Reactive Programming with Rx

QConSF - November 2014

Ben Christensen
Developer – Edge Engineering at Netflix
@benjchristensen

NETFLIX | OSS
http://techblog.netflix.com/

NETFLIX

ReactiveX

An API for asynchronous programming with observable streams

RxJava

http://github.com/ReactiveX/RxJava

http://reactivex.io

Maven Central: 'io.reactivex:rxjava:1.0.+'

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

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

```java
// Iterable<String> or Stream<String>
// that contains 75 Strings
getDataFromLocalMemory()
 .skip(10)
 .limit(5)
 .map(s -> s + "_transformed")
 .forEach(t -> System.out.println("onNext => " + t))
```

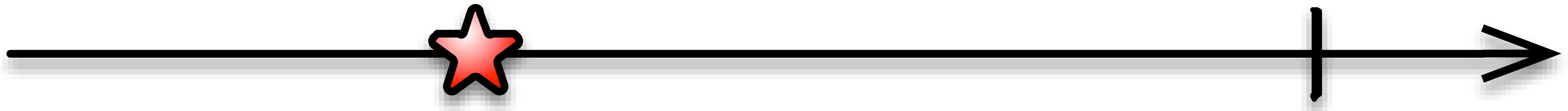
```java
// Observable<String>
// that emits 75 Strings
getDataFromNetwork()
 .skip(10)
 .take(5)
 .map(s -> s + "_transformed")
 .forEach(t -> System.out.println("onNext => " + t))
```

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

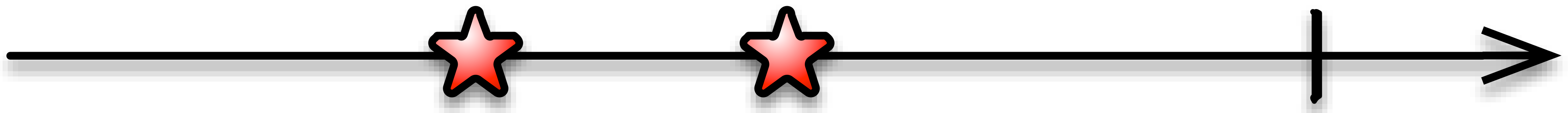
```
// Iterable<String> or Stream<String>
// that contains 75 Strings
getDataFromLocalMemory()
 .skip(10)
 .limit(5)
 .map(s -> s + "_transformed")
 .forEach(t -> System.out.println("onNext => " + t))
```

```
// Observable<String>
// that emits 75 Strings
getDataFromNetwork()
 .skip(10)
 .take(5)
 .map(s -> s + "_transformed")
 .forEach(t -> System.out.println("onNext => " + t))
```

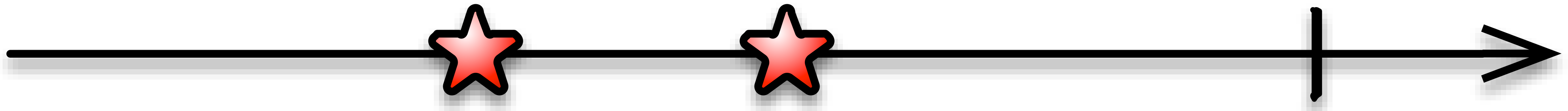
	Single	Multiple
Sync	**T** getData()	**Iterable**<T> getData() **Stream**<T> getData()
Async	**Future**<T> getData()	**Observable<T>** getData()



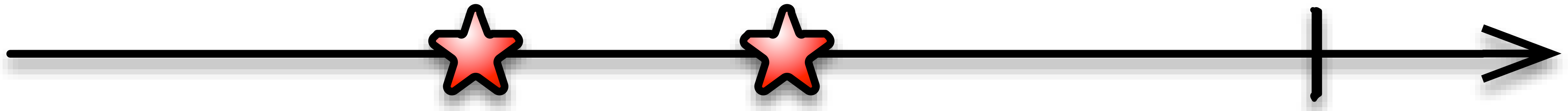
```
Observable.create(subscriber -> {
    subscriber.onNext("Hello World!");
    subscriber.onCompleted();
}).subscribe(System.out::println);
```



```
Observable.create(subscriber -> {
    subscriber.onNext("Hello");
    subscriber.onNext("World!");
    subscriber.onCompleted();
}).subscribe(System.out::println);
```

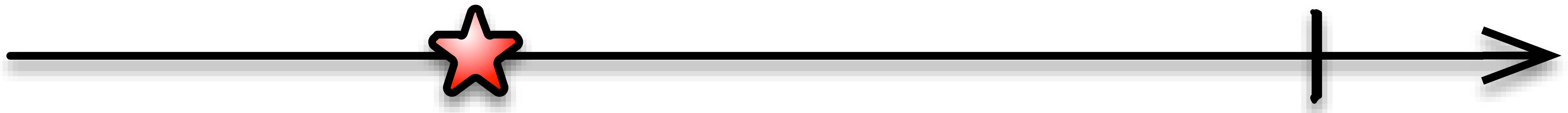


```
// shorten by using helper method
Observable.just(“Hello”, “World!”)
          .subscribe(System.out::println);
```

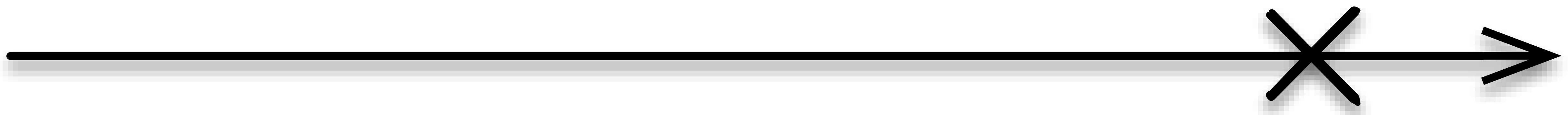


```
// add onError and onComplete listeners
Observable.just(“Hello”, “World!”)
         .subscribe(System.out::println,
                  Throwable::printStackTrace,
                  () -> System.out.println("Done"));
```

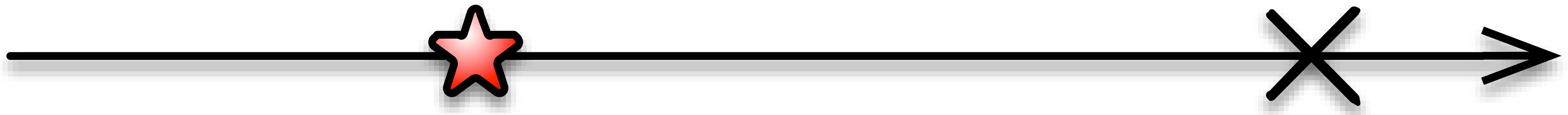
```java
// expand to show full classes
Observable.create(new OnSubscribe<String>() {

    @Override
    public void call(Subscriber<? super String> subscriber) {
        subscriber.onNext("Hello World!");
        subscriber.onCompleted();
    }

}).subscribe(new Subscriber<String>() {

    @Override
    public void onCompleted() {
        System.out.println("Done");
    }

    @Override
    public void onError(Throwable e) {
        e.printStackTrace();
    }

    @Override
    public void onNext(String t) {
        System.out.println(t);
    }

});
```



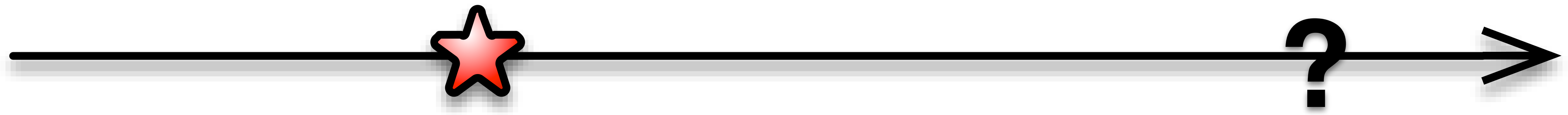
```
// add error propagation
Observable.create(subscriber -> {
    try {
        subscriber.onNext("Hello World!");
        subscriber.onCompleted();
    } catch (Exception e) {
        subscriber.onError(e);
    }
}).subscribe(System.out::println);
```



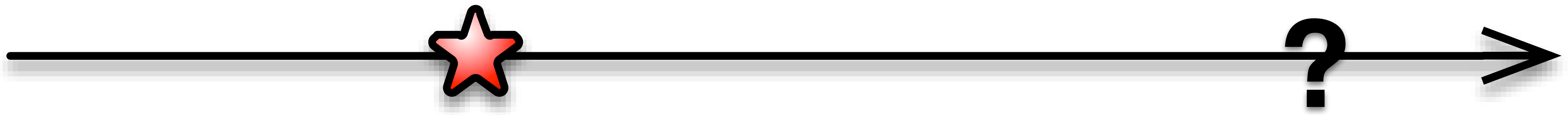
```
// add error propagation
Observable.create(subscriber -> {
    try {
        subscriber.onNext(throwException());
        subscriber.onCompleted();
    } catch (Exception e) {
        subscriber.onError(e);
    }
}).subscribe(System.out::println);
```



```
// add error propagation
Observable.create(subscriber -> {
    try {
        subscriber.onNext("Hello World!");
        subscriber.onNext(throwException());
        subscriber.onCompleted();
    } catch (Exception e) {
        subscriber.onError(e);
    }
}).subscribe(System.out::println);
```

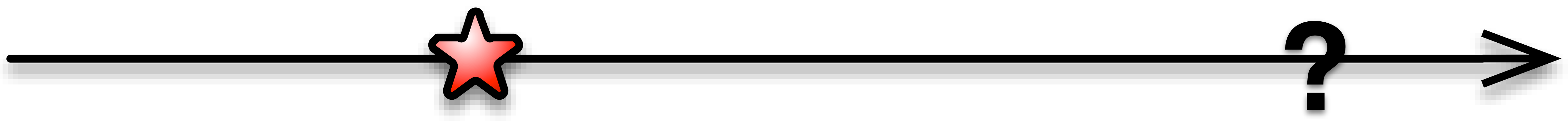


```
// add concurrency (manually)
Observable.create(subscriber -> {
    new Thread(() -> {
        try {
            subscriber.onNext(getData());
            subscriber.onCompleted();
        } catch (Exception e) {
            subscriber.onError(e);
        }
    }).start();
}).subscribe(System.out::println);
```

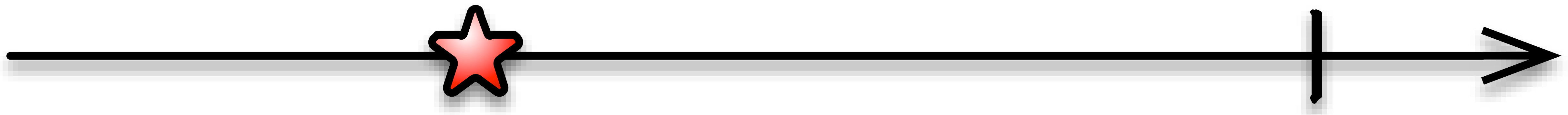


```
// add concurrency (using a Scheduler)
Observable.create(subscriber -> {
    try {
        subscriber.onNext(getData());
        subscriber.onCompleted();
    } catch (Exception e) {
        subscriber.onError(e);
    }
}).subscribeOn(Schedulers.io())
        .subscribe(System.out::println);
```

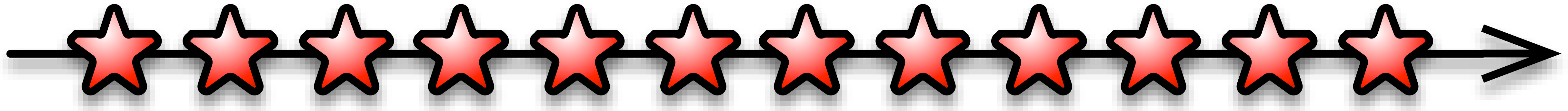
Learn more about parameterized concurrency and virtual time with
Rx Schedulers at https://github.com/ReactiveX/RxJava/wiki/Scheduler



```
// add operator
Observable.create(subscriber -> {
    try {
        subscriber.onNext(getData());
        subscriber.onCompleted();
    } catch (Exception e) {
        subscriber.onError(e);
    }
}).subscribeOn(Schedulers.io())
        .map(data -> data + " --> at " + System.currentTimeMillis())
        .subscribe(System.out::println);
```



```
// add error handling
Observable.create(subscriber -> {
    try {
        subscriber.onNext(getData());
        subscriber.onCompleted();
    } catch (Exception e) {
        subscriber.onError(e);
    }
}).subscribeOn(Schedulers.io())
        .map(data -> data + " --> at " + System.currentTimeMillis())
        .onErrorResumeNext(e -> Observable.just("Fallback Data"))
        .subscribe(System.out::println);
```



```
// infinite
Observable.create(subscriber -> {
    int i=0;
    while(!subscriber.isUnsubscribed()) {
        subscriber.onNext(i++);
    }
}).subscribe(System.out::println);
```

Note: No backpressure support here. See Observable.from(Iterable)
or Observable.range() for actual implementations

Hot

emits whether you’re ready or not

examples
mouse and keyboard events
system events
stock prices

```
Observable.create(subscriber -> {
  // register with data source
})
```

Cold

emits when requested
(generally at controlled rate)

examples
database query
web service request
reading file

```
Observable.create(subscriber -> {
  // fetch data
})
```

Hot

emits whether you’re ready or not

examples

mouse and keyboard events

system events

stock prices

```
Observable.create(subscriber -> {
  // register with data source
})
```

flow control

Cold

emits when requested
(generally at controlled rate)

examples

database query

web service request

reading file

```
Observable.create(subscriber -> {
  // fetch data
})
```

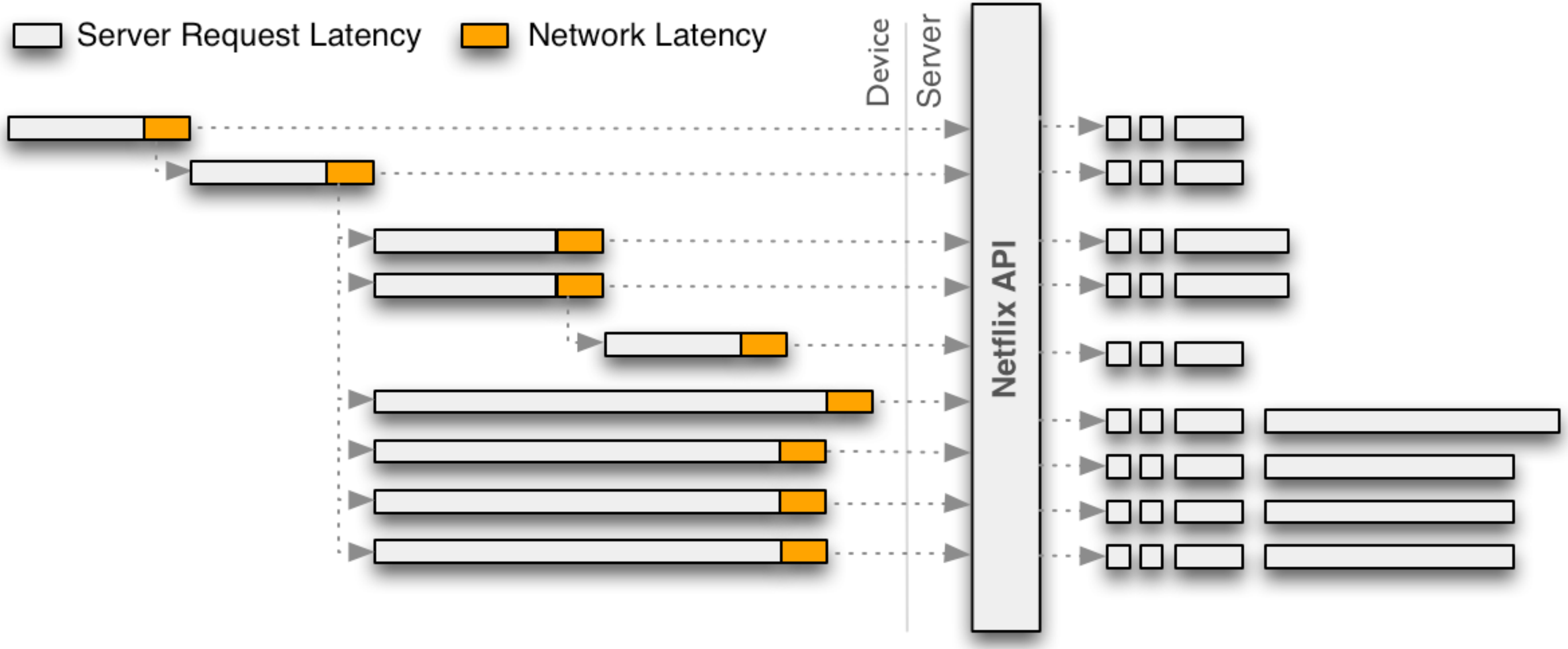
flow control & backpressure

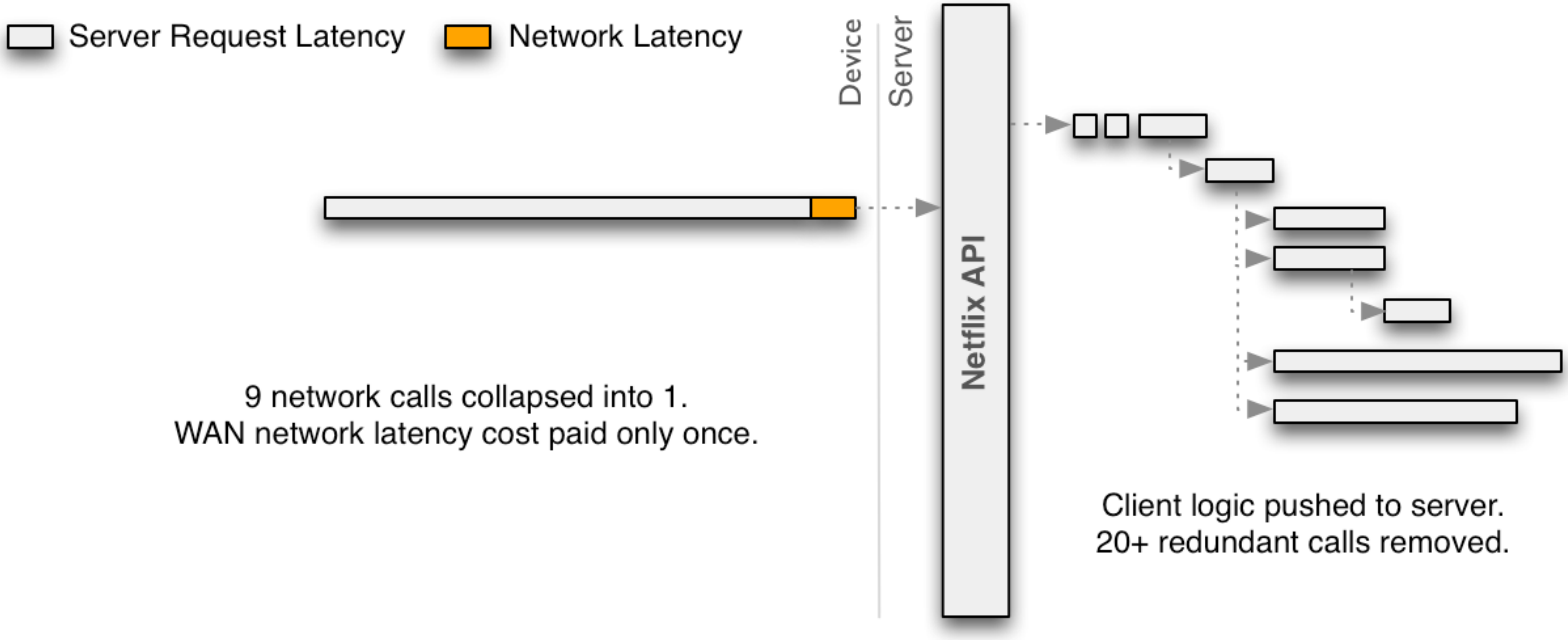
NETFLIX
Sign In

Watch TV shows & movies anytime, anywhere.

Plans from $7.99 a month.

Start Your Free Month

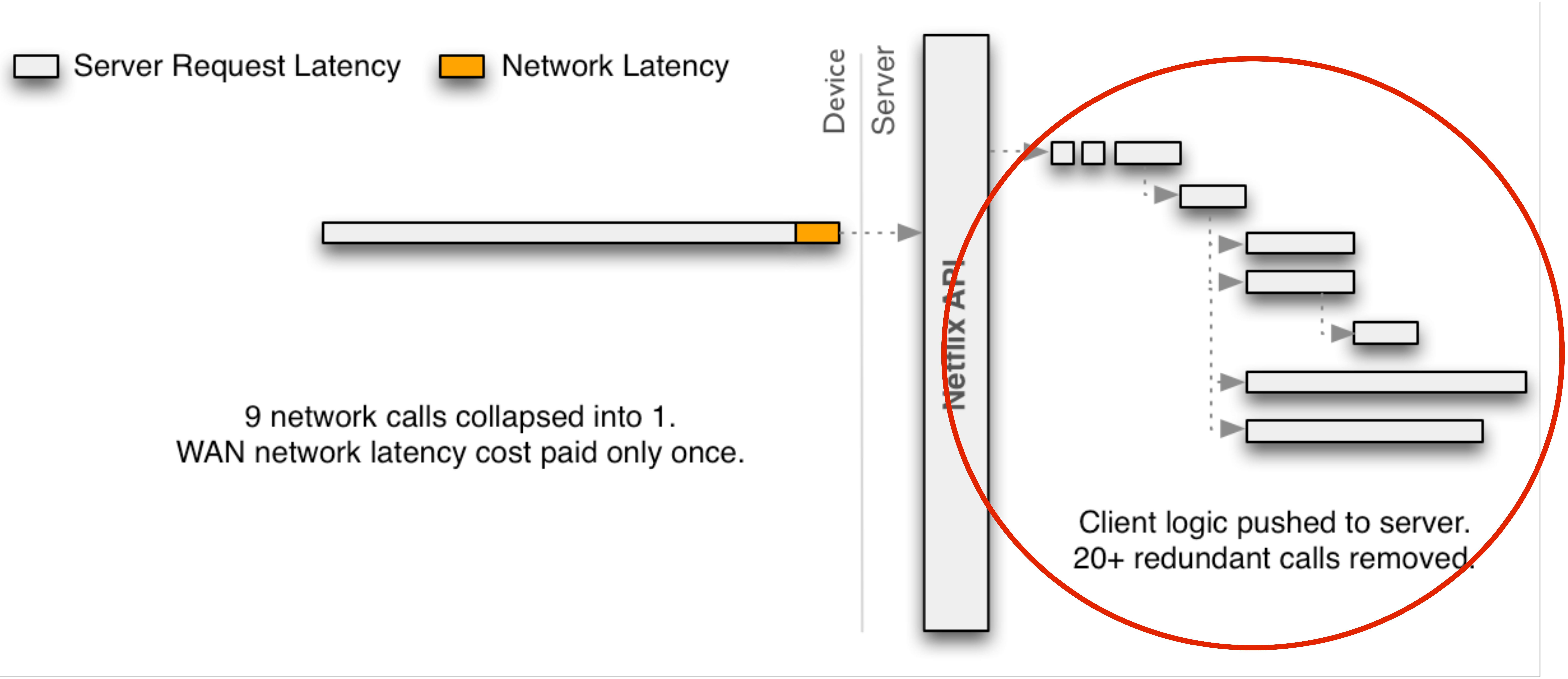

Server Request Latency
Network Latency
Device
Server
Netflix API



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

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

Abstract Concurrency



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

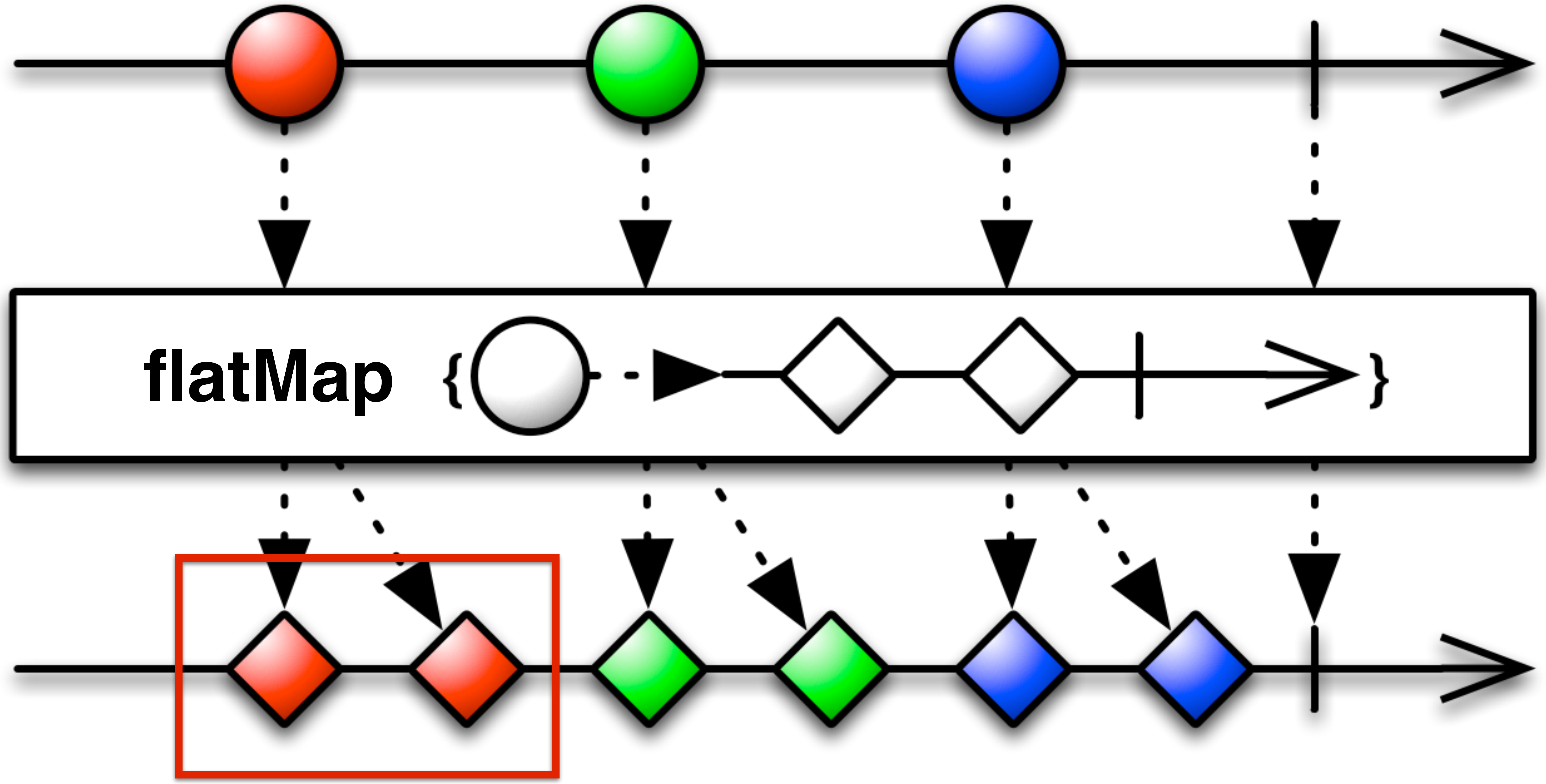
Client logic pushed to server.
20+ redundant calls removed.

