Freie Universität Berlin

Masterarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Analyse und Bewertung einer Erweiterung des Tools Saros für den Editor Visual Studio Code und Validierung konzeptioneller Optimierungen

Marvin Schendel

Matrikelnummer: 5280251

marvin.schendel@fu-berlin.de

Betreuer: Victor Brekenfeld

Eingereicht bei: Prof. Dr. Lutz Prechelt

Zweitgutachter: Prof. Dr. Heiko Schwarz

Berlin, 15. Juni 2021

Zusammenfassung

In dieser Arbeit wird der Prototyp einer VSCode-Erweiterung für das Tool Saros analysiert. Dabei wird nach einer Einführung in Saros in einem ersten Schritt die grundlegende Architektur des Prototyps untersucht, um eventuelle Probleme und Schwachstellen aufzuzeigen, zu bewerten und anschließend zu verbessern. Darauf aufbauend sollen weitere, noch nicht implementierte Funktionen der Anwendung Saros umgesetzt werden. Aufgrund der Ergebnisse der Analyse wurde festgestellt, dass das Konzept des aktuellen Prototyps in einigen Bereichen nicht wie geplant verwendet werden kann. Aus diesem Grund werden nach einer Betrachtung des Problems im Anschluss zwei verschiedene Lösungskonzepte erarbeitet. Nach einer weitergehenden Analyse dieser beiden Konzepte konnten weitere, bisher unbekannte Problemfälle identifiziert werden. Mit der Erfahrung aus der Entwicklung der ersten Konzepte, kann dabei nach einer weiteren Recherche zur Konsistenzerhaltung in verteilten Systemen abschließend ein finales Konzept, inklusive der Betrachtung verschiedener Abläufe und Beispiele vorgestellt werden.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Marvin Schendel, 15. Juni 2021

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Zielsetzung 1
1.2 Aufbau der Arbeit 2

2 Saros **2**

3 Vorstellung des bestehenden Prototyps **4**
3.1 Ziel des Prototyps 4
3.2 Architektur 5
3.3 Bekannte Probleme 6

4 Analyse des Prototyps **7**

5 Theoretische Hintergründe **8**
5.1 Operational Transformation 8
5.2 Jupiter 14

6 Das VSCode-Asynchronitäts Problem **19**

7 Aufbau eines Konzeptes für eine alternative Saros-Architektur **22**
7.1 Konzept 1: Das direkte Entfernen von Änderungen 23
7.2 Konzept 2: Erweiterung des SLS 29

8 Recherchen zu neuen Konzepten **36**
8.1 Verteilte Datenbanksysteme 37
8.2 CRDTs 38
8.3 Vergleich der verschiedenen Konzepte 41

9 Entwicklung eines neuen Konzeptes **42**
9.1 Allgemeine Definitionen 43
9.2 Ablauf einer von dem/der VSCode Nutzer*in initiierten Änderung im Saros-VSCode Klienten 46
9.3 Ablauf einer vom Jupiter-Server initiierten Änderung des Saros-VSCode Klienten 50
9.4 Betrachtung des Problembeispiels 57

10 Fazit **61**
10.1 Kritische Betrachtung 62
10.2 Ausblick 63

Literaturverzeichnis **63**

A Anhang **67**
A.1 Implementierung des ersten Konzeptes 67
A.2 Finales Aktivitätsdiagramm (Änderung eines/einer VSCode Nutzer*in) - Original 68

A.3 Finales Aktivitätsdiagramm (Änderung eines/einer VSCode Nutzer*in) über zwei Seiten 69
A.4 Finales Aktivitätsdiagramm (Änderung vom Jupiter-Server) - Original . 71
A.5 Finales Aktivitätsdiagramm (Änderung vom Jupiter-Server) über zwei Seiten 72

1 Einleitung

In der Geschichte der Informatik hat sich mit neuen technologischen Möglichkeiten auch die Art und Weise verändert, wie in Projekten gearbeitet wird und Code und Programme entwickelt werden können. Ein aus dieser Entwicklung resultierender Arbeitsansatz ist die Paar-Programmierung. Sie ist dabei ein zentrales Konzept des eXtreme Programming [1, Vgl. S. 2] mit dem Ziel, die Qualität der zu entwickelnden Software zu verbessern, Wissen innerhalb eines Teams breiter zu verteilen und Fehler früher zu erkennen. Für gewöhnlich wird in dem Konzept dabei mit zwei Rollen gearbeitet: dem Fahrer und dem Navigator. Der Fahrer kümmert sich um das explizite Schreiben des Codes, während der Navigator den geschriebenen Code auf Sinnhaftigkeit und Fehler überprüft und über die nächsten Schritte oder Alternativen nachdenkt [2, Vgl. S. 79]. In den Anfängen saßen dabei meistens zwei Menschen physisch vor einem Computer. Dies ist auch heutzutage zwar noch üblich, wurde aber erweitert. Um den Ansprüchen und Anforderungen der vernetzten Welt gerecht zu werden, wird Paar-Programmierung auch verteilt eingesetzt. Ziel dabei ist es, durch den Einsatz verschiedener Technologien, wie zum Beispiel Programmen zum Teilen eines Bildschirms, Paar-Programmierung an einem Projekt auch über mehrere physische Rechner zu ermöglichen. Mittlerweile gibt es für viele integrierte Entwicklungsumgebung (im Folgenden IDE genannt) Erweiterungen, mit denen sich Paar-Programmierung direkt in der Anwendung umsetzen lässt. Eine dieser Erweiterungen ist das open-source Projekt Saros, welches in der Arbeitsgruppe Software Engineering an der FU-Berlin entwickelt wurde. Das Ziel von Saros ist es dabei, Paar-Programmierung über verschiedenen IDEs zu ermöglichen, wodurch Softwareentwickler*innen nicht auf die Verwendung derselben Entwicklungsumgebung angewiesen sind. In seinem aktuellen Entwicklungsstand umfasst Saros eine Implementierung für die IDE Eclipse, eine Alphaversion für die Umgebung IntelliJ und einen ersten Prototyp für den Texteditor Visual Studio Code (im Folgenden VSCode genannt). Des Weiteren haben die Nutzer*innen von Saros die Möglichkeit, die Kommunikation zwischen den Klienten über eigene Server laufen zu lassen, was in anderen Erweiterungen meistens nicht möglich ist. Dadurch ist unter anderem ein höherer Datenschutz gewährleistet.

Der für diese Arbeit vorranging wichtige Texteditor ist Visual Studio Code. VSCode wurde von der Firma Microsoft erstmals auf der BUILD Konferenz 2015[1] als kostenloses, plattformübergreifendes Entwicklungswerkzeug vorgestellt. Der Editor sollte dabei erstmals nicht nur auf Windows, sondern „über Microsoft Azure auch auf Linux, iOS und Android“ [4, Zitat] lauffähig sein. Visual Studio Code beschränkt sich dabei, anders als Visual Studio, auf die Kernthemen Editor, Debugging und Quellcodeverwaltung, was zu einer schnellen Verbreitung des Editors geführt hat. Heutzutage ist VSCode einer der beliebtesten Editoren[2] für Softwareentwicklung weltweit.

1.1 Zielsetzung

Ziel dieser Arbeit ist es, den Prototyp einer Saros-Implementierung für den Editor VSCode aus der Bachelorarbeit „Implementierung des Saros-Plugins für Visual Stu-

[1] siehe [3]
[2] siehe [5]

dio Code“ von Michael Schäfer[3] zu analysieren. Dabei soll in einem ersten Schritt die grundlegende Architektur untersucht, eventuelle Probleme und Schwachstellen aufgezeigt, bewertet und anschließend verbessert werden. Zusätzlich sollen weitere, noch nicht implementierte Funktionen der Anwendung Saros umgesetzt werden. Abschließend sollen im Prototyp, je nach Ergebnis der Analysephase, alle grundlegenden Funktionen implementiert sein, um einen eventuelles erstes Release zu ermöglichen.

1.2 Aufbau der Arbeit

Die Arbeit beginnt daher mit einer grundlegenden Einführung in die Funktionalitäten und den Aufbau der Anwendung Saros. Darauf aufbauend wird der aktuelle Stand des Prototyps evaluiert und auf in der Bachelorarbeit beschriebene Probleme eingegangen. Es folgt eine grobe Analyse der Architektur sowie erste Funktionstests, um den Stand der Implementierung zu überprüfen.

Zu diesem Zeitpunkt der Analyse ändert sich, aufgrund der Ergebnisse des vorherigen Kapitels, der Fokus der Arbeit von einer allgemeinen, globalen Analyse des Konzeptes, zu einer möglichen Problemfindung für evaluierte Probleme. Hierzu folgen inhaltliche Grundlagen zu operational transformation (im folgenden OT genannt) und Jupiter, bevor das eigentliche Problem in einem nächsten Kapitel erläutert wird. Anschließend an die Problembeschreibung folgt die Vorstellung verschiedener Konzepte sowie eine Übersicht zu anderen möglicherweise einsetzbaren Technologien aus der Industrie, wie Algorithmen der verteilten Datenbanksysteme oder Conflict-free Replicated Data Types (im Folgenden CRDTs genannt). Diese Ausarbeitung wird darauf aufbauend mit einer Entscheidungsfindung für eine mögliche Technologie und einem finalen Konzept fortgeführt und mit einem Fazit und einem Ausblick final abgeschlossen.

2 Saros

In diesem Kapitel wird einleitend die Architektur von Saros erläutert. Wie bereits angedeutet, ist Saros eine dezentral organisierte Erweiterung für Paar-Programmierung, welche plattformübergreifend über mehrere IDEs unterstützt wird. Es wird dabei beschrieben als eine Anwendung für verschiedenste Anwendungsfälle, wie gemeinsamen Codeüberprüfungen, das Anlernen von Beginnern, verteilter Paarprogrammierung oder für verteilte Partyprogrammierung [7, Vgl.]). Hierbei wird eine Reihe von Funktionen unterstützt, welche im weiteren Verlauf dargestellt sind. Als Basis für die Übersicht wird die Implementierung für die IDE Eclipse verwendet, da diese im aktuellen Stand den größten Funktionsumfang besitzt.

Eine detaillierte Übersicht wurde bereits in der Bachelorarbeit von Michael Schäfer erstellt und ist dort in der vollständigen Fassung zu finden. Die Funktionen sind dabei zur besseren Übersicht in Gruppen zusammengefasst (siehe Tabelle 1):

[3] siehe [6]

G1: Konfiguration	Alle Funktionen zur allgemeinen Konfiguration der Anwendung Saros
G2: XMPP - Kontaktverwaltung	Alle Funktionen zur Verwaltung von Konten und Kontakten wie das Hinzufügen oder das Wechseln von Konten
G3: XMPP – Sitzungsverwaltung	Alle Funktionen zum Erstellen und Verwalten von Sitzungen
G4: XMPP – Allgemein	Alle sonstigen Funktionen mit XMPP wie das Teilen eines Projektes
G5: Saros	Alle allgemeinen Funktionen von Saros wie das Anwenden von Textänderungen, Markieren von Änderungen anderer Teilnehmer oder das Folgen von anderen Teilnehmern

Tabelle 1: Übersicht der Funktionen von Saros [6, Vgl. S. 9-12]

Wichtig ist dabei, einen Überblick über die zu Grunde liegende Architektur der Erweiterung zu erhalten. Grundsätzlich besteht eine Saros-Session hierbei aus einem Jupiter-Server (siehe Kapitel 5.2: Jupiter) und mehreren Saros-Klienten. Der Jupiter-Server übernimmt dabei die Kommunikation von Änderungen zwischen den Klienten, verarbeitet die Änderungen, löst eventuelle Konflikte auf und hält immer eine aktuelle Version des Projekt-Dateisystems vor. Die Klienten besitzen ebenfalls eine lokale Kopie des Dateisystems und verarbeiten die vom Jupiter-Server kommunizierten oder lokal durchgeführten Änderungen. Sie können dabei auch nur über den Jupiter-Server und nicht untereinander kommunizieren, wodurch gewährleistet ist, dass der Server immer den zuletzt validen und korrekten Datenstand besitzt (siehe Abbildung 1: Übersicht der Saros-Architektur).

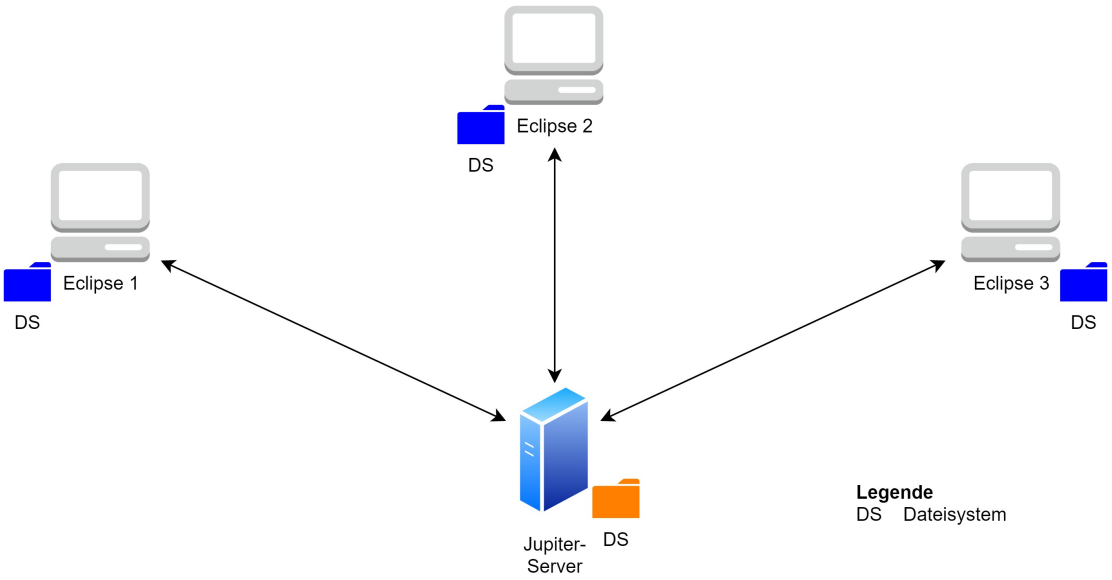

Abbildung 1: Übersicht der Saros-Architektur

Um in Saros gemeinsam an einem Projekt arbeiten zu können, muss einer der Klienten eine Session starten. Dieser Klient ist auch anschließend der Host der Saros-Session, wodurch von seiner Erweiterung neben dem Saros-Klienten auch der Jupiter-Server bereitgestellt wird. Alle Teilnehmer (aktuell werden bis zu 5 insgesamt unterstützt [7, Vgl.]), inklusive der IDE des Hosts selbst, verbinden sich mit dem Jupiter-Server und treten der Session bei. Die Konfliktlösung, also die Behandlung von gleichzeitigen Änderungen an derselben Stelle im Code, durch unterschiedliche Saros-Klienten oder durch Latenzen bei der Verbindung, werden vom Jupiter-Algorithmus aufgelöst. Der Algorithmus ist dabei sowohl im Jupiter-Server als auch in den Klienten implementiert und wird in einem späteren Kapitel näher betrachtet. Diese Funktionsweise zieht sich dabei durch die Implementierungen für alle IDEs und Editoren und ist im Allgemeinen auch in dem Prototyp für VSCode erkennbar. Da zwischen den bisherigen Implementierungen und der Implementierung für VSCode trotzdem einige große Unterschiede unter anderem in der Architektur bestehen, wird im nächsten Kapitel der in der Bachelorarbeit entwickelte Prototyp näher betrachtet.

