Targeting Tail Latency in Replicated Systems with Proactive Rejection

Laura Lawniczak
lawniczak@cs.fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Germany

Tobias Distler
distler@cs.fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Germany

Abstract

When put under stress, traditional state-machine replication protocols typically exhibit response times that by far exceed the average level of normal-case operation. The common way to mitigate such overload-induced tail latency is to overprovision computing and network resources. However, this method often leads to large amounts of resources left unused over extended periods of time, especially in application scenarios in which high loads are mostly limited to short phases. In this paper, we circumvent the need for overprovisioning with Idem, a replication protocol specifically designed to process client requests with low latency even during load spikes. Most notably, Idem replicas avoid overload by proactively rejecting requests in a collaborative manner. In contrast to centralized overload-prevention strategies, the collaboration among replicas allows Idem to always timely notify clients about rejections of their requests, not only under favorable conditions but also in the presence of replica crashes.

CCS Concepts

• **Computer systems organization → Reliability**.

Keywords

Crash fault tolerance, state-machine replication, tail latency, overload prevention, proactive rejection.

ACM Reference Format:
Laura Lawniczak and Tobias Distler. 2024. Targeting Tail Latency in Replicated Systems with Proactive Rejection. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3700775

1 Introduction

State-machine replication [63] enables a system to tolerate server crashes by maintaining the state of an application on multiple machines and keeping it consistent using a replication protocol [44, 56]. Providing fault tolerance for latency-sensitive applications (e.g., navigation services, key-value stores, or online games) requires replicated systems that are able to consistently maintain low and stable response times. In practice, accomplishing this goal is inherently difficult due to the phenomenon of tail latency [26], which refers to the existence of cases or phases over the course of a system's lifetime in which the achieved latency exceeds the average response time by multiple factors or even magnitudes.

With progress depending on only a quorum of replicas (instead of all servers), replicated systems in general have advantages when it comes to masking disruptive factors that typically slow down individual replicas at a time (e.g., hardware issues) [26]. However, the same does not apply to causes of tail latency that affect the entire replica group, such as periods of overload during which a large number of requests are concurrently issued to the replicated service.

Unfortunately, traditional replication-protocol designs [44, 56] usually do not take overload scenarios into account. As a consequence, replicated systems commonly provide two distinct tiers with regard to their service quality: (1) During times of low and medium utilization, clients on average experience comparably low response times. (2) Once the demand exceeds the saturation point, requests start to queue up and as a result end-to-end latency skyrockets [3, 62], thereby severely impacting clients across the board. The traditional way to reduce the risk of the latter is overprovisioning computing and network resources [26]. However, this is not only costly but also wastes significant amounts of resources in the common case of the overload being limited to relatively short phases, with long stages of lower utilization in between.

In this paper, we address the problem of load-induced tail latency in replicated systems by presenting Idem, a state-machine replication protocol that effectively prevents overload and hence offers clients a third option. Specifically, when the utilization of the replicated service is high, Idem replicas proactively (and with low latency) reject some of the incoming requests to keep the load at a manageable level without the need for overprovisioning resources.

As a special property, Idem's overload-prevention mechanism operates in a *collaborative* manner, meaning that replicas reach individual decisions on the rejection of requests, instead of delegating this task to a dedicated load balancer or replica (e.g., the leader). Thus, in contrast to solutions that only put a particular entity in charge of preventing overload, Idem's collaborative method does not comprise a single point of failure and is able to continuously deliver rejection notifications with low latency, even during leader crashes.

While being effective in many applications, our approach reaches its full potential in conjunction with *semi-autonomous clients*. Such clients achieve their highest degree of efficiency when using the replicated service, but are equipped with a (less powerful) local fallback procedure they can resort to in case their requests are rejected. Typical examples of this kind of procedure include auxiliary navigation subsystems in unmanned vehicles [36, 65] or movement prediction algorithms in multiplayer online games [69].

In particular, this paper makes the following contributions: (1) It presents Idem, a crash-tolerant replication protocol that relies

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

on collaborative overload prevention to deal with load-induced tail latency. (2) It shows how to implement our collaborative approach in such a way that it effectively keeps response times stable, is robust against replica crashes, and incurs only low overhead. (3) It experimentally evaluates IDEM in comparison with state-of-the-art replication protocols.

2 System Model

As shown in Figure 1, we target distributed systems that are composed of two parts: (1) a replicated service at the core and (2) a (potentially large) number of semi-autonomous clients at the edge that regularly access the replicated service (sending one request at a time) to perform their tasks. The provided application is time-sensitive, so the service's results are only of use if clients receive them in a timely manner. However, if necessary clients can fallback on internal computation to achieve an adequate, if inferior, result, for example when the replicated system is unavailable.

2.1 Replicated Service at the Core

Our approach targets data-center environments in which the network commonly experiences long phases of synchrony. Nodes, that is replicas and clients, are linked via fair-loss point-to-point connections [16], meaning that message losses and network partitions may occur, but (by retransmitting messages) two correct nodes will eventually be able to communicate with each other. To ensure service availability, the system at the core relies on a state-machine replication protocol that tolerates up to f replica crashes, with f in our data-center use cases usually being low (e.g., $f \leq 2$).

When deploying such a protocol, resources need to be considered carefully. While dimensioning the system for the maximum load or even overprovisioning guarantees a good and stable performance, this strategy also often results in resources being wasted [21, 64]. Underprovisioning resources, on the other hand, for many applications leads to a better utilization and suffices most of the time, but it hinders a system from handling bursts of high loads.

2.2 Semi-Autonomous Clients at the Edge

The clients at the edge of our system comprise a mechanism that allows them to fallback to internal computation and retain basic functionality during phases in which, due to temporary network unreliability, they cannot reach the replicated service. While this allows clients to make rough decisions on their own, the outputs of the replicated service offer a significantly better quality of service. Additionally, the internal fallback computation has disadvantages such as a high resource consumption or bad user experience, so it should only be activated when necessary. Overall, this means that (1) timely results from the replicated service should be used if available and (2) it is beneficial for a client to know early whether the service cannot provide this timeliness, as the client then can promptly activate its fallback measures.

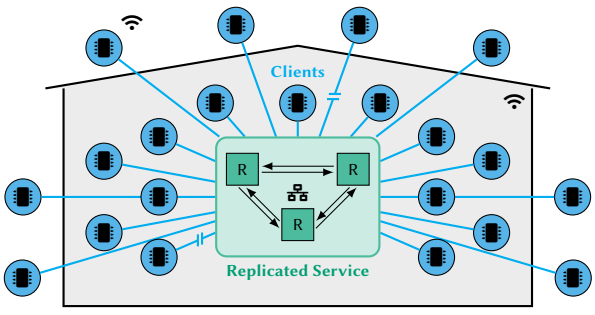


Figure 1: Basic system architecture

2.3 Example Applications

As shown by the following examples, several real-world application scenarios fit our targeted system model and architecture.

Robot Warehouse. In the first example, semi-autonomous robots transporting goods inside a warehouse act as clients, while a replicated service is responsible for route planning and coordination of the robots. For this purpose, the service maintains the last known position and destination of each robot and collects information about other important events (e.g., blockages or robot failures). With this holistic approach, it can route the robots very efficiently, minimizing congestion and travel-path length. However, as robots continue to move, the instructions of the service quickly become outdated and are of little to no use after a certain period of time. To tolerate phases of service unavailability, the robots are integrated with sensors such as Lidar [50] which allow them to navigate autonomously. Nevertheless, since a robot has no internal knowledge of the other robots' routes or of obstacles outside its own sensors' field of view, navigation relying on sensors is usually inferior to a timely result of the replicated service.

Live Data. To provide a seamless experience to users of Web applications such as chat programs or newsfeeds, short delays in service accesses should be masked by the client side (e.g., by temporarily displaying old data [67]), whereas long delays should be signaled to the user (e.g., via a loading animation or, in the worst case, an error message). In this context, it is essential for the client logic to be able to distinguish the first case from the second, as for example the frequent displaying of brief loading animations typically results in user frustration. Hence, there is generally great benefit in Web clients knowing early whether or not the replicated service is currently able to provide a timely result.

Massive Multiplayer Online Gaming. In massive multiplayer online gaming, only the servers know the exact positions and actions of all participants, but clients can use previous data to roughly predict player and object movement if there is no timely result from the service [69]. However, the movement prediction cannot consider sudden changes in direction and furthermore leads to additional computation than can decrease the performance of the game, and thus hamper the user experience. Consequently, this local fallback mechanism should only be resorted to when necessary. Besides, in online gaming there can be a very large number of concurrent clients (i.e., players) as well as a high fluctuation of clients and client numbers, as players may log in and out of a game frequently. Overall, this increases the danger of the server side being subjected to (sudden) overload bursts.

3 Problem Statement

In this section, we analyze the issues time-sensitive replicated applications face in the presence of overload. Furthermore, we outline our solution to address these problems.

