Modern patterns of concurrent and parallel programming in .NET

BY RICCARDO TERRELL - @TRIKACE

“I’ve spent several hours on attempts to parallelize my code, but it only get slower when I do so”

- on StackOverflow

“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?

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**.

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.

Agenda

- Functional paradigm related to concurrent programming
- Managed parallelism with Task Parallel Libraries
- Unbounded parallel computation with Asynchronous operations
- Message passing programming model (and Reactive Extensions)

Disclaimer

- Let’s keep the session interactive
- Skipping slides
- After this class you will need to keep practicing
- This is not just an introduction

Introduction - Riccardo Terrell

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

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

MVP
Microsoft®
Most Valuable Professional

terrellpc

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

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

What about you?

Concurrency Core Concepts

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 !

Functional Core Concepts

Imperative vs Functional

“It’s really clear that the imperative style of programming has run its course. … We’re sort of done with that. … However, in the declarative realm we can speculate a 10x improvement in productivity in certain domains”

— Anders Hejlsberg
(C# Architect)

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.

Concurrency paradigm and Functional Programming

BY RICCARDO TERRELL- @TRIKACE

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 Concurrency challenge

Modern computers come with multiple processors, each equipped with multiple cores, but the single processor is slower than it used to be!

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

Data Race

Two concurrent accesses to a memory location where at least one of them is a write.

Example:

```
int x = 1;
Parallel.Invoke(
                 () => { x = 2; },
                 () => { System.Console.WriteLine(x);}
                );
```

writes x

reads x

Possible outcomes?

- 1
- 2

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

Structural Sharing

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}
```

Passing immutable objects as messages between threads.

Concurrent modification ceases to be a problem!

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;
        });
    }
```

Avoiding side effects

… embrace referential transparency

with pure functions

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()));
```

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??
```

Reasoning about your code

```
int Sum(List<int> values) {

    int sum = 0;
    for(int i = 0;i < arr.Length; i++){
        sum += values[i];
        values[i] = 0;

    }
}
```

Reasoning about your code

```
int Sum(List<int> values) { … }

List<int> values = new List<int>{ 1,2,3,4,5 };

int result1 = Sum(values); // 15

int result2 = Sum(values); // ??

int Sum(IEnumerable<int> values) { … }
```

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);
}
```