Corso di Programmazione Concorrente

Reactive Streams

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Asynchronous Boundary
- Motivazioni
- Reactive Streams
- Back-Pressure
- Akka Stream
  - Specifica di *flussi* di processamento di stream
  - Materializzazione delle specifiche
  - Modello di Esecuzione Concorrente
- I confini di PC
  - Distribuzione verso Big-Data processing
  - Kafka, Gearpump ecc. ecc.

Asynchronous Boundary

- Un confine/limite per oltrepassare il quale un messaggio subisce un *hand-off* ovvero un viene scambiato tra f.d.e. distinti
- Questo può significare un scambio di msg
  - tra f.d.e. distinti sulla stessa macchina
  - tra macchine distinte
    - anche nodi diversi dello stesso cluster
    - classici client/server

  e, a livello applicativo, tra diversi
  - applicazioni
  - attori
  - worker-thread...

Una Scelta Obbligata

- Per molte applicazioni moderne, l'attraversamento degli asyncronous boundary non è più una possibile scelta, ma piuttosto una scelta *obbligata*
- Scale-up
  - distribuendo il calcolo su molteplici core
    - speed-up
- Scale-out
  - distribuendo il calcolo su molteplici nodi
    - speed-up
    - fault-tolerance
    - big-data

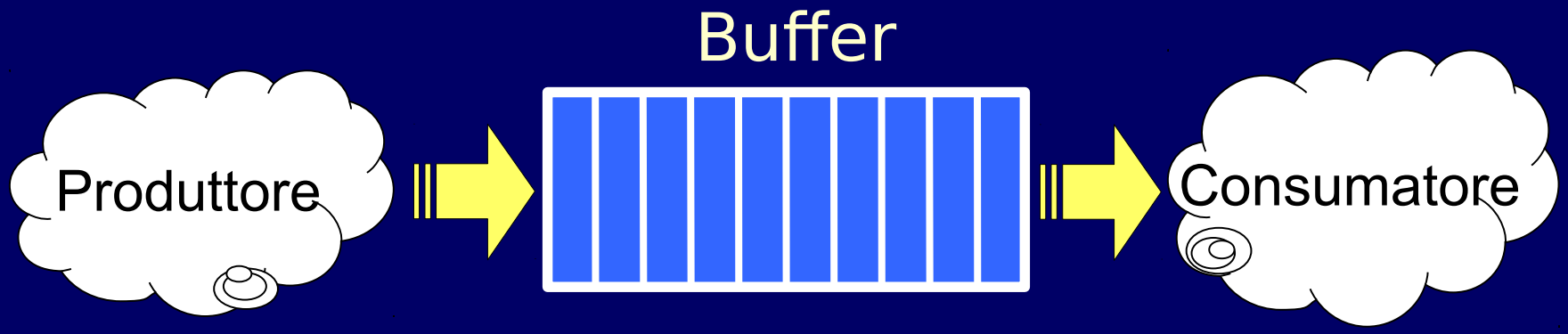
Un Problema Fondante

Come far attraversare ai messaggi gli asynchonous boundary utilizzando solo buffer di capacità finita

- Se non si risolve questo problema non è poi possibile garantire l'uso finito di memoria
  - Indipendentemente:
    - dal carico di lavoro…
    - dalla velocità dei nodi di calcolo collegati asincronicamente…

  e senza perdere i messaggi, ovviamente!

Una NON-soluzione



- Ovviamente la classica soluzione basata su buffer di dimensione infinita NON è una soluzione
  - In presenza di una sovraproduzione permanente l'uso di memoria sarebbe illimitato
- Una soluzione basata su buffer di dimensione finita sembra essere una possibile soluzione, ma con un decisivo svantaggio
  - introduce attese bloccanti in presenza di sovraproduzione
  - questo si sposa male con i nuovi framework *asincroni* che non le permettono/consigliano>>

Reactive Streams

"Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure on the JVM."

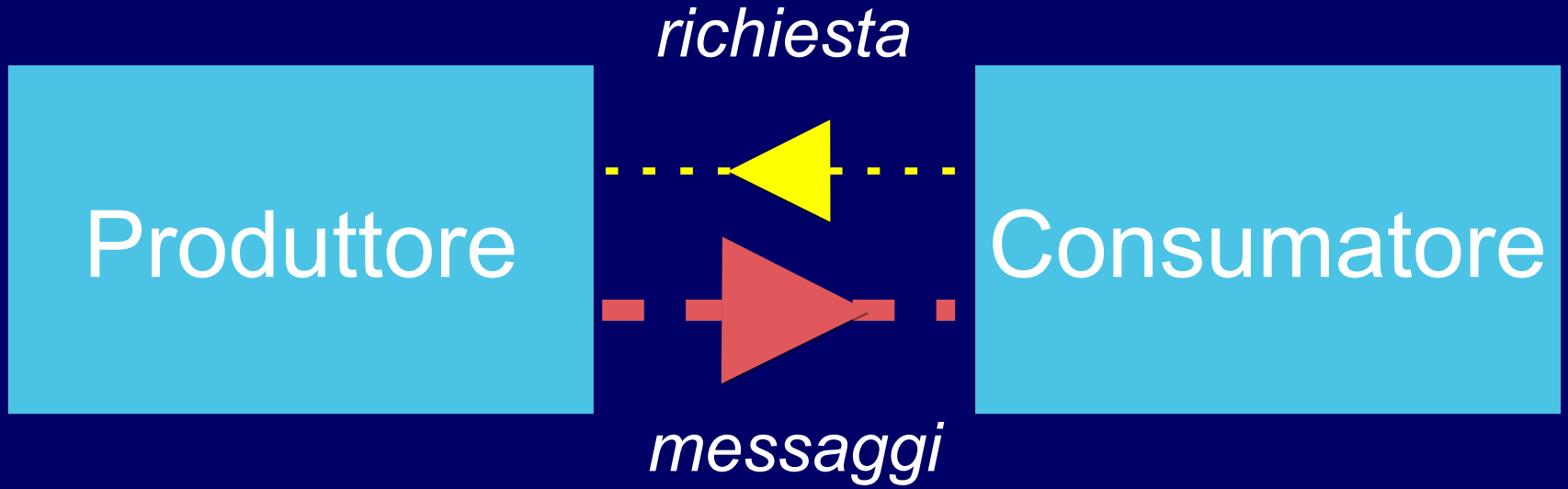
— *reactive-streams.org*

Reactive Streams

- “linea di pensiero” sulla migliore soluzione a questo problema
- iniziativa da parte di aziende ed individui che pur lavorando separatamente ed in contesti diversi, si sono tutti trovati a doverlo risolvere
- Iniziativa assolutamente positiva! Significa
  - Unire gli sforzi sul concepimento della migliore soluzione
  - Stabilire un vocabolario comune
  - Favorire l’interoperabilità delle soluzioni fissando
    - la sintassi di un’API comune
    - e vincolandone precisamente la semantica… come?
      - ✔ con un TCK, un insieme di regole codificate in test

