AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

FACULTY OF COMPUTER SCIENCE, ELECTRONICS AND TELECOMMUNICATIONS

DEPARTMENT OF COMPUTER SCIENCE

Master of Science Thesis

Aplikacje obliczeniowe na platformie Windows Azure
Computational applications on Windows Azure platform

Author: *Piotr Wiewiura*
Degree programme: *Computer Science*
Supervisor: *Maciej Malawski, PhD*

Kraków, 2014

Oświadczam, świadomy(-a) odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

I would like to express my gratitude to my supervisor, Dr. Maciej Malawski for his support, guidance and patience throughout the course of this thesis.

Abstract

This thesis evaluates suitability of Microsoft Azure as a platform for execution of computational applications. Microsoft Azure is a relatively new public cloud service which has potentially much to offer to the scientific community. Motivation for this work comes from the fact that available research publications on this subject are limited in numbers and often outdated.

Microsoft Azure's capabilities were tested in three scenarios: dynamic horizontal scaling, distributed execution of a CPU-intensive application – POV-Ray raytracer, and distributed execution of a bioinformatics application – ExonVisualizer. The second goal of the thesis was to create a Distributed Task Library (DTL) for the purposes of these tests, due to lack of free, simple solution for distributed execution of dynamically defined tasks in .NET.

After introducing the above goals, the thesis describes capabilities of Microsoft Azure. Next, it moves to presentation of the state of the art. The following two chapters are dedicated to DTL. The first one describes the whole design and architecture of the library with the help of detailed diagrams. The second one focuses on the implementation of DTL – used technologies and the most interesting challenges encountered and solved. Subequently, the next chapter describes all the conducted tests, including performance evaluation, together with their detailed procedures and results. Finally, the conclusions of the thesis are presented along with suggested future work.

In conclusion, the thesis shows that while dynamic horizontal scaling is quite slow, Microsoft Azure is a worthy platform for computational applications, offering, in conjunction with DTL, an easy way to speed up at least CPU-intensive, embarrassingly parallel problems. In the end, each application needs an individual assessment and may require a specific approach to fully exploit the capabilities of Microsoft Azure.

Contents

1. Introduction

This chapter presents the background of the thesis and explains its title by introducing a definition of computational applications. Then, it defines goals of this thesis and requirements for its implementation part - Distributed Task Library.

1.1. Background

Cloud computing has gained a lot of attention over the last few years. There are many reasons for this phenomenon e.g. increasing availability of broadband Internet access, advances in mobile technology, need for providing seamless experience on all devices, and the term "cloud" itself which aggregates a vast number of technical solutions under one, simple, attractive name. This last fact creates a major confusion throughout the community simply because it is often impossible to understand the meaning of "cloud computing" without providing the specific context in which this term is used. Nevertheless, this thesis revolves around cloud computing so it is good to establish some definition as a reference point. It turns out that probably the most "official" definition is good enough for our purposes. It was published by National Institute of Standards and Technology in September 2011 [19].

> ”Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

This is a very broad definition, because it tries to cover a very broad field. It focuses on defining a type of computing resource provisioning model but stays clear of saying anything about the purpose which those resources will serve. That is why we refrained from using this term in the title of this thesis as we wanted to focus on "computing" in its strictest sense - processing data using numerical methods. We therefore decided to incorporate "computational applications" term into the thesis’s title to make it more specific.

The second part of the title refers to Microsoft Azure (formerly known as Windows Azure) which, launched on February 1, 2010, is one of the newest public cloud platforms on the market. Because of its young age, there aren’t many publications evaluating its computational capabilities. Those which exist are usually at least partially outdated because there were many new features introduced into Azure since its conception.

1.2. Goals

The main goal of this thesis is to **evaluate suitability of Microsoft Azure as a computational applications execution platform**. Cloud platforms are designed to be versatile in order to meet the needs of the largest customer base possible. This means that one needs to carefully analyze what the given cloud has to offer and at what quality. It may excel in one areas and fail in others. We decided to check Azure's computational capabilities as such study may be helpful to scientists and other people who seek high-performing cloud platform. Our starting point was our earlier publication [17] which showcases component-based approach to cloud applications. We wanted to set up an architecturally comparable software environment and start from performing similar tests and then, extend them. From among many aspects which could be analyzed we selected three use cases to evaluate:

- **Dynamic horizontal scaling** - Microsoft Azure is not free and user pays for every minute of running node. It creates a need for dynamic adjustment of number of started instances according to the current and expected workload. The goal was to test how quickly Azure responds for scale out requests.
- **CPU-intensive application - POV-Ray** - Many scientific methods rely on pure CPU power, so it seemed natural to check how Azure copes with such problems. Moreover, we wanted to check how difficult it is to take and existing, proprietary application and wrap it in such way that it can be executed in a distributed manner on Microsoft Azure.
- **Real-world bioinformatics application - ExonVisualizer** - The goal was to take a scientific application with available source code and make it work on Microsoft Azure. The application [27] was provided by Dr. Monika Piwowar from the Department of Bioinformatics and Telemedicine of Jagiellonian University. In this way we could check how much impact on existing code has addition of cloud support. Additionally, ExonVisualizer is not as CPU-bound as POV-Ray so it provided a different perspective on distributed execution.

1.2.1. Distributed Task Library

During the analysis of tools which could facilitate our tests, we found out that none of the freely available libraries or applications fits our needs. We were forced to add another goal to our list - **create a a lightweight library which will allow to execute arbitrary task with arbitrary data on Microsoft Azure.** It turned out that achieving this goal took the most of the time spent on the thesis.

Distributed Task Library, as we named it, had to meet the following requirements:

- **Distributed execution** - DTL should be able to execute a given task on multiple machines simultaneously, distribute data required for this task and collect results.
- **Install-and-forget worker service** - Machines used for execution (workers) should have only a small, once-installed worker service/daemon responsible for task execution.

- **Dynamic execution** - Workers shouldn't require any prior knowledge about the executed task, i.e. task definition and all the necessary data and binary dependencies should be distributed by DTL itself.
- **Simplicity** - DTL must be very easy to use. Executing a task in a distributed manner with default behavior should take no more than a few lines of code.
- **Dynamic horizontal scaling** - The library should be able to add, remove, stop and start workers in runtime.
- **Portability and execution platform independence** - DTL should work with Microsoft Azure but it must be designed in such way that switching to another platform is possible and simple.
- **Written in C#** - Not exactly a requirement imposed by the thesis subject itself but included because of author's expertise in this language which should make the implementation part easier.
- **Support for Linux (Mono)** - DTL must work in Linux environment (which implies Mono support) because many computational applications are available only for Linux.
- **Worker service auto-update** - Updating workers should not require any manual actions as workers may be installed on many machines, making update process very tedious.

1.3. Summary

Summing up, this thesis takes up a task of checking if one of the newest public clouds - Microsoft Azure is a good platform for executing computational applications. It involves creation of Distributed Task Library, a simple solution for distributed execution of dynamically defined tasks in .NET.

2. Introduction to Microsoft Azure

The goal of this chapter is to present Microsoft Azure public cloud, especially the areas related to thesis's goals. Chapter starts with general overview of Azure and then proceeds to description of available execution models and reasoning behind choosing the best one for our purposes. Afterwards, Microsoft Azure Storage Services are described. The last section is dedicated to Azure's APIs which are exposed to developers.

2.1. Overview

Microsoft Azure (formerly known as Windows Azure) is a public cloud platform created and hosted by Microsoft. It was released on February 1, 2010 and from the beginning had to compete on relatively young market dominated by Amazon Web Services. Over the years Azure matured and as of May, 2014 it became one of the leaders in cloud computing, still far from reaching Amazon but already outrunning other competition [13] as it can be seen in Fig. 2.1. One of the contributing factors to such progress was Microsoft's policy of relative technological openness. The company didn't limit available operating system to Windows family but provided machines with various Linux distributions, often with preinstalled popular software. Moreover, developers are not limited to Microsoft-supported languages like C#, Visual Basic or C++ but also other popular ones e.g. Java, PHP, Python or Ruby. Managing Azure account is possible via web application called Microsoft Azure Management Portal (see fig. or through various APIs (as described in section 2.4). Microsoft Azure offers many features (see Fig. 2.2) but in this thesis we will describe only those who seemed to be useful in attaining our goal.

2.2. Execution Models

Execution models are ways in which execution of an applications is handled by Azure. They differ in ease of use, versatility, freedom of control and intended type of applications. There are four execution models available in Microsoft Azure:

- Cloud Services
- Virtual Machines
- Web Sites

Figure 2.1: Magic Quadrant for Cloud Infrastructure as a Service showing the increasing importance of Microsoft Azure. [13]

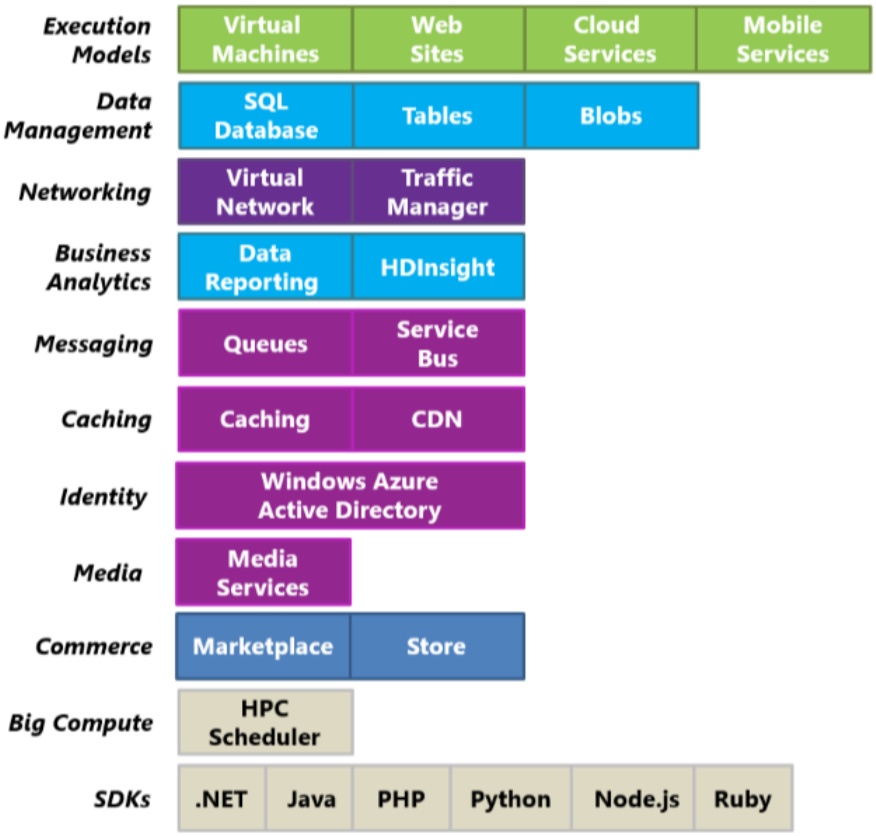

Figure 2.2: Microsoft Azure features. [4]

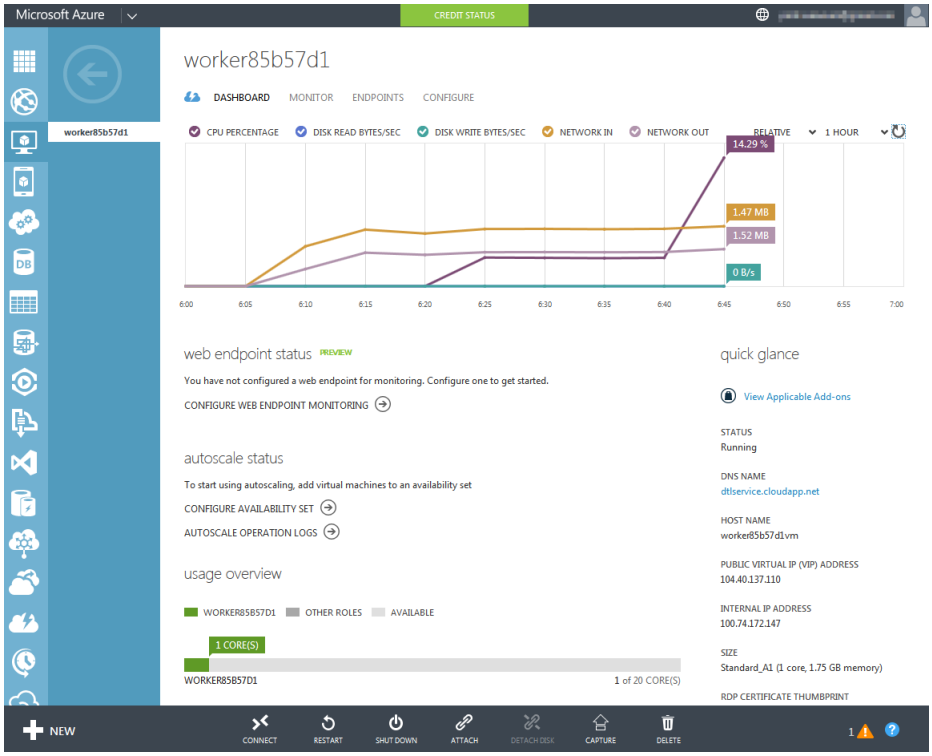

Figure 2.3: Microsoft Azure Management Portal

- Mobile Services

The last two - Web Sites and Mobile Services were too limiting for our purposes, we didn't want to be limited to testing web or mobile applications as we sought for a more versatile test bed. Hence, we considered using Cloud Services or Virtual Machines so we evaluated usefulness of both of them.

2.2.1. Cloud Services

Cloud service provides a relatively easy way to host an application in a cloud. It's a PaaS solution so it comes with a convenient features but at the same time imposes a number of requirements. Many tasks are automated or simplified, e.g. automatic installation of system patches and rolling out new versions of images. Furthermore, you don't have to worry about system configuration - the only thing you have to provide is package with your application and configuration. Configuration must, among others, contain list of "roles" used by the applications. Simply put, they are preconfigured virtual machines and exist in two varieties:

Web Role - designed to host a web applications, has IIS installed

Worker Role - best suited for backend services

Role instances are created within a "deployment". There are two deployment environments: staging and production. First is useful for application tests before moving it to the production one.
Cloud Services are convenient but not fully suitable for our needs for several reasons:

COMPUTE INSTANCE NAME	VIRTUAL CORES	RAM	OS DISK	TEMP DISK
Extra Small (A0)	Shared	768 MB	127 GB	20 GB
Small (A1)	1	1.75 GB	127 GB	40 GB
Medium (A2)	2	3.5 GB	127 GB	60 GB
Large (A3)	4	7 GB	127 GB	120 GB
Extra Large (A4)	8	14 GB	127 GB	240 GB

Table 2.1: Microsoft Azure Virtual Machines instances

- Application must be packaged in an Azure compatible cloud service package (.cspkg) and be accompanied by cloud service configuration file (*.cscfg) which makes deployment more complicated
- It is best suited for multi-tier applications
- Underlying operating system is always the latest Windows Server - so it can't be changed to Linux if needed
- Automatic system management requires that deployed applications are stateless, all data must be kept in storage service or other network location

2.2.2. Virtual Machines

Virtual machines are an IaaS solution available in Microsoft Azure. They are a different concept than Cloud Services, but in fact they are just another type of "roles" (PersistentVMRole). As with other roles, each virtual machine must be placed within deployment which in turn is placed within cloud service. Virtual machines provide the highest degree of flexibility but have very limited application hosting facilities. They are persistent, virtual hosts, which can be created, started, stopped and deleted on demand. User can create virtual machines by choosing one of multiple OS images available or upload their own image. Then, size of virtual machine instance must be selected. In April 2014, Microsoft introduced new, tiered pricing model for Virtual Machines [5] but while working on this thesis we used previous model, which effectively was the current Standard Compute Tier - General Purpose. List of its supported instance sizes is presented in table 2.1. The more powerful the machine, the more expensive it is. User is charged for every minute of running instance. After selecting instance size, administrator user name and password must be entered. Then, it is necessary to select a unique hostname within .cloudapp.net domain. Machine will be visible under this hostname from the Internet. Next step involves choosing Region, Affinity Group or Virtual Network where the VM will be placed.

Region is a physical location of the Microsoft's datacenter which will be hosting the created virtual machine. This allows for choosing a place closest to the user to minimize latency. It does not, however, say anything about placement within the datacenter.

Affinity Group is a way to group services by physical location within datacenter. Azure will try to place them as close as possible.

Virtual Network provides capability to extend on-premises network into Microsoft Azure

There is also an option to select new or existing storage account where the disks will be put (see section 2.3 for details on storage services). Finally, user can select availability set and define virtual machine's endpoints. An availability set is a group of virtual machines that are deployed across fault domains and update domains. This makes sure that single points of failure are removed. Endpoints define which ports should be open after VM is created.

Autoscale

Virtual Machines have an ability to automatically scale out based on some configurable conditions. Conditions are based on CPU usage or length of the chosen queue. The mechanism tries to keep within the set thresholds by starting or stopping VMs. It does not have ability to create new virtual machines so user must clone them beforehand and keep in stopped state until autoscale decides to start them. This means that machines are started quickly but user is billed for storage space occupied by their disk images. This may result in significant charges as each such image has 127 GB. Unfortunately, autoscale feature was introduced too late and eventually we hadn't enough time to test it.

2.3. Storage Services

Microsoft Azure provides a way to store data, which can be shared among deployed applications, virtual machines or accessed from outside which usually is consumed or produced by deployed applications. User is charged for amount of stored data, selected replication options, number of read and write requests and data transfer outside of storage service's region. There are 4 so-called storage services:

- Blob Storage
- Queue Storage
- Table Storage
- File Storage

We made use of Blob Storage and Queue Storage only. They are briefly described in the following sections.

2.3.1. Blob Storage

Blob Storage is designed to store large amounts of unstructured binary data. This makes it the most versatile storage service available in Windows Azure. Data is stored in the form of blobs. There are two types of blobs:

Block blob is optimized for streaming and storing cloud objects. It's good for uploading large amounts of data efficiently. Block blob is built from blocks which can be uploaded simultaneously and offer a kind of transaction support, which allows to discard a blob if not all blocks were uploaded successfully.

Page blob is optimized for random access. It's a collection of 512-byte pages. Page blobs are usually used for storing hard drive images.

Blobs are organized in containers. Unlike filesystem directories, containers cannot be nested (with exception of the root container).

2.3.2. Queue Storage

Queue Storage provides messaging solution, usually used for communication between decoupled application components. It gives the user a reliable Get\Put\Peek interface. Message inserted into queue can contain up to 64 KB of binary or text data.

2.4. API

Virtually all Azure management operations can be done through web-based Microsoft Azure Management Portal but it requires a user to click through all the menus and windows. Microsoft provides a few ways to automate the Azure management through several APIs:

- Native SDKs for .NET, Java, Python, Ruby, Node.js, PHP and mobile platforms (iOS, Android, Windows Phone)
- PowerShell cmdlets
- Cross-Platform Command-Line Interface
- REST API

In order to achieve our goals, we had to decide which API to use. It turned out to be an easy decision, because only REST service met our requirements. Other APIs had inherent properties which in one way or another forced us to not use them. In the following sections we explain the reasons behind it.

2.4.1. Native SDKs

Native SDKs are a solution which provides the most features and is easiest to use from the supported programming languages. Our goals included testing .NET applications, more specifically C# ones, so we evaluated Azure SDK for .NET. It's probably the best supported of the SDKs as Microsoft tries to popularize its own technologies. It's also quite up-to-date, new version being released every 3 to 6 months. Nevertheless, there are situation when brand new Azure feature is not yet supported by the

offical SDK and users must wait some time for the release. It seemed as the best of option of integrating Azure with C# applications, but we couldn't use it. One of our goals was providing support for Linux through Mono platform but Azure SDK for .NET doesn't support it. The problem is that the SDK relies on assemblies which are not available in Mono.

2.4.2. PowerShell and Cross-Platform Command-Line Interface

There are two ways to manage Azure from command line - PowerShell cmdlets and Cross-Platform Command-Line Interface (xplat-cli). They both are great way to automate Microsoft Azure management using scripts or by calling them from external applications. Cmdlets integrate with PowerShell environment seamlessly and can be undoubtedly very useful to people familiar with PowerShell. Unfortunately weren't able to use them for the same reason as native SDKs - they don't work in Linux. xplat-cli, on the other hand, is cross-platform by design. This made it potential candidate for our tests. In the end we decided to not use xplat-cli because it required separate installation and was less powerful than REST API. Additionally, both solutions could be potentially used only for scaling tests. Building a distributed task execution platform based on them wouldn't be possible (or at least would be more complicated) because they lack Queue Storage support.

2.4.3. REST API

The most universal of Microsoft Azure APIs is REST API. It has a few features which made it our API of choice:

Programming language and platform independence

Nowadays most of the programming languages support or have 3rd party libraries which add support for consuming REST services. C# has a few Microsoft-supported ways to do it. Most notably, there is a fairly new solution - Web API which greatly simplifies creating and using REST services. Additionally, REST can be used on Linux and any other platform where the aforementioned languages work. Mono supports Web API as well.

No external tools are required

REST support is either built-in into programming languages or added via libraries. In case of C# it's built-in into .NET Framework 4.5.

Features coverage

REST API covers virtually all Azure's features.

Up to date with all Azure changes

REST API is the first one to support all new features and changes introduced to Azure. It also supports versioning which means that user can specify which version of API he wants to use so in case of some breaking changes the application won't be affected.

Despite its many advantages, REST API has one, big drawback - it's complicated. One must essentially create a library in his language of choice which will build and wrap all the necessary requests (often complex, especially in terms of authentication), handle errors and returns results. Things are further complicated by asynchronous nature of many operations. There are some efforts throughout community to create such libraries but they often cover only a part of Azure features and usually become out of date very quickly. That's why we decided to create our own wrapper based on some samples we found and official documentation. It took a lot of time to implement but eventually gave us all the necessary facilities required for performing our tests.

3. State of the art

This chapter provides insight into current state of the art. It's divided into to two sections. First section presents a few publications which tackle the subjects of performance of Microsoft Azure and its suitability for computational applications. It also briefly lists the most interesting findings of each of them. Second section describes two software solutions - Aneka and Windows Azure HPC Scheduler - which are somewhat similar to Distributed Task Library but failed to meet our requirements.

3.1. Related publications

There are more and more publications on Azure as its gaining popularity recently, however there isn't as many of them as e.g. publications on Amazon cloud. We found a few which are related to the subject of this thesis. We present their findings concerning areas of our interest.

