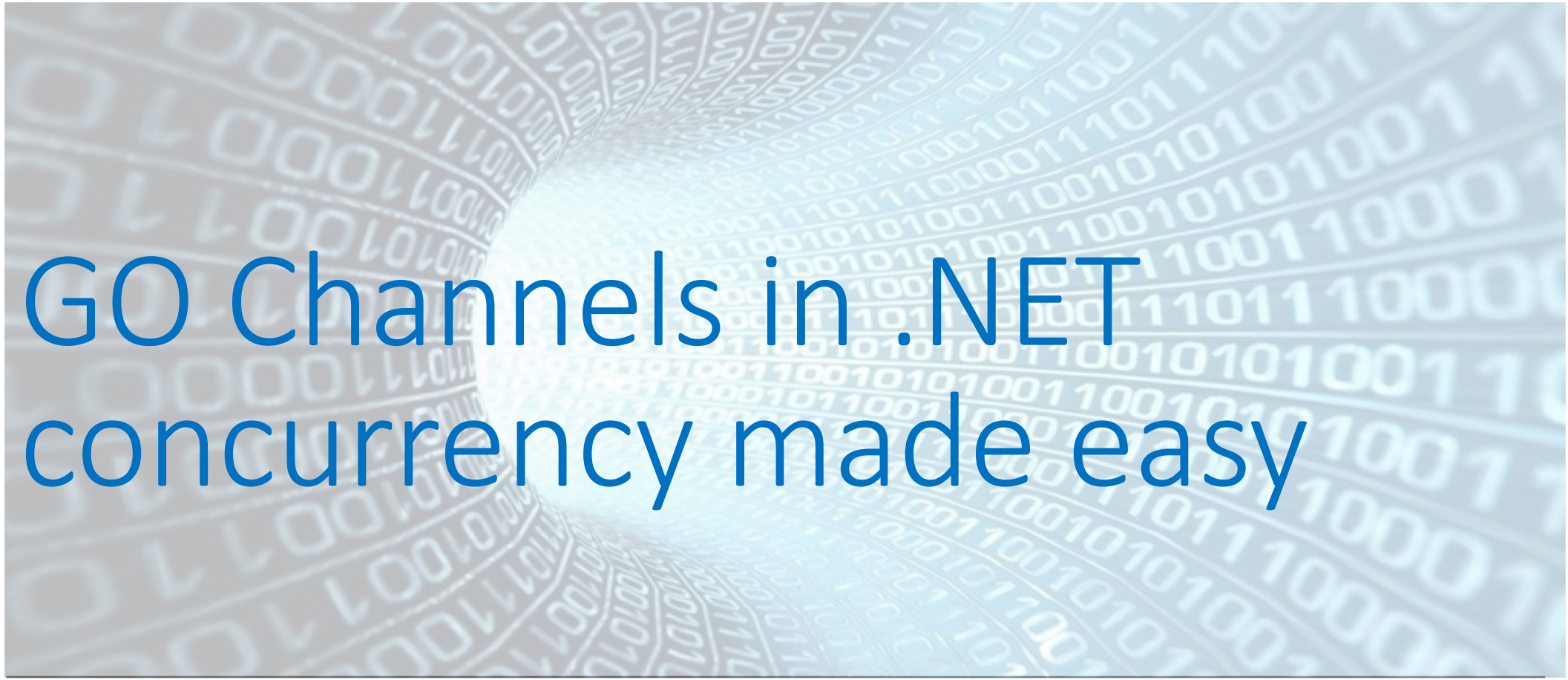

GO Channels in .NET
concurrency made easy

# CODING HORROR

programming and human factors

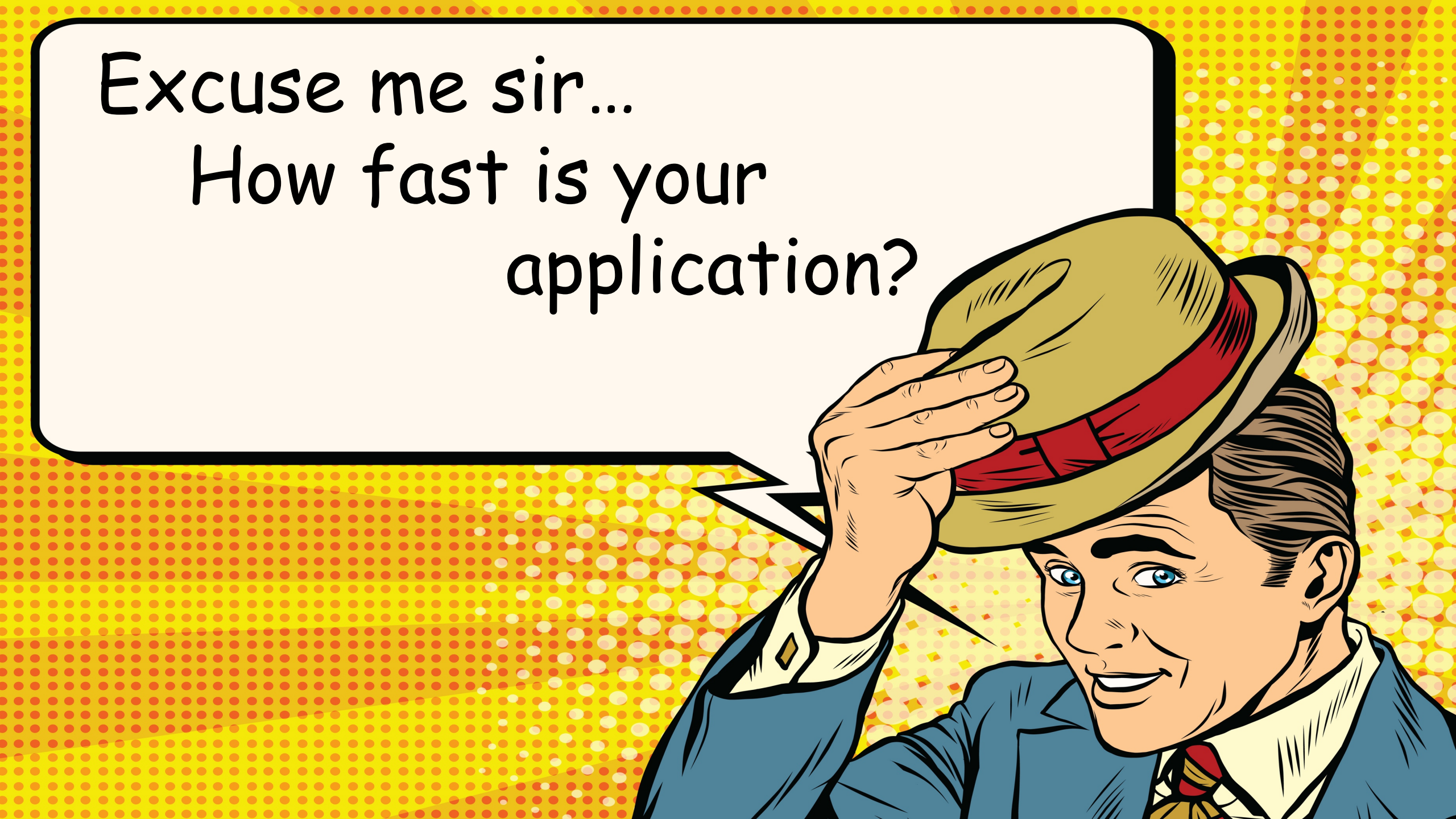
Google™ Custom Search

20 Jun 2011

# Performance is a Feature

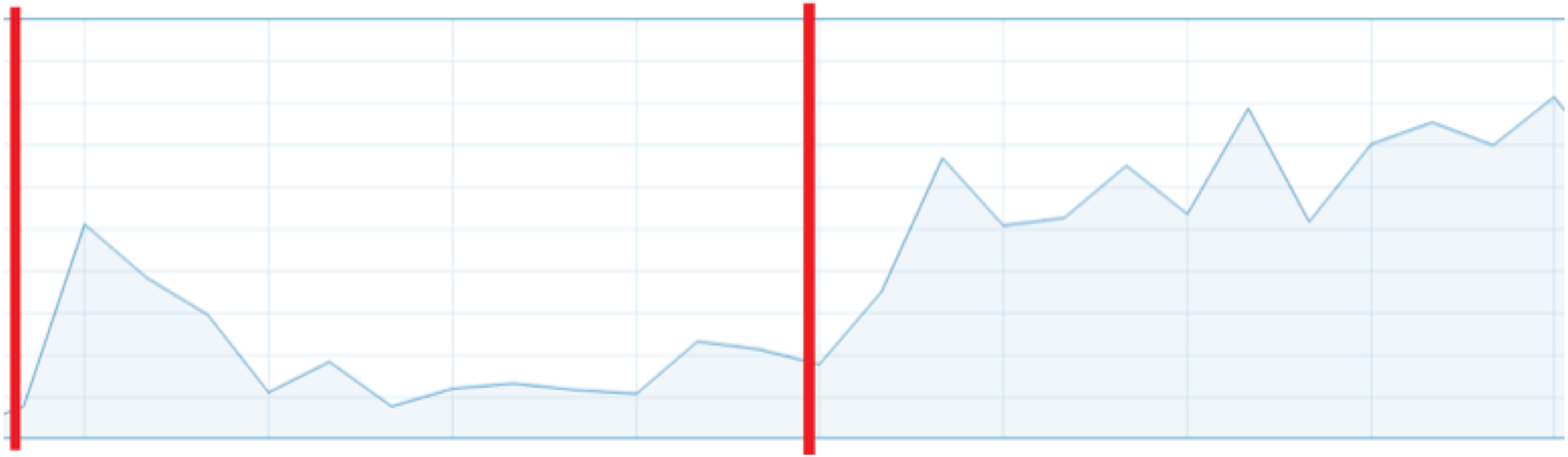
We've always put a heavy emphasis on performance at Stack Overflow and Stack Exchange. Not just because we're performance wonks (guilty!), but because we think speed is a competitive advantage. There's plenty of experimental data proving that **the slower your website loads and displays, the less people will use it.**

> [Google found that] the page with 10 results took 0.4 seconds to generate. The page with 30 results took 0.9 seconds. Half a second delay caused a 20% drop in traffic. Half a second delay killed user satisfaction.
>
> In A/B tests, [Amazon] tried delaying the page in increments of 100 milliseconds and found that even very small delays would result in substantial and costly drops in revenue.


Excuse me sir…
How fast is your application?


