C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection

Lalith Suresh[†] Marco Canini[⋆] Stefan Schmid[†‡] Anja Feldmann[†]
[†] *TU Berlin* [⋆] *Université catholique de Louvain* [‡] *Telekom Innovation Labs*

Abstract

Achieving predictable performance is critical for many distributed applications, yet difficult to achieve due to many factors that skew the tail of the latency distribution even in well-provisioned systems. In this paper, we present the fundamental challenges involved in designing a replica selection scheme that is robust in the face of performance fluctuations across servers. We illustrate these challenges through performance evaluations of the Cassandra distributed database on Amazon EC2. We then present the design and implementation of an adaptive replica selection mechanism, C3, that is robust to performance variability in the environment. We demonstrate C3's effectiveness in reducing the latency tail and improving throughput through extensive evaluations on Amazon EC2 and through simulations. Our results show that C3 significantly improves the latencies along the mean, median, and tail (up to 3 times improvement at the 99.9^{th} percentile) and provides higher system throughput.

1 Introduction

The interactive nature of modern web applications necessitates low and predictable latencies because people naturally prefer fluid response times [20], whereas degraded user experience directly impacts revenue [11,43]. However, it is challenging to deliver consistent low latency — in particular, to keep the tail of the latency distribution low [16,23,48]. Since interactive web applications are typically structured as multi-tiered, large-scale distributed systems, even serving a single end-user request (e.g., to return a web page) may involve contacting tens or hundreds of servers [17,23]. Significant delays at any of these servers inflate the latency observed by end users. Furthermore, even temporary latency spikes from individual nodes may ultimately dominate end-to-end latencies [2]. Finally, the increasing adoption of commercial clouds to deliver applications further exacerbates the response time unpredictability since, in these environments, applications almost unavoidably experience performance interference due to contention for shared resources (like CPU, memory, and I/O) [26,50,52].

Several studies [16,23,50] indicate that latency distributions in Internet-scale systems exhibit long-tail behaviors. That is, the 99.9^{th} percentile latency can be more than an order of magnitude higher than the median latency. Recent efforts [2, 16, 19, 23, 36, 44, 53] have thus proposed approaches to reduce tail latencies and lower the impact of skewed performance. These approaches rely on standard techniques including giving preferential resource allocations or guarantees, reissuing requests, trading off completeness for latency, and creating performance models to predict stragglers in the system.

A recurring pattern to reducing tail latency is to exploit the redundancy built into each tier of the application architecture. In this paper, we show that the problem of *replica selection* — wherein a *client* node has to make a choice about selecting one out of multiple *replica servers* to serve a request — is a first-order concern in this context. Interestingly, we find that the impact of the replica selection algorithm has often been overlooked. We argue that layering approaches like request duplication and reissues atop a poorly performing replica selection algorithm should be cause for concern. For example, reissuing requests but selecting poorly-performing nodes to process them increases system utilization [48] in exchange for limited benefits.

As we show in Section 2, the replica selection strategy has a direct effect on the tail of the latency distribution. This is particularly so in the context of data stores that rely on replication and partitioning for scalability, such as key-value stores. The performance of these systems is influenced by many sources of variability [16,28] and running such systems in cloud environments, where utilization should be high and environmental uncertainty is a fact of life, further aggravates performance fluctuations [26].

Replica selection can compensate for these conditions by preferring faster replica servers whenever possible. However, this is made challenging by the fact that servers exhibit performance fluctuations over time. Hence, replica selection needs to quickly adapt to changing system dynamics. On the other hand, any reactive scheme in this context must avoid entering pathological behaviors that lead to load imbalance among nodes and oscillating instabilities. In addition, replica selection should not be computationally costly, nor require significant coordination overheads.

In this paper, we present C3, an adaptive replica selection mechanism that is robust in the face of fluctuations in system performance. At the core of C3's design, two key concepts allow it to reduce tail latencies and hence improve performance predictability. First, using simple and inexpensive feedback from servers, clients make use of a replica ranking function to prefer faster servers and compensate for slower service times, all while ensuring that the system does not enter herd behaviors or load-oscillations. Second, in C3, clients implement a distributed rate control mechanism to ensure that, even at high fan-ins, clients do not overwhelm individual servers. The combination of these mechanisms enable C3 to reduce queuing delays at servers while the system remains reactive to variations in service times.

Our study applies to any low-latency data store wherein replica diversity is available, such as a key-value store. We hence base our study on the widely-used [15] Cassandra distributed database [5], which is designed to store and serve larger-than-memory datasets. Cassandra powers a variety of applications at large web sites such as Netflix and eBay [6]. Compared to other related systems (Table 1), Cassandra implements a more sophisticated load-based replica selection mechanism as well, and is thus a better reference point for our study. However, C3 is applicable to other systems and environments that need to exploit replica diversity in the face of performance variability, such as a typical multi-tiered application or other data stores such as MongoDB or Riak.

In summary, we make the following contributions:

1. Through performance evaluations on Amazon EC2, we expose the fundamental challenges involved in managing tail latencies in the face of service-time variability (§2).
2. We develop an adaptive replica selection mechanism, C3, that reduces the latency tail in the presence of service-time fluctuations in the system. C3 does not make use of request reissues, and only relies on minimal and approximate information exchange between clients and servers (§3).
3. We implement C3 (§4) in the Cassandra distributed database and evaluate it through experiments conducted on Amazon EC2 (for accuracy) (§5) and simulations (for scale) (§6). We demonstrate that our solution improves Cassandra's latency profile along the mean, median, and the tail (by up to a factor of 3 at the 99.9th percentile) whilst improving read throughput by up to 50%.

Cassandra	*Dynamic Snitching*: considers history of read latencies and I/O load
OpenStack Swift	Read from a single node and retry in case of failures
MongoDB	Optionally select nearest node by network latency (does not include CPU or I/O load)
Riak	Recommendation is to use an external load balancer such as Nginx [38]

Table 1: Replica selection mechanisms in popular NoSQL solutions. Only Cassandra employs a form of adaptive replica selection (§2.3).

2 The Challenge of Replica Selection

In this section, we first discuss the problem of time-varying performance variability in the context of cloud environments. We then underline the need for load-based replica selection schemes and the challenges associated with designing them.

2.1 Performance fluctuations are the norm

Servers in cloud environments routinely experience performance fluctuations due to a multitude of reasons. Citing experiences at Google, Dean and Barroso [16] list many sources of latency variability that occur in practice. Their list includes, but is not limited to, contention for shared resources within different parts of and between applications (further discussed in [26]), periodic garbage collection, maintenance activities (such as log compaction), and background daemons performing periodic tasks [40]. Recently, an experimental study of response times on Amazon EC2 [50] illustrated that long tails in latency distribution can also be exacerbated by virtualization. A study [23] of interactive services at Microsoft Bing found that over 30% of analyzed services have 95th percentile of latency 3 times their median latency. Their analysis showed that a major cause for the high service performance variability is that latency varies greatly across machines and time. Lastly, a common workflow involves accessing large volumes of data from a data store to serve as inputs for batch jobs on large-

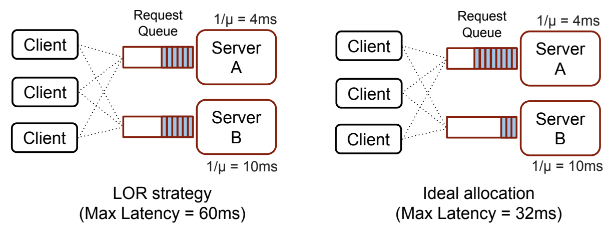


Figure 1: *Left*: how the least-outstanding requests (LOR) strategy allocates a burst of requests across two servers when executed individually by each client. *Right:* An ideal allocation that compensates for higher services time with lower queue lengths.

scale computing platforms such as Hadoop, and injecting results back into the data store [45]. These workloads can introduce latency spikes at the data store and further impact on end-user delays.

