Functional Concurrency with F#

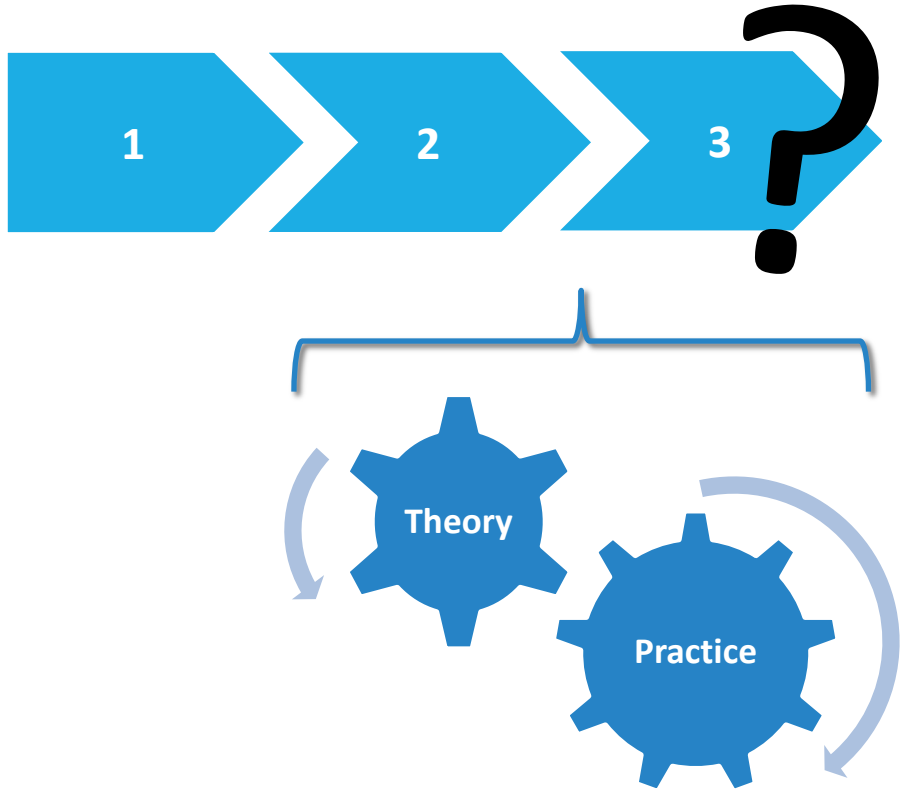
BY RICCARDO TERRELL - @TRIKACE

TERICCARDO@GMAIL.COM

Agenda

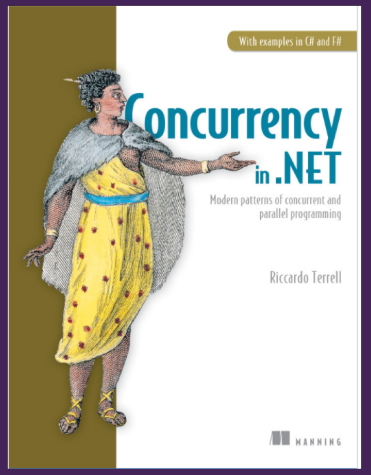
- Message Passing programming model & motivation
- How to parallelize Actors/Agents seamlessly
- Combinators to compose Asynchronous operations and Agent/Actors
- Message Passing and Reactive programming

Modules



Introduction - Riccardo Terrell

- Originally from Italy, currently - Living/working in Washington DC
- +/- 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 the DC F# User Group*



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

https://ti.to/luteceo-workshops/concurrent-functional-programming-in-net/en

LUTECEO workshops

Concurrent Functional Programming in .NET

April 25th–26th, 2019 Paris, France

29th - 30th April 2019 in London

https://skillsmatter.com/courses/589-functional-concurrency-in-dotnet-with-csharp-and-fsharp-with-riccardo-terrell


What about you?

Download links:
https://github.com/rikace/funconcurrency-ws

See README section pre-requisites

Pre-requisites

> Windows

- > Visual Studio Community or higher
- > Visual Studio Code
  - > C# Extensions
  - > F# Compiler + Ionide package
- > Rider

>Linux

- > Visual Studio Code + (Mono or dotnetcore) + Ionide package

>Mac

- > Visual Studio for Mac + (Mono or dotnetcore) or
- > Visual Studio Code + (Mono or dotnetcore) + Ionide package
- > Rider

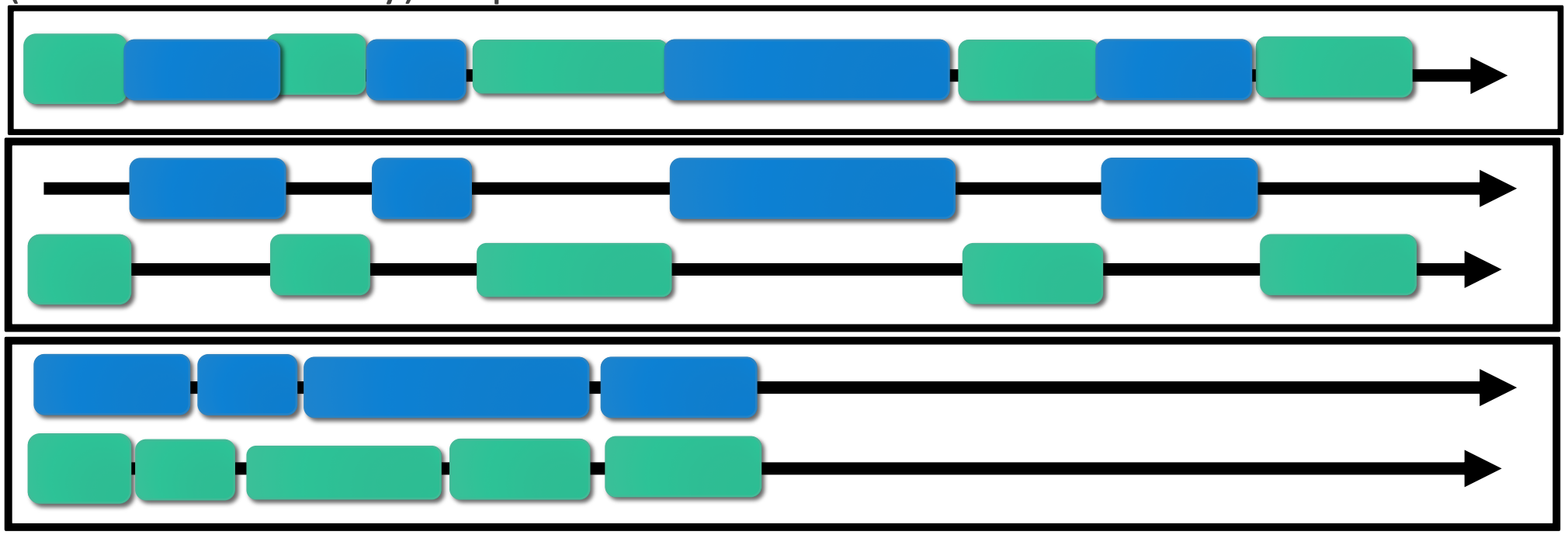
Download links:
https://github.com/rikace/funconcurrency-ws

See README section pre-requisites


We need a way to make sure our programs continue to improve their speed and performance as hardware evolves. Have any ideas?

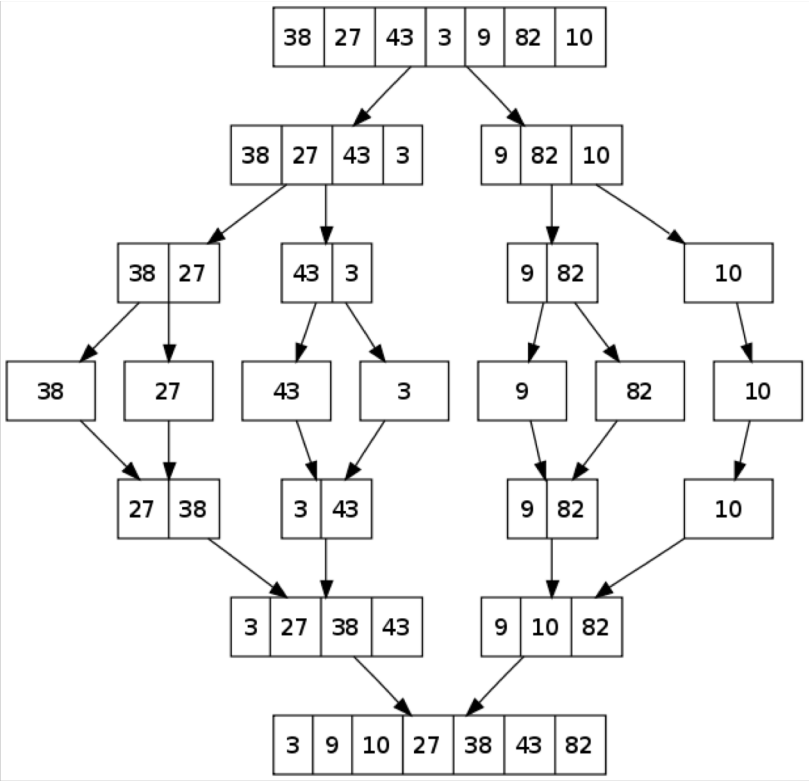
Concurrency programming

Concurrency provides a way to **structure a solution** to solve a problem that may (but not necessarily) be parallelizable.



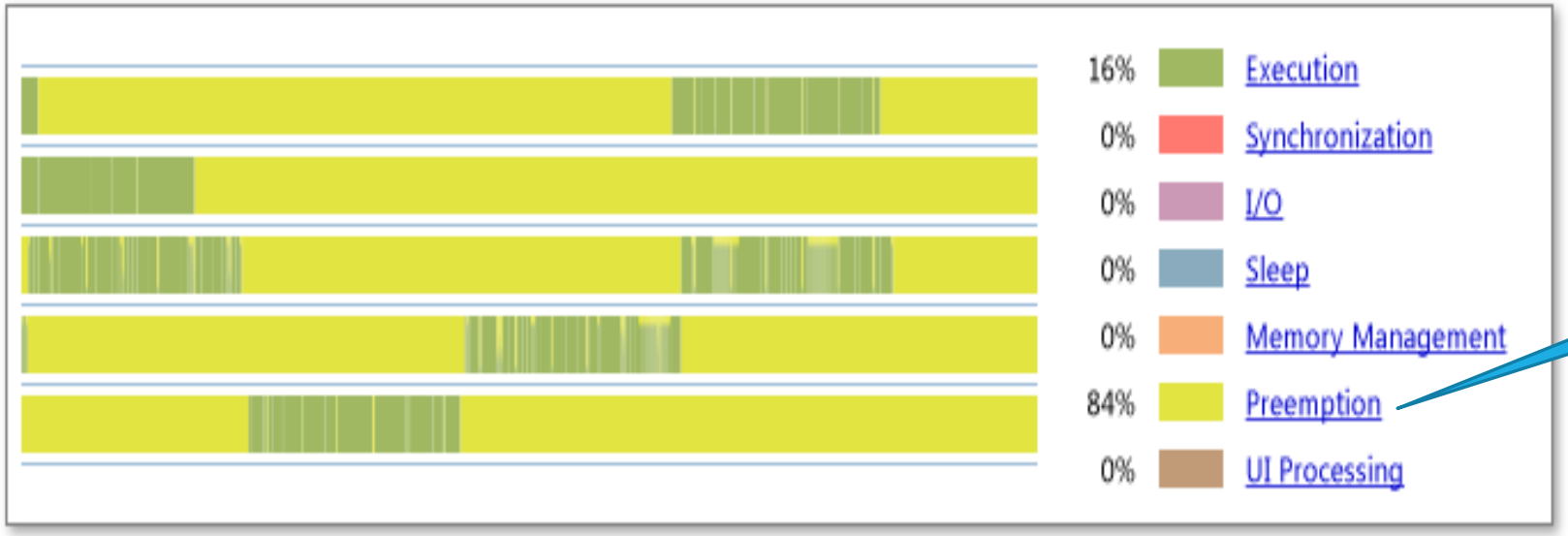
Concurrency is the **composition** of independently executing computations.

