A Scalable Cloud-based Queuing Service with Improved Consistency Levels

Han Chen, Fan Ye, Minkyong Kim, Hui Lei

IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532
{chenhan, fanye, minkyong, hlei}@us.ibm.com

***Abstract*— Queuing, an asynchronous messaging pattern, is used to connect loosely coupled components to form large-scale, highly-distributed, and fault-tolerant applications. As cloud computing continues to gain traction, queuing starts to emerge as a cloud-hosted, multi-tenant service with the goals of ease of consumption and lowered cost of ownership. A number of vendors currently operate shared queuing services. They provide high availability and network partition tolerance with reduced consistency. Although they offer at-least once delivery guarantee, that is, no message is lost, they do not make any effort in maintaining FIFO order in message delivery, which is an important aspect of the queuing semantics. Thus they are not adequate for many applications. This paper presents the design and implementation of a scalable cloud-based queuing service, called BlueDove Queuing Service (BDQS). It provides improved queuing consistency—at-least once and almost-in-order message delivery— while preserving high availability and partition tolerance. It also offers clients a flexible trade-off between duplication and message order. Comprehensive evaluation is carried out on an Infrastructure-as-a-Service cloud computing platform with up to 70 server nodes and 1000 queues. It shows that BDQS achieves linear performance scalability. Meanwhile, it offers an order-of-magnitude improvement in out-of-order measurement compared to existing no-order systems. Results also indicate that BDQS is highly reliable and available.**

I. Introduction

An important tool in reining in the growing size and complexity of software systems is modularization and loose-coupling. Instead of a large monolithic codebase, modern large software systems are often implemented as distributed components that are connected via reliable communication channels using asynchronous message passing. Queuing is one of the most commonly used asynchronous messaging patterns in building these large-scale, highly distributed, and fault-tolerant applications.

There are a number of ways queuing is used in such applications. Message queues may be used directly to provide point-to-point communication between two components with a unicast semantics. Alternatively, queues can use used to connect a number of providers and consumers with an anycast semantics. In both cases, the communication is asynchronous, allowing loosely coupled components to fail independently. The overall system can continue to function even in the face of component failure, thus providing improved reliability. Commonly used application patterns include workload dispatching/load balancing, MapReduce [7]-like pipelined processing, and information aggregation and dissemination. When synchronous request-response communication pattern is required, as often is the case in enterprise application integration, message queues may be used as a reliable transport for higher-level application protocols. For example, SOAP (Simple Object Access Protocol) over JMS (Java Message Service) is widely used to support reliable Web Services invocation. Finally, queues are also frequently used as a delivery mechanism for another asynchronous messaging pattern, publish/subscribe, where a broker uses queues to buffer notification events that need to be delivered to subscribers that may be disconnected temporarily.

In order to reduce capital and operational expenses, the IT industry is gradually adopting the cloud computing model. Several service providers now operate public queuing services that can be consumed by multiple users in the cloud, for example, Amazon Simple Queue Service (SQS) [1] and Microsoft Windows Azure Queue [3]. These services choose to sacrifice consistency in favor of service availability and network partition tolerance. For queuing, reduced consistency means violation of strict queue semantics—possible message loss, duplication, and out-of-order delivery. Existing services adopt a consistency model of at-least once delivery (guaranteed no loss) with no order (random sampling of available messages). This suffices for a number of applications. However, some applications, although tolerant of out-of-order delivery, still prefer in-order delivery so as to provide better service quality at application level. For example, in an e-commerce application, the shopping component sends verified orders to the warehouse for packing and shipping. Even though strict message order is not required, it would be desirable to serve earlier orders first so as to improve customer satisfaction. Therefore it is important to improve the queuing consistency while preserving the service's availability and partition tolerance.

This paper presents BlueDove[1] Queuing Service (BDQS), a cloud-based queuing service that is highly scalable, highly available, and offers improved queuing consistency. The system is built on top of Cassandra [11], a distributed storage system. BDQS guarantees at-least once delivery and offers almost-in-order message order. A customizable parameter further allows users to specify the desired trade-off between message order and duplication. Evaluations on a cloud computing platform with up to 70 server nodes and 1000 client queues

[1]BlueDove is a cloud-based messaging service that supports both queuing and publish/subscribe. This paper describes only the queuing related portion.

show that the service is highly scalable and available, and provides an order-of-magnitude improvement in out-of-order measure compared to existing services.

The main contributions of this paper are

- It proposes a cloud-based service that provides stronger queuing consistency, while preserving the same high availability and partition tolerance features available from other competing services;
- It introduces a number of novel techniques to optimize the performance and scalability of a cloud-based queuing service. Among these techniques are queue-specific control of desirable consistency and a scalable design of client protocol for end-to-end application-level reliability;
- It demonstrates an order of magnitude improvement in out-of-order measure and linear performance scaling through comprehensive experiments with up to 70 servers and 1000 queues.

The rest of the paper is organized as follows. Section III describes the system design and architecture. Section IV describes a prototype implementation of BDQS and Section V presents a comprehensive evaluation of its performance, reliability, and availability. Section VI concludes the paper and discusses relevant future work.

II. Related Work

There is a large body of work on distributed storage mechanism. Distributed hash tables, such as CHORD [16], Pastry [14], CAN [13], use structured overlay and multi-hop routing to implement key-value stores (hashtables) over a large number of nodes. Bigtable [6] provides additional data modeling capability and has proven to be extremely useful in a number of large-scale cloud-based services.

These systems focus on the scalability and performance aspects of distributed storage and offer a consistent view of stored data to clients. What has been popularly known as the CAP theorem [5], [10] states that, in a distributed system, only two out of the following three desired properties can be provided at the same time: consistency, availability, and partition tolerance. Because high availability and network partition tolerance are important in providing services in a cloud environment to a large number of users, this has led to the Basically Available, Soft state, and Eventual consistency (BASE) model [5], [12] as opposed to the traditional ACID model that is widely employed by relational databases. Dynamo [8] is a distributed key-value store that adopts the eventual consistency model [18]. Cassandra [11] improves on Dynamo by incorporating data models from Bigtable and provides an eventually consistent, distributed storage system.

