Blizzard: A Distributed Queue

Amit Levy (levya@cs), Daniel Suskin (dsuskin@u), Josh Goodwin (dravir@cs)
December 14th 2009
CSE 551 Project Report

1 Motivation

Distributed systems have received much attention in recent years. As the Internet increases in size and scope, there is an increasing availability of large data sets as well as massive numbers of concurrent users of a given resource or application. These are but two examples of tasks that require more than a single node to handle. As such, the technical challenges of building single, coherent systems that span multiple nodes have been consistently tackled and improved upon by both commercial entities and research groups. Topics such as durability, consistency, fault tolerance, and scalability are all current buzzwords, and each of these areas can be implemented in various ways with various trade-offs. As such, our primary goal is to undertake the creation of a distributed system addressing these various topics. Doing so will provide concrete insights into the planning, construction, and interaction of these various components of a distributed system.

One area where a distributed system could be necessary is the processing of large amounts of data. Specifically, many computing tasks involve the use of a queue data structure to logically enqueue data that needs to be processed and dequeue that data when it is ready to be processed. Various producer/consumer models also use a logical queue to handle the transmission of data between appropriate processes. E-commerce transactions and distributed computation are two possible examples of tasks that may benefit from the use of a logical queue, yet may involve workloads that reach proportions unmanageable by a single machine. We propose and undertake to create Blizzard, a distributed queue that would be able to handle these higher workloads. The design will involve considering the topics of fault tolerance, durability, scalability, and concurrency while maintaining a single, logical queue amongst a set of nodes.

2 Architecture/Implementation

In considering what programming language to proceed in, we took into account the experience of group members with various languages and thought about proceeding in either Ruby or Java. Ruby was decided upon, primarily because some of our initial design concepts may have benefited from using the Chord algorithm, and one group member already had a working Chord system in Ruby. Though we ended up abandoning Chord as an element of our final design, Ruby has worked reasonably well for this project. In particular the experience of group members with RPC's in Ruby has been beneficial in getting disparate nodes to function well with each other.

In terms of physical deployment, Blizzard is designed to run on commodity machines connected via any IP compatible network. For testing purposes we deployed Blizzard using CSE lab machines as the various nodes in the system.

2.1 Goal Properties

Below we list and explain the goals which affect the design of our system, in descending order of the goals' levels of influence. The goals are the support of: fault tolerance, persistence of enqueued data, concurrent operations, scalable performance, and preservation of FIFO order.

Fault Tolerance
The main goal of our system is fault tolerance, which means it should recover from failures without losing data in the queue and without losing information about the structure of the queue. To the user, a failure behind the scenes should appear as a request that returns an error or as a momentary slowdown in performance, rather than a permanent loss of data.

Persistence
As mentioned above, data stored by Blizzard should persist through failures. Specifically, unrecoverable machine failure and failure during recovery from other failures should not cause data stored in the queue to disappear from the system, and neither should it cause information about the structure of the logical queue to disappear. The percentage of nodes that can simultaneously experience states of failure should be a configurable property of Blizzard, and is controlled in our implementation by changing the factor of data replication.

Concurrency
As Blizzard is meant to serve requests from many clients simultaneously, it should handle multiple concurrent enqueue and dequeue requests correctly. And by correctly we mean that multiple requests should be able to return to their respective requesters as successes. When an enqueue operation completes successfully, the data enqueued should be represented in the logical queue, and retrievable by a future dequeue request. A successful dequeue operation means that the data dequeued is no longer represented in the logical queue, and thus is not retrievable by a future dequeue operation.

Scalable Performance
To justify the existence of Blizzard, we'd like it to be able to handle more operation requests per unit of time than a queue which exists on a single machine, and we'd also like it to allow for more elements to exist in a single queue than is possible using a queue which is contained on a single machine. Maximum request throughput should also scale with the number of machines in a Blizzard instance. However, we allow for throughput to be sacrificed while fault tolerance mechanisms recover from failures.

Preservation of Order
As a final consideration, exact FIFO behavior is not expected of Blizzard. However, if a value U enters the queue earlier than a value V, then U should have a higher priority than V to be selected to be returned by a dequeue operation, while they both exist in the queue.

2.2 Implementation

Initially, two disparate approaches were considered: using a single master that maintains the logical state of the queue, and using a peer to peer distributed system such that every node in the system would be equal and state would be distributed and replicated amongst them. Though the peer to peer approach avoids the downsides of a single point of failure and a single machine limit in scalability, we decided to pursue the single master approach in order to simplify the design and implementation of the system.