Cold Finite Streams

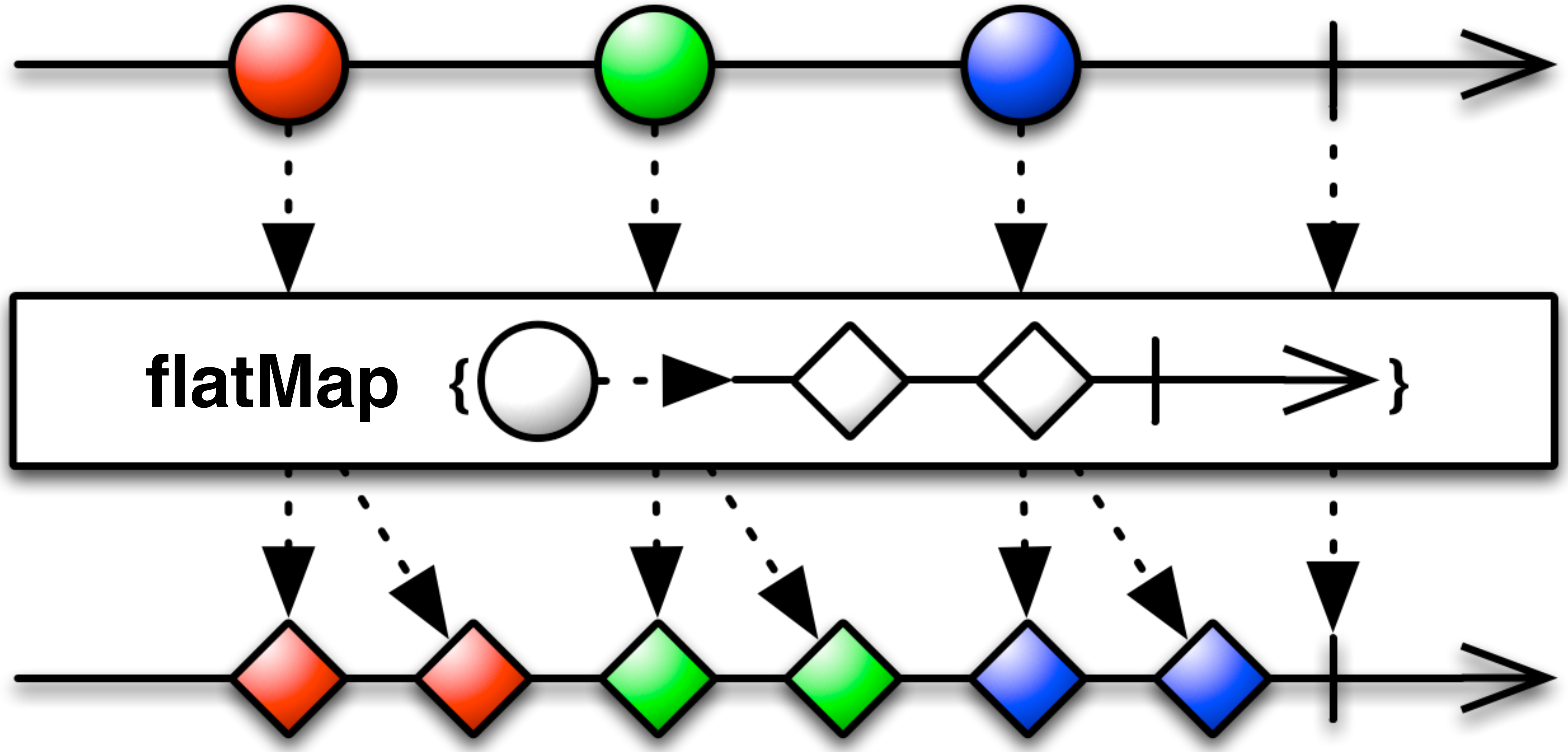
```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```java
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
    // first request User object
    return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
      // then fetch personal catalog
      Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
          .flatMap(catalogList -> {
            return catalogList.videos().<Map<String, Object>> flatMap(video -> {
              Observable<Bookmark> bookmark = getBookmark(video);
              Observable<Rating> rating = getRating(video);
              Observable<VideoMetadata> metadata = getMetadata(video);
              return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                return combineVideoData(video, b, r, m);
              });
            });
          });
      // and fetch social data in parallel
      Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
        return s.getDataAsMap();
      });
      // merge the results
      return Observable.merge(catalog, social);
    }).flatMap(data -> {
      // output as SSE as we get back the data (no waiting until all is done)
      return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
    });
  }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```



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



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

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```java
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```java
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

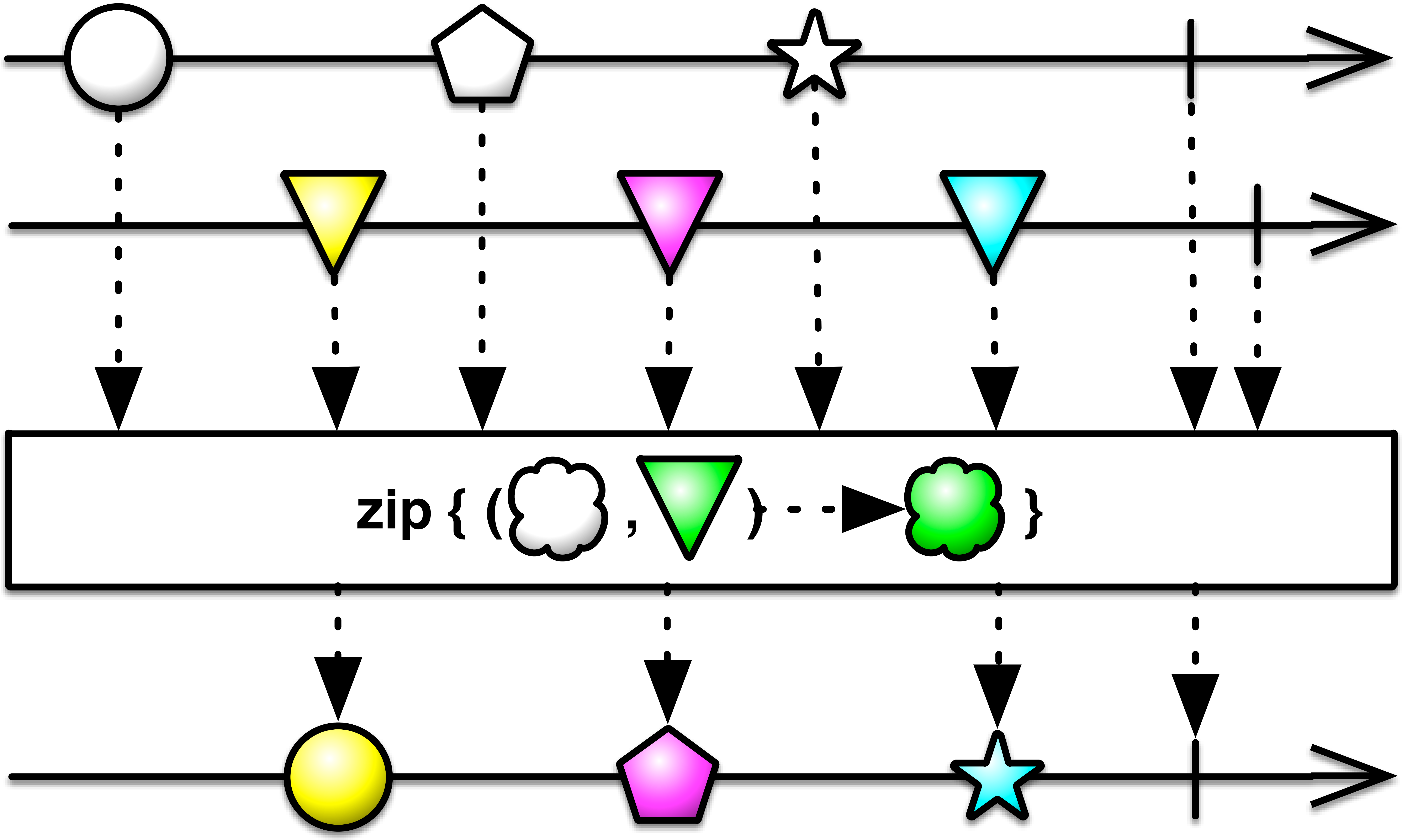
```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

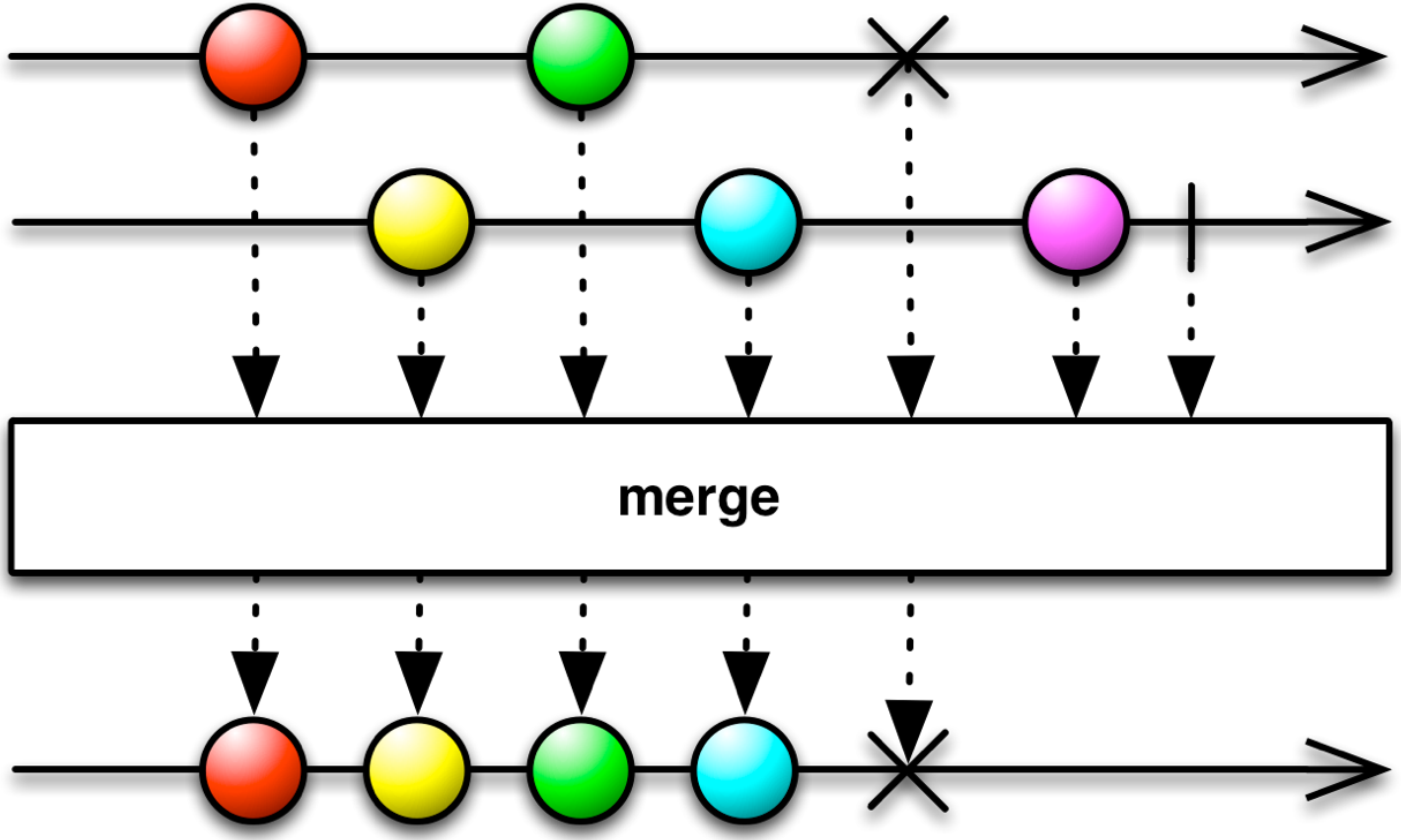
```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```



```
Observable.zip(a, b, (a, b) -> {
  ... operate on values from both a & b ...
  return Arrays.asList(a, b);
})
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```


merge

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

INSTEAD OF A BLOCKING API …

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

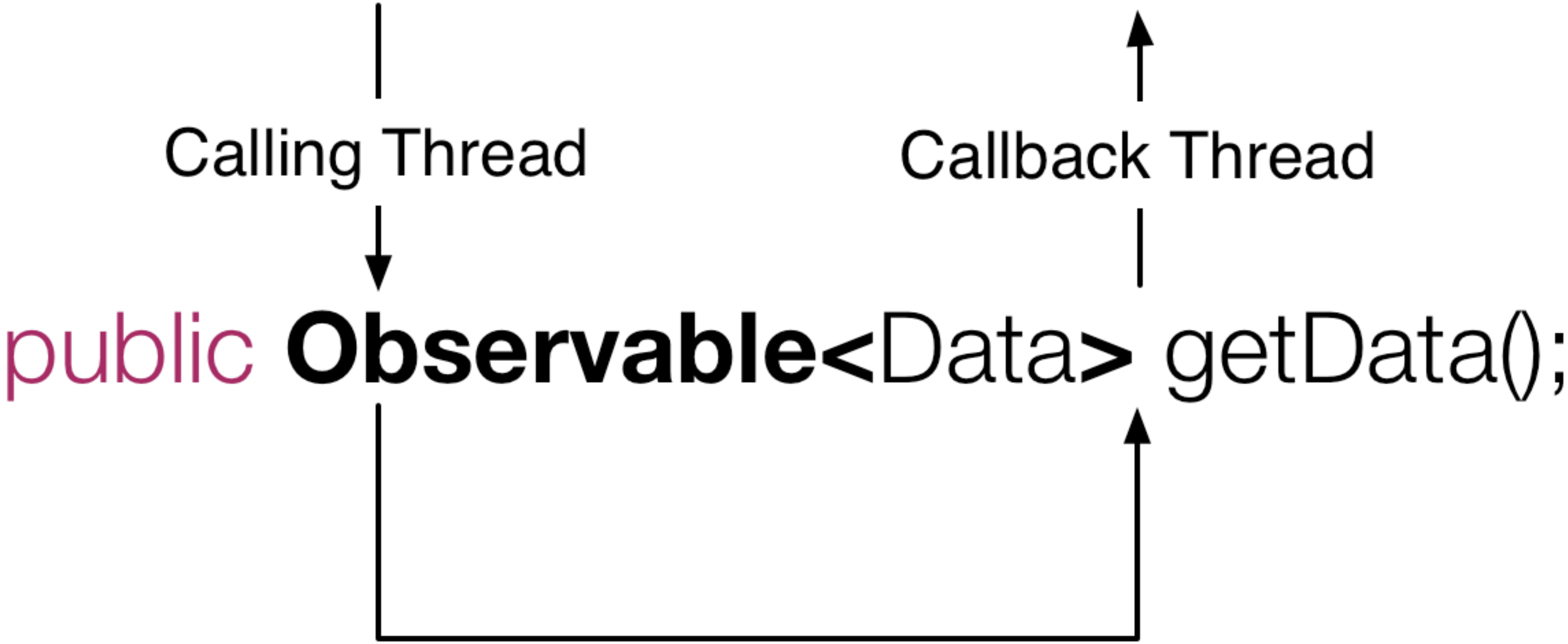
… CREATE AN OBSERVABLE API:

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

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
    // first request User object
    return getUser( 1 uest.getQueryParameters().get("userId")).flatMap(user -> {
      // then fetch personal catalog
      Observable<Map<String, Object>> catalog = getPersonalizedCatalog( 2 r)
          .flatMap(catalogList -> {
            return catalogList.videos().<Map<String, Object>> flatMap(video -> {
              Observable<Bookmark> bookmark = getBookmark( 3 eo);
              Observable<Rating> rating = getRating( 4 eo);
              Observable<VideoMetadata> metadata = getMetadata( 5 eo);
              return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                return combineVideoData(video, b, r, m);
              });
            });
          });
      // and fetch social data in parallel
      Observable<Map<String, Object>> social = getSocialData( 6 r).map(s -> {
        return s.getDataAsMap();
      });
      // merge the results
      return Observable.merge(catalog, social);
    }).flatMap(data -> {
      // output as SSE as we get back the data (no waiting until all is done)
      return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
    });
  }
```

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser( 1 uest.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog( 2 r)
                     .flatMap(catalogList -> {
                              n catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark( 3 eo);
                             Observable<Rating> rating = getRating( 4 eo);
                             Observable<VideoMetadata> metadata = getMetadata( 5 eo);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData( 6 r).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```

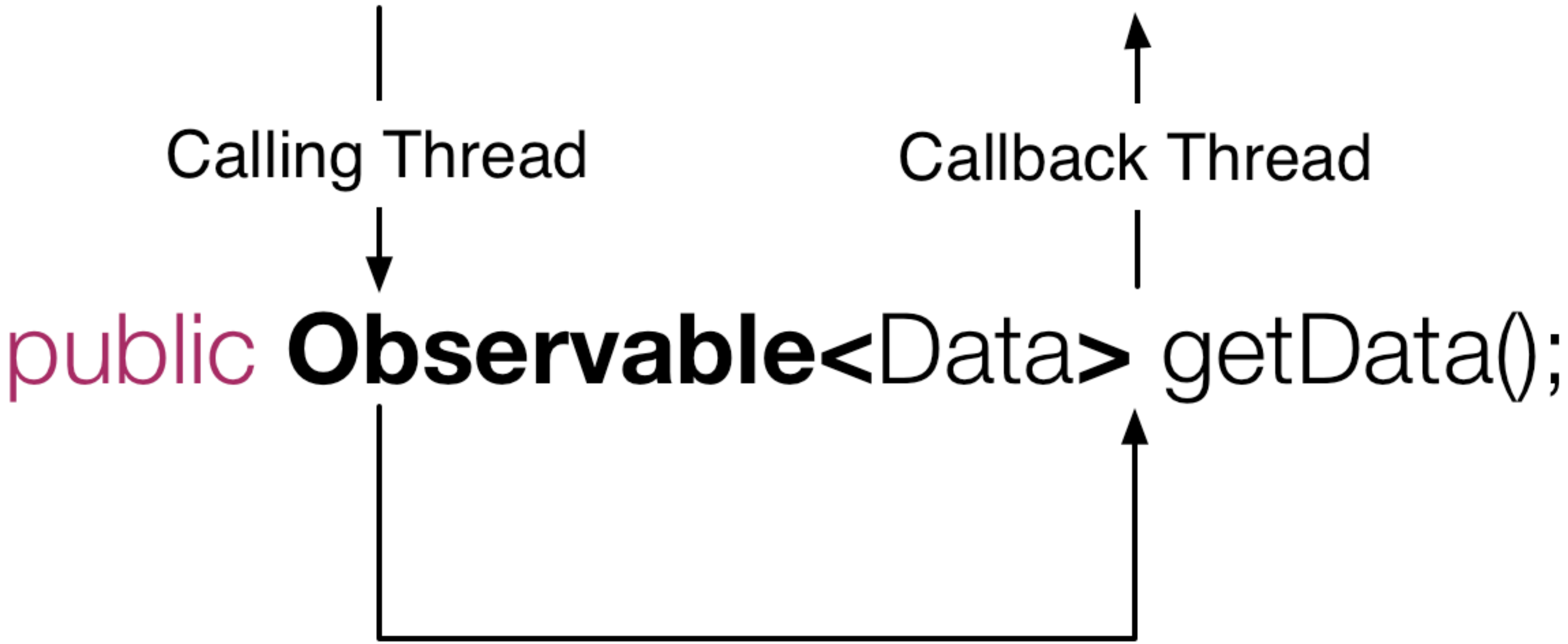
Non-Opinionated Concurrency



Calling Thread

Callback Thread

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



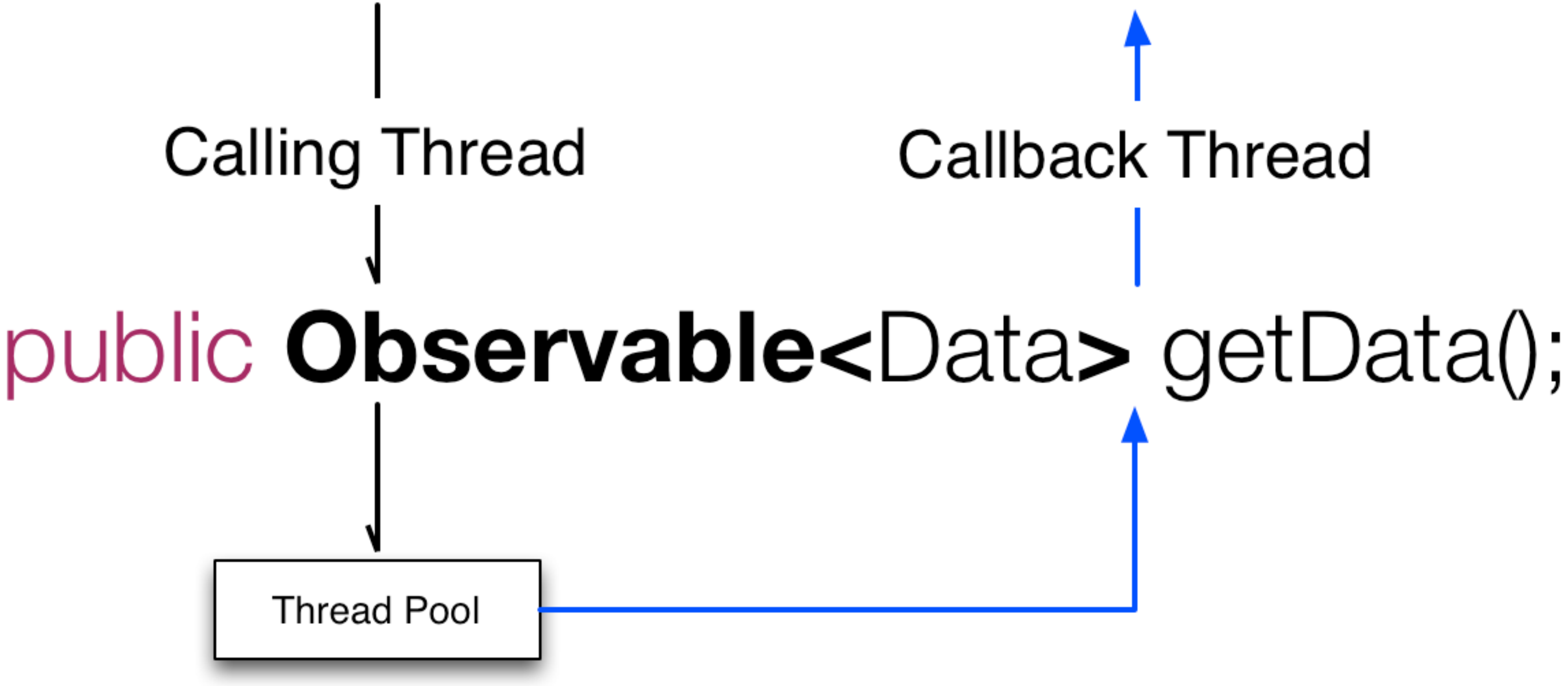
Calling Thread

Callback Thread

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

Do work synchronously on calling thread.

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
    }
```



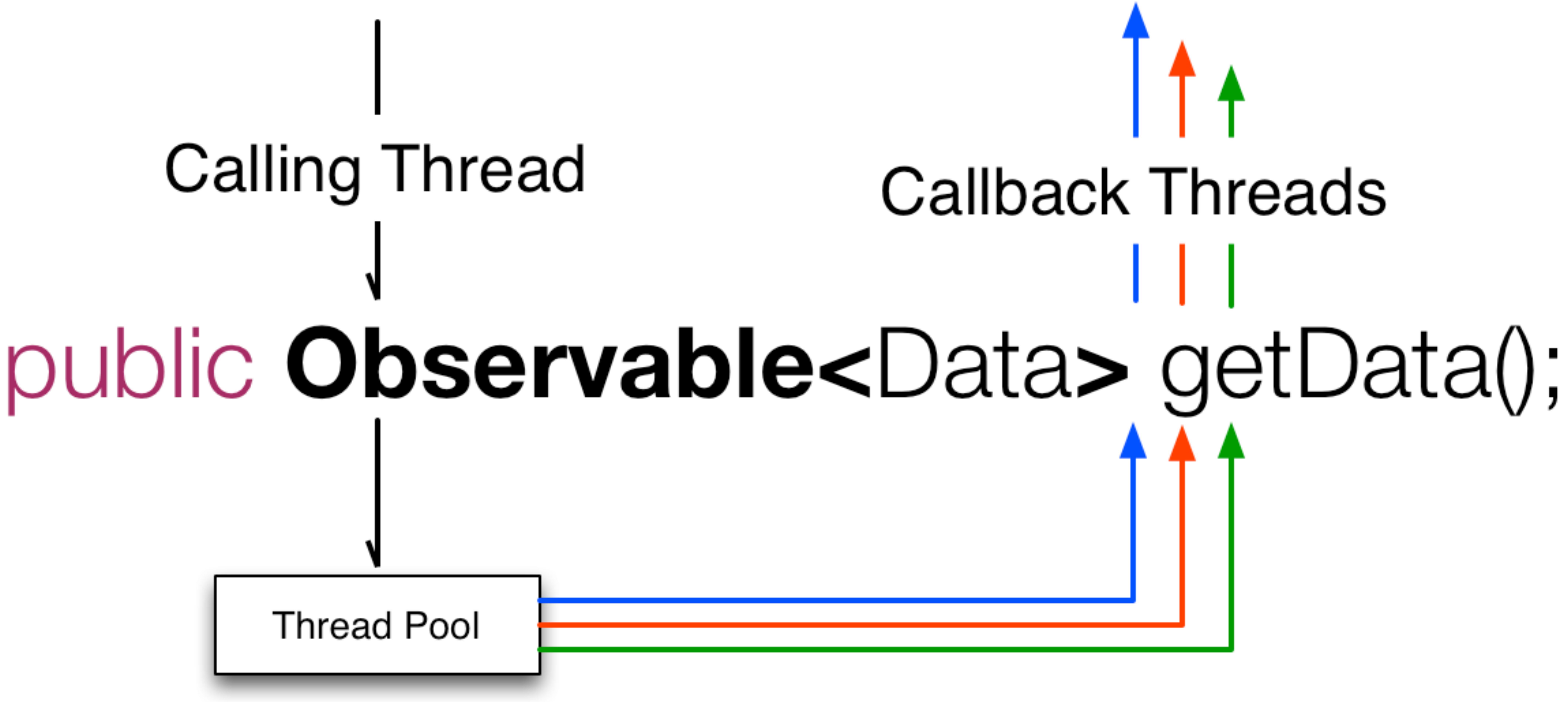
Calling Thread

Callback Thread

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

Thread Pool

Do work asynchronously on a separate thread.



