A cloud queuing service with strong consistency and high availability

Z. Zhang
Y. Wang
H. Chen
M. Kim
J. M. Xu
H. Lei

Message queues are widely used to connect loosely coupled components to form large-scale, highly distributed, and fault-tolerant applications. A number of vendors currently provide cloud-based queuing services that are designed to be elastic, scalable, and easy to use. However, unlike enterprise queuing systems, which provide strong queuing consistency and are suitable for many traditional enterprise workloads, these cloud-based queuing services offer reduced queuing consistency. They can deliver messages without loss, but they may deliver messages out of order or with duplications. This paper presents SilverDove Queuing Service (SDQS), a scalable cloud-based queuing service with stronger consistency than existing cloud-based queuing services. SDQS delivers messages without losses or duplications and provides in-order message delivery as an option. Built on top of IBM WebSphere® eXtreme Scale, i.e., an elastic in-memory object grid system, SDQS can be easily scaled up and down to accommodate a wide range of workloads. SDQS is able to provide high availability with either the no-order or the in-order message delivery option. We have performed a preliminary evaluation of SDQS with up to 70 nodes on a compute cloud platform, verifying its consistency offerings and providing insights into the tradeoff between performance and consistency.

Introduction

As computing reaches every corner of people's daily lives, including business informatics, personal entertainment, and real-world event sensing, the scale and complexity of software systems grow exponentially. Coming with this trend is the critical challenge to provide reliable, efficient, and flexible mechanisms for large-scale distributed computing components or applications to communicate with one another. One of the most widely adopted interapplication communication patterns is asynchronous message exchanges through message queues.

A message queue decouples the two communicating parties from each other. It provides temporary storage when the destination application is busy or suffering from poor connectivity. It reduces the involvement of application developers with the complexity of handling the communication mechanisms. It also simplifies the development, deployment, and management of complex distributed applications that span multiple heterogeneous operating systems and network protocols. Exemplary applications that can benefit from message queues include workload dispatching/load balancing, MapReduce [1]-like pipelined processing, distributed workflow management, and information aggregation/dissemination, to name just a few. In addition to being used for asynchronous message exchange, message queues may also be used to support synchronous request–response communication patterns that are common in traditional enterprise application integration. A typical usage scenario is the supporting of Simple Object Access Protocol over Java** Message Service.

With the advent of advanced virtualization technology, many enterprises are adopting cloud computing to reduce their capital and operational expenditure. Along with the trend comes the imperative to provide message queuing (MQ) as a common cloud service that can be consumed by

Digital Object Identifier: 10.1147/JRD.2011.2172312

0018-8646/11/$5.00 © 2011 IBM

multiple tenant organizations and applications. In order to fully realize the benefit of cloud computing and to accommodate the rapidly growing and highly dynamic workloads from today's data-intensive applications, a cloud-based MQ service should be highly reliable and highly elastic while providing good performance.

A common approach adopted by many cloud-based queuing service providers focuses on lower cost, elasticity, and availability as the main design objectives while relaxing queuing semantics. Examples include Amazon's Simple Queue Service (SQS) [2] and the Microsoft Windows** Azure Queue [3]. These services trade the strict semantics of a queue for improved availability and scalability. Although they can deliver messages without losses, a substantial number of messages may be delivered out-of-order, and some can even be duplicates. Since much of the queuing semantics [e.g., first-in–first-out (FIFO)] is not provided, these services are arguably more similar to data storage services than queuing services. As such, they are unable to meet the messaging needs of many applications. They are only appropriate for a distributed workload that is idempotent and order insensitive. An often cited example is a media sharing website, where queues are used to connect a web front end (where user-generated content is uploaded) and a data processing back end (where necessary format conversion and filtering are performed before the media data is stored). In this case, maintaining absolute order is not required, and occasional duplicate messages result in only a slight waste of computation resources but do not cause functional errors.

On the other hand, traditional enterprise-class MQ systems, such as IBM WebSphere* MQ and Microsoft MSMQ, and Apache ActiveMQ**, offer excellent system performance and various other desirable features such as exactly once delivery, FIFO message ordering, and distributed transaction support. These messaging middlewares play an important role in applications that cannot tolerate message duplication or require strict FIFO delivery order. Many financial applications fall into this category. To enable these types of applications to be eventually migrated to a cloud environment and thus realize the cloud computing benefits, it is important to provide a highly available and scalable queuing service with strict queuing consistency. We note that with the strict queuing consistency offerings comes the inevitable tradeoffs that one would have to make in other metrics, such as performance or availability. Therefore, it is also highly desirable for a queuing service to provide applications with options for selecting only the necessary queuing consistency level so that the service can maximize the level of other nonfunctional qualities.

Toward this end, this paper presents the design and evaluation of a cloud-based queuing service code-named SilverDove Queuing Service (SDQS). SDQS is built on top of IBM WebSphere eXtreme Scale [4], which is an elastic in-memory data grid system. Although WebSphere eXtreme Scale provides high storage consistency, it is a challenging task to provide high queuing consistency on top of it. SDQS overcomes the challenge by using a queue indexing approach and a visibility-timeout mechanism. With no failures in network transport and client applications, SDQS is able to provide exactly once message delivery, i.e., no loss or duplication, while offering two kinds of message order options to end users, which can be selected when establishing queues in the system. This represents stronger consistency than existing cloud-based offerings, approaching that of a traditional enterprise queuing product. Additionally, SDQS provides high availability and elasticity. We point out that end-to-end strong consistencies under all failure conditions are not currently provided by SDQS. Please refer to the ensuing sections for more detailed discussions.

The main contribution of this paper is twofold. First, it presents a cloud-based scalable queuing service, which to our best knowledge is the first in its kind capable of providing exactly once message delivery in FIFO order. Second, through evaluation, it quantitatively compares SDQS to other representative queuing services and validates its queuing consistency and high availability. We note that SDQS is a research prototype designed to investigate the possibility of providing strong queuing consistency with high availability. It does not represent the official product plan or roadmap of IBM in the area of cloud-based messaging.

The remainder of this paper is organized as follows. Background knowledge and related work are introduced in the next section, which is followed by a description of the system architecture and design. The following section provides the details of the implementation, evaluation methodology, and results. The final section concludes this paper.

Background and related work

In this section, we provide an overview of message queues and examine two classes of MQ offerings—traditional enterprise MQ products and emerging cloud-based MQ services. We also describe a few distributed storage mechanisms that are related to cloud-based MQ.

