A Flexible QoS Fortified Distributed Key-Value Storage System for the Cloud

Tonglin Li[1], Ke Wang[3], Shiva Srivastava[1], Dongfang Zhao[1], Kan Qiao[4], Iman Sadooghi[1], Xiaobing Zhou[5], Ioan Raicu[1,2]

[1]*Illinois Institute of Technology,* [2]*Argonne National Laboratory,* [3]*Intel,* [4]*Google,* [5]*Hortonworks*

***Abstract*—In the era of Big Data and Cloud, distributed key-value stores are increasingly used as building blocks of large scale applications. Comparing to traditional relational databases, key-value stores are particularly compelling due to their low latency and excellent scalability. Many big companies, such as Facebook and Amazon, run multiple different applications and services on top of a single key-value store deployment to reduce the deployment and maintenance complexity as well as economic cost. However every application has its own performance requirement but most of current key-value store systems are designed to serve every application request equally. This design works well when a single application accesses the key-value store, but it is not as good for the emerging concurrent multi-application scenario. In this paper we present ZHT/Q, a flexible QoS (Quality of Service) fortified distributed key-value storage system for clouds and data centers. It improves the overall throughput by an order of magnitude and still satisfies different applications' latency requirements with QoS by means of dynamic and adaptive request batching mechanisms. The experiment results show that our new system delivers up to 28 times higher throughput than the base solution while more than 99% of requests' latency requirements are satisfied.**

Keywords*—*Distributed key-value store, QoS, NoSQL database, Request batching, Distributed systems.

I. Introduction

Distributed key-value stores are known for their ease of use and attractive performance. Major technology companies such as Facebook, Amazon and Google have built their data infrastructure with key-value stores to accommodate their fast growing businesses. Due to the complexity of system design, deployment and maintenance, along with the running cost, more and more companies choose to share a single key-value storage system between different applications and services. Take Facebook as an example, workloads from user accounts information, web-app object metadata and system data on service location, etc., run on one Memcached deployment [1]. Most of applications have their unique performance requirements. Some applications may prefer lowest latency, some prefer high total throughput, while others may like to have a well-balanced performance profile. These potentially conflicting requirements can be very different from the design goals of conventional key-value stores, which mostly focus on low-latency. How to choose a good solution that meets many applications' needs is still an open question. The choice is even not obvious for latency – one of the most commonly used metrics. Different applications can tolerate very different latency ranges. For example, a shopping cart application can satisfy customers with 50 ms latency; instant messaging users are fine with 500 ms while a metadata service for databases or file systems requires as low as possible latency, ideally no longer than 5 ms [2]. Giving all applications same efforts and optimizing on the same aspect (single request latency) is not necessarily appropriate, as sometimes it might harm the total throughput provision of the system, and lower the resource utilization. This is especially true when key-value stores are delivered as cloud services that need to serve many different applications and users [3].

In this paper we present ZHT/Q, a flexible QoS fortified distributed key-value storage system for clouds. It enhances a high performance key-value store with flexible QoS (Quality of Service) properties such that both configurable latency and high aggregated throughput can be achieved. It satisfies different applications' latency requirements with QoS while improves the overall performance through dynamic and adaptive request batching mechanisms. The system QoS provides guaranteed and best-effort service on latency for different scenarios. It also watches the performance change and dynamically adjusts the batching strategy to alleviate performance degradation upon traffic.

The contributions of this paper include:

- We design and implement a flexible QoS fortified distributed key-value storage system on top of our previous plain key-value store [4]. The new system is optimized to satisfy QoS on latency while achieving high throughput;
- Our system supports different QoS latency on a single deployment for multiple concurrent applications, both guaranteed and best-effort services are provided;
- Extensive performance evaluation is conducted through both real system micro benchmarks (16 nodes) and simulations (512 nodes), and the comparisons show the advantages and limitations of this design.

II. Design and implementation

In this section we firstly describe the challenges and considerations in our design. Then, we present the design and implementation of the system. Finally, we analyze and model the performance.

A. Challenges and design considerations

1) Configurable QoS on performance: Different applications have different performance requirements, some times even one application can have different requirements when facing different scenarios. The first and most important question we face when designing the system is how to support user-configurable QoS. For storage systems, there are many ways to deliver different performance levels. Amazon EC2 and Google Compute Engine use different hardware resource (such as SSD v.s. HDD) and network bandwidth to offer different performance; some use software-defined network to manage performance [5], some uses different consistency models in storage replication to provide different response time [3]. Many of these solutions depend on special hardware and leave users few choice. We decide to use a pure software solution, request batching, so as to avoid hardware dependency.

2) Batching strategy: Request batching is not a new method to achieve better system efficiency. By aggregating individual requests, a batching system can reduce the total number of messages and amortize service overhead. The key question in request batching is when to send the batch. The situation is simple when there is no time limit for request delivery (latency), within network bandwidth limits, bigger batches generally bring better throughput and efficiency. If there is an inviolate request latency limit, the system designer has to give the latency limit a higher priority over the system efficiency and throughput. Various latency limits, which are usually associated with different applications, make the situation even more complicated. With this consideration, the design goal is to provide as high as possible throughput without violating the request latency limits. Dynamic request batching is a real-time scheduling problem [6]. Some theoretical works [7] have been done on various aspects of request batching. We use a modified Earliest-Deadline First (EDF) [8] algorithm in our dynamic batching mechanism.

3) Dynamic system performance tuning: In dynamic environments such as clouds and data centers, network and server workload may fluctuate all the time and impact the system performance. When the network and servers are heavily loaded, to keep trying sending more requests could make the situation worse. Therefore our system needs to be smart enough to adjust the request sending rate according to the network traffic, which requires the network/server traffic detection. Since the dynamic nature makes it difficult to predict a request latency, we design a history-based heuristic approach to detect the traffic and to tune system parameters.

