Software Construction

RWTH AACHEN UNIVERSITY

The present work was submitted to the Research Group Software Construction

of the Faculty of Mathematics, Computer Science, and Natural Sciences

Master Thesis

A simulation approach to assess microservice applications

presented by

Jonathan Ilk

Aachen, April 6, 2023

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Alex Sabau, M. Sc.

Acknowledgment

First of all, I would like to express my sincere gratitude to my supervisor Alex Sabau for the extensive supervision during my master thesis. I am aware that this cannot be taken for granted.

I would also like to thank Prof. Dr. rer. nat. Horst Lichter for the valuable feedback, particularly following my intermediate talk.

Finally, I want to thank my family for all their support throughout my studies. In particular, I would like to thank my brother for proofreading large parts of this thesis and especially for sparking my interest in software development in the first place.

Jonathan Ilk

Abstract

The microservices architectural style is an increasingly popular way of designing large, complex applications. Recent literature has proposed an automatic aid for verifying compliance of microservice architecture models with best practices, as a way to ensure good quality microservice architectures. However, compliance with best practices does not guarantee that the resulting microservice application will meet its performance requirements. In the literature, there is a lack of approaches to predict performance of microservice applications based on their architecture models.

In this thesis, we propose a discrete-event simulation approach to predict the performance of microservice applications. To do this, our approach uses user-generated architecture models, workload scenario descriptions, and deployment configuration to simulate the behavior of microservices and their interactions. The results can be used to gain detailed insight into the expected performance and its causes, allowing architectures to be adjusted cost-effectively before writing a single line of code.

We validate our approach by first simulating two microservice applications, one using broker-based messaging and the other using brokerless messaging. Subsequently, we implement these applications and perform real-world measurements. From a comparison of the results, we conclude that while our approach is promising in principle, further elaboration is needed, especially for the simulation of the hardware on which the microservices run.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Contents

Nowadays, software is pervasive in almost every aspect of our lives. Due to the increasingly complex tasks for software, the complexity of software itself grows further[Mit90]. Simultaneously, software quality is expected to meet a multitude of requirements, which are typically expressed by measuring quality attributes[Tek+16; BCK03]. Fast response times are critical as users are quick to abandon slow-performing services[Nah04]. However, this must be balanced against cost considerations, requiring a focus on minimizing resource utilization. At the same time, systems are often exposed to excessive workloads for short periods of time, during which the response times must be kept stable[Feh+14]. In addition, systems need to be easily adaptable to meet ever-changing requirements[NZP04]. Therefore, it is necessary to structure software systems carefully[BCK03].

Structuring a system through software elements, their associations with each other, and the attributes of both is the domain of software architecture[BCK03]. The most appropriate software architecture is determined by the requirements and requires balancing of various quality aspects[BCK03]. Since many software systems often face similar challenges, similar software architectures can be used to overcome them[BCK03]. Architectural patterns consist of proven design decisions and can help to find a suitable software architecture[BCK03]. A popular architecture pattern to handle the ever-increasing complexity of software is the microservices architectural style[Dra+17]. It structures the system as a set of self-contained services that are deployed independently and communicate solely through messages via service interfaces[BCK03]. Microservices are great for deploying changes quickly, are easily maintainable, and can scale resource-efficiently for unpredictable workloads[BCK03; Dra+17]. However, microservices also introduce a new dimension of complexity. The decomposition of the application logic into different services, the design of communication between the services, and the implementation of processes across service boundaries pose new challenges[Ric18; BCK03; BNK18]. Whether microservices bring the expected benefits depends largely on whether the decomposition into services is performed correctly[Ric18].

However, optimal decomposition cannot be guaranteed easily, and achieving it is a challenging task[Ric18]. To aid architects in creating a good decomposition, a considerable amount of literature provides best practices and architectural design patterns for microservice architectures[Ric18; HSS+17; Pac18]. Nonetheless, industrial-grade microservice architectures can consist of hundreds of microservices, making manual verification of compliance with best practices a time-consuming task[Nte+21]. Recent work by Ntentos et al. [Nte+21] has focused on developing a semi-automated approach to measure quality of microservice architectures, by validating their conformance to microservice architectural design patterns. For their approach, the architecture must first be modeled, then their developed algorithms can recognize architectural design pattern violations and repair them automatically[Nte+21]. However, the correct use of design patterns does not automatically lead to low response times for the offered services, as patterns exhibit different performance for different scenarios[AP19]. But good performance can be critical to the success of a software product, as high response times can lead to dissatisfied customers[SN18]. Thus, only verifying compliance with design patterns is not enough to ensure that a microservice architecture is well suited for the requirements of the resulting application. To assist architects in constructing suitable architectures that lead to satisfied customers, there is a need for a method to evaluate the performance implications of different microservice architectures.

One way to evaluate the performance of a microservice architecture is to implement and deploy the corresponding microservice application to then run performance tests. Although viable, this approach is not ideal, as identifying architectural errors during architecture creation is preferable to avoid high costs of fixing them[Tol+19]. To cost-effectively identify architectural flaws that lead to poor performance in specific scenarios, a predictive approach that works on the architecture models directly is needed. Combined with verification for compliance with architectural design patterns, such an approach could provide great guidance for creating good microservice architectures.

In the literature, there are approaches for performance prediction of microservices. However, most require benchmarks of the existing system and focus on the prediction of changes in performance[Bao+19; ZGD19; COQ21; Kha+21; MWW12]. Others work with models too specific, to assess them regarding architectural best practices[GIM+17], or are based on too general models that are not adequate for the simulation of microservices[KBB19]. Therefore, the current approaches are not appropriate for use alongside the verification of design pattern compliance in aiding software architects to develop an initial microservices architecture.

1.1. Research Questions

We consider this thesis as a contribution to a broader effort with the goal of developing a comprehensive exploration platform that provides tools for building good microservice architectures. To support such an exploration platform, we aim to investigate an approach that can predict the performance of microservice applications. By performance, we understand the request-response times that users experience when using a microservice application. The performance is composed of the communication time between the user and the application, the communication times between microservices within an application, and the processing times of the microservices to provide a response to the request. For this purpose, we implement a simulation for performance prediction based on the microservice architecture models of Ntentos et al. [Nte+21]. In combination with an automatic check of architectural design pattern conformance, such a performance prediction could be of great help when creating microservice architectures. By answering the following research questions, this work aims to provide the basis for the further development of a model-based simulation approach to predict performance.

> ***RQ 1*** *Can model-based architecture simulation be a meaningful approach to predict performance of microservice applications?*

Performance prediction of software using model-based simulation is an already established approach that has been extensively covered in literature[HM98; Reu+16; Bec+06]. However, modeling and simulating microservice applications for performance prediction poses new challenges[Hei+17]. In particular, we want to examine how architectural decisions affect the performance of a microservice application and whether simulation is a meaningful approach to predicting these effects. A meaningful performance prediction approach must provide accurate results that enable the evaluation of performance in the context of other quality attributes. Because inaccurate simulation results may lead to the rejection of good architectures based on predicted performance, even when this is not warranted. In addition, the prediction must be detailed enough to be able to make a statement about the causes of performance differences, so that an adjustment of the architecture is possible in the right places.

The extent of this work is not sufficient to examine all aspects of a microservice architecture that might affect performance. Therefore, the focus of this work is to determine if model-based simulation can accurately predict performance differences between messaging via a broker and direct messaging. This should give us an indication of whether model-based simulation is appropriate for predicting the performance of microservice architectures in general.

> ***RQ 2*** *What adaptations to the metamodel of Ntentos et al. [Nte+21] are necessary, to enable it to support performance prediction simulation?*

The metamodel for microservice architectures by Ntentos et al. [Nte+21] does not include all the information needed for performance prediction in specific scenarios. For example, the model does not include information about the hardware on which the different services run[Nte+21]. However, the hardware has a large impact on the expected performance, which is why we need to extend the metamodel[Bao+19].

1.2. Structure of this Thesis

The thesis is organized as follows. To provide a foundation for understanding our simulation, in chapter 2 we first give an overview of relevant microservice application concepts and explain the basics of discrete-event simulation. In chapter 3, we briefly present the work of Ntentos et al. on whose metamodel our metamodel for the simulation is based.

Furthermore, we give an overview of the existing approaches for the simulation of microservice applications and explain their shortcomings. In chapter 4, we then present the concepts for our simulation approach for performance prediction of a microservice application. Due to their scope, we explain the concepts for predicting message transfer times in a separate chapter (chapter 5). In chapter 6 we explain the design of the simulation implementation and in chapter 7 we present the details of our simulation implementation. To evaluate the accuracy of our simulation implementation, we compared simulation results with measurements of the performance of real-world microservice applications. We present and discuss the results of the evaluation in chapter 8. In chapter 9, we provide answers to our research questions. Finally, in chapter 10, we summarize this thesis and provide an outlook for future work.

2. Foundations

Contents

In this chapter, we first explain the fundamentals of microservice applications, which are necessary for understanding the concepts of our simulation approach. We then explain the basic concepts of a discrete-event simulation (DES) to provide an understanding of our implementation of a DES.

2.1. Microservice Applications

A microservice application is a distributed application that consists of several small applications (microservices) that implement little functionality, run independently, and communicate with each other via messages[Dra+17]. Utilizing microservices to implement an application's functionality provides advantages over implementing it as a single application (monolith)[Dra+17]. The limited functionality of each microservice allows for better maintainability[Dra+17]. In addition, changes can be made more easily because only individual microservices need to be updated[Dra+17]. Microservice applications also benefit from better scalability, as the microservices can be scaled independently with respect to their individual load[Dra+17].

However, creating microservice applications also leads to increased complexity compared to creating a monolith[Ric18]. One reason is that the communication between microservices to provide the application's functionality is more complex than communication in a monolith[Ric18]. In principle, there are two options for implementing communication between microservices: broker-based or brokerless, as illustrated in figure 2.1.

Broker-based architecture

In a broker-based architecture, a message broker acts as an intermediary for communication between the microservices[Ric18].

Brokerless architecture

In a brokerless architecture, microservices communicate directly[Ric18].

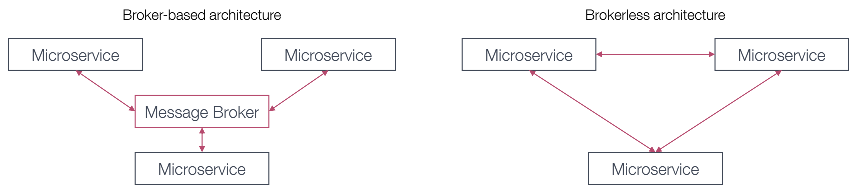


Figure 2.1.: Comparison of communication in a broker-based architecture and a brokerless architecture[Ric18]

A broker-based architecture is used by most enterprise applications because it leads to loose coupling between the services[Ric18]. In addition, a message broker can buffer messages when the system is under a high load, so that non-time-critical requests can be processed later[Ric18]. However, brokerless architectures can also be advantageous when fast application response times are critical. This is because brokerless communication leads to lower latencies between microservices and eliminates the possibility of the message broker becoming a performance bottleneck.[Ric18].

2.2. Discrete-event Simulation

Discrete-event simulation (DES) is a commonly used technique, to observe the behavior of a modeled system over time[Rob05]. The underlying assumption is that the observed **state** of the system changes only at discrete points in time[Fis01]. The modeled attributes of the state define the characteristics of the system behavior to be observed in the simulation[Fis01]. The behavior of the system is in turn determined by its modeled entities[Fis01].

Entity

An entity determines the behavior of the system by performing operations[Fis01].

An operation is thereby represented by a pair of events, which define the start and the end of the operation. Such a pair of events is referred to as an **activity**.

Event

An event represents a change in the state of a system at a point in time[Fis01].

The behavior of a system over time is then represented as a sequence of events[Fis01]. To run the simulation a **timing routine** is used to manage the sequence of events and advance in time to simulate step by step the state changes caused by the events[Fis01]. The sequence of events can then be used after a simulation to analyze the performance of the modeled system, where the accuracy of the simulation depends on the accuracy of the models used[Fis01].

3. Related Work

Contents

The first section of this chapter presents the effort of Ntentos et al. [Nte+21] for creating an automatic tool to assist in the evaluation of microservice architectures. We then examine existing approaches for performance prediction of microservice applications in section 3.2.

3.1. Evaluation of Microservice Architectures

There are many architectural patterns in the literature for building good microservice architectures[Nte+21]. However, it is a cumbersome task, especially for large architectures, to manually check compliance with architectural patterns[Nte+21]. A possible solution to help architects to create good architectures is an automatic verification of the created architectural models for compliance with architectural patterns[Nte+21].

To support such an automatic approach, Ntentos et al. [Nte+21] use a metamodel that allows microservice architectures to be modeled as a set of **components** and **connectors**. A component is an entity that interacts with other components in the context of the microservice architecture[Nte+21]. For example, microservices, databases, clients, facades are possible relevant components that can be considered[Nte+21]. Connectors describe which components interact with each other in which way[Nte+21].

To check these modeled microservice architectures for architectural pattern compliance, Ntentos et al. [Nte+21] define a catalog of Architectural Design Decisions (ADD). An ADD covers an area of a microservice architecture and defines the corresponding architectural design patterns that can be used in this area together with metrics to make the decisions in this area measurable[Nte+21]. For example, they define an ADD *Service Interconnections*, which covers different patterns for communication between services[Nte+21]. An algorithm then searches the model for direct synchronous interactions between services and fixes them, e.g. by replacing the direct synchronous interaction with asynchronous interaction through a message broker[Nte+21].

For their approach, Ntentos et al. [Nte+21] assume that asynchronous communication is usually preferable to synchronous communication. However, in some scenarios, for example when performance is of particular relevance, synchronous communication may also be preferable[Ric18]. However, the approach of Ntentos et al. [Nte+21] is not capable of predicting the performance impacts of the various component interaction options. This highlights the need for additional support to incorporate potential performance impacts of different architectures into the decision-making process.

3.2. Performance Prediction of Microservice Applications

In this section, we examine the existing approaches for performance prediction of microservice applications with respect to the concepts used and the possible uses in a **microservice architecture exploration platform (MAEP)**. We assume that at the time of using a MAEP, only architecture models of the microservice application exist and no implemented microservices exist yet.

To accurately simulate complex service chains created by the communication of different microservices Khan et al. [Kha+21] developed PerfSim. PerfSim simulates incoming requests, by simulation the resulting messaging chains between microservices that result from a request[Kha+21]. The simulation can make predictions for response times and host utilization for user-defined workload scenarios[Kha+21]. However, PerfSim requires prior extraction of network traces from an existing system[Kha+21]. These network traces are then used to create the messaging chains and information about the performance of the services as simulation input[Kha+21]. This makes this approach usable only if a microservice application already exists and is actively used. Therefore, it is not suitable to be used as part of a MAEP. PerfSim also offers the possibility to create the models for the simulation manually. But this approach has not been validated by the authors and the required models are complex which makes the practicability of this possibility questionable.

Zhang, Gan, and Delimitrou [ZGD19] have developed µqSim, a simulation framework that simulates messaging chains between microservices and the resulting message processing of microservices. To model the messaging chains between the different microservices, the user must create inter-microservice dependency graphs[ZGD19]. The processing of messages through microservices is represented using execution paths[ZGD19]. An Execution path is a detailed representation of the actual microservice logic, which consists of several execution stages that are linked with transition probabilities[ZGD19]. In this way, the execution paths precisely map the different paths in the code of the microservice implementation[ZGD19]. However, µqSim requires measurements of the individual execution stages under different loads from a real existing system for the simulation[ZGD19]. Therefore, µqSim is also not suitable for use in a MAEP.

Courageux-Sudan, Orgerie, and Quinson [COQ21] developed a simulation framework based on a simplified model of the execution logic of a microservice. Unlike µqSim, this approach does not require models of the exact execution stages of microservice logic. Instead, the processing times of a simulated microservice depend on the CPU cost, the I/O ratios and the parallelization degree of the executed logic[COQ21]. The use of this simplified model allows the use of generic tracing tools for the calibration of the input models[COQ21]. Generic traces can also be used for mapping the communication between the microservices for this approach[COQ21]. However, due to the required traces of a real existing system, this approach is also not suitable for use in a MAEP.

Bao et al. [Bao+19] represent the message processing of a microservice in their simulation by considering three sub-processes. They assume that microservices run in a container that limits the number of messages that can be processed simultaneously[Bao+19]. Based on this, they calculate the processing time as the sum of the times taken by the three sub-processes[Bao+19].

1. **Request caching time** This is the time the message waits in the queue until it is processed by the microservice.
2. **Business processing time** This is the time it takes to execute the corresponding business logic triggered by the message.
3. **Transaction processing time** This is the time it takes to perform I/O operations.

To calculate these times, they consider the hardware on which the microservices are hosted and the current load on a microservice at the time of processing[Bao+19]. For this, they consider for the hardware, the CPU clock speed, the number of CPU cores and the amount of available RAM[Bao+19]. For the hardware, they consider the CPU clock speed, the number of CPU cores, the amount of disk space, and the amount of available RAM on which the processing times depend[Bao+19]. However, their approach first requires benchmarks of a deployed microservice application on a wide variety of hardware to calibrate the simulation[Bao+19]. This also makes this approach, like the previously presented approaches, unsuitable for use in a MAEP as we envision it.

4. Concept

Contents

This chapter presents our concept for a discrete-event simulation for performance prediction of microservice applications. The basis for our approach is the metamodel of Ntentos et al. [Nte+21], which enables automatic checking for compliance with architectural design patterns. We refer to this model as the **static analysis metamodel**. The input models for the simulation should also be suitable to perform a static analysis on their own, but in addition they should enable simulation for performance prediction. We refer to this adapted model for simulation as the **simulation metamodel**.

It is not possible to reflect all possible influences of a microservice architecture on the performance in our simulation within the scope of this work. Therefore, we have chosen a bottom-up approach to develop the concepts and the implementation for the simulation. For this purpose, we first create a reference microservice architecture in order to derive the necessary concepts for the simulation of such an architecture, as presented in section 4.1. Section 4.2 presents adjustments we have made to the static analysis metamodel to enable simulation. Section 4.3 explains how the simulation uses the simulation metamodel to simulate a microservices application.

4.1. FruitCollector: A Multiplayer Game based on Microservices

In this section, we introduce FruitCollector, a multiplayer game using a microservices backend. We chose a reference architecture in the gaming domain for two reasons.

1. Previous experience in this domain allows us to quickly create a reasonable architecture.
2. The performance of the backend in multiplayer games plays an essential role, as increased latency significantly affects the players' enjoyment of the game[DWW05].

FruitCollector is intended to be representative of the performance behavior of real-world microservice architectures. In order to achieve this, FruitCollector should contain a wide range of performance influencing factors that are also contained in real existing microservice applications. For this the offered API of the application plays a crucial role. The request-response times for the functions of the API show how the performance affects the users. And the use of the API functions of a microservice application can influence the performance of the entire system in various ways. For example, some functions might start the execution of complex algorithms, while others might consume bandwidth by transferring large amounts of data, both of which could affect performance. So to be representative, the API of the FruitCollector backend should cover as large of a set of possible performance influences. To ensure high coverage, we score API functions against four factors that we expect to influence performance.

Message size We assume that functions that require a lot of data to be transferred, such as a file upload, have an impact on performance due to their increased bandwidth usage.

Frequency We assume that the frequency in which the functions are called has a performance influence.

Processing time Some functions run complex algorithms that consume a significant amount of computing power and we assume that this has an influence on the performance.

Messaging chain length Some functions lead to a lot of message exchange between the different microservices. We assume that long message chains have a greater performance influence than lower ones.

Based on our estimation, we score functions for each factor from 1–3 points, where 1 point means low expected influence and 3 points means high expected influence. We design the API of FruitCollector in such a way that we achieve a high coverage of the different factors with the functions. Furthermore, it should be ensured that we can implement an actual playable game with the given functions for the evaluation. Through this, we came up with 11 functions the API of the FruitCollector backend provides. The matrix resulting from our estimations for the functions with our evaluations for the different factors is shown in table 4.1.

Function	**Message Size**	**Frequency**	**Processing Time**	**Messaging chain length**
register	●○○	●○○	●●○	●○○
login	●○○	●○○	●●○	●○○
playerView	●●○	●●●	●○○	●○○
move	●○○	●●○	●○○	●○○
collectItem	●○○	●○○	●○○	●●○
inventoryState	●●○	●●○	●○○	●○○
sendChat	●○○	●○○	●○○	●○○
retrieveChat	●●○	●●○	●○○	●○○
retrieveQuests	●○○	●●○	●○○	●○○
finishQuest	●○○	●○○	●●●	●●●
uploadReplay	●●●	●○○	●●●	●●○

Table 4.1.: Our estimations for the scores of the API functions of FruitCollector.

Understanding the API

The API enables gameplay in which each user navigates his avatar through a two-dimensional world and tries to collect certain fruits distributed in the world in order to finish quests of the form "collect 10 fruits of type X". For a general overview of the seven services of the FruitCollector backend and for which functions of the API they interact with each other see figure 4.1. In the following we explain the functions of the API that we consider not self-explanatory. The explanations for the remaining functions are provided in appendix A.1.

playerView

Involved services World **Uses database** ✗

Description

Returns a view of the world, which is a list containing the positions and ids of all fruits and other players near the requesting player, so that they can be rendered by the game client. For performance reasons, the state of the world is saved in the service's memory so that all players connected to the same instance of the World service are playing in the same world.

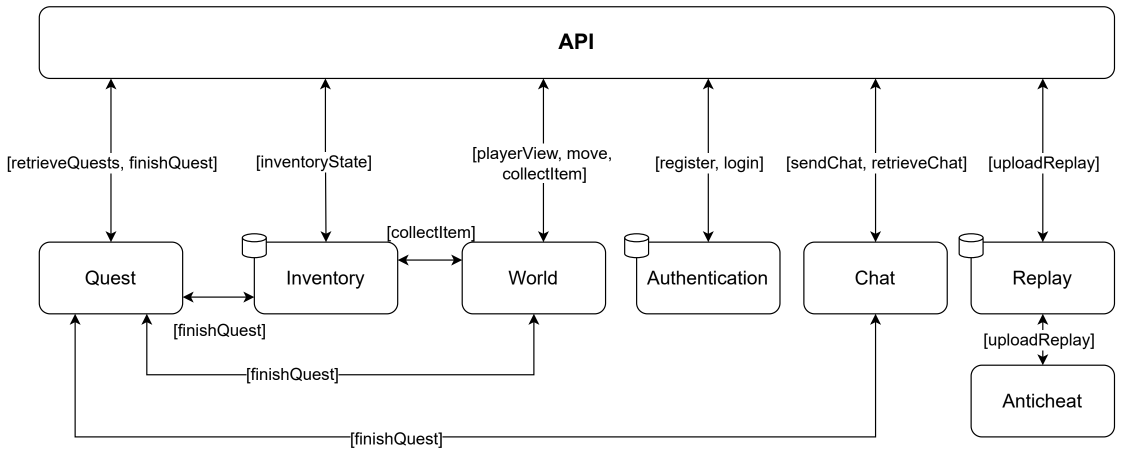


Figure 4.1.: An overview of FruitCollector's services and the API functions they realize. The cylinders indicate that a service requires a database to provide its functionality.

finishQuest

Involved services Quest, Inventory, World, Chat **Uses database** ✓
Description
When a player has collected enough fruit of a certain type to complete a quest, the game automatically calls *finishQuest*. The Quest service uses the Inventory service to check whether the player actually has enough fruit. Then the fruits of that type are deleted from the inventories of all players and new fruits are placed in the world. Finally, a new quest is created and an announcement is made in the chat.

uploadReplay

Involved services Replay, Anticheat **Uses database** ✓
Description
A player uploads a replay in the form of a list of world views received from *playerView*. The anticheat service checks if the replay is legitimate, i.e. if all movements within the replay are within the movement range. A legitimate replay is then stored in the database and the replay id is returned to the user.

4.2. Modeling a Microservice Application for Simulation

Ntentos et al. [Nte+20] model a microservice application to perform a static analysis and find architecture design decision violations. However, the static analysis metamodel lacks modeling capabilities for some information needed for a simulation for performance prediction. For example, information about the deployment hardware is not part of the metamodel, but has a great impact on the performance of the resulting application[Bao+19]. Therefore, to support simulation for performance prediction, we need to adapt the static analysis metamodel. Since a full adaptation is beyond the scope of this work, we remove the parts of the static analysis metamodel that we do not need for modeling the FruitCollector reference application. However, these removed parts are compatible with the simulation metamodel and can be added back in for future work.

This section discusses the adaptations we have made to the static analysis metamodel to obtain an adequate simulation metamodel. We reduce the number of component types, described in section 4.2.1. We extend the static analysis metamodel to allow for a more precise description of message exchanges between components, described in section 4.2.2. We add ways to model the deployment hardware, described in section 4.2.3. We also add ways to describe usage scenarios for the microservice application, described in section 4.2.4.

4.2.1. Modeling Microservice Components

Ntentos et al. [Nte+20] define a microservice architecture as a graph of component nodes and connector edges. Component nodes represent an entity that interacts with other components in the architecture. Connector edges provide information about how components interact with each other. To assign a role to each component and connector, they define 26 component types and 39 connector types.

Component type

Defines the role of a component in a microservice architecture. E.g. client, gateway, or database.

A connector's role is defined by its connector type, e.g. HTTP, synchronous, or JDBC.

An analysis of the differences in performance behavior between different implementations of services is beyond the scope of this work. Thus, we reduce the amount of component types available for the simulation model to exclude implementation specific types. For example, Ntentos et al. [Nte+20] define different component types for different database implementations. They define a *PostgreSQL DB* type and a *MySQL DB* type, both of which inherit from the Database type. In the simulation model these are removed and only a Database type is available. Furthermore, we only include types that are needed to model the FruitCollector architecture described in section 4.1. As a result we end up with five component types for the simulation metamodel:

1. **Client** A component that initiates requests to the microservice application.
2. **Facade** A component that routes requests to other components.
3. **Service** A generic microservice that executes some type of business logic.

4. **Database** A component that is used to store and retrieve data.

5. **Message Broker** A component that distributes messages between microservices.

In accordance with the static analysis metamodel, the simulation metamodel also includes the possibility to define the component nodes of the microservice architecture.

Component node

Represents an instance of particular component type that is specific to the modeled microservice architecture. For example, an authentication service that handles user login requests is an instance of the component type Service.

