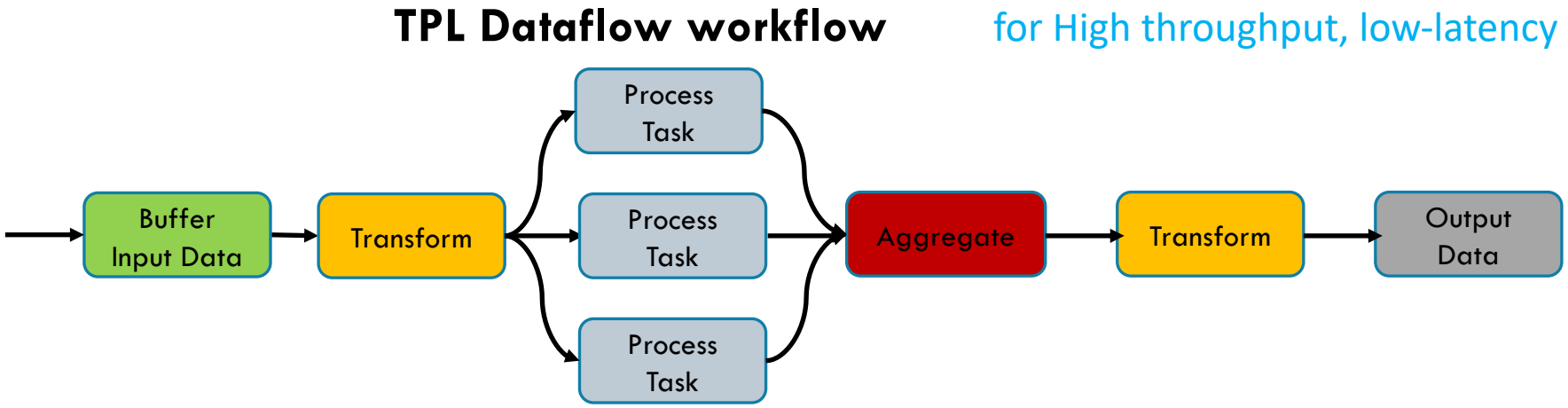
High Performance Stream Processing and Workflows with TPL Dataflow

RICCARDO TERRELL

TPL Dataflow … design to compose



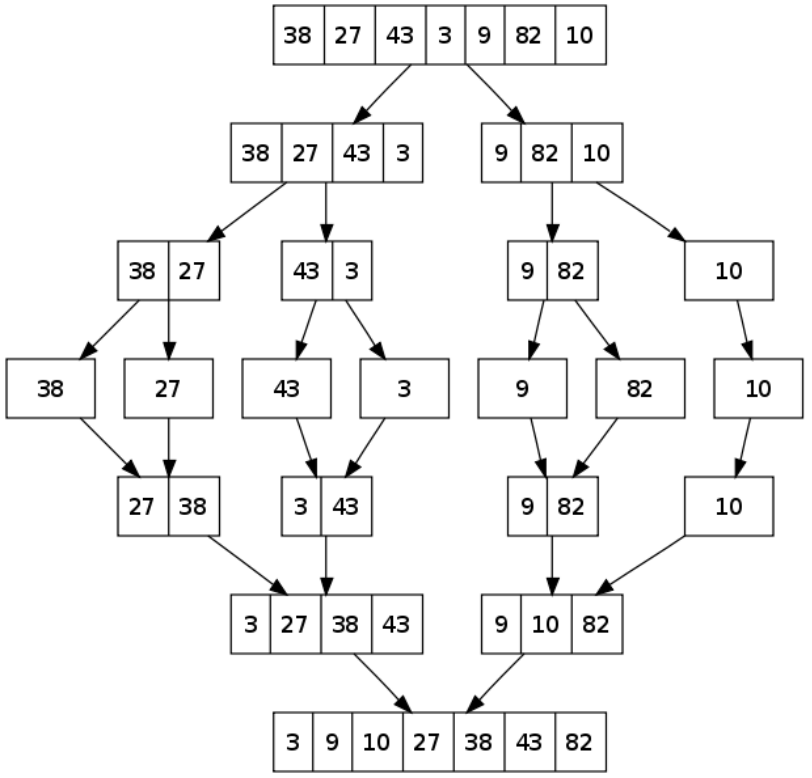
for High throughput, low-latency scenarios

implementing tailored asynchronous parallel workflow and batch queuing

Its about maximizing resource use

To get the **best** performance, your application has to partition and divide processing to take full advantage of multicore processors – enabling it to do **multiple** things at the same time, i.e. **concurrently**.

Divide and Conquer



When to use the TPL Dataflow

- Large amount of data to process and/or generate (Stream Data)
- To process any workflow that can take advantage of Parallel Computation
- When taking advantage of batch processing by breaking up each section of an application

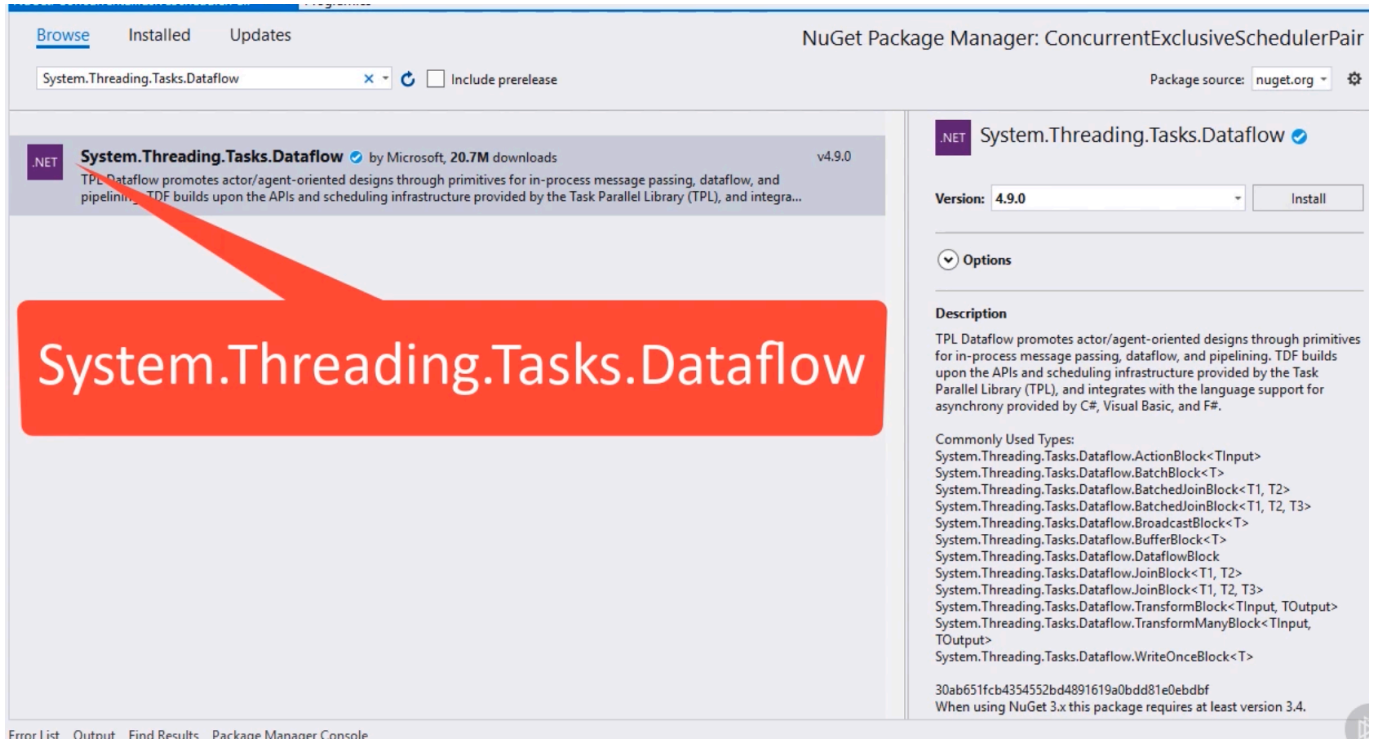
Why TPL Dataflow?

- Asynchronous programming evolution (for both I/O bound and CPU bound operations)
- Parallel and concurrent programming without Synchronization (no share of state)
- Reduce complexity of code structure (declarative)
- Process seamlessly large data in parallel with low latency

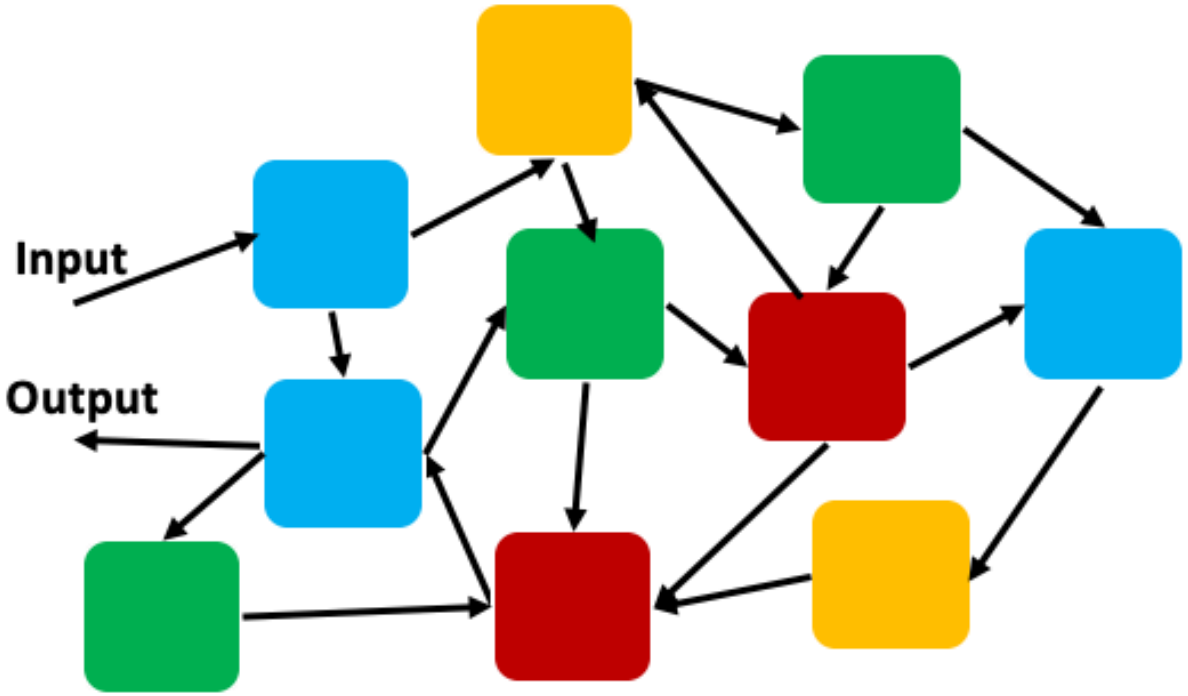
How to start?

