A Scalable Relaxed Distributed Priority Queue

Bill S Lin
bslin@stanford.edu

Xiaohua (Victor) Liang
xiaohual@stanford.edu

Abstract

We have designed and implemented a scalable and persistent distributed priority queue system. The system provides probabilistic priority ordering while staying horizontally scalable for both read and write. This system can serve as a fundamental building block for many modern distributed systems, especially those with high traffic volume producer/consumer services. In this paper, we discuss the various aspects of our design and implementation, the simulation process to verify the probabilistic behavior of the system, and the evaluation of the system's performance in a distributed deployment.

1. Introduction

Modern distributed systems often consist of a large number of loosely coupled services that are deployed over multiple machines and commute through message passing networks. A common design pattern in such systems is that of distributed producer/consumer, where a large number of services produce a high volume of messages that are processed by another group of services. A scalable, highly available, and persistent message buffer that has the logical functionality of a priority queue is crucial to the proper function of these systems.

We have found that in many distributed systems that need priority queues on their critical paths, the requirement for scalability often outweighs the requirement for absolute priority ordering. An example would be a scheduling system responsible for taking in tasks from many users before scheduling them on many worker nodes whenever they are available. For such a system, a strict priority order would not offer much additional value than a slightly relaxed one, since a large number of items would be popped (scheduled onto) in quick succession by many workers. The scalable relaxed distributed priority queue we present here is aimed for such scenarios.

2. Related Work

The problem of scalable distributed priority queues deployed over a loosely coupled environment has been well studied. Many studies have tried to scale the system horizontally by trying to coordinate a group of symmetrical queue nodes each responsible for maintaining part of the priority queue. One common approach is to use a global root node to load balance and route read and write requests to a group of sub nodes that are responsible for storing the queue entries in sorted order [1]. While the root node is usually only responsible for light weight tasks in such design, it can still become the scalability bottleneck. A slight variant of this approach is using an external distributed coordination service such as Zookeeper in place of a root node [2], the scalability of which is bounded by the scalability of the coordination service instead.

Other designs try to address this problem by using node to node asynchronous message passing to reconstruct a partial view of global order in each node instead. In QCID, each node is responsible for maintaining part of a global graph that eventually points towards the global top item of the priority queue [3]. In SEAP, this partial view comes in the form of an aggregation tree [4].

Another approach popular in the industries is bucketizing the items into a fixed number of priority levels and having separate nodes to serve each level [5]. While this approach avoids the bottleneck of the root node and the need for peer to peer message passing, it is much less scalable for read than for writes, since all reads will go to the nodes serving the top bucket.

All the aforementioned designs aim to have a distributed priority queue that exhibits the exact same logical behavior as a heap on a single node. However, as explained before, in many distributed applications, such strict requirements are not necessary. The relaxed distributed priority queue presented here is inspired by the idea that a small number of choices made based on information from a limited number of nodes in a large randomized system can lead to drastically different results [6]. The original study focuses on various load balancing scenarios, in which examining as few as 2 random nodes before placement can significantly decrease the maximum possible load on any node with high probability. We found this result to be potentially applicable for the construction of a distributed priority queue.

3. Design

The systems proposed in this paper is a distributed priority queue that guarantees at-least-once semantics with relaxed priority ordering. It is composed of a cluster of nodes, with each one maintaining its own local priority queue, or subqueue.[1] The elements inside the priority queue are called **PQItems**. The PQItem is composed of a **PQKey** (containing a numeric priority and a unique uuid) and a byte array containing arbitrary user data.

The PQItems are sorted in the priority queue based on the numeric priority and the unique uuid, with the uuid used to break ties between equal priority items. Lower numeric value in the priority field represents higher logical priority. PQItems with higher priority are popped first.

The overall architecture of the system, along with server client interactions, are depicted in Figure 1. The following sections will explore each component in more details.

3.1. Server Node

Each server node is composed of a Kafka topic (for adding new elements and durably recovering after failover) as well as a REST API server that supports the following API calls:

- **void add(long priority, byte[] data)**: Add an PQItem to the priority queue. Clients can produce messages to a random node's Kafka topic directly (for performance), or hit the Rest API, which will add the item to a random node's Kafka topic.
- **PQItem peek()**: Returns the PQItem with the highest priority.
- **PQItem pop()**: Returns + Removes the PQItem with the highest priority.
- **void commit(PQKey key)**: Commits the PQItem, marking it as consumed. If the client does not call commit() after pop(), the PQItem will be placed back into the priority queue after a timeout.
- **void abort(PQKey key)**: Aborts the previously popped PQItem. This adds the PQItem back into the priority queue.

