University of Applied Sciences and Arts
Northwestern Switzerland

Bachelor-Thesis

Event Based Real-Time Synchronization of Web Applications

Robin Christen, Etienne Gobeli

March 20, 2020

Supervisors:	Prof. Dierk König
	Dr. Dieter Holz
Experts:	François Martin
	Marco Sanfratello
Profile:	iCompetence

Abstract

Real-time synchronization has become more and more relevant due to the increasing demands on multi-user and collaboration software with the underlying idea that information can be shared instantly across different clients. Implementing this feature in a web environment brings various limitations and challenges that need to be overcome. In this project, we aimed to solve the problem that the application state is typically displayed in a web browser based on one point in time by introducing possibilities that allow a server to push messages to a client without an explicit request. This turned out to be more challenging with the requirement for a small unit of edit, which resulted in the need for offline capabilities to ensure fast processing. Along the way, we have discovered how an event-driven approach can help to solve the problem of merging several divergent but equivalent sources of information to eventually converge to a consistent state. This thesis documents our findings, their advantages, limitations and impacts on usability.

Contents

Summary **5**

1 Introduction **6**

1.1 Motivation . . . 6
1.2 Objective . . . 6
1.3 Problem Statement . . . 6
1.4 Methodology . . . 7
1.5 Scope . . . 7

I Technical Report **8**

2 Web Environment **9**

2.1 Client . . . 9
2.2 Server . . . 9
2.3 HTTP . . . 10

3 Two-way Communication **11**

3.1 Short Polling . . . 12
3.2 Long Polling . . . 14
3.3 Server-Sent Events . . . 17
3.4 WebSockets . . . 20
3.5 Comparison . . . 23

4 State Distribution **25**

4.1 Single Master . . . 25
4.2 Multiple Masters . . . 28
4.3 CAP Theorem . . . 29
4.4 Conclusion . . . 29

5 Conflict Management **30**

5.1 Conflict . . . 30
5.2 Prevention . . . 30
5.3 Convergence . . . 31
5.4 Ordering . . . 32

6 Event-Driven **33**
6.1 Event Sourcing . . . 33
6.2 Stream Processing . . . 35
6.3 Advantages . . . 35
6.4 Limitations . . . 36

7 Conflicting History **37**
7.1 Resolution . . . 37
7.2 GUI Adjustments . . . 41
7.3 Projection Conflicts . . . 41
7.4 Impacts . . . 42

8 View Rendering **43**
8.1 Data based . . . 43
8.2 Event based . . . 45

9 Usability **46**
9.1 Coupled View . . . 46
9.2 Impact of Conflicts . . . 47
9.3 Offline Management . . . 48
9.4 Too Much Real-Time . . . 49

II Proof of Concept **50**

10 Project **51**
10.1 Requirements . . . 51
10.2 Iterations . . . 52

11 Architecture **53**
11.1 Event . . . 53
11.2 Store . . . 53
11.3 Scheduler . . . 54
11.4 API . . . 54
11.5 DOM Projection . . . 55
11.6 Controller . . . 56
11.7 View . . . 56
11.8 Overview . . . 57

12 Implementation **58**
12.1 End-to-End Events 58
12.2 View Rendering 59
12.3 Conflict Management 60
12.4 Server 61

13 Conclusion **62**
13.1 Outlook 63
13.2 Reflection 65

III Appendix **66**

References **67**

List of Figures **68**

Glossary **69**

Declaration of Honesty **70**

Summary

With the advent of multi-client and collaboration software, real-time synchronization has become an important requirement. Users expect their actions to be instantly and automatically distributed across multiple clients, and there is already software that support this functionality. Paradoxically, however, web applications are still lagging behind this trend.

The problem is that web browsers have traditionally been designed to be mostly stateless with the purpose of viewing data at one point in time. They communicate based on the request-response pattern, making it difficult to receive unrequested messages. In this thesis we explored different approaches to provide real-time functionality for the web that can eliminate the manual process of reloading a page. The idea was to gather knowledge by researching various methods and applying them to realistic, practical examples.

It turns out that it is possible to bypass the limitations of receiving messages without an explicit request by using polling, streaming techniques or by switching to the WebSocket protocol. Yet the challenge takes on a new dimension when the goal is to provide real-time capabilities where the unit of edit is small, such as a single keystroke instead of a complete form submission. It pushes typical architectures that maintain state on a server to its limits, as the latency of a server round-trip can lead to a bad user experience.

To overcome these problems, the state must be managed locally on each client instead of on a single server. This eliminates the need to validate the changes on a central instance, providing speed advantages and even offline support. However, this results in different copies of equally valid states, and to eventually achieve convergence, these states must be synchronized with each other and potential conflicts must be resolved.

In this project an event-driven approach is used to solve the problems of real-time synchronization. Along the way, we discovered its advantages, its limitations and the usability trade-offs that need to be considered. The results are documented in this report and the practical demonstrations are available on GitLab[1].

[1]https://gitlab.fhnw.ch/p6-christen-gobeli

1 Introduction

The Internet enables the connection of applications and the exchange of data between them. As a result, software can now span across multiple systems. This raises new challenges for developers. One of these challenges is the synchronization between the different clients. In this thesis we discuss the difficulties of real-time synchronization in a web environment.

1.1 Motivation

Nowadays, a significant part of the computing power and data storage is no longer handled on personal computers, but in data centres accessible over the Internet. This is commonly known as cloud computing, which has led to an increase in the development of multi-client and collaboration software. Users expect to be able to use multiple devices for the same application state and to collaborate with other users. Therefore, sending and receiving updates in real-time is an important requirement. One way to interact with such a system is via a web application. These applications run in a browser, which limits their capabilities and making it difficult to provide a live experience.

1.2 Objective

With this project we want to explore different possibilities to solve the problems described in the next chapter (1.3) for web applications. The main goal is to acquire knowledge and to apply the accumulated knowledge in practice. This should result in a deeper understanding of the challenges, their implications and ideas for possible solutions.

1.3 Problem Statement

A typical web application consists of a server and multiple clients. Each client reads the state from the server and displays it in the browser. The problem is that this only happens at one point in time. After that, the client is usually static and only finds out about a potential change on the server after an update has been requested. In practice, this can be achieved for example by reloading the website. This is especially a problem when changes occur quickly and interested clients need to be up to date immediately.

Receiving updates To provide a live experience, the server must send updates to the client without the client explicitly requesting them. The communication between the client and the server must be bidirectional, so that both the client and the server can send and receive messages.

Consistency Every client should always display the correct information. Possible conflicts must be resolved in such a way that in the end everyone ends up with the same state.

1.4 Methodology

To achieve our defined goal from chapter 1.3, we approached the problem in two steps. A technical and a practical part. These parts were not created one after the other, but together. During the project ideas and approaches were researched, implemented and validated. The result was documented across the two sections, which are intended to complement each other.

1.4.1 Technical Report

Part I consists of our considerations, challenges, investigations and possible approaches to solve the problem of real-time synchronization in web applications. This section is mainly theoretical and not bound to a specific example. Its purpose is to form a better understanding about the topic.

1.4.2 Proof of Concept

Part II documents how we applied the knowledge in practice on an example project. This project showcases the insights we accumulated and consists of different technologies and approaches.

1.5 Scope

This project focuses on different approaches and the underlying ideas of real-time synchronization. For educational reasons we did not want to use any library or framework which already solve parts of the problem. This also has the advantage that the stack is simple and we as developers retain full control. Anything which is not explicitly needed in order to provide live synchronisation is considered out of scope.

Part I
Technical Report

2 Web Environment

An application that runs in a web environment is usually built using the client-server architecture. This makes it a distributed application in which the parties must be able to communicate with each other.

2.1 Client

A client is the part of a web application that runs locally in a browser. Traditionally, browsers have been mostly stateless and designed only to access information on the World Wide Web. In recent years, it has become popular to use JavaScript to interact with the User Interface (UI) which allows an application to modify the view without the need of an internet connection. This enables a web application to feel more like a desktop application and not just a viewer of Hypertext Markup Language (HTML) files. However, some limitations still remain. For example, the communication is mainly synchronous, forcing a client to request changes instead of being notified. This topic is further discussed in chapter 3.

2.2 Server

In a traditional web application, the server is often considered to be a presentation server and is responsible for maintaining an application state. It accepts requests from clients, processes them and returns information about the result of the request. As clients evolve to become more than just a viewer of results, different types of data to be transmitted increased in popularity. Instead of returning HTML files, JavaScript Object Notation (JSON) is now a widely used format to exchange data.

2.3 HTTP

The Hypertext Transfer Protocol (HTTP)[2] is the main communication protocol used in a web environment and it is based on the request-response pattern (2.3.1). In addition to the payload, each request on a resource includes a method[3] to inform the server of the intent, and each response contains a status code[4] to inform the client about the result of the request [Gri13].

TCP/IP HTTP mainly uses the Transmission Control Protocol/Internet Protocol (TCP/IP)[5] as the underlying transport protocol which guarantees that if the data is delivered, it is complete and in the correct order. It uses acknowledgement messages to inform the sender about a successful transmission [Gwe18]. For this reasons, TCP brings important features when trying to achieve consistency across multiple clients.

2.3.1 Request-Response

The request-response messaging pattern is a basic method that allows an issuer to send a message and receive a corresponding reply. HTTP allows a client to request something from a server and receive a response about the result (1). For security reasons, the client must always initiate the request and it is not possible for the server to start the HTTP connection. This restrictions limits the possibilities to push updates to the clients without their explicit request. However, it can be bypassed with some techniques that are discussed in chapter 3.

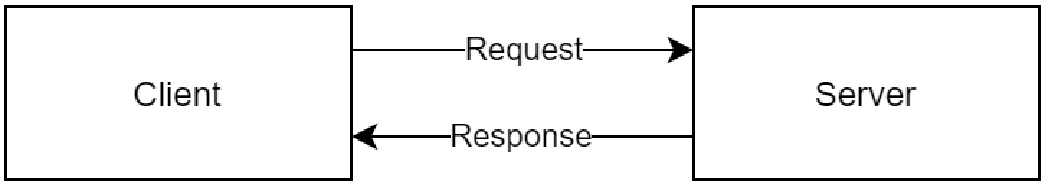

Figure 1: HTTP's request-response pattern

[2]https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[3]https://developer.mozilla.org/de/docs/Web/HTTP/Methods
[4]https://developer.mozilla.org/de/docs/Web/HTTP/Status
[5]https://en.wikipedia.org/wiki/Internet_protocol_suite

3 Two-way Communication

An important requirement to enable real-time synchronization between clients is not just the ability to send updates but also to receive changes without an explicit request. In chapter 2.3 was mentioned that web applications use the HTTP protocol to communicate between client and server, which is based on the request-response pattern.

3.0.1 Client Request

Methods for sending updates to a server using HTTP are well-known. For example, this can be achieved by using a REST[6] architecture. Since these approaches are well-established and not part of the problem, this thesis will not go into this topic in detail.

3.0.2 Server Push

Conversely, the server does not have the ability of pushing data directly to the client. Different approaches exist to bypass these limitations and allow clients to update automatically. The rest of this chapter discusses the various options available. The following principles are going to be explored:

- HTTP Polling
- HTTP Streaming
- Beyond HTTP

Code Disclaimer The following chapters contain code examples. These examples are by no means meant for production. They are simply ideas, prototypes or skeletons to demonstrate the underling principles with the goal to provide a better understanding of the explored concepts.

