National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Implementation of MBrace for large-scale cloud computing

DIPLOMA PROJECT

KONSTANTINOS RONTOGIANNIS

Supervisor : Nikolaos S. Papaspyrou
Assoc. Prof. N.T.U.A

Athens, July 2015

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Implementation of MBrace for large-scale cloud computing

DIPLOMA PROJECT

KONSTANTINOS RONTOGIANNIS

Supervisor : Nikolaos S. Papaspyrou
Assoc. Prof. N.T.U.A

Approved by the examining committee on the July 7, 2015.

Nikolaos Papaspyrou Assoc. Prof. N.T.U.A	Kostas Kontogiannis Assoc. Prof. N.T.U.A	Don Syme Principal Researcher Microsoft Research, Cambridge

Athens, July 2015

Konstantinos Rontogiannis
Electrical and Computer Engineer

Copyright © Konstantinos Rontogiannis, 2015.
All rights reserved.

This work is copyright and may not be reproduced, stored nor distributed in whole or in part for commercial purposes. Permission is hereby granted to reproduce, store and distribute this work for non-propfit, educational and research purposes, provided that the source is acknowledged and the present copyright message is retained. Enquiries regarding use for profit should be directed to the author.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Technical University of Athens.

Abstract

The purpose of this diploma dissertation is on one hand the description of `MBrace`; a programming model for performing large scale distributed computations, and on the other hand the implementation of MBrace on the Microsoft Azure cloud computing platform.

In the era of big data and cloud computing, the need for efficient and effective orchestration of distributed computations becomes a necessity. Cloud computing providers make it easy for someone to get access to computational resources needed.

Programming large scale distributed systems is a remarkably difficult task, that requires the management and orchestration of multiple concurrent processes, software and hardware failures, scalability and elasticity.

In this diploma dissertation we describe `MBrace`; a programming model for large scale cloud computing. Based on the F# programming language and the .NET framework stack, `MBrace` provides a declarative, expressive and rich model for creating and composing distributed computations. This pattern is also known as cloud workflows, or cloud monad. Finally, we have implemented this model on top of the Microsoft Azure platform, providing `MBrace.Azure`: a distributed execution runtime for cloud computations.

Key words

distributed programming, cloud computing, big data, F#, cloud monad, microsoft azure

Acknowledgements

I would like to thank my professor, Nikolaos Papaspyrou, for everything he taught me throughout my studies. Furthermore, I would like to thank Nikos Palladinos and Eirik Tsarpalis for our collaboration during the development of `MBrace`. I would also like to thank my family and friends for their support during my studies.

Konstantinos Rontogiannis,

Athens, July 7, 2015

This thesis is also available as Technical Report CSD-SW-TR-3-15, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, July 2015.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Objectives

This thesis aims at providing a description of the `MBrace` programming model and framework for performing large scale distributed computations. It also aims at implementing a runtime that fully supports all of the `MBrace` abstractions, build on top of the Microsoft Azure cloud computing platform.

1.2 Motivation

We live in the era of big data and cloud computing. A huge amount of data is stored and collected on data centers and a massive amount of information can be extracted from web services, social networks, etc. In addition to that, it's easier than ever to get access to large computational power, at low cost and even for a small amount of time, offered by a plethora of cloud computing providers like Microsoft Azure, Amazon, etc.

Programming these large scale systems can be a notoriously difficult task, as programmers need to take into account not only the algorithm sophistication or complexity, but also the management and orchestration of concurrent processes, scalability, elasticity and fault tolerance (in either software or hardware failures). That effort also includes the overhead of having to pick up each cloud vendor's specific software stack.

Several programming models have be proposed and developed over the years, with *Map-Reduce* [Dean08] being the most popular paradigm. One of the most successful Map-Reduce frameworks is the open source *Hadoop* [hado] implementation as well as *Apache Spark* [spar], an open source engine for large scale data processing. Other implementations include the *Akka* [akka] distributed actor framework, and programming models like *CloudHaskell* [Epst11] and *HdpH* [Maie12].

This thesis presents `MBrace`, a novel programming model for scalable cloud programming and data scripting. `MBrace` [mbra] is an open source effort with the goal to provide a simple, elegant and declarative approach to describing distributed computations in a cloud vendor agnostic manner. `MBrace` is written in the F#/.NET programming language and integrates with the F# Interactive (REPL – Read-Eval-Print-Loop) providing a powerful experience, which enables fast code prototyping in a cloud scale, job monitoring and code deployment. `MBrace` is capable of distributing arbitrary code and offers native access to the rich collection of libraries offered with the underlying .NET framework.

At the moment `MBrace` consists of two basic projects:

`MBrace.Core` which contains the `MBrace` programming model, basic data structures and algorithms,

as well as the foundations for implementing `MBrace` runtimes.

`MBrace.Azure` which contains a `MBrace` runtime implementation, built on top of the Microsoft Azure platform.

1.3 Outline of the thesis

The rest of the thesis is organized as follows : In Chapter 2 we describe the F# async computation expressions and the transition from asynchronous parallelism to distributed parallelism using `MBrace` and the cloud workflow. Chapters 3 to 5 include an overview of the `MBrace` core programming model, while chapters 6 and 7 provide an overview of the workflow execution and runtime implementation.

Chapter 3 provides and overview of the primitive operators for distribution and concurrency. In Chapter 4 we present the basic storage and data related primitives and data structures. Chapter 5 concludes the high level programming model overview, describing the fault tolerance model used by `MBrace`.

The remaining chapters (6 and 7) provide a more in-depth analysis on how a cloud workflow is actually executed as well as details on the `MBrace.Azure` runtime implementation and features. Finally, in Chapter 8, we reach our conclusion and present some future work and goals for `MBrace` and `MBrace.Azure`.

Chapter 2

From async to cloud

Although `MBrace` supports any .NET language like `C#, VB.NET, etc`, the primary focus for `MBrace` is F#. F# [fsha] is a functional-first programming language originated from `ML`. The essential pillar of the `MBrace` programming model is the cloud workflow, which is implemented using a language construct called *Computation Expressions*. This chapter includes an introduction to the F# computation expressions, needed to understand how the cloud workflows work, as well as the asynchronous computation expressions from which the cloud workflows originate.

2.1 Computation expressions and monads

In this section, we give an overview of the Computation Expressions [comp] [Syme05] construct. Computation expressions introduce a new syntactic category `cexpr`, used to indicate the syntax of computation expressions:

⟨*expr*⟩ := ⟨*cbuilder*⟩ **'{'** ⟨*cexpr*⟩ **'}'**

The grammar for F# computation expressions is shown in figure 2.1.

