A Technology to Expose a Cluster as a Service in a Cloud

Michael Brock and Andrzej Goscinski
School of Information Technology, Deakin University
Pigdons Road, Waurn Ponds, Victoria 3217
{mrab, ang}@deakin.edu.au

Abstract

Clouds refer to computational resources (in particular, clusters) that are accessible as scalable, on demand, pay-as-you-go services provided in the Internet. However, clouds are in their infancy and lack a high level abstraction. Specifically, there is no effective discovery and selection service for clusters and offer little to no ease of use for clients. Here we show a technology that exposes clusters as Web services in the form of a Cluster as a Service for publishing via WSDL, discovering, selecting and using clusters.

Keywords: Cluster as a Service, WSDL Publishing and Selection, Dynamic Brokering, Clouds.

1 Introduction

Cloud computing is made possible through the combination of virtualization, Service Oriented Architecture (SOA), and Web and RESTful services. Virtualization allows any computer platform to be supported regardless of hardware and software. By abstracting cluster and server software, it improves the efficiency, availability, access and use of resources and applications. Virtualization enables the use of idle cycles of resources of datacenters, which are in 80% unused. SOA forms an architectural basis for the cooperation of clients, services, and registries (Papazoglou and van den Heuvel 2007). Web and RESTful services provide a high level abstraction and highly interoperable communication subsystem. Scalable data centers offer dynamic and huge hardware provisioning. The end result is an inexpensive, Internet accessible on demand environment where clients use computing resources on a pay-as-you-go basis as a utility and are freed from hardware and software provisioning issues.

For clients to access resources they must be discovered. Clouds and their computational resources are not easy to discover and it is difficult to select and use services. An analysis of the three main clouds (EC2 (Amazon 2007), Azure (Microsoft 2009), and AppEngine (Google 2009), has found that they and their services/resources are difficult to discover and offer little to no ease of use unless the user is a software developer.

Clusters are a basic component of clouds. However, it is difficult to discover clusters and select a cluster that satisfies client requirements. This implies that clients have to access every cluster from a list provided by a registry to learn about their state and characteristics. It is also not easy to use cloud clusters as they are not exposed at a high level of abstraction (Jha et al. 2009).

This paper presents an outcome of our project that addresses some of these problems. It shows a technology that exposes a cluster as a service that offers a high level abstraction of clusters in the form of Cluster as a Service (CaaS). The proposed technology is based on the Resources Via Services (RVWS) framework (Brock and Goscinski 2008a; Brock and Goscinski 2008b). The technology proposed in this paper allows efficient exposing and publishing via WSDL documents of Web services exposing clusters, their discovery and selection of a requested cluster and makes its use easier. As the WSDL document is the most commonly requested and recorded object of a Web service, the inclusion of cluster's state and other information in the WSDL document makes the internal activity of the Web services exposing this cluster publishable.

It is important to mention that this paper does not address cloud SLAs (Service Level Agreements), business and provisioning models, security and reliability (network and computer system outages), although they are seen as critical aspects of clouds.

The rest of this paper is structured as follows. Section 2 discusses three well known clouds and concludes that they and their services/resources are difficult to discover and do not support service selection and ease of use well. Section 3 introduced a high level abstraction and architecture of the CaaS Technology. Section 4 discusses, following a brief characterization of the RVWS framework, the logical design of CaaS. All components responsible for the publication of clusters, their discovery and selection, and actual use are discussed extensively. Section 5 presents a proof of concept in the form of implementation and experiments carried out to demonstrate the way a cluster is published, found and used. Section 6 provides a conclusion.

2 Related Work

While the use of Web services has made cloud services interoperable, Web services are natively stateless, and can complicate the exposure of resources that depend heavily on state. The WSRF framework (Czajkowski et al. 2004) makes Web services stateful but the state itself is not publishable. The lack of published state forms a major obstacle: if the state of the Web service is not published, clients cannot learn if the Web service is ready for requests or not. Furthermore, while the standards behind the publication of Web services are extensive, their practice is limited greatly to static parameters such as the publication of Web service functionality, communication patterns, and provider contact details. Finally, the use the clouds are not easy and require clients to have good knowledge of them.

Virtualization lays the foundation for sharable on demand infrastructure, on which three basic cloud abstractions are offered on demand:

- Infrastructure as a Service (IaaS) – makes basic computational resources (e.g., storage, servers) available,
- Platform as a Service (PaaS) – makes offering that enable easy development and deployment of scalable applications, and
- Software as a Service (SaaS) – allows complete end user applications to be deployed, managed, and delivered over the Web.

The big four clouds, EC2 (Amazon 2007), Azure (Microsoft 2009), AppEngine (Google 2009) and Salesforce.com (Salesforce 2009) that represent these three basic cloud abstractions, only provide basic support.

Initially, EC2 was basically hardware as a service; it was for the user of EC2 to create the entire software stack starting with an operating system. Just recently, the Amazon announced their cluster services, i.e. Auto-Scale, Load-balance, and CloudWatch, moving the cloud to the PaaS category. However, features such as Auto-Scale, require involvement of the EC2 client. Before one can use Auto-Scale, one has to create multiple instances of Amazon images for Auto-Scale to utilize.

AppEngine is a PaaS cloud where clients are able to construct services and deploy them to AppEngine for execution without having to rebuild the software stack. However, AppEngine is very restricted in what language and technology can be used to build the services. At the time of writing, AppEngine only supports the Java and Python programming languages.

Both AppEngine and EC2 offer cluster like data processing in the form of MapReduce (Dean and Ghemawat 2004) and Hadoop (Apache 2009) respectively. In AppEngine, services are created to use Google's MapReduce framework while EC2 offers virtual servers with Hadoop installed. However, both MapReduce and Hadoop are restrictive because they are specialized for distributed data processing.

Salesforce is a SaaS cloud: specifically, CRM software as a service. Instead of maintaining hardware and software licenses, use of the software hosted on Salesforce servers for a minimal fee is offered. However, Salesforce is still primitive. The software cannot be customized and discovery of required software is only keyword based.

