Bachelor's thesis

Information and Communications Technology

2019

Sampsa Kaskela

APPLYING REAL-TIME USER INTERFACE PRACTICES TO GAMES

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S / MASTER'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Complete the info here as I have done on the title pagee programme

2019 | 29 pages

Sampsa Kaskela

APPLYING REAL-TIME USER INTERFACE PRACTICES TO GAMES

Real-time applications are becoming more common in the industry because the number of devices that are connected to internet is increasing. The game industry is also growing and different systems start to integrate game elements to user interfaces. Because of the large number of devices, it becomes increasingly more important for systems to have an optimized user interface.

The objectives of the thesis were to create real-time user interface and to figure out the best practices when developing real-time user interfaces and how they could be used in similar projects and games. Another objective of this thesis project was to create an web application that would be used to monitor and control trains in real-time. The available technologies were compared and React was chosen for the webapplication. The application also used server-sent events to communicate between client and server.

In conclusion, it is important to minimize the number of changes in the client that are performed when new data is received. The number of elements should be minimized in the screen so that large datasets can be managed without affecting the user experience. When communicating with the server, the number of connections should be minimized and connections should be kept open so all the relevant data can be communicated without delay. The received and computed data should be cached so that the same data can be rendered more quickliy when there are no changes to the data. Similar techniques can be used in games that have user interfaces.

KEYWORDS:

real-time, user interface, game, communication, web

OPINNÄYTETYÖ (AMK / YAMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Koulutus

2019 | 29 sivua

Sampsa Kaskela

REAALIAIKA KÄYTTÖLIITTYMÄ MENETELMIEN SOVELTAMINEN PELEISSÄ

Reaaliaika sovellukset ovat entistä yleisempiä, koska internettiin kytkettyjen laitteiden määrä kasvaa. Myös peliala on kasvussa ja eri järjestelmät yhdistelevät pelien ominaisuuksia käyttöliittymiin. Koska laitteiden määrä kasvaa, on tärkeää, että käyttöliittymät optimoidaan suurelle määrälle laitteita.

Opinnäytetyön tavoitteena oli kehittää reaaliaika sovellus ja selvittää mitä pitää ottaa huomioon kun kehitetään reaaliaika sovelluksia. Toisena tavoitteena oli selvittää, kuinka samoja menetelmiä voisi käyttää muiden sovellusten kehittämisessä kuten esimerkiksi peleissä. Työ oli www-sovellus, jota käytettiin junien monitorointiin ja ohjaukseen. Eri teknologioita verrattiin ja React valittiin käytetäväksi teknologiaksi. Käyttöliittymän ja palvelimen välisessä viestinnässä käytettiin Server-Sent Events-teknologiaa.

Työssä huomattiin, että on tärkeää tehdä mahdollisimman vähän muutoksia käyttöliittymässä. Ruudulla olevien elementtien määrä tulee pitää pienenä, jotta suuria määriä dataa pystytään käsittelemään ilman, että käyttökokemus huononee. Yhteydet palvelimeen tulisi pitää avoimina, jotta nopeasti muuttuva data voidaan lähettää käyttäjälle ilman viivettä. Yhteyksien määrä tulisi pitää pienenä. Saatu data ja siitä johdettu data tulisi tallentaa väliaikaisesti, jotta sama data voidaan päivittää käyttöliittymään nopeammin silloin kun data ei ole muuttunut. Samat tekniikat toimivat myös pelien käyttöliittymissä.

ASIASANAT:

reaaliaikaisuus, käyttöliittymä, pelit, kommunikaatio, www

CONTENTS

FIGURES

1 INTRODUCTION

Real-time applications are becoming more common in the IT industry and it is even more important to develop applications that can be used to monitor thousands of devices simultaneously. The increasing number of devices connected to the internet will increase the demand for efficient systems so that a user can perform necessary actions. It is also important to provide the necessary tools to the user so that connected devices can be maintained and problems can be addressed quickly.

There have been many studies on creating real-time user interfaces and also has been articles that explain the usages of real-time systems. (Harder, Zarnett, Montaghami & Giannikouris, 2014) None of these projects have explained how the end products performs in regards to the user interface but instead they have focused on how they performed with a specific technology like video or audio. Because of this, the results could not be generalised to work with other types of real-time applications or games.

The first objective of this thesis project was to develop a real-time user interface to remotely monitor and control trains. Each train can send different data at different times to user, so it is important to be able to provide relevant information to the user without delay. The second objective was to figure out the best practices when developing real-time applications. A common application, often overlooked as a real-time system, are games. This thesis evaluates the possibility of using system real-time development practices for game development. The project mainly focuses on the client part of the application and does not focus on finding what the best practices are for the server part of the application.

The thesis first introduces the real-time concept and different technologies that were considered during the project. Different technologies are compared and the best ones are chosen for the development. Different design patterns are explained as well as the reasoning behind them. At the end, the resulting application is evaluated and usages in game development are discussed.

2 THEORETICAL BACKGROUND

This section contains the theoretical background of the thesis which is required to understand the methodology of the thesis. This chapter includes an introduction to real-time systems and different technologies used to create such systems.

2.1 Real-time systems

A real-time system is a system that responds to user or real life actions within a short time frame. The time frame of the system is based on system requirements and what the purpose of the system is. System requirements define what the deadline of the system is. The deadline defines how a system should perform and what the consequences of not meeting the deadline are. This divides real-time systems to 3 types: hard, firm and soft real-time systems.

