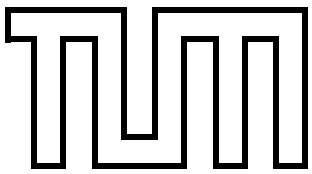


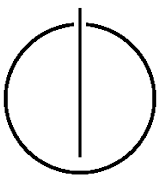
FAKULTÄT FÜR INFORMATIK

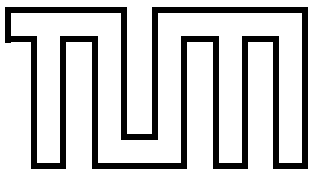
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Securing and Scaling Artemis WebSocket Architecture

Simon Leiß





FAKULTÄT FÜR INFORMATIK

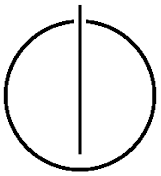
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Securing and Scaling Artemis WebSocket Architecture

Absicherung und Skalierung der WebSocket-Architektur in Artemis

Author:	Simon Leiß
Supervisor:	Prof. Dr. Bernd Brügge
Advisor:	Dr. Stephan Krusche
Date:	15.08.2020



I assure the single handed composition of this bachelor thesis only supported by declared resources,

Munich, 15.08.2020 Simon Leiß

Acknowledgements

I wish to show my gratitude to everyone who has supported me within the last months with this thesis. First, I want to thank my advisor Dr. Stephan Krusche, who allowed me to write this thesis with Artemis, always supported me, and was open to my ideas. He provided me with valuable feedback and showed great interest in my thesis, despite his many additional responsibilities.

Further, I want to thank the whole Artemis developer team for a great and fun time during the work on my thesis. They supported me wherever possible and also provided me with feedback.

I want to thank everyone who helped me implement the changed deployment of Artemis, especially Linus Michel, Matthias Linhuber, Christian Femers, Robert Jandow, Vincent Picking, and Jan Philip Bernius. They spent countless hours of work to implement the changes proposed in this thesis.

On a more general note, I want to thank my family and friends for guiding me through the last months. Their support was precious and without them, I would not have been able to be work on this thesis as I could.

Abstract

Artemis is an interactive learning platform that allows students to solve quiz, text, modeling, and programming exercises. Students receive automatic, individual feedback for programming and quiz exercises and receive manual feedback for the other exercise types. As an increasing number of students use Artemis, scalability and fault-tolerance become essential, especially for examinations that take place using Artemis.

Scaling vertically by adding more resources is not applicable beyond a certain point, thus a horizontal scaling approach has to be implemented. Security checks for real-time communication are limited. In this thesis, we scaled Artemis to multiple virtual machines and improved the security for real-time communication. We moved Artemis from one virtual machine that hosts all subsystems, including the database server and the load balancer, to a deployment on 14 virtual machines. This improved redundancy and performance and separated different parts of the system. We introduced additional subsystems such as a discovery service that are required when moving a web application from one instance to a distributed system.

Instructors of five courses were able to conduct over 2,500 individual exams, with over 1,200 students participating in the largest one. We could intercept failures of the system and fulfill performance requirements, even with more than 2,300 concurrently connected users. We also optimized the existing real-time communication to be less resource-intensive and more secure by grouping messages and enforcing security checks.

Zusammenfassung

Artemis ist eine interaktive Lernplattform, die es Studenten ermöglicht, Quiz-, Text-, Modellierungs- und Programmieraufgaben zu bearbeiten. Studenten erhalten automatische, individuelle Rückmeldungen zu Programmier- und Quizaufgaben und erhalten manuelle Rückmeldungen für die anderen Aufgabentypen. Da Artemis von einer größer werdenden Zahl von Nutzern in Anspruch genommen wird, ist Skalierbarkeit sowie Ausfallsicherheit ein wichtiges Anliegen, insbesondere, da auch Prüfungen über Artemis durchgeführt werden.

Vertikale Skalierung durch Hinzufügen weiterer Ressourcen ist nicht unbegrenzt anwendbar, daher muss horizontale Skalierung angewendet werden. Sicherheitskontrollen für die eingesetzte Echtzeitkommunikation waren bisher limitiert. In dieser Arbeit haben wir Artemis haben wir Artemis auf mehrere virtuelle Maschinen skaliert und die Echtzeitkommunikation abgesichert. Wir haben Artemis von einer einzelnen virtuellen Maschine, die alle Teilsysteme wie den Datenbankserver und den Load Balancer betrieben hat, auf eine Architektur auf 14 virtuelle Maschinen umgezogen. Diese Änderung verbessert die Redundanz und Leistung und separiert die einzelnen Teile des Systems. Wir haben zusätzliche Komponenten wie einen Discovery-Dienst hinzugefügt, die durch den Umzug einer Web-Anwendung auf eine verteilte Architektur nötig werden.

Instruktoren von fünf Kursen konnten über 2500 individuelle Klausuren abhalten, wobei über 1200 Studenten an der größten Klausur teilnahmen. Wir konnten Fehler im System abfangen und die Leistungsanforderungen erfüllen, auch mit mehr als 2300 gleichzeitig verbundenen Nutzern. Wir haben außerdem die Echtzeitkommunikation durch Gruppierung von Nachrichten und erweiterten Sicherheitskontrollen verbessert, damit sie weniger Ressourcen verbraucht und sicherer ist.

Contents

ASE Chair for Applied Software Engineering

CIS Continuous Integration Server

NFS Network File System

ORM Object/Relational Mapping

REST Representational State Transfer

SQL Structured Query Language

STOMP Simple Text Oriented Messaging Protocol

TLS Transport Layer Security

UMS User Management Server

VCS Version Control Server

VPN Virtual Private Network

WS WebSocket

Chapter 1

Introduction

Artemis is an online platform developed at the chair for Applied Software Engineering at Technical University Munich (TUM) [KS18]. It allows students to solve exercises created by instructors and receive individual feedback based on manual, semi-automatic, and automatic assessment. Students can solve the exercises individually or in teams, following the approach of interactive learning [KSB+17]. Artemis supports text, quiz, modeling and programming exercises, allowing for widespread use, especially in computer science courses. As many students use Artemis, problems regarding the performance occurred in the last semesters. In the winter semester 2019/20, more than 1500 students used Artemis concurrently before deadlines in large courses like the practical course „Fundamentals of Programming“[1], leading to performance issues.

The expected higher number of students using the platform in upcoming semesters inevitable forces to implement ways to improve the performance using both scaling and improving the current usage of resources.

This thesis focuses on the required horizontal scaling of the Artemis application server and subscription management for real-time communications.

1.1 Problem

Figure 1.1 shows the current setup with one application server that handles all requests of the clients. The application server uses a database server to persist data. This deployment scales not adequate as more users will higher the load on the server, therefore decreasing the performance of the server as well as the satisfaction of the users.

[1] Praktikum: Grundlagen der Programmierung (PGdP)

Scaling vertically can increase the performance of the system [LSL+14]. In this approach, administrators add more (hardware) resources (like more memory or faster CPUs). It does not require changes to the system's design as only the hardware of existing subsystems is updated. In the current deployment, scaling the *Artemis Application Server* would be sufficient. The gain is limited as, at some point, it is not useful or possible to add more resources to the system. Also, vertical scaling does not improve the redundancy in case of a server failure or maintenance.

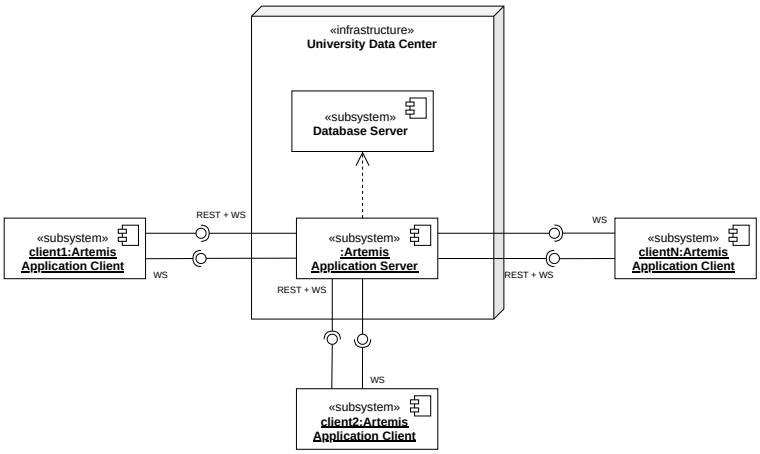


Figure 1.1: Current deployment with one application server, multiple application clients, and their interfaces. Clients communicate with the application server using REST and WebSockets (WS), which stores data into the database server.

Horizontal scaling increases the performance of a system by adding more instances of the application server. This technique is especially suitable for client/server architectural styles and three-tier architectural styles [BD09]. As Artemis uses the latter one, changes proposed in this thesis can be applied. It requires additional work during system design as the system has to support a distribution over multiple (virtual) machines.

Clients and server communicate using REST and WebSockets. A client uses REST[2] to perform operations at the server and processes the result. The server then stores the data in the storage layer, making Artemis use a three-tier architectural style.

[2]Representational State Transfer, an architectural style for distributed hypermedia systems [FT00]. REST is based on the client-server architectural style, is stateless and cachable, provides an uniform interface and supports a layered system style to allow for scaling.

The server can not send data to the client but must rely upon the client fetching updates using a pull-approach [FM11].
Artemis uses WebSockets[3] bidirectionally whenever the server has to send information to the client without a request from the client. Clients can subscribe to specific topics and the server notifies them as soon as new data regarding the subscribed topic is available, therefore using a push-approach [Lom15; WSM13].

Components that rely on fast bidirectional communication (e.g., live quizzes) use WebSockets, whereas tasks that do not require real-time updates (e.g., exercise creation) use REST calls. Due to the statelessness of REST, scaling the REST-API is comparatively simple as the server does not hold any information about the client but will use external data storage like databases [Mas11]. As Artemis uses WebSockets in cases where it holds information about the client internally (like in caches or the file system), scaling WebSockets is more complicated. Multiple servers must synchronize the internally stored data between each other once horizontal scaling is in place because clients are not guaranteed to connect to the same server at all times.

The high number of users causes performance issues that are further increased as in some cases, clients subscribe to topics they do not require as well as several similar topics. In the current setup, to get informed about new results for an exercise, the client of an instructor will create one WebSocket subscription for every student that participates in that given exercise. In large courses with several instructors, the instructors cause multiple thousand subscriptions. One subscription per instructor is sufficient if the server sends the updates not per student but exercise. The server then sends all new submissions for this exercise to the instructor using the same subscription.

Additionally, clients subscribe to topics where only a small part of the information is relevant for them. The server requires additional resources to manage these subscriptions, although only a very restricted group of clients needs this information.

Besides the additional need for resources, the current behavior can also cause problems in the range of data protection. Although the connection between the client and the server is encrypted using Transport Layer Security (TLS) [Erk12], issues can arise. TLS secures the integrity of the data transmitted using WebSocket connections, therefore the data can not be read or altered by a third party.
Encrypting the data does not ensure that the client receiving the data is

[3]WebSockets, based on the WebSocket protocol, are bidirectional messaging channels to communicate between client and server [FM11].

authorized to do so. Modified clients could use the data sent to them and extract unintended information. This especially applies to results of exercises within Artemis, as a maliciously modified client could receive results from other students.

1.2 Motivation

Figure 1.2 shows the aimed deployment after we implemented the changes proposed in the thesis. The system's performance improves by adding more instances of the application server to distribute the load. This diagram focuses on the central newly added subsystems (load balancer and several instances of the application server) and omits some additional subsystems required for this setup.

The clients are not connected to the same instance of Artemis but are connected to different instances as the load balancer delegates the requests.

In the aimed deployment, server administrators can perform maintenance work with less impact on the users. They can perform several tasks (such as updating Artemis) on the instances separately, allowing for higher availability. The redundancy also creates better fault tolerance, as a failure of one instance of the application server does not affect the whole system.

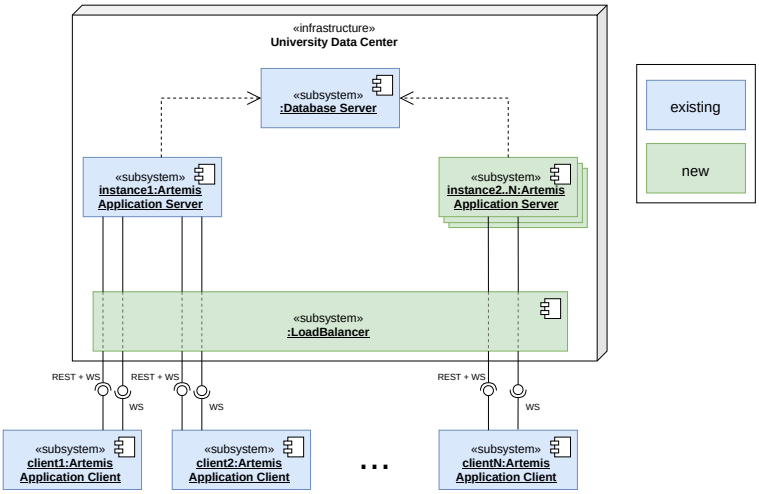


Figure 1.2: Aimed deployment with several instances of the application server with respect to the offered interfaces (REST, WebSocket (WS)) and the *LoadBalancer* subsystem. The LoadBalancer distributes requests from application clients to different instances of the application server.

The better usage of resources with regards to WebSocket subscriptions also increases the system's performance as fewer subscriptions have to be managed by the server. We will introduce a way to authenticate and authorize clients using WebSocket subscriptions based on topics to reduce the number of unnecessary subscriptions [MNS+88].

1.3 Objectives

During the research, we will analyze and design how we can scale Artemis horizontally based on existing solutions for the used Java framework Spring boot[4]. We will add additional instances of the Artemis application server in order to distribute the load. After implementing the proposed changes, we will validate that Artemis can fulfill the requirements, especially in terms of performance and availability.

Concerning the implementation, we will focus on a simple but effective way to distribute the workload over several servers. Once established, it should allow developers to implement their features at a reasonable expense, although the architecture of Artemis changed. We will evaluate multiple broker platforms (like Apache Kafka[5] and Redis[6]) during the research and will implement different ways to synchronize the application servers depending on the existing use cases to ensure efficient and reliable behavior.

Therefore, we are going to update the current WebSocket usage to the new architecture, including authorization and work out on how to both secure new WebSocket connections to prevent unauthorized entities from accessing data and designing the WebSocket architecture in a scalable way.

1.4 Outline

We moved a web-application, Artemis, from a single instance to multiple instances and focused on the problems that might arise when doing so, especially in terms of real-time communication (using WebSockets).

We focused on three levels of synchronization between multiple instances of the application server:

1. REST-calls using a shared database and shared cache

[4] `https://spring.io/projects/spring-boot`
[5] `https://kafka.apache.org/`
[6] `https://redis.io/`

2. WebSocket-communication using a broker to relay messages

3. file system using a shared file system

The system's existing functionality should not be modified, but the system should support multiple instances of the application server to increase performance and decrease failures. Administrators of the system should manage the system with comparable effort, although it is now distributed over several virtual machines. An advanced monitoring system aids them in doing so.

