AppScale: Open-Source Platform-As-A-Service

Chandra Krintz Chris Bunch Navraj Chohan
Computer Science Department
University of California
Santa Barbara, CA USA
UCSB Technical Report #2011-01
January 2011

ABSTRACT

In this paper we overview AppScale, an open source cloud platform. AppScale is a distributed and scalable cloud runtime system that executes cloud applications in public, private, and hybrid cloud settings. AppScale implements a set of APIs in support of common and popular cloud services and functionality. This set includes those defined by the Google App Engine public cloud as well as those for large-scale data analytics and high-performance computing.

Our goal with AppScale is to enable research and experimentation into cloud computing and to facilitate a ``write once, run anywhere'' programming model for the cloud, i.e., to expedite portable application development and deployment across disparate cloud fabrics. In this work, we describe the current AppScale APIs and the ways in which users can deploy AppScale clouds and applications in public and private settings. In addition, we describe the internals of the system to give insight into how developers can investigate and extend AppScale as part of research and development on next-generation cloud software and services.

INTRODUCTION

AppScale is a scalable, distributed, and fault-tolerant cloud runtime system that we have developed at the University of California, Santa Barbara as part of our research into the next generation of programming systems (Chohan, et al., 2009; Bunch, et al., 2010; Bunch (2) et.al, 2010). In particular, AppScale is a cloud platform, i.e. a platform-as-a-service (PaaS) cloud fabric that executes over cluster resources. The cluster resources underlying AppScale can be managed with or without virtualization, e.g. Xen, KVM, or via popular cloud infrastructures including Amazon EC2 and Eucalyptus.

The AppScale platform virtualizes, abstracts, and multiplexes cloud and system services across multiple applications, enabling *write-once, run-anywhere (WORA)* program development for the cloud. In addition to simplifying application development and deployment using cloud systems and modern distributed computing technologies, AppScale brings popular public cloud fabrics *on-premise*, i.e., to private clusters. To enable this, we emulate key cloud layers from the commercial sector, and do so (i) to engender a user community, (ii) to gain access to and to study real applications, and (iii) to investigate the potential implementations of and extensions to, public cloud systems using open-source technologies.

The initial APIs that AppScale emulates are those of Google App Engine. Google App Engine is a public cloud platform (a complete software stack) that exports scalable and elastic web service technologies via well-defined APIs to network-accessible applications. These APIs implement messaging, key-value data storage, map-reduce, mail, and user authentication, among other services. The platform facilitates easy asynchronous multi-tasking, web server support, elasticity, and resource management. Using Google App Engine, developers debug and test their programs using an open-source software development kit (SDK) provided by Google that implements non-scalable versions of the APIs. Developers then upload their code and data to Google clusters and use Google cluster resources and services on a free (up to some fixed set of per-resource quotas) and *pay-per-use* (resource rental) basis.

AppScale is API-compatible with Google App Engine. As such, applications that execute on Google App Engine can also execute on AppScale without modification, using private cluster resources or public cloud infrastructures. Our API and service implementations are scalable, distributed, fault tolerant, and facilitate high-performance and highly available service access. To enable this, we leverage mature open-source technologies as extensively as is possible. AppScale implements multiple language runtimes (Java, Python, Ruby) as application frontends and a wide range of open source database technologies (key-value and relational) as options for its internal, system-wide datastore. AppScale is not a replacement for Google App Engine or any other public cloud technology. Instead, AppScale is a robust and extensible research infrastructure and private cloud platform that provisions the resources it is allocated scalably across multiple applications.

The AppScale platform also exports services and APIs other than those provided by Google App Engine. These technologies are important for application domains beyond those of web services, including data analytics and data and computationally intensive applications. AppScale exports these technologies as services, i.e. AppScale "service-izes" libraries, tools, and software packages, including MapReduce, X10, R, and MPI. In addition, as it does for the Google App Engine APIs, AppScale provides automatic configuration, deployment, and distribution for these technologies and facilitates their elasticity, load balancing, and fault-tolerance.

Since AppScale provides a software layer between applications and the implementations of cloud APIs and services, we are also able to employ existing cloud services for the implementations. As such, AppScale is a *hybrid* cloud platform -- a programming system through which an application can access services from different clouds (a mix of public and private) at different times or concurrently. Developers can use this cloud hybrid support to move data between clouds, e.g., for data analytics, backup of application state and data, to reduce public cloud costs (to use lower-cost alternatives), and to "fall out" from a private cloud with limited resources to a public cloud, on-demand. Developers can also employ AppScale's hybrid support to provide a greater degree of fault-tolerance and availability: in the unlikely but possible scenario in which a public cloud fails or becomes inaccessible, users can still access their application via other cloud(s) – public or private – that concurrently host it.

The AppScale Internals

Figure 1 depicts the AppScale design. AppScale utilizes the standard three-tier web deployment model. The load balancer processes incoming requests from users and routes them to an application server, which may be running across a number of remote hosts. The application layer consists of a wide range of services that simplify application development and deployment by precluding the need for the reimplementation of common tasks. Behind the application layer is a data management layer that provides persistent storage for the applications that execute in the AppScale cloud. AppScale supports many database and datastore systems to implement this system-wide database service, each of which employs different designs and implementation trade-offs. AppScale employs a *plug-in* model for its implementation that enables multiple extant technologies to be employed within a particular AppScale instance. For example, the datastore layer is currently implemented via plug-ins for Cassandra, HBase, Hypertable, MongoDB, MemcacheDB, SimpleDB, MySQL Cluster, and Voldemort. AppScale exports a wide range of APIs to the applications that execute over it. These APIs include those from Google App Engine as well as those for large-scale data analytics and high-performance computing.

