Performance Analysis and Re-Engineering of ExplorViz's Collaboration Mode

Johannes Brück

Master's Thesis
October 14, 2023

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Alexander Krause-Glau, M.Sc.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Modern software systems are becoming increasingly complex, necessitating advanced tools for comprehension and effective communication among developers. ExplorViz, a Software Visualization as a Service (SVaaS) tool, aids developers in understanding complex software systems. ExplorViz's Collaboration Mode enables interactive exploration of software landscapes in multi-user sessions across various platforms, such as VR or AR. However, the Collaboration Mode has not been tested under high user load, and the underlying architecture is not inherently scalable.

This thesis aims to comprehensively analyze the performance of ExplorViz's Collaboration Mode and, based on the results, re-engineer it to better support multi-user sessions with high interaction. To achieve this, we define a benchmark to competitively evaluate our re-engineering effort.

The benchmarking analysis revealed the underlying Collaboration Service and high message traffic as primary performance bottlenecks. We have undertaken a re-engineering effort, resulting in a system that enables horizontal scaling, reliable WebSocket connections, and efficient platform-specific messaging to minimize network traffic. Evaluation of the re-engineered system demonstrated high responsiveness and effective resource utilization. Horizontal scaling allows for a successive increase in capacity. However, scalability is somewhat impeded due to the high overhead for server coordination.

Contents

Chapter 1

Introduction

1.1 Motivation

The complexity of modern software systems is on the rise. Software visualization plays a critical role in understanding these complex systems. Additionally, effective communication among developers is a fundamental aspect of program comprehension [Maalej et al. 2014]. The research tool *ExplorViz*[1] aids developers in comprehending software systems by offering *Software Visualization as a Service* (SVaaS) [Hasselbring et al. 2020]. ExplorViz's *Collaboration Mode* allows users to interactively explore software landscapes in multi-user sessions across various platforms, such as *Virtual Reality* (VR) or *Augmented Reality* (AR), regardless of location [Krause-Glau et al. 2022a]. A user study has affirmed that ExplorViz is both enjoyable and useful for program comprehension [Krause-Glau et al. 2022a].

From a technical perspective, ExplorViz adopts a cloud-native microservice architecture, enabling horizontal scaling of multiple services [Krause-Glau and Hasselbring 2022]. However, the Collaboration Mode has not undergone testing under high user load. Furthermore, the underlying Collaboration Service, responsible for synchronizing clients in a multi-user session, is not inherently scalability.

This thesis aims to conduct a thorough performance analysis of ExplorViz's Collaboration Mode and, based on the results, re-engineer the Collaboration Mode to enhance its support for multi-user sessions with extensive interaction.

1.2 Document Structure

The thesis is structured as follows. Firstly, we present the thesis goals in Chapter 2. In Chapter 3, we elaborate on the foundations and relevant technologies for our work. In Chapter 4, we define a benchmark to evaluate the performance of ExplorViz's Collaboration Mode, and subsequently apply it to the current system in Chapter 5. Based on these results, we undertake the re-engineering of the Collaboration Mode in Chapter 6 and analyze the performance of the re-engineered system in Chapter 7. Following this, we discuss our work's results in Chapter 8. Finally, we present related research in Chapter 9 and conclude the thesis in Chapter 10.

[1] https://explorviz.dev/

Chapter 2

Goals

In the following, we present the goals of the thesis.

2.1 G1: Design of a Performance Benchmark for ExplorViz's Collaboration Mode

The objective of this benchmark is to assess the Collaboration Mode's capability to facilitate multi-user sessions with high user interaction. The systems under examination are the backend, specifically the *Collaboration Service*, and the frontend, denoted by the corresponding source code of *ExplorViz's Frontend* which is responsible for collaboration. The benchmark should define the test environment, suitable test scenarios, and performance metrics.

2.2 G2: Performance Analysis of ExplorViz's Collaboration Mode

We will apply the benchmark outlined in G1 to the previous Collaboration Mode implementation. Further, we will conduct an analysis to identify the causes of performance limitations and explore potential areas for improvement.

2.3 G3: Re-Engineering of the Collaboration Mode

The goal of the re-engineering process is to enhance the performance of the Collaboration Mode under high communication loads while maintaining its existing functionality. To achieve this, we will redesign the backend to enable horizontal scalability without compromising the reliability of client connections. In order to improve the performance of the frontend, we will pursue an explorative approach based on our findings in G2.

2.4 G4: Performance Analysis of the Re-Engineered Collaboration Mode

We will conduct a subsequent performance analysis for the re-engineered system. Therefore, we will reuse the benchmark established in G1 and, furthermore, successively scale out the backend by additional service instances. The results will be compared to the previous outcome of G2 in order to evaluate our measures in G3.

Chapter 3

Foundations and Technologies

In this chapter, we clarify the foundations and technologies on which the thesis is based.

3.1 Performance and Scalability

Smith and Williams [Smith and Williams 2002] define software *performance* as any characteristic of a software system that can be measured by "sitting at the computer with a stopwatch in your hand", which means that performance objectives usually refer to timeliness. Further, their definition determines *responsiveness*, which can be measured by response time or throughput, and scalability as main aspects of performance.

Scalability refers to the ability of a software system to meet its performance objectives as the demand increases, e.g. in case of a growing user population, by adding more resources to the system [Kounev et al. 2020]. The two dimensions of scalability are vertical scaling and horizontal scaling. Vertical scaling intends to scale a software system by providing more resources to a single node in the system, e.g. a more powerful CPU. Horizontal scaling introduces additional nodes and distributes the load between them [Kounev et al. 2020].

In the thesis, we will analyze and, moreover, improve the performance and scalability of ExplorViz's Collaboration Mode.

3.2 Benchmarking of Software Systems

Benchmarking in the context of software systems involves the assessment of a software system using a standard tool, known as a benchmark, to competitively evaluate and compare methods, techniques, or systems based on specific characteristics such as performance, dependability, or security [v. Kistowski et al. 2015].

The *ACM SIGSOFT Empirical Standards for Software Engineering*[1] provide a standard that describe the process of benchmarking as follows. Either the choice of an existing benchmark must be justified or a new one must be created, specifying the quality to be benchmarked, quality metrics, and measurement methods. Additionally, the experimental setup and workload or usage profile details should be described in sufficient detail to

[1] https://acmsigsoft.github.io/EmpiricalStandards/docs/?standard=Benchmarking

support independent replication [Hasselbring 2021]. The benchmark should allow different configurations of the system under test to compete without artificial limitations and should assess stability or reliability using sufficient experiment repetitions and execution duration. Finally, construct validity of the benchmark, ensuring it measures what it is intended to measure, should be discussed.

In this work, we will benchmark the performance of ExplorViz Collaboration Mode.

3.3 The WebSocket Protocol

The *WebSocket* protocol is a bi-directional communication protocol, which allows both server and client to send messages in a persistent connection. In order for two nodes to communicate, they introduce a WebSocket via a HTTP handshake and channel messages via a single TCP connection. By this means, WebSockets are especially suitable for real-time applications, e.g. instant messaging or gaming. An alternative for using WebSockets is HTTP polling, in which a client regularly sends requests to a HTTP server in order to receive data. However, WebSockets are much more efficient for implementing real-time applications [Liu and Sun 2012].

3.3.1 Socket.IO

Socket.IO[2] is a client server library for implementing WebSocket connections. Socket.IO introduces initially uses HTTP polling as transport protocoll in order to reduce the start time. Then, a built-in upgrade mechanism tries to establish a WebSocket connection. In the presence of proxies or firewalls, Socket.IO provides a fallback to HTTP polling. Network faults are handled by a re-connection automatism or a disconnection detection, which utilizes a hearbeat mechanism.

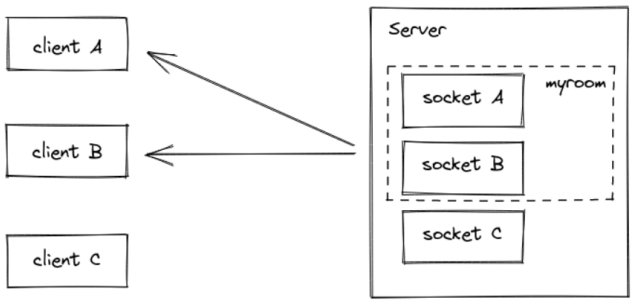


Figure 3.1. Socket.IO's room concept[3].

[2]https://socket.io/
[3]https://socket.io/docs/v3/rooms/

Furthermore, the library has cross-platform support and integrates separation of concerns by distributing connections between custom namespaces and rooms. A room is a server-only concept, which allows servers to broadcast messages to a subset of clients (see Figure 3.1). Sockets can join and leave rooms arbitrarily.

In our work, we will re-implement the ExplorViz's Collaboration Mode using Socket.IO in order to provide reliable connections in a scaled environment.

3.4 The Publish/Subscribe Pattern

The *Publish/Subscribe* pattern is a communication paradigm widely used in distributed computing to facilitate efficient information exchange among various system components [Eugster et al. 2003]. The core principle involves publishers which produce messages or events and subscribers which express interest in specific types of messages and receive relevant notifications (see Figure 3.2).

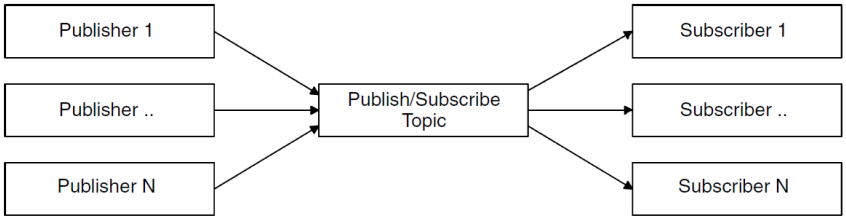


Figure 3.2. The Publish/Subscribe model [Curry 2004].

The messages are not transmitted directly but make a detour via a message bus. Therefore, Publish/Subscribe represents an alternative to straightforward point-to-point messaging [Curry 2004]. The decoupling between publishers and subscribers can be observed in three dimensions [Eugster et al. 2003]. Firstly, the nodes are decoupled in time since they do not need to handle the communication at the same time. The publisher can produce a message even if the subscriber is not active. Secondly, the nodes are decoupled in space. Publisher and subscriber do not need to know each other since all messages are transmitted indirectly via the message bus. Finally, there is a synchronization decoupling since the nodes are not blocked while exchanging messages. Instead, subscribers can execute concurrent tasks until they are notified.

3.4.1 Redis

Redis[4], which stands for Remote Dictionary Server, is an open-source, in-memory data structure store [Macedo and Oliveira 2011]. The core concept of Redis revolves around its

[4]https://redis.io/

key-value storage paradigm. Data is organized and stored in a key-value format where each key is a unique identifier associated with a corresponding value. These values can range from strings to complex data structures, providing flexibility in data representation and manipulation. Redis has built-in support for the Publish/Subscribe pattern [Macedo and Oliveira 2011]. Producers can publish messages to named channels while consumers can subscribe to messages that match a specific pattern to receive relevant updates. There are no conflicts between the Redis keyspace and channel names as they belong to different features.

Redis is a *NoSQL* database. NoSQL databases provide temporary, highly dynamic data, and stand in contrast to SQL databases, which focus on predictable, relational data [Macedo and Oliveira 2011]. The main advantages of NoSQL are reading and writing quickly, the ability to expand, and low cost [Han et al. 2011]. Han et al. [2011] provide a classification of Redis with regards to the *CAP* theorem in comparison to other NoSQL systems. The CAP theorem was founded by Eric Brewer and introduces the idea of a fundamental tradeoff between consistency, availability and partition tolerance in distributed systems [Gilbert and Lynch 2012]. Han et al. [2011] conclude that Redis is primarily concerned about consistency and partition tolerance, i.e., it works well in a high-scaled environment and ensures that all nodes have the same view of the data. However, there are other NoSQL approaches which come with a less restricted availability.

Finally, the Redis server keeps the datastore in-memory, i.e., the data is in volatile memory and not persisted on the disk. This allows reads and writes to be performed with extremely fast response times regardless of workloads [Kabakus and Kara 2017].

In the thesis, we choose Redis to implement the Publish/Subscribe pattern for inter-server communication.

3.5 NestJS

NestJS[5] is a server-side application framework for building scalable, maintainable, and efficient web applications in Node.js. The foundational principle of NestJS revolves around a modular, component-based architecture. It encourages developers to organize their applications into distinct modules, each responsible for a specific feature or functionality.

External communication interfaces are provided by *controllers* or *gateways*. Controllers in NestJS are responsible for handling incoming HTTP requests and orchestrating the application's response. Each controller is associated with a specific route and defines various endpoints. Controllers use decorators to define routes, request handling logic, and invoke appropriate services to process and manage data. However, gateways act as an abstraction layer for handling communication with external systems, APIs, or WebSocket connections. They provide a unified way to interact with external services, hiding the complexity of integration. Gateways in NestJS are platform-diagnostic and, particularly, compatible with any WebSocket standard.

[5] https://nestjs.com/

Components are integrated with the application by declaring *providers* (see Figure 3.3).

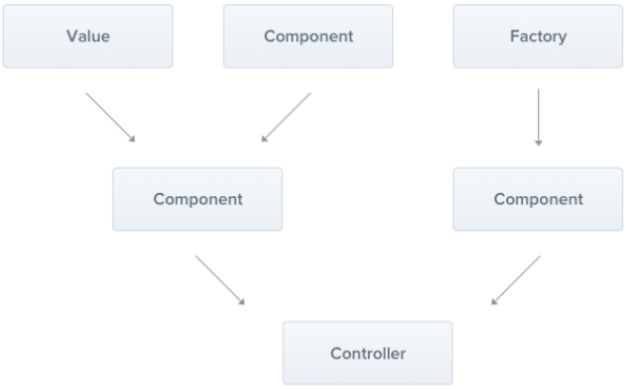


Figure 3.3. Dependency injection in NestJS[6].

Providers abstract the underlying implementation details to other components, promoting modularity and reusability. Through dependency injection, NestJS injects providers into controllers, services, or other providers. They can invoke providers to delegate complex tasks. Dependencies can be nested manifoldly and are resolved by the NestJS runtime system.

In our work, we will re-implement the Collaboration Service based on the NestJS framework.

3.6 Kubernetes

Kubernetes[7] is a widely adopted container orchestration platform designed to automate, manage, and scale the deployment and operation of containerized applications within a distributed environment [Hightower et al. 2017].

The basic building blocks of applications are *pods*. Pods encapsulate one or more containers sharing a common network and storage. Containers within a pod work together and can communicate seamlessly. Kubernetes allows the definition of resource limits for pods, specifying the maximum amount of CPU and memory that a Pod or container can use. Resource limits are crucial for maintaining resource fairness and preventing any single pod from monopolizing the available resources within the cluster.

Pivotal components in managing and enabling communication within a Kubernetes cluster are *services* and *ingress controllers*. Services act as an abstraction layer over a group of pods, providing a discoverable endpoint for clients within the Kubernetes cluster. An

[6]https://docs.nestjs.com/providers
[7]https://kubernetes.io/de/

ingress controller in Kubernetes manages network traffic which enters or passes through the cluster. It facilitates access to services and allows routing based on defined rules for optimizing application accessibility, e.g., load balancing or URL-based routing.

In the thesis, we will deploy ExplorViz inside a Kubernetes cluster to conduct a performance analysis.

3.7 ExplorViz

ExplorViz is a scientific software visualization tool which supports program comprehension of software systems [Hasselbring et al. 2020]. It allows users to explore software landscapes, which utilize a 3D city metaphor (see Figure 3.4), via various platforms, i.e., on-screen or *Extended Reality* (XR) [Krause-Glau et al. 2022b]. The visualization provides static aspects, i.e., the structure of the respective system, as well as dynamic aspects, e.g. communication between components at runtime.

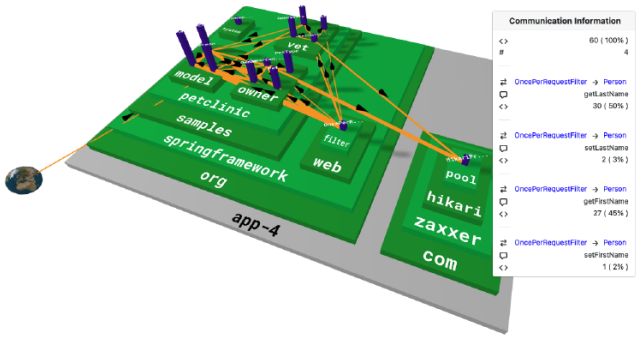


Figure 3.4. ExplorViz's application visualization [Krause-Glau and Hasselbring 2022]

3.7.1 Architecture

ExplorViz is built on a microservice architecture (see Figure 3.5). The *Adapter Service* is responsible for collecting and validating data from incoming monitoring sources. Analyzing the monitoring data and creating a software landscape model are the tasks of the *Structure Service* and the *Dynamic Service* (see 3.5-D). To render the visualization in a web browser, ExplorViz's Frontend (see 3.5-E) utilizes *WebGL*[8]. User interactions related to the landscape model are processed by the *User Service* (see 3.5-F), which then forwards events related to the software landscape to the respective analysis service.

[8]https://www.khronos.org/api/webgl

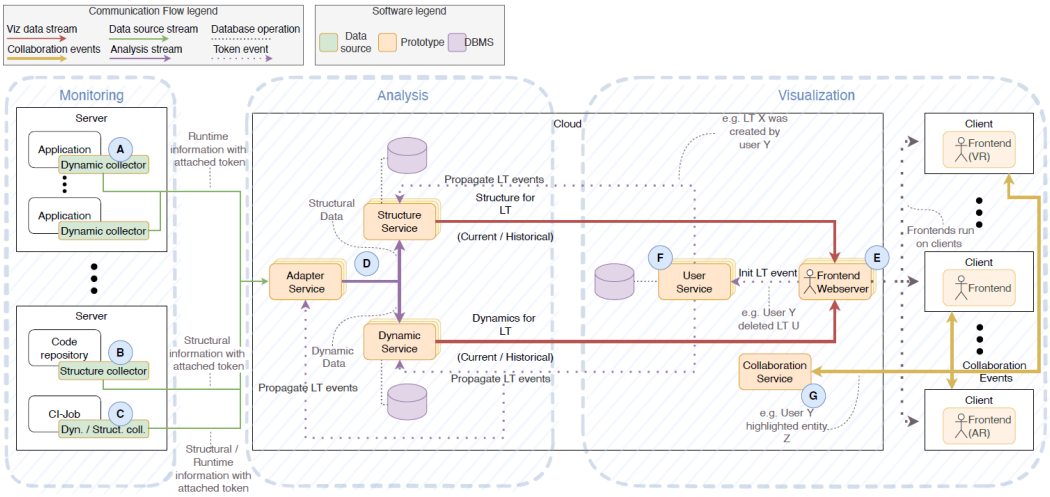


Figure 3.5. Conceptual design of ExplorViz [Krause-Glau and Hasselbring 2022]

The Frontend is an *Ember.js*[9] application and utilizes *Three.js*[10] for rendering purposes.

3.7.2 The Collaboration Mode

The Collaboration Mode allows multiple users to interactively explore a shared software landscape within the same virtual room [Krause-Glau et al. 2022a]. Apart from direct interaction with the software landscape, various features facilitate user-to-user interaction. For example, users can highlight interesting objects or attach technical insights in the form of pop-ups within the virtual room. A multi-user session is initiated when one user shares the current software landscape, and this landscape is synchronized via the Collaboration Service (see 3.5-G). Consequently, shared sessions are both location and platform independent.

Communication between clients and the Collaboration Service is enabled through WebSockets. The objective of the Collaboration Service is twofold: to validate user actions, such as opening a component of the software landscape, and to broadcast events to other clients to synchronize the scene's state, such as the position of individual users. Thus, the Collaboration Service acts as a single source of truth, managing a model of landscape and room relevant data. Clients also maintain a local replica of this data model to render the scene in the browser.

The Collaboration Service is implemented using the *Quarkus*[11] framework. The frontend

[9] https://emberjs.com/
[10] https://threejs.org/
[11] https://quarkus.io/

component is integrated with ExplorViz's Frontend as an Ember.js add-on.

In this thesis, we will re-engineer the Collaboration Mode to improve its performance.

Chapter 4

Design of a Performance Benchmark for ExplorViz's Collaboration Mode

In this chapter, we want to define a benchmark to measure the performance of ExplorViz Collaboration Mode.

4.1 Methodology

Since there is no existing standard benchmark which investigates the performance or any other relevant attribute of the Collaboration Mode, we must define a completely new one. Therefore, we consult the benchmarking standard ACM SIGSOFT Empirical Standards for Software Engineering [Hasselbring 2021].

The quality to be benchmarked is the performance of the Collaboration Mode to provide multi-user sessions with high interaction. The performance should be determined by conducting a load test with a high number of simulated users, which follow a certain usage profile as well as a stress test by ramping up the number of users [Menascé 2002].

However, the system under test is decomposed into two applications, the frontend and backend part, which are based on different runtime environments and fulfill different responsibilities. Therefore, we will organize the benchmark into two phases. In the first phase, we will benchmark the performance of the backend. The second phase will investigate the performance of the frontend.

