Message Passing Concurrency

BY RICCARDO TERRELL - @TRIKACE

Agent motivations

- You can manage **shared data** and resources **without locks**.
- You can easily follow the **SRP**, because each **agent** can be designed to **do only one thing**.
- It encourages a "pipeline" model of programming with "producers" sending messages to decoupled "consumers“
- It is straightforward to **scale**
- Errors can be handled gracefully, because the decoupling means that agents can be created and destroyed without affecting their clients.

Agent-based concurrency

Programs **composed** from agents

- Agents can be viewed as “running” objects

Agents **exchange** messages

- Receive message and react
- Trigger event when work is done

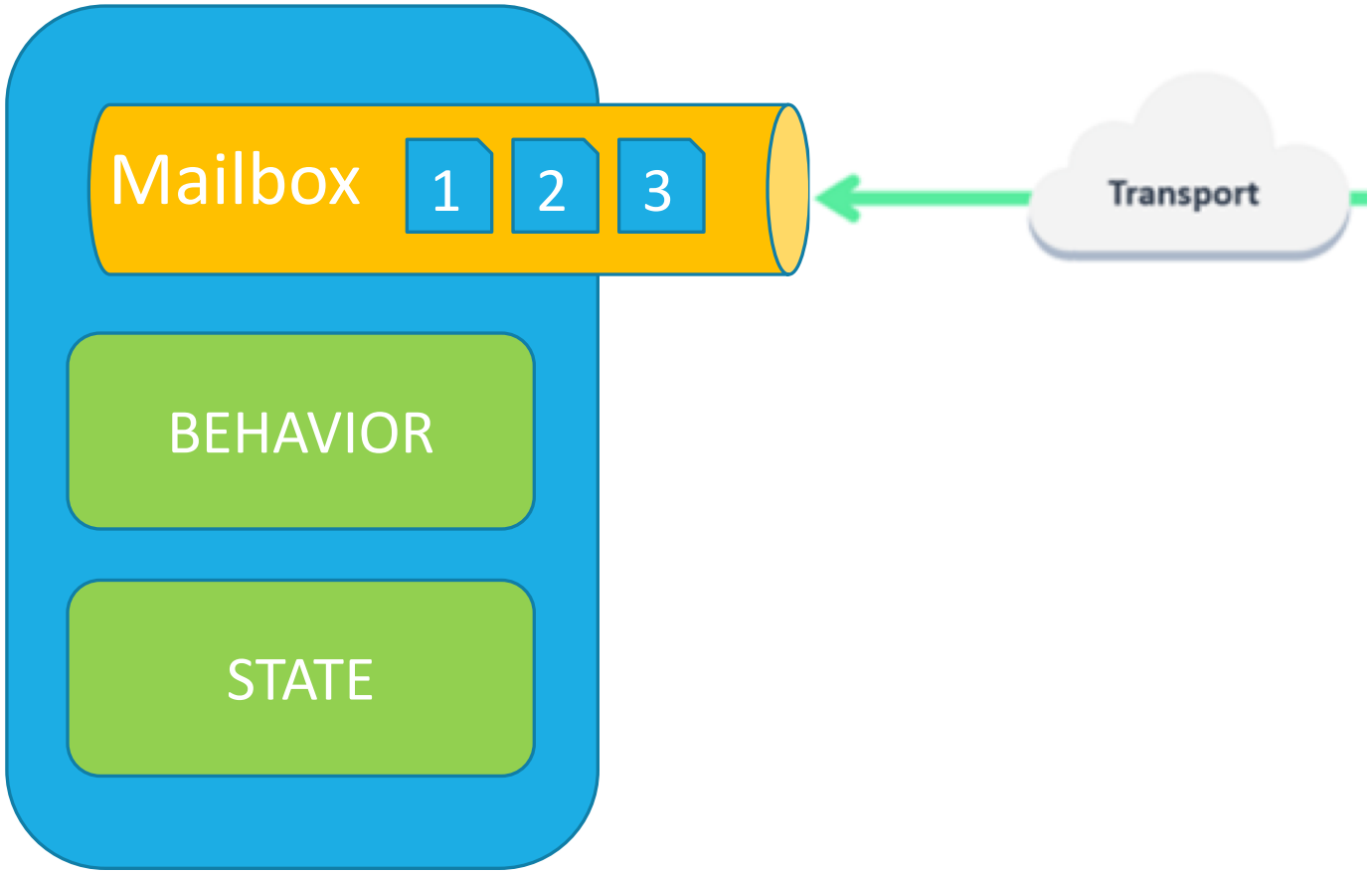
Reactive system

- Handle inputs while running
- Emit results while running

Agent is a single and independent unit of computation

Message-Driven

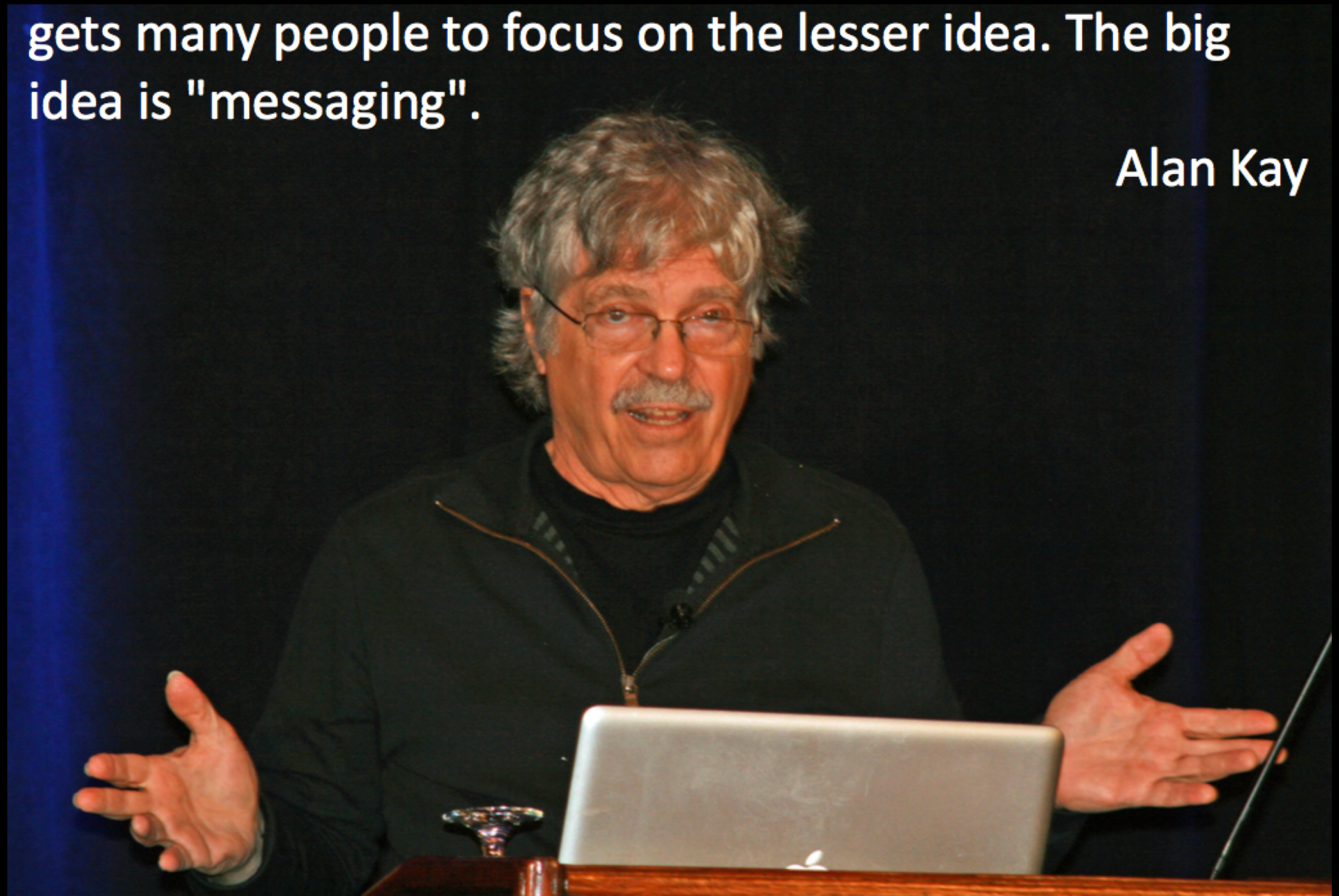
Message Passing based concurrency



- Processing
- Storage – State
- Communication only by messages
- Share Nothing
- Message are passed by value
- Lightweight object
- Running on it's own thread
- No shared state
- Messages are kept in mailbox and processed in order
- Massively scalable and lightening fast because of the small call stack

I'm sorry that I coined the term "objects", because it gets many people to focus on the lesser idea. The big idea is "messaging".