In hard real-time systems, it is very important to meet the deadline. Failure to do so will result in failure of the system or the data provided will not be sufficient to perfom required actions. These kinds of systems are, for example, patient monitoring in hospitals or weapon systems in the military. Firm real-time systems have lower requirements. If the deadline is not met in a firm real-time system, then it will not result in failure but data still has no value if it is not delivered within specified time frame. This will result in lower quality of service. In soft real-time systems, the data always has value, but if the deadline is not met, then the value of the data is reduced. These are usually monitoring systems that are used to detect faulty devices, but might not require immediate attention when errors are found.

Real-time systems are used to monitor, interact with and control different devices in real life. These devices can be anything that sends or receives data when interacting with the environment or receiving actions from the user. Such devices require a real-time system when they need to respond to user interaction or devices need to report information quickly so that the user can respond to it. Reporting improves safety and security of the system since any errors can be quickly seen and reacted to. Reactions are performed in real life or using a graphical user interface.

User interfaces give a way for humans to interact with the device or application. User interfaces can be another device, like a mouse or keyboard, or application with a graphical interface. Devices can, for example, have buttons that users can press to perform centain actions. Graphical user interfaces (GUIs) can, for example, have tables to display data or text fields to enter data. The goal of the user interface is to help users easily operate machines to produce a desired outcome. (Harder, Zarnett, Montaghami & Giannikouris, 2014)

2.1.1 Real-time systems in Games

Games are almost always real-time systems since they display graphics and information to the user continuously. Most games are not usually used to control or monitor real devices and instead are self-contained services. Depending on the style of the game, games can be firm real-time systems or soft real-time systems. Soft real-time systems are typically present in turn-based games where the timing does not need to be accurate. Firm real-time systems are usually multiplayer games, where players interact with each other in real-time. These types of games are, for example, first person shooters or real-time strategy games. It is important that a game is able to efficiently transfer data between clients so that the quality of the game does not suffer. When a game is not able to transfer data quickly enough, it usually appears as latency where the game in question freezes and then suddenly starts again after a short time. Games can also have control panels for the administrator to monitor the game. This is usually a soft real-time system that displays data about the game, such as the number of players and how active each player is. The control panel is not part of the game but exists as a separate service, possibly in a website.

2.1.2 Real-time systems in Web

Real-time systems are commonly used in web environments, since this allows end-users to access the system anywhere without installing any applications. Hosting an application in the web also allows developers to update the application without requiring effort from the end-users. Web applications usually consist of a user interface that users use to monitor the connected devices. It also has a server that responds to user interactions and transmits the actions to the correct devices. Devices report their status to the server

so that the server can update the information and send it to the user. The web has multiple technologies that are used to build user interfaces and communicate with the server.

2.2 Web technologies

Web experience for the user is made of two core parts, front-end and back-end. The front-end consists of the parts that user sees and interacts with. This is the user interface that is displayed to users and any logic that is executed within the client. The back-end contains the logic and data that users request from the server. The back-end is used to store data permanently, while the front-end is used to store data temporarily. Users use the front-end to interact with the back-end. Front-ends are mainly built using the core technologies of the web: HTML, CSS and JavaScript.

2.2.1 HTML

HTML was first introduced in 1993. The current version of HTML is called HTML5 and it was released in 2014. HTML standardisation is carried out by the World Wide Web Consortium (W3C). These standards are used by browser vendors to make everything work similarly in all browsers. HTML is plain text that defines the structure of web pages. It uses tags that can be used to define elements in browsers, like navigation or header of the page. Each tag is surrounded with angle brackets (<>). HTML forms a tree which is called a document object model (DOM). Each webpage usually has 1 root element which is the html tag (<html>) and it is usually followed by head and body tags, which contain information and content of the page. Each element can have attributes that can be used to define how a browser interprets the page. For example, it can be used to define what classes the element should use for styling or how the element should respond to user interaction. (MDN Web Docs, 2019b)

2.2.2 CSS

Cascading Style Sheets (CSS) is a language that is used to define the look of the webpage. It is usually used to style elements created in HTML. CSS can, for example, be used to define colors and size of the elements. CSS is one of the most common

methods of creating responsive applications and websites that can be displayed on multiple devices. Responsivenes means that different elements change their look based on screen resolution. For example, some elements, such as buttons and links can be larger in smaller screens to make them easier to interact with. (MDN Web Docs, 2019a)

2.2.3 JavaScript

JavaScript is a dynamic programming language. JavaScript is used to define the client logic in web applications. JavaScript is a weakly typed language. This means that types are defined based on each variable's values, meaning the developer does not explicitly define the type of each variable. JavaScript is usually run in a browser environment, but new technologies such as Node.js also allow it to be used on servers. JavaScript is based on ECMAScript specification that is meant to standardize the language and make it so that it is implemented consistently across all browsers. The 9th edition of ECMAscript, known as ES8, is the newest version of ECMAscript and is currently supported by most modern browsers. (MDN Web Docs, 2019d) If developers require support for older browsers, they can transpile their code to an older version using a transpiler, for instance Babel, so that applications become compatible with older browsers. (Babel, 2019)

2.3 Frameworks and libraries

JavaScript has multiple libraries and frameworks. A library is collection of functions that speeds up the development process by providing developers with different functions. These can include a collection of mathematical functions or different user interface components. Libraries can be easily added to existing projects and used with little to no configuration. Most libraries do not depend on any specific framework, but some do. Frameworks differ from libraries in that usually applications are built using them to define how applications should be structured and how they behave. A framework is usually decided when a project is started. It provides a baseline for the project, but the application code defines the usefulness of the application.

