Jug Summer Camp

-enjoy it-

Introducing reactive streams

Speaker : Alexandre Delègue - @hanksleroux

Alexandre Delègue

- Développer @ Serli
- Java
- Scala
- Javascript
- Web

@chanksleroux

https://github.com/larousso

Serli

Société de conseil et d'ingénierie

Développement, expertise, R&D, formation

70 personnes

Contribution à des projets OSS, membre du JCP

Catalogue de formation :

- langages : java 8, scala …
- framework : java EE, play, akka, node js
- no sql : mongo DB, elasticsearch
- big data : hadoop, spark

Stream

Stream

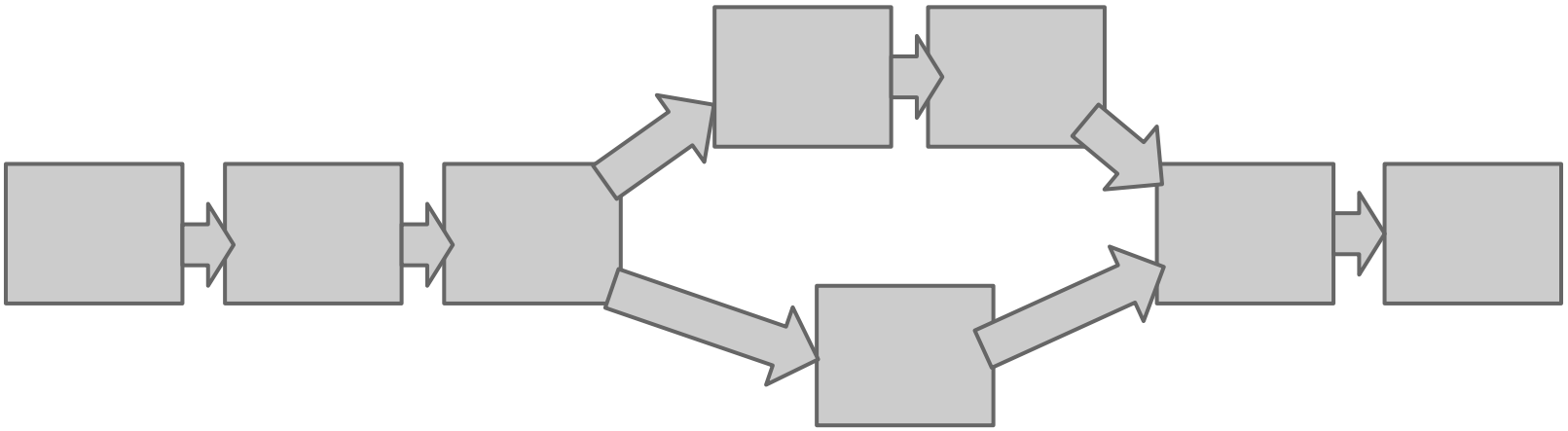
- flux éphémère de données
- continu
- peut être infini
- étapes de transformation

Use case

- Processing de gros fichiers
- Remonté de gros data sets depuis une base
- Processing depuis une queue
- Traiter des données en temps réél
- Echanges entre servers
- Monitoring
- Analytics

Besoin

- Transformation, composition, split, merge ...
- Consommateurs rapide ou lent
- Maîtrise de la conso mémoire
- Distribution des traitements
  - entre threads
  - entre core
  - entre CPU
  - entre applications
  - entre noeuds




JSC SUMMER CAMP

Transformer les éléments

```
List<String> minuscule = Arrays.asList("a", "b", "c", "d", "e");
List<String> majuscules = new ArrayList<>();
for (String elt : minuscule) {
   majuscules.add(elt.toUpperCase());
}
majuscules.forEach(System.out::println);
```

Collections

- Ensemble fini d'éléments
- Possibilité d'itérer sur ces éléments
- Difficile à transformer
- Difficile à composer

java.util.stream.Stream

java.util.stream.Stream

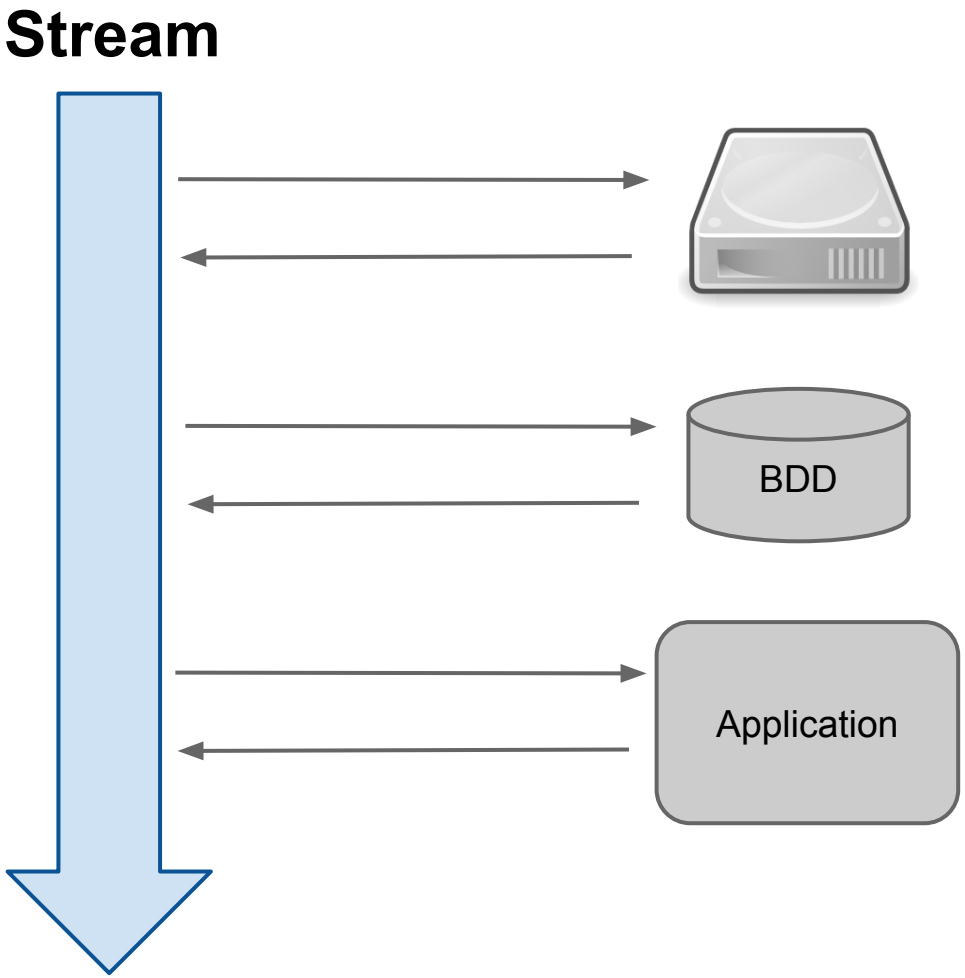
```
List<String> minuscule = Arrays.asList("a", "b", "c", "d", "e");
List<String> majuscules = minuscule
        .stream()
        .map(String::toUpperCase)
        .collect(Collectors.toList());

majuscules.forEach(System.out::println);
```

