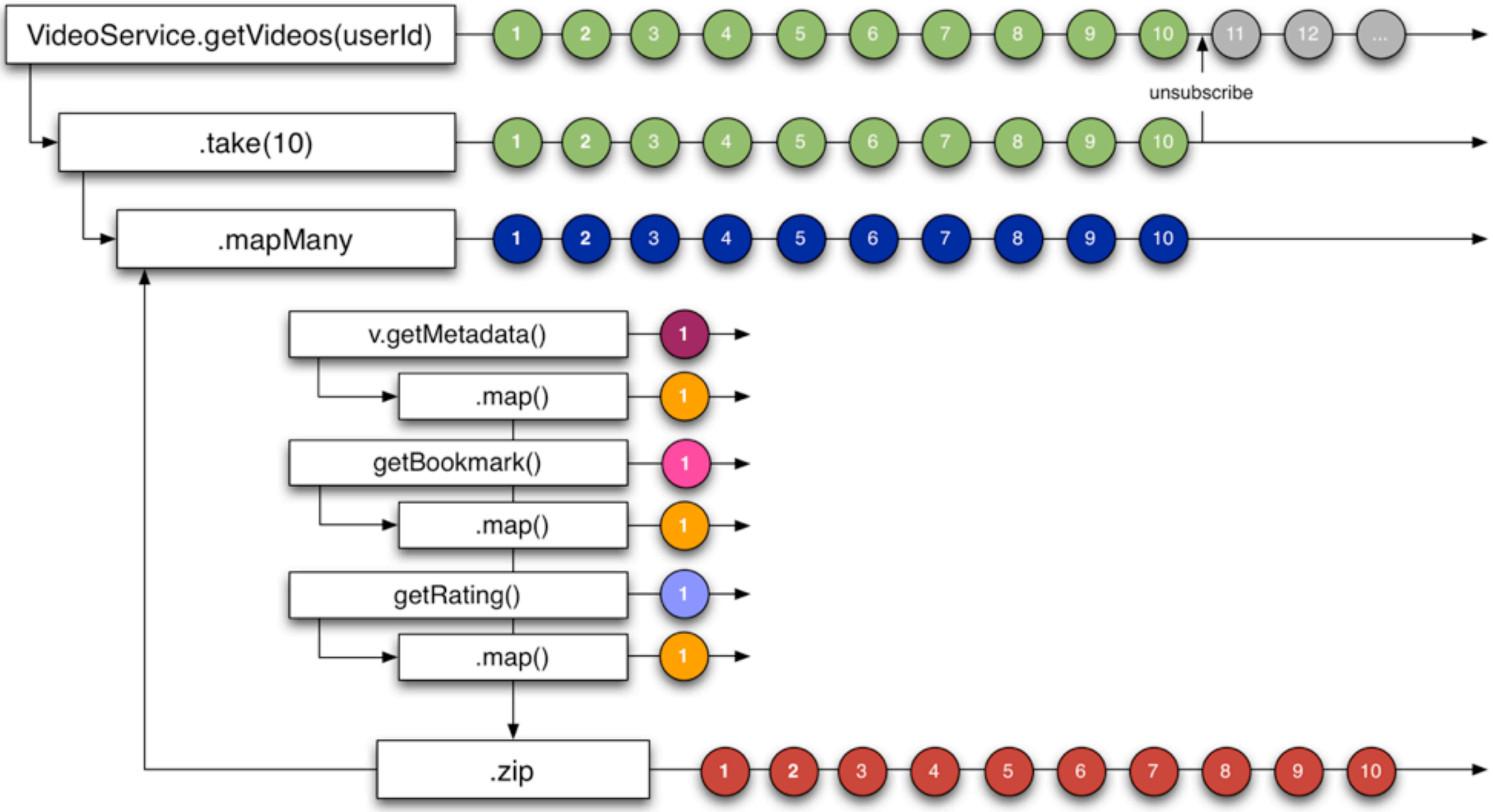
The ‘zip’ operator combines the 3 asynchronous Observables into 1



