AppScale: Scalable and Open AppEngine Application Development and Deployment

Navraj Chohan Chris Bunch Sydney Pang
Chandra Krintz Nagy Mostafa Sunil Soman Rich Wolski

Computer Science Department
University of California, Santa Barbara

Abstract. We present the design and implementation of AppScale, an open source extension to the Google AppEngine (GAE) Platform-as-a-Service (PaaS) cloud technology. Our extensions build upon the GAE SDK to facilitate distributed execution of GAE applications over virtualized cluster resources, including Infrastructure-as-a-Service (IaaS) cloud systems such as Amazon's AWS/EC2 and Eucalyptus. AppScale provides a framework with which researchers can investigate the interaction between PaaS and IaaS systems as well as the inner workings of, and new technologies for, PaaS cloud technologies using real GAE applications.

Key words: Cloud Computing, PaaS, Open-Source, Fault Tolerance, Utility Computing, Distributed Systems

1 Introduction

Cloud Computing is a term coined for a recent trend toward service-oriented cluster computing based on Service-Level Agreements (SLAs). Cloud computing simplifies the use of large-scale distributed systems through transparent and adaptive resource management. It provides simplification and automation for the configuration and deployment of an entire software stack. Moreover, cloud technology enables arbitrary users to employ potentially vast numbers of multi-core cluster resources that are not necessarily owned, managed, or controlled by the users themselves. Specific cloud offerings differ, but extant infrastructures share two common characteristics: they rely on operating system virtualization (e.g., Xen, VMWare, etc.) for functionality and/or performance isolation and they support per-user or per-application customization via a service interface typically implemented using high-level language technologies, APIs, and web services.

The three prevailing classes of cloud computing are Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS). SaaS describes systems in which high-level functionality (e.g., SalesForce.com [24], which provides customer relationship management software as an on-demand service) is hosted by the cloud and exported to thin clients via the network. The main feature of SaaS systems is that the API offered to the cloud client is for a complete software service and not programming abstractions or resources.

Commercial SaaS systems typically charge according to the number of users and application features.

PaaS refers to the availability of scalable abstractions through an interface from which restricted (e.g., HTTP(s)-only communication, limited resource consumption), network-accessible, applications written in high-level languages (e.g. Python, JavaScript, JVM and .Net languages) can be constructed. Two popular examples of PaaS systems are Google App Engine (GAE) [13] and Microsoft Azure [3]. Users typically test and debug their applications locally using a non-scalable development kit and then upload their programs to a proprietary, highly scalable PaaS cloud infrastructure (runtime services, database, distribution and scheduling system, etc.). Commercial offerings for both PaaS and IaaS systems charge a low pay-as-you-go price that is directly proportional to resource use (CPU, network bandwidth, and storage); these providers typically also offer trial or capped resource use options, free of charge.

IaaS describes a facility for provisioning virtualized operating system instances, storage abstractions, and network capacity under contract from a service provider. Clients fully configure and control their instances as root via ssh. The Amazon Web Services (AWS) which includes the Elastic Compute Cloud (EC2), Simple Storage System (S3), Elastic Block Store (EBS) and other APIs [1] is, at present, the most popular example of an IaaS-style computational cloud. Amazon charges per instance occupancy hour and for storage options at very competitive rates. Similar to those for PaaS systems, these rates are typically significantly less than the cost of owning and maintaining even a small subset of the resources that these commercial entities make available to users for application execution.

Eucalyptus [20] is an open-source IaaS system that implements the AWS interface. Eucalyptus is compatible with AWS to the extent that commercial tools designed to work with EC2 (e.g., Rightscale [22], Elastra [11], etc.) cannot differentiate between an Amazon and a Eucalyptus installation. Eucalyptus allows researchers to deploy, on their own cluster resources, an open-source web-service-based software infrastructure that presents a faithful reproduction of the AWS functionality in its default configuration. Furthermore, Eucalyptus provides a research framework for investigation of IaaS cloud technologies.