java.util.stream.Stream

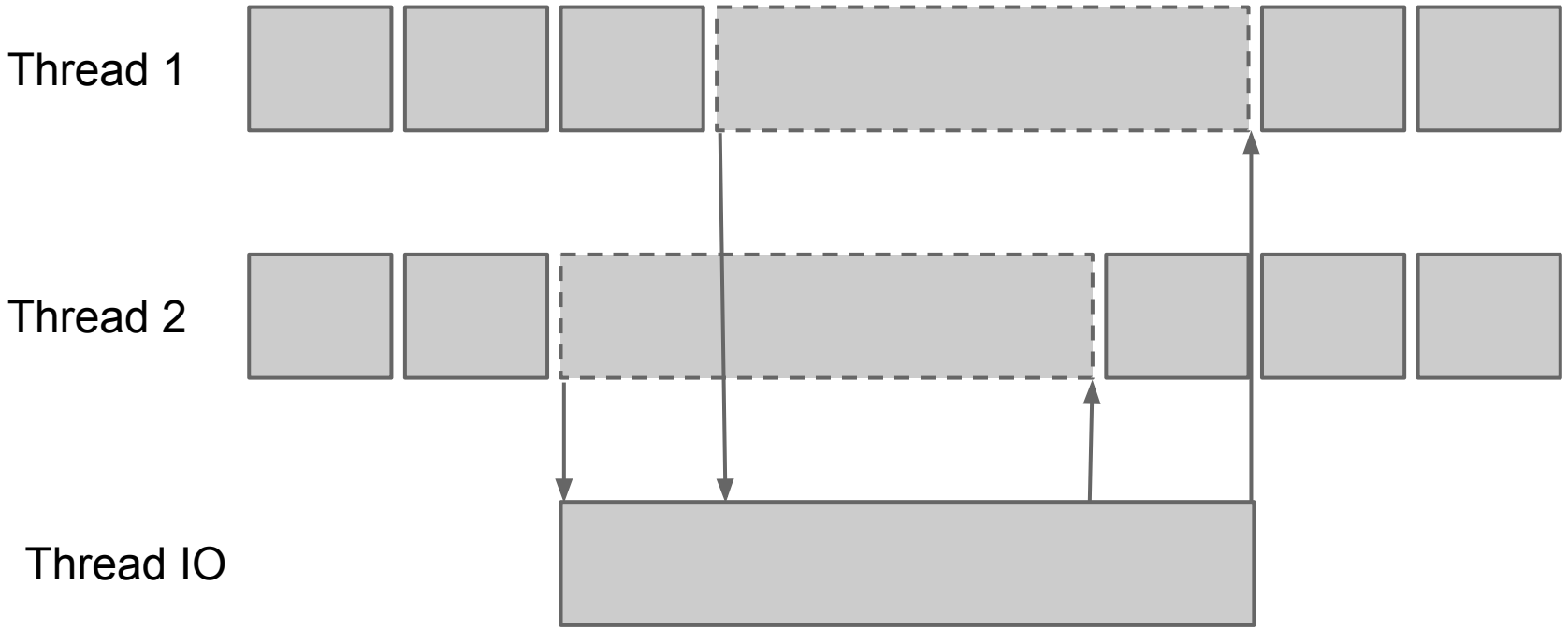
- Parallélisation
  - threads, core, cpu
- Composition
- Mais
  - Bloquant
  - Distribution entre applications / noeuds sur le réseaux :(

```java
public static void streamsBloquing() {

   Stream.iterate(0, i -> i + 1)
         .map(String::valueOf)
         .map(id -> getFromApi(id))
         .map(String::toUpperCase)
         .forEach(System.out::println);
}

public static String getFromApi(String id){
   try {
      return asyncHttpClient.prepareGet("http://localhost:8080/api/" + id).execute().get().
getResponseBody();
   } catch (IOException | InterruptedException | ExecutionException e) {
      throw new RuntimeException("Oups", e);
   }
}
```