Divide and Conquer




```
void QuickSort_Parallel<T>(T[] items, int left, int right)
{
    int pivot = left;


    SwapElements(items, left, pivot);

    Parallel.Invoke(
            () => QuickSort_Parallel(items, left, pivot - 1),
            () => QuickSort_Parallel(items, pivot + 1, right)
        );
}
```

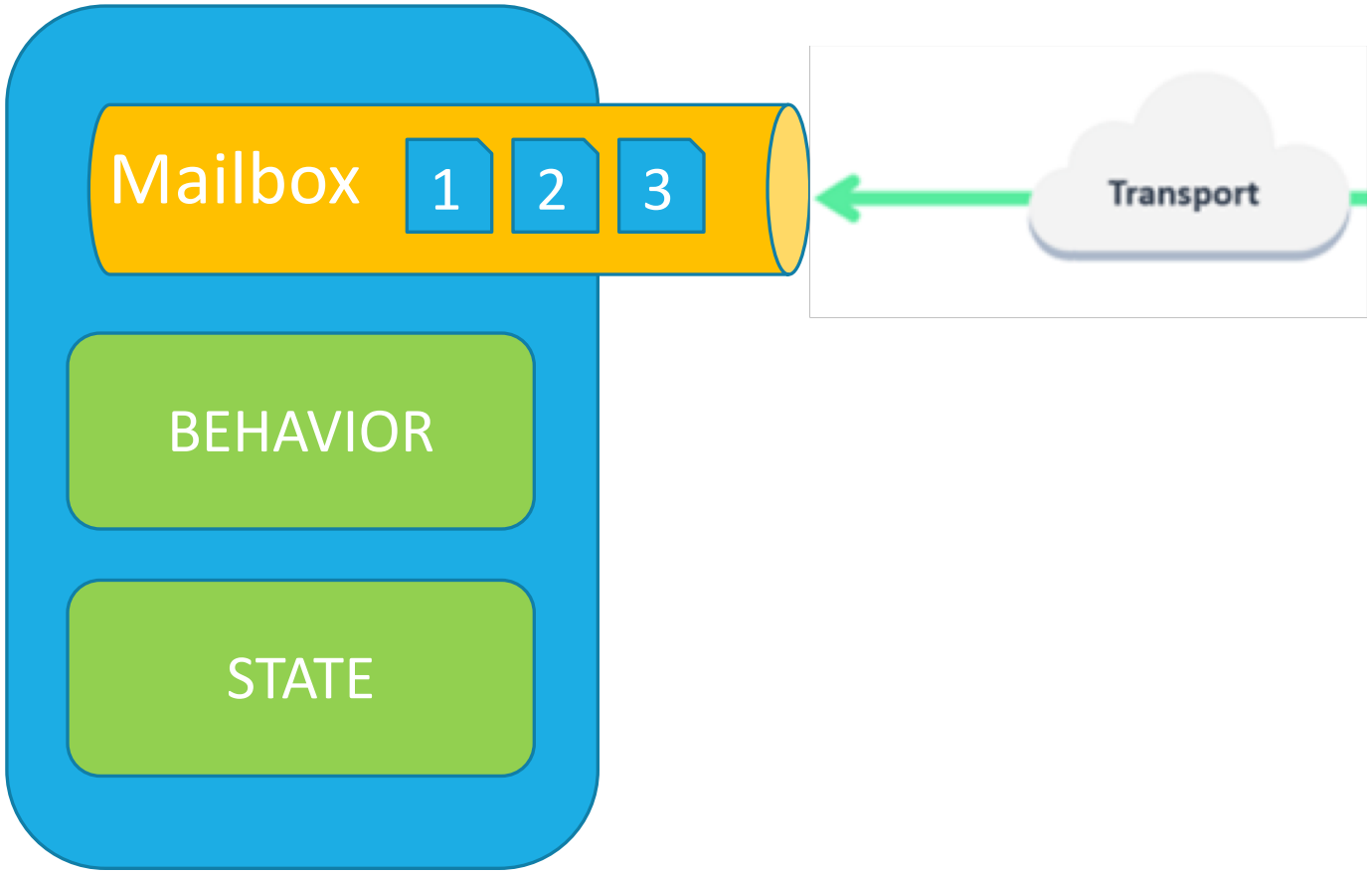
Quick Sort – Oversubscription



Preemption

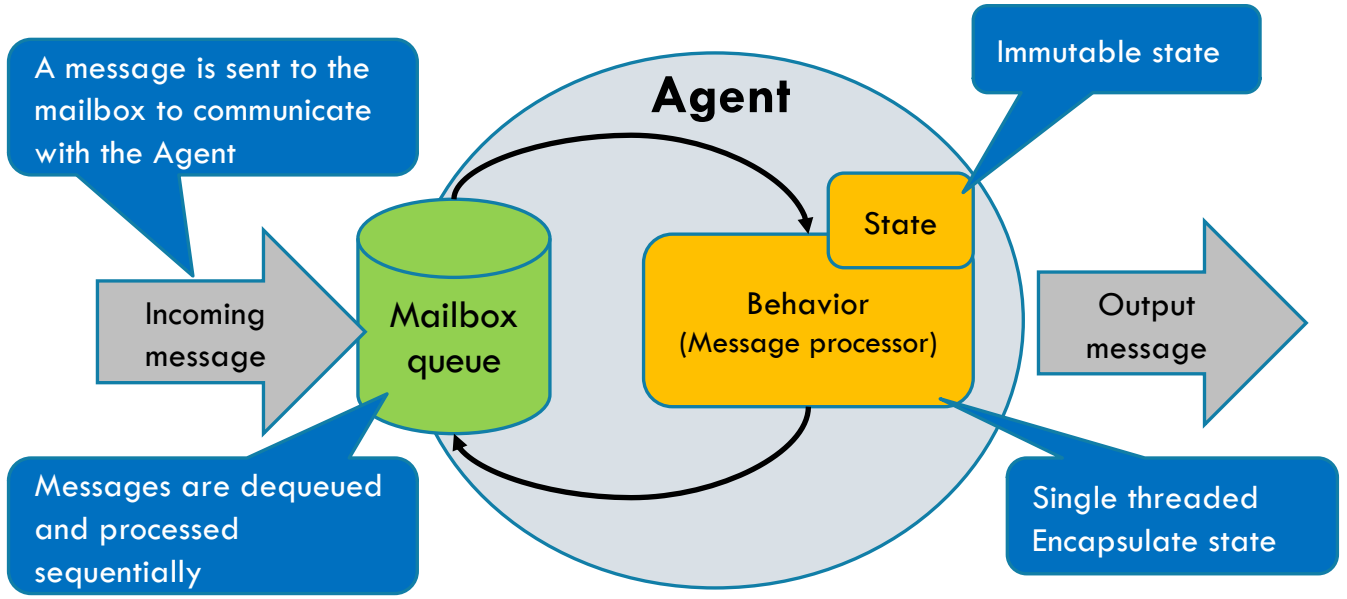
Message Passing

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

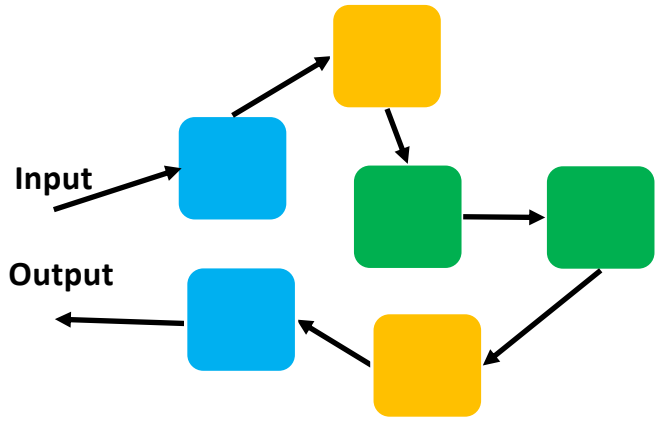
Agent anatomy



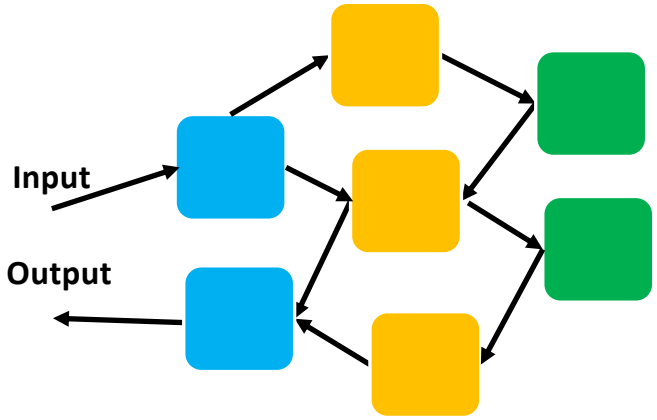
- Communication between agents done by **sending message**
- The messages are put on a **queue** for each agent
- Each agent process **one message** at time