Message queue and its consistency

Abstractly speaking, a queue is a linear data structure with two ends—a head and a tail. Data items can only be added to the queue (enqueued) at the tail end and removed from the queue (dequeued) at the head end. On a single host, queues are easily implemented and are often provided as part of the standard collections that are packaged in many program languages. It allows two communicating entities to exchange messages with strong queuing consistency: exactly once delivery and FIFO order.

Supporting strong consistency may require sophisticated protocols, when queues are used in distributed environments with multiple programs running on multiple hosts. Otherwise, weak queuing consistency may occur, which manifests in three different ways. First, messages may be lost. Second, messages may be delivered more than once. Third, messages may be delivered out of order.

Roughly speaking, there are three failure scenarios that will potentially cause reduced queuing consistency. First, the transport (network link) between a client and the queue may fail. Second, the clients may fail. Third, the queue itself may fail. There are different ways of handling these failure scenarios. We will see how they are dealt with in the following two classes of queuing offerings.

Enterprise queuing products

A properly implemented message queue provides a means for reliable asynchronous communication. It has become a standard middleware used in distributed computing environments. Many commercial vendors offer enterprise MQ products. Examples include IBM WebSphere MQ [5], Microsoft MQ (MSMQ), and Tibco Enterprise Messaging Service. There are also a number of open-source or free MQ products, such as Apache ActiveMQ, OpenMQ, and RabbitMQ.

These enterprise-class queuing products typically support strong queuing consistency. Clients usually connect to one or more queue managers using a connection-oriented protocol with transaction support to handle client and network failures. Server failures are dealt with by persisting messages on nonvolatile storage. Additionally, queues can be made highly available by using redundant compute and storage hardware, e.g., Redundant Array of Independent Disks and active standby queue managers. For these reasons, enterprise queuing products are widely used in applications that require reliability and performance, e.g., banking and finance, supply chain management, health care, and so on.

Enterprise queuing products also use clustering techniques to provide additional throughput and scalability. A logical queue can be partitioned into multiple segments on a cluster of queue managers, as shown in **Figure 1(a)**. Multiple senders and receivers can connect to any one of the queue managers to access the logical queue, thus achieving high overall throughput.

Cloud-based queuing service with reduced consistency

Information technology service providers have been making use of cloud technologies to deliver low-cost queuing services with high scalability, availability, and elasticity. This is achieved by adopting commonly used principles in cloud environments [6]. First, they use standard server hardware with standard networking and to avoid specialized setups, such as separate storage area networks or expensive high-end systems. Second, they use large pools of virtually uniform hardware for high elasticity. Moreover, automating the management of these large hardware pools further reduces costs. Representative systems in this category include the Amazon SQS [2] and the Microsoft Windows Azure Queue [3]. Instead of trying to provide strong queuing consistency using traditional enterprise queuing products as the building blocks, the design points of these services are high availability and elasticity, at the cost of reduced queuing consistency.

Figure 1(b) shows a commonly used architecture of these cloud-based queuing services. They are built on a cluster of uniform commodity servers with locally attached storage devices to ensure elasticity. Messages are replicated among storage nodes to tolerate server failure and provide high availability. When a client tries to retrieve messages from a queue, a number of available storage nodes are sampled, and the oldest messages are returned.

To enable a wide array of applications to access the service, a client application programming interface (API) is used that is based on Hypertext Transfer Protocol (HTTP). In order to protect messages from being lost as a result of either client or transport failure, these systems adopt a mechanism called *visibility timeout*. When a client tries to receive messages from a queue, the messages are returned but not immediately deleted. They are locked and made invisible to subsequent receive calls during a time window known as visibility timeout. After successfully processing the message, the client needs to issue an explicit delete request before the timeout happens to delete the message. If timeout happens before the delete request, the message reappears in the queue.

Replication and the visibility-timeout-based transport protocol protect the service from message loss under all three failure scenarios described above, but they cannot guarantee no duplication. Thus, the delivery mode is often called *at-least-once*. Finally, the random sampling design inherently cannot provide any order guarantee.

General-purpose distributed storage

Although, from a functional standpoint, queues are used to provide messaging capability, they can also be viewed as a special type of storage. In this regard, there is a large body of related work in the area of distributed storage. Distributed hash tables (DHTs) use structured overlays on large numbers of servers to support fast lookups and membership updates. DHTs, such as CHORD [7], Pastry [8], and CAN [9], are widely used in peer-to-peer environments.

Another category of a related system is distributed in-memory cache that provides a low-latency buffer for persistent storage in multitier web applications. Memcached [10] is a widely used free software in the open-source community. IBM WebSphere eXtreme Scale is a commercial offering that provides a highly scalable fault-tolerant data

