Modern patterns of concurrent and parallel programming in .NET

BY RICCARDO TERRELL - @TRIKACE

terrellpc

40% off Functional Concurrency in .NET
(all formats)

https://www.manning.com/books/functional-concurrency-in-dotnet

Agenda

- Managed parallelism with Task Parallel Libraries
- Parallel Pipeline (multi producers/multi consumers)
- Combinators to compose Tasks and Asynchronous operations deterministically
- Message passing programming model

Imperative vs Functional

“I’ve parallelize my code, and now it runs 4x slower”

- on Twitter

“I asked my team to parallelize the code… NOT to paralyze the code”

- on Twitter

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

Concurrency programming

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

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

Parallel and concurrent

Parallel programming
- Run CPU-intensive tasks on multi-core faster

Asynchronous programming
- Avoid blocking of threads when waiting for I/O

Reactive programming
- User-interfaces with lots of cheap event handling

Concurrent programming
- Make interactive programs more scalable

Parallel programming
- Task Parallel Library (TPL)

Asynchronous programming
- F# asynchronous workflows and C# async/await

Reactive programming
- Agent/Actor and Reactive Extensions (Rx)

Concurrent programming
- All the above !

Concurrency paradigm
&
Functional Programming

Twitter Paper

Your Server as a Function

Marius Eriksen

Twitter Inc.

marius@twitter.com

Abstract

Building server software in a large-scale setting, where systems exhibit a high degree of concurrency and environmental variability, is a challenging task to even the most experienced programmer. Efficiency, safety, and robustness are paramount—goals which have traditionally conflicted with modularity, reusability, and flexibility.

We describe three abstractions which combine to present a pow-

Services Systems boundaries are represented by asynchronous functions called *services*. They provide a symmetric and uniform API: the same abstraction represents both clients and servers.

Filters Application-agnostic concerns (e.g. timeouts, retries, authentication) are encapsulated by *filters* which compose to build services from multiple independent modules.

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

Twitter Paper

Your Server as a Function

Servers in a large-scale setting are required to process tens of thousands, if not hundreds of thousands of requests concurrently; they need to handle partial failures, adapt to changes in network conditions, and be tolerant of operator errors. As if that weren't enough, harnessing off-the-shelf software requires interfacing with a heterogeneous set of components, each designed for a different purpose. These goals are often at odds with creating modular and reusable software [6].

We present three abstractions around which we structure our server software at Twitter. They adhere to the style of functional programming—emphasizing immutability, the composition of first-class functions, and the isolation of side effects—and combine to present a large gain in flexibility, simplicity, ease of reasoning, and robustness.

Abstract

Building server software
hibit a high degree of con
a challenging task to eve
ficiency, safety, and robu
traditionally conflicted w
We describe three abs

presented by asynchronous
ovide a symmetric and uni-
represents both clients and

(e.g. timeouts, retries, au-
ters which compose to build
modules.

Objectives / Motivations

- Use functions as building block and **composition** to solve complex problems with simple code
- **Immutability** and **Isolation** are your best friends to write concurrent applications
- **Functional** and **imperative** styles should NOT stand in opposition … they **should COEXIST**
  - Poly-paradigm programming is more powerful and effective than polyglot programming

Moore’s law -The classical computer is reaching quantum state

"The number of transistors incorporated in a chip will approximately double every 24 months."
-- Gordon Moore, Intel Co- Founder

Transistors and the “End of Moore’s Law”

32 nm
22 nm
14 nm
10 nm
7 nm
5 nm
END

In a 22nm transistors there are about 50 silicon atoms between source and drain

Different schedulers for different OS

- Processes (tasks) are **run in an infinite loop**
- If task takes too much time (B), **need to divide** it
- **No overhead** occurs

- **Scheduling** is performed by the OS
- Long task is **suspended if needed**
- **Overhead**: caused by OS internals

MULTITHREADING
THEORY

PRACTICE

The issue is Shared Memory

- Shared Memory Concurrency
- Data Race / Race Condition
- Works in sequential single threaded environment
- Very difficult to parallelize
- Difficult to maintain and test
- Locking, blocking, call-back hell

Immutability

Like string in C# **everything** is immutable

Help in concurrent/parallel code

Immutability is **default** in F#

You can choose to go mutable (**use carefully!**)