As part of our study, we spoke with engineers at Spotify and SoundCloud, two companies that use and operate large Cassandra clusters in production. Our discussions further confirmed that all of the above mentioned causes of performance variability are true pain points. Even in well provisioned clusters, unpredictable events such as garbage collection on individual hosts can lead to latency spikes. Furthermore, Cassandra nodes periodically perform *compaction*, wherein a node merges multiple SSTables [5, 13] (the on-disk representation of the stored data) to minimize the number of SSTable files to be consulted per-read, as well as to reclaim space. This leads to significantly increased I/O activity.

Given the presence of time-varying performance fluctuations, many of which can potentially occur even at sub-second timescales [16], it is important that systems gracefully adapt to changing conditions. By exploiting server redundancy in the system, we investigate how replica selection effectively reduces the tail latency.

2.2 Load-based replica selection is hard

Accommodating time-varying performance fluctuations across nodes in the system necessitates a replica selection strategy that takes into account the load across different servers in the system. A strategy commonly employed by many systems is the *least-outstanding requests* strategy (LOR). For each request, the client selects the server to which it has the least number of outstanding requests. This technique is simple to implement and does not require global system information, which may not be available or is difficult to obtain in a scalable fashion. In fact, this is commonly used in load-balancing applications such as Nginx [34] (recommended as a load-balancer for Riak [38]) or Amazon ELB [3].

However, we observe that this technique is not ideal for reducing the latency tail, especially since many realistic workloads are skewed in practice and access patterns change over time [9]. Consider the system in Figure 1, with two replica servers that at a particular point in time have service times of 4 ms and 10 ms respectively. Assume all three clients receive a burst of 4 requests each. Each request needs to be forwarded to a single server. Based on purely local information, if every client selects a server using the LOR strategy, it will result in each server receiving an equal share of requests. This leads to a maximum latency of 60 ms, whereas an ideal allocation in this case obtains a maximum latency of 32 ms. We note that *LOR* over time will prefer faster servers, but by virtue of purely relying on local information, it does not account for the existence of other clients with potentially bursty workloads and skewed access patterns, and does not explicitly adapt to fast-changing service times.

Designing distributed, adaptive and stable load-sensitive replica selection techniques is challenging. If not carefully designed, these techniques can suffer from "herd behavior" [32, 39]. Herd behavior leads to load oscillations, wherein multiple clients are coaxed to direct requests towards the least-loaded server, degrading the server's performance, which subsequently causes clients to repeat the same procedure with a different server.

Indeed, looking at the landscape of popular data stores (Table 1), we find that most systems only implement very simple schemes that have little or no ability to react quickly to service-time variations nor distribute requests in a load-sensitive fashion. Among the systems we studied, Cassandra implements a more sophisticated strategy called Dynamic Snitching that attempts to make replica selection decisions informed by histories of read latencies and I/O loads. However, through performance analysis of Cassandra, we find that this technique suffers from several weaknesses, which we discuss next.

2.3 Dynamic Snitching's weaknesses

Cassandra servers organize themselves into a one-hop distributed hash table. A client can contact any server for a read request. This server then acts as a *coordinator*, and internally fetches the record from the node hosting the data. Coordinators select the best replica for a given request using *Dynamic Snitching*. With Dynamic Snitching, every Cassandra server ranks and prefers faster replicas by factoring in read latencies to each of its peers, as well as I/O load information that each server shares with the cluster through a gossip protocol.

Given that Dynamic Snitching is load-based, we evaluate it to characterize how it manages tail-latencies and if it is subject to entering load-oscillations. Indeed, our experiments on Amazon EC2 with a 15-node Cassandra

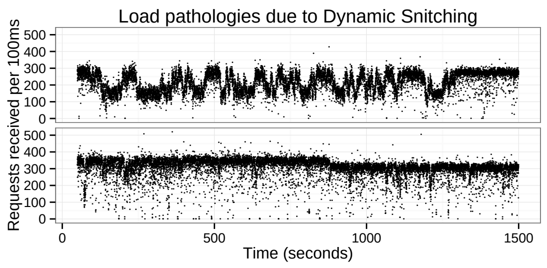


Figure 2: Example load oscillations seen by a given node in Cassandra due to Dynamic Snitching, in measurements obtained on Amazon EC2. The y-axis represents the number of requests processed in a 100 ms window by a Cassandra node. Even under stable conditions (*bottom*), the number of requests processed in a 100 ms window by a node ranges from 0 up to 500, which is symptomatic of herd behavior.

cluster confirm this (the details of the experimental setup are described in § 5). In particular, we recorded heavy-tailed latency characteristics wherein the difference between the 99.9th percentile latencies are *up to 10 times* that of the median. Furthermore, we recorded the number of read requests individual Cassandra nodes serviced in 100 ms intervals. For every run, we observed the node that contributed most to the overall throughput. These nodes consistently exhibited synchronized load oscillations, example sequences of which are shown in Figure 2. Additionally, we confirmed our results with the Spotify engineers, who have also encountered load instabilities that arise due to garbage-collection induced performance fluctuations in the system [29].

A key reason for Dynamic Snitching's vulnerability to oscillations is that each Cassandra node re-computes scores for its peers at fixed, discrete intervals. This interval based scheme poses two problems. First, the system cannot react to time-varying performance fluctuations among peers that occur at time-scales less than the fixed-interval used for the score recomputation. Second, by virtue of fixing a choice over a discrete time interval (100 ms by default), the system risks synchronization as seen in Figure 2. While one may argue that this can be overcome by shortening the interval itself, the calculation performed to compute the scores is expensive, as it is also stated explicitly in the source code; a median over a history of exponentially weighted latency samples (that is reset only every 10 minutes) has to be computed for each node as part of the scoring process. Additionally, Dynamic Snitching relies on gossiping one second averages of `iowait` information between nodes to aid with the ranking procedure (the intuition being that nodes can avoid peers who are performing compaction). These `iowait` measurements influence the scores used

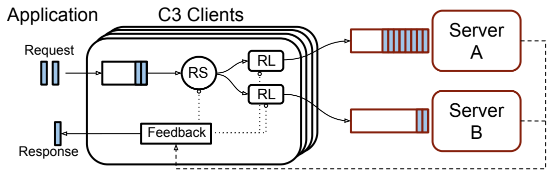


Figure 3: Overview of C3. RS: Replica Selection scheduler, RL: Rate Limiter of server $s \in [A, B]$.

for ranking peers heavily (up to two orders of magnitude more influence than latency measurements). Thus, an external or internal perturbation in I/O activity can influence a Cassandra node's replica selection loop for extended intervals. Together with the synchronization-prone behavior of having a periodically updated ranking, this can lead to poor replica selection decisions that degrade system performance.

3 C3 Design

C3 is an adaptive replica selection mechanism designed with the objective of reducing tail latency. Based on the considerations in Section 2, we design C3 while keeping in mind these two goals:

i) *Adaptive:* Replica selection must *cope and quickly react to heterogeneous and time-varying service times* across servers.

ii) *Well-behaved:* Clients performing replica selection must *avoid herd behaviors* where a large number of clients concentrate requests towards a fast server.

At the core of C3's design are the two following components that allow it to satisfy the above properties:

1. **Replica Ranking:** Using minimal and approximate feedback from individual servers, clients rank and prefer servers according to a scoring function. The scoring function factors in the existence of multiple clients and the subsequent risk of herd behavior, whilst allowing clients to prefer faster servers.
2. **Distributed Rate Control and Backpressure:** Every client rate limits requests destined to each server, adapting these rates in a fully-distributed manner using a congestion-control inspired technique [22]. When rate limits of all candidate servers for a request are exceeded, clients retain requests in a backlog queue until at least one server is within its rate limit again.