Such a framework is key to advancing the state of the art in scalable cloud computing software architectures and to enabling users to employ cloud technologies easily on their own local clusters. Yet, despite the popularity and widespread use of PaaS systems, there are no open-source implementations of PaaS systems or APIs. To address this need, we have designed and implemented an open-source *PaaS* cloud research framework, called *AppScale*. AppScale emulates the functionality of the popular GAE commercial cloud. Specifically, AppScale implements the Google App Engine open APIs and provides an infrastructure and toolset for distributed execution of GAE applications over virtualized clusters and IaaS systems (including EC2 and Eucalyptus). Moreover, by building on existing cloud and web-service technologies, AppScale is easy to use and able to execute real GAE applications using local and private cluster resources.

AppScale consists of multiple components that automate deployment, management, scaling, and fault tolerance of a GAE system. AppScale integrates, builds upon, and extends existing web service, high-level language, and cloud technologies to provide a system that researchers and developers can employ to investigate new cloud technologies or the behavior and performance of extant applications. Moreover, AppScale deployment requires no modifications to GAE applications. AppScale is not meant to compete with, outperform, or scale as well as, proprietary cloud systems, including GAE. Our intent is to provide a framework that enables researchers to investigate how such cloud systems operate, behave, and scale using real applications. Moreover, by facilitating application execution over important, lower-level cloud offerings such as Eucalyptus and EC2, AppScale also enables investigation of the interoperation and behavior of multiple cloud fabrics (PaaS and IaaS) in a single system. In the sections that follow, we describe the design, implementation, and a preliminary evaluation of AppScale.

2 Google App Engine

In April 2008, Google released a software framework for developing and hosting complete web service applications. This framework, called Google App Engine (GAE), enables users to write applications written in high-level programming languages and to deploy them on Google's proprietary and vast computing resources. The framework restricts the libraries that the application can use and limits the resources consumed by the program. This sandbox execution model limits application functionality in order to protect system stability, guarantee performance, and achieve scalability. The restrictions include communication limited to HTTP(S), program response to web requests within 30 seconds, no file system access except for files uploaded with the application, and persistent storage via simple in-memory or distributed key-value storage across requests.

Deployed GAE applications gain access to a high-quality, professionally maintained, and extremely scalable software infrastructure. This infrastructure is closed proprietary and includes the Google File System (GFS) [12], BigTable [8], MapReduce [9], Chubby [5] and Paxos [7]. GFS is a distributed, scalable, and reliable file system optimized for very large files and throughput-oriented applications. BigTable offers a distributed and highly available schema-free key-value store for fast access to structured data via a simple Datastore API. BigTable also integrates MapReduce for highly scalable concurrent execution of embarassingly parallel computations, such as data indexing and crunching for Google PageRank [4], Google Earth, and other applications. Chubby is a highly available naming service for GFS (that was originally designed as a locking service); the content of GFS are agreed upon using an optimized version of the original Paxos algorithm [15].

Google applications access these services through well-defined interfaces enabling the cloud to manage and controll resource usage very efficiently and scalably. GAE applications interoperate with other hosts via HTTP(S) using the

URL-Fetch API, manipulate images via the Images API, cache and store data via the Memcache and Datastore API, and access other Google applications via the Mail API and Accounts API. The web frontend of an application communicates via Remote Procedure Calls (RPC) with the datastore backend using protocol buffers [21] for fast and portable data serialization.

GAE developers write their web applications (webpage frontend, response computation, and data access) in Python using the GAE APIs, a subset of the Python libraries approved by Google, and the Django web framework [10] (or other similar and approved Python web framework). These frameworks significantly simplify and expedite common web development activities. Developers modify the data model in their programs to access the GAE Datastore API. In April 2009, Google made available a Java-based GAE framework. Developers employ the Java Servlet and Data Objects APIs and a subset of the Java libraries approved by Google to implement JVM-based GAE web applications.

Developers write a runtime configuration file for their application that identifies the program, specifies the versioning information, and identifies the handlers (code to execute or files to serve) for different URL accesses. Developers use a GAE software development kit (SDK) to test and execute their application locally and serially. The SDK implements the APIs using simple, slow, and non-scalable versions of the internal services. In particular, the SDK implements the Datastore API via a flat file (or very simple database). Once developers are ready to deploy their application on Google's resources, they do so by uploading a gzipped tar-ball of the code and configuration file to App Engine using an SDK tool. The developer also specifies and builds the indexes on the datastore for all queries that the application code can make, as part of the upload process.