A queue is an ordered storage that restricts puts to the tail and gets to the head. A number of providers offer cloud-based queue services, such as Amazon Simple Queue Service [1] and Microsoft Windows Azure Queue [3]. Reduced consistency in queue means possible duplication, loss, and out-of-order delivery. Both services offer at-least once delivery guarantee, that is, no loss but possible duplications. They use random sampling to retrieve messages from a queue, and therefore do not provide any order guarantee. Strauss [17] has presented his experience of using Cassandra in different ways to provide queue-like connectivity between components in applications. These are examples of using Cassandra but do not represent a general purpose shared queuing service and no systematic evaluation on performance or consistency is available.

III. System Design and Architecture

In this section, we first describe the common design elements of existing cloud-based queuing systems, and analyze their characteristics and limits on consistency. Then we explain how we use three key techniques to realize improved consistency while preserving high availability and fault tolerance.

A. Common Design Elements of Existing Systems

Amazon SQS and Microsoft Windows Azure Queue are two most well known and widely used cloud-based queuing systems currently on the market. They share some elements which are common to the design of cloud-based queuing systems. In designing BDQS, we adopt similar design features, while focusing on addressing their limits on consistency.

1) Distributed Replicated Storage: Queue being a special type of data structure, it is logical and desirable to take advantage of a distributed storage system and implement queue operations on top of it. To this end, existing queue systems rely on a cluster of commodity storage nodes for persisting messages. To provide high availability and disaster recovery capability, messages are usually replicated multiple times. The replication factor (R) is determined by the service level that a service is offering—two is minimum to offer simple redundancy and server failure protection, while three may be needed to protect data against catastrophic site failure.

When a client sends a message to a queue in such a system, the front-end receives the request and selects R storage nodes from the available storage pools. The message is then replicated R times and stored on each of the R selected storage nodes. When a client tries to retrieve a message from a queue, the front-end randomly samples a number of available storage nodes and collects the oldest messages from these sampled nodes and return them to the client.

2) Visibility Timeout Based HTTP Protocol: Both SQS and Azure Queue adopt an HTTP based client API, so that a wide variety of applications and clients can access the queuing service. To simplify the programming model, a REST (Representational State Transfer) [9] style session-less protocol is used[2]. The benefit is that any application with HTTP client capability can use the service without a heavy library. The implication, however, is that additional mechanism must be put into place to ensure application level reliability against network connection reset or client crash. Note that the mere delivery of message to a receiver is not sufficient: the receiver may crash before finishing processing the message. In order to ensure end-to-end reliability, the service has to hold the

[2]In a strict sense, their protocols are not REST as they neither map resources to URIs nor use HTTP verbs for actions.

message and delete it only after a receiver has successfully finished the processing.

To this end, the protocol is designed to work in the following way. A client invokes `SendMessage` to put a message on a given queue. When it returns the message is guaranteed to be persisted. If either network or client failure occurs, the client will retry after recovery, which may result in duplication but never loss. A client uses `ReceiveMessage` to retrieve the oldest messages in the queue. Upon servicing such a request, the queuing service returns the oldest messages to the client but does not immediately delete them. They are made invisible to subsequent `ReceiveMessage` calls during a time window known as *visibility timeout*. After successfully processing the messages, a client needs to issue a `DeleteMessage` request before the timeout happens to permanently delete the messages. If timeout happens before `DeleteMessage`, the messages reappear in the queue and will be available for retrieval again. This visibility timeout mechanism in a way provides a lightweight transaction over HTTP with automatic rollback specified by the timeout value.

We can see that this protocol prevents messages from being lost in the face of either network or client failure, but it may result in duplicate messages under these failure conditions.

3) Characteristics of Existing Designs: The replication gives existing system high availability because queued messages can survive up to $R-1$ storage node failures. Also, the random sampling strategy leads to higher throughput, because the system can serve parallel client requests of a queue on any of the R replicas.

However, their design also imposes limits on the level of consistency. Because of the use of random sampling, clients *will* receive messages in a different order than they were sent. Further, an attempt to retrieve messages may return nothing if it is served by an "empty" node, even though there are messages on other nodes. Also, duplication can also occur during server failure or under concurrent client access.

B. BlueDove System Design

The main design goals for BDQS are: 1) it shall offer at-least-once delivery guarantee (no message loss) as in existing systems, but provide much better message ordering (almost-in-order, but no guarantee); 2) it shall be highly available; 3) it shall have high performance and be highly scalable.

Like existing systems, BDQS uses distributed, replicated storage and adopts the visibility timeout protocol. But we introduce three key techniques that improve queuing consistency without sacrificing high availability and partition tolerance.

1) Queue Index for Best-effort In-order: As discussed earlier, existing design is inherently incapable of preserving message order, because messages are scattered throughout the system and retrieved using random sampling. In order to preserve message order, we need to introduce a data structure that maintains the order of messages. Instead of putting the actual messages in such a data structure, which limits the system's scalability, we build a message index for each queue—essentially a list of message IDs ordered by the messages' arrival times at the front-end nodes. The actual messages are still distributed among the storage nodes. To prevent the indexes from becoming a single point of failure, they are also replicated R times in the storage system.

With the addition of the message index, the queuing operations are modified as follows. During `SendMessage`, the message ID is appended to the message index of queue and the message itself is stored separately. During `ReceiveMessage`, the message index object is first consulted to determine the ID of the oldest message, which is then used to retrieve the actual message. Because the message IDs are sorted by the time they enter the system, concurrent insertions on different replicas of the index still result in the same ordering even without synchronized clocks. More details will be explained in the technique for visibility timeout.

2) Hinted Tradeoff between Order and Duplication: One of the overall design goals is to maintain high performance. Therefore, we use multiple independent system front-ends to serve client requests concurrently. In order to maximize system throughput and increase reliability, no distributed locks are used among these component instances. The result is that, when multiple clients invoke `ReceiveMessage` operation on the same queue index object using different entry points into the system, the same message may be returned to these clients. From a correctness point of view, this does not violate the at-least once delivery model. However, it may be desirable to reduce the number of duplicate messages.

