Oona Laitamäki

WEB APPLICATION ARCHITECTURE FOR REAL-TIME MOBILE NETWORK ANALYSIS

Master's Thesis
Faculty of Information Technology and Communication Sciences
Examiners: Kari Systä & David Hästbacka
April 2023

ABSTRACT

Oona Laitamäki: Web application architecture for real-time mobile network analysis
Master's Thesis
Tampere University
Master's Degree Programme in Information Technology
April 2023

Mobile networks have an important role in the functioning of society today, which makes it necessary to maintain and develop mobile networks. There are various programs for mobile network analysis that can automatically identify problems in the network even in real-time. In this work, a web application architecture enabling real-time analysis is designed as part of an existing analysis program. The work is also partially developed as a so-called proof of concept, i.e. to test whether the idea works in practice.

When designing the architecture, different solutions for key issues in architecture are sought using literature review and previous experience. The first challenges are rational placement of components in distributed architecture and responsibility sharing between components. Component division must provide sufficient scalability and performance to enable real-time analysis without the loss of user experience. In the selected component division, one component was placed on distributed servers, which forms distributions of key performance indicators collected from a mobile network system. In this way, the amount of traffic on a centralized server is lower. A dedicated component is responsible for combining calculated distributions on a centralized server. Moreover, the web application has a separate backend component and a component responsible for presenting a user interface.

Research will also consider suitable technologies for the implementation of the backend component. Go and Node.js were compared as backend technology options. Since scalability was emphasized in the choice of technology, Go was selected for the implementation of the component. The new component also needed an application programming interface suitable for its purpose. Compared API implementations were traditional REST API and GraphQL API. Performance and maintainability were emphasized in the comparative study and previous research on the subject was also used in the comparison. Eventually, REST API was selected for use in the backend component.

In addition, various solutions were sought for real-time communication between backend and frontend components. Compared real-time communication technologies were HTTP polling, WebSocket and server-sent events. Alternatives were compared especially based on quality attributes. Moreover, the performance of real-time communication technologies was compared by carrying out an empirical study in the assumed application use case. Server-sent event technology was selected for the implementation.

Based on the selected architecture and technology options, an architecture proposal was conducted, and a more detailed structure was presented for the new backend component. Based on this proposal, an architecture evaluation was carried out at the end of the research. The evaluation was based on a lighter version of the ATAM method.

Keywords: mobile networks, web application architecture, distributed architecture, scalability

TIIVISTELMÄ

Oona Laitamäki: Verkkosovellusarkkitehtuuri mobiiliverkkojen reaaliaikaiseen analysointiin
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Huhtikuu 2023

Mobiiliverkoilla on nykypäivänä tärkeä rooli yhteiskunnan toimivuudessa, minkä vuoksi on välttämätöntä huolehtia verkkojen ylläpidosta sekä kehittää niiden toimintaa. Mobiiliverkkojen analysointiin on olemassa erilaisia ohjelmia, joiden avulla voidaan tunnistaa verkossa esiintyviä ongelmia automaattisesti myös reaaliajassa. Tässä työssä suunnitellaan reaaliaikaisen analyysin mahdollistava verkkosovellusarkkitehtuuri osaksi jo olemassa olevaa analysointiohjelmaa. Työ toteutetaan myös osittain niin kutsuttuna soveltuvuusselvityksenä (engl. proof of concept) eli testataan, toimiiko idea käytännössä.

Arkkitehtuuria suunniteltaessa keskeisiin arkkitehtuurillisiin ongelmiin etsitään erilaisia ratkaisuja kirjallisuuskatsausta sekä aikaisempaa kokemusta hyödyntäen. Ensimmäiset haasteista ovat komponenttien järkevä sijoittelu hajautettuun arkkitehtuuriin sekä komponenttien keskinäinen vastuunjako. Komponenttijaon on tarjottava riittävän hyvä skaalautuvuus sekä suorituskyky, jotta analysointi onnistuu reaaliaikaisesti ilman käyttäjäkokemuksen heikkenemistä. Valitussa komponenttijaossa hajautetuille palvelimille sijoitettiin komponentti, joka muodostaa jakaumia mobiiliverkosta kerätyistä mittareista. Tällä tavalla keskitetylle palvelimelle kulkevan liikenteen määrä on pienempi. Keskitetyllä palvelimella jakaumien yhdistämisestä vastaa oma komponentti. Lisäksi verkkosovelluksella on erillinen backend-komponentti sekä käyttöliittymän esittämisestä vastaava komponentti.

Tutkimuksessa pohditaan myös backend-komponentin toteutukseen sopivaa teknologiaa. Go ja Node.js teknologioita vertailtiin vaihtoehtoisina backend-teknologioina. Koska teknologian valinnassa painotettiin erityisesti skaalautuvuutta, valittiin komponentin toteutukseen Go. Uusi komponentti tarvitsi myös käyttötarkoitukseen sopivan ohjelmointirajapinnan. Työssä vertailtavat vaihtoehdot ohjelmointirajapinnan toteutukseen olivat perinteinen REST-ohjelmointirajapinta sekä GraphQL-ohjelmointirajapinta. Näiden vertailussa painotettiin erityisesti suorituskykyä sekä ylläpidettävyyttä. Vertailussa hyödynnettiin myös aikaisempaa tutkimusta aiheesta. Vaihtoehdoista REST-ohjelmointirajapinta valittiin käytettäväksi backend-komponentissa.

Tämän lisäksi backend- ja frontend-komponenttien väliselle reaaliaikaiselle liikenteelle haettiin erilaisia ratkaisuja. Vertailtavat reaaliaikaiset kommunikointiteknologiat olivat HTTP polling, WebSocket ja server-sent events. Ratkaisuja vertailtiin erityisesti laatuattribuuttien pohjalta. Lisäksi reaaliaikaisten kommunikointiteknologioiden suorituskykyä vertailtiin toteuttamalla empiirinen tutkimus sovelluksen oletetussa käyttötapauksessa. Sovelluksen toteutukseen valittiin server-sent events -teknologia.

Valittujen arkkitehtuuri- ja teknologiavaihtoehtojen pohjalta muodostettiin arkkitehtuuriehdotus, sekä esiteltiin tarkempi rakenne myös uudelle backend-komponentille. Tämän ehdotuksen perusteella arkkitehtuurille toteutettiin arviointi työn lopussa. Arviointi pohjautui kevyempään versioon ATAM-arviointimenetelmästä.

Avainsanat: mobiiliverkot, verkkosovellusarkkitehtuuri, hajautettu arkkitehtuuri, skaalautuvuus

PREFACE

This thesis work was done for Nokia Solutions and Networks Oy. I would like to thank my team for providing the possibility to write my thesis about this topic. Especially, acknowledgements go to my co-workers Petri Aalto and Marjaana Laine. Also, special thanks to Professor Kari Systä for the excellent guidance and supervision of this thesis. I would also like to thank my family and friends for their invaluable help and knowledge throughout this process.

Tampere, 11.4.2023

Oona Laitamäki

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

3GPP	3rd Generation Partnership Project
API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
CIA (triad)	Confidentiality, Integrity and Availability
CPU	Central Processing Unit
DOM	Document Object Model
eNB	eNodeB, 4G base station
E-UTRAN	Evolved Universal Terrestrial Radio Access Network
gNB	gNodeB, 5G base station
gRPC	Google Remote Procedure Call
GUI	Graphical User Interface
Heccla	Health Check Control Language
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LAE	Lightweight Architecture Evaluation
MPA	Multi-Page Application
OSI	Open Systems Interconnection
REST	Representational State Transfer
RPC	Remote Procedure Call
RTPM	Real-Time Performance Monitoring
SQL	Structured Query Language
SPA	Single-Page Application
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
XML	Extensible Markup Language
B/s	byte per second
kB/s	kilobyte per second
MiB	mebibyte

1. INTRODUCTION

The first chapter introduces the background for the topic of this thesis work. Research questions and methodologies are formed after background. Furthermore, the outline for the thesis is described at the end of this chapter.

1.1 Background

Mobile networks are a significant part of people's everyday interactions and activities. Mobile networks and phones have evolved from an expensive and rare technology into an everyday commodity used by most people on a daily basis. Therefore, the amount of traffic on mobile networks has exponentially increased over the years. Failures in the mobile network may have big economic, social and security impacts. This makes it even more critical to identify problems that may exist in the network, as they may have widely recognized effects. With a continuous increase in demand of cellular networks, their capacity is challenged.

A cause of failure in mobile network can be found out by following certain measurements. These measurements consider for example throughput, packet sizes and data retransmissions. There are thousands of measurements per base station and even hundreds of thousands of base stations in the network. With 5G technology, the number of base stations has even increased in order to achieve the same coverage as with 4G technology (Lehr, Queder and Haucap, 2021). Additionally, there are many things that affect measurements, which makes the analysis even more complex. For example, there are multiple factors which affect throughput in cellular network. These factors may include radio technology, operator infrastructure and physical layer effects, like interference and fading (Walelgne *et al.*, 2018). As the manual analysis of failures in the cellular network requires a lot of time, there exist programs which collect and analyse different measurements from the network, also in real-time. It is good to find the cause of the failure quickly. Automatic troubleshooting of extensive cellular networks is vital to ensure the efficient use of network infrastructure and providing the quality of user experience (Rezaei *et al.*, 2016).

This thesis work is about developing a web application as a part of a system which collects different measurements and analyses the state of cellular networks in real-time. The designed web application is responsible for displaying informative statistics about the mobile network condition based on the KPI (Key Performance Indicators) analysis. 3GPP (3rd Generation Partnership Project) maintains technical specifications for KPI calculations. There are definitions for all used E-UTRAN (Evolved Universal Terrestrial Radio Access Network) KPIs in 3GPP technical specification (3GPP 32.450, 2022). A KPI is a measurable value used to track and evaluate the success of a system or a specific aspect of a system towards its objectives. There is a huge amount of different KPIs that indicate performance and operating status of the cellular network. In the context of cellular network KPIs are formulas that are built from different measurement counters. A measurement counter in a cellular network refers to values used to track and measure network performance metrics, such as signal strength, network utilization, and data transfer rates. This information is used by network operators to optimize network performance, ensure service quality, and troubleshoot issues. Measurement counters are important for network monitoring and management systems in cellular networks.

The naming of different KPIs might be confusing but in order to know something is wrong, the user must be able to interpret the KPI metrics. Sometimes user interfaces contain too much information and may not give the user a right picture of current situation of a system. User interface should be informative and easy to use. The intention is not to turn the user interface into a window filled with data from different metrics. Such an extremely informative interface can be playfully referred to as an engineering interface, as they are most often implemented from the developers' point of view and not so much from the user's point of view. Even though user interface design is important in this application, this thesis work is built around the architectural design.

The goal of this work is to design a web application architecture that would enable displaying real-time KPI statistics on the web user interface in a way that is informative to the user. A huge data traffic and amount of KPIs pose challenges for the architecture. During this research different architecture solutions are explored and evaluated. Thesis contains practical work but most of all it focuses on the architectural design of the web application. The designed application is developed as a proof of concept and implementation is started alongside this constructive research. This thesis is made for a mobile network software development team of the Nokia organization.

1.2 Research questions and methodologies

The main contribution in this work is to develop a web application architecture which is suitable for a real-time analysis of mobile network measurement data. In this context a web application is a software system that runs on multiple servers and can be accessed through a web browser over the internet. Web application considers both frontend component and its backend service, but it also considers all the services that are used by the backend service. The designed application is developed as a part of an already existing system that collects and analyses mobile network state.

This thesis finds and evaluates different architectural choices to use in the web application. During the research a part of the application is developed as a proof of concept to see if selected architecture is suitable or not. Thus, the main research question of this thesis is formalized as follows: *What kind of web application architecture is suitable for analysing and presenting mobile network condition in real-time?*

This study is carried out as constructive research. It aims to develop and create a new architectural solution by building on existing theories, concepts, and practices. The focus is on designing, prototyping, and testing of new solutions. Constructive research process in this work is described in Figure 1. Initial requirements for the web application are gathered in the beginning of the research process. Both functional and non-functional requirements are gathered in the requirement specification. Requirements are specified to define a clear goal and focus points. It also helps to identify potential risks and challenges that the designed architecture may have.

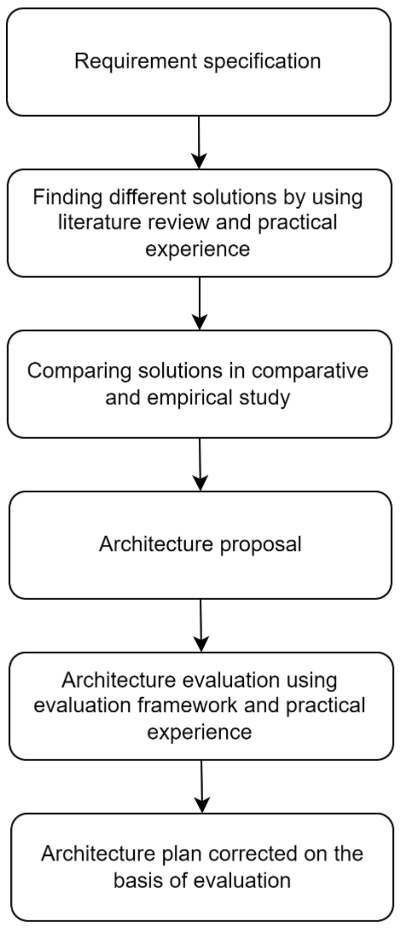
Requirement specification
Finding different solutions by using literature review and practical experience
Comparing solutions in comparative and empirical study
Architecture proposal
Architecture evaluation using evaluation framework and practical experience
Architecture plan corrected on the basis of evaluation

Figure 1. *Constructive research process*

Once the initial requirements have been gathered, architectural solutions are explored. Literature review is used as one primary research method to find different architectural choices for the designed application. These architectural choices are examined mainly from the perspective of the quality attributes in the comparative study. In addition to comparative study, empirical research is conducted to find out which communication technology best supports cost-effective real-time data updates from a backend server to clients. The empirical study considers CPU and memory usage of different technologies. Furthermore, communication technologies are also examined in terms of application requirements.

As the focus of this thesis is on architectural design, compliance with the quality attributes can be predicted already before implementation by examining the architecture. Compliance with both quality attributes and other requirements is evaluated in the end of this work. Architecture evaluation is partially arranged using a lightweight architecture evaluation framework. Architecture proposal is corrected based on the evaluation and there will be an architecture design as a result of this constructive research.

1.3 Outline

Chapter 2 introduces the environment where the web application is going to run. It presents the architecture at a general level and some of the services which are interacting with the new application. Also, initial requirements for the application are introduced in Chapter 2. Both functional and non-functional requirements are specified. It is followed by Chapter 3 which presents the theory of web application architecture and introduces web application architecture and technologies which are analysed more in the comparative study.

In Chapter 4 architecture and technology options are analysed based on literature review in comparative study. In addition, empirical study is used to evaluate performance of different real-time communication methods. Chapter 5 presents an architecture proposal based on the analysis in the previous chapter. It also presents a solution for scalable data processing in backend component in more detailed level. This architecture proposal is evaluated in Chapter 6. Evaluation is carried out with a lightweight evaluation framework and practical experience. In addition to architecture evaluation, Chapter 7 contains analysis and discussion about the research process and presents the plans for future development. The conclusion of this thesis is reported in Chapter 8.

2. APPLICATION ENVIRONMENT AND REQUIREMENTS

The designed application is developed as a part of an already existing system which consists of many different services. The system is built as distributed architecture and each service runs in its own Docker container. Docker containers are built from Docker image based on Linux operating system. Therefore, the web application must run on Linux operating system. Other architecture requirements are specified in sections 2.3 and 2.4. Requirements consist of functional and non-functional requirements, that are also considered as quality attributes.

2.1 Docker in distributed architecture

Docker has been commonly used as containerization technology for different services. It enables virtualization through containers and provides isolation from other applications and processes running on the same system (Sesto *et al.*, 2020). It has also been found that some surveys adopt containerization for better performance than virtual machines (Li *et al.*, 2021). Figure 2 shows how three Docker containers are placed on a single host.

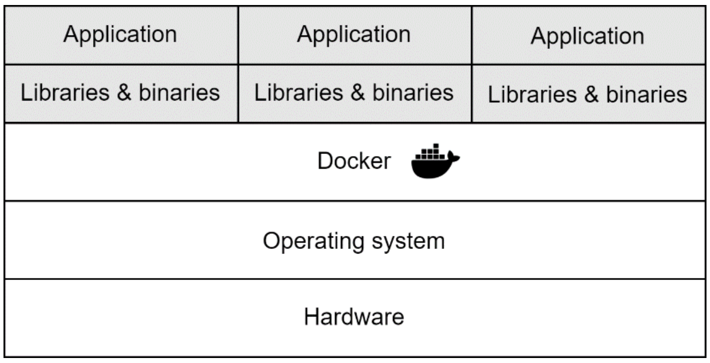

Figure 2. *Docker containerization*

There is no need to add a complete operating system when a new container is brought up. However, containerization still provides isolation and enables own environment inside every container. Each container hosts its own set of libraries, binaries and self-contained dependencies (Sesto *et al.*, 2020).