3.1 State of the Art

To examine their behavior under load, we conducted experiments with different state-of-the-art state-machine replication protocols; for details please refer to Section 7. In all cases, the protocols showed a similar reaction as the one recorded from Paxos [43], which is plotted in Figure 2. As illustrated by these measurements, the quality of service perceived by clients can be categorized into two tiers: (1) If the system is in a healthy state and faces small or medium load (marked by the green area), clients receive timely responses from the replicated service. We define this as the *good tier*. (2) In an overload situation, however, the latency in these systems can be dramatically higher than expected. A client will either get a late result that is no longer of use, or receive no response at all before running into a timeout. Such issues are characteristic of the red area and hence referred to as *bad tier*.

3.2 Challenge: Improving the Bad Case

Our goal in this paper is to offer clients of replicated systems a new *middle tier* with regard to the quality of service. By providing clients with timely notifications if the system is currently under high load and unable to provide a timely result, clients knows early enough when to switch to their own fallback mechanisms to still provide an adequate result. With this, we aim to minimize the bad case when clients obtain no useful results at all. We achieve this by introducing *proactive rejection*, a systematic way to monitor and actively reject requests before a system reaches an overload state, thereby allowing a system to maintain more stable response times.

Please notice that our focus in this paper is on **preventing overload-induced tail latency**, meaning that our approach is not designed to tackle other causes of tail latency (e.g., delays during the request execution at the application level). Consequently, we do not guarantee hard worst-case bounds on the provided latency, which for our clients does not pose a problem, because they can always resort to their fallback mechanism. Also, although the names may suggest otherwise, proactive rejection is not related to the concept of abortable consensus [8], as further discussed in Section 8.

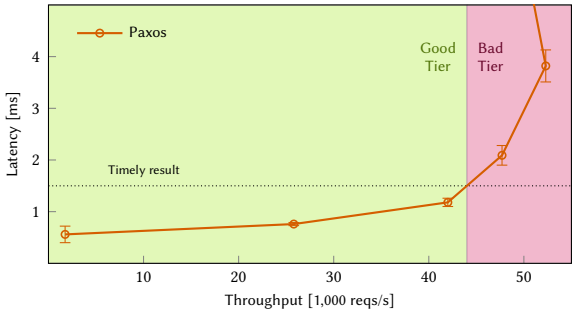


Figure 2: Behavior of existing replication protocols. Data points depict the average latency and standard deviation.

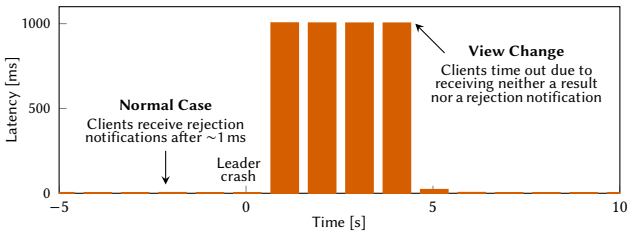


Figure 3: Impact of a leader crash on rejections in Paxos$_{LBR}$.

3.3 Why Not Simply Let the Leader Reject?

With the leader replica in protocols such as Paxos coordinating the consensus process for requests, our first intuition was to implement proactive rejection by putting the leader in charge of deciding which requests to reject. Although effective during normal-case operation, this leader-based rejection (LBR) unfortunately turned out to have a major drawback when it comes to view changes. Specifically, as shown in Figure 3, a crash of the leader means that clients neither receive a result nor a rejection notification until the view change is complete and clients performed a successful fail-over to the new leader. Given this significant period of uncertainty, such system behavior falls into the bad tier and hence rules out LBR as technique for our target use cases, for which we strive for a more robust solution.

Notice that for several reasons it is not straightforward to address the view-change problem by extending LBR to involve additional replicas: (1) Follower replicas in crash-tolerant protocols (during normal-case operation) typically do not directly interact with clients and thus only have limited knowledge about the amount of incoming requests. Similarly, resorting to a rotating-leader protocol [5, 53, 73] is not an option, as the continuous leader changes make it inherently difficult to assess the anticipated load of the entire system at a single replica. (2) Independent of whether the leader of the next view is chosen via election [56] or statically defined by the view number [43], a replica can only become the new leader once a majority of replicas in the system considers a view change to be necessary. That is, a new leader must not decide on rejecting client requests until having confirmation that a view change will actually take place. (3) View-change timeouts usually represent comparably large periods of time (e.g., Ongaro et al. [56] recommend the view-change timeout to be at least a magnitude larger than the broadcast time, other replicated systems use view-change timeouts of multiple seconds [11, 40]). Thus, any attempts to extend LBR to other replicas either result in significant delays (in case the timeout is respected) or are no longer leader-based (in case a follower acts before the timeout expired).

To avoid the issues associated with leader-based strategies, we design proactive rejection as a *collaborative approach* in which each replica monitors the system's load independently and for every request decides whether to accept or reject it. As a key benefit, this enables a replicated service to provide clients with timely rejection notifications even during view changes.

3.4 Requirements

Leveraging the insights gained from our problem analysis in the previous sections, we identify the following requirements for a mechanism providing proactive rejection:

- **Effectiveness.** Whenever its replicated service is subjected to request bursts, the rejection mechanism should preserve the system from reaching an overload-state, thereby avoiding overload-induced tail latency. Ideally, this goal is achieved while at the same time sustaining a high throughput of successfully processed client requests.
- **Robustness.** To give clients the chance of a prompt reaction, the mechanism should deliver rejection notifications in a timely and stable manner. Apart from normal-case operation, this should also be true under disruptive conditions (e.g., view changes, misconfigured rejection thresholds, phases of extreme load). This implies that the mechanism must operate in a decentralized manner and not comprise a single point of failure such as a dedicated replica or external load balancer.
- **Low Overhead.** Taking effect at high load levels, the mechanism itself should incur as little overhead as possible, both in terms of performance as well as network traffic. In particular, the mechanism must not prevent the replicated system from reaching saturation.

In the next section, we provide details on how we satisfy these requirements in Idem by directly incorporating proactive rejection into the design of a replication protocol.

4 Idem

This section presents Idem, a crash-fault tolerant state-machine replication protocol specifically designed to reduce overload-induced tail latency by actively rejecting requests under high load.

4.1 Collaborative Overload Prevention

Compared with traditional state-machine replication protocols, in which the interaction between clients and replicas is typically limited to the exchange of requests and replies, Idem introduces a third type of message ⟨Reject, *id*⟩ which a replica can send to inform the client about the id of a submitted request that the replica has opted not to process any further. This decision is made by each replica individually (i.e., without coordinating with other replicas) and based on an acceptance test performed on each request.

As shown in Figure 4, after multicasting its request to all replicas, an Idem client waits for a response, which may come in one of two forms. If the request has been accepted (and therefore agreed on and ordered by Idem), the client is provided with a regular Reply and can complete the operation. On the other hand, in case the client receives Rejects from at least $n - f$ of the n replicas, it abandons the operation and resorts to a local fallback mechanism to handle the temporary unavailability of the replicated service. As described in Section 2, the clients in our target use cases are equipped with such fallback procedures, and thus remain operational even if some of their requests are rejected.

We use the term "collaborative" for our approach to emphasize the concerted effort of Idem's replicas in preventing overload, and to underline that the approach neither delegates the task of rejecting requests to a single replica nor requires replicas to reach consensus on rejection decisions. The latter is crucial because a solution to relieve pressure on an agreement protocol itself should not require additional agreement between replicas.

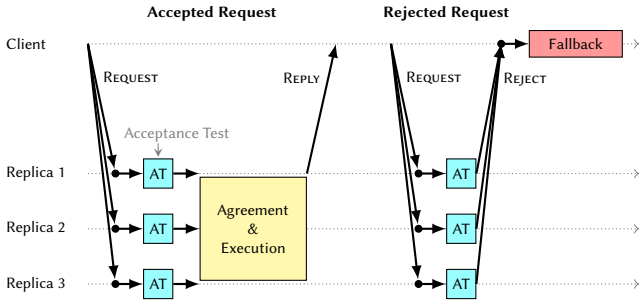


Figure 4: Collaborative overload prevention: Replicas use acceptance tests to decide whether or not to process requests.

Collaboration in Idem includes the guarantee that a request will be executed once it has been accepted by (at least) one correct replica, even if there are other replicas at which the acceptance test was negative. To ensure this, having accepted a request, a replica is responsible for keeping the request available in the system and (if necessary) relaying it to other replicas. Idem replicas meet this requirement by relying on a forwarding mechanism that incurs negligible overhead due to using lazy propagation, caching recently rejected requests, and offering to relay requests on demand. For better understandability, we defer the specifics of the forwarding mechanism to Section 5, in which we also discuss the semantics and provided guarantees of our collaborative overload prevention approach in a detailed and more formal manner.