TPL Dataflow

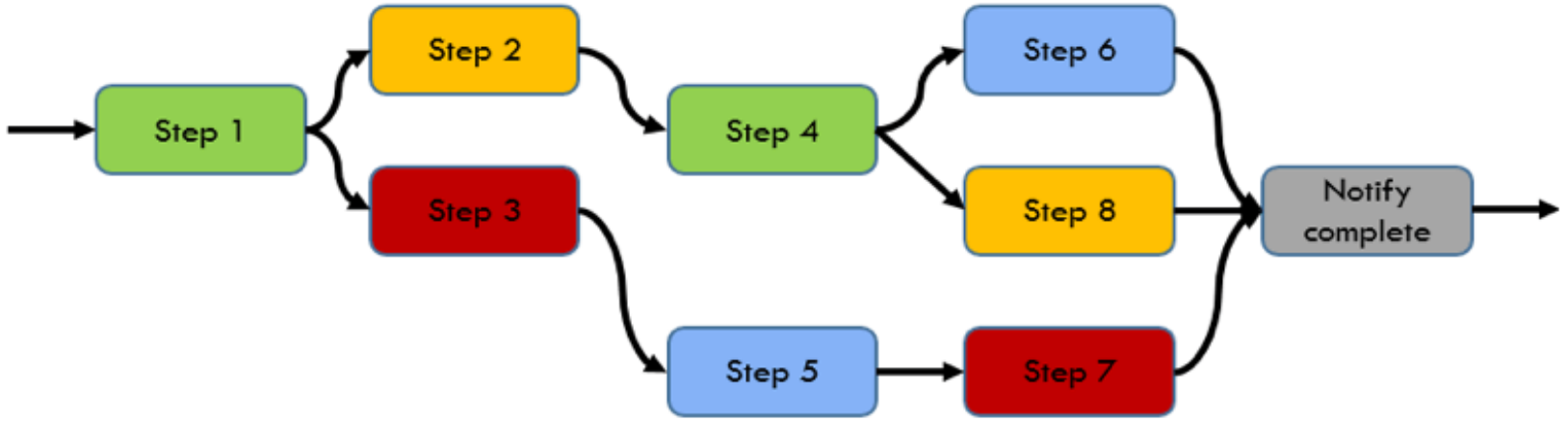
System.Threading.Tasks.Dataflow



TPL Dataflow promotes Message Passing programming model



TPL Dataflow Architecture Building Blocks



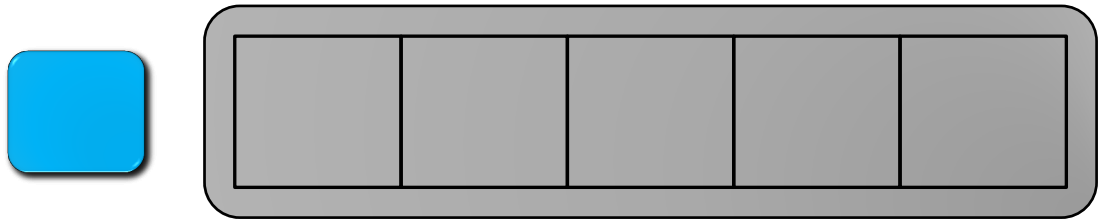
TPL Dataflow

Building Blocks

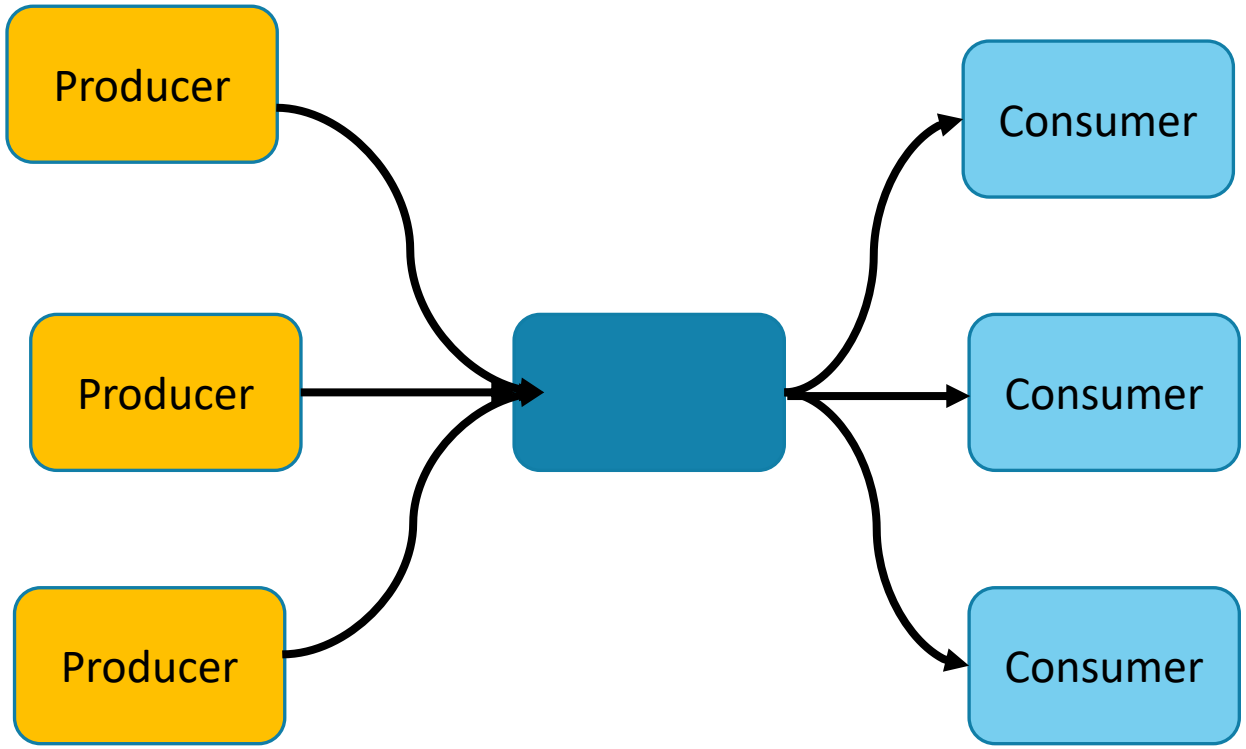
TPL Dataflow Architecture Building Blocks

```
public interface IDataflowBlock
{
    void Complete();
    void Fault(Exception error);
    Task Completion { get; }
}

public interface ISourceBlock<out TOutput> : IDataflowBlock
{
    bool TryReceive(out TOutput item, Predicate<TOutput> filter);
    bool TryReceiveAll(out IList<TOutput> items);
}

public interface ITargetBlock<in TInput> : IDataflowBlock
{
    DataflowMessageStatus OfferMessage(
        DataflowMessageHeader messageHeader, TInput messageValue,
        ISourceBlock<TInput> source, bool consumeToAccept);
}
```

