Near Real-Time Stream Analysis

Design and Implementation of a High-Performance Stream-Oriented Processing System

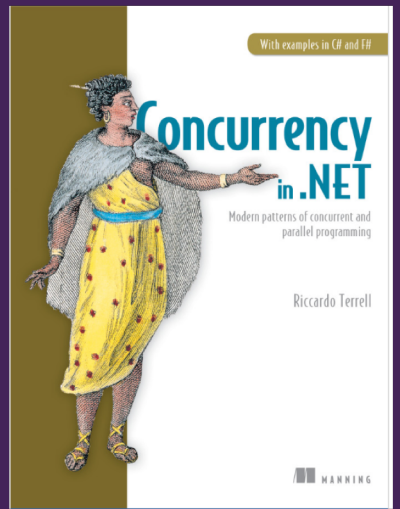
BY RICCARDO TERRELL - @RIKACE

Agenda

- What & Why Reactive Streams
- Concepts and goals of Reactive Stream
- Back pressure and how the Actor programming model fit
- How to design high-performant Stream-Processing
- Stream-Processing in action with Akka.Streams and F#

Introduction - Riccardo Terrell

- Originally from Italy, currently - Living/working in Charlotte, NC
- +/- 20 years in professional programming
  - C++/VB → Java → .Net C# → Scala → Haskell → C# & F# → ??
- Author of the book “Functional Concurrency in .NET” – Manning Pub.
- *Polyglot programmer - believes in the art of finding the right tool for the job*
- *Organizer of DC F# User Group and Pure Functional DC*

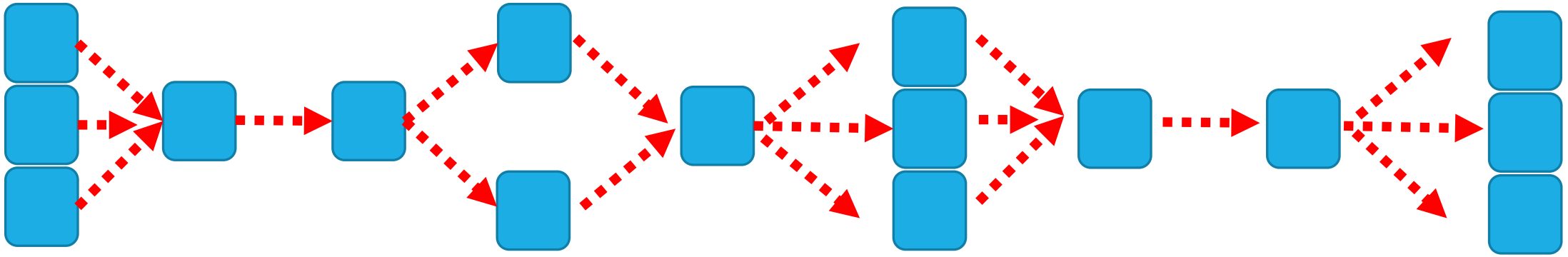


@trikace *www.rickyterrell.com* *tericcardo@gmail.com*

35

What is a Stream?

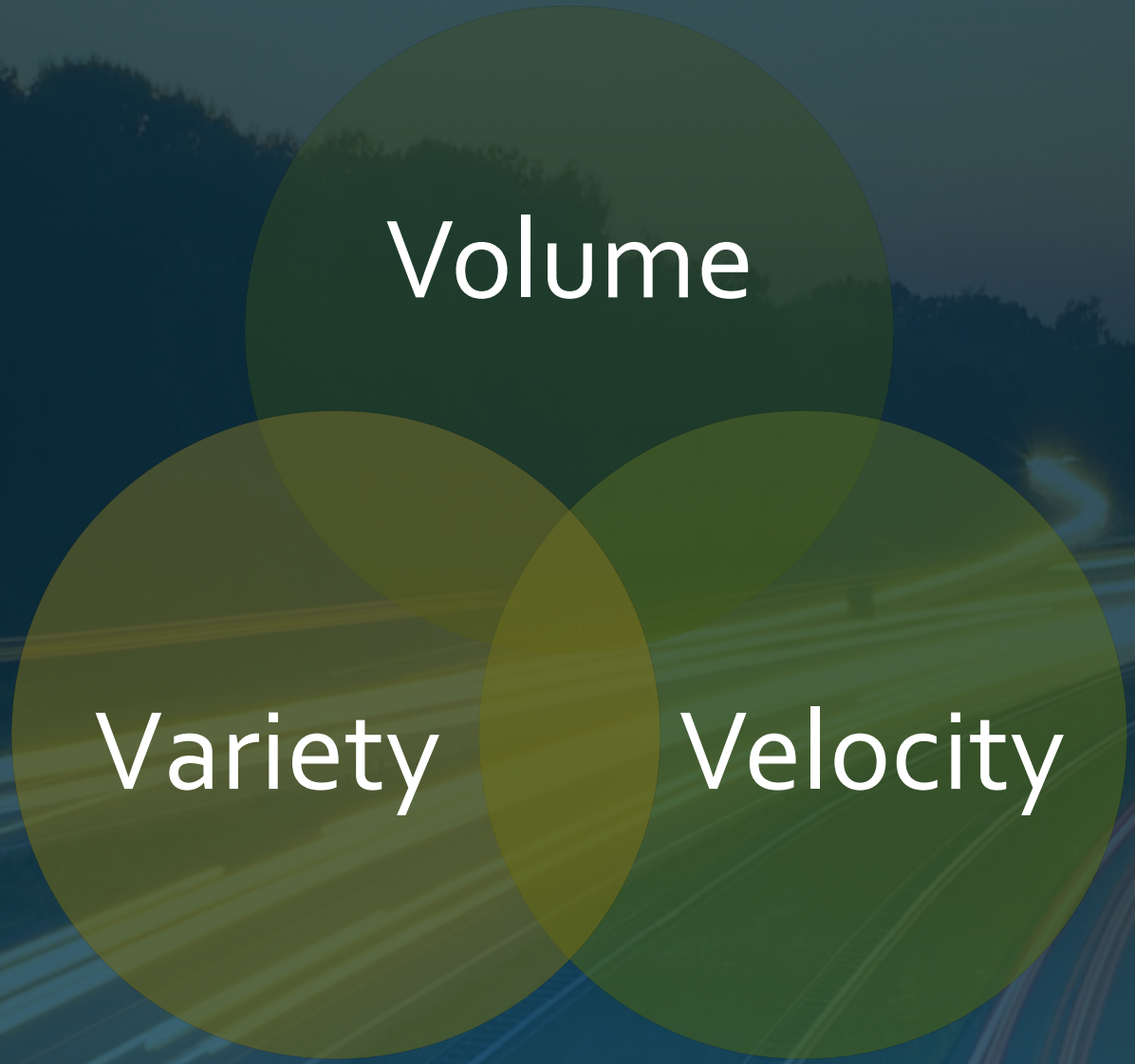
- Time-dependent sequence of elements
- Possibly unbounded in length
- Ephemeral flow of data
- Focused on describing transformation
- Can be formed into processing networks