The existing WebSocket communication should be improved to be less resource-intensive and more secure.

Chapter 2

Background

This chapter focuses on background information that is useful to understand the concepts applied in this thesis.

2.1 Scaling

Scaling, in general, allows a system to adapt to changing requirements. This can be required, e.g., due to an increasing user base of a system, which causes more load on the existing system. This leads to the system getting unable to handle the load within the expected performance requirements. Administrators want to ensure that their system is reachable for an increased user base. Therefore, they need to increase the resources the system can use. There are, in general, two approaches to do this, using either vertical or horizontal scaling [DGV+12].

Vertical scaling

Vertical scaling keeps the systems architecture untouched and only modifies the existing machines to use more resources [KAEC+18]. This technique is especially useful when using virtual machines, as administrators can usually increase their resources like CPU count or memory size with short notice. Some virtualization techniques also allow the change of these allocated resources without restarting the virtual machine, thus allowing a highly adopting solution.

As vertical scaling does not add new components to the system but only modifies existing ones, the software running on the system usually does not need to be adapted to the scaled environment. The system runs on the same deployment as before but uses additional resources, ensuring that it is better

capable of fulfilling the requirements.

Figure 2.2 shows an example of how the system's resources improve by updating the system's underlying hardware. After scaling the system twice, the system now has four CPUs and 16 GB of memory instead of one CPU and two GB of memory from the beginning. The costs of running the system change with the number of resources the system has, thus a small system is cheaper than a larger system.

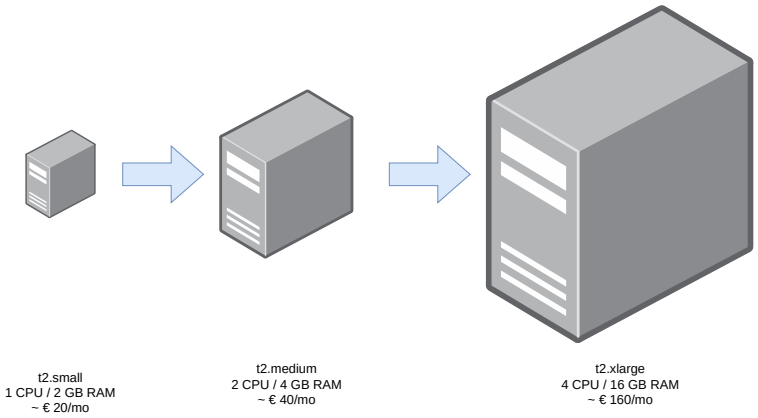


Figure 2.1: Vertical scaling of a system by using hardware with more/faster resources. Costs are exemplary prices taken from Amazon Web Services for one month for t2.small, t2.medium and t2.xlarge EC2 instances.

Decreasing the load on the system, reduces the needed resources of the (virtual) machine. This helps limiting the costs of running the system

Horizontal scaling

Horizontal scaling modifies the system's deployment by adding more instances of an existing application [KAEC+18]. In most cases, these instances can be added and removed on demand in order to handle load peaks. As the system's deployment changes, the software must be adjusted to support the new distributed system.

According to van Steen and Tanenbaum, a distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system [VST17].

We thus want that all instances of the application are be able to handle the same requests so that an additional component, like a load balancer, can forward the requests to one of the instances without applying additional logic.

This requirement may, depending on the application, require changes as not all systems support adding new instances. They might e.g., store data in the memory that is now only accessible by one instance and not other instances. Thus, additional synchronization between multiple instances may be needed, which the application must implement.

Figure 2.2 shows how the system's resources improve by adding additional machines that run the application. As the system now consists of multiple machines, the cost of running the system increases. The system, in this example, now also consists of four CPUs and has eight GB of memory available.

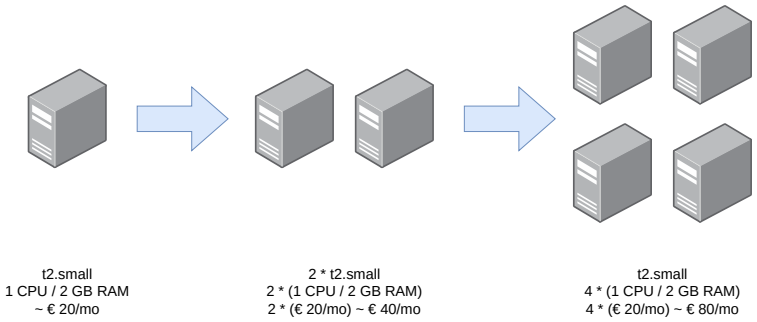


Figure 2.2: Horizontal scaling by adding more machines to the system. Costs are exemplary prices taken from Amazon Web Services for one month for t2.small EC2 instances.

One significant benefit of a horizontal scaling approach is that, in general, instances can be added/removed with less interruption than applying vertical scaling. Also, vertical scaling is not applicable beyond a certain point, as all resources that the system should use must still be available on the underlying hardware. In contrast, the resources can be split over several hardware nodes when using horizontal scaling. Horizontal scaling also allows for easier failure-tolerance: The other instances can handle an outage of one instance of the system if administrators configured the system in such a way.

2.2 Caching

To prevent unnecessary requests to the database server, Artemis uses caches to store data in the memory that the system might use again. Otherwise, the data has to be fetched from the database server multiple times. This improves the system's performance as loading data from the cache (memory) is faster than fetching it from the database (possibly over the network) [JF11].

As Artemis should use the cache extensively to provide performance improvements, the cache acts as another source of truth (next to the database) where only data is stored that is identical to the data stored in the database.

Distributed Caching

The cache that Artemis currently uses is present per instance after horizontal scaling, thus different instances of Artemis have different caches with different data stored in these caches. When using multiple instances, the additional source of truth that the cache should provide is not guaranteed.

If one instance of the system performs an update to both the database and their local cache, everything seems fine from this instance's perspective. A second instance that also uses a local cache as an additional source of truth will not receive a notification about the change the first instance applied to the database. Thus the data stored in the local cache of the second instance and the database deviates. As the second instance does not know about the inconsistent state of the data it uses, it assumes that the data in the local cache is still valid.

This leads to issues as now several instances of the horizontally scaled system do not behave in the same, way as a different version of the data is present in the different instances.

One solution to solve this issue, apart from disabling the cache, which decreases the performance, is the use of a distributed cache. This cache is present on all instances of the scaled system and shares the same data, thus preventing the existence of different versions of stored data. Once one instance applies changes to the database (and therefore also in their local cache), it will broadcast the change to other instances. They then apply the change in their local cache.

There exist different types of distributed caches that also use different types of architectural styles [KNO+02; YHM00]. One common style is the client/server architectural style, where every instance of the scaled system acts as a client and communicates with an additional server that manages all clients. Another typical style is the peer-to-peer architectural style, where all instances connect to each other. They send updates directly from one instance to all other instances because in this style, every instance acts as the client and also as the server [BD09].

2.3 Real-time communication

Artemis uses real-time communication to send data from the client to the server and vice versa without opening a new HTTP-connection. The connection stays open and both parts (client & server) can send data to the other part to which it responds.

Real-time communication is essential for exchanging messages from the server to the client. The server is not able send data without request from the client. Artemis sends, e.g., results from the server to the client in real-time so that the client can display updated results without users having to refresh the application manually. Another use case is the publishing of a new exercise, which a client should know about so that it can show it to the user. This is especially relevant for quiz exercises as they are generally limited in their duration (less than 10 minutes in most cases). In order to support students to use the whole duration for the exercise, quizzes start automatically. Real-time communication ensures that the students can participate in an exercise without having to reload the application.

Another common use case is the visualization of the feedback of programming exercises as soon as the Continuous Integration Server (CIS) executed the tests. Students then use the feedback to rework their submission and achieve a better performance.

WebSocket

WebSockets are the technical concept that Artemis uses to implement real-time communication.

WebSockets were standardized in 2011 in RFC 6455 and allow bidirectional communication between client and server/broker using TCP [FM11]. They allow exchanging arbitrary messages as long as both sender and receiver can interpret the message.

To use WebSockets, clients create an HTTP-connection, which they upgrade to a WebSocket-connection if the server supports WebSockets [SHS14].

STOMP

Although WebSockets support any protocol in general, a standard protocol used in combination with WebSockets is the STOMP protocol.

STOMP[1] is a *Simple Text Oriented Messaging Protocol* that allows clients to communicate with so-called message brokers to exchange messages independent of programming languages or platforms [Sto]. It is a protocol with

[1] `https://stomp.github.io/`

only a small amount of different operations (compared to other protocols like AQMP[2]).

The messages are sent bidirectionally between clients and brokers; there are implementations for a broad set of different clients and brokers.

Table 2.1 shows typically used commands. The *CONNECT* command is used by the client to initiate a connection to the server and, upon accepting the connection, the server will respond with *CONNECTED*.

The client can subscribe to updates (new messages) of specific destinations using the *SUBSCRIBE* command. This indicates to the server that it should send all updates regarding that particular destination to the client. The server can do so using the *SEND* command. The client can also use the *SEND* command to send messages to specific destinations on the server.

Command	**Parameter**	**Usage**
CONNECT		The client initiates a connection with the server
CONNECTED		The server accepts the connection from the client
SUBSCRIBE	destination	Subscribe to updates of the given destination
SEND	destination	Send a message to the given destination

Table 2.1: Selected STOMP commands with selected parameters. The client can connect to the server, which the server acknowledges. The client can then subscribe to topics and both client and server can send arbitrary messages.

[2]Advanced Message Queuing Protocol, see `https://www.amqp.org`

Chapter 3

Requirements Analysis

With respect to requirements, the main goal of this thesis is to keep the existing functionality. It is not the goal to introduce new functionality in terms of new requirements.

Figure 3.1 shows that clients connect to the Artemis server. This application server currently only exists once, causing a single point of failure if this one instance, for whatever reason, becomes unavailable. This is critical during lectures with live interactive exercises (e.g., quizzes). An interruption in the availability can interfere with the lecture's planned schedule and can negatively influence the students' learning experience.

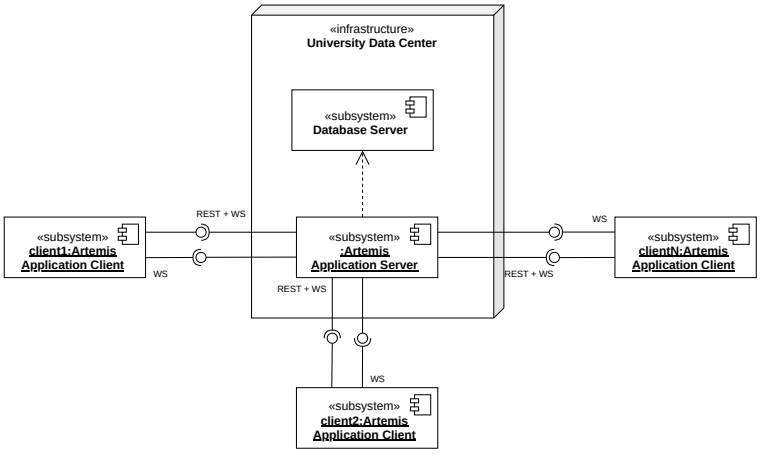


Figure 3.1: Current deployment with one application server, multiple application clients, and their interfaces. Clients communicate with the application server using REST and WebSockets (WS), which stores data into the database server.

As exams take place on Artemis in the summer term 2020 due to the corona crisis, instructors require that the system provides high availability. An outage during an exam can cause significant additional work. In the worst case, instructors can not grade the whole exam and need to seek different approaches to execute the examinations. Therefore, we plan to scale Artemis horizontally to provide the required failure tolerance in case one instance of the application server fails, e.g., due to a problem with the underlying hardware.

The performance of Artemis currently also depends on this one instance and once this one instance can not be scaled vertically any more, the performance is limited. This is critical as during large lectures and exams, up to 2,000 users can use the system at the same time, expecting a responsive system.

We also aim to improve the security and scalability of WebSocket connection as many WebSocket subscriptions cause high load on the system. Potential unsecured subscriptions can cause users to receive information to which they should not have access.

3.1 Current System

This section focuses on the current deployment of the system and the security checks currently in place for WebSocket connections.

3.1.1 Deployment

As already introduced in section 1.2, Artemis currently runs on one virtual machine that hosts both the application server and the database. Several external systems, like a Version Control Server (VCS) and a Continuous Integration Server (VCS), interact with this instance bidirectionally.

This system can handle several hundred students without issues. However, especially during live exercises with a high submission rate (like multiple choice quizzes, where every selected checkbox causes a new submission and therefore a request to the server), the system may have longer response times or might get unavailable. This impact can cause problems as users might not be able to interact with the system as fast as expected. Users who are unsatisfied with their experience will often reload the application, thus causing additional requests and a further increase in response times for all users.

Although Artemis has several hundred users connected every day during the semester (often over 700 concurrent users during regular working hours), these users often do not produce a high load on the system. Users often leave their computers running with the application open but do not interact with it. Artemis counts these users towards the user statistics, but they cause close to zero impact on the system.

During a lecture with live exercises, this situation changes quite drastically as now almost all students participating in the lecture interact with the system simultaneously. Artemis now has to deal with the increased load for the exercise duration and will then have fewer users again once the exercise is over.

Figure 3.2 gives one example: It shows the user statistic during the EIST ("Introduction to Software Engineering", the acronym for the German title of the course "Einführung in die Softwaretechnik") lecture that took place on July 16, 2020. One can identify the start of the lecture at 8.15 am as the lecture starts with a "good morning quiz" where every participating student has to open the Artemis application. As this quiz exercise only took a few minutes, the user count decreased from 1,380 users at 08.20 am to just over 1,000 connected users at 09.00 am. The students connected to Artemis interact with the system intensively as close to 900 students participated in the quiz, producing over 7,000 submissions in 5 minutes (as each student causes multiple submissions). The remaining users are students who study for other courses, students who leave the application running without interaction, or students who use multiple browsers.

3.1.2 Security

Another issue is that clients receive some information to which they should not have access. An unmodified client will not use this information (by not showing it to the user), but a maliciously modified client could leak data in some cases. This mainly affects WebSocket subscriptions: Currently, a client subscribes using a participation identifier if it expects data for this particular participation. This subscription ensures that users receive updates to their participation (e.g., new results) as soon as possible without reloading the application. While an unmodified client will only subscribe to the current user's participations, a modified client could also subscribe to the participations of other users. The server does not, in all cases, validate the legitimacy of the subscription and will allow clients to subscribe to participations of different users. Once the user who owns that participation receives a new result (e.g., by a manual assessment for text exercises or an automatic assess-

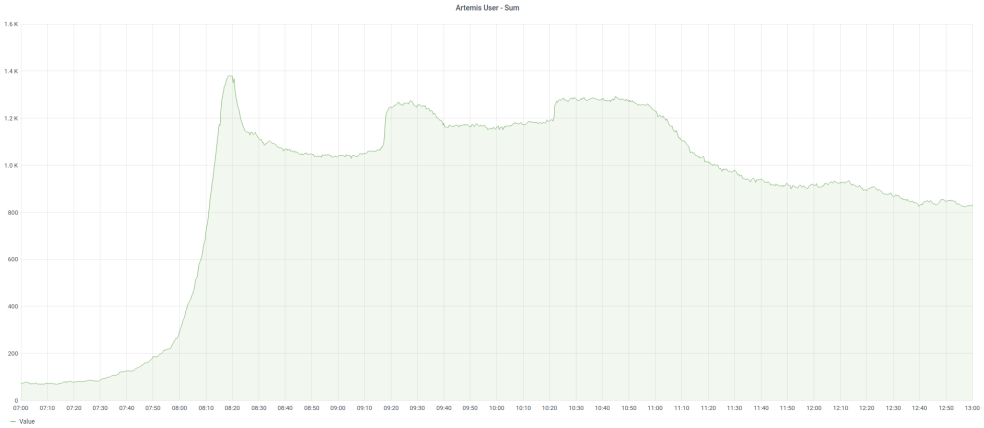


Figure 3.2: Statistic showing the user count during the EIST lecture on July 16, 2020. The user count increases from less than 200 at 08.00 am to over 1,300 at 08.15 am. The count decreases once the exercises are over, but decrease again when new exercises start, until the lecture ends at 11.00 am.

ment for programming exercises), all subscribed clients will be notified. As other clients can also subscribe to these new results, they can get unintended information.

