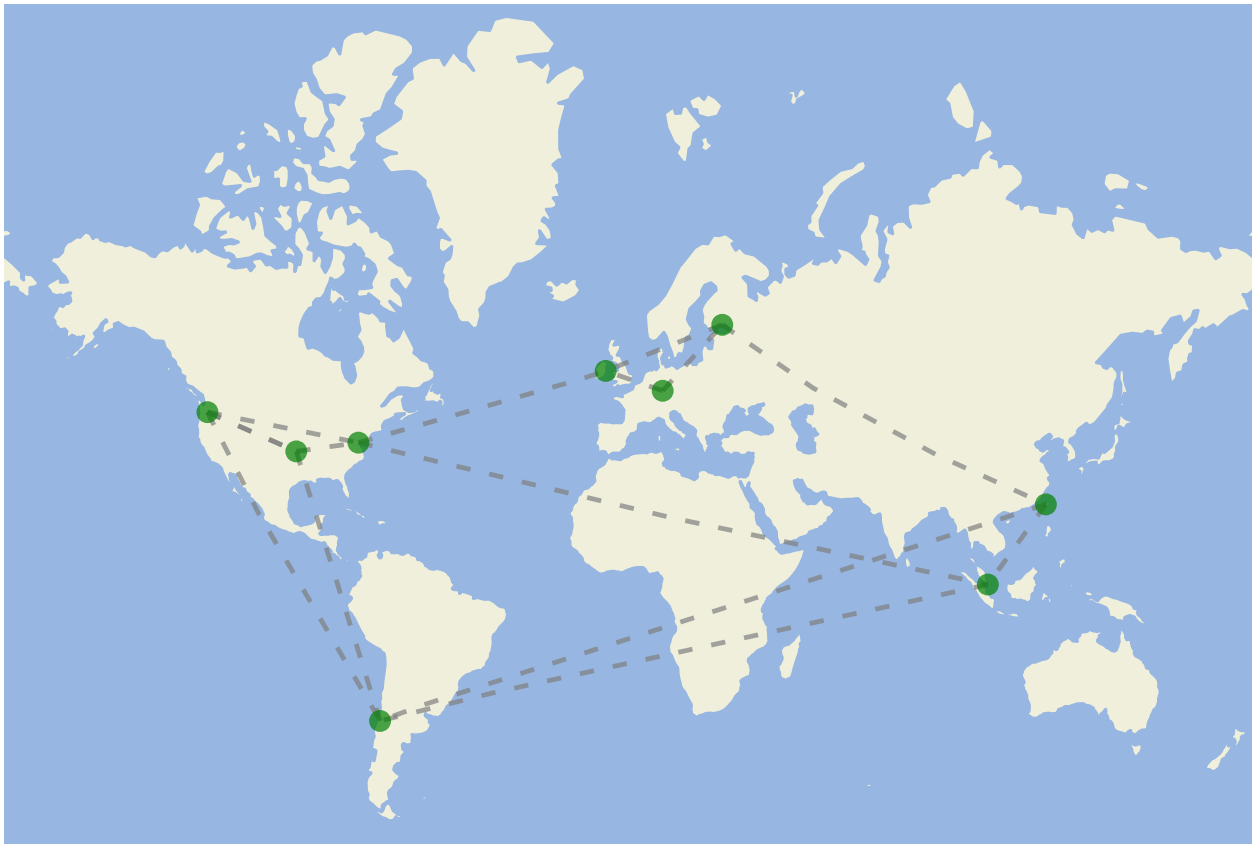


Trading performance for precision in a CRDT-based rate-limiting system

Master's thesis in Computer Science and Engineering

ANDREAS HENRIKSSON
SVANTE BENNHAGE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Trading performance for precision in a CRDT-based rate-limiting system

ANDREAS HENRIKSSON
SVANTE BENNHAGE

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2021

Trading performance for precision in a CRDT-based rate-limiting system

ANDREAS HENRIKSSON
SVANTE BENNHAGE

Supervisor: Vincenzo Massimiliano Gulisano, Computer Science and Engineering
Advisor: Zandra Norman, Spotify AB
Examiner: Philippas Tsigas, Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An illustration of a geo-replicated distributed system.

Typeset in LaTeX
Gothenburg, Sweden 2021

Trading performance for precision in a CRDT-based rate-limiting system
Andreas Henriksson & Svante Bennhage
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The past two decades have seen a rise in popularity of distributed cloud services, where some service is replicated and thus made available from several nodes to handle high loads of traffic and maintain operations at low latency. This rise in popularity has increased the need for solutions that enable fair distribution and limitation of these services' resources among its users, one of which is referred to as rate limiting. One of the rate-limiting solutions at Spotify, the company at which this thesis was conducted, employs optimistic replication and uses replicated counters to implement a token-bucket algorithm. The counters track the number of approved requests for users, rate limit the user based on this number, and operate in a conflict-free manner. The protocol for synchronizing nodes makes use of gossiping to lazily propagate synchronization messages to keep the costs of communication low. This solution strongly ties the system's ability to accurately rate limit users to the convergence rate between the replicated states.

A shortcoming of this solution is that users can exceed the limit by sending many requests to multiple nodes within a short period. To address this shortcoming, this thesis introduces and evaluates an alternative rate-limiting solution that enables the system to monitor the frequencies of users' requests continuously. These frequencies are then used to synchronize data for users that could reach their limit sooner than their states synchronize with gossiping. To determine the frequencies between users' requests, the requests' time of approval are stored as elements in a replicated queue. To uphold the token-bucket algorithm using this queue, each element additionally holds a timestamp of when the element should be removed. While storing the approval time of requests in a replicated queue caused the nodes to require more time to converge, our experiments have shown that it does not influence the number of approved requests, i.e., precision, in most cases. However, the precision could be impacted when dealing with requests sent at a specific rate. To prevent degradation of the precision, the system's parameters, for example, the rate at which nodes synchronize and the number of nodes, have to be set with care. The solution was shown to allow the rate-limiting system to enjoy the benefits of lazy synchronization for users who should not be rate limited. The rate-limiting system was simultaneously able to synchronize faster for misbehaving users, resulting in higher precision, at the cost of temporary increases to communication costs. Finally, the memory consumption was reduced to 13.6 % of the previous solution's memory usage when dealing with traffic representative of Spotify's system.

Keywords: rate limiting, gossip, precision, frequency, replicated queue, continuous distributed monitoring, delta synchronization

Acknowledgements

We would like to thank Vincenzo Massimiliano Gulisano for his honest and spot-on criticism for the report and presentations. We would also like to thank our supervisor Zandra Norman at Spotify for her continuous support and guidance, and Tim Wiklund for making the project possible in the first place.

Andreas Henriksson & Svante Bennhage, Gothenburg, July 2021

Contents

List of Figures

List of Tables

Listings

1

Introduction

The past decade has seen a massive rise in the popularity of cloud computing [11]. In essence, cloud computing is the use of one or more remotely located computers as hosts for purposes such as servers, software, and applications. It is an attractive alternative to using in-house computers as it allows for flexibility in terms of computing power, scalability, availability, and accessibility, among other things [12].

1.1 Geo-replicated distributed systems

In order to provide highly available services through the cloud, users should not have to rely on any individual data center or computer, i.e., node, for their use of such services. For example, in applications where user data plays a vital role in personalizing the user experience, having a single point of failure can leave users without access to the application. An example of such an application is Spotify, an audio streaming platform with 356 million monthly active users as of March 31, 2021 [1].

Cloud-based services often resort to geo-replication to scale with the increasing worldwide use [13]. An illustration of a geo-replicated distributed system can be seen in Figure 1.1. By replicating both data and computation across multiple data centers, the services are made both highly available and have the impact of failures reduced for users [13]. If a data center fails, users could still access the service elsewhere as both their data and the service are replicated and accessible in another location. For example, Spotify stores and replicates data about users' interactions with playlists and artists in order to personalize recommendations for music the user may be interested in [20].

While geo-replication is an attractive option for scaling distributed systems, it is not a seamless solution. The CAP theorem states that a distributed system cannot uphold more than two of the three properties consistency, availability, and partition tolerance at the same time [10]. These properties describe a distributed system's ability to maintain the same data on different nodes, serve requests at all times, and function during times when nodes in the system cannot communicate with each other. Since geo-distributed systems have a strong emphasis on maintaining high

Figure 1.1: An illustration of a geo-replicated distributed system

availability, this coincides with the consistency of such a system. Thus, these systems have to employ a consistency model[1] to synchronize the data stored on the replicas, also referred to as their states.

Implementations of consistency models may vary significantly between systems, for example, how frequently each node synchronizes, how many neighbors the nodes synchronize with at a time, and how the neighbors to synchronize with are chosen. Additionally, distributed and parallel systems are used for various purposes and operate under different circumstances in terms of both scale and implementation. As such, it is a natural conclusion that a single consistency model or implementation thereof will not be optimal or even practical for every possible system. However, it is common to use established consistency models as a guideline and modify them to fit the specific system's target in terms of availability versus consistency.

There are two major categories of approaches for achieving consistency in replicated systems: *strongly consistent replication* [16] and *optimistic replication* [23]. For this thesis, only the latter is of interest. The idea behind optimistic replication is to let replicas serve users without first having to synchronize with the other replicas and resolve any inconsistencies at a later time [23]. To this end, a consistency model known as *eventual consistency* is important for the correctness of systems that utilize this approach [24].

[1] A consistency model describes a strategy for reaching consistent states across nodes in distributed systems.

The approach of optimistic replication attempts to maximize the performance and availability of distributed systems. The systems that utilize this model are often designed to resolve conflicts between replicas that arise due to asynchronous modifications to their states. Such conflicts typically have to be solved by reverting replicas to some earlier state prior to the modifications which are then reapplied in an order that does not incur any conflicts [26]. However, depending on how a system is designed, one may utilize Conflict-Free Replicated Data Types (CRDTs). CRDTs are a category of abstract data types whose operations do not cause conflicts between replicas and are therefore able to utilize comparatively cheap consistency models for synchronization between replicas.

Optimistic replication schemes often utilize *gossip-based* dissemination protocols, sometimes simply referred to as *gossiping*, to slowly spread information between replicas. It has seen use in large-scale distributed storage systems such as Cassandra [17], and Dynamo [8], developed by Facebook and Amazon, respectively. Gossiping typically operates by choosing targets for exchanging information at random. In terms of spreading data in distributed systems, gossiping relies on similar probabilities that can be observed in how epidemic diseases spread through populations [14].

1.2 Rate limiting in distributed systems

The cloud infrastructure has massively increased the possibility to provide services to large numbers of users in a scalable way. However, since the resources of such services are shared between the users, there are plenty of reasons to limit the users' access to the resources. Examples of these reasons are prevention of resource starvation, management of policies and quotas, and flow control [4]. Solutions to this problem are broadly referred to as *rate limiting* and have historically been used for such things as protection against denial-of-service attacks [27], brute-force attacks, and limiting access to APIs [5].

With services transitioning to distributed infrastructures, it is only natural that algorithms for rate limiting must be adapted for these circumstances. An example of distributed rate limiting is presented by Raghavan et al. in [22], where a set of distributed traffic rate limiters collaborate to keep the global bandwidth usage of a cloud-based service below a global aggregate limit. One can naturally conclude that in systems similar to this one, the convergence rate between the rate limiters impacts the correctness of the rate limiting. Changes in bandwidth usage observed by one rate limiter have to be delivered to the other rate limiters so that they may increase or decrease the available bandwidth for the part of the service they are responsible for. Otherwise, their decisions do not reflect the system state, and the global limit could either be surpassed or not be utilized very well.

As previously mentioned, optimistic replication schemes often synchronize slowly to reduce communication costs. On top of this, the efficiency of rate-limiting algorithms relies on access to as much of the data used for rate limiting as possible, preferably

as early as it is available. Thus, for systems that utilize both optimistic replication and rate limiting, this is difficult to solve.

1.3 Background

Spotify has multiple backend services that respond to requests sent by users that are using the Spotify product. For some of these backend services, their resources are required to be fairly distributed among the users. To achieve fair distribution, the requests to those services first have to be approved by a rate-limiting system. If a user has sent too many requests within a too short time frame, the user's requests will be rejected.

In this thesis, a solution for performing rate limiting in a replicated manner was designed and evaluated in collaboration with Spotify. This solution is heavily inspired by one of the rate-limiting strategies used at Spotify, which uses CRDTs and an optimistic replication scheme to maintain a high availability and low costs for the synchronization of replicas. As a replicated rate-limiting system naturally relies on the rate of convergence between replicas, the solution was designed with two goals in mind. First, it was desired to preserve the same algorithm that this particular strategy for rate limiting at Spotify uses. In addition to this, the solution was desired to enable temporary increases of the convergence rate for the states of users that are likely to quickly breach a limit. This decision is based on analysis of the frequency users send their requests at and is done to prevent users from surpassing the system limit by too great of a margin before being limited across all nodes.

The structure of the rest of the report is as follows. Chapter 2 explains central concepts that appear throughout the report more in-depth. In Chapter 3, the strategy mentioned above for rate limiting is described. The design of a system that adopts this strategy acts as the baseline for the thesis, to which the proposed solution will be compared. This chapter also describes the aim, problem statement, and limitations of the thesis. Chapter 4 details the design and implementation of our solution. It also explains how the proposed solution maintains the same rate-limiting algorithm as Spotify's system. On top of this, it describes the framework used for testing and comparing our solution to the baseline, and the data sets used for evaluation. Chapter 5 lists all the results with accompanying graphs for evaluation combined with a discussion of the results. Chapter 6 presents ideas and solutions similar to the one presented in this thesis. Finally, Chapter 7 concludes with a summary of the report and a suggested future research direction.

2

Preliminaries

The following section describes two core concepts that are later used in the thesis more in-depth.

2.1 Conflict-free Replicated Data Types

Conflict-free Replicated Data Types, introduced by Shapiro et al. in [25], refer to data types that are designed to provide *strong eventual consistency* in distributed systems. Strong eventual consistency is an extension of eventual consistency, a form of weak consistency commonly found in systems that replicate data. A system that replicates objects is said to uphold eventual consistency if all replicas of an object eventually converge to the same value [24].

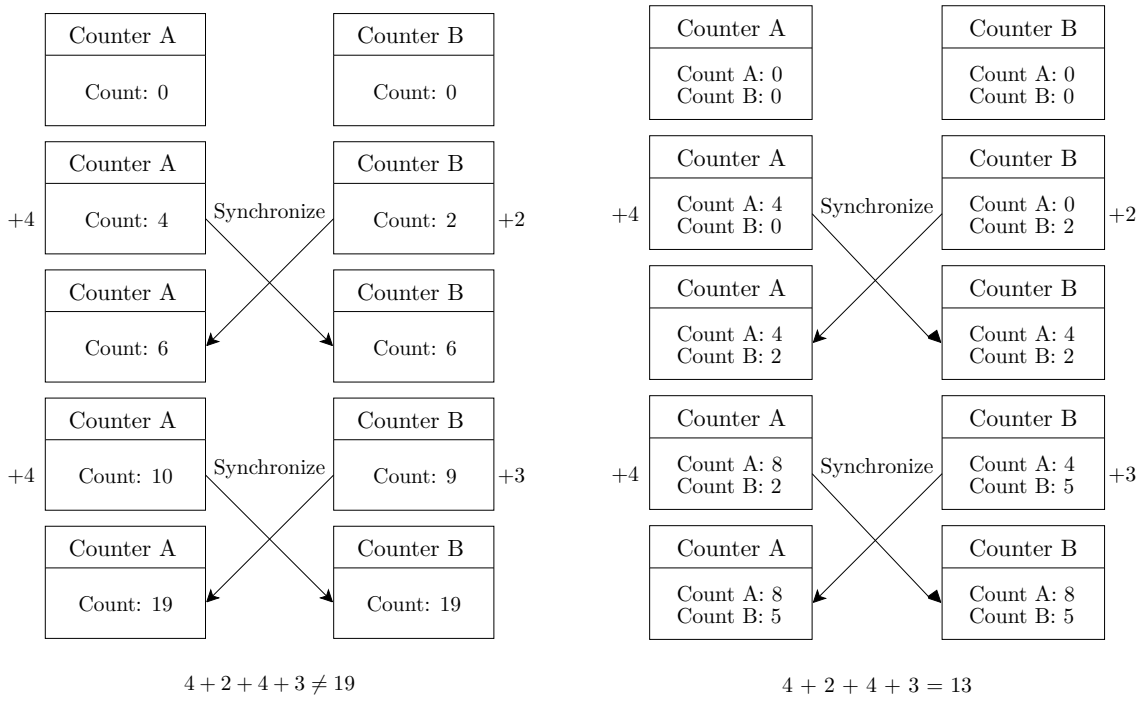
In order for a distributed system to uphold strong eventual consistency, it has to fulfill two criteria. First, it has to be eventually consistent. Secondly, it has to guarantee that any two replicas that have delivered[1] the same updates have equivalent states. When employing eventual consistency, replicas are typically allowed to change their state independently, which can introduce conflicts between the replicas' states. However, there is a difference between differing states and conflicting states. Conflicts can arise when two or more states synchronize with each other. By having the operations of a system employing specific semantics and mathematical rules, conflicts can be avoided altogether.

At the time of this thesis, there are two types of CRDTs we know of, namely *state-based* CRDTs and *operation-based* CRDTs. In simple terms, state-based CRDTs synchronize by sending their states to each other and merging the local state with the received one. Operation-based CRDTs synchronize by sending the operations they performed on their local state to be performed on the states of every other replica.

[1] A delivered update is an update part of the causal history of updates that have been applied on a replica [25]

2.1.1 Concurrency semantics

Abstract data types operate with well-defined behaviors according to their interfaces. For example, a simple counter could be represented by an integer, which is hereafter referred to as the *state* of the counter, and an addition operation. Consider an execution of a program that continuously updates the state of the counter using only the operation for addition. Addition is a *commutative* operation, meaning that the result of this operation is not affected by the order of the elements it is performed on.