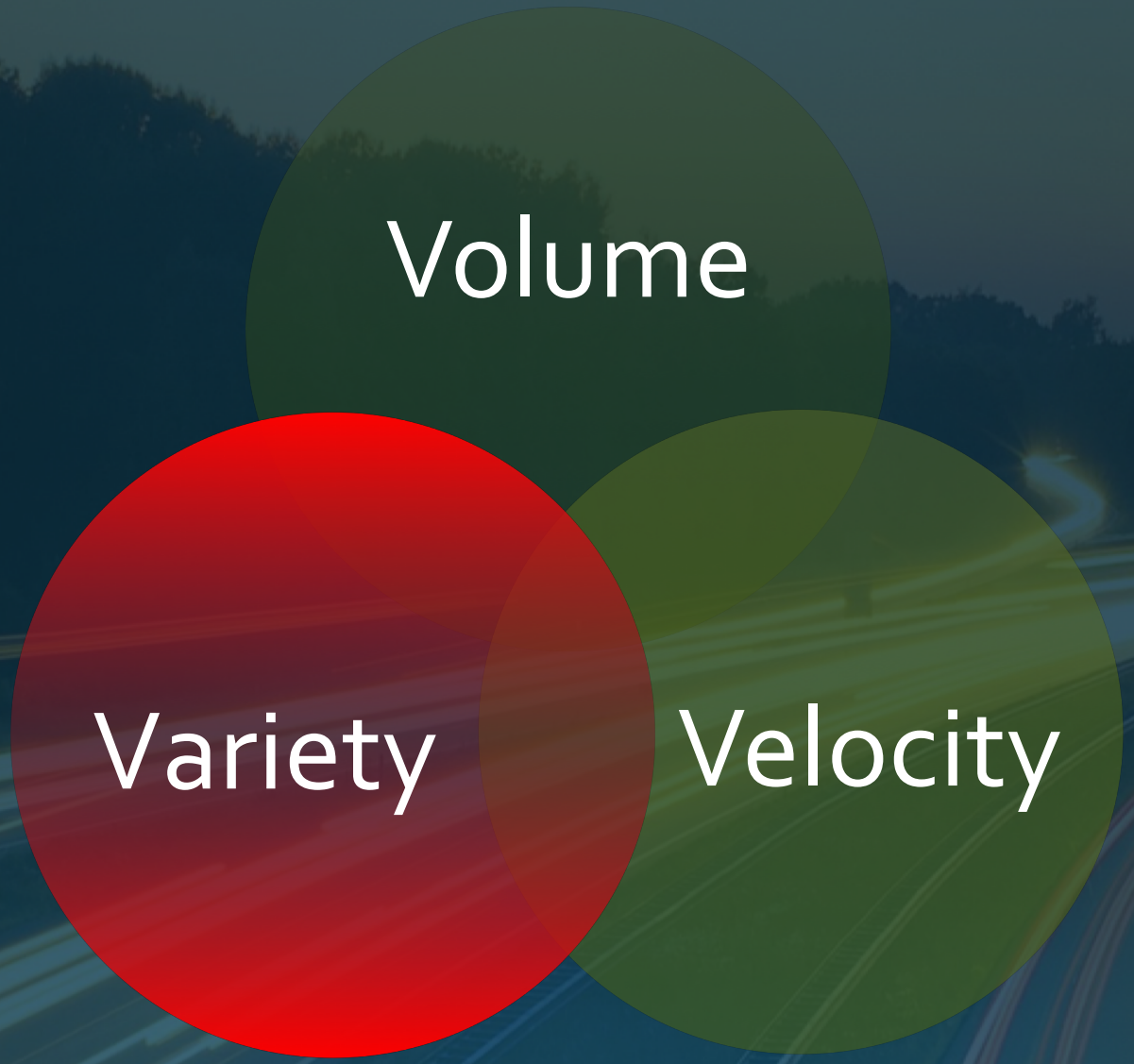

15:15
15:15


Data in motion

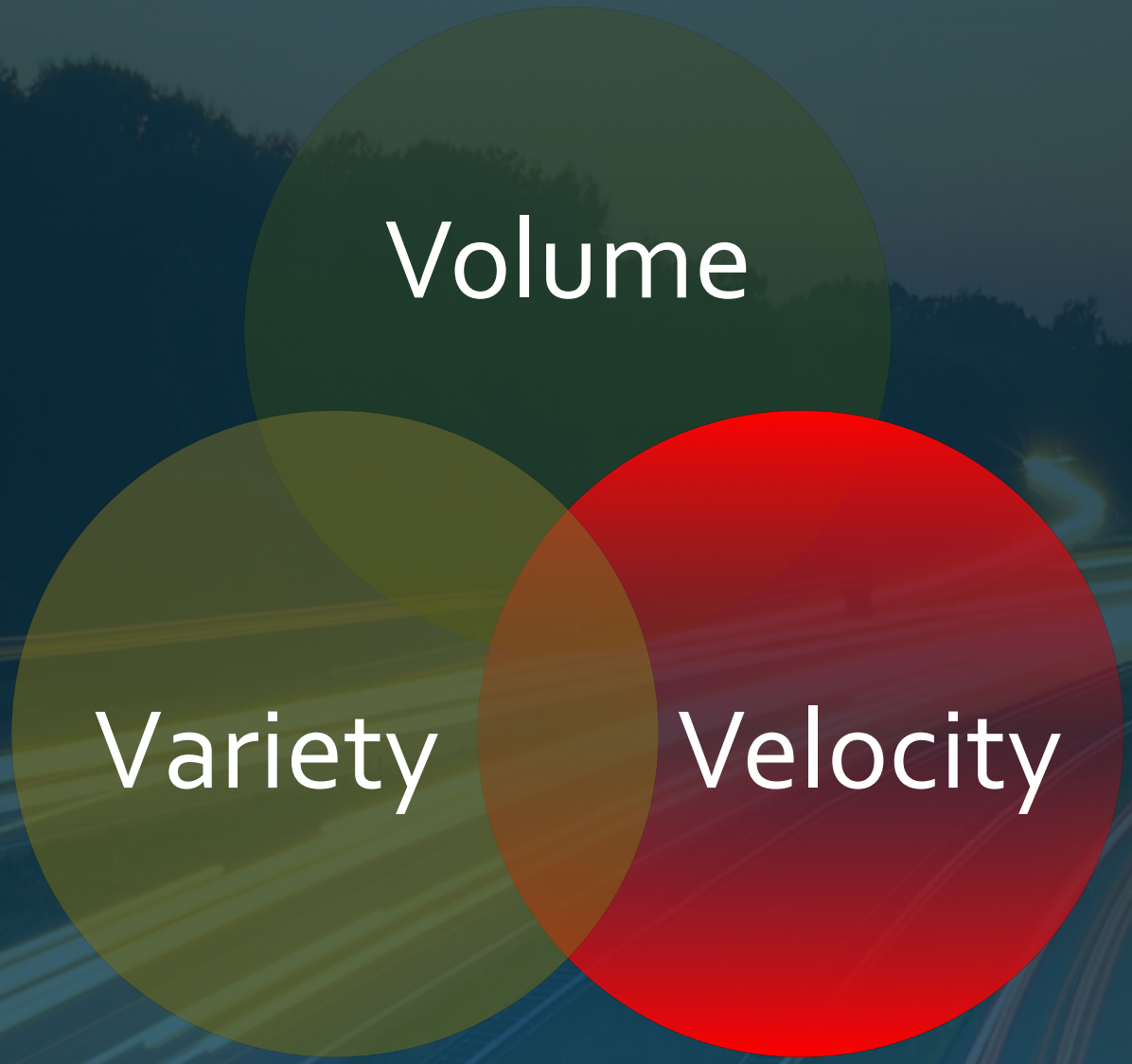
The 3 V's of data in motion



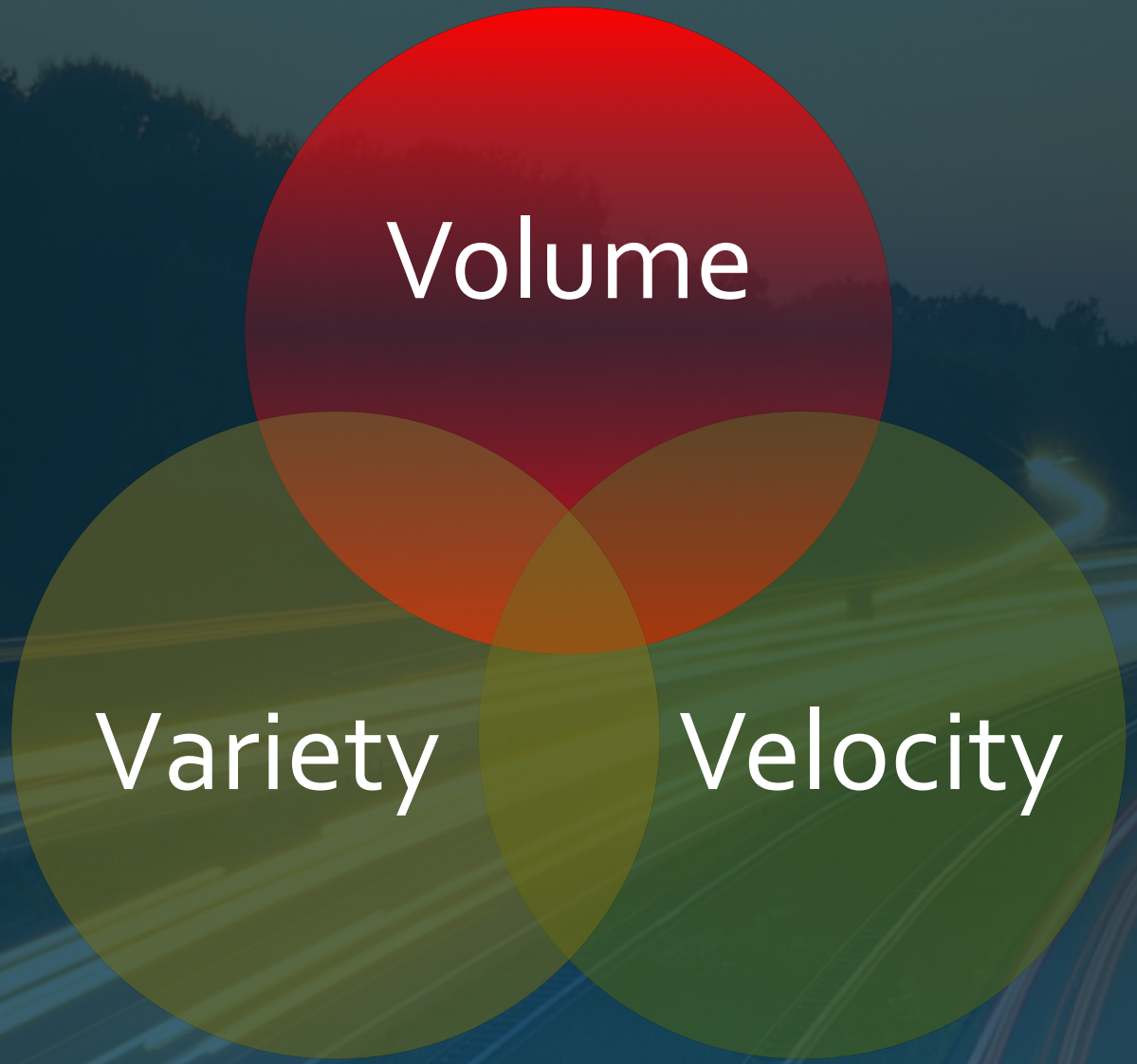
The 3 V's of data in motion



The 3 V's of data in motion



The 3 V's of data in motion



The 5 V's of data in motion

Volume

Variety

Velocity

Value

Veracity

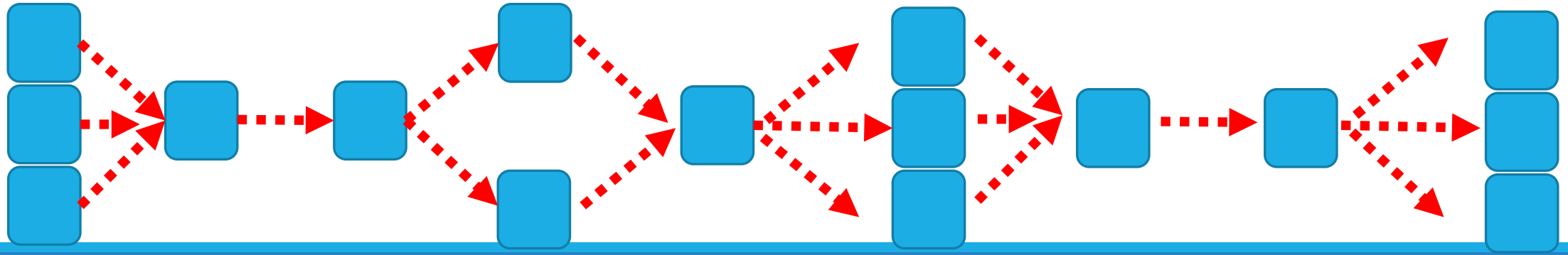
Attributes for a scalable Stream-Processing system

Asynchronous : The process has to be asynchronous in order to ensure optimal utilization of computing resources

Nonblocking : This is required to ensure bounded latency

Elastic for handling back pressure : This is needed when producers push data toward consumers and the consumer may be processing at a slower rate than what the producer is generating.

Compositional : This is a quality that we demand all APIs should have. This will lead to a better programming model.



Reactive Programming

(and 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

Reactive Streams – Properties and Design

“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

Reactive Systems – Properties and Design

Event Driven : react to events (events are data), events can be handled asynchronously

Scalable : react to load serving numerous clients simultaneously with bounded resources

Responsive : react to load responding in a timely manner if at all possible.

Resilient : react to failures The system stays responsive in the face of failure.

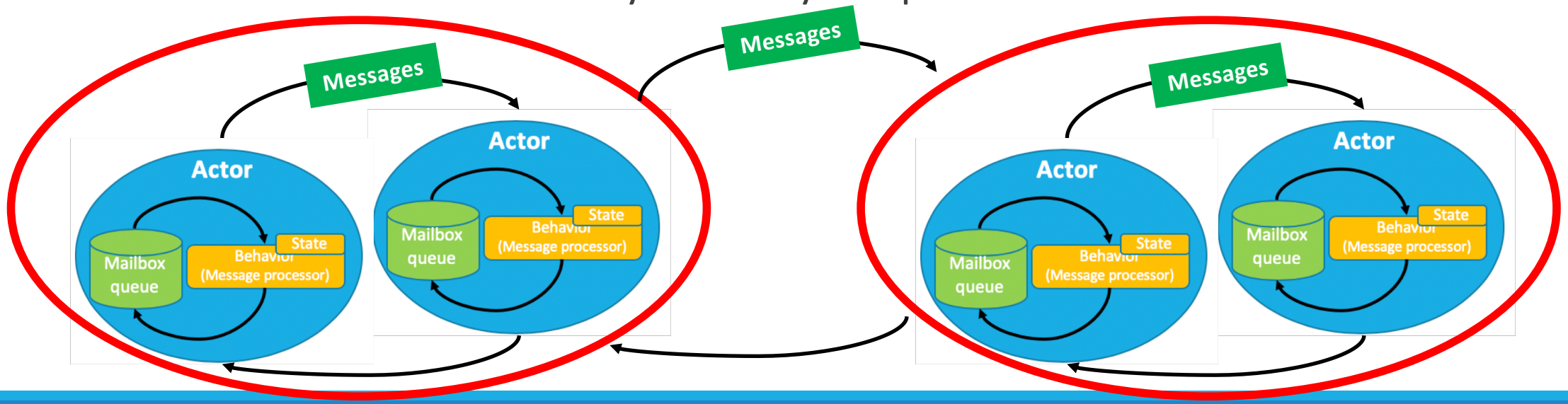
Reactive Systems – Properties and Design

Event Driven : react to events (events are data), events can be handled asynchronously

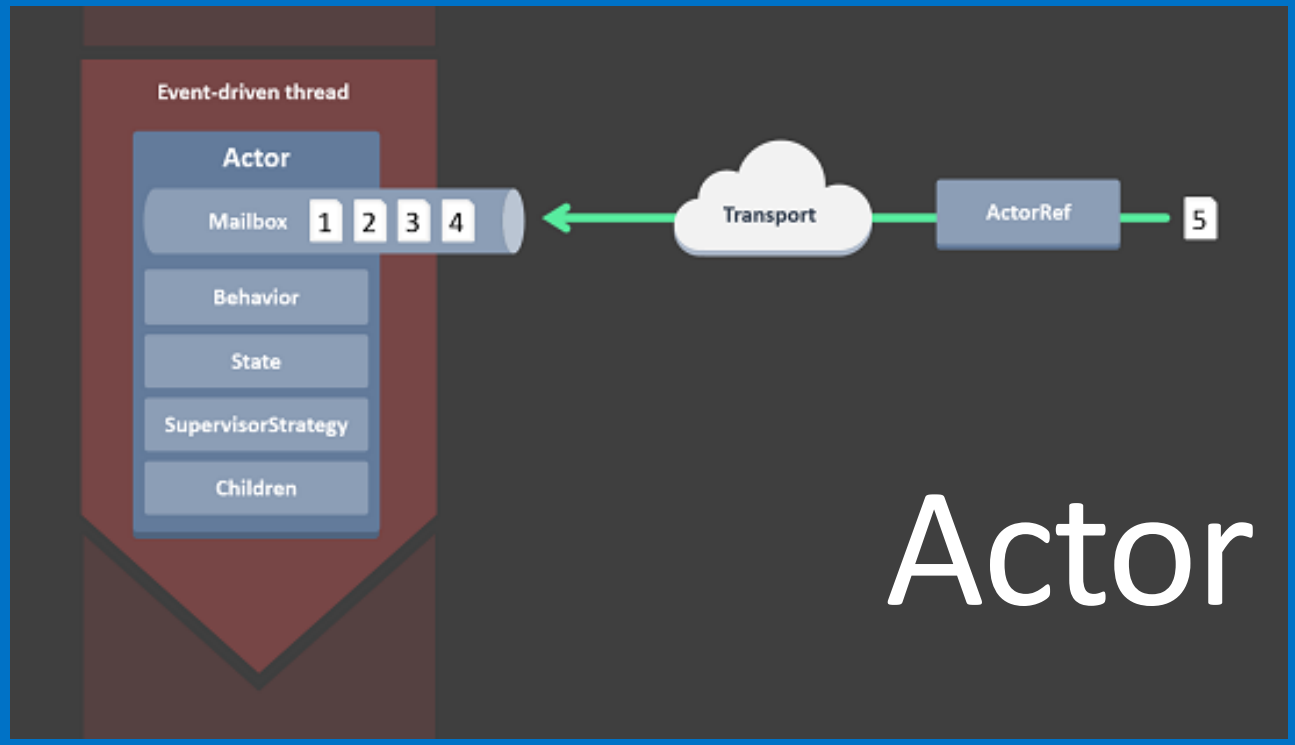
Scalable : react to load serving numerous clients simultaneously with bounded resources

Responsive : react to load responding in a timely manner if at all possible.

Resilient : react to failures The system stays responsive in the face of failure.