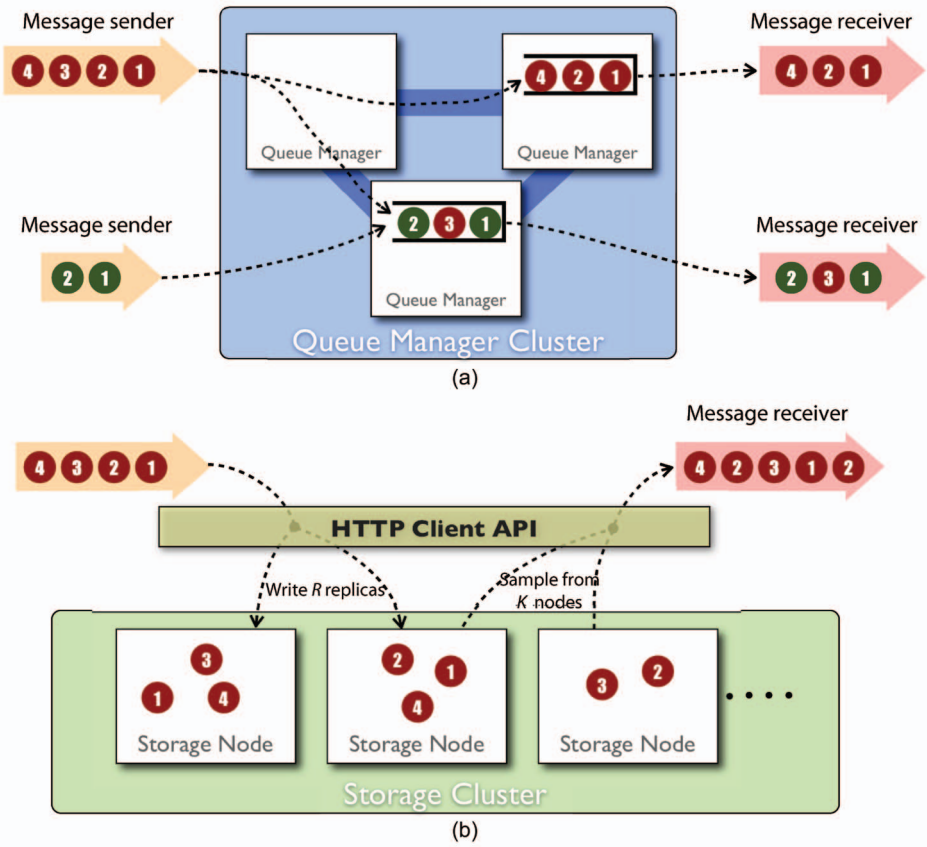

Figure 1

Simplified view of possible system architectures for a cloud queuing service using (a) clustered enterprise queuing system and (b) replicated distributed storage. In (a), which relates to an on-premises deployment of an enterprise queuing system, logical queues can be partitioned among multiple queue managers in a cluster to provide better throughput and scalability. Clients typically connect to one or more known queue managers, and transactional protocols are used to protect against client and transport failures. In (b), which relates to a typical cloud queuing service based on a replicated storage system, message order is not preserved. A message is delivered at least once, i.e., duplication is possible. The service can survive up to $R-1$ simultaneous node failures, where R is equal to the number of replicas.

grid with substantial horizontal scaling capability, which allows the performance of the system to increase by simply adding more servers. While these distributed caches differ from DHTs in how they organize the underlying data store, in general, they all provide a key-value map programming model to clients. Recently, more sophisticated data models have been adopted in a number of distributed storage systems, including BigTable (from Google) [11], HBase (from Yahoo!) [12], Dynamo (from Amazon) [13], and Cassandra (from Facebook) [14].

To obtain better scalability and performance, many distributed storage systems choose to forgo the strong consistency required by the traditional atomicity, consistency, isolation, and durability model. Instead, they adopt a relaxed "basically available, soft state, and eventual consistency" [15] model.

While many of these distributed storage systems offer the desired qualities of a cloud-based service, they do not offer a queue-based API to clients.

System design and architecture

As mentioned, in this paper, we present the SDQS, which is a cloud-based queuing service providing stronger queuing consistencies that approach enterprise queuing products. Acknowledging the fact that higher consistency levels require tradeoffs in other performance metrics, we also include the ability for application clients to select in-order delivery as an option in SDQS. Some earlier information in this paper is repeated in this section as a reminder of the important features.

Design rationale

As we discussed in the previous section, many cloud-based systems, such as the Amazon SQS and the Microsoft Windows Azure Queue, provide highly scalable, available, and elastic queuing services at low cost. Additionally, these systems offer reduced queuing consistencies, in which messages can be delivered more than once and out of FIFO order.

We note that the inconsistencies in existing cloud-based queuing services are caused by two reasons. First, their system design determines that the stored queues are presented as inconsistent to the receiving clients. More specifically, they replicate stored messages without enforcing distributed locks. Therefore, multiple replicas of one logical message can be delivered. They also use queue partitioning and random sampling among multiple partitions of a logical queue, causing out-of-order deliveries, as illustrated in Figure 1(b). Second, they use HTTP REST (representational state transfer) interfaces rather than client-server transactional protocols. This introduces possible duplicated or out-of-order message deliveries when a sending or receiving client fails or is malfunctioning. Additionally, broken connections in the HTTP protocol itself might cause degraded queuing consistency. For instance, after a client sends a message to SDQS, if the HTTP acknowledgement is lost, the client will send the message again, causing message duplication. It should be noted that, with the reliability mechanisms of the underlying TCP/IP, HTTP message losses occur only with major outages (e.g., caused by natural disasters), where multiple redundant network routes are destroyed. The probability of such outages is orders of magnitude lower than the probability of server and client failures.

As mentioned, in designing SDQS, we seek to approach the consistency level of enterprise queuing products such as IBM WebSphere MQ and Microsoft MSMQ, which achieve exactly once and in-order message deliveries. Additionally, we would like SDQS to retain the cost-effective feature and the lightweight client interface of cloud-based queuing services. Therefore, we focus on eliminating the aforementioned server-side sources of queuing inconsistency in cloud-based queuing services. Inconsistency scenarios caused by transport and client failures are orthogonal issues, and we do not include them in the scope of this paper. SDQS can be easily extended with approaches addressing these issues, such as the HTTPR protocol [16], which is a reliable variance of HTTP.

In summary, we design SDQS to achieve strong queuing consistency within the server boundary. The following consistency objectives are achieved: 1) Without HTTP transport or client failures, SDQS ensures that all message will be delivered exactly once, and the in-order option also delivers messages in FIFO order; and 2) when HTTP transport or client failures occur, SDQS ensures all messages sent to the system will be delivered (no message loss).

System architecture

As mentioned in the introduction, we designed SDQS based on IBM WebSphere eXtreme Scale, which is a distributed in-memory object caching system. IBM WebSphere eXtreme Scale uses standard commodity server hardware and networking protocol. Similar with many other distributed data storage systems, it provides high availability by replicating data across multiple servers. Moreover, it differs from most of its peer systems by providing transactional consistency to stored data. In addition to the high availability and consistency from the storage perspective provided by WebSphere eXtreme Scale, SDQS also uses a visibility-timeout approach and a queue indexing method to offer queuing consistency. These terms were discussed in the "Background and related work" section.