2.3.1 React

React is a JavaScript library used to build user interfaces. It was developed by Facebook. It can also sometimes be seen as a framework because there are several tools that provide developers a good baseline for the project. Despite that, React is not a framework since it does not force certain practices and instead only provides ways to structure the application. Developers can decide how much they want to use it in their project. It is still common to build applications that heavily use React functionalities and React can be used like a framework if needed. React is purely a user interface library so it only cares about how applications are structured and what they look like. It does not define how application logic is structured. This usually requires separate libraries to be used for the application logic.

React uses components that let developers split their user interface to smaller components that can be reused across the application. Components can have their own state and properties that can be used in application logic. Components are defined using JSX which is a syntax extension of JavaScript. This syntax is similar to HTML but it can be mixed with JavaScript logic. JSX can not be used directly in browsers and has to be compiled to actual JavaScript before deployment. Compiled results generate a DOM that has JavaScript in it and that can be run in browsers. Changing and rerendering DOM elements can be expensive and instead of changing the elements in DOM, React uses a virtual DOM. Virtual DOM can be seen as a JavaScript object that is a copy of a real DOM. Since modifying objects is faster than modifying the DOM, this increases the application performance. When something changes in the application, React checks what was changed and then renders only the changes to the real DOM. React also provides databindings that can be used to display object properties or states in the rendered application. These bindings are one-way, which means that data is only passed down in the application tree and can not be passed upward. If developers want to change the data, they must communicate the changes using events. (Facebook, 2019)

2.3.2 AngularJs

AngularJs is a JavaScript framework developed by Google and was released in 2010. Similar to React, it uses a component based structure. Developers can define components and split the logic and look into different files. Angular provides a way to

communicate with the server. It allows developers to specify services that fetch data from servers. Any component can use these services to communicate with the server. Angular provides ways to create automatic tests for the application. The test platform is called Karma. Karma allows developers to test their applications in different devices and platforms. Angular also allows developers to use other testing frameworks to test applications, but Karma is the one that Angular uses as a default test framework.

Angular uses regular HTML and CSS to structure and define components. Instead of JavaScript, Angular uses TypeScript to connect the files and application logic. TypeScript is a superset of JavaScript, which means that it uses JavaScript at its core and adds some features to it. For example, TypeScript adds strongly typed language features to JavaScript, which means that developers have to declare what type each variable is. This helps to avoid bugs and tells other developers what type each variable should be treated as. TypeScript can not be run directly in browsers and it has to be compiled to JavaScript. TypeScript is mainly a tool for developers to structure the application. Since TypeScript is compiled to JavaScript, then valid JavaScript code is also valid TypeScript code. TypeScript has some features (classes, modules and arrow functions) that were later implemented into the JavaScript language. (Google, 2019)

2.3.3 Polymer

Polymer is a JavaScript library developed by Google. It was first released in 2013 and the current version is called Polymer 2. The next version of it is called Polymer 3 which is in prerelease. Polymer allows developers to create reusable components using HTML, CSS and JavaScript. Polymer is based on web components. Web components is a suite of different technologies of the web which include custom elements, shadow DOM and HTML templates. Custom elements allow developers to define custom HTML tags that can be used in web pages, similar to standard HTML tags. A Shadow DOM allows different components to encapsulate their logic. This avoids conflict between components and makes them easier to reuse. (MDN Web Docs, 2019e) Polymer components are declared using HTML templates that can be added to web pages and applications using HTML tags. Polymer also introduces HTML imports that allow developers to include HTML files into other HTML documents. Unlike React and Angular, Polymer does not use its own implementation for the components and instead uses features directly supported by browsers. Currently not all browsers support all the

features required by Polymer so it automatically polyfills the missing features. Polymer also provides two-way binding which means that data can flow both ways without using events. Polymer is mostly designed to allow developers to use web components and in the future it will not be needed after browsers implement the necessary features. (The Polymer Project Authors, 2019)

2.4 HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is a protocol that is used to communicate in networks. Its predecessor is called Hypertext Transfer Protocol (HTTP), but it was not secure enough for sensitive applications and, therefore, Transport Layer Security (TLS) was implemented to provide more secure communication. HTTPS uses both of these technologies and can be referred to as HTTP over TLS. (Rescorla, 2000)

2.4.1 HTTP

HTTP was first introduced in 1990. It is a stateless protocol that can be used to communicate over the internet. Stateless protocol means that each request is always its own instance and no data is shared between requests. Each request should be handled individually and without being affected by other concurrent requests. Most of the HTTP requests are triggered by the client when it sends requests to the server and the server responds to the request based on given information. This information may consist of a request method, URL information or body that contains user specified content that can be in multiple forms. It can be for example text, static resources, like images or objects.

The method of request usually defines what kind of data can be sent or received. The most commonly used request methods are GET, POST, PUT and DELETE. The GET method is used to retrieve data and no data should be sent with the request. This is the most commonly used method, since it is used to fetch web pages or data from the server. POST is used to post data to server. This can be, for example, when adding a user to a database. PUT is similar to it but is used to update existing data. The POST and PUT methods usually include some kind of body which includes necessary information to add or update the data. For example, a unique id that is used to distinguish different entities in a database. The DELETE method is used to delete existing data from a database.

