Garuda: a Cloud-based Job Scheduler
Ashish Patro, MinJae Hwang, Thanumalayan S., Thawan Kooburat

We present the design and implementation details of Garuda, a cloud based job scheduler using Google App Engine as the underlying cloud provider. The goal of the project is to demonstrate the concept of a centralized cloud based job scheduler that manages a pool of worker machines and schedules the jobs submitted by the users to these machines. We chose App Engine to implement the scheduler and investigated the pros and cons of choosing App Engine as a platform for this purpose. We also evaluated the performance of the central scheduler within the various limits imposed by the App Engine’s free service and discuss our design choices in this setting.

1 Introduction

Cloud computing refers to the internet based computing service provided by various infrastructure providers on an on-demand basis. Examples of popular cloud computing services are Amazon's EC2, [2] Microsoft’s Azure [3] and Google App Engine (GAE). [4] Cloud Computing has become a very popular topic today due to a lot of benefits provided by such an infrastructure. In such a case, hardware, software and other services are available to the clients as a utility on an on-demand basis and they are charged proportionally to the amount of resources consumed by them. In some cases, the providers use a portion of their datacenter infrastructure for their purposes and provide the spare capacity as a cloud service to the clients. Such a setting allows the cloud providers to multiplex their resources efficiently and earn money from such deployments. To the clients, it is beneficial because it allows them to concentrate on their application/service and not worry about the infrastructure required to deploy their services. Thus, cloud services lower the entry barrier for new services/applications. Also, cloud services allow the clients to scale their services efficiently according to the demand for their service in an on-demand manner.

Given the rising popularity of cloud services, companies are increasingly looking towards cloud computing as an infrastructure solution. The various cloud services differ in the interface that they provide to the clients. For example, Amazon provides virtual machine instances to the clients and the clients have the flexibility to use the virtual machine instances according to their wishes. On the other extreme is the App Engine service provided by Google, which allows clients to host web applications on Google's infrastructure. Google scales the application's resources automatically according to the traffic and the data used by the web application. Thus, Google provides automatic scaling, fault tolerance and reliability to the web application making it easy for web application developers to deploy and run applications. App Engine also provides access to some other popular services from Google such as Memcache, Datastore and the Google accounts. We planned to explore Google App Engine, a popular cloud service during the project as solution to our problem.

The goal of the project was to design and deploy a centralized job scheduler on Google App Engine service. The result of the project was Garuda - a cloud based centralized scheduler that schedules jobs submitted by clients onto a pool of worker machines. The design of Garuda is based on Condor [1], a popular high throughput computing
platform that harvests computing cycles from idle workstations. We used the ClassAd mechanism provided by Condor and moved the matchmaking logic onto a central cloud based infrastructure. In such a scenario, the worker machines and the clients send their ClassAd requests to the central scheduler and the central scheduler machines the requested jobs onto the available machines.

The rest of this paper is organized as follows: Section 2 presents various background knowledge related to our work. Overview design is presented in Section 3. We describe our implementation in Section 4. In Section 5, we talk about evaluation, and discuss about GAE platform in Section 6. Finally, we state our conclusion in Section 7.

2 Background

2.1 Condor

Condor is a job scheduling system for a pool of workstations, along with a set of related policies and a framework for the scheduling. Condor's scheduler is based on a distributed model - responsibilities are delegated to various entities (typically 'daemons') distributed among many nodes. For example, a 'schedd' daemon manages a job queue and a 'negotiator' daemon performs matchmaking. In the Condor system, the term 'Client' represents the source of job descriptions that need to be matched and the term 'Worker' represents a machine where jobs can be assigned. Worker and job requirements are expressed in a format called 'ClassAd', with a built-in mechanism for the matching between jobs and machines. A 'startd' daemon runs on each participating worker machine, and periodically submits the worker's ClassAd to a 'collector' daemon. The collector daemon runs on a special node called the central manager. Another daemon, called the negotiator, pulls worker ClassAds from the collector, pulls job ClassAds from the schedd daemon, and performs matchmaking. It also notifies both schedd and startd about the matching. In the entire system, most of the activities take place without the involvement of the central manager. While this improves scalability and performance, it makes our design complex.

