Reactive Programming

Event-Based

BY RICCARDO TERRELL - @RIKACE

Objectives

- Stream processing motivation
- Real time stream analysis
- Simplify complex logic with reactive programming
- Composing events
- Back Pressure

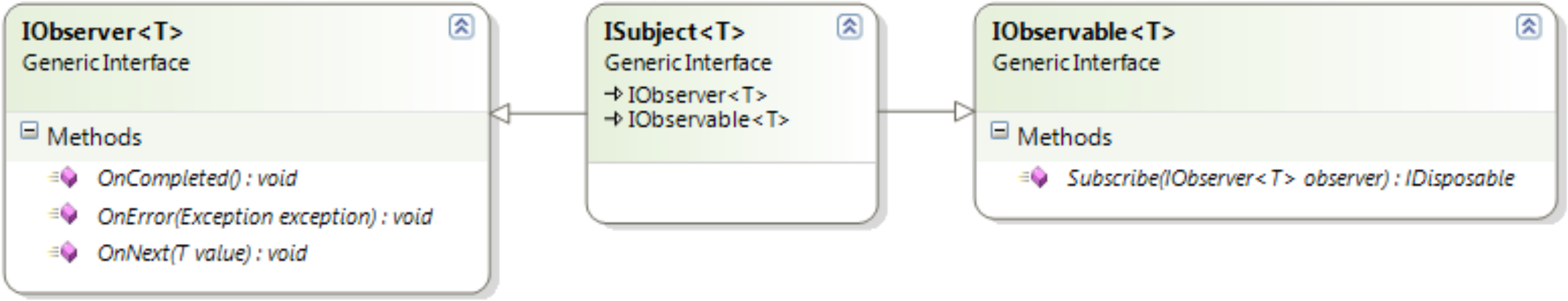
Stream processing motivation

- **React** to **high rate events** (near real time)
- **Compose events** from **different sources**
- **Simplify** complex logic
- Lots of built in **combinators** (operators)
- **Pipeline** for **combinators** over **event stream**.

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



IObserver & IObservable

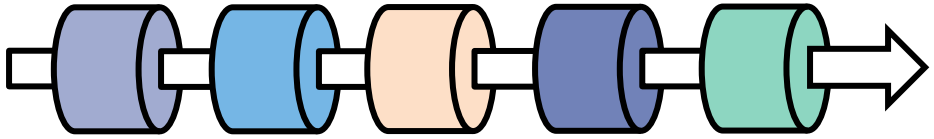
```
type IObserver<'a> = interface
    abstract OnCompleted : unit -> unit
    abstract OnError : exn -> unit
    abstract OnNext : 'a -> unit
end

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

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**



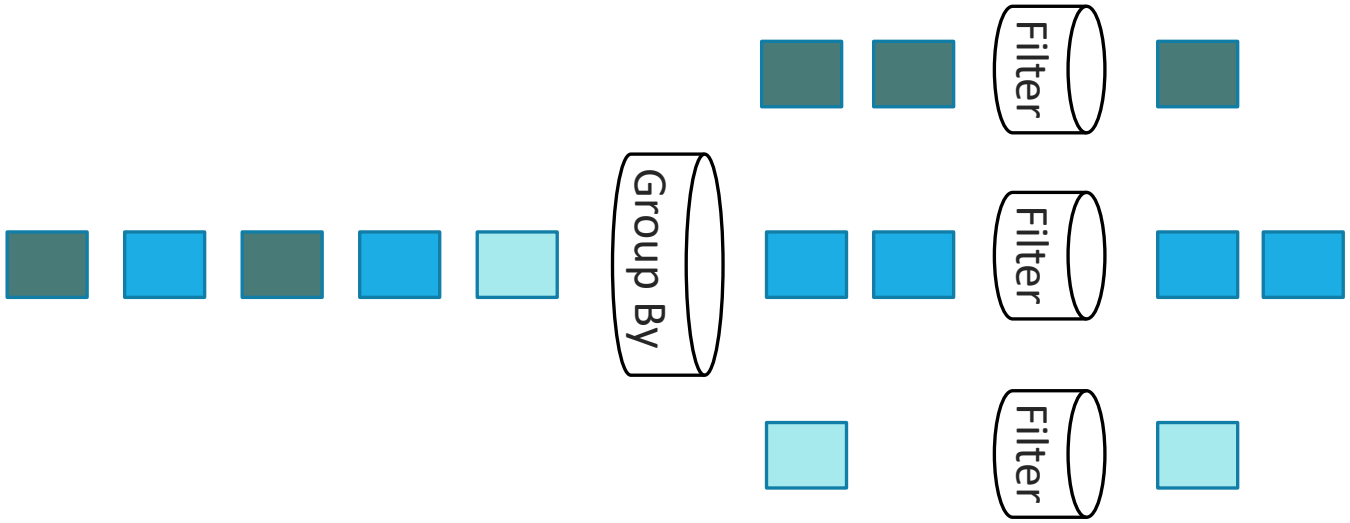
Rx = Async Data Stream Composition

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**



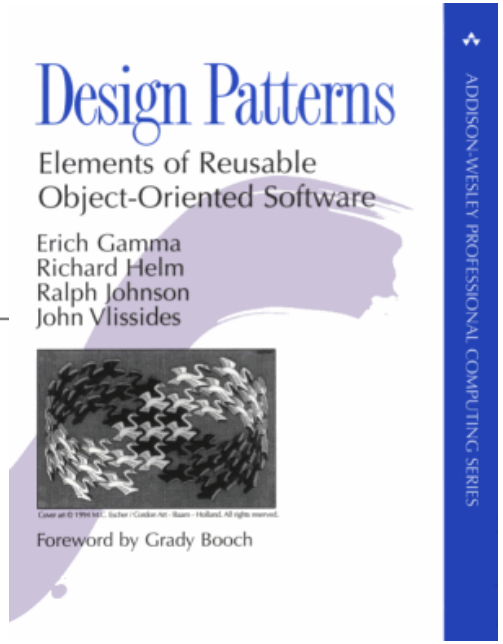
Rx = Async Data Stream Composition

Rx = LINQ to Async Data Stream

Fundamental Abstractions

Adapting the observer pattern

- Ensuring duality with the enumerator pattern
- More compositional approach



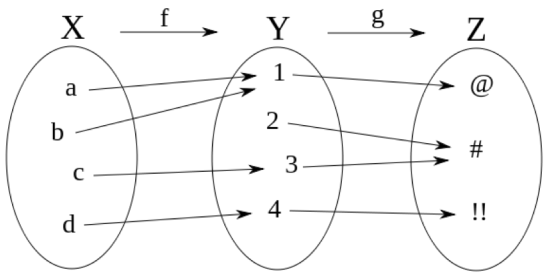
"Gang of four" book
Addison-Wesley

```
interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

interface IObserver<in T>
{
    void OnNext(T value);
    void OnError(Exception error);
    void OnCompleted();
}
```

Notification grammar

```
OnNext* (OnError | OnCompleted)?
```

Highly Compositional



Function composition

LINQ-style query operators over IObservable<T>

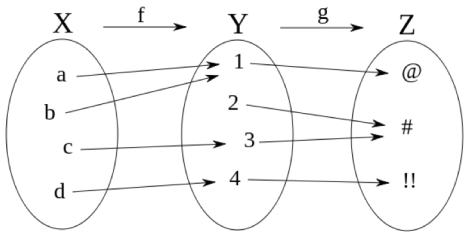
- Composition of 0-N input sequences

```
static class Observable
{
    static IObservable<T> Where<T>(this IObservable<T> source, Func<T, bool> f);
    static IObservable<R> Select<T, R>(this IObservable<T> source, Func<T, R> p);
    …
}
```

- Composition of disposable subscriptions and scheduler resources

```
interface IScheduler
{
    IDisposable Schedule(Action work);
    …
}
```

Highly Compositional



Function composition

Building a binary Merge operator

First-class: can build extension methods

```
public static class Observable
{
    public static IObservable<T> Merge<T>(this IObservable<T> xs, IObservable<T> ys)
    {
        return Create<T>(observer =>
        {
            var gate = new object();

            return new CompositeDisposable
            {
                xs.Subscribe(x => { lock (gate) { observer.OnNext(x); } }, …),
                ys.Subscribe(y => { lock (gate) { observer.OnNext(y); } }, …),
            };
        });
    }
}
```

Composition of resource management

The Role of Schedulers

Pure architectural layering of the system

- Logical query operators (~ relational engine)
- Physical schedulers (~ operating system)

```
public static IObservable<T> Return<T>(T value, IScheduler scheduler)
{
    return Create<T>(obs => scheduler.Schedule(() => { obs.OnNext(value);
                                                       obs.OnCompleted(); }));
}
```

Abstract over sources of asynchrony and time

- Threads, thread pools, tasks, message loops
- DateTimeOffset.UtcNow, timers

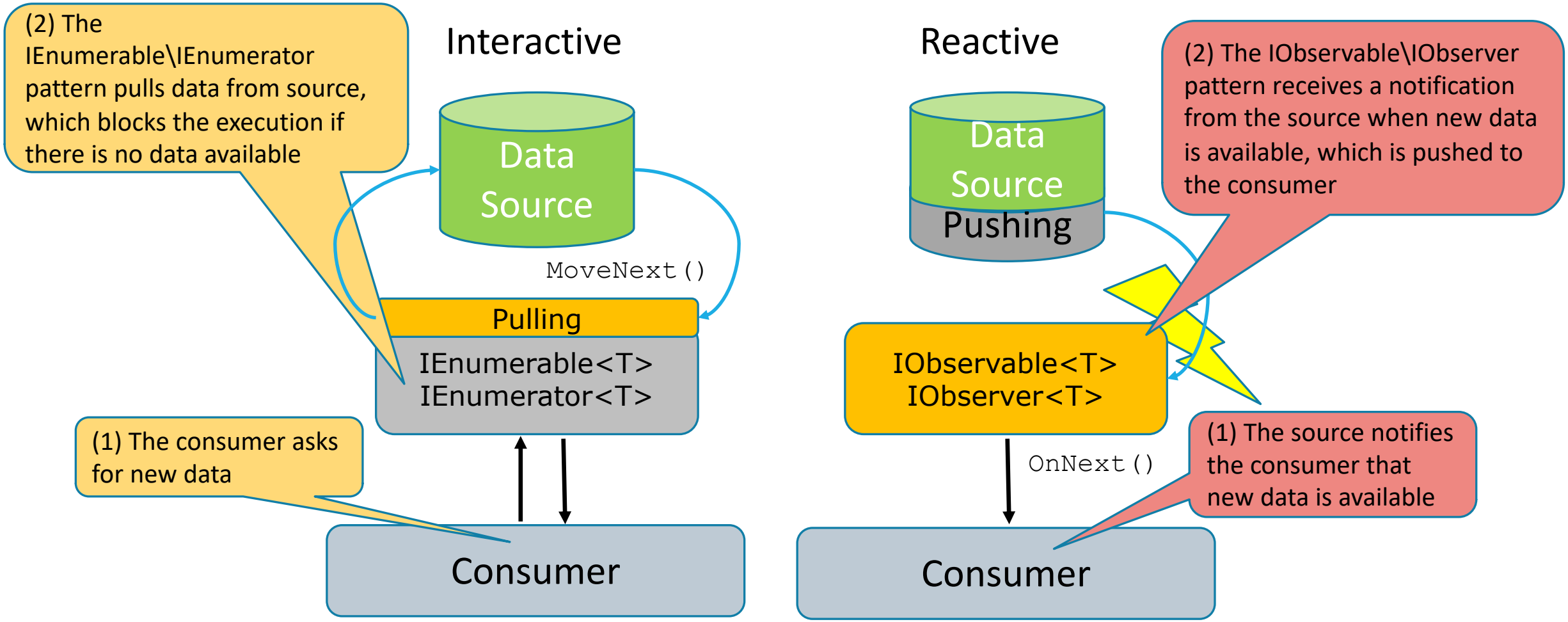
One versus Many

When Task and Task<T> were born…

- Single-value specializations
- Await-ability of sequences (for aggregates)

	Synchronous	Asynchronous
One	`Func<T> f` `var x = wait f();`	`Task<T> t` `var x = await t;`

Many	`IEnumerable<T> xs`
	`foreach (var x in xs) {`
	`  f(x);`
	`}`
		`IObservable<T> xs`
		`xs.Subscribe(x => {`
		`  f(x);`
		`});`


(2) The IEnumerable\IEnumerator pattern pulls data from source, which blocks the execution if there is no data available
Interactive
Data Source
MoveNext()
Pulling
IEnumerable<T>
IEnumerator<T>
(1) The consumer asks for new data
Consumer
Reactive
Data Source
Pushing
(2) The IObservable\IObserver pattern receives a notification from the source when new data is available, which is pushed to the consumer
IObservable<T>
IObserver<T>
OnNext()
(1) The source notifies the consumer that new data is available
Consumer

IObservable the dual of IEnumerable

Category theory to the rescue (Bierman, Meijer)

- Observable/observer (push) is dual to enumerable/enumerator (pull)
- Cross-influence of both domains

```fsharp
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
```

```fsharp
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
```

Stream processing basics - Event

Not first class in C#

Hard to compose

C#

```
button.Click += OnClick;

private void OnClick(object sender, RoutedEventArgs e)
{
    …
}
```

Stream processing basics – F# Event

First class in F#

Can have basic composition

```
button1.Click
    |> Event.merge button2.Click
    |> Event.add (fun a -> MessageBox.Show "Hii"
                           |> ignore)
```

F#

Stream processing basics – IObservable/IObserver

C#

```
var click1 = Observable
    .FromEventPattern(button1, nameof(button1.Click));

var click2 = Observable
    .FromEventPattern(button2, nameof(button2.Click));

