Taming and Composing High-Performance Stream Processing

reactive programming is a programming paradigm that maintains a continuous interaction with their environment, but at a speed which is determined by the environment, not the program itself

Gèrard Berry

Agenda

- The rise of Reactive Streams
- Reactive Extensions and back pressure
- Actor Model, does not fit well
- Reactive Streams
- Akka Streams
- Short demo & code review

Objectives

- React to high rate events
- Compose events from different sources
- Simplify complex logic
- Pipeline for combinators over event stream

Introduction - Riccardo Terrell

- Originally from Italy, currently - Living/working in Washington DC ~10 years
- +/- 19 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- Author of the book “Functional Concurrency in .NET” - Manning
- *Polyglot programmer - believes in the art of finding the right tool for the job*

@trikace *rickyterrell.com* *tericcardo@gmail.com*

Microsoft® Most Valuable Professional

Why Reactive?

- Billions of internet connected devices
- Data is transformed and pushed continuously
- Services must be always up
- Must be fast

bytes = 1000000000000000000
1000 petabytes
1 million terabytes
1 billion gigabytes

one **exabyte** is one quintillion bytes

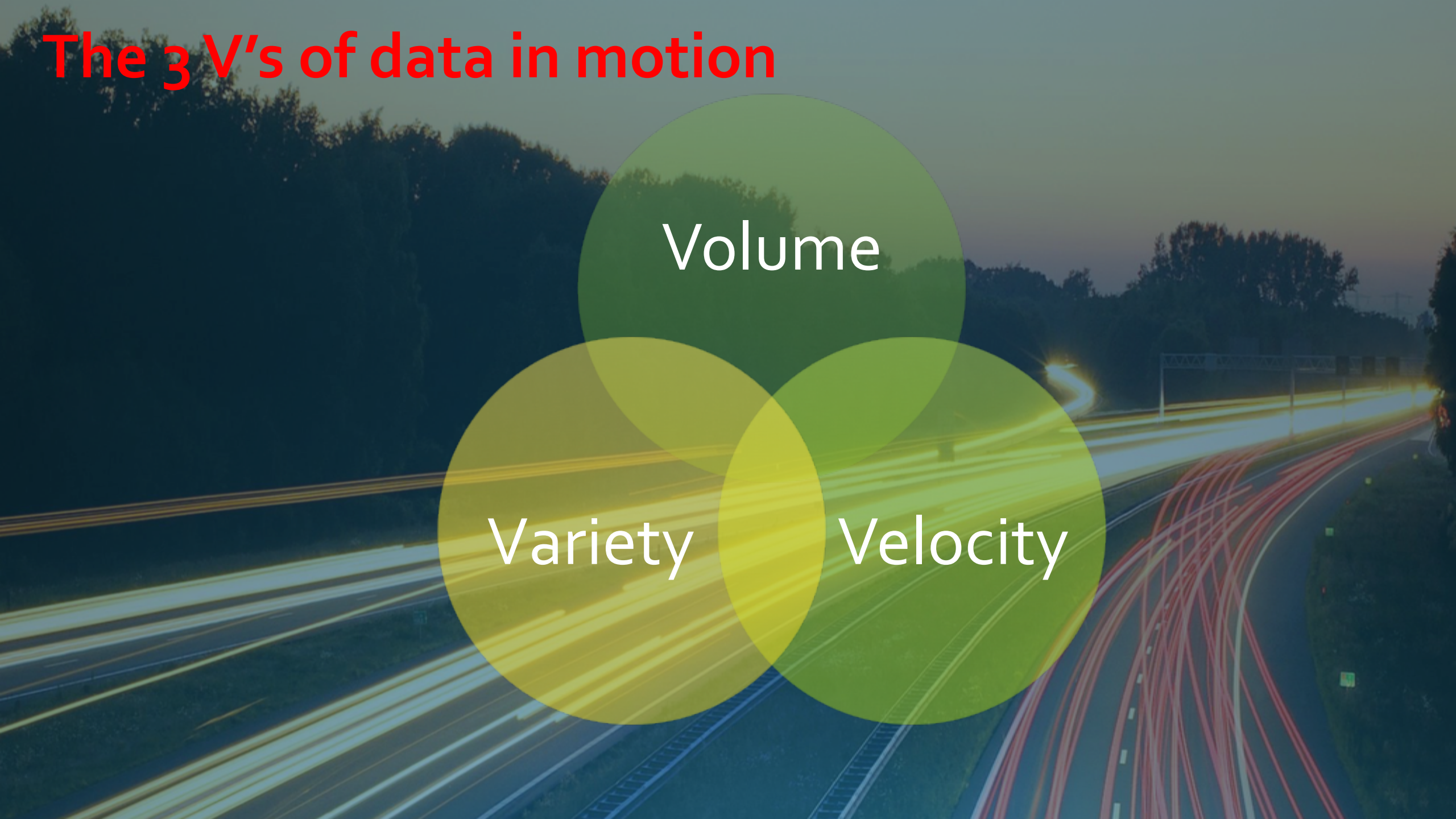
GOOGLE

- Current storage = 15 exabytes
- Processed per day = 100 petabytes
- Number of pages indexed = 60 trillion
- Unique search users per month > 1 billion
- Searches per second = 2.3 million

The Rise of Event Stream Systems

It's all about data these days. Data helps us make informed decisions. Big Data helps us make informed and insightful decisions. Big streams of data help us make informed, insightful and timely decisions. These continuously flowing streams of data are often called event streams. It's increasingly common to build software systems whose primary purpose is to process event streams.

The 3 V's of data in motion

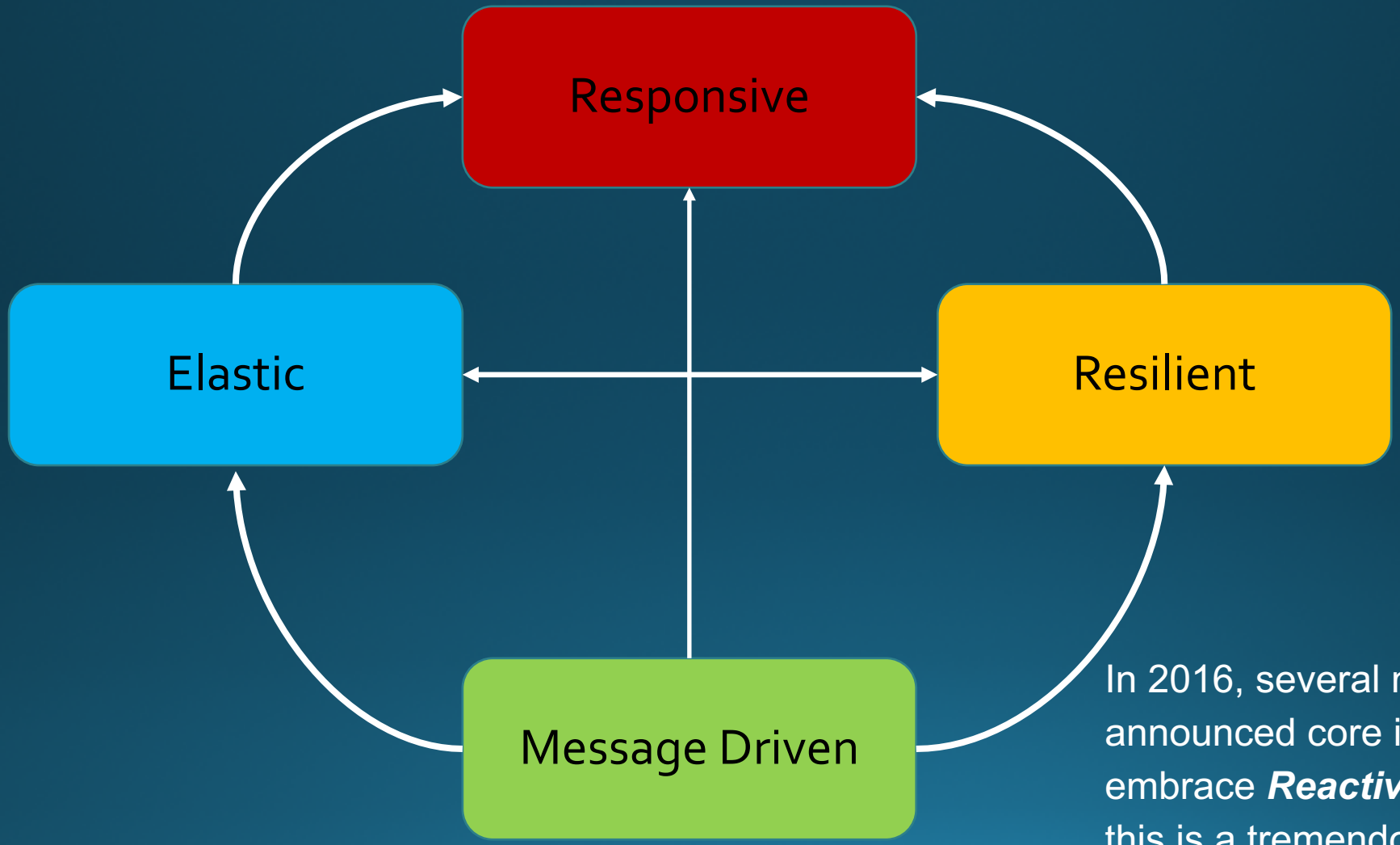


New Tools for a New Era

We to need to build systems that:

- *react* to events — **Event-Driven**
- *react* to load — **Scalable**
- *react* to failure — **Resilient**
- *react* to users — **Responsive**

Reactive Manifesto



In 2016, several major vendors have announced core initiatives to embrace ***Reactive Programming*** this is a tremendous validation of the problems faced by companies today.

http://www.reactivemanifesto.org

Functional Reactive Programming

Functional ~~Reactive~~ Programming

- declarative
- functions as values
- side-effects free
- referential transparency
- immutable
- composition

Twitter Paper

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a pow-

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

Twitter Paper

Your Server as a Function

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

It's really clear that the imperative style of programming has run its course. ... We're sort of done with that. … However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains.

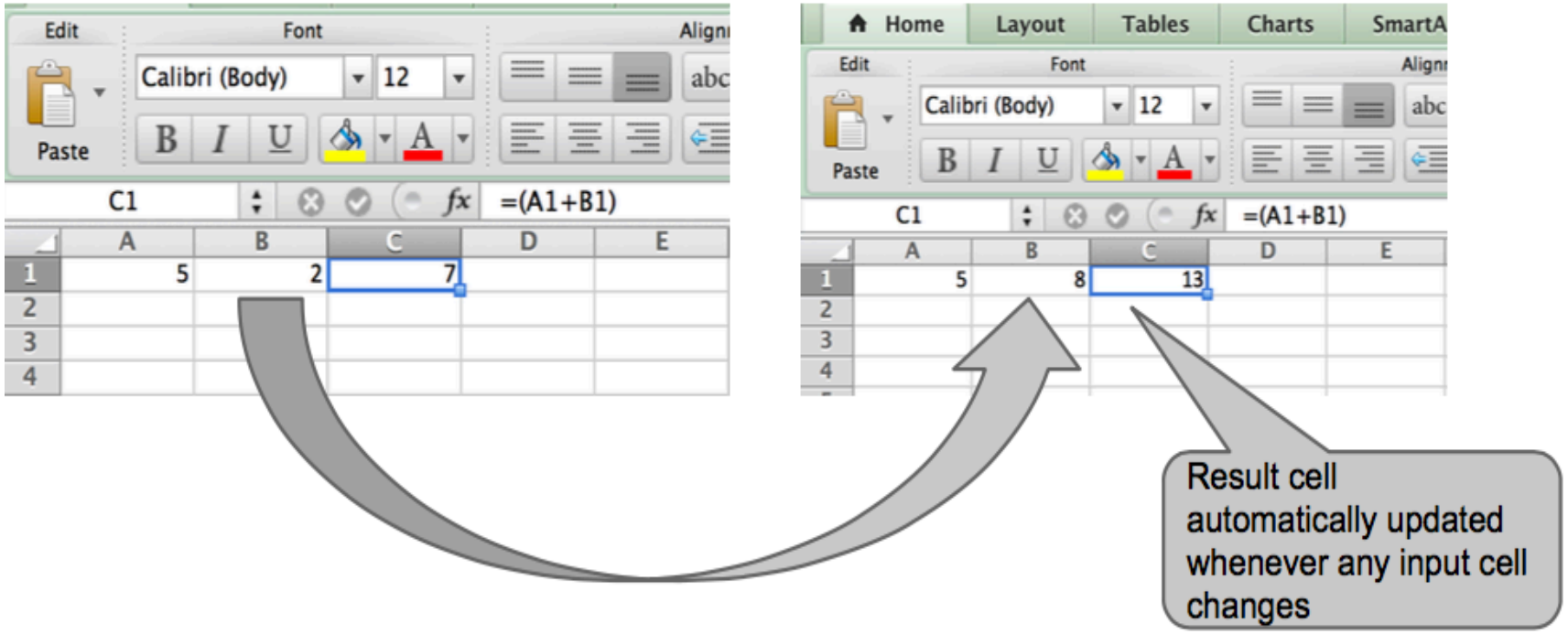
-Anders Hejlsberg
C# Architect

(from his MIX07 keynote)

~~Functional~~ Reactive Programming

What is Reactive Programming

SpreadSheet == Mother of All Reactive Programming



Functional Reactive Programming

Push-Pull Functional Reactive Programming

Push-Pull Functional Reactive Programming

Conal Elliott

LambdaPix

conal@conal.net

Abstract

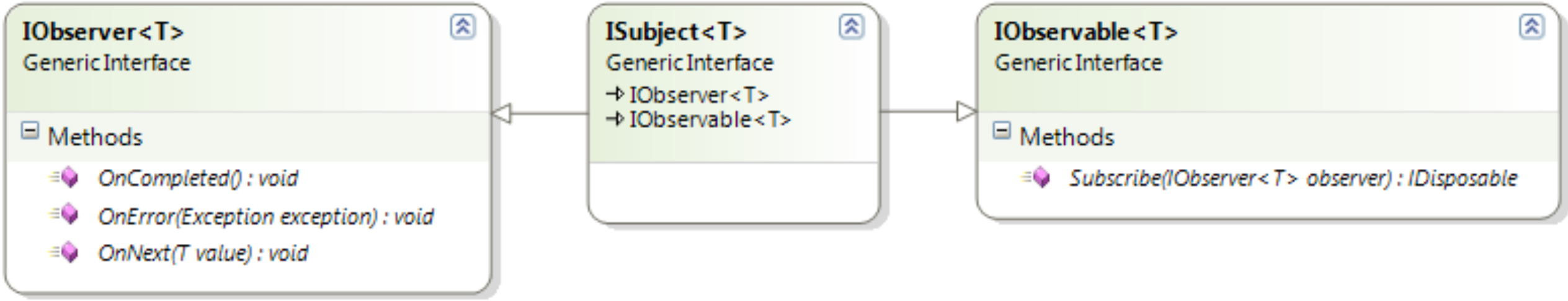
Functional reactive programming (FRP) has simple and powerful semantics, but has resisted efficient implementation. In particular, most past implementations have used demand-driven sampling, which accommodates FRP's continuous time semantics and fits well with the nature of functional programming. Consequently, values are wastefully recomputed even when inputs don't change, more composable than their finite counterparts, because they can be scaled arbitrarily in time or space, before being clipped to a finite time/space window.