Thus, an overview of the single master approach is as follows. A single master node is in charge of the logical queue structure and handles all enqueue/dequeue requests, assigning a unique ID to every element in the queue. However, to enable high throughput, the actual data does not pass through the master node. We feel the additional message throughput for the master is worth the tradeoff of sending more messages back and forth through the client. Rather, the master points clients to appropriate data nodes to store or retrieve data. In addition, the master handles replication by monitoring data nodes and initializing duplication of necessary data if a data node fails. More detailed discussion of the various components of Blizzard is given below.

2.2.1 Planned Goal fulfillment

As mentioned above, we would like Blizzard to meet certain goal properties. We envision that the single master approach could meet these properties in the following ways:

- Persistence and Fault Tolerance - The master handles data persistence in the presence of data nodes that may fail by ensuring that a given item in the queue is physically stored on multiple nodes at any give time, initiating replication as necessary to maintain a given duplication threshold. The master allows queue persistence in the presence of its own failure by logging all necessary queue metadata to disk, recovering from the log when necessary.
- Concurrency correctness - The master maintains the entire logical queue structure, and can thus ensure that any enqueue or dequeue of a given item ID is only performed once, since it doesn't have to synchronize changes to the queue with any other party.
- Scalable Performance - As the master only handles queue metadata, the enqueue/ dequeue rate should be high, enabling multiple clients to transfer the actual item data to/from data nodes simultaneously. As the data is distributed amongst many nodes, the maximum size of the queue is also increased compared to a queue on a single machine.
- Preservation of Order - Again, because the master - and only the master - maintains the logical queue, priority order is generally preserved in the majority of cases. Exceptions to this are discussed below.

2.2.2 Enqueue

The primary goal of the enqueue (and dequeue) algorithm is to ensure consistent state in the midst of failures and concurrent requests. Though the enqueue process involves talking to multiple nodes in the distributed system, the item is not inserted into the logical queue until all previous required operations are successfully completed. Thus, an error in any previous step has not affected the logical queue, and the queue is still in a consistent state. In other words, to a client of the queue, an enqueue operation is only complete when the client receives notification from the master that the data is now in the queue. The basic steps of the algorithm are as follows:

1. A client asks the master for a node to enqueue some data on and an ID for the data.
2. Master assigns a unique ID to this item and logs that this ID is "pending for enqueue".
3. Client stores the (ID, data) pair on the node designated by the master.
4. Client notifies the master of successful store.
5. Master notes the success, replicates the data, and enqueues the item ID on the logical queue, returning a "complete" to the node doing the enqueue.
6. If an error occurs or there is no success message from the client after some time out, the pending enqueue is discarded and the master instructs the data node to delete the data if it exists.

An important aspect of the above algorithm is that the actual data never passes through the master. Thus, the master can quickly assign unique ID's to requesting clients, those clients can transmit their data to different data nodes in parallel, and the master can quickly note the success when a client notifies the master of success. The replication is also lightweight for the master - the master merely instructs data nodes to replicate a given data item between themselves, again sparing the master from dealing with any actual queue data transfer.

Consideration was given to the approach of having the client transmit data to n nodes before reporting back to the master, where n is the replication threshold of the system. While this would work, it could result in decreased performance if the available bandwidth between data nodes is greater than the bandwidth from the client to the system. Such would be the case if

Blizzard was running on a cluster of machines in a given physical location, with remote clients using the queue.

2.2.3 Dequeue

Like the enqueue algorithm, the dequeue algorithm seeks to ensure consistent state in the face of a multi-step process. The queue should only complete a dequeue upon notification from the client that the item is now in their control. The basic steps for a dequeue are as follows:

1. Client asks master where to dequeue.
2. Master removes the head ID from the queue and puts it on a data structure holding pending dequeues.
3. Client requests the data associated with that ID from the node designated by the master.
4. Client notifies master of successful dequeue.
5. Master removes item from pending data structure and instructs appropriate data nodes to delete item data.
6. If the master is notified of a failed dequeue or is not notified of success after a time out, the pending dequeue is undone and it moves the data item back into the queue.

