CRDTs in the Wild

Jakob Meyer-Hilberg
jakob.meyer-hilberg@uni-ulm.de

ABSTRACT

Conflict-free Replicated Data Type (CRDT) ist eine Dateistruktur, welche vor allem in Verteilten Systemen Verwendung findet. Ihre Eigenschaft, dass zwei Dokumente ohne manuelle Konfliktlösung zusammen gemerged werden können, ist für gleichzeitiges Bearbeiten vor allem für kollaborative Web-Anwendungen interessant. Anwendungen, die auf CRDT-Algorithmen aufbauen, müssen einige Einschränkungen bezüglich möglicher Operationen sowie Skalierbarkeit hinnehmen.

Diese Arbeit vergleicht verschiedene Algorithmen und quelloffene Bibliotheken, welche auf github gefunden wurden. Es wird dargestellt, welcher CRDT zu welchem Anwendungsfall passt.

Um die Herausforderung, die ein CRDT-Algorithmus meistert, darzustellen, wird eine Fullstack-Web-Anwendung geschrieben, welche Konflikte durch simulierte Netzwerkverzögerungen erzeugt und die Resolution des Algorithmus damit besser nachvollziehbar macht.

Schlüsselwörter: Verteiltes Computersystem, distributed, strong eventual consistency, async Full-Stack Web App, State-based WOOT, G-Counter, SequenzCRDT

1 EINLEITUNG

Die Motivation ist die Verbesserung von kollaborative Anwendungen, dabei kann man die Netzwerklast, CPU-Last oder die möglichen Funktionen der Anwendung optimieren. Eine Funktion könnte sein das Löschen, Hinzufügen aber auch kompliziertere Operationen, wie das Verschieben von Textpassagen. Ein Beispiel von kollaborativen Anwendungen sind Google Docs, Etherpad, Zoom, Echtzeitstrategiespiele und Messenger im weitesten Sinne.

Kollaborative Anwendungen sind Programme, die von mehreren Personen gleichzeitig bedient werden. Sobald eine Anwendung auf mehr als einem Rechner läuft, müssen Zustände synchronisiert werden. Das bedeutet, dass der Zustand auf allen Maschinen abgeglichen werden muss. Bei den meisten Anwendungsfällen wird dabei einfach ein Server als die Quelle der Wahrheit herangezogen und Zustände werden von diesem verteilt. Falls dieser Server allerdings unerreichbar oder nur mit hoher Latenz erreichbar ist, wird es schwer bis unmöglich, das Programm zu bedienen: Das ist ein Problem.

ATVS, WS 2022/23, Institute of Distributed Systems, Ulm University

Um dieses Problem zu lösen, benötigt man ein System, das ohne einen zentralen Server funktioniert oder bei welchem der Server jederzeit gewechselt werden kann.

Um diesen Server zu wechseln, benötigt man ein kompliziertes Protokoll, welches aus einem Netz von Teilnehmern einen Rechner als den zentralen Rechner wählt, der Entscheidungen treffen kann. Bei einem Ausfall oder nach einer gewissen Zeit wird dann eine neue zentrale Instanz auf Zeit gewählt. Das Problem dabei ist, dass jenes kompliziertes Protokoll in den Umschaltphasen blockiert.

Deswegen wäre es gut, wenn das Synchronisieren ohne eine zentrale Instanz funktionieren würde.

CRDTs funktionieren ohne zentralen Server, da sie die folgenden drei Eigenschaften besitzen:

(1) CRDTS garantieren, dass jeder Teilnehmer zu jeder Zeit das Dokument verändern kann.
(2) Es gibt einen Algorithmus, der Dokumente von verschiedenen Teilnehmern mergen (zusammenführen) kann.
(3) Die Dokumentänderungen aller Teilnehmer konvergieren zu einem einheitlichen Dokument.

In neuerer Zeit haben einige Anwendungen genau aus diesem Grund CRDTs verwendet, um die gewünschten Eigenschaften zu nutzen.

2 RELATED WORK

Die im folgenden erwähnten Open-Source-Projekte werden in der Arbeit benutzt:

2.1 Automerge [1]

Auto-merge ist eine in Javascript geschriebene Library, welche Dateistrukturen und Algorithmen liefert, um kollaborative Anwendungen zu schreiben. Um einen leichten Einstieg in CRDT-Algorithmen zu bekommen, ist der Funktionsumfang von Automerge zu groß. In der Automerge-Beschreibung heißt es unter anderem: "Automerge is a pure data structure library, that does not care about what kind of network you use", was bedeutet, dass man sich selbst um sein Netzwerk kümmern muss.

2.2 Yjs [2]

Die in Javascript geschriebene Yjs-Bibliothek bietet dem Nutzer einen verteilten Datentyp, welcher aus Map und Array besteht. Der Nutzer muss sich hierbei nicht um die Netzwerk-Implementationen kümmern. Er kann zwischen verschiedenen Netzwerk-Providern wählen. Naheliegende Provider sind hierbei y-webrtc und y-websocket. Auch gibt einen Provider, welcher auf das Messenger-Protokoll Matrix setzt. Darüber hinaus gibt es noch andere Provider.