B. System design

We design the new system based on our previous work, ZHT [4, 9], a zero-hop distributed key-value store. ZHT follows a Memcached-like network architecture, in which servers are organized in a logical ring and each houses a contiguous key space. Clients have the knowledge of all servers and can send requests to servers by hashing the given keys. As we observed in [4] and [9], when a client is sending requests in very high rate (e.g. in a tight loop), the bottleneck is actually on the client side and is bounded by the ability to handle socket connections, which is limited by kernel and CPU performance [10]. Thus in most of scenarios the servers and network are not saturated. Additionally because the client-server communication dominates round-trip latency, it would be desired to reduce message number between clients and servers.

With these consideration in mind, we propose to add a proxy layer for dynamic batching mechanism on the client side instead of server side. The client proxy works on each client, collects and batches the requests that share a same destination server and sends to the server. The destination server unpacks the batch with a parser, executes the requests sequentially,

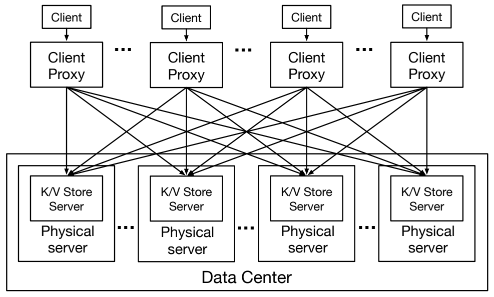


Fig. 1: Requests are batched on client side in a proxy, which controls how and when to send a batch to a server. The servers parse batch and execute the requests sequentially, and then send batched responses back to the client.

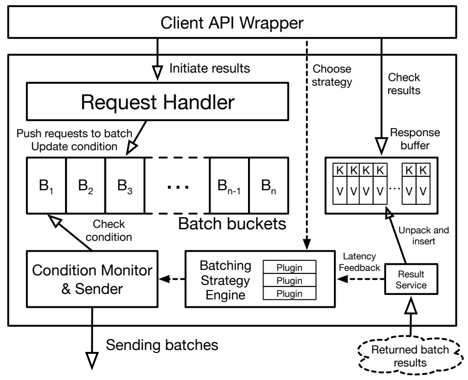


Fig. 2: Client proxy has a list of buckets (B_1 to B_n), each of which is dedicated to one server. A monitor thread checks and sends a batch if the sending condition is satisfied. Returned response batch is unpacked and stored in a local key-value map in client proxy, the client will be notified upon the map change.

packs the return status (including lookup results) in a batch and send back. This keeps the communication and storage layers of architecture of key-value store unchanged.

1) Client Proxy: The client proxy architecture is shown in Fig.2. The client proxy API wrapper offers the applications a set of interfaces that are compatible with conventional APIs (`put`/`get`). At the same time it also provides advanced controls to specify the working mode and QoS level so as to fine tune the performance.

The client proxy maintains a list of batch buckets, each of which is associated to a destination server. When a request is submitted in *single* mode, the client proxy sends it directly to the server like most of key-value stores do. If the request mode is *batch*, the request handler pushes it to the batch bucket that is associated to the corresponding destination server. Then it updates the condition variables of the batch bucket according to the new request. A batch monitor checks the condition variable for each batch bucket and decide to send it or not. Apparently the value of the condition variable is the key to the batching system behavior. It is calculated through different batching policies (Alg.1), which work as plugins in the client proxy. We implement multiple batching policies and discuss them in Section II-C.

When a server receives a batch, it sequentially executes the requests and pack the results into a new batch and then send back to the client. Note that any request batch and response batch only involves one client/server pair.

Algorithm 1 Batch requests handling

```
1: procedure THREAD_REQUESTHANDLER
2:     repeat
3:         $batch \leftarrow batch.addToBatch(newReq)$
4:         $deadline_{new} \leftarrow time_{newArrv} + QoS_{new}$
5:         $num\_req_{batch} \leftarrow num\_req_{batch} + 1$
6:         $size\_byte_{batch} \leftarrow size\_byte_{batch} + size_{new}$
7:         batch.mutex_lock()
8:         if $deadline_{batch} \geq deadline_{new}$ then
9:             $deadline_{batch} \leftarrow deadline_{new}$
10:        end if
11:        batch.mutex_unlock()
12:    until Terminated
13: end procedure
14:
15: procedure THREAD_MONITOR
16:    repeat
17:        for all batch in BatchList do
18:            batch.mutex_lock()
19:            if $condition_{send}(policy) = True$ then
20:                sendBatch()
21:                $deadline_{batch} \leftarrow \infty$
22:                $batch.requests \leftarrow \phi$
23:            end if
24:            batch.mutex_unlock()
25:        end for
26:    until Terminated
27: end procedure
```

TABLE I: Batch request data structure

Variable Name	Description
Client_IP	For server returning results
Client_Port	Client listening port
Dest_Server	Destination server
Curr_Deadline	Current batch deadline; Condition var
Batch_num_item	# of requests; Condition var
Batch_num_limit	limit of requests #; given threshold
Batch_Size_Byte	size in bytes; Condition var
Batch_Size_limit	limit of size in bytes; given threshold
Data_Requests	List of single requests

2) Client APIs: Most key-value stores' client APIs work in a synchronous manner, in which clients are blocked while waiting for the servers' response. To minimize the application change, ZHT/Q supports both synchronous and asynchronous APIs. Under the hood, in ZHT/Q's client proxy, requests are handled in non-blocking and asynchronous manner. All batch mode responses from servers are stored in a response buffer, which is a local in-memory hash map in the proxy (Fig.2). In this buffer map, keys and values are request keys and server responses respectively. For applications that require asynchronous access, they simply check the hash map at will, for example, to check after the QoS latency time. For synchronous applications, a dedicated thread is created in the client proxy, which blocks the application and checks the hash map within the specified QoS latency time. The application will return until response is found or a given time-out is reached.