From



Actor Model

to Reactive Streams

What an Actor Offers

High Throughput

Fault tolerance

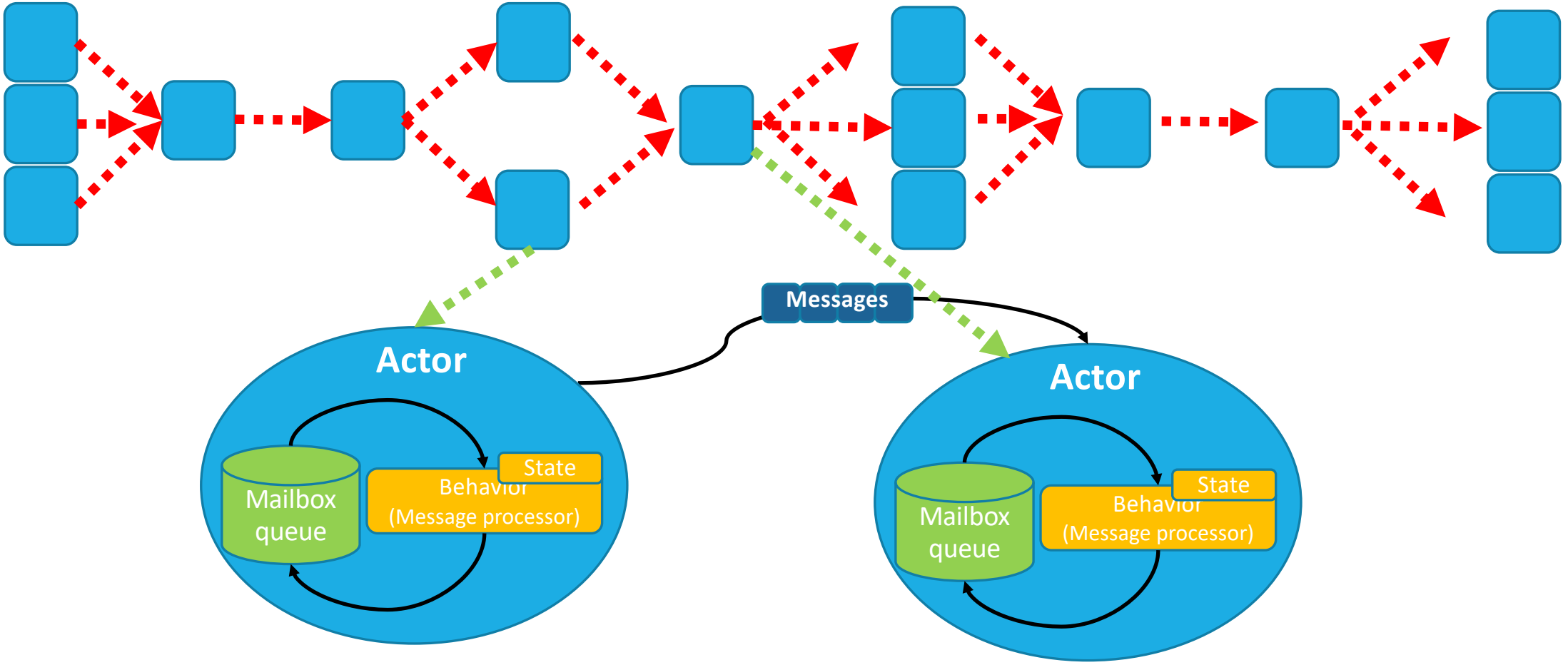
No manual thread management

Scale Out & Up

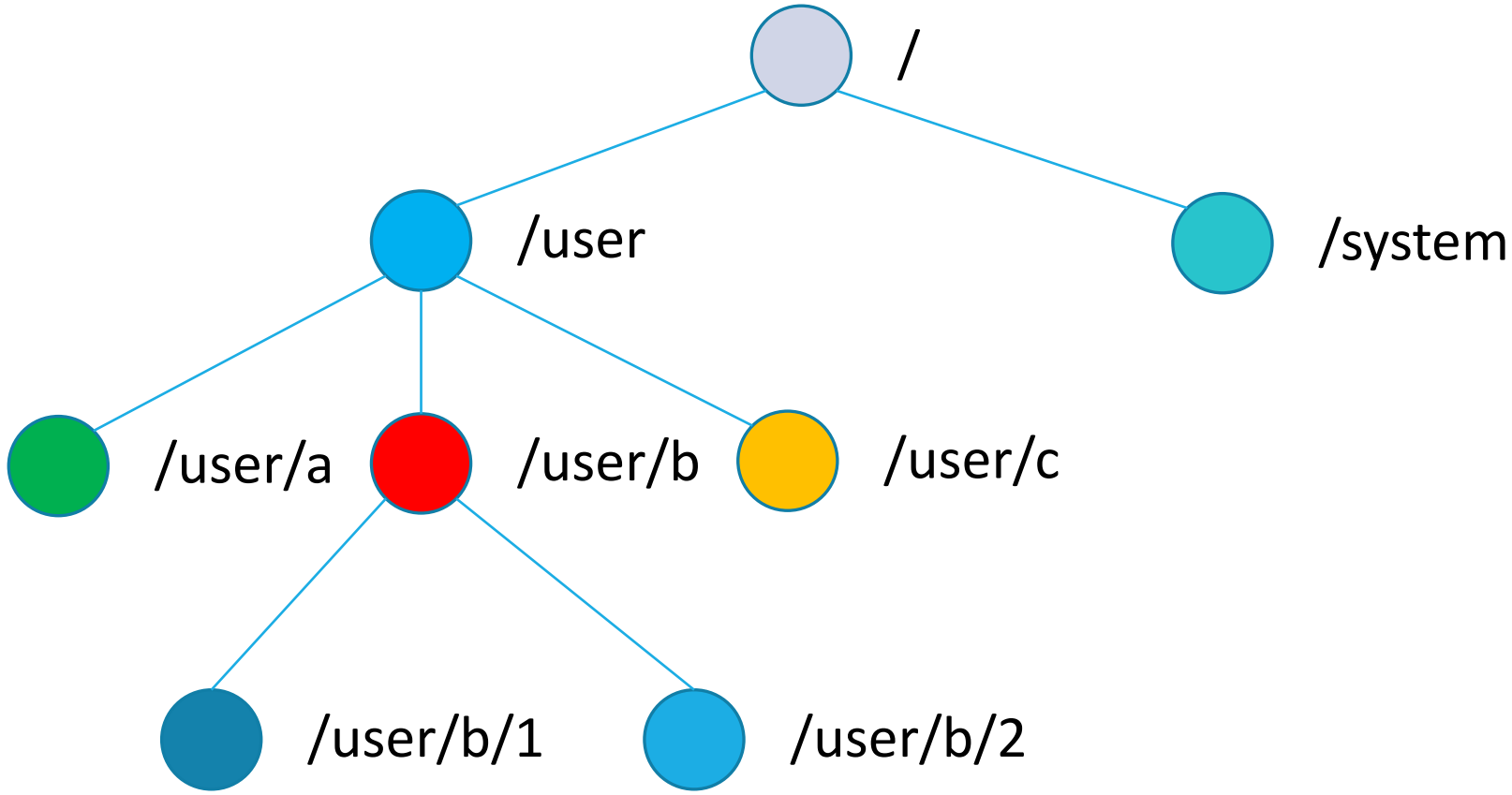
High Availability

Asynchronous

Message Passing & Stream Modeling



From Actors to Streams
actors are hard to compose



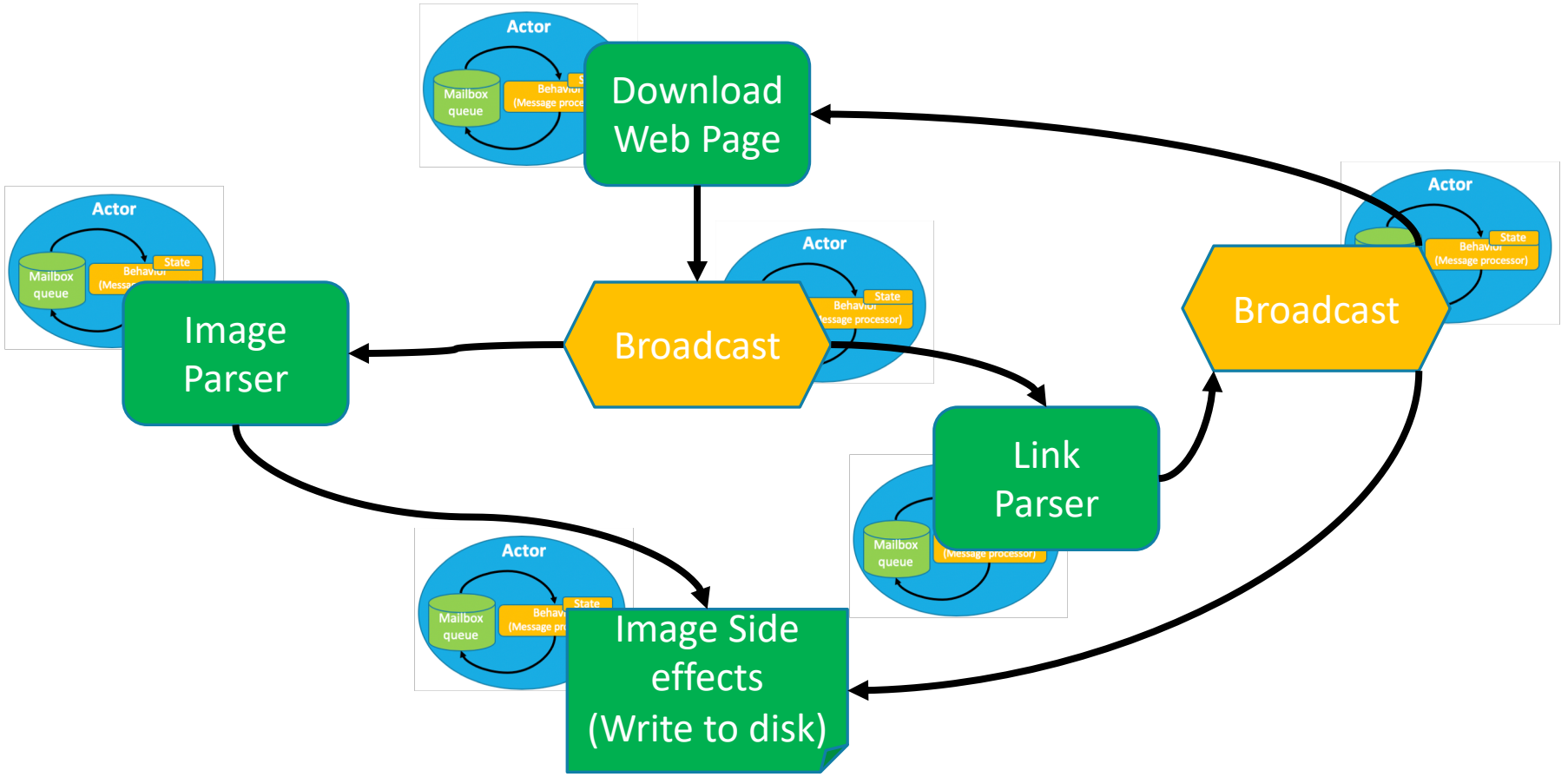
Actor implementation class in F#

```
let behavior (f:'a -> Async<'b>) (m:Actor<'a>) =
    let rec loop () = actor {
        let! msg = m.Receive ()
        printfn "Actor received : %O" msg
        let! res = f msg
        m.Context.Sender() <! res
        return! loop ()
    }
    loop ()
```

Actor implementation class in F#

```
let behavior (f:'a -> Async<'b>) (targetRef:ICanTell<'b>) (m:Actor<'a>) =
    let rec loop () = actor {
        let! msg = m.Receive ()
        printfn "Actor received : %O" msg
        let! res = f msg
        targetRef <! res
        return! loop ()
    }
    loop ()
```

Actors are hard to compose
An Actor-based WebCrawler



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

Like a in LINQ

```
var results = ctx.Employee
                 .Join(ctx.Manager,
                     c => c.EmployeeID,
                     p => p.ManagerID,
                     (c, p) => new { c, p })
                 .Where(z => z.c.Hired >= fromDate)
                 .OrderByDescending(z => z.c.Salary)
                 .Select(z => z.Fullname)
                 .ToList();
```