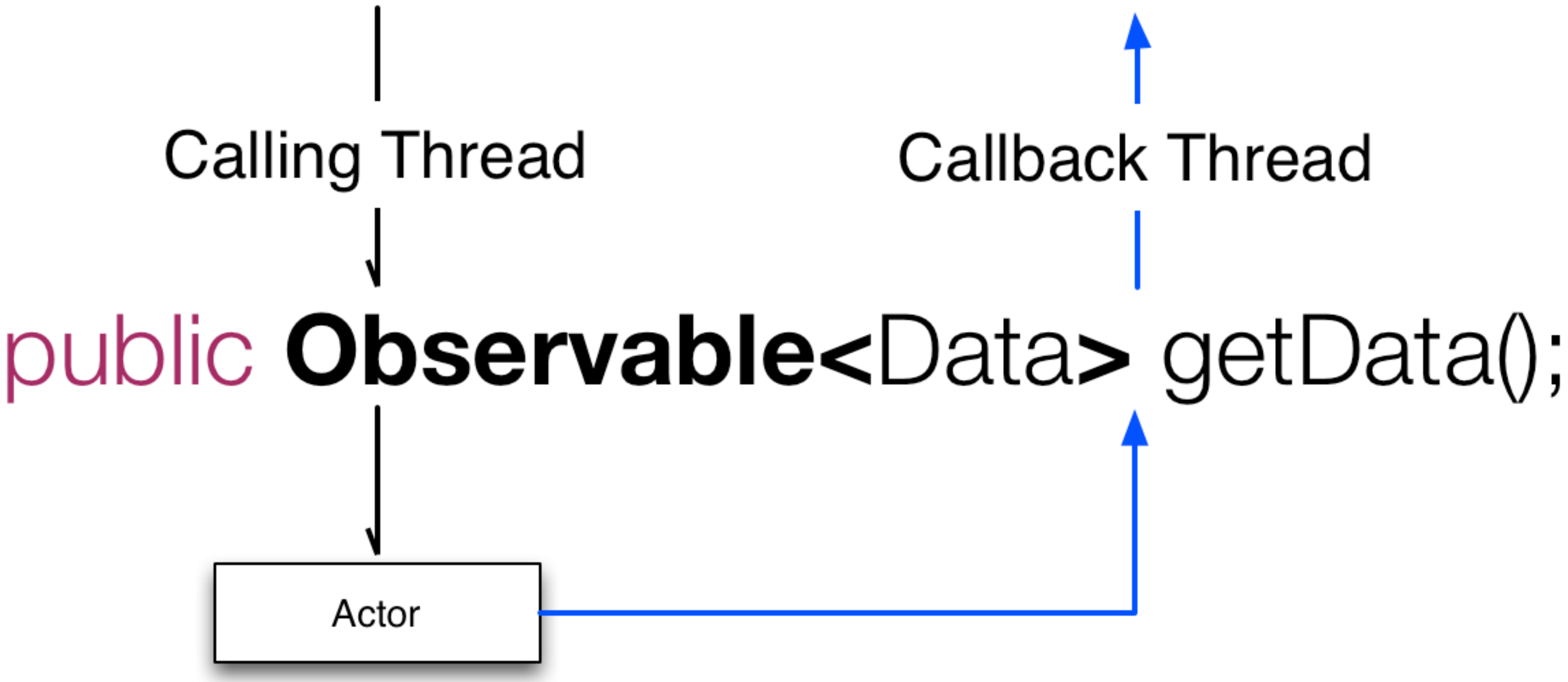
Calling Thread

Callback Threads

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

Thread Pool

Do work asynchronously on a multiple threads.



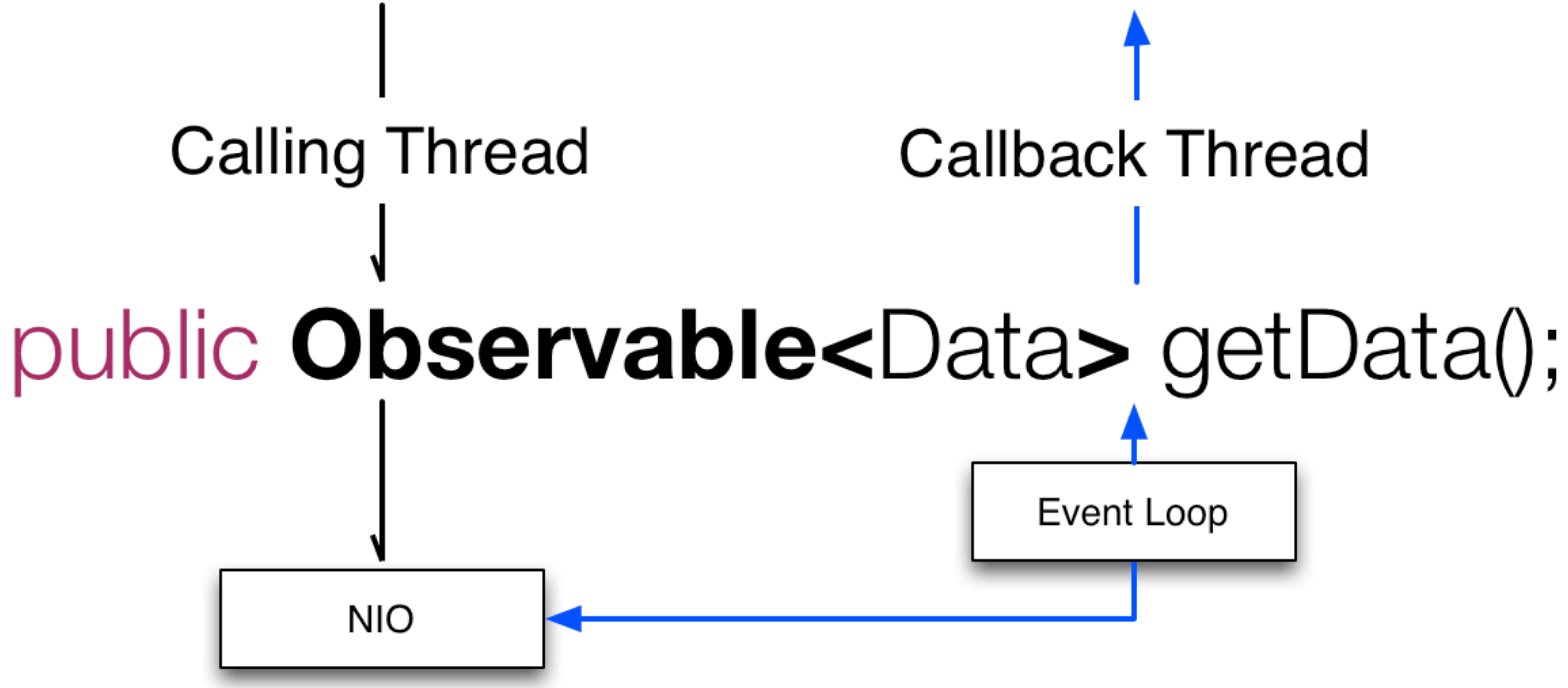
Calling Thread

Callback Thread

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

Actor

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



Calling Thread

Callback Thread

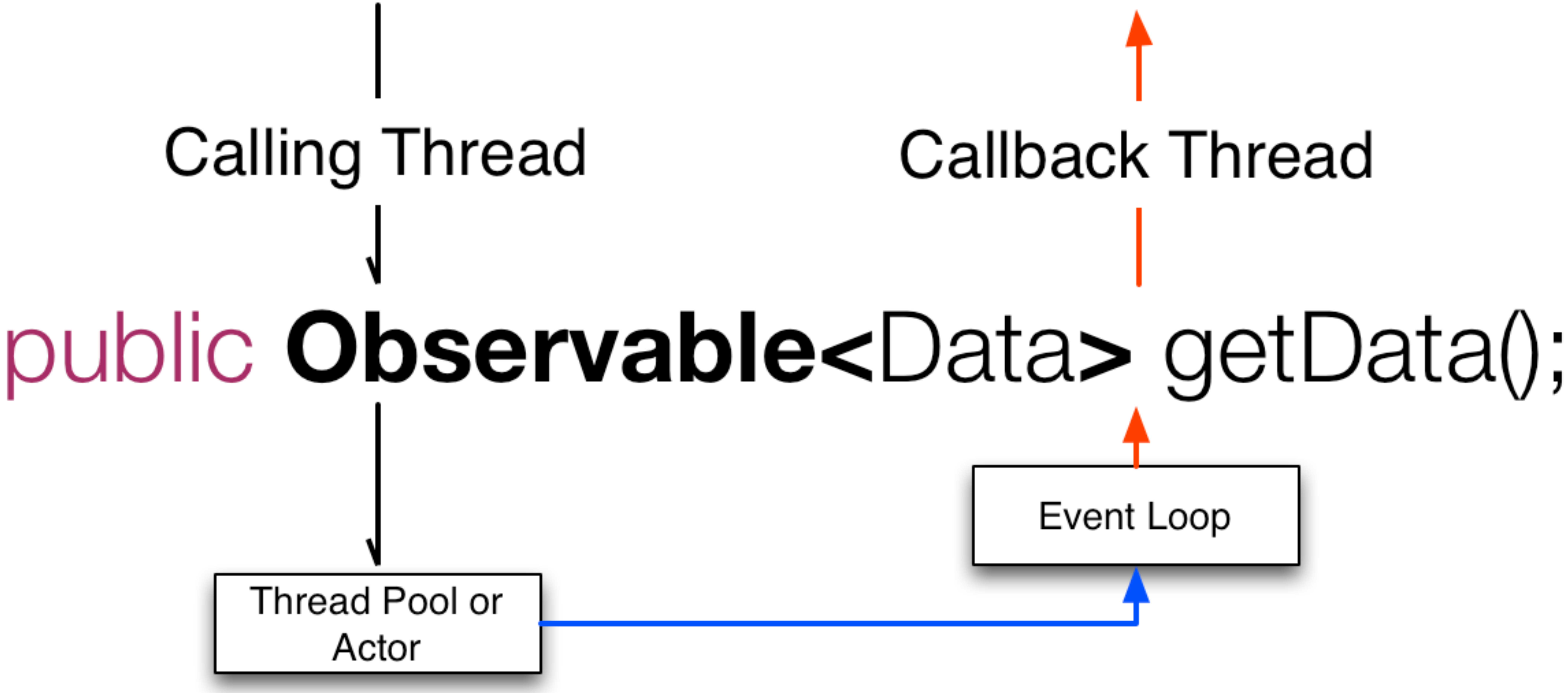
```
public Observable<Data> getData();
```

Event Loop

NIO

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

```
public Observable<Void> handle(HttpServerRequest<ByteBuf> request, HttpServerResponse<ByteBuf> response) {
        // first request User object
        return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
             // then fetch personal catalog
             Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
                     .flatMap(catalogList -> {
                         return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                             Observable<Bookmark> bookmark = getBookmark(video);
                             Observable<Rating> rating = getRating(video);
                             Observable<VideoMetadata> metadata = getMetadata(video);
                             return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                                 return combineVideoData(video, b, r, m);
                             });
                         });
                     });
             // and fetch social data in parallel
             Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
                 return s.getDataAsMap();
             });
             // merge the results
             return Observable.merge(catalog, social);
         }).flatMap(data -> {
             // output as SSE as we get back the data (no waiting until all is done)
             return response.writeAndFlush(new ServerSentEvent(SimpleJson.mapToJson(data)));
         });
     }
```



Calling Thread

Callback Thread

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

Event Loop

Thread Pool or Actor

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

Decouples Consumption from Production

```
// first request User object
return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
     // then fetch personal catalog
     Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
             .flatMap(catalogList -> {
                 return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                     Observable<Bookmark> bookmark = getBookmark(video);
                     Observable<Rating> rating = getRating(video);
                     Observable<VideoMetadata> metadata = getMetadata(video);
                     return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                         return combineVideoData(video, b, r, m);
                     });
                 });
             });
     // and fetch social data in parallel
     Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
         return s.getDataAsMap();
     });
     // merge the results
     return Observable.merge(catalog, social);
 })
```

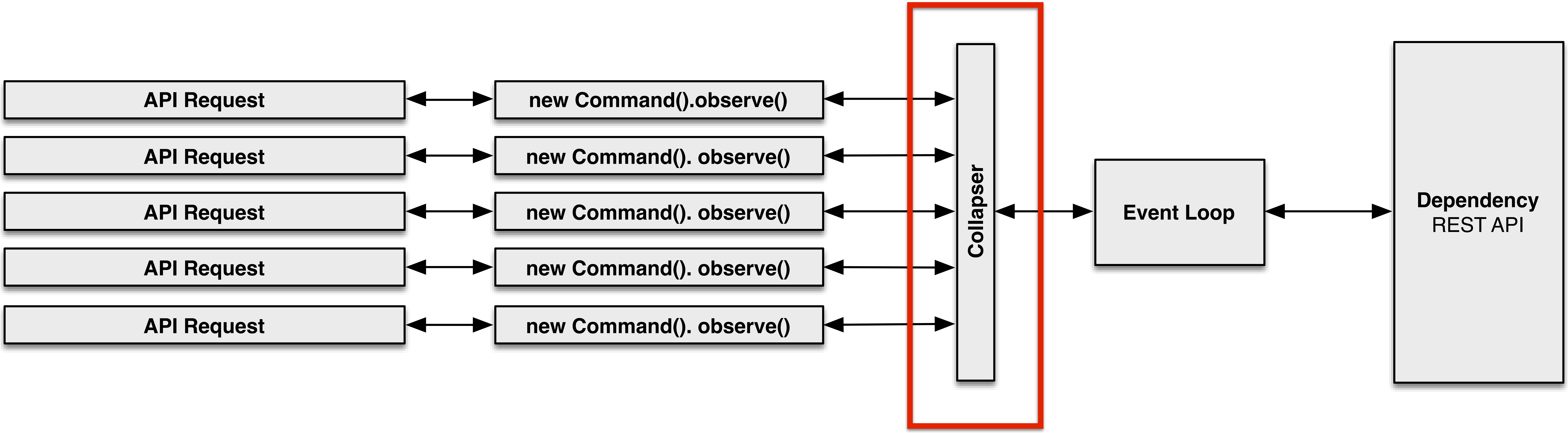
Decouples Consumption from Production

```
// first request User object
return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
     // then fetch personal catalog
     Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
             .flatMap(catalogList -> {
                 return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                     Observable<Bookmark> bookmark = getBookmark(video);
                     Observable<Rating> rating = getRating(video);
                     Observable<VideoMetadata> metadata = getMetadata(video);
                     return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                         return combineVideoData(video, b, r, m);
                     });
                 });
             });
     // and fetch social data in parallel
     Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
         return s.getDataAsMap();
     });
     // merge the results
     return Observable.merge(catalog, social);
 })
```

Decouples Consumption from Production

```
// first request User object
return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {
     // then fetch personal catalog
     Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)
             .flatMap(catalogList -> {
                 return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                     Observable<Bookmark> bookmark = getBookmark(video);
                     Observable<Rating> rating = getRating(video);
                     Observable<VideoMetadata> metadata = getMetadata(video);
                     return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                         return combineVideoData(video, b, r, m);
                     });
                 });
             });
     // and fetch social data in parallel
     Observable<Map<String, Object>> social = getSocialData(user).map(s -> {
         return s.getDataAsMap();
     });
     // merge the results
     return Observable.merge(catalog, social);
 })
```

Decouples Consumption from Production



~5 network calls

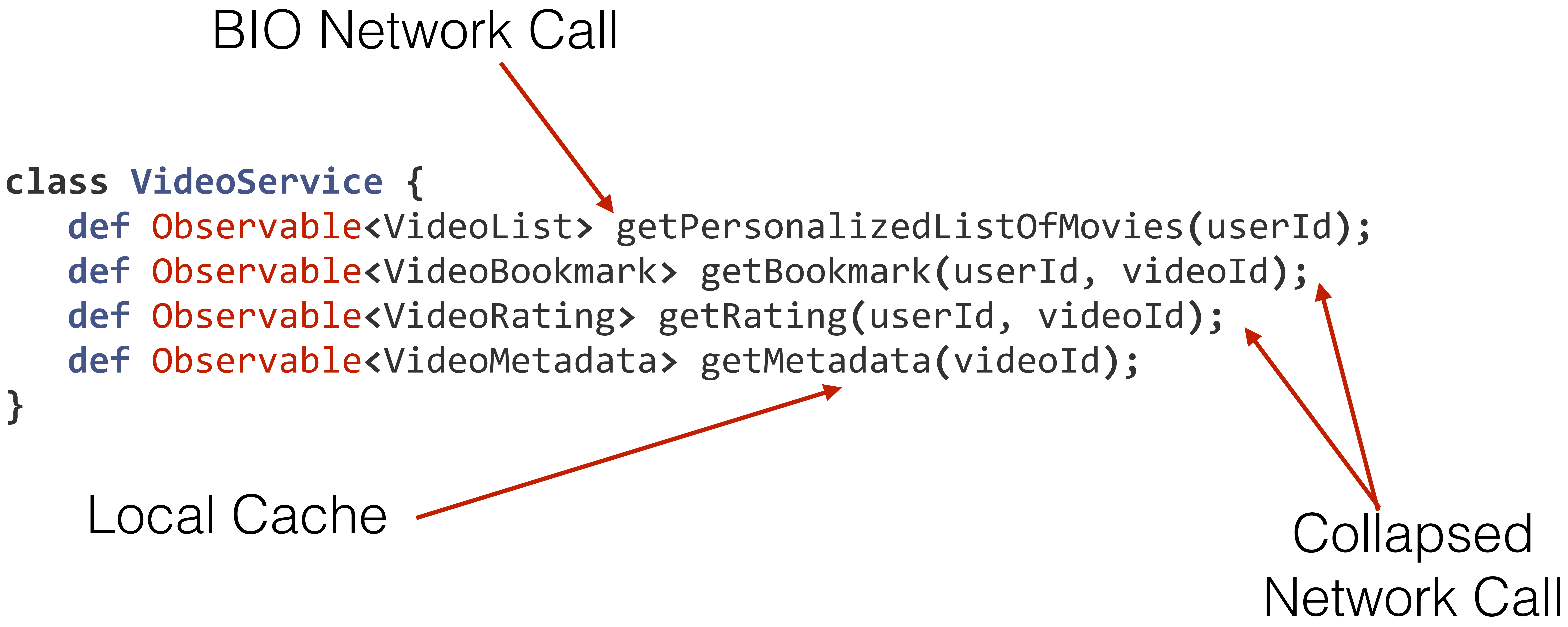
## (#3 and #4 may result in more due to windowing)

```
// first request User object
return getUser(request.getQueryParameters().get("userId")).flatMap(user -> {     1
     // then fetch personal catalog
     Observable<Map<String, Object>> catalog = getPersonalizedCatalog(user)     2
             .flatMap(catalogList -> {
                 return catalogList.videos().<Map<String, Object>> flatMap(video -> {
                     Observable<Bookmark> bookmark = getBookmark(video);     3
                     Observable<Rating> rating = getRating(video);     4
                     Observable<VideoMetadata> metadata = getMetadata(video);
                     return Observable.zip(bookmark, rating, metadata, (b, r, m) -> {
                         return combineVideoData(video, b, r, m);
                     });
                 });
             });
     // and fetch social data in parallel
     Observable<Map<String, Object>> social = getSocialData(user).map(s -> {     5
         return s.getDataAsMap();
     });
     // merge the results
     return Observable.merge(catalog, social);
 })
```

Clear API Communicates Potential Cost

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

Implementation Can Differ



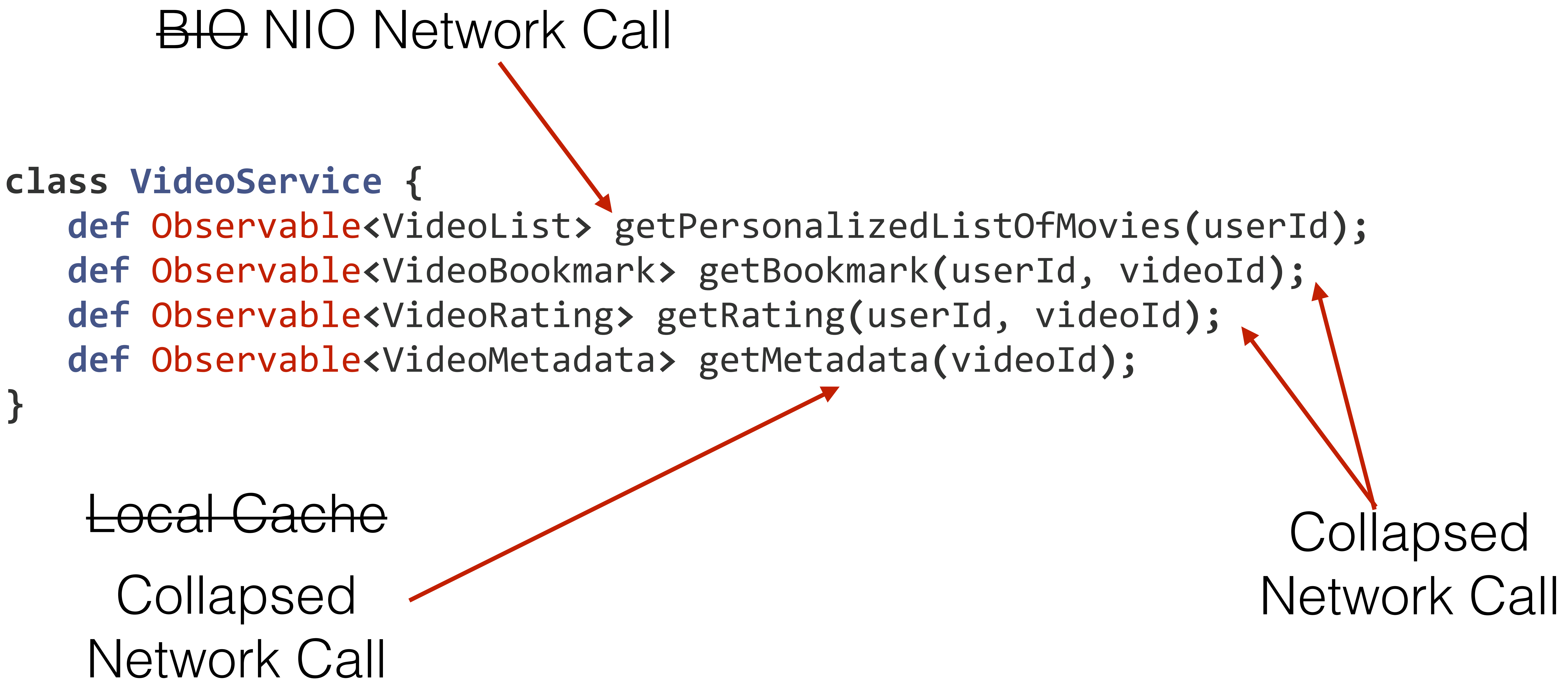
BIO Network Call

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

Local Cache

Collapsed
Network Call

Implementation Can Differ and Change



~~BIO~~ NIO Network Call

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

~~Local Cache~~

Collapsed Network Call

Collapsed Network Call

Retrieval, Transformation, Combination
all done in same **declarative** manner

What about … ?

Error Handling

```
Observable.create(subscriber -> {
    throw new RuntimeException("failed!");
}).onErrorResumeNext(throwable -> {
    return Observable.just("fallback value");
}).subscribe(System.out::println);
```

```java
Observable.create(subscriber -> {
    throw new RuntimeException("failed!");
}).onErrorReturn(throwable -> {
    return "fallback value";
}).subscribe(System.out::println);
```

```java
Observable.create(subscriber -> {
    throw new RuntimeException("failed!");
}).retryWhen(attempts -> {
    return attempts.zipWith(Observable.range(1, 3), (throwable, i) -> i)
            .flatMap(i -> {
                System.out.println("delay retry by " + i + " second(s)");
                return Observable.timer(i, TimeUnit.SECONDS);
            }).concatWith(Observable.error(new RuntimeException("Exceeded 3 retries")));
})
.subscribe(System.out::println, t -> t.printStackTrace());
```

```
Observable.create(subscriber -> {
    throw new RuntimeException("failed!");
}).retryWhen(attempts -> {
    return attempts.zipWith(Observable.range(1, 3), (throwable, i) -> i)
            .flatMap(i -> {
                System.out.println("delay retry by " + i + " second(s)");
                return Observable.timer(i, TimeUnit.SECONDS);
            }).concatWith(Observable.error(new RuntimeException("Exceeded 3 retries")));
})
.subscribe(System.out::println, t -> t.printStackTrace());
```

```
Observable.create(subscriber -> {
    throw new RuntimeException("failed!");
}).retryWhen(attempts -> {
    return attempts.zipWith(Observable.range(1, 3), (throwable, i) -> i)
            .flatMap(i -> {
                System.out.println("delay retry by " + i + " second(s)");
                return Observable.timer(i, TimeUnit.SECONDS);
            }).concatWith(Observable.error(new RuntimeException("Exceeded 3 retries")));
})
.subscribe(System.out::println, t -> t.printStackTrace());
```

```java
Observable.create(subscriber -> {
    throw new RuntimeException("failed!");
}).retryWhen(attempts -> {
    return attempts.zipWith(Observable.range(1, 3), (throwable, i) -> i)
            .flatMap(i -> {
                System.out.println("delay retry by " + i + " second(s)");
                return Observable.timer(i, TimeUnit.SECONDS);
            }).concatWith(Observable.error(new RuntimeException("Exceeded 3 retries")));
})
.subscribe(System.out::println, t -> t.printStackTrace());
```