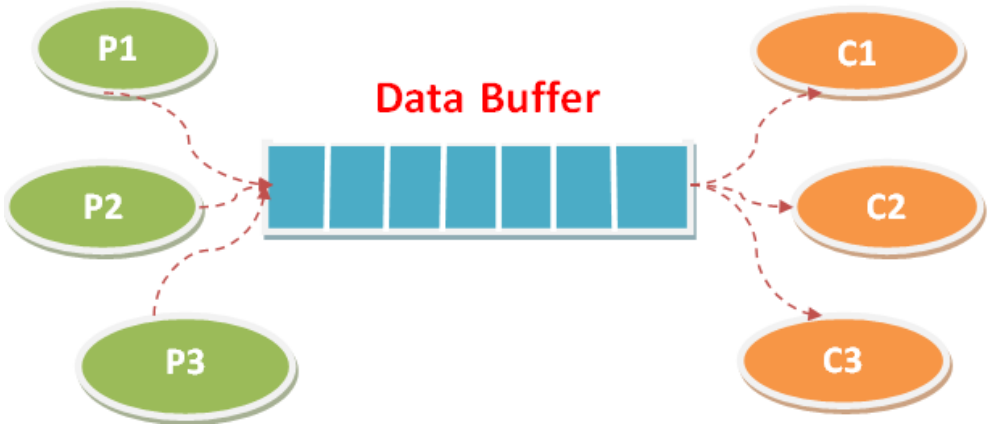
BufferBlock<T>



N producer
N consumers

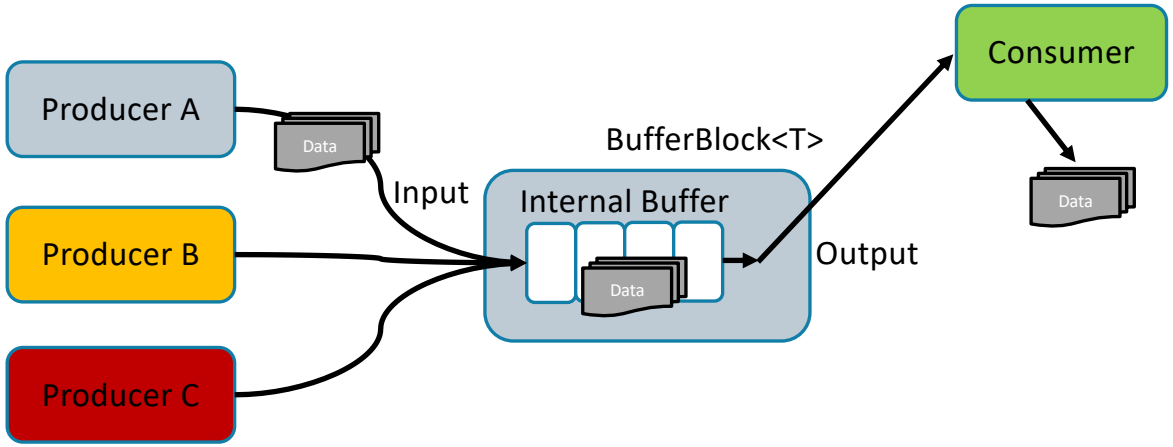


Produce Consumer



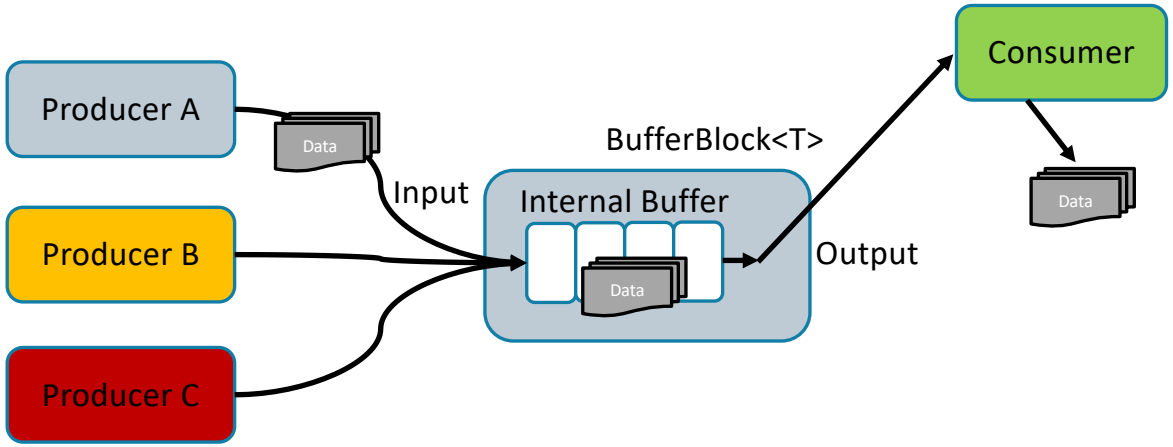

```
BlockingCollection<int> bCollection = new BlockingCollection<int>(10);

Task producerThread = Task.Factory.StartNew(() =>
{
    for (int i = 0; i < 10; ++i)
        bCollection.Add(i);

    bCollection.CompleteAdding();
});


Task consumerThread = Task.Factory.StartNew(() =>
{
    while (!bCollection.IsCompleted)
    {
        int item = bCollection.Take();
        Console.WriteLine(item);
    }
});


Task.WaitAll(producerThread, consumerThread);
```

BufferBlock<T> Producer-Consumer



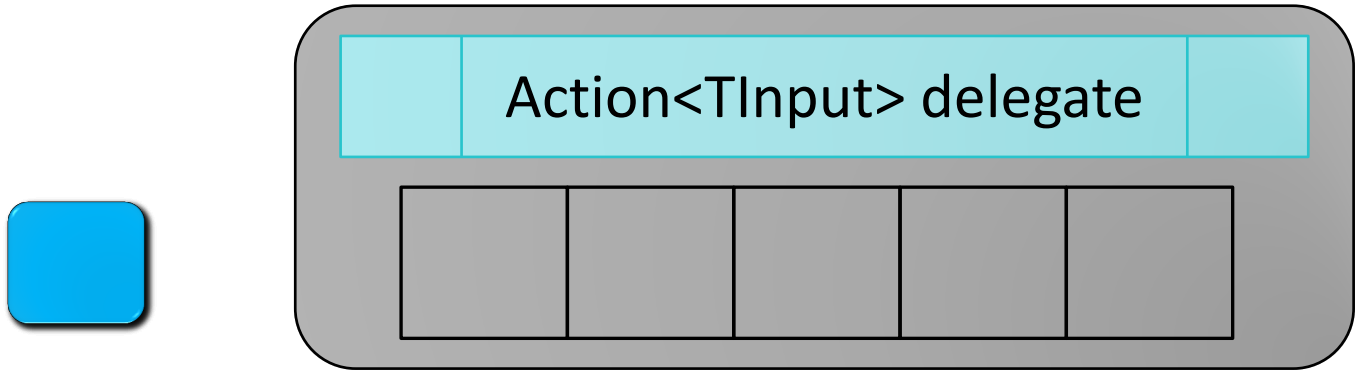

```
static BufferBlock<int> buffer = new BufferBlock<int>();

async Task Producer(IEnumerable<int> values) {
   foreach (var value in values)
      buffer.Post(value);
}

async Task Consumer(Action<int> process)  {
   while (await buffer.OutputAvailableAsync())
      process(await buffer.ReceiveAsync());
}

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

BufferBlock<T> Producer-Consumer



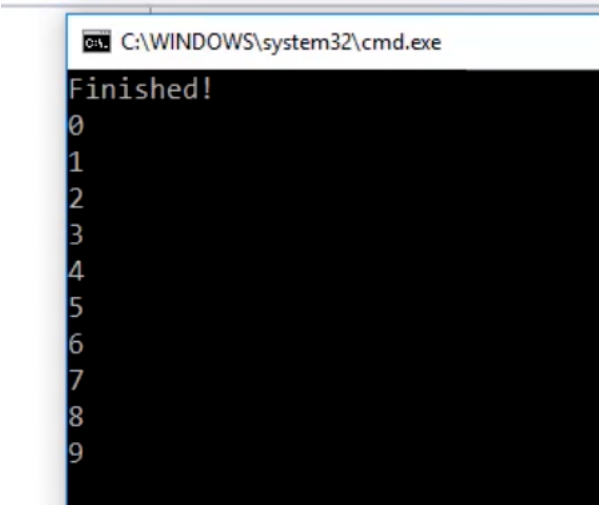

```
static BufferBlock<int> buffer = new BufferBlock<int>(
      new ExecutionDataflowBlockOptions { BoundedCapacity = 10 } );

async Task Producer(IEnumerable<int> values) {
   foreach (var value in values)
      await buffer.SendAsync(value);
}

async Task Consumer(Action<int> process)  {
   while (await buffer.OutputAvailableAsync())
      process(await buffer.ReceiveAsync());
}

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

ActionBlock<T>



ActionBlock<T>

```
var actionBlock = new ActionBlock<int>(n =>{
    Thread.Sleep(1000);
    Console.WriteLine(n);

});

for (int i = 0; i < 10; i++)
{
    actionBlock.Post(i);
}

Console.WriteLine(”Finished!");
```

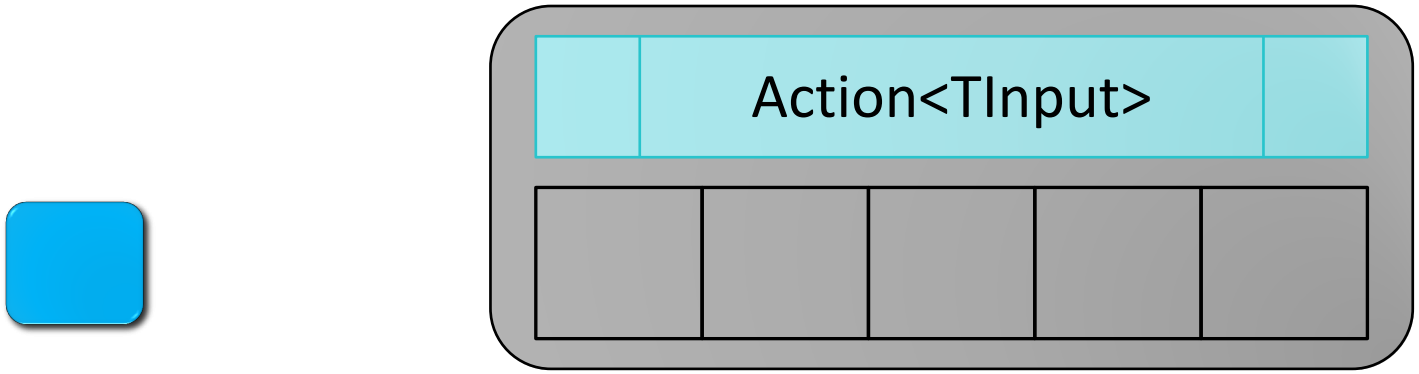


ActionBlock<T> Sync and Async

```
// Downloading Images Sequentially and Synchronously
var downloader = new ActionBlock<string>(url =>
{
   // Download returns byte[]
   byte [] imageData = Download(url);
   Process(imageData);
});

downloader.Post("http://msdn.com/concurrency");
downloader.Post("http://blogs.msdn.com/pfxteam");
```

```
// Downloading Images Sequentially and Asynchronously
var downloader = new ActionBlock<string>(async url =>
{
   byte [] imageData = await DownloadAsync(url);
   Process(imageData);
});

downloader.Post("http://msdn.com/concurrency ");
downloader.Post("http://blogs.msdn.com/pfxteam");
```

