2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

Interactive Use of Cloud Services: Amazon SQS and S3

Hobin Yoon, Ada Gavrilovska, Karsten Schwan
Center for Experimental Research in Computer Systems
Georgia Institute of Technology
Atlanta, USA
hobinyoon@gatech.edu, {ada, schwan}@cc.gatech.edu

Jim Donahue
Advanced Technology Labs
Adobe Systems
San Jose, USA
jdonahue@adobe.com

***Abstract*—Interactive use of cloud services is of keen interest to science end users, including for storing and accessing shared data sets. This paper evaluates the viability of interactively using two important cloud services offered by Amazon: SQS (Simple Queue Service) and S3 (Simple Storage Service). Specifically, we first measure the send-to-receive message latencies of SQS and then determine and devise rate controls to obtain suitable latencies and latency variations. Second, for S3, when transferring data into the cloud, we determine that increased parallelism in TransferManager can significantly improve upload performance, achieving up to 4 times improvements with careful elimination of upload bottlenecks.**

***Keywords*-Cloud computing, Amazon Web Services, Simple queue services, Simple storage services**

I. Introduction

Public compute clouds are becoming an attractive alternative to private in-house infrastructures for hosting services for a range of businesses. This is driven by factors that include cost effectiveness, availability, scalability, ease of maintenance, and, most importantly, elasticity.

However, to migrate business-critical services, it is important to fully understand the behavior of potential in-the-cloud service deployments, in part to better assess associated cost/benefits. In order to compare the service behavior in the cloud to that of internal private infrastructure, one option is to first fully implement the service given specific public cloud APIs, and to then perform detailed benchmarking and performance studies. A more efficient alternative is to first better understand the performance and behavior of the various cloud-provided service building blocks and their basic operations being used.

This paper provides detailed performance analyses of some of the building blocks and services offered by the popular and widely used Amazon cloud infrastructure. One such building block is a message queuing service. Public cloud providers offer message queuing APIs, like Amazon's Simple Queue Service (SQS), for connecting loosely coupled service components. In the case of Amazon's SQS, this is intended to be a highly reliable and scalable service that enables asynchronous message-based communication between the distributed components of an application [1]. However, there have been repeated concerns regarding the performance of SQS on the Amazon developers forum [2], which have hindered the migration of legacy services to public clouds. To better understand these concerns, we evaluate the SQS send-to-receive message latencies with a latency measurement tool specifically designed for Amazon SQS, and we demonstrate that while the current default performance levels of SQS may not be adequate for certain types of applications, significant improvements can be achieved by incorporating application-level rate-control methods to minimize the possibility of congestion-related bandwidth collapse. A secondary insight from these experiments reveals significant – 3x – performance asymmetry among the zones in the East and West regions. These insights can help developers determine how to best deploy their applications and achieve best behavior from the SQS service.

A widely known issue with using external clouds is the performance of data upload services, which move data from private infrastructures to public cloud sites, to make it easy to share data and more importantly, to leverage cloud elasticity to process it with varying degrees of parallelism – depending on data sizes and application needs. Concrete examples are 'big data' analysis applications, such as those performing time-consuming web analytics. For such applications, while communication among the nodes within the cloud data center are efficient and thus, permit substantial scaling in terms of parallelism in data processing, data uploading can be a severe bottleneck. To address this issue, Amazon provides several upload alternatives, among which the TransferManager library, which utilizes multi-part data upload to S3, is most efficient for large data transfers. We evaluate the behavior of the TransferManager, and show that with minimal modifications to the TransferManager and optimizations of the client's environment, we can achieve up to 6 times performance improvement.

In the remainder of this paper we present the performance analysis of the aforementioned AWS services and describe techniques useful for achieving their desired/improved behavior. Specifically, Section II focuses on the SQS messaging service, and Section III on the data transfer mechanisms for upload into S3. Related work and concluding remarks appear at the end.

978-0-7695-4691-9/12 $26.00 © 2012 IEEE
DOI 10.1109/CCGrid.2012.85

II. SQS Message Latency

We first evaluate Amazon's SQS (Simple Queue Service) – a highly reliable and scalable message queuing service that enables asynchronous message-based communication between the distributed components of larger-scale applications [1]. Cloud developers rely on this messaging service to build application workflows and couple EC2 or application components that rely on other elements of the Amazon Web Services infrastructure (AWS). In spite of claims of SQS reliability and scalability, repeated concerns have been raised on Amazon's developers forum regarding potentially very high send-to-receive message latencies. Toward this end, the performance analysis presented in this paper is aimed at understanding the behavior of the Amazon SQS service and its suitability for interactive web applications.