3.1.1. Early Observations on the Performance of Windows Azure

The first publication is "Early Observations on the Performance of Windows Azure" by Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey from Department of Computer Science University of Virginia [11]. It's interesting because it's one of the first (if not first) attempt to check performance of Microsoft Azure (formerly called Windows Azure). Published on June 21, 2010 it focuses on Azure's performance in the following areas:

- Blob Storage bandwidth
- Table Storage CRUD operations
- Queue Storage Add/Peek/Receive operations
- Dynamic horizontal scalability
- TCP communication
- SQL Azure Database

They got a few results interesting from point of view of this thesis and our planned tests. First of all, authors performed virtual machine instantiation time tests using Worker and Web roles. They noted

that these times may be too long for certain applications. It took 10 minutes, on average, to instantiate first Small Instance virtual machine and each next from the same request was ready in about 80 seconds. Deleting the entire deployment took about 6 seconds.

Another interesting aspect they tested was Blob Storage bandwidth. For 1 GB blob and single client they achieved slightly above 13 MB/s download and 6 MB/s upload speed.

Overall, they reported a fairly good performance of Azure, only weak points being aforementioned long instantiation times and disappointing SQL Azure Database performance.

3.1.2. A Performance Study on the VM Startup Time in the Cloud

Another interesting article is "A Performance Study on the VM Startup Time in the Cloud" published on June 24, 2012 by Ming Mao and Marty Humphrey from from Department of Computer Science University of Virginia [18]. After 2 years after they previous publication they decided to compare virtual machines instantiation time using multiple clouds, including Azure. They compared Amazon EC2, Windows Azure and Rackspace but of course Azure's results were most interesting for us. Authors performed tests using Worker, Web and VM roles in South Central US region. They observed a 200-second improvement in startup times compared to the previous study - Worker role started in 406.2 s, Web role in 374.8 and VM role in 356.6 s on average. Other interesting observations from the article

- Within each cloud provider, the VM startup time of both Linux and Windows machines are independent of time of the day.
- VM startup time increases linearly as the image size increases.
- VM startup time does not show significant differences across different data center locations.
- The VM release time is not affected by the OS image size, instance type or data center location.

3.1.3. Science in the Cloud: Lessons from Three Years of Research Projects on Microsoft Azure

One of the more comprehensive and up-to-date studies of past and ongoing scientific projects making use of Microsoft Azure is "Science in the Cloud: Lessons from Three Years of Research Projects on Microsoft Azure" by Dennis Gannon, Dan Fay, Daron Green, Wenming Ye and Kenji Takeda from Microsoft Research published on June 23, 2014. They observed that after initial problems, Microsoft Azure has been successfully used in solving such problems like watershed modeling, metagenomics based on Blast application, analyzing fMRI scans, GIS processing and many others. Here are the most interesting highlights from their work:

- Network performance is a bottleneck, especially for MPI-intensive distributed applications.

- Virtual machines deployment takes much longer than many scientific programmers expect, so if applications needs dynamic scaling then it's preferable that overall execution time is long, so startup delays are less noticeable.
- Uploading large amounts of data is problematic as lots of scientific applications need huge datasets to work with. This problem has been addressed by Microsoft Azure Import/Export service which achieved general availability on May 12, 2014 [22]. It allows for sending physical hard drives to Microsoft which will copy them to Azure Blob Storage.
- Microsoft Azure Virtual Machines are a great way to share images with complete suites of high quality scientific software within the community. Microsoft Open Technologies opened a catalog of such VM images called VM Depot [29]

On the whole, we recommend this article to everyone who considers using Microsoft Azure for scientific applications.

3.2. Solutions similar to Distributed Task Library

We found no free solution which would meet all our requirements and help us achieve all the thesis's goals. However, we come across a few interesting ones which are worth mentioning here.

3.2.1. Aneka

Overview

Aneka [14] is a commercial PaaS solution and a framework which facilitates development of distributed applications in the cloud. It's developed by Manjrasoft Pty Ltd, a Australia-based based startup company originating from The Cloud Computing and Distributed Systems (CLOUDS) Laboratory at The University of Melbourne. Aneka is a complete Cloud Application Platform which provides a complete SDK for .NET Framework applications and also comes with rich set of tools. It can be used to develop and run applications on so-called Aneka Clouds which can span multiple physical or virtual infrastructures including public or private clouds.

Architecture

Aneka's architecture is presented in Fig. 3.1. The top level is occupied by API which allow applications interact with the whole Aneka's infrastructure and tools which help managing it. The main component on this level is Aneka Management Studio. It's a GUI application for managing infrastructure and Aneka clouds. Aneka clouds are comprised of software deamons called containers which must be installed on all machines in the cloud and connected through a network. Containers host services that customize the runtime environment available for applications. There are three types of services: fabric, foundation, and execution services. Fabric services are placed just on top of Platform Abstraction Layer. They perform

hardware profiling and dynamic resource provisioning. Foundation services are the core part of Aneka middleware. They provide all the resources needed by executed applications as well us supporting facilities like billing, reporting, licensing or accounting. Execution services manage scheduling and execution of applications using several execution models:

Task Model

Independent tasks are scheduled and executed. This model is the one most akin to Distributed Task Library, so we decided to describe in detail below.

Thread Model

Aneka provides thread scheduling and execution services for applications which are designed to utilize multiple threads.

MapReduce Model

Intended for data-intensive applications. Makes use of classic MapReduce programming model. Aneka implementation comes with scheduling, execution and storage services.

Parameter Sweep Model

Built on top of Task Model. Provides means for executing a template task using a collection of parameters.

The general overview, common to all execution models is presented in Fig. 3.2. User defines one or more applications using Manager machine, then they are organized in Work Units and sent to Scheduler which lies within Aneka Cloud Infrastructure. Scheduler distributes Work Units between Executor machines which perform execution. Afterwards, results are collected and sent back to Manager.

Task Model

Task Model is a one of Aneka execution models and is quite similar to our Distributed Task Library. The main assumption of this model is that applications are represented by tasks which are independent. This an important quality, because it allows scheduler to reorder them in arbitrarily. User is expected to create a set of tasks, submit them to Aneka, wait for results which at the end have to be assembled manually by him. This extended user's responsibility makes it easier to optimize the execution of tasks. Lack of workflow and other dependency mechanisms makes Task Model ideally suited for embarrassingly parallel problems.

From programmers point of view, Aneka seems to try to keep things simple. It requires implementation of ITask interface which contains a single, parameterless Execute method. Second requirement is that ITask implementation must be binary serializable as it is transmitted over the network. Developer must remember to store the executions results within the task instance and can read them when the task is sent back to the controller.

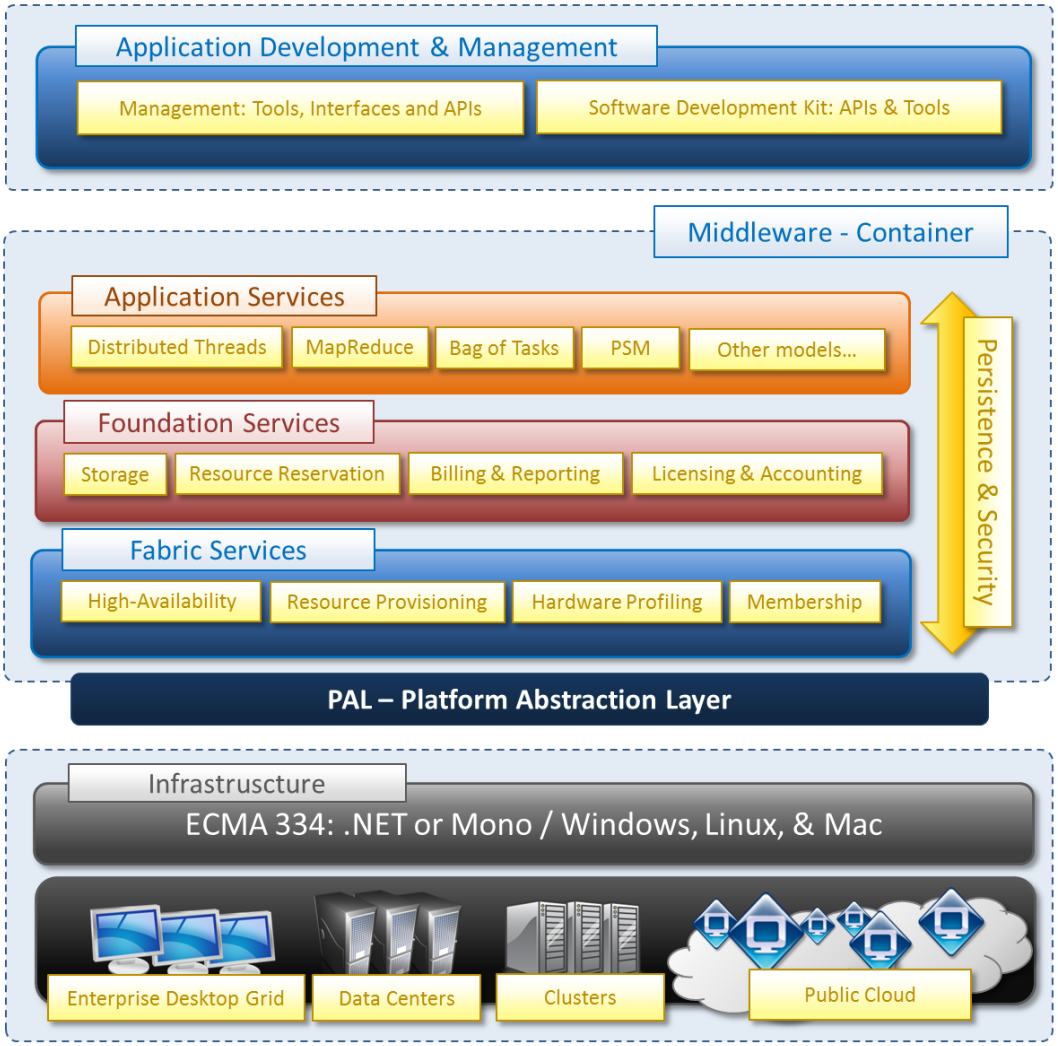

Figure 3.1: Aneka Architecture [14]

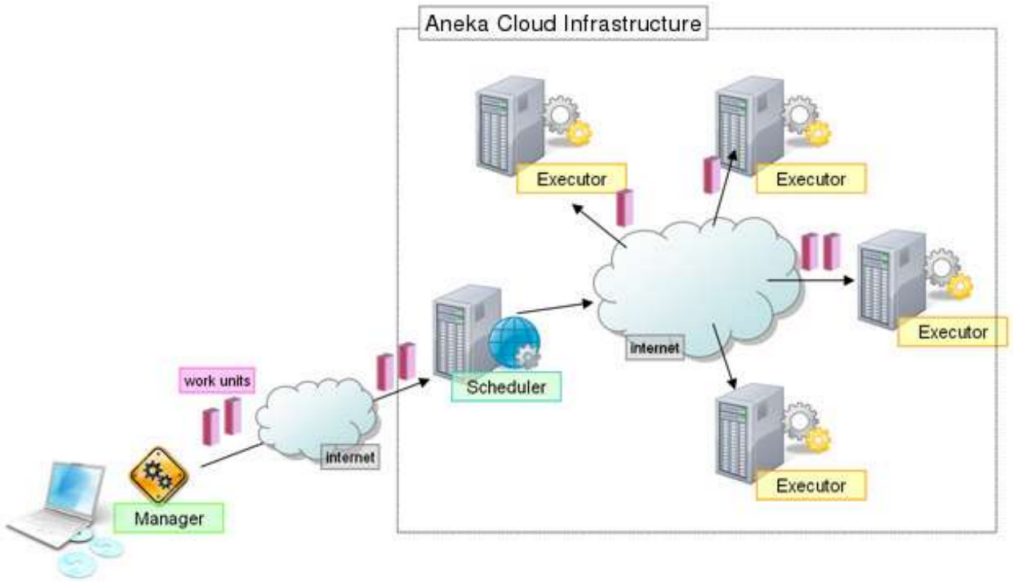

Figure 3.2: Aneka System Components View [16]

Applications

It appears that Aneka has been successfully used in many scenarios. Article written by Christian Vecchiola, Suraj Pandey and Rajkumar Buyya presents one of them [31]. It describes using Aneka for Classification of Gene Expression Data. Analysis of gene expression helps researchers to identify the relationships between genes and diseases and check how cells react to a particular treatment. It was possible to use Aneka to create Cloud-CoXCS classifier which is a cloud-based implementation oF CoXCS - the best such classifier according to authors' tests. It is intrinsically parallelizable, so the authors used Task Model which allowed them to easily execute the application using Amazon EC2 infrastructure.

Aneka was also used by GoFront Group to speed up Maya 3D rendering by utilizing idle power of legacy PC's. They achieved 24x speedup by using 50 machines. [15]

Summary

Aneka is a potentially powerful platform which could be used to not only execute but also manage all other aspects of scientific applications in a cloud. It's ability to create hybrid clouds using physical machines as well as virtual ones creates possibilities to scale out one's application into a public cloud if necessary. Aneka seemed to meet almost all of our requirements. It even supports Mono on Linux out of the box. However, the fact that this is a commercial software forced us to seek other solutions. We felt that there should exist a similar, even if greatly simplified, platform which would be open source and therefore available for everyone to use and build upon.

3.2.2. Windows Azure HPC Scheduler

Overview

Windows Azure HPC Scheduler [6], as its name suggest, is a Microsoft's High Performance Computing solution dedicated to Microsoft Azure. It uses the older name of the platform (Windows Azure), because it seems that it wasn't updated for a while (last version works with Windows Azure SDK for .NET released in October 2012). However, when we started work on this thesis and were evaluating existing solutions it seemed a viable option.

Windows Azure HPC Scheduler comes with an SDK which allows developers to create distributed, scalable, compute-intensive applications which can be executed over multiple Windows Azure role instances. It has built-in support for Message Passing Interface, Service-Oriented Architecture and Parametric Sweep Applications.

Architecture

Windows Azure HPC Scheduler deployment consist of a few elements which are presented in Fig. 3.3.

SQL Azure Database

Stores the job queue and configuration data.

Head node

Manages job scheduling and SOA workloads.

Compute node

One or more compute nodes are responsible for actual execution of jobs.

Front end

Web-based job submission portal (Windows Azure HPC Scheduler Portal)

Usage

First of all, Windows Azure HPC Scheduler must be deployed to a set of machines. It's done in a standard Azure cloud services way - one can use Visual Studio to create cloud service project, generate cspkg and associated configuration files and upload it to Azure. Then, configured role instances will be created and started. Microsoft published a sample project which does all that and also demonstrates execution of basic sample applications. The generated package is quite big - over 70 MB, where actual sample application binaries account for less than 5 MB. When all the nodes are started user may submit the job either by browsing to Windows Azure HPC Scheduler Portal and entering execution parameters or by connecting via remote desktop to the head node and using command line to do the same thing. While it may be useful for scenarios where users have some discrete, well-defined jobs to execute, it seems rather inconvenient as a way to integrate distributed execution capabilities with existing applications.

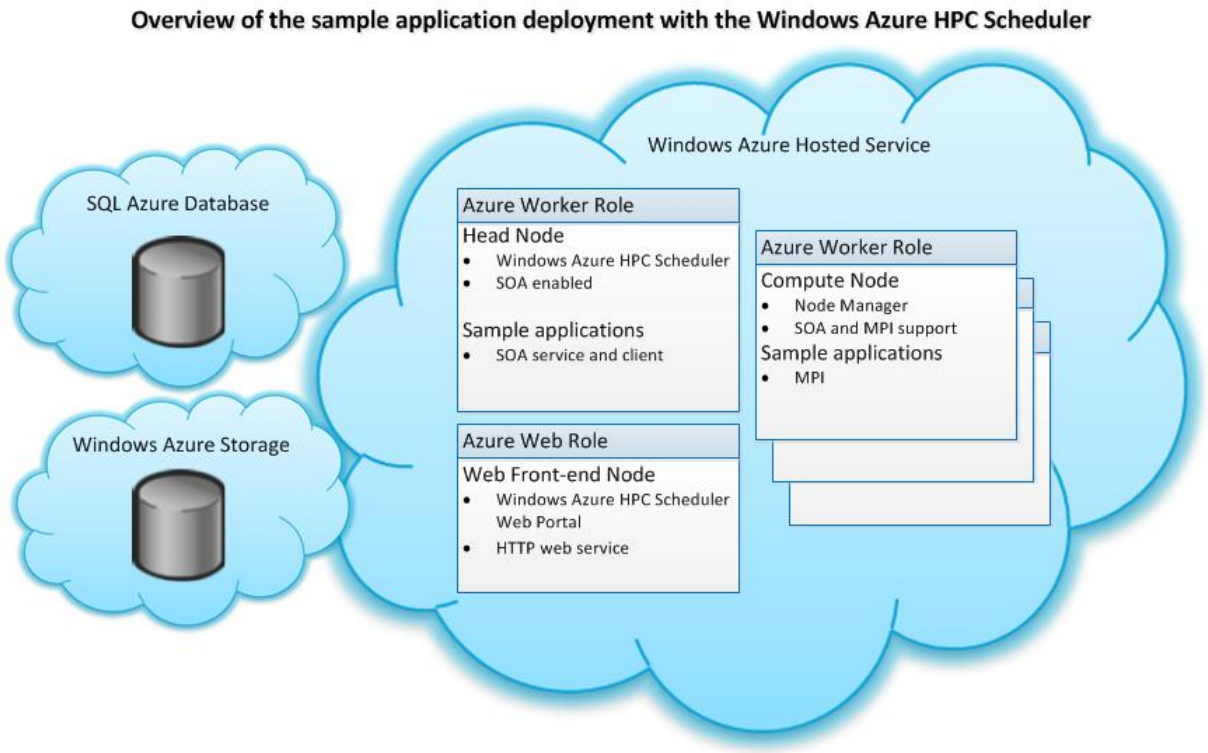

Figure 3.3: Windows Azure HPC Scheduler sample application deployment overview. [6]

Summary

All in all, Windows Azure HPC Scheduler is an interesting solution, however it didn't meet our requirements for a few reasons. To start with, as far as we discovered, defining new tasks requires redeployment of the whole HPC Scheduler package, which is quite inconvenient, or manual uploading new binaries. Additionally, Linux is not supported, which was one of our requirements. Lastly, as mentioned before, Windows Azure HPC Scheduler is not being updated to support newer Azure SDK versions and its documentation is quite limited. We needed something more lightweight, robust and versatile.

4. Design of the Distributed Task Library

The goal of this chapter is to show the design of Distributed Task Library which was created for purposes of this thesis and which allows for execution of arbitrary tasks in various environments in a distributed manner. At the beginning the initial concept is presented, then the final architecture is described and finally a few diagrams visualize the details of DTL design.

4.1. Overview

Distributed Task Library (DTL) is a utility which we decided to create in order to achieve the thesis's goals. It had to be lightweight, simple, portable .NET library allowing for distributed execution of tasks defined in runtime. It was supposed to also do some basic management operations like horizontal scaling. The complete list of requirements is included in section 1.2.1.

4.2. Initial concept

The initial vision was to develop an execution platform which would accept a job defined by user and distribute its execution over multiple nodes using a master-worker paradigm. The preliminary high-level overview of the platform is shown in Fig. 4.1. The idea was that in the end there would be a frontend component, e.g. a web application which would allow user to submit a job. Job would consist of set of binaries and a collection of data files. Then, they would be sent to the backend service. This service, core of the DTL, would be responsible for splitting the job into tasks, sending them to execution sites and collecting the results. The split would be made by dividing the collection of input files into chunks which could be processed by the binaries. This would effectively result in a parametric sweep model of execution. Tasks sent to execution sites would be processed by site controllers which would know how to dispatch them to worker nodes, collect the results and send them back to the main service. Execution sites were supposed to be public or private clouds, clusters or solitary physical machines. They would internally use queues or other native dispatch mechanisms.

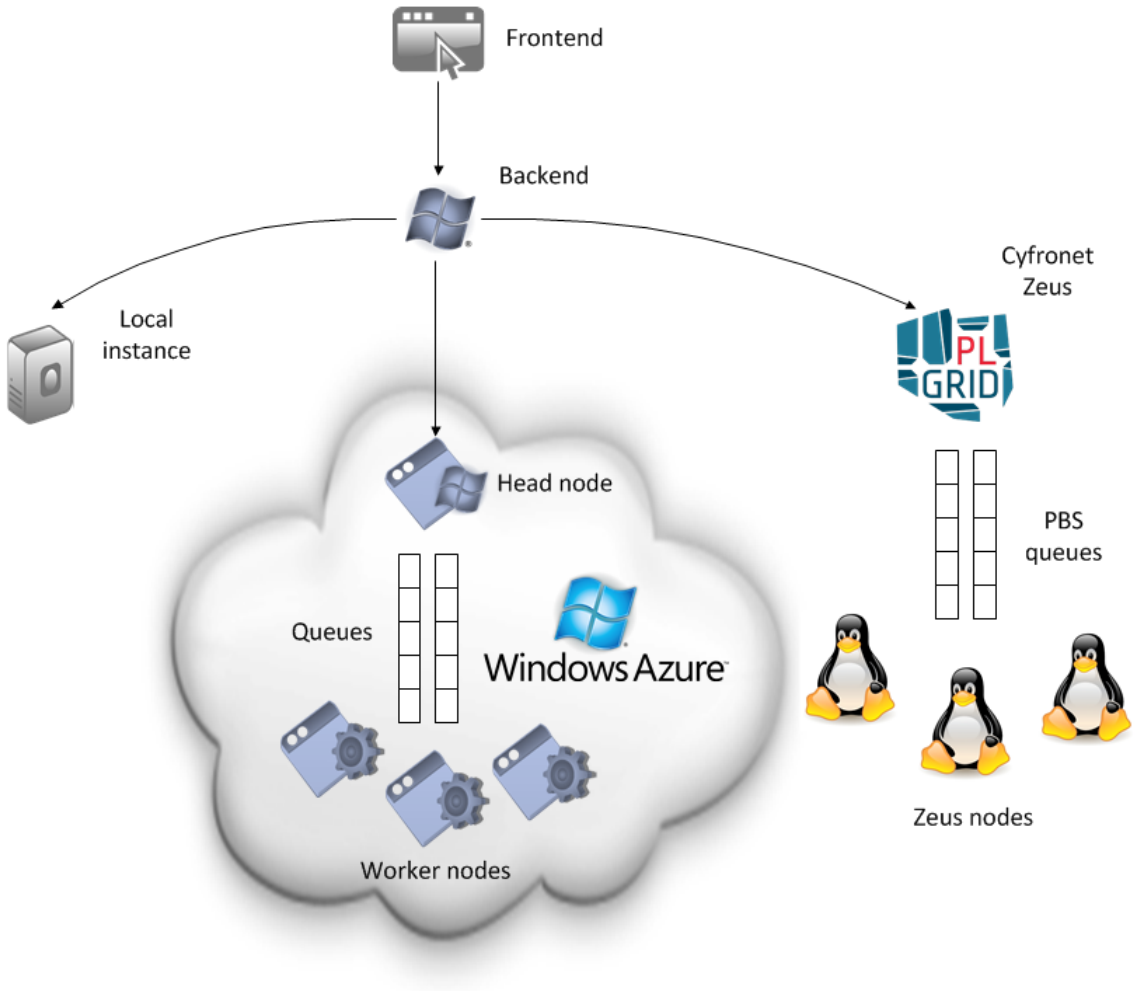

Figure 4.1: DTL preliminary high-level overview

4.3. Architecture

Eventually we simplified the initial concept in pursuit of better quality, versatility, maintainability and ease of use. We decided to focus on the core part of the platform and add a frontend application if time would allow. The mandatory goal was to create library which would be comparably powerful and provide as simple API as Microsoft's Task Parallel Library [12], but designed to work in a distributed environment, hence the name - Distributed Task Library.