Alan Kay



Simple agent in F#

Receive message and say “Hello”

```
let hello = Agent.Start(fun agent -> async {
  while true do
    let! name = agent.Receive()
    printfn "Hello %s"  name
    do! Async.Sleep 500 })

hello.Post(“World!")
```

- Single instance of the body is running
- Waiting for message is asynchronous
- Messages are queued by the agent

Mutable and immutable state

Mutable state

- Accessed from the body
- Used in loops or recursion
- Mutable variables (ref)
- Fast mutable collections

Immutable state

- Passed as an argument
- Using recursion (**return!**)
- Immutable types
- Can be returned from the agent

```
Agent.Start(fun agent -> async {
  let names = ResizeArray<_>()
  while true do
    let! name = agent.Receive()
    names.Add(name) })
```

```
Agent.Start(fun agent ->
  let rec loop names = async {
    let! name = agent.Receive()
    return! loop (name::names) }
 loop [])
```

Immutability OR Isolation

F#

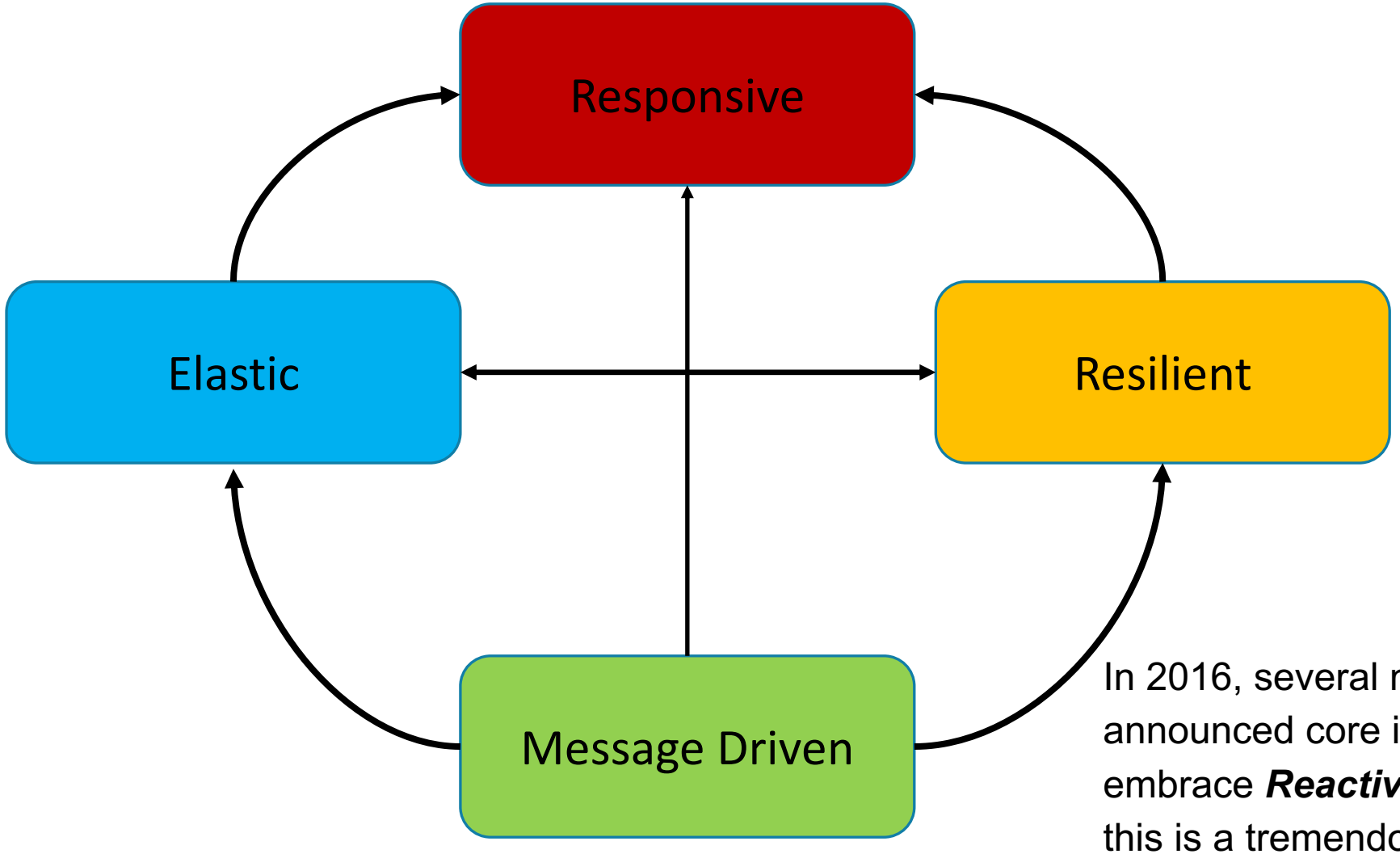
```
let printerAgent = MailboxProcessor.Start(fun inbox->
    let list = new List<string>()
    // the message processing function
    let rec messageLoop() = async{
        // read a message
        let! msg = inbox.Receive()
        // process a message
        printfn "message is: %s" msg
        // loop to top
         list.Add(msg)
        return! messageLoop()
        }
    // start the loop
    messageLoop()
    )
```

Send message to agent

```
printerAgent.Post "hello"

printerAgent.Post "hello again"

printerAgent.Post "hello a third time"
```

F#

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

Reactive Manifesto & Actor Model

Responsive	Event Driven
Message-Driven	Communication by messages
Resilient	Fault tolerant by Supervision
Elastic	Clustering and Remoting across multiple machines

Agent-based architecture

Lots of things going on!

- How to keep a big picture?

Using loosely coupled connections

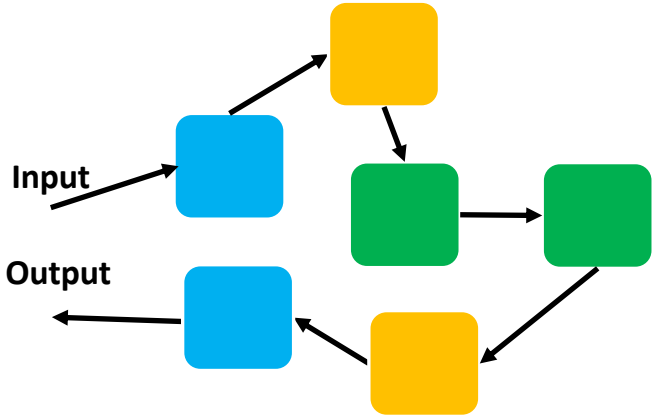
- Agents don’t reference each other directly

Common ways of organizing agents

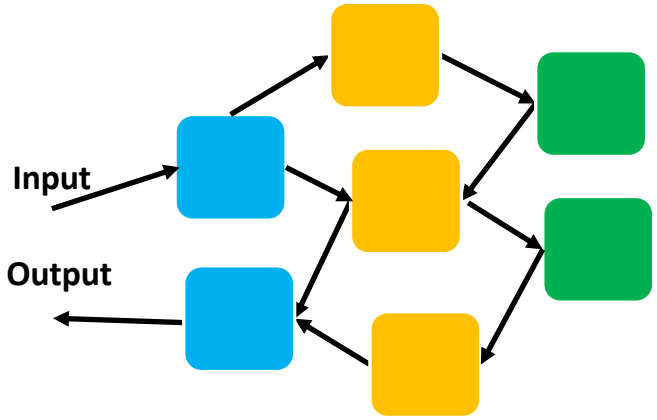
- **Worker agent** – Single agent does work in background
- **Layered network** – Agent uses agents from lower level
- **Pipeline processing** – Step-by-step processing