(a) a single value

(b) one value for each node

Figure 2.1: Two examples of concurrency semantics for a replicated counter

Now consider a counter with a state identical to the previously mentioned counter, but instead replicated across two nodes A and B. Replicas A and B perform additions on their respective states, individually from one another, and the state of the counter is now represented by the sum of additions made on both A and B. Consider the example of Figure 2.1. In (a), a single value is updated concurrently on the two replicas. Since the state of the counter is the sum of all additions made on each replica, it is apparent that a single value is not sufficient to maintain an accurate sum of additions on all replicas past the first synchronization. Thus, the semantics of a single instance of a counter do not naturally translate to a replicated counter. In (b), the replicas instead hold a value for additions made on each replica. When synchronizing with the other node, the sum of additions made on each node is shared and updated. This way, each node can summarize the value of its local additions and the additions made on the other node to calculate the correct sum of additions to the counter.

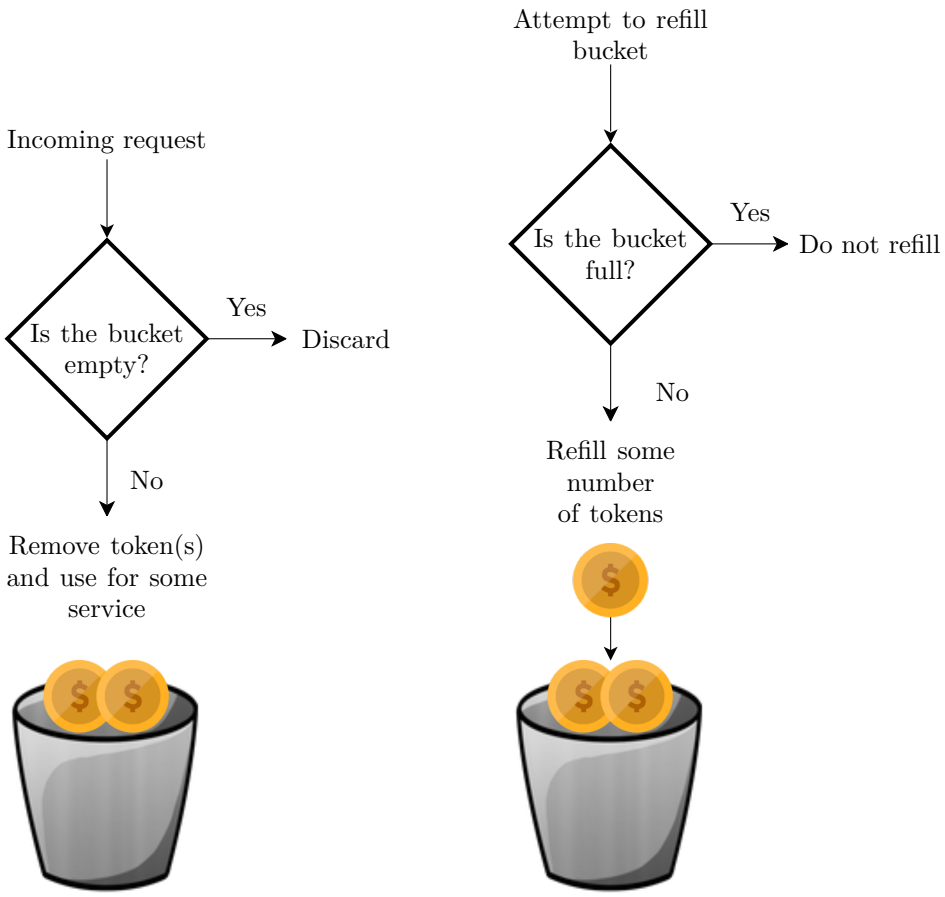
In the context of replicated data types, concurrency semantics refer to how concurrent alterations of replicated objects are dealt with in order for the states of all replicas to converge. The concurrency semantics are thus crucial to both the behavior and correctness of the data type. For most data types, there is not one concurrency semantic that naturally is suitable over another – it has to be chosen based on the application of the CRDT [21].

2.2 Token bucket algorithm

The token bucket algorithm emulates a physical bucket or dispenser and is often used for regulating some form of traffic. The algorithm has been used for flow access control in cloud storage [3], gateways for traffic control for micro-services [19], and in network utility maximization problems [9], among other things. It is a commonly used algorithm for rate limiting.

While the implementation of the algorithm can vary, the basic premise remains the same and is shown in Figure 2.2. A bucket is used to hold up to a maximum number of tokens. For as long as the bucket is not empty, tokens may be consumed and traded for some resource or service. Thus, a request made to a bucket that is either empty or does not contain enough tokens to pay for the requested service will be denied, effectively rate limiting the sender of the request. Whenever the bucket is not full, tokens are refilled according to some rate. The relation between the size of the bucket and the rate at which it is refilled therefore defines the rate at which tokens can be consumed from the bucket. Since the algorithm can be used in many different contexts as mentioned above, the services and resources that the bucket provides access to varies greatly depending on where it used. For example, a token bucket could be used to limit the number of queries to a database.

The token bucket algorithm is effective at shaping traffic. Once a bucket is empty, it will no longer permit any requests until it has been refilled with new tokens. However, an essential property of the algorithm's behavior is that it does not prevent bursts of traffic. A full bucket can be consumed rapidly with no regard to how the service which the tokens pay for is impacted.



(a) Rate limiting **(b)** Refilling the bucket

Figure 2.2: An example of the token bucket algorithm

3

System Model & Problem Statement

This chapter first provides a detailed description of the existing system design and describes an inherent flaw with the design. The aim and problem statement of the thesis is then presented. Finally, the metrics that will be used in the evaluation are introduced.

3.1 A token bucket design using counters

One of the methodologies for rate limiting at Spotify is to use a token bucket algorithm, as described in Chapter 2. This section describes a system, hereinafter referred to as *the system*, that employs the same algorithm and data structure for rate limiting as Spotify's system. The differences consist of minor performance changes that will not be described for confidentiality reasons.

The system keeps track of how many requests each user has had approved so far and the maximum number of requests each user is allowed to have approved until the present time. It does this by assigning a token bucket for every user. Approving a request consumes a token, which is removed from the bucket. A token bucket has a limit for how many tokens it can hold at any time, and requests are denied if the bucket is empty. It also has a rate at which tokens are refilled whenever the bucket is not full. However, saying that the bucket is refilled with tokens is a simplification. In reality, the act of refilling the bucket is represented by incrementing the limit of the bucket, thus increasing the disparity between the limit and the number of consumed tokens.

In the system, a token bucket is represented by three values: a value for the total number of tokens removed from the bucket, hereinafter referred to as *token counter*, another value for the maximum number of tokens a user is allowed to have consumed up until the present time, hereinafter referred to as the *limit*, and a timestamp of when the limit counter was last updated, hereinafter referred to as the *refill timestamp*. Two parameters are used to alter these values: a *refill rate*, which determines how many new tokens per time unit should be added to the limit, and a

capacity, which determines how many tokens the bucket can hold.

Whenever a new request arrives, the system first determines if a request originates from a new user or a user that already has a dedicated token bucket. As seen in Listing 3.1 on lines 2–3, the system creates a new token bucket when the request originates from a new user. The bucket is initialized with its limit set equal to the value of the capacity parameter. In the latter case, the system instead determines if and how much the user's token bucket limit should be increased. It does this by comparing the timestamp of when it was last refilled to the current time. The difference between the two multiplied with the refill rate capped by the capacity parameter yields the number of added tokens the user is allowed to consume.

After the bucket has been refilled by the calculated amount, the system checks if the bucket is empty, which it does by asserting that the limit is not lower than or equal to the number of consumed tokens. If the bucket is empty, the request is rejected. In all other cases, a token is removed from the bucket through the `consume` method before the request is accepted.

```
1  on received request:
2      if user is new:
3          user.token_bucket = new token_bucket(limit = config.capacity)
4      else:
5          user.token_bucket.refill()
6          if user.token_bucket.is_empty():
7              reject request
8
9      user.token_bucket.consume()
10     accept request
11
12 token_bucket:
13     refill():
14         now = current_timestamp()
15         delta_limit = (now - refill_timestamp) * config.refill_rate
16         if delta_limit > 0:
17             limit += min(delta_limit, config.capacity)
18             refill_timestamp = now
```

Listing 3.1: Pseudo-code for receiving a request

To deal with the large scale of Spotify's user base, the rate-limiting system is distributed and replicated, i.e., all nodes should be able to process any request from any user. Processing requests on different nodes causes their states to diverge. To achieve the goal of having high availability while having consistent states across nodes, the system makes use of CRDTs. More specifically, the token counter is represented as a map of node identifiers for the nodes that have processed consume requests for this user. Each identifier maps to a value representing the number of requests the respective node has accepted for this user. As shown in Listing 3.2, consume requests increment the value for the local node by one. The total number of approved requests consists of the sum of the map values.

```
token_bucket:
    consume():
        token_counter[local_node_id] += 1
```

Listing 3.2: Pseudo-code for the consume method

To keep consistent states, nodes periodically update their neighbors with changes to their state. To reduce the cost of synchronization and communication, the nodes keep a delta state for each neighbor. The delta states of a node contain the part of the state corresponding to the users that have consumed tokens since the node last synchronized with its respective neighbors. A synchronization message contains the delta state for a neighbor and is sent at a fixed interval. Once the message has been sent, the delta state is then removed from the sender. Which node to update is chosen according to flat gossiping, i.e., uniformly at random.

```
on received delta_state from sender:
    updates = {}
    for user, received_bucket in delta_state:
        if user not in local_state:
            local_state[user] = received_bucket
            updates[user] = received_bucket
        else:
            local_bucket = local_state[user]
            local_state[user] = local_bucket.merge(received_bucket)

            if local_bucket != local_state[user]:
                updates[user] = local_state[user]

    for node not sender, delta_state in delta_states:
        merge_updates(updates, delta_state)
```

Listing 3.3: Pseudo-code for receiving a delta state

```
token_bucket:
    merge(other_bucket):
        merge_token_counter(other_bucket.token_counter)
        merge_limit(other_bucket.limit,
                    other_bucket.refill_timestamp)

    merge_token_counter(other_token_counter):
        for node_id, node_value in other_token_counter:
            if node_id in token_counter:
                new_value = max(token_counter[node_id], node_value)
            else:
                new_value = other_token_counter[node_id]
            token_counter[node_id] = new_value

    merge_limit(other_limit, other_refill_timestamp):
        if other_limit > limit:
            limit = other_limit
            refill_timestamp = other_refill_timestamp(
```

Listing 3.4: Pseudo-code for merging two user states

When a node receives a synchronization message, it is merged with the node's local state by iterating over the received delta state. The merged token counter for every user will consist of the union of the local state and the delta state, as seen in Listing 3.3. In the case of conflicting values for a node identifier, the resulting token counter will keep the greatest value, which is shown in Listing 3.4 on line 10. The limit and the refill timestamp are also merged, where the greatest limit and its accompanying refill timestamp are kept, which is shown on lines 16–18 in the same listing.

Since the perception of time is used in multiple nodes, the decisions of the nodes may vary if their clocks are too different. For this reason, synchronized clocks are assumed.

After merging the state for a user, its token bucket is added to a map that holds updates if the merge updated the token bucket. Once this process has been repeated for each user in the delta state, the map of updates is merged with the delta state of every neighbor aside from the node that sent the initial synchronization message.

3.1.1 The problem of exceeding the limit

The current system is designed such that rate limiting is performed ad hoc based on the state of whichever node at which any request arrives. Any request that is approved by any node causes the state of the node to diverge from every other node in the network. Between the time of a node approving a request and the time at which the neighbors are updated with information about this request, neighbors operate as normal while unaware of the approval of this request. Consequently, a node may approve enough requests for a user to reach their limit before the node is made aware of requests the user has had approved by any other nodes. The result is that users can exceed the limit when they are served by more than one node. The more nodes the system consists of, the more a user can exceed its limit if many nodes process its requests.

While gossiping is an option suited for reducing the communication cost, it does so by trading a higher convergence rate by limiting the number of updates. This trade-off worsens the problem of users exceeding the limit as the node states diverge further from each other before they eventually synchronize.

3.2 Aim

The thesis aims to decrease the time it takes for a node in the distributed system to make the same decision as a centralized system that performs the same task and receives the same requests. This decrease should be accomplished while still mainly relying on gossiping for disseminating information.

The goal mentioned above could be accomplished by making sure nodes have the

most critical data sooner while not receiving all data as quickly. One way to decrease the time is to make sure the convergence rate is high. As a result, rate-limiting decisions across nodes would achieve higher precision. In this context, the precision of a decision refers to if it is correct based on the user's state of the entire system, not only the local user state of a node. To improve the convergence rate, it might be possible to make smarter decisions about which neighbor to synchronize with compared to uniformly at random. Suppose such a decision is feasible and results in each node sending an equal or larger number of updates on average with each synchronization step. In that case, the improvement of such a decision can be explained by the receiving neighbor having the same or more information sooner than before. We hypothesize that if all nodes apply this rule, the whole network of nodes will converge at a faster pace.

Another way is to give the system access to more nuanced data that could provide information such as the frequency between user requests, allowing the system to take action proactively.

Such an action could be to immediately update neighbors of a user's state if the frequency between this user's requests is deemed suspicious. By prioritizing the propagation of critical data, we aim to reduce the time between when the distinction of suspicious behavior can be made and when the data has reached the neighbors. Thus, the neighboring nodes would make informed decisions sooner than if the data were to propagate through normal means. This reduction of latency could, in turn, result in the user being rate-limited, for example, if the user has sent requests to many other nodes simultaneously.

To summarize, it is desired to increase the overall convergence rate and give the system the means of detecting specific user behavior to propagate critical information quicker, specifically in those cases.

3.3 Problem statement

The information of how frequently a user has had requests approved in the recent past can be represented by a queue holding the timestamps of when the requests were approved. To maintain the same availability as the baseline, replicas of the queue should not have to synchronize prior to adding timestamps to their local state, and instead be allowed to immediately add them to the end of the local state. For this reason, a timestamp's order in a replica of the queue may have to be changed when replicas are synchronized. We thus refer to the queue as *relaxed*, as the end of the local state at one replica is not necessarily the end of the queue given the timestamps of all replicas.

To realize the goal of a higher convergence rate and propagation of data related to suspicious user behavior, the following questions should be answered:

Q1 How can a relaxed, distributed, and delta-synchronized queue holding the

timestamps of the recently accepted requests per user be designed and implemented as a means for both rate limiting and frequency analysis?

Q2 Is the precision affected by using the queue, where consumed tokens are represented as individual elements, compared to the baseline?

Q3 Compared to the baseline, how is the precision affected by identifying and rapidly distributing potentially impactful data?

Q4 As opposed to the baseline, the queue enables nodes to determine how many updates their neighbors are missing by counting the number of elements in their delta states. Is the precision impacted by continuously synchronizing with the neighbor that is considered most behind in updates?

Moreover, when these questions have been answered, the system's performance will be evaluated with these additions compared to the unmodified system in terms of memory and bandwidth consumption and the convergence rate.

3.4 Limitations

This section describes the limitations that have been made for the project.

3.4.1 Upholding the definition of being a CRDT

This thesis is concerned with constructing a solution compatible with the system's way of propagating information using delta states. Therefore, proving whether the queue conforms to the requirements of being a CRDT or not is out of the scope for this thesis.

3.4.2 Proactive actions

Using frequency analysis, the nodes will be able to identify requests that have arrived suspiciously close to each other. When this happens, there are two actions a node could take. The first option is to propagate this information to the node's neighbors faster than by normal means. This option trades bandwidth for higher precision.

The other option is for the node to estimate the overall number of consumed tokens and possibly actively rate-limit the user even if the local limit has not been reached. This option may result in false positives since users might send a few frequent requests that happened to arrive at just one node. In that case, the node would end up rate-limit the user incorrectly. On the other hand, this approach can potentially react faster than simply waiting for the nodes to communicate first.

The thesis aims to investigate the feasibility of using a queue rather than counters to perform rate limiting. For this reason, it is out of the scope to find the most

accurate trigger rule. Instead, a rule that would ideally demonstrate the usefulness of the queue without necessary being optimal will be used. As such, it is desired to avoid any false positives since it otherwise might put the proposed design in a bad light because of simple rules. For that reason, the action of propagating information will be used as opposed to the one that prematurely rate-limits users.

3.5 Metrics

In this section, the metrics that will be used to evaluate the proposed changes are described.

Number of correctly rejected requests

When evaluating the performance of the proposed changes, the number of correctly rejected requests is ultimately what determines how much better they perform compared to the existing design using counters.

Memory and bandwidth consumption

Aside from the metric mentioned above, it is desired to see how much higher memory and bandwidth consumption is required for the queue to be used compared to the design using counters.

Convergence time

When evaluating the deterministic gossip approach compared to the traditional random gossip approach, it is desired to measure how long it takes for the two cases to converge. It is also interesting to evaluate how the queue's convergence time compares to the design using counters.

4

Methods

This chapter describes the process of fulfilling the aim of the thesis. Various design choices for a relaxed distributed delta-synchronized queue, hereinafter referred to as *queue*, used for rate limiting, are described and motivated. These design choices are followed by implementation details of the queue and how it is used for both what is referred to as Least Informed Neighbor First-gossiping and frequency analysis of approved requests. Finally, the process of testing the queue for the metrics found in Section 3.5 in order to evaluate its performance is described.