[6]https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

3.1 Short Polling

The simplest approach is to use a technique called short polling (2). The idea is, that the client automatically and periodically asks for updates from the server. This basically automates the manual reloading of a website but without having to re-render the entire page. Ideally, the response would be empty unless new data is available but this requires some extra considerations.

Interval A critical part is to decide on the polling interval. Long intervals translate to a delayed delivery of updates, whereas short intervals result in unnecessary traffic and high overhead both for the client and the server. (An average HTTP request adds about 850 bytes of request and response overhead [Gri13].)

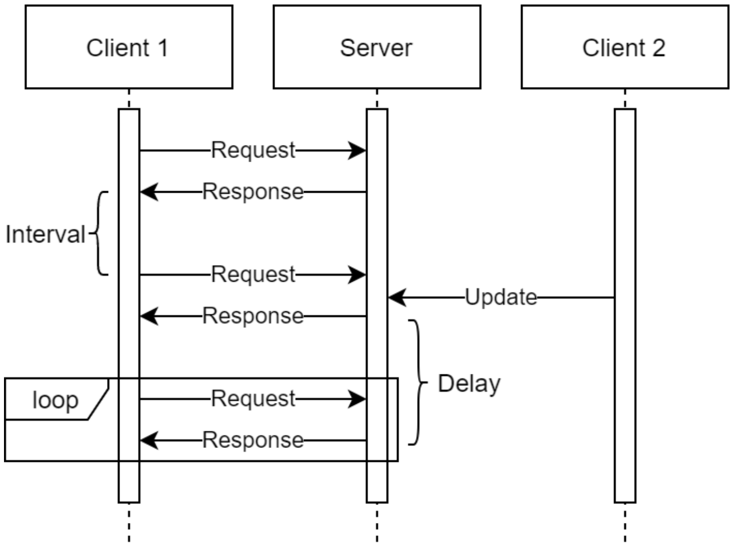

Figure 2: Sequence diagram for short polling

3.1.1 Example

The following code shows a skeleton of how the client could implement short polling with an interval of one second.

```
  const poll = async () => {
    const response = await fetch('https://puerro.io/api');
    // handle response
  };
  setInterval(poll, 1000); // repeat every second
```

The server can provide a typical HTTP endpoint that sends a responds and then closes the connection.

```
http.createServer((req, res) => {
  res.setHeader('Content-Type','text/plain');
  res.end('Puerro'); // send response and close connection
}).listen();
```

3.1.2 Advantages

The biggest advantage of short polling is that it is easy to implement. It can be implemented on existing API's without the need to change anything on the server by simply calling the endpoint periodically, even when the user has not been interacting with the application. The server can close the connection immediately after the response and does not need to provide any functionality to detect changes or track outdated state of a client. If the application state changes faster than the defined interval, it is less resource-intensive than sending each change, because several changes can be consolidated together.

3.1.3 Disadvantages

The main disadvantage of polling is that it is not actually real-time. Updates arrive at a client only after the next poll and not immediately. A short time interval brings faster updates, but most requests could end up being useless because nothing may have changed since the last poll. This is not optimal as it uses resources without bringing any value. Finding the right interval is difficult and often not possible without making any compromises.

3.2 Long Polling

The problem with short polling is that it could result in a lot of unnecessary network traffic. Long polling (3.2) tries to solve this problem by keeping the connection open until an update is available. The workflow is as follows:

1. The client initiates a new HTTP request.
2. The server does not respond immediately, but holds the connection open until there is new information for the client.
3. If new data is available, the server sends it with an HTTP response and ends the request.
4. The client immediately opens a new request and the process starts over.

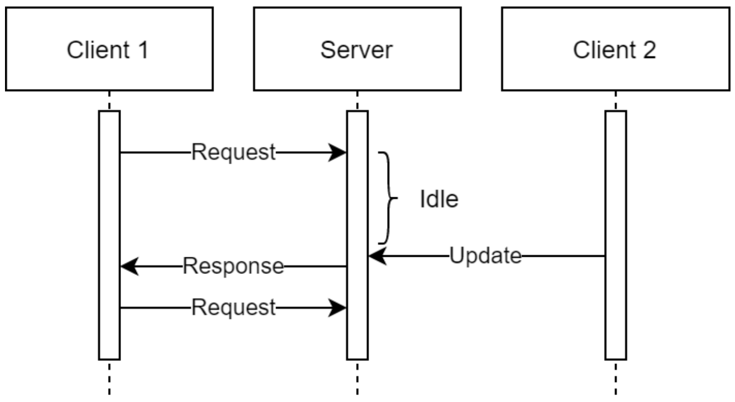

Figure 3: Sequence diagram for long polling

3.2.1 Example

The implementation of long polling on the client side is similar to short polling. The difference is that the request must be reopened immediately after the previous request has finished, instead of invoking the poll based on a time cycle.

```
  async function poll() {
    const response = await fetch('https://puerro.io/api');
    // handle response
    poll();
  }
```

The server can no longer close the connection right away but must use some kind of change detection. This could for example be achieved with the Observer pattern[7]. Upon a request from the client, the server adds a new subscription to an observable and only responds to the request after being notified by the observable.

```
const state = Observable('Puerro'); // see observer pattern

http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/plain');
  // only send response and close connection after a change
  state.once(state => res.end(state.toString()));
}).listen();
```

[7] https://github.com/robin-fhnw/IP5-Puerro/tree/master/src#observable

3.2.2 Advantages

The advantage of long polling is that the server can choose when to respond to the request and can therefore proactively push messages to a client as soon as data becomes available. This eliminates the ping-pong approach from short polling and the unnecessary network traffic that comes with it.

3.2.3 Disadvantages

A downside of long polling is that the implementation on the server part requires some additional considerations. Care must be taken to ensure that a client does not miss any changes in the process of reopening a connection, as seen in figure 4. In addition, an initial state is normally expected on the first request. Instead of using the Observer pattern, these problems can be solved by managing revision information to make the server aware of the current state of a client to provide any missing data.

Timeouts It also needs to be considered that a timeout can occur if no new updates are propagated for a longer period of time. Furthermore, reopening an HTTP request still adds additional network overhead (HTTP header). This is especially a disadvantage if updates are sent at a fast rate.

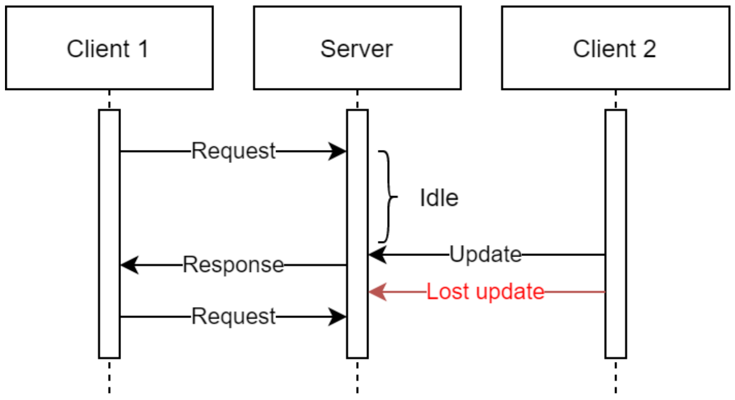

Figure 4: Sequence diagram for lost update in long polling

3.3 Server-Sent Events

The server-sent events (SSE) technology was launched with HTML5. Similar to long polling (3.2), it allows the server to proactively push data to the client. The difference with SSE is that the HTTP connection remains open and messages are transmitted in the form of chunks[8]. This makes it a streaming approach where the client must parse the received data in order to understand the message. Most browsers (93.17%[9] as of 01/2020) provide the specified `EventSource API`[10] to open a connection, receive updates and distribute the messages via DOM events. For the EventSource API to understand the messages, a specific message convention must be used:

- Messages are separated by a double line break (`\n\n`)
- The id of a message is prefixed with "`id:`"
- The type of a message is prefixed with "`event:`"
- The payload of a message is prefixed with "`data:`"

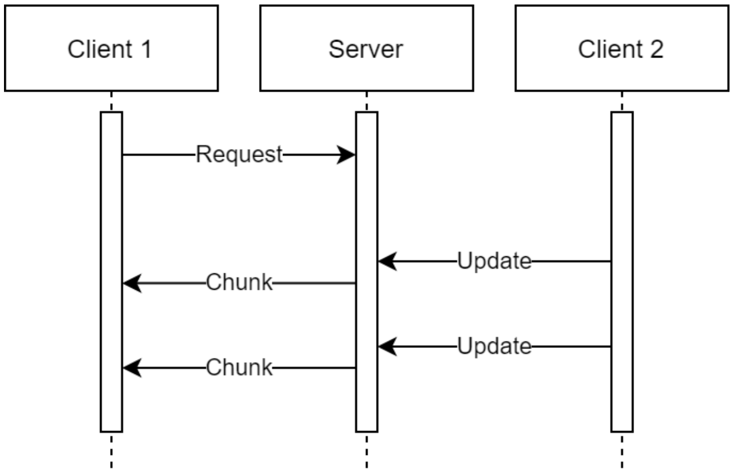

Figure 5: Sequence diagram for Server-Sent Events

[8]https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding
[9]https://caniuse.com/#feat=eventsource
[10]https://www.w3.org/TR/2009/WD-eventsource-20090421/

3.3.1 Example

The implementation on the client is straight forward and handled by the `EventSource` API. Based on the specified event type (with the `"event"` prefix), DOM events are fired on the source object.

```
  const eventSource = new EventSource('https://puerro.io/api');
  eventSource.addEventListener('card_changed', event => {
    // handle 'card_changed' events
  });

  eventSource.addEventListener('message', event => {
    // handle events to which no type has been annotated
  });
```

The server must set the the header `"Content-Type"` to `"text/event-stream"` and write the messages according to the convention to the stream. The `last-event-id` is automatically sent from the EventSource (based on the `"id"` prefix) and can be used to push the appropriate message.

```
http.createServer((req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  const lastEventId = req.headers['last-event-id'];
  // use lastEventId to provide appropriate message

  state.onEvent(event => {
    res.write('id:'    + event.id              + '\n');
    res.write('event:' + event.type            + '\n');
    res.write('data:'  + JSON.stringify(event) + '\n\n');
  });
}).listen();
```

An example of the network stream could look like this:

```
id:    1234
event: card_changed
data:  { text: 'Puerro' }

id:    1235
event: card_changed
data:  { text: 'Huerto' }
```

3.3.2 Advantages

The functionalities are similar to long polling but with the additional benefit that the HTTP request is not being closed and reopened after every update. This means that the network overhead is kept to a minimum and that it is not possible to miss any updates in between. In case the network gets interrupted, the `EventSource API` automatically tries to reconnect and sends the `last-event-id` which has been received on the client. This allows the server to easily identify the missing messages, which need to be re-transmitted. Furthermore, the `EventSource API` fires DOM events for convenience reasons and does not require any additional library or framework as it is built into most browsers.