Copy-on-Write

Immutability

```
public class ImmutablePerson
{
    public ImmutablePerson(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string LastName { get; }
    public string FirstName { get; }
    public int Age { get; }

    public ImmutablePerson UpdateAge(int age)
    {
            return new ImmutablePerson(this.FirstName, this.LastName, age);
    }
}
```

Immutability

```
public class ImmutablePerson
{
    public ImmutablePerson(string firstName, string lastName, int age) {
        FirstName = firstName;
        LastName = lastName;
        Age = age;
    }
    public string LastName { get; }
    public string FirstName { get; }
    public int Age { get; }

    public ImmutablePerson UpdateAge(int age)
    {
        return new ImmutablePerson(this.FirstName, this.LastName, age);
    }
}
```

```
type Person = {FirstName:string; LastName:string; Age:int}
```

Compare And Swap - CAS

```
void DoWithCAS<T>(ref T location, Func<T, T> generator)
                                        where T : class
      {
          T temp, replace;
          do
          {
              temp = location;
              replace = generator(temp);
          } while (Interlocked.CompareExchange(ref location,
                                  replace, temp) != temp);
      }
```

Compare And Swap - CAS

```
public class LockFreeStack<T> {
    private class Node {
        public T Data;
        public Node Next;
    }
    private Node head;

    public void Push(T element) {
        Node node = new Node {Data = element};
        DoWithCAS(ref head, h => {
            node.Next = h;
            return node;
        });
    }
```

Parallel Loop & Compare And Swap - CAS

```
var results = new AtomImmutable<TResult>(ImmutableArray<TResult>.Empty);
Parallel.ForEach(partitioner,
    pos,
    () => new List<TResult>(),
    (item, loopState, local) =>
    {
        local.Add(selector(item));
        return local;
    },
    // final => results.Swap(o => o.AddRange(final))
    final => DoWithCas(ref final, o => o.AddRange(final))
    );
```

Avoid Side Effects

… embrace referential transparency

Referential transparency

- An expression is said to be referentially transparent if it can be **replaced** with its corresponding value **without changing the program's behavior**

- As a result, evaluating a referentially transparent function gives the same value for same arguments.

- In general it means that the function **don’t have side effect** which may influence its result.

Referential transparency

Which of the following methods are an example of Referential Transparency?

```
DateTime Add(DateTime date, TimeSpen duration) =>
                        date.Add(duration);

DateTime AddDay(DateTime date) =>
                        Add(date, TimeSpan.FromDay(1));

DateTime Tomorrow() => AddDay (DateTime.Now)

DateTime AddTrigger(DateTime date, Func<int> f) =>
                        Add(date, TimeSpan.FromDay(f()));
```

Parallel words counter with side effects

Parallel words counter with side effects

```
public static Dictionary<string, int> WordsCounter(string source)
{
    var wordsCount =
            (from filePath in
                Directory.GetFiles(source, "*.txt").AsParallel()
                from line in File.ReadLines(filePath)
                from word in line.Split(' ')
                select word.ToUpper())
        .GroupBy(w => w)
        .OrderByDescending(v => v.Count()).Take(10);
    return wordsCount.ToDictionary(k => k.Key, v => v.Count());
}
```

Parallel words Counter with side effects

```
static Dictionary<string, int> PureWordsPartitioner
                              (IEnumerable<string> content) =>
    (from lines in content.AsParallel()
        from line in lines
        from word in line.Split(' ')
        select word.ToUpper())
               .GroupBy(w => w)
              .OrderByDescending(v => v.Count()).Take(10)
               .ToDictionary(k => k.Key, v => v.Count());

static Dictionary<string, int> WordsPartitioner(string source) {
    var contentFiles =
       from filePath in Directory.GetFiles(source, "*.txt")
             select File.ReadLines(filePath);
    return PureWordsPartitioner(contentFiles);
}
```

Parallel Task Programming

Task and Data Parallelism

Divide and Conquer

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


    SwapElements(items, left, pivot);