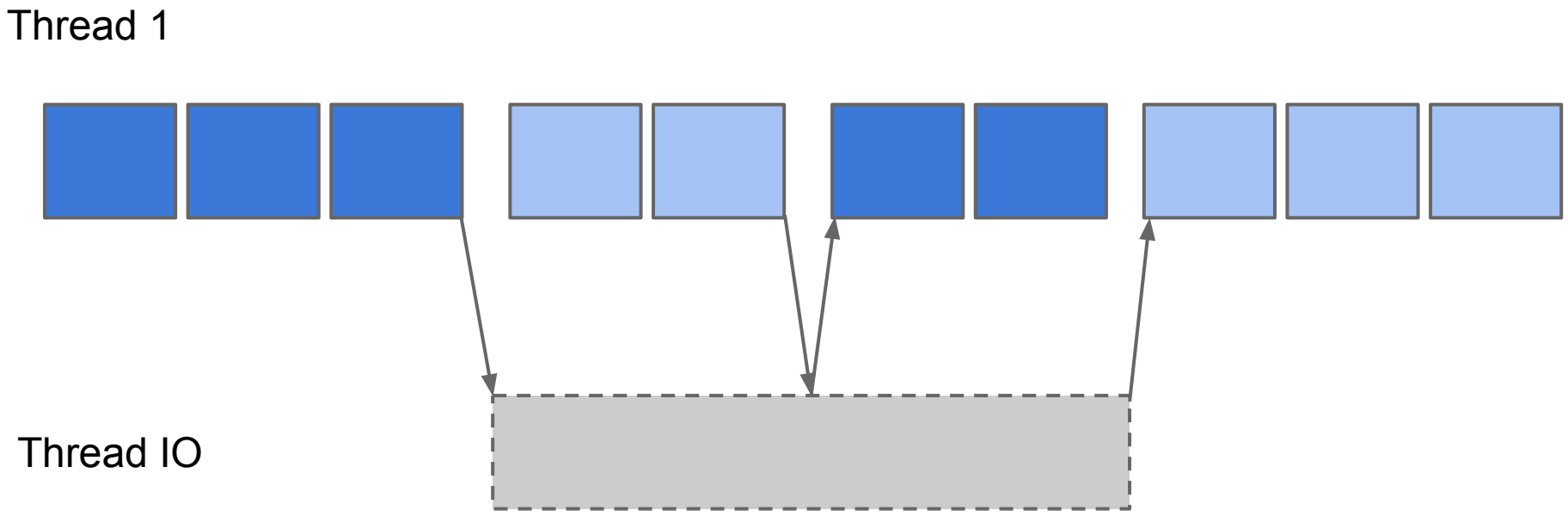
Latence



Latence



Async



```
public static void streamsAsync() {

  List<CompletableFuture<String>> results = IntStream.range(0, 20).boxed()
        .filter(i -> i % 2 == 0)
        .map(String::valueOf)
        .map(id -> getFromAsyncApi(id))
        .map(value -> value.thenApply(String::toUpperCase))
        .collect(Collectors.toList());

  CompletableFuture<List<String>> futureList = sequence(results);
  futureList.thenAccept(list -> list.forEach(System.out::println));
}

private static <T> CompletableFuture<List<T>> sequence(List<CompletableFuture<T>>
futures) {
  return CompletableFuture.allOf(futures.toArray(new CompletableFuture[futures.
size()]))
        .thenApply(any -> futures.stream()
                    .map(CompletableFuture::join)
                    .collect(Collectors.<T>toList())
        );
}
```

map / flatMap

```
Stream<Integer> ints = Arrays.asList(1, 2, 3, 4).stream();
Stream<String> strings = ints.map(i -> String.valueOf(i));
```

```
Stream<String> phrases = Arrays.asList("abcd", "efg", "hijk", "lmnopq", "rst", "uvw",
"xyz").stream();

Stream<Stream<String>> letters = phrases.map(p -> Stream.of(p.split("")));
```

```
Stream<String> letters = phrases.flatMap(p -> Stream.of(p.split("")));
```

Stream from Future<T> ?

```
List<String> results = IntStream.range(0, 20).boxed()
        .filter(i -> i % 2 == 0)
        .map(String::valueOf)
        .flatMap(id -> Stream.of(getFromAsyncApi(id)))
        .map(String::toUpperCase)
        .collect(Collectors.toList());
```

Stream from Future<T> ?

```
List<String> results = IntStream.range(0, 20).boxed()
        .filter(i -> i % 2 == 0)
        .map(String::valueOf)
        .flatMap(id -> Stream.of(getFromAsyncApi(id)))
        .map(String::toUpperCase)
        .collect(Collectors.toList());
```

Streams : pull api => impossible

Je te dis que t’es tendu comme une crampe

Pull vs Push

- Pull
  - T next();
- Push
  - void onNext(Consumer<T> apply)

Push or Pull

Ou tu sors ou je te sors mais faudra prendre une décision

RxJava

```
List<Integer> ints = IntStream.range(0, 20).boxed().collect(Collectors.toList());

Observable
    .from(ints)
    .filter(i -> i % 2 == 0)
    .map(String::valueOf)
    .concatMap(id -> Observable.from(getFromAsyncApi(id)))
    .subscribe(
      elt -> System.out.println(String.format("Next element %s", elt)),
      err -> System.out.println(String.format("Error %s", err)),
      () -> System.out.println("Completed")
    );
```

Rxjava parallélisation

```
List<Integer> ints = IntStream.range(0, 20).boxed().collect(Collectors.toList());

Observable
        .from(ints)
        .filter(i -> i % 2 == 0)
        //Ici on parallélise
         .flatMap(i ->
             Observable.just(i).observeOn(Schedulers.io())
             .map(String::valueOf)
             .flatMap(id -> Observable.from(getFromAsyncApi(id)))
         )
        .subscribe(
                elt -> System.out.println(String.format("Next element %s", elt)),
                err -> System.out.println(String.format("Error %s", err)),
                () -> {
                    System.out.println("Completed");
                }
         );
```

Opérateurs

- map
- flatMap
- reduce
- filter
- skip
- zip
- merge
- window

Démo

RxJava

- Parallélisation
  - threads, core, cpu, entre application
- Asynchrone
- Composition, split, merge ...
- Mais
  - Overflow :(
  - Buffer ou discard
  - Ok pour des consommateurs rapides