The system architecture of SDQS is shown in **Figure 2**. Components are organized in multiple layers in this architecture. The lowest layer is a reliable persistent storage system, which can be based on either a database or an enterprise queuing system. This layer is needed in two infrequent cases: when maintenance or disaster recovery is needed in SDQS or when a huge amount of data needs to be made persistent for a long time. On top of this layer is the in-memory distributed storage layer, which is provided by WebSphere eXtreme Scale. All transactions of queue operations occur in this layer, and data is written to the reliable persistent storage in an asynchronous manner. Based on WebSphere eXtreme Scale, the SDQS Server layer implements a set of queue operation interfaces. These interfaces are independent from the implementation of the SDQS server and the distributed persistence layer; thus, in the future, different SDQS server implementation can be coupled with different distributed persistence storage to offer more consistency options to users. To maximize overall system throughput, the SDQS Server is designed to store all states in the distributed storage layer

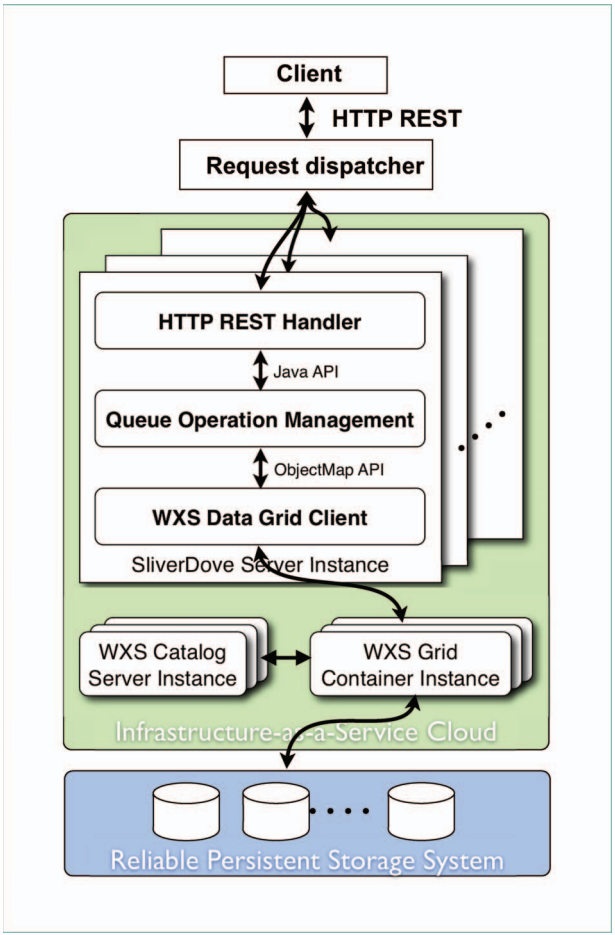

Figure 2

System architecture of SDQS. (WXS: IBM WebSphere eXtreme Scale.)

and maintains no state itself; therefore, multiple instances can be deployed to support concurrent client access. The Queue Operation Management component accesses WebSphere eXtreme Scale via its local ObjectMap API. In the top layer, a RequestDispatcher is responsible for routing incoming client requests to an SDQS server instance. This can be achieved using either a dedicated front-end HTTP router or round-robin domain name system.

Queue operation API

The core queue operation API deals with message access operations, and its design goals are to provide exactly once delivery over potentially unreliable HTTP and to support no-order or in-order delivery options. The following APIs are provided to clients:

- *SendMessage*—This operation places a message on a given queue. The call returns only when the message is persisted in the system.
- *ReceiveMessage*—There are two kinds of semantic interpretations for this operation: if the client adopts the exactly once and no-order consistency option for the queue, this operation tries to randomly retrieve any message available in the queue; if the client adopts the exactly once and in-order consistency option for the queue, this operation tries to retrieve the oldest message in the queue. We use the same visibility-timeout protocol for receiving messages as in Amazon SQS. Each ReceiveMessage request starts a timer, and the message is locked before the timer expires or the message is deleted.
- *DeleteMessage*—As mentioned above, DeleteMessage is always called with ReceiveMessage in pairs to confirm the successful processing of a message by the receiver application.

These message-operation APIs are similar to SQS; readers are encouraged to refer to [2] for more details. To support multitenancy as a cloud-based service, SDQS uses the concept of accounts. The main operations are CreateAccount, ListAccounts, and DeleteAccount. Within an account, a client can manipulate queues using the following operations:

- *CreateQueue*—A new queue is generated with a unique queue ID. The user can specify the consistency level desired to use in the queue.
- *ListQueues*—This lists all the queues that belong to an account.
- *DeleteQueue*—This deletes a queue and all the messages in the queue.

IBM WebSphere eXtreme Scale data model

IBM WebSphere eXtreme Scale stores data in *maps*; a map is an interface that stores data as key-value pairs. There are no duplicate keys in a map. A *key* is a Java object instance that identifies a single value. A *value* is a Java object instance that contains the data.

There are three main resource types in SDQS—account, queue, and message. They are stored as entries in WebSphere eXtreme Scale maps, i.e., each uniquely identified by a key, which takes the form of XYZ-ID, where XYZ is a three-character prefix and ID is created by hashing a human readable unique name. This design distributes data across the entire WebSphere eXtreme Scale cluster and allows direct access to any objects without having to search through multiple levels of indirection.

In WebSphere eXtreme Scale, maps are grouped into map sets [17]. When a map set is deployed to a WebSphere eXtreme Scale cluster, configurable parameters can be specified on the map set depending on the workload and data access pattern; thus, the performance of the map set can be tuned. To this end, the maps in SDQS are grouped into

three types of map sets. The first is the account and queue map set that stores the account and queue metadata. This map set contains the least amount of data among the three. The second is the message map set, which stores the message payloads. The remaining map sets are index map sets, which store IDs and visibility values for messages in FIFO order for in-order queues.

Accounts map

The prefix of the account key is ACT, and the ID is the hash of the account name, which is unique across the system. In the value, a Java HashMap is used to store two categories of data, i.e., Metadata and Queues. The Metadata contains account information such as owner's name, console login password, and secret key. The list of Queues contains reference indices to the queues in the account, i.e., in the form of (QueueName, QueueKey).

Queues map

The prefix of queue key is QUE, and ID is the hash of the fully qualified unique queue name. In the value, a Java HashMap is used to store two categories of data, Metadata and Permissions. Metadata stores information such as default visibility timeout and the consistency option. Permissions stores access control related information, which is not discussed in this paper.

Messages map

These maps are dynamic maps; they are created and destroyed during the run-time, with each corresponding to a queue. The prefix of the map name is the queue ID. The prefix of the map key is MSG, and the ID is the hash of a globally unique ID. The value stored is the message content.

TimeOut map

This map is used to implement the timeout mechanism for the queues. The map key is the same as the key of Messages Map, which identifies the unique ID of a message. In the value, a Java HashMap is used to store two categories of data, i.e., queue ID and message content.

Queue index map

These dynamic maps are used to maintain the order of the messages, which belong to queues with the in-order consistency option. Each map corresponds to a queue, and the prefix of the map name is the queue ID. The map key is the same as the key of Messages Map, which identifies the unique ID of a message. The value stored is a Boolean value that indicates whether the message is visible.

Queue operation algorithm for no-order queues

Given the WebSphere eXtreme Scale data model described above, SDQS realizes the queue operations by manipulating the stored data. The high-level algorithms of the two main message operations (send and receive) for no-order queues are described below.