4.3.1. Components

The entire system is built using loosely coupled components. They are tied together using well-defined interfaces, so every component may be easily swapped to one implemented by user or created by some third-party providers. Fig. 4.2 shows the final, high-level architecture overview diagram. Main customization points are Workers, Queues and Storage providers. They are external platforms which are accessed by custom DTL components. They don't have to come from one vendor, user can e.g. use Amazon SQS, Microsoft Azure Storage and a mix of workers from different platforms.

Controller

Main component of DTL. It accepts job requests, analyzes and prepares them for execution and sends to other components of the system. It also collects the results and makes them available to the user. Controller doesn't have direct contact with Workers. Instead, it uses Queues to send tasks and receive results from them. It's also responsible for transferring data from and to Storage nad Binaries Repository.

Worker

Executes queued tasks. Uses Queues, Storage and Binaries Repository to obtain task definitions, input data and upload results. By design it should be a long running application (daemon/service) and it's a component which can exist in multiple instances. Workers don't need to have connectivity with Controller, they must be able to access Queue and Storage Providers, however. Workers can use local Common Data Cache to cache data which is used by multiple tasks so it doesn't need to be downloaded from Data Storage every time.

Queue Provider

Hosts and gives access to queue infrastructure. It should have ability to create and delete queues as well as perform standard operations on them - put and get. Design assumes one queue of each presented kind, but thanks to modular nature of the system, one may e.g. create a set of queues, one for each platform, to ensure best performance and distribute the load over them. Another assumption, or rather observation is that publicly hosted queue systems have relatively low maximum message size limit (64 KB for Microsoft Azure Queues and Amazon Simple Queue Service). Task input data often goes well into mega- or even gigabytes making it impossible to use queues only. That's why DTL's queue message contain only reference to the data which is kept in Data Storage.

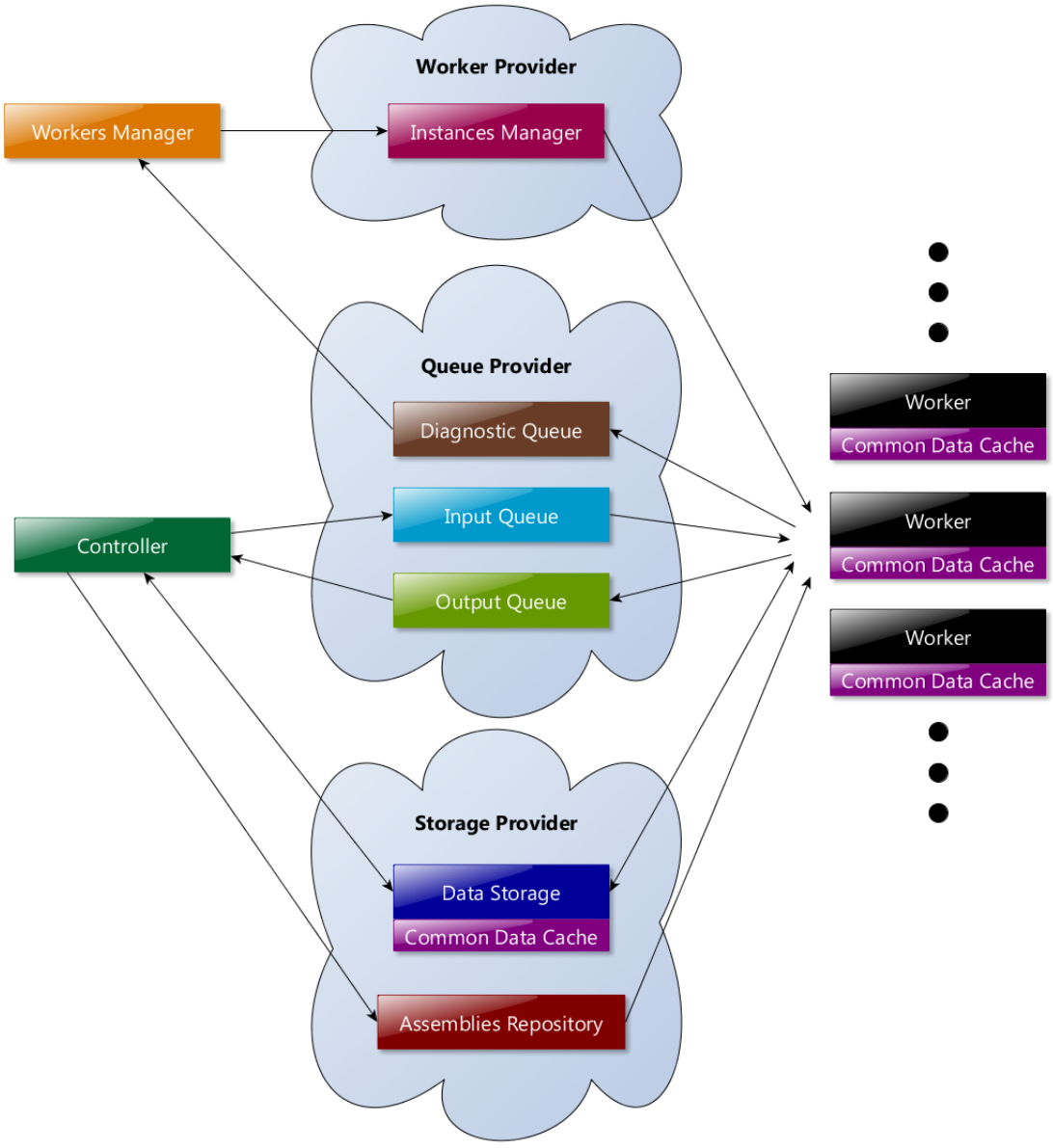

Figure 4.2: DTL architecture overview

Input Queue

Queue which contains collection of scheduled tasks inserted by Controller. Idle workers take them one by one and perform execution.

Output Queue

Contains task results inserted by workers.

Diagnostic Queue

Primary source of workers diagnostic information. Used by Workers Manager to receive information about number of workers and their status.

Storage Provider

Many tasks carry a large payload which must be delivered to workers and generate results which in turn have to be sent back. Storage Provider must provide storage space for this binary data (BLOBs) and an API to access it.

Data Storage

Component created on top of Storage Provider. Provides transparent access to stored data and means to manipulate it. Stored data is, by design, just an arbitrary collection of bytes, hence it doesn't have any inherent identity. Data Storage assigns ID to each BLOB and returns it to uploader. This way uploader can use this ID to refer to this BLOB later and avoid reuploading the same data. This mechanism is a part of Common Data Cache.

Assemblies Repository

We realized that applications should be handled differently than raw data. We knew beforehand that we will be using .NET as our programming and execution environment so idea behind Assemblies Repository is to store all the .NET assemblies used by tasks so they could be reused in future. Many .NET applications use common, popular assemblies so it would be a waste of bandwidth to upload them every time.

Workers Manager

Responsible for managing workers from all execution sites. It mainly deals with requesting change of the state of workers (create, start, stop, delete) based on current system state, execution site properties (e.g. VM performance and cost) and strategies set by user. It's and optional component, i.e. it doesn't have to be used at all to execute tasks if user manages the workers by himself.

Worker Provider

Platform which is able to provide workers in form of physical or virtual machines running DTL worker service/daemon.

Instances Manager

DTL component which is responsible for interfacing with specific Worker Provider and exposing standardized API for basic instance operations (create, start, stop, delete).

4.4. Diagrams

This section presents details of architecture using class and sequence diagrams. Each diagram is described in its caption.

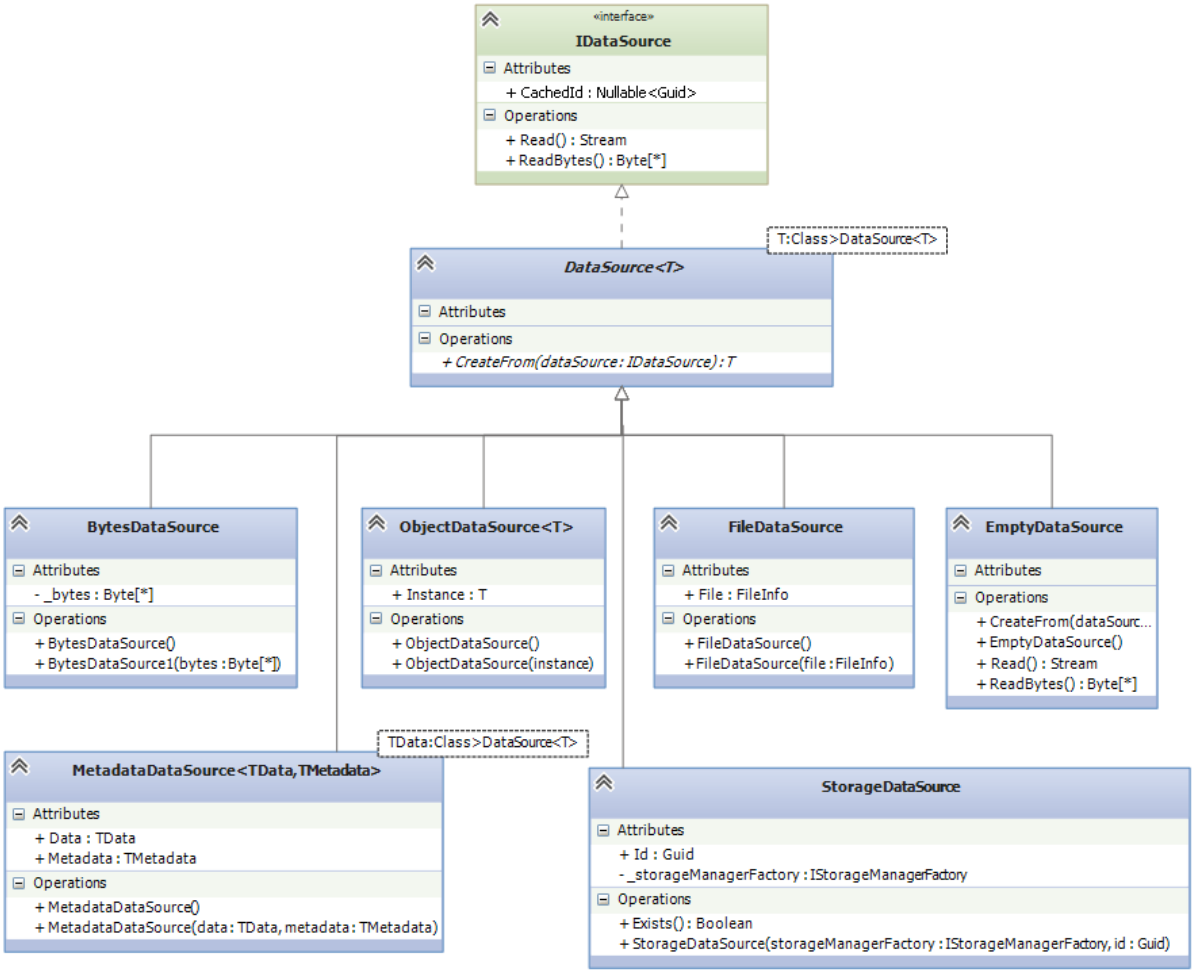

Figure 4.3: IDataSource represents every data on which DTL operates. It is a very simple interface which requires only ability to represent data as a stream of bytes. ReadBytes method is included for convenience reasons. CacheId is used by multi-level caching system which minimizes data transfers. IDataSource is versatile, can be easily sent over network but it is not very practical to use. Therefore, we created its base, generic implementation which enforces ability to generate concrete DataSource instance based on provided IDataSource instance. Then, there is a range of DataSource<T> implementations which are built in DTL and which should cover many use cases. ObjectDataSource has the ability to wrap any binary-serializable object. MetadataDataSource uses it to combine a metadata object with some other, arbitrary DataSource. EmptyDataSource is a placeholder for situations when there is no actual data needed.

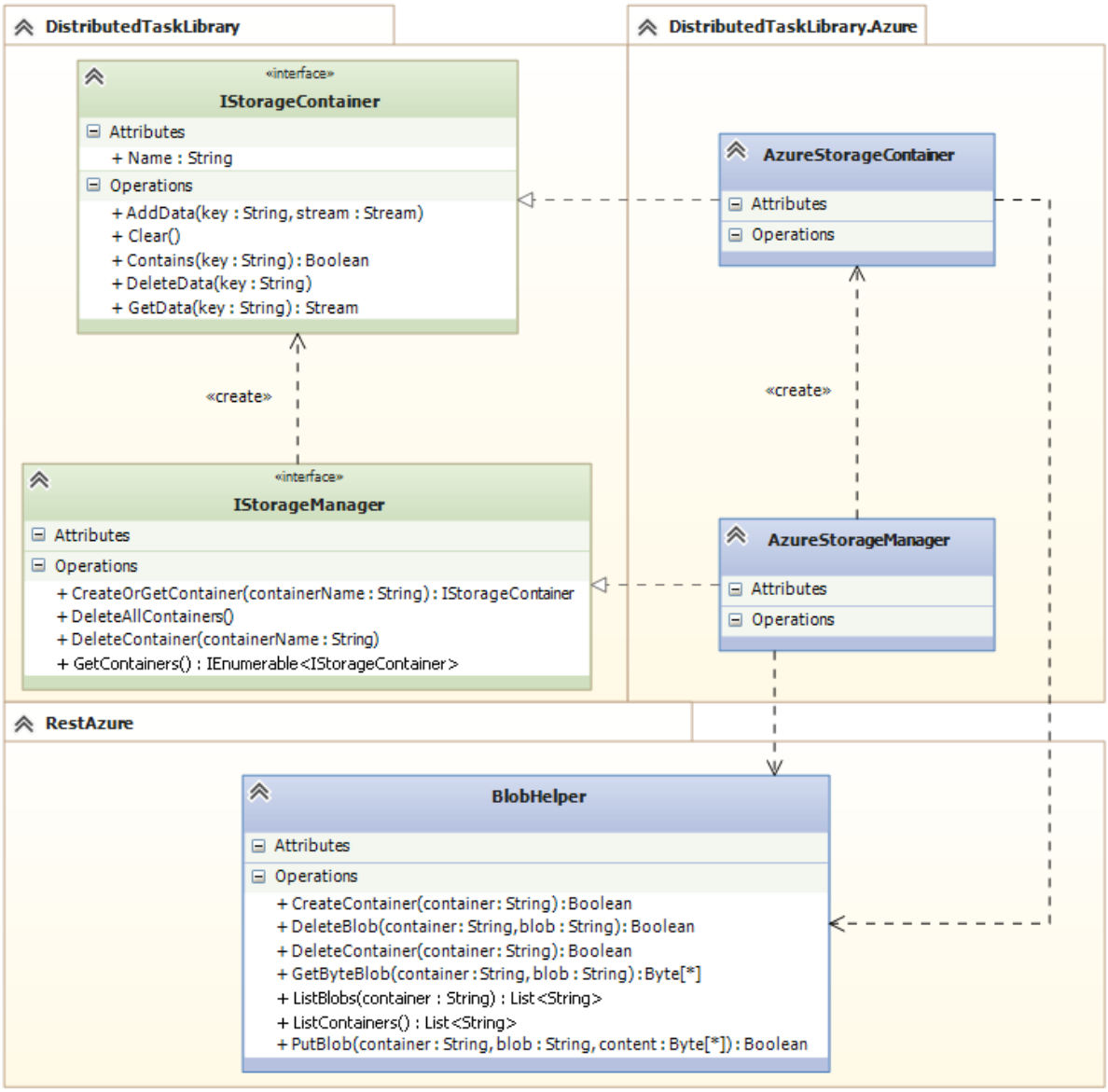

Figure 4.4: DTL relies on storage services for transferring data between Controller and Workers. It defines two interfaces: IStorageContainer, which governs adding, retrieving and clearing data, and IStorageManager which manages containers. They can have various implementations, based on filesystem, database, or dedicated blob data storage services. We created AzureStorageContainer and AzureStorageManger implementations which use REST BlobHelper under the hood.

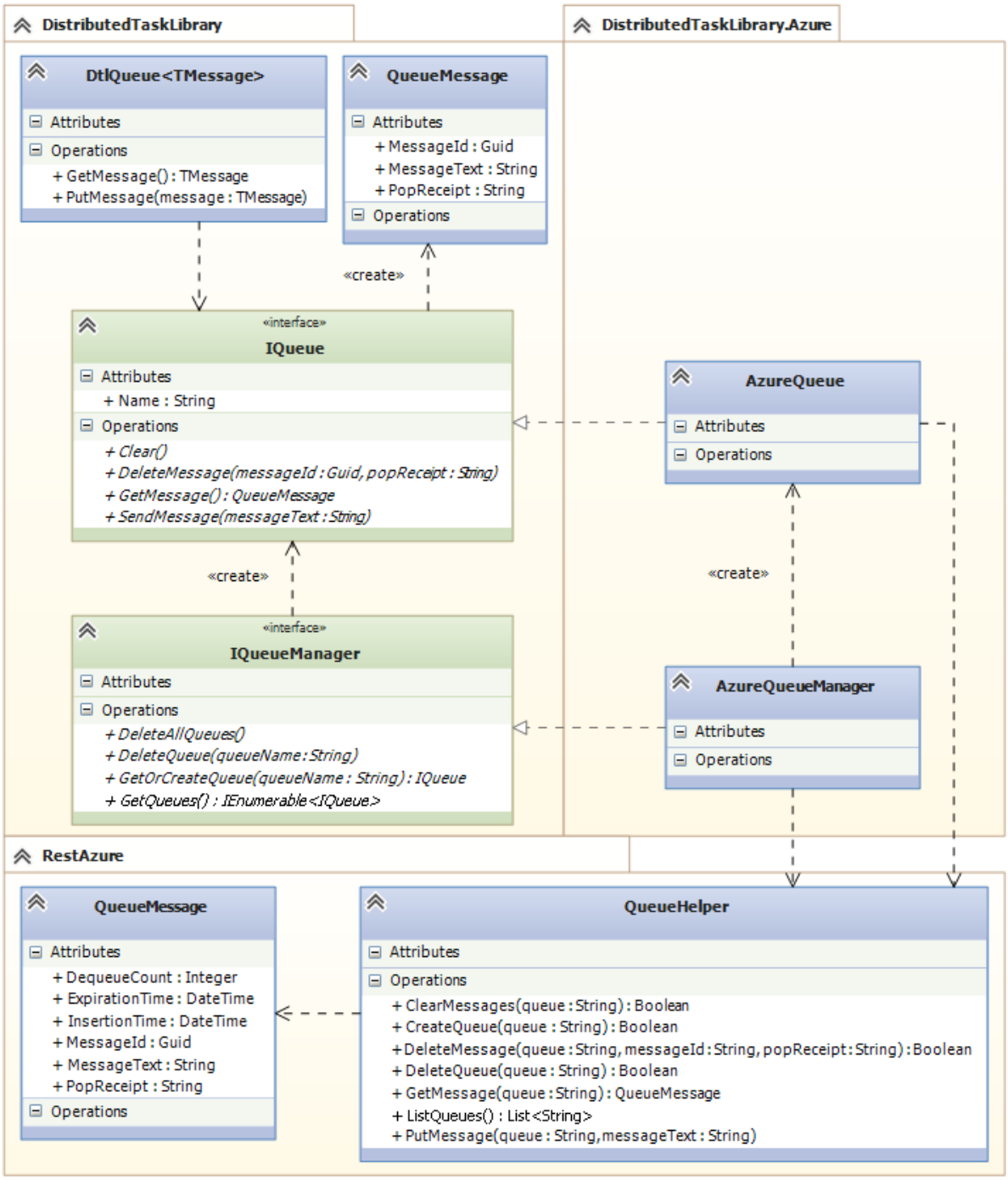

Figure 4.5: Queues are used as means of transferring small portions of data and load balancing tasks. Similarly to storage, there are two interfaces defined. IQueue provides standard queue operations like put and get. IQueueManager is used to manage queues. Additionally, there are two queue-related classes defined in DTL. QueueMessage is just a simple message string message carrier with added metadata. DtlQueue is a wrapper which supports concrete message types and is able to serialize and deserialize them. We created AzureQueue and AzureQueueManger implementations which use REST QueueHelper under the hood.

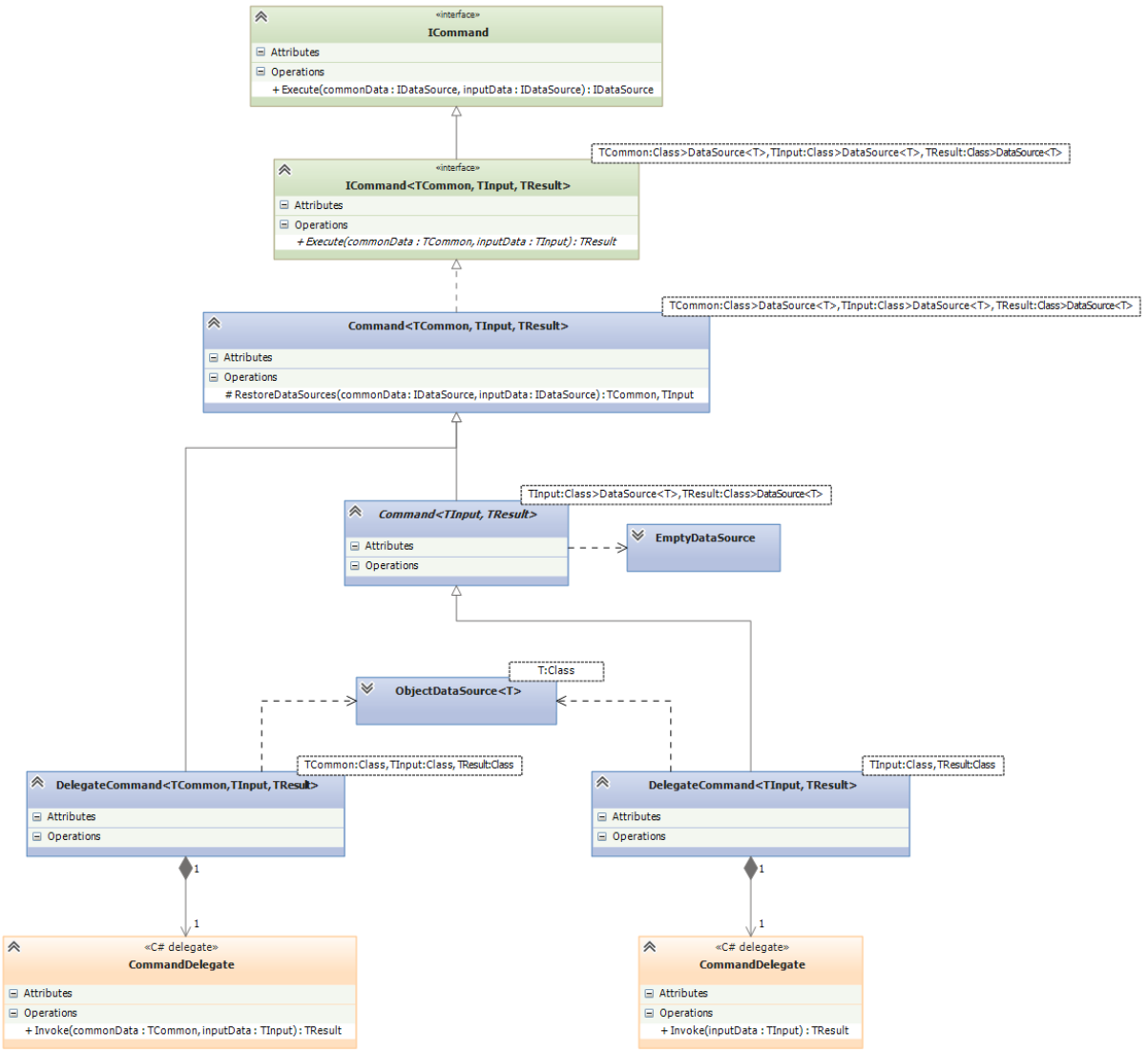

Figure 4.6: ICommand is a basis for all the commands. In this form it is actually used only when transferred between Controller and Worker. Its extension, generic ICommand, adds information about actual data types used. Command base class implements ability to restore these concrete types instances from plain IDataSource objects. Then, there are more built-in specializations and implementations. One branch uses EmptyDataSource in place of TCommon simplifying scenarios where common data is not needed. Both hierarchy branches end with DelegateCommands which greatly facilitate using DTL for plain .NET objects by taking care of wrapping them in data sources and by accepting delegates as command definitions.