3.3.3 Disadvantages

The biggest disadvantage of SSE is the browser support. However, this can be bypassed with polyfills[11] if older browsers must be supported. It would also be possible to parse the chunks manually to ensure the support for all required browsers. Furthermore, when using SSE it must be considered that dropped clients only get detected on the server after trying to send a message. The additional functionally of detecting offline clients could be implemented by periodically sending a heartbeat. Similar to long polling, SSE operates on an open HTTP connection which makes it possible for the request to timeout. Because the `EventSource` API automatically reconnects, this should not be a problem for most cases. Furthermore, it must also be considered that most browsers only allow 6 open HTTP connections per domain[12].

[11] https://www.npmjs.com/package/event-source-polyfill

[12] http://blog.olamisan.com/max-parallel-http-connections-in-a-browser

3.4 WebSockets

WebSockets is very different compared to the previous approaches. Instead of utilizing HTTP, it uses its own, dedicated protocol which enables bidirectional streaming of text and binary data over a single TCP connection. [Gri13]. HTTP is used only for the initial handshake to upgrade to the WebSocket protocol [I F11]. The client can connect to a WebSocket server via a browser API, which is supported by most of the modern borwsers[13] (almost 97% as of 01/2020). Similar to SSE, the browser's WebSocket API takes care of opening a connection and parsing the messages.

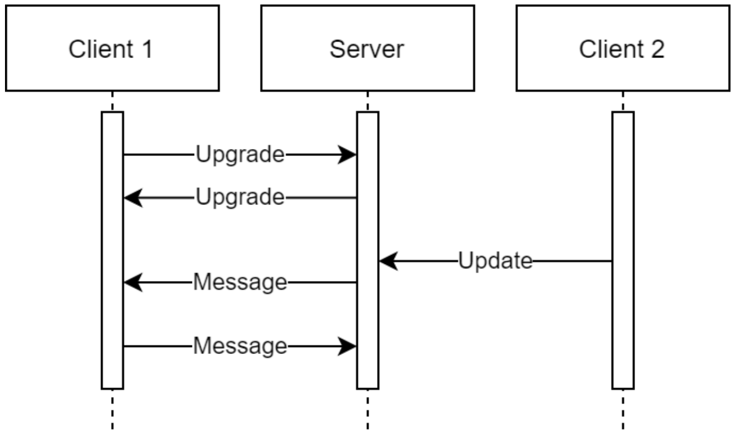

Figure 6: Sequence diagram for WebSockets

[13]https://caniuse.com/#feat=websockets

3.4.1 Example

The browser API on the client is handled similarly to the SSE API. After setting the URL to which the client should connect to, messages are received via DOM events. In addition to receiving updates from the server, messages can also be sent to the server using the same socket.

```
const socket = new WebSocket('ws://puerro.io/api');
socket.addEventListener('message', msg => {
  // handle message
});

socket.send('Lorem ipsum'); // send data over the same connection
```

The implementation of a WebSocket server, however, is completely different and often more complicated than the implementation of an HTTP server. Typically, it requires the help of a framework (e.g. *ws*[14]) which handles the connection management and provides the functionality to send and receive messages via the socket.

```
const server = new WebSocket.Server(); // uses 3rd party library
server.on('connection', ws => {
  ws.on('message', msg => {
    // handle incoming message
  });

  state.on(state => ws.send(state.toString())) // send message
});
```

[14] https://www.npmjs.com/package/ws

3.4.2 Advantages

WebSocket has the advantage that it is bidirectional. Both the sending and the receiving of messages can be handled via the same connection. Unlike the other approaches, WebSockets can automatically detect dropped clients, and the browser allows up to 1024 connections to be open in parallel instead of just 6. In addition, the network overhead is very low, which speeds up the data transfer.

3.4.3 Disadvantages

The biggest downside of the WebSocket protocol is that it cannot use the out of the box features of the browser that are support when using HTTP, such as caching or compression [Gri13]. All these additional functionalities must be handled at the application level, which often requires the use of a dedicated library. This additional burden makes the implementation and the developer experience more complicated. Furthermore, some proxies and firewalls block WebSocket connections, and it is hard to work with load balancers [Gri13].

3.5 Comparison

To compare the different approaches, we have implemented them using practical examples and compared their advantages and disadvantages. Based on the gathered knowledge, we haven given each of them points from 0 to 2 across different categories.

Ease of use This category indicates how easy the implementation is from a developer's perspective. 0 means that it is difficult to use and 2 means that it is easy to implement.

Support The support category compares how good the support across different browsers is. 0 means that support is bad whereas 2 means 100% support.

Network overhead The overhead is about how much unnecessary network traffic is used. 0 means that there is a lot of network overhead and 2 means that the overhead is kept to a minimum.

Functionality Functionality is about the amount of built-in features. 0 means that there are no additional features whereas 2 means that it comes fully equipped.

3.5.1 Result

The following table (1) shows our result including the total.

	Short Polling	Long Polling	SSE	Web Sockets
Ease of use	2	1	1	0
Support	2	2	1	1
Network overhead	0	1	2	2
Functionality	0	0	1	1
Total	**4**	**4**	**5**	**4**

Table 1: Comparison of the approaches

3.5.2 Recommendations

We have concluded that there is not a single best option for all use case. The following paragraphs contain our usage recommendations for pushing messages to a client based on the previous acquired result.

Short Polling Short polling is the easiest to implement and does not require any changes on a typical endpoint. It should be implemented when updates arrive at a steady rate and the requirements are not real-time. In case there are many updates, short polling can provide a simple way to aggregate messages.

Long Polling Long polling is usually a better option than short polling because it eliminates unnecessary server round-trips. It requires some additional consideration on the server, but it allows the server to push messages to the client as new updates arrive in near real-time.

Server-Sent Events In our opinion, server-sent events is usually the best option. It is similar to long polling but with less network overhead and more built-in features. It automatically tries to reconnect and informs the server about its last processed message. In case the `EventSource` API is not supported, long polling can be used as a fallback.

WebSockets If message latency is important (e.g. for real-time games), a WebSocket is probably the best option due to its speed advantages. However, the implementation is more complicated and by bypassing the HTTP protocol, many of the browsers built-in features are no longer supported. We think that WebSockets is an overkill for most use cases and normally not needed.

4 State Distribution

In a real-time environment, every client should display the same information. To ensure a consistent application state across multiple clients, each of them must have the same data. As soon as something changes that is relevant for the shared representation, it has to be communicated over the network. In this chapter, the different approaches to distribute changes are discussed.

4.1 Single Master

One approach is to maintain a single writable copy of the state on a central server. Each change request must be processed on the server before it can be displayed on the clients. The server contains the business logic and therefore has full control over the application and can enforce any integrity constraints.

4.1.1 Strong Consistency

Because the changes are applied on a single server, it can process the updates one after another in a sequential order and keep a conflict free state. Users must wait until previous requests have been handled or retry again at a later time. If there are multiple changes to the same data, the last update determines the final state.

Changing state When a client wants to update the state, it must send a request to the server that handles the change. In a web environment, it is common to use commands to interact with a server. For example, REST is a possible architecture style using URIs and HTTP methods as command endpoints. The request is normally answered by a synchronous response, which includes information about the status and the payload required to update the view.

4.1.2 Updating Clients

After a change, the server needs to inform the other clients about the new application state. To provide real-time functionality, updates should be pushed to the clients automatically without their explicit request (3). These messages are sent asynchronously because the server cannot be blocked until every client has received and confirmed the update.

Cross-channel updates Having a synchronous communication for the issuing clients but asynchronous updates for non-issuing clients can lead to an outdated view because the messages from different communication channels are unaware of each other (7). This can be prevented by coordinating the arrival of messages either by using some kind of revision information or by using a single update channel. A single update channel would mean that the issuer also receives its own updates through the asynchronous distribution channel, as seen in figure 8. The synchronous response could then solely be used to inform the user about occurred failures.

New clients In order to set up a new client, the application can query the state form the central server and start listening for the asynchronous update messages. Thereby, it must be considered that incoming messages between the query response and the attachment on the asynchronous update channel are not being dropped.

4.1.3 Performance

A disadvantage of using a central server is that the latency between the dispatching of an update and its visual representation on the client depends on the network connectivity. Each update must first do a server round-trip before it can be applied locally. Normally, this latency limitation is acceptable if the update does not need to be reflected immediately, for example an effect after pressing a button. It is more problematic, when the unit of edit becomes smaller, such as a single keystroke or the position of a slider. Then the view might update slowly, which is not very user-friendly (9).

4.1.4 Availability

Another consequence is that the client must always be online and connected to the server so that the user can interact with the application.

Optimistic Update Instead of waiting for an acknowledgement from the server, the performance problem could be bypassed by updating the changes locally in an optimistic way. If the update fails unexpectedly, the changes have to be undone on the client, which would result in a complicated conflict and error handling because without a history of operations it is difficult to restore an already aggregated state.

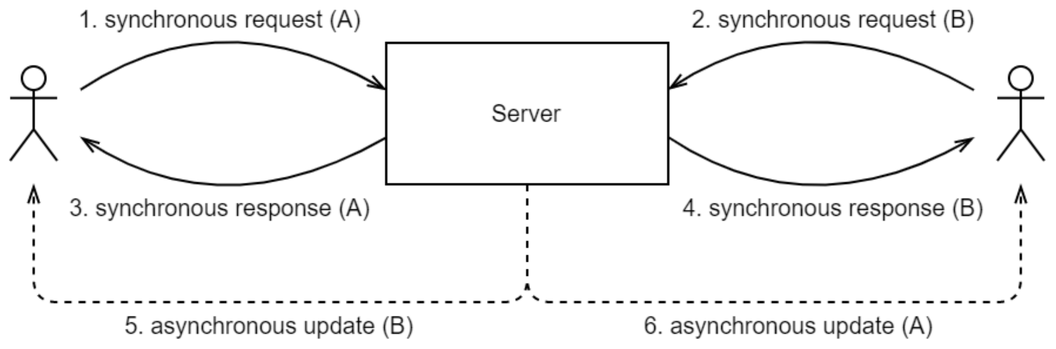
1. synchronous request (A)
2. synchronous request (B)
Server
3. synchronous response (A)
4. synchronous response (B)
5. asynchronous update (B)
6. asynchronous update (A)

Figure 7: Inconsistency with central state processing

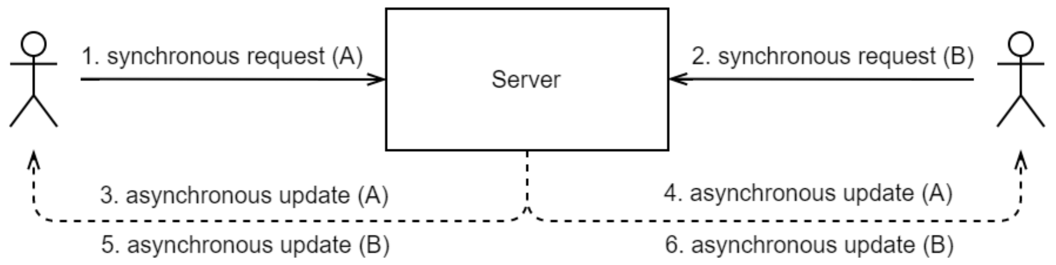
1. synchronous request (A)
Server
2. synchronous request (B)
3. asynchronous update (A)
4. asynchronous update (A)
5. asynchronous update (B)
6. asynchronous update (B)

Figure 8: Consistency with central state processing

4.2 Multiple Masters

Another approach is to allow the clients to manipulate the state locally without the restriction of a server round-trip to validate a change. This means that the source of truth is distributed across the clients and that each client must implement the business logic and comply with the integrity constraints.