Comparison between Sequential, Task-based and Message passing programming

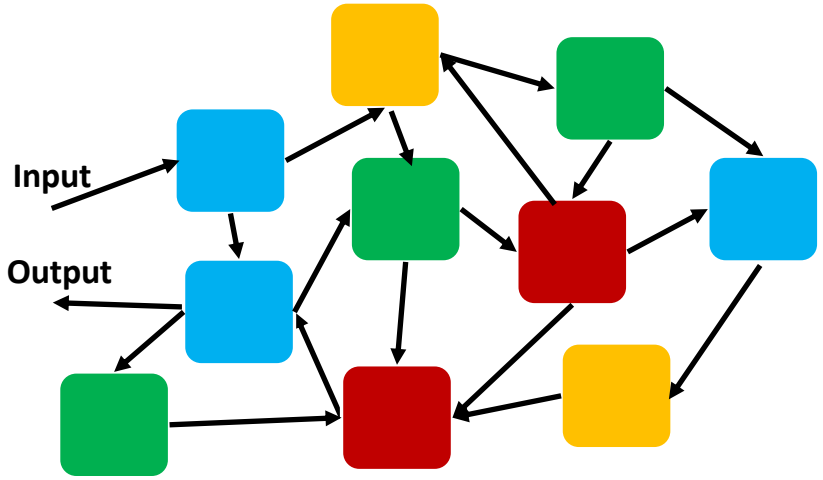
Sequential Programming



Task-Based Programming



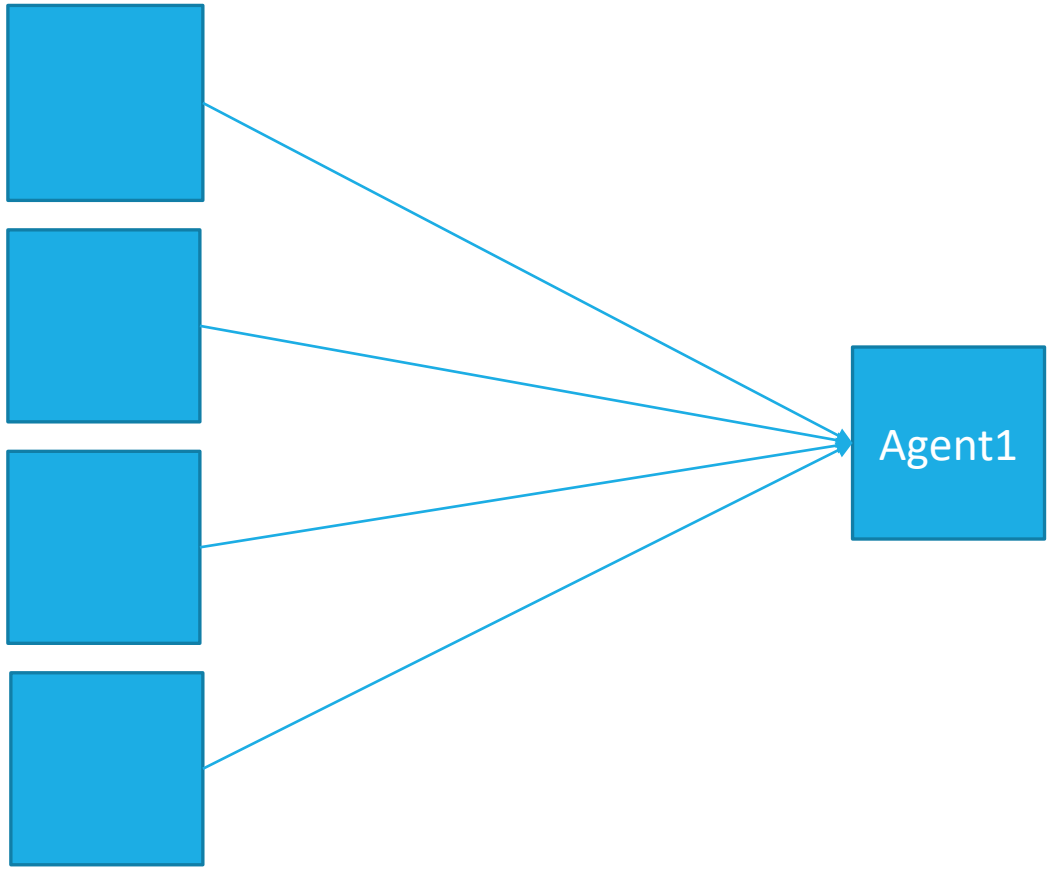
Message-Passing Programming



Agent Patterns

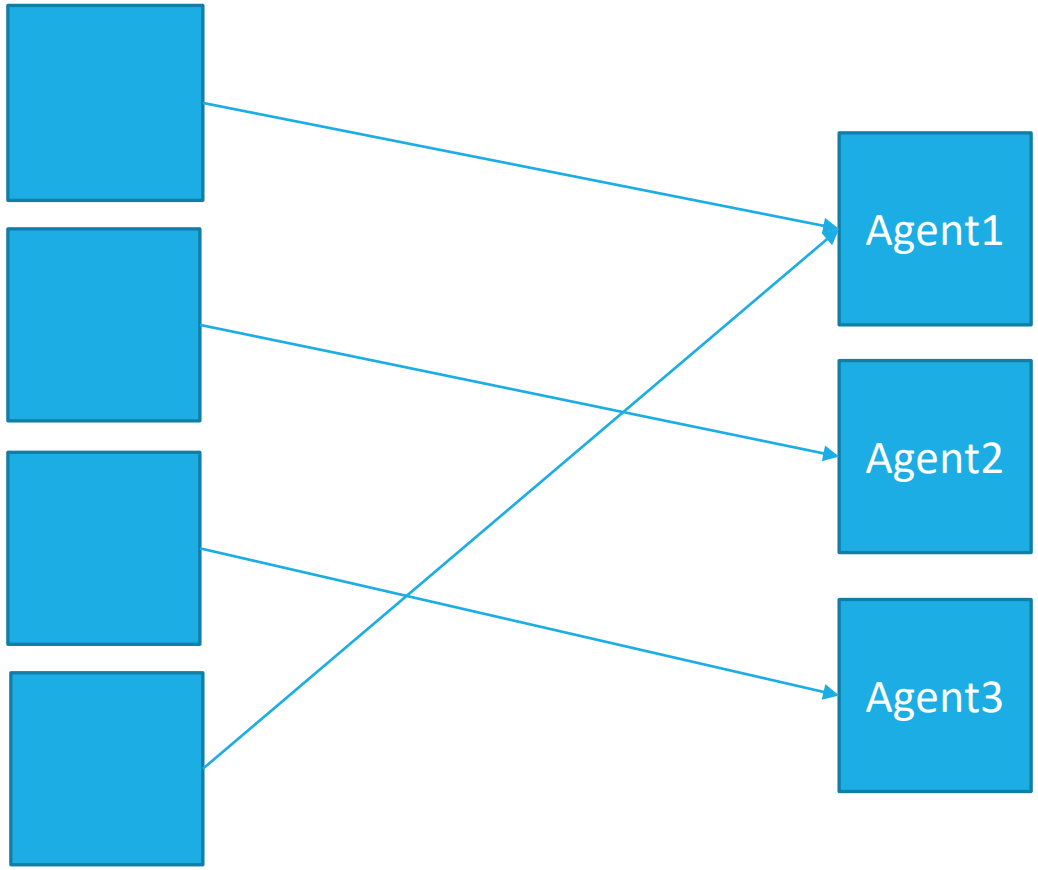
Agent patterns

Singleton



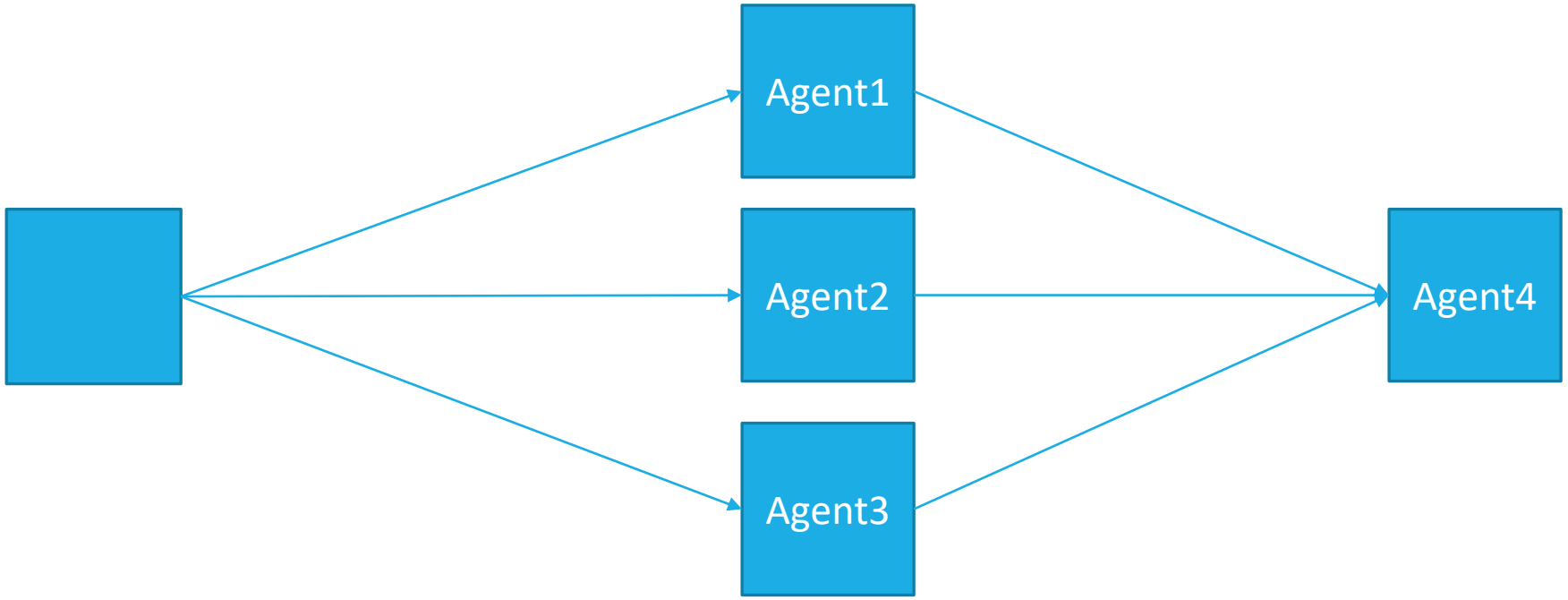
Agent patterns

Pool



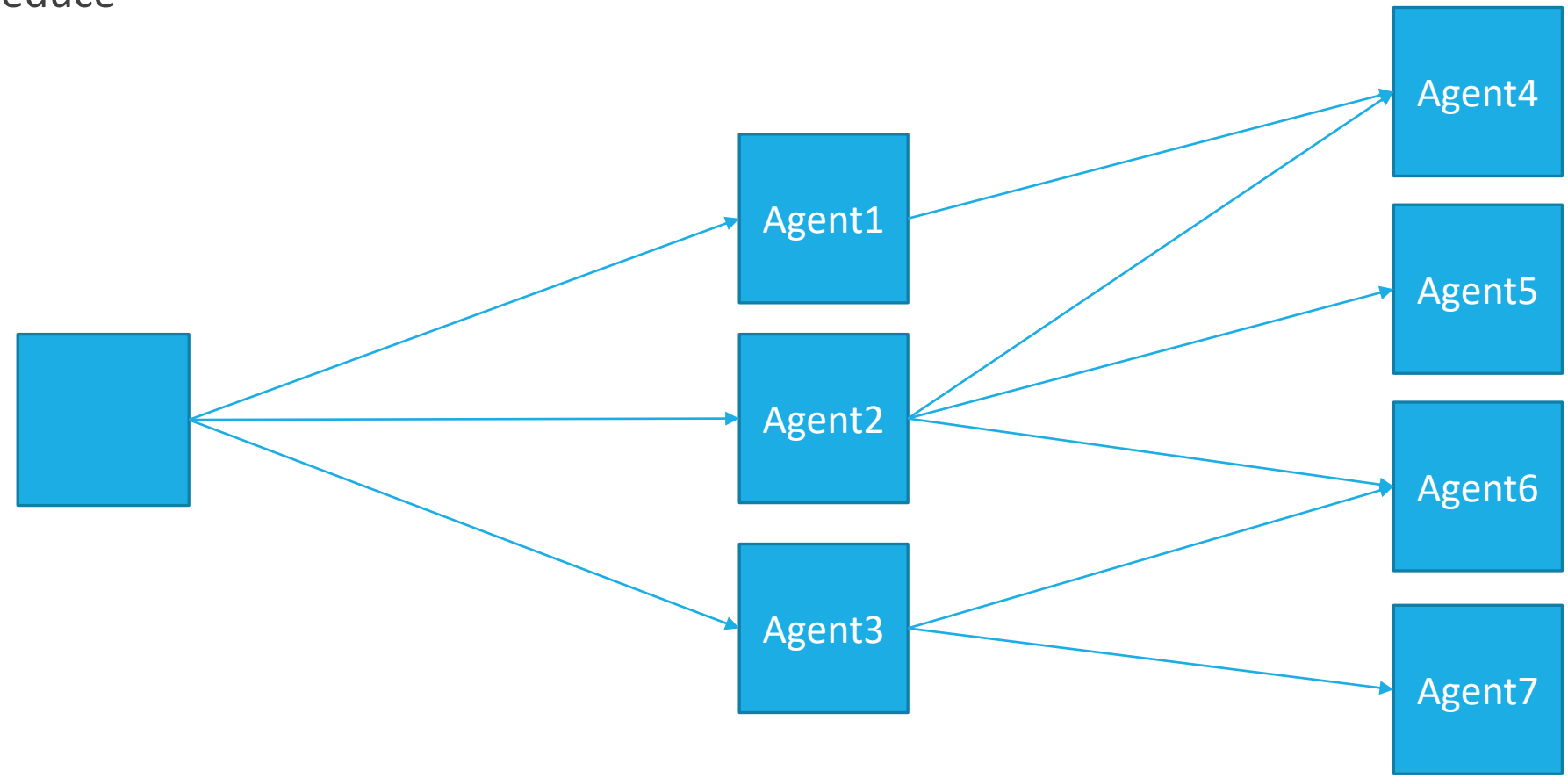
Agent patterns

Fork Join

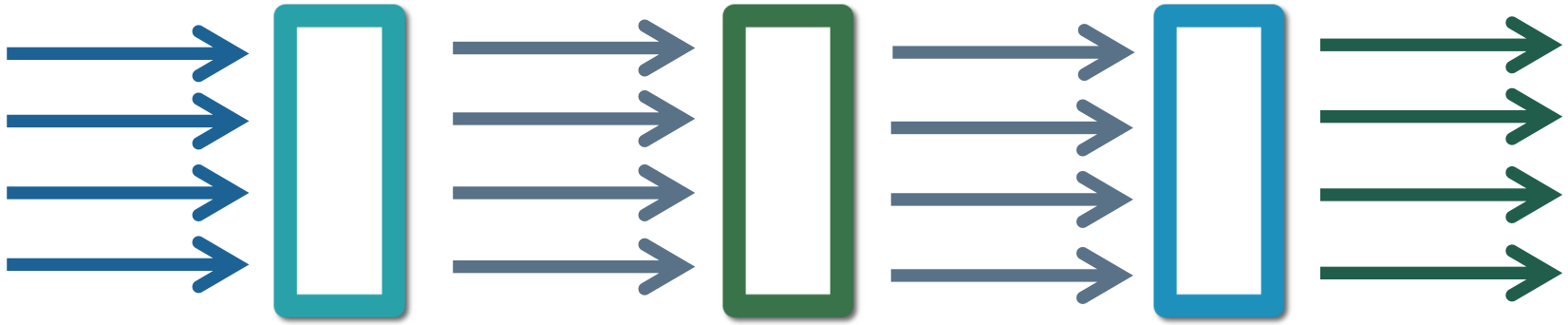


Agent patterns

Map Reduce



Pipeline Processing



- A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements

Agent Error Handling & Disposable

```
let errorAgent =
       Agent<int * System.Exception>.Start(fun inbox ->
         async { while true do
                   let! (agentId, err) = inbox.Receive()
                   printfn "an error '%s' occurred in agent %d" err.Message agentId })

let agent cancellationToken =
        new Agent<string>((fun inbox ->
                   async { while true do
                             let! msg = inbox.Receive()