Send message to a no-order queue

Upon receiving the SendMessage request, the system uses the specified account name and the queue name to compute the queue ID. It checks if the sender has the permission to perform the operation. The Messages Map corresponding to the queue is located using the queue ID. A unique message ID is then generated. The message content is inserted into the Messages Map using the message ID as the key.

Receive message from a no-order queue

Similar to the SendMessage operation, upon receiving the ReceiveMessage request, the system computes the queue ID, checks if the receiver has the permission to perform the operation, and locates the Messages Map. The system retrieves and removes one entry from the Messages Map, whose content will then be returned to the receiver. A timer for this message is started on the server handling the request, and the message itself is inserted into the Timeout Map. Similar to queue and message maps, the Timeout Map is replicated in WebSphere eXtreme Scale to protect against server failures.

If the receiver issues the DeleteMessage request within the visibility-timeout window, the message is located and removed from Timeout Map. The timer is also cancelled. If the timer triggers, the corresponding message is removed from the Timeout Map and reinserted into the Messages Map.

Queue operation algorithm for in-order queues

To preserve order for in-order queues, an order index is built. This index stores message IDs in the order that they are received and also keeps a visibility value for each message. It is stored in an index map, and these maps are grouped into several map sets. When these map sets are deployed, the number of the partitions is set to one. When a message is sent to the queue, the ID of the message is also stored in the index map. When a receive request arrives, SDQS first retrieves an entry from the index map. This should be the message ID of the oldest message of the queue because when the entry is retrieved from the index map, a FIFO ObjectMap API of WebSphere eXtreme Scale is used. The FIFO API supports the FIFO sequence of the map, which is stored in a single partition. Although each index map is stored on a single partition, SDQS still largely retains the benefits of multipartitioning because the workload of storing and processing the actual messages in each queue is still distributed among multiple servers. The following two subsections review the algorithms of send and receive operations for in-order queues.

Send message to an in-order queue

Upon receiving the SendMessage request, the system uses the specified account name and the queue name to compute the queue ID. It checks if the sender has the permission to perform the operation. The Messages Map and the Queue Index Map corresponding to the queue are located using the queue ID. A unique message ID is then generated. The message content is inserted into the Messages Map using the message ID as the key. The message ID is also inserted into the Queue Index Map for the queue, with the value set to "visible," indicating that the message is ready for retrieval.

Receive message from an in-order queue

Similar to the SendMessage operation, upon receiving the ReceiveMessage request, the system computes the queue ID, checks if the receiver has the permission to perform the operation, and locates the Messages Map and the Queue Index Map. A FIFO ObjectMap API is used to retrieve the first (oldest) message ID from the Queue Index Map that is marked "visible." The visibility is changed to "invisible." Using the message ID, the system retrieves the corresponding message content from the Messages Map, which will then be returned to the receiver.

A timer for this message is started. If the receiver issues the DeleteMessage request within the visibility-timeout window, the message is removed from Messages Map, and the message ID is removed from the Queue Index Map. The timer is also cancelled. If the timer triggers, the visibility for the corresponding message ID in the Queue Index Map is changed back to "visible."

Implementation and evaluation

We have implemented a prototype of SDQS and deployed it on an internal (company) research compute cloud platform. We have also performed preliminary experiments using a synthetic workload generated by simulation drivers to benchmark the system queuing consistency, availability, and performance. For future work, we would like to collect workloads from real-world usage scenarios and perform more comprehensive evaluations.

The implementation of SDQS uses IBM WebSphere eXtreme Scale version 7.0.0.3, a highly scalable fault-tolerant data grid with substantial horizontal scaling capability. In other words, WebSphere eXtreme Scale is designed to scale out to thousands of server instances by splitting large amounts of data into manageable chunks and distributing them across the grid. Each of the server instances hosts grid data in a grid container. Communication between WebSphere eXtreme Scale grid containers is designed to occur only for availability management and data replication purposes. WebSphere eXtreme Scale has been proven to run smoothly with more than 1,500 Java VMs with a 2-GB heap participating in a data grid managing almost 2 TB of data. WebSphere eXtreme Scale provides strong storage consistency via its built-in transaction support for all changes made to the cached data.

In the experiment deployment, we run WebSphere eXtreme Scale grid container servers and catalog servers as daemon processes using a 32-bit Java Development Kit 1.6 with a 1.5-GB maximum heap size. The queue operations component is implemented in about 7,200 lines of Java code. It provides a Java interface to account-, queue-, and message-related operations. The HTTP REST interface along with an administrative console is implemented with about 1,000 lines of code consisting of PHP, a web scripting language, and HyperText Markup Language (HTML), as well as 3,000 lines of Java servlet handlers. These two components are hosted on WebSphere sMash [18], a lightweight Web application container. A simple client-side Java wrapper of the HTTP REST interface is also implemented. It is used by the test drivers to access the queuing service.

Evaluation methodology and metrics

A cluster of N VMs is used to host the SDQS. Each VM has two processor cores, 2 GB of RAM, and 36 GB of disk space and is connected to a Gigabit Ethernet network. The number of synchronous replicas of the message map set is set to two, which is typically used in a multidata-center deployment to provide high availability. The test workload is generated by a sender program and a receiver program. The sender program simulates Q sending applications, each of which sends messages to a separate queue using T_{send} threads. Each sending thread sends m messages to the queue with δ_{send} seconds delay between successive send operations. For each sender program, there is a corresponding receiver program, which simulates Q receiving applications, i.e., each of which receives messages from the corresponding queue that the sending application uses. Each receiving application uses T_{recv} threads to receive and process the messages. Each receiving thread reads a message from the queue, processes the message in δ_{recv} seconds, deletes the message, and repeats the process. The total number of messages sent and received during a test run is $M = Q \cdot T_{\text{send}} \cdot m$. A separate cluster of VMs (with the same configuration as above) is used to host the senders and receivers. The senders and receivers are configured to use the HTTP REST interface to access the queuing service. Unless otherwise stated, all experiments are repeated seven times, and we report the average result for each data point.