⟨*cexpr*⟩ := **'do!'** ⟨*expr*⟩
| **'let!'** ⟨*pat*⟩ **'='** ⟨*expr*⟩ **'in'** ⟨*cexpr*⟩
| **'let'** ⟨*pat*⟩ **'='** ⟨*expr*⟩ **'in'** ⟨*cexpr*⟩
| **'return!'** ⟨*expr*⟩
| **'return'** ⟨*expr*⟩
| ⟨*cexpr*⟩ **';'** ⟨*cexpr*⟩
| **'if'** ⟨*expr*⟩ **'then'** ⟨*cexpr*⟩ **'else'** ⟨*cexpr*⟩
| **'match'** ⟨*expr*⟩ **'with'** ⟨*pat*⟩ **'->'** ⟨*cexpr*⟩
| **'while'** ⟨*expr*⟩ **'do'** ⟨*cexpr*⟩
| **'for'** ⟨*pat*⟩ **'in'** ⟨*expr*⟩ **'do'** ⟨*cexpr*⟩
| **'use'** ⟨*val*⟩ **'='** ⟨*expr*⟩ **'in'** ⟨*cexpr*⟩
| **'use!'** ⟨*val*⟩ **'='** ⟨*expr*⟩ **'in'** ⟨*cexpr*⟩
| **'try'** ⟨*cexpr*⟩ **'with'** ⟨*pat*⟩ **'->'** ⟨*cexpr*⟩
| **'try'** ⟨*cexpr*⟩ **'finally'** ⟨*expr*⟩
| ⟨*expr*⟩

Figure 2.1: Computation expression grammar

Computation expressions are just syntactic sugar to method calls. Below are some of the transforma-

tions done by the F# compiler when using computation expressions. [1] Defining a custom computation expression (like the cloud workflow) is straightforward by creating a `cbuilder` class and defining certain special methods on the class like `Bind, Return, Zero`, etc.

Expression	Desugaring
$[\![$ **let** binding **in** $cexpr]\!]$	= **let** binding **in** $[\![cexpr]\!]$
$[\![$ **let!** pattern = expr **in** $cexpr]\!]$	= *cbuilder*.Bind(expr,(**fun** pattern -> $[\![cexpr]\!]$))
$[\![$ **do!** expr **in** $cexpr]\!]$	= *cbuilder*.Bind(expr,(**fun** () -> $[\![cexpr]\!]$))
$[\![$ **return** expr$]\!]$	= *cbuilder*.Return(expr)
$[\![$ **return!** expr$]\!]$	= *cbuilder*.ReturnFrom(expr)
$[\![$ **use** pattern = expr **in** $cexpr]\!]$	= *cbuilder*.Using(expr,(**fun** pattern -> $[\![cexpr]\!]$))
$[\![$ **use!** pattern = expr **in** $cexpr]\!]$	= *cbuilder*.Bind(expr, (**fun** value -> *cbuilder*.Using(value, (**fun** value -> $[\![cexpr]\!]$))))
$[\![$ **if** expr **then** $cexpr_0$ **else** $cexpr_1]\!]$	= **if** expr **then** $[\![cexpr_0]\!]$ **else** $[\![cexpr_1]\!]$
$[\![$ **while** expr **do** $cexpr]\!]$	= *cbuilder*.While((**fun** () -> expr), *cbuilder*.Delay($[\![cexpr]\!]$))
$[\![$ **try** $cexpr$ **with** pattern$_i$ -> expr$_i]\!]$	= *cbuilder*.TryWith(*cbuilder*.Delay($[\![cexpr]\!]$), (**fun** value -> **match** value **with** | pattern$_i$ -> expr$_i$ | exn -> **reraise** exn)))
$[\![$ **try** $cexpr$ **finally** expr$]\!]$	= *cbuilder*.TryFinally(*cbuilder*.Delay($[\![cexpr]\!]$), (**fun** () -> expr))
$[\![cexpr_0$; $cexpr_1]\!]$	= *cbuilder*.Combine($[\![cexpr_0]\!]$; $[\![cexpr_1]\!]$)
$[\![$other-expr ; $cexpr]\!]$	= expr ; $[\![cexpr]\!]$
$[\![$other-expr $]\!]$	= expr ; *cbuilder*.Zero()

Table 2.1: Computation expression translation rules

Finally, computation expressions are used to provide convenient syntax for *monads*, since the `Bind` and `Return` methods correspond to the `bind` and `return` operators of the monad laws [mona].

```
Method      Signature

Bind      : M<'T> * ('T → M<'U>) → M<'U>
Return    : 'T → M<'T>
```

2.2 Async computation expressions

One of the most common computation expressions used in F# are the Asynchronous Workflows [Syme11]. F# async workflows are part of the language's core library and provide an easy way to author tasks that seem sequential, but are given an asynchronous semantic interpretation, without using callbacks.

[1] The complete list of transformations can be found in the F# specification [Syme05] and MSDN F# Computation Expressions [comp].

```
let getLengthAsync (uri : Uri) : Async<int> =
    async {
        let client = new System.Net.WebClient()
        let! html = client.AsyncDownloadString(uri)
        return html.Length
    }
```

Example 2.1: F # async example

The workflow above, spawns a new task that downloads the content of a web page. This task may or may not execute in the initial thread. Once the download is complete the result is bound to variable `html` and the total length is returned, also as an async computation.

As one can see the `Async<T>` type, which represents an asynchronous computation, is the primitive type for F# async programming. All expressions of the form `async{ … }` are of type `Async<T>`. When executed, an async value will *eventually* produce a value of type `T`.

The F# core library also provides some basic functions for composing and executing async workflows.

```
    Async.Parallel              : Async<'T>[] → Async<'T[]>
    Async.RunSynchronously      : Async<'T> → 'T
    Async.StartImmediate        : Async<unit> → unit
    Async.Sleep                 : int → Async<unit>
    ...
```

Example 2.2: Some of the basic F# Core async functions

2.3 Cloud workflow

Inspired by the asynchronous workflows, `MBrace` provides cloud workflows, used to specify distributed computations that can run on a cluster of workers. Similarly to async, a cloud workflow has type `Cloud<T>` which represents a delayed computation that can be executed by some worker and return a value of type `T`.

The simplest cloud workflow would be the following :

```
let helloWorld : Cloud<int> =
    cloud {
        return 42
    }
```

Example 2.3: The `MBrace` hello world

Given a runtime implementation that provides a `Cloud<T>` → `T` function, this workflow can be evaluated :

```
let runtime = Runtime.GetHandle(azureConfiguration)
let result = runtime.Run helloWorld

val result : int = 42
```

The runtime implementation (in our case `MBrace.Azure`) is responsible for packaging and uploading the image of the cloud computation, scheduling a new job in the available worker pool and returning the result to our client.

By using computation expressions, `MBrace` provides a strong foundation for expressing different kinds of algorithms and paradigms such as Map-Reduce, streaming, iterative or incremental algorithms, etc.

2.4 Summary

To sum up, we have been introduced to the computation expressions concept and especially the F# async workflows. We also transitioned from the async workflows to cloud workflows. Chapters 6 and 7 provide a more in depth view on the semantics of the cloud workflow and details on the workflow execution on Microsoft Azure.

Part I

The `MBrace` core programming model

Chapter 3

Distribution combinators

In this chapter we offer a description of the basic distribution combinators instructed by the `MBrace` programming model. `MBrace` provides a vast amount of primitive operations for running cloud computations in parallel, as long running tasks or in a non-deterministic way.