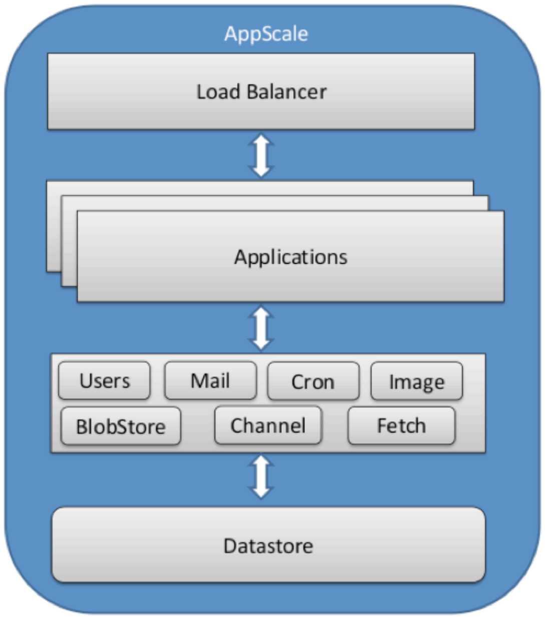

Figure 1: The multi-tiered AppScale design consists of a load balancer, multiple application servers, several services to support the different APIs, and a datastore layer for persistent storage of cloud-wide data and state (not all APIs and services are shown here).

Google App Engine APIs

The Google App Engine APIs that AppScale implements include Datastore, Blobstore, Channel, Images, Memcache, Namespaces, Task Queue, Users, URL Fetch, and XMPP. Google provides a description of the functionality of each of these APIs at http://code.google.com/appengine/docs/. We emulate the functionality of the Google App Engine APIs using open source software systems, tools, and services, as well as new modules and code for scalable interoperation of services across the system. We assume distributed execution and so integrate technologies into the platform that facilitate isolation, scalability, elasticity, and fault-tolerance for cloud platform applications. This is in sharp contrast to the various Google App Engine SDKs that are provided for testing and debugging using a single machine. We next overview the primary Google App Engine APIs that AppScale implements. We identify features and APIs not currently supported by AppScale in a limitations section at the end of this chapter.

Datastore API. The Datastore API grants users persistent data storage and retrieval. The API calls that the Datastore API exposes map to PUT, GET, DELETE, and QUERY calls to the underlying datastore.

Applications employ the Google Query Language (GQL) to access the datastore. GQL is similar to and is a subset of SQL's select statement; fundamentally, GQL lacks relational operations such as JOIN and MERGE. AppScale employs in-memory filters for GQL statements while Google App Engine maps them to range queries natively supported by the datastore.

AppScale implements transactional semantics (multi-row/key atomic updates) using the same semantics as Google App Engine. Transactions can only be performed by applications within an entity group. Entity groups are programmatic structures that describe relationships between datastore elements (keys). Applications request transactional semantics via a *run_in_transaction* operation. All AppScale datastore operations within a transaction are ACID-compliant with READ-COMMITTED isolation.

AppScale's implementation of transactions is database agnostic and relies on the open source distributed locking system called ZooKeeper to lock entity groups (Chohan, Bunch, Krintz, & Nomura, 2011). ZooKeeper stores information about which entities are currently locked. We employ garbage collection (GC) for locks that time out. We maintain entity version information in a journal within the database and implement rollback for failed transactions. Specifically, each lock in the system has a 30 second lease. A service scans the transaction list to identify expired transaction locks. The service adds any expired transaction to a *blacklist* and releases the lock. For correct operation with timeouts, virtual machines must be time-synchronized via NTP. Locks that are not successfully removed by an asynchronous call to ZooKeeper are garbage collected during the next iteration of the GC service. The GC service also cleans up entities and all related metadata that have been deleted within committed transactions. In addition, the GC collects entities identified in the journal that are older than the current valid version of an entity.

Blobstore API. The Blobstore API enables users to store large entities of text or binary data. The upload is performed by an application using a form post with a file name. The size of files that can be uploaded is not constrained by the 1MB limitation imposed by the Datastore API.

Channel API. The Channel API allows applications to push messages from the application server to a client's browser. A program registers an application token with the service, which is used to connect to a messaging service. The Channel API is useful for creating applications such as online chat or a real time multi-player game. The implementation in AppScale has the ability to send messages to multiple receivers who are registered using the same application token. Google App Engine imposes the restriction that there may be only one receiver per sending channel. AppScale uses Ejabberd and Strophe.js to provide a scalable Channel implementation.

Images API. The Images API facilitates programmatic manipulation of images. Popular functions in this API include the ability to generate thumbnails, perform rotations, composition, conversion between formats, and cropping images.

Memcache API. The Memcache API permits applications to store their frequently used data in a distributed memory grid. This can serve as a cache to prevent relatively expensive re-computations or database accesses. Since this service is a cache, it is possible for entries to be evacuated to create space for new updates, i.e., there is no guarantee that a recently accessed entry is in the cache.

Namespace API. The Namespace API enables developers to segregate their data into different namespaces. One common use of this API is to test new versions of applications in production without tampering with live production data. The Namespace API can also be used with the Memcache and Task Queue APIs.

Task Queue API. The Task Queue API facilitates asynchronous computation (tasks) by applications. Such background computation is important for applications that perform operations other than those in response to a web request. In AppScale, cloud administrators can set both the (inline) computation duration limit and asynchronous task duration limit if desired. Moreover, tasks can be chained so that one task can pick up where a previous one left off.

Users API. The Users API provides authentication for web applications through the use of HTTP cookies. Google App Engine's implementation leverages the Google Accounts infrastructure, so users with a Google Account can use it to access App Engine apps. Since AppScale does not have access to this infrastructure, it requires that users create an account through an AppScale portal URL. Alternatively, AppScale can be extended to employ other authentication services, e.g. those provided by the Eucalyptus open source cloud infrastructure or via LDAP. The AppScale implementation of this API distinguishes between regular users, application administrators, and cloud administrators (the latter categories possessing greater privileges).

URL Fetch API. Using the URL Fetch API, an application can perform POST and GET requests on remote resources. In addition, the application can access REST APIs from third parties using this API.

XMPP API. The XMPP API gives user the ability to receive and send messages using a valid XMPP account. Google App Engine leverages the Google Talk infrastructure; AppScale employs Ejabberd.