The Google runtime system automatically load-balances the application according to user load. If the application exceeds its billable or fixed resource quota within a 24-hour period or 1-minute interval, the system returns a HTTP 403 Forbidden status until the resource is replenished. Application activities that are monitored by the Google system include CPU usage, network communication (bandwidth), requests (total and per minute), data storage, and emails sent.

In summary, Google App Engine provides access to vast and extreme scale resources for a very specific and well-defined web service application domain. Applications can be implemented and deployed into the clould quickly and easily using high-level languages, simple and well documented API's, and Google's SDK tools. Furthermore, the Google platform monitors and scales the applications. GAE thus enables a broad user base to develop web applications and deploy them without owning and managing sufficient cluster resources. The GAE APIs and the SDK carry open-source licenses but the internal, scalable, implementations are closed-source.

3 AppScale

To provide a platform for GAE application execution using local and private cluster resources, to investigate novel cloud services, and to faciliate research for

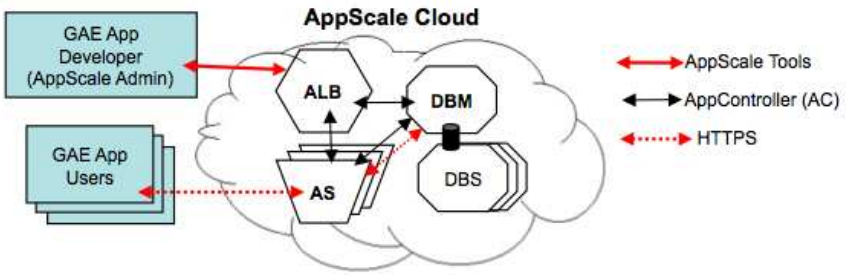

Fig. 1. Overview of the AppScale design. The AppScale cloud consists of an AppLoadBalancer (ALB), a Database Master (DBM), one or more Database Slaves (DBS), and one or more AppServers (ASs). Users of GAE applications interact with ASs; the developer deploys AppScale and her GAE applications through the head node (i.e. the node on which the ALB is located) using the AppScale Tools. AppControllers (ACs) on each node interact with the other nodes in the system; ASs interact with the DBM via HTTPS.

the next-generation of cloud software and applications, we have implemented AppScale. AppScale is a multi-language, multi-component framework for executing GAE applications. Figure 1 overviews the AppScale design.

AppScale consists of a toolset (the AppScale Tools), three primary components, the AppServer (ASs), the database management system, and the AppLoadBalancer (ALB), and an AppController (AC) for inter-component communication. AppServers are the execution engines for GAE applications which interact with a Database Master (DBM) via HTTPS for data storage and access. Database Slaves (DBSs) facilitate distributed, scalable, and fault tolerant data management. The AppController is responsible for setup, initialization, and tear down of AppScale instances, as well as cross component interaction. In addition, the AppController facilitates deployment of and authentication for GAE applications. The ALB serves as the head node of an AppScale deployment and initiates connections to GAE applications running in ASs. The AC of the head node also monitors and manages the resource use and availability of the deployment. All communications across the system are encrypted via the secure socket layer (SSL).

A GAE application developer interacts with an AppScale instance (cloud) remotely using the AppScale Tools. Developers use these tools to deploy AppScale, to submit GAE applications to deployed AppScale instances, and to interact with and administer AppScale instances and deployed GAE applications. We distinguish developers from *users*; users are the clients/users of individual GAE applications.

An AppScale deployment consists of one or more virtualized operating system instances (guestVMs). GuestVMs are Linux systems (*nodes*) that execute over the Xen virtual machine monitor, the Kernel Virtual Machine (KVM) [25] or IaaS systems such as Amazon's EC2 and Eucalyptus. For each AppScale deployment, there is a single AppLoadBalancer (ALB) which we consider the head node, one or more AppServers (AS), one Database Master (DBM) and one or

more Database Slaves (DBSs). A node can implement any individual component as well as any combination of these components; the AppScale configuration can be specified by the developer via command line options of an AppScale tool.

