Multicore Programming

Reactive Programming

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Bureau 414C

Master 1 computer science – Semester 8

Description

Reactive programming is programming with asynchronous data streams:

- Asynchronous like `CompleteableFuture`.
- Data streams like `Stream`.

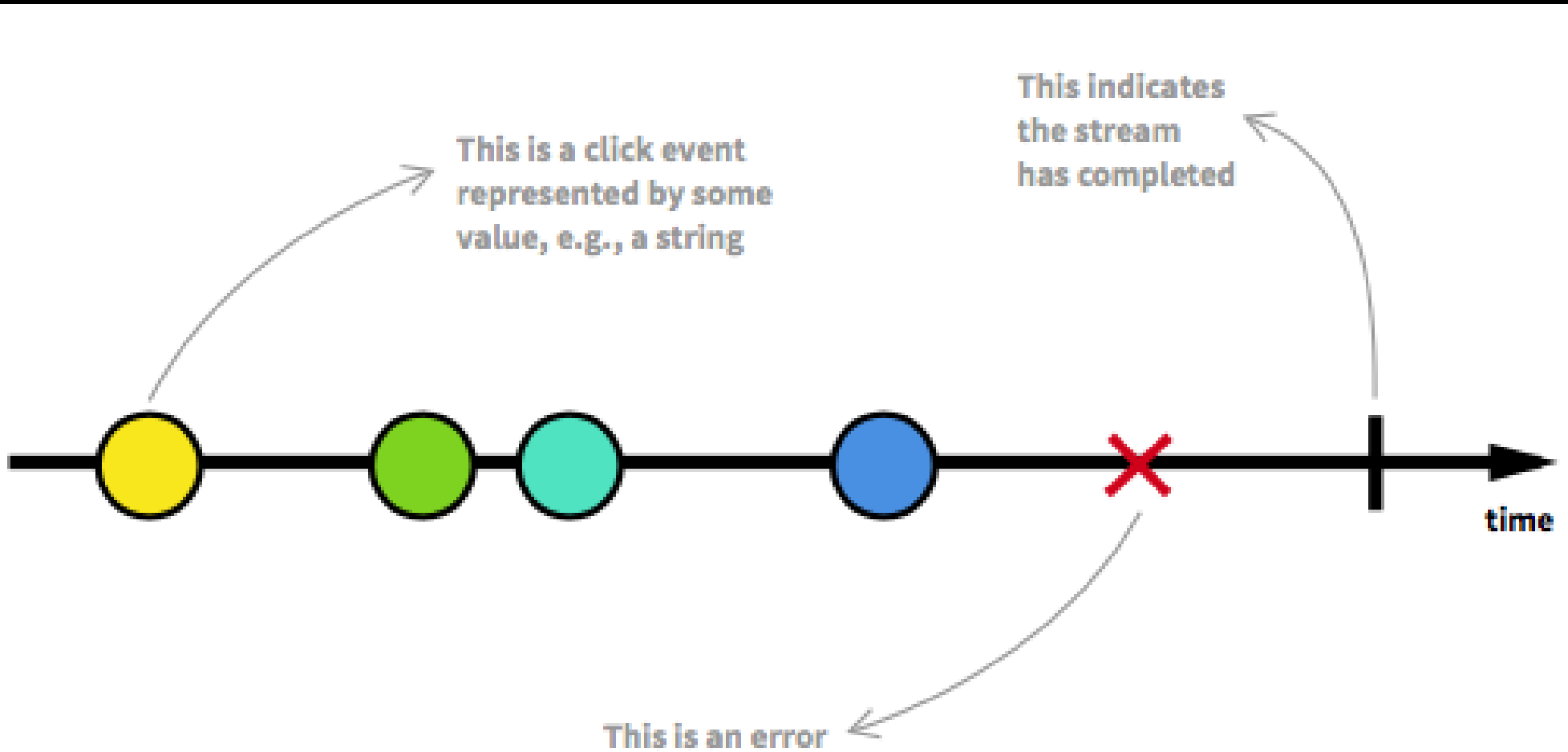
Outline

Outline

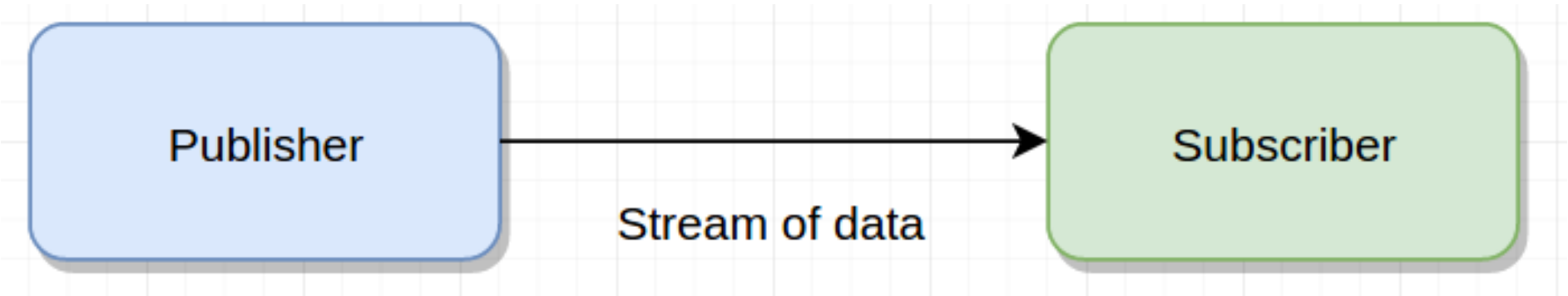
Relation with Existing API

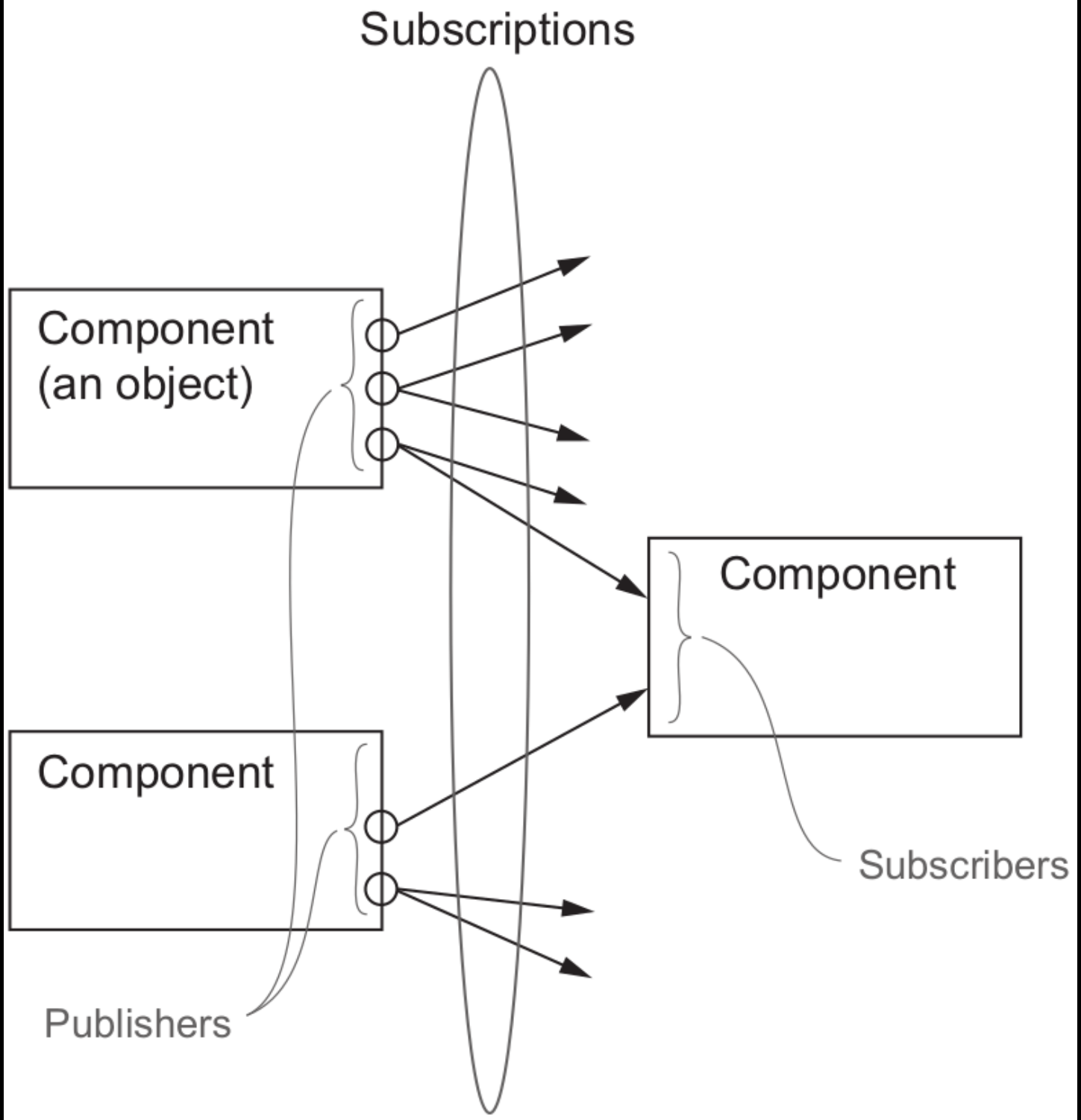
	single item	multiple items
synchronous	`T data`	`Stream<T> stream`
asynchronous	`Compl.Future<T> fut`	`Publisher<T> pub`