These approaches appear to have no means of discovery. At this time Windows Azure is seen as the only cloud that offers discovery. An underlying component to Azure is the .NET Services Bus. When a service is hosted in Azure, it is able to register a URI to the Bus so that clients can discover the service. As the Bus does the location resolution, clients are able to use the service no matter where the service is moved. While the Bus offers discovery, its solution is still not satisfactory. The only element that can be registered is a URI. All other elements such as attributes on activity and limitations are not published.

In summary, these four clouds provide some form of service management. However, they require, with the exception of Azure, new and dedicated programming environments. Furthermore, clients must be heavily involved in configuration of virtual servers and execution of their applications in the same manner as programmers did years ago when they used a command driven Unix system (Chaganti 2008; VCL 2008). Clients face difficult problems of resource discovery and automatic services selection; dynamic sharing toward efficient management of resources; QoS and reputation of providers and clients; and fault tolerance. What is needed is an approach to simplify the publication of clusters, their discovery and actual use. Clients should be able to easily place required files and executables on the cluster, and get the results back without knowing any cluster specifics. A solution is in the proposed Cluster as a Service, a high level abstraction of clusters within clouds.

3 Cluster as a Service

Our Cluster as a Service (CaaS) Technology belongs to the category of PaaS clouds. The purpose of the CaaS Technology is to expose a cluster as a dynamically changing stateful service, manage the discovery and selection of clusters, the specification of cluster jobs[1], the upload of required files, the monitoring of execution and the download of result files. The CaaS Technology does not require any special development environment; it supports development, deployment and execution of applications that traditionally could be executed on a standalone cluster. The CaaS abstraction and technology is applicable to both public and private clouds.

This section discusses the CaaS technology in detail. In particular, as this the technology is immersed in the RVWS framework, the framework is briefly introduced. That is followed by the presentation of the CaaS high level abstraction, architecture, behavior, and the use of a stateful WSDL document.

3.1 RVWS Basics

While Web services have simplified resource access, it is not possible to know if the resource behind the Web service is ready for a request. In fact, it is out right impossible to easily find Web services that satisfy the client requirements. To do so requires clients to research extensively the services themselves before they are used.

To address these issues, we proposed our Resources Via Web Service (RVWS) framework. Diagram 1 shows an overall vision of RVWS in relation to clients and clouds. A key element of RVWS is the discovery of services and resources using state and characteristic attributes published to Web service WSDL documents.

The automatic service discovery allows for both a single service (e.g., a cluster) discovery and selection, and an orchestration of services to satisfy computation workflow requirements. The SLA (Service Level Agreement) reached by the client and cloud service provider specifies attributes of services, in particular clusters, that form the client's request or workflow. This

[1] It is good to recall the difference between processes and jobs. Jobs contain programs, data and even configuration and/or management scripts. A process is a program that is in execution. When clients use a cluster, they submit jobs and one or more processes are created to execute the job.

is followed by the process of services' selection using brokers. Thus, selection is carried out automatically and transparently.

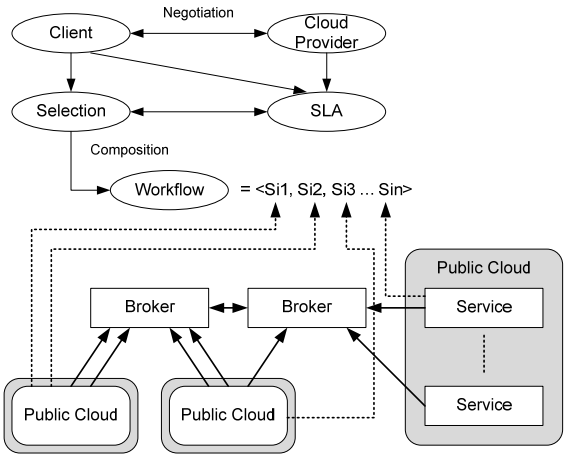

Diagram 1: Dynamic Discovery and Selection

There are two categories of dynamic attributes addressed in the RVWS framework (Brock and Goscinski 2008a), (Brock and Goscinski 2008b): state and characteristic. State attributes cover the current activity of the service and its resources thus indicating if a given service is ready for client requests. Characteristic attributes cover the operational and physical limitations of the service, the resources behind it, quality of service (QoS), price and even information about the providers of the services.

To keep stateful Web services current to their resources, a Connector (Brock and Goscinski 2008a) is used to detect changes in resources and then pass them on to the Web service. The Connector has three logical modules to keep the stateful Web service current: Detection, Decision and Notification. The Detection module routinely queries for attribute information from the resource. Any changes in the attributes are passed to the Decision module that decides if the attribute change is large enough to warrant a notification thus preventing excessive communication with the Web service. If notification is needed, the updated attributes are passed on to the Notification module. Once the attribute changes have been assessed, this module informs the stateful Web service that updates its internal state. When Clients requests the stateful WSDL document, the Web service returns the WSDL document with the values of all attributes at the request time.

Through the extensible nature of the WSDL document, it is possible to include additional information (state and characteristics) by encapsulating it in its own section. All information of service resources is kept in a new WSDL section called the Resources section. For each resource behind the Web service, a ResourceInfo element exists. Each ResourceInfo section has a resource-id attribute and two child sections, the state section and the characteristic section. When the Connector learns of the resource for the first time it publishes the resource to the Web service.

While the stateful WSDL document eliminates the overhead incurred from manually learning the attributes of the service and its resource(s), the issues behind discovering needed services are still unresolved.

To help ease the discovery of services with stateful WSDL documents, a Broker was proposed (Brock and Goscinski 2009). The Broker is able to transparently contact other known Brokers if it cannot satisfy a Client request.

When publishing to the Broker, the Provider sends attributes of the Web service. The provider is even able to publish attributes about itself, e.g., name, price. After providing attribute information about the Web Service, the Broker gets the stateful WSDL document from the Web service. After getting the stateful WSDL document, the Broker has the complete attribute set that is stored across three stores: the Service, Resources and Provider stores. As the Web service changes, it is able to send a notification to the Broker (and also to the client if necessary) that then updates the relevant attribute in the relevant store.