We next detail the implementation of each of these components. To facilitate this implementation we employ and extend a number of existing, successful, web service technologies and language frameworks.

3.1 AppController (AC)

The AppController (AC) is a SOAP client/server daemon written in Ruby. The AC executes on every node and starts automatically when the guestVM boots. The AC on the head node starts the ALB first and initiates deployment and boot of any other guestVM. This AC then contacts the ACs on the other guestVMs and spawns the components on each node. The head node AC first spawns the DBM (which then starts the DBSs) and then spawns the AppServers, configuring each with the IP of the DBM (to enable access to the database via HTTPS).

The AC on the head node also monitors the AppScale deployment for failed nodes and for opportunities to grow and shrink the AppScale deployment according to system demand and developer preferences. The AC periodically polls (currently every 10 seconds) the AC of every other node for a "heartbeat" and to collect per-application behavior and resource use (e.g. CPU and memory load). When a component fails, the AC restarts the component, respawning a node if necessary.

Although in this paper we evaluate the static default deployment of AppScale, we can also use this feedback mechanism to spawn and kill individual nodes of a deployment to respond to system load and performance. Killing nodes reduces resource consumption (and cost of resources are being paid for) and consists of stopping the components within a node and destroying the guestVM. We spawn nodes to add more AppServers or Database Slaves to the system. We are currently investigating various scheduling policies, feedback mechanisms, and capability to interact with the underlying cloud fabric to modify service level agreements. AppScale currently supports starting and stopping of any component in a node and automatic spawning and destroying nodes.

3.2 AppLoadBalancer (ALB)

The AppLoadBalancer is a Ruby on Rails [23] application that employs a simple HTTP server (nginx [19]) to select between three replicated Mongrel application servers [16] (for head-node load balancing). The ALB distributes initial requests from users to the AppServers (ASs) of GAE applications. Users initially contact the ALB to request a login to a GAE application. The ALB provides and/or authenticates this login and then selects an AS randomly. It then redirects the user request to the selected AS. The user, once redirected, continues to use the AppServer to which she was routed and does not interact further with the ALB unless she logs out or the AppServer she is using becomes unreachable.

3.3 AppServer (AS)

An AppServer is an extension to the development server distributed freely as part of the Google AppEngine SDK for GAE application execution. Our extensions to the development server enable fully automated execution of GAE applications on any virtualized cluster to which the developer has access, including EC2 and Eucalyptus. AppServers can also be used without virtualization which requires manual configuration. In addition, our extensions provide a generic datastore interface through which any database technology can be used. Currently we have implemented this interface to HBase and Hypertable, open-source implementations of Google's BigTable that execute over the distributed Hadoop File System (HDFS) [14]. We also have plugins for MySQL [17], Cassandra [6], and Voldemort [26].

We intercept the protocol buffer requests from the application and route them over HTTPS to/from the DBM front-end called the *PBServer*. The PBServer implements the interface to every datastore available and routes the requests to the appropriate datastore. The interaction is simple but fully supported by a number of different error conditions, and includes:

- Put: add a new item into the table (create table if non-existant)
- Get: retrieve an item by ID number
- Query: SQL-like query
- Delete: delete an item by ID number

Our other extensions facilitate automatic invocation of ASs and authentication of GAE users. The AC of the node sets the location of the datastore (passed in from a request from the head node AC), upon AS start. The AS also stores and verifies the cookie secret that we use to authenticate users and direct the component to authenticate using the local AppController (AC).

An AS executes a single GAE application at time. To host multiple GAE applications, AppScale uses additional ASs (one or more per GAE application) that it isolates within their own AppScale nodes or that it co-locates within other nodes containing other AppScale components.

