Polyphony: A Workflow Orchestration Framework for Cloud Computing

Khawaja S Shams[1], Dr. Mark W. Powell.[1], Tom M. Crockett[1], Dr. Jeffrey S. Norris[1], Ryan Rossi[1], Tom Soderstrom[1]

[1]NASA Jet Propulsion Laboratory – California Institute of Technology, Pasadena, CA 91109, USA
KSSHAMS@JPL.NASA.GOV

Cloud Computing has delivered unprecedented compute capacity to NASA missions at affordable rates. Missions like the Mars Exploration Rovers (MER) and Mars Science Lab (MSL) are enjoying the elasticity that enables them to leverage hundreds, if not thousands, or machines for short durations without making any hardware procurements. In this paper, we describe Polyphony, a resilient, scalable, and modular framework that efficiently leverages a large set of computing resources to perform parallel computations. Polyphony can employ resources on the cloud, excess capacity on local machines, as well as spare resources on the supercomputing center, and it enables these resources to work in concert to accomplish a common goal. Polyphony is resilient to node failures, even if they occur in the middle of a transaction. We will conclude with an evaluation of a production-ready application built on top of Polyphony to perform image-processing operations of images from around the solar system, including Mars, Saturn, and Titan.

Index Terms—Cloud Computing, Space Exploration, Distributed Computing, Cycle Stealing

1. INTRODUCTION

Cloud computing is a boon for various applications in industry and academia. It has unleashed a virtually limitless level of compute capacity that various research projects can leverage due to the affordable pricing. The NASA Jet Propulsion Laboratory (JPL) is NASA's premier facility for robotics exploration of our solar system. JPL spacecraft have roamed throughout the solar system, visiting all of the planets in the process. JPL is the first NASA center to partner with commercial cloud computing vendors to investigate cloud computing in its role to revolutionize spacecraft operations.

The motivation for Polyphony is to provide a framework that helps to streamline the operations of Mars rovers by processing and delivering Mars images with remarkably low turn around times. Prior to the advent of cloud computing, our image processing pipeline for missions like MER was designed to rely on a single machine with lots of compute power. Although we have exploited the multiple cores available to us on our machine, we have been bound by the limitations of the machine, which has resulted in extended bottlenecks in the image-processing pipeline. Although the machine has been remarkably reliable, we have run the risk of a single point of failure in the face of hardware or even OS level issues with our processing machine. Prior to cloud computing, adding another machine was untenable strictly from a cost perspective: it did not make sense to add one or five more machines when the current single machine sat idle for roughly 20 hours a day. On the other hand, cloud computing has enabled us to employ hundreds of machines for durations as small as an hour.

While JPL is working with several cloud vendors on our IAAS needs, the bulk of this research was performed on Amazon.com's Elastic Compute Cloud (EC2). EC2 allows us to programmatically request appropriate computing capacity based on the current demands. With cloud computing, it is important to note "using 1000 EC2 machines for 1 hour costs the same as using 1 machine for 1000 hours" [1]. This is a crucial benefit to any application; expediency of computational result comes for free simply due to the elasticity available in the cloud.

2. POLYPHONY

Polyphony is composed of several components that work in harmony. First of all, there is a distributed queue that is used to publish tasks and distribute them to nodes that can perform them. Secondly, we must have at least one subscriber that polls the queue for tasks and performs them: we will refer to this subscriber as a worker node. Polyphony makes no assumption about the number of nodes working on the tasks. Furthermore, it does not have a central component that attempts to track any or all nodes. Moreover, there are no assumptions about physical characteristics of the worker nodes: they can be Linux servers, personal laptops, machines in the cloud, or even supercomputing resources. Lastly, each worker node has a software application that interacts with the queue. This application extensively utilizes Eclipse Equinox [4], a popular implementation of the OSGi [2] specification, for modularity. The OSGi specification enables software developers to componentize their applications. These components, also referred to as plugins, allow clear separation of concern, drastically reduce the complexity of the software, and foster easy reuse of code. In order to integrate an application into the Polyphony framework, a develop has to simply write one module and include it in the Polyphony distributions without having to know anything else about the underlying details of task allocation, scheduling, or distributed computational resources. This is discussed in more detail in

IEEE computer society

2C.