Fast enough…

# Benchmark (30 flaps – 1000 clients)



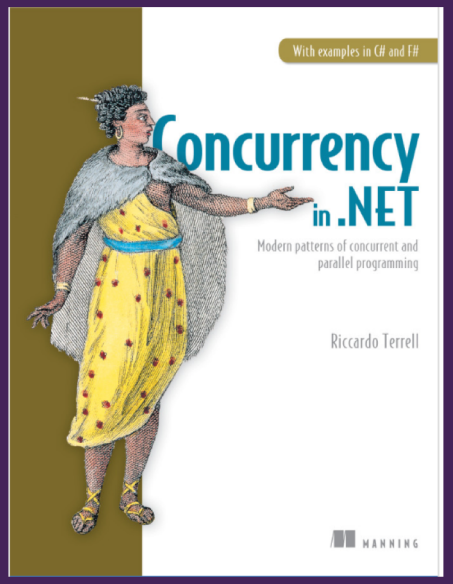
**Channels** **async/await**

# Objectives

- Understanding the CSP (Communicating Sequential Processes) model
- Advantages and benefits of channels
- Implementing a CSP using .NET (C# & F#)
- Composing Channels and patterns
- Implementing high performant pipelines without blocking any thread

# Introduction - Riccardo Terrell

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



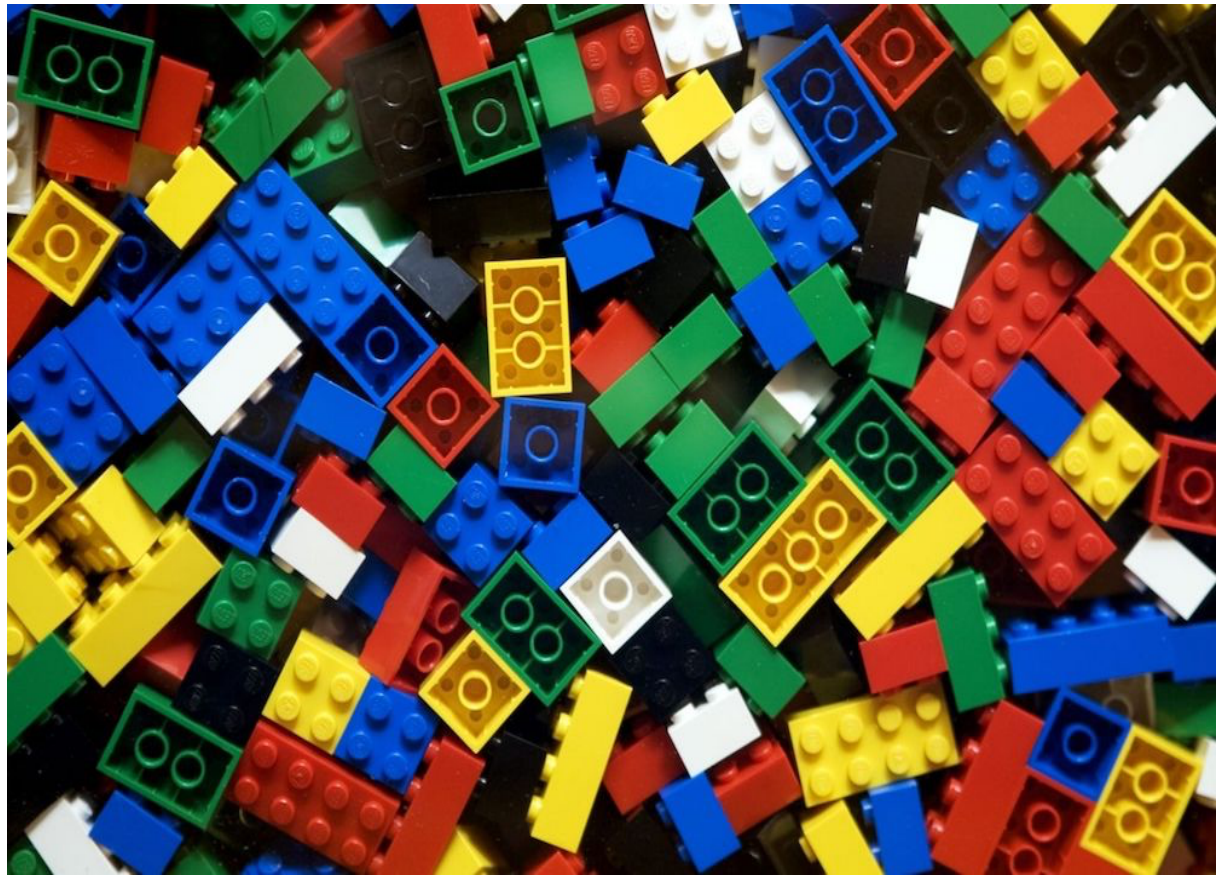


@trikace www.rickyterrell.com tericcardo@gmail.com

# Strategies to parallelize the code

The first step in designing any parallelized system is **Decomposition**.

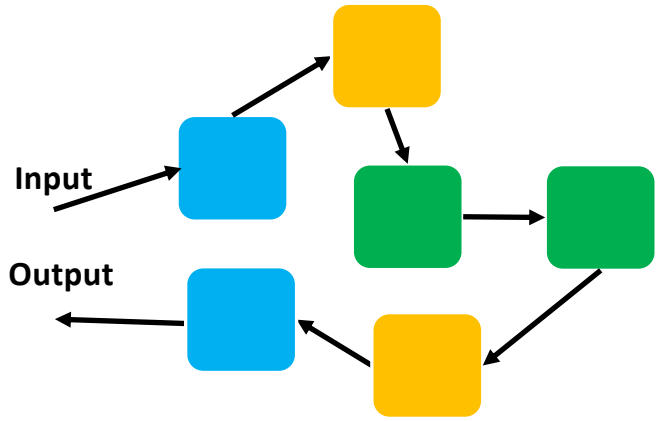
Decomposition is nothing more than taking a problem space and breaking it into discrete parts. When we want to work in parallel, we need to have at least two separate things that we are trying to run. We do this by taking our problem and decomposing it into parts.



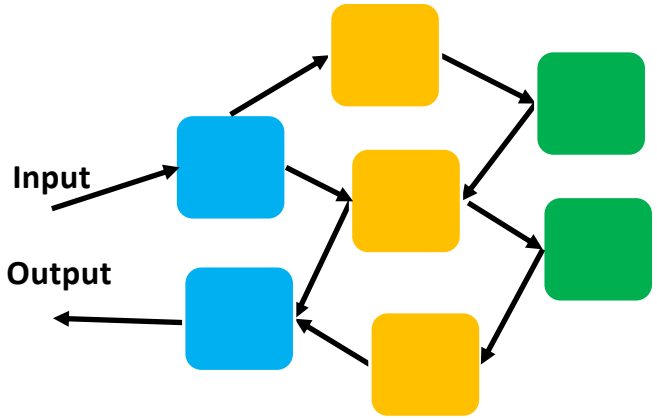
# What’s the value? … Juggling parallelism

# Different type of concurrency models lead to different challenges

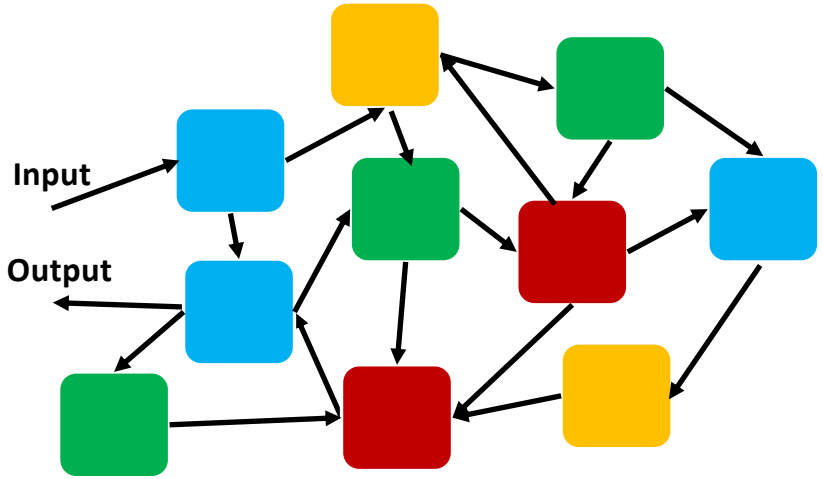
**Sequential Programming**



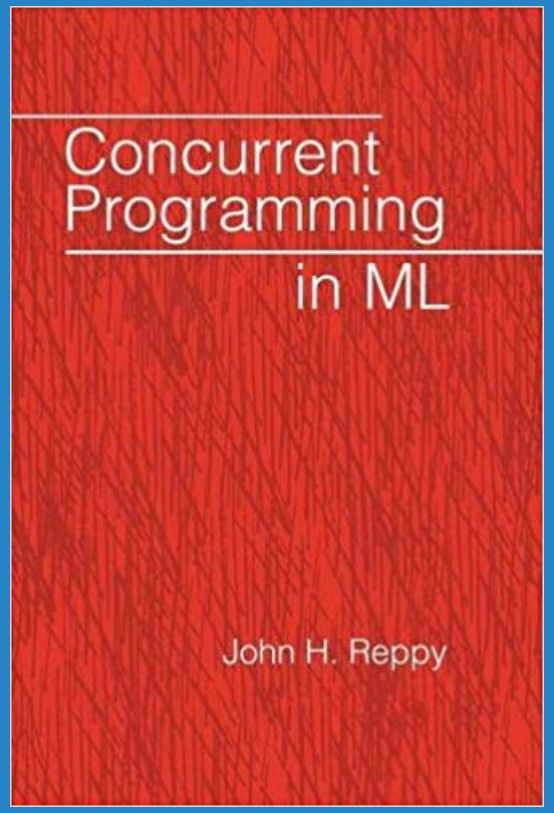

**Task-Based Programming**




**Message-Passing Programming**



# Communicating Sequential Process




Programming Techniques
S. L. Graham, R. L. Rivest Editors

## Communicating Sequential Processes

C.A.R. Hoare
The Queen's University
Belfast, Northern Ireland

**This paper suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method. When combined with a development of Dijkstra's guarded command, these concepts are surprisingly versatile. Their use is illustrated by sample solutions of a variety of familiar programming exercises.**

**Key Words and Phrases: programming, programming languages, programming primitives, program structures, parallel programming, concurrency, input, output, guarded commands, nondeterminacy, coroutines, procedures, multiple entries, multiple exits, classes, data representations, recursion, conditional critical regions, monitors, iterative arrays**
**CR Categories: 4.20, 4.22, 4.32**

grams, three basic constructs have received widespread recognition and use: A repetitive construct (e.g. the **while** loop), an alternative construct (e.g. the conditional **if..then..else**), and normal sequential program composition (often denoted by a semicolon). Less agreement has been reached about the design of other important program structures, and many suggestions have been made: Subroutines (Fortran), procedures (Algol 60 [15]), entries (PL/I), coroutines (UNIX [17]), classes (SIMULA 67 [5]), processes and monitors (Concurrent Pascal [2]), clusters (CLU [13]), forms (ALPHARD [19]), actors (Hewitt [1]).

The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself (as in the multiple function units of the CDC 6600) or by the software (as in an I/O control package, or a multiprogrammed operating system). However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.

