Cloud Performance Benchmark Series

Amazon Simple Queue Service (SQS)

Md. Borhan Uddin
Bo He
Radu Sion

Originally published: May 2011
Revised: July 2012

Revision Note. Recent interactions with the Amazon SQS team have clarified some of the original results towards a **significant overall improvement**. We have not had the chance to verify these results experimentally, but upon an analytical evaluation believe that they are most likely accurate and representative of the current state of SQS. In the following we introduce the results from Amazon (June 2012). Our original results (May 2011) are included further below.

ver. 0.6 revised

C3 Center of Excellence Wireless and Information Technology
Cloud Computing Center
NSAC Stony Brook Network Security and Applied Cryptography Lab

Amazon Provided Results (July 2012)

We show how, with minimal modifications to the test methodology, one can accurately measure SQS's steady state performance. Using improved methodology:

- SQS shows significantly reduced latency than previously reported. The time to send a 1KB message is reduced from the 400ms in the original measurement to 16ms.
- SQS shows significantly increased throughput. Send throughput increases 400X from the original measurements, reaching a maximum send throughput of 17MB/S.
- SQS returned 0 errors across the 720,000 requests we sent in the improved measurements.

Note that the above improved measurements are not due to changes in SQS as much as improved test client implementation more representative of best practices and better measurement methodology.

Reproducing the original measurements

We obtained the source code used to measure SQS performance from one of the authors Prof. Radu Sion of Stony Brook University. Using the source code and accompanying documentation, along with instructions provided in the original paper, we were able to reproduce similar measurements with only minimal modifications to the source code. As in the original paper, we performed all tests in the loopback setup, using the same EC2 instance type (M1.XLarge) in the same region (us-east-1). We were unable to use the same AMI (ami-e291668b, Fedora 14, 64-bit) as it could not be found at the time of our testing, but we did use an updated version of the same AMI type (ami-1b814f72 is the most recent version of Amazon Linux at the time of this writing). For reproducing the original measurements, we configured the SQS queue as described in the paper, with a Maximum Message Size of 64KB, a Message Retention Period of 4 days, and a Visibility Timeout of 0 seconds.

Flaws in measurement methodology

1. Command Line Client

The original measurements used a command line client, http://timkay.com/aws/, to perform all SendMessage, ReceiveMessage, and DeleteMessage operations. The command line client is unsuitable to perform performance measurements of SQS because:

- It involves 1 completely separate Perl interpreter, and 3 other Unix processes are created as part of making each call to SQS.

- Each time the command line client is used, it performs a "sanity check" call to S3 to verify things are working as expected.
- A new TCP connection is created and an SSL handshake is performed for each request to SQS.

This type of test is not representative of typical SQS application design.

All of these steps combined result in more than 300ms of overhead each time the command line client is used to send an SQS request. This was determined by measuring the minimum time to perform a call using the command line client (which is 400ms) v/s the minimum time to perform a call using the in-process library (< 20ms).

In addition, the reason the original measurements were unable to send messages larger than 9.5KB is because the command line client only uses HTTP GET requests to SQS. SQS supports messages up to 64KB for requests sent using HTTP POST, and all of the AWS SDK clients and open source SQS clients use HTTP POST to send large messages to SQS.

There's nothing wrong with using a command line client to communicate with SQS, but it should not be used for latency sensitive, high throughput applications. The approach used by the paper would be suitable for batch jobs that send a few messages to SQS, but for measuring latency, it is a bad choice because the latency of starting up the command line client for each request dominates the measurement. To get correct measurement of production performance, one needs to remove the "start-up" cost of spinning up thread and establishing connection by starting performance measurements after steady state has been reached.

2. Visibility Timeout of 0 Seconds

The original measurements set the visibility timeout of the queue to 0 seconds. The consequence of setting visibility timeout to 0 seconds is that the first message in a queue will likely be returned multiple times, until DeleteMessage is called for that message. Visibility Timeout is very rarely set to 0 seconds in production usage, and only in situations where you have a single consumer reading from a queue. The default value of SQS visibility timeout is 30 seconds.

For a simple latency and throughput test, setting visibility timeout to 0 should have no effect. What makes it significant for the original measurements is how errors are defined. The original measurements send a fixed number of messages from each producer, and consume a fixed number of messages by each consumer. A unique identifier for each message is recorded by both producer and consumer, and after the test is complete, an analysis is performed to match up sent messages to received

messages. If a message was sent, but cannot be matched up to a received message, that is reported as an error.

With the above definition for errors, it's easy to see why a visibility timeout of 0 seconds will cause errors when testing with multiple concurrent consumers. A single message will be received by multiple consumers, resulting in many messages being left in the queue and never being consumed, therefore being reported as errors.

Suggested Improved measurement methodology

In process SQS Client.

We updated the code to use an in-process SQS Client. I used the perl library Amazon::SQS::Simple which is freely available on CPAN at http://search.cpan.org/dist/Amazon-SQS-Simple/