3 Vorstellung des bestehenden Prototyps

Im folgenden Kapitel wird der Stand der Implementierung von Michael Schäfer erläutert. Dabei wird sowohl auf seine Zielsetzung als auch die Erreichung dieser in Bezug auf den letzte Stand der Implementierung eingegangen. Der Fokus liegt dabei auf den grundlegenden Aspekten der Implementierung. Detailliertere Informationen sind der Bachelorarbeit[4] zu entnehmen.

3.1 Ziel des Prototyps

Aufgrund der bereits in Kapitel 2 aufgezeigten Komplexität und des Funktionsumfangs von Saros, war bereits zu Beginn der Ausarbeitung des Prototyps klar, dass im Rahmen der Bachelorarbeit nur die grundlegendsten Funktionen implementiert werden können. Ziel der Arbeit war es daher, nur einen prototypischen Ansatz zu implementieren, um eine grundsätzliche Umsetzbarkeit von Saros in einer „Java fremden Entwicklungsumgebung wie VSC“ [6, Zitat S. 9] zu beweisen. Zusätzlich sollte die Ausarbeitung nur „der Grundstein für eine Weiterentwicklung“ [6, Zitat S. 9] darstellen. Für den Beweis sollten dabei in einem ersten Schritt vom Autor der Bachelorarbeit definierte „Grundfunktionen“ [6, Zitat S. 9] von Saros implementiert werden, „damit es möglich ist, eine Sitzung zwischen Saros-Nutzern aufzubauen“ [6, Zitat S. 9]. Im zweiten Schritt sollte anschließend die „Kernfunktionalität umgesetzt werden: das Teilen von Quelltext und das Annotieren von Änderungen“ [6, Zitat S. 9]. Ein weiteres Ziel des Prototyps war Universalität. Damit ist gemeint, dass die Saros-VSCode Erweiterung mit so wenig wie nötig Aufwand auch in anderen, vergleichbaren Editoren implementierbar sein soll. Welche Besonderheiten von VSCode dieses Ziel überhaupt möglich machen, wird dabei im nächsten Kapitel detaillierter betrachtet.

[4] siehe [6, S. 23-31]

3.2 Architektur

Die wichtigsten Überlegungen zum Design des Prototyps in der Bachelorarbeit, beziehen sich auf den Ort der Implementierung einzelner Funktionalitäten. In einer vorherigen Analyse in der Ausarbeitung von Michael Schäfer hatte sich dabei gezeigt, dass eine vollständige Implementierung von Saros, inklusive der Nutzer*innenoberfläche und der Saros-Logik in der Saros Erweiterung für VSCode (in dieser Arbeit Saros-VSCode Klient genannt), einige Probleme mit sich bringt, da VSCode Erweiterungen in Typescript geschrieben werden müssen. Hierfür wird in der Arbeit beschrieben, dass der Java Saros-Core entweder mit veralteten NPM-Paketen in das von VSCode vorgegebene Typescript eingebettet oder komplett neu in Typescript hätte geschrieben werden müssen. Aus diesem Grund wurde in der dort durchgeführten Analyse empfohlen, den Saros-Core in einen eigenständigen Java-Prozess auszulagern, um Probleme mit Typescript zu umgehen [6, Vgl. S. 20]. Diese Saros-Logik in einem eigenständigen Prozess, wird dabei als Saros Language Server (im Folgenden SLS genannt) bezeichnet.

Ein weiterer Grund für das Design des Saros-Prototyps ist die Architektur und Arbeitsweise von Visual Studio Code selbst. VSCode basiert auf dem Electron Framework und verwendet daher einen Chromium Webbrowser und NodeJs[5] als Basistechnologien [6, Vgl. S. 12]. Dabei ist der Editor auf Modularität und Erweiterbarkeit ausgelegt, jedoch mit dem Ziel, dass keine Erweiterung die Performance oder Stabilität des Editors beeinflussen kann [8, Vgl.]. Dies hat zur Folge, dass jede Erweiterung für den Editor grundsätzlich keinen direkten Zugriff auf VSCode besitzt und in einem abgekapselten Prozess arbeitet, welcher auch *Extension Host Prozess*[6] genannt wird. Um die Entwicklung von Erweiterungen trotzdem zu ermöglichen, bietet VSCode die *Extension API*[7] an, welche über bestimmte Funktionen eine Kommunikation mit festgelegten Bereichen von VSCode ermöglicht.

Neben vielen Events, für die sich eine VSCode Erweiterung zum Abrufen von Daten registrieren kann, bietet die *Extension API* auch Funktionen zum Registrieren von Kommandos, Anzeigen von Meldungen, Änderungen am Erscheinungsbild (zum Beispiel Änderung von Farben usw.) und Funktionen zur Einbindung deklarativer und programmatischer Spracheigenschaften, wie die Implementierung von Syntaxhervorhebung, Autovervollständigung oder automatische Einrückungen [6, Vgl. S. 14]. Die zuletzt genannten Funktionen sind dabei für die Implementierung von Sprachservern gedacht, um beispielsweise allen Entwickler*innen von Programmiersprachen eine gute Unterstützung Ihrer Programmiersprache im Editor zu ermöglichen.

Damit dabei nicht alle Entwickler*innen wie bisher für jede IDE oder jeden Editor einen neuen Sprachserver implementieren müssen, hat Microsoft das Language Server Protokoll (im Folgenden LSP genannt) entworfen. Dieses Protokoll soll es einem „Sprachanbieter ermöglichen, einen einzelnen Sprachserver zu erstellen, der von verschiedenen IDEs verwendet werden kann“ [11, Zitat]. Die VSCode *Extension API* bietet dabei alle Funktionen, welche zur Integration eines Sprachservers und zur Nutzung des LSPs benötigt werden [12, Vgl.]. Hierzu zählen Funktionen wie die eingangs

[5] siehe `https://nodejs.org/en/`
[6] siehe [9]
[7] siehe [10]

erwähnte Syntaxhervorhebung, Autovervollständigung, Autoeinrückung oder die automatische Anordnung von Klammern.

Da das LSP fast alle Saros-Funktionen nativ unterstützt, unter anderem alle Funktionen zum Anzeigen und Umsetzen von Textänderungen und der Saros-Core durch die Auslagerung in einen eigenständigen Prozess als SLS (zusätzlich zu dem *Extension Host Prozess*) sehr viele Ähnlichkeiten mit einem Sprachserver aufweist, wurde sich in der Bachelorarbeit für die Verwendung dieses Protokolls als Übertragungsprotokoll zwischen dem Saros-VSCode Klienten und dem SLS entschieden (siehe Abbildung 2).

Zusätzlich wurde das LSP als Möglichkeit gesehen, dass auch Saros in Zukunft einfacher an andere IDEs angebunden werden kann, um somit den zukünftigen Entwicklungsaufwand zu reduzieren. Da LSP einige Saros spezifische Funktionen, wie das Hinzufügen von Accounts oder Verwalten von Sitzungen, nicht standardmäßig unterstützt, wurde es zusätzlich um einige Funktionen erweitert [6, Vgl. S. 29]. Die Details für die Erweiterung von LSP sind jedoch für diese Ausarbeitung nicht relevant und werden daher nicht eingehender betrachtet.

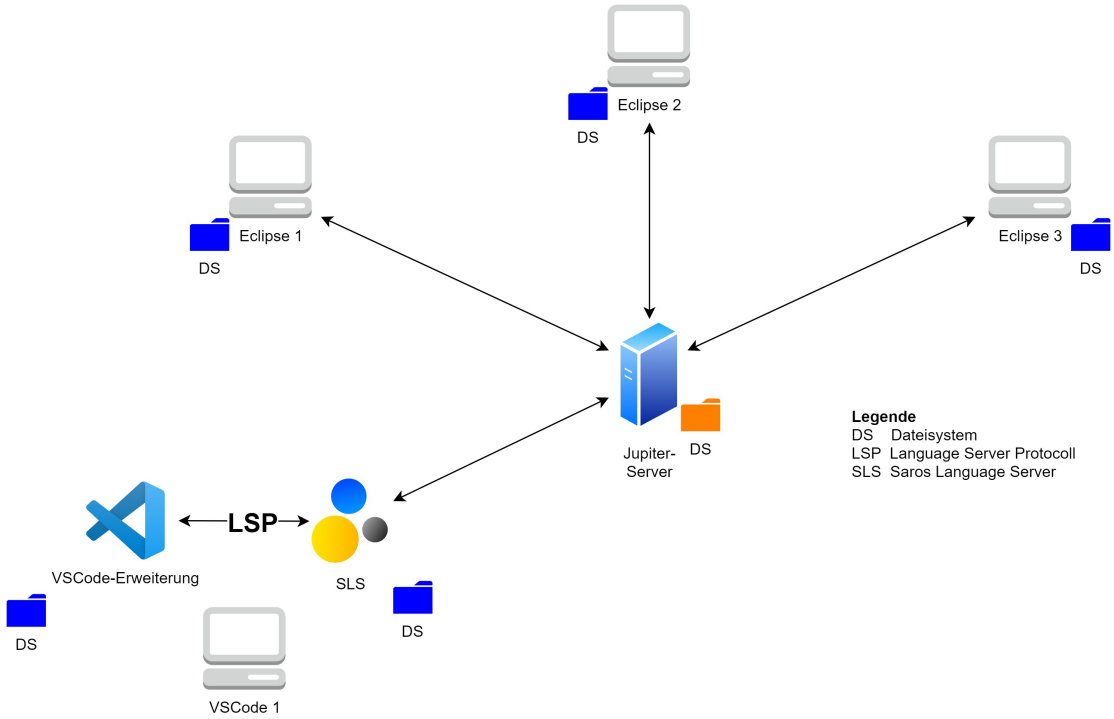

Abbildung 2: Architektur des Saros-VSCode Prototyps

3.3 Bekannte Probleme

Basierend auf der Vorstellung von Saros und der Architektur des Prototyps, wird im folgenden Kapitel auf Probleme eingegangen, welche bereits in der Ausarbeitung der Bachelorarbeit evaluiert wurden. So werden unter der Überschrift „Das Arbeitsbereich Problem“ [6, Zitat. S. 27] zwei Probleme genannt, für deren Lösung im Umfang der Ausarbeitung kein Platz gewesen war. Beim ersten Punkt wird dabei dargestellt, dass der SLS abstürzt, wenn beim Start der Erweiterung im VSCode-Arbeitsbereich

kein Ordner geöffnet ist. Als Übergangslösung deaktiviert sich die Erweiterung in diesem Fall von selbst [6, Vgl. S. 27]. Beim zweiten Punkt wird beschrieben, dass die Dateien der Sitzung immer im aktuell geöffneten Ordner in VSCode gespeichert werden und die Nutzer*innen den Speicherort nicht im Nachhinein selbst bestimmen können. Dem zugrunde liegt die grundsätzliche Architektur von VSCode, bei der alle Prozesse von verwendeten Erweiterungen als Kind-Prozesse des aktuellen VSCode-Arbeitsbereichs (*Extension Host Prozess*) laufen. Öffnet man in VSCode einen neuen Ordner (zum Beispiel, indem die Nutzer*innen die Möglichkeit hätten, einen anderen Speicherort für die Saros-Session festzulegen), wird auch ein neuer Arbeitsbereich geöffnet und die Nutzer*innen verlassen die Session des alten Arbeitsbereichs. In der Bachelorarbeit ist hierfür auch eine mögliche Lösung beschrieben, welche jedoch mangels Zeit nicht umgesetzt wurde [6, Vgl. S. 27]. Weitere Probleme sind entweder nicht bekannt oder nicht dargestellt. Das Ergebnis der Arbeit ist dabei wie folgt beschrieben: „Die Kernfunktionalität des Projektteilens ist ebenfalls enthalten und Änderungen werden über alle Teilnehmer hinweg synchronisiert. Der Prototyp weist sogar Funktionen auf, die über den minimalen Umfang hinaus gehen, [...]“ [6, Zitat S. 31]. Nachdem nun der bisherige Entwicklungsstand dargestellt wurde, kann im nächsten Kapitel mit der Analyse der Implementierung und Umsetzung der beschriebenen Konzepte begonnen werden.

4 Analyse des Prototyps

Die im vorherigen Kapitel beschriebene Architektur wirkte schlüssig und mit den genannten Technologien umsetzbar. Nach einer grundlegenden Einarbeitung in den Aufbau und die Funktionsweise von Saros und die Implementierung von Erweiterungen für den Editor VSCode, werden erste Funktionstests durchgeführt. Hierfür wird der Prototyp unter Windows 10 installiert und mit zwei getrennten Saros-VSCode Klienten getestet. Für Windows 10 wurde sich dabei als Betriebssystem entschieden, da die Entwicklung des Prototyps ebenfalls unter Windows 10 stattfand und im Gegensatz zum getesteten Ubuntu am wenigsten Probleme bei der Ausführung aufgetreten sind. In beiden Implementierungen sind vor der ersten Installation jedoch kleinere Bugfixes notwendig, mit denen zum Beispiel Probleme zwischen unterschiedlich verwendeten Versionen von NodeJS behoben wurden. Anschließend haben die implementierten Ansichten ohne Probleme funktioniert, die Assistenten zur Einrichtung der Konten haben den/die Nutzer*in ohne Probleme durch die Einrichtung geführt und es konnte sogar eine Session zwischen den beiden Klienten gestartet werden. Bei der Erstellung der Session sind dabei zwar in den Log-Kanälen *Saros* und *Saros (Server)* einige Java-Fehler gemeldet worden, welche jedoch in einem ersten Funktionstest vorerst ignoriert werden konnten, da die Session trotzdem erstellt wurde. Auch das Synchronisieren von Änderungen hat grundsätzlich funktioniert. Eine Änderung von Nutzer*in A wurde auch bei Nutzer*in B angezeigt, sowie umgekehrt. Probleme treten auf, wenn beide Nutzer*innen abwechselnd Änderungen machen. Vor allem bei Änderungen in derselben Zeile wurden die Änderungen beim jeweiligen Gegenpart nicht korrekt angewandt. Wenn beispielsweise Nutzer*in A ein Zeichen an einer bestimmten Stelle einfügt, hat der Saros-VSCode Klient bei Nutzer*in B ein bereits

existierendes Zeichen an dieser Stelle in manchen Fällen überschrieben, anstatt es einzufügen.