- A future/promise is to a value as publishers are to iterables/collections.
- Reacting to the completion of multiple futures (asynchronously, without blocking operation).

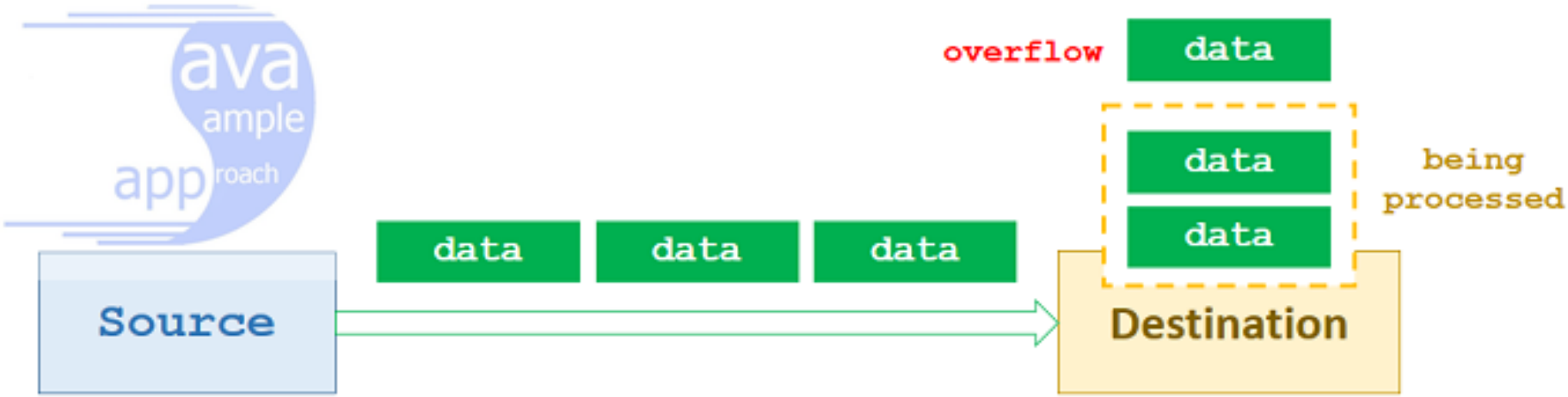

This is a click event
represented by some
value, e.g., a string
This indicates
the stream
has completed
time
This is an error

Publish/Subscribe Model




Subscriptions
Component
(an object)
Component
Component
Subscribers
Publishers

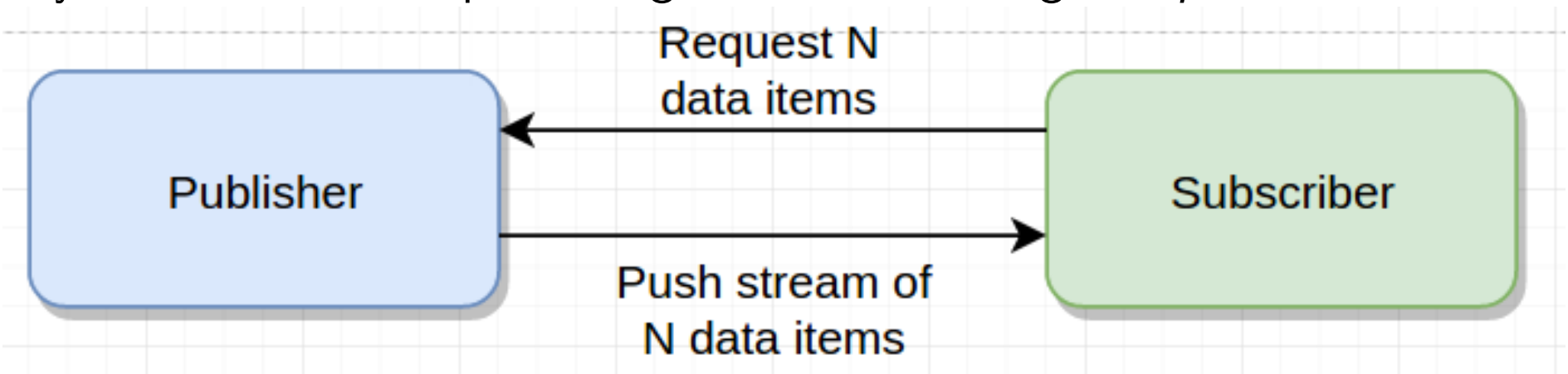
Overflow Problem



- Too much *pressure* with push-based methods.
- Reactive programming is thus "pull-based" to control the pressure.