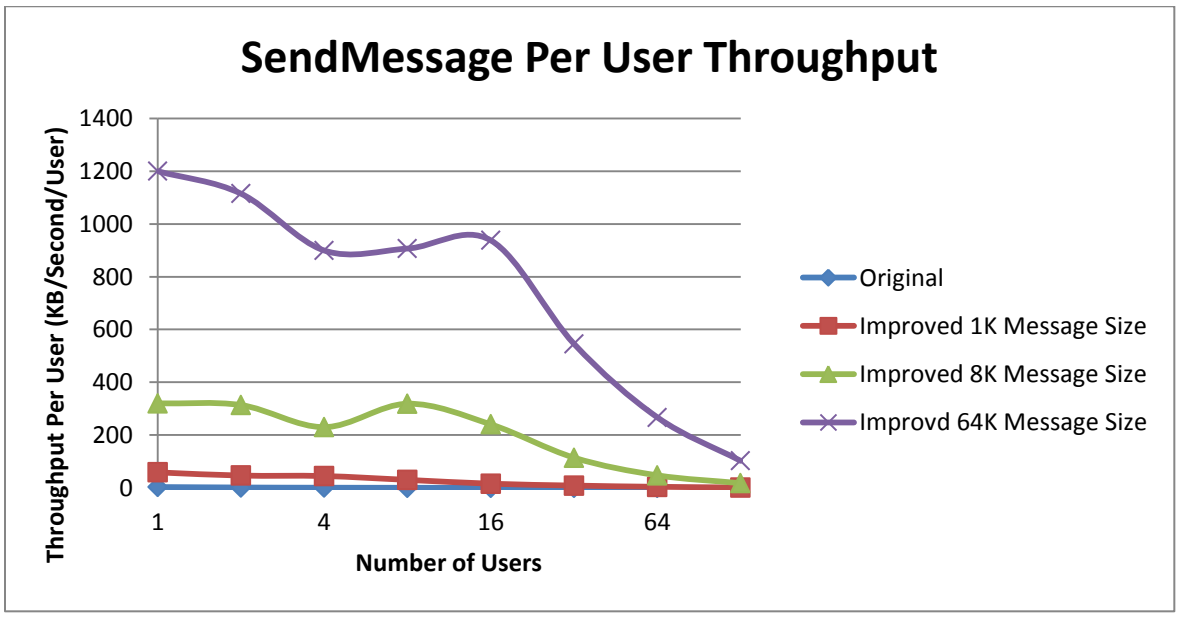
Visibility Timeout of 30 seconds.

We configured the queue to use the default visibility timeout of 30 seconds.

Observed performance with suggested improvements to methodology

1. Throughput Per User

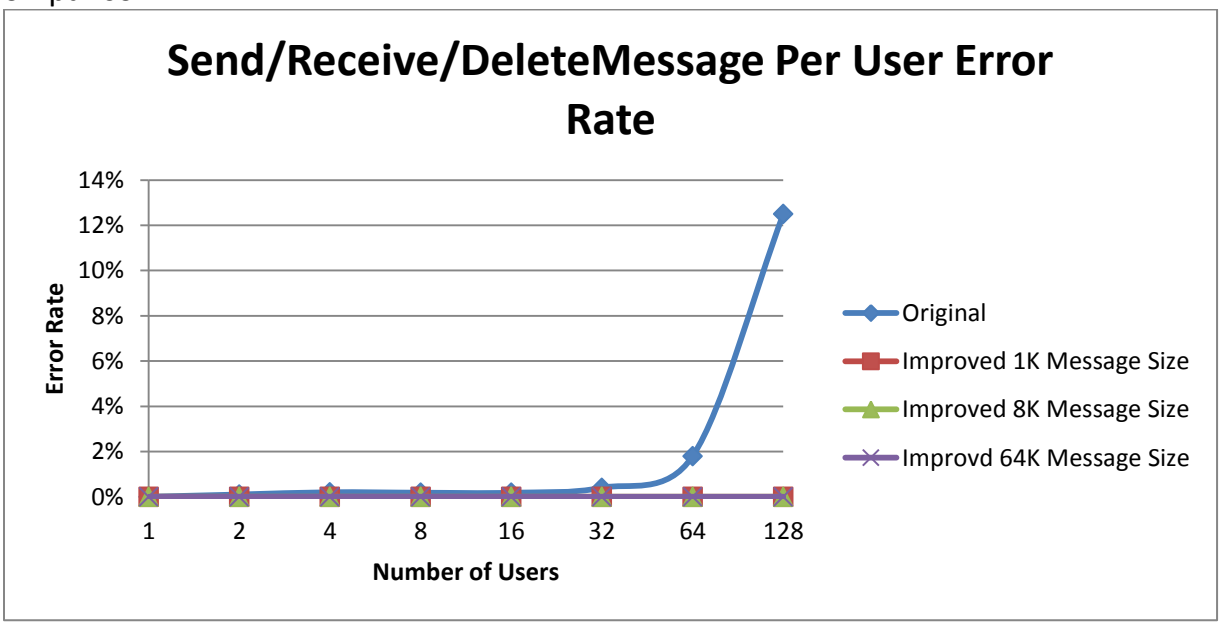
For measuring per user throughput, we performed the same test (so both producer and consumers were running on the same EC2 instance at the same time) and measured the SendMessage throughput per user. We graph it below for 1KB, 8KB, and 64KB message sizes. We have also included the data from the original paper as a comparison.



As you can see, the throughput is significantly improved. There is very little value in having more than 32 users on a single M1.XLarge client instance, as CPU Contention and Network contention prevents the host from achieving increased throughput. The maximum throughput measured is with 32 users sending 64KB messages for a maximum throughput to SQS of 17,444 KB/Second, a 4000 times improvement over the original maximum of 4KB/Second.

