North by Northwest: Infrastructure Agnostic and Datastore Agnostic Live Migration of Private Cloud Platforms

Navraj Chohan Anand Gupta Chris Bunch Sujay Sundaram
Chandra Krintz
Computer Science Department
University of California, Santa Barbara, CA

Abstract

Cloud technology is evolving at a rapid pace with innovation occurring throughout the software stack. While updates to Software-as-a-Service (SaaS) products require a simple push of code to the production servers or platform, updates to the Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS) layers require more intricate procedures to prevent disruption to services at higher abstraction layers.

In this work we address the need for rolling upgrades to PaaS systems. We do so with the AppScale PaaS, which is a multi-application, multi-language, multi-infrastructure, and multi-datastore platform. Our design and implementation allows for applications and tenants to be migrated live from one cloud deployment to another with guaranteed transaction semantics and minimal performance degradation. In this paper we motivate the need for PaaS migration support and empirically evaluate migrations between two AppScale deployments using highly scalable datastores.

1 Introduction

Companies with on-premise Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) systems employ private cloud technology, which provides the flexibility and power of the public cloud, yet allows for the utilization of on-premise resources and infrastructure. At the IaaS level, these technologies include Eucalyptus, Nimbus, and OpenStack, all of which provide the capability to automatically initialize and isolate virtual machines (VMs) on physical machines [8, 5, 10].

PaaS systems, provide a set of high level APIs that developers program to, abstracting away lower level VM details such as memory, disk, and CPU. Once the application is deployed, the PaaS layer will scale resources as needed to provide high availability and meet given service level agreements (SLAs). In the private cloud space, current PaaS offerings include AppScale, OpenShift, and CloudFoundry[2, 9, 3].

As more and more companies go towards private PaaS offerings, there is a critical concern for providing high reliability and availability while also enabling the ability to perform updates on the underlying hardware and software resources. At the OS level, within individual VMs, security patches must be installed that may require the system to be rebooted. At the PaaS level, user applications rely on a multitude of software subsystems that may be frequently updated (e.g., load balancers, application servers, and databases). Moreover, hardware updates can occur when moving to higher end servers, or moving to higher performing storage options (such as solid state drives).

Open source PaaS technologies rely on a multitude of components, which themselves are comprised of open source solutions that are rapidly changing. These changes come in response to getting community uptake or decline in popularity, for reasons such as performance and reliability. The communities following NoSQL datastore technologies, where there are well over 100 different options [7], are a prime example where there are constant shifts between selections as technologies improve with better performance and newer feature sets. Yet, while the capability to swap out a datastore should be possible, developers of such technologies are not incentivized to create portable systems.

We address the requirement that real PaaS systems face for frequent upgrades and the desire to swap out technologies with minimal downtime by using a technique called *live migration*. With live migration, PaaS users can be transplanted from one underlying technology to another, whether that technology is the virtualization layer, the IaaS, or some component technology of the PaaS, with minimal service disruption.

We do so with the AppScale PaaS framework. AppScale is an open source cloud platform capable of running Google App Engine (GAE) applications and does so scalably while supporting multiple infrastructures and datastores. AppScale has plug-in capability for datas-

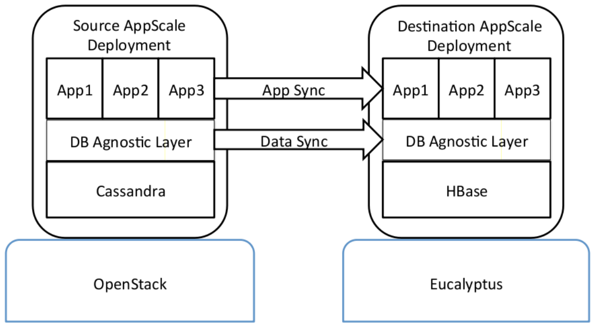

Figure 1: Live migration in AppScale.

tores, supporting datastores such as Cassandra, HBase, Hypertable, and MySQL Cluster.

Figure 1 shows an example of a live migration of two different AppScale deployments, where the underlying IaaS system and datastore used are being updated. In this paper we address the need to be able to move applications and tenants from one PaaS deployment to another, and to leverage the elasticity of private cloud infrastructures to perform live migrations.