Back Pressure

- Protocollo di comunicazione bidirezionale tra produttore e consumatore
- La produzione è vincolata
- Il consumatore *controlla* la velocità di produzione
  - Il produttore può produrre solo dietro esplicita richiesta del consumatore
  - Le richieste viaggiano in direzione opposta ai messaggi

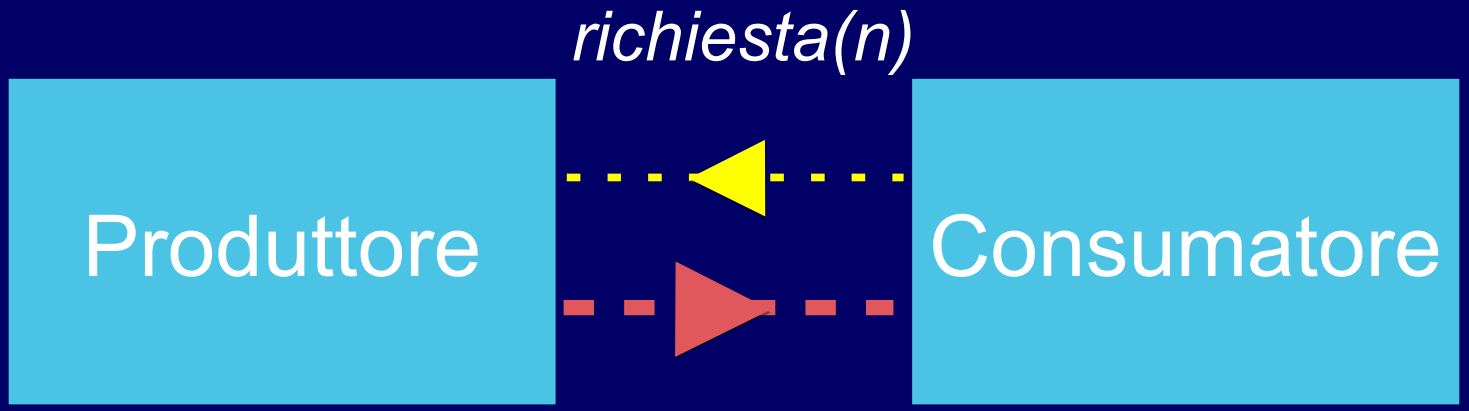


Push/Pull Dinamico

- Il protocollo permette di allineare la velocità delle due parti comunicanti
- Per un transitorio, questi possono essere disaccoppiati da un classico buffer capace di ospitare N messaggi
- Ma a regime...
  - Consumatore più veloce?
    - Il produttore sarà libero di produrre alla massima velocità (*comportamento* “push”)
  - Produttore più veloce?
    - Il consumatore, vedendo riempirsi il proprio buffer, limiterà la produzione (*comportamento* “pull”)
- La back-pressure rimanda la scelta del comportamento migliore a tempo dinamico

Batching delle Richieste

- Sembra costoso…
  - bisogna gestire la comunicazione in due direzioni
  - Ogni messaggio prodotto segue una richiesta, raddoppiando l'effettivo numero di messaggi scambiati?
- In realtà il consumatore può chiedere di inviare *n* messaggi con una sola richiesta
  - gli basta misurare la capienza residua del proprio buffer



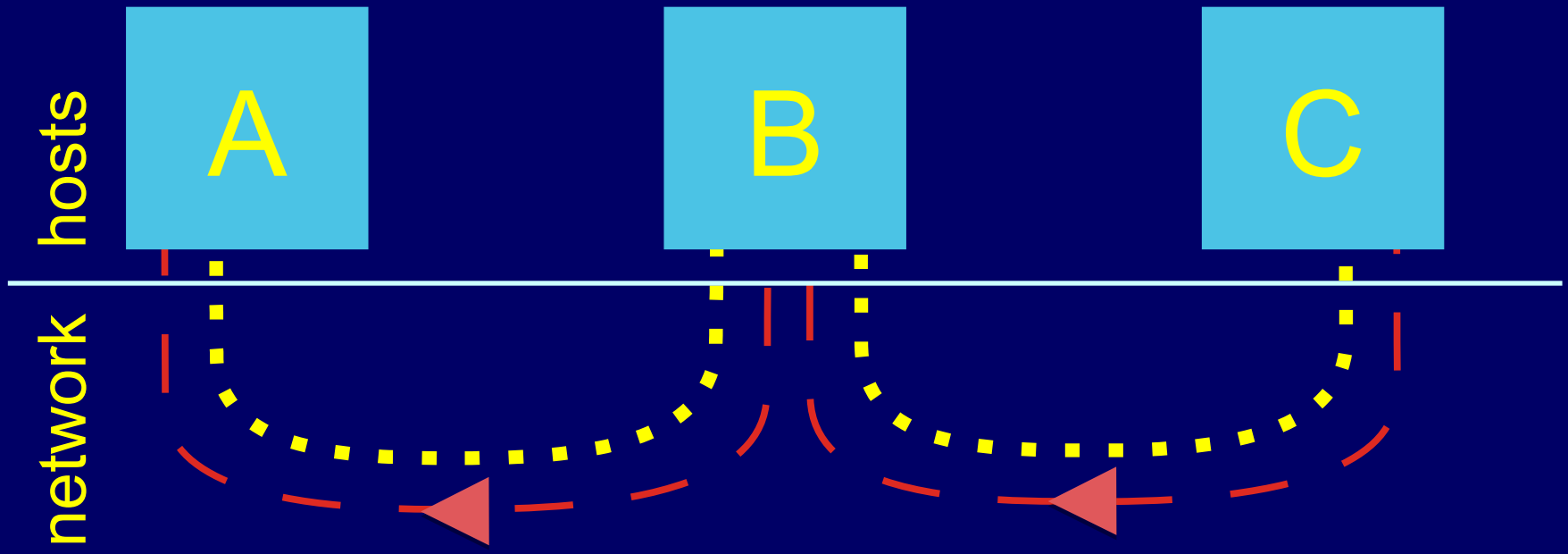
Back-Pressure Contagiosa

- Se C non riesce a reggere il passo di B...