We have designed a collision avoidance algorithm to balance the probability of duplicates and the message delivery order. During the `ReceiveMessage` operation, instead of always retrieving the oldest message ID from the message index, the system will retrieve a random one among the oldest K IDs. The larger the value of K, the less likely that concurrent receivers will obtain the same message, but the more out-of-order the returned message sequence will be. The system exposes the value of K as a configurable parameter for each queue. A value $K = 1$ produces the best order with potential duplications, whereas a large value of K reduces duplication.

3) Visibility Timeout without Using Timers: As discussed in previous section, the visibility timeout protocol used by existing cloud-based queuing systems offers a lightweight end-to-end protection mechanism for application level reliability. BDQS adopts the same protocol for its client API. Conceptually, realizing the visibility timeout requires simply maintaining a timer for each received-but-not-yet-deleted message in the system. However, a direct and naïve implementation using off-the-shelf timer packages, such as Java Timer or Quartz, has a number of deficiencies. First, most these timer packages keep the timer related objects in memory. Therefore the timer information cannot survive a node failure. Although such a failure will not result in message loss, as previous invisible messages will just reappear when the timers are lost, unnecessary duplication of messages will occur. Second, to ensure scalability, multiple front-end nodes are used in a cloud-based service. For the timer to work correctly, a `DeleteMessage` request must be forwarded to the node

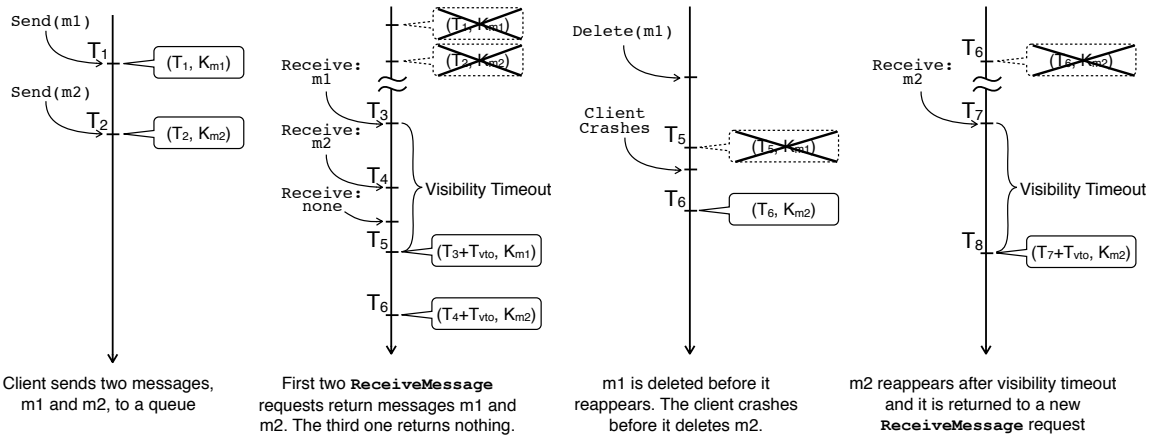

Fig. 1. A sequence of queue operations illustrating how visibility timeout is implemented in BDQS.

that previously handles the `ReceiveMessage` for the same message and thus holds the timer in memory. Otherwise, the `DeleteMessage` will fail, resulting in increased duplication.

To address these issues, we augment the semantics of the timestamp information in the message index to support visibility timeout. Each entry in the message index list is a tuple (T, I), where T is a timestamp indicating a point in time after which the message referenced by I is available for retrieval. The tuples in the list is kept sorted in ascending order of the timestamps. Before we delve into the details of the various operations, we point out that the essence of this design is the fact that T may be a timestamp in the future, in which case, the message is considered "invisible." To best describe how it work, it is easier to consider a sequence of operations (illustrated in Figure 1).

- When `SendMessage` is invoked, the system creates a unique message ID I_{msg} for the new message and records the current time T_{now}. A tuple $(T_{\text{now}}, I_{\text{msg}})$ is inserted to the message index list of the given queue. This indicates to the system that this particular message becomes available for retrieval (visible) at T_{now}, that is, it can be received immediately. As stated earlier, the content of the message itself is stored separately in the distributed storage system, indexed by I_{msg}.
- When `ReceiveMessage` is invoked, the system retrieves a random tuple (T, I_{msg}) among the first K ones in the sorted message index list for the queue, as described in *Hinted Tradeoff between Order and Duplication*. There are two cases with respect to the value of T in relation to the current time T_{now}. If $T \leq T_{\text{now}}$, this message is available to be returned. To temporarily make the message invisible, the value T of the tuple is modified to be $T' = T_{\text{now}} + T_{\text{vto}}$, where T_{vto} is the visibility timeout value. If $T > T_{\text{now}}$, it means that all messages in the queue are currently invisible, and therefore the queue is empty and nothing is returned.
- When `DeleteMessage` is invoked, the system obtains the message key I_{msg} from the request, removes the tuple (T, I_{msg}) from the message index list, and also deletes the message content itself.

This algorithm realizes timer functions with minimal overhead to the message index. Because the timer information is stored in the distributed storage as part of the message index, it solves the two aforementioned problems. First, the timers become persistent now and can survive node failure. Second, the timer information is available to any front-end node accessing the distributed storage, eliminating the need for maintaining states in the front-end to correlate a `DeleteMessage` request with its preceding `ReceiveMessage` request.

C. Deployment Architecture

BDQS consists of three main logical components. The *distributed storage* component provides persistence for the messages, message indexes and other queue related metadata across a commodity cluster. The *queue operations* component implements the aforementioned queue API supporting visibility timeout. Because all states related to queuing operation are persisted in the storage layer, the queue operations component is designed to be completely stateless so that multiple instances can be deployed to support concurrent client access. It exposes its functionality through a native API. To enable a wide variety of clients to access the service, an *HTTP REST* component provides a RESTful interface of the native queue API via HTTP binding, using a protocol similar to that of SQS.

