Automatic Software Deployment in the Azure Cloud

Jacek Cala, Paul Watson

Automatic Software Deployment in the Azure Cloud

Jacek Cała and Paul Watson

School of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, United Kingdom
{jacek.cala, paul.watson}@newcastle.ac.uk

Abstract. For application providers, cloud computing has the advantage that it reduces the administrative effort required to satisfy processing and storage requirements. However, to simplify the task of building scalable applications, some of the cloud computing platforms impose constraints on the application architecture, its implementation and tools that may be used in development; Microsoft Azure is no exception.
In this paper we show how an existing drug discovery system — Discovery Bus — can benefit from Azure even though none of its components was built in the .Net framework. Using an approach based on the "Deployment and Configuration of Component-based Applications Specfication" (D&C), we were able to assemble and deploy jobs that include different types of process-based tasks. We show how extending D&C deployment models with temporal and spatial constraints provided the flexibility needed to move all the compute-intensive tasks within the Discovery Bus to Azure with no changes to their original code.

1 Introduction

The emergence of cloud execution environments shifts the management and access to computing and storage resources to a new, higher, and more efficient level. Often, however, it requires developers to rebuild or at least substantially re-engineer existing applications to meet new requirements. This is true for Microsoft's Azure cloud that was designed for applications based on the .Net framework, and which supports a specific, queue-based software architecture. For many existing software systems, whose development may have consumed significant resources, it could be prohibitively expensive to have to redesign and reimplement them to fit these constraints. Therefore, the key question for us was whether it was possible for existing software to benefit from new execution environments such as the Azure Cloud without the need for significant, and so expensive, changes.

This paper presents details of the automatic deployment platform we developed for Azure, driven by the need of a chemistry application performing QSAR analysis. Quantitative Structure-Activity Relationship (QSAR) is a method used to mine experimental data for patterns that relate the chemical structure of a drug to its activity. Predicting the properties of new structures requires significant

computing resources and we wanted to exploit the Windows Azure Cloud in order to accelerate this process [7].

2 Motivation and Approach

The Discovery Bus is a multi-agent system that automates QSAR analysis. It implements a competitive workflow architecture that allows the exhaustive exploration of molecular descriptor and model space, automated model validation and continuous updating as new data and methods are made available [3]. Our main goal was to move to Azure as much of the computing-intensive agents as possible to make the most of the parallelization offered by the cloud. However, the key problem we encountered when moving Discovery Bus components to Azure was that none of them were created in the .Net framework. Instead, they were written in the Java, C++, and R languages. Therefore, to be able to use the Azure platform directly we needed a solution that would enable us to run and execute existing, non-.Net software in the cloud. For efficiency, and to reduce the amount of the software stack we need to maintain ourselves, this should ideally be a solution that allows the deployment of only a Discovery Bus agent instead of the whole OS stack including that agent, as in the Infrastructure as a Service (IaaS) approach. Our aim was therefore to build an open and lightweight deployment platform that can support a diverse set of existing Discovery Bus agents running in Azure.

The Windows Azure platform is a combination of processing and storage services. The compute services are divided into two types of nodes — web and worker role nodes. The web role enables the creation of applications based on ASP.NET and WCF and is the entry point for external clients to any Azure-based application. In contrast, the worker role runs applications as independent background processes. To communicate, web and worker roles can use the Azure storage service comprising of queue, table and blob storage. The primary means of communication for Azure-based systems are queues. The queue-based approach aims at maximising scalability because many web role nodes can insert tasks to a queue, while many worker role nodes can acquire tasks from the queue. By simply increasing the number of workers, the tasks remaining in the queue can be processed faster [10].

The proposed anycast-like operation model [1] fits very well problems that do not require much communication apart from a single request-response pattern with no state preserved in the workers. Much of the Discovery Bus processing has this kind of communication style, but to use Azure for QSAR processing, the missing link is the ability to install and execute non-.Net components. Moreover, as these components often have specific prerequisites such as the availability of third-party libraries and a Java runtime environment, a deployment tool must allow more sophisticated dependencies to be expressed.