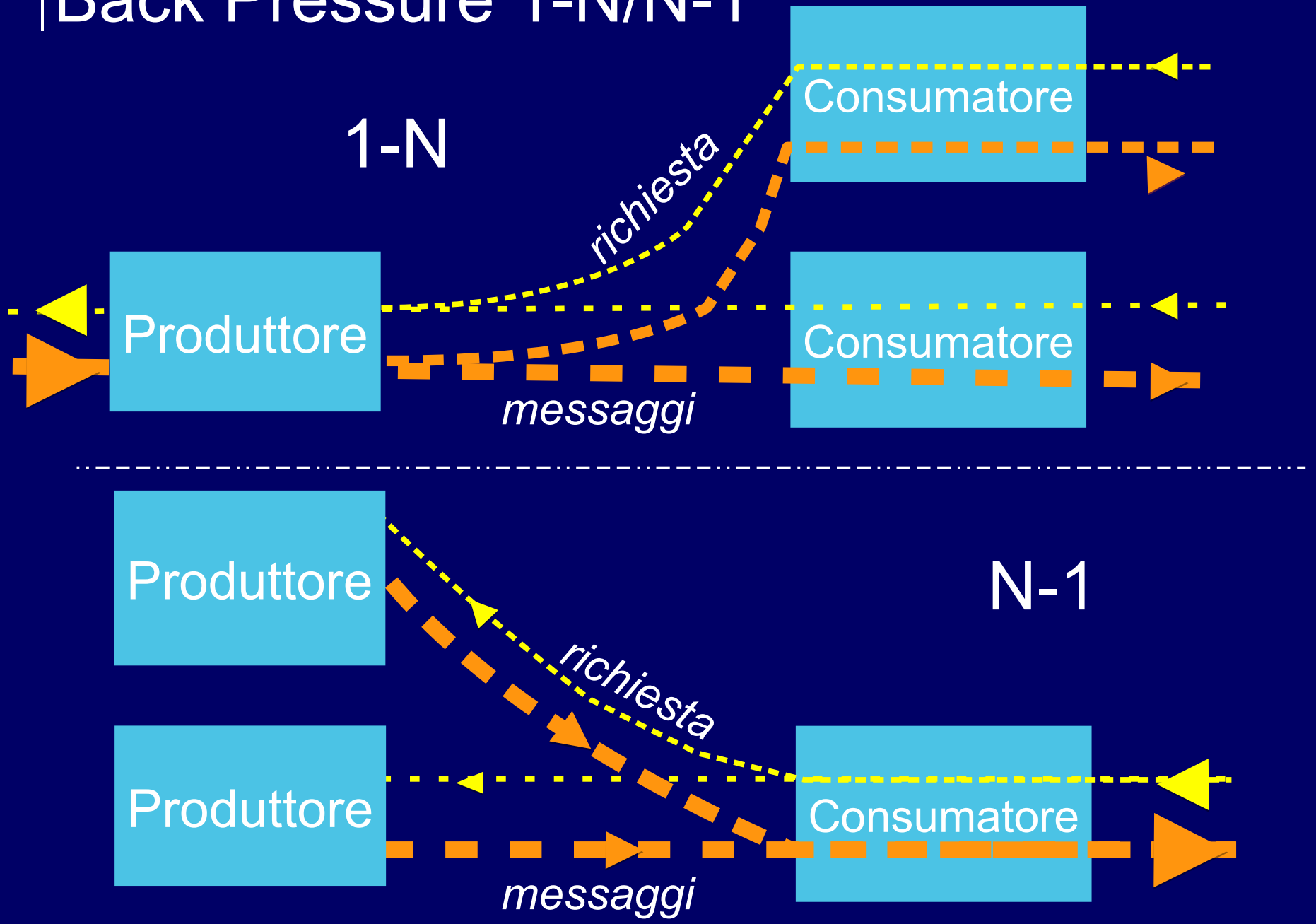
  ...allora fa in modo di rallentare B
- Questo potrebbe costringere B a rallentare
  - Se non fosse per la lentezza di C, B potrebbe comunque reggere la velocità di produzione di A, rispetto al quale figura come consumatore

TCP e Back-Pressure

- La back-pressure è una tecnica già consolidata per i protocolli di networking come TCP
  - in quel caso è però consentita e gestita la perdita e la ritrasmissione di messaggi/pacchetti
  - Negli scenari di nostro principale interesse sono invece considerati intollerabili



Back Pressure 1-N/N-1



Reactive Streams

- Riassumendo, mediante una tecnica nota come *back-pressure* prevedono
  - un flusso di messaggi asincrono e non-bloccante
  - un flusso di richieste asincrono e non-bloccante
  - coordinamento minimale (batching delle richieste)
- Questo consente l'attraversamento di tutti i tipi di asynchronous boundary (locali e distribuiti) utilizzando una quantità di memoria limitata superiormente
- Fondamentale per scalare il processamento di stream di dati su più core e su più macchine
  - Ma anche per l'esplosiva crescita del numero di dispositivi che producono e consumano dati meglio gestiti come stream
    - Mobile devices
    - Internet of Things
  - Big-Data

Reactive Manifesto

- Molti suggeriscono di rivisitare in tal senso anche i vecchi protocolli batch-based (ad es. quelli per interrogare dati persistiti – *http://slick.lightbend.com/download/*)
- I *Reactive Streams* possono inquadrarsi come una delle iniziative che mirano a rivisitare il modo di progettare le applicazioni
  - Le architetture standard (*n*-Tiers) cominciano a mostrare chiari segni della loro età quando usate per risolvere i problemi moderni
- La percentuale di I/O in un servizio moderno per singola richiesta, in confronto al passato, è enormemente aumentata
  - ✔ NON-bloccarsi in attesa dell'I/O è diventato necessario per garantire tempi di risposta contenuti, e sopportare carichi elevati, per tutte le applicazioni thread-pooled
- Tra le tante iniziative: *www.reactivemanifesto.org*

Un Esempio Illuminante

- Cosa accade e cosa accadeva quando un browser carica una singola pagina Web?
  - *Una volta*
    - si facevano attese bloccanti per aspettare l'arrivo di tutte le risposte necessarie
    - non si mostrava il risultato sino alla fine
    - ✔ Questo perché I/O necessario era limitato
  - *Oggi* è praticamente impossibile seguire lo stesso approccio
    - centinaia di richieste remote
    - accessi a decine di servizi remoti offerti da parti terze
      - servizi SSO, social, maps ecc. ecc.
    - la pagina viene mostrata immediatamente, a costo di aggiornarla in corso

Reactive Streams vs Java 8 Stream

- Abbiamo già incontrato gli *Stream* come libreria Java 8+ per la gestione di flussi di dati
- Molti concetti alla loro base rimangono utili e validi
- Java 8 Stream non prevedono al suo interno alcun meccanismo di back-pressure, e quindi non risolvono il problema da cui siamo partiti
  - Non permettono di oltrepassare tutti i tipi di asynchronous boundary con uso limitato di memoria
  - Non è progettata per essere eseguita su piattaforme distribuite
- Gli Java 8 Stream sono *pull-based*
  - le operazioni terminali scatenano il calcolo chiedendo a ritroso i dati senza materializzarli

Reactive Streams vs RxJava

- Esistono altre soluzioni come RxJava
- Inizialmente (v1.x) *push-based*, e con *unbounded-buffer*
  - bloccanti: era il “meccanismo” di back-pressure
- Successivamente (v2.x) hanno aderito all’iniziativa *Reactive Streams*
  - back-pressure
  - bounded-buffer
  - non-bloccanti
- Ma non supportano ancora la distribuzione su più macchine (almeno non open-source...)