Akka Streams

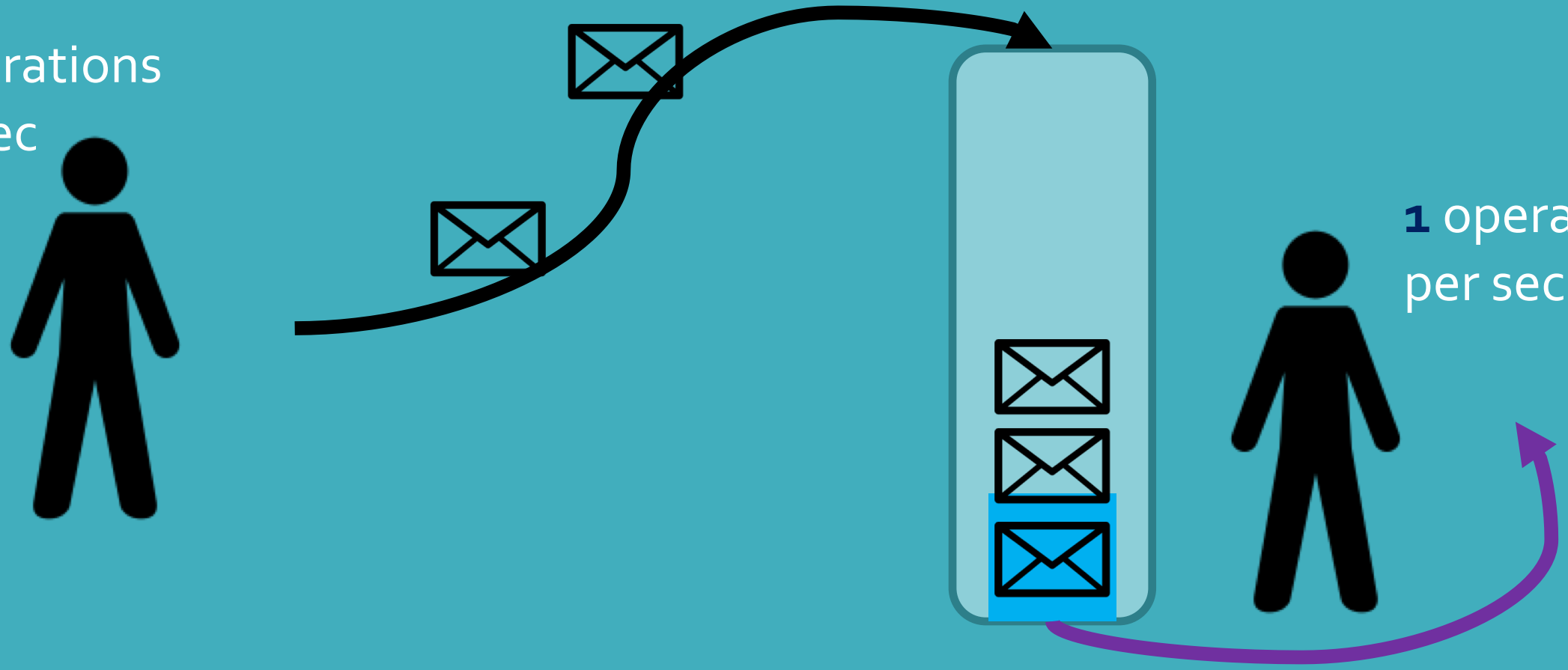
```
let merge = b.Add(new MergePreferred<Uri>(1))
let bcast = b.Add(new Broadcast<Uri>(2))

let flow =
    Flow.empty<Uri, _>
    |> Flow.asyncMapUnordered 4 downloadPageAsync
    |> Flow.collect resolveLinks

let flowBack =
    Flow.empty<Uri, _>
    |> Flow.conflateSeeded (fun (uris : ImmutableList<Uri>) uri -> uris.Add uri)
    |> Flow.collect id

b.From(bcast).Via(flowBack).To(merge) |> ignore

b.From(merge).Via(flow).To(bcast)
```

Push only model & Back-Pressure

Push Model

Subscriber usually has some kind of buffer.

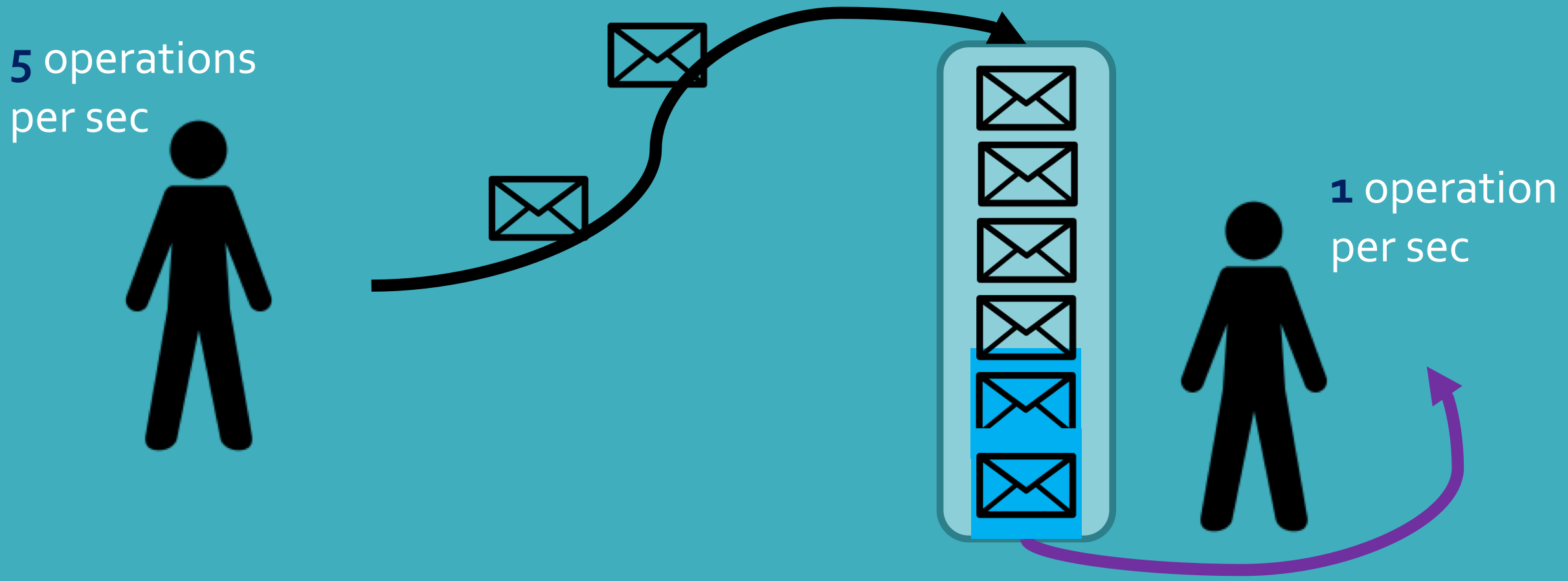
5 operations per sec



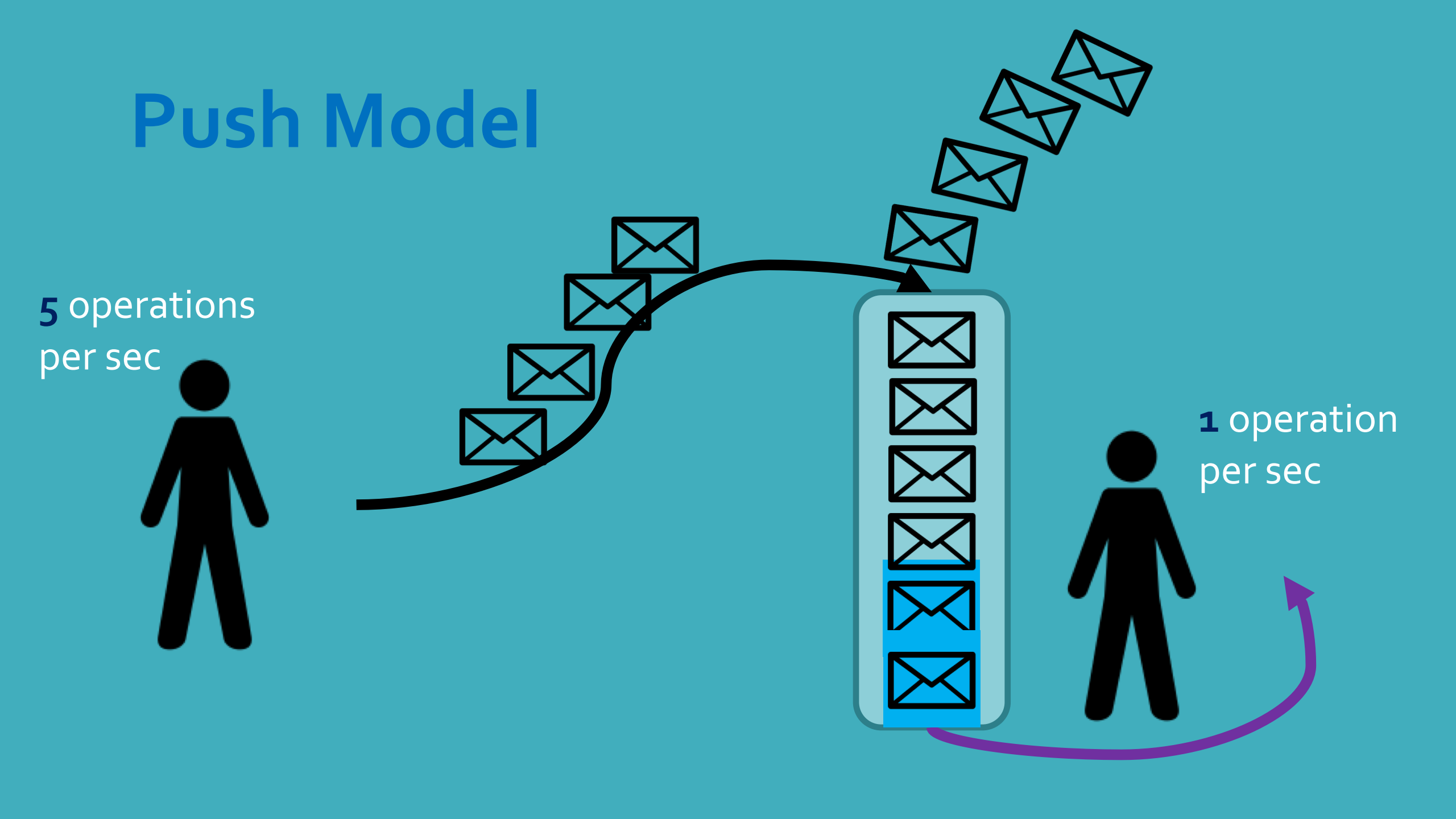
1 operation per sec

Push Model

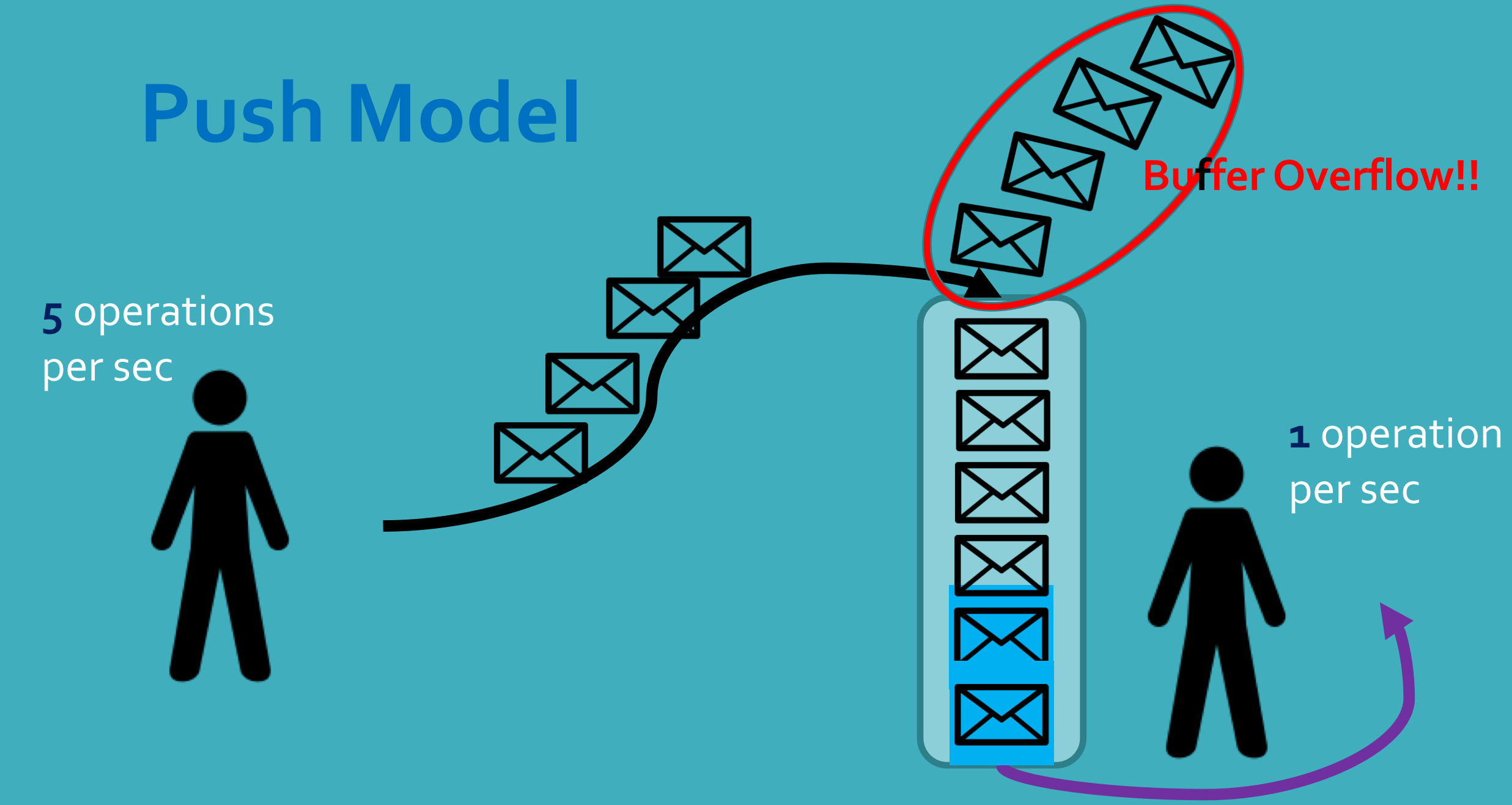
Subscriber usually has some kind of buffer.