We based our deployment solution on the "Deployment and Configuration of Component-based Distributed Applications Specification" (D&C) [11]. It defines

the model-based approach to deployment[1] and is built according to the Model Driven Architecture (MDA). D&C specifies a Platform Independent Model (PIM) of deployment which follows the general idea that the deployment process remains the same, independent of the underlying software implementation technology. The PIM can further be customized with Platform Specific Models (PSMs), such as PSM for CCM [12], to address aspects of deployment specific to a particular software technology. The D&C specification defines one of the most complete deployment standards [5], however, it originates from object-oriented and component-based domains, and PSM for CCM is the only existing PIM to PSM mapping. One of our intentions was to determine whether the deployment approach proposed by OMG can be used at a different virtualization level; in our case, a software component is usually an OS executable and a component instance is commonly a process in an operating system. This is in stark contrast to the definition of component proposed in the specification.

One of the key qualities of model-based deployment, which distinguishes it from the script-based and language-based approaches, is a separation between a model of execution environment and model of software (Fig. 1). This improves the reusability of software and environment definitions as the same software description can be used for deployment over different execution environments, and the same environment description can be used for the deployment of different software. This separation is especially important for heterogeneous environments where deployment planning enables the matching of software components to appropriate execution nodes. Previously, Azure offered one type of resource, but recently Microsoft enabled four different computing node sizes in their cloud platform. However, model-based deployment also provides the means to express execution node resources, which is crucial for planning deployments in multi-layer virtualized environments. The deployment of a component at a lower virtualization level enriches the set of software resources, enabling further deployments at higher levels. For example, if a Java runtime environment (JRE) is deployed on a node at the OS level, that node gains the ability to run Java applications. The JRE can be expressed as a node resource, which is then taken into account during planning. These advantages motivate the need for deployment planning, and the model-based approach.

3 Architecture of the Automatic Deployment Solution

In order to make the most of the scalability offered by the Azure cloud, our deployment platform is a queue-based system that allows a web role to request the deployment of jobs on worker nodes. For the purpose of the QSAR use case, jobs represent the code of Discovery Bus agents but our platform is generic and allows potentially any kind of program to be run. Every job is described by a deployment plan that defines what to install and execute. We extended models defined in the D&C specification with some of the concepts proposed in our previous work [2]. A deployment plan can describe multiple tasks to be deployed, each of which may

[1] In contrast to the script-based and language-based approaches [6].

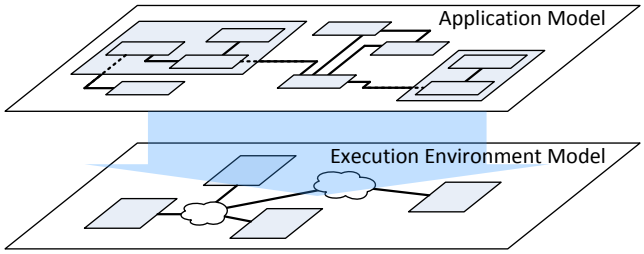

Fig. 1. Model-based deployment separates between software and execution environment models.

be of different type such as Java-based applications, or Python and R scripts. The tasks can be deployed independently, or can be bound with spatial and temporal dependencies. A deployment plan may also include dependencies on other plans that need to be enacted on first, e.g. Java-based applications that require Java runtime to be available first.

The overall architecture of our system is shown in Fig. 2. Plans are submitted by a web role `Controller` to a common queue and can then be read by workers. A submitted plan is acquired by a single worker `DeployerEngine` that tries to install and execute the job. Once the execution is finished, the worker returns results to the Azure blob storage where they can be found by `Controller`.