                         failwith "fail!" }), cancellationToken.Token)
         agent.Error.Add(fun error -> errorAgent.Post (error))
          agent.Start()
          agent


// (agent :> IDisposable).Dispose()
```

Agent Replying to the sender

Message carries input and a callback

```
type Message = string * AsyncReplyChannel<string>
```

Reply using the callback object

```
let echo = Agent<Message>.Start(fun agent ->
 async { while true do
           let! name, rchan = agent.Receive()
           rchan.Reply("Hello " + name) })
```

Asynchronous communication

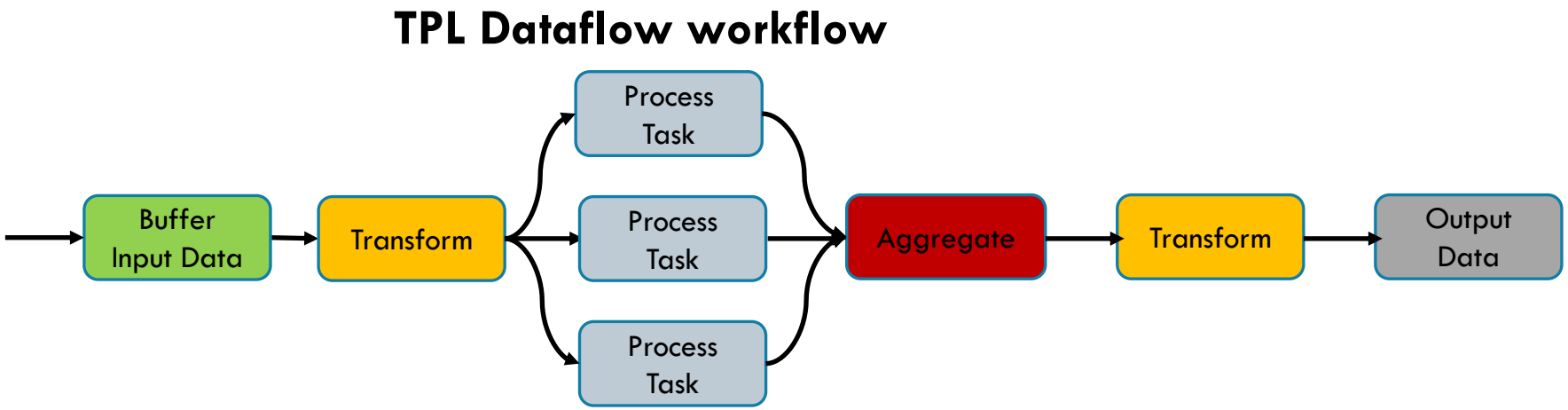
```
let! s = echo.PostAndAsyncReply(fun ch -> “F#", ch)
```

MailboxProcessor interoperability with C#

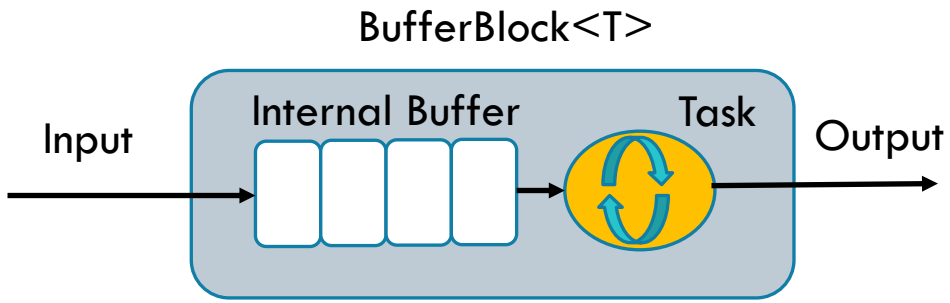
```
type internal OnePlaceMessage<'T> =
  | Put of 'T
  | Get of AsyncReplyChannel<'T>

type OnePlaceAgent<'T>() =
  let agent = Agent.Start(fun agent -> (* ... *))
  member x.Put(value) = agent.Post(Put value)
  member x.AsyncGet() = agent.PostAndAsyncReply(Get)
                                |> Async.StartAsTask
```