3.4 Data Management

In front of the Database Master (DBM) sits the The PBServer is the front-end of the DBM. This Python program processes protocol buffers from a GAE application and makes requests on its behalf to read and write data to the datastore. As mentioned previously, AppScale currently supports HBase and Hypertable datastores. Both execute over HDFS within AppScale which performs replication, fault tolerance, and provides reliable service using distributed Database Slaves. The PBServer interfaces with HBase, Hypertable, Cassandra, and Voldemort using Thrift for cross-language interoperation.

The AC on the DBM node provides access to the datastore via these interfaces to the other ACs and the ALB of an AppScale system. The ALB stores uploaded GAE applications as well as user credentials in the database to authenticate the developer and users of GAE applications.

3.5 AppScale Tools

The developer employs the AppScale tools to setup an AppScale instance and to deploy GAE applications over AppScale. The toolset consists of a small number of Ruby scripts that we named in the spirit of Amazon's EC2 tools for AWS. The tools facilitate AppScale deployment on Xen-based clusters as well as EC2 and Eucalyptus. The latter two systems require credentials and service-level agreements (SLAs) for the use, allocation (killing and spawning of instances) of resources on behalf of a developer; the EC2 tools (for either IaaS system) generate, manage, distribute (to deployed instances), and authenticate the credentials throughout the cluster. The AppScale tools sit above these commands and make use of them for credential management in IaaS settings. In a Xen-only setting, no credential management is necessary; the tools employ ssh keys for cluster management. The tools enable developers to start an AppScale system, to deploy and tear down GAE applications, to query the state and performance of an AppScale deployment or application, and to manipulate the AppScale configuration and state. There is currently no limit on the number of uploaded applications.

3.6 Tolerating Failures

There are multiple ways in which AppScale is fault tolerant. The AppController executes on all nodes. If the AC fails on a node with an AS, that AS can no longer authenticate users for a particular GAE application but authenticated users proceed unimpeded. Users that contact an ALB to re-authenticate (acquire a cookie) are redirected to a node with a functioning AS/AC to continue accessing the application. If the AC fails on the node with the ALB, no new users can reach any GAE applications deployed in the AppScale instance and the developer is not able to upload additional GAE applications; extant users however, are unaffected. This scenario (AC on the ALB node failure) is similar to AC failure on the DBM node. In this scenario (AC on the DBM node failure), ASs and users are unaffected.

The database system continues to function as long as at least one DBS is available. Similarly, the system is tolerant to failure of the PBServer (DBM front-end). If the PBServer fails on the DBM, the ASs will temporarily be unable to reach the database until the AC on the node restarts the PBServer. The ASs are not able to continue to execute (GAE applications will fail) if the DBM goes down or becomes unreachable. In this scenario, the ALB will restart the DBM component but unless the data from the original DBM is available to restore, the restart is similar to restarting AppScale.

Although, coupling multiple components per node reduces the number of nodes (resource requirements) and potentially better utilizes underlying resources, it also increases the likelihood of failure. For example, if all components are located in a single node, node failure equals system failure. If the node containing the ALB and DBM fails, the system fails. In these scenarios, component failure does not equal node failure however; the AC in the head node will attempt to restart components as described previously. The DBM issues 3

Benchmark	Description	LOC Python or JavaScript	Transactions in Loop
cccwiki	user-defined webpage creation	289/10948	74
guestbook	presents last 10 signatures on a page; users can sign as well	81/0	9
shell	an interactive Python shell via a webpage	308/6100	14
tasks	to-do list manipulation	485/1248	44

Table 1. Benchmarks Statistics. For each benchmark, Column 2 is its description and Column 3 is its number of lines of code (Python/JavaScript). Column 4 is the number of transactions in the Grinder user loop that we use to load the system in our experiments.

replicas of tables for DBSs to store, thus user data is available on failure of any individual DBS component. We are investigating the various failure scenarios and techniques for tolerating them within a deployed AppScale system as part of ongoing and future work.

We distribute AppScale as a single Linux image and the AppScale Toolset. The image contains the code for the implementation of all of the components and a 64-bit Linux kernel and Ubuntu distribution. The system is available from `http://appscale.cs.ucsb.edu/`; all new programs that we have contributed carry the Berkeley Software Distribution (BSD) License.