2. Error Rate Per User

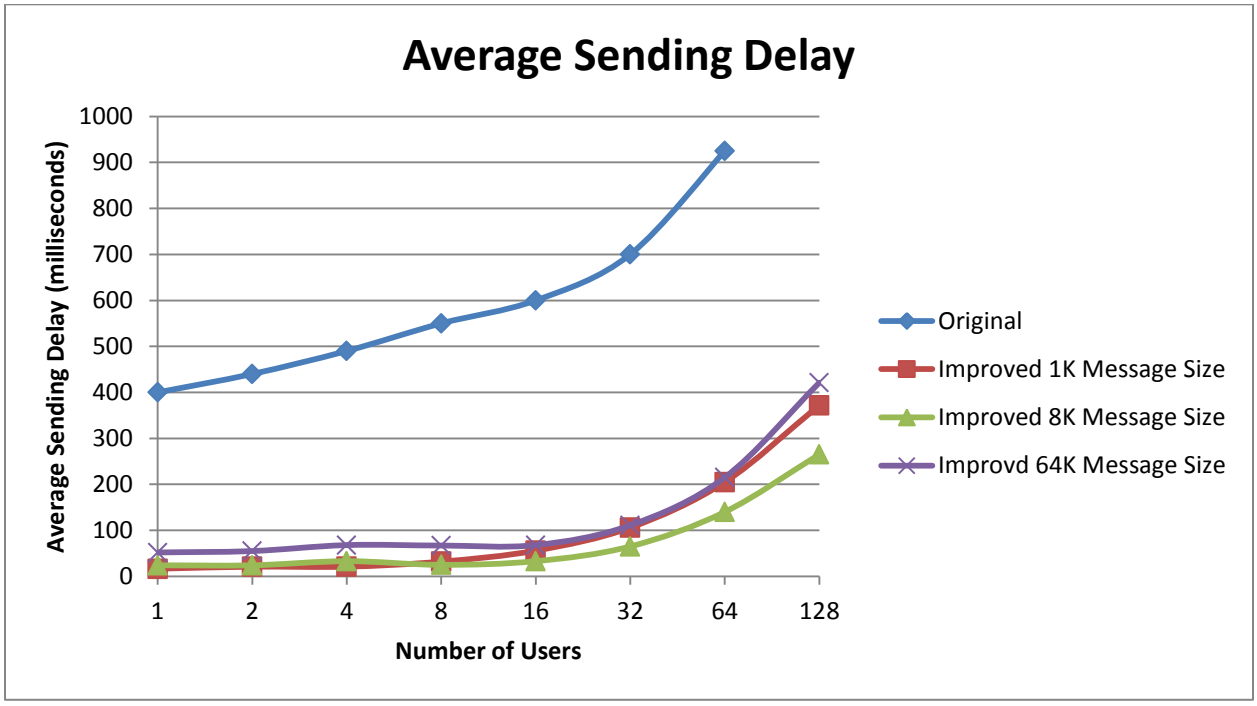
For measuring per user error rate, we performed the same test and measured both errors from SQS as well as the number of messages sent that were never received (the same definition of "Error" as the original paper uses). We graph it below for 1KB, 8KB, and 64KB message sizes. We have also included the data from the original paper as a comparison.



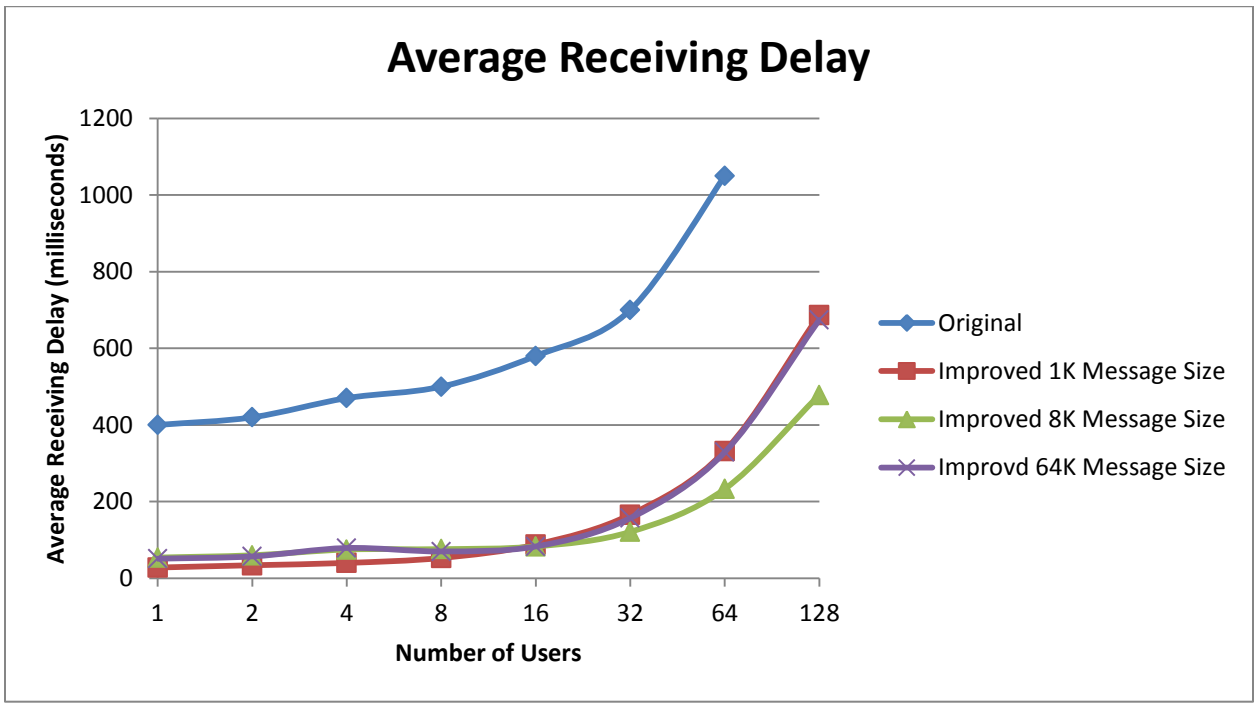
As you can see above, all of the improved requests, regardless of messages size or number of users, succeeded. The error rate was impacted by the fact that visibility timeout was set to 0 in the original test. We observed 0 errors from SQS across all of the improved measurements. For each of our tests (per message size, per number of users) we sent, received, and deleted 10,000 messages, which translates to 10,000 * 8 * 3 * 3 = 720,000 requests to SQS without a single error.

3. Sending and Receiving Delay Per User

For measuring SendMessage and ReceiveMessage/DeleteMessage delay, we took the same data as we've used for the previous experiments and graphed it against the original data.



You can see that the latency to communicate with SQS is significantly decreased by using an in process HTTP client. The reason for the increases in latency as number of users increases is CPU contention, once we past 16 Producers (and concurrent 16 Consumers) the host becomes overloaded and latency increases accordingly.