4.1 Queue design

To determine the frequency of accepted requests, the user state can be represented by a queue holding the timestamps for when recently accepted requests were processed, sorted in ascending order. As the queue is intended to be used for rate limiting as well, the queue size can be used to determine how many requests have been accepted. The queue size can be used to enforce a limit by comparing the size to the number of requests allowed to be accepted until the present time, similar to the baseline. However, in contrast to the baseline, which uses a fixed-sized map of integers to store how many requests a user has had accepted, the queue could grow indefinitely large as new elements are added. To prevent ever-growing user states while still enforcing a limit on how many requests should be accepted within a time frame, elements could reside in the queue for some time before being removed. Thus, removing an element from the queue can be equated to allowing another request to be accepted. Therefore, the limit can be enforced by the rate at which elements are removed from the queue.

For the process of synchronizing the replicated queues, a user state in a received delta state and the local user state of the receiving node could be merged, like so: `merge([1, 3, 5], [2, 4])` $\rightarrow$ `[1, 2, 3, 4, 5]`. As nodes will replicate the same queue, a single node might receive the same element from multiple neighbors as they synchronize. A problem with this is that multiple elements that stem from the same request would account for more than one slot in the queue after merging two user states containing such duplicates. To avoid this problem, elements with the same timestamp are assumed to stem from the same request, meaning du-

plicated values are treated as the same element: `merge([1, 2, 5], [2, 4])` → `[1, 2, 4, 5]`.

Treating duplicated timestamps as the same element could be problematic if the time unit is not sufficiently small. This is because multiple requests from the same user might be assigned the same timestamp. The result is that nodes would only account for one of the multiple accepted requests after synchronizing, allowing the user to use more resources than what is specified by the limit. As long as elements are represented as a time unit with high resolution, such as milliseconds, the risk of two different requests from the same user clashing is greatly reduced. Suppose this would turn out to be a problem despite a high resolution. In that case, a solution could be to assign a random identifier to every element, where the risk of collision is close to non-existent. However, that would increase the required memory consumption significantly.

Since the rate at which elements are removed from the queue dictates the rate at which requests can be served, a suitable algorithm to enforce this rate had to be used. The two following sections discuss two algorithms that were decided between. The algorithms' behaviors are described in terms of rate limiting and how the queue needed to be adjusted to be compatible with each algorithm.

4.1.1 Sliding-log queue

An elementary approach to removing elements at a specific rate is to offset an element's removal to a constant value from its timestamp of inception into the queue. This is the approach of the *sliding log algorithm* [7]. The sliding log algorithm enforces a rate by first storing the timestamps for approved requests in a sliding window[1]. New requests are then either approved or rejected based on whether or not the average frequency between requests in the sliding window is lower or higher than the enforced rate. The average frequency is obtained by dividing the delta between the oldest timestamp in the window and the current time by the number of elements in the window.

For illustration, consider the following queue: `[1, 3, 4, 5]`. If elements were to live in the queue for five time units, the first element would be removed at all nodes after timestamp `6`. Following this, the element should also be cleared from any delta state.

In an attempt for this design to grant at most the same amount of resources as the design outlined in Section 3.1, the lifetime of elements in the queue could be set according to Equation (4.1). In Figure 4.1, each request is represented as a dot with a unique color and where a strike-through dot represents the freeing of the request's allocated space. In this figure, a capacity of 4 tokens and a refill rate of 1 token per

[1] A sliding window holds some number of elements. The window slides past elements according to some criteria, such as elements having resided in the window for some period of time, which leaves room for new elements in the window.

second are used. It is shown that Equation (4.1) would allow both algorithms to enforce the same rate given that new requests are received infinitely often.

$$\texttt{lifetime} = \frac{\texttt{capacity}}{\texttt{refillRate}} \tag{4.1}$$

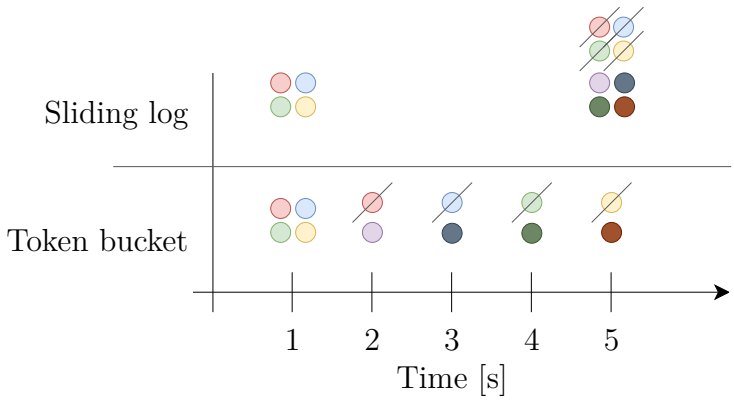


Figure 4.1: The same amount of accepted requests for the sliding log and token bucket algorithms given requests received infinitely often

While Figure 4.1 illustrates that the sliding-log algorithm can uphold the same rate as the token-bucket algorithm, it also shows that the token-bucket algorithm will approve different requests than the sliding-log algorithm. This is due to the token-bucket algorithm removing elements at a constant rate rather than after a fixed time from the elements' inception, meaning that elements reside longer on average in the sliding window compared to the token bucket. To illustrate this, recall the same parameters as mentioned above that yield a lifetime of $\frac{4}{1} = 4$ seconds. It is apparent that tokens will remain consumed for a longer period of time in the queue compared to in the solution using counters.

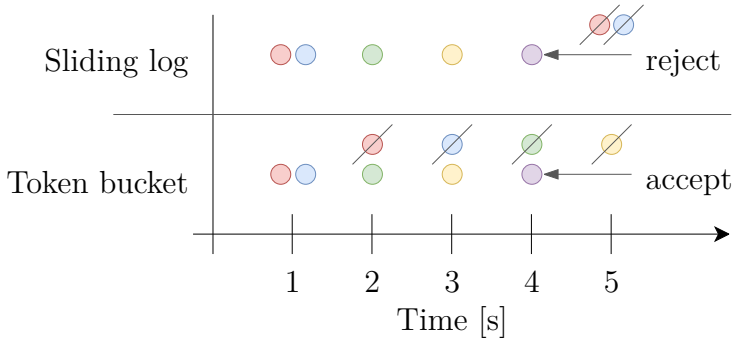


Figure 4.2: A difference of output between the sliding log and token bucket algorithms

This implies that the queue will be full for a longer period, resulting in users possibly having their requests rejected when the requests would otherwise be accepted if the

design using counters was used. Using the values stated above, this difference is shown in Figure 4.2. The figure shows that the queue will reject the request coming in at timestamp 4, whereas the token bucket solution will not. This means that given the same input, the two designs can produce different outputs. Additionally, by observing Figure 4.2, the sliding-log algorithm rejects one request, whereas the token-bucket algorithm rejects none. If this traffic is then repeated over and over with long enough pauses in-between to let the queues become empty, the total number of rejections for the sliding-log algorithm will grow faster than for the token-bucket algorithm.

4.1.2 Token-bucket queue

The previous section showed that the sliding-log algorithm can reject more requests than the token-bucket algorithm given the same traffic. It was therefore of interest to evaluate how a design using the token-bucket algorithm would compare against the design using the sliding-log algorithm. It is described in Section 4.1 that elements should be removed from the queue in order to prevent the state from growing indefinitely. For the token-bucket algorithm, elements would be removed from the queue at constant rate as the removal of an element equates to the refill of one token. However, in order for this rate to be constant for every element starting from when a user's first request was approved, the initial timestamp should offset this rate. This offset has to be preserved as the element that initiated it will eventually be removed. For this reason, storing only the timestamps for when requests are approved is not sufficient to uphold token-bucket behavior in a replicated queue.

An alternative design of a queue that uses the token-bucket algorithm is one where the elements instead consist of two timestamps: a *time-of-birth* and a *time-of-death*. When a new element is to be added to the queue, its time-of-birth is assigned to the current system time. The queue is also sorted ascending based on this value. The element's time-of-death is set based on elements that are already in the queue. This is shown in the following equation, where the variable `Q` denotes the queue:

$$\texttt{timeOfDeath} = \begin{cases} \texttt{now()} + \frac{1}{\texttt{refillRate}}, & \text{if } \texttt{Q} = \emptyset \\ \texttt{Q.last().timeOfDeath} + \frac{1}{\texttt{refillRate}}, & \text{otherwise} \end{cases} \tag{4.2}$$

According to Equation (4.2), elements have their time-of-death assigned to the time until the next refill given that it is a user's first approved request. Otherwise, the time-of-death is assigned to one refill after the most recently added element in the queue. Using this equation, elements will thus be removed according to a constant rate as opposed to a fixed time from their inception.

In this design, specific elements are identified by their time-of-birth timestamp, similar to the sliding-log design. This means that two requests processed by two different nodes but assigned the same timestamp will be considered the same element.

By assessing Equation (4.2), an element's time-of-death is thus relative to its position in the queue. As the position may change with the introduction of elements from other nodes, updating the time-of-death for elements accordingly is essential. For that reason, the merge works similarly to the sliding-log design when receiving a synchronization message from another node but with minor differences. One difference is that the time-of-death for each element is updated during a merge. This process of updating elements' time-of-death means that all elements succeeding the first inserted element from a delta state get their time-of-death changed.

Another difference is when deciding which element should be placed first in the queue when merging two queues. When two such candidate elements are deemed to stem from the same request, i.e., having identical time-of-birth, the greatest time-of-death dictates the instance that should be used. The greatest time-of-death is used because that value comes from the node with more elements older than the current element, resulting in the most accurate decisions.

To illustrate the outcome of a merge, consider the following example, where the refill rate is $\frac{1}{20}$:

```
merge([{1, 21}, {3, 41}, {5, 61}], [{2, 22}, {4, 42}])
     → [{1, 21}, {2, 41}, {3, 61}, {4, 81}, {5, 101}]
```

In this example, the element from the second queue with the time-of-birth 2, `{2, 22}`, is inserted at position 2 in the merged queue with an updated time-of-death. When that happens, all succeeding elements from both input queues also have their time-of-death updated when merged into the output queue. As previously mentioned, this results in that all modified elements are added to the delta states of the node's neighbors. Adding all modified elements as opposed to just new ones implies that this design will propagate more data than the sliding-log design, as the latter will only propagate an element to every neighbor at most once.

This also means that elements that have been removed from one node might need to be readded. This is because the element might have been added locally at one node where it was assigned a low time-of-death and then propagated to another node with a larger queue, thus updating the element with a higher time-of-death. When this happens, the second node would propagate this intel to the first node. If the first node already removed the element, it has to be readded.

4.1.3 Choosing a design

As previously mentioned, the goal of the thesis was to improve the rate limiting in a system that makes use of gossiping and CRDTs. When evaluating the success of the chosen design to compare with the baseline, the results are less open for interpretation if the designs operate with equal behaviors. In the context of this thesis, the behavior of a design is defined to be represented by which requests would be rejected in a centralized application using the design. By this definition, injecting a

set of requests with predetermined timestamps into the system will deterministically decide which requests should be rejected. Consequently, if two designs provide equal output given the same input, their behaviors are considered equal.

In the previous section, an example was shown where the sliding-log algorithm rejected a different amount of requests than the token-bucket algorithm given identical requests. Suppose that the queue using the sliding-log algorithm was to be compared with the baseline using the token-bucket algorithm. In that case, it would have been difficult to conclude how the results were affected by the implementation of the queue and how they were affected by the algorithm's behavior. On the other hand, the token-bucket queue uses the same algorithm as the baseline and only differs in terms of its implementation. For these two reasons, it was decided to evaluate the token-bucket queue design.

4.2 A token-bucket queue system model

As explained in Section 4.1.3, it was decided to design a distributed and replicated relaxed queue that preserves the rate-limiting behavior of the original system design. The following sections describe the implementation of this system model in detail. Section 4.2.1 describes how the token-bucket queue operates without frequency analysis or the new way of selecting neighbors to synchronize with. Section 4.2.2 motivates how the implementation described in Section 4.2.1 upholds the behavior of a token bucket, identical to that of the original system model, and a few other choices in terms of the performance of the queue. Section 4.2.3 describes the extension of frequency analysis, Section 4.2.4 describes the extension for deterministically choosing which neighbor to synchronize with, and Section 4.2.5 describes the implications of joint use of both extensions.

4.2.1 Plain token-bucket queue

On the highest level, the rate-limiting design using a token-bucket queue consists of several instances of the same data structure, which is shown as the type alias `State` in Figure 4.3. The state is a map, where each key is an ID for a user that has sent at least one request, and the value is a queue, represented as a linked list, that holds elements that represent requests that the system has approved. These elements consist of two timestamps: one for when the request was approved and one for when the element should be removed from the queue.

The state instances may use the same data structure, but their behaviors are categorized in two ways as they differ in what they are used for and how elements are added to and removed from them. The first category is referred to as the local state of a node. The local state is simply one instance of this data structure. The second category is constituted of delta states which are used for synchronizing replicas of the local state.

A node that receives a request first checks whether the user ID of the request exists

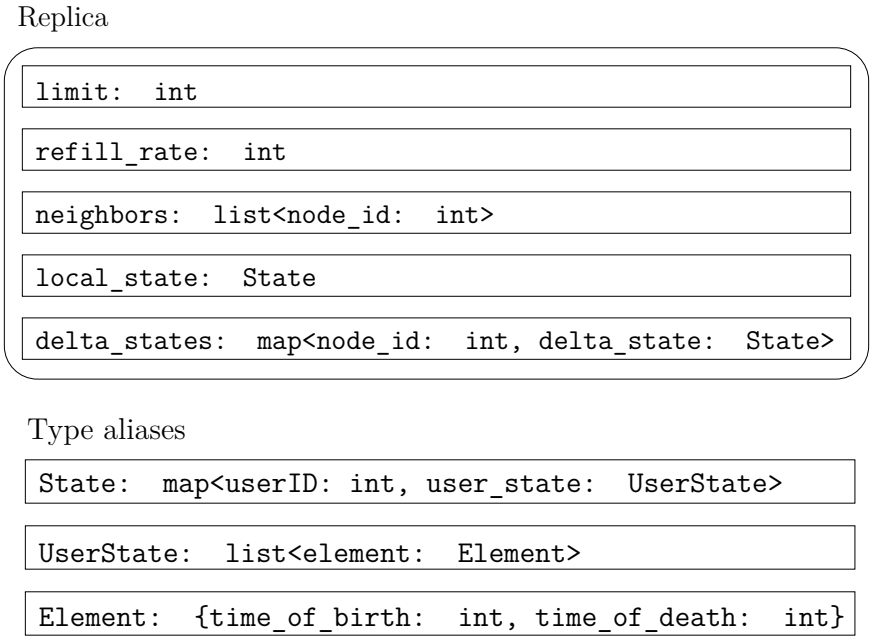


Figure 4.3: An illustration of what information a replica stores

in the local state. If the user ID already exists, any elements in that user state with a time-of-death older than the current value of the system clock are removed from the queue. This procedure is seen in Listing 4.1. The request is then approved based on if the length of the queue that the key is mapped to is smaller than its designated limit. If the key is not found in the local state, the request is approved, and a key with a corresponding empty list is made for this user. In both scenarios, where requests are approved, an element is created and put on the tail end of the queue.

```
1 token_bucket_queue:
2     refill():
3         while queue not empty and queue.head().time_of_death > now():
4             queue.remove_head()
```

Listing 4.1: Pseudo-code for refilling a user state

Each node has a delta state for each neighbor. Elements that are added to the local state through user requests are immediately added to the tail end of this user's queue in each delta state.

Replicas constantly strive for consistent local states, which is what the delta states are used for. The contents of a delta state are occasionally delivered to the neighbor the state is designated to. The neighbor then merges the contents with its local state. The changes that result from a merge between a received delta state and a local state, which will be covered in the following paragraphs, are then integrated into each delta state on the receiving node.

Merging the local state of a node with a delta state is performed according to

Listing 3.3, with the difference being the merge function of the user state. If a user in the delta state does not exist in the local state, a key for the user is added, which maps to the queue provided by the delta state. If the local state contains data for the user, that data is merged with the corresponding data in the delta state. The merging of two user states is similar to how the merge-sort algorithm merges two lists – a comparison of the heads of two lists decides the next element that should be moved to the resulting list. The following algorithm is used to determine the order of the queue resulting from the merge:

```
1  token_bucket_queue:
2      merge(other_queue, into_local_state):
3          result = []
4          while local_queue not empty and other_queue not empty:
5              local_head = local_queue.head()
6              other_head = other_queue.head()
7              if into_local_state and
8                local_head.time_of_birth == other_head.time_of_birth:
9                  result.append(
10                     element_with_largest_time_of_death(local_head,
11                                                        other_head)
12                 )
13                 local_queue.remove_head()
14                 other_queue.remove_head()
15             else if local.head.time_of_birth < other_head.time_of_birth:
16                 result.append(local_head)
17                 local_queue.remove_head()
18             else:
19                 result.append(other_head)
20                 other_queue.remove_head()
21
22         result.add_remaining_elements(local_queue)
23         result.add_remaining_elements(other_queue)
24         return result
```

Listing 4.2: Pseudo-code for merging two queue user states

According to lines 15–20 in Listing 4.2, the queue that results from a merge between two user states will be sorted based on which element was added earliest to either of the two nodes' local states. Lines 7–14 show that it will also not contain duplicates from both queues, as duplicate elements originating from different nodes are treated as the same element. The exception to this case is when not merging a received delta state into the local state, which will be explained shortly.

```
1  on merged head_element into result:
2      head_element.time_of_death = max(
3          result.last().time_of_death + 1 / config.refill_rate,
4          head_element.time_of_death
5      )
```

Listing 4.3: Pseudo-code for updating a merged element's time-of-death

Apart from merging the two user states, two additional things are done during the merge operation. One of the two is updating the elements' time-of-death as they

are merged into the new queue. This procedure is shown in Listing 4.3, where an element's time-of-death is updated based on the most recently merged element in the queue, as long as the calculated value is not lower than the element's current time-of-death. The other is to keep track of which elements in the queue resulting from the merge are new to the local state or have had their time-of-death updated by the previously described action. Finally, after a user state has been merged, the user state is refilled using the same refill method shown in Listing 4.1. If the refilled user state is an empty queue, the entry in the state for the user is removed. The entry would be removed regardless of whether the merge operation was carried out on a local state or a delta state. The latter case will be explained in the following paragraph.

