Reactive and Concurrent F#

& Workshop

"Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that non-determinism."

— *Edward A. Lee* *(The Problem with Threads, Berkeley 2006)*

Objectives

- Agents motivations
- What is Agents
- Agents vs Actors
- Agents patterns

What is an Actor?

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

Carl Hewitt’s Actor Model

The fundamental unit of computations that embodies:

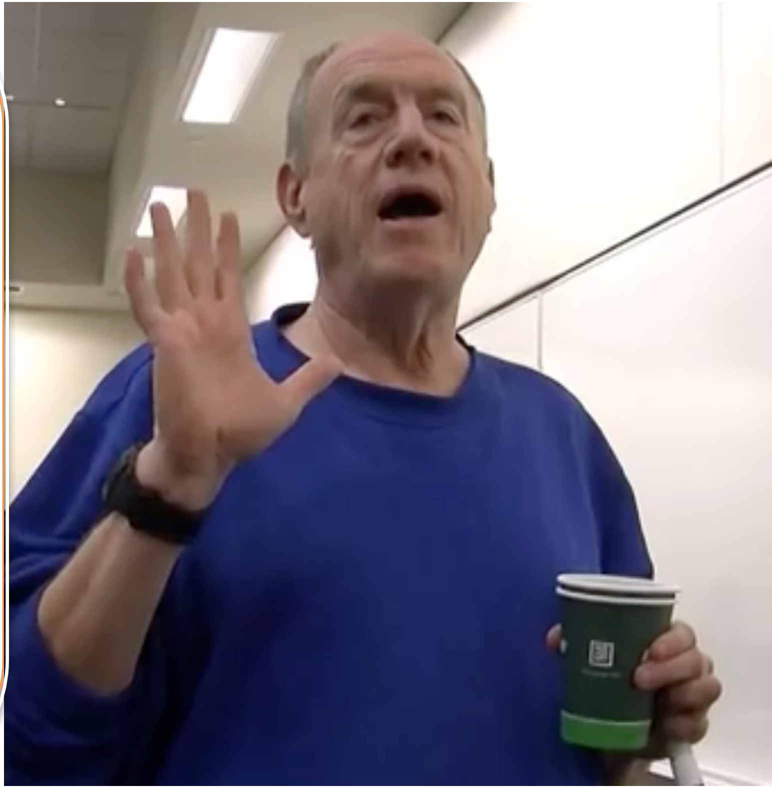
- Processing
- Storage
- Communication

Actor Model Three axioms:

1. Send messages to other Actors

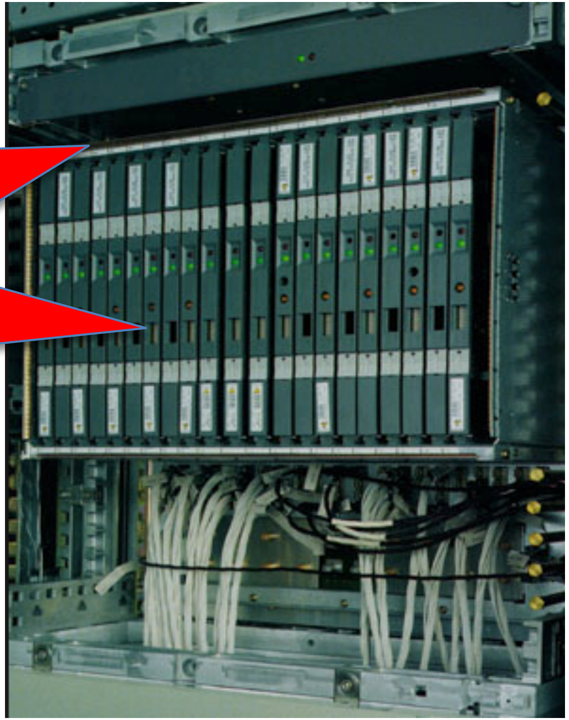
    - *One Actor is not Actor – bur great FSM*