In the sections that follow, we first provide background on AppScale and its data model. We then describe the requirements, design, and implementation of our live migration system and show an evaluation of our live migration support between two deployments of AppScale, where we transition the datastore used from Cassandra to Hypertable. Our evaluation looks at several components of the system, including the synchronization of our distributed transaction manager, datastore performance, and switchover time.

2 Background and Related Work

AppScale provides GAE application portability as well as infrastructure and datastore agnosticism. It provides this portability by implementing the GAE APIs, doing so scalably and with fault tolerance. While there are many APIs supported by AppScale for GAE compatibility, the only system state that requires migration is the datastore, as the other APIs are stateless or have no impact on correctness if transferred to a secondary deployment. Yet, for performance reasons we also address the preloading of memcache, a distributed memory caching system meant to alleviate load on the datastore, as to prevent having a cold cache upon the traffic handover.

Infrastructure agnosticism comes by the way of how AppScale is packaged as a virtual machine image. Any virtualization technology capable of running a Ubuntu virtual machine image can run AppScale (e.g., Xen, KVM), and any IaaS that is EC2 compatible (e.g., Eucalyptus, OpenStack) allows for AppScale to be automatically deployed over a varying number of nodes at initialization.

AppScale employs an abstraction layer above the datastore, allowing for the plugging-in of a variety of NoSQL technologies, which are automatically deployed at initialization. We contribute a unifying data migration layer that now allows for the ability to do rolling upgrades to new versions of the existing datastore or an entirely different datastore, a feature that many NoSQL datastores do not currently support.

The datastore layer within AppScale was extended to provide ACID transaction support, regardless of the underlying datastore [1], via a distributed coordinator. Lock granularity for transactions is at an "entity group" level, where entities that share a common root entity are within the same group. These groups are detailed by the developer within their application, and cannot be changed thereafter without deleting the entities.

Moreover, the query support in GAE, and thus AppScale, is limited to only scalable and real-time operations. There is no support for JOINs or queries which can do INSERTs, and hence all queries perform read-only operations. Since queries which can be performed in GAE are derived from the ability to do range queries on the datastore, certain queries are not allowed, such as inequality filters on multiple properties.

Related work includes Albatross [4], a migration technique for moving tenants in a cloud system between deployments. While we can also provide per-tenant movement between deployments, our data model allows for the capability to update the software stack at multiple levels, all while maintaining backwards compatibility with running applications. Furthermore, while much research has been done in VM migration [6], it does not address the problem of performing software stack upgrades above the IaaS layer or allow for per-tenant migration.

3 Design and Implementation

Live migration of data must adhere to certain requirements, such as high availability, backward compatibility, a minimal number of failed transactions, and minimal performance degradation. We have designed and implemented our PaaS migration techniques within AppScale with these requirements and metrics in mind. To do so, we leverage existing components, including the datastore-agnostic transaction support. Migration requires multiple phases, in which state is synchronized between two separate deployments. Figure 2 shows the different stages required to make a full transition from the current deployment to the next.

3.1 Migration Initialization

The first steps in our migration process require the configuration and deployment of a secondary AppScale instance (N_2), initiated by the primary AppScale instance (N_1). N_2's firewall is opened up to allow access by N_1 to control N_2's network channels (such as SOAP servers).

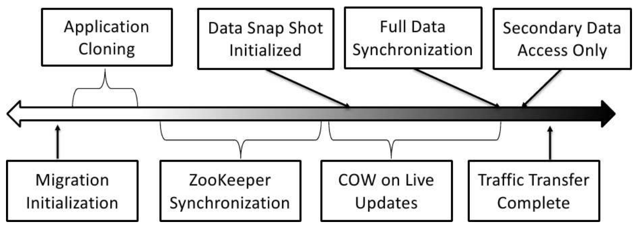
Application Cloning
Data Snap Shot Initialized
Full Data Synchronization
Secondary Data Access Only
Migration Initialization
ZooKeeper Synchronization
COW on Live Updates
Traffic Transfer Complete

Figure 2: Timeline of the migration process.

Once N_2 has been successfully initialized, N_1 utilizes the AppScale command-line tools (a toolset which cloud administrators can use to interact with AppScale deployments) to upload copies of the applications running in N_1 to N_2. At this point, no data has transferred and the applications themselves, while running, are not being accessed by users. We currently do not support the uploading of new applications to N_1 while the migration is taking place.