3.1 Replica ranking

With replica ranking, clients individually rank servers according to a scoring function, with the scores serving as a proxy for the latency to expect from the corresponding server. Clients then use these scores to prefer faster servers (lower scores) for each request. To reduce tail latency, we aim to minimize the product of queue-size (q_s)

and service-time ($1/\mu_s$, the inverse of the service rate) across every server s (Figure 1).

Delayed and approximate feedback. In C3, servers relay feedback about their respective q_s and $1/\mu_s$ on each response to a client. The q_s is recorded after the request has been serviced and the response is about to be dispatched. Clients maintain Exponentially Weighted Moving Averages (EWMA) of these metrics to smoothen the signal. We refer to these smoothed values as $\bar{q}_s$ and $\bar{\mu}_s$.

Accounting for uncertainty and concurrency. The delayed feedback from the servers lends clients only an approximate view of the load across the servers and is not sufficient by itself. Such a view is oblivious to the existence of other clients in the system, as well as the number of requests that are potentially in flight, and is thus prone to herd behaviors. It is therefore imperative that clients account for this potential concurrency in their estimation of each server's queue-size.

For each server s, a client maintains an instantaneous count of its outstanding requests os_s (requests for which a response is yet to be received). Clients calculate the queue-size estimate ($\hat{q}_s$) of each server as $\hat{q}_s = 1 + os_s \cdot w + \bar{q}_s$, where w is a weight parameter. We refer to the $os_s \cdot w$ term as the *concurrency compensation*.

The intuition behind the concurrency compensation term is that a client will always extrapolate the queue-size of a server by an estimated number of requests in flight. That is, it will always account for the possibility of multiple clients concurrently submitting requests to the same server. Furthermore, clients with a higher value of os_s will implicitly project a higher queue-size at s and thus rank it lower than a client that has sent fewer requests to s. Using this queue-size estimate to project the $\hat{q}_s/\bar{\mu}_s$ ratio results in a desirable effect: a client with a higher demand will be more likely to rank s poorly compared to a client with a lighter demand. This hence provides a degree of robustness to synchronization. In our experiments, we set w to the number of clients in the system. This serves as a good approximation in settings where the number of clients is comparable to the expected queue lengths at the servers.

Penalizing long queues. With the above estimation, clients can compute the $\hat{q}_s/\bar{\mu}_s$ ratio of each server and rank them accordingly. However, given the existence of multiple clients and time-varying service times, a function linear in $\hat{q}$ is not an effective scoring function for replica ranking. To see why, consider the example in Figure 4. The figure shows how clients would score two servers using a linear function: here, the service time estimates are 4 ms and 20 ms, respectively. We observe that under a linear scoring regime, for a queue-size estimate

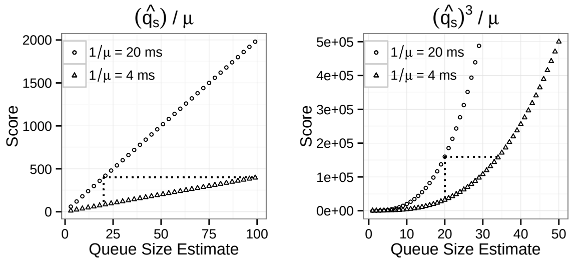


Figure 4: A comparison between linear (left) and cubic (right) scoring functions. For differing values of $1/\mu$, the difference in queue-size estimates required for the scores of two replicas to be equal is smaller for the cubic function (thus penalizing longer queues).

of 20 at the slower server, only a corresponding value of 100 at the faster server would cause a client to prefer the slower server again. If clients distribute requests by choosing the best replica according to this scoring function, they will build up and maintain long queues at the faster server in order to balance response times between the two nodes.

However, if the service time of the faster server increases due to an unpredictable event such as a garbage collection pause, *all requests* in its queue will incur higher waiting times. To alleviate this, C3's scoring function *penalizes longer queue lengths* using the same intuition behind that of delay costs as in [10,46]. That is, we use a non-decreasing convex function of the queue-size estimate in the scoring function to penalize longer queues. We achieve this by raising the $\hat{q}_s$ term in the scoring function to a higher degree, b: $(\hat{q}_s)^b/\bar{\mu}_s$.

Returning to the above example, this means the scoring function will treat the above two servers as being of equal score if the queue-size estimate of the faster server ($1/\mu = 4$ ms) is $\sqrt[b]{20/4}$ times that of the slower server ($1/\mu = 20$ ms). For higher values of b, clients will be less greedy about preferring a server with a lower μ^{-1}. We use $b = 3$ to have a cubic scoring function (Figure 4), which presents a good trade-off between clients preferring faster servers and providing enough robustness to time-varying service times.

Cubic replica selection. In summary, clients use the following scoring function for each replica:

$$\Psi_s = R_s - 1/\bar{\mu}_s + (\hat{q}_s)^3/\bar{\mu}_s$$

where $\hat{q}_s = 1 + os_s \cdot n + \bar{q}_s$ is the queue-size estimation term, os_s is the number of outstanding requests from the client to s, n is the number of clients in the system, and R_s, $\bar{q}_s$ and $\bar{\mu}_s{}^{-1}$ are EWMAs of the response time (as witnessed by the client),[1] queue-size and service time

[1]Note R_s implicitly accounts for network latency but we consider

feedback received from server s, respectively. The score reduces to R_s when the queue-size estimate term of the server is 1 (which can only occur if the client has no outstanding requests to s and the queue-size feedback is zero). Note that the $R_s - \mu_s^{-1}$ term's contribution to the score diminishes quickly when the client has a non-zero queue-size estimate (see Figure 4).

3.2 Rate control and backpressure

Replica selection allows clients to prefer faster servers. However, replica selection alone cannot ensure that the combined demands of all clients on a single server remain within that server's capacity. Exceeding capacity increases queuing on the server-side and reduces the system's reactivity to time-varying performance fluctuations. Thus, we introduce an element of rate-control to the system, wherein every client rate-limits requests to individual servers. If the rates of all candidate servers for a request are saturated, clients retain the request in a backlog queue until a server is within its rate limit again.

Decentralized rate control. To account for servers' performance fluctuations, clients need to adapt their estimations of a server's capacity and adjust their sending rates accordingly. As a design choice and inspired by the CUBIC congestion-control scheme [22], we opt to use a decentralized algorithm for clients to estimate and adapt rates across servers. That is, we avoid the need for clients to inform each other about their demands for individual servers, or for the servers to calculate allocations for potentially numerous clients individually. This further increases the robustness of our system; clients' adaptation to performance fluctuations in the system is not purely tied to explicit feedback from the servers.

Thus, every client maintains a token-bucket based rate-limiter for each server, which limits the number of requests sent to a server within a specified time window of δ ms. We refer to this limit as the *sending-rate* (*srate*). To adapt the rate limiter according to the perceived performance of the server, clients track the number of responses being received from a server in a δ ms interval, that is, the *receive-rate* (*rrate*). The rate-adaptation algorithm aims to adjust *srate* in order to match the *rrate* of the server.

Cubic rate adaptation function. Upon receiving a response from a server s, the client compares the current *srate* and *rrate* for s. If the client's sending rate is lower than the receive rate, it increases its rate according to a cubic function [22]:

$$srate \leftarrow \gamma \cdot \left(\Delta T - \sqrt[3]{(\frac{\beta \cdot R_0}{\gamma})} \right)^3 + R_0$$

that network congestion is not the source of performance fluctuations.

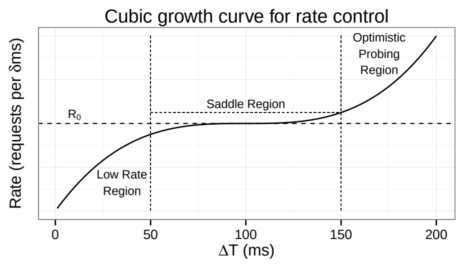


Figure 5: Cubic function for clients to adapt their sending rates

where ΔT is the elapsed time since the last rate-decrease event, and R_0 is the "saturation rate" — the rate at the time of the last rate-decrease event. If the receive-rate is lower than the sending-rate, the client decreases its sending-rate multiplicatively by β. γ represents a scaling factor and is chosen to set the desired duration of the saddle region (see § 4 for the values used).