2. Create other Actor
3. Decide how to handle next message

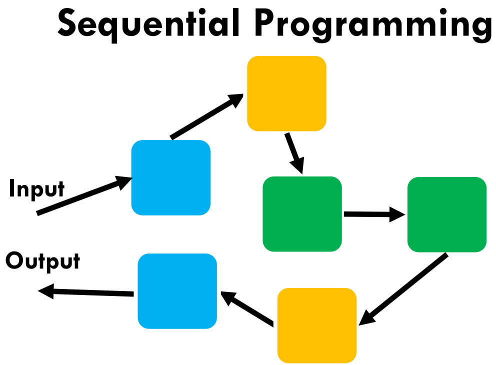


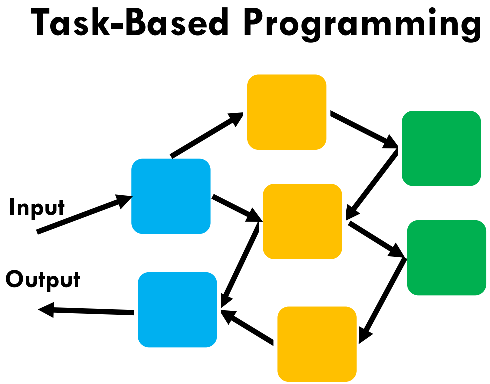
Ericson AXD 301 Switch – 1996

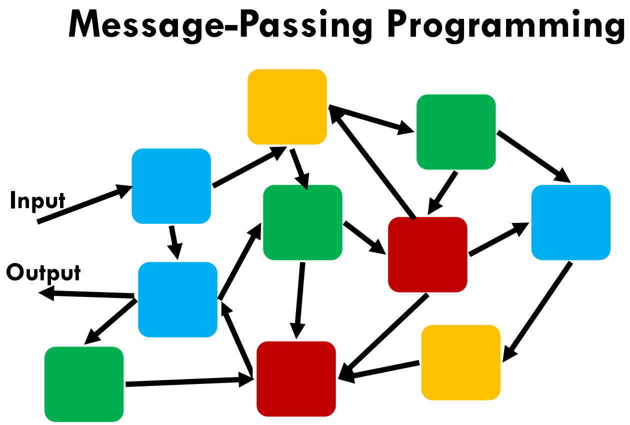
99.9999999 percent uptime

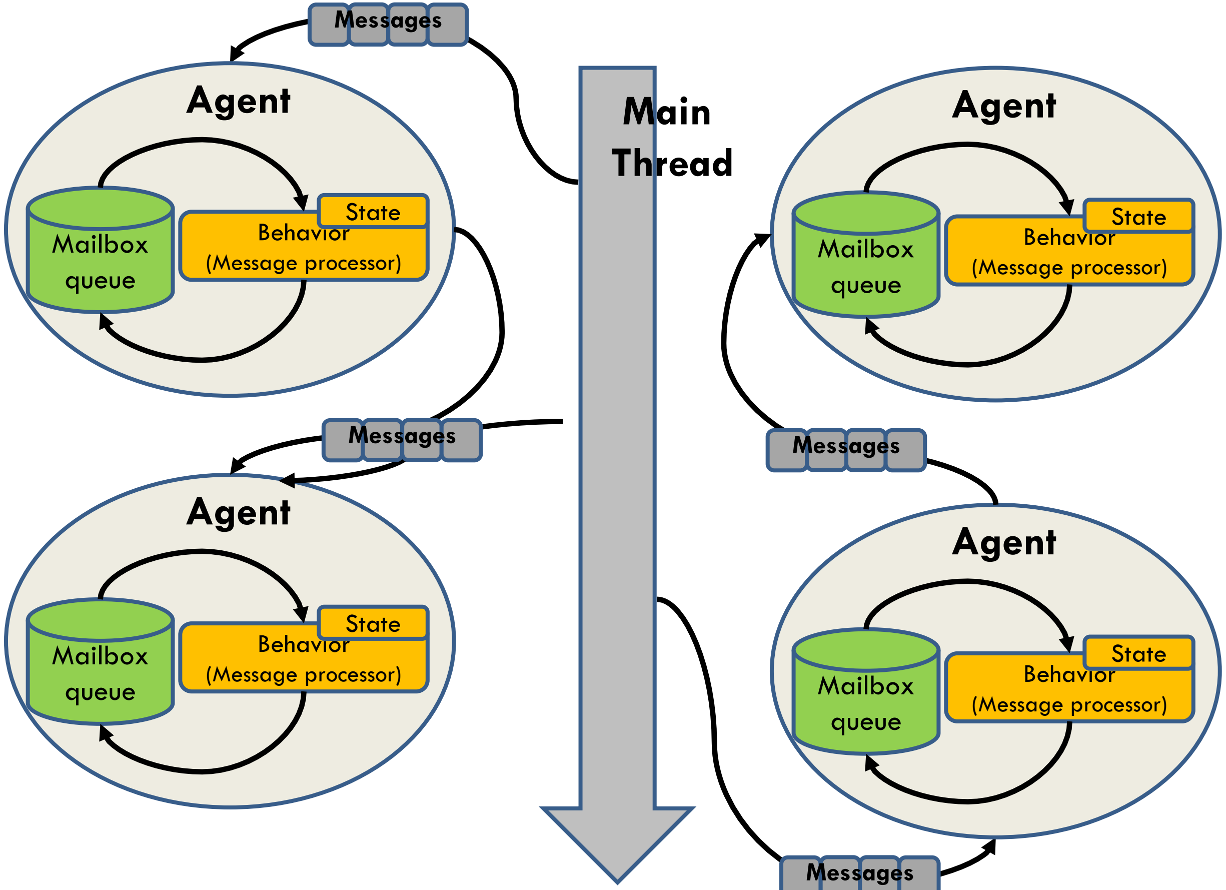


Comparison between Sequential, Task-based and Message passing programming








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

Agents motivations

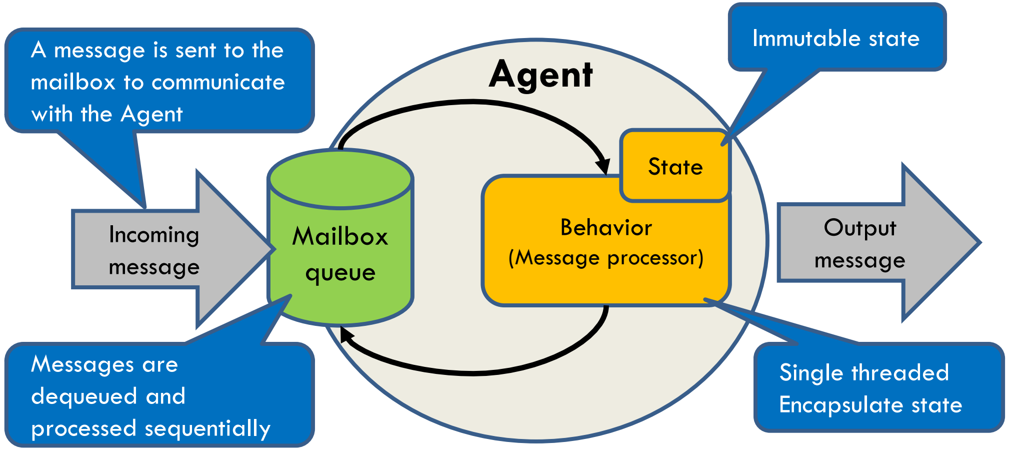
- You can manage **shared data** and resources **without locks.**
- You can easily follow the **SRP**, because each **agent** can be designed to **do only one thing.**
- It encourages a "pipeline" model of programming with "producers" sending messages to decoupled "consumers“
- It is straightforward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

What agents can do?