JEP-266 ; Java 9+: `java.util.concurrent.Flow`

Reactive Streams in Java 9...>>

```
public final class Flow {
  private Flow() {} // uninstantiable

  @FunctionalInterface
  public static interface Publisher<T> {
    public void subscribe(Subscriber<? super T> subscriber);
  }

  public static interface Subscriber<T> {
    public void onSubscribe(Subscription subscription);
    public void onNext(T item);
    public void onError(Throwable throwable);
    public void onComplete();
  }

  public static interface Subscription {
    public void request(long n);
    public void cancel();
  }
  public static interface Processor<T,R>
                extends Subscriber<T>, Publisher<R> {
  }
}
```

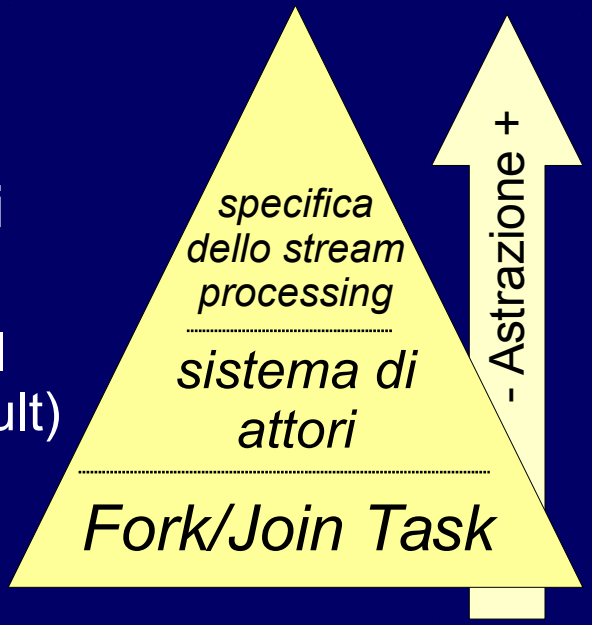
Akka Stream: Implementazione ad Attori dei Reactive Streams

- Una implementazione particolarmente interessante dei Reactive Streams
  - *back-pressure*
  - *bounded-buffer*
  - *non-blocking*

- Non nasce all'interno del progetto Akka casualmente
  - tra i principali promotori dei Reactive Streams ci sono alcuni degli iniziali sviluppatori di Akka
  - ...tra i molti che dovevano risolvere gli stessi problemi di processamento di stream con uso finito di memoria da cui siamo partiti

Piramide di Astrazione di Akka Stream

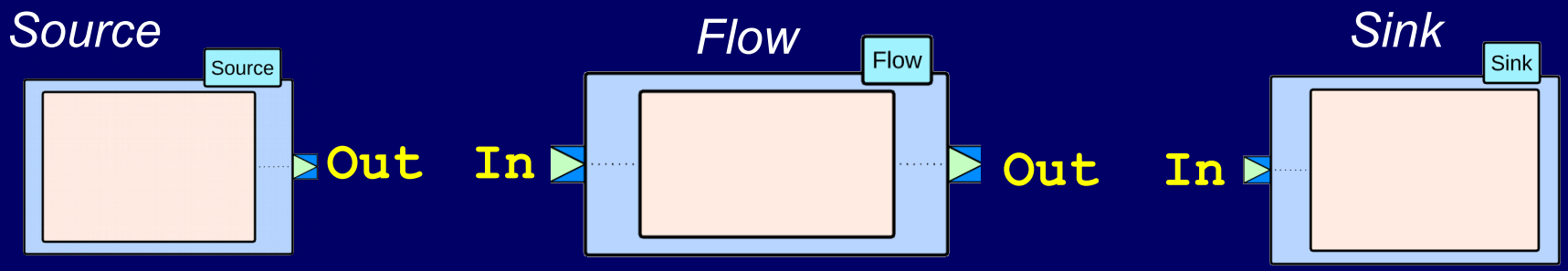
- Offrono un livello di astrazione decisamente superiore rispetto agli attori e comparabile a quello degli Java 8 Stream
  - calcolo “monadico” su flussi di dati
- Akka Stream provvede a tradurre (“materializzare”) le specifiche di un flusso di processamento in un sistema di attori
  - A ben vedere, visto che gli attori fanno uso del Fork/Join framework (come esecutore di default) si può pensare che le specifiche siano indirettamente tradotte in **`java.util.concurrent.ForkJoinTask`**
    - ✔ Come già accadeva per gli Java 8 Stream!



- Parallellizzazione resa possibile dal modello ad attori (>>)

Akka Stream API

- Un flusso di processamento può essere espresso tramite un grafo di nodi (**`akka.stream.Graph`**) di calcolo
  - Nodi Sorgenti: **`Source<Out,Mat>`**
  - Nodi Intermedi: **`Flow<In,Out,Mat>`**
  - Nodi Finali: **`Sink<In,Mat>`**



- Ciascuno specifica il tipo dei dati in ingresso e/o di uscita
- **`Mat`** è il tipo di un ulteriore dato (ad es. un eventuale dato aggregato) prodotto “fuori” dallo stream
  - se non serve basta usare **`NotUsed`**

Esempio di Akka Stream (Java 8)

- La creazione di una specifica completa, come usuale, si avvantaggia di molti factory method per processare come *akka stream* dati in I/O e collezioni tradizionali
- Per eseguire il processamento dello stream servono
  - Un grafo di computazione completo: **`Source + Sink`**;
    - ovvero un **`akka.stream.javadsl.RunnableGraph<Mat>`**
  - Un **`ActorSystem`** per *materializzarlo*

```
import akka.stream.javadsl.*;
[…altri import omessi…]…
static public void main(String[] args) {
  final ActorSystem system =
        ActorSystem.create("actor-system");
  final Materializer materializer =
        ActorMaterializer.create(system);
…
```

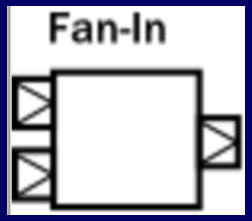
Esempio di Akka Stream (Java 8)

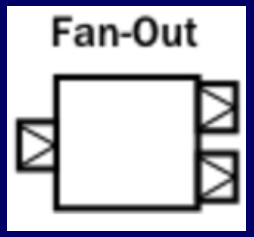
```
import akka.stream.javadsl.*;
[…altri import omessi…]…
static public void main(String[] args) {
  final ActorSystem system = …

  Source<Integer, NotUsed> source = Source.range(1, 256);

  Flow<Integer, String, NotUsed> flow =
    Flow.of(Integer.class)
      .map(i -> Integer.toBinaryString(i));

  Sink<String, CompletionStage<String>> sink =
    Sink.<String, String>fold("", (acc, elem) -> acc + elem + "\n");

  RunnableGraph<CompletionStage<String>> runnableGraph =
    source.via(flow).toMat(sink,Keep.right());

  CompletionStage<String> f = runnableGraph.run(materializer);
    f.thenAccept( s -> { System.out.println("Got:\n"+ s );
                         system.terminate(); });
}
```

```
Got:
1
10
11
100
…
11111110
11111111
100000000
```