Nach einer Analyse des Designs im Hinblick auf diesen Fehlerfall, wurde herausgefunden, dass es sich hier um ein grundsätzliches Architekturproblem des Prototyps handelt. Dieses Problem führt dazu, dass einige Kernfunktionalitäten nicht korrekt implementiert werden können und somit nicht voll funktionsfähig sind. Aufgrund der schwere des Problems wurde daher an dieser Stelle die Analyse des Prototyps abgebrochen und der Fokus dieser Ausarbeitung auf die Lösung dieses Problems gesetzt, welches *das VSCode-Asynchronitäts Problem* genannt wurde. Um jedoch das Problem verständlich erklären zu können, müssen vorab noch einige theoretische Hintergründe zu Saros erläutert werden.

5 Theoretische Hintergründe

Bevor im Anschluss an dieses Kapitel auf die genaue Problemstellung eingegangen werden kann, sind im folgenden Kapitel zuerst die Kernkomponenten von Saros näher erläutert. Das dabei erlangte Wissen ist zusätzlich auch für die Kapitel nach der Problembeschreibung für die Erarbeitung möglicher Lösungen relevant. Speziell folgen zwei Einführungen zu den wichtigsten Technologien in Saros: operational transformation und Jupiter. Die Grundlagen sind dabei ausführlich ausgearbeitet, um auch ein besseres Verständnis für die allgemeinen Probleme der verschiedenen Implementierungen und Konzepte in den späteren Kapiteln vermitteln zu können. Dieses Verständnis über beispielsweise historische Entwicklungen oder andere mögliche Konsistenzmodelle ist dabei auch notwendig, um die später entwickelten Konzepte verstehen und hinterfragen zu können. Außerdem stellt diese Ausarbeitung eine theoretische und literarische Grundlagenforschung dar, um darauf aufbauend weiterführende Arbeiten anschließen zu können.

5.1 Operational Transformation

Operational transformation ist ein grundlegender Bestandteil des Jupiter-Algorithmus und somit auch von Saros, wodurch dieser Bereich für ein besseres Verständnis zuerst betrachtet werden muss. Dabei ist OT die in Saros genutzte Technologie zur Erhaltung der Konsistenz bei der gemeinschaftlichen Bearbeitung eines Inhalts durch mehrere Klienten. Vor OT mussten Dokumente dabei noch manuell zum Beispiel per Mail verteilt werden, damit dann eine andere Person in ihrer lokalen Kopie Änderungen vornehmen konnte. Diese Kopie wurde dann anschließend wieder manuell zurück an den Absender geschickt. Um dieses Problem zu umgehen, wurden kollaborative Editoren entwickelt, mit denen Mitglieder eines Teams gemeinsam an demselben Dokument arbeiten können, ohne sich die Änderungen dabei manuell über ein weiteres Medium zusenden zu müssen. Ziel von operational transformation ist es hierbei dafür zu sorgen, dass im Fall eines Konfliktes, wie zum Beispiel der gleichzeitigen Bearbeitung derselben Ressource, diese im Anschluss weiterhin überall konsistent bleibt. Grundsätzlich gibt es dabei zwei Ansätze, wie das gemeinschaftliche Arbeiten an einer Ressource umgesetzt werden kann. Beim asynchronen kollaborativem Arbeiten werden die Änderungen zwischen den Teilnehmern nicht in Echtzeit, sondern

zum Beispiel regelmäßig einmal täglich oder durch einen manuellen Commit wie bei Git synchronisiert. Beim synchronen kollaborativem Arbeiten werden die Änderungen wiederum in Echtzeit synchronisiert. Die Nutzer*innen arbeiten dabei unterbrechungsfrei an einer gemeinsamen Ressource auf einem Server oder einer lokalen Kopie und sehen dabei ebenfalls in Echtzeit jede Änderung von anderen Nutzer*innen [13, Vgl.]. Populäre Beispiele hierfür sind Google Docs, die Microsoft Office Suite oder auch Saros.

Bei der Konzeptionierung einer solchen Architektur, wird weiterhin zwischen zwei Lock-Mechanismen unterschieden. Beim traditionellen pessimistischen Lock wird beispielsweise durch einen wechselseitigen Ausschluss den Nutzer*innen eine konfliktfreie Bearbeitung garantiert, indem die zu bearbeitende Ressource vorher reserviert und bei allen anderen Nutzern*innen die Bearbeitung dieser Ressource gesperrt wird. Nach der Bearbeitung wird die Ressource wieder freigegeben und den anderen Nutzer*innen zur Verfügung gestellt. Wie auch im Paper „Concurrency Control in Groupware Systems“ von C.A. Ellis und S.J. Gibbs, welches im weiteren Verlauf der Arbeit noch weiter erläutert wird, festgestellt, gehen für kollaborative Echtzeit-Texteditoren damit jedoch drei grundlegende Probleme einher:

1. Durch das ständige Anfordern und Freigeben von Locks entsteht ein großer Datenoverhead, welcher vor allem durch die hohen Latenzen im Internet problematisch ist.

2. Es muss die Granularität der Sperren geklärt werden – es ist dabei schwer zu entscheiden, ob nur der veränderte Buchstabe, ein ganzes Wort, die Zeile oder sogar das gesamte Dokument während der Bearbeitung gesperrt wird.

3. Es muss definiert werden, wann eine Sperre angefordert und wann wieder freigegeben wird. Soll zum Beispiel eine Sperre schon bei einem Mouseover über einem Wort aktiviert und anschließend wieder gelöst werden oder erst, wenn sich der Cursor in dem Wort zum Beispiel durch einen Klick befindet? [14, Vgl. S. 402].

Zur Umgehung dieser Probleme und zur Implementierung eines Echtzeit-Charakters, wurde unter anderem auch von den Autor*innen des Papers die Anforderung gestellt, dass harte Locks bzw. ein pessimistisches Locken vermieden werden muss [14, Vgl. S. 402]. Als Alternative hierzu wird dabei ein Verfahren beschrieben, welches auch als optimistische Locks bekannt ist. Dabei haben alle Nutzer*innen einer Session gleichzeitig lesenden und schreibenden Zugriff auf alle Ressourcen, wobei jeder/jede Nutzer*in auf einer lokalen Kopie desselben Dokuments arbeitet. Im Hintergrund werden die Änderungen zwischen den einzelnen Nutzer*innen z.B. mit einem zentralen Server oder direkt zwischen den Klienten synchronisiert und auftretende Konflikte aufgelöst. Mit dieser Architektur wird allein schon aufgrund der Lokalität der Daten die Reaktions- und Antwortzeit der Software stark verbessert, da unter anderem keine Locks die Arbeit an einer Ressource verbieten und trotzdem ein Arbeiten von mehreren Personen in Echtzeit an einem Dokument ermöglicht [14, Vgl. S. 401].

Zusätzlich zu verschiedenen Locking-Strategien, wird noch allgemein zwischen zwei Strategien zur Synchronisierung lokaler Daten mit zum Beispiel einem Server

unterschiedenen. Bei der differential synchronization wird auf Seite des Klienten eine Differenz zwischen dem Stand des Nutzers und dem auf dem Klienten gespeicherten Stand des Servers erzeugt und anschließend an diesen gesendet [15, Vgl. S. 14-15]. Dies hat den Vorteil, dass es zwar ressourcensparender als andere Konzepte ist, dafür jedoch zum Beispiel einzelne Änderungen nicht rückgängig gemacht werden können. Bei der operational transformation hingegen erfolgt eine Synchronisierung der Änderung von einem/einer Nutzer*in und zum Beispiel einem Server auf Basis von Änderungen. Jede Änderung an einer Ressource wird dabei protokolliert und mit dem Server synchronisiert [13, Vgl.].

Saros basiert durch die Verwendung des Jupiter-Algorithmus auf dem Ansatz der Änderungssynchronisierung mit OT. Der große Vorteil dieses Ansatzes ist eine Robustheit gegenüber hohen Latenzen aufgrund lokaler Datenstände, sowie einer Konfliktauflösung, welche basierend auf jeder einzelnen Änderung auch rückwirkend berechnet werden kann. Da beim hier verwendeten Ansatz der OT jeder Klient auf einer lokalen Kopie derselben Ressource arbeitet und die verwendeten Ressourcen nicht gelockt werden, kann auch schnell der Fall eintreten, dass zwei Nutzer*innen gleichzeitig eine Änderung an derselben Ressource vornehmen. Ein einfaches Beispiel hierfür ist in Abbildung 3 dargestellt. In diesem Beispiel werden, der Einfachheit halber Änderungen nur zwischen einem Klienten und einem Server synchronisiert, wobei sich die Problematik bei der Kommunikation zwischen einem Server und mehreren Klienten beziehungsweise Nutzer*innen nur verschärft. Sowohl der Klient als auch der Server starten dabei mit dem initialen Text „ABCDE“. Die Indexierung von Zeichen in einem Texteditor wie zum Beispiel VSCode, startet dabei beim Index 0 und beschreibt das Zeichen vor dem Index. In diesem Beispiel entspricht der Index 4 dem Zwischenraum der Buchstaben D und E, wobei D das Zeichen vor dem Index ist. Sowohl der Server, als auch der Klient beginnen gleichzeitig mit der Bearbeitung des initialen Textes in derselben Zeile. Dabei ist wichtig zu beachten, dass beide Seiten nicht auf die Sekunde genau gleichzeitig mit der Bearbeitung beginnen müssen. Ausschlaggebend ist, dass der Server und der Klient ihre Änderungen an die jeweils andere Seite senden, bevor die Änderungen der anderen Seite empfangen und verarbeitet wurden. In der Praxis ist dies auch in Saros durch bereits erwähnte Latenzen bei der Verwendung des Internets als Übertragungsmedium möglich.

Auf der Seite des Klienten wird das Zeichen mit dem Index 4 entfernt, wodurch der Text „ABCE“ entsteht. Auf der Seite des Servers wird gleichzeitig das Zeichen mit dem Index 2 entfernt. Dadurch entsteht der Text „ACDE“. Nachdem der Server das Zeichen mit dem Index 2 entfernt hat, wird er über die Änderung des Nutzers informiert, in welcher das Zeichen an Index 4 entfernt wurde. Das Problem an dieser Stelle ist, dass die Änderung vom Klienten auf Basis des Textes „ABCDE“ berechnet wurde und jetzt vom Server aber auf den bereits geänderten Text „ACDE“ angewandt wird. An Index 4 des Textes beim Server befindet sich daher nicht mehr das Zeichen „D“, wie beim Klienten, sondern das Zeichen „E“. Daraus resultiert beim Server der finale Text „ACD“. Der Klient empfängt zur selben Zeit die Änderung vom Server, in welcher das Zeichen an Index 2 entfernt wird. Beim Klienten gab es hier keine Verschiebung des Index, weshalb an der Stelle, wie auch beim Server, das Zeichen „B“ entfernt wird.

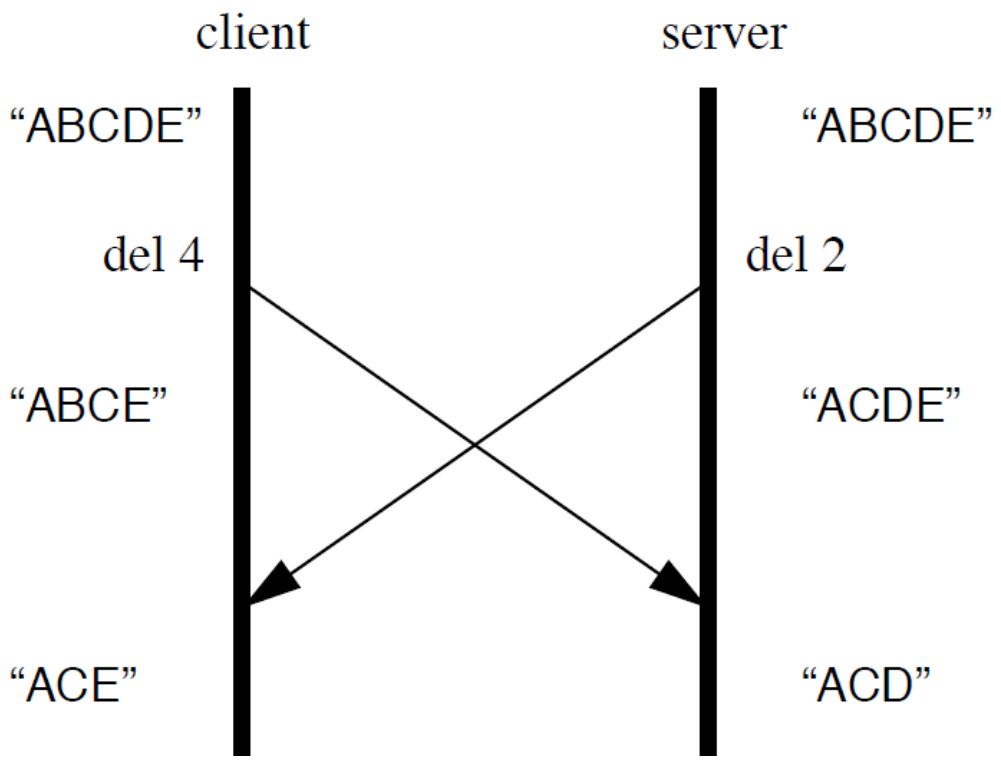

Abbildung 3: Darstellung eines möglichen Konfliktfalls zwischen Klient und Server [16]

Aufgrund der zwischenzeitlich unterschiedlichen Indexe zwischen Server und Klient, sind jedoch unterschiedliche Resultate entstanden, wobei der Klient abschließend bei „ACE“ und der Server bei „ACD“ endet. Das hier beschriebene Problem ist dabei ein klassischer Konfliktfall, welcher durch OT-Algorithmen erkannt und anschließend aufgelöst werden soll. Operational transformation besteht dabei aus mehreren Komponenten, welche im Folgenden näher betrachtet werden.