In the following sections, we concretize the essential attributes of the benchmark. Firstly, we will provide the general framework of the performance test, which applies to both phases. Subsequently, we describe individual aspects for the backend as well as the frontend analysis, i.e., the experimental setup, metrics, and measurement methods. Finally, we discuss the construct validity of the benchmark.

4.2 General Framework

In the following, we describe the general framework of the performance test, i.e., the infrastructure, the workload, the test data, and the test plan.

4.2.1 Infrastructure

The system under test should be deployed in a containerized environment for the test to be representable and repeatable. Therefore, we will use the Kubernetes framework to easily scale individual deployments and allocate hardware resources based on our needs.

Furthermore, the performance test should be conducted on sufficiently powerful hardware. For this thesis, we use the cluster of the Software Engineering Group at Kiel University. The system specifications are displayed in Table 4.1.

Table 4.1. System specifications of the Kubernetes cluster at Kiel University[1].

Type	**Specification**
5x Kubernetes Master/Nodes	CPU: 2x Intel Xeon Gold 6130 (2.1 GHz, 16 Cores) Ram: 384 GB Kubernetes v1.23
Storage	120TB TrueNAS
Cloud Network	10GBase-T
Intelligent PDU	2x Rack PDU Raritan PX3-5190CR

4.2.2 Test Data

ExplorViz is a software visualization tool, which allows users to observe a 3-dimensional model of a monitored software system and to interact with it to gain a deeper understanding. The 3-dimensionaly landscape visualization of ExplorViz is generally built from landscape and trace data, that was retrieved from a static and dynamic analysis of a provided software system. For the performance test, a representable landscape model must be provided, on which the simulated users can operate. For this, we will reuse the distributed version of the *PetClinic Sample Application*[2], which has already been applied to benchmarking software visualization tools for program comprehension [Krause-Glau and Hasselbring 2022].

4.2.3 Workload

In the following we present the theoretical usage profiles as well as our technical approach for generating user load.

[1] https://www.se.informatik.uni-kiel.de/en/research/software-performance-engineering-lab-spel
[2] https://github.com/spring-projects/spring-petclinic

Usage Profiles

In ExplorViz's Collaboration Mode, multiple users are joined in a virtual room observing the same landscape model. All interactions which manipulate the virtual scene are synchronized between the users. To define a usage profile, we derive a representative user interaction scenario from the set of features that comes with the application. In Algorithm 1, we provide the *Browser Usage Profile*, which represents a user which experiences ExplorViz in a web browser.

Algorithm 1 Browser Usage Profile

```
1: repeat
2:     Highlight a component that initially catches their attention
3:     Detach a pop-up menu to present further information to all users
4:     Open a component to observe lower-level aspects of the application
5:     Ping to a component for other users to localize it
6:     Close a component to observe high-level aspects of the application
7:     Close a pop-up menu to clean the virtual scene
```

To simulate a longer user session, the scenario should be executed very often, e.g., 50 times. Additionally, there should be a delay between all actions of a scenario, which at least exceeds the minimal human reaction time to visual stimuli, i.e., above 200 milliseconds [Abbasi-Kesbi et al. 2017].

Furthermore, the Collaboration Mode of ExplorViz provides cross-platform support, i.e., all interactions with the landscape model are also available using VR or AR. Moreover, both XR perspectives extend the collaborative features of the browser perspective by visualizing an avatar of every XR user in the virtual scene. For this, every XR user's relative position is synchronized any time the physical position has changed. Therefore, we define the *XR Usage Profile* (see Algorithm 2) to cover the previously described feature as well.

Algorithm 2 XR Usage Profile

```
1: do in parallel
2:     task
3:         Run Browser Usage Profile
4:     task
5:         repeat
6:             Change position.
```

From a technical perspective, the user updates the position regularly with a short delay, which nearly correlates with the average frame rate of the application, e.g., less than 50 milliseconds [Liu et al. 2023].

Finally, we define two user populations which represent different multi-user sessions:

- ▻ *Browser Population:* 100% of the users follow the Browser Usage Profile.
- ▻ *Cross-Platform Population:* 50% of the users follow the XR Usage Profile, the other 50% follow the Browser Usage Profile.

The absolute number of users is not constant but should be varied during the execution of the performance test to ramp-up the workload. Furthermore, we differentiate between browser-only and cross-platform sessions since we assume a higher workload in the cross-platform session due to the high frequency of positional updates. However, we do not investigate a session with XR users only since both VR and AR are extensions to the default browser visualization and ExplorViz is designed to bring those different platforms together.

Load Generator

We produce the load by implementing a custom *Load Generator*. The Load Generator creates a new virtual room and, thereafter, establishes several WebSocket connections with the Collaboration Service. Each WebSocket client represents one end user and sends messages based on the previously defined usage profiles. Every test run has a constant number of client connections, but we ramp up the load successively by increasing the number of clients in following test runs. We present the test variables which are decisive for the generated workload in the following:

V1: The number of browser users per room.
V2: The number of XR users per room.
V3: The number of repeated scenarios.
V4: The duration between user actions.
V5: The duration between XR position updates.

Additionally, we can increase the load by deploying multiple instances of the Load Generator, where each instance creates an individual virtual room. This way, the test load can be scaled in two dimensions, i.e., the number of rooms as well as the number of users per room.

4.2.4 Test Plan

Since the Collaboration Mode has never been tested with a high number of users and we have no previous performance insights, the load test will follow an explorative approach. We will start with a small number of users and ramp it up while observing the performance metrics. In case the metrics reach a certain threshold, the application runs into an error state, or the metrics does not change at all for multiple iterations, we will stop to increase the load. The thresholds for stopping the execution are metric-specific and will be defined later. We will scale the load generator up to more instances, i.e., virtual rooms, in the same manner.

The performance test should follow the same explorative pattern for each component, i.e., backend and frontend, and each user population, i.e., browser and XR. The complete test plan is structured as follows:

Phase 1: Performance test of the backend.

(a) Simulate the Browser Population.
(b) Simulate the Cross-Platform Population.

Phase 2: Performance test of the frontend.

(a) Simulate the Browser Population.
(b) Simulate the Cross-Platform Population.

4.3 Backend

In the following, we present the aspects which are specific for benchmarking the performance of the backend.

4.3.1 Experimental Setup

The backend of the Collaboration Mode is represented by the Collaboration Service. Thus, the Collaboration Service as well as the Load Generator must be deployed (see Figure 4.1). The Load Generator can be scaled to multiple instances, i.e., Kubernetes pods. Each instance creates a room and, subsequently, establishes various client connections via WebSocket.

The physical resources which are provided to the Collaboration Service, i.e., CPU and memory, should be constant for all test iterations in order for the runtime behavior to be repeatable. In Kubernetes, this can be done by setting the upper bound and the lower bound of available resources for a specific container to the same value (see Listing 4.1).

Listing 4.1. Examplary resource configuration of Kubernetes deployment.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  ...
4     containers:
5        resources:
6           limits:
7                 cpu: 1
8                 memory: 2Gi
9              requests:
10                cpu: 1
11                memory: 2Gi
12          ...
```

This way, the container will not be scaled vertically by the Kubernetes framework even during high load peaks. However, the container of the Load Generator must not

utilize its resource limits at any time. Otherwise, it may lead to a performance bottleneck while the system under test is not even at its limits. We can ensure the container to have enough physical resources by omitting the upper limit and setting the requested resources very high. Furthermore, the actual resource utilization of the Load Generator should be monitored during the performance test to prevent distortion of the results.

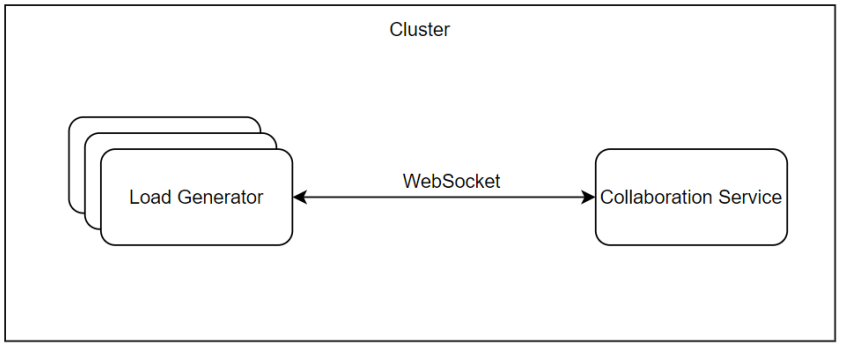


Figure 4.1. Deployment of the backend performance test.

4.3.2 Metrics

Since the backend is mainly responsible for handling and exchanging client messages, responsiveness is a suitable indicator for performance. From a high-level perspective, there are two types of client messages, *Request Messages* and *Synchronization Message*.

Request Messages are triggered by user actions that need server-sided validation, e.g., closing an object that may already be used by another user. The backend handles the request and sends a response back to the requesting client. We define the *Response Time* (RT) of Request Messages as follows:

$$RT(i) = T_{\text{request}}(i) + T_{\text{processing}}(i) + T_{\text{response}}(i) \tag{4.1}$$

Where:

- i is the message.
- $T_{\text{request}}(i)$ denotes the time taken to send request i to the backend.
- $T_{\text{processing}}(i)$ represents the time taken by the backend to handle request i.
- $T_{\text{response}}(i)$ signifies the time taken to send the response to request i back to the client.

However, Synchronization Messages are status updates, that needs to be shared with all clients in a room, e.g., highlighting a component of the landscape. Status updates are triggered by user actions that need no further validation. The backend integrates the update with its local data model and broadcasts the status message to all clients in the

room except the client that has sent the message. We define the *Mean Round Trip Time* (MRTT) of Synchronization Messages in one room as follows:

$$MRTT(i) = T_{\text{sending}}(i) + T_{\text{updating}}(i) + \frac{\sum_{\forall c \in C \setminus \{s\}} T_{\text{forwarding}}(i, c)}{|C| \setminus \{s\}} \tag{4.2}$$

Where:

- i represents the message that was sent by the client.
- s is the sending client.
- C is the set of connected clients without the sending client.
- $T_{\text{sending}}(i)$ denotes the time taken to send synchronization message i to the backend.
- $T_{\text{updating}}(i)$ represents the time taken by the backend to update the model based on message i.
- $T_{\text{forwarding}}(i, c)$ signifies the time taken to forward message i to client c.

In summary, the MRTT quantifies the average time it takes for a Synchronization Message to be delivered.

The RT as well as the MRTT should be kept low enough so that the user interaction is not significantly interrupted. For this, we will follow the guidelines of response time limits by [Nielsen 1994]. They identify 0.1 seconds as the limit for users to feel that they are directly manipulating objects in the user interface. Thus, we define the *Service Level Objectives* (SLO), which also represent the termination criteria of the performance test (see Table 4.2).

Table 4.2. SLO definition for the Collaboration Service.

Type of Measure	**SLO Requirement**
RT	The average RT for all Request Messages during one execution will be lower than 0.1 seconds.
MRTT	The average RT for all Synchronization Messages during one execution will be lower than 0.1 seconds.

Furthermore, the CPU and memory utilization of the Collaboration Service should be monitored to identify resource limitations. Additionally, we count the number of messages which are received and sent by the Collaboration Service.

4.3.3 Measurement Methods

The RT of Request Messages can be determined by measuring the difference between the timestamp in which a request is sent and the timestamp in which a corresponding response is received by the Load Generator. However, WebSocket messages are bidirectional and, therefore, responses do not inherently refer to a specific request message. However, we can

take over the same mechanism which the frontend uses to match request and response pairs by providing a nonce, i.e., a unique identifier, to every request message.

For the MRTT we instrument the backend code by adding a unique message identifier to Synchronization Messages, as well, so that the Load Generator recognizes forwarded messages. We measure the time between sending a status message and receiving the corresponding forwarded message for every client. After that, we aggregate the MRTT for every outgoing Synchronization Message.

In the Kubernetes environment we can retrieve the resource utilization of a container from the *control group* of the Linux kernel [Gao et al. 2019].

4.4 Frontend

The second test phase is concerned with the frontend of the Collaboration Mode and is described in the following.

4.4.1 Experimental Setup

The frontend of the Collaboration Mode was implemented in form of an Ember.js add-on on top of ExplorViz's Frontend. Thus, we must deploy the full ExplorViz Frontend, which again depends on ExplorViz's backend analysis services. These microservices serve user and landscape data for the frontend to authenticate the user and render the 3d-landscape model. However, we can replace the functionality of the microservices by the *Demo Supplier*[3], which is a minimal provider of relevant datasets. The replacement will not affect the performance results since we exclusively focus on the behavior of the collaborative features, which do not rely on the microservices. The complete setup is visualized in Figure 4.2.

Additionally, the experimental setup of the backend test is reused. Finally, we must deploy the *Client*, which accesses ExplorViz's Frontend in a web browser and enters the virtual room that has been created by the Load Generator before. Since the performance test should be fully automatized, we provide a script which uses the *Taiko*[4] framework to establish the browser session (see Listing 4.2).

Listing 4.2. Browser automation with Taiko.