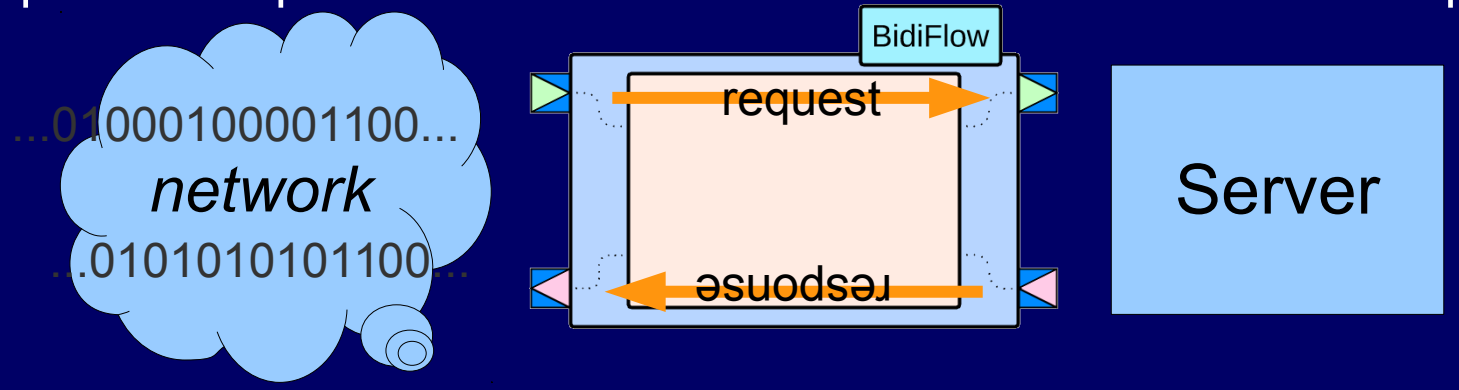
Più che Semplici Pipeline di Processamento

- Specifica di *flussi* di processamento *tipati*
- N.B. *flusso* di processamento e non *pipeline*
  - si possono creare complessi grafi di processamento
  - le pipeline risultano particolari e semplici casi di flusso *lineari*
- E' possibile utilizzare nodi con più di un'uscita e più di un'entrata



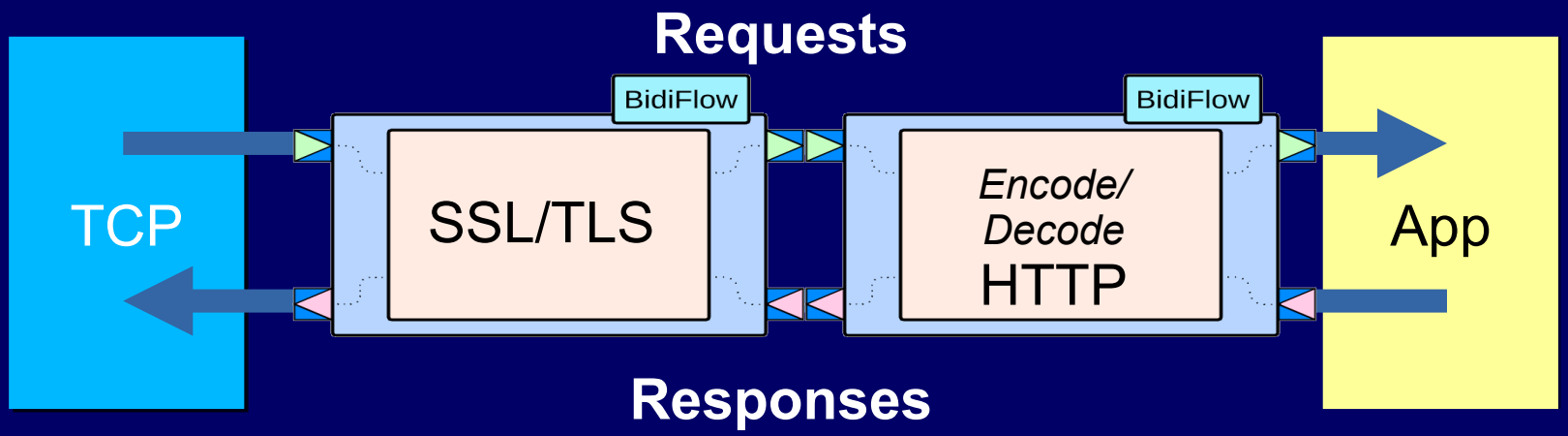


- Ad esempio, **`BidiFlow`** (doppia entrata / uscita) torna utile ogni qualvolta è prevista un fase di codifica/decodifica richiesta/risposta

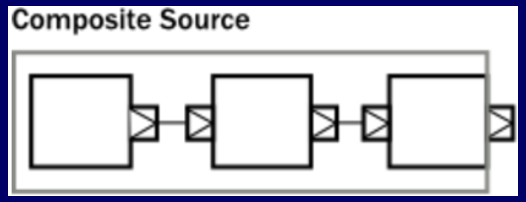


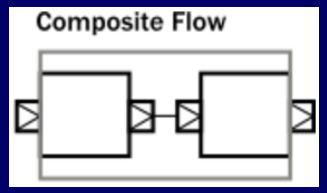
Akka Stream ed Akka Http

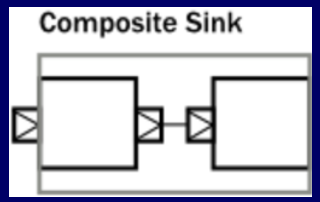
- Un delle motivazioni dietro lo sviluppo di **`akka-stream`** è stata proprio la necessità di supportare lo sviluppo di un altro modulo indispensabile
- **`akka-http`**



Componibilità degli Akka Stream

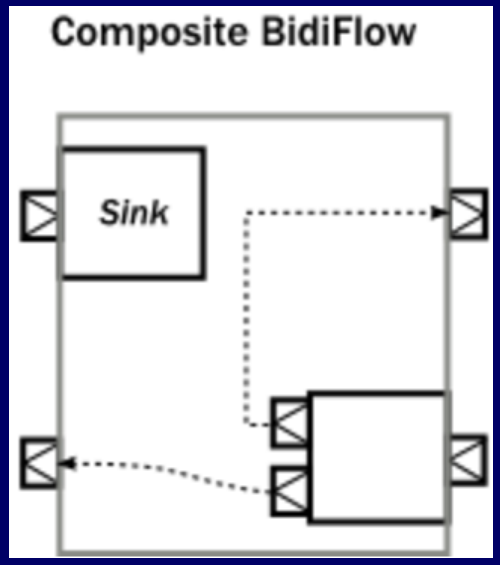






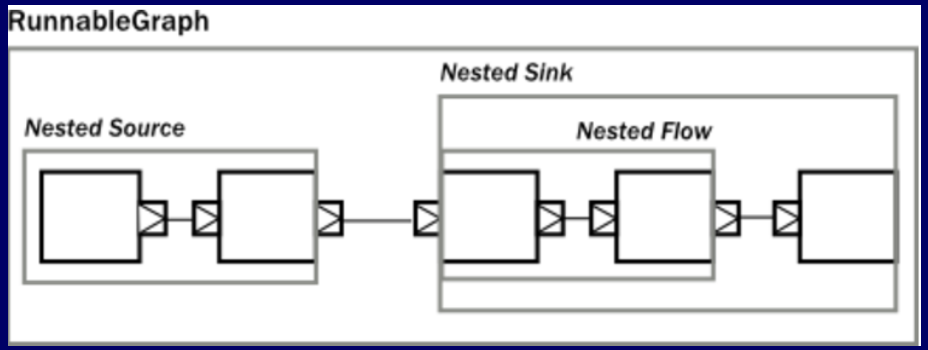
La composizione può essere nidificata a piacimento. E' possibile creare:

- una **`Source`** da un **`Graph`** a singola uscita
- un **`Sink`** da un **`Graph`** a singola entrata
- un **`Flow`** da un **`Graph`** a singola entrata ed uscita
- ...



akka.stream.javadsl.RunnableGraph<Mat>

- Per eseguire il processamento è necessario definire un **RunnableGraph** completo, ovvero un grafo che includa una **Source** ed un **Sink**




```
final Source<Integer, NotUsed> nestedSource =
  Source.single(0)    // An atomic source
    .map(i -> i + 1) // an atomic processing stage
    .named("nestedSource"); // wraps up the current Source and name it

final Flow<Integer, Integer, NotUsed> nestedFlow =
  Flow.of(Integer.class).filter(i -> i != 0) // an atomic processing stage
    .map(i -> i - 2) // another atomic processing stage
    .named("nestedFlow"); // wraps up the Flow, and gives it a name

final Sink<Integer, NotUsed> nestedSink =
  nestedFlow.to(Sink.fold(0, (acc, i) -> acc + i)) // wire sink nestedFlow
    .named("nestedSink"); // wrap it up

// Create a RunnableGraph
final RunnableGraph<NotUsed> runnableGraph = nestedSource.to(nestedSink);
```

Materializzazione

- E' la trasformazione di una specifica di processamento di stream (espressa come un **`RunnableGraph`**) in un sistema di attori effettivamente eseguibile

- In realtà, se non specificato diversamente, tutti i nodi sono resi con un *singolo* attore (allo scopo di risparmiare gli hand-off tra attori)
- Invocando invece il metodo **`Graph.async()`**…

async

`Graph<S,M> async()`

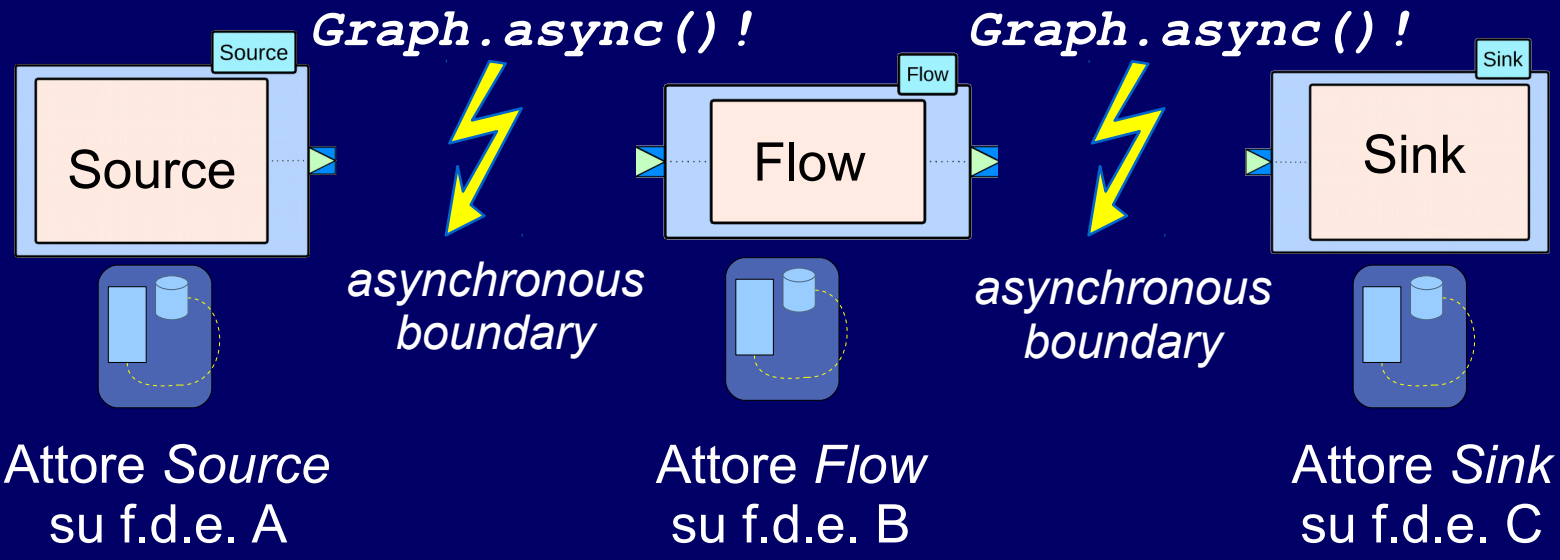
Put an asynchronous boundary around this Graph

Returns:

(undocumented)