Push Model

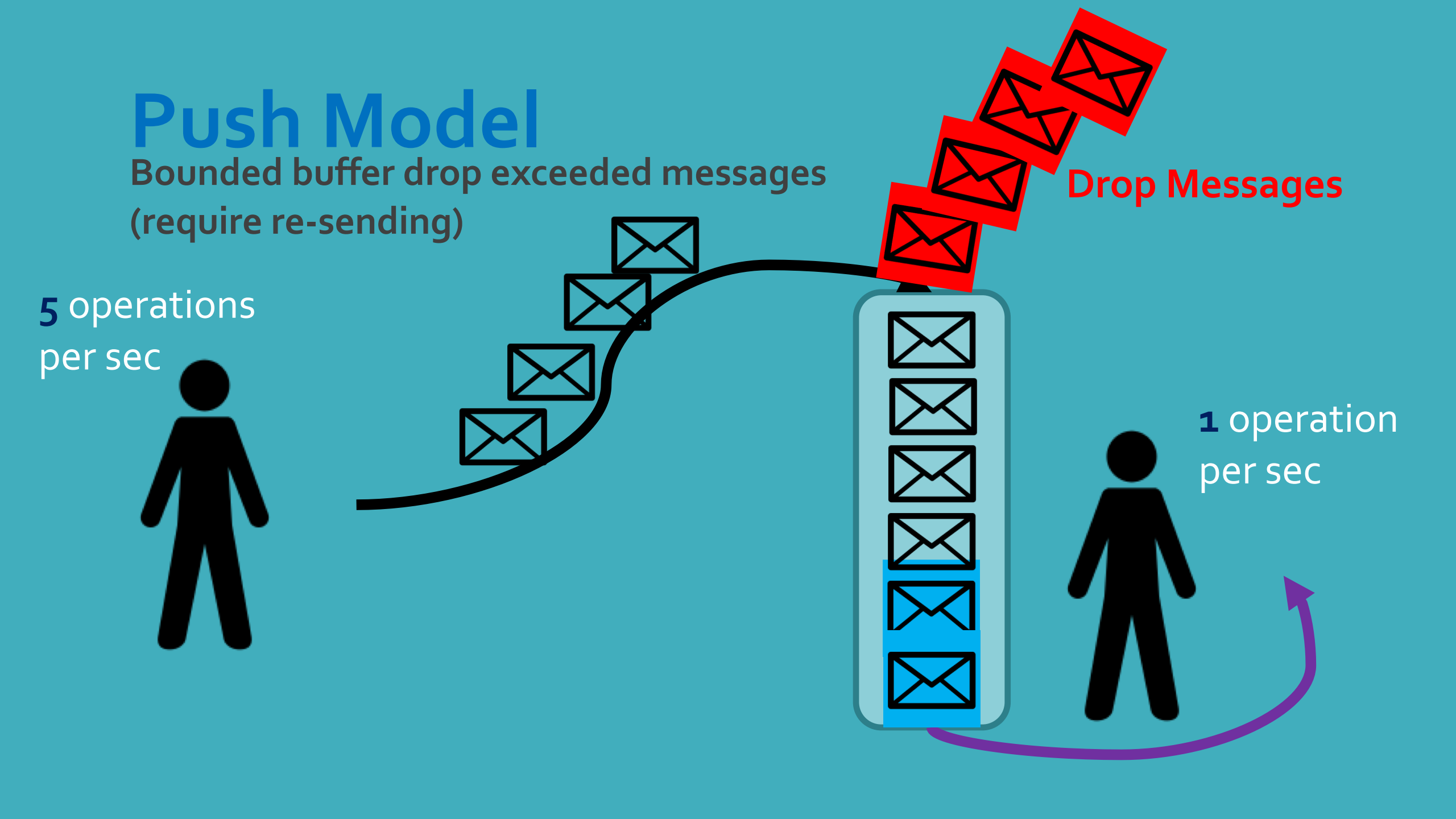


Push Model



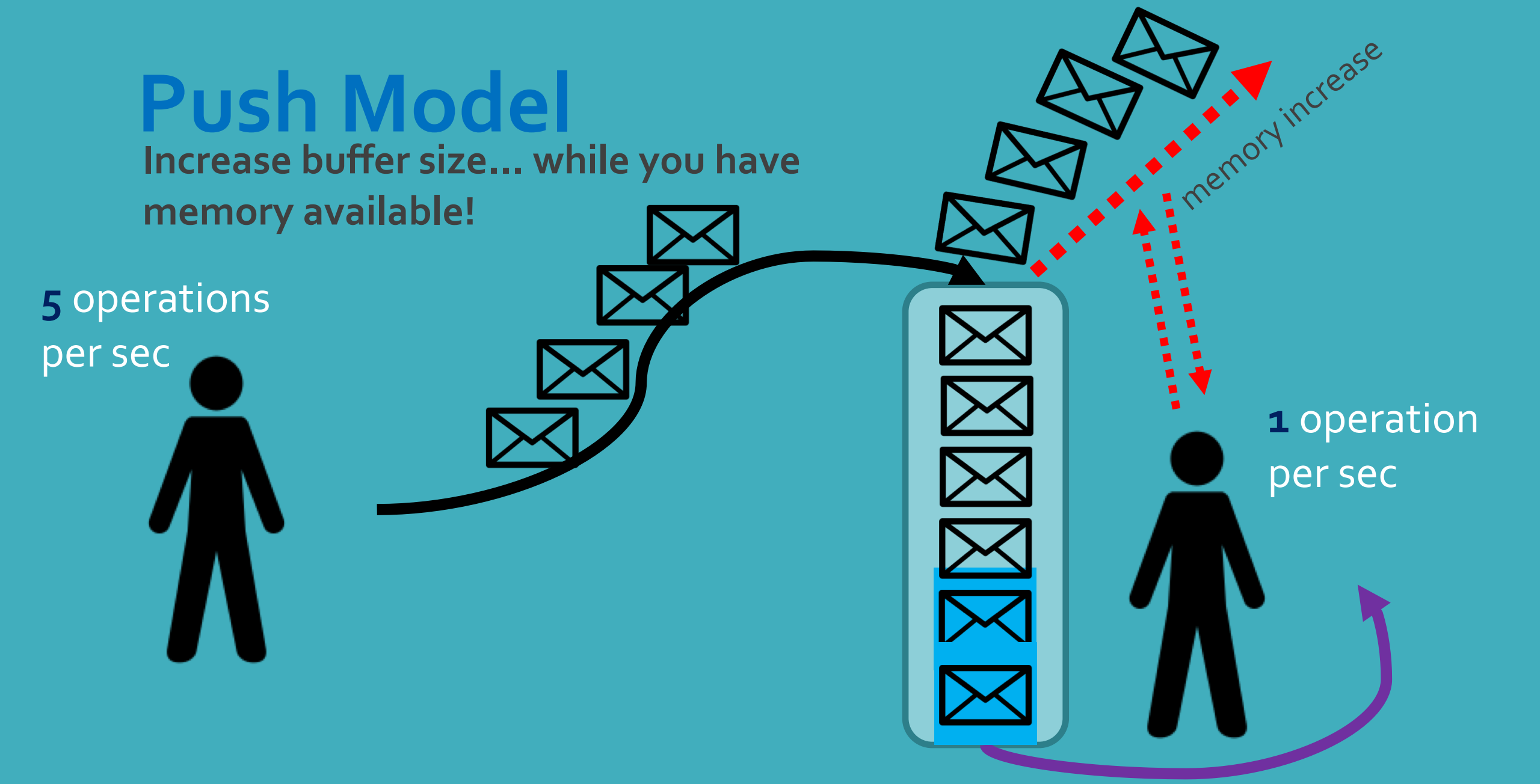
Push Model

Bounded buffer drop exceeded messages (require re-sending)

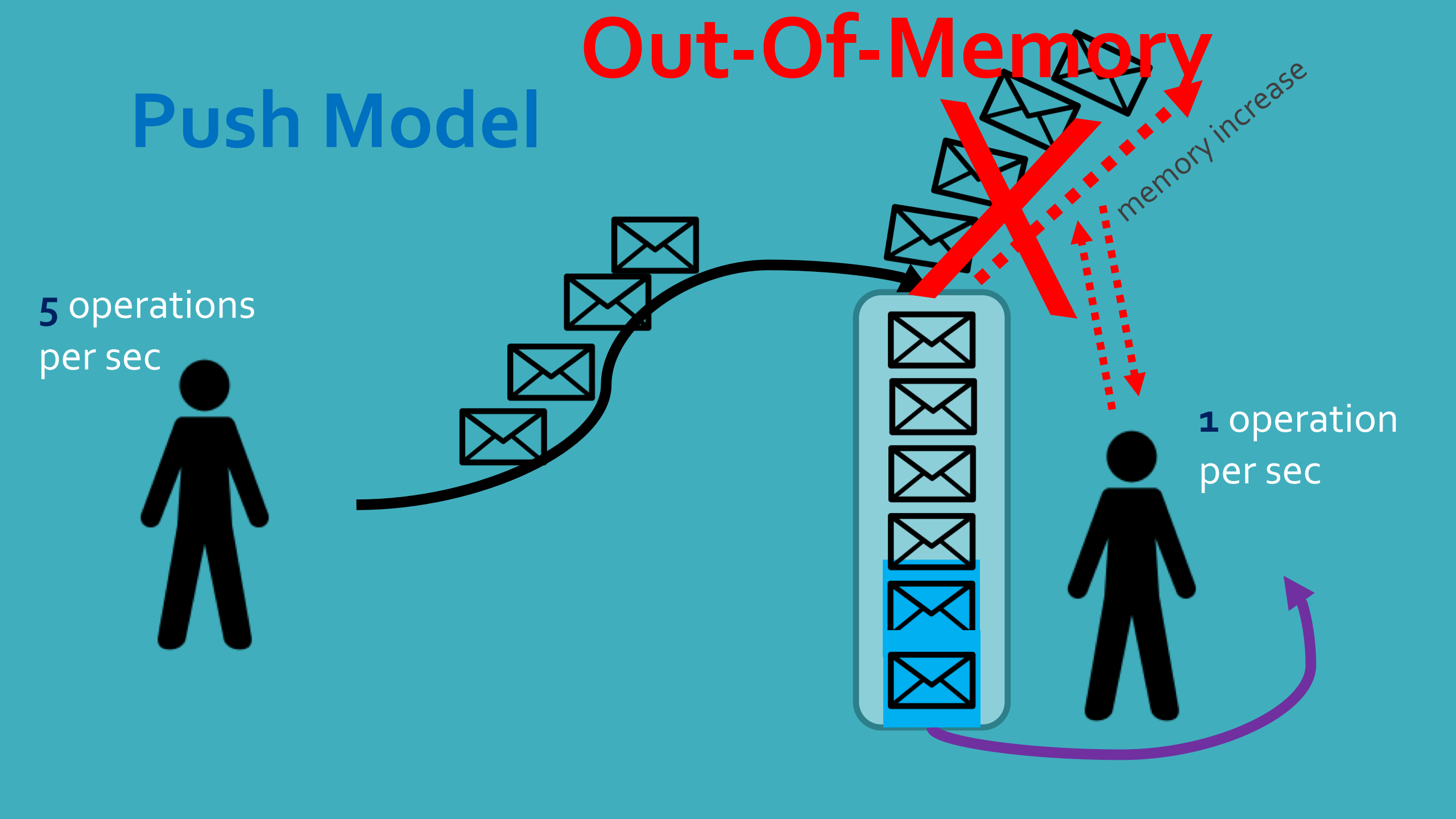


Push Model

Increase buffer size… while you have memory available!

