Bachelor's Thesis Computing Science

Distributed Virtual Waiting Room

Josja Koopmans
s1071047

July 28, 2024

First supervisor/assessor:
prof. dr. ir. A.P. de Vries (Arjen)

Second supervisor:
dr. J.C. Rot (Jurriaan)

Second assessor:
dr. J.S.L. Junges (Sebastian)

Radboud University

Abstract

With more people on the Internet than ever, websites need to be able to handle an ever-increasing number of simultaneous users. For some websites, this is not viable. These can use virtual waiting rooms to limit the number of people on the site simultaneously. Existing virtual waiting rooms are generally unfair, as throughput is prioritised over fairness. This thesis designs and implements a fair virtual waiting room, that still has an acceptable level of throughput.

Contents

Chapter 1

Introduction

Nowadays, handling a large number of people simultaneously trying to visit a website is an important challenge that more and more online platforms need to deal with. Whether it is ticket sales for a popular event like a concert, or giving out vaccination timeslots, tackling this challenge is complex. One way of addressing this problem is by using a virtual waiting room. This waiting room takes the excess of users trying to access the site and puts them into a queue so they can be let onto the site at a more manageable rate.

Unfortunately, there is a distinct lack of academic literature on the topic of distributed virtual waiting room. While several commercial solutions exist, these are often either closed-source or the interesting parts are abstracted away behind closed-source products. There is also no specification of desirable properties of distributed waiting rooms in the academic literature.

The commercial solutions that aim to solve this problem focus on throughput rather than fairness. Fairness in this context means how well the order in which users join the waiting room compared to leaving the waiting room matches up. If it is completely different, it has bad fairness. Most commercial virtual waiting rooms make no claim as to their fairness. This thesis aims to address these shortcomings.

To do this, we design a distributed virtual waiting room based on the distributed priority queue system called QPID [3]. We modify QPID, adding fault detection using SWIM [4] and user eviction using TAG [7]. We also add the ability to arbitrarily change the servers the queue is running on.

This results in a distributed virtual waiting room that is fair, has an acceptable level of throughput, and is horizontally scalable.

In this thesis, we make the following contributions:

- We specify the properties we expect from a virtual waiting room (Chapter 2).
- We design and implement a distributed virtual waiting room based on

an extended version of the QPID distributed priority queue (Chapters 3,4).

- We show that this design is correct and that it satisfies the desirable properties we propose in Chapter 2 (Chapter 5).
- We use deterministic simulation testing to demonstrate experimentally that the implementation of the waiting room behaves fairly (Chapter 6.1) and we show that it provides enough throughput for a real-world application (Chapter 6.2).

Further, we discuss how our waiting room compares to existing solutions (Chapter 7), and what further improvements can be made to it (Chapter 9).

Chapter 2

Specification

The goal of this chapter is to explicitly lay out the required specifications for a fair, distributed waiting room. We have not identified prior academic literature on distributed virtual waiting rooms, we need to define what we are discussing in this thesis. We start by defining what a waiting room is. Next, we precisely specify what is meant by fair and distributed in the context of this waiting room.

2.1 Virtual waiting room

A virtual waiting room sits in front of a regular website. If too many people try to access the website simultaneously, it puts the excess of people into a queue to ensure the site can handle all the incoming traffic.

When users are waiting in the queue, they are typically shown a special page created by the waiting room. This page may show additional information, like their expected wait time. As long as the user has this page open in their browser, the user stays in the waiting room. Once they are through, the waiting room's page redirects them to the real site, and they can start their visit.

To keep the users in the waiting room, the page, which is part of the *interface*, automatically performs actions on the user's behalf. The interface further consists of the communication layer between the user's device and the nodes running the waiting room.

The interface can trigger *interface operations*. The possible interface operations are enumerated below. The interface operations are performed by the interface on one of the servers running the waiting room. Which server is chosen for this is not part of the specification.

Some of these interface operations return small-sized data structures to the interface which identify the client and allow clients to prove who they are. These pieces of data are the *tickets*, for when the user is in the queue, and *passes*, for when the user is on the site. These identifiers are

cryptographically signed by the servers running the waiting room to prevent tampering. The interface ensures that these identifiers are included in the subsequent requests to the waiting room.

1. **Join** This request is triggered when the user first joins the waiting room. It returns a ticket, which lets the user prove the time they joined the waiting room in subsequent requests.

2. **Check in** This request is triggered repeatedly by the interface after it has obtained a ticket. The interface uses the ticket obtained during the **Join** operation to show who they are. The waiting room responds with the user's position in the queue, whether they can leave it yet, and an updated ticket. The interface discards the old ticket and uses the new ticket on subsequent requests.

3. **Leave** This request is triggered when the interface is informed that the user has been evicted from the waiting room's queue during the **Check in**. This happens once they are at the front, and an additional user is let on the site. To call this operation, the interface again sends the ticket. The interface is given a pass, which allows the user to access the site.

See figure 2.1 for a flow diagram of a typical user interaction, assuming fault-free operation.

2.2 Fairness and reliability

The most important property of this waiting room is that it is fair. The fairness of the waiting room is determined by how closely the order in which users leave the waiting room follows a first-in-first-out ordering. We use the normalised Kendall-Tau [5] distance metric to measure fairness.

In a distributed system, getting perfect fairness would usually imply a significantly reduced maximum throughput. Therefore, we will allow small deviations from the "perfect" ordering, also called *relaxed ordering*. While using this relaxed ordering does mean that the users are not let out in strictly the order they deserve, the number of "mistakes" in the ordering can be reduced to a degree that makes the slight unfairness an acceptable trade-off for the far greater throughput it allows. We aim to balance fairness and throughput, leaning towards fairness wherever reasonable.

The normalised Kendall-Tau distance is a good fit for our use case because it punishes elements far away from their fair position more harshly than elements close to their fair position. It does so by counting the number of swaps of adjacent elements needed to go from one order to another. This coincides with the number of required swaps when bubble-sorting an array,

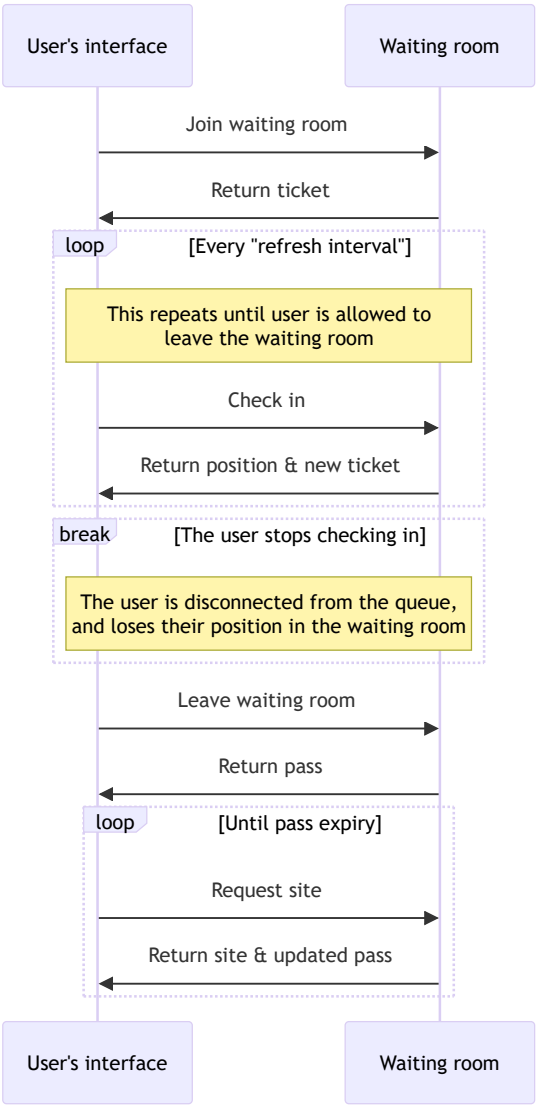

Figure 2.1: Flow diagram showing a typical user interaction without any errors.

hence why this metric is also called the bubble-sort distance. In this case, we compare the order in which users are let out of the queue to the order in which users joined the queue. The lower the Kendall-Tau distance, the closer the order of the two lists matches, the better the fairness.

The waiting room should also be reliable, meaning that there should be no single point of failure. The waiting room's reliability (also called resilience or high availability) is prioritized over the throughput. The waiting room should be able to run on multiple servers, and any of these servers should be able to go down without the waiting room ceasing to function. Of course, if *all* of these servers go down simultaneously, the waiting room will become unavailable.

These requirements are formalized as the following constraints that the waiting room specified here should adhere to.

1. The normalised Kendall-Tau distance of the waiting room should be close to zero (eg. $\tau < 0.005$).

2. The number of users that access the website simultaneously should stay close to the intended number of users. This means it may go over, but not by too much. It also means that, if there are too few users on the site, the waiting room should et users from the queue onto the site.