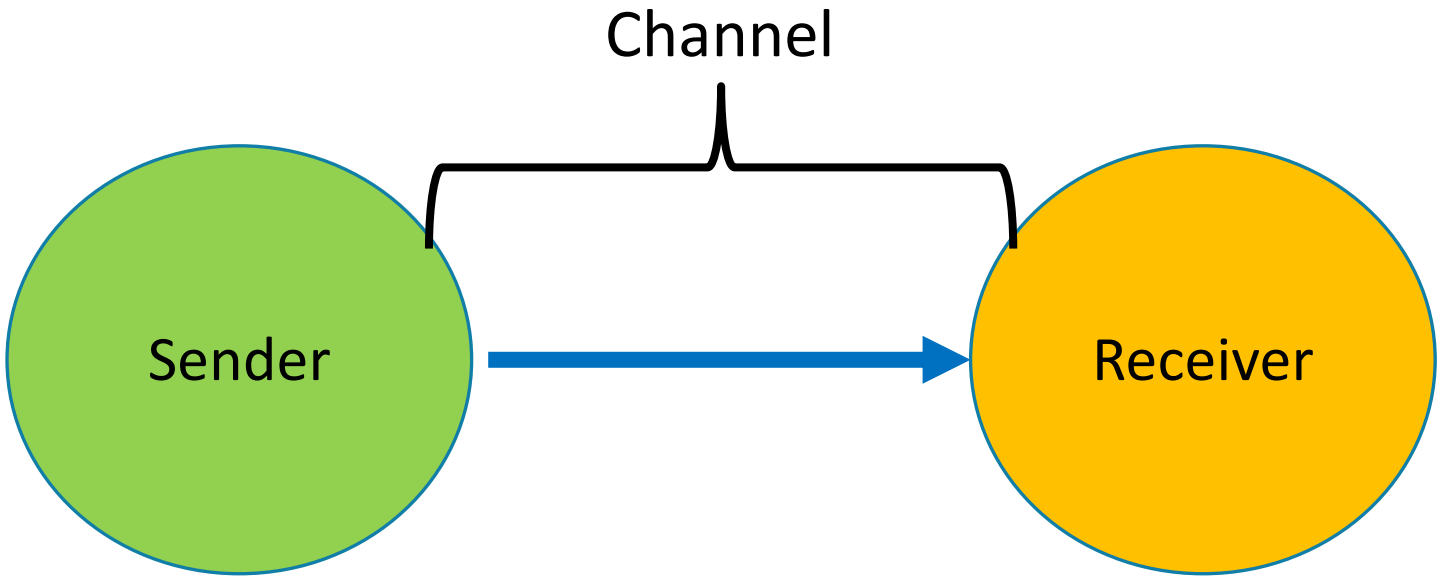
In order to use such a machine effectively on a single task, the component processors must be able to communicate and to synchronize with each other. Many methods of achieving this have been proposed. A widely adopted method of communication is by inspection and updating of a common store (as in Algol 68 [18], PL/I, and many machine codes). However, this can create severe problems in the construction of correct programs

**Channel is high performance zero-copy buffer-pool-managed asynchronous message pipes**

https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf

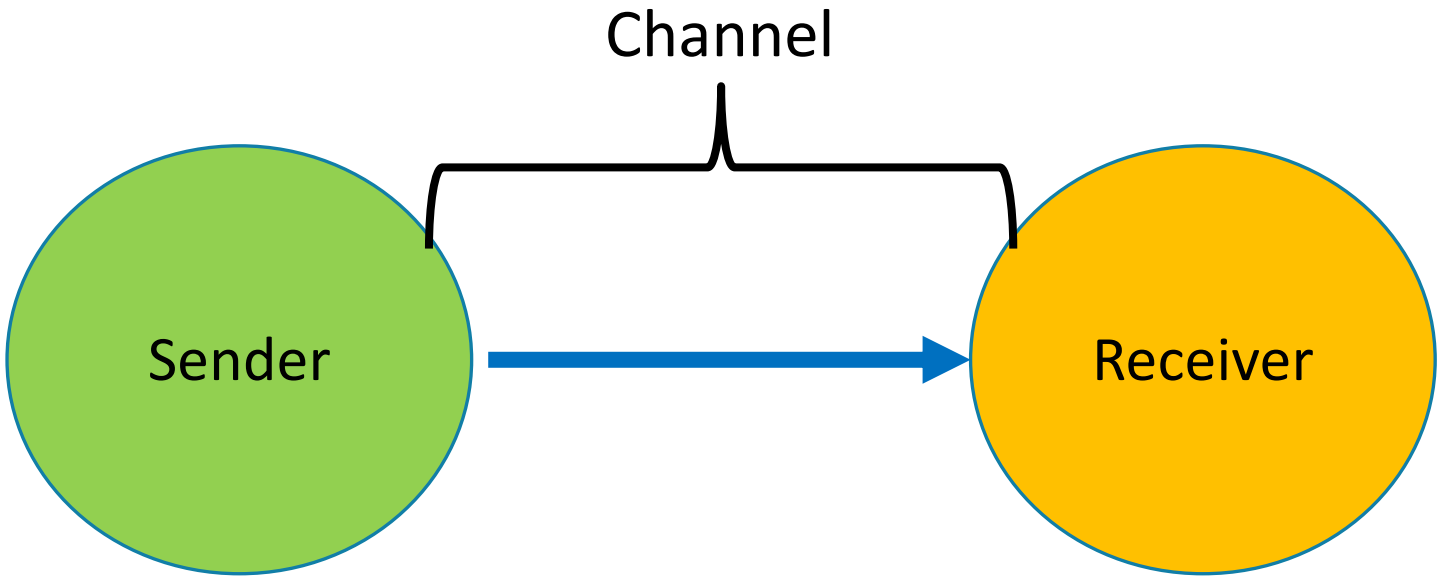
# What is a Channel?

*Do not communicate by sharing memory… share memory by communicating*



# What is a Channel?

*Do not communicate by sharing memory… share memory by communicating*

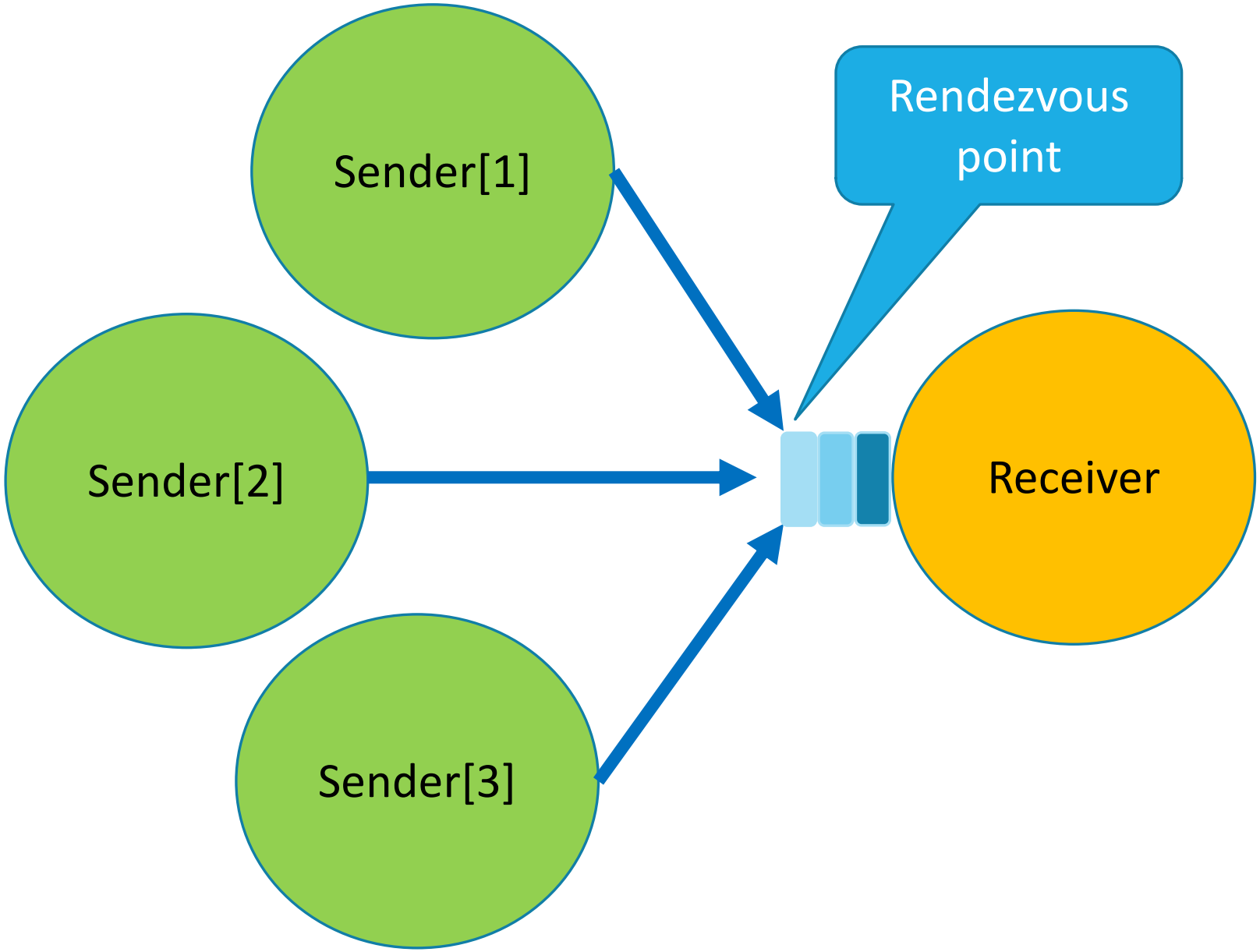



**Send** – send the value of an expression to a channel. The process calling the send operation is blocked until the message is received from the channel.

**Receive** – receive a value into local variable from a channel. The calling process is blocked waiting until a message is sent to the channel.

# What is a Channel?

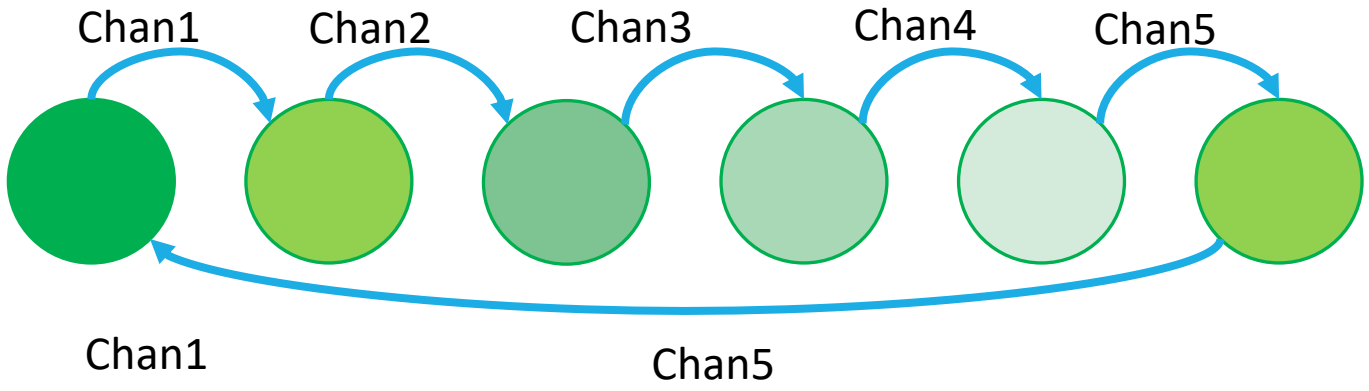
*Do not communicate by sharing memory… share memory by communicating*

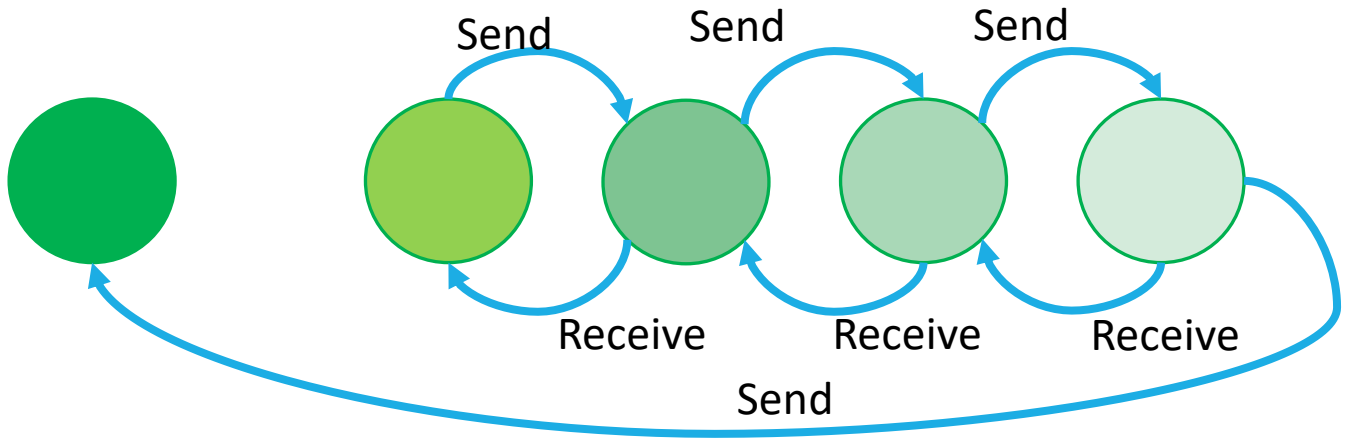


# Communicating Sequential Process

Share memory by communicating

## Share memory by communicating



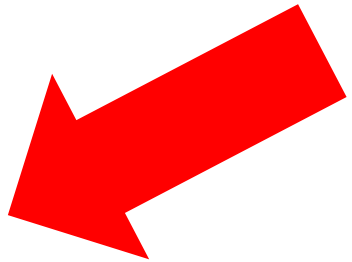



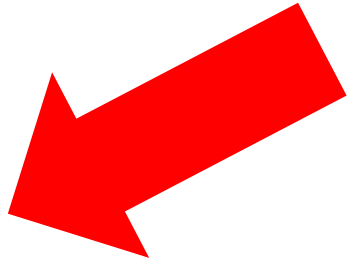
# What is a Channel?