Comparison between Sequential, Task-based and Message passing programming

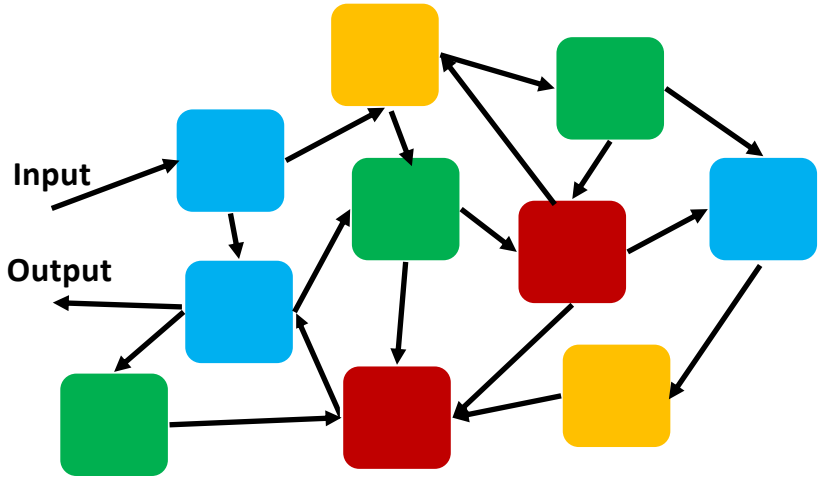
Sequential Programming



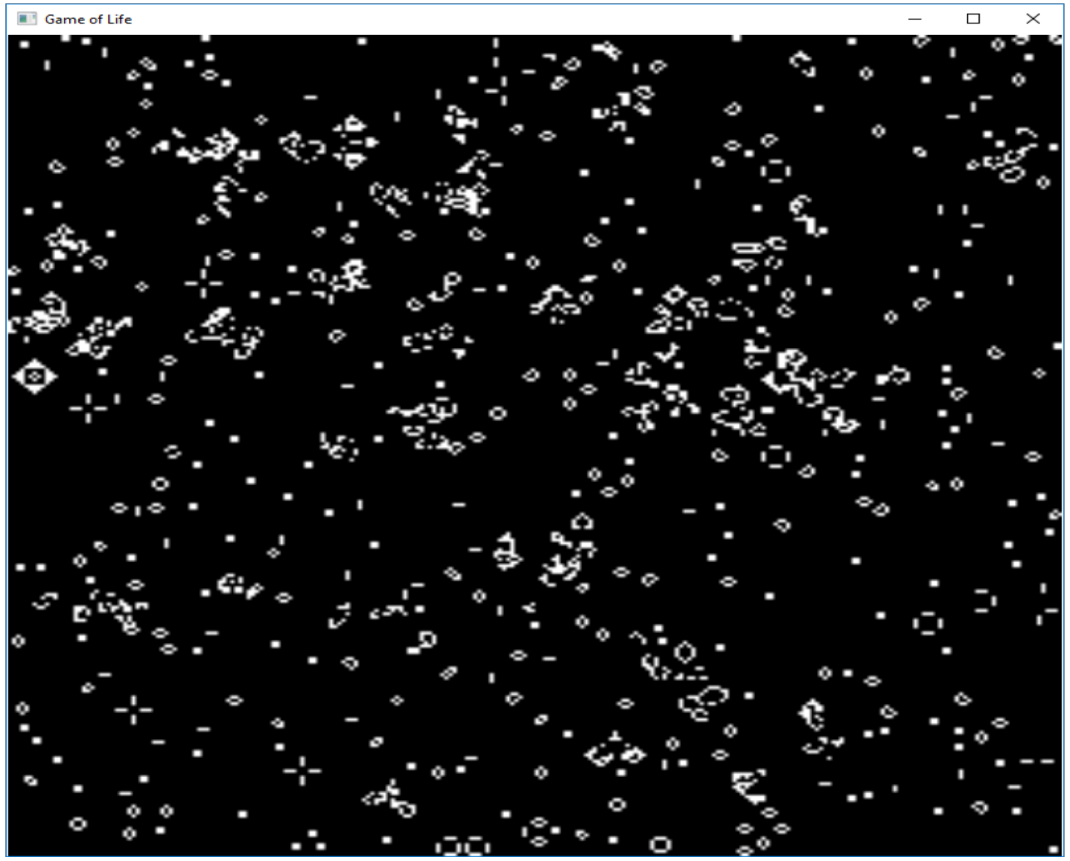
Task-Based Programming



Message-Passing Programming



Game of Life



The Game of Life rules:

- Each cell with one or no neighbors dies, as if by solitude.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.
- Each cell with three neighbors becomes populated (maximum).

Simple agent in F#

Receive message and say “Hello”

```
let hello = MailboxProcessor.Start(fun agent -> async {
  while true do
    let! name = agent.Receive()
    printfn "Hello %s"  name
    do! Async.Sleep 1000 })

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

```
type Agent<'a> = MailboxProcessor<'a>
```

```
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 [])
```

Declaring messages

Agents handle multiple messages

- Message type using discriminated union

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

Safety guarantees

- Agent will be able to handle all messages

Send message to agent

```
printerAgent.Post "hello"

printerAgent.Post "hello again"

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

Encapsulating agents

Create type and expose messages as members

```
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)
```

- Ordinary methods for encapsulating `Post`
- Asynchronous methods when waiting for a reply

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)
```

Agent State Machine

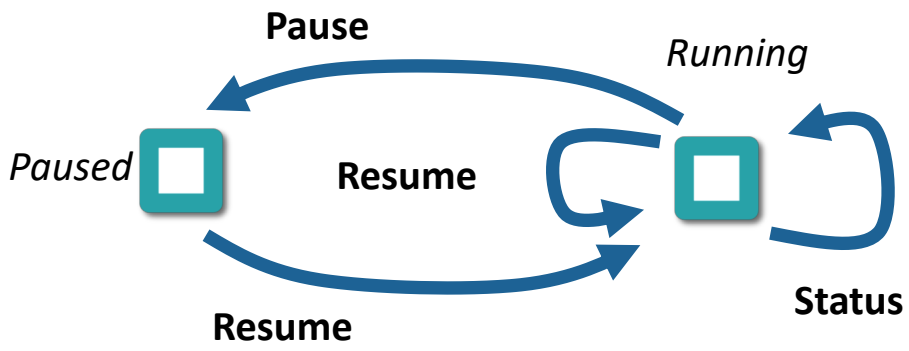
Multi-state agents use state machines

- Easy to implement as recursive functions
- Some states may leave messages in the queue

State transitions

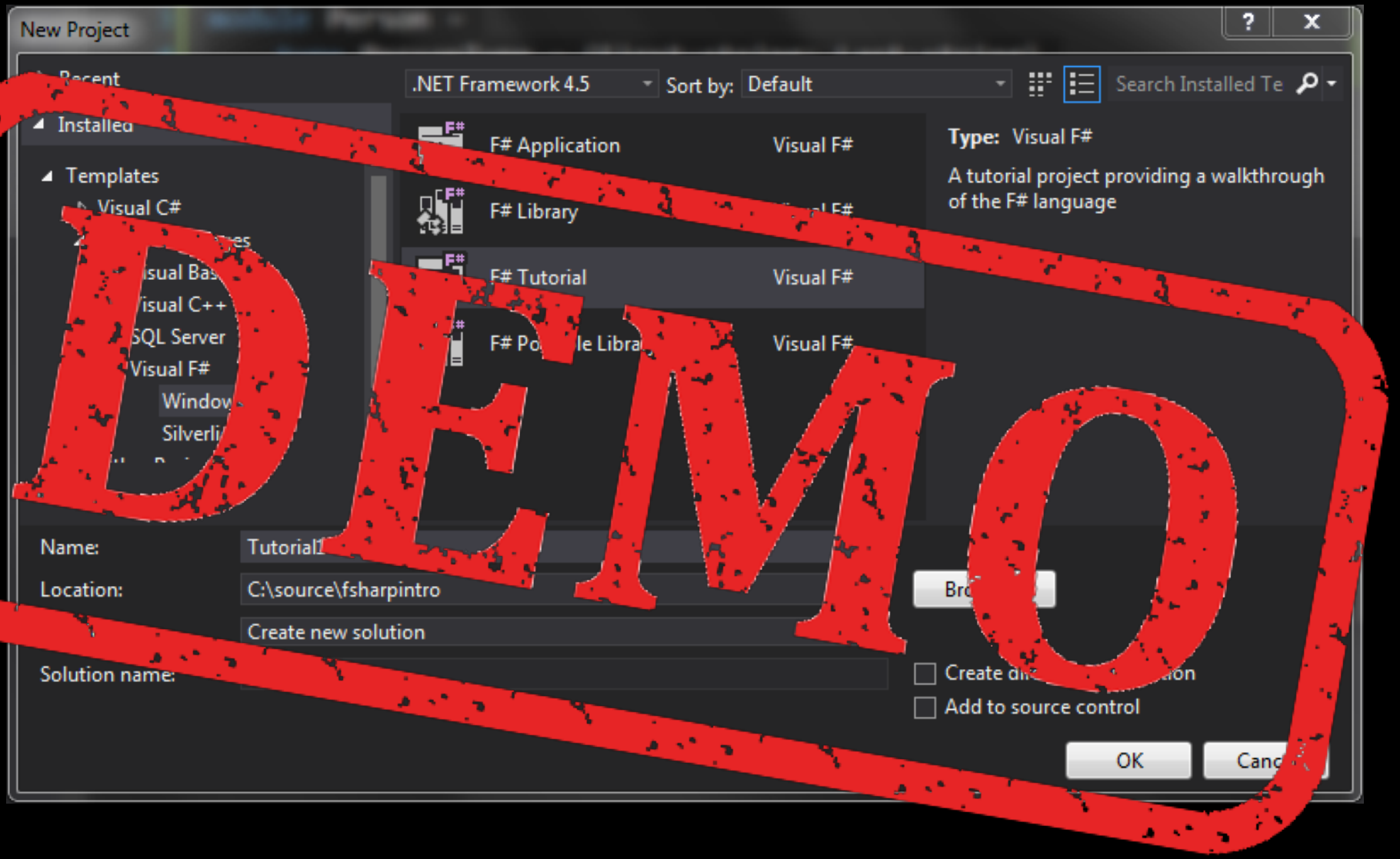
- Accepting all messages, Asynchronously Receive and use pattern matching
- Waiting for a specific message, other messages stay in the queue

```
Agent.Start(fun agent ->
  let rec paused = agent.Scan (function
    | Resume -> Some (async {
        printfn "Resumed!"
        return! running })
    | _ -> None)
  and running = (* ... *) )
```



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 agentI

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
```


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

Agent Patterns

Agent-based architecture

Using loosely coupled connections

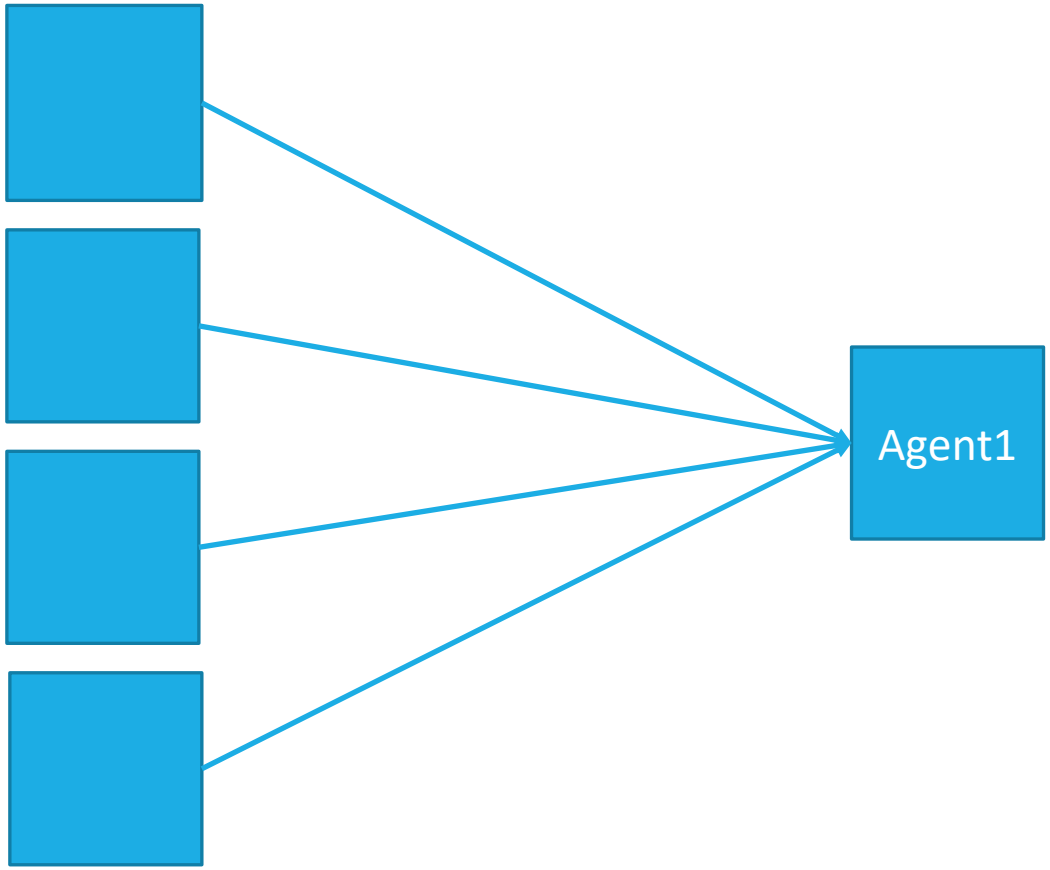
- Agents don’t reference each other directly