4.2.2. Modeling Communication

In the static analysis metamodel a connector expresses general messaging capabilities between two components. For the simulation, however, it is not enough to know which components can exchange messages with each other. If two components exchange a message once a day, this will certainly have less impact on performance than if the components exchange a message once a second. So for the simulation, we need to know the circumstances under which messages will be exchanged. Often message exchanges are part of a larger chain of messages caused by an initial message with a certain purpose. For example, a client sends an initial message with the goal of logging in, this message then triggers a messaging chain involving client, gateway, authentication service, and database. We assume that messages with the same purpose as the initial messages also trigger the same messaging chain. To model these messaging chains, we introduce the concept of request types.

Request type

A dependency graph that represents a messaging chain triggered by an initial message with a specific purpose.

The dependency graph of a request type is defined as $G = (N, E)$ for a request type R. The nodes in N represent a microservice processing a message and the edges in E represent a message transfer between two microservices. To reduce the complexity of the simulation implementation, connector types and thus the exact messaging protocol used are omitted at this point. Instead, message exchanges are treated as general network messages. Thus, the edges in E are only annotated with the size of the transmitted data.

Since the services in the FruitCollector reference architecture only become active through calls to API functions, it is sufficient to map each API function to a request type. An example dependency graph for the request type *collectItem* is shown in figure 4.2.

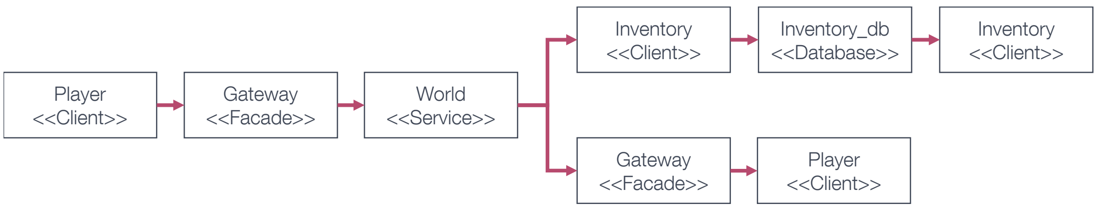


Figure 4.2.: The request type dependency graph of *collectItem*. Edge annotations omitted for brevity.

4.2.3. Modeling Deployment of Microservices

The component nodes and their interactions are not sufficient to run a simulation for performance prediction. Because the performance of a microservice application depends not only on the architecture, but also on the hardware on which the services are running[Bao+19]. However, the static analysis metamodel only considers the microservice architecture and does not consider how many instances of the different services are deployed on which hardware. To represent this information, we need to extend the metamodel to include deployment modeling capabilities.

Microservices usually run in data centers of a cloud provider[Ric18]. For the performance from the user's point of view, it is relevant in which data center the microservices are running, since the distance to the data center can influence the response times[AA19]. Therefore, the simulation metamodel needs to be able to model the data centers in which the services run. A data center consists of a geolocation and references a host topology.

Host topology

Defines the hardware and network connectivity of hosts within a data center and the distribution of components across these hosts.

Each host consists of an id, the number of available logical cores and their respective clock speeds, and the upload and download bandwidth. In addition, each host has a list of references to component nodes that are deployed on the host. By separating the data center and host topology definitions, it is easy to model multiple data centers with the same deployment configuration.

4.2.4. Modeling Usage

The performance of a microservice application also varies depending on how many users are connected and how these users behave. Different types of clients might use the microservice application in different ways. E.g. an online shop might have users that want to view the website and place orders and some external warehouse software that frequently requests shipping information. To accommodate this, the simulation metamodel includes means to define different kinds of clients for a microservice application. In order to represent the diversity of clients, the simulation metamodel includes the possibility to define **client definitions** consisting of client properties and client behavior.

Client properties

Define the geographic location of clients as well as their upload and download bandwidth.

In reality, the clients most likely won't all be in the exact same geolocation, but rather spread out over a certain area. To account for this, the geographic location of clients is defined by latitude, longitude, and a radius. This defines a circular area in the world from which clients with these client properties connect and send messages to the microservice application.

Client behavior

Defines the request types that a client triggers and the rate at which they are triggered.

A client behavior consists of a list of **request behaviors**. Each request behavior references a request type and defines a request frequency. Each simulated client then sends the appropriate simulated messages to trigger processing of that request type every X time units as defined by the request frequency. In addition, an initial delay can be defined before a client begins sending the messages repeatedly. In this way, it is easy to define a set of clients that represent normal shopping users connecting via cell phone from anywhere in France, and another set of clients that have the same behavior but connecting from anywhere in Germany.

With this approach, each client with the same client behavior behaves exactly the same and sends the same messages at exactly the same time. For user input independent requests, such as constantly requesting positions from other clients in a multiplayer game, this approach is well suited. However, this approach has limitations when it comes to describing user input dependent requests. For example, a visitor to an online store will not make a payment every X minutes. The frequency of this kind of requests strongly depends on the user's behavior. To model user input dependent requests and the underlying human behavior, probabilistic finite-state automata could serve as a complement to the frequency based approach described above[CS04]. However, to keep the effort for modeling the client behavior low, we do not adopt this approach in this work.

Microservice architectures are especially useful when scalability of the system is important[Bao+19]. Scalability is often needed when the workload, or number of connected users, can change rapidly. For example, websites for a particular product might see a rapid increase in users when the product is advertised on television[1]. Fehling et al.

[1]`https://blog.hubspot.com/marketing/how-to-prepare-website-for-traffic-influx`

[Feh+14] describe five different types of such workloads that an application may experience: Static workload, periodic workload, one-time workload, unpredictable workload, continuously changing workload. In the simulation metamodel, we therefore introduce scenarios that can be used to model these types of workload.

Scenario

Defines the workload experienced by the microservice application by mapping the number of connected clients and their client definition over time.

A scenario contains a list of **snapshots**. Each snapshot is associated with a timestamp and defines the number of connected users of a particular client definition at that point in scenario time. The simulation then simulates the period from the first to the last snapshot. In addition, the scenario contains a reference to an interpolation strategy. The interpolation strategy defines how the simulation should interpolate between snapshots when the simulation time is in between snapshots.

4.3. Simulating a Microservice Application

This section explains how the simulation uses the models discussed in section 4.2 to predict the performance of a microservice application. The performance of a microservice application is determined by two factors:

Message transfer time The time it takes to transfer messages between the different services within the application and to transfer messages between clients and the application.

Processing time The time the services need to process incoming messages.

The simulation must therefore simulate these two factors for the requests defined in the models in order to make a performance prediction.

First we discuss how the simulation uses request types as a basis to simulate the message chains for each request in section 4.3.1. Then we discuss how the simulation simulates the processing of messages by services in section 4.3.2. Due to its complexity, the estimation of message transmission times is discussed in depth in a separate chapter (chapter 5). In section 4.3.3 we discuss how the discrete-event simulation operates. Finally, in section 4.3.4 we discuss how the simulation generates output to allow analysis of performance.

4.3.1. Simulating Requests

The messaging chain between microservices triggered by an initial message is represented by a request type dependency graph. While the request types provide an overview of the messaging between components, they do not provide any information about the deployment. However, we need this information to determine processing times and message transfer times. Moreover, message processing and message transmission are complex tasks for which it would be useful to decompose them into a finer-grained representation. This, in turn, allows a simpler implementation of the simulation, since only a small subtask of the respective task has to be considered. To obtain a representation with small subtasks that also takes deployment into account, we transform the request types into Petri nets.

Petri nets are well suited to represent concurrent processes[Bar16], such as the processing of message chains by microservices. In addition, the transformation in Petri nets gives us the following benefits:

- Splitting the different simulation steps into a Petri net makes it easy to implement separation of concerns. One transition in the Petri net has one concern regarding the simulation.
- A Petri net can be easily extended at various points by simply adding places and transitions in between.
- A Petri net allows for easy debugging of the simulation implementation, since the current state of a request that has an error can be easily represented and debugged using a Petri net visualization.

For large dependency graphs, the Petri net representation with all the subtasks can quickly become confusing. However, the simulation of the components always follows the same pattern: A component receives one or more messages and then starts processing. For more clarity, we divide the Petri nets into different request stages based on these repeating patterns.

Request stage

A structure consisting of one node of a request type and all its ingoing edges.

In addition, we group places and transitions of the Petri net that are responsible for a subtask of the simulation into **segments**. We show an example request stage of a Petri net created in this way in figure 4.3. For more a more in-depth explanation of Petri net segments, refer to section 6.1.

For the simulation we use the Petri nets by creating a new **Petri net marking** for each initial message sent that belongs to the corresponding request type. This marking is then used to simulate the processing of the request by the microservice application. The segments delay the **consumable time** of the token for as long as the simulation predicts that the subtask of the segment requires. A token can only be moved through a transition if the consumable time is equal to the simulation time. Whenever the simulation time reaches the consumable time of a token in a marking all available transitions are fired. A request is considered processed when all request stages have been traversed.

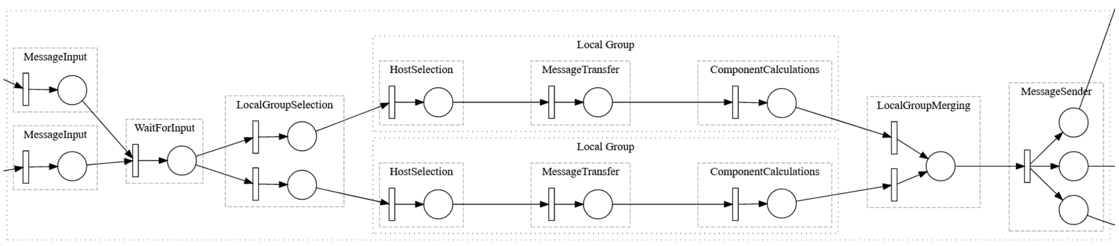


Figure 4.3.: Petri Net Representation of a request stage. Segments are grouped with dashed lines.

The segments of a request stage

In the following, we explain the tasks of each segment of a request stage. In this context, we distinguish between two types of segments:

Timed segments🕒 These segments increase the consumable time of a token when it passes through and thus increase the request-response time.

Organizational segments These segments are needed for the Petri net traversal, e.g. to create multiple tokens because a message is sent to multiple recipients. They do not affect the request-response time.

MessageInput Serves as a connector to the components that send messages for this request stage. For each message sender there is one MessageInput segment.

WaitForInput The processing of a request stage by the component can only start when all previous request stages are finished. This segment ensures this, because the transition can only fire if there is a token in each MessageInput segment. The segment also stores the instances of the services that sent the messages in the token as sender instances.

LocalGroupSelection Selects a suitable local group to which the messages should be transferred to, by using the stored locations of the message senders in the token. For the purpose of this work, the suitable local group is the one with the shortest distance.

Global groups & Local groups

The request type only describes what type of service processes a message in the request stage. However, to calculate the times, the simulation must also know which instance of a service in which data center receives the messages. A local group contains all instances of a type of service within one data center, so the simulation can decide for a target data center of a message transmission. A Global group contains all local groups of a type of service.

HostSelection Selects an instance of the service within the local group based on the load of the corresponding host and stores the selected instance in the token.

MessageTransfer◷ Since the token now contains the service instance to which the messages are sent, the message transfer time can be calculated. The segment calculates the message transfer times for a transfer of messages from all sender instances stored in the token and increases the consumable time by the highest one. The approach with which we calculate the message transfer times is explained in detail in chapter 5.

ComponentCalculations◷ The segment calculates the processing time the service instance needs for processing of the message using the approach described in section 4.3.2.

LocalGroupMerging This segment merges the paths of the different local group sections back into one path.

MessageSender This segment creates as many tokens as messages are sent in this request stage, so that the next request stages can process them in the MessageInput segment.

4.3.2. Simulating Components

The processing time required by the components to process the messages depends on the business logic executed and the hardware that executes this logic. Therefore the simulation needs a way to simulate the execution of logic on hardware. It is not possible for us to consider all possible hardware characteristics that might affect performance within the scope of this work. Since we assume that the CPU has the greatest influence on the performance, we only consider the number of CPU cores and the CPU clock speed for the simulation of the hardware. First, we discuss how to represent the complexity of tasks in terms of time. After that, we discuss the CPU model of the simulation that simulates the processing of many tasks by the CPU.

Task

In the simulation, a task is created by a service when it receives a message and represents the business logic that is executed as a result of that message.

Representing the complexity of tasks

To estimate the execution time of tasks, Buyya and Murshed [BM02] use the number of instructions the logic executes and the instructions per second the CPU can process. However, this approach has two problems:

1. The actual number of CPU instructions needed to process messages in a microservice is not intuitive. For example, Java is transpiled, then executed on the JVM, and some unknown framework code is also executed. This results in a number of actual instructions that may be significantly different from the number expected from the business logic code.

2. Instructions per second can be misleading, especially on modern CPUs. Tuomi [Tuo02] states: “One problem [...] is that the clock speed is not directly related to the amount of information processed. For example, the original Intel 8088 PC

microprocessor took on average 12 clock cycles to execute a single instruction. Modern microprocessors can execute three or more instructions per clock cycle."

For our simulation approach we do not want to use the instructions of business logic to calculate the execution time. We skip this intermediate step and directly define the time that business logic takes to execute via the base processing time.

Base processing time

The base processing time is defined as the time it takes a CPU core running at 1 GHz to process a certain task.

The base processing time is not very intuitive either, but can be measured easily. In future work, for example, a catalog of base processing times for common business logic could be provided like a login procedure, which would simplify the creation of simulation models. Within the scope of this work we cannot provide such a catalog, so for the purposes of this work we use a simple formula to calculate the base processing time t_b of all components which considers a constant time value a and the message size in kB s_m:

$$t_b = a + 0.1 \cdot s_m$$

We assume that the base processing time also depends on the size of the received message, since serialization and deserialization take longer. As value for a we use $3ms$, based on measurements of Akbulut and Perros [AP19]. In addition, the base processing time can be overridden by the user at the component node or request stage level. This simplified formula is likely to give inaccurate values for the base processing time, leading to inaccurately predicted request-response times, and needs more attention in future work.

Simulating processing of tasks

The simulation uses the base processing time of a task to predict the task processing duration.

Task processing duration

The amount of time a service actually needs to process a task, which depends on the current CPU load and the base processing time.

For this purpose we have developed a CPU model based on the principles of a CPU scheduler. When a CPU receives instructions, these are distributed over the available computing time on the different cores using a scheduling algorithm[AA18]. Within the scope of this work, we are not able to represent the complexity of real existing schedulers like the *Completely Fair Scheduler* of Linux[Pab09]. Therefore, we decided to develop a CPU model for simulation based on the principles of a first come first serve scheduler.

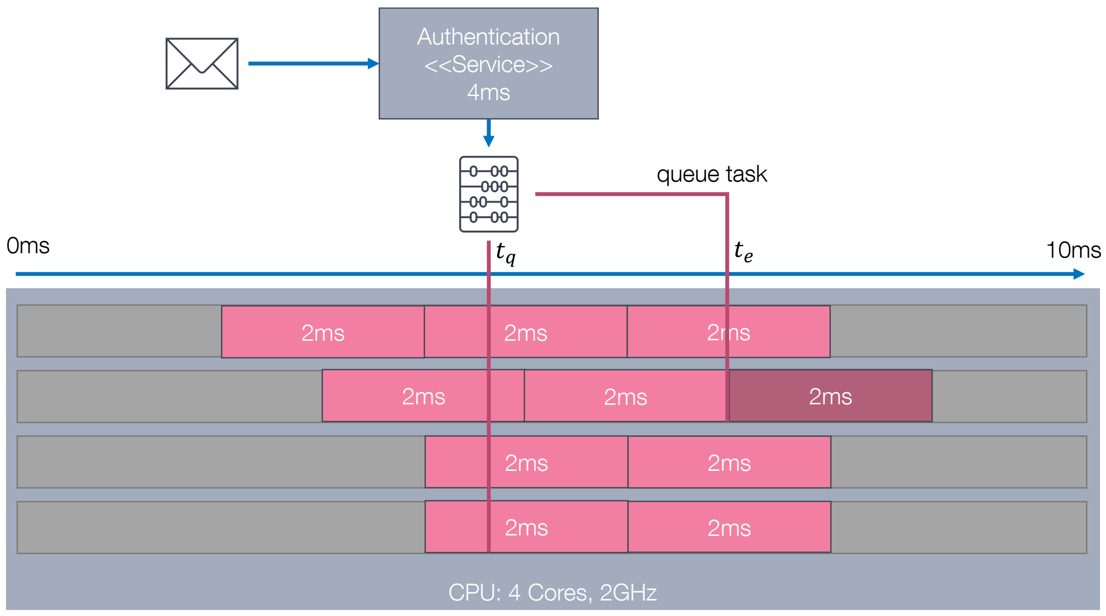


Figure 4.4.: Queuing and subsequent processing of a task in our CPU model.

We consider each task that a service wants to process as an instruction that must be scheduled on the corresponding host's CPU and then processed. A CPU consists of multiple cores, where each core can work on one task at a time. We assume that a core with 2 GHz clock speed can process a task twice as fast as a core with 1 GHz clock speed. Thus, with a core speed in GHz of S_c and a task base processing time of t_b we apply the following formula to get the **core processing time** t_c.

$$t_c = \frac{S_c}{T_b}$$

When a service wants to process a task, the task is then scheduled on the CPU and takes t_c time to be processed by a core. If the CPU does not have a free core to process the task, the task is queued and executed based on first come first serve scheduler basis as soon as a CPU core becomes available. As a result, the Task processing duration is made up of the time spent in the queue and the core processing time.

In figure 4.4 we show an example of how the Authentication service wants to process a task with a base processing time of $4ms$ for a message received at time t_q. However, no core is free at this time and the task is queued and executed at time t_e.

4.3.3. Running the Simulation

This subsection presents how the request type Petri nets, the message transfers, and CPU models are used to run the simulation. In order to run a simulation, the simulation suite needs the following input:

- An architecture, as described in section 4.2.1.
- A deployment, as described in section 4.2.3.
- A scenario, as described in section 4.2.4.
- Client definitions, as described in section 4.2.4.

The simulation is run as a discrete-event simulation. There is a global queue of events, called the simulation queue, where each event is associated with a timestamp. The event with the lowest timestamp is selected as the active event and then simulated. Then all subsequent events triggered by the active event are added to the queue, and the next active event is selected and simulated.

To fill the simulation queue with events, simulated clients are first created according to the scenario definition. During the simulation, active clients are added to or removed from a list of simulated clients according to the scenario definition. Each client initiates new simulated requests according to its defined client behavior. To initiate new simulated requests, the simulated client adds a request creation event to the simulation queue. When a simulated client creates a request, a Petri net marking is created for the Petri net of the request type and it is added to a list of active requests. The initial token is placed and the request is queued with the current simulation time as the consumable time. When the request event is handled by the simulation, the tokens are consumed and the marking is stepped from M to M'. The corresponding Petri net segments calculate the time the events take and return the new consumable times for M'. Where consumable times is a set of timestamps when the tokens in M' can be consumed by a transition. For the lowest token usage time in M', a new event is queued for the simulation. The simulation then runs until there are no active requests and no active clients.

4.3.4. Simulation Output

To provide useful insight into the predicted performance of a microservice architecture, the simulation must output data to the user. Fleckenstein, Fellows, and Ferrante [FFF18] define data as high quality if it is suitable for its intended use. However, the data provided by the simulation should not be collected with too narrow a domain of use in mind. As we want to restrict users in the analysis of the data as little as possible, so that users themselves can determine the focus of their analyses. For this reason, we decided that the simulation should not provide an interpretation of the data, but only provide data in a format that can be interpreted by the user according to his needs.

To represent the data collected by the simulation, we decided to create event logs. Using event logs has several advantages:

- We can use process mining to verify the correctness of the simulation. The processes discovered for a request type should match the dependency graphs configured by the user.

- For later evaluation with an implemented microservice application, we can create event logs in the same format. This allows us to use the same code to interpret the simulation data and its real-world counterpart.
- Event logs don't restrict users too much in their interpretation.

The simulation outputs two different event logs. A **request event log** for insight into the behavior of requests, and a **host event log** for insight into the usage of hosts.

Request event log

According to Van Der Aalst [Van16], each process mineable event log should have at least a case id and an activity column. Additionally, all events within a case should be ordered in some way[Van16]. To order the events, each event is given a timestamp of the current simulation time when it is created. Each event in the request event log should contain information about the request type to which it belongs. However, storing the request type id in each event would be too much redundant data to store. Therefore, the request event log is split so that each request type has a separate event log. For the request event log, we define the case id as the id of the simulated request to which it belongs. Each request gets a unique id from an ascending sequence of numbers when it is created. Events in the request event log are created to represent when an activity starts or ends.

Activity

Represents a piece of work of a certain duration that occurs when fulfilling a request in the microservice application.

In order to be able to calculate the duration of the activities, we add a type column, which can be either “Start” or “Finish”. Although this information is implicitly included in the event log by using the timestamp associated with each event, we decided to include it explicitly for two reasons:

1. It makes it easier to later merge two events of the event log to get the duration.
2. It allows us to sanity check the event logs of the real world microservice application. In which timestamp problems could arise due to clock synchronization issues with distributed systems.

It would be possible to include the duration of the activity directly in the event log for the simulation. However, we chose not to do so in order not to increase the complexity of the simulation and the real-world microservice application implementations.

The simulation records events for the following activities:

Transfer A message is transferred between components.

Queue A task is waiting in the queue of a CPU until it is executed.

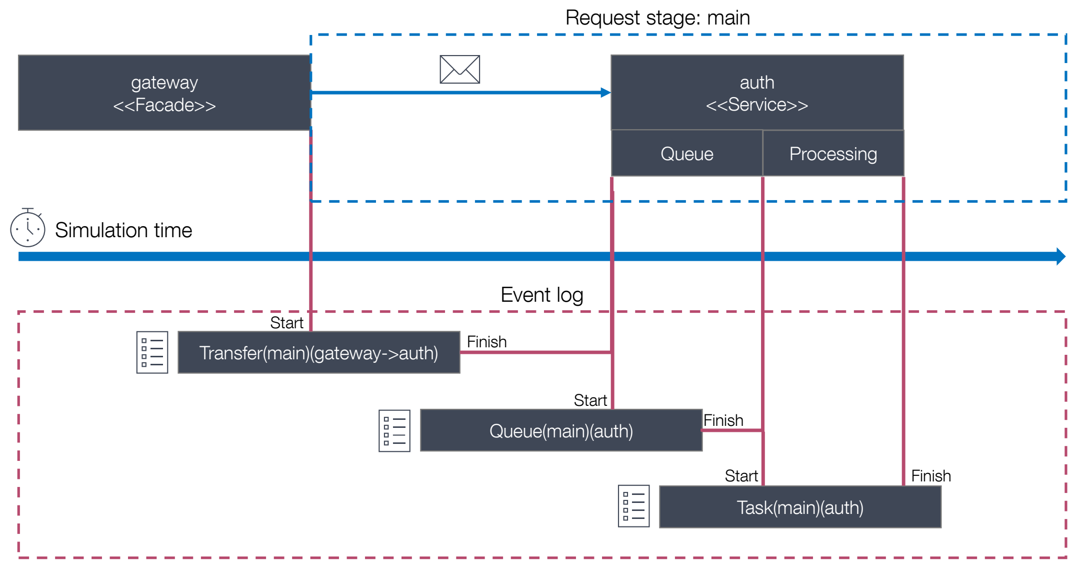


Figure 4.5.: Depiction of when events are recorded for which activity within a request stage.

Task A task is executed by a CPU.

Each activity name is generated using the component ids of the component nodes involved. To make the activity names unique within a request type, they are additionally labeled with the request stage. An example diagram illustrating when events are created with which activity name is shown in figure 4.5.

In addition, each event contains a resource name to indicate on which resource the corresponding activity was performed. For the Queue and Task activities, the resource name is the component id. For the Transfer activities, the resource is *network(A, B)* grouping all message transfer activities of the form *Transfer(X)(A, B)* and *Transfer(X)(B, A)*. See table 4.2 for a sample partial request event log for the request type *register*.

Host event log

Each event in the host event log should contain information about the host to which it belongs. However, storing the host id in each event would be too much redundant data to store. Therefore, the host event log is split so that each host has a separate event log. In the host event log, we create an event each time a host processes a task. A host can have multiple components running on it, and each component can have multiple request types in which it may be starting a task. Therefore, for the data to be useful, we need to include information about what triggered the task. To do this, each event contains the request id of the request that initiated the processing. It also contains the request type of the request and the component that initiated the task processing. To provide information about CPU usage, each event contains the index of the core that processed

id	type	activity	resource	time
0	Start	Transfer(stage1)(player->gateway)	network(gateway, player)	0
	Finish	Transfer(stage1)(player->gateway)	network(gateway, player)	6
	Start	Queue(stage1)(gateway)	gateway	6
	Finish	Queue(stage1)(gateway)	gateway	6
	Start	Task(stage1)(gateway)	gateway	6
	Finish	Task(stage1)(gateway)	gateway	7
	Start	Transfer(main)(gateway->auth)	network(auth, gateway)	7
...	...	...	...	...

Table 4.2.: Example request event log for a request sent from the player client to the gateway at simulation time 0.

request id	request type	component	core	start time	finish time
0	register	gateway	0	6	7
0	register	auth	0	10	20
0	register	gateway	0	23	24
1	register	gateway	1	23	24

Table 4.3.: Example event log for a host running gateway and authentication service.

the task, as well as a start time and a finish time for the task. With this information, a user can then derive CPU utilization on each host and find potential bottlenecks. See table 4.3 for a sample partial host event log.

4.4. Summary

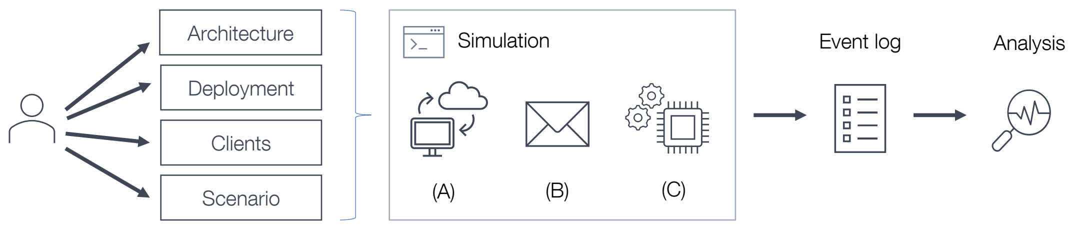


Figure 4.6.: Workflow for the performance prediction of a microservice application.

The concepts we have developed lead to a performance prediction workflow as shown in figure 4.6. First, an architect creates four models as input for the simulation:

- **Architecture** contains the components and request types.