CSP - Synchronous

Synchronous is not bad in this case

```
public interface IReceiver<T>
{
    T Receive();

    Task<T> ReceiveAsync();

    bool TryReceive(out T value);
}

public interface ISender<T>
{
  void Send(T value);

  Task SendAsync(T value);

  bool TrySend(T value);
}

public interface IChannel<T> : IReceiver<T>, ISender<T>
{    }
```





# What is a Channel?

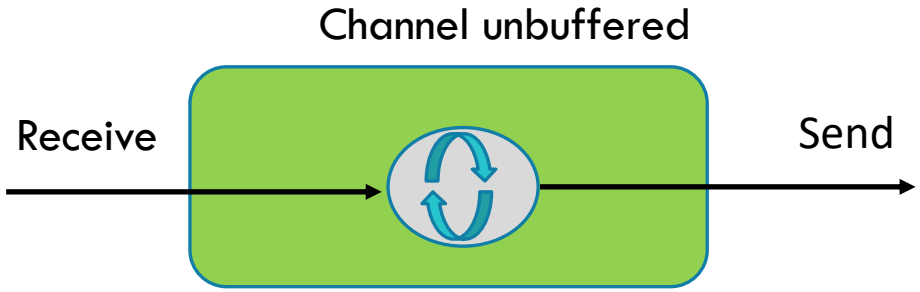
```
Task Ping(ISend<string> pings, string msg) =>   pings.SendAsync(msg);

Task async Pong(IReceive<string> pings, ISend<string> pongs)
{
   var (msg, _) = await pings.ReceiveAsync();
   pongs.Send(msg);
}

public static void ChannelDirections()
{
   var pings = Channel.Make<string>();
   var pongs = Channel.Make<string>();

   Task.Run( async () =>   {
      while (true)
          await Ping(pings, "passed message");
   });

   Task.Run( async () =>   {
      while (true)
      {
         await Pong(pings, pongs);
         Console.WriteLine(pongs.Receive().Message);
      }
   });
}
```

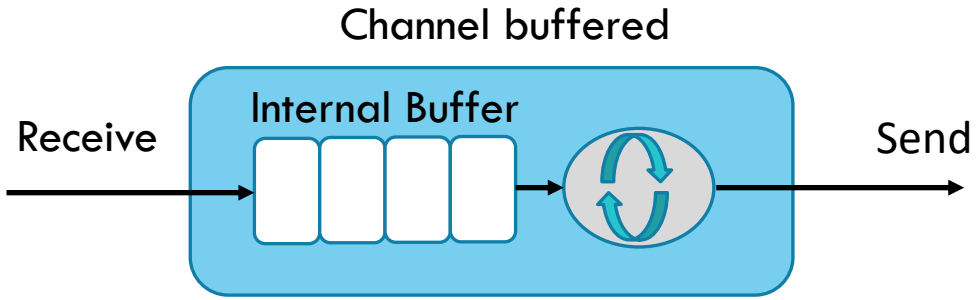
# Channel Producer Consumer

```
var chan = new ChannelAsync.Channel<int>();

for (int j = 0; j < 5; j++)
{
    Task.Run(async () =>
    {
        for (int i = 0; i < pos; i++)
        {
            await chan.SendAsync(i);
        }

        if (Interlocked.Increment(ref index) == 5)
            chan.Close();
    });
}


Task.Run(() =>
{
    var sum = 0;
    do
    {
        //var val = await chan.ReceiveAsync();
        if (chan.TryReceive(out int v))
            sum += v;
    } while (chan.IsClosed == false);

    Console.WriteLine($"Sum = {sum} (should be 2497500)");
});
```

# Channel Buffering

Buffered and Unbuffered Channels




```
public class BufferedChannel<T> : ISender<T>, IReceiver<T>, IChannel<T>
{
    readonly CircularQueue<T> _items;

    public BufferedChannel(int capacity)
        => _items = new CircularQueue<T>(capacity);
```

# Channel Buffering

Buffered and Unbuffered Channels

```
var ch = new BufferedChannel<int>(3);

var st1 = ch.SendAsync(1);
var st2 = ch.SendAsync(2);

if (st2.IsCompleted)
   Console.WriteLine("second send should wait for a receiver");

Task.Run(async () =>
{
   while (true)
   {
      int val = await ch.ReceiveAsync();
      Console.WriteLine($"received channel - value {val}");
   }
});
```

# Select

Choose among channels

```
var ch1 = new Channel<int>();
var ch2 = new Channel<int>();

var select = new Select()
    .OnReceive(ch1, val =>
    {
        Console.WriteLine($”chan1 completed {val}");
    })
    .OnReceive(ch2, val =>
    {
      Console.WriteLine($”chan2 completed {val}");
    });

var sendTask1 = ch1.SendAsync(1);
var sendTask2 = ch2.SendAsync(2);
```

# Select

Choose among channels

```
var ch1 = new Channel<int>();
var ch2 = new Channel<int>();
var ch3 = new Channel<int>();


var select = new Select();


Channel.Select()
        .Case(ch1, m => Console.WriteLine(m))
        .Case(ch2, m => Console.WriteLine(m))
        .Case(ch3, m => Console.WriteLine(m))
        .ReceiveAsync();

var sendTask1 = ch1.SendAsync(1);
var sendTask2 = ch2.SendAsync(2);
```

# Select

Choose among channels

Multiplexer

Fork-join

```
Task ComputeSelect<T, R>(IReceiver<T>[] receivers, ISender<R> sender,
                                                    Func<T, Task<R>> map)
{
   return Task.Run(async () =>    {
      while (true)   // << use cancellation token
      {

         var select = new Select();

         foreach (var chan in receivers)
         {
            select.OnReceiveAsync(chan, async val =>      {
               var result = await map(val);
               await sender.SendAsync(result);
            });
         }
         await select.ExecuteAsync();
      }
   });
}
```

# Merge

```
public static IReceive<T> Merge<T>(params IReceive<T>[] sources)
{
    var output = Channel<T>();
    Task.Run(() =>     {
        var loop = true;
        while (loop) // << use cancellation token
        {
            var select = Select();
            foreach (var chan in sources)
            {
                select.Case(chan, m => output.Send(m));
            }

            var err = select.Receive();
            loop = err == null;
        }

        output.Close();
    });
    return output;
}
```

# Channel-Multicast

Message-Broker

```
public class MulticastChannel<T> : ISender<T>
{
    public void Subscribe(ISender<T> channel)
                        => _subscriptions.Add(channel);

    public void Unsubscribe(ISender<T> channel)
                       =>  _subscriptions.Remove(channel);


 public void Send(T value)
{
        if (_closed.IsCancellationRequested)
            throw new OperationCanceledException();

         foreach (var s in _subscriptions)
         {
            s.TrySend(value);  // don't block if a receiver is not ready
         }
    }
 }
```

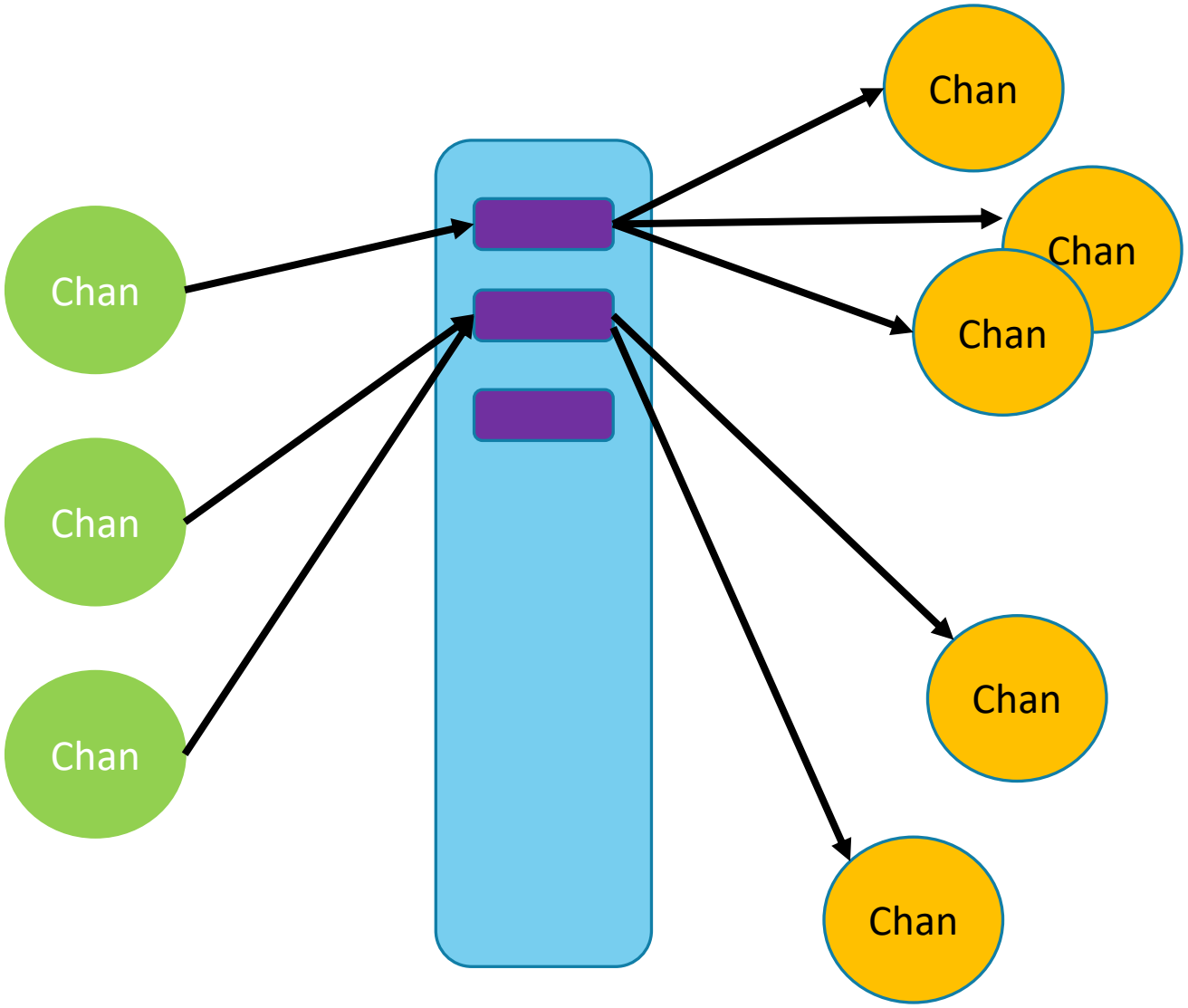
# Channel Publisher Subscriber

```
public sealed class PubSub<T> : IDisposable {

    readonly List<IChannel<T>> subscriberChannels = new List<IChannel<T>>();
    readonly IChannel<T> publisherChannel;

    public ISend<T> Channel => publisherChannel;

    public PubSub(int buffer) {
        publisherChannel = buffer == 0
            ? Channel.Make<T>()
            :  Channel.Make<T>(buffer);

        Task.Run(() =>      {
              while (loop) {
               var (msg, err) = publisherChannel.Receive();
                foreach (var chan in subscriberChannels) {
                      chan.Send(msg);
                }
              }
              foreach (var chan in subscriberChannels)
                  chan.Close();
        });

    public Task Publish(T message) => publisherChannel.Send(message);

    public IReceive<T> Subscribe() {
          var channel = Channel.Make<T>();
          subscriberChannels.Add(channel);
          return channel;
    }
```