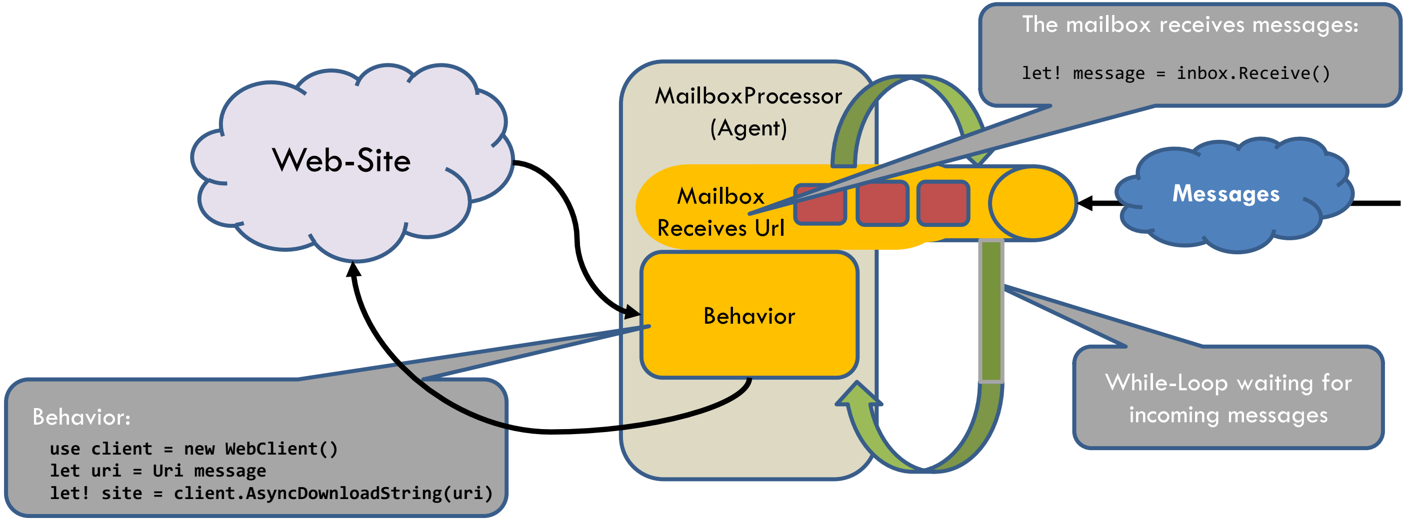
- Maintain **private state**
  - Accessed safely, can mutable or immutable
  - React to messages differently in different states

- Agents **perform actions**
  - Do calculations and update state
  - Notify other agents
  - Expose events that others can listen to
  - Send reply to the sender of a message

Agent anatomy



Agent anatomy



Simple F# Agent with while-loop

```
type Agent<'T> = MailboxProcessor<'T>

let agent =
  Agent<string>.Start(fun inbox -> async {
    while true do
      let! message = inbox.Receive()  //#B
      use client = new WebClient()
      let uri = Uri message
      let! site = client.AsyncDownloadString(uri)
      printfn "Size of %s is %d" uri.Host site.Length
    })

agent.Post "http://www.google.com"
agent.Post "http://www.microsoft.com”
```

F# Agent with async rec loop

```
let printerAgent = MailboxProcessor.Start(fun inbox->
    // 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
        return! messageLoop()
        }
    // start the loop
    messageLoop()
    )
```

Send message to agent

F#

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

Agent Replying to the sender

- **Message** carries input and a callback

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

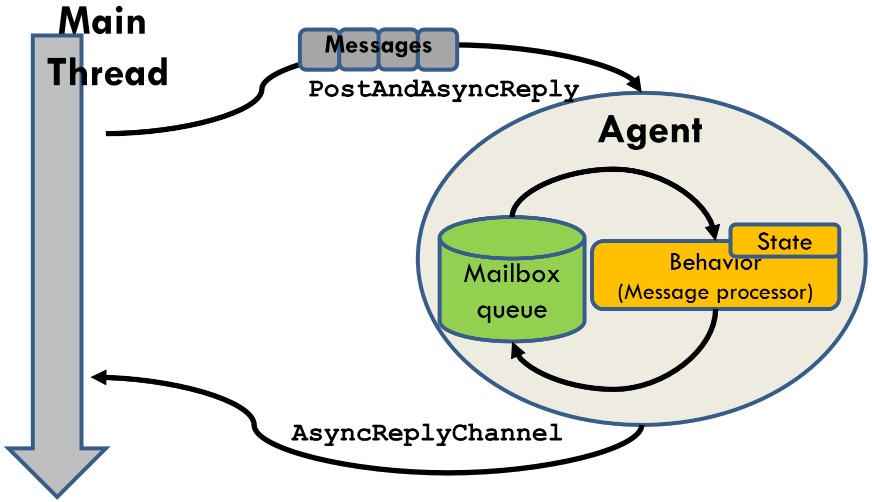
- **Reply** using the callback object

```
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 two-way communication



Send message to agent

F#

```
type StatsMessage = | Add of float | Clear | GetAverage of AsyncReplyChannel<float>

let stats = MailboxProcessor.Start(fun inbox ->
    let rec loop nums = async {
      let! msg = inbox.Receive()
      match msg with
      | Add num -> return! loop (num::nums)
      | GetAverage repl ->
          repl.Reply(List.average nums)
          return! loop nums
      | Clear -> return! loop [] }
    loop [] )
```

Send message to agent

F#

```
// Add error handler
stats.Error.Add(fun e -> printfn "Oops: %A" e)

// Post messages
stats.Post(Add(10.0))
stats.Post(Add(7.0))
stats.Post(Clear)
let average = stats.PostAndReply(GetAverage)
printfn "%A" average
```

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 [])
```

F# MailboxProcessor – aka Agent

```
let agent = MailboxProcessor<Message>.Start(fun inbox ->
          let rec loop n = async {
               let! msg = inbox.Receive()
               match msg with
               | Add(i) -> return! loop (n + i)
               | Get(r) -> r.Reply(n)
                           return! loop n }
          loop 0)
```

Anatomy of an Agent

```
let agent = Agent<_>.Start(fun mb ->
              let rec loop count = async {
                   let! msg = mb.Receive()
                   match msg with
                   | Add(n) -> return! (count + n)
                   | Get(reply) -> reply.Reply(count)
                   return! loop count }
              loop 0 )
agent.Post(Add(42))
```

Message passing using F# `MailboxProcessor`

Processors react to received messages

Immutability OR Isolation

```fsharp
let (lockfree:Agent<Msg<string,string>>) = Agent.Start(fun sendingInbox ->
    let cache = System.Collections.Generic.Dictionary<string, string>()
    let rec loop () = async {
        let! message = sendingInbox.Receive()
        match message with
            | Push (key,value) -> cache.[key] <- value
            | Pull (key,fetch) -> fetch.Reply cache.[key]
        return! loop ()
        }
    loop ())
```

Anatomy of an Agent

```
let agent =
        Agent<_>.Start(f
            let rec loop
                                              ount + n)
                                 r          eply(count)
                  etu    lo       ount }
            loop 0 )
```

