



## Comparing Cloud Messaging Platforms

*Executive Summary:* Cloud messaging platforms, or hosted message queue services, offer easy and reliable data exchange. Choose a provider wisely, balancing development methodologies, infrastructure demands, administration and service costs.

### Overview

Cloud-based computing promises to revolutionize distributed applications, capitalizing on the ubiquity of the internet and allowing corporate IT departments to provision a wide variety of information services without having to assume responsibility for every piece of the puzzle. This lowers the cost of entry, simplifies network infrastructure, and frees up internal IT staffers for other tasks. As you might imagine, cloud-based computing brings its own challenges to the table, among the most basic of which is communication between systems.

Consider, for example, an integrated supply chain application. Information needs to flow between the suppliers, manufacturers, distributors, warehouses, shippers, and purchasers of a product, as shown in Fig. 1.

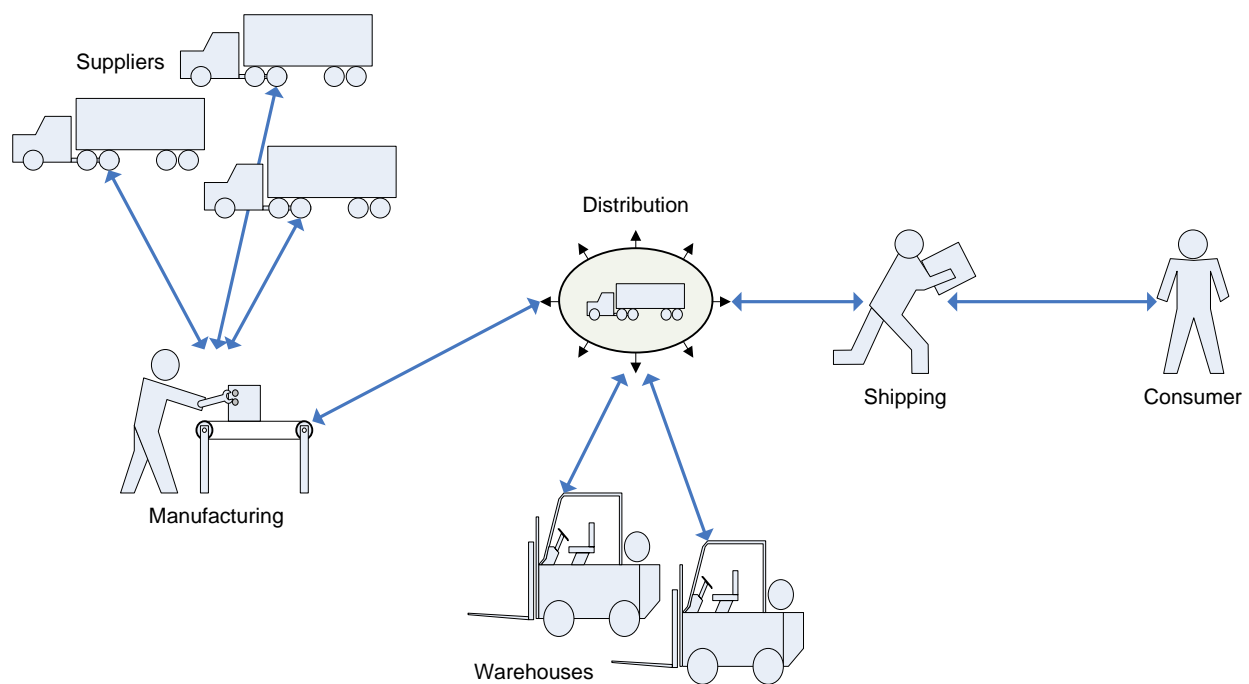


Fig. 1 – Sample Supply Chain with Multiple Integration Points

It's a simple and fundamental concept: connect systems together so they can securely exchange data. But sometimes the implementation isn't so simple. In the past, such solutions have relied on point-to-point connections between various sites, arcane or proprietary interface methods, and custom middleware to make the pieces fit together. This has often been a weak link in corporate systems and a distraction for software developers who are trying to create solid integration between applications, or send messages between distributed instances of a single application. In fact, it's not uncommon to spend as much or more time implementing communications as in creating the business logic of the application itself.