Grundsätzlich ist zu erwähnen, dass alle Änderungen an einem Dokument, wie das Einfügen oder Entfernen als Operationen repräsentiert werden. Die Operation kann dabei auf das aktuelle Dokument angewandt werden, wodurch ein neuer Status resultiert [17, Vgl.]. Wie in Abbildung 4 dargestellt, kann im Allgemeinen hierbei zwischen zwei Anordnungen von Operationen unterschieden werden: nicht-überlappende (links) und überlappende (rechts) Operationen. Differenziert wird dabei zwischen den unterschiedlichen Versandt- und Empfangszeiten auf dem Klienten (in der Abbildung als Seite bezeichnet) S1 und dem Server S2, sowie den Operationen o1 und o2. Während bei nicht-überlappenden Operationen die Operation o1 vom Klienten S1 und die Operation o2 vom Server S2 auf beiden Seiten nacheinander in der gleichen Reihenfolge ausgeführt werden, wird in dem Beispiel der überlappenden Operation die Operation o2 vom Server S2 ausgeführt und verschickt, bevor die zeitlich vorher ausgeführte Operation o1 vom Klienten S1 den Server S2 erreicht und dort ausgeführt

wird.

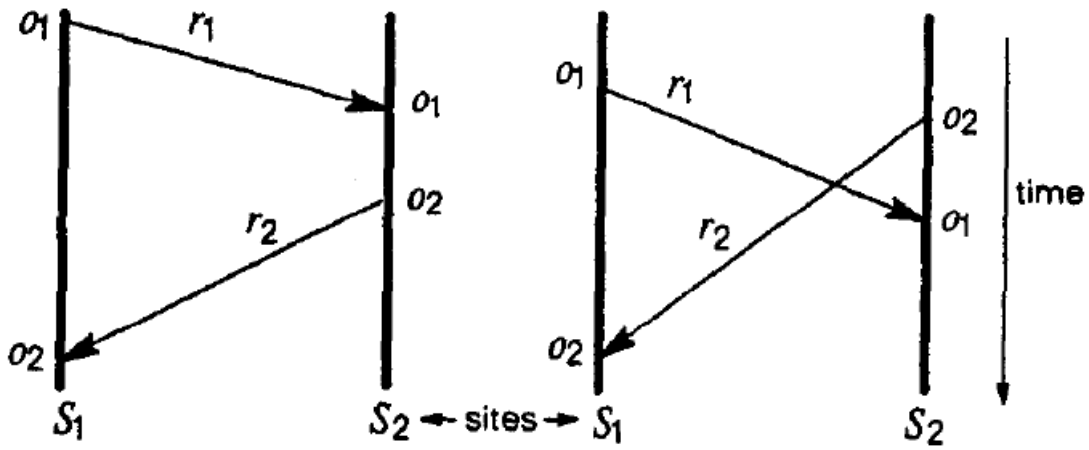

Abbildung 4: Überlappende- und nicht überlappende Operationen [14]

Einer der wichtigsten Bereiche bei OT ist dabei die Behandlung der im vorherigen Beispiel beschriebenen, konkurrierenden Änderungen. Hierfür werden mithilfe von Transformationsfunktionen aus zwei Operationen, die auf unterschiedlichen Klienten auf denselben Datenstand angewandt wurden, neue Operationen (o′) berechnet, welche nacheinander angewandt werden können. Ziel ist es dabei, bestehende Konflikte aufzulösen und dabei neben der Erhaltung der Konsistenz, über die verteilten Ressourcen auch die Intention beider Operationen beizubehalten.

In der Historie von OT hat sich das ursprüngliche Konsistenzmodell aus dem Paper von Ellis und Gibbs durch Erfahrung und weiterer Anforderungen weiterentwickelt. In dem 1989 veröffentlichtem Paper wurden initial drei Anforderungen für kollaborative Editoren definiert, wodurch die Konsistenz immer gewährleistet werden soll:

- Definition 1 (Konvergenz): „Die Konvergenzeigenschaft gibt an, dass die Objekte der Seiten auf allen Seiten im Ruhemodus identisch sein müssen“ [14, Zitat, frei übersetzt nach S. 402]. Als Ruhemodus wird dabei ein Zustand definiert, bei dem alle Operationen verarbeitet wurden und keine weiteren Operationen auf eine Abarbeitung warten [14, Vgl. S. 402].

- Definition 2 (Erhaltung der Kausalität): „Die Vorrangeigenschaft gibt an, dass wenn eine Operation, o, einer anderen Operation, p, voraus geht, dann muss auf jeder Seite die Ausführung von o vor p geschehen“ [14, Zitat, frei übersetzt nach S. 402]. Es wird dabei definiert, dass die Reihenfolge der Ausführung der Operationen auf jeder Seite identisch sein muss.

- Definition 3: „Das Groupware-System ist korrekt, wenn die Konvergenzeigenschaft und die Vorrangeigenschaft immer erfüllt sind“ [14, Zitat, frei übersetzt nach S. 402]. Mit Groupware-Systemen sind dabei computerbasierte Systeme gemeint, welche die Interaktion von zwei oder mehreren Nutzer*innen unterstützen [14, Vgl. S. 399].

Diese Definitionen sind unter den Konsistenzmodellen auch als CC-Modell (Causality preservation and Convergence) bekannt. Basierend darauf wurde von den Autor*innen des Papers auch der erste Transformationsalgorithmus, der „distributed Operational Transformation Algorithm (dOPT)“ [14, Zitat S. 403] entwickelt. Hierbei wird jede Seite mit einem Identifier identifiziert und über eine Transformationsmatrix auftretende Konflikte aufgelöst. Bereits in der Abhandlung konnte dabei ein Fall identifiziert werden, bei dem Definition 1, also die Konvergenz, nicht gewährleistet werden kann. Eine genaue Problemstellung sowie mögliche Lösungsansätze sind dabei sowohl in dem Kapitel[8] als auch in dem Paper „Operational Transformation in Real-Time Group Editors: Issues, Algorithms and Archivements“ von Chengzheng Sun und Clarence Ellis[9] zu finden. In dieser Arbeit wird darauf jedoch nicht weiter eingegangen, da die beschriebene Problemstellung mit zukünftigen Entwicklungen und Erweiterungen gelöst wird. Grundsätzlich kann auch bei Transformationsalgorithmen noch zwischen zwei verschiedenen Ansätzen unterschieden werden. Bei einer inkludierenden Transformation (im Folgenden IT genannt) von Operation o1 gegen o2 wird die Operation o1 so transformiert, dass der Effekt von o2 direkt enthalten ist. Bei einer exkludierenden Transformation (im Folgenden ET genannt) von Operation o1 gegen o2, wird die Operation o1 so transformiert, dass der Effekt von o2 nicht enthalten ist [19, Vgl.]. Der eben vorgestellte dOPT-Algorithmus ist dabei ein IT-Algorithmus.

Allgemein hat sich im Laufe der Forschung mit der Entwicklung neuer Algorithmen auch die Definition von Konsistenz verändert. Während im ersten Konsistenzmodell von Ellis und Gibbs das System als in sich konsistent galt, wenn die Konvergenzeigenschaft und die Vorrangeigenschaft erfüllt sind (CC-Modell), wurden mit neuen Erkenntnissen auch neue Bedingungen definiert. Im CCI-Modell (Causality preservation, Convergence and Intention preservation) spielte die Wahrung der Intention des/der Autor*in der Operation eine Rolle. Neben den bekannten Faktoren galt ein System jetzt als konsistent, wenn das Ergebnis der Ausführung der Operation bei allen Klienten der Intention der Operation entspricht. Die Intention wird dabei als der Effekt einer Operation auf der Quell-Seite definiert [20, Vgl. S. 63]. Im CSM-Modell (Causality, single-operation effects and multi-operation effects) wird dabei mit Effektbeziehungen und der Wahrung der Effektbeziehungen als neues Konsistenzkriterium die Wahrung der Intention formal spezifiziert [21, Vgl. S. 459]. Das CA-Modell (Causality, Admissibility) verfolgt den Ansatz, dass neben der Kausalität der Aufruf jeder Operation in Ihrem Ausführungszustand zulässig ist. Durch den Aufruf nur zulässiger Operationen werden dabei die Effektbeziehungen immer gewahrt[10].

Mit der Weiterentwicklung der Technologie hinter operational transformation sind auch bei der Entwicklung von Konsistenzmodellen neue Transformationsalgorithmen entstanden, wobei einige wichtige in Tabelle 2 dargestellt werden.

[8] siehe [14, S. 405-406]
[9] siehe [18]
[10] siehe [22]

adOPTed[11]	Stellt eine Weiterentwicklung des dOPT-Algorithmus dar und löst die meisten Probleme dessen. Jedoch wird auch bei adOPTed keine Wahrung der Intention gewährleistet [24, Vgl.].
Generic Operational Transformation (GOT)	Verwendet erstmals eine Historie von Änderungen (Historical Buffer) und unterstützt damit das Rückgängigmachen von Änderungen (Undo) und das Wiederholen rückgängig gemachter Änderungen (Redo). Zusätzlich wird die Wahrung der Intention gewährleistet [24, Vgl.].
Optimized Generic Operational Transformation (GOTO)[12]	Eine Weiterentwicklung von GOT, wobei die Eigenschaften von adOPTed in den Algorithmus für die Lösung von konkurrierenden Änderungen hinzugefügt wurden [24, Vgl.].
Context-based Operational Transformation (COT)[13]	Ein anderer Ansatz stellt der COT-Algorithmus dar, bei dem im Allgemeinen der Historical Buffer von GOT durch die Verwendung von Kontextvektoren für eine Operation abgelöst wird [24, Vgl.].

Tabelle 2: Übersicht über mögliche Transformationsalgorithmen

Zu erwähnen ist hierbei, dass die dargestellten Algorithmen nur einen kleinen Ausschnitt der Forschung darstellen und zum Beispiel wichtige Algorithmen für das CSM-Modell, wie der landmark-based transformation Algorithmus (LBT)[14], dabei aus zeitlichen Gründen nicht weiter betrachtet werden können. Basierend auf dem Wissen in diesem Kapitel über die allgemeinen Mechanismen und Konsistenzmodelle in OT, sowie die möglichen Ansätze und Algorithmen, kann nun die Herangehensweise und die Intention hinter zukünftigen Designentscheidungen der späteren Kapitel zu neuen Konzepten besser nachvollzogen werden. Um das erlangte Wissen auch auf Saros anwenden zu können, wird im nächsten Kapitel die in Saros verwendete Implementierung von OT betrachtet.

5.2 Jupiter

Nach der Einleitung in die Hintergründe und Funktionsweisen von operational transformation und dem von Saros verwendetem Algorithmus, wird im Folgenden das System näher betrachtet, welchem die Architektur von Saros zu Grunde liegt. Für Saros wurde dabei als Zugriffskontrollsystem die Implementierung Jupiter gewählt. In der Veröffentlichung „High-Latency, Low-Bandwidth Windowing in the Jupiter Collaboration System" von David A. Nichols, Pavel Curtis, Michael Dixon und John Lamping wird Jupiter dabei als „groupware toolkit" [16, Zitat S. 111] bezeichnet, welches im Rahmen der Abhandlung um einen Zugriffskontrollalgorithmus erweitert wurde.

[14] siehe [26]

Die Haupteigenschaft von Jupiter ist dabei, dass alle verwalteten Ressourcen „verteilt und persistent“ [16, Zitat S. 111] sind. Als Beispiel für die Arbeitsweise des Systems wird Jupiter dabei von den Autoren mit einer Tafel verglichen, bei der Nutzer*innen kommen und gehen und immer Änderungen am Inhalt der Tafel vornehmen können. Wichtig dabei ist, dass wenn ein/eine Nutzer*in nach einem gewissen Zeitraum zurückkommt, er/sie immer den aktuellen Stand sieht. Zusätzlich sind die Änderungen immer nachvollziehbar und persistent, sofern keine anderen Nutzer*innen die entsprechenden Bereiche verändert haben [16, Vgl. S. 111-112].

Jupiter besteht zu Erfüllung dieser Anforderungen dabei aus zwei Komponenten: dem Jupiter-Server und dem Jupiter-Klienten. Der Jupiter-Server besitzt grundsätzlich immer den korrekten Datenstand, welcher dann von verschiedenen Klienten aktualisiert wird. Weiterhin kümmert sich der Server um die Verteilung von Änderungen an alle anderen Jupiter-Klienten und die Auflösung von Konflikten. Da Jupiter jedoch auch bei hohen Latenzen und niedriger Bandbreite robust und einwandfrei arbeiten soll, besitzt jeder Klient zusätzlich seinen eigenen Datenstand. Die Nutzer*innen arbeiten dabei ausschließlich mit ihren Klienten auf den lokalen Daten, wobei sich der Jupiter-Klient unter anderem um die Kommunikation mit dem Server oder die lokale Auflösung von Konflikten kümmert. Wie bereits in Kapitel 3.2 bei der Architektur von Saros erläutert, kommunizieren die einzelnen Jupiter-Klienten ausschließlich mit dem Jupiter-Server, wobei dieser unter anderem die gesamte Verwaltung der Kommunikation, inklusive der Weiterleitung von Änderungen beinhaltet.

Bei der Auswahl eines Kontrollalgorithmus für das Jupiter-System, haben sich die Autoren aufgrund der Anforderungen, bei beispielsweise hohen Latenzen, gegen die Verwendung von pessimistischen Zugriffskontrollalgorithmen und für die Verwendung optimistischer Zugriffskontrollalgorithmen entschieden [16, Vgl. S. 112]. Verwendet wird dabei interessanter Weise der bereits im vorherigen Kapitel vorgestellte dOPT-Algorithmus. Interessant ist dies, weil beim dOPT-Algorithmus eigentlich eine Kommunikation direkt zwischen den Klienten vorgesehen ist und im jetzigen Ansatz zur Synchronisation der Änderungen zwischen dem Jupiter-Klient und dem Jupiter-Server verwendet wird [16, Vgl. S. 118]. Die Nachrichten werden dabei bidirektional über eine eigene TCP-Verbindung zwischen jedem Klienten und dem Jupiter-Server gesendet [16, Vgl. S. 114].