When seeking desired services, the Client submits to the Broker three groups of attribute values for Service, Resource, and Provider. The Broker compares each attribute group on the related data store. Then, after getting matches, the Broker applies filtering. As the Client using the Broker could be anything from a human operator behind a web browser to another software unit, the resulting matches have to be filtered to suit its needs. Finally, the filtered results are returned to the Client.

3.2 CaaS – High Level of Abstraction

Clouds represent a high level of abstraction of a large distributed system achieved through layers of virtualization and abstraction. One of highest levels of the abstraction hierarchy is the concept of a service. This implies that cloud should offer a simple high level interface that allows clients to discover, select and access cloud resources (storage, CPU cycles) exposed as services easily and transparently.

An attempt of forming a hierarchical model of abstraction to address stateless Web service, stateful Web services based on the WSRF framework that do not expose state directly, and stateful Web services based on the RVWS framework that exposes dynamically changing state directly was presented in (Brock and Goscinski 2008a, Brock and Goscinski 2008b).

A proposal of creating a higher-level abstraction for grids to provide an explicit support for usage modes was introduced in (Jha et.al 2009). However, the authors have not moved beyond a general description of their vision.

This section shows a layer of abstraction that takes advantage of the hierarchy presented in (Brock and Goscinski 2008a, Brock and Goscinski 2008b) that sits within the cloud abstraction. The relationship among existing stateless Web service standards and frameworks, and the level of abstraction provided by the CaaS, an example of the PaaS cloud, that offers an interface for clients are shown in Diagram 2.

The CaaS provides the highest level of abstraction that hides all hardware and software features of cloud clusters. Clients only receive minimum operational amount of data (service location, invocation interface, current state and characteristics information) and are provided Web pages to deploy, run, and control execution of their jobs.

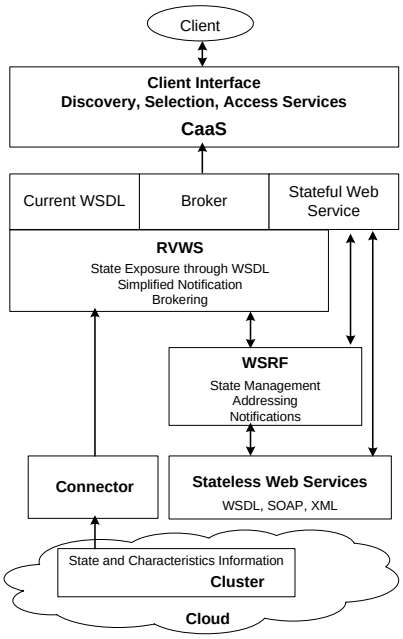

Diagram 2: CaaS Abstraction

3.3 CaaS Architecture and Behavior

The exposure of a cluster via a Service is intricate and comprises several services running with and on top of a physical cluster. Diagram 3 shows the complete solution with a cluster, RVWS and the proposed CaaS service.

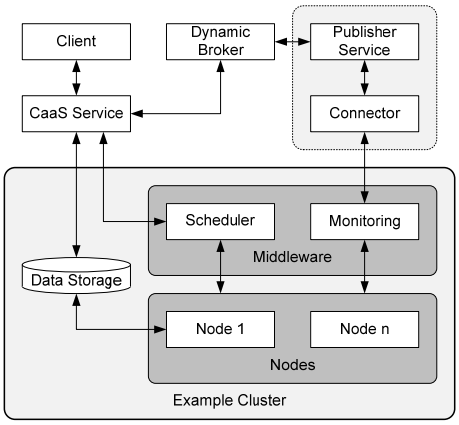

Diagram 3: Complete CaaS System

A typical cluster is comprised of four elements, nodes, fast networks, data storage and middleware. As the focus of this paper is abstraction, only cluster middleware is addressed here. Cluster middleware, a basic level of virtualization of clusters, is comprised of multiple components to manage the cluster and provide a single system image (Goscinski et al. 2002) thus preventing the process running on the cluster from needing to know the cluster organization.

With all the services in the middleware and the changes in node load, there is a lot of information to consider when finding a cluster. As time progresses, the amount of free memory, disk space and CPU usage of each cluster node changes dramatically. Furthermore, information about how quickly the scheduler can take a job and start it on the cluster is vital in choosing a cluster. Currently used Web services (stateless and even WSRF stateful) do not take this changing information into account.

Thus while the Broker makes information about a cluster known (location and service invoking information), easing the use of a cluster was still left open. Clients could find required clusters but they still had to manually transfer their files, invoke the scheduler and get the results back. All three tasks require knowledge of the cluster and are conducted using work demanding tools.

To make information about the cluster publishable the RVWS framework was used to create a Cluster Connector and Publisher Web service. The role of the Publisher Web service was to show current cluster dynamic attribute information via a stateful WSDL document. To make the Publisher Web service (and the cluster behind it) discoverable, our Broker was used.

The role of the CaaS Service is to (i) find (using the Broker) matching clusters based on client requirements, (ii) provide easy and intuitive file transfer tools so clients can upload jobs and download results, and (iii) offer an easy to use interface for clients to use the cluster.

It may be required that data for cluster jobs be stored in a designated location (a directory) within the storage. As clients to the cluster cannot know any of this information, it is for the CaaS service to abstract the transfer of data files to the point where clients appear to operate the cluster storage as one of their own storage systems. Finally, the CaaS Service communicates with the cluster's scheduler, thus freeing the client from needing to know how the scheduler is invoked when submitting and monitoring jobs.

```
<definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <resources>
    <resource-info resource-identifier="resourceId">
      <state element-identifier="elementId">
        <cluster-state element-identifier="cluster-state-root">
          <cluster-node-name free-disk="" free-memory="" native-os-name=""
            native-os-version="" processes-count=""
            processes-running="" cpu-usage-percent=""
            element-identifier="stateElementId" memory-free-percent="" />
          …Other Cluster Node State Elements…
        </cluster-state>
      </state>
      <characteristics element-identifier="characteristicElementId">
        <cluster-characteristics node-count=""
          element-identifier="cluster-characteristics-root">
          <cluster-node-name core-count="" core-speed="" core-speed-unit=""
            hardware-architecture="" total-disk="" total-memory=""
            total-disk-unit="" total-memory-unit=""
            element-identifier="characteristicElementId" />
          <supported-software>
            <software version="" type="" version="" />
            …Other Supported Software Elements…
          </supported-software>
          …Other Cluster Node Characteristic Elements…
        </cluster-characteristics>
      </characteristics>
    </resource-info>
  </resources>
  <types>...
  <message name="MethodSoapIn">...
  <message name="MethodSoapOut">...
  <portType name="CounterServiceSoap">...
  <binding name="CounterServiceSoap" …>...
  <wsdl:service name="CounterService">...
</wsdl:definitions>
```