Observable.Merge(click1, click2).Subscribe(OnClick);
```

Stream processing basics – IObservable/IObserver

F#

```
let button1 = Button()
let button2 = Button()

button1.Click
|> Observable.merge button2.Click
|> Observable.add (fun a -> MessageBox.Show "Hii“
                                |> ignore)
```

Stream processing basics – Combinators

F#

```
let observable = Observable
    .Interval(TimeSpan.FromSeconds(1.0))

observable.Subscribe (printfn "%d")
```

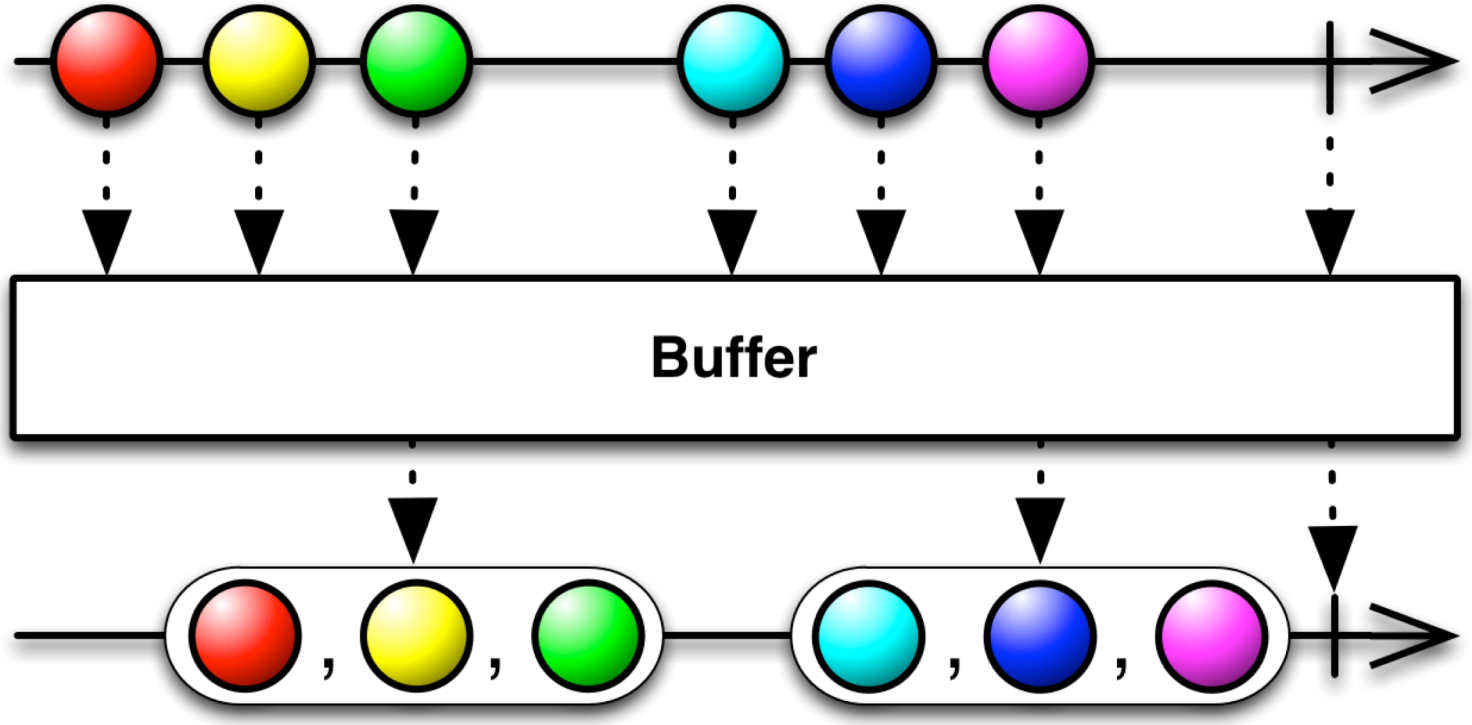
Stream processing basics – Combinators

F#

```
let observable = Observable
    .Interval(TimeSpan.FromSeconds(1.0))
    .GroupBy(fun x -> x % 3L)

let printKeyValue key value =
    printfn "%s %d" (String('*', key)) value


observable.Subscribe(fun obs ->
    obs.Subscribe (printKeyValue (int obs.Key)) |> ignore)
```

Stream processing basics – Combinators

Buffer — periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time



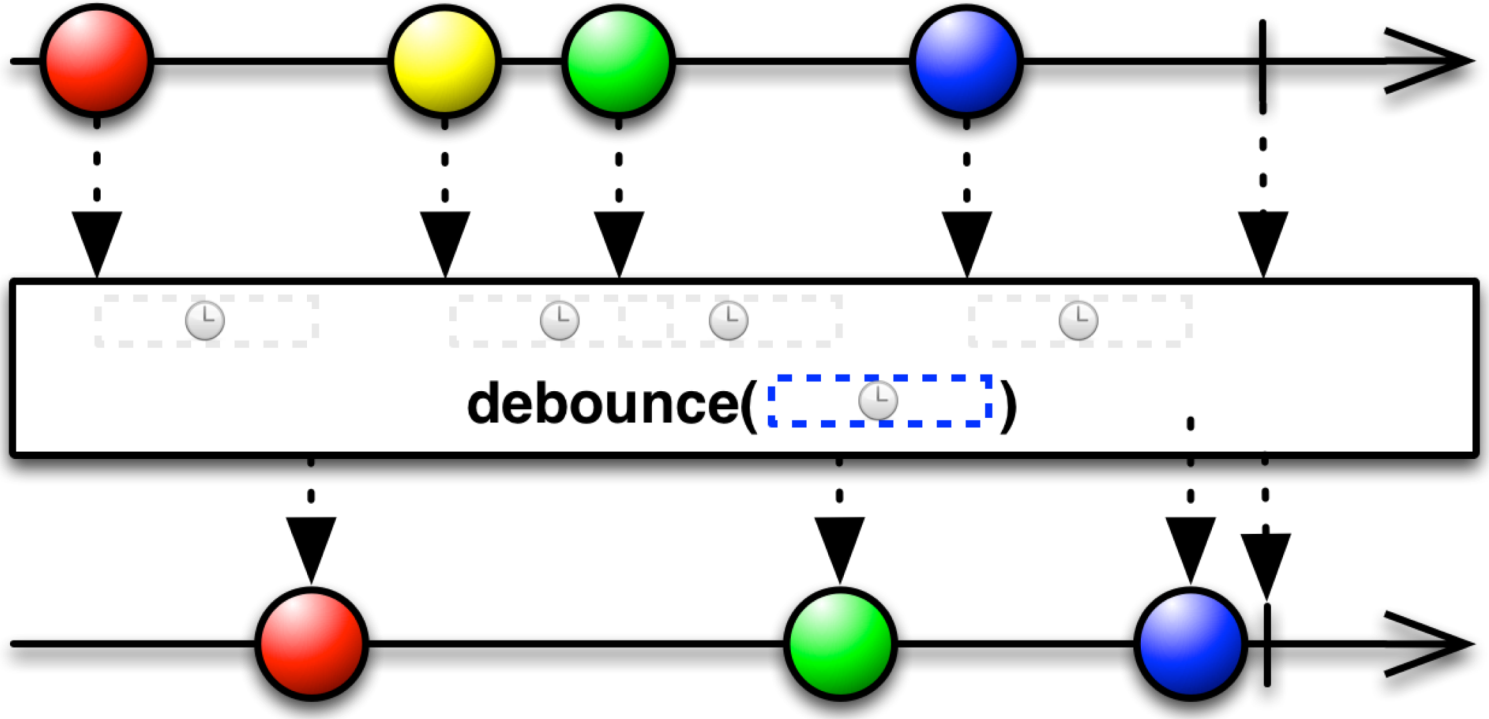
Stream processing basics – Combinators

F#

```
let observable = Observable
    .Interval(TimeSpan.FromSeconds(1.0))
    .Observable.Buffer(3)

observable.Subscribe (printfn "%A")
```

Stream processing basics – Combinators

Throttle — only emit an item from an Observable if a particular timespan has passed without it emitting another item



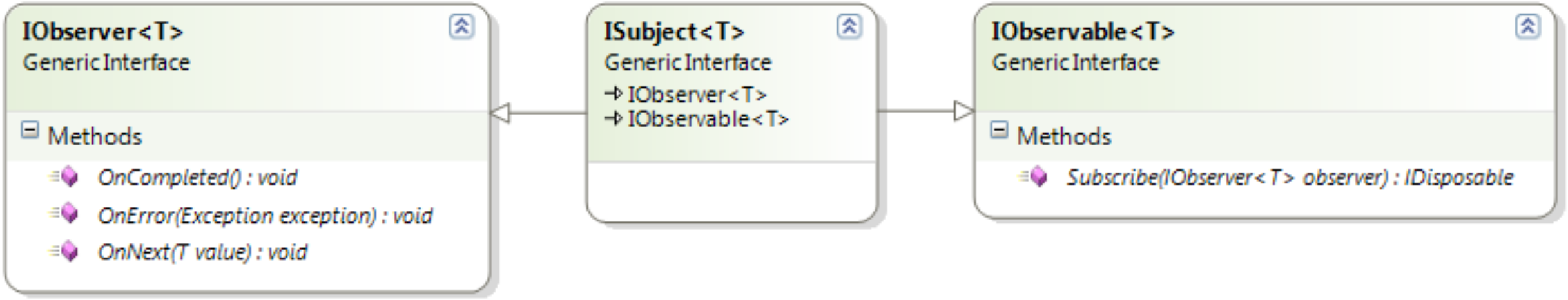
Stream processing basics – Combinators

F#

```
let observable = Observable
    .FromEventPattern(txt, "TextChanged")
    .Throttle(TimeSpan.FromSeconds(1.0))

observable
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe (fun text -> console.Text <- console.Text +
                            Environment.NewLine + txt.Text )
```

What are Reactive Extensions ?

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

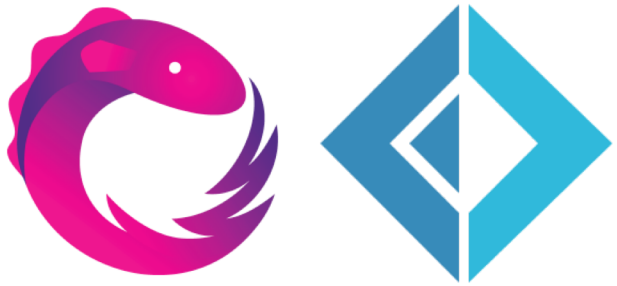


publisher *both* *subscriber*

Convert Task to Observable

```
IObservable<Unit> observable1 = Task.Run(() => Console.WriteLine("Working")).ToObservable();

IObservable<string> observable2 = Task<string>.Run(() => "Working").ToObservable();


Task task1 = observable1.ToTask();