In order to evaluate SQS, we first develop a simple latency measurement tool. Its initial naive implementation is designed to send, receive, and delete messages as fast as possible, which we term a 'blast' measurement. Figure 1 shows the architecture of this measurement tool. It consists of three threads – sender, receiver, and deleter – in a Java process and two shared data structures among them – message map and delete queue. The sender and receiver continuously make send and receive requests to the SQS server and the deleter makes delete requests when message arrive in the delete queue. The message map keeps track of sent-but-not-received messages, and is used to keep track of the number of messages in the SQS server and to detect duplicated messages delivered to the receiver. The delete queue is a FIFO queue that keeps received messages until they are deleted. The application is deployed in an EC2 instance in the Amazon Cloud infrastructure, thus the network latency to the SQS server is assumed to be minimal, i.e., not exposed to external latency overheads.

Each message generated by the latency-measurement application contains a 37-byte body, which consists of a session identifier, a unique sequence number, current date and time, and send time in milliseconds. The session identifier is an 11-byte string that uniquely identifies an experiment, which is used to guarantee that potentially delayed messages from other (prior) experiments are not delivered to the current one. The send time is used for both measuring the send-to-receive latency and for ordering messages in the delete queue of the rate-controlled experiment described later in this section.

Figure 2 shows the measurement results gathered during one run of the latency measurement application. In this case, during the first 35 minutes, the latencies are mostly less than 1 second, with a 10-second running average of mostly less than 100 ms. From around 35 minutes, latencies start to increase, and after 1 hour and 35 minutes, the application becomes unresponsive. Careful inspection of the system and application logs reveals that the received-but-not-yet-deleted messages start to accumulate in the delete queue after 35 minutes. The behavior's occurrence is consistent even though it is unpredictable when the message accumulation starts occurring in the delete queue and when the application ultimately becomes non-responsive. It also depends on factors such as message sizes, current system loads, etc.

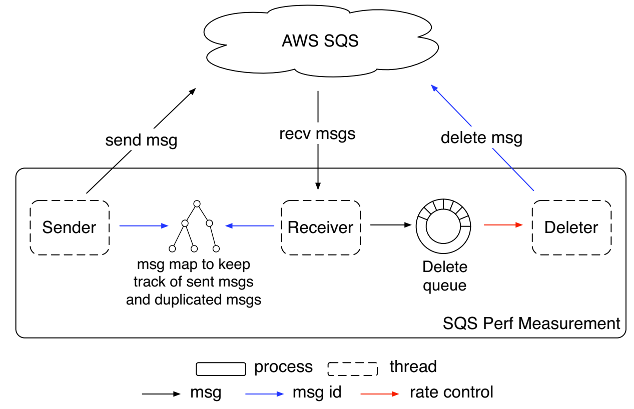

Figure 1. SQS latency measurement application architecture in blast mode.

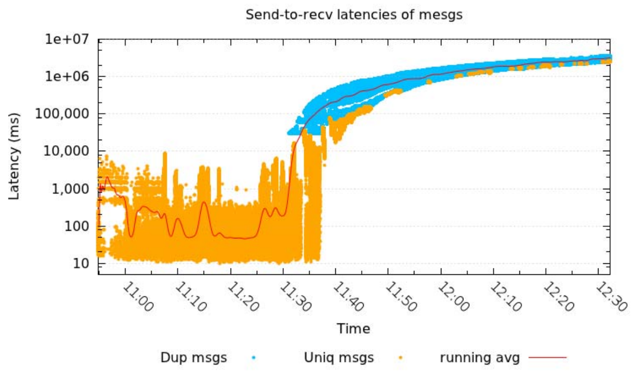

Figure 2. Message latencies in blast mode, measured from an EC2 micro instance. AWS Java SDK version 1.2.9 is used.

The reason for the message accumulation in the delete queue is the SQS built-in reliability mechanism. SQS defines a 30-second default message visibility window, and messages not explicitly deleted in this interval will be re-sent. These duplicate messages in turn cause further congestion and message accumulation in the delete queue, first slowing down message deletion, and ultimately causing full bandwidth collapse. Eventually the measurement application uses up all main memory, causing the Linux kernel to execute out-of-memory killer and select the JVM that the application is running in as a victim.