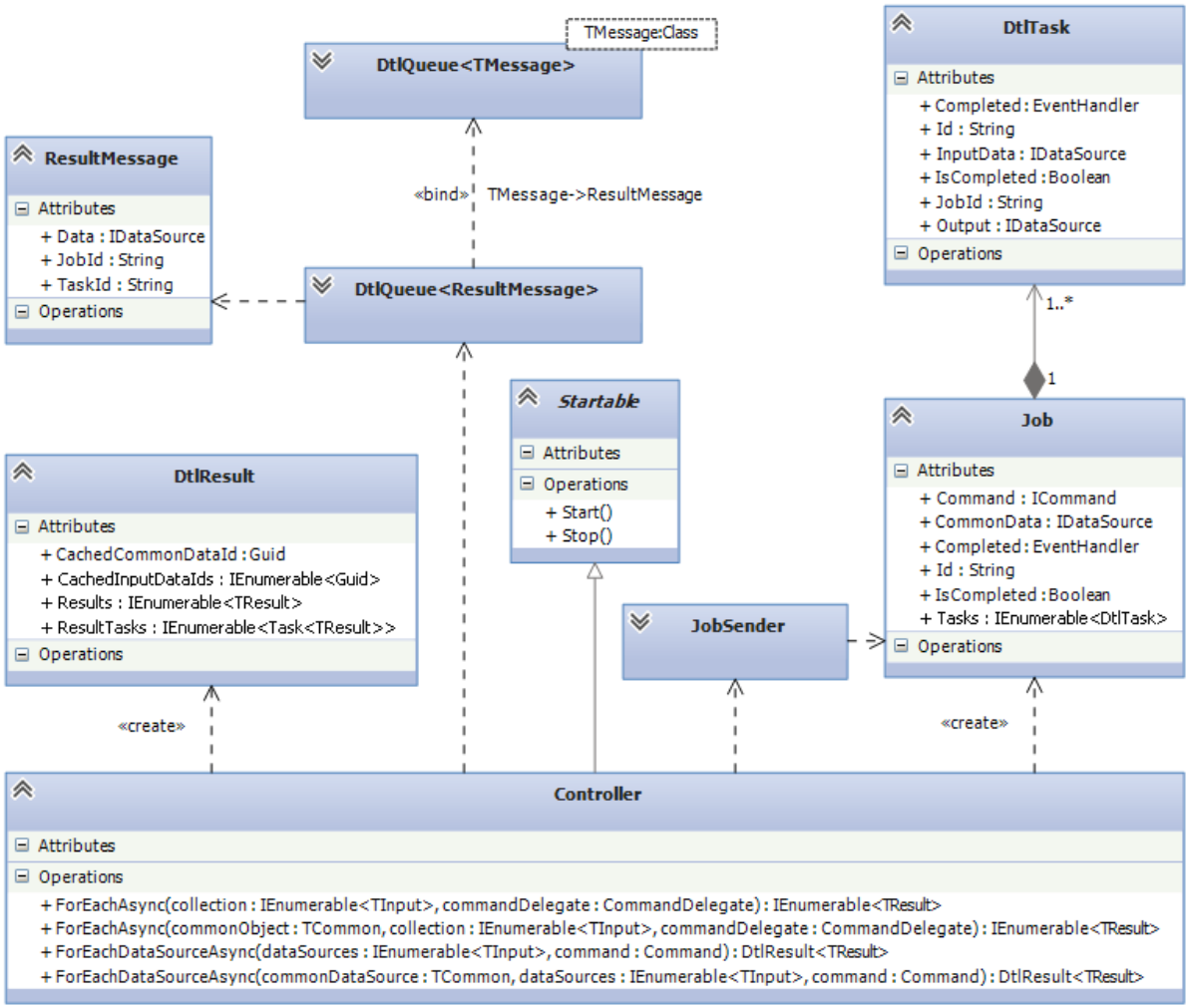

Figure 4.7: Controller is the main client component which exposes four ForEach* methods which work on collection of input data. Two ForEachAsync methods work with delegates and plain .NET objects. When other data sources are needed, then ForEachDataSourceAsync methods should be used. Both methods are async in such meaning that right after the job is sent, the caller gets a lazily evaluated object (IEnumerable or DtlResult) which may be used to retrieve results when they are available. DtlResult exposes the results in the form of Task objects making integration with .NET TPL easier. Data and command provided to Controller are wrapped in Job and then, DtlTask objects. They facilitate transfer to Worker and take part in the mechanism of completion notification. JobSender is described in Fig. 4.8

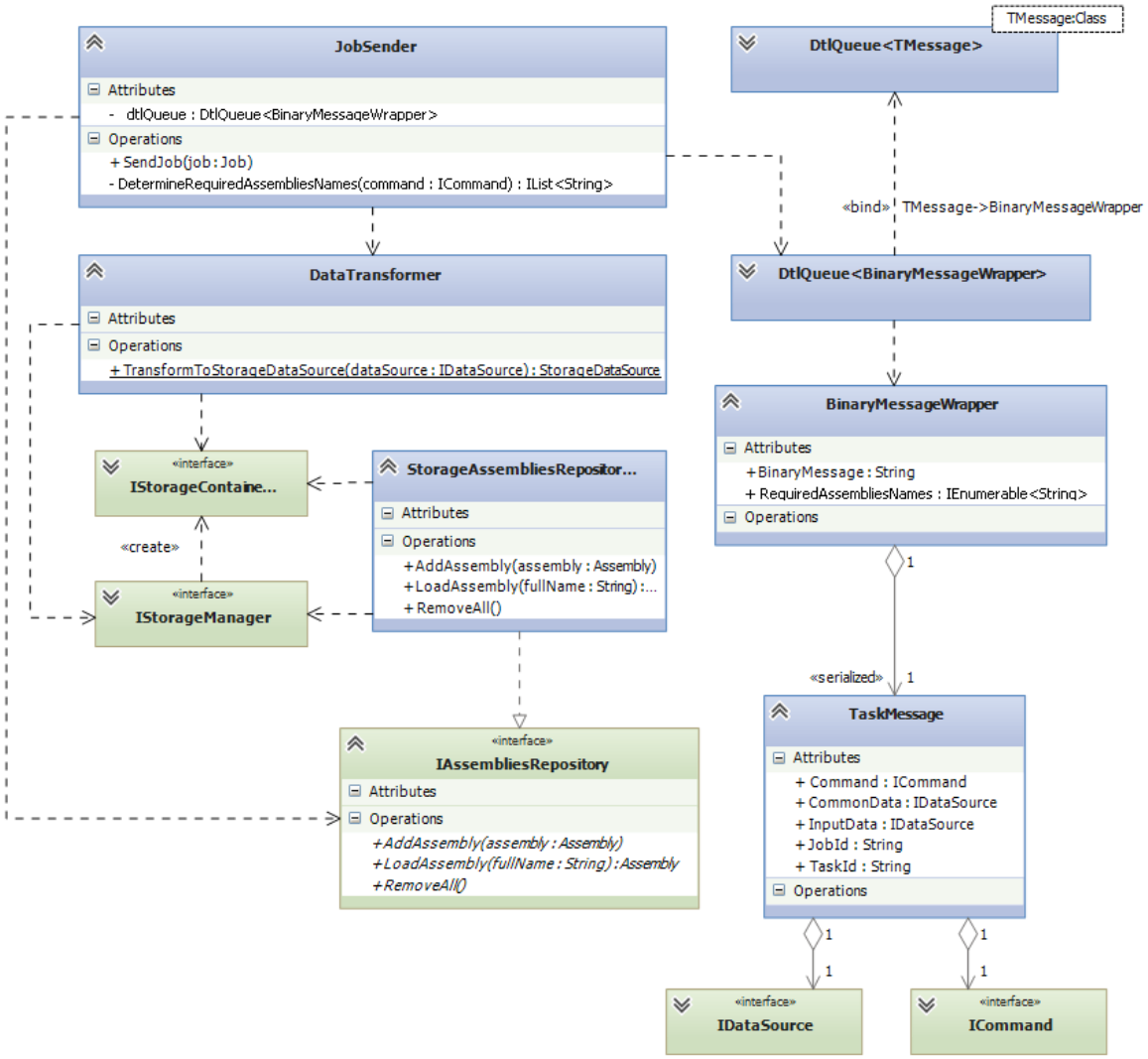

Figure 4.8: JobSender as its name suggests is responsible for sending jobs from Controller to Workers. The whole process is described in Fig. 4.13.

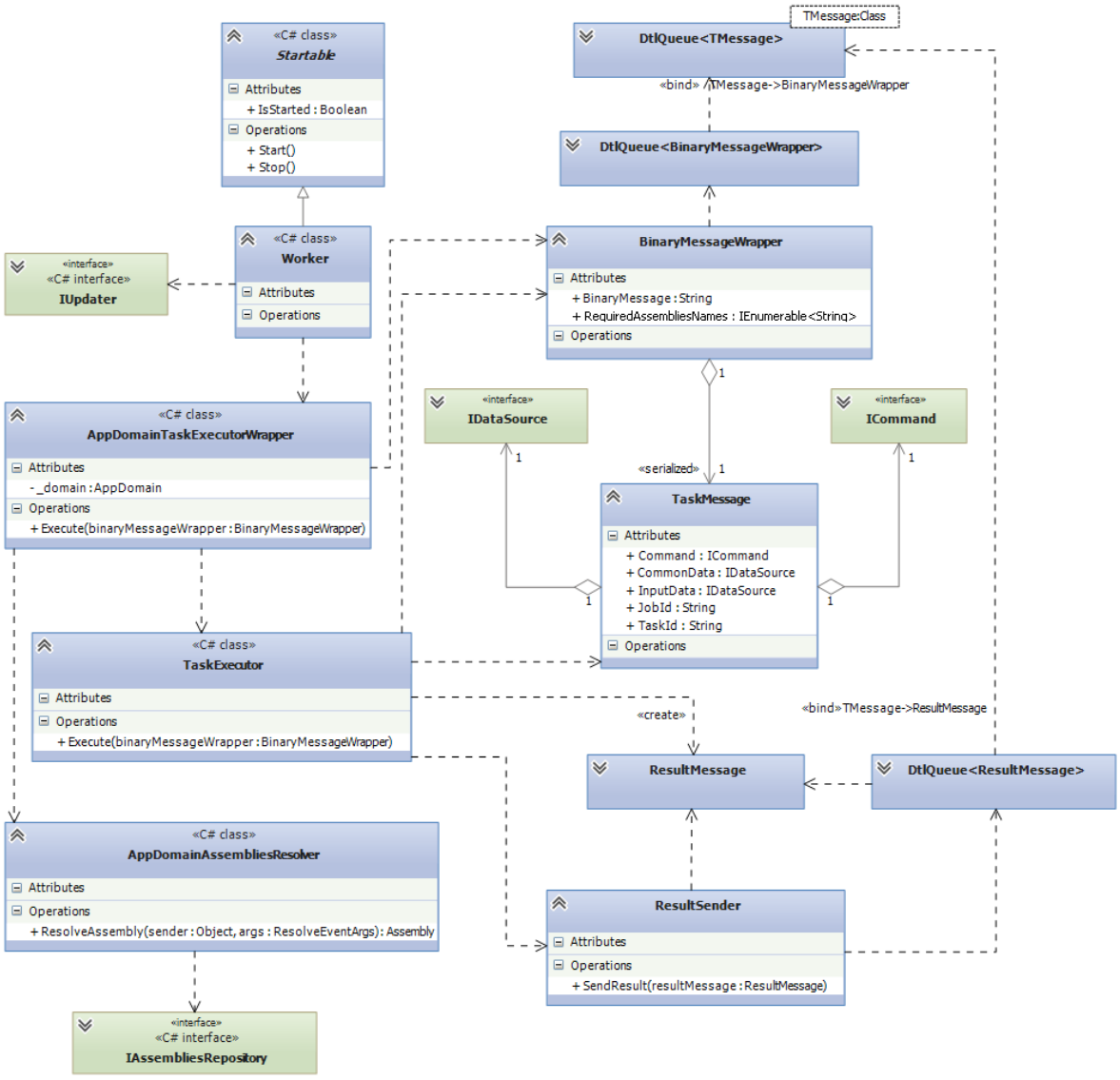

Figure 4.9: Worker is component deployed on one or machines and is responsible for execution of tasks. This process is described in Fig. 4.14

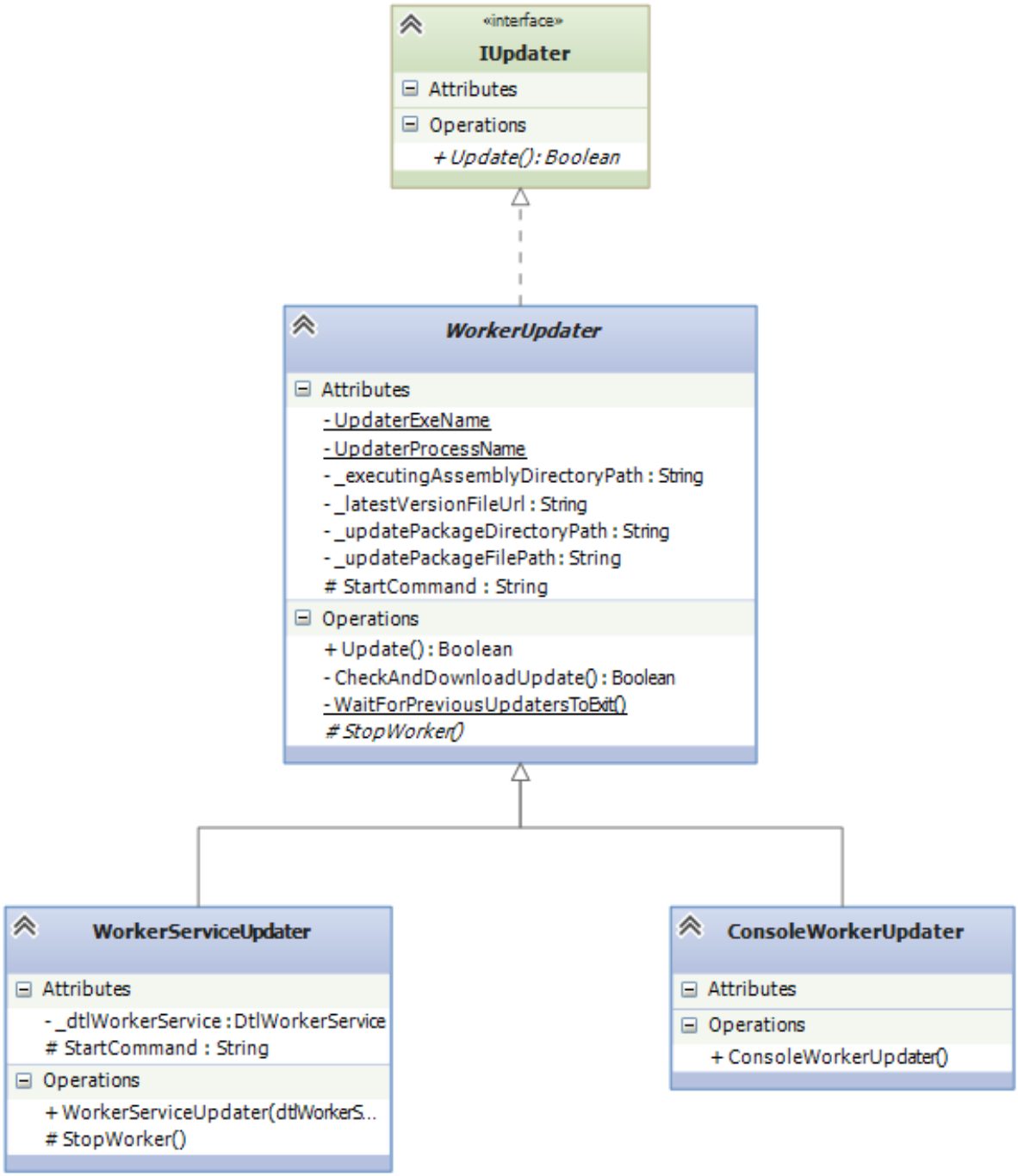

Figure 4.10: Updater is not a DTL component per se but we created it to make testing of new versions easier and may be used for distributing DTL worker updates to many machines. It is designed for both Console and Windows Service workers. Updater is able to check a given URL for newer version of DTL, download it, shutdown current worker, update it and start it again.

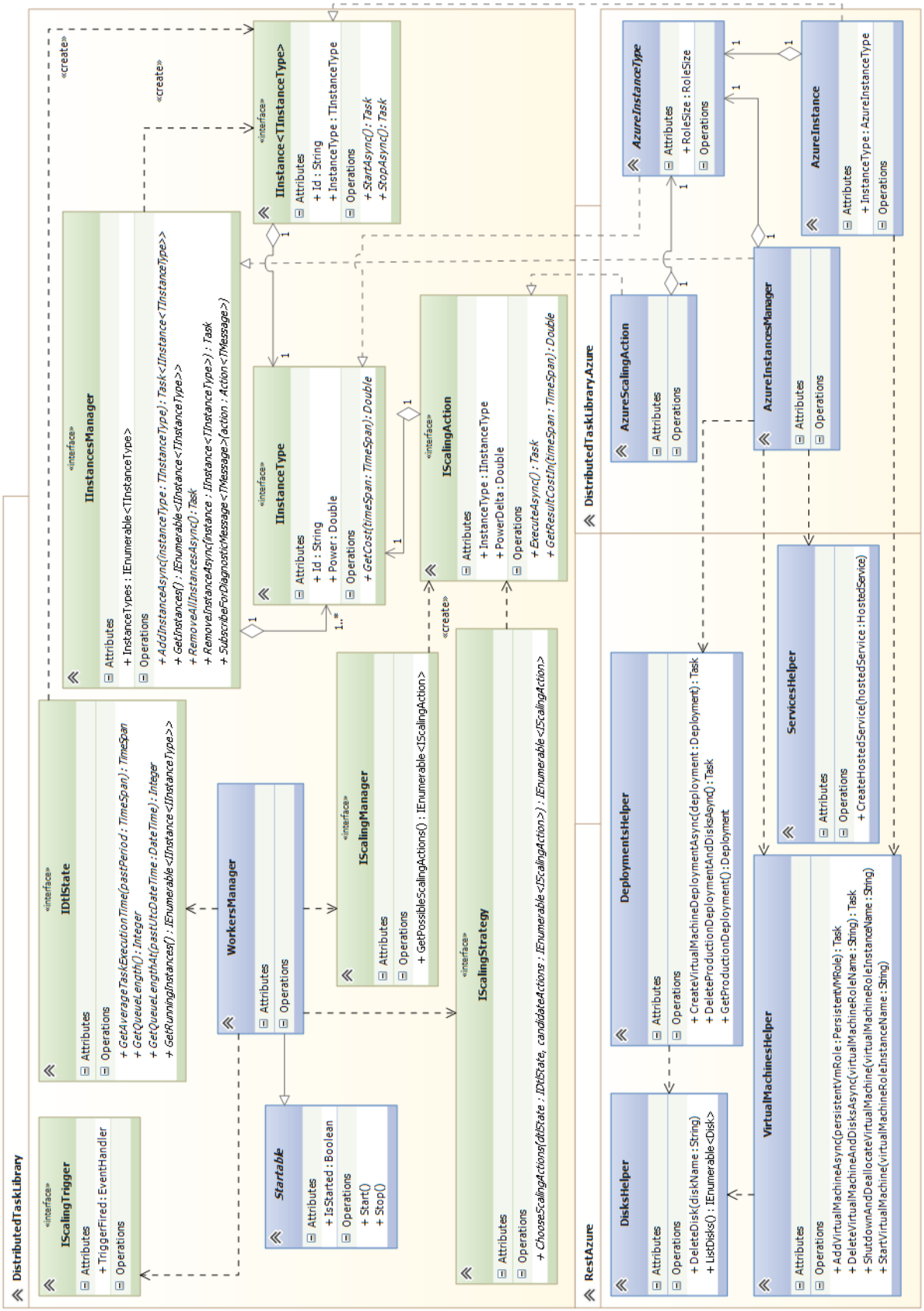

Figure 4.11: There is a part of DTL which is designed for fairly complex management of workers. It is centered on WorkerManager which, manages number and type of running workers.