4.2 Protocol Overview

Idem is a leader-based protocol that requires $2f + 1$ replicas to tolerate up to f replica crashes. As shown in Figure 5, it integrates collaborative overload prevention as a preceding protocol phase (Require), which is followed by a two-phase Paxos-style agreement process (Propose and Commit). Implementing overload prevention in the form of an individual phase offers the key benefit of applying a clean separation of concerns, leaving the properties of the agreement algorithm unaffected. Furthermore, it also makes it easier to combine our approach with other consensus protocols, however the specifics of doing so are outside the scope of this paper.

Idem clients access the replicated service by submitting their requests to all replicas in the system. This design decision has several advantages: (1) As explained in Section 4.1, it enables replicas to perform collaborative overload prevention by individually deciding on the acceptance of incoming client requests. (2) It avoids a common bottleneck [12] in many traditional replication protocols in which the leader is responsible for distributing requests to the rest of the replica group and hence the leader's network link is at risk to quickly become saturated, especially when the service workload consists of large requests. In contrast, with Idem clients directly sending their requests to all replicas, Idem replicas can perform the agreement on request ids instead of full requests (see Section 4.3). Due to request ids in most use cases being several magnitudes smaller than full requests, this approach mitigates another common cause of load-induced tail latency. (3) It allows Idem replicas to

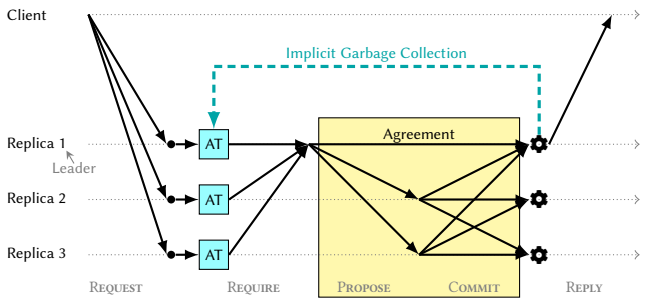


Figure 5: Protocol phases of Idem.

garbage-collect old requests and consensus state without the need for further coordination among each other (see Section 4.4), which is particularly beneficial during periods of high system load.

The following sections describe the individual protocol parts of Idem in more detail: First the request handling, and then the garbage collection and view-change mechanisms.

4.3 Request Handling

To issue a request, a client sends a ⟨Request, *id*, *command*⟩ message containing the actual command and a unique request id to all replicas. As is the case for many replication-protocol implementations [11, 43], Idem assumes that each client has at most one pending request at a time and that the request id consists of a tuple ⟨*cid*, *onr*⟩, with *cid* being a static client identifier and *onr* a client-specific operation number which increases for each new request the client submits. Consequently, replicas can use the operation number to distinguish a client's latest request from older ones.

Upon receiving a request, a replica uses a local acceptance test to decide whether to accept or reject the request based on the current load. This acceptance test is independent of the rest of the Idem protocol and can be tailored to specific use cases (see Section 5.1). If a replica rejects the request, it immediately sends a ⟨Reject, *id*⟩ message to the client, without taking further action. Otherwise, the replica stores the request locally and by means of Idem's forwarding mechanism ensures that eventually all replicas in the system learn about the accepted request (see Section 5.2). If a replica receives such a forwarded request, it accepts the request (if it has not already done so) regardless of the current load. As each replica accepts at most r client-issued requests at the same time, the total number of active requests in the system is limited to $r_{max} = n \times r$, with n being the total number of replicas.

After accepting a request (either from a client or via a forward), the replica sends a ⟨Require, *id*⟩ message with the request's id to the current leader. For each request id, the leader waits for $f + 1$ Requires before it proposes the request. This ensures that at least one correct replica has knowledge of the request and can potentially forward it to the other replicas, providing liveness in the presence of replica failures. Note that the leader itself does not need to be among those $f + 1$ replicas. To propose a request id, the leader of view v assigns it a unique sequence number *sqn* and sends a ⟨Propose, *id*, *sqn*, *v*⟩ message to all replicas.

Once a replica receives a Propose from the leader, it stores the message, sends a ⟨Commit, *id*, *sqn*, *v*⟩ message to all replicas, and waits for a total of $f+1$ Commits (including its own; the leader's proposal counts as a commit). Having obtained the necessary commits as well as the corresponding request (either directly or after a forward), a replica executes the request and (in case of the leader) sends a ⟨Reply, *id*, *result*⟩ to the client. Now, the request is no longer considered active and the replica can accept a new request in its stead.

4.4 Checkpointing and Garbage Collection

For improved efficiency, Idem executes multiple consensus instances in parallel, organized in a fixed-size window that defines the range of sequence numbers that a replica is currently working on. To shift the window forward, Idem relies on periodic checkpoints storing the current state of the application together with important metadata such as the highest executed request id for each client, which is used for duplicate detection. The creation of a checkpoint enables the garbage collection of the included requests and their corresponding consensus instances, allowing to move the window. These checkpoints can then be used to bring lagging replicas up to date without the need to replay the included requests.

While, in theory, requests can be deleted as soon as they are included in a checkpoint, both the creation and application of a checkpoint are often costly compared to the regular execution of a request. Hence, it is practical to wait until at least $f+1$ replicas have executed a request before deleting it, to ensure that sufficient up-to-date replicas can continue to participate in the consensus protocol (even in the presence of temporary message loss) without the need to apply a checkpoint. In Idem, replicas can determine this via the information about the currently active requests that is used for overload prevention, and which serves as implicit notification of the other replicas' progress, without the need for exchanging additional messages between replicas. For this purpose, the window size w must be at least as large as the number of concurrently allowed requests $r_{max} = n \times r$. Assuming this relation, a replica knows that it can discard old requests and consensus instances and move its window as soon as it receives new proposals (or commits) with a sequence number $sqn > sqn_{low} + r_{max}$, with sqn_{low} denoting the current start of the window. This indicates that at least $f + 1$ replicas must have executed these old requests (removing them from the number of active requests) in order to accept new ones.

As a key benefit, this approach permits Idem to omit superfluous checkpoint or progress messages during normal operation. Only when a replica is lagging behind, it explicitly asks another replica for the newest checkpoint to catch up.

4.5 View Change

As a leader-based protocol, Idem includes a view-change mechanism to reassign the leader role after a crash. To detect a crashed leader, each replica starts a view-change timer when a request has not been executed for a certain amount of time. This timer is stopped as soon as the replica perceives progress, for example by means of a new commit or checkpoint. If this is not the case and the timeout expires, however, the replica assumes that the current leader has crashed and abandons its current view v. From that point on, the replica ignores any messages concerning v and older views, so there will be

no more changes in its window regarding these views. Additionally, the replica restarts the view-change timer as a safeguard in case the current view change is not successful.

Having abandoned a view v, a replica requests a view change by multicasting a $\langle \text{ViewChange}, v_t, proposals \rangle$ message which includes the target view $v_t = v + 1$ and the current proposal window. The same happens when a replica receives $f + 1$ matching ViewChanges, indicating that the current view does not have enough support to maintain progress. The leader of the new view stores the latest ViewChange message of each replica, waits for $f + 1$ matches, and then merges them to update its own window, finally re-proposing the included requests for the new view.

5 Overload-Prevention Details

With collaborative overload prevention being Idem's core means to target overload-induced tail latency, in this section we provide further details on the mechanism. Apart from discussing design choices and possible optimizations, we also precisely define the semantics it provides to clients.

5.1 Acceptance Test

Whenever an Idem replica receives a new request from a client, it relies on an acceptance test to decide whether to accept or reject this request. We do not put any restrictions on how exactly this decision is reached and which checks or criteria the acceptance test might involve. As key benefit, this makes it possible to tailor the acceptance test to one's use-case application, system environment and workload characteristics.

Non-Determinism. In contrast to many other auxiliary functions used in replicated systems (e.g., methods to determine the state partition(s) accessed by a request during execution [29, 47]), the acceptance test in Idem does not have to be deterministic. That is, when a client invokes the same request on different replicas or at different points in time, each replica may return different results. This additional flexibility is enabled as a result of two specific properties of our approach: (1) The decision to accept or reject a particular request does typically not mainly depend on the request itself. Instead, it is commonly based on the question whether or not the system currently has enough resources to handle additional requests without overloading. (2) As a consequence of network asynchrony and non-uniform distances between clients and replicas, replicas in general receive requests at different points in time, and hence also perform the acceptance test on separate occasions. Without additional coordination between replicas, which as discussed in Section 4.1 would be impractical, this means that during periods of varying load replicas are likely to draw diverging conclusions anyway. Hence, we decided to enlarge the design space for acceptance tests and allow non-deterministic implementations.

Active Queue Management. In its basic form, it is sufficient for a replica to simply store accepted client requests in a queue while they are in progress, and reject all subsequent requests while the length of this queue reaches the configured threshold. In the context of queue management, this technique is known as "tail drop". While providing adequate performance, tail drop's efficiency significantly depends on the traffic variability and is especially problematic in high-load scenarios [13]. Hence, for our implementation of Idem, we took inspiration from another essential technique: active queue management [2]. Here, requests are not only dropped after the threshold has been reached but to a certain probability already before that, providing a more stable outcome.

