Message Passing Concurrency

BY RICCARDO TERRELL - @TRIKACE

Agenda

- Agents motivations
- What are Agents
- Agents vs Actors
- Agents patterns

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 programming

Program consists of agents

- **Lightweight** – can create lots of them
- Written using **asynchronous** workflows

Agents communicate via messages

- Communication is **thread-safe**
- Messages are **queued**

Enables parallelism

- **Different agents** vs. **multiple instances**

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

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”

What are Agents

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

Share Nothing, Async Message Passing

- Encourages shared-nothing process abstractions
  - mutable state - private
  - shared state - immutable
- Asynchronous message passing
- Pattern matching

Declaring messages

- Easier to reason about
- Higher abstraction level
- Easier to avoid
  - Race conditions
  - Deadlocks
  - Starvation
  - Live locks
- Distributed computing

Agents vs Actors

- Agent work in process
- Actors work also out of process
- Agent messages pass by reference
- Actor messages always serialized before sent

Actors / Agents

- React to received messages by executing a behavior function
  - can only change the state of the actor itself
  - can send messages to other actors
- Actors never share state and thus never need to compete for locks for access to shared data
- Actors exchange data by sending **immutable** messages
- Messages are sent asynchronously
- Actors do not block waiting for responses to their messages

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 [])
```

Declaring messages

Agents handle multiple messages

- Message type using discriminated union

```
type CacheMessage<'T> =
  | Add of string * 'T
  | Get of string * AsyncReplyChannel<option<'T>>
  | Reset
```

Safety guarantees

- Agent will be able to handle all messages

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	Expand and de-contract on demand

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#

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

Connecting agents

Applications consists of numerous agents

Agents communicate by sending messages

- Direct references are wrong!
- Difficult to follow, reuse and reconfigure

Decoupling of agents

- Expose events instead of sending messages
- Connect agents when creating them
- Use agent repository for dynamic configuration

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

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).

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 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 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 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 )})
```

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

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
```

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

Pipeline
with Bounded Agent

Pipeline with Bounded Agent

```
type PipelineAgent<‘a> =
abstract Add : ‘a -> unit
abstract Get : unit -> Async<‘a>
```

```
type internal BlockingAgentMessage<'T> =
  | Get of AsyncReplyChannel<'T>
  | Add of 'T * AsyncReplyChannel<unit>
```

Pipeline with Bounded Agent

```
type AsyncBoundedQueue<'T> (capacity: int) =
    let cancellationToken = defaultArg cancellationToken (new CancellationTokenSource())

    let rec balance() =
        if elts.Count > 0 && waitingConsumers.Count > 0 then
            elts.Dequeue() |> waitingConsumers.Dequeue(); balance()
        elif elts.Count < capacity && waitingProducers.Count > 0 then
            let x, reply = waitingProducers.Dequeue()
            elts.Enqueue x
            balance()
    let agent = MailboxProcessor.Start((fun inbox ->
            let rec loop() = async {
                    let! msg = inbox.Receive()
                    match msg with
                    | Enqueue(x, reply) -> waitingProducers.Enqueue (x, reply)
                    | Dequeue reply -> waitingConsumers.Enqueue reply
                    balance()
                    return! loop() }
            loop()), cancellationToken.Token)

    member __.AsyncEnqueue x =
          agent.PostAndAsyncReply (fun reply -> Enqueue(x, reply.Reply))
    member __.AsyncDequeue() =
          agent.PostAndAsyncReply (fun reply -> Dequeue reply.Reply)
```

Pipeline with Bounded Agent

```
let scalePipelinedImages = async {
    while not cts.IsCancellationRequested do
        let! info = loadedImages.AsyncDequeue()
        let! info = scaleImage info
        do! scaledImages.AsyncEnqueue(info) }
```

>==>

```
let filterPipelinedImages = async {
    while not cts.IsCancellationRequested do
        let! info = scaledImages.AsyncDequeue()
        for filter in filters do
            let! imageFiltered = filterImage info filter
            do! filteredImages.AsyncEnqueue(imageFiltered) }
```

Workshop:
Pipeline with Bounded Agent

TPL Dataflow Agent in .NET and C#

TPL DataFlow blocks –design to compose

for High throughput, low-latency scenarios

TPL DataFlow blocks –design to compose

for High throughput, low-latency scenarios

What does TPL Dataflow provides

- provides a foundation for message passing and parallelizing CPU-intensive and I/O-intensive applications
- gives you explicit control over how data is buffered and moves around the system
- improve responsiveness and throughput by efficiently managing the underlying threads
- allows you to easily create a mesh through which your data flows
- meshes can split and join the data flows, and even contain data flow loops
- allows to create custom blocks and extend functionality

Some Dataflow blocks

Building a Dataflow Network

TPL Dataflow Parallelism

```
var actionBlock = new ActionBlock<int>((i) => {
Console.WriteLine($"{Thread.CurrentThread.ManagedThreadId}\t{i}");
});
```