Benefits of the cubic function. While we have not fully explored the vast design space for a rate adaptation technique, we were attracted to a cubic growth function because of its property of having a saddle region. The functioning of the *cubic* rate adaption strategy caters to the following three operational regions (Figure 5): (1) *Low-rates:* when the current sending rate is significantly lower than the saturation rate (after say, a multiplicative decrease),the client increases the rate steeply; (2) *Saddle region:* when the sending rate is close to the perceived saturation point of the server (R_0), the client stabilizes its sending rate, and increases it conservatively, and (3) *Optimistic probing:* if the client has spent enough time in the stable region, it will again increase its rate aggressively, and thus probe for more capacity. At any time, if the algorithm perceives itself to be exceeding the server's capacity, it will update its view of the server's saturation point and multiplicatively reduce its sending rate. The parameter γ can be adjusted for a desired length of the saddle region. Lastly, given that multiple clients may potentially be adjusting their rates simultaneously, for stability reasons, we cap the step size of a rate increase by a parameter $s_{\max}$.

3.3 Putting everything together

C3 combines distributed replica selection and rate control as indicated in Algorithms 1 and 2, with the control flow in the system depicted in Figure 3. When a request is issued at a client, it is directed to a replica selection scheduler. The scheduler uses the scoring function to order the subset of servers that can handle the request, that is, the replica group ($\mathscr{R}$). It then iterates through the list of replicas and selects the first server s that is within

Algorithm 1 On Request Arrival (Request *req*, Replicas $\mathscr{R}$)

```
1: repeat
2:     $\mathscr{R} \leftarrow sort(\mathscr{R})$        ▷ sort replicas by cubic score function
3:     for Server $s$ in $\mathscr{R}$ do
4:         if $s$ within $srate_s$ then
5:             $consume\_token(srate_s)$
6:             $os_s \leftarrow os_s + 1$        ▷ update outstanding requests
7:             $send(req, s)$        ▷ send to server $s$
8:             return
9:     if $req$ not sent then
10:        wait until token available        ▷ Backpressure
11: until $req$ is sent
```

Algorithm 2 On Request Completion (Request *req*, Server *s*)

```
1: $os_s \leftarrow os_s - 1$        ▷ update outstanding requests
2: update EWMA of $q_s$, $\mu_s^{-1}$ feedback
3: if ($srate_s > rrate_s$ && $now() - T_{inc} > hysteresis\_period$) then
4:     $R_0 \leftarrow srate_s$
5:     $srate_s \leftarrow srate_s \cdot \beta$
6:     $T_{dec} \leftarrow now()$
7: else if ($srate_s < rrate_s$) then
8:     $\Delta T \leftarrow now() - T_{dec}$
9:     $T_{inc} \leftarrow now()$
10:    $R \leftarrow \gamma \cdot \left(\Delta T - \sqrt[3]{(\frac{\beta \cdot R_0}{\gamma})}\right)^3 + R_0$
11:    $srate_s \leftarrow \min(srate_s + s_{\max}, R)$
```

the rate as defined by the local rate limiter for s. If all replicas have exceeded their rate limits, the request is enqueued into a backlog queue. The scheduler then waits until at least one replica is within its rate before repeating the procedure. When a response for a request arrives, the client records the feedback metrics from the server and adjusts its sending rate for that server according to the cubic-rate adaptation mechanism. After a rate increase, a hysteresis period is enforced (Algorithm 2, line 3) before another rate-decrease so as to allow clients' receive-rate measurements enough time to catch up since the last increased sending rate at T_{inc}.

4 Implementation

We implemented C3 within Cassandra. For Cassandra's internal read-request routing mechanism, this means that every Cassandra node is both a C3 client and server (specifically, coordinators in Cassandra's read path are C3 clients). In vanilla Cassandra, every read request follows a synchronous chain of steps leading up to an eventual enqueuing of the request into a per-node TCP connection buffer. For C3, we modified this chain of steps to control the number of requests that would be pushed to the TCP buffers of each node. Recall that C3's replica scoring and rate control operate at the granularity of replica groups. Given that in Cassandra, there are as many replica groups as nodes themselves, we need as many backpressure queues and replica selection schedulers as there are nodes. Thus, every read-request upon arrival in the system needs to be asynchronously routed to a scheduler corresponding to the request's replica group. Lastly, when a coordinator node performs a remote read, the server that handles the request tracks the service time of the operation and the number of pending read requests in the server. This information is piggybacked to the coordinator and serves as the feedback for the replica ranking.

There are challenges in making this implementation efficient. For one, since a single remote peer can be part of multiple replica sets, multiple admission control schedulers may potentially contend to push a request from their respective backpressure queues towards the same endpoint. Care needs to be exercised that this does not lead to starvation. To handle this complexity, we relied upon the Akka framework [1] for message-passing concurrency (*Actor* based programming). With Akka, every per-replica-group scheduler is represented as a single actor, and we configured the underlying Java thread dispatcher to fair schedule between the actors. This design of having multiple backpressure queues also increases robustness, as one replica group entering backpressure will not affect other replica groups. The message queue that backs each Akka actor implicitly serves as the backpressure queue per-replica group. At roughly 600 bytes of overhead per actor, our extensions to Cassandra is thus lightweight. Our implementation amounted to 398 lines of code.[2]

For the rest of our study, we set the cubic rate adaptation parameters as follows: the multiplicative decrease parameter β is set to 0.2, and we configured γ to set the saddle region to be 100 ms long. We define the rate for each server as a number of permissible requests per 20 ms (δ), and use a hysteresis duration equal to twice the rate interval. We cap the cubic-rate step size (s_{max}) to 10. We did not conduct an exhaustive sensitivity analysis of all system parameters, which we leave for future work. Lastly, Cassandra uses read-repairs for anti-entropy; a fraction of read requests will go to all replicas (10% by default). This further allows coordinators to update their view of their peers.

5 System Evaluation

We evaluated C3 on Amazon EC2. Our Cassandra deployment comprised 15 m1.xlarge instances. We tuned the instances and Cassandra according to the officially recommended production settings from Datastax [12] as well as in consultation with our contacts from the industry who operate production Cassandra clusters.

On each instance, we configured a single RAID0 array encompassing the four ephemeral disks which served

[2]Based on a Cassandra 2.0 development version.

as Cassandra's data folder (we also experimented on instances with SSD storage as we report on later). As we don't have production workloads, we used the industry-standard Yahoo Cloud Serving Benchmark (YCSB) [14] to generate datasets and run our workloads while stressing Cassandra up to its maximum attainable throughput. We assign tokens to each Cassandra node such that nodes own equal segments of the keyspace. Cassandra's replication factor was set to 3. We inserted 500 million 1KB size records generated by YCSB, which served as the dataset. The workload against the cluster was driven from three instances of YCSB running in separate VMs, each running 40 request generators, for a total of 120 generators. Each generator has a TCP connection of its own to the Cassandra cluster. Generators create requests for keys distributed according to a Zipfian access pattern prescribed by YCSB, with Zipf parameter $\rho = 0.99$, drawing from a set of 10 million keys. We used three common workload patterns for Cassandra deployments to evaluate our scheme: read-heavy (95% reads – 5% writes), update-heavy (50% reads – 50% writes) and read-only (100% read). These workloads generate access patterns typical of photo tagging, session-store and user-profile applications, respectively [14]. The read and update heavy workloads in particular are popular across a variety of Cassandra deployments [18, 25]. Each measurement involves 10 million operations of the workload, and is repeated five times. Bar plots represent averages and 95th percentile confidence intervals.