3.2 Proposed System

The system should handle a higher number of users without noticeable performance impact for every individual user. We implement horizontal scaling as vertical scaling is only possible to a certain point, which we already reached [DGV+14]. This requires changes to the system and assumptions that hold in the current system do not hold anymore in the proposed system. We will identify these changed areas and rework them to support a distributed system.

We will also investigate the existing WebSocket communication and will detect scenarios with missing/insufficient authorization. Once we identify these scenarios, we will prioritize them in their severity and add authorization to the findings.

Figure 3.3 shows a more detailed view of the proposed system. Several instances of the application server handle requests from the clients, which a load balancer forwards to them. The instances communicate with the database to load data and use a shared cache, for which they directly com-

municate. Multiple instances use a discovery service to find each other. A broker relays WebSocket messages from one instance to another.

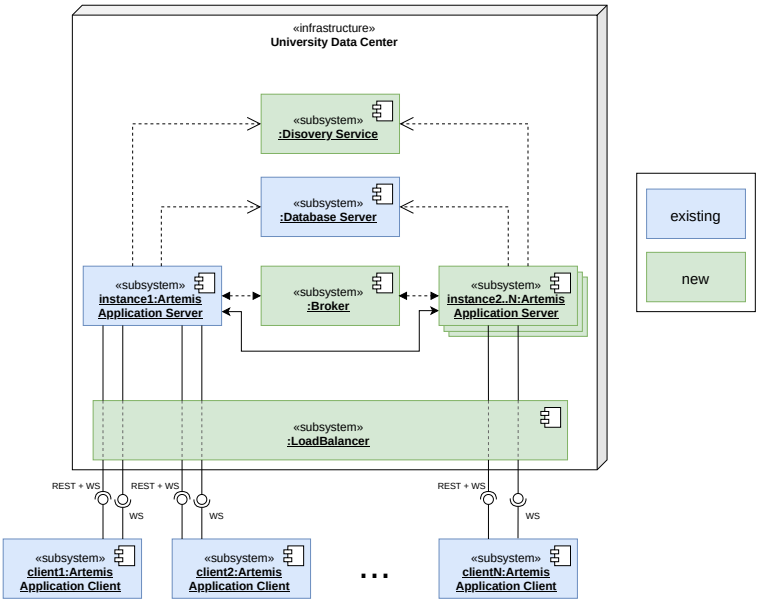


Figure 3.3: Aimed deployment with several instances of the applications server with respect to the offered interfaces (REST, WebSocket (WS)) and the subsystems *LoadBalancer*, *discovery service* and *Broker*. The LoadBalancer distributes requests to the different instances of the application server. The Broker allows the instances of the application server to exchange WebSocket messages. The discovery service provides information of currently running instances of the application server.

Nonfunctional Requirements

NFR1 **Performance** (Scalability): Artemis can deal with a higher number of users using horizontal scaling. New instances of the application are introduced, helping to distribute the load on the system. Artemis should be able to handle 10.000 concurrent users. Requests to save submissions during an exam should take less than 100ms for 95% of the users.

NFR2 **Reliability** (Availability): The system should automatically handle the failure of one subsystem of the application by passing all requests to available instances and re-balancing stored data. Artemis should be able to handle the outage of any of the subsystems and still provide full

functionality. This also affects the handling of scheduled tasks which should be executed even if one instance of the application server fails.

NFR3 **Reliability** (Security): Users can only subscribe to topics using WebSockets where they have sufficient permissions. Artemis will detect invalid requests and log them so that attacks can be detected.

NFR4 **Supportability** (Maintainability): Developers and administrators should still be able to deploy Artemis to a simplified setup that only consists of one machine, especially for development and testing environments, even after we implemented NFR1 & NFR2 and thus multiple instances are supported. Also, smaller deployments of the system, where the user base is smaller and therefore performance and reliability requirements are relaxed, should not be forced into an update that would require them to change their current deployment.

3.3 System Models

The focus of this part of this thesis lies on boundary use cases and non-functional requirements.

As this thesis does not implement any new functionality, the user interface is not modified. The existing functionalities should still be available after we implemented the changes proposed in this thesis.

3.3.1 Scenarios

The following scenarios show how Artemis should support a deployment with multiple instances. The scenarios do not propose new functionality but show that the existing functionality still works after moving Artemis from one instance to several instances.

Failure handling

Students can participate in quiz exercises in Artemis [Sch18]. Participants solve the quizzes, consisting of multiple-choice-, short-answer- and drag-and-drop-questions, during lectures (live quizzes) or personal study (practice quizzes) [CY19].

If the system fails during a live quiz, all submissions will be lost as they are not saved into the database before the quiz is over. The following scenario shows how a second instance can mitigate the failure of one instance

of Artemis.

Alice is registered in the EIST 2020 course and wants to join the morning quiz *L11E01 Quiz* that takes place during the lecture at 08.15 am on Thursday, July 16, 2020. She boots her Lenovo Thinkpad running Windows, opens Chrome, and navigates to the Artemis application at 08.10 am. As the quiz did not start yet, she has to wait until the quiz starts. Max, the instructor, decided that he wants to start the quiz manually and does so at 08.15 am using his MacBook Air running Safari. Joe is one of the administrators of Artemis and because Max and Joe did not communicate before, Joe does not know that a quiz is running and thus, several hundred students use Artemis. He wants to increase the amount of memory of one virtual machine that runs Artemis from 4GB to 6GB and reboots the virtual machine running instance 1 of Artemis at 08.17 am. Alice was connected to this instance of Artemis and worked on the quiz. She has already submitted answers to 5 of 10 questions. As the instance she was connected to is no longer available, she is automatically reconnected to instance 2 that runs on a different virtual machine. Instance 2 also has saved all submissions she made to the quiz so Alice can continue to answer the remaining questions without loss of progress.

Team exercises

Artemis supports exercises which are not solved individually but in teams [Wau20]. In these exercises, teaching assistants create teams and students that belong to the same team solve exercises together. They see the team's current submission to an exercise and can change it as long as the exercise is running. Every team member gets the grading of the exercise counted towards their score.

Artemis should still support team exercises even if the team members are not connected to the same instance, as shown below.

Alice and Bob are students enrolled in the EIST 2020 course. They are part of the same team and thus solve their team exercises together. On July 22, 2020, they realize that they have not yet solved the exercise *T05E03 Describe your Hardware Software Mapping*, which is a team text exercise where they have to describe the Hardware/Software Mapping of the game they developed during the EIST 2020 course. At 01.40 pm, Alice boots her Lenovo Thinkpad running Windows, opens Chrome, and navigates to the EIST 2020 course on Artemis. She starts the exercise as it is not yet started and begins to answer the question while being connected to instance 1 of Artemis. As Bob is not sure if Alice understood the concepts of Hardware/Software

Mapping correctly, he wants to check what Alice has already submitted in the exercise. He boots his Dell laptop running Ubuntu at 01.45 pm, opens Firefox, and navigates to Artemis, but the load balancer connects him to instance 2. As soon as he also opens the exercise, both Alice and Bob see that the other person is online and has opened the exercise. Bob verifies that Alice did not forget anything in their submission, corrects typos, and adds some details. Alice sees the changes Bob made despite being connected to a different instance.

3.3.2 Dynamic Model

Figure 3.4 shows a simplified communication procedure for a quiz exercise with two actors and two instances of the application server. Alice, the student, is connected to one instance of the application server and Max, the instructor, is connected to a different instance. Thus, no instance can directly send operations that one of the users performed to the other user. Max starts the quiz exercise and executes the quiz starts on the instance he is connected to. This instance relays the quiz start to the other instance so that users connected to a different instance can also participate in the quiz exercise.

Alice can then participate in the quiz exercise and send her submission to the instance she is connected to. This instance will now also relay the submission to the other instance. A failure of one instance can not cause a loss of the submission as two instances store the submission.

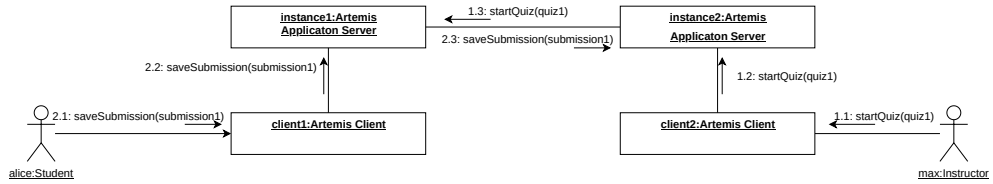


Figure 3.4: Communication diagram showing a simplified communication procedure during a quiz exercise. An instructor starts the quiz on one instance of the application server, which this instance relays to a second instance. A student then saves a submission on one instance of the application server using his/her client, which then replicates the submission to a second instance.

Chapter 4

System Design

We define the system's design goals and decompose it into smaller subsystems that can be realized by individual teams [BD09].

4.1 Overview

Artemis should support multiple instances of the application server to improve the reliability and performance for a large number of users and should use improved real-time communications regarding security and resource usage. It should not provide a changed behavior to the user as no functional requirements exist.

4.2 Design Goals

As described in section 3.2, we had to take several design goals into account for changing the current system design.

Online exams/graded exercises use the exam mode that requires a failure-tolerant system so that even in the unlikely case of a failure of one instance of the application server, the system is still available and no data gets lost. We, therefore, prioritize NFR2 (Reliability) so that Artemis can handle this kind of failure. NFR1 (Performance) will thus be less important as students can use a system that is a bit slower but will not use a system where we cannot guarantee the persistence of data and the availability of the system. NFR4 is closely related to NFR1 & NFR2, as NFR1 & NFR2 both focus on the scalability of the system (with different objectives) and NFR4 states that the system should still be usable without scaling. Thus, we require NFR4 if NFR1 or NFR2 is implemented, as a simple development and testing setup is crucial for a sound system.

NFR3 (Security) not only applies to the exam mode but also for every part of the application server that uses WebSockets and handles personal data. Due to time limitations, we focused on NFR1 and NFR2 (and thus also NFR4) to provide a failure-tolerant and fast exam mode. We decreased the importance of NFR3 because no exploits are known and Artemis uses WebSockets for only a limited set of operations that include students' data.

Another trade-off we had to decide on was also the deployment of the system (which we go into in section 6.1). We had to decide between machines that were available at short notice but have fewer resources and machines with more resources but higher provisioning times. We decided to use the machines that were available at short notice initially and use the newly acquired machines as soon as they are received and configured.

4.3 Subsystem Decomposition

The Artemis system, especially in a deployment with multiple instances, consists of 5 subsystems: application server, database, broker, load balancer, and discovery service. We discuss each subsystem in this chapter.

Application server

The application server contains the server-side logic related to the Artemis learning platform. It communicates with external systems like an external User Management Server and Version Control server, processes requests from clients and stores data into the database.

REST

The application server provides the REST API the client uses to show the information relevant for the current page. It fetches data from the database and writes changes to the database if the data is updated. If used in a multi-instance deployment, the application servers can answer the REST requests without further changes as REST is stateless, so every application server can always answer every request.

Caching

We use a cache within the application server so that the database does not have to serve all requests. This improves the system's performance as loading data from the cache is faster than fetching the data from an external subsystem like the database, as described in section 2.2.

As Artemis has to use a distributed cache to guarantee consistent data between several instances of the application server (see section 2.2), we had to take additional steps to support this setup. We decided to use a distributed cache based on a peer-to-peer architectural style. Thus we have to make sure that every instance of the application server can communicate with all other instances. All instances that share the same data form a so-called cluster and thus must be both known to and reachable from each other. Once all instances share the distributed cache, they can serve requests with comparatively little adjustments as the data in the database and the cache is synchronized. Developers working on the system should not need to take care of the cache manually. If the instances do not know or can not reach each other, they can not fully build the cluster, which leads to an inconsistent state.

A more detailed description of the cache changes is present in section 5.1.

Database

The database persists most of the data (except uploaded files) Artemis stores. The application server connects to the database and fetches, inserts, updates, and deletes data in the database using SQL[1]. Other subsystems do not interact with the database.

Broker

As clients can connect to any application server instance, only one instance can send messages to any specific client using WebSockets (the instance to which the client is directly connected). If one user performs an action that causes a message to a different client (e.g., an instructor starts a quiz which causes the client of the student to show the quiz automatically), only the instance to which the client is connected can send this message. An instructor connected to instance A can thus start the quiz, but a student connected to instance B will not receive the message that the quiz started as instance A can only send messages to clients that connected to itself.

The broker solves this issue as all instances of the application server relay the WebSocket connections to the broker. Suppose an instance of the application server tries to send a message to a user but fails to do so because the user is not connected to this instance. In that case, it can send the message to the broker instead, which then tries to deliver it to the user by sending it to the instance the user is connected to. The instance handling

[1]Structured Query Language, a way to manipulate data in a relational database [MS93].

the connection to this specific user can send the message to this client so that the message created by a different instance can still be delivered.

Figure 4.1 shows such a scenario where *instance1* uses the broker to deliver a message. *instance1* first tries to directly send the message to *client1* but is unable to do so as *client1* is not connected to *instance1*. *instance1* then sends the message to the broker, and the broker then forwards the message to the other connected instances. *instance2* receives the request to send the message to *client1* and can send the message as *client1* is connected to *instance2*.

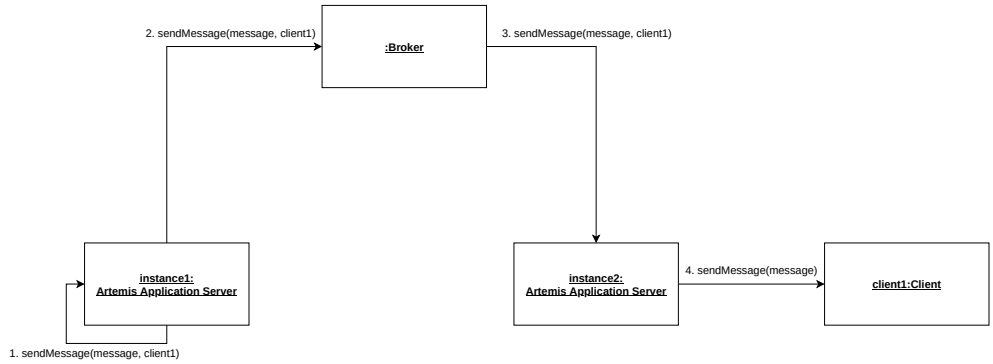


Figure 4.1: Communication diagram with a sample usage of the broker as relay: *instance1* tries to send a message to *client1*. *instance1* relays the message to the broker after failing to deliver the message directly. The broker forwards the message to *instance2*, which can deliver the message, as *client1*, as this client is connected to *instance2*.