```
for (var i = 0; i < 10; i++) actionBlock.Post(i);
```

```
actionBlock.Options = new ExecutionDataflowBlockOptions() {
      MaxDegreeOfParallelism = 4 }
 );
 for (var i = 0; i < 10; i++) actionBlock.Post(i);
```

BufferBlock<>

ActionBlock<TInput>

ActionBlock<TInput>

Multiple Blocks

MaxDegreeOfParallelism

MaxDegreeOfParallelism = 2

TransformBlock<TInput, TOutput>

TransformManyBlock<TInput, TOutput>

TransformBlock + ActionBlock

Mandelbrot Pipeline

BatchBlock<>

JoinBlock<>

BroadcastBlock<>

WriteOnceBlock<>

TPL Dataflow Blocks

Buffering	Execution	Joining
BufferBlock	ActionBlock	BatchBlock
BroadcastBlock	TransformBlock	JoinBlock
WriteOnceBlock	TransformManyBlock	BatchedJoinBlock

Simple producer-consumer

```
static 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 throttling tasks

```
async Task LotsOfWorkAsync() {
   var throttle = Throttle<string>(async message => {
      await Task.Yield();
      Console.WriteLine(message);}, maxParallelism: Environment.ProcessorCount);
    throttle.Post("lots"); throttle.Post("of"); throttle.Post("work");
    throttle.Complete(); // Signal that we're done enqueuing work.
    await throttle.Completion;
}

static ITargetBlock<T> Throttle<T>(Func<T, Task> worker, int max) {
   var opts = new ExecutionDataflowBlockOptions() {MaxDegreeOfParallelism = max};
   return new ActionBlock<T>(worker, opts);
}
```

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();
```

TPL Dataflow Pipeline Processor

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

Lab :
Implement TPL DataFlow Pipeline

Developing Custom Dataflow Blocks

```
class SlidingWindowBlock<T> : IPropagatorBlock<T,T[]> {
    private readonly int m_windowSize;
    private readonly ITargetBlock<T> m_target;
    private readonly ISourceBlock<T> m_source;

    public SlidingWindowBlock(int windowSize) {
        var queue  = new Queue<T>();
        var source = new BufferBlock<T[]>();
        var target = new ActionBlock<T>(item =>
        {
            queue.Enqueue(item);
            if (queue.Count >  m_windowSize) queue.Dequeue();
            if (queue.Count == m_windowSize) source.Post(queue.ToArray());
        });
        m_windowSize = windowSize;
        m_target = target;
        m_source = source;
    }
```

Agent anatomy

TPL DataFlow as Agent

Agent anatomy

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

Lab :

Implement 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;
});
```

request
request
request
request
request
Bottleneck
request
request
Multiple application requests for Database
Contention due to a limited number of available connections, causing waits and queuing of the requests
The number of concurrent requests is reduced

Multiple Application requests for Database

The Agent (MailboxProcessor) is buffering and enqueing the incoming requests ensuring a granular controller over the Database connection pool

The number of concurrent requests is tamed

Lab :
Agent Stock Ticker

Agent Stock Ticker

Agent Stock Ticker

Tasks:

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

Agent for producer and consumer pattern

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)

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

Streaming complex processing

Streaming complex processing

Agent/Actor model

Reactive Extesions

TPL DataFlow & Reactive Extensions for Extensibility & Interoperability

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

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

TPL Dataflow & Reactive Extension

Reactive Extensions

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();
```

Lab :

TPL DataFlow & Reactive Extensions

Channel

Channel

Channel

Recv
Subscribe
Behavior
1
2
3
4
5
Send
Send
Send
Send
Recv
Subscribe
Behavior
Recv
Subscribe
Behavior
Recv
Subscribe
Behavior
Recv
Subscribe
Behavior

```fsharp
type [<Sealed>] Channel<'T>() =
    let channelAgent =
        MailboxProcessor<ChannelMsg<'T>>.Start(fun inbox ->
            let readers = Queue()
            let writers = Queue()
            let rec loop () = async {
                let! msg = inbox.Receive()

                match msg with
                | Read ok when writers.Count = 0 ->
                    readers.Enqueue ok
                    return! loop()
                | Read ok -> let value, cont = writers.Dequeue()
                             Pool.Spawn cont
                             ok value
                             return! loop()
                | Write (x,ok) when readers.Count = 0 ->
                          writers.Enqueue(x, ok)
                          return! loop()
                | Write (x,ok) ->  let cont = readers.Dequeue()
                                   Pool.Spawn ok
                                   cont x
                                   return! loop() }
            loop() )

    member this.Read(read) = channelAgent.Post (Read read)
    member this.Write(v, ok) = channelAgent.Post (Write (v,ok))

    member inline this.Read() = Async.FromContinuations(fun (ok, _, _) -> this.Read ok)
    member inline this.Write x = Async.FromContinuations(fun (ok, _, _) -> this.Write(x,ok))
```

Lab :
Channel in action (broadcasting)

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

-- Edsger Dijkstra