Im Folgenden werden die Details des Jupiter-Systems betrachtet. Wenn ein/eine Nutzer*in eine Änderung an einer Ressource, beispielsweise einer Zeile in einem Dokument macht, wird diese Änderung sofort auf dem lokalen Datenstand des Jupiter-Klienten angewandt und der Jupiter-Klient sendet eine Benachrichtigung über die Änderung an den Jupiter-Server. Umgekehrt arbeitet der Server identisch, wobei Änderungen am Server-Datenstand vom Server für gewöhnlich aufgrund einer gemeldeten Änderung eines Klienten vorgenommen werden. Wenn während der Übertragung die andere Partei ebenfalls eine Änderungsbenachrichtigung versendet, wird beim Empfänger die Änderung entsprechend transformiert [16, Vgl. S. 114-115].

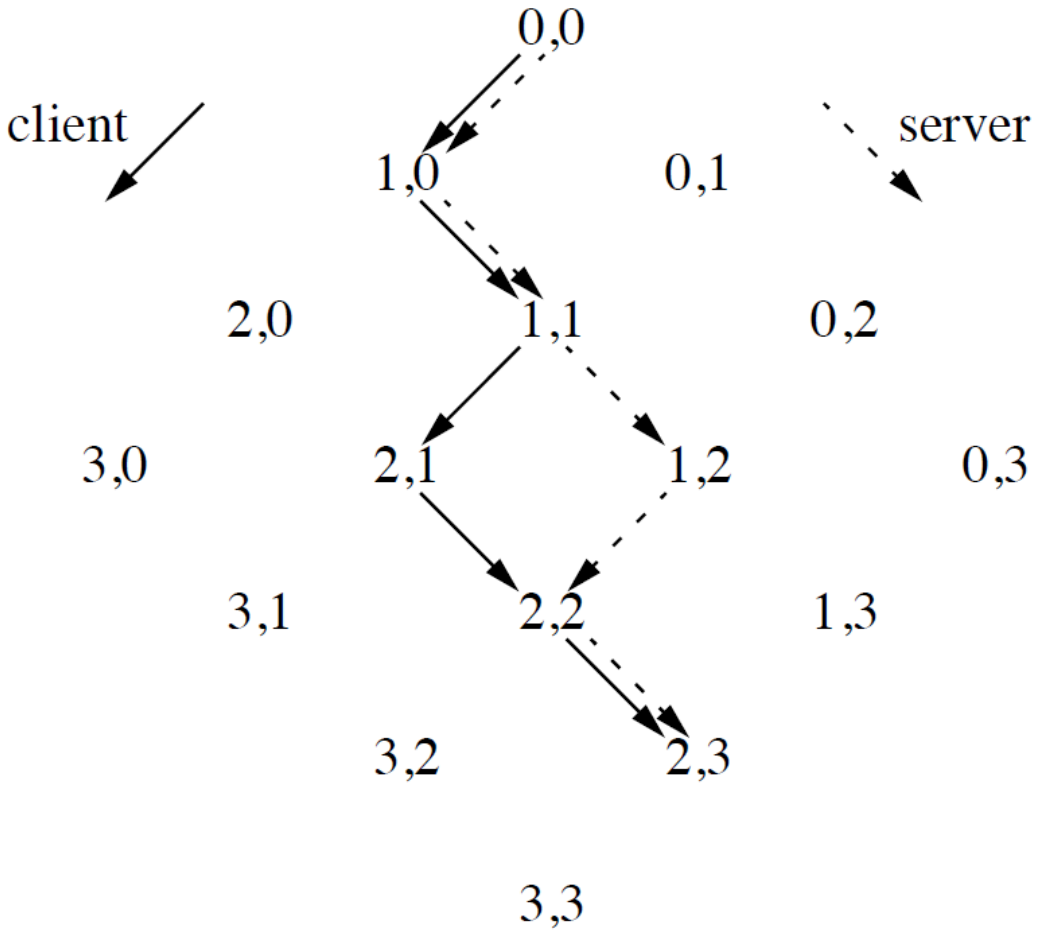

Abbildung 5: 2D-Zustandsraum zwischen Klient und Server [16]

Während der Verarbeitung vieler Nachrichten beziehungsweise in einem System, mit vielen Jupiter-Klienten, können die einzelnen Datenstände sehr schnell divergieren. Es entsteht dabei zwischen dem Jupiter-Klienten und dem Jupiter-Server ein 2D-Zustandsraum, welcher in Abbildung 5 ersichtlich ist. Jeder Knoten zeigt dabei die Anzahl an Nachrichten, welche bereits zu diesem Zeitpunkt durch den Klienten (links) und den Server (rechts) verarbeitet wurden. Wenn die Nachrichten ohne Konflikte in der gleichen Reihenfolge verarbeitet wurden, gehen beide Pfade dabei in dieselbe Richtung und bei einem Konflikt trennen sie sich. In der Abbildung ist zu sehen, dass beide Parteien im Status (1,1) erstmals eine Nachricht mit unterschiedlichem Nachrichteninhalt verarbeiten. Anschließend empfangen sie die jeweils andere Änderungsbenachrichtigung und verarbeiten diese, wobei beide Parteien wieder im Status (2,2) enden. Im letzten Schritt dieses Beispiels wird wieder eine Nachricht ohne Konflikt vom Server verarbeitet. Jede Nachricht wird neben der Änderung dabei mit dem Status gelabelt, in welchem sich der Versender zum Zeitpunkt der Erstellung der Nachricht befunden hat. Der Zugriffskontrollalgorithmus im Jupiter-System verwendet anschließend diese Label, um Konflikte erkennen zu können [16, Vgl. S. 116].

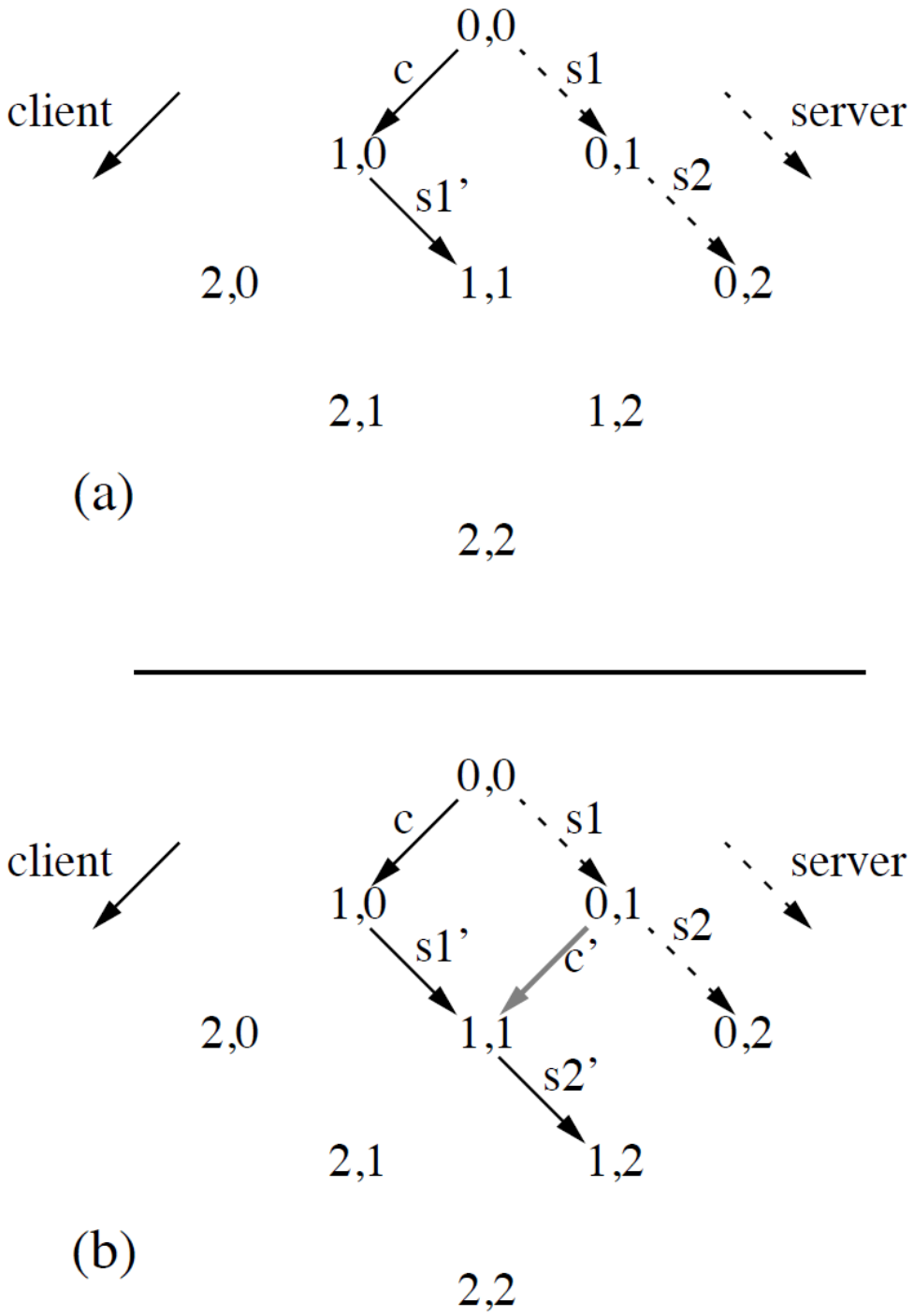

Abbildung 6: 2D-Zustandsraum bei mindestens 2 unverarbeiteten Änderungen [16]

Wenn der Zugriffskontrollalgorithmus über die Label einen Konflikt erkannt hat, wird dieser, mit der in Jupiter eingeführten *xform*-Funktion, wieder aufgelöst. Der Jupiter-Algorithmus garantiert dabei, dass egal wie stark der Datenstand von einem Klienten mit dem des Servers divergiert, mithilfe des Zustandsraumes und der Label

am Ende beide Parteien wieder dasselbe Label besitzen. Dabei wird bei gleichen Labeln garantiert, dass auch die Datenstände gleich sind. Die *xform*-Funktion generiert aus einem Paar von Änderungen des Jupiter-Server und Klienten, welche auf denselben Stand basieren, transformierte Nachrichten, wobei eine transformierte Nachricht die Änderung der ursprünglichen Nachricht jeweils so anpasst, dass Server und Klient wieder denselben Status erreichen [16, Vgl. S. 116]. Solange der Jupiter-Server und der Klient dabei nur eine Änderung divergieren, wie in Abbildung 5 beschrieben, kann die *xform*-Funktion so direkt verwendet werden. Komplexer wird es, wenn beide Datenstände mehr als nur eine Änderung divergiert sind, wie in Abbildung 6 zu sehen ist.

Im Bereich (a) der Abbildung 6 ist dabei ein Fall im 2D-Zustandsraum dargestellt, bei dem die Datenstände im Status (0,0) nach (1,0) und (0,1) divergieren. Der Klient hat Änderung c und der Server gleichzeitig die Änderung s1 durchgeführt. Der Klient empfängt daraufhin die Konfliktnachricht s1, transformiert diese zu s1‘ auf Basis von c und wendet sie an. Basierend auf dem Status (0,1) erstellt der Server jedoch die Änderung s2, ohne die Änderung c des Klienten verarbeitet zu haben. Der Jupiter-Klient empfängt ebenfalls die Konfliktnachricht s2, kann diese jedoch nicht einfach mit der *xform*-Funktion transformieren und anwenden, da auf Seiten des Klienten bereits die Änderung c verarbeitet wurde. In diesem Fall berechnet der Klient c‘, die transformierte Nachricht, welche der Server berechnet hätte und berechnet auf Basis von c‘ die transformierte Nachricht s2‘. Die Nachricht s2‘ hat dabei die Änderungen in s2 auf Basis von c neu berechnet und basiert somit auf der Änderung c. Wenn jetzt auf Seiten des Servers eine Änderung s3 generiert worden wäre, würde der Klient diese dann auf Basis von c“ transformieren und so weiter. Dieses Verhalten ist auch in Bereich (b) der Abbildung 6 zu erkennen. Anschließend kann der Server im Status (0,2) ebenfalls die Änderung c transformieren und bei sich c‘ anwenden, wodurch die Datenstände des Jupiter-Servers und Klienten im Status (1,2) enden. Mithilfe dieses Algorithmus können alle Datenstände, egal wie weit diese divergieren, mit zusätzlichem Rechenaufwand wieder zusammengeführt werden [16, Vgl. S. 117].

Dabei muss beachtet werden, dass alle Änderungen, welche noch nicht auf der anderen Seite angewandt und bestätigt wurden (in dem Beispiel von eben die Änderung c), auf der entsprechenden Seite für die Berechnung eventueller Transformationen vorgehalten werden müssen. Außerdem geben die Autoren des Papers an, dass Jupiter-Klient und Server in regelmäßigen Abständen explizite Bestätigungen bzw. Statusmeldungen austauschen, damit die Datenstände nicht in das Unendliche divergieren. Zusätzlich dürfen Änderungsbenachrichtigungen erst versendet werden, nachdem die Änderung lokal angewandt wurde [16, Vgl. S. 118].

Verglichen mit der initialen Implementierung von dOPT, wurde durch die Anpassung bei Jupiter eines der grundlegendsten Probleme des Algorithmus gelöst. Trotzdem sind dabei nicht alle Änderungen ohne Probleme transformierbar, weshalb bei Jupiter für diese Fälle feste Regeln definiert wurden. Wenn zum Beispiel Änderung c und s1 beide einen Text ersetzen, wird durch Jupiter ein finaler Stand erzeugt, welcher die Regionen aus beiden Änderungen entfernt und anschließend, sortiert nach der Startposition des ersten Zeichens des jeweils entfernten Textes, beide neuen Texte einfügt. Wenn jedoch Jupiter-Klient und Jupiter-Server Textinhalte an derselben Stelle einfügen möchten, wird immer der Text des Servers bevorzugt und ausschließlich ein-

gefügt. Ein anderer Fall ist, wenn zum Beispiel der Klient einen Text ersetzen möchte, welcher sich inmitten eines gelöschten Bereiches des Servers befindet. An dieser Stelle wird nach dem Löschen der neue Text immer eingefügt, um die Eingabe eines/einer Nutzer*in zu erhalten [16, Vgl. S. 119]. Dies ist nur ein Ausschnitt der Randfälle, welche ebenfalls in der Veröffentlichung zu Jupiter betrachtet wurden. Allgemein wurde von den Autoren dabei versucht, die Regeln für das Verhalten bei Randfällen nach folgenden Richtlinien zu definieren:

1. „Ein Set von Operationen sollte immer unter den Transformationsregeln geschlossen werden
2. Versuche, keine Nutzereingaben zu verwerfen
3. Versucht, verwirrende Anwendungscodes zu vermeiden“ [16, Zitat, frei übersetzt nach S. 119]

Nach dem nun die grundlegenden Konzepte hinter Saros, operational transformation und Jupiter bekannt sind, kann das im Kapitel der Analyse erkannte Problem im Detail betrachtet werden. Die Abläufe im nächsten Kapitel sowie in den folgenden Kapiteln mit den Konzeptanalysen, bauen dabei ebenfalls auf dem bisherigen Wissen auf.