Other AppScale APIs

In addition to the Google App Engine APIs, AppScale implements other APIs that are not supported by Google App Engine. These APIs are important for emerging application domains such as data analytics and high-performance computing.

MapReduce Streaming API. AppScale supports arbitrary map-reduce type computation via Hadoop Streaming. Through Hadoop Streaming, users can specify a Mapper and Reducer program that can run under the MapReduce programming paradigm. The framework provides fault-tolerance and data replication. AppScale exposes a simple API that interfaces to this framework. The API is:

- putMRInput(data, inputLoc): Given a string "data" and a Hadoop file system location “inputLoc”, creates a file on the Hadoop file system named “inputLoc”.
- runMRJob(mapper, reducer, inputLoc, outputLoc, config): Given the relative paths to a mapper and reducer file (relative to the main Python file being run), run a Hadoop MapReduce Streaming job. Hadoop MapReduce is supported via the Hadoop Distributed File System (HDFS). We feed input to the program via the HDFS file named “inputLoc”, and collect output via the HDFS file named “outputLoc”. If a hash is passed as "config", the key/value pairs are passed as configuration options to Hadoop Streaming.
- getMROutput(outputLoc): Given a Hadoop file system location “outputLoc”, returns a string with the contents of the named file.
- writeTempFile(suffix, data): Writes a file to /tmp on the local machine with the contents data. Is useful for passing a file with nodes to exclude from MapReduce jobs.
- getAllIPs(): Returns an array of all the IPs in the AppScale cloud. Is useful for excluding or including nodes based on some user-defined application logic.

- getNumOfNodes(): Returns an integer with the number of nodes in the AppScale cloud. Is useful for determining at MapReduce run time, how many Map tasks and Reduce tasks should be run for optimal performance (recommended value is 1 Map task per node).
- getMRLogs(outputLoc): Returns a string with the MapReduce job log for the job whose output is located at outputLoc. Data is returned as a combination of XML and key/value pairs, in the standard Hadoop format.

Currently, developers can use this API to write Mappers and Reducers in the Python, Ruby, and Perl programming language. To use this API, the cloud administrator must select HBase or Hypertable as the platform's datastore so that HDFS is automatically deployed (both of these datastores use HDFS for their implementations). A complete sample application that uses the MapReduce Streaming API is bundled with the AppScale Tools and is called *mapreduce*.

EC2 API. AppScale also provides a set of tools to interact with cloud infrastructures (infrastructure-as-a-service), including the Amazon EC2 public cloud and the Eucalyptus private/hybrid system. Users can use the tools to spawn virtual machines (VMs) and manage them entirely through an AppScale web service. This API provides mirror that of the EC2 command line tools for spawning, describing, terminating, and managing VM instances. We also provide other utilities to handle the storage and retrieval of EC2 and Eucalyptus credentials. An example of these utilities is available in the *ec2demo* application that we bundle with the AppScale Tools.

INSTALLATION AND DEPLOYENT

Although AppScale can be installed directly on hardware, we recommend (for elasticity reasons) that administrators install AppScale within a virtual machine (VM) image. We currently release the system as a pre-installed Xen image. We test both Xen and KVM image formats; however, any image format should work (using a virtualized cluster (non-cloud) deployment). We release AppScale as a Xen image so that it can be deployed via the Eucalyptus private/hybrid cloud infrastructure over any cluster virtualization and Linux host operating system technology that Eucalyptus supports (Xen, KVM, VMWare and most Linux systems available today). We currently support Ubuntu v9.10 Karmic Koala for the Linux distribution and version *within* the image in which AppScale is installed if users are interested in installing AppScale from source.

We support two deployment strategies for AppScale using this VM: a *virtualized cluster deployment* and a *cloud infrastructure deployment*. In the former, the cloud administrator must copy and instantiate the AppScale image manually. That is, the administrator makes a copy of the AppScale VM for each node in the AppScale cloud (thus an N-node AppScale deployment requires N copies of the AppScale VM). The administrator then must instantiate and boot each of these VMs across the cluster, verify that each booted instance has a working networking between each of the VM instances, and record each of the IP addresses in use by each node (VM instances).

The AppScale nodes must be able to resolve the IP addresses of and communicate with all other nodes. These nodes can be instantiated on any number of physical machines, depending upon the resources available. We recommend that each be allocated with at least 1GB of memory and

2 virtual CPUs per AppScale instance. The administrator must also be able to *ssh* into each of these instances as root from some designated machine. The administrator should not shut down these instances (as is done in the cloud infrastructure case below), but instead keep them running and proceed to the AppScale deployment process in the next section. We describe how the administrator employs the recorded IP addresses for each of the AppScale nodes in the next section. When the administrator wants to shutdown the cloud completely, she does so first by shutting down AppScale (as described in the next section) and then manually shutting down the VM instances.

The second AppScale deployment configuration is over a cloud infrastructure. This type of deployment automates the VM instantiation/boot and shutdown. We currently support both the Amazon Web Services (AWS) Elastic Compute Cloud (EC2) public cloud and the AWS API-compatible Eucalyptus private/hybrid open source cloud system (http://www.eucalyptus.com). We make available via AWS a public Amazon Machine Image (AMI) with AppScale pre-installed. For Eucalyptus, a cluster administrator must deploy Eucalyptus and verify that it is working prior to using the released AppScale VM with Eucalyptus.

For either AWS or Eucalyptus, the cloud administrator must obtain account credentials to use the infrastructure. We recommend that administrators, prior to installing AppScale, verify that a non-AppScale image can be deployed and obtain a public and private IP address (in Eucalyptus these can be the same depending on the cloud networking configuration). In addition, we recommend that the administrator perform the same test for the AppScale image. This will not start the AppScale cloud (we discuss how to do this via the tools below). However, it will verify that the image works with the underlying infrastructure configuration and can successfully obtain an IP address (verify via the infrastructure API for describing instances). Once both of these tests are completed successfully, the cloud administrator should shut the instances down (via the API for terminating instances) and proceed to the next section on deploying AppScale.