The above observations illustrate that although SQS provides a useful reliability mechanisms, it does not incorporate any congestion management techniques. In the blast

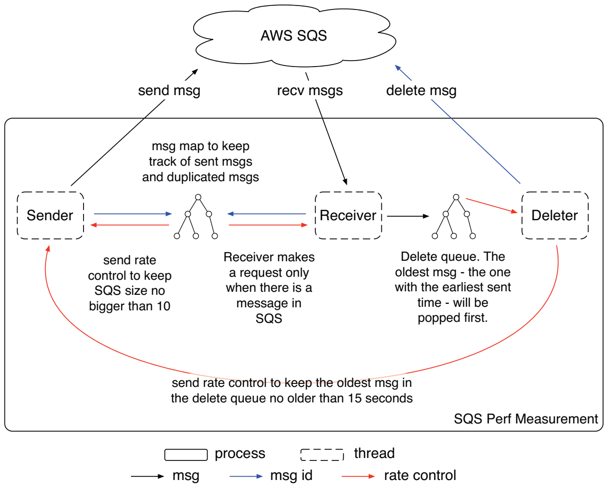
AWS SQS
send msg
recv msgs
delete msg
msg map to keep track of sent msgs and duplicated msgs
Sender
Receiver
Deleter
send rate control to keep SQS size no bigger than 10
Receiver makes a request only when there is a message in SQS
Delete queue. The oldest msg - the one with the earliest sent time - will be popped first.
send rate control to keep the oldest msg in the delete queue no older than 15 seconds
SQS Perf Measurement
process
thread
msg
msg id
rate control

Figure 3. Rate-controlled SQS latency measurement application architecture.

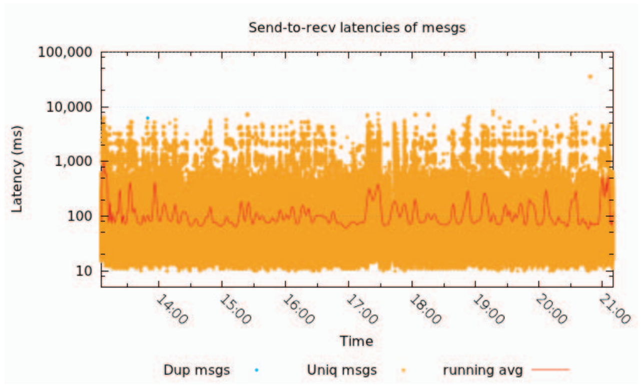

Figure 4. Message latencies from the rate-controlled measurement.

experiment, the overloading of the SQS server is caused by the unbalanced request rates of the sender, receiver and deleter application components. In order to deal with this and to provide better congestion management, we next employ application-level rate controls on each component of the measurement application.

Figure 3 shows the architecture of the rate-controlled measurement application. The specific rate control mechanism used in our measurement application is summarized in the steps listed below.

- Sender sends messages only when there are less than 10 messages in the SQS server. It keeps the number of messages in the server no larger than 10, which is the maximum number of messages imposed by the SQS server that a client can get with one request.
- Receiver makes a request only when there are messages in the server. This prevents unnecessary polling of messages.
- Deleter waits for a message to be deleted in the same manner as in the blast version.
- Sender sends messages only if the oldest message in the delete queue is no older than 15 seconds, which is arbitrarily chosen to be half of the 30-second default message visibility window. This prevents the delete queue from becoming too big, which may cause duplicated messages if messages are not deleted in a timely fashion.

Note that our main goal here is to demonstrate the importance of using a rate control mechanism to improve the behavior of the SQS service, and the actual implementation of such a mechanism will differ for different applications. A potential future extension of the SQS API may incorporate features better suited for implementation of rate-control mechanisms in distributed application, such as querying the buffer space availability or the oldest message in the delete queue.

The measurements gathered with the modified application are shown in Figure 4. The graph illustrates that using rate-control eliminates the flood of duplicate message and the corresponding latency increase. As a result, the latencies become very consistent with the first part of the blast measurement before the occurrence of the congestion collapse problem. During the 8-hour measurement, we measure an average latency of 153 ms, which we consider acceptable for many interactive web applications.

The results in Figure 4 also illustrate that even with rate control, some small number of late messages do appear. Further inspection of the measurement application logs reveals that some of these are duplicate messages – in this specific experiment, only 1 duplicated message is observed. Note that the semantics of the Amazon SQS service guarantee at-least-once message delivery, and do not eliminate duplicate messages. Therefore, applications must be written so as to be prepared to handle them in an idempotent manner.