Backpressure

Asynchronous stream processing with non-blocking *backpressure*:



Other Characteristics

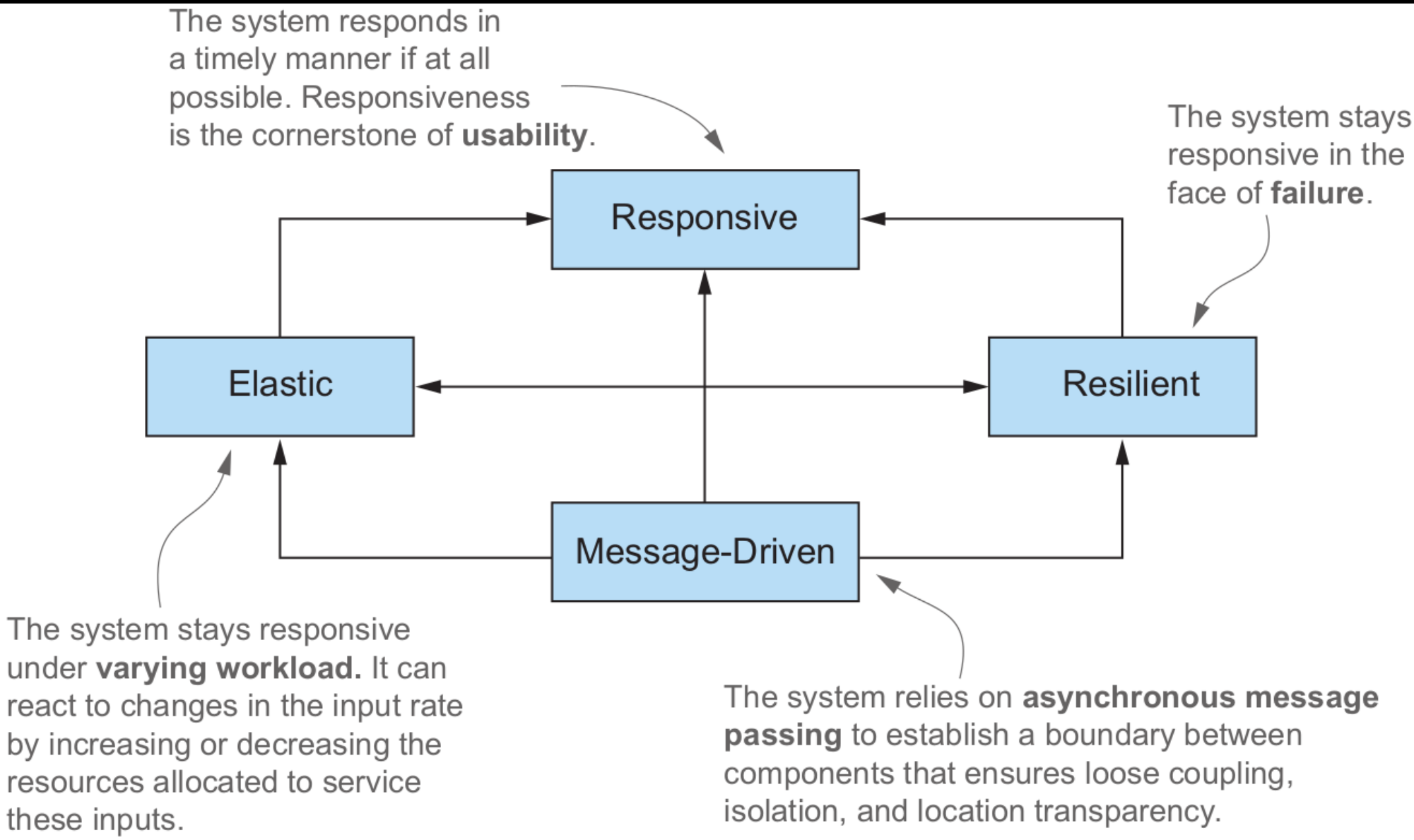
Composable	chaining operations (like stream and completable future).
Lazy evaluation	evaluated as late as possible (like stream).
Asynchronous	non-blocking operations (like completable future).
Reusable/Cacheable	results can be reused (like completable future).
Push-based	the data source initiates the processing (pull-based for stream).
Message passing	data producers exchange messages containing the data (no shared-memory mechanism).

History and Technological Context

- Concepts exist since 1960/1970.
- Kind of dataflow programming
- Also called FRP (Functional Reactive Programming): functional, no side effect, immutable state, pure function.
- Related to observer and iterator design patterns.
- Close to actor concurrency model.