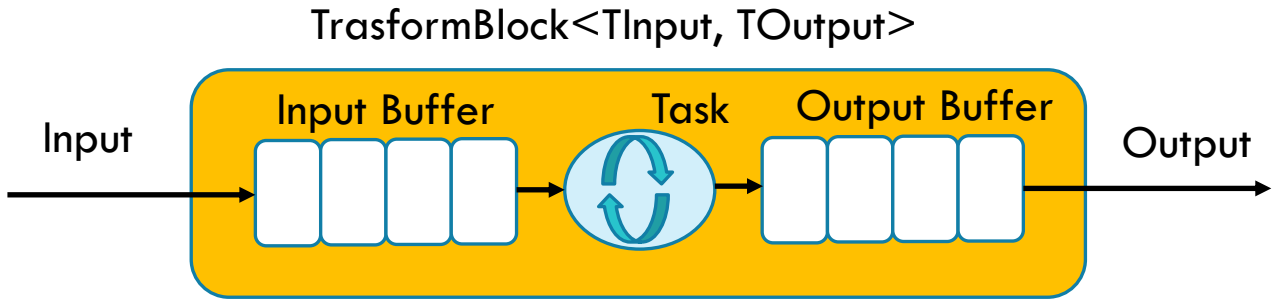
Agent in .NET and C#

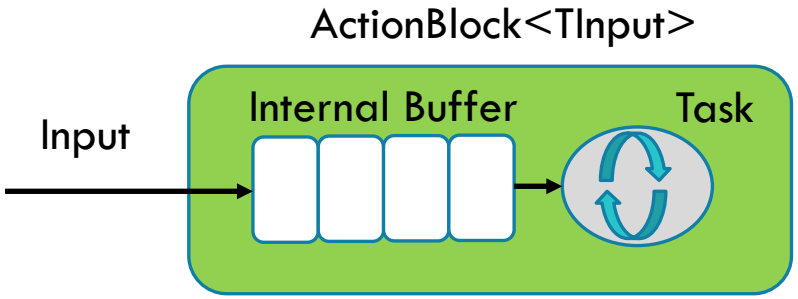
TPL DataFlow blocks –design to compose



Some Dataflow blocks



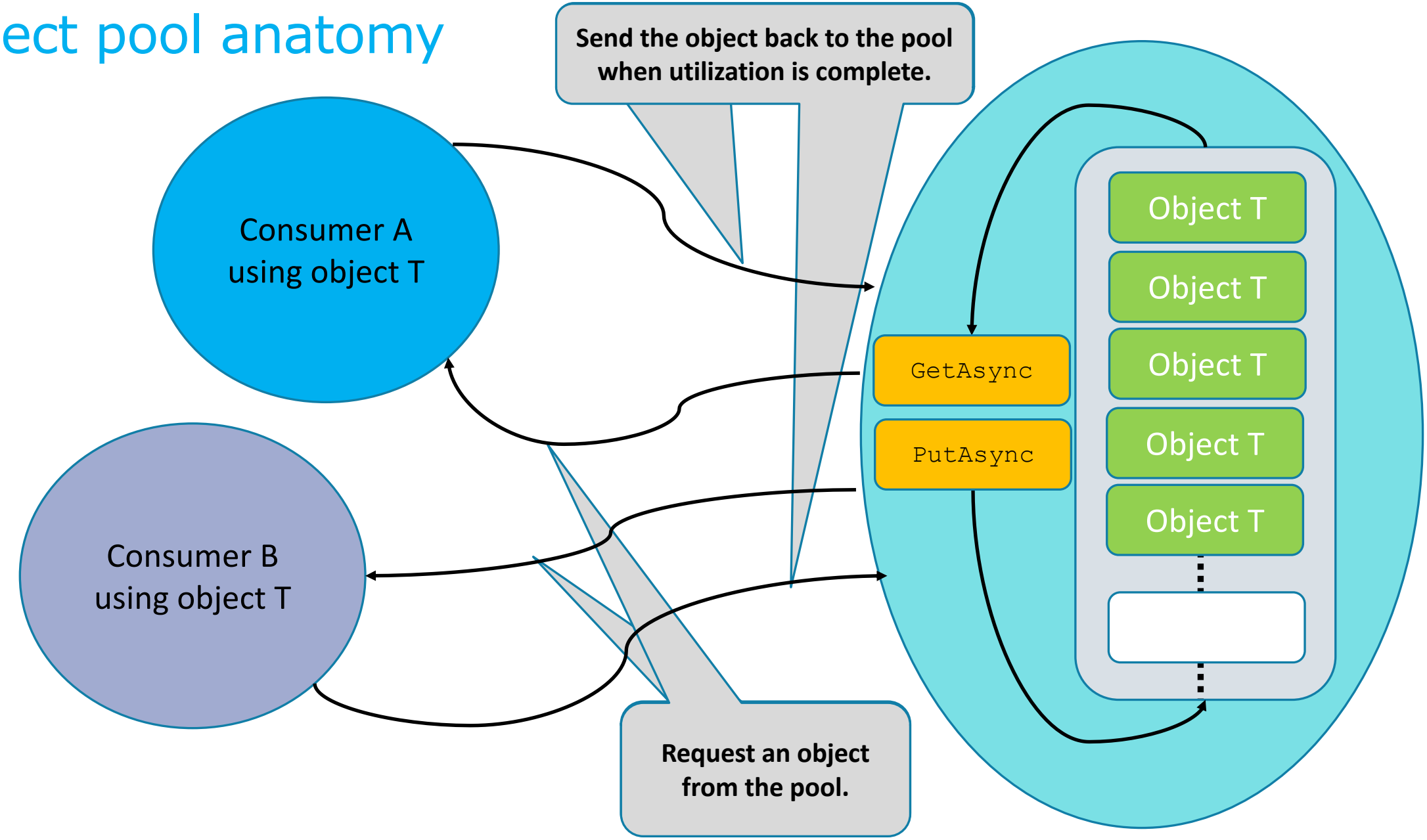




Simple producer-consumer

```
BufferBlock<int> buffer = new BufferBlock<int>();

async Task Producer(IEnumerable<int> values) {
        foreach (var value in values)
                buffer.Post(value);
        buffer.Complete();
}
async Task Consumer(Action<int> process)  {
        while (await buffer.OutputAvailableAsync())
              process(await buffer.ReceiveAsync());

}
async Task Run()  {
        IEnumerable<int> range = Enumerable.Range(0,100);
        await Task.WhenAll(Producer(range), Consumer(n =>
                Console.WriteLine($"value {n}")));
}
```