Figure 1: Publisher Web Service WSDL

3.4 Publisher Web Service Stateful WSDL

Through the extensible nature of the WSDL schema (Christensen 2001, Papazoglou 2008), it is possible to include additional information (specifically, state and characteristics) into existing WSDL documents. This is possible by encapsulating the additional information in its own WSDL section.

All information of service resources is kept in a new WSDL section called Resources. The core significance of RVWS is its combination of the WSDL of Web services with dynamic attributes. Figure 1 shows the resources section added to the WSDL of the cluster Web service.

Both the state and characteristics elements contain several description elements; each with a name attribute and (if the provider wishes) one or more attributes of the service. Attributes in RVWS use the {*name*: *op value*} notations. An example attribute is {cost: <= $5}. As well as showing the attributes in the stateful WSDL document, the attribute information has to be organized so Clients viewing the stateful WSDL document immediately understand what each attribute means.

For the CaaS service to properly support the role of cluster discovery, extensive information about clusters and their individual nodes needs to be published to the WSDL document of the Cluster Web Service and subsequently to the Broker. Table 1 shows attributes of each node in a cluster.

4 CaaS Service Design

This section discusses the CaaS Service design. In particular, it shows the CaaS components, user interfaces, and their behavior. The CaaS service carries out three main tasks: Cluster Discovery and Selection, Job Management and File Management. Given the size and number of tasks, the CaaS service was modularized. Diagram 4 shows the structure of the service.

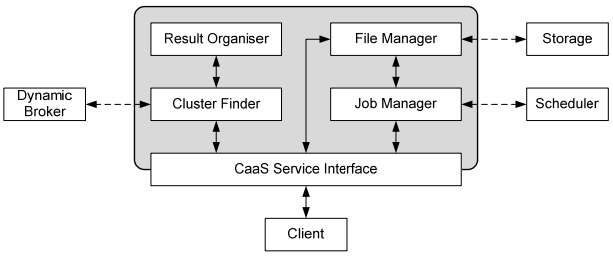

Diagram 4: CaaS Service Design

The modules inside the Web service are only accessed through an interface. The use of the interface means the Web service can be updated over time without requiring clients to be updated nor modified.

Invoking an operation on the CaaS Service Interface (discovery, selection, etc) invokes other operations on other modules. Thus, to describe the role each module plays in the CaaS service, the following sub-sections outline the various tasks the CaaS service carries out.

4.1 Cluster Discovery and Selection

The dynamic attribute information only relates to clients that are aware of them. Human clients know what the attributes are, owning to the section being clearly named. Software clients designed pre-RVWS ignore the additional information as they follow the WSDL schema that we have not changed.

When discovering services, the client must submit to the Broker three groups of attribute values (1 in Diagram 5); Service, Resource, and Provider. Thus to start discovery, clients provide cluster requirements in the form of attribute values, such as the number of nodes, to the CaaS Service Interface (1). Next, the CaaS Service Interface invokes the Cluster Finder module (2) that communicates with the Broker (3). The Broker returns (if any) an array of service matches which offer clusters that match the supplied requirements.

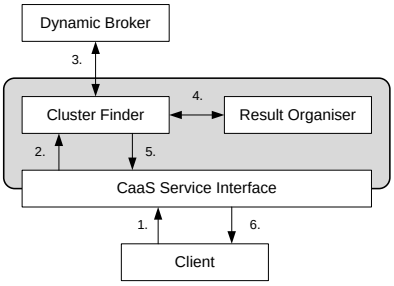

Diagram 5: Discovering suitable Clusters

To address the granularity of the Broker results, the Cluster Finder module invokes the Results Organizer module (4) that takes the Broker results and returns a summarized version. After getting the organized results, the results are returned to the client via the CaaS Service Interface (5-6). The organized results instruct the client what clusters exist and how each cluster matches up to the requirements. After reviewing the results, the client is

Table 1 Cluster Attributes

Type	Attribute Name	Attribute Description	Source
Characteristics	core-speed	Speed of each core	Cluster Node
	core-speed-unit	Unit for the core speed (e.g.: GigaHertz)	
	total-disk	Total amount of physical storage space	
	total-disk-unit	Storage amount unit (e.g.: Gigabytes)	
	total-memory	Total amount of physical memory	
	total-memory-unit	Memory amount measurement (e.g.: Gigabytes)	
	supported-software-name	Name of a single piece of software installed on the cluster	
	supported-software-type	Type of software installed on the cluster (eg: operating system)	
	supported-software-version	Version of a single piece of software installed on the cluster (eg: 6.1.0)	
	node-count	Total number of nodes in the cluster	Generated
State	free-disk	Amount of free disk space	Cluster Node
	free-memory	Amount of free memory	
	os-name	Name of the running operating system	
	os-version	Version of the running operating system	
	cpu-usage-percent	Overall percent of CPU used. As this metric is for the node itself, this value becomes averaged over cluster core	Generated
	memory-free-percent	Amount of free memory on the cluster node	

informed enough to choose a cluster.

The automatic selection of services is performed to optimize a function reflecting client requirements. Time critical and high throughput tasks benefit by executing a computing intensive application on multiple clusters exposed as services of one or many clouds.

4.2 Job Submission

After selecting a required cluster, all executables and data files have to be transferred to the cluster and the job submitted to the scheduler for execution, as shown in the Diagram 6 workflow.