Relation with Reactive Systems

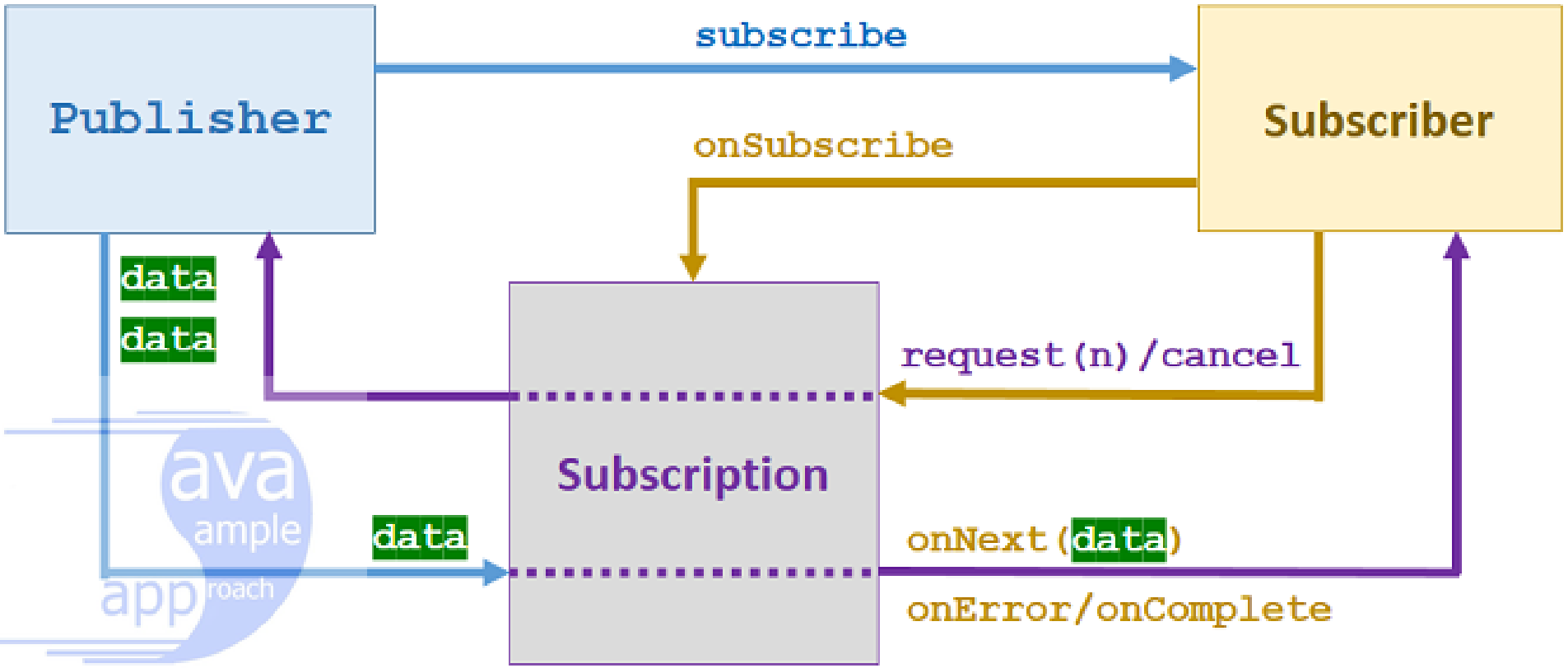
- Reactive programming can be used to build a reactive system.
- The Reactive Manifesto states the features that must be offered by a reactive system: responsive, resilient, elastic, message-driven.

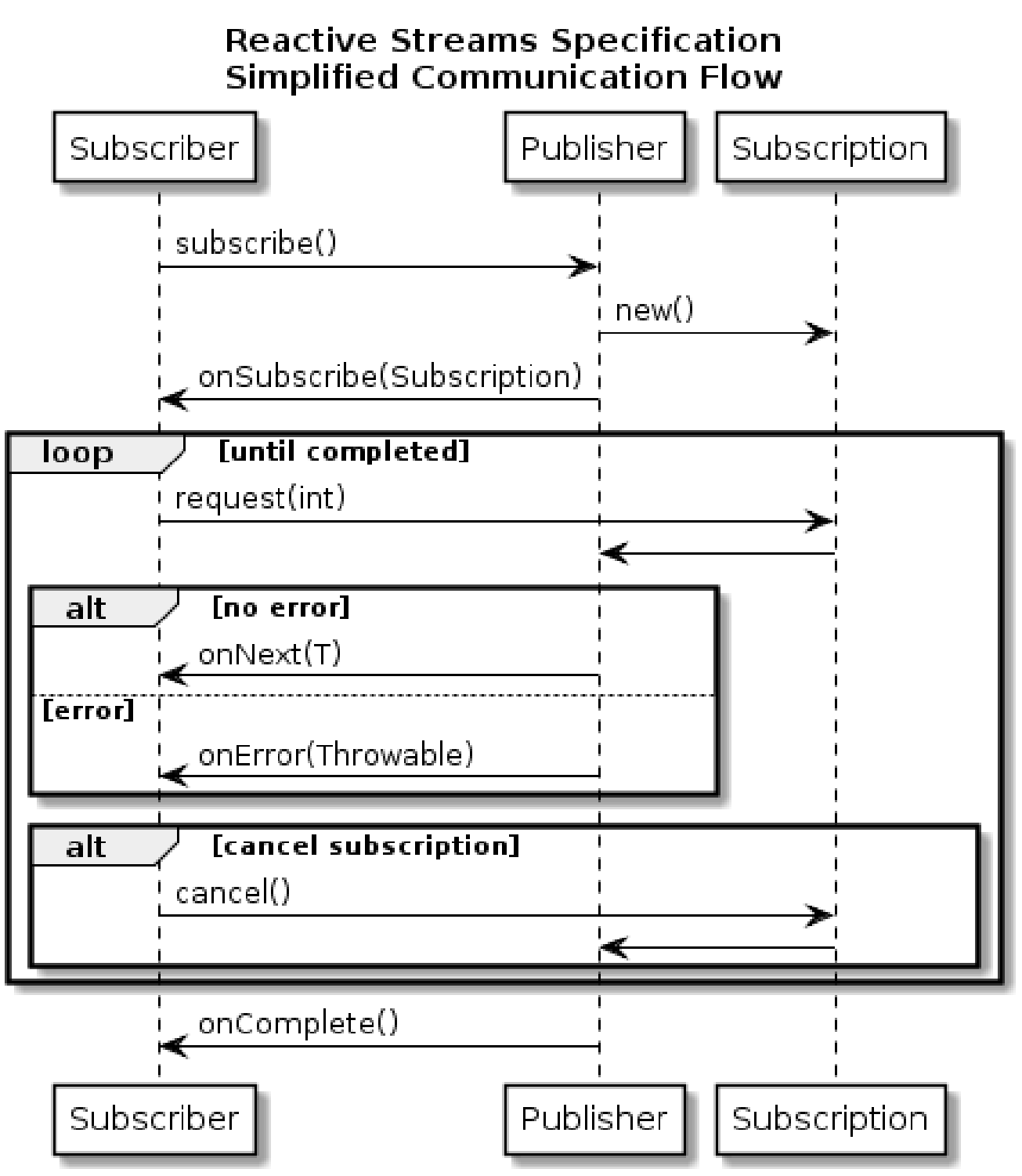

The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of **usability**.
Responsive
The system stays responsive in the face of **failure**.
Elastic
Resilient
Message-Driven
The system stays responsive under **varying workload.** It can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs.
The system relies on **asynchronous message passing** to establish a boundary between components that ensures loose coupling, isolation, and location transparency.