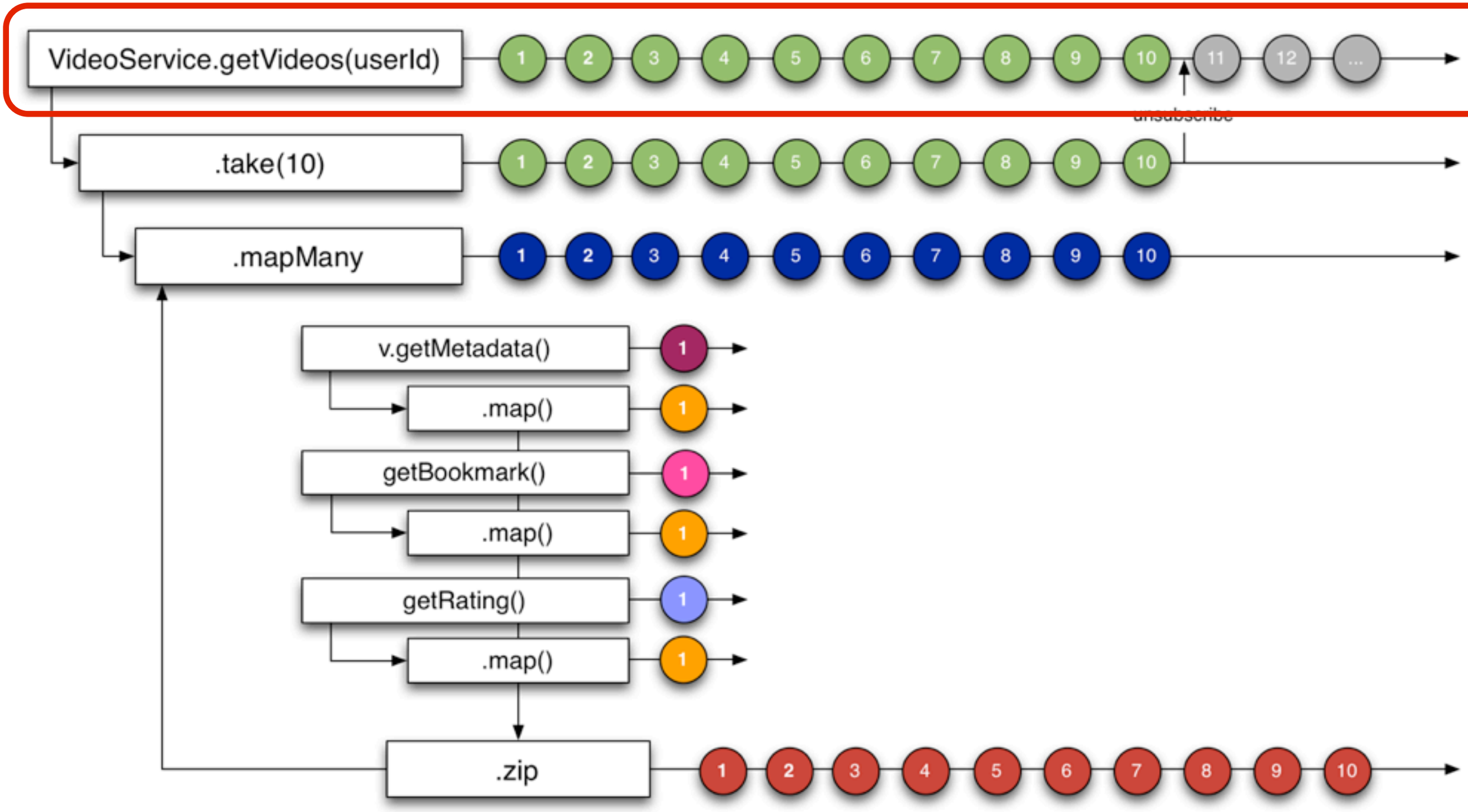

```
Observable.zip(a, b, { a, b, ->
    ... operate on values from both a & b ...
    return [a, b]; // i.e. return tuple
})
```

```
def Observable<Map> getVideos(userId) {
    return VideoService.getVideos(userId)
        // we only want the first 10 of each list
        .take(10)
        .mapMany({ Video video ->
          def m ...
          def b ...
          def r ...
          // compose these together
          return Observable.zip(m, b, r, {
                          metadata, bookmark, rating ->
            // now transform to complete dictionary
            // of data we want for each Video
              return [id: video.videoId]
                          << metadata << bookmark << rating
          })
    })
}
```

return a single Map (dictionary)
of transformed and combined data
from 4 asynchronous calls