Deploying AppScale

For either virtualized cluster or cloud infrastructure deployments, we automate the deployment of the AppScale system and its subcomponents via a set of tools called the *AppScale tools*. We release the tools in addition to the AppScale VM. These tools implement a cloud platform administration interface. The tools configure the AppScale platform components and constituent software and starts each in the correct order. In addition, the tools synchronize SSH keys and credentials across the AppScale VM instances.

The cloud administrator must execute these tools on a machine that has network access to (i.e., can resolve the IP addresses of) all of the AppScale nodes (VM instances). This machine is the one on which the AppScale tools are installed. Once installed, the administrator uses the tools to start the AppScale cloud software across the nodes and to upload and manage applications automatically.

The cloud administrator specifies the size of the AppScale platform (the number of machines to be used). The size can be one or more nodes. In addition, the administrator can optionally dictate the role of each node in an AppScale cloud. The administrator does so using a configuration file located on the machine on which the tools are installed. The configuration file (called *ips.yaml* in the examples below) is in the YAML format, a popular, easy-to-use, human-readable, standard data serialization format. For virtualized cluster (non-cloud) deployment, the administrator specifies the IP addresses of the AppScale VM instances that were started. The IP addresses specified must be resolvable from the machine from which the AppScale tools are run. An example of the AppScale configuration file (*ips.yaml*) for a 4-node virtualized cluster deployment is as follows:

```
---
:controller: 192.168.1.2
:servers:
- 192.168.1.3
- 192.168.1.4
- 192.168.1.5
```

Also, for a virtualized cluster deployment, the cloud adminstrator performs one additional step (one that is not necessary for a cloud infrastructure deployment). This step sets up an SSH keypair for use across the AppScale VMs instances that the administrator has started (identified in the *ips.yaml* file) via the *appscale-add-keypair* AppScale tool. For example:

```
user@mylaptop:~/appscale-tools/ips$ appscale-add-keypair --ips ips.yaml
root@192.168.1.3's password:
...
A new ssh key has been generated for you and placed at
/home/user/.appscale/appscale.
You can now use this key to log into any of the machines you specified
without providing a password via the following command:
        ssh root@XXX.XXX.XXX.XXX -i /home/user/.appscale/appscale
```

This example initiates this process and asks the administrator for the root password for each of the nodes that were started and identified in the *ips.yaml* file. AppScale sets up the ssh keys across the nodes in the cloud. For cloud infrastructure deployments, the cloud infrastructure implements this step automatically.

For a cloud infrastructure deployment, the administrator specifies a unique identifier for each of the nodes that can be used for an AppScale cloud deployment over Amazon EC2 or Eucalyptus. The *ips.yaml* file in this case identifies nodes using the *node-* keyword and a unique integer, as opposed to IP addresses. AppScale interacts with the underlying cloud infrastructure to automatically start the instances and extract their IP addresses. An example of the *ips.yaml* file for a 4-node cloud infrastructure deployment is a follows:

```
---
:controller: node-1
:servers:
- node-2
- node-3
- node-4
```

The configuration file (*ips.yaml*) starts with a line with three dashes (---). In both of the examples above, there is a controller that is the head node of the system. By default, AppScale deploys a Database Master service and a load balancer on the controller. The other nodes specified in the configuration file are called servers. AppScale by default deploys Database Slave services and Application servers on the server nodes. We detail how the cloud administrator can control the services deployed per node in a future subsection of this chapter. The AppScale Application Servers host the cloud applications. AppScale can be deployed using one or more nodes. An AppScale cloud with at least three nodes employs the fault tolerance features of the constituent services. A single node configuration file contains only the controller following the first (---) line.

From this point forward, the steps are the same for virtualized cluster and cloud infrastructure deployments. The next step initiates the automatic configuration and instantiation of AppScale components that make up the platform. This step is achieved using the AppScale tool *appscale-run-instances*. The cloud administrator specifies the database to use within the platform and (optionally) the first application that the administrator wishes to deploy over the cloud platform. The administrator can use the tool *appscale-upload-app* to upload one or more applications once the AppScale cloud is started. The output from the *appscale-run-instances* tool is similar to that which follows. In this example, we deploy AppScale over a virtualized cluster; the ips.yaml file specifies the roles and IP addresses as described above. To deploy over a cloud infrastructure, the user adds the flag `--infrastructure euca` for Eucalyptus or `--infrastructure ec2` for Amazon EC2, and replaces the IP addresses with node identifiers in the *ips.yaml* file as described above.

```
user@mylaptop:~/appscale-tools/ips$ appscale-run-instances --ips ips.yaml
--file ../sample_apps/guestbook/ --table cassandra

About to start AppScale over a non-cloud environment.
Head  node  successfully  created  at  192.168.1.2.  It  is  now  starting  up
cassandra via the command line arguments given.
Generating certificate and private key
Starting server at 192.168.1.2
Please wait for the controller to finish pre-processing tasks.

This AppScale instance is linked to an e-mail address giving it administrator
privileges.
Enter your desired administrator e-mail address: user@example.com

The new administrator password must be at least six characters long and
can include non-alphanumeric characters.
Enter your new password:
Enter again to verify:
Please wait for AppScale to prepare your machines for use.
Copying over needed files and starting the AppController on the other VMs
Starting up Cassandra on the head node

Your user account has been created successfully.
Uploading guestbook...
We have reserved the name guestbook for your application.
guestbook was uploaded successfully.
```

```
Please wait for your app to start up.

Your app can be reached at the following URL:
http://192.168.1.2/apps/guestbook
The status of your AppScale instance is at the following URL:
http://192.168.1.2/status
```

AppScale first initializes the controller. It then asynchronously starts up the other nodes in the system. In a cloud infrastructure deployment, this step includes the instantiation and booting of the AppScale VM. This process can take a significantly long time (tens of minutes) for large images and images not yet cached in the cloud infrastructure. Concurrently, the administrator enters an e-mail address and password for cloud administrator access. The cloud administrator (user) is given special privileges so that she can control all applications that execute in this AppScale deployment.