Docker provides a consistent environment for running applications, regardless of the underlying infrastructure, which makes it a great technology to be used also in distributed architecture. Applications can be developed and tested in a containerized environment and then deployed to any environment that supports Docker. Containers can also be easily scaled up or down to meet changing demands. This makes it easy to add or remove resources as needed, without having to modify the underlying infrastructure.

In distributed architecture the system is decentralized into multiple nodes. These nodes form a collection of autonomous computing elements which appears to users as a single coherent system (van Steen and Tanenbaum, 2016). The number of nodes behind the system can be different depending on the use case. Distributed architecture enables horizontal scalability by allowing a system to add or remove nodes dynamically according to user needs. If services are implemented in a way that supports horizontal scalability, in theory there is no limit for it how many nodes the system can have. However, the distributed structure is not ideal for all processes. For example, doing data queries to multiple nodes requires more effort than doing one data query on one central server.

Distributed architecture enables edge computing by having more ideal locations for handling and filtering data before it is transferred to some other server with a greater distance. In this way there is less data to be transported. As data is increasingly produced at the edge of system network, it is also more effective to process the data at the edge (Shi *et al.*, 2016). This way, the amount of data transfer in the system can be decreased. Even if it might be more efficient to process the data on a centralized server with more capacity, the data transfer is eating the capacity. Comparing to the fast evolution of data processing speed the network bandwidth has not kept up with the processing speeds. The data transportation speed becomes as the bottleneck for the cloud-based computing paradigm. (Shi *et al.*, 2016) The amount of data that can be transferred over a network is limited by the available network bandwidth, and increasing the bandwidth often requires upgrading network infrastructure. Moreover, the protocols used to transfer data over a network add overhead to the transfer process. This overhead can reduce the effective data transfer rate, particularly for small data packets.

2.2 Current system

The current system consists of many different services which are placed on the central server and the edge servers. Figure 3 represents components related to the designed application. Data collector components are located on the edge servers, which takes advantage of edge computing. As there can be even hundreds of thousands of base stations in the cellular network, there is also a lot of data collected from the network. This makes it reasonable to collect and filter the data on the edge nodes and take advantage of edge computing. Data collector services are responsible for collecting the data from base stations and transmitting the data in a specific format to other services. The GUI

(Graphical User Interface) application designed in this work will also receive and present the data that is collected by the Data collector components in real-time.

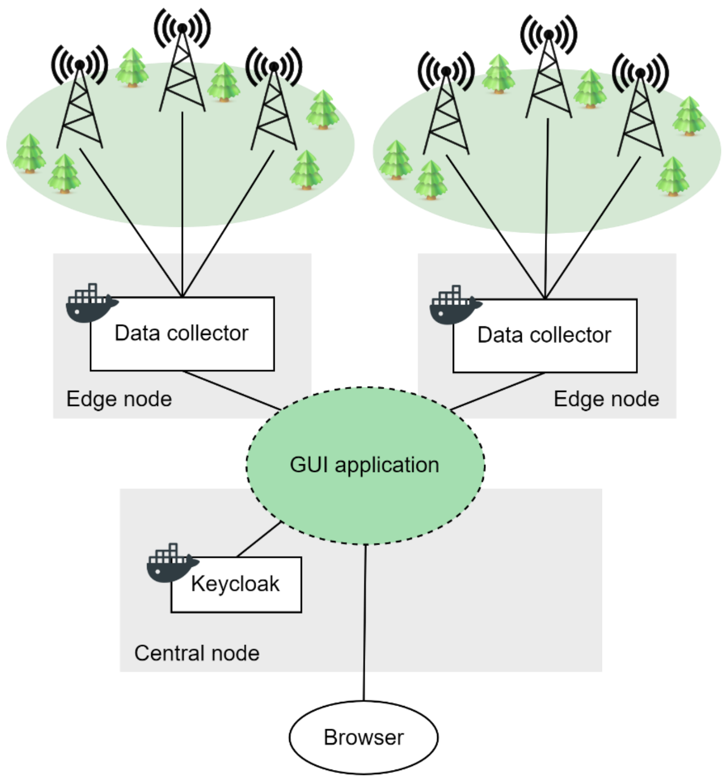

Figure 3. *New GUI application and related services in the current system*

There already exists a user authentication service in the system. Keycloak component is a separate service, which handles authentication of system users. It is an open-source software product for secure access management and identity management. It provides single sign-on for modern applications and services and allows administrators to manage authorization and user identity in a centralized manner. As Keycloak is used by other services in the current system, it will be used also by the designed GUI application. The green component represents the new application whose architecture is designed during this work. The component is located on both edge nodes and central node in the picture above as the location of the application services will be considered later in architecture design.

2.3 Initial functional requirements

At the beginning of this work, the designed web application did not yet have precise functional requirements. The main requirements already known are listed in this section.

1. **Statistics as objects in UI:** User interface must consist of different objects which are responsible for displaying information about the KPIs of a particular topic. The

application design is started with a throughput object, that is responsible for presenting throughput-related information. Throughput indicates the total number of octets carried over an interface. The application should also be able to display historical data and trends for different KPI values.

2. **Data update events:** Data is retrieved from the edge computing components if the user interface has at least one active user viewing statistics. A continuous data update process in the backend server is not required.
3. **Authentication:** Authentication is required whenever someone is using the web application. User authentication is handled with a Keycloak service. Thus, frontend and backend services must be compatible with Keycloak.
4. **Authorization:** There are at least two types of users: default and admin. Default users can log in to the system and view statistics, but they cannot modify the configuration of the statistics view. Admin is the only user who can do modifications to the configuration. Admin user can for example update calculation formulas which define KPI metrics that are used for checking statistics.
5. **UI library:** UI widgets for this application are imported from a library which is only compatible with React framework. Therefore, the frontend component must be developed with React.

2.4 Quality attributes

In addition to the functional requirements there are also non-functional requirements. In this work the non-functional requirements mainly consist of quality attributes which are not very precise in the base application development. However, it is important to consider quality attributes already in the architecture design phase.

2.4.1 Scalability

As mentioned in the previous section, scalability can be achieved for example with distributed architecture. Scalability measures a system's ability to add resources to handle a varying number of requests (Li *et al.*, 2021). In other words, scalability is increasing the system's capacity for work without loss of performance (Bass, 2022). Good scalability means that performance is maintained even if there are more application users or more data traffic. There are two types of scalability: horizontal scalability and vertical scalability. Horizontal scalability, which is also called elasticity, decomposes a monolithic application into independent services with ensuring that every instance of a service scales out conveniently. However, there may be some difficulties in implementing horizontal scalability. For example, sometimes it is hard to determine how many instances of each service are needed to maintain good application quality. (Li *et al.*, 2021)

Where horizontal scalability refers to increasing the number of nodes, vertical scalability refers to adding additional resources to a node (Li *et al.*, 2021). Vertical scalability can be for example increasing the number of CPUs (Central Processing Units). One enabler for implementing good vertical scalability is concurrency, which in this case stands for multi-threading. However, concurrency should not be sprinkled all over the place, but need to consider reasonable points of use. Implementing concurrent processes in places where they are not needed can, at worst, slow down a program.

One key thing mentioned to achieve horizontal scalability is to implement stateless services. That means each instance of application should not have any kind of internal state, in other words, data should not be internally saved for later use. (Andrawos and Helmich, 2017) The stateless constraint usually means a trade-off in design, as network performance may decrease. When services are fully stateless, the repetitive data sent in requests is usually increased, as data cannot be left on the service as a shared context. (Fielding, 2000) It is usually necessary to have an external storage such as a database service (Andrawos and Helmich, 2017). On the other hand, it is not a good option for example to save transient real-time data to a database. This might have a bad impact on scalability when data loads are big, and database is updated frequently. Therefore, it is necessary to think carefully about what data is stored in a database. Instead of using a database, for example in-memory cache might be a good option for transient data.

In the designed application scalability is handled as an ability to process high data traffic from multiple sources. The data traffic is based on user requests which are made through a web user interface. However, the web application will not have at least very many simultaneous users, so the application does not have to scale well for a large number of simultaneous user requests. In this case scalability is more about handling the varying amount of data traffic and sources.

2.4.2 Performance

In web-based systems events come in a form of user requests via clients. When a system is responding to an event, performance is a measurement of a system's ability to meet timing requirements. It is often linked to scalability, which means the system's modifiability while it still maintains its good performance. (Bass, 2022)

In the designed architecture, the amount of time between a client sending a request and receiving a response must be primarily considered from the performance point of view.

In addition, the frontend component must also be implemented in such a way that usability of the application is good even if some data query to backend servers takes a bit longer.

To achieve sufficient performance in the application this thesis work specifically examines how different real-time communication technologies affect performance. Technologies suitable for the designed architecture are selected partially based on this. In addition to different communication technologies, it is considered how certain backend technologies affect performance.

In the previous section, it was mentioned that in-memory cache can be an alternative for database storage. Using a cache can eliminate some interactions between services, and in that way, it can also improve performance by reducing the average latency of series of interactions. However, it has a trade-off as a cache can decrease reliability of service. The data that would be obtained with a new request directly from a server may differ significantly from the data in the cache. (Fielding, 2000)

2.4.3 Availability

Software availability refers to the application's readiness to carry out its tasks when it is needed. In addition to reliability, it also encompasses recovery aspect. It refers to the way how software is able to repair itself after system breaks. (Bass, 2022) Most of all software availability is the percentage of time during which the software is operational and able to receive requests.

Availability is closely related to other quality attributes, for example security. As an example, there are denial-of-service attacks which are a big security thread in web applications and their purpose is to make a system unavailable. Availability is also related to performance, as system failures may also cause slow responds in addition to unavailability. (Bass, 2022)

It is possible to ensure good availability when designing new software. Tactics to ensure the availability can be divided into three categories. The first category refers to fault detection. Different faults can be detected for example by running ping function for a server or an application to see if it is available. Another category contains fault recovery tactics, like software upgrade or just ignoring faulty behaviours. The third availability tactic category is about preventing faults. One prevention tactic is exception prevention, which is also a requirement for responsible code. (Bass, 2022) Code should check incorrect arguments and handle those gracefully by not throwing an exception.

2.4.4 Maintainability

Usability of code can also be considered from the developer's point of view. In that case it refers to maintainability. Maintainability defines the degree of effectiveness and efficiency with which a product or system can be modified by the maintainers (Bass, 2022). It is important that in the future it will also be easy for anyone other than the original developer of the program to read code files, make changes and further develop it.

One of the requirements for this application is to have code structure and patterns which support the usability of the code. Modularity is the key thing and specifically having reasonable module division which supports future development. Modularity makes it easier to read and modify the code. Moreover, having a single source of specific functions makes it faster to fix the code later. If multiple developers are updating the application, collaboration is easier when there is a clear division between code source files.

When considering maintainability, it would be good if the selected technologies were also relatively easy to use. There are differences between syntax, but there are also differences between library and community support. Of course, usually if technology is a good choice from other architectural viewpoints, it also has good community support.

In addition to modularity and technologies, testability is also important when maintainability is considered as tests verify that the program still works when it has been updated. When the code is easy to test, there will be probably fewer mistakes when modifying the code.

2.4.5 Security

Security is a software quality attribute which measures the system's ability to protect data from unauthorized access while still providing access to authorized parties (Bass, 2022). The CIA (Confidentiality, Integrity and Availability) triad can be used to describe the primary goals of information security (Ingeno, 2018). A balance between these attributes needs to be found also in this web application. Confidentiality stands for preventing unauthorized individuals from accessing application information. To achieve integrity, it needs to be ensured that information is not modified or destroyed by unauthorized parties. Availability involves that authorized parties have access to application information in a timely and reliable way. (Ingeno, 2018) To summarize this, the application must allow data access only for authorized users always when data is needed.

In this application data traffic must be encrypted. One way to encrypt the data is to use TLS (Transport Layer Security). TLS encryption ensures that data transmitted between components and browser is encrypted and cannot be intercepted by a third party. Encryption relies on trusted third-party certificate authorities to issue digital certificates to servers. This helps to ensure that the server is who it claims to be. Implementing TLS encryption also improves compliance with regulatory standards.

Another security requirement is to allow only authenticated users to access API endpoints. Token-based authentication must be used to verify confidential access. Token-based authentication is stateless, and the server does not need to maintain a session state for each user. This makes token-based authentication more scalable.

The application must also be robust. There can be intentional as well as unintentional overloads of traffic or network connections to the application. A denial-of-service attack is an example of an intentional traffic flood or a high connection amount towards an application. It can be avoided by recognizing what traffic is normal and what is not. One technique is to restrict the number of requests that come from a certain domain within a certain period. This may help prevent an attacker from overwhelming the application with too many requests.

3. WEB APPLICATION ARCHITECTURE

This chapter considers the theory of web application architecture. More specifically, it describes single-page application architecture, client-server and publish-subscribe communication patterns, API designs and real-time communication technologies. Moreover, frontend architecture items are considered, such as React principles, client-side data storage and ways to optimize performance on the frontend side.

3.1 Single-Page Application

In the early days of the web the focus of web development was on generating and serving individual HTML files, which formed complete web pages. This could mean static HTML pages or rendering HTML pages on a backend server using server-side logic. It was far from how native applications worked. When a web page was opened there were always additional calls to a backend server which were necessary to make, and a page refresh was required when data was updated. In a native application there were only calls needed for sending or getting data. Before SPA (Single-Page Application) design model, the emphasis was on sending complete HTML files from server to client. (Choi, 2020) However, server-side rendering is still widely used in web development as server-side rendered pages load fast also with worse internet connection.

SPA design is a relatively new architectural pattern that has been widely embraced in web application development. It can be even said that using SPA design is a common practice for building a large-scale web application. Before SPA design AJAX technology introduced a way to update only some parts of a web page without the need to reload the whole page. SPA architecture is largely used in client-side rendered web applications, but it is also possible to utilize SPA in a server-side rendered application. One reason for using SPA pattern is to develop a web application that feels more like a native desktop or mobile application. The program responds and looks like it was installed on the device. The application developed using SPA design redraws portions of the screen immediately without waiting for a new file from the backend server. (Choi, 2020)

The naming of Single-Page Application comes from the fact that the entire application lives on one HTML page only. It means the first page is the only page that ever loads on a SPA. Previously it has been common that all the scripts and files needed to run the application have loaded at the beginning of use. But this has been changing lately. (Choi,

2020) Techniques that can be used to implement late loading for scripts and files are discussed in the later paragraph.

One of the key things in SPA design is virtual routing, which means routing that only happens in the client browser. When an application is making a logical transition to a different screen, the browser does not make any calls to servers for changing the URL (Uniform Resource Locator) address. There is no real server path indicated by the URL when routing is happening in an application developed according to SPA design. Instead of indicating the server path the application uses the URL as a kind of container. The URL contains sections of the application and triggers certain behaviours when different URLs are given. Even though the URL is only a sort of container, it is still useful to have URL routing as it for example allows users to bookmark different screens. (Choi, 2020)

3.2 Communication patterns

In the history of the web one typical architectural style used has been client-server architecture. One commonly used communication pattern in client-server architecture is a request-response pattern, where the client sends a request and the server answers with a response. This is probably the most known communication pattern used in web applications. Another commonly known pattern is a publish-subscribe pattern which is also discussed in this section.

3.2.1 Client-server architecture

In the client-server architecture clients and servers have their own distinct responsibilities. Those can be implemented and deployed independently and using any language or technology. (Massé, 2011) The client is the component which makes various service requests to the server, whereas the server is the component which provides services to the clients as per requests placed by it. There is no limitation to the number of clients that can be serviced by a single server. Both client and server can reside on the same system or in separate systems, and communication is handled using a request-response pattern. (Raj, 2017)

As the client is unaware of server details in client-server architecture, maintenance is relatively easy. Server maintenance activities like upgrade and repair do not affect the functioning of the client. Also, access to data is centralized and most of the data is stored centrally, which makes data updates easier. Data centralization also offers greater control and higher level of security. (Raj, 2017)

The traditional client-server-architecture is two-tier architecture which consists of two layers. One layer is running on the client which is responsible for presenting the data. Another layer is responsible for storing the data on the server-side. There are some limitations in two-tier client-server architecture, like limited extensibility and scalability as application data and business logic usually reside on the same server. To overcome limitations in two-tier architecture, three-tier and multi-tier client-server architectures were developed. Most of the client-server architectures today are developed based on the three-tier or multi-tier models. The three-tier client-server architecture has a separate application layer between the presentation layer and the data layer. The data layer does not contain application logic, and it is usually a database server, which has a positive effect on scalability. (Raj, 2017)

3.2.2 Publish-subscribe pattern

Publish-subscribe model provides an efficient communication pattern for loosely coupled systems. The main entities in publish-subscribe pattern are the publishers and the subscribers of the data. In the general model of event notification, the subscribers define their interests by using topics, channel names, or filters. (Tarkoma, 2012) Publishers produce information in form of events, which are then consumed by subscribers issuing subscriptions. Unlike in the request-response pattern subscribers are not directly targeted by the publisher. Targeting is implemented indirectly according to the type of events. (Baldoni *et al.*, 2009) Figure 4 presents the example use case of publish-subscribe pattern.