In a cloud-based deployment, a number of VM instances are created on an Infrastructure-as-a-Service (IaaS) platform, each consisting of one instance of each component described above, as shown in Figure 2. A dispatching mechanism routes incoming client requests to a REST interface instance. This can be achieved using either a dedicated HTTP dispatcher or round-robin DNS. To provide adequate service level to clients, a separate monitoring mechanism controls the dynamic scaling of the VM cluster. There are two different reasons for scaling the system. First, more storage capacity is needed to handle increased number of queued messages. Second, more VMs are required to cope with increased request rate from clients. The

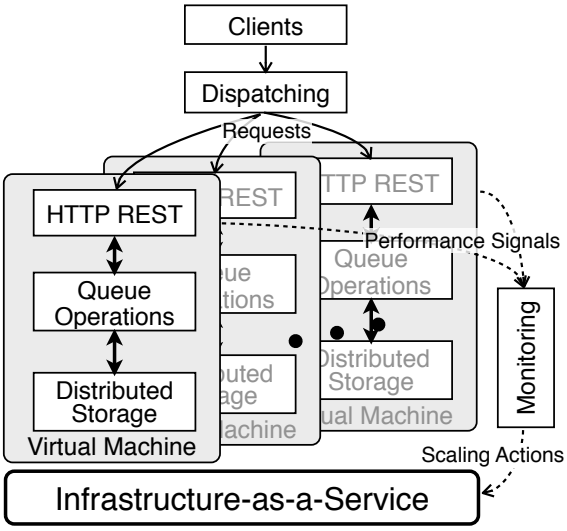

Fig. 2. The system architecture of BDQS allows it to be deployed as a shared service in an IaaS environment with elastic scaling.

monitor collects information about current storage size and average response time, decides if the system should be scaled up or down, and interacts with the IaaS layer to carry out these actions. Request dispatching and dynamic scaling are two issues common to any cloud-hosted shared service and are orthogonal to the proposed queuing service; hence they are not discussed in details in this paper.

IV. Implementation

We have implemented a prototype of BDQS according to the system architecture outlined in the previous section. This section describes a few details of the prototype system.

A. Cassandra and Data Store Design

We have chosen Cassandra as the distributed storage provider in the prototype of BDQS. Cassandra is a distributed storage system based on a DHT design with a more sophisticated data model. A simple DHT provides an abstraction of a map from keys to values ($k \rightarrow v$) where the entries are distributed across the system according to a hashing scheme. Without going into great details, Cassandra can be described as a four-level nested maps from keys to *super column families* to *column families* to *columns* to values ($k \rightarrow \text{SCF} \rightarrow \text{CF} \rightarrow C \rightarrow v$), where super column family is an optional construct. The collection of column families, columns, and values referred to by a key is called a *row* in Cassandra. As in a DHT, rows are distributed across the entire system according to some hash value of the key.

Cassandra supports high availability by replicating data among the storage nodes, where the number of replicas N is a configurable parameter. When reading and writing data, Cassandra provides several consistency policies that specify how many replicas have to acknowledge before a request is returned. They are commonly classified as 1 (single replica), N (all replicas), and Q (a quorum of at least $(N+1)/2$ replicas) [18]. To provide high system performance, no distributed locks are used in the higher layer algorithms as described before and strict queuing consistency is not guaranteed by design. As a result it is not necessary to require strong storage consistency from Cassandra. Therefore we use the single replica policy for both reads and writes.

There are three main resource types in BDQS—account, queue, and message. They are stored as Cassandra rows (shown in Figure 3), 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 the data across the entire Cassandra cluster and yet allows direct access to any objects without having to search through multiple levels of indirection.

`Accounts` is the root object holding references to all accounts. It contains pointers to individual account objects. An account has two column families: `Metadata` and `Queues`. The former contains account information such as owner's name, console logon password, secret key, etc. The latter contains references to the queues in the account. Queues are the cornerstones of the system. Among the several column families, `Messages` stores the IDs of all messages in the queue and `Appearances` stores the timestamped message index list as described in previous section. Finally, the message objects store the actual payload of the queued messages.

B. Prototype Implementation

The prototype implementation uses Apache Cassandra incubator version 0.6.0. No modifications are made to the source and the public Thrift interface is used to access its service. The aforementioned data store schema is defined in `storage-conf.xml`. Cassandra service is run as a daemon process using 32-bit JDK 1.6 with 3.5GB maximum heap size. Default locations for Cassandra data directories are used and they reside on the same VM storage volume. We remark that this is not the optimal deployment strategy for Cassandra, but it is sufficient for the scalability and consistency evaluation of BDQS. In a tuned production environment, the actual performance of BDQS would be higher than presented here.

The queue operations component is implemented in about 6000 lines of Java. It provides a Java interface to account-, queue-, and message-related operations. It also performs user authentication and access control. The HTTP REST interface along with an administrative console is implemented with about 1000 lines of PHP/HTML and 3000 lines of Java servlet handlers. These two components are hosted on WebSphere sMash [2], 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.

V. System Performance Evaluation

The prototype of BDQS is deployed on an internal research cloud computing platform. Simulation drivers are used to generate a synthetic workload to benchmark the system's

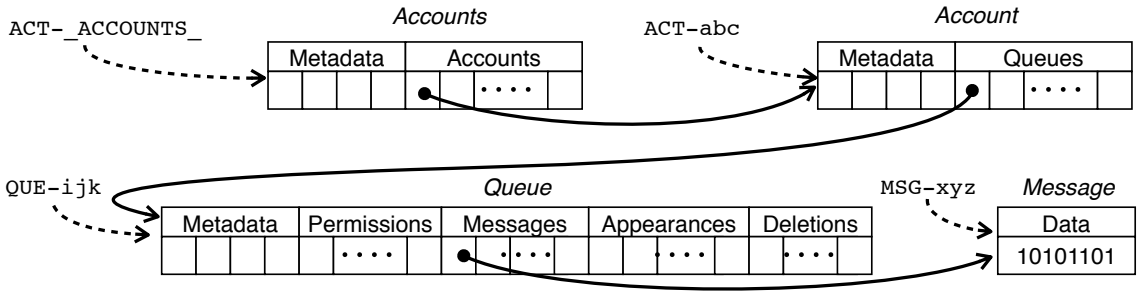

Fig. 3. Cassandra data store model of BDQS.

performance and consistency. We also compare the consistency level of BDQS against an existing service, Amazon SQS.

