Provider-Independent Use of the Cloud

Terence Harmer, Peter Wright, Christina Cunningham, and Ron Perrott

Belfast e-Science Centre, The Queen's University of Belfast, Belfast BT7 1NN, UK
{t.harmer,p.wright,c.cunningham,r.perrott}@besc.ac.uk

Abstract. Utility computing offers researchers and businesses the potential of significant cost-savings, making it possible for them to match the cost of their computing and storage to their demand for such resources. A utility compute provider enables the purchase of compute infrastructures on-demand; when a user requires computing resources a provider will provision a resource for them and charge them only for their period of use of that resource. There has been a significant growth in the number of cloud computing resource providers and each has a different resource usage model, application process and application programming interface (API)—developing generic multi-resource provider applications is thus difficult and time consuming. We have developed an abstraction layer that provides a single resource usage model, user authentication model and API for compute providers that enables cloud-provider neutral applications to be developed. In this paper we outline the issues in using external resource providers, give examples of using a number of the most popular cloud providers and provide examples of developing provider neutral applications. In addition, we discuss the development of the API to create a generic provisioning model based on a common architecture for cloud computing providers.

1 Introduction

Utility computing offers researchers and businesses the potential of significant cost-savings in that it is possible for them to match the cost of their computing and storage to their demand for such resources. Thus, a user needing 100 compute resources for 1 day per week may purchase those resources only for the day they are required. A utility provider may also enable the allocation of storage to hold data and network bandwidth for access to a user's applications and services. Again, this enables a user to match their storage and network capacity to meet their needs. More generally, it is possible for a business or researcher to work without owning any significant infrastructure and rely on providers when they need a compute infrastructure.

On-demand provision of resources to users has been around for some time with grid computing services, such as the UK National Grid Service[1] and the US TeraGrid[2], but they have largely been job focused rather than service-focused. In these infrastructures the goal is to optimise the compute jobs that are being requested by users—delivering the fastest compute for users and making the best use of the fixed compute infrastructure.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 454–465, 2009.

With the success of the cloud computing paradigm[3], dynamic infrastructures have become popular. As a result of the increased commercial demand, there are more providers offering infrastructure on-demand—such as Amazon EC2[4], Flexiscale[5], AppNexus[6], NewServers[7] and ElasticHosts[8]. Sun have also announced plans to become a cloud provider/. Most vendors provide wholly virtual servers where a user provisions an infrastructure by providing a virtual machine (VM). Within a VM a user can bundle applications, services and data together in the same package. VMs usually give reduced performance when compared to the native compute resource but do enable the provider to share its resources in a secure and reliable manner. However, there are vendors that offer physical machines on-demand, such as NewServers, that are suited to high computation/IO applications.

1.1 Why Are Resource Providers Interesting to Us?

At the Belfast e-Science Centre, our interest is in creating dynamic service-focused infrastructures which are primarily in the media[9][10] and finance[11] sectors. Media and finance are challenging domains to develop applications for in that:

- they require high computational throughput with millisecond-based quality of service—for example to create a particular video format for a user as they request to view it;
- they have high security requirements by the nature of the value of the data that is processed and the legislation that regulates the sectors—for example, in communicating the details of the ownership of shares; and
- they are subject to rapid peaks and troughs in demand—for example, in a volatile day trading volumes can double in minutes as traders react to news (and other trader behaviour!)

We have developed a framework of services that enables our applications to be deployed and configured on demand within a fixed infrastructure. In developing our application it quickly became clear that the peaks and troughs in demand could best be accommodated using on-demand providers to supplement a fixed infrastructure. More recently, we have created and deployed services for commercial and research partners where none of the infrastructure was owned by the user and instead relied entirely on cloud resource providers.

We have worked on extending our dynamic deployment services from supporting only owned and fixed compute resources to include resources allocated on demand by cloud resource providers. To do this our support software needed a way to provision resources, configure those resources, deploy software onto the resources and, when its task was complete, discard the allocated resources–all whilst providing accounting for the resources that were used. Investigation turned up no standard cloud API; the closest is the Eucalyptus[12] open source provider codebase, however it is a clone of the EC2 API. It does not seem likely that a standard model for cloud computing providers will appear soon given the speed at which cloud vendors are innovating and enhancing their services. In addition

(given the competition between providers) it seems unlikely at this stage that many providers would believe it in their interest to create or comply with a common API.