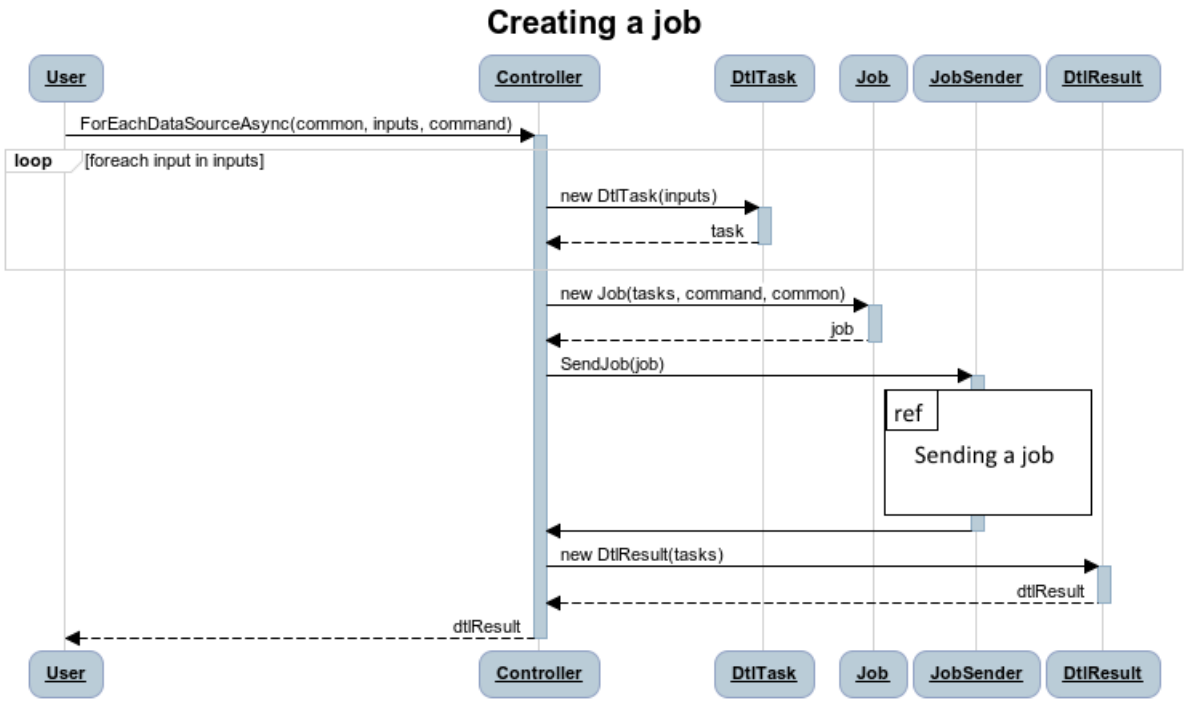

Figure 4.12: The whole process is initiated by a use calling one of the ForEach* methods of Controller. User provides common data, collection of input data and a command to execute. Controller creates one DtlTask for each piece of input data and aggregates them all into a job. Then the job is sent to workers which is shown in Fig. 4.13. After it is sent, User gets a DtlResult object (or IEnumerable) which can be used to obtain results when they are available.

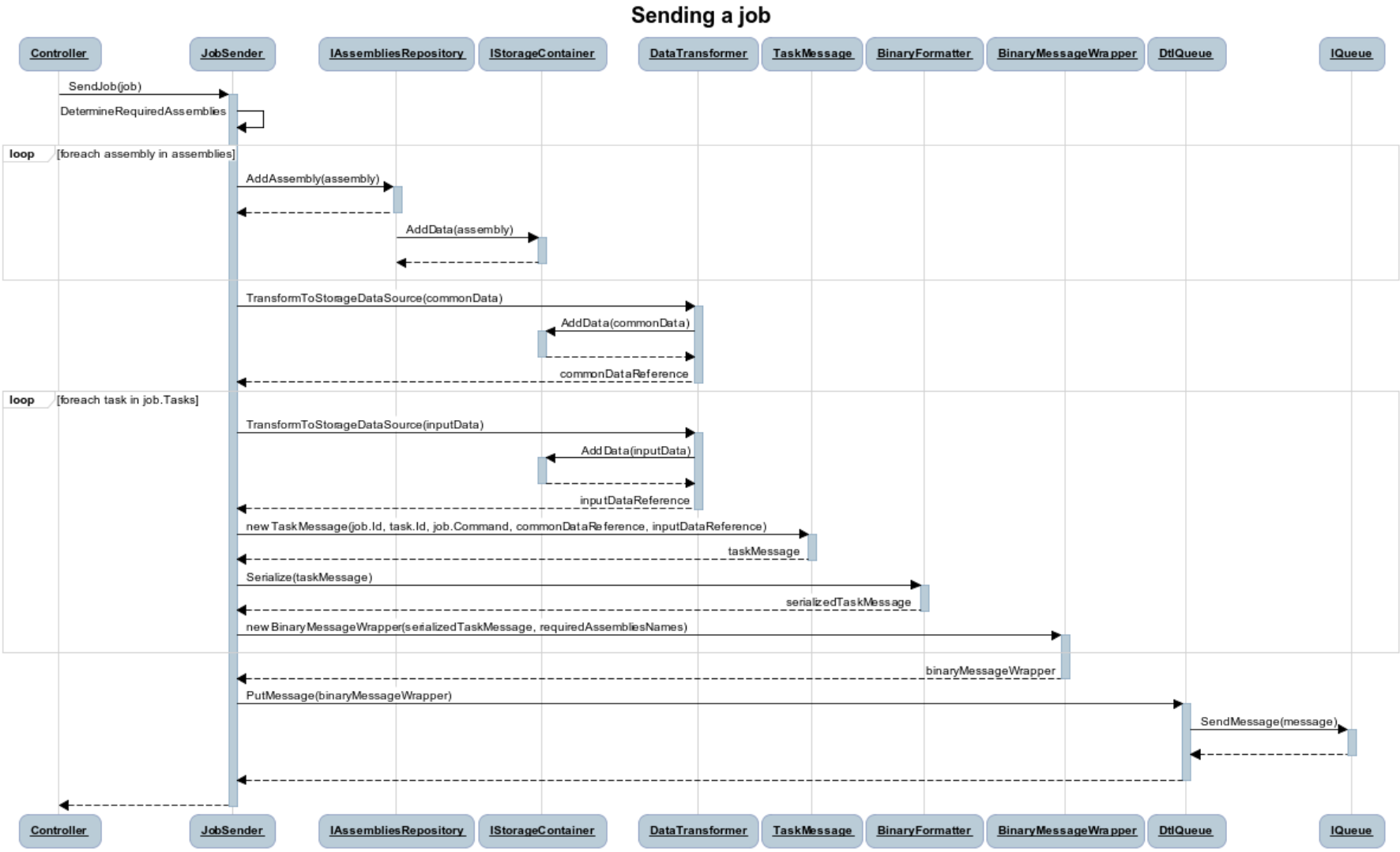

Figure 4.13: First, required assemblies are determined by analyzing command's dependencies and uploaded to assemblies repository. Then input and output data is uploaded to storage and original data sources are replaced with references. Finally, TaskMessage is created, serialized and sent.

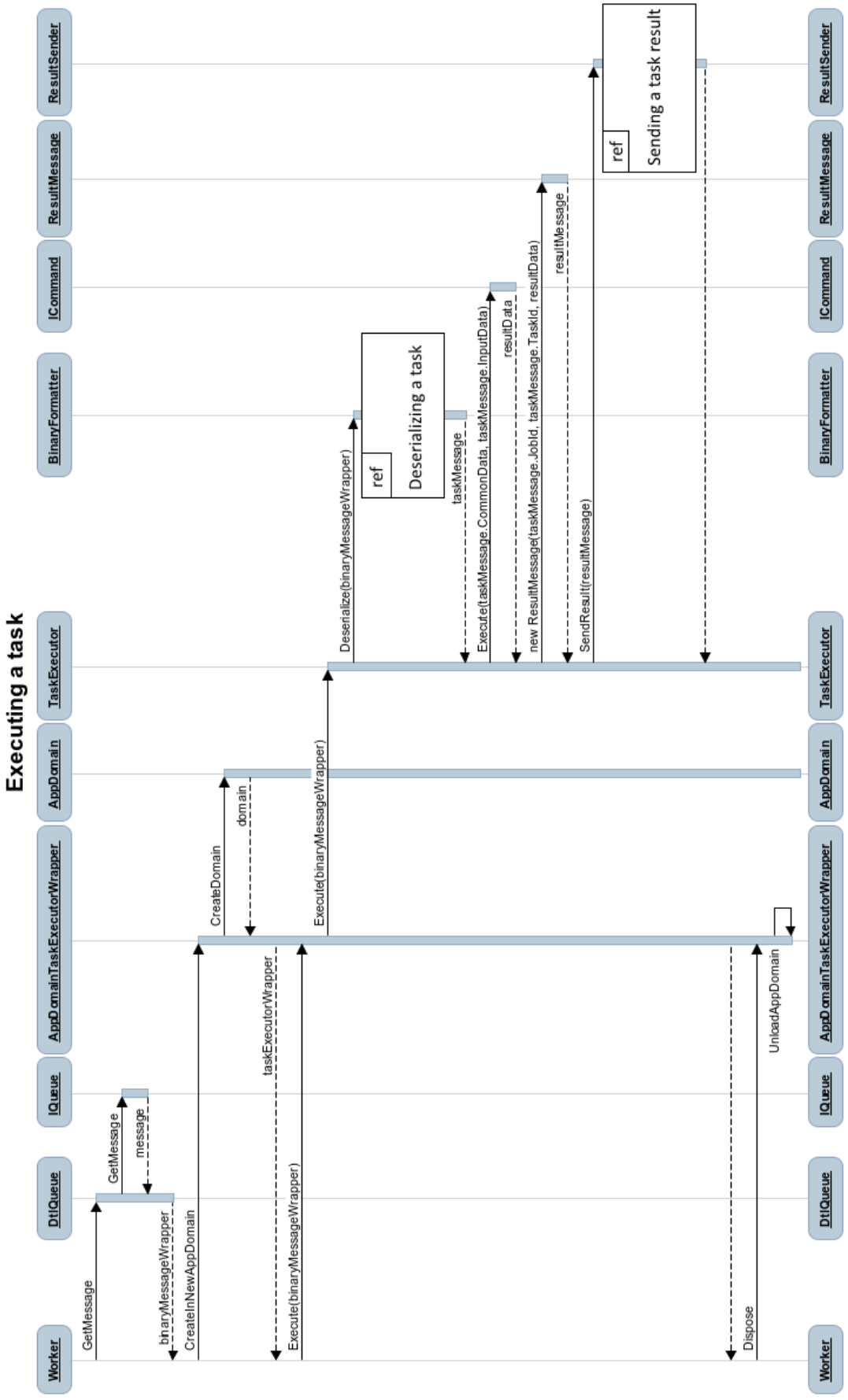

Figure 4.14: Task execution is the sole responsibility of Worker. When task appears in input queue, worker gets it and passes it to executor which creates a new app domain. Then it deserializes the TaskMessage (Fig. 4.15) Afterwards, executor can execute the command and send result to Controller (Fig. 4.16

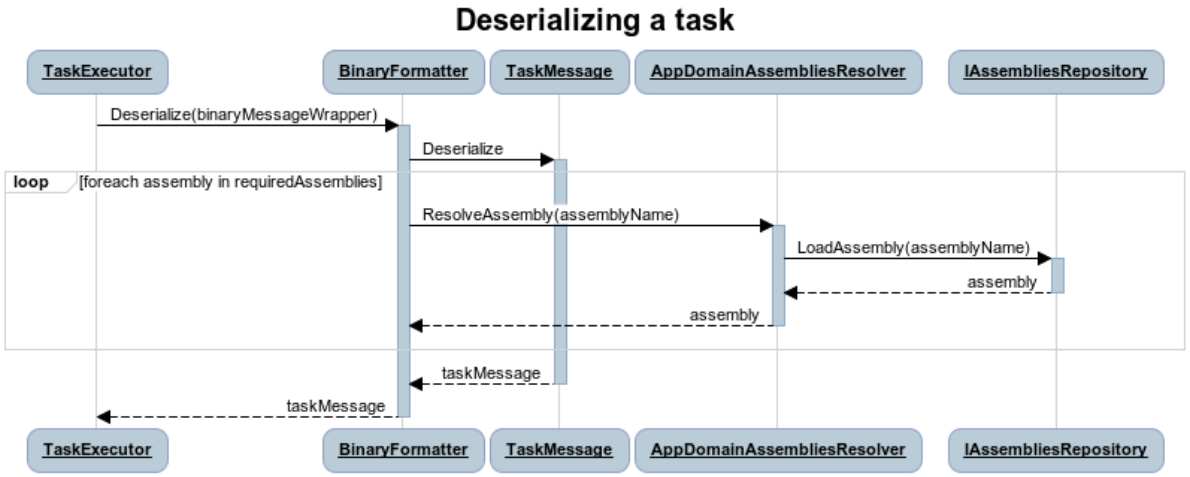

Figure 4.15: Deserializing a task causes .NET runtime to ask for all required assemblies. AppDomainAssembliesResolver is hooked to the ResolveAssembly event of the newly created app domain so each request is forwarded to IAssembliesRepository which supplies the assemblies. Then, the task message can be instantiated and returned.

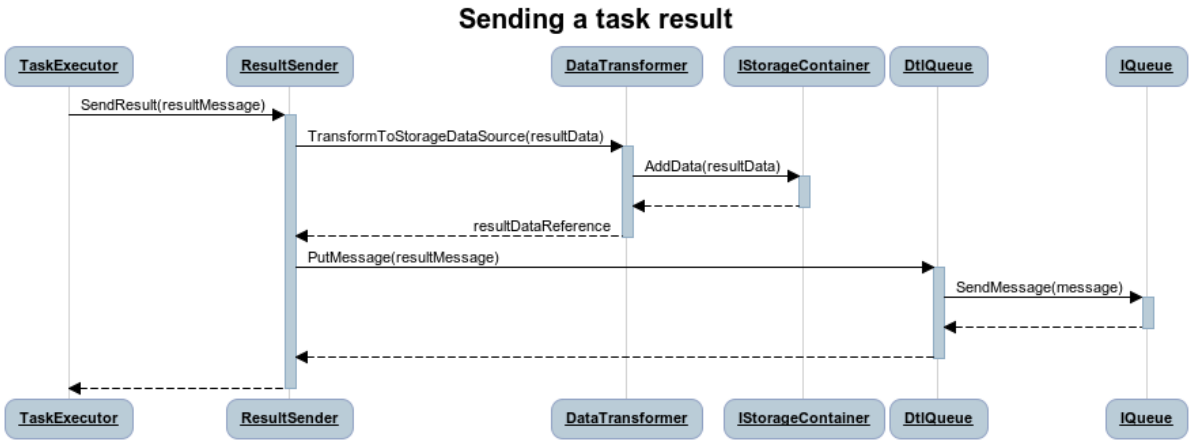

Figure 4.16: TaskExecutor creates a result message and passes it to ResultSender. Then its data is uploaded to storage and message is sent to Controller via result queue.

5. Implementation of the Distributed Task Library

This chapter is focused on details of Distributed Task Library implementation. First part presents technology stack DTL is built upon along with short description of each of its parts. Then, the most interesting challenges we encountered during development phase are described. Finally, we provide a few examples of DTL usage in a form of complete code snippets.

5.1. Technology stack

Minimizing number of technologies used in a project is a goal worth pursuing. We've had this idea in mind since the beginning of DTL development. Actually one of the goals was to make a lightweight library which means cutting down on external dependencies. On the other hand, it's even more important to avoid reinventing the wheel. Fortunately, .NET Framework is quite mature and provides many solutions out of the box. In the sections below, we briefly list technologies used during development with short information how they contributed to the final result.

5.1.1. Technologies

Microsoft .NET 4.5 and C# 5.0 - The latest major versions of .NET framework and C# language brings several new features. Probably the biggest one is introduction of async/await keywords and new asynchronous programming patterns coming with them. Distributed processing and many cloud operations like e.g. managing virtual machines or transferring data are inherently asynchronous so it helped to make the implementation more natural.

Mono 3.6.0 - We had some problems with earlier versions but right now DTL works correctly on Linux. See section 5.2.4 for details.

Microsoft Azure - The core technology of this thesis. Comprehensive overview can found in chapter 2

5.1.2. Libraries

Web API - Used for accessing Microsoft Azure REST API. Well-suited for use in asynchronous manner.

log4net - A popular .NET logging library. It's actually the only one external dependency of DTL right now. Eventually it should be optional.

NUnit - Framework which integration with ReSharper made testing more robust.

5.1.3. Tools

Microsoft Visual Studio 2012 - Microsoft's IDE which we could use thanks to MSDN Academic Alliance program.

ReSharper - Greatly improved Visual Studio user experience. Available to us through Academic License granted to AGH.

Git - As a revision control and source code management system.

5.2. Challenges

As it is the case in virtually every software project, we faced several challenges during Distributed Task Library implementation phase. Some of them were fairly easy to overcome, others required significant amount of time and effort to solve. We briefly describe and show our solutions for the most interesting ones below.

5.2.1. Dynamic task definition and execution

One of our main goals was to support dynamic task definition and execution. It means that workers shouldn't need any prior knowledge about tasks being executed. In other words, worker wasn't supposed to load some task definitions repository on startup and then support only those tasks. We wanted to send task definitions to workers through network and make them execute those tasks.

The simplest approach seemed to be sending a package with executable binaries, along with a string of parameters or a batch script which worker would execute after unpacking the package. Then, the executed application would have to create some artifacts, files most likely, which could be sent back to the client. This approach, albeit simple, had some drawbacks. First of all, it required that task contained a standalone executable which in some cases would have to be created just for the purpose of distributed execution. There are many applications which could potentially benefit from distributed processing but often, e.g. in case of web applications, it would be hard or inconvenient to extract a part of them into some stand-alone executable. Also, file-based input and output handling could be cumbersome. Additionally, supporting multiple platforms (Windows and Linux) would require specific handling because the client would have to provide separate binary packages for each platform or deal with it on his own (e.g. by using portable solutions like Java, .NET, Python, etc.).

We wanted to do something different, more natural, versatile and convenient, at least for .NET developers. The idea was to allow developers to take their demanding part of code, wrap it into a single Execute method, compile it, and after starting a few workers, run the resulting application as usual and see how its execution is automatically distributed over workers. The dynamic task definition and execution was also meant to help during application implementation changes. Let's suppose that developers

wanted to test performance after different code changes. We wanted to confine the whole process into simple - change, build, run loop. No configuration changes or manual tinkering with workers should be needed.

The whole idea is based on the concept of .NET assemblies which are essentially .NET executables or dll libraries. DTL accepts tasks in a form of a class which implements ICommand generic interface shown in listing 5.1 or in a form of a delegate which matches Execute method's signature. The non-generic interface is used when type parameters knowledge is irrelevant and would require unnecessary type parameters spread. Execute method takes two parameters: commonData which is data shared by all tasks of the same job and inputData which is piece of input dedicated to be processed by one instance of the task. Return value is the result. More details on data handling can be found in section 5.2.2. Upon execution, DTL detects in which assembly the task is implemented and what are its dependencies. Then it gathers all these assemblies and uploads them to our Assemblies Repository, if they are not already present there. Identification of assemblies is based on their fully qualified name which looks like this:

log4net, Version=1.2.13.0, Culture=neutral, PublicKeyToken=669e0ddf0bb1aa2a

This means that if developers depend on some of their own assemblies, they must remember to change their version along with code changes. This can be automated in Visual Studio. Task message which is sent to workers contains binary serialized command instance, list of required assemblies and references to common and input data stored in Data Storage. Workers can then ask Assemblies Repository for the listed assemblies, download them and keep in cache for later reuse.

Listing 5.1: ICommand interface - must be implemented by tasks

```
1  public interface ICommand
2  {
3      IDataSource Execute(IDataSource commonData, IDataSource inputData);
4  }
5
6  public interface ICommand<in TCommon, in TInput, out TResult> : ICommand
7      where TCommon : DataSource<TCommon>,  new()
8      where TInput : DataSource<TInput>, new()
9      where TResult : DataSource<TResult>, new()
10 {
11     TResult Execute(TCommon commonData, TInput inputData);
12 }
```

Loading assemblies was one of the major challenges. .NET does not allow loading multiple versions of the same assembly at the same time. Neither it allows unloading them so if each task loaded its assemblies then after some time they would all raise the memory usage considerably. After some research we found out that loading multiple versions of the same assembly is actually possible but only into separate application domains. It's also possible to unload an application domain, effectively unloading all

the assemblies. Unfortunately, creating and using additional application domains complicates design and code. Nevertheless, we decided to use this method in DTL. Each tasks is executed in separate application domain where it loads its required assemblies. After execution application domain is unloaded.

All in all, dynamic task definition and execution is one of the major accomplishments of the implementation part of this thesis and a feature which is hard to come by in such form.

5.2.2. Data format

One of the design decisions was to make data handling as flexible and as simple as possible. It had to work with all possible platforms and data storage providers and yet be trivial to use. Every piece of data is ultimately just a collection of bytes. Based on this, we decided that the only requirement for user is to provide a stream of bytes as input and common data. This led to creation of IDataSource generic interface and a base class for all data sources 5.2. As in case of ICommand, the non-generic interface is used when type parameters knowledge is irrelevant and would require unnecessary type parameters spread. IDataSource is not meant to be data container per se, merely an access point to it. DTL uses it to read data from the given source and send it to data storage. Then it is restored on the target machine in the original form thanks to CreateFrom method and guarantee that concrete DataSource implementation has a public, parameterless contructor. Of course, while such approach is extremely versatile, handling raw bytes is not very convenient from the task implementator's point of view. Therefore, we created a few DataSource implementations which may come handy in many scenarios. Here are the probably most useful:

EmptyDataSource

Sometimes common, input or result data is not used by the particular task. In such situation EmptyDataSource may be used to indicate this.

BytesDataSource

As simple as it gets - one can use this data source to transfer raw bytes and doesn't have to implement it himself.

ObjectDataSource

One of the most useful data sources. Wraps a .NET object and provides an automatic way to restore it for execution. The only requirement is that the object is binary serializable.

FileDataSource

Many tasks use files as input or output data format. This data source reads a file contents and is able to recreate the file from byte stream.

MetadataDataSource

Simple implementation of combined data sources. First one, called metadata, is read from in its entirety into memory, to determine its size. Then the size is written at the beginning of the output

stream, so later it's possible to separate and recreate the two original data sources. It's useful for scenarios when a large stream, like a file, must be accompanied by a relatively small one, e.g. object representing execution parameters.

StorageDataSource

Sometimes data already exists in data storage but was not cached e.g. if it was uploaded manually or by some third-part application. Then it may be used by task directly if StorageDataSource is used to point to that data. Additionally, it's used by DTL internally to access the data it uploads to storage.