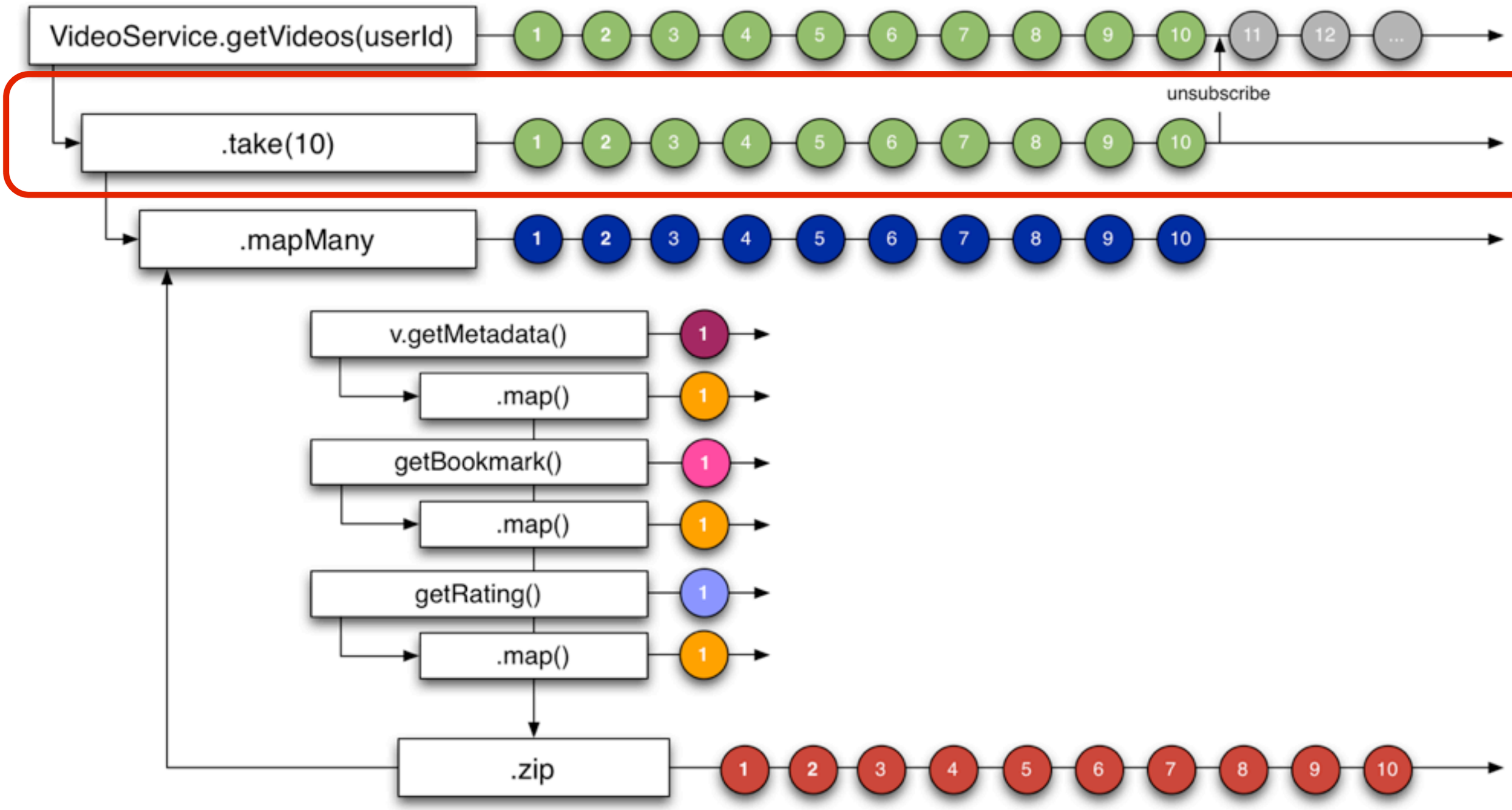


[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]



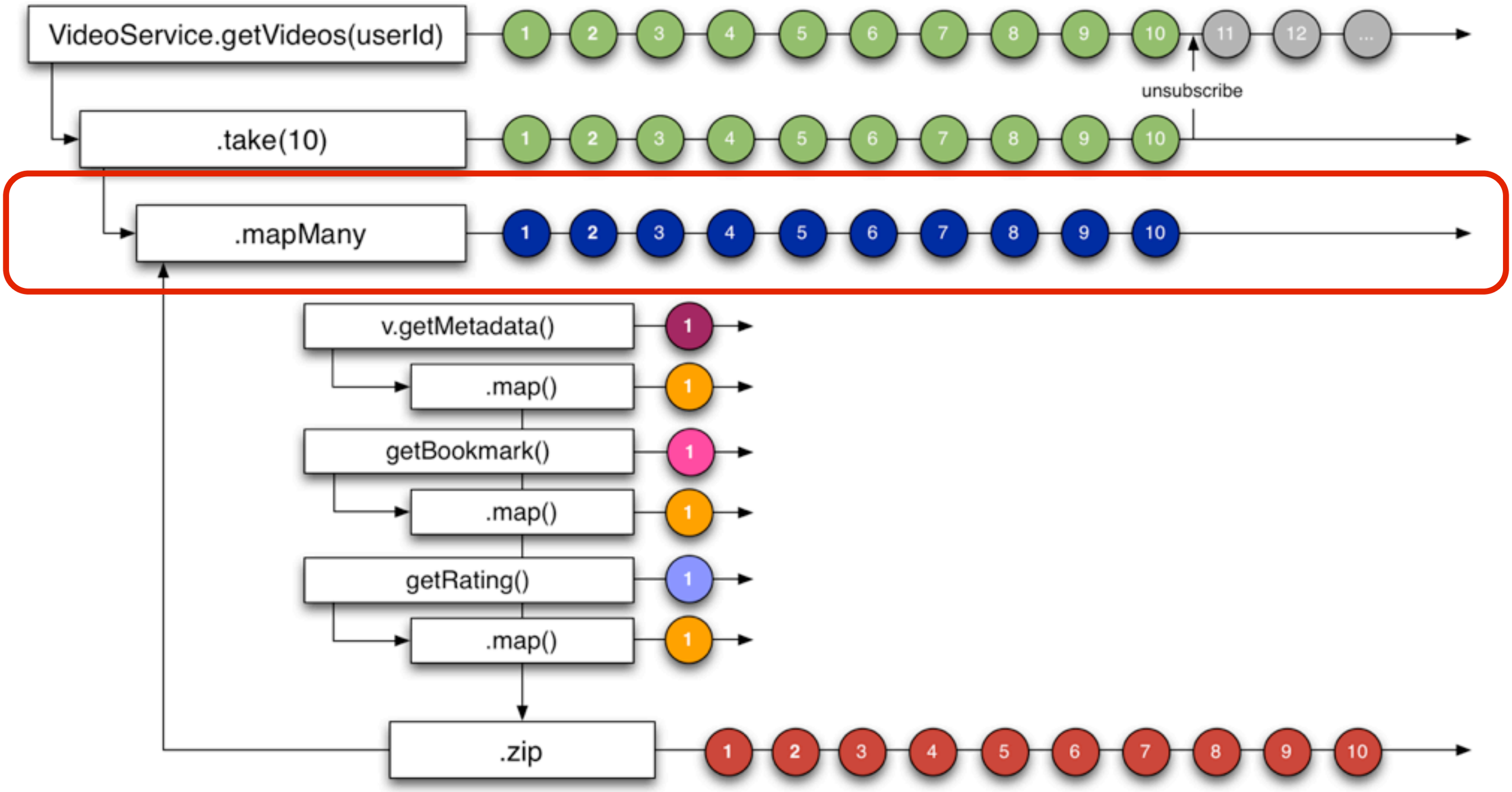
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