Additionally, we want to help replicas reach a unanimous decision on the acceptance/rejection of requests, as this avoids scenarios in which requests are only accepted at a single replica and block a request slot until the timeout-based forwarding mechanism is triggered. To achieve this, our acceptance test uses the following rules: Each client is assigned a group number, with a maximum of r clients being in the same group (r the number of concurrently accepted client-issued requests). These groups are now assigned time slices with one group being prioritized in each slice. This way, only up to r clients are prioritized at the same time. When the number of active requests r_{now} reaches a certain threshold (e.g., 60% of r), a replica starts differentiating between prioritized and non-prioritized clients. If a request arrives from a prioritized client, it is treated as in tail drop and accepted unless the maximum is reached. Non-prioritized clients, however, use active queue management and are rejected with a probability $p = r_{now}/r$. Replicas employ a pseudo-random function with the same seed for each request, further increasing the chance of reaching the same decision.

While a perfect unanimity between replicas due to differences in local clocks, scheduling and message arrival is highly unlikely, this nevertheless assists replicas in reaching a more global decision on request acceptance, especially under high load. Section 7.7 shows the effectiveness of this technique, particularly in the presence of replica failures. Additionally, this mechanism achieves fairness amongst clients as each client is regularly part of the prioritized group and has a high chance of getting their requests accepted. In our evaluations, this results in all clients having a similar share of accepted and rejected requests over the runtime of an experiment.

Further Options. Depending on the application and/or environment, other decision criteria could include for example different request priority categories or an analysis of the request depending on the estimated resource costs.

5.2 Forwarding Mechanism

Although conducting the acceptance test for requests independently of each other, as a group replicas cooperate to provide the following guarantee (see Section 6.2 for proof):

Property 5.1 (Liveness). *If a client request is accepted by at least one correct replica, the request will be eventually agreed on and executed by all correct replicas in the system.*

In order to guarantee that the overall replicated system meets this requirement, after having accepted a client request a replica's forwarding mechanism is responsible for keeping the request available. An Idem replica achieves this by relaying the request to other replicas, thereby ensuring that all correct replicas eventually are in possession of the request, including those that had not directly received the request from the client (e.g., due to network issues) or that previously rejected and discarded the request. However, as naively forwarding all accepted requests would put an unnecessary strain on replicas, Idem's forwarding mechanism applies the following optimizations to minimize overhead:

Delayed Forwarding. In practice, forwarding does not necessarily have to be performed immediately after the acceptance of a request. Instead, it can be delayed until the expiration of a timeout and only performed in case the corresponding request is not fully processed in the meantime. This time-based relaying of requests is limited to the rare scenarios in which a request is only available on f replicas.

Caching of Recently Rejected Requests. Replicas in Idem do not simply discard a request once they rejected it. As there is a chance that this request is accepted by other replicas, it might still be agreed on, which is why replicas in Idem store a limited number of recently rejected requests in a cache, thereby reducing the number of required forwarded requests.

Request Fetching. Idem allows replicas to explicitly ask for the forwarding of a particular request by sending a ⟨Fetch, *id*⟩ message to another replica. Specifically, a replica uses this option when the agreement process commits the id of a request that the replica at this point does not own (e.g., due to not having received the request or already having removed it from its recently rejected cache). Request fetching enables lagging replicas to quickly catch up without having to wait for the forward timeout, and it also removes the need to proactively forward requests that have already been executed but are not included in a checkpoint.

5.3 Client-Side Semantics

Combined with Property 5.1, the fact that replicas individually decide on the acceptance/rejection of a request may result in clients being able to observe different scenarios, which as further discussed below are not all mutually exclusive.

- **Success:** Once a client receives a **Reply** to its request, the client has proof that the request was accepted by at least one correct replica, completed the agreement process, and consequently either already has been or eventually will be executed by all correct replicas. As a result, the client is allowed to use the reply and consider the operation finished.
- **Ambivalence:** As soon as a client has obtained $n - f$ **Rejects** from different replicas, there is some indication that its request might not be processed. Whether or not this is the case depends on the reaction of the remaining f replicas from which the client has not heard of yet. However, with up to f replicas possibly having crashed, at this point there is no guarantee that the client will receive additional responses in the future. Hence, the client needs to make a decision based on the partial information it possesses at this point.
- **Failure:** If a client is able to collect n **Rejects**, it has confirmation that its request was rejected by all replicas in the system and hence will not be executed. Based on this knowledge, the client immediately aborts the operation and resorts to its fallback mechanism.

Our approach ensures that after having submitted its request to the replicated service, a client will eventually end up in one of these three scenarios (see Section 6.3). While the success and failure states are final and entail direct and definitive actions regarding the completion/abortion of the operation, the ambivalence state offers some flexibility on how the client decides to react:

- **Pessimistic client implementations** minimize rejection latency by immediately aborting an operation if the first $n - f$ responses from replicas are all Rejects.
- **Optimistic client implementations** delay their reaction for a certain (configurable) amount of time in the hope to obtain further responses (i.e., a late reply or f Rejects) in order to subsequently switch to the success or failure scenario. If this strategy is not successful, the client aborts the operation after a timeout. Compared with the pessimistic variant, this approach trades off a potentially higher rejection latency for an improved success rate of operations.

The outlined pessimistic and optimistic approaches represent the two ends of an entire spectrum of possible client-implementation variants. Additional and more complex decision strategies are conceivable. One optimization, for example, is to extend the optimistic approach with a warning to the user-side application upon receiving the $n - f$th Reject. This provides a client with the opportunity of being able to start fallback preparations early, while (in the background) still waiting for a late reply to arrive.

Independent of the particular strategy, if a client aborts an operation while being in the state of ambivalence, there can be cases in which the operation nevertheless is eventually executed by Idem, for example if the client's request has been accepted by a single replica that temporarily got disconnected from the rest of the group. Such scenarios do not pose an additional problem since they can occur in replicated systems anyway. Specifically, if a temporary network partition occurs between a client and Idem while the client submits a request, there may be uncertainty at the client with regard to whether or not its request reached the system and was processed there. A common way used by traditional clients to handle such situations is to issue a subsequent probe request once the network partition is resolved. For example, if the client of a key-value store is uncertain about the execution of a write, a subsequent read to the same key brings clarity. If demanded by the application, the same techniques can be used by clients to remove uncertainty in the context of Idem.

In both the ambivalence and failure state, the client can resort to its fallback mechanism and make decisions independent of the replicated system. If an application requires state to be shared between the client and the server, this can cause these two states to diverge, especially if the system is under an overload situation and rejecting requests for a prolonged amount of time. To avoid such divergence affecting the service quality, applications in Idem can mitigate this in numerous ways, for example by using aging mechanisms and giving stale data less weight, or by monitoring a client's state via external means.

6 Correctness

With our approach to overload prevention not affecting the internals of the consensus algorithm (see Section 4.2), in this section we focus on aspects that are specific to Idem. In particular, this includes correctness arguments for Idem's implicit garbage collection, the liveness property ensured by its forwarding mechanism, and Idem's client-side semantics.

6.1 Implicit Garbage Collection

Theorem 6.1. *Assuming $r_{max} <= w$ and a window W starting at sqn, a replica can discard the oldest request in W once it receives a consensus instance with sequence number $sqn + r_{max}$.*

Proof. To propose a new request $\langle R_{new}, sqn + r_{max} \rangle$, the leader first waits for $f+1$ Requires. That means that at least $f+1$ replicas have accepted R_{new} and at that point had less than r_{max} active requests. As a replica only removes a request from this set once it has been executed, at least $f+1$ replicas must have executed a request $\langle R_{old}, sqn \rangle$. A replica can now safely discard all requests up to this sequence number and move the window start forward to sqn+1. □

6.2 Server-Side Liveness

Theorem 6.2 (cf. Property 5.1). *If a client request is accepted by at least one correct replica, the request will be eventually agreed on and executed by all correct replicas in the system.*

Proof. If a replica accepts a request, the replica hands the request over to the agreement protocol, and furthermore adds the request to a local set of accepted requests whose contents the replica periodically multicasts to all replicas in the system. This implements a reliable-broadcast mechanism [16] which for the fair-loss point-to-point connections in our system model (see Section 2.1) guarantees that if the sending replica is correct, the request will eventually be received by all correct replicas. As replicas automatically (i.e., without conducting an acceptance test of their own) accept requests that are relayed by other replicas, all correct replicas eventually instruct the agreement protocol to order the request. With all correct replicas endorsing the request, independent of the specifics of the consensus protocol, the request is eventually agreed on and executed by all correct replicas in the system.