Listing 5.2: IDataSource interface and its base implementation which must be extended by all custom data sources

```
1  public interface IDataSource
2  {
3      /// <summary>
4      /// ID used to identify and find data source in cache
5      /// </summary>
6      Guid? CachedId { get; }
7
8      /// <summary>
9      /// Creates new data stream on each call
10     /// </summary>
11     Stream Read();
12
13     byte[] ReadBytes();
14 }
15
16 public abstract class DataSource<T> : IDataSource
17     where T : DataSource<T>, new()
18 {
19     private readonly Guid? _cachedId;
20     public Guid? CachedId
21     {
22         get { return _cachedId; }
23     }
24
25     protected DataSource()
26     {
27     }
28
29     protected DataSource(Guid cachedId)
```

```
30      {
31          _cachedId = cachedId;
32      }
33
34      public abstract Stream Read();
35      public abstract byte[] ReadBytes();
36
37      /// <summary>
38      /// Use do restore the concrete DataSource implementation from
            arbitrary IDataSource
39      /// </summary>
40      public abstract T CreateFrom(IDataSource dataSource);
41  }
```

5.2.3. Microsoft Azure REST API

Azure REST API exposed by Microsoft became our API of choice after thorough analysis of available options as described in section 2.4. This choice significantly increased the overall implementation time because we had to implement all the .NET client infrastructure which would correctly handle all the required requests and request. Following our modular design concept, all the Azure implementations of DTL interfaces were put into a separate assembly - DistributedTaskLibrary.Azure. It comes with its own configuration file which allows for setting up all the necessary Azure settings like e.g. account information. However, the thin wrappers for Azure REST API were extracted to yer another assembly. This allows for replacing it in future if one decides to use wrappers based on native SDK or some other solution.

Storage Services

Fortunately, we didn't have to write Storage Services REST client from scratch. We came by Azure Storage Samples project by David Pallmann [23] which showcases most of the Storage Services REST API. We took it, upgraded it to use the .NET latest REST client library - Web API, and added our own modifications to fit the needs of DTL. The greatest nuisance it solves is the so-called Shared Key Authentication which ensures that that the data stored in Storage Services is secure. This authentication method is based on a custom Authorization header sent with every request. Its construction is fairly complicated and involves several steps and requirements concerning parameters' case, order, coding, etc. [7]. Eventually, we created full Azure implementations of DTL's queue and storage interfaces.

Service Management

Service Management REST API is a separate and a quite different from Storage Services. First of all, authentication is much simpler. It's just a matter of attaching a correct certificate (registered beforehand

in the given Azure account) to each request. Secondly, most requests are based on XML posted in message's body while in Storage Services custom headers are used extensively. This time we created our client wrappers from scratch. We noticed that XML format of requests and responses is simple enough to be mapped to regular objects using XML serializer. This hugely improved readability and maintenance of the code. We decided to utilize natural asynchronicity of most of the Service Management operations. It was relatively easy to implement using the new async/await features of C# 5.0, however such approach required some change of mindset to anticipate and address all issues arising from asynchronous programming model.

5.2.4. Linux (Mono) support

One of our goals was to support execution on Linux-based systems by using Mono platform (see section 1.2.1). This ruled out making use of the official Microsoft Azure SDK for .NET which doesn't work for Mono. We had to use Azure's REST API and we decided to do it via Web API client. However, it was a brand new technology as of March 2013 and was only partially implemented in the official Mono release as of then (3.0.6). Right now, however, the latest Mono edition (3.6.0) has no problems whatsoever with running DTL. It's quite pleasantly surprising given that some very specific features of .NET are used in DTL like application domains combined with remoting, marshalling and binary serialization.

As of now, DTL worker for Linux comes only in form of command-line executable but there should be no problem in turning it into full-fledged daemon.

5.2.5. Worker auto-update

Pursuing the simplicity and minimum maintenance goal we chose to implement auto-update of DTL worker. During the intensive development phase it was very troublesome to update worker after each code change. Additionally, in the final environment, where DTL workers are deployed on dozens of machines it would be inconvenient to manually update worker on each of them. Of course, there are many solution which automate such tasks, like e.g. Puppet, but they are often fairly complicated tools and not always cross-platform. In some large, formalized environment they may be the best solution but for our purposes, creating a simple auto-update solution seemed like a better idea. This presumption was reinforced by positive outcome of the early spike and in a relatively short time we developed auto-update which proved to work sufficiently well.

Currently, worker checks for update only on startup but it could be triggered by other events, e.g. message from Controller or a timer event. Worker looks of update information under specified URL. If update is found, it's downloaded and extracted. Then, downloaded updater executable is started and old worker is stopped. Afterwards, updater replaces worker files and starts. It's a really simple process but is highly flexible thanks to the updater which is always downloaded and therefore update process can be modified with a new release if needed.

5.3. Examples of usage

This section presents a few examples showing how DTL can be used in simple scenarios.

Listing 5.3: Starting worker

```
var credentials =
    new AzureStorageCredentials("YourAccountName", "YourAccessKey");

var worker = AzureWorkerFactory.CreateAzureWorker(credentials);
worker.Start();
```

Listing 5.4: Simple example for the client side, showing extremely inefficient method of calculating Pi approximation. Plain, serializable objects are used for input and output so a delegate can be used to define the command. In this case no common data is used.

```
var credentials =
    new AzureStorageCredentials("YourAccountName", "YourAccessKey");

var controller =
   AzureControllerFactory.CreateAzureController(credentials);
controller.Start();

const int n = 200;
IEnumerable<double> terms =
    controller.ForEachAsync(
        Enumerable.Range(0, n),
        k => Math.Pow(-1, k) / (2 * k + 1));

Console.WriteLine("{0:#.##}", terms.Sum() * 4);

// Output:
// 3.14
```

Listing 5.5: Example of using common and input data along with command defined via delegate.

```
...
var commonData = "Hello, ";
var inputData = new[] { "World", "AGH", "DTL" };
IEnumerable<string> results =
    controller.ForEachAsync(
```

```
6           commonData,
7           inputData,
8           (common, input) => common + input);
9
10 // This will display each result as soon as it is processed by workers
11 foreach (var result in results)
12 {
13     Console.WriteLine(result);
14 }
15
16 // Output (lines order may differ):
17 // Hello, World
18 // Hello, AGH
19 // Hello, DTL
```

Listing 5.6: More complex example. The goal is to count number of word occurrences in text files. This requires using not only plain object but also FileDataSource, so we have to create a custom command class.

```
1  [Serializable]
2  public class CountingOccurrencesCommand
3      : Command<ObjectDataSource<string>,
4                FileDataSource,
5                ObjectDataSource<int>>
6  {
7      public override ObjectDataSource<int> Execute(
8          ObjectDataSource<string> commonData,
9          FileDataSource inputData)
10     {
11         var desiredWord = commonData.Instance;
12         var text = File.ReadAllText(inputData.File.FullName);
13         var occurrences = text.Split().Count(word => word ==
               desiredWord);
14         return new ObjectDataSource<int>(occurences);
15     }
16 }
17
18 ...
19
20 var desiredWord = new ObjectDataSource<string>("foo");
21 var inputFiles = new[] { "a.txt", "b.txt", "c.txt" }.Select(file => new
       FileDataSource(file));
```

```
22 // Each result is represented by TPL Task intance, so it is easy to to
       incorporate it into .NET async model
23 IEnumerable<Task<ObjectDataSource<int>>> resultTasks =
24     controller
25         .ForEachDataSourceAsync(
26             desiredWord,
27             inputFiles,
28             new CountingOccurrencesCommand())
29         .ResultTasks;
30
31 foreach (var resultTask in resultTasks)
32 {
33     resultTask.ContinueWith(task =>
34     {
35         int occurrences = task.Result.Instance;
36         // This will display each result as soon as it is processed by
               workers
37         Console.WriteLine(occurences);
38     });
39 }
```

6. Evaluation

This chapter contains results and finding of all the tests we performed. First section is dedicated to horizontal scaling performance tests. We measured how quickly a given number of virtual machines can be deployed and then, deleted. Second series of tests involves rendering a sample scene in popular raytracer - POV-Ray, using a varying number of Microsoft Azure virtual machines. Finally, we showcase implementation of DTL into a real-world, bioinformatics application - ExonVisualizer

6.1. Scaling performance

Ability to scale the number of running nodes (also called scaling out or scaling horizontally) robustly is very important regarding the Windows Azure's pay-as-you go billing model. Of course if you require constant computational power all the time then you don't need to change the number nodes dynamically but it is not always the case.

In a number of scenarios, public cloud is used as an execution environment for tasks which resource requirements vary greatly. One approach would be to instantiate as many resources, i.e nodes, as possible to satisfy even the most demanding tasks' needs. However, when you have to pay for each minute of a running instance, this strategy is far from cost-effective as some nodes would be, more or less often, idle. It seems more reasonable to keep active as few nodes as possible and add more on demand.

A perfect horizontal scaling should be:

- Fast - adding or removing an instance should not take much time
- Cheap - removed nodes should not generate any cost or it should be negligible compared with the cost of running instance

One can take a few different approaches to scaling out:

Using autoscale

Azure's built-in feature. We couldn't use it as described in section 2.2.2.

Starting and stopping instances

Manually emulating autoscaling behavior - creating as much instances as potentially needed but keeping those which are currently redundant in stopped state. User does not pay for stopped instances but is charged for use of the storage space which their disks occupy.

Creating and deleting instances

Instantiating virtual machines from a captured image and deleting them if not needed is slower but does not incur storage costs when machines are not needed. **This is the approach we used in the following tests.**

Mixed approach

One could imagine mixing the above methods and, based on some rules, first create a number of machines and leave them in stopped state, if potential need for them is predicted, and then start them when the time comes.

6.1.1. Horizontal scaling

As far as IaaS-based scaling is concerned, one of the possible approaches to horizontal scaling is adding and deleting virtual machines. The following scenario assumes that one wants to set up an environment consisting of n machines and then delete them all. Test procedure involved requesting instantiation of the given number machines and then deleting them all at once. More details on Microsoft Azure Virtual Machines can be found in section 2.2.2.

Test environment

A custom VM image was prepared for this test. It was created from a clean Windows Server 2012 VM with only one modification - DTL Worker Service was installed and set up to start right after the system boot. The image was kept in the Windows Azure Storage located in the same affinity group as the instantiated VMs (West Europe). User-created image has size equal to the size of the machine which it was created from. Currently it is 127 GB. It is far too large for this test purposes but Azure provides no way to request smaller OS disk. There are some unofficial ways [1] to trim the image but they are cumbersome and can render the image unusable in some cases. Additionally, a single Cloud Service was created upfront, because its existence is free of charge. Each time the requested VM type was Small Instance.

Test procedure

The test scenario consisted of the following steps:

1 **for** $n \leftarrow 1$ ***to*** *20* **do**
2 Request instantiation of n machines
3 Wait until received DTL Worker registration message from each machine
4 Wait 5 minutes for environment to settle
5 Request complete deletion of all machines
6 Wait until all machines are deleted
7 Wait another 5 minutes
8 **end**

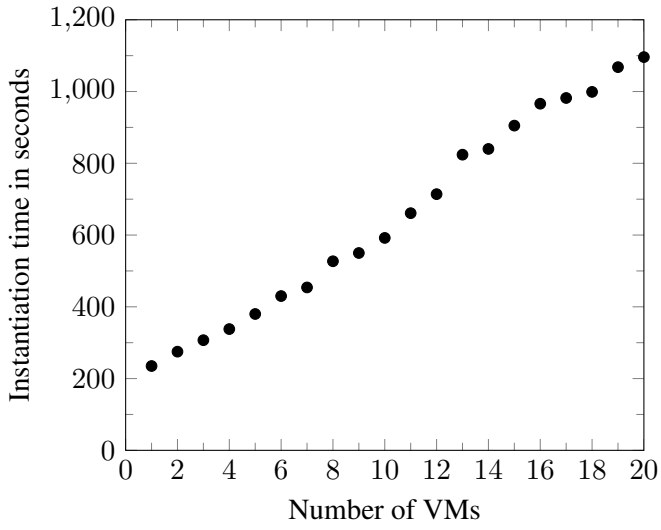

Figure 6.1: Instantiation times of 20 Small Instance VMs from 127 GB Windows Server 2012 image on West Europe Region.

The goal was to measure how fast is such scaling approach and to show the relation of number of requested machines to time of their instantiation and deletion. Three plots were chosen to present the tests' results. The first one 6.1 shows the times after which given number of machines were ready (sent registration message) when 20 VMs were requested. In the second plot you can see how long did it take to instantiate the given requested number of VMs. The third one shows time of complete deletion of all requested machines.

Instantiation time

Let's analyze the plot 6.1. First machine was up and running in about 235 seconds. It is not the worst result possible considering the big size of the image and the VM's specification. Each subsequent machine started within about 45 seconds after previous one, on average (with standard deviation around 25 seconds).

One may wonder why the plot is pretty much linear. It would be reasonable to expect that, given the Windows Azure's huge resources, all machines will start within similar time frame. However, step 2 of the test - "request instantiation of i machines" is not atomic from perspective of Azure's REST API. In fact, there is a separate request issued to add each of the VM's. The problem is that Azure does not allow for more than one operation on the given deployment at the same time. In this case, this means that request for adding another machine will be getting rejected until the currently requested machine goes into "Starting" state and that's exactly what takes those 45 seconds. The overall result is that a set of 20 machines will be instantiated in about 1096 seconds (above 18 minutes) instead of probably about 235 seconds if simultaneous operations on deployment were allowed.

The second Fig. 6.2 shows instantiation times when the given number of VMs were requested. When compared with 6.1 it is clearly visible that adding n machines takes approximately the same amount of

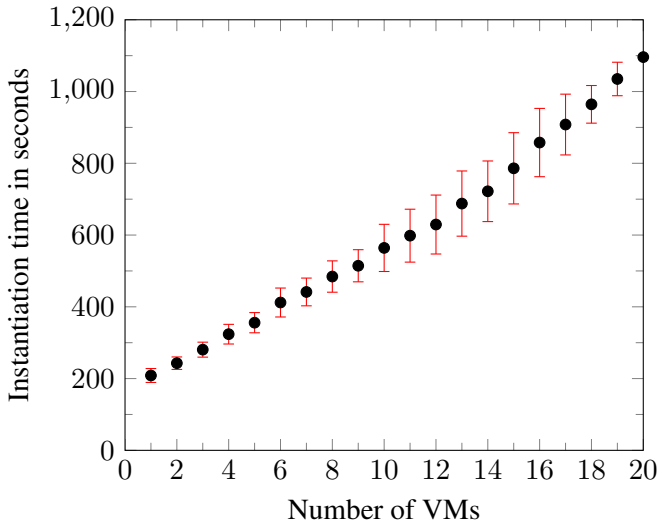

Figure 6.2: Average instantiation times when requesting 1 to 20 Instance VMs from 127 GB Windows Server 2012 image on West Europe Region

time whether n or more machines are requested.

The last plot 6.3 shows how long it takes to completely remove n VMs. Completely means that they no longer generate any cost to the user. This involves removing machines and deleting their disks from storage. While these times are generally increasing with number of VMs, it's not as smooth, consistent or significant growth as in the case of instantiation. I takes about 156 seconds to take down all machines, on average (with standard deviation around 25 seconds).

Problems

Unfortunately all the presented data comes from a single set of results. It was impossible to repeat tests and get more credible data because of severe problems with the described procedure of scaling. Very often one or more machines got stuck in "Running (Provisioning)" state. As a consequence worker couldn't send the registration message. It was also impossible to connect to the problematic VM with Remote Desktop. Such issue had been reported by some users in many threads on Microsoft's forums but no working solution was given by Azure's support team so far. Tests repetition possibilities were also limited by available Windows Azure Credits.

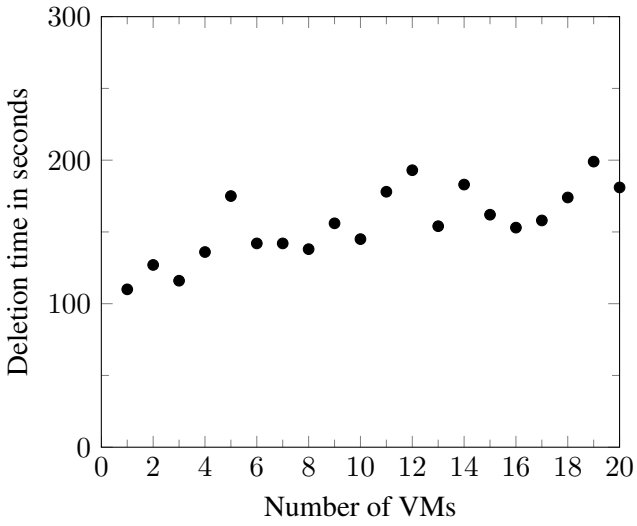

Figure 6.3: Deletion times of all instantiated VMs after requesting 1 to 20 Instance VMs from 127 GB Windows Server 2012 image on West Europe Region

6.2. CPU-intensive application - POV-Ray

Many computational problems are solved using methods which heavily rely on the CPU power. Therefore, it seemed natural to review the Windows Azure's abilities in this area. From magnitude of possible tests we chose rendering a scene in POV-Ray which is a 3D rendering program which uses ray tracing technique to generate images. This choice was made also for another reason. It was meant to be a proof of concept testing how easy it is to employ DTL to distribute workload of existing, popular software without modifying its source.

6.2.1. Why ray tracing?

Ray tracing is a 3D rendering technique based on tracing the path of simulated rays of light shot from the image plane into the scene where they interact with objects. It's the opposite of what happens in nature (rays being emitted from light sources) but it saves a lot of computational power by tracing only rays which form the resulting image. Nevertheless, it's an extremely computationally expensive process. Ray tracing based on idealized, geometric model was proven to be even undecidable in some cases [28], however assuming some less-ideal models it turned out to be feasible even if very costly [2]. Usually, image is composed of a few millions pixels. A separate ray is shot through each and every of them. Achieving higher degree of photorealism requires using more than one ray per pixel. Additional rays are generated upon hitting an object and are used to create refraction, reflections, shadows and other effects. This can easily lead to hundreds of rays per pixel which results in massive amounts of computations needed. Nevertheless, ray tracing is often a rendering method of choice because the resulting images can have photographic quality as seen in Fig. 6.4

Luckily, ray tracing is an embarrassingly parallel problem in the sense that each ray may be treated independently from the others. It does not mean that distributed ray tracing poses no challenges. One problem, for example, is wasteful data duplication in scenarios with large scenes [3]. Anyway, ray tracing's general parallelizability allows for splitting the workload into many computational nodes, processor cores, GPU stream processors [24] or even dedicated ray tracing hardware [32].

Sole reliance on CPU power and intrinsic parallelizability were reason of choosing ray tracing for testing DTL and Azure.

6.2.2. Why POV-Ray?

POV-Ray is a free and open-source ray tracing program which is also very often used for benchmarking. It even comes with an embedded benchmark scene (see Fig. 6.5 which is complex enough to tax virtually any CPU. POV-Ray can be run in interactive mode, where user may create or edit existing scene and render it or in batch mode where all rendering parameters are passed as program parameters at startup. Both modes, however, create a window, thus making it impossible to run POV-Ray in a headless fashion. This created some problems during testing (described later in the text). Scene to be rendered is described

Figure 6.4: "Glasses, pitcher, ashtray and dice" by Gilles Tran rendered in POV-Ray shows ray tracing photorealism [30]

using Scene Description Language [21]. Thanks to its basic programming constructs like loops or functions it is fairly easy to describe a complex scene in a concise manner (e.g. benchmark scene file size is about 25 kilobytes). The resulting image may be saved in a few image formats. The default is PNG which proved to be both easily manageable from C# and compact enough to be transferred quickly.

When it comes to parallelization, POV-Ray provides a solution out of the box. First of all, since version 3.7 it makes use of multiple CPU cores, however that's not relevant to our scenario. Secondly, image area can be split into arbitrary number of rectangular subregions using command line parameters. This allows for generating parts of the picture on multiple machines and later assembling them into final image just by placing them one next to another.

Summing up, here are the key reasons for choosing the ray tracing with POV-Ray as a test case:

- CPU-intensive
- naturally parallelizable
- small size of input data
- relatively small size of output data
- widely used in many benchmarks

6.2.3. Environment

Tests were performed using Small Instance Windows Azure VMs. Machines were created from Windows Server 2012 R2 image containing preinstalled DTL Worker Service.

Figure 6.5: POV-Ray benchmark scene [20]

6.2.4. POV-Ray setup

POV-Ray 3.7 was used to perform the following tests. As far as Windows binaries are concerned, it is distributed in an interactive installer. It wasn't, therefore, suitable for installation via DTL Worker Service. Fortunately, given access to already installed POV-Ray, it is possible to manually copy a few files (cmedit64.dll, povcmax64.dll, pvengine64.exe) and run pvengine64.exe with /QINSTALL switch to perform silent, non-standard installation [33]. Additionally it was necessary to copy ini and include directories from My Documents POV-Ray location to the respective directory on the destination machine. The whole process is less than ideal and it would be easier if POV-Ray could be launched from standalone binaries but unfortunately that's not possible.

6.2.5. Test procedure

Tests were performed in four stages - using 1, 2, 4 and 16 Windows Azure VMs. VMs were already instantiated and started before the test. Each stage consisted of five cases - image was divided into 1, 2, 4, 8 and 16 equally-sized segments. Rendering each of these segments was treated as a separate DTL task. Each case was tested three times to get more credible results. Entire image resolution, which directly affects ray tracing time, was chosen experimentally to be small enough to make tests reasonably short but large enough to make execution times conveniently measurable. Resolution of 128 x 128 pixels turned out to meet these criteria given the performance of Windows Azure's Small Instance VM.

As it was described earlier (see section 5.2.1), there is a concept of common data and input data in DTL. In this case common data was a directory containing POV-Ray binaries and configuration files zipped into 3.29 MB archive. Input data consisted solely of command line parameters which were used to render the given part of the image. After serialization they took 24.9 KB of storage space for each segment. Final image had 42.2 KB.

6.2.6. Problems solved

We encountered several problems while setting up POV-Ray tests.

First of all, as mentioned before, POV-Ray is a desktop application and creates a window even when run in batch mode. The problem is that DTL embedden in the Azure VM image runs as a service using Local System account. This account has no desktop session associated, therefore POV-Ray's window can't be displayed which causes the program to hang indefinitely. Even checking "Allow service to interact with desktop" didn't help. Fortunately, there is and option to run service under a custom user account and this approach worked. However, it required modification of DTL service installer but thanks to the DTL's auto-update feature it didn't pose much of a problem.

Second obstacle was the POV-Ray's license agreement window which appeared upon first launch of program, effectively stopping test execution. First idea was to send ENTER key signal to this window to close it. For unknown reasons this solution worked rather erratically and forced us to look for a different one. Tracing the changes made by POV-Ray during start-up revealed that the quite enigmatic setting named "ItsAboutTime" from "General" section of pvengine.ini holds an Epoch time value of the next license agreement window appearance. Setting it to a distant future solved the problem once and for all.

6.2.7. Measurements

We decided to measure not only scene rendering times for scenarios presented in 6.2.4, but also other times to show more aspects of parallelizing POV-Ray with DTL. Time of twelve operations were measured:

1. Uploading POV-Ray archive to blob storage
2. Uploading rendering parameters for each segment to blob storage
3. Sending task message
4. Receiving task message
5. Downloading POV-Ray archive from blob storage (only once, then cached on disk)
6. Downloading rendering parameters for each segment from blob storage
7. Installing POV-Ray if not installed
8. Running POV-Ray

9. Uploading image segment result to blob storage
10. Sending result message
11. Receiving result message
12. Downloading result image

6.2.8. Results

Overall times and speedup

Let's start with the most basic analysis - how adding more workers affects overall times? Overall time is a time span between moment when user issues rendering request and moment when entire image is displayed on the screen. This data will later allow us to compare distributed execution with normal, one-machine POV-Ray run and show the amount of overhead created by DTL.