Observable<Video> emits *n* videos to onNext()



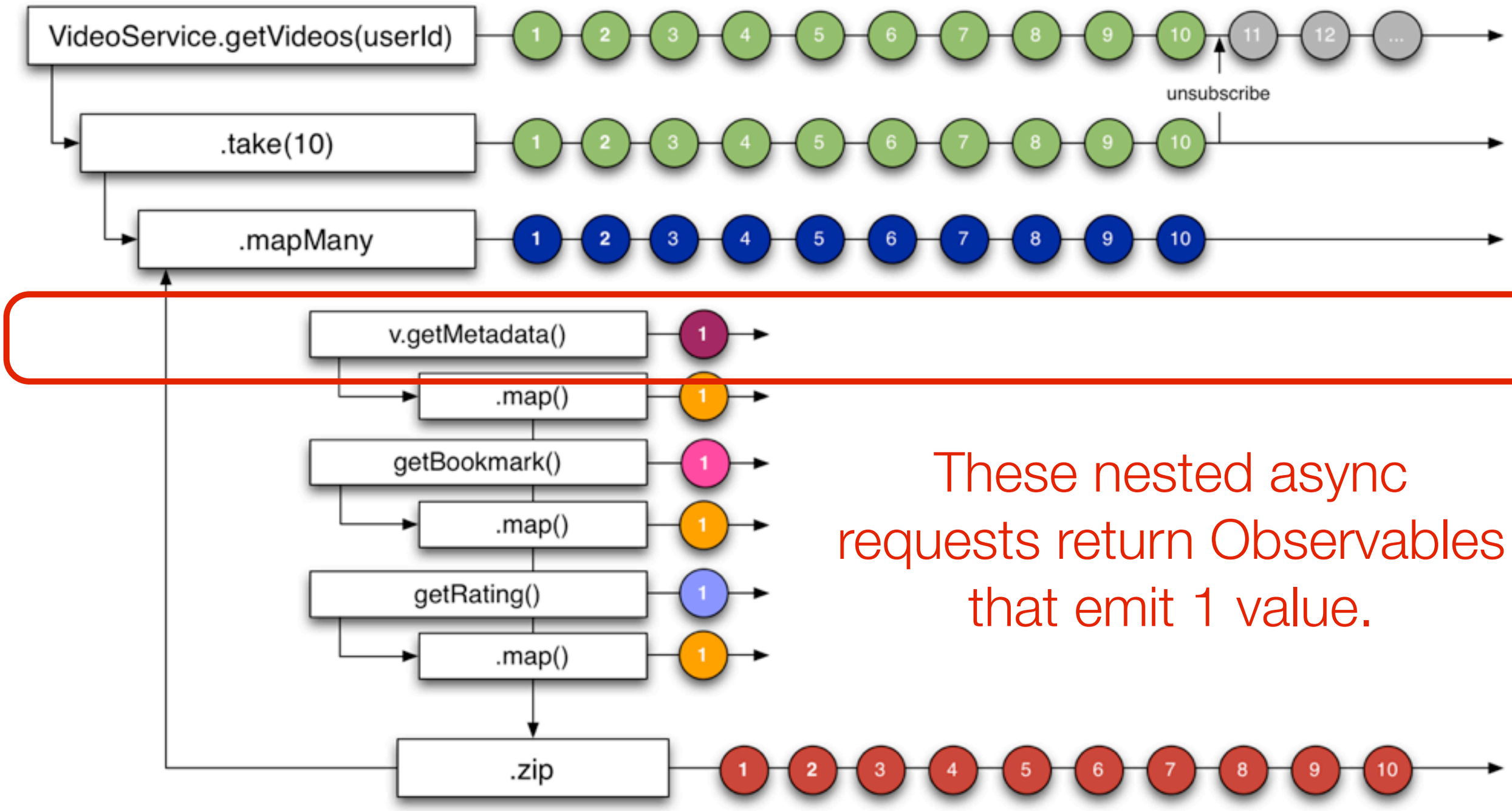
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

Takes first 10 then unsubscribes from origin.
Returns Observable<Video> that emits 10 Videos.