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

Quick Sort

```
static void QuickSort<T>(T[] items, int left, int right)
{
    if (right - left < 2)
    {
        if (left+1 == right &&
            items[left].CompareTo(items[right]) > 0)
            Swap(ref items[left], ref items[right]);
        return;
    }

    int pivot = Partition(items, left, right);

    QuickSort (items, left, pivot);
    QuickSort (items, pivot + 1, right);
}
```

Quick Sort

```
static void QuickSort_Parallel<T>(T[] items, int left, int right)
{
    if (right - left < 2)
    {
        if (left+1 == right &&
            items[left].CompareTo(items[right]) > 0)
            Swap(ref items[left], ref items[right]);
        return;
    }

    int pivot = Partition(items, left, right);

    Task leftTask = Task.Run(() => QuickSort_Parallel(items, left, pivot));
    Task rightTask = Task.Run(() => QuickSort_Parallel(items, pivot + 1, right));

    Task.WaitAll(leftTask, rightTask);
}
```

Quick Sort – Oversubscription

Preemption

What is ‘Hill Climbing’

Quick Sort – with depth

```
void QuickSort_Parallel_Threshold<T>(T[] items, int left, int right, int depth)
{
    if (left >= right)  return;
    SwapElements(items, left, (left + right) / 2);
    var pivot = Partition(items);
    if (depth >= 0)     {
        QuickSort_Parallel_Threshold(items, left, pivot - 1, depth);
        QuickSort_Parallel_Threshold(items, pivot + 1, right, depth);
    }
    else
    {
        --depth;
        Parallel.Invoke(
                () => QuickSort_Parallel_Threshold(items, left, pivot - 1, depth),
                () => QuickSort_Parallel_Threshold(items, pivot + 1, right, depth)
            );
    }
```

Quick Sort benchmark

Task parallel programming in action

Sequential Fuzzy Match

```
List<string> matches = new List<string>();
foreach (var word in WordsToSearch)
{
    var localMathes = JaroWinklerModule.bestMatch(Words, word);
    matches.AddRange(localMathes.Select(m => m.Word));
}
```

Fuzzy match 7 words against 13.4 Mb of text

Time execution in 4 Logical cores – 6 Gb Ram : **23,167** ms

Multi Thread Fuzzy Match

```
List<string> matches = new List<string>();
var threads = new Thread[Environm      Processo

for (int i = 0; i < threads.Lengt
                    var index = i

      threa

      var take = Words                                          hreads.Length));
      var start = index                                          e : index * take;


      foreach                                                   e)) {
                                                                bestMatch(Words, word);
                  matches                                       ect(                rd));

                      }
                  }

for (int i = 0;    < threads.Leng    ; i++)
    threads[i].Start();
for (int i = 0; i < threads.Length; i++)
   threads[i].Join();
```

WRONG!!!

Time execution : **6,857** ms

Multi Thread Fuzzy Match

```
List<string> matches = new List<string>();
var threads = new Thread[Environment.ProcessorCount];

for (int i = 0; i < threads.Length; i++) {
      var index = i;
      threads[index] = new Thread(() =>   {

      var take = WordsToSearch.Count / (Math.Min(WordsToSearch.Count, threads.Length));
      var start = index == threads.Length - 1 ? WordsToSearch.Count - take : index * take;


      foreach (var word in WordsToSearch.Skip(start).Take(take)) {
              var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, word);
              lock (matches)
                      matches.AddRange(localMathes.Select(m => m.Word));
                      }
                  });
                }

 for (int i = 0; i < threads.Length; i++)
    threads[i].Start();
for (int i = 0; i < threads.Length; i++)
   threads[i].Join();
```

Time execution : **7,731** ms

Parallel Loop Fuzzy Match

```
 List<string> matches = new List<string>();
object sync = new object();

Parallel.ForEach(WordsToSearch,
                 // thread local initializer
                 () => { return new List<string>(); },
                 (word, loopState, localMatches) =>  {
                      var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, word);
                        localMatches.AddRange(localMathes.Select(m => m.Word));// same code
                        return localMatches;
                 },
    (finalResult) =>
    {
        // thread local aggregator
        lock (sync) matches.AddRange(finalResult);
    }
);
```

Time execution : **7,429** ms

PLINQ Fuzzy Match

```
ParallelQuery<string> matches =
        (from word in WordsToSearch.AsParallel()
         from match in JaroWinklerModule.bestMatch(Words, word)
         select match.Word);
```