"Cloud messaging" helps alleviate these issues. You can think of it as a secure asynchronous message queue which you can provision as a hosted service. It uses the internet as a transport layer, so getting things connected is easy. The service uses a simple, common interface method, so implementing a message transfer is relatively simple. So now your application architecture looks more like Fig. 2.

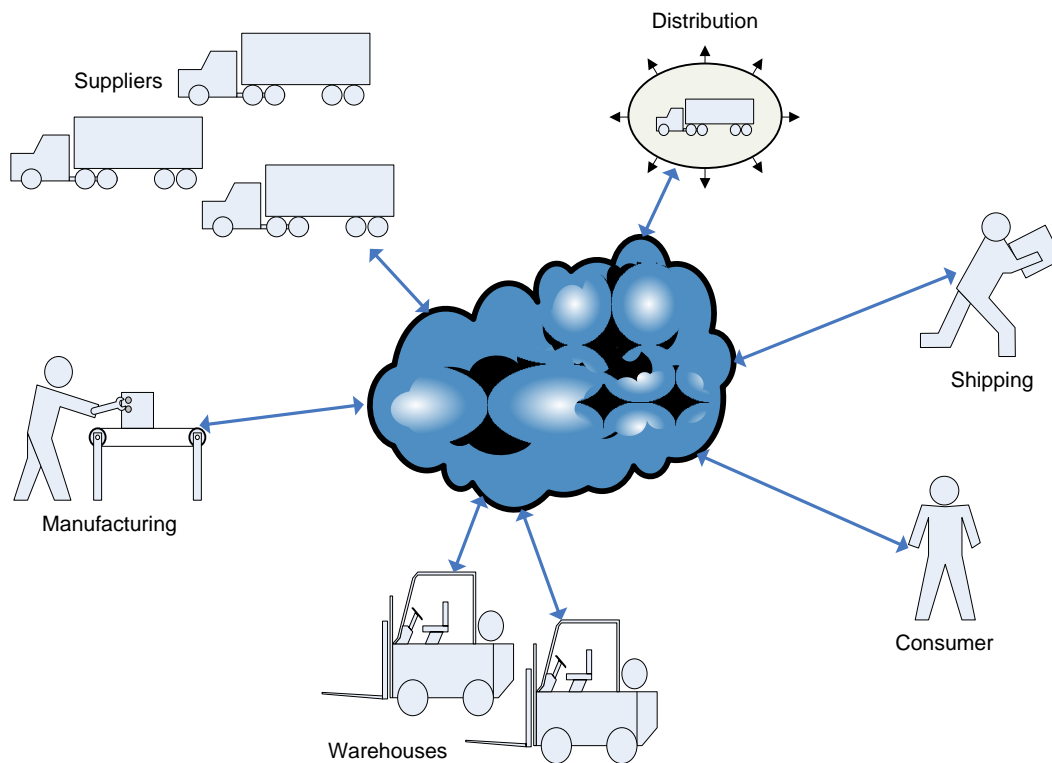


Fig. 2 – Supply Chain Integration Based on Cloud Messaging

There are a growing number of service offerings in the cloud messaging category. Microsoft Azure offers an integrated message queue, as does Amazon Web Services (in the form of their Simple Queue Service). Pure-play messaging providers include Linxter, OnlineMQ, and Gnip. In this whitepaper, we'll examine some of the characteristics of cloud messaging platforms, specifically comparing Amazon SQS and Linxter.

Although their fundamental task is the same, the two queue implementations differ in significant ways. Perhaps most notably, SQS is a REST implementation requiring calls to the service host (using a URI or encapsulated SOAP request), while Linxter is a fully-integrated API with proprietary server architecture. There are also differences in the security model, ease of implementation and resources required, as well as the pricing model and overall costs.