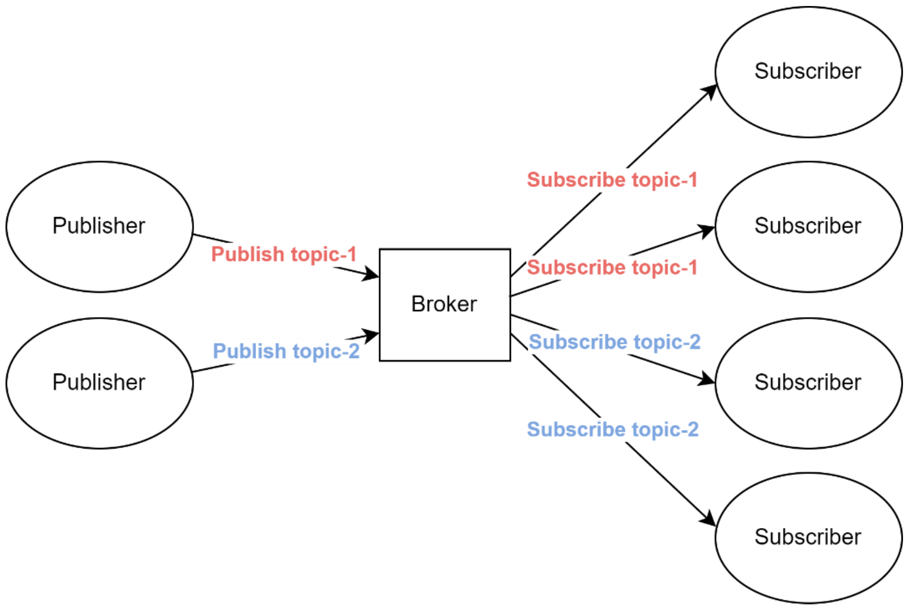

Figure 4. *Publish-subscribe pattern*

The core mechanism in publish-subscribe pattern is event routing, which is a process of delivering an event to the subscribers of the event (Baldoni *et al.*, 2009). The entity which is handling the event routing process is usually a broker. The broker is responsible for the data distribution, and it does not store the data, but scalability may be an issue also in the event routing process. Performance should not significantly decrease when the number of brokers, subscriptions and publications increases. (Baldoni *et al.*, 2009) On the other hand, publish-subscribe pattern may provide better scalability than commonly used request-response pattern thanks to parallel functioning of message delivery.

The publish-subscribe pattern may be a good option for systems with heavy traffic between multiple publishers and subscribers. The publish-subscribe model can mitigate the number of connections between these entities. Every service in the system only needs to connect to the broker service. In addition, the traffic load decreases as the data can be sent only when it is updated, and no unnecessary polling is needed.

Publishers and subscribers are decoupled in the publish-subscribe model. There is no dependency between these entities. However, if no subscriber is subscribing the topic which is published by a certain publisher, it may not make sense for a publisher to publish messages on that topic. Pausing the publisher process could save resources in this case. This, of course, depends on the use case. In case of a network disconnect, old messages from broker's cache memory might be needed, even if the subscribers have disappeared for a while.

3.3 Web server API

It is common that the server implements a specific API (Application Programming Interface) through which the clients can communicate with the server. An API exposes a set of data and functions to facilitate interactions between services and allow them to exchange information. When specifying API multiple things can be considered. Maybe the most known architectural concept for API implementation is REST (Representational State Transfer). This work also considers another way to implement API called GraphQL. It is an architectural API concept like REST, but GraphQL is also considered a query language. Even though these API styles are compared in this thesis they are not excluding each other, in fact, they can also be used as a combination. In addition to REST and GraphQL there are also other popular technologies to use in API implementation. One of these is gRPC (Google Remote Procedure Call), which is a new version of RPC (Remote Procedure Call). As a data format gRPC uses a concept of protobufs instead of JSON (JavaScript Object Notation). (Kornienko, Mishina and Melnikov, 2021) gRPC was also considered to be used in this application as it offers high-performance data transmission. Since the real-time data received by the backend server is JSON it was thought that it would not be worth converting it to protobuf format. Using JSON with gRPC may require additional code and configuration to handle serialization and deserialization of JSON data. Thus, only REST and GraphQL API implementations are considered in this work as JSON format is preferred to be used.

3.3.1 REST

The REST architectural style is commonly applied to the design of APIs. It is an abstraction of the architectural elements within a distributed hypermedia system (Fielding, 2000). If the API is implemented according to the REST architecture style, it is called the REST API (Massé, 2011). REST APIs are well-suited for client-server architectures as they are designed to work over HTTP (HyperText Transfer Protocol), which is the standard protocol used for communication between web clients and servers. REST APIs are based on a set of principles for building web services, including a uniform interface, statelessness, and the use of HTTP methods to represent different operations.

REST architecture has certain rules that the API must follow in order to be called REST API (Massé, 2011). The central feature of REST architectural style is an emphasis on a uniform interface. With the uniform interface details of component implementation can be ignored. To obtain the uniform interface, four interface constraints are needed for guiding the components' behaviour. These constraints include identification of resources, resource manipulation through representations, self-descriptive messages and hypermedia as the application state engine. (Fielding, 2000) Resource identification is done by using the URI (Uniform Resource Identifier) sequence. In REST architecture resources are not directly edited but they are manipulated through representations in requests. As messages are self-descriptive each message includes all the needed things to handle the request event. Hypermedia plays the role as an application state engine as the client and the server share state information via hypermedia, like body content of a request or response headers.

The uniform interface improves the visibility of interactions and simplifies the overall architecture of a system. However, as a trade-off it may decrease efficiency of the program since data is transferred in a standardized form and not in a form which might be more suitable to an application's needs. (Fielding, 2000) There are many applications which are using some REST architecture constraints, but which are not implementing the whole uniform interface according to the specification.

Most of all REST architecture emphasizes scalability of component interactions. This means that each request contains all the necessary information to understand the request. Every request is independent of any other requests. Thus, the server does not need to retain the client's state between requests. The stateless nature of REST improves scalability of the application, and it also may have a positive impact on the consumption of physical resources. (Fielding, 2000) It is still good to remember, that in some cases statelessness may increase resource consumption. For example, network traffic may increase if a high amount of client-specific information is included in each request.

The original REST architecture specification does not propose how to use HTTP request methods, but those methods have commonly adopted use cases in APIs implemented according to REST architecture. Each HTTP request method has usually specific and well-defined semantics within the context of a REST API's resource model. Commonly used HTTP request methods are GET, DELETE, PUT and POST:

1. **GET:** Get the representation of a resource's state
2. **DELETE:** Remove a resource
3. **PUT:** Add a new resource to a store or update a resource
4. **POST:** Create a new resource within a collection and execute controllers.

All the other HTTP methods mentioned may contain body except the GET method. (Massé, 2011)

HTTP methods provide a standardized way of performing operations on resources, making it easier for developers to understand and use APIs that follow the REST architectural style. Some HTTP methods, such as GET and DELETE, are idempotent, meaning that they can be safely repeated without causing any side effects. This makes it easier for developers to build reliable and robust applications with REST API.

3.3.2 GraphQL

GraphQL is a query language and an execution engine originally created by Facebook in 2012 (GraphQL, 2021). GraphQL is suitable to be used in client-server architecture as it provides a flexible and efficient way for clients to communicate with a server. Clients can send queries to the server using GraphQL and receive a response that includes only the data they requested. This can help to reduce the amount of network traffic and improve performance of the application. GraphQL query language allows clients to specify exactly what data they need and receive only that data, rather than retrieving a fixed set of data as is the case with REST APIs.

There are five design principles provided by GraphQL. The first principle is product-centric design as GraphQL is driven by the requirements of frontend views. GraphQL requests are structured hierarchically, and structure remains the same also in responses. Hierarchical design is another principle in GraphQL language. The third design principle is strong-typing, which means requests are executed in application-specific context defined by GraphQL service. Syntax and validity of operations are always ensured before execution. The fourth design principle is introspective implementation of services. GraphQL schemas can by queryable by the GraphQL language itself, which makes GraphQL APIs introspective. One of the main things that differs GraphQL APIs from common REST APIs is the principle of client-specified response. In GraphQL API the client is responsible for specifying how it consumes published capabilities, whereas the service determines the structure of data returned in the majority of client-server applications developed without GraphQL. (GraphQL, 2021) When a response is specified by clients the data usually contains only the needed information, and unnecessary information is left out of the response. This may also have a positive impact on the number of requests, as the client is able to do tailored and combined requests which might not be reasonable to implement in server-specified APIs.

GraphQL defines data structures as schemas and provides collective type system capabilities. Schemas contain definitions of supported types, directives, and root operations. Type is a fundamental unit of GraphQL schema. Directives on the other hand describe alternate runtime execution and type validation behaviour. (GraphQL, 2021) GraphQL schemas ensure that the data sent and received by clients and servers are of the correct type.

There are three different root operation types for interaction with GraphQL. These types have their own use cases:

1. **Query:** The query operation is used to retrieve data from the schema. It is the primary way that clients use to interact with a GraphQL API.
2. **Mutation:** The mutation operation is used to modify data in the schema. It allows clients to perform create, update, or delete operations on the data.
3. **Subscription:** The subscription operation is used to listen to changes in the data in real-time. It allows clients to receive updates as soon as they occur, rather than having to poll the server for updates.

The query root operation type is mandatory, and it must be provided for every schema. The mutation and subscription are optional root operation types, and if they are not provided the service does not support mutations or subscriptions. (GraphQL, 2021) In contrast to HTTP request methods that are usually used in REST API, GraphQL root operations provide a flexible way to retrieve and modify data, allowing clients to specify exactly what data they need and receive only that data.

3.4 Real-time communication technologies

Many things are called real-time even if timing of those things may vary a lot. Real-time means that system responds to events at the same time or in a very short time after events occur. In the context of the designed application, responds happen after short and configurable time after events occur. This kind of real-time system can also be called reactive. Reactivity refers to the behaviour of the system, but it does not take a position on the time the response takes place. Reactivity is ability of a system to respond to changes in its environment. In this section we go through communication technologies that enable sending data between the client and the server in a short time after events occur. Latency may differ between the methods, and the application requires latency to be inside configured time interval between data updates.

In web architecture there are multiple different techniques to retrieve data from a server to a client. This chapter introduces different methods for real-time communication between those entities. There is always some latency in getting the data to a user interface, but it needs to be considered which kind of implementation works best in this application. HTTP polling, WebSockets and server-sent events are common methods used for real-time communication, and they can be used as server-side push technologies. HTTP/2 server push method can also be used as a server-side push technology. It is similar to server-sent events but with HTTP/2 server push the server can send resources pre-emptively based on the initial request of the client (de Souza Soares *et al.*, 2018). However, HTTP/2 server push might not be supported by some primary browsers in the future as it is still quite rarely used. Thus, HTTP/2 server push method is not discussed in this work.

3.4.1 HTTP polling

In communication networks different network protocols set rules for data format and processing. The OSI (Open Systems Interconnection) Reference Model is a common model to describe data communication between systems. (Ibe, 2018) There are multiple different protocols which can be defined in the application layer of the OSI model. One of these protocols is HTTP which is a request/response protocol that defines client, proxy, and server entities (Saint-Andre *et al.*, 2011).

HTTP is a protocol for distributed, collaborative, hypermedia information systems. It is used with a client-server mode, and it is underlying protocol used in web applications. The HTTP protocol defines message formats and how those are transmitted. It also defines which actions clients and servers should take in response to various commands. (Ibe, 2018) In the standard HTTP model, the client needs to poll the server periodically for getting new data. In this case a server cannot initiate a connection with a client nor send an unrequested HTTP response to a client. In the standard model the server cannot push asynchronous events to clients. (Saint-Andre *et al.*, 2011) However, there are also ways to implement event pushing from the server-side. These ways are discussed in subsequent sections.

As stated, polling is a standard method for retrieving data from server to client. With the traditional polling technique, also called short polling, the client sends regular requests to the server. Requests attempt to get any available events or data, and an empty response is returned if there are no events or data available. This communication mechanism consumes resources such as server processing and network. By forcing a request-response round trip when no data is available, continuous polling can consume significant bandwidth. In some cases, it can also reduce the application responsiveness since data is queued until the server receives the next request from the client. (Saint-Andre *et al.*, 2011)

However, there is also a long polling method, which is a common server-push mechanism. The client starts a long polling process by making an initial request and then leaves waiting for a response. The server refers a response until a new update is available or until a certain status or timeout has occurred. The server sends a response to the client when an update is available. A new request is sent by the client either immediately upon receiving a response or after a while to allow an acceptable latency period. (Saint-Andre *et al.*, 2011) The long polling communication scheme is described in Figure 5.

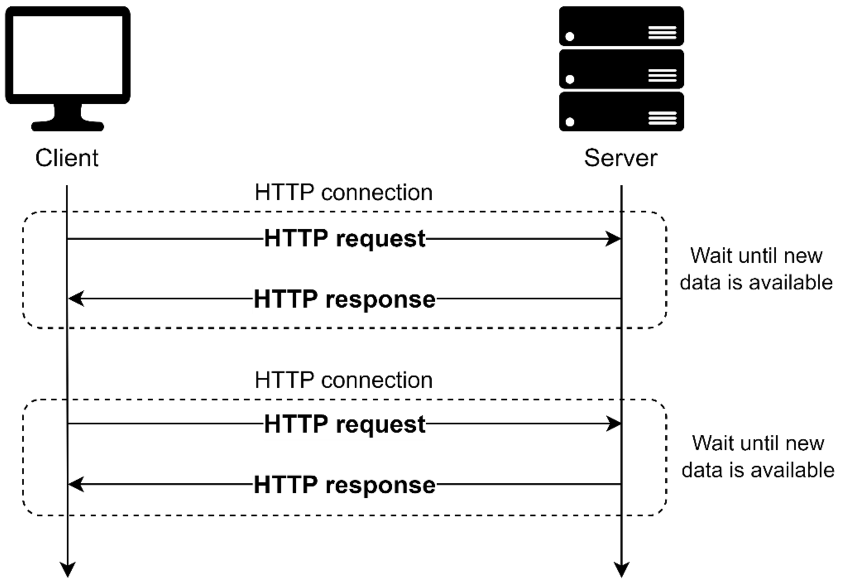

Figure 5. *The long polling communication scheme*

In contrast to short polling, in long polling the server attempts to hold HTTP requests open. There is always a pending request from the client, and the server responds only when there are events to deliver. This way the latency in message delivery can be minimized as the server can reply when certain events occur. (Saint-Andre *et al.*, 2011)

There are a few known issues with HTTP polling techniques. One issue is that these techniques might frequently open and close TCP/IP (Transmission Control Protocol / Internet Protocol) connections. However, polling method works well with persistent HTTP connections that can be reused for many poll requests. For example, a short pause between a long poll response and the following long poll request avoids the closing of idle connections. (Saint-Andre *et al.*, 2011)

HTTP polling is founded on the request-response messaging pattern. It is also possible to create the publish-subscribe pattern using HTTP polling, but network traffic and latency may increase as clients are needed to start every event with a request. HTTP polling consists of repeatedly sent update requests, which can also result in out-of-order responses. It can happen for example in case multiple updates are received by the server at the same time.

Scalability comes to an issue when the state of the continuously polling client must be kept in memory. Since there is no permanent connection, the starting point of the polling process must be somehow indicated. Detecting the ending point of the process also worsens scalability. When the client suddenly drops out, the server can notice it only by keeping track of how much time has passed since the latest poll request.

3.4.2 WebSockets

WebSocket protocol was developed to replace HTTP technologies such as short polling and long polling, and it is an abstraction on the application layer of the TCP socket (Hribernik and Kos, 2020). Like HTTP, it also uses the TCP protocol in the transport layer of the OSI model. However, the transmission delays are reduced when using WebSocket due to the combination of one connection and the appropriate optimizations. (Karla and Tarnawski, 2019) WebSocket can be used for bidirectional communication between client and server over a single TCP connection by using a “so-called” socket (Kulshrestha, 2013; Karla and Tarnawski, 2019). However, the socket used in WebSocket technology is not the same as the TCP socket. WebSocket technology is fully supported by all primary browsers (*The WebSocket API (WebSockets)*, 2022).

A WebSocket connection is established via HTTP protocol. Once a connection is established, both connected services can send and receive data independently and without any agreements. (Hribernik and Kos, 2020) The WebSocket communication scheme is described in Figure 6.