3.2 Metadata Synchronization

ZooKeeper is a distributed coordination system that AppScale employs to manage state between different services within a deployment, as well as for locking to provide transactional semantics, as explained in [1].

After the N_2 ZooKeeper instances are up, nodes are automatically synchronized with N_1 for new updates, as a consensus is required via the Paxos algorithm once they have joined the cluster. Existing data is then made available to the new ZooKeeper nodes by doing the synchronization functionality in a depth-first search. ZooKeeper nodes are decommissioned at N_1 after the full migration is complete.

3.3 Memcache Warm-up

Our objective for memcache is to have a warm cache in N_2 by the time the handover takes place. For this we do not require full synchronization but a best effort to keep all relevant and most recently used data in the cache. We achieve this by employing copy-on-write (COW) and also copy-on-read (COR) for memcache updates to N_2. The local read or write happens in parallel to the remote write to minimize overhead. We do not do asynchronous updates as to adhere to cache coherency when entries are invalidated. This step is initiated as soon as N_2's memcache system is operational (not shown in Figure 2).

3.4 Data Synchronization

After synchronizing the metadata in N_1, we can now synchronize application data. The data access layer at each node has a REST interface that signals the current stage of migration the process should be in. Each datastore process on each node is sent a message containing the IP address of N_2. Upon receiving this message, any writes or deletes are forwarded to N_2 in a copy-on-write manner.

It should be noted that because of the GAE Datastore API's transaction semantics, writes and deletes are always part of a transaction, even if the transaction is only a single operation. Therefore, each operation requires that a lock be acquired and held through ZooKeeper (which is shared state between deployments). Furthermore, COW updates are done in parallel with local writes to the transaction journal and datastore, to minimize latency.

Transactions must always verify that if it started during normal operation that it did not transition into COW mode mid-transaction upon being committed. If so, the transaction must be retried to ensure that its state is successfully synchronized with the secondary deployment via COW. By default, failed transactions will retry up to three times, before they permanently fail.

Once all datastore access layers acknowledge they are in COW mode, then the datastore snapshot process can begin. COW updates start before the snapshot is started and proceeds during and after, as to make sure no new updates are lost. The updates themselves are SOAP calls to a migration service running on N_2 which uses the datastore agnostic API.

A full snapshot of the datastore consists of serializing each table into a set of flat files which are then compressed. Each independent file can be loaded into N_2 in parallel as an optimization, yet we currently do it serially for simplicity.

The completed snapshot is then copied over to N_2 where it is loaded into the datastore via the datastore agnostic transaction layer [1]. Updates are done transactionally, where the key is first checked to make sure no live updates were done to the entry before updating it. This is only possible because ZooKeeper state is currently shared between deployments. If an entry has been updated during a live datastore write, the snapshot version is simply ignored, as it is stale data (a journaled version will still be available if a rollback is required). Furthermore, it is not possible for an entry to be loaded into N_2's datastore while an ongoing transaction is in place at N_1. N_2 will fail to get the lock on the given entity group and will exponentially backoff until the lock can be attained. After the lock has been acquired, it will then check to see if the given entry already exists, where it will find an entry due to the aforementioned transaction, and thus move onto the next entity to load.

3.5 Traffic Handover

Once full data synchronization as been achieved we then switch N_1 as a full proxy for data access to N_2, making it the primary replica for data access. This step is required as we make the transition onto N_2 for the traffic handoff.

We have two stages of traffic switching. The first stage does permanent redirects at the proxy routing layer (nginx), but because we cannot guarantee that all proxies on all nodes force redirection at the same time we require the full data-proxy stage to make sure there is no case where a user who has not been routed over does not see updates made by a user at the secondary deployment (independent updates at N_2 are not synchronized back to N_1).

Second, we use DNS updates to make sure that the secondary deployment has subsequent traffic from new users. DNS updates alone do not suffice, as many clients cache the DNS entry and it may take ample time before it refreshes its entry. Amazon's Route 53 was the DNS service we used because of its high availability and scalability. Modifications to the DNS was done using their RESTful API which allows for dynamic updates. Our updates consisted of updating the resource record field to point from N_1's IP to N_2's IP.