Load Balancer

The load balancer terminates the HTTPS connection by decrypting the SSL/TLS-encrypted traffic [MHP12]. It then forwards the decrypted traffic to the instances of the application server by distributing the requests. Although the software that runs the load balancer is already part of the current deployment as it handles the mentioned SSL-termination, it is not used as a load balancing component but as so-called reverse proxy, which only handles the SSL-termination.

Discovery service

All application servers register themselves to the discovery service so that they can find each other. On startup, the application server sends a REST

request to the discovery service to inform it about the application start. It also requests all other instances that are already running and, if there are any, will connect to these instances to form a distributed caching cluster as described in section 4.3.

Figure 4.2 shows this startup process with two instances of the application server and the discovery service. The first instance registers itself so that the discovery service is aware of which host and port this instance is running on. The second instance then also starts and fetches currently running instances from the discovery service. The discovery service returns the host and port of the first instance. The second instance then establishes a connection to the first instance to form the distributed caching cluster.

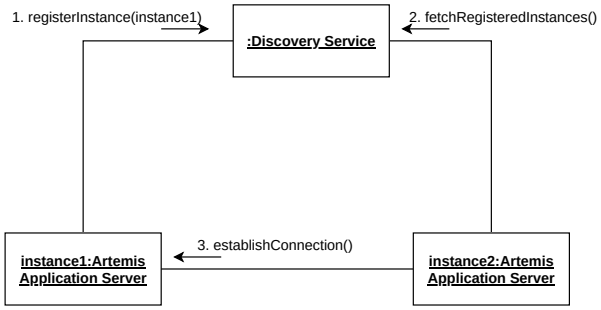


Figure 4.2: Startup process of two instances of the application server in combination with the discovery service. An instance registers itself to the discovery service during startup. A second instance fetches the registered instances and uses this information to establish a connection to the first instance.

4.3.1 Architectural style

The architectural style of Artemis deviates by applying the changes proposed in this thesis. The three-tier architectural style currently best represents Artemis, where a client runs in the user's web browser that allows the user to interact with the system. It thus is the *interface layer* as described by Bruegge and Dutoit [BD09]. The client communicates with the application server, which responds to the requests from the client and therefore is the *application logic layer*. The server persists the data into the database and file system, which act as *storage layer*.

The architectural style changes as now several subsystems exist that do not fit into the three-tier architectural style. Erb introduced a model of a scalable web infrastructure that can also be applied to Artemis. Figure 4.3

shows the model Erb introduced [Erb12]. It omits the client, thus only the former *application logic* and *storage* layers are displayed as well as the newly added components.

The updated architectural style of Artemis can be mapped to this model as follows: The *load balancer* introduced in this section is also present in the model but Artemis does not use *reverse caches* as the *HTTP servers* of the application servers serve all requests. The HTTP server Tomcat[2] is embedded within the application server and handles incoming requests before passing them to the application server's business logic [Erb12]. The application server contains the mentioned business logic and interacts with several other components. One of these components is the *storage backend*, which consists of the database and file system and was already present before as *storage layer*. Artemis integrates with *external services* like the Version Control Server (VCS), Continuous Integration Server (CIS), and the User Management Server (UMS), which were also already present before the change of the architectural style but not explicitly mentioned. The *Broker* can also be mapped to the model, as described in this chapter. *Cache Systems* were present before but are now in a more exposed role as the need for distributed caching requires changes as described in section 2.2.

Despite not being mentioned explicitly as *background worker services*, Artemis uses them, especially for scheduled tasks. The change of the architectural style also required changes to these *background worker services* as described in section 5.2.

Thus, we conclude that despite minor deviations, the updated architectural style of Artemis fits well into the style presented by Erb.

[2] `http://tomcat.apache.org`

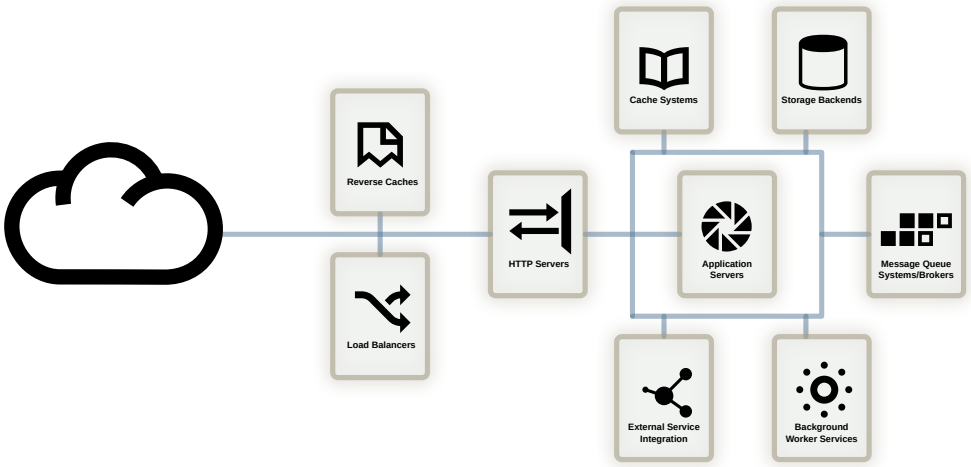


Figure 4.3: An architectural model of a web infrastructure that is designed to scale. Components are loosely coupled and can be scaled independently [Erb12].

4.4 Hardware/Software Mapping

Hardware/Software mapping describes the hardware configuration of the system [BD09]. Artemis uses a different hardware/software mapping as we move from a deployment with one instance of the application server to one with several instances.

In the current setup, one machine hosts all major subsystems that Artemis uses for its basic functionality. This includes the application server, the database server, and the load balancer that currently only provides SSL-termination. Different hardware hosts other subsystems like the User Management Server (UMS), the Version Control Server (VCS), and the Continuous Integration Server (CIS). Artemis communicates with these external subsystems via HTTP(S).

As required by NFR4, it should still be possible to deploy the system on a single machine. All features related to scaling are optional within Artemis, thus a simplified deployment is still possible, especially for testing and development environments and deployments with a smaller user base.

As this simplified deployment does not require the broker & discovery subsystems, administrators can omit the setup of these subsystems and only set up the application server, database server, and load balancer (which

only handles the SSL-termination and does not actually balance requests as described in section 4.3). Figure 4.4 shows this simplified setup.

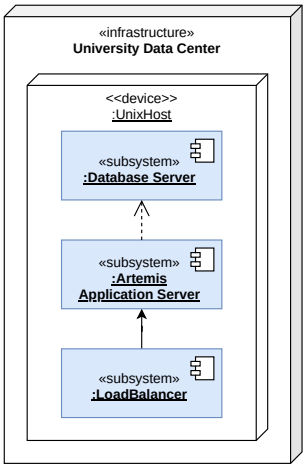


Figure 4.4: Simplified deployment on one machine, e.g., for a testing environment. The load balancer communicates with the application server, which uses the database as storage.

Figure 4.5 shows the aimed, more complex deployment concerning the used machines. Different machines run the different subsystems of Artemis, as described in this section. The figure does not show the client as this thesis focuses on scaling the application server. The client still communicates with the application server (through the load balancer) as figure 3.3 shows, but as the changes are transparent to the client, we will omit the client from the figure.

Database server

To allow for better scaling, we moved the subsystems that currently run on the same machine to different machines. We move the database to its own (virtual) machine to increase its performance. The resources of this machine do not have to be shared with other subsystems anymore. We can also scale this machine better vertically as we can adjust its resources for a database server and are not limited by running several services with different requirements on that one machine. If the database server would run on the same machine as the application server, increased resource consumption by the application server would also impact the performance of the database server. This particularly causes issues as we now run several instances of the application server. Therefore one instance could slow down all other instances as they rely on the database server.

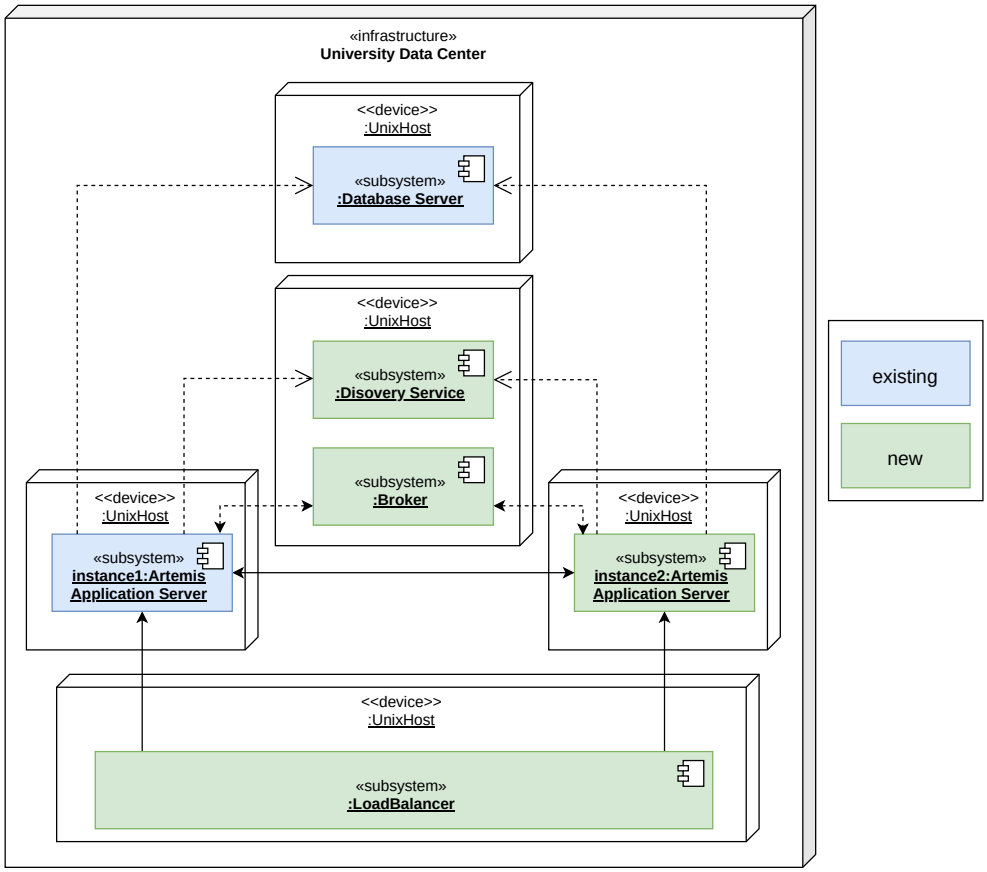


Figure 4.5: More complex deployment on several machines within the university's data center. We moved the existing load balancer and database server subsystems to own machines and deployed additional instances of the application server, the discovery service and broker subsystems on separate machines.

Load Balancer

The load balancer also runs on its own (virtual) machine to reduce the number of subsystems on one machine and allow for better scaling. As it handles all traffic to and from clients, the throughput of the network connection is essential. Memory and CPU are not as relevant as the load balancer is very resource-saving. However, it should support a large number of open files to handle a large number of users [3].

Artemis application server

Instances of the application server should be running on individual (virtual) machines so that they can neither affect each other nor different subsystems like the database server by having a high resource usage.

Administrators can assign different weights to the different instances of the application server so that instances with more abundant resources (like more CPU cores) get more requests than instances with less resources.

As the application server is mainly CPU intensive, the weight should depend on the available computation power rather than other resources like system memory.

A concrete setup can be seen in the case study in section 6.1.

Broker & Discovery service

One machine can host both the broker and the discovery service as the discovery service only requires minimal resources. The broker relays WebSocket messages between all instances of the application server. Thus it should have a reliable connection to the machines running the application server.

Encryption

As we use several different protocols for the communication between the different subsystems, we use a Virtual Private Network (VPN) tunnel between the machines. Therefore, we do not have to focus on different encryption methods for the protocols (including their key management) but can use a secured tunnel between the machines. Our decision was for WireGuard[4], a secure network tunnel that operates at layer 3 [Don17]. Every machine creates a tunnel to every other machine. Thus we do not have one server

[3] `https://medium.com/@cubxi/nginx-too-many-open-files-error-solution-for-ubuntu-b2adaf155dc5`

[4] `https://www.wireguard.com/`

that manages the connection, but every machine acts as a server for all other machines and also connects to all other machines as a client.

Only the load balancer is reachable from the public internet using the ports 80 (for HTTP) and 443 (for HTTPS). It forwards all traffic received on port 80 (unencrypted to port 443 (encrypted), therefore the request coming from the user will always be encrypted. The load balancer will then terminate the SSL/TLS encryption and forward these requests to one of the application server instances using HTTP. The traffic is no longer encrypted on layer 4 (SSL/TLS) but is still encrypted on layer 3 using the secure WireGuard tunnel. The selected application server will then e.g., interact with the database and broker (using the WireGuard tunnel) and responds with an answer through the tunnel to the load balancer. The load balancer then adds the SSL/TLS encryption to the data and sends it to the client.

4.5 Persistent Data Management

We moved the database server to a new machine, as we described in 4.4, but did not apply changes to the database schema.

We changed the storage of data in the file system due to the distributed deployment with multiple instances of the application server. The file storage hosts files like submissions for file upload exercises, icons for courses, and repositories students use in the code editor. Josias Montag implemented the online code editor for students in 2017 to work on a repository without having to checkout locally [Mon17]. The server will clone the repository from the Version Control Server and send the contained files to the client. The client then shows them to the user. The user updates, creates, and deletes files and folders using his/her web browser and, once finished, commits the changes. The server then applies the changes to the locally checked out repository and pushes them to the Version Control Server.

As the load balancer may connect users to different application server instances during the work on the repository in the online code editor, all instances have to share the file storage. This ensures that all instances have the same version of the files at any given time. Without sharing the storage, it would be possible that a user starts a programming exercise on one instance of the application server (thus, this instance checkouts the repository into the local file system), but the load balancer connects the user to a different instance after the checkout. The second instance then does not have the repository checked out, which leads to problems as the client of the user

expects the server to have the repository present.

More details of the implementation of the shared storage are present in section 5.3.

4.6 Access Control

Access control matrices show which actor can execute which operations on which context [BD09]. Table 4.1 shows to which topics administrators, instructors, teaching assistants, and students can subscribe. Artemis uses WebSockets subscriptions if the server should inform the client about updates, e.g., notifications or new results for exercises.