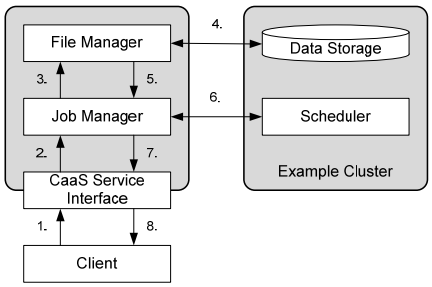

Diagram 6: Job Submission

All required job execution parameters, data files, executables, scripts, software libraries (if any) and are uploaded to the CaaS Service (1). Once the file upload is complete, the Job Manager is invoked (2). As the CaaS Service still has the required job files, the first task the Job Manager resolves is the transfer of all files to the cluster by invoking the File Manager (3). The File Manager makes a connection to the cluster data storage and commences the transfer of all files (4). Upon completion of the transfer (4), the outcome is reported back to the Job Manager (5). On failure, a report is sent and the client can decide on the appropriate response.

If the file transfer was successful, the Job Manager invokes the scheduler on the cluster (6) with the execution parameters given in (1). If the outcome of the scheduler (6) is successful, the client is then informed via the CaaS Service Interface (7-8). The information conveyed includes the response from the scheduler, the job identifier the scheduler gave to the job and any other information the scheduler provides.

4.3 Job Monitoring

Diagram 7 outlines the workflow the client takes when querying about his or her job after submitting it.

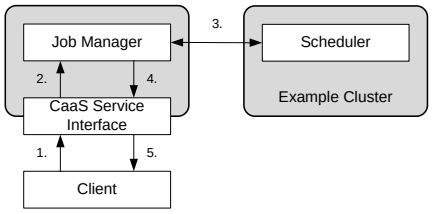

Diagram 7: Job Monitoring

The client first contacts the CaaS Service Interface (1) that then invokes the Job Manager module (2). No matter what the operation is (check, pause or terminate), the Job Manager only has to communicate with the Scheduler (3) and reports back a successful outcome to the Client (4-5).

4.4 Result Collection

The final role of the CaaS Service is addressing jobs that have terminated or have completed their execution successfully. In either case, data/error files meant for the client need to be transferred to the client, Diagram 8.

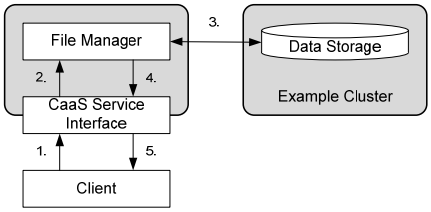

Diagram 8: Job Result Collection

Clients start the result/error file transfer by contacting the CaaS Service Interface (1) that then invokes the File Manager (2) to retrieve the files from the cluster's data storage (3). If there is a transfer error, the File Manager attempts to resolve the issue first before informing the client. If the transfer of files (3) is successful, the files are returned to the CaaS Service Interface (4) and then the client (5). When returning the files, URL link or a FTP address is provided so the client can retrieve the files.

4.5 User Interface

Users access systems based on their interfaces and how easy they are to use. Thus, to ease the use of CaaS service, a series of Web pages has been designed. Each page in the series covers a step in the process of discovering, selecting and using a cluster.

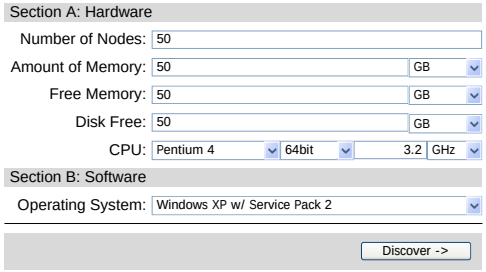

Diagram 9: Cluster Discovery – Cluster Specification

Diagram 9 shows the Cluster Specification Web page to start cluster discovery. In Section A, the client is able to specify attributes about the required cluster. Section B allows the client to specify any required software the cluster job needs. Once all attributes have been specified, the attributes are then given to the CaaS service, which performs a search for possible clusters and the results are displayed in a Select Cluster Web page, Diagram 10. A match is indicated by showing a tick in the cell or a value of an attribute. The client can choose a cluster or go back to refine the discovery.

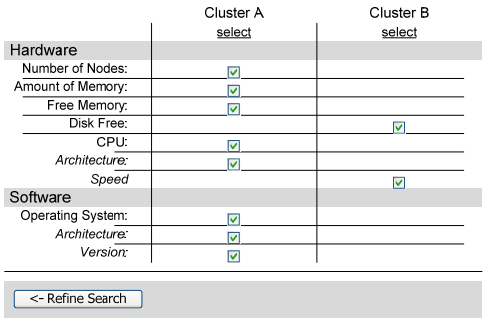

Diagram 10: Cluster Discovery – Cluster Selection

Next, the client goes to the Job Specification page, Diagram 11. Section A allows specifying information about the job. Section B allows the client to specify and upload all data files and job executables. If the job is complex, Section B also allows specifying a job script. Section C allows specifying an estimated time the job would take to complete.

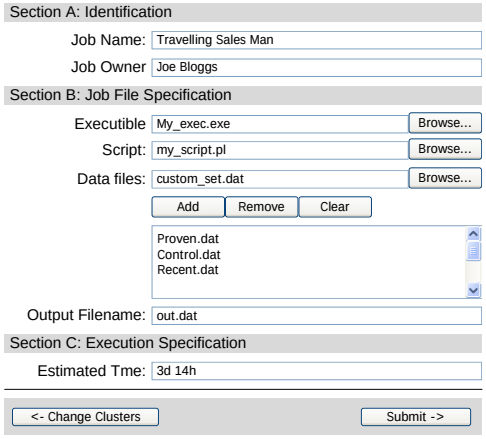

Diagram 11: Job Execution – Job Specification

Once submitted, the Cluster Web service attempts to submit the job. The outcome of the submit attempt is shown in the Job Monitoring page, Diagram 12. Section A tells the client whether the job is submitted successfully. Section B offers commands to allow the client to refresh the displayed information, pause the job, and even halt it.