[1] Due to the one to one mapping of nodes and subqueues, we sometimes use them interchangeably in the subsequent sections.

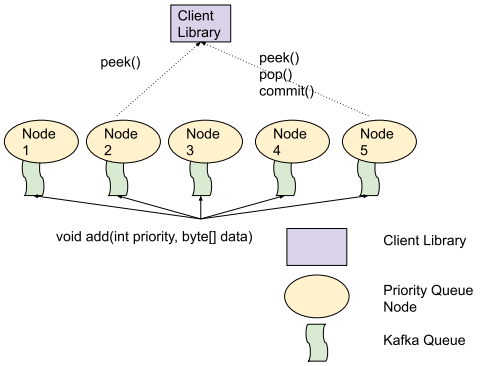

Figure 1. Architecture diagram of the distributed priority queue system. The client contacts one node for add, and multiple nodes for pop

3.1.1 Durable Snapshots

In order to maintain durability in case of failover, each Priority Queue Node performs periodic snapshots of all of its contents along with the current Kafka Offset at the start of the snapshot. The format of the snapshot is shown in Table 1.

Snapshots can be performed asynchronously by iterating through the entire priority queue data structure. When iterating, the priority iterator must return all the PQItems that existed when the iterator was constructed, but may or may not reflect subsequent inser-

⟨ Kafka Offset ⟩
⟨ PQItem 1 Priority Number ⟩
⟨ PQItem 1 UUID ⟩
⟨ PQItem 1 Data Size ⟩
⟨ PQItem 1 Data ⟩
⟨ PQItem 2 Priority Number ⟩
⟨ PQItem 2 UUID ⟩
⟨ PQItem 2 Data Size ⟩
⟨ PQItem 2 Data ⟩
...

Table 1. Server node snapshot layout

tions and removals to the priority queue after the iterator has been constructed.

Upon recovery, each node will load all the items from the most recent completed snapshot file. After this, it sets the Kafka Offset to the value indicated in the snapshot file.

This mechanism guarantees **at-least-once** semantics because any PQItem that has not been popped + committed since the start of the most recent snapshot will be in the snapshot file and all PQItems that are newly added since then will be re-consumed from Kafka. However, there may be some PQItems that are popped + committed after the snapshot started but are not reflected in the snapshot file. Clients might see these popped + committed elements a second time during failover, which is allowed by the at-least-once semantics.

3.2. Client

The client library provides a high level API to the system, which consists of **add()** and **pop()** operations. Underneath, the client library will call corresponding functions on a subset of the priority queue nodes.

When a client wants to add a new element to the priority queue, it will produce a PQItem to a random node's Kafka topic.

When a client wants to pop an element, it will peek at N different random nodes, and determine the priority queue with the highest priority PQItem. After the client determines the highest priority PQItem, it will then make a decision on which node to retrieve the next item to process based on the results from the peeks (the simplest policy being just pop from the node that returns the item with highest priority during the peek).The validity of this behavior is examined more extensively in Section 4. In order to guarantee at-least-once semantics, the client must then commit the PQItem to indicate that it is done with that item. Otherwise, the pop operation will be aborted after a timeout and the PQItem will be placed back into the queue.

4. Simulation

The validity of our design depends on how well the "peek n pop one" policy can approximate the behavior of a strict priority queue (one that always returns the top element), and thus before implementing the system, we performed a number of simulations to evaluate its correctness performance.

The main evaluation criteria used to measure the correctness performance of the system is the distribution of the **global index of the next element popped from the priority queue**. We will refer to this as the PEI (Popped Element Index). For simplicity, we measure the PEI distribution through aggregated statistics such as medium, average, or certain percentiles in our simulation experiments.

Note that since the priority queue is distributed into many different nodes, it is not possible to determine the PQItem with the global minimum priority across all nodes without querying every single node in the cluster. Therefore, the goal of this system is to approximate this as much as possible (the global index of the next element popped should be as close to 0 as possible).

4.1. System Variables

During simulation, it was determined that the following variables affect the PEI:

- **Number of priority queue nodes**: The more nodes, the more difficult it is to find the global highest priority element.
- **Consumer lag**: This is the number of elements remaining in the priority queue. If each client adds one element and then pops one element, this number remains constant for the simulation run.
- **Number to peek before pop**: This is the number of nodes to peek before determining from which node to pop.