3. There should be no single point of failure; any part of the waiting room should be able to go down without the entire waiting room becoming unavailable.

2.3 Environment assumptions

To achieve the requirements set out in the previous section, we make some assumptions about the environment the waiting room will operate in. These assumptions are outlined below.

These different nodes are networked together, and we assume that if a network message is sent and the recipient has not failed, they will receive it in a reasonable amount of time. This is an unrealistic assumption in real-world deployments. However, this greatly simplifies the implementation. Namely, we do not have to worry about network partitions, as these are considered out of the scope of this project.

Additionally, to simplify implementation, we assume that nodes either fail completely or do not fail at all. There are no cases where a node is partially available. Failures should also be relatively rare.

Finally, we assume that the clocks of the different nodes running the waiting room are in sync, stay in sync while the waiting room is running, and increment monotonically.

The users intending to visit the website can connect to the waiting room over the Internet, and the waiting room can connect to the web server running the website.

Chapter 3

Distributed Priority Queue

We have based the design of the distributed waiting room on a distributed priority queue. This chapter will explain what this is and why we need one. Additionally, we will explain the queue we chose, QPID, and why this is a good fit for our application.

3.1 What is a distributed priority queue?

A queue is a data structure that supports two operations: enqueue and dequeue. The enqueue operation adds an element to the queue, and dequeue removes an element from the queue. The dequeue operation always returns the element at the front of the queue. Depending on how the queue was configured, this is usually the element added least recently (FIFO, first-in-first-out).

In a priority queue, the element at the front of the queue is the element with the highest priority, also called weight. This weight is determined when the element is added to the queue but may be updated later.

A queue being distributed means that the queue runs on several servers, and the operations on the queue happen across these servers. For example, if element A is enqueued at node 1 and then dequeue is called on node 2, element A should be returned. The advantage of a distributed queue over a non-distributed one depends on the trade-offs the queue makes. It could, for example, support more elements in the queue simultaneously, allowing for higher throughput [1]. An example of this is discussed in Section 7.2.2. Another focus could be being resistant to server failure. This is also called fault tolerance and is seen in the queue discussed in Section 7.2.1.

3.2 Requirements for waiting room use case

In our waiting room, we use a distributed priority queue. For the priorities of the items, we will use the time at which users join the waiting room. Using

the join timestamps as the priority can ensure that users will be added in the correct position if they need to be re-added to the queue. The dequeue operation will let the users out in a first-in-first-out ordering.

For our use case, we need a distributed priority queue with the following trade-offs:

1. The queue must be first-in-first-out ordered the vast majority of the time. As discussed previously in Section 2.2, we use the normalised Kendall-Tau distance as a metric for this. We want the waiting room to always have $\tau \leq 0.005$. This target is picked arbitrarily.

2. The queue cannot have a single point of failure, meaning that if any node unexpectedly goes down, as long as there is still at least one node running, the queue should continue to function. It is acceptable if some items in the queue are lost since we can re-add them.

3. The maximum number of items in the queue may not be limited by the amount of memory available on any single node.

3.3 QPID

We chose to use QPID [3] for our distributed priority queue. QPID has some of the properties we need built-in, and we were able to add the remaining properties ourselves. A discussion of alternative distributed priority queue designs, and why we did not choose them, is included in Section 7.2.

In QPID, each node of the network has its own local priority queue. QPID then lets these multiple queues act like a single, global priority queue. Each node can individually process enqueue (or in QPID terms: `insert`) and dequeue (`deleteMin`) operations. Enqueued items are added to the node's local queue and never move to another node.

3.3.1 Operation

QPID uses a rooted spanning tree to connect all the nodes in the network. Each node has a pointer (the QPID pointer) that points to itself or one of its neighbours in the tree. This pointer indicates the direction to the root of the tree. The tree's root contains the item with the highest priority. Starting from any node, following these pointers will lead to the root node.

The `deleteMin` operation is triggered whenever an element is dequeued. This operation travels along the pointers until it finds the current QPID root. There, the element at the front of the queue local queue is removed. This will also be the element at the front of the global queue. Then, the pointers in the network need to be re-aligned in order to let the next dequeue operation complete successfully.

To re-align the pointers, a `findRoot` operation is triggered. This operation traverses the network, re-aligning the pointers to point to the node that now holds the element with the lowest weight. Since the operation that finds the new root always starts from the old root, QPID ensures that there is always at most 1 root. Additionally, if these pointers are aligned correctly, QPID also ensures the current root node has the item with the highest priority.

To decide how to re-align the pointers, each node in the network maintains a so-called weight table. This table holds an entry for the current node and all of its neighbours. These entries are initialised as the value with the lowest priority. The value stored in the weight table at the current node's entry is the weight of the lowest item in that node's local queue. The value stored in node A's weight table at the entry of node B is the lowest weight of node B and all of its children in the spanning tree, as seen from node A. This could be seen as the lowest value on that "branch" of the tree. The QPID paper provides a nice visual example of how the values in the weight tables are calculated [3].

The values in the weight table may become outdated, as QPID uses "lazy updating" [3] to minimise the number of messages required. Whenever the weight of the first element of a local queue changes, which happens when a new element is inserted into an empty queue, a QPID `update` operation is triggered. This operation traverses the tree, updating all the required weight tables and potentially also triggering a `findRoot` at the old root if the root needs to be changed. For a more detailed overview of all operations, refer to the QPID paper.

3.3.2 Advantages and considerations

The values in the local queues are only stored on a single node. Additionally, depending on the layout of the spanning tree, the number of values in the weight tables increases slowly if more nodes are added. This means the amount of memory used on each node increases an insignificant amount with the number of nodes. Due to this, we can add as many nodes to the network as we need to store all the entries in the queue.

Adding more nodes to the network does result in reduced throughput, as the `deleteMin` operations will need to travel through more nodes before reaching the root nodes, increasing the time taken to let a single user out of the queue. Therefore, QPID works best in a setting with low latency between nodes.

While QPID does not balance the incoming items across the nodes itself, it works best if the items are spread evenly across the nodes. This can be achieved by having whatever is triggering the enqueue operations send them to a random node each time.

QPID maintains the distributed priority queue structure without exces-

sive communication overhead, without moving items between nodes, and without creating bottlenecks on any single node. However, there are also some downsides to QPID. Namely, there is no built-in way to add and remove nodes. QPID starts out with one node chosen as the root node, and all pointers pointing in the direction of that node. If nodes were added or removed without additional considerations, the rooted tree structure could be violated, breaking the assumptions QPID relies on for correctness. There is also no built-in fault detection.

3.4 Modifications to QPID

Since QPID does not support some of the features we need in the waiting room, we must add support for them ourselves. We briefly discuss these required modifications here and go into detail in the next chapter.

The first feature we need is adding and removing nodes from the network. As this operation happens infrequently, it is fine if it takes some time to complete. The entries in our queue can be re-added if they are lost since the waiting room repeatedly requests the user's position in the queue. This means we can recover from losing some of the elements in the queue. The simplest way to implement the changes to nodes is to stop letting people out of the queue, re-make and re-distribute the minimal spanning tree, and then restart QPID. However, this is an inefficient solution. Therefore, we specify a technique to add and remove nodes arbitrarily without stopping the entire network, discussed in Section 4.7.

The second feature we need is to know the number of users in the queue. This is required to let the correct number of people out of the queue and onto the site (also called eviction). This would be easy to do with a central authority that the nodes report to. However, this would be a single point of failure. Therefore, we have implemented an approach for collecting this information and doing the eviction inside the QPID network. This is further discussed in the Section 4.5.

Chapter 4

Distributed waiting room

In this chapter, we discuss the design implementation of the distributed waiting room. An implementation of the waiting room described in this thesis can also be found at https://github.com/zegevlier/waitingroom. This chapter presents the main contribution of this thesis.

4.1 Components overview

The waiting room consists of four components. These components and their roles are described in this section.

1. **Users** These are waiting in the waiting room, intending to visit the website behind the waiting room. They are accessing the waiting room over the Internet.

2. **Interface** The interface sits between the user and the nodes running the waiting room. It is responsible for communicating with the waiting room nodes and the website. It sends requests on behalf of the user and routes these requests to either the correct node as defined in the protocol or to the website once the user has left the queue.

3. **Nodes** These are the servers that run the waiting room's queue. There must be at least one node, but there may be more. The nodes keep all their data in memory and do not require persistent storage.

4. **Website** This is the website behind the waiting room. Users can only access this website once they have passed through the waiting room's queue.

See Figure 4.1 for an overview of the components.