The reliable-broadcast mechanism only takes effect for requests contained in a replica's set of accepted requests. Due to a replica solely removing requests from this set that are already included in a stable checkpoint in the replica's possession, garbage collection does not interfere with the requests' agreement and execution. □

6.3 Client-Side Liveness

Theorem 6.3. *If a correct client issues a request to the replicated service, the client eventually will reach one of the three states: success, ambivalence, or failure.*

Proof. In accordance with our system model, the request distributed by a correct client is eventually received by all correct replicas in the system, potentially after one or more retransmissions (see Section 2.1); for the same reason, responses sent by correct replicas eventually arrive at a correct client. Correct replicas perform the acceptance test for each new request they receive, which leaves two cases: (1) If at least one of the correct replicas accepts the request, Theorem 6.2 ensures that the request will be executed by all replicas and therefore the client will eventually be provided with a reply, hence reaching the success state. (2) If n_c replicas in the group are correct and none of them accepts the request, the client will eventually receive at least n_c Rejects from these replicas and consequently either enter the ambivalence state ($n_c \geq n - f$) or the failure state ($n_c = n$). These two cases are exhaustive. □

Theorem 6.4 (Liveness). *If the probability of passing the acceptance test is lower-bound by $p > 0$ for every request (even if the system is saturated) and a correct client issues infinitely many requests, it will reach the success state infinitely many times with probability 1.*

Proof. For a newly issued request to be processed in a timely manner and lead to the success state, $f+1$ replicas need to accept this request. According to the assumption that a replica accepts a request with probability $p > 0$, the probability for a client's request being successful is hence $p^{f+1} > 0$. Using the second Borel-Cantelli lemma [24] and with individual requests of a client being independent, this ensures that, given infinitely many requests, the probability that infinitely many of them will reach the success state is 1. □

In a typical real-world system environment, the assumption of a non-zero probability for a request passing the acceptance test is justified. A live system regularly processes requests and can accept new ones, even while saturated. Due to variation (e.g., in scheduling or processing time of requests), this happens in non-deterministic intervals. Additionally, the network latency usually behaves non-deterministic as well. A newly issued request hence reaches the system at an essentially random point in time regarding the load and capacity of the system. As long as the acceptance test does not directly discriminate against specific clients or requests, this results in each request having an above-zero probability of reaching the system at the right moment and being accepted in the next slot. Note that this probability substantially varies based on the current load situation: Without overload, the probability is mainly 1, but it can drop significantly for severe overload situations.

For acceptance tests that use a prioritization scheme (such as the implemented version of active queue management) and hence do occasionally discriminate against certain clients, the argument still applies as long as the prioritization scheme itself is fair and regularly assigns each client to the prioritized group.

Also, if a client has a request it requires to be processed by the replicated service, Theorem 6.4 ensures that a single request is eventually processed when being continuously resubmitted.

7 Evaluation

In this section, we evaluate collaborative overload prevention in Idem, investigating the impact of different configurations and load levels, and comparing our solution to the state of the art. For this purpose, we distinguish between four main systems: (1) **Idem** refers to the Java implementation[1] of our protocol as presented in Sections 4 and 5. (2) **Idem**$_{noPR}$ denotes Idem with a disabled rejection mechanism, meaning that the system does not perform proactive rejection and therefore is not protected against overload. As a key benefit, this setting allows us to investigate the impact of the collaborative overload prevention itself. (3) **Paxos** represents an implementation of Kirsch and Amir's Paxos variant [43], which relies on a steady leader and hence is directly comparable to Idem. Sharing the same code base with Idem, this Paxos implementation enables us to precisely analyze the effects of the replication protocol on performance. (4) **BFT-SMaRt** is a widely used replication library [11], which we configure to run in its crash-fault tolerant setting. Based

[1]The prototype is publicly available under https://doi.org/10.5281/zenodo.13847918

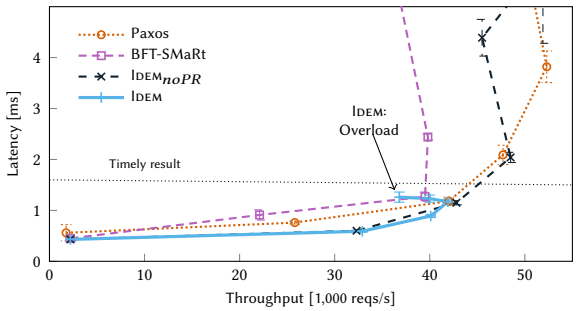


Figure 6: Performance comparison under increasing load.

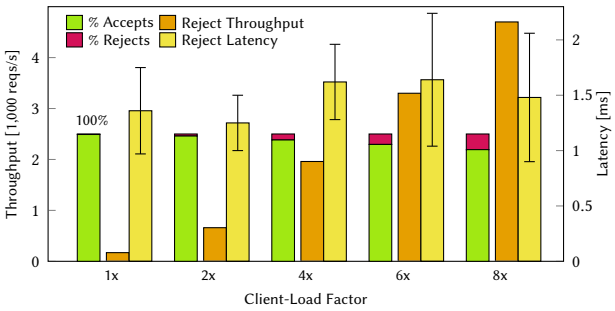


Figure 7: Reject behavior in IDEM under increasing load.

on BFT-SMaRt, we are able to study how a production-grade implementation reacts to overload. We configure all four systems (e.g., batch size, checkpoint intervals) beneficial to the corresponding protocol to allow for a fair comparison.

7.1 Experimental Environment

We conduct our experiments using the YCSB [25] benchmark with an update-heavy workload. The application is a key-value store and thus represents a typical example of a time-sensitive system that is either directly or indirectly used as replicated service in our target use cases such as the ones described in Section 2.3. Replicas run on a cluster of three servers (Intel Xeon CPU E3-1275 v5 @ 3,60 GHz, 16GB RAM), while clients are hosted on a separate server and are configured to submit requests in a closed loop.

For IDEM, clients apply the optimistic approach described in Section 5.3 with a timeout of 5ms, meaning that after having received $n-f$ rejections, a client waits up to 5ms in the hope of a reply before abandoning its request. Following the established technique to manage load in distributed client-server systems [15, 71], after having aborted an operation due to rejection, and thereby having learned that the replicated service is currently under high load, a client issues the next operation after a random delay (50–100ms). For deciding on rejects, IDEM replicas rely on a default reject threshold of RT=50 and perform active queue management (see Section 5.1) with time slices of 2s. Further, replicas in IDEM forward a request if it is not executed within 10ms after sending the initial REQUIRE. The evaluation results each represent an average of three runs and error bars show the standard deviation in latency across the duration of a run. Individual points in the graphs represent the load pressure on the system, caused by an increase in closed-loop clients.

7.2 Performance under Increasing Request Load

Our first experiment compares the performance of IDEM, Paxos and BFT-SMaRt under an increasing request load. As shown in Figure 6, Paxos and BFT-SMaRt perform poorly under overload. As soon as these systems reach their respective maximum throughput, the latency drastically escalates when the load is further increased. In contrast, for IDEM collaborative overload prevention effectively limits the latency of the system. Up until the point the rejection mechanism takes action, the system behaves the same as traditional protocols. But as soon as there are more requests in the system than the reject threshold allows (at around 43k requests/s), the latency no longer increases but plateaus at around 1.3ms.

We strained all systems with up to four times their maximum throughput, resulting in systems without active rejection to reach more than 600% of their normal latency; additional load would extend the degradation even further. A comparison between IDEM and IDEM$_{noPR}$ shows that enabling the rejection mechanism only has a negligible impact on the unsaturated system's performance; the two curves only diverge after the rejection threshold is reached and the rejection mechanism becomes effective. More importantly, it confirms that our collaborative overload-prevention mechanism is what effectively minimizes overload-induced tail latency in IDEM.

7.3 Reject Behavior

The reason that IDEM is able to avoid overload-induced tail latency is the usage of rejection to actively prevent an overload state. Since the rejection notifications are also an important means for the client to assess the current state of the system, we now investigate their behavior in detail.

The results shown in Figure 7 represent the throughput and latency for rejects in IDEM for different degrees of overload. As a metric for overload, we use the number of active clients that are concurrently interacting with the replicated service. Note that in an overload situation this does not necessarily scale linearly to the number of requests, since IDEM clients actively try to regulate the pressure (i.e., request load) on the system (see Section 7.1). For our experiments, we determine 50 clients as the baseline (client-load factor 1x), since this is the point where the system reaches saturation and latency starts increasing disproportionally compared to the increase in throughput. With our clients running in a closed loop, they issue requests at high frequency, achieving a throughput of 43k requests/s. Consequently, 50 benchmark clients are representative of thousands or tens of thousands of real-world clients.

The results show that the rejects also have a stable latency of around 1.3–1.5ms, even with an overload of up to 8 times the baseline client load. This is in the same range as a timely result and hence allows a client to quickly switch to its fallback computation once it learns that the system is busy.