1.2 What Is Involved in Using an On-Demand Cloud Provider?

Cloud computing is an increasingly compelling prospect for researchers, start-ups and large organisations, either as a substitution for owned hardware or to bolster their existing infrastructure in high-load scenarios. Multiple providers are emerging, offering different features such as hardware load-balancing, physical (rather than virtual) resources, and private VLANs capable of multicast. These are attractive because of the low cost of use when compared to buying, refreshing and supporting a large infrastructure.

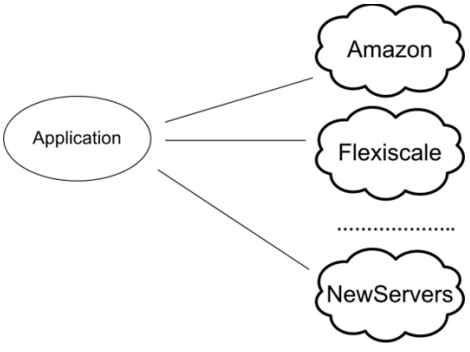

Fig. 1. A traditional multi-cloud application

It is in the interest of providers to have their own APIs as this simplifies their development task, fitting (perfectly) their business model and their implementation. However, these varying APIs complicate service clouds, such as our media and financial applications, that wish to sit on top of multiple cloud vendors that provide physical resources. Ideally, these applications should be written in a provider agnostic fashion using the cheapest resources when they are required and switching between providers when there are cost savings in doing so (Fig. 1). Unfortunately, the application must currently be aware of the provider being used, their utility model and their API. This makes developing applications that can use multiple providers difficult to write and maintain because you must be aware of the benefits and drawbacks of particular providers.

In addition, APIs are also evolving with new functionality, operating systems, SLA terms, pricing and resource specifications. Keeping up-to-date with the latest API changes can be time-consuming and can mean that applications are fixed to particular vendors and compute resources.

As an illustration of the complexities faced by various developers working in the cloud, we consider a simple and basic task in using a cloud provider – instantiate the cheapest machine available and display its instance identity and

IP address. We present Java code for 3 providers: Amazon EC2, Flexiscale and NewServers. (The APIs have been simplified for readability, however real use is nearly identical to what is presented here).

NewServers. NewServers provides on-demand access to native (*bare-metal*) compute resources. The simplicity and minimalism of their API is interesting to us however, since it provides a good model for small home-grown academic clouds.

```
String size = 'small'; // 2.8GHz Xeon, 1GB RAM, 36GB disk
String os   = 'centos5.2';

// create & start
Server server = newservers.addServer(size, os);

String instId = server.getId();
String ip     = server.getPublicIP();

System.out.printf('Instance %s has IP %s\n', instId, ip);
```

From their API we know that "small" is their most basic machine with a 2.8GHz Xeon and 1GB of RAM. We choose that, deploying a CentOS 5.2 image onto it. Once we call *addServer*, the server is created and started and its details are displayed. We can query their API for a list of available template machines and operating systems and their costs; we've elected not to do that in this case.

Flexiscale. Flexiscale is a cloud resource provider that supports deployment of a range of VM types. It provides near native network support capability: providing VLANs and multicast, for example, with high levels of VM quality of service and support for automatic re-deployment in the event of failure. The objective is to create an easy-to-use mechanism with near native machine capability and automated resilience.

```
String instId = 'MyNewServer'; // User-assigned id

// Get default hosting package, VLAN, and OS
Package fxPackage = flexiscale.listPackages().get(0);
Vlan fxVlan = flexiscale.listVlans().get(0);
OSImage os = new OSImage(27); // Ubuntu

Server s = new Server();
s.setName(instanceId);
s.setDisk_capacity(4); // 4GB disk
s.setProcessors(1); // 1 CPU
s.setMemory(512); // 512MB RAM
s.setPackage_id(fxPackage.id);
s.setOperatingSystem(os);
s.setInitialPassword('changeme');

flexiscale.createServer(s, fxVlan); // allocate
flexiscale.startServer(instId, 'no notes'); // start

String ip = flexiscale.listServers(instId)[0].getIP(0);

System.out.printf('Instance %s has IP %s\n', instId, ip);
```