While FRP has simple, pure, and composable semantics, its efficient implementation has not been so simple. In particular, past implementations have used demand-driven (pull) sampling of reactive behaviors, in contrast to the data-driven (push) evaluation typ-

Reactive Extensions

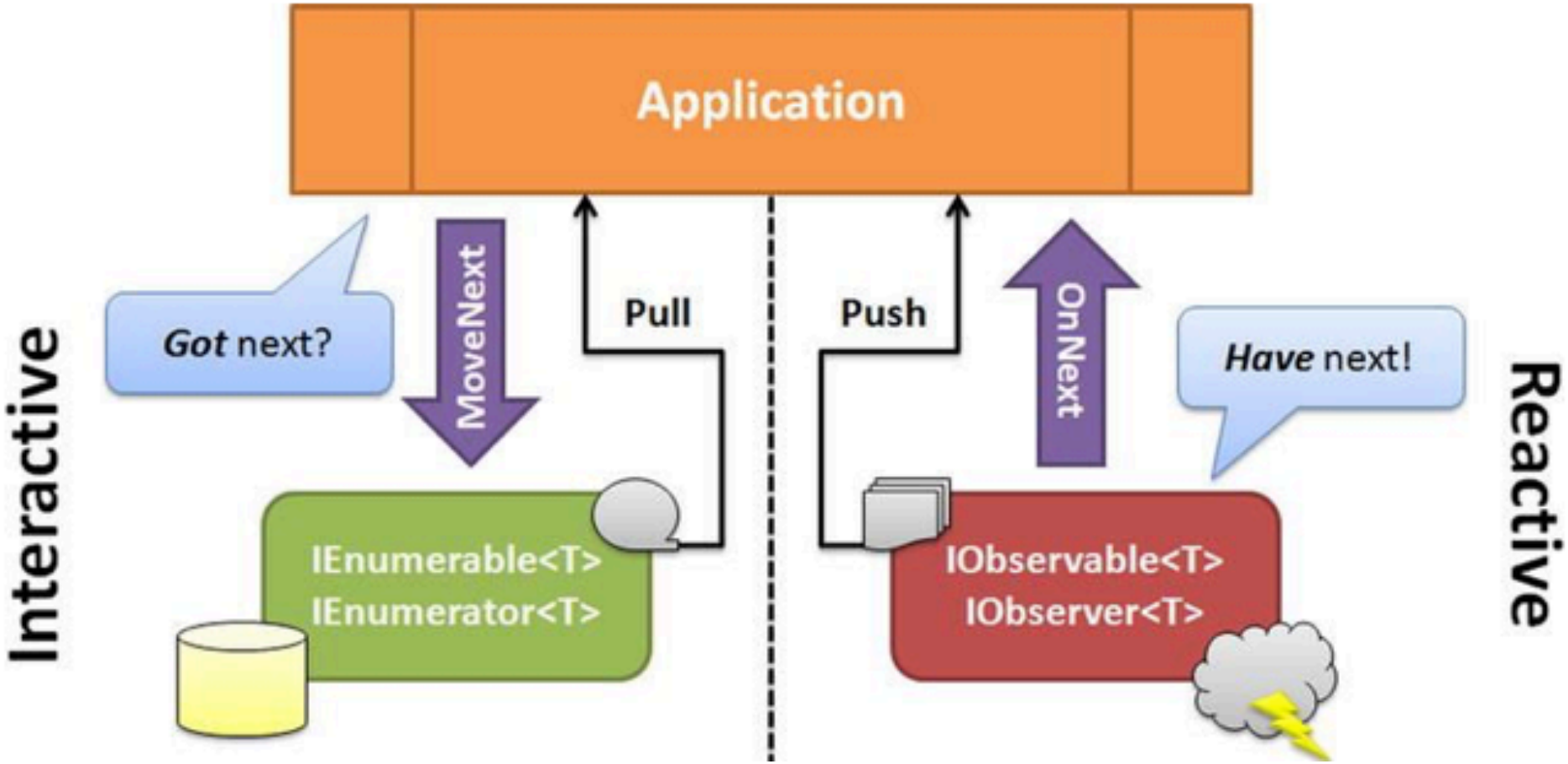
What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators




Push vs Pull

Push vs Pull


Application
Interactive
Got next?
MoveNext
Pull
IEnumerable<T>
IEnumerator<T>
Push
OnNext
Have next!
IObservable<T>
IObserver<T>
Reactive

IObserver & IObservable

```
type IObserver<'a> = interface
    abstract OnCompleted : unit -> unit
    abstract OnError : exn -> unit
    abstract OnNext : 'a -> unit
end

type IObservable<'a> = interface
    abstract Subsrcribe : IObserver<'a> -> IDisposable
end
```

IObservable the dual of IEnumerable

```
type IObserver<'a> = interface
  abstract OnNext : 'a with set
  abstract OnCompleted : unit -> unit
  abstract OnError : Exception -> unit
end


type IObservable<'a> = interface
  abstract Subscribe : IObserver<'a>
                    -> IDisposable
end
```

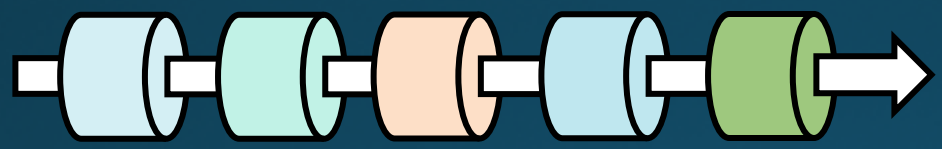
```
type IEnumerator<'a> = interface
   interface IDisposable
   interface IEnumerator

   abstract Current : 'a with get
   abstract MoveNext : unit -> bool
end

type IEnumerable<'a> = interface
   interface IEnumerable

   abstract GetEnumerator : IEnumerator<'a>
end
```

What is Rx?

Rx **compose pipeline** of operations over **single** or **multiple event's source**

Focus on what happens **between** the **Producer** and the **Consumer**

Rx is the **Pipeline**



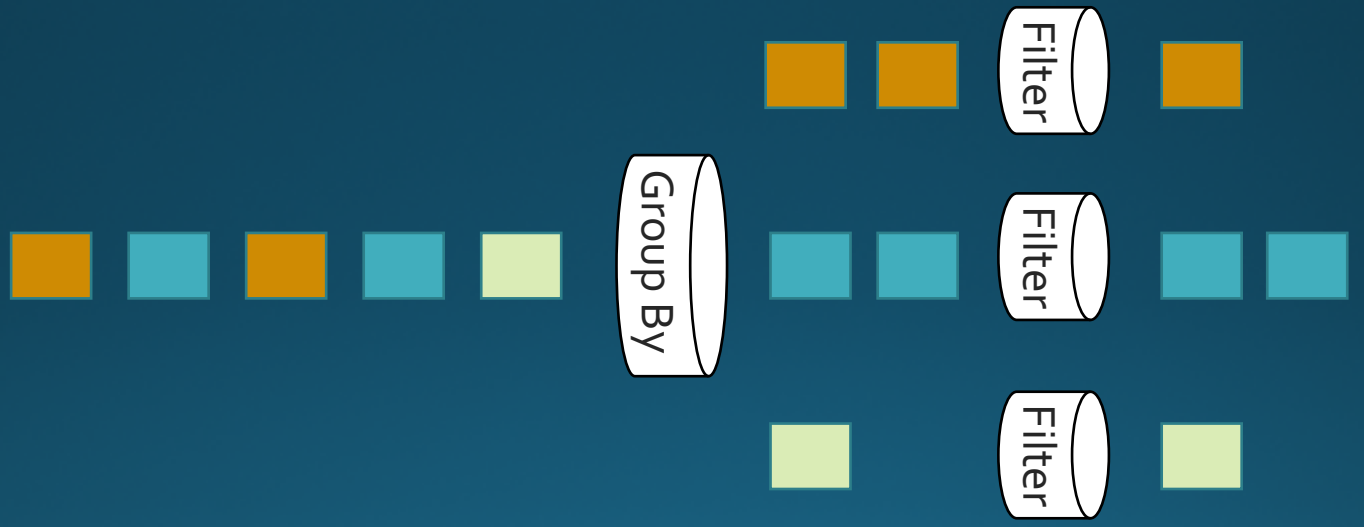
Rx = Async Data Stream Composition

Rx = LINQ to Async Data Stream

What is Rx?

Rx **compose pipeline** of operations over **single** or **multiple event's source**

Focus on what happens **between** the **Producer** and the **Consumer**

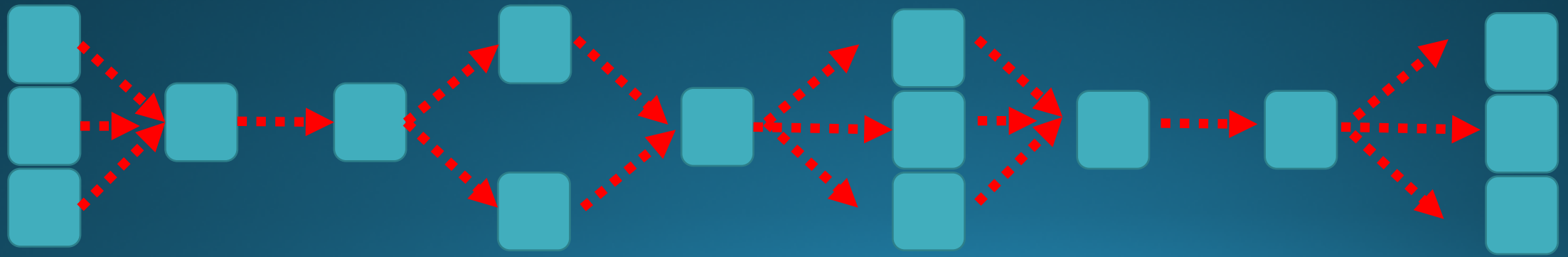


Rx = Async Data Stream Composition

Rx = LINQ to Async Data Stream

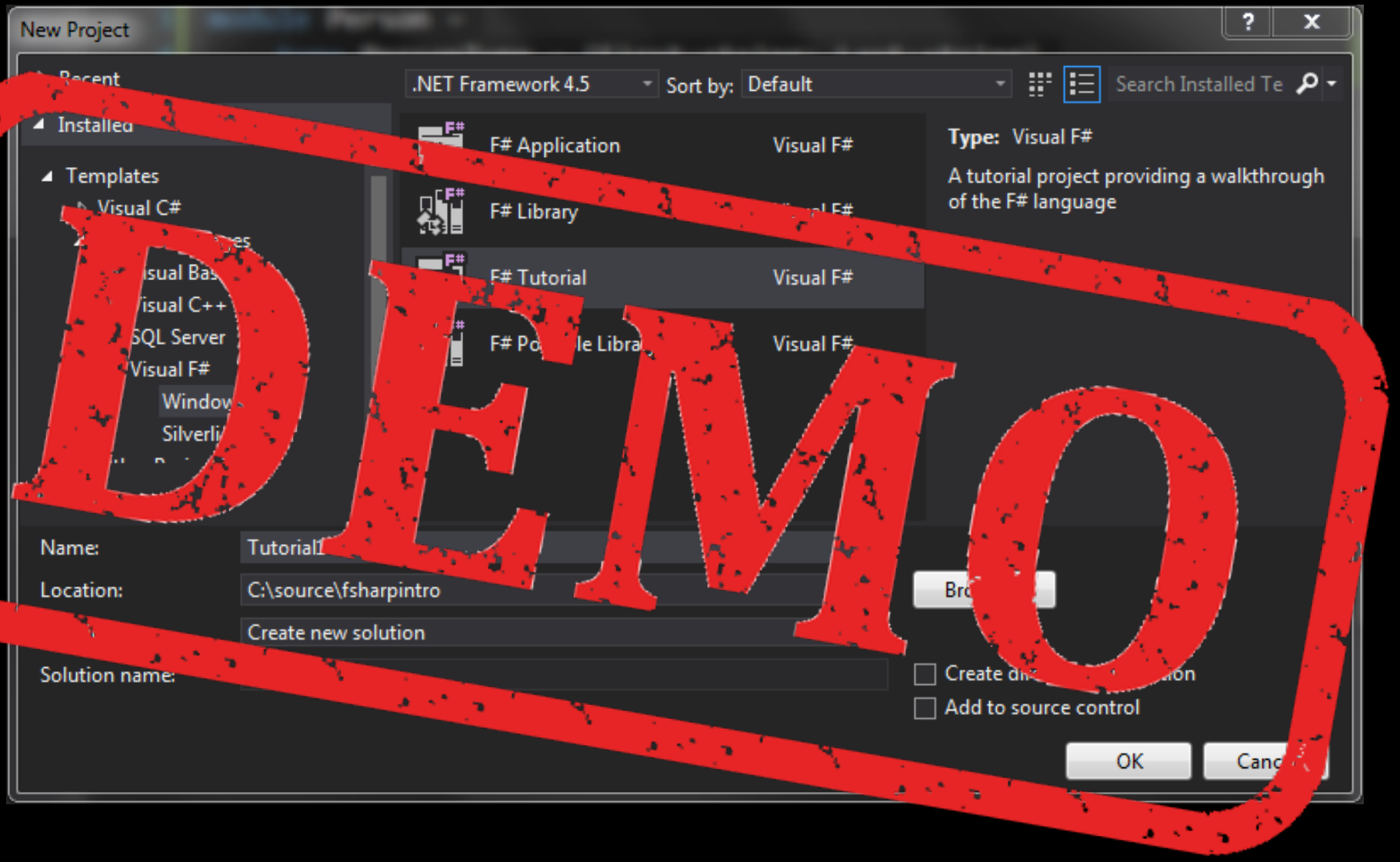
What is a Stream?