C. Batching strategies

A batching strategy is represented by a multi-parameter trigger condition, based on which the client proxy (batcher) decides when to send a batch. ZHT/Q provides several important strategies, each of them can work in two modes, **static** and **dynamic**. In static mode, the client proxy uses policies with user specified or system initialized parameters and thus will not change. In dynamic mode, the client proxy works with the same initial parameter set but it dynamically adjusts parameters based on currently measured performance so as to provide best-effort service when network or servers are heavily loaded (Alg.3).

Some applications do not have explicit QoS requirement. This type of requests can be sent in static mode with fixed batch size if the user specifies, which can fully utilize the advantage of batching. If users do not specify, these requests will be handled along with the QoS enforced requests by the system automatically with a modified Earliest-Deadline First (EDF) strategy (Alg.2). This strategy considers all three major parameters, namely batch latency limit (deadline), logical batch size (number of requests in a batch) and physical batch size in bytes. The batch deadline is calculated dynamically based on arrival time and QoS of every single request. The threshold values for the other two variables are given by the system administrator. A batch will be sent as soon as any one out of three conditions (batch deadline, logical batch size and physical batch size) are satisfied. In other words, the deadline for a batch is the closest deadline of all requests in that batch. EDF strategy works well to satisfy various QoS requirements. However it has a potential problem when the QoS range is very wide. The requests that have smaller QoS latency value can prevent the batching mechanism from accumulating many requests, because the system has to send batches more frequently to satisfy the smaller QoS latency.

Algorithm 2 Earliest-Deadline First Batching

```
1: procedure $condition_{send}$
2:     if $dl_{batch} \leq sys_{delay} + time_{cur} || num\_req_{batch} \geq max\_req || size\_byte_{batch} \geq max\_size_{batch}$
3: return True
4:     else
5:     $deadline_{batch} \leftarrow \infty$
6:     $batch.requests \leftarrow \phi$
7: return False
8:     endif
9: end procedure
```

D. QoS properties

In a request batching system, a potential problem is that a request could wait in the batch queue for a long period of time if the sending condition is not met. This could happen when the condition is not properly set or the request arrival rate is low. We avoid this problem by fortifying all batch mode requests with a maximal tolerable latency, called *QoS latency*, which can be defined in QoS or SLA and the system is supposed to return results before that. Requests in single mode require as low as possible latency, thus they are always served immediately with the lowest possible latency (best-effort service) and no explicit QoS definition needed. ZHT/Q offers a guaranteed service when the network has no congestion and a best-effort service when network or servers are busy.

1) Guaranteed service: When the network and servers' processing capability are not saturated, the QoS on latency is guaranteed. Assuming a request i has a different maximal tolerable latency, denoted as l_qos_i, the request is submitted at time Tc_i, then there is a deadline d_i presented in POSIX time for the request. To ensure that the QoS of all requests in a batch are satisfied, we define the deadline of a batch $d_B$to be the closest deadline to present (T_{now}) in the batch. A given sys_cost is a threshold that is greater than the possible round-trip transferring time plus server side execution. As long as the batch is sent before $d_B + sys\_cost$, the QoS of all requests are satisfied. Then, we have the lowest condition (Formula 1) to decide when to send a batch while keeping QoS.

$$\begin{aligned} d_i &= Tc_i + l\_qos_i, \\ d_B &= \max_{i=1}^{n} d_i, \\ d_B &\leqslant T_{now} + sys\_cost \end{aligned} \tag{1}$$

2) Best-effort service with feedback based adjustment: When the network or servers are heavily loaded, the client side measured performance can degrade significantly. ZHT/Q uses passive latency detection to adjust batching parameters so as to adapt to the dynamic network environment (Alg.3). Latency is measured on clients for each request and compare it with an threshold value to judge if the system is running normally. The threshold latency for single and batch request mode is set in different ways. For single mode, it looks straightforward: just set to be slightly shorter than the QoS latency. However this can cause a serious problem. On one hand, because of the delay between measured latency-based adjustment and measurable effects, and the presence of network noise, if simply using the latest measured latency as as the batching adjustment condition, the randomness of latency could lead the system to jitter. On the other hand, since individual requests are generally very small, the latency could fluctuate drastically due to the influence from client/server CPU utilization and network noise. Thus the request latency would has larger standard deviation, especially in dynamic environments such as clouds. This means there could always be a tiny part of requests are responded after the given threshold. This makes it very difficult to give a valid expected latency for threshold. To avoid this problem, we use a weighted arithmetic mean (Formula 2b) instead of the actually measured latency, in which newer recorded latencies have higher weight. In this way, newer measured latency always play more important roles while the older latency can be used to balance the jitter.

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{1}{2^n} = 1$$

$$\bar{L_n} = \sum_{i=1}^{n} \frac{L_i}{2^n} \tag{2a}$$

$$\begin{aligned} \bar{L_{n+1}} &= L_1/2 + 1/2 \sum_{i=2}^{n} \frac{L_i}{2^n} \\ &= (L_1 + \bar{L_n})/2 \end{aligned} \tag{2b}$$

Note that L_1 is the latest latency, L_n is the oldest latency recorded and $\bar{L_n}$ is the weighted average latency for past n requests. When n is reasonablely big, the error is negligible ($\bar{L_n}/2^n$).