### ***Security***

Linxter uses a series of globally unique identifiers (GUIDs) to help identify and isolate your application's data from the rest of the world. These GUIDs are used to identify the organization owning the data, the software developer, the individual user, the program, and the individual instance of the program. Each message is encrypted at the point of origin, and SSL is used to secure the data transfer channel. Security setup is handled only once<sup>1</sup>, when the application is initially installed. From that point forward, the Linxter API handles encryption of data and exchange of all authentication information needed to transport the message to the server. Amazon SQS uses two access identifiers<sup>2</sup> – roughly analogous to a user name and password. If additional identifying information is required, the developer will either need to implement it in the messaging protocol or use multiple Amazon account instances which would make things somewhat more complicated from a configuration and accounting perspective. However, SQS also allows the message authentication using a X.509 certificate. This is an advantage in environments already using certificate-based authentication or those which require third-party authorization.

Because the client itself is unknown in a SQS implementation, each individual request must not only include the access identifiers, but must also be signed prior to transmitting<sup>3</sup>. This is a resource-intensive task and requires additional steps in formatting and making the web service request. Using the Linxter API, the client is automatically known and encryption and authorization are handled for the developer. Linxter also has exceptional flexibility in controlling communication channels between client programs<sup>4</sup>. This is accomplished using a web-based management tool (Linxter Web Manager) which allows you to add and remove authorized programs and control communications between instances. Every client-to-client connection must first be authorized and you can choose whether such connections are accepted by default, accepted by the target client, or authorized by the administrator. Using Web Manager, you can also control some messaging behaviors centrally<sup>5</sup> (for example, message lifetime), making some application maintenance tasks easier.

### ***Code Complexity***

Overall, sending and receiving messages using Linxter requires minimal application code, and virtually eliminates the need to deal with message transport in any way. In contrast, SQS requires the developer to create classes to handle sending web service requests and parsing their responses. Most people will probably choose a third-party framework for this, or use built-in structures if the development platform provides them. But even having a ready-made web services framework available, the amount of code

required for message control in Amazon SQS is considerably more than that for Linxter. For additional discussion on this, see the section entitled Benchmarking the Platforms. Fig. 3 shows an overview of the basic steps needed to implement both solutions:

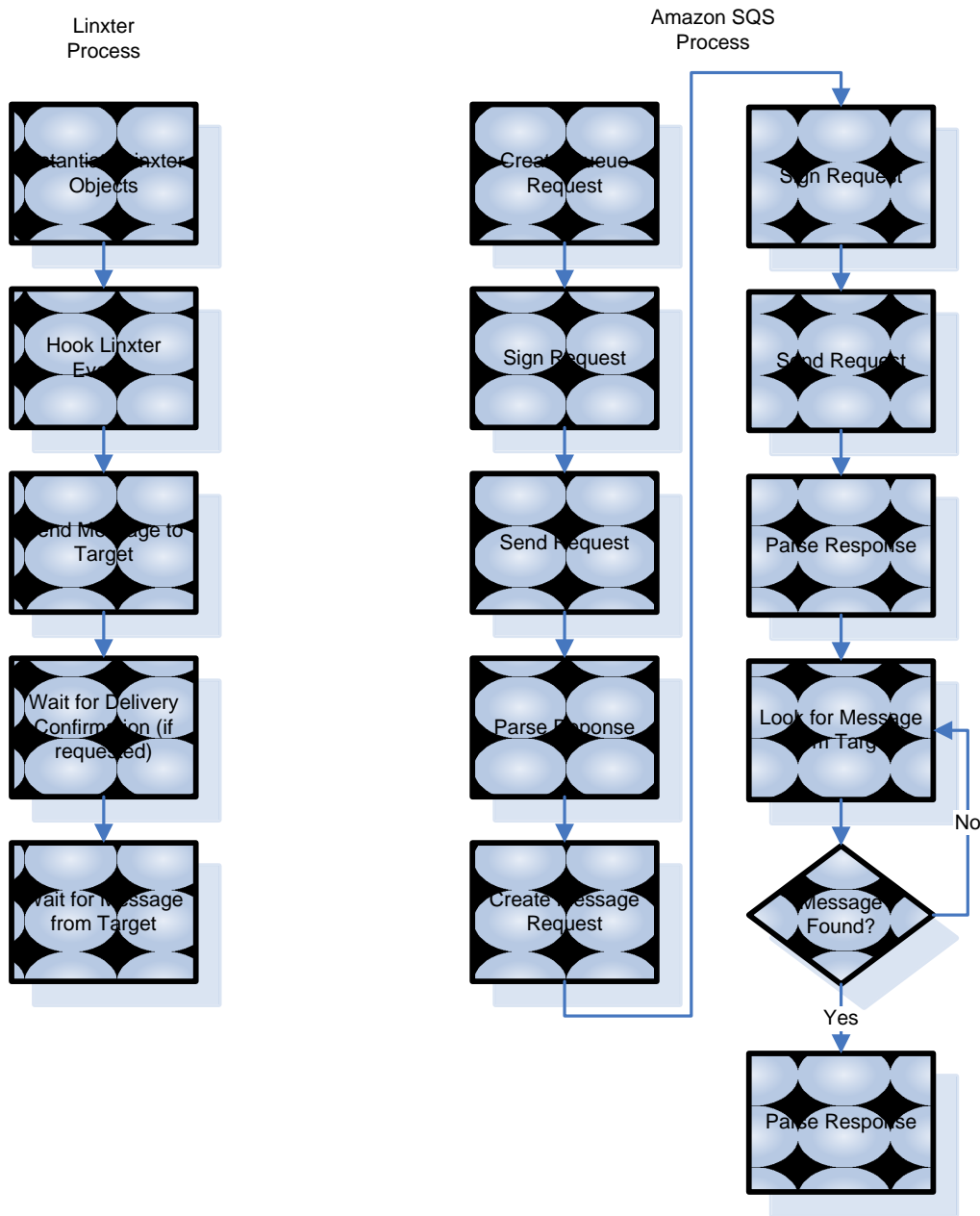


Fig. 3 – Messaging Flow Diagram

In SQS, all messages are sent to a named queue. This is the primary mechanism for specifying the meaning and/or target of a particular message. It also allows you to create an access policy to specify which accounts can communicate with a queue based on their Amazon Web Services (AWS) account ID, the machine IP address, and the message date/time. Each instance in a distributed application could be implemented as a queue within a particular AWS account or as individual accounts with one or more

queues. As discussed above, Linxter uses explicit communication channels between instances to control message routing. Each message sent using Linxter requires one or more recipients and an Activity ID (another GUID) which helps identify the message owner. With SQS, a separate request must be formed and sent for each recipient unless only one queue is being used. However, using only one queue may not be ideal because a program instance would need to poll a queue and parse every message to find any messages intended for it. Because there is no context information for a given message, an application developed using SQS would need to maintain a list of all the accounts and queues it can access, or otherwise implement some way to route and identify messages. Linxter manages this information centrally in Web Manager.

In a Linxter implementation, message handling is between the application and the service is asynchronous and event-driven. When a message arrives or other message event occurs, your application is notified. No polling is required as would be the case in SQS or any other REST-based system. When a message is queued for transfer, the Linxter API simply accepts the message and returns control to the calling program after minimal delay. Since communicating with a web service takes time, a SQS implementation will necessarily have to wait for the web service call to complete before it regains control. This will mean that certain applications (for example, those needing a responsive UI) will need to be multi-threaded, adding to code size and complexity. It's probably advisable to use third-party library which implements web service calls asynchronously for you, such as Microsoft's WSE.

Implementations using SQS are likely to require other complexities as well. Since a REST implementation is inherently stateless, your application will require its own state-handling and error recovery mechanisms. For example, if you need to verify that a message has been successfully sent and put in the queue, or locally queue and re-send a message if a connection is unavailable, you'll need to code them yourself. Linxter has a transactional queue which controls message handling at the client. Messages persist locally until they have been successfully delivered to the service. The API automatically controls error handling and the like, so the developer does not need to incorporate these things into the application.

Linxter is a .NET application, and is currently only available on that platform (although others are in the works). This isn't necessarily a limitation for most corporate applications, but since SQS is a web services-based architecture, it may have an advantage in some environments.