In the above code we acquire our hosting package, default VLAN and a pointer to the Ubuntu system image. We create a new server with 1 CPU, 512MB RAM and 4GB local disk. Once created, we start the machine and then request its details from the API to determine the IP address. With Flexiscale we must know the valid CPU, RAM, OS combination—this is available in a table on their API site. With the Flexiscale API we can allocate our VMs to specific VLANs, enabling applications to have secure and good low-level control of communications. Flexiscale must create a machine image when requested and so turnaround time is about 10 minutes and often it is useful to create a resource in advance of when it might be required by an application. This can complicate scaling to meet changes in application demand.

Amazon EC2. Amazon is the largest and most widely known cloud compute provider that uses a proven infrastructure (Amazon's own infrastructure) and enables rapid scaling of resources by a user. Amazon provides hosting capabilities in the US and in Europe.

```
RunInstancesRequest req = new RunInstancesRequest();
req.setImageId('ami-1c5db975'); // Ubuntu
req.setKeyName('someAuthKey'); // preloaded root ssh key
req.setPlacement('us-east-1a'); // the datacentre
// 1GHz xeon, 1.7GB RAM, 150GB disk
req.setInstType('m1.small');

// allocate & start
Reservation res = ec2.runInstances(req).getReservation();

String id  = res.getInstance()[0].getInstanceId();
String ip  = res.getInstance()[0].getPublicDNS();

System.out.printf('Instance %s has IP %s\n', id, ip);
```

With EC2, we specify the exact Amazon Machine Image for an Ubuntu operating system, an authentication key to secure SSH access, a datacentre ("us-east-1a") to place our VM in, and the name of the cheapest template ("m1.small"). EC2 runs as a large compute cloud that allocates resources in different physical datacentres (EC2 sub-clouds). With EC2 you must initially pre-store a VM image with Amazon and record Amazon's unique identity for that image. When deploying you must be careful to ensure that each VM compute resource you create is within the same datacentre sub cloud—this will improve performance but also avoids significant charges for inter-cloud bandwidth. It is also necessary to know the type of OS instance that is permissible—this is available from a table on their API site. EC2 boots VMs very quickly, giving a low turnaround time and enabling rapid scaling of infrastructure.

1.3 Lessons from Existing Cloud Provider APIs

As we can see from the previous examples, the initialisation and use of a resource in a resource provider varies significantly in implementation details. Some providers let you specify exact machine configurations while others have sets

of configurations to choose from when requesting a resource. Some allow user-assigned resource identifiers while others allocate an identifier for you. The usage models are different also. Some models have the concept of a "stopped machine"; others consider stopping to be analogous to discarding.

To create a generic, cloud-provider independent application, a simple consistent API is necessary that provides a consistent and flexible provider API and resource usage model. Such an API would allow users to specify requirements (such as RAM size, CPU performance, disk space) and enable a rich set of filters that match requirements with provider capabilities. In addition, the model would permit a consistent and simple resource model that attempts to hide the particular details of resource providers.

2 Our Abstraction Layer

Our need for an abstraction layer (Fig. 2) to provide a generic cloud provider was rooted in a number of our project applications which created highly dynamic service-based infrastructures[3][9][10] which

- were deployed and managed dynamically;
- used autonomic management services to control and monitor the infrastructure;
- that scaled to match user demand; and
- employed location-aware services to optimise user access and comply with data processing regulations.

Based on our experience of using cloud providers and their evolution over time (as well as where we see them going in the future) we formed an outline usage model which we see as common to them all:

1. Find viable compute resources (based on properties such on CPU speed, memory, disk capacity, etc.)
2. Select the best match from those available (based on specification/price and the trade-off from the options available from the resource vendors)
3. Configure compute resource parameters (such as OS, firewall, storage, etc)
4. Instantiate a box using that configuration