The reason that Receiving has a higher average delay than Sending is because the way the original code was written, both the ReceiveMessage and DeleteMessage latency is included in a single timer. Therefore, Receiving is measuring 2 calls to SQS while Sending is only measuring 1. There is no evidence to support the claim "Sending and Receiving delays also seem to illustrate a similar phenomenon. With increasing number of users, both delays increase. Due to the SQS's inability to handle high request volumes, individual user experience (specifically throughput) suffers significantly."

Conclusion

SQS shows solidly low latencies while exhibiting linear scalability at the levels we tested, with all bottlenecks observed coming from the client. SQS exhibited no errors during the experiments we performed. SQS seamlessly supports messages up to 64KB without issue, as long as HTTP POST is used.

Original Results (May 2011)

1. Overview

"Amazon Simple Queue Service (Amazon SQS) offers a reliable, highly scalable, hosted queue for storing messages as they travel between computers. By using Amazon SQS, developers can simply move data between distributed components of their applications that perform different tasks, without losing messages or requiring each component to be always available. Amazon SQS makes it easy to build an automated workflow, working in close conjunction with the Amazon Elastic Compute Cloud (Amazon EC2) and the other AWS infrastructure web services.

Amazon SQS works by exposing Amazon's web-scale messaging infrastructure as a web service. Any computer on the Internet can add or read messages without any installed software or special firewall configurations. Components of applications using Amazon SQS can run independently, and do not need to be on the same network, developed with the same technologies, or running at the same time." (amazon.com)

Experiments were performed to benchmark SQS. Two main different aspects were experimented with: (i) number of simultaneously supported users, and (ii) message size and throughput rate characteristics.

2. Setup

SQS can be created in 5 regions – US East (Northern Virginia), US West (Northern

California), EU (Ireland), Asia Pacific (Singapore) and Asia Pacific (Tokyo). The message body can contain up to 64 KB of text in any format (default is 8KB). However, we have noticed in our experiments, that **any message body that contains more than 9.5 KB of text cannot be sent to SQS** even after reconfiguring the maximum message size to be 64 KB. Messages can be retained in queues for up to 14 days (default is 4 days), and can be sent and read simultaneously. We performed tests under two settings: loopback (Figure 1a), and end-to-end (Sender-SQS-Receiver) (Figure 1b).

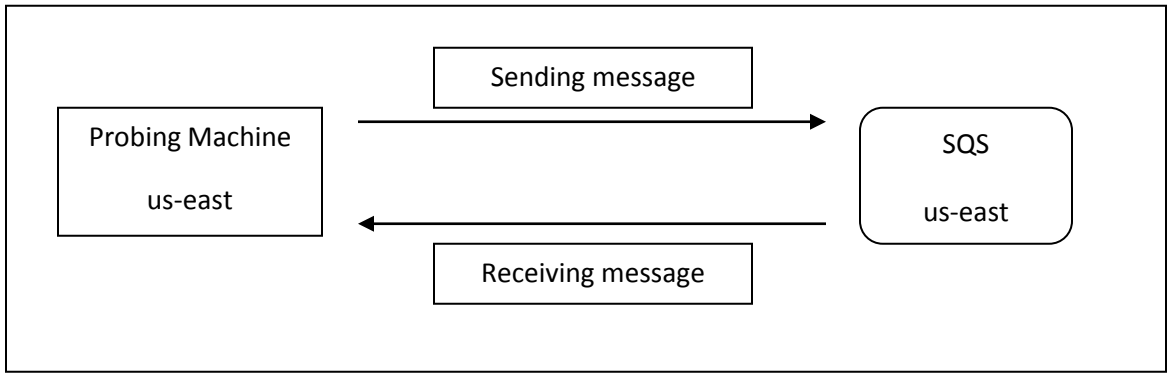


Figure 1a: Experimental setup for loopback setting

In the loopback setup (Figure 1a), SQS was probed from an XLarge EC2 instance with Amazon Web Service (AWS) API tools and PERL scripts installed on it. This instance was constantly sending and receiving the messages simultaneously.

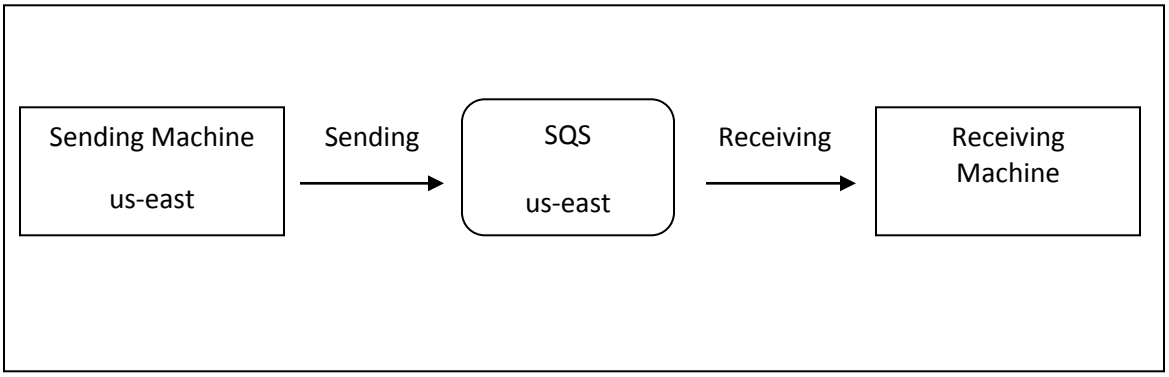


Figure 1b: Experimental setup for end-to-end (Sender-SQS-Receiver) setting

In the end-to-end setup (Figure 1b) two XLarge EC2 instances were used -- one acted as a sender (sending messages to SQS), while the second acted as a receiver (receiving messages from SQS).

No significant difference (throughput, delay etc.) has been observed between these two settings. For conciseness we therefore decided to present here mainly the loopback

setting results but note they apply to both scenarios.