MaxDegreeOfParallelism

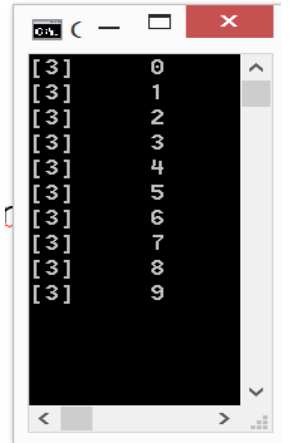


MaxDegreeOfParallelism = 2

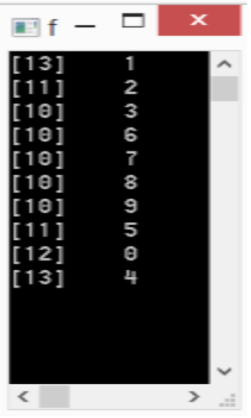
TPL Dataflow Parallelism

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

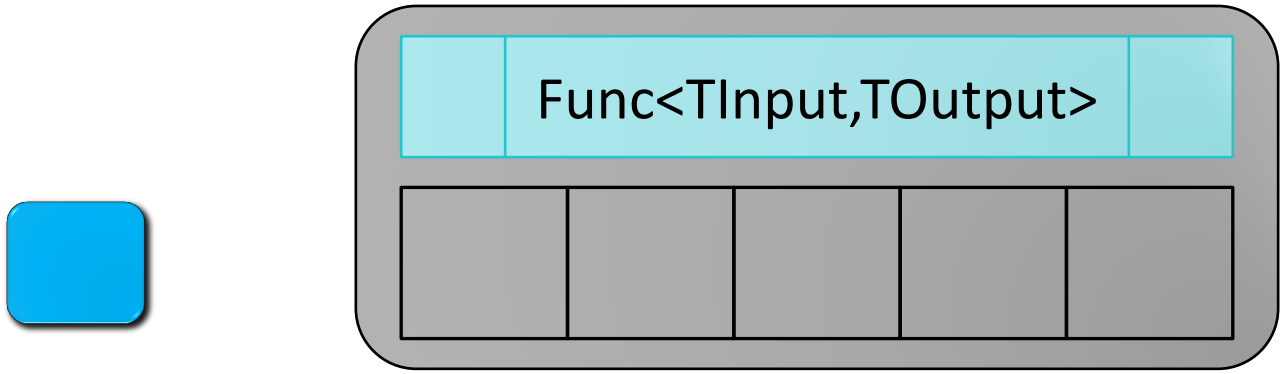
```
for (var i = 0; i < 10; i++) actionBlock.Post(i);
```




```
actionBlock.Options = new ExecutionDataflowBlockOptions() {
      MaxDegreeOfParallelism = 4 }
);

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



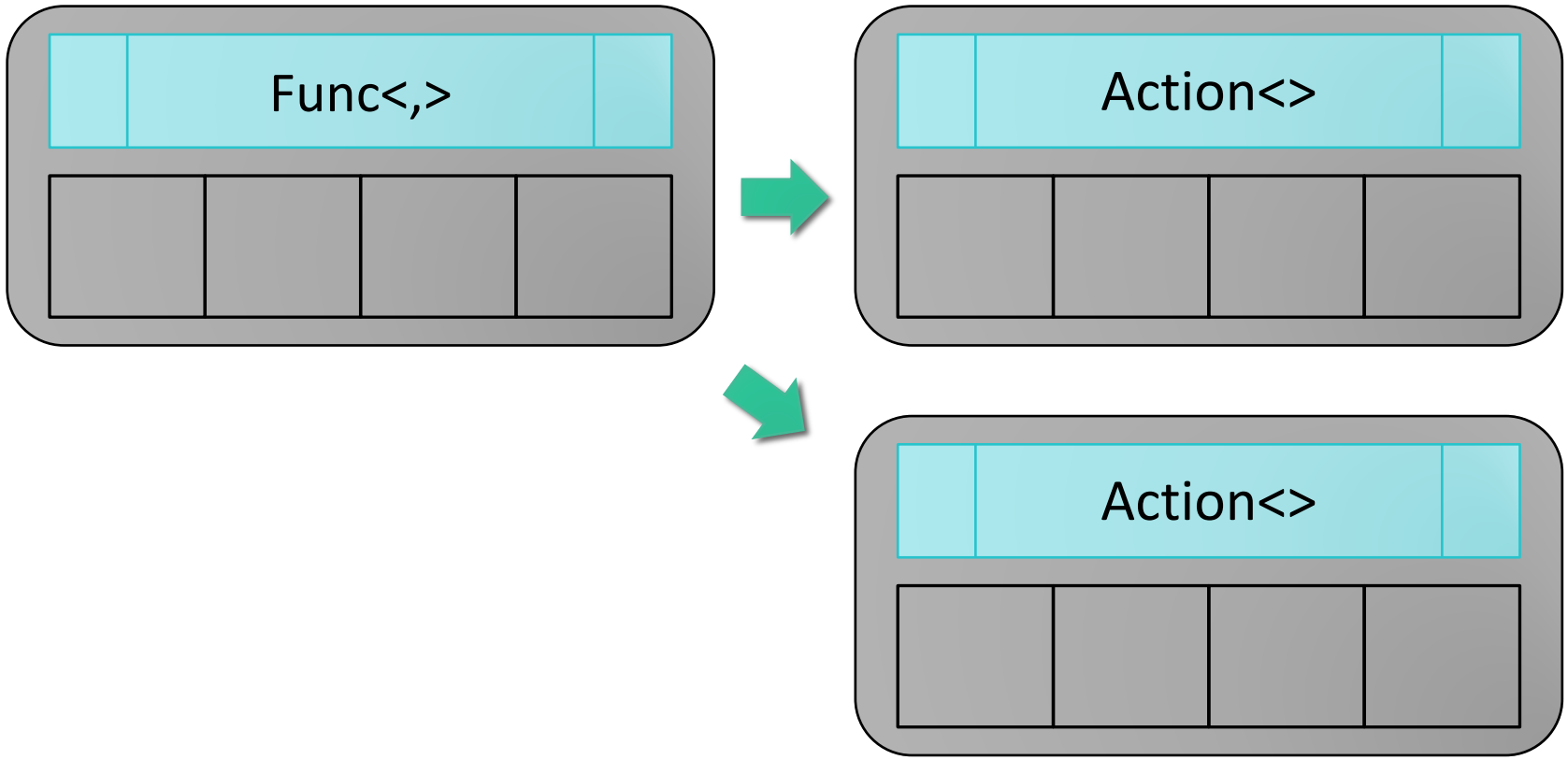
TransformBlock<T, R>



TransformBlock<T, R>

```
var tfBlock = new TransformBlock<int, string>( n =>
   {
      Thread.Sleep(500);
      return (n * n).ToString();
   }, new ExecutionDataflowBlockOptions() {MaxDegreeOfParallelism = 1});  // Change DOP

for (int i = 0; i < 10; i++)
{
   tfBlock.Post(i);
   Console.WriteLine($"Message {i} processed - queue count {actionBlock.InputCount}");
}
```

TransformBlock + ActionBlock



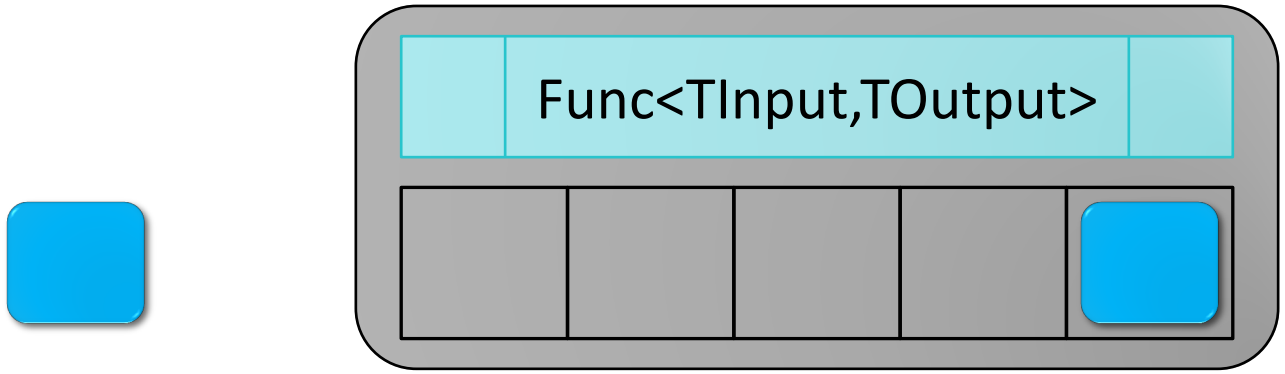
TransformBlock<T, R>

```
var actionBlock = new ActionBlock<int>(n =>
{
   Console.WriteLine($"Message : {n} - Thread Id#{Thread.CurrentThread.ManagedThreadId}");
});

var tfBlock = new TransformBlock<int, int>( n =>
   {
       return n * n;
   }, new ExecutionDataflowBlockOptions() {MaxDegreeOfParallelism = 1}); // Change DOP

tfBlock.LinkTo(actionBlock);

for (int i = 0; i < 10; i++)
{
   tfBlock.Post(i);
   Console.WriteLine($"Message {i} processed - queue count {actionBlock.InputCount}");
}
```

Link Filtering

```
var actionBlock1 = new ActionBlock<int>(n =>
{
    Console.WriteLine($"Message : {n} - Thread Id#{Thread.CurrentThread.ManagedThreadId}");
});

var actionBlock2 = new ActionBlock<int>(n =>
{
   Console.WriteLine($"Message : {n} - Thread Id#{Thread.CurrentThread.ManagedThreadId}");
});

var bcBlock = new TransformBlock<int, int>(n => n);

bcBlock.LinkTo(actionBlock1);
bcBlock.LinkTo(actionBlock2, n => n % 2 == 0);
```

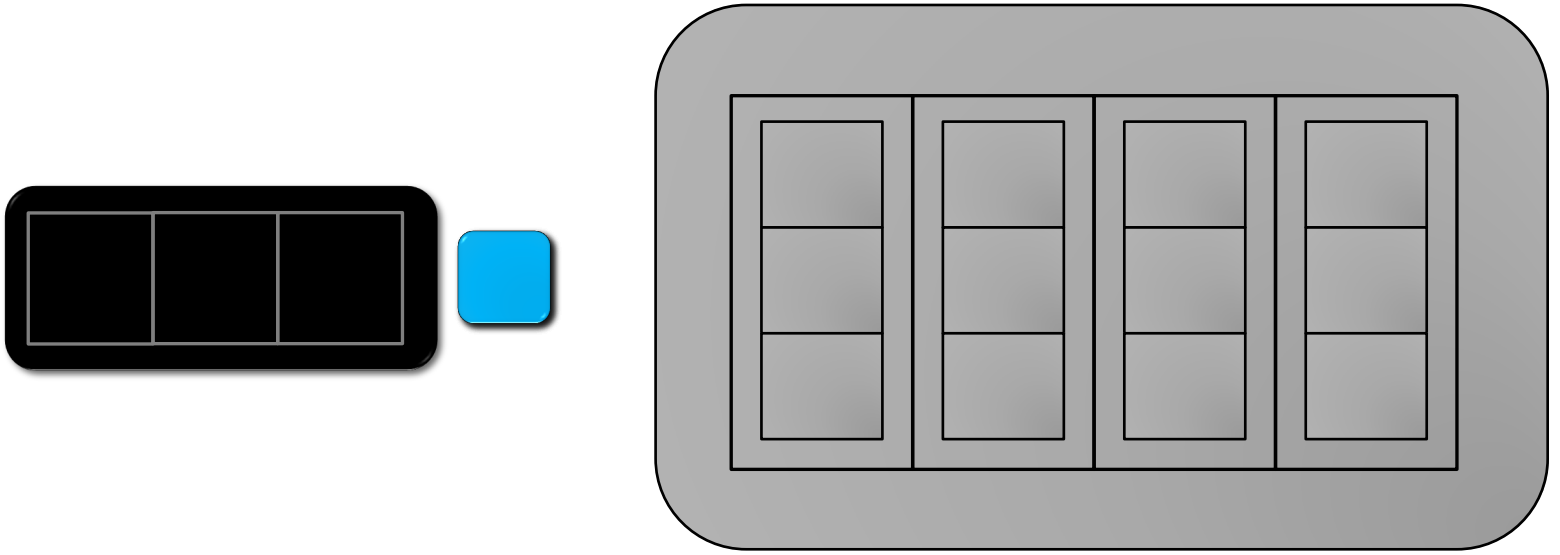
TransformManyBlock<T, R>



TransformManyBlock<T, R>