3.6 Fault Tolerance

In a distributed setting we are able to leverage AppScale's current fault tolerant capabilities for live migration. If transactions fail during a live migration the transaction handler identifier is recorded into ZooKeeper which is shared state between deployments. Any reads of an entity that has a blacklisted transaction identifier is ignored, and the correct version identifier, which is saved in ZooKeeper, is fetched from the transaction journal. While data is currently checked with md5 hashes when transferred across nodes to prevent data corruption, we do not handle Byzantine faults.

4 Evaluation

In this section we measure the overhead associated with live migration between one AppScale deployment to a secondary. We do so with two single node deployments of VMs with 7.5GB of RAM and 4 CPU cores. The initial deployment had Cassandra 1.0.7 as its storage layer, while the secondary deployment had Hypertable 0.9.5.5. The testing application was a GAE application with a RESTful interface. Reads and writes were done based on parameters passed to this application per request.

We first measure the time to synchronize our locking system with ZooKeeper. Next we empirically evaluate the time taken to upload different sized entities from a snapshot. Furthermore, we look at the overhead of updates to both the datastore and memcache which occurred during live migration. Lastly, we quantify the latency associated with a switch over using Amazon's Route 53.

4.1 ZooKeeper Synchronization

Table 1 shows the time taken for synchronizing a node given a different amount of ZooKeeper nodes in which transactional lock states are stored. The number of root entities signifies the number of locks required, and thus need to be synchronized. We see that the time taken on average has sub-linear growth as the number of entries grow while maintaining a relatively low standard deviation.

Lock Count	Min	Mean	Stdev	Max
1000	4.44	4.49	±0.03	4.53
5000	6.57	6.58	±0.04	6.62
10000	8.35	8.47	±0.07	8.53
50000	23.6	23.8	±0.13	24.0
100000	42.8	43.0	±0.22	43.4

Table 1: Time in milliseconds required for lock synchronization on a new ZooKeeper node with a varying number of lock entries.

State	Read %	Min	Mean	Stdev	Max
N	20	46.4	107.2	±23.6	240.2
N	50	44.5	100.2	±24.5	223.9
N	80	44.3	94.0	±24.1	348.1
M	20	43.7	115.8	±32.4	536.5
M	50	45.7	103.3	±27.1	274.5
M	80	47.0	94.2	±23.0	233.1

Table 2: A comparison of time taken for request in milliseconds between normal operation (N) and live migration (M) for different workload percentages of reads versus writes.

4.2 Memcache

We measured the time taken for migration of reads and writes to memcache and measured the overhead compared to normal operation. For entity sizes of 5KB, we found that COW added 0.17ms of overhead, while COR added 0.85ms, both adding less than one percent overall overhead per user request when doing both local and remote updates in parallel. COR added slightly more over overhead because writes are 10.3 times longer compared to a local read.

4.3 Datastore Performance

Table 2 has a comparison of the average latency with different workloads, from a 20/80 read-to-write ratio, to an 80/20 ratio. We compare the initial state (pre-migration) and during migration with 100,000 updates. Load is generated using the Apache Benchmark Tool with a concurrent setting of 10 requests which maxed out all the CPU cores. Read heavy operations see the least amount of overhead as it does not require copy-on-write operations with the secondary deployment. Figure 3 shows there is more overhead associated with write heavy workloads, yet because updates to the remote deployment are done in parallel with the local writes to the datastore, we minimize the additional required latency. Overall we see the overhead at an average of 7.4% with write heavy workloads, while being negligible for read heavy workloads at 0.2%. The most write heavy workload also sees a longer