The system next starts the system-wide datastore. In this example, we use Cassandra (we specify it as an *appscale-run-instances* option (--table)). The system creates the administrator's account, uploads the application (guestbook in this example, specified via the --file option that contains the path to the application directory). The tool then specifies a URL that implements the AppScale cloud status interface. AppScale implements a load balancer in front of this URL to direct traffic across the available server instances in the platform. The status page lists the services running on each node in the platform, the CPU and memory used by each, and the applications that are currently running in the cloud. The interface also provides a link to the AppScale monitoring system that displays a wide range of dynamic metrics about the system as time series.

The administrator can use the tool *appscale-describe-instances* to view the status of a cloud platform deployment. The output of this tool is similar to that which AppScale displays on its cloud status page and looks similar to the following:

```
user@mylaptop:~/appscale-tools/ips$ appscale-describe-instances
Status of node at 192.168.1.2:
    Currently using 1.0 Percent CPU and 96.94 Percent Memory
    Hard disk is 99 Percent full
    Is currently: load_balancer, shadow, db_master, zookeeper, login
    Database is at 192.168.1.2
    Current State: Preparing to run AppEngine apps if needed
Status of node at 192.168.1.3:
    Currently using 0.3 Percent CPU and 95.00 Percent Memory
    Hard disk is 98 Percent full
    Is currently: load_balancer, db_slave, appengine
    Database is at 192.168.1.3
    Current State: Preparing to run AppEngine apps if needed
    Hosting the following apps: guestbook
Status of node at 192.168.1.4:
    Currently using 0.3 Percent CPU and 96.45 Percent Memory
    Hard disk is 98 Percent full
    Is currently: load_balancer, db_slave, appengine
    Database is at 192.168.1.4
    Current State: Preparing to run AppEngine apps if needed
    Hosting the following apps: guestbook
Status of node at 192.168.1.5:
    Currently using 0.3 Percent CPU and 95.65 Percent Memory
    Hard disk is 98 Percent full
    Is currently: load_balancer, db_slave, appengine
    Database is at 192.168.1.5
    Current State: Preparing to run AppEngine apps if needed
    Hosting the following apps: guestbook
```

The *appscale-upload-app* and *appscale-remove-app* tools can be used by cloud platform users to add and remove applications to/from the AppScale platform, respectively. In this example, we add an application called "shell":

```
user@mylaptop:~/appscale-tools/ips$ appscale-upload-app --file
../sample_apps/shell/

This AppScale instance is linked to an e-mail address giving it administrator
privileges.
Enter your desired administrator e-mail address: user@example.com
Preparing to run AppEngine apps if needed

Uploading shell...
We have reserved the name shell for your application.
shell was uploaded successfully.
Please wait for your app to start up.

Your app can be reached at the following URL:
http://192.168.1.2/apps/shell
```

As part of this process, we specify the path to the directory containing our application as well as the e-mail address of the application owner (note this need not be the cloud administrator). We are not prompted for a password in this example because the user we specify already exists. In this next example, we remove the "shell" application:

```
user@mylaptop:~/appscale-tools/ips$ appscale-remove-app --appname shell

We are about to attempt to remove your application, shell.
Are you sure you want to remove this application (Y/N)? Y
Your application was successfully removed.
```

We provide *appscale-remove-app* with the name of the application to remove. After confirming our selection, the tool contacts the master node in the system and stops the "shell" application. Note that if the application actually isn't running in the system (e.g., because we misspelled the name of the application to remove or we already stopped it earlier), then the output informs us that this is the case:

```
user@mylaptop:~/appscale-tools/ips$ appscale-remove-app --appname shell2
We are about to attempt to remove your application, shell2.
Are you sure you want to remove this application (Y/N)? Y
We could not stop your application because it was not running.
```

Cloud administrators can terminate an AppScale deployment (entire cloud) using the tool *appscale-terminate-instances*. For virtualized cluster deployments, this step shuts down all AppScale services across the running VMs without shutting down the VM instances. In cloud deployments, this step performs both activities. The shutdown of an AppScale cloud that we have deployed over virtualized cluster looks similar to the following:

```
user@mylaptop:~/appscale-tools/ips$ appscale-terminate-instances
Not backing up data.

About to terminate instances spawned via Xen/KVM with keyname 'appscale'...
Shutting down AppScale components at 192.168.1.2................
Shutting down AppScale components at 192.168.1.3.....
Shutting down AppScale components at 192.168.1.4
Shutting down AppScale components at 192.168.1.5
Terminated AppScale across 4 boxes.
```

Terminating in cloud deployments is similar. However, we specify the cloud infrastructure type over which we are deploying.

```
user@mylaptop:~/appscale-tools/ips$ appscale-terminate-instances /
--infrastructure euca
Not backing up data.

About to terminate instances spawned via euca with keyname 'appscale'...
Terminated AppScale in cloud deployment.
```

Advanced Deployment Strategies

AppScale is also capable of performing more advanced and controlled placement of components and services. This is useful because different application use cases may benefit from different placement strategies: for example, applications that heavily use the database may wish to statically request extra database nodes or dedicate nodes to be used only as database nodes (as opposed to co-locating them with application servers). Alternatively, applications that need a low latency to the database may benefit from co-locating the application server with the database itself. In either scenario, these placement strategies give application hosts the ability to easily set up various placement styles for their application and use a load tester to quantify which placement style is optimal for the amount of users they are expecting and how their application in particular responds to it.

Specifying placement strategies is controlled through the YAML configuration file that we described in the previous subsection. In that previous example, the `ips.yaml` file identifies one node as the *controller*. This role consists of a load balancer and Database Master service. The roles of the other nodes, called *servers*, consist of Database Slaves and application servers. This is the default AppScale placement strategy and we illustrate it in Figure 2.