Agent is not Actor

F# agent are not referenced by address but by **explicit instance**

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 Supervisors

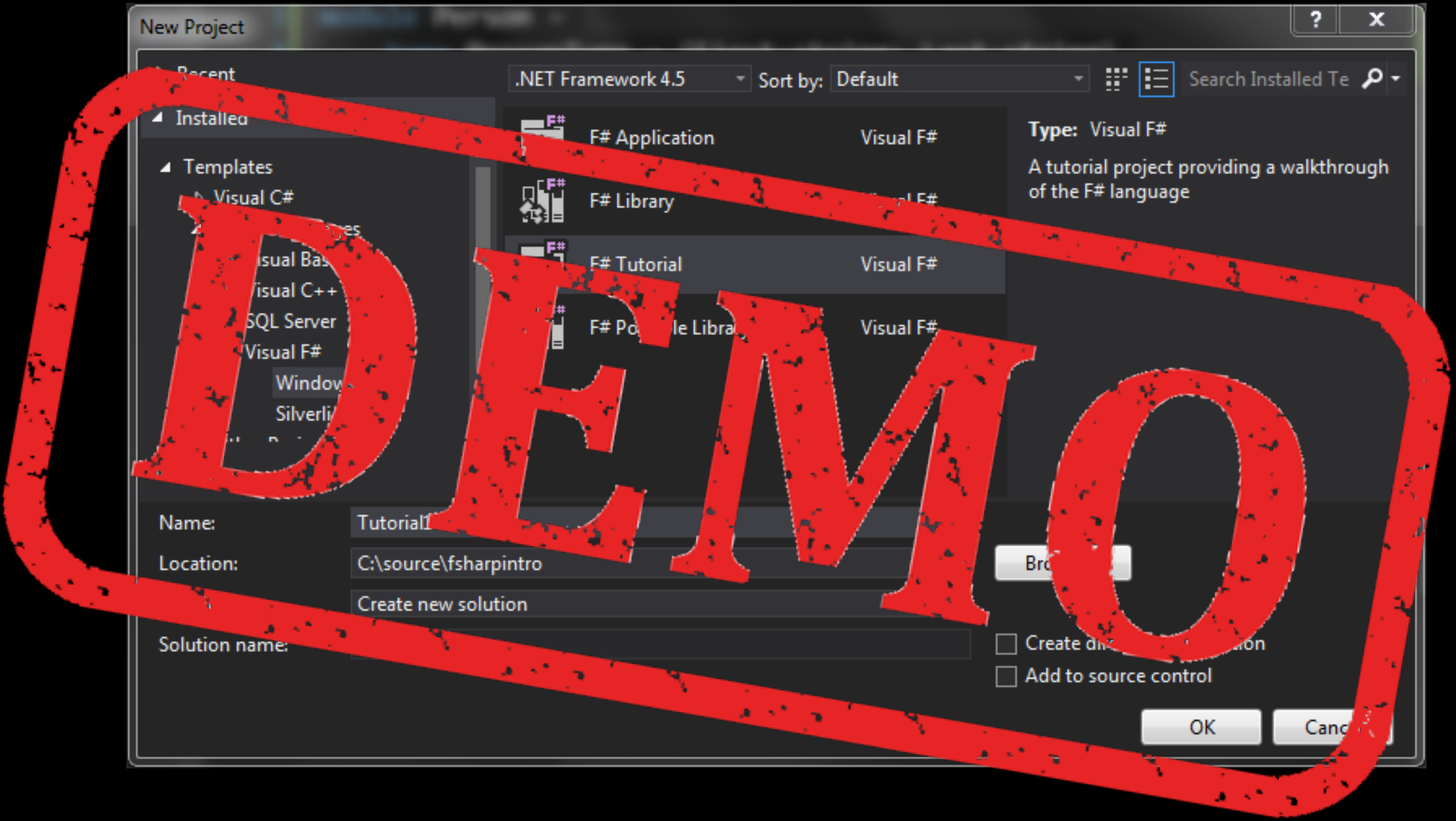
```
let errorAgent =
       Agent<int * System.Exception>.Start(fun inbox ->
         async { while true do
                    let! (agentId, err) = inbox.Receive()
                    printfn "an error '%s' occurred in agent %d" err.Message agentId })

let agents10000 =
       [ for agentId in 0 .. 10000 ->
            let agent =
                new Agent<string>(fun inbox ->
                   async { while true do
                              let! msg = inbox.Receive()
                              if msg.Contains("agent 99") then
                                  failwith "fail!" })
          agent.Error.Add(fun error -> errorAgent.Post (agentId,error))
            agent.Start()
            (agentId, agent) ]
```

Scaling agents on demand

```
let urlList = [ ("Microsoft.com", "http://www.microsoft.com/");
                ("MSDN", "http://msdn.microsoft.com/");
                ("Google", "http://www.google.com") ]

let processingAgent() = Agent<string * string>.Start(fun inbox ->
                        async { while true do
                                let! name,url = inbox.Receive()
                                let uri = new System.Uri(url)
                                let webClient = new WebClient()
                                let! html = webClient.AsyncDownloadString(uri)
                                printfn "Read %d characters for %s" html.Length name })

let scalingAgent : Agent<(string * string) list> = Agent.Start(fun inbox ->
                            async { while true do
                                    let! msg = inbox.Receive()
                                    msg
                                    |> List.iter (fun x ->
                                                  let newAgent = processingAgent()
                                                  newAgent.Post x )})
```

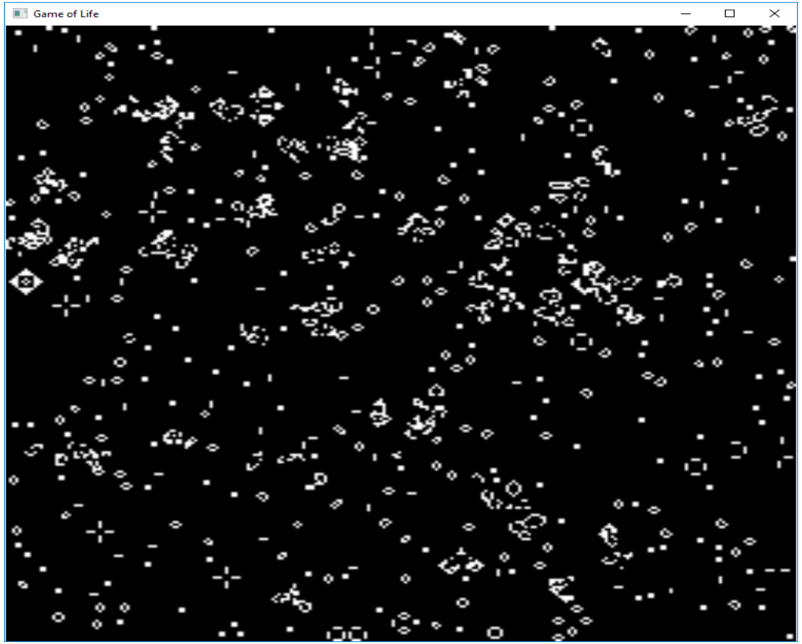
Async - StartWithContinuations

```
let runAgent = MailboxProcessor<Job>.Start(fun inbox ->
        let rec loop n =
            async {
                let! job = inbox.Receive()
                let str = sprintf "Starting job #%d" job.id
                match jobs.ParentId(job.id) with
                | Some id -> printAgent.Post <| sprintf "%s with parentId #%d" str id
                | None -> printAgent.Post str
                // Add the new job information to the list of running jobs.
                jobs.[job.id] <- { jobs.[job.id] with state = JobState.Running }

                Async.StartWithContinuations(job.comp,
                            (fun result -> completeAgent.Post(job.id, result)),
                            (fun _ -> ()),
                            (fun cancelException ->
                                    printAgent.Post <| sprintf "Canceled job #%d" job.id),
                            job.token)

                do! loop (n + 1)
                }
        loop (0))
```