2.3 xi-rope-engine [3]

Hierbei handelt es sich um eine CRDT-Engine, geschrieben in der Programmiersprache Rust, ausgelegt für spezielle SequenceCRDTs. Intern wird ein Operation-based Ansatz verwendet. In 3.1.3 wird Operation-based erklärt. Für das Löschen von Elementen bzw. Buchstaben in der Sequenz werden Tombstones erzeugt. Tombstones verhindern, dass gelöschte Elemente wieder hinzugefügt werden, wobei jedoch das Dokument niemals kleiner wird. Viele Änderungen könnten dabei das Dokument sehr groß werden lassen, obwohl es sich nur um ein kleines Dokument handelt. Eine sehr gute Beschreibung des Innenlebens der Engine ist in dem Artikel "CRDT - The Xi Text Engine" [4] zu lesen. Die Engine legt besonderen Fokus auf Performance. Um den Code zu kompilieren, benötigt man den Rust-Compiler. Man benötigt dann noch einen Schritt, den Rust-Code im Webbrowser ausführbar zu machen. Rust kann zu WebAssembly kompilieren, womit es möglich wäre, die xi-rope-engine im Web zu nutzen. Auch hier wird kein Netzwerk gestellt. Die asynchrone Natur der Engine macht es komplizierter, einfachste Funktionen wie Zeilenumbrüche zu implementieren, weswegen der xi-Texteditor niemals den Funktionsumfang eines herkömmlichen Texteditors wie zum Beispiel vscode erlangen wird. Eine Retrospektive, warum der xi-Editor in seinen Zielen gescheitert ist, lässt sich hier nachlesen [5]. Interessant für diese Arbeit ist insbesondere die xi-rope-engine, welche im Herzen des xi-Editors ist.

3 MATERIALS AND METHODS

Zum Vergleichen der verwendeten Bibliotheken verwende ich das am einfachsten zu implementierende CRDT und anhand von überschaubaren Codesnippet werden Unterschiede aufgezeigt werden.

3.1 Theorie

Es gibt genug Paper, die erklären, wie CRDTs funktionieren, wie zum Beispiel in der Quelle [6]. Allerdings werden in meinen Quellen Begriffe und deren Synonyme nicht einheitlich verwendet. Deswegen versucht das Theorie-Kapitel für diese Arbeit die Begrifflichkeiten an die vorhandenen Code-Beispiele anzupassen. Die Quellen können unter Umständen andere Begriffe verwenden oder die Dateiformate isomorph wählen.

3.1.1 eventual consistency & strong eventual consistency (SEC) & strong consistency. In Verteilen Systemen werden Zustände auf mehrere physikalische Rechner verteilt. Bei Änderung einer Variablen muss diese Änderung zu allen Rechnern propagiert werden. Dabei könnten Zustände inkonsistent werden. Der einfachste Ansatz, diese zu vermeiden, ist *strong consistency*, was man mit Locking implementiert kann. Dabei kann immer nur ein Rechner eine Variable schreiben. Das gesamte System wird in dieser Zeit blockiert. Der Vorteil von *strong consistency* ist, dass sich das gesamte System wie ein einzelner Rechner verhält. Damit ist es einfach, mit so einem System eine Anwendung zu schreiben. Dieses System zu implementieren ist unkompliziert indem einem zentralen Server die Aufgabe gegeben wird, immer nur einer Person Schreibzugriffe zu geben.

Eventual consistency erlaubt es mehreren Parteien gleichzeitig zu schreiben und garantiert, dass das verteilte Dokument gegen einen einheitlichen Dokument konvergiert, wobei es zwischenzeitlich zu invaliden Werten kommen kann. Invalide Werte sind problematisch, weshalb mehr gefordert werden muss.

Alle CRDTs sind *strong eventual consistency (SEC)*, was bedeutet, dass jederzeit gelesen und geschrieben werden kann und es immer einen validen Wert gibt. Es muss nicht gefürchtet werden, dass ein nicht-gelöster Konflikt Probleme bereitet. Warum alle CRDTs gleich SED sind, wird auch in Paper [7] beschrieben.

3.1.2 CRDTs versus OTs. Operation Transformation kurz OTs sind Algorithmen, welche auch eine *strong eventual consistency (SEC)* garantieren. Dabei setzen OTs jedoch darauf, dass es einen zentralen Server gibt, welcher die Nachrichten in eine chronologische Reihenfolge bringt. Anders ist es bei CRDTs, welche im Gegensatz zu OTs keinen zentralen Server benötigen. Indem man einen OT-Algorithmus verwendet, ergeben sich auch Vorteile: So kann auf Tombstones verzichtet werden und man ist in der Wahl der verfügbaren Operationen nicht eingeschränkt.