Lastly, Amazon SQS has a relatively small message payload of 8kB per message. Larger messages will require additional processing, such as breaking the message into pieces and serializing and transmitting each piece – another process that will need to be created from scratch. Another approach is to store the message in an alternate location (such as Amazon's Simple Storage Service) and transmitting the location of the message. Not only does this increase complexity, but also cost, as we'll see in a moment. Linxter allows you to specify the maximum message sizes you want your application to have. In addition, it supports file-based message attachments (much like an e-mail), so complex message structures are simple to implement.

### **Cost Structure**

Naturally, once you select a hosted service, build a solution around it and deploy it, it becomes very difficult to change. Therefore, it makes sense to take a close look at the costs involved before making a commitment.

As a hosted service, Linxter uses a simple pricing model based on a flat rate for a maximum amount of usage. At the time of writing, the Linxter Standard level costs the user \$50 for up to 10million messages and/or 50 GB of traffic per month<sup>6</sup>. There are some other constraints as to how many program instances may be run, maximum attachment size, etc. It's important to note that the pricing is based on the individual message, each message is only counted once – when it is sent.

Contrast Amazon's relatively more complex pricing model, wherein the charges are calculated on the number of requests and the total size<sup>7</sup>:

Per 10,000 Requests	Data In	Data Out
\$0.01	\$0.10/GB	\$0.17/GB

Fig. 4 – SQS Base Costs

Comparison becomes a little tricky here, because a Linxter message and an Amazon SQS request are not equivalent. To send a single message in SQS, you must make several requests. A request is logged whenever a queue is added or removed, queues are enumerated, messages are sent or received, messages are deleted, or permissions are changed. At a minimum, every message will require 3 requests: one to send it, one to receive it, and one to delete it (as messages persist in the queue until they are deleted). For situations where a message is received by more than one system, you'll log an additional request. Considering that SQS carries a message size limit of 8kB, larger messages will need to be sent using even more requests, whereas Linxter maintains a flat rate for whatever you're sending. To investigate further, let's take a look at some hard numbers.

### **Benchmarking the Platforms**

Both Linxter and Amazon offer sample code for their platforms. For the purposes of comparison, we created two applications using VB.net. The first, based on the Linxter Performance Tester<sup>8</sup> is a straightforward implementation allowing you to send a predetermined number of small messages. The second, Amazon's C# sample<sup>9</sup>, implements the Microsoft WSE 3.0 SOAP wrapper. We modified both sample apps to remove extraneous code, send and receive the same message payload and track the results. The C# sample, of course, was first converted to VB. During testing, we used a packet capture utility (Wireshark) to measure actual network impact.

In ten iterations, we sent one hundred 10-digit strings. We tracked the time to instantiate message exchange, send all the messages, and then receive all the messages. Refer to Fig. 5 for a comparison of the results.

Platform	Instantiation Time	Time to Send	Messages Sent Per Second	Time to Receive	Messages Received Per Second	Number of Data Packets	Number of Bytes
Linxter	0.8s	5.4s	18.6	15.0s	6.7	578	337kB
SQS	2.2s	52.6s	1.9	106.8s	0.9	4442	2336kB

Fig. 5 – Performance Test Results

In this test, Linxter had a significant edge in performance in each category. Owing to the time to create a SOAP header, encapsulate the message in XML, serialize and send the data, SQS takes much longer to process the same small message. Of course, different web services frameworks will yield different results. A true REST implementation may be slightly faster than SOAP. Notice the significant difference in Receive Time between the two platforms: this reflects the overhead of deleting the message in a separate request. Running the same test without deleting the messages cut SQS's total Receive Time roughly in half, bringing the Receive Time on par with its Send Time. Even so, Linxter receives about 4 times faster than SQS and sends almost 10 times faster. The network activity in particular reflects the difference in the two methodologies: SQS takes nearly 7 times the network bandwidth to send the same message. It should be noted that these numbers are highly dependent on the environment and while the comparison between the two platforms is valid, the actual figures could be significantly different.