Task<string> task2 = observable2.ToTask();
```

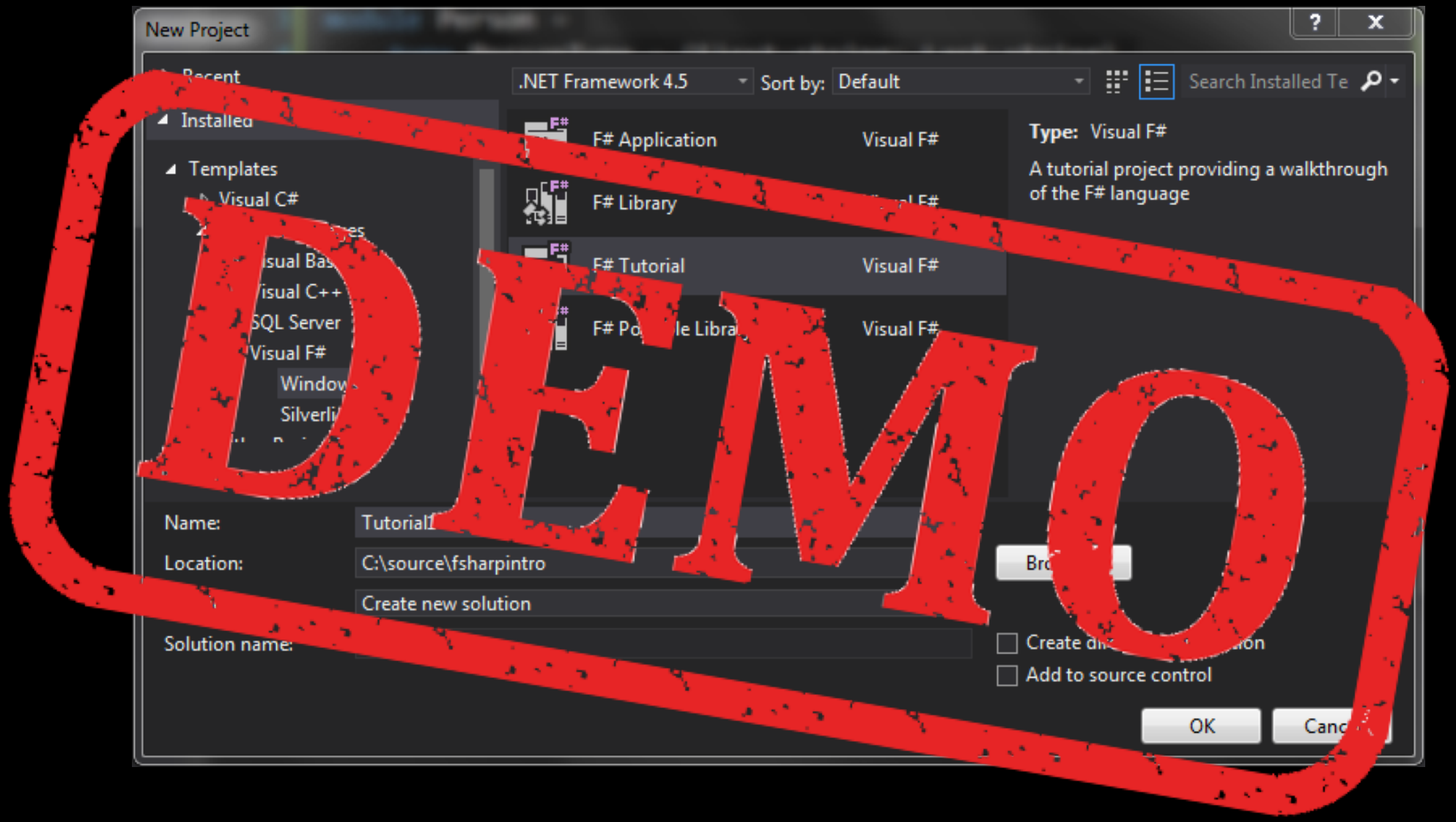
F# & .Net RX Api



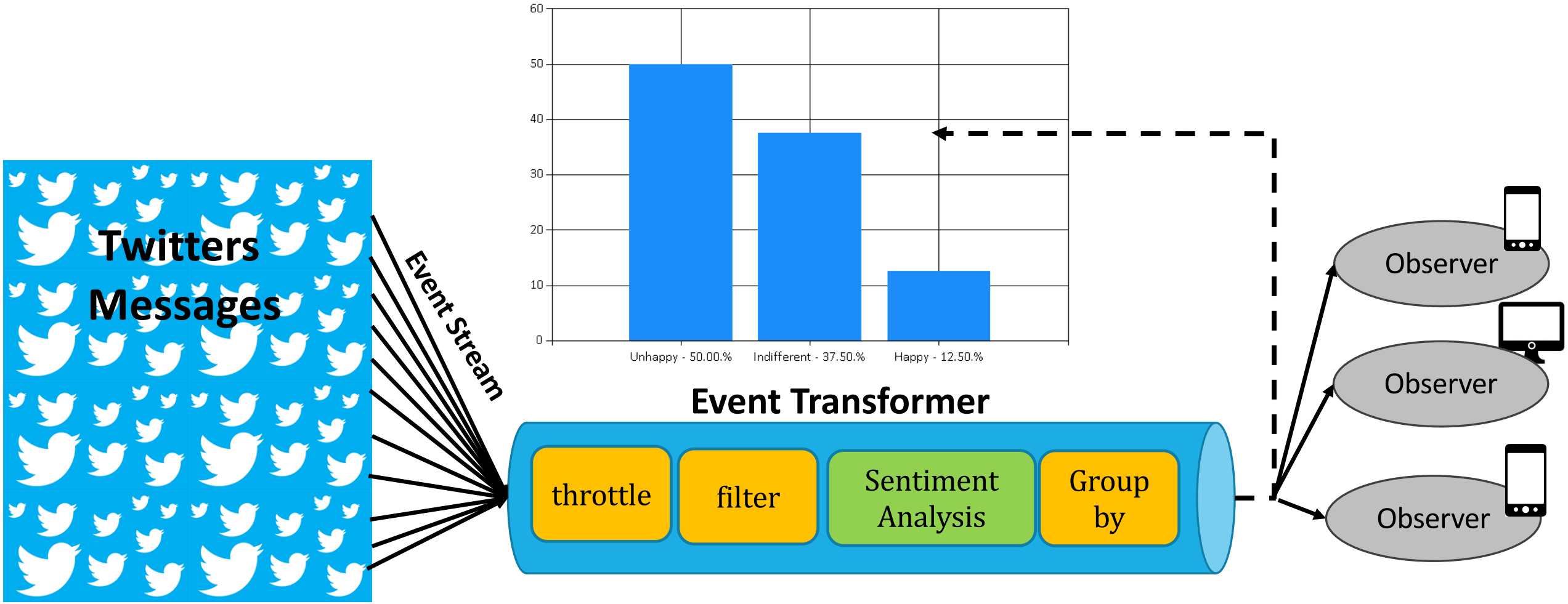
F# Observable
add
choose
filter
merge
pairwise
partition
scan
subscribe

.NET RX Api								
All	Amb	And	Any	Avarage	Case	Cast	Catch	Chunhify
Combine	Count	Defer	DefaultIfEmpty	Delay	Distinct	DistinctUntillChanged	ElemntAt	ElementAtOrDefault
Finally	First	FirstOrDefault	From	GroupBy	GroupUntill	GroupJoin	If	IgnoreElement
Last	Latest	LastOrDefault	Max	Merge	MaxBy	Min	MinBy	MostRecent
Multicast	Range	Repeat	Retry	RefCount	Reply	Return	Never	Sum
Switch	Select	SelectMany	Scan	Single	Skip	SkipLast	SkipUntill	Then
Take	TakeLast	TakeUntill	TakeWhile	Throttle	Throw	Zip	While	When
Window	When	While	Wait	Single	Ignore	SingleOrDefault		

http://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-Inside-the-NET-Reactive-Framework-Rx

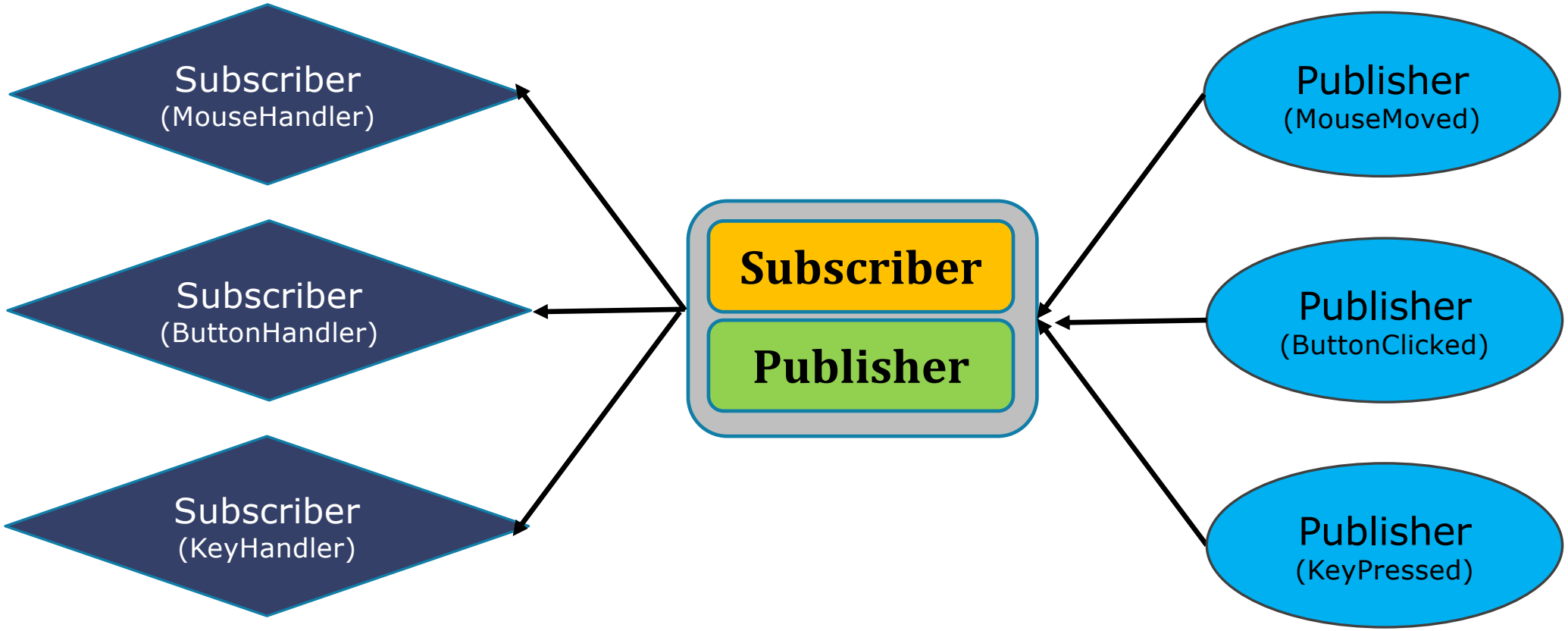

New Project
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
.NET Framework 4.5
Sort by:
Default
Search Installed Te
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Location:
C:\source\fsharpintro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

Lab :
Stock market
Twitter emotion


Twitters
Messages
Event Stream
60
50
40
30
20
10
0
Unhappy - 50.00.%
Indifferent - 37.50.%
Happy - 12.50.%
Event Transformer
throttle
filter
Sentiment Analysis
Group by
Observer
Observer
Observer

Lab :
Ping Pong Subject

Reactive Extensions as a Bus



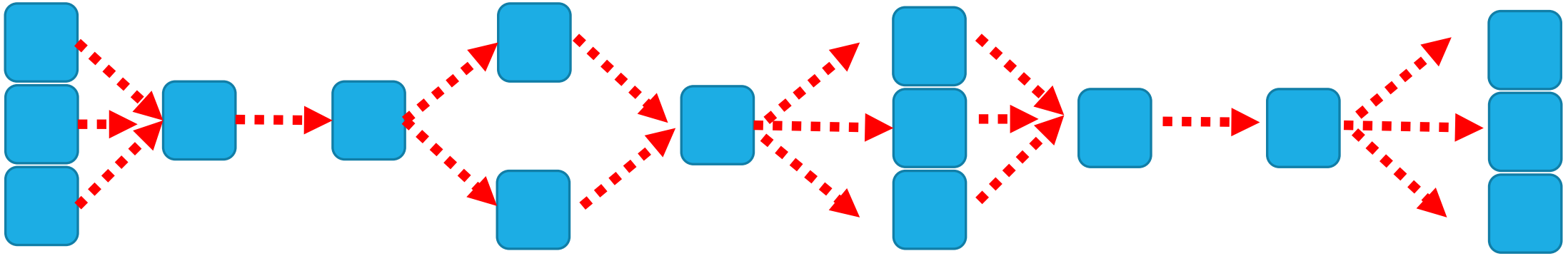
Ping Pong Actor Model with ISubject

```
// Simple Ping Pong Actor model using Rx

var ping = new Ping();
var pong = new Pong();


Console.WriteLine("Press any key to stop ...");

var pongSubscription = ping.Subscribe(pong);
var pingSubscription = pong.Subscribe(ping);


Console.ReadKey();

pongSubscription.Dispose();
pingSubscription.Dispose();