- **Deployment** contains the datacenters, hosts and distribution of services on the hosts.
- **Clients** contains the client behaviors and client properties.
- **Scenario** contains the period of observation and corresponding the workload.

These models are then given as input to the simulation, which simulates the transfer times between clients and application (A), the transfer of messages between microservices (B), and the processing of messages by microservices (C). As output, the simulation generates an event log which can be used to perform an analysis of the microservice application's performance.

5. Predicting Message Transfer Times

Contents

The simulation uses the CPU model presented in section 4.3.2 to predict how long it takes a component to process a received message. However, in distributed systems, the cost of communication also has a significant impact on the performance of the system. For this reason, the simulation needs a way to accurately predict the time it takes to send a message between two hosts. This chapter explains how the simulation predicts the message transfer times between the different services.

One challenging part of predicting transfer times is predicting the **Round-trip time (RTT)** between two hosts. Once the RTT is known, the **bandwidth** of the connected hosts and the **message size** can be used to predict the message transfer time. In section 5.1 we present existing literature about RTT prediction systems. Section 5.2 explains our approach based on a neural network to predict the RTT and how we use the predicted RTT to predict message transfer times. Section 5.3 discusses the limitations of our approach.

5.1. Existing RTT Prediction Systems

This section presents existing approaches to predict RTT between hosts.

Wong, Slivkins, and Sirer [WSS05] introduced Meridian, a framework for performing node selection based on the expected RTT. For large distributed systems, it is not practical to measure RTT for all possible nodes individually[WSS05]. Meridian reduces the number of nodes to measure by grouping nodes into rings[WSS05]. Each node creates a number of rings around itself and measures in each step only the RTT to a node in a certain ring[WSS05]. From this measurement the RTT to the other nodes in the same ring is estimated[WSS05]. This reduces the number of measurements needed[WSS05].

One group of approaches for predicting RTT between hosts are network coordinate systems[MAB18]. Network coordinate systems for predicting RTT calculate positions for nodes in a virtual geometric space[MAB18]. For this purpose, some measurements of already placed nodes are used to determine the position of a new node[MAB18]. Based on the positions in the virtual space, predictions can then be made about the expected RTT between two nodes between which there has never been a direct measurement[MAB18].

One subcategory of network coordinate systems are landmark based system[MAB18]. In landmark based systems, when new nodes are inserted, measurements are first made to specific selected nodes, called landmarks, to determine the position of the new node in the coordinate system[MAB18]. Francis et al. [Fra+01] presented IDMaps, a landmark based RTT prediction system where landmarks are distributed in a network. The distance D between two nodes A and B with the respective nearest landmarks L_A and L_B is then calculated as: $D = distance(A, L_A) + distance(L_A, L_B) + distance(B, L_B)$.

However, landmark based systems have the problem that a small number of landmark nodes represent a single point of failure and the infrastructure can also quickly reach its scalability limits[Cos+04]. To avoid these problems, there are distributed based systems in which every node that has already been assigned a position in the geometric space can also serve as a landmark[MAB18]. Vivaldi by Dabek et al. [Dab+04] is such distributed based system. To minimize errors, Vivaldi uses Hooke's Law and simulates springs connecting the nodes in the geometric space[Dab+04]. When a node is placed, the springs exert force on the connected nodes to reduce the error[Dab+04].

The presented network coordinate systems are based on the assumption that the Triangle Inequality holds for the RTTs between different nodes[MAB18]. In other words, the systems assume that the RTT between host A and host C is always less than the RTT between host A and B plus the RTT of host B and host C, if they are arranged in a triangle shape in the respective coordinate system. However, Dabek et al. [Dab+04] noted that the Triangle Inequality does not necessarily hold in datasets for RTTs between hosts. To solve the problem of the Triangle Inequality Violation, there are matrix factorization systems (MFS) for RTT prediction[MAB18].

However, just like all the previously explained approaches, the MFS approaches also have the problem that predicting RTT from one host to another requires at least some calibration requests. So for RTT prediction using the previous approaches we need an actual host connected to the Internet. This makes the previously presented RTT prediction approaches unsuitable for determining the RTT between two simulated hosts for which we have only the information from the deployment model. Since this is unsuitable for determining the simulated message transfer time for microservice simulation, a different approach must be taken.

Beverly, Sollins, and Berger [BSB06] use a Support Vector Machine approach to predict RTTs between two hosts of which the ip addresses are known. Their approach makes use of the inherent locality of ip addresses, to be able to predict RTT between two hosts[BSB06]. However, their approach requires IP addresses to work, which are not available from the deployment model of the simulation. Since it works by inferring RTT from the locality of IP addresses, it should also be possible to predict RTT directly from the geographic locations of hosts.

Landa et al. [Lan+13] collected RTT measurements between hosts distributed around the world. They enriched their data by adding relevant network and geographic metadata to each measurement[Lan+13]. They found that the pair of countries in which the communicating hosts are located is the best predictor of RTT[Lan+13]. In addition, using the distance between the hosts is useful in further improving the prediction[Lan+13].

5.2. Predicting Message Transfer Times

The previously presented approaches to RTT prediction are not suitable for use in our simulation. They either use input we do not have in the simulation context[WSS05; Dab+04; BSB06] or do not provide datasets with which we can replicate the approach [Lan+13]. However, they do reveal, that we cannot simply use the distance between two hosts to obtain an accurate RTT prediction[MAB18; Lan+13].

But, from the input model of the simulation we have only limited information. We know the geographic location of each host through the data center locations(cf. section 4.2.3), and we know the geographic location of the clients from the client properties(cf. section 4.2.4). Using this information, we must predict the RTT between hosts and between clients and hosts. Then we can use the predicted RTTs and the defined bandwidths of clients and hosts to predict the message transfer times for the simulation. In this section we explain our approach to predict message transfer times with the limited information available to the simulation.

5.2.1. Obtaining data

In order to predict RTT from the available variables, we first need a dataset with measured RTTs and geolocation information for the hosts. However, the available datasets are either outdated or missing geographic location information, so we need to obtain a dataset first. `CloudPing.co` provides RTT measurements between hosts in different **AWS regions**. For each AWS region, the time it takes to establish a TCP connection to a database in each region is measured multiple times a day. We downloaded measurements from `CloudPing.co` for one day[1]. This gives us a dataset of 2205 measurements of RTT between different AWS regions, including the ID of the sender and receiver AWS region. Next, we obtained the latitude and longitude of the AWS data centers for the regions and enriched the data with the obtained geolocations[2]. AWS has multiple AWS regions in a geographic region, for example, the US East Coast has two AWS regions, “us-east-1” and “us-east-2”. To summarize the AWS regions we introduce AWS districts.

AWS district

Aggregates the data centers from multiple AWS regions to assign each data center an approximate region in the world.

[1]The data was retrieved on 2022-10-09.

[2]The data center locations were obtained from `https://github.com/turnkeylinux/aws-datacenters`

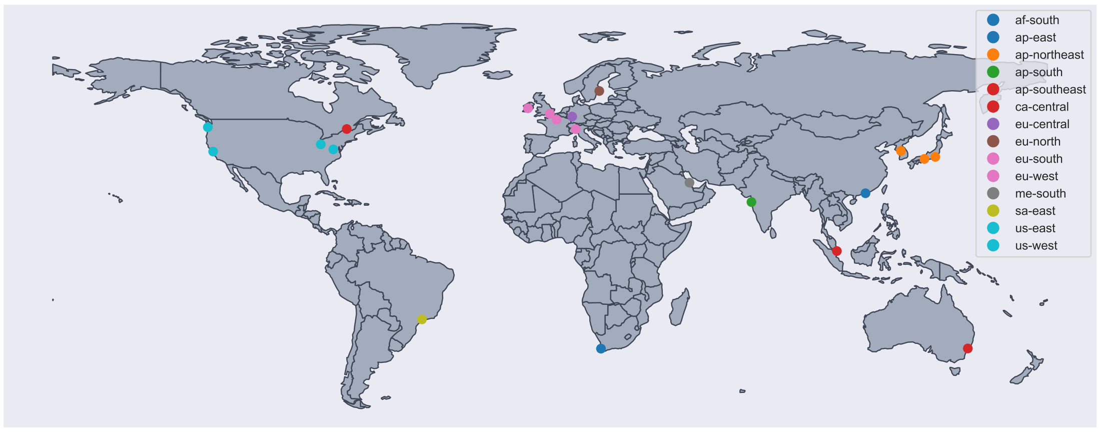


Figure 5.1.: Geographic distribution of the data centers in our dataset and their respective AWS districts.

AWS districts are obtained by removing the number suffix from the AWS region. The locations of the data centers and their corresponding AWS districts are shown in figure 5.1.

5.2.2. Predicting Round-Trip Time

Landa et al. [Lan+13] found that the distance between the two hosts is a good predictor of RTT. The distance between two hosts is given as great-circle distance on the world. Plotting the RTT from our data, as shown in figure 5.2, against the distance between the hosts seems to confirm that there is a correlation between distance and RTT. However, the data suggests that the variance of the measured RTT values increases with the distance between hosts. To confirm this hypothesis, we performed ordinary least squares (OLS) regression. The linear regression yields an adjusted $R^2 \approx 0.75$. The White test confirmed the heteroscedasticity of the data with $p < 0.001$. To further check decreasing linearity, we split the dataset into D_1 for entries with $distance < 7500km$ and D_2 for entries with $distance \geq 7500km$. Separate OLS regressions on D_1 and D_2 reported an adjusted R^2 of 0.81 and 0.63 showing that a linear regression is not suitable for higher distances.

Our results are consistent with the findings of Ziviani et al. [Ziv+05], who found that there is a strong correlation between distance and RTT in the regions of Europe or the US, but that this correlation does not hold for the Internet as a whole. One reason for this may be the routing of messages through the Internet infrastructure. The greater the distance, the greater the chance that the routing path will deviate significantly from a straight line. One reason could be the different density of Internet exchanges in different regions of the world. Another reason could be the layout of undersea cables connecting different regions of the world. For example, messages from Europe to the United States can be transmitted almost in a straight line, while messages from Europe to South Africa

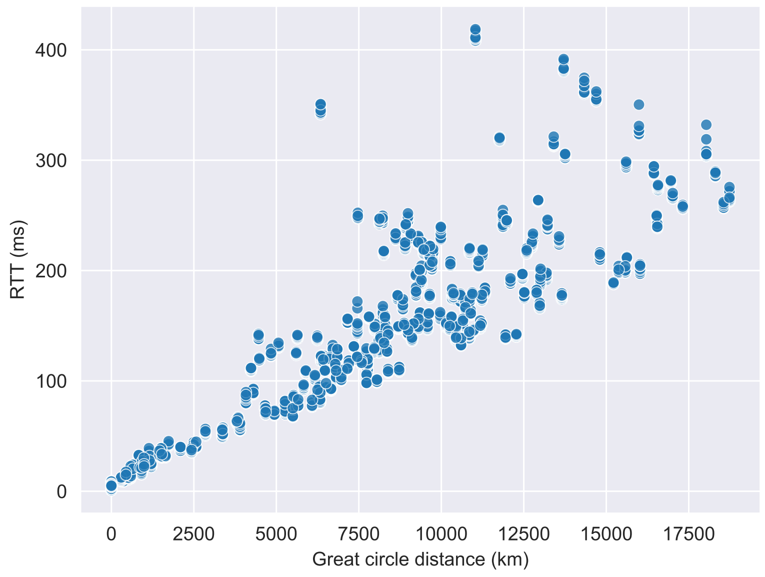


Figure 5.2.: Relationship between great circle distance and RTT in the measurements from our data set.

are likely to deviate from a straight line. figure 5.3 shows a map of Internet exchange locations and the undersea cables connecting them. For RTT prediction, this means that we need to consider not only the great circle distance, but also the regions between which the message transfers take place. This finding is consistent with the results of Landa et al. [Lan+13], who additionally consider the country pairs to improve their prediction. Based on our available dataset, we use the sender and receiver AWS district as an additional input for our RTT prediction. Including the sender and receiver AWS district as independent variables for the OLS regression yields an adjusted $R^2 = 0.83$.

Since the relationship of the great-circle distance and the RTT are not linear, a linear regression can not be accurate[PO71]. Instead, we use a multilayer perceptron (MLP) for regression, which is well suited to solve non-linear problems. A multilayer perceptron for regression is a feedforward neural network architecture to learn a mapping function from input variables to continuous output values. Since the MLP can not handle strings as input directly, we use One-Hot encoding to encode the sender and receiver AWS district categorical values. We use the rectified linear unit function as activation function. Based on an iterative process we found that an initial hidden layer of 50 neurons and one subsequent layer of with three neurons yields the highest accuracy. Applying a train-test split of 70% train and 30% test data, yields a $R^2 = 0.98$.

5.2.3. Validation

The accuracy of the multiplayer perceptron is good for the randomized train-test splits used previously. However, a randomized train-test split of the training data is meaning-

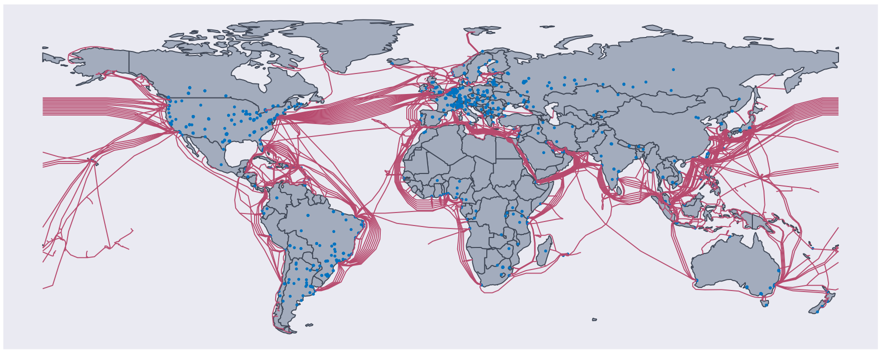


Figure 5.3.: Geographical distribution of internet exchanges and undersea cables. The data was obtained from `https://github.com/telegeography` on 2022-10-21.

less if we want to use the neural network to predict RTT to data centers that do not yet exist. If the training data contains an RTT measurement between two data centers, it is easy to accurately predict another RTT between the same data centers.

To better validate the RTT prediction, we need a more meaningful train-test split. For the simulation, we want to predict RTTs for non-existent data centers for which the neural network has no training data. To emulate this behavior, we design a **meaningful train-test split** as follows: We select a data center and move all records where the data center is either a sender or a receiver to the test set. We use all other records as the training set. Now the neural network has to predict RTTs to a data center it has never seen before. We repeat this process for all data centers where there are at least 2 data centers in the corresponding AWS district. Otherwise, the neural network has no knowledge of the AWS region for which to predict RTT. Using this approach, we end up with 12 train-test splits, for which the R^2 results are available in table 5.1

We observe, that the neural network performs poorly for the ap-southeast AWS district. A reason for that could be the great-circle distance between the ap-southeast-1 and ap-southeast-2 data centers, as shown in figure 5.1. Because ap-southeast-1 is located in Singapore and ap-southeast-2 is located in sydney, their respective RTTs to the other regions differ greatly. Data centers in other AWS districts are closer geographically and their RTTs have greater similarity. Representing the average RTTs between the different data centers in a dendrogram, as presented in figure 5.4, shows the similarity of RTT measurements. This shows, that the clustering into AWS districts based on the AWS regions is not optimal. The AWS district classification is excessively granular for data centers in the EU and US, yet inadequate in granularity for data centers in the Asia-Pacific region.

Test data center	**AWS district**	**Adjusted R^2**
ap-northeast-1	ap-northeast	0.98
ap-northeast-2		0.92
ap-northeast-3		0.99
ap-southeast-1	ap-southeast	0.04
ap-southeast-2		0.57
eu-west-1	eu-west	0.97
eu-west-2		0.99
eu-west-3		0.98
us-east-1	us-east	0.98
us-east-2		0.98
us-west-1	us-west	0.98
us-west-2		0.97

Table 5.1.: Results from the meaningful train-test splits.

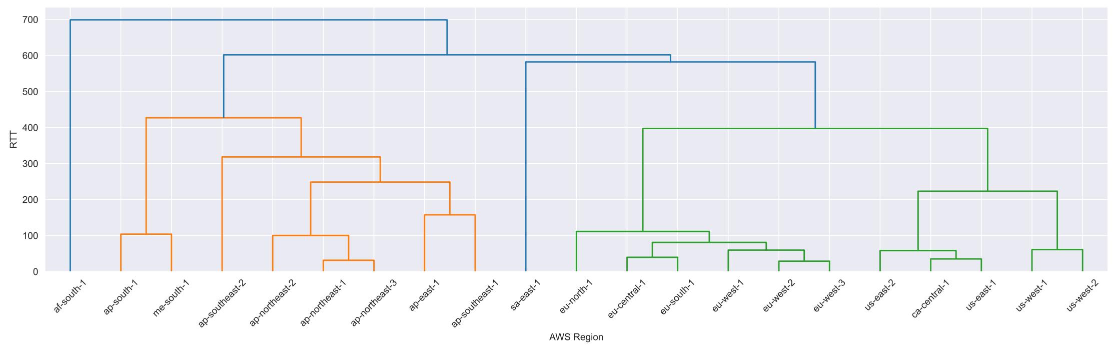


Figure 5.4.: Dendrogram of a hierarchical clustering of the AWS regions using the average RTT between them.

5.2.4. Deriving Message Transfer Time from RTT

From the neural network discussed in section 5.2.2, we can predict the RTT between two hosts. For the simulation suite, however, we are interested in how long it takes to transfer a message from one host to another. The actual time for an end-to-end message transfer is mainly influenced by the transmission delay and the propagation delay. The transmission delay d_{trans} is the time it takes a host to push a message of size L bits onto the network at a rate of R bits/sec. The propagation delay d_{prop} describes the time it takes for a packet of data to travel through the network until it reaches its destination[JK+13]. For the simulation, we assume that $d_{prop} = \frac{RTT}{2}$ and d_{trans} can be calculated with the defined bandwidth of the communicating hosts. Thus, we can predict the message transfer times t_m for the simulation with $t_m = d_{trans} + d_{prop}$.

5.3. Limitations

Predicting message transfer times by using a multilayer perceptron to predict RTT is a promising approach. However, the approach has several shortcomings.

The underlying data is based on the time it takes to establish a TCP connection. However, a TCP handshake consists of two SYN messages and an ACK message being transmitted. Thus, the assumption $d_{prop} = \frac{RTT}{2}$ oversimplifies the complexity of network messaging and may result in incorrect values for d_{prop}[CY05].

The categorization of AWS regions into AWS districts is flawed, as shown in figure 5.4. Better categorization approaches are needed to make RTT predictions reliable worldwide.

The underlying data consists only of measurements of RTT between data centers. However, computers in data centers are naturally more connected to the Internet than other hosts. Smartphones connected over cellular networks or home computers connected over copper lines may have completely different RTTs.

RTT prediction only considers the geographic location of the two connected hosts. In reality, RTT can vary by time of day or season. For example, the total traffic exchanged at DE-CIX Frankfurt can vary from 6 Tbps at night to 14 Tbps in the evening[3], which may affect the RTT.

Another issue is the distribution of AWS data centers. To predict RTT for a non-existent data center, it must first be assigned to the nearest AWS district. However, for many parts of the world, there are no data centers in the underlying data, making it difficult to assign an AWS district without compromising prediction quality.

[3]`https://www.de-cix.net/en/locations/frankfurt/statistics`

6. Design

Contents

In this chapter, we first present the design of the Petri net framework that we developed to enable the request simulation (section 6.1). Then, in section 6.2, we present the design of the simulation and how it uses the Petri net framework.

6.1. A Petri Net Framework

For the simulation the request types are transformed into Petri nets as we describe in more detail in section 4.3.1. The resulting Petri nets can be very large, so we need a data structure that allows us to easily create these large Petri nets. Also, the tokens moving through the petri nets are a main driver of the simulation, so we need to be able to link the movements of the tokens through the petri nets and the simulation logic in a meaningful way. Since we did not find a framework that supported these requirements for our chosen technology, we decided to develop a Petri net framework ourselves.

Constructing Petri nets

It is time-consuming to construct a request type Petri net from the individual places, transitions and arcs. Thus, the petri net framework introduces segments as a higher-level structure for creating Petri nets.

Segment

A structure consisting of **places** and **transitions** and directed connections between them (**arcs**). They are grouped in a segment according to their purpose in a Petri net.

To create the request type Petri nets out of segments, we have identified five different types of segments that are needed[1]. The segment types supported by the Petri net

[1]The identified segments are based on the primitive structures from `https://www.techfak.uni-bielefeld.de/~mchen/BioPNML/Intro/pnfaq.html`.

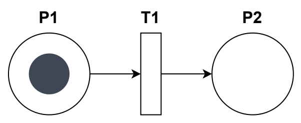
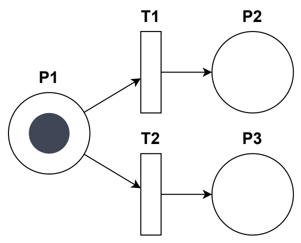
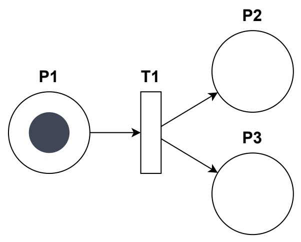
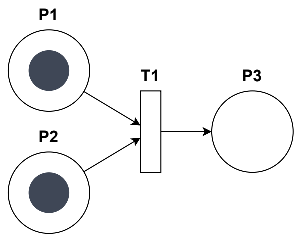
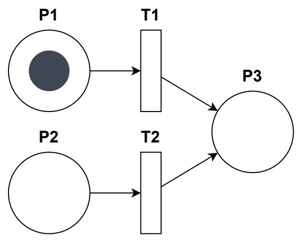
Segment type	**Visual representation**	**Description**
Sequential		If P1 holds a token the transition T1 can fire and move the token to P2.
Conflict		If P1 holds a token, either T1 or T2 can fire and move the token to P2 or P3. Can be extended with any number of transitions and corresponding output places.
Concurrency		If P1 holds a token, T1 can fire and produces one token each in P2 and P3. Can be extended with any number of output places for the transition.
Synchronization		Transition T1 can only fire if both P1 and P2 hold a token and produces one token in P3. Can be extended with any number of input places.
Merging		If either P1 or P2 hold a token, the associated transition can fire and produce a token in P3. Can be extended with any number of input places and the associated transition.

Table 6.1.: Overview of the segment types available in the Petri net framework.

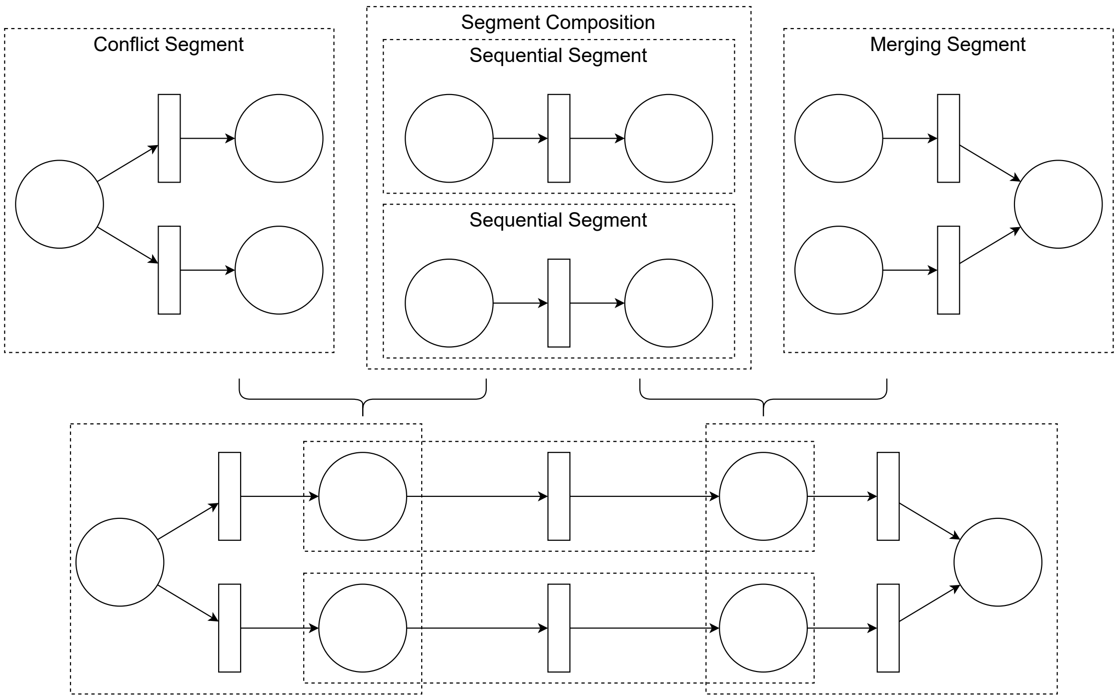


Figure 6.1.: Linking of segments to create a Petri net.

framework are presented in table 6.1. Each segment has a set of **input places** (left of the transitions) and a set of **output places** (right of the transitions). To create a Petri net, first the segments are created and the number of input and output places of the segments are defined. The transitions and arcs between the input and output places are automatically generated based on the segment type. Then the segments are linked by merging the output places of one segment with the input places of another segment. To do this, the number of output segments of the first segment must match the number of input segments of the second segment. However, this cannot be always guaranteed, for example, when linking a conflict segment with two sequential segments. In order to make these links possible, we introduce a **segment composition** for the linking of segments, which is a structure that combines several segments and can be linked to the other segments like a normal segment. In figure 6.1 we show how a Petri net can be created with a conflict segment, a merging segment and two sequential segments. The original segments remain in the resulting Petri net and can be used to execute logic, as we describe in the next section.

Petri nets & custom logic

In the following we explain which mechanisms the Petri net framework provides to be used by the simulation to represent the message processing by the microservice application. To be usable for simulation there are several requirements for the Petri net framework.

1. The simulation must be able to move tokens through the Petri net, to represent messages moving through the microservices application in a request type.
2. Tokens must be able to be deactivated for a certain time, so that time consumption for message transfers and message processing can be reflected.
3. The tokens must be able to hold attributes, e.g. about the chosen host.

To map the current state of a request of a particular request type we use a petri net **marking**. A marking contains all tokens and references to the places in which they are currently located and every token can hold attributes chosen by the framework user. To move the tokens in a marking through the petri net, the framework offers a **step** functionality. Step is a function $s(M, P)$ that takes a marking M and a Petri net P and fires all transitions for which there are enough tokens as input and thereby generates a new marking M'. To map the time consumption by operations in the microservice application, we introduce a **timed marking** in which each token is given a **consumable time** value t. For timed marking, step is a function of the form $s(M, P, T)$ where T is a time value and for the transitions only those tokens are considered that fulfill $t < T$.