```
1 (async () => {
2   await openBrowser({ headless: true });
3   await goto(FRONTEND_URL);
4   await click(button({ class: 'button-svg-with-hover' }, toRightOf('Room')));
5   ...
6 })();
```

[3]https://github.com/ExplorViz/deployment
[4]https://taiko.dev/

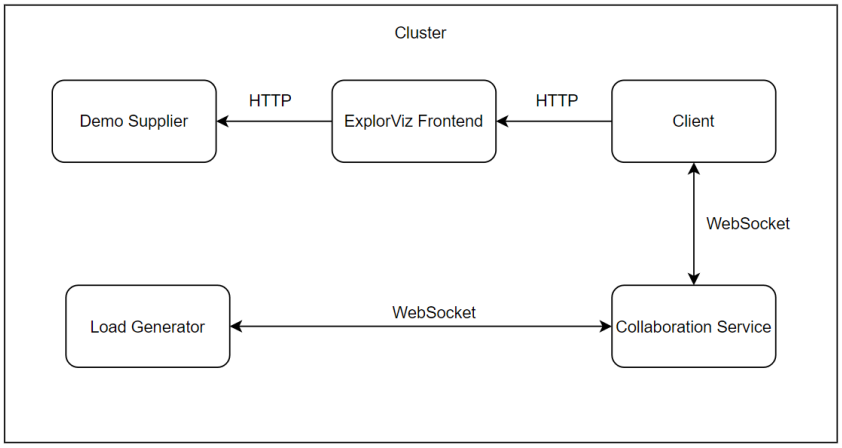


Figure 4.2. Deployment of the frontend performance test.

The Load Generator again simulates user load, which leads to incoming messages in the Client's browser. However, we use only one instance of the Load Generator since the monitored client can only be connected to one room at a time. As we did in the backend configuration, we provide constant physical resources to the Client container for the performance test to be repeatable.

4.4.2 Metrics

The goal of the frontend analysis is to measure the performance of those frontend functionalities that refer to user collaboration. The process which renders the 3-dimensional scene, for example, is part of the code base of ExplorViz's Frontend but no feature of the Collaboration Mode. Additionally, I/O operations are handled by the core of ExplorViz's Frontend and are also out of scope. However, the frontend of the Collaboration Mode manages the collaborative session, handles WebSocket messages, and updates the client-local model of the scene. Thus, we define the *Exectution Time* (ET) of received Synchronization Messages as follows:

$$ET(i) = TS_{\text{updated}}(i) - TS_{\text{received}}(i) \tag{4.3}$$

Where:

- i represents the message.
- $TS_{\text{updated}}(i)$ denotes the timestamp in which the client-local model is updated (but not necessarily rendered) based on message i.
- $TS_{\text{received}}(i)$ is the timestamp in which message i was received.

Again, we provide a suitable SLO definition (see Table 4.3).

Table 4.3. SLO definition for the frontend.

Type of Measure	SLO Requirement
ET	The average ET for all received Synchronization Messages during one execution will be lower than 0.1 seconds.

Futhermore, the CPU and memory utilization of the client instance should be collected. Finally, we count the number of Synchronization Messages that are received by the client.

4.4.3 Measurement Methods

To measure the ET of messages, we instrument the code of ExplorViz's Frontend as follows. For each WebSocket message we log a timestamp in the moment the message is received. Additionally, we log a timestamp after all message-specific event listeners have terminated. The difference of the previous values leads to the ET. However, the code is executed in the headless web browser. Therefore, the relevant logs must be retrieved from the browser console, which can also be done with Taiko.

4.5 Threads to Validity

In the following sections, we discuss internal or external factors which concern the construct validity of the benchmark.

4.5.1 Usage Profile

ExplorViz provides various features and allows the user a lot of room for maneuver. Therefore, it is a very complex task to predict the behavior of a real-world user. However, we identified two categories of user interactions, i.e., requests and status updates, each of which follows a specific computational pattern. The previously defined usage profiles include various representative actions for both categories.

4.5.2 User Population

Since ExplorViz has cross-platform support, a real-word user base may be very diverse with regards to the chosen device. Moreover, different devices enable alternative features. Even though we cannot cover all distributions of devices, we define two user populations which differentiate between the used technologies. This duality should provide insights into potential performance differences.

4.5.3 Instrumentation

The frontend and backend code must be instrumented for measurement reasons. On the one hand, WebSocket messages are extended by a unique identifier. On the other hand, various events must be monitored and logged. However, the additional execution time of those necessities should be minimal as they are attributed to basic I/O operations. Therefore, we do not expect a noticeable impact on the performance metrics.

4.5.4 Resource Limitations

As previously stated, the CPU and memory of the respective system under test is limited. However, in production environment the Collaboration Service may be scaled vertically to a higher resource capacity. Additionally, the client's computer may be equipped with more computing power than we decide for the benchmark. Thus, the performance results may be weaker than in a real-word scenario. Nevertheless, the resources must be bounded to enable a competitive analysis.

Chapter 5

Performance Analysis of ExplorViz's Collaboration Mode

In this chapter, we present the performance analysis of ExplorViz's Collaboration Mode. We begin by demonstrating the preparation for the performance test. Following that, we present the results. Lastly, we engage in a discussion of the results and evaluate them with regard to our goals.

5.1 Preparation

The objective of the performance analysis is twofold, to establish a basis for evaluating future development activities and to gain insights into potential performance bottlenecks. Therefore, we execute the benchmark as defined in Chapter 4. The system under test is the current Collaboration Service and ExplorViz's Frontend [Brück 2023].

To generate representative results, it is essential to limit the resources of the containers that encapsulate the respective system under test. During the backend test, we must restrict the resources as strongly as possible, as we aim to retain the option to investigate the horizontal scalability of the Collaboration Service. However, the resources must be sufficient for the backend to simulate a realistic runtime behavior. The configuration details are presented in Table 5.1 using Kubernetes notation.

Table 5.1. Resource configuration of the Collaboration Service's container.

Type of Resource	Value
CPU	1
Memory	2Gi

For the performance test of the frontend, we need to constrain the Client's container. The Client is responsible for executing ExplorViz's Frontend, including the 3D animation of the landscape in its local web browser. We have determined the necessary resources for this task through exploratory means. The resulting configuration is outlined in Table 5.2.

In Table 5.3, we showcase the final configuration of the test variables, their values being determined based on the considerations in Chapter 4.

Table 5.2. Resource configuration of the Client's container.

Type of Resource	Value
CPU	12
Memory	4Gi

Table 5.3. Test configuration.

Variable	Value
V3: The number of repeated scenarios	50
V4: The duration between user actions	500 milliseconds
V4: The duration between XR position updates	50 milliseconds

Throughout the test execution, we vary the number of users per room (i.e., test variables *V1* and *V2*) for each user population in an exploratory manner. Additionally, during the backend test, we scale the Load Generator to simulate one, two, four, and eight rooms. This allows us to investigate the runtime behavior of the Collaboration Service in managing multiple rooms simultaneously.

5.2 Results

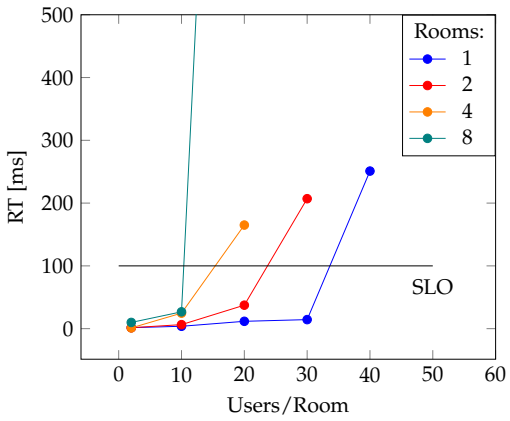
In this section, we present the results of the benchmark execution [Brück 2023]. Firstly, we showcase the results of the backend test for the two distinct user populations, i.e., the Browser Population and the Cross-Platform Population. Following this, we offer the performance results of the frontend setup for both user populations.

5.2.1 Backend

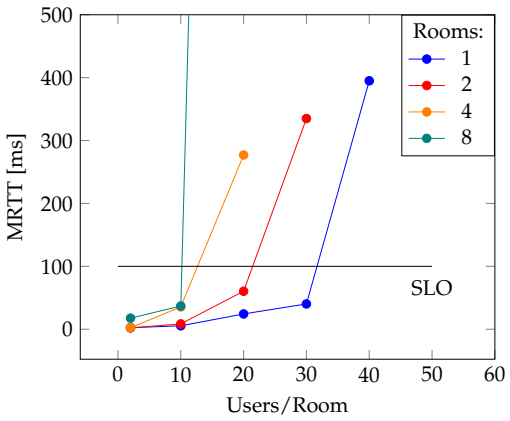
The results of the backend performance test with the Browser Population are depicted in Figure 5.1.

Each diagram corresponds to a different metric and presents curves for room counts of one, two, four, and eight. Specifically, each graph of a function represents a mapping from the number of connected clients to the metric-specific value space. For clarity, we offer the average value of each metric over one test iteration. For instance, each function value of the RT metric is an aggregation of all RTs of Request Messages for one test execution with a constant number of clients and rooms.

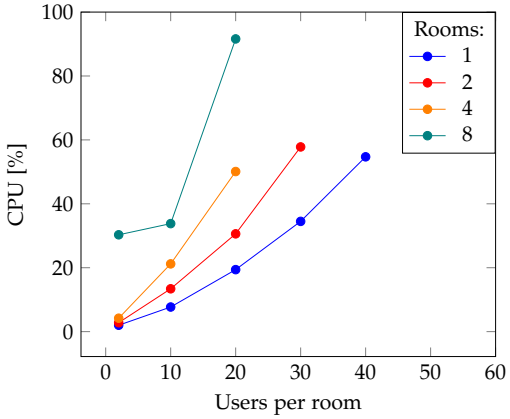
Furthermore, we provide the SLO's threshold for both the RT and MRTT to visualize SLO violations. If the function values are situated below the black-colored curve, the SLO requirement is met. If the graph of the function exceeds the curve, the metric surpasses the SLO's threshold and, consequently, does not meet the requirement. Once the SLO is



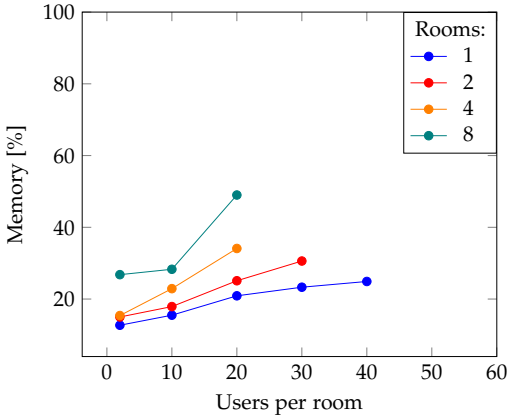
(a) Response time.



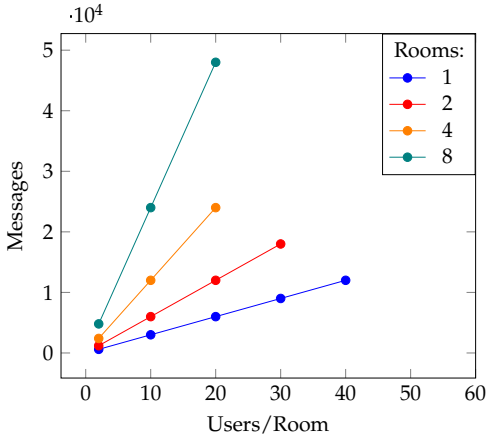
(b) Mean round trip time.



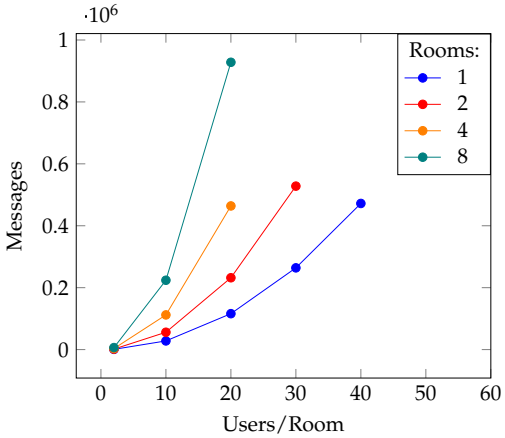
(c) CPU utilization.



(d) Memory utilization.



(e) Number of messages received by server.



(f) Number of messages sent by server.

Figure 5.1. Backend performance results for the Browser Population.

not met for at least one metric, the termination criterion is fulfilled, and the graph of the function is interrupted.

The minimal service time of the Collaboration Service is represented by two clients, as at least two users are required to simulate collaboration, triggering Synchronization Messages. It consistently remains below 20 milliseconds for all room counts and both RT and MRTT. Similarly, the minimal resource utilization is typically below 20%, except for eight rooms, where the values of CPU and memory usage are significantly higher.

Furthermore, the RT and MRTT exhibit very similar behavior as they both progressively increase with a growing workload. The increase is even more pronounced when the backend serves multiple rooms simultaneously. For all room counts, we eventually reach the SLO's threshold. While a single room can accommodate up to 30 active users, eight rooms can only be managed if each room holds around 10 users. In general, the metrics do not grow linearly with the number of clients. The gap between 30 and 40 users in one room is considerably larger than the gap between 20 and 30 users.

The resource utilization increases with intensified workload. While memory utilization never exceeds half of its limit, CPU utilization increases significantly. However, the CPU is not fully occupied even when the RT or MRTT exceeds the SLO's threshold by a significant margin.

Finally, we examine the number of transmitted messages. The number of messages received by the server grows linearly with the number of users. The increase of the functions directly depends on the number of rooms. However, the functions of messages sent by the Collaboration Service display a stronger growth and are not linear at all.

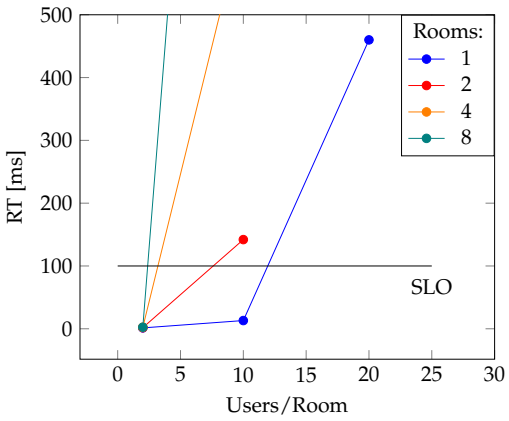
The results of the Cross-Platform Population are visualized in Figure 5.2.

The diagrams follow a structure similar to the previous visualization of the Browser Population. However, the x-axis does not pertain to homogeneous user groups. Instead, the user count aggregates both browser and XR users. For example, the value 30 on the x-axis signifies 15 browser users and 15 XR users, as defined by the Cross-Platform Population. Therefore, the minimal service time is represented by the RT and MRTT while the Collaboration Service hosts two client sessions, each with a different device.

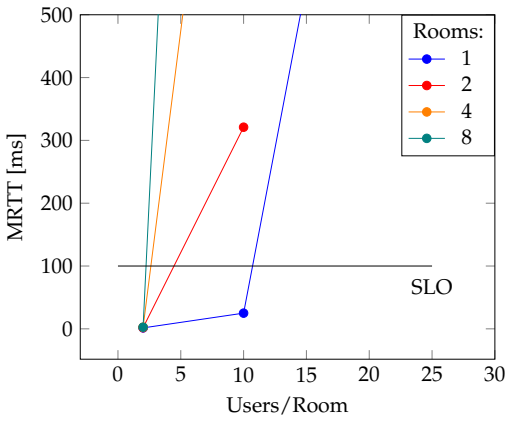
Overall, all metrics exhibit a strong tendency to increase as the number of users or rooms grows. Additionally, the RT and MRTT violate the SLO requirement very quickly. It appears that the backend can handle 10 interacting users, but only if they represent the only room. The graphs that pertain to multiple rooms increase drastically with a growing interaction per room.

The CPU utilization reaches values around 80% for many workload configurations. The memory never occupies more than 60% of the limit. However, the initial utilization with the minimal user count of two increases with a higher number of rooms.

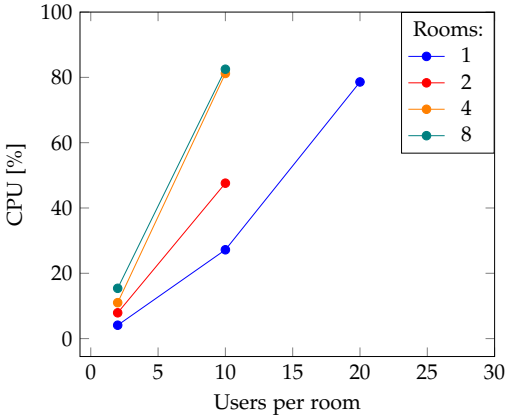
Finally, the number of transmitted messages follows the pattern of the other metrics. In the case where the server provides eight rooms in parallel, the message count grows significantly stronger than for other configurations.



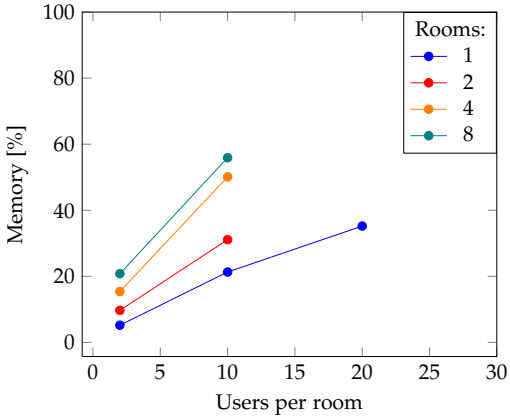
(a) Response time.



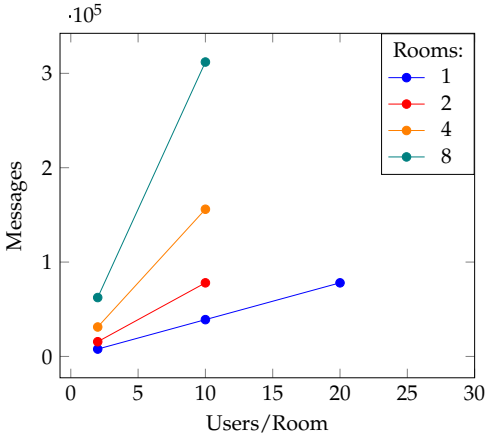
(b) Mean round trip time.



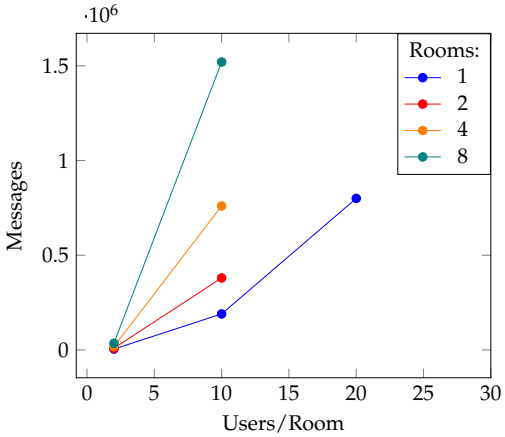
(c) CPU utilization.



(d) Memory utilization.



(e) Number of messages received by server.

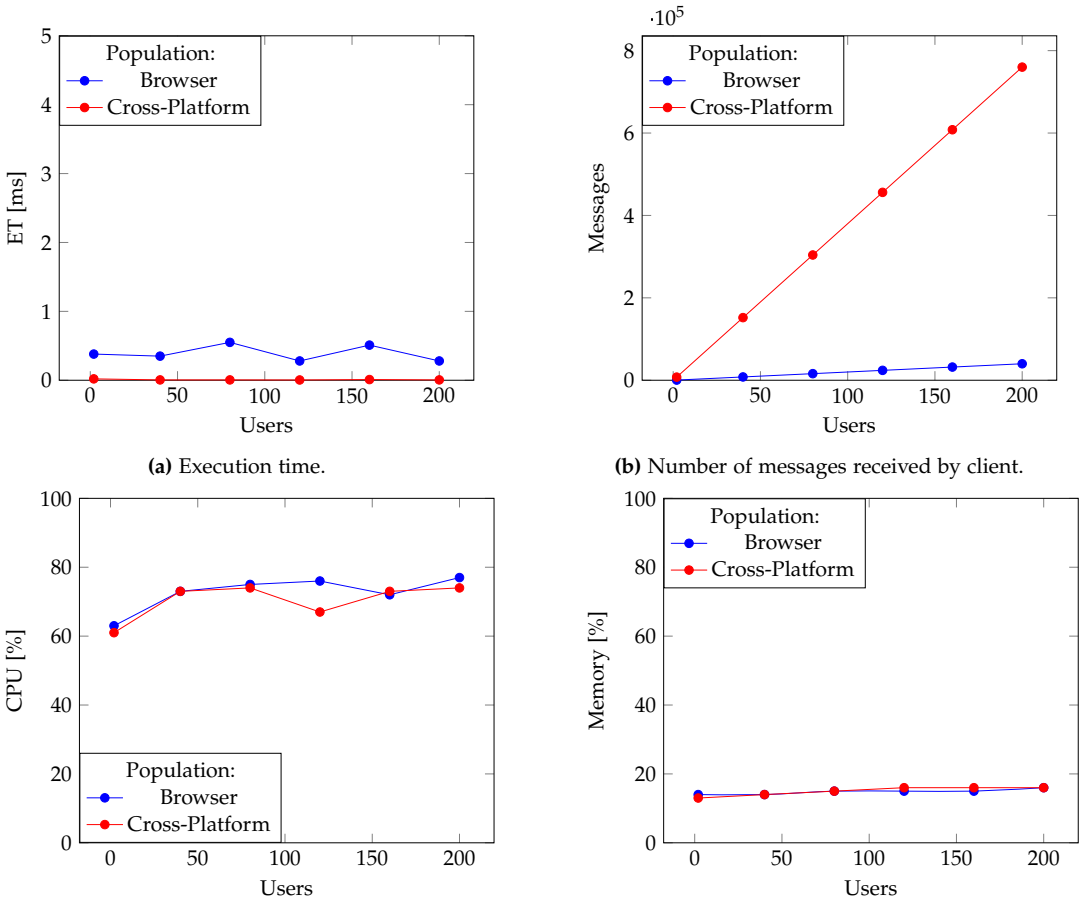


(f) Number of messages sent by server.

Figure 5.2. Backend performance results for the Cross-Platform Population.

5.2.2 Frontend

The results of the frontend test are illustrated in Figure 5.3. These diagrams depict the runtime behavior of two distinct populations: the Browser Population and the Cross-Platform Population, distinguished by color. Additionally, the curves for the metrics ET, CPU utilization, and memory utilization represent the averages of all values for each configuration. Furthermore, it is important to note that we employ a single Load Generator instance for all iterations, since the Client instance responsible for running ExplorViz's Frontend in the browser is assigned to a specific room, and is not associated with clients from other rooms.



(a) Execution time.

(b) Number of messages received by client.

(c) CPU utilization.

(d) Memory utilization.

Figure 5.3. Frontend performance results.

In general, the function values of ET, CPU, and memory exhibit minimal variation,

and high workloads do not substantially impact these metrics. The average ET consistently remains below 1 millisecond. Notably, the handling of messages from XR users is exceptionally swift, with execution time changes occurring on a microsecond scale.

Resource utilization does increase when handling 200 users compared to two users. However, this growth is relatively small, and there appears to be a consistently high, constant resource usage overall, with CPU utilization consistently exceeding 60% and memory utilization ranging between 10% and 20%.

Nevertheless, the graphs tracking the number of received messages exhibit different behaviors concerning user population. While both populations show a linear growth in message counts, the Cross-Platform Population's function experiences a considerably steeper incline.

Finally, we terminated the test after reaching 200 users. This decision was made because the metrics remained relatively stable across multiple iterations, and the load generator is not designed for vertical scaling to significantly increase the load in one room.

5.3 Discussion

Based on the results, the Collaboration Mode supports multi-user sessions with up to 30 browser users within a single room. However, performance degrades noticeably if the user count exceeds 30, potentially hindering real-time interaction. Even a slight adjustment to the SLO's threshold would still breach these standards, as both RT and MRTT escalate significantly with higher workloads.

In a cross-platform environment, the backend effectively synchronizes sessions for a maximum of 10 diverse users. In contrast, the frontend seamlessly handles 200 users without performance degradation. Consequently, we contend that the backend poses a bottleneck for overall performance, particularly with larger room sizes.

Additionally, while the backend can manage multiple rooms with interacting browser users concurrently, the maximum room capacity diminishes notably as the number of rooms increases. In the context of the Cross-Platform Population, the Collaboration Service struggles to efficiently handle multiple rooms.

Analysis of hardware resource metrics demonstrates a direct correlation between resource utilization and client workload. Specifically, the CPU experiences high activity during periods of intense user interaction. As a result, we hypothesize that redistributing the workload across additional physical resources could potentially enhance performance metrics. However, it is important to note that the CPU is not consistently operating at full capacity in configurations where the SLO requirement is not met. Consequently, we conclude that other factors, such as network issues, suboptimal multithreading, or blocking operations, are contributing to performance limitations.

In summary, the minimal service time (i.e., the RT and MRTT for two users) complies with the SLO in every configuration and is significantly shorter than the specified threshold. The message exchange via the Collaboration Service appears suitable for fast client synchro-

nization. However, as the number of client connections increases, performance experiences a significant decline. To comprehend this growth, we investigate the number of transmitted messages. For every Request Message sent by a client to the backend, the server responds with exactly one unicast message. However, in the case of a Synchronization Message, the server broadcasts the message to all other connected users in the same room. Consequently, adding one more user to a room leads to one more sender, triggering broadcast messages, and one more receiver for each broadcast from other clients. Mathematically, the number of Synchronization Messages transmitted by the server in one room is represented by the following function:

$$f(n, m) = m * n * (n - 1) \tag{5.1}$$

Where:

- n is the number of users in the room.
- m denotes the number of Synchronization Messages sent by each user (assumed to be constant for all users).

Thus, the message count grows quadratically with the number of users. This growth of messages has an even more pronounced impact on a cross-platform population, as the number of triggered Synchronization Messages is higher. This leads us to hypothesize that broadcasting Synchronization Messages constitutes a performance bottleneck. Reducing the number of outgoing messages on the server side could potentially enhance the Collaboration Mode's ability to serve larger room sizes.

Furthermore, the results of the frontend test demonstrate a consistent effort in processing collaborative sessions for various room sizes. We conclude that rendering the 3-dimensional software landscape in the browser demands a significant amount of scripting time and resources. However, WebSocket messages from the Collaboration Service are processed very efficiently and do not impact performance as strongly as we initially assumed.

Chapter 6

Re-Engineering of ExplorViz's Collaboration Mode

In this chapter, we present the re-engineering process of the Collaboration Mode in ExplorViz.

6.1 Methodology

The goal of the re-engineering is to enhance the performance of the Collaboration Mode, facilitating multi-user sessions with extensive user interaction. We will adhere to the software re-engineering framework outlined by Majthoub et al. [2018] to ensure a reliable and structured re-engineering process. This framework outlines five key phases: *Reverse Engineering*, *Requirements Engineering*, *Re-Design*, *Re-Implementation*, and *Testing*.

To begin, we will conduct a thorough reverse engineering analysis of the components and features of the previous Collaboration Mode. Following this, we will define the re-engineering system's requirements in alignment with our objective. Subsequently, we will proceed to design, implement, and conduct testing on the Collaboration Mode. However, it is important to note that the Testing phase primarily focuses on verifying the system's correctness, while performance testing will be addressed in the Chapter 7.

In general, we will employ the *Big Bang* approach [Majthoub et al. 2018], wherein the entire system is replaced. This approach is necessary as the Collaboration Service is not inherently scalable, as discussed in detail in the subsequent section.

6.2 Reverse Engineering

To re-engineer the current Collaboration Mode, we consult the work from Bader [2022] and review the codebase of the project[1]. Initially, we model the primary components of the Collaboration Mode and describe their functionality. Afterward, we proceed to address the implementation.

[1] https://github.com/ExplorViz/collaboration-service

6.2.1 Components and Responsibilities

The main components of the existing Collaboration Mode are visualized in Figure 6.1.

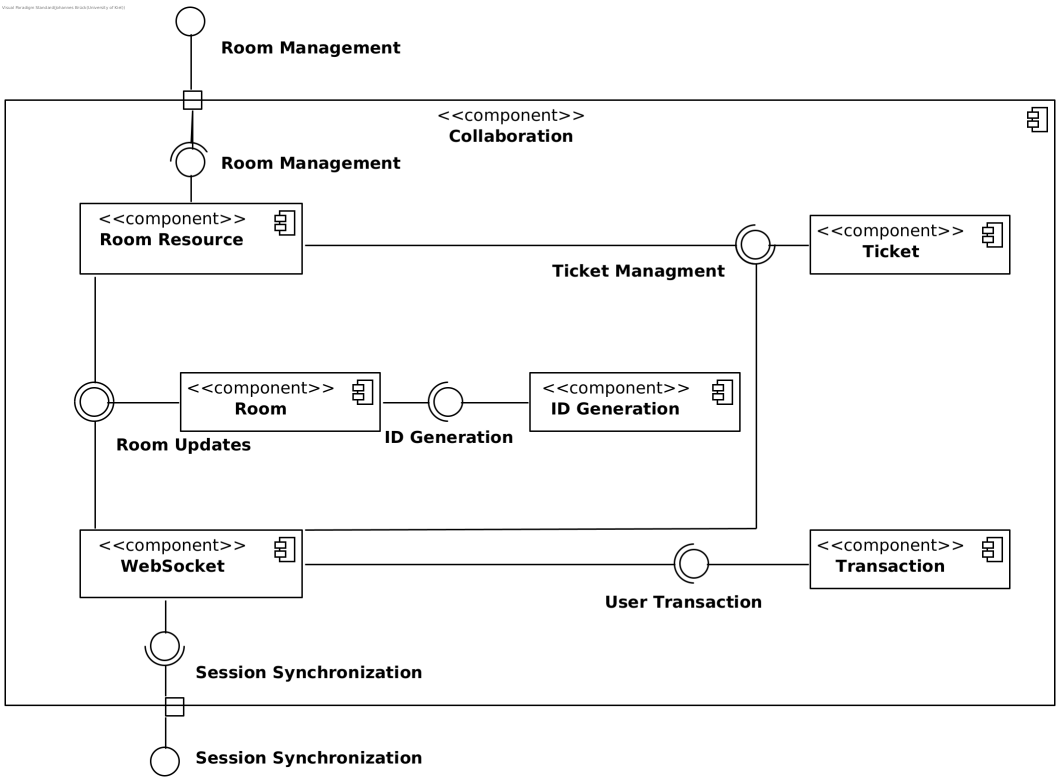


Figure 6.1. Main components of the reverse engineered Collaboration Mode.

The core responsibilities of the Collaboration Mode include *room management*, *client synchronization*, and *mutual exclusion*. In the subsequent sections, we will detail these responsibilities in relation to the main components.

Room Management

Every collaborative session is linked to a specific room. Each room operates independently, entirely separate from others, and every client's session is tied to exactly one room. Clients have the option to access a list of rooms, create a new room, or join an existing room through the *Room Resource* component. During the creation of a room, a new room model is initialized via the *Room* component. This room model encompasses data regarding the virtual scene, including the state and position of landscape components and users.

When a client wishes to join a room, they obtain a *ticket* from the *Ticket* component. This ticket acts as an association between the user and the room, permitting the respective user to join the room within a specific timeframe. Subsequently, the client establishes a connection with the WebSocket component and redeems the ticket.

Client Synchronization

All room events are updated within the Room component and then propagated to the respective clients. These room events can encompass various types of information, such as status updates, for instance, when a user leaves the room, or collaborative user actions, such as when a user highlights a component. Each user action was previously transmitted to the *WebSocket* component by the client, leading to an update of the Room component. The room model serves as the single source of truth, enabling new users to initialize their scene and allowing other users to validate their own actions. Each element within the underlying data model is associated with a unique ID for every user to distinguish it. In most cases, these IDs must be generated dynamically at runtime using the *ID Generation* component, including IDs for new users or newly opened applications.

Mutual Exclusion

In general, two types of collaborative user actions are observed. On one hand, a user can initiate a collaborative event, such as highlighting a landscape component. These events trigger local model updates and are then propagated through the network. On the other hand, certain user actions require validation by the Collaboration Service, necessitating mutual exclusion among the clients. The primary focus of these actions is on the functionality of *grabbing* objects: XR users can grab most of the objects within the room, enabling them to reposition these objects.

For clarity, mutual exclusion refers to the challenge in concurrent systems of serializing concurrent access to shared resources, ensuring that actions performed on these shared resources are *atomic* [Singhal 1989]. In our specific use case, these shared resources are represented by grabbable objects. Interactions with grabbable objects must be atomic, implying that if any XR user has already grabbed an object, no other user is permitted to grab, move, or manipulate that particular object.

The serialization of user actions is managed by the *Transaction* component, which maintains the state of all objects and restricts access for other clients when these objects are in use.

6.2.2 Implementation Overview

The current state of the Collaboration Mode is integrated within ExplorViz's microservice architecture, as depicted in Figure 3.5. It incorporates the Collaboration Service as an additional component to the existing services. Communication between clients and the

Collaboration Service is facilitated through an extension of ExplorViz's Frontend. This frontend extension accesses rooms via HTTP requests to the Collaboration Service's API and exchanges room updates using the WebSocket protocol. The implementation is realized through an Ember.js add-on.

Update messages are encapsulated in *JSON* format [Pezoa et al. 2016] and are identifiable by each party through an event description. Browser users can send only a subset of the messages that XR users can send, as XR users possess additional features such as visualizing user avatars. However, all clients receive the same set of messages, even if certain features associated with the messages are not supported.

The Collaboration Service operates as a Quarkus application. It manages all data in-memory, without utilizing a persistence layer to store data in a database. Furthermore, the Collaboration Service does not support horizontal scalability, as it lacks inter-server communication. Consequently, attempting to scale out to multiple instances would result in independent deployments, unable to effectively manage the same rooms collectively.

6.3 Requirement Engineering

Bourque and Fairley [2014] identify two categories of system requirements, *functional* and *non-functional* requirements. We will present our requirements of both categories in the following sections.

6.3.1 Functional Requirements

From a functional perspective, the re-engineered system should aim to retain the same set of user features offered by the current Collaboration Mode. Drawing insights from the documentation provided by Bader [2022], we outline the following *user stories* to serve as a guide for development:

US1: As a user I want to view a list of all active collaborative sessions.
US2: As a user I want to create a room to share my current session with other users.
US3: As a user I want to join rooms to explore landscapes collaboratively.
US4: As a user I want to highlight components to color them for all users.
US5: As a user I want to open components/applications to gain a lower-level view for all users.
US6: As an XR user I want to visualize avatars of all XR users.
US7: As a user I want to share pop-ups to provide detailed information to all users.
US8: As a user I want to switch the timestamp of the landscape synchronously with all users.
US9: As a user I want to share my heatmap configuration with all users.
US10: As a user I want to ping to a component to show them to all users.
US11: As a user I want to close components/applications to gain a higher-level view for all users.