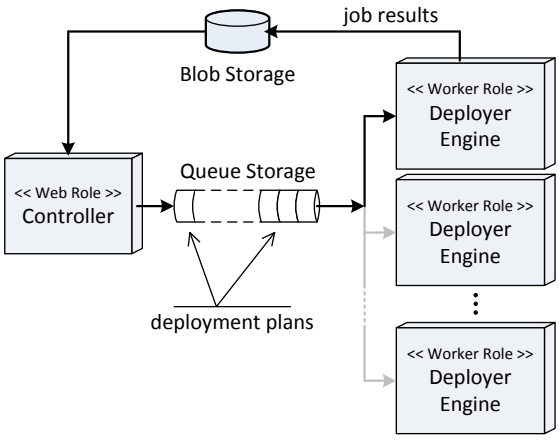

Fig. 2. An illustration of the communication path between the web role Controller and worker role Deployers.

Usually, the `Controller` sends different types of jobs to the queue, while the `DeployerEngine` matches a job to an appropriate deployer class that can handle

it. To deal with the plans we distinguished two levels of deployers: operating system level and process level. All of them realize the same interface `IDeployer` (Fig. 3). It follows a one-phase activation scenario, which is a slightly simplified version of the two-phase activation proposed by D&C. The `IDeployer` interface includes operations related to plan installation and activation, whereas the two-phase deployer also includes an intermediate, initialization step. The purpose of install is to collect together all artifacts required to activate the plan on the target node. The activate operation is responsible for running the plan, and its behavior heavily depends on the level of the implementing deployer. The meaning of deinstall and deactivate is to reverse install and activation respectively.

When implementing process-level deployers, we noticed that irrespective of the type of a job to be run (a Java class/jar, python or R script and a standalone executable) the install operation remains the same and only minor changes are required in the activate operation. The changes are mainly related to running a different virtual machine executable or standalone exec and setting their parameters according to the configuration properties provided in the plan. Moreover, for all process-based components we distinguished exactly the same input and output ports: standard input, standard output and standard error. This unifies the way in which the results of job execution can be returned back to `Controller`.

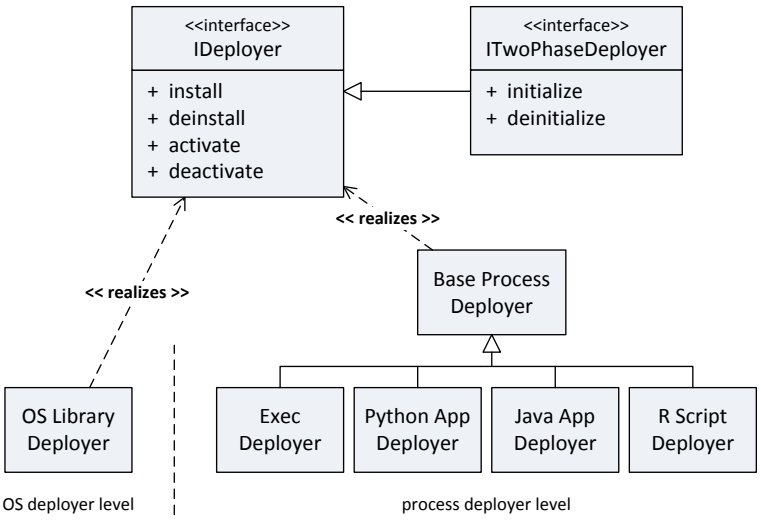

Fig. 3. Class hierarchy showing relation between different types of deployers.

Conversely, implementation of `OSLibraryDeployer` differs from the others significantly. The main task of this entity is to enable access to a requested software package for deployers at the higher-level. The idea is that `OSLibrary-`

`Deployer` installs a package and during activation exposes its contents as node resources. We have implemented an automated plan generator that produces deployment plans using manually prepared plan templates. Its main disadvantage is that it does not take node resources into account and treats all worker nodes equally irrespective of what resources they currently offer. However, in the future we would like to adapt this to search for node resources during the deployment planning phase and select the node which best fits a submitted plan.

4 Expressiveness of the Deployment Solution

One of the essential elements of the model-based deployment is expressiveness of the models that are used to describe the software and the execution environment. Although D&C defines one of the most complete deployment standards, we noticed in our previous work [2] that there are some key aspects that this specification does not address. In particular, it lacks: modelling of multilayer virtualized systems, support for different software technologies, and the ability to express temporal constraints on deployment actions.