# Channel-Multicast

Message-Broker

Decoupling concurrent system



# Merge & Range

```
static IReceive<int> Generate(int n)
{
    var numbers = Channel.Make<int>(n);
    for (var i = 0; i < n; i++)
        numbers.Send(i);
     return numbers;
}

static void Resend(IReceive<int> numbers, ISend<string> results, string name)
{
    foreach (var n in numbers.Range())
    {
        Task.Delay(5).Wait();
        results.Send($"{name} : {n}");
    }
    Console.WriteLine($" [+] {name} done");
}

public static void MergeTest()
{
    var numbers = Generate(50);
    var r1 = Channel.Make<string>();
    var r2 = Channel.Make<string>();
    var r3 = Channel.Make<string>();

    Task.Run(() => Resend(numbers, r1, "first"));
    Task.Run(() => Resend(numbers, r2, "second"));
    Task.Run(() => Resend(numbers, r3, "third"));

   foreach (var r in Channel.Merge(r1, r2, r3).Range())
        Console.WriteLine(r);
}
```

# Merge & Range

```
public static void MergeTest()
{
    var numbers = Generate(50);
    var r1 = Channel.Make<string>();
    var r2 = Channel.Make<string>();
    var r3 = Channel.Make<string>();

    Task.Run(() => Resend(numbers, r1, "first"));
    Task.Run(() => Resend(numbers, r2, "second"));
    Task.Run(() => Resend(numbers, r3, "third"));

   foreach (var r in Channel.Merge(r1, r2, r3).Range()
                        .Where(msg => msg.IndexOf("first") > 0)
                        .Select (msg => msg.ToUpper()))

        Console.WriteLine(r);
}
```

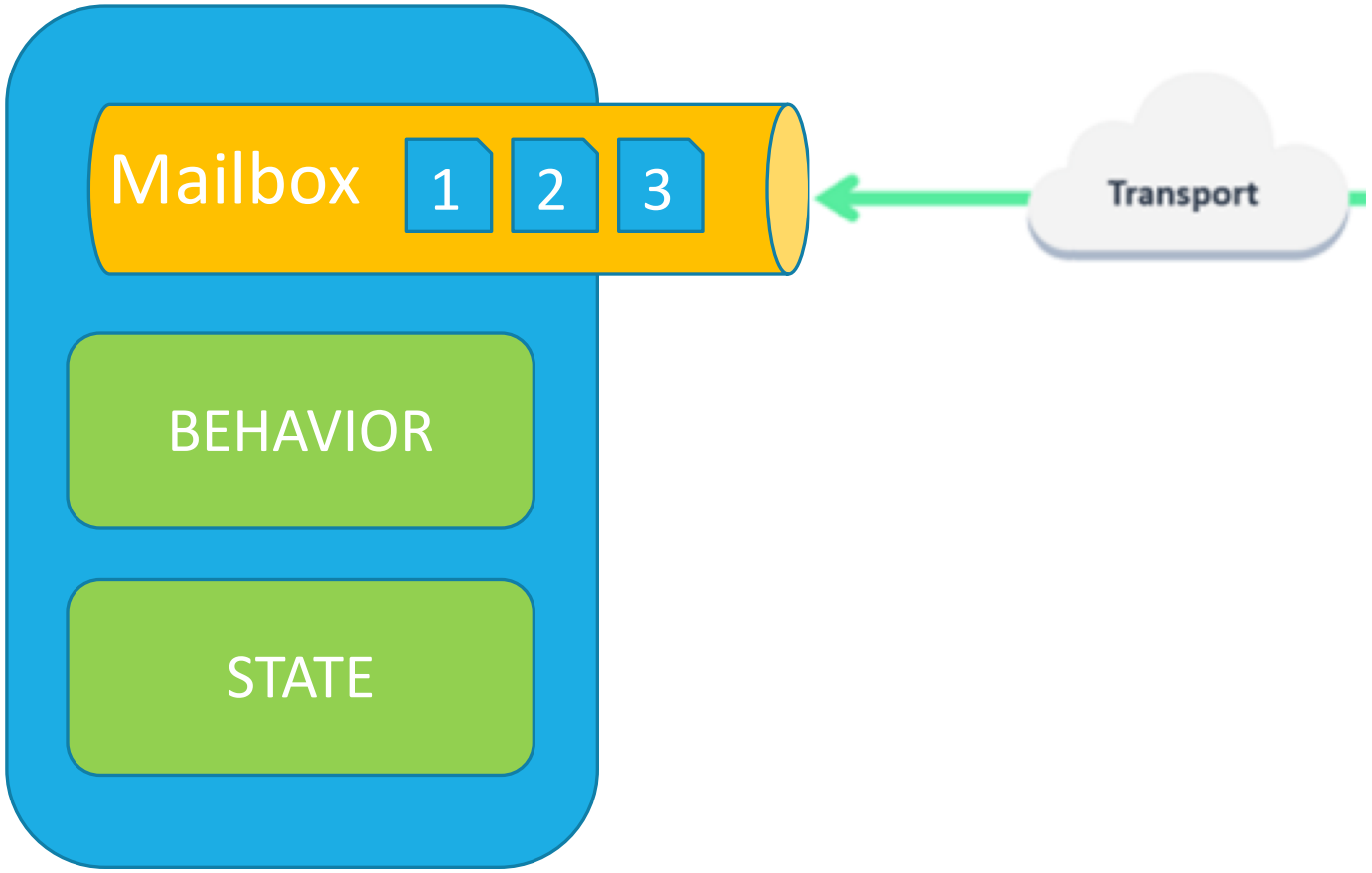
# Merge & Range

```
public static void MergeTest()
{
    var numbers = Generate(50);
    var r1 = Channel.Make<string>();
    var r2 = Channel.Make<string>();
    var r3 = Channel.Make<string>();

    Task.Run(() => Resend(numbers, r1, "first"));
    Task.Run(() => Resend(numbers, r2, "second"));
    Task.Run(() => Resend(numbers, r3, "third"));

    Channel.Merge(r1, r2, r3).Range()
      .ToObservable()
      .Where(msg => msg.IndexOf("first") > 0)
      .Select(msg => msg.ToUpper())
      .Subscribe(r => {
       Console.WriteLine(r);
    });

}
```

# CSP vs Message Passing

## Channel  vs Agent

# 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

# Channel vs Agent

## Share Nothing, Async Message Passing

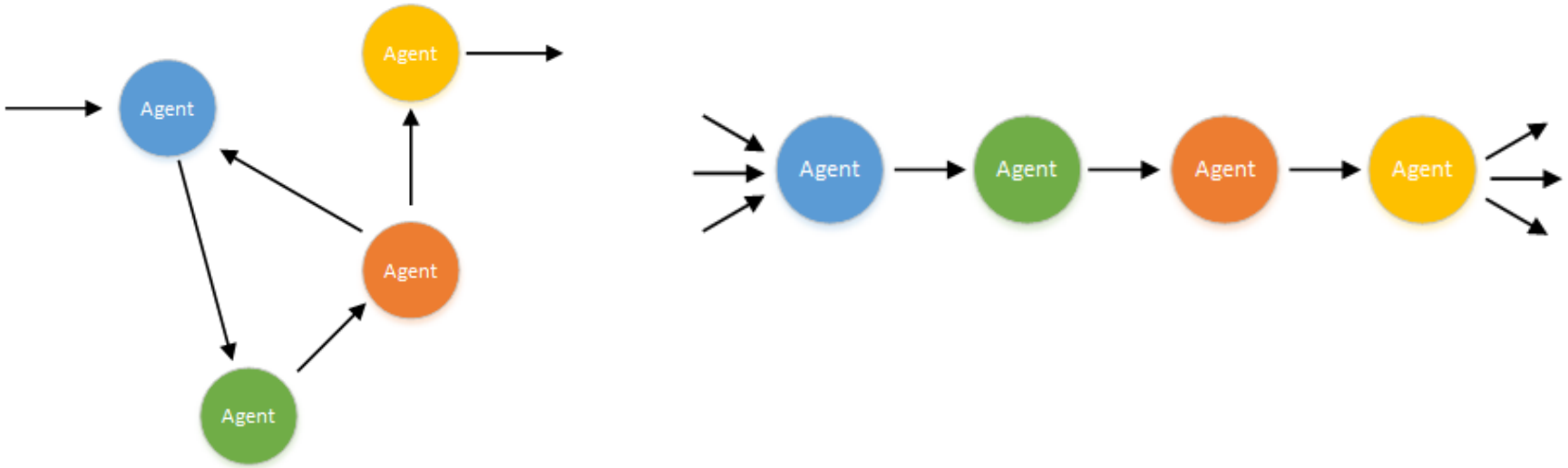
- Encourages shared-nothing abstractions
  - mutable state - private
  - shared state - immutable
- Asynchronous message passing
- Pattern matching
- Unbounded
- Backpressure

# F# Channel using Mailbox

```fsharp
type internal ChannelMsg<'a> =
    | Recv of ('a -> unit) * AsyncReplyChannel<unit>
    | Send of 'a * (unit -> unit) * AsyncReplyChannel<unit>

type [<Sealed>] ChannelAgent<'a>() =
    let agent = MailboxProcessor<ChannelMsg<'a>>.Start(fun inbox ->
        let readers = Queue<'a -> unit>()
        let writers = Queue<'a * (unit -> unit)>()

        let rec loop() = async {
            let! msg = inbox.Receive()
            match msg with
            | Recv(ok , reply) ->
                if writers.Count = 0 then
                    readers.Enqueue ok
                    reply.Reply( () )
                else
                    let (value, cont) = writers.Dequeue()
                    TaskPool.Spawn  cont
                    reply.Reply( (ok value) )
                return! loop()
            | Send(x, ok, reply) ->
                if readers.Count = 0 then
                    writers.Enqueue(x, ok)
                    reply.Reply( () )
                else
                    let cont = readers.Dequeue()
                    TaskPool.Spawn ok
                    reply.Reply( (cont x) )
                return! loop() }
        loop())

    member this.Receive() =
        Async.FromContinuations(fun (ok, _,_) ->
            agent.PostAndAsyncReply(fun ch -> Recv(ok, ch)))

    member this.Send (value:'a) =
        Async.FromContinuations(fun (ok, _,_) ->
            agent.PostAndAsyncReply(fun ch -> Send(value, ok, ch)))
```