This design principle is suitable for distributed systems as it prevents a single point of failure. Additionally, it also gives each resource owner authority over their own resources. However, the system is complex because of fault-torrent logic built into each daemon and the fact that the number of daemons grows as more functionality is added into the system. The rest of the report will use the terms 'Client', 'Worker' and 'ClassAd' in a context similar to the Condor system.
2.2 Cloud Platforms

While planning to design a cloud based job scheduler, we were faced many platforms. On one end of the spectrum was Amazon EC2, which provides on-demand virtual machines. EC2 provided us with complete control over spawned VMs. On the other end, we had Google App Engine (GAE), a web hosting platform. However, GAE also provides automatic scaling, load balancing and fault-tolerance capabilities for the web application. We decided to use GAE as our platform for hosting the job scheduler, since it would allow us to concentrate on the application logic rather than scalability or reliability issues. In addition, GAE also provided us with other facilities such as Cron Jobs, Task Queues, Email and XMPP, some of which we believed could be used in our design.

3 Design

The basic design of the system is based on a simplified and centralized version of Condor's job scheduler. The final system should provide users (and machines) with a central manager to which they can submit job and machine descriptions. The central manager should try to match job's descriptions with machine's descriptions, schedule jobs on participating machines, make sure that jobs get completed and notify users when the jobs are done. Our system is only concerned about the job scheduling part - the design of the central manager is independent of other issues that are at the core of a remote job execution platform, such as the data backplane. These issues are considered orthogonal to the goal of this project. We believe that, for a basic system, such problems could be solved with out of box solutions (such as a shared filesystem for the data backplane). Figure 2 portrays the interaction of the central scheduler with the rest of the system.

3.1 Design Choices

In essence, a job scheduler processes a large amount of data in order to provide a matchmaking service. If we could implement the scheduler as a single process, or a couple of processes over which we have complete control, we can store and process the information in memory. However, in a traditional web application, in order to process large amount of data, the application must translate the data processing logic into SQL statements and delegate the task to the database. On GAE, a service needs to be designed with a 3-tier architecture. For the internal design of our
system, we had some choices concerning the nature of the match-making logic and how the scheduling data is organized within the central manager. As far as the actual scheduling goes, the basic choice was on where the matching logic had to be placed - it could either be incorporated into the application tier or the data tier. While the application tier will allow us to customize the matching logic extensively, placing the logic in the data tier will probably lead to a more optimized solution. Another choice was as to when the match making should actually take place. The match making could either take place every time a Client submits a job ('online scheduling'), or batches of jobs can be scheduled in fixed intervals('batch scheduling'). Batch scheduling might lead to lesser consumption of resources, while online scheduling might lead to lesser latencies for the matchmaking. Apart from scheduling, another set of choices need to be taken with respect to each type of data maintained (Machine ClassAds, Job ClassAds, Machine states) by the central manager. GAE allows data to be stored in three 'types' of storage, which can be thought of as a memory hierarchy. For each type of data, we need to decide on which type of storage is to be used. The next subsection will describe the different types of storage.

3.2 Memory Hierarchy

We visualize the 'memory hierarchy' of GAE to be composed of three levels. The first, topmost level ('local memory') is just the direct, physical memory that can be accessed by the web-request handler (servlet). This memory cannot be assumed to be persistent across different web-request handler invocations. The only guarantee that GAE provides with this type of memory is that it will be alive throughout the scope of handling of one web request instance. Moreover, there is no guaranteed, direct way to access the memory corresponding to one invocation in another invocation. However, writes and reads to this memory are very fast, since they directly access the physical memory.