4 Evaluation

We next present the basic performance characteristics of AppScale default deployment. We note that we have not optimized AppScale in any way and that this study presents a baseline from which we will work to improve the performance and scalability of the system over time. Our goal with AppScale to provide a research framework for the community, thus, we and others will likely identify ways to improve its performance over time. We simply provide a framework with which to investigate existing open source GAE applications, services, and execution characteristics using local cluster resources.

4.1 Methodology

For our experimental methodology, we investigate four open source GAE applications made available as Google AppEngine Samples (`http://code.google.com/p/google-app-engine-samples/`). The applications are Python programs and Python/JavaScript programs. We overview them and their basic characteristics in Table 1. The cccwiki and tasks applications require the user to log in. Each application uses the AppScale datastore for all data manipulation. We record a user session that we replay for an increasing number of users repeatedly using the Grinder load testing framework (`http://grinder.sourceforge.net`) and its extensions [18].

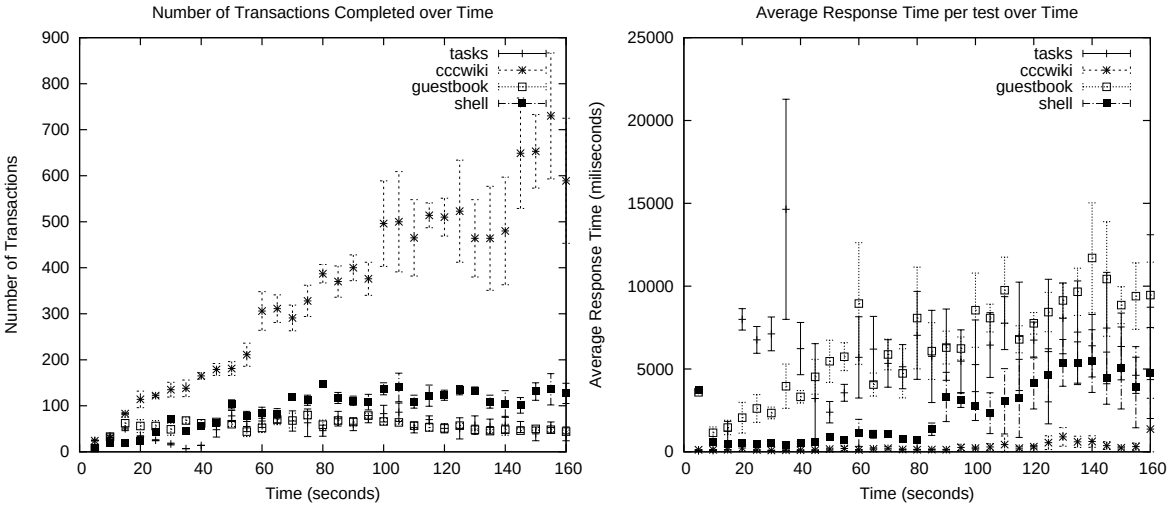

Fig. 2. Application performance under stress: Transactions over time (left) and average response time (right). The x-axis is time in seconds; Grinder introduces three additional users for load every 5 seconds. In the left graph each point is the number of transactions that completed in that interval, on average across five runs (y-axis). In the right graph, each point is the average response time across the transactions that began in that interval, on average across five runs (y-axis).

For each experiment, we investigate two metrics, **(i) the total number of transactions completed over a five second interval**, and **(ii) the average response time for transactions that start during the interval**. Specifically, each *Grinder user* repeatedly executes a series of transactions (Table 1 Column 3). The *user* repeats this loop for 160 seconds. Grinder adds three users every five seconds to load the system.

For each five second interval in the 160 seconds of each test, we count the number of transactions that complete in that interval (for transactions completed per interval). For average response time, for each five second interval of the 160 seconds, we compute the average response time for the transactions that *started* in that interval. We repeat each experiment five times and compute the average and standard deviation for each interval across all of the runs.

Our cluster consists of quad-core 2.66GHz machines with 8GB RAM connected via gigabit Ethernet. We employ three of these machines for Grinder load generators. The machines are synchronized and each Grinder instance introduces a single user every five seconds. We specify the number of machines we use for the AppScale deployment with each experiment below.