In addition to latency, we also compare the message throughput for the blast and the rate-controlled scenarios. As shown in Figures 5 and 6, in the blast measurement, the average throughput is initially around 50 messages/second. However, once duplicated messages start to occur, the throughput of unique messages drops significantly. In contrast, in the rate-controlled measurement, we obtain a sustained throughput of 45 messages/second, which is only a 10% decrease from the first part of the blast experiment. We believe that additional refinements of the rate control parameters can result in further improvements in the achievable throughput levels, so as to reduce the observed decrease compared to the blast case.

In addition to demonstrating that the use of rate controls can help applications attain satisfactory message latencies and throughput from the Amazon SQS service, we investigate potential differences in SQS performance across the regions and zones provided by the Amazon Cloud service. To do this, we run the same rate control experiment as above

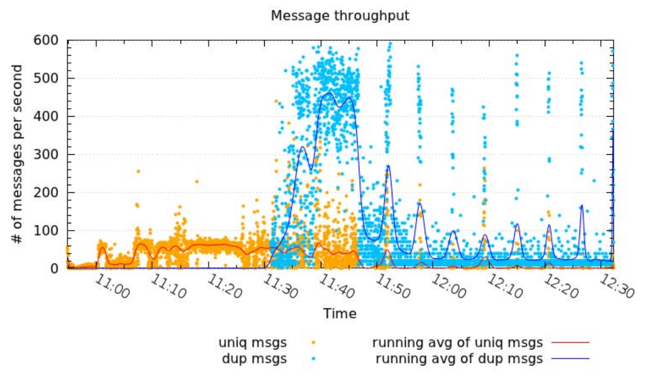

Figure 5. Message throughput in blast mode.

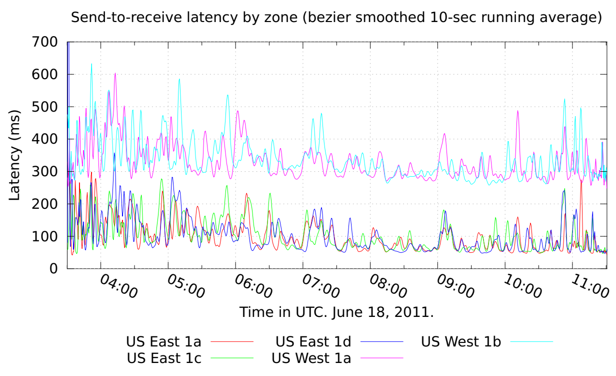

Figure 7. SQS latencies by different zones in the US regions.

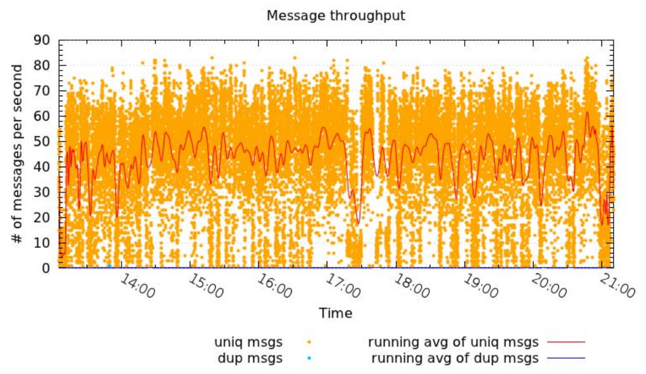

Figure 6. Message throughput in rate-controlled mode.

in each of the five different zones of the two US regions. Our results indicate that there are noticeable performance differences between regions. As shown in Figure 7, the zones in the East region experience latencies averaging about 100 ms, whereas all three zones in the West region observe latencies of about 300 ms. Furthermore, we observe strong correlation in the latency variations among the different zones, and the trends of big vs. small latencies are very similar across zones within the same region. Possible explanations for these observations include either that all SQS servers are in some way synchronizing and thereby affecting everyone's performance in a similar manner, or that all SQS queues are created and managed by a single entity in the East region.

These insights can serve as a hint to application developers when selecting where to deploy an application, unless a specific geographic affinity is required for other reasons. These observations regarding the performance imbalance across zones are consistent with those reported elsewhere [3], [4].

