Haechi: A Token-based QoS Mechanism for One-sided I/Os in RDMA based Storage System

Qingyue Liu
Electrical and Computer Engineering Department
Rice University
Houston, USA
ql9@rice.edu

Peter Varman
Electrical and Computer Engineering Department
Rice University
Houston, USA
pjv@rice.edu

2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-4513-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICDCS51616.2021.00025

***Abstract*—Advances in persistent memory and networking hardware are changing the architecture of storage systems and data management services in datacenters. Distributed, one-sided RDMA access to memory-resident data shows tremendous improvements in throughput, latency and server CPU utilization of storage servers. However, the silent nature of one-sided I/O simultaneously creates new challenging problems for providing QoS in such systems. In this paper, we propose Haechi, a work-conserving, token-based QoS mechanism to guarantee reservations and limits in storage systems that provide one-sided I/O services. Haechi decouples QoS enforcement into a QoS engine at the client and a QoS monitor at the data node. It leverages adaptive token dispatch, token conversion, and silent I/O reporting to guarantee the reservations of distributed clients while maintaining high server utilization. Empirical evaluations on the Chameleon cluster, with different reservation distributions and I/O access patterns, show that Haechi is successful in providing differentiated QoS with negligible overhead for token management.**

***Index Terms*—QoS, Reservation guarantee, One-sided RDMA, Memory-resident data storage**

I. Introduction

Mechanisms to guarantee the performance of client applications sharing datacenter services are necessary to enforce Service Level Objectives (SLOs). These performance Quality of Service (QoS) measures typically take the form of reservation and limit guarantees, which apply to either physical resources like network bandwidth and CPU cycles, or to service-related metrics like request throughput and response time. In this paper, we focus on *throughput reservation* for storage I/Os in a distributed cluster using RDMA communication.

Continuing technological developments are leading to the migration of traditional Ethernet-based TCP/IP networking in datacenter storage to fast, user-level RDMA-enabled networks. In RDMA-based distributed storage clusters, clients can communicate with a data node to get I/O services using either one-sided or two-sided RDMA operations. Distributed storage systems based on two-sided RDMA have similar processes to handle I/Os as TCP/IP based storage systems, but they have better performance due to the smaller access latencies and kernel bypass of RDMA networks. However, it is a different situation for storage systems based on one-sided RDMA. These systems authorize clients to actively pull data from or push data to the target data node. Therefore, clients can *silently* perform I/O operations on the remote data nodes without interrupting the remote CPU. This results in tremendous improvements in the performance of RDMA-based storage systems in terms of throughput, latency and CPU utilization. As a result, more and more storage system designs leverage one-sided RDMA to provide I/O services. Examples of such systems include FaRM [1], HERD [2], Octopus [3], HyperLoop [4], Derecho [5], AsymNVM [6], and Telepathy [7].

QoS mechanisms for distributed storage built with traditional hardware architectures have been studied before. Server-centric QoS frameworks like bQueue [8], pShift [9], dSFQ [10], dClock [11] and mClock [12] are all well-known QoS solutions based on TCP/IP networking and HD/SSD-based backend storage. It is possible to directly adapt these traditional approaches to provide QoS when systems use two-sided RDMA operations. In this environment, the CPU at the data node is aware of all the requests made by its connected clients and I/Os are completed by its backend devices. Hence it is possible to use a traditional QoS scheduler at the data node to provide either weight-proportional or reservation and limit (minimum and maximum) throughput guarantees,

However, the situation is very different for one-sided RDMA communication. In this case, the CPU at the data node is not involved in processing the clients' requests. A client may directly read data from a remote memory address or write to a remote data location, without involving the CPU at data node [1], [2], [4]–[7]. As a result, data nodes are oblivious to the I/O accesses being performed by connected clients and cannot interpose a QoS scheduler to arbitrate requests. Consequently, the service received by different clients using one-sided operations depends only on their request rates and the default NIC scheduling of concurrent requests. However, current RNICs do not support reservation or differentiated I/O services. Even though we could force clients to register each I/O with the data node before it is triggered, this mechanism breaks the beauty of silent one-sided I/Os and involves additional CPU overhead at the data node for every I/O.

A. Contributions

In order to guarantee QoS for storage systems providing one-sided I/O services, new mechanisms are necessary: for

2575-8411/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDCS51616.2021.00025

performance, these mechanisms must allow one-sided I/Os to remain largely transparent to the data node, yet the I/Os must still be controlled to enforce differentiated QoS. Therefore, a new framework to provide QoS in storage systems supporting one-sided I/Os with small overhead is necessary.

In this paper we propose Haechi, the first QoS mechanism to provide reservation guarantees for services utilizing one-sided data access. Our paper makes the following contributions.

- We propose a token-based QoS architecture that splits QoS enforcement between a *QoS Engine* on the client and a *QoS Monitor* on the data node. Different RDMA communication modes are used to realize efficient, nearly-silent token dispatching and I/O monitoring operations.
- We design a lightweight, work-conserving QoS protocol to guarantee reservation and limits in storage services that employ one-sided I/O communications. The protocol uses two types of tokens: *reservation* tokens that are used to enforce I/O guarantees and a *global token pool* for additional best-effort I/Os.
- We propose an Adaptive Capacity Estimation Algorithm for continuously monitoring and adjusting the current IOPS capacity of the data node.
- We implement Haechi in the Chameleon cluster [13] and demonstrate its effectiveness using YCSB workloads.

The remainder of the paper is organized as follows. Section II describes Haechi's QoS model and design. Section III presents the experimental setup and evaluation results. Section IV discusses previous work on storage QoS. Finally, Section V summarizes the paper.

II. Haechi Overview and Design

Heachi is a lightweight, work-conserving token-based QoS mechanism designed to support *reservation* guarantees in storage systems that provide *one-sided* I/O services to their clients. The reservation of a client is the minimum number of I/Os it is guaranteed in a QoS period. Requests in excess of the reservation are served in a best-effort manner based on available server capacity. These requests have lower priority than requests from clients with remaining reservation. Unserved requests will be fulfilled in the next QoS period. Haechi is easily extended to handle limits on client requests.

The server (data node) implements a key-value store using a protocol like Telepathy [7] with one-sided I/Os. The overhead of Haechi is small and the storage system incurs only negligible performance loss after Heachi is deployed.

A. Haechi QoS Protocol

Haechi uses a QoS *engine* at the clients and a QoS *monitor* at the data node (Figure 1) to implement its protocol. The QoS engine regulates its client's I/Os and periodically reports its statistics on the data node. The QoS monitor dynamically updates clients' I/O allocation and re-estimates the server capacity based on reported statistics.

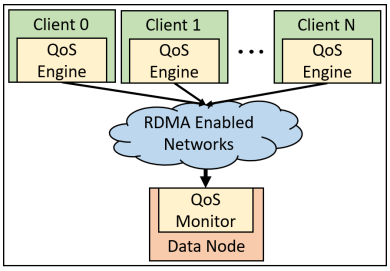


Fig. 1: Architecture for Haechi-based QoS.

B. Haechi QoS Model

In Haechi's QoS model, time is divided into QoS *periods* of length T. Each client i has a specified I/O *reservation* R_i; the client will receive at least R_i I/Os in a QoS period provided it is sufficiently *backlogged* (defined formally below). It may also have a specified *limit* L_i equal to the maximum number of I/Os it should receive in the period.

Haechi uses a token-based mechanism to allocate I/Os. A client must possess a token in order to perform an I/O; the token is consumed by the I/O. At the start of the period the data node generates a total of C tokens equal to its *capacity*, *i.e.* the number of I/Os it can perform in the QoS period. Each client i is allocated R_i of these tokens (called *reservation tokens*); these are pushed to the client and replace its existing tokens, if any. The remaining $C - \sum_i R_i$ tokens are held at the server in a pool of *global tokens*. Tokens in the global pool represent available unreserved capacity, which may be claimed dynamically by clients that have consumed all their reservation tokens.