The high standard deviation of reject latency is a side effect of the fact that clients pursue the optimistic approach of additionally waiting up to 5ms after receiving $n-f$ rejects in an effort to obtain a late reply. As the latency represents the time between the client sending its request and either getting a result or abandoning the operation, this timeout is included in the latency if only some replicas reject a request while others accept it. The comparably low average latency shows that this does not happen often.

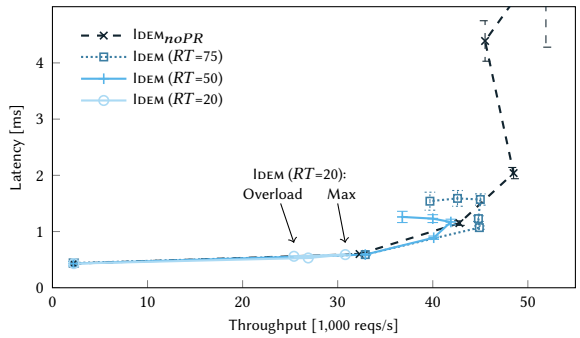
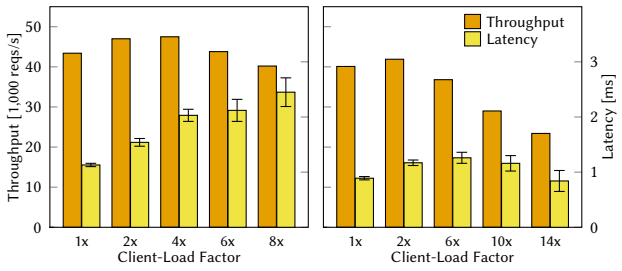


Figure 8: Variation of reject threshold in IDEM.



(a) Misconfigured threshold **(b) Extreme load**

Figure 9: IDEM's behavior under disruptive conditions.

Also, even for a severe overload, the amount of rejects compared to the total throughput is very low (only around 10% for a client-load factor of 8) and in moderate overload scenarios even less than 3%. This is caused by the clients' behavior of delaying their subsequent requests when they receive a reject and know the system is overloaded. The higher the client-load factor is, the more clients behave this way, thereby regulating the overall pressure on the system and leading to a slower increase of rejects.

7.4 Rejection Overhead

In our next experiment, we evaluate the overhead of IDEM's rejection mechanism by comparing IDEM with IDEM$_{noPR}$ in terms of their network usage. For this purpose, we issue a fixed number of 1,000,000 requests to both systems under different load scenarios (client-load factor 0.5x, 1x and 4x) and monitor the total network traffic both of clients and between replicas. A request is only counted as completed upon a successful response. Rejected and therefore aborted requests in IDEM do not count towards the total request number and need to be resubmitted.

The results in Table 1 show clearly that IDEM's rejection mechanism has a negligible impact on the total network traffic of the system. With a variation between individual measurements of the same setup of around 2-3%, there is no visible difference between IDEM and IDEM$_{noPR}$. This highlights the effectiveness of our measures to minimize the rejection mechanism's overhead (especially by reducing the number of forwarded requests and the caching of recently rejected requests; see Section 5.2) and emphasizes the low number of additional rejects even in high overload scenarios.

7.5 Variation of Reject Threshold

As shown in Section 7.2, the reject mechanism has a significant impact on IDEM's behavior. The defining factor here is the reject threshold RT, so how many concurrent active requests each replica allows before further ones are rejected. This directly impacts the possible maximum throughput of the system. Hence, in our next experiment, we investigate the impact of adjusting the reject threshold on the system's performance. Figure 8 shows how the throughput and latency change depending on different reject thresholds.

	Medium Load	High Load	Overload
IDEM$_{noPR}$	3.26 GB	3.15 GB	3.19 GB
IDEM	3.24 GB	3.08 GB	3.19 GB

Table 1: Overhead of IDEM's rejection mechanism.

The two reject threshold settings of RT=50 and RT=75 both provide a good balance between low latency and achieved throughput. RT=50 is more conservative and set just below what the system could potentially handle. This results in a slightly lower throughput of 43k requests/s, but achieves a stable latency of less than 1.3ms. RT=75, on the other hand, sets the threshold somewhat above the overload edge of the system. This allows it to offer a higher throughput of approximately 46k requests/s at a slightly higher latency (up to 1.6ms), which nevertheless shows the desired plateau effect.

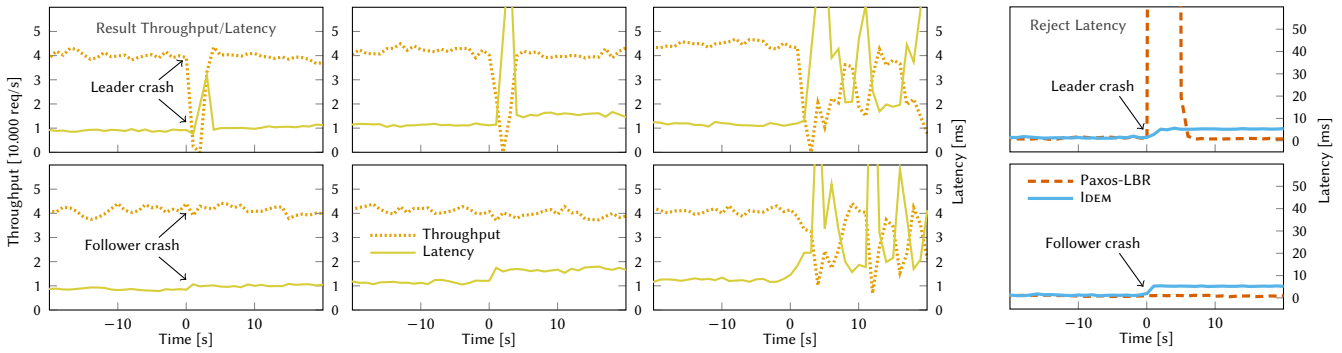
Additionally, we repeat the experiment with an artificially low reject threshold of 20, which is considerably below what the system can handle. Such a configuration affects the achievable throughput, which is restricted to only 32k requests/s, around 65% of the possible maximum throughput. However, it provides a very low and stable latency: even for severe overload situations, the latency is constantly close to the minimum latency of the system and never above 0.6ms.

In summary, the reject threshold directly impacts the effectiveness of the proactive rejection mechanism. Note that even though the configurations exhibit different behaviors in an overload situation or when their respective reject threshold is reached, they all have nearly identical performance below this threshold. Hence, when the underlying base performance of the system is known (e.g., by benchmarking the system with a disabled rejection mechanism), it is possible to configure the reject threshold to target a desired latency and throughput on that graph.

7.6 IDEM under Disruptive Conditions

As the previous sections have shown, a well-configured IDEM can offer a stable low latency even when the system suffers from an overload situation. In this section, we investigate how an inappropriate usage of IDEM does affect this behavior. First, we misconfigure IDEM with an unreasonably high reject threshold, and afterwards we evaluate the impact of extreme overloads up to 14 times as high as our baseline client load.

Misconfiguration. In the first part of this experiment, we set the reject threshold to 100, which is way above what the system can handle and means that the system reaches an overload state before the active rejection mechanism can prevent it. The results shown in Figure 9a represent the average throughput and latency for different stages of overload, again represented by the client-load



(a) Normal load **(b) Overload** **(c) Overload without AQM** **(d) Comparison with LBR**

Figure 10: Impact of a replica crash on Idem. In all measurements, a respective replica was crashed after 250s of service.

factor as defined in Section 7.3. As in previous experiments with Idem, the latency rises in an overload scenario and this upward trend is only slowed at around 2ms, which is when the rejection mechanism is activated. However, the mechanism is still able to considerably slow the increase in latency, which is nearly constant between 4x and 6x of the client-load factor. Only after this already severe overload, the latency again starts to slowly increase. This is likely caused by the fact that, with the system already being in an overload state just by the immense number of requests, replicas are no longer able to withstand the increasing number of rejects without it having an impact on performance. Our experiment shows how important it is to consider the maximum throughput of the system when configuring the reject threshold. However, Idem is nevertheless able to prevent the immediate explosion in latency shown by state-of-the-art protocols at such overload.

Extreme Load. The second part of this experiment puts Idem under extreme overload, with up to 14 times our defined client-load baseline. The results are shown in Figure 9b. For a medium overload of up to a client-load factor of 6, the throughput stays relatively stable and latency slightly increases to 1.3ms, which is only around 0.4ms higher than the latency just before the rejection mechanism takes action. After that, the system behavior shifts and both the throughput and latency decrease with more load. At 14 times the client load, the system has a reduced throughput of 24k requests/s (55% of its highest throughput), but offers an average latency of only 0.9ms. We reason this decrease in throughput is caused due to the vast majority of clients facing mostly rejects and thus delaying their subsequent requests to relieve the system. Overall, the experiment shows that Idem can easily tolerate even extreme overload bursts and still offer a stable latency.