A. Evaluation Methodology and Metrics

A cluster of N virtual machines is used to host the cloud-based message queuing service. Each VM has 1 processor core, 6GB of RAM, 36GB of disk space, and is connected to a Gigabit Ethernet network. The replication factor of Cassandra is set to be 3, which is typically used in a multi-data-center deployment to provide high availability. Cassandra configuration recommendation is followed to set its thread pool size appropriately for uniprocessor VMs: 4 threads for read and 8 threads for write. The thread pool size for the sMash container is set accordingly to 16, so that the HTTP REST handler layer does not become a bottleneck.

The test workload is generated by two programs—sender and receiver. To simulate a shared service deployment scenario, a separate cluster of 20 VMs (with the same hardware configuration as the server VMs) is used to host the senders and receivers. Each VM runs a single sender and receiver. They are configured to use the HTTP REST interface to access the queuing service. A sender program simulates q sending applications, resulting in a total number of queues in the system $Q = 20q$. Each sending application 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, 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 $M = Q \cdot T_{\text{send}} \cdot m$.

Unless mentioned otherwise the queues are configured with consistency level hint of $K = 1$, that is, favoring order, and the default visibility timeout value is set to 10 seconds. This relatively small number reflects the nature that the service is being stress-tested with very short turnaround time. In real application scenarios, a timeout value would have to be set commensurate with the application's processing speed.

To better characterize the sending and receiving behavior of the system, the sending and receiving operations are executed in two separate 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, the trace data are collected, and the average values of the following performance metrics are computed. **Send rate** is the number of `SendMessage` operations that the system services in one second. **Receive+Delete rate** is the number of combined `ReceiveMessage` and `DeleteMessage` operation cycles that the system services in one second. **Send response time** is measured from the moment a `SendMessage` operation is invoked to the moment the response is received. **Receive response time** is measured from the moment a `ReceiveMessage` operation is invoked to the moment the response is received.

The following consistency metrics are also measured: **duplication rate**, **loss rate**, and **message order**. 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 it is not received at all. The loss rate is $\epsilon = M_{\text{loss}}/M$. 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-orderness, two metrics are defined. The first metric, **out-of-order rate**, is defined as the ratio of the minimum number of messages that have to be re-ordered to the total number of messages in the sequence. To calculate this value, messages are tagged with sequence numbers at the sending end and dynamic programing algorithm is used to find the longest monotonically increasing sub-sequence at the receiving end. Messages not in the sub-sequence are considered out-of-order. The second metric, **average displacement**, is defined as the average difference of message positions in the received sequence versus those in the correct sequence.

Out-of-order rate measures the absolute number of out-of-order messages while average displacement reflects the average re-ordering delay at the receiving end if order is desired. For example, consider a sequence of message $\{m_1, m_2, m_3, m_4, m_5\}$. If the received order is

$\{m_2, m_1, m_3, m_4, m_5\}$, the out-of-order rate is 20%, since only m_1 needs to be re-ordered, and the average displacement is $(1 + 1 + 0 + 0 + 0)/5 = 0.4$. If the received order is $\{m_2, m_3, m_4, m_5, m_1\}$, the out-of-order rate is still 20% but the average displacement is $(1 + 1 + 1 + 1 + 4)/5 = 1.6$. Together these two metrics paint a comprehensive picture 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 [4], which combines out-of-order measure with duplicate measure.

B. Performance of the Queuing Service

1) System Scalability: To evaluate the overall scalability of the system, the cluster size N is varied from 10 to 70. For each number N, a series of tests is performed with increasing workloads. During the tests, the number of queues (q) is varied from 1 to 50 per sender, resulting in total number of queues (Q) in the system between 20 and 1000. The following parameters are fixed: $T_{\text{send}} = T_{\text{recv}} = 3$, $m = 100$, $\delta_{\text{send}} = \delta_{\text{recv}} = 0$, and $L = 2048\text{B}$.

The top two plots in Figure 4 show how the rates of send and receive react to increasing workload. For each cluster size, the throughput increases as the workload increases. After a certain point, the utilization of the system reaches saturation and the throughput stops increasing. Note that, the `ReceiveMessage` operation involves more Cassandra operations than `SendMessage`. Therefore the Receive+Delete rate is slower than the Send rate under the same test condition.

The bottom two plots in Figure 4 show how the throughput changes in relation to cluster size under the same workload. The result indicates that the system exhibits linear throughput scalability. Since the highest workload (1000 queues) is not able to saturate the two largest systems ($N = 50$ and $N = 70$) completely, the total curves in these two plots show less than linear increase. We expect that a higher workload would demonstrate the full capacity of the system.

2) Queuing Behavior: The previous scalability test stresses the system to determine its maximum throughput. Queuing theory states that as a system approaches its maximum throughput, its response time increases exponentially, rendering the system unresponsive. Another series of tests is performed to study the queuing behavior of the queuing service, that is, how response time changes in relation to throughput. The number of queues (Q) and the send/receive delays (δ_{send} and δ_{recv}) are varied to create different levels of workloads, from very light to very heavy. For four different cluster sizes ($N = 10, 30, 50, 70$), the response times of `SendMessage` and `ReceiveMessage` are measured and plotted against the respective rates.

The results are shown in Figure 5. It shows that the system exhibits a typical M/M/k queuing behavior. When the system is not stressed, e.g., when the throughput is less than 80% of the maximum, the system is very responsive—the average response times for `SendMessage` and `ReceiveMessage` are below 200ms. When the workload increases beyond a threshold the system saturates and the throughput stops increasing. Meanwhile, the response times start to increase dramatically.

As discussed earlier, the average response time of the system can be monitored and used as a trigger to control the scaling of the server cluster. When it increases past a threshold, new server nodes are added to the system to increase its overall capacity. This enables the system to become extremely elastic and adapt to workload fluctuation. The control mechanism is common among a number of hosted shared cloud-based services and is not unique to the queuing service. Therefore it is not discussed in this paper.

C. Consistency Level of the Queuing Service

A series of experiments is performed to evaluate the consistency level offered by BDQS. The following configuration parameters are used: $N = 50$, $Q = 400$, $T_{\text{send}} = 3$, $m = 100$, $L = 2\text{KB}$, $\delta_{\text{send}} = 0$, $\delta_{\text{recv}} = 1\,\text{s}$. The number of receive threads per queue, T_{recv}, is varied from 1 to 3 to generate different levels of concurrency in `ReceiveMessage` operations. Three consistency level hints are evaluated, $K = 1, 2, 3$. To compare them against a no-order system, the same workload is also tested on Amazon SQS.