When a request is processed, the server responds with some kind of status code. The most common is code “200 OK”, which means that the request was processed successfully. When servers encounter an error, then some other status code is sent. Error codes are between 400 and 410. Status code “400 Bad Request”, means that the client made a request the server could not process because it was possibly missing necessary data. When using authentication in applications, then status code “401 Unauthorized” means that user is not authenticated and because of this the request can not be processed. When the user has authenticated but has no permission to complete some action then error code “403 Forbidden” is returned. If server completely fails to process requests because of possible errors in the server code, then it can return “500 Internal Server Error” which means that server failed to fulfill the request. This should almost never happen in ideal situations and it is good practice to always return meaningfull error codes when encountering a errors. (Fielding, Gettys, Mogul, Nielsen, Masinter, Leach & Berners-Lee, 1999)

2.4.2 TLS

The goal of the TLS protocol is to provide security between communicating applications. It uses symmetric cryptography to encrypt the data and, therefore, it can not be decrypted without a proper encryption key. Keys are generated uniquely for each connection, which means that same key can not be used to decrypt other connections. This fulfills the main goal of TLS, which is to provide secure connections between parties. TLS also allows programmers to extend and incorporate their own solutions to it. This eliminates the need to develop new protocols, which risk introducing new security vulnerabilities. When using HTTP over TLS, a proper certificate must be provided. Certificates contain public and private encryption keys. Certificates can also be used to confirm websites or application indentity to make sure the server really is what it claims to be. When a certificate is uploaded to the server, then developers are able to switch to HTTPS by using “https” at the start of the url instead of “http”. (Dierks & Rescorla, 2008)

2.5 Polling

There are two types of polling, short polling and long polling. In short polling, the client sends a request to the server and the server responds with available data. After the client

receives response from server, the client waits before sending a new request. This can be interval-defined by the application developer or a request can be sent as soon as a response is received. Because this technique usually involves continuous requests, it can cause the server to overheat, especially when multiple different clients try to request the same data. Requests are performed even if there is no new data in the server so a request is practically done for no reason.

Long polling tries to solve this issue. Instead of always returning data upon request, the client keeps the connection open and waits until server detects an update and sends the data. After the response, the client sends a new trequest and waits for new changes. This reduces unneeded requests. The problem with this method is that it still introduces latency, since new connections need to be established after response. Clients send the same request headers and data as before and the server needs to parse it again before returning new responses after an update has occurred. There is also a problem that requests might timeout if it takes too long for an update to occur. Clients can choose to timeout the request earlier so new requests can be done. Timeout should be chosen carefully, since servers can have varying timeouts and can return “408 Request Timeout” status code if requests take too long. Making timeouts too small can introduce unnecessary overheating to the server—similar to short polling. (Loreto, Saint-Andre, Salsano, & Wilkins, 2011)

2.6 Server-sent events

Server-sent events are a method for HTTP to keep connections open and it allows servers to send data to a client when the connection is open. Servers can send any data to the client and clients can then handle the received data. Server-sent events are unidirectional, which means that events can only be sent from server to client. Because the connection stays open, no new request is created and there is no overhead for establishing new connections compared to polling. This reduces latency since server events are always sent and they don’t need to wait for a previous event. This reduces the risk of data coming in the wrong order or pieces of data being skipped. The request can not timeout, since the server is aware that the connection should stay open and the server can send “keep-alive” messages to keep the connection open. If a connection is interrupted, then the client automatically tries to establish a new connection. Browsers have a limit on how many connections can be open and this can cause problems when

using server-sent events. If too many connections are open, the new ones can not be established until the old ones are closed. If connections have the same URL, then they can share the same source and this frees the connections for other use. Currently, the server-sent events are supported by all browsers except Microsoft Edge and Microsoft Internet Explorer. If developers want to use server-sent events in these browsers, then polyfill is required. Polyfill is a piece of code that implements the feature using code instead of using the browser implementation. (Whatwg, 2019)

2.7 Websocket

Websockets provide two-way connections between server and client. This means that the server can send events to the client and the client can send events to the server. Websockets first establish initial connections, called handshake, with an HTTP request. After that, the connection is kept open and data can be sent both ways. Websockets have many similiarities with server-sent events and also suffer from the same problems. Websockets only allows a limited number of connections and has varying support for browsers. Websockets have no way to automatically reconnect if a connection is lost and reconnection is left for the developer to implement.

Websockets use Transmission Control Protocol (TCP) to communicate between client and server. TCP protocol provides error free ways to send and receive data. TCP sends data in packets. If any data is lost during the transmission, it is detected and sent again with the next packet. It also makes sure that data always comes in the same order, regardless if packets are lost. TCP also has flow control, which limits how much data can be sent at once and will automatically send remaining data with the next packet. (Fette & Melnikov, 2011)

2.8 Flux

Flux is an application architecture pattern that is used in the React library. Flux has 3 major parts: dispatcher, store and view. Flux has unidirectional data flow, which means that data only flows one-way and changes to data can only be done using events, which are also called actions (Figure 1).