- ephemeral flow of data
- possibly unbounded in size
- focused on describing transformation
- can be formed into processing networks



Reactive Extensions

```
stockFiles
      .ObservableStreams(StockData.Parse, 10)
      .Throttle(TimeSpan.FromMilliseconds(10))
      .ObserveOn(ctx)
      .GroupBy(stocks => stocks.Symbol)
      .SelectMany(group => group
                           .Subscribe(stock =>
                                  UpdateChart(chart, stock));
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Reactive Extensions

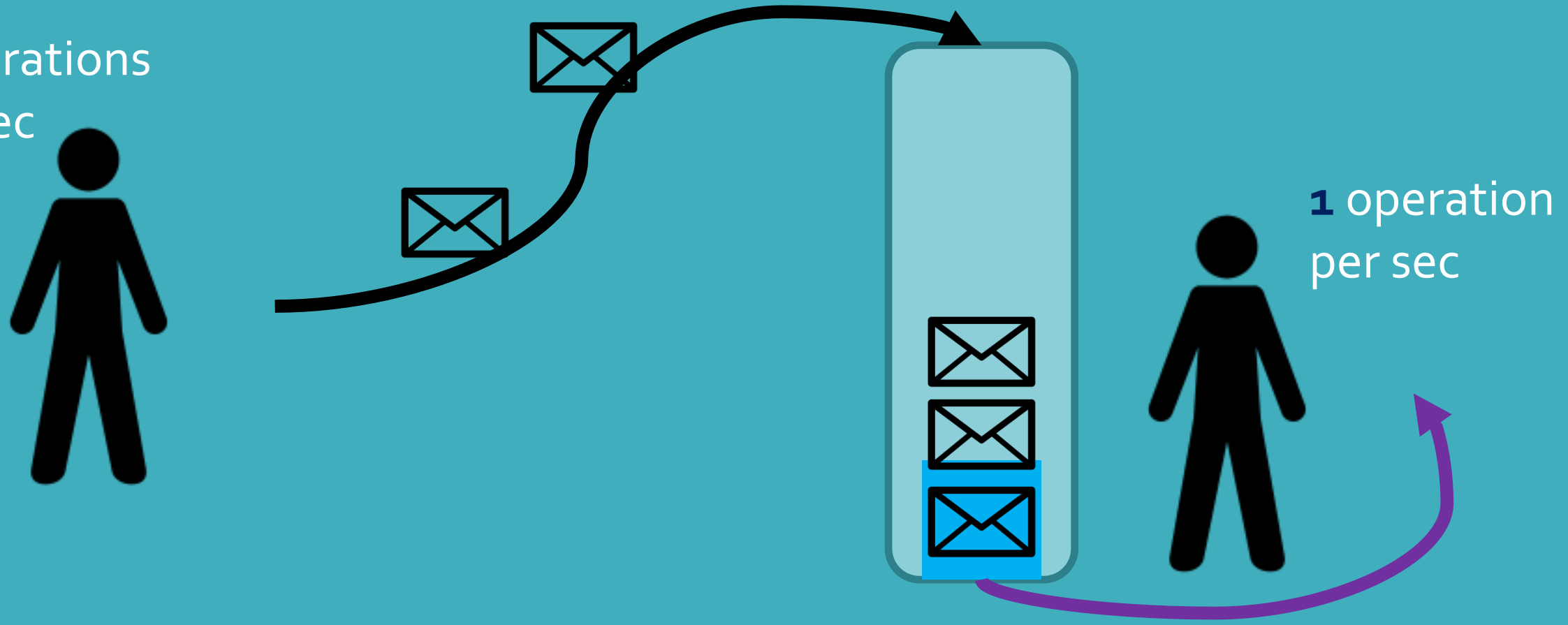
```
stockFiles
      .ObservableStreams(StockData.Parse, 10)
      .Throttle(TimeSpan.FromMilliseconds(10))
      .ObserveOn(ctx)
      .GroupBy(stocks => stocks.Symbol)
      .SelectMany(group => group
                           .Subscribe(stock =>
                                  UpdateChart(chart, stock));
```

Throttle is destructive

Push Model

Subscriber usually has some kind of buffer.

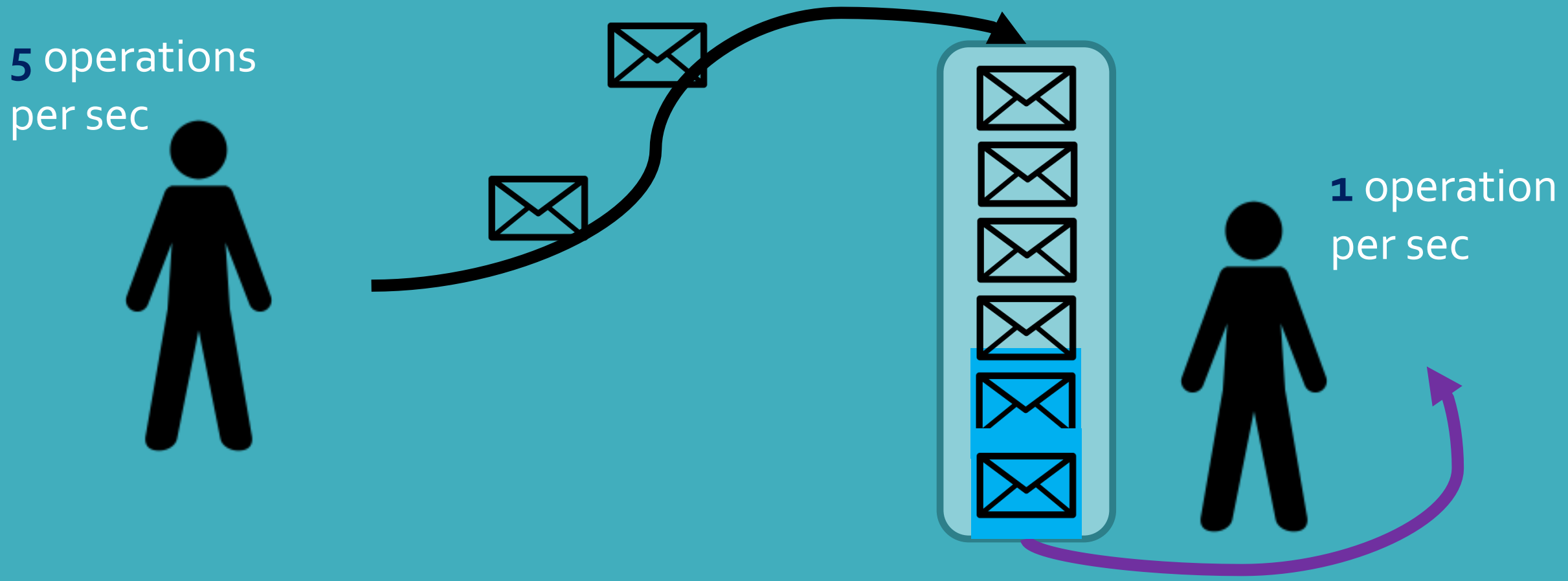
5 operations
per sec



1 operation
per sec

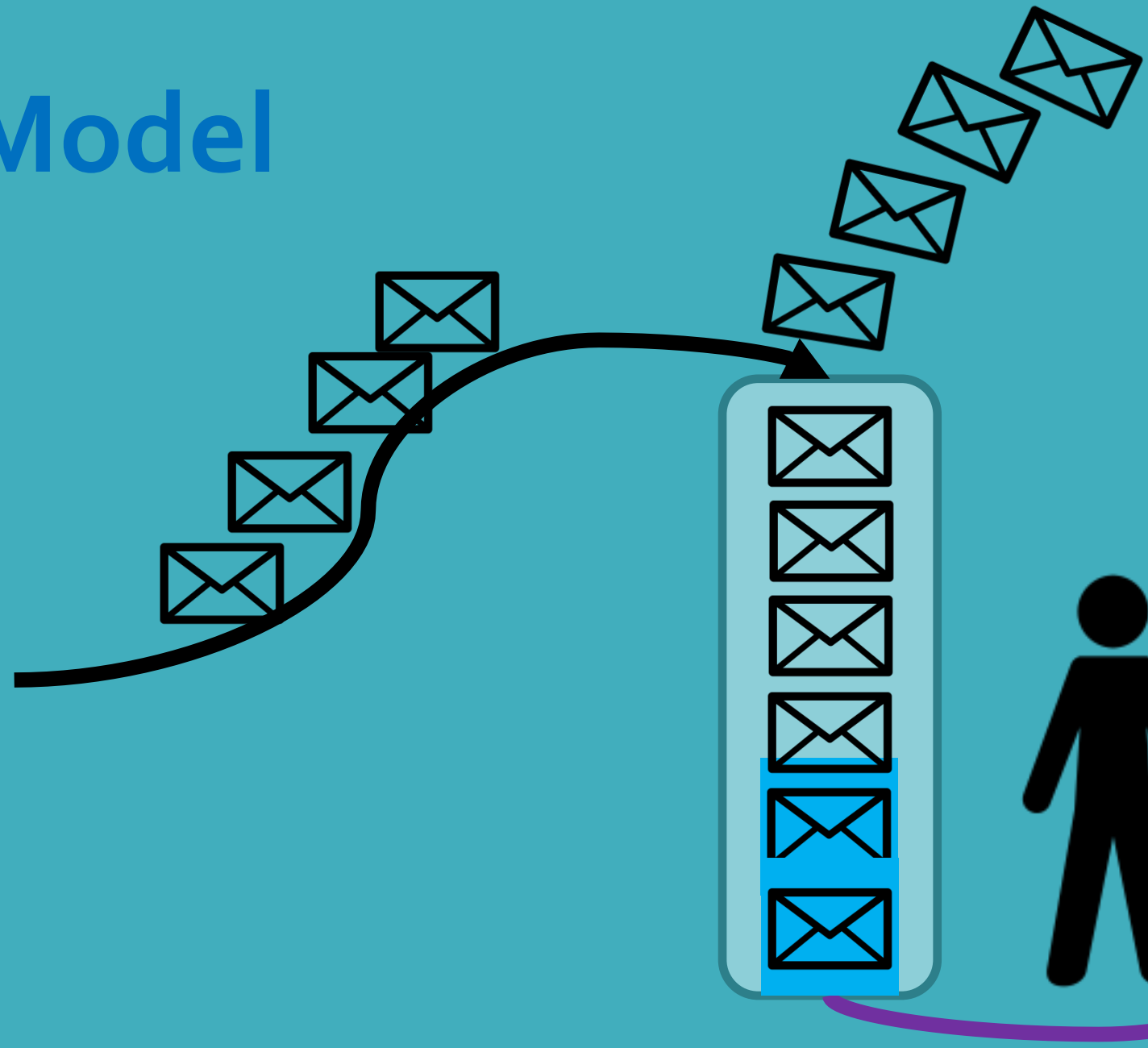
Push Model

Subscriber usually has some kind of buffer.



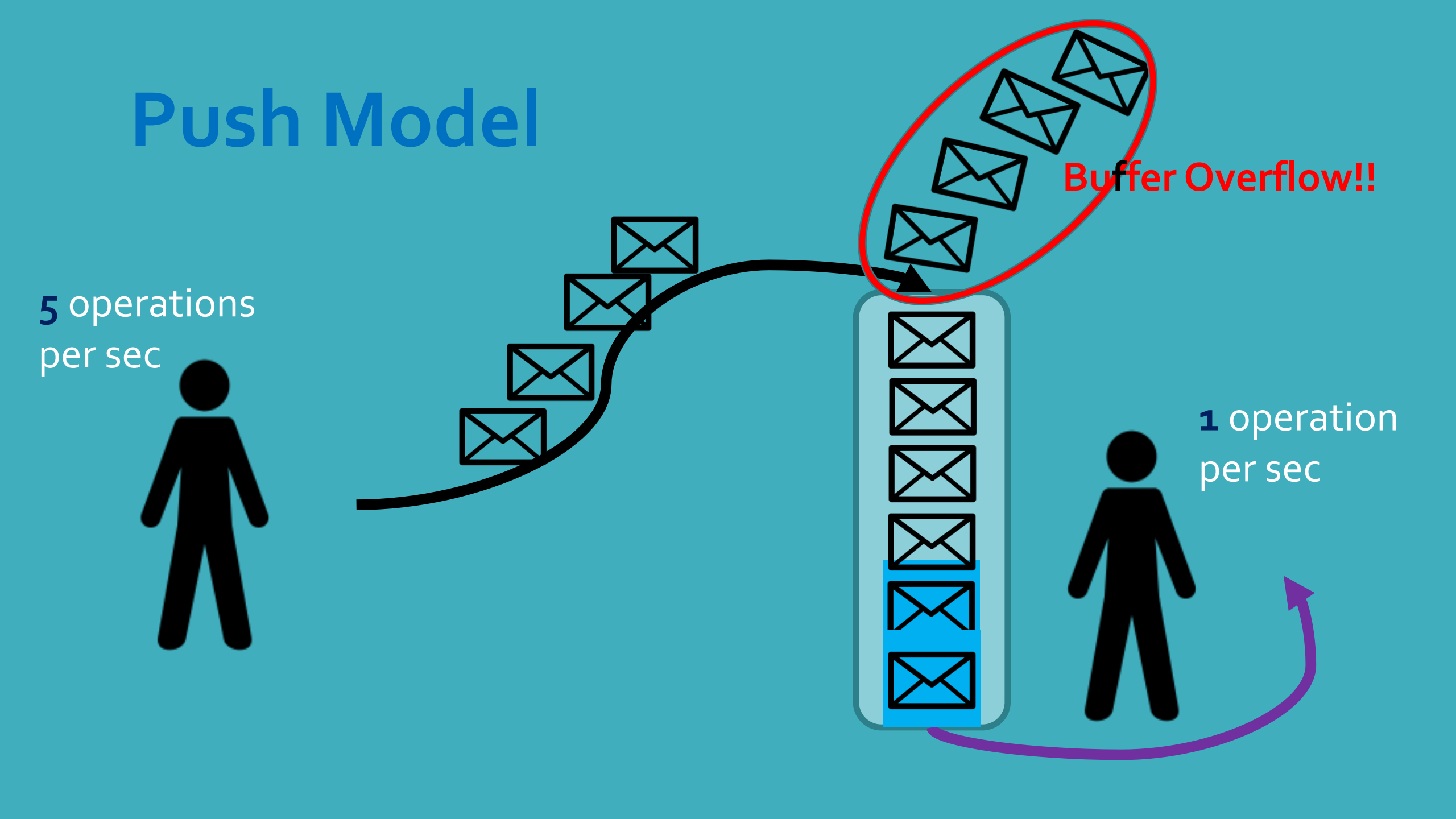
Push Model

5 operations