Back pressure

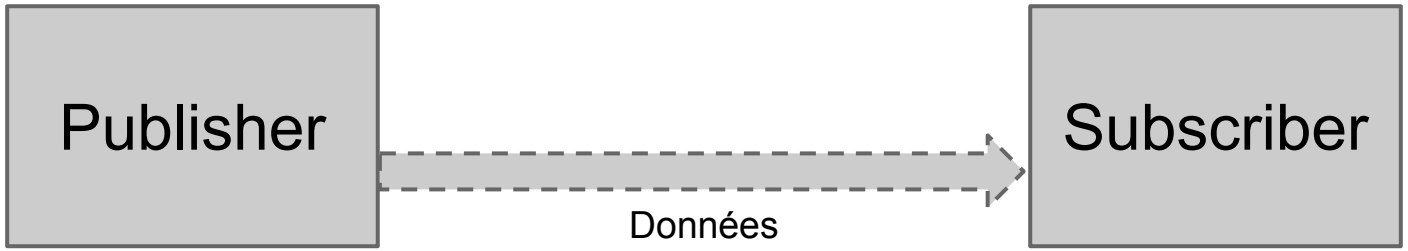
Solutions

- BUFFER
  - => out of memory
- BOUNDED QUEUE/ DISCARD
  - => perte d'éléments

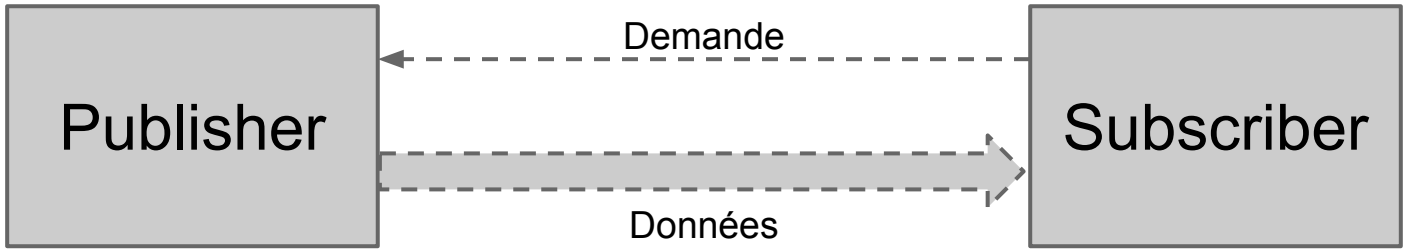
Reactive streams

- Netflix
- Oracle
- Twitter
- Pivotal
- Redhat
- Typesafe

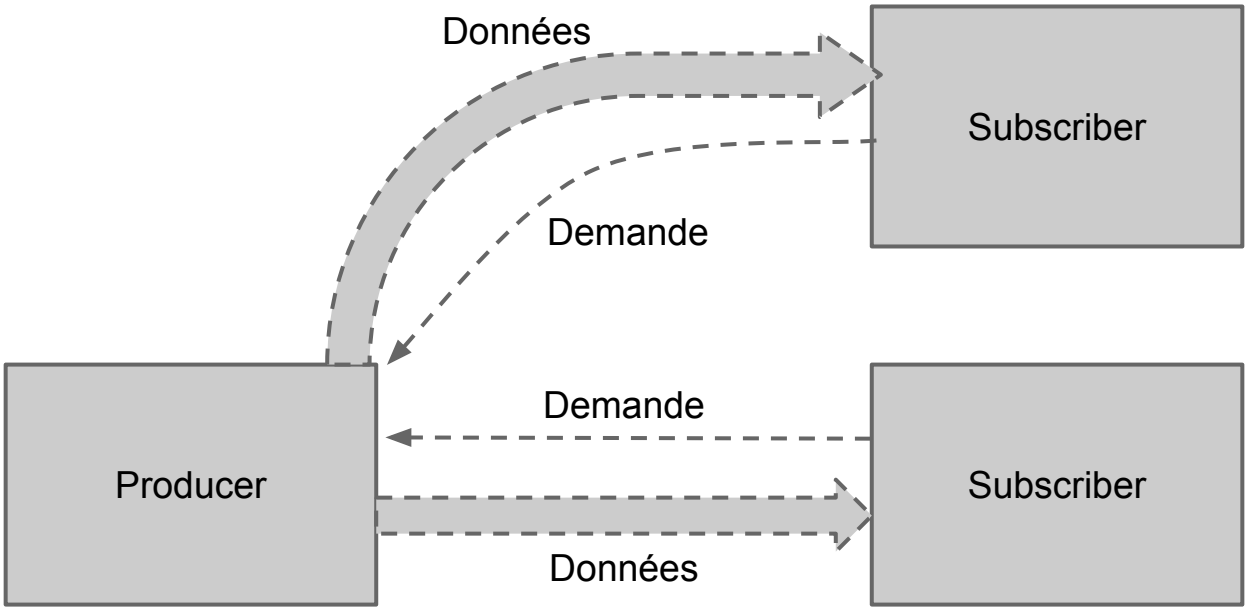
Reactive streams



Reactive streams

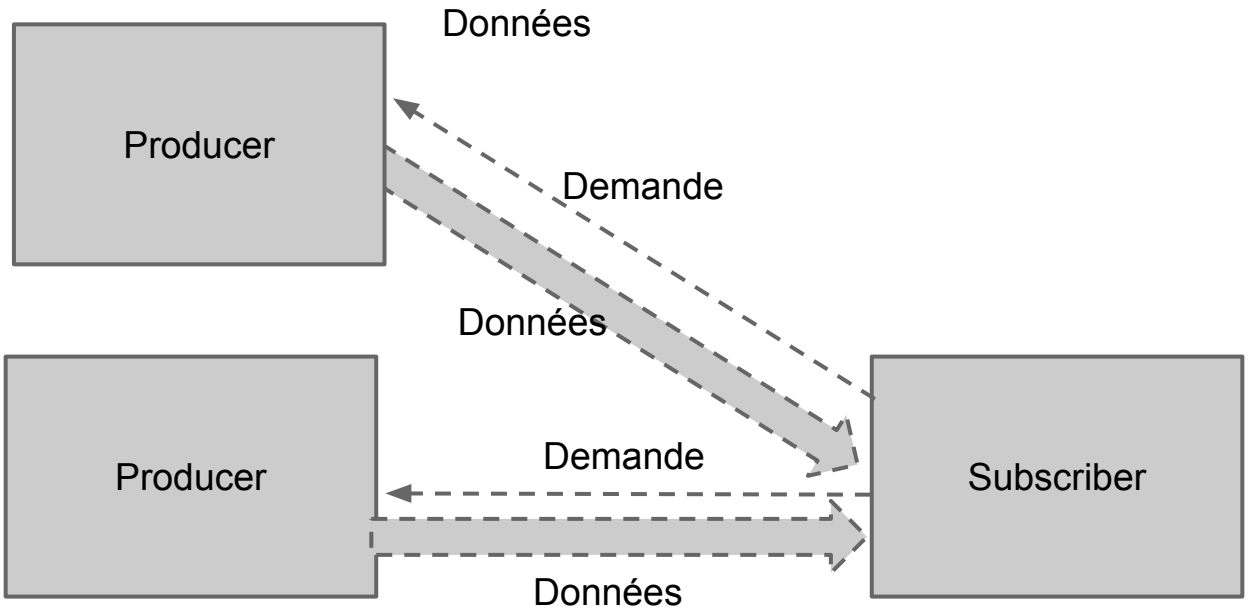


Split



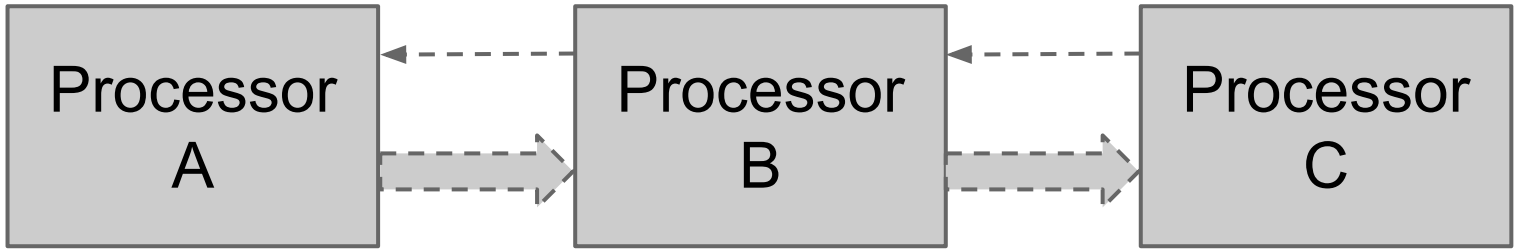
Merge des demandes au niveau du producer