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

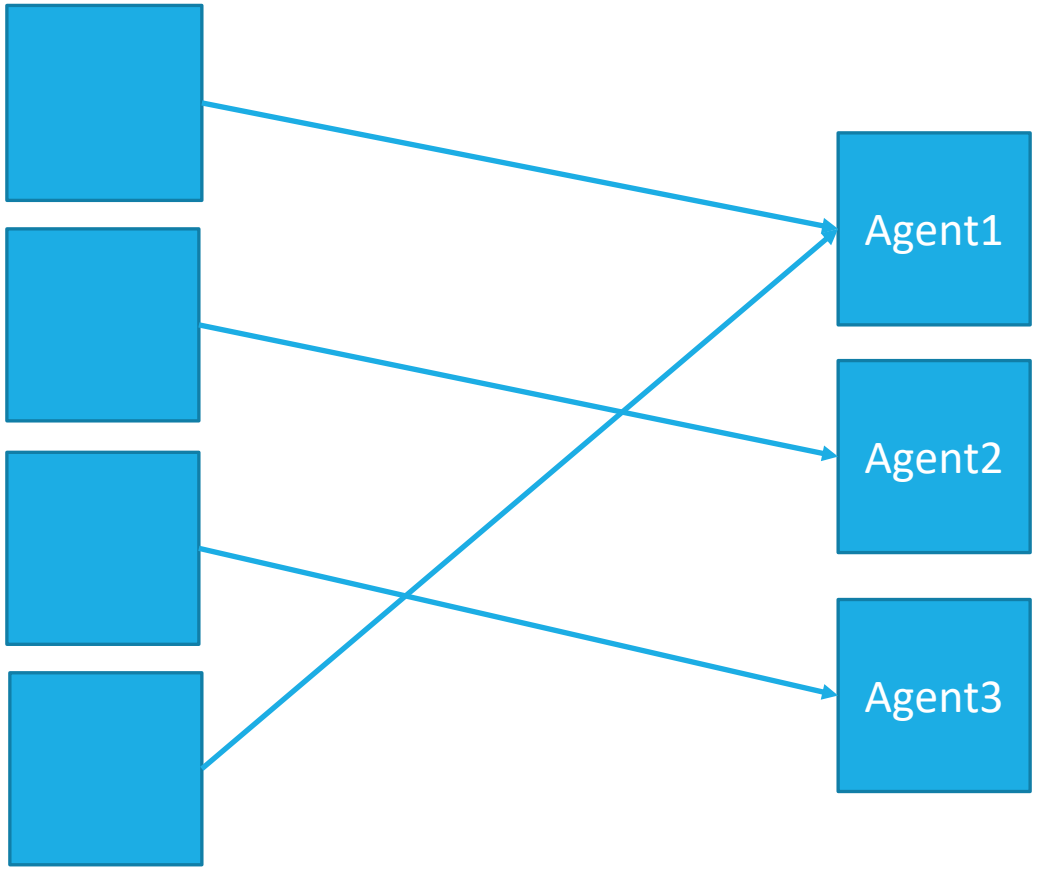
Agent patterns

Singleton



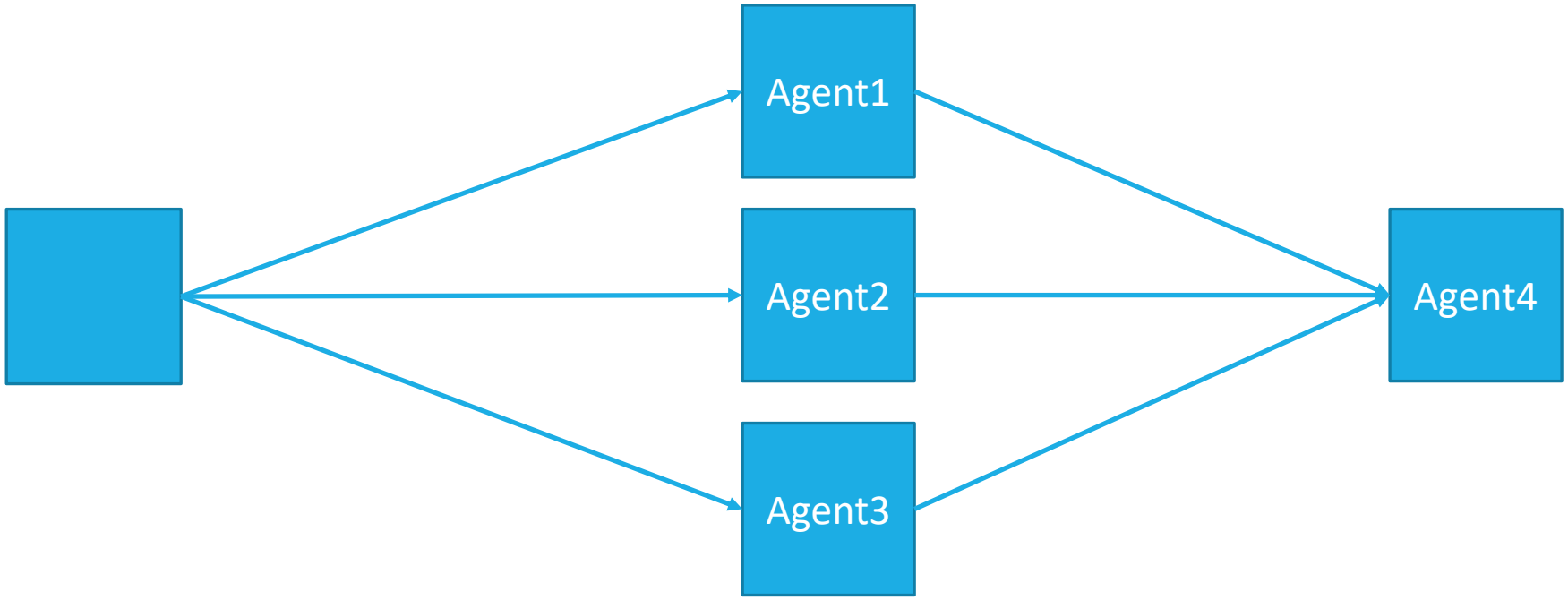
Agent patterns

Pool



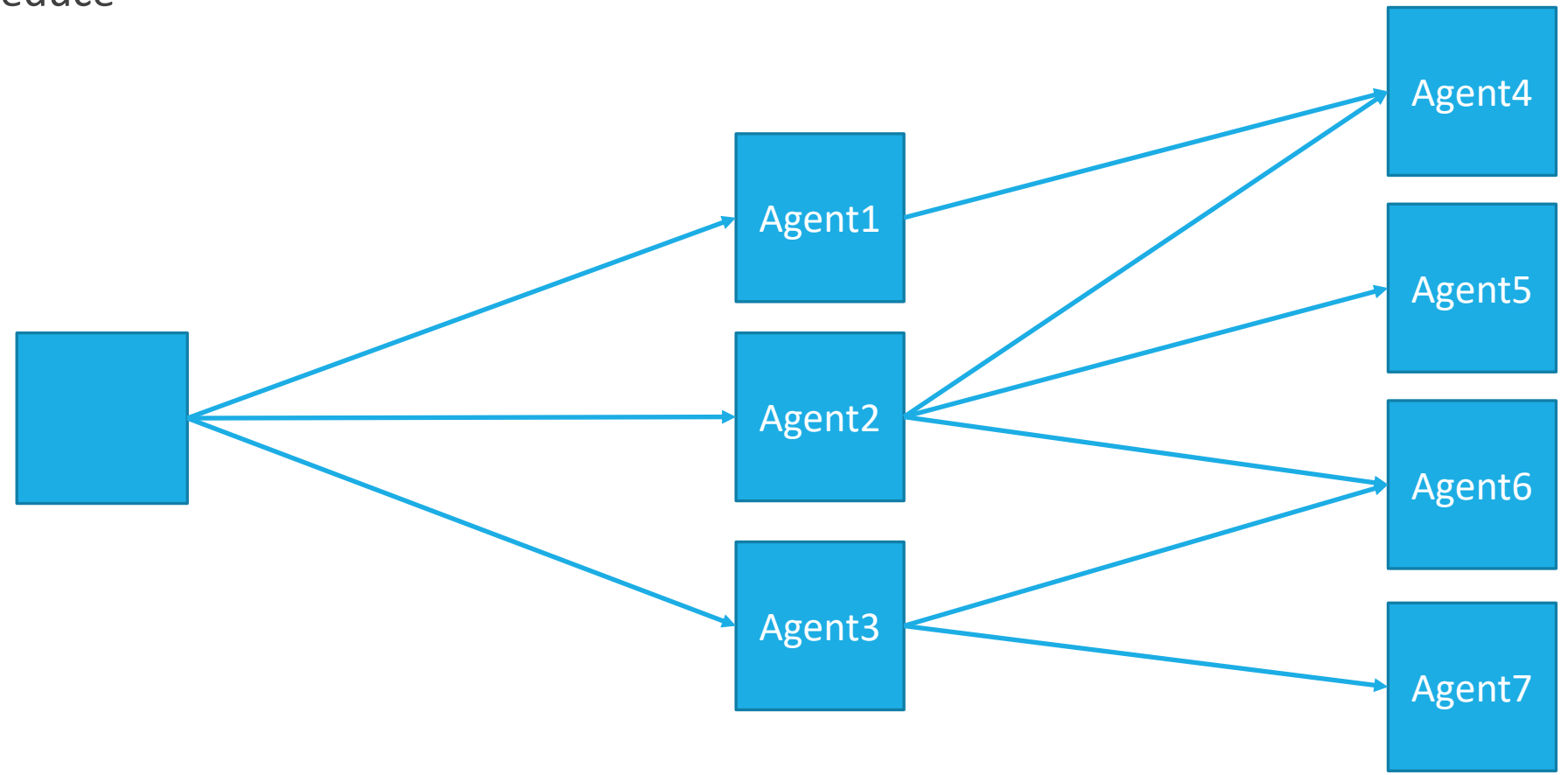
Agent patterns

Fork Join

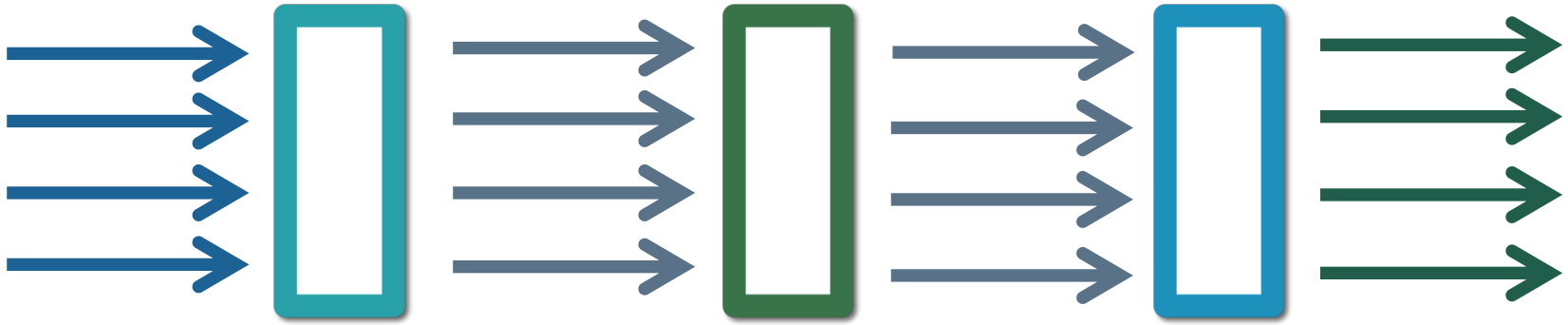


Agent patterns

Map Reduce



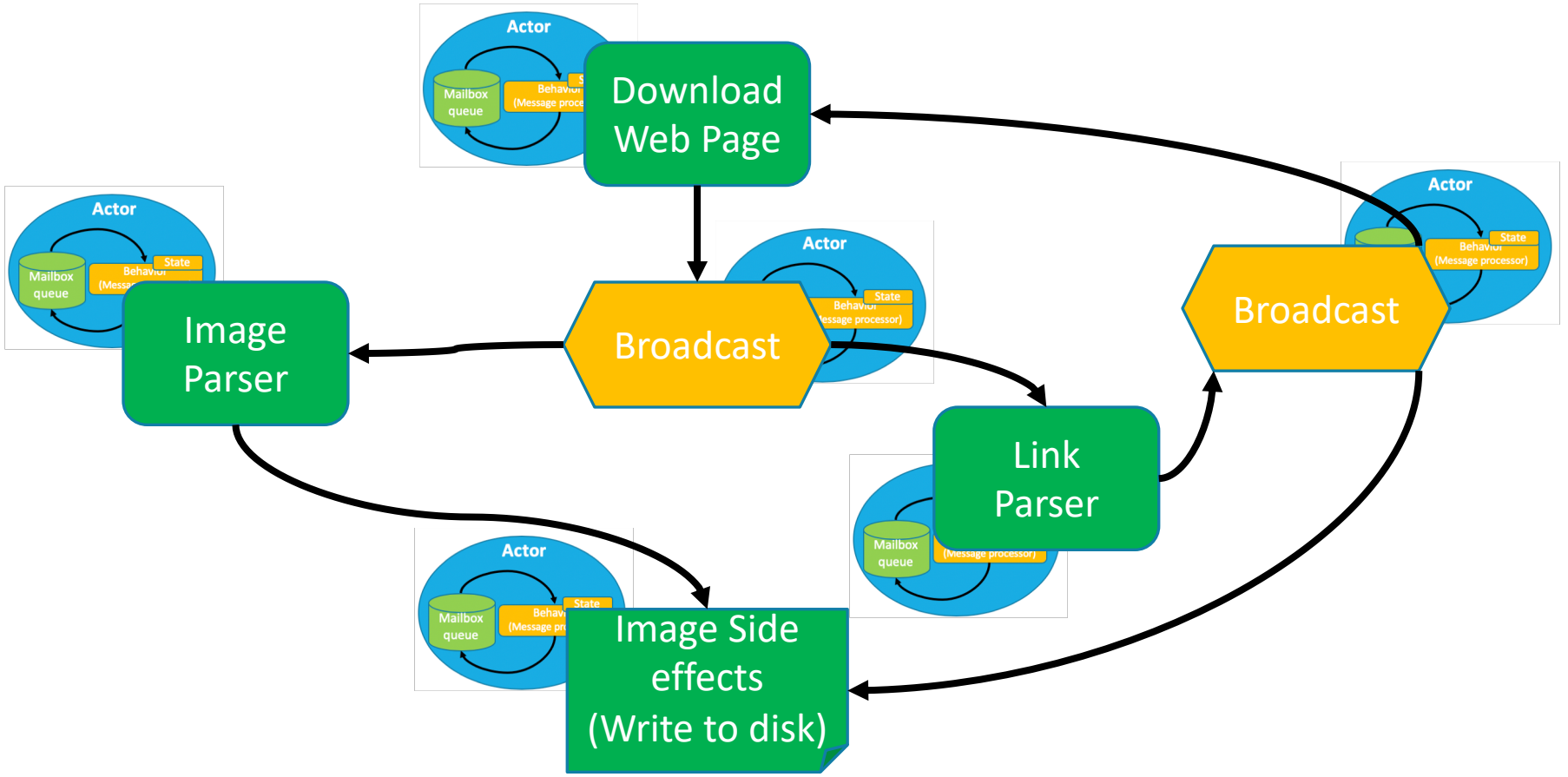
Pipeline Processing



- 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

Parallel Web-Crawler

Actors are hard to compose
An Actor-based WebCrawler



Web Crawler with Agents (assumptions)

- Web crawler must not spam websites, so each request to a single website should be delayed.
- All operations must be non-blocking.
- We assume that the web connection is not good
- Since we will be running the crawler on a regular computer, then we need to stop the system when we scrap enough links. The check itself will be regulated by a simple threshold value.

Web-Crawler elements

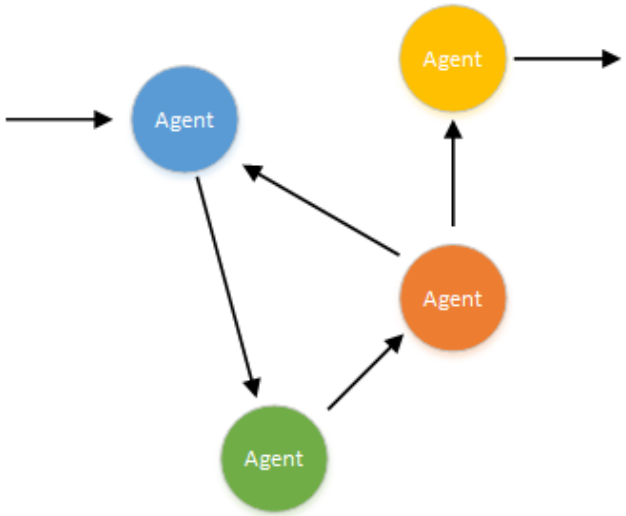
- **Downloader**: the responsibility of the downloader is to download the html of a web page. It takes a url as input and it produces the related page's html as output.
- **Link-parser:** analyzes the downloaded html to look for links elements <a href="…"/>
- **Image-parser:** analyzes the downloaded html to look for image elements <image src="…"/>
- **Broadcast**: the broadcast block is construct from an input and output of single item linking one or more agents