The second level in the memory hierarchy is provided by Memcache. This is similar to local memory, but differs in some important ways. GAE tries to keep the data stored in this memory persistent across different invocations of the web-request handler, and provides a way to access the memory in different invocations. However, GAE does not guarantee that the data stored in the memory will be persistent, even within the scope of a single request handler invocation. Moreover, the API for accessing this memory only allows set and get method, and only serializable objects can be stored in Memcache.

The third level in the memory hierarchy is the Datastore - it provides a RDBMS-like model, with a very constrained set of features. The query language used to retrieve data from the Datastore tries to resemble SQL, but again provides a very constrained set of features compared to the standard SQL. However, the Datastore provides a persistent storage model, provided neither by local memory nor Memcache. Although GAE provides another form of persistent storage too (BlobStore), its only advantage compared to the Datastore or the Memcache seems to be handling large collections of persistent data. Since this is not relevant to the scope of our application, we do not discuss it in detail.

Since a practical study of GAE which evaluates various characteristics was not previously performed, it was not immediately apparent as to which of these choices will result in optimal performance. Hence, we chose to
evaluate a combination of the design choices that we believed would result in an efficient system, and to change the choices as we discovered the nature of GAE from our experience. In fact, our experience suggests that the three levels of memory described in the previous section might have very different characteristics from what one might naturally expect.

4 Implementation

As explained in the previous section, we had to experiment with many variations of the design. However, all the variations shared a skeleton model. Job ClassAds and machine ClassAds were received from the clients and the workers respectively through a normal web request (HTTP). For notifications that had to be sent from the scheduler to the client or the workers (such as job completion notifications to the clients and job scheduled notifications to the workers), XMPP was used. For tasks in the scheduler that need to be done at regular intervals (such as periodic batch scheduling), the Cron job feature of GAE is used. The following subsections will describe each variation of the core logic.

4.1 First Variation

In this version of Garuda, we try to store data entities that need not be persistent in Memcache. We notice that only the machine state information and the job-related information need to be persistent. Machine ClassAds are periodically updated by the worker machines; as a result, they can be stored in Memcache. This should result in a low consumption of Datastore CPU cycles (GAE has a hard limit on Datastore's CPU cycles which we could easily reach if we store ClassAds into Datastore). When Garuda receives a machine ClassAd, it parses the ClassAd and serializes into a binary array before storing to Memcache. This is done in order to reduce the cost of retrieval during match making. However, Memcache only provides a get and set interface, so we need to use exact key name in order to retrieve object from Memcache. We avoid keeping track of exact keys by allocating a range of keys for storing all machine ClassAds. The key range is actually an integer number ranging from 1 to 10K. When Garuda wants to store machine ClassAds into Memcache, it will retrieve a key associated with each machine from the Datastore. If this is a ClassAd of a new machine, it randomly selects a key from the key range until the selected key does not collide with existing ClassAd. To perform batch scheduling, the scheduler will try to retrieve objects from Memcache using all the possible keys. The overhead of specifying a large amount of non-existing keys seems to be low, so this method proves to be quite efficient. We can consider this design to be a trade-off between the correctness of matchmaking and performance, since we might not be able to match a job with all available machines if some machine ClassAds were dropped from Memcache.

4.2 Second Variation

Although the first prototype successfully reduces the amount of data retrieval from Datastore, we suspected that the overall data retrieval cost was still high, and so it would not be efficient to do online scheduling. At this point, the only way to reduce the overhead of data retrieval is to store data in local memory. Storing data into local memory is tricky in a GAE-style Java servlet, since the servlet is not supposed to have any state by itself. This restriction is due to the fact that Java servlets can be created in any machine (within the Google cluster) and can be
destroyed anytime after it becomes idle. We decided to investigate how a Java servlet is created, managed and, destroyed. Our intuition was that, this information could be used to actually store more data into the local memory. We found that every Google App Engine project is hosted as JVM processes and each JVM process handles at most connection at a time; thus, programmers do not need to worry about thread safety. We also found that a JVM process will not be killed if it processes at least one request in every 110 seconds in Google App Engine hosting environment. Thus, if we keep "pinging" each JVM process, we can store data in local memory. We will describe this trick in Discussion section in detail.