```
 // Asynchronous Web Crawler
var downloader = new TransformManyBlock<string,string>(async url =>
{
   Console.WriteLine("Downloading " + url);
   return ParseLinks(await DownloadContents(url));
});

var actionBlock = new ActionBlock<string>(n =>
{
   Console.WriteLine($"Message : {n} - Thread Id#{Thread.CurrentThread.ManagedThreadId}");
});

downloader.LinkTo(actionBlock);
```

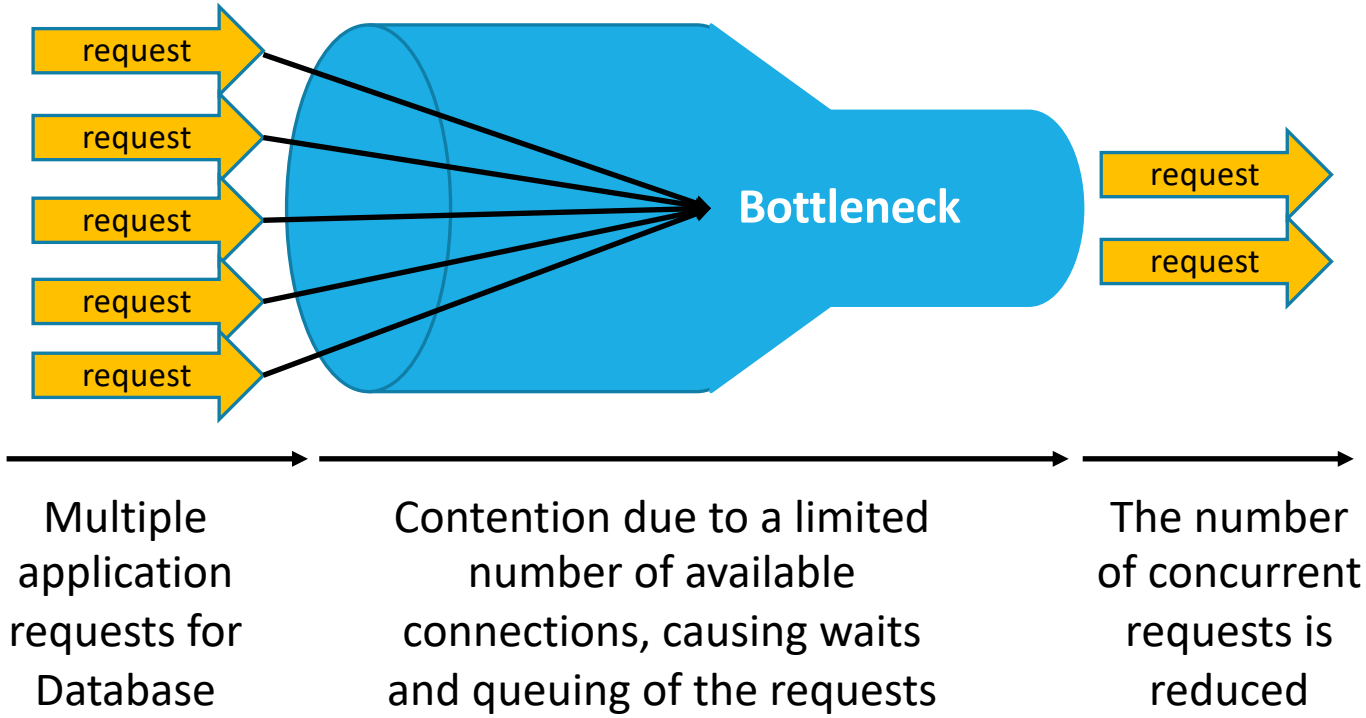
BatchBlock<T>



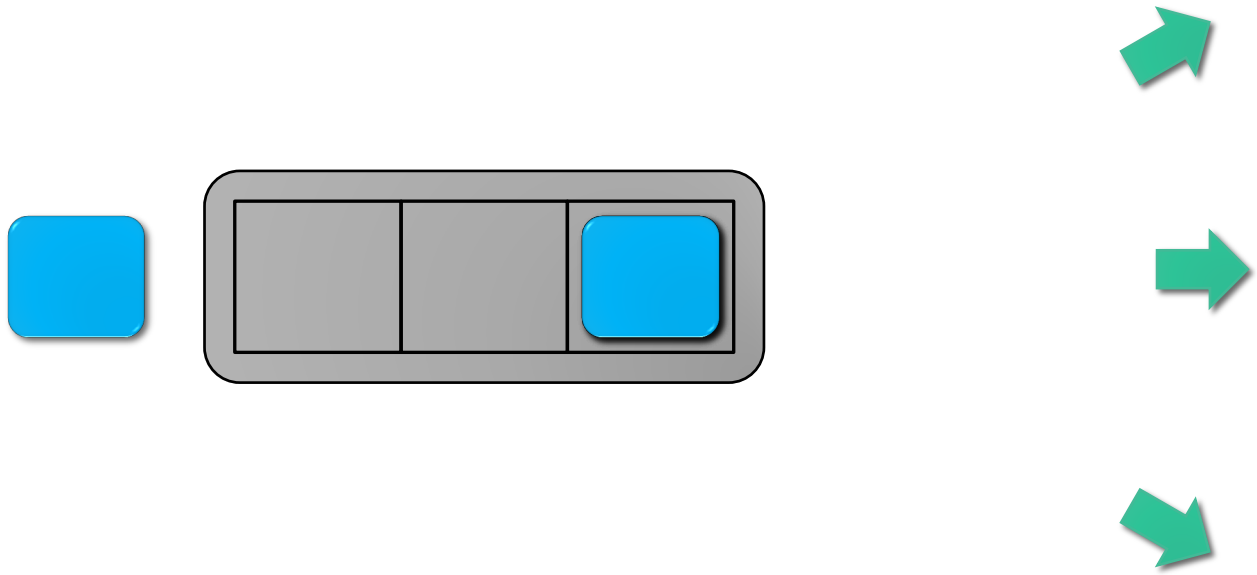
BatchBlock<T>

```
// Batching Requests into groups of 100 to Submit to a Database

var batchRequests = new BatchBlock<Request>(batchSize:100);

var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));

batchRequests.LinkTo(sendToDb);