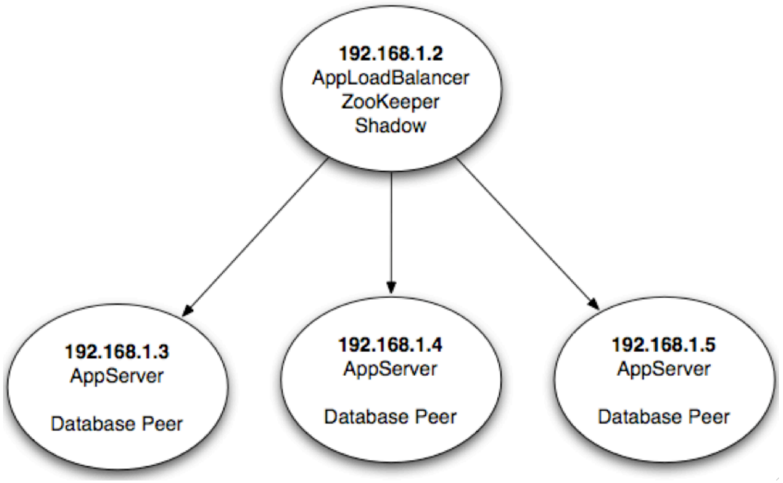

Figure 2: The default placement strategy within AppScale. Here, one node (the controller) handles load balancing and a single database server, while the other nodes (the servers) deploy application servers as well as database servers.

Users can define their own placement strategies instead of using the default. The possible roles are:

- Load balancer: The service that routes users to their Google App Engine applications. It also hosts a status page that reports on the state of all machines in the currently running AppScale deployment.
- App Engine: Our modified version of the non-scalable Google App Engine SDKs that adds in the ability to store data to and retrieve data from databases that support the Google Datastore API, as well as our implementations of the various other APIs that App Engine offers.
- Database: Runs all the services needed to host the chosen database.
- Login: The primary machine that is used to route users to their Google App Engine applications. Note that this differs from the load balancer - many machines can run load balancers and route users to their applications, but only one machine is reported to the cloud administrator when running *appscale-run-instances* and a*ppscale-upload-app*.
- ZooKeeper: Hosts metadata needed to implement database-agnostic transaction support.
- Shadow: Queries the other nodes in the system to record their state and ensure that they are still running all the services they are supposed to be running.
- Open: The machine runs nothing by default, but the Shadow machine can dynamically take over this node for use it as needed (for elasticity and/or fault recovery purposes).

We show how to implement a placement strategy via example. In this example, we wish to have a single machine for load balancing, for execution of the database metadata server via ZooKeeper, and for routing users to their applications. These functions are commonly used together, so the role *master* signifies all of them in a single line of configuration. On the other nodes we host applications (two nodes) and our database (one node). The configuration file for this deployment strategy using a virtualized private cluster (because we specify IP addresses) looks like the following:

```
:master: 192.168.1.2
:appengine:
- 192.168.1.3
- 192.168.1.4
:database:
- 192.168.1.5
```

In another example deployment, we use a cloud infrastructure with one node hosting the *master* role, two nodes host web applications and database instances, and a final node for routing users to their applications. The configuration file for this looks like the following:

```
---
:master: node-1
:appengine:
- node-2
- node-3
:database:
- node-2
- node-3
:login:
- node-4
```

Figure 3 illustrates this placement strategy, where the machines have been assigned IP addresses beginning with 192.168.1.2.

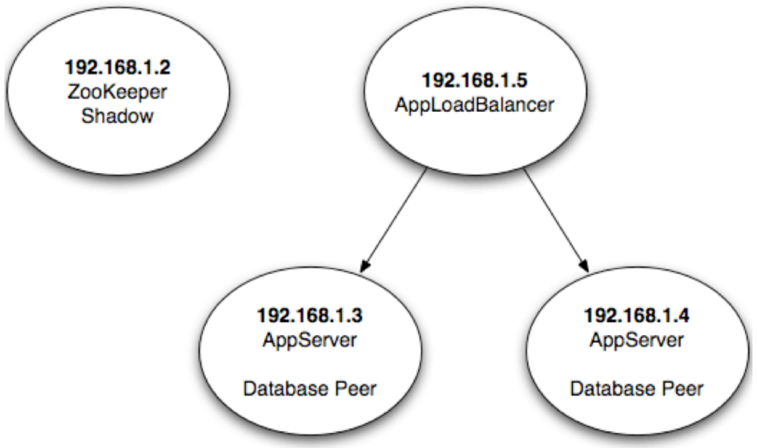

Figure 3: A more complex placement strategy, in which a single node handles load balancing, two nodes handle application servers and database nodes, and a final node handles transaction management and system monitoring.

For databases that employ a master/slave relationship, the first machine specified in the *database* section is the master, while the others are slaves. For peer-to-peer databases, all nodes are peers, but the machine specified first in the *database* section is the first peer to be started.

To keep this example simple, we employ a single machine to run ZooKeeper. However, doing so manifests as a single point of failure (SPOF). A cloud administrator can avoid this SPOF by specifying that multiple nodes implement the ZooKeeper role. As ZooKeeper uses the Paxos algorithm to establish consensus, there should be an odd number of ZooKeeper nodes.

Common Errors

While AppScale aims to simplify cloud service and application deployment, there are many layers of abstraction on which it relies to do so. If any one of these layers fail, AppScale can fail as well. To make troubleshooting easier, the *appscale-run-instances* tool can be with the ``-v'' flag to produce verbose output, which oftentimes is verbose enough to capture information about errors in the system. When this flag is used, a log file is placed on the *master* node (or *controller* node in default deployments) in the /tmp directory, named <ip of master>.log. Similarly, the master node places log files on each other machine at the same location in case those machines encounter errors.

Also, not all error messages within the verbose log are fatal: SSL-related exception messages within this log can be safely ignored, as well as load-balancer (*nginx)* related messages. Other exceptions may identify actual problems occurring with an AppScale deployment. One common way that AppScale can fail is if, in the virtualized cluster deployment, AppScale is installed on one virtual machine but not the others. As the components needed are not on all machines, AppScale does not have the necessary components to run and will fail. Users can easily verify if this is the problem they are running into by checking the logs AppScale produces; it will contain an error message saying this is the problem and how to remedy it.