Merge



Split de la demande pour le subscriber

Propagation de la back pressure



C ralentit qui B qui ralentit A

Composants

Publisher :

- publie les items du stream au subscriber

Subscriber :

- réçoit les items

Subscription :

- permet de faire le pont entre subscriber et publisher

```
public interface Publisher<T> {
   public void subscribe(Subscriber<? super T> s);
}

public interface Subscriber<T> {

   public void onSubscribe(Subscription s);

   public void onNext(T t);

   public void onError(Throwable t);

   public void onComplete();
}

public interface Subscription {

   public void request(long n);

   public void cancel();
}

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> { }
```

Api

- Interfaces
  - Subscriber, Publisher, Processor
- Spécification
  - Description des comportements de chaque composant
- TCK complet

Hot vs Cold

- Cold
  - Les éléments sont lues au moment de la souscription au stream
  - Chaque souscription va recevoir les mêmes données
- Hot
  - Les éléments arrivent de manière continue et sont éphémères
  - Les éléments lues dépendent du moment de la souscription au stream

Exemple de subscription

```
Publisher<String> publisher = ...;
publisher.subscribe(new Subscriber<String>() {

   private Subscription subscription;

   @Override
   public void onSubscribe(Subscription subscription) {
      this.subscription = subscription;
      this.subscription.request(1);
   }

   @Override
   public void onNext(String s) {
      System.out.println("onNext "+s);
      this.subscription.request(1);
   }

   @Override
   public void onError(Throwable throwable) {
      System.out.println("Oups : "+throwable.getMessage());
   }

   @Override
   public void onComplete() {
      System.out.println("End");
   }
});
```

Exemple de subscription

```
Publisher<String> publisher = ...;
publisher.subscribe(new Subscriber<String>() {

   private Subscription subscription;

   @Override
   public void onSubscribe(Subscription subscription) {
      this.subscription = subscription;
      this.subscription.request(1);
   }

   @Override
   public void onNext(String s) {
      System.out.println("onNext "+s);
      this.subscription.request(1);
   }

   @Override
   public void onError(Throwable throwable) {
      System.out.println("Oups : "+throwable.getMessage());
   }

   @Override
   public void onComplete() {
      System.out.println("End");
   }
});
```

Implémenteurs

- Reactor
- RxJava
- Akka Streams / Akka http
- Vert.x
- Ratpack
- Slick
- MongoDb
- Rabbit MQ
- Kafka
- ...

RxJava

RxJava

- Netflix
- Le plus mature
- Multiples langages
  - RxJava, RxScala, RxJs ….
- Adaptateur pour RxJava version 1.0
  - observable -> publisher
  - publisher -> observable
- RxJava version 2 : basé directement sur les reactives streams

```
Observable
  .from(ints)
  .filter(i -> i % 2 == 0)
  .map(String::valueOf)
  .flatMap(id -> Observable.from(getFromAsyncApi(id)))
  .subscribe(new Subscriber<String>() {
     @Override
     public void onStart() {
        request(1);
     }
     @Override
     public void onCompleted() {
        System.out.println("Completed");
     }
     @Override
     public void onError(Throwable throwable) {
        System.out.println("Oups : "+throwable.getMessage());
     }
     @Override
     public void onNext(String s) {
        System.out.println("Next : "+s);
        request(1);
     }
  });
```

Démo

Hot stream

- Quand la source ne supporte pas request(n)
  - onBackpressureBuffer
  - onBackpressureDrop
  - onBackpressureBlock

Reactor

Reactor

- http://projectreactor.io/
- Pivotal
- Seulement java
- Créé from scratch
- Basé sur Imax disruptor en interne
- API Proche de RxJava
- Opérateurs :
  - map, flatMap, concat, reduce, merge, broadcast
- IO streams

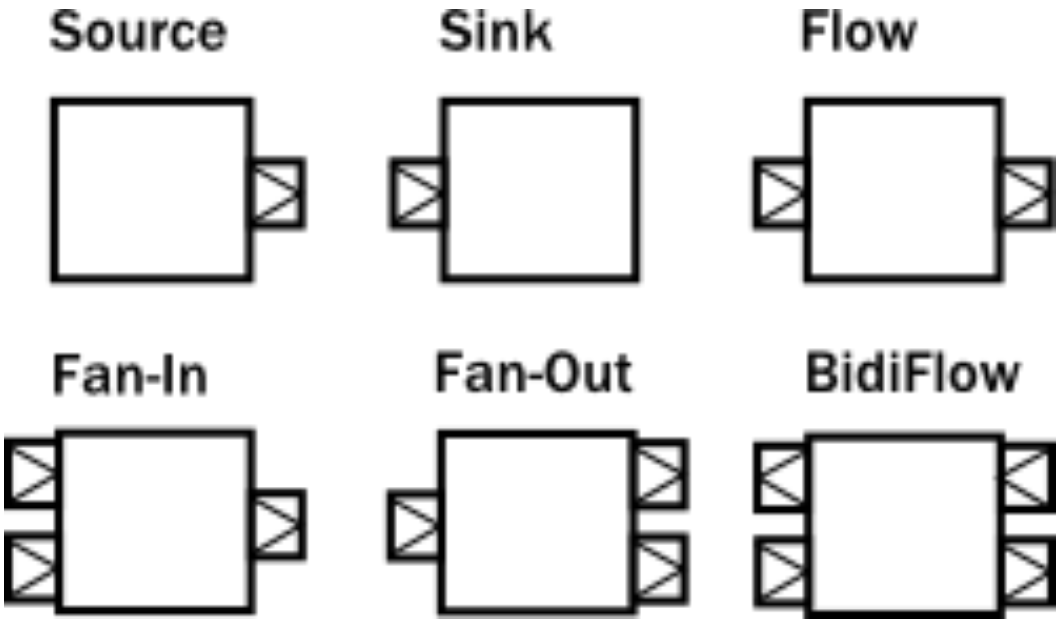
Démo

Akka streams

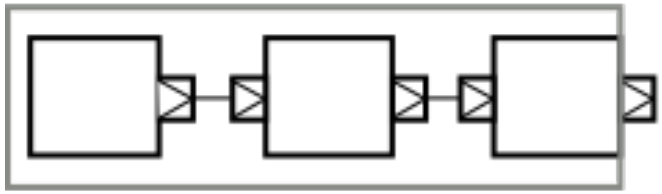
Akka streams

- Typesafe
- java et scala
- Type “blueprint”
  - on décrit le stream et on le matérialise ensuite
- Materializer basé sur les acteurs d’akka
- DSL : source, flow, graph, sink