To set the consumable times of the tokens and other information of the tokens the Petri net framework offers a possibility to manipulate the tokens when they pass the transitions in a segment. For each segment, the user can specify a transition hook that can be used to modify the attributes stored in the token. There is one hook that allows changes before the tokens are taken out of the input places and one hook that allows changes after the tokens are placed in the output places. This allows the simulation to specify in the tokens, for example, the hosts to which the messages are routed or the time required for message transfer.

6.2. Simulation

In this section, we first explain the input the simulation receives from the user and then show how the simulation uses the models to simulate the processing of requests by a microservice application.

6.2.1. Simulation Input

The simulation needs the information described in section 4.2 as input to run a simulation. The models are first created by the user in a readable format and are then parsed by the simulation. For this, the simulation expects four models from the user, which we explain in the following.

Architecture The architecture contains the different components of the microservice architecture and the request types that specify the communication between the different components. The request types are converted by the simulation into a Petri net representation using the Petri net framework, on which the tokens can then be moved in order to reflect processing by the system.

Deployment The deployment defines the data centers, the hosts in the data centers and how the components are distributed to the different hosts.

Clients The clients model specifies the types of clients and defines their behavior and network connectivity.

Scenario The scenario specifies which time period the simulation should simulate and how many clients of which type are connected in which time period.

The division into four input models allows the user to evaluate the performance of his microservice architecture in different scenarios, with different clients, and different deployments, and only has to exchange individual models.

6.2.2. Running the Simulation

The simulation runs as a discrete-event simulation, with a queue of events sorted by their time in simulation time. In the simulation, the events are associated with a simulation entity that does something at the time of the event.

Simulation entity

An entity that performs actions that change the simulation state.

Simulation state

The simulation state contains the current simulation time, all active clients, and all active requests.

A simulation entity that is to perform an action at a certain time is called a **simulation step** and its execution is called **stepping**. The simulation runs by taking the first event from the queue, forwarding the simulation time to the event time, and then stepping using the simulation entity from the event. This is done repeatedly until the queue is empty and the simulation state contains no more active clients. The simulation distinguishes between four different types of simulation entities which are explained in the following.

Simulation entity: Scenario

The scenario entity is stepped at regular intervals and determines the number of clients connected to the system. When stepping, the previous and the following scenario snapshot from the scenario defined by the user are considered. Then the user's scenario interpolation strategy is used to calculate the number of desired clients of a certain client definition at the current simulation time. Depending on how many clients are desired, the appropriate number of clients are added to or removed from the current simulation state. When the last snapshot is reached, all clients are removed from the simulation state.

Simulation entity: Client

Each active client is represented as a simulation entity that steps whenever a client needs to send a request of a certain request type as defined by the user in the client behavior. The simulation step creates a request simulation entity of a certain request type for each request sent and adds it to the simulation state as active request.

Simulation entity: Request

The simulation entity request represents the current state of processing a request of a certain request type sent by the client. To represent the current state of processing we use a timed marking for the Petri net of the corresponding request type. The Petri net consists of different request stages as shown in figure 4.3. The simulation extends the segment types available in the Petri net framework with its own segment types that contain the simulation logic. The two segments relevant for the calculated time processing time of requests are *ComponentCalculations* and *MessageTransfer*. *ComponentCalculations* initiates a calculation using the host simulation entity and adds the time required for the calculation to the token consumable time. *MessageTransfer* uses the approach described in Chapter 4 to calculate a message transfer time between two hosts. In addition, the segment also creates the transfer events for the request event log, as described in . For an explanation on the role of the other segments in the simulation, refer to section 4.3.1. Additionally, at the end of the Petri net a *Finish* synchronization segment is inserted that indicates that a request was processed completely if a token is in its output place. This is necessary because in case of parallel execution it might be necessary to wait for several request stages to complete. A step on the request entity executes a step on the underlying Petri net marking. The request entity then uses the smallest time of the existing tokens in the marking after a Petri net step to insert itself back into the queue as a simulation step.

Simulation entity: Host

The host simulation entity represents a host with a CPU on which the components perform calculations. When a token is consumed by the *ComponentCalculations* segment, a **Task** corresponding task is created to process the incoming message. The task is then sent to the component host, which processes the task based on the CPU model described in section 4.3.2 and returns the completion time to the segment. In addition, the host also creates the events for the time the task waits in the CPU queue and the time it is executed. The segment then increases the consumable time of the token by the corresponding value. For components of type client the corresponding host has a special *infinite CPU* on which an infinite number of cores are available for processing. Thus,

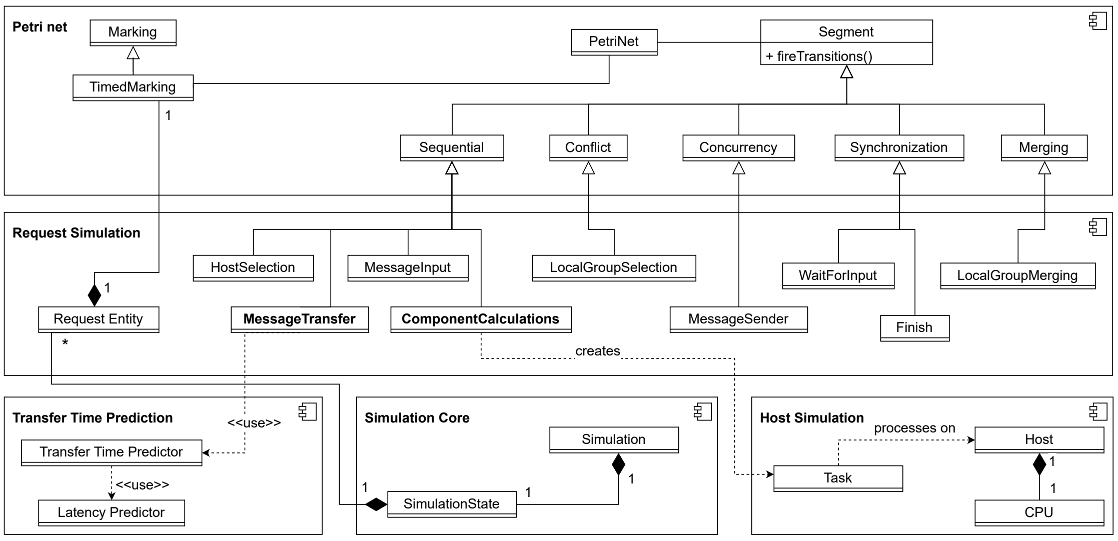


Figure 6.2.: Conceptual model of the most important components of the simulation.

there are no queue times for processing the hosts and the same host and CPU concept can be used for the clients, which in reality do not share a host.

Overview

In figure 6.2 we give an overview of the components of the simulation. The **Simulation Core** runs the simulation and holds the current simulation state. The **Request Simulation** uses the Petri net framework to represent the defined request types as Petri nets. A *Request Entity* manages the current state of the processing using a *TimedMarking* on the defined Petri net. The segments defined in the request simulation execute the individual simulation logic and use **Transfer Time Prediction** and **Host Simulation** to determine the time consumption that occurs during the processing of the requests.

7. Realization

Contents

In this chapter we first discuss how we implemented round-trip time prediction with Python. Then we present our implementation of the simulation in Java in section 7.2.

7.1. Round-Trip Time Prediction

To implement round-trip time (RTT) prediction with a multi-layer perceptron (MLP) for regression we used a Jupyter notebook with Python 3. First, we downloaded the measured RTTs between AWS data centers from `cloudping.co` using node.js and JavaScript[1]. Then we imported the data downloaded as csv as dataframe using pandas 1.4.3[2]. After that, we downloaded the geographic locations of the AWS data centers as csv from the internet[3] and imported them with geopandas 0.11.1[4] as geopandas dataframe. To calculate the great-circle distances between the data centers from the geographic positions of the data centers, we used geopy 2.2.0[5].

To investigate the relationship between distances and RTT we performed a linear regression with statsmodels 0.13.5[6] and plotted the data with seaborn 0.12.1[7] and matplotlib 3.5.3[8]. Then, we trained a neural network on the data using the MLPRegressor from scitkit-learn 1.1.2 to learn a target function that can predict the RTT between two data centers. As input variables, we used the distance between the two data centers and the one-hot encoded AWS districts of sender and receiver. As activation function we used the rectified linear unit function and we configured the MLPRegressor to use 50 neurons in the first hidden layer and 3 neurons in the second hidden layer.

[1] JavaScript code to download was adapted from `https://github.com/mda590/cloudping.co/issues/35#issuecomment-706523186`
[2] `https://pandas.pydata.org/`
[3] Data was obtained from `https://github.com/turnkeylinux/aws-datacenters/blob/master/input/datacenters`
[4] `https://geopandas.org`
[5] `https://geopy.readthedocs.io`
[6] `https://www.statsmodels.org`
[7] `https://seaborn.pydata.org`
[8] `https://matplotlib.org`

To make the RTT prediction usable by the simulation, we exported the trained MLPRegressor with joblib 1.2.0. Then we created a Python 3 script, which imports the trained MLPRegressor at startup and expects the input parameters via command line and outputs the predicted RTTs. The simulation, written in Java, then runs the python script as a process, inputs the inputs as strings, and parses the RTT outputs to predict the message transfer times.

7.2. Simulation

The simulation is implemented in Java 18 with Java Spring Boot 2.7.3[9] and built using Gradle 7.5. An implementation in Spring Boot has the advantage that a future extension of the simulation to a web service is easily possible. While the input models are passed to the simulation as local files in the current implementation, they could also be passed via a web interface in the future. Thereby the current implementation could be extended in the future with a graphical interface on the web to simplify the modeling process. In the following, we first explain the format of the input models for the simulation and then the implementation details of the simulation.

Simulation input

The input **model definitions** are passed to the simulation as JSON and parsed into Java objects using Jackson 2.13.3[10]. The simulation expects an *architecture.json* in which the component nodes and the request types are defined. Objects like component nodes or request types get an *id* property to reference them at other points of the model definitions. The properties ending with the suffix *Ref* serve as reference. In source code 7.1 we show an excerpt from *architecture.json* in which the components *player* and *gateway* are defined. These are in turn referenced in the excerpt shown in source code 7.2 to define a request type, which is shown in graph form in figure 7.1. A request type consists of several request stages and a special start stage that specifies which component sends the initial message. In the current implementation the start request stage is ignored as an initial message is always sent by a client according to his client behavior. In the future, an extension is possible at this point to enable other components to send the initial message.

In addition, the simulation expects a *clients.json* for the definition of the client properties and client behavior, a *deployment.json* for the definition of the hosts and the distribution of the components on the hosts, and a *scenario.json* for the definition of the scenario. Excerpts of these model definitions are provided in source codes A.1 to A.4 in the appendix.

```
...
"componentNodes": [
  {
    "id": "player",
    "name": "Player Client",
    "type": "Client"
  },
  {
    "id": "gateway",
    "name": "Gateway",
    "type": "Facade"
  }
],
...
```

Source Code 7.1: Excerpt of an *architecture.json* defining component nodes.

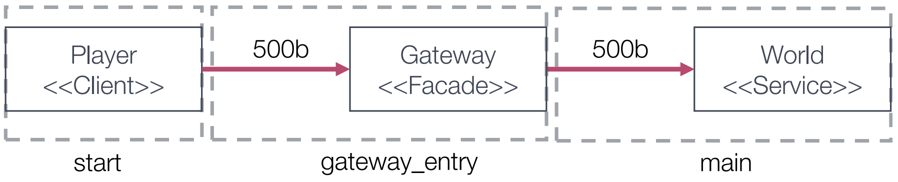


Figure 7.1.: Visual representation of the request type and the resulting request stages defined in source code 7.2.

```
...
"requestTypes": [
  {
    "id": "playerView",
    "start": {
      "sender": "player"
    },
    "stages": [
      {
        "id": "gateway_entry",
        "requires": [{
          "stageRef": "start",
          "messageSizeBytes": 500
        }],
        "receiver": "gateway"
      },
      {
        "id": "main",
        "requires": [{
          "stageRef": "gateway_entry",
          "messageSizeBytes": 500
        }],
        "receiver": "world"
      }
    ]
  }
]
...
```

Source Code 7.2: Excerpt of an *architecture.json* defining a request type.

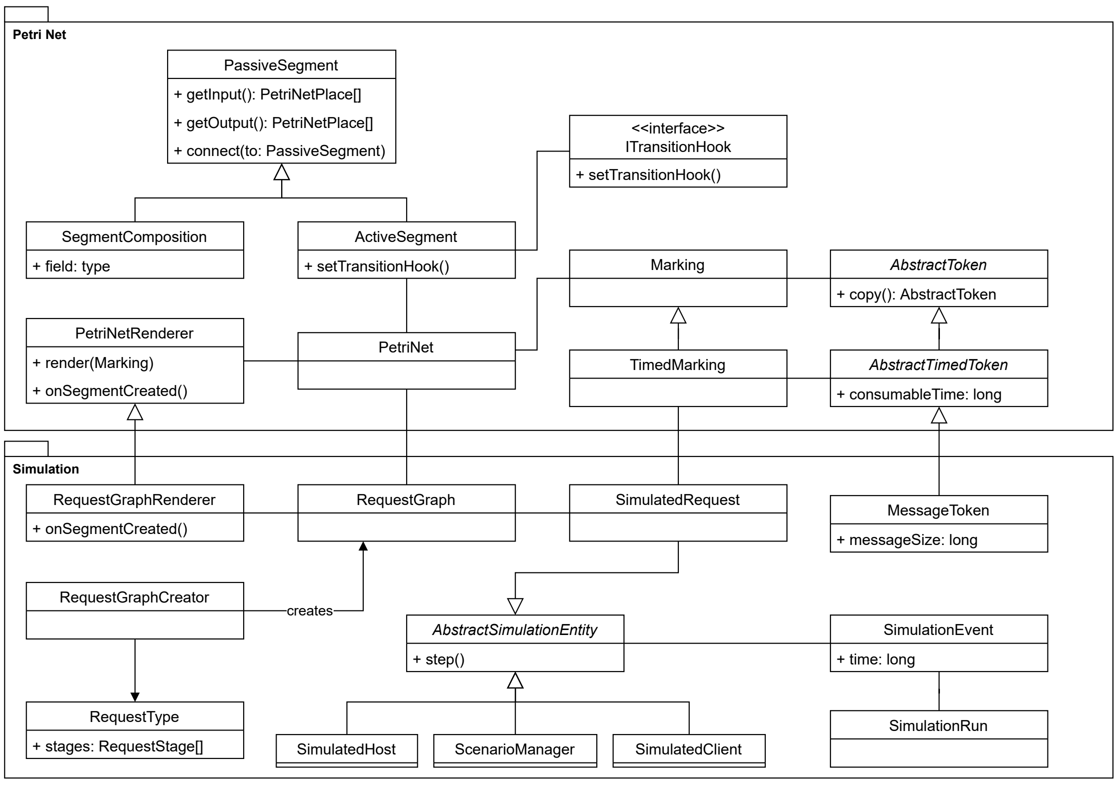


Figure 7.2.: Overview of relevant classes used to realize request simulation.

Requests & Petri nets

The simulation consists of two Java packages, one package for the simulation and one package for the Petri net framework. The Petri net package has no dependencies on the simulation package and can be separated out as an independent library in the future. In the following, we explain how these two packages cooperate in the simulation implementation to simulate the processing of requests. We provide a simplified class diagram of the relevant classes to accompany the explanations in figure 7.2.

At the beginning of a simulation, the *RequestGraphCreator* uses the parsed *RequestTypes* from the *architecture.json* to create *RequestGraphs* from them. Each *RequestGraph* represents the steps necessary to process a request with a *PetriNet*, which in turn consists of *ActiveSegments*. An *ActiveSegment* can be assigned a *ITransitionHook* via dependency injection to determine logic to be executed before and after the processing of tokens by the segment. A *PassiveSegment* does not contain logic, but only information about the input and output places and can be connected to other segments. To enable the combination of several segments for connection a *SegmentComposition* can be used. These are used, for example, to create and connect the segments assigned to a local group. Source code 7.3 shows an excerpt from the *RequestGraphCreator* which uses a SegmentComposition to combine multiple segments, the corresponding visualization is shown in figure 7.3. For visualization, the Petri net framework provides a *PetriNetRenderer* that can be extended via inheritance to display custom clustering, e.g. of local groups. The Petri nets are visualized using graphviz-java 0.18.1[11], a Java wrapper for GraphViz[12]. To facilitate debugging, the simulation also allows visualizing the current state of a request by rendering the Petri net together with the corresponding marking.

Once the *RequestGraphs* have been created, an instance of a *SimulationRun* is created which uses a queue of *SimulationEvents* to determine when a *AbstractSimulationEntity* becomes active. Each SimulationEvent has a time in milliseconds associated with, indicating when it should be handled by the *SimulationRun* The *ScenarioManager* creates *SimulatedClients*, which in turn create *SimulatedRequests* that use *SimulatedHosts* to calculate the processing times of messages. The current state of a *SimulatedRequest* is displayed on the *PetriNet* of the *RequestGraph* using *TimeMarkings* and *Messagetokens*.

Host simulation

A *SimulatedHost* is used by a *SimulatedRequest* to add the processing times of services to the token consumable times. First, tasks are created and then placed in a queue by the *SimulatedCPU* from which tasks are taken and processed on a first come first serve scheduling basis. With this scheduling strategy, it is known in advance when the task will be processed, but this is not the case with all scheduling strategies. To support other scheduling strategies in the future, the host uses asynchronous event-based programming

[9]`https://spring.io/projects/spring-boot`
[10]`https://github.com/FasterXML/jackson`
[11]`https://github.com/nidi3/graphviz-java`
[12]`https://graphviz.org`

```java
private SegmentComposition createLocalGroupSegments(RequestStage requestStage, GlobalComponentGroup globalGroup) {
  if (globalGroup.getLocalComponentGroups().isEmpty()) {
      throw new InvalidInputModelException("Request Stage '" + requestStage.id() + "' references a component '" + globalGroup.getComponentType().id() + "' that is not deployed anywhere.");
  }
  SegmentComposition localGroupsSegment = new SegmentComposition();
  for (var localGroup: globalGroup.getLocalComponentGroups()) {
      var calculations = new ComponentCalculations(requestStage, localGroup)
              .inOut(1, 1)
              .addTo(requestGraph.petriNet);
      var messageTransfer = new MessageTransfer(requestStage, localGroup)
              .inOut(1, 1)
              .connect().to(calculations)
              .addTo(requestGraph.petriNet);
      var hostSelection = new HostSelection(requestStage, localGroup)
              .inOut(1, 1)
              .connect().to(messageTransfer)
              .addTo(requestGraph.petriNet);
      localGroupsSegment
        .add(hostSelection)
        .add(messageTransfer)
        .add(calculations);
  }
  var localGroupMerging = new LocalGroupMerging(requestStage, globalGroup)
          .inOut(globalGroup.getLocalComponentGroups().size(), 1)
          .addTo(requestGraph.petriNet);
  localGroupsSegment.connect().to(localGroupMerging);
  localGroupsSegment.add(localGroupMerging);
  return localGroupsSegment;
}
```

Source Code 7.3: Excerpt of the *RequestGraphCreator* used to create a segment composition.

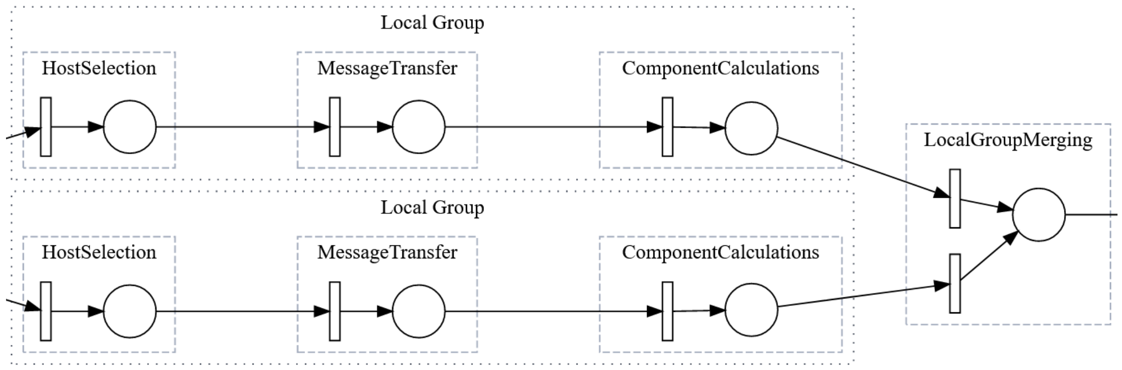


Figure 7.3.: Visualization of a *SegmentComposition* created by the *RequestGraphRenderer* with the code shown in source code 7.3

using RxJava 3.1.5[13]. When a task is placed in the queue, initially only an observable is returned, which then emits the finish time of the task at a later time when it is removed from the queue.

Event logs

The event logs of a simulation are created as Java objects and serialized as JSON at the end of a simulation with Jackson. The resulting JSON is divided into two objects as shown in source code 7.4. The *requestTypeStats* contains the transfer, queue, and task events divided for the different request types. While the *hostStats* contains the events that can be used to take a closer look at the usage of a host. The event logs use the ids specified in the model definitions for the references to components or hosts.

For the analysis of the event log we used a Jupyter notebook with Python 3 and imported and processed the data with pandas 1.4.3. For plotting the data we used seaborn 0.12.1. We also used pm4py 2.2.32[14] to generate Petri nets from the event logs. Then we compared the generated Petri nets from pm4py with the Petri nets generated from the simulation to check the reasonableness of the generated event logs.

[13]`https://github.com/ReactiveX/RxJava`
[14]`https://pm4py.fit.fraunhofer.de/`

```
{
  "requestTypeStats": {
    "uploadReplay": {
      "requestEvents": [
        {
          "requestId" : 3166,
          "type" : "Start",
          "time" : 10000,
          "activity" :
              "Transfer(gateway_entry)(player->gateway)",
          "resource" : "network(gateway<->player)"
        },
  ...
  },
  "hostStats": {
    "Host(host-2,0)": {
      "hostEvents": [
        {
          "requestId": 0,
          "requestDefinition": "register",
          "component": "auth",
          "core": 0,
          "startTime": 11,
          "finishTime": 13
        },
  ...
}
```

Source Code 7.4: Excerpt of a *statistics.json* event log generated by a simulation.

8. Evaluation

Contents

The goal of the evaluation is to compare the simulation of a microservices application with the actual behavior of a real microservices application. The results can then be used to check the accuracy of the simulation prediction and to find the reasons for possible differences between prediction and reality. For the evaluation, we consider erroneous simulation of microservices behavior and erroneous assumptions in the models that serve as input for the simulation as possible sources of error. In order to understand potential problems that may occur when modeling the simulation input, we mimic the actual workflow when using a microservices application simulation. For the evaluation, we assume that the simulation is used to facilitate a decision between brokerless messaging and broker messaging.

In the evaluation, we proceed accordingly so that we first create the models for the simulation as described in section 8.2. Then, based on the models, we implement and deploy the actual microservices applications as described in section 8.3. Afterwards, we show the collected results in section 8.4. Then we explore the reasons for the differences between simulation and reality in section 8.5.

8.1. Approach

For the evaluation of the simulation we created two microservice applications with identical functionality. The functionality of the microservice applications is described in section 4.1. The difference between the two microservice applications is the messaging pattern used for communication between the services. One application uses brokerless messaging, where the service exchange messages directly[Ric18]. The other application uses broker-based messaging, where the messages between services are passed through intermediary message broker[Ric18]. We expect the brokerless application to have a better latency[Ric18].

For both applications we first created the models for the simulation suite. Then we ran the simulation for both modelled applications for the scenario described in section 8.1.1. After that we implemented both applications and deployed them in the cloud, closely following the models defined for the simulation. Then we ran the same scenario for the deployed applications to obtain the real system results. A comparison of simulation results and real system results then gives us insights into whether the simulation is able to provide meaningful information.

8.1.1. Evaluation Scenario

For the scenario, we define a two-minute window of scenario time in which each client creates an account and plays the FruitCollector game described in section 4.1. At 0 seconds, 25 clients connect to the server. Between 0 and 60 seconds there are 25 clients in total connected, we call this time interval **phase 1**. At 60 seconds, 25 new clients connect to the server. Between 60 and 120 seconds there are 50 clients in total connected, we call this time interval **phase 2**. After 120 seconds the scenario stops and the clients stop sending requests. After connecting, each client must register with the server and then log in to obtain an authentication token. Once a client has received an authentication token, it can begin playing the game by moving around the world, collecting fruit, and completing quests. After 10 seconds of being connected each client uploads a replay containing a list of all the playerViews received so far.

8.2. Modeling for Simulation

Accurately modeling the behavior of clients for the simulation is a challenging task. In reality behavior of the clients depends on the behavior of a human player. For example some players might constantly move through the world to finish quests while others might stop often to chat with other players. The request types whose frequency depends on human behavior are designated as behavioral request types. While behavioral request types pose a challenge for predicting behavior of human clients, we can somewhat mitigate this issue for the evaluation. For the evaluation we will use the models created for the simulation to define the behavior of the clients for the reference application. This assures that the reference application client behavior closely resembles the simulated client behavior for most request types. However, the similarity of simulated client behavior and real client behavior can not be guaranteed for all request types. Some request types can only be executed once a condition is fulfilled, which we call conditional request types:

- *Login* can only be executed once *register* has been called and an account has been created.
- *FinishQuest* can only be executed once a client has collected enough fruits.
- *CollectItem* can only be executed if there is a fruit in the player's vicinity.
- All request types except *register* and *login* are only possible if the client has obtained an authentication token from *login*.

Since the client behavior for a request type is modeled only by its frequency and an initial delay, it is not possible to represent these conditions for the simulation. Therefore, we make some simplified assumptions for the client behavior model regarding conditional requests:

- We assume that *register* and *login* are sent at the same time when a client connects.
- We assume that a client collects enough items to finish a quest once every minute.
- We assume that there are enough fruits available so that a client can always collect a fruit.
- We assume that a client immediately has an authentication token and can start all requests immediately.

Thus, we only categorize *login* and *finishQuest* as conditional request types. This leads to the problem that the behavior of the clients in the simulation and in the real system application do not match. However, this problem is likely to occur in any real-world usage of the simulation suite. Therefore, the evaluation can also give us hints whether the simulation can generate useful information for real use cases.