# Channel Composition and patterns

# Channel Pipeline




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

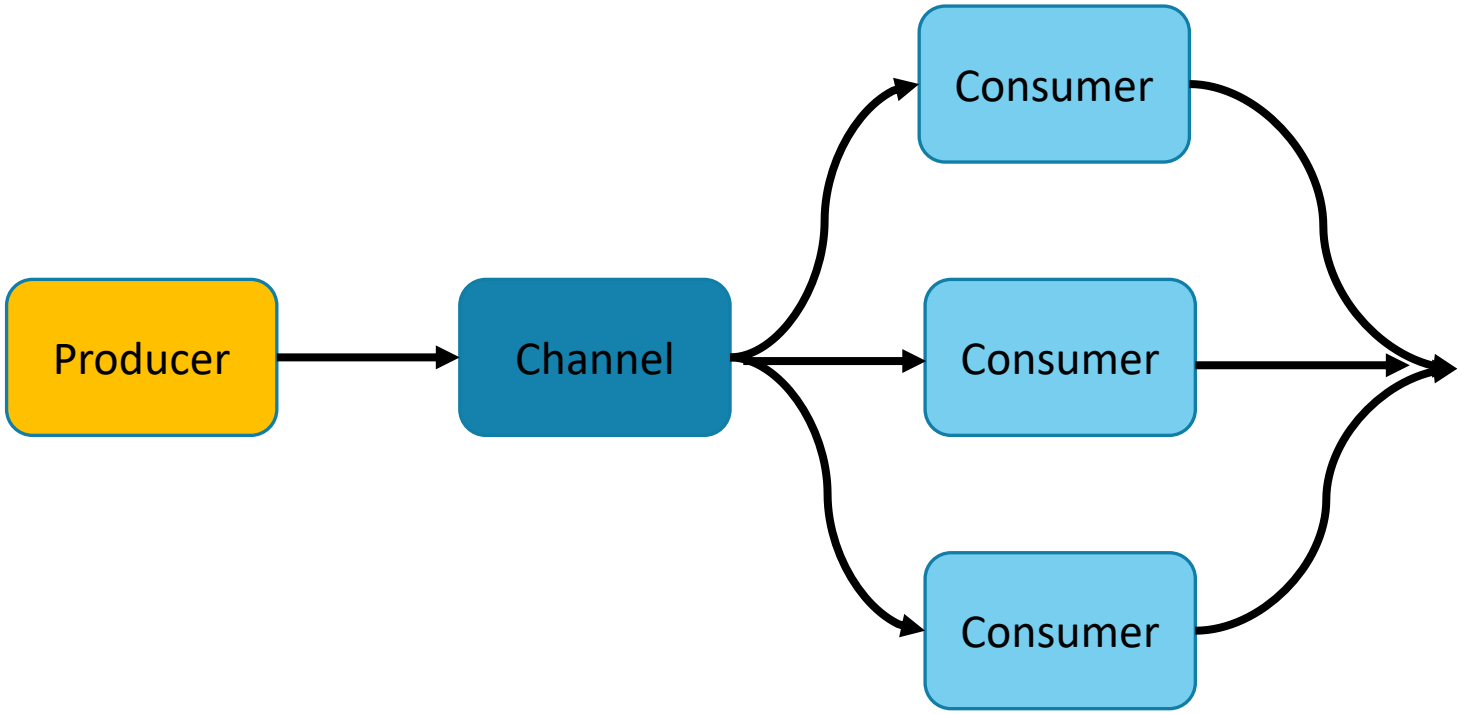
# Channel Pipeline

```
let pipelineChan (operation:'a -> Async<'b>) (chan: Channel<'a>) =
    let pipeChan = new Channel<'b>()
    let rec compute () = async {
        let! item = chan.Receive()
        let! result = operation item
        do! pipeChan.Send result
        return! compute ()    }
    compute () |> Async.Start
    pipeChan

let chan = new Channel<string>()
let loadImageChan = pipelineChan ImageHelpers.loadImage chan
let apply3DEffectChan = pipelineChan ImageHelpers.apply3D loadImageChan
let saveImageChan = pipelineChan ImageHelpers.saveImage apply3DEffectChan
```

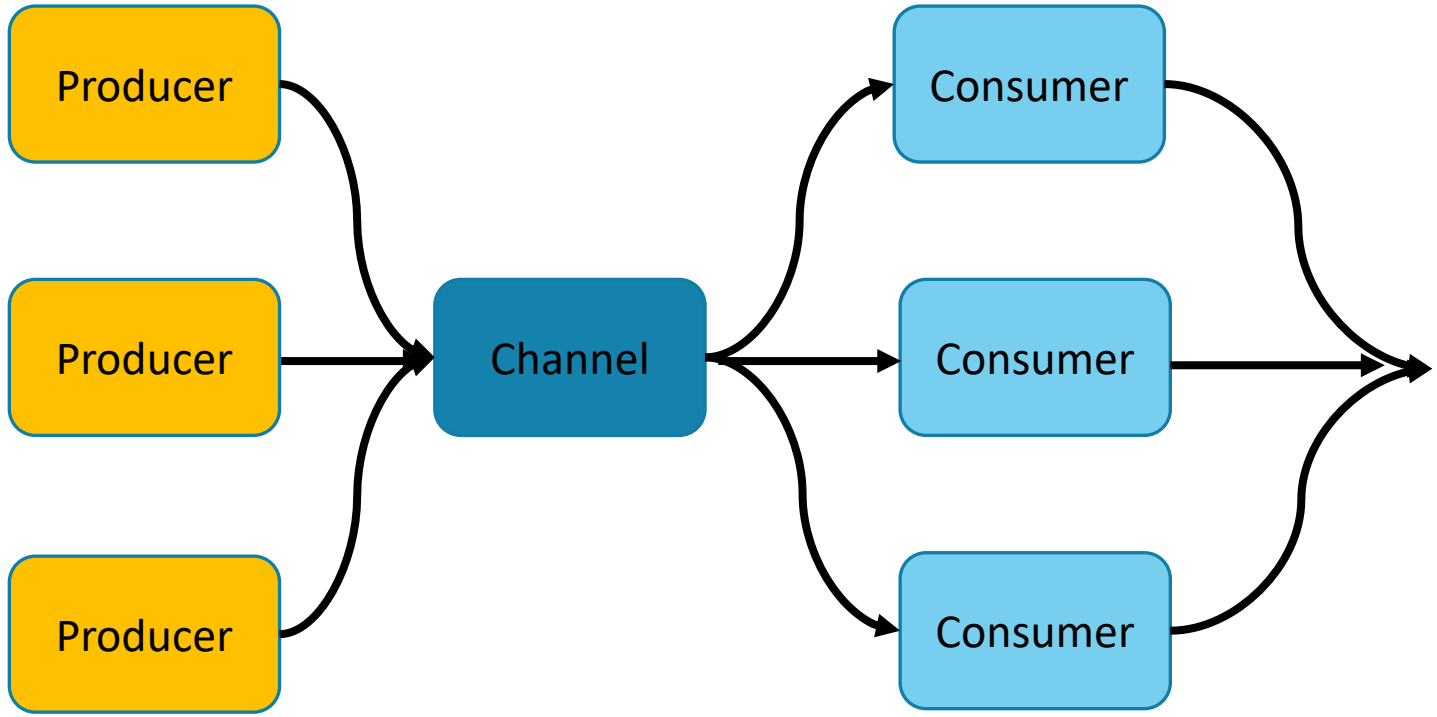
# Channel Pipeline

```
let pipelineChan (operation:'a -> Async<'b>) (chan: Channel<'a>) =
    let pipeChan = new Channel<'b>()
    let rec compute () = async {
        let! item = chan.Receive()
        let! result = operation item
        do! pipeChan.Send result
        return! compute ()    }
    compute () |> Async.Start
    pipeChan

let loadImageChan : (Channel<string> -> Channel<ImageInfo>) = pipelineChan ImageHelpers.loadImage
let apply3DEffectChan =  pipelineChan ImageHelpers.apply3D
let saveImageChan = pipelineChan ImageHelpers.saveImage

let composition (chan :Channel<string>) : Channel<ImageInfo> =
    chan |> loadImageChan |> apply3DEffectChan |> saveImageChan
```