4.2.1 Eventual Consistency

Without a central instance that coordinates updates, changes can occur simultaneously on different clients. This inevitably leads to a temporarily divergent system with multiple copies of the application state. To eventually achieve a consistent state again, an asynchronous synchronization process must take place to update the clients. In a live environment, this process should happen as often as possible, ideally after each change.

4.2.2 Synchronization

The synchronisation process must guarantee that eventually every client ends up with the same state. The changes cannot simply be transmitted to the other users, as this could lead to inconsistency and ultimately data loss if changes were made simultaneously. The difficulty is that two diverged states are equally valid but that in the end both need to converge to the same state. How to mange these conflicts is discussed in chapter 5.

New clients New clients must initially receive the current state of the application. Without a server, however, this would mean that at least one additional client would need to be online to synchronize the state peer-to-peer. This cannot be guaranteed in a web environment, therefore it is recommended to also have a server that maintains the state.

4.2.3 Performance

The advantage of allowing local updates is that the updates are instantly applied on the client without the need of a server round-trip that implies network latency.

Offline An additional benefit of working on a local copy of the state is that the clients do not need a permanent network connection, but can be offline.

4.3 CAP Theorem

The CAP theorem states that in a distributed system only two of the following three properties can be guaranteed: Consistency, availability and partition tolerance [Dai13]. Because network interruptions cannot be prevented, there needs to be a decision to either cancel an operation to ensure consistency or proceed with the operation to provide availability [Dai13]. The two above discussed approaches of either having a single master or multiple masters demonstrate this conflict. It is possible to either have a consistent system using a central server, or to apply each change immediately with multiple masters. Both together is unfortunately not possible.

4.4 Conclusion

Having a central server to process the state changes in a serial order is well-known, but its usage in a live environment brings the difficulty of coordinating synchronous and asynchronous messages to ensure consistency. The latency of a server round-trip might be too slow when using small units of edit and *optimistic updates* can lead to a complicated conflict and error handling. Furthermore, it requires the client to be constantly online in order to interact with the application.

On the other hand, having the source of truth distributed across the clients solves the performance and availability issues because the changes do not require a server round-trip to be validated. Having the business logic locally even has the benefit that the application can be used offline. However, the state can be temporally inconsistent and a synchronization process and possibly conflict resolution is needed to eventually achieve a consistent state again.

5 Conflict Management

Multiple equally valid states can emerge by allowing the clients to independently modify the application state locally. Eventually, they must converge back together to guarantee a consistent application state. This chapter discusses how conflicts can be resolved.

5.1 Conflict

A conflict occurs when non commutative changes happen concurrently on somehow related data [Kle17]. In a distributed system with several masters that can modify the application state, different types of conflicts can arise:

- Independent users change the same data concurrently
- A user updates data that was simultaneously removed by another user
- Independent changes that together violate an integrity constraint

5.1.1 Detecting conflicts

To detect a conflict, concurrency must be detected. Operations are concurrent when they are unaware of each other and thus it is impossible to tell which happened first. This can for example be determined with the use of version numbers [Kle17].

Commutative operations Not all concurrent updates on the same data will cause a conflict. Whenever the final outcome does not depend on processing the updates in a certain order, both can and should be applied to avoid losing information. For example, incrementing a counter is a commutative operation.

5.2 Prevention

The best strategy is to avoid conflicts whenever possible. This can be achieved by informing other users about the start and end of an editing process. Chapter 9 discusses how locking and visual hints can help to support preventing conflicts and what it means for the user. These approaches can help to reduce the number of conflicts but it is often not possible to eliminate them completely.

5.3 Convergence

When a conflict could not be prevented, it must be resolved in a way that the system eventually converges. This means that all clients must reach the same final state when all changes have been synchronized [Kle17]. Unfortunately, if the updates are not commutative, it is not possible to simply apply them in the order in which they are received, as this could lead to an inconsistent state.

5.3.1 Merge

The best case scenario would be to somehow merge the changes while trying not to lose any information. This is a challenging and often domain specific task for two equally valid states.

Automatically Merging concurrent operations automatically is a highly studied field. The following algorithms exist but only work on a defined set of data structures:

- Conflict-free replicated data type [15]
- Operational Transformation [16]
- Mergable persistent data structures [17]

Manually Sometimes the method of merging depends on the application and cannot be generalized. This requires each individual application to write custom conflict resolution code, which might be error-prone. Usually this is done by allowing the end user to select the correct state based on the use case.

[15]https://arxiv.org/pdf/1608.03960.pdf
[16]Operational Transformation in Real-Time Group Editors
[17]http://gazagnaire.org/pub/FGM15.pdf

5.3.2 Choose one

Another possibility is to simply choose one of the concurrent operations and delete the others. However, this means that some information might get lost. This can be disadvantageous, especially with regards to usability (9), because the individual changes have already been successfully applied on the clients and are now silently discarded. Different strategies exist for choosing one operation over another. The choice can be arbitrary but must be the same on each client to guarantee consistency. Typically, a strategy called *last wins* or *first wins* is used. But this requires some kind of ordering in simultaneous operations.

5.4 Ordering

When dealing with concurrency, order is an important aspect. A lack of order can lead to a conflict that can only be solved by bringing them back into some kind of order again because it is important to have the same preconditions to solve a conflict deterministically on each independent client.

5.4.1 Total ordering

Having a total order means that any two changes can be compared with each other. This can be achieved, for example, by having a single server through which every change must go. This server forces an order on originally concurrent updates which can then be used to resolve the conflict. However, a central server introduces a single point of failure and possible performance problems.

5.4.2 Consensus

Without a central authority that can guarantee a total order, the clients must find another way to define an order to reach a consensus. This can be any kind of arbitrary but deterministic order.

Unique Id One possibility is to assign a unique id to each change or to each client and order the concurrent changes by this key. The id could be a random Universally Unique Identifier (UUID) or a timestamp (but be aware of problems with synchronizing clocks [GL12]).

6 Event-Driven

In the previous chapters, the focus was mainly on a data-driven approach where the source of truth was in form of a state. This chapter introduces a new way of thinking about state by discussing how an event-driven approach can be used in a web environment.

6.1 Event Sourcing

The idea behind event sourcing is that the source of truth is a log of events instead of a state object. User actions are not directly applied to a state but stored as an immutable event in a history.

6.1.1 Commands and Events

It is important to distinguish a command from an event in order to work with a log-based programming model.

Command A command is an imperative operation with the intend to change something. Therefore, the issuer must know where to place the command and wait for a response to know the result of his inquiry, since the command may also fail. The communication between the issuer and the receiver is generally synchronous and thus coupled. A common example is the REST architecture which uses the request-response pattern to command changes.

Event An event, on the other hand, is simply an immutable fact of something that happened in the past. It cannot fail but only be reacted upon after consuming it. The event producer is decoupled from the consumer allowing a one-way and asynchronous data flow. For example, the browser's DOM utilizes events to inform the JavaScript about what happened on the user interface.

Interaction An event implies that something happened in the first place. This could for example be something non-negotiable like a change in the stock market. However, in a web application it is usually based on a user action which needs to be validated. Such a request normally starts as a command that becomes an immutable event after its acceptance.

6.1.2 Publish-Subscribe

Transferring events from an issuer to a receiver requires an asynchronous messaging pattern as opposed to the synchronous request-response approach. The public-subscribe pattern can be used to broadcast events from one producer to many consumers. Events are categorized into topics allowing consumers to choose which types of messages they want to receive [Kle16]. In a web environment, the consumer is typically a client that runs in a browser.

6.1.3 Event Broker

Although producers could send messages directly to consumers[18], using a centralized event broker to implement the publish-subscribe pattern has the advantage that the system can better tolerate clients which are currently not available. Producers can publish events to the broker, while consumers can receive the events by subscribing to the broker.

Log Storage A central broker is a good place to store events because every message passes through it. Instead of storing a state, the broker only needs to save the messages in a log, which is an append-only operation. Therefore it is possible to simply use an in-memory array or a file in which each line represents an event.

Offset Storing a history of events in the form of a log has the advantage that unavailable or completely new consumers can catch up to the latest state without loosing any messages. This works by maintaining some kind of offset value that can identify the last successfully processed message [Kle16].

[18]http://zguide.zeromq.org/

6.2 Stream Processing

A stream processor is a consumer of events that uses the incoming messages to do something with the received data. Every message is processed as it arrives, instead of grouping multiple events together into batches. Sometimes a stream processor creates new events, but eventually the idea is to generate an output [Kle16].

Stream A stream is just a chronologically ordered sequence of events. Consuming one event after another over time results in streaming the events.

6.2.1 Deriving State

An application must ultimately display the current state and not an event history. Therefore, a common use case for a stream processor is to derive the current state based on the history of events.

6.3 Advantages

The technique of taking a step back and having a new abstraction level in form of raw and immutable events instead of an aggregated state brings some advantages.

6.3.1 Reproducability

Having immutable events means that additional information is available to deterministically reproduce what has happened in the application.

Auditing A log of changes can be beneficial when trying to understand how a specific problem occurred. Errors can easily be audited and retraced.

Projections Multiple representations can be derived from the same log. This can be useful when multiple views are required based on the same data. It can also be helpful to provide new features at a later stage of development.

Recovery Another advantage is that it can be used for undo/redo functionality. This is especially helpful for restoring an application state after an unwanted change.

6.4 Limitations

Storing the source of truth as a log of changes also comes with some limitations.

6.4.1 Disk Space

An append-only log can eventually take up too much disk space, and the larger the history is, the more time it takes to process the messages. Several options exist to truncate and optimize a log.

Snapshots An instance could be responsible for periodically or continuously taking a snapshot in the form of an aggregated state including the last processed offset [Kle17]. New clients could initialize based on this snapshot, and the system could discard all events older than the snapshot. With this approach, it must be ensured that clients who could not catch up with the messages do not miss any changes. (e.g. by keeping the events until all consumers acknowledged the delivery of the message.)

Log Compaction Another option would be to only keep the relevant messages that are used to derive the correct state. This can be a background process that removes all unnecessary events. However, using this technique at an application level would require a complex algorithm.

6.4.2 Querying

Querying a history of events is usually more complicated and time consuming than querying an already aggregated state because it is not in the optimal format for reading.

CQRS This problem can be solved when splitting the write part from the read part as suggested in the CQRS (Command Query Responsibility Segregation) pattern[19]. The idea behind it is that stored data is often not optimized for reading. In an event-driven approach this would mean that instead of querying the log of events, a projection is derived from the history. A stream processor could for example write an aggregated state to a database which can be queried with a familiar query language.

[19]https://cqrs.nu/

7 Conflicting History

Conflicts must be managed differently if an event log is used as the source of truth instead of the current state. As seen in chapter 5, a conflict can occur when changes happen concurrently. In the case of an event history, a conflict arises when the total order of the events is unknown. This can be detected when multiple events point to the same previous event.

7.1 Resolution

An event is immutable and should therefore not be changed or deleted. This means that in order to resolve a conflict, events cannot be merged together and it is not possible to choose one and discard the other. Therefore, the only sensible way to resolve a log conflict is by keeping every event but putting them in a total order (5.4.1).