6 Das VSCode-Asynchronitäts Problem

In diesem Kapitel wird das eigentliche Problem der aktuellen Architektur des Prototyps analysiert. Wie bereits in Kapitel 4 evaluiert wurde, gibt es im getesteten Prototyp Probleme mit dem Synchronisieren der Änderungen zwischen zwei Klienten. Nach eingehender Analyse der Implementierung, konnte dabei das im folgenden dargestellte Problem in der Architektur gefunden werden.

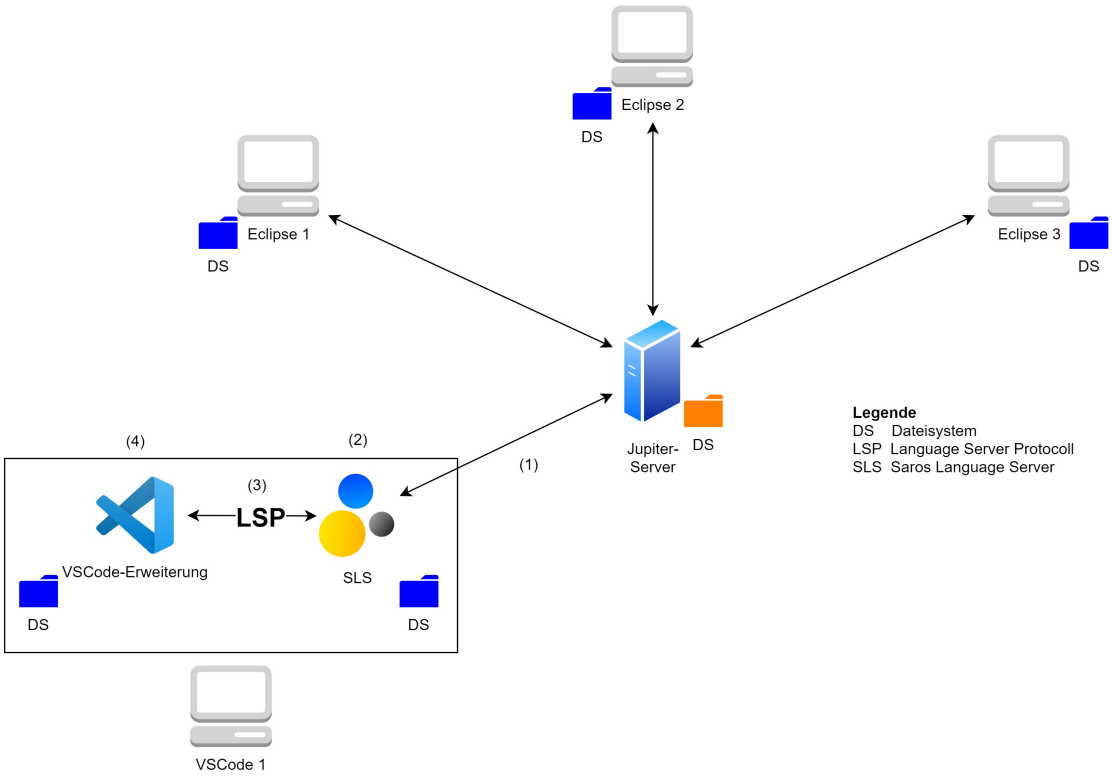

Abbildung 7: Architektur des Saros-VSCode Prototyps

In Abbildung 7 ist erneut die Architektur von Saros in Verbindung mit dem neuen Prototyp zu sehen. Wie bereits bekannt ist, stellt die Markierung (1) dabei die Verbindung zwischen dem SLS und dem Jupiter-Server dar, (2) den Saros Language Server selbst, wobei der SLS die Java-Implementierung mit den eigentlichen Saros-Funktionalitäten darstellt, (3) die Verbindung zwischen dem Saros-VSCode Klienten und dem SLS über das neu eingeführte Language Server Protokoll und (4) den Saros-VSCode Klienten, welcher die Implementierungen für die Anzeigen auf der Benutzer*innenoberfläche, sowie Assistenten für die Anmeldung und vieles mehr beinhaltet. Die Logik von Saros ist dabei, wie in Kapitel 3.2 erläutert, extra in den SLS ausgelagert worden und der Saros-VSCode Klient sollte ausschließlich Anzeigelogik beinhalten.

Wenn jetzt der dOPT-Algorithmus aus Kapitel 5.2, beziehungsweise die Abwandlung von Jupiter betrachtet wird, ist es für die Konfliktlösung der *xform*-Funktion erforderlich, dass der Jupiter-Algorithmus bei allen Komponenten implementiert ist. Genau diese Implementierung fehlt aufgrund der Architektur jetzt jedoch im Saros-VSCode Klienten und bildet damit den ersten Teil des Problems. Der zweite Teil basiert auf der Architektur von VSCode selbst, wie sie bereits in Kapitel 3.2 erläutert wurde. Der aktuelle Prototyp arbeitet so, dass wenn vom Jupiter-Server eine Änderungsbenachrichtigung an den SLS gesendet wird, dieser die Konfliktauflösung übernimmt und dann die eventuell überarbeitete Änderung mithilfe von LSP an den Saros-VSCode Klienten schickt. Wenn umgekehrt eine Änderung vom Saros-VSCode Klienten vorgenommen wird, sendet dieser die Änderung über LSP an den SLS und der SLS leitet anschließend die Änderung an den Jupiter-Server weiter. Da der SLS

über das Dateisystem auf den entsprechenden Dateiinhalt zugreift [6, Vgl. S. 32], geht der SLS davon aus, immer auf dem aktuellen Stand des Saros-VSCode Klienten zu arbeiten. VSCode hat den Saros-VSCode Klienten dabei in einen extra Prozess ausgelagert und dieser wird über die *Extension API* im Nachhinein über eine Änderung informiert. Das Problem ist hierbei, dass die Events der *Extension API* asynchron sind, damit die Performance der Anwendung VSCode nicht beeinflusst wird. Es kann also sein, dass der SLS eine Änderung an den Saros-VSCode Klienten sendet, welche auf einem Datenstand zum Zeitpunkt X basiert. Mittlerweile ist jedoch eine Änderung des/der Saros-VSCode Nutzer*in getätigt worden, welche aufgrund der Asynchronität der *Extension API* zum Zeitpunkt des Versendens der Nachricht vom SLS an den Saros-VSCode Klient, dem SLS noch nicht bekannt ist. Der Datenstand im Saros-VSCode Klient basiert somit schon auf einen Datenstand zum Zeitpunkt Y und der Saros-VSCode Klient würde es nicht mitbekommen, da die Logik hierfür in den SLS ausgelagert ist. Zusammenfassend kann das Problem so beschrieben werden, dass aufgrund der Asynchronität der *Extension API* in der vorliegenden Architektur Änderungen auf falschen beziehungsweise unbekannten Datenständen angewandt werden können, da der Saros-VSCode Klient nur die Änderungen an den SLS weiterleitet. Der SLS, als auch der Saros-VSCode Klient beziehen die Änderungen aktuell auch nicht auf die zu diesem Zeitpunkt bestehenden Datenstände, weshalb dieser Fehlerfall auch nicht vom Prototyp erkannt werden kann.

Verknüpft mit dem ersten Teil des Problems, beinhaltet der Saros-VSCode Klient generell keine Funktionalität, um mit solchen speziellen Randsituationen umgehen zu können. Um ein vergleichbares Beispiel auch für die Anwendung bei möglichen Lösungen verwenden zu können, wird der eben beschriebene Problemfall im Folgenden einmal dargestellt. Das Beispiel wurde dabei bereits mit der Projektgruppe von Saros öffentlich bei GitHub diskutiert und ist dort[15] auch einsehbar.

Gegeben ist ein Szenario mit zwei Nutzer*innen, in der „Alice“ als Host mit Saros für Eclipse und „Bob“ Saros für VSCode verwendet (siehe Abbildung 8). Beide Nutzer*innen arbeiten dabei am gleichen Dokument mit dem Inhalt *'Hello world'*.

1. Alice beginnt mit der Änderung *' my unique'* an Index 5, woraus sich der Inhalt *'Hello my unique world'* für Alice ergibt. Diese Änderung wird in eine Jupiter-Aktivität übertragen und an Bob geschickt.

2. Gleichzeitig arbeitet Bob am gleichen Text und fügt ein *'!'* an Index 11 hinzu, woraus sich für Bob *'Hello world!'* ergibt. Diese Änderung wird an den SLS geschickt.

3. Bevor der SLS Bobs Änderung erhält und bearbeitet, empfängt er die Änderung von Alice. Der SLS sendet an den Saros-VSCode Klient die Änderung von Alice (*' my unique'* an Index 5), wodurch sich bei Bob *'Hello my unique world!'* ergibt. Dabei ist zu erwähnen, dass diese Änderung dabei nur ohne Probleme funktioniert, da Alice Änderung vom Index her vor dem *'!'* stattfindet. Damit wird das *'!'* nur weiter nach hinten geschoben.

4. Nachdem Alice Änderung verarbeitet wurde, erreicht die Änderung von Bob den SLS. Diese wurde dabei jedoch erstellt, bevor im Saros-VSCode Klient die

[15] siehe `https://github.com/saros-project/saros/pull/1111#issuecomment-799384566`

Änderung von Alice angewandt wurde. Da der SLS denkt, dass Bob die Änderung von Alice´s verarbeitet hat, sieht er auf Bobs Seite keine weiteren notwendigen Schritte und schickt Bobs Änderung ('!' an Index 11) an Alice. Bei Alice wird die Änderung verarbeitet und es ergibt sich bei Ihr '*Hello my un!ique world*', da beim Klienten von Bob der Index falsch berechnet wurde[16].

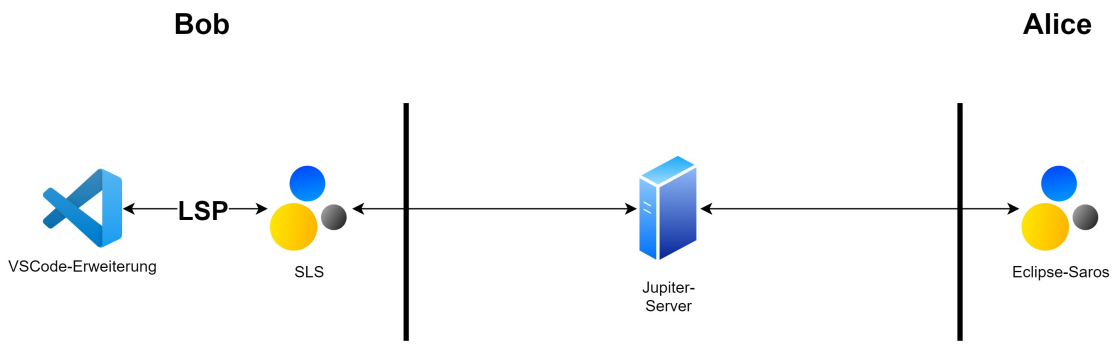

Abbildung 8: Ausgangssituation des Beispiels

Mithilfe dieses Beispiels kann die Problemstellung sehr gut nachgestellt werden. Durch die fehlende Logik zur Behandlung von konkurrierenden Änderungen beim Saros-VSCode Klient beziehungsweise die Asynchronität der *Extension API*, wurden die Änderungen nicht auf die richtigen Datenstände transformiert und es resultierten jeweils andere Ergebnisse bei Alice und Bob. Aufbauend auf dem Wissen hinter der Funktionsweise von Saros, den Verhaltensweisen von VSCode und dem evaluierten Problem in der bisherigen Architektur, werden im nächsten Kapitel einige Lösungsansätze hierfür vorgestellt und diskutiert.

7 Aufbau eines Konzeptes für eine alternative Saros-Architektur

Anschließend an die Analyse des Problems inklusive der theoretischen Grundlage dafür, werden in diesem Kapitel einige entwickelte Konzepte und Ansätze betrachtet, um eine zielführende und vor allem auch schnelle Lösung für das angesprochene Problem zu finden. Dabei wird besonderen Wert auf eine schnelle Lösung gelegt, damit anschließend trotzdem noch mit der vorherigen Analyse des Prototyps fortgefahren werden kann. Für die möglichen Lösungen wurden dabei im Gespräch mit Victor Brekenfeld grundlegende Anforderungen definiert, in deren Rahmen sich neu entwickelte Konzepte bewegen sollen:

1. Der eben auch anhand des Beispiels erklärte Problemfall muss gelöst werden

2. Der SLS soll erhalten bleiben, um den Saros-VSCode Prototypen mit vergleichsweise wenig Programmieraufwand auch auf andere Editoren oder IDE´s zu unterstützen

[16] siehe `https://github.com/saros-project/saros/pull/1111`

3. Das aktuelle Verhalten von Saros, beziehungsweise die Nutzer*innenerfahrung, soll nachvollziehbar bleiben und mit anderen Implementierungen von Saros vergleichbar sein

Auf Basis dieser Rahmenbedingungen können nun Konzepte und Ideen gesammelt und bewertet werden. Eine naheliegende und einfache Idee ist es, die fehlende Logik einfach in den Saros-VSCode Klienten zu integrieren. Diese Möglichkeit scheidet jedoch aus, da es einerseits den Entwicklungsaufwand auch für alle folgenden Klienten über das LSP signifikant erhöht und außerdem die Java-Implementierungen, wie auch in der Bachelorarbeit beschrieben [6, Vgl. S. 19-20], nicht ohne weiteres mit dem Typescript von VSCode kompatibel sind. Außerdem wäre damit die Problematik der Asynchronität weiterhin nicht gelöst, da die Änderungen trotzdem beim Erreichen des Saros-VSCode Klienten bereits veraltet sein können und dieser Fall weiterhin nicht behandelt wäre. Ein möglicher Ansatz ist jedoch das direkte Entfernen von Änderungen in VSCode. Von Nutzer*innen getätigte Änderungen werden damit erst sichtbar, wenn sie vom SLS verarbeitet und bestätigt wurden. Damit soll erreicht werden, dass alle Änderungen in VSCode vom SLS gemeldet werden, wodurch dieser dafür sorgen kann, dass auch bei asynchron gemeldeten Änderungen immer vorher eine mögliche Konfliktauflösung durchgeführt wurde. Eine Abwandlung hierzu wäre das direkte Abfangen von Tastatureingaben, womit das gleiche Ziel erreicht werden soll. Eine Implementierungsmöglichkeit hierfür wäre, dass für jede Taste ein Shortcut definiert wird und erst nach der Verarbeitung durch den SLS die Änderung an den Saros-VSCode Klienten weitergegeben wird, da Events für einzelne Tastendrücke in der *Extension API* von VSCode nicht definiert sind und somit auch standardmäßig nicht abgefangen werden können. Ein komplett anderer Ansatz wiederum wäre es, das aktuelle Konzept erstmal grundsätzlich zu verwenden und im Problemfall die Berechnung der Konfliktlösung, welche theoretisch im Saros-VSCode Klient stattfinden müsste, in den SLS auszulagern. Ein einfaches Konzept ist es auch, während der Verarbeitung einer Änderung durch den SLS regelmäßig den Datenstand vom Saros-VSCode Klienten mit dem Datenstand des SLS zu überschreiben und somit eine Blockierung zu simulieren. Diese und auch weitere Ideen wurden gesammelt und evaluiert. Nicht alle Konzepte passen jedoch in den zeitlichen Rahmen dieser Arbeit und können daher auch nicht eingehender analysiert werden. Aus diesem Grund wurden die zwei vielversprechendsten Ansätze ausgewählt, im Detail betrachtet und teilweise für prototypische Tests implementiert.