for (int i = 0; i < 100; i++)
    batchRequests.Post(new Request (i));
```

BatchBlock<T>



BroadcastBlock<T>



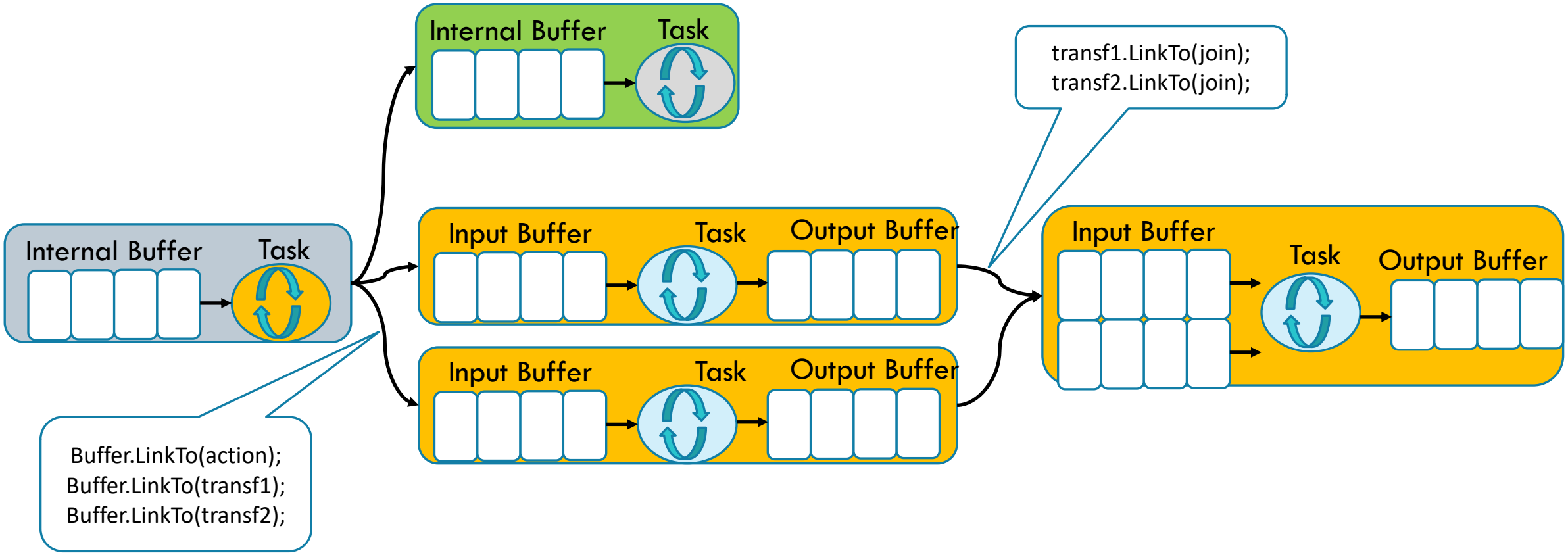
BroadcastBlock<T>

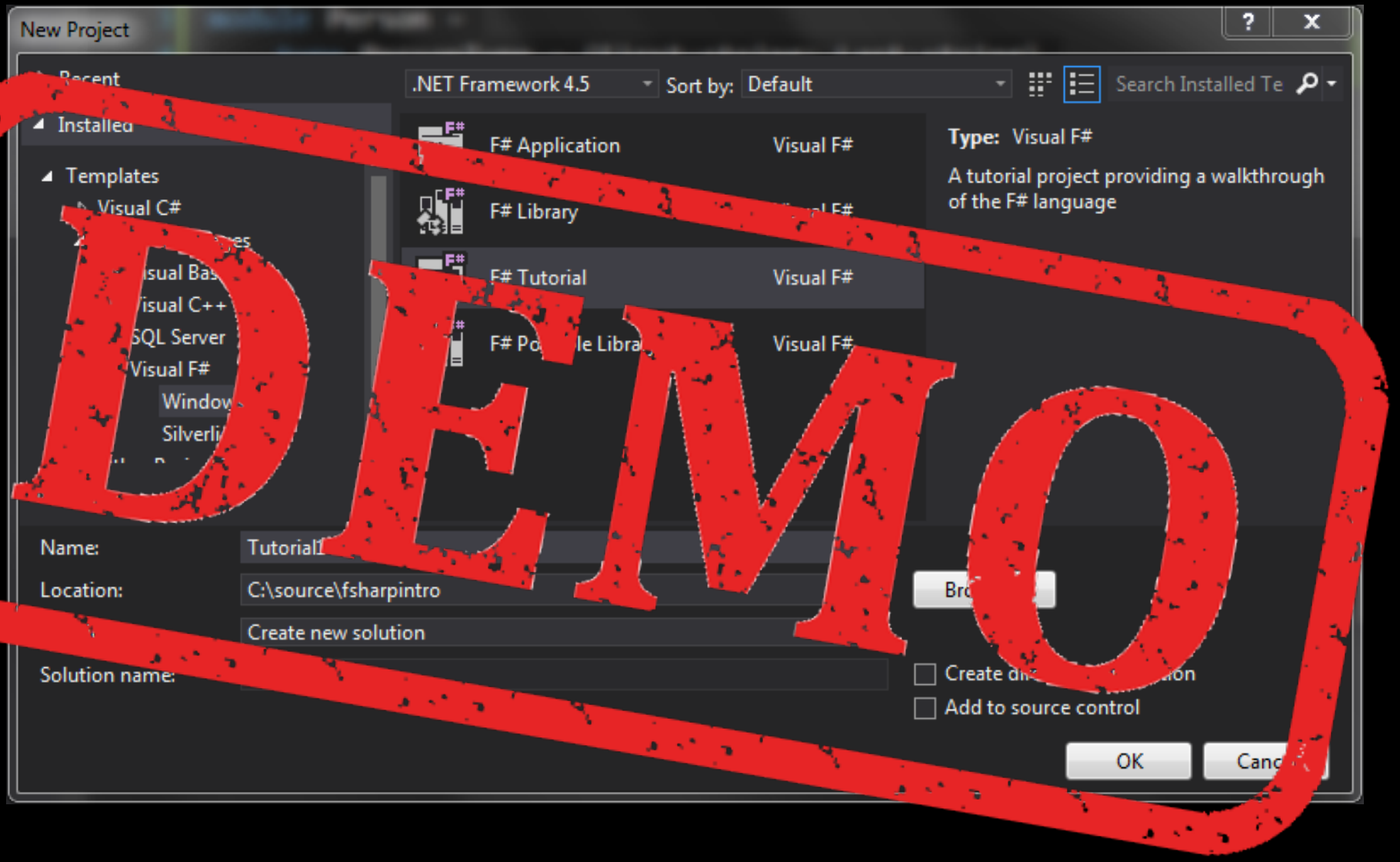
```
var bcBlock = new BroadcastBlock<int>(n => n);
var actionBlock1 = new ActionBlock<int>(n =>  Console.WriteLine($"Message {n} processed - ActionBlock 1"));
var actionBlock2 = new ActionBlock<int>(n =>  Console.WriteLine($"Message {n} processed - ActionBlock 3"));

bcBlock.LinkTo(actionBlock1);
bcBlock.LinkTo(actionBlock2);

for (int i = 0; i < 10; i++)
    bcBlock.Post(i);
```

Completion and Error propagation

```
var source = new BufferBlock<string>();
var actionBlock = new ActionBlock<string>(n =>
{
    Console.WriteLine($"Message : {n} - Thread Id#{Thread.CurrentThread.ManagedThreadId}");
});

// Completation is not propagated by default
source.LinkTo(actionBlock, new DataflowLinkOptions() { PropagateCompletion = true });