In this section we present more details of how we adapted the platform independent model of deployment proposed by the OMG to our needs. The first important assumption is that in this work we address different virtualization levels than the one proposed in PSM for CCM. As mentioned above, a deployment plan can include components of different types such as OS libraries, executables and Python scripts. The key to map the D&C PIM to the virtualization levels addressed by our deployment engine was to notice that a component instance may represent either a process or an installed and active OS library. This allowed us to develop the mapping for a deployment plan and all its elements. Table 1 includes details of this mapping.

The main feature of the mapping presented here is the definition of a component interface. For the process level we defined a component as having three ports: input, output and error. Using these ports, components within a plan can be connected together much like processes in an operating system shell. Moreover, they can be bound to external locations such as a URL or a blob in Azure storage. Fig. 4 depicts a sample plan comprising two tasks that run concurrently. They send their error streams to a common blob; `Task1` sends its output stream to `Task2` and to an external URL. Also, it acquires input from an external location. As shown in the figure, at the process virtualization level, deployment plans that represent a composite component, also have the three ports mentioned. This allowed us to unify the way in which the results of process-based jobs are returned. By default, the outcome of a deployment plan is two blobs: output and error. Any component in a plan can be bound to these ports, which means that contents of their outputs are redirected appropriately. It is also possible to redirect ports of multiple tasks to a single output and redirect a single output to multiple locations.

At the process virtualization level, different components can also be combined together in a single plan using temporal and spatial constraints (Fig. 5). Temporal

Table 1. Mapping of D&C execution model entities to process and OS virtualization levels.

D&C entity	Process virtualization level	OS virtualization level
deployment plan	a composite of processes that may be interconnected and bound with spatial and temporal constraints	usually a single library component; potentially a composite of multiple libraries
component interface description	defines three ports: input, output and error, and a single property: priority	simple type referring to a library
artifact deployment description	a file; a part of program code (e.g. an executable, resource file, configuration script)	a file; a part of library code (e.g. an executable, resource file, configuration script)
monolithic deployment description	program code; groups all program files together	library code; groups all library files together
instance deployment description	a process	installed and active library; presents resources for higher level deployers
instance resource deployment description	assigned node resource; may refer to resources of the lower level OS deployer	assigned node resource (e.g. disk space)
plan connection description	connection between standard input, output and error process streams	n/a
plan property mapping	process priority	n/a

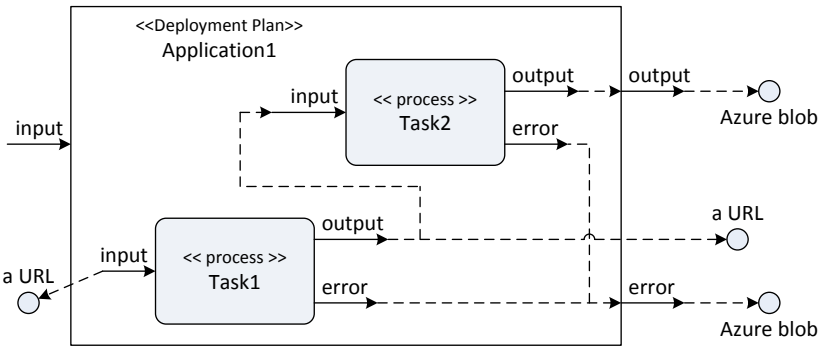

Fig. 4. A deployment plan showing input and output ports of process-based tasks. Ports may refer to URLs and Azure blobs.

constraints allow the expression of component dependencies that are usually modelled in the form of a direct acyclic graph. The `StartToStart` and `FinishToFinish` collocations enable synchronization barriers to be created between components' deployment activities, whereas `FinishToStart` allows instances to be linked in a deployment chain. We also adapted the original D&C spatial constraints[2] to the process virtualization level. Instead of `SameProcess`/`DifferentProcess`, the `SpatialConstraintKind` allows the expression of the need for running selected processes in the same or different working directories. This may be useful when components within a plan have some implicit dependencies. For example, if they use common input files, they may need to run in the same directory. Conversely, when they produce result files with the same names, they need to execute in separate working directories.