To better characterize the sending and receiving behavior of the system, the sending and receiving operations are executed during two different phases. First, all senders are launched simultaneously. After they have finished sending the specified messages, all receivers are launched simultaneously. Each experiment is repeated three times

and the average values of the following performance metrics are collected. *Send rate* is the number of *SendMessage* operations that the system services in one second. *Receive rate* is the number of combined *ReceiveMessage* and *DeleteMessage* operation cycles that the system services in one second. Four consistency metrics are also measured: duplication rate, loss rate, and two metrics of message order, to be explained in the next paragraph. During each test run, duplicate messages are detected. If the number of duplicates is M_{dup}, the duplication rate is $\lambda = M_{\text{dup}}/M$. Each message contains a payload of L random bytes with a checksum. The receiver verifies the checksum to determine if a message is corrupted in transit. A message is considered lost if its payload is corrupted or if it is not received at all. The loss rate is $\varepsilon = M_{\text{loss}}/M$.

For each sender–receiver pair of a queue, there exists a deterministic correct order of message delivery sequence. To quantify the degree of out-of-order, we define two metrics for each sender–receiver pair. The first metric, i.e., out-of-order rate, is defined as the ratio of the minimum number of messages that have to be reordered to the total number of messages in the sequence. The second metric, i.e., average displacement, is defined as the average difference of message positions in the received sequence versus those in the correct sequence. Note that the degree of out-of-order for multiple senders and receivers depends on the processing rates of different clients and cannot be clearly defined in the practical use cases that we have observed.

For example, consider a sequence of message {m1, m2, m3, m4, m5}. If the received order is {m2, m1, m3, m4, m5}, the out-of-order rate is 20% since only m1 needs to be reordered, and the average displacement is $(1 + 1 + 0 + 0 + 0)/5 = 0.4$. If the received order is {m2, m3, m4, m5, m1}, the out-of-order rate is still 20%, but the average displacement is $(1 + 1 + 1 + 1 + 4)/5 = 1.6$. We can see that out-of-order rate measures the absolute number of out-of-order messages, whereas average displacement reflects the average reordering delay at the receiving end if order is desired. Together, these two metrics provide a comprehensive representation of the message order. Because the duplication metric is defined and measured separately, duplicates are excluded from these two metrics. This is slightly different from the method used in [19], which combines the orderness measure with duplicate measure.

To compare the queuing consistency of SDQS to that of a typical cloud queuing service with reduced consistency, we have also executed the same test workload using Amazon SQS. Since our goal is to improve the queuing consistency of a cloud-based queuing service so that it approaches that of traditional enterprise queuing products, we have also implemented a bridge that links an Apache ActiveMQ queue manager to the aforementioned HTTP REST interface. A logical queue is partitioned into K segments and stored on K different queue managers. Upon receiving a request for Send or Receive, a dispatcher routes the request to one of the K hosting queue managers to be executed. All messages in queues are made persistent on the hard disk.

Queuing consistency under normal conditions

A series of experiments was performed to evaluate the consistency of different queuing systems and policies, including SQS, which provides at-least-once delivery with no order guarantee; ActiveMQ with different numbers of queue partitions ($K = 1, 3, 50$); and SDQS with no-order and in-order consistency options. In this set of experiments, we use 50 server VMs and 10 sending/receiving VMs to host and operate on 50 queues, and 100 messages are sent to each queue ($N = 50, Q = 50, m = 100$). We consider four different metrics as introduced above: message loss rate ε, message duplication rate λ, out-of-order rate, and average displacement. In this set of experiments, we consider a normal operating condition, where no failure occurs to the servers or clients.

Figure 3 illustrates the results with different numbers of receiving threads per queue ($T_{\text{recv}} = 1, 2, 3$). It can be seen that all systems and policies can deliver messages without losses under the presumed normal condition. Both ActiveMQ and SDQS also deliver each message exactly once (no duplication). With SQS, the message duplication rate increases rapidly with the number of receiving threads because, without a distributed locking mechanism, multiple receiving threads can retrieve different copies of the same message from different replicas.

If queues are not partitioned in ActiveMQ ($K = 1$), all *ReceiveMessage* operations for a queue will be routed to the server hosting the queue, and for each such operation, the oldest message will be returned and no out-of-order delivery will occur. With the in-order consistency option, SDQS also produces no out-of-order message delivery despite the partitioning of message queues. This is achieved via distributed locking inside the WebSphere eXtreme Scale client library and the message queue index mechanism introduced in the system design section. SDQS with the no-order option and ActiveMQ with $K = 50$ has very high out-of-order rates and average displacement values because, for each *ReceiveMessage* operation, a random server is selected from the N servers hosting the queue, and the oldest message on the selected server is returned. When $K = 3$ is used in ActiveMQ, the out-of-order rate and average displacement are much lower. Intuitively, there is a one-third probability that the oldest message on the selected server is the oldest message in the queue.

It can also be observed that the out-of-order rate and average displacement decrease with T_{recv} because out-of-order metrics are calculated within each single receiving thread. An out-of-order delivery occurs when