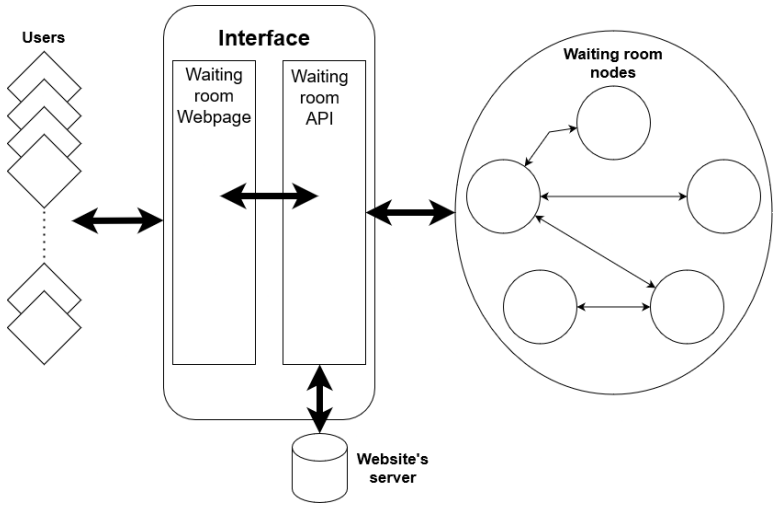

Figure 4.1: Overview of the different components of the waiting room and how they interact.

4.2 Interface operations

The operations discussed in this section are executed by the interface the user sees when they visit the website and are put in the waiting room. This interface includes both the front end, loaded in the user's browser, and the back end, loaded on the server running the waiting room. The interface makes sure the messages are sent to the same node as was previously used, except when that node has gone down, in which case a new, random node is picked.

4.2.1 Join

A **Join** is executed when the user is put in the waiting room. The initial join message is sent to a random node, determined by the interface. The user is issued a ticket (see Section A.1) and enqueued into the QPID local queue of the node the message got routed to, which triggers the QPID insert operation.

4.2.2 Checkin

The interface provides the ticket it received from either the **Join** operation, or the previous **CheckIn** operation. The node then verifies if the user is already in the queue at the current node. If not, it queues them at this node. This re-queueing happens only if the node the user was previously queued at went down; the interface ensures that the re-queueing takes place. Since the user is no longer in the queue when they re-queue, the user resides in

the queue in at most one node.

The node then estimates the user's position in the queue. If the user has already reached the front of the queue and was evicted, it instead signals to the interface that the user is allowed to leave the queue and enter the site.

Finally, it generates an updated ticket with the new information and sends it to the interface.

4.2.3 Leave

When the user's interface receives the signal that the user can leave the queue, their interface calls this method using the ticket. The user's interface will then receive a pass that grants them access to the site. This pass is automatically refreshed every time they send a request to the site. The user's ticket is removed from the list of evicted users, and their pass is added to the list of users on the site. Periodically, the waiting room will remove any expired passes from this list.

4.3 Priority queue

As discussed in Chapter 3.3, our system uses QPID as the priority queue. However, we must modify QPID to make it work well in our system.

4.3.1 Removing unnecessary messages

The first modification is not strictly necessary but does improve performance without any downsides for our use. In traditional QPID, when the `deleteMin` operation is triggered on a node that is not the root, the operation is forwarded to the parent, *and the result of this operation executed on said parent is returned*. Since we only need to know that a user can be evicted at the node the user is queued at, we do not have to return this result to the node that started the `deleteMin`, as they would do nothing with it; reducing the number of network messages transmitted. These returning messages are only used to return a value, and do not update the QPID state (e.g. the weight table or pointers), so we can safely remove them without altering the correctness of QPID.

4.3.2 FIFO workaround

One of QPID's assumptions is that the message channel between nodes is first-in-first-out. We are not making that assumption, so we introduce an approach to allow us to send the messages over a non-FIFO channel. We can do this with a simple counter attached to a subset the QPID messages.

Each node keeps a table with node IDs of the other nodes in the network, each with a corresponding count. This count is increased by one each time

an `update` or `findRoot` message is sent. This count is then attached to the message, and the receiving node ignores any messages with a count lower than the one previously received.

As we assume that no messages are lost, and a new value overrides any older values, this behaves identically to sending the messages over a first-in-first-out channel.

4.3.3 Miscellaneous other changes

We include an extra timestamp in the `findRoot` message as well; this is further explained in the upcoming section on user eviction. Finally, one additional check is needed when receiving an `update` message. This is discussed in the section on fault detection and recovery.

The rest of the implementation is based on standard QPID, although our codebase has been written from scratch.

The queue weights are made up of three components. When weights are compared in QPID, these are compared in order. The first part is the time the users joined the queue, then their ticket ID, and finally the node the user is currently talking to. The first ensures that users are sorted in the correct order. Users with a lower (earlier) join time will be sorted before users with a higher (later) join time. Then, if two users joined at the exact same time, the random value in their ticket ID is used to break that tie in a random but consistent way. Finally, the node ID is to ensure that any users that need to re-join the queue join the queue at a different position from their old ticket. This is required as QPID does not allow duplicate entries.

The items in the local priority queues are "tickets". The contents of tickets are detailed in Section A.1 of the appendix. The most important fields are the join time, user ID and the node the user is connected to. The first two are used to look up the user's entry in the local queue, and the latter is used to see if a user is already in the local queue or needs to be added.

4.4 Initialisation

When the network is initialised, one random node is picked as the starting node. This node sets its own value in the QPID weight table to 0 and sets its parent to itself. Once this happens, the node can start processing users. For any subsequent nodes added to the network, these nodes send a message to a node already in the network asking to join the network. This existing node then starts a membership change, adding the new node. Once this membership change is done, the newly added node will have all the information it needs to serve users. Section 4.7 describes how these membership changes work.

4.5 User eviction

User eviction is the event when we let users out of the waiting room onto the site. To do an eviction, we need to know the number of users on the site at a given moment. We use this number and the number of people we want on the site to determine the number of users we need to let out of the queue. How many users we want on the site depends on the load that the website can handle. In the current implementation, this amount needs to be set at the start and cannot change, but it could easily be made more dynamic.

Each node knows how many users are on the site using passes issues by that node. To do an eviction, we need to know the number of users on the site in total. This count is not known to any single node. Therefore, we need to aggregate this data across all nodes. We do this using TAG [7], a simple tree-based aggregation protocol, on the QPID tree.

4.5.1 Triggering the eviction

At a specific interval specified by the administrator, we trigger an eviction on all nodes simultaneously. We can do this simultaneously on all nodes, under the assumption the nodes' clocks are in sync (Section 2.3). When the eviction is triggered, we do nothing unless the node triggering the eviction is the root node. The eviction operation should only happen on the root node, and using this mechanism we can ensure this happens without having the nodes communicate with each other. On the node that is currently the QPID root, we start the aggregation process discussed below.

QPID ensures that there are either one or zero root nodes at any moment. If there is a root node at the time of the trigger, the eviction starts right when the timer triggers. Otherwise, the eviction is skipped. Since a delayed eviction only slows down letting users out of the queue, it is not a problem if aggregations are sometimes skipped. Since the root only changes if the element at the front of the queue changes, and this happens most often when we are letting users out of the queue, there being no root likely means that we are currently still processing the previous eviction. If we are, we do not want to trigger a new eviction, as this would let users out of the queue who should not be let out yet. Therefore, it is desirable to skip the eviction in this case.

4.5.2 Aggregation

Before we can do the eviction, we need to know how many users are on the site. Additionally, we collect the number of users currently in the waiting room. This aggregation is started as the first step of the eviction process after we decide to go ahead with the eviction on that particular node. The goal of the aggregation is to collect the total number of users on the site and

in the queue by summing up the amount on all individual nodes. We call these numbers the "counts". The algorithm for the aggregation is TAG [7].

When an aggregation happens, the node running the aggregation sends a message to all its neighbours in the QPID network, asking them for their counts. All of these nodes then ask all their neighbours, except the one they got the message from, for their counts. This happens until a node has no other neighbours, in which case it responds with how many users are on the site at that node. Then, once the parent node has received a response from all its children, it adds its own counts to the totals and forwards the result to the node from which it originally got the request. This happens until the root node has received responses from all its neighbours. The total counts that end up at the initiator of the aggregation are the number of users on the site and in the queue over the entire network. This is then used in the actual eviction step.

4.5.3 Eviction

Finally, using the counts obtained in the previous step, we do the eviction. The total number of people on the site is compared to the desired number of people on the site. If fewer people are on the site than desired, we let min(number of people in the queue, desired user count − number of people on the site) users out of the queue. Letting a user out of the queue is done by calling the QPID `deleteMin` operation. When this operation happens at the root, the ticket at the front of the queue is dequeued and added to a separate list containing users who have been evicted from the queue but have not yet entered the site.