The number of global tokens decreases when claimed by a client; it may also increase when clients (with insufficient demand) return their reservation tokens to the global pool. The total number of tokens (global and reservation) at any time is limited to the server capacity for the rest of the QoS period.

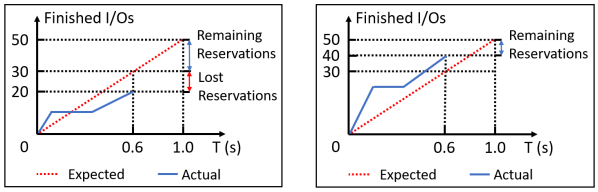
The backlog requirement characterizes constraints on clients' demand necessary to guarantee their reservation. For instance, a client i that makes less than R_i requests in the QoS period, clearly cannot meet its reservation. Similarly, if the client makes most of its requests just prior to the end of a QoS period, in general there can be insufficient capacity to finish them in this period; they will be satisfied in the next QoS period by using newly allocated tokens.

Definition 1: Client i has a reservation of R_i I/Os in the QoS period $[0, T]$. Let $r_i = R_i/T$ denote the average reservation rate over T and let $\rho_i(t) = r_i \times t$. Denote the *demand* (*i.e.* the number of requests) of client i in $[0, t]$ by $D_i(t)$. A *continuously-backlogged* client must satisfy $D_i(t) \geq \rho_i(t)$ at all $t \in [0, T]$. A *continuously-backlogged* client is guaranteed to receive its reservation R_i in the QoS period.

At any time $t \in [0, T]$, Haechi upper bounds the number of remaining reservation tokens for client i by $R_i - \rho_i(t)$; this is the maximum number of I/Os that Haechi will guarantee in the remaining interval $[t, T]$. Reservation tokens in excess of

the upper-bound are returned to the global pool. This allows the server to redirect service, in a principled way, from clients not using their reserved allocation to clients with demand and thereby maintain work conservation.

Example 1: Let $T = 1$ sec. Client 1 has reservation $R_1 = 50$ I/Os. Suppose that $D_1(0.6) = 20$ I/Os, which is less than $\rho_1(0.6) = 50 \times 0.6 = 30$ I/Os. Due to insufficient demand, it returns $\rho_1 - D_1 = 10$ reservation tokens to the global pool. Its residual reservation for the remainder of the period is $R_1 - \rho_1 = 20$ (see Figure 2(a)). Suppose instead that $D_i(0.6) = 40$, which exceeds $\rho_1(0.6)$. In this case, its residual reservation in the remaining period is $R_1 - \rho_1 = 10$ (see Figure 2(b)). Finally, if $D_i(0.6) \geq 50$ then it has already received its reservation guarantee for the QoS period and its residual reservation is 0.



(a) Insufficient demand. (b) Sufficient demand.

Fig. 2: Two cases of demand generated by a client.

C. Admission Control

Haechi's admission control ensures that the system capacity is sufficient to meet the reservation of admitted clients by checking whether admitting the new client will lead to an *aggregate capacity violation* or a *local capacity violation*. The latter constraint arises when using one-sided I/Os because it can require many clients operating at their maximum rate to saturate the server. In our experiments (Section III-B), each client can achieve a maximum throughput of only 400 KIOPS, while the system can achieve a throughput of 1570 KIOPS when there are 4 or more clients accessing the data node.

Definition 2: Let C_G denote the saturated system throughput and C_L the maximum client throughput. Let $\mathcal{A}$ denote the set of admitted clients. Denote the number of I/Os completed by client i at time t by $N_i(t)$. The *aggregate capacity constraint* requires that the saturated system capacity is sufficient to serve the reservation of all clients *i.e.* $\sum_{i \in \mathcal{A}} R_i \leq T \times C_G$. The *local capacity constraint* requires that all times the local capacity of any continuously backlogged client is sufficient to meet its reservation *i.e.* $R_i - N_i(t) \leq (T - t) \times C_L$.

Example 2: Suppose $C_G = 100$ IOPS and $C_L = 50$ IOPS. There are 5 clients with reservation $R_1 = 40$ and $R_i = 10$, $i = 2, \cdots, 5$. The total reservation $\sum_i R_i = 80 < C_G$. The local capacity constraint of each client at is met at $t = 0$ since $R_i < C_L$. Suppose all clients send a burst of R_i requests at $t = 0$. Then, $N_i(0.5) = 10$, $i = 1, \cdots, 5$, since C_G will be divided equally among the clients. At this time, only client 1 has remaining reservation (30 I/Os). However, this exceeds the number that can be done in 0.5s at the rate C_L.

D. Haechi Client QoS Engine

The Haechi QoS engine at the client has three components: *data access*, *token management*, and *reporting*. None of these operations involve the CPU at the data node, and are all either local or one-sided remote operations.

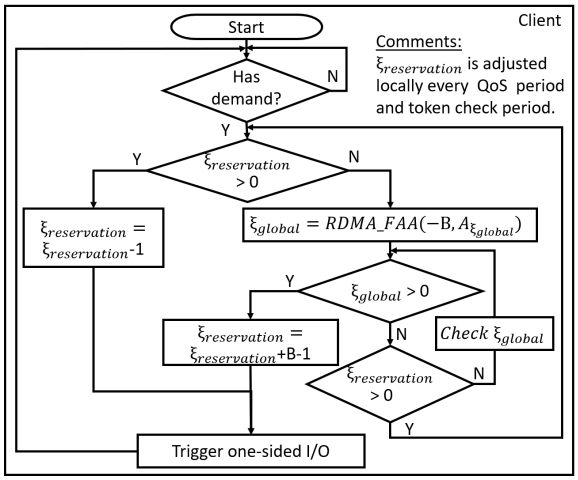


Fig. 3: Flow chart for data access in Haechi.

Client Data Access: Figure 3 shows a flowchart of the data access operations in the QoS engine. At the start of a QoS period, each QoS engine is sent *reservation tokens* ($\xi_{reservation}$) equal to its client's reservation. On receiving a request from its client, the engine checks if there are reservation tokens available, and if so it decrements the token count and performs a one-sided I/O. If there are no reservation tokens, the client must first acquire a token (ξ_{global}) from the pool of global tokens at the server. Global tokens are acquired by performing a remote Fetch-and-Add (FAA) atomic operation. To reduce the frequency of remote atomic operations, Haechi gets global tokens in a batched manner by fetching multiple tokens ($\mathcal{B} = 1000$ in our implementation) in each FAA operation. FAA returns the number of global tokens currently available at the server while atomically reducing their number by the amount specified. The client continues acquiring global tokens and performing one-sided I/Os as long as it has demand and can acquire global tokens. If the number of global tokens falls to zero or below, it means that the unreserved capacity of the server in this period has been consumed by I/Os. The client needs to wait for either the start of the next QoS period to receive fresh reservation tokens ($\xi_{reservation} > 0$), or for the monitor to convert unused reservation tokens to additional global ones.

Client Token Management: As discussed previously, clients that do not consume their reservation tokens must yield them back to the system. A local counter X is initialized to R_i at the start of the QoS period, and reduces at a steady rate of $r_i = R_i/T$. In our implementation, a thread updates X and $\xi_{reservation}$ every $\delta = 1$ ms. X is first reduced by $\delta \times r_i$. If $\xi_{reservation}$ is less than the updated value of X, then there is

sufficient demand and no update is necessary. Otherwise, the value of $\xi_{reservation}$ is reduced to the new value of X.