3.1 Parallel

Probably the most essential `MBrace` operator is `Cloud.Parallel`. This combinator can be used to execute multiple cloud workflows, in a parallel fork-join pattern. Its type signature is

```
Cloud.Parallel : Cloud<'T> seq → Cloud<'T []>
```

This takes a sequence of cloud computations and returns a workflow that executes them, possibly in parallel, and returns an array of all results. Although exception handling will be covered later in this chapter, we can mention that the exception handling semantics are similar to the `Async.Parallel` ones; the first exceptions not handled by the child computations will trigger the exception to the overall computation and cancel the others. Likewise, the first cancellation results in overall cancellation.

You can see below an example of the `Cloud.Parallel` combinator.

```
1   let parallelWorkflow : Cloud<int []> =
2       cloud {
3           let compute i = cloud { return i * i }
4           let! results =
5               [1..10]
6               |> List.map compute
7               |> Cloud.Parallel
8           return results
9       }
10
11  // Evaluate workflow
12  runtime.Run(parallelWorkflow)
13
14  val it : int [] = [|1; 4; 9; 16; 25; 36; 49; 64; 81; 100|]
```

Example 3.1: A simple `Cloud.Parallel` computation

The root computation will spawn 10 jobs across the cluster. Once the results are aggregated the parent computation continues and returns the result. Note, that the way `Cloud.Parallel` jobs are going to

be executed (order, worker allocation, etc) is dependent to the underneath runtime implementation, in our case `MBrace.Azure`.

3.2 Choice

Another primitive combinator offered by the `MBrace.Core` is `Cloud.Choice`. This combinator enables non-deterministic computations.

```
Cloud.Choice : Cloud<'T option> seq → Cloud<'T option>
```

This combines a collection of non-deterministic computations into one. The non-deterministic computation is encoded by the `option` type, where `Some` value declares success and `None` declares failure. This combinator executes its input in parallel and returns a result as soon as the first child declares success (`Some` value). If none of the child computations complete without success then `None` is returned. Triggering an exception on a child computation causes the overall computation to cancel and raise this exception.

You can see below an example of the `Cloud.Choice` combinator.

```
1  let findIndex (xs : 'T []) (item : 'T) : Cloud<int option> =
2      cloud {
3          return! xs |> Seq.mapi (fun index x → index, x)
4                     |> Seq.map (fun (i,x) →
5                          cloud {
6                              return if x = item then Some i else None
7                          })
8                     |> Cloud.Choice
9      }
10
11 // Evaluate workflow
12 runtime.Run(findIndex [|0..100|] 42)
13
14 val it : int option = Some 42
```

Example 3.2: Parallel search using `Cloud.Choice`

This example implements a naive distributed search. It creates multiple parallel jobs (one for each element of the input) that search for the index of the given element. When a job finds the element the overall computation is canceled and the index is returned.

3.3 CloudTask

The third distribution operator that we are going to mention is `Cloud.StartAsTask`. The `MBrace` programming models defines the notion of cloud tasks. An `ICloudTask<T>` is a unit of code that can be executed independently as a child computation.

The `Cloud.StartAsTask` combinator returns a cloud workflow that queries the result.

```
Cloud.StartAsTask    : Cloud<'T> → Cloud<ICloudTask<'T>>
Cloud.AwaitCloudTask : ICloudTask<'T> → Cloud<'T> [1]
Cloud.StartChild     : Cloud<'T> → Cloud<Cloud<'T>>
```

As can someone notice, `ICloudTask<T>` is the `MBrace`/cloud analogous to the `.NET Task Parallel Library Task<T>` type.

```
1  let workflow : Cloud<int * TimeSpan> =
2      cloud {
3          // Spawn a CloudTask
4          let! ctask =
5              Cloud.StartAsTask(
6                  cloud {
7                      do! Cloud.Sleep 20000
8                      return 42
9                  })
10
11         let watch = Stopwatch.StartNew()
12         // Block until ctask completes
13         let! value = ctask.AwaitResult()
14         watch.Stop()
15
16         return value, watch.Elapsed
17     }
18
19 // Evaluate workflow
20 runtime.Run workflow
21
22 val it : int * TimeSpan = (42, 00:00:22.4323827)
```

Example 3.3: Spawning a `CloudTask`

The example above demonstrates the usage of the `Cloud.StartAsTask` combinator. A cloud task is spawned, while the parent computation awaits for the result.

3.3.1 Cancellation

At this point it is natural to give an overview of how cancellation is materialized in `MBrace`. Both the `Cloud.Parallel` and `Cloud.Choice` combinators involve the notion of cancellation in their semantics. `MBrace` provides a cancellation mechanism, similar to .NET's, using cancellation tokens.

An `ICancellationToken` enables cooperative cancellation between jobs or `ICloudTask<T>` objects. You create a cancellation token by instantiating a `ICancellationTokenSource` object, which manages cancellation tokens retrieved from its `CancellationTokenSource.Token` property. You then pass the cancellation token to any jobs, or tasks that should receive notice of cancellation.

In contrast with .NET's cancellation tokens, those abstractions have distributed semantics: A computation in a worker (or even a client), can request cancellations for jobs, tasks, or even thread-pool

[1] The actual return type is `Local<T>`,subtype of `Cloud<T>`. For the sake of simplicity we use `Cloud<T>` as a return type until Section 3.6.

work items running in another machine.

Below there is an example of wrapping the result of a computation to an option type, depending on whether the task completed in the given timespan.

```
let tryExecute (workflow : Cloud<'T>) (interval : int) : Cloud<'T option> =
    cloud {
        let! cts = Cloud.CreateCancellationTokenSource()
        let! ctask = Cloud.StartAsTask(workflow,
                        cancellationToken = cts.Token)
        do! Cloud.Sleep interval
        cts.Cancel()

        try
            let! value = ctask.AwaitResult()
            return Some value
        with :? OperationCanceledException →
            return None
    }

let test =
    cloud {
        do! Cloud.Sleep 10000
        return 42
    }

runtime.Run(tryExecute test 20000)
val it : int option = Some 42

runtime.Run(tryExecute test 5000)
val it : int option = None
```

Example 3.4: Using cancellation tokens in `MBrace`

3.4 Exceptions

An important feature of cloud workflows is exception handling. `MBrace` permits the usage of exceptions just like any other `.NET`/F# value. Exceptions can be raised, handled and transferred between machines.

```
let workflow =
    cloud {
        try
            let! ctask = Cloud.StartAsTask(cloud { return 1 / 0 })
            return! ctask.AwaitResult()
        with ex →
            do! Cloud.Log ex.StackTrace
            return! Cloud.Raise(ex)
    }

```

```
11    runtime.Run(workflow)
12
13    System.DivideByZeroException: Attempted to divide by zero.
14       at FSI_0022.workflow@30-28.Invoke(Unit unitVar)
15       at MBrace.Core.Builders.Delay@287-1.Invoke(ExecutionContext ctx, Unit t)
16           in C:\workspace\MBrace.Core\src\MBrace.Core\Continuation\Builders.
                fs:line 287
```

Example 3.5: Exception handling in `MBrace`

3.5 WorkerRef

Up to this point we have seen job execution without having to specify where each job should execute. Although a runtime implementation might provide some scheduling optimizations, there are often cases where a job should be executed in a specific worker.