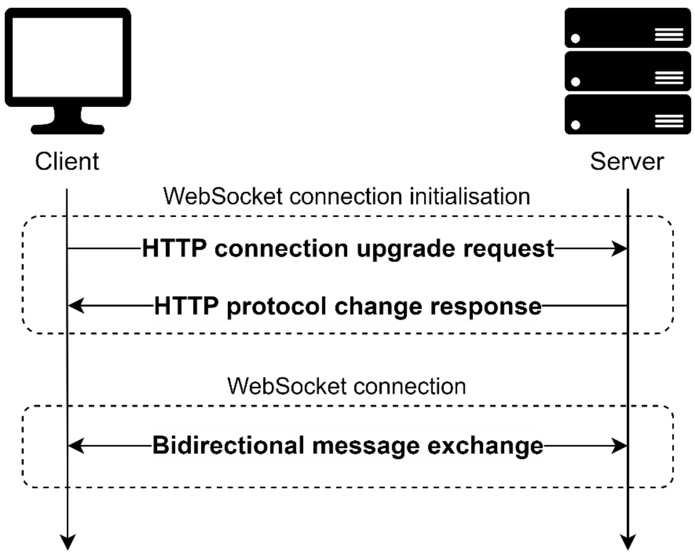
Client
Server
WebSocket connection initialisation
HTTP connection upgrade request
HTTP protocol change response
WebSocket connection
Bidirectional message exchange

Figure 6. *The WebSocket communication scheme*

First, to establish a connection with WebSocket the client sends an HTTP GET request. There is HTTP Connection Upgrade located in the header of the HTTP message. The server receives the HTTP Connection Upgrade request and triggers protocol change via the HTTP response if it supports WebSocket and can communicate with the client via WebSocket protocol. After the response the HTTP connection will be closed and replaced by the WebSocket connection over the same TCP connection. The server and the client can begin communicating via WebSocket once they have completed the first handshake using HTTP. (Hribernik and Kos, 2020)

HTTP protocol allows WebSocket to run via proxies and on top of TLS. HTTP and WebSocket protocols do not provide security themselves, but they can be used with TLS protocol to establish a secure connection. (Hribernik and Kos, 2020) WebSocket itself does not provide a solution either for denial-of-service attacks. Defence against a denial-of-service attack can be implemented by verifying the origin of incoming requests (Wang, Salim and Moskovits, 2013). The Origin header is sent during the WebSocket handshake along an HTTP request, but it cannot be relied on as the attacker is able to modify the header.

WebSocket technology is also vulnerable to Cross-Site WebSocket hijacking which can happen if WebSocket handshake relies only on HTTP cookies. It is similar to Cross-Site Request Forgery, but in Cross-Site WebSocket hijacking vulnerability a complete two-way communication channel is established and it is not restricted by the same-origin policy. However, protection against Cross-Site WebSocket hijacking can be implemented by using individual random tokens in the WebSocket handshake. (Mei and Long, 2020)

3.4.3 Server-sent events

Where HTTP polling and Websocket technologies provide bidirectional communication between the client and the server, server-sent events are developed for unidirectional communication. Server-sent events operate using HTTP protocol in the application layer. In contrast to traditional HTTP polling server-sent events use a persistent client-server connection, which is initiated using HTTP. (de Souza Soares *et al.*, 2018)

It is relatively easy to create and parse events as they are text-based (de Souza Soares *et al.*, 2018). On the other hand, server-sent event technology is based on HTTP streaming technique, which makes it possible to send several application messages within a single HTTP response. In that case the separation into application messages need to be performed by the application (Saint-Andre *et al.*, 2011).

The server-sent event process starts when the client sends an HTTP request to register an EventSource HTML (Hypertext Markup Language) element. The EventSource is a client-side object used to follow the event process. When the permanent HTTP connection is initialized with a response from the server the requested resources can be returned through the EventSource element. (de Souza Soares *et al.*, 2018) The server-sent event communication scheme is described in Figure 7.

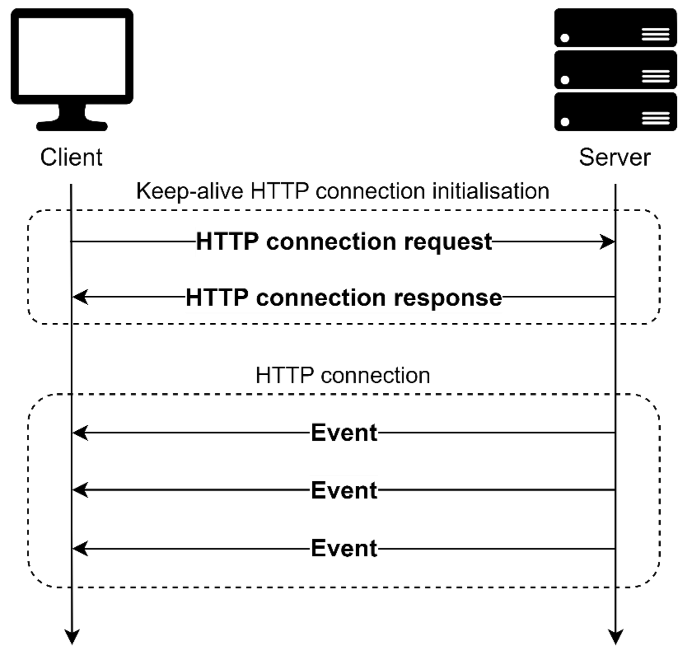

Figure 7. *The server-sent event communication scheme*

Server-sent events support the publish-subscribe paradigm. Clients subscribe to a channel and receive published messages from the server in real time without a separate request. (de Souza Soares *et al.*, 2018) It is not mandatory to have a broker or a message bus when implementing server-sent events, but it has a positive impact on scalability. The server can push messages to multiple subscribers at once if a broker or a message bus is used.

EventSource is compatible with all primary browsers. But it is not supported by Internet Explorer. There is also a limitation when server-sent events are not implemented over HTTP/2. In that case the maximum number of open connections is six per browser. (*Using server-sent events*, 2022) In practise, it means that no more than six tabs can be open at once.

3.5 Frontend architecture

On the client architecture side one of the major technologies is React, which is used for example in the development of Facebook and Netflix. React is a JavaScript framework which can be used for building user interfaces. (Thomas, 2018) There are also other good options to be used as a frontend framework, for example Vue.js and Svelte. However, in this work React is selected as a client-side framework since it provides support for the necessary UI (User Interface) component libraries.

3.5.1 Basic principles with React library

React has multiple features which support application development according to the SPA architectural pattern. Firstly, React has its own virtual DOM (Document Object Model). DOM is a logical tree that represents all the HTML elements according to their order and relative to other elements in the structure. Virtual DOM maintained by React is distinct from the browser's DOM, and it is created and updated based on a reconciliation process by React service. The reconciliation process, also called as the render phase, is a comparison where React looks at the browser's DOM and contrasts that with its own virtual DOM. If there are differences found, React will send instructions to the browser's DOM for modifying elements. (Choi, 2020).

Another main attribute of React is that it is state-driven. React application is made up of components, and each component may have some local state. The reconciliation process is triggered if the state of some component changes. In that way the DOM tree of the browser will be updated. (Choi, 2020)

It can be said that routing is ubiquitous in web development. React has also own Router framework for managing client-side URL routing. Router allows SPA application to behave as a classic web application by indicating the URL address to users. Router in React acts as a parent component and screen rendering components are its children components. (Choi, 2020)

3.5.2 Client-side data storage

There are multiple ways to manage and store application state, in other words data, in the client-side application. If a change in the application state should trigger a reconciliation process, there are two ways to handle the state. Either using a component-based state or Redux store. According to Choi (2020) Redux is a relatively new framework, which creates a common location, the Redux store, for storing the application state. React has also its own Redux library which is designed to work well with React. It has been the most popular enterprise-level framework for managing the global state in a React application. (Choi, 2020)

Component-based nature of React has its advantages, but a component-based state can be limiting. The state might not be always specific to a component or even to a component hierarchy. In React, the state is passed down only one way, from the parent component to the child components as props. (Choi, 2020) Redux provides a way to share and modify the application state globally, by storing all of the application state data in a single location called Store (Chinnathambi, 2018).

The process of updating Redux Store is described in Figure 8. Adding a new state or updating an existing state in the Store is done by using actions. Actions are objects which provide the data for specific reducers and describe what to change. (Choi, 2020) As a result of a given action, the reducer determines what the final state will be (Chinnathambi, 2018).

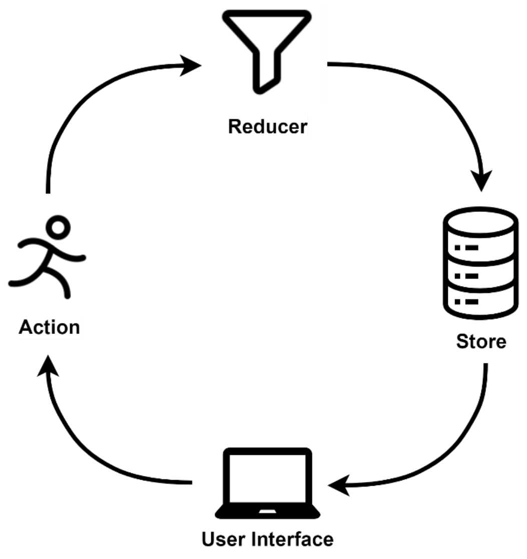

Figure 8. *The process of updating Redux store*

It might seem that there are unnecessary steps in Redux Store update process, but all these steps ensure scalability of the application. Using Redux Store makes it easy to maintain the application state as it is stored in a single location. By using actions, Redux ensures that wrong parts of the application cannot modify the application state. Reducers specify what the final state will be so the state is never directly modified or mutated. (Chinnathambi, 2018)

Redux ensures good maintainability of the code with centralized state management. It is easier to maintain the code when the state is always modified in the same way using actions and reducers. When the logic for managing state is separated from the logic for rendering components, code is probably easier to understand and debug. Redux also makes it easier to reuse components across an application or between different applications.

3.5.3 Performance optimization

In the application designed according to SPA architecture, the first page is the only page that ever loads (Choi, 2020). Despite of that it is not necessary to load all scripts and files needed to run the application at the same time. Lazy loading can be used to increase client-side application performance. According to Elrom (2021) it is one of the easiest ways to increase performance. By using the lazy method for importing the component it can be ensured that the component loads only once it is used. There is no clear rule about when the lazy method should be used. It is up to the use case what is the best way to implement component imports. (Elrom, 2021)

Another thing to consider when optimizing frontend performance is the way how big lists of data are rendered. Usually, these kinds of lists are viewed with scrolling, and it is not necessary to render all the information at once. When there are too many items rendered as a list scrolling is sluggish. To optimize performance, it is a good choice to render only a subset of the list. One option is to use pagination when rendering a subset of the data at a time. Only the data that is visible on a current page is rendered. In addition, there are libraries which implement rendering subsets while a user is scrolling. When the page is initially loaded, only a subset renders. And later when the user scrolls down the list more subsets are rendered.

4. ARCHITECTURE AND TECHNOLOGY ALTERNATIVES

In this chapter different architectural solutions are presented and compared. First, the component structure and division of responsibilities between components is designed. There are three possible solutions from which the best will be chosen. Secondly, it is considered which technology best enables good scalability in the backend component. Finally, communication between the backend server and clients is considered. Different communication solutions contain the backend component's API implementation and real-time communication technology used between the backend and frontend components.

4.1 Component division

One of the main things to keep in mind when designing software components is that the responsibilities for one component are reasonable. Component division must support a clear division of responsibilities. Three different alternatives for the component division were observed in this section. The first alternative is the simplest one, but on the other hand it might not be the best solution. The main thing considered is that good performance and scalability are supported.

4.1.1 First alternative

Once there is a clear understanding of the system requirements, it is possible to identify the different components that will make up the system. Each component should have a well-defined responsibility. When a component is responsible for a specific task, it can be updated later without affecting other parts of the system. It may also affect scalability as each component can be duplicated more easily as needed to handle increased load. On the other hand, when the number of components in a system increases, so does the complexity of the system. In addition, a higher amount of components may also increase maintenance and testing efforts when the most benefits of component division are achieved.

The first alternative for component division is presented in Figure 9. There are only data collector components located on edge nodes in the first alternative. Data collectors provide measurement counters in binary format. The frontend component located on central node receives binary measurement counters from one or multiple data collectors, which creates a huge traffic load on the frontend component. The frontend component is required to be developed using React and JavaScript, but for example Go or C++ may be better choices for CPU-intensive tasks, like processing binary measurement counters as such. According to Aalto (2022) one measurement counter is typically 32 bits. In a typical use case 496 RTPM (Real-Time Performance Monitoring) counters are received from one eNB (eNodeB) per second. In the case of gNB (gNodeB) the same number of counters is 177. The number of analysed base stations varies depending on the use case. However, in the case where the amount of eNBs is 800 and the amount of gNBs is 200, the estimated incoming RTPM report stream is about 123 Mbit/s. (Aalto, 2022)

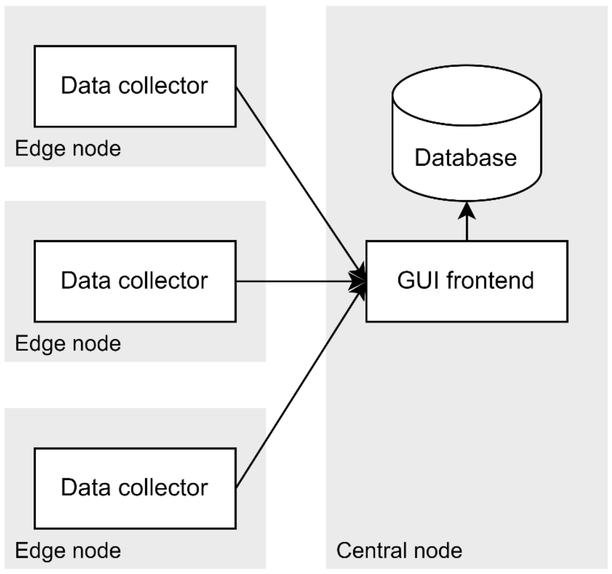

Figure 9. *The first alternative for component structure*

Overall, there are a lot of data elements to be monitored to analyse the state of a mobile network system. RTPM is giving meaningful info for most of the cases, and it is still relatively easy to handle. However, in applications that are analysing mobile networks bigger data volumes are also needed to take care of. Data amount will increase if going to on more detailed level, for example to layer 3 level radio protocol events. In this study the focus is more on RTPM.

Besides receiving and analysing measurement counters, the frontend component would also be required to handle user interface events and interact with the database. The performance of a frontend component can directly impact the success of a web application. If the frontend is slow or unresponsive, users may become frustrated and stop using the application.

Generally, it is a good idea to keep components small and focused, with a single and well-defined responsibility. In this solution the so-called frontend component is having too many responsibilities. First of all many different responsibilities make the component very complex. The component is difficult for developers to understand, which can make it hard to extend and maintain. Moreover, a component with many responsibilities may become large and unwieldy, which makes it difficult to work with and can also negatively impact performance. Many different responsibilities may lead the component to be tightly coupled to other parts of the system, which can make it harder to reuse or modify the component independently.

4.1.2 Second alternative

The second alternative for component division is taking more advantage of distributed computing. The solution is described in Figure 10. In this alternative, a new Parser component has been added to edge nodes. It forms parametric distributions from binary measurement counters and thus reduces the traffic load to the central node. Formed KPI distributions contain all the needed information for further analysis. Cause-symptom relations based on KPI distributions are utilized later for root-cause analysis.

One way to identify problems in a mobile network system is to represent the relationship between KPIs and fault causes. The cause-symptom relations can be modelled through parametric distributions. One possible distribution is Gaussian, which can use for example average and standard deviation as parameters to define the model. KPI can be analysed for each fault cause by comparing its behaviour with the distribution describing the normal behaviour. (Gómez-Andrades *et al.*, 2016) It was planned to take advantage of cause-symptom distributions also in this use case. As it requires very high computing capabilities to form distributions, it is a good choice to form those on edge nodes. In addition, handling measurement counter data in binary format and creating KPI distributions based on the data is a very heavy computing task, so that functionality is good to locate to own component. It improves a clear responsibility division between components.

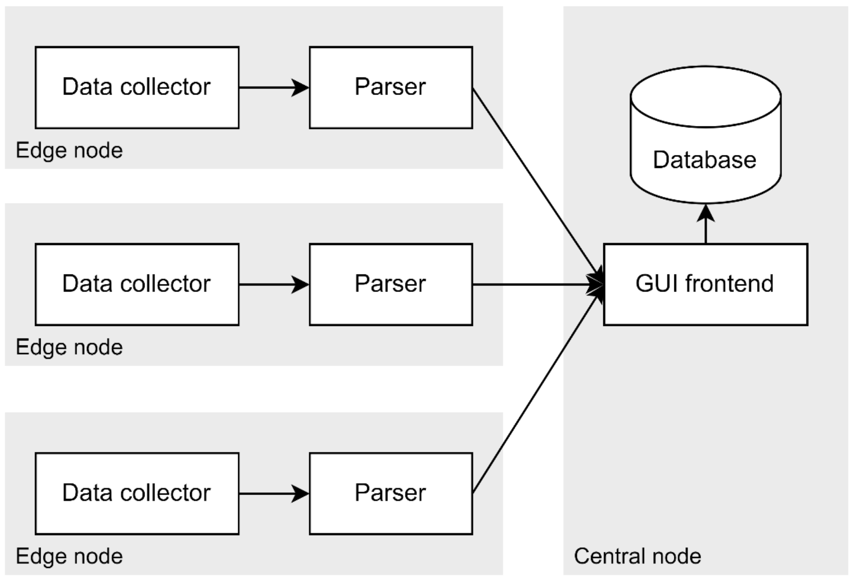

Figure 10. *The second alternative for component structure*

In this solution GUI frontend component receives KPI distributions instead of binary measurement counters. The incoming traffic has decreased as KPI distributions provide information in a more compact form and the amount of irrelevant data has reduced. However, there is still quite a lot of information for a frontend component to receive. Also, KPI distributions need to be analysed more for providing the information to a user in an easily understandable format. So, all the received KPI data is not needed in the frontend for providing needed information in the user interface. Transferring a large amount of data over the network can slow down the page load time and result in a slower user experience. In addition to the data transfer browser storage has also limited resources. Storing a large amount of data in the browser's memory can consume a significant amount of system resources, leading to slower performance and an increased risk of crashing.