2A. TASK ALLOCATION AND DISTRIBUTION

The distributed queue is a core component of Polyphony. In the initial implementation, we employ Simple Queuing Service (SQS) available as part of Amazon Web Services (AWS). However, since a dedicated module is tasked with handling all transactions with SQS, we can replace SQS with another queuing service very easily without impacting any other part of the software.

 SQS is a distributed queue with some elegant concepts that facilitate resilient behavior. It provides a Restful interface to interact with the queue: tasks can be added via an HTTP PUT method, obtained via GET method, and deleted upon completion via the HTTP DELETE method. SQS provides guaranteed-once delivery semantics: each task added to the queue is delivered to at least one node. Upon retrieving a task, the worker node sets the visibility of the message, with a timeout, so that it is invisible to other nodes while it is being performed. Upon completion of the task, the worker node deletes the task from the queue. If a worker node faces a failure that prevents it from completing a task, the task eventually becomes visible to and obtained by another worker node.

Due to the distributed and eventually consistent nature of the queue, some of the tasks may be assigned to multiple nodes and be performed multiple times. While this may seem inefficient, it is a trade off that most applications are willing to make to gain better reliability with node failures and higher throughput. In practice, we have rarely observed a task executing multiple times. Nonetheless, this important characteristic forces the application designer to create idempotent tasks that do not have side effects if they are executed more than once.

2B. WORKER NODES

This section describes the worker nodes and the software architecture of the application that runs on them. As stated earlier, Polyphony makes no assumption about the worker nodes.

 In the experiments that we have conducted thus far, the majority of the compute capacity comes from machines rented on the cloud. On the AWS cloud, we profiled our application with a variety of nodes from the small instances (1 core at 1 GHz with 1.7GB of memory) to the quadruple extra large instances (8 cores with 2.5 GHz each and 68GB of memory). While the individual performance of a particular machine size depends on specific applications, we have observed that the larger instances not only offer higher CPU and RAM capacity, but they have provided significantly higher I/O throughput to the disk as well as the network.

Any organization with desktop computers and servers has significant compute capacity that is unutilized or wasted when machines are idle. One of the goals of Polyphony is to leverage the spare cycles and perform tasks to assist the worker nodes in the cloud. We started with the assumption that these nodes and the results they provide can be trusted. On our Linux servers, we ran the Polyphony client with the Unix Nice command, with a priority of 10. This simple process allows Polyphony clients to steal spare cycles and enable workstations to contribute work at their own pace. Should a server become busy in the middle of the computation, the overall infrastructure will not be impacted as the computation will eventually time out and be assigned to a node that can finish it faster. However, in scenarios where the queue may have thousands or millions of tasks in it, contribution at any pace can be extremely valuable.

We are integrating the supercomputer center nodes as Polyphony clients. While this may not be the most effective approach to scheduling jobs on a super computer, it will enable us to leverage excess capacity on the super computer: in production environment, we would control the number of jobs spawned dynamically based on how busy the super computing cluster is at a given time.

Polyphony makes no assumption about the worker nodes, and our initial runs have served as a proof of concept that dedicated cloud machines and spare capacity on JPL computers. The JPL machines access the NFS server over the Internet. However, we judiciously utilize the security settings to allow network access only to other nodes in our security group and specific CIDRs (Classless Inter-Domain Routing) in the JPL space.
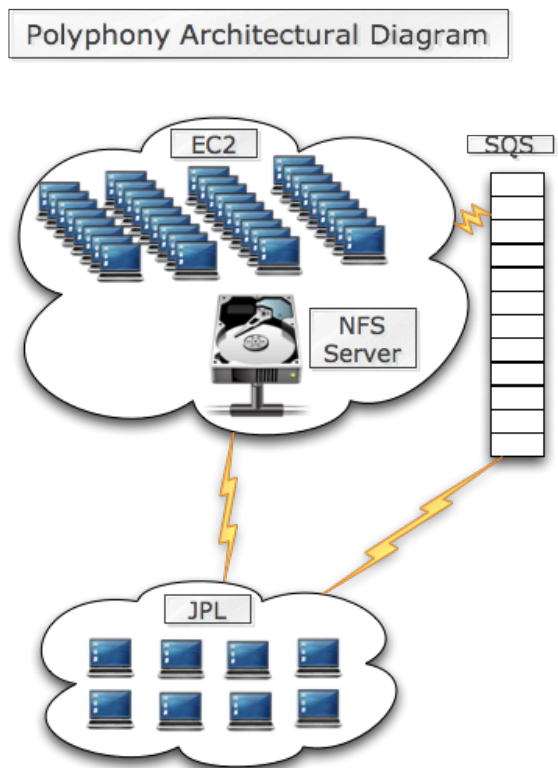