`MBrace` introduces the concept of worker references. An `IWorkerRef` denotes a reference to a unique worker node in the cluster topology. All of the distribution combinators mentioned, provide overloads so that the given computations will execute in the targeted workers.

```
Cloud.Parallel    : seq<Cloud<'T> * IWorkerRef> → Cloud<'T []>
Cloud.Choice      : seq<Cloud<'T option> * IWorkerRef> → Cloud<'T option>
Cloud.StartAsTask : Cloud<'T> * IWorkerRef → Cloud<ICloudTask<'T>>
```

The `IWorkerRef` abstraction provides information about the workers identity, number of processor cores, etc. In addition the `MBrace.Azure` implementation includes performance monitoring information like CPU/Memory/Network utilization, the number of currently executing jobs, heartbeat information, etc.

This feature:

- Enables user-level scheduler implementations where the programmer can use the information mentioned above in order to load balance the jobs.
- Enables efficient execution on heterogeneous clusters: e.g. cloud blocks can execute only in machines with graphics units and use the underlying hardware, specific services that run in particular machines, as an optimization for data processing libraries in order to allocate jobs in machines where the data is stored/cached.
- Is used by the library implementations that `MBrace.Core` includes, in order to balance the computations according to worker processing capacities.

3.6 Constraining execution

So far we have seen that `Cloud<T>` is the premium type used to denote execution. Although `MBrace` is a framework for distributed execution, sometimes it's quite useful to constraint the execution of a code block only in-memory, suppressing any distribution effects. For this reason `MBrace` introduces a `Local<T>` type, subtype of `Cloud<T>`, that describes a computation with in-memory only execution.

The `Local` module redefines many `Cloud` combinators to use local parallelism semantics, as well as a `local` computation expression builder.

```
Local.Parallel : (computations : seq<Local<'T>>) → Local<'T []>
Local.Choice   : (computations : seq<Local<'T option>>) → Local<'T option>
Cloud.AsLocal  : (computation : Cloud<'T>) → Local<'T>
Cloud.OfAsync  : (computation : Async<'T>) → Local<'T>
...
```

Local workflows have two important use cases:

- Managing computation granularity. Especially in recursive/divide and conquer algorithms, where spawning more jobs would actually slow down the computation, local workflows are used to force in-memory execution.
- Safely using and composing cloud workflows that handle non-serializable object or in general resources that do not make sense in a distributed context: e.g. filestreams, network connections, etc.

Consider the following example. We are using a `WebClient` (non-serializable, disposable) object in order to download in parallel some webpages. The `downloadParallel` function is declared as a local workflow and uses `Cloud.Parallel` in order to execute those operations in parallel. Note that the code does not type check because the type `Cloud<T>` returned by `Parallel` does not compose with the type `Local<'T>`, expected by the `local` builder:

```
Error: Cloud<string * string> is not compatible with type Local<'a>.
```

Function `downloadParallel'` fixed this error by using the `Local.Parallel` combinator.

```
1  let downloadParallel uri1 uri2 : Local<string * string> =
2      local {
3          use webClient = new System.Net.WebClient()
4          let download uri =
5              webClient.AsyncDownloadString(uri)
6              |> Cloud.OfAsync
7
8          // Error: Cloud<string * string> is not compatible with
9          // type Local<'a>
10         let! values = Cloud.Parallel(download uri1, download uri2)
11         return values
12     }
13
14 let downloadParallel' uri1 uri2 : Local<string * string> =
15     local {
16         use webClient = new System.Net.WebClient()
17         let download uri =
18             webClient.AsyncDownloadString(uri)
19             |> Cloud.OfAsync
20
21         let! values = Local.Parallel(download uri1, download uri2)
22         return values
```

```
23          }
```

Example 3.6: Local execution in `MBrace`

3.7 Pitfalls

As we can see the `MBrace` programming model offers a very rich, expressive and easy way to declare computations that run in a distributed environment. As a result, it's fairly easy to write code that seems valid, but doesn't make sense in the context of cloud workflows, or may behave in a different way than expected.

3.7.1 Non serializable types

This is the category that is easier to recognize and address. In case of cloud blocks returning or capturing non-serializable objects both the `MBrace.Core` and `MBrace.Azure` return a proper exception as a result.

```
1   let workflow =
2       cloud {
3           return IO.File.OpenWrite "foo.txt"
4       }
5
6   runtime.Run(workflow)
7
8   Failed to execute job 'd85985885a8d4eb3802bbf2b46dfe87f' ->
9       Nessos.FsPickler.NonSerializableTypeException:
10          Type 'MBrace.Azure.Runtime.Primitives.Result[System.IO.FileStream]'
11          contains non-serializable field of type 'System.IO.FileStream'.
```

Example 3.7: Non-serializable filesystem stream

3.7.2 Mutation

Traditional in-memory value mutation doesn't make sense when being in a distributed environment like `MBrace`. Consider the following example that mutates in parallel the contents of a ref cell. In an asynchronous workflow this code would be a race condition and the result would be non-deterministic. In `MBrace` this workflow will always return 0. Each one of the child jobs will receive a copy of the original value, and alter that copy. The original value remains 0.

```
1   let workflow : Cloud<int> =
2       cloud {
3           let cell = ref 0
4           do! [1..10]
5               |> List.map (fun _ -> cloud { cell := !cell + 1 })
6               |> Cloud.Parallel
7               |> Cloud.Ignore
8           return !cell
```

```
9         }
10
11    runtime.Run workflow
12    val it : int = 0
```

Example 3.8: Value mutation

3.7.3 Large objects

Although `MBrace` can handle and serialize efficiently any value, regardless of size, it's quite common for user code to create unnecessary serialization of big values and thus slow down the computation.

Consider the following example.

```
1     let workflow =
2         cloud {
3             let largeObj = [|1..10000000|]
4             return! [1..10]
5                     |> List.map (fun _ → cloud { return largeObj.Length })
6                     |> Cloud.Parallel
7         }
```

Example 3.9: Capturing large objects

All ten computations created capture the `largeObj` value. This results in ten serializations of the environment instead of one that the programmer might expect. Although it might be possible to optimize such cases, it is very helpful for the programmer to understand how job distribution actually happens and address this type of issues. In Section 4.2.1 we provide a user-space workaround for this common problem.