This component division alternative also indicates that the frontend component has too many responsibilities and so it is too complex. Also with more data, the complexity of the frontend component increases, which can lead to increased difficulty in maintaining and updating the code.

4.1.3 Third alternative

The third component division alternative in Figure 11 has a new backend component added to the central node where the web application logic and database interactions are located. In this case the frontend component gets only high-level information about the network status. It just subscribes statistics from certain KPI topics, but it does not have to know in detail which KPIs to subscribe to. Of course, it is possible to inspect and modify the KPIs through the user interface, but the frontend component is not required to mention KPIs when it is requesting data updates for example about network throughput topic.

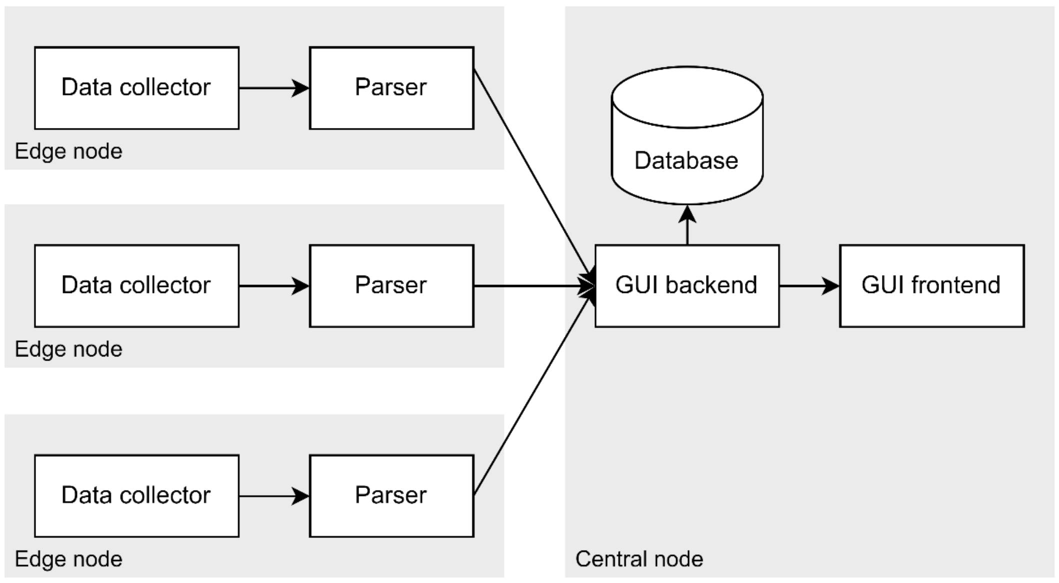

Figure 11. *The third alternative for component structure*

GUI backend component contains the basic web application logic, that is handling the API requests that come from the frontend component and communicating with the database. It is better to have a backend component to communicate with a database rather than communicating with the database directly from the frontend as the backend component provides an abstraction layer between the frontend and the database. Database schema and implementation are hidden from the frontend, and it is easier to make changes to the database schema without affecting the frontend component. Also separating the frontend and backend components improves the modularity of the application.

GUI backend component also communicates with Parser instances. Basically, the GUI backend component is aware of the KPI data at a more detailed level. When it sends requests to get the KPI data from Parser instances, it must also include the KPI formula information in the request body. It receives KPI distributions and analyses those a bit further before sending more informative statistics to the GUI frontend component.

4.2 Scalable backend technology

Two different technologies were considered to use in the implementation of the web application backend server. Go and Node.js are compared as they are commonly used in other services in the system environment. Whereas Go is a statically typed programming language, Node.js is a runtime environment for code written in JavaScript language, which is dynamically typed. It makes these alternatives quite different, but both are good options for the implementation of a scalable backend service.

4.2.1 Go

Go, also known as Golang, is a relatively new technology developed by Google and it is currently used by numerous of companies (Andrawos and Helmich, 2017). It has multiple features which make it a great choice to be used as a backend technology. It was developed as an attempt to combine good features from different statically and dynamically typed languages. Go combines the ease of dynamically typed programming languages with the efficiency and security of statically typed languages. (Golang, 2022a) Go was developed specifically with its performance in mind, making it a good competitor to the C language family. However, it supports also a relatively simple syntax similar to dynamic languages like JavaScript. (Andrawos and Helmich, 2017)

As Go has been developed with simplicity in mind, it also has an easy learning curve. There are the same practices for maintainable code in Go as there are in other languages. It is important to keep function and module sizes reasonable and have meaningful names. However, Go’s standard library offers a great library for code formatting. The greatest thing in the library is that its format is uncontroversial and the code format does not change. It is easier to read and maintain code when it all looks the same.

As Go is still a relatively new language, there are not so many third-party libraries available for Go. This may cause issues, but as Go has been quickly increasing its popularity the number of actively maintained libraries should also increase. Go’s standard library also offers a great number of features.

Go supports building applications which take advantage of concurrency and parallelism. Concurrency is implemented in the language as a native feature, and it can be exploited by using goroutines, which is more lightweight than using threads. Native support makes it easier to implement concurrency. With goroutines the application can execute multiple tasks concurrently. Goroutine is a logical execution unit that contains the instructions for a program to run. Execution of a goroutine is handled by a processor, which has own scheduler and queue that consists of goroutines. (Anurag, 2018)

Uni-directional communication between goroutines is handled with channels. Channels can be buffered or unbuffered. If a channel is buffered, the capacity of the channel buffer can limit the data amount waiting in the channel. The sender blocks and waits for the receiver to receive the data only if the buffer is full. In unbuffered channels the sender always blocks until the receiver has received the value. (Golang, 2022b) Channels can be used for message passing from one goroutine to another, but channels can also be used for message broadcasting when there are multiple listeners for a channel.

Even if goroutines are easy to create with Go, jammed connections may consume a huge number of resources. Go’s standard library net/http offers timeout implementation which can be used for mitigating hanging connections. Timeouts are not implemented by default, but they can be set when configuring HTTP server. Go’s net/http package also supports HTTP/2 protocol transparently when HTTPS and TLS are used since Go version 1.6. (*http package - net/http - Go Packages*, 2022)

4.2.2 Node.js

Node.js is an ope-source and cross-platform JavaScript runtime environment (Node.js community, 2022). It is commonly used in web development, as it provides great support for different API implementations. Node.js is a great choice for the backend technology of a web application as frontend code is also written in JavaScript. In this case there would be no need to have multiple different languages used in the application.

Node.js runs the V8 JavaScript engine outside the browser, which allows it to have good performance. The V8 is a high-performance JavaScript engine, used for example in Google Chrome to parse and execute JavaScript code. Even though programs developed with Node.js run in a single process and without multiple threads, it provides asynchronous primitives which prevent code from blocking. When an I/O operation is performed Node.js will resume the operations when the response comes back, and this way any thread execution is not blocked. This way it is possible to handle multiple concurrent connections without managing thread concurrency, which is easily coming a source of bugs. (Node.js community, 2022)

The event-based model of Node.js supports the implementation of good scalability. However, when the application needs to perform concurrent tasks which require high CPU load, good scalability may not be achieved. The lack of parallelism can become an obstacle in that case. The scalability issues can be handled for example with multiple application instances and a load balancer in that case.

There are also built-in HTTP and HTTP/2 modules in Node.js, which provide a great implementation for HTTP features. With HTTP/2 multiple requests are sent over the same TCP connection between a client and a server, which can make communication more efficient and secure. Also timeouts can be configured through the HTTP/2 module to avoid hanging connections (*HTTP/2 | Node.js v19.1.0 Documentation*, 2022). For example, denial-of-service attacks can be mitigated when proper server timeouts are configured.

4.2.3 Conclusion

When considering performance and scalability Go is a better choice than Node.js. The performance of Go is at the same level with C family languages. There is previous research made considering the performance of Go language in comparison to other languages. A study published by BMC Bioinformatics compares Go, C++ and Java for the development of a sequencing tool. The research concludes that Go performs best for the implementation of the tool presented in the study. Go provides the best balance between memory usage and runtime performance. In the study it was observed that Java benchmarks had slightly better runtime performance than Go runs, but Go benchmarks used less memory than both Java and C++ runs. (Costanza, Herzeel and Verachtert, 2019)

Go also supports concurrency, and programs written with Go can take advantage of using multi-thread execution through goroutines. Goroutines make it easy to implement good scalability. Node.js as a single-threaded engine cannot keep up with the performance of Go. Single-threaded architecture also makes it harder to achieve good scalability with Node.js, even if concurrency can be achieved with asynchronous callback functions. Node.js may be more suitable for applications that handle a large number of connections with low throughput requirements.

Maintainability is considered from the viewpoint of development tools and community support. As Go is a newer technology than Node.js and JavaScript, there are fewer libraries and packages available for Go. There is also more community support available for Node.js and JavaScript as they are older technologies. However, the popularity of Go is increasing fast, so it can be assumed that maintenance support is also increasing in the future.

Go was decided to use as language in the backend service as it supports better scalability and performance through concurrency, and those are the most critical requirements for this web application. Concurrency in Go can also be implemented in a very easy way, and it is supported by Go’s standard library.

4.3 Web server API implementation

The performance of applications implemented using REST architecture and GraphQL language is compared in the study published by Lublin University of Technology. The research found that GraphQL application was more efficient when a large number of queries were made to one endpoint to retrieve relatively small data sets. However, when the data set was increased, it was seen that the service with REST interface was more efficient. This was caused by the large size of requests in GraphQL application, as the client had to specify all the information that was needed for responses. When the client is able to define the queries in a way that the data in response does not contain unnecessary information, and the data obtained is smaller in that sense, the GraphQL-based service processes requests faster. The effectiveness of GraphQL can also be seen in situations where multiple requests can be combined into one. (Mikuła and Dzieńkowski, 2020) However, this unnecessary fetching can also be avoided in applications with wisely developed REST architecture. This depends on the use cases of API. REST architecture itself does not declare the implementation of API endpoints. The architecture only offers certain design constraints.

There is also another study considering the performance of services implemented with GraphQL and REST APIs. It states that GraphQL is a good choice when data requirements change often, whereas REST architecture is better when data is frequently retrieved. When considering resource utilization GraphQL is the best choice, but this is affected by the decreased number of endpoints in GraphQL implementation. (Lawi, Panggabean and Yoshida, 2021)

The comparison of REST and GraphQL API technologies is summarized in Table 1. As stated, the performance depends on the use case. If multiple endpoints or data queries can be optimized by the client, the GraphQL API is a great choice. However, in this web application there are no plans for combining endpoints or client-side query optimization. So it might be that using GraphQL would not offer a great advantage to performance in this case.

***Table 1.** Comparison of API technologies*

	REST API	**GraphQL API**
Performance	Good when querying large data sets, but over- and underfetching decreases performance	Good when multiple endpoints are combined or queries can be optimized, complex queries decrease performance
Maintainability	A conventional and well-known standard	A steeper learning curve, but later development might be faster, simplifies complex API
Typing	Weakly typed	Strongly typed

From a maintainability perspective, there is no obvious choice. REST API is a conventional and well-known standard, and most people are familiar with it. On the other hand, GraphQL simplifies complex API implementations and might make development faster. However, GraphQL has a steeper learning curve.

It can be said that REST API is more weakly typed than GraphQL. It is up to the client and server to agree on the format and interpretation of the data being exchanged. Depending on the use cases of the system weak typing can be both an advantage and a disadvantage. Weak typing allows for more flexibility and ease of development. On the other hand weak typing can lead to more errors as there is less strict control over the data being exchanged. This can improve the overall reliability and security of the application, and it is an advantage. A strongly typed API can provide a level of abstraction that allows clients to interact with software components without knowing their implementation details. It also helps to prevent runtime errors and improves the overall stability of the application.

However, API based on REST architecture was decided to use in this web application backend component as the possibility for query optimization by clients was not recognized in this use case. Of course, if it turns out later that client-driven queries could be useful, the change of API technology could be considered.

4.4 Real-time client-server communication

This section considers the evaluation of client-server communication methods for real-time data transfer from the backend server to frontend clients. Evaluated methods are HTTP polling, WebSockets and server-sent events. Evaluation is done both with empirical performance testing and by comparing features and quality attributes of each method.

4.4.1 Empirical research on performance

It is not obvious that there are no bottlenecks in traffic between application components. The data is updated according to a configured time interval, and the interval between data updates in the frontend will not be less than one second. This empirical testing is done to observe the performance of the backend server when certain communication methods are used. When considering the client side, it was noticed that the CPU load and memory load were quite the same with different methods. Moreover, it is not yet known how many endpoints there will be in the backend API, and that would probably affect test results more from the client's point of view. Bigger data amount will be analysed and extracted which requires more effort from the backend component than the client. The amount of statistics data sent to clients will be tried to keep as low as possible.

Empirical research was conducted regarding CPU and memory usage of backend services with different communication technologies. Three simple backend and frontend components were developed using different methods for real-time communication. Frontend components request continuous updates from backend servers either using HTTP polling, WebSocket or server-sent events. Each of these components runs in its own Docker container. Backend servers are developed with Go language, and frontend components are React applications served with Nginx. The system used in empirical testing is presented in Figure 12.

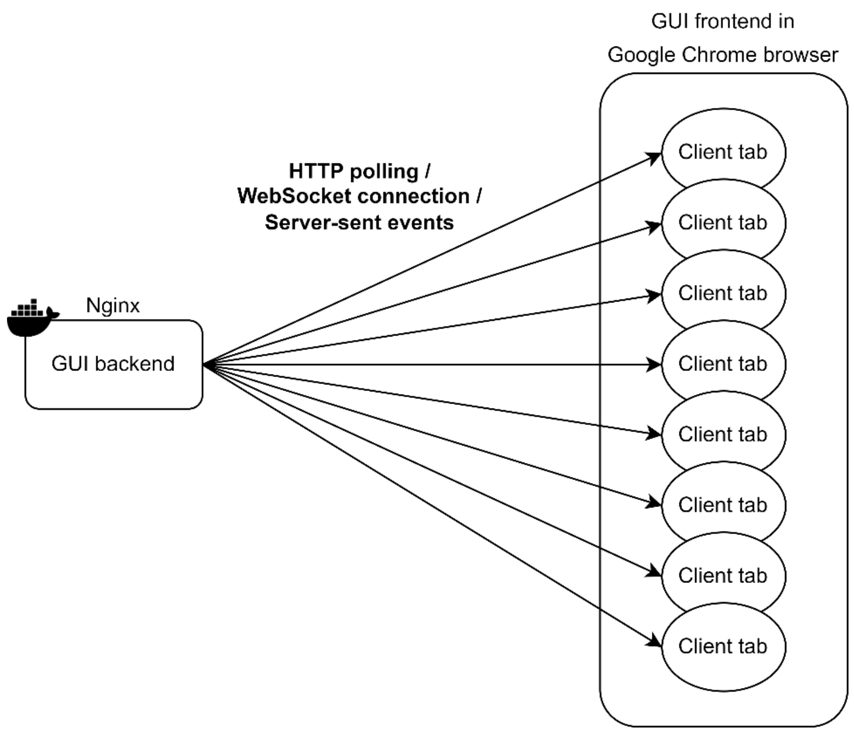

Figure 12. *The system used in empirical testing*

In the test phase eight Google Chrome browser tabs were opened to connect to each frontend component. This triggers eight parallel processes between clients and the backend component. Each process triggers data updates from a backend server to clients every second. Data transfer is secured using TLS, but user authentication is not implemented for the test phase. TLS might have a slight impact on performance, but not considerable. The impact of using TLS encryption is not examined as TLS encryption is in any case used for data traffic. In addition to Google Chrome tests were also run by using Microsoft Edge browser for clients. Test results were broadly similar so the conclusion was that the browser did not significantly affect the outcome of the tests.

Figure 13 presents the received data by each backend server during the performance test. Received data is presented as B/s (byte per second). As can be seen from the figure the received data amount is the largest in the implementation where HTTP polling method is used for data updates. HTTP polling contains additional HTTP header data which is sent back and forth during every data update event triggered by a new HTTP request. It probably explains why the received data amount is the largest in traditional HTTP polling method.