- **Link-together (broadcast)**: linking the block to each other
- **Writer**: persists the image and/or produce a side effect

the **downloader** should be link to the **content broadcaster** which in tern should be linked both to the **image and link parser**, the image parser should be linked to the **writer** and the **link parser** should be **linked back to the downloader** (so it can crawl farther)

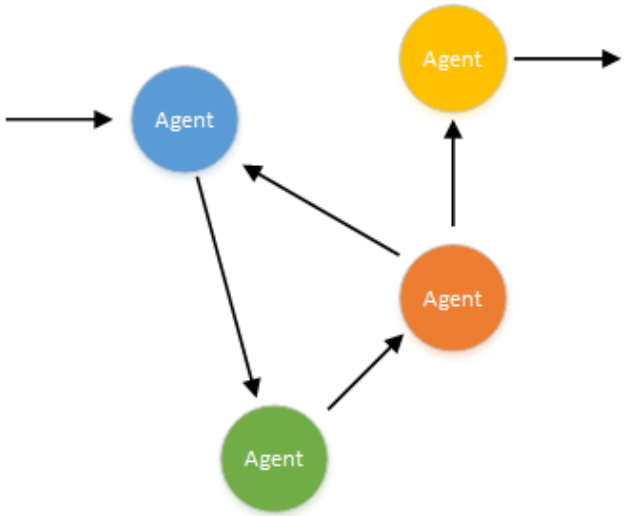
Lab: Agent WebCrawler

Agent Composition

Agent Pipeline



Agent Pipeline



Agent Pipeline

```
type Agent<'a> = MailboxProcessor<'a>
type Message = string * AsyncReplyChannel<string>

let agent f = Agent<Message>.Start(fun inbox ->
    let rec loop () = async {
        // TODO

        return! loop() }
    loop() )
```

Agent Binding

('a -> Async<'b>) -> Async<'a> -> Async<'b>

```
let bind f xAsync = async {
    let! x = xAsync
    return! f x }

let retn x = async { return x }

let (>>=) xAsync f = bind f xAsync
```

Agent Pipeline

```
let agent f = Agent<Message>.Start(fun inbox ->
    let rec loop () = async {
        let! msg, replyChannel = inbox.Receive()
        f msg
        replyChannel.Reply msg

        return! loop() }
    loop() )
```

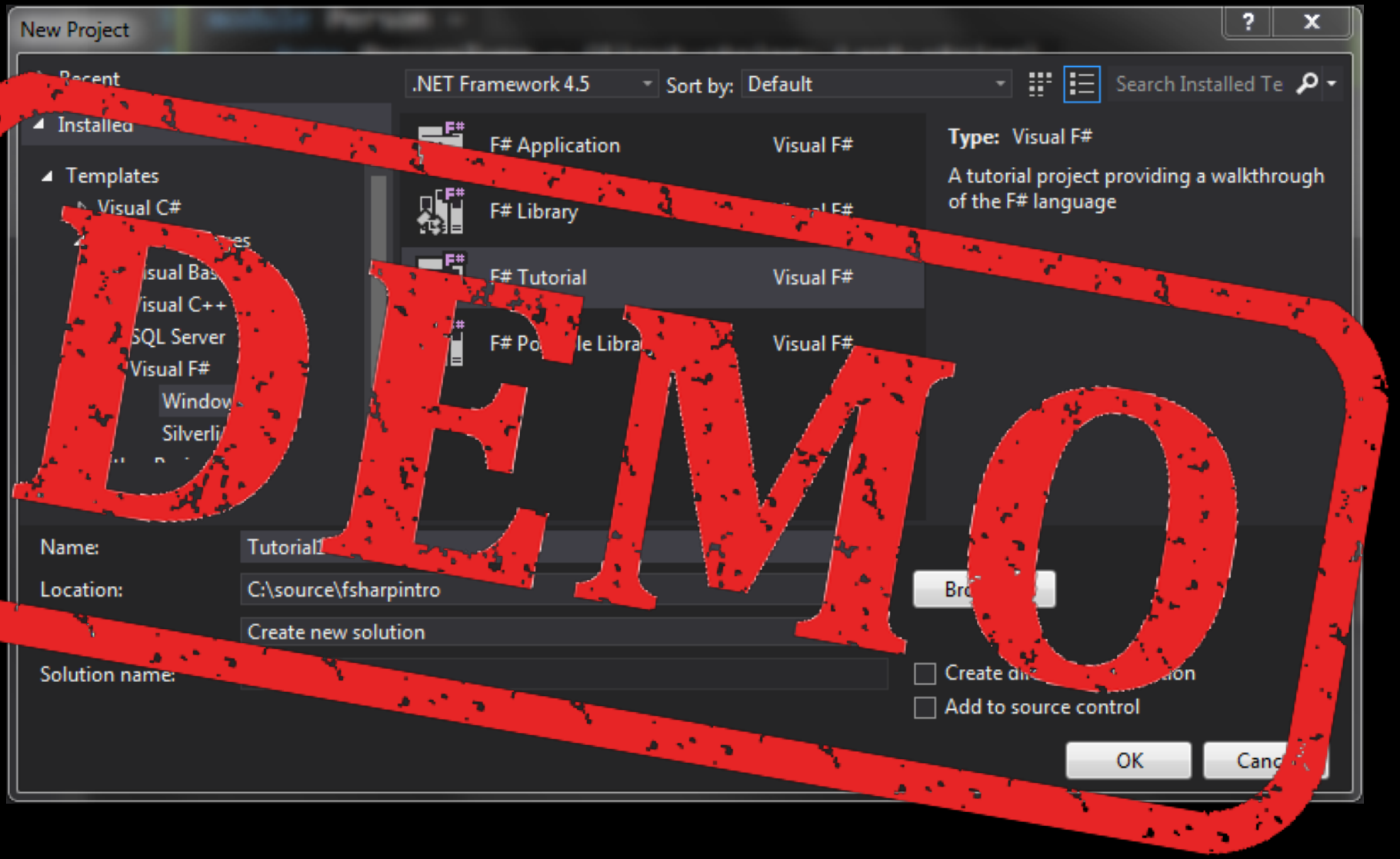
Agent Binding

```
let (>>=) x f = Async.bind f x
let pipeline x = Async.retn x >>= agent1 >>= agent2


let (>=>) f1 f2 x = f1 x >>= f2
let pipeline = agent1 >=> agent2

pipeline message
```