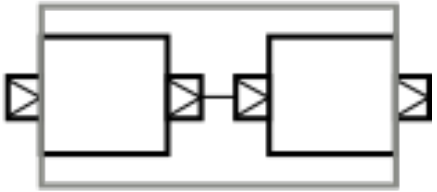
Modularité, composition et hiérarchie



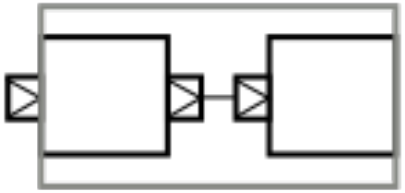
Composite Source



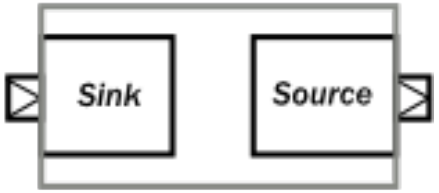
Composite Flow



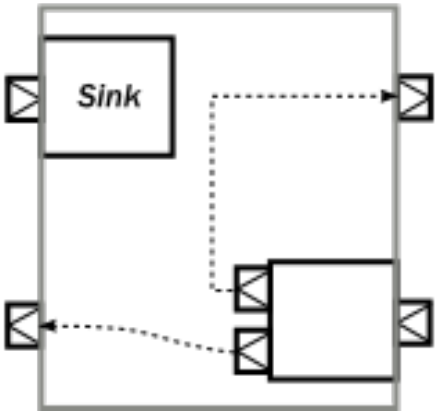
Composite Sink



Composite Flow
(from Sink and Source)