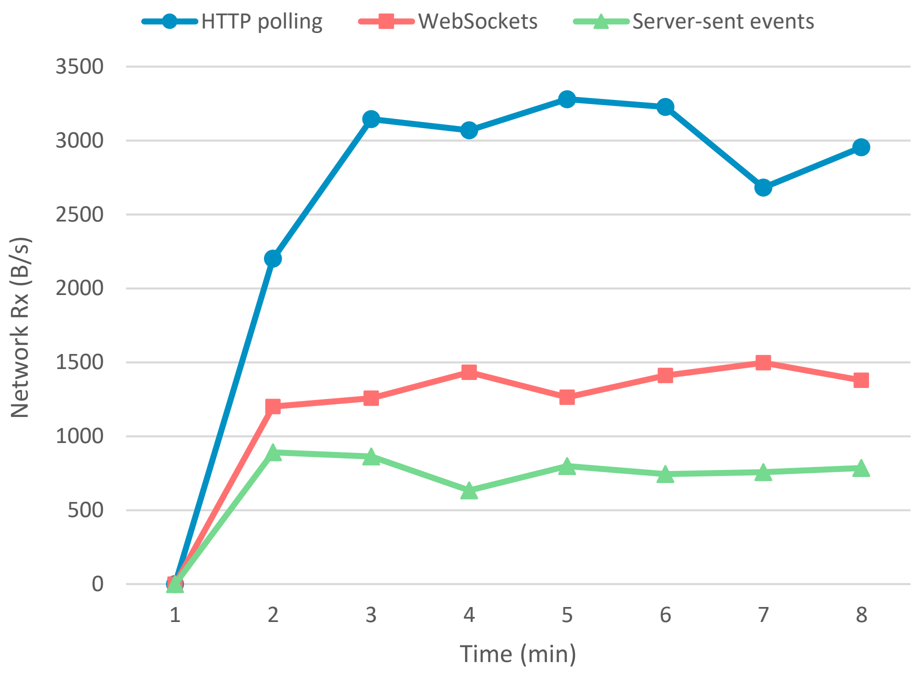

Figure 13. *Data sent from frontend clients to backend services*

The backend server which uses WebSockets receives more data than the backend server implemented with server-sent event updates. Clients of the WebSocket server are sending heartbeat messages to inform the server that they are listening, but it still does not explain the discrepancy entirely. In server-sent event implementation heartbeat messages sent by a client are not required as client activity can be followed by supervising the context of an HTTP request. The context is cancelled when the client closes the connection.

Where the previous figure presented inbound data traffic, the transmitted data by backend servers is presented in Figure 14. Transmitted data is represented as kB/s (kilobyte per second).

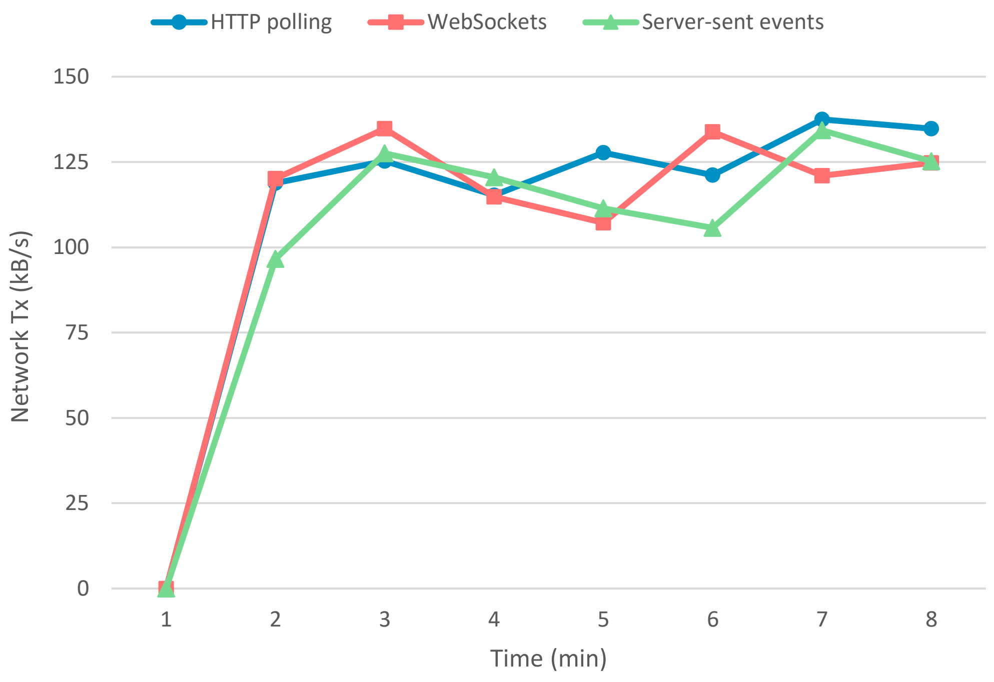

Figure 14. *Transmitted real-time data from backend instances to frontend clients*

The amount of transmitted data is almost on the same level in every backend server implementation. Measurement timepoints may cause a slight variation between results. The amount of transmitted data is approximately 125 kB/s.

The differences in performance can be observed when comparing CPU usage statistics for each backend implementation. CPU usage for backend servers is presented as percentages in Figure 15. CPU usage is the highest in backend implementation which is responding to continuous HTTP polling requests. On the other hand CPU usage is lower in backend servers which are using WebSockets or server-sent events to deliver data updates. The results are quite similar in these two implementations.

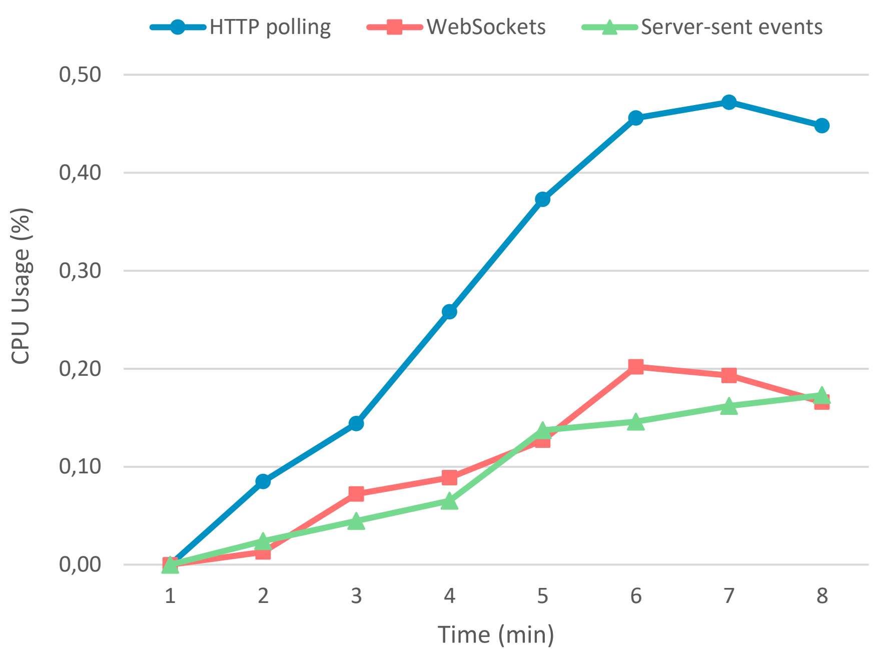

Figure 15. *CPU Usage in backend implementations*

In the HTTP polling method, every request is considered as a new client, and a new connection is created. This may have a major impact on CPU load in test cases. However, in practice most of CPU time will be spent on data parsing and computing in the finished application.

Another measurement for observing the performance of different methods is the memory usage of backend servers. Memory usage is presented as MiB (mebibyte) in Figure 16. Memory usage is the highest in backend implementation which is responding to continuous HTTP polling requests. The lowest memory usage is in the backend service which is using server-sent events for data updates. Memory usage of the backend implemented with WebSockets is between the memory usage of the previous mentioned backend services. It may be explained by the fact that an external library is used for the WebSocket implementation.

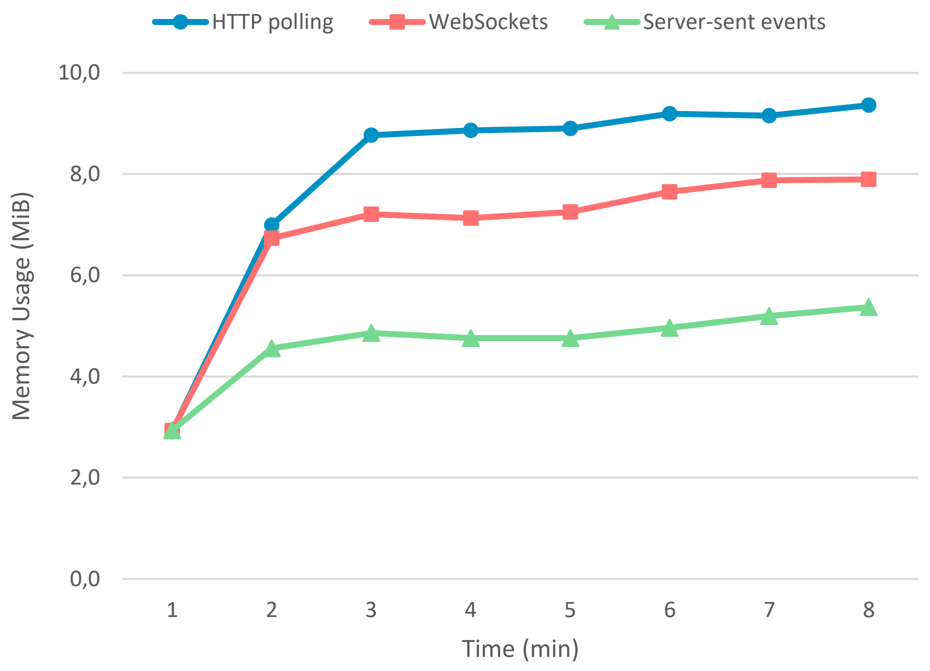

***Figure 16.** Memory usage in backend implementations*

All in all, seems that the backend implemented with server-sent events performed slightly better than servers using HTTP polling method or WebSockets for data updates. However, results may differ for example if another WebSocket library would be used. Benchmarking between SSE, WebSocket and HTTP polling is hard to normalize due to the frequency of updates. In addition, server configuration may still affect the result even if Docker containers were used in the deployment of services.

There is also previous research made considering the performance of web-based communication methods. In the study published by Wroclaw University of Technology authors analyze the performance of web-based systems using XMLHttpRequest, server-sent events and WebSocket. It was stated that WebSocket technology performed best in terms of bidirectional communication. In the case of unidirectional communication from the server to the client server-sent events and WebSocket technologies performed similarly. (Słodziak and Nowak, 2016) The same outcome was also reached in the tests executed in this work.

4.4.2 Feature comparison

In addition to performance, real-time communication methods were also evaluated from other viewpoints. A comparison summary of these methods is presented in Table 2. As can be noted from the empirical performance test results presented in the previous section, the performance of the HTTP polling method is the weakest in the use case of this web application. CPU usage was quite similar in both WebSocket and server-sent event implementations. There was a bit higher memory consumption when using WebSocket for data updates.

Table 2. *Comparison of real-time communication methods*

	HTTP polling	**WebSockets**	**Server-sent events**
Performance	Weak on the backend server side	Good	Good
Scalability	Keeping track of starting & ending points of the polling process makes it worse	Good	Good
Maintainability	Supported by Go standard library	Requires third-party library	Supported by Go standard library
Browser support	Good	Good	No Internet Explorer support, only 6 con-nections/browser with HTTP/1

Performance and scalability depend highly on the practical implementation but different communication methods, however, also have an effect. Different methods require things to be implemented differently in code logic. For example, the HTTP polling process does not happen with a continuous connection, instead, a separate HTTP connection is cre-ated with each data update request. All the requests a client is making during the polling process look the same. This contributes to the fact that the starting point of the HTTP polling process must be somehow indicated by the client if it affects the function of the server. However, this is not the case if WebSocket connections or server-sent events are used. Those methods create a continuous connection for the date update process, and the process between a client and a server starts when the connection is created. Moreover, in the case of the HTTP polling, the ending point of the polling process can be only noticed by keeping track of how much time has passed since the client’s latest poll request. This makes scalability poor when using HTTP polling for this use case.

From a maintainability perspective HTTP polling and server-sent event methods are the best choices as those are supported by Go standard library. WebSocket implementation requires some third-party library. However, there currently exist multiple libraries for WebSocket implementation with Go. The libraries seem to be maintained by volunteers.

When considering browser support HTTP polling and WebSocket methods are well supported by all common browsers. EventSource interface to server-sent events is not supported by Internet Explorer. As Internet Explorer support is already ended by Microsoft, the lack of EventSource support is not considerable. Server-sent events also have another limitation. If those are used with HTTP/1 there can be up to 6 connections per browser and domain with some common browsers. However, when HTTP/2 is used the maximum number of simultaneous HTTP streams can be configured.

When security is considered none of these communication methods provides secure solutions by themselves. TLS encryption can be used with each of the methods. And when considering for example denial-of-service attacks securing must be implemented in code logic or with proxies or firewalls.

WebSocket would support bi-directional communication between client and server, but it does not bring any noticeable advantage for this web application as the web user interface is not designed to have much two-way interactive communication. Therefore, based on performance, scalability and maintainability server-sent events were decided to use for the real-time data updates between backend and frontend services.

5. ARCHITECTURE PROPOSAL

This chapter presents the proposed architecture based on the analysis of different solutions in the previous chapter. The architectural design is not complete, but the main architectural issues have been considered. The component view for the proposed architecture is explained in the first section. The view contains those components that are essential to this web application. The second section describes the module view of the backend component on a more detailed level. In addition, the scalable real-time data processing method is explained through the module view.

5.1 Component view

The proposed architecture consists of components placed on both a central server and edge servers. The component division is based on the alternative in Figure 11, which takes advantage of distributed computing and has a separate backend service for GUI. There is also one addition to the structure. Analyzer service (Haavisto, 2019) is a new component between Parser services and the GUI backend component. Analyzer is located on the central node, and it is responsible for collecting and analysing the data from the services located on edge nodes. It is already part of the system, and it was decided to use it between the components as it already has implemented conversion from Parser's proprietary interface to the REST interface. Analyzer forms weighted averages of data coming from multiple Parser instances on distributed servers and provides a solution to scalability with multiple distributed servers.

Figure 17 represents how components are located on central and edge nodes. Some of the heavy computing tasks are located on edge servers, which takes advantage of edge computing. Data collector services are responsible for collecting the data from base stations and transmitting the data to other services. One of these services is Parser which filters the data coming from Data collectors. It subscribes only needed measurement counters. It creates statical distributions of the KPI data and transmits those to Analyzer service.

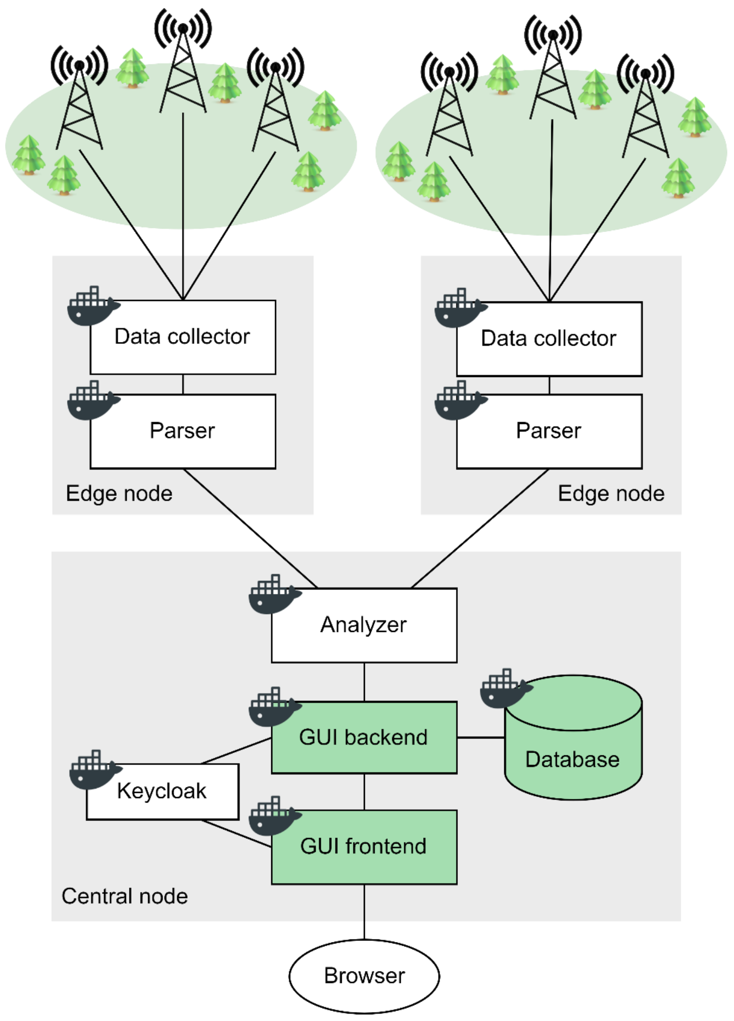

Figure 17. *Web application located in the current system*

Green components are the new components created during this work. GUI frontend component is developed with React and it is responsible for presenting the user interface and application data. The user interface is fetched by the web browser running on a user's device.