One of the goals we desired to meet was ensuring that a given item is not removed from the system until the master is notified by the client that the data was successfully retrieved. To accomplish this without blocking for the entire dequeue process, a "pending dequeue" is used to store item ID's that are currently being dequeued. Thus, if a dequeue fails (say the client crashes after requesting a dequeue and before pulling the data), the item can be re-inserted into the logical queue. This is one scenario that could corrupt to some degree FIFO ordering, as item A could be reinserted into the queue after item B had already been dequeued, even if originally A was enqueued before B.

2.2.4 Replication

The replication system is essentially a map of what data resides on which nodes at any given time, a heartbeat monitor, and triggers to initiate replication and update those maps appropriately. The purpose of the replication system is to ensure durability of data in the face of node failures. By replicating a given queue item on n nodes, the simultaneous failure of n-1 nodes can occur without losing any data in the logical queue. This replication threshold is tunable by the user of the queue depending on durability requirements. The replication system is run by the master, on the same node as the master. The steps of the replication system are as follows:

1. During an enqueue, the replication system marks what node is used to store the data item.
2. As part of the enqueue process, the replication system instructs n-1 additional nodes to grab the data from the initial node, where n is the replication threshold, keeping track of this metadata as well.
3. All nodes send regular heartbeat messages to the replication system, currently set at every 3 seconds.
4. If a node misses 3 heartbeat messages in a row (approximately 10 seconds), the replication system initiates replication of all of the data items on that node by issuing appropriate data copy instructions to data nodes still living. The actual queue item data is always transferred amongst the data nodes, never through the master.

2.2.5 Master Logging

To allow for the master to recover from failures, changes to the logical queue and to state information about the data nodes in the system are written to a log file before they take effect. During recovery of the master node, Blizzard reconstructs the last good state by applying the logged changes to the logical queue and other stored state.

3 Evaluation

We set up a Blizzard cluster on undergraduate lab machines with *N* data-nodes to qualitatively evaluate our original goals as *N* increases and the number of concurrent clients increase. While a measurement of throughput yielded expected results, the current implementation proved immature for measuring order perseverance and persistence. Specifically, even a simulation with high concurrency (hundreds of requests per seconds) and a very aggressive churn model (one in which a cluster of 57 data-nodes had multiple failures every minute) resulted in no observed data loss or change in ordering. However, beyond a certain threshold - a probability of failure of about 1% per second per machine or 100 concurrent clients - the entire system crashes. We think this is due to an exception in the RPC code that cascades through the system, but we did not have enough time to confirm or fix this bug. Therefore we discuss the theoretical properties of our implementation for order perseverance and persistence and the empirical properties for throughput.