7.1 Konzept 1: Das direkte Entfernen von Änderungen

Das erste analysierte Konzept basiert auf der Idee, von Nutzer*innen getätigte Änderungen an einem Dokument nach Meldung der *Extension API* direkt wieder vom Saros-VSCode Klienten löschen zu lassen. Von der Funktionalität her werden die Änderungen dabei rückgängig gemacht. Diese Idee folgt dabei dem Ansatz, dass der in Kapitel 6 beschriebene Problemfall nicht eintritt, wenn es zu keinem Zeitpunkt eine noch nicht registrierte Änderung beim SLS geben kann. Alle im Saros-VSCode Klienten sichtbaren Änderungen, sollen dabei ausschließlich durch den SLS gesteuert und angestoßen werden und wenn die Saros-VSCode Nutzer*innen des Saros-VSCode Klienten dabei eine Änderung selbstständig machen, soll diese erst sichtbar

werden, wenn sie durch den SLS bestätigt und zurück an den Klienten gesendet wurde. Der zusätzliche Programmieraufwand für diesen Ansatz im Saros-VSCode Klienten sollte dabei minimal ausfallen, da die grundsätzlichen Befehle zum Einfügen, Ändern und Löschen bereits im Saros-VSCode Klient implementiert sind und nur um eine Funktion ergänzt werden müssen, um alle Änderungen der Saros-VSCode Nutzer*innen direkt wieder rückgängig zu machen. Als einziger möglicher Kritikpunkt für diesen Ansatz wurde dabei eine mögliche negative Auswirkung auf die Nutzer*innenerfahrung identifiziert, indem das Zurücksetzen von getätigten Änderung der Nutzer*innen nicht in akzeptabler Reaktionszeit durchgeführt werden kann und bei den Nutzer*innen dabei Artefakte auftreten. Um die angesprochene Reaktionszeit des Prototyps ermitteln zu können, wurde dieser Ansatz prototypisch und mit minimalen Funktionen in eine neue VSCode-Erweiterung implementiert. Der in Anhang A.1 ersichtliche Prototyp registriert sich hierbei für die entsprechenden Events zur Benachrichtigung bei Änderungen bei der *Extension API* und führt anschließend bei entsprechender Benachrichtigung die Zurücksetzung der Änderung durch.

Da die *Extension API* asynchron arbeitet und für jede getätigte Änderung der Nutzer*innen ein neues Event erstellt wird, werden eintreffende Änderung beim Saros-VSCode Klienten vor der Abarbeitung in einer Warteschlange hinterlegt. Damit soll verhindert werden, dass eintreffende Änderungen von Seiten des/der Saros-VSCode Nutzer*in während der Abarbeitung einer vorherigen Änderung verloren gehen. Zusätzlich muss sich der Prototyp beim Öffnen einer Datei in einem Projektorder diese in einem Array sichern, damit beim Eintreffen einer Änderungsbenachrichtigung der alte Stand, also der Dateistand vor der Änderung, noch nachvollzogen werden kann. Die Kopie des Dateistandes wird anschließend nach der erneuten Durchführung der Änderung auf Anweisung des SLS ebenfalls aktualisiert. Der komplette Ablauf der Verarbeitung einer Änderung sieht dabei wie folgt aus:

1. Beim Öffnen einer Datei durch den/die Nutzer*in speichert der Saros-VSCode Klient eine Kopie des Inhalts

2. Wenn der/die Nutzer*in eine Änderung am Inhalt durchführt, wird der Saros-VSCode Klient über Events informiert

3. Die Änderung wird der Warteschlange von Änderungen hinzugefügt

4. Bei der Verarbeitung der Änderung wird zuerst die durchgeführte Änderung des/der Nutzer*in wieder entfernt, der vorherige Inhalt an der entsprechenden Stelle aus der Kopie des Datenstands ermittelt und anschließend wieder an seiner ursprünglichen Position im Dokument eingefügt

5. Weiterführend wird vom Saros-VSCode Klient die Änderung über LSP an den SLS gesendet und dort verarbeitet

6. Nach erfolgreicher Verarbeitung sendet der SLS die Änderung zurück an den Saros-VSCode Klient

7. Der Saros-VSCode Klient führt abschließend die gemeldete Änderung durch

Um die entsprechenden Funktionen im Saros-VSCode Klienten implementieren zu können, muss vorerst verstanden werden, wie eine Änderungsbenachrichtigung in Visual Studio Code aufgebaut ist. Die wichtigsten Informationen, welche eine von der *Extension API* gemeldete Änderung an den Saros-VSCode Klienten enthält, ist der Startpunkt, der Endpunkt und der geänderte Text. Start- und Endpunkt werden dabei aus einer Kombination aus dem Index der Zeile und dem Index des Zeichens an entsprechender Stelle repräsentiert. Wenn also in einem Textdokument in Zeile 3 hinter dem zweiten Buchstaben (zum Beispiel bei Text *Hallo* in Zeile 3 hinter dem *a*) der Buchstabe *e* in einen bestehenden Text eingefügt wird, sieht die von der *Extension API* gemeldete Änderung wie in Abbildung 9 aus:

```
{
  "range": {
    "start": {
      "line":2,
      "character":2

    },
    "end": {
      "line":2,
      "character":2
    }
  },
  "text": "e"
}
```

Abbildung 9: Struktur einer gemeldeten Änderung in VSCode

Dabei ist zu beachten, dass wenn in VSCode Text einfach nur eingefügt wird, ist der Start- und der Endpunkt hierbei derselbe. Wenn in dem eben beschriebenen Beispiel der dritte Buchstabe, also das erste *l* durch ein *e* ersetzt worden wäre, würde sich in der gemeldeten Änderung der Endpunkt der Änderung auf Zeichen 3 verschieben. Wenn das *l* einfach nur gelöscht wird, würde sich sowohl das Zeichen des Endpunktes auf 3, als auch der Text auf einen leeren String verändern, da durch leere Strings bei einer Änderungsbenachrichtigung durch die *Extension API* die Löschung von Inhalten signalisiert wird. Komplizierter wird die Änderung, wenn ein Text durch einen ungleich langen Text ersetzt wird. Wenn zum Beispiel in *Hallo* das erste *l* durch ein *eo* ersetzt wird, sieht die Änderung wie beim normalen Ersetzen

aus, nur dass die Länge des Strings hinter dem Schlüssel *text* nicht dem Abstand des Start- und Endpunktes entspricht. Der komplizierteste Fall stellt sich dar, wenn ein über mehrere Zeilen markierter Text durch einen kürzeren oder längeren Text, auch mit mehr oder weniger Zeilen, ersetzt wird. Der geänderte Text ist dabei noch relativ einfach über unterschiedliche Zeilenangaben im Start- und Endpunkt der gemeldeten Änderung auslesbar. Da der geänderte Text jedoch als ein einzelner String gemeldet wird, müssen die Änderungen pro Zeile erst durch das Teilen des Strings an den Windows-Zeilenenden $\backslash r \backslash n$ ermittelt werden. Ähnlich kompliziert ist dabei das Berechnen des ursprünglichen Textes vor der Änderung aus dem gespeicherten, vorherigem Datenstand. Aufgrund der Komplexität einzelner Fälle, wurde bei der Implementierung des Prototyps einerseits zwischen einzeiligen und mehrzeiligen und andererseits zwischen den verschiedenen Fällen Einfügen, Entfernen und Ersetzen jeweils differenziert. Als zusätzlicher Sonderfall muss außerdem das Entfernen und Hinzufügen einzelner Zeilenumbrüche extra behandelt werden.

Ziel dieses Prototyps ist es dabei, zu evaluieren, ob die Idee einerseits grundsätzlich in VSCode technisch umsetzbar ist und andererseits, ob eine durch ein wenig nachvollziehbares Verhalten des Saros-VSCode Klienten generierte, schlechte Nutzer*innenerfahrung entsteht. Da der SLS in einem separaten Prozess läuft und an keine Regularien durch VSCode gebunden ist, kann vorerst davon ausgegangen werden, dass die nötigen Funktionen auf Seiten des SLS implementierbar sind. Aufgrund dieser Annahme wurde sich dafür entschieden, die nötigen Implementierungen im SLS für eine erste Machbarkeitsstudie vorerst nicht vorzunehmen und stattdessen nur einen SLS zu simulieren. Diese Entscheidung hat dabei zusätzlich den Vorteil, mit weniger Zeitaufwand trotzdem ein valides Ergebnis für eine mögliche Nutzer*innenerfahrung gewinnen zu können. So kann auch mit einem kleinen Prototyp bereits die Reaktionszeit gemessen werden. Die Simulation des SLS nimmt dabei Änderung entgegen, verzögert den Ablauf um einen bestimmten Zeitraum und gibt anschließend die laufende Berechnung der aktuellen Änderung frei. Zu Beginn wurde dabei, durch Schätzungen der Antwortzeit des SLS, ein Zeitraum für die Verzögerungen von 20 Millisekunden gewählt. Anschließend wird die neue Änderung zurück an den Saros-VSCode Klienten gesendet und von diesem dann mit der simulierten Bestätigung des SLS durchgeführt. Weiterhin bei der Implementierung zu beachten ist, dass jedes Zurücksetzen einer Änderung durch den Saros-VSCode Klienten oder vom SLS eingesteuerte Aufträge wiederum Änderungen sind, welche ein Event in der *Extension API* auslösen. Diese Fälle wurden ebenfalls in dem Prototyp behandelt, indem die entsprechenden Events erwartet und anschließend ignoriert wurden.

Zum Abschluss der ersten Implementierungen entstand dabei ein Prototyp, welcher die beschriebenen Änderungen implementiert und den zu Beginn dieses Konzeptes beschriebenen Ablauf vollständig unterstützt. Ein entsprechendes Aktivitätsdiagramm ist in Abbildung 10 dargestellt. Somit kann anschließend getestet werden, wie gut die Reaktionszeit der *Extension API* und des Saros-VSCode Klienten wirklich ist und wie die vom Klienten vorgenommenen Änderungen im Nutzer*innenoberfläche der Anwendung aussehen. Getestet wurde dabei zuerst mit einem Szenario, in dem ein/eine Nutzer*in eine Änderung in Form eines einfachen Einfügens eines einzelnen Buchstabens, ohne Ersetzen eines anderen Textes durchgeführt hat. Die entsprechenden Datenerhebungen zu den Laufzeiten der prototypischen VSCode-Erweiterung,

wurden durch Zeitmessungen im Code realisiert. Das Hinzufügen der Änderung zur Warteschlange und das Starten des Berechnungsprozesses für die Zurücksetzung der Änderung, wurde in unter einer Millisekunde durchgeführt. Problematisch ist jedoch, dass bis zum Entfernen der Änderung durch die Funktionen im Klienten im Durchschnitt 2-3 Millisekunden vergehen. Dies ist dabei insofern problematisch, dass die Änderung sichtbar für die Nutzer*innen in der Oberfläche erscheint und erst dann wieder gelöscht wird. Das Ziel, die Änderungen ohne sichtbare Benachrichtigung schnell genug entfernen zu können, konnte daher nicht erreicht werden. Mit der Simulation des SLS fügt der Prototyp die ursprüngliche Änderung nach 23 Millisekunden wieder ein und die Nutzer*innen sehen die von Ihnen erstellte Änderung.

Zusätzlich dazu, dass erstellte Änderungen nur mit Verzögerung wieder entfernt werden können, hat sich durch die Reaktionszeit des SLS ein weiteres Problem ergeben. Bei einer eingestellten Verzögerung von 20 Millisekunden, flackern die Änderungen zwar kurz durch das schnelle Einfügen der Änderung vom simulierten SLS auf, jedoch stört dieses Verhalten zu Beginn nicht weiter. Tests mit höheren Latenzzeiten, beispielsweise 100 Millisekunden, welche aufgrund einer sehr hohen Auslastung des SLS durch viele Änderungen des Saros-VSCode Klienten und generell vielen anderen Klienten theoretisch auftreten könnten, zeigen ein anderes Problem dieser Architektur. Da getätigte Änderungen bei Verzögerungen für einen Moment entfernt bleiben, führt schnelles Schreiben beispielsweise von einem längeren Wort zu Fehlern bei den gemeldeten Änderungen. Wenn ein Buchstabe geschrieben wird, während der vorherige Buchstabe gerade durch den neuen Prototyp entfernt und bearbeitet wurde, wird der aktuelle Buchstabe an der Position des eben geschriebenen Buchstaben gemeldet und nicht an seiner eigentlichen Position ein Zeichen weiter. Die falschen Indizes führen dann weitergehend dazu, dass sehr schnell geschriebene Buchstaben auch vertauscht in der Oberfläche der Nutzer*innen eingefügt und angezeigt werden können. Dieses Verhalten kann dabei auch in Tests mit einer niedrigeren Verzögerung, von zum Beispiel 10 Millisekunden, im simulierten SLS reproduziert werden.

Da diese Vertauschung nur sehr schwer zu unterscheiden ist, für die Lösung des Problems viel mehr Logik als bisher bekannt in den Saros-VSCode Klienten ausgelagert werden muss und weil die Reaktionszeiten der prototypischen VSCode-Erweiterung beim Entfernen von Änderungen nicht zufriedenstellend sind, wurde dieses Konzept wieder verworfen.