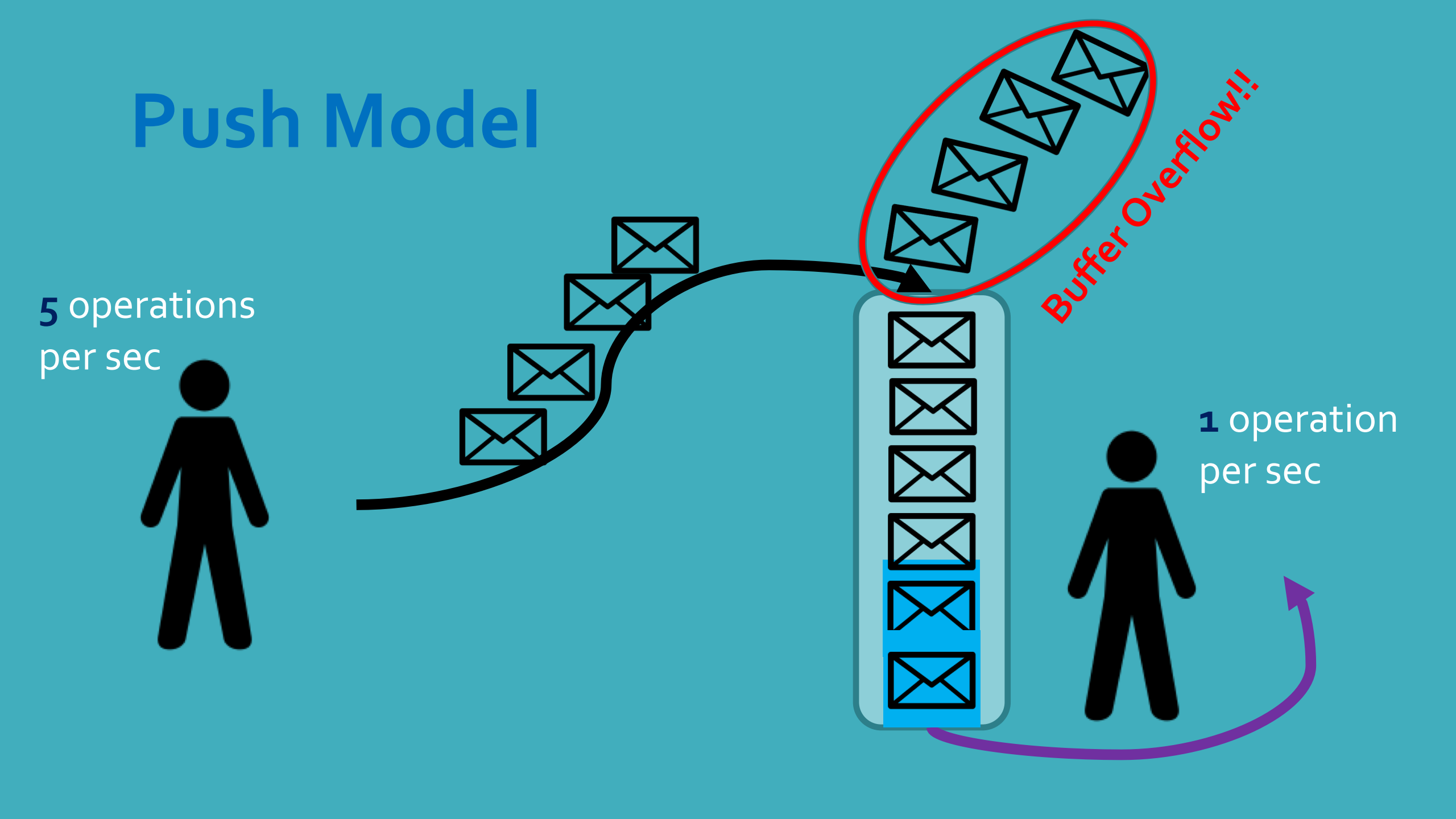


Push Model


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

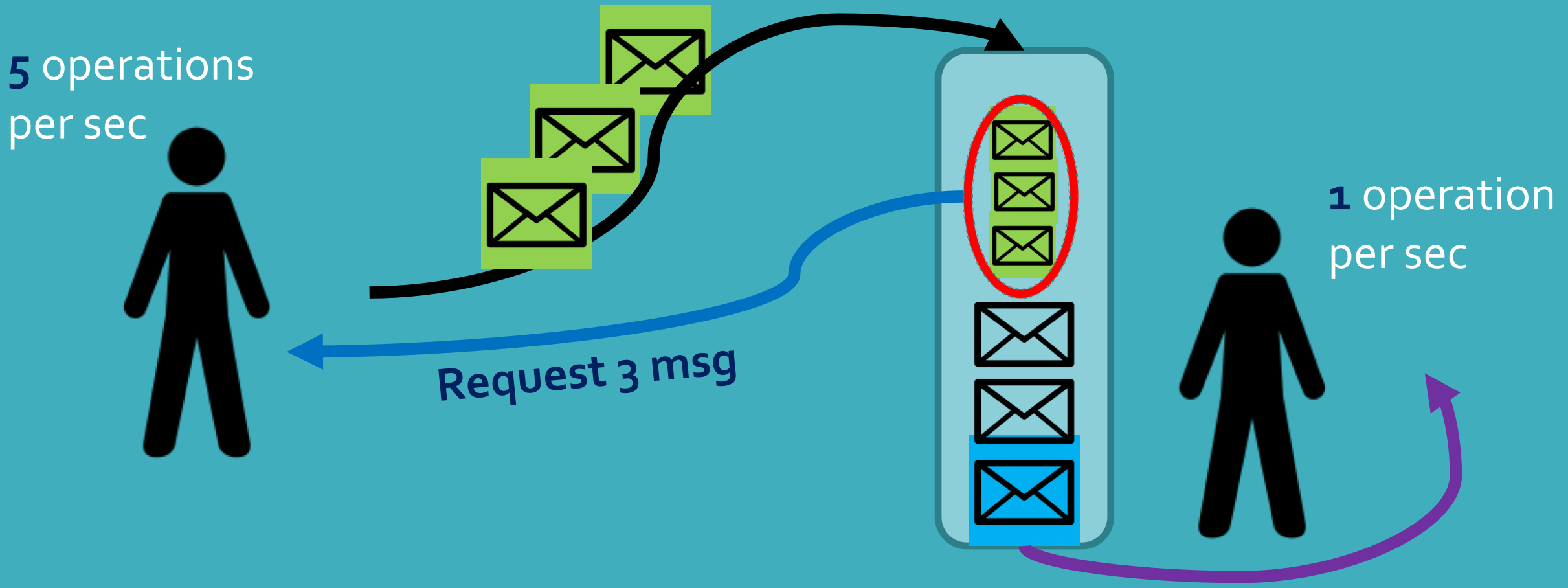
How to buffer the data without running out of memory?

Push Model

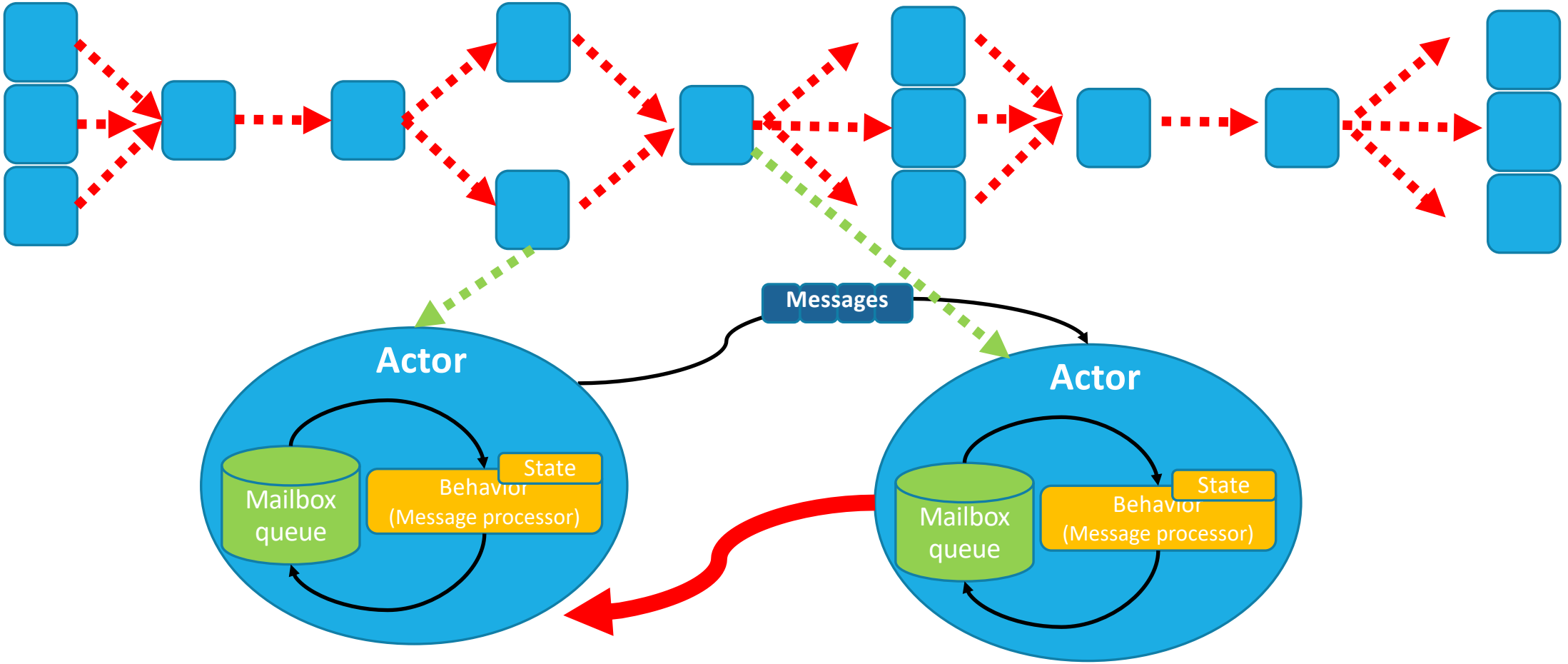


Dynamic Push-Pull

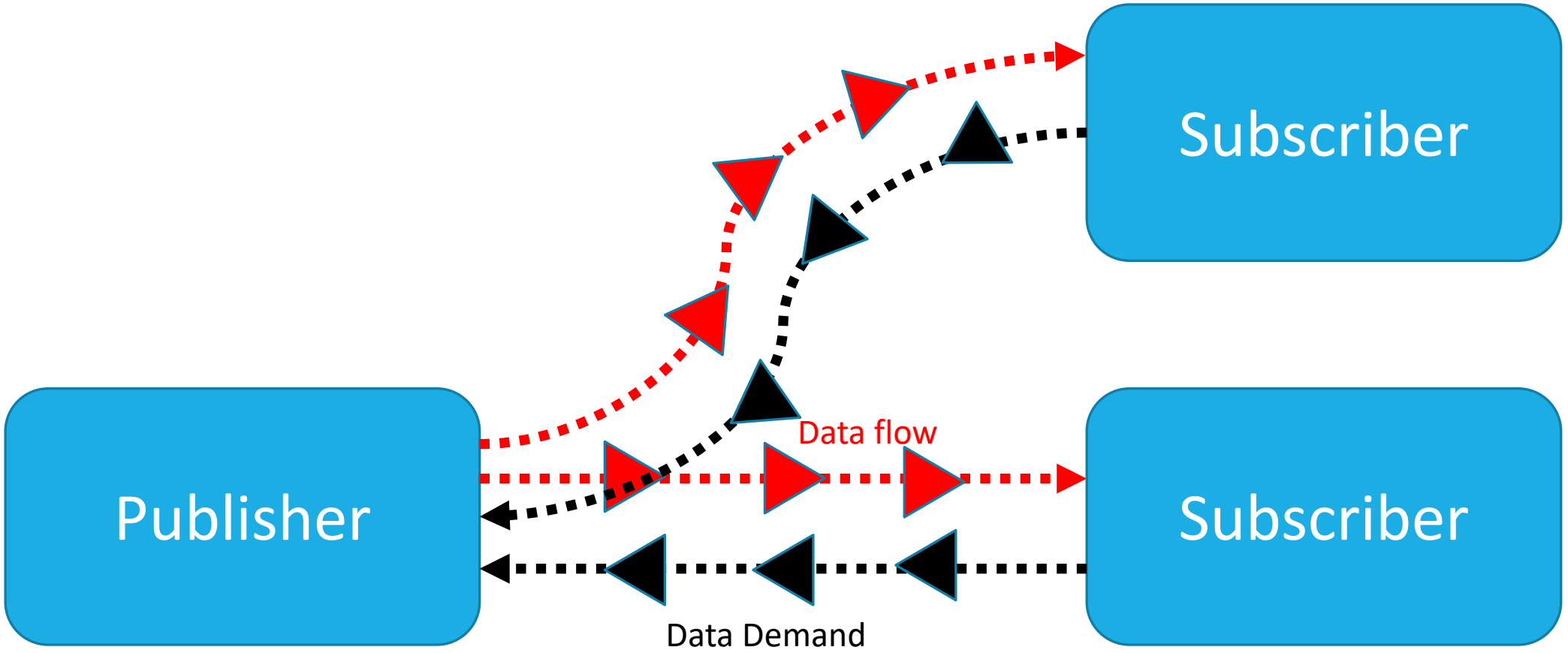
Pull based backpressure



Message Passing & Stream Modeling



Splitting and merging the data flow



Reactive Stream interfaces

```
[<Interface>]
type ISubscription =
    abstract Request : int64 -> unit
    abstract Cancel : unit -> unit

[<Interface>]
type ISubscriber<'a> =
    abstract OnSubscribe : ISubscription -> unit
    abstract OnNext : 'a -> unit
    abstract OnError : Exception -> unit
    abstract OnComplete : unit -> unit

[<Interface>]
type IPublisher<'a> =
    abstract Subscribe : ISubscriber<'a> -> unit
```

Akka Streams

- Streams complement Actors, they do not replace them
- Modular and composable
- Leveraging Actors distribution and supervision
- Easily concurrent and parallel
- Back pressure handling