Outline

Main Interfaces

```
interface Publisher<T> {
    void subscribe(Subscriber<T> subscriber);
}
interface Subscriber<T> {
    void onSubscribe(Subscription subscription);
    void onNext(T item);
    void onError(Throwable throwable);
    void onComplete();
}
interface Subscription {
    void request(long n);
    void cancel();
}
```


Publisher
subscribe
Subscriber
onSubscribe
data
data
request(n)/cancel
Subscription
data
onNext(data)
onError/onComplete


Reactive Streams Specification
Simplified Communication Flow
Subscriber
Publisher
Subscription
subscribe()
new()
onSubscribe(Subscription)
loop
[until completed]
request(int)
alt
[no error]
onNext(T)
[error]
onError(Throwable)
alt
[cancel subscription]
cancel()
onComplete()
Subscriber
Publisher
Subscription

Dual to Iterable

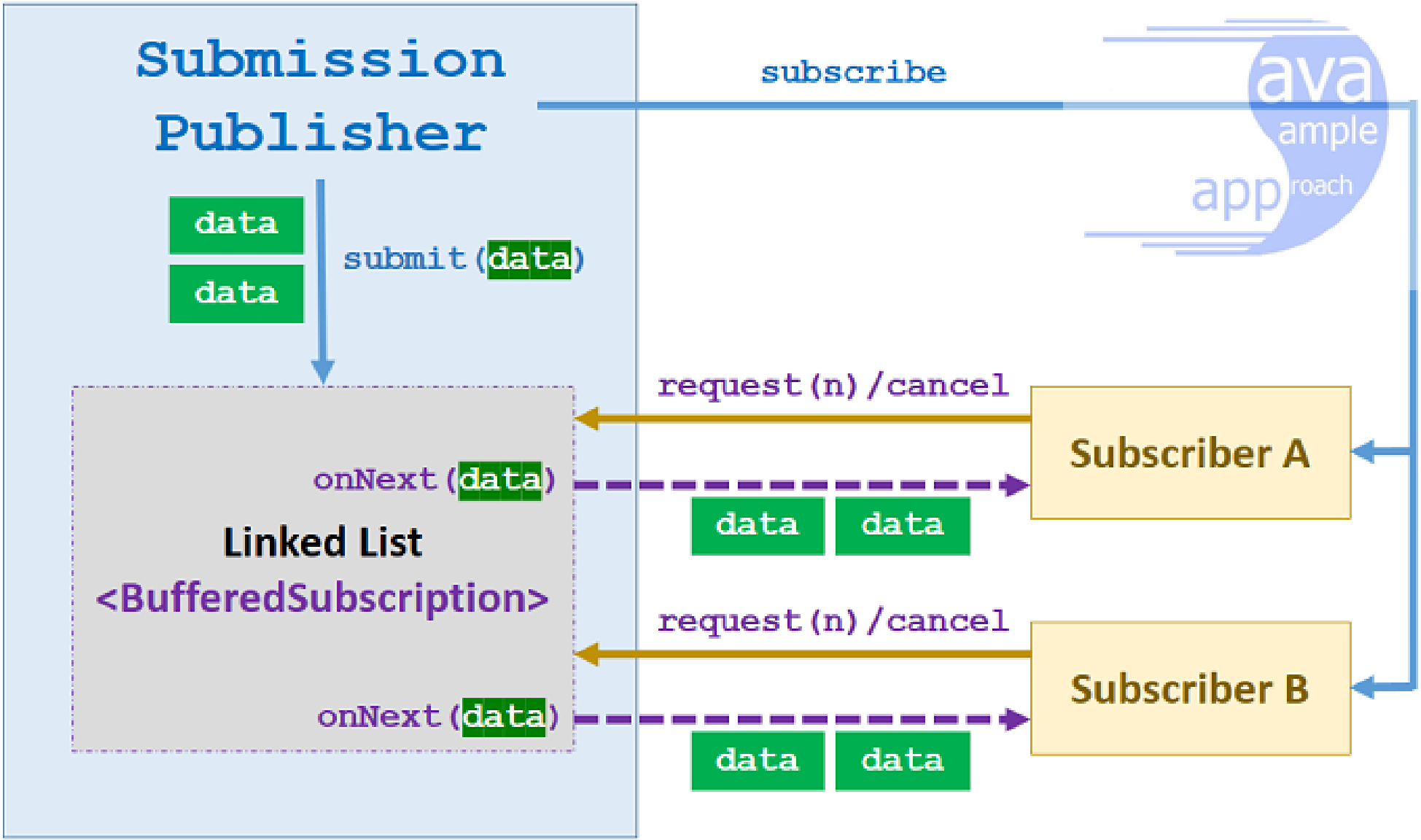
Event	Iterable (pull)	Observable (push)
retrieve data	`T next()`	`onNext(T)`
discover error	`throws Exception`	`onError(Exception)`
complete	`!hasNext()`	`onCompleted()`

Outline

Interface

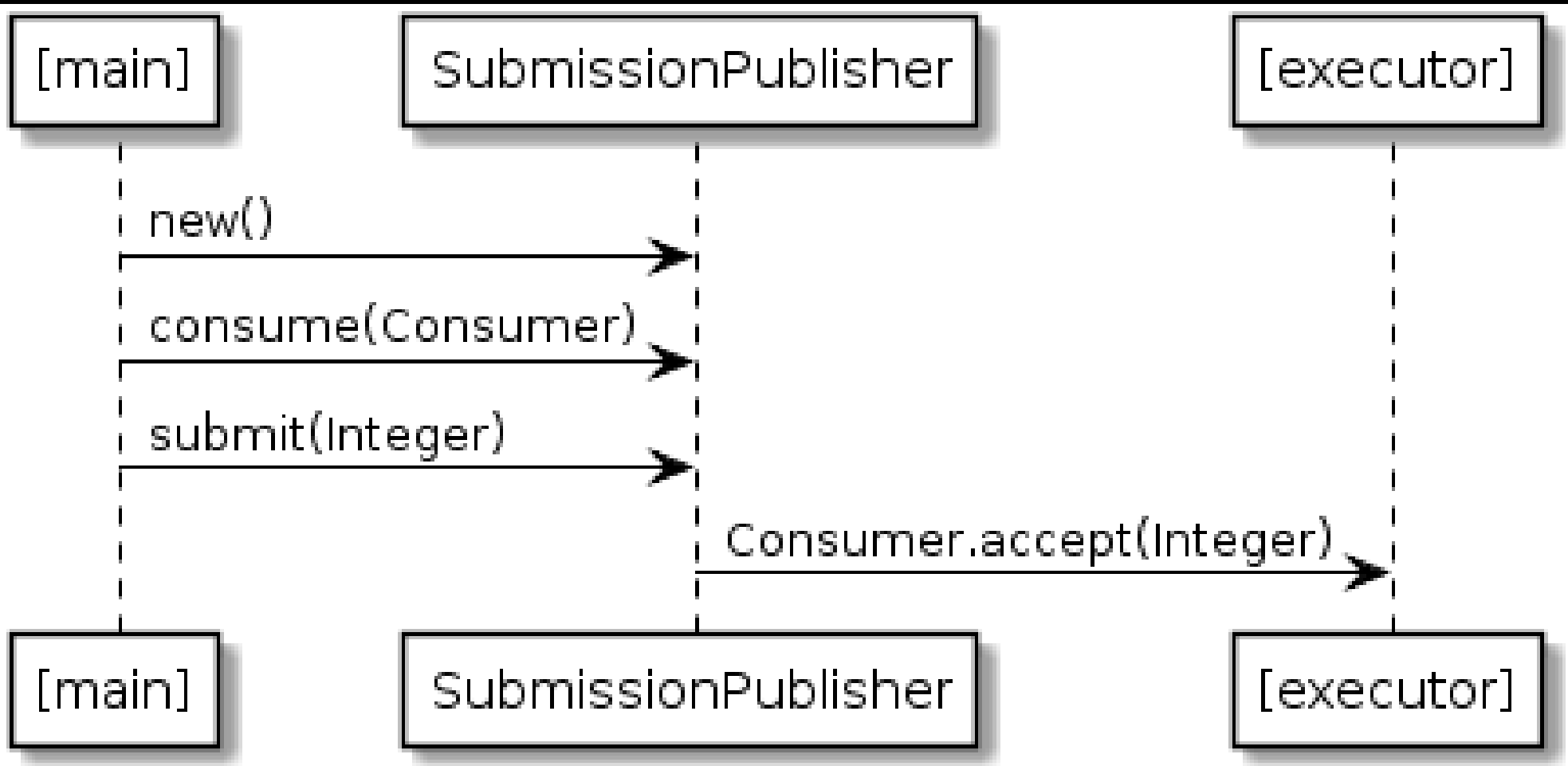
Implementation of `Flow.Publisher`:

```
SubmissionPublisher()
SubmissionPublisher(Executor executor,
    int maxBufferCapacity);
CompletableFuture<Void> consume(Consumer<T> cons)
int submit(T item)
void close()
```