for (int i = 0; i < 10; i++)
{
   source.Post($"Item #{i}");
}
actionBlock.Completion.ContinueWith(a => Console.WriteLine("actionBlock completed"));
source.Complete();
actionBlock.Completion.Wait();
```

Building a Dataflow Network

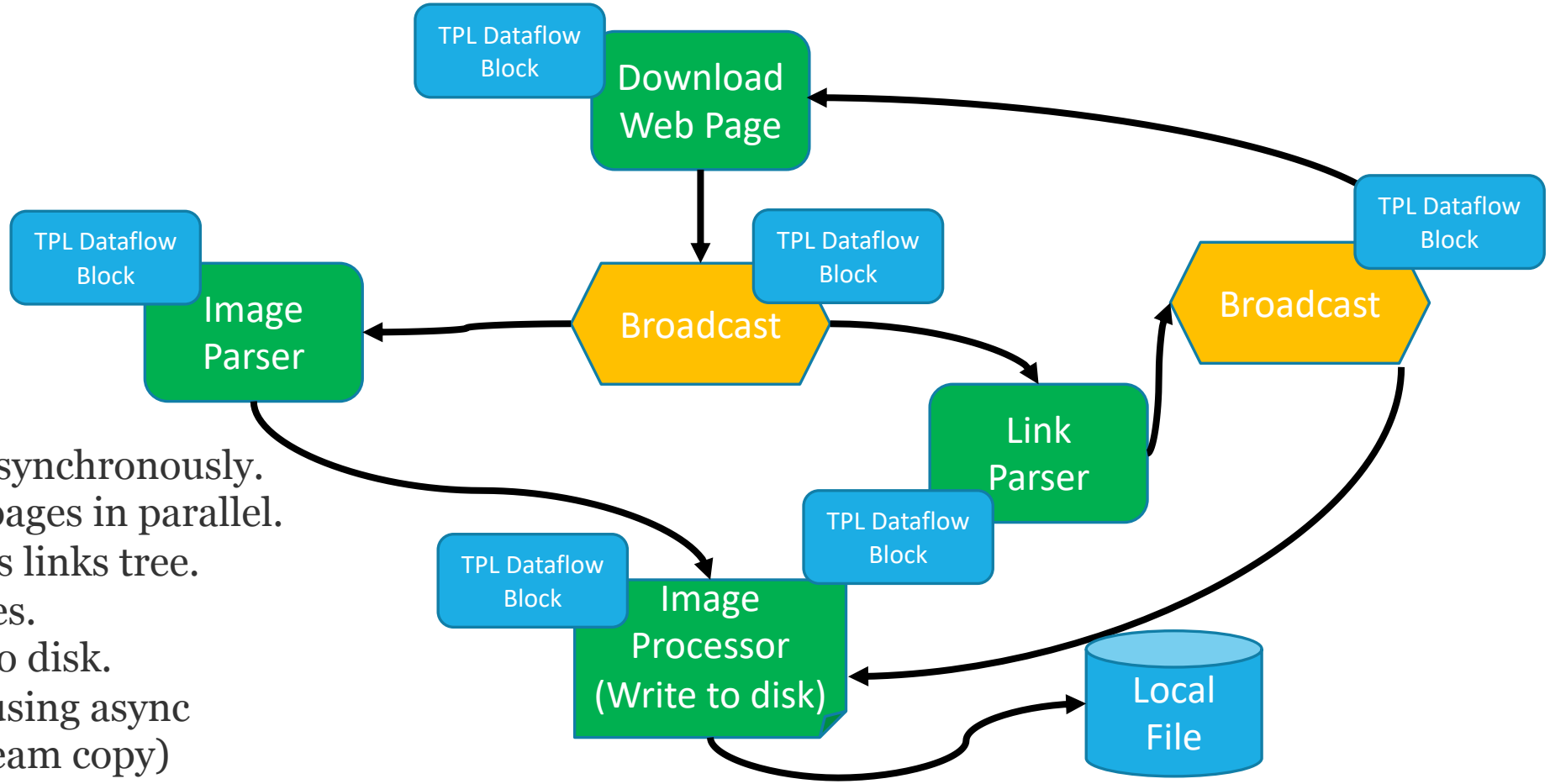



New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1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

Parallel Web-Crawler

A TPL Dataflow based WebCrawler

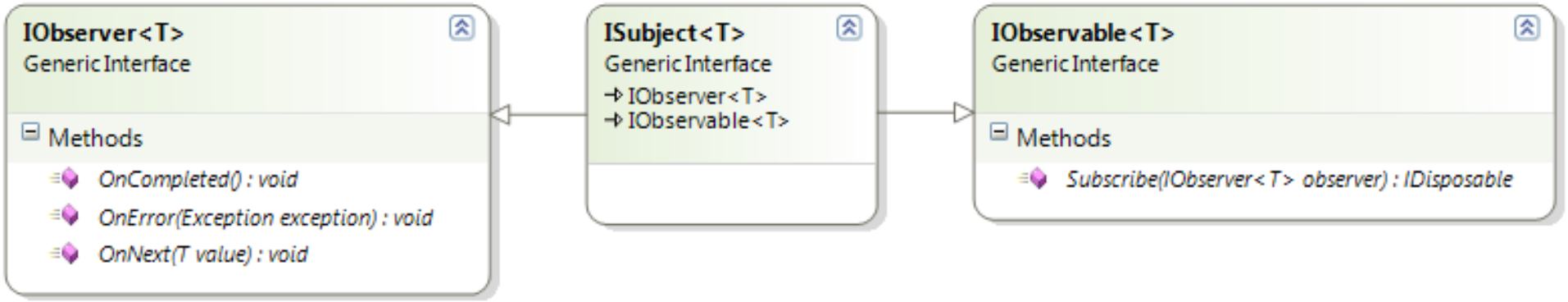
- Download web pages asynchronously.
- Download max 4 web pages in parallel.
- Traverse the web pages links tree.
- Parse for links to images.
- Download jpg images to disk.
- Download the images using async
  - (parallel async stream copy)

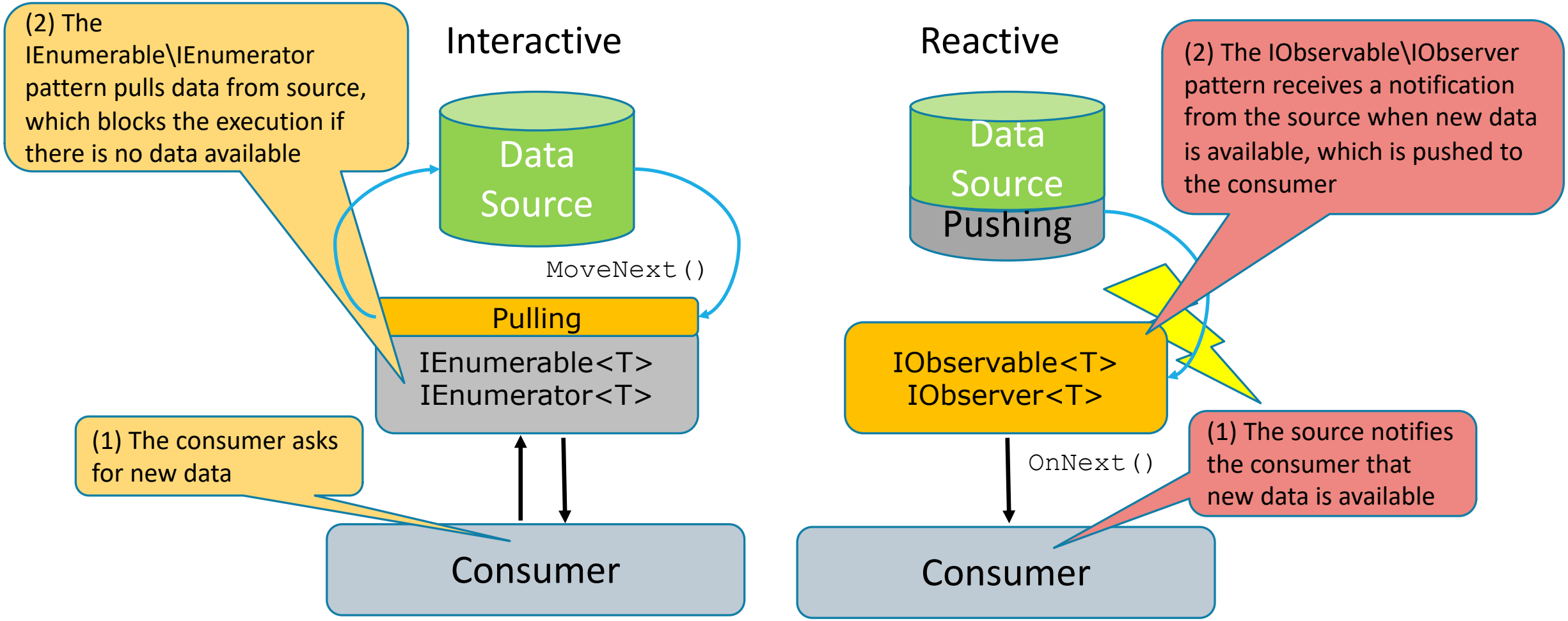


Reactive Extensions

What are Reactive Extensions

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




(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

TPL Dataflow & Reactive Extension

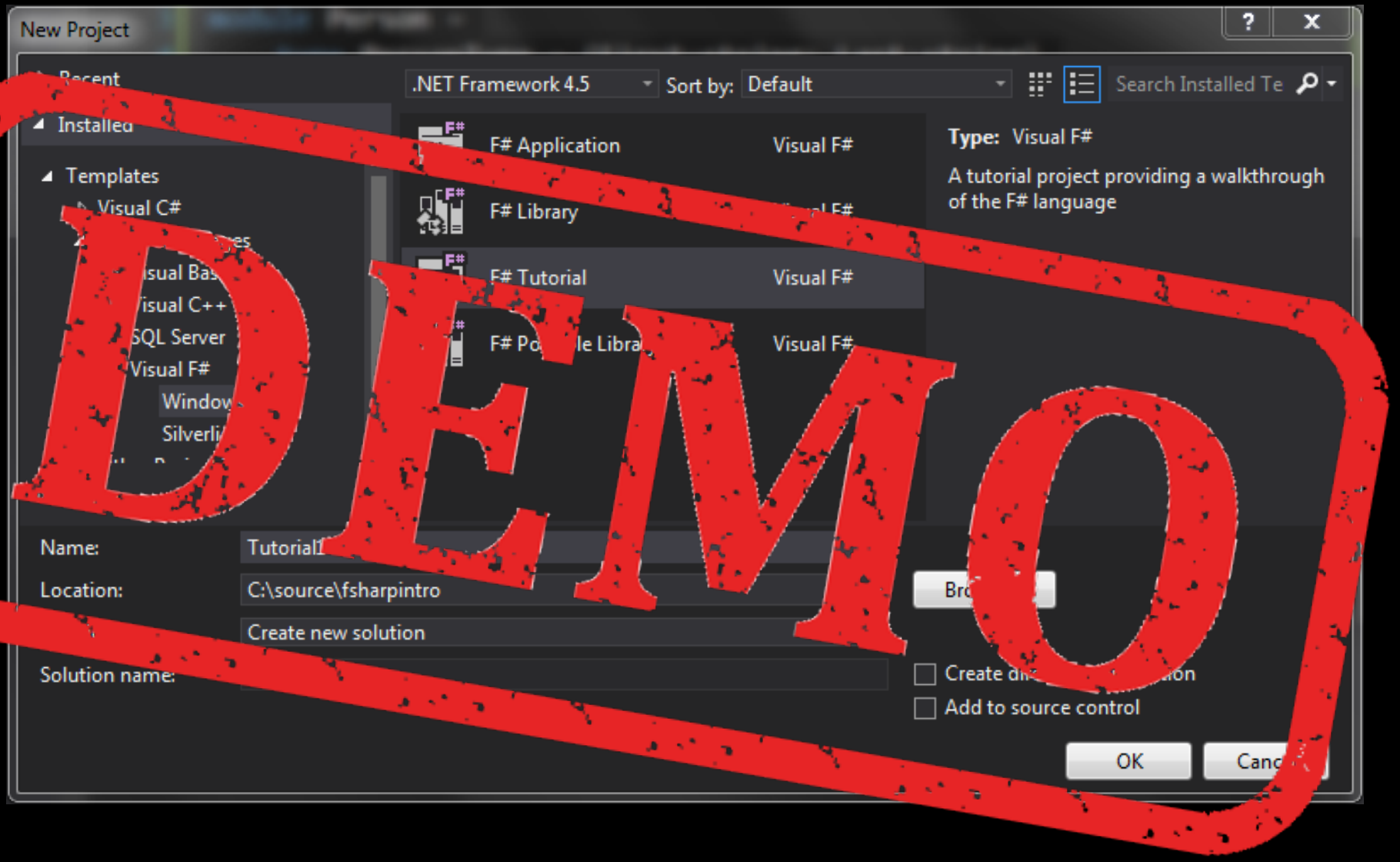
```
IPropagatorBlock<int, string> source =
            new TransformBlock<int, string>(i => (i + i).ToString());

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

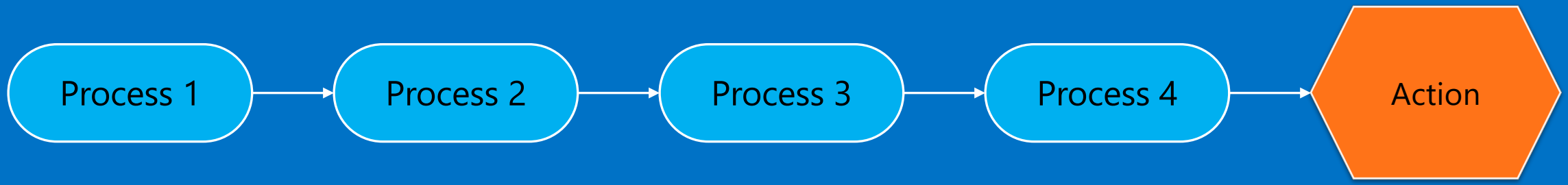
```
IPropagatorBlock<string, int> target =
                            new TransformBlock<string, int>(s => int.Parse(s));

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

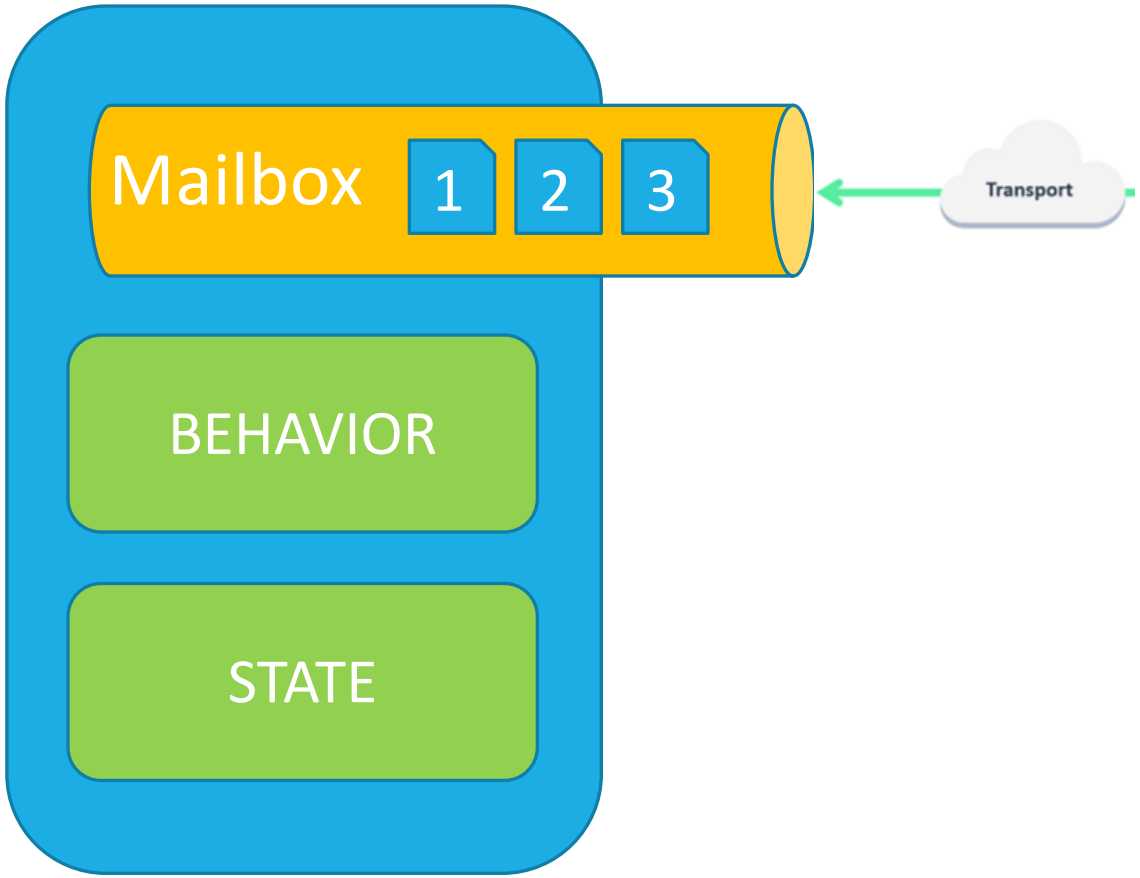
```
var encryptor = new TransformBlock<CompressDetails, EncryptDetails>();