4.6 Fault detection

We need to be able to detect when a node goes down. We do this using a simplified version of SWIM [4]. In our simplified version, this consists of picking a random node from the network, sending it a ping, and, if it does not reply within the specified timeout, assuming it is unavailable and removing it from the network. Since we assume that all nodes can talk to all other nodes in the network (Section 2.3), we do not need to do the indirect probes of the traditional SWIM protocol.

Using SWIM, all nodes have a list containing all nodes in the network. Each node periodically requests a ping from a random other node, which must reply before the timeout. If the node does not reply, it is marked as faulty and removed from the network. When we notice a node is defective during fault detection, we trigger a membership change, with the faulty node being removed.

4.6.1 Optional components

SWIM offers several optional components. We only implement one of these, and our reasoning for choosing whether or not to implement these is discussed here.

We implement the Round-Robin Probe Target Selection ([4] Section 4.3) modification. The time-bounded detection of faults is desirable for our use case. This makes it easier to write tests that determine whether a node failure is dealt with correctly, as we can assume that if it is not detected after a specific time, it will never be detected.

The infection style dissemination proposed in the modifications section of the original paper ([4] Section 4.1) is not desirable for us, as this greatly complicates the fault recovery. Sending a single membership change event to all nodes rather than having each node receive it at a later point is a sufficient method in our case.

The suspicion mechanism ([4] Section 4.2) would delay the detection of faulty nodes. Since the entire queue may be paused if there is a defective node, detection speed is more important than a reduced false-positive rate. Additionally, the reduced risk of false positives is negligible for us, as we assumed all nodes reply in a reasonable amount of time if they are online.

4.7 Membership changes

Membership changes happen when a node is declared faulty by the fault detection system or a new node joins the network. These membership changes need to uphold certain requirements to ensure the correctness of the queuing in the waiting room. These requirements are outlined below:

1. The QPID spanning tree stays a spanning tree.
2. There is always, at most, one root in the QPID network.
3. All nodes in the network are informed of the membership change.

Since a membership change may be triggered at any node, all nodes must have enough information to complete the membership change. To do this, each node has the latest QPID spanning tree and a number indicating the version of that tree (the spanning tree iteration number). This number is increased each time the tree is modified and is included any time the tree is sent to other nodes.

When a membership change happens, the spanning tree is modified locally on that node; then, the new tree is broadcast to all other nodes, along with whether any nodes should be added or removed from the membership list. These other nodes will then apply the modified spanning tree by establishing or demolishing connections as outlined below.

4.7.1 Modifying the spanning tree

Modifying the spanning tree is either done by adding or removing a single node at a time. We want to ensure this change happens deterministically to reduce the number of conflicts we get. If the same change is made to the same spanning tree on two different nodes, the output should be the same. The only strict requirement for these changes is preserving the spanning tree structure. Desirable properties would include minimizing the number of connections added or removed and keeping the spanning tree balanced. This would reduce overall latency during the QPID communication, improving maximum throughput. The spanning tree is modified by the node that initiats the membership change.

First, we naively add the required node without any edges, or remove the required node and remove all edges that connect to that node. Then, we need to re-establish the spanning tree structure. We do this by establishing a consistent way that nodes are reconnected when two disconnected spanning trees are created.

We do this by finding all the connected components of the modified graph and taking two random ones. In both components, we select one node to connect to the other component. Which nodes are connected does not matter, so some arbitrary heuristics may be used to determine this. Once we have these two nodes, we connect them by adding a single edge. We repeat this, starting from finding the connected components until only one connected component exists. This process will return the spanning tree structure to any graph consisting only of disconnected sub-graphs with spanning trees. Removing or adding a single node to the graph will result in this structure, so we can apply this operation to reconstruct the spanning tree.

This method only creates a single connection when adding a node. When removing a node, the number of connections added and removed depends on the number of connections the original node had.

Once the modified spanning tree is created, the node that initiated the membership change broadcasts the tree, along with which node needs to be added or removed, to all other nodes in the network.

4.7.2 Applying a new spanning tree

Now we need to apply the new tree on all nodes on the network. When a node receives a new spanning tree from another node, it first checks whether the spanning tree it receives is newer than the one it currently has applied. This is determined by the iteration number of the spanning tree. If it is older, it is ignored. If it has the same iteration number, a conflicting change is detected, and this conflict is dealt with as outlined below. If the spanning tree iteration number is higher than the node currently has, it applies the

new tree. Note that at this point, a node may not have been initialised yet and thus may not have a QPID parent.

To apply the new tree, the node first determines the difference between the connections it is supposed to have and the ones it currently has. For any connections that were removed, we remove the entry from the weight table. We also discard the current parent, which means the new parent will be determined later. We then also set a flag that we need to trigger a `findRoot` operation once this node has decided on its new parent.

For any connections that need to be added, we only add this entry to our weights table.

Next, since we discarded the current parent if any connections were removed, we need to pick a new parent. If we have also added connections, we need to wait for an update from the other nodes before we can decide which neighbour should be the new parent. If we have only removed connections, we can immediately set the parent. How the parent is set is explained in the next section. If only connections were added, the parent is not changed.

We then send an update message to each of our neighbours that does not currently have the most up-to-date value. We determine this by keeping track of the latest value that was sent, and if it is not the same, sending a new update. This update message ensures that the values for both nodes in their respective weight tables is up-to-date.

We may have already received the update from our neighbour if their membership change message arrived earlier than ours. In this case, we can determine our new QPID parent immediately. If we have not received these update messages, we must wait until we receive this update before deciding on our new parent. When we receive an update message when we do not have a parent, we update the value in the weight table like normal, then we see if we have received a response from all neighbours. If we have, we pick a new parent. If not, we continue waiting for update messages. The decision on the QPID parent is explained in the next section.

4.7.3 Picking the QPID parent

If we have removed entries from the weight table while applying the new spanning tree, we need to decide on a new QPID parent. We can only decide on a new QPID parent if we have all our neighbours in the spanning tree as entries in our weight table. Before this, if we were to decide on the QPID parent, this decision could be incorrect. Once we have all the required entries, we know the weight of all of our neighbours and can thus decide on the correct parent.

Entries in the weight table can be "empty", which means they do exist but have no value. A node's weight is empty if there are no entries in the local queue. If any of the weights in our table are *not* empty, we pick the value in our weight table that is the smallest.

When all of the weights in our weight table are empty, which happens when no users are in the waiting room, we need to pick an arbitrary node to point to. In our case, we pick the node with the lowest ID in the network. We then take the neighbour on the path from the current node to the node with the lowest ID as our parent. This ensures that we always only have a single root.

Then, once we have decided on our new parent, if we had previously set the flag that we needed to send a find root, we do so now. This ensures that, if the correct root changed while the membership change was happening, the network still ends in the correct state. Finally, we send another round of update messages to all nodes that do not already have the latest value. Again, this is done because otherwise, operations that happen during the membership change can cause the network to get into an incorrect state.

4.7.4 Conflicting tree changes

Two spanning tree changes may happen at the same time. When this happens, these changes will have the same spanning tree iteration number. This is detected because nodes receive two trees with the same iteration number. If this happens, the tree is entirely reconstructed in a way that is not dependent on the order in which the nodes were added.

This reconstruction from scratch is done by taking the node list, sorting it on node IDs, and then adding each node to an empty spanning tree in this sorted order. This will happen on all nodes that notice the conflicting trees, and will result in the same trees on each node. These nodes then broadcast this new tree to all other nodes, just as with the regular membership changes.

Because this results in the same tree on all nodes, the next time the trees are broadcast, they will be the same and have the same version number. This ensures that, eventually, the network will stabilise on a single tree that all nodes agree on.

4.8 Scalability

We have now created a waiting room that can, just like QPID, run on an arbitrary number of nodes. Due to the modifications made, all nodes need to know of the existence of all other nodes, which the amount of memory needed for each subsequent node added does increase more quickly than with just QPID, but still not quickly enough to lead to problems. Adding more nodes does increase the time it takes for the network to process one `deleteMin` operation, and thus the throughput of the waiting room. However, for any reasonable network size with nodes close together, this is unlikely to cause problems.

Additionally, our waiting room can be put in front of virtually any website. The waiting room can operate completely independently from the web-

site. It knows how many users are on the site and how many users should be on the site, and can thus operate without getting any additional operation from the website. This makes deployment and scaling of the waiting room easier.

The number of nodes in the waiting room can also be scaled up or down without much overhead, which makes deploying it into an unpredictable context easier than one where one needs to pre-allocate the number of nodes.

Chapter 5

Analysis of Distributed Virtual Waiting Room

In this chapter we discuss the correctness of two parts of our waiting room. First we discuss the correctness of the network membership changes. Second, we discuss the correctness of the counting aggregation part of the user eviction.