III. Upload Performance Optimization to S3

In addition to understanding the messaging latencies for component interactions, another important aspect of hosting applications in public clouds is to assess the data movement costs to/from the remote infrastructure. For storing data, Amazon provides S3 – a scalable and reliable storage service that can best be used with EC2 (Elastic computing cloud) instances. In Big Data applications, the main bottleneck in using this service can be transfer costs associated with moving large amounts of data from external event source(s) to S3. Toward this end, in the remainder of this paper, our goal is to understand the data movement alternatives and improvement opportunities associated specifically with S3 'upload' performance.

In order to address the needs of diverse customers and their application requirements, Amazon provides a range of data movement services. Two extreme options are the Import/Export service [5] and the Direct Connect service [6]. With the Import/Export service, users can upload data in bulk by FedEx-ing physical storage devices for importing data. In contrast, Direct Connect uses dedicated networks between customer sites and Amazon data centers, thus providing more reliable performance and better real-time characteristics. However, this service comes with additional cost and is provided only in limited areas. This forces most users to rely on data upload via the public Internet.

Among the Internet-based data movement alternatives, we evaluate several popular S3 client applications and programming libraries, with different features in terms of the number of concurrent connections they use to perform a file transfer and the sizes of each part of file used by each connection:

- The Amazon Web Console provides a user-friendly graphical interface, and most users choose this by default. However, it does not seem to support multi-part file uploading.
- s3cmd is a popular command line tool, which is useful for shell scripting.
- Cyberduck is an easy-to-use GUI application, written in Python using the jets3t library.
- AmazonS3Client.putObject() is a low level Java API that uses a single connection per transfer.
- TransferManager.upload() is a high level Java API that

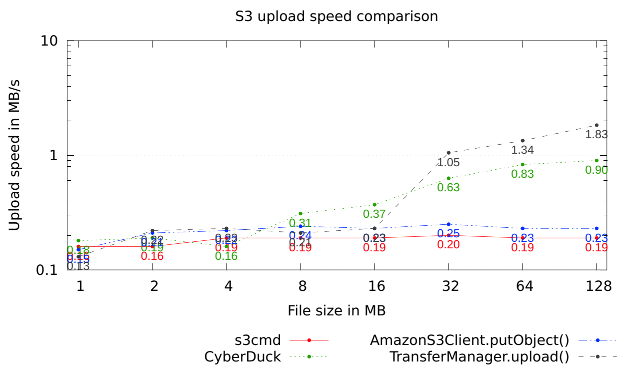

Figure 8. S3 upload performance of various tools. The client machine consists of Intel Core 2 Duo 2.6 GHz, 4 GB of DDR2 memory, and 5400 rpm hard drive

uploads a file in parts using multiple connections and threads.

The results from the comparison of the upload data rates for each of these upload services for different file sizes are shown in Figure 8. From these graphs, we observe that for larger data sizes, TransferManager achieves the highest throughput, followed by Cyberduck. This trend is directly related to the number of concurrent connections used by the upload client: for a 128 MB file, TransferManager uses 10 connections, Cyberduck uses 5 connections, and others use 1 connection. With small files the performance difference is not noticeable, since TransferManager does multi-part upload only when the size of a file is greater than a specific threshold, with the default value 5 MB. We tried to override the behavior by modifying the client code; however, the SQS server limits the size threshold.

Figure 9 shows the simplified execution sequence of TransferManager. The client application creates an instance of TransferManager, requests upload of a file, polls on the upload progress or has a progress listener report on progress, and when the upload is finished, it shuts down the TransferManager instance. TransferManager, upon creation, creates a thread pool with some fixed number of threads and a HTTP connection manager, which has a limit on the maximum number of HTTP connections. Upon an upload request, TransferManager calculates the optimal part sizes for multi-part upload, creates upload requests for each part, and submits the task to the thread pool manager. The thread pool manager runs each task, polls on the completion of all uploads, and requests merging of all parts. At this point, TransferManager throws an exception if it has received a request with part size smaller than 5 MB or file size no bigger than 16 MB. A simple improvement would be to generate the exception when the optimal part sizes are originally calculated.