Instructors should have permissions regarding the creation and management of exercises as well as results. Teaching assistants should not receive updates about exercises but only about participations of students (and their results). Students should only be able to receive information relevant to them as it affects them personally (e.g., their result of an exercise they participated in) or affects all students (e.g., the notification that an instructor updated an exercise of a course in which they are enrolled). They should not be able to access any information that is related to other students.

Unmodified clients will not show the personal information of other students (e.g., their results), but maliciously modified clients can access it if the server does not implement the access control correctly.

Action	**Administrator**	**Instructor**	**Teaching Assistant**	**Student**
Receive personal notifications	✓	✓	✓	✓
Receive student notifications for course	✗	✗	✗	✓
Receive teaching assistant notifications for course	✗	✗	✓	✗
Receive instructor notifications for course	✓	✓	✗	✗
Receive system notification	✓	✓	✓	✓
Receive update about toggled features	✓	✓	✓	✓
Track open pages	✓	✓	✓	✓
Receive update about own team assignments	✓	✓	✓	✓
Receive automatically submitted modeling submission (specified by id)	✗	✗	✗	✓(Own submissions)
Receive updated test cases for solution repository of exercise	✓	✓	✗	✗
Receive updates about changed tests repository of exercise	✓	✓	✗	✗
Receive update that an instructor triggered a build for programming exercises	✓	✓	✗	✗
Receive statistic update for quiz exercise	✓	✓	✓	✗
Save quiz submission	✗	✗	✗	✓
Receive result of quiz exercise	✗	✗	✗	✓
Receive update about own programming submissions	✓	✓	✓	✓
Receive update about own programming results	✓	✓	✓	✓
Receive update about submissions of exercise	✓	✓	✓	✗
Receive update about results of exercise	✓	✓	✓	✗

Table 4.1: Access control matrix showing the subscription permissions for WebSocket topics.

4.7 Global Software Control

Global Software control describes the handling of synchronization and concurrency within the system [BD09]. The setup with multiple instances of the application server requires additional handling in the synchronization, especially for scheduled tasks.

When using multiple instances of one (sub-)system, issues may arise about ensuring that the system executes tasks *exactly* once. We do not want the system not to execute tasks at all, as then some parts of the application logic may behave incorrectly. We also do not want the system to execute tasks more than once. Not all tasks are necessarily idempotent and even if idempotent, several executions of the same task require additional resources.

We decided to use two different strategies for different use cases, which we describe in section 5.2.

4.8 Boundary Conditions

Boundary conditions focus on conditions that the system must handle but are not part of its key requirements, including start-up, shutdown, and exceptions [BD09].

This section focuses on the changed procedures that the administration of the distributed system requires.

4.8.1 Starting procedure

As we now have a distributed deployment, the starting procedure of the system gets more complicated. Figure 4.6 shows several dependencies between the subsystems. The application server depends on the database, the broker, and the discovery service (requiring them to start before itself). The load balancer depends on the application server. For the load balancer, it is sufficient if one of the application server's instances is reachable. It can detect if the other instances are unavailable and prevent routing traffic to them if they are. Also, the dependency between the load balancer and the application server is a soft dependency (represented by the dashed line). Despite failing requests from the client, the load balancer can handle a startup procedure where it is started before the application server as well. The load balancer sends an error message to the client, stating that the application server (so-called upstream) is not yet reachable.

As the instances of the application server form a cluster to support distributed caching, they need to know each other. The discovery service pro-

vides this information. In order to form a correct cluster, at least one instance has to be known to the discovery service once an additional instance starts. Thus, there should be a delay between the start of the primary and the secondary instances, allowing the primary instance to start up before the secondary instances create a cluster with the primary instance.

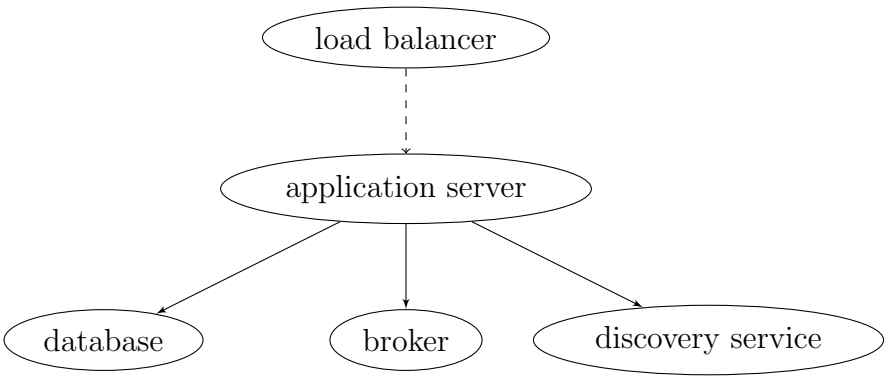


Figure 4.6: Dependency graph of the subsystems. The application server depends on the database, broker and discovery service. The load balancer (soft)-depends on the application server.

4.8.2 Shutdown procedure

We can derive the shutdown procedure by inverting the starting procedure. The administrators should stop the load balancer before they stop the application server as this ensures that the application server receives no new requests. As the application server depends on the database, broker, and discovery service, administrators should only stop them after they stopped *all* instances of the application server.

After they stopped the application server, they can safely stop the database, broker, and discovery service.

4.8.3 Update procedure

An administrator usually wants to update the system (which most of the time requires downtime and makes the system unavailable) when only a comparatively small amount of users use to the system. This causes them to often update the system either very early in the morning or very late in the evening, which, for most persons, is not optimal as these are not typical working hours.

By adding multiple instances of the application server, administrators can remove instances from the system without making the whole system

unavailable as other instances can still handle requests. This allows a *rolling-release*, where the system is always available.

This type of release is not applicable for all updates of the application server but can decrease the downtime for some updates.

Critical deployment

These deployments include updates that require all instances of the application server to use the same version. This might e.g., be due to updates of the database structure, where one instance updates the database on startup, which causes other instances to fail as the updated database structure does not match the one they expect.

Critical deployments still require a downtime of the system, as administrators must stop all instances and once no instance is running, they can start the updated version. There should never be multiple versions of the application server running, as this might cause issues when an unexpected database structure is present.

Non-critical deployment

Non-critical deployments (*rolling-releases*) allow the administrator to update the system without making the whole system unavailable. The administrators have to split to instances into two subsets, which they update successively. Once they stop the first subset of instances, the load balancer connects the users connected to these instances to the remaining instances. They can then update the first subset and start them with the updated version. Once these instances are running and start serving requests again, the system runs with two different versions simultaneously.

The administrators can now restart the second subset, causing all users to reconnect to the first subset. Once the second subset is updated, it gets added to the system again and can respond to requests.

The administrators were able to update the whole system without causing downtime for the users as a subset of the instances was always available. Only some deployments support this update method as they must not cause changes to the database, as pointed out above.

4.8.4 Failure handling

As one of the non-functional requirements is improving the reliability (availability), we have to point out which subsystems of the system are fault-tolerant and which are not.

Load Balancer

As we only have one load balancer deployed, a failure of it would cause all clients to lose connection to the server; thus, Artemis becomes unavailable. The load balancer is very reliable and broadly used. We hence do not expect it to fail. If the underlying machine fails, users cannot reach the whole system anymore.

Application server

If one instance of the application server fails, other instances can mitigate this. The load balancer can detect the failure and will no longer send requests to the unavailable instance. Submissions of users are not affected by the failure as they are either stored in the database or within the distributed cache. We configured a replica-count of 1 for the distributed cache to ensure that a failure of one instance can not cause data loss.

Database server

The database is a crucial part of the system as its failure will cause the unavailability of the whole system. A second server can mitigate this issue by taking over if the primary server fails. We did, however, currently not implement this.

Broker

A failure of the broker causes WebSocket connections to no longer work. All clients will show a 'Disconnected' message once they detect the outage due to a missing heartbeat[5]. This outage affects the functionality that relies on bi-directional messages, especially on messages sent from the server to the client. Automatic updates for new results, notifications, and the submission state of team exercises will no longer be available. The functionality that does not depend on WebSockets, e.g., loading courses and exercises, starting and participating in (individual) exercises is still possible. Once the broker gets available again, clients re-establish the connections and can use all functionality. A secondary broker can decrease the risk of an outage but is, at the moment, not fully supported.

[5] A message that is periodically sent between two systems so that they can confirm that the other system is available.

Discovery service

The main functionality of the discovery service is to provide information about currently running instances. As described in section 4.8.1, secondary instances depend on the discovery service to find the primary instance. If the discovery service fails during this startup, the distributed cache can not create the cluster correctly, thus causing an inconsistent state. A failure of the discovery service after the startup has no impact as the cluster already exists. It is also possible to work with multiple instances of the discovery service that replicate each other. Although Artemis supports this deployment, we currently have not added a second instance of the discovery service.

Chapter 5

Object Design

This chapter of the thesis focuses on selected design decisions that we made to implement the behavior described in chapter 4.

5.1 Caching

Artemis uses a cache to reduce the load on the database server, as we described in section 2.2.

Artemis uses Hibernate[1], a framework for Object/Relational Mapping (ORM), that automatically maps the objects used within Artemis to relations stored in the database [BK05]. Hibernate offers different caching strategies and integrates with several caching providers [JF11]. The cache consists of different regions (e.g., all cached objects of a class form a region) and developers can apply different configuration options or each region.

Artemis uses EhCache[2], a commonly used cache provider for Hibernate [Win13]. We first tried to manually manage the cache for the distributed system (e.g., by invalidating regions on all instances if one instance updated a region), but this led to a significant overhead during development as we had to write much code manually. Every time a developer applied a change to the database and thus to the cache, he had to manually clear the affected cache regions, making the code both hard to write and maintain. Although EhCache natively supports distributed caching, its usage is not well documented, especially in combination with Spring and Hibernate.

Hence, we decided to use Hazelcast[3] as cache provider as it supports

[1] `https://hibernate.org`
[2] `https://www.ehcache.org`
[3] `https://hazelcast.com`

distributed caching without manual management of the cache and provides proper documentation[4]. Hazelcast is an in-memory data grid that uses a peer-to-peer architectural style [Joh15]. Thus, We added the discovery service described in section 4.3 to allow Hazelcast to form a cluster containing all instances of the application server.

Hazelcast not only integrates into Spring and Hibernate but also provides common data structures like maps that all instances of the cluster can access. Artemis uses them for caching that is not directly integrated into Spring and Hibernate, e.g., they are used to store the submissions during live quizzes, as described in section 3.3.2. Therefore, all instances of the cluster access the same submissions, so that in case one instance fails, other instances still have a copy stored.

Figure 5.1 shows the shared usage of the database and the cluster built by Hazelcast to offer a distributed cache. Every instance of the application server is connected to every other instance to broadcast updates to the cache to ensure a consistent state. The cache acts as a proxy and receives all requests that interact with the database. It can either provide the requested data from the local cache (thus preventing a request to the database) or, if the data is not present locally, request the data from the database (and store the data in the local cache to prevent additional requests), implementing a proxy pattern.

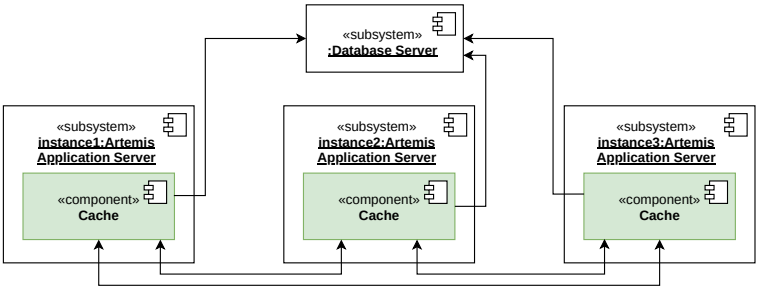


Figure 5.1: Shared usage of one database by three instances of the application server and communication within the distributed caching cluster. Every instance of the application server directly communicates with the database server and communicates with all other instances to handle cache invalidation.

[4] e.g., `https://reflectoring.io/spring-boot-hazelcast`, `https://hazelcast.com/blog/spring-boot` and `https://www.baeldung.com/java-hazelcast`

5.2 Delegating scheduled tasks

As we already shortly described in section 4.7, we use different scheduling algorithms for different use-cases, which we split into two categories: *primary instance* algorithms and *failover* algorithms, which we will describe in this section.

5.2.1 Primary instance

Artemis has to schedule a wide range of tasks. Some tasks handle clean up of no longer required artifacts, especially in combination with external systems like the Version Control Server (VCS) and the Continuous Integration Server (CIS). Artemis typically executes these tasks at fixed times (e.g., at 3 am every day or at 5 am every Friday), independent if there are artifacts present which Artemis has to remove. Artemis can schedule these tasks at the system's startup as the exact time of execution is known.

It schedules other tasks dynamically, especially tasks that individual exercises require. One example of this is the execution of unlocking and locking operations for student repositories. Students should not be able to perform submissions before the exercise/exam starts or after it ends. Instructors should create the repositories on the Version Control Server and build plans on the Continuous Integration Server before the exam starts to prevent issues during the exam if Artemis has to create all repositories at once.
Artemis, the VCS, and CIS need approximately 5 seconds when creating a repository and the corresponding build plan for a single student. If 1,000 students participate in an exam and the instructors create three programming exercises per student, the system has to create 3,000 repositories. Artemis can create approximately one repository per thread per five seconds; so, a machine with 12 threads can create roughly 12 exercises per five seconds, which corresponds to three exams. The creation of 1000 exams therefore takes $5s * \frac{1000}{3} \approx 1667s$, which is more than 25 minutes. As the system can only create the last repository after more than 25 minutes, the student owning this repository can only start working on it after a substantial amount of his/her working time passed. Instructors, therefore, pre-generate the repositories to ensure that all students can work on their submission once the exam starts.

Students should not be able to access the repositories and build plans before the exam starts. Artemis can prevent this by comparing the current time with the start date of the exam before displaying the files in the code editor. However, it cannot prevent students from directly accessing their

repository on the VCS. The system has to lock the repositories to ensure that students cannot access their pre-generated exam repositories until Artemis unlocks them a few minutes before the exam starts.

Artemis has to schedule this unlock task for each exercise independently and the time of the unlock operation can also change while the system is running. It should also lock the repository once the exam is over to prevent students from changing code after their working time is over.

We decided that only one instance of the applications server (the primary instance) is responsible for scheduling these tasks. Other instances (secondary instances) will not execute scheduled tasks. Administrators make this distinction using a Spring profile[5] called `scheduling`, which is only active on the instance that should handle the scheduling (for which the administrators of the system have to take care of).

Delegation of scheduled tasks

Artemis has to support updates for scheduled tasks as the execution time of tasks is not always known at the startup of a system but can change during the runtime. Instructors introduce these changes by creating, updating, or deleting exercises/exams. As Artemis does not know to which instance the load balancer will connect an instructor (as we described in section 4.4), every instance needs to be able to create, update, and delete scheduled tasks. This raises an issue as only the primary instance can actually schedule tasks.

We solve this issue by implementing a proxy pattern: Secondary instances forward the scheduling of the tasks to the primary instance. Figure 5.2 shows this proxy pattern with an example for scheduling programming exercises: The `ProgrammingExerciseService`, which is present both at primary and secondary instances, uses a provided `Scheduler` to schedule the tasks of an updated/created/deleted programming exercise. It does not know whether the provided `Scheduler` is scheduling the tasks or delegating them to a different instance. A `RealScheduler` (present on the primary instance) can immediately schedule the tasks. A `ProxyScheduler` (present on the secondary instances) delegates the operation to the `RealScheduler`, which then performs the scheduling.