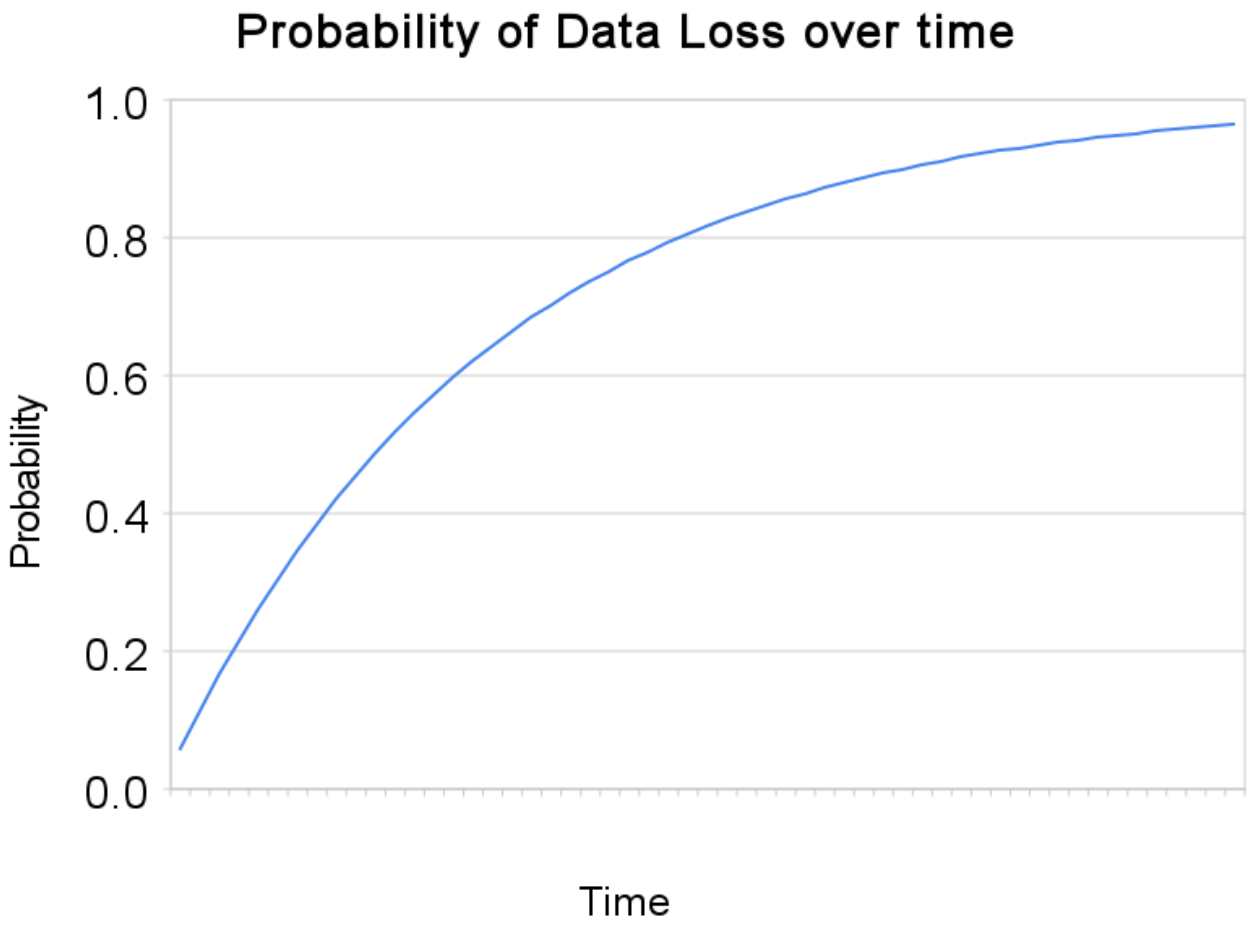
3.1 Order Perseverance

We define the observed order perseverance for a client to be the expected distance a client would have to move out-of-order dequeued values in order to achieve the original enqueue order of the values returned from Blizzard. To measure this we can assign a sequence number to each value enqueued. Since normal queues are FIFO data structures clients in general expect values dequeued early in time have lower sequence numbers than those returned later. Since nodes fail, and therefore dequeues are aborted, we expect some clients to observe imperfect ordering of dequeues. This can be modeled as a function of the average number of dequeue operations that succeed before a failed dequeue is aborted and added back to the head of the queue. Therefore, the order perseverance is a function of both the throughput of successful operations and the probability of a dequeue operation failing. In our experiments we observed no loss in ordering, however we were operating on very stable machines over a network with virtually no packet loss. We believe the expected value in a real world deployment would be small. If this is true, clients can obtain near-perfect ordering by buffering dequeued values.