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

Lab – Game of Life

Game of Life



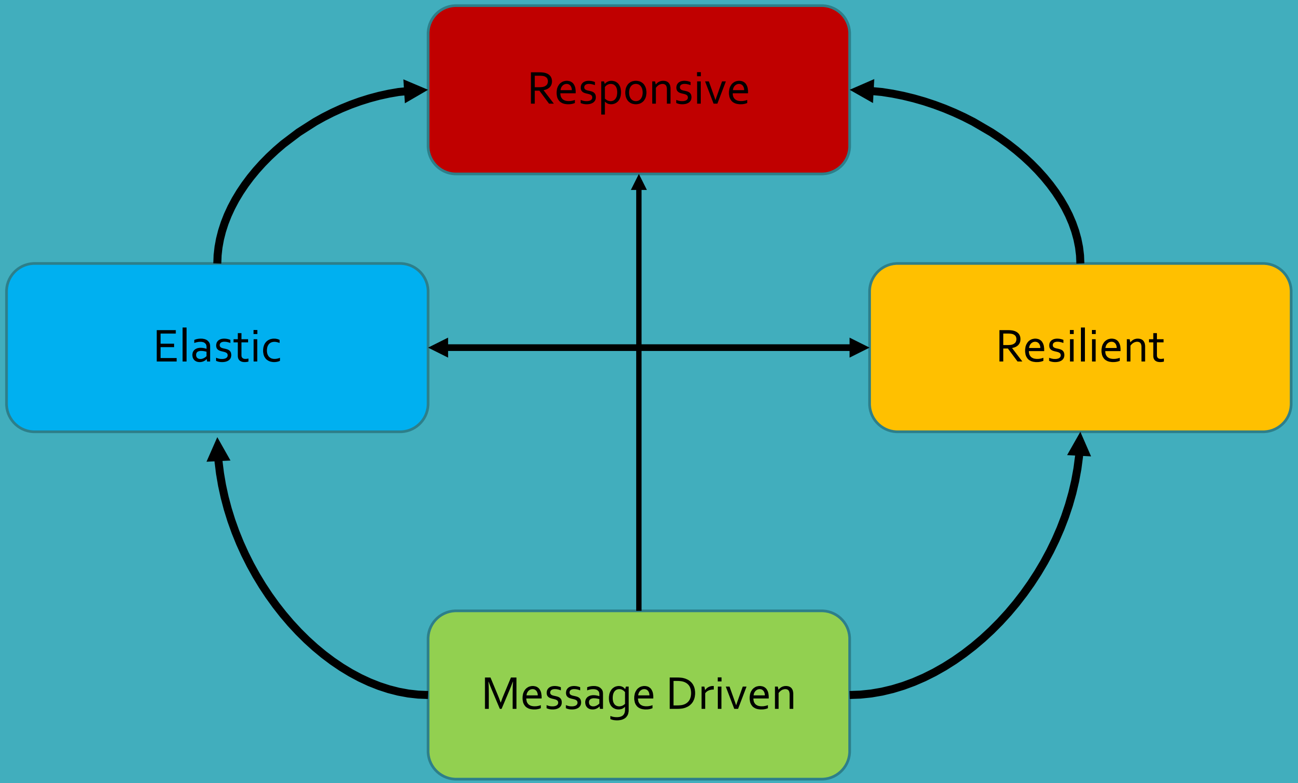
The Game of Life rules:

- 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.

Agent-based concurrency

- Programs **compose** 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

Reactive Manifesto

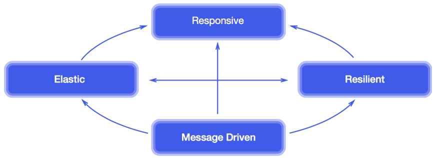


http://www.reactivemanifesto.org

Reactive Manifesto & Actor Model

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

Reactive Manifesto



Responsive

Message-Driven

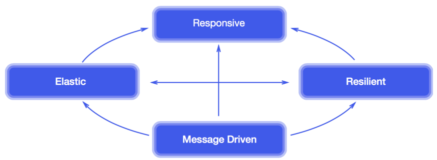
Resilient

Elastic

The **system responds in a timely manner** if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that **problems may be detected quickly** and dealt with effectively. Responsive systems focus on providing **rapid and consistent response times**, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behavior in turn simplifies error handling, builds end user confidence, and encourages further interaction.

http://www.reactivemanifesto.org

Reactive Manifesto

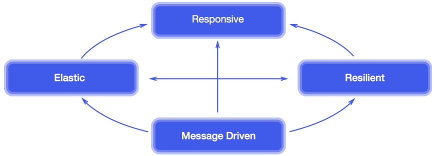


- Responsive
- **Message-Driven**
- Resilient
- Elastic

Reactive Systems rely on **asynchronous message-passing** to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing **enables load management, elasticity, and flow control** by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host.