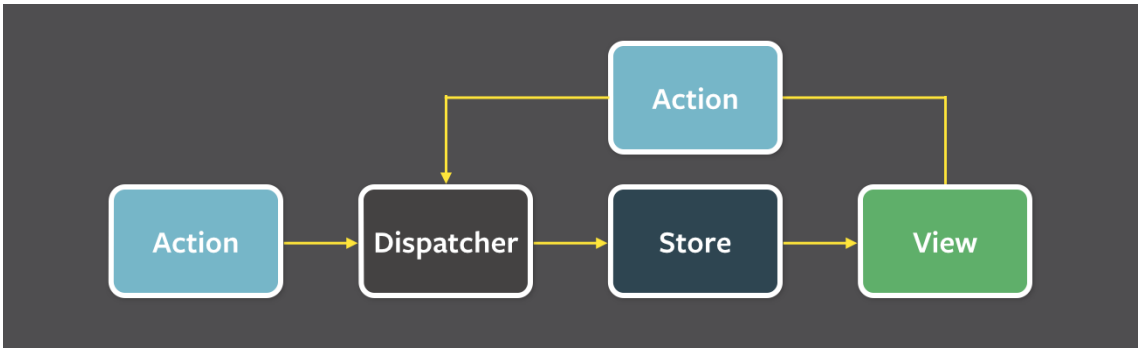

Figure 1. Flux application flow.

Actions are triggered by other actions, like when requesting data from a server or when users interact with the views. Actions usually have type and payload. Type is used to define what the action should do and payload is the data that is sent with an action. Dispatchers send the actions to the store. The dispatcher does not know what actions do or what action should trigger in the store. Dispatchers only distribute the actions to stores. Stores have callbacks which are triggered by actions. The callbacks can modify the data in store. After modification, the store sends change events that views listen to. When a view receives the change event, the view updates its data so changes can be shown to the user. (Facebook, 2015)

3 METHOLOGY

This section includes the methology of the thesis. It explains what was decided at each step, what technologies were used and why they were used. This section only includes abstract of the created system and doesn't include the implementation. To understand some figures fully, basic knowledge of JavaScript is required.

3.1 Application

The application is a user interface that is run in browser. The application is used to monitor the status of trains in real-time and perform actions to them. User interface consist of 4 views that display different data. All views share the same layout but have different actions. The layout consists of an application header, toolbar and table that displays a list of all connected devices (Figure 2). The header contains the navigation used for changing views.

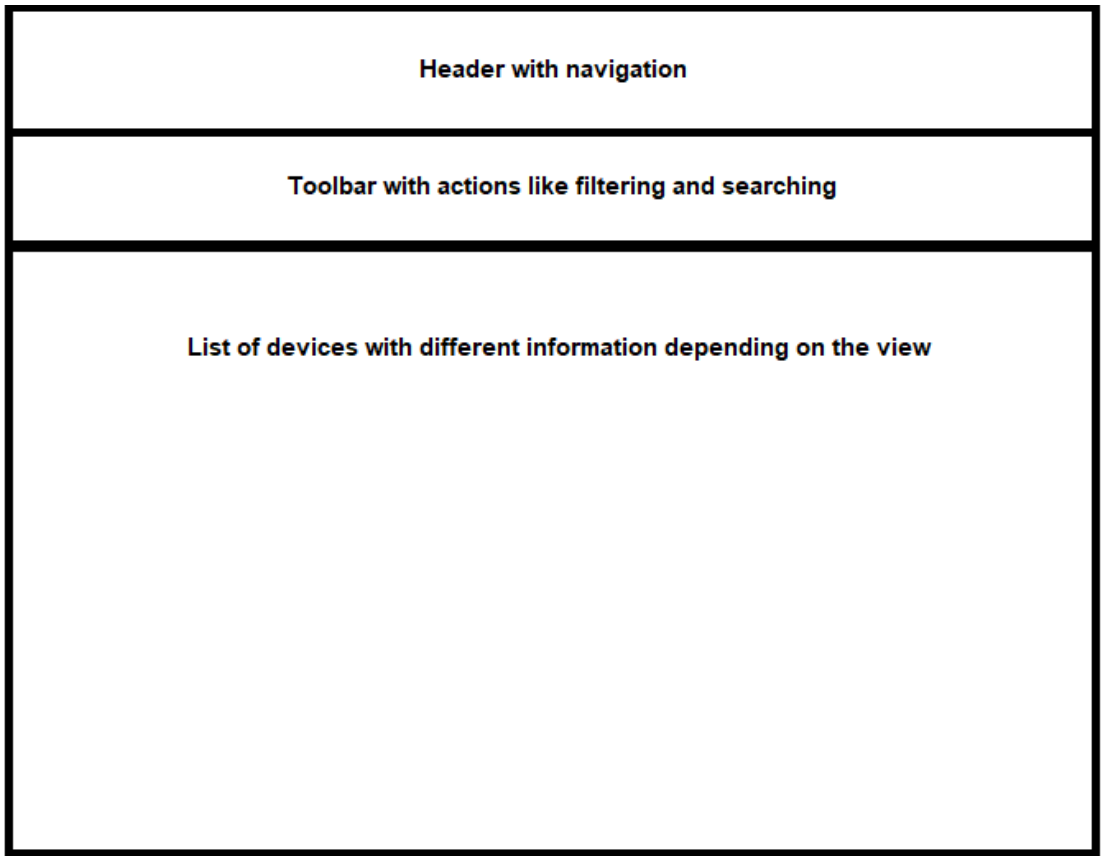

Figure 2. Application layout.

The tables have different columns depending what view is active. All views share search functionality that can be used to search devices with their name or id. Depending on the columns in each views table, approriate filters are displayed in the toolbar. Filters are displayed as dropdowns with multiple options. These options can be for example in case of device status: offline, online or unknown. The toolbar might contain buttons that can be used to perform actions to multiple devices. These buttons are disabled when no device is selected in table. The table contains checkboxes in first column that can be used to select devices. Performing an action usually triggers dialog that asks users to confirm the action or fill some information. In case of renaming the device, dialog would display the input field where user inputs new device name.