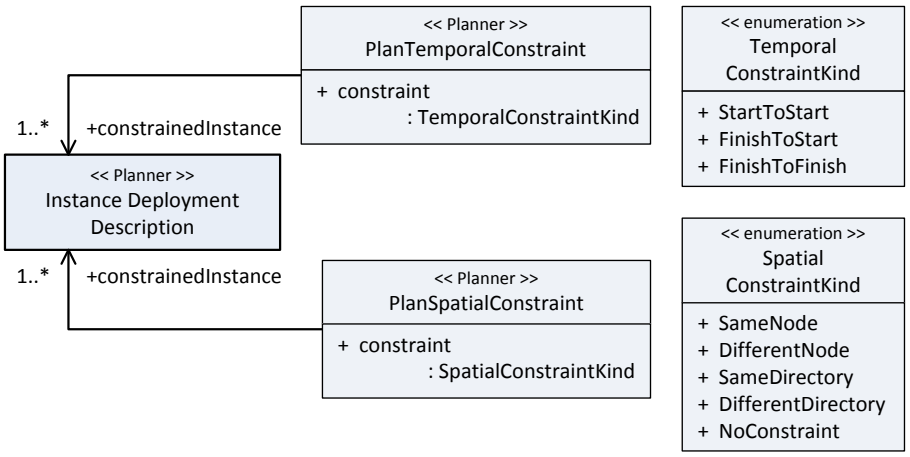

Fig. 5. Temporal and spatial constraints in the proposed deployment model.

The last element of the proposed mapping is related to resources assigned to deployed components. We made a separation between the lower level, OS deployer and higher level, process deployers. Our intention is that when the OS deployer activates a library on a node, it also registers a set of resources associated with that library. This will enable a deployment planer, working at the process virtualization level, to search for the best node to deploy. For example, if a node executed a Java-based application previously, it has a JRE deployed. Therefore, any subsequent Java-based components should preferably be deployed on this node instead of any other that does not provide any JRE. Although we do not address deployment planning yet and use manually prepared deployment plans, our platform is well suited for this.

[2] i.e. `PlanLocality` and `PlanLocalityKind` entities.

5 Application of Our Deployment Solution to Discovery Bus

Discovery Bus is a system that implements the competitive workflow architecture as a distributed, multi-agent system. It exhaustively explores the available model and descriptor space, which demands a lot of computing resources [3]. Fortunately, the architecture of Discovery Bus makes it easy to move agents to different locations in a distributed environment. This enabled us to place the most compute-intensive agents in the cloud. However, as agents are not limited to any particular implementation technology, the key requirement for our deployment platform was to support different software technologies. These include Java, R and native applications.

Our platform can deploy arbitrary jobs, however, for the purpose of the QSAR use case we prepared a number of predefined deployment plans that describe specific Discovery Bus agents. One of the most compute-intensive tasks is the calculation of the molecular descriptors. Figure 6 shows a deployment plan that we designed to run this job in Azure. First, the plan expresses a dependency on the JRE and CDK libraries. These dependencies are directed to the lower level, OS deployer that installs and activates them on an Azure worker node. Second, the plan includes an artifact description for the input data file which is provided to the descriptor calculator task on its activation. The location of this artifact is given when the plan is created. Third, the calculate descriptor program sends results to the standard output stream, which we redirect to the external output port of the plan. By default, the output of a deployment plan is stored in the Azure blob storage. Similarly, the error stream of the task is redirected to the external error port and is transmitted to the blob storage.

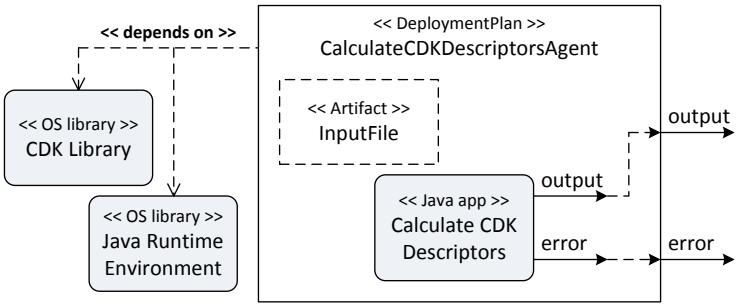

Fig. 6. A deployment plan created for a selected, Java-based Discovery Bus agent.