Console.WriteLine("Ping Pong has completed.");
```

Stream Processing

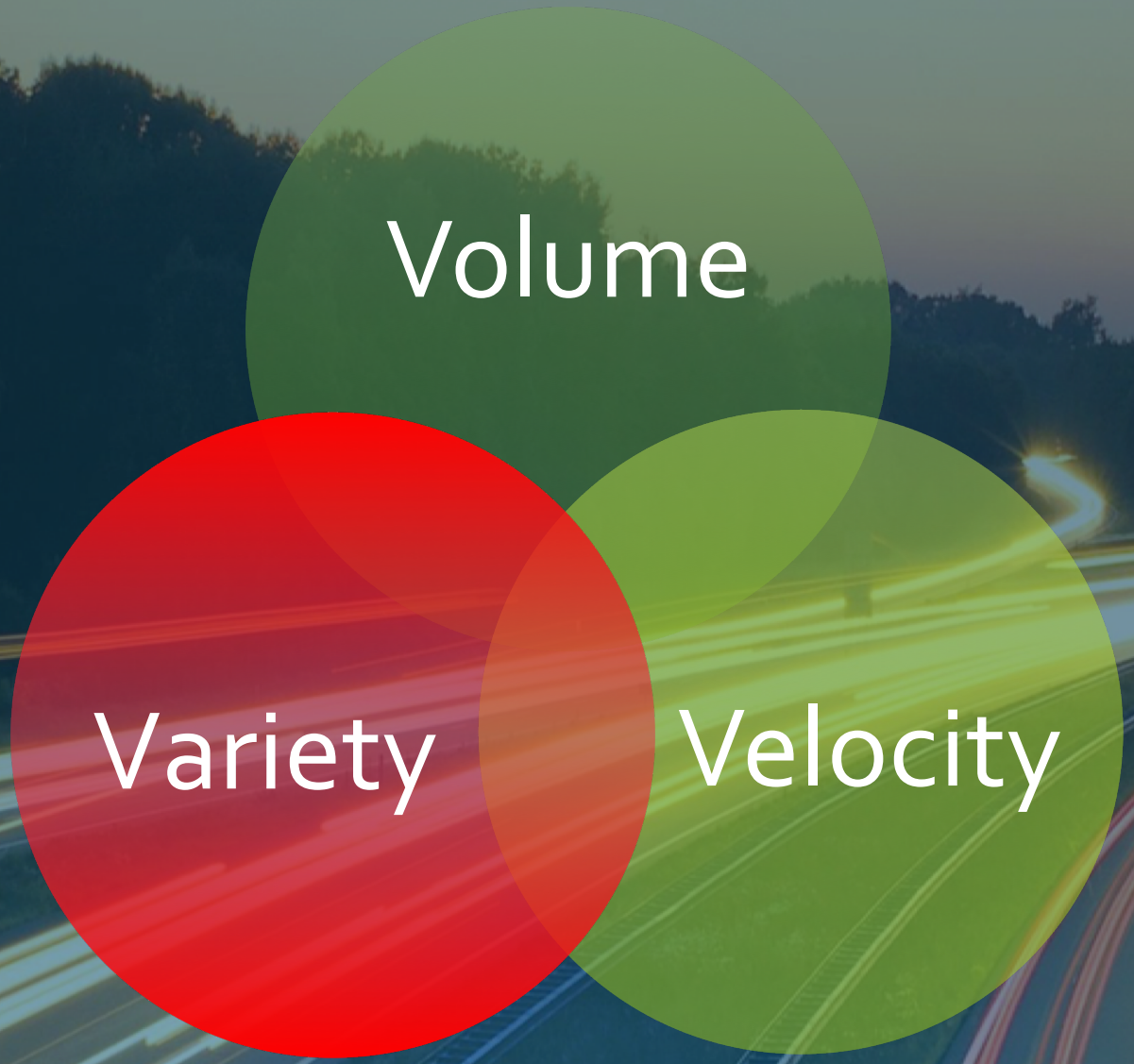
What is a Stream?

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



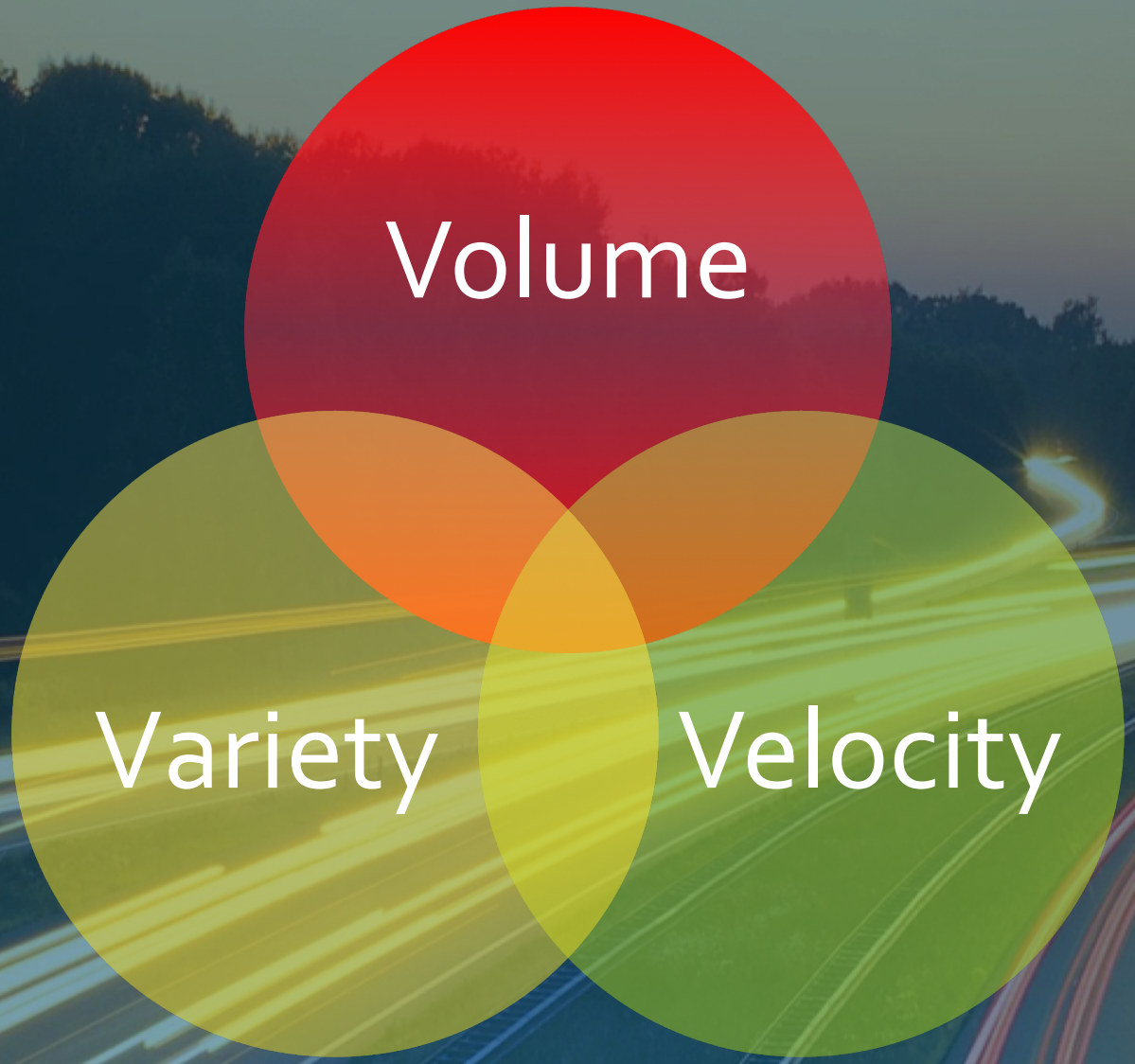
The 3 V's of data in motion

The 3 V's of data in motion

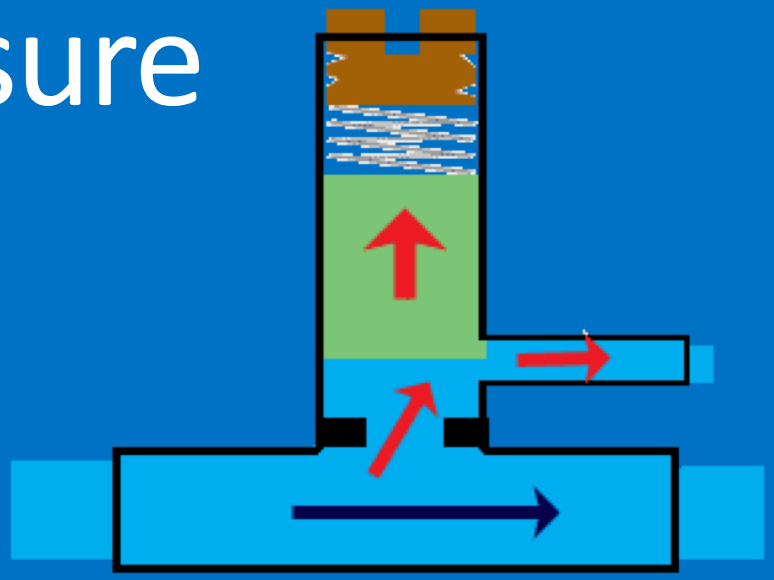


The 3 V's of data in motion

The 3 V's of data in motion



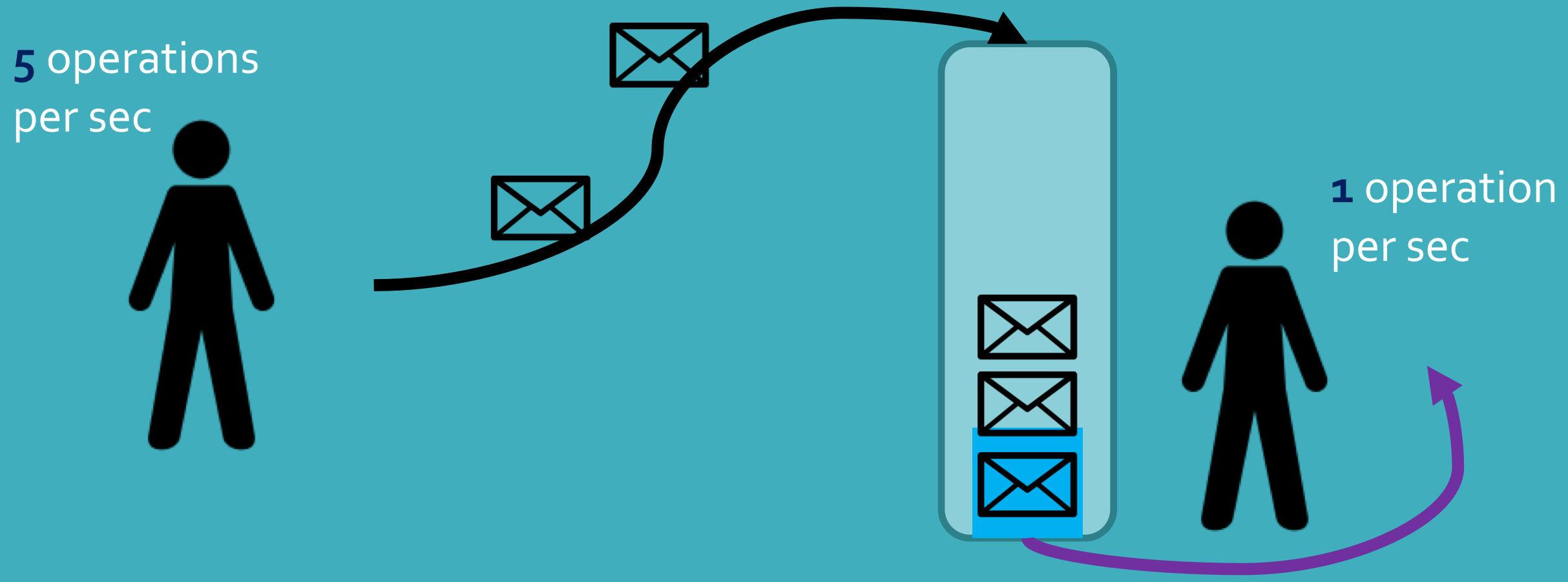
Back-Pressure



- Throttle
- Sample
- Buffer
- Window
- Debounce

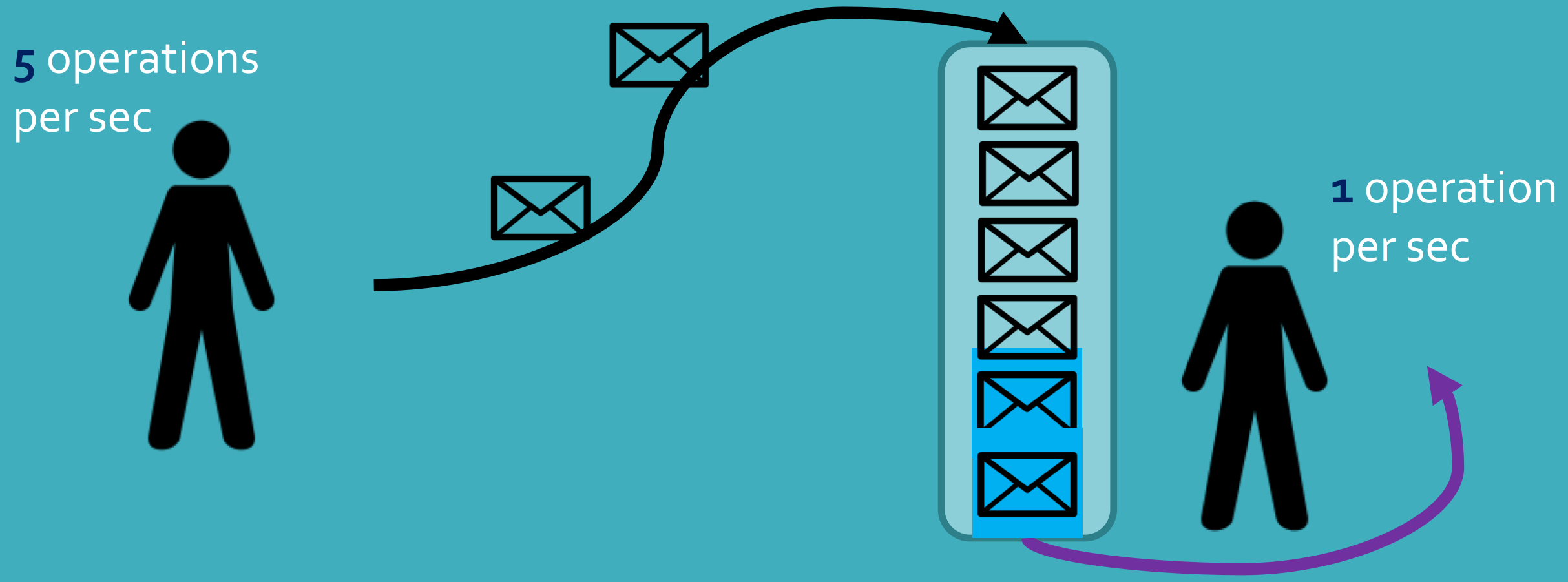
Push Model

Subscriber usually has some kind of buffer.