In batch mode, since there is no QoS latency for batches, and the time to send a batch can not be determined before it meet the predefined sending condition, it is hard to give a reasonable threshold based on given QoS. However we can still find if a batch is delayed. When a batch reaches the condition for sending, its physical size and logical size are known. With these sizes, we can calculate a expected batch latency by Formula 3 where α_{tr} is the constant transferring factor, $S_{req+res}$ is total data size to transfer, n is the number of requests in a batch, and C_{exe} is the time of executing a request.

Algorithm 3 Dynamic parameter tuning

```
 1: function ParameterTuner($L, size_n, size_b, sys\_delay$)
 2:     if $L > ExpectedLatency()$ then
 3:         $size_n \leftarrow size_n/2$
 4:         $size_b \leftarrow size_b/2$
 5:         $sys\_delay \leftarrow 2 \times sys\_delay$
 6:     end if
 7: end function
 8: function ExpectedLatency($L_{cur}, \bar{L}$)
 9:     if $isIndividualRequest$ then
10:         $ExpectedLatency \leftarrow WeightedAvgLatency()$
11:     else
12:         $ExpectedLatency \leftarrow ExpectedBatchLatency()$
13:     end if
14: return ExpectedLatency
15: end function
16: function WeightedAvgLatency($L_{cur}, \bar{L}$)
17:     if $\bar{L} = 0$ then
18:         $\bar{L} \leftarrow L_{cur}$
19:     else
20:         $\bar{L} \leftarrow (L_{cur} + \bar{L})/2$
21:     end ifreturn $\bar{L}$
22: end function
```

$$L_{exp} = \alpha_{tr} \times S_{req+res} + n \times C_{exe} + C \tag{3}$$

From Formula 3 we can see that the batch response time is a linear function of two variables: request number (n) and total data size to transfer ($S_{req+res}$, including requests to and response from a server). Since the profiles from different network environment are different, the parameters need to be determined for all ZHT/Q deployments. Running a set of test requests with different sizes at the initial stage and measure the latency, we can calculate these parameters through linear regression.

When the measured latency (from send batch to response received, Fig.3) is longer than the expected duration, ZHT/Q will switch to best-effort service mode to ensure the QoS time l_qos_i is met if at all possible. In this mode, a compensatory mechanism is triggered to tune the current batching strategy to reduce latency. The predefined system cost (sys_delay) will be increased such that batches are sent more frequently. Since reducing requests in batches will benefit latency, this attempt is to sacrifice throughput for latency.

III. Performance Evaluation

In this section we evaluate the performance of our system with different batching policies and various workloads through micro benchmarks. We run real system micro benchmarks on Amazon EC2 with moderate scales (up to 32 instances) and

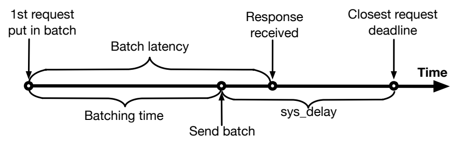


Fig. 3: Batching events and time composition. sys_delay is the reserved time for batch transferring and can be automatically changed by the system to adapt traffic.

simulations on large scales (up to 512 nodes) to measure the performance.

A. Workloads

For better coverage of different application scenarios, we define 3 types of workloads with various requirements. In all three workloads, requests are sent from clients in tight loops. Like Facebook [1] and MICA's [10] workloads, we focus on small requests with fixed key (10 bytes) and value length (20 bytes), 95% `get` and 5% `put`.

Workload with no explicit latency QoS: In this type of workload, application requests are relatively less latency-sensitive and well tolerant of a wide range of response latencies. This covers a category of applications that do not have a explicitly specified response time limit, such as logging and archiving systems. For this scenario, logical batch size (the number of requests in a batch) is the only parameter, meaning that setting a limit for batch size would work for most of cases.

Workload with single QoS latency: In this workload, key-value requests in each experiment are given same QoS latency. Assuming one application only has one QoS setting, this workload represents the scenario in which one application with many clients is served by the data store. In the test data set, the QoS time varies from 1 ms to 1000 ms.

Workload with multiple QoS latency: This workload simulates the scenario that multiple different applications use a single deployment of key-value store as a service. Note that applications in this case have various QoS latency time, ranging from 1ms to 1000 ms. The workload is organized as shown in table II. Requests in workloads of pattern 1, 3 and 4 have more even QoS distribution, while pattern 2 represents a highly skewed workload.

TABLE II: Workload with multiple QoS

QoS latency time	1ms	10ms	100ms	1000ms
Pattern 1	25%	25%	25%	25%
Pattern 2	4%	32%	32%	32%
Pattern 3	0%	33%	33%	33%
Pattern 4	0%	0%	50%	50%

B. Experiment setup and metrics

For a detailed performance study, we conduct a real system micro benchmark on Amazon EC2 with 2 to 32 C3.large instances, half as servers, half as clients. We separate servers from clients to avoid any local communication. For better understanding the performance and scalability on large scale deployments, we construct a PeerSim [11] based simulator. We use the data captured from real system to calibrate and validate our simulation results (Section III-F).

We focus on 3 metrics that accurately reflect batching performance, namely individual request latency, batch latency and node throughput. Request latency presents the duration from a request is submitted to the response is returned by a server. Batch request presents the duration from the first