All deployed probing machines' configurations were identical in terms of allocated memory (RAM) in Gigabytes (GB), Platform, and EC2 Compute Units [ECU; 1 ECU aims to approximate an Intel 1.0-1.2 GHz 2007 Opteron or 2007 Xeon CPU] etc. The processing power is structured as a set of multiple virtual cores (VC). A VC in turn is considered as a collection of ECUs (and has an associated number of ECUs/VC). Thus, the total number of ECU compute units can be computed by knowing the number of virtual cores and the number of ECUs per virtual core. Table 1 shows the configurations of the probing machine:

Instance Size	Memory (GB)	Total ECU (=VC* ECU/VC)	Platform	AMI Id
Extra Large	15	8 (=4*2)	Fedora 14 64-bit	ami-e291668b

Table 1: Probing machines' configurations (ECU=EC2 Compute Unit, VC=Virtual Core)

There are some setup parameters (options) in SQS, but the SQS server configuration does not seem to be open to subscribers (i.e., no console/terminal access and no way to customize CPU, IO, memory etc. by users). The following are the queue attributes of SQS (which cannot be altered):

- **VisibilityTimeout -** length of time (seconds) for which a message received from a queue will be invisible to other receivers. An integer from 0 to 43200 (12 hours). If message is not deleted from queue, it will become visible to others after this interval.
- **MaximumMessageSize –** allowed bytes per message. An integer from 1024 to 65536. The default for this attribute is 8192 (8KB).
- **MessageRetentionPeriod –** time (seconds) SQS retains a message. An integer from 3600 (1 hour) to 1209600 (14 days). The default for this attribute is 345600 (4 days).

The configuration details of the SQS are illustrated in Table 2.

Name	Visibility Timeout	Maximum Message Size	Message Retention Period
SQS	0 second	65536 (64 KB)	345600 (4 days)

Table 2: Configurations of Simple Queue Service (SQS)

To test SQS behavior for multiple users sending and receiving messages simultaneously the following setup was deployed: One XLarge probing machine in the us-east region was set up to run from 1 to 128 sender threads simultaneously which pushed messages to an SQS instance, also in the us-east region. Simultaneously, the same machine was running from 1 to 128 polling threads to receive (pull) the messages.

a. Throughput

SQS throughput was measured as follows: One XLarge probing machine in the us-east region, was configured to repeatedly send from 0.5KB to 9.5KB sized plain text messages (in 0.5KB increments) to an SQS instance, also in the us-east region. Average achieved throughput per message size was calculated. Messages (and their corresponding threads) were identified by cross-matching cryptographic hash values at both sender and receiver side.

b. Cost

The dollar cost per message was also computed using the Amazon pricing model at the time of writing (included here for reference). Two types of pricing need to be considered: requests cost and cost of data transfer (in and out). Figure 2a shows an example Data Transfer pricing chart of Amazon SQS. Figure 2b shows an example Requests pricing chart of Amazon SQS.

Data Transfer**

The pricing below is based on data transferred "in" and "out" of Amazon SQS.

Region: US East (Virginia)

	Pricing
Data Transfer IN	
All data transfer in	$0.100 per GB
Data Transfer OUT* **	
First 1 GB / month	$0.000 per GB
Up to 10 TB / month	$0.150 per GB
Next 40 TB / month	$0.110 per GB
Next 100 TB / month	$0.090 per GB
Greater than 150 TB / month	$0.080 per GB

Figure 2a: Pricing (Data Transfer) of Amazon Simple Queue Service (amazon.com)

Requests

- $0.01 per 10,000 Amazon SQS Requests ($0.000001 per Request)

Figure 2b: Pricing (Requests) of Amazon Simple Queue Service (amazon.com)

The cost per message was calculated in US microCents (1 microCent = 10^{-6} cent) as follows:

Cost per message = SQS Message Request message price + SQS Data Transfer price x Amount of Data Transferred per message

3. Results

3.1. Behavior as a function of number of users:

With increasing number of users SQS throughput seems to drop significantly, and the error percentage increases (Figure 3a).