Request types that are neither behavioral nor conditional are easy to model for the simulation suite. For example, the *playerView* request acts as a client tick that synchronizes the client game world state with the server game world state. This is a periodic request that is defined by the game code and does not depend on any user behavior. Other periodic requests defined by the game code are *inventoryState*, *retrieveChat*, *retrieveQuests*. The defined client behavior for the simulation and the categorization of the request types is shown in table 8.1. An example of the resulting request times from the defined client behavior is shown in figure 8.1.

For the definition of the message sizes for the requests we sort each message into one of four categories.

- Small: 340 bytes

Request Type	Frequency	Delay	Behavioral	Conditional
register	once	0	✓	
login	once	0	✓	✓
playerView	$8/s$	0		
move	$1/s$	0	✓	
collectItem	$1/s$	0	✓	
inventoryState	$1/s$	0		
sendChat	$1/min$	15s	✓	
retrieveChat	$1/s$	0		
retrieveQuests	$12/min$	0		
finishQuest	$1/min$	5s	✓	✓
uploadReplay	once	10s	✓	

Table 8.1.: Frequencies and classifications of the different request types sent by clients in the evaluation scenario.

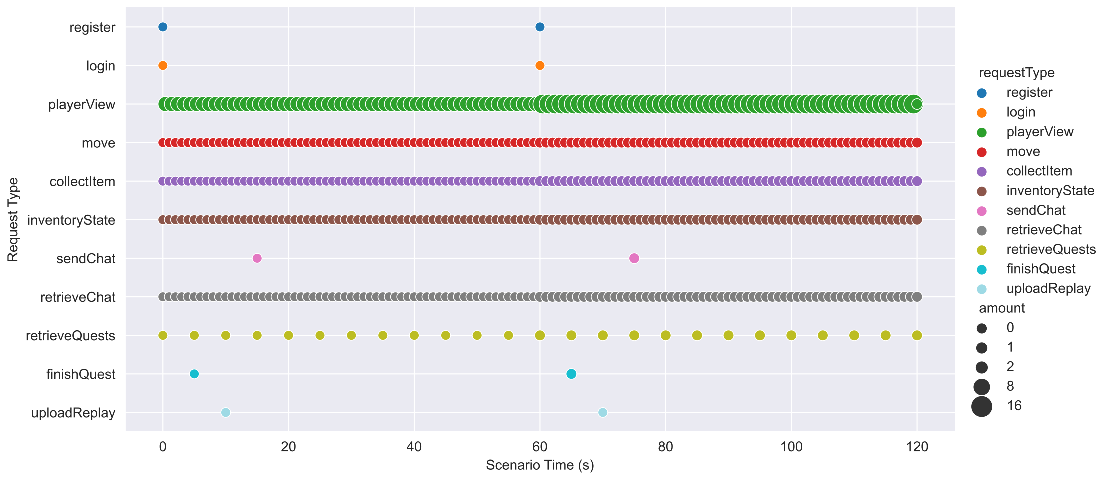


Figure 8.1.: Overview of the requests that are sent from the clients in the evaluation scenario. Shown for one client connecting at 0 seconds and one client connecting at 60 seconds.

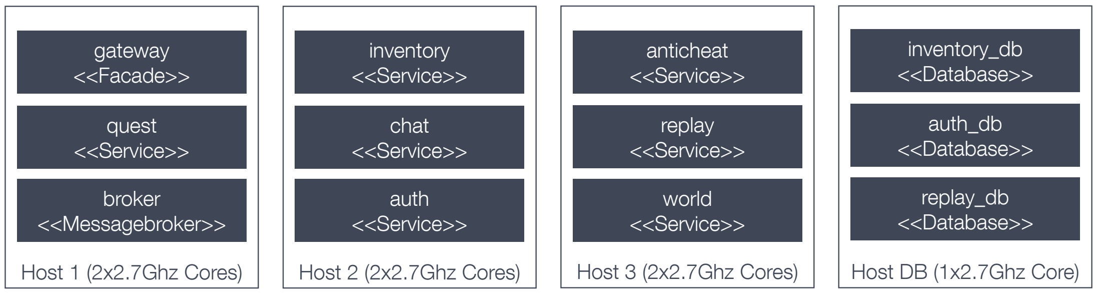


Figure 8.2.: Distribution of the services on the hosts.

- Medium: 500 bytes
- Large: 1500 bytes
- Replay Transfer: 2MB

We measured the size of an HTTP GET request from a browser with no payload in Wireshark, which transferred about 300 bytes. We add 40 bytes to account for a minimal json payload and an authentication token, which is also transferred for most request types, to end up with 340 bytes for a small message. For a Medium message, we assume a slightly larger payload, e.g., a service sending a list of quests to the client, along with their names and item types. For a large message, we assume the payload is significant, e.g. the world service sending a list of all items along with their locations near the player. The Replay transfer is the estimated size of a 10 second replay, consisting of a list of 10 seconds worth of *playerView* jsons. The client bandwidths are defined to match the bandwidth of the machine that later simulates real players for the reference applications.

The component processing times are not overwritten for the simulation and remain at their default value, described in section 4.3.2. The deployment model is defined based on the specifications provided DigitalOcean, where the reference application is hosted. The microservice application is hosted in a datacenter for which we use the latitude and longitude of London[1]. The services of the microservice application are distributed on three hosts as seen in figure 8.2. Due to budgetary constraints for the reference applications cloud deployment we only evaluate with one replica per service.

8.3. Reference Applications

To evaluate whether the simulation suite is actually useful for predicting real-world application behavior, we build the FruitCollector game as a real playable microservice application. The reference application is built as close as possible to the models used for the simulation. However, some aspects of a real microservice application, such as a

[1]Latitude: 51.509865 Longitude: -0.118092, taken from `https://www.latlong.net/place/london-the-uk-14153.html`

discovery service, can not be represented in the simulation models. Differences between the reference applications and the simulation are explained in section 8.3.3.

The individual services were implemented in Java using Spring Boot. Spring Boot was selected as the technology for this due to prior experience using it. For the Gateway we used Spring Cloud Gateway[2] and the for a discovery service we used Eureka[3]. The public API of the services is available via HTTP from the gateway. For the message broker we use Apache ActiveMQ[4].

The game world state contains a list of all the players and items associated with their respective positions in the game world. The world service uses an R-Tree to store the locations of the items. This allows the world service to quickly find all items in the players vicinity and only send those to the client, which is needed to reduce the size of the *playerView* response. For performance reasons each instance of the world service keeps the continuously changing game world state of one game world in its memory. The inventory, authentication, and replay service persist their data in a relational database.

For deployment, each service is packaged as a Docker image. The Docker containers are then orchestrated using Kubernetes. We chose DigitalOcean Managed Kubernetes[5] as the cloud deployment solution because of previous experience with it. For the database, we chose DigitalOcean Managed PostgreSQL[6].

The Kubernetes cluster is configured so that the services are distributed on the available hosts in the same way as in the simulation model shown in figure 8.2. We use the DigitalOcean General Purpose Droplets which have two vCPUs with a clock speed of 2.7Ghz and 4GB of RAM as hosts[7]. For the Managed Database we use the Starter plan with 1 vCPU and 1 GB of RAM. The hosts have upload and download bandwidth of 2 Gbit/s[8]. The speed of the database vCPU could not be found out through the Digitalocean documentation, but we assume that it is also clocked with 2.7GHz. The Kubernetes cluster and the managed Database are hosted in the LON1 region located in London.

For budgetary reasons, we cannot run the 50 clients for evaluation on 50 different machines. Therefore, we implemented a player simulator in Java that plays 50 computer players that behave similarly to the client behavior defined for the simulation. The player simulator runs on a Windows machine with 32 GB of RAM and an AMD Ryzen 7 3700X 8-core processor which is located in Aachen. The machine has a measured download bandwidth of $\approx 944 Mbit/s$ and an upload bandwidth of $\approx 839 Mbit/s$.

When a player simulation is started, the player immediately registers and creates an account. Once the account is created, the player logs in and waits for the authentication token in response. Once a player receives their authentication token, they begin playing

[2]`https://cloud.spring.io/spring-cloud-gateway/reference/html/`

[3]`https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-eureka-server.html`

[4]`https://activemq.apache.org/`

[5]`https://www.digitalocean.com/products/kubernetes`

[6]`https://www.digitalocean.com/products/managed-databases-postgresql`

[7]`https://docs.digitalocean.com/products/droplets/concepts/choosing-a-plan/`

[8]`https://docs.digitalocean.com/products/droplets/details/limits/`

the game. A player chooses a random location in the game world as their destination. He then sends periodic *move* requests to move toward the destination in small steps. When he reaches the destination, he selects a new destination. Throughout the game, the player periodically collects items from the environment. There are always enough items in the game world for a player to collect an item if they wish. A player completes a quest when they have collected enough items to do so. After a 15 second delay, a player sends a chat message every 60 seconds. After 10 seconds, a player uploads a replay, which is a list of all player view responses received so far.

To collect the data for the reference applications, we first let the player simulator run through the simulation scenario without collecting statistics as a warm-up run. Immediately afterwards, the database tables are cleared and the states of the stateful services are reset. After that, the player simulator simulates the simulation scenario and statistics are collected. The warm-up run is necessary to avoid possible distortions of the statistics by a cold start. For example, TCP connections are initially slower due to the congestion control algorithms used[Gou+02].

8.3.1. Microservices Design

The microservices share a common codebase that contains the public interface definitions of all available services and the messaging logic. For message handling, a service implements an IMessageHandler interface for each message type it handles. For example, the authentication service has a RegisterHandler that takes username and password as input and the new account id as output. Each service has an IMessageReceiver that is responsible for receiving messages, finding the correct message handler, and forwarding the message content to the correct message handler. The abstract class AbstractMessageReceiver contains the message forwarding logic. There are two versions of the AbstractMessageReceiver, the RestMessageReceiver, which handles HTTP messaging for the brokerless application, and the BrokerMessageReceiver, which handles JMS messaging for the broker application. The same concept applies to sending messages. The IMessageSender interface has two implementations, RestMessageSender and BrokerMessageSender.

This design decouples the business logic of each service from the messaging logic. Depending on the configuration, either the broker or brokerless versions of the message receivers and senders are instantiated on the services. This allows quick switching between the broker and brokerless versions when deploying the application for evaluation in the cloud. A diagram showing how the Authentication service uses the shared classes to realize its functionality is shown in figure 8.3.

8.3.2. Statistics Collection

Each message contains statistical metadata so that the system can collect an event log similar to the simulation. When a player sends a request for a request type to the gateway, the message is assigned a unique number from an ascending number sequence. This number is used as the request ID and is shared by all messages triggered by that

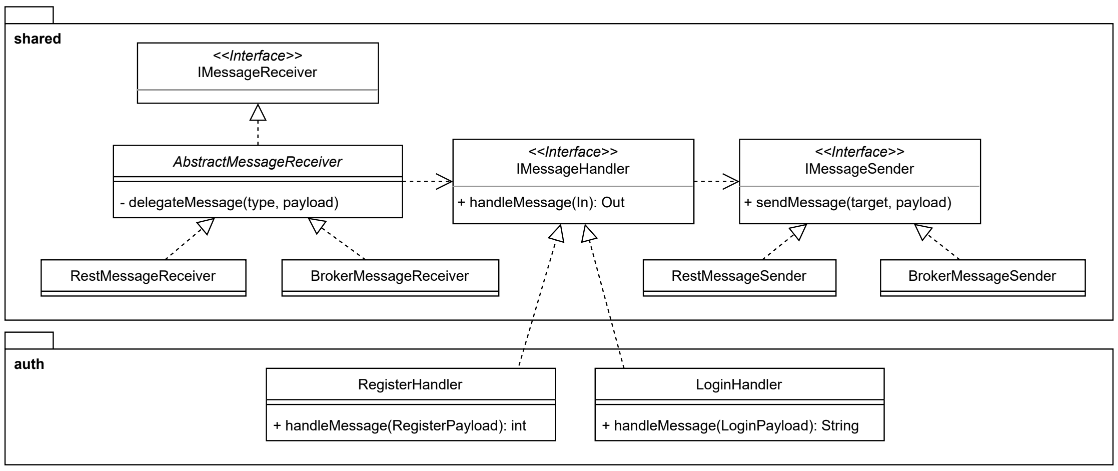


Figure 8.3.: Overview of how the authentication service uses the shared classes to provide its business logic for brokerless and broker-based communication.

request. Each message also contains the request stage ID to which the message and subsequent message handling belongs. Additionally, each message contains the request type to which it belongs, since some message handlers are called in multiple request types, this information is necessary.

To collect statistics, each service has its own event log. The player simulator also has its own event log. We use Spring Boot's messaging hooks to create the events for the event log. The timestamp for each event is the current system time in milliseconds when the event is created. For Task Start and Transfer Finish events, the events are logged as soon as Spring Boot receives a message, before any serialization has taken place. For the Task Finish and Transfer Start events, the events are logged after deserialization has taken place, just before a message is sent. Thus, the length of task activity in the reference application should be the same as the queue and task activity from the simulation. We have not found a way to log the CPU core on which a message is being processed. In addition, due to constant context switching on the CPU, it is difficult to definitively assign a start and end time to the processing of a message. Therefore, we do not collect host statistics for the reference application.

In addition, we have not been able to find a way to log the PostgreSQL database processing times for each call. Therefore, we assume that database processing starts when a service commits a transaction to the database and ends when the transaction completes. We get these times by extending the JpaTransactionManager of the services.

For the message broker implementation chosen, we were also not able to find hooks to precisely log the timings of messages being forwarded. For this reason, the processing time on the message broker is part of the message transfer activities.

To collect the statistics event logs, the gateway provides a method that collects all the individual event logs and merges them. The player simulator receives this merged event log and merges it with its own event log, resulting in an event log in the same format as the one generated by the simulation. This allows us to use the same code to analyze the simulation event logs and the event logs of the reference applications.

Perfect clock synchronization cannot be guaranteed in distributed systems[Gen+18]. There are protocols that allow clock synchronization in the nanosecond range, but they require special hardware[Gen+18]. Since we need accurate time measurements in milliseconds on 5 different machines to create the event logs, we deploy an NTP time server in addition to the services. NTP allows time synchronization with an accuracy of tens of milliseconds[Gen+18]. We use chrony[9] as implentation for our NTP time server. The clock drift between two hosts can be up to 30 microseconds per second[Gen+18]. So over the 120 seconds of a scenario the clocks can accumulate up to 3.6 milliseconds deviation. To prevent this, the services and the player simulator continuously synchronize their clocks with chrony during the entire run. To avoid possible outliers caused by the NTP protocol, only synchronizations that change the current time by a maximum of one millisecond are accepted after the initial synchronizations.

8.3.3. Differences between the Simulation and the Reference Applications

We tried to build the reference applications as close to the simulation models as possible. However, some differences between the reference applications and the simulation models still remain.

There are some differences due to deployment in the cloud. The simulation assumes there are only the modeled services running on the hosts. However, due to the managed Kubernetes solution from DigitalOcean there are 20 more services used for management of the Kubernetes Cluster from DigitalOcean running on the hosts. For the services to be able to find each other, we deploy an Eureka discovery service. The simulation models do not contain a discovery service. Since the idle CPU usage of the Kubernetes hosts without any FruitCollector services is always $< 2\%$, we assume that the performance impact of these additional services is negligible.

The simulation assumes that each host is a physical host with a dedicated CPU. However, for budgetary reasons, we chose to use virtual hosts with a vCPU. The performance of a vCPU can vary depending on the load on the underlying physical processor. Therefore, the hosts on which the services are hosted may actually have more processing power to handle the requests than is defined in the simulation. In addition, the simulation only takes into account the processing power of the CPU, but in reality, the amount of RAM available can also affect performance. During test runs for the simulation, RAM usage remained below 70%, so we assume that the amount of available memory is not a limiting factor.

There are also some differences due to the collection of statistics on the services. Each time a message is processed by a service, more computing power is required than under normal circumstances because additional events are stored in the event log. Also, because of the statistics header, the messages sent are slightly larger than they would be without statistics. This behavior is not included in the simulation models. In addition,

[9]`https://chrony.tuxfamily.org/`

chrony is running on one of the hosts to allow time synchronization between the various services and the player simulator. This additional component is also not represented in the simulation models.

Another difference is the difference in behavior between simulated and real clients. The simulation assumes precise request timing, while the message timing of the player simulator is subject to hardware and network limitations. In the simulation, 25 clients call *uploadReplay* with a delay of exactly 10 seconds of scenario time. For the player simulator, the upload is scheduled to occur 10 seconds after a player connects. However, because all players are simulated on one machine, I/O and CPU limitations can delay this call for some time.

8.4. Results

This section presents the collected results of the simulation and reference applications. The metrics used to analyze the event logs are presented in section 8.4.1. The results of the simulation are presented in section 8.4.3. The results of the reference applications are presented in section 8.4.4.

8.4.1. Metrics

Since the simulation and reference applications generate event logs, we first need to define metrics to extract analyzable data from the event logs. This section presents the metrics we use to analyze the event logs generated by the simulation and reference applications.

Request metrics

We define the following metrics for a request:

Request Completion Time (RCT) This is the time it takes the application to finish all activities triggered by a client's message of a certain request type. Each request generates a set of n events with timestamp $E = e_1, e_2, \ldots, e_n$. The operation $first_t(E)$ returns the lowest timestamp from E. The operation $last_t(E)$ returns the highest timestamp from E. The request completion time for a given request is defined as $RCT(E) = last_t(E) - first_t(E)$. Note that for *collectItem* and *finishQuest*, this is not necessarily the response time for the request a client experiences. For example, for *collectItem* the client may get a response before the collected item is added to his inventory by the inventory service. To keep the analysis of the results short, we decided not to record the response times separately. Since with the RCTs we can look at how long the system is influenced by the processing of a request and still observe the response time for the client for a large part of the requests.

Activity Duration The request completion time is a result of the durations of the activities needed to complete a request. Each activity has a start and a finish event, the activity duration is the time between these events. The activity durations can give insights into what services or message transfers in the messaging chain of a request type cause high RCTs.

Resource Duration Each activity has a resource associated with it. The resource duration is the average activity duration of all activities on a resource. The use of the resource duration as a metric has two advantages: first, the clarity can be maintained even with request types with many activites, and second, it is easier to see which resource represents a possible bottleneck.

Host metrics

We define the following metrics for a host:

Host Queue Size This is the amount of tasks that are waiting to be processed on the CPU of a given host at a given time. The host queue size does not take the amount of work behind a queued task into account. So a higher host queue size does not necessarily mean that there is more work waiting to be processed.

Host CPU Usage We define the CPU Core Usage as the fraction of time in a given time interval in which the CPU is busy with processing tasks. The Host CPU Usage for a given time interval is the sum of the CPU Core Usages for all cores on the Host divided by the amount of cores.

Visualizations

Request completion times To visualize request completion times, we map each request with an event log E to a start time $t_s(E) = first_T(E)$. Then we create time intervals with a length of 2.5 seconds. Requests of the same request type with start times within the same time interval are grouped and displayed with markers. The larger the marker, the more requests of that type were created in that time frame. The position of the marker on the vertical axis indicates the average request completion time of the requests in that time frame. We chose a logarithmic scale for the vertical axis to be able to plot the request completion times of the different request types on one graph.

Activity Durations To visualize activity durations, we map each activity to a start time, which is the timestamp of the start event. Then we create time intervals with a length of 2.5 seconds and represent the average activity durations with markers similar to the request completion times. The visualization has a linear scale on the vertical axis between 0 and 2 milliseconds and is logarithmic for higher values.

Resource Durations Resource durations are visualized similar to activity durations.

Host Queue Sizes To visualize Host Queue Sizes for multiple hosts at once, we use a stacked bar chart. We create time intervals with a length of 1 second and create a bar in the chart to indicate the maximum amount of tasks in the queue of a given host in the time interval.

Host CPU Usage To visualize Host Queue Sizes for one host, we use a stacked bar chart. We create time intervals with a length of 1 second and create a bar in the chart to indicate the CPU usage a given component on the host in the time interval.

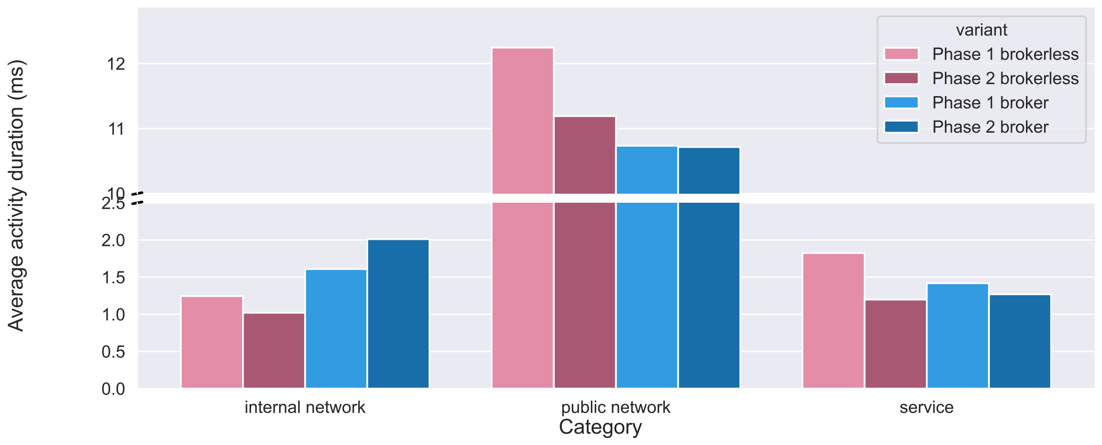


Figure 8.4.: Average activity durations of the reference applications.

8.4.2. A Brief Overview

The request completion times are determined by the individual activity durations of the request. To allow a manageable comparison of the results, we categorize the activities into three categories:

- **Internal network** The activities of the message transfers between the services within the microservice application. For the broker applications, the time needed by the message broker to forward the message is included in this category.
- **Public network** The activities of message transfers between client and gateway.
- **Service** All task activities, i.e. the durations it takes the services to process messages.

Within a category we distinguish between activities started during **phase 1** (before 60 seconds scenario time) or during **phase 2** (after 60 seconds scenario time). This allows a comparison between the influence that a doubling of the number of clients has on the performance.

First, we compare the average durations of the activities in the categories for the reference systems, shown in figure 8.4. During the entire scenario time, the internal network activities for the broker variant are slower than for the brokerless variant. For the internal network activities of the broker variant, we observe a deterioration in performance in phase 2. Whereas the same durations for the brokerless variant decrease slightly, although more users are connected. However, we also observe that the public network activities account for the majority of the performance experienced by the user for all variants. The slightly higher internal network durations of the broker variant are therefore only really noticeable for requests with very long messaging chains. Moreover, we observe that for broker and brokerless variants the times of service activities decrease in phase 2, although the number of clients and therefore the load on the hosts is higher.

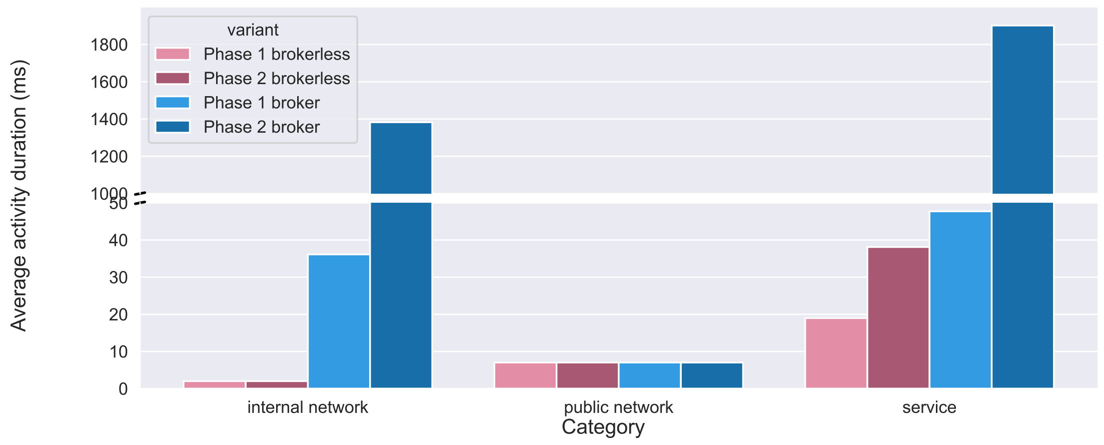


Figure 8.5.: Average activity durations of the simulations.

However, the average durations of the simulation's activities, as shown in figure 8.5, paint a different picture. Although the internal network activities for the broker variant are also higher than for the brokerless variant, the extent of the increase is many times higher than measured in the reference applications. In addition, the public network activities play a much smaller role in the performance of the requests as perceived by the user. In the simulation, the service durations and the performance of the message broker are much more significant for the perceived performance than in reality.

8.4.3. Simulation Results

To find an explanation for the differences in performance between reference applications and simulations, we examine the event logs of the simulations in more detail in this section. To reduce the complexity of the visualizations and to make the simulation results easier comparable to the reference application results, we adapt the event log in the following way: We remove the activities for a task being queued on a host and added the duration of the queue activities to the duration of the task activities. Since the reference applications do not record separate queue activities, this makes the activity durations more comparable.

Brokerless messaging simulation

This section presents the results of the simulation of the FruitCollector Microservice Application using brokerless messaging.

Running the simulation took 30.9s on the same machine as the player simulator runs on. The count column shows the number of requests of a request type that were recorded during the scenario time. For example, we see that a total of 4550 *inventoryState* requests were simulated. Since *inventoryState* is called 1 time per second, we expect at 120 seconds scenario duration, for phase 1 (from 0 to 60 seconds) $25 \cdot 60$ requests and for

Request Type	**Count**	**Request Completion Time (ms)**			
		Mean	Min	Max	Std
register	50	243.1	125	389	95.3
login	50	234.1	119	377	77.0
playerView	36050	120.4	25	1706	213.5
move	4550	214.9	77	1341	188.7
collectItem	4550	214.2	77	1369	187.8
inventoryState	4550	205.1	78	1278	146.6
sendChat	75	249.2	104	388	88.6
retrieveChat	4550	196.8	75	1271	142.3
retrieveQuests	950	402.0	159	1626	321.0
finishQuest	75	501.8	249	637	175.2
uploadReplay	50	2323.1	1254	2854	457.8

Table 8.2.: Brokerless Messaging Simulation: Statistics

phase 2 (from 60 to 120 seconds) $50 \cdot 60$ requests, so a total of 4500 requests. The simulation, however, simulates including the last millisecond of the scenario time, which is why another 50 requests are started in the last millisecond of the scenario time. The other columns show the mean, min, max and standard deviation of the measured request completion times (RCTs). We observe that *uploadReplay* has by far the highest average RCT. We also see that for some requests the maximum measured RCT is very different from the average RCT and the standard deviation is high, suggesting that there are some RCT peaks during the scenario.