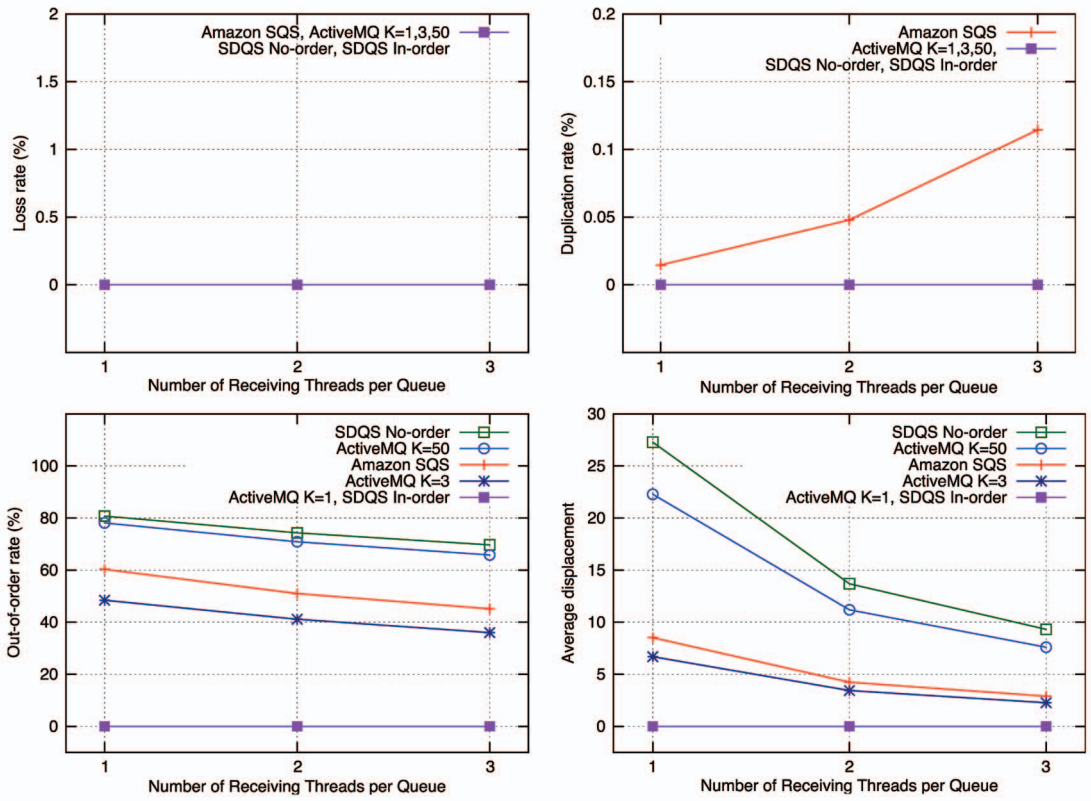

Figure 3

Comparison of queuing consistency (message loss rate, message duplication rate, message out-of-order rate, and average displacement) under normal operating conditions among Amazon SQS, ActiveMQ, and the proposed SDQS with either in-order and no-order options.

two *ReceiveMessage* operations from receiving thread R are routed to two different servers (server_1 and server_2), and the oldest message in server_2 is older than the oldest message in server_1. With more peer receiving threads, between the time that R visits server_1 and server_2, some oldest messages are retrieved from server_2, and the probability of the above out-of-order scenario is lower.

The out-of-order rate and average displacement of SQS largely depends on the sampling mechanism to serve *ReceiveMessage* or, more specifically, the proportion of sampled servers to the entire set of servers hosting the queue.

Tradeoff between performance and consistency

We also investigate the tradeoff between queuing consistency and system performance. We vary the cluster size N from 10 to 70. To emulate a realistic deployment scenario, we let the workload (number of queues) grow proportionally with the cluster size ($Q = N$). We fix the number of sending and receiving threads to be $T_{\text{send}} = T_{\text{recv}} = 3$.

The top two charts in **Figure 4** illustrate the rates of *SendMessage* and *ReceiveMessage* + *DeleteMessage* operations. It can be seen that both in-order and no-order consistency options scale approximately linearly with the cluster size. Overall, the no-order option has higher performance than the in-order option. As discussed in the system design and architecture section, additional operations need to be performed to deliver messages in FIFO order.

The bottom two charts in Figure 4 show the average response time for *SendMessage* and *ReceiveMessage* requests. In general, the response time does not change much with different cluster sizes because the workload

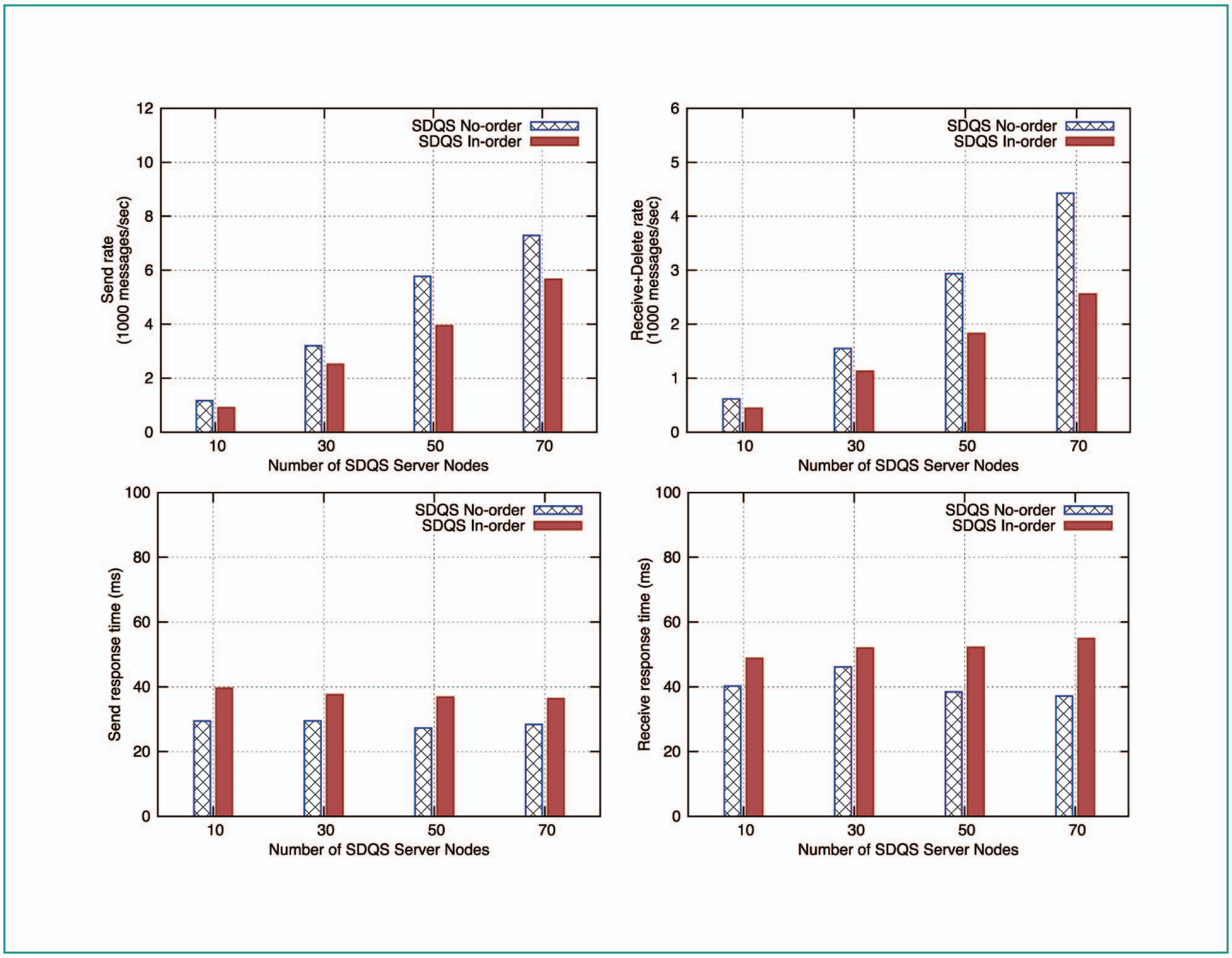

Figure 4

Comparison of system throughput between no-order SDQS queues and in-order SDQS queues. The result shows that both options achieve good scalability. FIFO delivery order can be provided at the cost of reduced overall system performance.

increases proportionally with the cluster size. Therefore, the number of requests routed to each server remains about the same level. The in-order option incurs higher latency than no-order, i.e., for the same reason outlined above.