3.2 Selecting framework and libraries

The company had 3 frameworks that could be used for creating the application. These were React, Polymer and AngularJS. React is not technically a framework but for clarity it is treated as such. The application had 4 views that displayed different data. These views are very similar so it is important that components can be reused across the views. Views have large lists that can contain a lot of updates to data so it is important that the selected framework is able to handle these updates efficiently. All 3 frameworks are component-based and components can be reused in different places. Out of 3 frameworks, React is most efficient of updating large lists of data since it uses Virtual DOM to efficiently only update the data that was changed. Polymer and Angular use regular DOM, which is slow when updating large lists of data. Polymer has problem that it uses web component technologies that not all browsers support yet and because of this, features need to be polyfilled and this can cause slowdowns. Angular defines very clearly how everything need to be done and is usefull in large projects, but since this project only involved 2 developers, 1 doing user interface and other doing the server, it was more of a restriction than a benefit. It was decided that React is used during the project.

React only contains view part of the application. Because of this, some other libraries and technologies were needed to be included to project. Team used create-react-app (CRA) to create the project base and this automatically includes the necessary tools to test and build the application. This includes Webpack for application bundling, Babel for transpiling the code, ESlint to enforce the coding style and Jest testing framework to

create automatic tests for the application. For standard HTTP requests, the project group decided to use Fetch API. Fetch is native solution to perform HTTP requests and is very common when using React. It is not supported by Internet Explorer but it can be polyfilled in that case. Customer also didn't use the Internet Explorer in their systems so even if there is slight slowdown due to it, it should not affect the usability. For routing the team used react-router which can be used to read and update the url in browser address bar.

3.3 State management

React components can have state to hold data, but the application uses the same data across the views so the team decided to implement global state to hold all the data. For this, team decided to use Redux. Redux is state management library for React. It implements Flux patterns. Redux has 3 core principles: single source of truth, read-only state and pure function changes. Single source of truth means that there is a single store that holds the application data and this data can be shared with all components. Read-only state is similar to Flux patterns store, where only actions are used to implement the store changes. States can only be modified with actions and Redux also enforces that these actions are pure functions. This means that functions only depend on their own input and not from any outside source. (Abramov, 2018)

Redux has functions called reducers, which listen to actions and perform changes to states based on action type. This is usually done using switch statement (Figure 3).

```
const myReducer = (state = initialState, action) => {
    switch (action.type) {
        case LOGIN:
            return {...state, loggedIn: true};
        case LOGOUT:
            return {...state, loggedIn: false};
        default:
            return state;
    }
}
```

Figure 3. Selecting action using switch statement.

Reducers always return new state since in Redux state can't be modified so new state has to be created each time action happens. Components can send actions to reducers when for example button is clicked. Redux has standard format for actions. (Kurian, 2019) Developers can use any format they choose, but the team decided to use the redux-actions library to create flux standard actions. These actions contain objects, which have type and payload properties and optional error and meta properties. (Figure 4) Redux-actions also reduces boilerplate needed to write the code by providing simple action creators, which create the object automatically based on given parameters. (Cheung, 2018)

```
{
    type: 'LOGIN',
    payload: {
        data: 'somedata'
    },
    error: false,
    meta: 'extra information'
}
```

Figure 4. Flux standard action object structure.

Sometimes there might be large amounts of data in store and user needs to retrieve only part of the data that is used in application. Selectors can be used for this. Selectors can be used to compute data from store. They can be used for example to find items with given id from list. This reduces duplicated data and makes the state smaller. The team used a library called Reselect which optimizes selectors. It uses memorization to store the selector data, so complex selectors don't need to be computed again if nothing has been changed. Memorization is way to cache data so if parameters given to selector are same then the old data can be returned without computing it again. Data is recomputed only when parameters change. This increases performance when handling large lists. For smaller lists, the performance is not very significant since list search done by computers is already very fast, but when dealing with list over 10 000 items, it can have an effect on usability. (Erikson, 2019)

3.4 Server communication

Since this is as real-time application, the team had to choose method how client and server communicate changes in system efficiently. Long polling has many problems so that option was dismissed. The remaining options were server-sent events or websockets. Most of the application logic was to receive data and there was only few actions that required sending request to server. This could be implemented with servers-sent events or websockets but team decided to use server-sent events because most of the communication was done from server to client so implementin websockets in server would not add anything to application. JavaScript provides Eventsource object for server-sent events. (MDN Web Docs, 2019c) Eventsource is not supported in Microsoft Edge and Internet Explorer so the team used polyfill called Yaffle/EventSource. (Yaffle, 2019) Eventsource API allows developers to listen for events from a given url. Developer declares Eventsource object and then can listen for events from it using onmessage function. (Figure 5)

```
const ev = new Eventsource('/api/source');
ev.onmessage = function(event) {
    // Event data here
}
```

Figure 5. Creating Eventsource and listening events.