3.8 Streaming operations with MBrace.Flow

Although not part of the Core library, at this point it's worth mentioning `MBrace.Flow`. `MBrace.Flow` is a framework, implemented on top of the `MBrace.Core` primitives that provides and easy way to describe data parallel streaming computations, with caching and in-memory computation capabilities. `MBrace.Flow` shares many similarities with the Apache Spark engine [spar].

```
1     open MBrace.Flow
2
3     let workflow : Cloud<(string * int64) []> =
4         CloudFlow.OfCloudFileByLine "foobar.txt"
5         |> CloudFlow.collect(fun line → line.Split(' '))
6         |> CloudFlow.filter(fun word → word.Length > 3)
7         |> CloudFlow.countBy id
8         |> CloudFlow.toArray
```

Example 3.10: A CloudFlow pipeline

`MBrace.Flow` handles transparently input partitioning, parallelism, workflow scheduling to workers where the data is cached, etc, all in a declarative way.

Chapter 4

Storage abstractions and data structures

4.1 Overview

So far we have seen `MBrace` as a framework for big computations. In addition to the many combinators that enable distribution and parallelism `MBrace` also offers a plethora of abstractions for managing data in a global, machine-wide scope. The following sections include an overview of the basic storage primitives.

4.2 CloudFile

`CloudFile` constitutes the premium storage primitive offered by `MBrace`. A `CloudFile` is nothing more than a string, the path of a file, in a probably distributed but with uniform access filesystem. In other words a `CloudFile` is a reference to untyped binary data, either created using the `MBrace` APIs or existing in the above mentioned storage.

`MBrace` provides several methods for creating, accessing, deleting, etc, CloudFiles. Below there is a typical example of creating a `CloudFile` and reading it's contents back in the client.

```
1   let writeLines path (lines : string seq) : Cloud<CloudFile> =
2       cloud {
3           let writer (stream : Stream) =
4               async {
5                   use sr = new IO.StreamWriter(stream)
6                   for line in lines do
7                       sr.WriteLine(line)
8               }
9           return! CloudFile.Create(path, writer)
10      }
11
12  // Evaluate workflow
13  let file = runtime.Run(writeLines "temp/foobar.txt" ["Hello"; "world"])
14  val file : CloudFile = temp/foobar.txt
15
16  // Evaluate in local client rather than the cluster
17  runtime.RunLocally(CloudFile.ReadLines(file.Path))
18  val it : seq<string> = seq ["Hello"; "world"]
```

Example 4.1: Implementation of a `CloudFile.WriteLines` function

4.2.1 CloudValue

Even though cloud files are a powerful primitive, their untyped nature makes them inconvenient to use when dealing with typed data. For this reason `MBrace` includes a primitive called `CloudValue<T>`. A `CloudValue` is a lightweight reference (similarly to `CloudFile`) to typed and serialized data. CloudValues resemble *references* found in the *ML* family of languages. CloudValues:

- Are immutable references that can be either initialized or dereferenced.
- Provide in-memory caching capabilities
- Support any F#/`.NET` serializable value as a payload.
- Are extremely useful in performance optimizations.
- Fit nicely with iterative algorithms (like *kmeans*).

At this point, instead of giving a typical example for the `CloudValue` API or defining a distributed data structure, let's return to example 3.9, and update it to use a CloudValue for the large shared object.

```
1   let workflow =
2       cloud {
3           let! cvalue = CloudValue.New [|1..10000000|]
4           return! [1..10]
5                   |> List.map (fun _ → cloud {
6                       // Dereference and compute
7                       let! value = cvalue.Value
8                       return value.Length })
9                   |> Cloud.Parallel
10      }
```

Example 4.2: Using a `CloudValue`

This way the large array is only serialized once, instead of multiple times, and a lightweight reference is send to child computations.

4.2.2 CloudSequence

While CloudValues are useful for storing small or medium sized data, using them for large collections of objects may create unnecessary memory pressure. A `CloudSequence<T>` is similar to a `CloudValue`, but offers on-demand fetch for a collection of values of type `T`, thus a smaller memory footprint.

```
1   let workflow =
2       cloud {
3           let! cseq = CloudSequence.New [|1..10000000|]
4           return! [1..10]
5                   |> List.map (fun _ → cloud {
6                       // Dereference and fetch on demand
7                       let! values = cseq.ToEnumerable()
8                       return Seq.length values })
```

```
                    |> Cloud.Parallel
    }
```

Example 4.3: Using a `CloudSequence`

4.3 CloudChannel

In `MBrace` a `CloudChannel<T>` is a reference to some queue implementation. Essentially `CloudChannel` is a pair of an `ISendPort<'T>` and an `IReceivePort<'T>`, that can be used to send messages between cloud workflows. The semantics of a `CloudChannel` depend on the actual implementation with regards to multiple senders/receivers, message ordering, etc.

Below there is an example of an agent that 'ticks' every second.

```
let tick : Cloud<IReceivePort<int>>  =
    cloud {
        // Both send and receive port are serializable in MBrace.Azure
        let! send, recv = CloudChannel.New<int>()

        let rec aux count = cloud {
            do! send.Send(count)
            do! Cloud.Sleep 1000
            return! aux (count + 1)
        }

        // Spawn
        do! Cloud.StartAsTask(aux 0)
            |> Cloud.Ignore

        return recv
    }


let recv = runtime.Run(tick)
val recv : IReceivePort<int>

runtime.RunLocally(recv.Receive())
val it : int = 0
runtime.RunLocally(recv.Receive())
val it : int = 1
runtime.RunLocally(recv.Receive())
val it : int = 2
```

Example 4.4: Defining a 'tick' agent using CloudChannels

4.4 CloudAtom

An `ICloudAtom<T>` represents distributed updatable value reference. A cloud atom can be mutated (either atomically or not) and is very useful for implementing global counters, synchronisation prim-

itives like locks, semaphores, etc.

In the example below the result is guaranteed to be 42.

```
let workflow =
    cloud {
        let! atom = CloudAtom.New(0)
        do! [1..42]
            |> List.map (fun _ →
                // Atomic updates
                CloudAtom.Update(atom, fun x → x + 1))
            |> Cloud.Parallel
            |> Cloud.Ignore
        return! CloudAtom.Read(atom)
    }

runtime.Run(workflow)
val it : int = 42
```

Example 4.5: Using a `CloudAtom`

4.5 CloudDictionary

An `ICloudDictionary<'T>` is a distributed key-value store abstraction. It offers similar functionality to `.NET ConcurrentDictionary` data structure. Below there is an example of defining a workflow that memoizes function values.

```
let memoizer (dictionary : ICloudDictionary<'T>) (f : 'U → 'T) (x : 'U) =
    cloud {
        let key = x.ToString()
        let! value = dictionary.TryFind(key)
        match value with
        | Some v → return v
        | None →
            let value = f x
            do! dictionary.Add(key, value)
            return value
    }
```

Example 4.6: Implementing function memoization

4.6 CloudDisposable

All of the primitives mentioned above use space in the storage back-end used by the runtime. `MBrace` provides a way of deallocating this space by implementing the `ICloudDisposable` interface.

```
type ICloudDisposable =
    abstract Dispose : unit → Local<unit>
```

Example 4.7: ICloudDisposable

This concept is similar to the `IDisposable` interface available in `.NET`. The `MBrace` cloud builder offers functionality for using scoped disposable resources, similar to the `use` and `using` keywords in F# and C#.