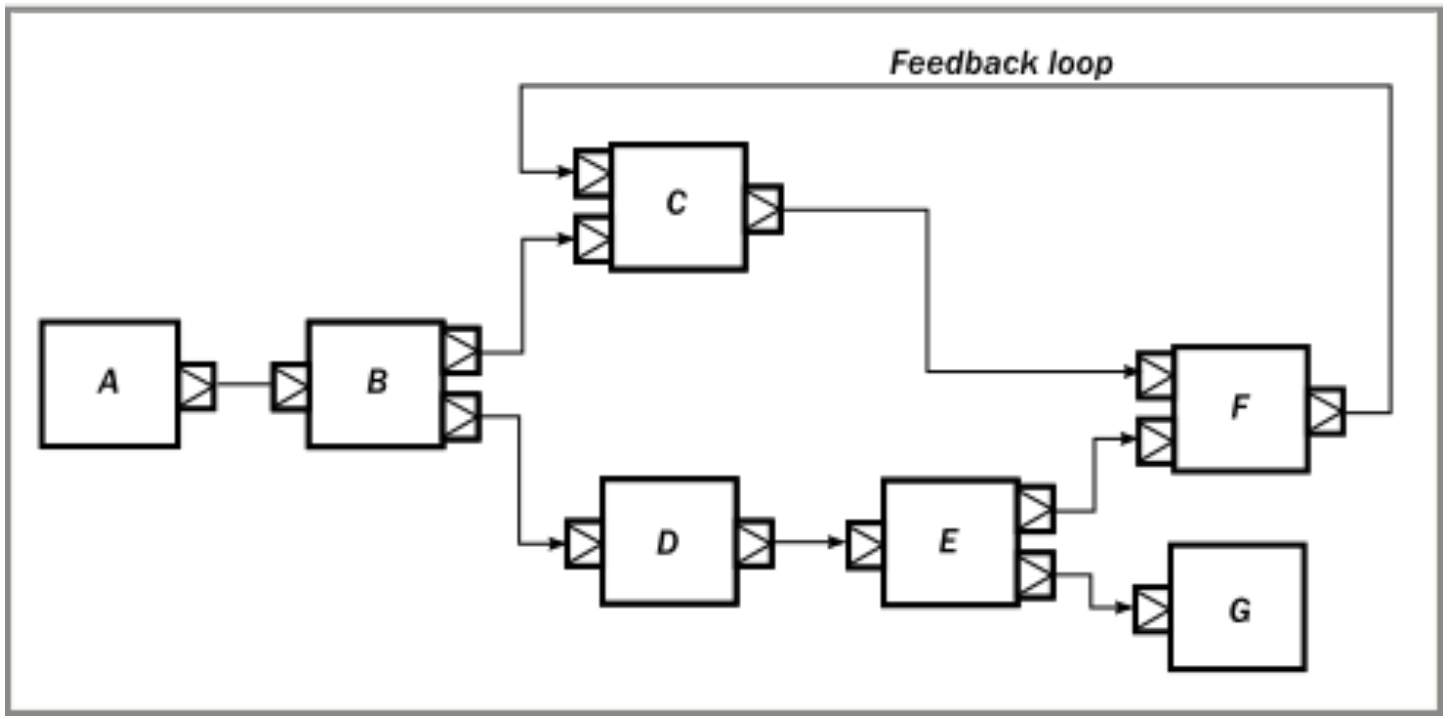
Composite BidiFlow



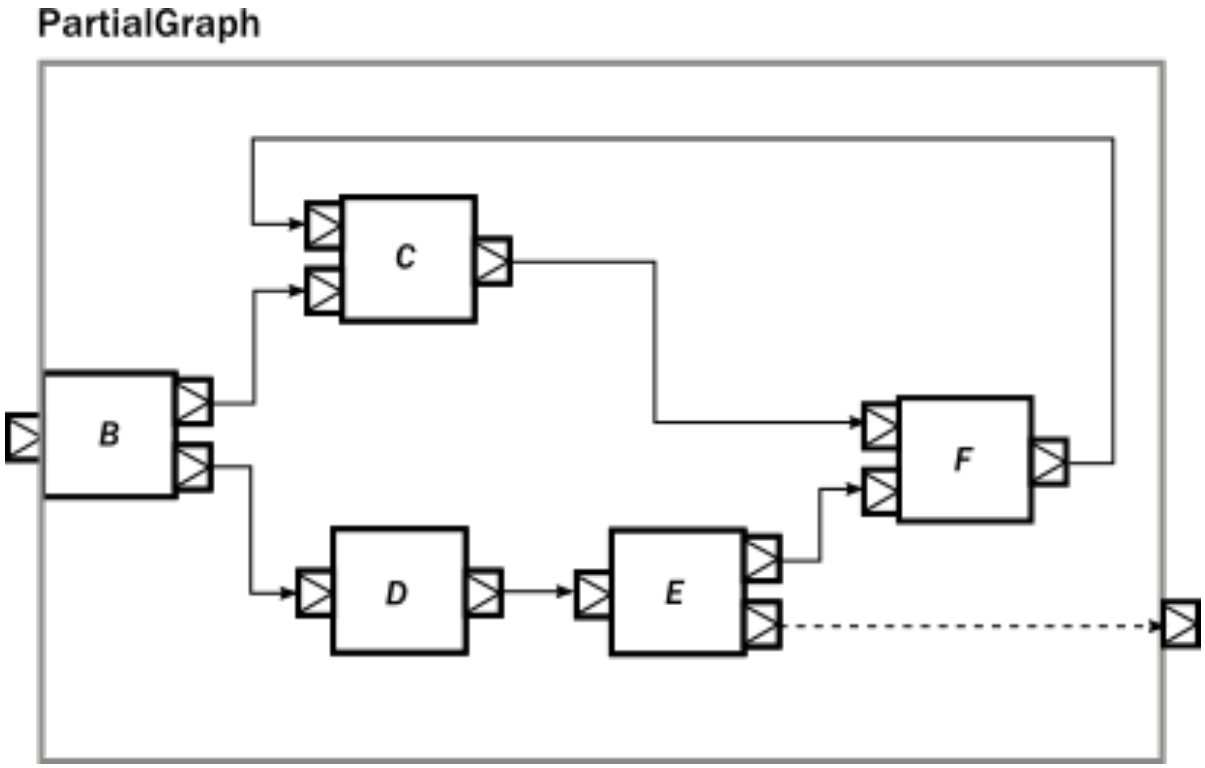
Démo

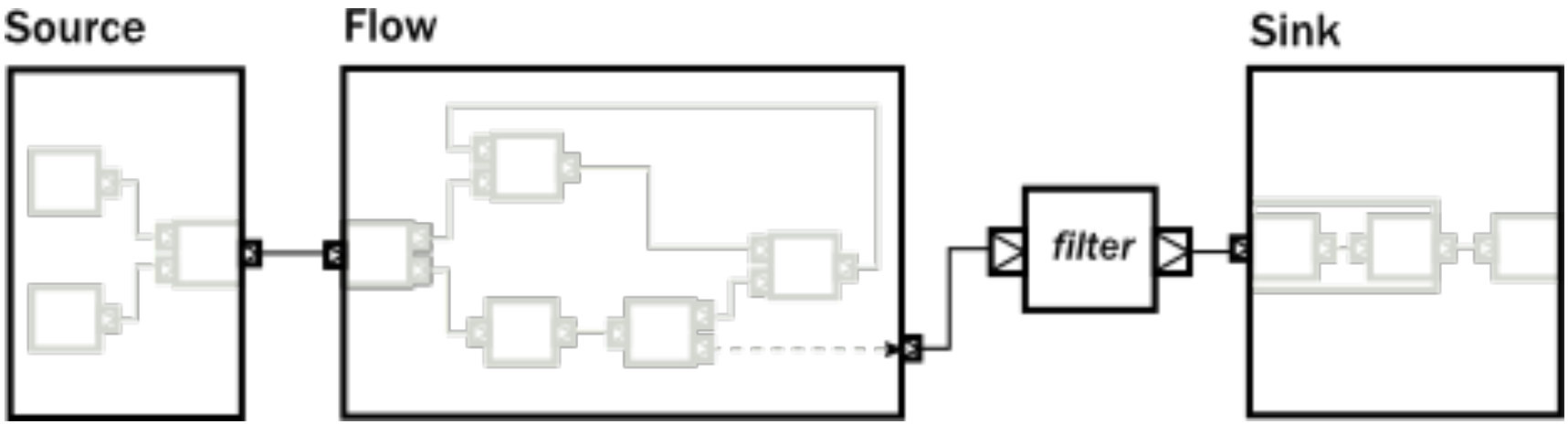
Graphs

RunnableGraph

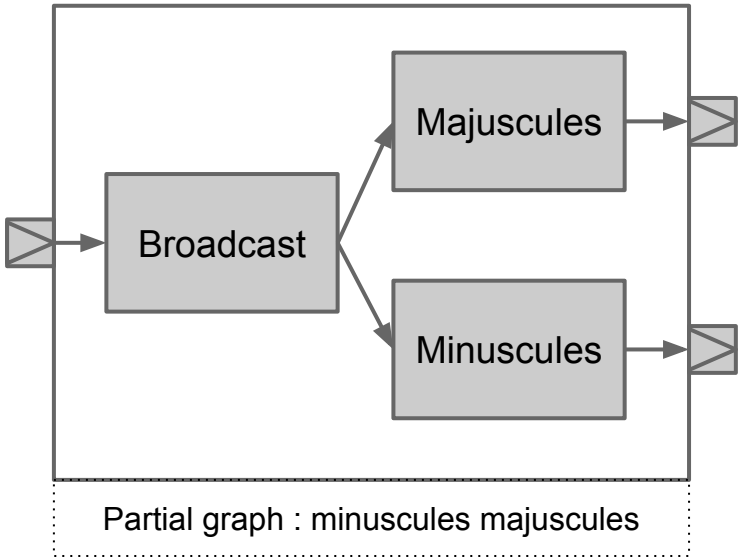


Graphs

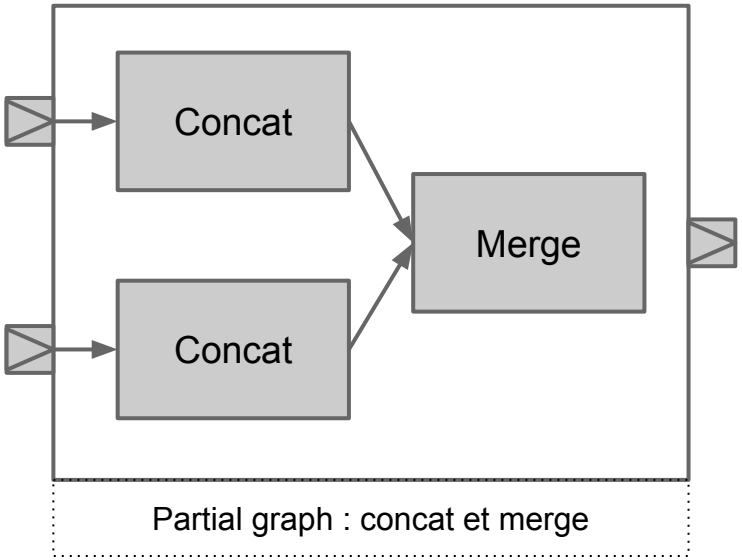



Source
Flow
filter
Sink

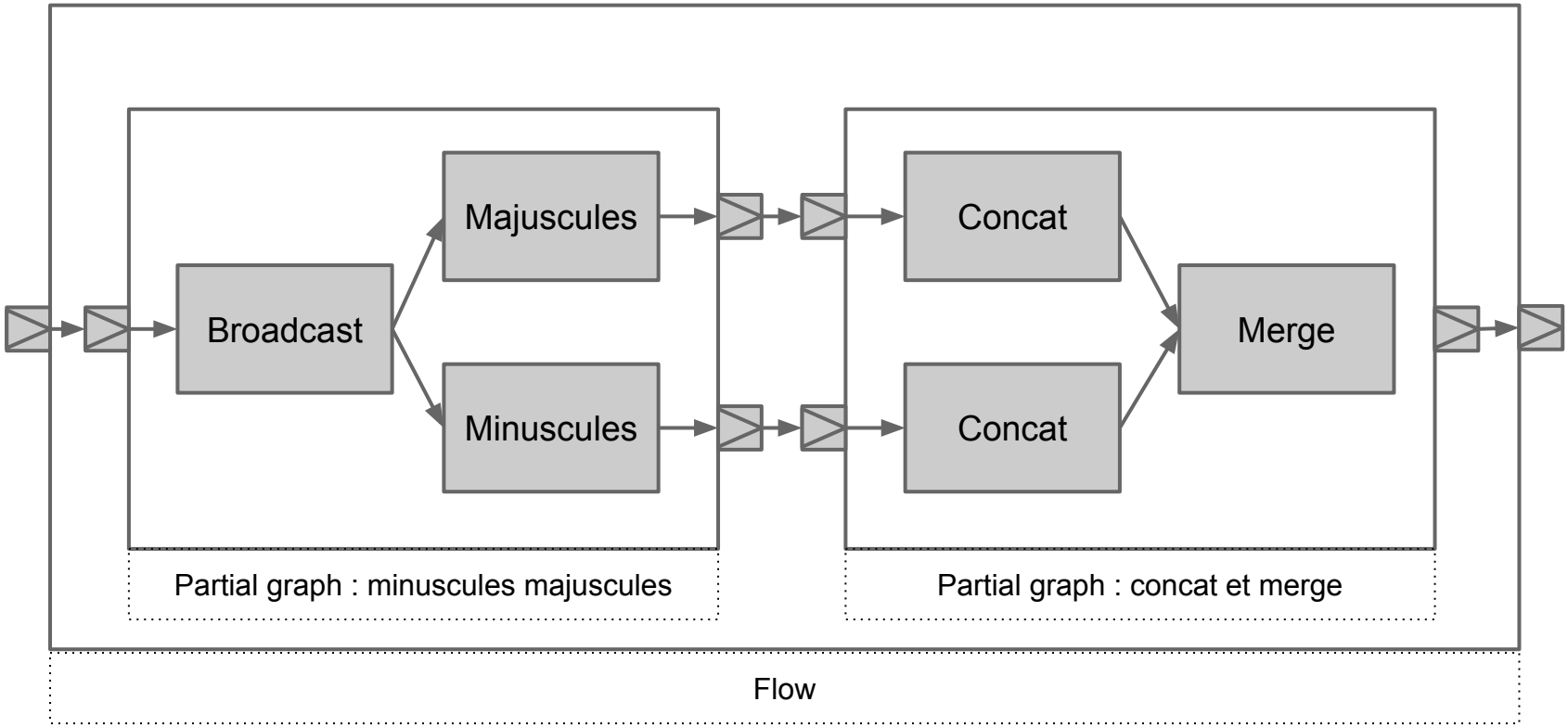
Exemple


Majuscules
Broadcast
Minuscules
Partial graph : minuscules majuscules

Exemple



Exemple


Broadcast
Majuscules
Minuscules
Concat
Concat
Merge
Partial graph : minuscules majuscules
Partial graph : concat et merge
Flow

Démo

Backpressure sur le réseaux

```
object TcpEcho extends App {

 private val connections: Source[IncomingConnection, Future[ServerBinding]] = Tcp().bind
("127.0.0.1", 8888)

 connections runForeach { connection =>
   println(s"New connection from: ${connection.remoteAddress}")

   val echo = Flow[ByteString]
     .via(Framing.delimiter(ByteString("\n"), maximumFrameLength = 256, allowTruncation = true))
     .map(_.utf8String)
     .map(_ + "!!!\n")
     .map(ByteString(_))

   connection.handleWith(echo)
 }

}
```

akka http

- Basé sur les reactives streams
- Boite à outils
- API client et server
- Gestion native de la backpressure
- Vient en remplacement de spray

```
implicit val system = ActorSystem()
implicit val executor = system.dispatcher
implicit val materializer = ActorMaterializer()

val requestHandler = path("") {
 get {
   complete {
     <h1>Welcome to the home page</h1>
   }
 }
} ~ path("hello") {
 get {
   complete{
     <h1>Welcome to the hello page</h1>
   }
 } ~