In all tests, the loss rate ϵ is zero, that is, no messages are lost and corrupted. The three plots in the top row of Figure 6 show how the three consistency metrics—duplication rate, out-of-order rate, and average displacement—changes with different degrees of concurrency. The three plots in the bottom row show how these metrics vary with different consistency level hints (K) given the same concurrency. These results show that, with no concurrency ($T_{\text{recv}} = 1$), both BDQS and a no-effort system produce negligible amount of duplication. When concurrency increases, duplicate rate increases for BDQS. The rate of increase depends on the consistency level hint—the more order is favored, the more duplicates are produced. In a no-order system, random sampling is used to retrieve message. Therefore the duplication rate remains low when concurrency increases.

On the other hand, BDQS produces significantly fewer out-of-order messages. With consistency level hint $K = 1$, almost all messages are delivered in order, whereas the no-order system delivers about 50% of messages out of order. Out-of-order measures increase as K increases, but they are much smaller than those of the no-order system. Even with $K = 3$, the average displacement of BDQS is an order-of-magnitude smaller than that of a no-order system. This result shows that BDQS offers client a flexible way to specify the desired tradeoff between the two aspects of consistency—order and duplication. In fact, the no-effort approach can be viewed as a special case of BDQS, where $K = \infty$.

D. Reliability and Fault Tolerance of the Queuing Service

An important aspect of a shared, cloud-based service offering is its reliability and availability. To study how well BDQS tolerates failures, three tests are performed with artificially injected failures in the system. The first test evaluates how

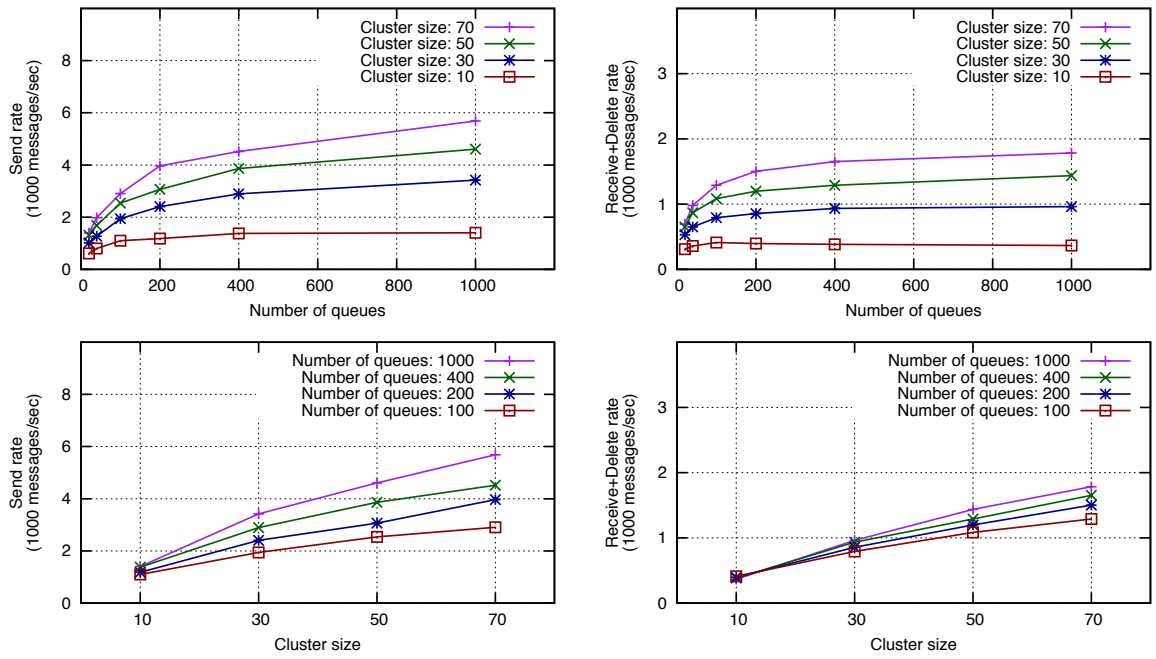

Fig. 4. How system throughput reacts to workload and cluster size changes.

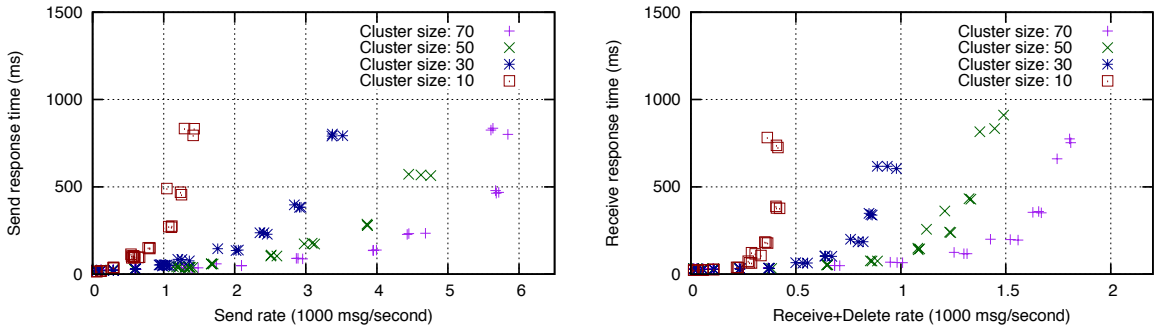

Fig. 5. The relationship between response time and throughput roughly follows a M/M/k queuing model—the response time increases gradually initially as the workload increases; when the throughput approaches its maximum value the response time increases dramatically.

well the system handles network failures while the second test evaluates how well the system copes with client failures. The third test evaluates the system's ability to deal with server failure.

Network failure is simulated by randomly dropping either the HTTP request or HTTP response according to a probability in the client API library. Client failure includes two cases—sender failure and receiver failure. Sender failure can not result in loss message, but will cause duplicate messages to be sent to the service in proportion to the sender failure rate. Therefore sender failure is not explicitly evaluated. Receiver failure may result in end-to-end message loss if no reliability protocol is used. It is evaluated in the test by randomly quitting and restarting a receiving thread according to a preset probability.