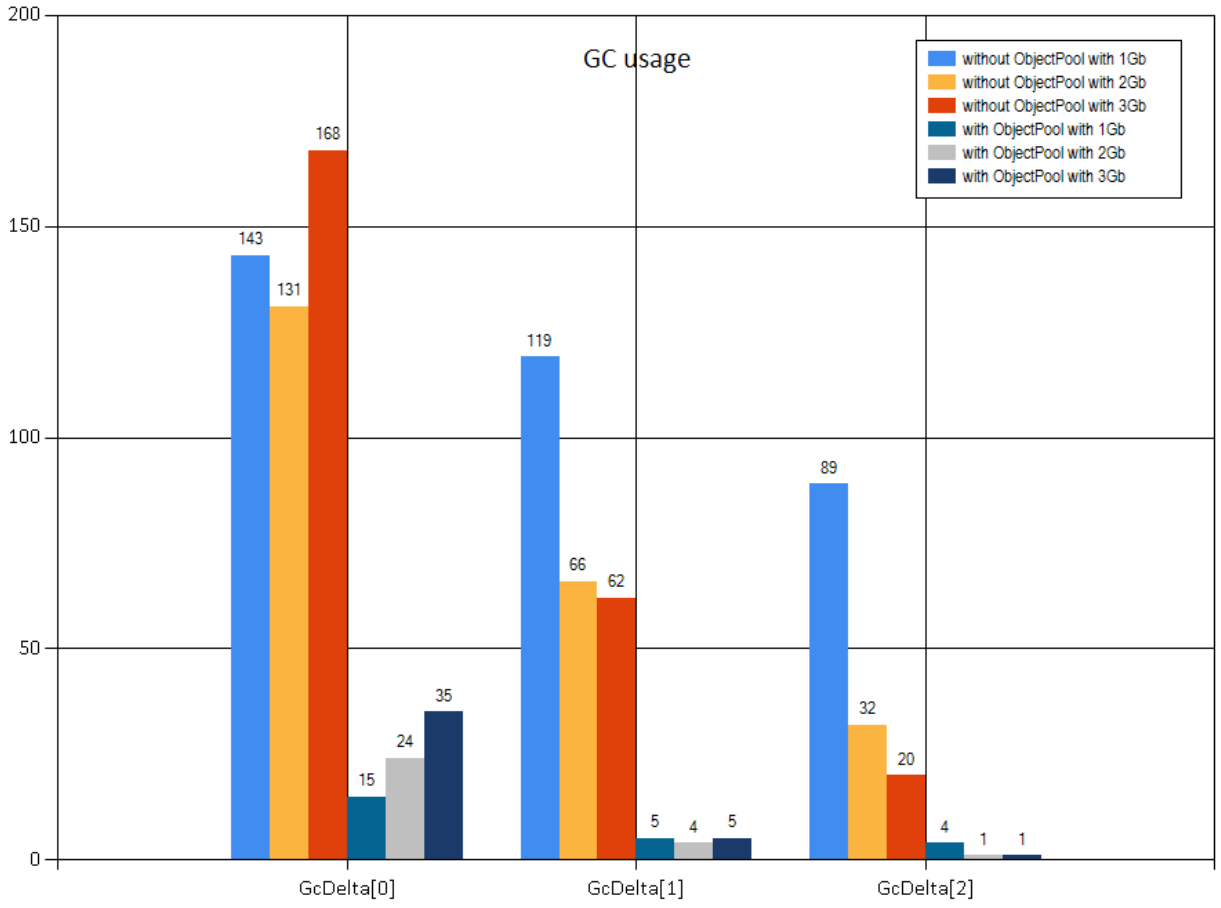
Object pool anatomy



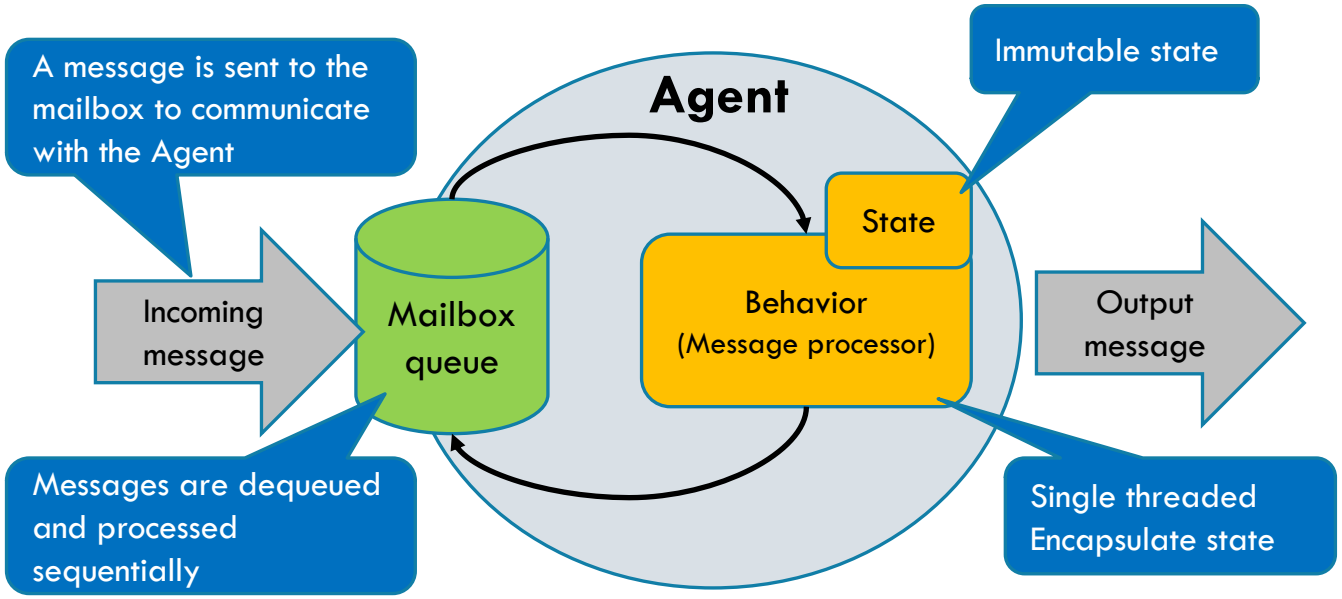
Async-PoolObject

```
public class ObjectPoolAsync<T>
{
    private readonly BufferBlock<T> buffer;
    private readonly Func<T> factory;
    private readonly int msecTimeout;
    private int currentSize;

    public ObjectPoolAsync(int initialCount, Func<T> factory, CancellationToken cts, int msecTimeout = 0)
    {
        this.msecTimeout = msecTimeout;
        buffer = new BufferBlock<T>(  // #A
            new DataflowBlockOptions { CancellationToken = cts });
        this.factory = () => {
            Interlocked.Increment(ref currentSize);
            return factory();
        };
        for (int i = 0; i < initialCount; i++)
            buffer.Post(this.factory());  // #B
    }
    public int Size => currentSize;

    public Task<bool> PushAsync(T item) =>  …
    public Task<T> GetAsync(int timeout = 0)   …
```