4.2 Experimental Results

We first present data for each application, executed in isolation over AppScale, over time and increasing load. For this experiment, we employ the default AppScale configuration: one head node (ALB+DBM) and three slave nodes

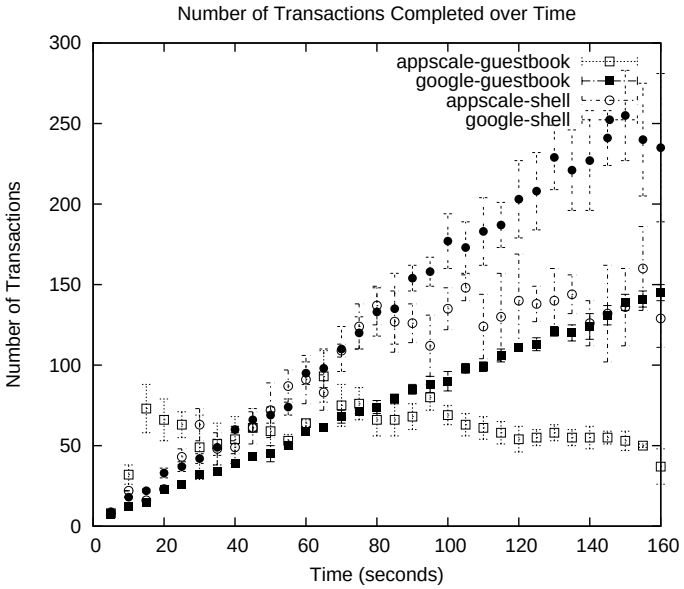

Fig. 3. Transactions over time under increasing load (3 users per 5 seconds) for two applications (guestbook and shell), when hosted by Google and AppScale.

(AS+DBS each) with each node/guestVM on its own machine. Each of the three Grinder machines accesses the AS of one slave node.

Figure 2 shows the results. The left graph is transactions over time (higher is better), the right graph is average response time (lower is better). Each graph plots a point every five seconds. The x-axis is time and load: Grinder adds three additional users every 5 seconds. In the left graph each point, is the number of transactions that completed in that interval, on average across five runs (y-axis). In the right graph, each point is the average response time across the transactions that began in that interval, on average across five runs (y-axis).

All of the applications except guestbook tend to grow in the number of transactions as load increases. Guestbook's transaction count decreases after 100 seconds. This is because each guestbook posting increases the size of the database table. Our current (naive) implementation of database queries is to return the entire table to the node so that we can apply any filters at the GAE client side. As the database grows, each call is more expensive. We are currently extending our query process to return only the individual entries required, to address this issue. Cccwiki scales much better because each transaction only modifies an existing page, altering an entry in the table, as opposed to creating a new entry as guestbook does.

We also evaluated the difference between executing the four guestVMs on a single (quadcore) machine versus on individual machines. We find that we achieve very similar results for both for transactions completed and response time. This is interesting since it shows that the overhead of virtualization and co-location of virtual machines on these systems is not the performance bottleneck

at this point. We find that in some cases the single machine case outperforms the distributed case due to network communication. This indicates that it may be beneficial to consider co-location of interoperating AppScale components for some behaviors and applications.

Finally, we investigate how AppScale performs relative to the Google proprietary infrastructure to better understand our baseline performance. We consider guestbook and shell applications since neither require the user to log in. We execute these applications using a Google AppEngine account. Figure 3 shows the results for transactions completed over time. AppScale transaction counts are more variable and do not scale for guestbook as load increases. Shell over AppScale scales up to a time/load of 80s. Google transaction counts scale perfectly. For response times (not shown) for guestbook Google consistently responds in 290-330ms regardless of load. For shell, Google's response time is more variable but still within a similar range. Shell performs more computation per request than guestbook. Google therefore starts to deny resources to the application at 150 seconds due to resource consumption limitations.

5 Related Work

The open-source offering most similar to AppScale is AppDrop [2]. AppDrop is a simple Ruby-on-Rails application that emulates and hosts AppEngine applications on Amazon's EC2. AppDrop is a proof-of-concept that GAE applications can be executed in an environment other than that of Google.