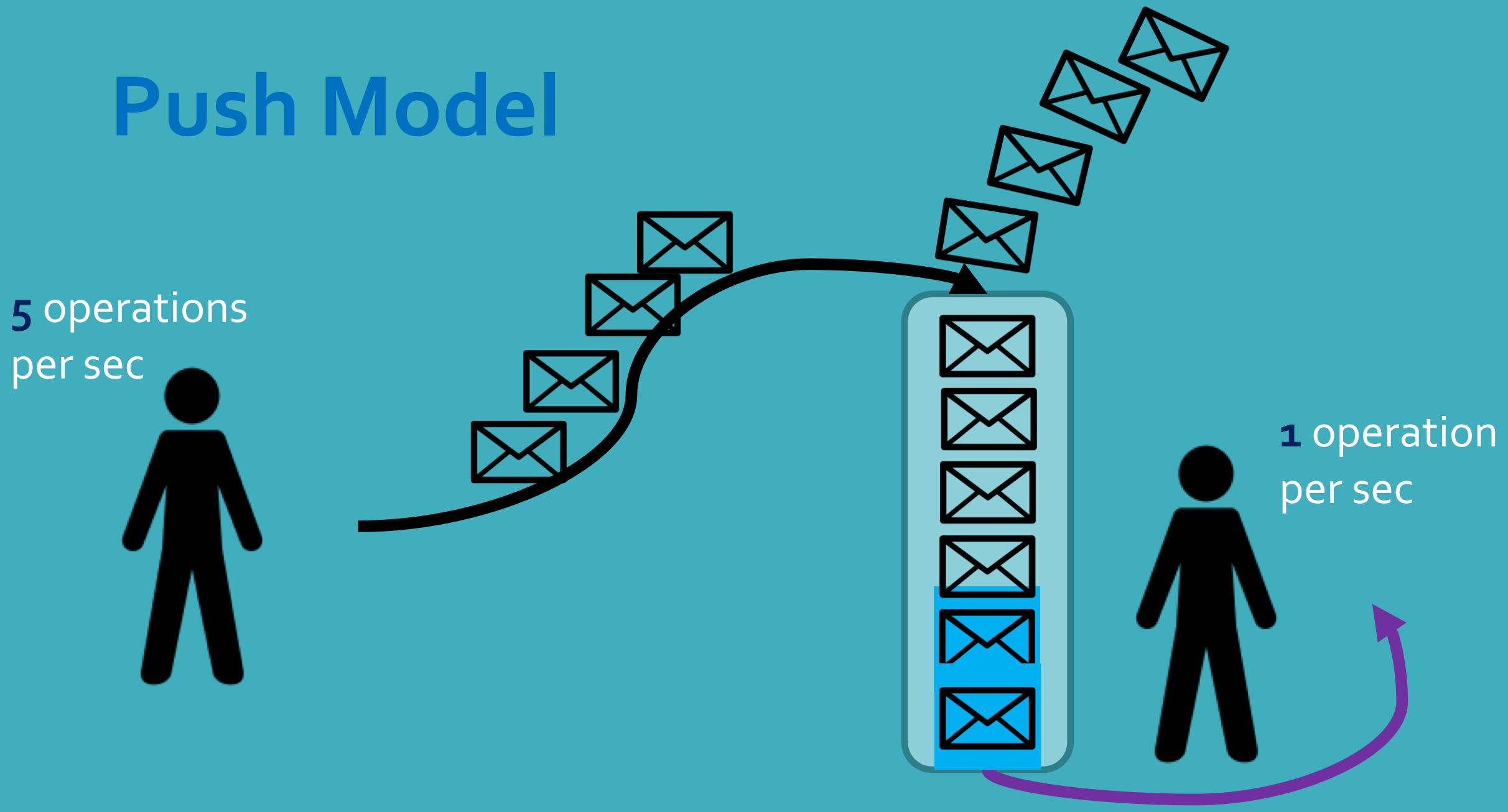


Push Model

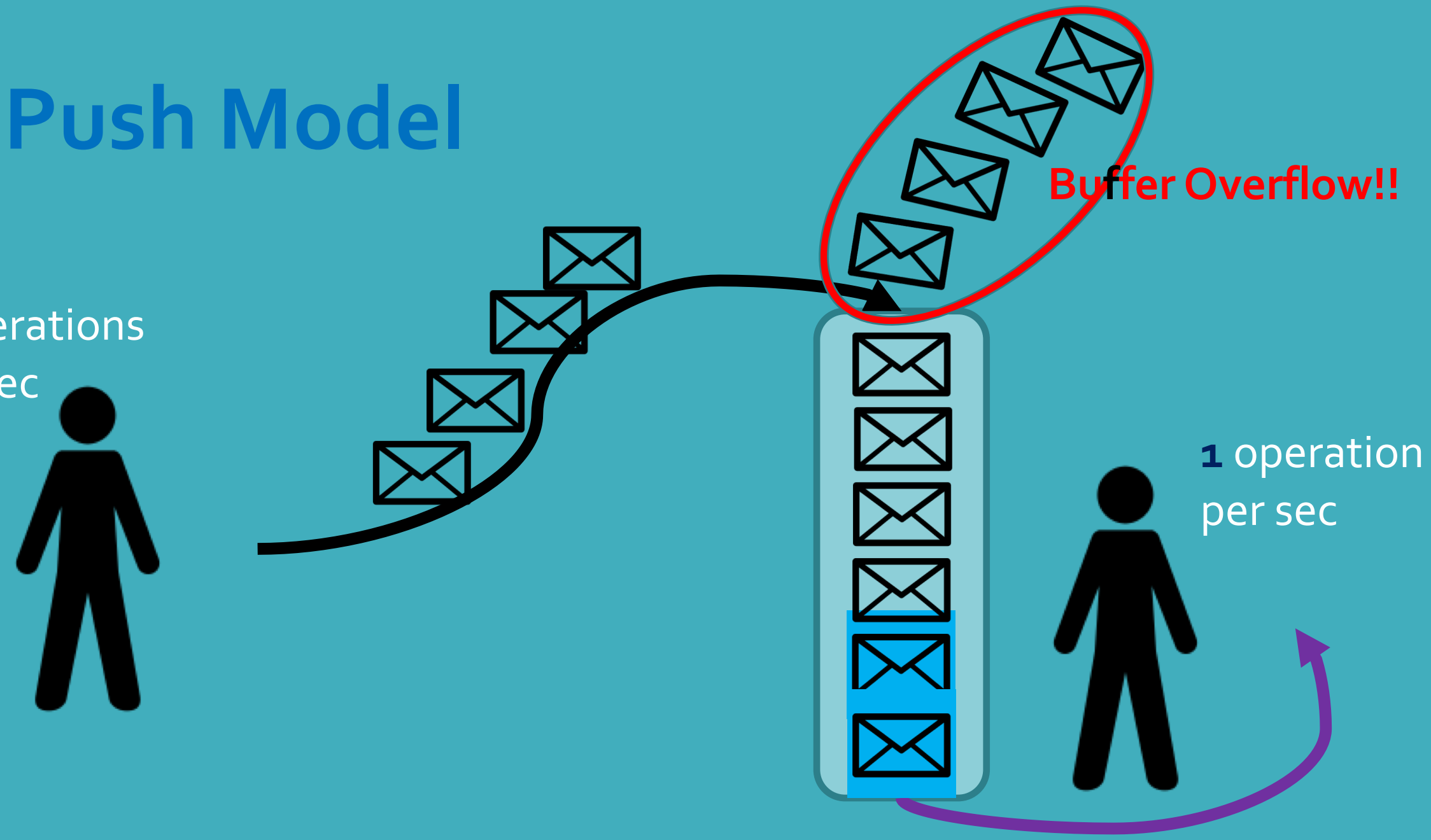
Subscriber usually has some kind of buffer.



Push Model

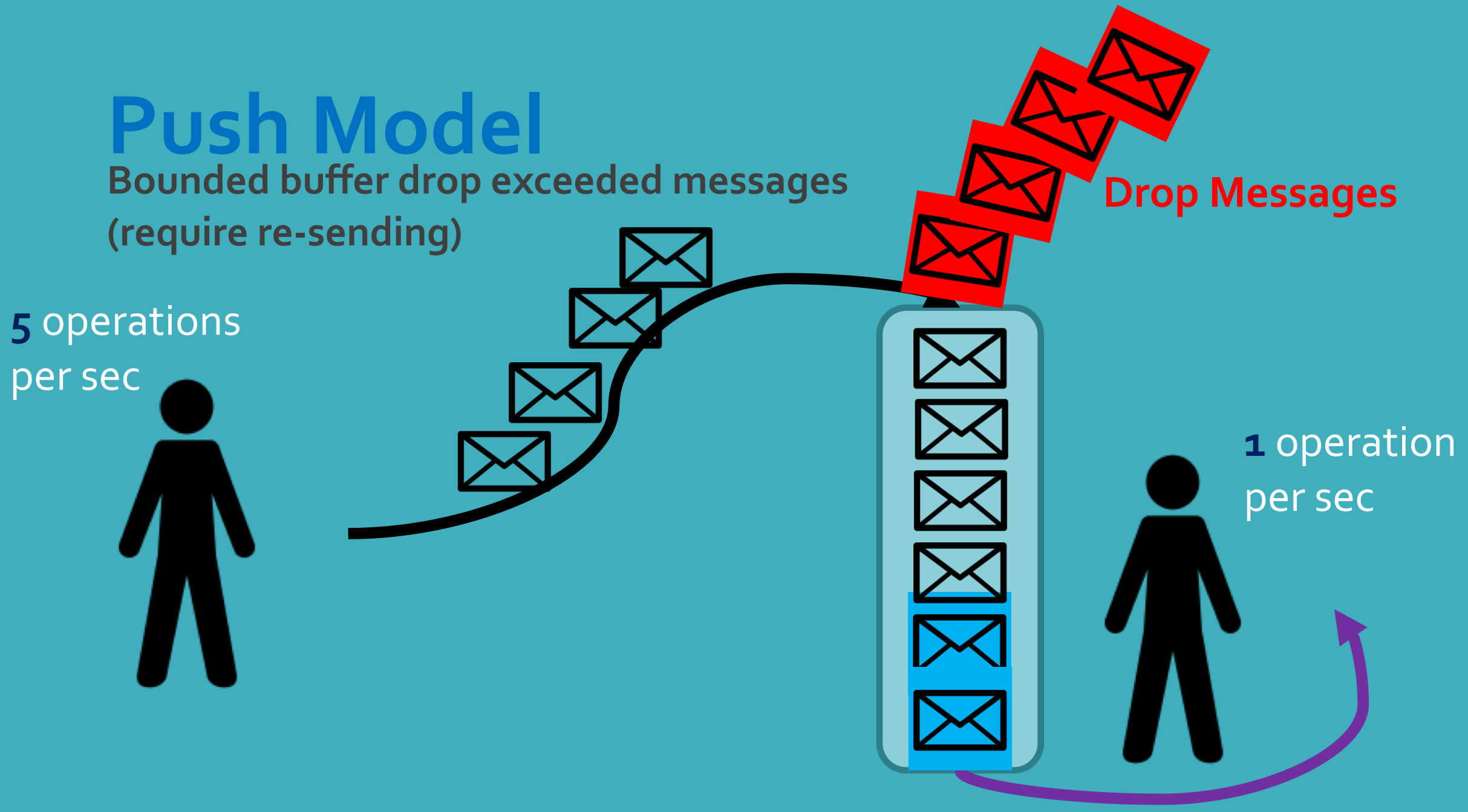


Push Model



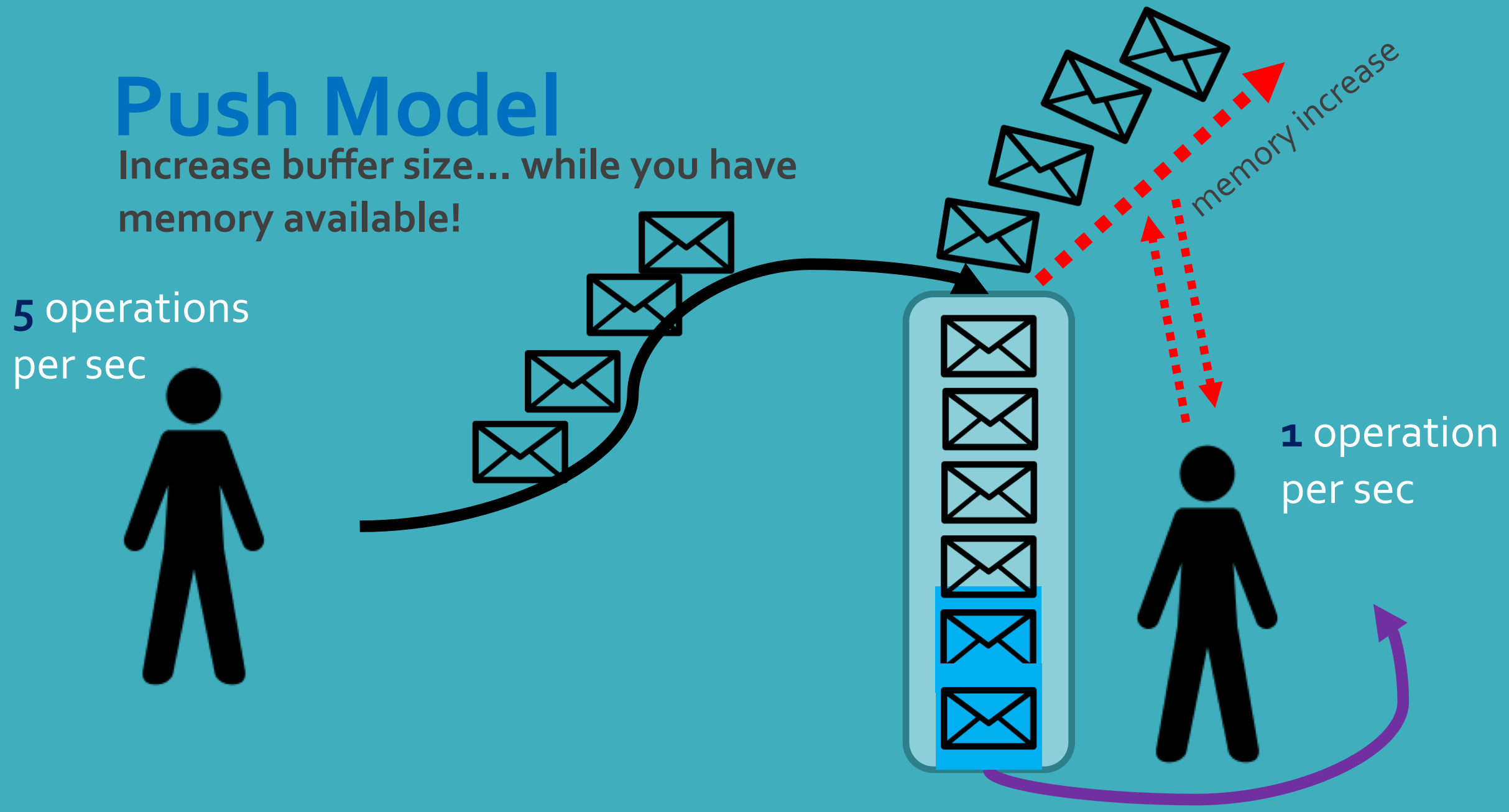
Push Model

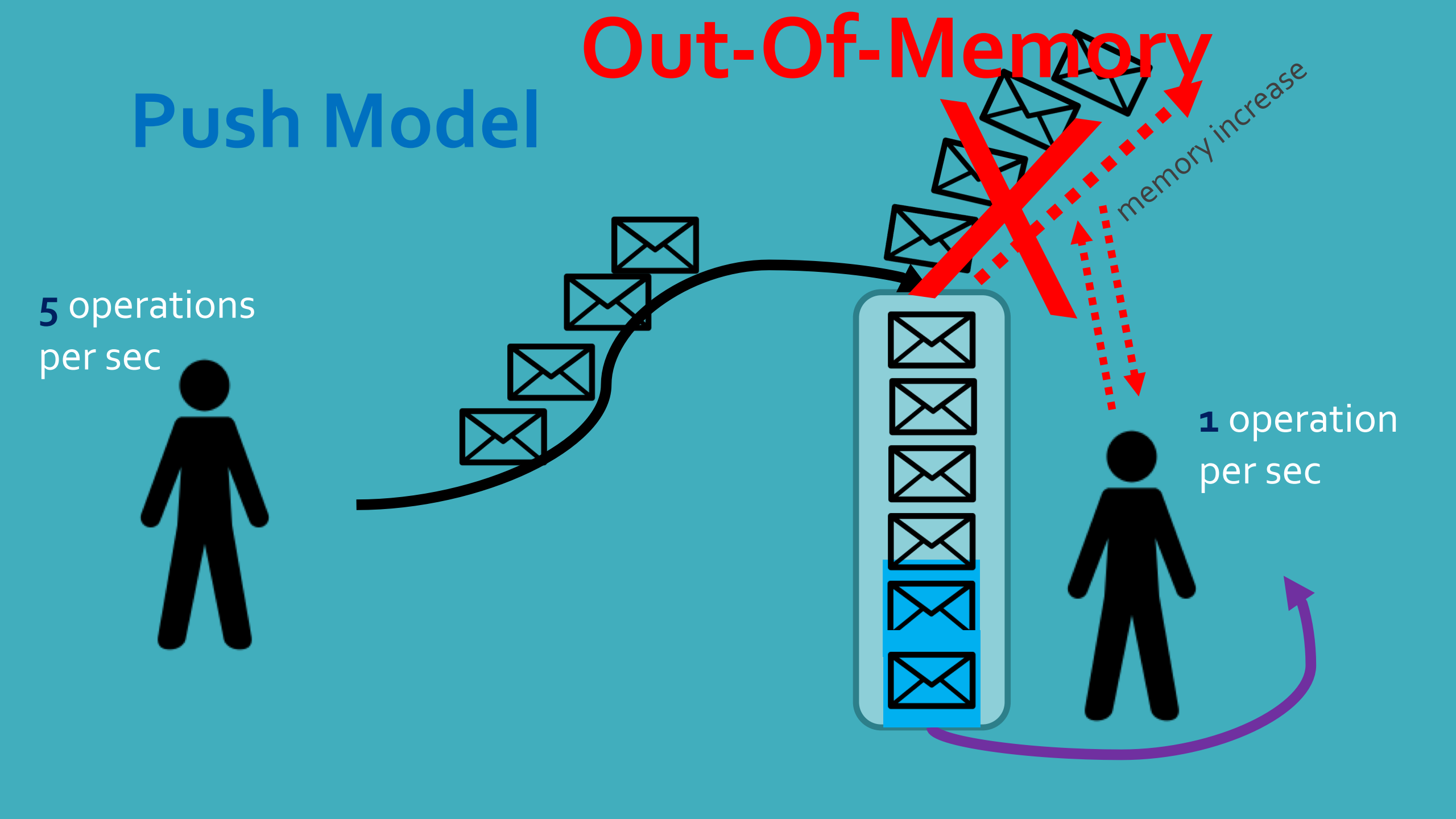
Bounded buffer drop exceeded messages (require re-sending)

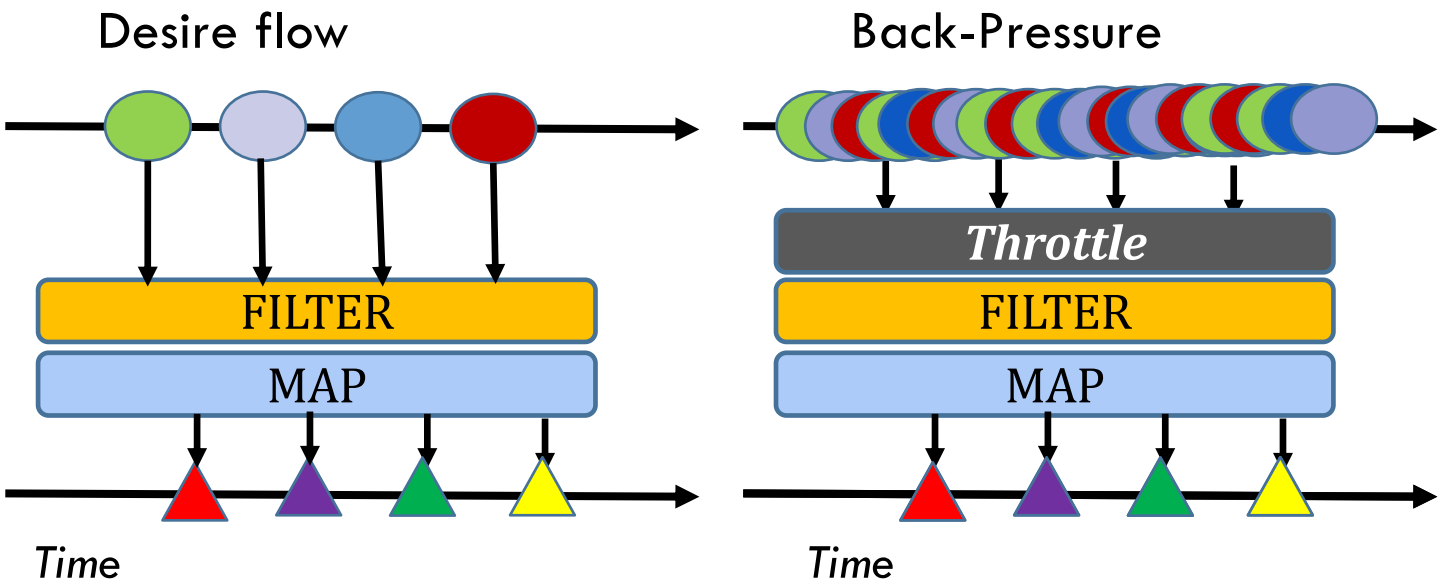


Push Model

Increase buffer size… while you have memory available!




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


Desire flow
FILTER
MAP
Time
Back-Pressure