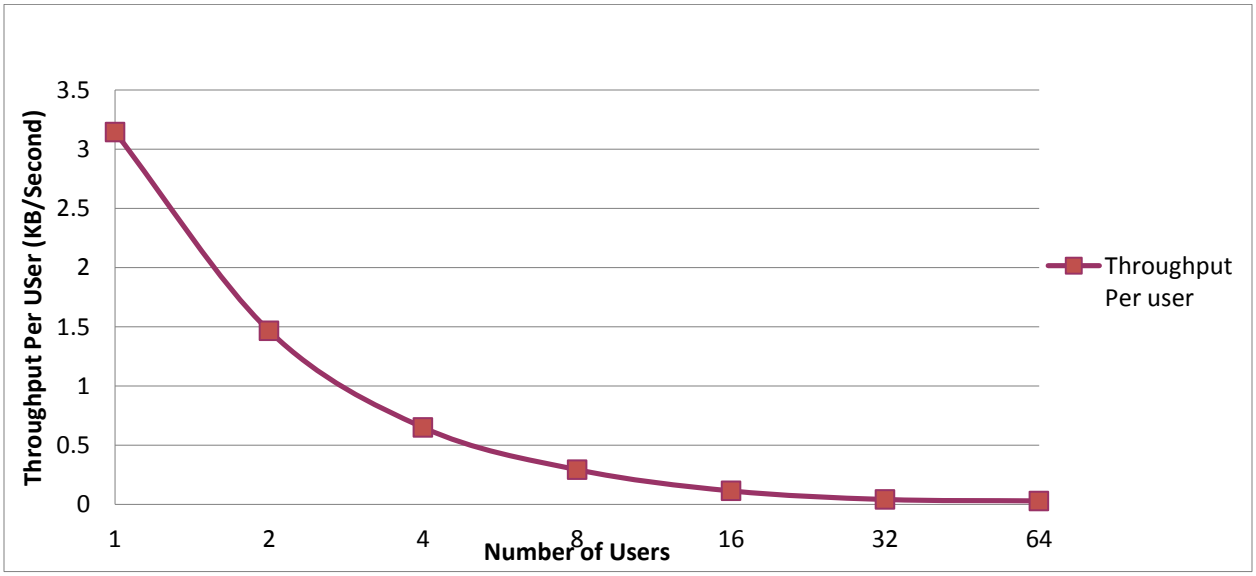


Figure 3a: Observed Throughput per User vs. Number of users in Multi-user Setup

The results for a 128 user setting were dropped due to too many errors (message loss or explicit SQS error message). This is a clear indication that SQS suffers from significant scalability issues, its throughput is low (maximum 3-4 KB/Second at low load) and it doesn't scale with an increasing number of concurrent users.

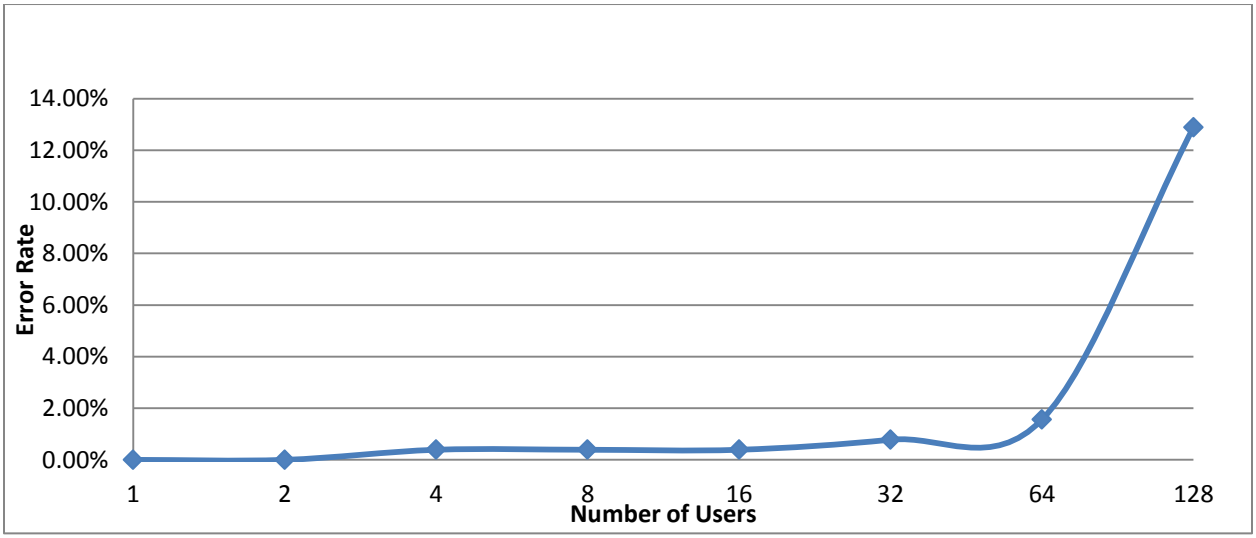


Figure 3b: Error Rate vs. Number of users in multi-user setup

Once we detected such high error rates we believed it may be interesting to understand the error behavior as a function of number of simultaneous users. Up to 256 send requests were run for an increasing number of users (from 1 to 128). Figure 3b depicts the behavior. As can be seen, error rates are relatively low up to 48-64 users. Beyond that, however a significant spike is observed that we could not immediately explain.

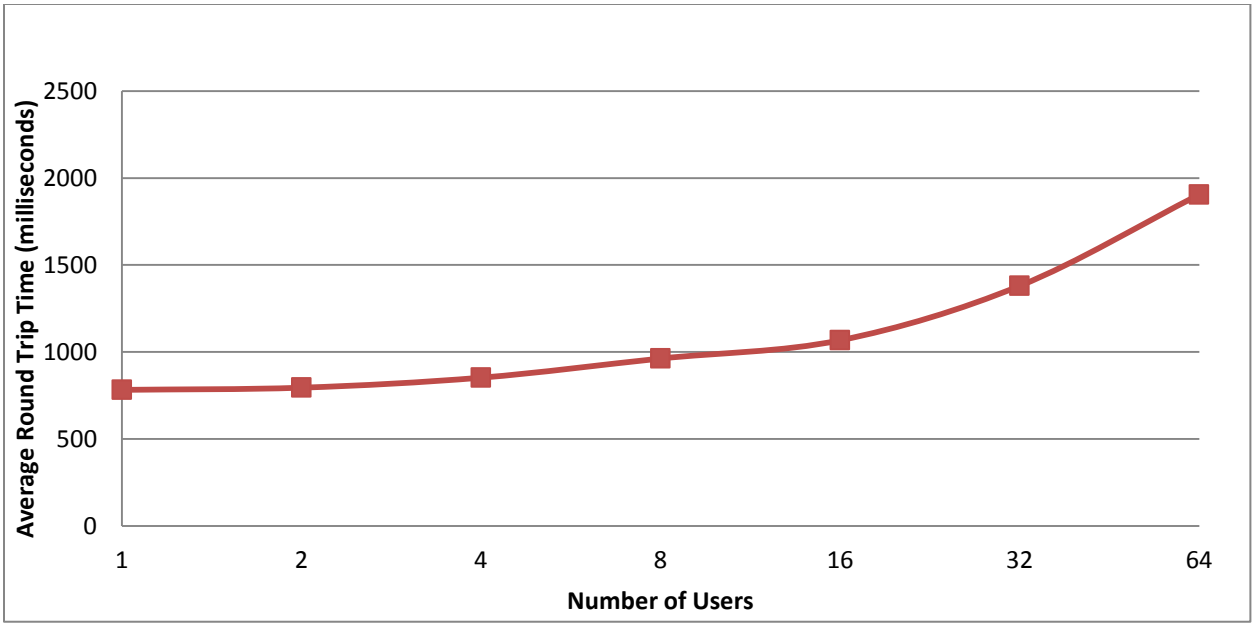


Figure 3c: Average Round Trip Time vs. Number of users in multi-user setup

Figure 3c, depicts the average message Round Trip Time (RTT) as a function of increasing number of users. A positively correlated behavior can be observed. This may be the result of increasing message processing latencies inside SQS.