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

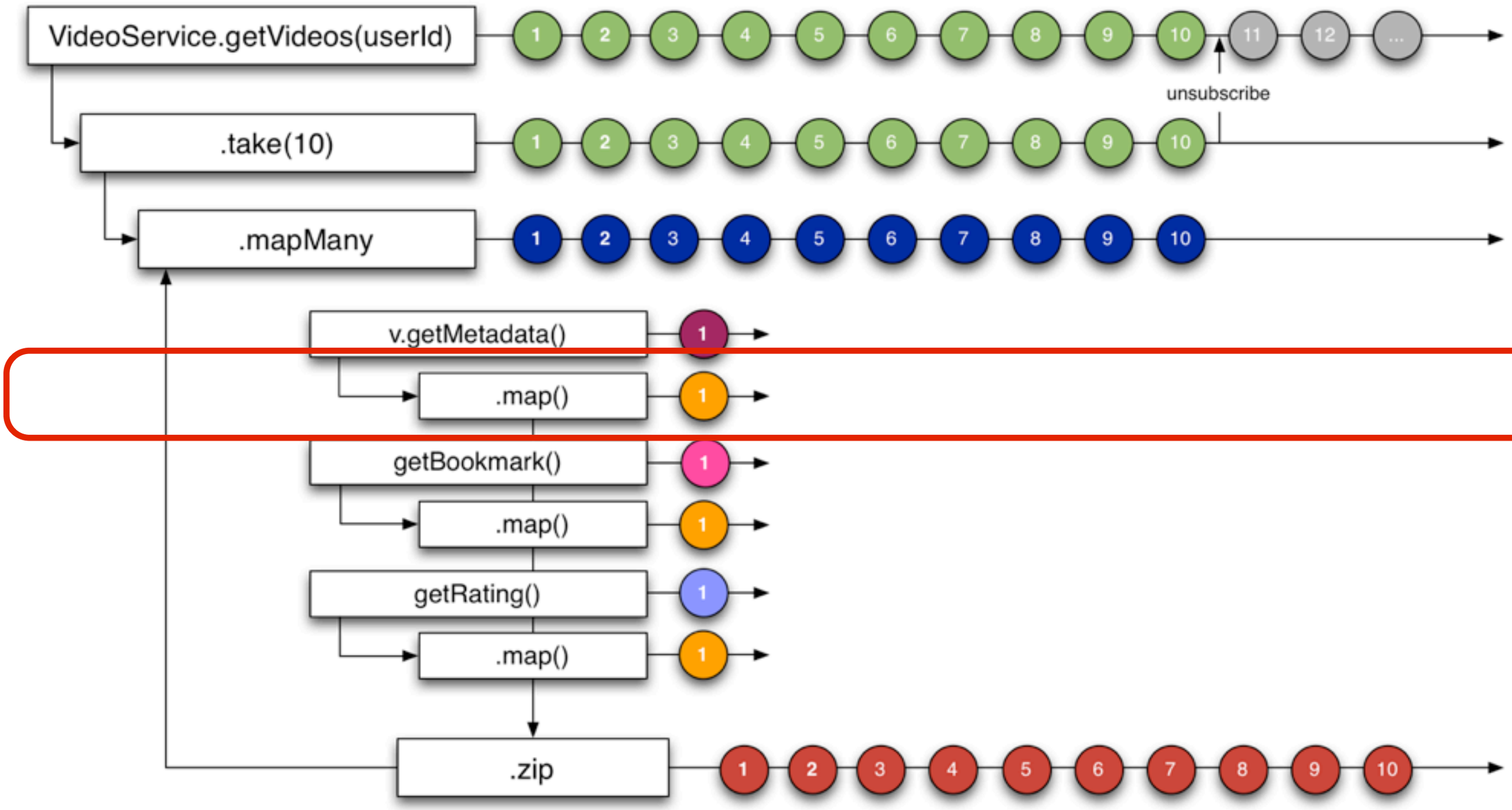
For each of the 10 Video objects it transforms via ‘mapMany’ function that does nested async calls.



These nested async requests return Observables that emit 1 value.