After a delta state is merged with a node's local state, the delta states for the node's neighbors are updated to reflect the newly updated local state. Instead of merging each local delta state with the received delta state, the list of new and updated elements for each user is instead merged with the delta states. In this merge, elements that reside in both the list of updates and the delta state that share the same time-of-birth should not be skipped. That is because all elements in the list of updates are known to be new and originating from different nodes. Hence, `into_local_state` in Listing 4.2 is used to control this behavior, as seen on line 7. The merging of a local delta state and the list of updates is done for every neighbor. To avoid sending the same information back to the sender, duplicates between the sender's delta state after merging the updates and the received delta state are removed from the former. Finally, all delta states of the neighbors are refilled according to Listing 4.1.

Since the local state is refilled after a merge with a received delta state, some user states can be refilled even when the resulting list of updates does not contain any elements for this user. Suppose there are no updates for a user. If that is the case, this user's state in the delta state would not be refilled as opposed to the local state. For this reason, refilling is performed for all the users that were found in the initial received delta state instead of after each merged user state in the delta states. This is shown in Listing 4.4.

```
1 after merged received_delta_state:
2     for delta_state in delta_states:
3         for user, _ in received_delta_state:
4             delta_state[user].refill()
```

Listing 4.4: Pseudo-code for refilling the user states in the delta states after merging a received delta state

4.2.2 Motivating design choices

In Section 3.3, the question of how to design a relaxed distributed delta-synchronized queue for timestamps of recently accepted requests was asked. As previously mentioned, the timestamps of accepted requests are used for frequency analysis to spread urgent data, and the delta synchronization was the model of choice used to strive for

consistent states among replicas. Combining these two staples in the design gives rise to a few issues when nodes synchronize, impacting how well replicas converge and, in turn, the correctness of the token-bucket behavior. The following section describes how the design causes these issues, as well as how the algorithms surrounding the queue are designed to deal with the issues and thus maintain token-bucket behavior. The importance of an element's time-of-death is also explained.

As explained in Section 2.2, token bucket algorithms are typically used for limiting some type of traffic and enforcing a maximum rate at which traffic can be served. On a singular instance of the queue, this is enforced by simply comparing the size of a user's queue with the limit. Requests are rejected if the size of the queue is equal to or greater than the limit. The act of approving and rejecting requests is, therefore, quite simple, and a single instance of the queue would easily maintain token bucket behavior. However, the most complicated part of the design was attempting to treat requests the same regardless of which node handles a request, enabled by maintaining consistent states across replicas.

First of all, there has to be a way to distinguish one request from another. If not, the following problem may arise. Two instances of the same element could be counted several times and recognized as different elements, effectively resulting in one request consuming multiple tokens in a bucket. In the proposed design, there are two scenarios where duplicate elements end up in a replica.

The first scenario is during synchronization between three or more nodes. Consider Figure 4.4 with the three nodes A, B, and C. Node A has information about some user and decides to exchange it with node B. After this information is merged with node B's state, node B exchanges the information with node C, which also merges the information to its local state. If node C has yet to receive information about the same user from node A, node C may send this information to A from which the data originated. In this case, if identical elements are not treated as the same element, the same request will end up occupying two slots in the queue, representing two consumed tokens. This deviates from token-bucket behavior as one request should only be able to consume one token.

The second scenario is, although unlikely, when two or more nodes approve requests from the same user simultaneously. The timestamps in the implementation are represented by UNIX time in milliseconds. Since two or more nodes can approve a request on the same millisecond, the created elements will have identical time-of-birth. Thus, approving two requests at two different nodes at the same time effectively lets a user have two or more requests served for the price of one after the nodes merge states.

The proposed solution was designed as a middle ground between these scenarios. From the description of the merge function in Listing 4.2 and how the described process of accepting new requests, the following can be gathered. Elements that are originally approved at a node and return to the same node according to the

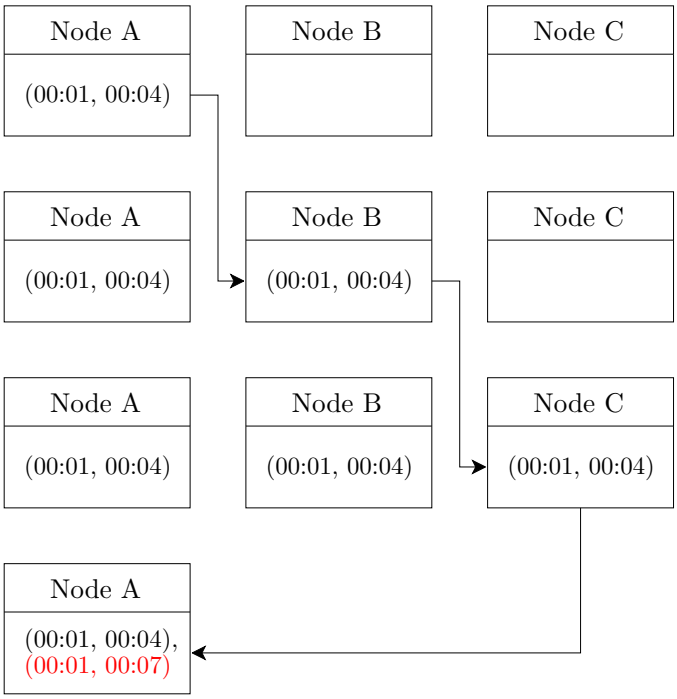


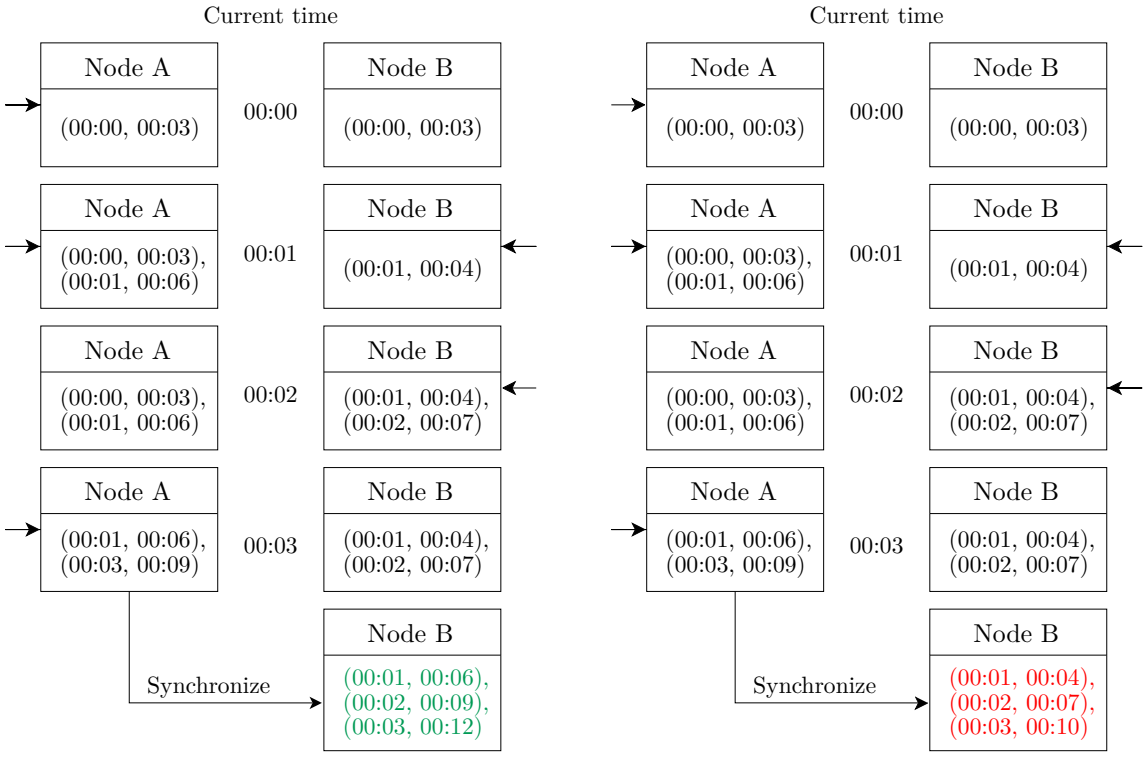
Figure 4.4: An illustration of the same request being accounted for twice when two identical time-of-birth are not assumed to stem from the same request

first scenario will be compared to an identical element and thus not added to the node's local state. However, the proposed solution does not account for the second scenario.

These two scenarios could be avoided altogether by employing some mechanism for Globally Unique Identifiers (GUIDs). A comparison of two identical elements could then guarantee that the elements are copies of each other. While this is a good solution in theory, it was left out of the implementation for the following two reasons. Firstly, it was suspected that this design would be heavier than the design using counters in terms of memory usage. Secondly, as it is already unlikely to begin with that the second scenario occurs, it was expected to have a negligible impact on the precision.

As explained in the previous section, merging a node's local state with a delta state involves updating the time-of-death for elements that follow the first merged element from the delta state. There exists a special case that is important to deal with to uphold token-bucket behavior due to the continuous removal of elements. Recall Equation (4.2), where an element's time-of-death is calculated based on the time-of-death of the element prior to this element in the queue. Depending on the synchronization protocol and how frequently nodes synchronize, an element can be removed from the local state and the delta states of the node that the element originated from prior to being sent to every other node. Whenever this happens, elements must not have their time-of-death recalculated to a lower value when merging states with another node. Avoiding lowering the time-of-death is

vital, as this essentially means disregarding the history of requests that collectively has resulted in the element's current time-of-death.



(a) with max **(b)** without max

Figure 4.5: An illustration of a scenario where max of an element's current time-of-death and a calculated value based on the previous element's time-of-death makes a difference

To understand how this can happen, consider Figure 4.5, where arrows represent inbound requests. In (a), node A adds a number of elements, one of which is removed prior to the synchronization with node B due to moving past its time-of-death. The element removed from node A had an impact on its following elements, as can be observed from timestamp 00:01 and onward. Both nodes approved a request on timestamp 00:01, and the elements' time-of-death differs as node B did not approve of a request prior to this one. When node A synchronizes with node B, they each have an element with time-of-birth equal to 00:01. In order to account for the possibility of removed elements, the element whose time-of-death is greatest is chosen according to Listing 4.3. If this was not the case, example (b) shows that the first element from node A is disregarded. Disregarding this element means that a request would be approved but never being counted toward the user's limit.

Another reason for using the function in Listing 4.3 when merging two queues is because elements are only removed from the queue whenever a new request is approved or after a merge between a local state and a delta state. The result is that for as long as no new elements are either added or merged into the local state, elements may reside in the queue past their time-of-death. Consider the merge between a local state consisting solely of elements that have lived past their time-of-death, and a

received delta state consisting of elements whose time-of-birth and time-of-death are greater than those of the local state. Elements from the local state will be merged into the new queue first, as each element in the local state has a time-of-birth lesser than those of the delta state. Suppose the function from Listing 4.3 was not included in the merging function. The elements would then be assigned new values for their time-of-death following Equation (4.2), which could then lower the time-of-death for these elements and thus deviate from the token-bucket behavior.

4.2.3 Frequency analysis

Frequency analysis of a user's recent history is performed whenever the user has a request accepted by a node. As described before, the node adds an element for this request to the user's part of the delta states of the node's neighbors. Meanwhile, it also computes the average frequency of the last few elements from the user in each delta state. How many elements that are included in this computation is specified by the *window size* parameter. The delta states where the user has fewer elements than the window size are skipped. For the remaining delta states, the average frequency of the elements in the window is calculated. If the average frequency is higher than the so-called *frequency threshold*, all elements in the respective delta state are propagated right away to the designated neighbor. This procedure is shown in Listing 4.5.

```
on accepted request from user:
    for node, delta_state in delta_states:
        user_delta_state = delta_state[user]
        user_delta_state.insert_first(request.to_element())

        window_size = config.window_size
        if length(user_delta_state) < window_size:
            continue

        earliest = user_delta_state[window_size].time_of_birth
        latest = request.timestamp
        if (latest - earliest) / window_size > config.freq_threshold:
            send(user_delta_state, node)
```

Listing 4.5: Pseudo-code for analyzing the frequency of accepted requests after accepting a new request

This procedure is the result of two design choices. The first design choice was whether to perform the frequency analysis on the local state instead of the delta states. The former would require only one comparison of a user's recent history compared to one per neighbor. However, doing so would mean that the node might send propagation messages to its neighbors containing only one element if the node accepts yet another frequent request. This approach was decided against to avoid sending too many messages on the network. The chosen approach instead limits each propagation message sent due to frequency analysis to the value of the window size parameter.

The other design choice was if frequency analysis should also be performed when

receiving a delta state from a neighbor sent through normal gossiping. Doing so would make sure that important information would reach all neighbors faster than the chosen approach. However, to avoid flooding the network with messages, it was decided only to perform the analysis when accepting a new user request.

4.2.4 LINF-gossiping

What is referred to as Least Informed Neighbor First (LINF) is enabled by using the queue. It is accomplished by every node keeping a map of its neighbors' node IDs and a respective value representing the number of updates queued for that neighbor.

This map is updated whenever a request from a user is accepted. As shown in Listing 4.6, the order of the priorities is unchanged since the value for every neighbor is incremented by one.

```
1 on accepted request:
2     for node in neighbors:
3         node_gossip_priorities[node] += 1
```

Listing 4.6: Pseudo-code for changes to the neighbor priorities map when accepting a request

Another case where the map is updated is whenever a delta state is received from a neighbor. The handling of this case is shown in Listing 4.7. The received delta state is first merged with the node's local state, yielding the updated elements. After that, the priority value of every neighbor aside from the sender is incremented by the difference of the number of elements in each respective delta state before and after merging with the list of updates. This procedure is seen on lines 4–7. For the sender, its value is instead recalculated by looking at how many elements are queued in its corresponding delta state, which is shown on line 12.

```
1  on received delta_state from sender:
2      local_state, updates, _ = merge(local_state, delta_state)
3
4      for node not sender in neighbors:
5          delta_states[node], _, additions = merge(delta_states[node],
6                                                   updates)
7          node_gossip_priorities[node] += additions
8
9      delta_states[sender] = remove_duplicates(merge(delta_states[sender],
10                                                    updates),
11                                             delta_state)
12     node_gossip_priorities[sender] = delta_states[sender].sum_elements()
```

Listing 4.7: Pseudo-code for changes to the neighbor priorities map when receiving a delta state

The final case where the map is updated is when it is time to synchronize with another node. Which node to synchronize with is decided by iterating through the map to find the node with the highest priority value. If multiple neighbors have

equal priorities, the decision between them is made by randomly selecting one. The selected node's entry in the priority map is then set to zero.

4.2.5 Frequency analysis and LINF-gossiping combined

Propagating, and thus clearing, a part of a neighbor's delta state should update the neighbor's priority of which node to gossip to next. For this reason, the priorities for the neighbors that will receive the part of their delta states corresponding to one user will be decremented by the number of elements that were sent. Listing 4.8 depicts the handling of this case.

```
1 after sent user_delta_state to neighbor:
2     node_gossip_priorities[neighbor] -= length(user_delta_state)
```

Listing 4.8: Pseudo-code for changes to the neighbor priorities map when sending a delta state of one user with frequent requests

The receiving node will process the received user delta state in the same way as shown in Listing 4.7, but with the difference being that only the single user is present in the delta state.

4.3 Test setup

In this section, the framework used for evaluating the token-bucket queue is described. The equipment and the parameters used and the data sets that constitute the different tests are also explained.

4.3.1 Test framework

To evaluate a solution that would allow the system to decrease the number of incorrectly accepted requests from a global perspective, two implementation approaches were identified.

The first option was to modify Spotify's system with the proposed changes. Compared to building a project from scratch, this approach would have given much for free since it contains an implementation of the token-bucket counter design. However, the current system was built to be hosted on cloud services, making the process of debugging, testing, and evaluating more complex compared to a solution that runs locally on the same machine.

The other option was to build a simulation project from the ground instead. To simulate a rate-limiting system that consists of multiple nodes, Erlang was a suitable choice that allows cheap spawning of processes to act as the nodes on the same local machine. To compare the proposed queue design to the token-bucket counter design, this approach would have required implementations for both the proposed solution and the token-bucket counter design. However, simulating a physically distributed rate-limiting system would have enabled the evaluation to assume no node crashes or

packet losses. In turn, this assumption would have made it easier to evaluate whether using the queue is a feasible solution to the problem described in Section 3.3.

In favor of easier testability, it was decided to use the second option, i.e., building a new project that aimed to simulate the system as described in Section 3.1 and the proposed changes. Still, it was desired to use real data to evaluate the two different designs in the simulation project. Data recorded on Spotify's system could be exported on the following form:

```
[
  {
    "user_id": "deadbeefdeadbeefdeadbeefdeadbeef",
    "time": <13:47:02.382>
  },
  ...
]
```

The value that `user_id` maps to is a 32 character string of hexadecimal numbers, and the corresponding value for `time` is a UNIX timestamp in milliseconds. Processing such data required an Erlang module to be built that could read a file with recorded requests and inject them into the simulation system. A test would then be made up of a process, hereinafter referred to *the injector*, running the code of the module, simulating the different users sending requests.

To achieve a realistic distribution of the requests among the nodes in the system, the injector utilizes consistent hashing[2] to assign requests to the different nodes. The node responsible for a request is determined by the hash of the request's recorded time multiplied by the user ID. The combination of both fields was used to disperse the requests as much as possible. Simply using the user ID would yield all requests for any user to be received by the same node. Moreover, only using the time field would result in all requests within the same timestamp being received by the same node, possibly inducing an uneven load in the system. However, the chosen combination means that all requests recorded at the same millisecond for the same user are assigned to the same node.