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

encryptor.AsObservable()

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

         return (details, lastIndexProc);

}) .SingleAsync())
.SubscribeOn(TaskPoolScheduler.Default).Subscribe();
```


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
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:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

TPL Dataflow Agent in .NET

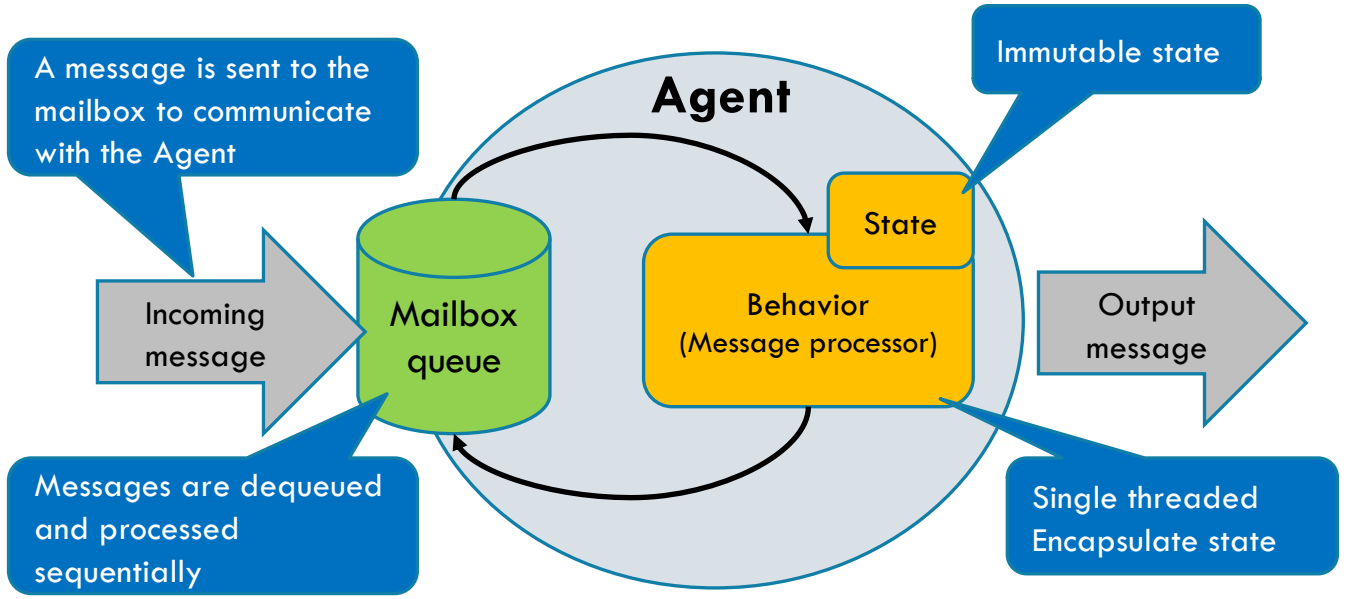


Message Passing based concurrency



- 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

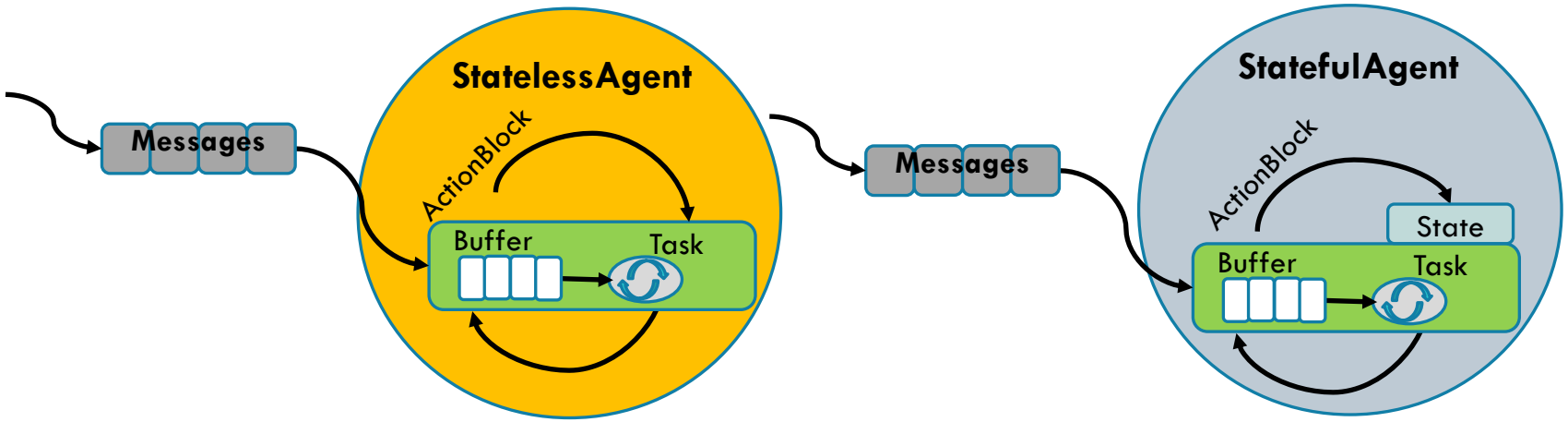
Agent anatomy

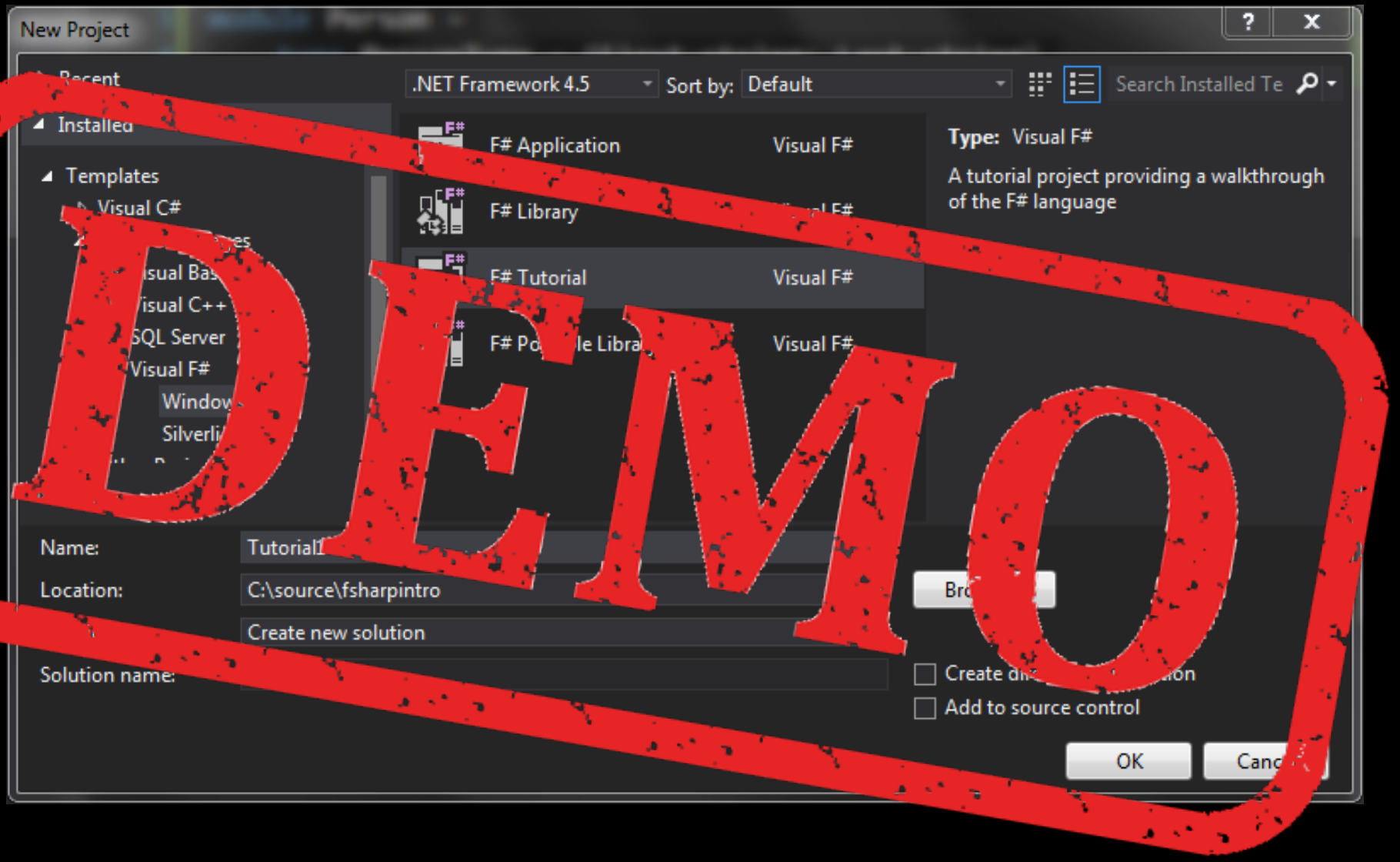


TPL DataFlow a statefull Agent in C#

```
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);
}
```

TPL DataFlow as Agent




New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
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:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

contacts

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

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

-- Edsger Dijkstra


That's all Folks!