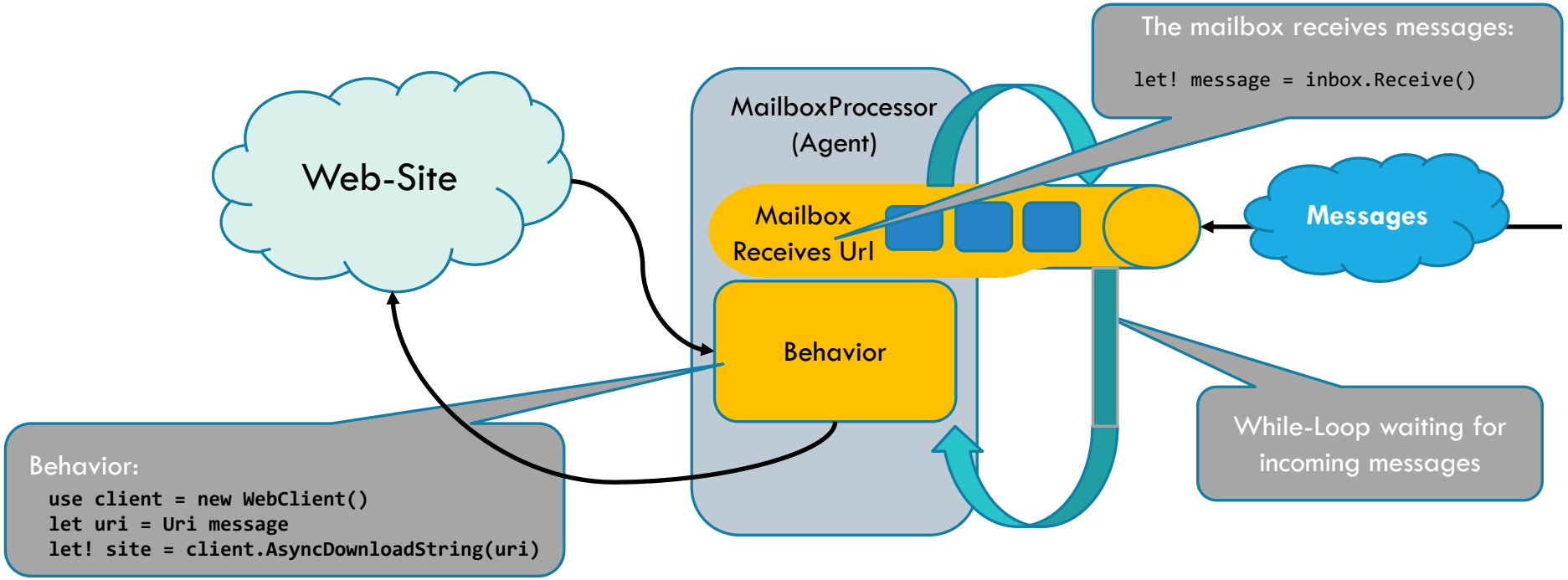
Async-PoolObject benefit



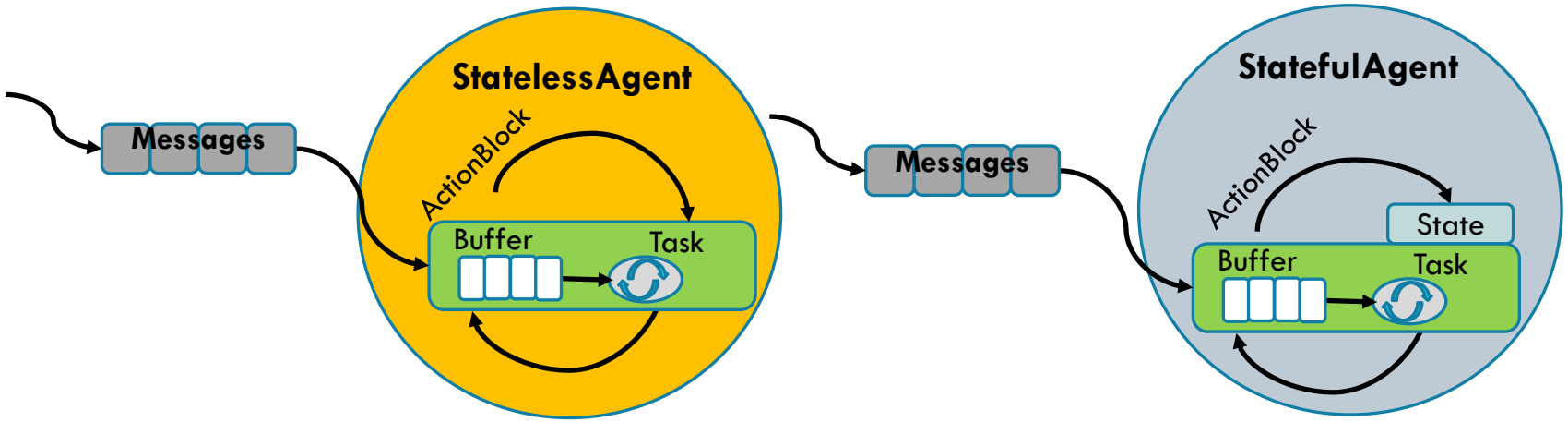
Agent anatomy



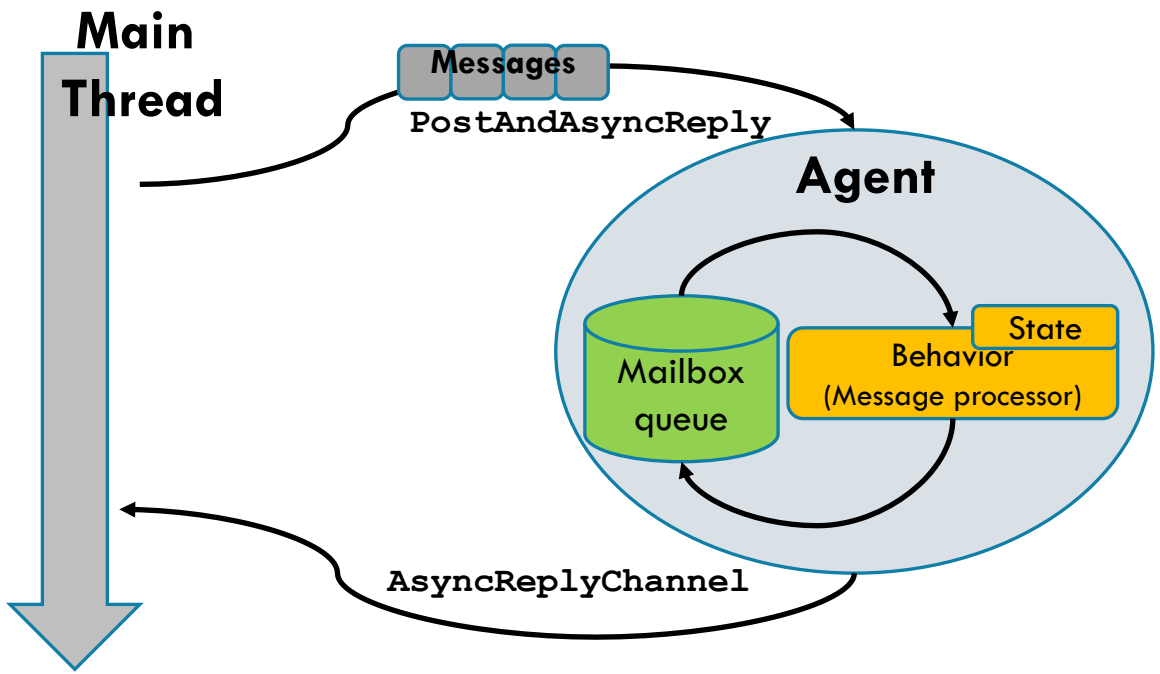
Agent anatomy

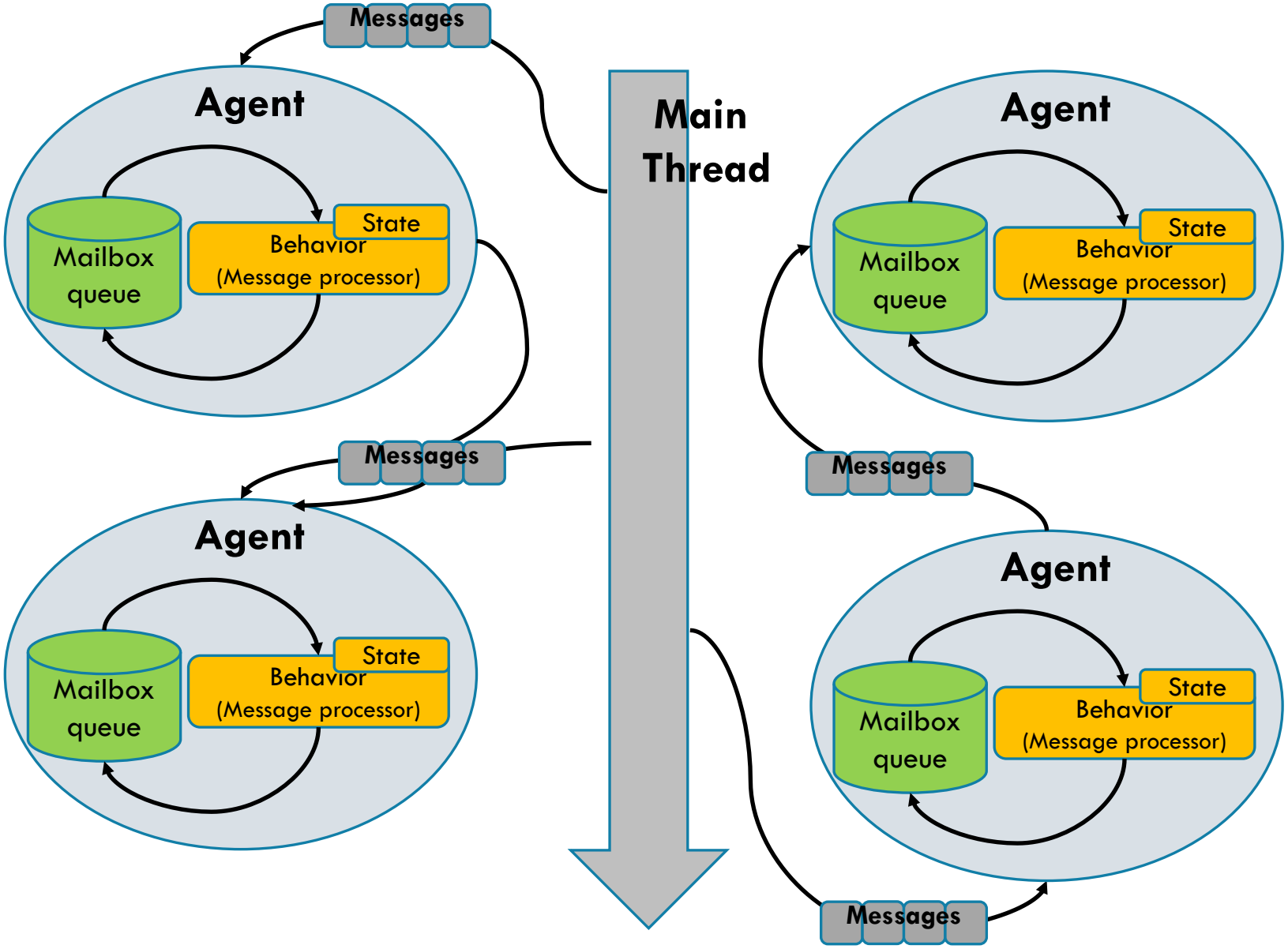


TPL DataFlow as Agent



Agent two-way communication




Messages
Agent
Mailbox queue
State
Behavior
(Message processor)
Main Thread
Agent
Mailbox queue
State
Behavior
(Message processor)
Messages
Agent
Mailbox queue
State
Behavior
(Message processor)
Messages
Agent
Mailbox queue
State
Behavior
(Message processor)
Messages

Implementing a state full agent using TPL DataFlow

TPL DataFlow a statefull Agent in C#

```
class StatefulDataflowAgent<TState, TMessage> : IAgent<TMessage>
{
    private TState state;
    private readonly ActionBlock<TMessage> actionBlock;


    public StatefulDataflowAgent(
        TState initialState,
        Func<TState, TMessage, Task<TState>> action,
        CancellationTokenSource cts = null)
    {
        state = initialState;
        var options = new ExecutionDataflowBlockOptions
        {
            CancellationToken = cts != null ?
                cts.Token : CancellationToken.None
        };
        actionBlock = new ActionBlock<TMessage>(
            async msg => state = await action(state, msg), options);
    }

    public Task Send(TMessage message) => actionBlock.SendAsync(message);
    public void Post(TMessage message) => actionBlock.Post(message);
}
```

Agent two-way communication in C#

```
public Task<TReply> Ask(TMessage message)
{
    var tcs = new TaskCompletionSource<TReply>();
    actionBlock.Post((message, Some(tcs)));
    return tcs.Task;
}

public StatefulReplyDataflowAgent(TState initialState,
Func<TState, TMessage, Task<TState>> projection,
Func<TState, TMessage, Task<(TState, TReply)>> ask,
CancellationTokenSource cts = null)
{
    state = initialState;
    actionBlock = new ActionBlock<(TMessage, Option<TaskCompletionSource<TReply>>)>(
        async message =>
        {
            (TMessage msg, Option<TaskCompletionSource<TReply>> replyOpt) = message;
            await replyOpt.Match(
                    none: async () => state = await projection(state, msg),
                    some: async reply => {
                        (TState newState, TReply replyresult) = await ask(state, msg);
                        state = newState;
                        reply.SetResult(replyresult);
                    });
```

TPL DataFlow caching web-sites downloaded

```
List<string> urls = new List<string> {
                        ”http://www.google.com",      "http://www.microsoft.com",
                        "http://www.bing.com",        "http://www.google.com"
            };

var agentStateful = Agent.Start(ImmutableDictionary<string,string>.Empty,
   async (ImmutableDictionary<string,string> state, string url) => {

if (!state.TryGetValue(url, out string content))
               using (var webClient = new WebClient()){
                        content = await webClient.DownloadStringTaskAsync(url);
                        await File.WriteAllTextAsync(createFileNameFromUrl(url), content);
                        return state.Add(url, content);
             }
          return state;
   });

urls.ForEach(url => agentStateful.Post(url));
```

Agent fold-over state and messages (Aggregate)

```
Agent(ImmutableDictionary<string,string>.Empty,
        async (state, url) => {
if (!state.TryGetValue(url, out string content))
  using (var webClient = new WebClient())
  {
        content = await webClient.DownloadStringTaskAsync(url);
        await File.WriteAllTextAsync(createFileNamFromUrl(url),     content);
        return state.Add(url, content);
  }
  return state;
});
```

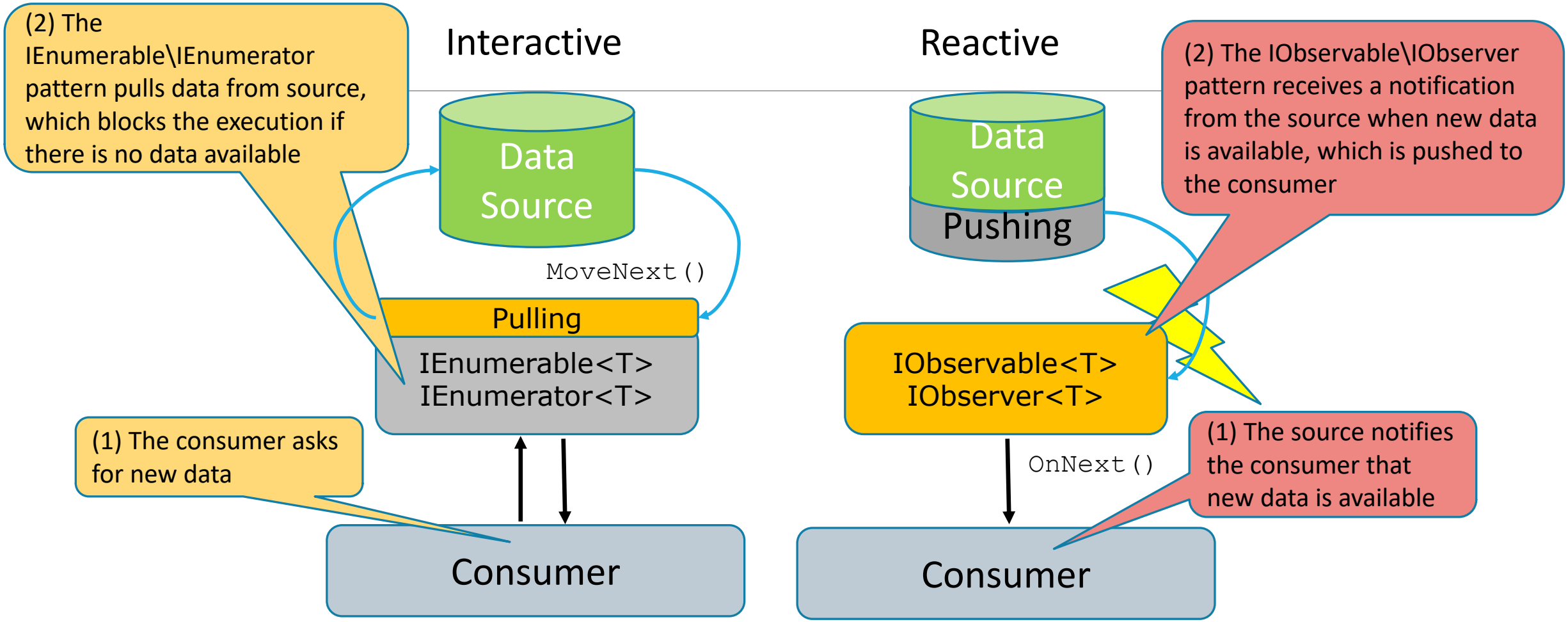
Agent fold-over state and messages (Aggregate)

```
urls.Aggregate(ImmutableDictionary<string,string>.Empty,
        async (state, url) => {
if (!state.TryGetValue(url, out string content))
  using (var webClient = new WebClient())
  {
        content = await webClient.DownloadStringTaskAsync(url);
        await File.WriteAllTextAsync(createFileNamFromUrl(url),     content);
        return state.Add(url, content);
  }
  return state;
});
```