We use Hazelcast topics[6] for this delegation as they allow instances to subscribe to these topics and publish data for these topics. Hazelcast then forwards the published to the registered listeners, which can execute application logic depending on the received data. Different listeners (subscriptions

[5]Spring profiles allow parts of the code to only be executed if the specific profile is active. The profiles will be set at the startup of the system using configuration options.

[6]`https://hazelcast.org/use-cases/messaging`

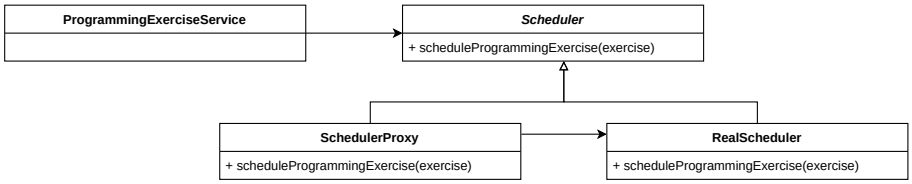


Figure 5.2: Example usage of the Scheduler with delegation. The `ProgrammingExerciseService` wants to schedule an operation and uses the `Scheduler` to do so. Either the `RealScheduler` immediately schedules the task or the `SchedulerProxy` delegates the task.

to different topics) can execute different logic.

The primary instance registers one listener for each topic (which represents one type[7] of possible update). Secondary instances will then publish the exercise's identifier to the corresponding topic, allowing the primary instance to receive the identifier, fetch the exercise from the database and execute the logic to create, update, or delete scheduled tasks based on the exercise and context.

Using this approach, we solve the issue that every instance should be able to schedule tasks, but only one instance should actually execute these scheduled tasks.

5.2.2 Failover

For live quizzes, Artemis uses a failover strategy. Hazelcast offers a scheduled executor service[8] that allows tasks to be registered and executed either on one, several, or all instances within a Hazelcast cluster. We configured the executor service to execute the task only on the instance that is the first member of the cluster and not on members that join later. The instance of the application server that receives the request to save a submission from a user stores the submission into a map. This map is shared using Hazelcast's distributed maps. Every instance can add entries to/update entries in this map. Once the live quiz is over, the instance that started first (usually the primary instance as described in section 5.2.1) will process the entries in the map (as the executor service selects it). This instance then evaluates the submissions and saves the submissions and results in the database.

[7]Possible types are e.g., the creation of a programming exercise or the deletion of a text exercise.

[8]`https://docs.hazelcast.org/docs/latest/javadoc/com/hazelcast/scheduledexecutor/IScheduledExecutorService.html`

5.3 Shared storage

A Network File System (NFS) implements the synchronization of the user content[9] that allows different machines to access shared storage [SGK+85]. Once one instance writes a file to the shared storage, the other instances can access it. Hence, one instance checking out a repository allows all instances to apply changes to the repository.

The system uses the same setup for the other file types it stores in the file system, such as submissions to file upload exercises. Only instances of the application server need to access the shared file system as only they access user content stored in file storage. The other subsystems (database server, broker, and discovery service) do not need to access this storage.

We also only store data on the shared file system that is relevant for all nodes, so we e.g., do not store configuration files, .war files (the executable that contains the application server), or log files in the shared storage. Every instance has an own version of it (which may also differ from each other e.g., in the case of configuration and log files).

5.4 WebSocket broker

We use Apache ActiveMQ Artemis[10] (later called *ActiveMQ*) as the WebSocket broker. ActiveMQ supports a wide range of messaging protocols, including the Advanced Message Queuing Protocol (AMQP), Message Queuing Telemetry Transport (MQTT), and STOMP (see 2.3). The Apache Software Foundation develops ActiveMQ in Java.

Spring already includes a simple broker used by default when processing WebSocket messages, but this broker does not support a deployment with multiple instances that share the same user base. It can be replaced by an external broker such as ActiveMQ[11].

We also tested other messaging systems, especially Apache Kafka[12], but as Kafka does not support STOMP, we decided to use ActiveMQ.

Redis[13], an in-memory data structure store that provides a message bro-

[9] Data that users of the system generate such as uploaded files or repositories. This does not include files that the systems needs to run like executables and configuration files.

[10] `https://activemq.apache.org/components/artemis`

[11] `https://docs.spring.io/spring/docs/5.2.7.RELEASE/spring-framework-reference/web.html#websocket-stomp-handle-broker-relay`

[12] `https://kafka.apache.org`

[13] `https://redis.io`

ker, is also not applicable for Artemis. Redis heavily uses a publish/subscribe model, which has similarities to STOMP but is not compatible. There are efforts to integrate STOMP support to Redis (e.g., using plugins), but as this is still experimental and not well documented, we decided not to use Redis.

5.5 WebSocket security checks

As we outlined in section 4.6, WebSocket subscriptions of a topic should only be possible if the user is allowed to access the topic. To implement these checks, we extended the existing WebSocket interceptor to check every subscription before executing it.

The application server receives a request and first identifies its type (e.g., if it is a request to subscribe to a topic or a message that should be handled by the server). If the request is of type `SUBSCRIBE`, the topic to which the client wants to subscribe to gets extracted. If there are security checks in place for this particular topic, then these checks get executed. The checks can be on a user- and group-level as some topics should only be accessible for a specific user (e.g., the owner of a participation). In contrast, other requests should be available for a group of users (e.g., all instructors of a course).

If the user is allowed to subscribe to the specific topic, the server adds the subscription. If the user is not allowed, the application server does not add the subscription but logs the incident.

5.6 WebSocket message grouping

Before the start of this thesis, clients create multiple WebSocket subscriptions for similar requests. This means that the client of one student will, e.g., create one subscription per exercise where he expects a result (meaning an exercise that is over and where he does not have a result yet). Although this is not problematic for courses with a small number of students and only a few exercises, in courses with several hundred or thousand students and e.g., ten unrated exercises per student, this causes many subscriptions the application server must handle.

For instructors, this behavior is even worse as there are pages displaying multiple student who participated in a particular exercise. This view updates itself automatically as soon as a new result is available for any of the shown students. The instructor's client has to create one subscription per student (as it can only subscribe to participations, which are bound to a student). If the client displays 50 students with their corresponding results, it creates

50 subscriptions which the application server all has to handle. Figure 5.3 shows the score page for a quiz exercise. The client creates 50 subscriptions as it shows 50 results and expects updates for every result.

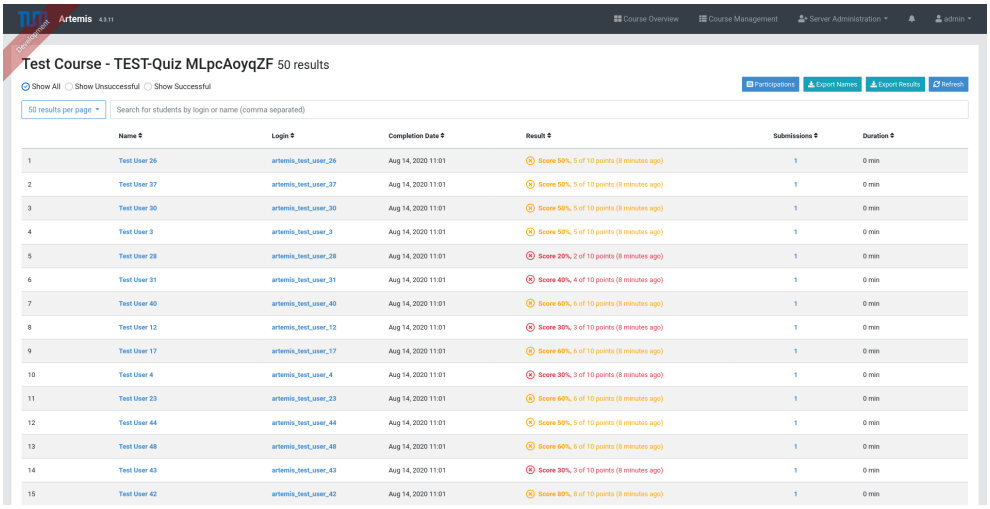


Figure 5.3: Score page for a quiz exercise. Clients of instructors create one subscription per participation (thus per student).

We optimize this behavior by grouping the messages into two topics so that a client only has to subscribe once: All students receive their updates of results through a `personal` topic. One `personal` topic now receives results from multiple exercises (that might even be in different courses). The client then extracts the participation to which the results belongs and triggers the corresponding logic.
We cannot use this `personal` topic for instructors as they should be able to receive updates for other users (their students). To solve this issue, the client of instructors now subscribe to new results on an `exercise` level to receive all new results belonging to this exercise. The client then again extracts the participation from the received data and triggers the correct logic.

Figure 5.4 shows the relationship between the grouped messages: Before applying the changes, there was one subscription per participation. Artemis now groups them either per user (for students) or per exercise (for instructors), leading to a reduced amount of subscriptions.

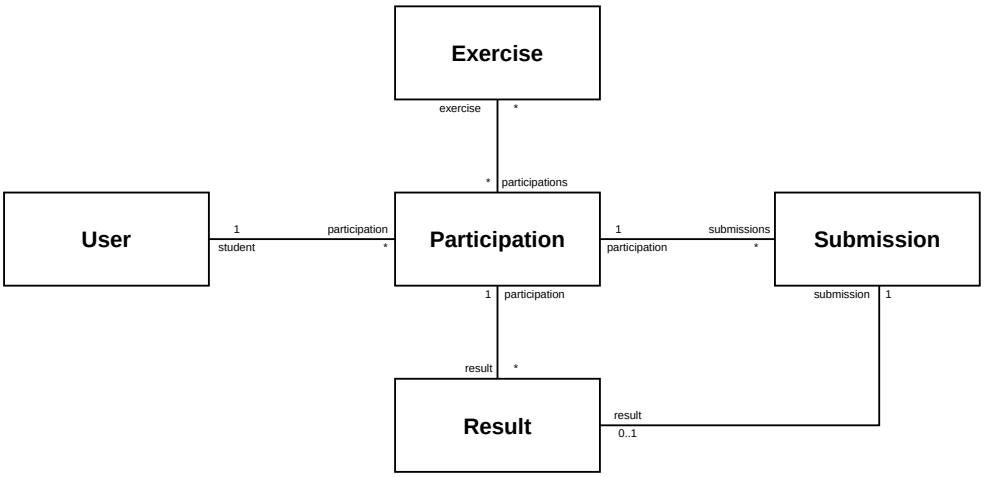


Figure 5.4: Relationship of exercises, participations, submissions, results and users. Instructors create exercises in which students participate. Students submit multiple submissions which are associated to their participation and receive results for their submissions.

We also applied this update from many subscriptions to few with regard to submissions.

5.7 Discovery

As multiple instances of the application server should find each other to create the distributed caching cluster without further configuration, a discovery service is required.

The software we use to run the discovery service is JHipster Registry[14], a JHipster application that includes Spring Cloud Netflix Eureka and Spring Cloud Config. Spring Cloud Config allows the management of configuration files for a large number of instances, but Artemis does currently not use it. Spring Cloud Netflix Eureka allows applications to register themselves and fetch other registered instances as described above. Netflix develops Eureka[15] and uses it for the management of servers within AWS[16] to allow for load balancing and fail-over [LSB18].

Eureka can also deal with a more complex deployment, including several regions where services are located (e.g., *us-east*, *us-west*, and *europe-central*),

[14] `https://github.com/jhipster/jhipster-registry`
[15] `https://github.com/Netflix/eureka`
[16] Amazon Web Services, a cloud-computing platform from Amazon.

but we do not need these features as we deploy Artemis at one region.

Figure 5.5 shows the instance overview with 11 running instances. The identifier of the instances can be seen as well as the status of every instance, including version, port and running Spring profiles.

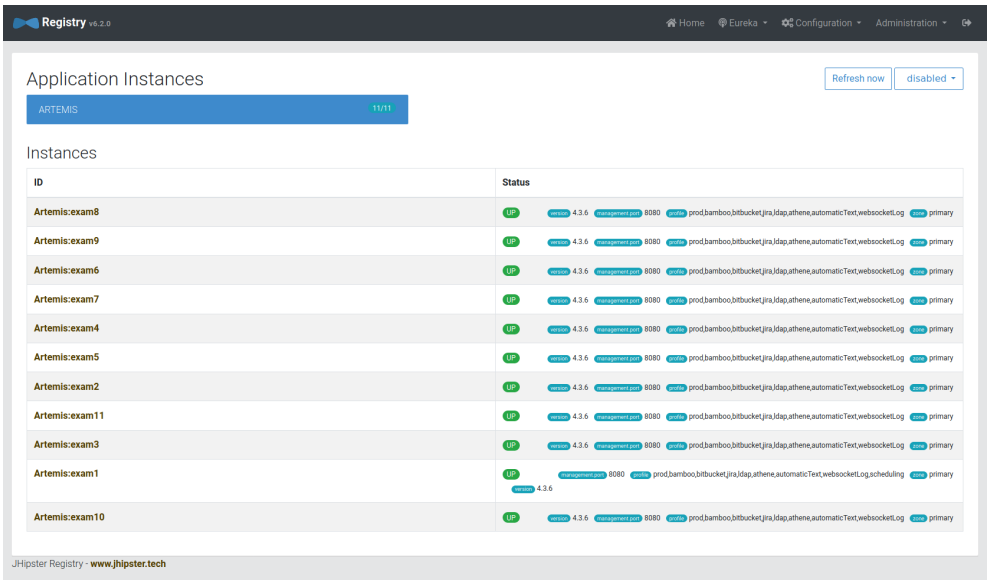


Figure 5.5: Instance overview of the discovery service with 11 instances of the application server. The discovery service also provides administrators with additional meta-data like versions used by the different instances.

Administrators can also use the discovery service to inspect each registered instance's logs and metrics and the discovery service itself. Figure 5.6 shows the metrics of one instance (currently instance 1). JHipster Registry displays the memory and CPU usage, the current threads within the java application and the HTTP requests, and their frequency and average response time. Administrators can navigate between different instances of the application server using a dropdown on the top right.