Figures 1.5-1.8 show the relationship of overall time to number of segments into which the image was divided. Data points show average times while error bars indicate standard deviation. If execution via DTL imposed no overhead and POV-Ray started up instantly, then in case of n workers, given that entire image is rendered in t_1 seconds, we should get rendering times as a simple function of number of segments as shown in Fig. 6.6.

$$t(s) = \begin{cases} \frac{t_1}{s} & if\ s < n \\ \frac{t_1}{n} & if\ s \geq n \end{cases}, s \in \mathbb{N}^+$$

Figure 6.6: Ideal case rendering time to number of segments function formula

In case of 4 workers it would give a plot presented in Fig. 6.7
Fig. 6.12 combines data from all plots so it's easier to see relations between different numbers of workers, segments and time.
Ideal scenario function formula 6.6 represents a monotonically decreasing function, however the real world results show that this may be the case only for infinite number of workers. Obviously there is some execution overhead and it is especially pronounced when number of workers is low. It you have only one worker, there is no point in using DTL, exactly as intuition suggests, because dividing into more and more segments means that there will be more tasks generated, more data transferred and, in the end, the whole process will take more time. In case of one worker, dividing image into 16 segments nearly doubles rendering time. Plots for greater number of workers show that we indeed have a speedup when there are segments but only until these numbers are equal. Beyond that point, the overall rendering time will only increase. The conclusion is consistent with intuition - the optimal number of segments is equal to the number of workers.
Fig. 6.13 shows the same data as Fig. 6.12 but in the form of a speedup graph. It's clearly visible at which point adding more workers doesn't reduce rendering times but can even increase them - as it is in the case

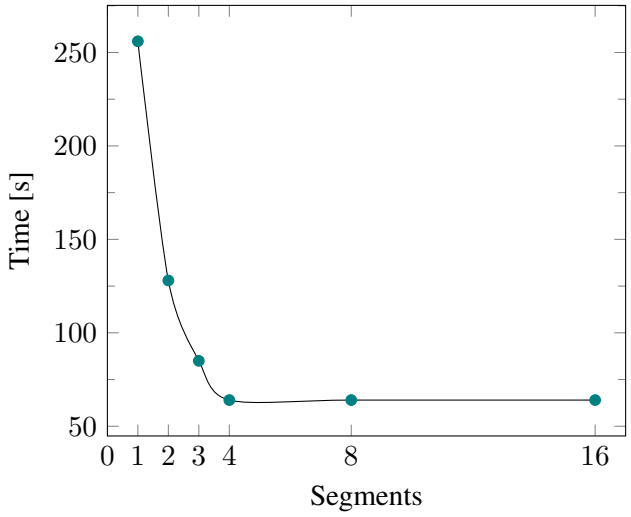

Figure 6.7: Rendering times for 4 Small Instance workers in the ideal case

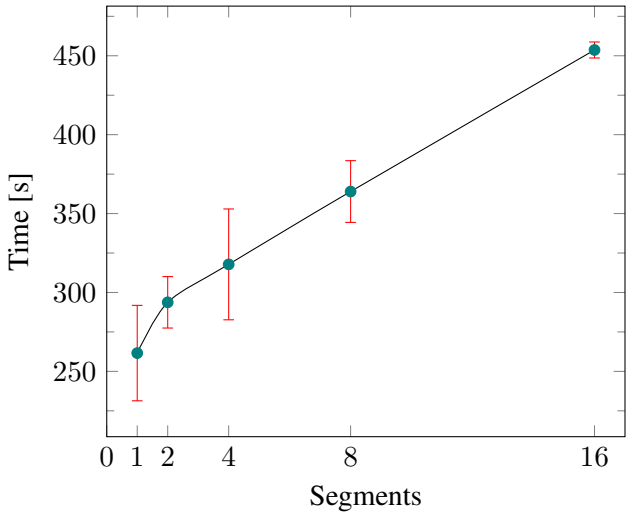

Figure 6.8: One Small Instance workerrendering times

of 2 segments and transition from 4 to 16 workers. Another observation is that while 2 workers perform almost 2 times better than 1, then even if we increase number of segments and workers proportionally we don't get proportional speedup. This is the more visible, the more we increase the numbers, e.g. for the largest values in our scenario - 16 workers and 16 segment we don't get even a sevenfold speedup. One reason behind this may be fairly constant time of POV-Ray startup and shutdown (i.e. time when process is running but nothing is being rendered). There was no straightforward way to measure it during tests so it was incorporated into overall POV-Ray running time.

Overall times decomposition

As we've mentioned in section 6.3.5, the overall times are composed of a few parts. Here, we'll present typical contribution of each part. Fig. 6.14 shows the time composition of the average run. It presents

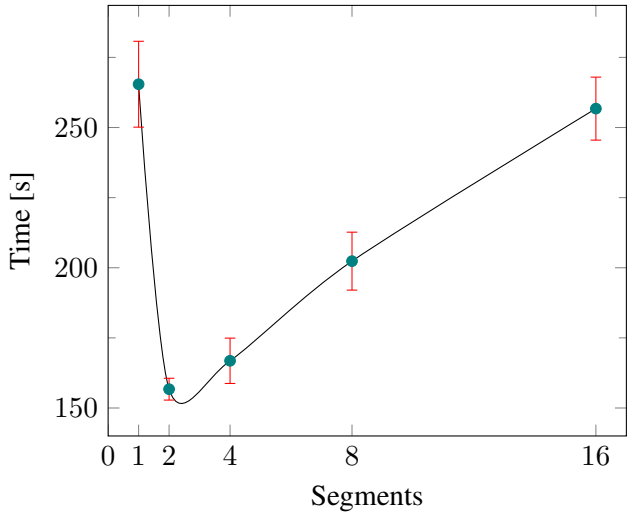

Figure 6.9: Two Small Instance workersrendering times

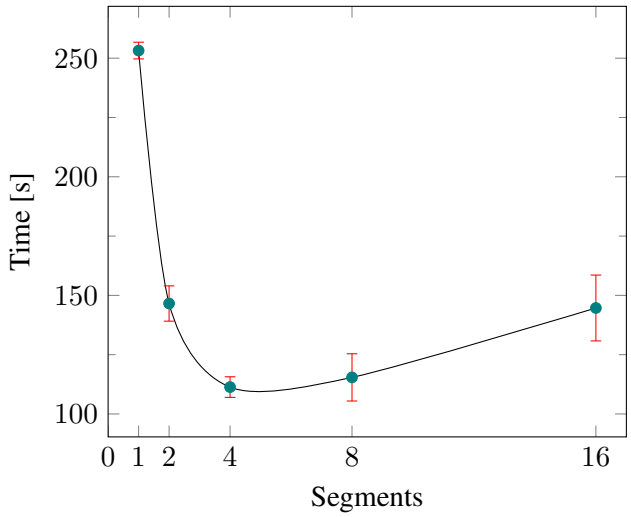

Figure 6.10: Four Small Instance workersrendering times

only two components - POV-Ray execution time and other time. As you can see POV-Ray execution is so lengthy that it would be pointless to divide the "other" section into more components. Rendering takes 97% of the whole run. This is great illustration of the purpose of the whole test - most of the test time is spent on CPU-heavy ray tracing.

The second graph 6.15 presents detailed composition of the "other" part from the first graph. Here, the most of the time (5.19 s) is spent on uploading common data, i.e. POV-Ray package weighing 3.29 MB uploaded via connection declared by ISP as "up to 768 kB/s (or 6 Mb/s)". The same data is downloaded to VMs in about 0.33 s which translates into around 10 MB/s (or 80Mb/s) download speed. Of course if we wanted to do real Azure bandwidth tests we would've to send much larger data to get more accurate and credible results. Sending and receiving queue messages is fast due to they small size. Input message contains more data - not only links to stored blobs but also embedded serialized parameters objects

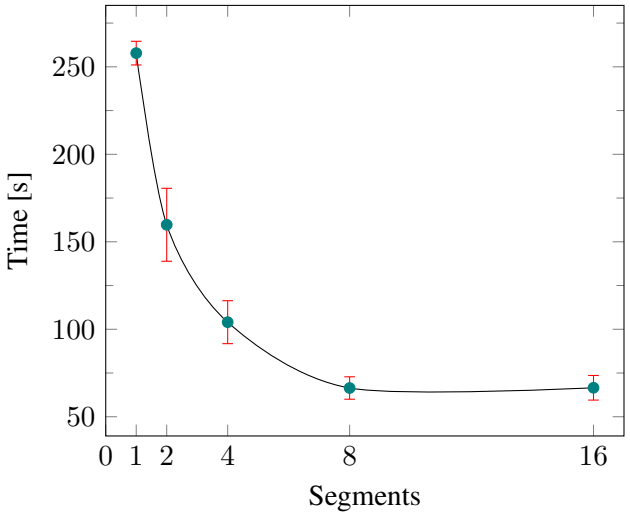

Figure 6.11: Sixteen Small Instance workersrendering times

and list of required assemblies. Installation of POV-Ray, which takes 0.22 seconds on average, is also negligible in comparison to the duration of rendering process.

6.2.9. Summary

The performed tests show that it is possible to use DTL to run 3rd party application in a distributed manner (provided it has built-in support for splitting its workload). Moreover, it doesn't require much effort in such a simple scenario. We encountered some minor problems but all in all we achieved out goal. It turned out that Windows Azure copes quite well with CPU-intensive tasks. We didn't encounter any unexpected problems with Azure and connection to blob and queue storages was sufficiently fast and reliable for the test's purposes. Due to time and resources limitations we managed to perform tests for Small Instance only, but in future it could be interesting to run these or similar scenarios on the whole range of Azure's VMs.

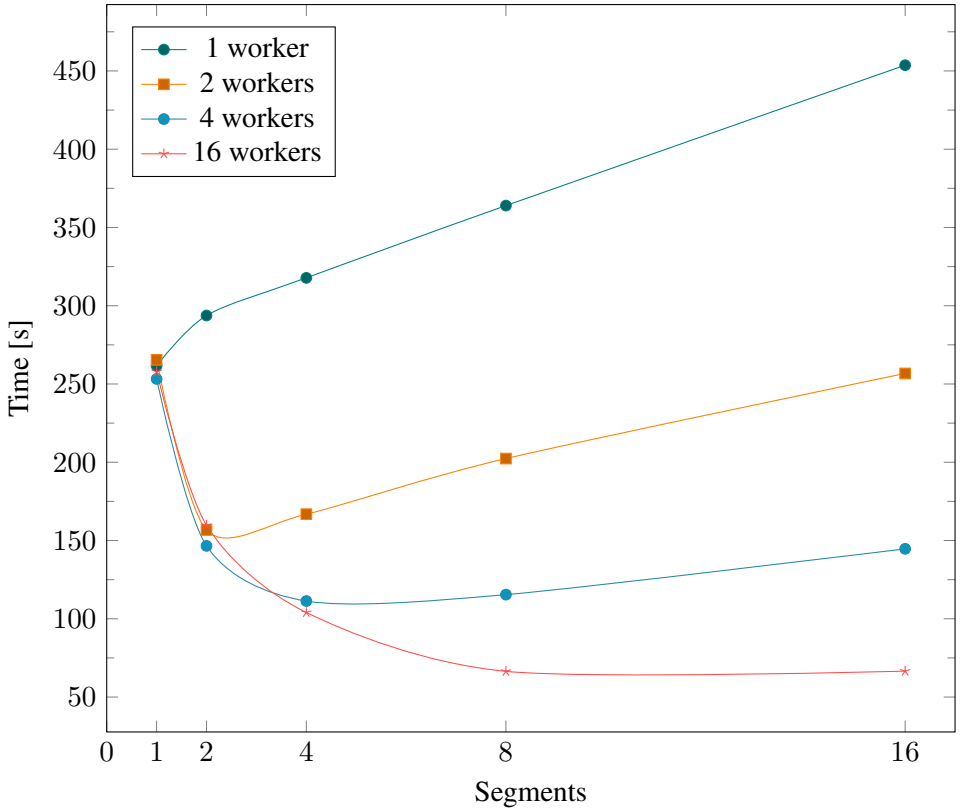

Figure 6.12: Rendering times for 1, 2, 4 and 16 Small Instance workers combined

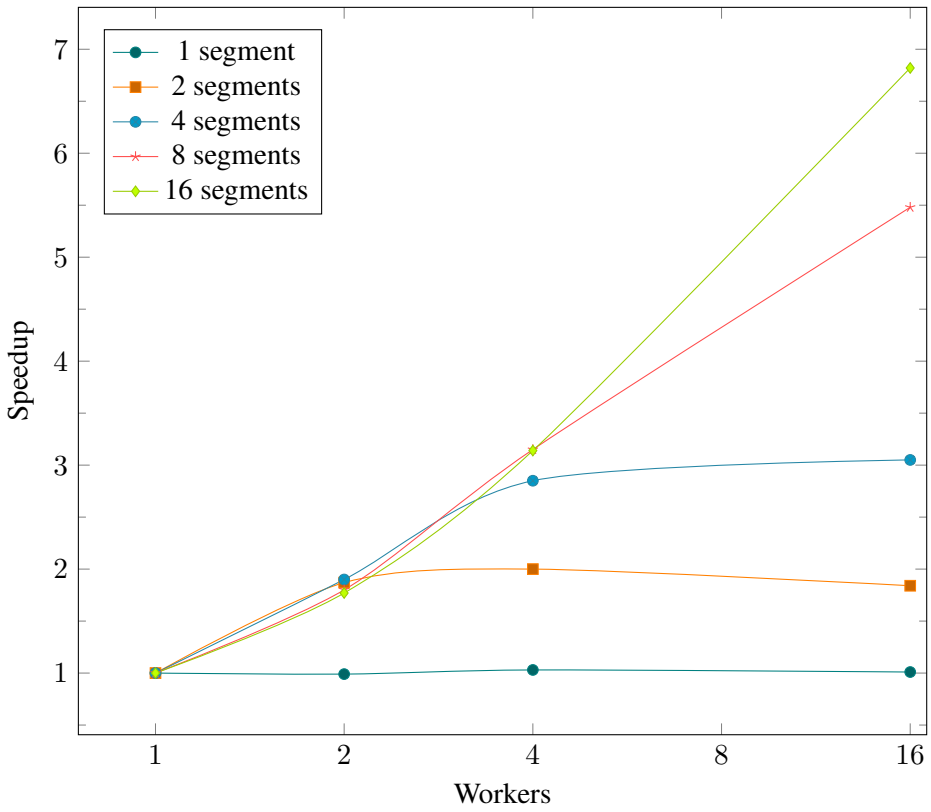

Figure 6.13: Rendering speedup for 1, 2, 4 and 16 Small Instance workers

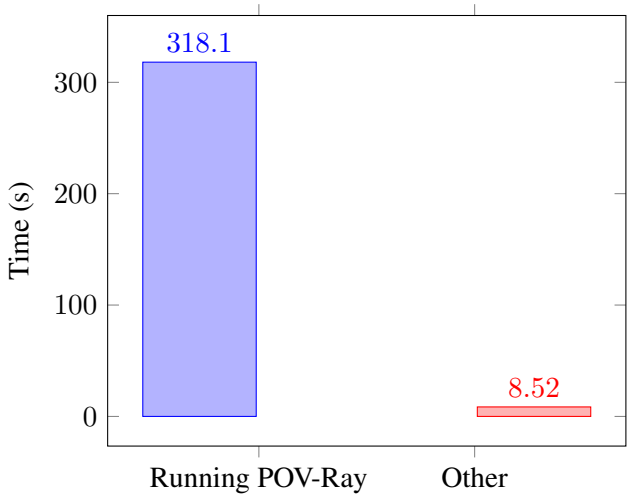

Figure 6.14: Decomposition of overall execution time showing POV-Ray rendering time dominance

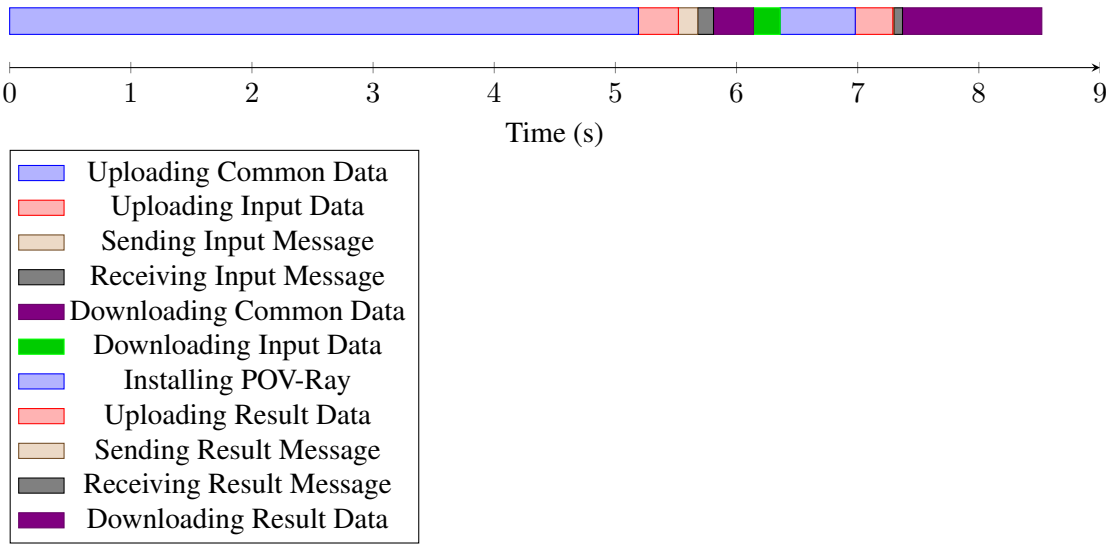

Figure 6.15: Decomposition of "Other" time section from Fig. 6.14

6.3. Real-world bioinformatics application - ExonVisualizer

Real-world application is the final test for every solution. Therefore we decided to take DTL and plug it into an already existing application. We wanted something written in C# with available source code, so we could utilize DTL exactly as it was intended to be used. At the same time we hoped to evaluate Windows Azure as a platform for solving the type of problems the application solved.

6.3.1. ExonVisualizer

Choosing suitable application proved to be effortless. Before we started considering possible options, we were approached by Dr. Monika Piwowar from Department of Bioinformatics and Telemedicine at Jagiellonian University Medical College. She had an application which ideally fit our aforementioned requirements. It was created as a part of master thesis by Krzysztof Porembski under Dr. Piwowar's supervision [27].

Overview

The application is called ExonVisualizer and, as its author writes, is dedicated to everyone (e.g computer scientists, biotechnologists, biologists) who is interested in visualization of protein regions (2D and 3D structures) encoded by particular exons [26]. It allows "tracking" gene expression by identification parts of the proteins being in relation with gene structure (coding regions - exons) [25]. It can be good tool for analysis of proteins that are synthesized in the alternative splicing process and also in identification protein regions created in the post-translational modifications process. It uses 3rd party BLAST 2 Sequences (bl2seq) application which is a part of BLAST [8] suite and is used to find matching protein sequences. ExonVisualizer is a web application written in C# using ASP .NET and as such is intended to be used through web browser where user can provide input data and analyze output which is generated on a web server.

Input

Input data consists of following file pairs:

- GenBank Flat File (.gb extension) - contains nucleotide data; can be obtained from GenBank database [9]
- PDB file (.pdb extension) - Protein Data Bank file, provides a standard representation for macromolecular structure data derived from X-ray diffraction and nuclear magnetic resonance studies. [10].

There two operation modes:

- Single task - user provides a single GenBank file and then a single, matching PDB file

- Batch task - user provides a zip archive containing arbitrary number of GenBank and PDB file pairs, matching files must have identical filenames

Finally, user must specify alignment properties:

- PDB domain (only for single task) - user can choose to perform analysis for one or more domains available in the PDB file
- Gap cost - cost of creating gap in the sequence, it actually consists of two values: GapOpenCost and GapExtendCost
- Substitution matrix - describes the rate at which one character in a sequence changes to other character states over time

Output

Application output consists of identified and visualised exons found in the provided protein structure. Results are available in three formats:

- HTML - default format, presented in web browser. It contains symbols and names of amino acids existing in each of chosen PDB domains, numbered according to the notation in the PDB file. Moreover, symbols and names of secondary structures are included along with their location in the amino acid sequence and some statistical data.
- Text file - the minimal, most crucial subset of information from HTML format
- PyMol file - script for PyMol software, which allows to visualize the result in the graphical form
- VMD file - same as PyMol but dedicated to be used by VMD software

6.3.2. Goal

The goal was to add distributed processing to ExonVisualizer and test it afterwards. We decided to augment batch mode only, because single task mode was very fast (input to output time was mostly less than one second) while batch mode could take much more time as the archive can contain arbitrary number of files.

6.3.3. Adaptation for DTL

Right after we obtained the source code of ExonVisualizer, we started its analysis and decided to apply a few modifications before adding DTL support. First of all, the core part of application which governed reading data, processing and aggregating results was tightly coupled with web UI and ASP .NET-related code in one, large project called Expression. We took this core part and extracted it to a separate project - Expression.Core. Thanks to this, it was possible to create another project, a console application for

the purposes of automated tests which are described later in this chapter. Afterwards, we updated all referenced libraries to the latest versions and .NET Framework to version 4.5 to ensure high level of compatibility with DTL. Then it was necessary to track down parts of execution path eligible for being executed in a distributed manner. It turned out that parsing the input files and BLAST execution take the most time. In the original code parsing was done twice unnecessarily, so we optimized that. In the end we decided that DTL command would accept the original user input data, i.e. two files and alignment parameters and would return object containing result data. This meant that we had to extract the input zip file, find all matching file pairs, pack them again and send each pair as a single input item. Therefore, each pair could be processed in parallel on different nodes. After making necessary changes, ExonVisualizer web application began successfully processing data using DTL and Windows Azure.

6.3.4. Test procedure

We wanted to test how well will ExonVisualizer and DTL work with various numbers of workers. A console application written in C# was used to drive the test process. We decided to check scenarios with 1, 2, 4, 8 and 16 Windows Azure Small Instance VMs. Each such machine was instantiated from Windows Server 2012 R2 image with preinstalled DTL worker. Test data was provided by Dr. Monika Piwowar - 64 pairs of GenBank and PDB files, zipped into an 6.7 MB archive. As it was described earlier (see section 5.2.1), there is a concept of common data and input data in DTL. Common data was a 1.45 MB file with zipped bl2seq executable file. Each piece of input data consisted of zipped, matching GenBank and PDB file pairs. Their sizes ranged from 25 KB to around 1 MB. Alignment parameters were left at default: GapOpenCost = 11, GapExtendCost = 1, SubstitutionMatrix = Blosum62. Each scenario was executed three times.