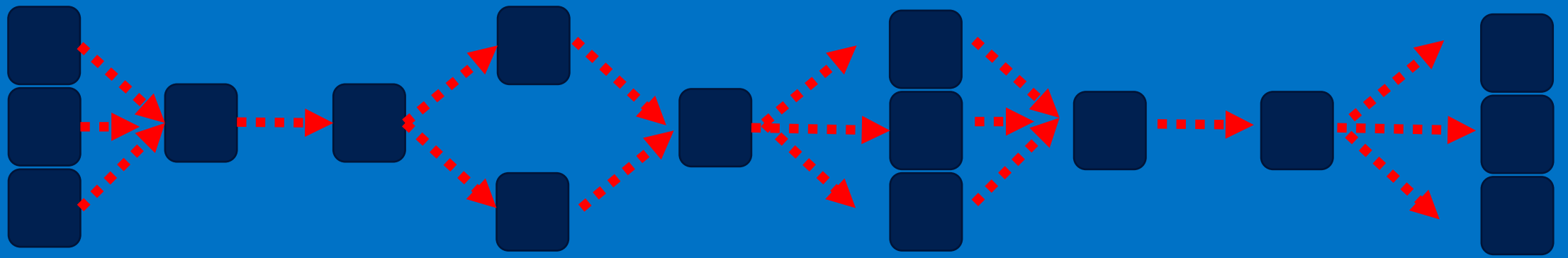
Materializing a stream

```
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

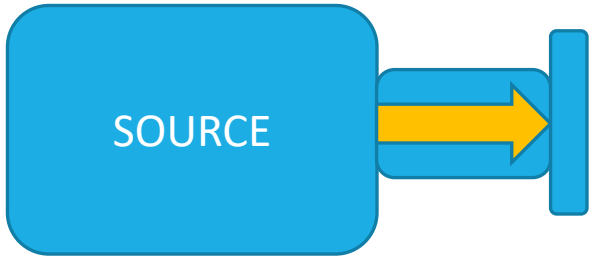
```
var runnable =
    Source
        .From(Enumerable.Range(1, 1000))
        .Via(Flow.Create<int>().Select(x => x * 2)
        .To(Sink.ForEach<int>(x => Console.Write(x.ToString));

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

Reactive Stream compose



Elements of a Stream

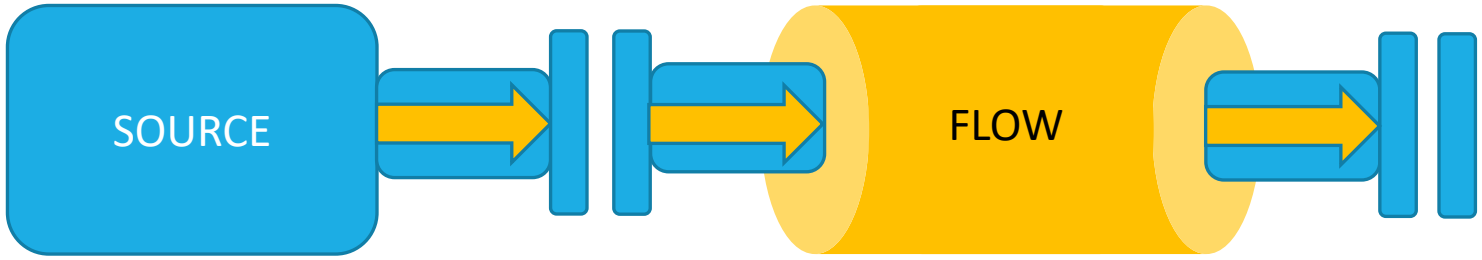
Source : This is where the pipeline starts. A Source takes data from input and has a single output to be written into.



Elements of a Stream

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

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

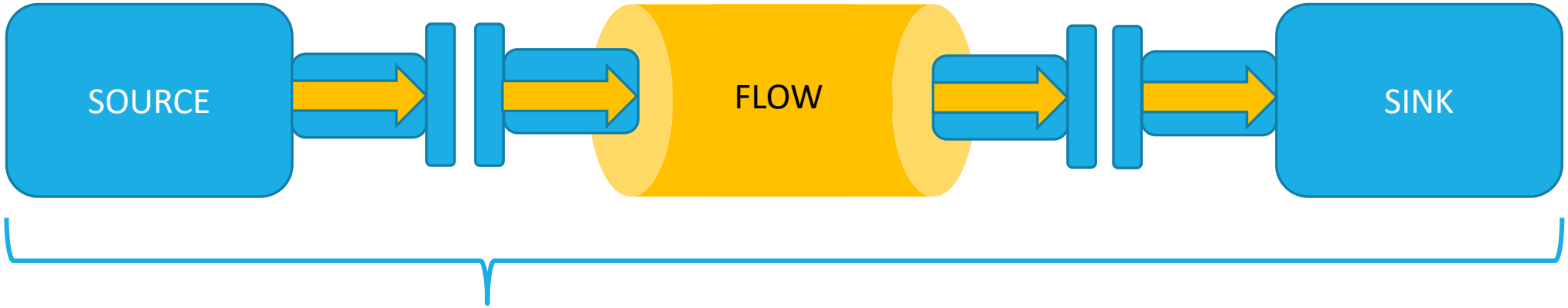


Elements of a Stream

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

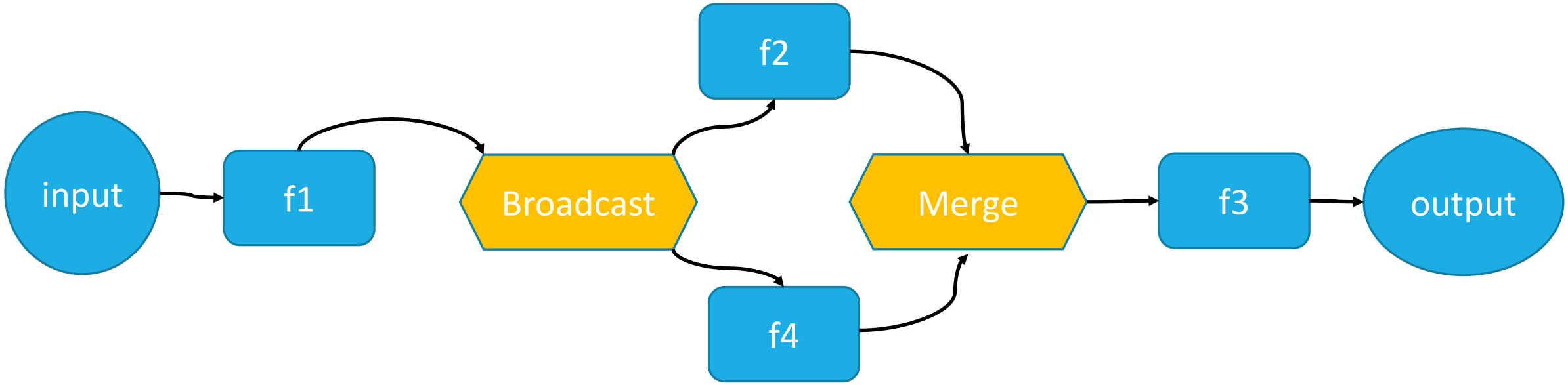
Flow : This is the basic abstraction where transformation of data takes place. A Flow has one input and one output.

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

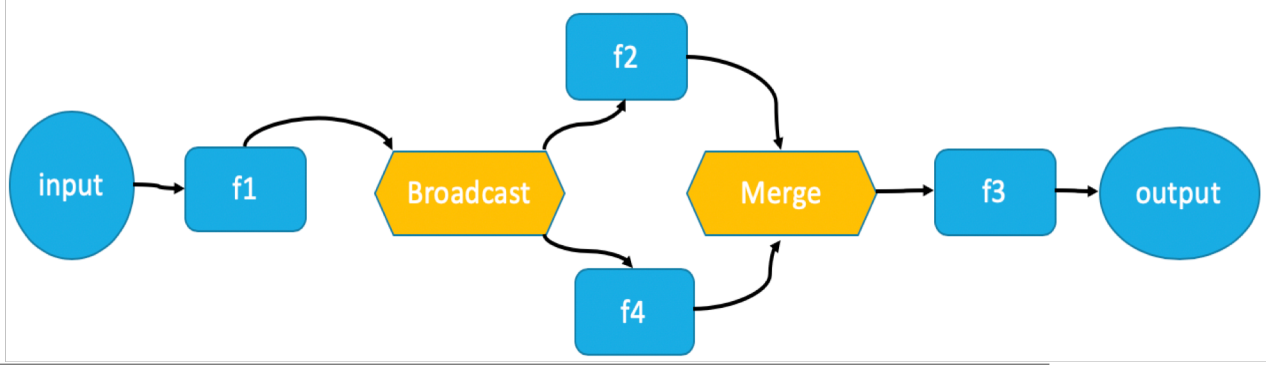


RunnableGraph represents the entire topology (Source -> Flow* -> Sink)

GraphDSL



Graph DSL in F#

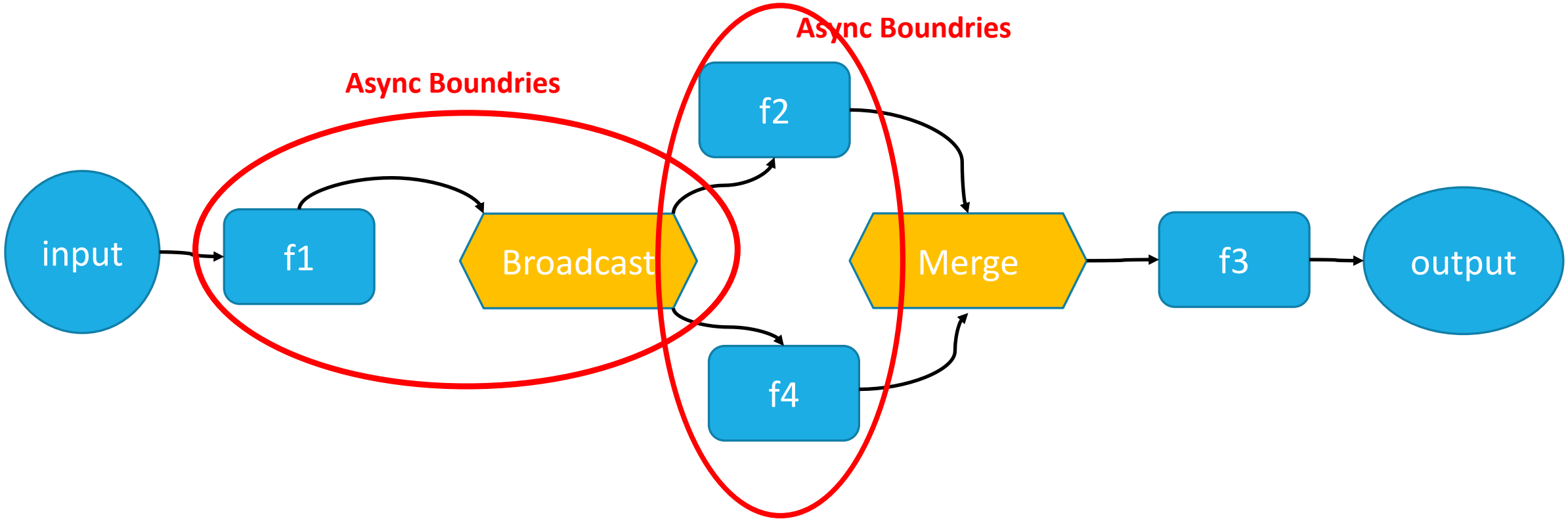



```
let graph = Graph.create(fun builder ->
    let source = Source.from(Enumerable.Range(1, 10))
    let sink = Sink.Ignore<int>().MapMaterializedValue(fun _ -> NotUsed.Instance)

let broadcast = builder.Add(new Broadcast<int>(2))
    let merge = builder.Add(new Merge<int>(2))

    let f1 = Flow.Create<int>().Select(fun x -> x + 10)
    let f2 = Flow.Create<int>().Select(fun x -> x + 10)
    let f3 = Flow.Create<int>().Select(fun x -> x + 10)
    let f4 = Flow.Create<int>().Select(fun x -> x + 10)

    builder.From(source).Via(f1).Via(broadcast).Via(f2).Via(merge).Via(f3).To(sink)
    builder.From(broadcast).Via(f4).To(merge)
    ClosedShape.Instance )
```