Figure 1. Architecture of Polyphony

2C. Software Architecture for Polyphony Client

The Polyphony client is written entirely in Java, and it utilized the Eclipse Equinox, an implementation of the OSGi specification. Equinox and OSGi offer an unparalleled level of modularity by allowing developers to write modules, or plug-ins that work in harmony in the context of an application. For instance, the Eclipse IDE (Integrated Development Environment) is built on top of this framework, which allows third-party developers to contribute plug-ins to enhance or add functionality to the IDE. Similarly, Ensemble Rest, a modular framework for developing Restful applications open sourced by OPS Lab, leverages Equinox to provide simple hooks that enable developers to deploy a Restlet with a few lines of code and XML configuration. Equinox allows plug-ins to be added even after an application has started and can incorporate a contribution without restarting the application. For example, Ensemble Rest allows developers to add or update a specific set of web applications without restarting the server and exposes the new or updated services immediately.

At the heart of the Polyphony client is the Polyphony engine. Upon the start of the application, the engine solicits all available plug-ins for a distributed queue. In our original prototype, the queuing plug-in interacts with SQS. However, replacing SQS with a different distributed queue would be as simple as replacing a single JAR (Java archive) in our application. The Polyphony engine then acquires tasks in bulk from the queue and attempts to accomplish them. To make this application completely modular, the Polyphony engine makes no assumption about the tasks that it would be able to perform. Instead, it maintains a registry of **task handlers** that are available to it as an application from the included plug-ins. Each task handler is a Java class that implements the ITaskHandler interface with the following methods:
- boolean handles(String task);
- void handle(String task) throws Exceptions

In order to add one or more task handlers to Polyphony, a developer can create a plug-in with implementations of ITaskHandler and register them via an XML configuration file. By simply adding this plug-in to the client, new capabilities are introduced without any need for redeployment or recompilation.

After receiving a task, the Polyphony engine asks each class in the registry if it knows how to handle the task. Upon finding the first handler that can handle the job, the engine assigns the job to the handler. If the job is completed successfully, no exceptions would be thrown and the engine deletes the task from the queue. As part of the execution, a task may chose to add more tasks to the queue, or it may simply terminate successfully by not throwing an exception. If an exception is thrown, the engine simply moves on to the next task so that the failed task can be tried again. The exception could be due to a temporary failure, so it is important to retry the task. Nonetheless, we intend to add functionality that will remove a task from the queue after a predetermined, configurable number of failures.

The Polyphony client is easy to extend, and it enables developers to write TaskHandlers without having any knowledge or assumptions about the distributed queuing system. With this framework, new TaskHandlers can be added to existing Polyphony applications or the underlying queue can be changed seamlessly without impacting any other part of the system.

3. Harmonic: Parallelized Image Tiling for Planetary Images

Embarrassingly parallel problems are the norm in almost any satellite or panoramic image processing application. When a particular operation needs to be applied to every single image in a large collection, it is easy to perform the tasks in parallel across a large number of machines. Similarly, if there is an operation that needs to be applied to an extremely large image, it is often natural to recursively divide the image into smaller regions and perform the task on each region on a different machine.

When dealing with large images, it is typical for an image to have more resolution than what is available at the screen of our end users. Since our operations software runs around the world with laptops with wireless connectivity, it is wasteful to transfer an entire image when the user is only viewing a small part of it or has zoomed out to view the image at a much lower resolution. To make this process more efficient, we tile our images are different resolutions so that we can deliver only what the end user has on the screen. This allows us to effectively utilize the bandwidth and improve the user-experience by delivering the images faster.

Harmonic is an application built on top of the Polyphony framework to streamline the production of tiles by employing a large number of machines. For our initial run of Harmonic, our goal was to tile and scale 184,000 images from the Cassini spacecraft as quickly as possible. When running the tiling software on production machine in serial, it took more than two weeks for the job to finish. For this application, we only processed one image at a time to avoid overwhelming the server. On the other hand, when employing cloud machines, our goal is to utilize every cycle that is available to us. The Polyphony client on the cloud employs thread pools to ensure efficient utilization of all our resources, especially in the face of I/O intensive tasks.