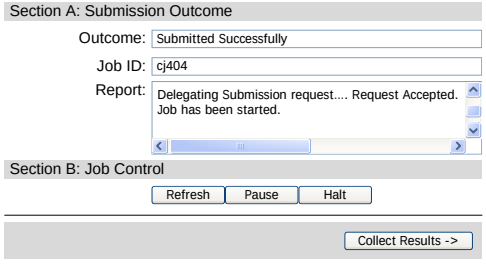

Diagram 12: Job Execution – Job Monitoring

When the job is complete, the client is able to collect the results from the Collect Results page Diagram 13. Section A shows the outcome of the job. Section B allows the client to easily download the output file generated from the completed/aborted job via HTTP or using an FTP client.

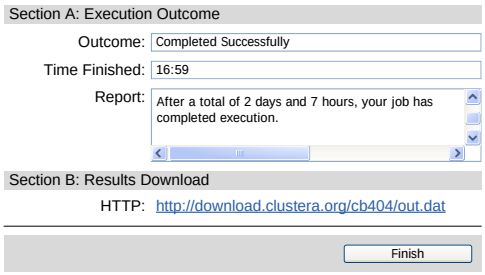

Diagram 13: Job Execution – Result Collection

5 Proof of Concept

This paper proposes a new technology. Thus, it is important to demonstrate that it is feasible. This proof of concept is provided by showing the CaaS implementation and experiments carried out.

5.1 CaaS Implementation

The CaaS service was implemented using Windows Communication Foundations (WCF) of .NET 3.5 that uses Web services. The CaaS is shown in Diagram 14.

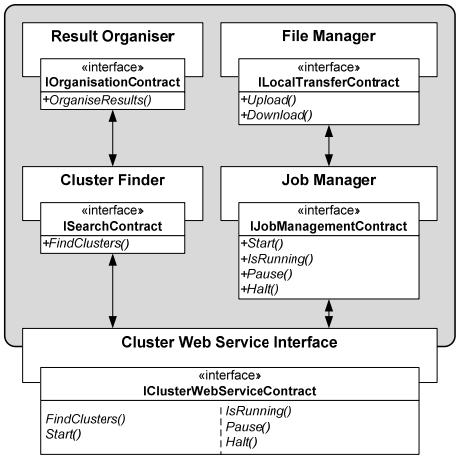

Diagram 14: Cluster Service Implementation

Each module presented in Section 4 is implemented as its own Web service. An open source library for building SSH clients in .NET called sharpSsh (Gal 2005) was used in the implementation of the Job and File Managers. As schedulers are mostly command driven, the commands and outputs were wrapped into a Service.

One final implementation change we made was adding automatic hostfile generation. This is to prevent over allocation of cluster nodes to the job. For example, if we only ask for 2 nodes, a host file containing two nodes will be generated to prevent the cluster from allocating more than two nodes.

5.2 Environment

The experiments were carried out on a single cluster exposed via CaaS; communication was carried out only through the CaaS service interface. To manage all the services and databases needed to expose and use the cluster via CaaS, VMware virtual machines were used extensively. Diagram 15 shows the complete test environment with the contents of each virtual machine.

All virtual machines have 512 MB of virtual memory and all (expect the client VM) ran the Windows Server

2003. The client virtual machine ran Ubuntu 9.04. All Windows virtual machines run .NET 2.0; the CaaS virtual machine runs .NET 3.5.

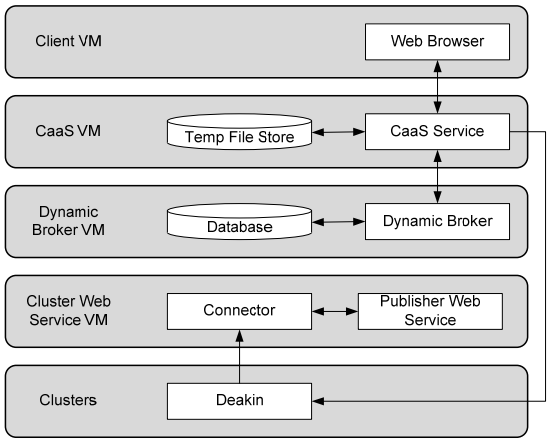

Diagram 15: Complete CaaS Environment

The first virtual machine is the Publisher Web service virtual machine. It contains the cluster Connector, the Publisher Web service and all required software libraries to make their execution possible. The Broker virtual machine contains the Broker and its database. The CaaS virtual machine houses the CaaS Server and a temporary data store. To improve reliability, all file transfers between the cluster and the client are cached.

5.3 Cluster and Job Specifications

The cluster used in the proof of concept consists of 20 nodes plus two head nodes (one running Linux and the other running Windows). Each node in the cluster has a two Intel Cloverton Quad Core CPUs running at 1.6 GHz, 8 Gigabytes of memory, 250 Gigabytes of data storage and all nodes are connected via Gigabit Ethernet and Infiniband. The head nodes are the same except they have 1.2 Terabytes of data storage.

For our tests, an mpiBLAST application was used. mpiBLAST is a distributed application used to find gene sequences in genome databases: a common testing bioinformatics. When running mpiBLAST, at least two files are needed: the database and a sequence file. To simplify testing, we placed the database on the cluster and only uploaded the sequence file.

5.4 Experiments and Results

5.4.1 Publication

Due to space limitations, the process of publishing a cluster and learning of its dynamic attributes via the Broker are omitted. However, this idea was well tested and the outcomes documented in (Brock and Goscinski 2009a).

5.4.2 Discovery and Selection

Diagram 16 shows the workflow behind the cluster‘s discovery. The required cluster information was submitted to the CaaS Service (1). We requested a cluster with at least 20 nodes, each with at least 6 Gigabytes of free memory and all nodes running a Linux system. The CaaS Service then contacted the Broker with the specified requirements (2). The Broker queries its database for matches (3) and returns the results to the CaaS Service (4). The cluster matches are returned in (4). If ‘information overload’ is in place, the CaaS through its Results Organizer takes the matches, tabulates them and returns to the client (5).