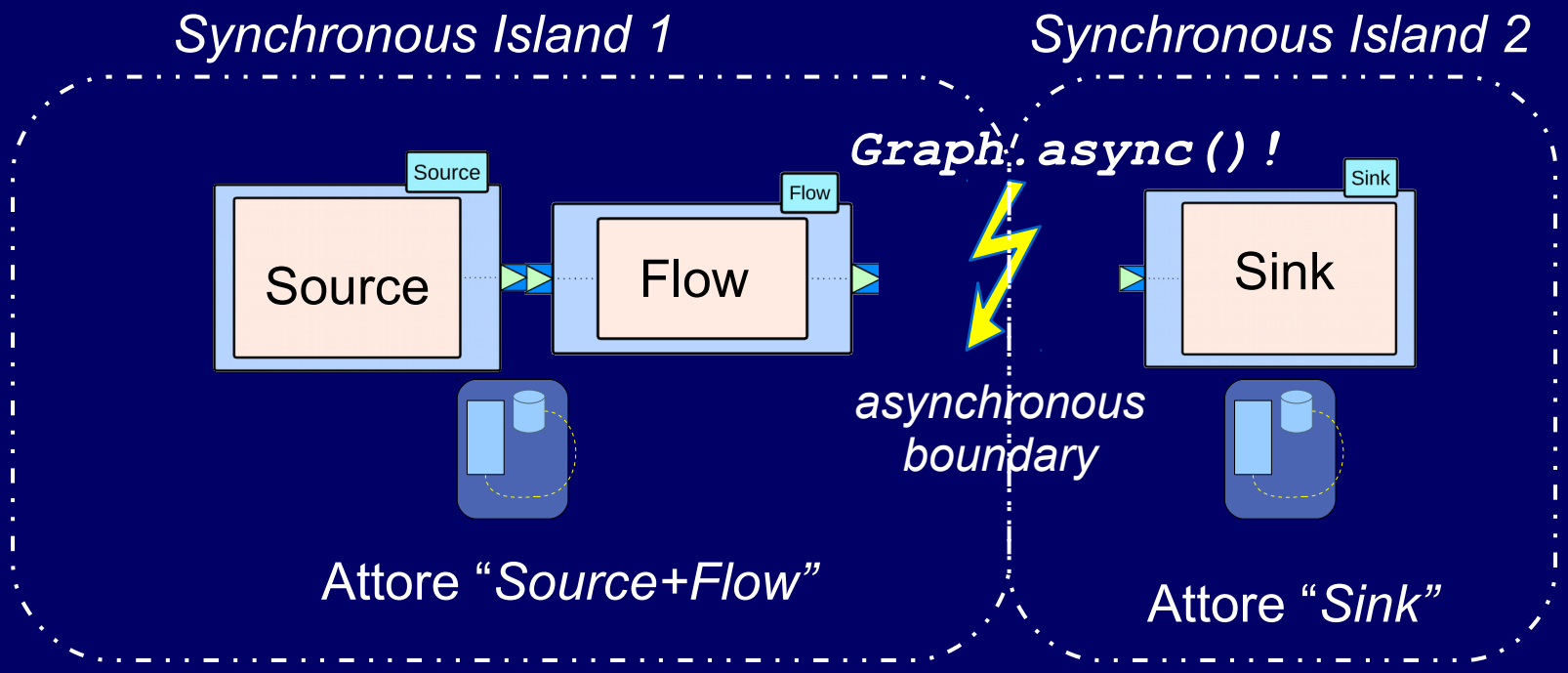
Akka Stream ed Asynchronous Boundary

- Gli asyncronous boundary rappresentano un concetto chiave della fase di materializzazione
- Il metodo **`Graph.async()`** permette di introdurli dietro esplicita richiesta
- Supponiamo che un akka stream sia materializzato con un attore distinto per ogni nodo del grafo materializzato
  - La materializzazione del più classico dei **`Graph`** apparirebbe siffatta:



Synchronous Island

- Con il metodo **`async()`** è possibile controllare la formazione di cosidette *synchronous island*
- Ovvero, sequenze di nodi di un **`RunnableGraph`** che sono rese con un singolo attore in fase di materializzazione



Synchronous Island e Speed-Up

- Un ovvio obiettivo è ottenere ragionevoli speed-up senza particolari sforzi di programmazione
  - Per default: unica synchronous island
- Quindi, per default, lo speed-up non c'è!
  - Scelta forse sorprendente… ed in parte spiegabile con la relativa giovinezza della libreria
  - Comunque modificabile introducendo apposite chiamate a `Graph.async()` per forzare la materializzazione in più attori
- Inoltre si richiede l'uso di *buffer* opportuni per disaccoppiare lo scambio di msg tra attori
  - Il tempo necessrio per i vari stadi di processamento potrebbe differire sensibilmente creando disparità di carico tra i diversi attori
  - Non vanno confusi con e le *mailbox* interne agli attori: questi non sono usate allo scopo perché l'implementazione della back-pressure ha richiesto l'uso di buffer alternativi per il loro disaccoppiamento

Akka Stream Speed-Up

```
import java.math.BigInteger;
import akka.stream.*;
import akka.stream.javadsl.*;
[…altri import omessi…]
public static void primes(int bsize, int n, int min, int max)
  throws InterruptedException, ExecutionException {
  final ActorSystem system = ActorSystem.create("actor-system");
  final Materializer materializer = ActorMaterializer.create(
  ActorMaterializerSettings.create(system)
      .withInputBuffer(bsize, bsize), system);
  Source<Integer, NotUsed> source = Source.range(1,n);
  Flow<Integer, BigInteger, NotUsed> flow = primeFlow(min, max);
  Sink<BigInteger,CompletionStage<BigInteger>> sink = Sink.last();
  RunnableGraph<CompletionStage<BigInteger>> runnableGraph =
    source.via(flow).toMat(sink,Keep.right());
  final long start = System.currentTimeMillis();
  CompletionStage<BigInteger> f = runnableGraph.run(materializer);
  f.thenAccept( p -> {  System.out.println(bsize+"\t"+n+"\t"+
                    (System.currentTimeMillis()-start)/n+"\tms");
                        system.terminate(); });
  f.toCompletableFuture().get(); // solo per aspettare la fine
}
```

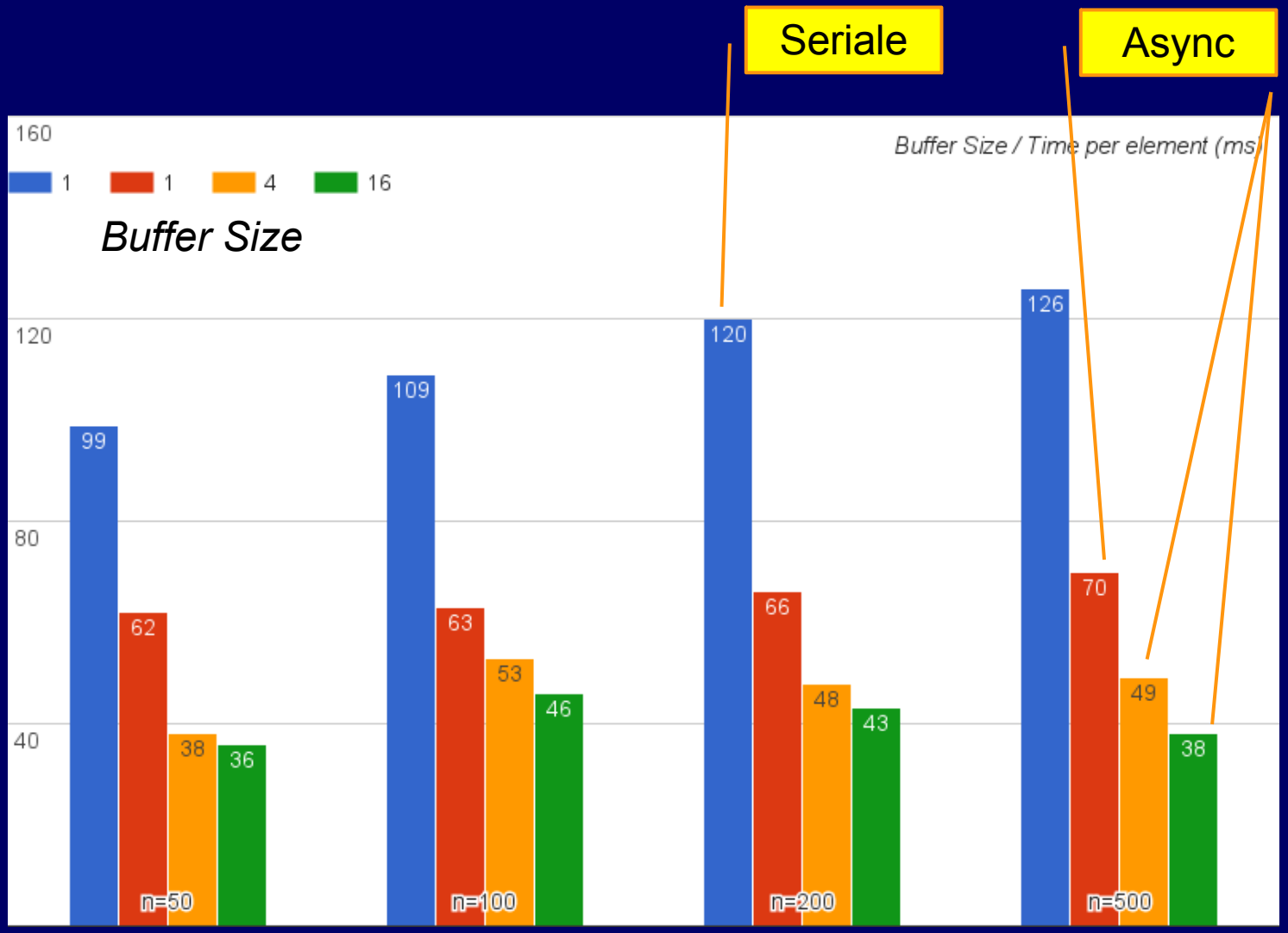
Next Probable Prime

```
public static Flow<Integer, BigInteger, NotUsed>
              primeFlow(int min, int max) {
   Flow<Integer, BigInteger, NotUsed> flow =  Flow.of(Integer.class)
     .map(i -> BigInteger.valueOf(numeroCasuale(min, max)))
     .map(i -> factorial(i))
     .map(i -> i.nextProbablePrime() )
     .map(i -> i.nextProbablePrime() )
     .map(i -> i.nextProbablePrime() )
     .map(i -> i.nextProbablePrime() )
     .map(i -> i.nextProbablePrime() )
     .map(i -> i.nextProbablePrime() )
     .map(i -> i.nextProbablePrime() );
   return flow;
}

final private static Random rnd = new Random();
final private static BigInteger one = BigInteger.valueOf(1);

public static int numeroCasuale(int MIN_NUMBER, int MAX_NUMBER) {
  return MIN_NUMBER+rnd.nextInt(MAX_NUMBER);
}
private static BigInteger factorial(BigInteger n) {
  if (n.equals(one)) return one;
  else return n.multiply(factorial(n.subtract(one)));
}
```