Figure 8.6 shows the request completion times throughout the simulation. The data shows, that the average request completion times scale approximately proportional to the number of connected clients. Furthermore, we make three main observations regarding the performance of the system:

Fluctuations Throughout the scenario time, we can observe fluctuations in the request completion times. Since these fluctuations coincide with the *retrieveQuests* requests every 5 seconds, they are likely the cause for the fluctuations.

Connection Peaks At the beginning of phase 1 and phase 2 respectively, 25 clients call *register* and *login*. We observe that at these times the request completion times increase slightly for around 10 seconds after which we observe another peak.

Replay Peaks At 10 and 70 seconds scenario time, 25 clients each call *uploadReplay*. We observe that shortly after the clients start uploading the replays, the request completion times of the other requests increase sharply. However, the request completion times recover quickly here as well. These calls seem to have a drastic influence on the request completion times of the other request types. The *move* and *playerView* request completion times seem to be particularly affected by this. This suggests that the world service is particularly affected by replay peaks.

We use the *playerView* requests to further investigate the behavior of the system. This request type is particularly well suited for this purpose, since we have a high

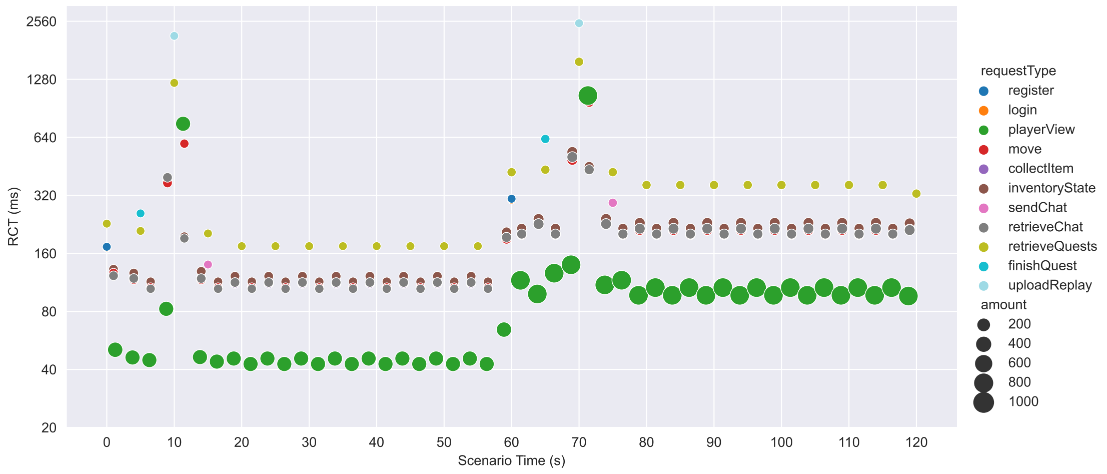


Figure 8.6.: Brokerless Messaging Simulation: RCTs

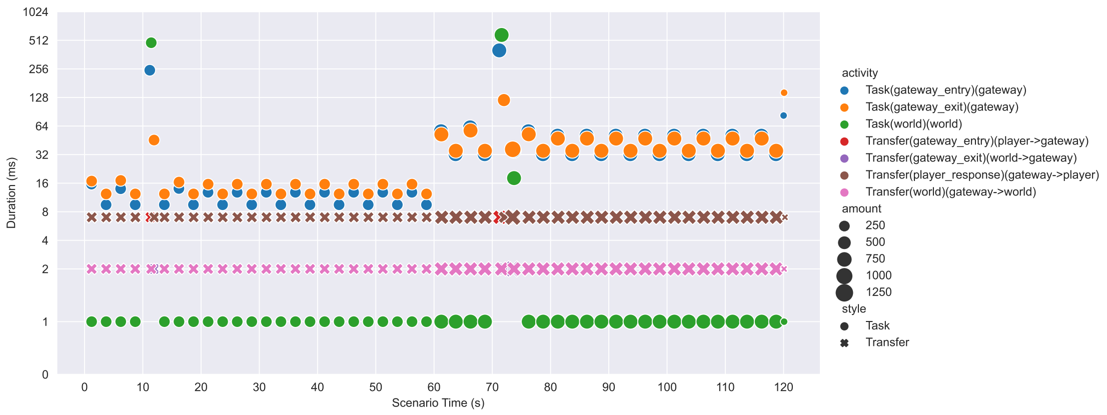


Figure 8.7.: Brokerless Messaging Simulation: Activity Durations playerView

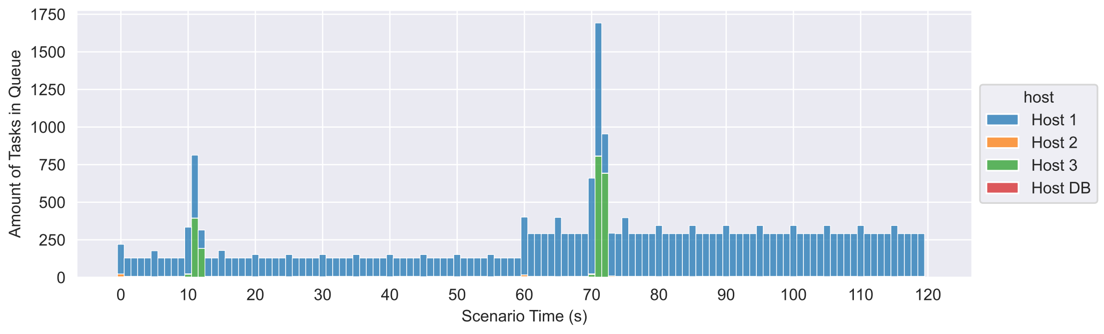


Figure 8.8.: Brokerless Messaging Simulation: Host Queue Sizes

coverage of *playerView* requests during the entire scenario time. Also, the replay peaks are particularly devastating for the *playerView* RCTs, which requires closer observation. We plot the activity durations for *playerView* in figure 8.7. The transfer times between the different components are constant, since the simulation assumes that the round-trip time between two hosts is constant.

We see that the fluctuations are not caused by the task durations on the world service, but stem from the fluctuating durations of the gateway tasks. This supports the assumption that *retrieveQuests* is responsible for the fluctuations, since gateway and quest service are both running on host 1.

We observe that the first connection peak is triggered by high task activity durations in the request stage gateway_entry. Since all clients send their *register* and *login* requests simultaneously, there is a short congestion for all requests when entering the gateway. However, since the gateway processes the messages one after the other, this congestion dissipates and there is no more congestion at the gateway exit. We also observe that the second connection peak is caused by higher task durations at both gateway entry and gateway exit. It is possible that congestion caused by *register* and *login* requests in the gateway host's queue may not resolve as quickly due to the higher overall load, causing delays at entry and exit.

The activity durations reveal that high activity durations on both the world and gateway services are responsible for the replay peaks. Since the replay and world service both run on host 3, it makes sense that both gateway and world service are affected by the replay peaks. This also explains why *playerView* and *move* are especially affected by the replay peaks.

To further investigate the behavior of the system we plot the hosts queue sizes in figure 8.8. We see that host 2 and host db, hardly have any tasks waiting in their queues. This means that these two hosts provide mostly constant activity durations for tasks processed on them and are not responsible for the peaks or fluctuations.

At 10 and 70 seconds we observe a drastic increase in the host queue sizes for host 1 and host 3. This confirms that a slow processing of tasks on both host 1 and host 3 is indeed responsible for the replay peaks. Since the world service runs on host 3, this confirms that the workload on host 3 is the reason why *playerView* and *move* are

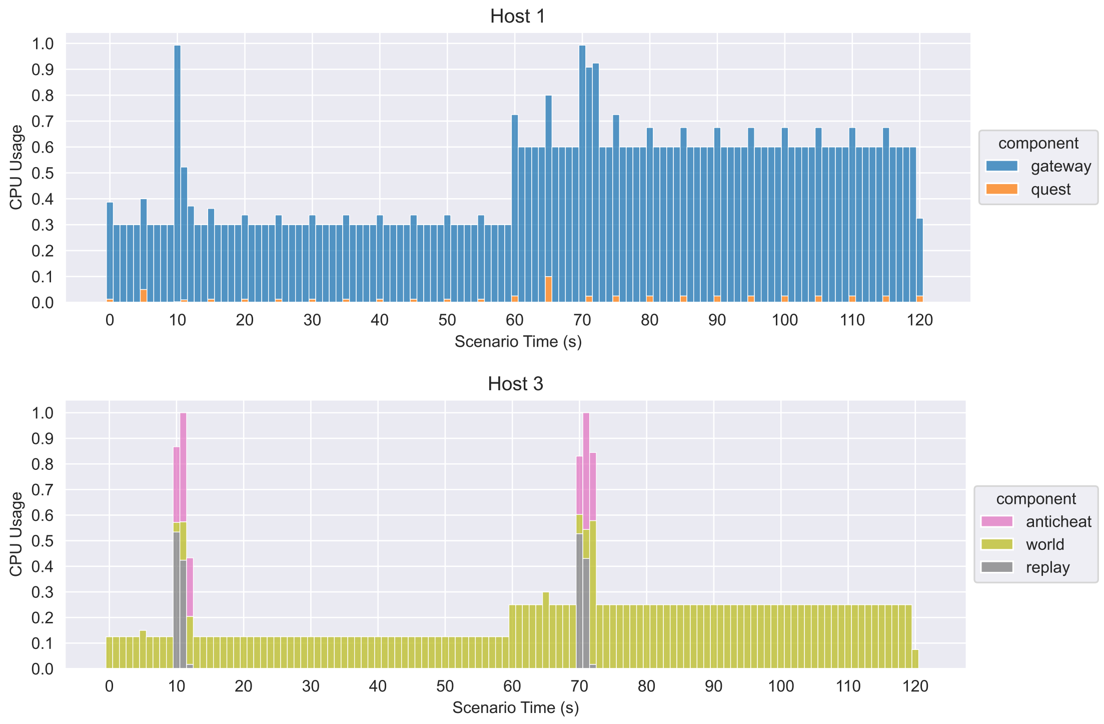


Figure 8.9.: Brokerless Messaging Simulation: Host 1 and 3 CPU Usage

particularly affected.

To further investigate the reasons for the observations, we plot the CPU usage for Host 1 and Host 3 in figure 8.9. The CPU usage for host 1 confirms that the *retrieveQuests* requests are indeed the reason for the observed fluctuations. Every 5 seconds, the quest service and the gateway use a little more CPU due to the *retrieveQuests* requests. Since the gateway is also hosted on Host 1, and all inbound and outbound requests must be processed by the gateway, this explains why the fluctuations affect all request types.

Also, the CPU usage for host 1 shows us that the CPU is fully utilized at 10 and 70 seconds, probably caused by the *uploadReplay* requests. This explains why the *uploadReplay* requests affect the request completion times of the whole system. The CPU usage of Host 3 shows why the request types of the world services are particularly affected by the replay peaks. The anticheat service and the replay service use almost all the available CPU time on host 3, which also hosts the world service, for two seconds.

Broker messaging simulation

This section presents the results of the simulation of the FruitCollector Microservice Application using broker based messaging.

Running the simulation took 62.8s on the same machine as the player simulator runs on. We see that *finishQuest* is the request type with the highest average RCT. Moreover,

Request Type	**Count**	**Request Completion Time (ms)**			
		Mean	Min	Max	Std
register	50	593.9	346	927	190.3
login	50	587.7	342	963	189.3
playerView	36050	11606.3	47	25382	9920.1
move	4550	11628.2	225	25309	9857.2
collectItem	4550	11625.8	220	25310	9858.5
inventoryState	4550	11637.6	232	25312	9856.7
sendChat	75	9396.1	1558	13471	5542.1
retrieveChat	4550	11644.4	226	25311	9863.0
retrieveQuests	950	14135.3	347	30637	11788.9
finishQuest	75	12849.3	736	19800	8672.2
uploadReplay	50	11093.3	4837	18087	6098.3

Table 8.3.: Broker Messaging Simulation: Statistics

the average RCTs of the request types other than *register* and *login* are exceptionally high. Since the min and max values of the RCTs of the respective request types differ so much, this could indicate an overload of the system from phase 2 onwards, which affects all request types equally.

To check whether the system is actually overloaded in phase 2, we plot the RCTs over the course of the scenario in figure 8.10. We observe that request completion times increase dramatically in phase 2 and that all request types are affected. We can also observe that the replay peaks at 10 and 70 seconds have a strong influence on the request completion times of the subsequent requests. In phase 1, the request completion times can still recover from the replay peaks, while in phase 2 the request completion times remain very high until the scenario is finished. The presence of the connection peak at 0 seconds is noticeable, in the time directly after that the RCTs are slightly higher than between 20 and 60 seconds. The connection peak at 60 seconds also seems to have an impact. But it is difficult to detect if the high RCTs are a result of the *register* and *login* requests or if the system is generally not able to handle 50 clients. We observe that the broker application in phase 1 is also affected by fluctuations due to the recurring *retrieveQuests* requests. In phase 2, we cannot see any fluctuations because the system is generally overloaded during this period.

For further analysis we plot the activity durations for *playerView* in figure 8.11. We see that even 20 seconds after the last client disconnects and no new requests arrive, the system is still busy processing *playerView* requests. Similar to the brokerless system, the replay peaks cause the world service to shortly take longer to process the tasks in the *playerView* request. However, the main cause of the high RCTs in phase 2 are the increase task durations on the gateway and broker. We can also observe that the tasks on the broker and gateway are responsible for the fluctuation of the request completion times between 20 and 60 seconds scenario time.

To further explore the reasons for the increased request completion times, we plot the

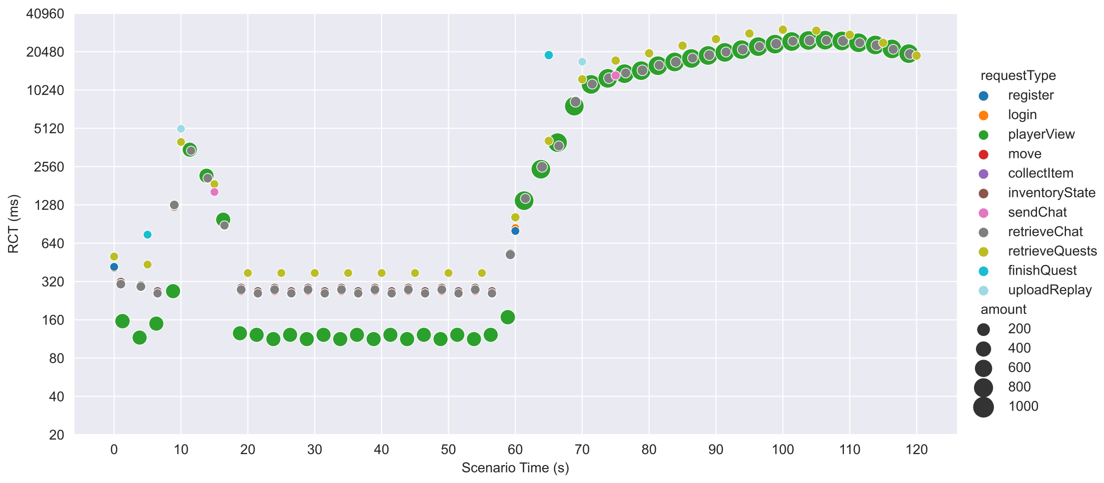


Figure 8.10.: Broker Messaging Simulation: RCTs

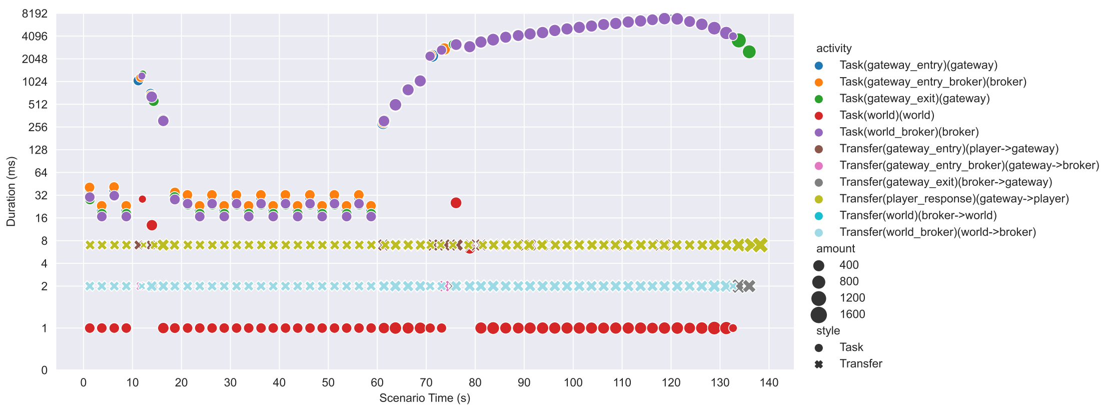


Figure 8.11.: Broker Messaging Simulation: Activity Durations playerView

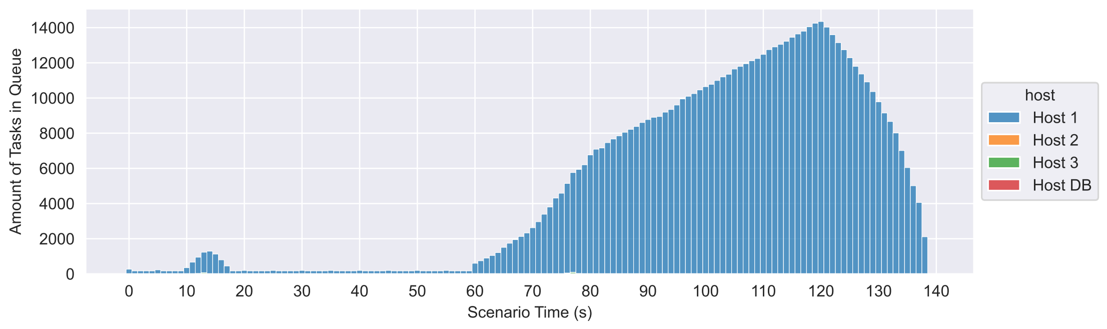


Figure 8.12.: Broker Messaging Simulation: Host Queue Sizes

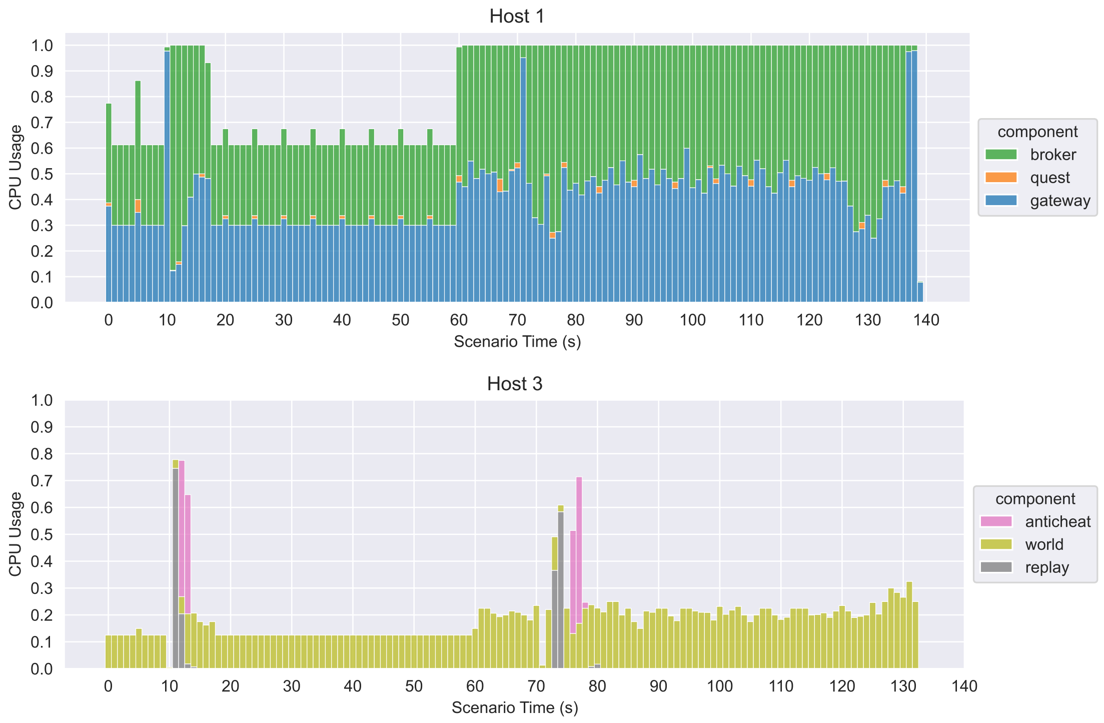


Figure 8.13.: Broker Messaging Simulation: Host 1 and 3 CPU Usage

host queue sizes in figure 8.12. It can be seen that the queue for host 1 continues to grow as soon as phase 2 starts. Tasks enter the queue faster than host 1 can process them, so the queue continues to grow. The other hosts do not experience such a high growth of queue sizes. This is probably due to the fact that the gateway and broker are hosted on host 1, which means that all other hosts can only receive their messages as quickly as host 1 can process and forward them. The gateway and broker clearly represent a bottleneck for the entire system.

To investigate the causes for the growth of the host queue sizes, we plot the CPU usages of the hosts that are relevant for the processing of *playerView* in figure 8.13. We observe that broker and gateway take up almost the entire CPU time on host 1. The short increase of the first replay peak at 10 seconds can be processed relatively quickly and the CPU usage returns to its normal value. However, as soon as a total of 50 clients send requests to the system at 60 seconds, the CPU is fully utilized throughout. For host 3, we can see that at 10 and 70 seconds, the world service is barely processing any tasks. The reason for this is probably that the gateway and broker are clogged with *uploadReplay* requests and therefore no messages arrive for the world service for a short time. We can also observe that at the second replay peak, a second elapses between the times when the replay service processes tasks and the anticheat service processes tasks. The reason for this is that the replay service routes messages through the broker to send them to the anticheat service. Since the broker is overloaded, this forwarding takes some

Request Type	Count	Request Completion Time (ms)			
		Mean	Min	Max	Std
register	49	1191.4	192	2006	515.7
login	49	1610.8	1272	2062	230.4
playerView	30875	25.8	19	2719	57.3
move	3990	26.3	19	2723	76.1
collectItem	4027	26.4	19	2844	71.5
inventoryState	4061	29.3	22	2395	57.6
sendChat	73	24.8	21	36	2.8
retrieveChat	4115	26.1	20	2618	55.0
retrieveQuests	846	28.2	20	1822	62.5
finishQuest	19	83.4	55	146	19.6
uploadReplay	38	511.3	165	776	180.0

Table 8.4.: Brokerless Messaging Reference Application: Statistics

time.

8.4.4. Reference Applications Results

In this section we examine the event logs of the reference applications to find out which factors influence the performance of microservice applications.

Brokerless messaging reference application

This section presents the results of the reference application of FruitCollector using brokerless messaging.

Table 8.4 shows the request completion time statistics for each request type. One main problem with the player simulator is immediately apparent. The player simulator is not able to keep the modeled frequencies for the real sent requests. Part of the problem could be caused by the long request completion times for *register* and *login* requests, since a client only receives its authentication token after login to send further requests. However, the player simulator seems to be overloaded in general, as the 50th client does not even send a *register* request. Moreover, out of 50 expected replay uploads, only 38 are actually made.

Figure 8.14 shows the request completion times over the scenario time. We can observe that the *playerView* requests start after about 2.5 seconds, because the players have received the required authentication tokens only then. We can observe that the first 25 clients are still able to upload their replays at about 10 seconds scenario time. For the 25 clients that connect later, however, the timing of the replay uploads is severely delayed. We also observe that the amount of connected clients seems to have little impact on the request completion times. After 80 seconds of scenario time, the request completion times even stabilize at a lower level than before. However, we can observe that shortly

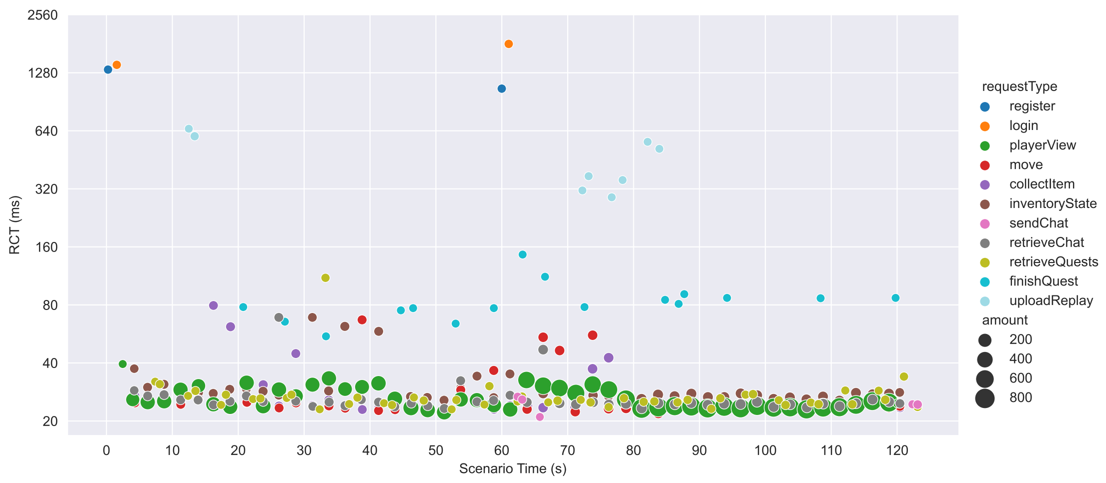


Figure 8.14.: Brokerless Messaging Reference Application: RCTs

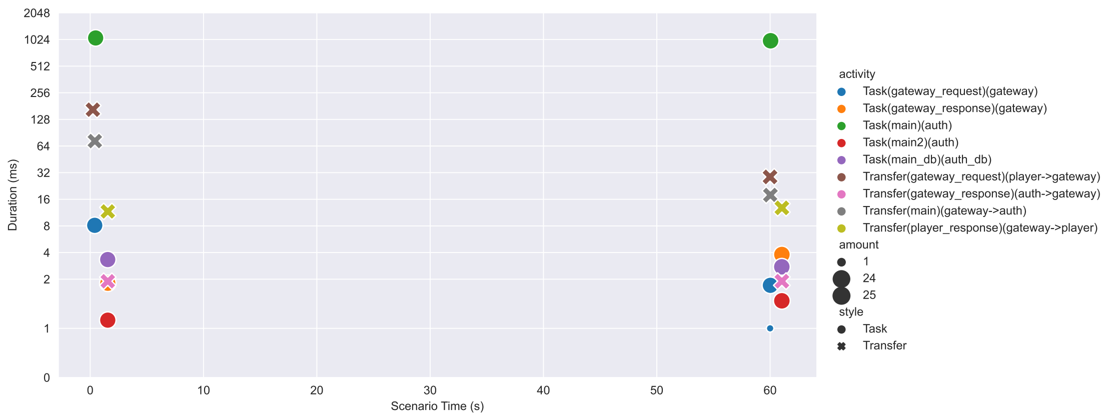


Figure 8.15.: Brokerless Messaging Reference Application: Activity Durations register

after the 25 new clients connect, the RCTs for all requests increase shortly. The RCTs of *finishQuest* seem to be particularly affected. Moreover, we can observe that the RCTs are affected by more or less strong fluctuations during the whole scenario time. Some requests even have strong outliers for the RCT, for example *retrieveChat* at about 25 seconds, *inventoryState* between 30 and 42 seconds, and *collectItem* between 15 and 30 seconds.