6.3.5. Measurements

Apart from measuring the overall execution time there were several other time measurement points defined:

1. Uploading required assemblies from blob storage (636 KB in total)
2. Uploading bl2seq archive to blob storage
3. Uploading GenBank file, PDB file and alignment parameters for each file pair to blob storage
4. Sending task message
5. Receiving task message
6. Downloading bl2seq archive from blob storage (only once, then cached on disk)
7. Downloading GenBank file, PDB file and alignment parameters for each file pair to blob storage
8. Downloading required assemblies (only once, then cached on disk)

9. Running Command
    (a) Extracting files
    (b) Parsing GenBank file
    (c) Parsing PDB file
    (d) Executing bl2seq
10. Uploading result to blob storage
11. Sending result message
12. Receiving result message
13. Downloading result from blob storage

Their purpose was to give a deeper insight into execution process which could help localizing potential bottlenecks. Some of these measurements are part of the others, e.g. "Executing bl2seq" is a part of "Processing Files" which in turn is a part of "Running Command".

6.3.6. Results

Overall times and speedup

The first thing to analyze is overall execution time. Fig. 6.16 shows overall execution time in relation to number of workers instantiated.

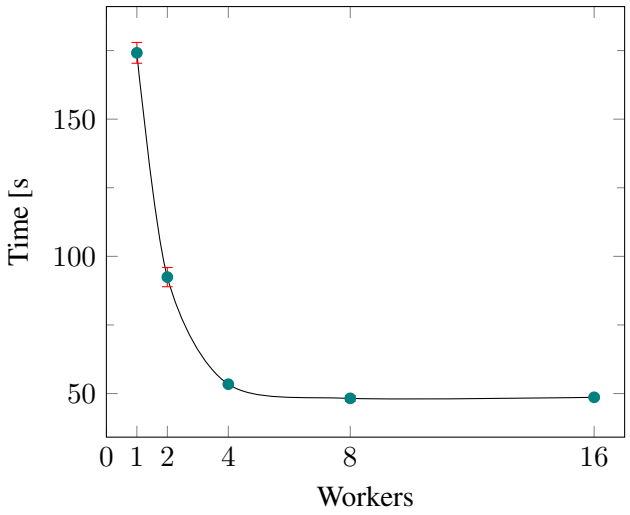

Figure 6.16: ExonVisualizer overall execution times for 1, 2, 4, 8 and 16 Small Instance workers

If there is only one worker available then overall execution takes around 174 seconds, which is slightly under 3 minutes. Adding another worker cuts it down approximately to half that value. Doubling the number of nodes to 4 results in further, but less spectacular decrease - to around 53.41 s. Distributing the

load over 8 workers results in just a slight speedup - 48.25 s. Adding another 8 VMs has no, or even a negative effect - 16 workers completed the job in 48.63 s. As you can notice, the error bars are almost invisible, because each scenario repetition took nearly the same amount of time.
Fig. 6.17 presents the results on a classic speedup graph, which shows the parallelization limit even more clearly than the previous one. There is no point in instantiating more than 8 workers, because above that number the overhead gets so big that we get not only no speedup but even a slowdown.

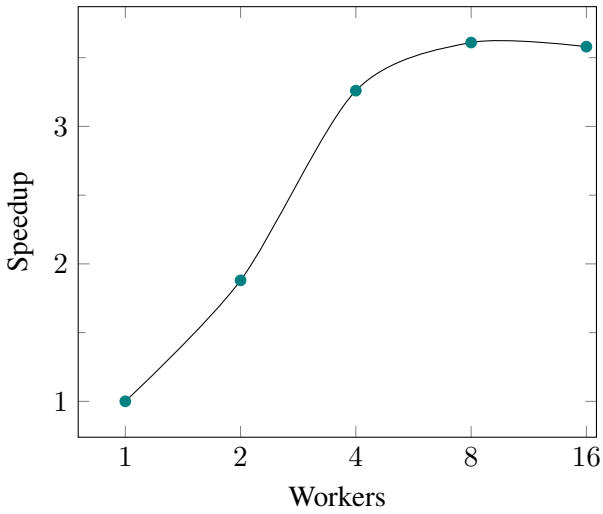

Figure 6.17: ExonVisualizer speedup for 1, 2, 4, 8 and 16 Small Instance workers

Overall times decomposition

There is a number of operations which must be performed during the whole execution. We listed them in section 6.3.5. In this section we analyze how much time each of these operations take in scenario with one machine, for the sake of simplicity. Fig. 6.18 shows the times for a simplified case, in which we have only one pair of files. Over half of the time is taken by uploading data from the client machine to Windows Azure Blob Storage. The second longest operation is downloading this data from storage on the VM. Third, but most productive, is the actual execution of DTL command.
Fig. 6.19 presents breakdown of the "Executing Command" part. Most of the time is spent on extracting archives with input data. GenBank file parsing takes neglible amount of time. Parsing PDB file is longer, but still very fast. Lastly, execution of bl2seq requires around 0.37 seconds.
Things change a bit when we analyze the case with the full, 64-piece set of input data. Uploading and downloading common data and assemblies no longer dominate, because they are done only once for the whole process. Significant amount of the time is spent on uploading, downloading the input data and executing the DTL command.

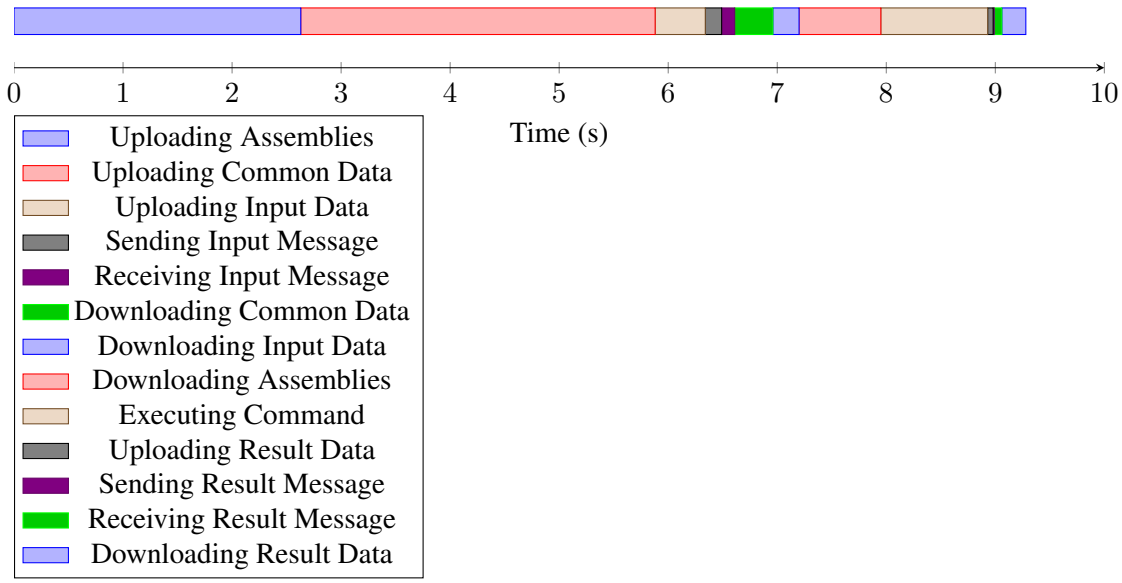

Figure 6.18: Decomposition of overall ExonVisualizer execution time for an average file pair

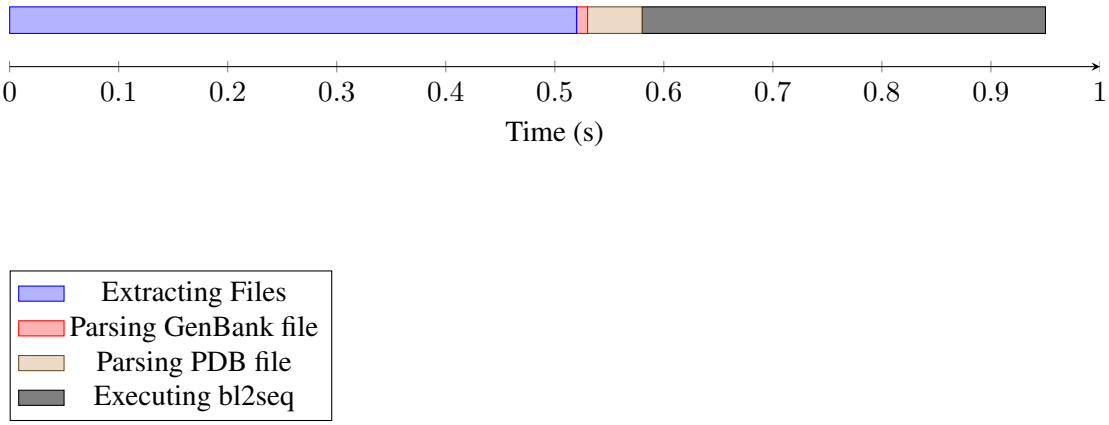

Figure 6.19: Decomposition of "Executing Command" time section from Fig. 6.18

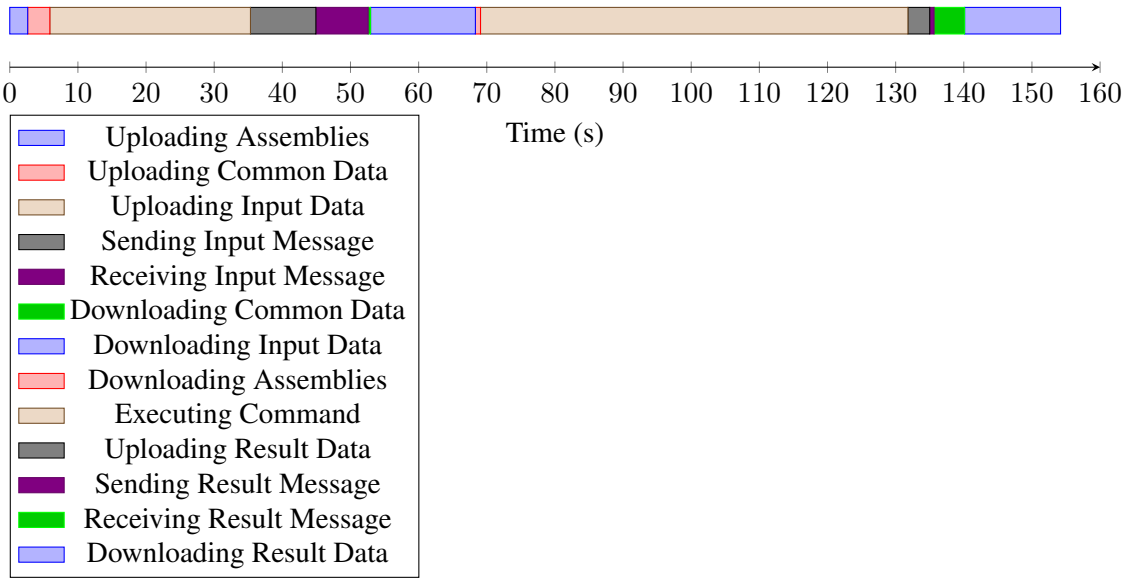

Figure 6.20: Decomposition of overall ExonVisualizer execution time for 64 file pairs

6.3.7. Summary

Implementation of DTL into ExonVisualizer proved to be feasible and resulted in a fully-functional solution. Presented graphs show that distributing the workload shortens the overall execution time considerably. However, we must realize that employing DTL and Windows Azure for ExonVisualizer computations yielded virtually no gain from user's perspective. Quick look at Fig. 6.20 shows that command execution took around 60 seconds. If ExonVisualizer was launched without DTL it would take just this amount of time to process the test data (or even less considering there would be just one big archive to unpack). Even at the peak of our speedup graph, with 8 workers we get total execution time of about 50 seconds. It's negligible difference considering the fact that one would need to prepare and start the workers and pay for their running time. It brings us to conclusion that this type of a problem is a bad use case for employing distributed execution, at least the way we did it. The main problem here is that the actual processing takes very little time while transmitting files over the network and other communication is relatively slow.

There are, however, a few other ways to potentially speed up ExonVisualizer. First of all, processing could be done on multiple cores in parallel. This should result in almost no overhead as no network communication would occur. Second idea is to send not single file pairs but entire batch archive as one DTL task. It should work well in scenarios with multiple users requesting batch processing simultaneously. Nevertheless, even right now multiple users can use the modified application and with small effort it may be further enhanced to handle dynamically changing number of users by adjusting the number of workers in runtime. All the necessary implementation is ready in DTL, it just needs to be employed in ExonVisualizer.

7. Conclusions and future work

This last chapter summarizes our entire work and presents our conclusions. Its second part presents our ideas for further research directions and possible useful additions to Distributed Task Library

7.1. Conclusions

The main goal of this thesis was to **evaluate suitability of Microsoft Azure as a computational applications execution platform**. In order to attain it, we created a .NET **Distributed Task Library**. Thanks to it, we could quite easily adapt tested applications execution in a distributed manner on Microsoft Azure. DTL was created with intention to not only be used in this thesis, but also to serve as a tool for other users. We successfully met this and other design requirements (see section 1.2.1).
We evaluated Microsoft Azure by completing our intermediate objectives - a series of tests described in detail, along with their results, in chapter 6. We came to the following conclusions regarding those three scenarios:

- **Dynamic horizontal scaling**, particularly its low speed, may be problematic for some applications. Adding a new machine takes nearly 4 minutes on average, which may be too long for scenarios where rapid response to increased load is crucial. Scientific applications, however, often work in batch mode and their demands may be better predicted beforehand.

- Testing the **CPU-intensive application - POV-Ray** proved that ray tracing scales very well with increasing number of workers. It also showed that using DTL, it is not difficult to take existing, proprietary software and execute it on many Microsoft Azure virtual machines.

- Finally, we managed to modify **real-world bioinformatics application - ExonVisualizer**, and test it on Microsoft Azure platform. Unfortunately we didn't observe significant speedup. However, we proposed a different approach to the problem which may yield better results in section 6.3.7

All in all, Microsoft Azure is definitely a worthy platform for computational applications. However, each application have different requirements and therefore in each case Azure must be analyzed for both helpful features and blocking obstacles.

7.2. Future Work

7.2.1. Computational Applications on Microsoft Azure

We analyzed only a few aspects of Microsoft Azure which are important when deciding whether use it for deploying computational applications or not. Microsoft Azure provides quite large number of features so we list only those ideas which closely related to tests performed in this theses.

Linux-based tests

Microsoft Azure offers a number of Linux images which may be even more attractive to scientific community than Windows ones. It could be interesting to perform instantiation speed tests on them or check how Linux-exclusive applications behave in such environment. Moreover, it would be great to test DTL on Linux more extensively.

Testing other applications

There are many applications and benchmark which may be helpful in assessing Microsoft Azure capabilities. Getting a more comprehensive image of Azure's performance would require testing at least a few more than those present in this thesis.

Microsoft Azure Autoscale

As mentioned in section 2.2.2, Autoscale feature has been added to Azure VMs in the meantime. It may be useful in some scenarios and simpler than custom scaling solutions like the one implemented in DTL. Therefore, it seems interesting to test it and compare with this thesis findings.

Adaption of tests to create a direct comparison with other clouds

While the results presented in Evaluation section have value on their own, it could be useful to perform a direct comparison between multiple cloud platforms. It's most likely possible to adapt our test methods to other clouds so one can take our data and compare with results from other platforms like e.g. Amazon EC2 and S3.

7.2.2. Distributed Task Library

Distributed Task Library was created for purposes of this thesis and as such it has the minimal capabilities which allowed achieving our goals. There are a few areas which should be improved before the solution is ready for general use.

Execution reliability

Currently, there is no way to detect failed tasks. Controller waits for all tasks to complete but if execution fails e.g. because of worker crash or power failure then it will wait infinitely. The simplest approach would be to implement a kind of heartbeat sent by worker signaling that task is being still executed and also some timeout for the whole execution in case the task hangs for some reason. Then, the failed task should be retried. Of course some better and more advanced solution may be introduced.

Execution environment improvements - tasks sandboxing and application domain reuse

Ideally, task should be execute in a kind of a sandbox, so that they cannot touch any files outside of their dedicated, working directory nor affect the system in any other way which could compromise its security. Sandboxing is a complicated concept on its own so it could a be a great material for further work. Right now, task are executed in separate application domains which somewhat increases stability of entire solution but also creates an overhead because creation of appdomain, loading assemblies and then unloading it all takes time. It could be optimized in such a way that tasks which belong to the same job (so they use the same assemblies) reuse the same application domain.

Parallel tasks on one machine

Running multiple workers on one machine can lead to potential problems with the current implementation. There are some directories created in user's temp folder that may get corrupted if access simultaneously by two workers. Additionally, worker is identified by machine's hostname so Workers Manager would be unable to differentiate between multiple workers. This can be solved by adjusting implementation so either more than one worker process is safe to run or allow multiple simultaneous tasks to be run at once within the same worker process. The first approach seems simpler and more crash-resistant.

Bibliography

[1] Maarten Balliauw. Tales from the trenches: resizing a windows azure virtual disk the smooth way. http://blog.maartenballiauw.be/post/2013/01/07/Tales-from-the-trenches-resizing-a-Windows-Azure-virtual-disk-the-smooth-w aspx, January 2013. Accessed: 2014-08-31.

[2] Ed Blakey. Ray tracing - computing the incomputable? In *Proceedings 8th International Workshop on Developments in Computational Models*, pages 32–40, April 2014.

[3] Alan Chalmers, Erik Reinhard, and Tim Davis. *Practical Parallel Rendering*. CRC Press, June 2002.

[4] Microsoft Corporation. Introducing azure. http://azure.microsoft.com/en-us/documentation/articles/fundamentals-introduction-to-azure. Accessed: 2014-07-30.

[5] Microsoft Corporation. Virtual machines pricing details. http://azure.microsoft.com/en-us/pricing/details/virtual-machines/. Accessed: 2014-07-31.

[6] Microsoft Corporation. Windows azure hpc scheduler. http://msdn.microsoft.com/en-us/library/hh560247%28v=vs.85%29.aspx, April 2012. Accessed: 2014-09-07.

[7] Microsoft Corporation. Authentication for the azure storage services. http://msdn.microsoft.com/en-us/library/azure/dd179428.aspx, May 2014. Accessed: 2014-08-31.

[8] National Center for Biotechnology Information (NCBI). Blast: Basic local alignment search tool. http://blast.ncbi.nlm.nih.gov/Blast.cgi. Accessed: 2014-07-18.

[9] National Center for Biotechnology Information (NCBI). Genbank. http://www.ncbi.nlm.nih.gov/genbank/. Accessed: 2014-07-17.

[10] Research Collaboratory for Structural Bioinformatics (RCSB). Pdb file format. http://www.rcsb.org/pdb/static.do?p=file_formats/pdb/index.html. Accessed: 2014-07-16.

[11] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphre. Early observations on the performance of windows azure. In *HPDC '10 Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 367–376. ACM New York, NY, USA ©2010, June 2010.

[12] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 227–242. Microsoft Research, ACM New York, NY, USA ©2009, October 2009.

[13] Lydia Leong, Douglas Toombs, Bob Gill, Gregor Petri, and Tiny Haynes. Magic quadrant for cloud infrastructure as a service. http://www.gartner.com/technology/reprints.do?id=1-1UM941C&ct=140529&st=sb, May 2014. Accessed: 2014-07-29.

[14] Manjrasoft Pty Ltd. Aneka: Enabling .net-based enterprise grid and cloud computing. http://www.manjrasoft.com/products.html. Accessed: 2014-08-07.

[15] Manjrasoft Pty Ltd. Aneka trial customer reference. http://www.manjrasoft.com/GoFront-Aneka-Use_Case.pdf, December 2008. Accessed: 2014-08-09.

[16] Manjrasoft Pty Ltd. *Developing Task Model Applications*, May 2012.

[17] Maciej Malawski, Jan Meizner, Marian Bubak, and Paweł Gepner. Component approach to computational applications on clouds. In *Proceedings of the International Conference on Computational Science, ICCS 2011*, volume 4, page 432–441. Elsevier B.V., May 2011.

[18] Ming Mao and Marty Humphre. A performance study on the vm startup time in the cloud. In *CLOUD '12 Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430. IEEE Computer Society Washington, DC, USA ©2012, June 2012.

[19] National Institute of Standards and Technology. The nist definition of cloud computing. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, September 2013. Accessed: 2014-07-26.

[20] Persistence of Vision Raytracer Pty. Ltd. Benchmarking with pov-ray. http://www.povray.org/download/benchmark.php. Accessed: 2014-07-26.

[21] Persistence of Vision Raytracer Pty. Ltd. Pov-ray: Documentation: 2.2 scene description language. http://www.povray.org/documentation/view/3.6.0/224/. Accessed: 2014-07-26.

[22] Aung Oo, Jai Haridas, Pradnya Khadapkar, and Brad Calder. Announcing microsoft azure import/-export service ga. http://blogs.msdn.com/b/windowsazurestorage/archive/

2014/05/13/announcing-microsoft-azure-import-export-service-ga.aspx, May 2014. Accessed: 2014-08-07.

[23] David Pallmann. Azure storage samples. http://azurestoragesamples.codeplex.com/, February 2011. Accessed: 2014-08-31.

[24] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[25] Monika Piwowar, Mateusz Banach, Leszek Konieczny, and Irena Roterman. Structural role of exon-coded fragment of polypeptide chains in selected enzymes. *Journal of Theoretical Biology*, 337(7):15–23, November 2013.

[26] Monika Piwowar, Krzysztof Porembski, and Piotr Piwowar. Exonvisualiser - application for visualization exon units in 2d and 3d protein structures. *Bioinformation*, 8(25):1280–1282, December 2012.

[27] Krzysztof Porembski. Exons identification and visualization in 2d and 3d protein structures. Master's thesis, Jagiellonian University, 2011.

[28] J. H. Reif, J. D. Tygar, and A. Yoshida. Computability and complexity of ray tracing. *Discrete & Computational Geometry*, 11(1):265–288, December 1994.

[29] Microsoft Open Technologies. Vm depot - find, deploy and share images for windows azure. http://vmdepot.msopentech.com. Accessed: 2014-08-07.

[30] Gilles Tran. Glasses, pitcher, ashtray and dice (pov-ray). http://www.oyonale.com/modeles.php?page=40, 2006. Accessed: 2014-07-25.

[31] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyyae. High-performance cloud computing: A view of scientific applications. In *ISPAN '09 Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 4–16. IEEE Computer Society Washington, DC, USA ©2009, December 2009.

[32] Alexandru Voica. Powervr gr6500: ray tracing is the future... and the future is now. http://blog.imgtec.com/powervr-developers/powervr-gr6500-ray-tracing, March 2014. Accessed: 2014-07-25.

[33] POV-Ray wiki. Documentation:windows section 3 - pov-wiki: Non-standard installations. http://wiki.povray.org/content/Documentation:Windows_Section_3#Non-Standard_Installations. Accessed: 2014-07-26.