Time execution : **6,347** ms

Multi Task Fuzzy Match

```
var tasks = new List<Task<List<string>>>();
var matches = new ThreadLocal<List<string>>(() => new List<string>());

foreach (var word in WordsToSearch) {

        tasks.Add(Task.Factory.StartNew<List<string>>((w) => {
             List<string> localMatches = matches.Value;
             var localMathes = FuzzyMatch.JaroWinklerModule.bestMatch(Words, w);
             localMatches.AddRange(localMathes.Select(m => m.Word));
             return localMatches;
    }, word));
}

Task.WhenAll(tasks.ToArray(), (ts) =>
      return new List<string>(tasks.SelectMany(t => t.Result).Distinct()));
```

Time execution : **4,492** ms

Partitioning

Partitioning in PLINQ

Range Partitioning

Chunk Partitioning

Striped Partitioning

Hash Partitioning

PLINQ Fuzzy Match

```
Create a load-balancing partitioner, specify false for static partitioning.

Partitioner<string> partitioner = Partitioner.Create(WordsToSearch, false);

ParallelQuery<string> matches =

       (from word in partitioner.AsParallel()

        from match in JaroWinklerModule.bestMatch(Words, word)

        select match.Word);
```

Time execution : **4,217** ms

PLINQ Fuzzy Match

```
Create a load-balancing partitioner, specify false for static partitioning.

Partitioner<string> partitioner = Partitioner.Create(WordsToSearch, false);

ParallelQuery<string> matches =

        (from word in partitioner.AsParallel()

         from match in JaroWinklerModule.bestMatch(Words, word)

         select match.Word);
```

Time execution : **4,217** ms

Deforesting your query

Deforesting your query

```
var data = new int[100000];
for(int i = 0; i < data.Length; i++)
    data[i]=i;


long total =
    data.AsParallel()
        .Where(n => n % 2 == 0)
        .Select(n => n + n)
        .Select(n=> (long)n)
        .Sum();
```

Lab intro :

Task continuation
Think Continuations, not Callbacks

Continuation Passing Style

```
void GetMaxCPS(int x, int y, Action<int> f)
{
    if (x > y)
        f(x);
    else
        f(y);
}
```

Task continuation

```
void RunContinuation<T>(Func<T> input, Action<T> rest)
        => ThreadPool.QueueUserWorkItem(new WaitCallback(o => rest(input())));


Action<string> printMsg = msg =>
    Console.WriteLine($"ThreadID = {Thread.CurrentThread.ManagedThreadId}, Url = {url}, {msg}");

RunContinuation(() => {
    printMsg("Creating webclient...");
    return new System.Net.WebClient();
}, (webclient) => RunContinuation(() => {
        printMsg("Downloading url...");
        return webclient.DownloadString(url);
    }, (html) => RunContinuation(() => {
            printMsg("Extracting urls...");
            return Regex.Matches(html, @"http://\S+");
        }, (matches) =>
                printMsg("Found " + matches.Count.ToString() + " links")
            )));
```

Task continuation

```
ThreadID = 19, Url = http://www.google.com/, Creating webclient...
ThreadID = 21, Url = http://microsoft.com/, Creating webclient...
ThreadID = 16, Url = http://www.wordpress.com/, Creating webclient...
ThreadID = 18, Url = http://www.wordpress.com/, Downloading url...
ThreadID = 11, Url = http://www.google.com/, Downloading url...
ThreadID = 19, Url = http://microsoft.com/, Downloading url...
ThreadID = 15, Url = http://www.wordpress.com/, Extracting urls...
ThreadID = 16, Url = http://www.wordpress.com/, Found 15 links
ThreadID = 18, Url = http://www.peta.org, Creating webclient...
ThreadID = 05, Url = http://www.peta.org, Downloading url...
ThreadID = 19, Url = http://www.google.com/, Extracting urls...
ThreadID = 19, Url = http://www.google.com/, Found 14 links
ThreadID = 20, Url = http://www.peta.org, Extracting urls...
ThreadID = 20, Url = http://www.peta.org, Found 60 links
ThreadID = 21, Url = http://microsoft.com/, Extracting urls...
ThreadID = 21, Url = http://microsoft.com/, Found 1 links
```