To handle limits, the QoS engine monitors the number of requests made by the client in the QoS period, and throttles it if the limit is exceeded. Note that the system will idle if all clients having requests have reached their limits.

Client Reporting: The reporting phase of a client is initiated by a signal from the QoS monitor at the data node (when the monitor detects that global tokens are being consumed). In this phase, a client reports two pieces of information to the data node: the number of remaining reservation I/Os for the rest of the period and the current value of N_i (number of I/Os completed so far this QoS period). In our implementation, the reporting is done at 1 ms intervals by performing a silent one-sided RDMA write of a single 64-bit value to a known location on the data node.

We end the discussion of client side operations with an overview shown in Figure 4. The steps T1 to T4 in the Figure are described below. As can be seen in the Figure, the client operations proceed largely silently at the clients without involving the CPU at the data node.

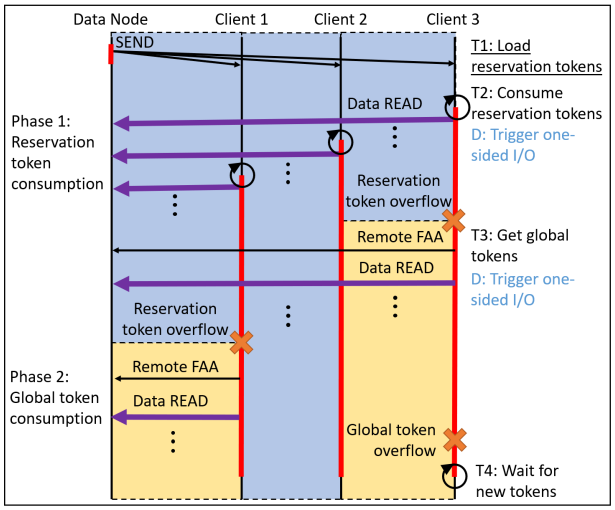


Fig. 4: Overview of Client operations in Haechi.

Step T1: When a client connects to the storage system it must provide its reservation requirement to the admission control module. Once admitted, the QoS monitor is given the reservation value. At the start of a QoS period, each client is sent this value by the QoS monitor to initialize $\xi_{reservation}$.

Step T2: Once a client receives the tokens for the current QoS period, it enters Phase 1 and begins performing I/Os on demand. In this phase, each I/O consumes one of its reservation tokens followed by the one-sided I/O (step D). The I/O sender function in the QoS engine will reject I/Os that are not backed by a token (global or reservation token). During this phase, the token management thread will reduce the token count if the client does not perform sufficient I/Os, and the reporting thread will inform the QoS monitor of the token count.

Step T3: After all the reservation tokens are consumed, the client enters Phase 2, where it competes for tokens in the global pool with other clients in Phase 2. Different clients will enter this phase at different times in the QoS period. In this phase, a client must first acquire a token from the pool of global tokens at the server before it can perform a one-sided I/O. Global tokens are acquired in a batch by performing a remote Fetch-and-Add (FAA) atomic operation, and one-sided I/O operations are performed (step D), until no more global tokens are available. The reporting thread continues to update the number of completed I/Os.

Step T4: The client waits till either the QoS monitor adds additional global tokens (by converting unused reservation tokens released by clients) or a new QoS period begins.

E. Haechi Data Node QoS Monitor

The Haechi QoS monitor at the data node has two components: *token management* and *capacity estimation*.

Data Node Token Management: Figure 5 shows an overview of the operations of the QoS monitor at the data node. At the start of a QoS period it generates C_G tokens equal to its current capacity (estimated at the end of the last QoS period). The QoS monitor then pushes R_i tokens to each client i using two-sided RDMA (*step* T1). This message also serves as a signal to notify the clients of the start of a new QoS period. After reservation token dispatching, the QoS monitor initializes the number of global tokens to $\xi_{global} = C_G - \sum_i R_i$ by writing the value to a location known to all clients.

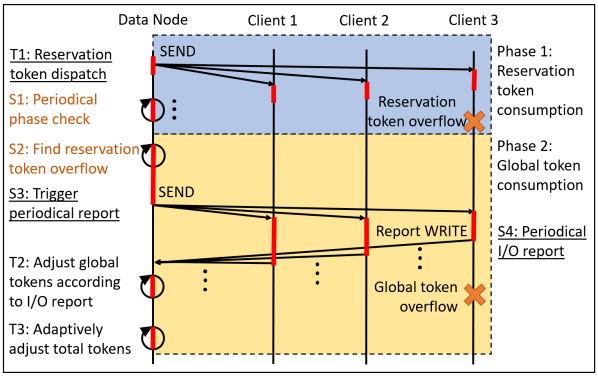


Fig. 5: Overview of Data Node operations in Haechi.

During the QoS period, the QoS monitor at data node wakes up once in each *check interval* (chosen as 1 ms in our implementation). It monitors the number of global tokens using a RDMA CAS instruction (*step* S1). If the QoS monitor finds that the number of global tokens has decreased from its initial value (*step* S2), it means that at least once client has finished its reservation tokens (reservation token overflow). It then requests all clients to begin periodic reporting of their usage statistics (*step* S3). For the rest of the period, the QoS monitor uses the reported statistics to reclaim unused

reservation tokens and add them to the global pool (*step* T2). Finally, just prior to the end of the QoS period (*step* T3) it calibrates the data node capacity C_G based on the received statistics, as discussed later.

To implement token conversion in step T2 at time t, the monitor uses the statistics reported by the clients at the current check interval. It computes the sum $\mathcal{L}$ of the last reported values of the residual reservation of the clients. $\mathcal{L}$ represents the maximum number of outstanding reservation I/Os in the QoS period after adjusting for insufficient demand. The monitor updates the number of global tokens (ξ_{global}) to $\max\{C_G \times (T-t) - \mathcal{L},\ 0\}$.

Data Node Capacity Estimation: The QoS monitor at the data node uses an adaptive capacity estimation algorithm to determine the IOPS capacity for the next QoS period. Initial system profiling is used to provide a baseline estimate of the system capacity. However, the actual capacity can fluctuate due to changes in the operating environment such as network congestion, demand distributions and request patterns. If the capacity is underestimated the monitor will allocate a smaller number of tokens than the number of I/Os that can be performed in the QoS period, causing unnecessary idleness and reducing system throughput. In extreme cases, admission control may unnecessarily deny clients admission. If the capacity is overestimated, the QoS monitor will allocate more tokens than can be completed in the QoS period. This overestimation can cause clients to miss their reservation: some clients may use excess tokens to exceed their reservation, while reservation of other clients remain pending. In our experiments we noticed that this situation happening in practice when we overestimate the capacity of the system.

Algorithm 1 shows the capacity estimation algorithm used in Haechi. Initially, the profiled value Ω_{prof} is used as the capacity estimate. To adjust the estimated capacity of the system in the next QoS period, we sum up the number of completed I/Os reported by all the clients. If the total number $\mathbf{U} = \sum_{i=1}^{n} U_i^t$ is equal to the estimated capacity Ω_t, we may have underestimated the capacity of the storage node. In this case, we add an increment η to Ω_t and use it as the estimated capacity for next QoS period. If $\mathbf{U}$ exceeds the capacity lower bound, we add $\mathbf{U}$ to the I/O history buffer $\mathcal{W}$. The average value of the history I/O records $\bar{\mathcal{W}}$ becomes Ω_{t+1}.