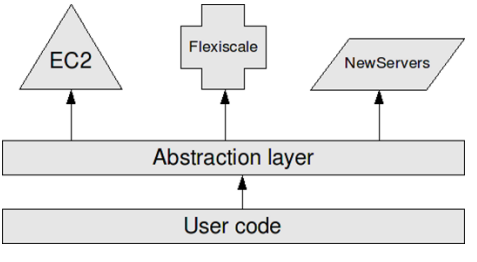

Fig. 2. Multi-cloud applications with our abstraction layer

5. User does some work on the box
6. Discard the compute resource

This model is intentionally simple and attempts to reflect an ideal for the application user who wishes to specify resources without knowing the internal behaviour of the resource provider. Currently, when developing cloud applications, *Step 1: Find viable compute resources* and *Step 2: Select the best match*, are performed manually by the development/deployment team in creating/deploying the application, limiting the extent to which an application can dynamically take advantage of new resource types and requiring the implementation team to refine an application to take advantage of benefits of new vendors or machine types/pricing.

Fig. 3. Simple management model

In essence, our workflow (Fig. 3), reflecting our usage model, is find, instantiate, manage and discard. There is an increased focus on predictability within the cloud provider community to encourage traditional business IT to use cloud technology. For example, Amazon has recently-announced a Service Level Agreement (SLA) to bring them in line with other cloud providers. These SLAs aim to provide, for example, a quantifiable up-time for their resources and penalty clauses if this is not met. In addition, a large-scale EU project, SLA@SOI[13], aims to make SLA negotiation a core part of its resource provisioning model with resource providers chosen to satisfy SLA requirements and being expected to enforce SLA requirements.

Fig. 4. Expanded management model

To allow for this concept of predictability we add a *reserve* stage (Fig. 4) to allow users to indicate their near-term interest in allocating some resource–that may incur a small cost with some providers. This smoothes out the differences between providers who do have the concept (e.g. Flexiscale) and those who do not (e.g. Amazon) and will in the future allow calls to enable the provider to keep ahead of demand by requiring reservation and potentially minimising the number of hosting machines switched on at any given time.

We see the role of the resource provider, once the resource has been allocated, as completely behind the scenes—communication and control of the allocated resource is identical to every other machine in the user's datacentre. Thus, we

talk about the dynamic resources as "boxes" to indicate that the vendor is providing nothing more than a machine with an OS preinstalled that is configured according to the user's specification.

So, with our model resource providers are called *Box Providers*; they offer various *Box Templates* that describe properties of the boxes they can allocate. These templates define the type of box, how it can be configured and define the pricing structures and usage policies for the box. The templates can then be queried by generic questions like "How much would 48 hours of uptime with 1GB of internet transfer up and 10GB down cost me?" or specific questions for particular providers "How much is 50GB of transfer to Amazon S3 going to cost me?". The use of templates completely isolates programs from the variations in pricing and available resources, making the process naturally extensible. A user's focus is on the type of resource they require with the allocation process finding a provider to satisfy the generic resource request and the non-functional requirements (such as price) that might be appropriate.

Once a template has been chosen it may be reserved with particular configuration properties—for example, setting the required OS, firewall rules, storage mounts, etc. At a later stage the reservation can then be instantiated, giving a *Box Instance* which can be managed, queried and finally discarded.

Allocating a Resource with Our Abstraction. Using our layer, the previous examples do not change much:

```
EC2Credentials account = ...;
BoxProvider provider   = new EC2BoxProvider(account);

// Use the cheapest template this provider knows about
BoxTemplate templ = BoxTemplateHelper.cheapest(provider);

// Deploy 1 ubuntu box with quickstart defaults
BoxConfiguration os = BasicBoxConfig.getUbuntu(templ);
ReservedBox res     = provider.reserve(templ, 1, os);

// Create & start the machine
BoxInstance inst = provider.instantiate(res);

String instId = inst.getId();
String ip = inst.getPublicIp();
System.out.printf('Instance %s has IP %s\n', instId, ip);

provider.discard(inst); // throw the machine away
```