Harmonic was originally designed to run on a single machine, and we modified it to work in a distributed environment. However, due to the embarrassingly parallel nature of the problem, it was very easy to integrate it with Polyphony. To kick off the process, we add 184,000 tasks to the queue: one task for each image that we need to process. The task description has two parts: a prefix that contains a unique ID recognized by the tiling task handler and a URI of the image. For the purposes of our tests, all images were stored on a central server that exposed the storage device via NFS to each worker node. When a worker node receives the task, it reads in

the image, generates the tiles, and writes them back to the central file system.

Harmonic demonstrates that Polyphony can be easily extended to do parallel operations on images. We are currently working on a very similar application that analyzes each image with machine vision algorithms to recognize salient features in images. While we have numerous features and algorithms that we want to execute on our image collection, plugging in the capability into Polyphony is straightforward and streamlines the processing of each image.
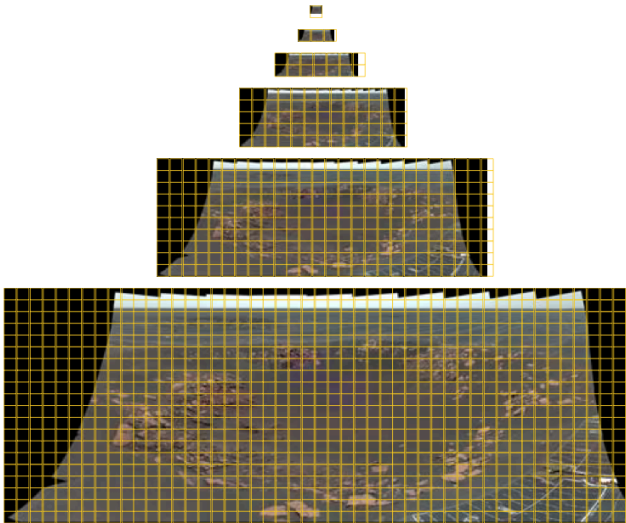

Figure 2. The Yellow Regions Indicate Generated Tiles in an Image Pyramid at Different Resolutions [3].

3A. Queue As a Service

AWS SQS provides a very natural interface to interact with the queue. We were able to integrate Polyphony with SQS as a queue fairly easily, and we were fairly pleased with the performance and robustness it provides. To streamline the operation, SQS provides the ability for a client to request up to ten tasks from a single request. This facilitates a Polyphony client running with a thread pool to handle multiple requests simultaneously.

A few key features are lacking in SQS that we may see in the near future, if not in SQS, then in other competing queues. The biggest missing feature is the lack of bulk PUTs. In our application, we had to make 184,000 individual PUTs even though we knew the tasks a priori. Meanwhile, we find the artificial limit of ten tasks in the bulk GETs are an inconvenience. Similarly, bulk DELETEs would be nice to coalesce the delete requests on a client. The lack of these bulk features is not a showstopper. Appropriate ways of dealing with this is to create larger tasks. For instance, in our case, instead of making a task to handle each image, we could make a task to handle a set of arbitrary number of images.

SQS is designed to have virtually infinite scalability. It supports a large number of clients and can handle any number of messages without a hiccup. However, for small queues like the ones Polyphony has handled so far, we must pose the following question: does the queue really need to be distributed? Aside from the fault tolerance and high availability, a distributed queue offers little more to most applications. A single server, with persistence storage, can easily handle thousands of Polyphony clients, without suffering other side effects of a distributed queue. For instance, SQS' eventual consistency prevents clients from getting accurate estimates of how many messages are in the queue.

When working with a queue, it is tough to assign a task to the client that may be optimally suited to handle the task. The lack of this capability makes it hard to enjoy the move-computation-to-data paradigm offered by Hadoop. A client may have all the data cached required to do the computation or it may be able to receive this data from a neighboring node on the same rack or even the same data center. However, the queuing paradigm effectively trades this functionality for the added simplicity offered by a queue.