4.2.1 Communication between JVM processes

Even if we have a permanent local memory, there is still the problem of synchronization between JVM processes. Since a free Google App Engine account provides 10 JVM process per user, these 10 JVM processes should kept synchronized. We used a Memcache counter to solve this issue. Basically, we assign a counter as a version number to local memory. Whenever the local memory is updated, the JVM process updates the counter by one and writes update log to Memcache. Whenever a JVM process reads the local memory, it should check if it has the most recent version. Thus, for each request, the JVM process reads one counter from Memcache. If it is not the most recent one, the JVM process tries to update from current version to most recent version by reading update logs. This process is quite similar to the behavior of a version management system. This update-log algorithm guarantees 100% consistency if there is no Memcache failure. In other words, it is as consistent as Memcache. Considering each Machine ClassAd is about 5KB, the amount of saving in bandwidth is substantial if online scheduling is performed (simple calculation: without the local memory, the scheduler requires 500MB of data movement with the Memcache module for 100 job-matching requests on 1,000 machines). In this variation, we assume that the update frequency of Machine ClassAds is quite low. It will mostly be updated when there is new system to be added or to be pulled off.

4.3 Third Variation

After evaluating the previous iterations of the job scheduler, we suspected that the performance of the scheduler was being degraded a lot because of the time spent in parsing and unparsing ClassAds. We also suspected that fitting the ClassAd matching logic somehow into the Datastore query model provided by Google would improve efficiency. The ClassAd matching that was being used was too complex to directly fit into the query model, and so we decided to experiment with a simpler version of ClassAd.

We created a model that would just match a fixed list of job ClassAd parameters and machine ClassAd parameters with simple relational operations like equal to, greater than, less than and contains. This would remove a lot of features that were provided by the ClassAd logic - requirements could not be mentioned in the form of expressions, and a ranking logic for the matching was not present. However, if a list of machine ClassAds were stored in a table in the datastore, matching a job ClassAd can be performed with a single query to the Datastore. This version of the job scheduler turned out to be relatively simple to implement. It also consumed less CPU cycles, and
could support online scheduling. The downside, as already explained, was the lack of features compared to a Condor-style matchmaking logic.

5 Evaluation

We decided to evaluate each variation by the number of jobs that could be scheduled per second. In addition, we also evaluate the amount of money that needs to be spent for matching in the GAE platform. We suspected that the number of jobs that could be scheduled per second would vary with the number of worker machines, and hence we plotted the jobs per second compared to the number of machines in each version. The actual experiments were performed by supplying the central manager with a fixed number of machine ClassAds, and then trying to schedule (in online mode) 200 jobs (beyond which evaluating the performance consumed our allocated GAE quota too fast) in 20 threads. If the variation could either match the job online, or if it believed that machines were unavailable for the given requirements, then we considered the job to be 'scheduled'. Sometimes, the variation could not do either of these within the limits (such as the datastore cpu-per-second limit) imposed by GAE. These things were considered 'errors', although practically, the job will be queued and will later be scheduled in batch-mode.

![Figure 3: Job requests per seconds](image)

Considering the number of jobs that could be scheduled per second, we find that for a low number of workers, Version 2 performs markedly better than the other two versions. However, as the number of machines reaches around 400, the performance degrades a lot (the number 400, we believe, actually depends on the quota limit imposed by GAE, and might be different if we had a 'Billable Quota' which has different limits). Variation 1 had a lower performance compared to Version 2, but followed the same trend. This is quite expected, since Version 1 and 2 compare the given job ClassAd with every available machine (in order to obtain a rank, Condor-style) before actually scheduling the job. Version 3 had a consistent performance with respect to the number of machines. This was also expected, since the scheduling logic of Version 3 does not depend on the number of machines. However, we had hoped the throughput of Version 3 to be a little higher.
For the number of job requests which resulted in an error, it was found that Version 1 never produced an error when 200 jobs were being scheduled. This was because Version 1 uses Datastore minimally, and uses MemCache (which has a much higher effective quota limit) instead. The Datastore quota seemed to limit the other two versions - the error percentage of Version 2 seemed to increase very fast beyond 400 machines for Version 2, while it remained consistent for Version 3.