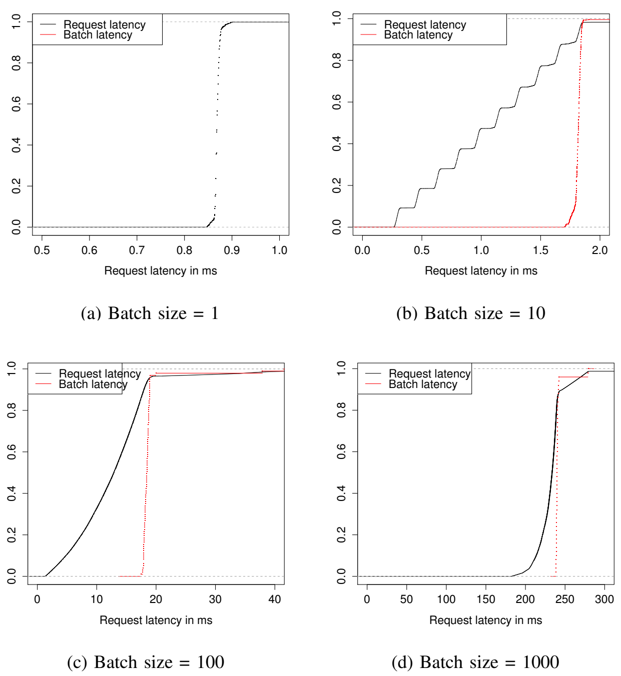


Fig. 4: Batching with fixed batch size. Expected longer batch latency can be observed in experiments wirh larger size (Fig.4c and fig.4d). Note that batch latency is proportional with batch size.

request enter the batch, to the batch response is returned by a server. node throughput presents the number of finished requests in one second by the system. Please note that in individual request mode, low latency directly means high throughput. But in batch mode, it is a different story. Low average request latency imply a smaller batch size, which is not likely to make good use of the system resource. On the contrary, longer average request latency (while still within QoS) usually implies high throughput, due to the system has longer time to accumulate requests.

C. Applications with no explicit QoS

There are also many applications that have a response time expectation but do not have QoS options in their APIs, we set a maximal time limit 1 second during the experiment, by then a batch will be sent even if the batch size has not reached the threshold. As expected, the throughput (Fig.5) increases with the batch size. However it is worth to note that the throughput increasing rate is much slower than that of batch size. This is because the batching cost and the time for waiting requests are accumulated during batching. When the batch size is n, it takes $n*(t_c+t_{cost})$ time to wait and to push all the n requests into the batch, where t_c is the interval between 2 contiguously arrived requests, and t_{cost} is the time cost for processing a request in the batch. This implies a linear batching cost with logical batch size. Under this workload (Fig.4), as expected, the batch latency is significantly longer with larger batch size (Fig.4c and Fig.4d).

On throughput (Fig.5), unsurprisingly bigger batch sizes bring higher single node throughput, but the increment is not proportional to the batch size due to the accumulated batching overhead and increased data transferring cost. On different scales, the single node throughput stays consistent, which indicates excellent scalability.

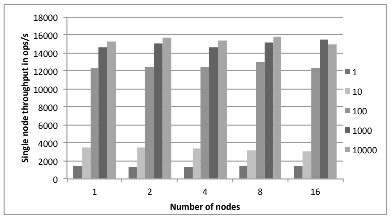
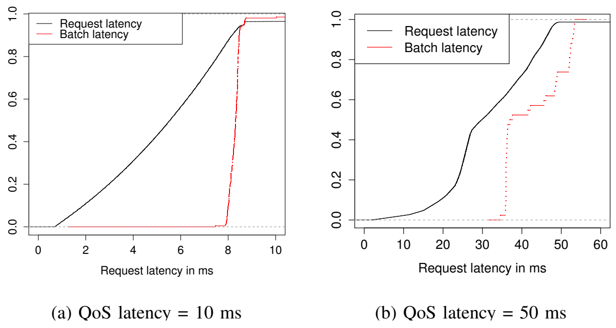
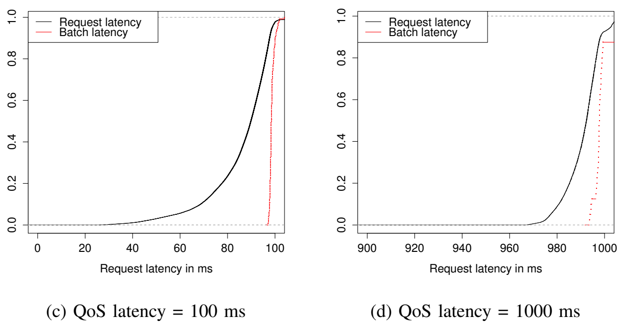


Fig. 5: Throughput with fixed batch size. Bigger batch sizes bring higher equivalent throughput, but the increment is not linear due to the accumulated batching overhead and increased data transferring cost. Corresponding latency distribution is shown in Fig.4.



(a) QoS latency = 10 ms

(b) QoS latency = 50 ms



(c) QoS latency = 100 ms

(d) QoS latency = 1000 ms

Fig. 6: Workload with single QoS latency represents single-application scenarios. Higher batch latency (red line) is desired because it can accumulate more requests and yield higher throughput (Fig.7). Batch latency is proportional to the QoS, and is close to the single request QoS, implying that the system and network are still far from being saturated.

D. Single application with latency QoS

Typical applications with QoS requirement usually specify only one QoS value. This experiment represents the use case that multiple clients of single application access the data store. We can see that more than 99% requests are satisfied within the QoS time (Fig.6), except for the workload with very long QoS latency, in which 95% requests are satisfied. The throughput increases with the QoS time (Fig.7). Due to the longer QoS time, each batch can accumulate more requests before sending, which means larger batch size and throughput. This is also the reason why the throughput shows similar pattern with fixed size batching (Fig.5).

In Fig.6 we see that the measured batch latency is proportional to the specified QoS latency. Note that higher batch latency (red line) is desired because it can accumulate more requests and yield higher throughput (Fig.7). This also implies that if measured request latency is much shorter than QoS, it causes waste to the system efficiency. Thus a "lazy" but good enough (just to satisfy QoS) batching strategy is welcomed. On throughput (Fig.7), similar to the trends shown in static batching (Fig.5), longer QoS brings higher throughput.