# 1 producer
# N consumers

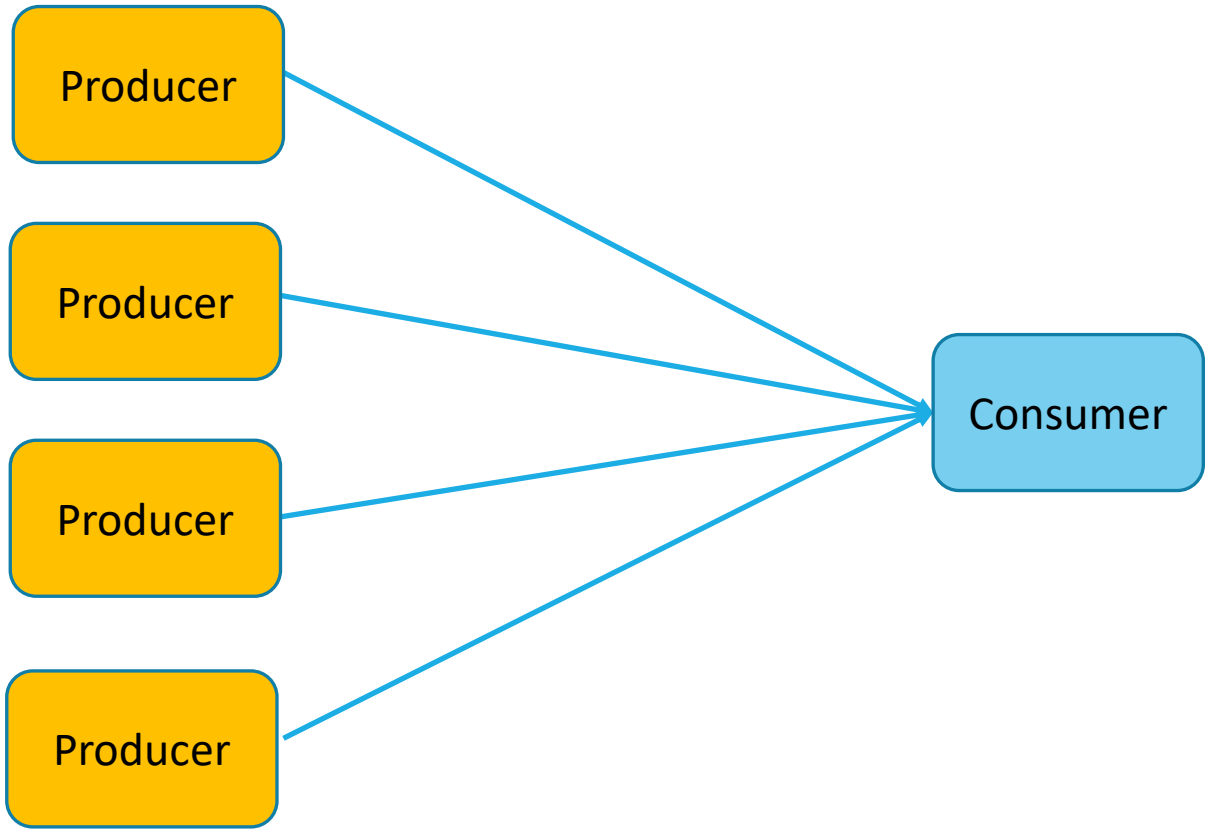


# N producer
# N consumers



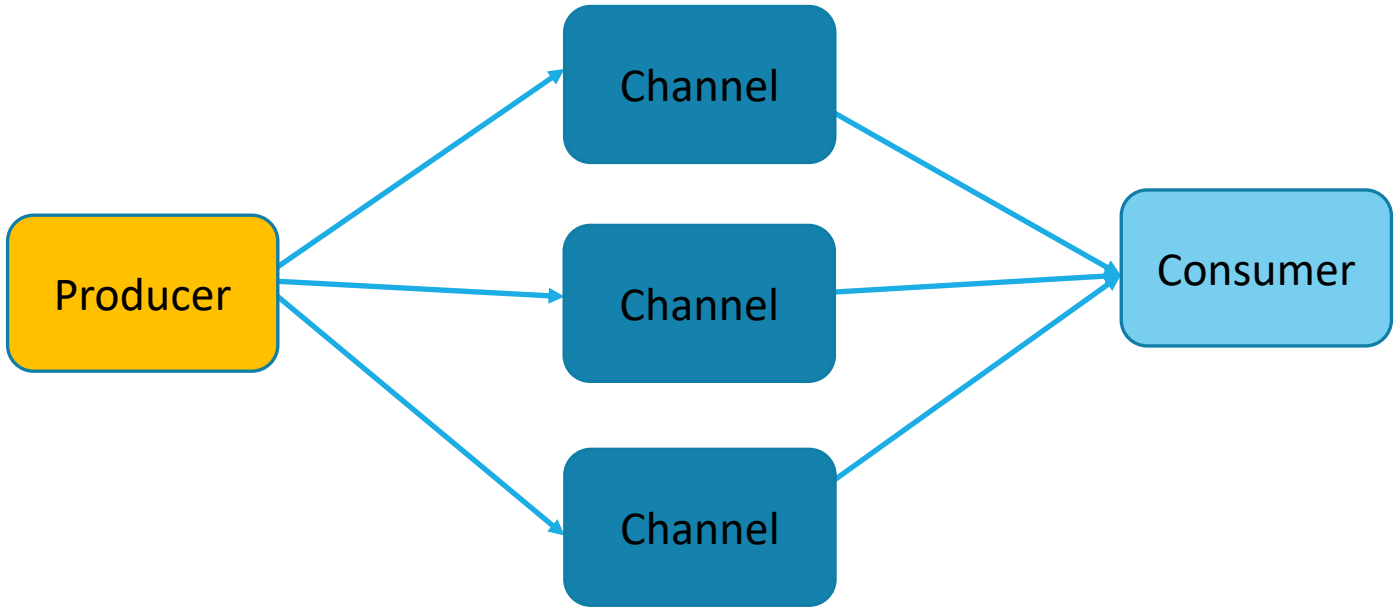
# Channel Patterns

Singleton

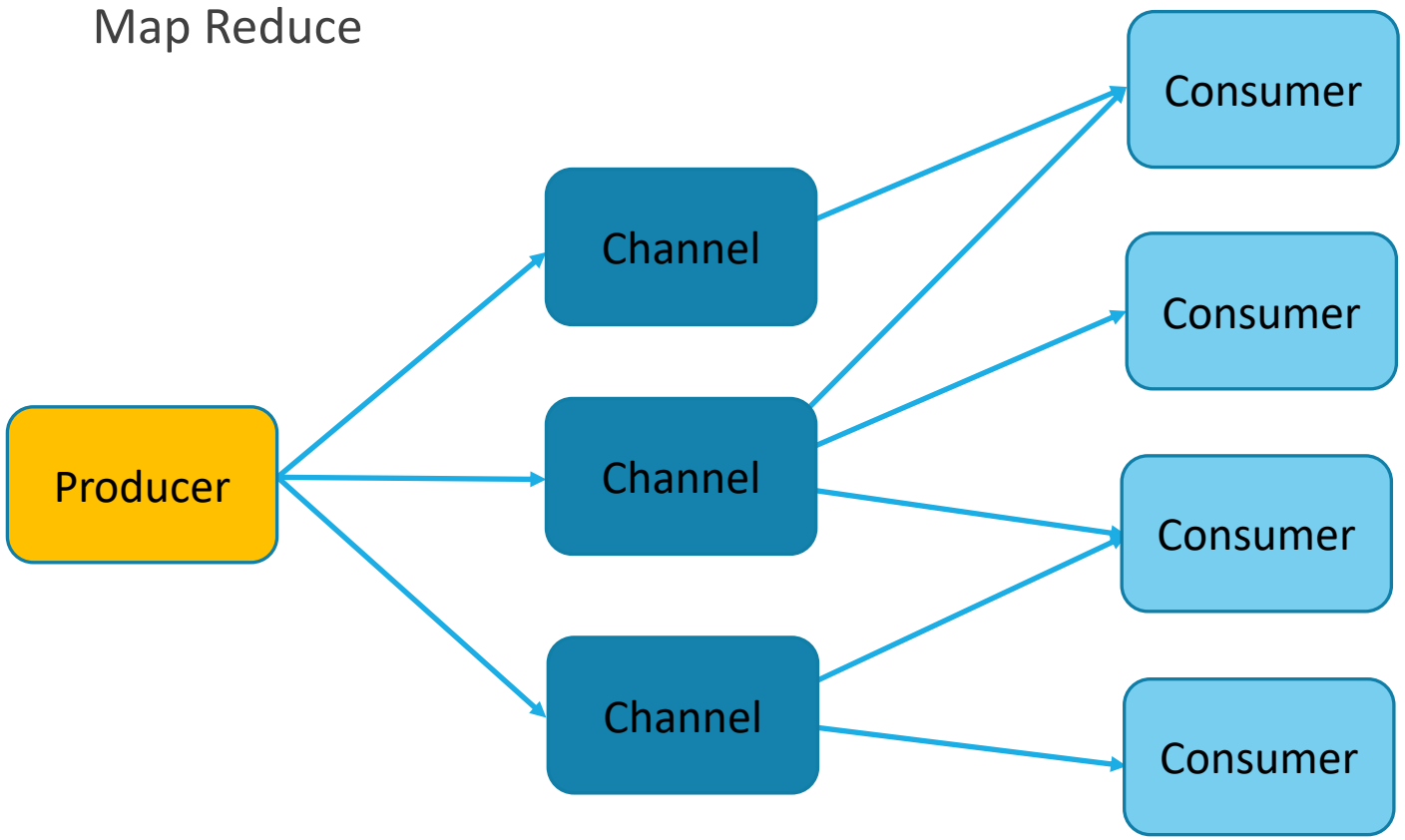


# Channel Patterns

Fork Join

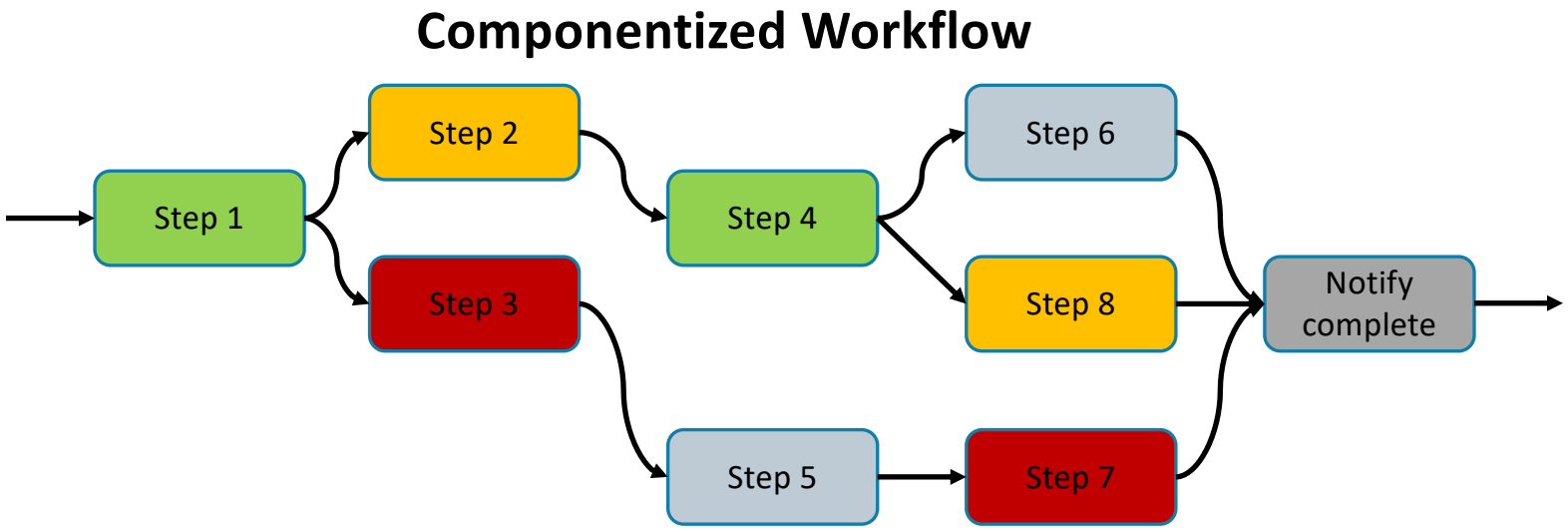


# Channel Patterns



# TPL DataFlow blocks as Channels design to compose

for High throughput, low-latency scenarios



# TPL DataFlow a stateful 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);
}
```

# Channel
# TPL Dataflow

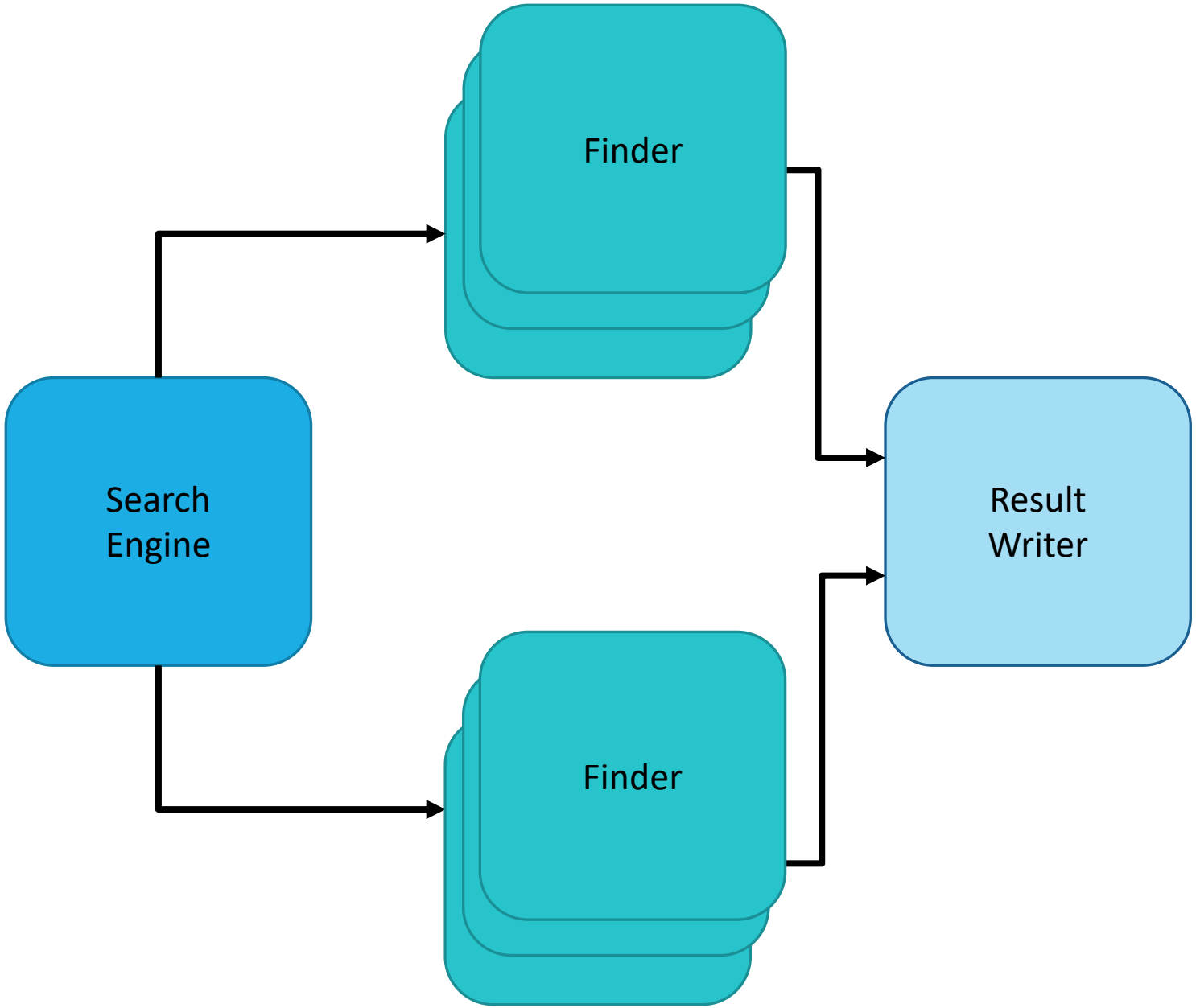
```
public class ChannelAgent<T>
{
    public ChannelAgent()
    {
        agent = new StatefulDataflowAgent<(Queue<T>, Queue<Recv<T>>), IChannelMsg<T>>(
                (new Queue<T>(), new Queue<Recv<T>>()),
                (state, message) => {    // … rest of the code here