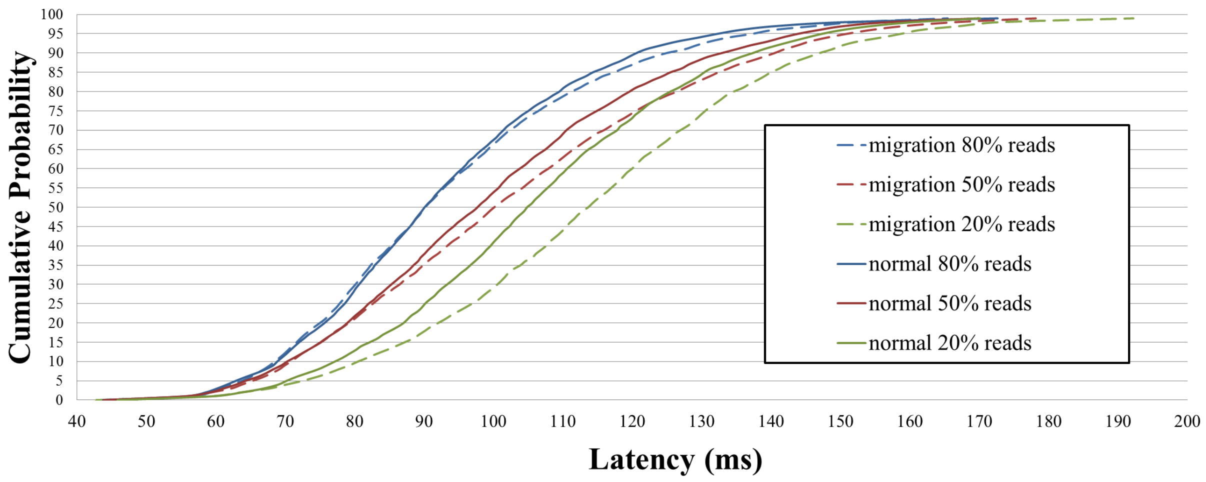

Figure 3: CDF of latency of different work loads comparing normal operation to live migration. The x-axis is latency in milliseconds.

Size	Min	Mean	Stdev	Max
100B	1.72	2.45	±0.92	20.53
500B	1.71	2.29	±0.80	19.08
1KB	1.70	2.43	±0.97	17.03
5KB	1.78	2.62	±0.75	12.53
10KB	1.79	2.71	±1.09	18.26
50KB	1.82	2.88	±1.03	20.02
100KB	2.17	3.18	±1.00	24.45

Table 3: Time for transactionally loading entities of different sizes into Hypertable through the datastore agnostic transactional layer. Times are in milliseconds.

tail past the 95th percentile, from 170ms to 190ms. For both scenarios no failed requests were reported.

Table 3 presents the time taken for different entity sizes when loaded from a snapshot. For this experiment 10,000 updates of each size were loaded and measured. These times include the time to acquire the lock, to check if the current key had an existing value, and to do the write. There were insertion times over 20ms as the max times show, but these were well into the 95th percentile (CDF not shown).

4.4 Traffic Handover

We use the AWS REST-based API to dynamically update the resource record names in Route 53. We measure the switchover time with the Apache Benchmark Tool which continuously sends HTTP request to the initial deployment. The average time to switch over was 46.4 seconds with a standard deviation of 0.97 with a total of 10 trials. The time measured is the difference between when the first HTTP request appears in the access logs of the secondary deployment to the the initial time the API request was sent.

5 Conclusion

In this paper we have designed, implemented, and evaluated a PaaS live migration technique that provides minimal performance degradation and little to no service disruption. As part of future work, we will evaluate different combinations of rolling upgrades throughout the cloud stack, as well as migrations across WANs.

References

[1] Chohan, N., Bunch, C., Krintz, C., and Nomura, Y. Database-Agnostic Transaction Support for Cloud Infrastructures. In *IEEE International Conference on Cloud Computing* (July 2011).

[2] Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., and Wolski, R. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *ICST International Conference on Cloud Computing* (Oct. 2009).

[3] Cloud Foundry. `http://cloudfoundry.com/`.

[4] Das, S., Nishimura, S., Agrawal, D., and El Abbadi, A. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow. 4* (May 2011), 494–505.

[5] Keahey, K., and Freeman, T. Nimbus or an Open Source Cloud Platform or the Best Open Source EC2 No Money Can Buy. In *Supercomputing 2008* (2008).

[6] Liu, H., Jin, H., Liao, X., Hu, L., and Yu, C. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing* (New York, NY, USA, 2009), HPDC '09, ACM, pp. 101–110.

[7] Nosql databases. `http://nosql-databases.org`.

[8] Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., and Zagorodnov, D. The Eucalyptus Open-source Cloud-computing System. In *IEEE International Symposium on Cluster Computing and the Grid* (2009). `http://open.eucalyptus.com/documents/ccgrid2009.pdf`.

[9] OpenShift. `https://openshift.redhat.com/`.

[10] OpenStack. `http://openstack.com/`.