Task continuation

```
Task.Run(() =>
{
    printMsg("Creating webclient...");
    return new System.Net.WebClient();
}).ContinueWith(taskWebClient =>
{
    var webclient = taskWebClient.Result;
    printMsg("Downloading url...");
    return webclient.DownloadString(url);
}).ContinueWith(htmlTask =>
{
    var html = htmlTask.Result;
    printMsg("Extracting urls...");
    return Regex.Matches(html, @"http://\S+");
}).ContinueWith(rgxTask =>
{
    var matches = rgxTask.Result;
    printMsg("Found " + matches.Count.ToString() + " links");
});
```

Task continuation

```
var matches = from webclient in Task.Run(() => new System.Net.WebClient())
              from html in Task.Run(() => webclient.DownloadString(url))
              from rgx in Task.Run(() => Regex.Matches(html, @"http://\S+"))
              select rgx;
```

Lab – Faster Fuzzy Match

Composing Tasks – Then

Task SelectMany

```
static member Then(input : Task<'T>, binder : 'T -> Task<'U>) =
     let tcs = new TaskCompletionSource<'U>()
     input.ContinueWith(fun (task:Task<'T>) ->
        if (task.IsFaulted) then
          tcs.SetException(task.Exception.InnerExceptions)
        elif (task.IsCanceled) then tcs.SetCanceled()
        else
             try
                (binder(task.Result)).ContinueWith(fun
(nextTask:Task<'U>) -> tcs.SetResult(nextTask.Result)) |> ignore
             with
             | ex -> tcs.SetException(ex)) |> ignore
     tcs.Task
```

Task async/await is a monad

```
static async Task<R> Then<T, R>(this Task<T> task,
        Func<T, Task<R>> then) => await Bind(await task);

static async Task<R> Then<T1, T2, R>(this Task<T1> task,
         Func<T1, Task<T2>> bind, Func<T1, T2, R> project)
        {
            T taskResult = await task;
            return project(taskResult, await bind(taskResult));
        }

static async Task<R> Then<T, R>(this Task<T> task, Func<T, R> project)
            => await Map(task, project);

static async Task<R> Return<R>(R value) => Task.FromResult(value);
```

Task Inline - ExecuteSynchronously

```
public static Task<TOut> Then<TIn, TOut>(this Task<TIn> first, Func<TIn, Task<TOut>>
next)
{
    var tcs = new TaskCompletionSource<TOut>();
    first.ContinueWith(delegate
    {
        if (first.IsFaulted) tcs.TrySetException(first.Exception.InnerExceptions);
        else if (first.IsCanceled) tcs.TrySetCanceled();
        else
        {
                next(first.Result).ContinueWith(t =>
                {
                    if (t.IsFaulted) tcs.TrySetException(t.Exception.InnerExceptions);
                    else if (t.IsCanceled) tcs.TrySetCanceled();
                    else tcs.TrySetResult(t.Result);
                }, TaskContinuationOptions.ExecuteSynchronously);
    }, TaskContinuationOptions.ExecuteSynchronously);
    return tcs.Task;
}
```

Mathematical patterns for better Tasks composition

Function Composition

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

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

How can we compose Tasks?

```
Task<int> RunOne() => // ...
Task<decimal> RunTwo(int input) => // ...

var result = RunTwo(RunOne()); // Error!!
```

C#

```
Task<string> RunOne(int input) =>  // ...
Task<bool> RunTwo(string input) => // ...

Func<T1, T3> Compose<T1, T2, T3>(Func<T1, T2> f1,
                                 Func<T2, T3> f2) => a => f2(f1(a));

var result = RunOne(42).Compose(RunTwo); // Error!!
```

C#

Composing Tasks

Task<R> **Bind**<T, R>(this Task<T> m, Func<T, Task<R>> k) …

Task<T> **Return**(T value) …

Functional Design Patterns – Monad

Generic type

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

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

Task Bind and Task Return

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

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

SelectMany

```
// M<T> * (T -> M<U>) -> M<U>
IEnumerable<R> SelectMany<T, R>( this IEnumerable<T> source,
                                 Func<T, IEnumerable<R>> selector )


// M<T> * (T -> M<U>) -> (T -> M -> U) -> M<U>
IEnumerable<R> SelectMany<T, M, R>( this IEnumerable<T> source,
              Func<T, IEnumerable<M>> collectionSelector, Func<T, M, R> selector )


// M<T> * (T -> M<U>) -> M<U>
IEnumerable<R> Select<T, R>( this IEnumerable<T> source, Func<T, R> selector )
```