The test workload for evaluating network and client failure tolerance is generated with the following parameters: $N = 50$, $Q = 400$, $T_{\text{send}} = 3$, $m = 100$, $L = 2\text{KB}$, $\delta_{\text{send}} = \delta_{\text{recv}} = 0$.

Figure 7 shows how the performance metrics and consistency metrics vary with increasing failure probability ($p = 0.0001, 0.001, 0.01, 0.1$). (In all cases, the message loss rate ϵ is 0, and therefore is not plotted.) The results indicate that BDQS handles network and client failure well. The performance metrics degrade slightly with a very high failure rate of 10%. The consistency metrics degrade significantly with failure rate of 0.1, but stay relatively constant for other lower failure rate numbers. We remark that the failure rate in reality is probably closer to the lower end of the simulation than the upper end. This shows that BDQS is able to provide good reliability in the face of network and client failures.

Server failure is simulated by randomly killing the Cassandra and sMash processes on a server VM and restarting those processes after a recovery period. During the recovery period, clients continue to send and receive messages without stopping. The frequency and duration of the server failure are

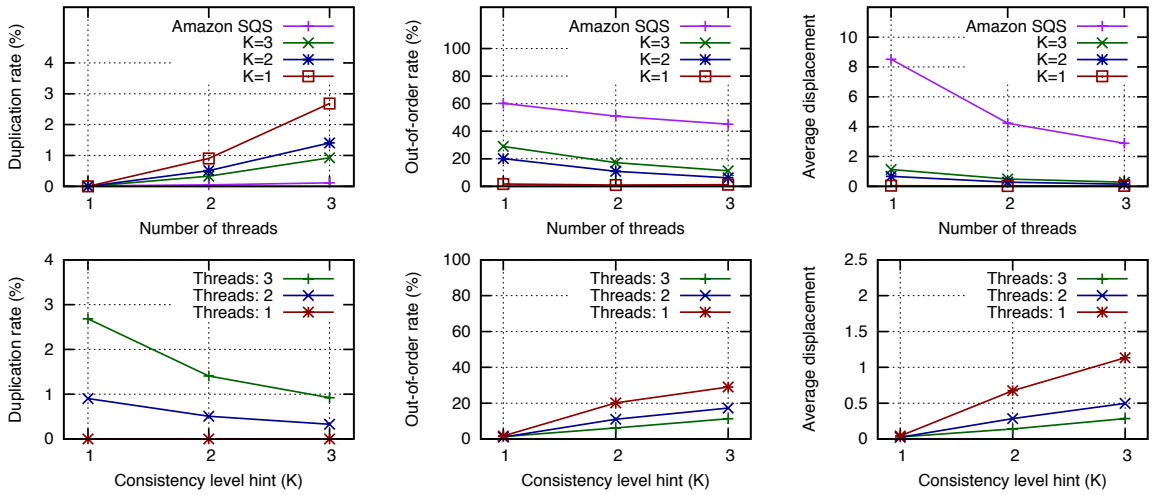

Fig. 6. By setting the value of consistency level hint (K), an application can adjust the tradeoff between delivery order and duplication.

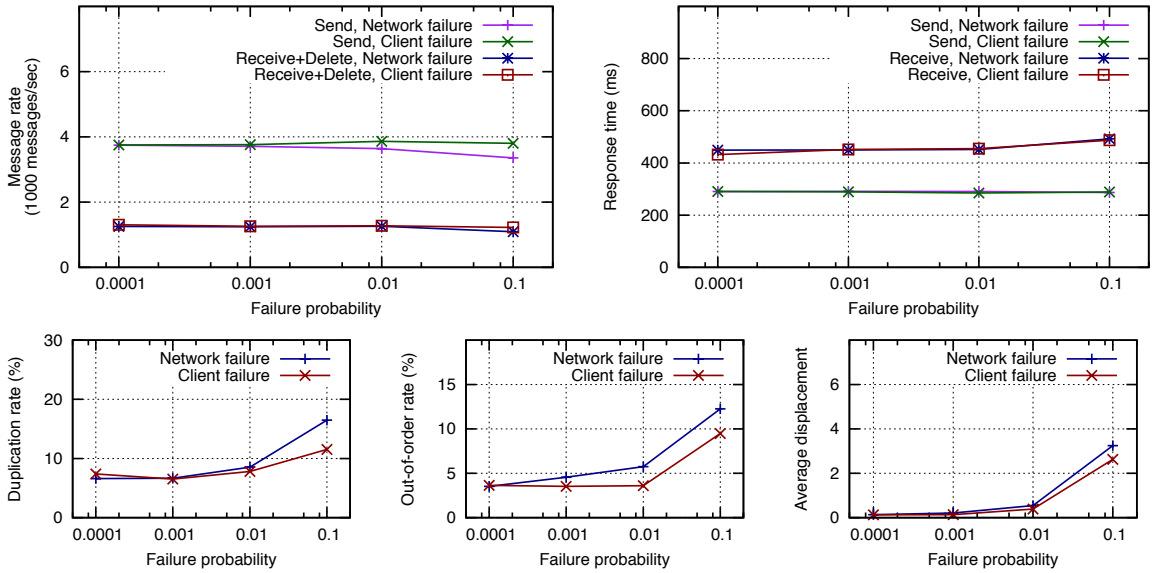

Fig. 7. Queuing performance and consistency metrics in network and client failure scenarios.

controlled by two parameters: Mean Time To Failure (MTTF) and Mean Time To Recovery (MTTR). In order to complete the test in a reasonable amount of time, three MTTF numbers are used: 100s, 1000s, and 10000s. MTTR is fixed at 10s. We remark that these numbers are extremely short compared to what would be expected from a real world deployment. Therefore, the results illustrate worst-case scenarios. The following parameters are used to generate the test workload[3]: $N = 50$, $Q = 200$, $T_{\text{send}} = 3$, $m = 100$, $L = 2\text{KB}$, $\delta_{\text{send}} = 0.5\,\text{s}$, $\delta_{\text{recv}} = 2\,\text{s}$.