US12: As a user I want to close pop-ups to clean the virtual room for all users.
US13: As an XR user I want to grab objects to move them in the virtual room for all XR users.

However, the user stories are very high-level and lack specifics regarding the user interface. We will bridge this gap by referencing the existing code base during the development process.

6.3.2 Non-Functional Requirements

In the following section, we present the non-functional requirements addressing *interoperability*, *performance*, *scalability*, *reliability*, and *maintainability*.

Interoperability

The re-engineered Collaboration Mode should seamlessly integrate into the current microservice architecture of ExplorViz. To achieve this, we will develop a Collaboration Service that fulfills all identified tasks in the reverse engineering process: room managment, client synchronization, and mutual exclusion. We are also bound by technical constraints from ExplorViz's Frontend. On one hand, the frontend part of the Collaboration Mode needs to integrate with the core frontend. On the other hand, it must align with the external interface of the Collaboration Service.

Performance

Our primary objective is to enhance the performance of the Collaboration Mode. We have already conducted a performance analysis of the current implementation, identifying limitations and performance bottlenecks. Our aim is to push these boundaries and resolve the identified bottlenecks. However, we have not set specific quantifiable performance goals. Instead, we aspire to create a performance-efficient system that delivers stable performance even under high user interaction, leading us to the next requirement.

Scalability

To meet our performance requirements, especially during high user load, we aim to design the system to be horizontally scalable. In contrast to the current system, the re-engineered Collaboration Service should distribute the client load from multi-user sessions across multiple instances.

Reliability

Ensuring a reliable connection between clients and the Collaboration Service is vital, even in a scaled environment.

Maintainability

ExplorViz operates as an agile project, incrementally developed by students and researchers [Zirkelbach et al. 2018]. Consequently, all software artifacts must be well-structured and easily comprehensible. Moreover, the architecture should be sustainable and readily extensible to accommodate additional features.

6.4 Re-Design

In the following section, we will redesign ExplorViz's Collaboration Mode in accordance with the requirements previously defined.

6.4.1 Communication Pattern

To meet the scalability requirement, we must address two key challenges: *inter-server communication* and *load balancing*.

In terms of inter-server communication, there are three primary reasons for the service instances to communicate. Firstly, the Collaboration Service maintains in-memory model data of the virtual scene. When multiple instances are deployed, each instance's model data must be complete to ensure correct service, and data replicas must remain consistent to prevent ambiguity. Hence, all model updates must be communicated between the instances. The second reason is that clients within the same room might be connected to different service instances. Consequently, collaborative user events must be exchanged between the instances to propagate these events across the network. The final reason relates to mutual exclusion. In the absence of a single decision-maker, the instances must agree on the order of atomic events.

The second challenge revolves around load balancing. Distributing the client load across service instances is crucial to fully utilize additional resources. However, WebSockets require a stable connection between two endpoints and cannot be balanced easily between different service instances on demand.

From an abstract perspective, Alexeev et al. [2019] address a similar problem in the context of browser-based grid computing with WebSocket workers. They suggest using the Publish/Subscribe pattern for inter-server communication. This pattern has the advantage that servers do not need to know each other and do not block each other for message exchange [Curry 2004]. Moreover, Publish/Subscribe serves as a viable substitute for direct messaging, enhancing reliability by mitigating the potential for message loss [Ganaputra and Pardamean 2015].

Furthermore, Alexeev et al. [2019] introduce *sticky* load balancing, wherein every message from the same client is always forwarded to the exact same server instance, ensuring intelligent load distribution during WebSocket connection establishment.

Based on these considerations, we present our intended server architecture in Figure 6.2.

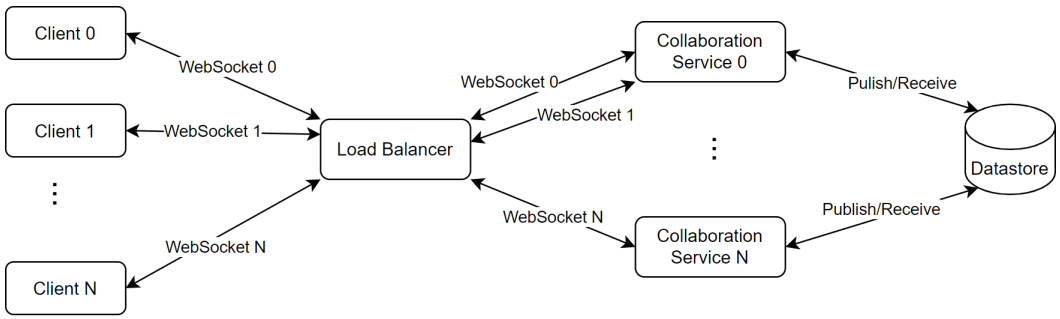


Figure 6.2. Communication pattern of the scalable Collaboration Mode

The architecture implements the Publish/Subscribe pattern, wherein each instance can publish and subscribe to events. There is no direct communication between clients. All published events are stored in a central datastore and delivered to the subscribed instances. Additionally, we provide a load balancer. New client connections are distributed among the server instances, while established WebSocket connections remain sticky. Thus, the load balancer doesn't balance the number of messages among the servers, but it balances the number of clients. This way, each server is responsible for a specific set of clients.

The communication pattern operates as follows. Assuming *Client 0*, *Client 1*, and *Client N* (refer to Figure 6.2) are all in the same room, and Client 0 triggers an update, e.g., moving its position in the virtual scene, both Client 1 and Client N need to be informed. Since Client 0 is connected to *Collaboration Service 0*, it is the only server that knows about the update. Client 1 can be informed directly as it is also connected to Collaboration Service 0. However, Client N is connected to Collaboration Service N. Therefore, Collaboration Service 0 must publish an event, which is subscribed to by *Collaboration Service N*. Finally, Collaboration Service N receives the event, updates its own model, and informs Client N.

6.4.2 Components

In general, we will retain all the components that constitute the previous Collaboration Mode (see Figure 6.1). This approach enables us to seamlessly integrate the basic functionality into our new architecture. However, we will introduce several components primarily responsible for inter-server communication and adjust the dependencies between persistent components accordingly.

The main components of the redesigned Collaboration Mode are visualized in Figure 6.3.

All inter-server communication is managed by the *PubSub* component, serving as an interface for other components to push or fetch data from the shared datastore. Communication events are triggered or processed by various components based on the context of the respective event, such as the *Publisher*, *Subscriber*, Ticket, Transaction, or ID Generation component. We distribute these responsibilities among these components to ensure that

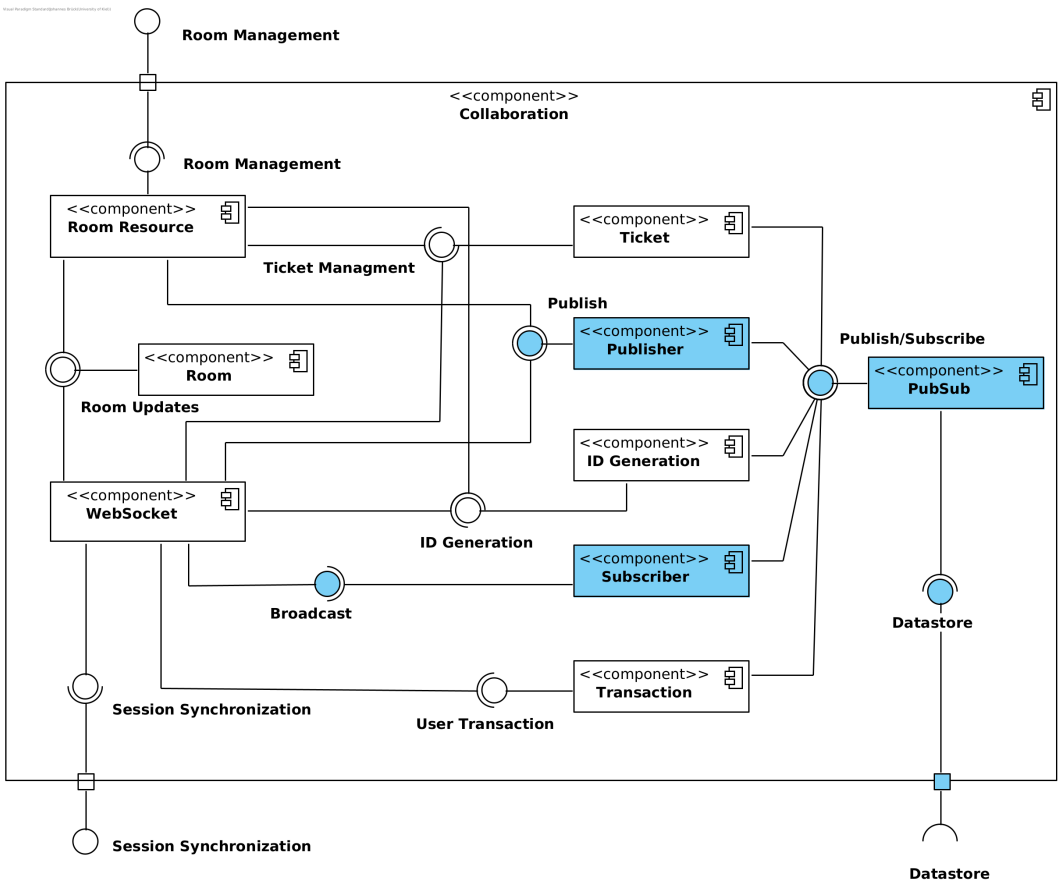


Figure 6.3. Main components of the re-designed Collaboration Mode. The new elements with respect to Figure 6.1 are colored blue.

the application logic remains modular and easily understandable. Moreover, this approach helps prevent a circular dependency between the WebSocket and the PubSub component, which could result from bidirectional communication flow. Such a circular dependency is considered an antipattern due to its adverse impact on maintainability [Oyetoyan et al. 2015].

In the following section, we detail how these components handle the core responsibilities of the Collaboration Mode identified during reverse engineering, namely room management, client synchronization, and mutual exclusion.

Room Management

When a client wishes to create a new room, the Room Resource component publishes the initialization data via the Publisher to the other server instances. The Subscriber receives the data and creates a local room data model. If the client retrieves a list of existing rooms from any server instance, the newly created room should be part of that list. When a client joins a room, a ticket is obtained from the Ticket component, automatically triggering the publication of the Ticket. If the client establishes a WebSocket connection to any server in the next step, the Ticket is redeemed, meaning it is validated by looking up the ticket in the shared datastore.

Client Synchronization

All room events are initially published in the shared datastore via the Publisher. Subsequently, these events are received by the Subscriber component for the server to process the event within its local subnetwork. The Subscriber updates the local model using the Room component and prompts the WebSocket component to forward the event to the subset of connected clients. If a user event expands the model by introducing a new unit, the event is enriched with a unique ID before being published to the datastore. This ensures that all model replicas have the same ID. As we must generate a unique ID on any server, we utilize the datastore for this purpose. The ID Generation component generates IDs locally, ensuring they are human-readable since certain IDs appear in the UI. It also keeps track of already assigned IDs in the shared datastore.

Mutual Exclusion

In the case of a user event that requires mutual exclusion, the Transaction component secures the relevant resources by publishing a corresponding notice in the datastore. Conversely, the Transaction component can query the datastore to check for previously blocked objects, identifying them using their unique IDs, and reject a user request if the requested object is already in use. For the later implementation, it is crucial that the permission process remains entirely transactional and is not at risk of a race condition.

6.4.3 Consistency Model

In general, we implement *eventual consistency*, meaning that individual servers may retain data that lags behind, but in the absence of updates, all instances eventually converge to the same state [Vogels 2009]. We chose this relaxed consistency model primarily due to our need for high availability. Given that high performance is a core requirement, we must minimize both network traffic and the blocking time of server instances. We contribute to the accuracy of data replicas by implementing connection protocols that incorporate TCP retransmission. Additionally, we handle atomic actions in a distributed manner to ensure that conflicting events are resolved consistently across all servers.

On the client side, we ensure *read-your-writes* consistency [Vogels 2009]. Every update resulting from a user action is initially executed locally on the client-side model and is then forwarded to the connected server instance. Therefore, every user always sees the result of previous actions. However, inconsistencies between individual users in a session may be perceived if the forwarded update experiences delays either between a client and a server or between servers. This risk materializes if the response times of the concerned nodes become excessively high. Thus, minimizing this risk aligns with our objective of providing high performance by keeping RT and MRTT low.

6.4.4 Messaging Optimization

Finally, we will optimize the WebSocket traffic. In the previous chapter, it was highlighted that performance strongly correlates with the number of messages in the network. An increasing number of client connections results in a significant rise in outgoing messages on the server-side. XR users in particular trigger a flurry of messages due to regular position updates. Consequently, we aim to minimize the negative performance impact of XR clients by selectively transmitting XR-related messages only to XR users. Practically, we will implement *platform-specific messaging*. While this approach contradicts the platform abstraction as intended by Bader [2022], we aim to maintain platform-independent messaging as the default case and introduce special cases for optimization.

6.5 Re-Implementation

In the following, we delve into a detailed description of the implementation of the redesigned Collaboration Mode. We begin by presenting the programming frameworks. Subsequently, we explain how we implement the application logic. Following that, we optimize the network traffic by introducing platform-specific messaging. The final sections address implementation details regarding the integration of Socket.IO and Redis, as well as load balancing.

6.5.1 Frameworks

On the backend side of the Collaboration Mode, we completely reimplement the functionality by replacing the current code artifacts of the Collaboration Service with an entirely new code base.

We choose NestJS as the core framework for the Collaboration Service. NestJS offers high scalability [Pham 2020], aligning well with our requirements. Moreover, NestJS provides a robust architecture while being highly opinionated [Pham 2020], meaning the framework inherently guides developers towards NestJS-specific best practices. One of these best practices is integrating a service layer with dependency injection, which supports extensibility, testability, and reusability [Yang et al. 2008]. Furthermore, projects that utilize

dependency injection tend to exhibit lower coupling between modules [Razina and Janzen 2007]. Additionally, NestJS is based on *Node.js*[2], which is known for its high throughput and scalability of web applications due to its event-driven architecture and asynchronous, non-blocking I/O operations [Tilkov and Vinoski 2010]. The primary programming language for the source code is *TypeScript* [Bierman et al. 2014], aligning with the codebase of ExplorViz's Frontend. TypeScript's type system aims to establish a foundation for maintainability.

For client-server communication, we will once again employ the WebSocket protocol. However, we manage the WebSocket connections using the Socket.IO framework. Socket.IO inherently supports room-specific broadcasting by default, aligning well with the room concept of the Collaboration Mode. Additionally, Socket.IO's disconnection detection and reconnection automatism sustain the reliability of the client-server communication.

Finally, we utilize Redis as a remote datastore for inter-server communication. Redis not only aligns with the Publish/Subscribe pattern but also provides high performance for operations and good memory efficiency, even when compared to other in-memory databases [Kabakus and Kara 2017].

On the frontend side, we maintain the current code base and adapt it to meet our new requirements. In particular, we do not alter the manipulation of the frontend-local model due to user actions, as this approach proved to be highly efficient. However, we need to migrate the WebSocket implementation to the Socket.IO framework, similar to what we do on the backend.

6.5.2 Application Logic

In the following, we explain how we implement the application logic. However, we do not delve into every user story individually, as the abstract logic for those is already outlined in [Bader 2022]. Furthermore, we do not showcase the evolution of the frontend and backend separately. Instead, we focus on how we implement the logic of the core responsibilities and the main components in a scaled environment.

Room Management

Room management encompasses the creation, joining, and fetching of rooms. In the following, we consider a scenario where an arbitrary client intends to create a new room and then join it (see Figure 6.4).

At first, the frontend sends an HTTP request to the provided API of the Collaboration Service, facilitated by the *AppController* class, which handles endpoints for creating, joining, and fetching rooms. The HTTP payload contains all the necessary information to initialize the landscape on the client-side. Prior to room creation, the AppController must assign a new unique room ID for all nodes to identify the room. Therefore, the AppController calls the injected *IdGenerationService*. For clarity, we present the ID generation process in a separate diagram (see Figure 6.5).

[2]https://nodejs.org/en

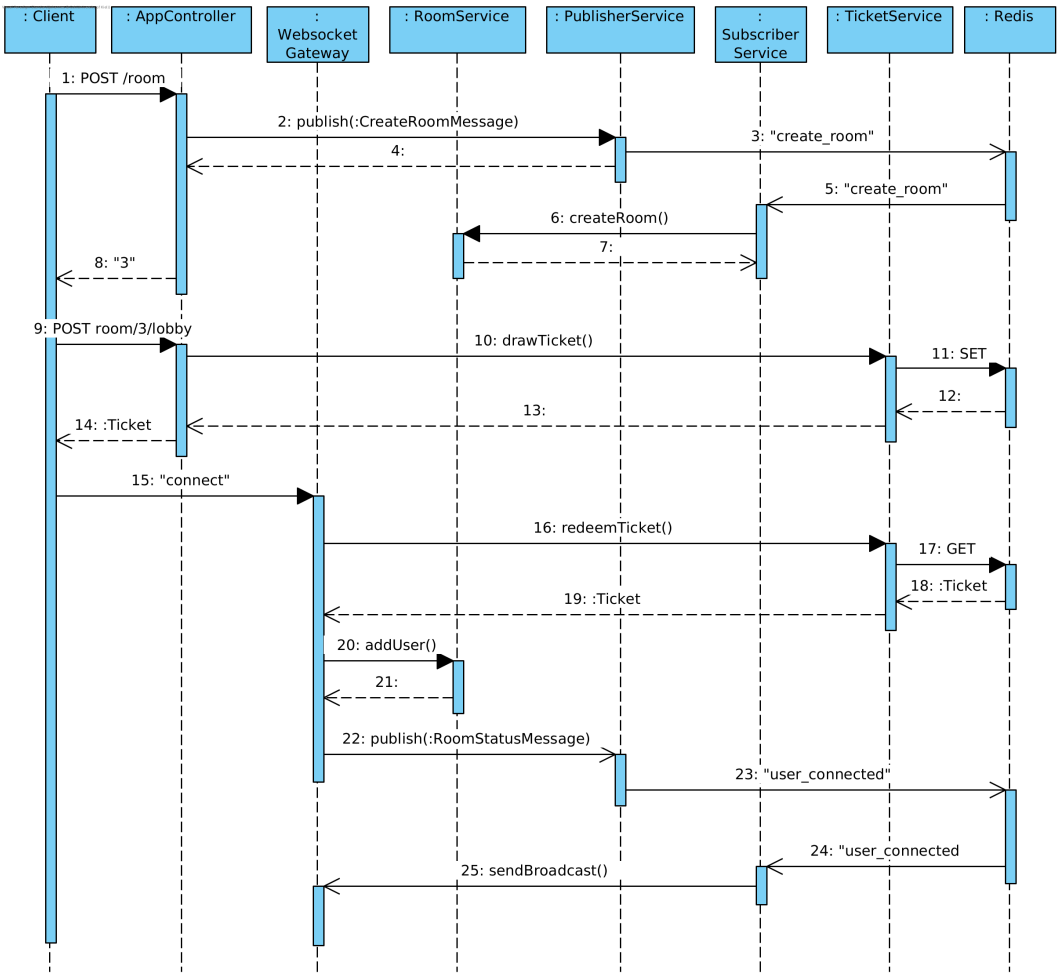


Figure 6.4. A client creates a new room and joins it.

The IdGenerationService generates the ID from the Redis datastore, which is utilized here as a shared *key-value storage*. The key, representing the location of the next ID value, is hardcoded identically for all instances. Initially, the IdGeneration Service initializes the storage if it has not already been declared, using the transactional Redis function *SETNX* to ensure that the value is not null before usage. In the next step, the current value is both incremented and fetched using the transactional Redis function *INCR*, ensuring the value is not read again by another process.

With the room ID set to "3", a new *CreateRoomMessage* is published, encapsulating the event of room creation with all necessary information, including the room's ID (see

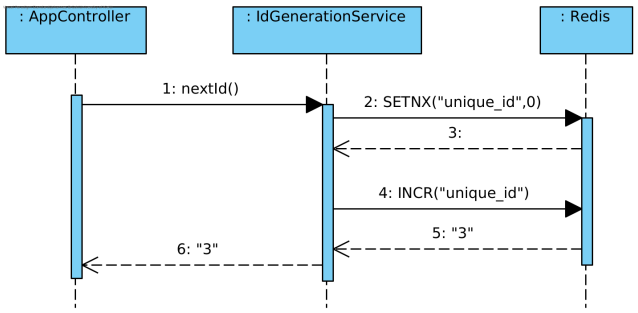


Figure 6.5. Distributed generation of a unique ID via Redis's key-value storage.

Figure 6.4). When the *SubscriberService* class receives the message, it creates the room by initializing a new local model via the *RoomService* class. The RoomService is responsible for managing and modifying all server-local room models and represents the Room component (see Figure 6.3).

If the client wishes to join the previously created room, the corresponding HTTP request triggers the ticketing process. The *TicketService* generates a new Ticket, including a ticket ID, a user ID, and an expiration date. The ticket ID is obtained not from the IdGenerationService, but from a *UUID* module, ensuring the ID is unguessable by clients [Leach et al. 2005]. Subsequently, the ticket is stored in the key-value storage of Redis. Using the ticket ID, the client can authenticate itself when establishing a WebSocket connection. The Socket.IO server is integrated with the *WebsocketGateway* class, which realizes the WebSocket component (see Figure 6.3). It verifies the validity of the provided ticket ID by looking it up in the key-value storage. If the ticket is valid and not expired, the connection is established successfully. In the next step, a user model is created locally, and the WebSocket session is registered in Socket.IO's room "room-3," taking the session into account for room-specific broadcasts. Additionally, a *RoomStatusMessage* is published, informing other nodes about the new client connection.

In summary, the distributed room management, including the API as facilitated by the AppController, covers the user stories *US1* to *US3*.

Client Synchronization

Client Synchronization refers to synchronization messages triggered by basic user actions. In the following, we consider a scenario where a user with the ID "2" in the room with the ID "3" has just highlighted a component with the ID "30", which belongs to the application with the ID "5" (see Figure 6.6).

In this case, the Socket.IO client triggers an update message to the Socket.IO server. The context of the message is determined by the Socket.IO server through channel naming, i.e., the message is transmitted via a virtual channel whose name describes the category of

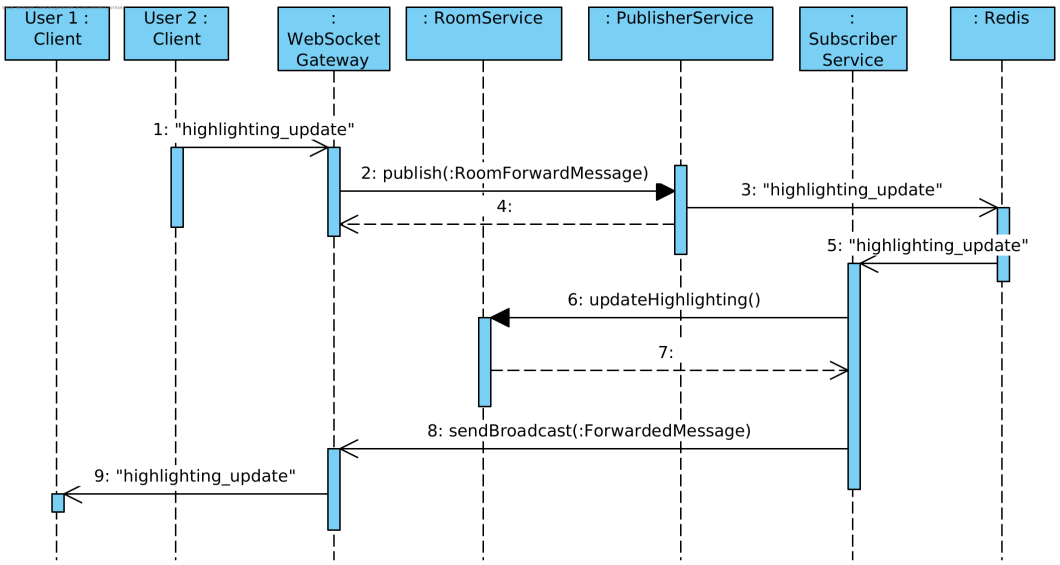


Figure 6.6. A client highlights a component.

the message. In this particular instance, the channel is named "highlighting-update". The WebsocketGateway implements a channel-specific handler method. However, it does not directly modify the server-local model, but publishes the message via the *PublisherService*. Published synchronization messages are encapsulated in a *RoomForwardMessage*, including the room and user IDs.

Analogously to Socket.IO, Redis provides channels to differentiate message streams. The SubscriberService class listens to the respective channel and implements a handler method. This handler method updates the server-local model of room "3", marking the component with ID "30" in application "5" as highlighted by user "2". It then broadcasts the synchronization message to all connected clients in room "3" by encapsulating it in a *ForwardedMessage* with the user's ID. The broadcast can easily be initiated by transmitting the message to Socket.IO's room with identifier "room-3". In case the server has an active connection to user "2", the broadcast excludes the respective connection, as user "2" is the creator of the event and has already processed it locally.

The event of highlighting was provided as an example for client synchronization. The user stories *US4* to *US10* are implemented similarly. However, they use different communication channels and result in diverse model updates.

Mutual Exclusion

Mutual exclusion is necessary when a user attempts an action that requires validation for atomic processing. For instance, consider the scenario where user "2" in room "3" intends

to grab the object with the ID "45" to move its position (see Figure 6.7).

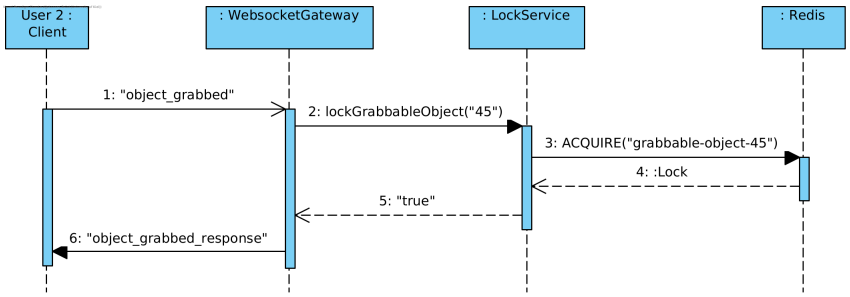


Figure 6.7. A client grabs an object.

Firstly, a request message is transmitted via the "object_grabbed" channel. The WebsocketGateway then calls the LockService to block object “45”. The *LockService* represents the Transaction component (see Figure 6.3) and is responsible for resource *locking*, i.e., single-user blocking of virtual objects [Hastings 1990]. However, in a decentralized architecture, all instances must reach a consensus regarding the blocking user. Therefore, we implement a *distributed lock* mechanism [Hastings 1990] with *Redlock*[3]. Redlock is a distributed lock manager that can lock resources using the Redis environment. Redlock’s algorithm attempts to acquire a lock for a unique resource and resolves conflicts based on timeliness. The LockService initiates the locking process with a configurable timeout and retry count to keep the blocking time low. The locked resource is the ID of the grabbed object, i.e., “45”. Subsequently, all user actions concerning object “45” will be attributed to user “2”. The lock is maintained indefinitely until the user releases object “45” or disconnects. In the event of an unexpected disconnection, Socket.IO recognizes the absence of user “2” as well. Thus, the mechanism is fail-safe in terms of client nodes.

Closing objects in the virtual room, i.e., components, applications, or menus, must also be handled atomically since they may already be grabbed by another user. Therefore, user stories *US11* to *US13* must all be implemented in a similar manner.

In summary, the Redis instance serves various purposes. On one hand, it acts as a distributed event stream for room events, i.e., RoomStatusMessages, and user actions, i.e., RoomForwardMessages. On the other hand, it acts as a key-value storage for tickets and unique IDs. Finally, it acts as a single point of truth for distributed locks.

6.5.3 Platform-specific Messaging

As stated in the re-design of the Collaboration Mode, we aim to minimize the flood of XR-related messages. To achieve this, we register two distinct Socket.IO broadcasting rooms for each virtual room: the default room “room-<ID>” and the XR room “room-<ID>-xr”.

[3]https://redis.com/glossary/redlock/

The default room encompasses all device categories, namely browsers, VR, and AR clients. On the other hand, the XR room includes only the subset of users using XR devices. Consequently, regular XR positional updates, which may be triggered more than 20 times per second for every XR user, are sent only to those clients that actually process the information.

However, the server is not aware of the platforms its clients are using. To address this, we incorporate a platform registration process. The platform type can be determined by the frontend application by querying the current visualization mode and encoded in a simple string notation, such as "browser", "ar", or "vr". Initially, the platform type is determined and transmitted to the server when establishing the WebSocket connection. In case the client switches the visualization mode during runtime, we integrate a *listener* in the frontend that promptly informs the server via a WebSocket message. In general, the platform differentiation can be easily extended for further optimizations.

6.5.4 WebSocket Client Migration

Socket.IO is the framework of our choice for managing bidirectional WebSocket messages between the client and the Collaboration Service. Consequently, we structure the Collaboration Service around a Socket.IO server, making use of all Socket.IO-specific features. However, the previous frontend utilized the standard WebSocket Web API, which is not compatible with Socket.IO. Thus, we need to migrate the frontend's WebSocket implementation to Socket.IO's client interface.

The former frontend provided WebSocket messaging to other application components through an injectable Ember.js service known as *WebSocketService*. We intend to retain the WebSocketService while modifying the implementation, as explained below.

The updated WebSocketService implementation encapsulates a nullable Socket.IO client, allowing WebSocket I/O operations to function independently of the specific WebSocket standard being used. This approach ensures the WebSocket implementation's flexibility and replaceability. The Socket.IO client is nullable because the WebSocket connection is established only when the user joins a multi-user session. If a WebSocket connection is absent, the WebSocketService remains available but does not transmit any messages.

Upon a user's intention to join a collaborative room, the Socket.IO client is initialized, as illustrated in Listing 6.1. During an active connection, the Socket.IO client manages incoming server messages in the following manner.

Response messages, serving as reactions to previous server requests, trigger optional callback functions dynamically registered by various application components. The identification of request and response pairs is facilitated through unique nonces.

On the other hand, forwarded messages, prompted by updates from remote clients, prompt the WebSocketService to trigger an event corresponding to the name of the receiving Socket.IO channel. This event-driven approach decouples the reception of updates from their subsequent processing, enabling smoother coordination between different processes.

Listing 6.1. Initializing a WebSocket connection on client-side.