Sending and receiving delays also seem to illustrate a similar phenomenon. With increasing number of users, both delays increase. Due to the SQS's inability to handle high request volumes, individual user experience (specifically throughput) suffers significantly.

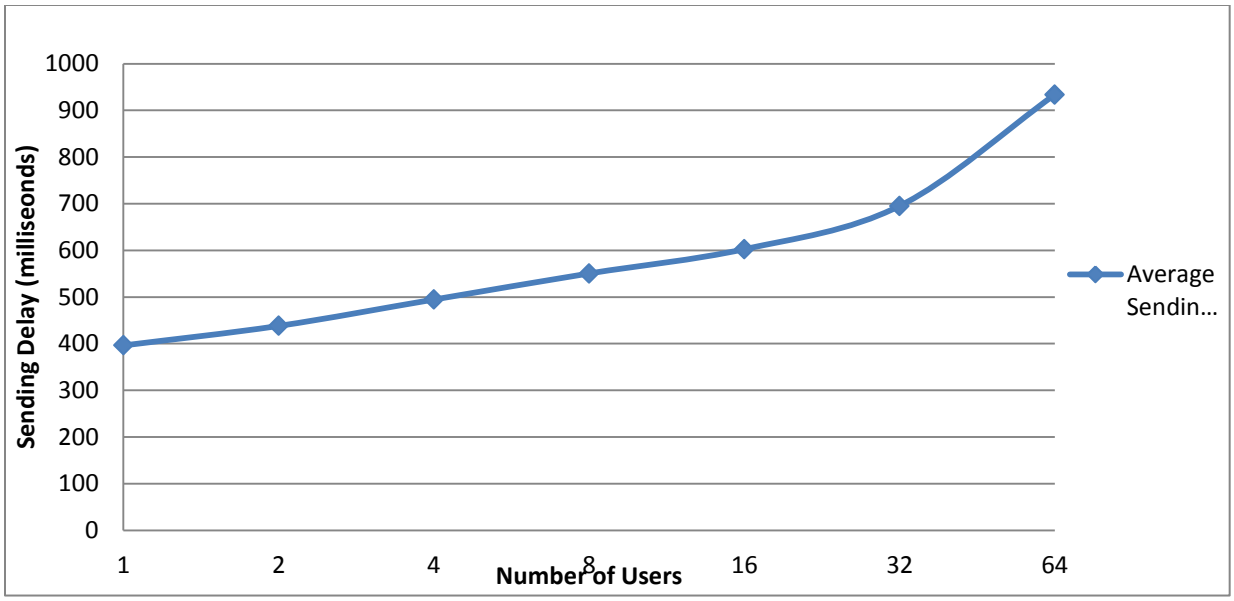


Figure 3d: Average Sending Delay vs. Number of Users in Multiple Users Setup

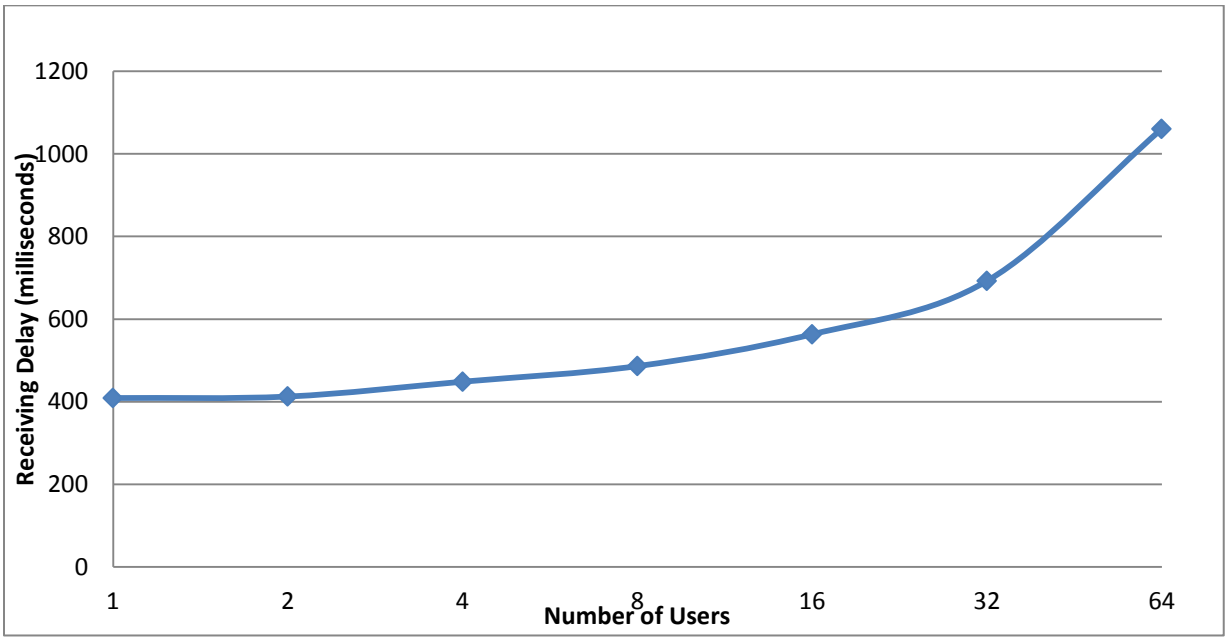


Figure 3e: Average Receiving Delay vs. Number of Users in Multiple Users Setup

3.2. Behavior as a function of Message Size

a. Throughput

With increasing messages sizes (0.5-9.5KB, 0.5KB increments) in a single-user setup, SQS throughput naturally increases (Figure 4a). Overall throughput increases from 1.5

KB/second to 22.5 KB/second as the message size increases from 0.5 KB to 9.5 KB. Messages larger than 9.5KB could not be sent.