This example shows how easy it is to prepare a suitable deployment plan for an existing application. However, there are many cases when a process-based task returns results not only via standard output or error streams but creates files, or communicates results through a network. Neither our approach, nor Azure,

prevents sending data via a network, but the more difficult case occurs when the task execution produces files. As they are stored in local worker storage, we supplemented the deployment plan with a post-processing task that was able to transfer them to a desired location.

Figure 7 shows a sample two-task plan that sequentially executes the main R script, and an additional post processing program that converts the results and sends them to a specific location. Although this required some extra development effort, we were able to use this approach to move all of the compute-intensive Discovery Bus agents to Azure without changing any of their original code.

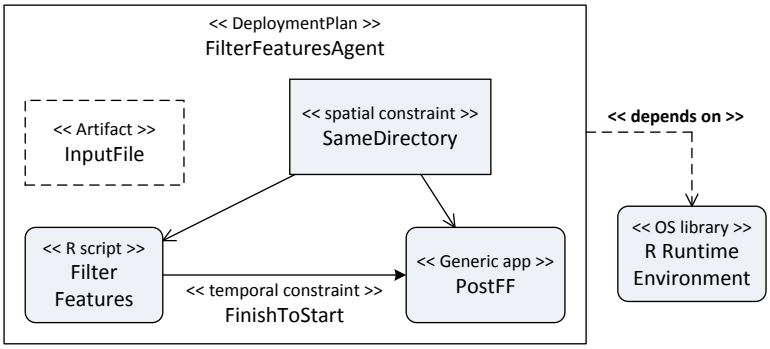

Fig. 7. One of the predefined deployment plans used in processing a Discover Bus task.

The tasks have the spatial constraint `SameDirectory` that they must be executed in the same working directory. This requirements stems from the fact that there are implicit dependencies between both tasks i.e. the `PostFF` program reads files produced by the `FilterFeatures` script. Therefore, we imposed the temporal constraint `FinishToStart` on these two tasks; as a result, `PostFF` cannot run until `FilterFeatures` completes.

The plan also shows a dependency on the R runtime environment, which is not shipped with Azure by default and, therefore, needs to be deployed prior to the execution of this plan. This dependency is directed at the `OSLibraryDeployer`. It is not, however, the same as the temporal constraints, which create a chain of deployment actions. Instead, "depends on" demands only prior availability.

6 Related Work

The problem of deployment of computing intensive jobs in distributed systems has its roots in the beginnings of distributed systems. Although cloud computing creates new opportunities to exploit, most prior work on scientific applications has been carried out in the area of Grid Computing.

Condor[3] is one of the main examples of Grid-like environments that provide batch execution over distributed systems. It is a specialized workload management system for compute-intensive jobs. Serial or parallel jobs submitted to Condor are placed in a queue and decisions are made on when and where to run the jobs based on a scheduling policy. Execution progress is then monitored until completion. To match jobs to the most appropriate resources (compute nodes) Condor uses the ClassAd mechanism, which is a flexible and expressive framework for matching resource requests (jobs) with resource offers (machines). Jobs can state both job requirements and preferences, while compute nodes can specify requirements and preferences about the jobs they are willing to run. All of these may be described through expressions, allowing a broad range of policies to be implemented [9].