Submission Publisher
subscribe
data
data
submit(data)
request(n)/cancel
Subscriber A
onNext(data)
data
data
Linked List
<BufferedSubscription>
request(n)/cancel
Subscriber B
onNext(data)
data
data

Example

```
var pub = new SubmissionPublisher<Integer>();
pub.consume(System.out::println);
pub.submit(1);
```


[main]
SubmissionPublisher
[executor]
new()
consume(Consumer)
submit(Integer)
Consumer.accept(Integer)
[main]
SubmissionPublisher
[executor]

Outline

Subscriber Example

```
class CustomSub implements Subscriber<Integer> {
    private Subscription subscription;
    public void onSubscribe(Subscription
            subscription) {
        this.subscription = subscription;
        subscription.request(1); }
    public void onNext(Integer value) {
        System.out.println(value);
        subscription.request(1); }
    public void onError(Throwable t) {
        System.err.println(t.getMessage()); }
    public void onComplete() {
        System.out.println("Done!"); }
}
```

Chaining SubmissionPublisher

```
var pub1 = new SubmissionPublisher<Integer>();
var pub2 = new SubmissionPublisher<Integer>();
var pub3 = new SubmissionPublisher<Double>();
var pub4 = new SubmissionPublisher<Double>();
pub1.consume(x -> { pub2.submit(x * x); });
pub1.consume(x -> { pub3.submit(x / 2.); });
pub2.consume(x -> {
    System.out.println("Square is: " + x); });
pub3.consume(x -> { if (x > 2) pub4.submit(x); });
pub4.consume(x -> {
    System.out.println("Half (> 2) is: " + x); });
pub1.submit(3);
pub1.submit(5);
```

Outline

Main Libraries

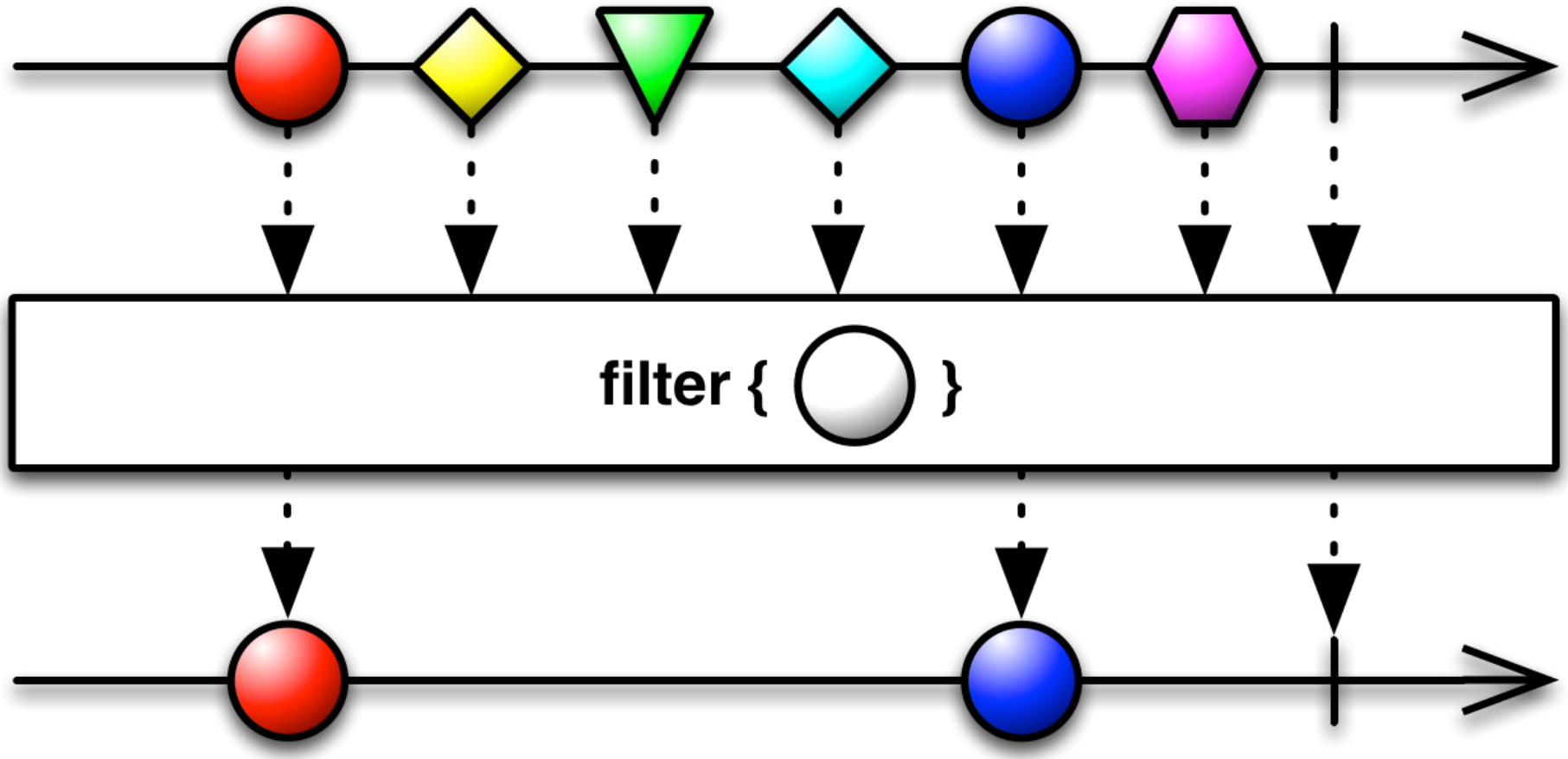
RxJava	Reactive Extensions, Netflix (more than 50 different operations)
Reactor	from Spring (equivalent to RxJava)
Akka	actor model