7.7 Impact of Replica Failure

The next experiments investigate the impact of replica failures on Idem, both under normal load and when it is in an overload situation. In this context, we inspect replica crashes for an additional system: Idem$_{noAQM}$ is a variant of Idem without active queue management (see Section 5.1), meaning without measures to provide a more unanimous decision on rejects. During the experiments, we run each system under a constant load and deliberately crash a leader or follower after a certain amount of time. We conduct our experiments with two load configurations: one with 50 clients, which represents a normal load and is just before the reject mechanism takes action, and one with 100 clients, which is already past the overload threshold of the systems.

Leader Crash. Let us first inspect the impact of a leader crash on Idem and Idem$_{noAQM}$ as shown in the top row of Figures 10a to 10c. After the leader has crashed, Idem needs to complete a view change before it can process requests again. This takes around 1.5 seconds, which mostly consists of the timeout before a view change is triggered. In the overload scenario, Idem shows a slight decrease in throughput of around 9% and an increase in latency of around 45% after the view change. Note that the latency is still stable below 1.7ms. This performance decrease is caused by the fact that the leader in Idem waits for $f + 1$ Requires before proposing a request. As there are double the amount of concurrent requests in the system than the request threshold allows, the $f+1$ remaining replicas have a high chance of accepting a different subset of requests that will then only be agreed after the forward timeout of 10ms. When comparing to Idem$_{noAQM}$, which shows a high instability in both throughput and latency with only $f + 1$ replicas, it is obvious that the active queue management succeeds to effectively mitigate this problem.

Follower Crash. When crashing a follower replica (as shown in the bottom row of Figures 10a to 10c), there is no need for a view change and no complete interruption in service for either system. After the follower crash, both Idem and Idem$_{noAQM}$ display a similar behavior as discussed above, as there are again only $f + 1$ remaining replicas in the system.

7.8 Comparison with Leader-based Rejection

In our final experiment we compare Idem with Paxos$_{LBR}$, the Paxos variant from Section 3.3 that uses leader-based rejection. Both approaches successfully prevent an overload situation and sustain a low reply latency, which is why we focus on the reject latency for this experiment. Figure 10d shows the reject latency of both systems when a leader or follower fails during an overload situation.

In case of a follower crash, Paxos$_{LBR}$ is not affected, resulting in no change in the reject latency. Idem shows a slight increase in reject latency after the crash, but also offers a continuous rejection. This latency increase is caused by the optimistic client implementation,

which leads clients to wait a short time (~5ms) after $n - f$ rejects for either a result or, more likely in a high-overload scenario, n rejects. As the latter cannot be fulfilled with the missing replica, it is more likely for a client to wait the full timeout before abandoning the request, leading to an increase in measured reject latency.

A more distinct difference between Paxos_{LBR} and Idem can be seen when the system experiences the crash of the current leader. As the rejection in Paxos_{LBR} depends solely on the leader, it cannot offer any rejects after the crash until the view change is completed and the client has identified the new leader. As clients in Paxos initially only send a request to the presumed leader, this takes multiple client-side timeouts as well as the time for the view change to be effective, resulting in a reject downtime of around 4 seconds. Idem, on the other hand, is able to continuously provide the client with rejects even during a leader crash and view change, demonstrating the advantage of collaborative overload prevention.

8 Related Work

The topic of correct provisioning and overload prevention under changing operative conditions is widely researched [1, 35] and has recently gained even more relevance with the categorization of metastable failures [14, 39], a failure state of permanent overload with very low goodput. Auto-scaling [37, 51] allows to dynamically adjust the number of resources to match the current load, but is often designed for elastic applications in the cloud and not easily applicable to state-machine replication protocols as it would require configuration changes. Load-balancing [17, 23, 42, 61, 72] works by trying to fairly distribute load among multiple nodes, decreasing the risk of tail latency caused by the contention on individual nodes. Another option is to add admission control [9, 10, 22, 32, 71] and simply block or delay requests when the system reaches an overload situation. In contrast to proactive rejection, this does not provide the client with any means to react other than waiting for a timeout.

In the context of leader-based crash-tolerant state-machine replication exist numerous works on improving the effectiveness and performance of a system [19, 27, 30, 34, 38, 45, 48, 57]. Copilot [55] aims to sustain normal-case latency despite the slowdown of one replica by adding a "copilot" to the leader, thereby offering redundancy to the leader role. S-Paxos [12] separates the replication of commands from the agreement process to enable an id-based ordering. PigPaxos [20] alleviates a bottleneck in the leader by introducing "relay nodes" taking over part of the leader's communication. While such measures reduce the overload or slowdown potential at the leader, they cannot avoid overload-induced tail latency when the system as a whole is under pressure. SwiftPaxos [60] targets contention-based latency in a geo-replicated setting by allowing replicas to dynamically switch between a fast and slow path based on a request's dependencies. This allows the system to achieve an optimal latency both during low and high system load. However, it is not designed to prevent or mitigate overload. To our knowledge, there is no work targeting the prevention of overload-induced tail latency in state-machine replication protocols at the system level and to the extent we present with Idem in this paper.

For replicated systems, rotating-leader or multi-leader protocols [4, 5, 7, 33, 53, 54, 70, 73, 74] aim at increasing the scalability and reliability of state-machine replication protocols by allowing all replicas to act as command leaders. While removing the bottleneck of a single stable leader, this cannot protect a system when the overload affects all replicas equally, as is the case for our target applications. However, we expect that the concept of collaborative overload-prevention can be integrated into such multi-leader protocols with little adjustments.

Abortable consensus [8] is a generic technique to dynamically switch the protocol used by a group of replicas to ensure consistency. While also potentially declining the execution of requests, in contrast to our approach, it is not suitable to prevent overload due to the abortion of operations being limited to (rare) reconfiguration events.

Similar to crash-tolerant protocols, Byzantine fault-tolerant replication protocols [6, 11, 18] also suffer from overload induced tail latency. Protocols such as Omada [31], Spinning [68] or Mir-BFT [66] already provide measures to reduce the load at the leader, however, their main goal is to restrict the capability of faulty actors (i.e., clients or the leader replica) to impede the system's performance, and they cannot prevent overload at a system level induced by legitimate requests. While Idem is currently designed for a crash fault-tolerant setting, we believe it is possible to adapt the approach to the Byzantine fault model. In this context, the concept of micro replication [28] may help to provide collaborative overload prevention in a modular and consensus-algorithm-independent way.

As mentioned in Section 3.2, overload is not the only source of tail latency in distributed systems [26, 49]. Other factors such as network contention, unfavorable scheduling or garbage collection can also increase the response time of individual requests, even independent of the current load. Existing approaches tackle these problems via considerate scheduling [58, 59], special hardware [41] or programming and communication models [46], or adapting the language runtime [52]. As a protocol-level approach, proactive rejection is independent of the actual implementation and thus can be combined with many of these existing solutions.

9 Conclusion

Idem uses collaborative overload prevention to effectively reduce overload-induced tail latency via proactive rejection. We showed that proactive rejection enables Idem to sustain adequate throughput and stable latency in severe overload situations. Furthermore, the collaborative approach allows Idem to provide its clients with continuous stable reject latency even during replica crashes.

Acknowledgments

We thank the anonymous reviewers as well as our shepherd, Matej Pavlovic, for their invaluable feedback. This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 446811880, 541017677.

References

[1] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. 2002. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems* 13, 1 (2002), 80–96.

[2] Richelle Adams. 2013. Active Queue Management: A Survey. *IEEE Communications Surveys & Tutorials* 15, 3 (2013), 1425–1476.

[3] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. 1696–1710.

[4] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.

[5] Yair Amir, Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, and Paul Ciarfella. 1995. The Totem Single-Ring Ordering and Membership Protocol. *ACM Transactions on Computer Systems* 13, 4 (1995), 311–342.

[6] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2024. The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation. In *Proceedings of the 21st Symposium on Networked Systems Design and Implementation (NSDI '24)*. 371–400.

[7] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN '17)*. 49–60.

[8] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems* 32, 4, Article 12 (2015).

[9] Novella Bartolini, Giancarlo Bongiovanni, and Simone Silvestri. 2007. Distributed Server Selection and Admission Control in Replicated Web Systems. In *Proceedings of the 6th International Symposium on Parallel and Distributed Computing (ISPDC '07)*. Article 31.

[10] Novella Bartolini, Giancarlo Bongiovanni, and Simone Silvestri. 2009. Self-* Through Self-Learning: Overload Control for Distributed Web Systems. *Computer Networks* 53, 5 (2009), 727–743.

[11] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*. 355–362.

[12] Martin Biely, Zarko Milosevic, Nuno Santos, and André Schiper. 2012. S-Paxos: Offloading the Leader for High Throughput State Machine Replication. In *Proceedings of the 31st Symposium on Reliable Distributed Systems (SRDS '12)*. 111–120.