```
private currentSocket: Socket|null = null;

async initSocket(ticketId: string, mode: VisualizationMode, userName: string) {
    this.currentSocketUrl = this.getSocketUrl();
    this.currentSocket = io(this.currentSocketUrl, {
      query: {
        "ticketId": ticketId,
        "userName": userName,
        "mode": mode
    }});

    FORWARDED_EVENTS.forEach(event => {
      this.currentSocket?.on(event, message => {
        this.trigger(event, message);
      });
    });

    RESPONSE_EVENTS.forEach(event => {
      this.currentSocket?.on(event, message => {
        const handler = this.responseHandlers.get(message.nonce);
        if (handler) handler(message.response);
      });
    });
   ...
  }
```

6.5.5 Redis Integration

In this section, we'll provide a detailed overview of the technical integration of Redis.

When deploying the Collaboration Mode, it's essential to ensure a fully functional Redis instance. This includes having a datastore that effectively manages reads and writes of key-value pairs while handling message subscriptions. In a Cloud environment, this can be achieved by running the official Docker[4] image of Redis [link to Redis Docker image].

To integrate Redis with our application, we refer to the conceptual implementation of the PubSub component (see Figure 6.3). Although we've simplified the discussion until now, technically, we use the *nestjs-redis*[5] module. This widely-used Node.js module acts as an adapter, connecting the NestJS application to a remote Redis datastore. The setup

[4]https://www.docker.com/

[5]https://www.npmjs.com/package/@liaoliaots/nestjs-redis

for this adapter is performed in our main module, with configurations being provided externally (see Listing 6.2).

Listing 6.2. Integration of Redis with the NestJS application.

```
@Module({
   imports: [
      RedisModule.forRoot({
         config: {
            host: process.env.REDIS_HOST,
            port: parseInt(process.env.REDIS_PORT),
            password: process.env.REDIS_PASSWORD
         }
        })
   ]
```

This setup allows us to declare an application-wide injectable *RedisService*, offering interfaces for executing Redis operations on a specific instance. The RedisService can be effortlessly injected and utilized within any class declaration, as demonstrated in Listing 6.3.

Listing 6.3. Injection of the *RedisService*.

```
private readonly redis: Redis;

constructor(private readonly redisService: RedisService) {
   this.redis = this.redisService.getClient().duplicate();
}
```

Providing separate Redis clients to each consumer class allows for high concurrency, isolation, and individualized configuration.

6.5.6 Load Balancing

We have already emphasized in our design process that message load distribution cannot be arbitrary but rather sticky with respect to client nodes. This means that while different clients may transmit messages to several server nodes, each client is bound to exactly one instance. Consequently, the load must be distributed during WebSocket connection establishment. For instance, we can achieve this using a Round Robin algorithm, where incoming requests alternate between the provided server instances following a fixed order. Additionally, we need to control the stickiness of HTTP requests due to the dynamic nature of Socket.IO. For example, Socket.IO provides a fallback from WebSocket to HTTP polling to reduce startup time or in case of firewalls.

However, the implementation of the load balancer heavily depends on the environment in which ExplorViz is deployed. Since we are conducting our performance analysis in a

cloud environment based on the Kubernetes framework, we present a solution using a Kubernetes Ingress Controller. Ingress Controllers are API objects that manage access to services based on rules. There are various implementations, but we use the *Traefik Ingress Controller*[6] as the basis for our load balancer.

In order for the load balancer to identify clients, *cookie-based routing* is a common approach [Chin et al. 2010]. The ingress controller achieves this by providing an identifier in the initial server response that references the server instance fulfilling a client's first request (see Listing 6.4). Subsequent requests which have the cookie with the name "collaboration-cookie" set to that specific identifier will automatically be forwarded to the exact same server instance. Thus, the frontend must manage this cookie accordingly to ensure that requests reach the correct server.

Listing 6.4. Configuraton of the Traefik Ingress Controller for sticky load balancing.

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: collaboration-service
5   annotations:
6     traefik.ingress.kubernetes.io/service.sticky.cookie: "true"
7     traefik.ingress.kubernetes.io/service.sticky.cookie.name: collaboration-cookie
8     traefik.ingress.kubernetes.io/service.sticky.cookie.secure: "true"
9 ...
```

6.6 Testing

We conclude the re-engineering process by conducting comprehensive testing of the re-implemented Collaboration Mode. This testing process has two primary objectives. Firstly, we must ensure that the functional requirements are correctly fulfilled. Secondly, we evaluate the performance with regard to our goals. This section is dedicated to assessing the functional correctness of the application, while Chapter 7 focuses on evaluating the performance characteristics.

The Collaboration Mode's functionality is delivered by both the frontend and backend application. However, the frontend component is seamlessly integrated into ExplorViz's Frontend, which has its own established test strategy encompassing acceptance, integration, and unit tests. As the re-engineering process does not impact the functional attributes of the frontend, we consider the existing tests as a quality gate for the frontend migration.

In contrast, the backend of the Collaboration Mode has undergone a complete re-implementation. It operates within a different runtime environment and offers a distinct external interface compared to the previous system. Consequently, we must design suitable automated tests to validate the functionality of the Collaboration Service.

[6] https://doc.traefik.io/traefik/providers/kubernetes-ingress/

During the requirement engineering process, we identified a set of user stories summarizing the abstract functionality of the Collaboration Mode. These user stories serve as our guiding framework for designing the tests. Since user stories encapsulate functionality from a user's perspective, we adopt an *end-to-end* testing approach to transform these stories into automated tests. End-to-end testing involves verifying the functionality of the fully integrated software system from the end-user's viewpoint [Bai et al. 2001]. In our context, we narrow the scope to focus on the backend and interpret network requests as user actions. Nevertheless, our end-to-end tests encompass the fully integrated backend, abstracting from internal dependencies. This approach ensures that the end-to-end tests can be conducted in production-like environments and at different scaling levels.