5.1 Membership change correctness

We start by discussing the correctness of the membership change system. For this discussion, we assume the entire membership change operation is atomic, meaning that no other messages are happening while the membership change is happening. This is not true in the real application, but significantly simplifies the correctness discussion here. Therefore, we only show that the system works for an atomic membership change. We start by defining the properties that need to hold for QPID to function properly. Then we show that if these properties hold before the membership change, they still hold after the membership change.

These invariants were already briefly outlined in Section 4.7.

1. First, the QPID spanning tree must stay a spanning tree. This means any node must be able to reach any other node via exactly one path.
2. Second, there must be at most one QPID root node.
3. Finally, we need all nodes to be informed of the latest spanning tree, and of the network members.

The membership changes work in three steps: Updating the spanning tree, applying this spanning tree, and finding the QPID parent of every node.

5.1.1 Spanning tree update

In the first step of updating the tree, we take the old spanning tree and add or remove all the requested nodes. Which nodes are added and removed is specified by what initiated the membership change, and all nodes should support adding new nodes and removing all nodes except itself. This step will result in a spanning tree that can then be applied to the network.

We start by naively adding and removing nodes on the old spanning tree. Nodes that are added do not result in any changes to the edges. Nodes that are removed remove all edges connected to that node as well. This results in a graph that has one or more connected components, where all the connected components are spanning trees.

Now we start a looping process, where each iteration of this loop decreases the number of disconnected spanning trees by one, until there is a single spanning tree left. First, we find all connected components, which will identify all these disconnected sections of the graph. Then, we connect two of these components with a single edge, this connects these two separate spanning trees to a single, larger spanning tree. Now, we repeat this until there is a single spanning tree left.

This loop starts with a number of disconnected spanning trees, and results in a single connected spanning tree. We can clearly see that this loop results in a single spanning tree. Assuming this spanning tree gets applied correctly in the next step, the network has a single connected spanning tree. With this, we have partially shown invariant 1.

5.1.2 Applying a new spanning tree

When applying a spanning tree, there are two things a node could need to do: Adding a connection or removing a connection. Adding a connection means we initiate communication between two nodes, and removing a connection means we stop communicating between those nodes. This applies the new spanning tree correctly, thus showing that, after this process, invariant 1 holds. Additionally, since this applying of the spanning tree happens on all nodes, invariant 3 also holds.

In the algorithm, if we remove any connections from the node, we discard the QPID parent. This means the QPID parent will be re-picked later. This is required because the connection to the previous parent might have just gotten removed. If only nodes were added, we know the current QPID parent can stay parent. Since we send the update messages, if the parent was incorrect, after the update is processed the parent will be correct. This is not necessarily the case if any connections were removed. Therefore, we reset the parent, wait until we have enough information to pick the new one, and do so.

Now we have shown that invariant 2 holds when we have only added

connections to nodes.

5.1.3 Picking QPID parents

If a node has discarded its old QPID parent, it can pick our new QPID parent once it has the required up-to-date information. For this, it needs to have the weight for all its neighbours in the spanning tree. Once it has this, it can decide its new parent. If the entire tree is empty, all the entries in the weight table will be empty. In this case, it sets the QPID parent towards the node with the lowest ID. This is a single, consistent node and will let us make sure there is only ever one root.

If there are elements in the tree, we apply the same method of setting the parent as QPID does. The QPID paper mentions that the parent can only be set if we know we have up-to-date information from all nodes. Since we had previously sent the most up-to-date information to all nodes, we know we have the latest information and can set the parent correctly. This shows invariant 2 for all other nodes.

5.1.4 Conclusion

We have now shown that all 3 invariants hold. The network remains a spanning tree, all nodes are notified of the update, and there is at most a single root node.

5.2 Eviction counting correctness

For the user eviction, we need to know the number of users on the site and in the queue across the entire network. This information is gathered in an aggregation. This section discusses the correctness of this aggregation.

QPID ensures there is only ever one root, and since the aggregation can only start on the root node, two aggregations can never start at the same time. Since an aggregation can only be initiated on a single node, this node can never trigger the aggregation twice at the same time, and the current time is used as the aggregation iteration, we will never have two aggregations with the same aggregation iteration.

Nevertheless, it is possible to have two counts running concurrently if one takes a long time to complete. In this scenario, the aggregation with the higher iteration value, whichever starts later, can still be completed normally. The second aggregation causes the collected responses for the first aggregation to be discarded at any nodes that had already acted on the first aggregation. Additionally, since the iteration value is lower, all nodes that processed the second aggregation first will ignore the first aggregation. Responses for all but the current aggregation at a node are also ignored, causing only the second aggregation to be collected.

If a node is faulty during an aggregation, the aggregation will never be completed. However, this is not a significant problem because another aggregation will occur soon. Since one of our assumptions is that nodes rarely go down, we can be confident that an aggregation will be completed shortly.

Chapter 6

Testing

In this section, we discuss how we tested the waiting room behaviour using testing. We implemented the waiting room described in this thesis in code and simulated it running in different scenarios. We used this to test whether there were any cases where the waiting room behaved incorrectly. Additionally, we tested the waiting room's throughput in a non-simulated deployment to test its true throughput.

6.1 Simulation Testing

In this section, we discuss how we used simulation testing. Simulation testing replaces some components of a system with simulations of these components. This means the system can be tested in a more structured, extensive way.

6.1.1 Implementation

For our implementation[1] of the waiting room we used the popular programming language Rust[2]. All code that is directly related to the waiting room was implemented from scratch, using libraries for functionality like JSON parsing, logging, collecting metrics, randomness, etc. The non-simulation version also notably uses libraries for the HTTP server.

6.1.2 Hand-made tests

We used several hand-made tests to verify the waiting room's behaviour. These tests can be found in the code[3]. While these tests could point out major issues after changes were made, they are obviously not enough to conclude definitively that the system works correctly.

[1] https://github.com/zegevlier/waitingroom

[2] https://www.rust-lang.org/

[3] https://github.com/zegevlier/waitingroom/blob/main/waitingroom-distributed/src/distributed/test.rs

6.1.3 Deterministic simulation testing

For a more convincing set of tests, we turned to deterministic simulation testing. This technique was popularized by FoundationDB [11]. It involves writing the system being tested so that it behaves fully deterministically, then running behaviour tests on it while (deterministically) randomly triggering disruptions to the system, and then checking whether the system behaved as it should have.

Determinism means that if the program is run several times with the same input, it results in the same output every time. This might seem true for most programs, but if a program uses time, IO, multiple threads, randomness, etc., it is no longer deterministic. These factors might not behave the exact same way the second time the program is run, meaning they may introduce variation into the output, causing non-determinism. These nondeterministic parts can be avoided by using deterministic simulations of these parts, as done in our implementation.

In these tests, the goal is to verify if the waiting room behaves as expected. This means we do not look at the internal state of the waiting room, we just look at the behaviour that can be observed from the outside. This lets us be confident in our design, while still being able to change *how* the waiting room works without having to change the tests as well.

The waiting room here is much simpler than the database the FoundationDB team built, so it was easier to write it deterministically. In our tests, we used a deterministic random number generators, a simulated time, and a simulated network between nodes. Then, while developing the waiting room, we ran the tests explained below. If these tests failed, we knew we had a bug in our code.

As described earlier, to test whether the system is operating correctly, we introduce disturbances. This could be a latency spike in the network, a node losing connectivity to another node, or a node completely stopping abruptly. In our simulations, we only introduce the disturbances of a node losing power and a node being added to the network. For each test we specified how many times either of these things should happen, and the testing system would make it happen that many times. These disturbances are introduced randomly, the timing of which is decided using a pseudo-random-number generator seeded with the run ID. This way, we can still repeat the runs, but each different run will also have a different set of disturbances.

The advantage of deterministic testing is that, if the tests trigger a bug, it is trivially easy to reproduce this bug as many times as needed to solve the issue. This was helpful during development. Additionally, it means that the testing results are perfectly reproducible under the same version of the waiting room.

Methodology

As described Section 4.7, nodes need to already know at least one other member of the network before they are able to join the network. To accomplish this, we first initialise a single node on its own, then initialise all other starting nodes by having them join the network at the original node. Then, we decide when to initiate the disturbances in the network. We do this before the loop starts, to make sure we always run all the disturbances we want to run. We use the same system to decide when to have the simulated users join the waiting room.

Next, we start the main loop of our simulation. Each iteration of this loop represents one millisecond of time. During this loop, the following actions happen in order:

- First, increase the simulated time of the system by 1.
- It checks if the waiting room is in a consistent state. This is further explained below.
- Then, it lets all nodes process any messages they receive at that timestep.
- Then, it calls all the functions that happen on a timer. These are the eviction, fault detection and cleanup functions.
- Then, we let the simulated users refresh their tickets if their refresh time is it. We also let them leave the queue once they have been evicted.
- Then, if we currently need to have a user join the queue, we add a new user to the simulation.
- Then, if we need to do one of the disturbances, we do that.
- Finally, we check if the waiting room is done processing users. When this happens, or when the waiting room has taken so long that we can be certain it will never finish, we exit the loop.