In the example below the `CloudValue` will be disposed at the end of it's scope regardless of exceptions, etc.

```
1    let workflow =
2        cloud {
3            use! cvalue = CloudValue.New 42
4
5            failwith "boom!"
6
7            // cvalue is out of scope, dispose
8            return ()
9        }
```

Example 4.8: ICloudDisposable

4.7 Summary

Bellow you can see a summary of all the data abstractions offered by `MBrace`/`MBrace.Azure` and their basic properties.

Type	Underlying	Gives	Caching
`CloudFile`	File store	Stream/byte[]/lines/text	no
`CloudAtom<T>`	Table store	T	no
`CloudDictionary<T>`	Table store	string -> T	no
`CloudChannel<T>`	Service Bus	T values	no
`CloudValue<T>`	CloudFile + deserializer for T	T	on by default
`CloudSequence<T>`	CloudFile + deserializer for seq<T>	seq<T>	off by default

Table 4.1: Summary of Data abstractions in `MBrace`

Chapter 5

Fault tolerance

This chapter covers the fault tolerance model instructed by `MBrace` and implemented in `MBrace.Azure`. By using the term *fault tolerance* we are referring to either hardware failure or unrecoverable runtime failure that causes user workflows to not complete with success. Note that user level exceptions are not considered as abnormal behavior (i.e. faults) and are handled the way described in section 3.4.

`MBrace` offers a way of defining a fault recovery policy, that after a number of attempts will either fail or retry after a given delay.

```
type FaultPolicy = { Policy : int → exn → TimeSpan option }

Cloud.FaultPolicy     : Local<FaultPolicy>
Cloud.WithFaultPolicy : FaultPolicy → Cloud<'T> → Cloud<'T>
Cloud.StartAsTask     : (Cloud<'T> * FaultPolicy) → Cloud<ICloudTask<'T>>
```

Example 5.1: Fault tolerance in `MBrace`

The high-level strategy within a workflow is:

- Define your retry strategy and policy.
- Try the operation that could result in a transient fault.
- If transient fault occurs, invoke the retry policy.
- If all retries fail, catch a final exception.

An interesting observation is that, although a fault policy can be set for any workflow, fault tolerance is closely related to the underlying runtime implementation. This means that a fault policy is a property of the runtime's scheduling unit i.e. a job (see Chapters 6 and 7 for more information). As a result setting a fault policy only takes effect when the given workflow introduces distribution.

Part II

The `MBrace` implementation

Chapter 6

The Cloud Monad and Continuation Passing Style

6.1 Overview

The two remaining chapters provide an in-depth view of the cloud workflow execution. This chapter's goals are :

- Make the reader familiar to the `Cloud<T>` internals and representation.
- Describes the cloud monad/builder interaction with the underlying runtime.
- Define what is considered as a *job* in `MBrace.Azure`.
- Give an overview of how a core operator like `Cloud.Parallel` is implemented.

Although giving such overview seems like exposing the user to implementation details, it is fundamental for having a correct understanding of the workflow semantics and execution.

6.2 Continuation Passing Style

In chapter 2 we presented the computation expression builder translation rules (table 2.1). By performing a CPS transformation on cloud expressions we can view a `Cloud<T>` as continuation based distributed computation. At this point it is useful to define a `Continuation<T>` type that represents the `Cloud<T>` continuations:

```
type Continuation<'T> = {
    SCont : 'T → unit
    ECont : exn → unit
    CCont : OperationCancelledException → unit
}
```

Example 6.1: `Cloud<T>` continuations

A cloud expression becomes a function that accepts three continuations: a success continuation `scont`, an exception continuation `econt` and a cancellation continuation `ccont`. When evaluating a `Continuation<T>` *exactly one* of the continuations should be called.

But how can someone implement a primitive combinator like `Cloud.Parallel` using those three continuations? It is implied that the three continuations have access to some sort of runtime han-

dle, responsible for enqueueing jobs, access to storage, logging, etc. The next section describes the functionality provided by this handle.

6.3 Runtime resources

Creating meaningful combinators presupposes accessing runtime resources. This section gives a short description of those resources.

6.3.1 Distribution Provider

Probably the most important interface needed for a `MBrace` runtime implementation is the `IDistributionProvider`. This type should be the starting and core point of any runtime and provides the needed functionality for implementing combinators described in chapters 3 and 5.

IDistributionProvider	
ScheduleParallel	*enqueue given workflows as a Parallel*
ScheduleChoice	*enqueue given workflows as a Choice*
ScheduleStartAsTask	*enqueue given workflow as a CloudTask*
FaultPolicy	*get current FaultPolicy*
WithFaultPolicy	*switch FaultPolicy*
WithForcedLocalParallelismSetting	*switch between* cloud *and* local
…	*other functionality like logging, creating cancellation tokens, executing* local *workflows, etc*

Table 6.1: The core of a runtime implementation

6.3.2 Other resources

Other resources provided by a runtime to a computation include, but are not limited to:

- Storage access mechanisms. In particular implementations of `CloudFile`, `CloudAtom`, `CloudChannel` and `CloudDictionary` factories.
- In-memory/local filesystem caches, etc.
- Access to a job execution pool.
- Primitives used for distributed coordination, like counters, etc.
- Worker specific data.
- Custom, user-specified resources.

At this point let's define the aggregation of the above mentioned resources as an `ExecutionContext`.

```
type ExecutionContext = {
    Token               : ICloudCancellationToken
    Runtime             : IDistributionProvider
    JobQueue            : JobQueue
    CloudFileProvider   : CloudFileProvider
     …
    CounterFactory      : CounterFactory
     …
    CustomResources     : seq<obj>
}
```

Example 6.2: Workflow's execution context

6.4 The Cloud Monad

One could argue that the `Continuation<T>` type is adequate for representing a cloud workflow. Implementing any non-trivial operator shows the need to use the `ExecutionContext` in order to interact with the runtime. This context could be threaded from the current computation to it's continuations.

While this is fine in a local/in-memory environment, the fact that `Continuation<T>` is serialized and send across machines, as well as the fact that the `ExecutionContext` holds non-serializable and worker specific resources makes this representation insufficient.

6.4.1 Combining with the Reader Monad

The Reader Monad offers a great way of passing state and reading values from a shared environment. By using the `ExecutionContext` as the machine-local environment and changing the `Continuation<T>` type so that the context is passed as an argument we get:

```
type Continuation<'T> = {
    SCont : ExecutionContext → 'T  → unit
    ECont : ExecutionContext → exn → unit
    CCont : ExecutionContext → OperationCancelledException → unit
}
```

Example 6.3: The updated `Cloud<T>` continuations

Now `Continuation<T>` becomes an entity that can be serialized and distributed across machines. Executing `Continuation<T>` in a worker is done by supplying the non-distributable local `ExecutionContext` to the workflow's continuations.

As a conclusion: the *cloud monad* offered by `MBrace` forms a continuation over reader monad.