5.8 Monitoring

Monitoring allows administrators of the system to get an overview of the system's status and makes deviations from the expected state visible. The monitoring system collects different types of metrics that are either provided by an application that runs on the (virtual) machines to receive information

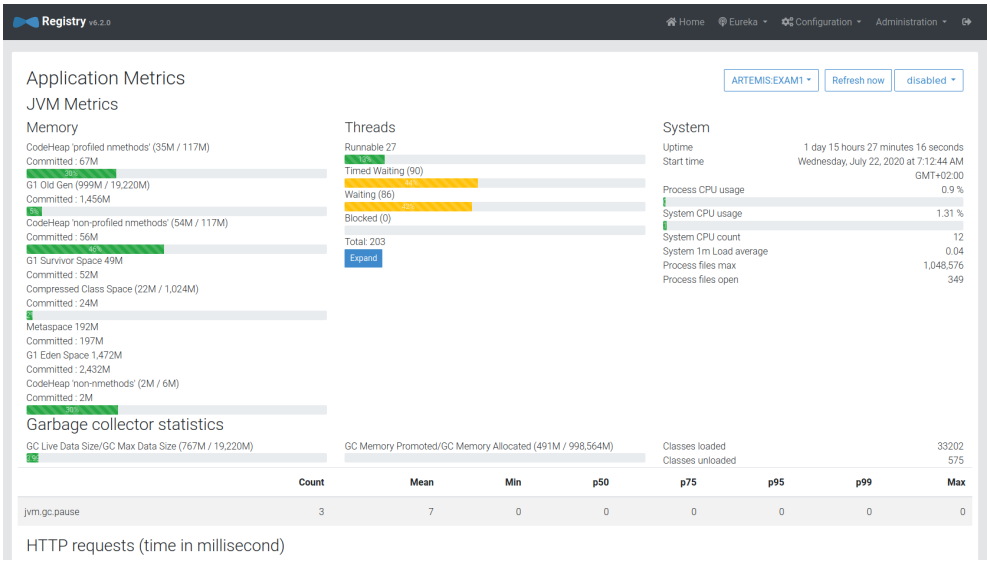


Figure 5.6: Metrics overview within the discovery service. Administrators use these metrics to validate the system's status.

about the system's used resources or provided by the application that it should monitor directly (here the application server).

We use Prometheus[17], an open-source, metrics-based monitoring system, to collect the metrics provided from the different systems [Bra18]. Prometheus collects data and Grafana[18], a tool that supports data from a wide range of data sources like InfluxDB, MySQL, PostgreSQL, and Prometheus, visualizes it. Prometheus sends alerts to administrators to inform them about problems with the monitored system.

5.8.1 Machine monitoring

We use the tool *node-exporter*[19] to export metrics from the machines that run Artemis. These metrics include e.g., CPU-, file system- & memory-usage, the number of open processes, and network statistics.

We need these metrics, especially for parts of the system that do not export more detailed usage like the database server, the load balancer, and the broker. These metrics are also relevant for subsystems that expose custom metrics (like the application server), as the application server cannot export metrics if it is unavailable. The metrics exposed by the application

[17] `https://prometheus.io`
[18] `https://grafana.com`
[19] `https://github.com/prometheus/node_exporter`

server might not be as detailed as they do not measure the hardware metrics accurately.

Prometheus periodically collects the metrics *node-exporter* provides within a HTTP-endpoint.

Figure 5.7 shows a subset of the hardware metrics Prometheus collects for every virtual machine. The figure shows the hostname, number of CPU cores, CPU usage, and memory usage for the virtual machines that run the broker, database, and load balancer. Other metrics like the disk space usage, disk throughput, context switches, and network traffic are also available.

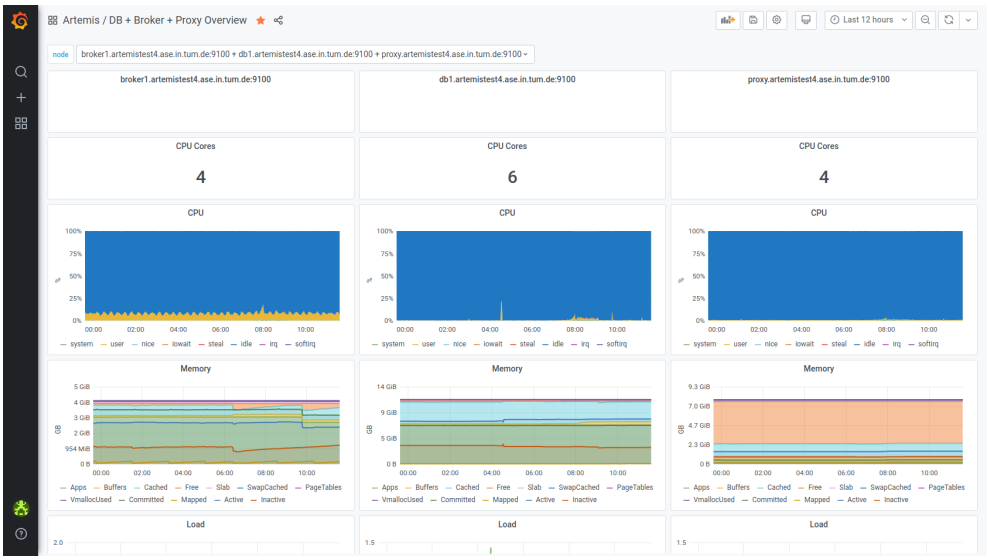


Figure 5.7: Hardware metrics of the virtual machines running the broker, database and load balancer. Administrators monitor the CPU and memory usage.

5.8.2 Application monitoring

Artemis supports some metrics by default as the *spring-metrics* dependency[20], which Artemis uses, includes them. These metrics, however, only handle general metrics that apply for all/most Spring applications like HTTP request duration and CPU/memory usage.

We extended the existing metrics to include custom metrics like the number of users connected to a particular instance of the application server and the health of the external systems Artemis uses.

[20] `https://docs.spring.io/spring-metrics/docs/current/public/prometheus`

Figure 5.8 shows custom metrics (the number of users connected to an instance of the application server) that Prometheus collects. Artemis exposes the application metrics via an HTTP endpoint and Prometheus pulls them periodically. We configured Artemis only to allow access to this endpoint from a predefined IP-address so that only Prometheus can access these metrics.

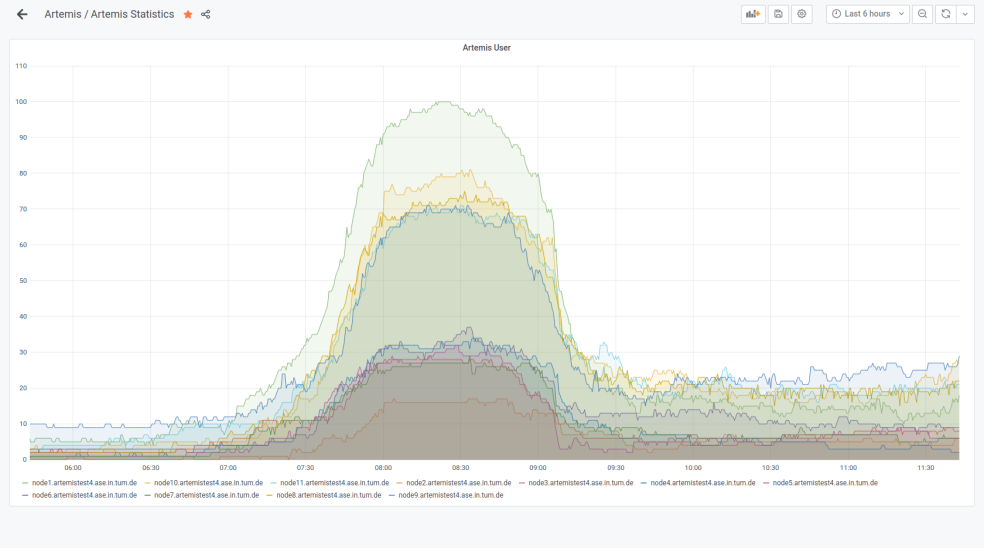


Figure 5.8: Application metrics of the application server. The user count on all instances increases while students participate in an exam on Artemis. The user count decreases once the exam is over.

5.9 IP hashing in load balancer

We want to distribute the load for the application server instances equally (with respect to their corresponding weight). As we also want one user to be connected to the same instance if possible (we can, however, not guarantee it), we configured a hashing method within the load balancer to force "sticky sessions".

nginx[21], the load balancer, supports several methods to select the responsible instance for an incoming request, including round-robin and least connections[22].

[21] `https://nginx.org`

[22] `https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/#method`

These methods do not support sticky sessions. Thus we decided to use the `ip_hash` method, where the first three octets of an IPv4-address/the whole IPv6-address get hashed. nginx will then connect addresses with the same hash to the same instance. This ensures that multiple requests from one user will be sent to the same server as long as his/her IP-address stays the same.

We later decided to use the more generic `hash` method, which supports a wide range of attributes of a request, like the request URI or the used remote IP-address. When using `hash $remote_addr` as a load-balancing method, the behavior is the same as with the `ip_hash` method, but all four octets of an IPv4-address get hashed, leading to a more equally distributed load.

Chapter 6

Case Study

Several universities and institutions use Artemis, but as the Chair for Applied Software Engineering (ASE) at TUM develops Artemis, it has the most advanced setup and, as far as we know, the most users.

In this chapter, we will show how we moved Artemis to a deployment with multiple instances, especially for the exam mode that students used for examinations during the summer term 2020.

6.1 Artemis application server

We configured different weights for the Artemis application server instances as the underlying hardware varies, as we suggested in section 4.4. More powerful instances have often three times as much CPU cores as smaller instances, thus we assign the larger instances with a more significant weight.

Table 6.1 shows the assignment of weights for the deployment that we used during exams. We assigned one unit of weight for every four virtual CPUs the underlying virtual machine has as Artemis is more intense in its CPU usage in comparison to its memory usage. Table 6.1 also shows who hosts the instances: The instance with id `node1` is hosted by the RBG[1] and provided the only instance before we performed the horizontal scaling. The RBG also hosts `node2` - `node7` as these virtual machines were available at short notice once the Artemis project leaders decided to implement the exam mode.

The Chair for Applied Software Engineering[2] (ASE) at TUM hosts `node8` - `node11` on dedicated machines it bought for the Artemis exam mode. We

[1] Rechnerbetriebsgruppe, the service group that maintains the infrastructure at the faculties for Computer Science and Mathematics at TUM

[2] `https://ase.in.tum.de`

could only add these nodes later as these dedicated machines have higher provisioning times compared to virtual machines on existing hardware.

By splitting the hosting of the instances, we decrease the dependence on other systems. Systems that use virtual machines on the same hardware in the data center of the RBG can impact the performance of the instances of the application server hosted by the RBG. We can mitigate this performance loss by manually increasing the weight of the instances hosted by ASE once we detect such service degradation.

Instance Id	#CPU cores	Available RAM in GB	Weight	Hosted by
node1	12	24	3	RBG
node2	4	8	1	RBG
node3	4	8	1	RBG
node4	4	8	1	RBG
node5	4	8	1	RBG
node6	4	8	1	RBG
node7	4	8	1	RBG
node8	12	64	3	ASE
node9	12	64	3	ASE
node10	12	64	3	ASE
node11	12	64	3	ASE

Table 6.1: Mapping of weights to the different instances of the Artemis application server depending on the available resources. Instances with a higher number of CPU cores receive a larger weight.

6.2 Performance evaluation

We will focus on the graded online exercise of the lecture *Introduction to Software Engineering* in the summer term 2020 that took place on July 27, 2020.

1,288 students participated in the exam that started at 08.00 am and ended at 09.40 am for most students. Some students had an extended working time due to disadvantage compensations, but we will only focus on the requests that Artemis processed from 08.00 am to 09.40 am as most students could not submit after 09.40 am.

Setup

We used the setup described above with 11 instances of the application server and one instance of the load balancer, database server, broker, and discovery service.

Processing times

The students submitted 43,658 answers for text exercises, 45,159 answers for modeling exercises, and 21,120 answers for quiz exercises during the 100 minutes that we will focus on for this analysis. The client automatically saves every 30 seconds if there are unsaved changes and saves the current exercise when navigating to a different exercise.

Table 6.2 shows the average processing time as well as the 0.5-, 0.75, 0.9-, 0.95- and 0.99-quantiles.

Type	Average	50%	75%	90%	95%	99%
Text	48ms	43ms	47ms	53ms	62ms	217ms
Modeling	51ms	44ms	48ms	55ms	67ms	279ms
Quiz	60ms	53ms	59ms	67ms	80ms	380ms

Table 6.2: Average and 0.5-, 0.75, 0.9-, 0.95- and 0.99-quantiles of the processing duration of the text, modeling, and quiz exercises during the EIST graded online exercise.

Figures A.1, A.2, and A.3 show the processing duration of the text-, modeling, and quiz-exercises. All figures show that requests took longer at the start of the exam, but after circa 15 minutes, most of the requests took less than 100ms.

Artemis could, therefore, fulfill the requirements concerning performance.

6.3 Failure handling

During the development of the exam mode, developers introduced some bugs within the application server as they could not test all features in all scenarios due to time limitations. This caused errors where database queries that load large amounts of connected objects, cause an unexpected high memory usage. The memory usage caused Artemis to either become unresponsive for several minutes or crash. The score page in the exam overview contained such a problematic query. The memory usage went from less than 1GB (that Artemis usually uses even with several hundred users) to 8GB (the maximum that we assigned to the process).

On July 6, 2020, at 03.15 pm, this problematic query was executed for the exam of the course IN0003[3], which took place on July 4, 2020. This caused one instance of Artemis to become unavailable, sot that all requests

[3]Introduction to Informatics 2

coming to Artemis had to be answered by a different instance. We only had two instances in the system at the time of the incident, so that all requests had to be answered by the second, remaining instance.

As the course IN0001[4] held an exam on July 6, 2020 at 04.15 pm, only 1 hour after the incident, Artemis needed to be available for this exam. As far as we know, the failure handling went as expected so that the load balancer reconnected all users of the failing instance to the remaining instance. Users reported no problems and once we restarted Artemis, the users began connecting to both instances again. The horizontal scaling combined with the failure handling saved us from significant problems as we could ensure that the system was available.

The same problem also occurred due to a faulty query when evaluating quiz exercises for exams, which also caused Artemis to become unavailable due to the high memory usage when executed for an exam with a large number of students. The failure handling here also went as expected. The load balancer reconnected all users connected to the faulty instance to one of the ten other instances that we deployed at the time the incident occurred.

6.4 File System

We used a Network File System (NFS) provided by the IT-group of the ASE chair. Despite some concerns at the beginning that we had due to a possible bottleneck of the file system, the NFS fulfilled the performance requirements without problems.

We could locate some issues in the logic of the application server, which caused that some lock-files[5] were created and never removed, but this was not a problem of the NFS, but of the existing application server code. The application server caused the issue when checking out a repository (e.g., so that a student can use it in the online code editor), which caused the lock-files to not always getting removed. This caused an issue as soon as a different instance wanted to perform some requests on the same repository. We fixed the issue by adding unlock-operations in the code of the application server.

[4]Introduction to Informatics 1

[5]Temporary files that a process creates when it interacts with a file to indicate that other processes should not use the file. The process removes it once it no longer interacts with this file.

Chapter 7

Summary

This chapter summarizes the results of this thesis. We first show the status of the thesis by describing realized & open goals. We then conclude the outcome of the thesis and show aspects of future work.

7.1 Status

We were able to implement to main goals for this thesis, as stated in the requirements analysis chapter. We could not achieve some goals due to lack of time.

The requirements can be grouped into three categories:

- ● We could fully implement these requirements.
- ◐ We could partially implement these requirements.
- ○ We could not implement these requirements.

Table 7.1 shows the status of the implementation of the non-functional requirements.