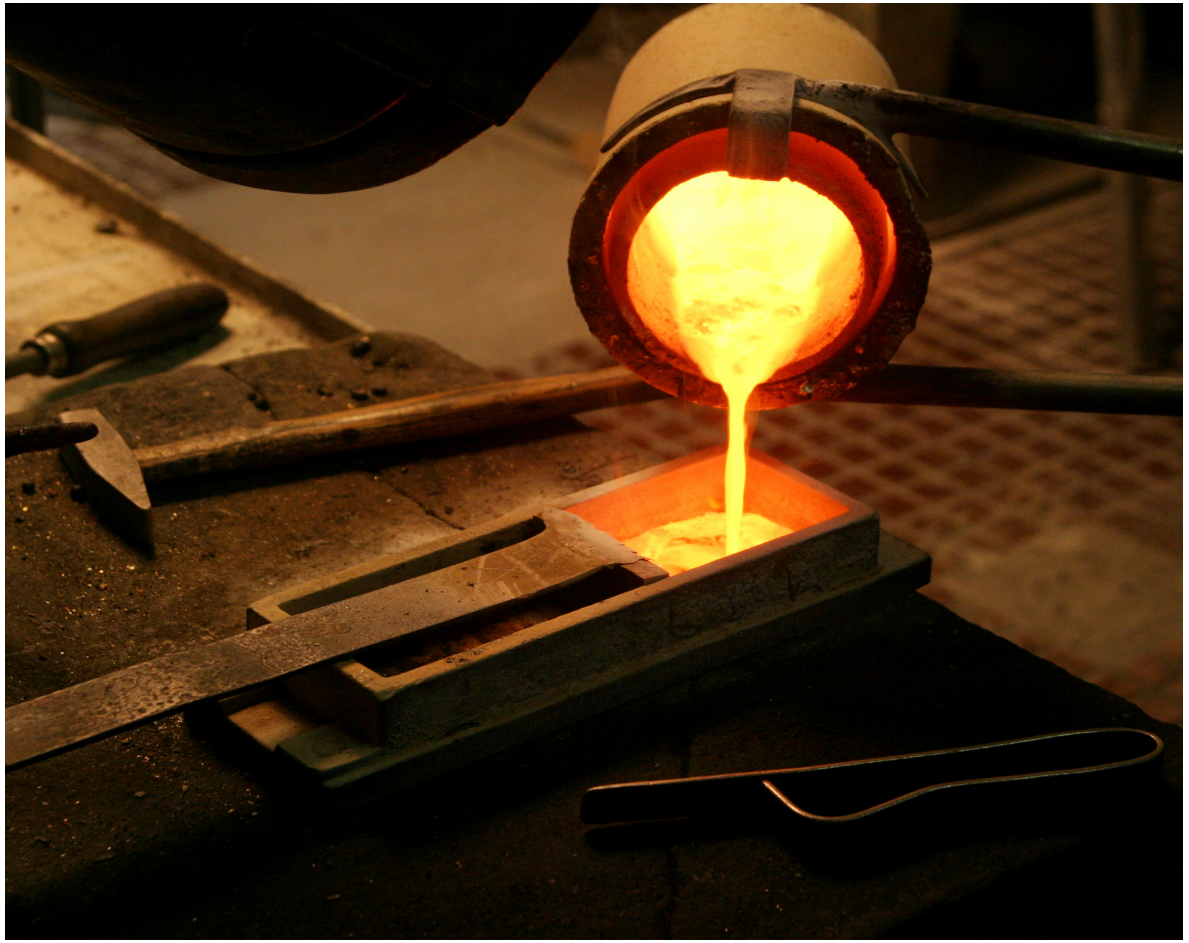
Implementing High performant Stream-Processing with Reactive Stream

Throughput Throughput Throughput

Async Boundaries



Stream – Operator fusion



- Fast elements passing elements processing stage to the next
- Avoid asynchronous messaging overhead
- Explicit parallel processing across boundaries

Async Boundaries

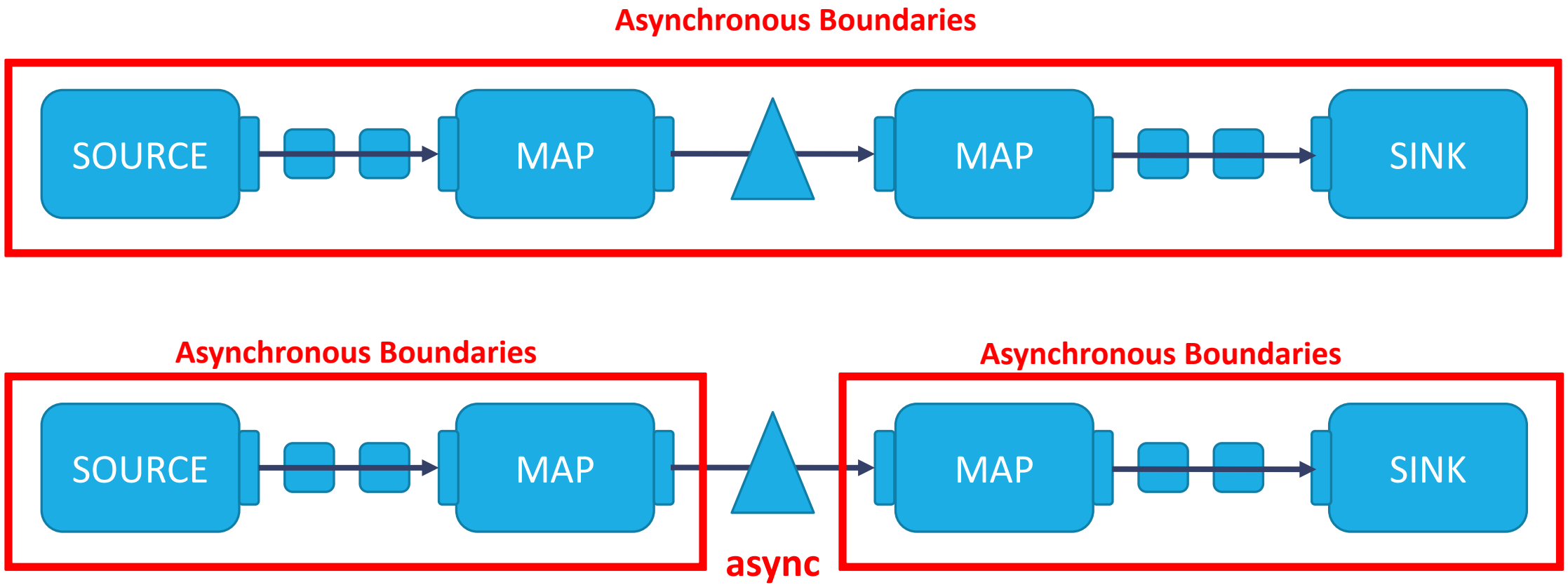
```
let spin (value: int) =
    let start = DateTime.Now.Ticks
    while DateTime.Now.Ticks - start < int64 value do
        Thread.Yield() |> ignore
    value

let count = 3000
```

```
let runSync() =
    Source.ofList [1..count]
    |> Source.map(spin)
    |> Source.map(spin)
    |> Source.runWith materializer Sink.ignore
```

Async Boundaries

```
let runAsync() =
    Source.ofList [1..count]
    |> Source.map(spin)
    |> Source.async
    |> Source.map(spin)
    |> Source.async
    |> Source.runWith materializer Sink.ignore
```

Async Boundaries



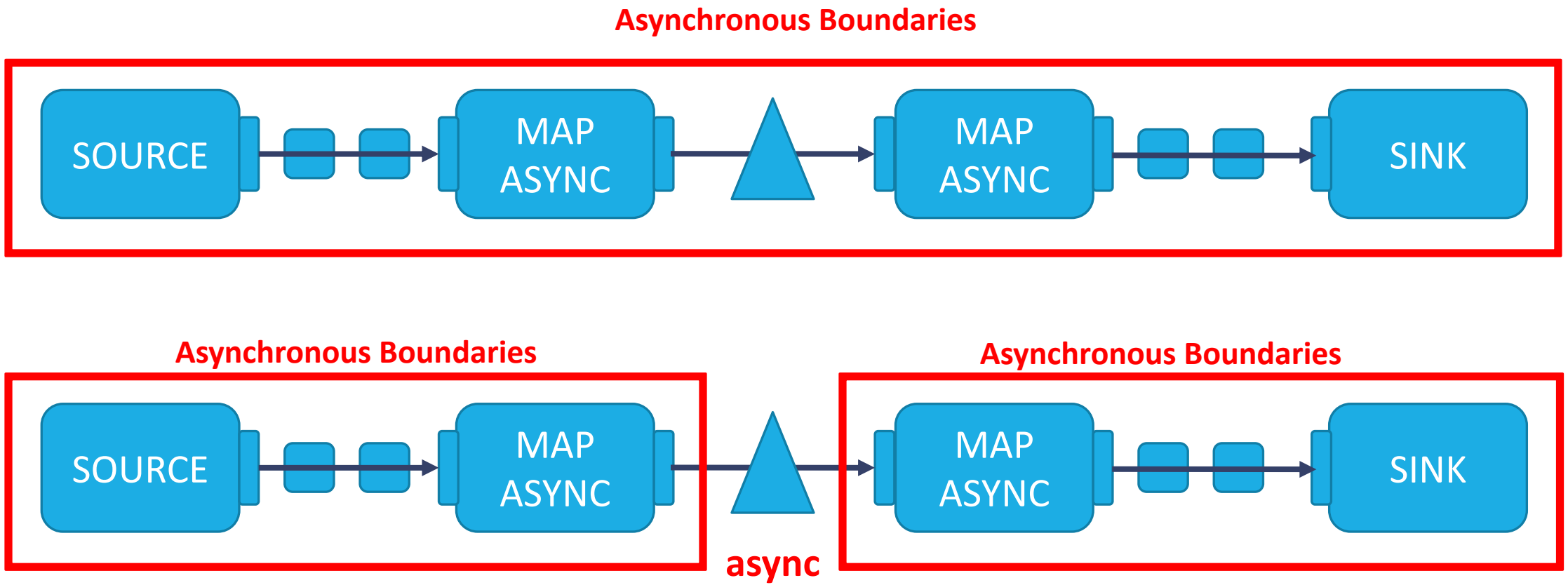
Async Boundaries & Concurrency

```
let spawn f x = async { return! f x }

let runParallelAsync() =
        Source.ofList [1..count]
        |> Source.asyncMap 1 (fun i -> spawn spin i)
        |> Source.asyncMap 1 (fun i -> spawn spin i)
        |> Source.runWith materializer Sink.ignore
```

Async Boundaries & Concurrency

```
let runAsyncParallel () =
        let n = 4
        Source.ofList [1..count]
        |> Source.asyncMap n (fun i -> spawn spin i)
        |> Source.asyncMap n (fun i -> spawn spin i)
        |> Source.runWith materializer Sink.ignore
```

Async Boundaries & Concurrency understanding AsyncMap



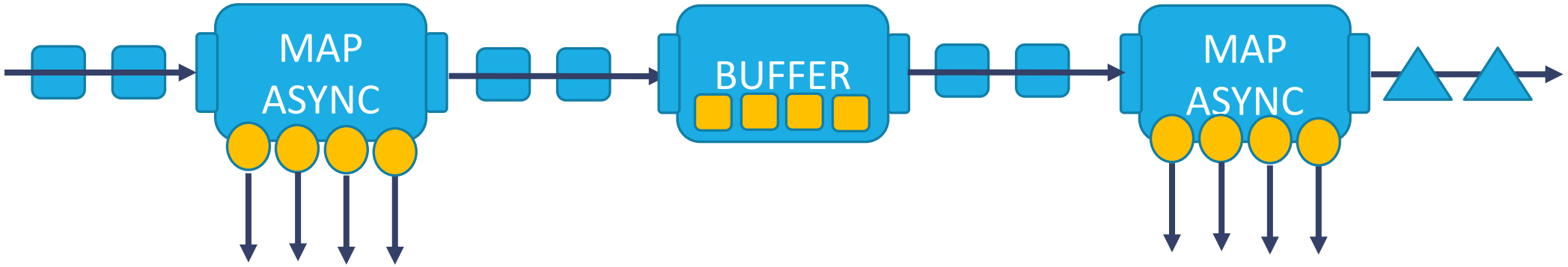
Async Boundaries & Concurrency

```
let runAsyncParallel () =
        let n = 4
        Source.ofList [1..count]
        |> Source.asyncMap n (fun i -> spawn spin i)
        |> Source.asyncMap n (fun i -> spawn spin i)
        |> Source.runWith materializer Sink.ignore
```

Async Boundaries & Concurrency understanding AsyncMap

```
let runAsyncParallel () =
        let n = 4
        Source.ofList [1..count]
        |> Source.asyncMap n (fun i -> spawn spin i)
        |> Source.async
        |> Source.asyncMap n (fun i -> spawn spin i)
        |> Source.async
        |> Source.runWith materializer Sink.ignore
```

Async Boundaries & Concurrency understanding AsyncMap



Buffer for improving performance

```
let runAsyncParallelBuffer () =
    let n = 4
    Source.ofList [1..count]
    |> Source.asyncMap n (fun i -> spawn spin i)
    |> Source.buffer 16 OverflowStrategy.Backpressure
    |> Source.asyncMap n (fun i -> spawn spin i)
    |> Source.buffer 16 OverflowStrategy.Backpressure
    |> Source.runWith materializer Sink.ignore
```

Batching Stream Events

```
let updateSink =
    Sink.ForEach<ITweet * MarkerLocation list>(fun (tweet, emotions) ->
    agentUpdate.Post emotions // update UI
    )

tweetSource
|> Source.map sentimentAnalysis
|> Source.map coordinateFlow
|> Source.grouped 100
// |> Source.groupedWithin 100 TimeSpan.FromSeconds(15.)
|> Source.runWith updateSink
```

Throttling

```
tweetSource
|> Source.map sentimentAnalysis
|> Source.map coordinateFlow
|> Source.grouped 100
|> Source.throttle ThrottleMode.Shaping 10  TimeSpan.FromSeconds(1.)
|> Source.runWith updateSink
```

Summary

The Benefits (And Limitations) Of Reactive Programming

- Reactive Streams can be easily composed to construct sophisticated, reliable, and resilient systems
- Akka Streams are powerful and natural for constructing streaming-data systems processing
- To perform tasks concurrently in Reactive Stream use multi-threaded patterns such as Batching, Asynchronous boundaries and Throttling

Contacts

Twitter	*@trikace*
Blog	*www.rickyterrell.com*
Email	*tericcardo@gmail.com*
GitHub	*www.github.com/rikace/presentation*