7.1.1 Merging Streams

Merging multiple streams means that some events need to be inserted in the middle of a log and not just appended. As a consequence, the ordering of the events can change over time.

Example Figure 9 shows an example with the initial situation and the desired end goal after merging the two streams.

1. Client 1 and Client 2 already have the event A in their local history
2. Client 1 adds event B and Client 2 adds event C concurrently
3. The diverged streams must by synchronized and both have to communicate their changes to the other client
4. Client 1 appends event C to its local history
5. Client 2 inserts event B after event A in its local history

In this example, Client 1 could append the remote event and Client 2 had to insert the received event in between. However, the other way around would have been equally valid. The clients must reach a consensus and decide which event should be given priority (5.4.2).

7.1.2 Server Priority

Reaching a consensus is important to make sure that each client ends up with the same event history. This can be achieved by using a central server through which every event passes. This server can force a total order on the history based on when the event is received. This means that the server only appends the messages to its event log and the clients always prioritize the order received from the server over the local order.

Example The example in the figure 10 demonstrates how a consensus is being achieved by forcing a total order on initially concurrent events with the help of a central server.

1. Everyone starts with the event A in their local history
2. Client 1 appends event B, Client 2 appends event D and the server appends event C concurrently to their history
3. The diverged streams must by synchronized and all communicate their changes via the server
4. The server sends event C. Both clients insert the event after event A
5. Client 1 sends event B to the server which appends the event
6. The server sends event B. Both clients insert the event after event C
7. Client 2 sends event D to the server which appends the event
8. The server sends event D. Both clients insert the event after event B

In this example, every received event has an order priority over the local events. It does not matter which client receives an event first. The order is determined solely by which event is received first on the server.

Deduplication Every event is normally received by every client. The issuing client usually already has the event in its history. Therefore, it must be ensured that already inserted events are not inserted twice. The server must send the event to the issuing client as well, because the client may have reloaded the page and lost the event.

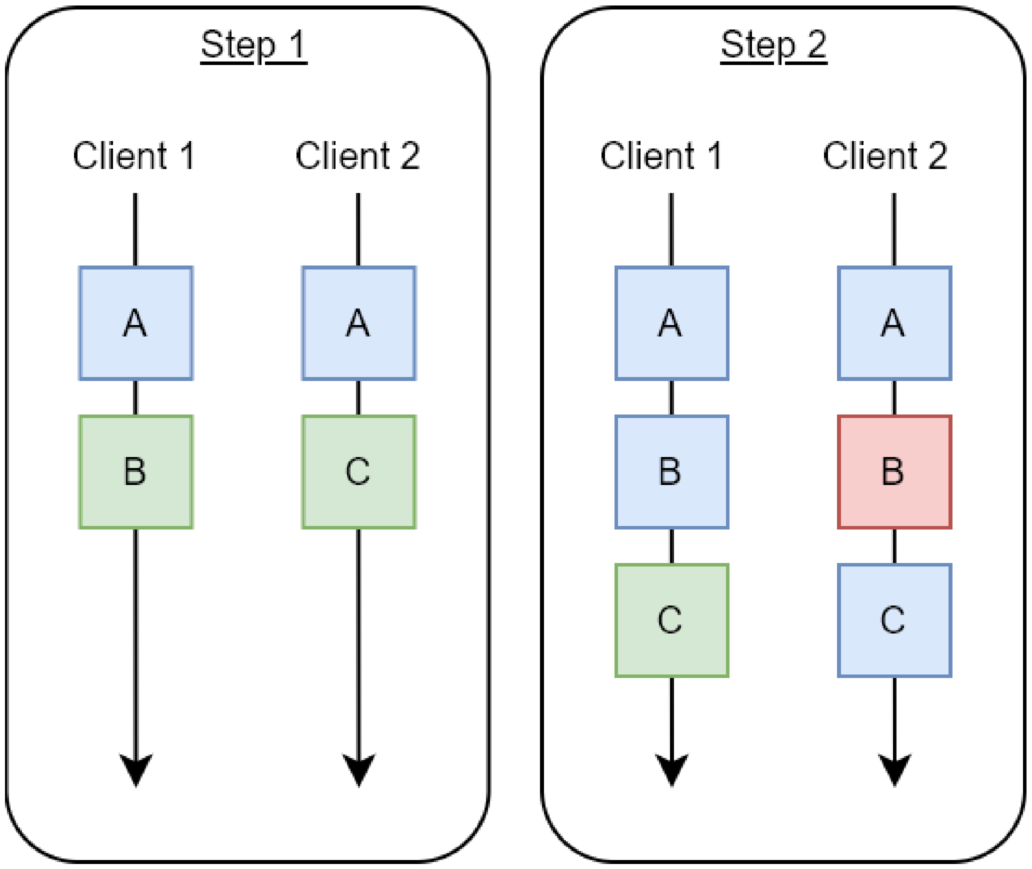

Figure 9: Merging streams

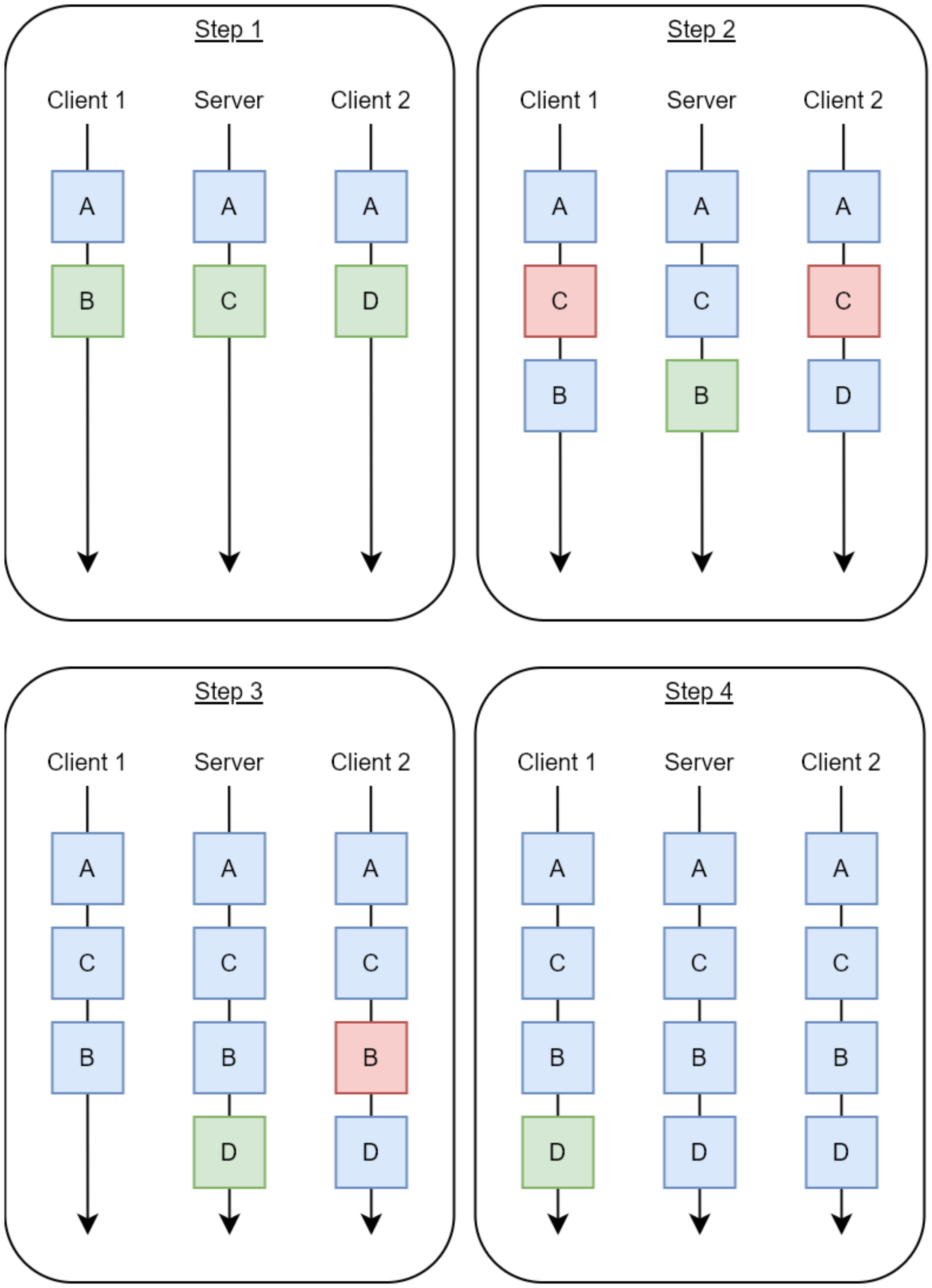

Figure 10: Merging streams with server

7.2 GUI Adjustments

As discussed in a later chapter (8.2), a view is an aggregated projection derived from the event history and every appended event is directly reflected in the projection. Whenever an event is not appended but inserted in between, the Graphical User Interface (GUI) must be restored and reconstructed to reflect the additional fact.

7.2.1 Impossible History

Merging multiple event streams could result in an impossible history. For example, the history could indicate that a field was edited after it has been deleted or that an integrity constraint was violated. The view processor of the application must be able to handle such errors from the history.

7.3 Projection Conflicts

The newly generated projection overwrites the local view, but these two aggregated states could be conflicting. In most cases, the insertion of an event in the middle of the history does not lead to a conflict because it is a small, self-contained and independent unit which can often be merged without loosing any local changes. However, there are some scenarios where the two projections conflict witch each other. As discussed in chapter 5, these are the following:

- Edits on the same field
- Updating data which was simultaneously being removed
- Violating integrity constraints

7.3.1 Resolution

The preconditions for resolving these conflicts deterministically are already given by the total order of events. It depends on the implementation of the stream processor how these conflicts are handled.

Choosing A stream processor can reliably decide which events to apply to the user interface and which to ignore. A custom logic could be implemented to decide on the relevant events.

Merging To merge concurrent events for the derived view, revision information would have to be included in the events. The stream processor could use this to detect concurrency and apply the desired algorithm.

Manually It is often not possible to manually let a user resolve a conflict because the order of the events is given by the server and cannot be changed. The only one way to arrive at a desired state is by dispatching new events.

7.4 Impacts

A stream processor can implement any kind of logic to derive the view. However, the sequential processing and validation of events typically leads to the following effects on the conflicts described in the previous chapter.

7.4.1 Last Wins

Changes on the same field are simply overwritten by the next event and the last event determines the final value of the field.

7.4.2 First Wins

A violation of an integrity constraints is checked in the stream processor for each incoming event. Events are accepted and processed as long as they are valid. Subsequent events that violate the constraints are ignored.

7.4.3 Remove over Edit

If the order implies that something was first edited and then remove, the data will understandably be discarded. However, if the history indicates that something was first deleted and then edited, the stream processor first deletes the data and then tries to apply the edit. As it is not possible to edit an already removed item the edit cannot be applied. This results in always favoring a delete operation over an edit operation.

8 View Rendering

Previously, the focus was mainly on data management and the resolution of potential conflicts. This chapter explores how real-time applications can display a human readable view based on the source of information.