The injector performs the task of injecting requests by first looking at the initial entry in the data holding all the recorded requests. The node responsible for the request is calculated and then synchronously contacted with a message containing the user ID. Upon receiving a reply from the node with information of whether the request was accepted or not, the injector stores this information before moving on to the next entry.

One key element in a set of recorded requests is the timings between them. In an attempt to recreate the same timings, the injector compares the next entry's time

[2]In consistent hashing, some part of the input data is hashed. That output is then mapped to the various available options.

to the time of the entry that was just processed. The injector uses the difference in time between them to determine how long it should sleep before processing the subsequent request. Since sleeping is handled by an external scheduler that manages multiple processes on a shared processing core, it is not guaranteed for a process to be woken up at the exact desired time. Although it is guaranteed that a process will sleep for at least until the desired time, in some cases, a sleeping process is woken up slightly later than desired. For this reason, the injector stores how long it desires to sleep along with a timestamp of the time just before sleeping. After waking up, this allows the injector to become aware of any additional undesired slumber duration and adjust the upcoming sleeping periods by this number. To account for cases where multiple requests were recorded within the same millisecond, meaning no sleeping in between them, this value is stored and used to adjust the next sleeping period.

Since the call with the request to the node is made synchronously, the injector can be delayed if the two solutions require different amounts of time when processing a request. For this reason, the injector also measures the duration between when the call to the node is sent and when the corresponding reply is received. This duration is accumulated and used to offset the following sleeping periods combined with the value that keeps track of any prolonged sleep.

The above-described procedure of the injector is then continued until the last request for a specific test has been processed. When reaching that point, the injector waits until the nodes have converged. It does so by periodically querying the delta states and message input queues of all nodes. When all delta states and message input queues are empty, no more updates are queued anywhere in the network. The test is then considered to be completed, and all processes are thus terminated.

Since the injector keeps track of how many requests were rejected, the two solutions can be compared in terms of this metric. In order to see how far from an ideal result the solutions lie, their results can also be compared to that of a centralized system. By executing the same test in such an environment, i.e., one where a single node has access to all data for any user, a ground truth of the number of requests that should be rejected will be retrieved.

To evaluate the solutions with regards to the memory and bandwidth metric, another module that periodically monitors the node processes in these areas was built as well.

Finally, to evaluate the convergence time, the memory and bandwidth data can be used to see when the test system has converged. Since the gathering of such data ceases when the test has been completed, it reveals how long it took for the system to converge.

4.3.2 Test equipment

The hardware used when executing the tests was a 6-core Intel Core i9 2.9 GHz CPU with 16 GB 2400 MHz DDR4 memory. The operating system was macOS 10.15.7, and the Erlang OTP version was 24 RC2.

4.3.3 Default parameters

When evaluating the proposed design, multiple variables influence the outcome of the tests. Unless otherwise stated in the description of a test, a set of default values is used. The ones that apply to both token bucket designs can be found in Table 4.1. The capacity parameter is among these and has the same value as in Spotify's system. For confidentiality reasons, it is expressed as c and other values are described in terms of this variable. Parameters specific to the queue design can instead be found in Table 4.2.

Parameter	**Value**
Network topology	Fully connected
Number of nodes	30
Gossip interval	300 ms
Capacity	c
Refill-rate	$0.001c$ tokens per second

Table 4.1: General default parameter values

Parameter	**Value**
Window size	$\frac{c}{\texttt{Number of nodes}}$
Frequency threshold	2400 ms

Table 4.2: Default parameter values specific to the queue

The value of the window size parameter was decided to be a node's share of the capacity as if it would have been divided to the different nodes. This value was chosen because if all nodes fill the bucket for a user to that point, the total limit across the entire system has been filled. If that is the case, the nodes should propagate this information to their neighbors to prevent overfilling. If the frequency between these requests is deemed high, it is desired to propagate this information right away instead of relying on the normal gossip interval.

The frequency threshold value was set based on the average frequency per user within a sliding window of size equal to the window size parameter. The data used when calculating this was requests from 10 000 users that were not rate-limited but sent the most requests during the chosen data duration of one hour. The reason for using data from users that were not rate-limited but generated the most data was because such data was deemed to be representative of data where our solution could improve rate limiting. To avoid the cases where users had frequent traffic in chunks with long periods without any requests sent in between, window frequencies below

a certain threshold were discarded. This threshold was set to five seconds. The average frequencies of all users were then averaged before rounding the outcome to the closest hundred ms.

4.3.4 Test data

Multiple data sets were used to evaluate the metrics. Down below are descriptions of them and motivations why they were chosen.

Data set A

First and foremost, data from users that were rate-limited by Spotify's system was extracted. The idea behind using data from such users is that the token-bucket counter implementation should behave closely to that of Spotify's system. It was also desirable for the implemented counter and queue token-bucket designs to yield the same number of rejections. To verify these two cases, the total number of rejected requests for the different systems was measured on a data set spanning over five minutes from 10 randomly selected users that had been rate-limited at least once during the same period. In total, this data set contained 68.3c requests. The number of requests per user varies, as is shown in Figure 4.6.

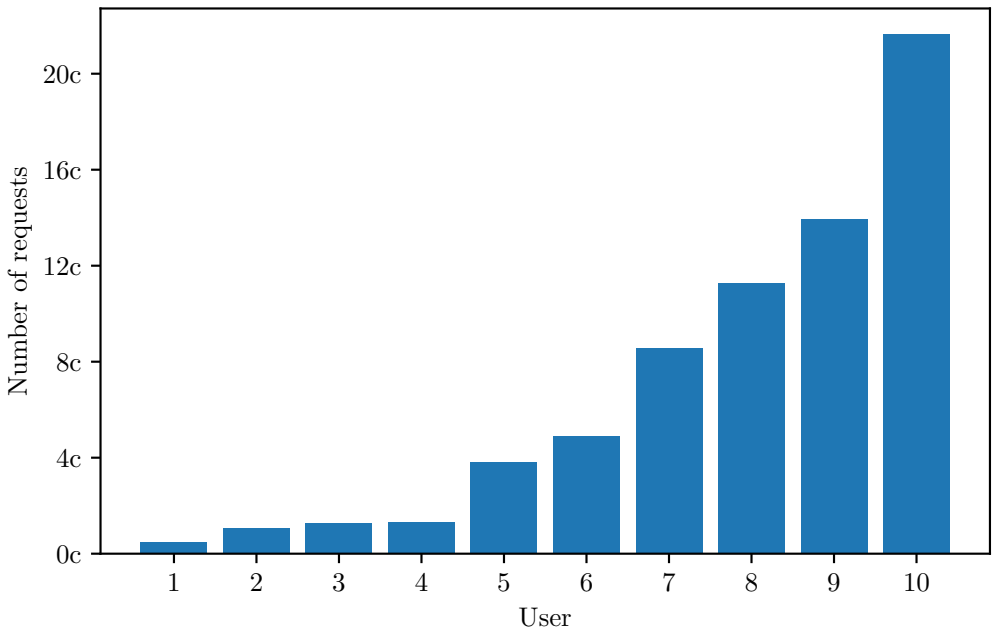


Figure 4.6: The number of requests distributed among the 10 users in data set A

This data set was also used to see how much the solution for each respective research question impacted the outcome individually regarding the number of rejections and memory and bandwidth consumption.

Data set B

To evaluate how much quicker the system would react with the proposed changes to a substantial number of requests sent within a short time from the same user, three smaller data sets were used. These three data sets each contained requests from a single user over a shorter period. The period was set to be one minute. The idea of using three data sets was to see whether the proposed changes performed better or worse with different rates of requests. As such, the users were hand-picked based on their amount of requests sent within this period, where it was desired for the selected users to have a substantial difference in their rate of sending requests.

The lower bound of the number of requests was set to the capacity parameter in addition to the number of refilled tokens during the one-minute duration. For this reason, the first of the three chosen users was the one with its number of sent requests closest to – yet above – this number. The second user was the one with the largest number of requests within the period, whereas the third was the user with the number of requests closest to the value in between the first two users. The number of requests in each subset resulted in $1.1c$, $21.5c$, and $3.2c$, respectively. These three subsets will be referred to as *extreme*, *substantial* and *barely rate-limited* in descending order of their number of requests.

With the three subsets mentioned above, the two solutions were to be evaluated for users who, on average, have a request rate higher than the refill rate. However, the vast majority of users in the real system are users that are not rate-limited. It was thus natural to evaluate the two solutions in this regard as well. For this reason, a fourth subset was used where the request rate was lower than the refill rate. This subset could have been extracted from the existing system. However, doing so would include a risk of getting data where the request rate was temporarily higher than the refill rate, thus leading to ambiguity in the results. For this reason, this subset was fabricated, where all requests were given timestamps on a fixed interval slightly faster than the refill rate. This fourth subset is referred to as *not rate-limited* and spans over two minutes.

Data set C

While data set B evaluates the two solutions when handling traffic from users who should and should not be rate-limited, the solutions were evaluated with one user at a time. It was thus natural to evaluate the two solutions regarding the memory and bandwidth metric on a data set that matches the distribution of traffic that Spotify's system is exposed to.

For this reason, a larger data set that spanned over a longer period and that contained data from many randomly selected users was used. However, since the nodes in the simulation system are represented as processes on the same machine, the number of users in the data set was limited to 5000 in order to run the test with limited memory and processing resources. Although this number is much lower than the number of users Spotify's system handles, the relation between the two solutions

should remain the same despite the difference in the number of users.

This data set spanned over one hour. The 5000 users were selected to match the same distribution of rate-limited users and non-rate-limited users Spotify's system handled in the same hour.

Data set D

The queue solution works by dropping old elements. To see if this could have a negative impact where users could circumvent the desired limit given that the system configuration was known, a fourth data set was fabricated. This data set contained requests from a single user on a fixed interval slightly more frequently than the refill rate. The larger capacity and refill rate, the longer duration would be required for the user to exceed the limit in a centralized system. The test duration was set to 100 seconds. Some of the parameters were therefore changed, and their new values are presented in Table 4.3. The request rate was set to one token every 0.9 seconds, meaning the user would, in the long run, send a request 100 ms earlier than allowed by a centralized system. Three different gossip intervals were used, resulting in one, two, or three synchronization messages in-between any two adjacent requests.

Parameter	**Value**
Gossip interval	300 ms, 450 ms, 900 ms
Capacity	5
Refill-rate	1 token per second

Table 4.3: Parameter values when testing using data set D

5

Evaluation

In this chapter, results from the tests with the four data sets A, B, C, and D are presented regarding the metrics described in Section 3.5. A discussion of these results is also included.

5.1 Data set A

This section presents the results from tests using data set A. Section 5.1.1 includes a comparison of the implemented token-bucket counter and Spotify's system. A comparison of test runs with centralized systems running the token-bucket counter and the token-bucket queue implementations is included in Section 5.1.2. Section 5.1.3 evaluates the impact of the queue's two extensions.

5.1.1 The baseline compared to Spotify's system

In Figure 5.1, the data set with 10 users that were rate-limited at least once during a five-minute duration was used. Results averaged over 10 runs from the simulation system running the implemented token-bucket counter were compared against the recorded number of rejected requests from Spotify's system. For this test, the parameter values were assigned to match the ones Spotify's system is using, i.e., not the ones presented in Table 4.1. In addition to the comparison between the two systems, the graph also displays the total number of requests received per time unit and the results of a run with a centralized setup, i.e., one that always will take the correct decision. The closer a solution's result lies to the result of the centralized run, the greater precision the solution achieved.

Figure 5.1 shows that both distributed systems achieved similar results, even though the simulation system performed slightly more accurately. To see how close the lines of the two solutions lie to the centralized line, all points for each respective solution were summed and then divided by the same value of the centralized run. This calculation yielded that the baseline achieved 95.7 % of the number of rejections of the centralized version, whereas Spotify's system achieved 91.2 % in the same metric. These numbers show that there is room for improvement.

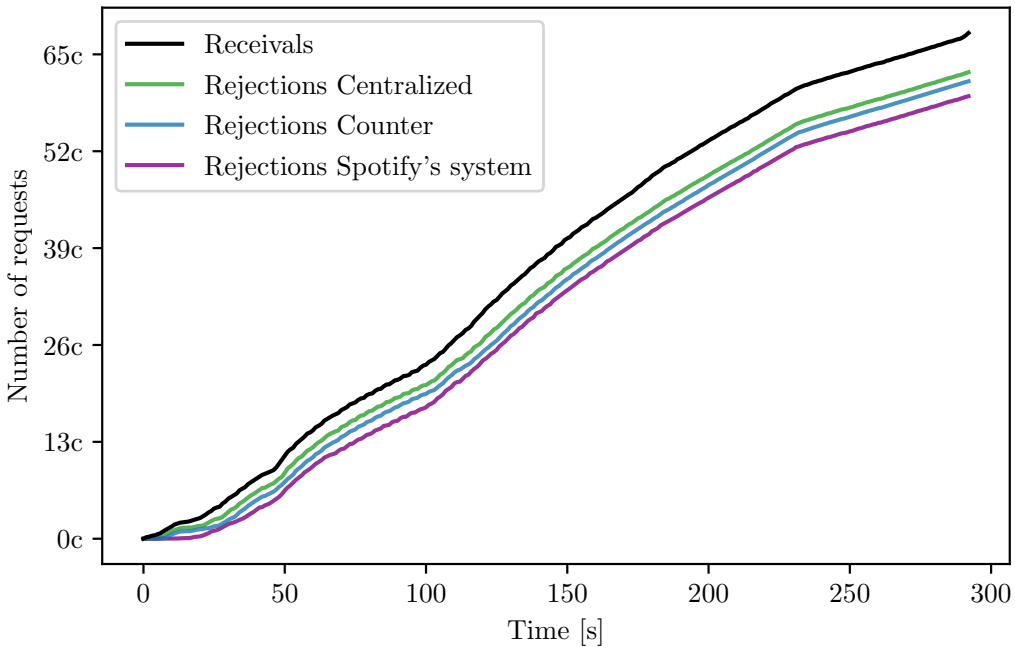


Figure 5.1: The implemented token-bucket counter versus Spotify's system

5.1.2 Centralized counter compared to centralized queue

In the next test, the same data set was used. As opposed to the previous test, the default parameters were instead used. Figure 5.2 shows of the number of rejects of the token-bucket counter and the token-bucket queue in a centralized system. The graph shows that the two implementations yielded nearly identical results. With this information, it can be concluded that the designed and implemented queue fulfills the desired token-bucket behavior when used in a centralized system.

5.1.3 Evaluating the two extensions of the queue

Recall research questions **Q2**, **Q3**, and **Q4**. To evaluate these questions, combinations of the queue with and without the two extensions are tested using data set A.

In Figure 5.3, the difference in the number of rejections for each solution compared to the centralized system over time is shown. The ideal solution would always have a difference equal to zero, meaning a low value is better than a high value. The results are averaged over 10 iterations.

It is shown that the counter, the queue with LINF only, and the plain queue reacted much slower than the queue using frequency analysis but that their differences to the centralized system then decreased over time. Moreover, it is shown that LINF decreases the precision when used in combination with frequency analysis. However, without frequency analysis, LINF achieved roughly the same precision as the plain queue.

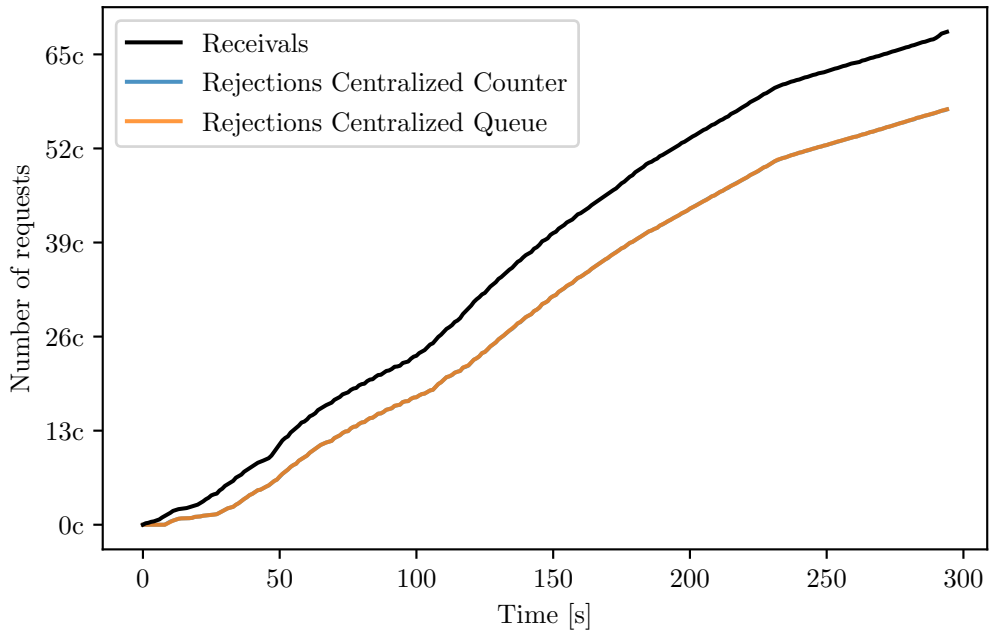


Figure 5.2: Centralized token-bucket counter versus centralized token-bucket queue

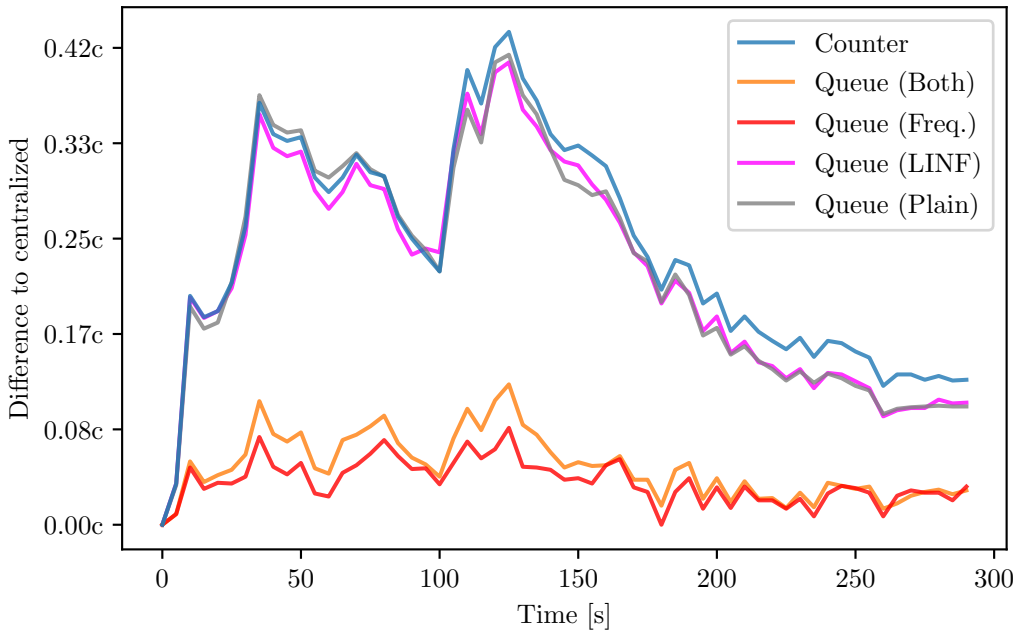


Figure 5.3: The difference of the number of rejected requests to the centralized system averaged over 10 runs

The memory and bandwidth consumption for the same test is displayed in Figure 5.4. In this and every other figure that displays the memory and bandwidth consumption, the highest value for the counter solution is assigned 100 % on the y axis.