Given the TransferManager design described in the previous paragraph, potential limiting factors in the achievable

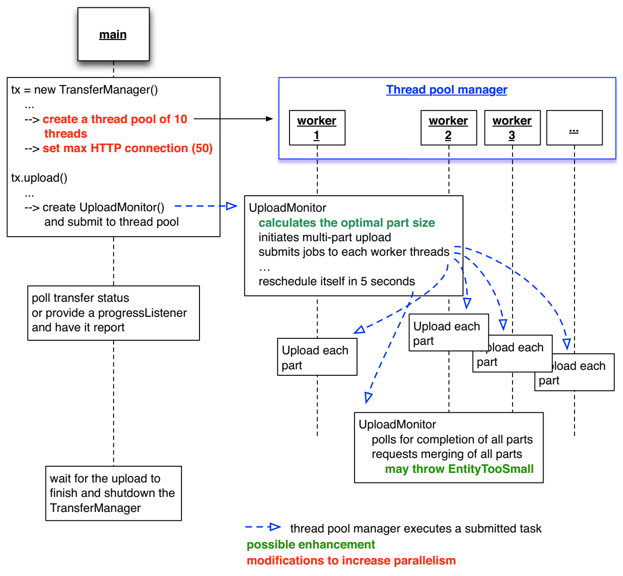
main
tx = new TransferManager()
...
--> create a thread pool of 10 threads
--> set max HTTP connection (50)
tx.upload()
...
--> create UploadMonitor() and submit to thread pool
Thread pool manager
worker 1
worker 2
worker 3
...
UploadMonitor
calculates the optimal part size
initiates multi-part upload
submits jobs to each worker threads
...
reschedule itself in 5 seconds
poll transfer status or provide a progressListener and have it report
Upload each part
Upload each part
Upload each part
Upload each part
UploadMonitor
polls for completion of all parts
requests merging of all parts
may throw EntityTooSmall
wait for the upload to finish and shutdown the TransferManager
thread pool manager executes a submitted task
possible enhancement
modifications to increase parallelism

Figure 9. Execution sequence of S3 TransferManager. AWS Java SDK version 1.2.1 was used.

performance can be due to the fixed number of threads and the limit on the number of HTTP connections it uses. In order to analyze potential improvement opportunities, we modify two of the TransferManager components as follows:

- *Dynamic thread creation*: in TransferManagerUtils.java, we replace the FixedThreadPool that uses 10 threads with CachedThreadPool as an instance of ThreadPoolExecutor. CachedThreadPool creates as many threads as needed and reuse ones when they are available.
- *Increased connection count limit*: in HttpClientFactory.java we increase the maximum number of HTTP connections from 50 to 200; this turns out to be sufficient for our experiments.

Figure 10 shows the throughput increase for 512MB data transfer as a result of increasing the number of threads involved in the transfer. With 10 threads, which is the maximum number of threads of the unmodified code, the average throughput is 5.20 MB/s. Increasing the number of concurrent connections to 14 boosts the throughput level by near 20% to 6.21 MB/s with 14 threads. This result, along with the results from the comparisons of the different upload utilities in Figure 8 above demonstrate opportunities to improve the performance of the upload transfer processes by more flexible selection of parameters which determine the level of concurrency in the transport process, chunk-sizes used by each concurrent connection, or various thresholds.

A second observation from Figure 10 is that in this scenario, when using beyond 14 connection the transfer throughput levels starts decreasing. In order to further understand the opportunities for improvement in the transfer

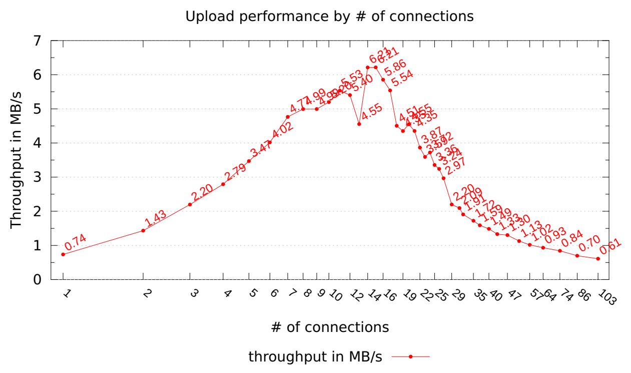

Figure 10. TransferManager upload throughput with more connections and threads.

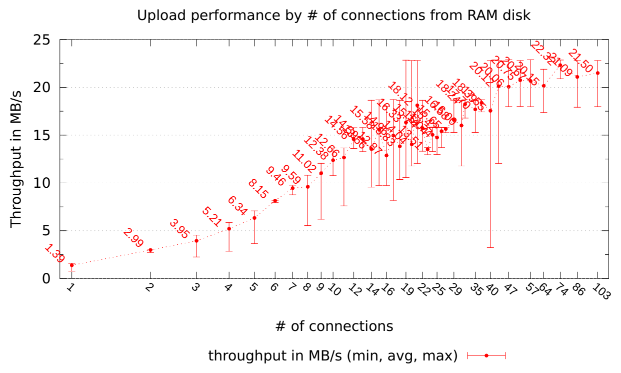

Figure 11. TransferManager upload throughput from RAM disk

module performance we undertake a series of steps to eliminate several performance bottlenecks. First, in order to eliminate the hard disk as a bottleneck, we modify the transfer module to upload data from a RAM disk. As shown in Figure 11, this results in a significant performance boost to up to 22.96 MB/s with 20 concurrent connections, which is 3.7 times than that of the HDD.