The lower bound of the capacity is set to be $\Omega_{prof} - 3\sigma$, where σ is the standard deviation of Ω_{prof}. In the profiling, we trigger continuous back-to-back 4 KB one-sided I/Os from 10 clients to the storage server for one QoS period and repeat the process 1000 times. Ω_{prof} and σ are calculated from the empirically determined capacity distribution. The lower bound of the capacity is set to prevent low-demand QoS periods from heavily influencing the capacity estimation and setting it to an unreasonably low value. We maintain the $\mathbf{U}$ value from the last M periods in which it exceeded the lower bound in the history buffer $\mathcal{W}$. The algorithm also monitors clients that consistently use less than their allocated reservation tokens and alert them that they may have overestimated their reservation.

Algorithm 1: Adaptive Capacity Estimation.

Input:
Number of clients: N; Profiled capacity: Ω_{prof};
Client Reservation: $\mathcal{R}_i$; History window size: M;
Token increment amount: η;
Capacity lower bound: $\Omega_{min} = \Omega_{prof} - 3\sigma$;
Output: Estimated capacity (in QoS period): Ω_{t+1}
Initial: $\Omega_{t+1} = \Omega_{prof}$
Collect $U^t = \{U_1^t, U_2^t, \cdots, U_N^t\}$ the number of completed I/Os from clients in t^{th} QoS period.
Compute $\mathbf{U} = \sum_{i=1}^{n} U_i^t$ the total number of completed I/Os in the t^{th} QoS period.
if ($\mathbf{U} == \Omega_t$) **then**
  $\Omega_{t+1} = \Omega_t + \eta$
else if ($\Omega_{min} \leq \mathbf{U} < \Omega_t$) **then**
  Add $\mathbf{U}$ into I/O history buffer $\mathcal{W}$.
  if $\mathcal{W}.size() > M$ **then**
    Remove the earliest record.
  end if
  $\Omega_{t+1} = \bar{\mathcal{W}}$.
else
  $\Omega_{t+1} = \Omega_t$
end if
Increment counter for all clients that did not use their reservation in this period.
Clear the counter for the remaining clients.

F. Security and Isolation

Haechi relies on the security of one-sided RDMA to prevent clients from violating QoS mechanisms. QoS engines and the QoS monitor are assumed to be trustworthy and tamper resistant. Channels between the client nodes and the data nodes are made exclusively through authenticated QoS engines. Applications go through the QoS engine to access remote NVRAM for both data and metadata.

Isolation between clients is enforced at both QoS engines and monitor. The monitor controls the total number of tokens. Requests which are not backed by tokens are blocked in the QoS engine. This prevents a runaway client from swamping the system with I/Os and interfering with well-behaved clients. The engines and monitor work cooperatively to move tokens from idle to busy clients.

III. Experiments and Evaluation

In this section, we empirically evaluate the performance of Haechi using four groups of experiments. The influence of temporal *request patterns* and spatial *reservation distributions* on system throughput, latency and the ability to meet QoS guarantees are analyzed through these experiments.

A. Experimental Setup

Haechi is implemented using the RDMA verbs library [14] and is deployed on a cluster of 11 servers in the Chameleon datacenter [13]. Servers are connected through

an Infiniband (IB) network. The hardware configurations of the servers and network are summarized in Table I. The Key-Value store is installed on one of the servers as the data node, and the remaining ten servers act as the the client nodes. We implement our QoS monitor in the data node to perform server-side token management and capacity evaluation functions (Section II-E). A QoS engine is installed in each client to provider client-side functions, specifically one-sided data access, token management, and reporting (Section II-D). We are not aware of any existing system or design with QoS support for one-sided I/Os. Hence, we compare Haechi with a bare system having the same setup but without QoS support.

CPU information	Intel(R) Xeon(R) CPU E5-2670 v3
Processor speed	2.30 GHz
Number of CPUs	48
Threads/Core	2
Memory size	128 GB
NIC Card	MT27500 Family [ConnectX-3]
Network	Infiniband

TABLE I: Hardware configuration in the Chameleon cluster.

Our experiments employ the YCSB [15] benchmark to replay one-sided 4KB read I/Os from clients to the data node. Initially, the memory-resident KV store is populated with a set of 1M 4 KB key-value records. For the experiments on the bare system, clients directly make their I/O requests to the data node. For the experiments using Haechi, the triggered reads go through the client QoS engine and follow the Haechi protocol. Each experiment lasts for 150 sec. The first 30 sec are used to warm up the system and the remaining 120 sec are used to collect the measured performance data. In order to show the distribution of completed I/Os in each QoS period clearly, we display only the results of the first 30 sec after the warm-up period.

B. Set 1: I/O characterization of system

In this set of experiments, we profile the bare system under different load distributions and request patterns. We measure the throughput of individual clients (Experiment 1A), followed by measuring the system throughput as the number of concurrent clients is varied (Experiment 1B). In Experiment 1C we vary the temporal request patterns as well as the distribution of total demand among the client nodes. The results from this set of experiments are used for admission control and scheduling.

Experiment 1A: The first experiment is to measure the throughput of a single client using both one-sided and two-sided I/Os. A client sends an initial burst of 64 requests to its QoS engine, and subsequently keeps 64 requests outstanding at all times. This is sufficient to saturate the client throughput without significantly increasing request latency. We will refer to this I/O pattern as *burst requests*. One client is active at a time and it triggers one-sided or two-sided read I/Os to the server.

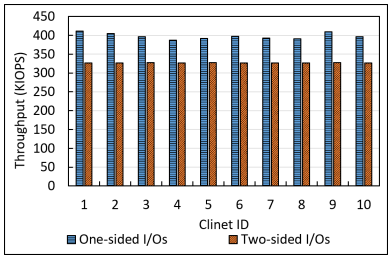


Fig. 6: Throughput of clients when run separately with 1-sided and 2-sided I/Os.

Figure 6 shows the results for each of the 10 clients when they are run one at-a-time. The results show that for one-sided I/Os all the clients have similar saturation throughput of around 400 KIOPS. For two-sided I/Os the maximum throughput of the clients is about 20% lower at around 327 KIOPS. This is used by admission control as the local capacity limit C_L. The I/O rate required to complete the residual reservation of a client using 1-sided I/Os must not exceed $C_L = 400$ KIOPS. This also implies that the maximum reservation of a client in a 1 sec QoS period is limited to 400K I/Os.

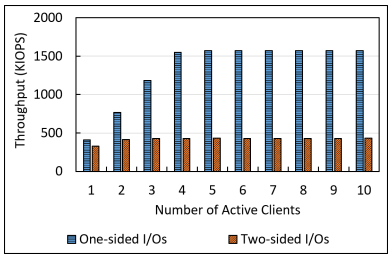


Fig. 7: Data node throughput versus number of active clients.

Experiment 1B: In this experiment, we vary the number of active clients from 1 to 10 and measure the system throughput using the same setup as Experiment 1A. Figure 7 shows the system throughput as the number of active clients is varied; this is the sum of the throughput measured at the clients.

As expected, for both one-sided and two-sided I/Os, the throughput increases initially and then levels off. For one-sided I/Os the throughput increases linearly for the first four nodes and then becomes saturated at slightly less than 1600 KIOPS. However, for two-sided I/Os, the throughput begins to flatten out almost immediately and reaches its saturation value of around 430 KIOPS for just two clients. This means that when using two-sided I/Os just one client can almost saturate the data node, while it takes 4 clients to saturate the throughput when using one-sided I/Os. This is because one-sided I/Os place almost no demand on the CPU at the data node, allowing it to serve several times the number of I/Os than can be sustained by a single client.

This is used by admission control as the global capacity limit C_G. The sum of the reservation of the admitted clients must not exceed $C_G = 1570$ KIOPS when using one-sided

I/Os, and $C_G = 427$ KIOPS when using two-sided I/Os. This maximum throughput is also the profiled capacity Ω_{prof} used in the capacity estimation algorithm.