[13] Christof Brandauer, Gianluca Iannaccone, Christophe Diot, Thomas Ziegler, Serge Fdida, and Martin May. 2001. Comparison of Tail Drop and Active Queue Management Performance for Bulk-Data and Web-like Internet Traffic. In *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC '01)*. 122–129.

[14] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable Failures in Distributed Systems. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS '21)*. 221–227.

[15] Marc Brooker. 2019. Timeouts, Retries, and Backoff with Jitter. *The Amazon Builders' Library* (2019).

[16] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming (2nd Edition)*. Springer Publishing Company, Inc.

[17] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. 1999. Dynamic Load Balancing on Web-Server Systems. *IEEE Internet Computing* 3, 3 (1999), 28–39.

[18] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. 173–186.

[19] Tarcisio Ceolin, Fernando Dotti, and Fernando Pedone. 2020. Parallel State Machine Replication from Generalized Consensus. In *Proceedings of the 39th Symposium on Reliable Distributed Systems (SRDS '20)*. 133–142.

[20] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2021. PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 235–247.

[21] Liuhua Chen and Haiying Shen. 2016. Towards Resource-Efficient Cloud Systems: Avoiding Over-Provisioning in Demand-Prediction Based Resource Provisioning. In *Proceedings of the 2016 International Conference on Big Data (BigData '16)*. 184–193.

[22] Xuan Chen and John Heidemann. 2005. Flash Crowd Mitigation via Adaptive Admission Control Based on Application-Level Observations. *ACM Transactions on Internet Technology* 5 (2005), 532–569.

[23] Timothy C.K. Chou and Jacob A. Abraham. 1982. Load Balancing in Distributed Systems. *IEEE Transactions on Software Engineering* SE-8, 4 (1982), 401–412.

[24] Kai Lai Chung and Paul Erdös. 1952. On the Application of the Borel-Cantelli Lemma. *Trans. Amer. Math. Soc.* 72, 1 (1952), 179–186.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st Symposium on Cloud Computing (SoCC '10)*. 143–154.

[26] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.

[27] Christian Deyerl and Tobias Distler. 2019. In Search of a Scalable Raft-based Replication Architecture. In *Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '19)*. 1–7.

[28] Tobias Distler, Michael Eischer, and Laura Lawniczak. 2023. Micro Replication. In *Proceedings of the 53rd International Conference on Dependable Systems and Networks (DSN '23)*. 123–137.

[29] Tobias Distler and Rüdiger Kapitza. 2011. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys '11)*. 91–105.

[30] Dan Dobre, Matthias Majuntke, Marco Serafini, and Neeraj Suri. 2010. HP: Hybrid Paxos for WANs. In *Proceedings of the 8th European Dependable Computing Conference (EDCC '10)*. 117–126.

[31] Michael Eischer and Tobias Distler. 2017. Scalable Byzantine Fault Tolerance on Heterogeneous Servers. In *Proceedings of the 13th European Dependable Computing Conference (EDCC '17)*. 34–41.

[32] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. 2004. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the 13th International Conference on World Wide Web (WWW '04)*. 276–286.

[33] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient Replication via Timestamp Stability. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*. 178–193.

[34] Ian Aragon Escobar, Eduardo Alchieri, Fernando Luís Dotti, and Fernando Pedone. 2019. Boosting Concurrency in Parallel State Machine Replication. In *Proceedings of the 20th International Middleware Conference (Middleware '19)*. 228–240.

[35] Stephane Genaud and Julien Gossa. 2011. Cost-Wait Trade-Offs in Client-Side Resource Provisioning with Elastic Clouds. In *Proceedings of the 4th International Conference on Cloud Computing (CLOUD '11)*. 1–8.

[36] Antonio G. C. Gonzalez, Marcos V. S. Alves, Gustavo S. Viana, Lilian K. Carvalho, and João C. Basilio. 2018. Supervisory Control-Based Navigation Architecture: A New Framework for Autonomous Robots in Industry 4.0 Environments. *IEEE Transactions on Industrial Informatics* 14, 4 (2018), 1732–1743.

[37] Jordi Guitart, Jordi Torres, and Eduard Ayguadé. 2010. A Survey on Performance Management for Internet Applications. *Concurrency and Computation: Practice and Experience* 22, 1 (2010), 68–106.

[38] Heidi Howard, Aleksey Charapko, and Richard Mortier. 2021. Fast Flexible Paxos: Relaxing Quorum Intersection for Fast Paxos. In *Proceedings of the 22nd International Conference on Distributed Computing and Networking (ICDCN '21)*. 186–190.

[39] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI '22)*. 73–90.

[40] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC '10)*. 145–158.

[41] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. 2017. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. *ACM Transactions on Embedded Computing Systems* 16, 5s (2017), 1–20.

[42] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the 3rd Symposium on Cloud Computing (SoCC '12)*. 1–14.

[43] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders: An Overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS '08)*. 14–18.

[44] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[45] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (2006), 79–103.

[46] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling Low Tail Latency on Multicore Key-Value Stores. 13, 7 (2020), 1091–1104.

[47] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. 2016. SAREK: Optimistic Parallel Ordering in Byzantine Fault Tolerance. In *Proceedings of the 12th European Dependable Computing Conference (EDCC '16)*. 77–88.

[48] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*. 467–483.

[49] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 5th Symposium on Cloud Computing (SoCC '14)*. 1–14.

[50] You Li and Javier Ibanez-Guzman. 2020. Lidar for Autonomous Driving: The Principles, Challenges, and Trends for Automotive Lidar and Perception Systems. *IEEE Signal Processing Magazine* 37, 4 (2020), 50–61.

[51] Tania Lorido-Botrán, Jose Miguel-Alonso, and Jose A. Lozano. 2014. A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12 (2014), 1572–9184.

[52] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS '15)*.

[53] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*. 369–384.

[54] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. 358–372.

[55] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. 2020. Tolerating Slowdowns in Replicated State Machines using Copilots. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI '20)*. 583–598.

[56] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*. 305–320.

[57] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*. 43–57.

[58] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 325–341.

[59] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. 2017. Rein: Taming Tail Latency in Key-Value Stores via Multiget Scheduling. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*. 95–110.

[60] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. 2024. SwiftPaxos: Fast Geo-Replicated State Machines. In *Proceedings of the 21st Symposium on Networked Systems Design and Implementation (NSDI '24)*. 345–369.

[61] Aya A. Salah Farrag, Safia Abbas Mahmoud, and El Sayed M. El-Horbaty. 2015. Intelligent Cloud Algorithms for Load Balancing Problems: A Survey. In *Proceedings of the 7th International Conference on Intelligent Computing and Information Systems (ICICIS '15)*. 210–216.

[62] Rainer Schiekofer, Johannes Behl, and Tobias Distler. 2017. Agora: A Dependable High-Performance Coordination Service for Multi-Cores. In *Proceedings of the 47th International Conference on Dependable Systems and Networks (DSN '17)*. 333–344.

[63] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[64] Haiying Shen and Liuhua Chen. 2018. Resource Demand Misalignment: An Important Factor to Consider for Reducing Resource Over-Provisioning in Cloud Datacenters. *IEEE/ACM Transactions on Networking* 26, 3 (2018), 1207–1221.

[65] Vason P. Srini. 2006. A Vision for Supporting Autonomous Navigation in Urban Environments. *Computer* 39, 12 (2006), 68–77.

[66] Chrysoula Stathakopoulou, Tudor David, Matej Pavlovic, and Marko Vukolić. 2022. Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *Journal of Systems Research* 2, 1 (2022).

[67] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP '13)*. 309–324.

[68] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 28th Symposium on Reliable Distributed Systems (SRDS '09)*. IEEE, 135–144.

[69] Alf Inge Wang, Eivind Sorteberg, Martin Jarrett, and Anne Marte Hjemas. 2008. Issues Related to Mobile Multiplayer Real-time Games over Wireless Networks. In *Proceedings of the 2008 International Symposium on Collaborative Technologies and Systems (CTS '08)*. 237–246.

[70] Weilue Wang, Yujuan Tan, Changze Wu, Duo Liu, Yu Wu, Longpan Luo, and Xianzhang Chen. 2022. Towards Highly-Concurrent Leaderless State Machine Replication for Distributed Systems. *Journal of Systems Architecture* 127, Article 102516 (2022).

[71] Matt Welsh and David Culler. 2003. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*.

[72] Bartek Wydrowski, Robert Kleinberg, Stephen M Rumble, and Aaron Archer. 2024. Load Is Not What You Should Balance: Introducing Prequal. In *Proceedings of the 21st Symposium on Networked Systems Design and Implementation (NSDI '24)*. 1285–1299.

[73] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 38th Symposium on Principles of Distributed Computing (PODC '19)*. 347–356.

[74] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SD-Paxos: Building Efficient Semi-Decentralized Geo-replicated State Machines. In *Proceedings of the 9th Symposium on Cloud Computing (SoCC '18)*. 68–81.