3.2 Persistence

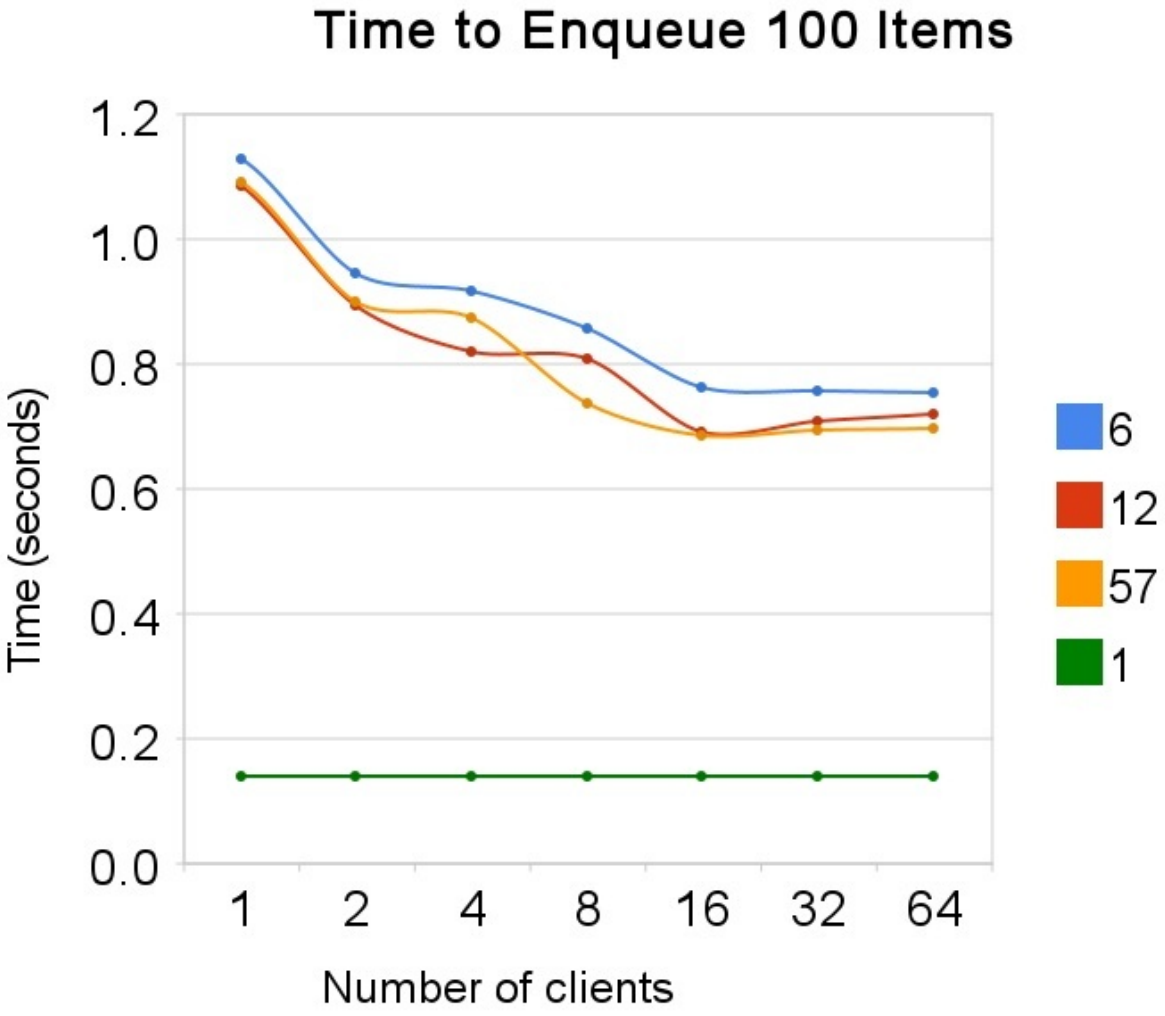
We model data persistence as a function of churn. For a particular value to be lost, all three nodes storing it must fail within the same 20 second time period. If at least one is alive at the end of the 20 seconds, the master will be able to re-replicate the value. Thus, the probability of losing a particular value within a certain number of 20 second periods is the probability of losing all three replicas during one of those periods. We model nodes as having a certain probability of failing every second. This description yields three assumptions. First, the probability of a certain machine failing each second is independent of the same in other seconds. Second, re-replication occurs immediately at the end of the 20 second period and therefore failures between time periods are independent. Finally, we assume machine failures are independent of other machine failures and therefore do not account for such things as network partition or rack power loss.

The graph below shows the shape of the theoretical CDF of a data loss occurring over time. If the probability of failure for a machine is 1% each second (i.e. expected lifetime is around 100 seconds), the probability of a losing a particular value after 7 hours is over 50%. If the probability of a failure for a machine is 0.01% each second (i.e. expected lifetime is just under 3 hours - still very low) the probability of losing a particular value after 2 days is under 0.002%.



3.3 Throughput

To measure throughput we varied both the number of data nodes from 6 to 57 and the number of clients operating concurrently from 1 to 64. Each client enqueued 100 values and dequeued 100 values. We calculated the average amount of time to perform 100 operations (enqueues or dequeues) from all clients. We compare this result with the same experiment on a basic single node queue implementing a similar remote interface. While the single node queue was faster by up to a factor of 7, scaling the system both by clients and data nodes increased performance significantly. At 57 data-nodes and more than 15 clients we observed times of around 0.65 seconds for 100 operations. This is about 150 operations per second which should be sufficient for many use cases. The figure below shows times for 100 enqueue operations as a function of the number of clients. Each line represents a different cluster size except the bottom flat line, which represents the single-node queue we used for comparison.


Time to Enqueue 100 Items
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Time (seconds)
1
2
4
8
16
32
64
Number of clients
6
12
57
1