per sec



1 operation
per sec

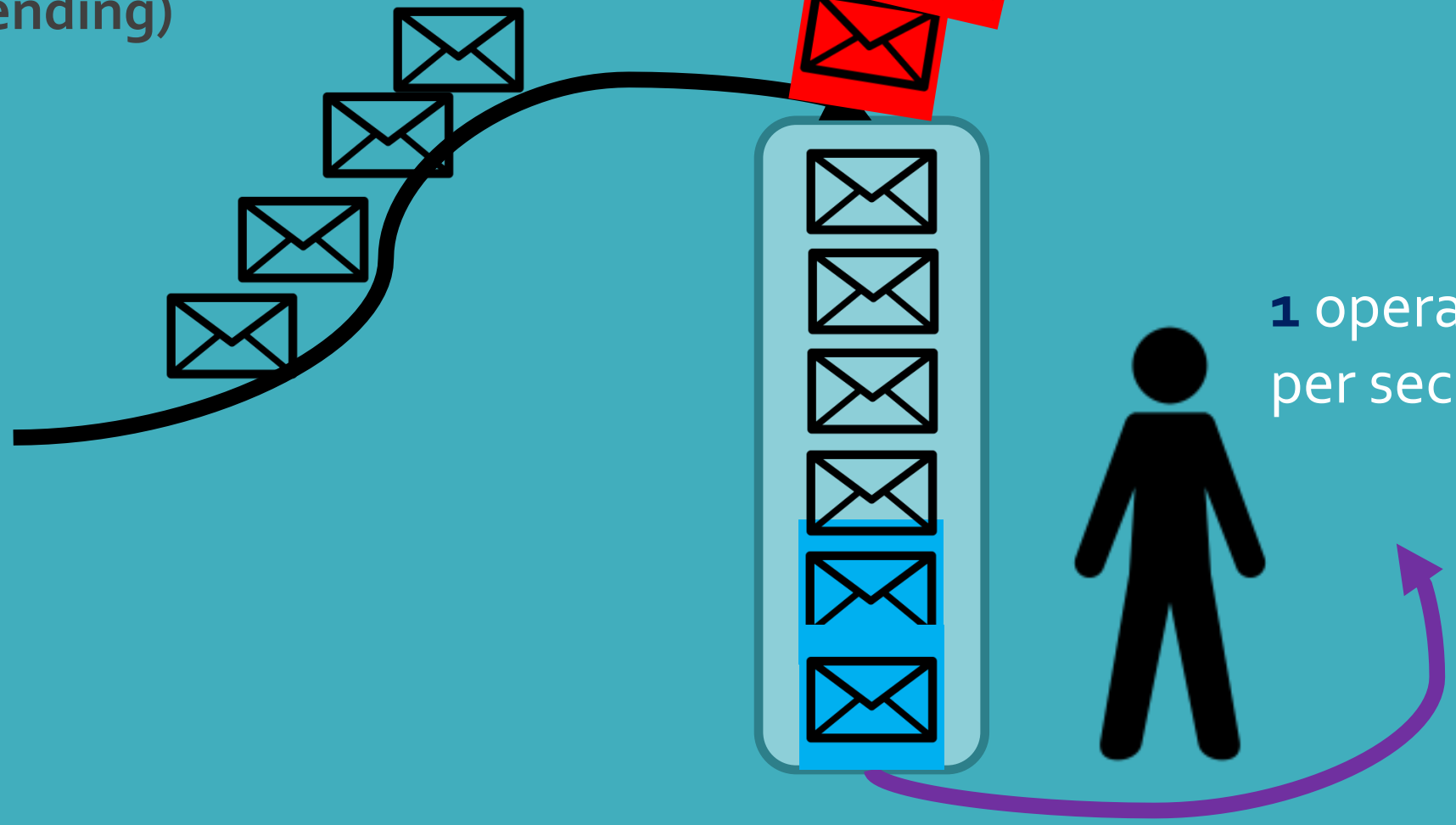
Push Model



Push Model

Bounded buffer drop exceeded messages (require re-sending)

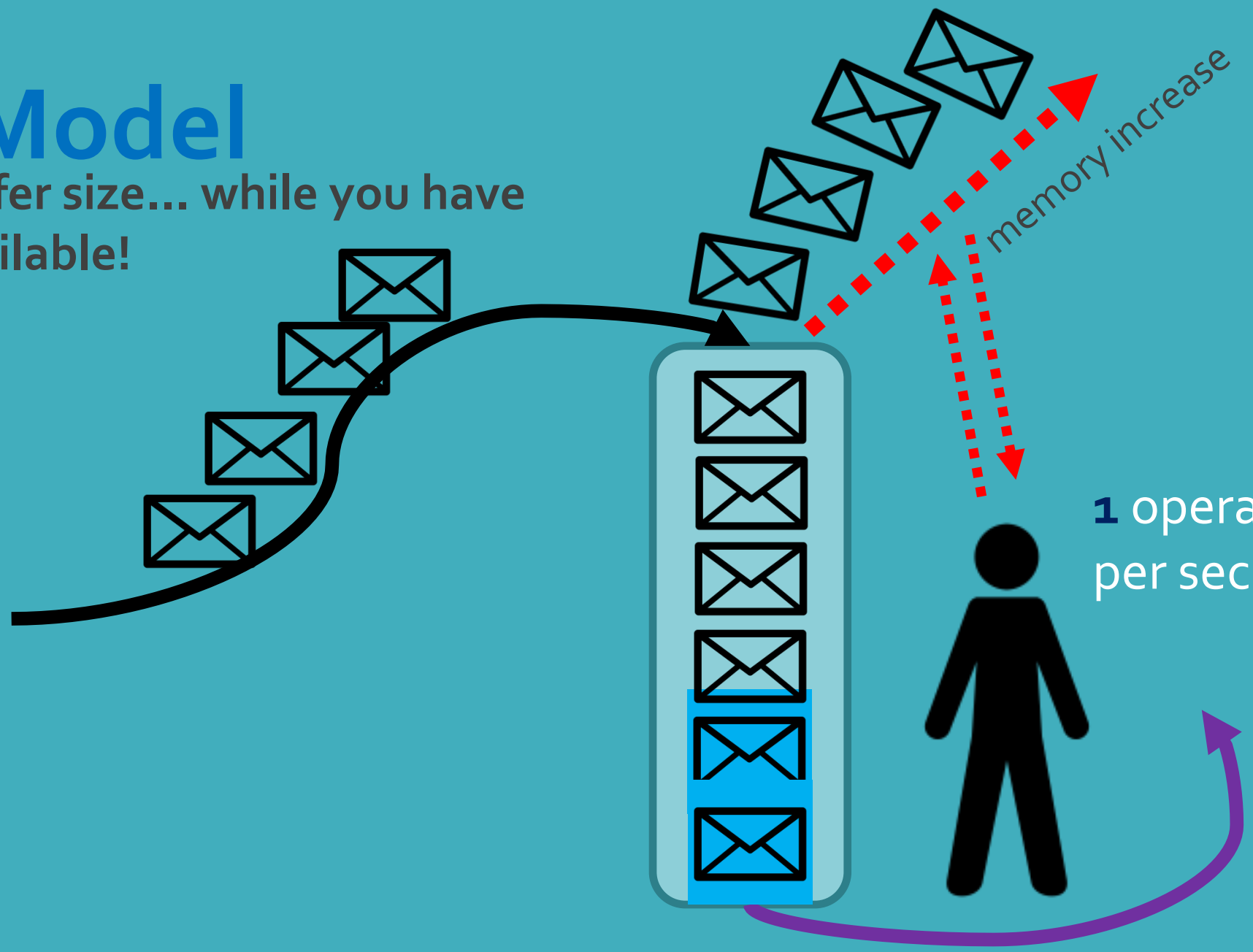
5 operations per sec



Push Model

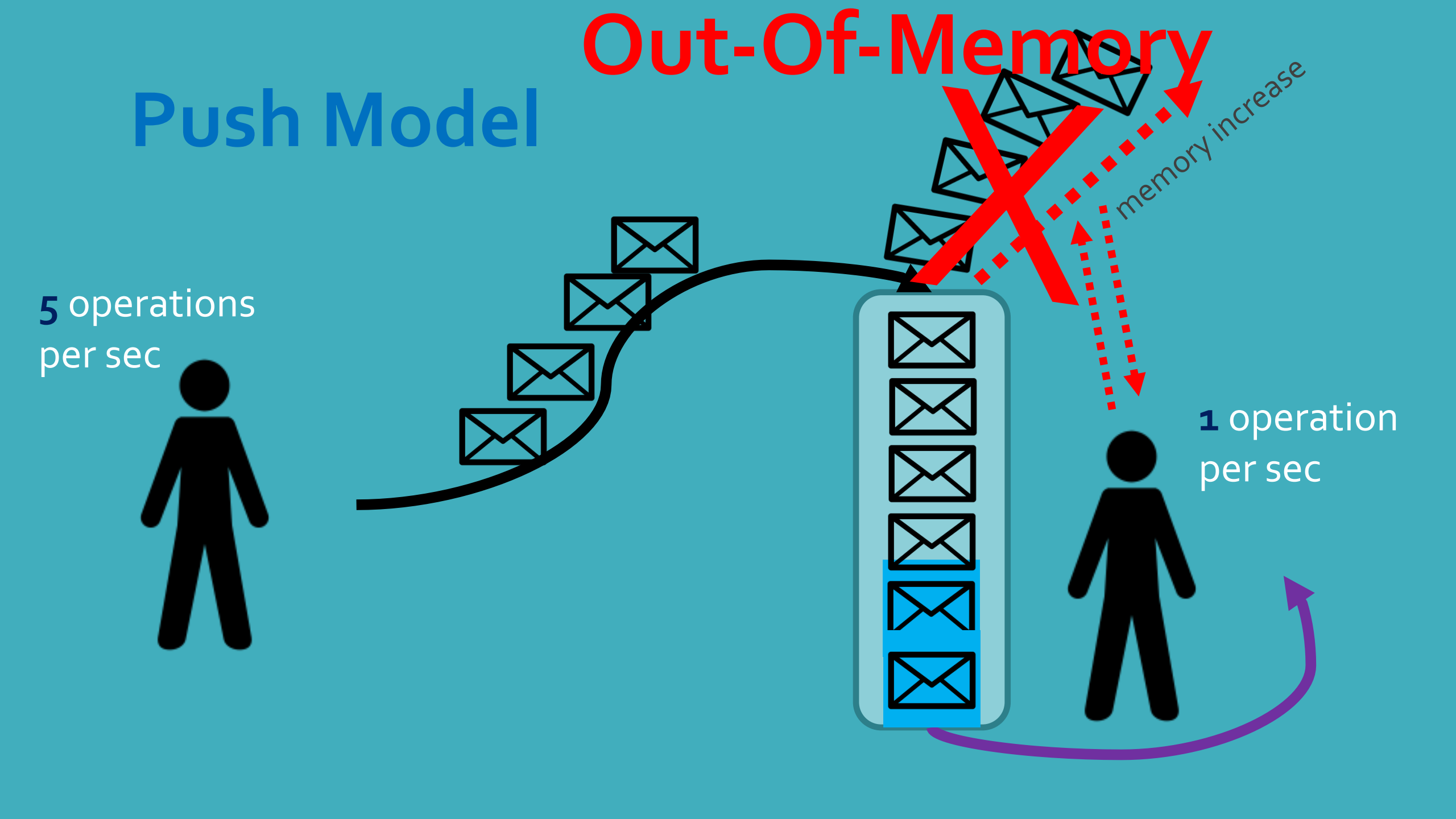
Increase buffer size… while you have memory available!

5 operations per sec



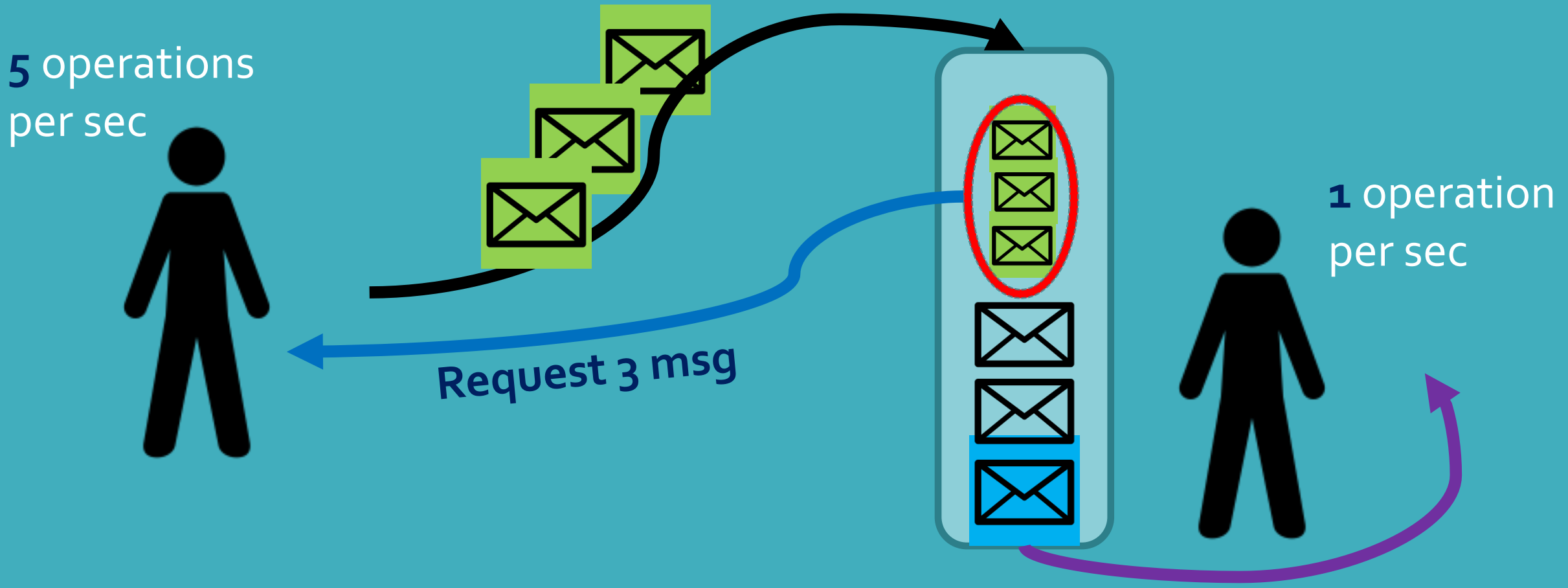
1 operation per sec

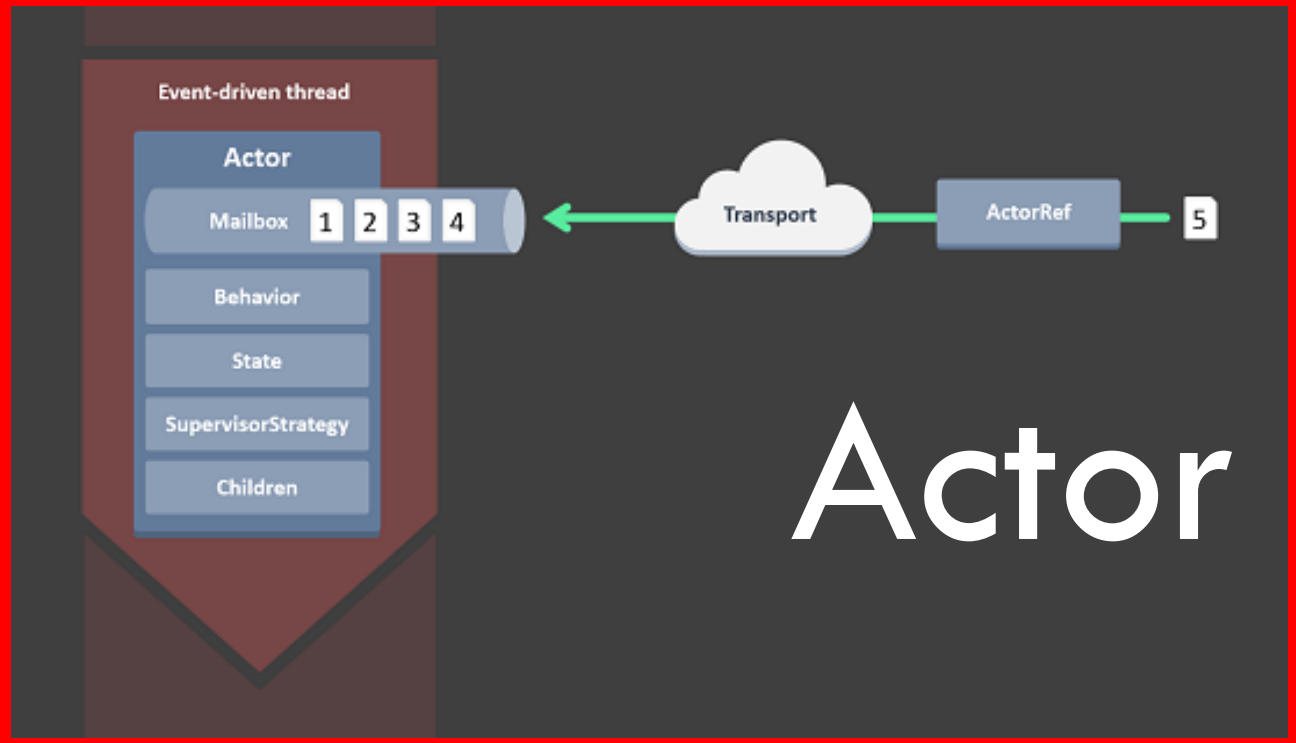
Push Model


Out-Of-Memory
memory increase
5 operations per sec
1 operation per sec

Dynamic Push-Pull

Pull based backpressure




Event-driven thread
Actor
Mailbox
1
2
3
4
Behavior
State
SupervisorStrategy
Children
Transport
ActorRef
5

Actor Model

Actor is a single and independent unit of computation

Actor Model Message-Driven

What is an Actor?

- **Share Nothing**
- **Message are passed by value**
- Light weight processes/threads communicating through messaging
- Communication only by messages
- Lightweight object
- Processing
- Storage – State
- Running on it’s own thread.
- Messages are buffered in a “mailbox”

Actor

What an Actor Offers

High throughput

Fault tolerance

No manual thread management

Scale Out & Up

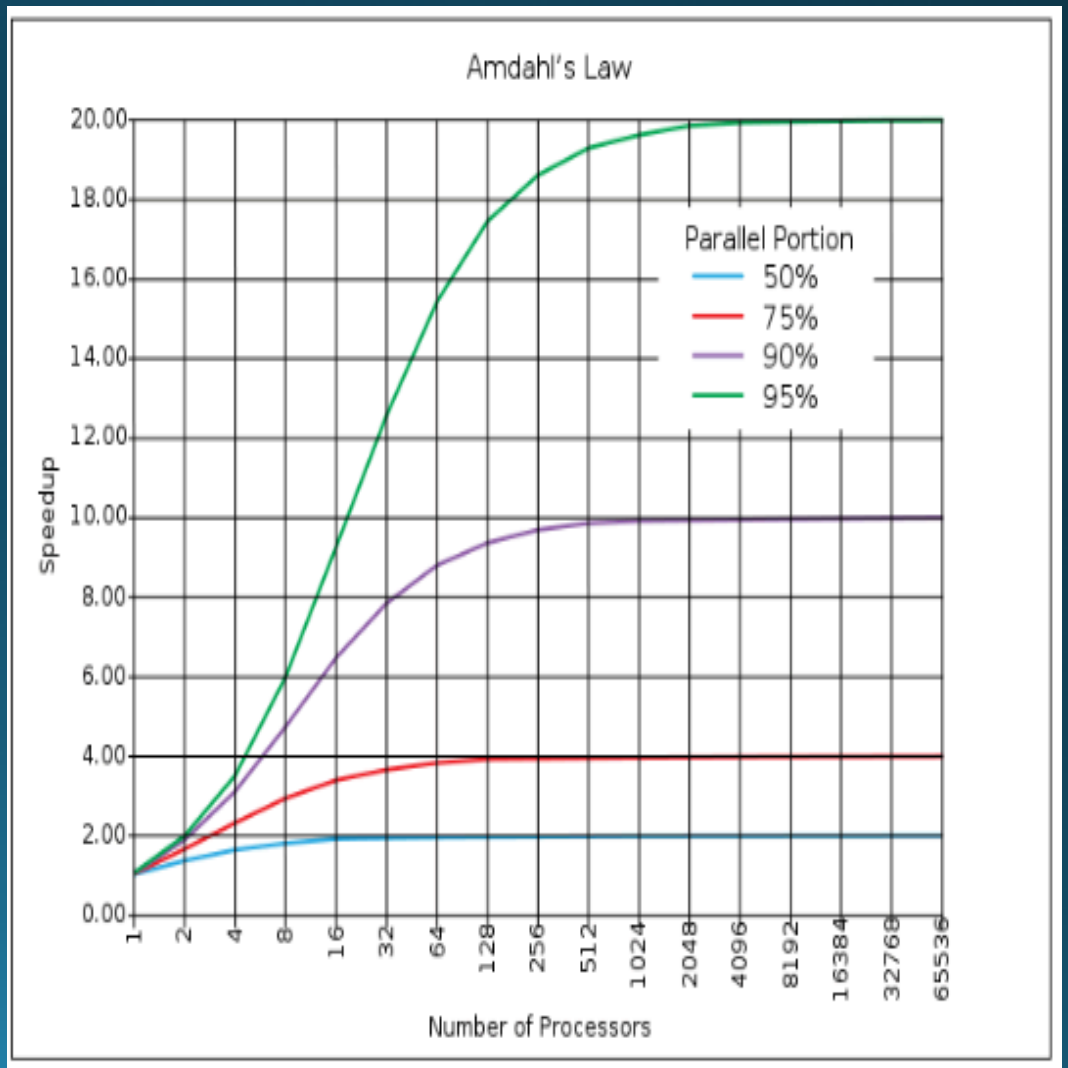