http://www.reactivemanifesto.org

Reactive Manifesto

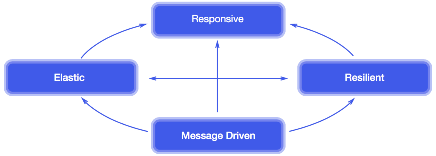


- Responsive
- Message-Driven
- **Resilient**
- Elastic

The system **stays responsive in the face of failure**. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. **Recovery of each component is delegated to another (external) component** and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

http://www.reactivemanifesto.org

Reactive Manifesto

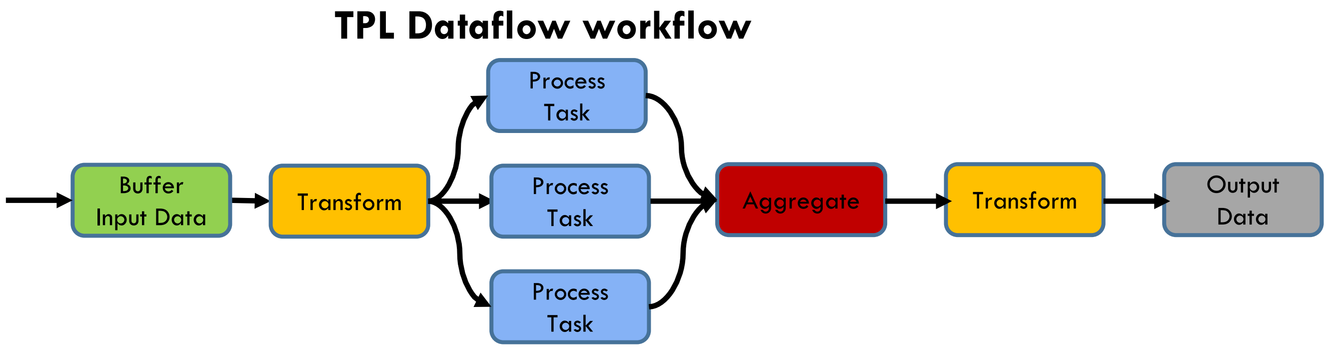


- Responsive
- Message-Driven
- Resilient
- **Elastic**

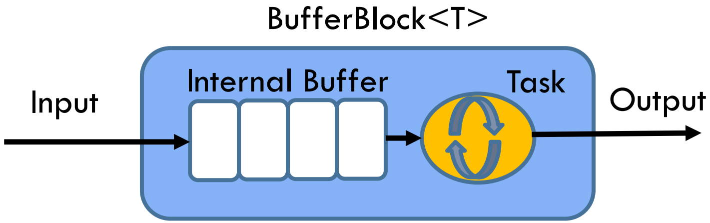
The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by **increasing or decreasing the resources allocated** to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, **scaling algorithms** by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