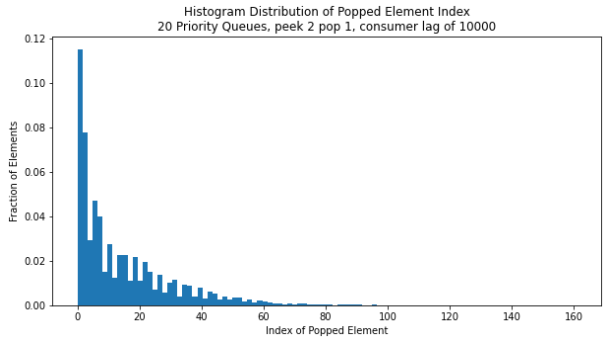

Figure 2. PEI distribution for a 20 node relaxed distributed priority queue

- **Number to peek before add**: This is an alternative idea we have explored during the simulation, in which we have the clients trying to choose the node at adding time instead of popping time.

Our subsequent simulation experiments are organized in a way to exam the effect on PEI distribution by each of these variables.

4.2. Simulation Result Analysis

We first present a basic simulation, in which one client adds 1 element (with priority being uniformly distributed) and then pops 1 element from the priority queue in a loop. In this setup, the system has 20 priority queue nodes. The client performs 2 peeks before popping the node with highest priority, and has a consumer lag of 10000. The purpose of this simulation is to demonstrate (and verify) that even a very rudimentary setup can have surprisingly good performance. The full PEI distribution of this simulation is presented in Figure 2 .From the figure, it is possible to see, for example, that the 0th index element popped first over 10% of the time.

For the next simulation, we examine the effect of consumer lag on PEI distribution. Figure 3 shows the average and median PEI as we vary the consumer lag in a 20 node setup. In all cases, it is possible to see that the PEI levelling out as the consumer lag increases, which means performance does not significantly degrade as there are more elements in the priority queue. Furthermore, it is possible to see that doing peek before pop performance much better than peek before add (i.e. adding new element to the peeked queue with the highest top priority). Because of this, in the rest

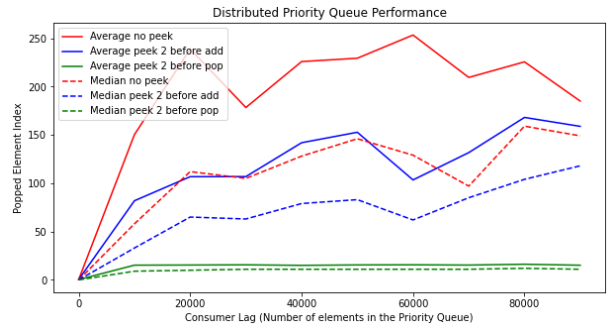

Figure 3. PEI(medium and average) variation as consumer lag increases under different client strategies. Notice the the much better performance of "peek before pop" over "peek before add", which itself is better than the "no peek" baseline.

of our experiments and the implementation, we only perform "peek before pop".

Also shown in Figure 3, under the "peek before pop" strategy, once we have a large number of elements with the distributed queue system, the PEI distribution becomes independent of the total number of elements.

Empirically, we found that under large enough consumer lag, the nth percentile of PEI values generally follows equation 1, where $num_{subqueue}$ is the total number of nodes/subqueues within the system, num_{peek} is the total number of subqueues to query before the final pop, and f_n is a function of num_{peek} for the corresponding n^{th} percentile value.

$$PEI_{n^{th}percentile} = num_{subqueue} * f_n(num_{peek}) \quad (1)$$

The scaling of PEI with $num_{subqueue}$ while holding num_{peek} constant can be seen in Figure 4, which shows a remarkably linear scaling, especially considering the randomness introduced in the simulation.

To illustrate the relation of PEI and num_{peek}, we ran another simulation with an 100 node setup. The results are shown in Figure 5. The figure can also be seen as a plot of $f_n(num_{peek})$. As demonstrated in the figure, checking even a very small number of nodes makes it possible to achieve a reasonably low PEI even at 90th percentile.