High Availability

Asynchronous

The new reality : Amdahl's law

The speed-up of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

For example if 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20 times, no matter how many processors are used



From Actors to Reactive Streams

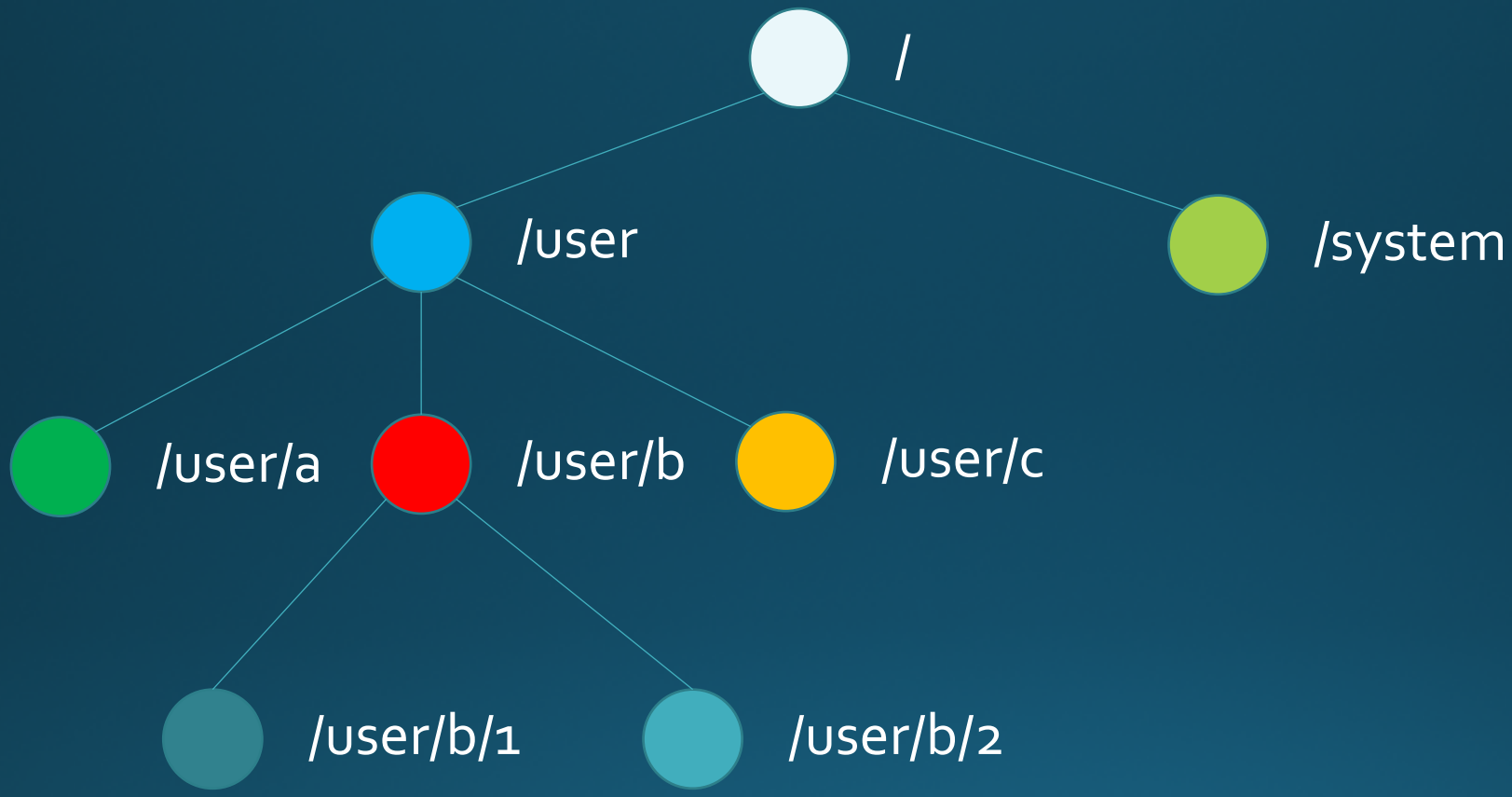
From Actors to Streams

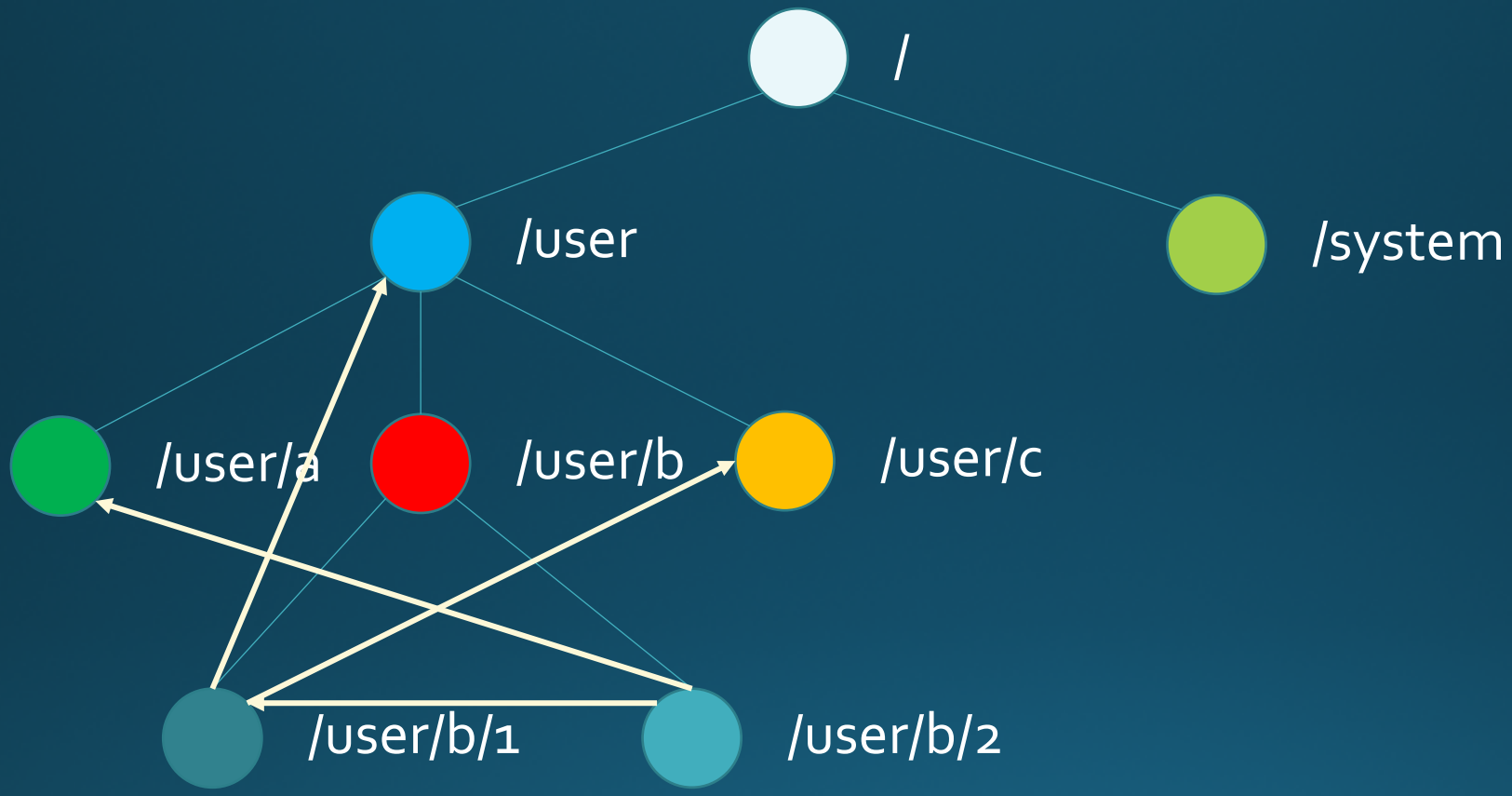
Claim: “actors do not compose”

“By default actors hard-code the receiver of any messages they send. If I create an actor A that sends a message to actor B, and you want to change the receiver to actor C you are basically out of luck.”

Noel Welsh (Underscore.io)

http://tinyurl.com/AkkaStreamsNdc2


/
/user
/system
/user/a
/user/b
/user/c
/user/b/1
/user/b/2


/
/user
/system
/user/a
/user/b
/user/c
/user/b/1
/user/b/2

Claim: “actors do not compose”

“Taken at face value (looking at the API) actors are functions which do not return anything: in Akka both tell() and Receive return Unit. This is why the statement arises that they cannot be composed, and what is meant is that they do not compose like normal functions.”

Roland Kuhn (Akka Tech Lead)

http://tinyurl.com/AkkaStreamsNdc3

Actor implementation class in C#