There was an already exisiting implementation where server sends change event with information about the changed data. Clients can then fetch data from relevant url. It would be more optimal to send new data using Eventsource, but for quicker implementation, the existing solution was used. Application has 4 views so each view has a main endpoint that can be used to fetch data. When Eventsource sends event and if that events type matches the active view in user interface then client is instructed to fetch the data again. (Figure 6) The views are stored in list so they can be referred using data received from Eventsource.

```
const ev = new Eventsource('/api/changes'):
ev.onmessage = (event) => {
    const data = JSON.parse(event.data);
    const view = data.view;
    // Event from different view so do nothing
    if (view !== currentView) {
        return;
    }
    // Already fetching so cache for later use
    if (fetching) {
        cache = view;
        return;
    }
    switch(view) {
        case 'monitor': {
            views[view].fetch();
            break;
        }
        case 'messages': {
            views[view].fetch();
            break;
        }
    }
}
```

Figure 6. Fetching data when event is received and caching if fetching is already in progress.

If user changes view then data is always fetched again. This solution requires extra roundtrip to the server since a new connection has to be established and if there are quick changes that happen before new data is fetched then client might miss the latest changes and this causes the client to not be in sync with the server. Quick changes might cause multiple requests at the same time to be sent and this can reduce the responsiveness of the client. To improve this, the client is not allowed to fetch new data if fetching is already in progress. Instead, the latest change is cached and once the initial fetching is done then the new request is created based on the latest change event. This keeps the client in sync with server and reduces the overhead during quick changes.

The client needed to send some actions to server. Since the team decided to use Eventsource, the client can't use it to send events to server. Instead client used standard HTTP requests to send data to server. These requests usually triggered change in server

and caused server to send updates through Eventsource. This was deemed as acceptable since the client usually couldn’t predict the effects of the action so some response from server was required. Server didn’t return any relevant data with HTTP response. Server only returned if action was successfull. If something went wrong then the server returned messages that explained what went wrong. Messages could for example include what values were invalid or if action could not be performed to device. Reasons for failed actions could be for example if device was disconnected from the server and could not establish new connection.

4 RESULTS AND DISCUSSION

This section contains the results and discussion of the thesis. It outlines what was discovered during the project and what was accomplished. The section explains what the results mean and how these results can be used in future projects.

4.1 Performance

The application was tested using 1000 devices and 10 000 devices that were simulated to mimic the real devices. Testing the server contained data that resembled the data used in production. The tests didn't take into account the communication between server and the device and instead focused on client and server interaction. Target refresh rate measured in frames per second (FPS) was 60. The performance was measured using Chrome developer tools.

With 1000 devices the render was nearly instant and there was no significant delay when data was received from server. When the server updated some data and the client fetched the new data, the changes in table didn't cause any major FPS drops. Around 4 megabytes of memory was used by browser. If there were multiple quick changes in short intervals, then the client might have missed some changes in between. For example, normally when user sends message to device, the status changes from "queued" to "executing" and at the end to "done" status. If execution was quick, then the client may never see the "executing" status in the device and instead it skips straight to "done" status. This was expected when deciding the server communication method and didn't cause the client to go out of sync with server.

With 10 000 devices the issues with render started to show. FPS dropped to around 40 when scrolling the created list of devices. Memory usage was around 20 megabytes. This was mostly caused by the amount of rendered rows that were generated to the table. When devices were updated, the updates were again nearly instant since only relevant data was updated. The amount of updates didn't affect the rendering performance significantly. Most of this is because most of the data was cached and reused between updates. Similarly to the 1000 device test case, the changes could be missed with quick updates. This happens more frequently the more devices there are but in the end the user interface always ends up matching the server.

4.2 Improvements

The main problem found during the testing were that performance dropped when number of devices was high. The other problem was that changes were not updated in time when changes were done to the same device in quick succession. The reason that a large number of devices reduces performance is that browser has to render 10 000 rows of data even if only 10 rows are seen. The way to solve this would be to calculate visible rows and then only render them. This is also known as virtualisation. Libraries that can do this in React are for example react-virtualised and react-window. These libraries render lists of data and calculate the visible rows and only render those. This reduces the memory usage and amount of render, so performance is mostly unaffected even if the amount of data is increased. The method is similar to occlusion culling used in games where objects that are not visible in games are not rendered to save resources.

The problems with quick changes were expected during the project. The implementation was not optimal because of the schedule of the project. The reason for this problem is that server-sent events are used to tell the client that changes have occurred. This causes the client to send a new request to server to fetch the changed data. The better way would be to send the changed data with the server event and make the client update the data. For example, the event could contain the type of item so client could determine where the update occurred, id of item to find the changed item in list and data that was changed for the item so client can update the data. Since Eventsource establishes a constant connection to the server, it would not suffer from as much delay as standard HTTP request since no new connection is established and only changed data is sent instead of all the data. This would also reduce the amount of calculations when clients are determining the changes, but since the calculations were already very fast, the performance would not improve much.

4.3 Usage in games

The same methods for the user interface structure and server communication can be applied to game development. Games have to preferably render at 60 frames per second, similarl to target frames per second that was used in the project. This applies to the game world itself and also to the user interface. All the interface elements need to get the data from somewhere and it’s usually most efficiently done when all the data is

stored in the same place. Storing all the data in the same place reduces data duplication and saves memory. It's also important to try to reduce the computation of each frame by caching the computed data. For example, if an item is found from the list then it should be cached so following frames don't have to compute the data again. This might duplicate data but can improve the performance at the cost of memory. It's also important to reduce the amount if elements that are rendered to the screen. In game world this is done using occlusion culling algorithms. For the user interface, it's best to disable elements that are behind other elements since not all game engines perform occlusion culling to user interface elements.