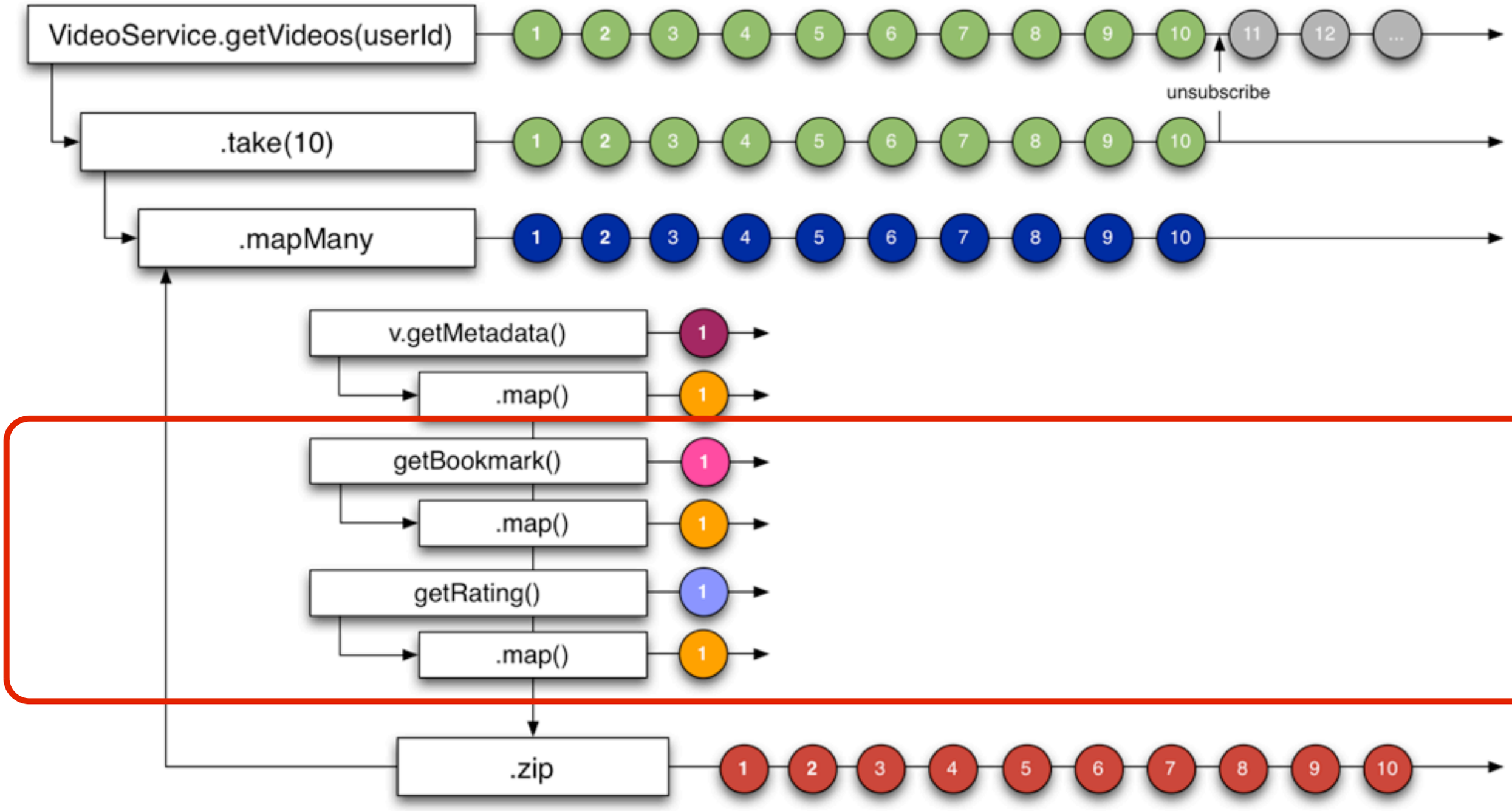
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

For each Video ‘v’ it calls getMetadata() which returns Observable<VideoMetadata>



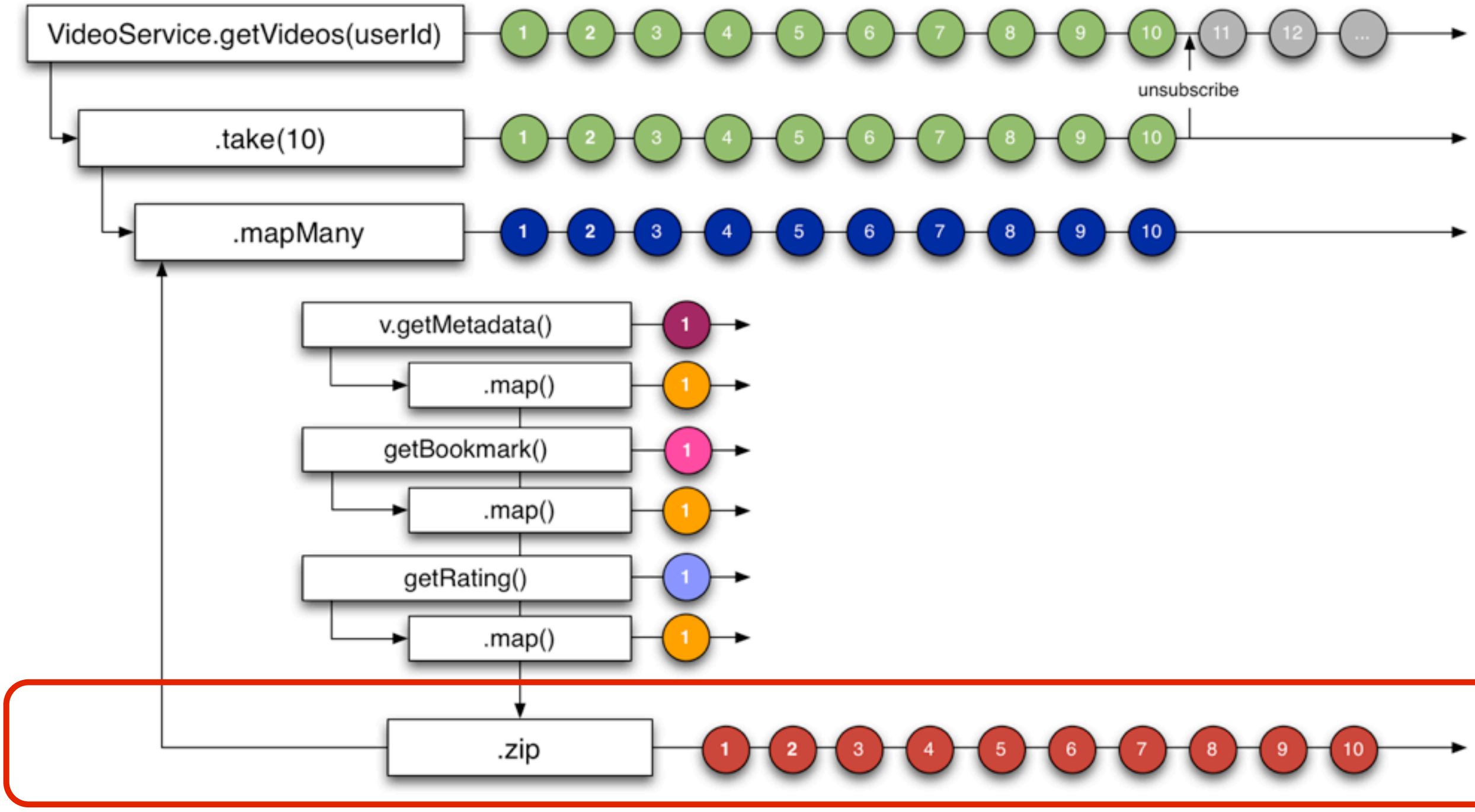
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