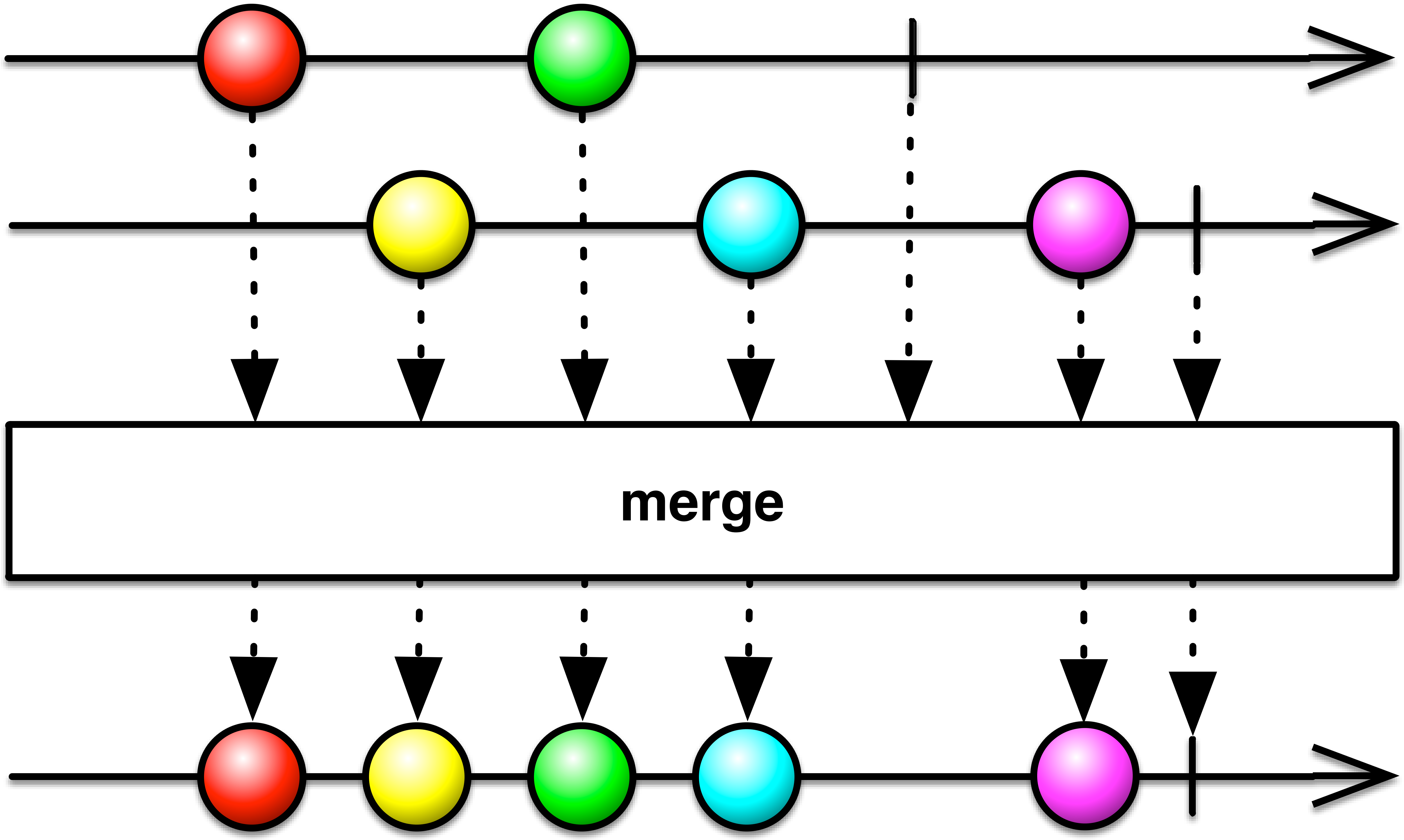
Concurrency

Concurrency

an Observable is sequential
(no concurrent emissions)

scheduling and combining Observables
enables concurrency while retaining sequential emission

```
// merging async Observables allows each
// to execute concurrently
Observable.merge(getDataAsync(1), getDataAsync(2))
```



```
// concurrently fetch data for 5 items
Observable.range(0, 5).flatMap(i -> {
    return getDataAsync(i);
})
```

```
Observable.range(0, 5000).window(500).flatMap(work -> {
    return work.observeOn(Schedulers.computation())
       .map(item -> {
         // simulate computational work
         try { Thread.sleep(1); } catch (Exception e) {}
         return item + " processed " + Thread.currentThread();
    });
})
```

```
Observable.range(0, 5000).buffer(500).flatMap(is -> {
    return Observable.from(is).subscribeOn(Schedulers.computation())
       .map(item -> {
         // simulate computational work
         try { Thread.sleep(1); } catch (Exception e) {}
         return item + " processed " + Thread.currentThread();
    });
})
```

Flow Control

Flow Control

(backpressure)

```
Observable.from(iterable).take(1000).map(i -> "value_" + i).subscribe(System.out::println);
```

no backpressure needed

```
Observable.from(iterable).take(1000).map(i -> "value_" + i).subscribe(System.out::println);
```

no backpressure needed

synchronous on same thread
(no queueing)

```
Observable.from(iterable).take(1000).map(i -> "value_" + i)
                    .observeOn(Schedulers.computation()).subscribe(System.out::println);
```

backpressure needed

```
Observable.from(iterable).take(1000).map(i -> "value_" + i)
                    .observeOn(Schedulers.computation()).subscribe(System.out::println);
```

backpressure needed

asynchronous
(queueing)

Flow Control Options

Hot

emits whether you're ready or not

examples
mouse and keyboard events
system events
stock prices

```
Observable.create(subscriber -> {
  // register with data source
})
```

flow control

Cold

emits when requested
(generally at controlled rate)

examples
database query
web service request
reading file

```
Observable.create(subscriber -> {
  // fetch data
})
```

flow control & backpressure

Block

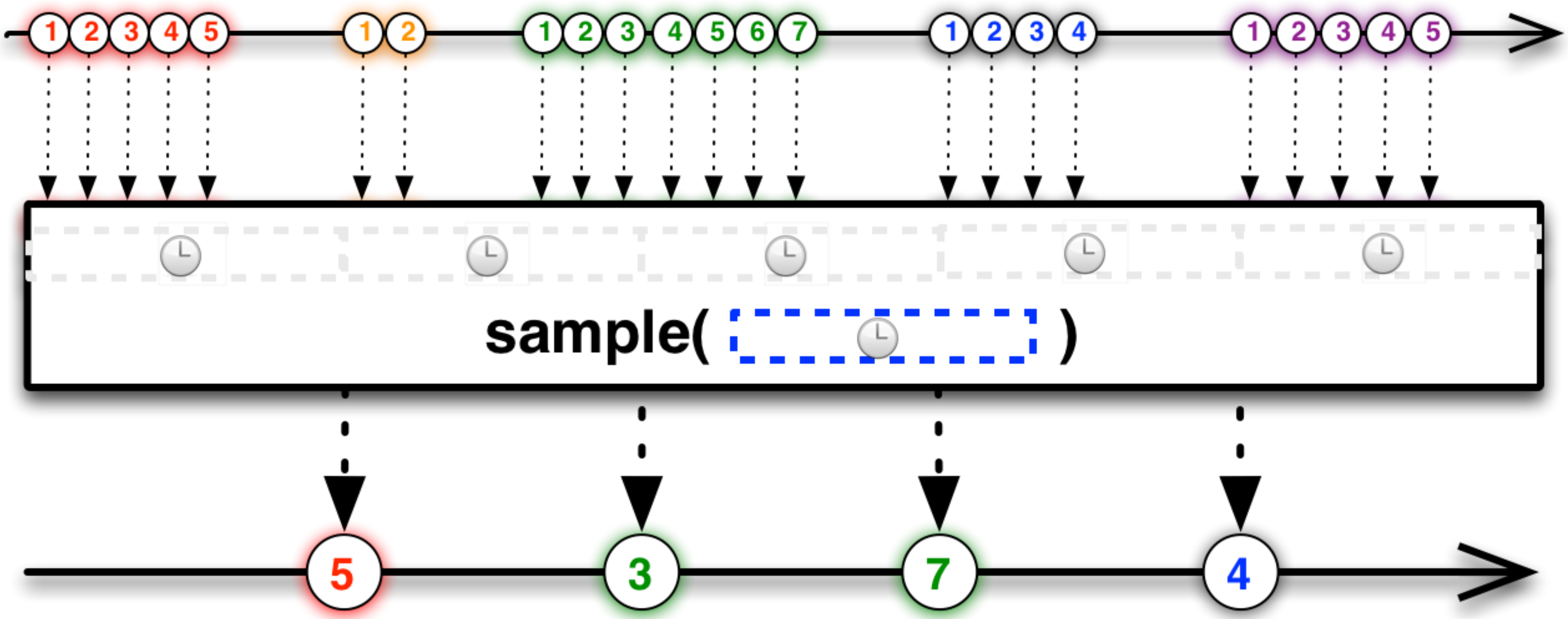
(callstack blocking and/or park the thread)

Hot or Cold Streams

Temporal Operators

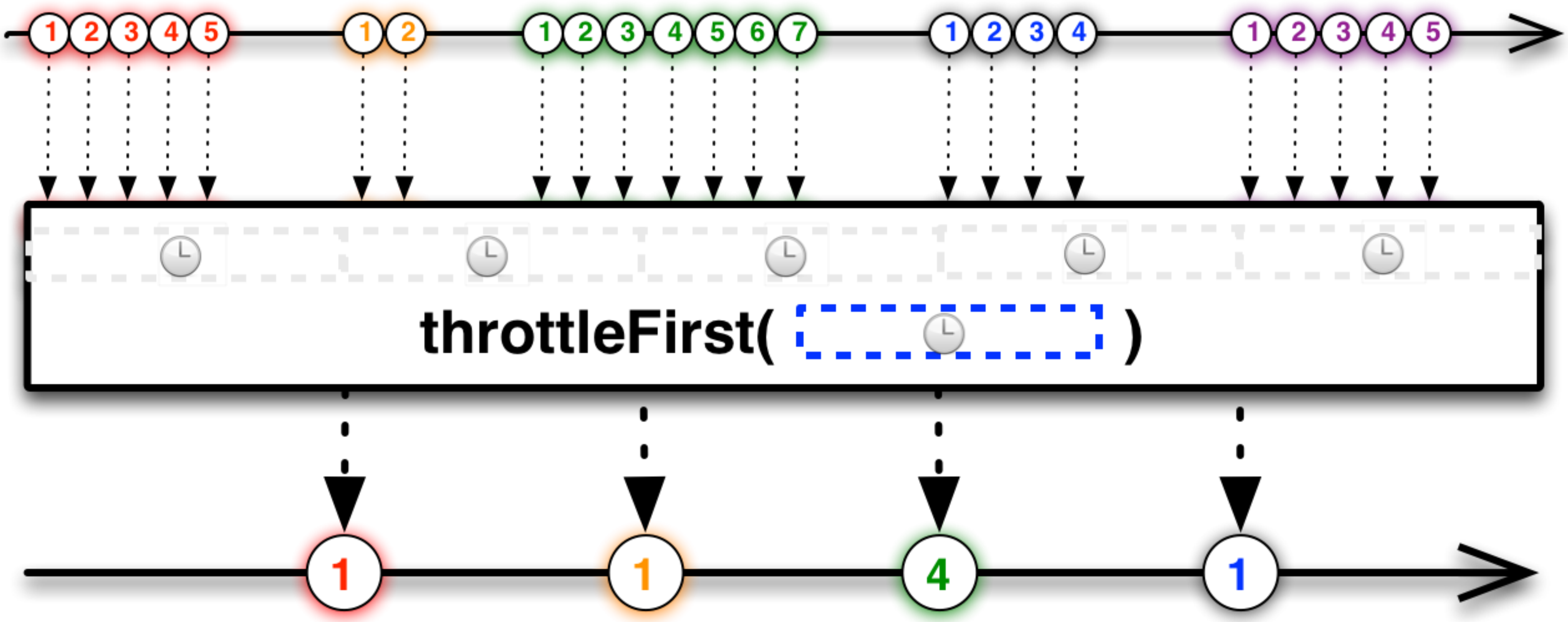
(batch or drop data using time)

Hot Streams



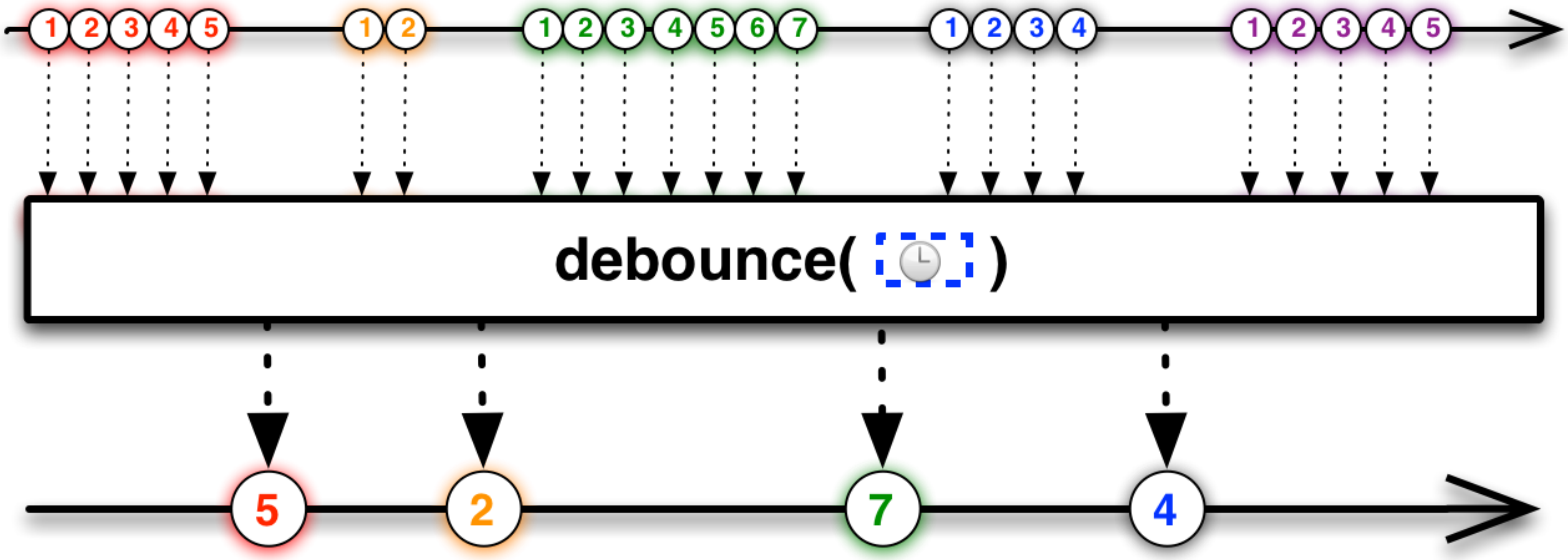
```
Observable.range(1, 1000000).sample(10, TimeUnit.MILLISECONDS).forEach(System.out::println);
```

```
110584
242165
544453
942880
```



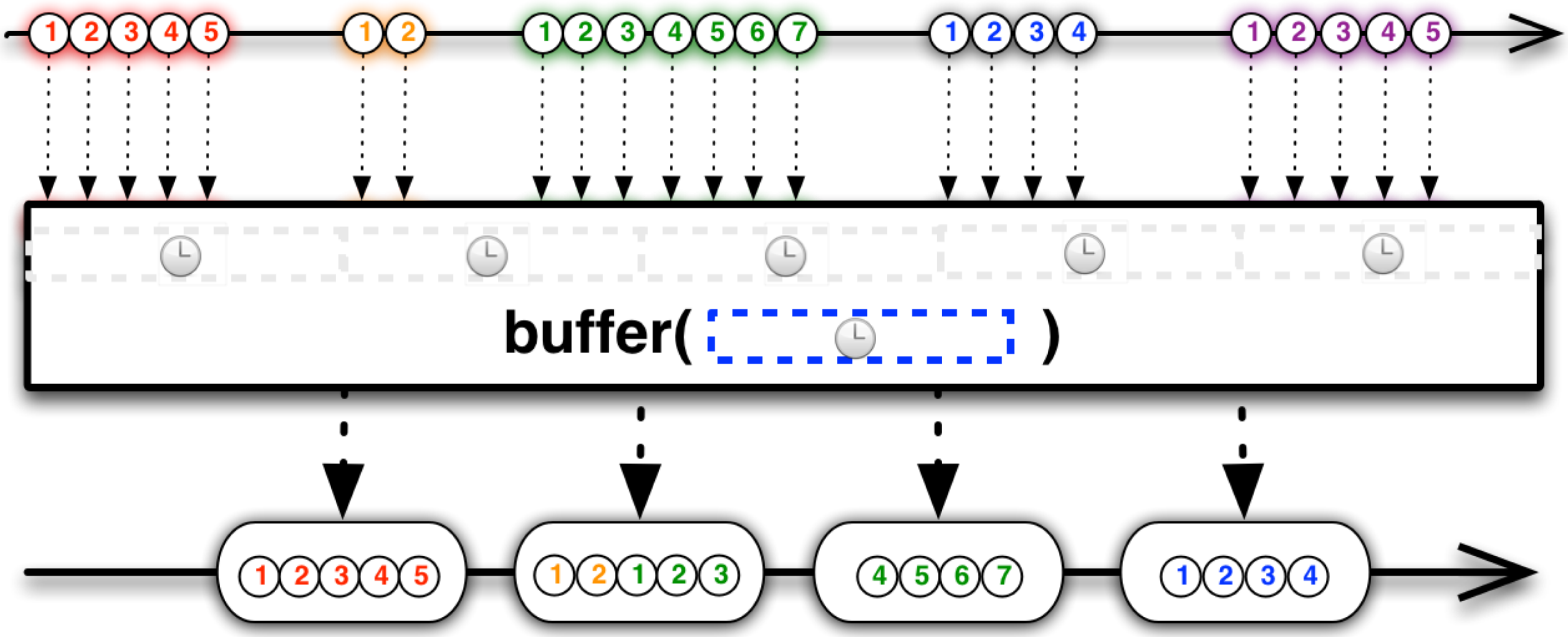
```
Observable.range(1, 1000000).throttleFirst(10, TimeUnit.MILLISECONDS).forEach(System.out::println);
```

```
1
55463
163962
308545
457445
592638
751789
897159
```



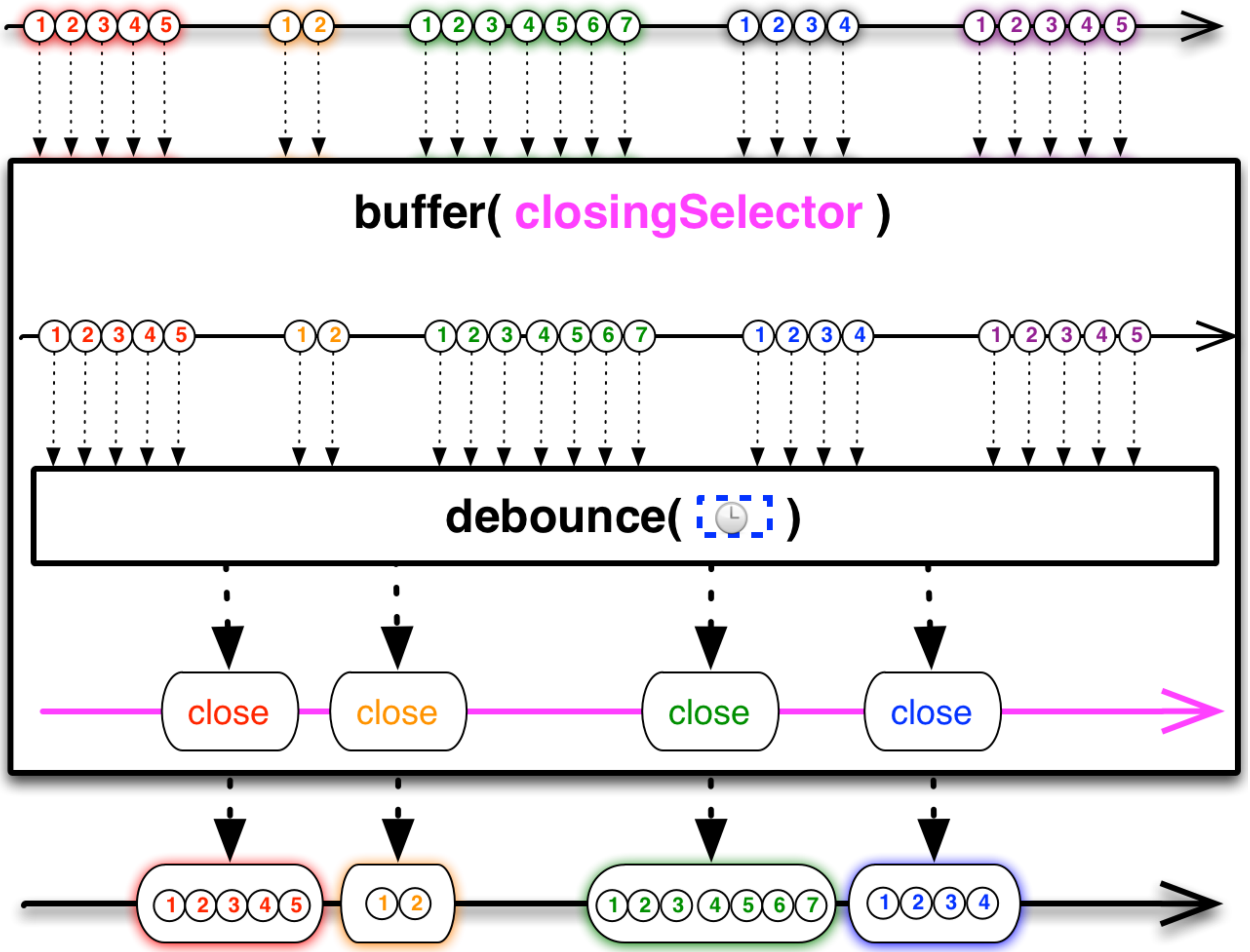
```
Observable.range(1, 1000000).debounce(10, TimeUnit.MILLISECONDS).forEach(System.out::println);
```

```
1000000
```

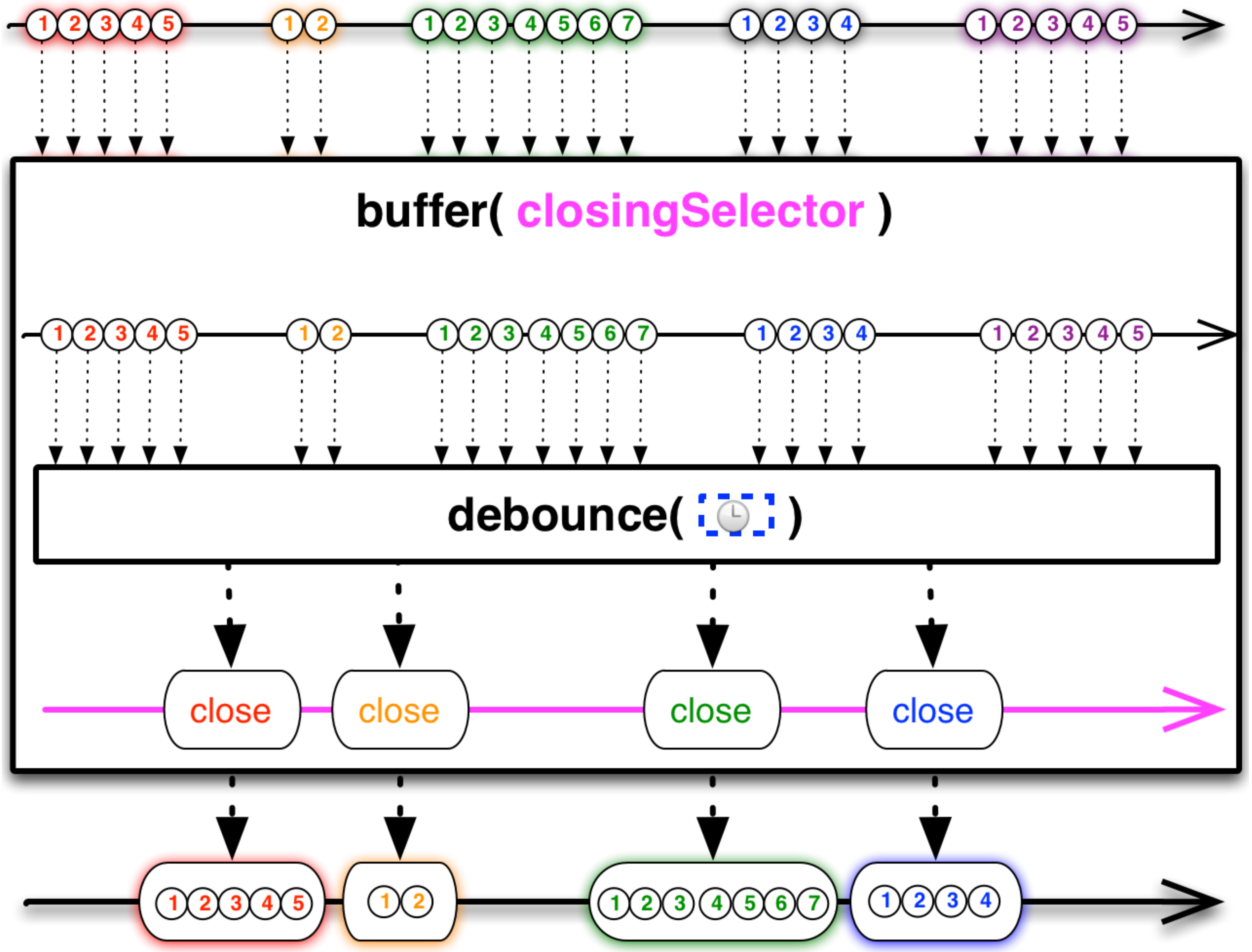


```
Observable.range(1, 1000000).buffer(10, TimeUnit.MILLISECONDS)
                    .toBlocking().forEach(list -> System.out.println("batch: " + list.size()));
```

```
batch: 71141
batch: 49488
batch: 141147
batch: 141432
batch: 195920
batch: 240462
batch: 160410
```

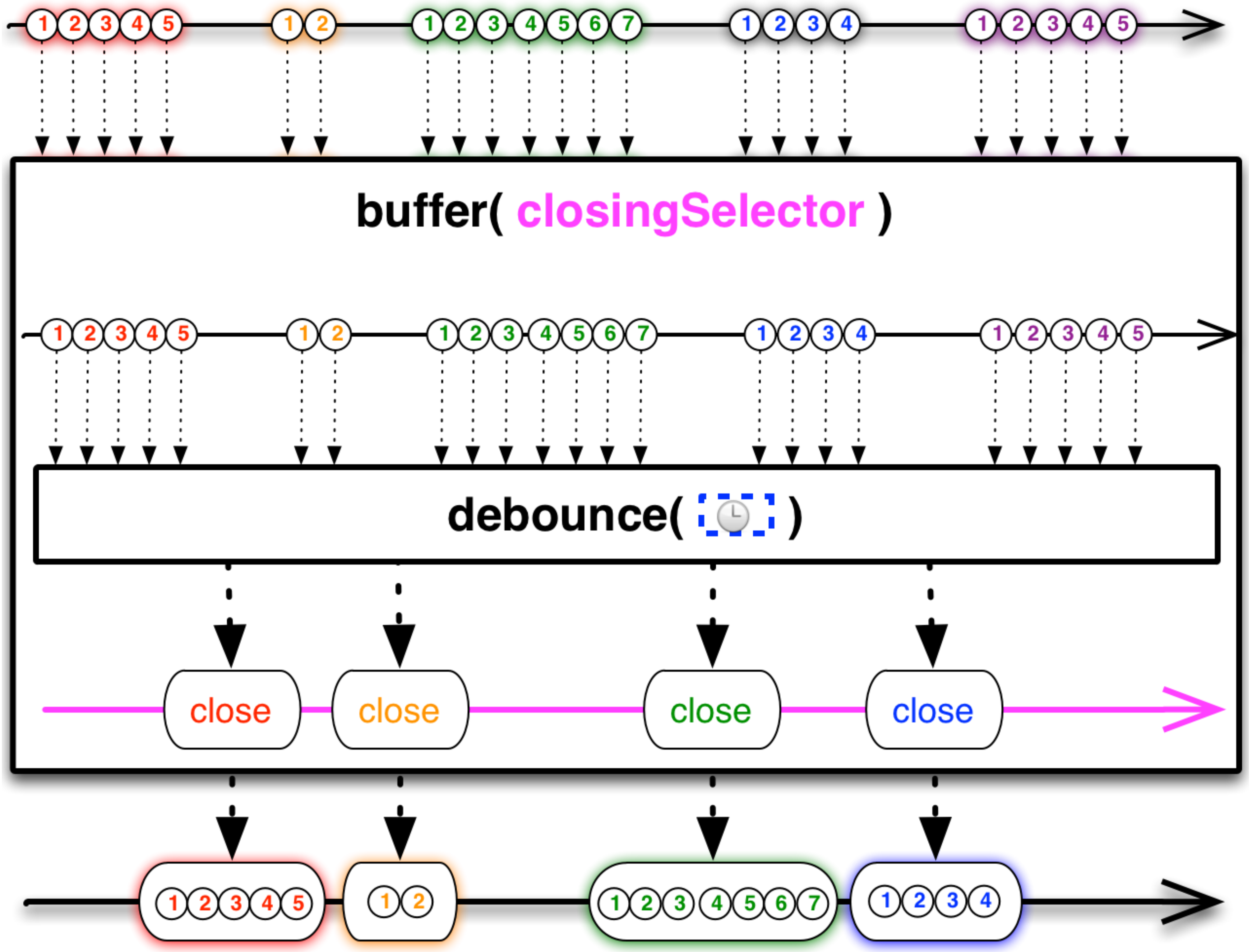

1 2 3 4 5
1 2
1 2 3 4 5 6 7
1 2 3 4
1 2 3 4 5
buffer(closingSelector)
1 2 3 4 5
1 2
1 2 3 4 5 6 7
1 2 3 4
1 2 3 4 5
debounce(🕒)
close
close
close
close
1 2 3 4 5
1 2
1 2 3 4 5 6 7
1 2 3 4

```
/* The following will emit a buffered list as it is debounced */
// first we multicast the stream ... using refCount so it handles the subscribe/unsubscribe
Observable<Integer> burstStream = intermittentBursts().take(20).publish().refCount();
// then we get the debounced version
Observable<Integer> debounced = burstStream.debounce(10, TimeUnit.MILLISECONDS);
// then the buffered one that uses the debounced stream to demark window start/stop
Observable<List<Integer>> buffered = burstStream.buffer(debounced);
// then we subscribe to the buffered stream so it does what we want
buffered.toBlocking().forEach(System.out::println);
```