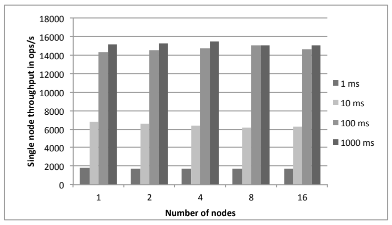


Fig. 7: Throughput: batching with static QoS latency. Longer QoS latency requests can wait longer in the batch, which allows the system to accumulate more request thus higher throughput. Corresponding latency distribution is shown in Fig.6.

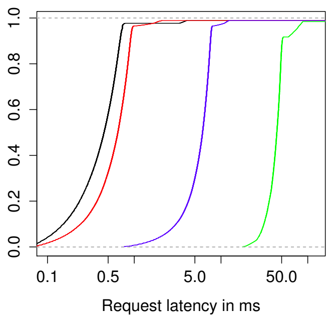


Fig. 8: Batch latency CDF for different workloads. From left to right, the lines represent pattern 1 (black), pattern 2 (red), pattern 3 (blue) and pattern 4 (green) respectively. The lines for pattern 1 and 2 are pretty close because they both have some low QoS latency (1ms) requests, which significantly increases the batch sending frequency. Corresponding throughput is shown in Fig.9.

E. Multiple applications with different latency QoS

The workload that we use to test the system is organized as table II. Requests have different QoS latency requirements, and are submitted in random order. Although QoS latency are mostly satisfied, the workload pattern has huge impact on throughput. In pattern 1, 3 and 4, requests QoS latencies appear with same probability, while pattern 2 presents a highly skewed workload. Similarly as the results shown in fixed batch size experiments (Fig.5), longer QoS latency implies larger batch size, thus higher throughput (Fig.9).

Interestingly we find workload pattern 2 and 3 only have 4% difference, but the throughput of workload pattern 3 is almost 3x higher (Fig.9), the measured batch latency (Fig.8) also shows almost 10x difference. On the contrary, performance profiles of pattern 1 and 2 are similar, but the workload distributions are totally different (tab.II). The results shows how a small part of requests with low QoS latency can significantly influence overall performance. It also remind us that EDF batching strategy still has great potential to improve.

F. Throughput comparison on large scales

In this section we discuss the experiments on large scale deployments with simulation results. We construct the simula-

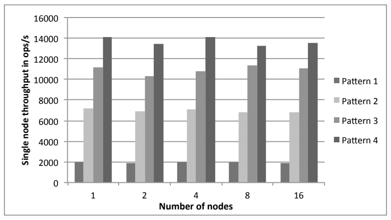
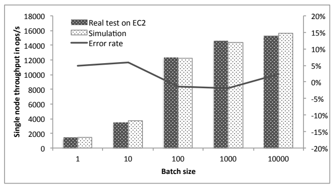
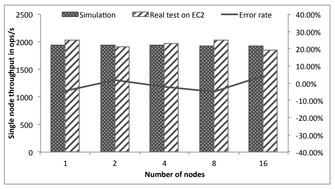


Fig. 9: Throughput of batching with different workloads. Low QoS latency (1ms) requests in pattern 1 and 2 significantly lower the total throughput, because they force the system to send batches more frequently. Corresponding latency distribution is shown in Fig.8.

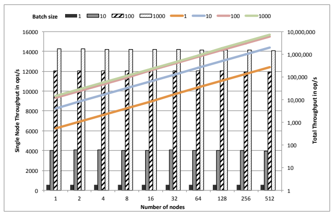


(a) Fixed batch size: The minimal difference between simulation result and real tested result from EC2 shows that the simulation can precisely predicts the performance of batching mechanism.

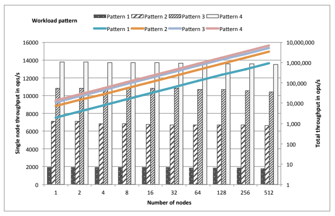


(b) EDF dynamic batching with workload pattern 1. The data captured from EC2 cloud differentiates slightly from simulation result, meaning the system scalability character is well simulated.

Fig. 10: Simulation validation



(a) Fixed batch size



(b) Various workload patterns

Fig. 11: Throughput comparison on scales

tor on top of PeerSim [11].

1) Simulation Validation: We firstly validate the simulation with real experimental results from fixed batch-size (Fig.10a) and EDF dynamic batching (Fig.10b) on Amazon EC2 cloud. The result only shows less than 5% error between real test and simulation result. This implies that the simulator can precisely predict the batching mechanism and the simulated throughput result on large scale is validated.

2) Simulation Results: We conduct experiments to evaluate the system scalability and total throughput with different batching mechanisms. Up to 512 nodes, both fixed-size static batching (Fig.11a) and EDF batching (Fig.11b) with different workloads show nearly constant single node throughput which demonstrate excellent scalability.

IV. Related Work

A. Approaches to boost performance of distributed storage systems

To achieve high throughput, low latency and better scalability in distributed storage systems on clouds, numerous works have been done in many aspects. Some focus on optimizing network communication. Kielmann's work [12] adopts dynamic load balanced multicast to offer more efficient communication for data-intensive application. Liu's work deploys a burst buffer [13] between parallel file system servers and compute nodes in supercomputers so as to avoid burst writing to the file system. Sata developed a model-based algorithm [14] for optimizing I/O intensive applications in clouds through VM layer coordination. Wolf's work [15] attempts to optimize massively parallel I/O and data locality. Some are trying to exploit new hardware, such as NV-RAM. Panda's team, in another hand, proposed new storage primitive [16] for emerging storage hardware. For large-scale storage-class memory systems, Jung [17] attempts to utilize Resistive Random Access Memory (RRAM), another promising memory technology to offer higher bandwidth and lower latency. There are also works done on parallel SQL databases, such as ParaLite [18], which supports collective queries to parallelize User-Defined Executables (EDU).