```
public class SampleActor : ReceiveActor
{
    public SampleActor()
    {
        Idle();
    }

    protected override void PreStart() { /* ... */ }

    private void Idle()
    {
        Receive<Job>(job => /* ... */);
    }

    private void Working()
    {
        Receive<Cancel>(job => /* ... */);
    }
}
```

What if we could describe the data flow process on a higher level?

Like in LINQ…

```
var results = db.Companies
                .Join(db.People,
                    c => c.CompanyID,
                    p => p.PersonID,
                    (c, p) => new { c, p })
                .Where(z => z.c.Created >= fromDate)
                .OrderByDescending(z => z.c.Created)
                .Select(z => z.p)
                .ToList();
```

… or pipeline of functions

```
HttpGet pageUrl
|> fun s -> Regex.Replace(s, "[^A-Za-z’]", " ")
|> fun s -> Regex.Split(s, " +")
|> Set.ofArray
|> Set.filter (fun word -> not (Spellcheck word))
|> Set.iter (fun word -> printfn "   %s" word)
```

How about that?

```
val in = Source(1 to 10)
val out = Sink.ignore
val bcast = builder.add(Broadcast[Int](2))
val merge = builder.add(Merge[Int](2))
val f1,f2,f3,f4 = Flow[Int].map(_ + 10)
```

```
source ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> sink
            bcast ~> f4 ~> merge ~>
```

Source: http://doc.akka.io/docs/akka/2.4/scala/stream/stream-graphs.html

Reactive Streams

Let's unlearn how to create individual actors

…and learn instead high level composition primitives

…that will internally create and connect instances of actors

Example in C#

```
var runnable =
    Source
        .From(Enumerable.Range(1, 1000))
        .Via(Flow.Create<int>().Select(x => x * 2)
        .To(Sink.ForEach<int>(x => Console.Write(x.ToString));
```

Graph DSL in C#

```
var graph = GraphDsl.Create(builder =>
{
    var bcast = builder.Add(new Broadcast<int>(2));
    var merge = builder.Add(new Merge<int, int>(2));
    var count = Flow.FromFunction(new Func<int, int>(x => 1));
    var sum = Flow.Create<int>().Sum((x, y) => x + y);
    builder.From(bcast.Out(0)).To(merge.In(0));
    builder.From(bcast.Out(1)).Via(count).Via(sum).To(merge.In(1));
    return new FlowShape<int, int>(bcast.In, merge.Out);
});
```

Reactive Streams

"Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

http://www.reactive-streams.org

Participants in Reactive Stream initiative

Reactive Streams

"Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

http://www.reactive-streams.org

asynchronous

Is all about Asynchronicty !

Reactive Streams

"Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

http://www.reactive-streams.org

asynchronous

stream processing

Reactive Streams

"Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols

http://www.reactive-streams.org

asynchronous

stream processing

non-blocking back pressure

What do Reactive Streams solve?

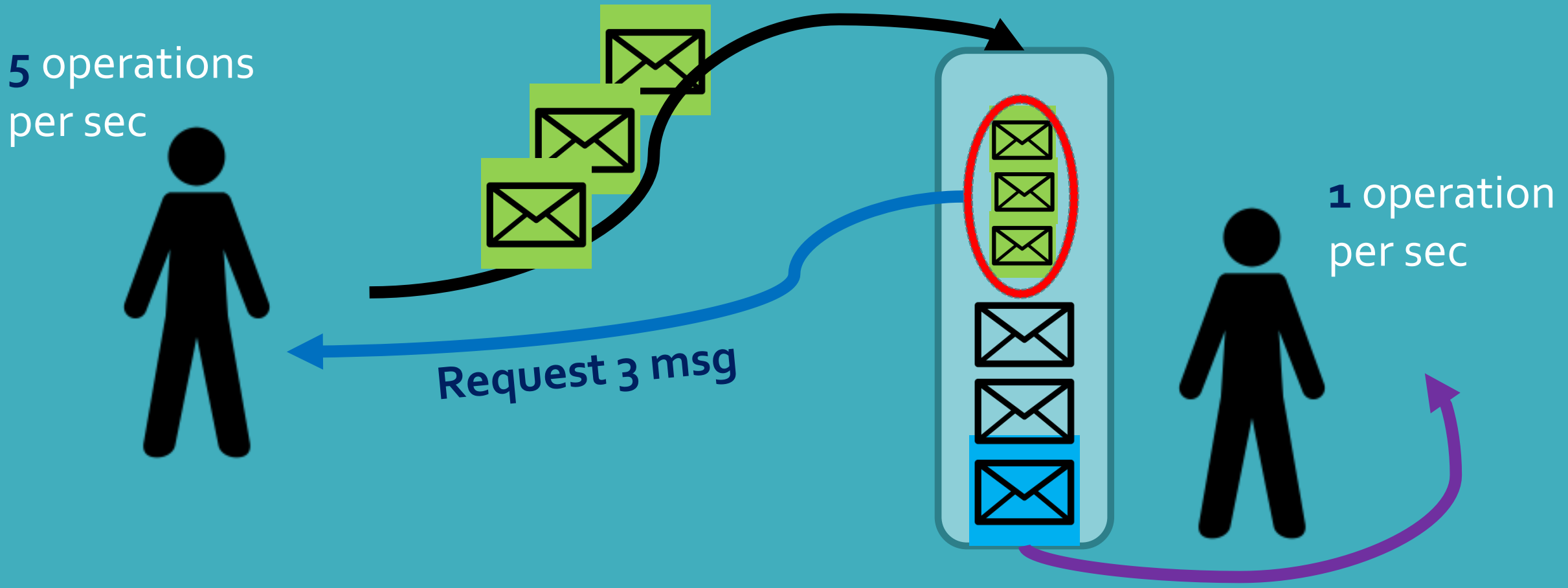
Getting **Data** across an

Asynchronous Boundary

without running **Out-Of-Memory**

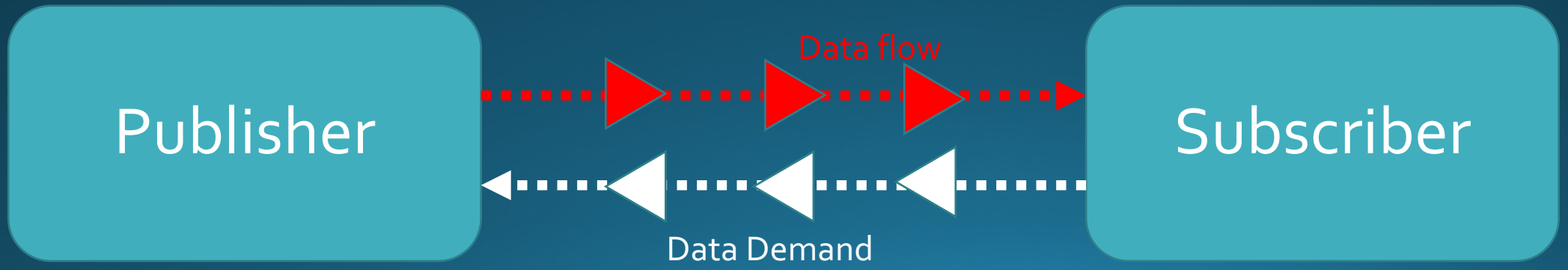
Dynamic Push-Pull

Pull based backpressure

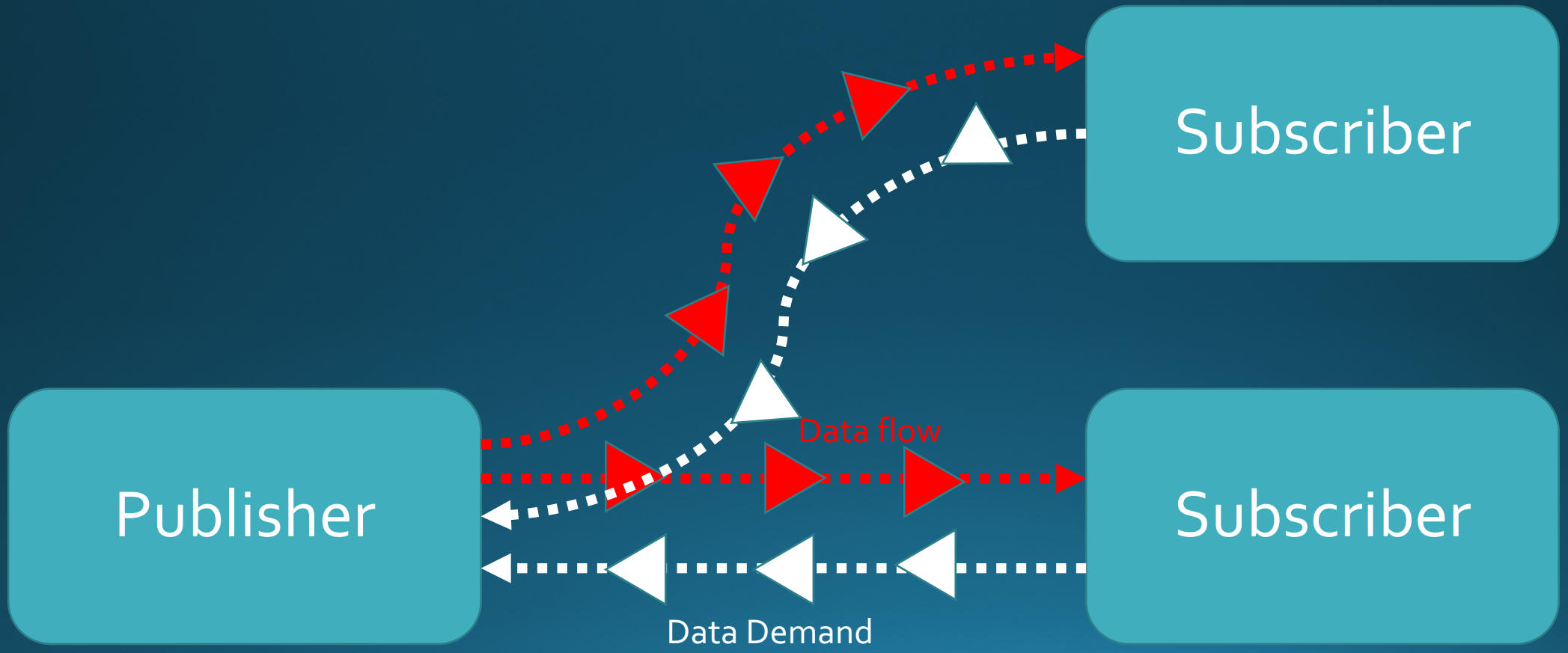


Push and Pull

- data items flow downstream
- data demand flows upstream
- data items flow only when there is demand
  - recipient is in control of incoming data rate
  - data in flight is bounded by signaled demand
  - **"push" behavior when consumer is faster**
  - **"pull" behavior when producer is faster**



Splitting and merging the data flow



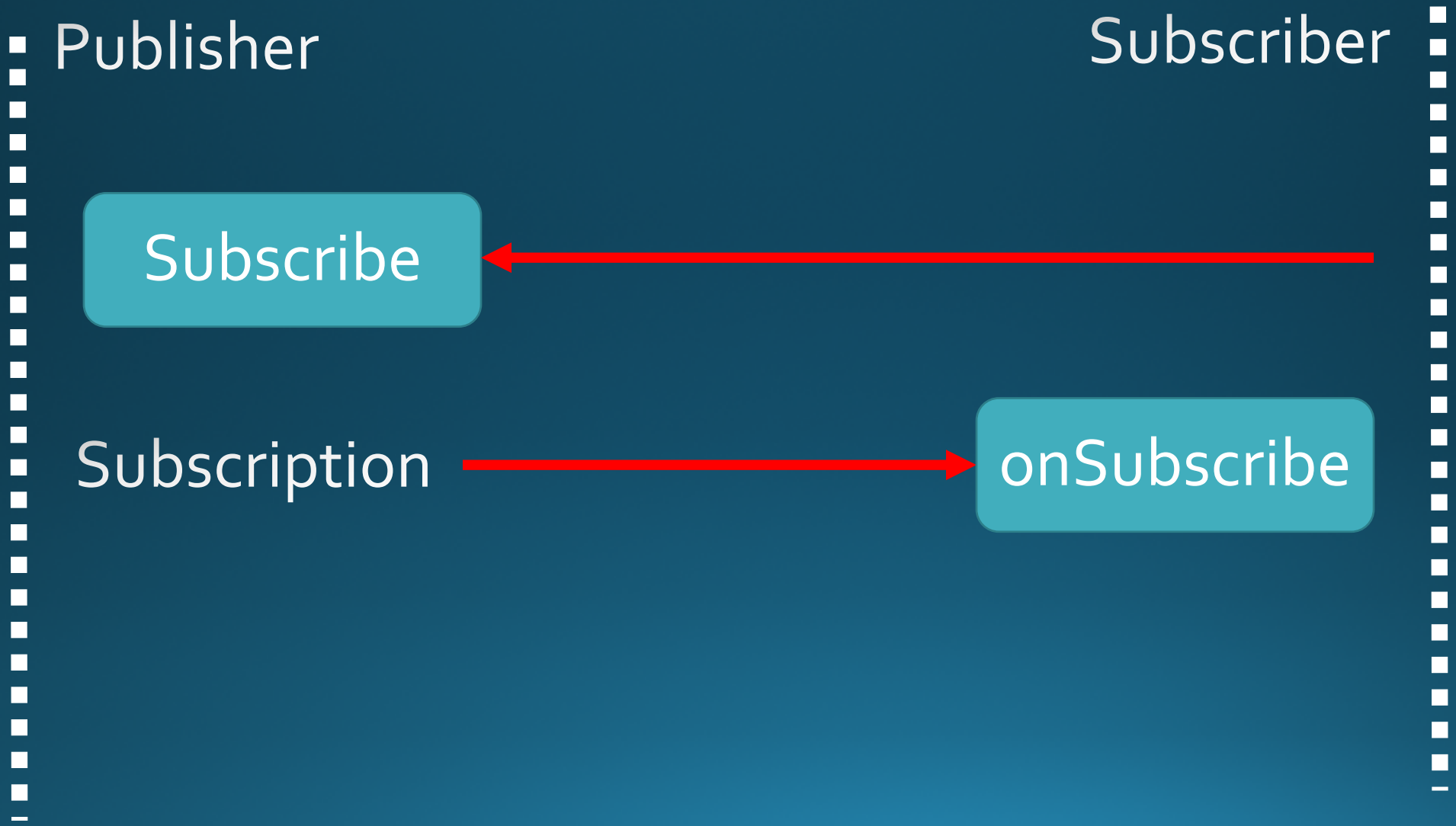
You might consider Akka Streams if…

- You have used Akka but want to save yourself from details of actor definition and management
- You are familiar with Reactive Streams and looking for ready-for-use implementations of its specification
- You find Akka Streams stages building blocks suitable to compose your process workflows
- You want to take advantage of Akka Streams backpressure to have a fine grained control over throughput of your process workflow stages

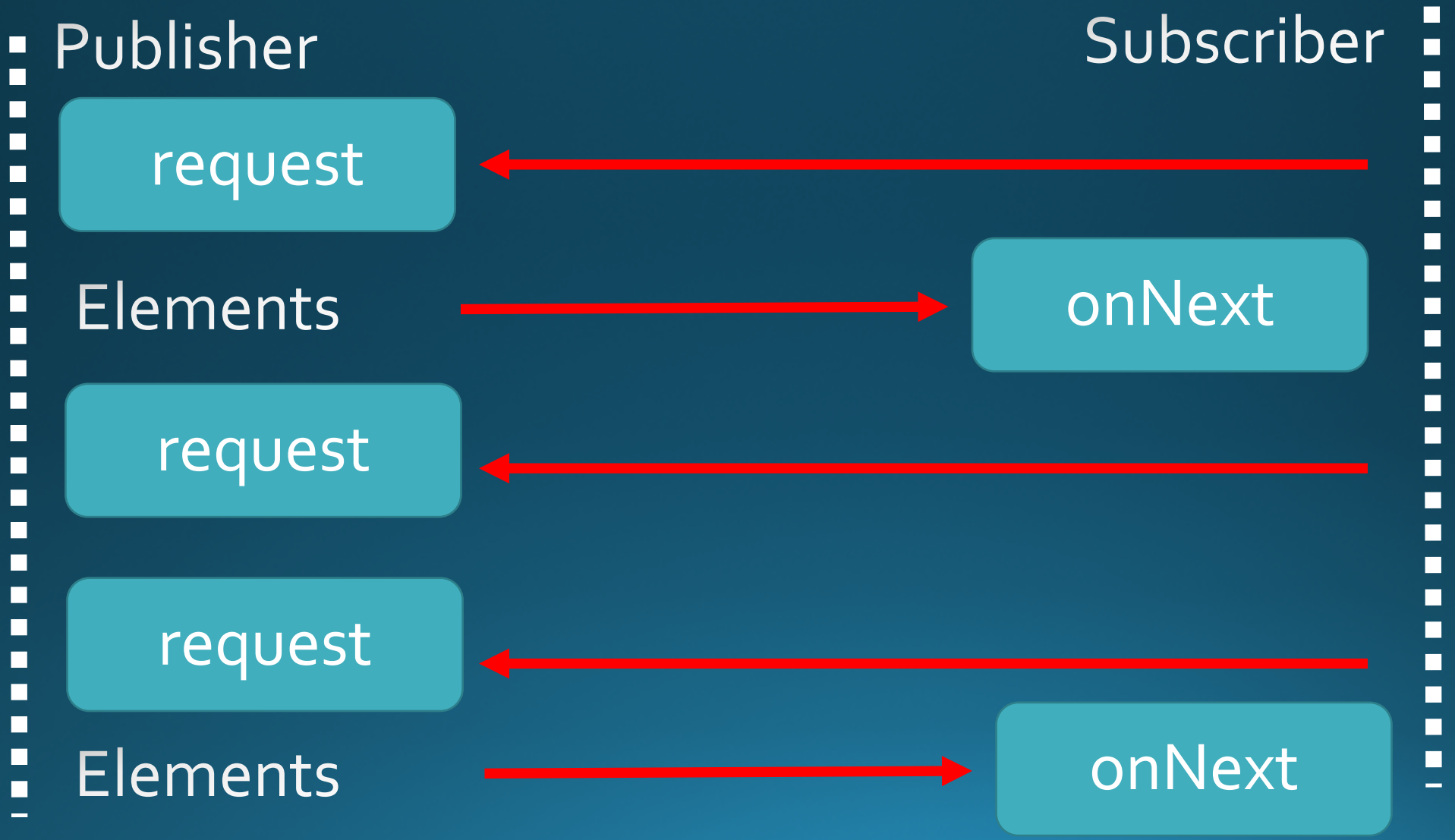
Reactive Stream interfaces