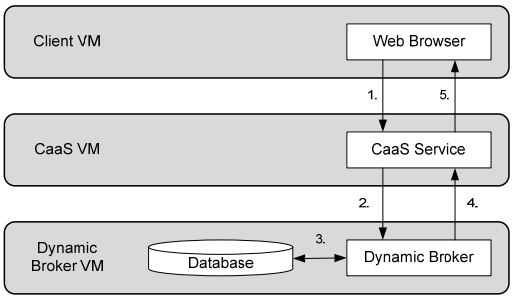

Diagram 16: Cluster Discovery

5.4.3 Job Submission

Diagram 17 shows the workflow behind submitting a job to the cluster. First, the cluster job is specified and submitted to the cluster (1). During submission, parameters such as the name of the job and its required execution time are specified. Along with the job parameters, our job script file and data files contained in a zip file are also submitted. To improve reliability, all files are kept in a temporary file store (2). Once all the files have been transferred, the CaaS service transfers the files to the chosen cluster (3). After all files are transferred, the scheduler is invoked and the outcome returned to the client (4).

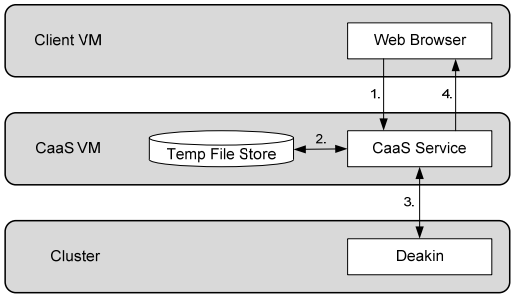

Diagram 17: Job Submission

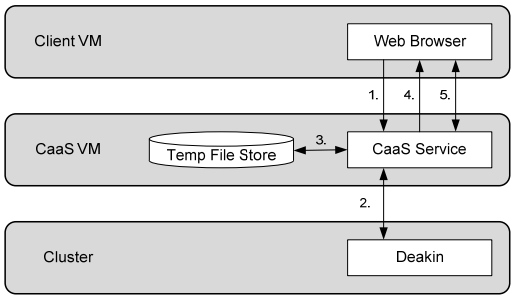

Diagram 18: Result Collection

5.4.4 Collection

The final experiment commences once the Job Monitoring Web page (Diagram 12) reports that the job has finished. Using the Web browser, a request to collect

the results is to be made (1 in Diagram 18) and the CaaS Service retrieves the result file to be stored in the File Store (2-3).

Once the file is transferred, it is expected that the CaaS Service show the Result Collection page (Diagram 13) and provide hyperlinks to download the result files over HTTP (4). After downloading the results files, confirmation is sent to the CaaS service (5) that removes the file from the File Store and instructs the cluster to remove its copy (6).

5.4.5 Results

Experiment 1: Discovery: As stated in Section II, cluster clients still have to discover clusters first. This is a problem as there is no discovery system for clusters. Hence, our first experiment was to see if a cluster was easily discovered through our CaaS Technology.

Diagram 19 shows the cluster discovery page populated with the requirements for our first mpiBLAST job. For this experiment, we only needed four cluster nodes, each with 8 Gigabytes of memory and did not have more than 10% utilization of their CPUs.

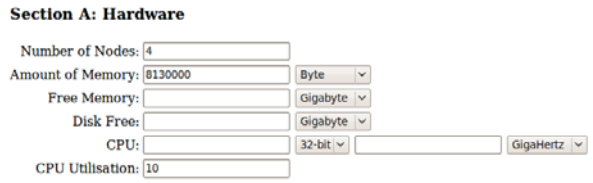

Diagram 19: Specifying Cluster Requirements

After specifying our requirements, the Web page passed the requirements to the CaaS Service for processing. As stated in Section III, the Dynamic Broker is first queried with the requirements and the results then processed by the Results Organiser. Diagram 20 shows a formatted version of the organized results.

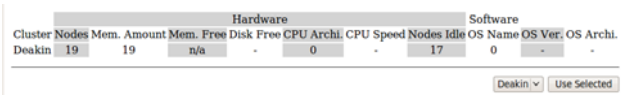

Diagram 20: Cluster Match Results

While the Web page was designed to have ticks, it was decided late in development to use numbers instead. The reason for this was the numbers gave a clearer view on how each requirement was satisfied by each cluster.

Overall, this experiment was a complete success. We had our cluster exposed via the Publisher Web service, and easily discovery the cluster through the CaaS Service. This is a significant contribution as a discovery service now exists to allow human operators to easily locate a required cluster.

Experiment 2: Job Submission: As stated in Section II, clusters can vary in how they are built. Specifically, not all clusters run the same middleware. Hence this experiment was carried out to see if the middleware specifics could be hidden.

Even though our cluster consisted of 20 nodes, only 19 were active at the time of testing. Furthermore, for this test, we only wanted to use two cluster nodes. Thus, a successful outcome of this test had to show only two cluster nodes being used and not the whole cluster.

Diagram 21 shows the Job Specification Web page where our first mpiBLAST job was specified. For this test, we specified an mpiBLAST launch script, a sequence file to compare against a mouse database and the name of an output file was specified.

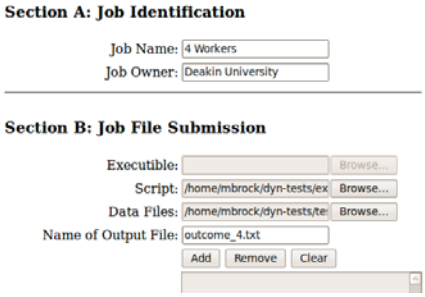

Diagram 21: Specifying the Job

After completing the Job Specification Web page, our script and test file were uploaded to the CaaS VM and then transferred to the cluster. After all files were transferred, the scheduler (GridEngine) was invoked. Diagram 22 shows the resulting Job Monitoring Web page.