Next, we consider the CPU as potential bottleneck. To verify this we track the CPU usage for the different upload module configurations from above, and, as shown in Figure 12, observe that indeed both system and application-level CPU usage does increase and ultimately becomes a bottleneck. We repeat the same experiment on a faster client machine, and as shown in Figure 13, we achieve additional improvements and average throughput of 30.69 MB/s. From the graph and the fact that the CPU usage remains around 50%, we believe that additional throughput increases are possible with larger number of threads.

A potential bottleneck is communications. Since we do not have direct access to the S3 server, we run a network throughput test between the client machine and an EC2 instance. As we increase the number of connections up to 120, we observe that the aggregate throughput reaches 1

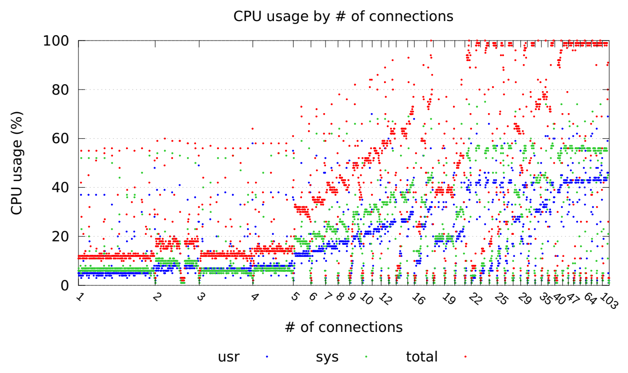

Figure 12. CPU usage of client machine that uploads data from RAM

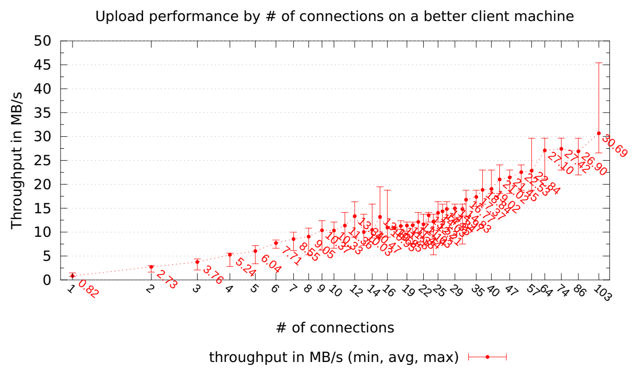

Figure 13. TransferManager upload throughput on a better machine with Intel Xeon Quad Core 2.8 GHz, 12 GB of DDR3 memory, and 7200 rpm hard drive

Gb/s, which is the maximum sustainable data rate supported on the client machine. This indicates that communications via network and/or intermediate nodes are not the bottleneck.

It pays to organize or reorganize files after upload. Specifically, when merging parts of the uploaded file, we observe that this exhibits good scalability. As shown in Figure 14, the time to merge all file parts increases slowly with the increase in the number of parts, and in all cases completes in under 1 second on average, which is under 3% of the total upload time.

In summary, from these experimental results, it is apparent (i) that end users must appropriately select the data transfer services they use, (ii) that it is important to tune such a service's parameters, such as to increase transfer parallelism, or to tune the transfer unit size, and (iii) improved availability of client-side resources (i.e., faster disks and CPUs and more memory), can significantly impact the client's data upload experience and maximize the overall aggregate bandwidth utilization, even without requiring any additional modifications on the cloud-side upload service.

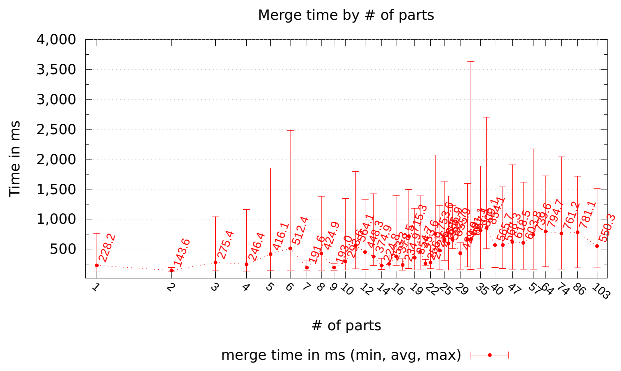

Figure 14. Time spent merging parts

IV. Related Work

The contributions can be grouped into two categories.