Throttle
FILTER
MAP
Time

How to buffer the data without running out of memory?

From Actors to Reactive Streams

What an Actor Offers

- High Throughput
- Fault tolerance
- No manual thread management
- Scale Out & Up
- High Availability
- Asynchronous

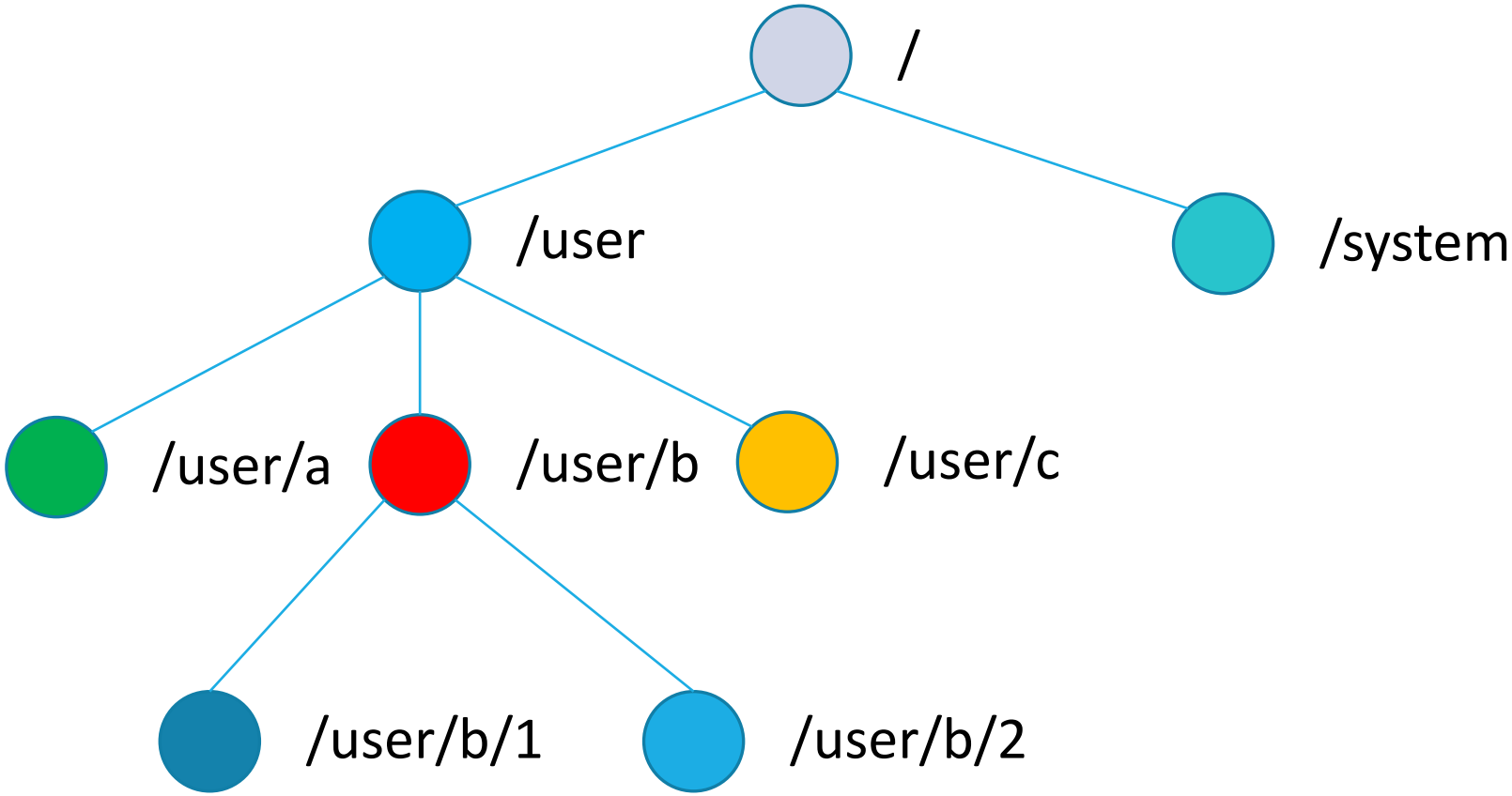
From Actors to Streams

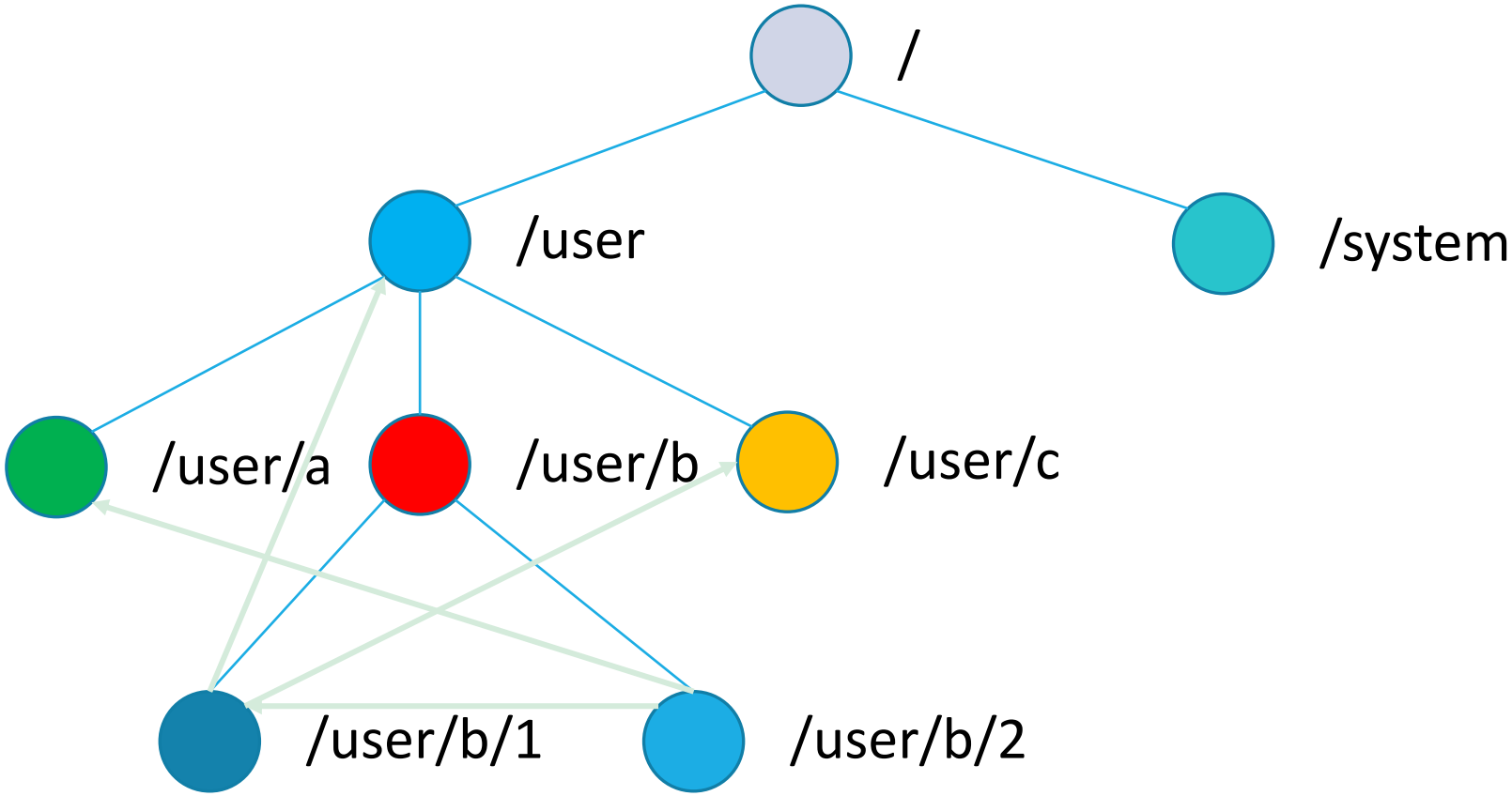
Claim: “actors do not compose”

“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

From Actors to Streams

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

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)
```