There are three key differences between Condor and our solution. First, Condor uses the ClassAd mechanism and a central manager to schedule jobs, whereas Azure proposes, and we use, the queue-based approach for scalability. However, Condor supports a flexible matching of jobs to resources, while we treat all computing resources as being indistinguishable. We leave the problem of resource discovery and deployment planning for future work. Second, the ability to express dependencies between deployment plans such as between an R script and its execution environment allows for the clear and concise definition of job prerequisites. This simplifies complex deployment in distributed environments, and is not supported by Condor. Third, we based our solution on D&C deployment models that provide the recursive definition of a component which either can be a monolith or an assembly of subcomponents. Condor instead uses the ClassAd mechanism that describes a job without any relationship to others, except where temporal constraints are expressed by directed acyclic graphs and evaluated by the DAGMan scheduler.[4] We believe that temporal constraints are an important feature in a deployment plan and therefore supplemented D&C models to enable their definition. Condor does not support more sophisticated constraints, such as the resources required to realize a connection between executing jobs. Neither do we currently consider this aspect of D&C models, but we expect to extend our system to use it in the near future work.

The Neudesic Grid Computing Framework (Neudesic GCF)[5] is dedicated to the Azure platform. It provides a solution template and base classes for loading, executing, and aggregating grid tasks on the Microsoft cloud. Neudesic GCF offers a set of templates such as Worker, Loader and Aggregator. A Worker implements each of the computing tasks, a Loader is added to read input data from local resources and generate tasks, and an Aggregator receives results and stores them. Unlike with our solution, Neudesic GCF requires building applications from scratch. A developer is given a set of templates that they can use for programming, but as everything needs to be built in .Net, this framework was not suitable for our use case. Moreover, the Neudesic GCF assumes the homogeneity

[3] http://www.cs.wisc.edu/condor
[4] http://www.cs.wisc.edu/condor/dagman
[5] http://azuregrid.codeplex.com

of Azure resources as all workers are identical. With the use of model-based deployment, our solution is better prepared to be extended to take into account of heterogeneous Azure computing resources. Several Cloud computing solutions are available apart from Azure. The most prominent ones are the Google App Engine (GAE) and the Amazon Elastic Computing Cloud (Amazon EC2).

The Google App Engine[6] is a cloud computing platform primarily dedicated to Python and Java language applications. It provides a high-level platform for running distributed systems and has the built-in ability to automatically scale the number of working nodes following changes in the incoming workload. However, unlike Microsoft Azure, this environment is closed with respect to running native programs, which would require us to rewrite the existing code. Moreover, as it is aimed at building scalable web servers, GAE limits the time for processing a request to 30 seconds[7] which makes it unsuitable for scientific computation in general and our QSAR scenario in particular, as this often needs much more time to complete a single job.

Amazon EC2[8] provides a virtual computing environment controlled through a web service interface. It gives complete control over the operating system installed on a computing node, which makes it more open than Azure [8]. However, as a result, it demands more administrative and management effort, whereas Azure can adopt a declarative approach to configuration through simple configuration descriptors. Despite the fact that our solution is based on Azure, we believe that it could be ported to Amazon EC2 as well, providing the same benefits of deployment automation as for Azure.

RightScale[9] offers a range of products that can run on top of different cloud platforms such as Amazon EC2, Rackspace, FlexiScale. RightScale's Grid Edition framework (GE framework) provides a queue-based solution for running batch processes. Overall, it follows similar pattern that we use. Additionally, the GE framework enables auto-scaling that can adapt the number of worker nodes in response to changing factors. The key difference between RightScale's GE framework and our deployment solution is in the granularity of workers. The GE framework as a basic unit of deployment uses server templates. A server template is a preconfigured image of an operating system that is used to launch new server instances in the Cloud. For each particular task or set of tasks a user needs to prepare an appropriate server template. Instead, in our approach a basic unit of deployment is a process-level component that is much smaller when comparing to an OS image. Our deployment platform allows running arbitrary jobs on any active worker irrespective of which type of job it is.[10] This promotes better resource sharing and guarantees more effective solution, especially for

[6] http://code.google.com/appengine

[7] See Google App Engine Documentation (ver. 2009-12-15) Sect. What Is Google App Engine; Quotas and Limits.

[8] http://aws.amazon.com/ec2

[9] http://www.rightscale.com

[10] Currently this is any kind of task from a supported set of task types: a Windows executable, a Java jar/class file, an R script, a Python script or a command-line script.

smaller and short running tasks. Moreover, our deployment plans can include many interrelated subtasks, which results in a much more expressive framework and enables the assembling of applications from existing components.