6.4.2 Cloud.FromContinuations

Given that knowledge, at this point we introduce two of the most important core combinators:

```
Cloud.FromContinuations<'T> :
    (ExecutionContext → Continuation<'T> → unit) → Cloud<'T>
Cloud.GetResource<'T>       : unit → 'T
```

Example 6.4: cloud/cc

Using the functions above, one can either access the monad's resources, or construct a new computation from a triple of continuations.

Finally, from the `Cloud.FromContinuations` type signature we can derive the `Cloud<T>` definition:

```
    type Cloud<'T> = ExecutionContext → Continuation<'T> → unit
```

Example 6.5: The `Cloud<T>` signature

6.4.3 Implementing a primitive combinator

Implementing a combinator like `Cloud.Parallel` in `MBrace.Azure` is a straightforward procedure. After defining a set of useful distributed coordination primitives like counters, result aggregators, as well as a job execution pool, all that remains to be done is provide an implementation that constructs continuations with the expected semantics (described in 3.1) and enqueues them in the runtime job pool.

```
1  let scheduleParallel(computations : seq<#Cloud<'T>>) : Cloud<'T []> =
2    Cloud.FromContinuations(
3      fun (ctx : ExecutionContext) (cont : Continuation<'T []>) →
4          let n = computations.Length
5
6          // aggregate and call cont.SCont when all complete
7          let sconts = mkSuccessConts n cont
8
9          // call cont.ECont on first exception
10         let econts = mkErrorConts n cont
11
12         // call cont.Ccont on first cancellation
13         let cconts = mkCancelConts n cont
14
15         let (jobs : Job []) = pack(sconts, econts, cconts) // serialize, etc
16         ctx.JobQueue.Enqueue(jobs)
17   )
18
19 let parallel (computations : seq<#Cloud<'T>>) : Cloud<'T []> = cloud {
20   let! runtime = Cloud.GetResource<IDistributionProvider> ()
21   let workflow = runtime.ScheduleParallel computations
22   return workflow
23 }
```

Example 6.6: `ScheduleParallel` and `Cloud.Parallel`

At this point it becomes clear what is considered as a *job* in `MBrace.Azure`. *A `Job` is the minimum a unit of work that can be executed by the distributed runtime.* Usually a job carries its parent contin-

uations and as a result code that is not logically/visually part of the child computation is executed in child jobs.

Chapter 7

The `MBrace.Azure` runtime

7.1 Overview

For a runtime implementation we chose the Microsoft Azure platform. With `MBrace` written in F#/.NET and rich Azure tooling available in IDEs like Visual Studio, Azure was an obvious choice that a large amount of the .NET community is familiar with. Azure Service Bus [serv] and Storage [stor] are the backbone of `MBrace.Azure`.

Service Bus is a cloud-based messaging system for connecting distributed applications. Service Bus is used by our runtime in order to implement a job execution pool. A Service Bus Queue is used to hold the set of pending computations.

Azure Storage provides the capability to store large amounts of binary data in Azure Blobs, as well as structured NoSQL based records with Azure Tables. Blob storage is used by our runtime as a backing store for large serialized entities (continuations, etc), and Table storage is used to implement runtime primitives like counters, runtime state like worker records and logs.

Both Service Bus and Storage are designed with scalability and fault tolerance and elasticity in mind, and our runtime incorporates those features.

In order to create `MBrace.Azure` clusters an Azure account is required. After creating an account the only thing required is acquiring a service bus and storage connection string. Both of these connection strings are the only configuration needed for `MBrace.Azure` and can be used to create clients or worker nodes.

```
1  let azureConfiguration =
2   { Configuration.Default with
3      StorageConnectionString = myStorageConnectionString
4      ServiceBusConnectionString = myServiceBusConnectionString }
5  let runtime = Runtime.GetHandle(azureConfiguration)
```

Example 7.1: Creating a client

7.2 `MBrace.Azure` features

The following sections provide an overview of the features implemented by `MBrace.Azure`. Our runtime provides both worker implementations and a client that can be used for interaction with the

runtime. `MBrace.Azure` fully implements the `MBrace.Core` requirements and offers a rich set of APIs on top of that.

7.2.1 Runtime Service

`MBrace.Azure` offers a runtime `Service` that can be used to spawn a worker node. The worker node is an entity responsible for dequeueing and executing jobs from the global job pool. Note that, in contrast with other implementations like Apache Hadoop, in our runtime all runtime node are workers i.e. there are no nodes dedicated on scheduling, holding runtime state, etc. All of the runtime state is stored in Azure Service Bus and Storage, that guarantee its persistence.

`MBrace.Azure` provides two samples using the runtime service:

- An Azure Worker Role template used to create multiple virtual machines in Microsoft Azure, that runs our service. Note that we are referring to Platform As A Service (PaaS) VMs, where the user doesn't need to maintain the infrastructure.
- A standalone executable that can be used for local testing i.e. without the need to deploy virtual machines.

7.2.2 Worker monitoring

Our implementation also offers monitoring of the worker nodes. Periodically each node sends information about cpu, memory, network utilization, the number of jobs currently executing, as well as heartbeats. Nodes fail to give heartbeats in a given interval are considered by the runtime dead and are not used by the runtime until they are recycled.

```
1  runtime.GetWorkers() |> Seq.head
2      val it : WorkerRef =
3        MBrace.Azure.WorkerRef {Id = "3d1dd41c72814386a569de644dbb7d11";
4                                Hostname = "RDOO156D3A3723";
5                                Status = Running;
6                                Version = "0.6.9.0";
7                                ActiveJobs = 4;
8                                MaxJobCount = 16;
9                                CPU = 37.97297287;
10                               Memory = 2756.0;
11                               TotalMemory = 6111.0;
12                               NetworkUp = 3.726053238;
13                               NetworkDown = 8.563692093;
14                               HeartbeatTime = 21/06/2015 15:21:32 +00:00;
15                               InitializationTime = 21/06/2015 15:20:55 +00:00;
16                               ProcessId = 4012;
17                               ProcessName = "mbrace.azureworker";
18                               ProcessorCount = 8;
19                               AutoUpdate = false;
20                               Fault = null;
21                               ConfigurationId = MBrace.Azure.ConfigurationId; }
```

7.2.3 Cloud process

`MBrace.Azure` allows the execution of cloud workflows as 'processes'. A process is a wrapper of the workflow's root job and contains useful information about the execution time, current status, number of jobs spawned, etc.

```
1   let ps = runtime.CreateProcess(cloud { return 42 })
2   val it : Process<int> =
3     MBrace.Azure.Client.Process'1[System.Int32]
4       {Id = "03656ac893674b629ec77c36106384cc";
5        Status = Running;
6        ActiveJobs = 1;
7        CompletedJobs = 0;
8        TotalJobs = 1;
9        Completed = false;
10       ExecutionTime = 00:00:00.9161719;
11       InitializationTime = 21/06/2015 18:46:13 +03:00;
12       CancellationTokenSource = dfbb549d5461480c96f66fa305e408a2;
13       Name = "";
14       Type = System.Int32;}
15
16  ps.AwaitResult() // 42
```