8.1 Data based

Many dynamic web applications operate on a state which is represented by a JavaScript object. Depending on the framework or approach, this state is specific to one component within an application or shared throughout the web page. With frameworks like React[20], components are re-rendered as soon as the underlying state changes. Figure 11 shows a typical state-based render cycle.

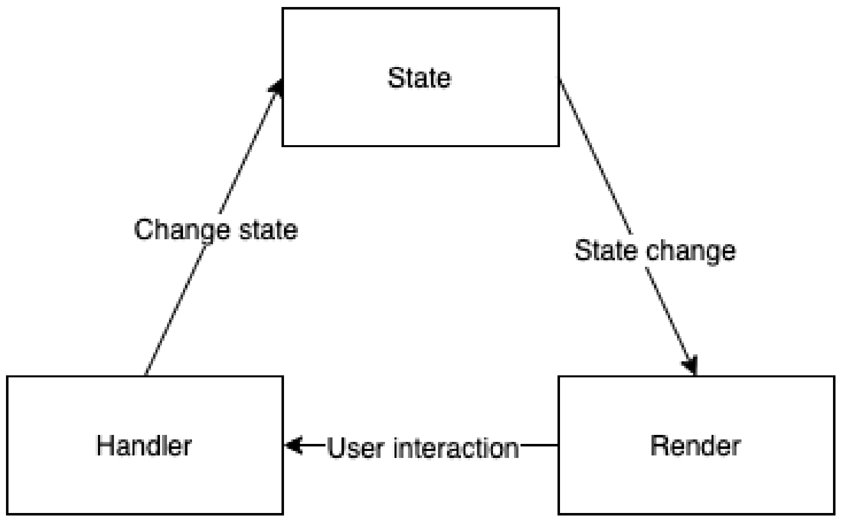

Figure 11: State-based render cycle

[20]https://reactjs.org/

8.1.1 Virtual DOM

In an abstract sense, the UI is always a function of the current state (which is held in a presentation model). With each change, the UI is rendered (based on a render function) to the Document Object Model (DOM). A lot of modern frontend frameworks add an additional step before rendering the state to the DOM. This step is called "virtual Document Object Model (vDOM)". The vDOM can be understood as a virtual copy of the DOM which can be updated in a more effective way than the DOM itself [Ade18].

Only after changing the vDOM the state is rendered to the actual DOM. This can be done by overwriting the entire UI at the root element or by diffing the current DOM to the new vDOM and patching the UI according to the changes. In our IP5 project *Puerro*[21] we dove deeper into how state based rendering works.

8.1.2 Live Updates

In the case of live synchronization, new state is not only generated by user interaction, but also by updates from the server. This data-based approach works best with the single master state distribution used in chapter 4.

8.1.3 Re-Rendering

A possible outcome of an updated presentation model is a refresh of the entire view on each client, which is not ideal in a multi-client application because the temporary state will be overwritten. Temporary state is everything that is not (yet) transmitted to other clients, such as an input text which is only sent after a button is pressed or identity information like the focus on an element. The refresh can also be visually noticeable for example in pictures. As mentioned before, a possible workaround is to use a diffing algorithm that only re-renders the changed elements. Most of the modern front-end frameworks have one built in.

[21] https://github.com/robin-fhnw/IP5-Puerro

8.2 Event based

Rendering based on a stream of events is a different approach to construct a view. As introduced in chapter 6, this can be achieved with a stream processor.

8.2.1 Derived

This works by applying each event after the other as an instruction to create the view. Instead of managing the state as an object and binding it to the actual DOM, each event is directly applied to the DOM upon arrival. For example, if a new element is to be added to a table, the table is not re-rendered, but a new element is simply added by the corresponding handler. If a user changes something in the UI, the changes are transmitted as events as well. In contrast to state based rendering, the view is not a function of the state, but rather the UI is the state.

8.2.2 Initial View

By manipulating the DOM through events, it is difficult to render an initial view when a user opens up the application. This can be handled, by storing the events and sending all in order as soon as a new client connects.

8.2.3 Memory Leaks

Because event listeners are involved at a low level of the application, memory leaks can become a problem. In the example of a table, imagine that each row contains multiple event listeners attached to the central event store to update the cells of the row if something changes. If the row is deleted, those event listeners have to be removed as well. This is made even worse if the initial view is loaded by applying all events: Rows are added and removed and the event listeners are still dangling. One way to combat this is to use central event handlers which patch the DOM based on attributes on the HTML elements. We further discuss this technique in chapter 12.2.

9 Usability

Live synchronization brings a lot of interesting problems from the perspective of usability. In this chapter, the goal is to lay out some of those challenges and discuss possible approaches to manage them. The focus is not on creating a good-looking user interface, but rather examining the impacts of live updates from a practical perspective.

9.1 Coupled View

A main problem occurs because the view output is at the same time the input for changes. As a result, the display and edit modes are often connected with each other. In non-live environments, this has not been a problem because the view represented one point in time and was not dependent on any external changes. The user was able to make changes in isolation and communicate them once completed. In a live environment, however, changes are automatically pushed to the client, which can lead to usability issues.

9.1.1 Asynchronous Message Processing

With the arrival of a new message, the client usually has to adjust the local view based on the information received. In most cases, this causes some parts of the page to be re-rendered (8.1.3) or otherwise updated. This can be unpleasant if the user concurrently tries to update the same data that is changed by the received message. In this case it depends on the implementation on how this situation is dealt with.

Cancel One approach is to cancel the local edit mode when a new message arrives. Thereby, temporary changes which have not yet been communicated to the data source are deleted which can be irritating for the user.

Apply Another possibility is to silently apply the changes, but letting the user continue to edit. This can lead to confusion or an unintentional overwriting, as the user may not notice that the data has already been changed. This could be improved by visualizing the change.

Postpone It could also be an option to detect the local edit mode and wait for the user to finish its changes before applying the newly received data.

9.2 Impact of Conflicts

As discussed in chapter 5, applying state locally without a server validation can lead to unavoidable conflicts. Chapter 5 focuses on resolving conflicts from a technical perspective. This chapter discusses some of its considerations from a usability point of view.

9.2.1 Prevention

Its always a good approach, to prevent conflicts from happening in the first place. From a usability perspective, this can be supported trough the following techniques.

Lock One option is to lock elements that are currently being edited by another user. This can prevent unwanted overwrites, but restricts the possibilities for users to collaborate on the same data.

Display Another option is to visually communicate to the users about the elements that are currently being edited. This might stop users from interfering with the same data, but still provides the option for collaboration.

9.2.2 Deciding on Correctness

Ideally, concurrent changes could always be merged together without loosing any information. However, this is not always possible and sometimes it has to be decided on one change over another.

Manually A typical approach is to allow users to manually resolve the conflict by letting them decide on the correct information. This can be done by displaying the conflict and providing the functionality to decide on the preferred result. It is comparable to the approach that Git[22] uses to merge unsolvable conflicts. Such a functionality is complex to implement and a time-consuming process for a user if conflicts occur frequently. The concept could also be confusing for users that are not tech savvy or familiar with the concept of merging. It must also be considered that messages arrive asynchronously and that it might be too late to ask the user for help because new messages may have already arrived on the client.

[22]https://git-scm.com/

Automatically Another approach is to let the application decide on the final outcome of a conflict as discussed in chapter 5. This leads to some information being lost, which can be fatal and frustrating for users, depending on the use case. With this approach, it may be advisable to inform the users about the discard information instead of silently overruling changes.

9.3 Offline Management

It is possible for a user to go offline temporarily or for a longer period of time. This could be caused, for example, by an interruption in the connection due to a railroad tunnel. The interruption could lead to an unusable application when a server is required to validate each change, as discussed in chapter 4. However, when the data is applied locally without the need for a server round-trip, the application can continue to work offline. This chapter discusses considerations that should be taken if offline editing is possible.

9.3.1 User Alert

Users should be informed about their offline state trough an appropriate warning. Since offline editing is predestined for conflicts, it is also possible to spread awareness that changes could be lost depending on how conflicts are managed.

9.3.2 Disallow Critical Changes

Instead of allowing all changes locally, it can be considered to disallow certain changes. For example, all non commutative changes (e.g. removing elements) could be prevented to protect the users from losing information at a later point. Another possibility is to prevent integrity constraints validations.

9.3.3 Rollbacks

After being offline for a longer period of time, the local information may differ significantly from the other clients. When the user is back online, the information is synchronized and it must be considered that the asynchronous messages (9.1.1) could severely manipulate the local view. This could result in many of the local changes being rolled back, causing some information to be lost. In such a case, it may be a good idea to save the undone changes or apologize to the user for the lost information.

9.3.4 Access Handling

In an offline system, a server is not available to intervene with changes. This can be a problem, for example if changes have to be rejected for security reasons or integrity constraints. Every client can modify its local state directly and changes can only be validated at a later point.

9.4 Too Much Real-Time

Real-time updates are indeed a useful feature when used properly. Without them, pages would need to be reloaded frequently and manually in order to allow collaboration, which does not result in a great user experience. But it is not easy to find the right amount of live updates without overwhelming the users with changes that were not made by themselves. As motion attracts attention, too many changes can overload and distract a user. Therefore, we recommend to only use live updates where the user will benefit from them. For example, displaying when an item is removed from a list. On the other hand, dragging an item and displaying the exact position to other users might be too much, especially if multiple users are using the application.

The amount of live updates is very context specific and depends on the use case of the application. In a collaborative text editor, for example, it is important that even small changes are updated live. While for most business applications, real-time updates only make sense in parts of the user interface.

9.4.1 Throttle and Debounce

Throttling and debouncing are techniques which can be used to prevent too many updates from being sent.

Throttling Throttling allows the user to only send one update in a given time frame (e.g. every 100 milliseconds). If applied to the previous drag example, the position of the element would only change every 100 milliseconds and not immediately.

Debounce On the other hand, debouncing waits until the user does not change anything for a given time. If the element drag was debounced, the update would not be sent until the drag position is static for a specified time.

Part II
Proof of Concept

10 Project

In order to apply our knowledge from part I in a practical use case, we implemented an example project alongside the research process to test the theoretical findings and assumptions in a realistic scenario. We decided to implement a collaborative kanban board. We used this example because it is simple, but covers most of the challenges of real-time synchronization. A kanban board is used to plan and visualize tasks for any project. The board is divided into different columns, each representing a different stage that a work package goes through. This use case really benefits from live updates because users don't have to reload the page when something within the board changes, allowing collaborative planning.

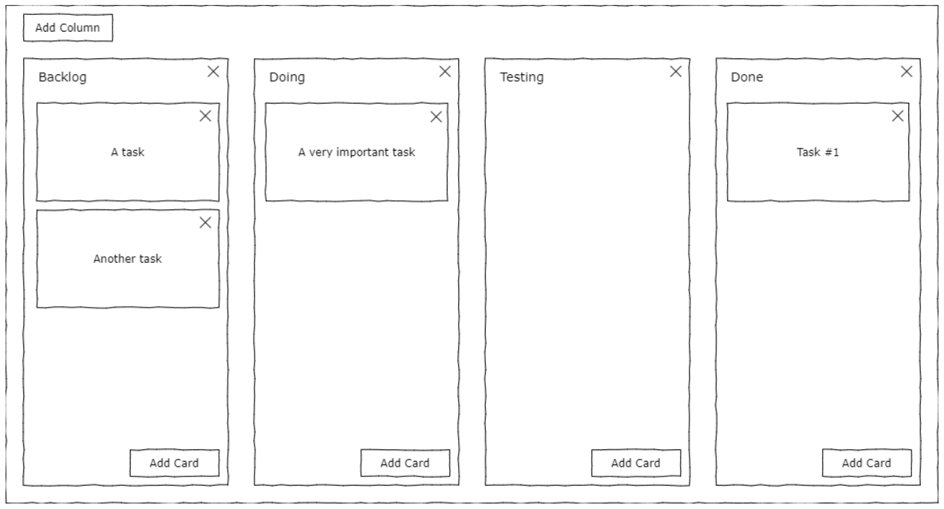

Figure 12: Kanban board mockup

10.1 Requirements

To showcase the techniques and patterns explored we wanted to implement the following common requirements for a kanban board:

- Add and remove a card or a column.
- Change a card or the title of a column.
- Move a card from one column to another.