Task SelectMany

```
// M<T> * (T -> M<U>) -> M<U>
Task<R> Task<T, R>( this Task<T> source,
                                        Func<T,  Task<R>> selector )


// M<T> * (T -> M<U>) -> (T -> M -> U) -> M<U>
Task<R> SelectMany<T, M, R>( this Task<T> source,
                Func<T,  Task<M>> collectionSelector, Func<T, M, R> resultSelector )


// M<T> * (T -> M<U>) -> M<U>
Task<R> Select<T, R>( this Task<T> source, Func<T, R> selector )
```

Composing Tasks

```
from image in Task.Run<Emgu.CV.Image<Bgr, byte>()
from imageFrame in Task.Run<Emgu.CV.Image<Gray, byte>>()
from faces in Task.Run<System.Drawing.Rectangle[]>()
select faces;
```

Composing Task async/await

```
static Task<T> Return<T>(T task)=> Task.FromResult(task);


static async Task<R> Bind<T, R>(this Task<T> task,
    Func<T, Task<R>> cont)
     => await cont(await task.ConfigureAwait(false))
                                        .ConfigureAwait(false);

static async Task<R> Map<T, R>(this Task<T> task, Func<T, R> map)
=> map(await task.ConfigureAwait(false));
```

Composing Task async/await

```
static async Task<R> SelectMany<T, R>(this Task<T> task,
        Func<T, Task<R>> then) => await Bind(await task);

static async Task<R> SelectMany<T1, T2, R>(this Task<T1> task,
         Func<T1, Task<T2>> bind, Func<T1, T2, R> project)
        {
            T taskResult = await task;
            return project(taskResult, await bind(taskResult));
        }

static async Task<R> Select<T, R>(this Task<T> task, Func<T, R> project)
            => await Map(task, project);

static async Task<R> Return<R>(R value) => Task.FromResult(value);
```

Task Inline - ExecuteSynchronously

```
public static Task<TOut> SelectMany<TIn, TOut>(this Task<TIn> first, Func<TIn,
Task<TOut>> next)
{
    var tcs = new TaskCompletionSource<TOut>();
    first.ContinueWith(delegate
    {
        if (first.IsFaulted) tcs.TrySetException(first.Exception.InnerExceptions);
        else if (first.IsCanceled) tcs.TrySetCanceled();
        else
        {
                next(first.Result).ContinueWith(t =>
                {
                    if (t.IsFaulted) tcs.TrySetException(t.Exception.InnerExceptions);
                    else if (t.IsCanceled) tcs.TrySetCanceled();
                    else tcs.TrySetResult(t.Result);
                }, TaskContinuationOptions.ExecuteSynchronously);
    }, TaskContinuationOptions.ExecuteSynchronously);
    return tcs.Task;
}
```

LAB : SelectMany Tasks
&
Run Tasks as they complete

Task.WhenAny - Interleaving

```
List<Task<Bitmap>> imageTasks =
        (from imageUrl in urls select GetBitmapAsync(imageUrl)).ToList();

while(imageTasks.Count > 0)
{
        Task<Bitmap> imageTask = await Task.WhenAny(imageTasks);
        imageTasks.Remove(imageTask);
        Bitmap image = await imageTask;
        ProcessImage(image);
}
```

Handle Tasks as they complete

```
string[] stocks = new[] { "MSFT", "FB", "AAPL","GOOG", "ORCL" };

List<Task<Tuple<string, StockData[]>>> stockHistoryTasks =
    stocks.Select(ProcessStockHistory).ToList();

while (stockHistoryTasks.Count > 0)
{
    Task<Tuple<string, StockData[]>> stockHistoryTask =
        await Task.WhenAny(stockHistoryTasks);

     stockHistoryTasks.Remove(stockHistoryTask);

    Tuple<string, StockData[]> stockHistory = await stockHistoryTask;

    ShowChartProgressive(stockHistory);
}
```