As in our previous examples, the process is the same, however since we are expressing things generically it would be trivial to swap out our EC2BoxProvider for a FlexiscaleBoxProvider and, without any other code changes, allocate machines using Flexiscale. Given this view, we can consider a more interesting scenario – we want a 5-machine cluster, each with 1GB of RAM and 160GB of disk; we will be uploading a 1GB file for some light processing by our mini-cluster into a 10GB output file which will then be retrieved within 2 days. The calculation will be very integer-heavy so we'll use a Java integer performance benchmark as a fitness function.

```
BoxProvider provider = new BoxProviderUnion(getEC2(),
                                            getFxscale(),
                                            ...);
// 1GB of RAM, 160GB disk
Restriction r1 = new MemoryRestriction(1024);
Restriction r2 = new LocalDiskRestriction(160);

BoxTemplate[] templates = provider.find(r1, r2);

// Find cheapest for 48 hours with 1GB up, 10GB down
int hoursOn       = 48;
int mbUpload      = 1024, mbDownload = 10240;
BoxFitness benchmark = CostWeighting.JAVA_INTEGER;

BoxTemplate best = BoxTemplateHelper.getBestFor(
                     benchmark, hoursOn,
                     mbUpload, mbDownload);

// Deploy 5 of the boxes with an Ubuntu OS
BoxConfiguration os = BasicBoxConfig.getUbuntu(best);
ReservedBox res     = provider.reserve(best, 5, os);

BoxInstance[] insts = provider.instantiate(res);

for (BoxInstance instance: insts) {
  String id       = instance.getId();
  String provider = instance.getProvider.getName();
  String ip       = instance.getPublicIp();

  System.out.printf('Instance %s on %s has IP %s\n',
     id, provider, ip);

  // If they're EC2, print out the datacentre too
  if (instance instanceof EC2BoxInstance) {
    String zone = ((EC2BoxInstance)instance).getZone();
    System.out.printf(' In datacentre %s\n', zone);
  }
}
```

In the above example we set up a meta-provider, *Union Provider*, which knows about EC2, Flexiscale and NewServers—in effect we are setting those providers we have established accounts with. We call *find* with restrictions, the results from all three providers are combined – making the selection of the cheapest provider completely transparent to our code. The cheapest template is selected using, as our fitness function, an estimate of the cost of 48 hours of use with 1GB uploaded and 10GB downloaded. Once complete, a generic helper, *BasicBoxConfig*, is used to find an Ubuntu operating system in the providers list of operating system images. We then reserve and instantiate 5 boxes as in the previous example.

Our generic selection process means that as soon as the abstraction layer knows about new options (whether through the vendor API or in its own code), the user can take advantage of this with no changes to their code. In addition, if the users Flexiscale account is out of credit they can easily (and transparently) failover to their EC2 account. The fitness function uses a table of performance values (cached in the provider implementations) we have computed for each template for various instance types from providers; this allows us to simply

specify the *performance* we want rather than worry about the differences between various provider virtualisation platforms.

3 Use Case – A Media Service Cloud

The cloud provider library described here has been used in support of the PRISM media service cloud[9] to assist in managing a dynamic infrastructure of services for on-demand media provision of transcoding, storage and federated metadata indexing (Fig. 5). The PRISM infrastructure supports access to BBC content from a set-top box, web browser and from a range of media devices, such a mobile phones and games consoles. The infrastructure is currently being used for a large-scale user trial that provides access to all of the BBC's content.

The PRISM system uses resources within the Belfast e-Science Centre, British Telecom and BBC datacentres to provide core infrastructure. When user demand for content is high, PRISM services are hosted on utility resources from 3^{rd} party resource providers. The infrastructure is managed in a uniform way as a collection of cloud providers—partner clouds that are the core infrastructure (BeSC, BBC and BT) and utility providers that can be called upon for extra resources. The PRISM application is written in a way similar to the examples outlined above—initially the baseline services are allocated from a cloud provider. When the services within PRISM detect that the system is overloaded, additional resources are deployed from a suitable provider. The cost model used in selection ensures that, initially, core infrastructure is selected; as this is used and no resources are available in the core clouds, 3^{rd} party clouds are used to provide supporting services. This model has proven to be highly flexible—it has meant that the loss of a core infrastructure cloud was automatically compensated for with increased utility provider use.