Nach dem Paper "Formal design and verification of operational transformation algorithms for copies convergence" [8] gibt es einige OT-Algorithmen, die nicht konvergieren, obwohl es von den Autoren behauptet wird. Das Mergen von CRDTs immer kommutativ und von OTs in der Regel nicht. Damit sind OTs komplexer und skalieren nicht.

3.1.3 Operation-based und State-based. Es gibt zwei Ansätze, die Eigenschaften von CRDTs zu verwenden: Das eine sind Operation-based CRDTs, welche den Status nur durch eine Update-Operation übermitteln. Das andere sind State-based CRDTs, welche den ganzen Status verschicken und zu einem neuen mergen. In dieser Arbeit werden nur State-based CRDTs verwendet, da sie beim Versenden einem vollständigen Dokument entsprechen. Im Paper "Conflict-free Replicated Data Types" [7] wird beschrieben, dass Operation-based und State-based äquivalent sind. Das bedeutet, falls wir einen State-based Algorithmus implementiert haben, können wir einen äquivalenten Operation-based Algorithmus formulieren, und anderes herum.

3.1.4 G-Counter. Der G-Counter (Grow-only Counter) zählt nur nach oben.

In Abbildung 1 sind Funktionen des G-Counters aufgelistet. Die auch in den Programmierbeispielen verwendeten Begriffe sind *increment*, *value* und *merge* und die Funktion myID() ist analog der "actorid" in Automerge und der "session_id" in xi_rope. Im folgenden wird *id* verwenden. Die *id* ist universell eindeutig. Es kann nur der lokale G-Counter

verändern werden, welcher durch die *id* gegeben ist. Jeder Teilnehmer hat eine eigne *id*. Dabei entspricht *increment* dem Erhöhen eines lokalen G-Counters. Die payload besteht aus einem Zähler je *id*. Somit kann der Gesamtwert des G-Counters mit der Summe aller Counter errechnet werden: Dies wird *value* genannt. Die *merge*-Operation nimmt zwei G-Counter und iteriert über *n* Counter der payload von X und Y. Der größere der Werte von X oder Y ist dabei der neue Wert, kurz das Maximum von beiden. Hier wären *n* Teilnehmer möglich, mathematisch könnte man beim Hinzufügen eines Teilnehmers, *n* um eins vergrößern. Bei der Implementation wird später nicht ein Array, sondern ein Objekt gewählt: Dies begrenzt einen nicht auf *n* Teilnehmer und es ist zusätzlich möglich, direkt mit der *id* auf den lokalen Counter zuzugreifen.

Abbildung 1: Mathematische Definition G-Counter, Quelle: "A comprehensive study of Convergent and Commutative Replicated Data Types" [9]

1: payload integer$[n]$ P
2: initial $[0, 0, \ldots, 0]$
3: update $increment$ $()$
4: let $g = myID()$
5: $P[g] := P[g] + 1$
6: query $value$ $()$: integer v
7: let $v = \sum_i P[i]$
8: compare (X, Y) : boolean b
9: let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
10: merge (X, Y) : payload Z
11: let $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i], Y.P[i])$

3.1.5 SequenceCRDT. Dieser Datentyp ist von der Struktur eine Liste. Diese Liste hat eine Ordnung und man kann verschiedene Operationen ausführen, um sie zu ändern, wie Hinzufügen und Löschen. Andere Operationen sind möglich, wie in dem von Martin Kleppmann geschriebenen Paper "Moving Elements in List CRDTs " [10], in dem er eine explizite move-Funktion vorschlägt. Da Listen zum Implementieren von Anwendungen sehr nützlich sind, werden bessere Algorithmen und interne Datenstrukturen gesucht. Wenn man die Liste modifiziert, macht man das durch eine Operation, die dann für immer in der Dateistruktur gespeichert wird. Auch wenn ein Element gelöscht wird, muss es weiter gespeichert werden, da spätere Operationen darauf verweisen könnten. Dabei entstehen sogenannte *Tombstones*.

3.1.6 Vorhandene SequenzCRDT-Algorithmen. Nun werden einige vorhandene SequenzCRDT-Algorithmen vorgestellt. Gute Implementationen für SequenzCRDTs sind sehr gefragt und dies nicht nur für kollaborative Texteditoren. Welche Implementation geeignet ist, muss abgewogen werden. Die folgenden Algorithmen wurden in mehreren Programmiersprachen implementiert. Im folgenden wird erwähnt, welche Stärken und Schwächen die Algorithmen haben.

Logoot [11]: Funktioniert, indem jedem Element eine Position und eine *(Logoot-)id* gegeben wird. Die Position kann dabei eine Zahl zwischen 0 und 1 sein. Die *(Logoot-)id* wird als Tie-Breaker verwendet. Ein Element kann hinzugefügt werden, indem eine Zahl zwischen zwei Positionen gewählt wird. Im Worst-Case kann diese Positions-Zahl sehr viele Nachkommastellen bekommen und damit steigt der Speicherbedarf im Dokument. Außerdem kann es kann zu Interleaving-Effekten kommen: Dies passiert, wenn sich zwei Wörter ineinander verzahnen, was bei einem Texteditor weniger intuitiv nachzuvollziehen ist.