GUI backend service is a middle tier, which is running the business logic of the web application. It is developed with Go to enable good performance and scalability. The backend service has a REST interface through which frontend clients can communicate it. Real-time data updates to clients are provided by using server-sent event method. The backend service receives real-time KPI distribution data from Analyzer component. In addition, it also detects problems and suggests possible fault causes based on the KPI distributions. A more detailed module view of the GUI backend component is presented in the following section.

In addition to backend and frontend components, a new database service is needed. SQLite is used as a database management system, as it is lightweight and requires minimal setup and configuration. It provides a good performance capability and can handle a large number of concurrent read and write operations.

5.2 Real-time data processing in the backend server

Data is retrieved from edge computing components only if the interface has at least one active user viewing the statistics. If a request for continuous data updates is received from the client, the backend service starts fetching the data from edge computing components if there is not already another client getting the same data updates.

The backend component has modules which form a publish-subscribe pattern to enable scalable processing of the data. Every real-time data request that comes from a client is handled in a specific controller module. The controller module creates a subscription for a certain topic in the broker module and establishes a continuous connection with the client. If there is at least one subscriber for a topic, a related service module publishes data updates for that data topic throughout the broker module. A simplified module view of the backend service is presented in Figure 18. It shows publish-subscribe pattern located between controllers and services.

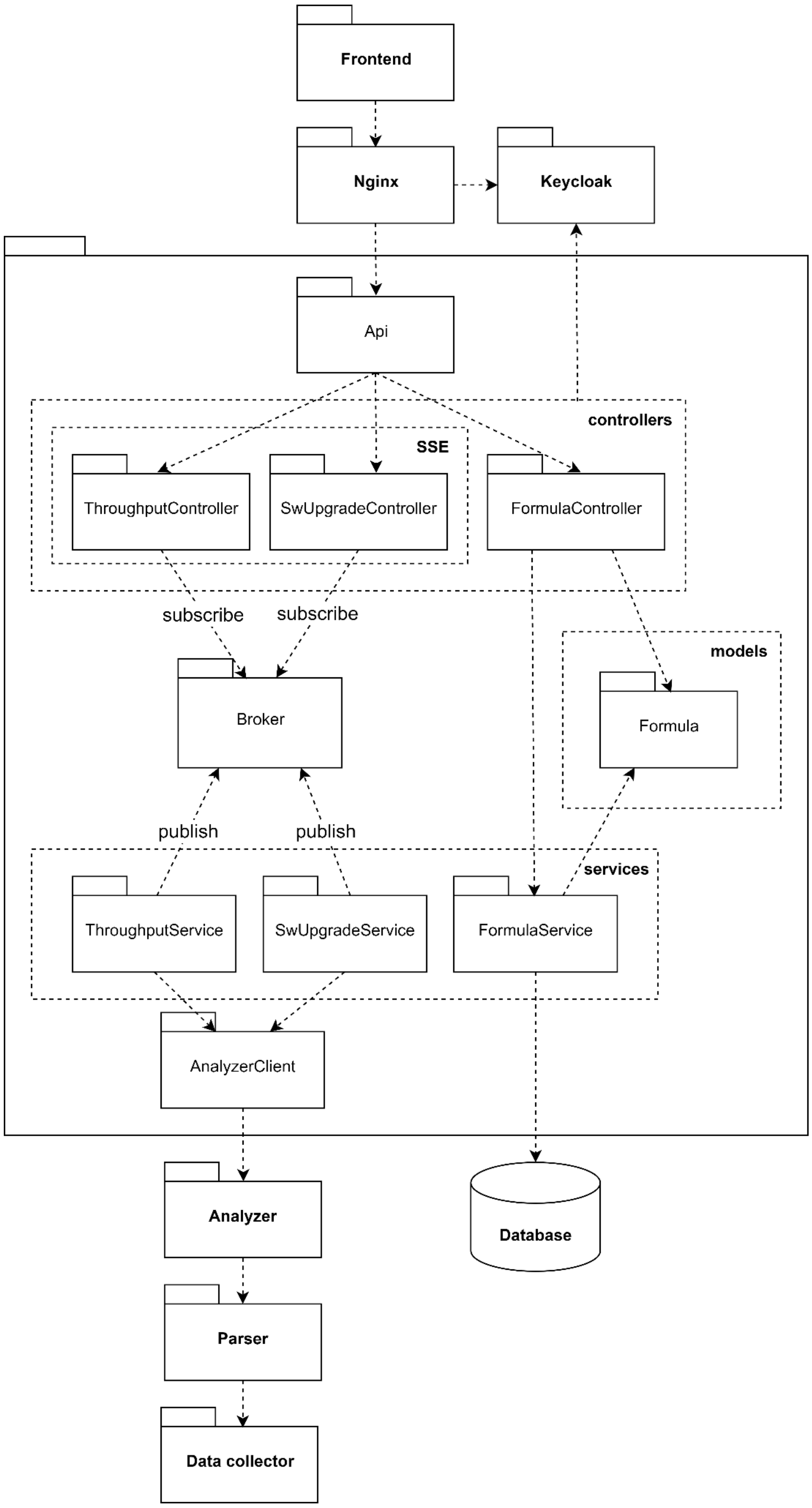

Figure 18. *Backend modules and publish-subscribe pattern*

There are different controller modules for different REST API endpoints. For example, ThroughputController is responsible to handle requests considering the throughput data and SwUpgradeController is responsible to handle requests which consider software upgrade statistics. There are also services which publish similar data topics, like ThroughputService and SwUpgradeService. Those services interact with AnalyzerClient module which is responsible to request data from the external Analyzer component.

In addition to HTTP requests that trigger continuous data updates, in other words server-sent events, there are HTTP requests that trigger events to interact with the database. These requests, for example, update formulas stored in the database. Requests are handled with basic controllers which do not create continuous connections with clients. The FormulaController in Figure 18 is an example of a basic controller, which handles requests considering formula data stored in the database. Through the formula REST API endpoint authorized clients can for example get a list of formulas or create a new formula.

When server-sent events provide real-time data updates to frontend clients, the data will be stored in Redux store. As the data need to be shared between multiple components it is more clear and easier to test when the data will be stored in a common location. Redux will dispatch subscribe action for real-time data updates when certain React components render has committed to the screen. To avoid memory leaks, an unsubscribe action will be called in the clean-up function before the component is removed from the UI.

6. ARCHITECTURE EVALUATION

Architecture evaluation is a process of determining the extent to which an architecture fits the intended purpose (Bass, 2022). This chapter contains an evaluation of the proposed architecture. Evaluation is carried out by using a lightweight architecture evaluation framework based on ATAM (Architecture Tradeoff Analysis Method). In the first section the framework used in the evaluation is presented. Secondly, the evaluation process is described. The last section goes through the results of the architecture evaluation process.

6.1 Lightweight architecture evaluation framework

Software architecture can be evaluated using different evaluation frameworks. One of these is the popular ATAM framework. It has already been used over two decades to evaluate a variety of large-scale architectures. Using ATAM does not require the system to be constructed yet, which makes it suitable for the evaluation of the architecture designed in this thesis. (Bass, 2022) However, ATAM is quite a heavy process as it requires many stakeholders to be present in the evaluation phase, so as such is not a proper choice to use in the evaluation of this architecture. It is resource-intensive and time-consuming, and for small projects, the cost of conducting an ATAM analysis may outweigh the benefits it provides.

The LAE (Lightweight Architecture Evaluation) method (Bass, 2022) uses the same concepts as the ATAM, but it is intended to be used in a project-internal context and to have fewer participants than the ATAM. In the LAE method stakeholder identification can be skipped. It is not required to define stakeholders who will be affected by the architecture and their concerns and requirements. A quality attribute utility tree can also be implemented before the architecture evaluation session. It articulates quality attribute goals in detail and defines quality attribute requirements that architecture must provide. A quality attribute tree is a hierarchical tree structure that provides a systematic way of specifying and organizing the quality attributes of a system.

All the phases of the LAE process are shown in the following list:

1. Present the method steps
2. Review the business goals
3. Review the architecture

4. Review the architectural approaches
5. Review the quality attribute utility tree
6. Brainstorm and prioritize scenarios considering quality attributes
7. Analyze the architectural approaches
8. Capture the results

The evaluation process is quite effective and can be kept in less than a day. The downside is that the evaluation team is internal, and evaluation is typically more subjective than it would be if the evaluation team would also contain externals. (Bass, 2022)

6.2 Evaluation process

Lightweight ATAM evaluation based on the LAE method was kept during this work to evaluate the architecture design. The evaluation was arranged as two one-hour meetings. First, the evaluation method was presented. After that business goals and architecture were reviewed. The architecture proposal was presented, including its key components, relationships, and behaviours. The review did not take so much time as the evaluation was internal. The utility tree was partially implemented already beforehand, and scenarios were then updated in the evaluation meeting. Moreover, importance and difficulty ratings were analysed during the meeting. The final version of the utility tree is presented in Figure 19. The root of the tree represents the overall quality goal for the system, and the branches represent the various quality attribute refinements, that contribute to the quality goal. Branches are further divided into sub-branches, which represent more specific aspects of the quality attribute as scenarios.

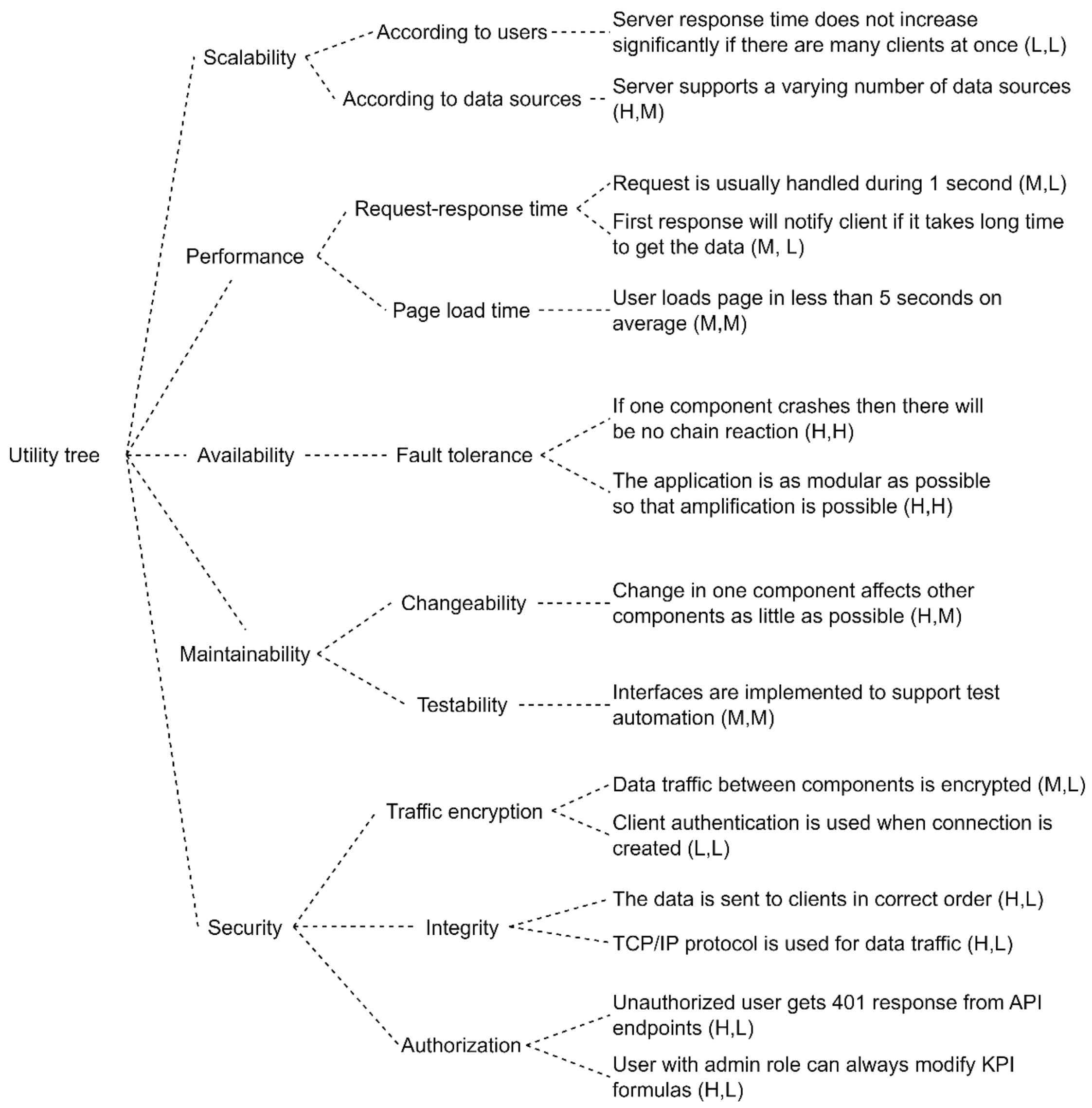
Utility tree
Scalability
According to users
Server response time does not increase significantly if there are many clients at once (L,L)
According to data sources
Server supports a varying number of data sources (H,M)
Performance
Request-response time
Request is usually handled during 1 second (M,L)
First response will notify client if it takes long time to get the data (M, L)
Page load time
User loads page in less than 5 seconds on average (M,M)
Availability
Fault tolerance
If one component crashes then there will be no chain reaction (H,H)
The application is as modular as possible so that amplification is possible (H,H)
Maintainability
Changeability
Change in one component affects other components as little as possible (H,M)
Testability
Interfaces are implemented to support test automation (M,M)
Security
Traffic encryption
Data traffic between components is encrypted (M,L)
Client authentication is used when connection is created (L,L)
Integrity
The data is sent to clients in correct order (H,L)
TCP/IP protocol is used for data traffic (H,L)
Authorization
Unauthorized user gets 401 response from API endpoints (H,L)
User with admin role can always modify KPI formulas (H,L)

Figure 19. *The final version of the utility tree after lightweight ATAM evaluation*

Utility tree identifies the most important quality attributes, which are in this case scalability, performance, availability, maintainability, and security. These attributes are then divided into refinements, such as request-response time and page load time. According to these refinements there are scenarios listed in the utility tree. The application and its architecture are wished to implement these scenarios.

The importance and difficulty of each scenario were rated on a scale of high (H), medium (M) and low (L). These rates are mentioned in parentheses. The first letter is a rating for the importance and the second letter is a rating for the approximated difficulty of implementation of a scenario.

Scalability is observed both from an application user perspective and a data source perspective. When the number of data sources increases, also the amount of incoming data increases. The first scenario considers that the backend server's response time does not increase significantly if there are many clients at once making requests to the backend server. This scenario has a lower importance as the application will probably not have multiple concurrent users. Its implementation is estimated to be easy with server-sent events and the current backend structure, so the difficulty is also low. On the other hand, scalability according to the varying number of data sources has a high importance. Surely, the application must be able to maintain good performance in an environment which has only one edge server but also in an environment which is having multiple edge servers and hence multiple data sources and high data load.

The performance-related scenarios consider the request-response time and page load time. In this work user interface is not designed, and thus user experience is not considered as such. However, the application's performance should be good. The scenarios considering the application performance are rated with medium importance. Otherwise, difficulties are rated as low, but it is not sure yet how hard it will be to load the page with all the needed statistics in less than 5 seconds on average. If the page load time will not be good enough, rendering can be optimized by loading only critical user interface items first.

The availability of the application is considered from the fault tolerance point of view. Good fault tolerance has high importance. The fault tolerance is largely affected by code logic and style decisions, and the application's availability is probably mostly ensured with good error handling. The application must not crash if another component cannot be reached. Modularity affects usability, but it also has an impact when multiplying components. It is important to have a modular application structure as when part of the application cannot be reached it can easily be multiplied.

The maintainability-related scenarios consider loose coupling and testability. The system must be loosely coupled so that change in one component affects other components as little as possible. When the impact of a change in another component is minor the changeability is good. Testability has been not considered yet in this phase, and it often gets less attention when architecture is designed. Tactics to achieve good testability are divided into two categories. The first category concerns the ability to control and observe a system state. Another category is about limiting the complexity of software. Localizing state storage and storing application state in a single place improves testability, as it is easier to control and observe the system state from a single place. (Bass, 2022) By using Redux store for storing the application state in a web browser the state is localized and the application is easier to test.

The fifth quality attribute in the utility tree is security. The first two security scenarios consider traffic encryption. Data traffic between components will be encrypted by using TLS encryption. In addition, client authentication is supported between backend components. Security scenarios related to data integrity consider the order in which the real-time data is sent from the backend component to clients. The order must be correct, and it can be indicated by observing timestamps in the data. When using server-sent events instead of traditional HTTP polling, the backend server is responsible to maintain the correct order for the real-time data updates. If HTTP polling method would have been used, clients should have sent indicators to the backend server to tell more precisely which data packet is required. Another data integrity scenario is to use TCP/IP protocol for data traffic as it provides a checksum which can be used to detect corruption in data integrity. The third refinement of the security attribute is an authorization. Authorization scenarios consider the API implementation of the backend server. An unauthorized user gets 401 error from all API endpoints, which describes Unauthorized Error. Another scenario considers the rule that users with an admin role are the only ones authorized to make modifications to KPI formulas.

After this architectural sensitivity and trade-off points were considered. It proved to be the most challenging part of the evaluation process. In addition to sensitivity and trade-off points, evaluation results were reviewed at the end of the evaluation meetings. Results of the architecture evaluation, including the sensitivity and trade-off points are considered in the next section.