Lab: Agent Composition

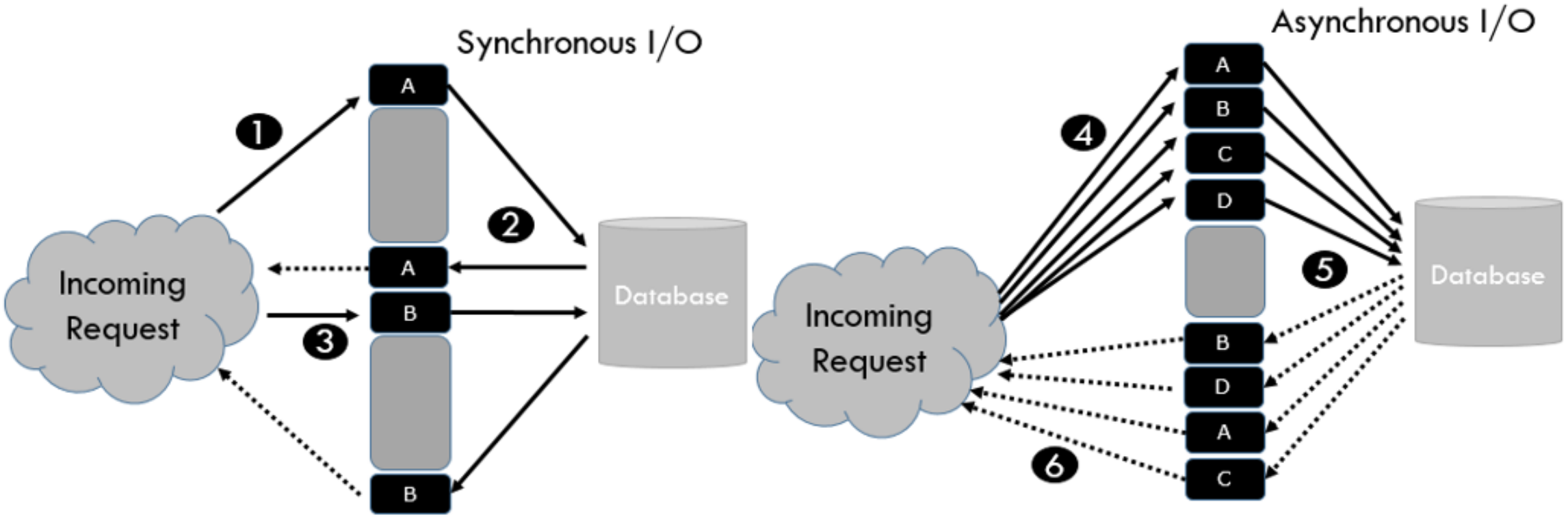
Agent Pipeline

```
let agent f = Agent<Message>.Start(fun inbox ->
    let rec loop () = async {
        let! msg, replyChannel = inbox.Receive()
        f msg
        replyChannel.Reply msg

        return! loop() }
    loop() )

let pipelineAgent f m =
    let a = agent f
    a.PostAndAsyncReply(fun replyChannel -> m, replyChannel)

let agent1 = pipelineAgent (printfn "Pipeline processing 1: %s")
let agent2 = pipelineAgent (printfn "Pipeline processing 2: %s")
```


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

Asynchronous Programming and Composition

Synchronous vs Asynchronous



“If you do nothing, you can scale infinitely.”

— Hanselman’s rule of scale

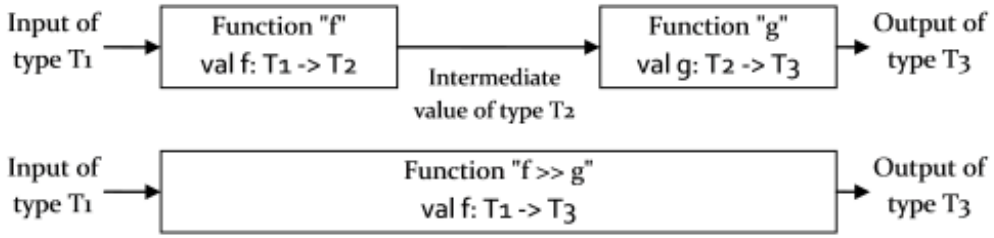
Its all about Scalability

```
let httpAsync (url : string) = async {
    let req = WebRequest.Create(url)
    let! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    return! reader.ReadToEndAsync() }

let sites =
    [   "http://www.live.com";"    “http://www.fsharp.org";
        "http://news.live.com";    "http://www.digg.com";
        "http://www.yahoo.com";    "http://www.amazon.com"
        "http://www.google.com";   "http://www.netflix.com";
        "http://www.facebook.com"; "http://www.docs.google.com";
        "http://www.youtube.com";  "http://www.gmail.com";
        "http://www.reddit.com";   "http://www.twitter.com"; ]

sites
|> Seq.map httpAsync
|> Async.Parallel
|> Async.Start
```

Function Composition

Function Composition - Building with composition. Composition is the 'glue' that allows us build larger systems from smaller ones. This is the very heart of the functional style. Almost every line of code is a composable expression. Composition is used to build basic functions, and then functions that use those functions, and so on.

Composition — in the form of passed parameters plus first-class functions



How can we compose Async?

```
Async<int> runOne () -> // ...
Async<decimal> runTwo(input : int) -> // ...

let result = runTwo(runOne()) // Error!!
```

F#

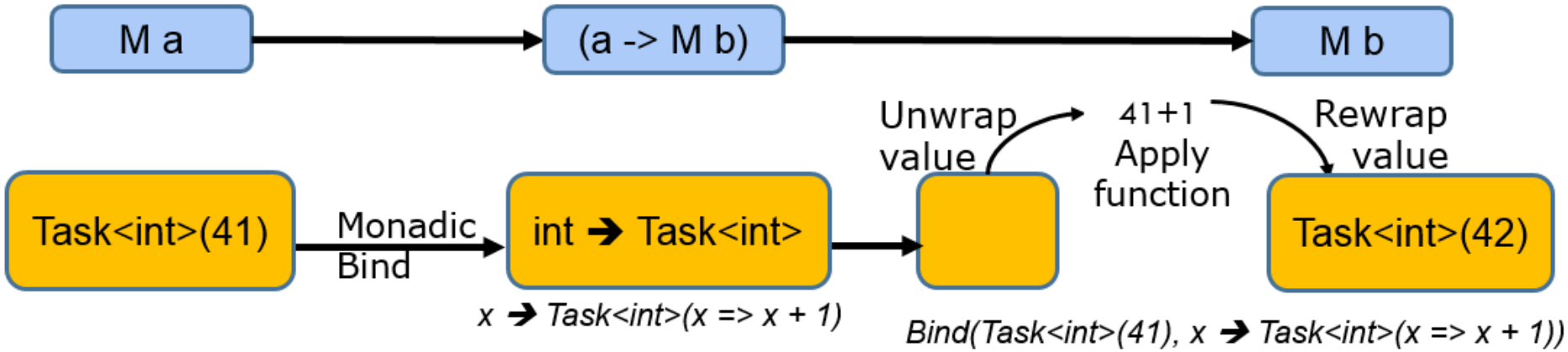
```
Async<int> runOne () -> // ...
Async<decimal> runTwo(input : int) -> // ...

let compose (f1 : a -> Async<b>) (f2 : b -> Async<c>) =
                f1 >> f2 // Error!!
```

F#

Composing Async (and Task)

Async<R> **Bind**<T, R>(Async<T> m, k : T -> Async<R>) …

Async<T> **Return**(T value) …

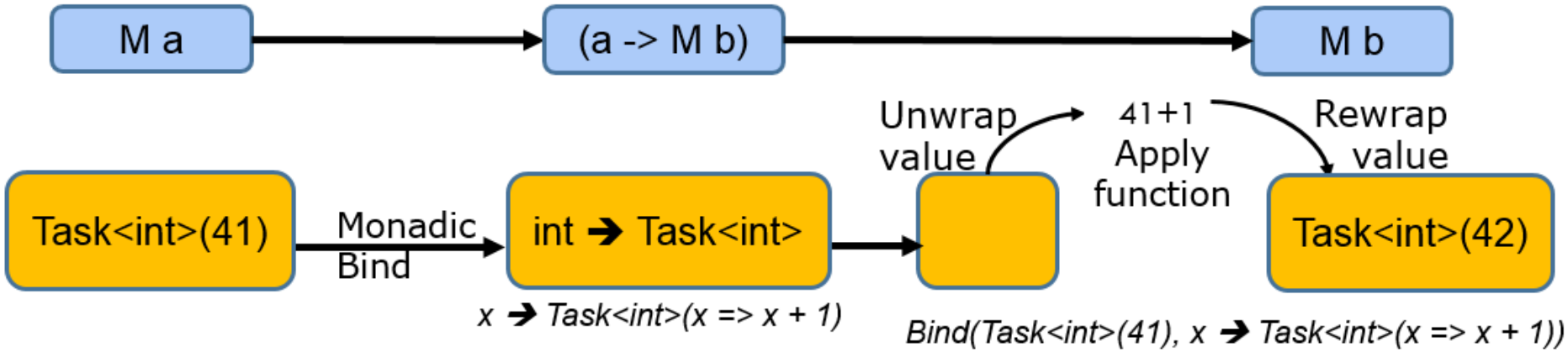


Functional Design Patterns – Monad

Generic type

With function (>>=): **bind** : M<‘a> -> (‘a -> M<‘b>) -> M<‘b>

With function: **return** : ‘a -> M<‘a>



Task/Async Bind and Return

```
// 'T -> M<'T>
static member Return (value : 'T) : Task<'T>

 // M<'T> * ('T -> M<'U>) -> M<'U>
static member Bind (input : Task<'T>, binder : 'T -> Task<'U>) : Task<'U>
```

```
// 'T -> M<'T>
static member Return (value : 'T) : Async<'T>

 // M<'T> * ('T -> M<'U>) -> M<'U>
static member Bind (input : Async<'T>, binder : 'T -> Async<'U>) : Async<'U>
```

How can we compose Async?

```
Async<int> runOne () -> // ...
Async<decimal> runTwo(input : int) -> // ...

let compose = async {
    let! res = runOne()
    return! runTwo res
}
```

F#

Kliesli operator in F#

```
fAsync:('a -> Async<'b>) -> gAsync:('b -> Async<'c>) -> arg:'a -> Async<'c>
```

```
let Kliesli (fAsync:'a -> Async<'b>) (gAsync:'b -> Async<'c>) (arg:'a) = ...
```

```
let (>=>) = Kliesli
```

```
let processStockHistory =
      asyncOne (_ -> Async<string>) >=> asyncTwo (string -> Async<int>)
```

Lab: Async Composition

Error handling in functional way

Async Futures - 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.