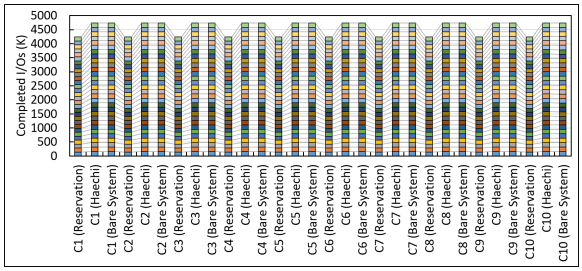
Experiment 1C: In this experiment, we investigate the behavior of the system with different *spatial demand distributions* and *temporal request patterns*. Each client is allocated a target number of I/Os that it must perform in a 1 sec period (called its demand). The client will make these many I/Os at a rate specified by its request pattern. The total demand of all the clients is chosen to be 1580 KIOPS, just enough to saturate the data node.

- In the first experiment we employ a *uniform demand* distribution and instruct each client to perform the same number (158K) I/Os every second. A *burst request* pattern is employed by each client. After it has completed its allocated target I/Os, the client idles for the rest of the period. The result of the experiment is shown in the Figure 8(a). As may be expected, the saturated system capacity is uniformly divided among the the clients. The figure shows that each client finishes around 157K I/Os and the total number of completed I/Os per second is around 1570K.
- We next change the client demand distribution to a *spike* distribution. In this scenario the total demand of 1580K I/Os is allocated to clients in a skewed manner. Clients 1-3 are allocated 340K I/Os each, and the remaining 7 are each allocated 80K I/Os. All clients use the *burst request* pattern. The distribution of the completed I/Os of the 10 clients is shown in Figure 8(b). The total number of completed I/Os per second drops to 1380K, significantly below the data node capacity of 1570 KIOPS. Moreover, clients $1, 2$ and 3 have a large gap between their allocated demand target (340K I/Os) and the actual number of completed I/Os (around 278K). The remaining clients receive around 80K I/Os, essentially matching their demand.
- Finally, we use the same *spike demand* distribution as before, but employ a *constant-rate request* pattern. In this case, a client's target I/Os are made at equal time intervals spread across the 1 sec period. Figure 8(c) shows the results. Clients 1 to 3 each now complete significantly more I/Os (332K) compared to the second experiment, approaching their target allocation of 340K I/Os. Moreover, the total number of completed I/Os increases to 1564K, close to that obtained with the uniform demand distribution of the first experiment.

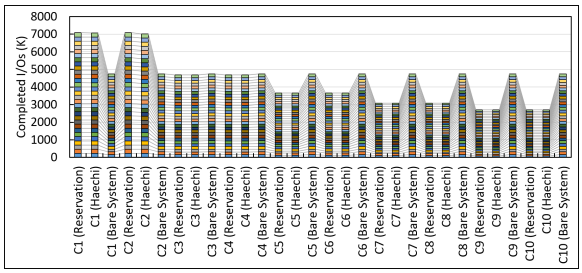
There are several reasons for the behaviors described above. The root cause for the performance drop in the second experiment is the inability of clients 1 to 3 to saturate the server by themselves, after clients 4 to 10 complete their I/Os. This behavior was observed in the plot shown in Figure 7.

At the start of the period, all 10 clients are backlogged. The server is saturated and completes I/Os at around 1570 KIOPS distributed equally among the 10 clients. This situation lasts for around 0.5 sec, at which time clients 4 to 10 will have completed their demand of 80K I/Os. This leaves clients 1 to 3 with an unfinished demand of around 260K $(340 - 80)$ I/Os to complete in the remaining 0.5 sec. However, this is higher than the individual client capacity of 400 KIOPS, and these clients only complete 200K I/Os in the remaining period.

When we change to the constant-rate request pattern in the third experiment, the problem no longer occurs. In this case, I/Os are triggered at the rate proportional to a client's demand. Hence more requests of clients 1 to 3 are performed in an interval compared to the other clients, and the data node will be saturated for the entire period. Section III-D will show how this behavior affects the ability to guarantee reservation in certain situations in Haechi.



(a) Uniform reservation distribution.



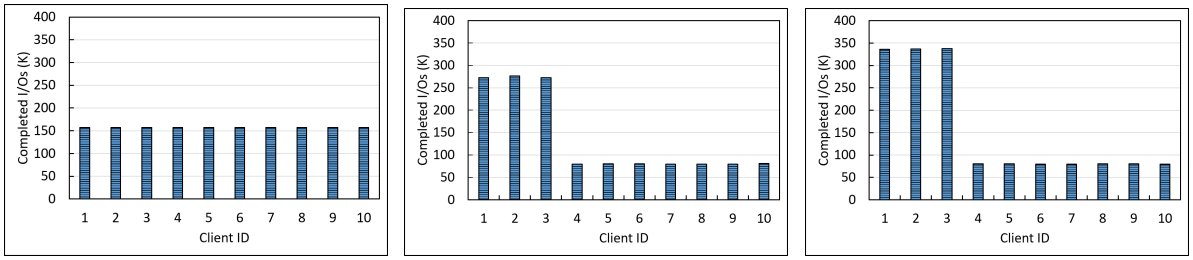
(b) Zipf reservation distribution.

Fig. 9: Time based completed I/O changes when there is enough demand from all clients.

C. Set 2: Haechi QoS

In this set of experiments, we compare the system running Haechi with the bare system without QoS support. Haechi is deployed on 11 servers with 1 data node and 10 clients that perform one-sided I/O reads. We consider two situations: (*i*) all clients have enough demand to use their reservation in each QoS period (Experiment 2A), and (*ii*) the case when some clients have less demand than their reservation (Experiment 2B). Two different reservation distributions, Uniform and Zipf, are used in the experiments. These are analogous to the uniform and spiked demand distributions. For the Uniform reservation distribution, all clients have equal reservation R_i. For Zipf, we divide the 10 clients into 5 groups and apply the Zipf distribution (with exponent 0.6) to the groups. Both the clients in a group will have the same reservation. All clients perform burst request pattern.

Experiment 2A: We compare the completed I/Os of different clients under the two reservation distribution patterns.



(a) Uniform demand with burst requests. (b) Spike demand with burst requests. (c) Spike demand with constant-rate requests.

Fig. 8: I/O completions with different demand distributions and request patterns.

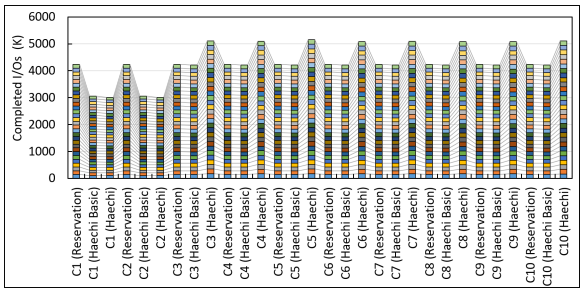
In this experiment all clients have demand in excess of their reservation. We use the profiled capacity of 1570 KIOPS as our capacity estimation. 90% of the capacity is reserved by the clients and their individual reservation tokens are sent to them at the beginning of each QoS period. The remaining 10% of capacity forms the initial global token pool. A client's demand equals the sum of the initial global tokens and its reservation.

The number of completed I/Os of each client in a 30 sec interval for the Uniform and Zipf reservation distributions are shown in Figure 9. In the figures, a group of three consecutive bars represents the results for one client. The first bar in a group is the reservation of the client. The second and third bars show the number of I/Os completed by the client using Haechi and on the bare system respectively. Each small block in the bar is the number of I/Os done in one QoS period.