Finally, at the end of the run, we calculate several metrics, including the total time taken and the Kendall-Tau distance between the order users were put into the queue and let out. We determine this order by, for each user in the simulation, taking the timestep at which they entered the queue and the timestep at which they were evicted from the queue, then comparing the two resulting lists.

As part of the simulation testing, we test whether several properties hold. The most important of which are that there is always at most one root, and that the QPID node invariant, as described in the QPID paper

([3] Equation 1, start of Chapter 5), holds. Since these properties might not hold if a node has gone down and the recovery has not happened yet, we do not fail the test until the test has passed at least once after we run the disturbance that removes a node from the network.

When running our tests, we were able to configure several properties. The most important of these are detailed in Table 6.1, along with an explanation of what they mean. Whenever multiple properties are named, every combination is tested. This results in 27 different tests. For each test, we collect the total amount of time *after* we stopped letting users in and the Kendall-Tau distance between the order in which users were put in and users left the queue. Each test is repeated 1000 times, and the average of these is reported.

Due to limitations in the simulation framework, we were not able to simulate more than 2500 users in a single run. Therefore, we carried out the throughput tests in a non-simulated environment (see Section 6.2).

Parameter	**Value(s)**	**Explanation**
Target user count	20/100/INF	The target number of users on the site. If fewer users are on the site, more are let on.
Ticket refresh time	600ms	How often a user's ticket is refreshed.
Pass expiry time	0 seconds	How long it takes for a pass to expire, 0 seconds means the pass expires immediately, making it so there are always 0 users on the site.
Fault detection period	0.5 second	How often the fault detection is ran.
Fault detection timeout	0.2 seconds	The amount of time a node has to respond to the fault detection probe.
Eviction interval	1 second	How often the eviction is ran.
Latency between nodes	Unif. random between 10-20ms	The simulated latency between any two nodes.
Starting node count	8	The number of nodes at the start of the simulation.
Total user count	100/500/2500	The total number of users that join and go through the waiting room.
Nodes added/removed count	0/1/5	We start with a certain number of nodes. Then, throughout the simulation, we both add and remove nodes. The number of nodes added is always the same as the number of nodes removed, and the amount is listed in this parameter.
Test duration	100 seconds	The amount of simulated time in which all users join and disturbances happen.

Table 6.1: Test Parameters

Results

Each run had a combination of a different number of nodes that was added and removed, a target number of users, and a different total number of users. For each run, we collected the total simulated time taken and Kendall-Tau distance between the real and fair order. See Table A.1 for a table with the raw results.

We never reached the throughput limit of the waiting room in our testing. This means that we cannot accurately make predictions of the maximum throughput using these simulations. Therefore, we test for this throughput separately (Section 6.2). We were, however, able to collect the Kendall-Tau distance for all of these different runs.

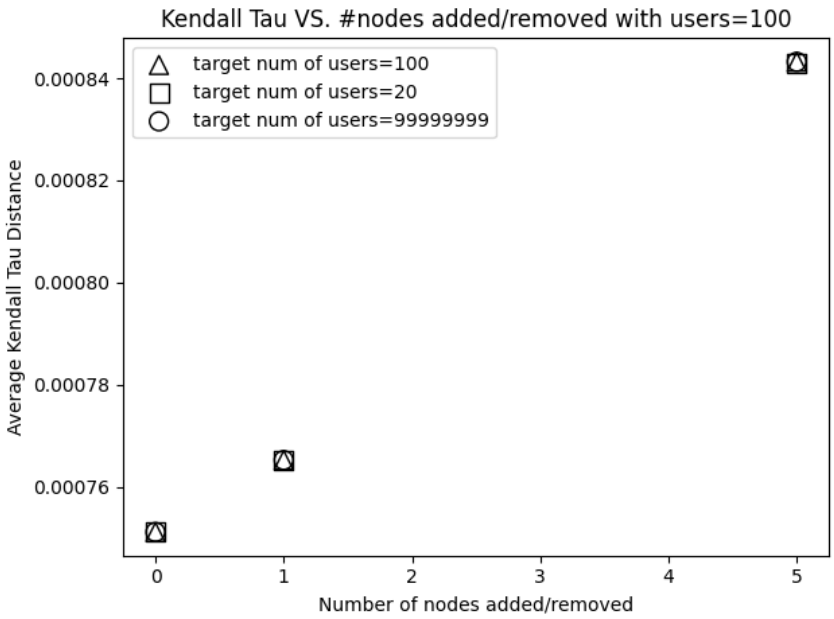

Figure 6.1: Kendall Tau VS. number of nodes added and removed with 100 users in total through the waiting room.

We have plotted the results of these experiments in three graphs, each for a different number of users that go through the waiting room (Figure 6.1, 6.2, 6.3). We plot the average Kendall Tau distance over the 1000 runs against the number of nodes that were added and removed in each of these runs. Take note of the different Y-axis on each of the graphs.

In Figure 6.1 and 6.2, the average Kendall-Tau distance for the different target user counts is nearly identical, as all users are immediately let onto the site in all of these tests. This means the dots for the markers overlap nearly completely.

We can see that having more users in the simulation results in a lower

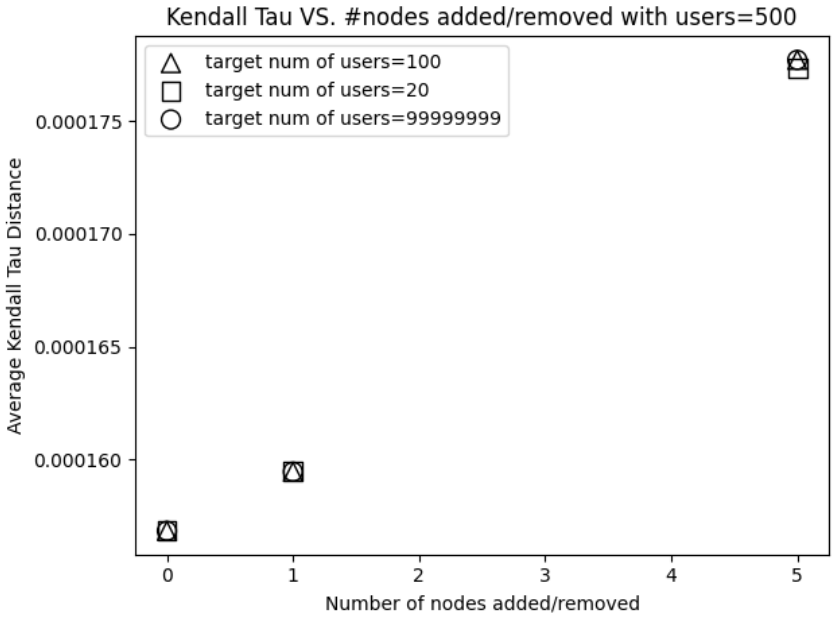

Figure 6.2: Kendall Tau VS. number of nodes added and removed with 500 users in total through the waiting room.

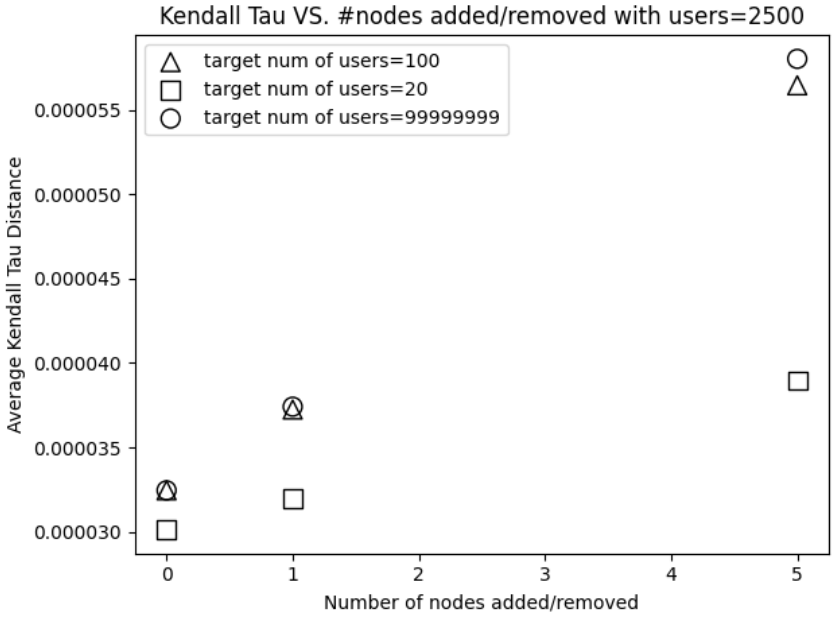

Figure 6.3: Kendall Tau VS. number of nodes added and removed with 2500 users in total through the waiting room.