```
public interface IPublisher<out T>
{
    void Subscribe(ISubscriber<T> subscriber);
}

public interface ISubscriber<in T>
{
    void OnSubscribe(ISubscripion subscription);
    void OnNext(T element);
    void OnError(Exception cause);
    void OnComplete();
}

public interface ISubscription
{
    void Request(long n);
    void Cancel();
}
```

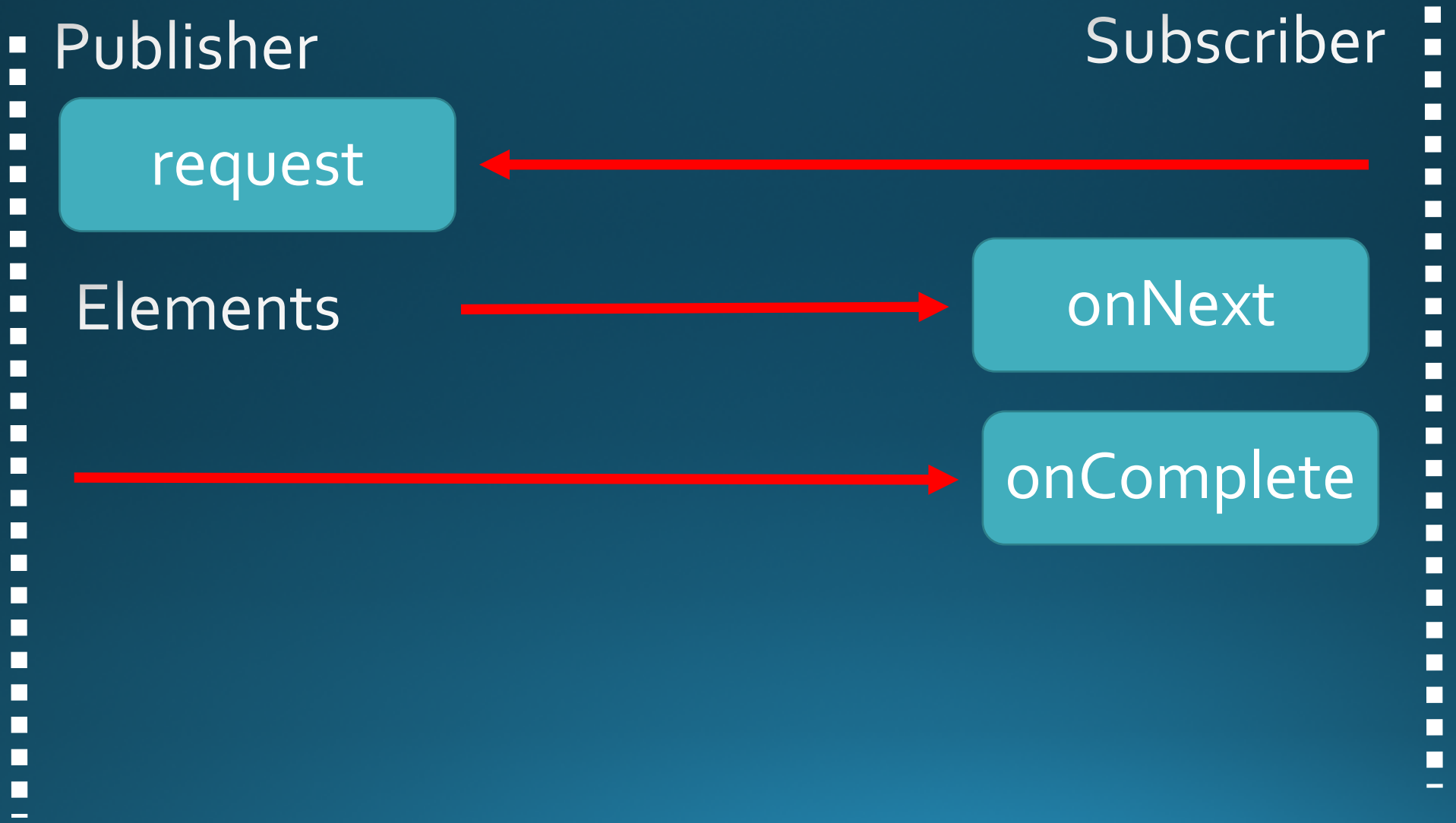
Connection


Publisher
Subscriber
Subscribe
Subscription
onSubscribe

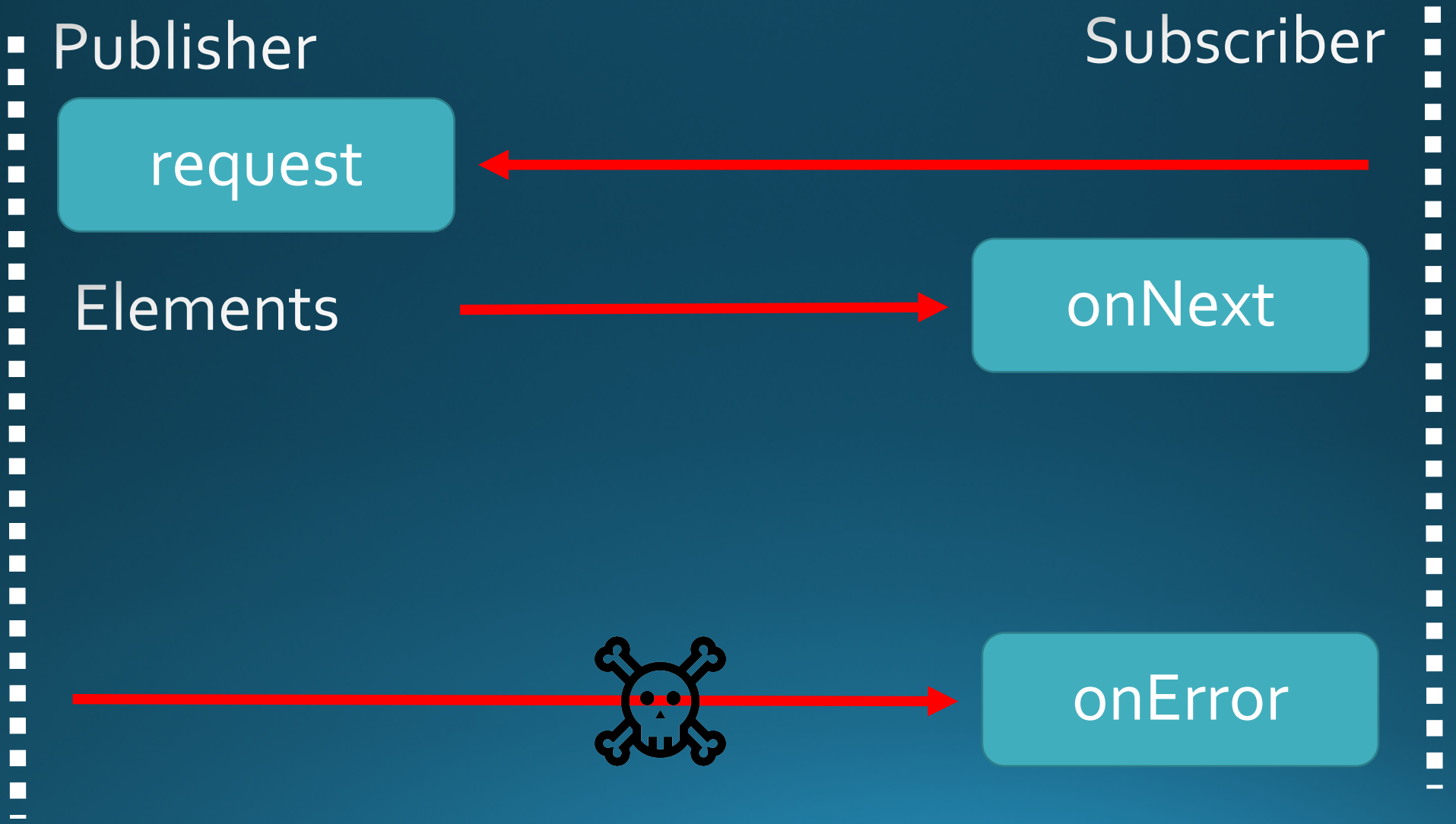
Flow

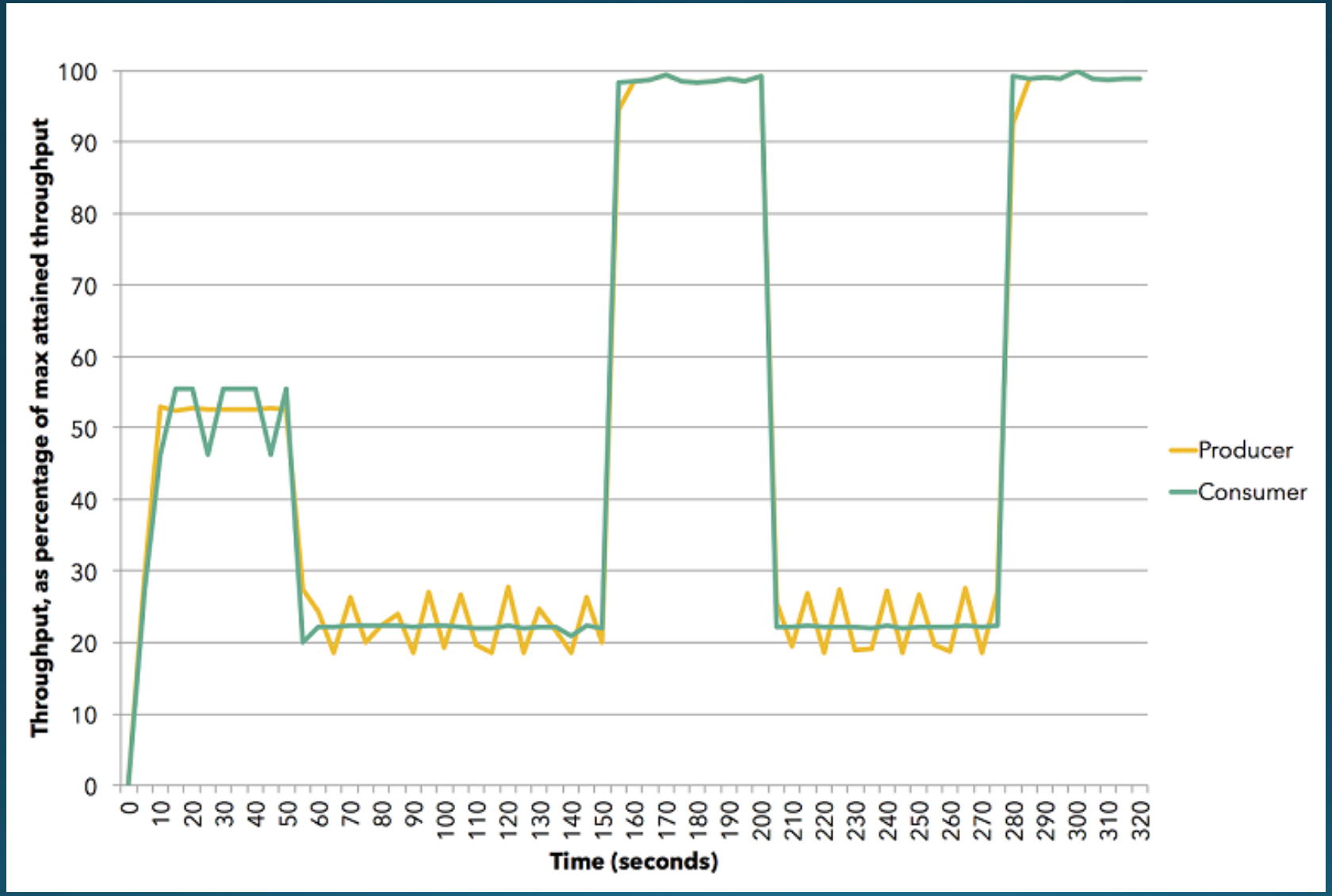

Publisher
Subscriber
request
Elements
onNext
request
request
Elements
onNext

Complete


Publisher
Subscriber
request
Elements
onNext
onComplete

Failure

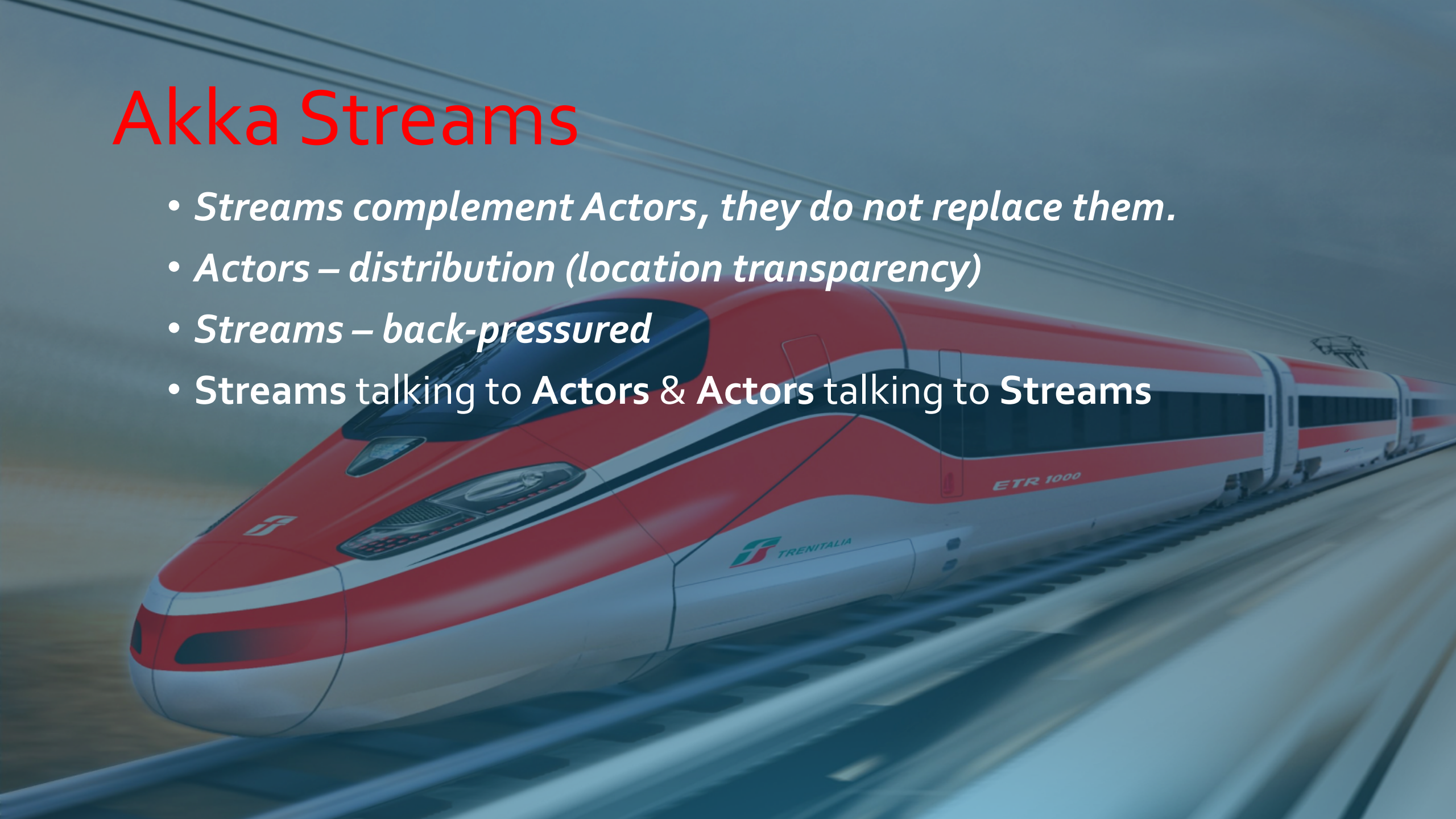




Source: https://data-artisans.com/blog/how-flink-handles-backpressure (Apache Flink)

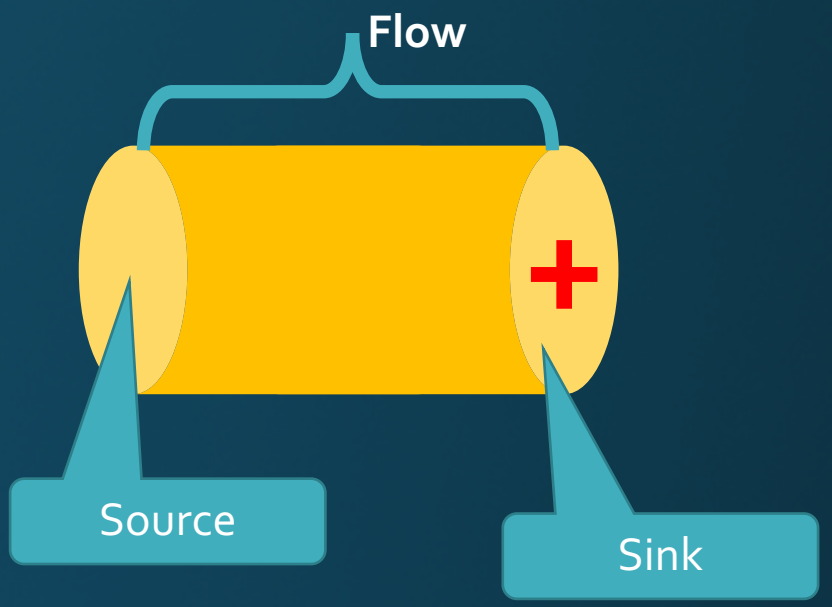
Akka Streams

- ***Streams complement Actors, they do not replace them.***
- ***Actors – distribution (location transparency)***
- ***Streams – back-pressured***
- **Streams** talking to **Actors** & **Actors** talking to **Streams**



Push-Pull bidirectional communication

Akka Stream Lexicon



Source

This is where the pipeline starts. A Source takes data from input and has a single output to be written into.

Sink

The pipeline ends here. A Sink has a single input to be written into.

Flow

This is the basic abstraction where transformation of data takes place. A Flow has one input and one output

RunnableGraph

The entire topology of (Source -> Flow* -> Sink) a graph in Akka Streams.

Materializer

A materializer defines how the transformations are converted into asynchronous processes.

Source stages in Akka.NET Streams

- FromEnumerator
- From
- Single
- Repeat
- Cycle
- Tick
- FromTask
- Unfold
- UnfoldAsync
- Empty
- Maybe
- Failed

- ActorPublisher
- ActorRef
- Combine
- UnfoldResource
- Queue
- AsSubscriber
- FromPublisher
- ZipN
- ZipWithN
- FromInputStream
- AsOutputStream
- FromFile

Sink stages in Akka.NET Streams

- First
- FirstOrDefault
- Last
- LastOrDefault
- Ignore
- Cancelled
- Seq
- Foreach
- ForeachParallel
- OnComplete
- Aggregate
- Sum

- Combine
- ActorRef
- ActorRefWithAck
- ActorSubscriber
- AsPublisher
- FromSubscriber
- FromOutputStream
- AsInputStream
- ToFile

Flow stages in Akka.NET Streams

- Select
- SelectMany
- StatefulSelectMany
- Where
- Collect
- Grouped
- Sliding
- Scan
- Aggregate
- Skip
- SkipWhile
- Take
- TakeWhile
- Recover
- RecoverWith
- Detach
- Throttle
- SelectAsync
- SelectAsyncUnordered
- TakeWithin
- SkipWithin
- GroupedWithin
- InitialDelay
- Delay
- Conflate
- ConflateWithSeed
- Batch
- BatchWeighted
- Expand
- Buffer
- PrefixAndTail
- GroupBy
- SplitWhen
- SplitAfter
- ConcatMany
- MergeMany
- InitialTimeout
- CompletionTimeout
- IdleTimeout
- BackpressureTimeout
- KeepAlive
- InitialDelay
- Merge
- MergeSorted
- Zip
- ZipWith
- ZipWithIndex
- Concat
- Prepend
- OrElse
- Interleave
- Unzip
- UnzipWith
- Broadcast
- Balance
- WatchTermination
- Monitor
- InitialTimeout

Stream materialization

- Stream declaration prepares a blueprint (an execution plan)
- Execution begins with stream materialization
- Only runnable graphs can be materialized
- Multiple stream stages can be allocated to a single actor
- Most of flow stages preserve input order of elements

Compare with LINQ statements