4.3. The Issue of Concurrent Operations

The simulations above assume that the n peeks and one pop which constitute one complete client side pop

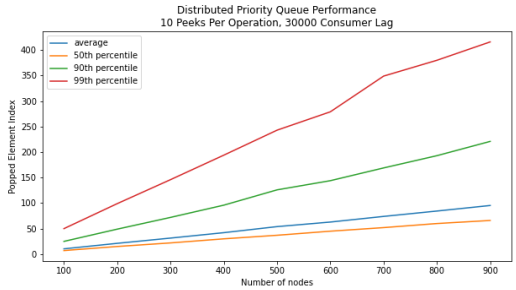

Figure 4. PEI variation as number of nodes/subqueues increases. The number of peeks before popping is kept at constant, and consumer lag is kept at a large number.

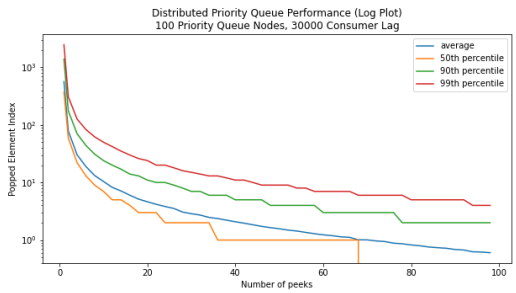

Figure 5. PEI variation as number of peeks before the pop increases. The total number of queues are kept constant, and consumer lag is kept at a large number.

operation are done on some global atomic snapshot of all the subqueues involved. More specifically, it assumes that if a certain element is returned upon the peek on a particular subqueue, the same element would be returned upon the pop on the same subqueue. However, this assumption is generally not true when we have a large number of concurrent operations from multiple clients. Imagine when client A and client B both peek at subqueue Q, and then both decide to pop from Q, only one of the clients is going to get the element it expects (i.e. no longer the top element from Q). This would degrade the correctness of the system (in terms of PEI distribution).

We perform the similar simulation as before with an 100 node set up, but allowing the peeks and pops from different clients to interleave. The results are shown in Figure 6. It can be seen that the system's correctness performance degrades as the number of concurrent requests increases The effect is way more significant on the 99th and 90th percentile of PEI, while much less so on medium and average PEI.

There are a few potential methods to address this

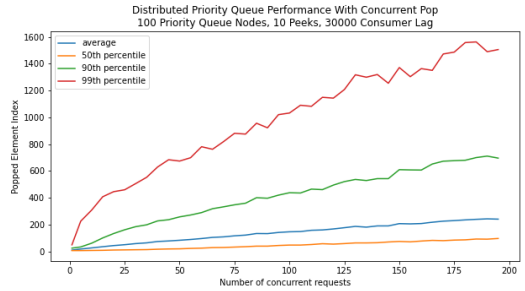

Figure 6. PEI variation as the number of concurrent request increases. In this simulation, the client peeks 10 queues before popping from the queue that returns the highest priority upon the peek.

problem.

1. We can consider hiding the peeked elements by one client from other clients during the time of its operation. This would ensure that the popped element is the same as the one being peeked at, but would also mean that the peeked element itself might not be the top element in the subqueue, and thus the correctness degradation shown in Figure 6 would still persist.
2. We can consider completely locking the subqueues involved in the operation of one client from other clients. This eliminates the concurrency problem, but would severely limit the overall throughput of the system, and thus undermining the main advantage of a distributed priority queue.
3. Upon the popping from the chosen subqueue, check whether the item is expected, abort the current pop if not so, and retry another peek and pop loop. This approach avoids the problems of the above two, but is not guaranteed to terminate when concurrency levels are high.

The approach that we adopt, as shown below, is a slight variation of approach 3.

(a) Peek n subqueues and try to pop from the one that returns highest priority element.

(b) If the actual popped element's priority is not lower than the second highest priority item retrieved during the peek, commit the pop and return the element.

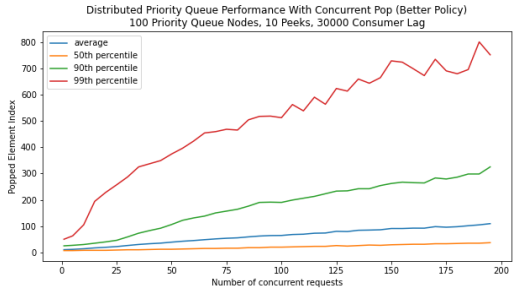

Figure 7. PEI variation as the number of concurrent request increases, with more sophisticated client side policy.

(c) Otherwise pop and commit from the node that returns the second highest priority item during the peek.

Figure 7 shows the same simulation set up as in Figure 6, but with this new policy. It can be seen that the PEI is cut in half for all percentile levels, at the potential cost of just one extra pop.