The $R = 1, W = 1$ curves in Figure 8 show how the performance metrics and consistency metrics change with the different server MTTF numbers. (Again, there is no message loss, and therefore ϵ is not plotted.) The results indicate that the send operation is not significantly affected by server failure, while the performance of the receive operation degrades as the server failure frequency increases. The consistency metrics degrade when servers become less reliable. However the values are still acceptable. Again, we caution that server MTTF in real life will be significantly higher than even the highest end number in the test. Therefore, we have reason to expect that BDQS will perform extremely well in real deployment.

[3]The test clients are slowed down so that there will be enough server failure events during a test run.

VI. Conclusion and Future Work

Queues provide reliable, asynchronous communication channels for connecting software components. Queuing is widely used as a form of connectivity to support large-scale, distributed, and fault-tolerant applications. It plays an

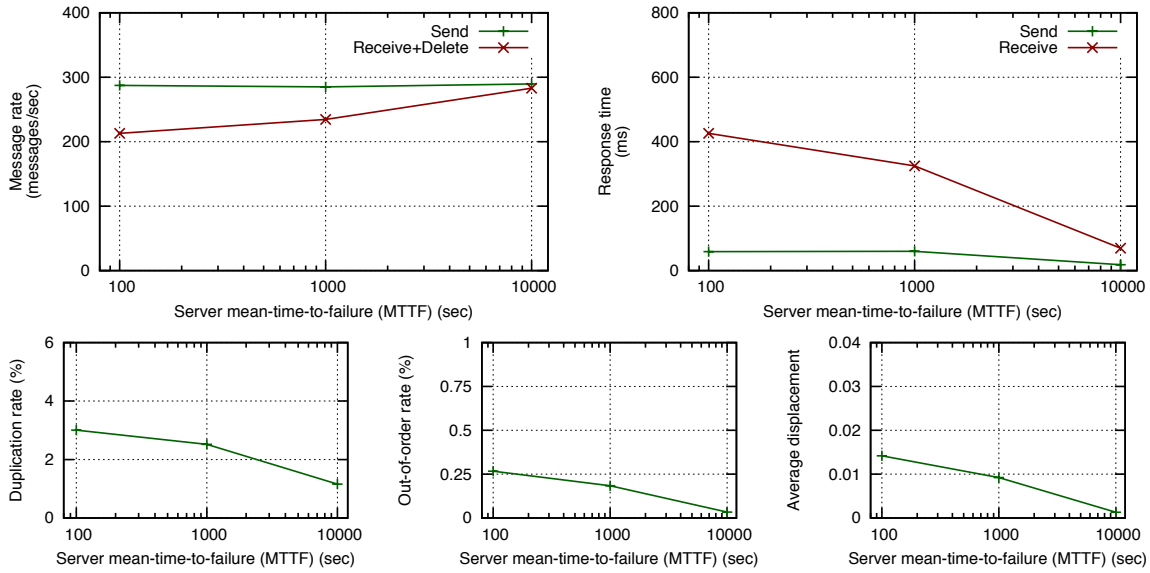

Fig. 8. Queuing performance and consistency metrics under different server failure models.

important role in cloud computing when offered as a shared could-based service to applications. This paper presents the design and implementation of BlueDove Queueing Service, a cloud-based queuing system. It builds on Cassandra and provides high availability and network partition tolerance. At the same time, it offers almost-in-order and at-least once delivery guarantee with user-configurable trade-off between delivery order and duplication. This is an improved consistency model than current state-of-the-art queuing services.

Extensive evaluation indicates that BDQS performs well and is highly scalable. It also provides clients with flexible consistency tradeoff options between order and duplication. Experiments show that BDQS delivers significantly reduced number of out-of-order messages with similar duplication rate when compared to existing systems. We remark that each replica of the message index of a queue resides on a single storage node. Thus the node's storage capacity divided by the size of message keys is the upper bound of the number of messages a queue can hold. This upper bound is large because a message key contains only a timestamp and an ID. In return, the system is able to offer almost-in-order delivery. Therefore, it is a trade-off that potential applications need to consider in deciding what queuing service to use.

There are a number of future improvements that can be made to the system. First, the current consistency level hint is set statically. As the experiments show, the duplication rate depends on the degree of concurrency. Therefore, it is desirable to design an algorithm that adjusts the value K dynamically, thus achieving the best possible combination of in-order delivery and no-duplication. Second, under the CAP framework, Cassandra provides the AP combination with reduced consistency. It will be interesting to study the implication of how distributed storage systems with other combinations (e.g., CP) affect the overall queuing consistency.

REFERENCES

[1] Amazon Simple Queue Service. `http://aws.amazon.com/sqs/`.

[2] Project zero. `http://www.projectzero.org/`.

[3] Windows Azure Platform. `http://www.microsoft.com/windowsazure/`.

[4] BANKA, T., BARE, A. A., AND JAYASUMANA, A. P. Metrics for degree of reordering in packet sequences. In *Annual IEEE Conference on Local Computer Networks* (2002), p. 333.

[5] BREWER, E. A. Towards robust distributed systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing* (2000), p. 7.

[6] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (2008), 1–26.

[7] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113.

[8] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: amazon's highly available key-value store. In *Proceedings of SOSP '07* (2007), pp. 205–220.

[9] FIELDING, R. T., AND TAYLOR, R. N. Principled design of the modern web architecture. *ACM Trans. Internet Technol. 2*, 2 (2002), 115–150.

[10] GILBERT, S., AND LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (2002), 51–59.

[11] LAKSHMAN, A., AND MALIK, P. Cassandra-a decentralized structured storage system. In *The 3rd Intl. Workshop on Large Scale Distributed Systems and Middleware* (2009).

[12] PRITCHETT, D. Base: An acid alternative. *Queue 6*, 3 (2008), 48–55.

[13] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SHENKER, S. A scalable content addressable network. In *SIGCOMM '01* (2001).

[14] ROWSTRON, A. I. T., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware '01*.

[15] SLEE, M., AGARWAL, A., AND KWIATKOWSKI, M. Thrift: Scalable cross-language services implementation. *Facebook* (2007).

[16] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*.

[17] STRAUSS, D. Scalable queuing with flexible service levels. (video). *The first Apache Cassandra Summit* (2010).

[18] VOGELS, W. Eventually consistent. *Commun. ACM 52*, 1 (2009), 40–44.