The Observable<VideoMetadata> is transformed via a ‘map’ function to return a Map of key/values.



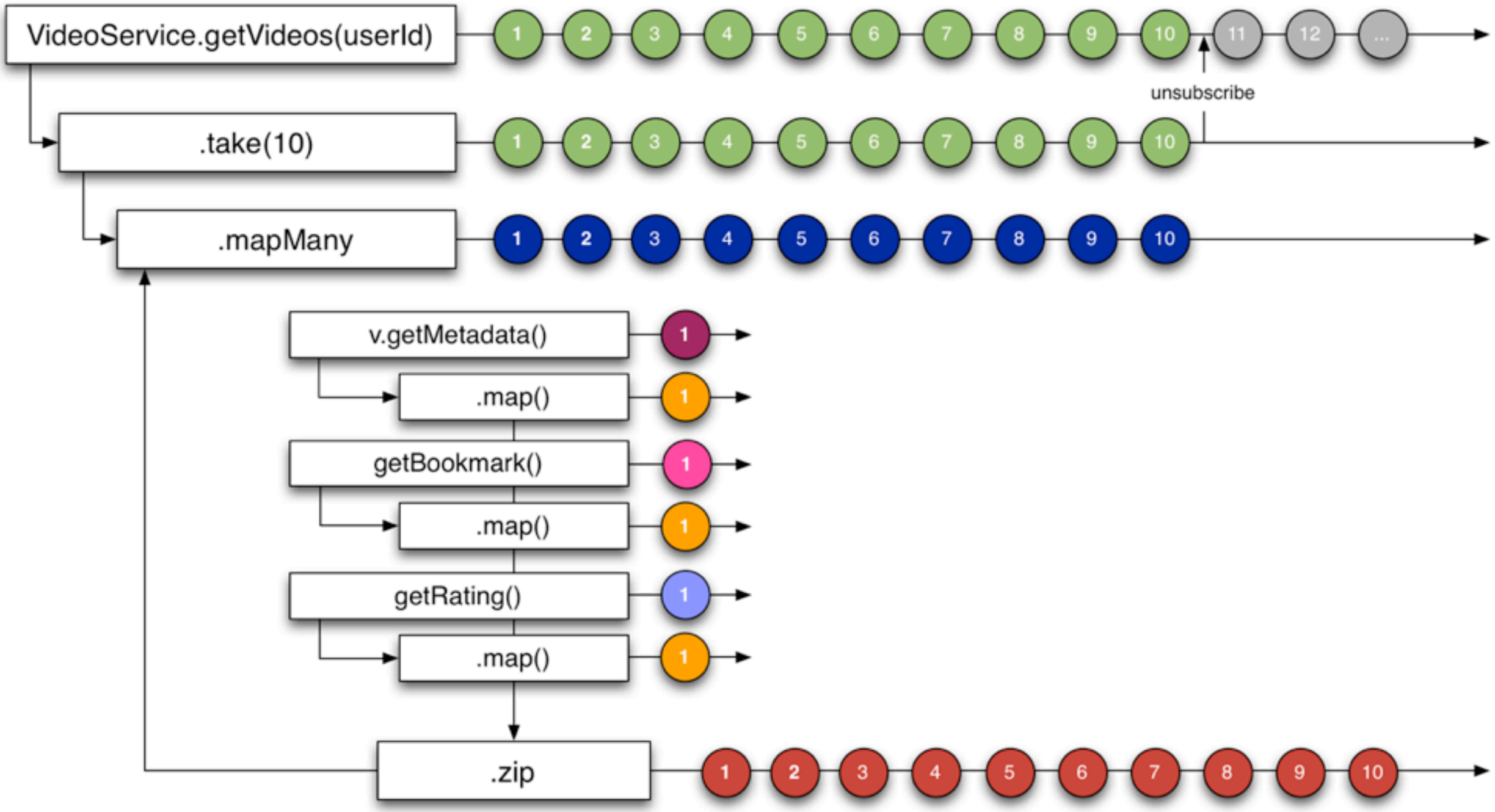
[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

Same for Observable<VideoBookmark> and Observable<VideoRating>



[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]

The 3 ‘mapped’ Observables are combined with a ‘zip’ function that emits a Map with all data.