7 Conclusions and Future Work

In this paper we discuss an automatic deployment framework for the Azure cloud platform. We use the framework to run compute-intensive Discovery Bus agents in the cloud. Despite the fact that none of the Bus components was implemented in the .Net framework, our framework allowed us to seamlessly integrate most existing agents with Azure without any code modification. The exception was for those agents that produced results in the local file system. Here, some additional development effort was required; we had to implement a simple task to transfer output files to a designated location. Then, using our deployment platform we could express spatial and temporal constraints between processes so ensuring that all the results produced are correctly transmitted.

Apart from results specifically focussed on the Discovery Bus as a motivating scenario, this work showed that the platform independent model defined in the D&C specification can be successfully applied to the process and OS virtualization levels. The PIM creates a very expressive deployment framework that with only minor modifications allowed us to build composite deployment plans describing process-based applications. Further, the ability to express dependencies between deployment plans supports the clear and concise definition of component prerequisites.

By using the proposed deployment framework we were able to successfully run Discovery Bus agents in Azure. To generalise the solution, we see three major directions for future work. Firstly, we would like to support the two-phase initialization of process-based components. This will enable us to distribute components included in a deployment plan over multiple worker nodes even in the case when components are interconnected. Secondly, we see the need for deployment planning to make our solution more effective. The ability to determine node resources would allow the matching of work to those nodes that best fit the submitted plan. However, our major focus remains on preserving efficient scalability for a queue-based system, while enabling resource discovery and deployment planning (an interesting approach to this problem is presented in [4]). Finally, an interesting future direction for our framework is the dynamic deployment of new deployer types. If our DeployerEngine implements the IDeployer interface, we can dynamically install and activate deployers that support new component types. This in turn will increase flexibility and facilitate the runtime evolution of the deployment platform according to changing needs.

Acknowledgements We would like to thank Microsoft External Research for funding this work. Also, we are grateful to the wider team working on the project: the Discovery Bus team (David Leahy, Vladimir Sykora and Dominic Searson), the e-Science Central Team (Hugo Hiden, Simon Woodman) and Martin Taylor.

References

1. Abley, J., Lindqvist, K.: Operation of Anycast Services. Request for Comments 4786, Best Current Practice 126 (2006)
2. Cala, J.: Adaptive Deployment of Component-based Applications in Distributed Systems. PhD thesis, AGH-University of Science and Technology, Krakow (submitted Feb. 2010)
3. Cartmell J., Enoch, S., Krstajic, D., Leahy, D.E.: Automated QSPR through Competitive Workflow. Journal of Computer-Aided Molecular Design, 19: 821–833 (2005)
4. Castro, M., Druschel, P., Kermarrec, A.-M., Rowstron, A.: Scalable Application-Level Anycast for Highly Dynamic Groups. Networked Group Communication. LNCS 2816, 47–57 (2003)
5. Dearle, A.: Software Deployment, Past, Present and Future. FOSE'07: 2007 Future of Software Engineering, 269-284, IEEE Computer Society (2007)
6. Talwar, V., Milojicic, D., Wu, Q., Pu, C., Yan, W., Jung, G.: Approaches for Service Deployment. IEEE Internet Computing, 9(2):70–80 (2005)
7. Watson, P., Hiden, H., Woodman, S., Cała, J., Leahy D.: Drug Discovery on the Azure Cloud. Poster presentation on Microsoft e-Science Workshop, Pittsburgh (2009)
8. Amazon Web Services LLC, Amazon Elastic Compute Cloud: User Guide; API Version 2009-11-30 (2010)
9. Condor Team: Condor Version 7.5.0 Manual. University of Wisconsin-Madison (2010)
10. Microsoft Corp.: Windows Azure Queue — Programming Queue Storage. Whitepaper (2008)
11. Object Management Group, Inc.: Deployment and Configuration of Component-based Distributed Applications Specification; Version 4.0 (2006)
12. Object Management Group, Inc.: Common Object Request Broker Architecture (CORBA) Specification, Version 3.1; Part 3: CORBA Component Model (2008)