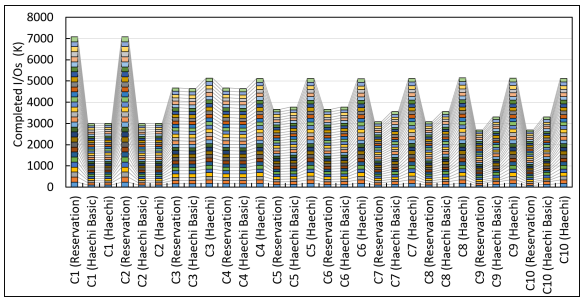
With the Uniform distribution all clients have equal reservation. From Figure 9(a), all clients meet their reservation in each QoS period in both the bare system and with Haechi enabled. Since the bare system does not differentiate between requests from different client nodes, equal allocations are the default in the bare system. The throughput of the system drops by less than 0.1% when Haechi is applied.

The number of completed I/Os for the Zipf reservation distribution is shown in Figure 9(b). For the bare system, the I/Os received by the clients are still equal, insensitive to their actual reservation. Consequently, clients whose reservation are higher than the average will fall short of their reservation. For instance, C1 and C2 complete only around 4742K I/Os each in the 30 sec interval rather than their reserved amount of 7080K I/Os. Clients with reservation lower than average (C5-C10) get more I/Os than their reservation. However, after Haechi is applied, all the clients will at least meet their reservation. C1 and C2 increase their I/Os at the expense of C5-C10. The results show how Haechi provides differentiated QoS guarantees to the clients.

Experiment 2B: We next consider the case when some clients have less demand than their reservation. The setup is same as Experiment 2A, except that in every QoS period clients C1 and C2 stop issuing requests before they have made all their reserved I/Os. We use this setup to show the benefit of token conversion (step T2 of Data Node Token Management in Figure 5) at the data node. We compare Haechi with a version



(a) Uniform reservation distribution.



(b) Zipf reservation distribution.

Fig. 10: Number of completed I/Os when the demand of clients C1, C2 is smaller than their reservation.

called *Basic Haechi* that does not perform this conversion *i.e.* it simply allows unused reservation to be wasted.

Figure 10 shows the number of completed I/Os of each client in a 30-sec interval for the Uniform and Zipf reservation distributions. For both distributions, clients C1 and C2 do not meet their reservation since they do not have enough demand. However, *Basic Haechi* simply wastes these unused reservation tokens allocated to the two clients; Haechi instead monitors and reclaims these tokens, and makes them available to the other clients. As can be seen, clients C3-C10 uses these converted global tokens to exceed their reservation, and achieve higher throughput than in *Basic Haechi*.

Figure 11 compares the total throughput of *Basic Haechi*, Haechi, and the bare system when clients C1 and C2 have insufficient demand. Haechi provides differentiated QoS while

achieving system throughput comparable to a bare system with no QoS support. Furthermore, token conversion results in work conservation and higher system throughput than simple QoS based on static token assignment.

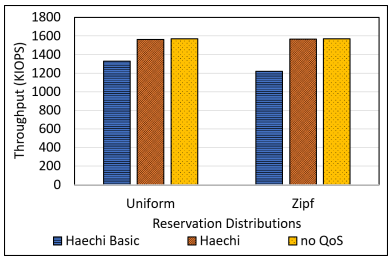


Fig. 11: Throughput of Haechi when the demand of clients C1, C2 are less than their reservation.

Experiment 2C: In this experiment, we vary the percentage of system capacity reserved by the clients and analyze how this affects the performance of Haechi. As before, all clients use the burst request pattern. We use the profiled capacity $C_G = 1570$ KIOPS as the initial estimated capacity. We vary the percentage of capacity reserved from 50% to 90% of C_G. The reservation distributions follow Uniform and Zipf reservation distributions respectively.

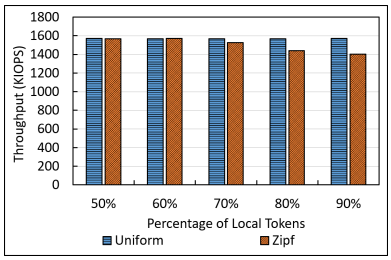


Fig. 12: Throughput of Haechi changes with varying reserved capacity and reservation distributions.

In this experiment, all the clients meet their reservation in each period using Haechi with both reservation distributions. Figure 12 shows the system throughput when the percentage of reserved capacity changes. With the Uniform reservation distribution, the demand of all clients are similar and the aggregate load does not vary during the QoS period; the system throughput is only limited by C_G. The throughput for the Zipf reservation distribution approaches the Uniform distribution as the percentage of reserved capacity decreases, This behavior is expected when requests backed by global tokens begin to dominate, since the competition for global tokens is fair and tends to distribute them equally.

When the reserved capacity is high, the system throughput drops for the Zipf distribution. In this case the smaller number of global tokens are quickly exhausted, causing clients with low reservation to idle after they meet their reservation. The throughput is then increasingly limited by the local capacity C_L. As described in Section III-B, when the number of active clients falls below 4, the system throughput falls.

D. Set 3: Varying the Request Patterns

In this set of experiments, we analyze the performance of Haechi with the two request patterns described earlier. The first pattern is burst requests, while the second pattern (constant-rate requests) performs the specified number of requests with equal time spacing over the QoS period. We use the Spike reservation distribution described in Experiment 1C. Clients C1-C3 have reservation of 285 KIOPS and C4-C10 have reservation of 80 KIOPS. The reservation amount is shown as the first bar of each client in Figure 13. The initial estimated capacity of the system is 1570 KIOPS and 90% of the capacity is reserved.

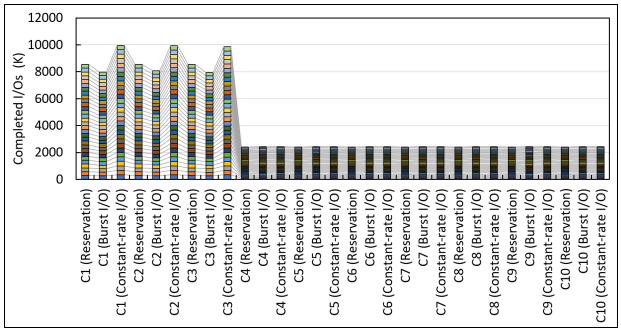


Fig. 13: Number of completed I/Os for Burst and Uniform request patterns and Spike reservation distribution.

Figure 13 shows the number of I/Os completed by the clients in the first 30 sec interval after the warm-up period. The results show that with burst requests and a Spike demand distribution, the clients with larger reservation such as clients C1-C3, do not meet their reservation. However, when all the clients use the constant-rate request pattern, clients C1-C3 not only meet but surpass their reservation. This observation matches the analysis in Experiment 1C. Figure 14 shows the throughput of the data node. The performance of the data node drops 12.9% and 0.7% for burst and constant-rate request patterns respectively. The better performance with constant-rate requests is because this pattern automatically gives clients with greater demand a higher I/O rate, thereby keeping the data node saturated for the entire period.

Figure 15 shows the average, 99% and 99.9% latency for read requests with the two different request patterns. We can see that both the average and the tail latencies with constant-rate requests are significantly lower than with burst requests. This is expected since the burst request pattern causes high queuing delay, while the constant-rate pattern has almost no queue buildup at the client nodes.

E. Set 4: Analysis of Capacity Estimation

In this set of experiments, we analyze the effectiveness of the Adaptive Capacity Estimation algorithm in handling throughput fluctuations. In Haechi, the estimated capacity

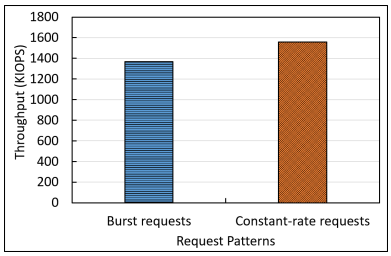


Fig. 14: Throughput of Haechi for Burst and Uniform request patterns and Spike reservation distribution.