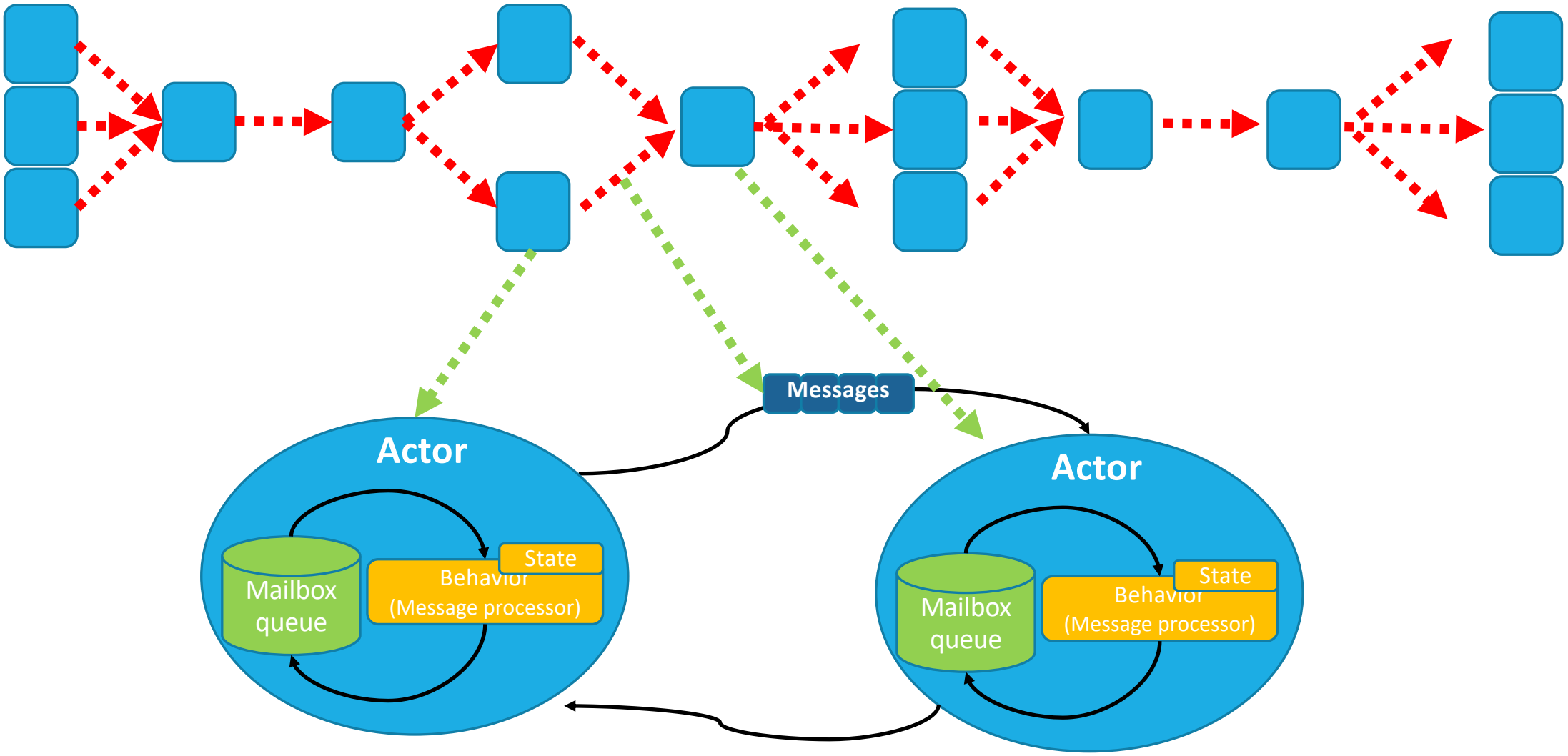
Reactive Streams

Let’s unlearn how to create individual actors

…and learn instead high level composition primitives

…that will internally create and connect instances of actors

High-Level Actor and Stream



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);
});
```

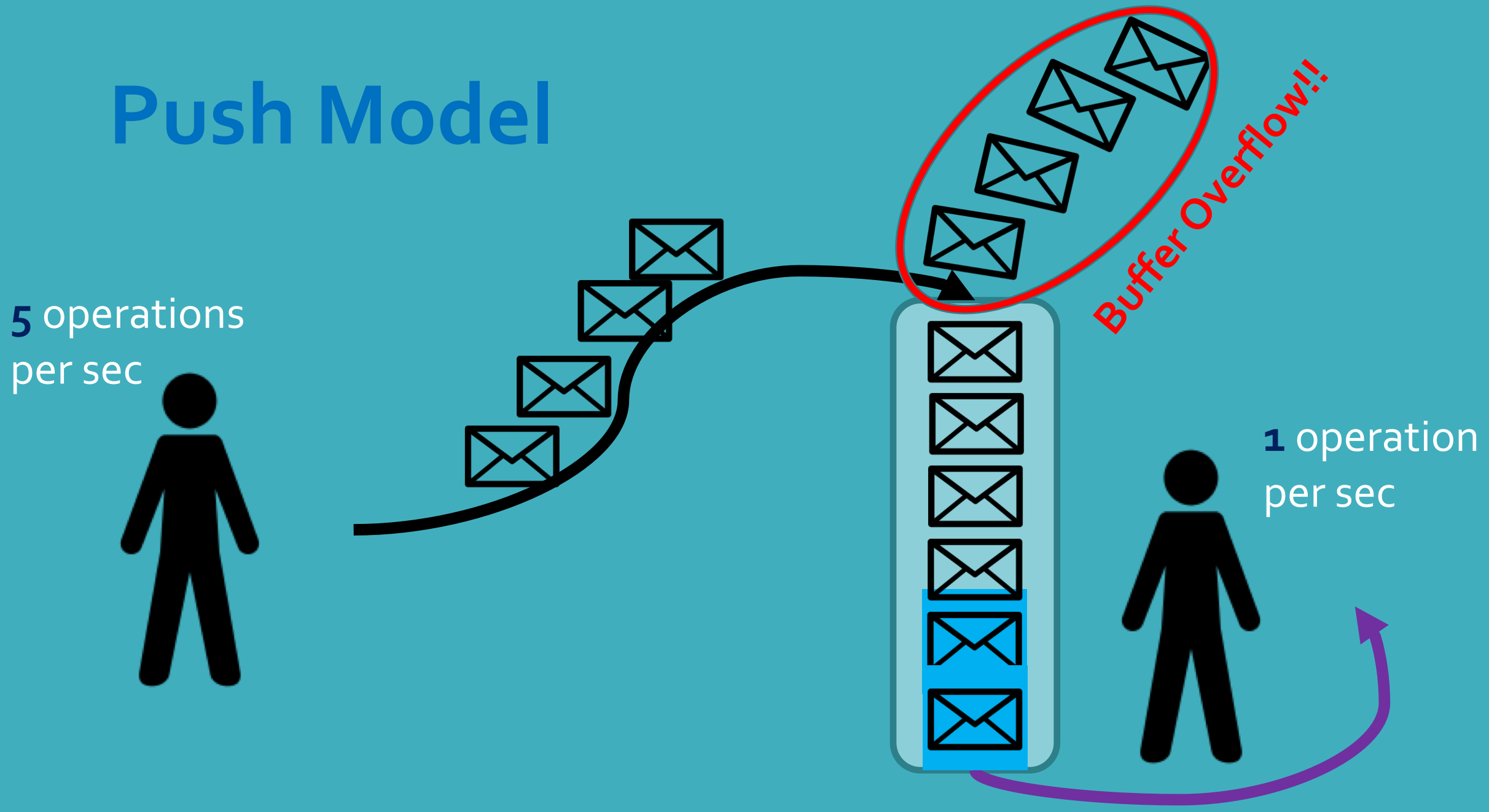
What do Reactive Streams solve?

Getting **Data** across an

Asynchronous Boundary

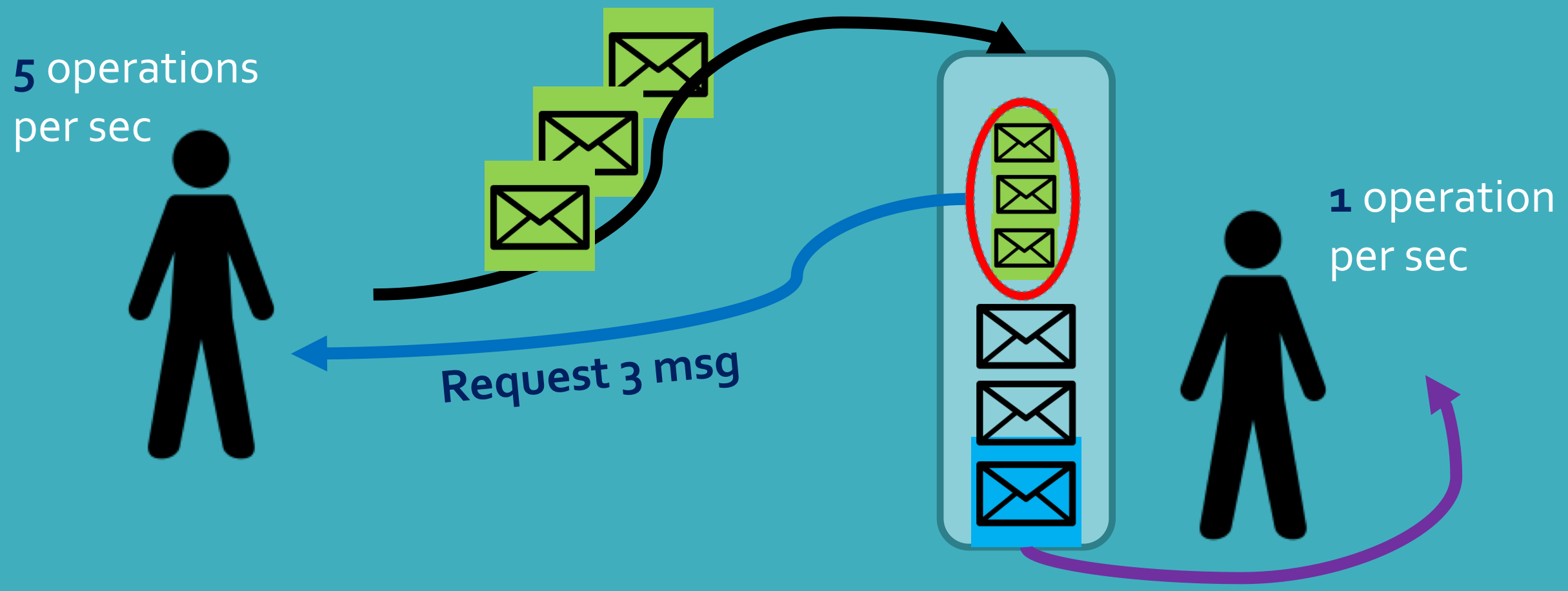
without running **Out-Of-Memory**

Push Model

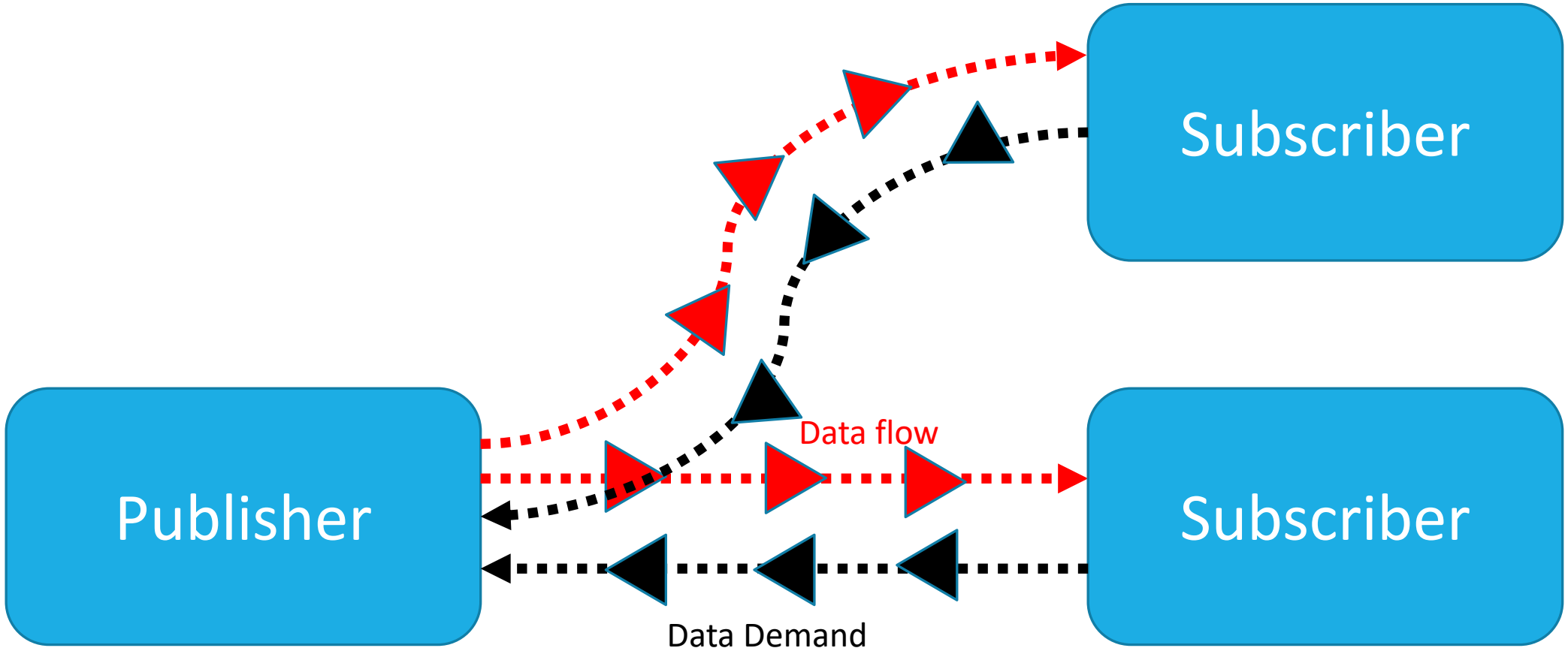


Dynamic Push-Pull

Pull based backpressure

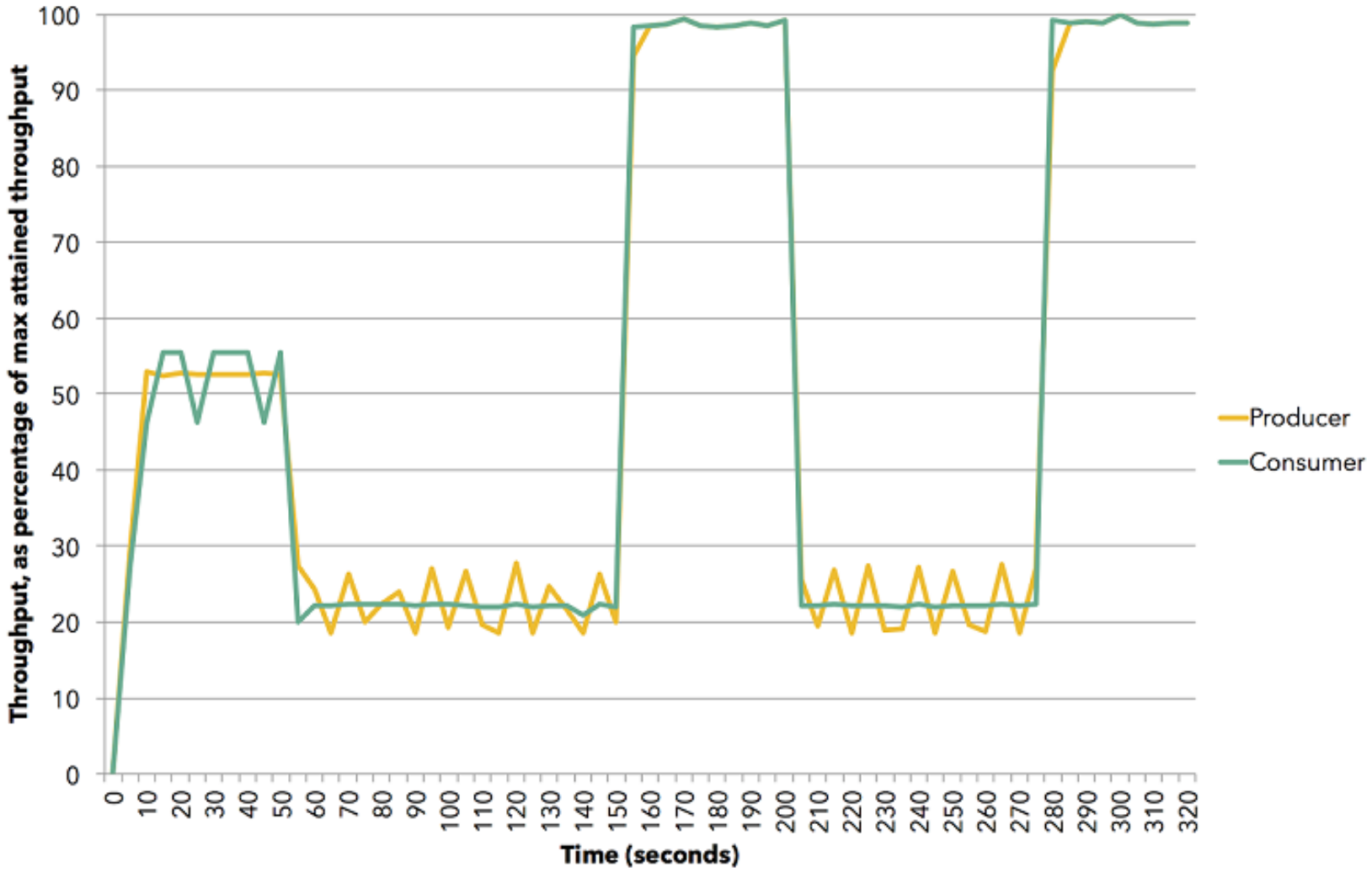


Splitting and merging the data flow



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();
}
```