First, related to performance measurements of public cloud services, Kagan and Bitcurrent conduct performance measurements of various cloud services, including Amazon S3, but also for Google App Engine, Joyent, Rackspace CloudServer, and Windows Azure [3], [4]. Results show evidence that cloud performance varies significantly across cloud vendors, ISPs, countries, and days and, furthermore, they demonstrate that performance-based load balancing can deliver reasonable performance gains compared to static geo-load balancing. These results also confirm the performance asymmetry among zones in the East vs. West region, also observed in our measurements.

Second, the optimization techniques we use to improve SQS latency and throughput for the upload utility are similar to a range of techniques developed and adapted for transport protocols by the high-performance networking community, such as loss- or delay-based congestion control algorithms, use of parallel connections, such as multiple TCP streams in GridFTP, tuning of transfer unit sizes, TCP window and socket buffer sizes, or packet pacing [7]. We show that techniques such as these can be beneficial even when used at the application level. Better selection TCP- or reliable UDP-based transports which are more suited for large wide-area transfers requires modifications at the AWS-server side, which we have no permissions to perform, but may result in additional performance improvement opportunities.

Specifically, regarding parallel data transfers, Garfinkel and Palankar et al. evaluate the performance of Amazon S3 and show that parallel downloading of files from a bucket can increase the aggregate downloading performance [8], [9]. They focus on the evaluation of download performance from multiple VMs. Our work differs in that we evaluate and optimize multi-part upload performance from a single client, and in addition, more generally explore other techniques for improving the performance of data transfer and other services offered by AWS.

Finally, Amazon recently added batching of send and delete operations [10], which we believe will increase the sustainable throughput rates, but will not eliminate the opportunities for additional benefits from the other techniques discussed in this paper.

V. Conclusions and Future Work

Amazon's highly available, scalable, and fault-tolerant services exist in a purposefully restricted computing environment. We have learned that although client applications do not have any control over these services and how they operate, there are multiple opportunities for client-side optimizations concerning their effective use – with careful rate control of SQS messages and by exploiting parallelism for data upload to S3. There are obvious future extensions to our work, such as those that evaluate other services and devise additional optimization. More importantly, however, this paper demonstrates client-side opportunities for optimizing cloud services, and it presents examples in which performance models maintained by clients can be used to carry out such optimizations. This suggests the utility of a future model-driven approach to cloud service usage and optimization, which we have begun to explore in our recent work.

References

[1] "Getting started with Amazon EC2 and Amazon SQS: Building scalable, reliable Amazon EC2 applications with Amazon SQS," AWS Whitepaper, 2008. [Online]. Available: http://sqs-public-images.s3.amazonaws.com/Building\textunderscoreScalabale\textunderscoreEC2\textunderscoreapplications\textunderscorewith\textunderscoreSQS2.pdf

[2] "AWS SQS Forum." [Online]. Available: https://forums.aws.amazon.com/forum.jspa?forumID=12&start=0

[3] Bitcurrent, "Cloud performance from the end user perspective," 2011. [Online]. Available: http://www.bitcurrent.com/download/cloud-performance-from-the-end-user-perspective

[4] M. Kagan, "Global cloud performance data," 2011. [Online]. Available: http://www.cloudconnectevent.com/2011/presentations/free/76-marty-kagan.pdf

[5] "AWS Import/Export." [Online]. Available: http://aws.amazon.com/importexport

[6] "AWS Direct Connect," 2011. [Online]. Available: http://aws.amazon.com/directconnect

[7] A. Gavrilovska, "High Performance IP-Based Transports," September 2009.

[8] S. L. Garfinkel, "An Evaluation of Amazons Grid Computing Services: EC2, S3, and SQS," Center for, Tech. Rep., 2007.

[9] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: a viable solution?" in *Proceedings of the 2008 international workshop on Data-aware distributed computing*, ser. DADC '08. New York, NY, USA: ACM, 2008, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/1383519.1383526

[10] Amazon, "Release: Amazon simple queue service on 2011-10-20," 2011. [Online]. Available: http://aws.amazon.com/releasenotes/7651101893906539