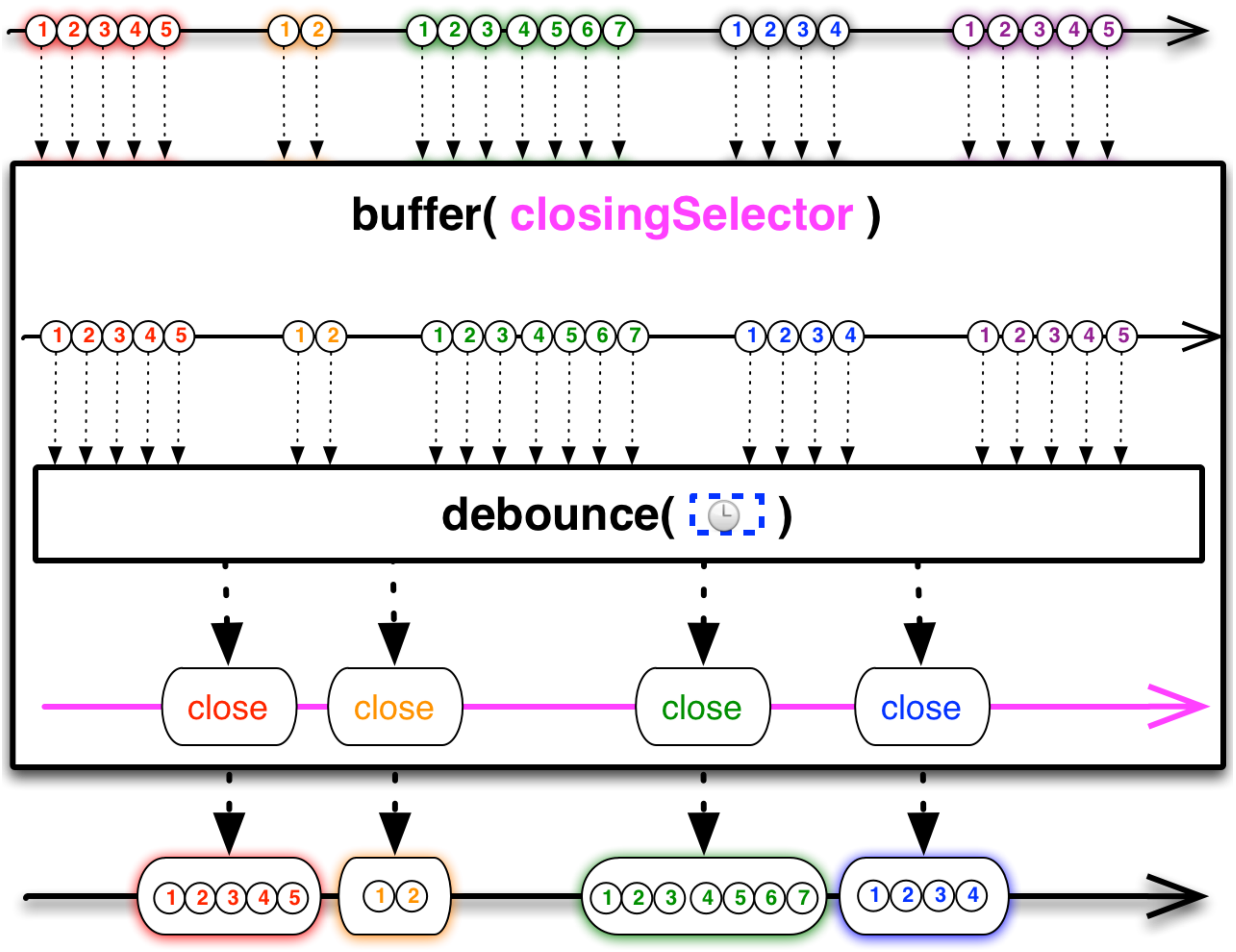
```
[0, 1, 2]
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4]
[0, 1]
[]
```

```
/* The following will emit a buffered list as it is debounced */
// first we multicast the stream ... using refCount so it handles the subscribe/unsubscribe
Observable<Integer> burstStream = intermittentBursts().take(20).publish().refCount();
// then we get the debounced version
Observable<Integer> debounced = burstStream.debounce(10, TimeUnit.MILLISECONDS);
// then the buffered one that uses the debounced stream to demark window start/stop
Observable<List<Integer>> buffered = burstStream.buffer(debounced);
// then we subscribe to the buffered stream so it does what we want
buffered.toBlocking().forEach(System.out::println);
```



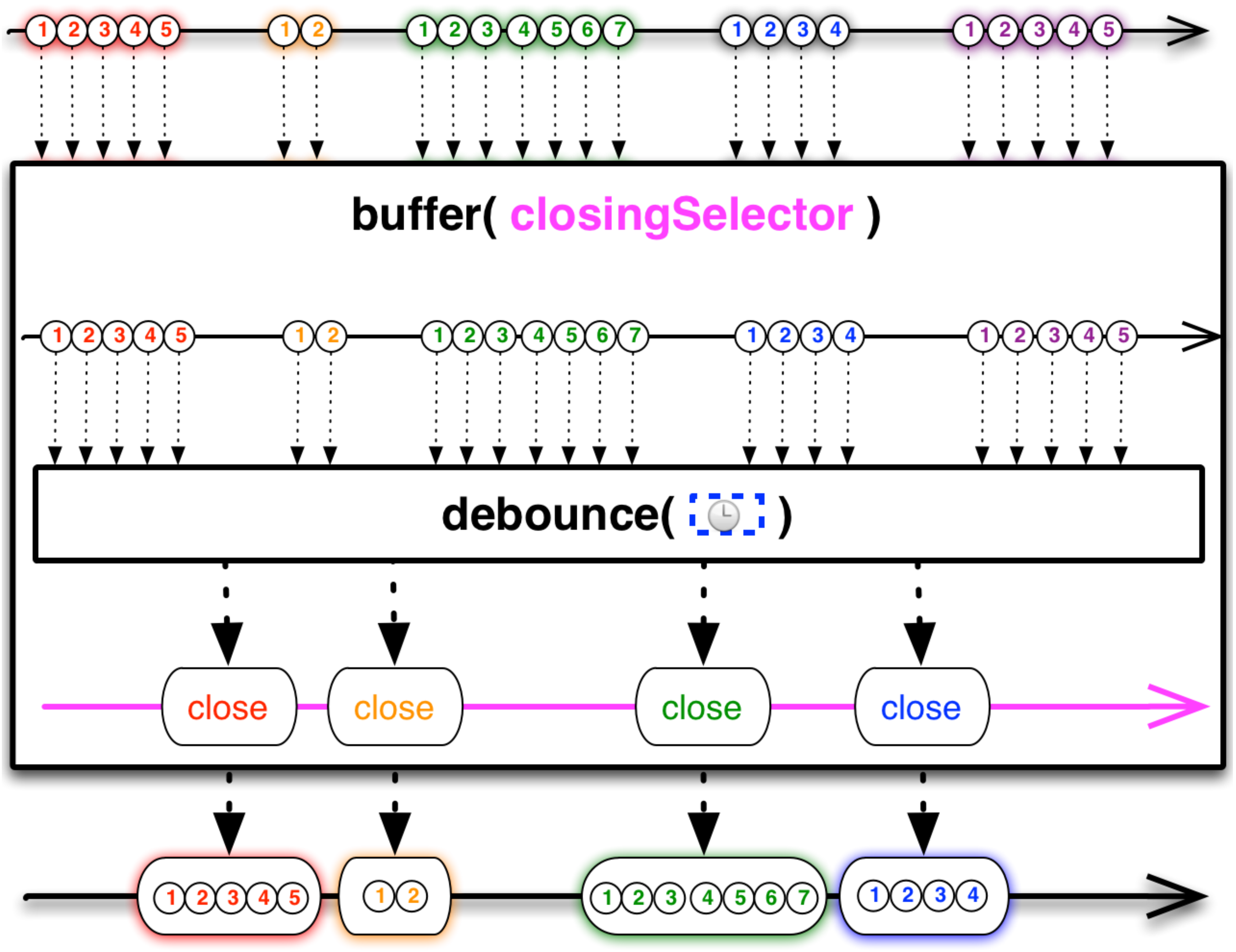
```
[0, 1, 2]
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4]
[0, 1]
[]
```

```
/* The following will emit a buffered list as it is debounced */
// first we multicast the stream ... using refCount so it handles the subscribe/unsubscribe
Observable<Integer> burstStream = intermittentBursts().take(20).publish().refCount();
// then we get the debounced version
Observable<Integer> debounced = burstStream.debounce(10, TimeUnit.MILLISECONDS);
// then the buffered one that uses the debounced stream to demark window start/stop
Observable<List<Integer>> buffered = burstStream.buffer(debounced);
// then we subscribe to the buffered stream so it does what we want
buffered.toBlocking().forEach(System.out::println);
```



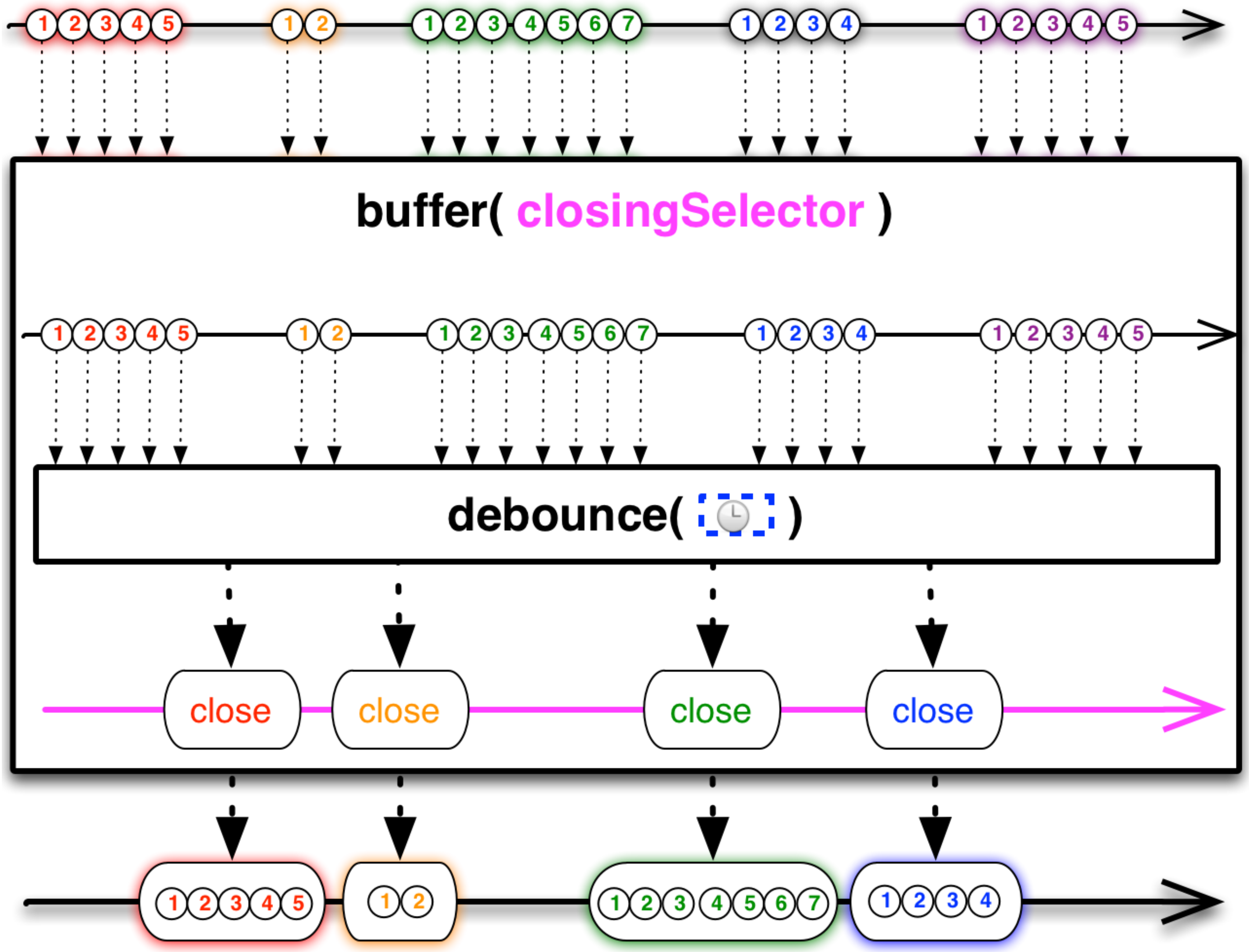
```
[0, 1, 2]
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4]
[0, 1]
[]
```

```java
/* The following will emit a buffered list as it is debounced */
// first we multicast the stream ... using refCount so it handles the subscribe/unsubscribe
Observable<Integer> burstStream = intermittentBursts().take(20).publish().refCount();
// then we get the debounced version
Observable<Integer> debounced = burstStream.debounce(10, TimeUnit.MILLISECONDS);
// then the buffered one that uses the debounced stream to demark window start/stop
Observable<List<Integer>> buffered = burstStream.buffer(debounced);
// then we subscribe to the buffered stream so it does what we want
buffered.toBlocking().forEach(System.out::println);
```



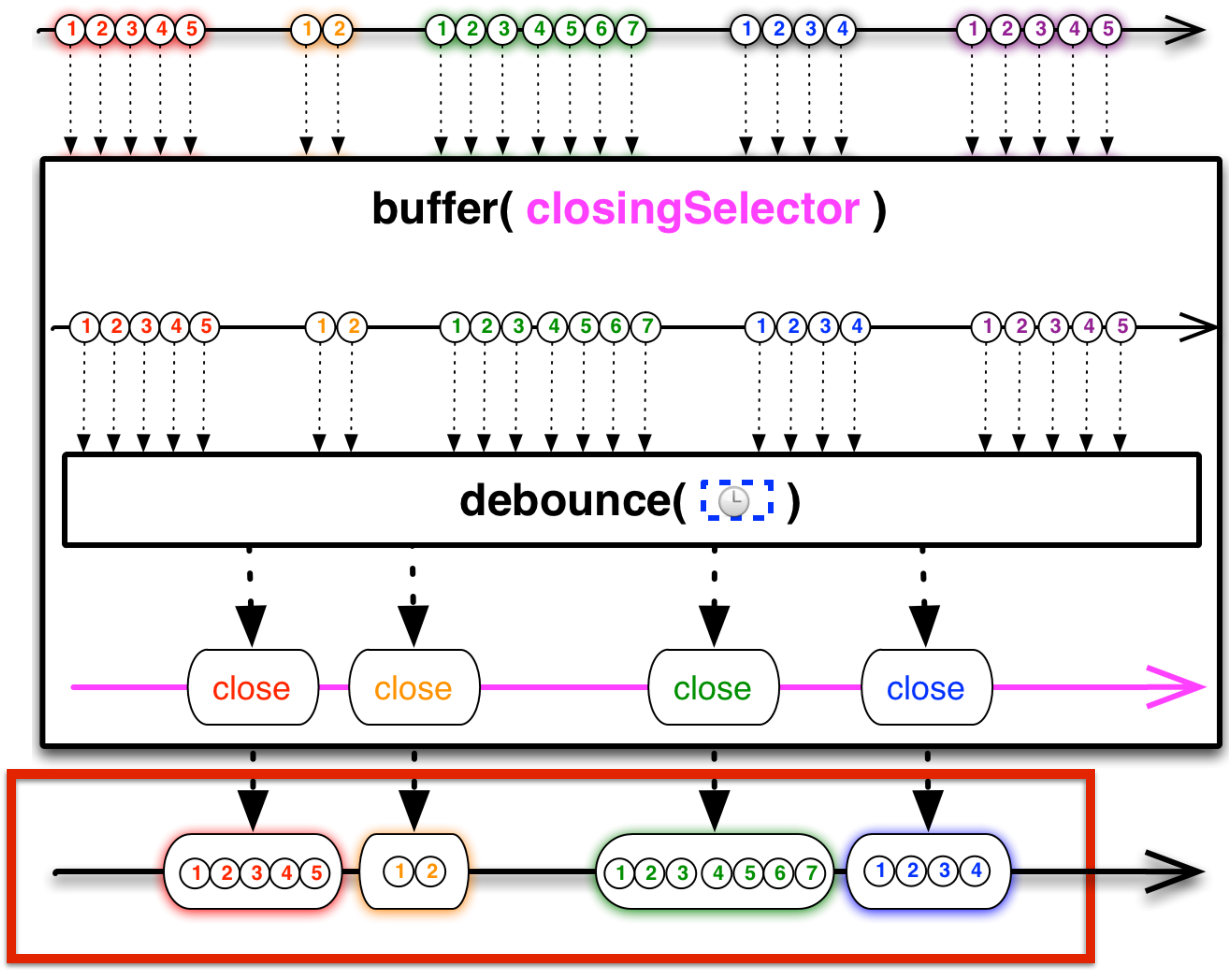
```
[0, 1, 2]
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4]
[0, 1]
[]
```

```
/* The following will emit a buffered list as it is debounced */
// first we multicast the stream ... using refCount so it handles the subscribe/unsubscribe
Observable<Integer> burstStream = intermittentBursts().take(20).publish().refCount();
// then we get the debounced version
Observable<Integer> debounced = burstStream.debounce(10, TimeUnit.MILLISECONDS);
// then the buffered one that uses the debounced stream to demark window start/stop
Observable<List<Integer>> buffered = burstStream.buffer(debounced);
// then we subscribe to the buffered stream so it does what we want
buffered.toBlocking().forEach(System.out::println);
```



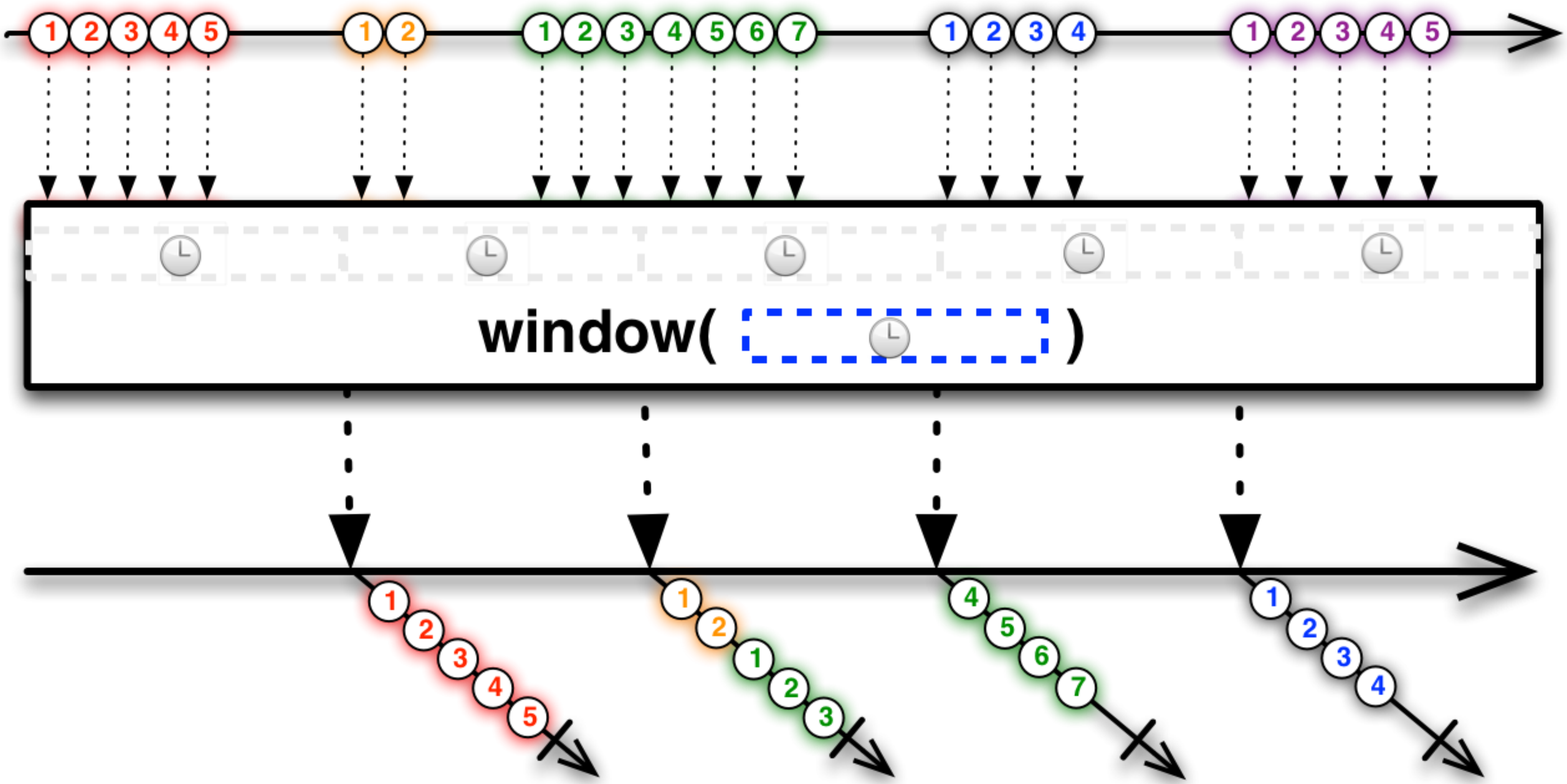
```
[0, 1, 2]
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4]
[0, 1]
[]
```

```java
/* The following will emit a buffered list as it is debounced */
// first we multicast the stream ... using refCount so it handles the subscribe/unsubscribe
Observable<Integer> burstStream = intermittentBursts().take(20).publish().refCount();
// then we get the debounced version
Observable<Integer> debounced = burstStream.debounce(10, TimeUnit.MILLISECONDS);
// then the buffered one that uses the debounced stream to demark window start/stop
Observable<List<Integer>> buffered = burstStream.buffer(debounced);
// then we subscribe to the buffered stream so it does what we want
buffered.toBlocking().forEach(System.out::println);
```



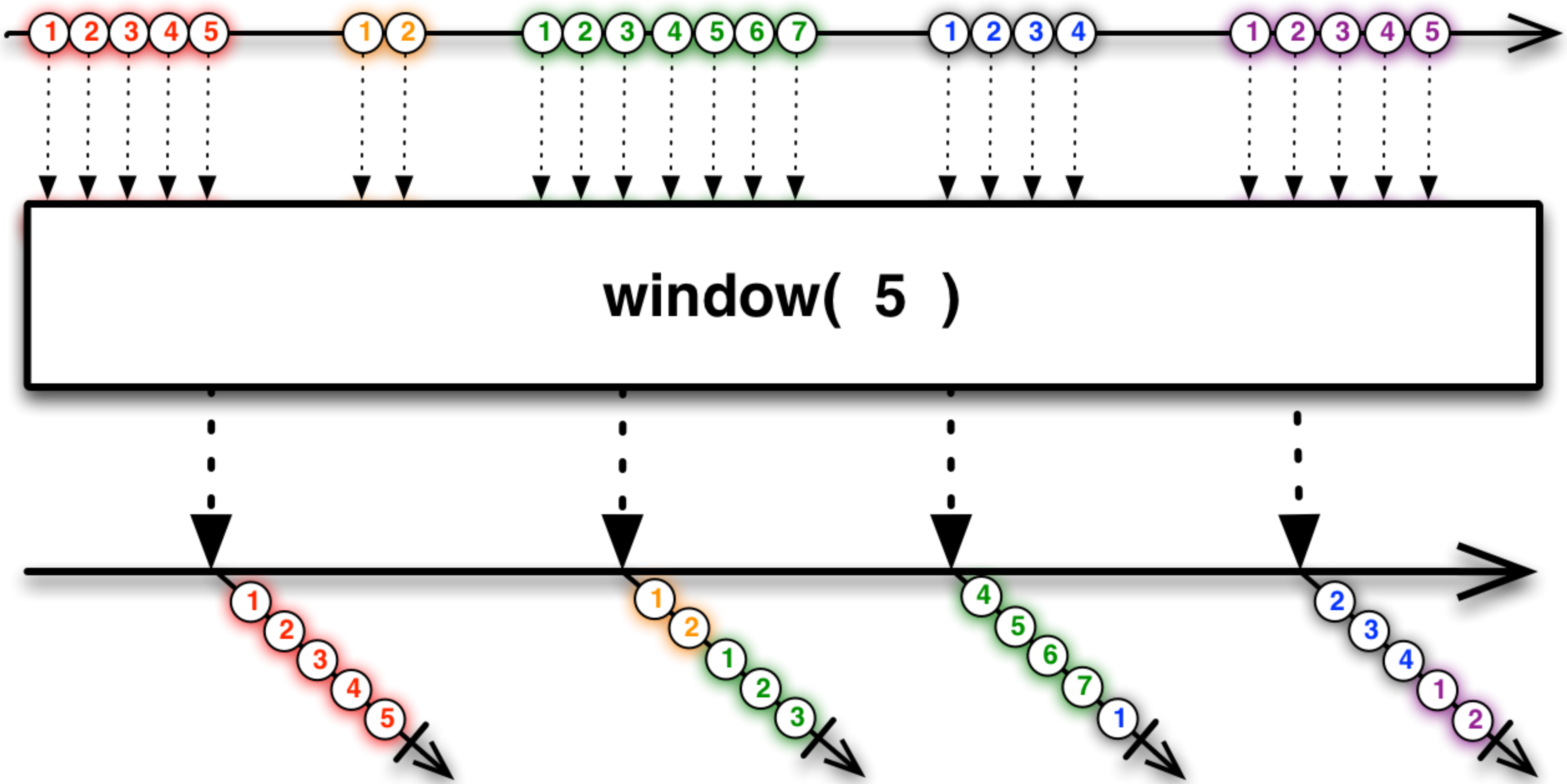
```
[0, 1, 2]
[0, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 3, 4]
[0, 1]
[]
```

https://gist.github.com/benjchristensen/e4524a308456f3c21c0b



```
Observable.range(1, 1000000).window(50, TimeUnit.MILLISECONDS)
        .flatMap(window -> window.count())
        .toBlocking().forEach(count -> System.out.println("num items: " + count));
```

```
num items: 477769
num items: 155463
num items: 366768
```



```
Observable.range(1, 1000000).window(500000)
         .flatMap(window -> window.count())
         .toBlocking().forEach(count -> System.out.println("num items: " + count));
```

```
num items: 500000
num items: 500000
```

Reactive Pull

(dynamic push-pull)

Push (reactive) when consumer keeps up with producer.