10.2 Iterations

The kanban board was developed through the following three iterations. The code of the project can be found on the FHNW GitLab[23] instance.

10.2.1 1st Iteration

The first iteration of the board is a Minimum Viable Product (MVP) which was used to experiment with the different approaches to push messages to the clients. The focus of this board is to incorporate the different possibilities and to let the user decide on how the data should be exchanged (short polling, long polling, SSE or WebSockets). This iteration uses a data-based (8.1) approach to communicate and consume changes. The source for this project can be found in the "Laboratory"[24] repository on Gitlab. It formed the decision to continue with server-sent events as the way to push data to the clients.

10.2.2 2nd Iteration

In the second iteration, we focused on the server-sent event technology, but implemented it with an event-driven approach. The view in this iteration is based on streams and components (12.2). The source of the second iteration can be found within the "Proof of Concept" repository[25] on the branch `iteration-2`.

10.2.3 3rd Iteration

The focus of the last iteration was in optimizing the view layer. Especially in its ability to handle conflicts. The rest of part II will focus mainly on this final outcome of the kanban board where we will dive deeper into the architecture and implementation of the application. The final implementation of the project can be found in the same repository as iteration two, but on the `master` branch. The final version of the application is deployed and accessible online[26].

[23] https://gitlab.fhnw.ch/p6-christen-gobeli
[24] https://gitlab.fhnw.ch/p6-christen-gobeli/lab
[25] https://gitlab.fhnw.ch/p6-christen-gobeli/PoC
[26] https://ip6-frontend-pitc-fringebenefit-eg.ocp.puzzle.ch/

11 Architecture

This chapter describes the architecture and the abstractions used to implement the proof of concept. The following sections contain JSDoc[27] extracts to illustrate the corresponding interfaces.

11.1 Event

An `event` is a self-contained object that expresses a user action of something that happened in the past. It is considered immutable and contains a payload as well as additional metadata. An event object is defined by the following properties:

```
/**
 * @typedef {Object} Event
 * @property {String} id        unique id for each event
 * @property {String} type      type/topic of an event
 * @property {String} userId    issuer of the event
 * @property {Object} payload   contained data
 */
```

11.2 Store

The `store` contains a history/log of events by using an in-memory array and allows to append to the log or to insert an event in between. It is important that the insert operation is only used on the client side and never on the server side (7.1.2).

`lastEventId` The order of the events is given by the `lastEventId` attribute which is managed by the store. It belongs to an event but is not immutable because it needs to change when an event is inserted in between the log.

[27] https://jsdoc.app/

11.2.1 Stream

Based on the publish-/subscribe-pattern (6.1.2), the store allows to listen for changes by providing stream subscriptions. The implementation of the stream is inspired by rxjs[28] observables [Osm12]. The idea is to let the consumers filter the stream to decide which events they want to see. This is achieved by allowing a consumer to chain the filtering methods for a given stream. The API for a stream looks as follows:

```
/**
 * @typedef {Object} Stream
 * @property {function(function): Stream} map
 * @property {function(function): Stream} filter
 * @property {function(function): Stream} skipUntilAfter
 * @property {function(function): function} subscribe
 */
```

11.3 Scheduler

The `scheduler` was inspired by the FHNW module *Web Programming* [29] and allows to order actions by chaining them into a queue. This ensures that an action is only started when the previous action has been completed. This is especially useful if different asynchronous actions would be running at the same time.

11.4 API

The application contains a central `API` module, which communicates with the server. It sends and receives events over the HTTP protocol. It utilizes server-sent events (3.3) to push messages to a client.

[28]https://github.com/ReactiveX/rxjs
[29]https://www.fhnw.ch/de/studium/module/9248673

11.5 DOM Projection

The DOM projection abstracts the DOM handling. In addition to direct DOM manipulation, this abstraction records each operation and makes it possible to reverse the changes at a later time. Having a revertible DOM is important to resolve history conflicts (7).

11.5.1 Operation

An operation contains the code to execute a DOM manipulation and the corresponding inverse to undo the modification. The following operations were implemented for the kanban board application (this is just a baseline, which can be extended as needed):

```
/**
* @typedef {Object} DomProjection
* @property {DomAppendOperation}          append
* @property {DomPrependOperation}         prepend
* @property {DomInsertBeforeOperation}    insertBefore
* @property {DomRemoveChildOperation}     removeChild
* @property {DomRemoveOperation}          remove
* @property {DomSetAttributeOperation}    setAttribute
* @property {DomRemoveAttributeOperation} removeAttribute
* @property {DomSetTextContentOperation}  setTextContent
* @property {DomSetValueOperation}        setValue
*/
```

11.5.2 Checkpoint

In between the operations, checkpoints are added after each event to know which operations the reverse function must call to revert an event. This allows handlers to have multiple manipulations per event and provides the functionality to travel backwards on an event level.

11.6 Controller

The `controller` is the central point of the kanban web application. It is where the `store`, the `DOM projection` and the `API` connection is managed. Each event has to go through this central instance. Events published by the client pass through the controller to be distributed within the application and then sent via the API. Furthermore, the controller decides what to do with an incoming event. It checks whether the event already exists or whether it must be inserted somewhere in the history. To prevent multiple instances of the controller, it is implemented using the singleton pattern [Osm12].

11.7 View

In this project, two different approaches were implemented to represent the view (12.2). The following sections contain the concepts for deriving a view from an event log. These concepts will be used later in the implementation.

11.7.1 Event Handler

An event handler manipulates the DOM based on the information received. The corresponding handlers are executed when a event is published in the application. They can use the projector functions for display purposes and provide the desired parameters.

11.7.2 UI Projector

A UI projector is a function that creates a `HTMLElement` based on a set of input parameters. It can attach itself to DOM events and dispatch events based on user actions. It returns the projected DOM element in order to be attached to the view. However, it does not handle incoming events directly and is therefore mainly responsible for the look and feel.

11.7.3 Component

A component is simply the combination of the look and feel of a projection with the actions of an event handler. In this project we used this idea of a component in 2 different ways, which are explained further in chapter (12.2).

11.8 Overview

The following diagram (13) shows the architectural design of the earlier discussed modules and how they are connected with each other. The specific implementation details are described in the next chapter (12).

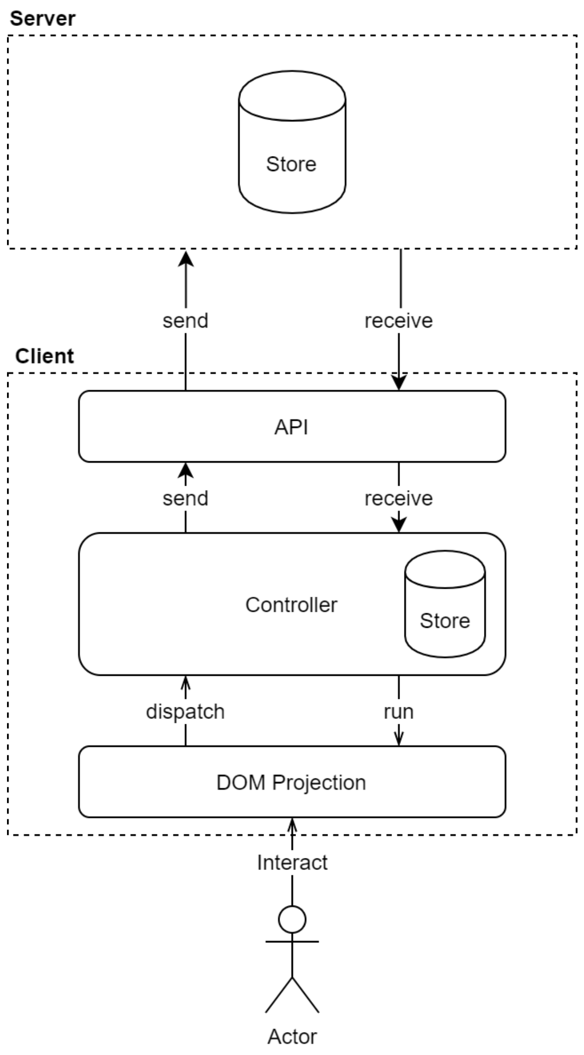

Figure 13: Structure of the architecture

12 Implementation

This chapter describes the key design decisions taken for the example project.

12.1 End-to-End Events

Events are the type of information that is communicated over the network. They are emitted on the clients and eventually sent to the server. A current state is not being aggregated on the server, but the events are forwarded directly to the clients, where they are processed to construct the view. Every event is distributed to every client, including the issuer.

12.1.1 Offline First

To enable a fast processing with a small unit of edit, despite network complications, we decided to implement an *offline first* approach. This allows a client to directly apply the operations (4.2) instead of waiting for a validation from the server. This means that events are processed locally before they are being sent to the server. Using event sourcing (6.1), every client has a local copy of the event log (11.2).

12.1.2 Sending Events

Events are sent trough a single POST endpoint provided by the server. The server maintains the events with an append-only operation and broadcasts them to all clients.

Queue The `scheduler` (11.3) is used to send the events one after another. If the server cannot keep up with the incoming messages (e.g. due to network complications), they are not lost, but are queued up on the client. Sending will only continue when the server can receive events again.

12.1.3 Receiving Events

To allow for two-way communication, we decided to use server-sent events (3.3) to push messages to the client. The sending and the receiving channel are independent simplex channels but together they are one full duplex channel.

12.2 View Rendering

The view is derived from the local history of events and forms the current state of the application in the DOM. No separate state object is being maintained alongside the view. Two different approaches for view rendering have been implemented.

12.2.1 Streaming Components

For the first approach (iteration 2), the view uses components as stream processors. Thereby, each component directly uses the stream of events provided by the store in the desired way. Every component has access to the log and can independently manipulate the view based on the events.

Unsubscribe A problem with this approach is that a stream must be unsubscribed when the component is removed to prevent memory leaks. For example, a card component must remove its own subscriptions if it is deleted. This was being generalized in a `Component` function amongst other recurring functionalities. The details can be found on GitLab[30].

Irreversible Since the component directly manipulates the DOM on incoming events, it is impossible to go back in time to resolve a conflict. The entire view has to be rerendered from the beginning of the event log. This leads to a bad performance as the event history grows.

12.2.2 Event Handlers

For the second approach (iteration 3), the logic of the event handling is separated from the representation. Each event handler is registered at application level and called when a corresponding event is emitted via the controller. A handler manipulates the DOM based on the event payload and can use a projector function for displaying purposes.