We use *Jest*[7] as our testing framework, allowing us to define independent test cases and make precise assertions. In general, we define two test suites: one for testing room management via the corresponding API, and another for testing client synchronization via WebSockets. The latter also covers edge cases related to mutual exclusion.

[7] https://jestjs.io/

Chapter 7

Performance Analysis of the Re-Engineered Collaboration Mode

In this chapter, we delve into the performance analysis of the re-engineered Collaboration Mode. Initially, we outline the steps taken to conduct this performance analysis. Subsequently, we showcase the results. Based on these findings, we proceed to evaluate the overall performance.

7.1 Preparation

Our long-term goal is to improve the performance of the Collaboration Mode. Until now, we have analyzed the performance of the previous Collaboration Mode. Our next step involves conducting a performance analysis of the re-engineered Collaboration Mode to evaluate the outcomes of the re-engineering process. For the sake of comparability, we execute the benchmark as defined in Chapter 4 and employ the same configuration as in Chapter 5. However, the re-engineered Collaboration Mode operates on a distinct system architecture. Consequently, we need to slightly adapt the experimental setup of the performance test to align with the altered requirements while ensuring benchmark comparability.

The refined experimental backend setup for the benchmark execution is outlined in Figure 7.1. A fundamental feature of the re-engineered Collaboration Mode is its ability to scale horizontally. Therefore, we introduce the sticky Load Balancer, as explained in Chapter 6, between the Load Generator and the Collaboration Service deployment. During runtime, this Load Balancer evenly distributes incoming traffic among a configurable number of Collaboration Service replicas. Additionally, we incorporate a Redis instance within the cluster to manage inter-server communication. The resource limitations, as defined in Chapter 5, also apply to the re-engineered Collaboration Service. However, when scaling the deployment out, each replica is endowed with these resources.

The experimental setup for the frontend performance test must also be adjusted, as the execution of the frontend depends on the Collaboration Service. Consequently, we substitute the backend components in the frontend setup with the corresponding components from Figure 7.1. Once again, only one instance of the Load Generator is deployed. Furthermore, the Collaboration Service is provided by a single replica since the frontend test exclusively examines frontend-specific metrics and only requires the backend to forward workload to

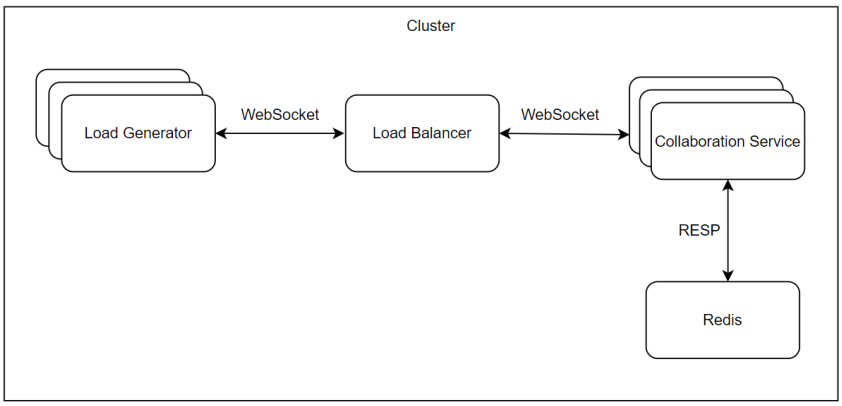


Figure 7.1. Deployment of the performance test with re-engineered backend.

the frontend instance.

The backend performance test is divided in two parts. Firstly, we execute the benchmark with a single instance of the Collaboration Service, following the approach in the previous benchmark execution. This ensures that the results remain comparable, as we maintain consistent resource limits and only alter the internal behavior of the Collaboration Mode. The second part of the backend analysis explores system scalability. Thus, we successively increase the number of replicas while monitoring the performance metrics of each instance. Manifestly, deploying multiple instances leads to higher resource allocation and consequently changes the prerequisites of the benchmark execution compared to Chapter 5. However, each instance individually meets the resource requirements and thus provides comparable insights.

7.2 Results

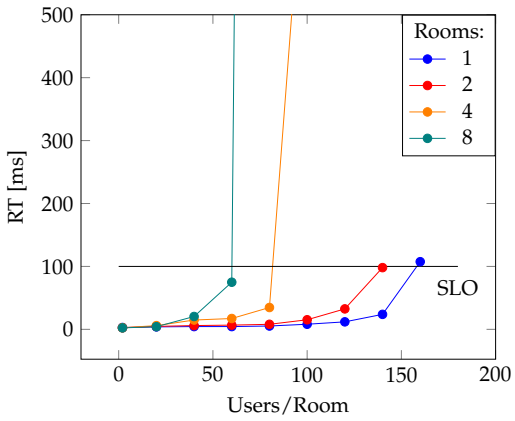
In this section, we present the results of both the backend and frontend performance analyses [Brück 2023].

7.2.1 Backend

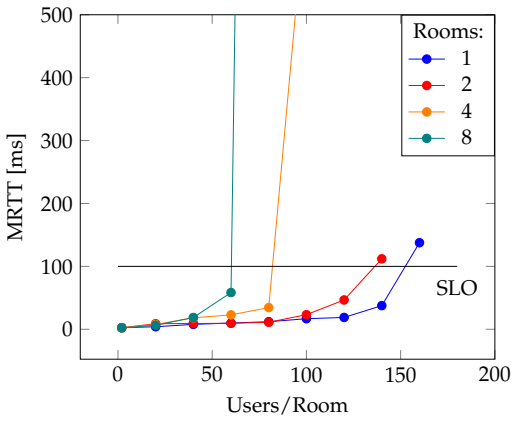
The first subsection presents the results of the performance benchmarking with a single backend instance for both user populations: the Browser Population and the Cross-Platform Population. Following this, the performance is analyzed for various scaling configurations.

Unscaled

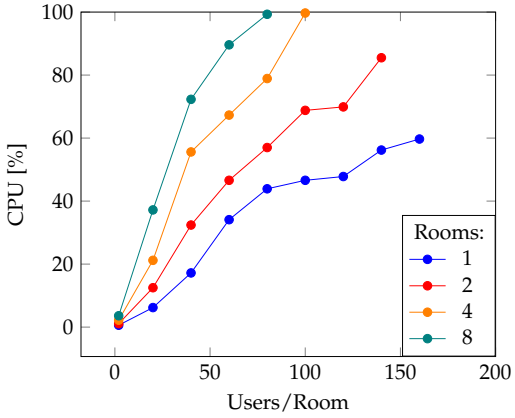
The performance results for the Browser Population are depicted in Figure 7.2.



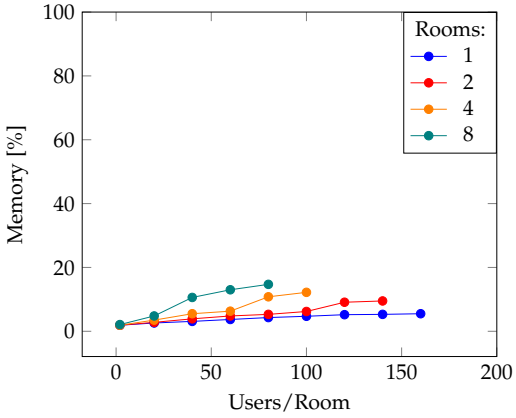
(a) Response time.



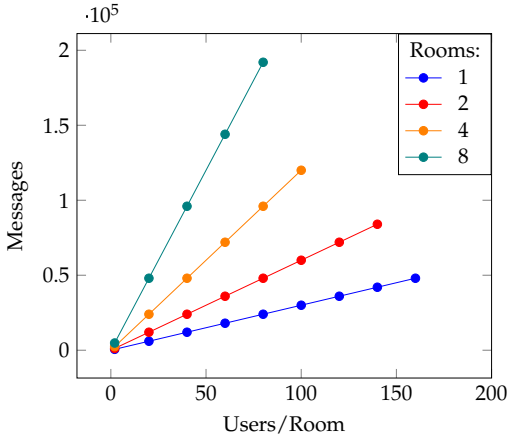
(b) Mean round trip time.



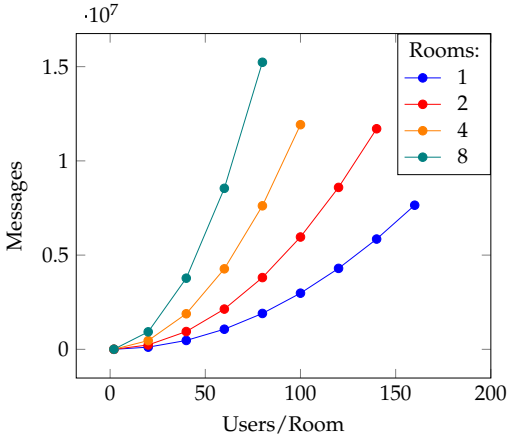
(c) CPU utilization.



(d) Memory utilization.



(e) Number of messages received by server.



(f) Number of messages sent by server.

Figure 7.2. Performance results of the re-engineered backend for the Browser Population.

The diagrams follow the structure introduced in Chapter 5. The minimal service time, denoted as RT and MRTT for a minimum of two users, remains consistently low across all room counts, never exceeding 5 milliseconds. This pattern is mirrored in the resource utilization as well. Under minimal workload per room, the resource utilization for both CPU and memory remains below 5%.

In general, RT and MRTT exhibit similar behavior. When the backend serves only one room at a time, both function graphs have a very gentle slope. Even when synchronizing 140 browser users simultaneously, the responsiveness remains solid, with RT and MRTT staying below 40 milliseconds. It is only when the user count reaches a minimum of 160 that the metrics suddenly spike, with MRTT exceeding the SLO's threshold by 30%. Moreover, the backend efficiently handles multiple rooms in parallel. Specifically, two parallel rooms maintain very high responsiveness until a maximum of 120 users per room. However, assuming four rooms simultaneously, the maximum users per room drops to 80. Likewise, for eight rooms, the maximum users per room falls to 60.

The CPU and memory utilization result in substantially different diagrams. While both metrics show a monotonic increase, the function values have significantly different value ranges. Memory utilization consistently remains below 20%, while CPU usage hits its limit in certain configurations. For instance, configurations with four rooms, each with 100 users, or eight rooms, each with 80 users, require nearly 100% of CPU time.

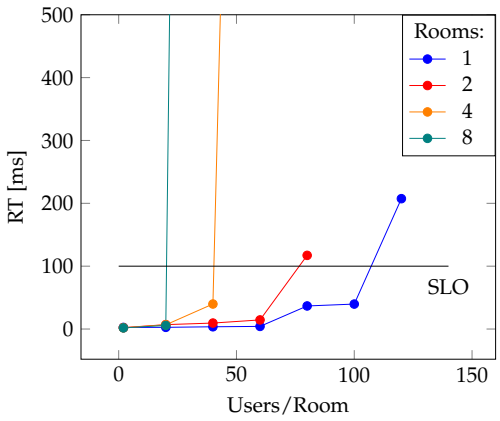
Lastly, the metrics related to message traffic depict strictly monotonically increasing curves, quickly reaching high values. The server can handle a maximum of 144,000 received messages during the test period, leading to 8,544,000 messages sent.

The results of synchronizing the Cross-Platform Population are visualized in Figure 7.3.

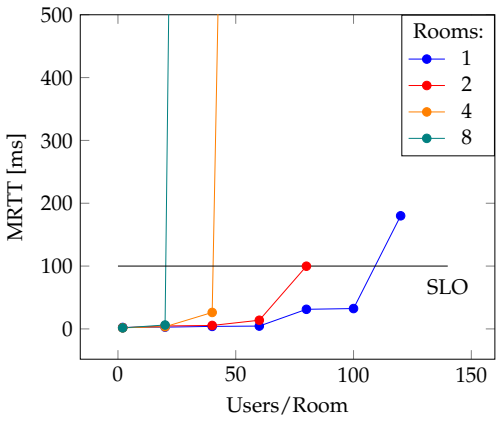
In cross-platform sessions as well, the Collaboration Service demonstrates high responsiveness across various room and user counts. The maximum number of clients that can be served in a single room is approximately 100. Even when providing two rooms in parallel, 60 users experience satisfactory responsiveness. However, with an increased user count of 80 users for each of two rooms, the SLO's threshold is slightly exceeded by 18% in RT. Scaling the number of parallel rooms consequently results in a performance dip. Nonetheless, both RT and MRTT meet the SLO requirement for up to 20 users in eight rooms.

The CPU metric exhibits steep curves for all room counts. While the minimum number of users per room utilizes at most 15% of the CPU capacity, the maximum configuration leads to almost 100% utilization for every room count. However, memory utilization remains consistently low even during load peaks.

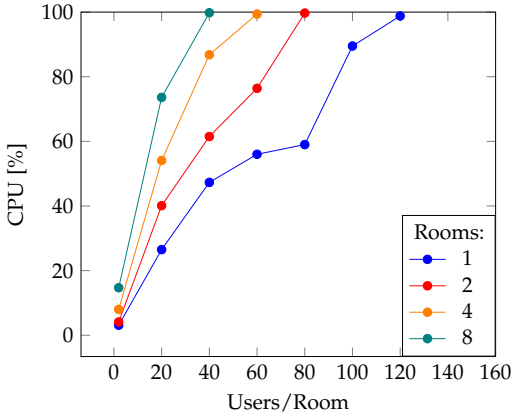
The number of transmitted messages dramatically increases with higher workload. For example, when eight parallel rooms are filled with 20 users each, the Collaboration Service receives 624,000 messages and sends over 3,000,000 messages. Despite this, the backend maintains stable responsiveness. Message loads exceeding these numbers eventually lead to SLO violations.



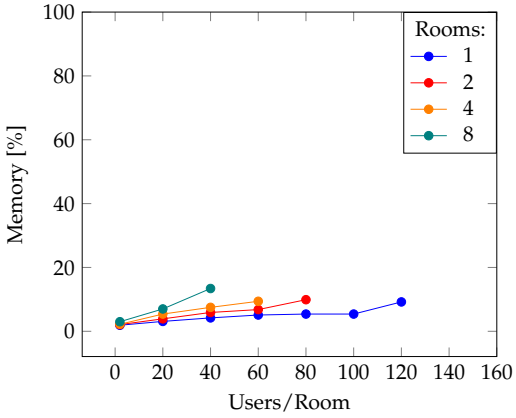
(a) Response time.



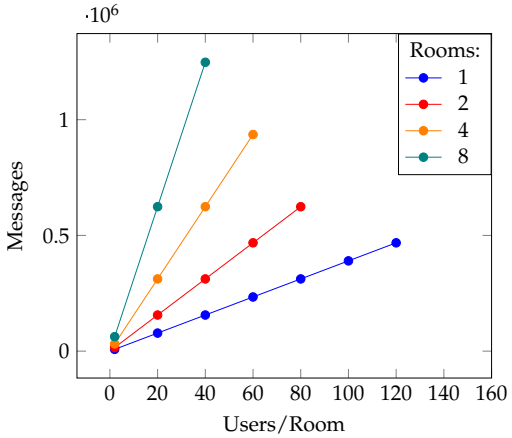
(b) Mean round trip time.



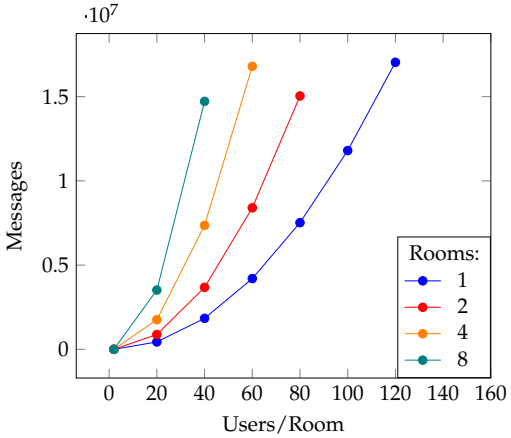
(c) CPU utilization.



(d) Memory utilization.



(e) Number of messages received by server.



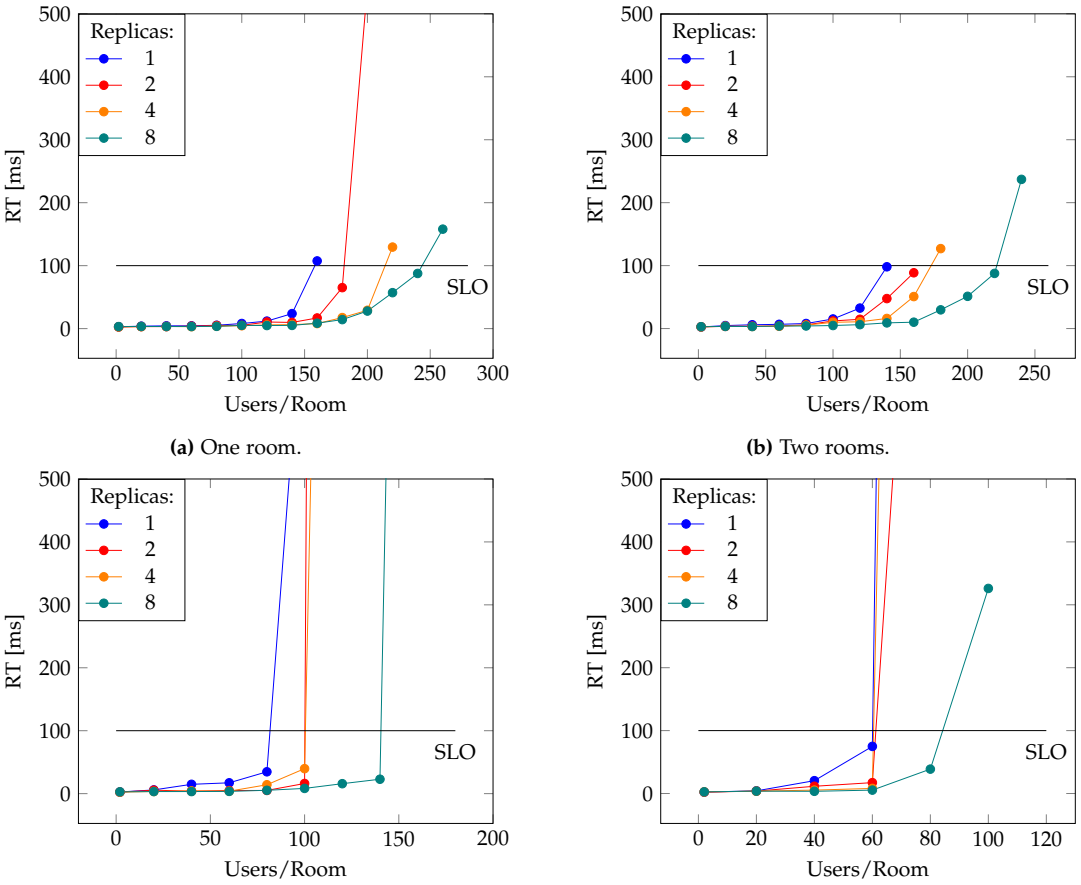
(f) Number of messages sent by server.

Figure 7.3. Performance results of the re-engineered backend for the Cross-Platform Population.

Scaled

In this section, we present the performance results of the Collaboration Service while scaling it out to two, four, and eight parallel instances.

The results of simulating the Browser Population are illustrated in Figure 7.4.



(a) One room.

(b) Two rooms.

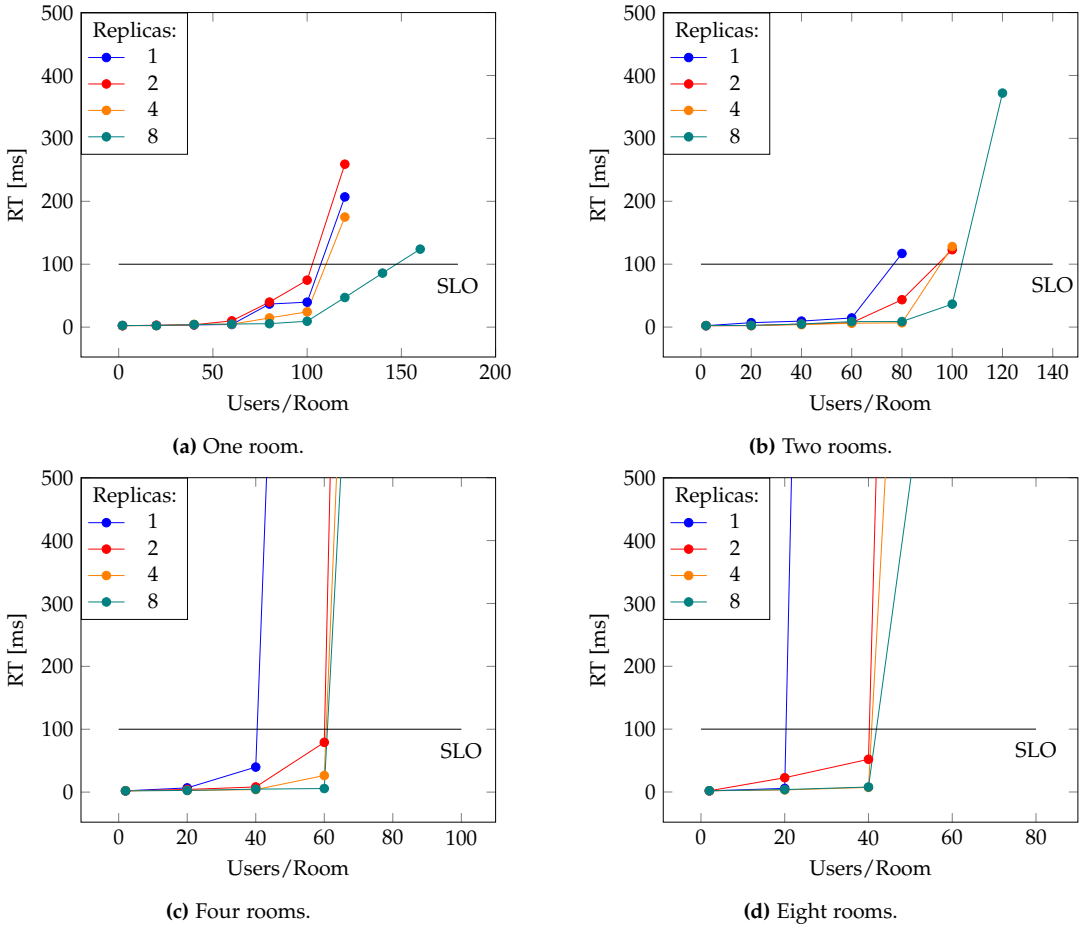
(c) Four rooms.