Games more commonly use sockets instead of server-sent events since games often require two-way communication. Sockets are usually reserved for communicating game actions that have high priority while playing the game. Depending how much data is sent with the socket, it can be most optimal to also send other actions via the socket. If there is a lot of data for example in real-time strategy games with many moving units, it can be best to have 2 separate sockets; 1 for the high priority actions and other for the rest. If standard HTTP is used then for the user to feel like actions are instant, the user interface can make predictions how the data is gonna be handled and then update user interface before the actual request is processed. If errors occur during the request then the data should be reverted back to the previous one and the client should be notified with the error message.

5 CONCLUSION

This thesis project was carried out to figure out what was the best way to create real-time user interface and how these techniques could be used in other fields, such as games. When syncing the changes to the client, it is important to minimize the number of changes that are performed when new data is received. It is also to minimize the number of elements in the screen so large datasets can be managed without affecting the user experience. When communicating with the server, the number of connections should be minimized and connections should be preferably kept open so all the relevant data can be communicated without delay. It is also important to not open too many connections and to try to handle most of communication in single open connection. All received data should be cached so same data can be rendered quicker when there are not changes to data. This is especially important when working with large dataset. Game user interfaces can also benefit from the similar methods since game user interface have much in common with real-time user interfaces. Games often run at 60 FPS so it is important to reduce computing power so constant FPS can be maintained.

The project did not test the efficiency of the sockets versus the server-sent events so it is not clear which one is more efficient with large real-time datasets. The project was limited to specific frameworks and technologies and it is possible that more efficient technologies exist in different platforms or programming languages. The project focused mainly on technologies in front-end and did not consider what techiques could be used in the back-end side of the application. The practices were not tested in gaming environment but they should be applicable since games have much in common with standard real-time applications.

REFERENCES

Abramov, D. 2018, 12.9.2018-last update*, Getting Started with Redux*. Available: https://redux.js.org/introduction/getting-started [2019, 4/30].

Babel 2019*, What is Babel?*. Available: https://babeljs.io/docs/en/ [2019, 4/30].

Cheung, T. 2018, *, Introduction - redux-actions*. Available: https://redux-actions.js.org/introduction [2019, 4/30].

Dierks, T. & Rescorla, E. 2008, 1.8.2008-last update*, RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*. Available: https://tools.ietf.org/html/rfc5246 [2019, 4/30].

Erikson, M. 2019, 21.1.2019-last update*, Reselect*. Available: https://github.com/reduxjs/reselect [2019, 4/30].

Facebook 2015, *, Flux - In Depth Overview*. Available: https://facebook.github.io/flux/docs/in-depth-overview.html [2019, 4/30].

Facebook 2019*, Getting Started - React*. Available: https://reactjs.org/docs/getting-started.html [2019, 4/30].

Fette, I. & Melnikov, A. 2011, 1.12.2011-last update*, RFC 6455 - The WebSocket Protocol*. Available: https://tools.ietf.org/html/rfc6455 [2019, 4/30].

Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. & Berners-Lee, T. 1999, 1.6.1999-last update*, RFC 2616 - Hypertext Transfer Protocol -- HTTP/1.1*. Available: https://tools.ietf.org/html/rfc2616 [2019, 4/30].

Google 2019*, Angular - Getting Started*. Available: https://angular.io/guide/quickstart [2019, 4/30].

Harder, D., Zarnett, J., Montaghami, V. & Giannikouris, A. 2014, . Available: https://ece.uwaterloo.ca/~dwharder/icsrts/Lecture_materials/A_practical_introduction_to_real-time_systems_for_undergraduate_engineering.pdf [2019, 4/30].

Kurian, J. 2019, 15.4.2019-last update*, Flux Standard Action*. Available: https://github.com/redux-utilities/flux-standard-action [2019, 4/30].

Loreto, S., Saint-Andre, P., Salsano, S. & Wilkins, G. 2011, 1.4.2011-last update*, RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP*. Available: https://tools.ietf.org/html/rfc6202 [2019, 4/30].

MDN Web Docs 2019a, 18.4.2019-last update*, MDN Web Docs Glossary: Definitions of Web-related terms - CSS*. Available: https://developer.mozilla.org/en-US/docs/Glossary/CSS [2019, 4/30].

MDN Web Docs 2019b, 3/18/2019-last update*, MDN Web Docs Glossary: Definitions of Web-related terms - HTML*. Available: https://developer.mozilla.org/en-US/docs/Glossary/HTML [2019, 4/30].

MDN Web Docs 2019c, 23.3.2019-last update*, Using server-sent events*. Available: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events [2019, 4/30].

MDN Web Docs 2019d, 22.4.2019-last update*, Web technology for developers - JavaScript*. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript [2019, 4/30].

MDN Web Docs 2019e, 3.3.2019-last update*, Web technology for developers - Web Components*. Available: https://developer.mozilla.org/en-US/docs/Web/Web_Components [2019, 4/30].

Rescorla, E. 2000, 1.5.2000-last update*, RFC 2919 - HTTP Over TLS*. Available: https://tools.ietf.org/html/rfc2818 [2019, 4/30].

The Polymer Project Authors 2019*, Polymer library - Polymer Project*. Available: https://polymer-library.polymer-project.org/2.0/docs/devguide/feature-overview [2019, 4/30].

Whatwg 2019, 29.4.2019-last update*, HTML Standard - Server-sent events*. Available: https://html.spec.whatwg.org/multipage/#server-sent-events [2019, 4/30].

Yaffle 2019, 9.2.2019-last update*, EventSource polyfill*. Available: https://github.com/Yaffle/EventSource [2019, 4/30].