IDs Because the handlers work at application level, they must identify the relevant elements in the DOM. This is achieved by utilizing IDs to find the correct references.

[30] https://gitlab.fhnw.ch/p6-christen-gobeli/PoC

12.3 Conflict Management

As soon as a user performs any change, the local event history is different from the other clients. These divergent copies of event streams are synchronized as soon as possible to eventually guarantee a consistent application state (7).

12.3.1 Prevention

An attempt is made to avoid conflicts by notifying the users about simultaneous edits on a specific element. This should prevent that several users unknowingly change the same data. Two approaches are being utilized.

Lock If a card is dragged, a lock is used to prevent other users from updating the card while it is being dragged.

Display To inform users that someone else is editing a text, the `input` component displays all users who are currently editing the text.

12.3.2 Replaying

When a conflict occurs in the history of events, it is automatically resolved by restoring the DOM to the pre-conflicting event and then replying the operations in the correct order. This is achieved by using the `DOM projection` abstraction (11.5).

12.3.3 Error Handling

As discussed in chapter 9, it is important to inform the user about any possible problems or inconsistencies. Therefore, the application informs the user about the connection status and also when the view is being replayed due to a conflict in the history of events.

Log For demonstration purposes, the event log is displayed in the view. This also helps to understand the changes made in the application.

12.4 Server

The final implementation of the server is a simple Node.js[31] HTTP server. Its purpose is to receive events and to broadcast them to all connected clients. It uses the `last-event-id` of the SSE (3.3) specification to only send the messages that were not yet being processed by a client. Incoming events are also persisted in a file to ensure that no events are lost after restarting the server.

12.4.1 Id Management

Because resources are not created on the server, id management cannot be handled centrally. With this restriction we had to find a way to generate Ids on the client. For this sample project, we implemented an id-generation function that uses the browser's `Math.random` API to generate a 10 character alphanumeric string. For our demo purposes this id is random enough. However, we don't recommend using this function in a productive environment, as it uses a pseudo-random number generator, which is not truly random.[32] A better solution would be to use a UUID[33].

[31] https://nodejs.org/
[32] https://hackernoon.com/how-does-js-math-random-generate-random-numbers
[33] https://de.wikipedia.org/wiki/Universally_Unique_Identifier

13 Conclusion

In our opinion, real-time capabilities for web applications are not just a trend, but increasingly a requirement for various use cases. In this project we were able to demonstrate different approaches to bring a live experience into the browser without having to constantly refresh the page manually.

After studying the methods of pushing data to a client without an explicit request, we concluded that there is not one single best option for all use cases and that the choice is project specific. In general, our recommendation is to use the server-sent events technology because it is simple, works with HTTP and, unlike polling techniques, provides additional features.

After that we focused more on the architecture side of a web application with live capabilities in mind. We have noticed that the browser is traditionally only intended to view a state at one point in time and updating the state is usually achieved by requesting a modification on a server. This has the disadvantage that each change must do a server round-trip for validation purposes, which can result in a poor user experience if many updates are sent over a short period of time.

To overcome these challenges, we decided to concentrate on an *offline first* approach, where changes are applied locally and then distributed asynchronously. This enables a fast execution and an offline capable user interface, but brings with it the additional challenge of conflict management. To resolve the conflicts, we have used an event-driven approach where each event is a self-contained and immutable user action. With the idea of event sourcing, the server manages a pure append-only log of the events and can therefore guarantee a total order that allows conflicts to be resolved deterministically and therefore in a consistent way for each client.

In conclusion, we were able to create a fast, offline capable and automatically synchronized web application that uses events as the source of truth instead of an aggregated state. Along the way, we have encountered that a real-time environment can sometimes lead to unexpected behaviours in the user interface. It is important to be aware of the usability trade-offs that real-time synchronization brings. In general, we recommend not to overdo the live features, because in most cases less is usually more.

13.1 Outlook

There still are some unsolved issues and challenges which could be taken on in a further research.

13.1.1 Initial State

The initial state of our kanban board is loaded by sending the entire history form the server to the client. This approach leads to several pain points. One problem is, that the history of events increases with time, which also increases the initial loading time. In addition, the UI currently cannot differentiate between events that are sent as part of the history and events that are live. This results in not knowing exactly when the UI loading process is finished, which in turn leads to a mistimed dispatch of the login command. This creates conflicts for each new user, which slows down the loading process.

13.1.2 Snapshots

A solution to this complication could be to send an initial snapshot of the state before moving to event based updates, as discussed in chapter 6.4.1. This would require a server-side entity to record the events onto a state object. Additionally, the UI would need to be able to visualize state in form of an object, which it is not currently set up to do.

13.1.3 Undo/Redo

With an event based UI, which is revertible, implementing undo/redo capabilities would be fairly simple taken at face value. A big challenge though, would be to differentiate between changes made by the user which uses the undo feature and other users. If not differentiated, users could undo and redo changes which are not made by them.

13.1.4 User Authorisation

Currently, there is no way to restrict users from executing certain operations or mutations. For example, it is not possible to only allow a specific group of users to add and remove columns in our kanban board. This is further complicated by the offline capabilities of the sample application.

To resolve this shortcoming, there would need to be an authentication/authorisation authority on the server side. This authority should inform the client side application of what the user is allowed to do. The information about the user's rights could be signed to ensure authenticity, for instance with `JWT`[34]. The `JWT` token could then be stored on the client side to ensure that the user cannot do illegal actions while being offline. Of course, the token would have to be validated on the server with each new event being sent.

13.1.5 Persistence

In our current approach, the entire history is stored on the server. Because the history is an append-only operation its size increases over time. This can eventually lead to a problem because the disk space is limited. This challenge is also discussed in chapter 6.4.1. Furthermore, the event history is persisted in a plain text file. To enable better performance, querying and other capabilities, a database system could be set up that is optimized for event sourcing.

[34]https://jwt.io/

13.2 Reflection

The following sections contain our personal reflections about the project.

13.2.1 Robin Christen

For me, the project was very fascinating and I enjoyed the combination of the research process and the implementation of the ideas in a practical example. It was a challenging task because it involved a lot of uncertainty without having a clear idea of the final goal or a predefined method. Nevertheless, this was also the most interesting and rewarding part. It involved a lot of trial and error, but I think the resulting learning aspect was enormous.

In addition, I particularly enjoyed the regular exchange with my team mate and our superiors. It was a good experience to be able to talk about difficulties and to receive suggestions from others, which helped us to overcome many challenges together.

I am very happy with the outcome of this project and I am convinced that this gathered knowledge can be used for future projects.

13.2.2 Etienne Gobeli

Since a lot of concepts explored were new, this thesis was quite the adventure for me. The adventurous aspect was also the most exciting part of the project. Each week we stumbled upon different problems and challenges, which required solutions. Although challenging, the project was very interesting and educational.

The thesis was conducted in an iterative approach and always in exchange with the supervisors of the project. The path taken was very exciting and constructive. Most of the coding for the proof of concept was done in a pair-programming approach, which proofed to be very helpful and enriching.

Personally, I am pleased with what we managed to accomplish. I learned a lot and feel like we shed light on upcoming challenges which will arise with the advent of real-time requirements for web applications.

Part III
Appendix

References

[I F11] A. Melnikov I. Fette. *The WebSocket Protocol*. Tech. rep. 2011.

[GL12] Seth Gilbert and Nancy Lynch. “Clocks Are Bad, Or, Welcome to the Wonderful World of Distributed Systems”. In: *riak blog* (2012).

[Osm12] Addy Osmani. *Learning JavaScript Design Patterns*. O’Reilly Media, 2012. Chap. 10. ISBN: 9781449331818.

[Dai13] John Daily. “Perspectives on the CAP Theorem”. In: *IEEE Computer Magazine* (2013).

[Gri13] Ilya Grigorik. *High Performance Browser Networking*. O’Reilly Media, 2013. Chap. 10, pp. 920–926. ISBN: 9781449344764.

[Kle16] Martin Kleppmann. *Making Sense of Stream Processing*. O’Reilly Media, 2016. ISBN: 9781491937280.

[Kle17] Martin Kleppmann. *Designing Data-Intensive Applications*. O’Reilly Media, 2017. ISBN: 9781449373320.

[Ade18] Ire Aderinokun. *Bitsofcode*. Dec. 2018. URL: https://bitsofcode.de/understanding-the-virtual-dom/.

[Gwe18] Martin Gwerder. *Lecture ”Communication in distributed Systems”*. 2018.

List of Figures

1 HTTP's request-response pattern 10
2 Sequence diagram for short polling 12
3 Sequence diagram for long polling 14
4 Sequence diagram for lost update in long polling 16
5 Sequence diagram for Server-Sent Events 17
6 Sequence diagram for WebSockets 20
7 Inconsistency with central state processing 27
8 Consistency with central state processing 27
9 Merging streams 39
10 Merging streams with server 40
11 State-based render cycle 43
12 Kanban board mockup 51
13 Structure of the architecture 57

Glossary

DOM Document Object Model. 44, 45, 55, 56, 59

GUI Graphical User Interface. 41

HTML Hypertext Markup Language. 9, 45

HTTP Hypertext Transfer Protocol. 10–14, 17, 19–21, 24, 25, 54, 61

JSON JavaScript Object Notation. 9

MVP Minimum Viable Product. 52

TCP/IP Transmission Control Protocol/Internet Protocol. 10

UI User Interface. 9, 44, 45, 56, 63

UUID Universally Unique Identifier. 32

vDOM virtual Document Object Model. 44

Declaration of Honesty

We hereby declare that we have written the presented thesis independently, without the help of third parties and only by using the sources stated.

Location, Date: Windisch, 20.03.2020

Names, Signatures

Etienne Gobeli Robin Christen

Fachhochschule Nordwestschweiz

Design UX | App Web

IMVS18: WebUI Next Generation ++

			Priorität 1	Priorität 2
Betreuer:	Dierk König Dieter Holz	**Arbeitsumfang:**	P6 (360h pro Student)	---
		Teamgrösse:	2er Team	---
Sprachen:	Deutsch oder Englisch			

Ausgangslage
Das Vorgänger P5 Projekt untersuchte verschiedene Ansätze für moderne Web-UI-Konstruktion. Das P6 Projekt soll auf diesen Konzepten aufbauen und sie um sichere, asynchrone, reihenfolgetreue Kommunikationskonzepte ergänzen.

Ziel der Arbeit
Entwicklung und Umsetzung eines belastbaren Konzepts für die sichere, asynchrone, reihenfolgetreue Kommunikation von anspruchsvollen Web UI mit einem Presentation-Server.
Die Kommunikation erlaubt dann

- Daten auf dem Server zu verwalten
- Präsentationsinformation über Geräte und Benutzer hinweg zu teilen
- Ergebnisse wieder nahtlos und zeitnah in das UI einzupflegen.

Problemstellung
Viele ad-hoc Lösungen für die client-server Kommunikation von Web UIs mit einem Presentation Server sind nicht zuverlässig genug. Es gilt, die speziellen Anforderungen an Bandbreite und Latenz strukturell und algorithmisch erfüllbar zu machen.

Technologien/Fachliche Schwerpunkte/Referenzen
Java, JavaScript

Bemerkung
Dieses Projekt ist für Etienne Gobeli und Robin Christen reserviert.