It is important to understand the characteristics of the different versions from these evaluations - the decrease in the performance of Versions 1 and 2 is because they incur a scan of all the ClassAds for each scheduling. If however, we discard the Condor ranking logic (while still making use of ClassAds), or adopt an 'approximate ranking' approach (such as choosing the best suited machine among the first hundred machines), we will still be able to obtain a high throughput system. Version 3, while providing a lower throughput initially, will probably scale a little more, and offers the guarantee that the job will be immediately scheduled for sure if there is at least one free worker matching its requirements (Versions 1 and 2 do not guarantee this).

Figure 5 represents the expected amount of cost for scheduling jobs and registering machines. The data was derived by observing the amount of CPU cycles and Datastore CPU cycles being consumed and multiplying them with the price table provided in the GAE documentation. The scheduling price represented in the figure corresponds to scheduling 200 jobs while 1600 worker machines were present, while the registration price represents the price for registering 1600 machines.
6 Discussion

6.1 Datastore Limits

Datastore is the persistent and reliable storage mechanism provided by Google App Engine. Our experience while using Datastore for Garuda was not very pleasant. While using the Datastore API in the project, we found that it consumed a lot of CPU cycles for small operations. For example, each insert operation for a 5KB ClassAd took around 0.22 CPU seconds to complete. We found that we were frequently hitting the Datastore hard limit of 20 CPU-sec/sec for GAE’s free version. We also found that Datastore rejected the requests on the high contention cells. But Datastore was a faster choice while retrieving large amounts of data (even faster than Memcache). Datastore provided query facilities but the query predicates required pre-computed indices before the query was performed.

6.2 Memcache Limits

Like Datastore, we experimented with Memcache by using it for various components in Garuda’s design. While working with Memcache, we found that it had an upper limit in terms of the amount of space that could be used by an application. We observed that we could store about 10,000 ClassAds of size 5KB without any problems but anything beyond that limit frequently resulted in Memcache misses. This gave us a sense of the limit till which we could use Memcache without any problems. However, the memory limit imposed by Memcache was never an issue for our application. In fact, it was the data retrieval time that was a cause of concern. We found that we could retrieve about 1,200 entries of size 5KB within a single HTTP request before a timeout (30 seconds) occurred. Compared to Datastore, there was no notion of indices or group-query in Memcache as each retrieval/write was a get/put operation.

6.3 Persisting Servlets

In this section we describe how we could 'hack' GAE into persisting servlets by utilizing some hidden mechanics. We conducted many experiments that allowed us to discover key mechanisms required to persist the servlets. These experiments uncovered a lot of interesting properties about the Java platform that may be important for other developers, and was used extensively for variation 2 of Garuda.

6.3.1 JVM Process

GAE spawns JVM processes to host Java servlets. This information can be obtained by looking at the GAE logs. When GAE spawned a new process, requests took a bit more time than usual and GAE logged the creation of a new JVM process. Also, we could observe the creation of multiple JVM process under the presence of heavy workload. We verified that one JVM process contained all the Java classes present in a single GAE project and that the address space is shared within a project. From our observation, JVM process does not handle more than one request at a time, which suggests single thread per process. This is reasonable because it eliminates burden of thread safety.
6.3.2 Identifying each JVM Process

Since there were multiple JVM processes, our first question was how to determine how they could identify each other. We assigned a random number to each Java servlet when it was first created. It could be done within the constructor of each servlet. Each servlet returned its identifier when it got a request. Thus, we could see which JVM process handled a given request. We could precisely detect the new creation of new JVM processes but could not detect the destruction of a JVM process. There is a destroy() callback in servlet interface but current GAE does not call it. So our estimation of lifetime of servlet is based on guessing, not a hard fact.