6.3 Evaluation results

There were also some architectural sensitivity and trade-off points considered during the evaluation. Sensitivity points are architecture decisions which affect the attainment of some quality attribute. Trade-off points are considered decisions that affect more than one quality attribute. Trade-off point decisions can for example improve the achievement of some attribute but at the same time make some other quality attribute worse.

The main trade-off points that were noticed during the evaluation considered security decisions. Most of them have a small degrading effect on performance. For example, using TLS encryption or authentication consumes more computing resources and processing time. However, the effect on performance is very little, and security decisions are mandatory to implement so there is no uncertainty in trade-off points considering security.

Using goroutines in the backend server implementation affects positively multiple different quality attributes. The first thing to mention is performance, which is the main reason to use concurrency in this application. The implementation also affects scalability positively as it is relatively easy to adapt the number of goroutines to current resource needs. The third quality attribute affected is maintainability as communication via channels between threads is implemented in a user-friendly way with Go. It makes it also easy to maintain the code in the future.

Regarding the real-time statistics updates with server-sent events, it was decided to implement cache memory to improve performance in the REST API of the GUI backend component. REST architecture supports cache mechanisms, but it does not automatically implement those. HTTP methods include caching mechanisms that can help improve performance by reducing the number of requests that need to be made to the server. One approach for caching server-sent events data is to use the Last-Event-ID header, which allows clients to request only new events from the server.

7. ANALYSIS AND DISCUSSION

This chapter contains an analysis and discussion about research and future development ideas. The chapter is divided into two sections. The first section covers the research process, and the second section is about future development. Future development considers new ideas on how to improve current architecture solutions and continue development work in the future.

7.1 Research process

At the beginning of the research the application requirements were defined. As the user interface was not designed in the first place, the requirement specification was not complete. In addition, a detailed API structure could not be designed. However, it was recognized that the amount of data used in the analysis would be large, and that should be considered in architecture design. The suitable component division was found quite easily. It took well into account the current components of the system. As the project was implemented as a proof of concept there was not much time allocated to it. When already existing components were reused, there was no need to write so much new functionality. Moreover, component reuse ensured consistency across the application.

Evaluating architecture before the software is ready was challenging as the architecture was a quite high-level representation of the system. It was not as easy as first thought to understand and evaluate the implications of the architecture. The design for the system was limited and incomplete, making it difficult to predict how the architecture will perform in real-world conditions. Moreover, without the final version of the software developed, the empirical performance testing was slightly unreliable. It was not possible to generate realistic testing data and scenarios that accurately reflect how the system will be used. It is also difficult to simulate real-world conditions, such as load and stress when evaluating architecture. This can make it challenging to accurately predict the performance and scalability of the architecture. In addition to this, software requirements can change later and evolve over time, making it difficult to properly anticipate and evaluate all the trade-offs and impacts of different architectural choices.

Moreover, for this kind of small internal project, other evaluation techniques that are less formal and require fewer resources may be more appropriate. ATAM may be a useful tool for ensuring the quality and reliability of the architecture for larger or more complex projects. However, the quality attribute tree helped to ensure that the quality attributes are understood and consistently considered throughout the development process. It was convenient to process the overall picture of the designed architecture solution.

Despite these challenges, evaluating architecture before the software is ready is still an important part of the development process. Different techniques and tools, for example prototyping and scenario-based evaluation help to mitigate these challenges. By combining these techniques with expertise and experience, it is possible to effectively evaluate architecture and make proper decisions about the design of a software system.

7.2 Future development

KPIs are widely used in fault identification but it is still hard to know the effects of each fault cause on the KPIs. Even if the network problem is solved, the real cause is unknown on many occasions, which may result in the same problem happening again. This web application is developed as a part of the so-called rule-based analysis system. One of the major advantages of rule-based diagnosis systems is that they are designed based on the reasoning and thinking of experts, and so are easily understandable and interpretable. The rule-based system only diagnoses problems when all the rules are satisfied, so it cannot identify problems whose impact on KPIs is a small deviation from the defined threshold. However, the diagnosis error is quite small. In addition to rule-based diagnosis systems there are also fuzzy-logic-based systems, where the undiagnosed errors are reduced at the cost of a small increase in the faulty diagnosed error. (Gómez-Andrades *et al.*, 2016) Fuzzy-logic-based systems are designed to handle uncertainty and imprecision in data and inputs, whereas rule-based systems rely on explicit rules. In the future it could be considered if the fuzzy-logic-based diagnosis would be easier to implement on a larger scale and if it would give good analysis results. In some cases, fuzzy-logic-based systems can handle complex systems more effectively than rule-based systems. Rule-based systems require manual updates to adapt to changing conditions. In addition, a fuzzy-logic-based diagnosis could take advantage of machine learning. Machine learning can be used to automatically generate fuzzy-logic rules, which can improve the accuracy and effectiveness of fuzzy-logic-based systems.

Another thing to consider in future development is the component structure. In the current solution Analyzer component combines and centralizes the data it receives from Parser instances located in multiple edge servers. In addition to this it also provides a REST interface for the GUI backend component. Analyzer was in the current system already at the beginning of this project, and it provided a REST interface and the possibility to edge data centralization already at that point. Therefore, it was decided to use it also at least in the proof of concept phase of this project. Even though the current application may work well also with Analyzer as a part of it, the GUI backend component could also handle data centralization in the future. Similar functionality could be used but previous C++ code should be written with Go. If also data centralization would be handled in GUI backend, there would be fewer components and interfaces between them. Fewer components and interfaces would help to improve system performance by reducing communication overhead and data transfer between components. In practice, fewer interfaces would help to reduce the attack surface of the system and minimize the potential for security vulnerabilities.

At this point of the design there were only Parser services used as a data source in edge nodes. However, there might be also additional services to be used as a data source in the future. Besides the data from Parser services, additional data sources could be used in the analysis of cellular network problems. This might require the reconstruction of component interfaces, and one possible solution would be to use a distinct centralized API component. Centralized API would act as an API gateway between GUI backend and services on edge nodes. It would create a unified API interface through which the GUI backend could interact with multiple servers without knowing the detailed interfaces for each service. A centralized API component could handle the details of interacting with the backend services on edge nodes.

8. CONCLUSIONS

This research was carried out for a software development team in the Nokia organization. The research problem appeared as there was a need for a web application that would provide an analysis of a state of mobile network systems. Analysis should be done based on the measurements collected from mobile networks. In addition, possible solutions should be provided if problems occur. Descriptive information about a state of mobile network systems is needed, but it requires a deep analysis in backend services. In this constructive research an architecture proposal is conducted to provide a solution for a web application that would be able to analyse a condition of a mobile network system and indicate possible solutions for these problems.

At the beginning of the research initial application requirements were collected and specified. Based on the requirements some problems were found. These points were acknowledged when designing an architecture proposal and technology selections. It was also important to design architecture in a way that supports future changes and enhancements. Possible solutions were collected via literature review and from previous experience.

After architecture and technology alternatives were found, different solutions were analysed in a comparative study. At the beginning of the comparative study alternative component structures were compared. The preferred structure was the alternative which made use of edge computing in the formation of KPI distributions. The scalability problem with distributed services in edge nodes was resolved with an already existing component in the system. This component receives data from multiple edge computing components and provides weighted averages for KPI distribution data.

Research also compared Go and Node.js technologies for backend implementation. Most of all backend component was preferred to provide good performance and scalability and consequently Go was chosen. Goroutines support scalability well, and Node.js as a single-threaded engine is not able to have the same performance as Go.

Comparative analysis was also conducted for a web server API technology. REST and GraphQL were compared even if they do not completely exclude each other. However, REST architecture was decided to use in backend API implementation as a possibility for query optimization by clients was not recognized in this use case. In addition to the comparative analysis, previous research about API implementations was reviewed in the literature review.

As the backend component was required to send continuous real-time data updates to clients, it was needed to decide what real-time communication method would be used in communication between the backend and frontend components. In addition to comparative study empirical research was used for performance evaluation of different real-time communication methods. It was found that server-sent events and WebSockets performed better than traditional HTTP polling method. Performance was evaluated in the simulated use case of the designed application. Eventually, server-sent events were decided to be used as a communication method, as they are supported by Go's standard library.

An architectural proposal was conducted based on the above-mentioned studies. After the presentation of the proposal, an architecture evaluation was done. The evaluation was carried out by using a lightweight architecture evaluation framework. In the final analysis the architecture design seems successful but as implementation is still ongoing it is not possible to make an accurate assessment about the success of the research results. Nevertheless, there are still multiple plans for future development.

REFERENCES

3GPP 32.450 (2022) ‘Telecommunication management; Key Performance Indicators (KPI) for Evolved Universal Terrestrial Radio Access Network (E-UTRAN): Definitions’, in. 3GPP. Available at: https://www.3gpp.org/DynaReport/32450.htm (Accessed: 26 March 2023).

Aalto, P. (2022) *Method for health check of radio networks from real-time data streams*. Master’s thesis. University of Tampere. Available at: https://trepo.tuni.fi/handle/10024/139167.

Andrawos, M. and Helmich, M. (2017) *Cloud native programming with Golang: develop microservice-based high performance web apps for the cloud with Go*. 1st edition. Birmingham, England: Packt Publishing.

Anurag, V.N. (2018) *Distributed Computing with Go*. 1st edition. Packt Publishing.

Baldoni, R. *et al.* (2009) ‘Distributed Event Routing in Publish/Subscribe Communication Systems’, *Middleware for Network Eccentric and Mobile Applications*.

Bass, L. (2022) *Software architecture in practice*. 4th edition. Boston: Addison-Wesley (SEI series in software engineering).

Chinnathambi, K. (2018) *Learning React: a hands-on guide to building web applications using React and Redux*. 2nd edition. Addison-Wesley Professional.

Choi, D. (2020) *Full-Stack React, TypeScript, and Node*. Packt Publishing.

Costanza, P., Herzeel, C. and Verachtert, W. (2019) ‘A comparison of three programming languages for a full-fledged next-generation sequencing tool’, *BMC Bioinformatics*, 20(1). Available at: https://doi.org/10.1186/s12859-019-2903-5.

Elrom, E. (2021) *React and libraries: your complete guide to the React ecosystem*. 1st ed. 2021. Berkeley, CA: Apress. Available at: https://doi.org/10.1007/978-1-4842-6696-0.

Fielding, R.T. (2000) *Architectural Styles and the Design of Network-based Software Architectures*. University of California.

Golang (2022b) *Effective Go - The Go Programming Language*, *Go.dev*. Available at: https://go.dev/doc/effective_go (Accessed: 15 October 2022).

Golang (2022a) *Frequently Asked Questions (FAQ) - The Go Programming Language*, *Go.dev*. Available at: https://go.dev/doc/faq (Accessed: 15 October 2022).

Gómez-Andrades, A. *et al.* (2016) ‘Methodology for the Design and Evaluation of Self-Healing LTE Networks’, *IEEE Transactions on Vehicular Technology*, 65(8), pp. 6468–6486. Available at: https://doi.org/10.1109/TVT.2015.2477945.

GraphQL (2021) *GraphQL*. Available at: http://spec.graphql.org/October2021/ (Accessed: 5 October 2022).

Haavisto, O. (2019) *Matkapuhelinverkkojen analysointiohjelmisto*. Master’s thesis. University of Tampere. Available at: https://trepo.tuni.fi/handle/10024/118077.

Hribernik, M. and Kos, A. (2020) ‘Secure WebSocket Based Broker and Architecture for Connecting IoT Devices and Web Based Applications’, *IPSI Transactions on Advanced Research*, 16(1). Available at: http://ipsitransactions.org/journals/papers/tar/2020jan/p2.pdf.

http package - net/http - Go Packages (2022). Available at: https://pkg.go.dev/net/http (Accessed: 20 November 2022).

HTTP/2 | Node.js v19.1.0 Documentation (2022). Available at: https://nodejs.org/api/http2.html (Accessed: 20 November 2022).

Ibe, O.C. (2018) *Fundamentals of data communication networks*. 1st edition. Hoboken, New Jersey: Wiley.

Ingeno, J. (2018) *Software Architect’s Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts*. Birmingham: Packt Publishing, Limited.

Karla, T. and Tarnawski, J. (2019) ‘Soft Real-Time Communication with WebSocket and WebRTC Protocols Performance Analysis for Web-based Control Loops’, in *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR)*. *2019 24th International Conference on Methods and Models in Automation and Robotics (MMAR)*, pp. 634–639. Available at: https://doi.org/10.1109/MMAR.2019.8864680.

Kornienko, D.V., Mishina, S.V. and Melnikov, M.O. (2021) ‘The Single Page Application architecture when developing secure Web services’, *Journal of Physics: Conference Series*, 2091(1). Available at: https://doi.org/10.1088/1742-6596/2091/1/012065.

Kulshrestha, A. (2013) ‘An Empirical Study of HTML5 Websockets and their Cross Browser Behaviour for Mixed Content and Untrusted Certificates’, *International Journal of Computer Applications*, 82(6).

Lawi, A., Panggabean, B.L.E. and Yoshida, T. (2021) ‘Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System’, *Computers*, 10(11). Available at: https://doi.org/10.3390/computers10110138.

Lehr, W., Queder, F. and Haucap, J. (2021) ‘5G: A new future for Mobile Network Operators, or not?’, *Telecommunications Policy*, 45(3). Available at: https://doi.org/10.1016/j.telpol.2020.102086.

Li, S. *et al.* (2021) ‘Understanding and addressing quality attributes of microservices architecture: A Systematic literature review’, *Information and software technology*, 131. Available at: https://doi.org/10.1016/j.infsof.2020.106449.

Massé, M. (2011) *REST API Design Rulebook*. 1st edition. Place of publication not identified: O’Reilly Media Incorporated.

Mei, W. and Long, Z. (2020) ‘Research and Defense of Cross-Site WebSocket Hijacking Vulnerability’, in *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*. *2020 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, pp. 591–594. Available at: https://doi.org/10.1109/ICAICA50127.2020.9182458.

Mikuła, M. and Dzieńkowski, M. (2020) ‘Comparison of REST and GraphQL web technology performance’, *Journal of Computer Sciences Institute*, 16, pp. 309–316. Available at: https://doi.org/10.35784/jcsi.2077.

Node.js community (2022) *Introduction to Node.js*, *Introduction to Node.js*. Available at: https://nodejs.dev/en/learn/ (Accessed: 15 October 2022).

Raj, P. (2017) *Architectural patterns: uncover essential patterns in the most indispensable realm of enterprise architecture*. 1st edition. Birmingham, England: Packt.

Rezaei, S. *et al.* (2016) ‘Automatic fault detection and diagnosis in cellular networks using operations support systems data’, *IEEE/IFIP Network Operations and Management Symposium*, pp. 468–473. Available at: https://doi.org/10.1109/NOMS.2016.7502845.

Saint-Andre, P. *et al.* (2011) *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. Request for Comments RFC 6202. Internet Engineering Task Force. Available at: https://doi.org/10.17487/RFC6202.

Sesto, V. *et al.* (2020) *The Docker Workshop: Learn How to Use Docker Containers Effectively to Speed up the Development Process*. Birmingham: Packt Publishing, Limited.

Shi, W. *et al.* (2016) ‘Edge Computing: Vision and Challenges’, *IEEE Internet of Things Journal*, 3(5), pp. 637–646. Available at: https://doi.org/10.1109/JIOT.2016.2579198.

Słodziak, W. and Nowak, Z. (2016) ‘Performance Analysis of Web Systems Based on XMLHttpRequest, Server-Sent Events and WebSocket’, in, pp. 71–83. Available at: https://doi.org/10.1007/978-3-319-28561-0_6.

de Souza Soares, E.F. *et al.* (2018) ‘Evaluation of Server Push Technologies for Scalable Client-Server Communication’, in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pp. 1–10. Available at: https://doi.org/10.1109/SOSE.2018.00010.

van Steen, M. and Tanenbaum, A.S. (2016) ‘A brief introduction to distributed systems’, *Computing*, 98(10), pp. 967–1009. Available at: https://doi.org/10.1007/s00607-016-0508-7.

Tarkoma, S. (2012) *Publish/subscribe systems: design and principles*. London ; Wiley (Wiley series on communications networking & distributed systems). Available at: https://doi.org/10.1002/9781118354261.

The WebSocket API (WebSockets) (2022) *MDN Web Docs*. Available at: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (Accessed: 13 September 2022).

Thomas, M.T. (2018) *React in action*. 1st edition. Shelter Island, New York: Manning Publications.

Using server-sent events (2022) *MDN Web Docs*. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events (Accessed: 13 September 2022).

Walelgne, E.A. *et al.* (2018) 'Analyzing throughput and stability in cellular networks', in. *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pp. 1–9. Available at: https://doi.org/10.1109/NOMS.2018.8406261.

Wang, V., Salim, F. and Moskovits, P. (2013) 'WebSocket Security', in *The Definitive Guide to HTML5 WebSocket*. Berkeley, CA: Apress, pp. 129–147. Available at: https://doi.org/10.1007/978-1-4302-4741-8_7.