(d) Eight rooms.

Figure 7.4. Response Time of the scaled backend for the Browser Population.

The diagrams adopt a different structure than the previous ones. Firstly, we do not visualize all metrics to maintain clarity and understanding. Instead, we prioritize the RT metric, representing the responsiveness of the Collaboration Service and user satisfaction concerning the corresponding SLO requirement. Additionally, the RT and MRTT exhibit very similar behavior and correlate regarding SLO violations. The function values of the RT are averaged based on the individual RTs of each Request Message. Furthermore, the

diagrams are distinguished by the room counts, where each curve in a diagram represents a different number of parallel backend instances. This structure was chosen to highlight the runtime behavior at various scales. Lastly, we include curves for the unscaled Collaboration Service for comparison.



(a) One room.

(b) Two rooms.

(c) Four rooms.

(d) Eight rooms.

Figure 7.5. Response Time of the scaled backend for the Cross-Platform Population.

While serving a single room, the backend effectively manages high user numbers across all scaling levels. Additionally, the function values are quite consistent for various configurations, mostly staying below 20 milliseconds. However, the maximum number of users that can be synchronized with high responsiveness differs among the scaling levels. While a single instance can serve 140 users, eight replicas can handle up to 240 users in parallel. However, the difference between one and two instances is not significant, as the

latter reaches its maximum at 180 users.

For two rooms, a similar behavior is observed. However, the curves already show an increase for smaller user counts. A single instance can effectively serve 120 users per room without violating the SLO requirement, while eight parallel instances reach their maximum at 220 users. For higher room counts, such as four or eight rooms simultaneously, the performance evolves differently. Nonetheless, the maximum number of users monotonously increases with the scaling level. However, providing two or four replicas leads to the same maximum in both room counts. Additionally, most curves exhibit a steep slope precisely when the RT is about to exceed the SLO's threshold, indicating a significant SLO violation.

The results of the Cross-Platform Population are depicted in Figure 7.5.

In general, a similar pattern is observed as in the results of the Browser Population. However, the overall maximum number of users that can be synchronized is smaller. For small workloads per room, the RTs of the different scaling levels do not differ significantly. Again, the maxima monotonously increase with the number of server instances. However, in many configurations, a higher scaling level does not result in a higher maximum user count, even though the overall responsiveness is slightly better—i.e., the respective RT is marginally smaller. For instance, in a single room scenario, the SLO is violated above a user count of 100, regardless of the scaling level—be it one, two, or four. When considering a higher room count, such as eight rooms, scaling to two, four, or eight replicas makes no substantial difference, as the SLO cannot be fulfilled above a maximum of 40 for all these scaling levels.

7.2.2 Frontend

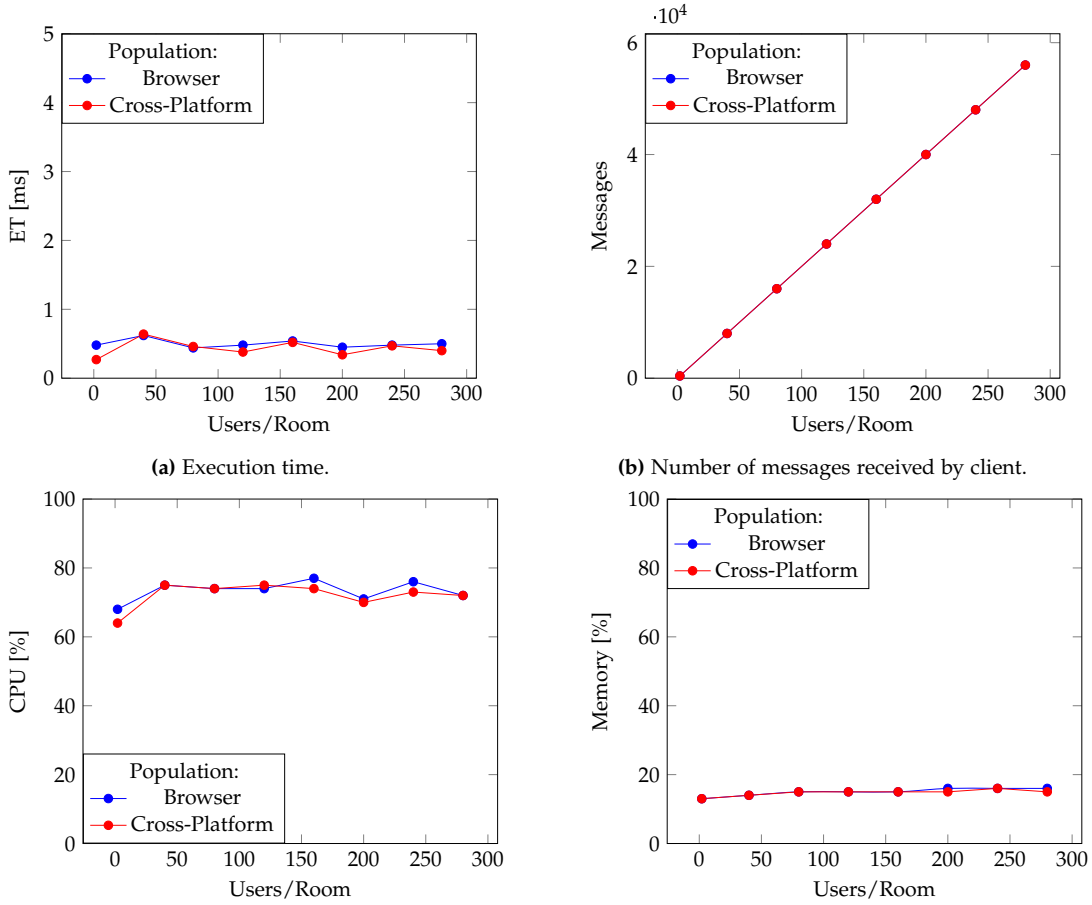
The results of the frontend performance test are presented in Figure 7.6. As in Chapter 5, we terminated the benchmark execution after reaching 280 users since the metrics remained unchanged. Additionally, the structure of the diagrams mirrors that of Chapter 5, with each curve representing one of the two user populations.

Overall, it is evident that the performance of both user populations behaves similarly. The ETs of received messages differ only on a microsecond scale between the populations. Additionally, the resource utilization shows minimal differences—less than 4% for the CPU and less than 2% for the memory. Moreover, the number of received messages is identical for both the Browser Population and the Cross-Platform Population.

The ET remains relatively consistent across different configurations. For all workloads ranging from 2 to 280 users, the ET is very fast, consistently staying below 1 millisecond.

The CPU utilization is relatively high even for minimal workloads, consistently exceeding 64%. However, there is no noticeable increase observed with a growing user count, and the CPU utilization never exceeds 77%. The variance in memory utilization is even smaller, with memory occupancy hovering around 15% throughout.

Lastly, the message count displays a linear increase with the growing user load, reaching a value of 56,000 received messages for the highest configuration of 280 users.



(a) Execution time.

(b) Number of messages received by client.

(c) CPU utilization.

(d) Memory utilization.

Figure 7.6. Performance results of the re-engineered frontend.

7.3 Discussion

The previous results demonstrate that the Collaboration Mode offers high responsiveness across various configurations. The unscaled deployment of the Collaboration Service effectively serves up to 140 browser users without any performance degradation. Within this capacity, WebSocket events are processed and transmitted rapidly. In cross-platform scenarios, up to 100 users can be synchronized in a single room. For even higher user counts, there is a gradual decrease in responsiveness when the SLO is violated. However, service availability remains generally stable even during peak loads.

Furthermore, the backend is well-suited to support multiple rooms with high interaction in parallel. It handles eight rooms with 60 browser users or 20 cross-platform users very effectively. However, in cases of high room counts or many XR users, the backend tends to experience abrupt performance degradation. This may be attributed to the intense message traffic. While the number of messages received by the backend grows linearly with the number of users, the count of sent messages increases quadratically.

The frontend of the Collaboration Mode easily supports 280 users, regardless of the platform distribution. Specifically, the frontend is platform-independent, as it only receives browser-specific messages, as evidenced by the congruency of population-specific message counts.

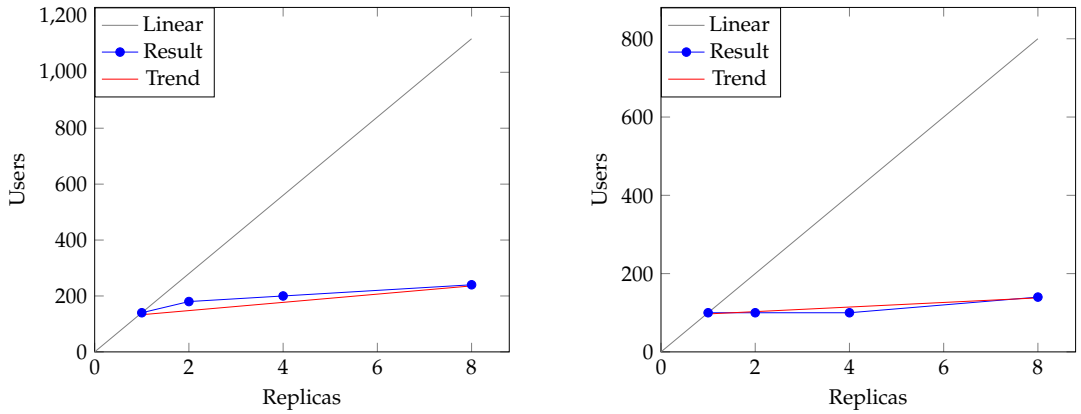
The evolution of resource utilization shows that the Collaboration Service efficiently utilizes CPU resources, as responsiveness usually does not degrade when the CPU is not fully occupied.

Furthermore, scaling out the Collaboration Service improves its overall behavior. Higher scaling levels do not significantly improve responsiveness for small user sessions. However, having multiple instances in parallel generally does not worsen responsiveness, demonstrating the effective and rapid handling of inter-server communication via Publish/Subscribe. The most significant improvement in the Collaboration Mode with a scaled backend is the increased capacity. By scaling out to eight replicas, the backend can progressively serve up to 280 browser users or 140 cross-platform users in a single room.

Finally, we analyze the effectiveness of horizontal scaling [Henning and Hasselbring 2021]. Although a comprehensive scalability analysis is beyond the scope of this thesis, our objective is to gain a rudimentary understanding of the system's general ability to scale. Therefore, we plot the maximum user capacity of individual scaling levels for one room in Figure 7.7. It provides two diagrams for both populations under test. Each diagram visualizes the linear trend estimation as well as the ideal scalability graph, which is derived linearly from the capacity of the unscaled deployment.

In general, there is an eventual increase with growing scaling level. However, the effect of adding a single replica to the deployment is not substantial, and the capacity does not grow linearly. Based on the linear trend estimation, the browser scenario shows an increase of capacity by approximately 10%, while the cross-platform scenario exhibits an increase of around 6% for every additional replica.

However, the user count does not directly indicate the actual workload during execution.

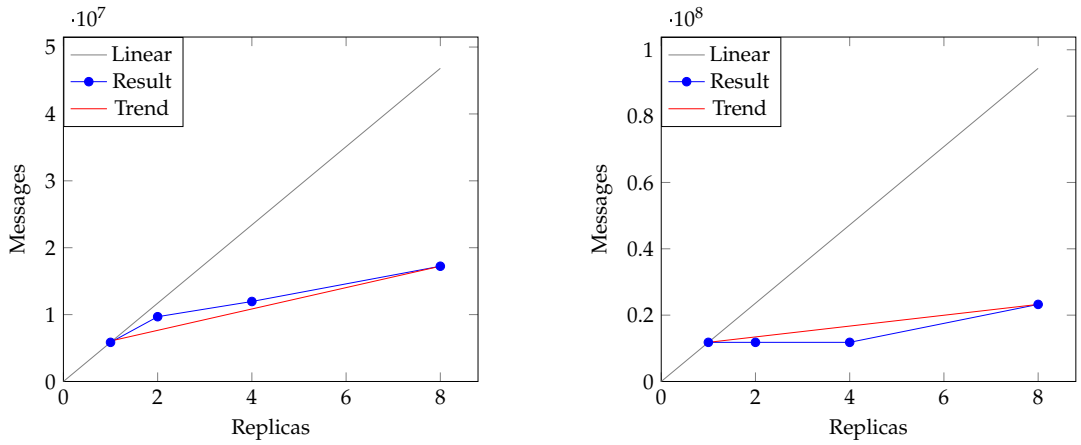


(a) Browser Population.

(b) Cross-Platform Population.

Figure 7.7. Maximum number of users within the SLO requirement for different scaling levels.

The metrics reveal a linear correlation between the user count and the number of received messages on the server-side. Still, the number of messages sent by the server grows quadratically with the user count. Therefore, we visualize the capacity in terms of actual network load due to transmitted WebSocket messages in Figure 7.8. The diagrams follow the same structure as the previous one but visualize the corresponding message count on the y-axis.



(a) Browser Population.

(b) Cross-Platform Population.

Figure 7.8. Maximum number of transmitted messages within the SLO requirement for different scaling levels.

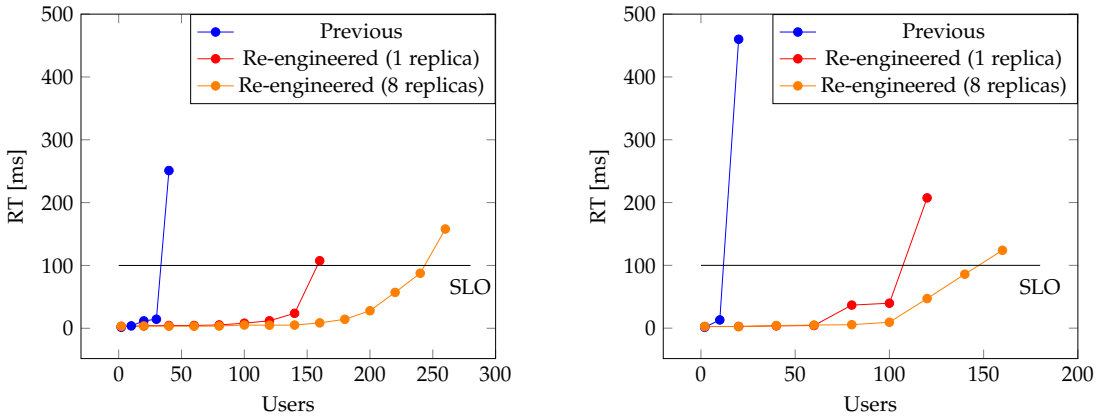
The scalability is not nearly linear, but it can be observed as a stepwise growth in the browser-based scenario. The Cross-Platform Population requires higher scaling levels until a significant growth in capacity is noticeable. The increase in capacity by adding one server replica is approximately 26% for the Browser Population and 14% for the Cross-Platform Population. Thus, scaling the Collaboration Service out is not as efficient as initially assumed, which may be attributed to the overhead from intense inter-server communication.

The varying scalability results of the two user populations may be attributed to the higher message traffic in cross-platform scenarios. Similarly, an increased number of simultaneous rooms dramatically increases message load, resulting in weaker performance even in a scaled environment. However, in our simulated environment, we distributed the users of each room equally based on the Round-Robin policy of the Load Balancer. A more intelligent distribution of connections, keeping users of the same room together, may allow the scaled infrastructure to be utilized more effectively.

Chapter 8

Discussion

In this chapter, we present and analyze our findings in order to evaluate our primary objective: enhancing the performance of ExplorViz's Collaboration Mode to facilitate multi-user sessions with high interaction. Our strategy involved re-engineering the Collaboration Mode based on a performance analysis. To assess the efficacy of our approach, we conducted two competitive benchmark executions, one for the previous system and another for the re-engineered system. Our intent is to compare the outcomes and evaluate the re-engineering effort in light of our defined goal.



(a) Browser Population in one room.

(b) Cross-Platform Population in one room.

Figure 8.1. Response time comparison between the previous and the re-engineered Collaboration Mode.

A crucial performance aspect is responsiveness, gauged through the metrics RT and MRTT. In Figure 8.1, we compare the responsiveness of the re-engineered system with that of the previous system, using RT as a benchmark as both RT and MRTT showed similar results in each respective analysis. The figure depicts two graphs representing different user populations. Both graphs display the resulting RT for increasing user counts in a single room, illustrating curves for different versions and scaling levels of the Collaboration Mode: the inherently unscalable previous system, the scalable re-engineered system with one replica, and the re-engineered system with eight replicas.

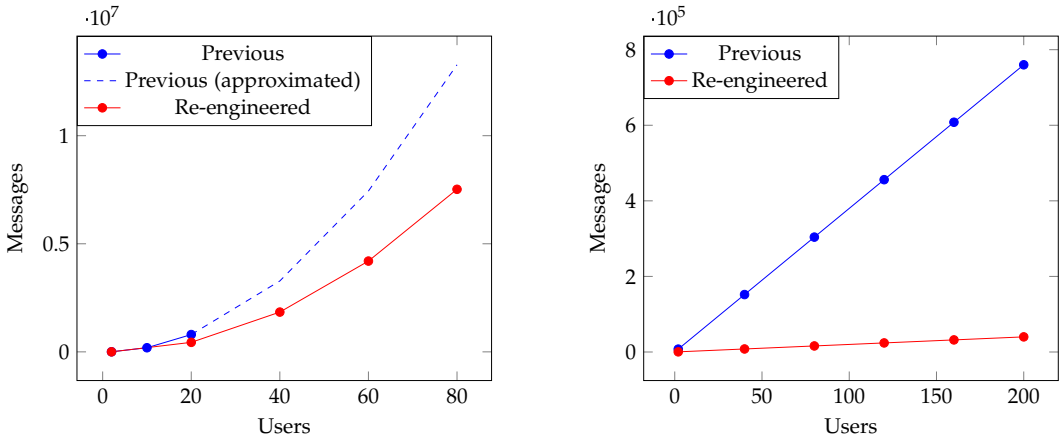
Evidently, the performance has improved significantly. For smaller user counts, the responsiveness remains low for both versions. However, the re-engineered system showcases a significantly higher maximum user capacity that can be served with high responsiveness. While the previous system capably handles a maximum of 30 browser users, the re-engineered system effortlessly accommodates over triple that amount. For the cross-platform users a similar improvement can be observed. Moreover, the re-engineered system can effectively handle multiple rooms in parallel, while the previous system was quickly overstrained by simultaneous session. We conclude that the modular application architecture of the Collaboration Service based on NestJS and Socke.tIO constitutes an effective approach for developing performant and efficient real-time applications. This conclusion is substantiated by the efficient utilization of the CPU in the re-engineered system, while the previous system's performance drops despite the CPU being far from its limits. This observation suggests that the re-engineered backend encounters fewer issues related to blocking operations and thread stagnation.

Furthermore, our enhancements enable the Collaboration Mode to be horizontally scaled by implementing inter-server communication with Redis. The performance results indicate a noticeable increase in user capacity with the addition of multiple replicas, all while maintaining an instantaneous user experience. Even at high scaling levels, responsiveness remains uncompromised, showcasing the rapid handling of Publish/Subscribe communication. Although the efficiency of scalability is not optimal, with approximately 25% improvement of message capacity upon adding one replica, successive addition of replicas progressively increases capacity. We conclude that distributing clients among multiple replicas is a suitable approach to balance the overall application load in the presence of WebSocket connections.

Moreover, the re-engineering effort significantly contributed to system reliability. We introduced automatic disconnection detection with Socket.IO, coupled with cleanup mechanisms based on this detection. These mechanisms ensure fail-safety in the face of client-node errors. Additionally, Socket.IO allows the Collaboration Mode to handle connections even in environments that restrict WebSockets, providing a fallback to HTTP polling. To support this even in a scaled infrastructure, we presented an approach for sticky load balancing, ensuring stable connections through cookie-based routing. The high responsiveness observed in our results indicates that the overhead associated with reliability is minimal.