LSEQ [12]: Die Performance des Algorithmus' ist beim Einfügen von an zufälligen Positionen und am Ende des Dokuments sehr gut. Wird hingegen am Anfang des Dokuments etwas hinzugefügt,steigt der Aufwand wie bei Logoot.

(Compressing) UniquePositions[13]: Dieser Algorithmus hat die gleichen Probleme wie Logoot, jedoch wird ein immer größer werdender Identifier komprimiert abgespeichert, was für eine reale Hardware optimiert werden kann.

WOOT [14]: *Without Operational Transformation* benutzt den Operation-based Ansatz. Das Hinzufügen von Element e mit $ins(a < e < b)$ und das Löschen dieses Elements mit $del(e)$ sind hierbei die beiden Operationen.

Treedoc [15]: Dieser Algorithmus verwendet eine Baumstruktur. Dieser Baum muss *rebalanced* werden. Außerdem ist das Hinzufügen an einer bestimmten Stelle im Baum erst durch Traversieren möglich.

3.1.7 Auswahl einiger üblicher CRDTs. Es folgt eine Auswahl einiger üblicher CRDTs und exemplarischer Anwendungsfälle. Durch Kombination lassen sich weitere CRDTs erstellen.

G-Counter: (Grow-only Counter) Zähler, der nur hochzählt.

SequenceCRDT: Datentyp, der in kollaborativen Texteditoren verwendet wird (geordnete Liste).

PN-Counter: (Positive-Negative Counter) Besteht aus zwei G-Countern, mit denen hoch- und runtergezählt werden kann. Exemplarischer Anwendungsfall: Zählung im Parkhaus mit mehreren Zählern.

G-Set: (Grow-only Set) Beispielsweise die Besucherliste einer Website.

2p-Set: (Two-Phase Set) gültige und ungültige öffentliche Schlüssel (public keys)

LWW-Element-Set: (Last-Write-Wins Element-Set) Zeitlich letzte Änderung überschreibt alle vorhergehenden Änderungen.

OR-Set: (Observed-Remove Set) Like-Button, der wieder zurückgenommen werden kann.

und weitere CRDTs . . .

4 PROGRAMMIERBEISPIELE

4.1 G-Counter Programmierbeispiel

Die hier geführten Programmierbeispiele sind mit Javascript programmiert. Javascript ist die dominante Programmiersprache für Web-Anwendungen. Für den Code auf der Server-Seite verwende ich auch Javascript. Der Server hat nur die Aufgabe, Nachrichten weiterzuleiten. Auf dem Server werden keine Daten gespeichert. Er ist nur dazu da, damit die einzelnen Teilnehmer bzw. Clients miteinander kommunizieren können. Ein CRDT-Algorithmus wird nur auf dem Client ausgeführt: Damit funktionieren die Code-Beispiele auch auf einem komplizierteren Netzwerk.

Anders als bei einem mathematischen Ausdruck muss bei der Implementierung eines nützlichen CRDT-Algorithmus eine Datenstruktur gewählt werden. In Listing 1 wurde darauf geachtet, dass sich der G-Counter mit den Javascript internen Funktionen JSON.stringify() und JSON.parse() serialisieren und de-serialisieren lässt.

Als Referenz-Implementation kann CRDT.GCounter [16] genommen werden.

Listing 1: Das ist eine Implementation eines GCounters in Javascript.

```
export class GCounter {
    constructor({id,map ={}}) {
        this.id = id;
        this.map = map;
        this.map[id] = map[id]|| parseInt(0);
    }
    merge(document) {
        for(var iter_id in document.map){
            if(this.map[iter_id] == undefined ||
               document.map[iter_id] > this.map[iter_id]){
                this.map[iter_id] = document.map[iter_id];
            }
        }
    }
    increment() {
        this.map[this.id] = this.map[this.id] + 1;
    }
    value() {
        let sum = 0;
        for(var iter_id in this.map){
                sum +=this.map[iter_id];
        }
        return sum;
    }
    localvalue() {
        return this.map[this.id];
    }
    serialize(){
        return JSON.stringify(this)
    }
    static deserialize(input){
        return new GCounter(JSON.parse(input))
    }
}
```

Listing 2: Eine Demonstration des GCounters. Hierzu werden zwei Dokumente erstellt. Das zweite Dokument wird ins erste konfliktfrei gemergt. Hier wird das folgende nicht demonstriert: In einer Anwendung dürfte nur der Besitzer der *id* die Funktion increment aufrufen. Änderungen werden empfangen, deserialisiert und sofort gemerged.

```
import {GCounter} from "./gcounter.js";
let doc1 = new GCounter({id:"alice"});
let doc2 = new GCounter({id:"bob"});
doc1.increment();
doc1.increment();
doc2.increment();
console.log(doc1.value()); // output: 2
console.log(doc2.value()); // output: 1
doc1.merge(doc2)
console.log(doc1.value()); // output: 3
```

4.2 Broadcast-Server

Um die Nützlichkeit und reale Performance von CRDTs zu demonstrieren, habe ich einen Broadcast-Server geschrieben. Möglich wären auch andere Netzwerktopologien wie P2P oder WebRTC. An den Broadcast-Server gesendete Nachrichten werden an alle Teilnehmer weitergeleitet. Die Nachrichten werden vom Server weder gefiltert noch modifiziert. Um es einfach zu halten, vertrauen wir dem Server in dieser Arbeit. Wenn man dem Server nicht vertraut, müsste eine End-zu-End-Verschlüsselung verwendet werden, was auch theoretisch möglich ist.[1]

Listing 3: Die Web-Seite besteht nur aus index.html und index.js

```
import { GCounter } from "./gcounter.js"

let globalcounter =
↪  document.getElementById("globalcounter");
let localcounter =
↪  document.getElementById("localcounter");
let uuid = crypto.randomUUID();

let localdocument = new GCounter({ id: uuid });
let globaldocument = new GCounter({ id: 0 });

let url = "ws://" + location.host + "/websocket/gcounter";
let socket = new WebSocket(url);

socket.onmessage = function (event) {
    let g2 = GCounter.deserialize(event.data);
    localdocument.merge(g2)
    globaldocument.merge(g2)
    globalcounter.innerText = globaldocument.value();
    localcounter.innerText = localdocument.value();
};
let button = document.getElementById("incrementbutton");
button.addEventListener('click', function (e) {
```

[1] Man könnte hier auch von einer SFU-Architektur sprechen. Anders als bei einer MCU-Architektur, in welcher der Server beliebig komplexe Berechnungen anstellt, wird bei einer SFU-Architektur nur weitergeleitet. Für eine SFU-Architektur wird auf Server-Seite weniger Rechenleistung benötigt und man kann wie im WebRTC-Protokoll auch einen direkten Weg zwischen den Teilnehmern aufbauen. Erwähnenswert sind hier Jitsi(SFU) versus Zoom(MCU).

```
22      localdocument.increment();
23      localcounter.innerText = localdocument.value();
24      let message = localdocument.serialize()
25      let delay =
    ↪   parseInt(document.getElementById('delay').value)
26
27      //Simulierte Netzwerkverzögerung
28      setTimeout(() => {
29          socket.send(message);
30      }, delay)
31  });
```

Abbildung 2 zeigt drei Webbrowser mit der index.html, welche im Grunde aus einem script-tag besteht, dessen Inhalt in Listing 3 beschrieben wurde. Es wurde mehrmals der Increment-Button gedrückt. Der *localcounter* übernimmt den Wert sofort. Mit eine Verzögerung folgt dann der *globalcounter*.

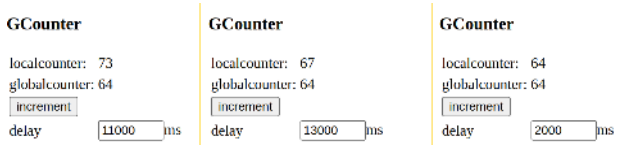

Abbildung 2: Der CRDT-Algorithmus ist erst für mehrere Teilnehmer interessant. Hier wurden einfach drei Browser-Fenster geöffnet. Die drei Teilnehmer senden dabei alle mit einer Verzögerung(delay). Der *globalcounter* ist dabei bei allen gleich. Der *globalcounter* ist immer kleiner als der *localcounter*.

4.3 Vergleichen von zum Programmieren verwendbaren Bibliotheken

In Abbildung 3 werden drei Bibliotheken aus der Sicht eines Programmierers verglichen.

Bei allen drei gibt es ein Objekt, welches das Dokument repräsentiert. Diesem Dokument wird eine *id* übergeben, welche den Teilnehmer eindeutig im ganzen Netzwerk identifiziert. Falls man bei Automerge keine *id* übergibt, wird eine neue *uuid4* automatisch bei jedem Laden des Browser-Fensters gesetzt. Die Bibliothek Yjs verwaltet eine Zahl, die mit 32-bit die Eindeutigkeit garantiert. Für xi-rope muss die *id* manuell gesetzt werden.

Automerge und Yjs sind für Web-Anwendungen geschrieben worden: Das Versenden über ein Netzwerk ist hierbei ein Schwerpunkt.

Die xi-rope-engine ist speziell für den Xi-Texteditor geschrieben worden. Diese Engine gibt auf dem lokalen Rechner jedem Prozess ein nicht-blockierendes Dokument. Dafür werden Teile des Dokuments im Arbeitsspeicher von den Prozessen geteilt. Das funktioniert, da die Dateistruktur im großen Teil unveränderlich(immutable) ist.

Automerge (javascript)

```
1  const doc1 = Automerge.init("1234")
2  doc1 = Automerge.change(doc1,  doc => {
3      doc.cards[0].done = true
4  })
5  doc3 = Automerge.merge(doc2, doc1)
```

Yjs (javascript)

```
1  const ydoc = new Y.Doc()
2  ydoc.getArray('my-array').insert(0, ['val'])
```

xi-rope-engine (rust)

```
1  let mut doc = Engine::empty();
2  doc.set_session_id((1234 as u64, 0));
3  doc.merge(&doc2);
```

Abbildung 3: Vergleich von drei Bibliotheken

4.4 Programmierbeispiel SequenzCRDT

Ein für Anwendungen nützlicher Algorithmus ist SequenzCRDT. Für einen kollaborativen Texteditor werden im Produktiveinsatz einige Optimierungen eingesetzt: So werden, wenn möglich, mehrere Buchstaben zusammengefasst. Hierbei kommt auch der Algorithmus von Nagle [17] zum Einsatz, der das Absenden der Änderungen verzögert, um Änderungen blockweise verschicken zu können. Auch wird ein 2p-Set mit einem Element verwendet, um gelöschte Buchstaben zu markieren. Dabei können Buchstaben nur einmal gelöscht werden. Löschen zwei Teilnehmer gleichzeitig denselben Buchstaben, wird dies wie ein einziger Löschvorgang gewertet. Die xi-rope-engine verwendet eine *rope*-Datenstruktur [18]. Da dies etwas mehr Codezeilen erfordert, wurde in Listing 4 als Basisdatenstruktur eine Liste verwendet. Diese Liste wurde *sequence* genannt. Auch die Automerge-Bibliothek verwendet intern eine Liste.

4.4.1 *Implementation eines SequenzCRDTs in Javascript.*

Die interessantesten Zeilen in Listing 4 beschreiben die merge-Methode. Es existieren zwei Listen a und b, von welchen immer das nächste Element genommen wird. Diese zwei Elemente werden miteinander verglichen. Das Paar von *id* und *counter* macht jeden Buchstaben eindeutig identifizierbar: Damit lassen sich die Buchstaben vergleichen und in eine eindeutige Sequenz bringen. Somit ist sichergestellt, dass egal in welcher Reihenfolge eine Anzahl von Dokumenten gemergt werden, es immer zum gleichen Ergebnis kommt. Die neue Liste wird hier *newsequence* genannt. Das Paper "Modular Verification of Op-Based CRDTs in Separation Logic" [19] stellt Kriterien auf, welche erfüllt sein müssen, damit der Algorithmus wirklich ein CRDT ist. Der in Listing 4 implementierte SequenzCRDT lässt sich als eine State-based Implementation des Woot-SequenzCRDT-Algorithmus sehen. Um es zu einer vollständigen Woot-Implementation zu machen, müsste man diese um eine Löschfunktion erweitern. Die entsprechende Stelle in der Merge-Methode ist mit einem Kommentar in Zeile 33 markiert. Weiter müsste die Dateistruktur um ein Gelöscht-Flag erweitert werden.

Listing 4: SequenzCRDT plain javascript

```
const inserthelper = (arr, index, newItem) => [
  ...arr.slice(0, index),
  newItem,
  ...arr.slice(index)
]
const notcontainhelper = (array, element) => {
    return !array.some(
        x => x .id == element.id
        && x.counter == element.counter );
}
export class SequenceCRDT {
    constructor(id){
        this.id = id;
        this.sequence = []
        this.counter = 0;
    }
    insert_into(position,character){
        this.sequence = inserthelper(this.sequence,
            position,{
                character,id:this.id,counter:this.counter
            });
            this.counter++;
    }
    merge(other){
        let a = this.sequence;
        let b = other.sequence;
        let newsequence = [];
        let ai =0;
        let bi =0;
        while(ai<a.length && bi < b.length){
            if(a[ai].id == b[bi].id ){
                if(a[ai].counter == b[bi].counter){
                    // CRDT Kombination hier möglich
                    newsequence.push(a[ai])
                    ai++;
                    bi++;
                }else{
                    if(notcontainhelper(a,b[bi])){
                        newsequence.push(b[bi++]);
                    }
                    if(notcontainhelper(b,a[ai])){
                        newsequence.push(a[ai++]);
                    }
                }
            }else if(a[ai].id < b[bi].id ){
                if(notcontainhelper(b,a[ai])){
                    newsequence.push(a[ai++]);
                }else{
                    newsequence.push(b[bi++]);
                }
            }else{
                if(notcontainhelper(a,b[bi])){
                    newsequence.push(b[bi++]);
                }else{
                    newsequence.push(a[ai++]);
                }
            }
        }
        while(ai<a.length){
            newsequence.push(a[ai++])
        }
        while(bi < b.length){
            newsequence.push(b[bi++])
        }
        this.sequence = newsequence;
    }
    value(){
        return this.sequence.map(x => x.character)
                            .join("");
    }
    serialize(){
        return JSON.stringify(this)
    }
    static deserialize(input){
           let {id, sequence, counter} = JSON.parse(input);
            let tmp = new SequenceCRDT(id)
            tmp.sequence = sequence;
            tmp.counter = counter;
            return tmp;
    }
}
```

Zuerst wird in Listing 5 der eigentliche Anwendungsfall demonstriert. Dazu werden zwei Strings "hello" und "world" gemerged. Da in diesem Fall beide an dieselbe Stelle hinzufügen werden wollen, muss der Algorithmus bestimmen, welches Wort Vorrang hat: Dabei greift die *id* als der letzte Entscheidungswert. Mögliche Ausgaben sind deswegen "helloworld" oder "worldhello". Da in diesem Fall die *id* mit 1 und 2 gewählt wurde, ist die Ausgabe "helloworld".

Listing 5: SequenzCRDT plain javascript Hello World Demo

```
import {SequenceCRDT} from "./sequencecrdt.js"
let eins = new SequenceCRDT(1);
let zwei = new SequenceCRDT(2);
eins.insert_into(0, "hello")
zwei.insert_into(0, "world")
console.log(eins.value()) //output: "hello"
console.log(zwei.value()) //output: "world"
eins.merge(zwei)
console.log(eins.value()) // output: "helloworld"
```

Im folgenden Listing 6 wird gezeigt, dass man mit einem SequenzCRDT auch einen G-Counter implementieren kann. Um zu inkrementieren, wird dem Dokument ein String "1" hinzugefügt. Dabei entsteht ein String, der aus vielen Einsen besteht ("1111111111111111111"). Der Wert des G-Counter ist dann gleich der Länge des Strings. Es muss gelten und es gilt, dass bei der Konflikt-Lösung keine Einträge verloren gehen dürfen.

Listing 6: GCounter intern in Verwendung eines SequenzCRDT geschrieben in plain javascript

```
import {SequenceCRDT} from "./sequencecrdt.js"
export class GCounter {
    constructor({id}) {
        this.sequencecrdt = new SequenceCRDT(id);
    }
    merge(document) {
        this.sequencecrdt.merge(document.sequencecrdt)
    }
    increment() {
        this.sequencecrdt.insert_into(0,"1")
    }
    value() {
        return this.sequencecrdt.value().length
    }
    serialize(){
```

```
16          return JSON.stringify(this)
17      }
18      static deserialize(input){
19          let temp = new GCounter(0)
20          temp.sequencecrdt = JSON.parse(input);
21          return temp2;
22      }
23  }
```

5 RESULTS

Es werden jetzt die zuvor beschriebenen G-Counter verglichen:

In Tabelle 1 sehen wir, dass der G-Counter, welcher mit plain javascript und mit automerge geschrieben wurde, mit der Teilnehmerzahl steigt, da jedem Teilnehmer ein neuer Platz im Dokument zugeteilt wird und dieser immer wieder überschrieben wird, analog der Merge-Zeit. Falls man sich dazu entscheidet, die xi-rope-engine als G-Counter zu verwenden, steigt der Speicher mit dem Zähler an: Da bei jedem Inkrementieren das Dokument um ein weiteres Einser-Zeichen zunimmt. Gleiches gilt für die Javascript-Version. Die Merge-Zeit für die xi-rope-engine ist nach Quelle [18] im Worst-Case $O(c)$. Der SequenzCRDT-Algorithmus aus Listing 4 hat in seiner naiven Implementation eine Komplexität des Merge von $O(c^2)$: Dies begründet sich durch die While-Schleife und die darin enthaltene Such-Funktion notcontainhelper(), welche beide von c abhängen.

Tabelle 1: Hier wird die Komplexität in Abhängigkeit von Teilnehmer t und Zähler c zusammengefasst. Diese Tabelle wurde basierend auf theoretischen Überlegungen erstellt.

	Speicher	Merge
gcounter plain javascript	$O(t)$	$O(t)$
gcounter automerge	$O(t)$	$O(t)$
xi-rope-e. sequenz als gcounter	$O(c)$	$O(c)$
javascript sequenz als gcounter	$O(c)$	$O(c^2)$

6 DISCUSSION

In Listing 6 wurde der G-Counter mit Hilfe eines SequenzCRDT implementiert. Intuitiv würde ein Entwickler, wenn er nur *einen* Zähler braucht, den G-Counter verwenden. Vorstellen kann man sich hier beispielsweise einen Besucherzähler auf einer Website. Falls man zusätzlich auch die Namen der Besucher speichern möchte, wäre der G-Counter nicht mehr die geeignete Wahl. Indem man Namen in einem SequenceCRDT speichert, könnte man damit zwei Fliegen mit einer Klappe schlagen, d.h. die oben gezeigte Implementation scheint für die Praxis geeignet zu sein.

Allgemein kann man das folgende behaupten: Ob ein CRDT-Algorithmus geeignet ist, hängt von der Anwendung ab.

7 CONCLUSION AND OUTLOOK

In Kapitel 5 wurde die Speichereffizienz betrachtet. Meistens ist jedoch der Speicher kein limitierender Faktor, da Speicherplatz heutzutage sehr billig ist. Bei Verwendung eines Delta-Algorithmus' müsste dabei noch nicht einmal immer das ganze Dokument verschickt werden. Dabei würde sich der Operation-based Ansatz anbieten, welcher zusätzlich zu anderen Delta-Algorithmen auch Rechenzeit einsparen würde. Es muss allerdings beim initialen Verbinden das gesamte Dokument übertragen werden: Deswegen kann es dennoch wichtig sein, bei einem System die Netzwerkbelastung zu betrachten.

Weiter ist auch wichtig, wie viel Rechenzeit benötigt wird, um Änderungen zu mergen.

Jeder Teilnehmer hat eine *id*, mit der er angibt, dass er es ist, der eine Änderungen am Dokument veranlasst hat. Dies funktioniert solange sich der Teilnehmer in einem vertrauenswürdigen Netzwerk befindet. Um zu vermeiden, dass ein Teilnehmer sich als ein anderer ausgibt, müsste man mit Hilfe eines Public-Key-Verfahrens die einzelnen Änderungen/Nachrichten signieren.

LITERATUR

[1] 2022. Welcome to Automerge. (2022). Retrieved 2022-12-11 from https://automerge.org/docs/hello/

[2] 2022. Yjs is a CRDT implementation. (2022). Retrieved 2022-12-11 from https://github.com/yjs/yjs

[3] 2022. Yjs is a CRDT implementation. (2022). Retrieved 2022-12-11 from https://github.com/xi-editor/xi-editor/blob/master/rust/rope/src/engine.rs

[4] 2022. CRDT - The Xi Text Engine. (2022). Retrieved 2022-10-21 from https://xi-editor.io/docs/crdt-details.html

[5] Raph Levien. 2020. CRDTs and the Quest for Distributed Consistency. (2020). Retrieved 2022-10-26 from https://raphlinus.github.io/xi/2020/06/27/xi-retrospective.html

[6] 2018. Verteilte Daten ohne Mühe: Conflict-Free Replicated Data Types. (2018). Retrieved 2022-10-21 from https://m.heise.de/developer/artikel/Verteilte-Daten-ohne-Muehe-Conflict-Free-Replicated-Data-Types-3944421.html?seite=all

[7] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free Replicated Data Types. In *13th International Conference on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*. Springer LNCS volume 6976, 386–400. DOI: http://dx.doi.org/10.1007/978-3-642-24550-3_29

[8] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. 2006. Formal design and verification of operational transformation algorithms for copies convergence. *Theoretical Computer Science* 351, 2 (2006), 167–183. DOI:http://dx.doi.org/https://doi.org/10.1016/j.tcs.2005.09.066 Algebraic Methodology and Software Technology.

[9] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[10] Martin Kleppmann. 2020. Moving Elements in List CRDTs. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2020)*. ACM, Article 4. DOI:http://dx.doi.org/10.1145/3380787.3393677

[11] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems - ICDCS 2009 (2009 29th IEEE International Conference on Distributed Computing Systems)*. IEEE, Montreal, Canada,

404–412. DOI: http://dx.doi.org/10.1109/ICDCS.2009.75

[12] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. 2013. LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering (DocEng '13)*. Association for Computing Machinery, New York, NY, USA, 37–46. DOI: http://dx.doi.org/10.1145/2494266.2494278

[13] 2018. Selected Sequence CRDTs. (2018). Retrieved 2023-2-13 from https://blog.r6l7.com/selected-sequence-crdts/

[14] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data Consistency for P2P Collaborative Editing. In *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work (CSCW '06)*. Association for Computing Machinery, New York, NY, USA, 259–268. DOI: http://dx.doi.org/10.1145/1180875.1180916

[15] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*. 395–403. DOI: http://dx.doi.org/10.1109/ICDCS.2009.20

[16] 2022. CRDT.GCounter. (2022). Retrieved 2022-10-21 from https://hexdocs.pm/crdt/CRDT.GCounter.html

[17] 2022. Nagle's algorithm. (2022). Retrieved 2022-12-13 from https://en.wikipedia.org/wiki/Nagle27s_algorithm

[18] 2022. Rope (data structure). (2022). Retrieved 2022-12-13 from https://en.wikipedia.org/wiki/Rope_(data_structure)

[19] Abel Nieto, Léon Gondelman, Alban Reynaud, Amin Timany, and Lars Birkedal. 2022. Modular Verification of Op-Based CRDTs in Separation Logic. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2, Article 188 (Oct. 2022), 1788–1816 pages. DOI: http://dx.doi.org/10.1145/3563351