RxJava Example

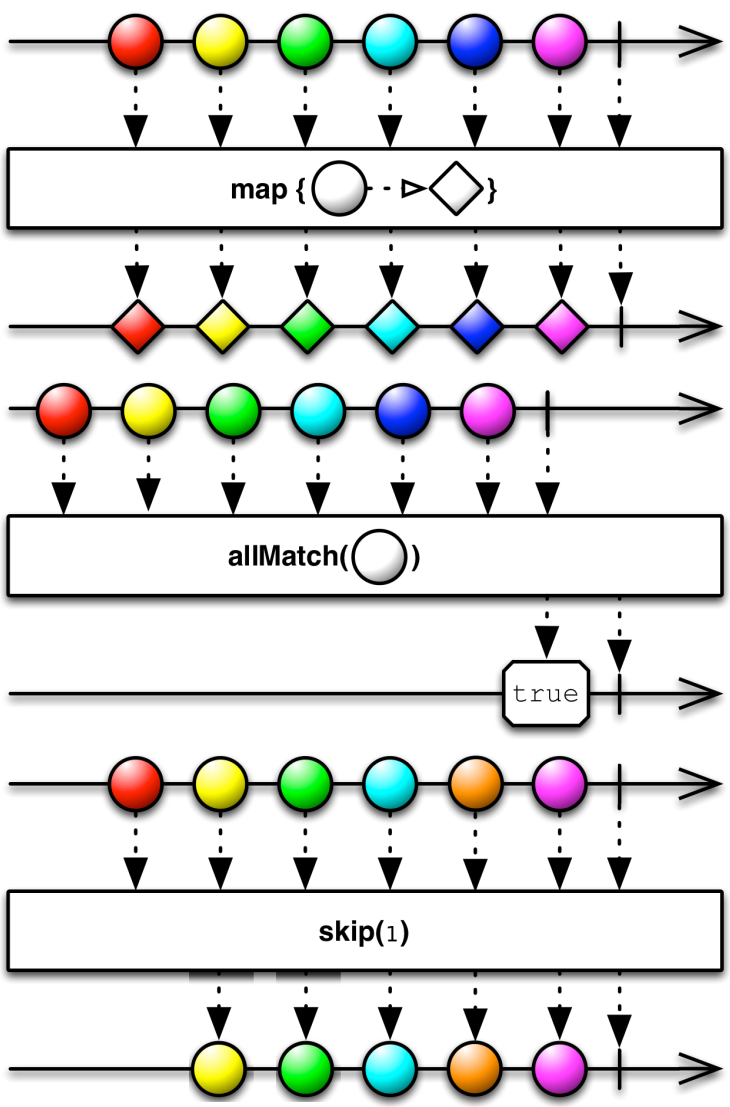
```
Flowable<Integer> flow = Flowable.range(1, 5)
    .map(v -> v * v)
    .filter(v -> v % 3 == 0)
    .subscribe(System.out::println);
```

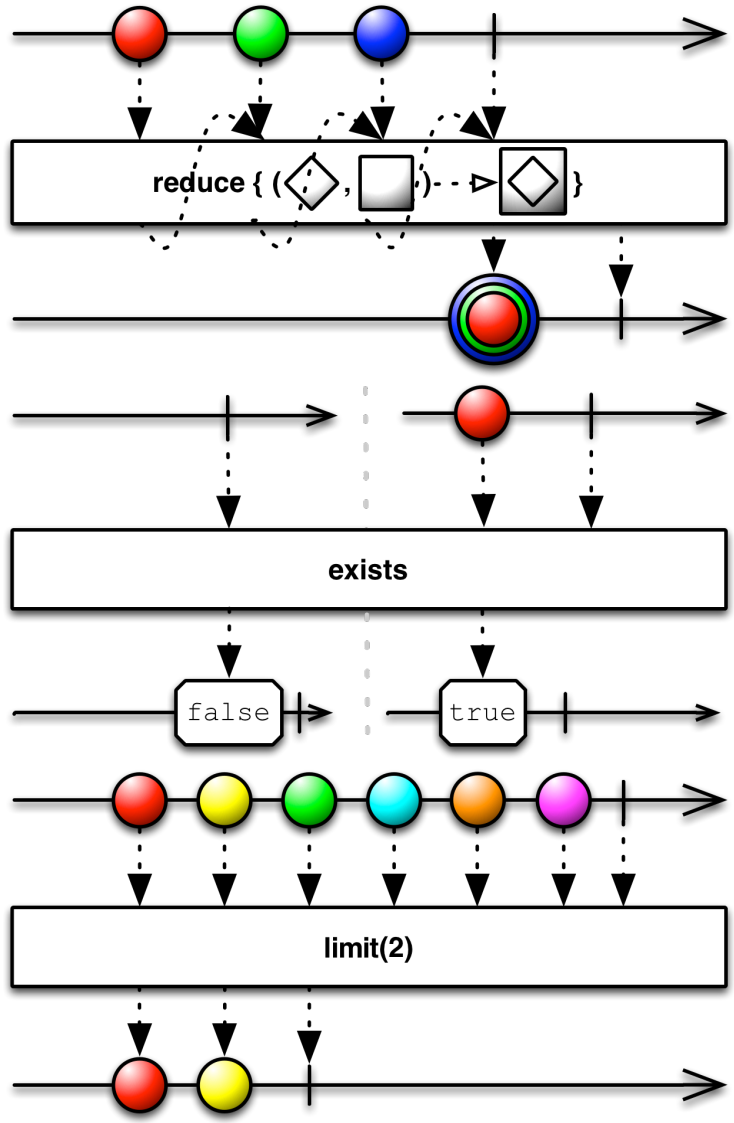
- `Flowable` implements `Publisher`.
- `subscribe` triggers the execution (as with terminal operations, lazy evaluation).