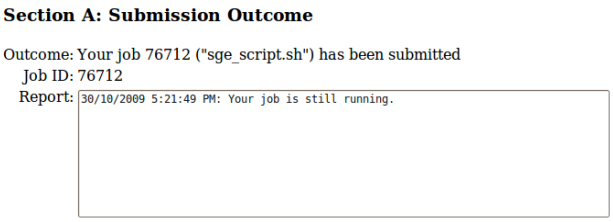

Diagram 22: Monitoring the Progress of the Job

As the hostfile for our job was generated dynamically, we could not tell its contents during the submission process. After the completion of Experiment 4, the contents of the host file would be checked.

Experiment 4: Results Collection: Just as how clients need to be able to easily upload their jobs to clusters, they need to be able to download any result or error files.

To know when our job finished, the Job Monitoring Web page was refreshed a few times before indicating that the second job had completed (Diagram 23).

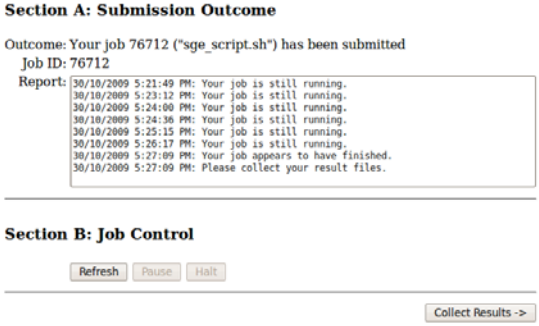

Diagram 23: Completion Notification

Diagram 24 shows the Results Collection Web page

with a hyperlink to our result file. By clicking the link, we were able to download our results just like any other file on the Web.

With the completion of this experiment, our CaaS Technology with its Web pages, as a whole was proven successful. We were able to expose a cluster via Web services, discover it, run multiple jobs without any clashes, and easily get result data back. All of this was done with no computing expertise at all.

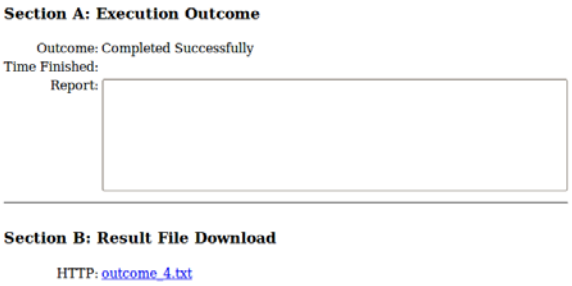

Diagram 24: Collecting Job Results

With the completion of the job execution, we needed to examine the hostfile used to influence how the job was scheduled to the cluster. As Figure 2 shows, only two nodes were listed. This is a significant advancement as not only was the cluster made easy to use, but the client was also reserved the nodes available at the time of his or her request.

```
west-lin (mrab) 1005 $cat hostfile
west-07
west-16
west-19
west-12
```

Figure 2: Hostfile Contents

6 Conclusion

We have achieved the goal of this project by the development of a technology for building a Cluster as a Service (CaaS) using the RVWS framework. Through the combination of dynamic attributes, Web service's WSDL and Brokering, we successfully created a Service that quickly and easily published, discovered and selected a cluster, allowed to specify a job and execute it, and finally got the result file back.

The proposed technology forms a bridge between the dynamic attribute and Web service based Resources Via Services (RVWS) framework and a high level abstraction of clusters in clouds in the form of a CaaS was specified. This outcome could have significant impact on cloud computing.

7 References

Amazon (2007) Amazon Elastic Compute Cloud. Accessed 1 August 2009, http://aws.amazon.com/ec2/.

Amazon (2009) EC2StartersGuide, https://help.ubuntu.com/community/EC2StartersGuide

Apache (2009) Hadoop, Accessed 1 August 2009, http://hadoop.apache.org

M. Brock & A. Goscinski (2008a) Publishing Dynamic State Changes of Resources Through State Aware WSDL. Int. Conf. on Web Services (ICWS) 2008. Beijing.

M. Brock and A. Goscinski (2008b) State Aware WSDL. Sixth Australasian Symposium on Grid Computing and e-Research (AusGrid 2008). Wollongong, Australia, 82, 35-44, ACM.

M. Brock and A. Goscinski (2009) Attributed Publication and Selection for Service-based Distributed Systems. Int. Workshop on Service Intelligence and Computing (SIC 2009). Los Angeles, IEEE.

P. Chaganti 2008. Cloud Computing with Amazon Services, http://ibm.com/developerswork/architecture/library/ar-cloudaws3/

E. Christensen, F. Curbera, G. Meredith and S. Weerawarana (2001) Web Services Description Language (WSDL) Version 1.1. Updated 15 March 2001, Accessed, http://www.w3.org/TR/wsdl.

K. Czajkowski, et al. (2004) The WS-Resource Framework. 5 March 2004. http://www.globus.org/wsrf/specs/ws-wsrf.pdf.

J. Dean and S. Ghemawat (2004) MapReduce: Simplified Data Processing on Large Clusters. Sixth Symposiuym on Operating System Design and Implementation, San Francisco, CA, December, 2004.

T. Gal (2005) sharpSsh - A Secure Shell (SSH) library for .NET. Updated 30 October 2005, Accessed 1 March 2009, www.codeproject.com/KB/IP/sharpssh.aspx.

Google (2009) App Engine. Accessed 17 February 2009, http://code.google.com/appengine/.

A. Goscinski, M. Hobbs and J. Silcock (2002) GENESIS: An Efficient, Transparent and Easy to Use Cluster Operating System. Parallel Computing, Vol. 28 (2002), No. 4, April, 557-606.

S. Jha, A. Merzky, G. Fox (2009) Using Clouds to Provide Grids Higher-Levels of Abstraction and Explicit Support for Usage Models, Version: 1.0, GFD-I.150.

Microsoft (2009) Azure. Accessed 5 May 2009, http://www.microsoft.com/azure/default.mspx.

M. Papazoglou and W-Jan van den Heuvel (2007) Service oriented architectures: approaches, technologies and research issues, The VLDB Journal (2007) 16:389–415.

M. Papazoglou (2008) Web Services: Principles and Technology, Prentice Hall.

Salesforce (2009) Accessed August 1 2009, www.salesforce.com

VCL (2008) http://vcl.ncsu.edu/.