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

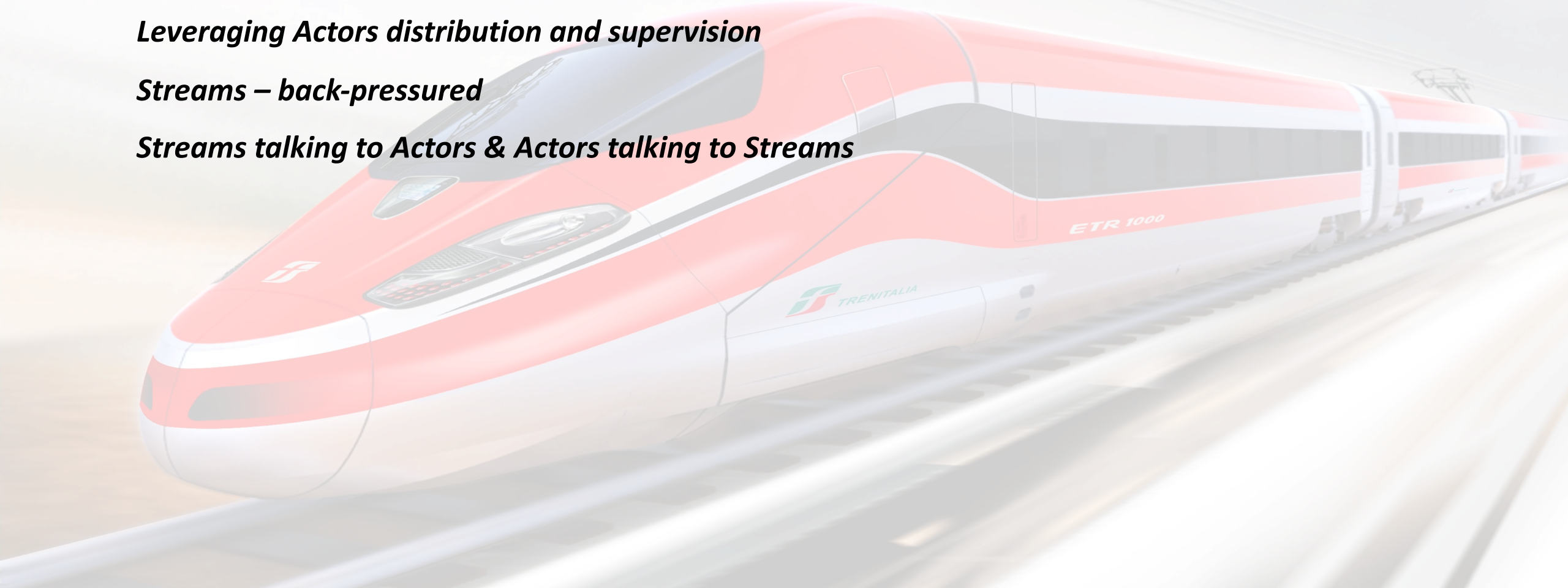
Akka Streams

Streams complement Actors, they do not replace them.

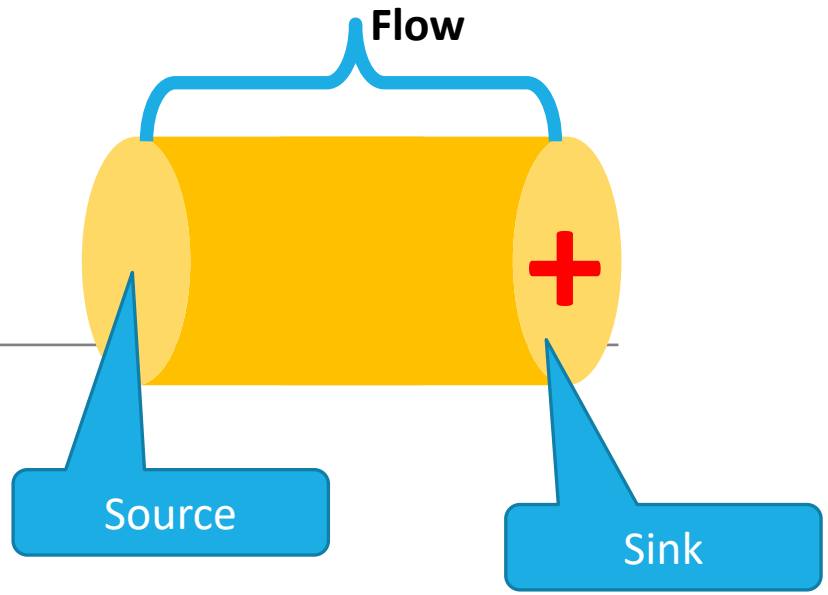
Leveraging Actors distribution and supervision

Streams – back-pressured

Streams talking to Actors & Actors talking to Streams



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)

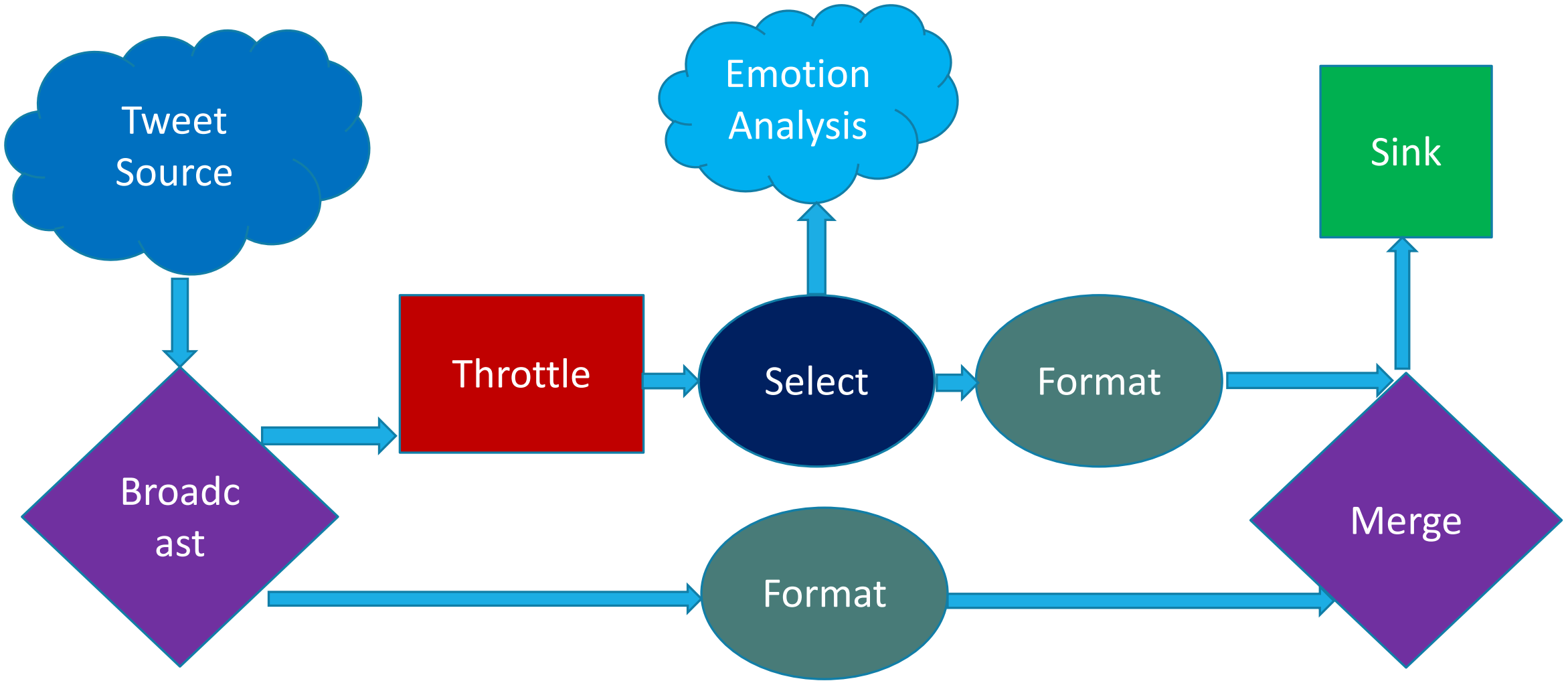
Execution begins with stream materialization

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

Demo - Case Study Reactive Tweets


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

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra