A Web Component for Real-Time Collaborative Text Editing

Master of Science Thesis
University of Turku
Department of Future Technologies
Software Engineering
2019
Pekka Maanpää

UNIVERSITY OF TURKU
Department of Future Technologies

PEKKA MAANPÄÄ: A Web Component for Real-Time Collaborative Text Editing

Master of Science Thesis, 76 p., 18 app. p.
Software Engineering
April 2019

Real-time collaborative software allows physically distinct people to co-operate by working on a shared application state, receiving updates from each other in real-time. The goal of this thesis was to create a developer tool, which would allow web application developers to easily integrate a collaborative text editor into their applications. In order to remain technology agnostic and to utilize the latest web standards, this product was implemented as a web component, a reusable user interface component built with native web browser features.

The main challenge in developing a real-time collaboration tool is the handling of concurrent updates, which might conflict with one another. To tackle this issue, many consistency maintenance algorithms have been presented in the academic literature. Most of these techniques are variations of two main approaches: operational transformation and commutative replicated data types. In this thesis, we reviewed some of these methods and chose the GOTO operational transformation algorithm to be implemented in our component.

Besides selecting and implementing an appropriate consistency maintenance technique, the contributions of this thesis include the design of an easy-to-use application programming interface (API). Our solution also fulfills some practical requirements of group editors not covered by the consistency maintenance theory, such as session management and cleaning of the message queue. The created web component succeeds in encapsulating the complexity related to concurrency control and handling of joining peers in the client-side implementation, which allows the application logic to remain simplistic. This open-source product enables software developers to add a collaborative text editor to their web applications by broadcasting the updates provided by an event-based API to participating peers.

Keywords: collaborative software, web components, consistency maintenance, operational transformation

Contents

List of Listings

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

CRDT Commutative Replicated Data Type.

CSS Cascading Style Sheets.

DOM Document Object Model.

DX Developer Experience.

ES6 ECMAScript 6.

ET Exclusion Transformation.

HTML Hypertext Markup Language.

HTTP Hypertext Transfer Protocol.

IT Inclusion Transformation.

OT Operational Transformation.

P2P Peer-to-Peer.

TCP Transmission Control Protocol.

UI User Interface.

URL Uniform Resource Locator.

UUID Universally Unique Identifier.

UX User Experience.

W3C World Wide Web Consortium.

WHATWG Web Hypertext Application Technology Working Group.

1 Introduction

The Internet has enabled real-time collaboration among physically distinct people. The means of collaboration are not restricted to communication via voice or text messages, as modern software tools allow users to edit shared data in real time. Such software is also referred to as *groupware*. A common application domain for collaborative software is word processing, one popular example being Google Docs. Its usage has been shown to increase co-operation [1], enhance learning [2] and improve the perceived quality of the created document [3].

The modern web has evolved from a simple file sharing system into a powerful platform for building distributed applications. Users may often find web applications more attractive than traditional software, due to the fact that they do not need to be installed. Simply clicking a link or typing the URL into the browser will give the user an almost instant access to the software. A well designed web application also works seamlessly on any system, so the user can e.g. use it on a desktop computer at home and continue on the go with her mobile device. From the developer's point of view, web browsers provide a single interface for targeting multiple platforms, which saves the trouble of writing separate versions of the same software for different operating systems.

The growing complexity of the web has given more business incentives for developers to build tools for other developers. Frameworks provide fully featured workflows that speed up the development, while reusable UI components solve more specific problems. The latest web standards enable developing framework-agnostic reusable UI components,

called *web components*. The web component standards and the means for building web components are examined in Chapter 2. By utilizing various developer tools, application developers can spend more time on implementing their business logic and deliver working software faster.

Collaborative text editing is an interesting subject for a developer tool. Such a tool allows developers to enrich their applications with collaborative editing without implementing their own solutions. We could envision this tool as a UI component. It is a text editor, after all. The problem is that this component needs to exchange data with other component instances, which reside at separated client applications. Networking is not a concern of a generic UI component, but an application-level issue. How to handle the routing of the messages depends on the selected technology stack and in most cases involves also server-side functionality.

There exists several collaborative text editor products for web developers. All of them need a server-side counter-part for the client-side UI component. CKEditor [4] is a rich text editor for web, which introduced collaborative features in its version 5. Its server-side program can be either utilized through the provider's cloud service, or installed on the developer's own premises. Firepad [5] is an editor which enables its users to collaborate on either rich text or code. It depends on Firebase databases to synchronize the data. Quill [6] is another rich-text editor for web. There is a project which adds collaborative features to this editor and requires ShareDB groupware to run in the backend [7]. We are not aware of a web-based developer tool for collaborative text editing that would not force their users to either rely on external services or to run a specific technology on their own servers.

The main algorithmic challenge in the development of a collaborative editor is consistency maintenance. When distributed users make changes to a shared application state, it is possible that concurrent updates conflict with one another. The editor should ensure that each user will have the same text in their editors after executing all of the local and

remote editing operations. Also, the effect of each edit should preserve the intention of its original author at each client. Over the last few decades, academics have developed various algorithms for consistency maintenance, which we will review in Chapter 3.

The goal of this thesis is to design and implement a client-side collaborative text editor component for the web platform, which will not depend on any specific technology running in the backend and can be integrated into any web technology stack with minimal effort. To fulfill this goal, we hope to succeed in implementing the consistency maintenance logic purely in the client-side. Since the message routing is an application-level concern, we can not encapsulate this communication logic in the component. Instead, the component's API should provide a simple interface for sending and receiving updates. We want the component to work with native browser features, which is why it will be implemented as a web component.

Following the background chapters on web components and consistency maintenance, Chapter 4 describes the design and implementation of our collaborative web component. In Chapter 5, we evaluate the created product by integrating it into two test applications. One of them uses a traditional centralized architecture, while the other connects the clients directly to each other in a P2P network. The component design is then improved based on the pitfalls found in the developer experience. The next steps for this work are described in Chapter 6, and Chapter 7 concludes the thesis.

2 Web Components

2.1 Background

Frontend web development typically consists of writing the structure of the user interface in HTML markup, defining styles in CSS and programming functionality with JavaScript. Web browsers interpret this information and render the page content accordingly. Organizations such as the World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG) develop standards for the web technologies, which browser vendors follow to maintain consistency in the web. This ensures that when a web developer adds e.g. an `<input>` element to her application, it will behave pretty much the same way when opened in Google Chrome as in Mozilla Firefox.

A common pattern in UI development is to create reusable custom components as building blocks for the application, and to use such components provided by other developers. HTML elements are UI components for the web, and by combining these elements and adding functionality with JavaScript, web developers can build larger and more specific building blocks for their applications. However, before web components, developers could not encapsulate the component interiors and easily reuse one component, without copying its content and bloating the document with the implementation details of every component instance. Several frameworks fixed this issue with their own component models, but a standardized way for building components that work across frameworks, or without any framework, was missing.

2.2 Standards

A web component encapsulates the UI component with its structure and functionality into a custom HTML element. Rather than referring to a single feature in the web platform, the term web components is actually used to describe a development pattern which is enabled by three web standards. First, the *custom elements* standard (Section 2.2.1) allows creating your own HTML elements. Next, *shadow DOM* (Section 2.2.2) enables encapsulating the component's content from the rest of the document. Last, *HTML templates* (Section 2.2.3) allow defining the HTML content of the element once, but reusing it for all of the component instances. [8] These features reflect the state of web component development at the time of writing, while the web platform keeps on rapidly evolving.

2.2.1 Custom Elements

The custom elements standard is the foundation for making web components. This feature set allows extending the JavaScript class `HTMLElement` with custom functionality and defining the tag name for this new element with the `customElements.define` function. When the tag name is registered to the specified class, the browsers know what to render when they encounter this tag while parsing an HTML document. The custom tag name must contain a dash in order to avoid naming conflicts with the native HTML elements. [9] A minimal JavaScript example is shown in Listing 2.1. After executing this code, the element can be used like any other HTML element by inserting tags `<my-element></my-element>` into the document.

```
1 class MyElement extends HTMLElement {
2 }
3 customElements.define('my-element', MyElement);
```

Listing 2.1: Registering a custom element

Custom elements have lifecycle callback functions which can optionally be implemented for handling some events in their lifespan. Following from the element's ES6 class definition, a function named as `constructor` will be called automatically when the object is created. A function named as `connectedCallback` will be called each time the element is attached to the DOM, and `disconnectedCallback` is called each time it is detached. The `attributeChangedCallback` can be used for executing custom logic when the value of an attribute has changed. It provides the name of the changed attribute, its old value and the new value as the arguments. The attribute's name must be listed in the element's `observedAttributes` array in order to trigger this function call. [9]

It is often a good practice to synchronize the HTML attributes with the properties of the corresponding DOM node [10]. This can be achieved by overriding the property's setter and getter to set and retrieve the value of the attribute [11]. Listing 2.2 shows an example of a custom element which updates its content when the attribute or the property called *name* changes. The content is rendered by setting the `innerHTML` property of the element to contain a heading with the desired message.[1] Displaying a "Hello World!" message with this element can be achieved either by setting the HTML attribute:

```
<hello-heading name="World"></hello-heading>
```

or by setting the property in JavaScript:

```
document.querySelector('hello-heading').name = 'World';
```

As we can see from the example, custom elements alone provide a way to create reusable custom UI components for the web. The internal logic is hidden in the class implementation and the user is provided with an API to interact with it. In our example, the API consists of the single attribute `name`, which can be changed to configure the

[1] Note that there are other ways to populate content into an HTML element with JavaScript. We could create the heading with `document.createElement` and append it into our custom element with the `appendChild` function [12].

```
class HelloHeading extends HTMLElement {

  static get observedAttributes() {
    return ['name'];
  }

  get name() {
    return this.getAttribute('name');
  }

  set name(value) {
    if (value) {
      this.setAttribute('name', value);
    } else {
      this.removeAttribute('name');
    }
  }

  connectedCallback() {
    this._updateContent();
  }

  attributeChangedCallback(attrName, oldValue, newValue) {
    if (attrName === 'name') {
      this._updateContent();
    }
  }

  _updateContent() {
    this.innerHTML = '<h1>Hello ' + (this.name || 'stranger') + '!</h1>';
  }
}
customElements.define('hello-heading', HelloHeading);
```

Listing 2.2: A custom element with property-attribute synchronization

`<hello-heading>` element. However, a simple custom element lacks many aspects of encapsulation, which can be fixed by utilizing the shadow DOM.

2.2.2 Shadow DOM

Typically, the identifiers and class names of DOM nodes are stored in the document's global shared namespace. This causes some issues if we want to provide a self-contained component and e.g. the `innerHTML` property is used for rendering the content, like in the previous example. [13] When developing a complex web component, it is common to use the `id` attribute to recognize certain nodes inside the component in its implementation code. An application developer who takes this component into her project should not know about the internals of the component, and she can accidentally use the same id somewhere else in her application, causing a conflict. Also, searching for elements with e.g. `document.querySelector` could return an internal part of our component, which should not be reachable when working in the document scope. The consumer of the component should see only the custom element without its internals in the DOM, similarly as the HTML5 `<video>` tag does not reveal what kind of complex structure it consists of.

Styling introduces another problem for a web component implemented as a simple custom element. It is a common pattern to include some internal styling in a web component implementation so that it will behave like intended, especially in terms of layouting [10]. For example, we could have a tab sheet component with CSS rules that place the tabs always horizontally and squeeze them when their combined widths exceed the width of the container. An application developer who uses this tab sheet most probably defines some CSS rules of her own. If these rules match the internal elements, e.g. the tab elements inside the tab sheet, they may break the component's intended behavior.

Shadow DOM fixes these issues by providing an isolated document fragment, which is hidden from the scope of the parent document [13]. Because of this, the same `id` attributes can be used in the shadow DOM scope and at the document level, and querying elements with `document.querySelector` does not return the shadow DOM contents. Also, global styles do not have effect in the shadow DOM, and styles inside the

shadow DOM can not leak outside. As a special case, *CSS custom properties* can penetrate the shadow DOM, allowing the component developer to define some properties that the component user can change from the outside. A draft for *CSS shadow parts* specification has been published by W3C [14], which could make it easier to define styling APIs for web components in the future.

A shadow DOM is connected to its parent element (called the `host`) by attaching a shadow root to it with the `attachShadow` function [12]. Listing 2.3 presents some modifications to our previous `<hello-heading>` example, applying simple styling to the heading and wrapping the content into a shadow DOM. In the element's constructor, a shadow root is attached to it with the `open` mode, which allows us to modify the shadow root's contents in the implementation code. The `closed` mode would deny scripting access to the shadow DOM, which is not very useful in practice [13]. Once the shadow root is attached, it can be referenced by the element's `shadowRoot` property [12]. To populate the shadow DOM, we are adding a `<style>` tag with a CSS rule that centers the content of the heading element in the shadow DOM. We are also adding the `<h1>` element itself. The `_updateContent` function is changed to find the `<h1>` element inside the shadow DOM and update the text inside it. It is still called when the `name` attribute changes. Now the `text-align` CSS property can not be changed by the outside world, nor can the `<h1>` element be found in the main document scope.

2.2.3 HTML Templates

Setting the content of a web component as a string may not be the ideal method for declaring HTML markup, nor is constructing the element hierarchy imperatively with the DOM API. It becomes more cumbersome as the complexity of the component grows. That is why the `<template>` element is often used in web component development to declare a reusable fragment of HTML, which can be applied to each instance of the component [8].

```
class HelloHeading extends HTMLElement {

  constructor() {
    super();
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
        h1 {
          text-align: center;
        }
      </style>
      <h1></h1>
    `;
  }

  _updateContent() {
    this.shadowRoot.querySelector('h1').textContent =
      'Hello ' + (this.name || 'stranger') + '!';
  }
  ...
```

Listing 2.3: Encapsulating element's content and styles in a shadow DOM. Only the relevant parts are included. See Listing 2.2 for the rest of the implementation.

The `<template>` element is ignored by browsers when they render an HTML document. Its only purpose is to store a piece of HTML that can be cloned and inserted with JavaScript. The content of a template can be copied and retrieved either by calling the template element's `cloneNode` function or with `document.importNode`. [9] Listing 2.4 presents how we can declare the content of our `<hello-heading>` example in a template and how to clone the template contents when creating a new instance of our component.

Although HTML templates are part of the web component standards, they are not necessary for building web components. They do not bring any extra features to the component users, such as custom elements or shadow DOM encapsulation. It is mostly

```
<template id="hello-heading-template">
  <style>
    h1 {
      text-align: center;
    }
  </style>
  <h1></h1>
</template>

<script>
  class HelloHeading extends HTMLElement {


    constructor() {
      super();
      this.attachShadow({mode: 'open'});
      const template = document.querySelector('#hello-heading-template');
      this.shadowRoot.appendChild(template.content.cloneNode(true));
    }
    ...
</script>
```

Listing 2.4: Declaring the content of a web component in an HTML template. See Listings 2.2 and 2.3 for the rest of the component implementation.

a matter of preference how the developer wants to maintain the contents of the shadow DOM. On one hand, declaring the markup in HTML allows code editors to recognize the syntax and the developers can exploit features such as syntax highlighting and code completion. On the other hand, in its current state, the web lacks proper methods for importing HTML files to each other.

HTML templates used to be more convenient when *HTML imports* [15] were part of the web component specifications. With this feature it made sense to build a web component in a single HTML file which contained the element content in a `<template>` and the required functionality in a `<script>` tag. However, the HTML imports specification was never standardized, and it was later deprecated in Google Chrome [16]. Since

then, ES6 modules have been the more preferred method for importing web components [17]. This feature allows importing a piece of code from another JavaScript file, so the full component implementation should be written in JavaScript. Thus, the component can not include native HTML markup in its source code, and the content must be handled programmatically as in our earlier examples.

3 Consistency Maintenance

In a distributed collaborative editor, changes made by users are transmitted over a network to other collaborating users, in order to keep the application states in sync. Instead of sending the full updated application state, it is often preferable to send a message that contains information of what was changed. As the size of the change is often smaller than the size of the application state, less data needs to be sent over the network. In consistency maintenance theory, these messages are often referred to as *operations*. Transmitting only the necessary pieces of information is especially important in real-time editing, because changes are made often and responsiveness is important.

In the case of plain text editors, the application state is the string value of the text, and operations are additions and deletions to specified indices in the string. In order to keep our system simple, we define only two primitive operations, as presented in Definition 1. Any larger text editing operations, such as pasting text from the clipboard, or replacing all occurrences of a word, can be composed of these two basic operations.

Definition 1. Character-wise text editing operations

insert[i, c] adds character c to index i

delete[i] deletes the character at index i

Following the publications on this subject, we refer to the distributed application instances as *sites*. In order to provide a smooth user experience, operations generated at the local site are executed immediately. It is not acceptable to have noticeable lag while interacting with a text editor. After local execution, the operations are transmitted to the

other collaborating sites, where they are executed after a transmission delay.

Because of these requirements, we can not use a strict memory consistency model. For example, in *sequential consistency* [18], only one process has access to the shared memory at a time, ensuring that all processes have a consensus on the execution order of the operations. As the immediate local execution allows the operations to be executed in different orders at different sites, a distributed text editor should aim for *eventual consistency*. This means that the separated client application states should converge after having all the same information, even though the states might differ while users have received different updates. The editor should still remain responsive for the users at all times.

In a best case scenario, a site has received and executed all the remote operations before generating a new one. In other words, the execution order of operations is the same at all sites. For example, consider two users editing text with an initial state 'A'. The first user adds 'B' to the end of the text, generating operation `insert[1,B]`. This operation is executed immediately at the local site, changing the text to 'AB'. The operation is then transmitted to the second site, and once received after the transmission delay, it is executed, changing the text to 'AB' also. Now, if the second user writes letter 'C' to the start of the text, the same process happens in the opposite direction, and eventually both sites will end up with text 'CAB'. This is illustrated in Figure 3.1.

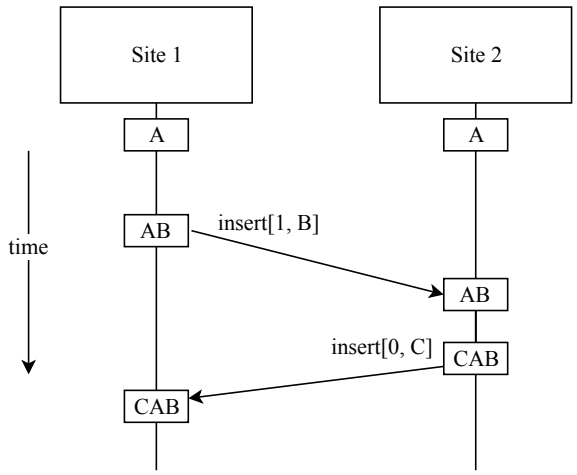

Figure 3.1: Transmitting operations in a best-case scenario

Problems arise because the distributed nature of the system allows concurrent updates. Consider how the previous example would change if both of the users would generate the operations at the same time (or within the transmission delay). Site 1 generates and executes operation `insert[1,B]`, changing the text to 'AB', at the same time as site 2 generates and executes operation `insert[0,C]`, changing the text to 'CA'. After receiving the remote operations from each other, the first site executes `insert[0,C]`, changing the text to 'CAB', but when site 2 executes `insert[1,B]`, it ends up with 'CBA'. This is illustrated in Figure 3.2. The states at the two sites diverged because the operations were executed in different orders. For both operations, the text was in a different state when executed at site 1 compared to when it was executed at site 2. Consistency maintenance techniques are needed to resolve problems such as this.

Later in this chapter, we will review two methods for consistency maintenance: operational transformation and commutative replicated data types. Based on the extensive amount of academic work and real-life applications on this topic, these are the most suitable approaches for real-time text editing applications [19]. Operational transformation has been the core technique for this purpose since it was first introduced in 1989 [20].

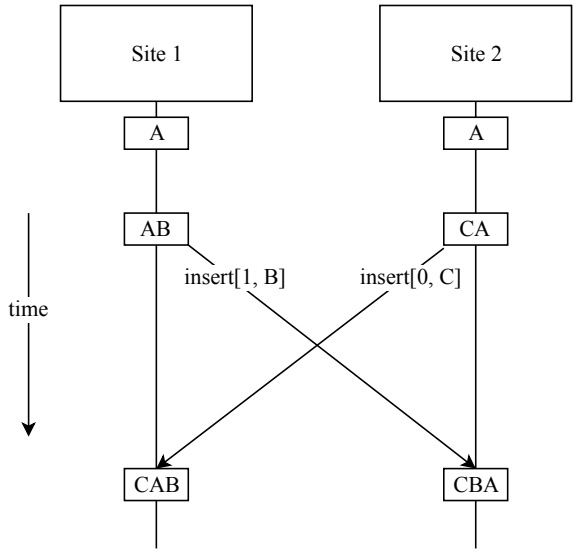

Figure 3.2: Diverging documents after concurrent inserts

Over the decades, multiple complex variations have evolved to fix the problems in the first algorithms and to support more features such as undoing an operation [21]. Commutative replicated data types were first introduced around 2006 [22], and were claimed to be a superior alternative to tackle the correctness and complexity issues of operational transformation. However, this technique has its own problems, which have prevented it from being adopted in most real-life applications [19]. Traditional methods for managing state in distributed applications, such as transactions and locking, are not considered in this study, as they do not satisfy the requirements for this kind of applications, such as immediate local response [20]. Before jumping into the consistency maintenance techniques, we will take a look at the criteria for a consistent group editor.

3.1 Consistency Criteria

Before developing a consistency maintenance system for a collaborative editor, we need to define a set of criteria that we need to fulfill so that the system can be called consistent. In many studies, this set of criteria consists of causality preservation, convergence and intention preservation. [23]

3.1.1 Causality Preservation

To preserve the causality of operations, they must be executed in their natural cause-effect order. Because operations are transmitted independently, it is possible that they arrive at a site out of order. If operation O_1 is the cause for a later operation O_2 to happen, it is not meaningful to execute O_2 before O_1. For example, consider O_2 deleting the character inserted by O_1.

In a distributed system, it may be hard to determine a total ordering of events happening concurrently at different sites. However, concurrent operations are also independent of each other, so we do not have to worry about their execution order to preserve causality.

A partial ordering can be defined for operations in a distributed system. In this ordering, operation O_1 precedes another operation O_2, if O_1 was executed at the site that generated O_2 before the generation of O_2. This is also called the *happened-before* relationship, and it is denoted with $\rightarrow$. If $O_1 \rightarrow O_2$, it is possible that O_1 causally affects O_2. [24]

To preserve the causality of operations, the system must satisfy the *precedence property*. It states that if O_1 precedes O_2, then O_1 must be executed before O_2 at all sites. This ensures that no operation will be executed in a state where something that causally affects the operation has not yet happened. [20]

3.1.2 Convergence

The precedence property ensures that dependent operations are executed in order, but because the ordering is only partial, independent operations can still be executed in different orders at different sites. One operation can produce a different outcome when applied to a different state, causing the sites to diverge from each other. This was already demonstrated in a previous example and Figure 3.2.

To be considered consistent, the system must satisfy the *convergence property*. It states that the sites must have identical states when they have executed the same set of operations. [20] This is clearly an essential property for a groupware system. If the sites would diverge little by little with each operation, after a while the users would not be editing the same text anymore, and the state of the document would be ambiguous.

3.1.3 Intention Preservation

In addition to divergence, the fact that concurrent operations may be executed in different orders causes another problem; intention violation. The effect of executing an operation should follow the initial intention of the user, regardless of the document state it is executed in. [25]

Consider the earlier example, where one user performed `insert[1,B]`, and the second user performed `insert[0,C]`, on the initial document state 'A'. In this case, the intention of the first user is to add letter 'B' after the letter 'A', and the intention of the second user is to add letter 'C' before the letter 'A'. When site 2 executes the remote operation `insert[1,B]` after its own operation, the effect of this operation is to add letter 'B' before the letter 'A'. This was not the intention of the first user.

It might seem that intention preservation comes hand in hand with the convergence property, but this is not the case. The system can make the document states converge without caring about the intentions of the users. For example, the system could decide that each site should end up with the second user's final state 'CBA'. The states would converge, but each site would violate the first user's intention of adding 'B' after 'A'.

3.2 Operational Transformation

Operational transformation (OT) has been the state-of-the-art method for maintaining consistency in groupware editors since it was introduced in 1989 [20]. Since then, research groups have developed many improvements and variations of this techniques, using the same base idea [23]. Multiple real-life group editor products, such as Google Docs [26], use algorithms based on OT. This technique is not restricted to text editing, but can be also used in other applications, such as 3D modeling [27].

The basic idea of operational transformation is to transform concurrent operations in a way that the effect of executing the operation is the same as at the site where it was generated. For this purpose, we need a transformation function and a control algorithm. The control algorithm is generic and can be used for multiple kinds of group editors. It decides when to transform an operation with another one. The transformation function, on the other hand, is application-specific.

The rest of this section is organized as follows. Next, we will take a look at the basic transformation function for text editing, as defined in the original OT algorithm called dOPT (distributed OPerational Transformation) [20]. Then, we describe how OT utilizes a data structure called *state vector*. After this, we will review some of the OT algorithms. We are intentionally omitting techniques that require server-side functionality, such as Jupiter [28] and the Google Docs algorithm [26], as distributed consistency maintenance is a requirement for our own groupware implementation.

3.2.1 Transformation Function

When a site receives an operation O_1, which is concurrent with an already executed operation O_2, O_1 is not executed as is. Instead, the site transforms O_1 against O_2 with a transformation function T, to produce the transformed operation O_1':

$$O_1' = T(O_1, O_2)$$

To ensure convergence, the dOPT developers defined a transformation property (later labeled as TP1). It states that executing O_1' after O_2 must produce the same document state as executing O_2' after O_1. [20] It was later shown that an additional transformation property TP2 is required to ensure convergence along any path taken in the operation space [29]. Both properties are formally listed in Definition 2, where $\circ$ indicates the composition of operations and $\equiv$ indicates the equivalence of the resulting document states. We return to this topic in Section 3.2.3.

Definition 2. Transformation properties

Transformation Property 1 (TP1):

$$O_1' \circ O_2 \equiv O_2' \circ O_1$$

Transformation Property 2 (TP2):

$$T(T(O_3, O_1), T(O_2, O_1)) \equiv T(T(O_3, O_2), T(O_1, O_2))$$

In our example in Figure 3.2, site 2 ended up with 'CBA' instead of the expected state 'CAB'. The remote operation received from site 1 was `insert[1,B]`. To achieve convergence, site 2 should insert 'B' to index 2 instead. The transformation function should make this change when transforming the remote operation against the locally executed concurrent operation:

$$T(insert[1, B], insert[0, C]) = insert[2, B]$$

In general, when a concurrent insert operation has been executed with an index smaller than in the current operation, the index should be incremented.

At site 1, there is no problem in executing the concurrent remote operation as is, because a previously inserted character at a greater index does not affect the insert operation. In this case, the transformation function should return the original operation:

$$T(insert[0, C], insert[1, B]) = insert[0, C]$$

If both of the insertions would happen at the same index, we would need to break the tie by deciding which operation is shifted. In the dOPT algorithm (and most of the others as well), the sites have unique identifiers which are used for sorting the conflicting indices.

We have now described how to transform an insert operation against another insert operation. With the delete operation, there are four permutations for a pair of concurrent operations to be handled by the transformation function. All of them have the basic idea that the index needs to be shifted if the earlier concurrent operation happened at a smaller index. Inserting and deleting at the same index are special cases. The insert tie needs to be resolved like described earlier, and deleting a character at the same index results in no operation (the same character cannot be removed twice). The full set of transformations is listed in Listing 3.1.

```
Tii(insert[i1,c1], insert[i2,c2], id1, id2) {
    if (i1 < i2  or (i1 == i2 and id1 < id2)) {
        return insert[i1, c1]
    } else {
        return insert[i1 + 1, c1]
    }
}
Tid(insert[i1,c1], delete[i2]) {
    if (i1 <= i2) {
        return insert[i1, c1]
    } else {
        return insert[i1 - 1, c1]
    }
}
Tdi(delete[i1], insert[i2, c2]) {
    if (i1 < i2) {
        return delete[i1]
    } else {
        return delete[i1 + 1]
    }
}
Tdd(delete[i1], delete[i2]) {
    if (i1 < i2) {
        return delete[i1]
    } else if (i1 > i2) {
        return delete[i1 - 1]
    } else {
        return identity operation
    }
}
```

Listing 3.1: Basic transformations of insert and delete

3.2.2 State Vectors

In order to satisfy the precedence property, an operation can not be executed until all of its dependent operations have been executed. Operational transformation algorithms use state vectors (closely related to vector clocks [30]) for detecting the happened-before

relationship between the operations, and to decide whether an operation is ready to be executed. [20], [31]

Following the definition in [31], a state vector SV is an array of logical clocks, one for each of the participating sites. Each site maintains its own replica of the state vector. Initially all of the values are zero. When a site executes an operation generated at site i, the executing site increments the logical clock of the site that generated the operation: $SV[i] := SV[i] + 1$. Before broadcasting a generated operation to other sites, it is timestamped with the current value of the state vector.

The causal readiness of an operation received from a remote site is resolved as follows. Let SV_o be the timestamp of the received operation, i the index of the remote site that generated the operation, k the index of the receiving site and SV_k the state vector of site k. An operation received from a remote site is causally ready to be executed once it fulfills two conditions:

1. $SV_o[i] = SV_k[i] + 1$
2. $SV_o[j] \leq SV_k[j]$, for all $j \neq i$

The first condition makes sure that site k has executed all of the previous operations from site i, and the second condition ensures that site k has executed all of the operations from the other sites that may have causally affected the operation. If the operation depends on another operation that site k has not yet executed, the timestamp vector contains a larger value than the current state vector SV_k. [31]

An example is illustrated in Figure 3.3. Site 1 generates an operation, updates its state vector, and sends the operation with timestamp `[1,0,0]`. This is received by site 2, which executes the operation and updates its state vector. Site 2 then generates a new operation with timestamp `[1,1,0]`. This is received by site 3 whose state vector is still `[0,0,0]`. By comparing its state vector to the incoming timestamp, site 3 recognizes that the received operation depends on another operation generated at site 1, which site

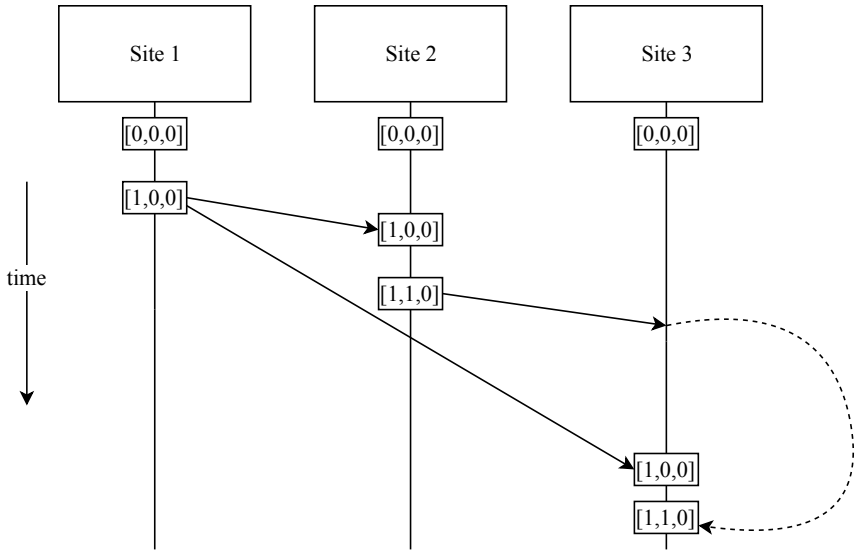

Figure 3.3: Preserving causality with state vectors. Site 3 needs to postpone the execution of the received operation.

3 has not yet executed. Thus, the operation is not causally ready and it is postponed. Later, when site 3 has received and executed the first operation, it compares the queued operation with the updated state vector. At that point it is causally ready, so site 3 executes it and updates its state vector.

Besides determining if an operation is causally ready to be executed at a site, the OT algorithms need state vectors to figure out the dependency relation between two operations. Following [32], we can determine if operation O_1, generated at site i, causally precedes another operation O_2, generated at site j, as follows:

$$O_1 \rightarrow O_2 \text{ iff } SV_{O_1}[i] \leq SV_{O_2}[i]$$

3.2.3 dOPT and adOPTed Algorithms

As many later algorithms, dOPT maintains a log of executed operations and a queue of postponed operations at each site. After generating and executing a local operation, it is appended to the log and broadcasted to the other sites. When receiving a remote operation, the site compares the operation's timestamp to its own state vector to decide if

the operation is causally ready. A ready operation can be executed and appended to the log immediately. A non-ready operation is appended to the queue instead. Later, after other operations have been executed, the queue is scanned to see if some postponed operations are now causally ready. Those are then removed from the queue and executed. [20]

Before executing an operation O, dOPT scans the log for all of the concurrent operations of O that the site has already executed. This is done by comparing the timestamps as described in Section 3.2.2. O is then transformed against the concurrent operations and the transformed operation O' is executed and appended to the log. [20] This is illustrated in Listing 3.2.

```
1 ExecuteOperation(O, LOG) {
2   O' = O
3   for each C in LOG which is concurrent with O {
4     O' = T(O', C)
5   }
6   execute O'
7   append O' to LOG
8 }
```

Listing 3.2: Integrating a causally ready operation in the dOPT algorithm

This simple idea of transforming the operation against all of its concurrent ones has been proven to be faulty. The documents fail to converge in some cases, where one operation is concurrent with two or more dependent operations. [28], [29] The flaw has been later referred to as the "dOPT puzzle", and multiple separate solutions to it have been developed.

Ressel et al. [29] solved the dOPT puzzle in their adOPTed algorithm. They recognized that the transformation property TP1 defined with the dOPT approach is not enough to ensure convergence, and they defined TP2 (presented earlier in Definition 2). The adOPTed algorithm maintains an N-dimensional graph (where N is the number of sites) called the *interaction model*. Vertices in the interaction model represent the docu-

ment states and the edges represent the operations, moving from one state to another. This complex data structure keeps track of all the different paths of original and transformed operations that can be taken to reach the different document states. We do not go into more detail with this approach, as the added space complexity of the interaction model is not necessary for solving the dOPT puzzle, as proven by the GOT algorithm [25].

3.2.4 GOT Algorithm

The GOT (General Operational Transformation) approach, by Sun et al. [25], was the first one to include intention preservation in its correctness criteria. A total ordering of operations and an *undo/do/redo* scheme are defined to satisfy the convergence property. The operational transformation algorithm is needed just to preserve user intentions.

If the same sequence of operations are executed in the same order at different sites, convergence is clearly achieved. The GOT approach defines a total order for operations based on their timestamps and site identifiers, as described in Definition 3. [25]

Definition 3. Total ordering in GOT

$$O_1 < O_2 \text{ iff } sum(SV_{O_1}) < sum(SV_{O_2}) \text{ or } (sum(SV_{O_1}) = sum(SV_{O_2}) \text{ and } i < j)$$

where the $sum()$ function sums all elements in a given state vector, O_1 was generated at site i and O_2 was generated at site j.

GOT allows sites to execute operations in any order, but still maintains the total order by carrying out the following undo/do/redo process for executing a causally ready operation O:

1. Undo locally executed operations that follow O according to Definition 3.
2. Execute operation O.
3. Redo all reversed operations.

Undoing and redoing are just internal operations that allow executing O in its correct place in the history, and only the final result is displayed to the user. [25]

In GOT, the transformation functions defined in Section 3.2.1 are called *inclusion transformations (IT)*, as they transform an operation O_1 against O_2 so that the impact of O_2 is included in O_1'. This term is necessary to differentiate them from *exclusion transformations (ET)*, which transform O_1 against O_2 so that the impact of O_2 is excluded from O_1'. It was recognized that an inclusion transformation works correctly only when both of the operations are defined in the same context, i.e. the same document state. Omitting this fact can be seen as the root of the dOPT puzzle. The exclusion transformation removes the effect of the previous dependent operation, which allows changing the operation's context to a state where inclusion transformation can be applied. Formally, the inclusion and exclusion transformations need to satisfy the pre- and postconditions which are presented in Definition 4. [33]

Definition 4. Transformation pre- and postconditions
Inclusion transformation $IT(O_1, O_2) : O_1'$

1. Precondition for the parameters: O_1 and O_2 are *context-equivalent*, i.e. defined in the same document state.

2. Postconditions for the result: O_2 is *context-preceding* O_1', i.e. O_1' is defined in a context which results from applying the effect of O_2 on its definition context, and the effect of O_1' in this new context is the same as the effect of O_1 in the original context.

Exclusion transformation $ET(O_1, O_2) : O_1'$

1. Precondition for the parameters: O_2 is context-preceding O_1.

2. Postconditions for the result: O_1' and O_2 are context-equivalent, and the effect of O_1' in this new context is the same as the effect of O_1 in the original context.

The GOT control algorithm executes transformations on a causally ready operation O based on three possible scenarios:

1. Each operation in the log[1] is causally preceding O.
2. All the concurrent operations of O are in the end of the log, after the operations that causally precede O.
3. There is at least one operation causally preceding O in the log after a concurrent operation of O.

In case 1, O can be executed without any transformations. In case 2, the effects of the concurrent operations need to be included to O by using inclusion transformations. These two cases work the same way as in the dOPT algorithm, but dOPT failed to handle the third case. In the third scenario, ET functions are needed to make two operations context equivalent, which is the precondition for applying the inclusion transformation. [25]

Consider a log consisting of operations O_1 and O_2 respectively. A new operation O_3 arrives, which is dependent on O_2, but concurrent with the earlier operation O_1. To include the effect of O_1, we need to transform O_3 to the same context where O_1 was defined, i.e. exclude the effects of O_1 and O_2 from O_3. First, the effect of O_1 needs to be excluded from O_2 by running $O'_2 = ET(O_2, O_1)$, after which we can exclude both operations from O_3 with $O'_3 = ET(O_3, O'_2)$. Now we have O'_3 which is context equivalent with O_1, so we can include the effect of O_1 into the new operation with $O''_3 = IT(O'_3, O_1)$. The resulting O''_3 is context equivalent with O_2 (initial document state plus the effect of O_1), and we can get the final execution form of the new operation with $O'''_3 = IT(O''_3, O_2)$. We omit the exhaustive description of the algorithm and how it integrates with the undo/do/redo scheme, which can be found in [25].

[1] In the GOT paper [25], the log of executed operations is actually called *history buffer (HB)*, but we are sticking with the original term by Ellis and Gibbs [20] in this text to avoid obscurities.

The developers of the GOT algorithm used string-wise operations in their implementation (see Definition 5) to reduce the amount of required transformations and network traffic. Their transformation functions still follow the same idea of shifting indices based on earlier insertions and deletions, but handling overlapping operations require a lot more maintenance. For example, consider operations $O_1 = delete[2, 1]$ and $O_2 = delete[0, 5]$. Running the inclusion transformation is simple, as O_2 nullifies the effect of O_1: $O_1' = IT(O_1, O_2) = identity\ operation$. The problem is that we need to be able to revert this process with the exclusion transformation, but we can not know the original parameters of O_1 based on O_1' and O_2. For situations like this, additional data structures are needed for saving and retrieving the lost information [33]. It is important to remember that the general purpose OT control algorithms can be used with any type of operations and transformation functions, as long as they satisfy the transformation properties and the pre- and postconditions.

Definition 5. String-wise editing operations

insert[i, s] adds string s starting at index i

delete[i, n] deletes n characters starting at index i

Sun et al. also provide a garbage collecting scheme for removing old operations from the log [25]. This is an important feature, because the log grows quickly in an editing session and has a negative effect on the algorithm performance. The basic assumption behind the garbage collection is that when we are confident that the operation in the beginning of the log will be causally preceding all forthcoming operations, it will not be needed by any upcoming transformations nor the undo/do/redo scheme, and thus it can be removed. For this purpose, each site must maintain a *State Vector Table (SVT)* and a *Minimum State Vector (MSV)*. After executing a remote operation, a site updates the vector corresponding to the originating site in its SVT to match the operation's timestamp. This way, each site can keep track of the state vectors at other sites. A periodical *state message* should be broadcasted by a site that hasn't generated a new operation for a while,

in order to still inform other sites about its state from time to time. Each time the SVT is updated, the values in the MSV are updated to reflect the minimum of the state vectors:

$$MSV[i] = min(SVT[0][i], ..., SVT[N-1][i]) \text{ for all } i \in 0, 1, ..., N-1$$

The oldest operation in the log, generated at site i and timestamped with SV_O, can be removed from the log if $SV_O[i] \leq MSV[i]$.

3.2.5 GOTO - GOT Optimized

The way in which the GOT control algorithm handles the dOPT puzzle requires a lot of transformations, and integrating with the undo/do/redo scheme makes the implementation quite complex. Sun and Ellis [23] later improved how the control algorithm handles the case where dependent operations of a new operation have been executed after concurrent ones. This method was named as GOTO (GOT Optimized). It reduces the amount of needed transformations, simplifies the basic design of the algorithm, and makes the undo/do/redo scheme obsolete for achieving convergence.

The basic idea of GOTO is that before executing a causally ready operation O, the log is reordered so that all the concurrent operations of O are in the end of the list. Inclusion and exclusion transformations need to be applied in the process to keep the current context valid. In other words, executing the operations in the modified log in order should produce the current document state. After the reordering, inclusion transformations can be applied directly against the concurrent operations, like in the dOPT algorithm and the GOT algorithm's case 2, as described in Section 3.2.4. [23]

To enable reordering the log in a context-preserving manner, a utility function called *Transpose* was introduced in [23]. It takes two consecutive operations O_1 and O_2 in the log and swaps their order as described in Listing 3.3. To bring the latter operation O_2 into the context before O_1, the effect of O_1 is excluded from it to produce O_2'. At this point O_1 and O_2' are defined in the same context, so we need to include the effect of the new O_2' to O_1 so that it can be ordered after O_2'.

To demonstrate GOTO, let O be a causally ready remote operation to be executed, O_C the first operation in the log which is concurrent with O, and O_P such an operation causally preceding O which is placed after O_C in the log. O_P is shifted in front of all the concurrent operations of O by applying the *Transpose* function repeatedly, moving O_P backwards in the log one step at a time, until it is positioned right before O_C. This is repeated for all the causally preceding operations of O which are located after O_C in the log. The GOTO control algorithm is presented in Listing 3.4, in a slightly modified form compared to how it was originally described in [23].

```
1 Transpose(O1, O2) {
2   O2' = ET(O2, O1)
3   O1' = IT(O1, O2')
4   return (O2', O1')
5 }
```

Listing 3.3: Transpose function for reordering the log

3.3 Commutative Replicated Data Types

Commutative replicated data types (CRDT) are consistency maintenance techniques much different from operational transformation. A CRDT is an internal data structure that maintains the document state and has only commutative operations. [34] Commutativity means that executing the same set of operations produces the same result, regardless of the execution order. As a simple example, consider a replicated integer data type with an initial value 0, and operations *add* and *subtract*. Concurrent operations add(5) and subtract(3) converge to value 2 at each site, because add and subtract commute.

Commutativity makes the transformations and complex control algorithms of OT redundant, as well as removes the need to maintain state vectors. One motivation behind the first CRDTs was to enable massive collaboration in P2P networks, which is not possible

```
GOTO(O, LOG) {
  C = the first operation in LOG which is concurrent with O
  if (C is not found) {
    return O
  }
  PRECEDING = list of operations in LOG after C which are causally preceding O

  C_IND = index of C in LOG
  for each P in PRECEDING {
    P_IND = index of P in LOG
    for (i = P_IND; i > C_IND; i--) {
      (LOG[i-1], LOG[i]) = Transpose(LOG[i-1], LOG[i])
    }
    C_IND++
  }
  O' = O
  for (i = C_IND; i < LOG.length; i++) {
    O' = IT(O', LOG[i])
  }
  return O'
}
```

Listing 3.4: GOTO control algorithm

when the sizes of the state vectors grows with the amount of participants [22]. The challenge is to design the data type and commutative operations for a text document. Next, we will review some of the well-known CRDTs, and how they ensure commutativity.

3.3.1 WOOT

As an internal data model, WOOT [22] maintains a set of objects called W-characters, presented in Definition 6. As each W-character has a reference to the previous and next W-characters, the WOOT data structure can be represented as a Hasse diagram. See Figure 3.4 for an example.

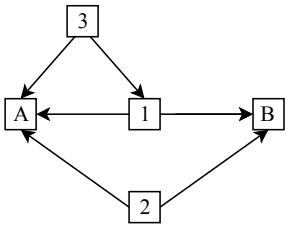

Figure 3.4: A Hasse diagram of a WOOT document

Definition 6. W-character

A W-character is a five-tuple $< id, \alpha, v, id_p, id_n >$, where id is its universally unique identifier (UUID), α is its alphabetical value in the document, v is its visibility flag, and id_p and id_n are the identifiers of the previous and next W-characters.

Each site has a globally unique identifier (such as the network address) and a logical clock, which is incremented with each generated operation. The character identifier is made globally unique by combining the site identifier with the logical clock. Since each character has a globally unique identifier, the insert and delete operations can refer directly to these objects, instead of indices in the document:

Definition 7. WOOT operations

insert[a $\succ$ e $\succ$ b] inserts element e between a and b

delete[e] sets the visibility of e to false permanently, thus making it a *tombstone*

The operation pairs insert-delete and delete-delete are natively commutative, so they can be executed in any order to produce the same outcome. To achieve convergence, only insert-insert needs special handling for some cases. The $\succ$ relation defines only a partial order, so characters inserted in the same spot are ordered by their identifiers. This still leaves a problem which can be seen in Figure 3.4. Three operations were executed on the initial document 'AB': insert[A $\succ$ 1 $\succ$ B], insert[A $\succ$ 2 $\succ$ B] and insert[A $\succ$ 3 $\succ$ 1]. The order of the character identifiers is $id(1) < id(2) < id(3)$. In this situation, if a site receives the second operation as the last one, it has two valid options for ordering: insert '2' after '1' or before '3'. Because '3' depends on '1', it's known that the first operation

happened before the third one. WOOT uses this information to do the ordering based on the earlier character '1'. Thus, the document state is resolved to 'A312B'. [22]

Satisfying the rest of the consistency criteria is quite effortless with the ability to uniquely identify each character. An insert operation is causally ready when both the previous and next characters (or their tombstones) are present, and a delete operation is causally ready when the element to delete (or its tombstone) is present. State vectors are not needed for verifying causality. The user intentions are preserved by inserting the character between the same characters as at the original site, and deleting the same character instance as at the original site. [22]

3.3.2 Logoot

Logoot [35] is another CRDT which attaches a universally unique identifier to each member of a linear data structure. In the original paper, these members are lines of text in a document, but we can as well increase the granularity by identifying each character. The data structure of the identifiers is presented in Definition 8. The operations are similar to WOOT; a character is inserted between two existing characters and a character is removed by its identifier. Thus, causality preservation and intention preservation are also ensured in a similar fashion.

Definition 8. Logoot UUID

Logoot identifiers are lists of pairs $< pos, site >$, where pos is an integer and $site$ is a site identifier.

Logoot achieves commutativity by defining a total ordering of these sequences of pairs. In order to satisfy intention preservation, i.e. insert the typed text into the intended spot, Logoot must always be able to generate a new unique identifier between the previous and the next character. This is why the data structure is a list, which can be extended indefinitely. At first, the identifiers consist of only one pair. When a new character is inserted between existing characters identified by $< pos1, site1 >$ and $< pos2, site2 >$,

the new identifier is created by generating a random integer between $pos1$ and $pos2$. If such a number can not be generated, the UUID is created by extending the UUID of the previous character with a new pair. [35]

3.3.3 Treedoc

Treedoc [36] uses a binary tree as its data structure, which enables unique identifying and extensibility. Each node in the tree contains a character, which can be identified by the path from the root of the tree to that node. The document is constructed from the data structure by traversing the tree in infix order. This ensures that the tree can be always extended by a new leaf node, so that the new character ends up in the correct spot in the text. In case of concurrent inserts at the same position, multiple *mini-nodes* are inserted into the same node. These can be ordered e.g. by the site identifiers.

One issue with Treedoc is that the tree becomes easily unbalanced, which grows the sizes of the identifiers. For example, when a user types a sentence to the end of the document, each character grows the tree to the right-hand side of the previous one. A couple of methods for balancing the tree were proposed in [36]. The first idea is that instead of appending the new leaf node directly to the previous one, a larger sub-tree can be generated, and the new character appended to the left-most node in the sub-tree. The other proposed method is a procedure which flattens the tree while keeping the order relations. The downside of this procedure is that it requires a commitment from each site in order to commute with the insert and delete operations.

3.4 Comparison of Methods

Since its inception, the use case for OT has been real-time collaboration among a relatively small set of peers ($1 \leq N \leq 5$) known to each other [20]. CRDT, on the other hand, was originally designed to enable conflict-free collaboration for a massive user group in

non-real-time systems, such as wikis [22], [35]. Still, it is often considered as a simpler alternative to OT for real-time editing. In this section, we compare OT and CRDT from the point of view of real-time text editing by a reasonable number of collaborators.

In their recent study [19], Sun et. al. note that despite CRDTs having been available for over a decade, they are rarely utilized in actual cooperative text editor products. Meanwhile, OT remains as the more common technology. One recognized practical problem with CRDTs is that these data types are not natively supported by text editors. The internal data structures of text editors are optimized for text editing and index-based operations. To integrate a CRDT into a text editor, one must maintain an extra data structure of the CRDT in addition to the one maintained by the editor. Changes in the CRDT must be converted to index-based operation on the text, which nullifies some of the claimed performance benefits of CRDTs.

Table 3.1 presents the real space and time complexities between the operational transformation technique GOTO, and the commutative replicated data type WOOT, while taking into account the conversions from CRDTs to index-based operations [19]. In the table, N is the number of collaborating peers, C is the number of concurrent operations involved in transforming a remote operation and S is the size of the document (including WOOT's tombstones). Although the compared methods are not the most optimized variations of OT and CRDT, these results show the general principles of OT and CRDT complexities. The efficiency of OT is determined by the number of concurrent operations and the number of peers, while CRDTs are bounded by the size of the content.

	GOTO	WOOT
Space complexity	$O(C * N)$	$O(S)$
Time complexity of integrating a local operation	$O(1)$	$O(S^3)$
Time complexity of integrating a remote operation	$O(C^2)$	$O(S^3)$

Table 3.1: Comparison of space and time complexities of GOTO and WOOT

The CRDT developers consider tracking the states of collaborators with state vectors to introduce too much overhead for their purposes, in terms of both space and time [22], [35]. On the other hand, CRDTs attach identifiers and possibly other metadata to each entry in the editable data, which is why the space overhead grows with the document size. When a reasonable number of users collaborate on a document on character-level granularity, the increased memory requirements of character identifiers clearly exceed those of the state vectors. The time complexity of state vector changes is not a problem either, when a typical amount of users collaborate together. In practice, the input parameters for OT complexity functions are relatively small in real-time group editors ($N \leq 5$ and $C \leq 10$) [19]. The number of characters in the text S, on the other hand, can grow orders of magnitude larger.

It seems that OT is the better approach for implementing a real-time text editor for a small set of peers. The main benefit of CRDTs is that they do not need complex concurrency control between collaborators, but this is not a major issue if we expect only a few users to collaborate at a time. The theoretical performance benefits of CRDTs fail in practice due to converting the identifier-based operations to index-based text editing operations. The added memory overhead is also considerable when working with data of high granularity. Maintaining identifiers and additional metadata with each character multiplies the memory consumption of the document.

4 Implementation

4.1 Requirements

In the scope of this thesis, the collaborative web component will work only with plain text. Supporting rich text formatting, i.e. bolding text, changing the font, adding headings and lists etc., would add a lot more use cases for the component. However, it also introduces additional complexity to consistency maintenance, which is why we leave the rich text support for future work. The component should have an API for plugging it into a web application as easily as possible. After connecting the clients, the component should transmit changes in the text to other users, and receive the changes made by other users. These changes should be integrated to the local document, while maintaining consistency according to the criteria described in Section 3.1. The caret positions of collaborating users should be also displayed in the editor.

Consistency maintenance should be handled in a distributed fashion in client side, i.e. inside the web component's JavaScript code running in the browsers of the collaborating users. The similar existing UI component products, reviewed in Chapter 1, use a centralized architecture. This forces the application developer to either setup the required technology stack on their servers, or to rely on the provider's service in the cloud. Even though using a provided service can be convenient in many cases, setting that kind of limitations for server-side functionality is not a good feature for a web component. It should be up to the application developer to choose how to handle the routing of mes-

sages between clients, e.g. via their own server or with a P2P browser communication solution, such as WebRTC [37]. In Chapter 3, we learned that there are many methods for maintaining consistency even without a central server.

4.2 Selecting the Technologies

4.2.1 Editor Core

Building a new text editor from scratch would not be worthwhile, as multiple solutions already exist. To focus on the main purpose of this thesis, we want to build the collaboration functionality and API on top of an existing web-based text editor.

Native Web Approaches

For editing plain text, the native HTML `<textarea>` element would seem to be the most appropriate core for the component. The problem is that it is not possible to render other DOM elements inside a `<textarea>`. We need this kind of functionality in order to display the carets of collaborating users.

Another way of building a text editor with native features of the web browsers is the `contenteditable` attribute [9]. Setting this attribute to an element allows users to edit text and HTML inside it with keyboard shortcuts for e.g. bolding and other rich text features. Rendering carets inside this kind of an element would be possible, but maintaining their position programmatically might be cumbersome, as users can delete the caret elements as well as any other HTML contents. The HTML editing has been reported to work very inconsistently in different browsers [38], and we don't want to support rich text anyway at this point. In the latest W3C specification draft [39], a `plaintext-only` state for `contenteditable` is proposed, but it has not been standardized.

Both the `<textarea>` and elements with the `contenteditable` attribute fire `input` events when the user is editing the content. From these events it is possible

to parse the information of what was added or removed and where. Another event, `beforeinput`, is introduced in a W3C draft [40]. Its purpose is to allow overriding the default browser behavior by cancelling the upcoming `input` event and executing custom logic instead.

Custom Editors

Because the editor features of the web are not in a stable state, the web-based editors often contain their own JavaScript implementations for handling user input and rendering the document in the DOM. Executing the same custom code in all browsers avoids the differences in their default implementations.

There exist several reusable text editor components for web application developers, such as CKEditor [4], TinyMCE [41] and Quill [6]. These are all designed for rich text editing, with easily configurable toolbars. A couple of notable editors focused on code editing in the browser are Ace [42] and CodeMirror [43]. Out of these options, Quill was chosen as the editor core for several reasons:

- Permissive BSD 3-Clause license
- Easy to configure into plain text mode
- Provides insert and delete events in a convenient form for our purpose
- Already has a module for rendering multiple carets [44].

4.2.2 Consistency Maintenance Technique

In Section 3.4, we already concluded that operational transformation is the most appropriate solution for our use case. In Section 3.2, we reviewed some of the OT algorithms, but did not perform an exhaustive search of these techniques. We want the algorithm to have good performance and preferably be relatively easy to implement. In the future, we

might want to implement an undo feature for our editor. Undoing the latest executed operation would be straightforward, but in a collaborative editor we expect to undo the last operation generated by the local site. Undoing the latest local operation, when remote operations have been executed after it, requires some more complex control logic, in order to not change the effect of the later operations [21].

Out of the reviewed algorithms, GOTO satisfies our requirements best and was selected for the implementation. The dOPT algorithm would not suffice due to its correctness issues, and GOT is just a lower-performance and more complex version of GOTO. Compared to adOPTed, GOTO maintains consistency by using a more effective linear data structure for logging the operations, instead of an N-dimensional graph. Even though the GOTO approach requires additional exclusion transformation functions, those should be relatively simple to implement for text editing operations. An algorithm called ANYUNDO [21] has been invented to enable group-undo feature with the GOTO algorithm. We can integrate it to our editor in the future.

4.3 API Design

The API of our web component is the public interface which allows application developers to use its features. In practice, the API consists of the custom element tag name, the component's public functions, attributes and properties, and the events it fires. The attributes can be set declaratively in HTML, while properties can be modified only with JavaScript, although these are usually synchronized (see Section 2.2.1). The component may fire event objects with metadata related to that event. The user can add handlers for these events with the native `addEventListener` function. To clearly and compactly communicate the core purpose of the component, being a collaborative text editor, and to satisfy the requirement of custom elements having a dash in the tag name, we are going to call the component `<co-editor>`. Accordingly, the corresponding JavaScript class

will be named as `CoEditor`.

In order to effectively integrate `<co-editor>` into a web application, we need to provide the application developers with the following set of features:

1. Managing a collaborative editing session
2. Transmitting updates between collaborators
3. Setting and reading the text in the editor
4. Defining a username which is displayed in the carets of collaborating editors.

For session management, we need to have one editor which provides identifiers for joining editors. Uniquely identifying each client is a requirement of the consistency maintenance, and there needs to be a single source of truth which generates these identifiers to ensure consistency. We call this session managing editor the *master* editor. For setting this quality for a component instance, the component has a boolean attribute/property `master`. To generate a new identifier for a joining editor and to provide it with the initial state of the document, the component has `generateJoinMessage` function. This function returns a message containing all the required information, and it should be forwarded to the joining client with the message routing API, which is described next.

When the state of the document changes, e.g. by user typing in some characters, the component should notify its user about the change and provide information required to integrate this change to other editors. The DOM event system is the most suitable solution for this purpose [45]. Every time when there is an update which should be communicated to the other editors, the component fires a `CustomEvent`, with its type being "update". Custom data can be passed into a `CustomEvent` in its `detail` property. The message data is set as the `detail` property in a string format, so it is easy to transmit without further serialization. After transmitting this message over the network, the collaborating editor needs to integrate it. For this purpose, the component has a function called `receive`, which should be called with the message as the parameter.

With this set of API, the `<co-editor>` component can be integrated to a web application by following these steps:

1. Set the first editor in the session as the master.

2. When a new editor wants to join the session, call `generateJoinMessage` on the master editor, transmit the result over the network to the joining client and pass it to the editor's `receive` function.

3. Listen for the update events in each component, transmit the detail messages to each of the collaborating clients and pass them to their `receive` functions.

For setting and reading the text in the editor, as well as setting and reading the username, string properties are the most appropriate solution. Thus, we add two properties, `value` and `username`, to the component. Both of these properties, as well as the boolean `master` property, will be synchronized to reflect the corresponding attribute, as described in Section 2.2.1. The full set of the public API is listed in Table 4.1.

`CoEditor`
`value`: string
`username`: string
`master`: boolean
`generateJoinMessage()`: string
`receive(string)`: void
Fires event: `{ type: "update", detail: string }`

Table 4.1: Component API

4.4 Internal Design

This section describes how the component was implemented. The project is open-sourced and the source code can be found in GitHub[1]. As the plan is to build a web component, our product is essentially a JavaScript class which extends `HTMLElement`. We want to embrace the separation of concerns design principle as much as possible in the implementation, in order to keep the code maintainable. For this reason, the component is built as a hierarchy of classes, each being responsible of a separate part of the functionality, and pieces of the code that do not need to be coupled with the component are implemented as external modules. Each piece of the implementation is written in their own JavaScript files as ES6 modules, defining dependencies between each other with the `import` statement.

Figure 4.1 presents the components of our solution. As JavaScript classes do not have access modifiers, we use a convention of prefixing protected properties and functions with an underscore. These should be used only by other classes in the hierarchy, and they are not part of the component's public API. The class hierarchy starts from `EditorBase`, which handles the text editor integration. This is extended by `OTHandler`, which takes care of the operational transformation by utilizing external functions for executing the GOTO control algorithm with character-wise text editing operations. `SessionHandler` adds functionality needed to maintain the editing session between collaborating peers, and `CoEditor` is the final component class bringing it all together. It provides the public API for message routing, handles local text editing changes and forwards incoming messages accordingly. The rest of this chapter describes these solutions in more detail, starting by introducing the design of messages that the component sends and receives.

[1] https://github.com/pekam/co-editor

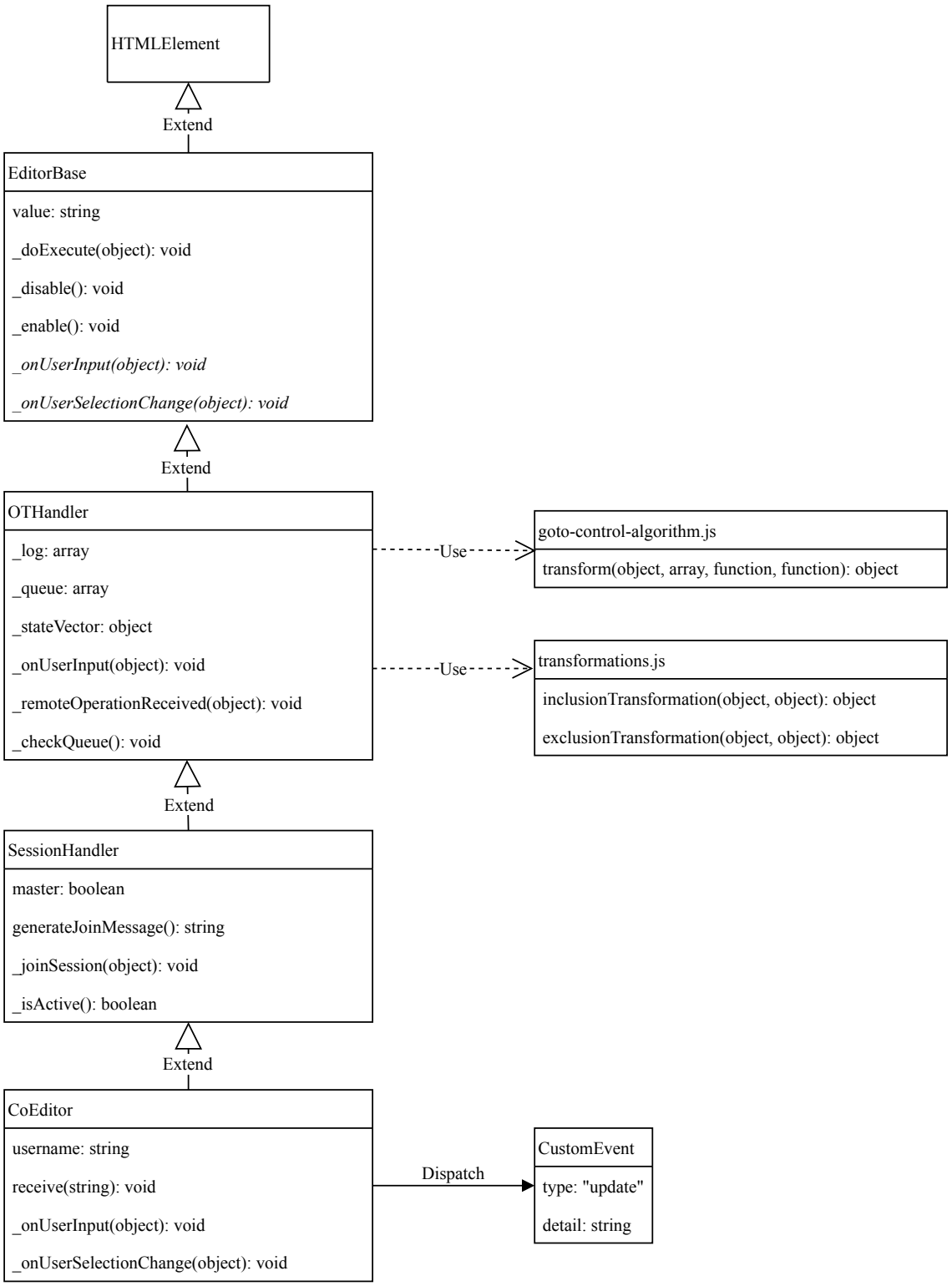

Figure 4.1: Class diagram of the component

4.4.1 Message Format

Definitions 1 and 5 might give the impression that we can transmit the text editing operations as simple strings that follow the defined formats. However, we need to attach additional information to these messages, such as data related to the sender component and its state. We also need to handle some other actions in addition to the plain text changes. For these reasons, each message is a JavaScript object, into which we can insert arbitrary amount of data as properties. These properties are also a lot more straightforward to set and read compared to parsing the information from a string. During the transmission, the message objects are serialized into string form though, as described in Section 4.3.

The system needs four types of messages. Two of these are the text editing operations insert and delete. The other types are *join*, which gives the necessary information for a new peer to join the editing session, and *caret*, which informs changes in a user's caret position or text selection. To identify different kinds of operations, all of the message objects have a string property called *type*. Table 4.2 presents the properties included in each message type. The first two property rows include the actual operation data, while the last rows contain information about the site which generated the message. We included *length* to the delete message in case we want to support string-wise operations later. The presented message structure is clarified more throughout this chapter.

type: "insert"	type: "delete"	type: "caret"	type: "join"
index: number text: string	index: number length: number	index: number length: number	id: number text: string
userId: number username: string stateVector: object	userId: number username: string stateVector: object	userId: number username: string	stateVector: object

Table 4.2: Message types and their properties

4.4.2 Text Editor Integration

The responsibility of `EditorBase` is to abstract away the text editor implementation, providing an interface for the subclasses to integrate the operations. It populates the shadow DOM, in our case with a Quill editor instance. The content of the shadow DOM is quite simple. We only need one `<div>` element. A reference to this element is given to Quill, which transforms it into a text editor without us having to construct any more DOM by ourselves. Besides the `<div>` element, we define a couple of CSS rules to make the custom element look and behave as wanted, and inject some core styles of Quill into the shadow DOM. As the shadow DOM content is so simple, we are setting it simply as a string to the `innerHTML` property, instead of utilizing a `<template>` element.

`EditorBase` is responsible for listening for user input in the text editor, and notifying about it by calling the `_onUserInput` and `_onUserSelectionChange` functions, which are expected to be implemented by the subclasses. `EditorBase` also handles converting the editor actions into a format which is used by the operational transformation algorithm. For example, when a user types the character 'a' into the beginning of the document, `EditorBase` should call `_onUserInput` with an object:
`{ type: "insert", index: 0, text: "a" }`.
The rest of the properties will be populated into the message object by the subclasses.

`EditorBase` also provides a set of protected functions to be called by the other layers of the implementation. The most important one is `_doExecute`, which executes the given insert, delete or caret operation in the internal editor. The function name implies that the operation is executed in its current form without any further transformations. The rest of the protected APIs allow enabling and disabling the editor for user input, which are used by the session management. The public `value` property allows the subclasses, as well as the component user, to set or retrieve the current text of the editor. The property getter and setter functions have been overridden to propagate to Quill's `getText` and `setText` functions. The `_setValueSilently` function is needed when a compo-

nent joins an editing session, as we do not want the initial value setting to generate insert operations.

An important benefit of this design comes from the fact that every Quill-specific piece of our code is encapsulated in this one base class. If we later want to change the implementation to use a completely different editor in the core, we only need to touch this one file, as long as we keep its interface unchanged and make sure that a usable text editor is rendered for the end-user.

4.4.3 Operational Transformation

`OTHandler` extends `EditorBase` by adding the operational transformation implementation to the web component. It contains the necessary data structures; log, state vector and operation queue, as well as the API and functionality to handle incoming remote operations. `OTHandler` also reacts to the user input by updating and attaching the state vector and pushing the operation to the queue. This differs from the OT literature, where the user input is usually captured before executing, and the local operation is then handled through the same logic as the remote ones. However, since the local operation would be executed immediately anyway, it does not make a difference if we first execute the operation and then integrate it with the OT algorithm.

The data structures of the GOTO algorithm are described as dynamic arrays in the literature, and thus we have implemented the log and the queue as JavaScript arrays which store the executed and queued operation messages. The state vector, however, was implemented as an object, with each property name (or key) corresponding to the id of the site whose logical clock is tracked in the value[2]. The reason behind this is to maintain the state vector more easily when a collaborating user disconnects. In the original GOTO approach, logical clocks are identified in the state vector by their indices in the array. If we want to remove a value from an array after a client disconnects, for example at index

[2]Basic JavaScript objects are often used as map/dictionary data structures (sets of key-value pairs).

1, all the other clients must know that the site whose logical clock used to be at vector index 2, is now at index 1, and so on. This might cause problems when the peers receive the notification about the client disconnection at different times and in different states. From an object we can remove any property without affecting how the other values are referenced. Thus, this data structure was chosen, although garbage collecting the state vector values was left out of the scope of this thesis work.

The core API for the subclasses to interact with `OTHandler` is a function called `_remoteOperationReceived`, which should be called whenever a new remote insert or delete operation has arrived. `OTHandler` implements the causality check with state vectors as described in Section 3.2.2, and based on this check either integrates the operation or adds it to the queue. Integrating a remote operation consists of running the transformations with the GOTO algorithm, asking the `EditorBase` to execute the transformed operation with `_doExecute`, pushing the executed operation to the log and updating the state vector. Each time when a new remote operation has been integrated, `OTHandler._checkQueue` function is used to find a causally ready operation from the queue. If such an operation is found, it gets integrated and `_checkQueue` is called again.

The GOTO control algorithm and the transformation functions can perfectly exist without requiring anything from our web component implementation or from each other. Thus, our solution contains two ES6 modules without any dependencies. The first independent JavaScript module is `goto-control-algorithm.js`, which exports a single function called `transform`. This function takes care of all the transformations following the GOTO algorithm, based on its four parameters; a causally ready remote operation to be executed, the log of executed operations and the application-dependent inclusion and exclusion transformation functions. According to the operational transformation theory, the algorithm should work with any kind of application (not just text editors), as long as the provided transformation functions satisfy the transformation properties (see

Definition 2) and the post-conditions defined by the GOT approach (see Definition 4). One requirement of the `transform` function is that the operations are objects which contain their state vector timestamps in a property named as `stateVector`.

The second module which is independent of the web component contains the character-wise transformation functions. It is called `transformations.js`, and it exports two functions: `inclusionTransformation` and `exclusionTransformation`. Both of the functions take two operations as parameters and return the transformed version of the first operation. Although this implementation does not explicitly depend on any other module, it expects a certain format from the operations. In addition to the data properties of insert and delete, such as index and length, it requires properties `userId` and `stateVector`. With a strongly typed language, we would package these operation interface definitions with the transformations module, or put them into a separate package which the transformations module would depend on.

4.4.4 Session Handling

The `SessionHandler` class contains the `master` property/attribute and reacts to its changes via the `attributeChangedCallback` (see Section 2.2.1). In the constructor, `SessionHandler` disables the editor from user interaction by utilizing the protected functions of `EditorBase`. The editor is enabled when either the `master` attribute is activated, or a join message is received.

`SessionHandler` implements the public `generateJoinMessage` function. To generate a unique identifier for a joining client, `SessionHandler` maintains an internal counter which is incremented each time when generating a join message. The message is populated with the generated id and the master editor's state, including the state vector and the current text content of the editor.

An incoming join message is handled by `SessionHandler._joinSession` function. It sets the editor's own identifier from the message and initializes the component's

state with the provided state vector and text. As long as the joining editor receives all the messages that have timestamps exceeding the values of the state vector, the editor succeeds in joining the session and integrating into the message stream.

In the end of `_joinSession`, the `OTHandler._checkQueue` function is called. This ensures that if the component has received messages that the master editor had not yet integrated while generating the join message, those get integrated immediately. It is also possible that the editor has received and queued messages which are already effective in the initial text provided by the master editor. The `_checkQueue` function clears these by comparing the logical clocks of the site where the operation was generated. A queued operation generated at site k, with timestamp SV_O, is already effective in the text and should be removed if $SV_O[k] \leq SV[k]$, where SV is the state vector of the joining site. Because the queue is handled like this when the join message is received, the component user can start routing messages to the joining editor as soon as possible, instead of waiting for it to join the session.

4.4.5 Component Class

`CoEditor` is the final class in the hierarchy. It contains the most crucial user-facing API of the component; the `update` event and the `receive` function. `CoEditor` implements the `_onUserInput` and `_onUserSelectionChange` functions called by `EditorBase`. It attaches the `userId` and `username` properties to an outgoing message, converts this object into a string, and fires an `update` event with this string attached to it.

The `receive` function converts an incoming message from string back to an object and propagates it to the appropriate handler based on its type. The join message is given to `SessionHandler._joinSession`, text editing operations are handled by `OTHandler._remoteOperationReceived` and caret movements are executed immediately via `EditorBase._doExecute`.

Before propagating the message, `CoEditor` checks if it was originating from the component itself based on the `userId`, and ignores it if that is the case. Application developers should not implement the routing so that the messages are transmitted back to the original editor, but simply broadcasting each message to each client might be done either accidentally[3], or intentionally to simplify the application code. With this check in place, propagating messages back to the sender does not cause issues such as rendering the user's own caret twice or bloating the operation queue with never causally ready operations.

4.4.6 Caret Rendering

The previous sections already describe some aspects of how the carets of remote users are rendered, but this logic is not fully covered. We are utilizing a third-party module called quill-cursors [44] for this purpose. Its API allows rendering a caret in the given index (or range in case of selection) with the given name and color. When inserting or deleting text before the caret, the module updates its position automatically. In the current implementation, we generate a random color for each new user who joins the session.

An important thing to note is that the caret message is not fired when the user types or deletes text. In the case of delete, we know that the caret should be rendered in the deletion index. When inserting text, the caret should be rendered after the last inserted character. With this information, `EditorBase` takes care of moving the caret also after integrating text editing operations.

The caret message is needed only when the user moves the caret or selects text, without inserting or deleting anything. As described in the earlier sections, caret messages are internally handled through the `_onUserSelectionChange` function, instead of the `_onUserInput` like insert and delete. By taking this different path of control, caret

[3]This actually happened once while experimenting with the component, which is why this check was added.

messages bypass all the operational transformation logic. This causes a known bug in the component; one user editing the text and another moving the caret concurrently may cause the caret to be rendered in a wrong position. We could have implemented more transformation functions to include the caret messages in the GOTO algorithm, but decided to leave this for future work. After all, rendering a remote caret in a wrong index is just a visual bug which gets corrected via the next update.

4.5 Test Automation

An important part of software development is automated testing, which allows us to verify that the product works as expected after each change that we make in the code. In the JavaScript ecosystem, there exists a lot of testing tools and frameworks for frontend products. Following the Open Web Component Recommendations [46], we used the Mocha framework [47] for describing our tests and Chai [48] as an assertion library.

The most complex and error-prone part of the `<co-editor>` component is the consistency maintenance. To test this functionality with two `<co-editor>` instances on the same page, we need to simulate concurrent operations and incorrect receiving order, which may happen in a distributed environment. One option to simulate concurrency is to use the `setTimeout` JavaScript function to postpone the receiving of remote operations for a specified amount of milliseconds. However, this is not an efficient nor technically reliable way of testing, because it adds the delay to each test case and expects the computation to take less than the delay to produce expected results. Another solution, which also allows us to reorder the reception of updates, is to save the received operations and provide them manually to the other editors when appropriate.

Listing 4.1 shows an example of a test case which verifies that concurrent inserts at the same position produces convergent document states which contain the effect of each operation. The test components are rendered by providing the HTML content to the

`fixture` helper function. The editors are configured to save the operations into arrays, instead of passing them to the other editor immediately. This allows us to insert text into both editors without either receiving the updates from the other. The helper function `insertText` simulates a user writing text into the specified index. After the insertions, we flush the changes by providing the cached updates to the `receive` functions. Finally, the Chai API is used to verify that the editors contain expected values. Most of this code can be reused for many test cases, and is actually written in a function which is executed before each case in a set of tests.

At the time of writing, the project's test suite consists of 27 test cases, divided into several categories and subcategories. The first tests verify the components functionality before and after setting the `master` attribute or receiving the join message. For example, the editor should be disabled until either of these actions happen. After receiving the join message, the editor should set the text content copied from the master, and execute or clear causally ready queued operations.

The rest of the test cases focus on consistency maintenance. Starting from simple cases, the first subcategory of tests verifies convergence after synchronous inserts, deletes and setting of the `value` property. The remainder of the test cases focus on concurrent operations, starting with convergence on different pairs of operations. Causality preservation tests make sure that operations are not executed while they depend on non-executed operations, and that operations received in a wrong order will eventually be executed. Finally, we ensure that the editor maintains user intentions along convergence, by testing not only that the text values are equal, but that they are exactly what we would expect after the executed operations. We have test cases for different combinations of concurrent operation types, as well as different position relations (having smaller, larger or equal index compared to the other operation) for each of those combinations.

The tests described up to this point are executed simply on two clients. In the last few tests, we ensure that a third client can join the session and the changes by each client

```
it('should converge on concurrent inserts', async () => {
  const testDom = await fixture(`
    <div>
      <co-editor id="master" master></co-editor>
      <co-editor id="client"></co-editor>
    </div>
  `);
  const master = testDom.querySelector('#master');
  const client = testDom.querySelector('#client');

  client.receive(master.generateJoinMessage());

  const masterOps = [];
  const clientOps = [];

  master.addEventListener('update', e => masterOps.push(e.detail));
  client.addEventListener('update', e => clientOps.push(e.detail));

  insertText(master, 0, 'foo');
  insertText(client, 0, 'bar');

  masterOps.forEach(op => client.receive(op));
  clientOps.forEach(op => master.receive(op));

  expect(master.value).to.equal('barfoo\n');
  expect(client.value).to.equal('barfoo\n');
});
```

Listing 4.1: Automated convergence testing

get executed by the other editors. We should improve the test coverage by verifying that consistency is maintained in complex scenarios, where multiple editors send concurrently operations to each other in different orders. Also, the remote caret rendering is currently not tested at all, which is not acceptable for such a core feature.

5 Evaluation

In order to test how the `<co-editor>` component works in real world, we implemented a couple of sample applications integrating with different technologies and architectures. Besides verifying that the component is usable when integrated in a real application, we wanted to evaluate its *developer experience (DX)*. Developer experience measures the ease-of-use and understandability of an API, similarly to how user experience (UX) testing evaluates the usability of an application from the end-user's point of view.

From the end-user's point of view, the test application used for the evaluation works as follows. There is at most a single active editing session, and any user can join it. If the session is not currently active, the first user connecting to the application will initiate the session with an empty document, and this user will be referred to as the *document owner*. Any browser connecting after that will join this editing session, allowing to collaborate with the document owner and possibly other clients. Each client, except the document owner, can disconnect and rejoin as they wish without affecting the session. If the document owner disconnects, all the other clients will be notified that the session has ended, and the connection will be closed, still allowing the client to view and edit the text locally. At this point, the client can refresh the page to become the new document owner, as there is no active session in place.

This kind of a demo application is rationalized by an assumption, that in real use cases the document is most likely owned by one user. The document might be saved on the document owner's account in the cloud, or the document owner might be filling some

kind of a form or a forum message on her account. Other users may join to help the document owner to write the text, but they are not responsible of the document and can not press the save or send button. In a proper use case, the collaborators would need to be authorized to join the editing session, but this does not directly affect the web component integration, which is why we can keep the application simple by letting anyone join.

5.1 Testing in a Centralized Architecture

5.1.1 Server Push

Routing the messages of our client-side component through a web server is as easy or hard as it is to handle any kind of bidirectional real-time communication between a server and a browser. Integrating the component into a centralized architecture means that the clients will send the update messages to the server, which forwards them to the other clients. Thus, the web server needs to be able to send data to the clients on its own initiative.

The web and the HTTP protocol were originally designed to work in such a way that the browsers request data from the server. The server can not send data on its own, without the client first initiating the transaction. As the web has evolved from static file serving into a much more feature-rich application platform, solutions for server-initiated messages, known as *server push*, have emerged. One popular strategy has been *long-polling*, which means that after each response, the client will open a new request, to which the server can respond when it wishes [49]. Finally, a technology called *WebSockets* emerged, which provides a full-duplex TCP channel between the server and the browser, eliminating the overhead of HTTP's verbose message structure [50].

5.1.2 Test Application Implementation

To test the component in a centralized architecture, we built a simple web application on Node.js [51], which is a server-side JavaScript runtime environment. We installed two

frameworks, Express.js [52] and Socket.IO [53], to easily start our project and let us focus on the application logic. Express.js is a minimal web application framework, which we are using to serve the client-side HTML and JavaScript content to the browsers. Socket.IO is a library which enables real-time bidirectional data communication between the server and the browsers with an API based on events and the concept of sockets. Under the hood the framework uses WebSockets when possible.

The relevant parts of the demo application code are listed in Appendix A, starting with the server-side implementation. The server-side Socket.IO library fires a global `connection` event when a new client makes a request for the app, and `disconnect` event when a client closes the app or the socket connection is terminated by other means. We also define some event types of our own: `update` for sending and receiving operations, and `set-master`, `request-join-message` and `master-disconnected` for session handling. The `connection` event provides a reference to the new socket connection, which is utilized in the session logic and to register listeners for other events. Figure 5.1 illustrates the event flows of connecting clients and a disconnecting document owner, which will be further described next.

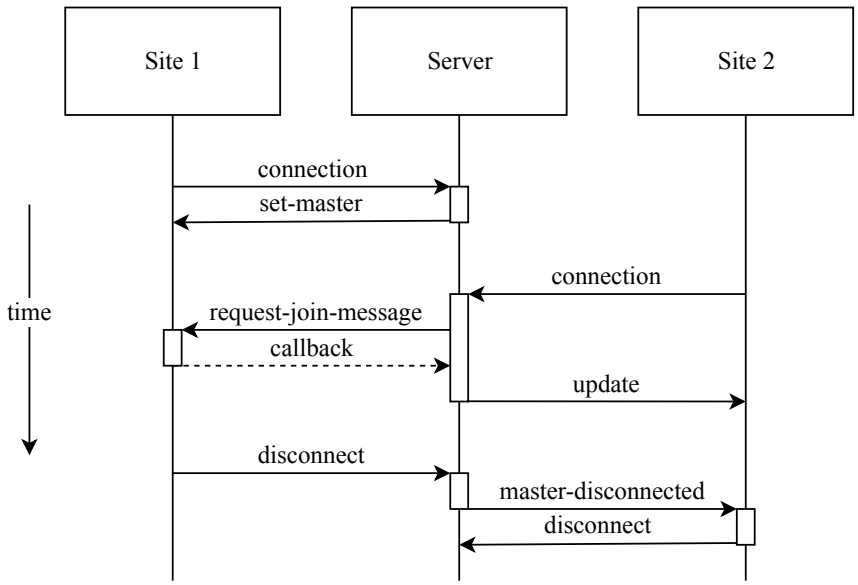

Figure 5.1: Session control in the centralized test application

A reference to the document owner's socket connection is saved in the `master` variable. When there is no active session, i.e. `master` is not defined, the connecting client is notified with a `set-master` event. Otherwise, we need to make a roundtrip to the current master editor to get a join message for the new client. This is implemented by firing a `request-join-message` event to the master socket and defining a callback which should be called by the client-side master editor with the join message as a parameter. At the server-side, the callback forwards the message to the joining client via an `update` event. When the document owner disconnects, the other clients are notified with a `master-disconnected` message. Routing the update events is straightforward with Socket.IO's broadcast API, which sends the event to every other socket except the one which fires it.

The relevant parts of the client-side HTML and JavaScript are also presented in Appendix A. Again, routing the updates with the Socket.IO API is straightforward; events from the web component are emitted into the socket, and incoming updates from the socket are passed to the component's `receive` function. The `set-master` event is handled by setting the `master` property/attribute and registering the listener for the `request-join-message` event. The callback of this event is called with a newly generated join message. The `master-disconnected` event is handled by closing the connection and showing a popup notification for the user. The page contains also an `<input>` element for changing the username which is displayed in the carets of collaborating editors.

5.1.3 Developer Experience

The `<co-editor>` component is designed to work in a P2P architecture, each client sending messages to each other. For this reason, it may feel cumbersome to handle all the communication through a central server, which is actually not required at all by the component's logic. However, with an easy-to-use bidirectional communication framework,

such as Socket.IO, broadcasting the updates to participating peers is quite effortless, as proven by our demo application.

The biggest pain point of integrating the component in a centralized architecture is the session management. Even though the component implements pretty much all of the actual session handling, it requires quite a lot of control logic to get one join message from the master editor and pass it back to the joining client through the server.

Another confusing thing that emerged while developing the test application is the `master` property/attribute. As described in the beginning of this chapter, one document should be owned by a user, and changing the ownership from one user to another is not really meaningful, at least in the use case which we have considered. With this in mind, it makes no sense that the ownership of the text is determined by a boolean value which can be turned on and off whenever the developer wishes. It also became evident that "master" is not necessarily the most obvious name for this feature.

5.2 Testing in a P2P Architecture

5.2.1 WebRTC

When a web application is used to interact with other users, the data is usually transmitted through the web server. Traditionally web browsers communicate only with the server, and not directly with each other. The problem is that routing the data through an extra node in the network adds more delay to each message. In many real-time communication applications, such as voice and video conferencing software, having as low latency as possible is really important for the user experience.

WebRTC (Web Real-Time Communications) [37] is a project which enables direct peer-to-peer communication between web browsers. Its main use case is the real-time transmission of latency-sensitive video and audio streams, usually captured from the computer's camera and microphone. WebRTC also supports transmission of arbitrary data.

To establish a P2P connection between two clients, WebRTC requires a server which coordinates the connections with control messages. This process is called *signaling*. After the server has been utilized for opening up the connection, the real application data can flow directly from browser to browser. The signaling server can also notice when a connection to a peer has closed, and let the other clients know about it. [54]

5.2.2 Test Application Implementation

To test `<co-editor>` in a P2P architecture, we implemented the test application described in the beginning of this chapter by utilizing a framework called EasyRTC [55]. It provides a more easy-to-use abstraction over the core WebRTC technology, which helped us to build the test application faster. For the purposes of evaluating the component, the used API does not matter, as long as we have an interface to send and receive messages between browsers. While describing the implementation, we will ignore everything which is not directly related to integrating the `<co-editor>` component with the P2P data streams, such as setting up the signaling server. Also, setting the username is omitted as it does not differ from the previous example (see Appendix A).

Appendix B presents the relevant parts of the demo application's client-side implementation. After the WebRTC connection has been established, we have an interface to communicate directly with the other browsers within the client-side code. With the EasyRTC API, broadcasting and receiving updates is straightforward, as can be seen in code lines 2-3 and the related helper functions. After connecting to the signaling server, the client determines who is the document owner based on who has joined the EasyRTC session first. If the client is the first one itself, the editor's `master` property is set. The `setRoomOccupantListener` function is used to establish a new data channel each time a new peer is joining the session. After the data channel has been opened between the document owner and a new peer, the document owner sends a join message to the joining editor. The signaling server notifies the clients when a peer has disconnected, and

the disconnection of the document owner is handled by notifying the user and closing the WebRTC connections.

This kind of a mesh topology, where each node sends messages directly to each other, causes one extra challenge. It is possible that a new peer skips receiving some operations while it is joining. This is illustrated in Figure 5.2. Site x sends an operation to master, which is the only peer it is connected to, at that moment. Before this operation reaches its destination, a new site joins the session and data channels are opened with the existing clients. The master editor sends the join message to the new peer, but the operation from site x is still on its way. The joining site will get an initial state which does not include the operation's effect, and site x will never send this operation to the joining site.

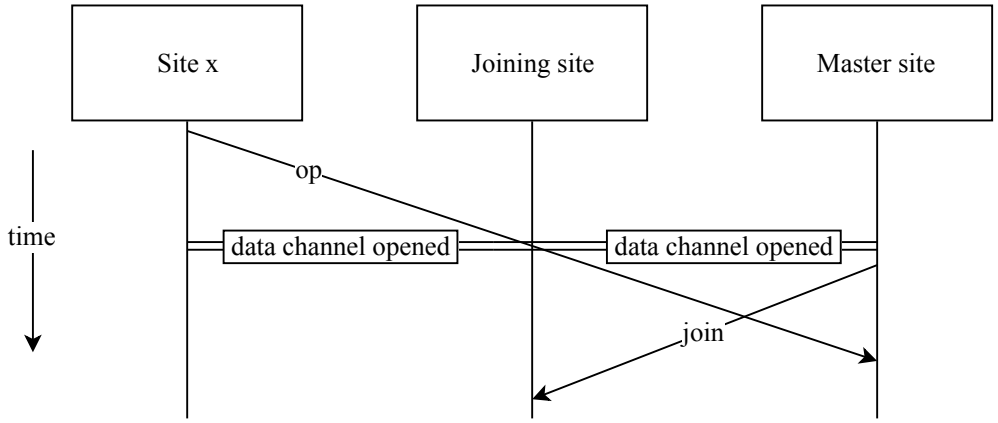

Figure 5.2: Operation skipping in P2P architecture

To tackle this issue, our implementation includes a queuing mechanism, which can be seen in Appendix B. If the signaling server has informed the client that a new peer has joined, but the data channel is not opened yet, outgoing operations for that peer are cached in an operation queue. The queued operations are sent as soon as the data channel becomes available. This solution is not bullet-proof either. It is theoretically still possible that the master opens a data channel with the new site and sends the join message before the other site has even received the notification from the signaling server that the new site exists. An exhaustive solution for this puzzle is left out of the scope of this test application.

5.2.3 Developer Experience

Compared to routing the messages through a server, it is convenient to have direct references to the data streams of each participating user in the client-side code. In particular, this improves the experience of transmitting the join message, which does not follow the normal broadcasting conventions of other messages. Transmitting this one type of message from the document owner only to the joining client is simple when the document owner has direct access to the receiver.

However, the added trouble of handling the concurrency issue, which was described in the previous section, outweighs the simplicity of routing the join messages. If the user of `<co-editor>` is required to take care of complex issues such as this one, we have not succeeded in creating a component API which is easy to plug into any kind of network topology.

5.3 Improvements Based on the Evaluation

We assume that most developers using the component would be routing the messages through a server for a few reasons. First of all, web application are already built on top of a central server, so this does not need additional architectural setup. Secondly, integrating WebRTC into the application can be really laborious process. Lastly, the latency requirements for collaborative text editing are not as hard as for e.g. video or speech communication. Thus, it is acceptable for the communication to take a little bit longer. Based on this assumption of preference in centralized operation routing, we will focus improving the developer experience of this use case.

In a centralized architecture, handling the situation when a new client joins the editing session requires some extra effort from the developer. A new join message needs to be requested from the master editor, and once this message is received by the server, it needs to be propagated to the joining client. The trouble comes from the fact that this message

needs to be specifically requested from the master, and it needs to be transmitted only to the one client who requested it.

We can make this process simpler by broadcasting the join message to each of the participants, just like the other operations. This way we only need to notify the master editor that a new join message should be fired into the operation stream. All the clients, except the new one which has not yet joined, will simply ignore this message. The added transmission cost of sending the join message also to those clients who do not need it is insignificant, considering the amount of data flow when users are writing the text.

The problem with this approach is that if two clients join at the same time, the master will generate two join messages, but the clients will not know which message belongs to which client. In our example application, the clients would receive the same message first and ignore the latter one as they already joined the session at that point. Thus, two clients would end up having the same id.

To avoid this problem, we need to be able to identify two clients before the master editor has provided them with identifiers. For this purpose, a joining client will generate a random temporary identifier for itself. We also introduce a new message type, *request-join*, which contains the temporary random identifier. When a new client joins the session, it broadcasts this message to the other peers. When the master editor receives the request-join message, it automatically generates a new join message and broadcasts it to everyone. The temporary identifier is also attached to the join message to know which client it is targeted at. The join message is then ignored by any site which has already joined, or whose temporary identifier does not match the one in the message.

It is still possible for two clients joining at the same time to generate the same random temporary identifier, but by making the identifier long enough, we can make it practically impossible. One could argue that if we consider these random identifiers safe enough, why do not we just generate a random identifier for every client, obviating the need of asking the identifier from the master editor. This would, however, greatly increase the

risk of generating colliding identifiers. Also, we want to avoid long identifiers, as they would significantly increase the message sizes via the `clientId` and `stateVector` properties.

The public `generateJoinMessage` function is no more needed, but we need to add some API for a client to send the request-join message. For this purpose, we introduce a function called `joinSession`. This function does not take any arguments. It should be simply called after connecting the message streams, if the client is joining an existing session where a master editor exists. A single call of this function takes care of getting the required information from the master editor as described above.

Another API improvement was made based on the idea that the master-state of an editor should not be a boolean property, as described in Section 5.1.3. Similarly to how we can now declare a joining client to not own the document with `joinSession`, we should declare the master editor to be the master once and for all. To align the new APIs, the `master` property was replaced with an `initSession` function. The revised API after these changes is listed in Table 5.1. The code in Appendix C presents how updating the component's API simplified the centralized test application. The source code of `<co-editor>` component itself, after the API revision, can be seen in Appendix D.

`CoEditor`
`value`: string
`username`: string
`initSession()`: void
`joinSession()`: void
`receive(string)`: void
Fires event: `{ type: "update", detail: string }`

Table 5.1: Improved component API

6 Future Work

We have published some pre-releases of the component to the npm repository[1] [56] under the BSD 3-Clause license. This allows web developers who use the npm package manager to easily install `<co-editor>` and integrate it into their projects. At the time of writing, version 1.0.0-alpha4 is the latest release, while 1.0.0-alpha2 contains the original API without the improvements introduced in Section 5.3. These alpha releases are meant only for testing and to get early feedback for our product. The component still needs some more work before we can publish a stable release.

The most crucial missing feature is garbage-collecting the operation log, which can be implemented as described in Section 3.2.4. In the component's current state, the log grows indefinitely with each operation. This will eventually use up the client machine's memory and slow down the GOTO algorithm exploring the log. Adding the garbage-collecting behavior should not affect the API in any way, so it was not needed to evaluate the component's usability. This is why this feature did not have a higher priority.

We should also handle disconnecting users by removing their carets and corresponding clocks from the state vectors. Many real-time communication frameworks, such as Socket.IO and WebRTC, used in Chapter 5, provide API for handling disconnecting clients. We could provide a function to notify other peers about the leaving user, but then the application developer would have to know the internal identifier of the dropping client. The components could also recognize disconnecting peers automatically by sending regu-

[1] https://www.npmjs.com/package/co-editor

lar heartbeat messages to each other. Once the component has not received any messages from a client for a set amount of time, this client can be considered as disconnected.

After these improvements, the component is feature complete, and we can enter the beta-phase of version 1.0.0. In later versions, we are planning to add support for rich text formatting and undoing operations. For rich text, we need to investigate if we can convert the formatting operations into inserts and deletes. This should be possible if the rich text formatting is implemented in the editor's state by adding metadata (such as HTML tags) in the middle of the text. Otherwise, we need to introduce additional operations and implement more transformation functions. Undoing the latest executed operation is straightforward, but in a collaborative editor the latest operation might be created by a collaborating user. The desired undo effect is that the change made by the latest locally generated operation is reverted, but all of the later remote operations are still effective [21]. Transformation functions need to be applied to achieve this.

We also recognize some internal performance and code maintenance improvements which could enhance the quality of our product. Currently, the consistency maintenance works with character-wise operations, as their transformation functions were easier to implement and allowed us to deliver the product and test the API faster. Implementing transformations for string-wise operations should greatly improve the component's performance in cases where a long piece of text is either pasted or removed by a single action.

We could also investigate more operational transformation techniques, as we were not able to cover all of them in the scope of one thesis, and GOTO proved to satisfy our requirements. For example, in [57] Sun and Sun claim that their COT (Context-Based Operational Transformation) algorithm allows undoing any operation in the history without requiring exclusion transformations, while having a superior time complexity for remote operation handling compared to GOTO. Dropping the exclusion transformations would simplify our code base and also significantly help us in integrating the string-wise

operations. Further research is needed to determine whether changing to COT or some other consistency maintenance technique would be worthwhile.

In reflection, we could have written our code base in TypeScript [58] to make it more robust and avoid many coding errors. For example, the messages already have types (implemented as string properties), and each message must include all the properties defined in Table 4.2. TypeScript interfaces would help us to avoid mistakes by ensuring that the messages always conform to their type definitions. Luckily, TypeScript is a superset of JavaScript, meaning that including the TypeScript compiler to our build pipeline would not break any of our existing code. This means that instead of making a huge refactoring at once, we can include the type system to our code one piece at a time, e.g. starting with the message definitions.

7 Conclusions

In this thesis we have designed and implemented a web component called `<co-editor>`, which enables web developers to integrate a real-time collaborative text editor into their applications with minimal effort. By utilizing the latest web standards, we managed to build an encapsulated UI component while avoiding to depend on any specific web frameworks. Implementing the consistency maintenance logic entirely in the client-side allowed us to minimize the requirements of the server-side implementation. In fact, `<co-editor>` can be used even without a server in a P2P network (with some caveats), as demonstrated in Section 5.2. As the consistency maintenance algorithm was completely based on literature, our main contributions in this paper are the user friendly component API and the client-side session management logic.

One of our main issues was that networking is an application-level concern, and can not be implemented in a client-side UI component. This is also one reason why many existing developer tools include a server-side counterpart (another reason being centralized consistency maintenance algorithms). Because of the wide variety of technologies used in web application backends and the varying preferences of developers, we wanted to avoid forcing the `<co-editor>` users to run a specific server-side integration of our component. Instead, we aimed make its usage so simple, that it can be integrated into any technology stack with a nominal amount of programming. Our component provides a simple interface for connecting the clients via the `update` event and the `receive` function. The application developer just needs to broadcast the messages included in the

fired events to other clients. We consider this being the minimal possible effort required to connect the message streams, considering the limitations of a pure client-side implementation. As a bonus, our queue cleaning mechanism recognizes and removes outdated operations from the operation queue. This further improves the developer experience by forgiving some common memory leaking mistakes, such as sending the same operation twice, or redirecting operations back to the sender.

In Chapter 3, we reviewed several methods for consistency maintenance, and found out that the GOTO algorithm is a suitable solution for our purposes. Although operational transformation can be quite complex, implementing GOTO was relatively straightforward based on the research papers. However, these papers assume a constant number of participants. Our `<co-editor>` component introduces an intelligent session maintenance mechanism, which enables a new client to join after connecting the message streams by simply calling the `joinSession` function. Without any specific message routing, the joining editor communicates with the document owner's editor to get started with an initial editor state. We invented this mechanism after evaluating that the original API required the application developer to implement too much control logic.

Although our product is still in its alpha stage, we have met our goals for this thesis. We have proven that `<co-editor>` can handle both the consistency maintenance and session handling in client-side, while providing a simple message passing interface for application developers. The component API is easy to use, and building a collaborative text editing application with `<co-editor>` requires only a few lines of code when using a sophisticated communication framework. The core value of our research is that our component integrates easily into any web technology stack, compared to competing products which rely on specific server-side technologies. From the academic point of view, we have created a session management system and a queue cleaning function based on state vectors, which fulfill some groupware requirements not covered by the GOTO algorithm itself.

References

[1] O. Suwantarathip and S. Wichadee, “The effects of collaborative writing activity using google docs on students’ writing abilities”, *Turkish Online Journal of Educational Technology-TOJET*, vol. 13, no. 2, pp. 148–156, 2014.

[2] W. Zhou, E. Simpson, and D. P. Domizi, “Google docs in an out-of-class collaborative writing activity”, *International Journal of Teaching and Learning in Higher Education*, vol. 24, no. 3, pp. 359–375, 2012.

[3] I. Blau and A. Caspi,
“What type of collaboration helps? psychological ownership, perceived learning and outcome quality of collaboration using google docs”,
in *Proceedings of the Chais conference on instructional technologies research*, vol. 12, 2009, pp. 48–55.

[4] *CKEditor*, `https://ckeditor.com/`, [Online; accessed 23-March-2019].

[5] *Firepad*, `https://firepad.io/`, [Online; accessed 23-March-2019].

[6] *Quill*, `https://quilljs.com/`, [Online; accessed 23-March-2019].

[7] *Quill-ShareDB-Cursors*,
`https://github.com/pedrosanta/quill-sharedb-cursors`,
[Online; accessed 23-March-2019].

[8] MDN, *Web Components*, `https://developer.mozilla.org/en-US/docs/Web/Web_Components`, [Online; accessed 24-January-2019].

[9] Web Hypertext Application Technology Working Group, *HTML Living Standard*, https://html.spec.whatwg.org/, [Online; accessed 28-January-2019].

[10] Google Developers, *Custom Element Best Practices*, https://developers.google.com/web/fundamentals/web-components/best-practices, [Online; accessed 30-January-2019].

[11] E. Bidelman, *Custom Elements v1: Reusable Web Components*, https://developers.google.com/web/fundamentals/web-components/customelements, [Online; accessed 30-January-2019].

[12] Web Hypertext Application Technology Working Group, *DOM Living Standard*, https://dom.spec.whatwg.org/, [Online; accessed 29-January-2019].

[13] E. Bidelman, *Shadow DOM v1: Self-Contained Web Components*, https://developers.google.com/web/fundamentals/web-components/shadowdom, [Online; accessed 28-January-2019].

[14] World Wide Web Consortium, *CSS Shadow Parts*, https://www.w3.org/TR/css-shadow-parts/, [Online; accessed 25-March-2019], Nov. 2018.

[15] ——, *HTML Imports*, https://www.w3.org/TR/html-imports/, [Online; accessed 28-March-2019], Feb. 2016.

[16] Polymer Team, *Web Components v0 Deprecations*, https://www.polymer-project.org/blog/2018-10-02-webcomponents-v0-deprecations, [Online; accessed 1-February-2019].

[17] J. Fagnani, *Polymer 3.0 preview: npm and ES6 Modules*, https://www.polymer-project.org/blog/2017-08-22-npm-modules, [Online; accessed 1-February-2019].

[18] D. Mosberger, "Memory consistency models", *Oper. Syst. Rev.*, vol. 27, no. 1, pp. 18–26, Jan. 1993.

[19] C. Sun, D. Sun, Agustina, and W. Cai, "Real differences between OT and CRDT for Co-Editors", Oct. 2018. arXiv: 1810.02137 [cs.DC].

[20] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems", *Acm Sigmod Record*, 1989.

[21] C. Sun, "Undo as concurrent inverse in group editors", *ACM Trans. Comput. -Hum. Interact.*, vol. 9, no. 4, pp. 309–361, Dec. 2002.

[22] G. Oster, P. Urso, P. Molli, and A. Imine, "Data consistency for P2P collaborative editing", in *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, ser. CSCW '06, Banff, Alberta, Canada: ACM, 2006, pp. 259–268.

[23] C. Sun and C. Ellis, "Operational transformation in real-time group editors: Issues, algorithms, and achievements", in *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '98, Seattle, Washington, USA: ACM, 1998, pp. 59–68.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[25] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems", *ACM Trans. Comput. -Hum. Interact.*, vol. 5, no. 1, pp. 63–108, Mar. 1998.

[26] J. Day-Richter, *What's different about the new Google Docs: Making collaboration fast*, https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html, [Online; accessed 3-January-2019], 2010.

[27] Agustina, F. Liu, S. Xia, H. Shen, and C. Sun, "CoMaya: Incorporating advanced collaboration capabilities into 3D digital media design tools", in *Proceedings of the 2008 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '08, San Diego, CA, USA: ACM, 2008, pp. 5–8.

[28] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping, "High-latency, low-bandwidth windowing in the jupiter collaboration system", in *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, ser. UIST '95, Pittsburgh, Pennsylvania, USA: ACM, 1995, pp. 111–120.

[29] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser, "An integrating, transformation-oriented approach to concurrency control and undo in group editors", in *Proceedings of the 1996 ACM conference on Computer supported cooperative work - CSCW '96*, 1996.

[30] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering", *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, pp. 56–66, 1988.

[31] C. Sun, Y. Yang, Y. Zhang, and D. Chen, "A consistency model and supporting schemes for real-time cooperative editing systems", *Australian Computer Science Communications*, vol. 18, pp. 582–591, 1996.

[32] M. Raynal and M. Singhal, "Logical time: Capturing causality in distributed systems", *Computer*, vol. 29, no. 2, pp. 49–56, Feb. 1996.

[33] C. Sun, D. Chen, and X. Jia, "Reversible inclusion and exclusion transformation for string-wise operations in cooperative editing systems", in *Proceedings of the 21st Australasian Computer Science Conference*, 1998, pp. 441–452.

[34] M. Shapiro and N. Preguiça, “Designing a commutative replicated data type”, Oct. 2007. arXiv: 0710.1784 [cs.DC].

[35] S. Weiss, P. Urso, and P. Molli, “Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks”, in *2009 29th IEEE International Conference on Distributed Computing Systems*, Jun. 2009, pp. 404–412.

[36] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia, “A commutative replicated data type for cooperative editing”, in *2009 29th IEEE International Conference on Distributed Computing Systems*, Jun. 2009, pp. 395–403.

[37] *WebRTC*, https://webrtc.org/, [Online; accessed 27-February-2019].

[38] N. Santos, *Why ContentEditable is Terrible*, https://medium.engineering/why-contenteditable-is-terrible-122d8a40e480, [Online; accessed 30-March-2019], May 2014.

[39] World Wide Web Consortium, *ContentEditable*, http://w3c.github.io/editing/contentEditable.html, [Online; accessed 30-March-2019], Mar. 2019.

[40] ——, *Input Events Level 1*, https://www.w3.org/TR/input-events-1/, [Online; accessed 30-March-2019], Nov. 2018.

[41] *TinyMCE*, https://www.tiny.cloud/, [Online; accessed 30-March-2019].

[42] *Ace*, https://ace.c9.io/, [Online; accessed 30-March-2019].

[43] *CodeMirror*, https://codemirror.net/, [Online; accessed 30-March-2019].

[44] *Quill-Cursors*, https://github.com/reedsy/quill-cursors, [Online; accessed 30-March-2019].

[45] MDN, *Creating and triggering events*, https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events, [Online; accessed 28-February-2019].

[46] *Open Web Component Recommendations*, https://open-wc.org/, [Online; accessed 30-March-2019].

[47] *Mocha*, https://mochajs.org/, [Online; accessed 30-March-2019].

[48] *Chai Assertion Library*, https://www.chaijs.com/, [Online; accessed 30-March-2019].

[49] J. Hanson, *What is HTTP Long Polling*, https://www.pubnub.com/blog/2014-12-01-http-long-polling/, [Online; accessed 21-February-2019], Dec. 2014.

[50] M. West, *An Introduction to WebSockets*, https://blog.teamtreehouse.com/an-introduction-to-websockets, [Online; accessed 21-February-2019].

[51] *Node.js*, https://nodejs.org/, [Online; accessed 31-March-2019].

[52] *Express.js*, https://expressjs.com/, [Online; accessed 27-February-2019].

[53] *Socket.IO*, https://socket.io/, [Online; accessed 27-February-2019].

[54] *WebRTC - Signaling*, https://www.tutorialspoint.com/webrtc/webrtc_signaling.htm, [Online; accessed 27-February-2019].

[55] *EasyRTC*, https://easyrtc.com/, [Online; accessed 27-February-2019].

[56] *npm*, https://www.npmjs.com/, [Online; accessed 17-March-2019].

[57] D. Sun and C. Sun, “Context-Based operational transformation in distributed collaborative editing systems”,
IEEE Trans. Parallel Distrib. Syst., vol. 20, no. 10, pp. 1454–1470, Oct. 2009.

[58] *TypeScript*, `https://www.typescriptlang.org/`,
[Online; accessed 17-March-2019].

Appendix A Centralized Test Application

This appendix presents the relevant parts of the application source code used to test `<co-editor>` in a centralized architecture, as described in Section 5.1.2.

Server-side implementation:

```
let master;

io.on('connection', socket => {
  joinSession(socket);
  socket.on('disconnect', () => clientDisconnected(socket));
  socket.on('update', message => updateReceived(socket, message));
});

function joinSession(socket) {
  if (!master) {
    socket.emit('set-master');
    master = socket;
  } else {
    master.emit('request-join-message', null, response => {
      socket.emit('update', response);
    });
  }
}

function clientDisconnected(socket) {
  if (socket === master) {
    socket.broadcast.emit('master-disconnected');
```

```
    master = undefined;
  }
}

function updateReceived(socket, message) {
  socket.broadcast.emit('update', message);
}
```

Client-side implementation:

```
<h1>CoEditor With Socket.IO Demo</h1>
<input placeholder="username">
<co-editor></co-editor>

<script>
  const socket = io();
  const editor = document.querySelector('co-editor');

  editor.addEventListener('update', event => socket.emit('update', event.detail));
  socket.on('update', message => editor.receive(message));

  socket.on('set-master', () => {
    editor.master = true;
    socket.on('request-join-message', (data, callback) => {
      callback(editor.generateJoinMessage());
    });
  });

  socket.on('master-disconnected', () => {
    socket.disconnect();
    alert('The document owner disconnected. Connection disabled.');
  });

  document.querySelector('input').addEventListener('input', e => {
    editor.username = e.srcElement.value;
  });
</script>
```

Appendix B P2P Test Application

This appendix presents the relevant parts of the application source code used to test `<co-editor>` in a P2P architecture, as described in Section 5.2.2.

```
const editor = document.querySelector('co-editor');
editor.addEventListener('update', e => broadcast(e.detail));
easyrtc.setPeerListener((peer, type, message) => editor.receive(message));

let master;
easyrtc.connect('co-editor-demo', myId => {
  master = getAllPeers()[0];
  editor.master = (master === myId);
});

let joining = true;
easyrtc.setRoomOccupantListener((roomName, occupantList) => {
  if (joining) {
    joining = false;
    return;
  }
  const peers = Object.keys(occupantList);
  peers.filter(peer => !isConnected(peer)).forEach(peer => easyrtc.call(peer));
});

easyrtc.setDataChannelOpenListener(peer => {
  if (editor.master) {
    send(peer, editor.generateJoinMessage());
  }
  flushQueue(peer);
});

easyrtc.setDataChannelCloseListener(peer => {
```

```
29    if (peer === master) {
30      easyrtc.disconnect();
31      alert('The document owner disconnected. Connection disabled.');
32    }
33  });
34
35  function broadcast(message) {
36    getAllPeers().forEach(peer => {
37      if (isConnected(peer)) {
38        send(peer, message);
39      } else {
40        addToQueue(peer, message);
41      }
42    });
43  }
44
45  function send(peer, message) {
46    easyrtc.sendDataP2P(peer, 'message', message);
47  }
48
49  const queues = {};
50  function addToQueue(peer, message) {
51    queues[peer] = queues[peer] || [];
52    queues[peer].push(message);
53  }
54
55  function flushQueue(peer) {
56    if (!queues[peer]) {
57      return;
58    }
59    queues[peer].forEach(message => send(peer, message));
60    delete queues[peer];
61  }
62
63  function getAllPeers() {
64    return easyrtc.getRoomOccupantsAsArray('default');
65  }
66
67  function isConnected(peer) {
68    return easyrtc.doesDataChannelWork(peer);
69  }
```

Appendix C Centralized Test Application With Revised Component API

This appendix presents the relevant parts of the application source code used to test `<co-editor>` in a centralized architecture, after the component API was improved as described in Section 5.3.

Server-side implementation:

```
let master;

io.on('connection', socket => {
  joinSession(socket);
  socket.on('disconnect', () => clientDisconnected(socket));
  socket.on('update', message => updateReceived(socket, message));
});

function joinSession(socket) {
  if (!master) {
    socket.emit('init-session');
    master = socket;
  } else {
    socket.emit('join-session');
  }
}

```

```
function clientDisconnected(socket) {
  if (socket === master) {
    socket.broadcast.emit('master-disconnected');
    master = undefined;
  }
}

function updateReceived(socket, message) {
  socket.broadcast.emit('update', message);
}
```

Client-side implementation:

```
<h1>CoEditor With Socket.IO Demo</h1>
<input placeholder="username">
<co-editor></co-editor>

<script>
  const socket = io();
  const editor = document.querySelector('co-editor');

  editor.addEventListener('update', event => socket.emit('update', event.detail));
  socket.on('update', message => editor.receive(message));

  socket.on('init-session', () => editor.initSession());
  socket.on('join-session', () => editor.joinSession());

  socket.on('master-disconnected', () => {
    socket.disconnect();
    alert('The document owner disconnected. Connection disabled.');
  });

  document.querySelector('input').addEventListener('input', e => {
    editor.username = e.srcElement.value;
  });
</script>
```

Appendix D Component Source Code

This appendix presents the source code of `<co-editor>` version 1.0.0-alpha4, including the API changes described in Section 5.3. Only the relevant parts are included. The full source code is available at:

`https://github.com/pekam/co-editor/tree/1.0.0-alpha4`

co-editor.js

```
import SessionHandler from './session-handler.js';

class CoEditor extends SessionHandler {

  get username() {
    return this.getAttribute('username');
  }

  set username(value) {
    if (value) {
      this.setAttribute('username', value);
    } else {
      this.removeAttribute('username');
    }
  }

  _onUserInput(message) {
    super._onUserInput(message);
    this._send(message);
  }

  _onUserSelectionChange(message) {
```

```
23      this._isActive() && this._send(message);
24    }
25
26    _send(message) {
27      message.userId = this._id;
28      message.username = this.username;
29      this.dispatchEvent(new CustomEvent(
30        'update', { detail: JSON.stringify(message) }));
31    }
32
33    receive(message) {
34      message = JSON.parse(message);
35
36      if (this._isActive() && message.userId === this._id) {
37        return;
38      }
39      switch (message.type) {
40
41        case 'request-join':
42          this._joinRequested(message);
43          break;
44        case 'join':
45          this._joinMessageReceived(message);
46          break;
47
48        case 'insert':
49        case 'delete':
50          this._remoteOperationReceived(message);
51          break;
52
53        case 'caret':
54          this._isActive() && this._doExecute(message);
55          break;
56
57        default:
58          throw new Error(`Unhandled message type ${message.type}`);
59      }
60    }
61  }
62  customElements.define('co-editor', CoEditor);
```

editor-base.js

```
import '../vendor/quill.core.js';
import '../node_modules/quill-cursors/dist/quill-cursors.min.js';
import quillStyles from '../vendor/quill-styles.js';
import { generateRandomColor } from './helpers.js';

export default class EditorBase extends HTMLElement {

  constructor() {
    super();

    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
        <style>
          :host {
            display: block;
            border: 1px solid lightgrey;
          }
        </style>
        <style>${quillStyles}</style>
        <div id="editor-container"></div>
        `;
    const container = this.shadowRoot.querySelector('#editor-container');

    Quill.register('modules/cursors', QuillCursors);
    this._quill = new Quill(container, {
      modules: {
        cursors: true,
        history: { maxStack: 0 } // Disables Quill's undo/redo
      },
      formats: []
    });
    this.__quillCursors = this._quill.getModule('cursors');
    this.__caretData = {};

    this._quill.on('selection-change', function (range, oldRange, source) {
      range && this._onUserSelectionChange({
        type: 'caret',
        index: range.index,
        length: range.length
      });
```

```
    }.bind(this));

    this._quill.on('text-change', function (delta, oldDelta, source) {
      if (source !== 'user') {
        return;
      }

      // Transforms the changes to a simpler format in a single object:
      // { retain: number, insert: string, delete: number }
      const ops = delta.ops.reduce((acc, op) => Object.assign(acc, op), {});

      // Generate character-wise operation messages
      const index = ops.retain || 0;
      ops.delete && [...Array(ops.delete)].forEach(_ => this._onUserInput({
        type: 'delete',
        index,
        length: 1
      }));
      ops.insert && [...ops.insert].forEach((c, i) => this._onUserInput({
        type: 'insert',
        index: index + i,
        text: c
      }));
    }.bind(this));
  }

  get value() {
    return this._quill.getText();
  }

  set value(value) {
    if (this._quill.isEnabled()) {
      this._quill.deleteText(0, this._quill.getLength(), 'user');
      this._quill.insertText(0, value, 'user');
    }
  }

  // This doesn't generate any operations
  _setValueSilently(value) {
    this._quill.setText(value);
  }
```

```
82
83  _disable() {
84    this._quill.disable();
85  }
86
87  _enable() {
88    this._quill.enable();
89  }
90
91  _doExecute(op) {
92    switch (op.type) {
93
94      case 'insert':
95        this._quill.insertText(op.index, op.text);
96        this.__updateCaret(op.userId, op.username, op.index + op.text.length, 0);
97        break;
98
99      case 'delete':
100       if (op.disabledBy && op.disabledBy.length) {
101         return;
102       }
103
104       this._quill.deleteText(op.index, op.length);
105       this.__updateCaret(op.userId, op.username, op.index, 0);
106       break;
107
108     case 'caret':
109       this.__updateCaret(op.userId, op.username, op.index, op.length);
110       break;
111   }
112 }
113
114 __updateCaret(id, username, index, length) {
115   const range = { index, length };
116   if (!this.__caretData[id]) {
117     this.__addCaret(id, username, range);
118   } else if (username !== this.__caretData[id].username) {
119     // Needs to be removed and re-added to update the name
120     this.__quillCursors.removeCursor(id);
121     this.__addCaret(id, username, range);
122   } else {
```

```
      this.__quillCursors.moveCursor(id, range);
    }
  }

  __addCaret(id, username, range) {
    const color = (this.__caretData[id] && this.__caretData[id].color)
      || generateRandomColor();
    this.__quillCursors.setCursor(id, range, username, color);
    this.__caretData[id] = { username, color };
  }
}
```

goto-control-algorithm.js

```
/**
 * Transforms the given operation to its execution form by using the
 * GOTO (General Operational Transformation Optimized) algorithm.
 * As a side effect modifies the log.
 *
 * @param {Object} op a causally ready operation, with a state vector timestamp
 *                    stored in its property 'stateVector'
 * @param {Array} log the log of executed operations (AKA history buffer)
 * @param {Function} it the inclusion transformation function
 * @param {Function} et the exclusion transformation function
 *
 * @return the execution form of op
 */
export default function transform(op, log, it, et) {

  let firstIndependentIndex = log.findIndex(oldOp => !isDependentOn(oldOp, op));

  if (firstIndependentIndex === -1) {
    return op;
  }

  const dependentOps = log.slice(firstIndependentIndex)
    .filter(oldOp => isDependentOn(oldOp, op));

  dependentOps.forEach(depOp => {
    const ind = log.indexOf(depOp);
    for (let i = ind; i > firstIndependentIndex; i--) {
      const transposed = transpose(log[i - 1], log[i], it, et);
```

```
      log[i - 1] = transposed[0];
      log[i] = transposed[1];
    }
    firstIndependentIndex++;
  });

  return log.slice(firstIndependentIndex).reduce(it, op);
}

function isDependentOn(op1, op2) {
  return op1.stateVector[op1.userId] <= op2.stateVector[op1.userId];
}

function transpose(op1, op2, it, et) {
  const transformed2 = et(op2, op1);
  const transformed1 = it(op1, transformed2);
  return [transformed2, transformed1];
}
```

ot-handler.js

```
import EditorBase from './editor-base.js';
import transform from './goto-control-algorithm.js';
import { inclusionTransformation, exclusionTransformation } from './transformations.js';

export default class OTHandler extends EditorBase {

  constructor() {
    super();
    this._log = [];
    this._stateVector = {};
    this._queue = [];
  }

  _onUserInput(op) {
    this._stateVector[this._id]++;
    op.stateVector = Object.assign({}, this._stateVector);
    this._log.push(op);
  }

  _remoteOperationReceived(op) {
    if (this._isActive() && this.__isCausallyReady(op)) {
```

```
22        this._integrateRemoteOperation(op);
23      } else {
24        this._queue.push(op);
25      }
26    }
27
28    _integrateRemoteOperation(op) {
29      const transformed = transform(op, this._log,
30        inclusionTransformation, exclusionTransformation);
31      this._doExecute(transformed);
32      this._log.push(transformed);
33
34      this._stateVector[op.userId] = this._stateVector[op.userId] || 0;
35      this._stateVector[op.userId]++;
36
37      this._checkQueue();
38    }
39
40    _checkQueue() {
41      // Remove operations which are already effective in the text
42      this._queue = this._queue.filter(op =>
43        op.stateVector[op.userId] > this._stateVector[op.userId]);
44
45      const causallyReadyOpIndex = this._queue
46        .findIndex(op => this.__isCausallyReady(op));
47      if (causallyReadyOpIndex > -1) {
48        const causallyReadyOp = this._queue.splice(causallyReadyOpIndex, 1)[0];
49        this._integrateRemoteOperation(causallyReadyOp);
50      }
51    }
52
53    __isCausallyReady(op) {
54      const clockAhead = Object.keys(op.stateVector)
55        .filter(id => id !== op.userId.toString())
56        .find(id => op.stateVector[id] > (this._stateVector[id] || 0));
57
58      return !clockAhead &&
59        (op.stateVector[op.userId] === (this._stateVector[op.userId] || 0) + 1);
60    }
61  }
```

session-handler.js

```
import OTHandler from './ot-handler.js';
import { generateUUID } from './helpers.js';

export default class SessionHandler extends OTHandler {

  constructor() {
    super();
    this._disable();
  }

  initSession() {
    this._master = true;
    this._nextId = 0;
    this._id = this.__generateId();
    this._stateVector[this._id] = 0;
    this._enable();
  }

  joinSession() {
    this.__tmpId = generateUUID();
    this._send({
      type: 'request-join',
      tmpId: this.__tmpId
    });
  }

  _joinRequested(op) {
    if (!this._master) {
      return;
    }
    const id = this.__generateId();
    this._stateVector[id] = 0;

    const joinMessage = {
      type: 'join',
      tmpId: op.tmpId,
      id: id,
      stateVector: Object.assign({}, this._stateVector),
      text: this.value
      // TODO: include caret positions
```

```
    };
    this._send(joinMessage);
  }

  _joinMessageReceived(message) {
    if (this._isActive() || message.tmpId !== this.__tmpId) {
      return;
    }
    this._enable();

    this._id = message.id;
    this._stateVector = message.stateVector;
    this._setValueSilently(message.text);

    this._joined = true;
    this._checkQueue();
  }

  _isActive() {
    return this._master || this._joined;
  }

  __generateId() {
    return this._nextId++;
  }
}
```

transformations.js

```
export function inclusionTransformation(op1, op2) {
  const copy = Object.assign({}, op1);
  IT[`${op1.type}_${op2.type}`](copy, op2);
  return copy;
}

export function exclusionTransformation(op1, op2) {
  const copy = Object.assign({}, op1);
  ET[`${op1.type}_${op2.type}`](copy, op2);
  return copy;
}

// Inclusion transformations
```

```
const IT = {
  insert_insert(op1, op2) {
    if (op1.index < op2.index) {
    } else if (op1.index === op2.index && op1.userId > op2.userId) {
    } else {
      op1.index++;
    }
  },

  insert_delete(op1, op2) {
    if (op1.index > op2.index) {
      op1.index--;
    }
  },

  delete_insert(op1, op2) {
    if (op1.index >= op2.index) {
      op1.index++;
    }
  },

  delete_delete(op1, op2) {
    if (op1.index > op2.index) {
      op1.index--;
    } else if (!(op2.disabledBy && op2.disabledBy.length) && op1.index === op2.index) {
      op1.disabledBy = (op1.disabledBy || []).concat(op2);
    }
  }
}

// Exclusion transformations
const ET = {
  insert_insert(op1, op2) {
    if (op1.index > op2.index) {
      op1.index--;
    }
  },

  insert_delete(op1, op2) {
    if (op1.index > op2.index) {
      op1.index++;
```

```
    }
  },

  delete_insert(op1, op2) {
    if (op1.index >= op2.index) {
      op1.index--;
    }
  },

  delete_delete(op1, op2) {
    if (op1.index > op2.index) {
      op1.index++;
    }
    if (op1.disabledBy)
      op1.disabledBy = op1.disabledBy.filter(op => !opEquals(op, op2));
  },
}

function opEquals(op1, op2) {
  return op1.userId === op2.userId &&
    op1.stateVector[op1.userId] === op2.stateVector[op2.userId];
}
```