As noted previously, it takes several SQS requests to send a single message. To investigate the impact of this on our bill, we ran a separate test, sending the same 10-byte message 1000 times. The chart in Fig. 6 was taken directly from the usage report provided by Amazon:

Service	Operation	UsageType	UsageValue
AWSQueueService	Receive	DataTransfer-Out-Bytes	9990
AWSQueueService	Receive	Requests-RBP	1000
AWSQueueService	List	DataTransfer-Out-Bytes	101
AWSQueueService	List	Requests-RBP	1
AWSQueueService	Send	DataTransfer-In-Bytes	10000
AWSQueueService	DeleteMessage	Requests-RBP	1000
AWSQueueService	Create	Requests-RBP	1
AWSQueueService	Send	Messages-RBP	1000
AWSQueueService	Send	Requests-RBP	1000
<b>Requests:</b>			4002
<b>Bytes:</b>			20091

Fig. 6 – SQS Usage Report

The test required four requests per message sent plus two requests for queue setup. The number of bytes charged totaled about twice the actual size of the message – once for transfer in and once for transfer out, plus queue overhead. There was some discrepancy (in our favor) in the data transferred out which we couldn't explain. Also, the code was written to make three requests per message (send, receive and delete), but the report shows four (Receive/Requests-RBP, DeleteMessage /Requests-RBP,

Send/Messages-RBP and Send/Requests-RBP). We couldn't find any explanation for this in the documentation, and a request for clarification to Amazon's customer service department has gone unanswered.

Although there is a significant difference in the number of trips to the server needed to send the same data, the actual charges incurred were fairly similar in this example. Using this data, we estimated the costs required to attain the same throughput that you would be allocated using Linxter Standard. The comparison is shown in Fig. 7.

	Linxter	SQS
Requests	10,000,000	40,000,000
GB	50	50
<b>Total \$</b>	50	53.5 <sup>10</sup>

Fig. 8 – Linxter vs. SQS cost comparison

This is arguably the simplest comparison possible. Other factors come into play, however. As mentioned previously, since SQS has a message size limit of 8kB, if you're transferring larger messages, you'll need more requests. Multiple recipients of the same message will also log a request.

Architecture of your application will be very important here, as you'll want to minimize creation of any messages larger than 8kB. And you'll want to minimize polling, because each receive operation counts as a request – whether a message was retrieved or not. Amazon's claim that you can make a million requests for a dollar – while not untrue – doesn't equate to moving around a million messages. On the upside, you pay only for what you use.

### Summary

Cloud messaging has certainly made a lot things easier when it comes to connecting systems that need to exchange data. No longer do you need to "roll your own" communication foundation to adapt disparate systems. These hosted message queues differ significantly in implementation, price model and performance. Key learning: the offerings are solid, but make sure you thoroughly evaluate the solutions before implementing.

1. <http://linxterdeveloper.com/registering-programs-isb>
2. <http://docs.amazonwebservices.com/AWSSimpleQueueService/2009-02-01/SQSDeveloperGuide/RequestAuthenticationArticle.html>
3. [http://docs.amazonwebservices.com/AWSSimpleQueueService/2009-02-01/SQSDeveloperGuide/Query\\_QueryAuth.html](http://docs.amazonwebservices.com/AWSSimpleQueueService/2009-02-01/SQSDeveloperGuide/Query_QueryAuth.html)
4. <http://linxterdeveloper.com/communication-channels>
5. <http://linxterdeveloper.com/registering-programs-isb>
6. <http://www.linxter.com/pricing>
7. <http://aws.amazon.com/sqs/#pricing>. The cost/GB for data transferred out of the Amazon system goes down as the amount of data transferred increases. The figure shown is for the first 10Tb.
8. <http://linxterdeveloper.com/sample-apps/performance-tester>
9. <http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1172&categoryID=30>
10. \$40 for 40million requests, \$5 for 50GB transferred in, \$8.50 for 50GB transferred out
11. [http://aws.amazon.com/sqs/faqs/#How\\_much\\_does\\_Amazon\\_SQS\\_cost](http://aws.amazon.com/sqs/faqs/#How_much_does_Amazon_SQS_cost)