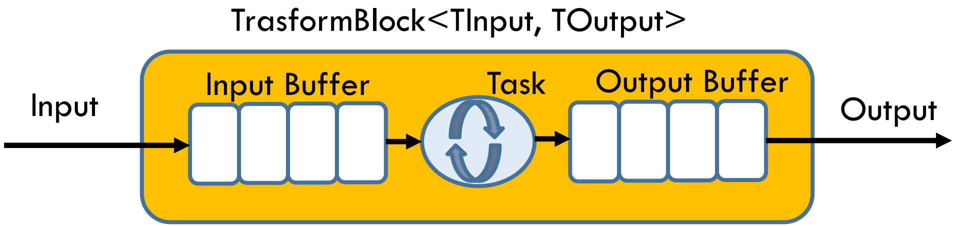
http://www.reactivemanifesto.org

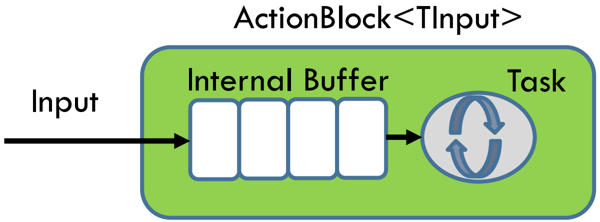
TPL DataFlow blocks –design to compose



Few block


BufferBlock<T>
Input
Internal Buffer
Task
Output

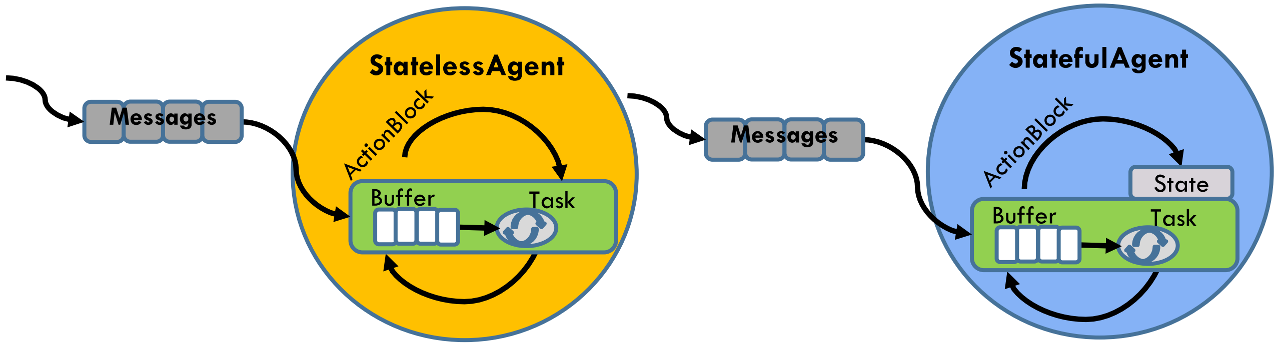

TrasformBlock<TInput, TOutput>
Input
Input Buffer
Task
Output Buffer
Output


ActionBlock<TInput>
Input
Internal Buffer
Task

Simple producer-consumer

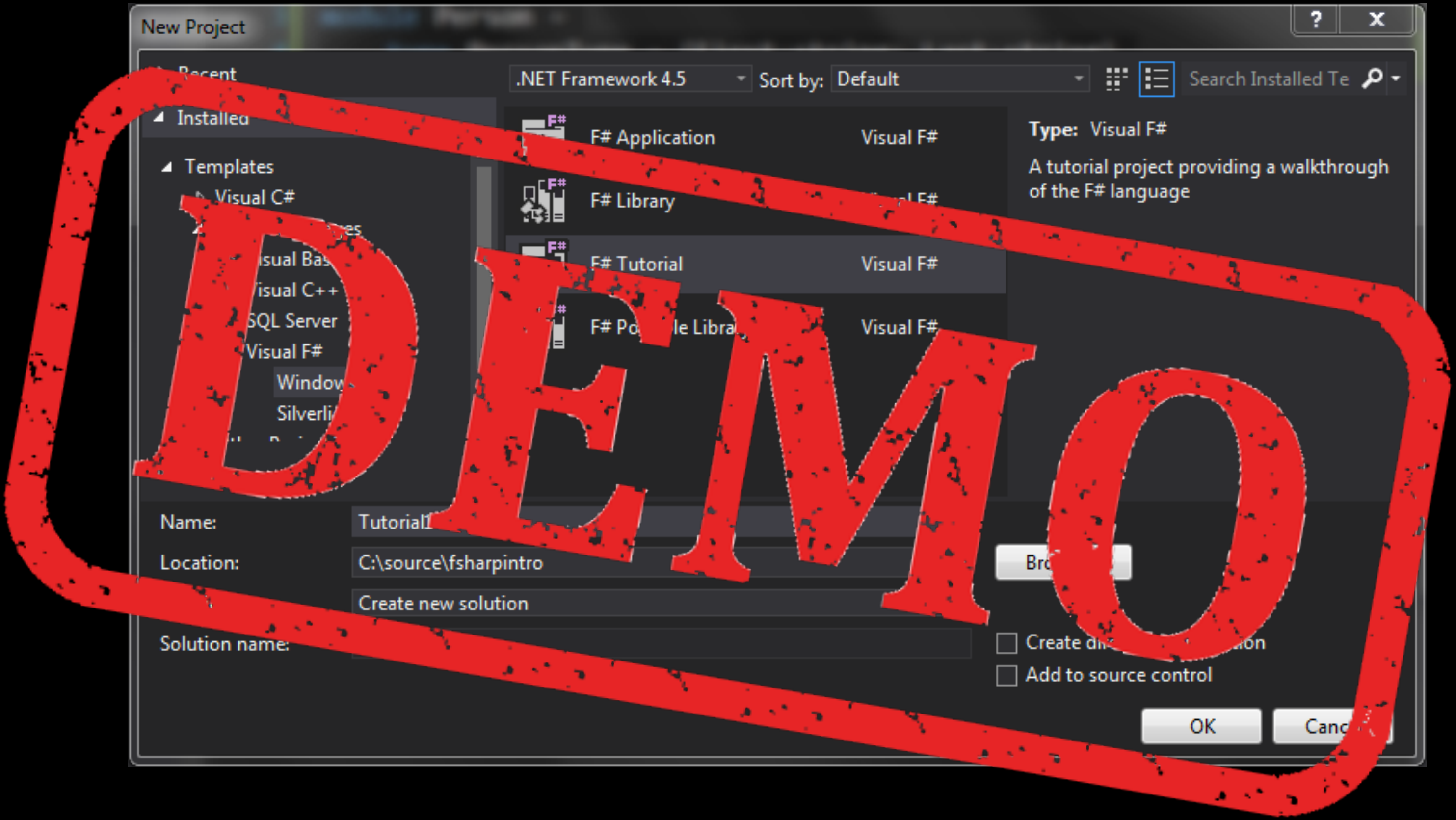
```
BufferBlock<int> buffer = new BufferBlock<int>();

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}")));
}
```

TPL DataFlow as Agent



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();
```


DEMO
New Project
Recent
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#
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

Lab - Implement a state full agent using TPL DataFlow

Cache web sites

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)

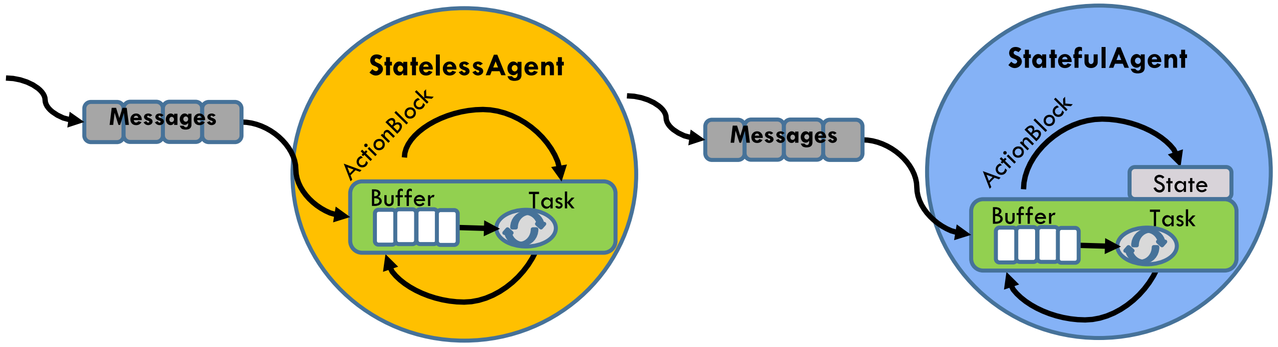
```
urls.Aggregate(ImmutableDictionary<string,string>.Empty,
       async (state, url) => {
if (!state.TryGetValue(url, out string content))
  using (var webClient = new WebClient())
  {
       content = await webClient.DownloadStringTaskAsync(url);
       await File.WriteAllTextAsync(createFileNamFromUrl(url),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