5. Implementation

The design described in Section 3 was implemented in Java as a Spring Rest service. The APIs were modeled as Rest endpoints, and there is also a background thread in each Priority Queue node that consumes from a dedicated Kafka Topic. The implementation is available at `https://github.com/bslin/distributed-pq/tree/master/priorityqueue`

6. Deployment & Evaluation

6.1. Pseudo-Distributed Deployment

The priority queue is first deployed and tested in a 2 host setup. A 2016 MacBookPro and a 2015 MacBookPro are connected over a 5G Wifi network. The 2016 MacbookPro is running a number of different Priority Queue Nodes in different processes. The 2015 MacbookPro spawns a number of client processes, with each one popping from the cluster of priority queue nodes.

The distributed priority queue cluster starts with 1000 elements (with priorities 0 through 999) and the test is run until the client cannot find an element to pop after peeking from 2 nodes.

Since all the clients are running on the same host, it is possible, using system timestamps, to determine

Figure 8. Distributed priority queue performance with pseudo-distributed deployment

the order these items are popped, which can then be used to calculate PEI. In this test, we test how PEI changes as we vary the number of concurrent clients and the number of priority queue nodes. The figure below summarizes the results for 3, 10, and 30 priority queue node setup. The number of clients is varied (via powers of 2) from 1 to 128.

From Figure 8, it is possible to see that as the number of queues increase and as the number of concurrent clients increases, the PEI also increases proportionately. While this seemingly degrades performance as we scale, we should note that as the number of clients/server nodes increase, there will be more elements processed (throughput), which means that it is not as bad if a higher logical priority element gets popped earlier.

6.2. Distributed Deployment on AWS

We have also deployed the system on a fleet of virtual machines on Amazon Web Service to evaluate its latency performance in a truly distributed setting.

We set up a cluster of five EC2 t3.xlarge VMs (4 vCPU, 16 GiB memory each) within the same availability zone and connected them to the same subnet to host the system. Each VM is responsible for hosting one subqueue of the distributed priority queue system, including both the Java Spring server and the corresponding single partition Kafka topic. The Zookeeper ring for the Kafka brokers is hosted as a single node setup on one of the five VMs, since it is not likely to be a bottleneck of the system. For simplicity, we did not set up a dedicated load balancer for the system. Instead, the subnet IPs of the 5 VMs are hardcoded in a configuration from which clients can randomly

choose.

The clients ran in other VMs within the same subnet, using a basic strategy of peeking 2 nodes before deciding where to pop from. We were able to run 4 concurrent clients from a single VM, each with a tight pop()-commit() loop. They reached a total qps of 340 without any errors, with a query latency of 12ms/13ms/17ms (medium / 90th percentile / 99th percentile). Further increase in number of concurrent clients on a single VM do not further increase the total qps, likely due to network constraints on a single VM limiting the number of requests in flight. When clients are deployed onto multiple VMs, we can achieve a total qps of over 600 qps without noticeable degradation of latency.

7. Conclusion

This paper described the design, implementation and deployment of a scalable and persistent relaxed distributed priority queue. It demonstrated how a simple but elegant idea of random choices can be employed to build a system that provides sufficiently correct behavior for many applications while avoiding the bottlenecks and complexities of more sophisticated systems. The extensive simulation presented in this paper provides the necessary guidance for parameter tuning should such a system is deployed in production, while the prototype implementation proves the feasibility of the design.

References

[1] B. Mans, "Portable distributed priority queues with mpi," *Concurrency, practice and experience*, vol. 10, no. 3, pp. 175–198, 1998.

[2] The Apache Software Foundation, "Distributed priority queue," Available at https://curator.apache.org/curator-recipes/distributed-priority-queue.html (2020/05/20).

[3] R. Bajpai, K. K. Dhara, and V. Krishnaswamy, "Qpid: A distributed priority queue with item locality," in *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2008, pp. 215–223.

[4] M. Feldmann and C. Scheideler, "Skeap & seap: scalable distributed priority queues for constant and arbitrary priorities," in *SPAA 19: The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, 2019, pp. 287–296.

[5] Microsoft Corporation, "Priority queue pattern," Available at https://docs.microsoft.com/en-us/azure/architecture/patterns/priority-queue (2020/05/20).

[6] M. Mitzenmacher, A. Richa, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Handbook of Randomized Computing*, vol. 1, pp. 255–312, 2001.