Conclusion

Message queues provide reliable asynchronous communication channels among software components and are widely used as a form of connectivity to support large-scale, distributed, and fault-tolerant applications. As cloud computing reaches wider adoption, MQ plays an increasingly important role as a common could-based service. This paper has presented the design and implementation of SDQS, which is a cloud-based queuing service. SDQS provides a better queuing consistency model than current state-of-the-art cloud-based queuing services in that it offers exactly once delivery with the option of FIFO order while being highly available.

Preliminary evaluation has been performed on a large cloud computing test bed. The evaluation showed that SDQS provides significantly enhanced queuing consistency. While SDQS delivers messages exactly once with no duplication under all conditions, it offers clients an option to trade performance for improved delivery order. We believe that this is a significant improvement over existing cloud-based queuing services.

Acknowledgments

The authors would like to thank our colleagues Ramesh Gopinath, Mark Phillips, Jun Rao, Marc-Thomas Schmidt, Sandeep Tata, Graham Wallis, Fan Ye, and Liangzhao Zeng for their support and help.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the U.S., other countries, or both.

**Trademark, service mark, or registered trademark of Sun Microsystems, Microsoft Corporation, or Apache Software Foundation in the U.S., other countries, or both.

References

1. J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
2. Amazon, *Amazon Simple Queue Service*. [Online]. Available: http://docs.amazonwebservices.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/
3. Microsoft Corporation, *Windows Azure Platform*. [Online]. Available: http://www.microsoft.com/windowsazure/
4. IBM Corporation, *Users Guide to WebSphere eXtreme Scale*. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247683.pdf
5. IBM Corporation, *WebSphere MQ V6 Fundamentals. IBM International Technical Support Organization*, 2005. [Online]. Available: http://www.redbooks.ibm.com/abstracts/sg247128.html
6. Y. Kahlidi, "Building a computing platform for new possibilities," *IEEE Comput.*, vol. 44, no. 3, pp. 29–34, Mar. 2011.
7. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. SIGCOMM*, San Diego, CA, Aug. 27–31, 2001, pp. 149–160.
8. A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Proc. 18th IFIP/ACM Int. Conf. Distrib. Syst. Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001.
9. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proc. SIGCOMM*, San Diego, CA, Aug. 27–31, 2001, pp. 161–172.
10. Memcached. [Online]. Available: http://memcached.org/
11. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.
12. Apache Software Foundation, *HBase*. [Online]. Available: http://hbase.apache.org/
13. G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. SOSP*, 2007, pp. 205–220.
14. A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.
15. D. Pritchett, "Base: An acid alternative," *ACM Queue*, vol. 6, no. 3, pp. 48–55, 2008.
16. S. Todd, F. Parr, and M. Conner, *A Primer for HTTPR—An Overview of the Reliable HTTP Protocol*. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-phtt/
17. IBM Corporation, *WebSphere eXtreme Scale Administration Guide*. [Online]. Available: ftp://public.dhe.ibm.com/software/webserver/appserv/library/v70/xSadminguide_PDF.pdf
18. IBM Corporation, *Project Zero*. [Online]. Available: http://www.projectzero.org/
19. T. Banka, A. A. Bare, and A. P. Jayasumana, "Metrics for degree of reordering in packet sequences," in *Proc. Annu. IEEE Conf. Local Comput. Netw.*, 2002, pp. 333–342.

Received February 11, 2011; accepted for publication March 18, 2011

Zhe Zhang *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (zhezhang@us.ibm.com).*

Dr. Zhang received his B.E. degree in computer science in 2003 from the University of Science and Technology of China. He received his M.S. degree in operations research, and Ph.D. degree co-majoring in operations research and computer science from North Carolina State University, in 2006 and 2009. He worked on parallel file system techniques for the Jaguar supercomputer at the Oak Ridge National Laboratory from 2009 to 2010. Since joining IBM in 2010, he has been involved in cloud-based messaging systems.

Yuan Wang *IBM Research Division, China Research Lab, Zhongguancun Software Park, Haidian District Beijing, P.R.C. 100193.*

Dr. Wang is an IBM Research Staff Member involved in research and development work for extensible application framework. His research interests include cloud computing, service composition, and M2M infrastructure. He received his Ph.D. degree in control theory and engineering from Tsinghua University, Beijing, China, in 2006.

Han Chen *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (chenhan@us.ibm.com).*

Dr. Chen is an IBM Research Staff Member involved in research and development work in the areas of distributed computing. His research interests include messaging and event systems, cloud computing, service-oriented computing, and multimedia systems. He received a B.S. degree in computer science from Tsinghua University, Beijing, China, in 1997. He also holds Ph.D. (2003) and M.A. degrees (1999) in computer science from Princeton University.

Minkyong Kim *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (minkyong@us.ibm.com).*

Dr. Kim is working in the area of messaging systems and cloud computing. Her research interests include distributed systems, mobile computing, and pervasive computing. She received her B.S. (1996) and M.S. (1998) degrees in computer engineering from Seoul National University. She received her Ph.D. degree in computer science and engineering from the University of Michigan, in 2004.

Jing Min Xu *IBM Research Division, China Research Lab, Zhongguancun Software Park, Haidian District Beijing, P.R.C. 100193. (xujingm@cn.ibm.com).*

Mr. Xu is a Research Staff Member and the manager of the Service Composition Research department. He joined IBM after receiving his M.S. degree from Xi'an Jiaotong University in 1998. He currently works in the area of service composition for business service cloud. He holds five Granted patents and has published more than ten papers in various journals and conferences.

Hui Lei *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (hlei@us.ibm.com).*

Dr. Lei is a Research Staff Member at the IBM T. J. Watson Research Center, where he manages the Messaging and Event Systems department. His research interests include distributed systems, cloud computing, and mobile and pervasive computing, as well as service-oriented and event-driven architectures. He was a recipient of an IBM Outstanding Technical Achievement Award and more than ten IBM Invention Achievement Awards. He received his Ph.D. degree in computer science from Columbia University. Dr. Lei is a Senior Member of the Institute of Electrical and Electronics Engineers, and holds a Visiting Professor appointment at Sun Yat-sen University.