First, we examine why the *register* and *login* requests have such high request completion times. Since both take a similar amount of time and have a similar message flow, we do this exemplarily by plotting the *register* activity durations in figure 8.15. We see that the task on the auth service is responsible for most of the request completion time. This is likely due to the fact that this task hashes passwords using the Blowfish encryption algorithm (BCrypt), which is an algorithm purposely designed to be expensive to protect

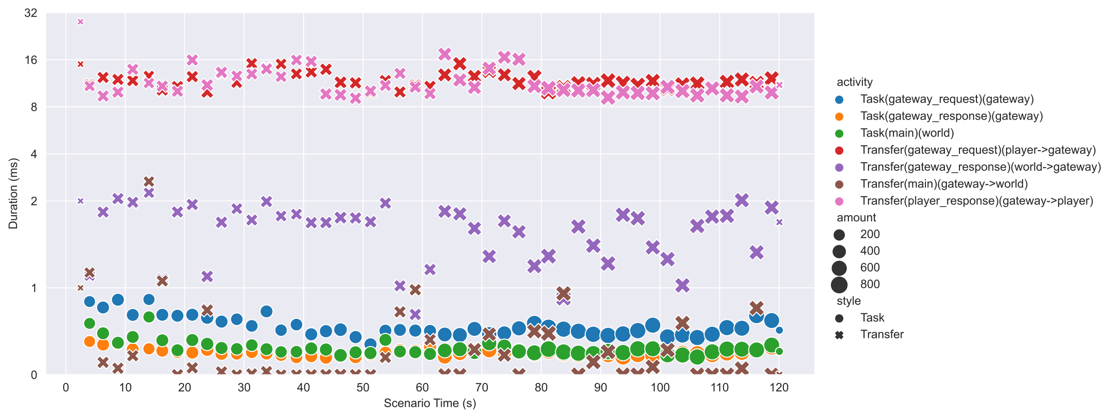


Figure 8.16.: Brokerless Messaging Reference Application: Activity Durations playerView

passwords[SK15]. Since passwords have to be hashed for both registration and login, this also explains the high request completion times for *login*. Furthermore, this could explain the high standard deviation for *register* and *login* RCTs, since the expensive hashing algorithms require a lot of CPU time, the earlier requests block the later ones, which in turn leads to the later requests having a higher completion time. We can also see that the message transfer from the player to the gateway and from the gateway to the auth service is higher for the first 25 clients that sign up. A possible explanation is that despite the warm-up run, there are still some TCP congestion control algorithm effects that cause the initial HTTP message transfers to be slower than later.

To investigate the reasons for the increase in request completion times between 60 and 80 seconds of scenario time, we plot the activity durations of the *playerView* in figure 8.16. We see that the gateway and world service tasks take less than a millisecond on average. This is because we only record events in the millisecond range, so if a task is processed too quickly, the associated activity duration may be recorded as 0 milliseconds. We also see that message transfers are the main factor behind the measured RCTs. The message transfers between the gateway and the world are also hardly relevant. The main part comes from the message transfers between the player simulator and the gateway. These message transfers are also the reason for the increase in the measured request completion times between 60 and 80 seconds of scenario time. The reasons for this increase are difficult to pinpoint, as there are many factors that can affect message transfer on the Internet. However, we do know that excessive host load due to client registration and login at 60 seconds is unlikely to be the cause of this increase.

To explore the reasons for the RCT outliers, we plot the resource durations of *collectItem* in figure 8.17 as an example. We can see that also for the *collectItem* requests, the message transfers account for the main part of the measured RCTs. The message transfers between player and gateway are also the reason for the outliers. However, we can also observe that the processing time of the tasks on the inventory service increases

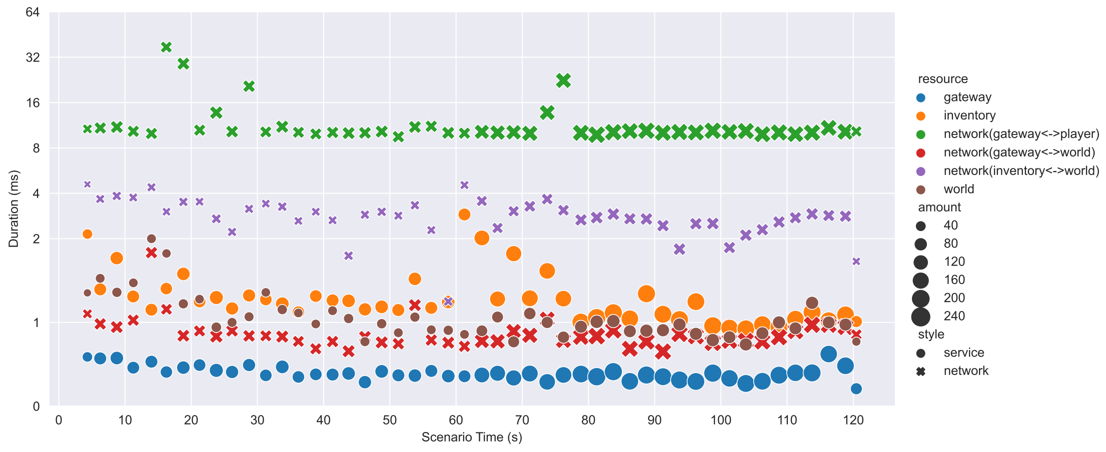


Figure 8.17.: Brokerless Messaging Reference Application: Activity Durations collectItem

slightly at 60 seconds and then returns to normal. Since the inventory service runs on the same host as the auth service, this could be an indication that the registration and login requests are affecting the processing times of the other services on the same host.

To check this we plot the resource durations for *finishQuest* in figure 8.18. We observe that the processing times of the tasks on the inventory service actually increase significantly shortly after the new clients connect at 60 seconds scenario time. This confirms the hypothesis that the increased load on the host due to *register* and *login* requests has an impact on the other requests.

Broker messaging reference application

This section presents the results of the reference application of FruitCollector using broker based messaging.

Table 8.5 shows the request completion time statistics for each request type. Again, the data shows that the player simulator is not able to send all requests as planned. We also observe that *register* and *login* are again the request types with the highest request completion time. We also observe a high standard deviation for *register* and *login* again. The reason for this is again the expensive hashing algorithm used for the passwords.

Figure 8.19 shows the request completion times over the scenario time. We observe that the average RCTs for phase 2 are higher than for phase 1. For example, the average RCT of *playerView* increases from 24.5ms for phase 1 to 25.9ms for phase 2. The increase is particularly significant for *finishQuest*, where the average RCT increases from 52.3ms to 88.2ms. Since we did not find an increase in RCTs for phase 2 for the brokerless messaging application, we assume that the message broker is the cause of these increases. Since the message broker acts as an intermediary for each message transfer, it is to be expected that request types with long message chains, such as *finishQuest*, will particularly suffer from a slowdown of the broker. However, the RCTs for *finishQuest*

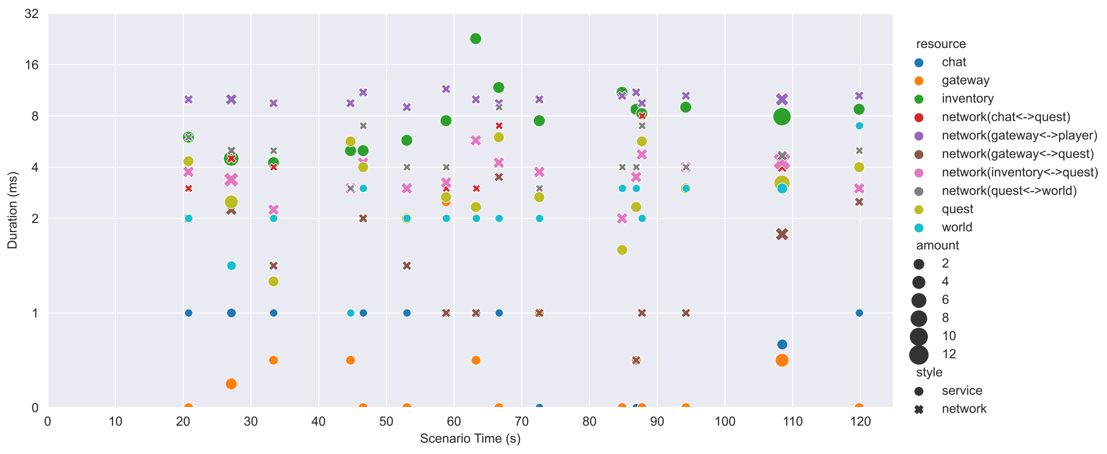


Figure 8.18.: Brokerless Messaging Reference Application: finishQuest

Request Type	**Count**	**Request Completion Time (ms)**			
		Mean	Min	Max	Std
register	50	1793.4	751	2862	614.3
login	50	822.8	149	1753	449.6
playerView	27743	25.4	18	1045	7.5
move	3632	24.2	19	52	4.0
collectItem	3110	25.8	20	75	4.9
inventoryState	3592	27.6	21	1044	17.8
sendChat	51	26.4	22	56	5.5
retrieveChat	3573	24.6	20	76	4.2
retrieveQuests	720	24.0	19	48	4.1
finishQuest	18	72.2	44	212	44.0
uploadReplay	45	457.4	241	815	159.1

Table 8.5.: Broker Messaging Reference Application: Statistics

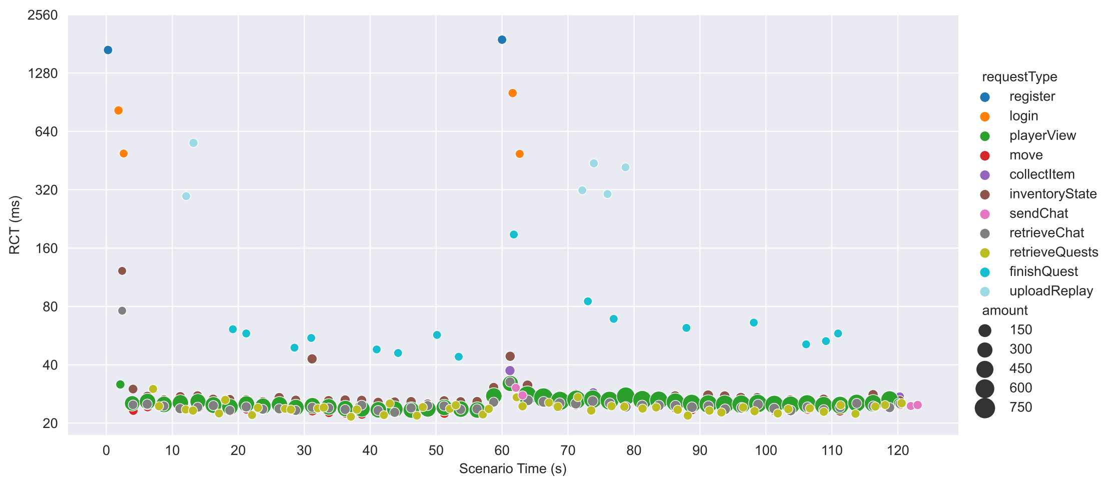


Figure 8.19.: Broker Messaging Reference Application: RCTs

requests from phase 2 only seem significantly higher between 60 and 80 seconds. These could be strongly influenced by the *register*, *login*, and *uploadReplay* requests from phase 2. To allow a better comparability between the *finishQuest* requests from phase 1 and phase 2, we only take *finishQuest* requests that start after the last replay upload is finished. After this adjustment, the average RCT for *finishQuest* is 52.3ms for phase 1 and 58.7ms for phase 2.

To verify that the message broker is indeed responsible for the increased RCTs in phase 2, we plot the *playerView* resource durations in figure 8.20. We observe that message transfers between gateway and world service indeed take longer in phase 2. This shows that the message broker is measurably slower for messages from 50 clients than for messages from 25 clients.

8.5. Discussion of Results

In this section, we discuss the reasons for the differences between the performance prediction of the simulation and the actual measured performance of the reference applications.

Workload and request completion times

The simulations show that the number of clients and thus the load on the system has a clear influence on the request completion times. The simulation results of the brokerless application show that the request completion times scale proportionally to the number of connected clients. For the broker application simulation, the RCTs in phase 2 increase even more and show that the system is completely overloaded. For the brokerless reference application, the data shows that the request completion times actually decrease in phase 2 even though the number of clients doubles. For the broker reference application, we can observe an increase in request completion times in phase 2, which is due to an

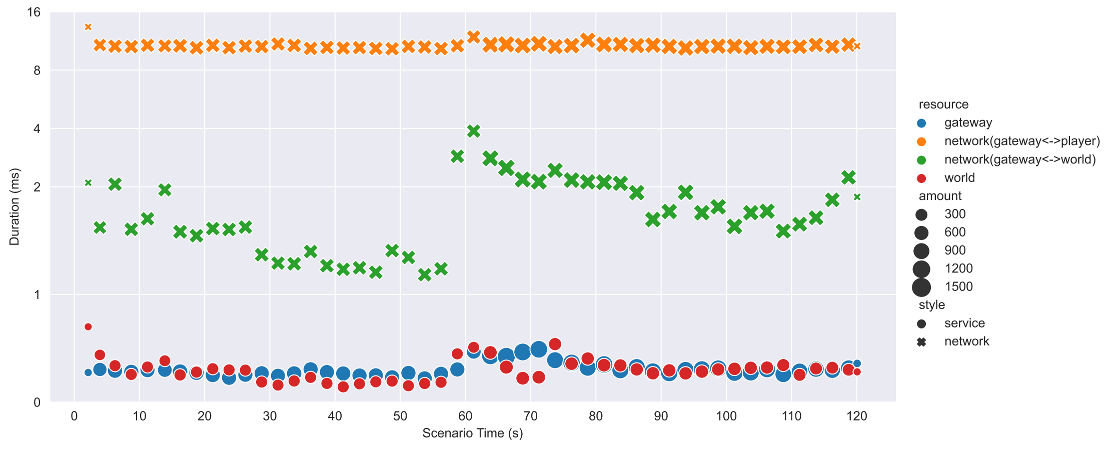


Figure 8.20.: Broker Messaging Reference Application: Resource Durations playerView

increased workload of the broker. However, this increase is of smaller magnitude than in the corresponding simulation. In other words, the simulation overestimates the impact that more workload has on a microservice application.

One reason that simulation overestimates the increase in RCTs by more clients is due to the simultaneity of the requests. Since for the simulation all clients have the same behavior, all clients send their messages at the same time. Since the message transfer durations for the same request types are also always the same, the messages also arrive at the gateway at the same time. This quickly causes congestion in the gateway in the simulation, which leads to an increase in the request completion times, even though the CPU is not fully utilized. For the reference applications, the various requests arrive more evenly distributed over time, which means that such a congestion cannot occur as quickly.

Another reason is the processing of requests with large messages in the simulation. Large messages lead to tasks with long processing times, e.g. the gateway needs about $750ms$ to process a $2MB$ message in the simulation. If now the service is scheduled to process two such tasks on a CPU with two cores, the CPU is completely blocked for $750ms$. In a worst case scenario, a short task with only $1ms$ of processing time that is queued in the same millisecond has to wait for $750ms$. This would lead to a total task duration of $2251ms$ for these three tasks, whereas the task duration would only be $1501ms$ if the short task is queued first. Small messages that also want to be processed by the CPU in this time must therefore wait up to 750ms in the queue, although their processing would perhaps only require 1ms. Real CPUs are however able to interrupt the execution of a process, in order to grant other processes processing time. At this point, the CPU model of the simulation falls short.

Another reason could be that the hosts used for the reference application, so-called droplets, are only virtual machines that use vCPUs. With the droplets, the physical hardware and thus both the network access and the computing power are shared, so it is not guaranteed that the same performance is available for the hosts at all times.

Another reason is the overload of the player simulator, which is not able to send the same amount of requests as defined in the simulation models. This problem is even more severe in phase 2, whereby twice the number of clients in the reference application does not cause twice the workload. For example, in phase 1 we expect 12000 *playerView* requests and in phase 2 24050 *playerView* requests. However, for the brokerless reference application, the player simulator sends only 10935 in phase 1 and only 19940 *playerView* requests in phase 2, so still $\approx 91\%$ for phase 1 but only $\approx 83\%$ in phase 2. The reasons for this could be the hardware of the player simulator machine. However, none of the CPU cores of the running machine reached 100% utilization, so we assume that the CPU of the player simulator machine is not a limiting factor. The player simulator could also be limited by network limitations, as it has to send a lot of HTTP messages at the same time. Due to time constraints, we were not able to further investigate the poor performance of the player simulator.

Expensive requests

For the simulation, the expensive requests, i.e. the requests with high request completion times, are those with long messaging chains and high message sizes. In the simulation, *uploadReplay* is the most expensive request in the time interval in which it is sent. For the reference applications, however, *uploadReplay* is only the third most expensive request. *Register* and *login* are even more expensive, although both have a small messaging chain and small message sizes.

The simulation uses only the message size as parameter to calculate the processing times. The simulation provides possibilities to overwrite the base processing times per service or per request stage. However, during modeling we only have a description of the functionality of the different tasks on the services. We did not find a way to infer a reasonable base processing time from these descriptions. Even if we know that an expensive hashing algorithm is needed to *register* and *login*, too many questions remain. For example, we do not know how long an expensive hashing algorithm takes on a 1 GHz CPU, which exact hashing algorithm is used and with which parameters. Without reference values it is difficult to make an estimation.

Short duration activities

In the reference applications there are many average activity durations of less than 1 millisecond. For example, the average duration of tasks on the world service for *playerView* is $0.3ms$. However, the reference applications only work with millisecond precision for the timestamps of the events. So we can assume that the actual task durations are between 0ms and 1ms.

However, the simulation works with only one millisecond accuracy. Tasks with such short durations cannot be simulated at all, but are always rounded to the nearest millisecond. For the transfer activities, this leads to the average request completion times deviating from the actual length by a constant factor. For the task durations, however, this can have far-reaching consequences for the simulation results. The task processing durations t_p are calculated with the base processing time t_b and the CPU clock speed in GHz s with the formula $t_p = \frac{t_b}{s}$. If the CPU clock speed is 2GHz this results in $t_p = 0.5$ and is rounded up to 1ms by the simulation, which means one core can process 1000 tasks in one second. If the CPU now has a clock speed of 2.1GHz, we expect that in reality the CPU can process a few more tasks in the same time. However, due to the rounding to milliseconds, the simulation calculates a task processing time of $0ms$. This means that the CPU can process an infinite number of tasks in one second. This in turn can make the difference between a completely overloaded system and a stable system in the simulation. The millisecond accuracy of the simulation is therefore not sufficient to accurately simulate microservice applications.

Long distance message transfers

The simulation underestimates long distance message transfer durations, i.e. message transfers from player to gateway and vice versa. All other message transfers take place within the same data center. As an example we consider the message transfer times from player to gateway for brokerless messaging. To compare the message transfer durations we take the average durations of these message transfers for *playerView* and *uploadReplay*. *PlayerView* is an example for a message transfer with a small payload, while *uploadReplay* is an example for a transfer with a large payload. For the simulation the message transfer time is constant and is $7ms$ for *playerView* and $23ms$ for *uploadReplay*. For the reference application the average time message transfer time from player to gateway is $9ms$ for *playerView* and $133ms$ for *uploadReplay*.

One reason for the deviations between the real message transfer times and the simulated message transfer times could be the underlying formula that the simulation uses. The simulation uses the formula $t_m = d_{trans} + d_{prop}$ to calculate the message transfer durations, as explained in section 5.2.4. The actual measured RTT between the player simulator machine and the reference application data center is $17ms$[10]. Applying the formula $d_{prop} = \frac{RTT}{2}$, gives us the propagation delay for the reference applications $d_{prop} = 8.5ms$. Since the message size for *playerView* is very small and the upload bandwidth of the player simulator machine is very high, d_{trans} can be disregarded. Thus, we assume $t_m = d_{prop}$ for *playerView*. The average measured t_m is $9ms$, while the formula using the measured RTT gives us $8.5ms$ Since the reference applications only measure in the millisecond range, the values match well. So the formula seems to be accurate for *playerView* and not the reason for the deviations.

Another reason for the deviations could be the predicted RTTs that the simulation uses. The multi layer perceptron predicts an RTT of $13.54ms$ between a host in Aachen and a host in London, which leads to $d_{prop} = 6.77ms$. This in turn results in the $7ms$ message transfer time for *playerView* from the simulation. So we see that the discrepancy of the actual RTT and the predicted RTT is responsible for the $2ms$ difference in the average message transfer times for *playerView*.

[10]Measured with an ICMP ping to `speedtest-lon1.digitalocean.com`.

However, the question remains how the discrepancy between the message transfer durations at *uploadReplay* comes about. d_{prop} is the same for all message transfers and explains only $2ms$ difference. So we are still missing an explanation for the remaining $110ms$ between the $23ms$ duration in the simulation and $133ms$ duration in the reference application. During simulation configuration, we assumed a replay size of $2MB$, which with the upload bandwidth of the client properties leads to $d_{trans} = 16ms$. However, the actual size of the replays in the reference application is about $10MB$, which leads to the simulation calculating d_{trans} as 5 times smaller than it should be. So if the simulation was configured with the correct message size, it would give a message transfer duration of $16ms \cdot 5 + 7ms = 87ms$, which is already much closer to the measured average duration of $133ms$.

However, even with the adjustments we still have a discrepancy of $46ms$. This could be due to the fact that the simulation assumes that all clients independently use their own bandwidth. However, the player simulator sends the requests of up to 50 clients simultaneously on the same machine. This results in all requests having to share the available bandwidth, leading to a higher d_{trans}. In addition, the available bandwidth of a machine may be subject to fluctuations[Eke+06]. Both these reasons suggest, that in reality the replays are uploaded with a lower bandwidth than expected. The measured message transfer durations of the reference application, have a standard deviation of $55.9ms$ with a minimum value of $86ms$ and a maximum value of $436ms$. Which supports the hypothesis that the replays are uploaded with different amounts of available bandwidth.

8.6. Threats to Validity

In this section we discuss the threats to validity of this evaluation. For the evaluation, we compared the results of the simulation with the results of the simulation. However, the collection of the reference application results is affected by inaccuracies, which in turn may affect the inferences drawn, which we discuss in section 8.6.1. For the evaluation, we implemented the FruitCollector game as a microservices application as an example. However, microservices applications are diverse, which might make it difficult to generalize the results of the evaluation to other microservices applications. We discuss possible difficulties of generalizability in section 8.6.2.

8.6.1. Internal Validity

The events of the reference applications are captured on four different hosts. Although we synchronize the times of the different hosts using the network time protocol, perfect clock synchronization is not given. This can lead to the durations of the transfer activities not being recorded accurately. For the brokerless reference application we have 196785 recorded transfer activities of which 7038 have a negative duration with an average of $-1ms$. For the broker reference application we have 173680 recorded transfer activities of which 825 have a negative duration with an average of $-1ms$. The existence of negative

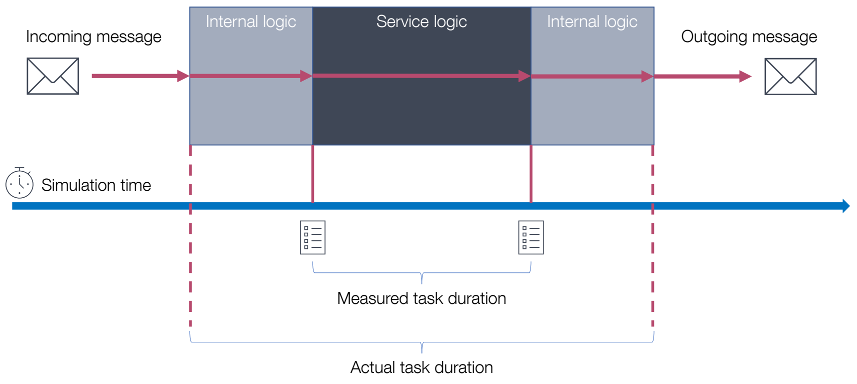


Figure 8.21.: Internal logic as a possible cause for a discrepancy between actual and measured task durations.

transfer durations indicates that all transfer durations can be affected by an inaccuracy.

The simulation assumes that all hosts are independent machines, where the performance is only influenced by the services running on them. However, for cloud deployments of microservices, including the evaluation reference application, virtual machines are often used to host the services. Therefore, the assumption of the simulation that only the modeled services affect the performance does not hold. We can not know how many other virtual machines are running on the underlying hardware during our measurements, which could have an impact on the performance of the virtual machines.

Furthermore, the simulation assumes that a message transfer between two hosts is always a transfer between two different physical machines. This assumption is also not necessarily given when using virtual machines. However, DigitalOcean guarantees that the virtual machines of a Kubernetes cluster run on different physical hosts. Although we could not verify this statement, we assume that the assumption of the simulation about the message transfers between hosts is correct.

To capture the events for the reference applications we use hooks in the Spring Boot implentations of the services. However, incoming messages must first pass through some internal logic of the operating system and spring boot before we can capture an event. This causes transfer finish and task start events to be logged slightly later than they actually happen. Outgoing messages also have to go through some internal logic after logging the task finish and transfer start events. This leads to the fact that the measured task durations and therefore also the measured transfer durations can deviate from reality, as illustrated in figure 8.21.