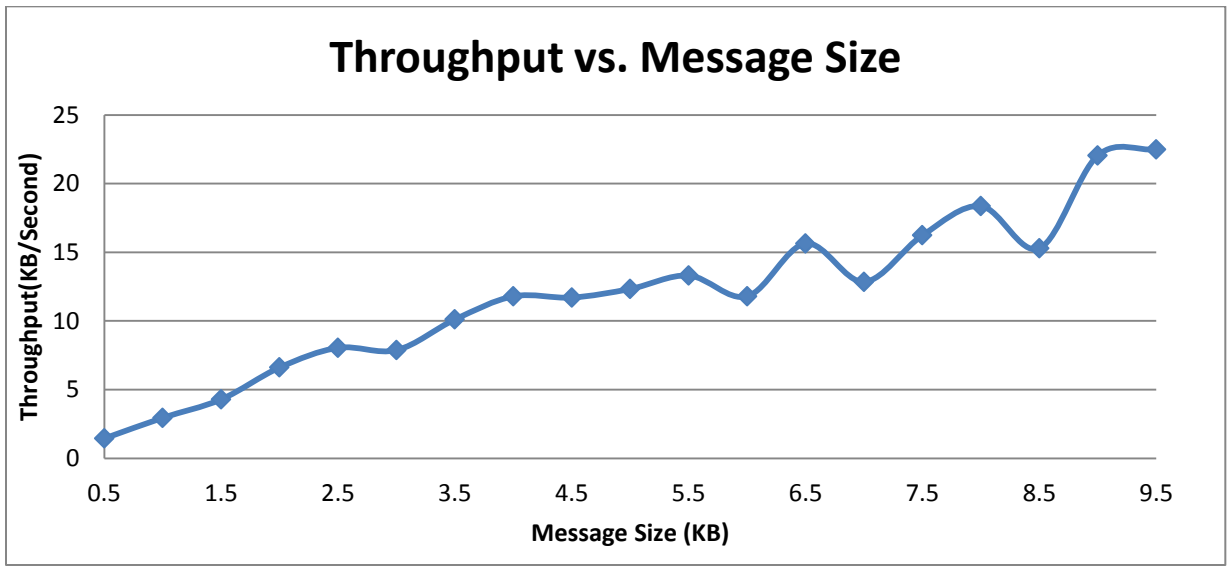


Figure 4a: Throughput vs. Message Size

b. Cost

The individual message cost was also calculated for each message size that we could send. As expected, with increasing message sizes, the cost of sending each individual message also increases. Figure 4b shows the cost per message increases from about 105 µCent to 190 µCent as the message size increases from 0.5 KB to 9.5 KB.

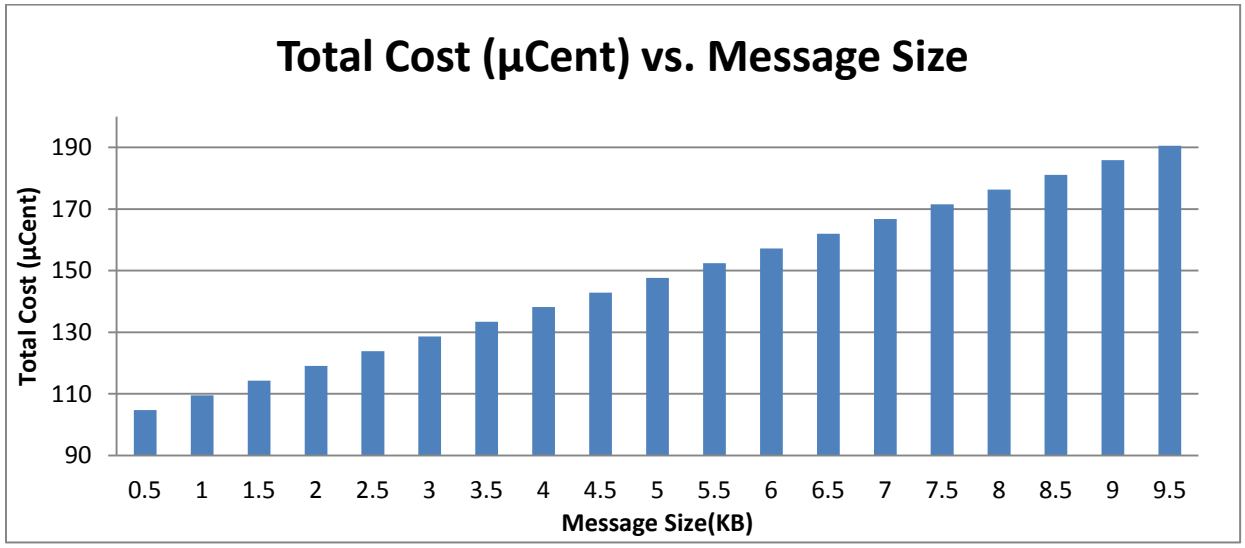


Figure 4b: Cost vs. Message Size in Different Message Size Experiment

4. Conclusion

Unfortunately, the performed experiments only led to one conclusion. SQS featured poor scalability and does not seem to handle enough end-to-end messages to saturate even a small percentage of the available network bandwidth. This may be the result of inefficient message queue service design or unexplained SQS networking or latency bottlenecks.

SQS may be ok for small message sizes and relatively small numbers of concurrent users. For providing a reliable, scalable, and simple message queue service, it incurs a relatively low cost.

However, for applications with a large number of concurrent users and/or messages larger than 10KB, SQS may be (significantly) less than optimal.