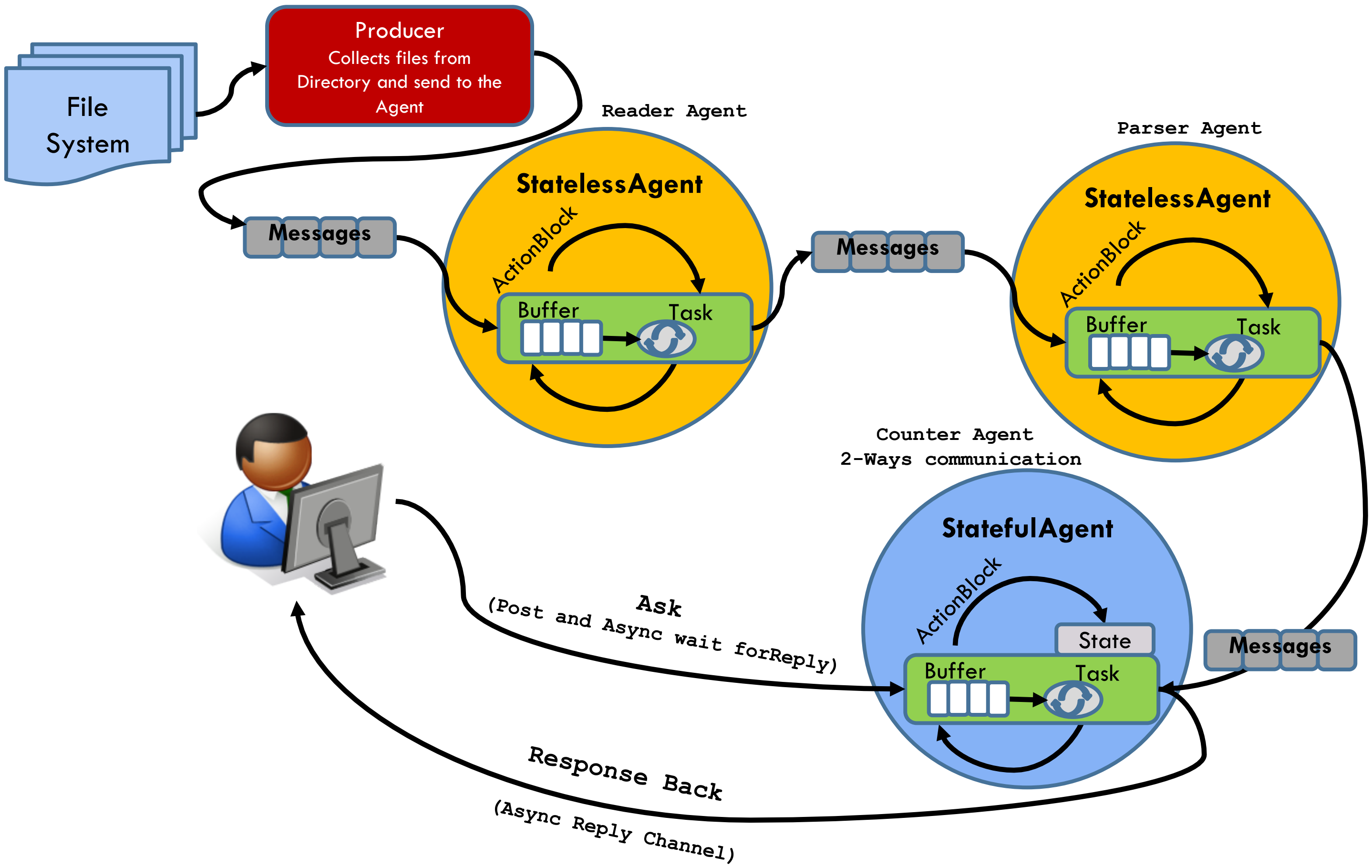
TPL DataFlow a statefull agent

```
class StatefulDataFlowAgent<TState, 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);
}
```

Lab - Agent

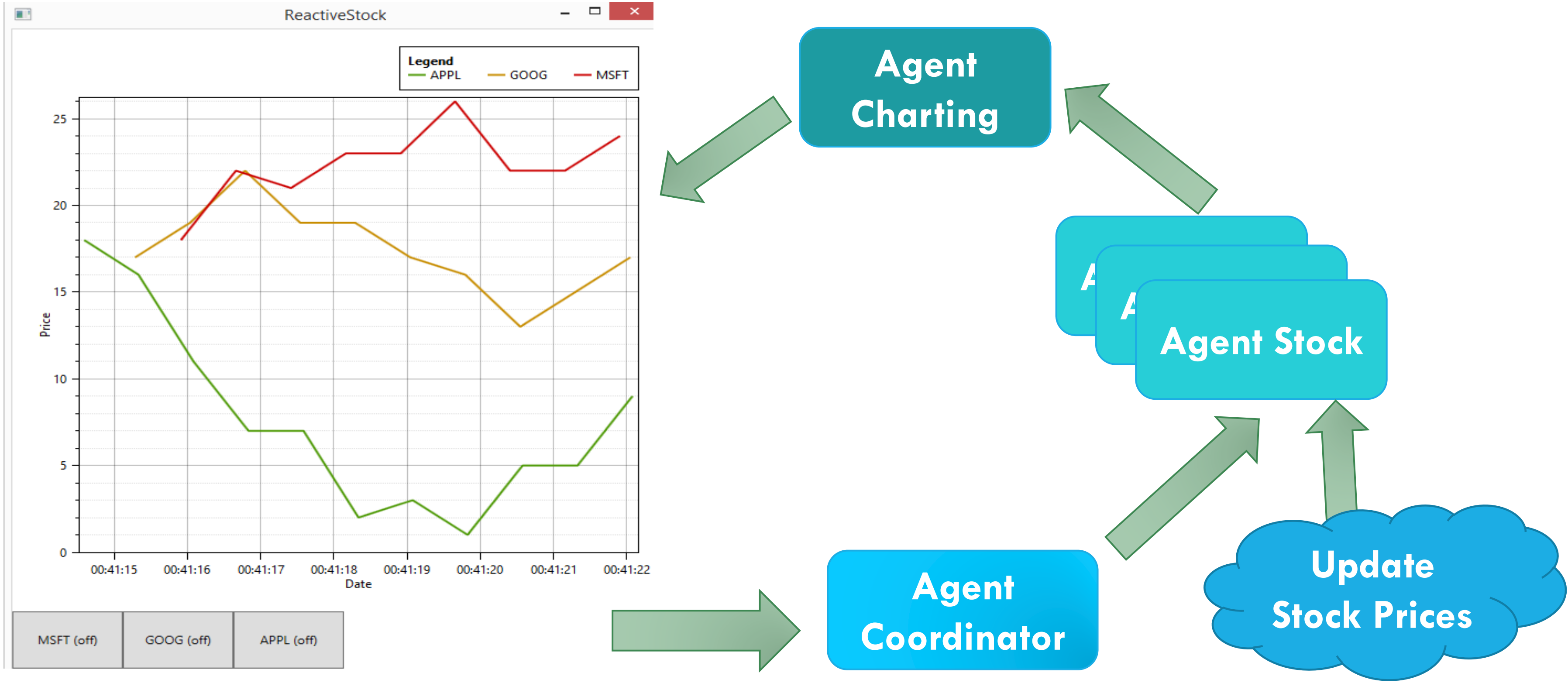
TPL DataFlow as Agent





File System
Producer
Collects files from Directory and send to the Agent
Reader Agent
StatelessAgent
ActionBlock
Buffer
Task
Messages
Messages
Parser Agent
StatelessAgent
ActionBlock
Buffer
Task
Counter Agent
2-Ways communication
StatefulAgent
ActionBlock
State
Buffer
Task
Messages
Ask
(Post and Async wait forReply)
Response Back
(Async Reply Channel)

Agent Stock Ticker



Agent Stock Ticker

Tasks

1. Create Agent hierarchy Children-Parent (sub-pub)
   1. Create Agent Stock (one per stock symbol)
   2. Create Agent Coordinator for subscribe/unsubscribe Agent Stocks
2. Connect Agents using messages

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)   …
```

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

-- Edsger Dijkstra