8.6.2. External Validity

The measurements of the reference applications provide insights into the relationship of the simulation results to a microservices application with the technologies of the reference applications. However, one strength of microservices is that the languages and frameworks with which the individual services are implemented are not predefined and can also differ within a microservices application[BCK03]. In the reference applications, however, all microservices are implemented with Java and Spring Boot, so it is unclear whether the findings can be transferred to microservices applications that use other technologies.

Moreover, in the reference applications all message transfers are implemented as HTTP messages. However, HTTP messages are hardly used in multiplayer games of a similar nature. For real-time communication, e.g. for transmitting the state of the game world, there are better alternatives. For example, `slither.io` uses WebSocket to synchronize the available items and the locations of the other players. Websocket connections have less overhead than HTTP and thus allow for faster message transfers[PFH13]. Therefore, it is possible that the results do not generalize to microservice applications that use other protocols for communication.

In addition, the reference applications use the HTTP/1.1 protocol without encryption for communication. However, a large proportion of websites now use encrypted http connections[11] by default. Encryption and decryption of messages can have an impact on performance[He03]. In addition, about one third of all websites use the newer HTTP/2 protocol[12]. Also, using HTTP/2 over HTTP/1.1 brings performance changes[BMR17]. Therefore, the measurements of the reference applications may not translate to other applications that use different HTTP protocols.

The logic of the reference applications might be less complex than the corresponding logic in production systems. For example, the authentication logic is very lean. The authentication token contains only the user ID of the corresponding account. For comparison, Gmail sends a 1024 byte cookie with the GET request to the server when the web interface is initially called. This means that in the reference applications generally less data is sent from the clients to the server than would be the case in a similar production system. Furthermore, the token signature is not verified on the server side, which is essential in any production system. This results in slightly less logic being executed on each request than would be the case in a similar production system. In addition, the number of deployed services in the FruitCollector architecture might be lower than in production microservice architectures. For example, in 2020, Uber used 1100 microservices to realize its business model[13]. In addition, the load on production systems is much higher and is distributed across many replicas of the services. For example, the Netflix microservices process more than 2 billion requests per day[14]. In our evaluation, we have relatively low load on the FruitCollector services. Furthermore every service in our Evaluation only has one replica, so the impact of load balancing and provisioning of new replicas is not explored.

[11]`https://w3techs.com/technologies/details/ce-httpsdefault`
[12]`https://w3techs.com/technologies/details/ce-http2`
[13]`https://www.uber.com/en-us/blog/microservice-architecture/`
[14]`https://netflixtechblog.com/optimizing-the-netflix-api-5c9ac715cf19`

9. Discussion

Contents

In this chapter, we first answer the research questions and then discuss the limitations of our answers.

9.1. Research Question Findings

The goal of this thesis is to contribute to a microservice architecture exploration platform by examining whether model-based simulation is suitable to predict the performance impact of architecture decisions. With the first research question we want to answer whether it is worthwhile to pursue this approach further.

> ***RQ 1*** *Can model-based architecture simulation be a meaningful approach to predict performance of microservice applications?*

We examined the performance impact of an architectural decision between messaging via a message broker or direct messaging (brokerless). In our evaluation, we have shown that there are measurable performance differences between broker messaging and brokerless messaging in a real-world microservice application. The message broker leads to increased communication times between the different services, which is exacerbated when there is an increased load on the system. Our developed model-based simulation was also able to detect that broker messaging leads to increased communication times between services. And the event logs provided as output are also detailed enough to detect the cause of the performance differences, namely the overload of the broker host machine. However, the simulation vastly overestimated the performance impact of using a message broker and found a complete overload of the microservice application already with 50 simulated users. While the real broker messaging microservice application only felt a minor performance impact due to the increased number of users. Consequently, the impact of this architectural decision could not be accurately represented by our model-based simulation approach.

To determine whether these findings mean that model-based simulation is generally not a meaningful approach for performance prediction, we need to examine the underlying issues in our approach. For this we consider the two factors that influence the performance of a microservice application, the transfer of messages and the processing of messages by services.

Our approach for the **prediction of message transfer durations** presented in chapter 5 already provides promising results. However, this approach suffers from inaccuracies due to the data used for prediction. Also, fluctuations in the performance of the data transmission infrastructure of the Internet could not be reproduced in this work. Further development of this approach is necessary to predict the performance of microservice applications more accurately.

However, the main reason for the large discrepancy between actual and predicted performance is the processing of messages by the services. Our **CPU model** and the milliseconds resolution of the simulation times are insufficient and cannot correctly represent the actual functioning of modern CPUs. Complex message processing blocks the entire computing time of a CPU in our model, which does not correspond to reality and the milliseconds resolution leads to accumulating rounding errors. This causes the broker service to act as a bottleneck between all services and rapidly degrade the overall simulated application performance. Thus, enhancements to the CPU model and a more fine-grained time resolution are necessary to achieve accurate performance predictions.

Furthermore, a fundamental problem of the model-based approach is the **modeling of the processing effort**. It is difficult to estimate how much processing effort is required by the business logic of a service, especially when the implementation details are not yet known at the time of architecture creation. In our models, for example, we underestimated the processing effort for a registration, while we overestimated the relevance of message sizes for the processing times of our own algorithms. Existing performance prediction approaches use benchmarks of already running microservice applications for this purpose, which are not available for our approach. Therefore, an estimation aid for the processing effort of message processing by services is necessary to obtain accurate performance predictions.

Due to the identified issues with our simulation approach, we cannot provide a conclusive answer to this research question. However, we remain optimistic that the identified issues can be addressed and the accuracy of the simulation can be improved. Therefore, we suggest that model-based simulation is a promising approach to predict the performance of microservice applications.

> ***RQ 2*** *What adaptations to the metamodel of Ntentos et al. [Nte+21] are necessary, to enable it to support performance prediction simulation?*

To answer this question, we propose an extended metamodel based on the Ntentos et al. [Nte+21] metamodel in section 4.2. For this purpose, we have extended the metamodel with request types, a more detailed modeling capability for the communication between the components. With the request types, the messaging chains that arise for processing requests to the application can be modeled precisely. We have also introduced a deployment model that can be used to model the hardware on which the services of a microservice application run. This allows a simulation of the usage of the hardware and the transfer of messages between different machines. Furthermore, with the scenarios and client definitions we have created possibilities to precisely model the usage of the microservice application for which the performance is to be predicted.

With these extensions, it is possible to integrate performance prediction into a microservice architecture exploration platform to help software architects build good architectures, along with automatic verification of compliance with architectural patterns by Ntentos et al. [Nte+21].

9.2. Limitations

In this work, we focused on the performance impact of the architectural decision between broker-based and brokerless messaging. This limits our findings to only a tiny fraction of possible architectural patterns. For example, in his microservice architecture pattern language, Richardson [Ric18] lists 50 patterns for constructing microservices. Our simulation and the input models were tailored to the messaging pattern. Whether the basic concepts of our simulation still hold when predicting performance impacts of all the other patterns remains unanswered.

Even within the decision between broker and brokerless messaging, there are countless factors that we have not considered in this work. For example, Wikipedia lists 39 different implementations for message brokers[1]. In this work, we have only considered the performance impact of a single implementation. In addition, message brokers provide tuning options, for example for different types of message delivery guarantees. Whether our findings can be applied to all combinations of message broker implementations and different configurations also remains unanswered.

For the evaluation of the simulation, we developed and deployed an exemplary microservice application. However, due to hardware limitations, we only evaluated the performance under a relatively minor workload by simulating 50 users sending requests. Thus, the number of requests is lower than the number of requests we would expect in a production microservice application. Also, our reference application consists of only 7 services, but in reality microservice architectures may consist of many more services and thus trigger much longer messaging chains between them. Furthermore, we have not considered some aspects of deployment in our work, such as horizontal and vertical scaling or load balancing. How the performance behaves when these aspects are included is also unclear. As a result, the question of whether our findings can be applied to real-world microservice applications also remains unanswered.

[1]`https://en.wikipedia.org/wiki/Message_broker`

10. Conclusion

Contents

10.1. Summary

This thesis investigated, whether model-based simulation is a meaningful approach to predict the performance of microservice applications. Such performance prediction could then be used in the future in conjunction with existing methods for verification of compliance with microservice architectural design patterns to help software architects create good microservice architectures.

For our investigation, we first created two variants of a reference microservice architecture for a multiplayer game backend, which only differ in how the services communicate with each other. While the services in one variant communicate with each other via a message broker, the services in the other variant communicate directly (brokerless). We have created these reference architectures to assess whether the simulation is able to predict the difference in performance between broker and brokerless communication. Based on the reference architectures, we then derived the necessary concepts to develop a model-based simulation for performance prediction. The developed discrete-event simulation then uses models describing an architecture, a deployment and a usage scenario as input to simulate the transfer of messages between services and the processing of messages by services.

To evaluate the accuracy of the simulation, we first modeled both variants of the reference architectures for the simulation and then simulated them to obtain the predicted performance. Then we implemented both reference architectures as microservice applications and measured the actual performance in the same scenarios as defined in the simulation inputs. The simulation predicted a complete overload of the system when using a message broker, while the actual performance revealed only slightly increased communication times between the services. Our investigations have shown that this is mainly due to the oversimplified CPU model of the simulation.

As a result of these findings, we conclude that model-based simulation is fundamentally a meaningful approach for performance prediction of microservice applications, although significant refinements of the concepts are still necessary.

10.2. Future Work

We see this thesis as part of an effort to create a comprehensive microservice architecture exploration platform, which can help architects building good microservice architectures. In this section we give an overview of future work that builds on this thesis to lead to such an exploration platform.

Improving prediction of processing times

First, the issues with the current version of the simulation need to be addressed. For this, an improved CPU model for simulating hardware must be created. The current CPU model is based on the principles of a first come first serve scheduler, which is, however, not suitable to represent the functioning of real CPUs. A more suitable approach could be, for example, to mimic the functionality of the Completely Fair Scheduler of Linux for the simulation. In particular, it is important that the scheduler is able to interrupt processes to avoid a blocking of the CPU by a few complex tasks.

In addition, assistance is needed for the estimation of base processing times, i.e. the times that the CPU model takes as a basis for the simulation. For non-application-specific functionalities of services, such as login and registration, a library of validated base processing times could be created that users of the simulation can access. Approaches for estimating the base processing times of application-specific algorithms also require investigation.

Refining message transfer time prediction

The message transfer time prediction of the simulation can also be improved in future work. The current implementation suffers particularly from two deficiencies in the training data that make the prediction inaccurate for some cases.

First, the clustering of the different data centers into AWS districts is too simplistic, as the clustering does not take into account the different quality of Internet infrastructure in the different AWS districts. For example, Europe is divided into several clusters, while Singapore and Australia are considered in the same cluster, even though the underlying round-trip times between the data centers in the clusters do not support this division. Thus, improving the clustering could lead to an improvement in message transfer time prediction.

Second, for message transfer time prediction, we only use the measured round-trip times between different data centers. However, data centers are usually better connected than users of a microservice application. Our message transfer time prediction is therefore not able to accurately predict the transfer times for messages sent by users, since they are often connected to the Internet via mobile data connections or copper lines, for example. Therefore, an improvement of the message transfer time prediction, which also takes into account the connection type of the user, is needed.

Supporting more microservice applications

In this work, we designed the concepts of the simulation based on the FruitCollector reference application, so the simulation can only represent a small number of possible performance impacts. Thus, the performance impact of other messaging protocols such as UDP or the performance impact of document-oriented databases cannot be simulated. Furthermore, we have not considered the performance impact of horizontal scaling strategies, such as on-demand deployment of more instances or serverless deployment. However, the ability to easily scale horizontally is one of the strengths of microservice applications[Ric18]. Also, for many other possible performance impacts concepts are still missing in the simulation. Therefore, the simulation needs to be extended to include concepts that can be used to simulate other performance impacts.

Towards a comprehensive microservice architecture exploration platform

In the current implementation, input models are passed to the simulation using lengthy JSON files, which is a cumbersome manual process. A better option could be the graphical creation of the input models. Especially the definition of the different components and the messaging chains between the components could be done much easier via a graphical user interface. Therefore, we see a possibility to support the development of a future exploration platform by creating a user interface for the simulation. The models created in this way could then also be used to verify compliance with architectural design patterns.

A. Appendix

A.1. FruitCollector API

register

Involved services Authentication **Uses database** ✓

Description

The client sends username and password to the API to create an account entry in the database.

login

Involved services Authentication **Uses database** ✓

Description

The client sends username and password to the API. The authentication service checks whether a corresponding account exists and sends back an authentication token for future authentication.

move

Involved services World **Uses database** ✗

Description

The user sends a new desired position within his moving range and is moved to the desired position by the service.

collectItem

Involved services World, Inventory **Uses database** ✓

Description

The user sends the id of a fruit in the game world that is within his collection range. The World service checks if the fruit is in range and sends a message to the Inventory service to add the fruit to the player's inventory.

inventoryState

Involved services	Inventory	**Uses database**	✓

Description
Returns a list of all fruits in the player's inventory.

sendChat

Involved services	Chat	**Uses database**	✗

Description
The user sends a chat message to communicate with the other players.

retrieveChat

Involved services	Chat	**Uses database**	✗

Description
The client periodically queries which messages the chat contains in order to display them to the player.

retrieveQuests

Involved services	Quest	**Uses database**	✗

Description
The client periodically queries which quests are currently active and how many of what kind of fruits need to be collected for them.

A.2. Simulation JSON Model Definitions

```json
...
"properties": {
  "aachen-university": {
    "networkAccess": {
      "uploadSpeed": 117964800,
      "downloadSpeed": 104857600,
      "location": {
        "latitude": 50.775555,
        "longitude": 6.083611,
        "radiusKm": 1
      }
    }
  }
},
...
```

Source Code A.1: Excerpt of a *clients.json* defining client properties.

```json
...
"behaviors": {
  "default": {
    "requests": {
      "login": {
        "frequency": {
          "amount": "1",
          "time": "DAYS"
        }
      },
      "playerView": {
        "frequency": {
          "amount": "8",
          "time": "SECONDS"
        }
      },
...
```

Source Code A.2: Excerpt of a *clients.json* defining a client behavior.

```json
...
"dataCenters": [
  {
    "topologyRef": "config-1",
    "latitude": 51.509865,
    "longitude": -0.118092
  }
],
"hostTopologies": {
  "config-1": {
    "hosts": [
      {
        "id": "host-1",
        "logicalCores": 2,
        "coreSpeedGhz": 2.7,
        "components": ["gateway", "quest"],
        "uploadSpeed": 2000000000,
        "downloadSpeed": 2000000000
      },
...
```

Source Code A.3: Excerpt of a *deployment.json* defining a data center and the corresponding host topology.

```json
...
"interpolater": "step",
"snapshots": [
  {
    "clients": {
      "default": 25
    },
    "atTime": "PT0S"
  },
  {
    "clients": {
      "default": 50
    },
    "atTime": "PT60S"
  },
...
```

Source Code A.4: Excerpt of a *scenario.json* defining a scenario.

Bibliography

[AA18] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating systems: Three easy pieces.* Arpaci-Dusseau Books, LLC, 2018 (cit. on p. 23).

[AA19] F. Al-Turjman and F. Al-Turjman. *Edge computing.* Springer, 2019 (cit. on p. 17).

[AP19] A. Akbulut and H. G. Perros. "Performance analysis of microservice design patterns." In: *IEEE Internet Computing* 23.6 (2019), pp. 19–27 (cit. on pp. 2, 23).

[Bao+19] L. Bao et al. "Performance modeling and workflow scheduling of microservice-based applications in clouds." In: *IEEE Transactions on Parallel and Distributed Systems* 30.9 (2019), pp. 2114–2129 (cit. on pp. 2, 3, 9, 15, 17, 18).

[Bar16] M. Barad. "Petri Nets—A Versatile Modeling Structure." In: *Applied Mathematics* 07.09 (2016), pp. 829–839. ISSN: 2152-7385. DOI: `10.4236/am.2016.79074`. URL: `https://www.scirp.org/journal/paperinformation.aspx?paperid=66770` (cit. on p. 20).

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice.* Addison-Wesley Professional, 2003 (cit. on pp. 1, 88).

[Bec+06] S. Becker et al. "Performance prediction of component-based systems: A survey from an engineering perspective." In: *Architecting Systems with Trustworthy Components: International Seminar, Dagstuhl Castle, Germany, December 12-17, 2004. Revised Selected Papers.* Springer. 2006, pp. 169–192 (cit. on p. 3).

[BM02] R. Buyya and M. Murshed. "Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing." In: *Concurrency and computation: practice and experience* 14.13-15 (2002), pp. 1175–1220 (cit. on p. 22).

[BMR17] L. Bach, B. Mihaljević, and A. Radovan. "Exploring HTTP/2 advantages and performance analysis using Java 9." In: *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* IEEE. 2017, pp. 1522–1527 (cit. on p. 88).

[BNK18] S. Baškarada, V. Nguyen, and A. Koronios. "Architecting microservices: Practical opportunities and challenges." In: *Journal of Computer Information Systems* (2018) (cit. on p. 1).

[BSB06] R. Beverly, K. Sollins, and A. Berger. “SVM learning of IP address structure for latency prediction.” In: *Proceedings of the 2006 SIGCOMM workshop on Mining network data*. 2006, pp. 299–304 (cit. on pp. 32, 33).

[COQ21] C. Courageux-Sudan, A.-C. Orgerie, and M. Quinson. “Automated performance prediction of microservice applications using simulation.” In: *2021 29th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2021, pp. 1–8 (cit. on pp. 2, 8).

[Cos+04] M. Costa et al. “PIC: Practical Internet coordinates for distance estimation.” In: *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE. 2004, pp. 178–187 (cit. on p. 32).

[CS04] L. Chittaro and M. Serra. “Behavioral programming of autonomous characters based on probabilistic automata and personality.” In: *Computer animation and virtual worlds* 15.3-4 (2004), pp. 319–326 (cit. on p. 18).

[CY05] J.-H. Choi and C. Yoo. “One-way delay estimation and its application.” In: *Computer Communications* 28.7 (2005), pp. 819–828 (cit. on p. 38).

[Dab+04] F. Dabek et al. “Vivaldi: A decentralized network coordinate system.” In: *ACM SIGCOMM Computer Communication Review* 34.4 (2004), pp. 15–26 (cit. on pp. 32, 33).

[Dra+17] N. Dragoni et al. “Microservices: yesterday, today, and tomorrow.” In: *Present and ulterior software engineering* (2017), pp. 195–216 (cit. on pp. 1, 5).

[DWW05] M. Dick, O. Wellnitz, and L. Wolf. “Analysis of factors affecting players’ performance and perception in multiplayer games.” In: *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. 2005, pp. 1–7 (cit. on p. 12).

[Eke+06] S. Ekelin et al. “Real-time measurement of end-to-end available bandwidth using kalman filtering.” In: *2006 ieee/ifip network operations and management symposium noms 2006*. IEEE. 2006, pp. 73–84 (cit. on p. 86).

[Feh+14] C. Fehling et al. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Vol. 545. Springer, 2014 (cit. on pp. 1, 18).

[FFF18] M. Fleckenstein, L. Fellows, and K. Ferrante. *Modern data strategy*. Springer, 2018 (cit. on p. 25).

[Fis01] G. S. Fishman. *Discrete-event simulation: modeling, programming, and analysis*. Vol. 537. Springer, 2001 (cit. on p. 6).

[Fra+01] P. Francis et al. “IDMaps: A global Internet host distance estimation service.” In: *IEEE/ACM Transactions On Networking* 9.5 (2001), pp. 525–540 (cit. on p. 32).

[Gen+18] Y. Geng et al. "Exploiting a natural network effect for scalable, fine-grained clock synchronization." In: *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18).* 2018, pp. 81–94 (cit. on p. 65).

[GIM+17] M. Gribaudo, M. Iacono, D. Manini, et al. "Performance evaluation of massively distributed microservices based applications." In: *Proceedings-31st European Conference on Modelling and Simulation, ECMS 2017.* European Council for Modelling and Simulation. 2017, pp. 598–604 (cit. on p. 2).

[Gou+02] D. Gourley et al. *HTTP: the definitive guide.* " O'Reilly Media, Inc.", 2002 (cit. on p. 63).

[He03] X. He. "A performance analysis of secure http protocol." In: *STAR Lab Technical Report, Department of Electrical and Computer Engineering, Tennessee Tech University* (2003), pp. 517–524 (cit. on p. 88).

[Hei+17] R. Heinrich et al. "Performance engineering for microservices: research challenges and directions." In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion.* 2017, pp. 223–226 (cit. on p. 3).

[HM98] F. Howell and R. McNab. "A discrete event simulation package for Java." In: *1998 International Conference on Web-Based Modeling & Simulation.* 1998 (cit. on p. 3).

[HSS+17] M. Hofmann, E. Schnabel, K. Stanley, et al. *Microservices best practices for Java.* IBM Redbooks, 2017 (cit. on p. 2).

[JK+13] K. James F, R. Keith W, et al. *Computer Networking: A Top-Down Approach.-6th.* Addison-Wesley, 2013 (cit. on p. 38).

[KBB19] F. Klinaku, D. Bilgery, and S. Becker. "The applicability of palladio for assessing the quality of cloud-based microservice architectures." In: *Proceedings of the 13th European Conference on Software Architecture-Volume 2.* 2019, pp. 34–37 (cit. on p. 2).

[Kha+21] M. G. Khan et al. "Perfsim: A performance simulator for cloud native computing." In: *arXiv preprint arXiv:2103.08983* (2021) (cit. on pp. 2, 8).

[Lan+13] R. Landa et al. "Measuring the relationships between Internet geography and RTT." In: *2013 22nd International Conference on Computer Communication and Networks (ICCCN).* IEEE. 2013, pp. 1–7 (cit. on pp. 32–35).

[MAB18] D. Mirkovic, G. Armitage, and P. Branch. "A survey of round trip time prediction systems." In: *IEEE Communications Surveys & Tutorials* 20.3 (2018), pp. 1758–1776 (cit. on pp. 31–33).

[Mit90] R. J. Mitchell. *Managing complexity in software engineering.* 17. IET, 1990 (cit. on p. 1).

[MWW12] D. Meisner, J. Wu, and T. F. Wenisch. “Bighouse: A simulation infrastructure for data center systems.” In: *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE. 2012, pp. 35–45 (cit. on p. 2).

[Nah04] F. F.-H. Nah. “A study on tolerable waiting time: how long are web users willing to wait?” In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163 (cit. on p. 1).

[Nte+20] E. Ntentos et al. “Assessing architecture conformance to coupling-related patterns and practices in microservices.” In: *Software Architecture: 14th European Conference, ECSA 2020, L’Aquila, Italy, September 14–18, 2020, Proceedings 14*. Springer. 2020, pp. 3–20 (cit. on pp. 14, 15).

[Nte+21] E. Ntentos et al. “Evaluating and Improving Microservice Architecture Conformance to Architectural Design Decisions.” In: *Service-Oriented Computing: 19th International Conference, ICSOC 2021, Virtual Event, November 22–25, 2021, Proceedings 19*. Springer. 2021, pp. 188–203 (cit. on pp. 2, 3, 7, 11, 90, 91).

[NZP04] N. Nurmuliani, D. Zowghi, and S. Powell. “Analysis of requirements volatility during software development life cycle.” In: *2004 Australian Software Engineering Conference. Proceedings.* IEEE. 2004, pp. 28–37 (cit. on p. 1).

[Pab09] C. S. Pabla. “Completely fair scheduler.” In: *Linux Journal* 2009.184 (2009), p. 4 (cit. on p. 23).

[Pac18] V. F. Pacheco. *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices.* Packt Publishing Ltd, 2018 (cit. on p. 2).

[PFH13] D. G. Puranik, D. C. Feiock, and J. H. Hill. “Real-time monitoring using ajax and websockets.” In: *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*. IEEE. 2013, pp. 110–118 (cit. on p. 88).

[PO71] M. A. Poole and P. N. O’Farrell. “The assumptions of the linear regression model.” In: *Transactions of the Institute of British Geographers* (1971), pp. 145–158 (cit. on p. 35).

[Reu+16] R. H. Reussner et al. *Modeling and simulating software architectures: The Palladio approach.* MIT Press, 2016 (cit. on p. 3).

[Ric18] C. Richardson. *Microservices patterns: with examples in Java.* Simon and Schuster, 2018 (cit. on pp. 1, 2, 5–7, 17, 58, 91, 95).

[Rob05] S. Robinson. “Discrete-event simulation: from the pioneers to the present, what next?” In: *Journal of the Operational Research Society* 56 (2005), pp. 619–629 (cit. on p. 6).

[SK15] P. Sriramya and R. Karthika. “Providing password security by salted password hashing using bcrypt algorithm.” In: *ARPN journal of engineering and applied sciences* 10.13 (2015), pp. 5551–5556 (cit. on p. 79).

[SN18] W. Stadnik and Z. Nowak. “The impact of web pages’ load time on the conversion rate of an e-commerce platform.” In: *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology–ISAT 2017: Part I.* Springer. 2018, pp. 336–345 (cit. on p. 2).

[Tek+16] B. Tekinerdogan et al. “Quality concerns in large-scale and complex software-intensive systems.” In: *Software Quality Assurance.* Elsevier, 2016, pp. 1–17 (cit. on p. 1).

[Tol+19] S. S. de Toledo et al. “Architectural technical debt in microservices: a case study in a large company.” In: *2019 IEEE/ACM International Conference on Technical Debt (TechDebt).* IEEE. 2019, pp. 78–87 (cit. on p. 2).

[Tuo02] I. Tuomi. “The lives and death of Moore’s Law.” In: *First Monday* (2002) (cit. on p. 22).

[Van16] W. Van Der Aalst. *Process mining: data science in action.* Vol. 2. Springer, 2016 (cit. on p. 26).

[WSS05] B. Wong, A. Slivkins, and E. G. Sirer. “Meridian: A lightweight network location service without virtual coordinates.” In: *ACM SIGCOMM Computer Communication Review* 35.4 (2005), pp. 85–96 (cit. on pp. 31, 33).

[ZGD19] Y. Zhang, Y. Gan, and C. Delimitrou. “uqSim: Scalable and Validated Simulation of Cloud Microservices.” In: *arXiv preprint arXiv:1911.02122* (2019) (cit. on pp. 2, 8).

[Ziv+05] A. Ziviani et al. “Improving the accuracy of measurement-based geographic location of Internet hosts.” In: *Computer Networks* 47.4 (2005), pp. 503–523 (cit. on p. 34).