A similar type of problem can occur if users install support for a subset of supported databases in AppScale and then try to run a non-installed database. For example, if users only install Cassandra support for AppScale and then try to run HBase, AppScale will fail to start, as it does not have the necessary files to start HBase. Once again, the verbose logs will contain an error message that indicates that this is the problem and how to remedy it.

Problems can also arise when custom networking setups are employed. For example, virtual machines requiring a special */etc/hosts* file to access other machines or the machine that the AppScale tools are located on may find these rules coming into conflict with the rules employed by services running within AppScale. Diagnosing these problems tends to be more difficult; however, the verbose logs should contain error messages indicating that network-related problems are occurring, and if these are seen, reverting to the standard */etc/hosts* file is often beneficial.

Another difficult class of problems arises when other types of software upon which AppScale relies are not installed or are configured incorrectly. For example, if deploying AppScale over Eucalyptus, it is critical to ensure that Eucalyptus can deploy other images without error before trying to deploy the AppScale image to it. Here, it is vital to ensure that images can acquire both public and private IP addresses correctly, and that these IP addresses resolve in a sensible manner (e.g., that the public IP addresses actually resolve to the machine running the AppScale Tools). We have encountered many problems where users have had DHCP misconfigured for Eucalyptus or have not allocated enough disk space for Eucalyptus images: it is thus crucial to ensure that Eucalyptus is working correctly before attempting to run AppScale over it.

Community Support

To this point, we have overviewed AppScale. However, there is much more beneath the surface. We maintain a Google Code site at http://code.google.com/p/appscale that contains a comprehensive and up-to-date listing of each of the commands that users can employ to deploy and manage their AppScale deployments as well as the state of supported APIs. This site also contains information about published papers that use AppScale for research as well as add-ons to the AppScale platform.

On this Google Code site, users can also find a link to the most recent pre-built AppScale Xen image. Due to the large image size, we are not able to host the image on Google Code itself, but the link does provide a BitTorrent file for quick download of the image and other necessary files.

Our team supports an active mailing list hosted by Google Groups for the AppScale Community, which can be found at http://groups.google.com/group/appscale_community. Users that have problems deploying AppScale should search this mailing list for answers to their problems, and ask their own if others have not reported the problem thus far. Users that ask questions should include a detailed description of their environment and AppScale configuration, as well as the verbose log in /tmp from their master node.

Links to these sites and others are available from http://appscale.cs.ucsb.edu. This site also hosts our *apt* repository where the most recent version of the AppScale installation packages is available.

APPSCALE-IN-DEPTH

In this section, we discuss the critical services that make up an AppScale deployment. We describe how they serve the applications that execute over the platform and that employ the AppScale APIs.

Database Services

The AppScale tools start the datastore plug-in (database instance) using one or more nodes. The AppController component starts and stops the database instance. In front of each instance, we provide a software layer that implements Google App Engine Datastore APIs (PUT, GET, DELETE, and QUERY functions) using the database.

There are three services that use and support the Datastore API: the Protocol Buffer Server (PBServer), the User/App Server (UAServer), and the Blobstore. The PBServer receives protocol buffers (a fast and compact data serialization method developed and released as open source by Google) from applications that use this API and interfaces with Zookeeper to provide database-agnostic transaction semantics for applications.

The UAServer uses the database to store user and application metadata in tables; it SOAP functions for simple RPC calls. All persistent data for the Users API is stored via the UAServer. Application metadata includes the IP addresses at which it is available, application administrators, XMPP accounts, and session information from the Channel API.

Finally, the AppScale blobstore service stores blobs that are uploaded by users in the database in 1MB chunks. A Blobstore Server runs on all nodes that host applications. POSTs of blob files are forwarded to this service and once the file has completed uploading, the service redirects the user back to the application's Blobstore upload success handler.

Monitoring Services

To report low-level system information about the behavior and performance of an AppScale cloud (and its applications), AppScale collects and exports performance data via a component called *Monitr*. Monitr uses the *collectd* utility to collect system-wide information such as CPU, disk utilization, memory usage, and network access from all nodes in the deployment. Users and administrators access Monitr via a link on the AppScale cloud status page.

AppScale utilizes the God process monitor to ensure that all services offered in an AppScale cloud are continuously running. If God detects that any service is using more than a system-specified amount of CPU or memory for an extended amount of time (or if it is not running at all), God will terminate the process and restart it. One instance of God runs on each node running in an AppScale deployment and monitors only the services that are assigned to that node. This information is stored and updated dynamically by the AppController service that executes on the same node (and on all nodes in the deployment).

Neptune

Neptune is a domain specific language for AppScale cloud configuration. With Neptune, users can dictate workflows for certain languages, frameworks, and scientific applications. These include but are not limited to X10, R, MPI, and Hadoop MapReduce. Users of Neptune have the capability to control execution of jobs between different clouds (hybrid cloud execution). As Neptune consists of a set of extensions to the Ruby programming language, users can write web applications in the popular Rails and Sinatra web programming frameworks that can initiate high performance computing jobs across AppScale clouds.

LIMITATIONS AND FUTURE WORK

There are several current limitations in AppScale that developers should be aware of before deploying AppScale and their applications. The list of limitations is as follows; for each, we provide possible workarounds and/or our plans for addressing them.