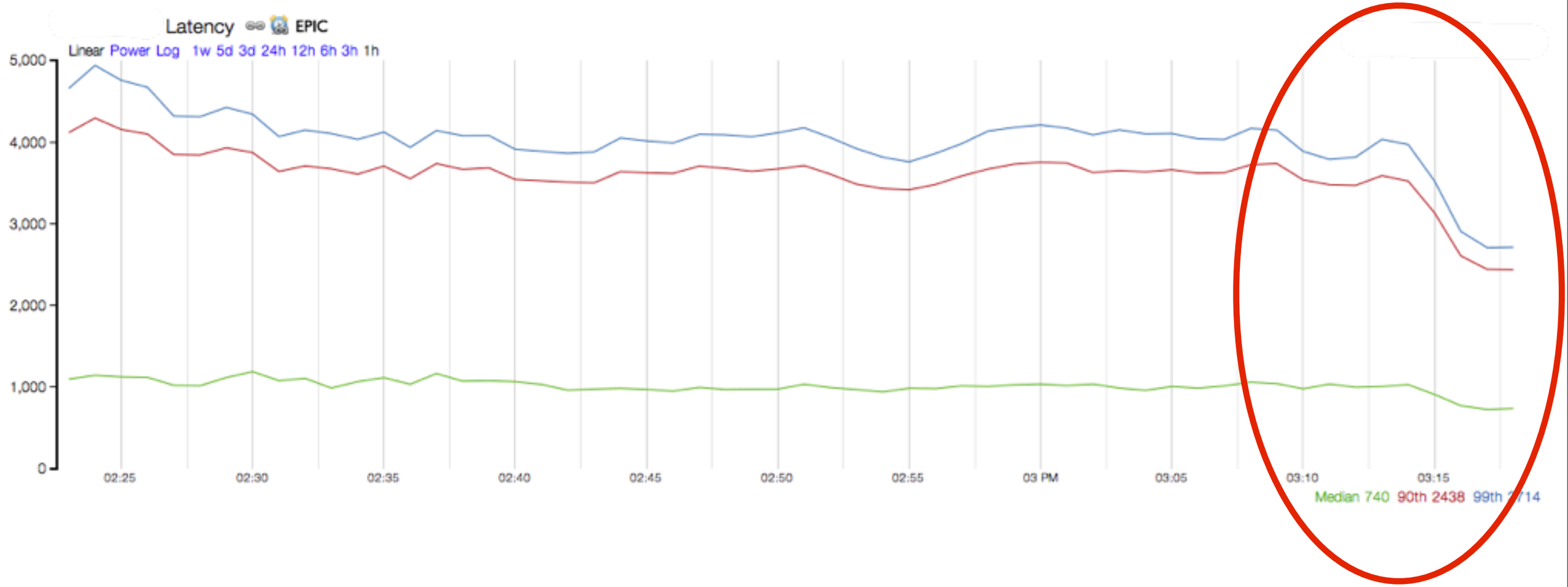


[id:1000, title:video-1000-title, length:5428, bookmark:0, rating:[actual:4, average:3, predicted:0]]]

The full sequence emits Observable<Map> that emits a Map for each of 10 Videos.

Client code treats all interactions
with the API as asynchronous

The API implementation chooses
whether something is
blocking or non-blocking
and
what resources it uses.



Example of latency reduction achieved by increasing number of threads used by Observables.

+

```
Observable<User> u = new GetUserCommand(id).observe();
Observable<Geo> g = new GetGeoCommand(request).observe();

Observable.zip(u, g, {user, geo ->
        return [username: user.getUsername(),
                currentLocation: geo.getCounty()]
})
```

RxJava coming to Hystrix

https://github.com/Netflix/Hystrix

To get started ...

```
<dependency>
    <groupId>com.netflix.rxjava</groupId>
    <artifactId>rxjava-core</artifactId>
    <version>x.y.z</version>
</dependency>
```

```
<dependency org="com.netflix.rxjava" name="rxjava-core" rev="x.y.z" />
```

... or for a different language ...

```
<dependency org="com.netflix.rxjava" name="rxjava-groovy" rev="x.y.z" />
<dependency org="com.netflix.rxjava" name="rxjava-clojure" rev="x.y.z" />
<dependency org="com.netflix.rxjava" name="rxjava-scala" rev="x.y.z" />
<dependency org="com.netflix.rxjava" name="rxjava-jruby" rev="x.y.z" />
```

Netflix is Hiring

http://jobs.netflix.com

Functional Reactive in the Netflix API with RxJava

http://techblog.netflix.com/2013/02/rxjava-netflix-api.html

Optimizing the Netflix API

http://techblog.netflix.com/2013/01/optimizing-netflix-api.html

RxJava

https://github.com/Netflix/RxJava

@RxJava

Ben Christensen

@benjchristensen

http://www.linkedin.com/in/benjchristensen