Switch to **Pull** (interactive) when consumer is slow.

Bound all* queues.

Push (reactive) when consumer keeps up with producer.

Switch to **Pull** (interactive) when consumer is slow.

Bound all* queues.

*vertically, not horizontally

Reactive Pull

hot vs cold

Reactive Pull

cold supports pull

Cold Streams

emits when requested
(generally at controlled rate)

examples
database query
web service request
reading file

```
Observable.from(iterable)
Observable.from(0, 100000)
```

Cold Streams

emits when requested
(generally at controlled rate)

examples
database query
web service request
reading file

```
Observable.from(iterable)
Observable.from(0, 100000)
```

Pull

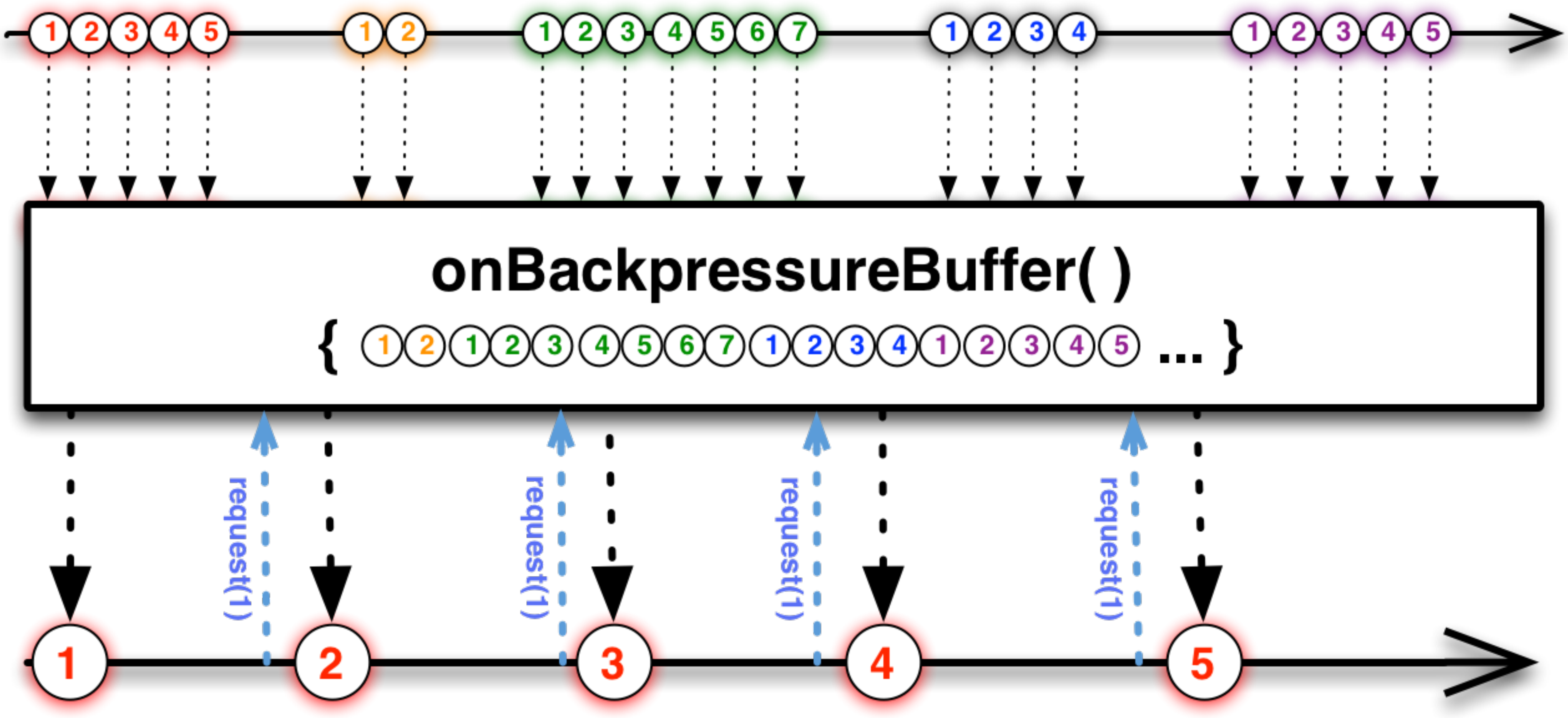
Reactive Pull

hot receives signal

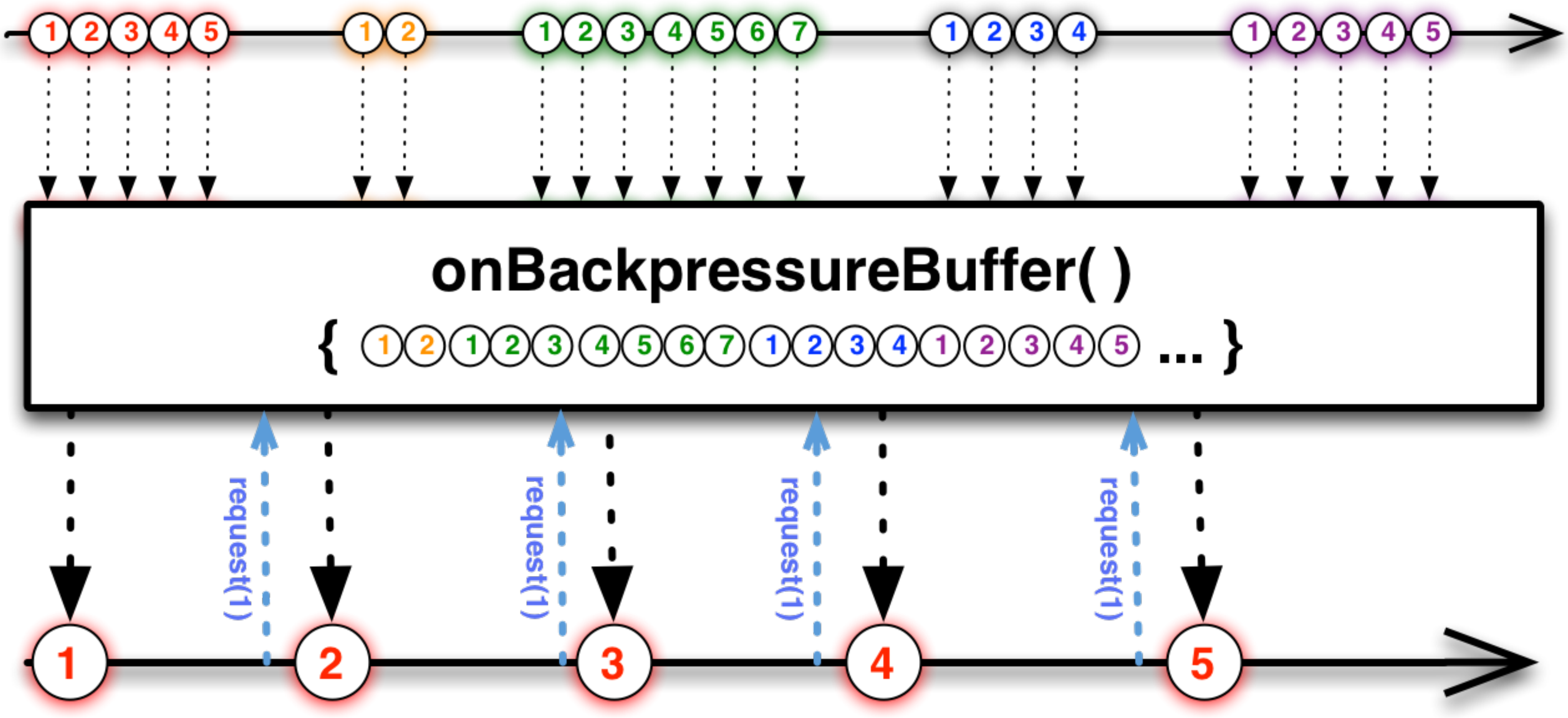
Reactive Pull

hot receives signal

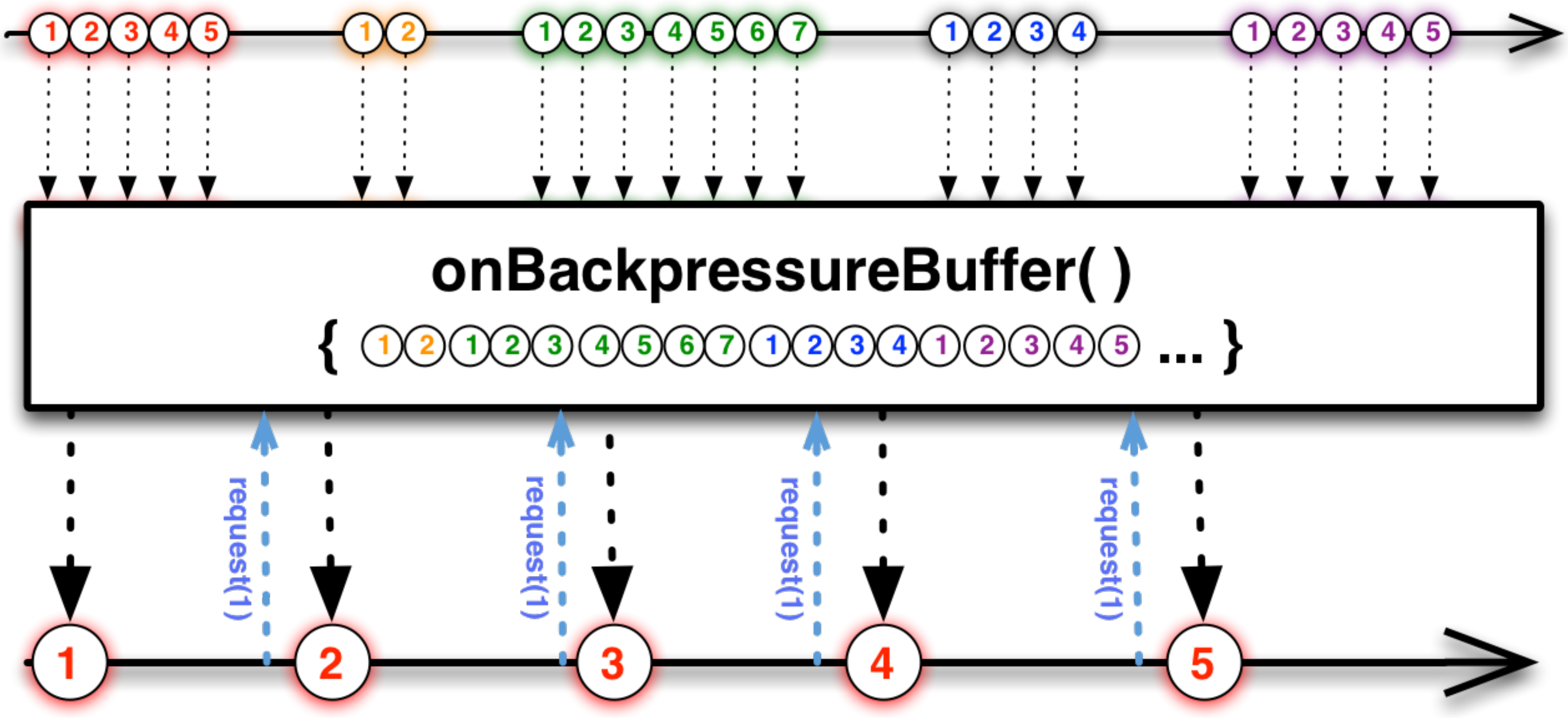
*including Observables that don’t implement reactive pull support



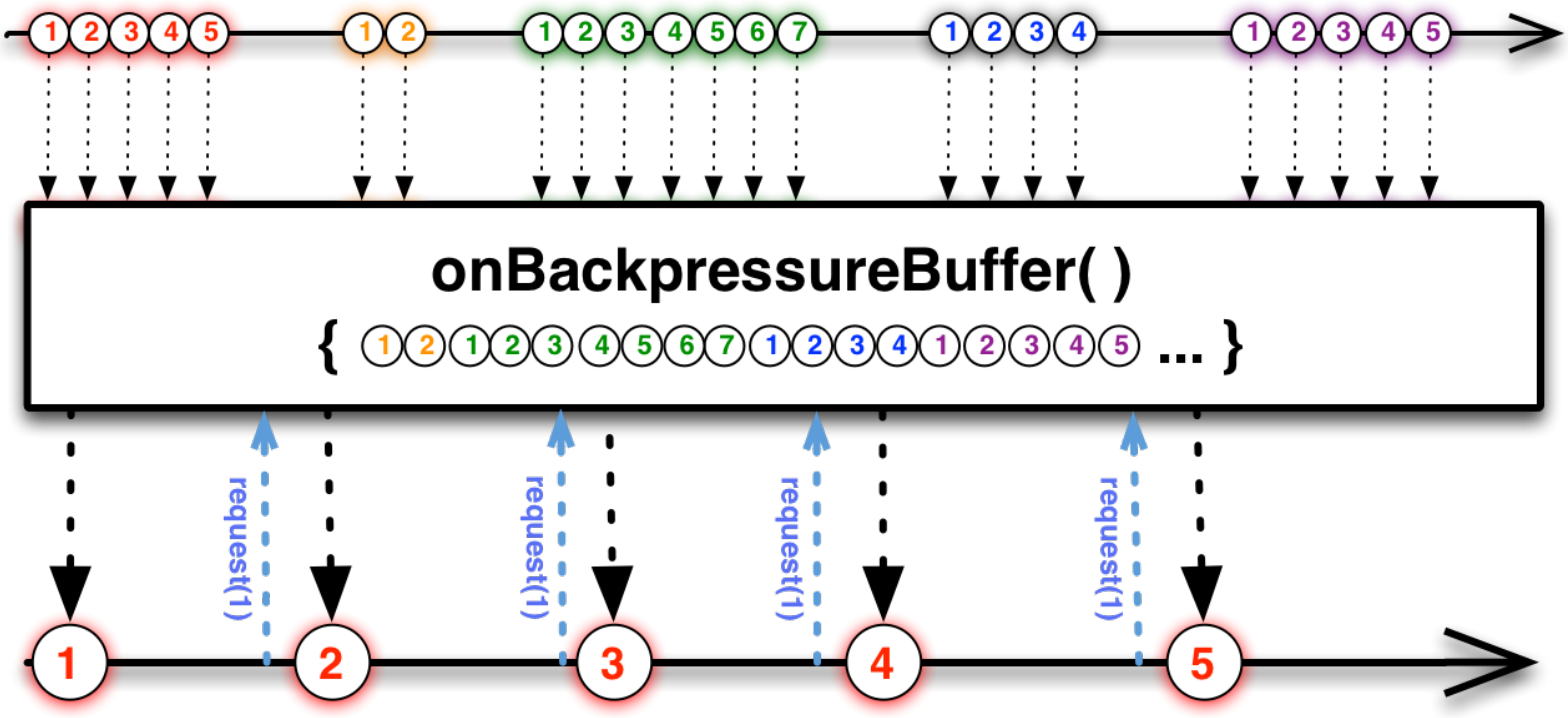
```
hotSourceStream.onBackpressureBuffer().observeOn(aScheduler);
```



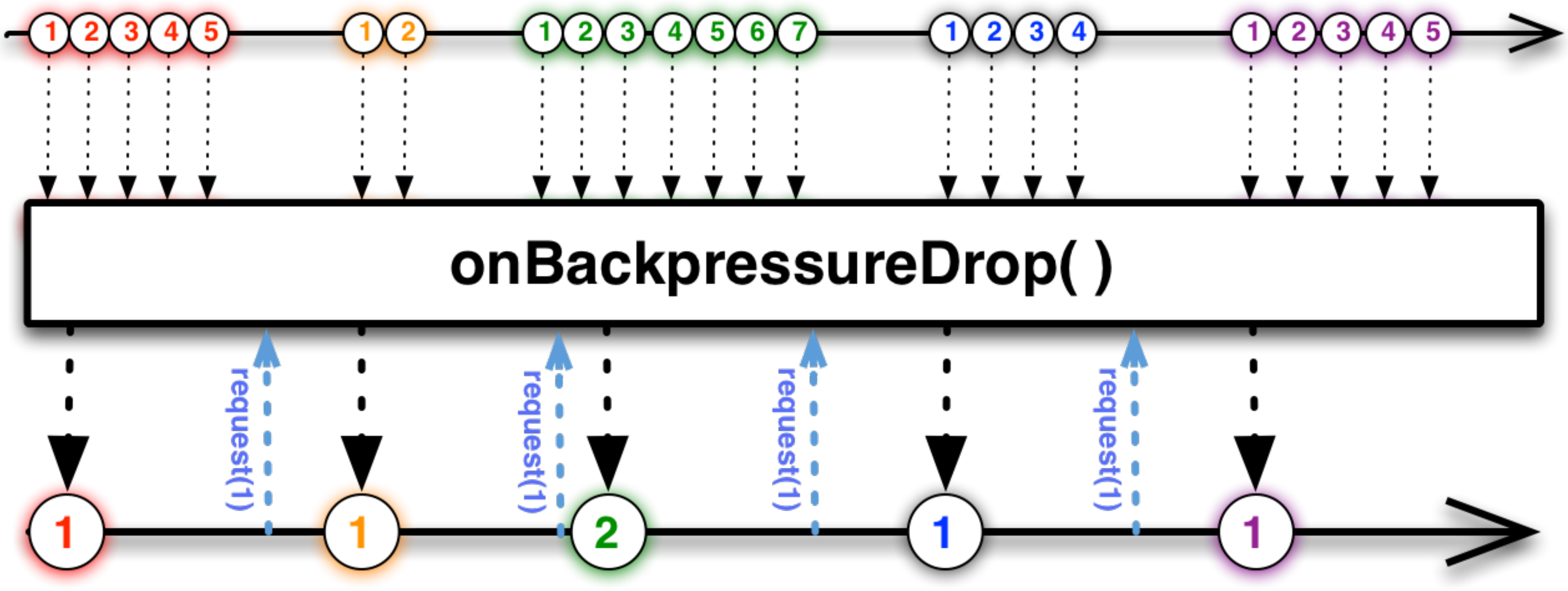
```
hotSourceStream.onBackpressureBuffer().observeOn(aScheduler);
```



```
hotSourceStream.onBackpressureBuffer().observeOn(aScheduler);
```



```
hotSourceStream.onBackpressureBuffer().observeOn(aScheduler);
```



```
hotSourceStream.onBackpressureDrop().observeOn(aScheduler);
```

stream.onBackpressure(*strategy*).subscribe

Hot Infinite Streams

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

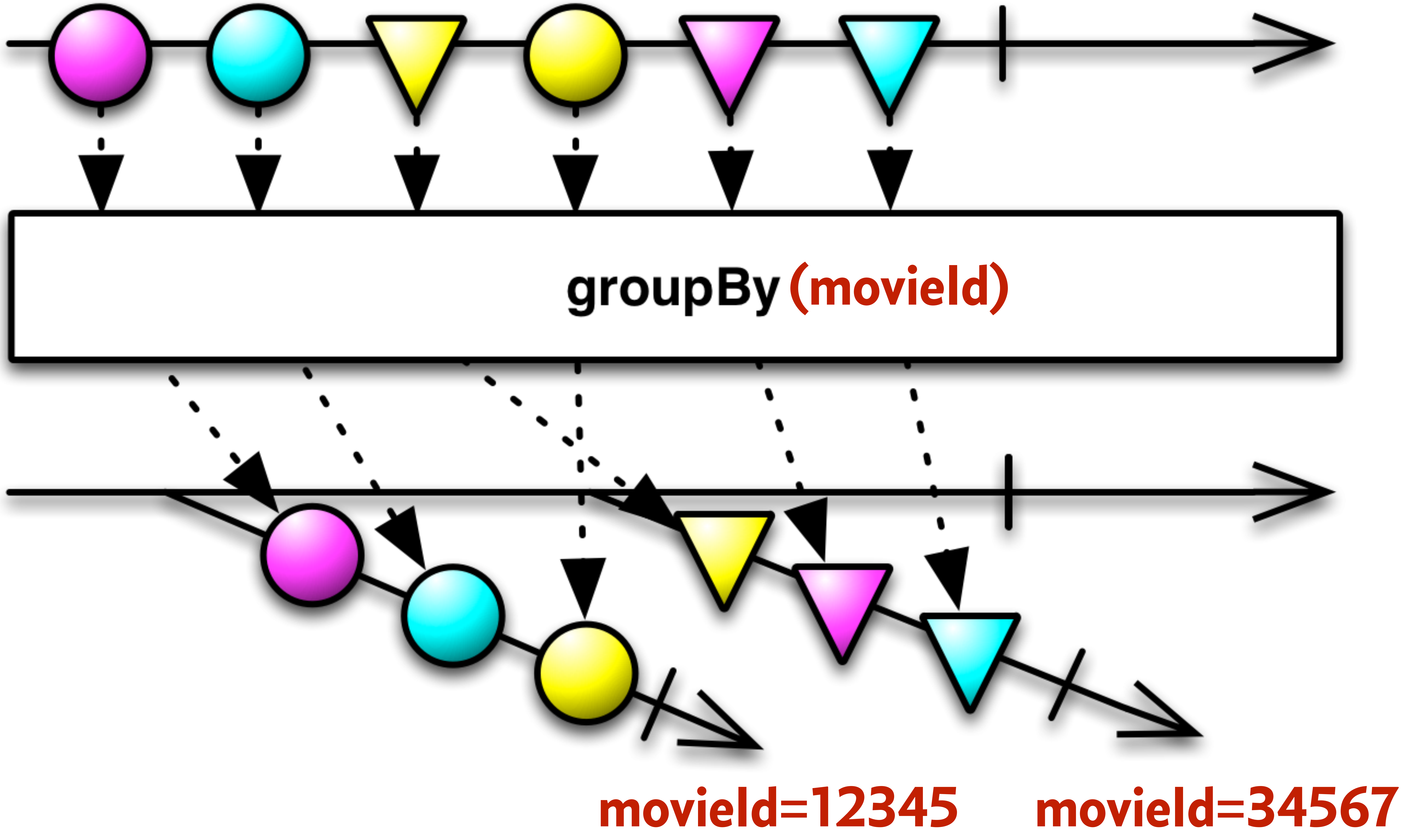
Hot Infinite Stream

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

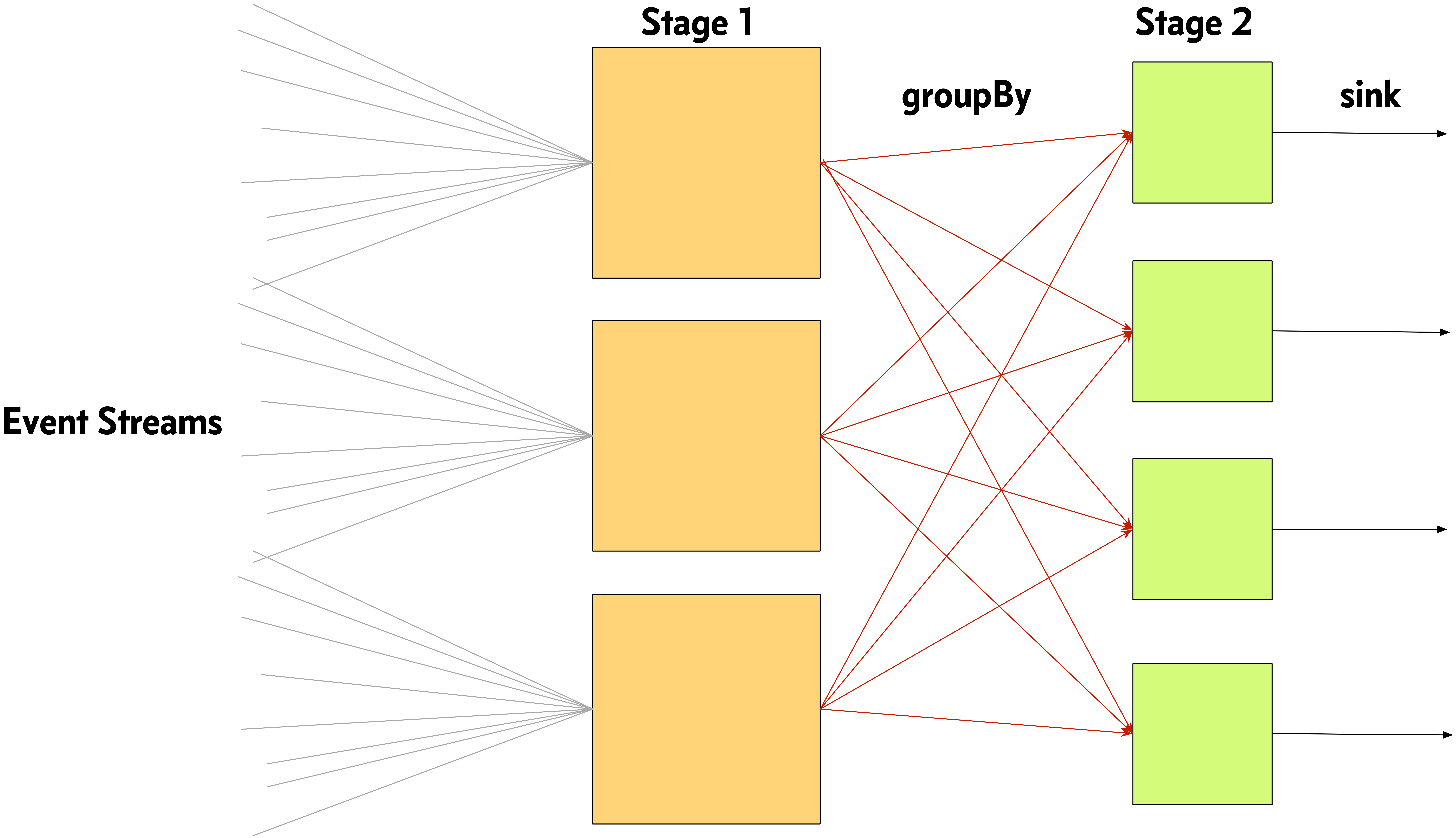
```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

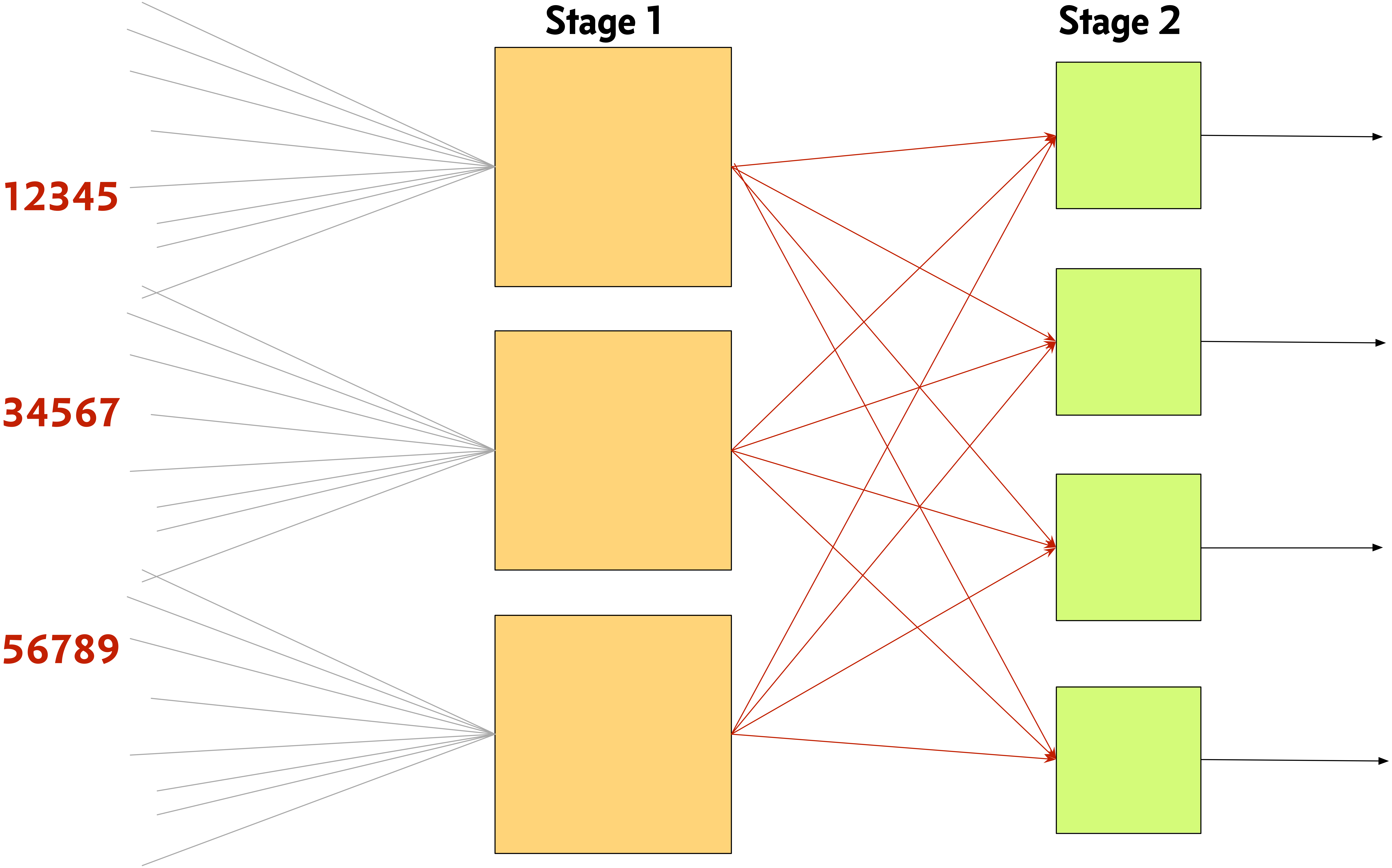
```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