	Non-Functional Requirement	Status
NFR1	Performance improvements	●
NFR2	Fault-tolerance of Artemis	◐
NFR3	Adjust security checks for real-time communication	◐
NFR4	Preserving simple deployment	●

Table 7.1: Status of the implementation of non-functional requirements. We could fully implement NFR1 & NFR4 and partially implement NFR2 & NFR3.

7.1.1 Realized Goals

As described in chapter 3, we did not implement any functional requirements within this thesis.

We realized NFR1 (Scalability), as shown in the case study for the exams/graded online exercises. We were not able to test Artemis with 10000 users. Problems can arise concerning the WebSocket connections, as the broker requires a substantial amount of resources for a large number of users.

We fulfilled NFR2 (Availability) for the application server instances as the outage of one, or several, of them will not cause a system failure. However, we could only partially implement it overall as outages of other parts of the system can not be tolerated.

We implemented NFR3 (Security) for the most critical requests (that could leak personal data such as results). Nevertheless, we did not implement it for all topics that clients can subscribe to using WebSockets.

We realized NFR4 (Maintainability) as the distributed setup of the system implemented for NFR1 & NFR2 is optional, as described in section 4.4.

7.1.2 Open Goals

Although we were able to implement the failure-tolerance for the application server (which is the most common cause of failures), we could not make every outage of a subsystem automatically resolvable, as stated in section 4.8.4. We also did not yet implement the security checks for all WebSocket subscriptions.

A test of the scalability with 10,000 users still has to be performed to ensure the system's availability, even for very large courses. However, as we currently do not have more than 2000 users recorded on record, we did not yet perform this test.

We also did not use a unified scheduling approach for all components of the system, as described in section 5.2, but used two different approaches alongside each other. A unification of these approaches is missing but should be aimed for as it decreases the complexity during the system's development.

7.2 Conclusion

We solved the scaling of a web application, Artemis, on three levels:

- REST: We modified the cache provider and updated the database setup by moving the database to its own machine

- WebSocket: We added the broker so that actions performed on one instance of the application server can reflect to users connected to a different instance
- File System: We added a file system that allows all instances of the application servers to access the same stored data

We also improved the monitoring tools and ensured a deployment that administrators can manage without extensive intervention by adding the discovery service.

We modified the existing WebSocket subscriptions to be less resource-intensive (by grouping them) and more secure (by adding security checks for relevant topics).

7.3 Future Work

Future work that extends this thesis is improving the availability by making every subsystem redundant. We already prepared this for the discovery service (so that several instances always update each other) and also started to add this for the broker (again, so that are replicas that can take over if one instance fails). However, we did not fully implement this yet due to time limitations.

The scaling of the database is more complicated as design decisions have to be made concerning the setup that the database should use: One possible deployment is a primary/secondary setup using one server (primary) responsible for all write-operations, whereas every server (primary and secondary) can handle read-operations. The secondary than could take over the role of the primary in case the primary fails.

Another point that one should look into is automatic failure recovery, e.g., the restart of a subsystem if it fails without manual interactions. The main challenge in this topic is to ensure the system's consistency as a wrong sequence of start operations might lead to a violation of the starting procedure (as described in section 4.8.1), which can cause an inconsistent system. One possible inconsistency would be that the system can not build its distributed caching cluster. Instead of one cluster containing all instances of the application server, there might be several clusters that only contain subsets of the instances.

Appendix A

Performance evaluation

The following figures belong to section 6.2 and focus on the processing time of submissions during the EIST graded online exercise.

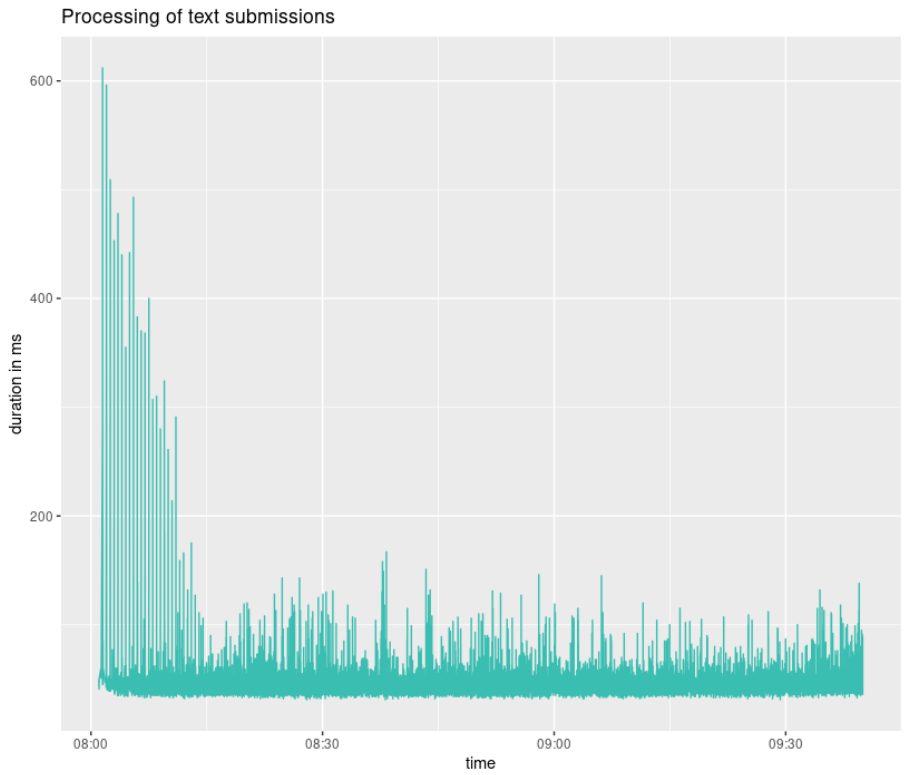


Figure A.1: Processing time of text submissions during the EIST graded online exercise in milliseconds over time. The average processing time decreases as the exam progresses and is less than 100ms after the exam has ended.

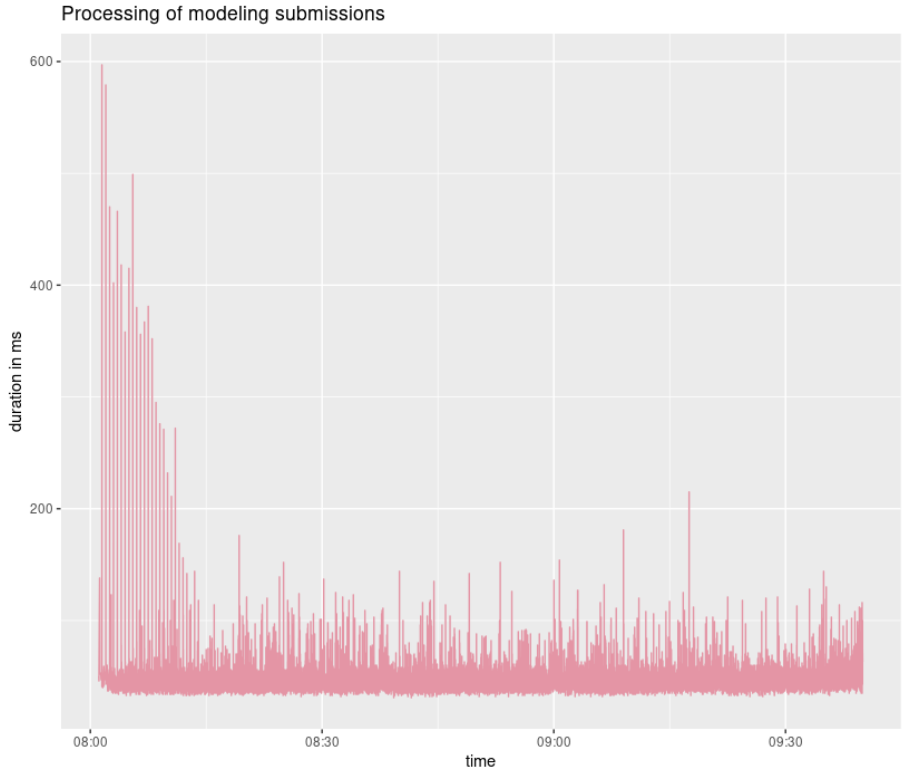


Figure A.2: Processing time of modeling submissions during the EIST graded online exercise in milliseconds over time. The average processing time decreases as the exam progresses and is less than 100ms after the exam has ended.

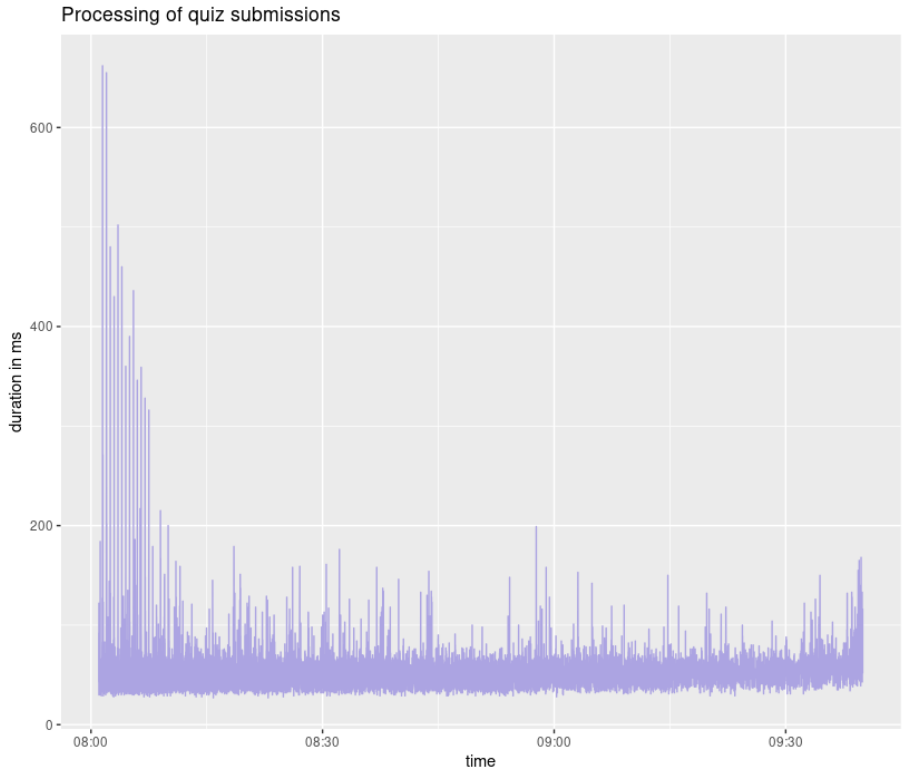


Figure A.3: Processing time of quiz submissions during the EIST graded online exercise in milliseconds over time. The average processing time decreases as the exam progresses and is less than 100ms after the exam has ended.

List of Figures

List of Tables

Bibliography

[BD09] Bernd Bruegge and Allen H Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 2009.

[BK05] Christian Bauer and Gavin King. *Hibernate in action*, volume 1. Manning Greenwich CT, 2005.

[Bra18] Brian Brazil. *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring*. O'Reilly Media, Inc., 2018.

[CY19] Franciso Javier De las Casas Young. *Extension of Quiz Exercises in ArTEMiS*, Technical University of Munich (TUM), 2019.

[DGV+12] Sourav Dutta et al. Smartscale: automatic application scaling in enterprise clouds. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 221–228. IEEE, 2012.

[DGV+14] Gargi Banerjee Dasgupta et al. Dynamically scaling multi-tier applications vertically and horizontally in a cloud environment, 2014. US Patent 8,756,610.

[Don17] Jason A Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *NDSS*, 2017.

[Erb12] Benjamin Erb. Concurrent programming for scalable web architectures, 2012. DOI: 10.18725/OPARU-2423.

[Erk12] Jussi-Pekka Erkkilä. Websocket security analysis. *Aalto University School of Science*:2–3, 2012.

[FM11] Ian Fette and Alexey Melnikov. The WebSocket Protocol, 2011.

[FT00] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[JF11] Wang Jing and Rui Fan. The research of Hibernate cache technique and application of EhCache component. In *2011 IEEE 3rd International Conference on Communication Software and Networks*, pages 160–162. IEEE, 2011.

[Joh15] Mat Johns. *Getting Started with Hazelcast*. Packt Publishing Ltd, 2015.

[KAEC+18] Jakub Krzywda et al. Power-performance tradeoffs in data center servers: DVFS, CPU pinning, horizontal, and vertical scaling. *Future Generation Computer Systems*, 81:114–128, 2018.

[KNO+02] Panos Kalnis et al. An Adaptive Peer-to-Peer Network for Distributed Caching of OLAP Results. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 25–36, 2002.

[KS18] Stephan Krusche and Andreas Seitz. ArTEMiS: An automatic assessment management system for interactive learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, pages 284–289, 2018.

[KSB+17] Stephan Krusche et al. Interactive learning: increasing student participation through shorter exercise cycles. In *Proceedings of the Nineteenth Australasian Computing Education Conference*, pages 17–26, 2017.

[Lom15] Andrew Lombardi. *WebSocket: lightweight client-server communications*. O'Reilly Media, Inc., 2015.

[LSB18] Andrew Leung, Andrew Spyker, and Tim Bozarth. Titus: introducing containers to the Netflix cloud. *Communications of the ACM*, 61(2):38–45, 2018.

[LSL+14] Chien-Yu Liu et al. Vertical/horizontal resource scaling mechanism for federated clouds. In *2014 International Conference on Information Science & Applications (ICISA)*, pages 1–4. IEEE, 2014.

[Mas11] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011.

[MHP12] Peter Membrey, David Hows, and Eelco Plugge. SSL load balancing. In *Practical Load Balancing*, pages 175–192. Springer, 2012.

[MNS+88] Steven P Miller et al. Kerberos authentication and authorization system. In *In Project Athena Technical Plan*. Citeseer, 1988.

[Mon17] Josias Montag. *Conducting Interactive Programming Exercises in Online Courses*. Master's thesis, Technical University of Munich (TUM), 2017.

[MS93] Jim Melton and Alan R Simon. *Understanding the new SQL: a complete guide*. Morgan Kaufmann, 1993.

[Sch18] Valentin Schlattinger. *Extending ArTEMiS: Interactive Live Quizzes in the Classroom*. Master's thesis, Technical University of Munich (TUM), 2018.

[SGK+85] Russel Sandberg et al. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985.

[SHS14] Dejan Skvorc, Matija Horvat, and Sinisa Srbljic. Performance evaluation of websocket protocol for implementation of full-duplex web streams. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1003–1008. IEEE, 2014.

[Sto] STOMP - The Simple Text Oriented Messaging Protocol. URL: `https://stomp.github.io/`.

[VST17] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[Wau20] Martin Wauligmann. *Team-based Exercises in Artemis*. Master's thesis, Technical University of Munich (TUM), 2020.

[Win13] Daniel Wind. *Instant Effective Caching with Ehcache*. Packt Publishing Ltd, 2013.

[WSM13] Vanessa Wang, Frank Salim, and Peter Moskovits. Using Messaging over WebSocket with STOMP. In *The Definitive Guide to HTML5 WebSocket*, pages 85–108. Springer, 2013.

[YHM00] David J Yates, Abdelsalam A Heddaya, and Sulaiman A Mirdad. Method and system for distributed caching, prefetching and replication, 2000. US Patent 6,167,438.