 post {
   redirect("/", StatusCodes.MovedPermanently)
 }
} ~ path("assets") {
   getFromBrowseableDirectory("Users/adelegue/http")
}

Http().bindAndHandle(requestHandler, "localhost", 8080)
```

Backpressure http

Démo

Interopérabilité

```
//Init
Environment.initialize();
final ActorSystem system = ActorSystem.create("interrop");
final Materializer mat = ActorMaterializer.create(system);

//Rx
Observable<Integer> from = Observable.from(Arrays.asList(1, 2, 3, 4, 5)).map(integ -> integ + 1);

Publisher<Integer> integerPublisher = RxReactiveStreams.toPublisher(from);

//Reactor
Stream<String> stream = Streams.wrap(integerPublisher).map(String::valueOf);

//Akka
Source<Integer, BoxedUnit> source = Source.from(stream)
                                          .map(Integer::valueOf).map(elt -> elt * 10);
Sink<Integer, Publisher<Integer>> publisher = Sink.publisher();

Publisher<Integer> akkaPublisher = source.runWith(publisher, mat);
```

L’application du future (proche)

- Accès aux données sous forme de publisher
  - BDD : Slick, Mongo DB, couchbase ...
  - Queue : Rabbit MQ, Kafka
- IO sous forme de publisher
- L’application devient un pipe de transformation de la donnée
- Streaming jusqu’au client à travers toutes les couches
  - Http, TCP

Intégration a java

- JDK 9
- java.util.concurent.Flow

- Servlet 4

MERCI

Liens utiles

- http://www.reactive-streams.org/
- https://github.com/reactive-streams/reactive-streams-jvm/
- http://projectreactor.io/
- http://reactivex.io/
- http://akka.io
- https://github.com/larousso/reactives-streams-exemples