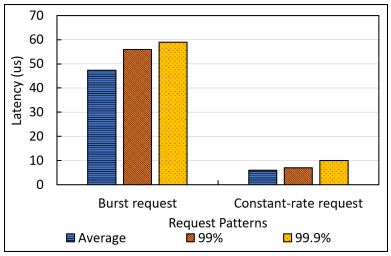


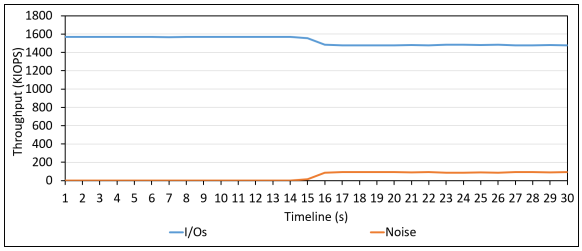
Fig. 15: Request latencies with Burst and Uniform requests.

determines the total number of tokens (and hence I/Os) that are allocated in a QoS period. We have shown (Experiment Set 2) that reservation can be met when the initial capacity estimate is accurate and there are no fluctuations in the network or data node performance. However, in practice the capacity of the storage system can deviate from the profiled estimate due to dynamic events. We simulate the effect of network fluctuations by injecting network traffic outside of Haechi's domain[1] The experiments intend to show that the Adaptive Capacity Estimation algorithm can successfully adjust the capacity based on dynamic measurements and guarantee QoS in unanticipated situations.

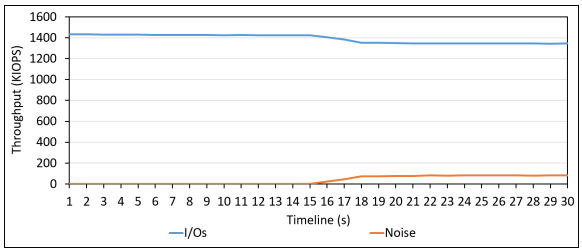
In this experiment, we induce a change in the network load during normal Haechi execution. We consider two situations. In the first setup, after 15 sec of normal operation each client node starts a background communication job. The background job generates burst I/Os to the data node, thereby decreasing the capacity available for the QoS-controlled Haechi clients. In the second setup, Haechi clients run together with the background communication jobs for the first 15 sec, after which the background jobs stop, thereby increasing the capacity available to Haechi clients. In the first setup, Haechi *over-estimates* the system capacity following the change, while in the second it *under-estimates* it. For both of the cases, 80% of the initial estimated capacity (based solely on Haechi clients) is reserved, so 20% is available in the initial global pool. We perform the experiment for Uniform and Zipf reservation distributions, using the burst request pattern.

Capacity Overestimation: Figure 16 shows the change in system throughput for the first setup when network congestion

[1]The resource reservation permitted in our experimental testbed are for short intervals that do not capture longer-term network fluctuations.

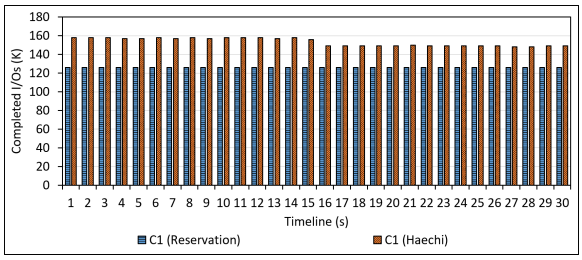


(a) Uniform reservation distribution.

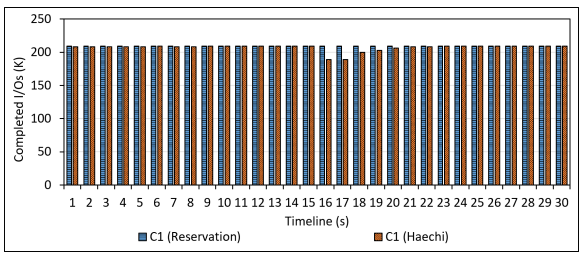


(b) Zipf reservation distribution.

Fig. 16: Effect of increased network congestion on Haechi.



(a) Uniform reservation distribution.



(b) Zipf reservation distribution.

Fig. 17: Effect of increased network congestion on QoS of high-reservation clients.

begins after 15 sec, for both Uniform (Figure 16(a)) and Zipf (Figure 16(b)) reservation distributions. In both cases, the throughput of the Haechi tasks starts to fall after the network congestion starts. This is as expected since the fixed system capacity needs to be shared with the background tasks. The Haechi monitor on the data node detects the change in capacity and adjusts the number of tokens accordingly.

Figure 17 shows the completed I/Os of the client with the highest reservation (client C1) for both reservation distribu-

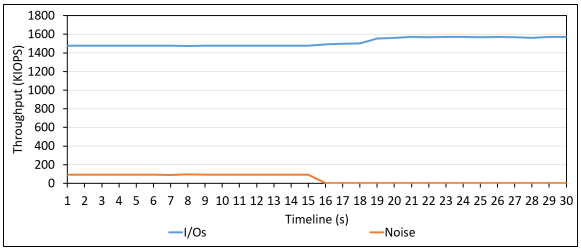
tions. Figure 17(a) shows the throughput of this client with the Uniform reservation distribution plotted against time. When network congestion starts, the number of I/Os done by C1 drops and remains constant at a lower value for the rest of the experiment. Figure 17(b) shows the throughput of C1 with the Zipf reservation distribution. In this case, C1 is not able to meet its reservation when network congestion starts, but its throughput increases gradually till it is able to do so. We discuss these results below.

When network congestion begins, the capacity available for Haechi's clients drops. As long as the background tasks do not consume more than 20% of the capacity, the aggregate reservation can still be met. Haechi initially overestimates the capacity of the system. This means that it creates more global tokens than can be supported by the actual system capacity. With the Uniform reservation distribution all clients complete their reservation at around the same time, and then compete equally for the global tokens. The QoS period will end before all the tokens can be consumed. Hence clients receive less total I/Os than without congestion, but still meet their reservation.

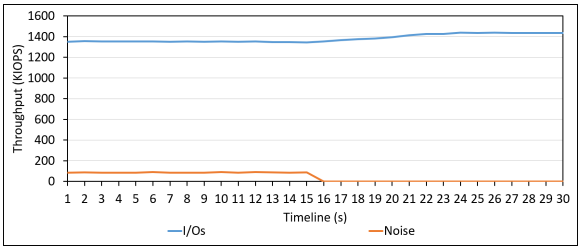
For the Zipf distribution the situation is more interesting. Clients with small reservation begin consuming global tokens before C1 (a high reservation client) has completed its reservation. These overcommitted I/Os compete with the reservation I/Os of C1, causing it to initially miss its reservation. However, our Adaptive Capacity Estimation algorithm will begin to correct this situation. It will gradually decrease the capacity estimate and reduce the number of excess global tokens, till it stabilizes at the new value. Therefore, C1 meets its reservation after a few rounds of adjustment as shown in Figure 17(b). Similar trends are also seen for the other high reservation clients. Low-reservation clients will always meet their reservation, but their total number of I/Os will decrease as the system stabilizes to the new capacity.

Capacity Underestimation: We now consider the case when Haechi underestimates the capacity after it changes. In this experiment, initial network congestion at the start of the experiment disappears after 15 sec as shown in Figure 18. For both Uniform and Zipf reservation distributions, the throughput of the Haechi system gradually increases after the network congestion stops. The QoS monitor detects the change and adjusts the token allocations accordingly.