In evaluating C3, we are interested in answering the following questions across various conditions:

1. Does C3 improve the tail latency without sacrificing the mean or median?
2. Does C3 improve the read throughput (requests/s)?
3. How well does C3 load condition the cluster and adapt to dynamic changes in the environment?

Impact of workload on latency: Figure 6 indicates the read latency characteristics of Cassandra across different workloads when using C3 compared to Dynamic Snitching (DS). Regardless of the workload used, C3 improves the latency across all the considered metrics, namely, the mean, median, 99^{th} and 99.9^{th} percentile latencies. Since the ephemeral storage in our instances are backed by spinning-head disks, the latency increases with the amount of random disk seeks. This explains why the read-heavy workload results in lower latencies than the read-only workload (since the latter causes more random seeks). Furthermore, C3 effectively shortens the ratio of tail-latencies to the median, leading to a more predictable latency profile. With the read-heavy workload, the difference between the 99.9^{th} percentile latency and

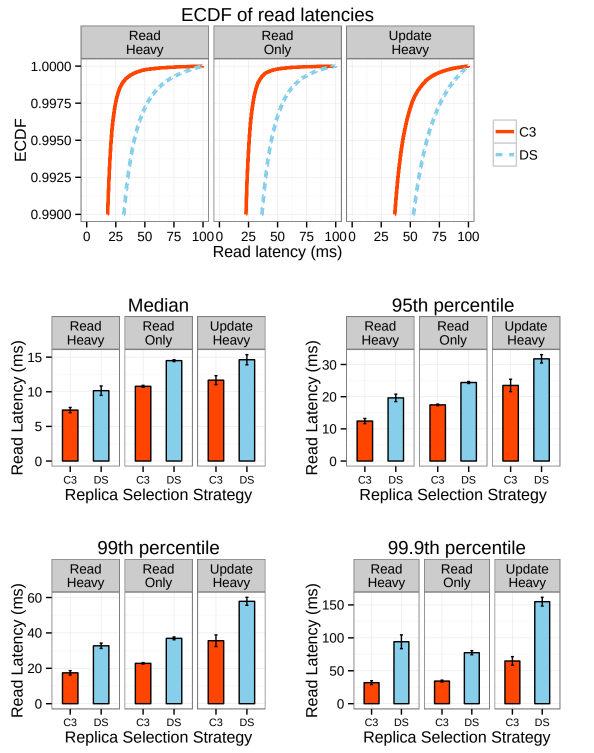


Figure 6: Cassandra's latency characteristics when using Dynamic Snitching and C3. C3 significantly improves the tail latency under different workloads without compromising the median.

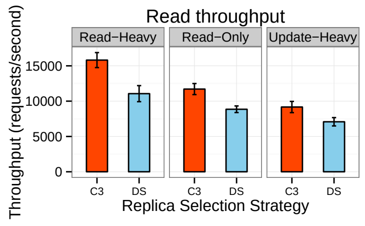
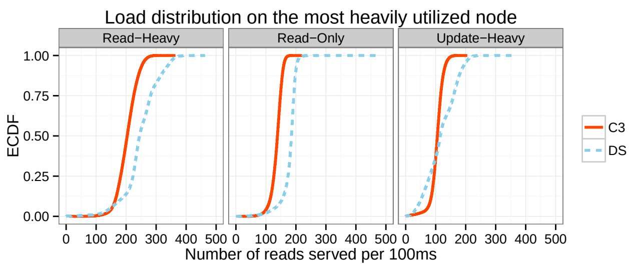


Figure 7: Throughput obtained with C3 and with Dynamic Snitching. C3 achieves higher throughput by better utilizing the available system capacity across replica servers.

the median is 24.5 ms with C3, whereas with DS, it is 83.91 ms: *more than 3x improvement*. In the update-heavy and read-only scenarios, C3 improves the same difference by a factor of 2.6 each. Besides the different percentiles, C3 also improves the mean latency by between 3 ms and 4 ms across all scenarios.

Impact of workload on read throughput: Figure 7 indicates the measured throughputs for C3 versus DS. By virtue of controlling waiting times across the replicas, C3 makes better use of the available system capacity, resulting in an increase in throughput across the considered workloads. In particular, C3 improves the throughput by



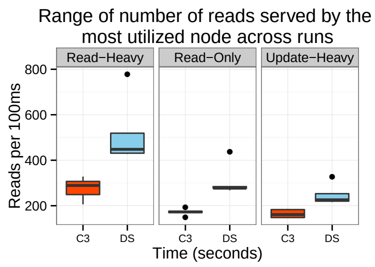


Figure 8: Aggregated distribution of number of reads serviced per 100 ms, by the most heavily loaded Cassandra node per run. With C3, the most heavily utilized node has a lower range in the load over time, wherein the difference between the 99th percentile and median number of requests served in 100 ms is lower than with Dynamic Snitching.

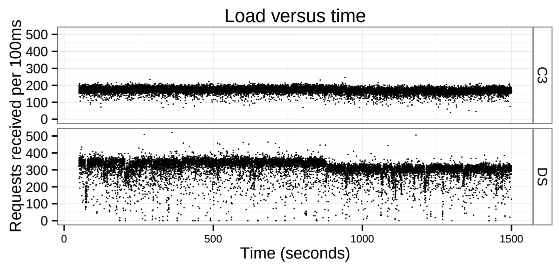


Figure 9: Example number of reads received by a single Cassandra node, per 100ms. With C3 (top), Cassandra coordinators internally adjust sending rates to match their peers' perceived capacity, leading to a smoother load profile free of oscillations. The per-server load is lower in C3 also because the requests are spread over more servers compared to DS (bottom).

between 26% and 43% across the considered workloads (update-heavy and read-heavy workloads respectively). We also note that the difference in throughput between the read- and update-heavy workloads of roughly 75% (across both strategies) is consistent with publicly available Cassandra benchmark data [18].

Impact of workload on load-conditioning: We now verify whether C3 fulfills its design objective of avoiding load pathologies. Since the key access pattern of our workloads are Zipfian distributed, we observe the load over time of the node that has served the highest number of reads across each run, that is, the most heavily utilized node. Figure 8 represents the distribution of the number of reads served per 100 ms by the most heavily utilized node in the cluster across runs. Note that *despite improving the overall system throughput*, the most heavily utilized node in C3 serves fewer requests than with DS. As a further confirmation of this, we present an example load profile as produced by C3 on highly utilized nodes (Figure 9). Unlike with DS, we do not see synchronized load-spikes when using C3, evidenced by the lack of oscillations and synchronized vertical bursts in the time-series. Furthermore, given that C3's rate control absorbs and distributes bursts carefully, it leads to a smoother load-profile wherein samples of the load in a given interval are closer to the system's true capacity unlike with DS.

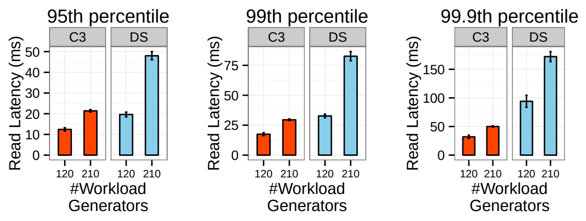


Figure 10: Overall performance degradation when increasing the number of workload generators from 120 to 210.

Performance at higher system utilization: We now compare C3 with DS to understand how the performance of both systems degrade with an increase in overall system utilization. We increase the number of workload generators from 120 to 210 (an increase of 75%). Figure 10 presents the tail latencies observed for the read-heavy workload. For a 75% increase in the demand, we observe that C3's latency profile, even at the 99.9^{th} percentile, degrades proportionally to the increase in system load. With DS, the median and 99.9^{th} percentile latencies degrade by roughly 82%, whereas the 95^{th} and 99^{th} percentile latencies *degrade by factors of up to 150%*. Furthermore, the mean latency with Dynamic Snitching is *70% higher* than with C3 under the higher load.