B. Key-value stores

This work is built on top of ZHT [4, 9, 19–23], a zero-hop distributed key-value store system, which has been tuned for the requirements of high-end computing systems. ZHT aims to be a building block for future distributed systems, such as parallel and distributed file systems [24, 25], distributed job scheduling systems [26–29], cloud storage systems [30–36], and parallel programming systems. There are some other recent works that have been done to improve key-value store performance and scalability in various aspects. MICA [10] is another scalable key value store that can handle millions of operations per second using single general multi purpose core system. MICA achieved this by encompassing all aspects of request handling by enabling parallel access to data, network request handling, and data structure design. SPANStore [37] presents a key value store that exports a unified view of storage services in geographically distributed data centers. SPANStore combines three main principles, spans multiple cloud providers to minimize cost, estimating application workload at the right granularity and finally minimizing use of compute resources. SPANStore in some scenarios was able to lower cost by 10X. Masstree [38] presents anther key value store designed for SMP machines. Masstree functions by keeping all data in memory in a form of concatenated B+ trees. Lookups use optimistic concurrency control, a read-copy-update like technique but no writing on shared data. With these techniques Masstree is able to execute more than six million simple queries per second. LOCS [39] is system equipped with customized SSD design, which exposes its internal flash channels to applications, to work with LSM-tree based KV store, specifically LevelDB in LOCS. Main motivation of LOCS was to overcome inefficiencies to naively combining LSM-tree based KV stores with SSD. They were able to show 4X increase in storage throughput after applying the proposed optimization techniques. Small Index Large Table (SILT) [40] presents a memory efficient high performance key value store system based on flash storage that can scale to serve billions of key value items on a single node. SILT focuses on using algorithmic and systemic techniques to balance the use of memory, storage and computation.

C. Request batching and QoS in key-value stores

For performance improvement, request batching are already used in some production systems, such as Facebook Memcached [41] and Amazon DynamoDB [42]. In these systems, users explicitly wrap requests into batches. Memcached allows users to call a `multiget` API to submit a batch of `get` requests. Note that the batch contains requests that will go to multiple servers. Then the server that initially received the multiget request will have to communicate to many other servers, which may increase the actual latency. Adding more servers will not help this case because the busy server is CPU bounded. This problem is now known as Multiget Hole [43]. In ZHT/Q, batching works in the background and users do not know any details. Multiget Hole problem is avoided by making the requests in a batches have a same destination server. DynamoDB has a similar mechanism for request batching. Another issue with these solutions is that an user must have all the requests ready by hand and then pack them in batches.

This requires users to use a very different set of APIs, thus some times change the applications' logic.

Request batching is also used to reduce power consumption. In [44] Cheng used a request batcher on server side to buffer requests so to keep the CPU in idle mode for longer time to save energy. In [45] Wang proposed a batching technique with DVFS for virtual machines to save power. But neither focuses on performance perspective and multiple QoS requirements.

There are couple of key-value store projects support QoS or SLA (service level agreement). Pileus [3] is a key-value store that allows applications to declare their consistency and latency requirements. The performance difference is implemented via choosing different consistency level and replication options. Zoolander [46] is a key value store that supports latency SLAs. Similar with Pileus, Zoolander makes use of systems data and workload conditions along with different replication options to deliver different performance level.

V. Conclusions

Every application has its own performance requirement but most of current key-value store systems are designed to serve every application request equally. In this paper we propose a flexible distributed key-value storage system which can be used by cloud providers and data centers to satisfy various applications' QoS requirement concurrently. It uses dynamic and adaptive request batching mechanisms to achieve both QoS on latency and high aggregated throughput. The experiment results show that our new system delivers up to 28 times higher throughput than the base solution while more than 99% of requests' latency requirements are satisfied. The results also remind us that wide range of latency requirements need to be handled carefully.

VI. Acknowledgments

This work is supported in part by the National Science Foundation grant NSF-1054974. This work used Amazon cloud resources through Amazon research grant.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," SIGMETRICS '12, 2012.

[2] J. C. Corbett, J. Dean, and M. Epstein, "Spanner: Google's globally-distributed database," OSDI, 2012.

[3] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '13.

[4] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," IPDPS '13.

[5] P. Xiong, H. Hacigumus, and J. F. Naughton, "A software-defined networking based approach for performance management of analytical queries on distributed data stores," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 955–966, ACM, 2014.

[6] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real Time Systems, The International Journal of Time-Critical Computing Systems*, 1990.

[7] M. Berg, F. V. D. D. Schouten, and J. Jansen, "Optimal batch provisioning to customers subject to a delay-limit," *Manage. Sci.*, vol. 44, pp. 684–697, May 1998.

[8] T. Cucinotta, "Access control for adaptive reservations on multi-user systems," in *Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE*, pp. 387–396, April 2008.

[9] T. Li, R. Verma, X. Duan, H. Jin, and I. Raicu, "Exploring distributed hash tables in highend computing," *SIGMETRICS Performance Evaluation Review*, 2011.

[10] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: a holistic approach to fast in-memory key-value storage," NSDI'14.

[11] K. Wang, A. Kulkarni, M. Lang, D. Arnold, and I. Raicu, "Using simulation to explore distributed key-value stores for extreme-scale system services," SC '13, 2013.

[12] T. Chiba, M. den Burger, T. Kielmann, and S. Matsuoka, "Dynamic load-balanced multicast for data-intensive applications on clouds," CCGrid 2010, pp. 5–14, May 2010.

[13] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, MSST'12, (Pacific Grove, CA, USA), pp. 1–1, 2012.

[14] K. Sato, H. Sato, and S. Matsuoka, "A model-based algorithm for optimizing i/o intensive applications in clouds using vm-based migration," CCGRID '09, pp. 466–471, May 2009.