Figure 19 shows the completed I/Os in each QoS period by client C1 that has the highest reservation. In all cases, C1 (as well as all the other clients not shown) meets its reservation. This is expected since removing network congestion will not negatively affect clients who met their reservation during the congested period. When network congestion disappears, Haechi initially underestimates the capacity of the system. The Adaptive Capacity Estimation algorithm will increase the estimated capacity by the token increment amount in each QoS period. With Uniform reservation distribution, the increase in the number of global tokens will increase the number of I/Os done by all clients as shown in Figure 19(a). However, for the Zipf reservation distribution shown in Figure 19(b), the clients with high reservation (C1) do not perform more I/Os

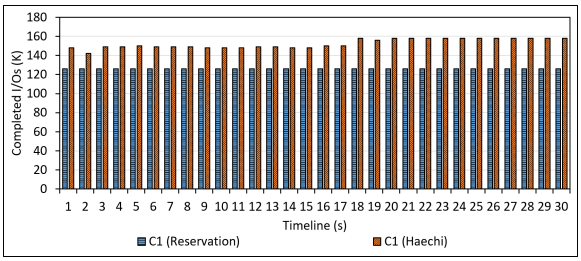


(a) Uniform reservation distribution.

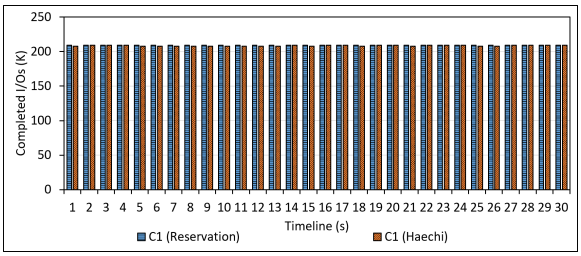


(b) Zipf reservation distribution.

Fig. 18: Effect of decreased network congestion on Haechi.



(a) Uniform reservation distribution.



(b) Zipf reservation distribution.

Fig. 19: Effect of decreased network congestion on QoS of high-reservation clients.

than their reservation. This is because the additional global tokens created will be consumed by the low-reservation clients as they finish their reservation. Hence, the number of I/Os of clients with low reservation (not shown in the Figure) starts to increase as the Capacity Estimation algorithm begins adjusting the capacity upwards.

IV. Related Work

QoS mechanisms in traditional distributed storage using both server-based and client-based QoS have been widely stud-

ied. In the former scheme, a scheduler on the server chooses the order in which queued clients' requests are serviced to meet QoS requirements. The scheduling decision is informed by metadata available at the server. Previous works such as Zygaria [16], dSFQ [10], dClock [11], mClock [12], bQueue [8] and pShift [9], [17] are all server-based QoS schemes. They employ either token-based QoS I/O allocation (*e.g.* [8], [9], [17]) or tag-based QoS scheduling (*e.g.* [16], [10], [11], [12]). In token-based schemes, the client's I/O allocations are controlled by credits, similar in principle to those used by Haechi. In tag-based schemes, rate-control mechanisms are used to label requests with relative priorities, which are used by the scheduler to order the dispatching of the requests. In all of these schemes the clients do not participate directly in QoS enforcement. The QoS scheduler runs at the data node using tag or token metadata to schedule the requests.

Client-based QoS schemes draw their inspiration from network flow control mechanisms. PARDA [18] is the representative example in this category. It provides weight-proportional throughput allocation (rather than reservations and limits of Haechi), using run-time latency measurements to regulate the request rate from the client. Client-based QoS has not been adopted in practice, due to its instability and sensitivity to capacity fluctuations.

As discussed in section I, server-based QoS mechanisms can also be applied to two-sided RDMA-based storage systems. In this case, the storage server is aware of all the I/Os made to it. Hence it can interpose a QoS scheduler to manage request traffic just as in conventional server-based QoS. However, with one-sided I/Os, the storage server is unaware of the I/Os performed by a client. A traditional QoS engine at the storage node is ineffective in this case. This is especially true for those services that authorize clients to actively pull data from or push data to the target storage node (e.g. FaSST [19], FaRM [1] and Telepathy [7]). Haechi solves these problems using a robust token-based control strategy while using one-side RDMA operations.

V. Conclusion and Future Work

In this paper, we proposed a novel QoS support mechanism called Haechi, for storage systems that provide one-sided I/O services. Haechi provides clients with guaranteed reservations and limits, while maintaining high server efficiency.

Haechi employs a dynamic, work-conserving token-based QoS mechanism using client-specific reservation tokens and shared global tokens to control the I/Os. Haechi can guarantee the reservations of the clients in a QoS period whose demands match or exceed their reservation. It transfers reservation tokens from clients without sufficient demand to clients that have unmet demand, to provide work conservation.

Haechi's admission control checks for potential global and local capacity violations to ensure that nodes are not overloaded. An adaptive capacity estimation algorithm is designed to dynamically adjust the capacity if it deviates from its profiled value. Empirical evaluations with different reservation distributions and I/O request patterns show that Haechi can guarantee the QoS of the storage system with negligible overhead for token management.

In future work we plan to extend Haechi to environments with multiple servers and distributed clients, similar to that for conventional distributed storage [8], [9], [17].

References

[1] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[2] A. Kalia, M. Kaminsky, and D. Andersen, "Using RDMA efficiently for key-value services," in *Proceedings of the 2014 ACM conference on SIGCOMM*, 2014, pp. 295–306.

[3] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an RDMA-enabled distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 773–785.

[4] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 297–312.

[5] S. Jha, J. Behrens, T. Gkountouvas, M. Milano, W. Song, E. Tremel, R. V. Renesse, S. Zink, and K. P. Birman, "Derecho: Fast state machine replication for cloud services," *ACM Transactions on Computer Systems (TOCS)*, pp. 1–49, 2019.

[6] T. Ma, M. Zhang, K. Chen, Z. Song, Y. Wu, and X. Qian, "AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 757–773.

[7] Q. Liu and P. Varman, "Silent Data Access Protocol for NVRAM+ RDMA Distributed Storage," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 1–10.

[8] Y. Peng and P. Varman, "bqueue: a coarse-grained bucket qos scheduler," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 93–102.

[9] Y. Peng, Q. Liu, and P. Varman, "Scalable qos for distributed storage clusters using dynamic token allocation," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2019, pp. 14–27.

[10] Y. Wang and A. Merchant, "Proportional-Share Scheduling for Distributed Storage Systems," in *FAST*, 2007, pp. 4–4.

[11] A. Gulati, A. Merchant, and P. Varman, "d-clock: Distributed QoS in heterogeneous resource environments," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 330–331.

[12] A. Gulati, A. Merchant, and P. J. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," in *OSDI*, 2010, pp. 437–450.

[13] N. S. Foundation, "A configurable experimental environment for large-scale cloud research," in *https://www.chameleoncloud.org/*, 2019.

[14] "RDMA Consortium. Architectural Specifications for RDMA over TCP/IP," in *http://www.rdmaconsortium.org/*, 2018.

[15] Yahoo, "Yahoo! Cloud Serving Benchmark (YCSB)," in *https://github.com/brianfrankcooper/YCSB*, 2018.

[16] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy, "Zygaria: Storage performance as a managed resource," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*. IEEE, 2006, pp. 125–134.

[17] Y. Peng and P. Varman, "pTrans: A Scalable Algorithm for Reservation Guarantees in Distributed Systems," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, 2020, pp. 441–452.

[18] A. Gulati, I. Ahmad, C. A. Waldspurger *et al.*, "PARDA: Proportional Allocation of Resources for Distributed Storage Access." in *FAST*, 2009, pp. 85–98.

[19] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 185–201.