Adaptation to dynamic workload change: We now evaluate a scenario wherein an update-heavy workload enters a system where a read-heavy workload is already active, and observe the effect on the latter's read latencies. The experiment begins with 80 generators running a read-heavy workload against the cluster. After 640 s, an additional 40 generators enter the system, issuing update-heavy workloads. We observe the latencies from the perspective of the read-heavy generators around the 640 s mark. Figure 11 indicates a time-series of the latencies contrasting C3 versus DS. Each plot represents

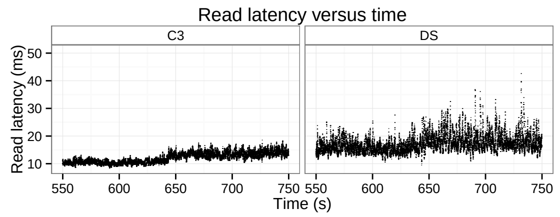


Figure 11: Dynamic workload experiment. The moving median over the latencies observed by the read-heavy generators from a run each involving C3 (left) and DS (right). At time 640 s, 40 new generators join the system and issue update-heavy workloads. With C3, the latencies degrade gracefully, whereas DS fails to avoid latency spikes.

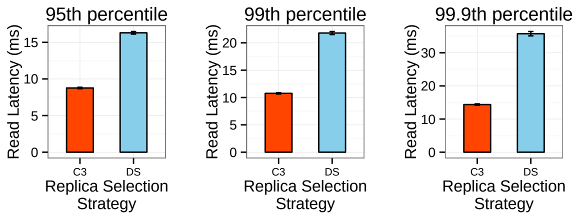


Figure 12: Results when using SSDs instead of spinning-head disks.

a 50-sample wide moving median[3] over the recorded latencies. Both DS and C3 react to the new generators entering the system, with a degradation of the read latencies observed at the 640 s mark. However, in contrast to DS, C3's latency profile degrades gracefully, evidenced by the lack of synchronized spikes visible in the time-series as is the case with DS.

Skewed record sizes: So far, we considered fixed-length records. Since C3 relies on per-request feedback of the service times in the system, we observe whether variable length records may introduce any anomalies in the control loop. We use YCSB to generate a similar dataset as before, but where field sizes are Zipfian distributed (favoring shorter values). The maximum record length is 2KB, with each record comprising the key, and ten fields. Again, C3 improves over DS along all the considered latency metrics. In particular, with C3, the 99^{th} percentile latency is just under 14 ms, whereas that of DS is close to 30 ms; *more than 2x improvement*.

Performance when using SSDs: As a further demonstration of C3's generality, we also perform measurements with m3.xlarge instances, which are backed by two 40 GB SSD disks. We configured a RAID0 array encompassing both disks. We reduced the dataset size to 150 million 1KB records in order to ensure that the dataset fits the reduced disk capacities of all nodes. Given that with SSDs, the system can sustain a higher workload, we used 210 read-heavy generators (70 threads per YCSB instance). Figure 12 illustrates the latency improvements obtained when using C3 versus DS with SSD backed instances. Even under the higher load, both algorithms have significantly lower latencies than when using spinning head disks. However, C3 again improves the 99.9^{th} percentile latency by *more than 3x*. Furthermore, the difference between the 99^{th} and 99.9^{th} percentile latencies in C3 is *under 5 ms*, whereas with DS, it is on the order of 20 ms. Lastly, C3 also improves the average latency by roughly 3 ms, and increases the read throughput by 50% of that obtained by DS.

Comparison against request reissues: Cassandra has an implementation of speculative retries [16] as a means of reducing tail latencies. After sending a read request to a replica, the coordinator waits for the response until a configurable duration before reissuing the request to another replica. We evaluated the performance of DS with speculative retries, configured to fire after waiting until the 99^{th} percentile latency. However, we observed that latencies actually degraded significantly after making use of this feature, up to a factor of 5 at the 99^{th} percentile. We attribute this to the following cause: in the presence of highly variable response times across the cluster (already due to DS), coordinators potentially speculate too many requests. This increases the load on disks, further increasing seek latencies. Due to this anomaly, we did not perform further experiments. We however, leave a note of caution that speculative retries are not a silver bullet when operating a system at high utilization [48].

Sending rate adaptation and backpressure over time: Lastly, we turn to a seven-node Cassandra cluster in our local testbed to depict how nodes adapt their sending rates over time. Figure 13 presents a trace of the sending rate adaptation performed by two coordinators against a third node (*tracked node*). During the run, we artificially inflated the latencies of the tracked node thrice (using the Linux `tc` utility), indicated by the drops in throughput in the interval (45, 55) s, as well as the two shorter drops at times 59 s and 67 s. Observe that both coordinators' estimations of their peer's capacity agree over time. Furthermore, the figure depicts all three rate regimes of the cubic rate control mechanism. The points close to 1 on the y-axis are arrived at via the multiplicative decrease, causing the system to enter the low-rate regime. At that point, C3 aggressively increases its rate to be closer to the tracked saturation rate, entering the saddle region (along the smoothened median). The stray points above the smoothened median are points where C3 optimistically probes for more capacity. During this run,

[3]A moving median is better suited to reveal the underlying trend of a high-variance time-series than a moving average [7]

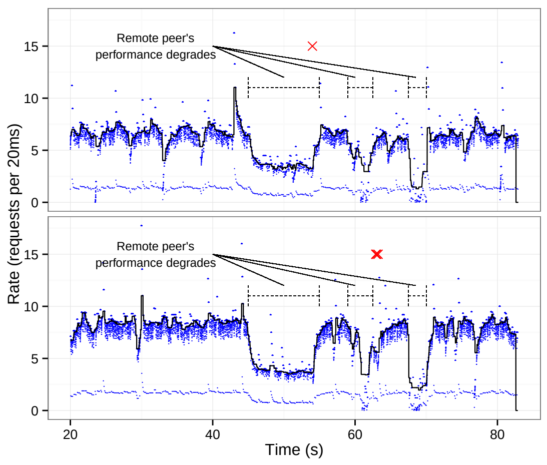


Figure 13: Sending rate adaptation performed by two coordinators against a third server. The receiving server's latency is artificially inflated thrice. The blue dots represent the sending-rates as adjusted by the cubic rate control algorithm, the black line indicates a moving median of the sending rates, and the red X marks indicate moments when affected replica group schedulers enter backpressure mode.

the backpressure mechanism fired 4 times (3 of which are very close in time) across both depicted coordinator nodes. Recall that backpressure is exerted when *all* replicas of a replica group have exceeded their rate limits. When the tracked node's latencies are reset to normal, the YCSB generators throttle up, sending a heavy burst in a short time interval. This causes a momentary surge of traffic towards the tracked node, forcing the corresponding replica selection schedulers to apply backpressure.

6 Evaluation Using Simulations

We turn to simulations to further evaluate C3 under different scenarios. Our objective is to study the C3 scheme independently of the intricacies of Cassandra and draw more general results. Furthermore, we are interested in understanding how the scheme performs under different operational extremes. In particular, we explore how C3's performance varies according to *(i)* different frequencies of service time fluctuations, *(ii)* lower utilization levels, and *(iii)* under skewed client demands.