[15] W. Frings, F. Wolf, and V. Petkov, "Scalable massively parallel i/o to task-local files," in *High Performance Computing Networking, Storage and Analysis, Proceedings of the Conference on*, pp. 1–11, Nov 2009.

[16] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. Panda, "Beyond block i/o: Rethinking traditional storage primitives," HPCA '11, pp. 301–311, Feb 2011.

[17] M. Jung, J. Shalf, and M. Kandemir, "Design of a large-scale storage-class rram system," ICS '13, (New York, NY, USA), pp. 103–114, ACM, 2013.

[18] T. Chen and K. Taura, "Paralite: Supporting collective queries in database system to parallelize user-defined executable," CCGRID '12, (Washington, DC, USA), pp. 474–481, IEEE Computer Society, 2012.

[19] T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang, and I. Raicu, "A convergence of key-value storage systems from clouds to supercomputers," *Concurr. Comput. : Pract. Exper.(CCPE)*, 2015.

[20] T. Li, A. P. de Tejada, K. Brandstatter, Z. Zhang, and I. Raicu, "ZHT: a zero-hop DHT for high-end computing environment," GCASR' 12.

[21] T. Li, X. Zhou, K. Brandstatter, and I. Raicu, "Distributed key-value store on HPC and cloud systems," GCASR' 13, 2013.

[22] K. Brandstatter, T. Li, X. Zhou, and I. Raicu, "NoVoHT: a lightweight dynamic persistent NoSQL key/value store," GCASR' 13, 2013.

[23] T. Li, "A convergence of NoSQL storage systems from clouds to supercomputers," *Illinois Institute of Technology, PhD Proposal*, 2014.

[24] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "Fusionfs: Towards supporting data-intensive scientific applications on extreme-scale high-performance computing systems," in *Big Data, 2014 IEEE International Conference on*.

[25] D. Zhao, C. Shou, Z. Zhang, I. Sadooghi, X. Zhou, T. Li, and I. Raicu, "FusionFS: a distributed file system for large scale data-intensive computing," GCASR' 13.

[26] I. Sadooghi, S. Palur, A. Anthony, I. Kapur, K. Belagodu, P. Purandare, K. Ramamurty, K. Wang, and I. Raicu, "Achieving efficient distributed scheduling with message queues in the cloud for many-task computing and high-performance computing," CCGrid'14, 2014.

[27] K. Wang, X. Zhou, T. Li, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," IEEE BigData'14.

[28] K. Wang, N. Liu, I. Sadooghi, X. Yang, X. Zhou, T. Li, M. Lang, X.-H. Sun, and I. Raicu, "Overcoming Hadoop scaling limitations through distributed task execution," IEEE Cluster'15, 2015.

[29] K. Wang, K. Qiao, I. Sadooghi, X. Zhou, T. Li, M. Lang, and I. Raicu, "Load-balanced and locality-aware scheduling for data-intensive workloads at extreme scales," *Concurrency and Computation: Practice and Experience*, 2015.

[30] T. Li, I. Raicu, and L. Ramakrishnan, "Scalable state management for scientific applications in the cloud," BigData Congress '14.

[31] T. Li, K. Keahey, R. Sankaran, P. Beckman, and I. Raicu, "A cloud-based interactive data infrastructure for sensor networks," IEEE/ACM Supercomputing/SC'14.

[32] I. Sadooghi, K. Wang, S. Srivastava, D. Patel, D. Zhao, T. Li, and I. Raicu, "Fabriq: Leveraging distributed hash tables towards distributed publish-subscribe message queues," IEEE/ACM International Symposium on Big Data Computing (BDC), 2015.

[33] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, "A dynamically scalable cloud data infrastructure for sensor networks," ACM ScienceCloud 15.

[34] T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, and I. Raicu, "GRAPH/Z: A key-value store based scalable graph processing system," Cluster'15.

[35] I. Sadooghi, J. Hernandez Martin, T. Li, K. Brandstatter, Y. Zhao, K. Maheshwari, T. Pais Pitta de Lacerda Ruivo, S. Timm, G. Garzoglio, and I. Raicu, "Understanding the performance and potential of cloud computing for scientific applications," *IEEE Transactions on Cloud Computing*, 2015.

[36] I. Sadooghi, D. Zhao, T. Li, and I. Raicu, "Understanding the cost of cloud computing and storage," GCASR, 2012.

[37] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," SOSP '13.

[38] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," EuroSys '12.

[39] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *The European Conference on Computer Systems*, EuroSys '14.

[40] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 1–13, ACM, 2011.

[41] R. Nishtala, H. Fugal, S. Grimm, and M. Kwiatkowski, "Scaling memcache at facebook," in *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI, (Lombard, IL), pp. 385–398, 2013.

[42] "DynamoDB." http://aws.amazon.com/dynamodb/. Accessed: 2014-12-15.

[43] High Scalability Blog, "Facebook's Memcached multiget hole: More machines != more capacity." http://highscalability.com/blog/2009/10/26/facebooks-memcached-multiget-hole-more-machines-more-capacit.html. Accessed: 2014-12-15.

[44] D. Cheng, Y. Guo, and X. Zhou, "Self-tuning batching with dvfs for improving performance and energy efficiency in servers," MASCOTS '13, (Washington, DC, USA), pp. 40–49, IEEE Computer Society, 2013.

[45] Y. Wang and X. Wang, "Virtual batching: Request batching for server energy conservation in virtualized data centers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, pp. 1695–1705, Aug. 2013.

[46] C. Stewart, A. Chakrabarti, and R. Griffith, "Zoolander: Efficiently meeting very strict, low-latency slos," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, (San Jose, CA), pp. 265–277, USENIX, 2013.