By assessing Figure 5.4, the system using the counter implementation is seen to converge at the five-minute mark, whereas the systems using the queue implementations require at least another minute to converge.

LINF was shown to decrease the memory usage both with and without frequency analysis. Compared to the plain queue and the queue used with frequency analysis, LINF decreased the memory usage by 18.3 % and 24 % respectively.

As for the bandwidth consumption, LINF yielded in similar usage both when used in conjunction with frequency analysis and without. Another thing to note is that the difference in bandwidth consumption between the counter and the queue used with only LINF was relatively small compared to that between the counter and the queue used with frequency analysis. The queue with LINF used 31 % more bandwidth than the counter, whereas the queue with frequency analysis used 243 % more than the counter.

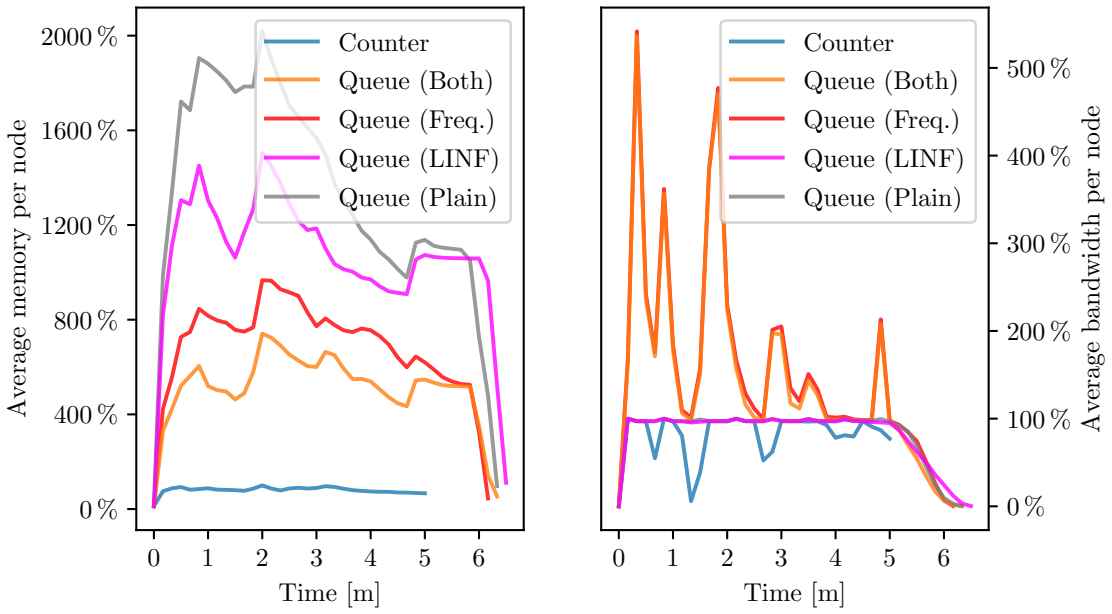


Figure 5.4: A comparison of memory and bandwidth usage of the queue with all combinations of extensions and the counter. Results are averaged over 10 runs.

In Section 3.2, it was hypothesized that a solution that enables the system to analyze the frequencies of users' requests to synchronize more eagerly would achieve higher precision than the baseline. It was also hypothesized that this solution would cause bursts in the usage of bandwidth. By assessing Figure 5.4 and Figure 5.3, it can be concluded that both hypotheses are true. In the same section, it was also hypothesized that prioritizing synchronization with the neighbor that is considered most behind in updates would improve the precision of the queue. This result could not be observed in Figure 5.3. However, Figure 5.4 shows an unexpected side-effect of substantially reduced memory usage.

5.2 Data set B

In this section, the results of the four subsets in data set B are presented – namely, the data sets each containing requests from a single user with different request rates. All results presented are averaged over 10 iterations of the same test. A combined discussion of all the results is presented in Section 5.2.5.

5.2.1 Extreme

In Figure 5.5, the average number of rejects for both the counter and queue implementations are shown from tests with 21.5c requests from a single user. Just as before, the total number of requests received and the result of a centralized system are seen in the same plot.

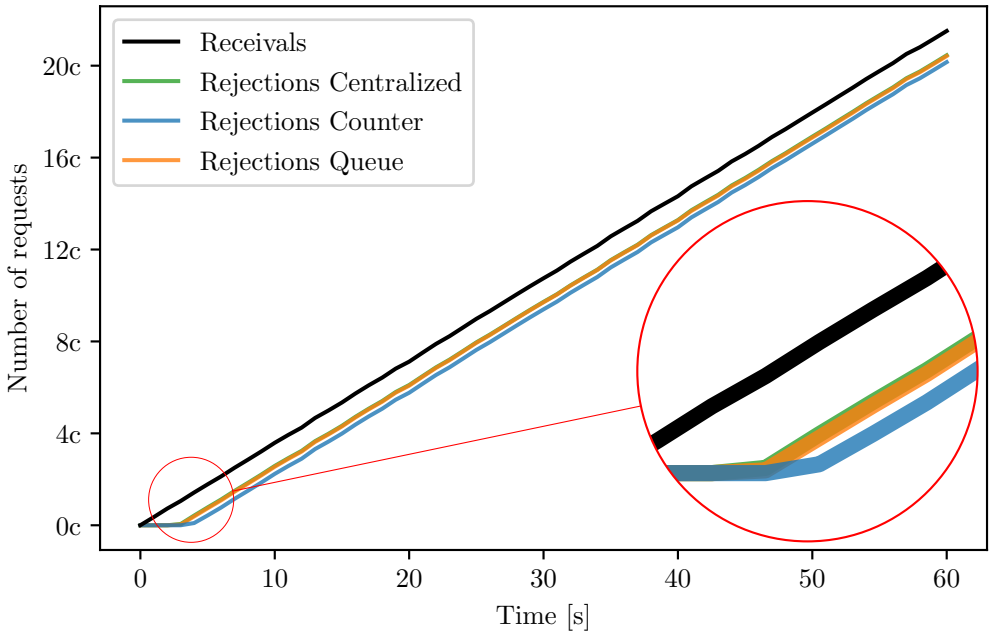


Figure 5.5: A comparison of precision between the counter and the queue for a rate-limited user generating an extreme amount of requests within one minute. Results are averaged over 10 runs.

It is difficult to see where the centralized line lies. For this reason, the part where the three systems start to rate-limit the user is zoomed in. In this view, it is shown that the queue implementation reacted quicker than the implementation using counters – the result of the former almost follows the centralized line.

In total, the counter implementation achieved 96.9 % of the result of the centralized version, where the corresponding value for the queue implementation was 99.7 %.

In Figure 5.6, the average memory and bandwidth consumption per node from the

same tests are shown. The figure reveals that the queue implementation used more memory and bandwidth than the counter implementation. The ratio between the area under the curves is 7.98 for the memory and 8.83 for the bandwidth consumption. The queue implementation thus used 798 % more memory and 883 % more bandwidth for this test.

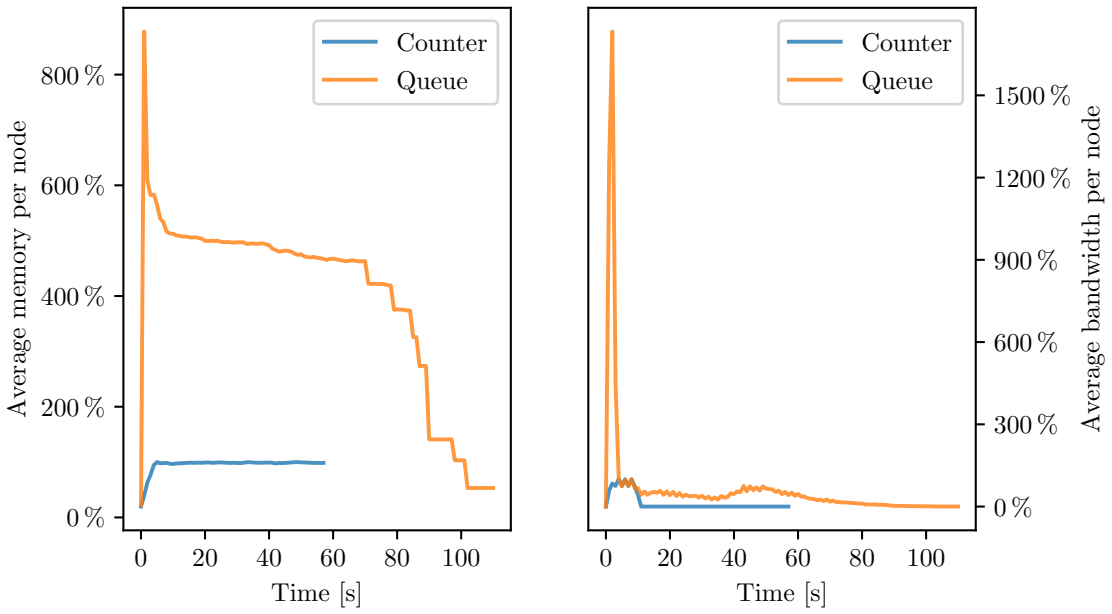


Figure 5.6: A comparison of memory and bandwidth usage between the counter and the queue for a rate-limited user generating an extreme amount of requests within one minute. Results are averaged over 10 runs.

By observing the graph that displays the bandwidth usage, it can also be seen that the system using the counter implementation did not propagate any more messages after the ten seconds mark. In contrast, the system using the queue implementation propagated messages long after no new requests were received.

5.2.2 Substantial

Figure 5.7 shows the same information as Figure 5.5, but from tests with the data containing $3.2c$ requests from a single user.

It is again shown that the queue reacted quicker than the counter, although not as quick as the centralized version. Additionally, an aspect that can be seen, which was also visible in the previous test but not as easy to spot, is that the distributed systems' lines converged to the centralized system over time.

The counter solution achieved a precision of 96.4 % of the ground truth, whereas the queue solution achieved 98.6 % in the same metric.

As for the bandwidth and memory metric evaluated in Figure 5.8, it is shown that the difference between the two solutions was lower in this test compared to the

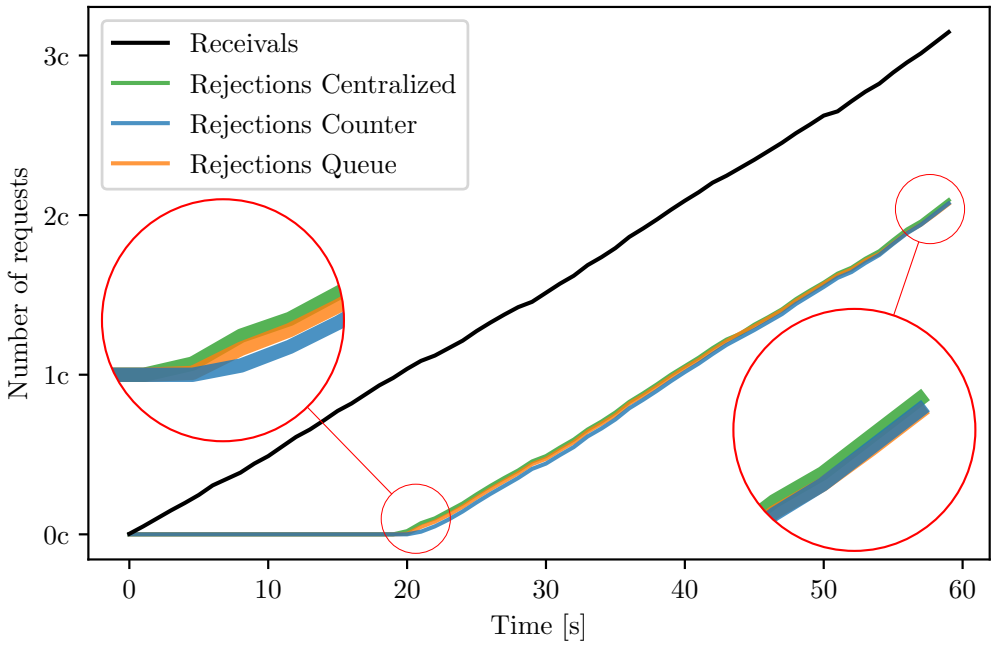


Figure 5.7: A comparison of precision between the counter and the queue for a rate-limited user generating a substantial amount of requests within one minute. Results are averaged over 10 runs.

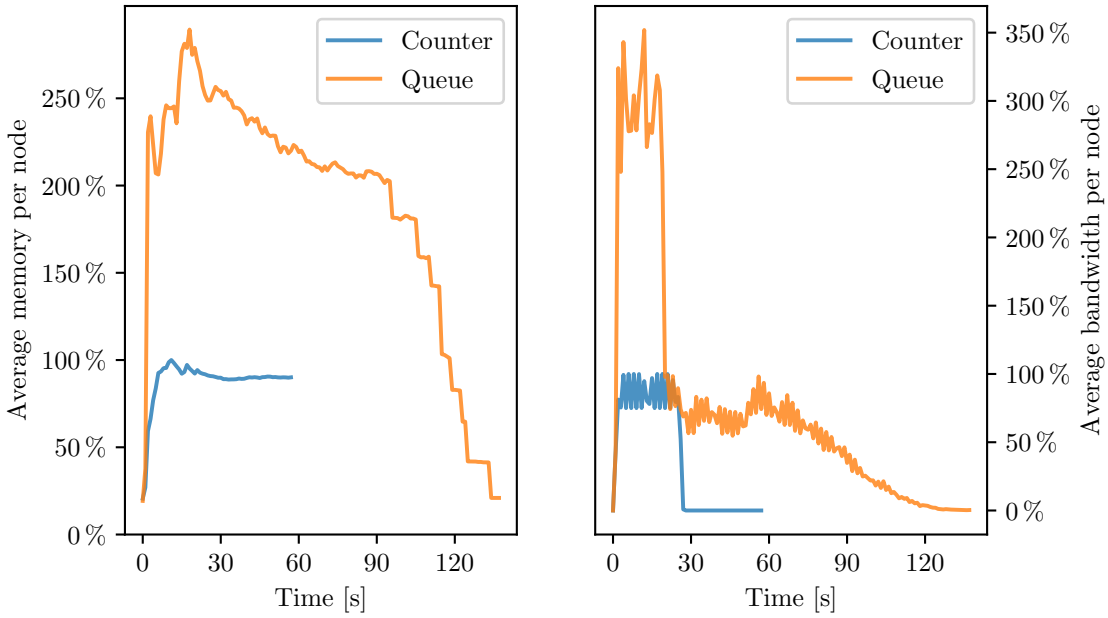


Figure 5.8: A comparison of memory and bandwidth usage between the counter and the queue for a rate-limited user generating a substantial amount of requests within one minute. Results are averaged over 10 runs.

results shown in Figure 5.6. For the results described here, the ratio between the curves yielded that the queue used 513 % more memory and 495 % more bandwidth

than the counter.

It is also apparent that the counter solution did yet again stop propagating information earlier than the last received request.

5.2.3 Barely rate-limited

Figure 5.9 displays the average precision when tests with data from a user with $1.1c$ requests were executed. Both solutions rejected requests towards the end, but the graph shows that the queue achieved higher precision than the solution using counters. The former achieved 80.0 % of the centralized system, whereas the latter achieved 10.0 %.