3B. Infrastructure As A Service – EC2

Polyphony leverages EC2 resources extensively to improve throughput. We start with a central storage server that exposes EBS based storage to all other Polyphony worker nodes via NFS. After configuring this server and setting up the storage that needs to be exposed, we create a snapshot of this node as an AMI (Amazon Machine Image). The snapshot works as a backup in case something goes wrong with the instance. Nonetheless, we employ instances that are booted with EBS, which allows us to stop the instance when it is not in use to save money and restart the machine when we are ready for computation.

The AMIs are a great tool for creating a snapshot and launching more instances of the snapshot as needed. This came in very handy for the Polyphony client nodes on EC2. We configured an EC2 machine to mount the proper NFS mounts from the NFS server and to start Polyphony java client at boot-time. After testing this instance with several reboots, we used the AWS Management Console to create an AMI of the image; it takes 15 minutes to create the snapshot. After we have the AMI, we can launch N instances of the AMI with two clicks. When we launch 30 clients, each of the clients wakes up, mounts the NFS storage, and starts asking SQS for tasks to complete.

EC2 nodes are also available in the form of Spot instances. Spot instances are a new feature by AWS that enables customers to bid on the excess capacity in AWS's data centers. Spot prices vary based on the demand on the capacity available. AWS customers set the maximum price they are willing to pay for the compute power. At any point, if the spot rate goes above the max price, the Spot instance is terminated without notice. Spot prices are often at less than half of the EC2 prices. In order to effectively utilize Spot instances as a

worker node, one needs an architecture that is designed for failure and is tolerant of an arbitrary number of node failures. Fortunately, Polyphony works even if all the worker nodes fail for extended durations and come back online at a later time. Furthermore, it degrades gracefully in the face of partial failures. Polyphony is well suited for Spot instances and can deliver significant cost savings by leveraging computation at lower prices.

3C. STORAGE AS A SERVICE – EBS, S3, AND NFS

Harmony currently utilizes a single central server, which can potentially become an I/O bottleneck and prevents scalability beyond a certain number of nodes. In this section, we discuss the issues encountered with the bottleneck, how we approached the issue.

Our central server initially exposed an EBS (Elastic Block Storage) volume via NFS on a large EC2 instance. This approach enabled us to tile our benchmark Cassini ISS image set in 11 hours using 20 machines. While performing something that originally took half a month in half a day was nice, we wanted to explore simple optimizations that could streamline our process. We quickly found out that with tens of clients, the IOWait grew as we added more clients. We availed three I/O based optimizations: moving to a larger instance, employing RAID, and configuring NFS parameters on both the server and client.

Moving from a large EC2 instance to a double extra-large instance gave us a much higher throughput to the EBS volume. In order to further improve the throughput on the local disk, we employed 6 EBS volumes and configured them as RAID 0 by using Linux mdadm utility. Lastly, we optimized NFS to support the large number of Polyphony clients that we intend to support. We changed the rsize and wsize (the chunk size of data as exchanged by the client and server) on the clients that mounted the file system. Since we are using NFSv3 and we read the entire image at a time, we experienced the highest performance with rsize and wsize to be 32K. The default rsize is 4K, which requires lots of individual exchanges that fail to fully leverage the available bandwidth over TCP. On the server side, we increased the number of NFS daemons to from 8 to 256. This enabled our server to serve more simultaneous requests. We also increased the memory limit available to the NFS queues on the server side.

With these minor optimizations, we were able to finish processing the same set of images in 5.5 hours instead of 11 hours. We expect this time to reduce as we add more instances, but after a certain number, we will run into a bottleneck with the I/O. We would like to accomplish the same task within tens of minutes instead of hours. In order to accomplish this goal, we would need to employ more machines and a distributed file system.

In our tests, we tried using s3fs, a FUSE based file system that exposes an S3 bucket as a mount on the local machine, but we ran into several limitations. First of all, s3fs does not yet

support S3 buckets located in N. California region. Second, s3fs is designed to read entire files at a time instead of only the parts requested by a read operation. While s3fs would work for tiling small images, it would not work for large images where the responsibilities may be distributed by regions of the image. A block-based FUSE system built on top of S3 would be more suitable for these applications. From S3's perspective, this can be supported through partial GETs. In the near future, we hope to employ a distributed file system like S3 or HDFS to enable us to add an infinite number of Polyphony clients.