Experimental setup: We built a discrete-event simulator[4], wherein workload generators create requests at a set of clients, and the clients then use a replica selection algorithm to route requests to a set of servers. A request generated at a client has a uniform probability of being forwarded to any replica group (that is, we do not model keys being distributed across servers according to consistent hashing as in Cassandra). The workload generators create requests according to a Poisson arrival process, to mimic arrival of user requests at web servers [35]. Each server maintains a FIFO request queue. To model concurrent processing of requests, each server can service a tunable number of requests in parallel (4 in our settings). The service time each request experiences is drawn from an exponential distribution (as in [48]) with a mean service time $\mu^{-1} = 4$ ms. We incorporated time-varying performance fluctuations into the system as follows: every T ms (*fluctuation interval*), each server, independently and with a uniform probability, sets its service rate either to μ or to $\mu \cdot D$, where D is a range parameter (thus, a bimodal distribution for server performance [41]). We set the D parameter to 3 (qualitatively, our results apply across multiple tested values of D, which we omit for brevity). The request arrival rate corresponds to 70% (high utilization scenario) and 45% (low utilization scenario) of the average service rate of the system, considering the time-varying nature of the servers' performance (that is, as if the service rate of each server's processor was $(\mu + D \cdot \mu)/2$). As with our experiments using Cassandra, we use a read-repair probability of 10% and a replication factor of 3, which further increases the load on the system. We use 200 workload generators, 50 servers, and vary the number of clients from 150 to 300. We set the one-way network latency to 250 μs. We repeat every experiment 5 times using different random seeds. 600,000 requests are generated in each run.

We compare C3 against three strategies:

1. **Oracle (ORA):** each client chooses based on perfect knowledge of the instantaneous q/μ ratio of the replicas (no required feedback from servers).
2. **Least-Outstanding Requests (LOR):** each client selects a replica to which it has sent the least number of requests so far.
3. **Round-Robin (RR):** as in C3, each client maintains a per-replica rate limiter. However, here it uses a round-robin scheme to allocate requests to replicas in place of C3's replica ranking. This allows us to evaluate the contribution of just rate limiting to the effectiveness of C3.

We also ran simulations of strategies such as uniform random, least-response time, and different variations of weighted random strategies. These strategies did not fare well compared to *LOR*, and due to space limits, we do not present results for them. We do not model disk activity in the simulator, and thus avoid comparing against Dynamic Snitching (since it relies on gossiping disk `iowait` measurements).

Impact of time-varying service times: Given that C3 clients rely on feedback from servers, we study the effect

[4]Code at `https://github.com/lalithsuresh/absim`.

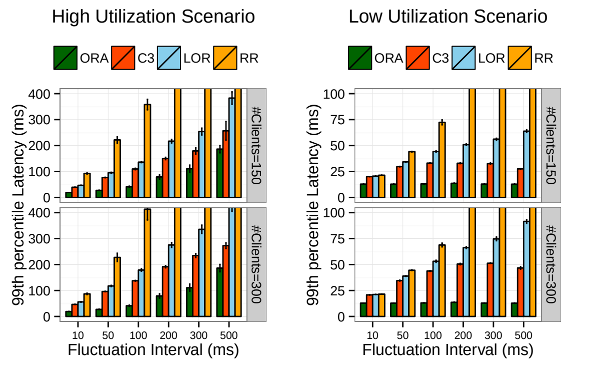


Figure 14: Impact of time-varying service times at high-utilization and low-utilization scenarios. Bars exceeding 400 ms are not shown.

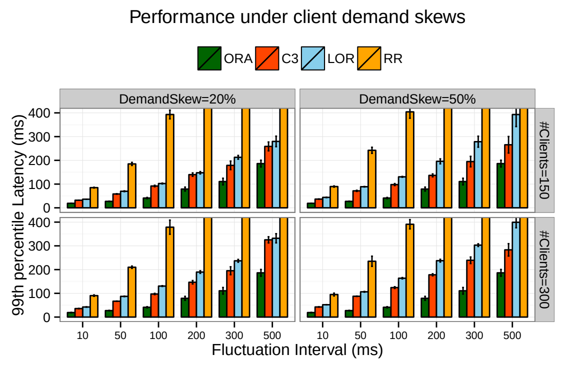


Figure 15: Impact of demand skews: 20% and 50% of the clients generate 80% of the requests to the servers. Bars exceeding 400 ms are not shown.

of the service-time fluctuation frequency on C3's control over the tail latency. Figure 14 presents the 99^{th} percentile tail-latencies when using C3 with 150 and 300 clients. When the average service times of the servers in the system change every 10 ms, C3 performs similarly to *LOR* and *RR*. This is expected, because at such a high frequency of performance variability, clients can make use of one round-trip's worth of feedback for at most another request, before that information is stale. However, as the interval between service-time changes increases, *LOR*'s performance degrades more compared to that of C3. Furthermore, the performance of *RR* suggests that rate-limiting alone does not improve the latency tail. This is because *RR* does not proactively prefer faster servers.. We also note that C3's performance remains relatively close to that of the *ORA*.

Performance at low utilization: While C3 is geared towards high-utilization environments with a number of requests in flight [35,42], we now demonstrate the efficacy of C3 under low-utilization settings as well. We set the arrival rate to match a 45% system utilization. While the performances of *LOR* and *RR* degrade with higher fluctuation intervals, C3's performance begins to plateau instead. This is because a client using *LOR*, will allocate requests to slow servers as long as it has assigned more requests to other replicas. This leads to poor allocations as initially explained in Figure 1. Thus, the longer a server remains bad, the higher the chance that it will receive some requests when clients use the *LOR* strategy. On the other hand, C3 explicitly aims to equalize the product of the queue-size and μ^{-1} across servers, and thus does not use slow servers any more than required to balance the latency distribution. We reiterate that while the average service times of a server may change more slowly, the response times are still subject to the dynamic waiting times as a result of queuing at the server.

Performance under heavy demand-skews: Lastly, we study the effect of heavy demand skews on the observed latencies. Figure 15 presents results when 20% and 50% of C3 clients generate 80% of the total demand towards the servers, respectively. Again, regardless of the demand skew, C3 outperforms *LOR* and *RR*.

7 Discussion

How general is C3? C3 combines two mechanisms in order to carefully manage tail latencies in a distributed system: ***(i)*** a load-balancing scheme that is informed by a continuous stream of in-band feedback about a server's load, and ***(ii)*** distributed rate-control and backpressure. We believe that the ideas discussed here can be applied to any low-latency data store that can benefit from replica diversity. Furthermore, our simulations compared C3 against different replica selection mechanisms, and allowed us to decouple the workings of the algorithms themselves from the intricacies of running them within a complex system such as Cassandra. That said, we are currently porting C3 onto systems such as MongoDB and token-aware Cassandra clients such as Astyanax [8] (which will avoid the problem of clients selecting overloaded coordinators).

Long-term versus short-term adaptations: A common recommended practice among operators is to over-provision distributed systems deployed on cloud platforms in order to accommodate performance variability [37]. Unlike application servers, storage nodes that handle larger-than-memory datasets are not easily scaled up or down; adding a new node to the cluster and the subsequent re-balancing of data are operations that happen over timescales of hours. Such questions of provisioning sufficient capacity for a demand is orthogonal to our work; our objective with C3 is to carefully utilize *already provisioned* system resources in the face of performance variability over short timescales.

Strongly consistent reads: Our work has focused on selecting one out of a given set of replicas, which inherently assumes eventual consistency. This applies to common use-cases at large web-services today, including Facebook's accesses to its social graph [47], and most of Netflix's Cassandra usage [24]. However, it remains to be seen how our work can be applied to strongly consistent reads as well. In particular, the gains in such a scenario depend on the synchronization overhead of the respective read protocol, and the effect of a straggler cannot be easily avoided.

8 Related Work

Dean and Barroso [16] described techniques employed at Google to tolerate latency variability. They discuss short-term adaptations in the form of request reissues, along with additional logic to support preemption of duplicate requests to reduce unacceptable additional load. In D-SPTF [30], a request is forwarded to a single server. If the server has the data in its cache, it will respond to the query. Otherwise, the server forwards the request to all replicas, which then make use of cross-server cancellations to reduce load as in [16]. Vulimiri *et al.* [48] also make use of duplicate requests. They formalize the different threshold points at which using redundancy aids in minimizing the tail. In the context of Microsoft Azure and Amazon S3, CosTLO [49] also presents the efficacy of duplicate requests in coping with performance variability. In contrast to these works, our approach does not rely on redundant requests and is in essence complementary to the above in that request reissues could be introduced atop C3.