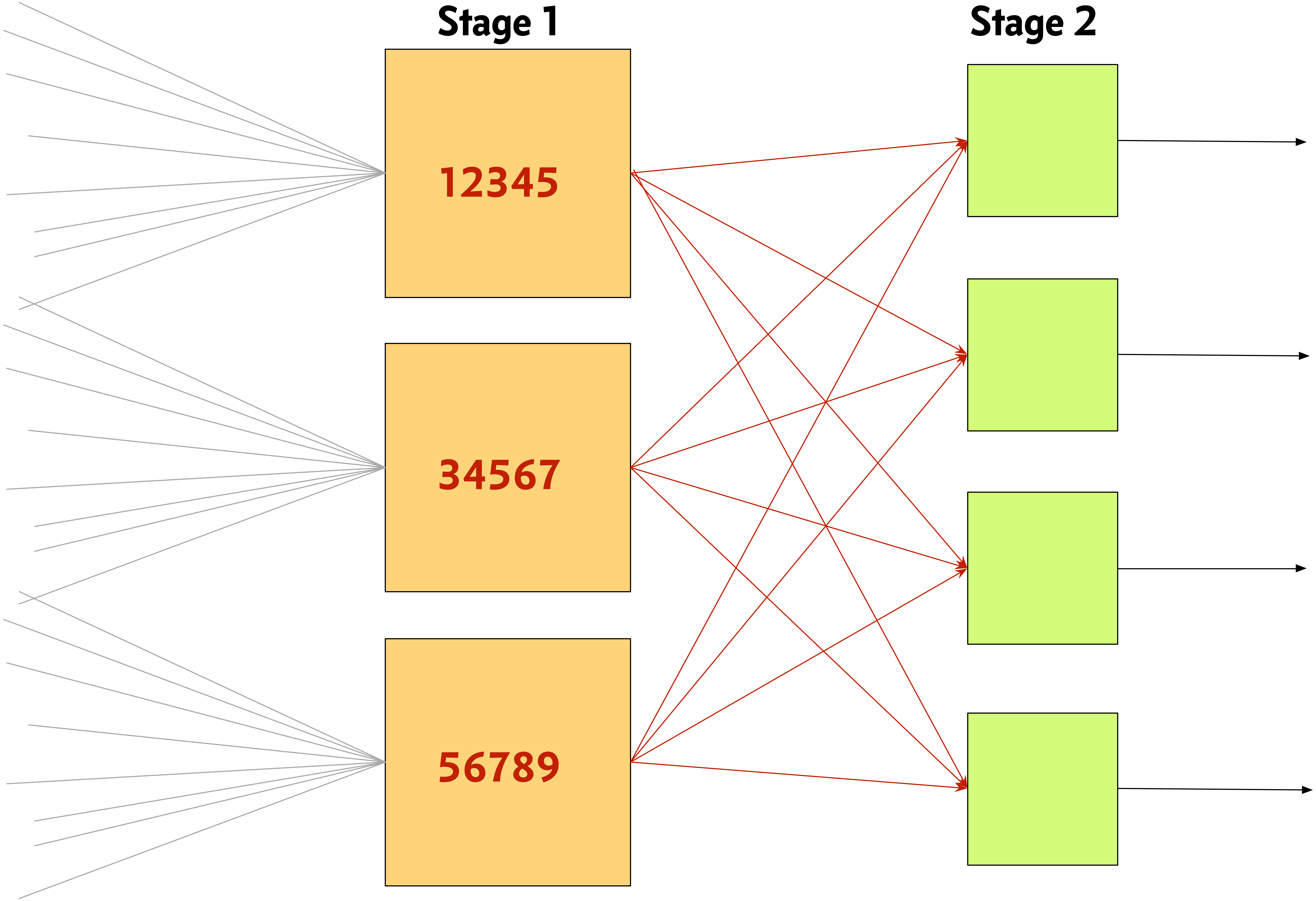

groupBy (movieId)
movieId=12345
movieId=34567

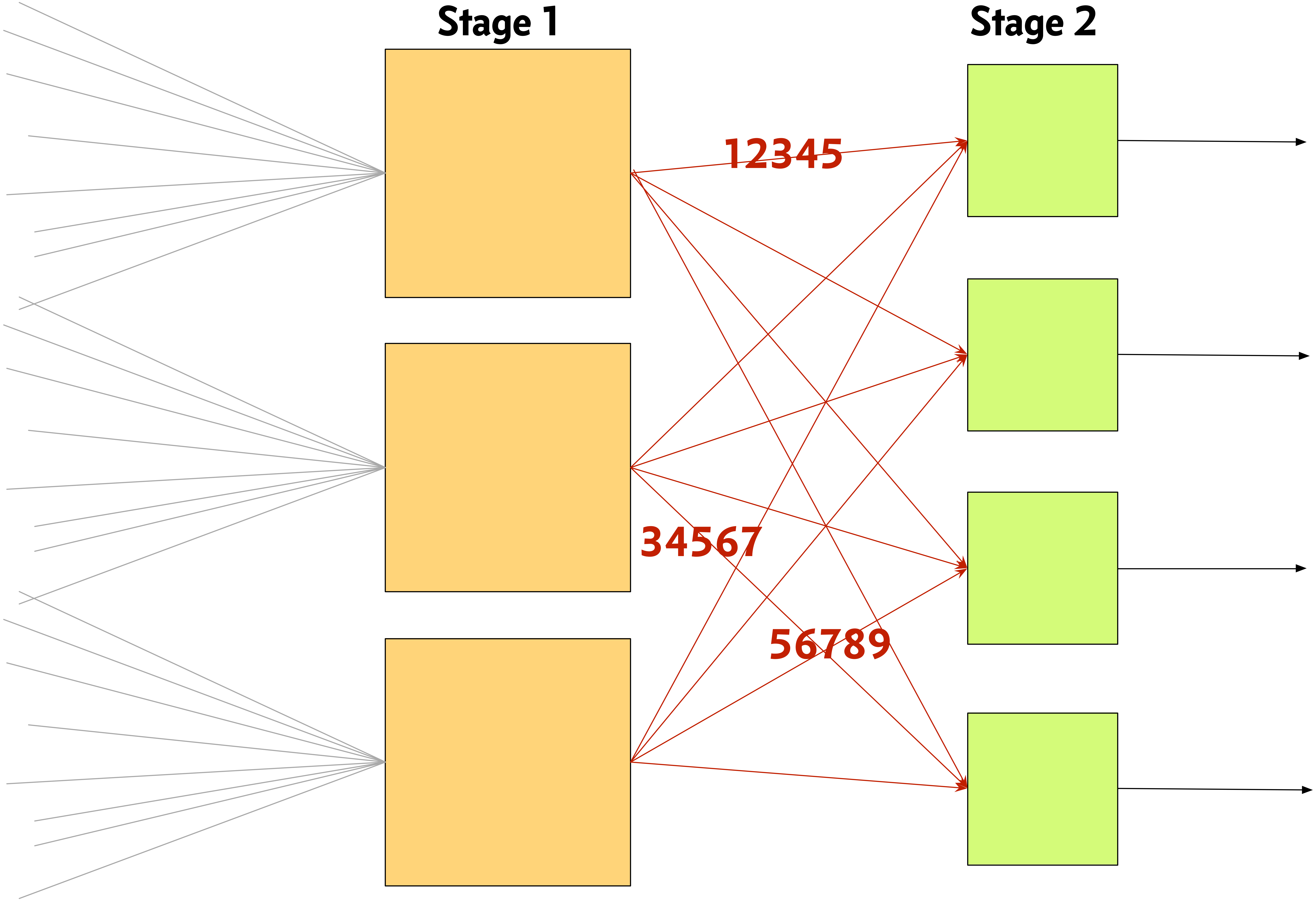
```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

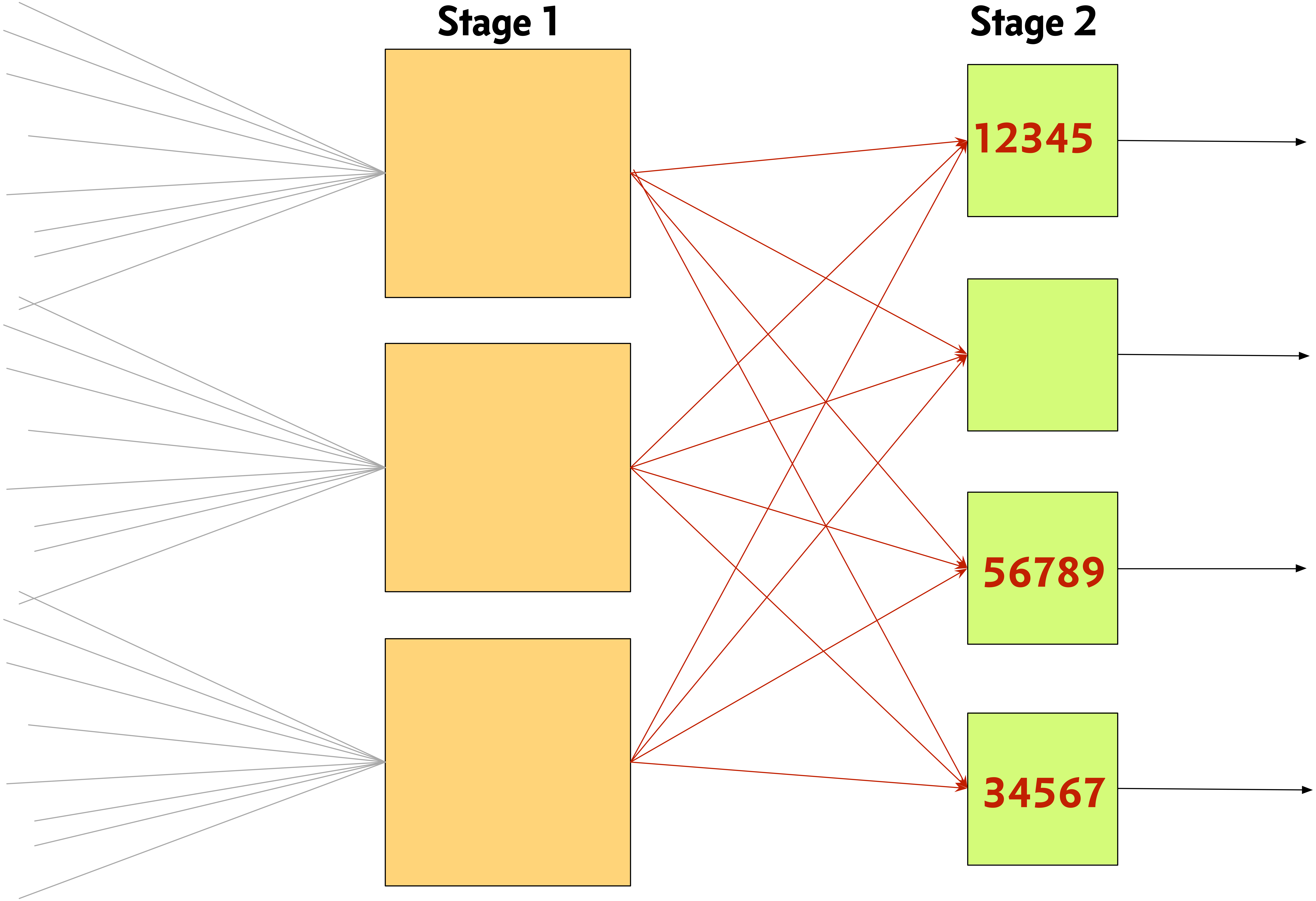
```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

