A Scalable Cloud-based Queueing 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`

Queuing, an asynchronous messaging paradigm, is used to connect loosely coupled components to form large-scale, highly-distributed, and fault-tolerant applications. As cloud computing continues to gain traction, a number of vendors currently operate cloud-based shared queuing services. These services provide high availability and network partition tolerance with reduced consistency—at-least once delivery (no-loss) with no effort in message order. This paper presents the design and implementation of BlueDove Queuing Service (BDQS), a scalable cloud-based queuing service with improved queuing consistency. BDQS provides at-least once and best-effort in-order message delivery model, while preserving high availability and partition tolerance. It also offers clients a flexible trade-off between message duplication and message order. Performance evaluation shows that BDQS achieves linear throughput scalability and offers significantly reduced out-of-order messages compared to no-order queuing services.

BDQS consists of three main components. Cassandra, an open source distributed key-value store, provides highly available and partition tolerant *persistence*. The *queue operations* component implements a queue API using Cassandra API. The queue operations component stores all states in Cassandra; therefore, multiple instances can be deployed to maximize overall system throughput. 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. In a cloud-based deployment, VM images consisting of one instance of each component described above are deployed on an Infrastructure-as-a-service cloud, as shown in Figure 1. A dispatching mechanism routes client requests to a REST interface instance. To provide adequate service level, a separate monitoring mechanism controls the dynamic scaling of the cluster.

Messages in BDQS are distributed among all available nodes. To provide best-effort in-order delivery, an index of message sequence is maintained for each queue. In order to maximize system throughput and assure availability, no distributed locks are used among the multiple queue operations instances. The result is that, when multiple clients invoke `ReceiveMessage` operation on the same queue object using different entry points into the system, the same message may be returned to these clients. BDQS uses a collision avoidance algorithm to balance the probability of duplication and the message delivery order. Instead of always retrieving the first message in the index, the system retrieves a random one among the first K messages. The larger the value of K, the less likely that concurrent receivers will obtain the same message, but the more out-of-order

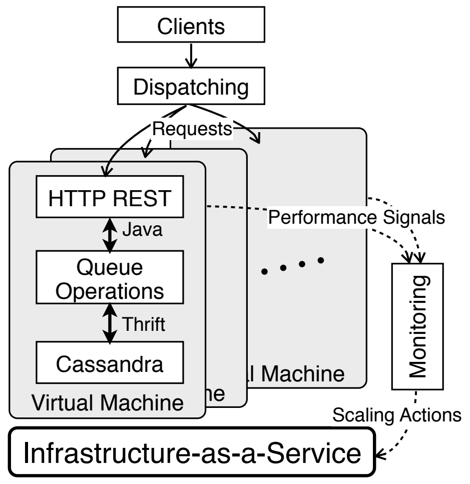

Fig. 1. Deployment architecture of BDQS on an IaaS cloud.

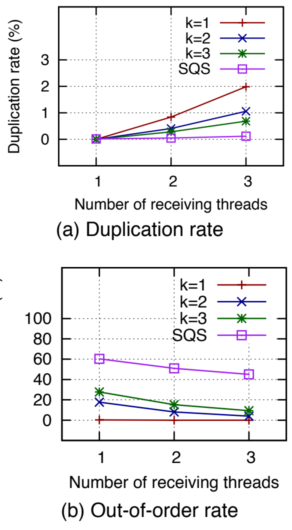

Fig. 2. Consistency level hint (K) controls the tradeoff between order and duplication.

the returned message sequence will be. Thus K acts as a consistency level hint, which the system exposes as a configurable parameter for each queue.

A prototype of BDQS has been implemented and deployed on an IaaS platform. Simulation drivers are used to generate a synthetic workload to benchmark the system's performance. Evaluation results show that the system's throughput scales linearly versus the cluster size. (Details are not reported here.) To quantify the improvement in consistency, we vary the number of receiver threads per queue, T_{recv}, from 1 to 3 to generate different levels of concurrency in `ReceiveMessage` operations. Three consistency level hints are used, $K = 1, 2, 3$. To compare the consistency level against a no-order system, the same workload is tested on Amazon SQS. Figure 2(a) shows that with no concurrency ($T_{\text{recv}} = 1$), both BDQS and the no-order system produce negligible amount of duplication. When concurrency increases, duplication 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, Figure 2(b) shows that 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. 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 $K = \infty$.