http://monkey.org/~marius/funsrv.pdf

Error handling in functional way

```
module Option =
    let ofChoice choice =
        match choice with
        | Choice1Of2 value -> Some value
        | Choice2Of2 _ -> None

module AsyncOption =
        type AsyncOption<'a> = Async<‘a Option>

        let handler (operation:Async<'a>) : AsyncOption<'a> =  ...
```

Error handling in functional way

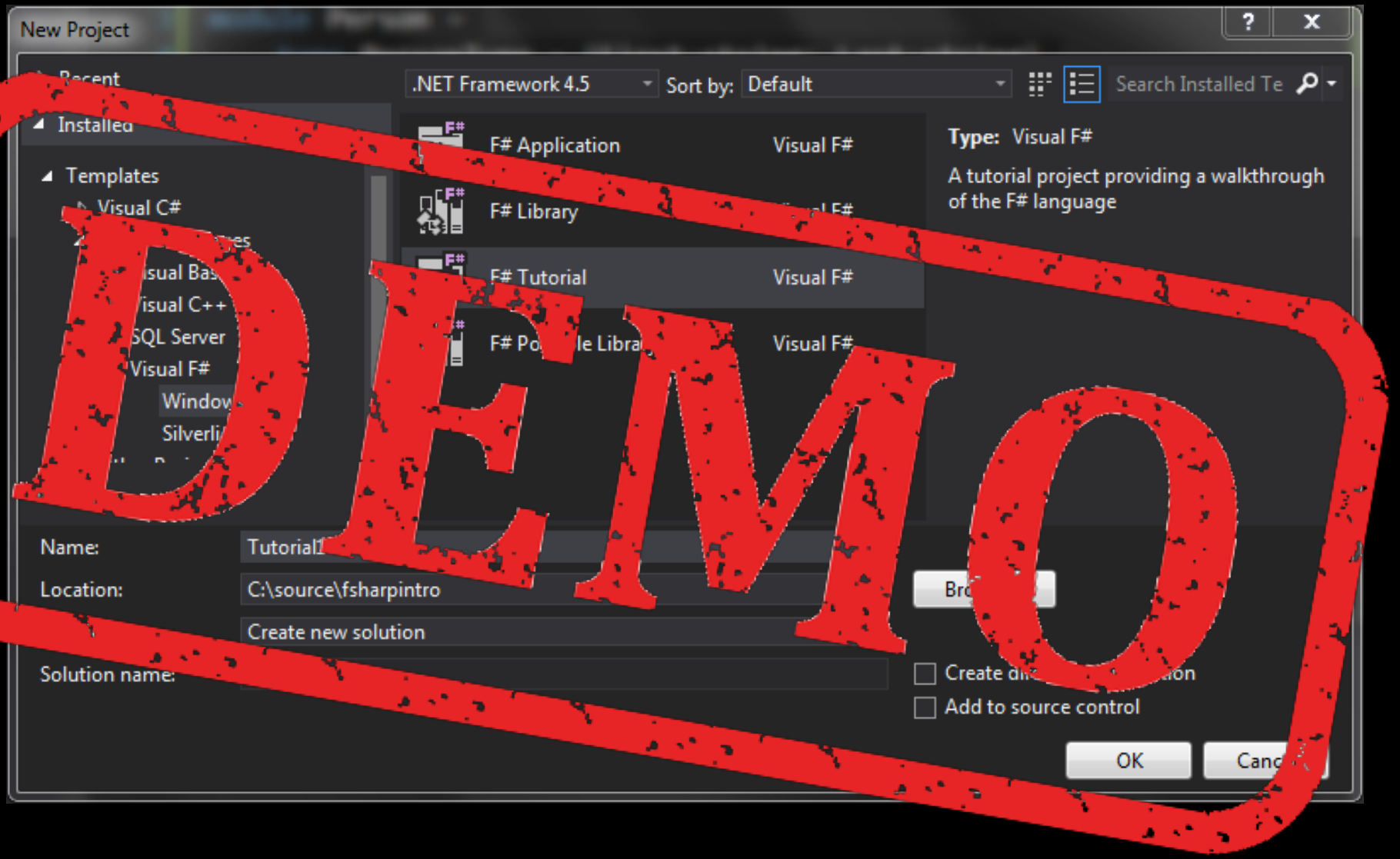
```
let downloadAsyncImage(blobReference:string) : Async<Image> = async {
        let! container = Helpers.getCloudBlobContainerAsync()
        let blockBlob = container.GetBlockBlobReference(blobReference)
        use memStream = new MemoryStream()
        do! blockBlob.DownloadToStreamAsync(memStream)
        return Bitmap.FromStream(memStream)
    }


downloadAsyncImage "Bugghina001.jpg"
|> AsyncOption.handler
|> Async.map(fun imageOpt ->
    match imageOpt with
    | Some(image) -> image.Save("ImageFolder\Bugghina.jpg")
    | None -> log "There was a problem downloading the image")
|> Async.Start
```

Preserving Exception semantic - Result Type

```
type Result<'TSuccess,'TFailure> =
  | Success of 'TSuccess
  | Failure of 'Tfailure

module Result =
        let ofChoice value =
            match value with
            | Choice1Of2 value -> Success value
            | Choice2Of2 e -> Failure e

module AsyncResult =
       type AsyncResult <'a> = Async<Result<‘a, exn>>

       let handler (operation:Async<’a>) : AsyncResult<’a> =  ...
```

AsyncResult monadic operators

```
module AsyncResult =
    let retn (value:'a) : AsyncResult<'a> =  value |> Ok |> async.Return

    let map (selector : 'a -> 'b) (asyncResult : AsyncResult<'a>) =  ...

    let bind (selector : 'a -> AsyncResult<'b>)
                          (asyncResult : AsyncResult<'a>) = ...
```


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

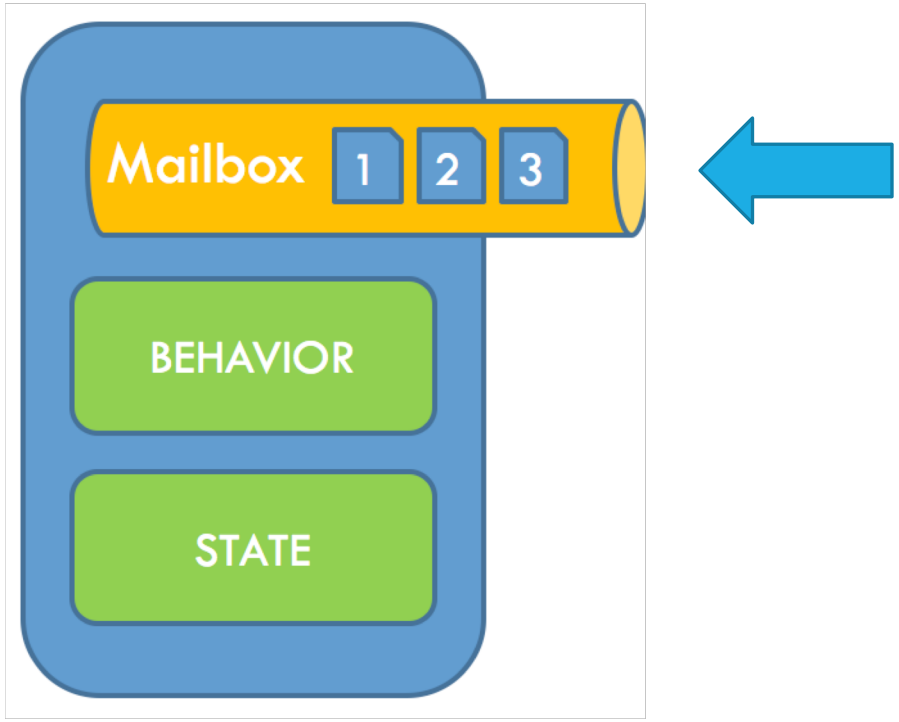
Agent & Reactive Extensions for Extensibility & Interoperability

Reactive Extension (Rx)
– Coordination and composition of event streams
– LINQ-basedAPI

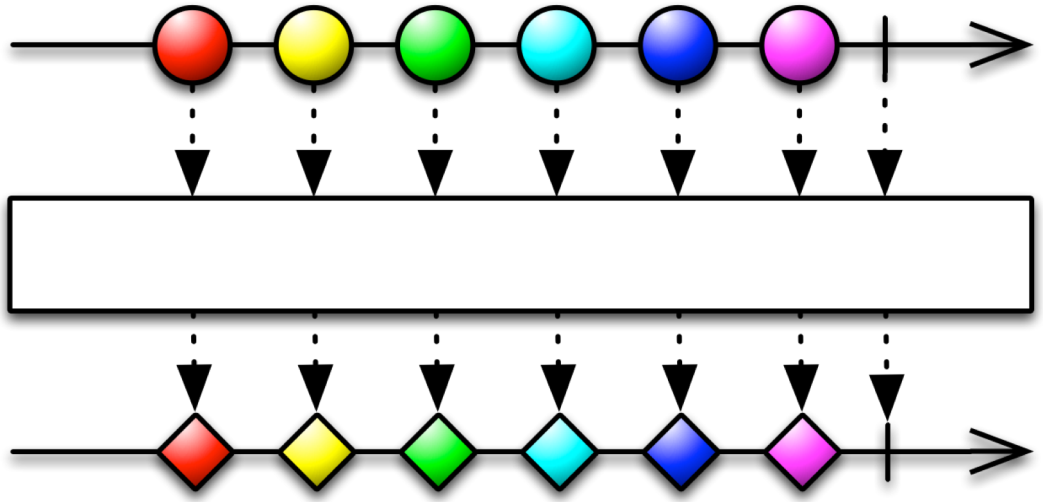
MailboxProcessor (MBP)
– Building blocks for message passing and parallelizing
– Explicit control over how data is buffered and moved