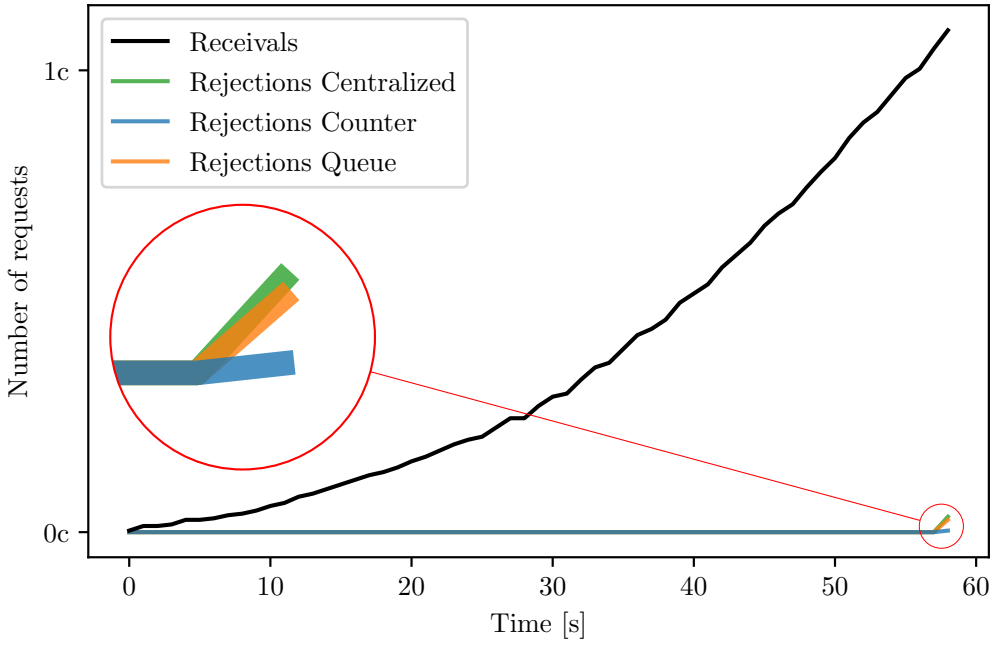


Figure 5.9: A comparison of precision between the counter and the queue for a user that is barely rate-limited within one minute. Results are averaged over 10 runs.

In Figure 5.10, the memory and bandwidth usages for the same tests are shown. These graphs follow the same pattern as the graphs from the two sections above regarding the decreasing difference in memory and bandwidth consumption between the two implementations. By calculating the ratio between the area under the curves, the queue implementation used 427 % more memory and 215 % more bandwidth for this test than the counter.

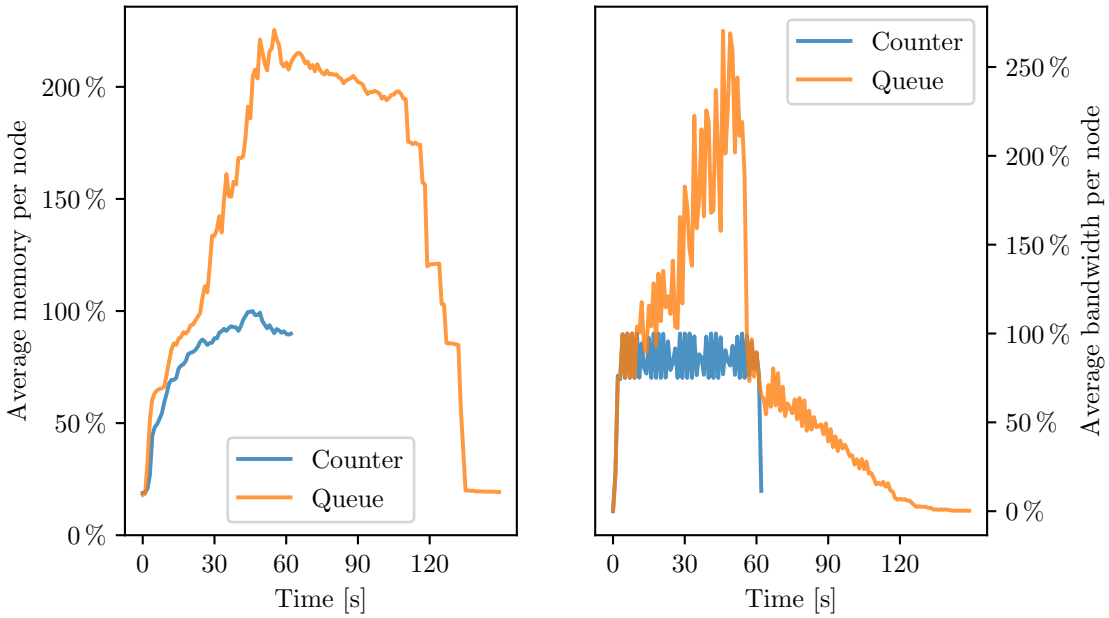


Figure 5.10: A comparison of memory and bandwidth usage between the counter and the queue for a user that is barely rate-limited within one minute. Results are averaged over 10 runs.

5.2.4 Not rate-limited

This test was executed with fabricated data representing traffic from a user that sends requests with a frequency lower than the refill rate. LINF-gossiping was disabled in this test as the map of 30 nodes would influence the memory usage disproportionately to the number of requests. At the same time, since there was only going to be a few elements queued up in the delta states at once, any benefit of using LINF-gossiping would have been diminutive.

Figure 5.11 is included for completeness. It shows that the number of requests received is evenly distributed and that no solution rejects any request.

As for the memory and bandwidth consumption, it can in Figure 5.12 be seen that the queue solution used slightly less memory and much less bandwidth than the counter implementation. Dividing the areas under the curves with each other for the respective graph yields that the queue solution used 10 % less memory and 77 % less bandwidth than the solution using counters.

Moreover, it can also be seen that the queue yet again took longer to converge than the counter, but that the period was greatly reduced from earlier tests.

Relating to the convergence time, the sudden drop in memory is due to the different iterations that took longer to complete. When that happens, the remaining values are divided by the total number of iterations, resulting in a low average. The queue's variance of convergence time can also be seen in the previous tests.

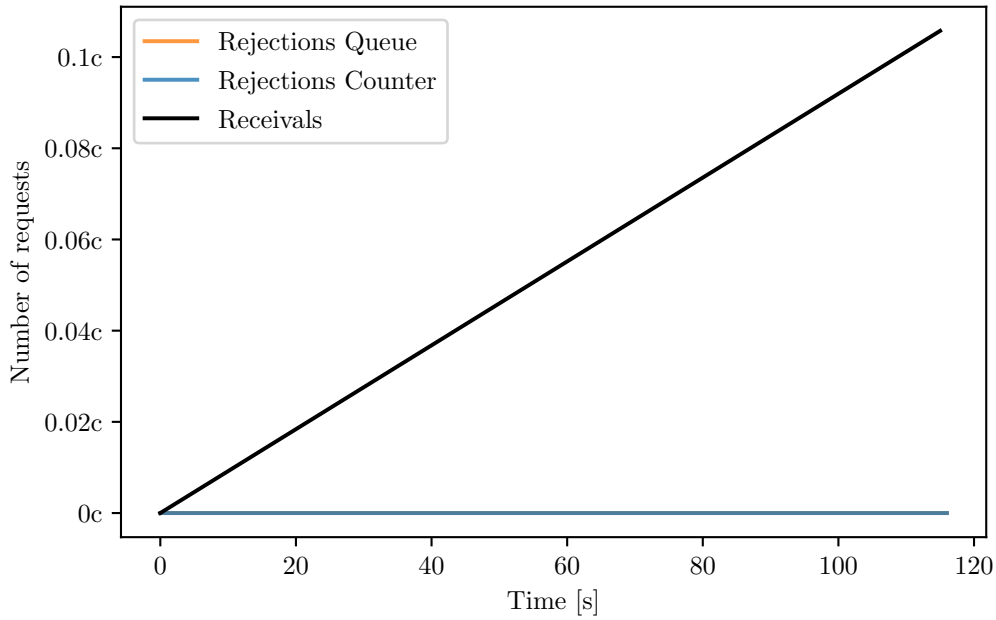


Figure 5.11: The number of requests received and rejected for a not rate-limited user generating few and evenly time-distributed requests within two minutes. Results are averaged over 10 runs.

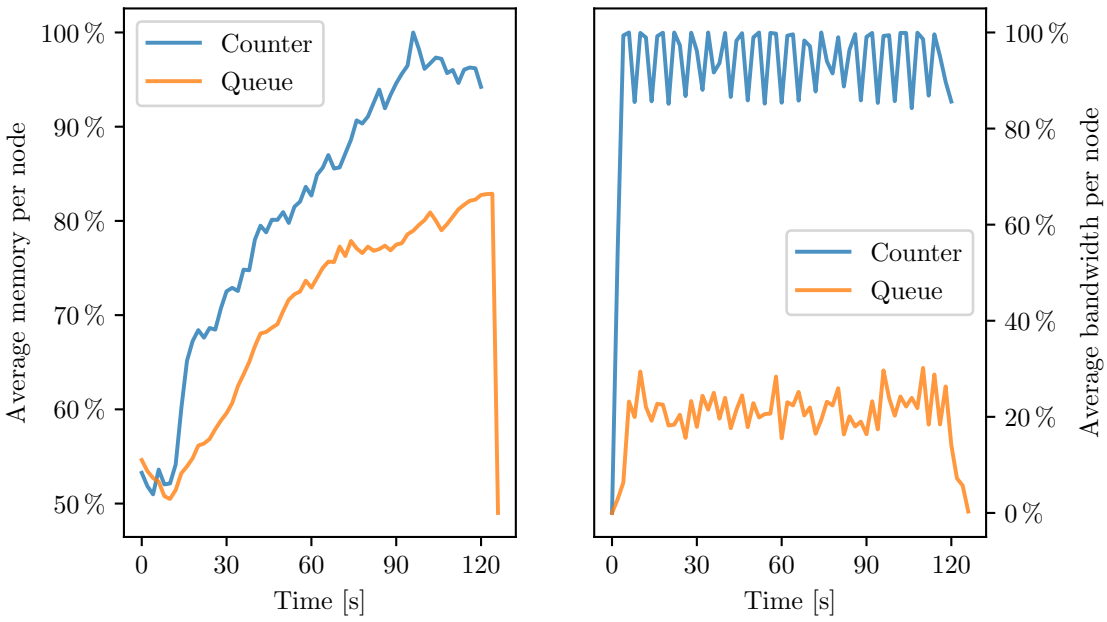


Figure 5.12: A comparison of memory and bandwidth usage between the counter and the queue for a not rate-limited user generating few and evenly time-distributed requests within one minute. Results are averaged over 10 runs.

5.2.5 Discussion

From the results shown in Section 5.2.1 and Section 5.2.2, the counter solution was seen to stop consuming bandwidth long before the last request had been received. The sudden halt of bandwidth usage was due to the counter reacting too slowly to the rapid influx of requests. When the nodes finally synchronized with each other, they discovered that the limit had been exceeded by a great margin. As a result, every following received request was rejected by any node. The nodes did thus not queue any update to their neighbors, which explains the lack of bandwidth use after this point.

The fact that the number of consumed tokens exceed the limit also explains why the counter solution cuts down the distance to the centralized system over time. It can reject all requests and thus catch up compared to the centralized system that will allow new tokens to be consumed at the refill rate. This catching-up phase means that a user that has consumed too many tokens has to wait a long time to use the service again. Although this phase allows the counter to improve its final result, it does not enforce the desired token-bucket behavior in this situation.

In the memory graphs from the tests using the data set B, the steep decreases in memory of the queue imply that it has a wide variance in how long the system takes to converge. This variance was never observed for the counter. However, that might partly be because it reacted too slow in the previous tests and thus had a debt in tokens as described above. When dealing with users that send few requests, shown in Figure 5.12, the queue took longer to converge than the counter but did not show a wide range of convergence times like it did in previous tests. This difference was because the number of requests for the same user was significantly reduced, thus minimizing the number of times the same element was queued for propagation. Despite the queue requiring longer to converge, it still rate-limited users much more similar to the centralized system as opposed to the counter.

All four memory graphs for the tests carried out with the data sets in B show that the counter's memory use over time in general increases or remains the same. It can be seen that it sometimes temporarily decreases, which is due to some delta states being cleared. However, the local state of a node will increase in size or remain the same, depending on if new users send requests, thus leading to the memory curves of the counter pointing upwards. This gradual increase in memory usage does not apply to the queue to the same extent. Before the steep decreases in memory due to different convergence times, the memory consumption decreases steadily the more time that passes. This decrease is because old elements are cleared as new requests or synchronization messages are received.

5.3 Data set C

Figure 5.13 shows the memory and bandwidth usages for the test executed with the data set containing traffic from 5000 users over a duration of one hour.

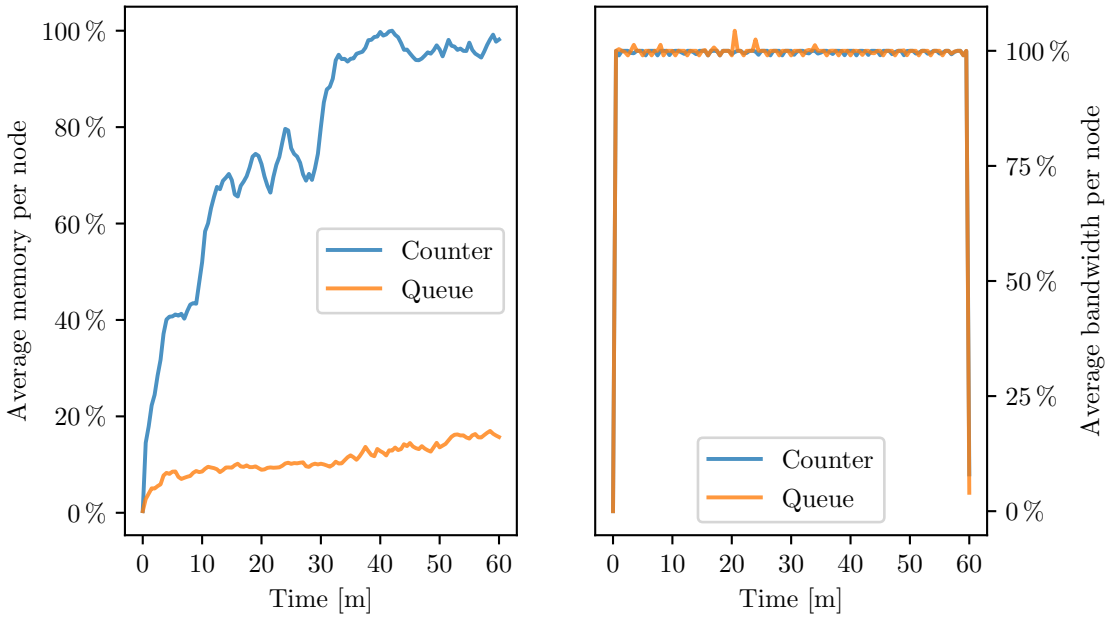


Figure 5.13: A comparison of memory and bandwidth usage between the counter and the queue for requests from 5000 users within one hour

The queue solution used 86 % less memory than the solution using counters, whereas the bandwidth usage for the queue was 0.0025 % more than the counter's.

Figure 5.13 confirms that the counter increases its memory consumption substantially more for every user than the queue. Still, the queue's memory usage goes upwards, albeit not as steep as the counter's. However, this is due to new users constantly allocating space, which can be concluded by assessing Figure 5.14.

5.4 Data set D

In Figure 5.15, the number of rejections for the queue solution when used with different gossip intervals is shown. In this test, the requests are from a single user and received on a frequency slightly higher than the refill rate. It is shown that the longer the intervals are between gossip, the more delay in the response of rejections if there is any response at all.

For the queue solution to even start rate limiting a user with such a request rate, multiple adjacent elements must collide on the same node, meaning information about the first request must be on the node that receives the second, and so on. The opposite would be problematic as previous elements would be removed from the system before the new elements are affected by them, possibly resulting in users exceeding the desired limit without ever being noticed.

As the proposed solution does not guarantee that an element corresponding to a request will be sent to all nodes before the element is considered dead, this becomes

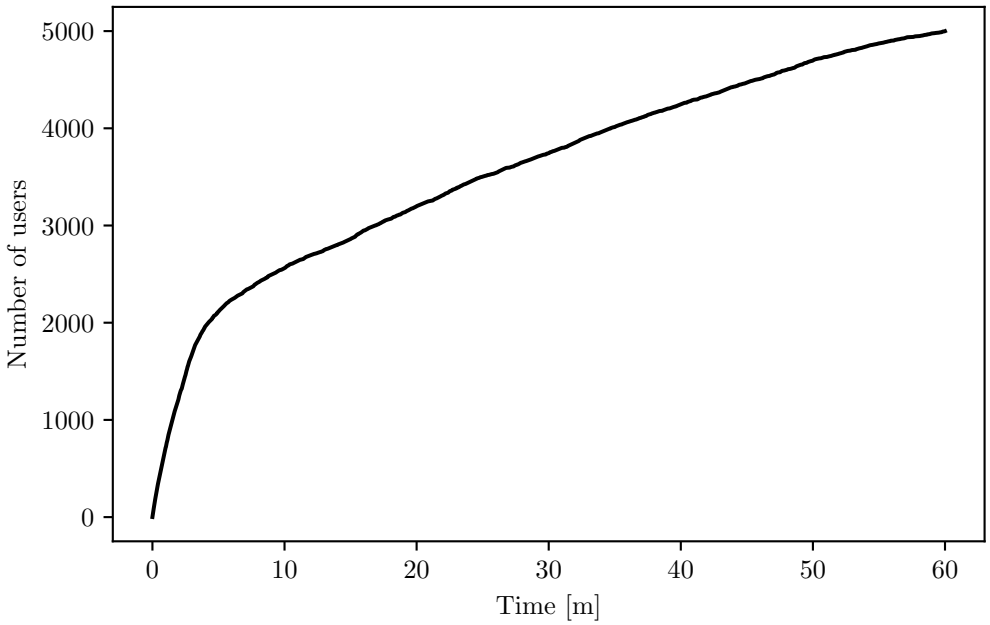


Figure 5.14: The number of users over time in data set C

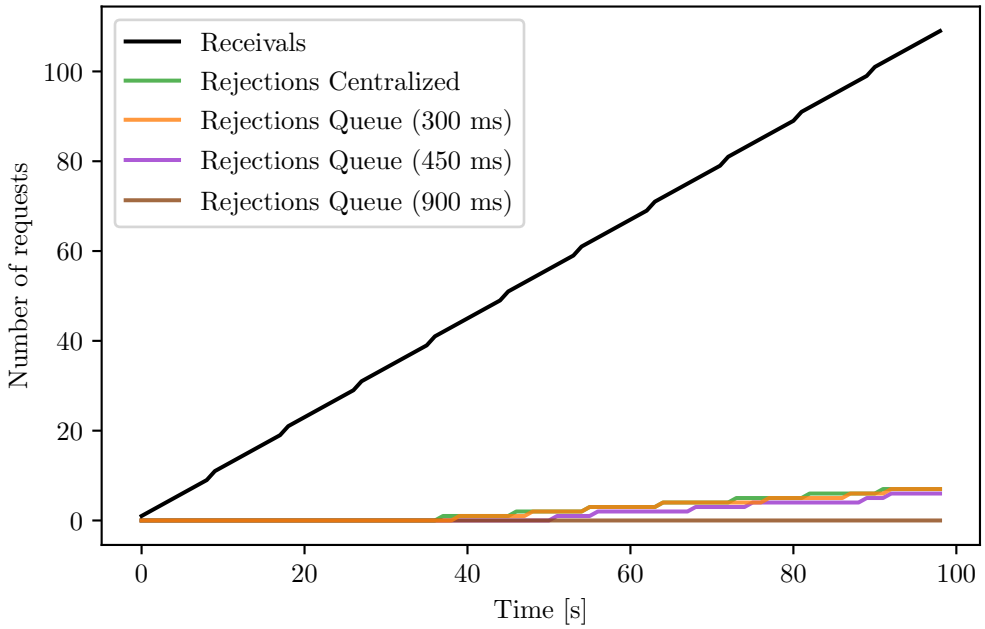


Figure 5.15: The number of rejections for data with requests at a slightly higher frequency than the refill rate when using different gossip intervals

a probabilistic problem. Given the request rate of one every 0.9 seconds, the first request is sent at the test's inception and the second request 900 ms after. There are two scenarios for a collision between the elements corresponding to the two first requests. The first scenario is for a collision to happen immediately on the node that receives the second request. For this to occur, the element corresponding to the first request must be available on the node that will receive the second request

for the new element to have the desired 100 ms extension to its time-of-death.