```
var results = db.Companies
                .Join(db.People,
                    c => c.CompanyID,
                    p => p.PersonID,
                    (c, p) => new { c, p })
                .Where(z => z.c.Created >= fromDate)
                .OrderByDescending(z => z.c.Created)
                .Select(z => z.p)
                .ToList();
```

Materializing a stream

```
var runnable =
    Source
        .From(Enumerable.Range(1, 1000))
        .Via(Flow.Create<int>().Select(x => x * 2)
        .To(Sink.ForEach<int>(x => Console.Write(x.ToString));
```

Materializing a stream

```
var runnable =
    Source
        .From(Enumerable.Range(1, 1000))
        .Via(Flow.Create<int>().Select(x => x * 2)
        .To(Sink.ForEach<int>(x => Console.Write(x.ToString));

var system = ActorSystem.Create("MyActorSystem");
using (var materializer = ActorMaterializer.Create(system))
{
    await runnable.Run(materializer);
}
```

Case Study
Streaming Event Journal

PLANET OS DASHBOARD

TOTAL

286 MW
Active Power

6.2 M/S SSW 256°
Winds

9.1 °C
Air Temp

14.6 °C
Water Temp

FR 03 06 09 12 15 18 21 SA 03 06 09 12

12
8
4
0

WINDS [M/S] 6.2 M/S FROM THE SSW 256°

3
2
1
0

WAVE HEIGHT [M] 2.3 M

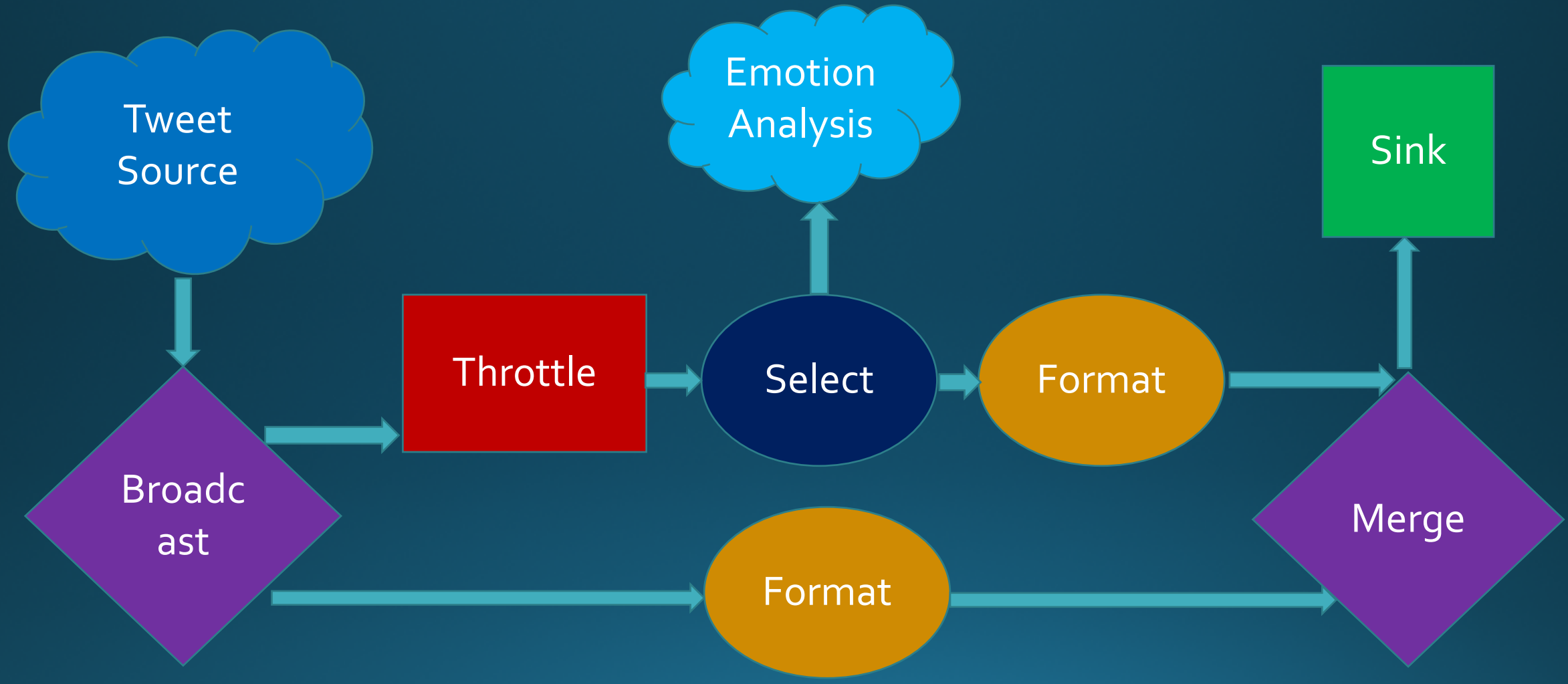
12
8
4
0

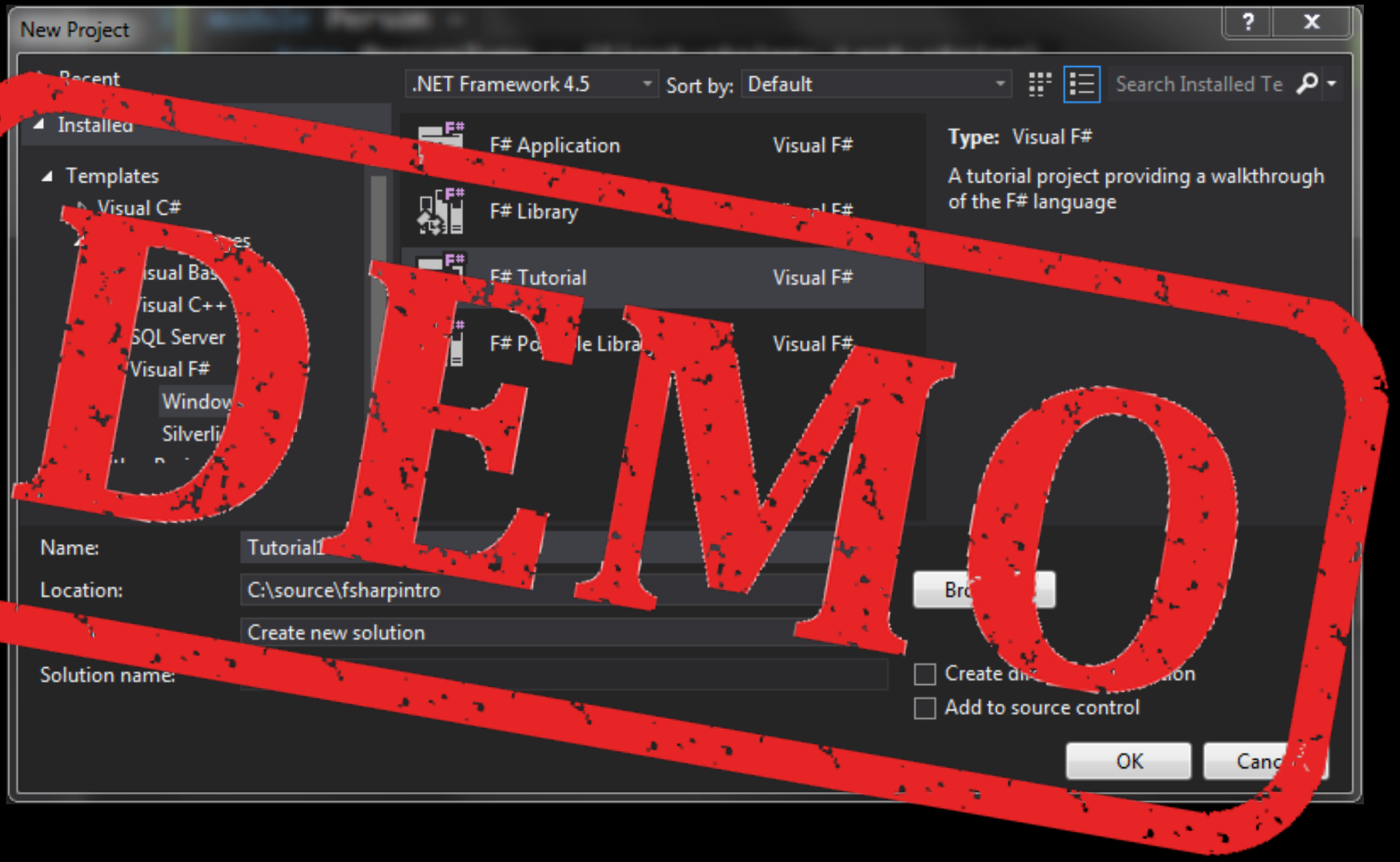
TIDE [M] 7.8 M

Case Study
Reactive Tweets

Reactive Tweets

- Both Akka (Scala/JVM) and Akka.NET contain samples showing a stream of Twitter tweets
- Akka.NET sample is based on Tweetinvi (https://github.com/linvi/tweetinvi)
- Tweetinvi expose several streams of Twitter data that don't support Reactive Streams
  - IFilteredStream
  - ISampleStream
  - ITrackedStream
  - ITweetStream
  - IUserStream
- Reactive Tweets use Source.ActorRef built-in primitive to emit elements from Tweetinvi streams to Akka Stream source


Tweet Source
Emotion Analysis
Sink
Throttle
Select
Format
Broadc ast
Format
Merge


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO


That's all Folks!