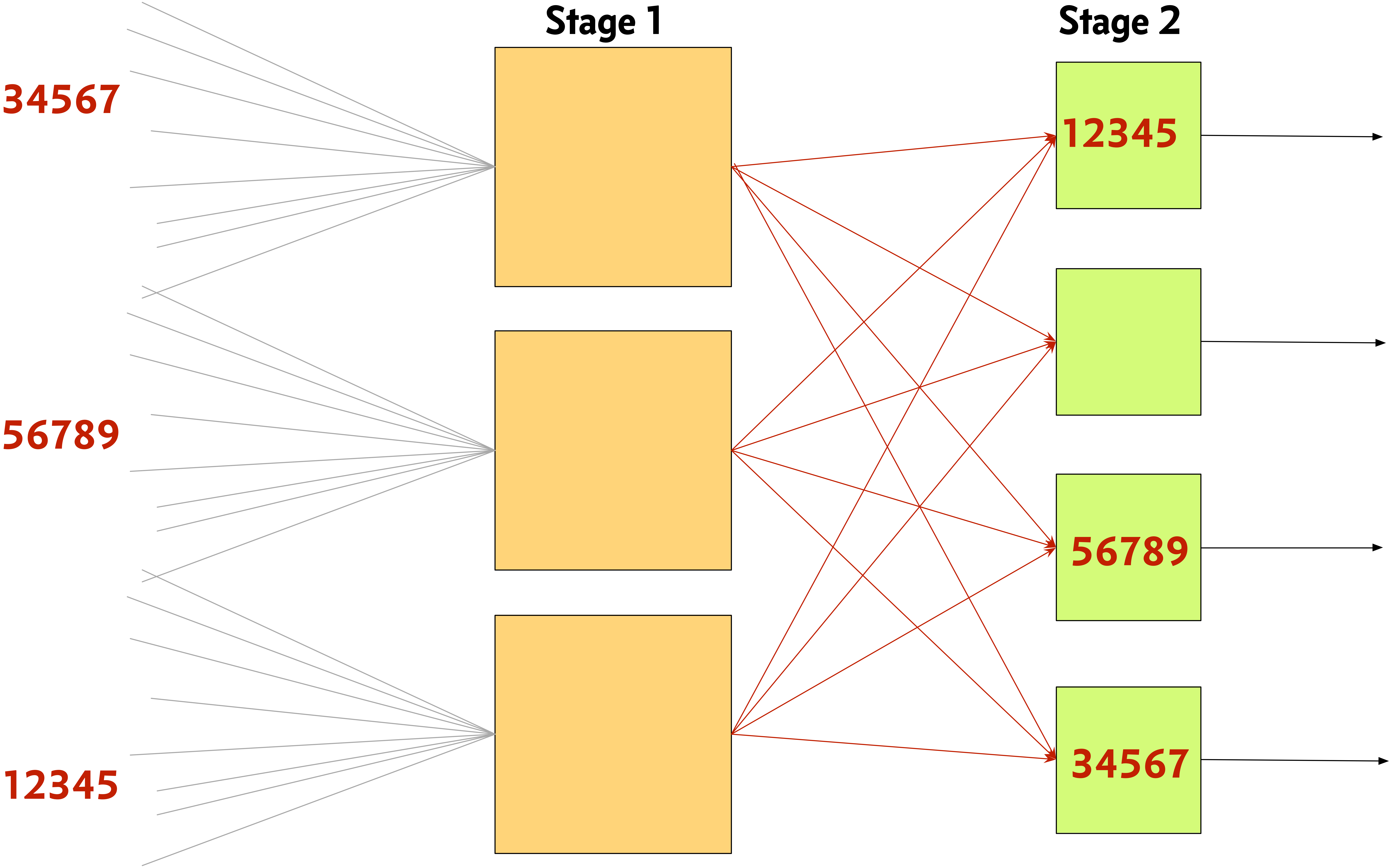

Stage 1
Stage 2
groupBy
sink
Event Streams

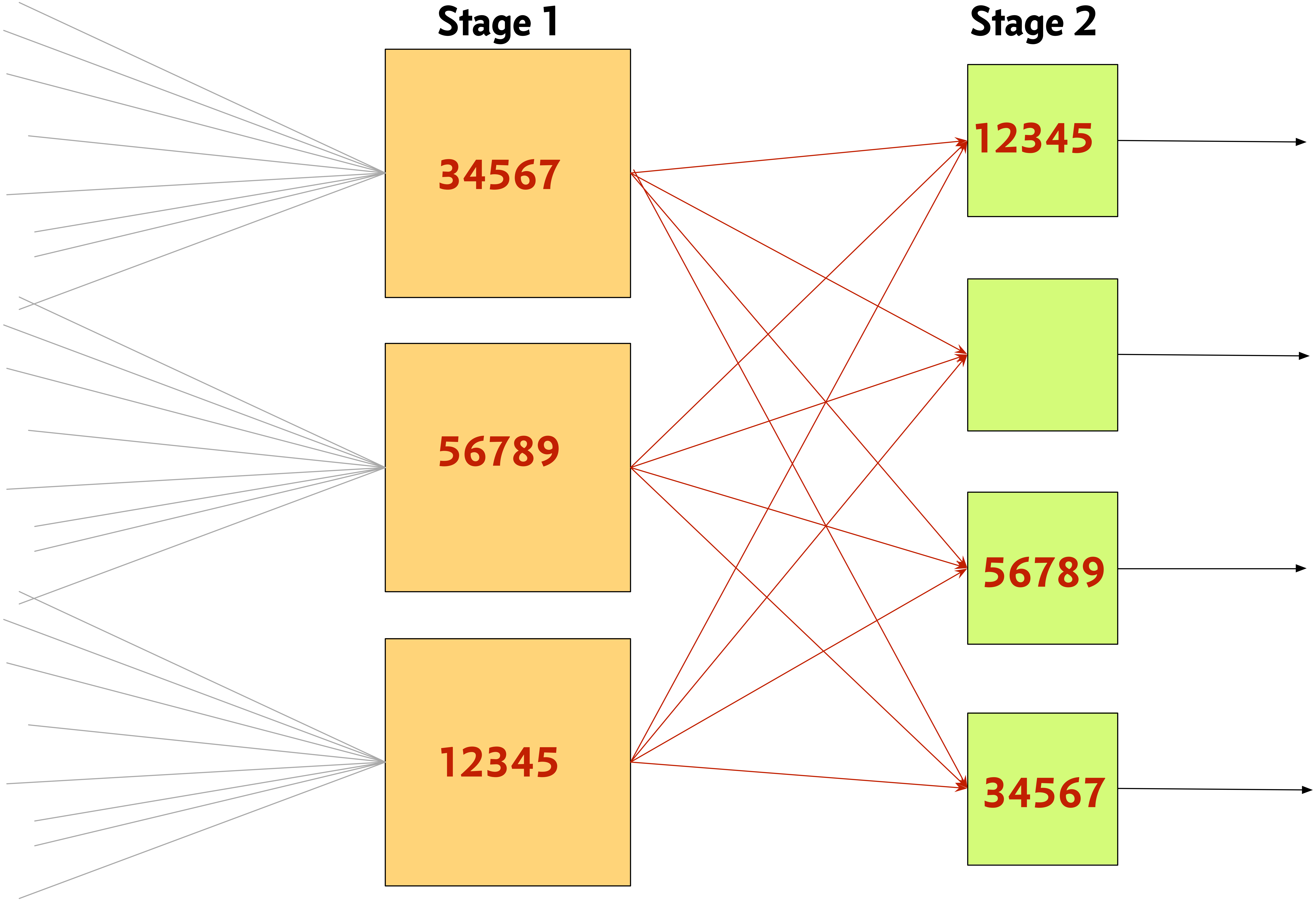

Stage 1
Stage 2
12345
34567
56789

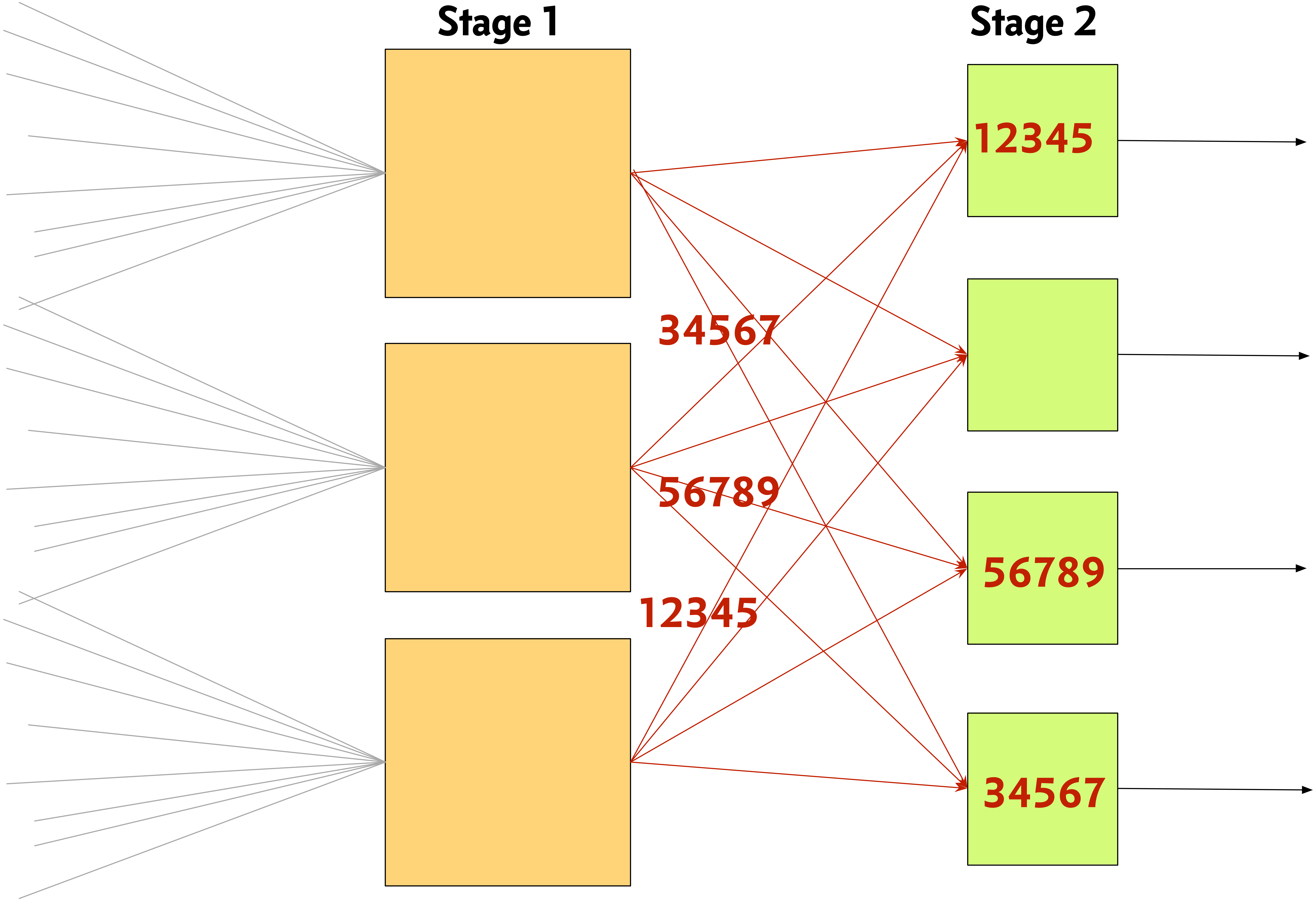

Stage 1
Stage 2
12345
34567
56789


Stage 1
Stage 2
12345
34567
56789


Stage 1
Stage 2
12345
56789
34567


Stage 1
Stage 2
34567
56789
12345
12345
56789
34567


Stage 1
Stage 2
34567
56789
12345
12345
56789
34567



Stage 1

Stage 2

12345

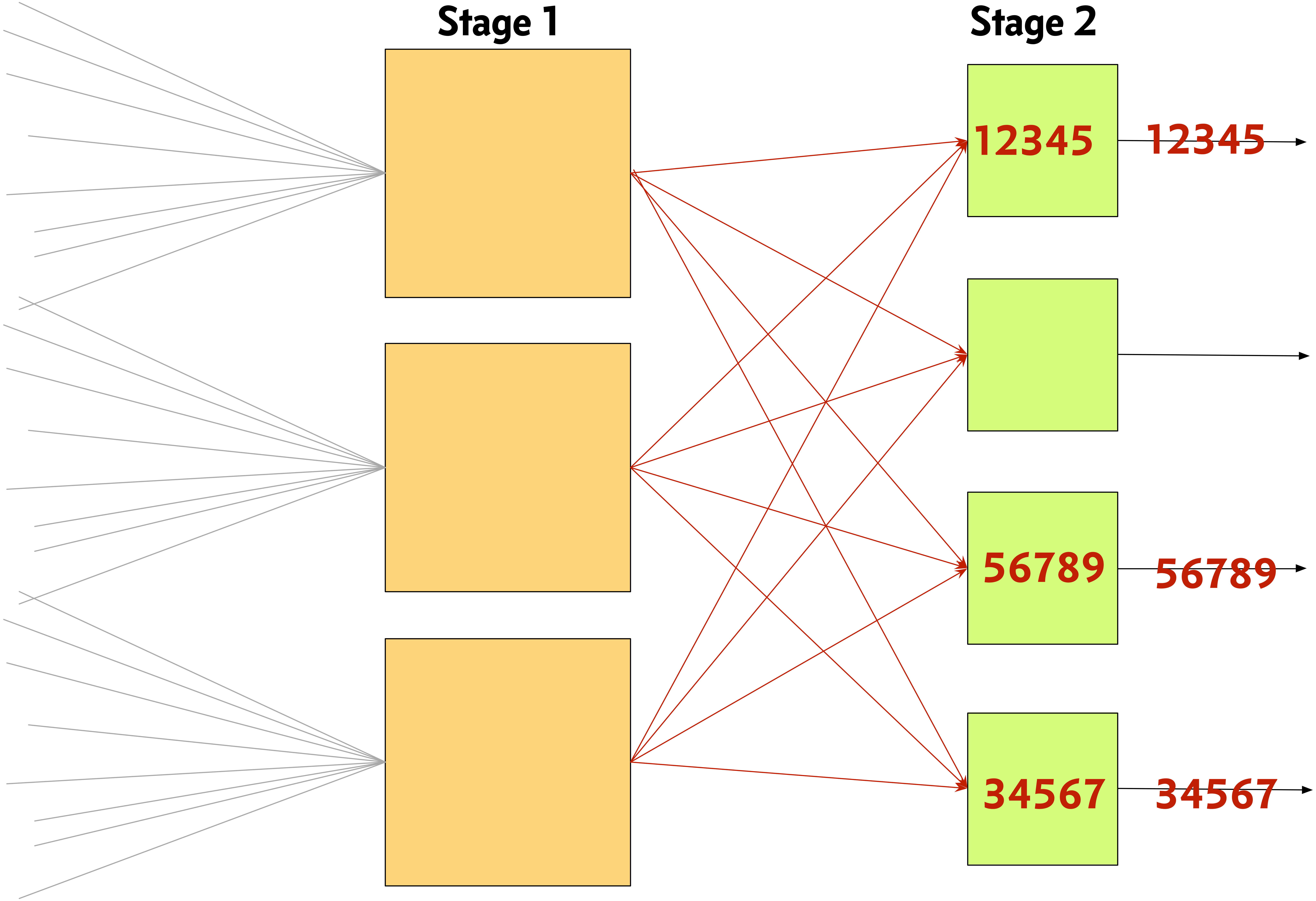
34567

56789

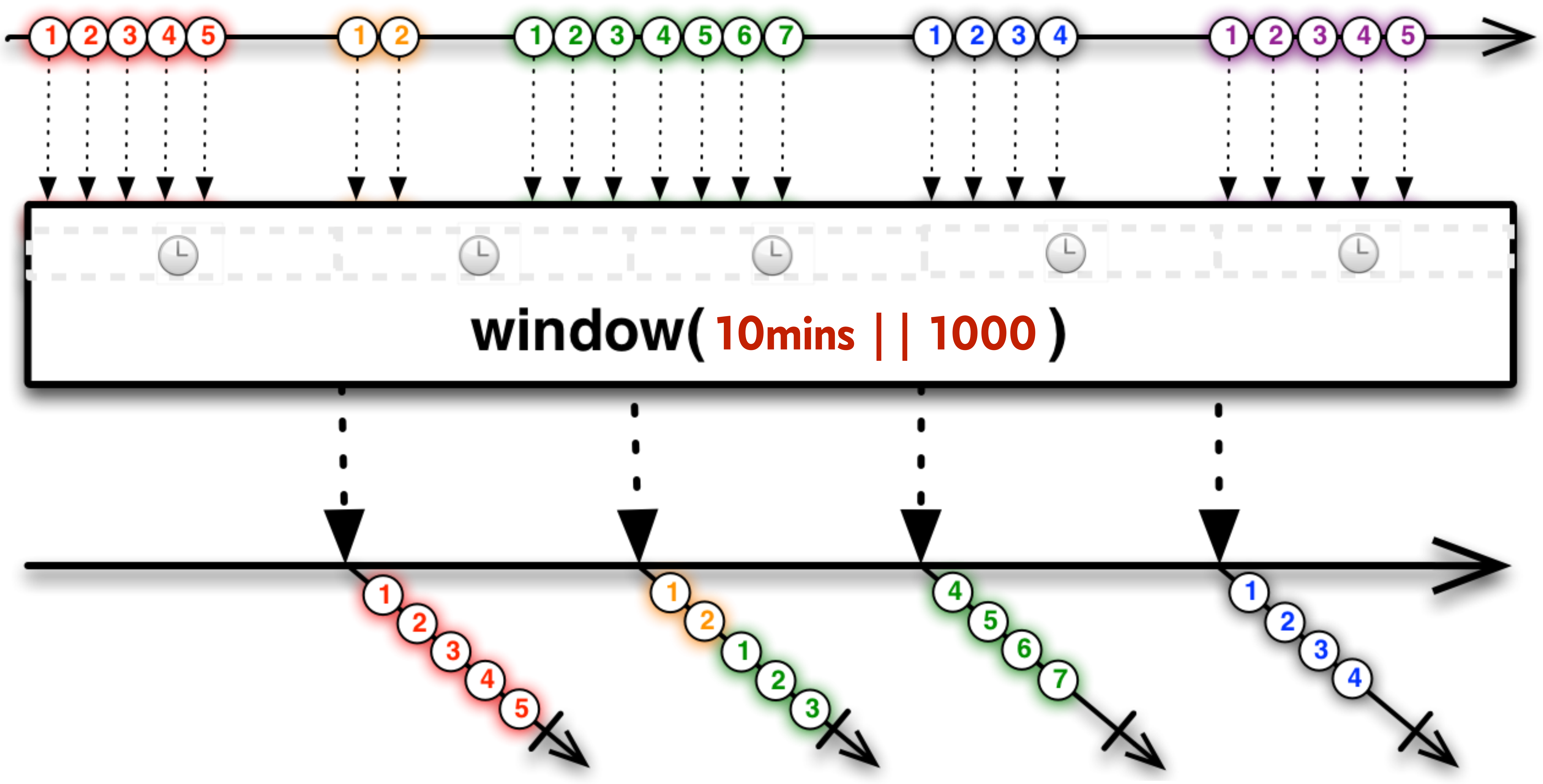
12345

56789

34567


Stage 1
Stage 2
12345
12345
56789
56789
34567
34567

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

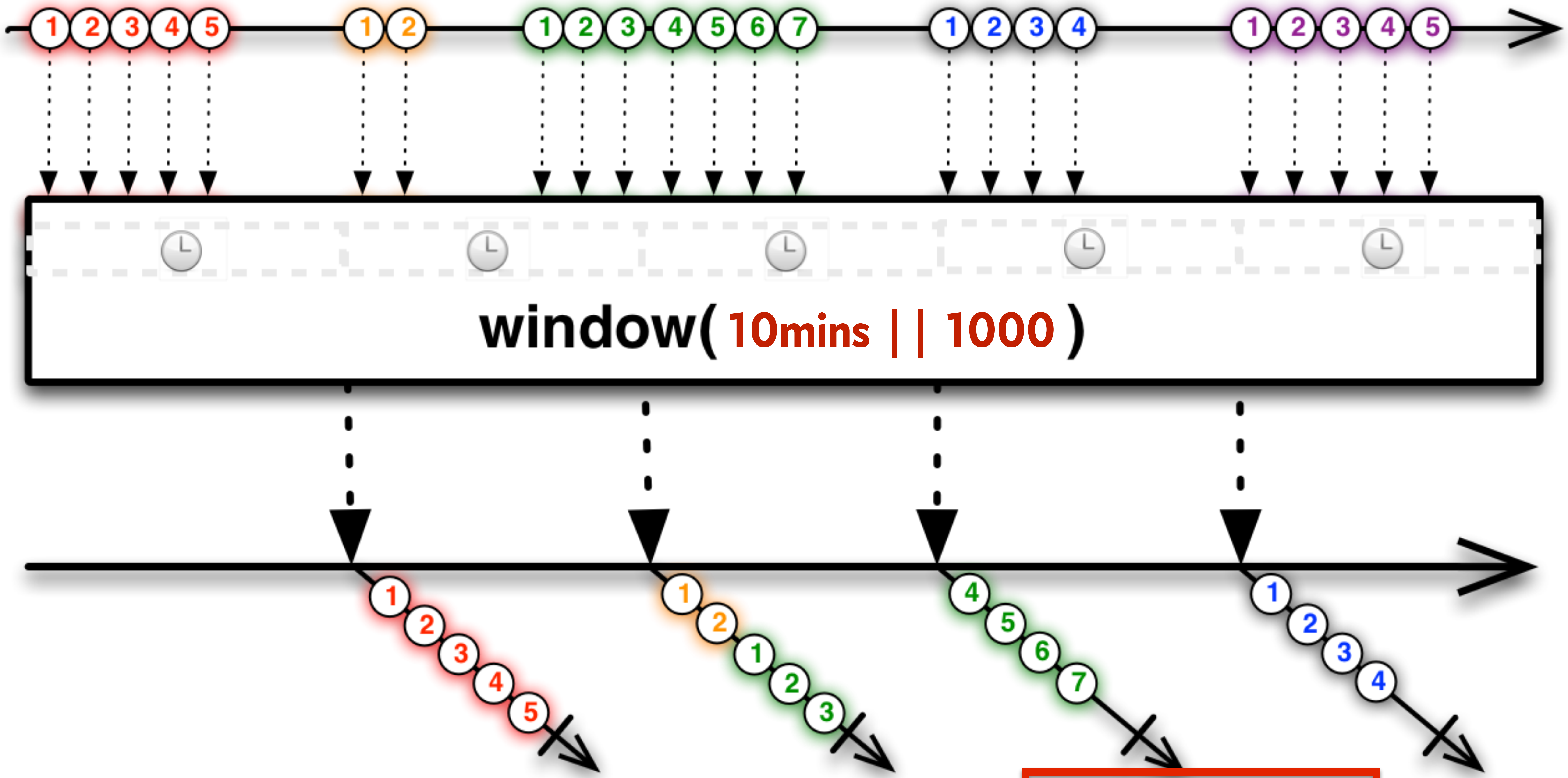


4:40-4:50pm

5:00-5:07pm
(burst to 1000)

4:50-5:00pm

5:07-5:17pm



window(10mins || 1000)

4:40-4:50pm

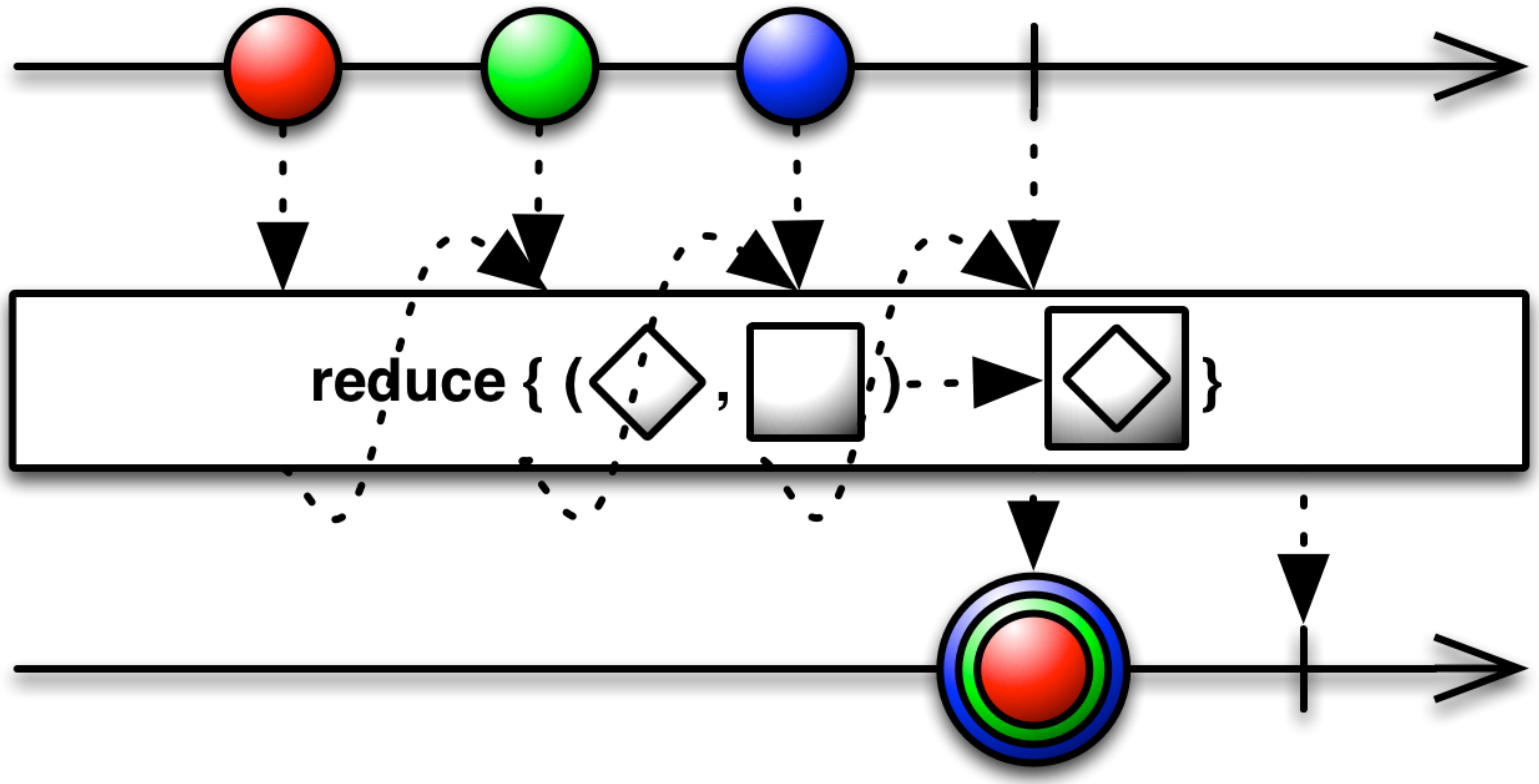
4:50-5:00pm

5:00-5:07pm
(burst to 1000)

5:07-5:17pm

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
       .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
         experiment.updateFailRatio(playAttempt);
         experiment.updateExamples(playAttempt);
         return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```


reduce { (,) }

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

```
MantisJob
.source(NetflixSources.moviePlayAttempts())
.stage(playAttempts -> {
  return playAttempts.groupBy(playAttempt -> {
    return playAttempt.getMovieId();
  })
})
.stage(playAttemptsByMovieId -> {
  playAttemptsByMovieId
  .window(10,TimeUnit.MINUTES, 1000) // buffer for 10 minutes, or 1000 play attempts
  .flatMap(windowOfPlayAttempts -> {
     return windowOfPlayAttempts
        .reduce(new FailRatioExperiment(playAttemptsByMovieId.getKey()), (experiment, playAttempt) -> {
          experiment.updateFailRatio(playAttempt);
          experiment.updateExamples(playAttempt);
          return experiment;
     }).doOnNext(experiment -> {
        logToHistorical("Play attempt experiment", experiment.getId(),experiment); // log for offline analysis
     }).filter(experiment -> {
        return experiment.failRatio() >= DYNAMIC_PROP("fail_threshold").get();
     }).map(experiment -> {
        return new FailReport(experiment, runCorrelations(experiment.getExamples()));
     }).doOnNext(report -> {
        logToHistorical("Failure report", report.getId(), report); // log for offline analysis
     })
  })
})
.sink(Sinks.emailAlert(report -> { return toEmail(report)})) // anomalies trigger events (simple email here)
```

stream.onBackpressure(*strategy?*).subscribe stream.onBackpressure(*buffer*).subscribe stream.onBackpressure(*drop*).subscribe

stream.onBackpressure(*sample*).subscribe stream.onBackpressure(*scaleHorizontally*).subscribe

Reactive-Streams

https://github.com/reactive-streams/reactive-streams

Reactive-Streams

https://github.com/reactive-streams/reactive-streams

https://github.com/ReactiveX/RxJavaReactiveStreams

```
final ActorSystem system = ActorSystem.create("InteropTest");
final FlowMaterializer mat = FlowMaterializer.create(system);

// RxJava Observable
Observable<GroupedObservable<Boolean, Integer>> oddAndEvenGroups = Observable.range(1, 1000000)
        .groupBy(i -> i % 2 == 0)
        .take(2);

Observable<String> strings = oddAndEvenGroups.<String> flatMap(group -> {
   // schedule odd and even on different event loops
   Observable<Integer> asyncGroup = group.observeOn(Schedulers.computation());

   // convert to Reactive Streams Publisher
   Publisher<Integer> groupPublisher = RxReactiveStreams.toPublisher(asyncGroup);
   // convert to Akka Streams Source and transform using Akka Streams ‘map’ and ‘take’ operators
   Source<String> stringSource = Source.from(groupPublisher).map(i -> i + " " + group.getKey()).take(2000);
   // convert back from Akka to Rx Observable
   return RxReactiveStreams.toObservable(stringSource.runWith(Sink.<String> fanoutPublisher(1, 1), mat));
});

strings.toBlocking().forEach(System.out::println);
system.shutdown();
```

```
compile 'io.reactivex:rxjava:1.0.+'
compile 'io.reactivex:rxjava-reactive-streams:0.3.0'
compile 'com.typesafe.akka:akka-stream-experimental_2.11:0.10-M1'
```

```
final ActorSystem system = ActorSystem.create("InteropTest");
final FlowMaterializer mat = FlowMaterializer.create(system);

// RxJava Observable
Observable<GroupedObservable<Boolean, Integer>> oddAndEvenGroups = Observable.range(1, 1000000)
        .groupBy(i -> i % 2 == 0)
        .take(2);

Observable<String> strings = oddAndEvenGroups.<String> flatMap(group -> {
   // schedule odd and even on different event loops
   Observable<Integer> asyncGroup = group.observeOn(Schedulers.computation());

   // convert to Reactive Streams Publisher
   Publisher<Integer> groupPublisher = RxReactiveStreams.toPublisher(asyncGroup);
   // convert to Akka Streams Source and transform using Akka Streams ‘map’ and ‘take’ operators
   Source<String> stringSource = Source.from(groupPublisher).map(i -> i + " " + group.getKey()).take(2000);
   // convert back from Akka to Rx Observable
   return RxReactiveStreams.toObservable(stringSource.runWith(Sink.<String> fanoutPublisher(1, 1), mat));
});

strings.toBlocking().forEach(System.out::println);
system.shutdown();
```

```
compile 'io.reactivex:rxjava:1.0.+'
compile 'io.reactivex:rxjava-reactive-streams:0.3.0'
compile 'com.typesafe.akka:akka-stream-experimental_2.11:0.10-M1'
```

```
// RxJava Observable
Observable<GroupedObservable<Boolean, Integer>> oddAndEvenGroups = Observable.range(1, 1000000)
        .groupBy(i -> i % 2 == 0)
        .take(2);

Observable<String> strings = oddAndEvenGroups.<String> flatMap(group -> {
   // schedule odd and even on different event loops
   Observable<Integer> asyncGroup = group.observeOn(Schedulers.computation());

   // convert to Reactive Streams Publisher
   Publisher<Integer> groupPublisher = RxReactiveStreams.toPublisher(asyncGroup);

   // Convert to Reactor Stream and transform using Reactor Stream ‘map’ and ‘take’ operators
   Stream<String> linesStream = Streams.create(groupPublisher).map(i -> i + " " + group.getKey()).take(2000);

   // convert back from Reactor Stream to Rx Observable
   return RxReactiveStreams.toObservable(linesStream);
});

strings.toBlocking().forEach(System.out::println);
```

```
compile 'io.reactivex:rxjava:1.0.+'
compile 'io.reactivex:rxjava-reactive-streams:0.3.0'
compile 'org.projectreactor:reactor-core:2.0.0.M1'
```

```
// RxJava Observable
Observable<GroupedObservable<Boolean, Integer>> oddAndEvenGroups = Observable.range(1, 1000000)
       .groupBy(i -> i % 2 == 0)
       .take(2);

Observable<String> strings = oddAndEvenGroups.<String> flatMap(group -> {
   // schedule odd and even on different event loops
   Observable<Integer> asyncGroup = group.observeOn(Schedulers.computation());

   // convert to Reactive Streams Publisher
   Publisher<Integer> groupPublisher = RxReactiveStreams.toPublisher(asyncGroup);

   // Convert to Reactor Stream and transform using Reactor Stream ‘map’ and ‘take’ operators
   Stream<String> linesStream = Streams.create(groupPublisher).map(i -> i + " " + group.getKey()).take(2000);

   // convert back from Reactor Stream to Rx Observable
   return RxReactiveStreams.toObservable(linesStream);
});

strings.toBlocking().forEach(System.out::println);
```

```
compile 'io.reactivex:rxjava:1.0.+'
compile 'io.reactivex:rxjava-reactive-streams:0.3.0'
compile 'org.projectreactor:reactor-core:2.0.0.M1'
```

```java
try {
    RatpackServer server = EmbeddedApp.fromHandler(ctx -> {
        Observable<String> o1 = Observable.range(0, 2000)
                .observeOn(Schedulers.computation()).map(i -> {
                    return "A " + i;
                });

        Observable<String> o2 = Observable.range(0, 2000)
                .observeOn(Schedulers.computation()).map(i -> {
                    return "B " + i;
                });

        Observable<String> o = Observable.merge(o1, o2);

        ctx.render(
                ServerSentEvents.serverSentEvents(RxReactiveStreams.toPublisher(o), e ->
                        e.event("counter").data("event " + e.getItem()))
                );
    }).getServer();

    server.start();
    System.out.println("Port: " + server.getBindPort());
} catch (Exception e) {
    e.printStackTrace();
}
```

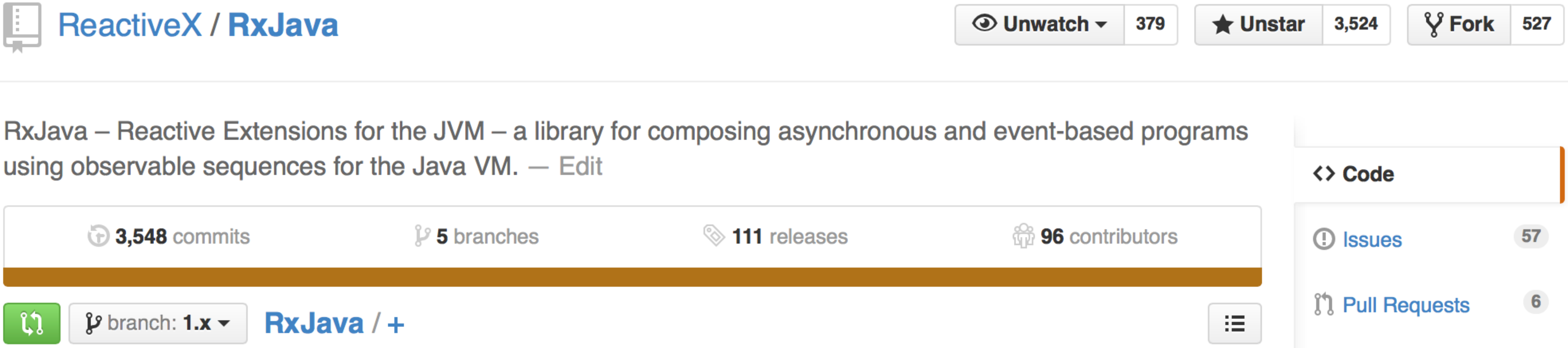
```groovy
compile 'io.reactivex:rxjava:1.0.+'
compile 'io.reactivex:rxjava-reactive-streams:0.3.0'
compile 'io.ratpack:ratpack-rx:0.9.10'
```

```
try {
    RatpackServer server = EmbeddedApp.fromHandler(ctx -> {
        Observable<String> o1 = Observable.range(0, 2000)
                .observeOn(Schedulers.computation()).map(i -> {
                    return "A " + i;
                });

        Observable<String> o2 = Observable.range(0, 2000)
                .observeOn(Schedulers.computation()).map(i -> {
                    return "B " + i;
                });

        Observable<String> o = Observable.merge(o1, o2);

        ctx.render(
                ServerSentEvents.serverSentEvents(RxReactiveStreams.toPublisher(o), e ->
                        e.event("counter").data("event " + e.getItem()))
                );
    }).getServer();

    server.start();
    System.out.println("Port: " + server.getBindPort());
} catch (Exception e) {
    e.printStackTrace();
}
```

```
compile 'io.reactivex:rxjava:1.0.+'
compile 'io.reactivex:rxjava-reactive-streams:0.3.0'
compile 'io.ratpack:ratpack-rx:0.9.10'
```

RxJava 1.0 Final

November 18th



ReactiveX / RxJava

RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM. — Edit

3,548 commits
5 branches
111 releases
96 contributors

Mental Shift

imperative → functional

sync → async

pull → push

Concurrency and async are non-trivial.
Rx doesn’t trivialize it.

Rx is powerful and rewards those
who go through the learning curve.

	Single	Multiple
Sync	**T** getData()	**Iterable**<T> getData() **Stream**<T> getData()
Async	**Future**<T> getData()	**Observable<T>** getData()

Abstract Concurrency

Non-Opinionated Concurrency

Decouple Production from Consumption

Powerful Composition of Nested, Conditional Flows

First-class Support of Error Handling, Scheduling & Flow Control

ReactiveX

An API for asynchronous programming with observable streams

RxJava

http://github.com/ReactiveX/RxJava
http://reactivex.io

NETFLIX

jobs.netflix.com

Reactive Programming 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

Reactive Extensions (Rx) http://www.reactivex.io

Reactive Streams https://github.com/reactive-streams/reactive-streams

RxJava

https://github.com/ReactiveX/RxJava
@RxJava

RxJS

http://reactive-extensions.github.io/RxJS/
@ReactiveX

Ben Christensen
@benjchristensen