    private StatefulDataflowAgent<(Queue<T>, Queue<Recv<T>>), IChannelMsg<T>> agent;

    private Task Recv(Action<T> handler)
        => agent.Send(new Recv<T>(handler));

    public Task Send(T value)
        => agent.Send(new Send<T>(value));

    public Task Subscribe(Action<T> handler)
        => Task.Run(async () =>  {
            while (true)
            {
                var tcs = new TaskCompletionSource<bool>();
                await Recv(x =>
                {
                    handler(x);
                    tcs.SetResult(true);
                });
                await tcs.Task;
            }
        });
}
```

# Channel TPL Dataflow Example

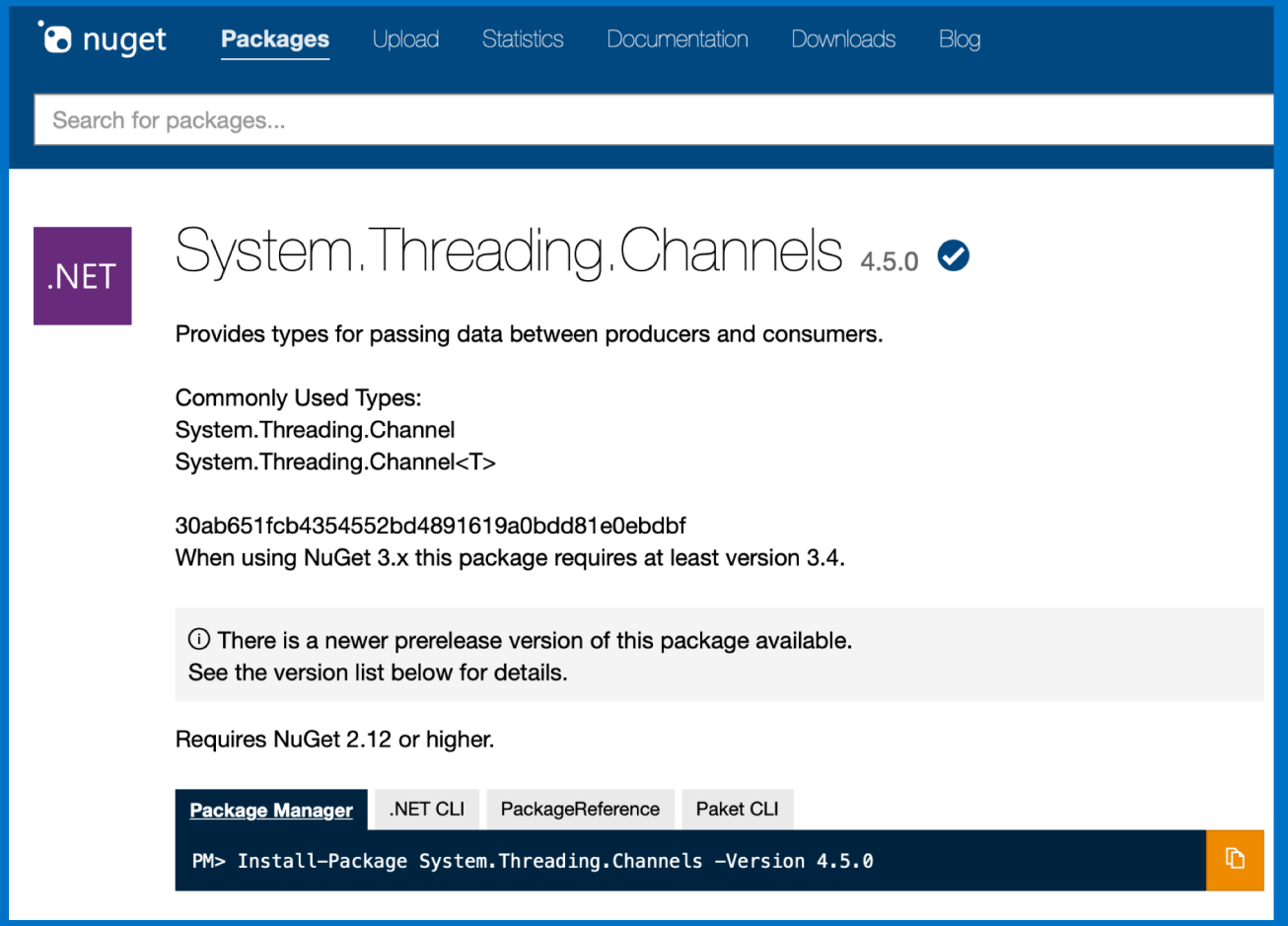
```
var chanLoadImage = new ChannelAgent<string>();
var chanApply3DEffect = new ChannelAgent<MyImageInfo>();
var chanSaveImage = new ChannelAgent<MyImageInfo>();

chanLoadImage.Subscribe(image =>
{
   var imageInfo = new MyImageInfo   {
      Path = Environment.GetFolderPath(Environment.SpecialFolder.MyPictures),
      Name = Path.GetFileName(image),
      Image = Image.Load(image)
   };
   chanApply3DEffect.Send(imageInfo);
});

chanApply3DEffect.Subscribe(imageInfo => {
   imageInfo.Image = Helpers.ImageHandler.ConvertTo3D(imageInfo.Image);
   chanSaveImage.Send(imageInfo);
});

chanSaveImage.Subscribe(imageInfo => {
   Console.WriteLine($"Saving image {imageInfo.Name}");
   var destination = Path.Combine(imageInfo.Path, imageInfo.Name);
   imageInfo.Image.Save(destination);
});

var images = Directory.GetFiles(@"../Data/Images");
foreach (var image in images)
   chanLoadImage.Send(image);
```


f10
delete
DEMO

# Parallel Text finder using Channels



# Streaming with Threading.Channels



# Channels in .NET

**System.Threading.Channels**

```
public abstract class Channel<T> : Channel<T, T> { }

public abstract class Channel<TWrite, TRead>
{
    public abstract ReadableChannel<TRead> In { get; }
    public abstract WritableChannel<TWrite> Out { get; }
    ...
}

public abstract class ReadableChannel<T>
{
    public abstract bool TryRead(out T item);
    public abstract Task<bool> WaitToReadAsync(CancellationToken token);
    public abstract Task Completion { get; }
    public virtual ValueTask<T> ReadAsync(CancellationToken token);
    public virtual ValueAwaiter<T> GetAwaiter();
}

public abstract class WritableChannel<in T>
{
    public abstract bool TryWrite(T item);
    public abstract Task<bool> WaitToWriteAsync(CancellationToken token);
    public abstract bool TryComplete(Exception error = null);
    public virtual Task WriteAsync(T item, CancellationToken token);
    public virtual ValueAwaiter<bool> GetAwaiter();
}
```

# Channels in .NET

**System.Threading.Channels**

```
public static class Channel
{
    public static Channel<T> CreateUnbounded<T>();
    public static Channel<T> CreateBounded<T>(int bufferedCapacity,
                                              BoundedChannelFullMode mode);
}

Channel.CreateBounded<T>(new BoundedChannelOptions(capacity)
{
    FullMode = BoundedChannelFullMode.Wait

// Wait - Wait for space to be available in order to complete the operation.
// DropNewest - Remove and ignore the newest item in the channel in order to make room
// DropOldest - Remove and ignore the oldest item in the channel in order to make room
// DropWrite - Drop the item being written.
});
```

```
Channel<T> channel = Channel.CreateUnbounded<T>();

ChannelReader<T> reader = channel.Reader;
ChannelWriter<T> writer = channel.Writer;
```

# .NET Channels

System.Threading.Channels

```
private ChannelWriter<string> _writer;

public ChannelsQueueMultiThreads(int threads)
{
    var channel = Channel.CreateUnbounded<string>();
    var reader = channel.Reader;
    _writer = channel.Writer;
    for (int i = 0; i < threads; i++)
    {
        var threadId = i;
        Task.Factory.StartNew(async () =>
        {
            // Wait while channel is not empty and still not completed
            while (await reader.WaitToReadAsync())
            {
                var job = await reader.ReadAsync();
                Console.WriteLine(job);
            }
        }, TaskCreationOptions.LongRunning);
    }
}

public void Enqueue(string job)
{
    _writer.WriteAsync(job).GetAwaiter().GetResult();
}
```

# .NET Channels

System.Threading.Channels

```
private ChannelWriter<IJob> _writer;
private Dictionary<Type, Action<IJob>> _handlers = new Dictionary<Type, Action<IJob>>();

public ChannelQueuePubSub()
{
   var channel = Channel.CreateUnbounded<IJob>();
   var reader = channel.Reader;
   _writer = channel.Writer;

   Task.Factory.StartNew(async () =>     {
      while (await reader.WaitToReadAsync())
      {
         var job = await reader.ReadAsync();
         bool handlerExist = _handlers.TryGetValue(job.GetType(), out Action<IJob> value);
         if (handlerExist)
          value.Invoke(job);
      }
   }, TaskCreationOptions.LongRunning);
}

public void RegisterHandler<T>(Action<T> handlerAction) where T : IJob
{
   Action<IJob> action = job => handlerAction((T) job);
   _handlers.Add(typeof(T), action);
}

public async Task Enqueue(IJob job) => await _writer.WriteAsync(job);
```

# SignalR SSE with WebSockets

Streaming with WebSockets and Channels

# Summary

- The CSP (Communicating Sequential Processes) model allows to share information among threads safely
- The CSP (Communicating Sequential Processes) model allows to build decoupled and high performant concurrent applications
- Channels ease the synchronization among processes
- We have saw how to implement a CSP using .NET (C# & F#)
- Channels can be composed easily
- .NET System.Threading.Channel can dispatch well over 20 million messages in under a second and does not block any thread while doing so.

# contacts

| | |
|---|---|
| *Source* | https://github.com/rikace/IOChannels |
| *Twitter* | *@trikace* |
| *Blog* | *www.rickyterrell.com* |
| *Email* | *tericcardo@gmail.com* |
| *Github* | *www.github.com/rikace/* |