When the gossip interval is set to 300 ms, it means that the node receiving the first request will be able to send that element to three of its neighbors. After the first synchronization message at 300 ms, another node will also propagate this element to one of its neighbors at 600 ms. At timestamp 900 ms, both neighbors who have received this element through a synchronization message will propagate it another time. This scenario is illustrated in Figure 5.16, where the circles represent nodes and arrows the synchronization steps that take place at each respective timestamp. In total, a gossip interval of 300 ms thus yields that eight nodes will have received the update before any node in the system receives the second request. This is the best-case scenario, where the fact that nodes can choose the same neighbor to propagate to is disregarded. Out of 30 nodes in the system, this gives a probability of 27 % for an immediate collision with the next request.

There is also a probability for collisions between the two elements after the second element's initial inception. Recall from Section 4.2.1 that nodes do not remove elements from their delta states unless they either send a delta state to a neighbor or refill their own delta states due to a merge with a received delta state. For this reason, the nodes that hold elements that are considered dead will continue to propagate these elements until either of these two cases occur. The second element will also be propagated and can then collide with any of the nodes that still hold the first element in their local state. In both of these cases, the time-of-death of the second element will increase, which will then increase the probability of a collision when the third request is inbound. Once a collision has occurred three times in a row, the last element will have its time-of-death prolonged by 300 ms compared to what the first clause would have assigned it in Equation (4.2). This extension to the element's time-of-death means that the element will live in the queue long enough to be propagated during four rounds of gossip rather than three, as opposed to the previous elements. This additional round of gossip implies that the probability of a collision when receiving a new request increases as eight additional nodes now have access to the last received element. As this continues to occur, elements will gradually increase their probability of colliding with other previous elements. The likelihood of losing data then grows increasingly improbable.

Using the gossip interval of 450 ms instead means that the receiving node will send the element to two other neighbors. In total, this interval yields that at most four nodes will have received the first element before the second request is received by the system. Four nodes give a collision probability of 13 % for an immediate collision. The numbers corresponding to the gossip interval of 900 ms become two nodes and a probability of 7 % for an immediate collision.

The more nodes the system consists of, the more the probability for collisions decreases, meaning the relationship between the gossip interval and the number of nodes requires more careful consideration than the solution using counters.

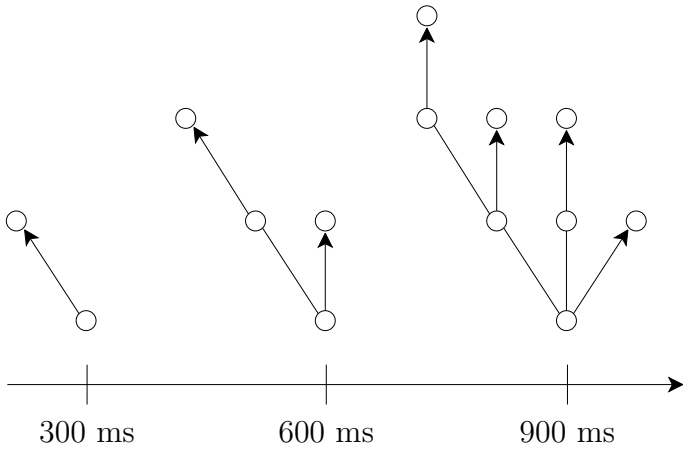


Figure 5.16: The spread of an element with time-of-death 1000 ms and a gossip interval of 300 ms

On the other hand, a possible solution to this problem is not to remove the last few elements when refilling. Keeping a safe margin when refilling would make collisions happen inevitable but at the cost of higher memory consumption. However, since the queue solution uses much less memory than the solution using counters when dealing with traffic from an average user, one can argue that an increase in memory can be justified. How much extra memory would be required depends on how many nodes the system consists of, the refill rate, and the gossip interval.

5.5 Final discussion

On the same theme as the closing discussion in Section 5.4, it is important to keep in mind that the results shown in this chapter apply in the context of the parameters used. How the proposed solution performs in other contexts is not evaluated in this thesis. For example, it is unknown how the queue would perform compared to the baseline in other network topologies than a fully connected one.

It can also be said that the baseline and the token-bucket queue have been compared in a simulated distributed system on one machine. Evaluating them in a physically distributed environment would introduce a few additional factors that could impact the results. Examples of such factors are communication delays and node failures. While it is not possible to make any claims on how the solutions would compare in such a system, this can be speculated on by assessing the results of this chapter and the implementations described in Chapter 3 and Chapter 4.

It is reasonable to believe that communication delays could impact the precision of the token-bucket queue. As discussed in Section 5.4, the probability for colliding elements are affected by how often nodes synchronize with each other, and these collisions are an essential part of not losing information about approved requests. However, the delay of synchronization messages would have to be large enough to consistently miss out on synchronization rounds to have an impact. For example,

if nodes synchronize every 200 ms and the communication delay is 250 ms, the elements in each synchronization message would miss out on at most two rounds of synchronization at each node it is sent to, compared to if there was no delay. The counter would also miss out on synchronization rounds, but it is not susceptible to losing data due to removing elements. The effects of communication delays may seem as if they could impact the performance of the token-bucket queue. However, we argue that since the main cause of this impact is that the rate at which nodes send synchronization messages is no longer equal to the rate at which messages are received, this could be mitigated by synchronizing more often. It is already argued for in Section 5.4 that the token-bucket queue requires more careful consideration of the gossip interval than the token-bucket counter. Synchronization delays would thus be an additional parameter to consider when deciding a gossip interval.

In terms of node failures, we believe that the two solutions should not behave with much difference. Any user data that has not already been propagated to any other node will be lost, and any data sent to at least one node will be further distributed through the system. However, the presence of node failures often requires dynamic reintroduction of nodes, and a reintroduced node for a token-bucket counter would likely catch up to the other nodes faster than a token-bucket queue would. The counter will likely receive values close to the greatest value that each counter has accumulated for its users with each synchronization message and thus require relatively few messages to catch up with other counters. The queue, on the other hand, would not catch up to the states of the other nodes before all elements that the other nodes held prior to the reintroduction have been removed, unless all elements have their time-of-death updated and thus sent to other nodes again. We would not consider this as something that speaks against the use of the token-bucket queue in a real-world system, as it could be solved by having the reintroduced node copy the full state of another node. However, this procedure would be necessary to add to the implementation of the queue solution, as the implementation described in Chapter 4 does not account for the reintroduction of nodes.

6

Related Work

In this thesis, we designed and evaluated a replicated token bucket for performing rate limiting in a distributed system. The idea of distributing observation of a system's state is not a new concept. The model of continuous distributed monitoring is described by Graham Cormode as "a number of observers [that] each see a stream of observations. Their goal is to work together to compute a function of the union of their observations" [6]. This could describe a number of specific approaches to observing and analyzing data, but the description encapsulates distributed rate limiting quite well. Cormode also describes the difficulty of finding a balance between sharing observations too often or not often enough to obtain desirable results without incurring heavy communication costs to justify the results. As the world of distributed systems deals with data in many shapes and quantities, implementations following the continuous distributed monitoring-model have to be tailored to the needs of the system that employs it.

One approach for performing distributed rate limiting is to distribute the rate limiting across some number of nodes and continuously update a joint storage for data that is utilized by each rate limiter globally across a system. This approach has seen use in the prevention of distributed denial-of-service attacks in network switches. Kim et al. [15] propose a solution for DNS amplification attacks, a type of distributed denial-of-service attack that targets a victim's network bandwidth with an overwhelming number of DNS requests. Their solution utilizes software-defined networks as a medium for switches to maintain a database with the history of previous DNS messages. This is done to maintain a larger history than each individual switch can maintain, and thus mitigate attacks that target multiple switches. This approach to continuous distributed monitoring is feasible for the token bucket algorithm. It is however more susceptible to failures in databases than our solution, as it separates the rate limiting from the bulk of data that is used for it. Failures would in turn allow more requests to be accepted as nodes have fewer options for transmitting updates, and this could, in turn, lead to worse precision than what is desirable.

Another approach is to divide the resources evenly between a number of rate limiters. For example, if a token bucket with refill rate r and capacity c is distributed over n nodes, each distributed part of the bucket could be given a refill rate of r/n and

a capacity of c/n. Li et al. [18] designed a solution for distributed rate limiting in a scalable real-time messaging platform by following this approach. This approach seemingly requires traffic to be evenly distributed between nodes to fully utilize the capacity of each bucket. However, their solution enables dynamic redistribution of both capacity and refill rate between buckets to accommodate traffic when necessary. While this approach is a feasible option for distributed rate limiting, it is not a good fit for our problem. We had a heavy emphasis on the availability that our solution provides, and dividing the total number of tokens each user can consume between different nodes would be counterproductive.

A third approach is to replicate both the data used for rate limiting and the rate limiting itself. Abu-Libdeh et al. have designed and evaluated Ajil [2], a distributed rate-limiting protocol for data centers. Ajil consists of nodes, each responsible for some part of the data center, that adjust the rate at which data is distributed through multicast channels based on locally available information. The nodes exchange information about how much data is being distributed over the multicast channels, and the adjustments to their sending rates are based on estimates of the system's total sending rates. This approach favors high availability, or rather maintaining high utilization of a system's communication channels and overwhelming the system capacity as infrequently as possible. This protocol performs rate limiting in a preemptive manner, as the allowed utilization of communication channels is based on estimates of the system traffic. Thus, it suffers from false positives where the utilization of communication channels would be reduced incorrectly. As stated in Section 3.4.2, it was not desirable to introduce false positives for the solution evaluated in this thesis. Therefore, a preemptive approach would not have been ideal.

7

Conclusion

In this thesis, a problem of the methodology used by one of the rate-limiting systems at Spotify was investigated. Rate-limiting systems employing the methodology are vulnerable to users exploiting the fact that the systems are distributed among multiple nodes that synchronize seldom. The thesis aimed to decrease the time it takes for a node in such a distributed rate-limiting system to make the same decision as a centralized system that performs the same task and receives the same set of requests.

A relaxed, distributed, and delta-synchronized queue to be used specifically for distributed rate limiting was designed, implemented, and evaluated. It was shown that it enforces the same token-bucket behavior as the baseline that makes use of CRDT counters. This queue was utilized to allow frequency analysis of user requests. The frequency analysis allowed the system to detect users that approached the limit of their token bucket abnormally fast and respond by synchronizing data for these users faster than the nodes normally synchronize.

The queue solution was compared to the baseline with user traffic from one of the rate-limiting systems used by Spotify. Compared to the baseline, the queue solution could react quicker to users that generate large amounts of traffic and should be rate limited. However, as a side-effect, the queue solution used substantially more memory and bandwidth when dealing with such users. More specifically, it was shown that the more traffic a user generates, the more memory and bandwidth it made use of in contrast to the counter solution. Moreover, for users that do not exceed the limit or are on the verge of doing so, which constitutes the vast majority of users for this rate-limiting system at Spotify, the queue solution used only 13.6 % of the baseline's memory consumption, whereas similar amounts of bandwidth were used.

On top of this, the queue solution took longer than the counter solution to converge the more requests that the system received from a user. However, the longer convergence time was shown not to influence the precision of the queue solution.

Moreover, it was shown that the queue solution requires more careful consideration when assigning values to the different configuration parameters such as the gossip

interval, the number of nodes, and the refill rate. While an evaluation of the proposed solution deployed on a set of real servers was not carried out in this thesis, which instead focused on the feasibility of the solution in a simulated environment, an interesting future direction is to evaluate the queue in a physically distributed environment.

To summarize, the proposed solution provides the system with the tool to combat users that send many frequent requests to multiple nodes while also improving the memory consumption substantially in the standard use case.

Bibliography

[1] Spotify AB. Company Info. Available at `https://newsroom.spotify.com/company-info/` (Accessed: 2021-05-13).

[2] Hussam Abu-Libdeh, Ymir Vigfusson, Ken Birman, and Mahesh Balakrishnan. Ajil: Distributed rate-limiting for multicast networks. *Computer Science Dep., Cornell University, Tech. Rep*, 2008.

[3] Mingzhen Ai and Wen Yu. Design and Implementation of Cloud Storage Flow Access Control Based on Token Bucket Algorithm. In *Proceedings of the 2018 2nd International Conference on Advances in Energy, Environment and Chemical Science (AEECS 2018)*, pages 211–219. Atlantis Press, 2018/03.

[4] Google Cloud. Rate-limiting strategies and techniques. Available at `https://cloud.google.com/architecture/rate-limiting-strategies-techniques` (Accessed: 2021-05-11).

[5] Cloudflare. What is rate limiting? Available at `https://www.cloudflare.com/learning/bots/what-is-rate-limiting` (Accessed: 2021-05-15).

[6] Graham Cormode. The Continuous Distributed Monitoring Model. *SIGMOD Rec.*, 42(1):5–14, May 2013.

[7] Guanlan Dai. How To Design A Scalable Rate Limiting Algorithm. Available at `https://konghq.com/blog/how-to-design-a-scalable-rate-limiting-algorithm` (Accessed: 2021-06-13), 2021.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[9] Jeremy Van den Eynde and Chris Blondia. Token Bucket-based Throughput Constraining in Cross-layer Schedulers. *CoRR*, abs/1911.12079, 2019.

[10] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consis-

tent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[11] William Goddard. The Evolution of Cloud Computing – Where's It Going Next? Available at `https://the-report.cloud/the-evolution-of-cloud-computing-wheres-it-going-next` (Accessed: 2021-05-13).

[12] Suji Gopinath and Elizabeth Sherly. A comprehensive survey on data replication techniques in cloud storage systems. *Int. J. Appl. Eng. Res.*, 13(22):15926–15932, 2018.

[13] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Trade-offs in replicated systems. *IEEE Data Engineering Bulletin*, 39:14–26, 2016.

[14] I. Gupta, A.M. Kermarrec, and A.J. Ganesh. Efficient epidemic-style protocols for reliable and scalable multicast. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pages 180–189, 2002.

[15] Soyoung Kim, Sora Lee, Geumhwan Cho, Muhammad Ahmed, Jaehoon Jeong, and Hyoungshick Kim. Preventing DNS Amplification Attacks Using the History of DNS Queries with SDN. In *European Symposium on Research in Computer Security*, pages 135–152, 08 2017.

[16] Konstantinos Krikellas, Sameh Elnikety, Zografoula Vagena, and Orion Hodson. Strongly consistent replication for a bargain. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 52–63, 2010.

[17] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[18] Chong Li, Jiangnan Liu, Chenyang Lu, Roch Guerin, and Christopher D. Gill. Impact of Distributed Rate Limiting on Load Distribution in a Latency-sensitive Messaging Service, 2021.

[19] Lang Li., Xinming Tan., and Chao Deng. An Improved Token Bucket Algorithm for Service Gateway Traffic Limiting. In *Proceedings of the International Conference on Advances in Computer Technology, Information Science and Communications - CTISC*, pages 222–226. INSTICC, SciTePress, 2019.

[20] Kinshuk Mishra and Matt Brown. Personalization at Spotify using Cassandra. Available at `https://engineering.atspotify.com/2015/01/09/personalization-at-spotify-using-cassandra/` (Accessed: 2021-05-13).

[21] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free Replicated Data Types (CRDTs). *CoRR*, abs/1805.06358, 2018.

[22] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 337–348, New York, NY, USA, 2007. Association for Computing Machinery.

[23] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[24] Marc Shapiro and Bettina Kemme. Eventual consistency, 2009.

[25] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-Free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, page 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] Priya Venkitakrishnan. Rollback and Recovery Mechanisms In Distributed Systems. *Department of Computer Science, University of Texas at Arlington*, 2002.

[27] Xiaowei Yang, David Wetherall, and Thomas Anderson. A DoS-Limiting Network Architecture. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, page 241–252, New York, NY, USA, 2005. Association for Computing Machinery.