(= better) Kendall Tau distance. This is because the swaps are more likely when there are few users in the queue, which happens more often when there are fewer users going through the waiting room. Additionally, we can observe that the distance increases if we have more nodes added and removed. This is because during the membership changes there is are more chances for users at the front of the queue to be let out in the wrong order.

Notably, in Figure 6.3, for a target user count of 20, we see only a small increase at the 5 nodes failed vs the 1 node failed, which can be explained if we look at the time taken for that test and the other tests in that set. We can see that these tests, in contrast to all the others, take significantly longer than the 100s to complete. This means that, at the end of the test, we still have a lot of users in the queue. This can be explained by the rate at which they are let out being lower than the rate at which they join. Since, like previously discussed, the swaps only happen when users join towards the front of the queue, and this happens less when there are more users in the queue, the number of swaps is lower here than in other situations. From this we can conclude that the waiting room's order becomes more accurate when there are more users in the queue.

6.2 Live Environment Testing

We also tested the waiting room in a live environment, which is closer to a real deployment than the simulation. These tests help us verify that the waiting room's throughput is not too limited for real-world deployments.

6.2.1 Methodology

As opposed to the simulation testing in the previous section, these test ran with seperate processes for each of the nodes, communicating over HTTP. As opposed to a real deployment, for our tests we ran all nodes, along with our interface and demo website on a single server (Hetzner Dedicated vCPU CCX13 with 2vCPUs and 8 GB RAM). The users were still simulated, this ran on a seperate VM in the same datacenter as the testing server. All simulated users ran on the same node. The latency between these servers was $\sim 0.25ms$.

As all nodes ran on the same system, the latency between them was near 0. This is unrealistic for a multi-region deployment but is close to what can be expected from a real deployment of several servers in the same physical rack.

For these tests, as the goal was to examine the maximum throughput of the waiting room, we did not simulate any nodes going down. Similarly, we also did not add any nodes after the waiting room started.

We simulated the users using a program that would send many simultaneous requests. In each run, we had 5000 simulated users going through

Number of nodes	3	8	16
Time Run 1 (s)	136.86	156.17	218.51
Time Run 2 (s)	138.79	140.55	222.72
Time Run 3 (s)	137.43	152.97	200.61
Average time taken (s)	137.69	149.90	213.94
Average users / s	36.3	33.4	23.4

Table 6.2: Table showing the results of the live testing.

the queue, with up to 500 user simulations running simultaneously. These users would follow the waiting room's instructions to refresh the page after a number of seconds and would stop once they had obtained the pass that allows them on the site. These users also communicate with the waiting room over HTTP.

We ran three different tests, one with 3 nodes, one with 8 nodes, and one with 16 nodes. We ran the 3 node and 8 node tests, as these would give expected ranges for real-world deployments. The 16 node test was chosen to more clearly display how the throughput changes with more nodes. We collected the amount of time it took to let all 5000 users through the queue. Each time, we could see the CPU usage spiking to 100%, which means we were sending enough requests to fully saturate the waiting room. Each test was repeated 3 times, and the nodes were restarted between tests.

6.2.2 Results

We got the results shown in Table 6.2.

As we can see, the waiting room is more than capable of handling 30+ users/second of throughput, which is more than enough for most sites. This shows that we can provide a fair waiting room that is fast with negligible overhead.

We can see that the throughput goes down when the number of nodes goes up. This is expected behaviour, as each delete request will need to, on average, travel further to get to the correct node. The chance of two consecutive deletes needing to happen on the same node is also smaller. Even though the throughput is lower, it is still likely fast enough for most use cases.

The throughput could be improved by using a more powerful system to run the nodes. Additionally, there is still a lot of low-hanging fruit optimisation-wise in the implementation. This proof of concept shows that the algorithm does not inherently bottle-necked the waiting room.

Chapter 7

Related Work

In this section, we discuss other waiting rooms and their implementations. Additionally, we will discuss other options for distributed queues, and why we did not use these to build our distributed waiting room.

7.1 Other distributed virtual waiting rooms

We were unable to find any academic papers discussing virtual waiting rooms. There are, however, several commercial companies that have blog posts or documentation explaining how their implementation of a virtual distributed waiting room works. These are discussed below, along with the advantages and disadvantages of their approach compared to ours.

7.1.1 Cloudflare

Cloudflare's waiting room [9] works by grouping all users into buckets at 1-minute intervals. The different nodes running the waiting room then communicate to aggregate the number of users in each bucket. Subsequently, the waiting room "evicts" (lets out of the queue) either entire buckets or parts from the queue. If a user is in a partially evicted bucket, it is random whether they will be in the part that gets let out or not.

This method has a major advantage over the user-by-user approach of our waiting room: the maximum amount of data that needs to be transferred between nodes is limited and predictable. This, combined with the other scaling mechanisms described in Cloudflare's blog posts, ensures that the throughput of the Cloudflare waiting room is virtually unlimited. However, this trade-off means that the order users are let out of the queue often deviates from the fair, first-in-first-out ordering. This trade-off is suitable for use cases where the site operator only needs to give the impression of fairness. Take, for example, online ticketing services. They do not inherently care about *who* buys the ticket, but for publicity reasons, they want to

ensure that the order feels fair for the user. There are other cases where true fairness is more important. Take, for example, a government handing out vaccination timeslots on a first-come-first-serve basis.

Another benefit of Cloudflare's approach is that the waiting room is only activated when too many users are trying to reach the site. In contrast, our waiting room enqueues all users regardless of the number of users on the site. While this creates more friction for users, it avoids the need for a complex system [10] that ensures that needs to decide when to start queueing. This needs to not happen too early, as this creates unnecessary user friction, nor to late, as this would overload the site. Although the implementation of such a system in our waiting room is technically possible, it was considered out of scope for this thesis.

In summary, the trade-offs of the Cloudflare waiting room differ from ours. We prioritize fairness, while they emphasize throughput.

7.1.2 AWS

AWS has a virtual waiting room template [2] that can be deployed on AWS onto one's own account. This template is open source[1], and they provide documentation[2]. This waiting room works by giving all the users in the waiting room an incrementing ID. The users are then added to a database, which stores information about when they joined and left the waiting room. Then, a globally shared counter is updated, and all users with an ID lower than the current count are let on to the site. The user can also get an estimate of their position by querying how many users in the database have not been let out and have an ID lower than the current user's.

In AWS's waiting room, all users are stored in a central place, namely DynamoDB. Additionally, the global counters are stored in ElastiCache for Redis. These are AWS products, and thus this waiting room can only run on AWS infrastructure. This requires complex deployments and has hard-to-predict pricing. Additionally, ensuring a complex system like this has no single point of failure is difficult, even with the managed databases offered by large cloud providers.

7.2 Distributed Priority Queues

While looking for a distributed priority queue for our project, we considered several different options. We briefly discuss these options here, and we evaluate their characteristics in the context of deployment in our distributed waiting room design.

[1] https://github.com/aws-solutions/virtual-waiting-room-on-aws

[2] https://docs.aws.amazon.com/solutions/latest/virtual-waiting-room-on-aws/welcome.html

7.2.1 FOQS

First, there is FOQS [8]. This queue utilizes a database, in this case, MySQL, to store all the users and provide the ordering. This offloads the task of keeping track of and sorting the entries in the queue to the database. This means the waiting room application only communicates with the database. This queue is distributed, because the database used for this can be made to be distributed.

This approach has several benefits. For example, the complex part of the queue is built on existing, well-tested software and algorithms. Additionally, the nodes the users would connect to are completely stateless (they do not store any data), meaning they are much easier to deploy at scale. Finally, as FOQS queues are backed by a database, it becomes easy to support auxiliary operations, such as seeing how many users are in the queue or removing someone not at the front of the queue.

However, FOQS has some noteworthy downsides as well. First, this approach shifts the problem of providing redundancy and strict ordering to the database rather than the application, which means we have far less control over how this happens. Additionally, it adds a layer of complexity in the deployment of the waiting room, as FOQS need the database storage to persist across server failure.

7.2.2 A scalable relaxed distributed priority queue

There is also another approach [6] that builds upon a local priority queue, similar to QPID. This approach randomly picks several nodes in the network to query and picks the element with the lowest priority returned by any of these nodes. This has some compelling benefits, namely that adding and removing nodes from the system is very easy. Additionally, the logic at each node is extremely simple and explainable since all they need to do is manage a local queue.

However, there are also some notable downsides to this approach. Firstly, the queue provides quite poor fairness when compared to QPID. As the order is based largely on probabilities, an unlucky user can be let out far behind their fair position. This possible unfairness only increases with the number of nodes and can make applications with a lower throughput extremely unfair.