Performance overheads with Task.WhenAny

```
public static Task<Task<T>>[] Interleaved<T>(IEnumerable<Task<T>> tasks)
{
    var inputTasks = tasks.ToList();
    var buckets = new TaskCompletionSource<Task<T>>[inputTasks.Count];
    var results = new Task<Task<T>>[buckets.Length];
    for (int i = 0; i < buckets.Length; i++)
    {
        buckets[i] = new TaskCompletionSource<Task<T>>();
        results[i] = buckets[i].Task;
    }
    int nextTaskIndex = -1;
    Action<Task<T>> continuation = completed =>  {
        var bucket = buckets[Interlocked.Increment(ref nextTaskIndex)];
        bucket.TrySetResult(completed);
    };

    foreach (var inputTask in inputTasks)
        inputTask.ContinueWith(continuation, TaskContinuationOptions.ExecuteSynchronously, TaskScheduler.Default);
    return results;
}
```

```
foreach(var bucketinInterleaved(tasks))
{
    var t = await bucket;
    Process(await t);
}
```

Lab : Task Pipeline

Pipeline Processing

- A pipeline is a set of data processing elements connected in series, so that the output of one element is the input of the next one. The elements of a pipeline are often executed in parallel or in time-sliced fashion; in that case, some amount of buffer storage is often inserted between elements

Parallel Pipeline

```
int BufferSize = ...
var buffer1 = new BlockingCollection<string>(BufferSize);
var buffer2 = new BlockingCollection<string>(BufferSize);
var buffer3 = new BlockingCollection<string>(BufferSize);
var f = new TaskFactory(TaskCreationOptions.LongRunning);

var stage1 = f.StartNew(() => ReadStrings(buffer1, ...));
var stage2 = f.StartNew(() => CorrectCase(buffer1, buffer2));
var stage3 = f.StartNew(() => CreateSentences(buffer2, buffer3));
var stage4 = f.StartNew(() => WriteSentences(buffer3));
Task.WaitAll(stage1, stage2, stage3, stage4);
```

Enable Multi-Producers/Multi-Consumers

```
BlockingCollection<int>[] sourceArrays = new BlockingCollection<int>[5];
for(int i = 0; i < sourceArrays.Length; i++)
        sourceArrays[i] = new BlockingCollection<int>(500);

Parallel.For(0, sourceArrays.Length * 500, (j) => {
        int k = BlockingCollection<int>.TryAddToAny(sourceArrays, j);
        if(k >=0) // do something here
});

int i = BlockingCollection<TInput>.TryTakeFromAny(m_input, out receivedItem);
if ( i >= 0) {
         // do something here
}
```

Asynchronous Programming

Techniques & Best Practices

CPU-bound vs I/O-bound

Synchronous vs Asynchronous

Scalability
is the ability to cope and perform under an increasing workload

Its all about Scalability

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

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

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

Concurrent Server Side programming

Reliability and scalability : effort is needed to ensure that applications don't stop working. The throughput needs to scale as new clients start using the service.

Concurrent requests : server-side applications must be designed to enable handling multiple requests concurrently. A program. With a single sequential control flow is bad design.

I/O bound operations : server-side applications exploit large number of I/O operations, which may occupy the largest portion of the request-processing time. This applies to receiving input from the client and sending a response back.

Shared state : servers often exist to enable state sharing among a large number of client. This means that concurrently running processes in a server-side application need to be able to safely access shared states.

Brain Teasers

?????

Message Passing Concurrency

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 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

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

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	Clustering and Remoting across multiple machines

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 [])
```

Immutability OR Isolation

```
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()
    )
```

Comparison between Sequential, Task-based and Message passing programming

Sequential Programming

Task-Based Programming

Message-Passing Programming

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 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)
```

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
```

Agent in .NET and C#

TPL DataFlow blocks –design to compose

Some Dataflow blocks

Building a Dataflow Network

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}")));
}
```

Agent anatomy

Agent two-way communication

TPL DataFlow as Agent

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

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 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;
});
```

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);
                    });
```

Lab :

TPL DataFlow & Reactive Extensions

TPL Dataflow & Reactive Extension

Reactive Extensions

(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

IObserver & IObservable

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

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

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();
```