7.2.4 Job tracking

In addiction, the runtime provides detailed information regarding job execution in the cluster like: execution times, number of retries, serialized job size, current status, assigned worker, etc. Below there is a view of the job tree for some process with nested parallelism.

```
01aa02a...355 Root 5.06 KiB Completed 00:00:10.1961841
├──55eee12...025 Parallel (0,4) 9.55 KiB Completed 00:00:06.1362068
├──3738d33...0de Parallel (1,4) 9.55 KiB Completed 00:00:08.6851059
|   ├──0fb1bd3...121 Parallel (0,1) 13.26 KiB Completed 00:00:10.1691793
|   ├──b3f32e0...794 Parallel (1,1) 13.26 KiB Completed 00:00:06.1571005
├──e5708e1...31b Parallel (2,4) 9.56 KiB Completed 00:00:09.6100630
|   ├──f56160b...c3c Parallel (0,2) 13.26 KiB Completed 00:00:07.1040602
|   ├──ca6d137...137 Parallel (1,2) 13.26 KiB Completed 00:00:10.6542046
|   ├──5df170d...686 Parallel (2,2) 13.26 KiB Completed 00:00:05.6180358
├──5159dae...ea0 Parallel (3,4) 9.56 KiB Completed 00:00:09.5770645
|   ├──f7b1871...9e1 Parallel (0,3) 13.26 KiB Completed 00:00:07.1542197
|   ├──128c691...c8c Parallel (1,3) 13.26 KiB Completed 00:00:22.0414365
|   ├──5cab915...ef5 Parallel (2,3) 13.26 KiB Completed 00:00:06.8551199
|   ├──a4cba07...dc7 Parallel (3,3) 13.26 KiB Completed 00:00:07.1122064
├──a562d87...503 Parallel (4,4) 9.57 KiB Completed 00:00:05.9571426
|   ├──072c78e...4c8 Parallel (0,4) 13.26 KiB Completed 00:00:05.9480752
|   ├──a78944c...38b Parallel (1,4) 13.26 KiB Completed 00:00:06.4501444
|   ├──8c93cd9...196 Parallel (2,4) 13.26 KiB Completed 00:00:06.9761758
|   ├──4a57daf...6ba Parallel (3,4) 13.26 KiB Completed 00:00:07.5290255
|   ├──8ccfa0d...59e Parallel (4,4) 13.26 KiB Completed 00:00:10.4790779
```

7.2.5 Fault tolerance

The runtime implements the required fault tolerance semantics for job execution (see chapter 5) using the lease/lock mechanism offered by Service Bus. Consider the following code that crashes one of the worker nodes. Running the code with a `NoRetry` fault policy causes a `FaultException` as a result.

```
let ps =
    runtime.CreateProcess(
        [1..4]
        |> List.map (fun i → cloud { return i <> 2 || exit 1 })
        |> Cloud.Parallel
    , faultPolicy = FaultPolicy.NoRetry)

ps.AwaitResult()

MBrace.Core.FaultException:
 Fault exception when running job 'O71b96a23264423890cb6577184c6fa6', faultCount
    '1'
at Cloud.Parallel(seq<Cloud<Boolean>> computations)
— End of stack trace from previous location where exception was thrown —
Stopped due to error
```

7.2.6 Storage primitives

`MBrace.Azure` provides implementations for all of the storage primitives of chapter 4:

- *CloudFiles* are implemented as Azure Blobs.
- *CloudAtoms, CloudDictionaries* are implemented on top of the Azure Table storage.
- *CloudChannels* are implemented using Service Bus queues.

7.2.7 Assembly distribution

At this point it is worth mentioning that the runtime supports transmission of either static or dynamic assemblies (e.g. assemblies created in the F# REPL) between clients and workers.`MBrace.Azure` makes use of the *Vagabond* library which enables such scenarios. This way our implementation delivers a smooth programming experience with rapid prototyping and easy code deployment from a REPL.

Part III

Final Remarks

Chapter 8

Conclusion and future work

So, to conclude, we presented the design and programming model of `MBrace`. MBrace offers a rich and elegant approach for describing distributed cloud computations and implementing algorithms without the need to manage the 'low level 'details that appear in a distributed environment.

In addition, we implement a runtime for `MBrace` on top of the Microsoft Azure stack. `MBrace.Azure` aims to support all concepts described in `MBrace.Core`. Although `MBrace.Azure` is still a young project, we believe that our current implementation is on a stable enough level, with many useful features, that someone can download and start using. Also having a active community that uses `MBrace` would help us find bugs and also help us prioritize on runtime features.

Our first goal is to keep up with the core library and stabilize our APIs and features once `MBrace.Core` reaches its *1.0* release milestone.

With `MBrace.Azure` being the first distributed runtime for `MBrace`, we acquired a lot of knowledge on how a real distributed runtime is implemented. The goal is to provide any abstractions extracted from our runtime as a generic library, part of `MBrace.Core`, for creating runtime systems for `MBrace`.

Both `MBrace` and `MBrace.Azure` are open source projects. After accomplishing the above mentioned goal of providing common foundations for runtime implementations, we intent to write a technical runtime overview that will make it easier for someone to contribute to our project.

Finally, we hope that our implementation helps `MBrace` become a solid choice for anyone who wants to perform distributed computations in the .NET ecosystem.

Bibliography

[akka] "Akka framework", URL `http://akka.io/`.

[comp] "Computation Expressions (F#)", URL `https://msdn.microsoft.com/en-us/library/dd233182.aspx`. MSDN.

[Dean08] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[Epst11] Jeff Epstein, Andrew P Black and Simon Peyton-Jones, "Towards Haskell in the cloud", in *ACM SIGPLAN Notices*, vol. 46, pp. 118–129, ACM, 2011.

[fsha] "The F# Software Foundation", URL `http://fsharp.org/`.

[hado] "Apache Hadoop", URL `https://hadoop.apache.org/`.

[Maie12] Patrick Maier and Phil Trinder, "Implementing a high-level distributed-memory parallel Haskell in Haskell", in *Implementation and Application of Functional Languages*, pp. 35–50, Springer, 2012.

[mbra] "MBrace big data framework and related projects", URL `https://github.com/mbraceproject/`.

[mona] "Monad laws", URL `https://wiki.haskell.org/Monad_laws`.

[serv] "Azure Service Bus", URL `http://azure.microsoft.com/en-us/services/service-bus/`.

[spar] "Apache Spark", URL `https://spark.apache.org/`.

[stor] "Azure Storage", URL `http://azure.microsoft.com/en-us/services/storage/`.

[Syme05] Don Syme, Luke Hoban, Tao Liu, Dmitry Lomov, James Margetson, Brian McNamara, Joe Pamer, Penny Orwick, Daniel Quirk, Chris Smith et al., "The F# 3.0 Language Specification", URL `http://fsharp.org/specs/language-spec/3.0/FSharpSpec-3.0-final.pdf`, 2005.

[Syme11] Don Syme, Tomas Petricek and Dmitry Lomov, "The F# asynchronous programming model", in *Practical Aspects of Declarative Languages*, pp. 175–189, Springer, 2011.