This approach was too considered unfair for our waiting, but if fairness is less of a priority this mechanism could be a great option.

Chapter 8

Conclusions

In this thesis, we designed and implemented a distributed virtual waiting room. As opposed to other virtual waiting rooms, this one focused on fairness over throughput. This resulted in an exceedingly fair waiting room, with still an acceptable level of throughput. Our waiting room also has excellent scalability, with nearly no theoretical limit on the number of users in the queue simultaneously.

In order to make this waiting room, we took an existing distributed priority queue, QPID, and added several features to it. These include supporting changing the network members while the queue is operating, being able to run on a non-FIFO network, and detecting and automatically recovering when servers go down. Additionally, we optimised QPID for our use case.

Chapter 9

Future work

While the waiting room we have designed and implemented here is already fully functioning in the testing, we are missing some features that a real-life deployment could benefit from. Additionally, we have only outlined the correctness of this system. In this chapter, we will discuss these areas for future work.

9.1 Better position estimates

Currently, the position estimates provided to the user are calculated by multiplying their position in the node's local queue by the number of nodes in the network. For a small number of users, or when the user is near the front of the queue, this estimate significantly deviates from their real position. For example, if the user is at the front of the queue but not yet evicted, and there are 8 nodes in the network, their position estimate would be 8, while their true position would be 1. Future work should use a more complex system to estimate the user's true position in the queue, and should also provide the user with an estimated wait time.

9.2 Not queuing users when traffic is low

As discussed in the previous chapter on related work, we currently queue all users trying to reach the site. This occurs even if there are very few users on the site, resulting in users being let out of the queue almost immediately. This creates unnecessary friction for the user. Future work should implement this as well, possibly using a similar system of "user slots" that Cloudflare's waiting room uses [9]. These user slots allow each node to let a limited number of users through without queuing them. This reduces friction for users. Future research should add this feature to this waiting room.

9.3 More thorough analysis

Another way in which the waiting room could be improved is by providing a formal correctness for the membership changes. We informally discussed the correctness of this waiting room in the Section 5.1, but this was far from a formal proof. We conducted extensive tests to verify the behaviour, which provided confidence in our design. However, these tests were conducted on a simulation and it is almost certain we missed an edge case that could cause the system to malfunction. Future research should focus on creating a formal proof that verifies the modifications made to QPID work as intended, even in failure scenarios.

9.4 Advanced simulation testing

While our simulation testing is able to test a lot of situations, there are still many situations possible in the real world that cannot be simulated yet. These include users dropping out of the queue, messages from users and other nodes being handled interlaced, etc. Due to technical limitations we were also not able to test with more than 2500 users in the queue at the same time. Future work should improve this testing setup by making it possible to test these scenarios as well.

Chapter 10

Acknowledgements

Special thanks to Aditi Paul and George Thomas from the Cloudflare Waiting Room team and Joseph M. Hellerstein from UC Berkeley, who gave their time to meet with me to discuss my thesis.

Bibliography

[1] Dan Alistarh et al. "The SprayList: A Scalable Relaxed Priority Queue". In: *ACM SIGPLAN Notices* 50.8 (Dec. 18, 2015), pp. 11–20. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/2858788.2688523. URL: https://dl.acm.org/doi/10.1145/2858788.2688523 (visited on 07/05/2024).

[2] Amazon AWS. *Virtual Waiting Room on AWS*. AWS Solution. URL: https://aws.amazon.com/solutions/implementations/virtual-waiting-room-on-aws/ (visited on 07/04/2024).

[3] Ratan Bajpai, Krishna Kishore Dhara, and Venkatesh Krishnaswamy. "QPID: A Distributed Priority Queue with Item Locality". In: *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2008, Sydney, NSW, Australia, December 10-12, 2008*. IEEE Computer Society, 2008, pp. 215–223. DOI: 10.1109/ISPA.2008.90. URL: https://doi.org/10.1109/ISPA.2008.90.

[4] Abhinandan Das, Indranil Gupta, and Ashish Motivala. "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol". In: *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. IEEE Computer Society, 2002, pp. 303–312. DOI: 10.1109/DSN.2002.1028914. URL: https://doi.org/10.1109/DSN.2002.1028914.

[5] M. G. Kendall. "A New Measure of Rank Correlation". In: *Biometrika* 30.1/2 (1938), pp. 81–93. ISSN: 00063444. DOI: 10.2307/2332226. JSTOR: 2332226. URL: http://www.jstor.org/stable/2332226 (visited on 06/27/2024).

[6] Bill S Lin and Xiaohua Victor Liang. *A Scalable Relaxed Distributed Priority Queue*. Spr. 2020. URL: http://www.scs.stanford.edu/20sp-cs244b/projects/Distributed%20Priority%20Queue.pdf (visited on 07/05/2024).

[7] Samuel Madden et al. "TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks". In: *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. Ed. by David E. Culler and Peter Druschel. USENIX

Association, 2002. DOI: 10.1145/844128.844142. URL: http://www.usenix.org/events/osdi02/tech/madden.html.

[8] Akshay Nanavati and Girish Joshi. *FOQS: Scaling a Distributed Priority Queue*. Engineering At Meta. Feb. 22, 2021. URL: https://engineering.fb.com/2021/02/22/production-engineering/foqs-scaling-a-distributed-priority-queue/ (visited on 07/04/2024).

[9] Fabienne Semeria, George Thomas, and Mathew Jacob. *Building Waiting Room on Workers and Durable Objects*. The Cloudflare Blog. June 16, 2021. URL: https://blog.cloudflare.com/building-waiting-room-on-workers-and-durable-objects (visited on 07/04/2024).

[10] George Thomas. *How Waiting Room Makes Queueing Decisions on Cloudflare's Highly Distributed Network*. The Cloudflare Blog. Sept. 20, 2023. URL: https://blog.cloudflare.com/how-waiting-room-queues (visited on 07/04/2024).

[11] Will Wilson, director. *Testing Distributed Systems w/ Deterministic Simulation*. Sept. 2014. URL: https://www.youtube.com/watch?v=4fFDFbi3toc (visited on 07/04/2024).

Appendix A

Appendix

A.1 Data structures

A.1.1 Ticket

identifier (`TicketIdentifier`):
The ticket identifier is a random number used to uniquely identify a ticket. This same identifier is set on the pass the user gets when they are let out of the queue.

join_time (`Time`):
The time at which the user joined the queue.

next_refresh_time (`Time`):
The time at which the ticket should be refreshed next. They may refresh it sooner, but this is the time at which it should be refreshed automatically by the user's client.

expiry_time (`Time`):
The time at which the ticket will expire if it is not refreshed. The ticket will become invalid at this time.

node_id (`NodeId`):
The node ID where the ticket was last refreshed.

previous_position_estimate (`usize`):
The previous position estimate of the user. If the current position estimate is greater than this, the user is still shown their previous position estimate to prevent them from seeing their position go up, as this would be very discouraging.

eviction_time (`Option<Time>`):
The eviction time is the time at which the user was let out of the queue. When they call the refresh function after they have been let out, they'll be able to leave.

A.1.2 Pass

identifier (`TicketIdentifier`):
The identifier of the ticket that this pass was created from.

node_id (`NodeId`):
The node ID the pass was last refreshed on.

queue_join_time (`Time`):
The time the original ticket was added to the queue.

pass_creation_time (`Time`):
The time the pass was created.

expiry_time (`Time`):
The time the pass expires if it is not refreshed.

eviction_time (`Time`):
Eviction time is when the user was let out of the queue.

A.2 Simulation results

Note that these are the simulation results without any rounding applied to them. The first three columns are inputs, the last two are the averages of the outputs when running the simulations 1000 times.

Target user count	Total num of users	Nodes added/removed	Kendall Tau	Simulated time taken (ms)
100	100	0	0.000751	100276
100	100	1	0.000765	100324
100	100	5	0.000843	100436
100	2500	0	3.25e-05	101191
100	2500	1	3.73e-05	101238
100	2500	5	5.64e-05	101586
100	500	0	0.000157	100601
100	500	1	0.000159	100640
100	500	5	0.000178	100740
20	100	0	0.000751	100276
20	100	1	0.000765	100324
20	100	5	0.000843	100436
20	2500	0	3.01e-05	137183
20	2500	1	3.20e-05	139685
20	2500	5	3.90e-05	145619
20	500	0	0.000157	100601
20	500	1	0.000159	100642
20	500	5	0.000177	100744
99999999	100	0	0.000751	100276
99999999	100	1	0.000765	100324
99999999	100	5	0.000843	100436
99999999	2500	0	3.25e-05	101191
99999999	2500	1	3.74e-05	101241
99999999	2500	5	5.80e-05	101596
99999999	500	0	0.000157	100601
99999999	500	1	0.000159	100640
99999999	500	5	0.000178	100740

Table A.1: Simulation results