Kwiken [23] decomposes the problem of minimizing end-to-end latency over a processing DAG into a manageable optimization over individual stages, wherein the latency reduction techniques (e.g., request reissues) are complementary to our approach.

Pisces [42] is a multi-tenant key-value store architecture that provides fairness guarantees between tenants. It is concerned with fair-sharing the data-store and presenting proportional performances to different tenants. PARDA [21] is also focused on the problem of sharing storage bandwidth according to proportional-share fairness. Stout [31] uses congestion control to respond to to storage-layer performance variability by adaptively batching requests. PriorityMeister [53] focuses on providing tail latency QoS for bursty workloads in shared networked storage by combining priorities and rate limiters. As in C3, these works make use of TCP-inspired congestion control techniques for allocating storage resources across clients. While orthogonal to the problem of replica selection, we are planning to investigate the ideas embodied in these works within the context of C3. Pisces recognizes the problem of weighted replica selection but employs a round robin algorithm similar to the one used in our simulation results.

Mitzenmacher [33] showed that allowing a client to choose between two randomly selected servers based on queue lengths exponentially improves load-balancing performance over a uniform random scheme. This approach is embodied within systems such as Sparrow [36]. However, in our settings, replication factors are typically small compared to cluster size. Given a common replication factor of 3, ranking 3 servers instead of 2 only incurs a negligible overhead. Moreover, the basic power of two choices strategy does not include a rate limiting component to avoid exceeding server capacities, in contrast to C3. A thorough comparison between the two approaches is left for future work.

Lastly, there is much work in the cluster computing space on skew-tolerance [4, 19, 27, 44, 51]. In contrast to our work, cluster jobs operate at timescales of at least a few hundreds of milliseconds [36], if not minutes or hours.

9 Conclusion

In this paper, we highlighted the challenges involved in making a replica selection scheme explicitly cope with performance fluctuations in the system and environment. We presented the design and implementation of C3. C3 uses a combination of in-band feedback from servers to rank and prefer faster replicas along with distributed rate control and backpressure in order to reduce tail latencies in the presence of service-time fluctuations. Through comprehensive performance evaluations, we demonstrate that C3 improves Cassandra's mean, median and tail latencies (by up to 3 times at the 99.9^{th} percentile), all while increasing read throughput and avoiding load pathologies.

Acknowledgments: We thank our shepherd Lidong Zhou and the anonymous reviewers for their feedback. We are grateful to Sean Braithwaite, Conor Hennessy, Axel Liljencrantz, Jimmy Mårdell, and Radovan Zvonček for their valuable inputs and suggestions. This paper benefited greatly from discussions with Ramin Khalili, Georgios Smaragdakis and Olivier Bonaventure. This research is (in part) supported by European Union's Seventh Framework Program FP7/2007-2013 under the BigFoot project (grant agreement 317858) and the Trilogy2 project (grant agreement 317756), and by an AWS in Education Grant award.

References

[1] Akka. http://akka.io/, accessed Sept 25, 2014.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] Amazon ELB. http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/TerminologyandKeyConcepts.html, accessed Sept 24, 2014.

[4] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *OSDI*, 2010.

[5] Apache Cassandra. http://cassandra.apache.org/, accessed June 10, 2013.

[6] Apache Cassandra Use Cases. http://planetcassandra.org/apache-cassandra-use-cases/, accessed Sept 25, 2014.

[7] G. R. Arce. *Nonlinear Signal Processing: A Statistical Approach*. Wiley, 2004.

[8] Astyanax. https://github.com/Netflix/astyanax, accessed Jan 5, 2015.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[10] C. F. Bispo. The single-server scheduling problem with convex costs. *Queueing Systems*, 73(3), 2013.

[11] J. Brutlag. Speed Matters, accessed Sept 24, 2014. http://googleresearch.blogspot.com/2009/06/speed-matters.html.

[12] Cassandra Documentation. http://www.datastax.com/documentation/cassandra/2.0, accessed Sept 25, 2014.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2), June 2008.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[15] DB-Engines Ranking of Wide Column Stores. http://db-engines.com/en/ranking/wide+column+store, accessed Sept 25, 2014.

[16] J. Dean and L. A. Barroso. The Tail At Scale. *Communications of the ACM*, 56:74–80, 2013.

[17] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[18] J. Ellis. How not to benchmark Cassandra: a case study, 2014. http://www.datastax.com/dev/blog/how-not-to-benchmark-cassandra-a-case-study.

[19] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.

[20] W. D. Gray and D. Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of Experimental Psychology: Applied*, 6, 2000.

[21] A. Gulati, I. Ahmad, and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST*, 2009.

[22] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev.*, 42(5), 2008.

[23] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up Distributed Request-Response Workflows. In *SIGCOMM*, 2013.

[24] C. Kalantzis. Eventual Consistency != Hopeful Consistency, talk at Cassandra Summit, 2013. https://www.youtube.com/watch?v=A6qzx_HE3EU.

[25] C. Kalantzis. Revisiting 1 Million Writes per second, 2014. http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html.

[26] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim. Measuring Interference Between Live Datacenter Applications. In *SC*, 2012.

[27] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *SoCC*, 2012.

[28] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *SoCC*, 2014.

[29] A. Liljencrantz. How Not to Use Cassandra, talk at Cassandra Summit, 2013. https://www.youtube.com/watch?v=0u-EKJBPrj8.

[30] C. R. Lumb and R. Golding. D-SPTF: Decentralized Request Distribution in Brick-based Storage Systems. *SIGOPS Oper. Syst. Rev.*, 38(5):37–47, Oct. 2004.

[31] J. C. McCullough, J. Dunagan, A. Wolman, and A. C. Snoeren. Stout: An Adaptive Interface to Scalable Cloud Storage. In *USENIX ATC*, 2010.

[32] M. Mitzenmacher. How Useful Is Old Information? *IEEE Trans. Parallel Distrib. Syst.*, 11(1), Jan. 2000.

[33] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10), Oct. 2001.

[34] Nginx. http://nginx.org/en/docs/http/load_balancing.html, accessed Sept 24, 2014.

[35] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *NSDI*, 2013.

[36] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *SOSP*, 2013.

[37] Riak. AWS Performance Tuning, accessed Sept 24, 2014. http://docs.basho.com/riak/latest/ops/tuning/aws/.

[38] Riak. Load Balancing and Proxy Configuration, accessed Sept 24, 2014. http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/.

[39] M. Roussopoulos and M. Baker. Practical Load Balancing for Content Requests in Peer-to-Peer Networks. *Distributed Computing*, 18(6), 2006.

[40] S. Sanfilippo. Redis latency spikes and the 99th percentile, 2014. http://antirez.com/news/83.

[41] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *VLDB Endowment*, 3(1-2), Sept. 2010.

[42] D. Shue, M. J. Freedman, and A. Shaikh. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *OSDI*, 2012.

[43] S. Souders. Velocity and the Bottom Line, 2009. `http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html`.

[44] C. Stewart, A. Chakrabarti, and R. Griffith. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *USENIX ICAC*, 2013.

[45] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah. Serving Large-scale Batch Computed Data with Project Voldemort. In *FAST*, 2012.

[46] J. A. van Mieghem. Dynamic Scheduling with Convex Delay Costs: The Generalized $c|mu$ Rule. *The Annals of Applied Probability*, 5, 1995.

[47] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook Serves the Social Graph. In *SIGMOD*, 2012.

[48] A. Vulimiri, P. B. Godfrey, R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker. Low Latency via Redundancy. In *CoNEXT*, 2013.

[49] Z. Wu, C. Yu, and H. V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *NSDI*, 2015.

[50] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *NSDI*, 2013.

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

[52] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *EuroSys*, 2013.

[53] T. Zhu, A. Tumanov, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *SoCC*, 2014.