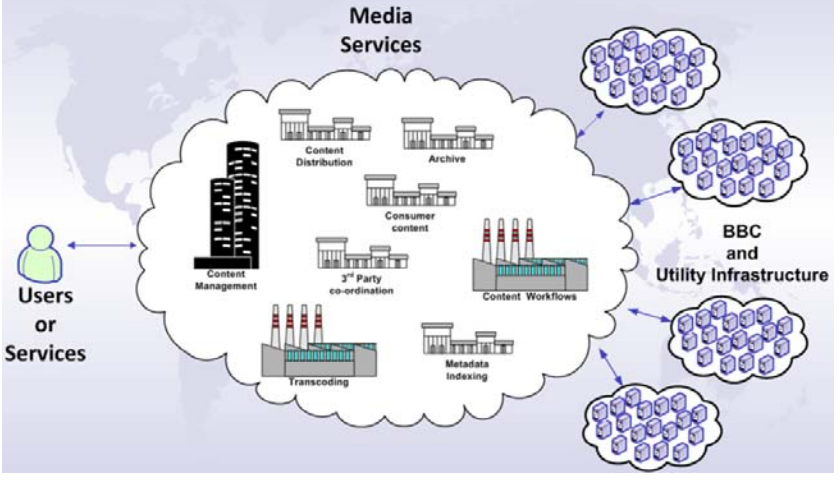

Fig. 5. PRISM Media Services Cloud

3.1 Evaluation of Use within PRISM

This abstraction layer, in combination with the supporting dynamic service deployment framework, has saved a large amount of time and development effort in PRISM, as well as dynamic infrastructure costs. The restrictions supplied put a heavy weight on the amount and cost of bandwidth to existing machines, allowing the most cost-effective locations to be chosen (internal infrastructure, where available). This has been demonstrated twice with the PRISM infrastructure:

1. Building contractors cut through our dedicated fibre optic link to the BBC Northern Ireland datacentre; the system dynamically failed over to free infrastructure in the BeSC datacentre
2. The SAN attached to the PRISM transcoding cloud failed; the framework failed over to EC2 for the 2 days it required to correct the problem, costing $51.32 per day (67% of the cost was the transcoder VMs), however this came with a performance drop due to content being served over a 250mbit line

4 Conclusion and Future Work

As the cloud computing landscape gains acceptance in research and established businesses, users will be looking for cloud platform independence. This is crucial if cloud computing is to fulfil its promise of providing a robust infrastructure. Amazon is currently the major player, although its competitors are offering compelling feature sets for researchers and enterprises looking to deploy existing applications with minimal changes.

Our main aim was to simplify the allocation, management and discarding of on-demand resources from a multitude of providers. The initial generic API provides this functionality cleanly and efficiently and has back-ends for the APIs of Amazon EC2, Flexiscale, NewServers and the BeSC Service Hosting Cloud. It has been used directly for applications that want fine-grained control over their allocated VMs and by our service deployment framework that is available to UK NGS users. It has enabled our developers to be more experimental with the design of our own Service Hosting Cloud, as we must only maintain the binding to our abstraction layer to ensure our internal services can continue to use the service uninterrupted.

As Amazon and other vendors improve their feature sets we are seeing yet more common areas – in particular, customisable SAN mount, static IP address management, and resource resilience and scaling. The groundwork for the inclusion of these features has been laid, however they were not the main focus of its creation so they will be incorporated at a later date.

References

1. Wang, L., Jinjun Chen, W.J.: Grid Computing: Infrastructure, Service, and Applications. CRC Press, Boca Raton (2009)
2. Teragrid (2008), `http://www.teragrid.org`

3. Wladawsky-Berger, I.: Cloud computing, grids and the upcoming cambrian explosion in IT (2008)
4. Amazon, Inc.: Amazon Elastic Compute Cloud (2008), http://aws.amazon.com/ec2/
5. Xcalibre, Inc. (2008), http://www.flexiscale.com
6. AppNexus, Inc. (2008), http://www.appnexus.com
7. NewServers, Inc. (2008), http://www.newservers.com
8. ElasticHosts, Ltd. (2008), http://www.elastichosts.com
9. Perrott, R., Harmer, T., Lewis, R.: e-Science infrastructure for digital media broadcasting. Computer 41, 67–72 (2008)
10. Harmer, T.: Gridcast–a next generation broadcast infrastructure? Cluster Computing 10, 277–285 (2007)
11. CRISP: Commercial R3 IEC Service Provision (2008), http://crisp-project.org
12. Eucalyptus Systems: The Eucalyptus Open-source Cloud-computing System (2008), http://open.eucalyptus.com/
13. SLA@SOI Project (2008), http://sla-at-soi.eu