6.3.3 GAE Load Balancing and Queuing

We observed that GAE does not use round-robin scheduler but redirects a request to most recent used one. From our observations, we hypothesized that GAE recorded the last access time to each JVM process and scheduled a request to most recently used one first. In this way, GAE could maintain a minimal number of JVM processes and kill JVM processes that were idle for more than 110 seconds.

Also, we observed that if all the JVM process were busy, it queued up the waiting requests upto 100 requests (for free account). If queues were full, GAE simply returned a HTTP 500 error.

6.3.4 Counting Number of JVM Process

Our next question was obtaining the count of the number of available JVM process. When a request was sent to GAE, it will be most likely handled by the most recently used one. Under a light amount of load, requests were handled by a few number of JVM process. Even, if we we intentionally put a high load for GAE, we could not be sure how many JVM processes were there.

Since each servlet could process one request a time, we could do some tricks. A servlet can send a HTTP request to another servlet; this is an intrinsic feature for web applications. Thus, if a servlet sends a request to itself, this request should be redirected to another JVM process. If all the available JVM processes are busy, the requests will get queued up and the caller will see a timeout. Then, the depth of recursive call is the number of JVM process available

6.3.5 Keeping JVM Process Alive

If we want to use local memory as a cache, we need to know when exactly JVM process is destroyed. Since we can count number of current JVM processes and enumerate each JVM process by looking at servlet identifier, we can detect "the death" of JVM processes. We found that GAE killed a JVM process when it was idle for more than 110 seconds.

Now, we knew that GAE did not kill a JVM if it received a request within every 110 seconds. So, if we could make an artificial request to each and every JVM process periodically, then no JVM process would get killed. We used the same trick used for counting number of JVM process to ping each JVM process. We could observe that GAE did not kill any JVM process even there was no actual workload.
6.3.6 Memory Size

Each JVM processes are allowed to use around 110MB of local memory. Thus, if we keep 10 JVM processes alive, we get approximately 1.1GB in aggregate local memory.

6.3.7 Other Services

An added feature of Google App Engine is the set of useful utilities available. Examples of such services are XMPP and Email notification. Moreover, when a job has been matched, a mechanism is needed to notify the workers. One way to notify the workers is through an explicit polling mechanism by the workers. Moreover, for notifying the clients about job completion and status updates, it requires periodic requests from the clients which can be pretty annoying to them. A better way to solve the problem is to notify the workers and clients by using an XMPP or Email based notification mechanism. Google App Engine provides an excellent XMPP implementation (it took us about 10 minutes to learn the API and get it working). App Engine's email mechanism can be used a backup when XMPP fails.

7 Conclusion

In this paper, we tried to fit a Condor-style job scheduler in the Google App Engine platform. We discovered that adopting the Condor-style match making, as such, does not directly fit the model provided by Google App Engine - we could either change the matching logic (approximate ranking or no-ranking for ClassAds, or adopting the matching logic fully to the Datastore model), or design such that the service "scales with money". We also delved into the internals of Google App Engine, and explored it's characteristics. We discovered a set of 'hacks' in GAE which can be used to efficiently store and retrieve data for which consistency is not very important. We evaluated and experimented with different design choices for Garuda that were inspired by different priorities and compared their performance on metrics such as number of job request/sec and resource usage on GAE. The actual contribution of our work, through the design of Garuda, is on how a non-traditional web service can be adopted to a web framework model implemented on top of a cloud. Such an adoption will decrease the complexity of the service, and we hope that our experiences can be used in such projects in the future.

8 References