- Blobstore Max File Size: 100MB. This value is configurable within the code and we will allow for the setting of this value in a configuration file.
- Datastore: AppScale does not index data but rather does filtering of queries in-memory. As the size of the database gets larger, users may see a decrease in performance for queries. We are working on an indexing system which can create indexes yet stay compliant with the ACID semantics needed for transactions. Data persistence across AppScale deployments is also not automatically supported. Instead, we provide an extension to the bulkloader that is part of Google App Engine that user can use to download and upload data from their application or an AppScale deployment prior to shutting down and starting either. The syntax and semantics of the bulkloader are equivalent to those of Google's implementation.
- Task Queue: Tasks that are enqueued are not fault-tolerant, nor does AppScale handle delayed workers. We are currently in the process of refactoring our Task Queue support using an open source distributed queuing technology to fix this issue.
- Mail: Only the administrator is allowed to send mail, and reception of mail is not implemented. Extensions to this API are on our road map.
- OAuth API is not implemented in AppScale. We have this Google App Engine API on our road map.
- AppServer components currently are not elastic. We are currently investigating the necessary technology for AppServer elasticity and plan to enable such functionality in the future.
- Ubuntu Karmic Koala is currently (as of this writing) the only distribution and version that we support for the implementation of AppScale VM images. This is to limit testing and allow for faster release cycles. AppScale is portable to other distributions and in the past has successfully run within a VM that implements Redhat Linux
- Some datastores have no method for retrieving the entire table to run a query. They must use a set of special keys that track all keys in the table. These datastores therefore have the added overhead of always accessing the special keys. These datastores are: MemcacheDB, Voldemort, and SimpleDB.

Future Directions

We are continuously working to improve the design and implementation of AppScale to address the limitations above and to increase transparency, performance and scale. In addition, we have focused on the automation and control of, as well as more coordinated interaction (for scheduling, automatic service-level agreement negotiation, elasticity, etc.) with cloud infrastructures (e.g. Amazon EC2 and Eucalyptus).

As part of future work, we are extending AppScale with new services for large-scale data analytics as well as data and computationally intensive tasks. In addition, we are investigating a wide range of hybrid cloud technologies to facilitate application development and deployment that is cloud-agnostic. This entails new services, APIs, scheduling, placement, and optimization techniques. Finally, we are investigating new language support, performance profiling, and debugging support for cloud applications as well as the integration of mobile device use within the platform.

In summary, we have presented AppScale, its design, implementation, and use. We have detailed the APIs that AppScale supports and how they relate to Google App Engine. Moreover, we have described extensions that are specific to AppScale that expand the applicability of the platform, while retaining the simplicity and automation that clouds offer. We encourage readers to try AppScale; details to do so can be found here: http://code.google.com/p/appscale/. We appreciate all feedback and suggestions that users can provide through our community mailing list http://groups.google.com/group/appscale_community.

REFERENCES

Bittorrent: http://www.bittorrent.com

Cassandra Datastore: http://incubator.apache.org/cassandra

Amazon Elastic Compute Cloud: http://aws.amazon.com/ec2

EJabberD: http://www.ejabberd.im

Eucalyptus: http://www.eucalyptus.com/

Google App Engine Transaction Support:
http://code.google.com/appengine/docs/python/datastore/transactions.html

Google App Engine: http://code.google.com/appengine/

HBase Datastore: http://hadoop.apache.org/hbase

MySQL: http://www.mysql.com

SimpleDB: http://aws.amazon.com/simpledb

Strophe.js: http://strophe.im

virt-install: http://manpages.ubuntu.com/manpages/lucid/man1/virt-install.1.html

VMBuilder: http://manpages.ubuntu.com/manpages/lucid/en/man1/vmbuilder.1.html

Voldemort Datastore: http://project-voldemort.com

YAML: http://www.yaml.org/

Zookeeper: http://hadoop.apache.org

Hypertable Datastore: http://hypertable.org

MemcacheDB Datastore: http://memcachedb.org

MongoDB Datastore: http://mongodb.org

AppScale Tools Wiki: http://code.google.com/p/appscale/wiki/AppScale_Tools_Usage

Bunch, C., Chohan, N., Krintz, C., & Shams, K. (2011). Neptune: A Domain Specific Language for Deploying HPC Software on Cloud Platforms. *ScienceCloud.*

Bunch, C., Chohan, N., Krintz, C., Chohan, J., Kupferman, J., Lakhina, P., et al. (2010). An Evaluation of Distributed Datastores Using the AppScale Cloud Platform. *IEEE International Conference on Cloud Computing.*

Bunch, C., Kupferman, J., & Krintz, C. (2010). Active Cloud DB: A RESTful Software-as-a-Service for Language Agnostic Access to Distributed Datastores. *ICST International Conference on Cloud Computing.*

Chohan, N., Bunch, C., Krintz, C., & Nomura, Y. (2011). Database-Agnostic Transaction Support for Cloud Infrastructures. *IEEE International Conference on Cloud Computing.*

Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., et al. (2009). AppScale: Scalable and Open App Engine Application Development and Deployment. *ICST International Conference on Cloud Computing.*

Key Terms:

AppScale, Google App Engine, cloud computing, platform-as-a-service, elasticity, cloud application development, cloud runtime system, distributed systems, utility computing.

Brief Author Biographies

Chandra Krintz is a Professor of Computer Science at the University of California, Santa Barbara (UCSB). She joined the UCSB faculty in 2001 after receiving her M.S. and Ph.D. degrees in Computer Science from the University of California, San Diego (UCSD). Chandra's research interests include automatic and adaptive compiler, programming language, virtual runtime, and operating system techniques that improve performance and reduce energy consumption. Her recent work focuses on programming language and runtime support for cloud computing. Chandra has supervised and mentored over 40 students, has published her work in a wide range of ACM and IEEE venues, and leads several educational and outreach programs that introduce computer science to young people. Chandra's efforts have been recognized with an NSF CAREER award, the CRA-W Anita Borg Early Career Award, and the UCSB Academic Senate Distinguished Teaching Award.

Chris Bunch is a Ph.D. student in Computer Science at the University of California, Santa Barbara. Chris received his B.S. in Computer Science from California State University, Northridge in 2007. Chris' research interests include language support for emerging virtualization-based systems with an emphasis on accessibility and ease-of-use for programmers with diverse skill levels and backgrounds. Chris' recent work targets how cloud computing platforms intersect with other domains such as scientific and high-performance computing.

Navraj Chohan is a Ph.D. student in Electrical and Computer Engineering at the University of California, Santa Barbara (UCSB). Navraj received his B.S. in 2005 and his M.S. in 2008 in Computer Engineering from UCSB. Navraj's research interests focus on distributed networks, including large-scale data analytics and cloud software systems. Navraj's past work targeted sensor networking and embedded programming.