TPL DataFlow & Reactive Extensions

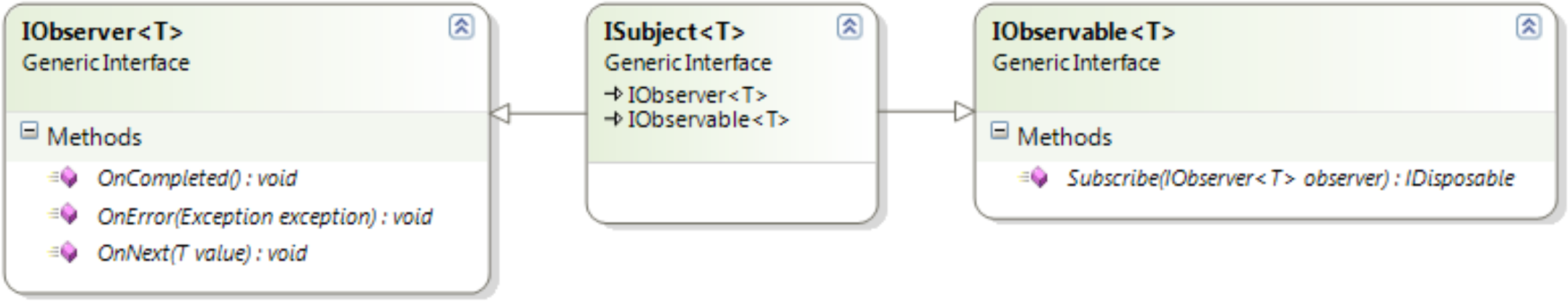
TPL Dataflow & Reactive Extension

Reactive Extensions


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

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

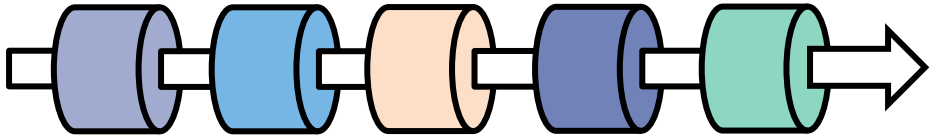
```
type IObserver<'a> = interface
    abstract OnCompleted : unit -> unit
    abstract OnError : exn -> unit
    abstract OnNext : 'a -> unit
end

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

TPL Dataflow & Reactive Extension

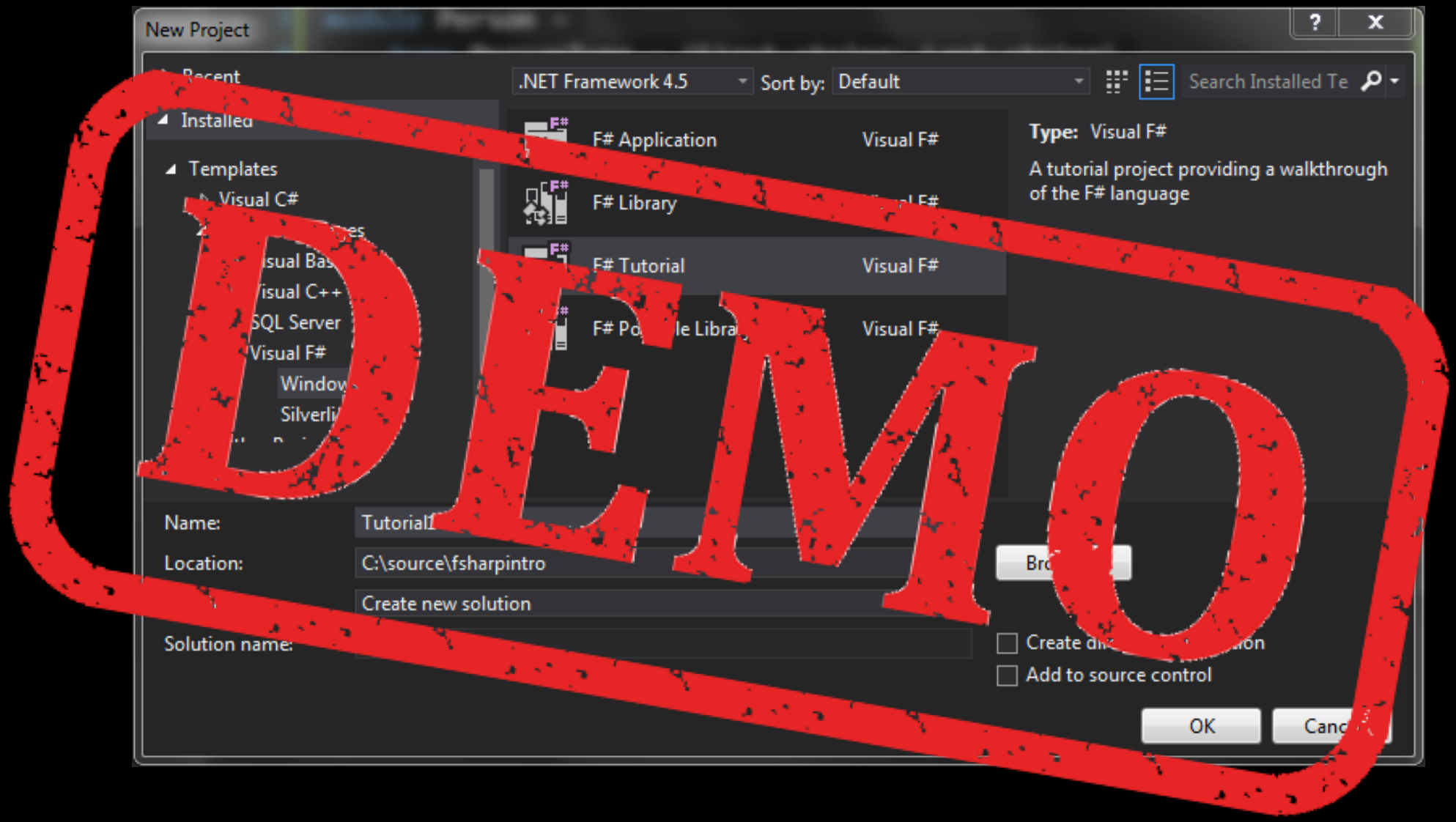
```
IPropagatorBlock<int, string> source =
            new TransformBlock<int, string>(i => (i + i).ToString());

IObservable<int> observable = source.AsObservable().Select(int.Parse);

IDisposable subscription =
              observable.Subscribe(i => $"Value {i} - Time
                                {DateTime.Now.ToString("hh:mm:ss.fff")}".Dump());


for (int i = 0; i < 100; i++)
         source.Post(i);
```

TPL Dataflow & Reactive Extension

```
IPropagatorBlock<string, int> target =
                                new TransformBlock<string, int>(s => int.Parse(s));

IDisposable link = target.LinkTo(new ActionBlock<int>(i => $"Value {i} - Time
                                    {DateTime.Now.ToString("hh:mm:ss.fff")}".Dump())

IObserver<string> observer = target.AsObserver();
IObservable<string> observable = Observable.Range(1,20)
                                        .Select(i => (i *i).ToString());
observable.Subscribe(observer);

for (int i = 0; i < 100; i++)
        target.Post(i.ToString());
```

TPL DataFlow and Rx

```
inputBuffer.LinkTo(compressor, linkOptions);
compressor.LinkTo(encryptor, linkOptions);

encryptor.AsObservable()

          .Scan((new Dictionary<int, EncryptDetails>(), 0),
        (state, msg) => Observable.FromAsync(async() => {
         (Dictionary<int,EncryptDetails> details, int lastIndexProc) = state;
         details.Add(msg.Sequence, msg);

         return (details, lastIndexProc);
}) .SingleAsync())
.SubscribeOn(TaskPoolScheduler.Default).Subscribe();
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
F# Application
Visual F#
F# Library
F# Tutorial
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

contacts

Source https://github.com/rikace/mpconc-ws

Twitter *@trikace*

Blog *www.rickyterrell.com*

Email *tericcardo@gmail.com*

Github *www.github.com/rikace/*

Thank you

RICCARDO TERRELL - @TRIKACE

That's all Folks!