During the re-engineering process, we made efforts to reduce WebSocket message traffic, given our detection of a correlation between the number of messages and responsiveness. To achieve this, we implemented platform-specific messaging. The results of this effort are visualized in Figure 8.2, which presents message counts for different room sizes gathered during performance analysis, distinguishing between messages sent by the backend and those received by the frontend. Although we lacked the precise message count for the previous backend across various user configurations, we approximated the count based on our knowledge gained through reverse engineering message generation.

Both diagrams demonstrate a reduction in the number of transmitted messages due to



(a) Number of messages sent by server.

(b) Number of messages received by client.

Figure 8.2. Comparison of WebSocket message traffic between the previous and the re-engineered Collaboration Mode.

the re-engineering. The number of sent messages still exhibits quadratic growth concerning the user count, though at a slower rate. This has a noticeable impact on the previously mentioned responsiveness results. The frontend of a browser user is now completely independent of XR users, resulting in significantly lower message traffic per browser user. However, the effect on responsiveness is negligible as WebSocket messages are processed very quickly on client-side.

Chapter 9

Related Work

In this chapter, we present topics related to this thesis, focusing on the scientific discussion and investigations, as well as concrete tools and architectures dealing with collaboration at a high scale. We will discuss selected works in the following sections.

Mouton et al. [2011] provides an insightful overview of trends concerning collaborative visualization tools. They identify browser-based rendering with WebGL and WebSocket communication as established solutions. Additionally, they advocate minimizing message traffic between nodes while shifting computational effort to the clients. From an abstract perspective, ExplorViz aligns with these trends. It implements WebGL and WebSocket communication. Furthermore, user actions are primarily processed in the client's browser, with only collaborative events exchanged between client nodes.

Marion and Jomier [2012] present an architecture based on WebGL and WebSocket using the example of the *Visible Patient* project[1], an online laboratory for 3D modeling of medical images. The designed system allows geographically distributed experts to collaboratively explore medical data by real-time synchronization of client instances. A significant difference in ExplorViz's collaboration approach is that the Visible Patient architecture introduces only one master node, which can manipulate the visualization data, while all other clients are only spectators, receiving visualization updates from the master (see Figure 9.1). Thus, most of the network traffic is one-directional.

Furthermore, Marion and Jomier [2012] benchmark the performance of the system to compare WebSocket messaging using Socket.IO to HTTP-polling with *AJAX* [El Moussaoui and Zeppenfeld 2010]. The results show that AJAX has a 3-times higher latency for data updates. Finally, they identify a performance limitation regarding the size of the dataset, as the medical model has to be propagated to all connected clients during the collaborative session. We do not face this issue in ExplorViz since the landscape data is initially fetched from the respective backend services, and the WebSocket session only exchanges collaborative user events.

Grasberger et al. [2013] discuss a data-efficient approach for collaborative modeling of 3D sketches. They use WebSocket connections for exchanging collaboration events and the *BlobTree* paradigm, a representation method used in computer graphics that allows for the creation and manipulation of complex shapes in a compact and efficient manner [Nishino et al. 1999], for minimizing data traffic. Similarly to our work, Grasberger et al.

[1] https://www.visiblepatient.com/en/

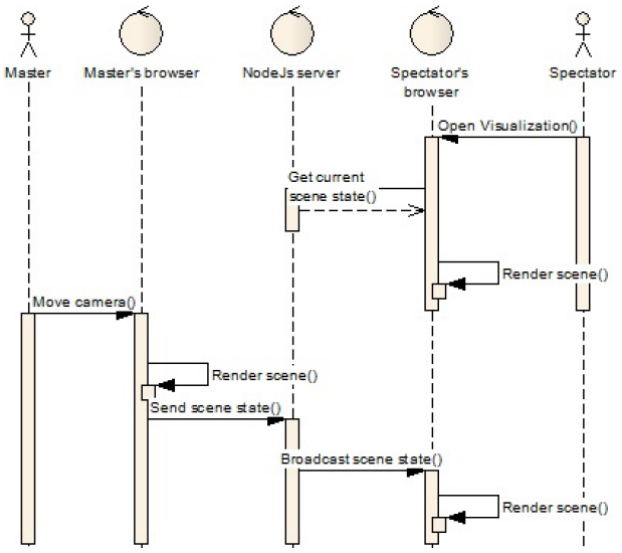


Figure 9.1. Communication pattern of Visible Patient [Marion and Jomier 2012].

[2013] identified the central WebSocket server as the main performance bottleneck for fast client synchronization. However, they resolve the bottleneck by introducing a server-less architecture, a significant contrast to our architecture, where we address performance issues through horizontal scaling. As stated by Grasberger et al. [2013], the server-less approach presents some challenges. For instance, conflicting user actions, such as altering an object that has already been deleted, must be resolved among clients, as there is no single point of truth. They resolve such conflicts by implementing *optimistic timestamp ordering* [Kung and Robinson 1981], allowing user actions without prior validation and undoing them in case of a preceding conflicting event. Therefore, every user action is equipped with a timestamp based on *Lamport clocks*, which are a concept of logical time that allows a partial ordering of timestamps [Lamport 1978]. In ExplorViz' Collaboration Mode, we resolve conflicting user events, such as grabbing and closing an object simultaneously, pessimistically through prior server validation. In a scaled environment, we use distributed locks to ensure consensus between multiple servers.

The work of Alexeev et al. [2019] faces the challenge regarding horizontal scaling of WebSocket servers in the context of large-scale, browser-based *grid computing*. Grid computing is a computing paradigm that involves coordinating a large number of heterogeneous computers to solve complex computational problems in a distributed manner [Fedak 2015]. In the work of Alexeev et al. [2019], the clients act as workers for computational tasks, and the servers are responsible for distributing the problem-solving tasks among the clients. As we do in this thesis, they implement the Publish/Subscribe pattern based on Redis for effective inter-server communication and use WebSocket for client connections. However,

there are some differences compared to our approach. To distribute the load among the servers, they balance sticky client connections via an intelligent Orchestrator service (see Figure 9.2), which assigns clients to server instances based on the individual client's behavior profile, e.g., life-term or maximum traffic expectation, and server monitoring, e.g., CPU or RAM utilization.

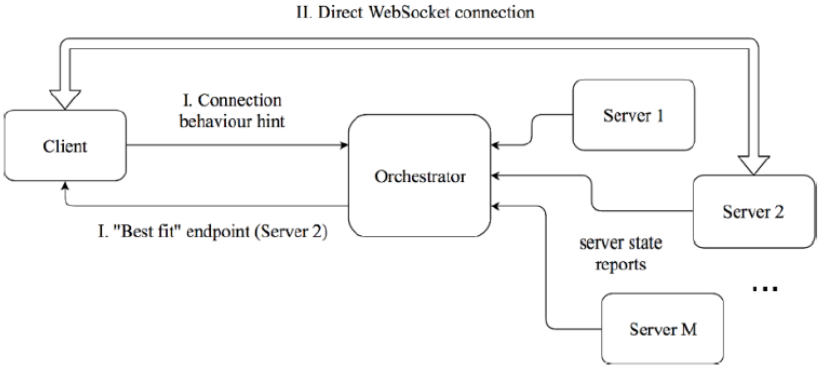


Figure 9.2. Intelligent load balancing through Orchestrator service [Alexeev et al. 2019].

Additionally, to handle the high data traffic between the servers, they scale Redis out to multiple instances. However, the work does not provide performance insights into running the architecture in production.

Chapter 10

Conclusions and Future Work

In this chapter, we provide a summary of the thesis and propose ideas for future work.

10.1 Conclusions

The primary objective of this thesis was to enhance the performance of ExplorViz's Collaboration Mode, particularly in facilitating multi-user sessions with extensive interaction. To evaluate the performance of the previous Collaboration Mode, we defined a representative performance benchmark and applied it to the respective system. The benchmarking analysis revealed the Collaboration Service and the substantial message overhead as the main bottlenecks for performance, while underscoring the robustness of the frontend.

In response, we conducted a comprehensive re-engineering of the Collaboration Mode. The resulting system enables horizontal scaling through inter-server communication via Publish/Subscribe and provides reliable WebSocket connections with disconnection detection and sticky load distribution. Additionally, we introduced platform-specific messaging to minimize overall network traffic.

Conducting a second competitive benchmarking analysis with the re-engineered system demonstrated high responsiveness and effective resource utilization for different user populations and multiple rooms in parallel. Furthermore, we significantly increased the maximum capacity of users capable of experiencing high responsiveness simultaneously by more than threefold using the same resources. Scaling the Collaboration Service out allows for a successive increase in capacity. However, the scalability is somewhat impeded due to the high overhead for server coordination, adding one additional replica increased the processable message capacity by approximately 26%.

10.2 Future Work

In the future, a more in-depth scalability analysis of the Collaboration Service could provide valuable insights on improving the efficiency of horizontal scaling. Benchmarking strategies to reduce the overhead of coordinating multiple server instances could be a promising approach. For instance, implementing timestamp ordering, as demonstrated by Grasberger et al. [2013], or adopting a more relaxed consistency model, where not every server instance needs to replicate the entire application data, might be viable alternatives.

In our thesis, we introduced sticky load balancing based on the Round Robin algorithm. However, a more sophisticated load balancer, such as the Orchestrator proposed by Alexeev et al. [2019], could route the traffic based on the utilization of individual server instances or distribute client messages room-wise to keep clients from the same room together on a server. Furthermore, horizontally scaling the Redis datastore could enhance inter-server communication speed [Alexeev et al. 2019].

Additionally, for future work, it would be beneficial to test the architecture in various scenarios that align with the practical use of ExplorViz. For instance, running ExplorViz with geographically distributed users may impact message transmission latency and should be considered in the evaluation.

Bibliography

[Abbasi-Kesbi et al. 2017] R. Abbasi-Kesbi, H. Memarzadeh-Tehran, and M. J. Deen. Technique to estimate human reaction time based on visual perception. *Healthcare Technology Letters* 4 (2017), pages 73–77. (Cited on page 15)

[Alexeev et al. 2019] V. Alexeev, P. Domashnev, T. Lavrukhina, and O. Nazarkin. The Design Principles of Intelligent Load Balancing for Scalable WebSocket Services Used with Grid Computing. *Procedia Computer Science* 150 (2019). Proceedings of the 13th International Symposium "Intelligent Systems 2018" (INTELS'18), 22-24 October, 2018, St. Petersburg, Russia, pages 61–68. (Cited on pages 38, 70, 71, 74)

[Bader 2022] M. Bader. Design and Implementation of Collaborative Software Visualization for Program Comprehension. Master's thesis. Kiel University, June 2022. (Cited on pages 33, 36, 42, 43)

[Bai et al. 2001] X. Bai, W. T. Tsai, T. Shen, B. Li, and R. A. Paul. Distributed end-to-end testing management. *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference* (2001), pages 140–151. (Cited on page 52)

[Bierman et al. 2014] G. Bierman, M. Abadi, and M. Torgersen. Understanding typescript. In: *European Conference on Object-Oriented Programming*. Springer. 2014, pages 257–281. (Cited on page 43)

[Bourque and Fairley 2014] P. Bourque and R. E. Fairley, editors. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Version 3.0. Los Alamitos, CA: IEEE Computer Society, 2014. (Cited on page 36)

[Brück 2023] J. Brück. *Supplementary package for thesis: Performance Analysis and Re-Engineering of ExplorViz's Collaboration Mode*. Oct. 2023. URL: https://doi.org/10.5281/zenodo.8435907. (Cited on pages 25, 26, 54)

[Chin et al. 2010] M. L. Chin, C. E. Tan, and M. I. Bandan. Efficient load balancing for bursty demand in web based application services via domain name services. In: *8th Asia-Pacific Symposium on Information and Telecommunication Technologies*. 2010, pages 1–4. (Cited on page 51)

[Curry 2004] E. Curry. "Message-Oriented Middleware". In: *Middleware for Communications*. Edited by Q. H. Mahmoud. Chichester, England: John Wiley and Sons, 2004. Chapter 1, pages 1–28. (Cited on pages 7, 38)

[El Moussaoui and Zeppenfeld 2010] H. El Moussaoui and K. Zeppenfeld. *AJAX: Geschichte, Technologie, Zukunft*. Informatik im Fokus. Heidelberg: Springer, 2010. (Cited on page 69)

[Eugster et al. 2003] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35.2 (June 2003), pages 114–131. (Cited on page 7)

[Fedak 2015] G. Fedak. *Contributions to Desktop Grid Computing : From High Throughput Computing to Data-Intensive Sciences on Hybrid Distributed Computing Infrastructures*. 2015. (Cited on page 70)

[Ganaputra and Pardamean 2015] J. Ganaputra and B. Pardamean. Asynchronous Publish/Subscribe Architecture over WebSocket for Building Real-time Web Applications. *Internetworking Indonesia Journal* 7 (Dec. 2015). (Cited on page 38)

[Gao et al. 2019] X. Gao, Z. Gu, Z. Li, H. Jamjoom, and C. Wang. Houdini's Escape: Breaking the Resource Rein of Linux Control Groups. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pages 1073–1086. (Cited on page 20)

[Gilbert and Lynch 2012] S. Gilbert and N. Lynch. Perspectives on the CAP Theorem. *Computer* 45.2 (Feb. 2012), pages 30–36. (Cited on page 8)

[Grasberger et al. 2013] H. Grasberger, P. Shirazian, B. Wyvill, and S. Greenberg. A Data-Efficient Collaborative Modelling Method Using Websockets and the BlobTree for over-the Air Networks. In: *Proceedings of the 18th International Conference on 3D Web Technology*. Web3D '13. San Sebastian, Spain: Association for Computing Machinery, 2013, pages 29–37. (Cited on pages 69, 70, 73)

[Han et al. 2011] J. Han, H. E, G. Le, and J. Du. Survey on NoSQL database. In: *2011 6th International Conference on Pervasive Computing and Applications*. 2011, pages 363–366. (Cited on page 8)

[Hasselbring 2021] W. Hasselbring. Benchmarking as Empirical Standard in Software Engineering Research. In: *Evaluation and Assessment in Software Engineering*. EASE 2021. Trondheim, Norway: Association for Computing Machinery, 2021, pages 365–372. (Cited on pages 6, 13)

[Hasselbring et al. 2020] W. Hasselbring, A. Krause, and C. Zirkelbach. ExplorViz: Research on software visualization, comprehension and collaboration. *Software Impacts* 6 (2020). (Cited on pages 1, 10)

[Hastings 1990] A. Hastings. Distributed lock management in a transaction processing environment. In: *Proceedings Ninth Symposium on Reliable Distributed Systems*. 1990, pages 22–31. (Cited on page 47)

[Henning and Hasselbring 2021] S. Henning and W. Hasselbring. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research* 25 (2021), page 100209. (Cited on page 62)

[Hightower et al. 2017] K. Hightower, B. Burns, and J. Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. 1st. O'Reilly Media, Inc., 2017. (Cited on page 9)

[Kabakus and Kara 2017] A. T. Kabakus and R. Kara. A performance evaluation of in-memory databases. *Journal of King Saud University - Computer and Information Sciences* 29.4 (2017), pages 520–525. (Cited on pages 8, 43)

[Kounev et al. 2020] S. Kounev, K.-D. Lange, and J. von Kistowski. *Systems Benchmarking. For Scientists and Engineers*. 1st edition. Springer International Publishing, 2020. (Cited on page 5)

[Krause-Glau et al. 2022a] A. Krause-Glau, M. Bader, and W. Hasselbring. Collaborative Software Visualization for Program Comprehension. In: *2022 Working Conference on Software Visualization (VISSOFT)*. Limassol, Cyprus, 2022, pages 75–86. (Cited on pages 1, 11)

[Krause-Glau et al. 2022b] A. Krause-Glau, M. Hansen, and W. Hasselbring. Collaborative program comprehension via software visualization in extended reality. *Information and Software Technology* 151 (2022), page 107007. (Cited on page 10)

[Krause-Glau and Hasselbring 2022] A. Krause-Glau and W. Hasselbring. Scalable Collaborative Software Visualization as a Service: Short Industry and Experience Paper. In: *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 2022, pages 182–187. (Cited on pages 1, 10, 11, 14)

[Kung and Robinson 1981] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6.2 (June 1981), pages 213–226. (Cited on page 70)

[Lamport 1978] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21.7 (July 1978), pages 558–565. (Cited on page 70)

[Leach et al. 2005] P. J. Leach, R. Salz, and M. H. Mealling. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. July 2005. (Cited on page 45)

[Liu and Sun 2012] Q. Liu and X. Sun. Research of Web Real-Time Communication Based on Web Socket. *Int'l J. of Communications, Network and System Sciences* 5 (2012), pages 797–801. (Cited on page 6)

[Liu et al. 2023] S. Liu, A. Kuwahara, J. J. Scovell, and M. Claypool. The Effects of Frame Rate Variation on Game Player Quality of Experience. In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. CHI '23. Hamburg, Germany: Association for Computing Machinery, 2023. (Cited on page 15)

[Maalej et al. 2014] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke. On the Comprehension of Program Comprehension. *ACM Trans. Softw. Eng. Methodol.* 23.4 (Sept. 2014). (Cited on page 1)

[Macedo and Oliveira 2011] T. Macedo and F. Oliveira. *Redis Cookbook: Practical Techniques for Fast Data Manipulation*. O'Reilly Media, Inc., 2011. (Cited on pages 7, 8)

[Majthoub et al. 2018] M. Majthoub, M. H. Qutqui, and Y. Odeh. Software Re-engineering: An Overview. *2018 8th International Conference on Computer Science and Information Technology (CSIT)* (2018), pages 266–270. (Cited on page 33)

[Marion and Jomier 2012] C. Marion and J. Jomier. Real-Time Collaborative Scientific WebGL Visualization with WebSocket. In: *Proceedings of the 17th International Conference on 3D Web Technology*. Web3D '12. Los Angeles, California: Association for Computing Machinery, 2012, pages 47–50. (Cited on pages 69, 70)

[Menascé 2002] D. A. Menascé. Load Testing of Web Sites. *IEEE Internet Computing* 6.4 (July 2002), pages 70–74. (Cited on page 13)

[Mouton et al. 2011] C. Mouton, K. Sons, and I. Grimstead. Collaborative Visualization: Current Systems and Future Trends. In: *Proceedings of the 16th International Conference on 3D Web Technology*. Web3D '11. Paris, France: Association for Computing Machinery, 2011, pages 101–110. (Cited on page 69)

[Nielsen 1994] J. Nielsen. *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. (Cited on page 19)

[Nishino et al. 1999] H. Nishino, K. Utsumiya, K. Korida, A. Sakamoto, and K. Yoshida. A Method for Sharing Interactive Deformations in Collaborative 3D Modeling. In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '99. London, United Kingdom: Association for Computing Machinery, 1999, pages 116–123. (Cited on page 69)

[Oyetoyan et al. 2015] T. D. Oyetoyan, J.-R. Falleri, J. Dietrich, and K. Jezek. Circular dependencies and change-proneness: An empirical study. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2015), pages 241–250. (Cited on page 40)

[Pezoa et al. 2016] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč. Foundations of JSON schema. In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2016, pages 263–273. (Cited on page 36)

[Pham 2020] A. D. Pham. Developing Back-End of a Web Application with NestJS Framework: Case Study of Integrify Oy's Student Management System. Bachelor's Thesis. Lab Universtiy of Applied Sciences LTD, Autumn 2020, page 46. (Cited on page 42)

[Razina and Janzen 2007] E. Razina and D. Janzen. Effects of Dependency Injection on Maintainability. In: *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*. SEA '07. Cambridge, Massachusetts: ACTA Press, 2007, pages 7–12. (Cited on page 43)

[Singhal 1989] M. Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers* 38.5 (1989), pages 651–662. (Cited on page 35)

[Smith and Williams 2002] C. Smith and L. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison Wesley Longman Publishing Co., Inc., 2002. (Cited on page 5)

[Tilkov and Vinoski 2010] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing* 14 (2010), pages 80–83. (Cited on page 43)

[V. Kistowski et al. 2015] J. v. Kistowski, J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, and P. Cao. How to Build a Benchmark. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: Association for Computing Machinery, 2015, pages 333–336. (Cited on page 5)

[Vogels 2009] W. Vogels. Eventually Consistent. *Commun. ACM* 52.1 (Jan. 2009), pages 40–44. (Cited on pages 41, 42)

[Yang et al. 2008] H. Y. Yang, E. D. Tempero, and H. Melton. An empirical study into use of dependency injection in java. In: *19th Australian Software Engineering Conference (ASWEC 2008), March 25-28, 2008, Perth, Australia*. IEEE Computer Society, 2008, pages 239–247. (Cited on page 42)

[Zirkelbach et al. 2018] C. Zirkelbach, A. Krause, and W. Hasselbring. On the Modernization of ExplorViz towards a Microservice Architecture. In: *Software Engineering*. 2018. (Cited on page 38)