Streaming complex processing

Agent/Actor model



Reactive Extesions

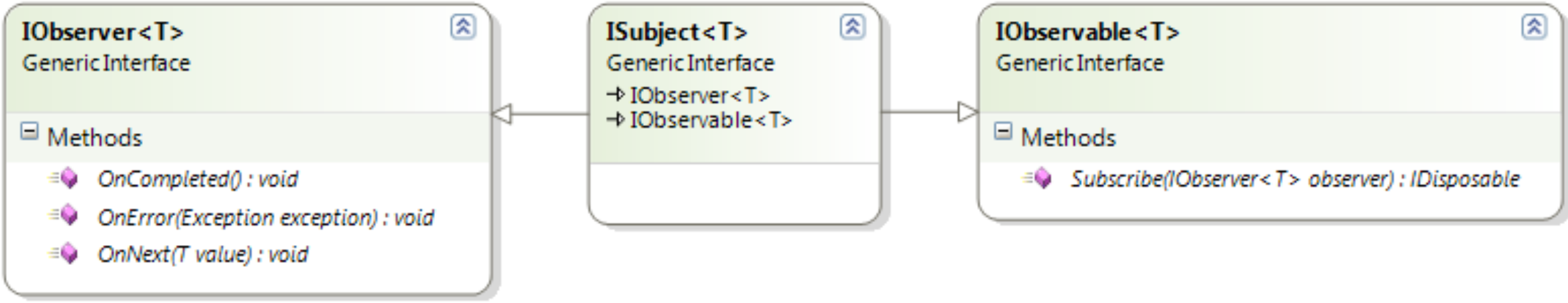


TPL Dataflow & Reactive Extension

Reactive Extensions

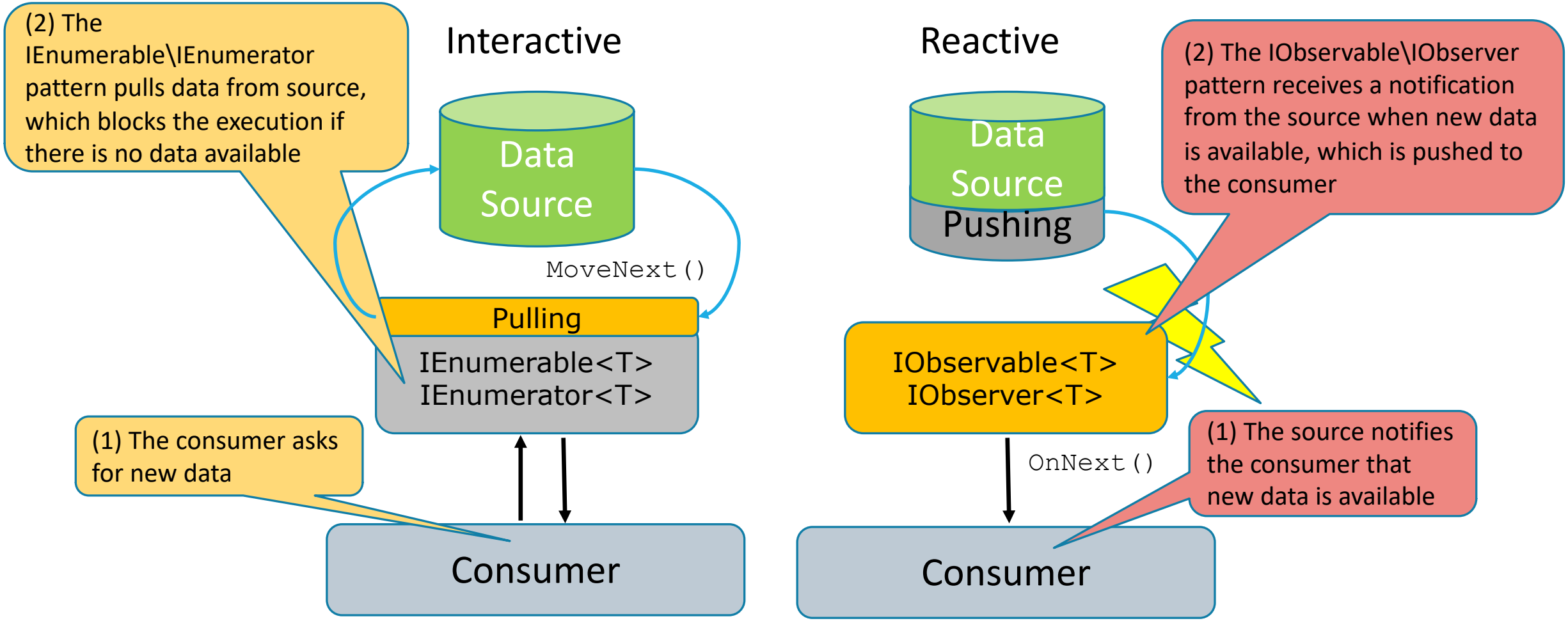
What are Reactive Extensions

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



IObserver & IObservable

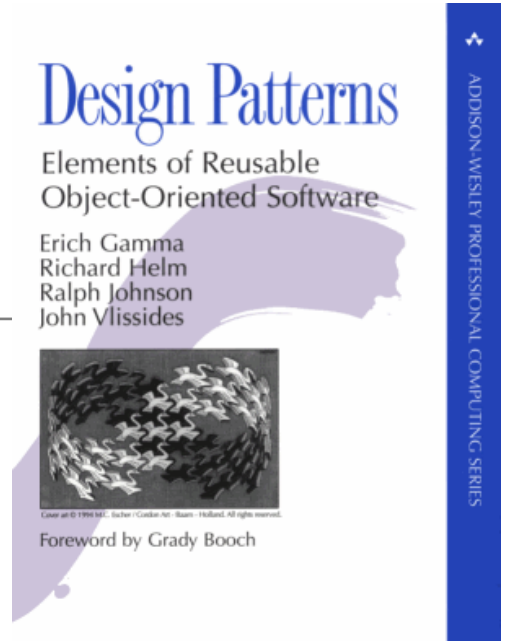
```
type IObserver<'a> = interface
    abstract OnCompleted : unit -> unit
    abstract OnError : exn -> unit
    abstract OnNext : 'a -> unit
end

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


(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

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)?
```

Stream processing basics – Combinators

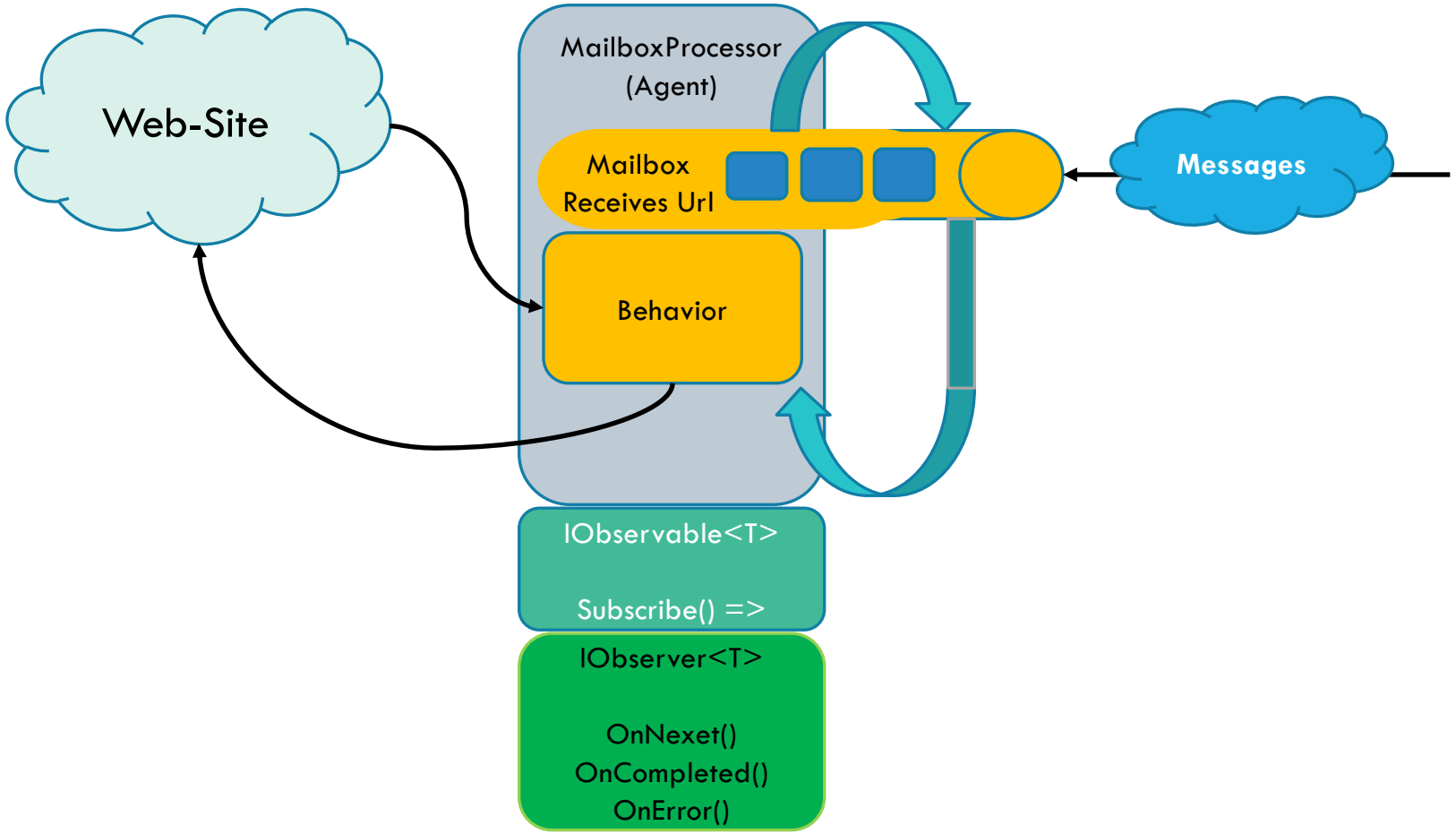
```
let observable = Observable
    .Interval(TimeSpan.FromSeconds(1.0))

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

Stream processing basics – Combinators

```
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)
```

Reactive Agents


Agent as Observable



MailboxProcessor 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();
```

Lab :
Reactive WebCrawler with Agents

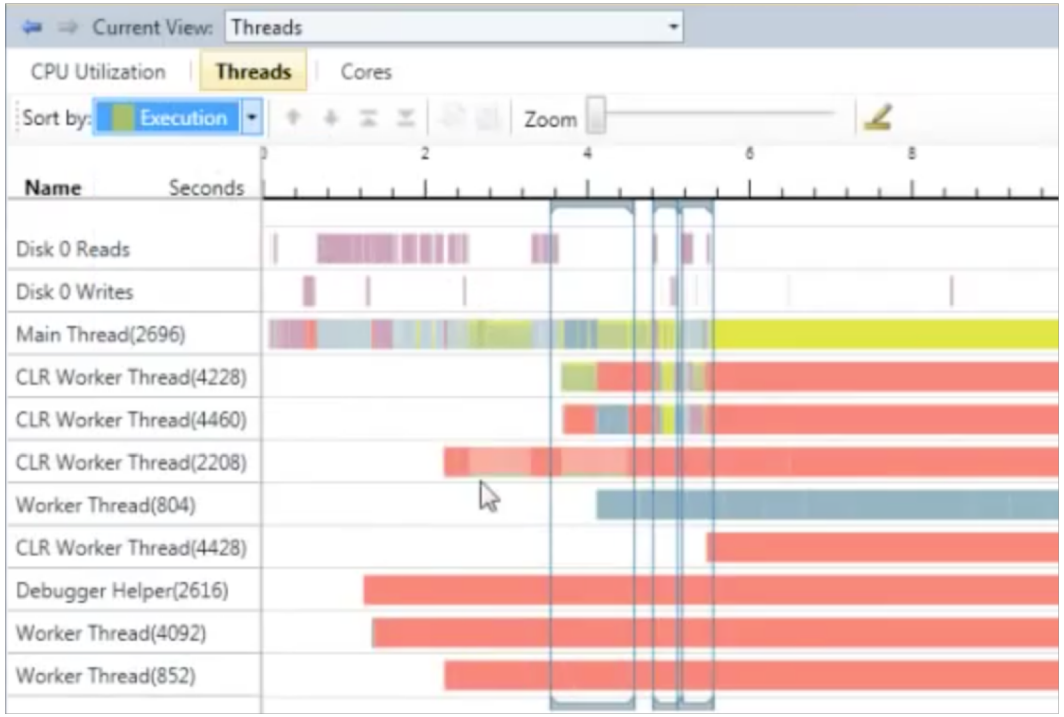
Summary

- The free lunch is over
- Applications must be built for concurrency
- Shared state concurrency is hard
- Agent can be compose and parallelize
- Alternative concurrency models
  - Message passing concurrency

Measure twice and cut once

Assume nothing – Profile everything

- Be scientific
  - Do test multiple implementations
  - Don’t set out to confirm your bias
  - Instrument and profile your code



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

-- Edsger Dijkstra

https://ti.to/luteceo-workshops/concurrent-functional-programming-in-net/en

LUTECEO workshops

Concurrent Functional Programming in .NET

April 25th–26th, 2019 Paris, France

29th - 30th April 2019 in London

https://skillsmatter.com/courses/589-functional-concurrency-in-dotnet-with-csharp-and-fsharp-with-riccardo-terrell