Marble Diagrams

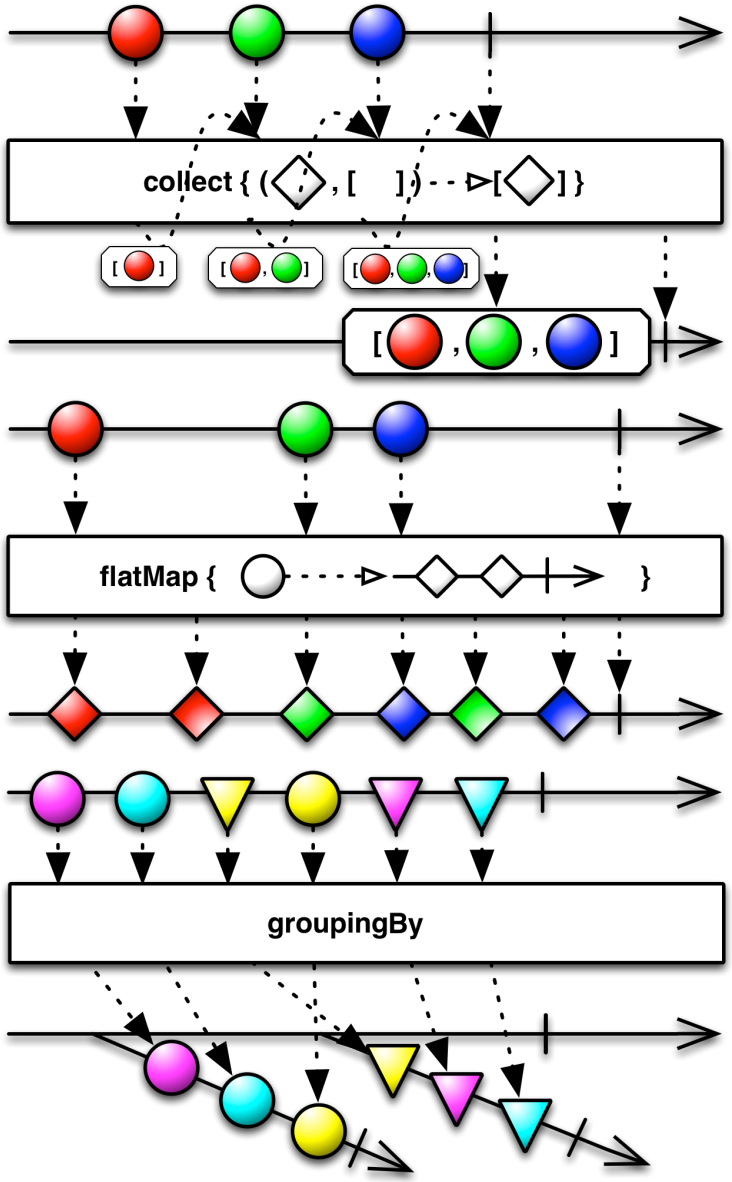


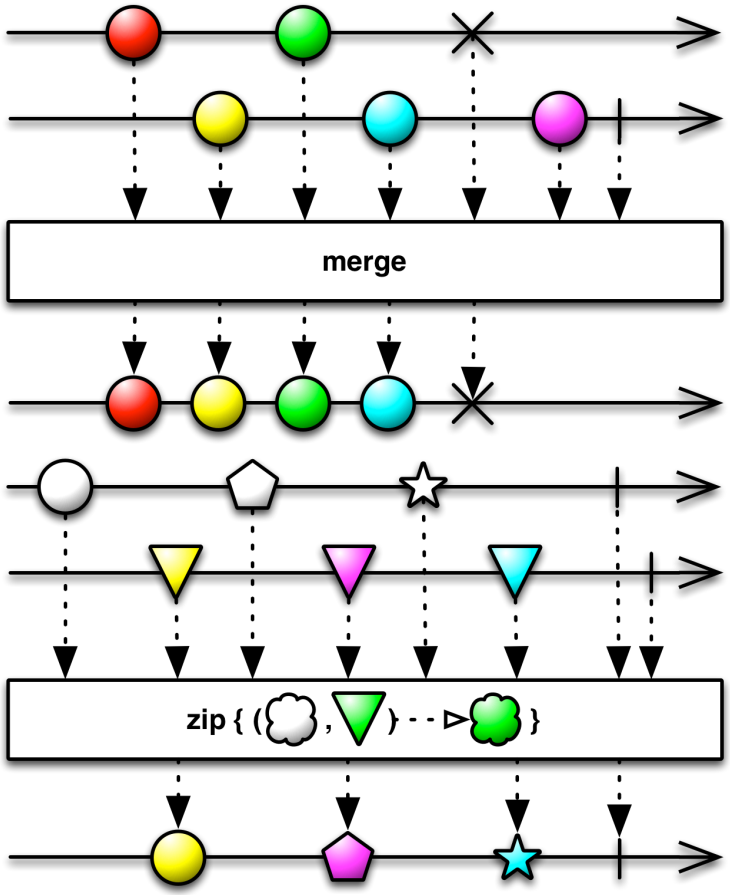
Classic Operations





Advanced Operations





Concurrency Source

- By default, the main thread performs all operations (blocking).
- The operation `subscribeOn` specifies how data processing can be processed concurrently.
- The operation at each stage can be performed asynchronously on a specific executor.

Outline

Official Documentation

- Documentation of class `Flow`
- Documentation of class `SubmissionPublisher`
- Documentation of class `Flowable`
- Documentation of class `Flux`

To Go Further

- https://akarnokd.blogspot.com/
- https://github.com/reactive-streams/reactive-streams-jvm
- Reactive Extensions (Rx) for Java or JavaScript:
  - http://reactivex.io/
  - https://github.com/ReactiveX/RxJava
  - http://introtorx.com/
  - http://xgrommx.github.io/rx-book/
  - https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- Reactor:
  - https://github.com/reactor/reactor-core
  - https://spring.io/blog/2016/06/13/notes-on-reactive-programming-part-ii-writing-some-code
  - https://spring.io/blog/2016/07/20/notes-on-reactive-programming-part-iii-a-simple-http-serve

Other Sources

- http://alexsderkach.io/comparing-java-8-rxjava-reactor/
- https://spring.io/blog/2016/06/07/notes-on-reactive-programming-part-i-the-reactive-landsca
- https://www.hascode.com/2018/01/reactive-streams-java-9-flow-api-rxjava-and-reactor-examp
- https://grokonez.com/java/java-9-flow-api-reactive-streams
- https://www.futurice.com/blog/top-7-tips-for-rxjava-on-android/
- https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c
- https://dzone.com/articles/5-things-to-know-about-reactive-programming