4. RESULTS

To test the scalability of Polyphony, we tested it on various AWS EC2 nodes. We first started testing the bandwidth throughput of each type of instance in our environment. As expected, the throughput for a single transfer correlated with the instance price: larger instances had better network performance for single transfers. However, this difference, for single transfers, was not as drastic as we expected. Small instances were able to get roughly 30MB/sec of throughput to our NFS server, while the 2XL instances, which cost an order of magnitude more only experienced 50 MB/sec of throughput for single transfers. That said, these results should be taken with a grain of salt as they were acquired in a virtualized environment with varying loads. Locally, we were able to reliably obtain write speeds of nearly 500MB/seconds.
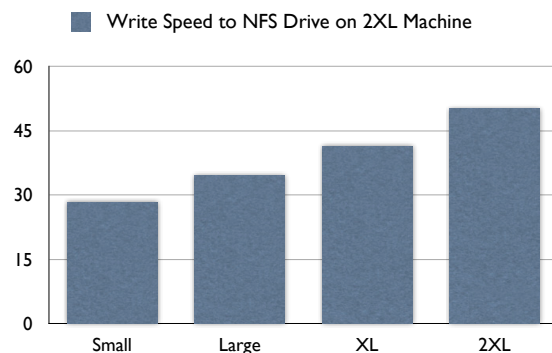

Figure 3. Write Speed over NFS

For the rest of the testing, we used the large instances for consistency. We observed that, for the number of instances we tested, there was not a significant degradation. Although the improvement was not linear, we noticed that adding machines helped us improve our throughput. It is important to note that while the small instances may have better network throughput for the price, they lack CPU capacity that the large instances can provide.
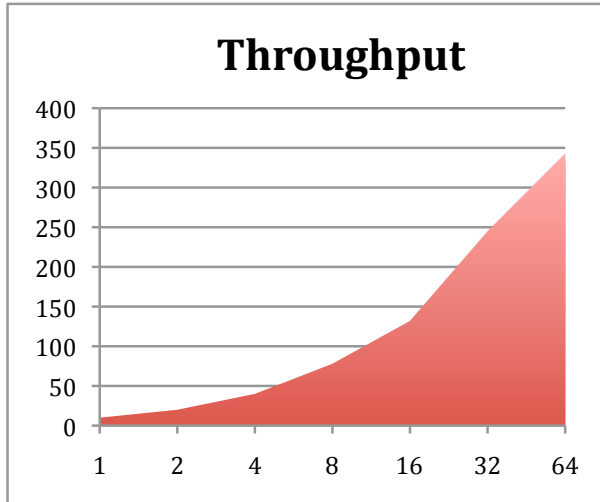
Figure 4. Cassini VIMS Throughput / Instances

5. CONCLUSION

In this paper, we outline the underlying details of Polyphony: a framework designed to handle a variety of parallel tasks for NASA mission operations via distributed computations. Polyphony is more than just an application built around SQS because it provides a modular framework that makes it easy to application developers to add task handlers. Furthermore, we demonstrate that Polyphony can be used to effectively utilize idle machines in our organization. This paper also provides an analysis of the various components of Polyphony and outlines various ideas to optimize the process by improve individual components.

6. REFERENCES

[1] Michael Armbrust et al. Above the Clouds: A Berkeley View of Cloud computing. Technical Report No. UCB/EECS-2009-28, University of California at Berkley, USA, Feb. 10, 2009

[2] OSGi Alliance. (2010 February). *The OSGi Architecture* Available: http://www.osgi.org/About/WhatIsOSGi

[3] Mark Powell, Thomas M. Crockett, Jason M. Fox, Joseph Joswig, Jeffrey S. Norris, Khawaja Shams, Recaredo Jay Torres, "Delivering Images for Mars Rover Science Planning," IEEE Aerospace 2008.

[4] Eclipse Equinox (2010 February). *Equinox.* Available: http://www.eclipse.org/equinox/

[5] AWS (2010 February). Amazon Web Services. Available: http://aws.amazon.com

[6] AWS S3 (2010 February). Amazon Simple Storage Service. Available: http://aws.amazon.com/s3

[7] AWS SQS (2010 February). Amazon Simple Queuing Service. Available: http://aws.amazon.com/sqs/

7. ACKNOWLEDGEMENTS