Next Probable Prime Async

```
public static Flow<Integer, BigInteger, NotUsed>
              primeFlow(int min, int max) {
   Flow<Integer, BigInteger, NotUsed> flow =  Flow.of(Integer.class)
     .map(i -> BigInteger.valueOf(numeroCasuale(min, max)))
     .map(i -> factorial(i)).asynch()
     .map(i -> i.nextProbablePrime() ).asynch()
     .map(i -> i.nextProbablePrime() ).asynch()
     .map(i -> i.nextProbablePrime() ).asynch()
     .map(i -> i.nextProbablePrime() ).asynch()
     .map(i -> i.nextProbablePrime() ).asynch()
     .map(i -> i.nextProbablePrime() ).asynch()
     .map(i -> i.nextProbablePrime() );
   return flow;
}

final private static Random rnd = new Random();
final private static BigInteger one = BigInteger.valueOf(1);

public static int numeroCasuale(int MIN_NUMBER, int MAX_NUMBER) {
  return MIN_NUMBER+rnd.nextInt(MAX_NUMBER);
}
private static BigInteger factorial(BigInteger n) {
  if (n.equals(one)) return one;
  else return n.multiply(factorial(n.subtract(one)));
}
```

Akka Stream Speed-Up



Dichiaratività degli Akka Stream

- Gli Akka Stream consentono di separare nitidamente la specifica di ***cosa*** si vuole ottenere da ***come*** ottenerla
  - Un linguaggio dichiarativo per il processamento di stream
  - Analogamente alle query SQL
    - vengono tradotte in un linguaggio procedurale eseguibile (come l'algebra relazionale e derivati)
    - i **`RunnableGraph`** sono tradotti in un sistema di attori
- Viene anche offerto un DSL (ma solo in Scala)
  - Un linguaggio per esprimere i grafi di processamento in maniera ancora più compatta ed espressiva tramite operatori dedicati
- Tutto questo permette esecuzioni concorrenti senza nemmeno conoscere i dettagli della parallelizzazione
  - Ottimo! Ma solo nei casi più semplici… serve comunque una buona consapevolezza del processo per tutti gli altri casi

Il Livello di Consapevolezza Richiesto

- Diverse possibili traduzioni possono risultare semanticamente equivalenti, ma talune, anche dipendentemente dall'ambiente di esecuzione, possono risultare più efficienti
  - Ottimizzazioni: al momento è ancora necessario fornire decisive indicazioni per ottenere speed-up ragguardevoli...
  - Serve una conoscenza di base del modello di esecuzione...
  - ...sapere quello che si sta facendo,…
  - …perché,...
  - ...e prepararsi a qualche misura sperimentale.
- Discorsi simili valgono praticamente per tutti i linguaggi per il processamento di stream, locali e distribuiti

✔… e spesso la curva di apprendimento trae in inganno

- È possibile eseguire semplici esempi ed i primi tutorial con facilità
- E' possibile ottenere quello che serve in produzione solo dopo aver capito praticamente tutto!

Ambienti di Esecuzione

- Gli Akka Stream si prestano a materializzazioni per diversi ambienti di esecuzione, sia su singolo nodo, che distribuiti
  - cluster deployment
- Al momento **`akka-stream`**, internamente, supporta solo la materializzazione in un actor system locale
  - prevedibilmente questa limitazione sarà rimossa...
- La distribuzione è comunque già possibile utilizzando e/o integrando progetti esterni
  - *Kafka*: **`akka-stream-kafka`**
    - integrazione con Akka / Kafka
      - (vedi anche progetto *alpakka*)
  - *GearPump*: fornisce un materializzatore per ambienti distribuiti utilizzabile direttamente con le specifiche di processamento di stream espressi tramite *akka stream*

Akka Stream: Conclusioni

- Un linguaggio di programmazione dichiarativo per il processamento di stream di dati
- Reintroducono una tipizzazione forte
  - Lasciata da parte dal modello ad attori
- Aumentano notevolmente il livello di astrazione a cui si programma e prevedibilmente, per molte applicazioni, è un livello più idoneo del modello ad attori
- Il modello ad attori sembra invece essere già particolarmente apprezzato, a livello industriale, come corretto livello di astrazione al quale risolvere i problemi della PC che sorgono in fase di *materializzazione* di una specifica dichiarativa per il processamento di stream (vedi progetto *GearPump*)

I Confini di PC

- La distribuzione della computazione diventa inevitabile quando la mole di dati da processare e/o la velocità di processamento richiesta non è più ottenibile con una singola macchina
- La distribuzione introduce certamente nuove problematiche
- *Oltre…* quelle già affrontate in questo corso...

Riferimenti

- *http://akka.io/*
  - *http://blog.akka.io/streams/2016/07/06/threading-and-concurrency-in-akka-streams-explained*
  - *http://blog.akka.io/integrations/2016/08/23/intro-alpakka*
- *http://reactive-streams.org/*
- *Raymond Roestenburg, Rob Bakker, Rob Williams.* Akka in Action - *Manning*
- Akka Concurrency - Building reliable software in a multi-core world. *Derek Wyatt. Artima.*
- *RxJava: https://github.com/ReactiveX/RxJava*
- *GearPump: https://gearpump.apache.org/overview.html*
- *Kafka: https://kafka.apache.org/*
- Credits: diverse presentazioni Typesafe/Lightbend