There are multiple differences between AppScale and AppDrop. First, AppDrop (and any GAE applications that execute using it) is hosted entirely using a single guestVM image, which places significant limitations on IaaS usage/accounting, performance, scalability, and fault tolerance. The AppDrop progenitor uses his own EC2 account to host GAE applications on behalf of GAE developers. Thus, AppDrop is responsible for all EC2 charges and resource use as well as any "bad behavior" by the GAE applications. Each AppScale instance and its GAE applications is deployed and "owned" by each individual GAE developer.

AppDrop implements the flat file database integrated in GAE SDK development server for its datastore. This system is not distributed, scalable, or fault tolerant. AppDrop also employs a secondary database (implemented using Rails ActiveRecord and PostGreSQL) to store and retrieve the user's session data. AppScale uses the same distributed and fault tolerant database infrastructure as it does for its GAE applications and facilitates any database to be "plugged into" AppScale. AppScale currently integrates HBase, Hypertable, MySQL, Cassandra, and Voldemort as distributed, fault tolerant datastore options.

6 Conclusions

We present AppScale, an open source PaaS cloud computing research framework that emulates the Google AppEngine-based cloud offering. AppScale is

easy to use and to extend and automatically deploys itself and GAE applications over Xen-based cluster resources and IaaS clouds such as Amazon EC2 and Eucalyptus. AppScale implements a number of different components that facilitate deployment of GAE applications using local (non-proprietary resources). Moreover, AppScale provides a framework with which cloud researchers and application developers can investigate new techniques (services, tools, schedulers, optimizations), and the performance and behavior of these techniques, and for real (GAE) applications.

References

1. Amazon Web Services. `http://aws.amazon.com/`.
2. AppDrop. `http://jchris.mfdz.com`.
3. Microsoft Azure Service Platform. `http://www.microsoft.com/azure/`.
4. S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Computer Networks and ISDN Systems*, pages 107–117, 1998.
5. M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI'06: Seventh Symposium on Operating System Design and Implementation*, 2006.
6. Cassandra. `http://incubator.apache.org/cassandra/`.
7. T. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live - An Engineering Perspective. In *PODC '07: 26th ACM Symposium on Principles of Distributed Computing*, 2007.
8. F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A Distributed Storage System for Structured Data. *Proceedings of 7th Symposium on Operating System Design and Implementation(OSDI)*, page 205218, 2006.
9. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of 6th Symposium on Operating System Design and Implementation(OSDI)*, pages 137–150, 2004.
10. Django. `http://www.djangoproject.com/`.
11. Elastra Inc.. `http://www.elastra.com`.
12. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, 2003.
13. Google AppEngine. `http://code.google.com/appengine/`.
14. Hadoop. `http://hadoop.apache.org/core/`.
15. L. Lamport. The Part-Time Parliament. In *ACM Transactions on Computer Systems*, 1998.
16. Mongrel. `http://mongrel.rubyforge.org`.
17. MySQL. `http://www.mysql.com`.
18. P. Nagpurkar, W. Horn, U. Gopalakrishnan, N. Dubey, J. Jann, and P. Pattnaik. Workload characterization of selected jee-based web 2.0 applications. *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on Workload Characterization (IISWC)*, pages 109–118, Sept. 2008.
19. Nginx. `http://www.nginx.net`.
20. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. Eucalyptus : A technical report on an elastic utility computing architecture linking your programs to useful systems. In *UCSB Technical Report ID: 2008-10*, 2008.

21. Protocol Buffers. Google's Data Interchange Format. http://code.google.com/p/protobuf.
22. Rightscale Inc. http://www.rightscale.com/.
23. Ruby on Rails. http://www.rubyonrails.org.
24. Salesforce Customer Relationships Management (CRM) System. http://www.salesforce.com/.
25. I. Sun Microsystems. White paper: Java(TM) 2 Platform Micro Edition(J2ME(TM)) Technology for Creating Mobile Devices, May 2000. http://java.sun.com/products/cldc/wp/ KVMwp.pdf.
26. Voldemort. http://project-voldemort.com/.