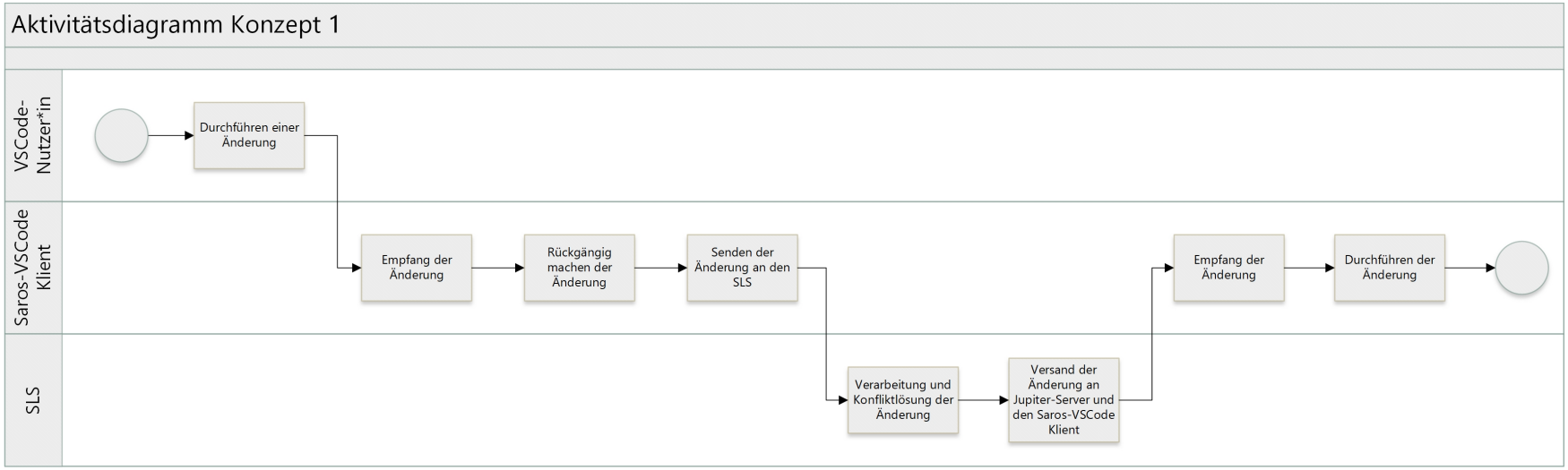

Abbildung 10: Aktivitätsdiagramm Konzept 1

7.2 Konzept 2: Erweiterung des SLS

Das zweite analysierte Konzept basiert auf der Idee, die Konfliktauflösung zwischen den Änderungen vom Saros-VSCode Klient und vom Jupiter-Server für den Saros-VSCode Klienten vollständig im SLS zu berechnen. Die Architektur des bestehenden Prototyps wird hierfür um ein zweites Dateisystem im SLS erweitert. Das neue VSCode Filesystem (im Folgenden VSCF genannt) repräsentiert dabei den Datenstand des Saros-VSCode Klienten, während das bisherige Dateisystem im SLS nur noch ausschließlich den Datenstand des Jupiter-Server repräsentiert. Im neuen Konzept wird es daher Jupiter-Server Filesystem (im Folgenden auch JSF) genannt. Die Idee hinter der Verwendung von zwei Dateisystemen im SLS ist, dass dieser einerseits besser mögliche Konflikte erkennen kann, da er nun auch den vorherrschenden Datenstand im Saros-VSCode Klienten kennt. Andererseits kann der SLS damit Änderungen berechnen, welche auf dem Stand des Saros-VSCode Klienten und nicht einer Mischung des Datenstandes vom Saros-VSCode Klient und dem des Jupiter-Server basieren. Damit die beidem Dateisysteme auch repräsentativ für den Saros-VSCode Klienten und den Jupiter-Server stehen können, ist es erforderlich, dass sie auch nur ausschließlich durch gemeldete Änderungen von der entsprechenden Stelle durch den SLS aktualisiert werden. Das VSCode Filesystem darf dabei nur durch eine Meldung vom Saros-VSCode Klienten und das Jupiter-Server Filesystem nur durch eine Meldung vom Jupiter-Server aktualisiert werden. Um eine entsprechende Aktualität der Dateisysteme gewährleisten zu können, sieht dieses Konzept daher vor, dass der Saros-VSCode Klient und der Jupiter-Server immer die entsprechende Änderung als Bestätigungsnachricht nach der Verarbeitung an den SLS zurücksenden.

Um das Verhalten der Architektur besser zu verstehen, werden im Folgenden zwei Anwendungsfälle betrachtet: beim SLS eintreffende Änderungen vom Jupiter-Server und beim SLS eintreffende Änderungen vom Saros-VSCode Klienten. Wenn eine Änderung vom Jupiter-Server beim SLS eintrifft, muss der SLS zuerst prüfen, ob die entsprechende Stelle im JSF und VSCF identisch ist. Damit soll sichergestellt werden, dass der Jupiter-Server und der Saros-VSCode Klient zum Zeitpunkt der Verarbeitung der Änderung im SLS denselben Datenstand besitzen. Ist dies der Fall, besitzt der Saros-VSCode Klient nach Wissen des SLS somit den gleichen Datenstand wie der Jupiter-Server zum Zeitpunkt der Erstellung der Änderung. Der SLS trifft diese Entscheidung dabei basierend auf dem Wissen, dass ansonsten das JSF bereits eine vorherige Änderung empfangen hätte und anschließend bei einer fehlenden Verarbeitung der vorherigen Änderung durch den Saros-VSCode Klient, das VSCF und das JSF nicht identisch gewesen wären. Aus diesem Grund wird die Änderung des Jupiter-Servers vom SLS am JSF angewandt und mit LSP an den Saros-VSCode Klienten geschickt. Dieser bestätigt dem SLS abschließend die Durchführung der gemeldeten Änderung, worauf der SLS die Änderung auch im VSCF einträgt.

Komplizierter wird der Ablauf, wenn beim Abgleich des VSCF mit dem JSF bei einer Änderung vom Jupiter-Server im SLS die beidem Dateisysteme nicht identisch sind. Hierbei wird die neue Änderung zuerst wieder am JSF angewandt. Anschließend berechnet der SLS die Differenz zwischen dem JSF inklusive der neuen Änderung und dem VSCF. Diese Differenz wird anschließend als Änderung mit LSP an den Saros-VSCode Klienten gesendet. Ziel hinter diesem Verfahren ist, den Saros-VSCode

Klienten wieder auf einen aktuellen Stand zu bringen. Durch die Berechnung der Differenz wird dabei nicht nur die neue Änderung, sondern auch noch nicht dem Saros-VSCode Klienten gemeldete Änderungen als vereinfachte Konfliktlösung direkt in die neue Änderung beim Saros-VSCode Klienten mit einbezogen. Der Saros-VSCode Klient verarbeitet daraufhin die vom SLS gemeldete Änderung und schickt abschließend wieder eine Bestätigungsnachricht an den SLS, damit dieser die durchgeführten Änderungen auch in das VSCF eintragen kann und alle Datenstände wieder gleich sind.

Wenn eine Änderung vom Saros-VSCode Klienten beim SLS eintrifft, trägt dieser die Änderung direkt in das VSCF ein. Anschließend wird die Änderung an den Jupiter-Server geschickt. Nach Bestätigung der erfolgreichen Durchführung vom Jupiter-Server an den SLS, trägt dieser die Änderung abschließend noch in das JSF ein. Auf einen Abgleich des VSCF mit dem JSF wird dabei verzichtet, da spätestens durch die nächste Änderung vom Jupiter-Server eventuell auftretende Konfliktfälle behandelt werden.

Bei einer ersten Analyse dieses Konzeptentwurfes wurde dabei ein Problem identifiziert, welches noch nicht betrachtet wurde. Hierbei handelt es sich um die Problematik, dass im SLS aktuell nicht unterschieden werden kann, welche gemeldeten Änderungen durch den/die Nutzer*in der Saros-VSCode Erweiterung bzw. durch den Jupiter-Server initiiert sind und welche Änderungen nur als Bestätigungen versendet werden. Erschwerend kommt, das bereits im vorherigen Kapitel 7.1 angesprochene Verhalten von VSCode hinzu, wodurch auch eine durch den Saros-VSCode Klienten durchgeführte Änderung in VSCode über die *Extension API* wieder ein identisches Event auslöst, als würde ein/eine VSCode Nutzer*in selbst eine Änderung durchführen.

Damit die neue Architektur auch mit diesem identifizierten Problem umgehen kann, wird der SLS um zwei weitere Komponenten erweitert: das VSCode Log und das Jupiter-Server Log (siehe Abbildung 11). Mit der Einführung der Logs wird dabei das Ziel verfolgt, dass der SLS sich alle ausgehenden Änderungen merkt. Dadurch kann der SLS bei eintreffenden Änderungen unterscheiden, ob es sich um eine Änderung initiiert vom Jupiter-Server beziehungsweise vom Saros-VSCode Klienten, oder um eine Bestätigungsänderung handelt, welche vom SLS initiiert wurde. Im VSCode Log speichert der SLS dabei alle Änderungen, welche aktuell zur Bearbeitung an den Saros-VSCode Klienten gesendet wurden und von diesem noch nicht als angewandt bestätigt wurden. Im Jupiter-Server Log speichert der SLS wiederum alle Änderungen, welche aktuell zur Bearbeitung an den Jupiter-Server gesendet wurden und von diesem noch nicht als angewandt bestätigt sind. Mit diesen neuen Informationen kann der SLS zusätzlich auch beim Eintreffen, nicht vom SLS initiierten Änderungen erkennen, ob bei der Anwendung der Änderung Konflikte auftreten werden, weil zu diesem Zeitpunkt noch Änderungen an den Jupiter-Server oder den Saros-VSCode Klienten versendet wurden, welche noch nicht als bestätigt zurückgemeldet und somit abschließend im SLS angewandt sind.

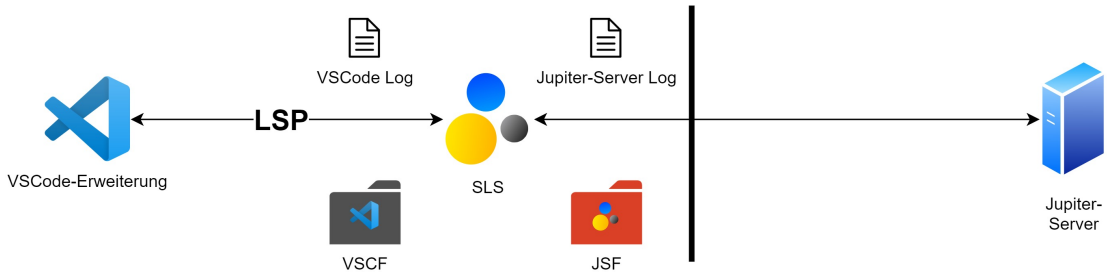

Abbildung 11: Übersicht über die Architektur des zweiten Konzepts

Mit der Erweiterung der Architektur um die Logs, müssen einige Randfälle aus einer anderen Perspektive betrachtet werden. Ein Fall sind hierbei eintreffende Änderungen vom Jupiter-Server, welche den SLS erreichen, während dieser auf die Bestätigung der Abarbeitung einer zeitlich früheren Änderung vom Saros-VSCode Klienten beim Jupiter-Server wartet. Wenn dieser Fall eintritt, sind die Datenstände des VSCF und JSF unterschiedlich, da die zeitlich frühere Änderung vom Saros-VSCode Klienten bereits im VSCF eingetragen ist und im JSF aufgrund der noch nicht beim SLS eingetroffenen Bestätigung vom Jupiter-Server fehlt. Ein ähnlicher, zu betrachtender Fall ist, dass beim SLS eine Änderung vom Saros-VSCode Klienten eintrifft, während dieser auf die Bestätigung der Abarbeitung einer zeitlich früheren Änderung vom Jupiter-Server beim Saros-VSCode Klienten wartet.

Der einfachste Ansatz zur Behandlung dieser Randfälle wäre es, die konflikterzeugende Änderung einfach zu verwerfen. Dies würde jedoch zu einer negativen Erfahrung der Nutzer*innen führen und den gesamten Umbau der Architektur hinfällig machen. Aus diesem Grund wurde zu Behandlung ein Ansatz gewählt, bei dem der langfristige Verlust von Änderungen vermieden und lange Wartezeiten bei der Synchronisation der Datenstände verhindert werden sollen, um keine negative Erfahrung der Nutzer*innen zu erzeugen. Die Idee dabei ist, beim Auftreten von Konflikten die einzelnen Datenstände ohne Konfliktauflösung zu überschreiben und anschließend das VSCode Log und das Jupiter-Server Log zu verwenden, um dabei verloren gegangene Änderungen wiederherzustellen. Vor der Anwendung einer konfliktbehafteten Änderung würde das VSCF und das JSF dabei in einem ersten Schritt auf einen konfliktfreien Stand gebracht werden. Um dies zu erreichen, müssen alle aktuell noch nicht bestätigten Änderungen je nach Quelle der eingehenden Änderung rückgängig gemacht werden, bevor die neue Änderung angewandt werden kann. Wenn alle Änderungen rückgängig gemacht sind, wird im SLS dabei simuliert, dass die Datenstände des VSCF und des JSF und somit auch des Saros-VSCode Klienten identisch sind.

Damit die bereits im SLS angewandten Änderungen, welche aktuell rückgängig gemacht sind, dabei nicht verloren gehen, werden die Änderungen aus dem Log, von dessen Seite noch eine Bestätigung erwartet wird, entfernt. Wenn dann die Bestätigung über die Durchführung der zeitlich früheren Änderung beispielsweise vom Jupiter-Server den SLS erreicht, wird die Änderung nicht als Bestätigung, sondern als neue Änderung behandelt. Somit sind die zeitlich früheren, konfliktbehafteten Änderungen langfristig nicht verloren. Hierbei ist jedoch erwartbar, dass die Implementierung im SLS für diese Architektur sehr kompliziert werden wird. Der SLS muss

dabei neben den normalen Änderungen jetzt auch solche behandeln, welche vom SLS initiiert sind, jedoch als vom Jupiter-Server oder vom Saros-VSCode Klienten erstellte Änderungen beim SLS eintreffen. Da hierbei eine erhöhte Komplexität im SLS kein Ausschlusskriterium für ein Konzept darstellt, wurde für diese Architektur noch der explizite Ablauf anhand von zwei Beispielen evaluiert: von VSCode-Nutzer*innen und vom Jupiter-Server ausgehende Änderungen an den SLS. Für eine bessere Übersicht der einzelnen Prozessschritte sind in Abbildung 12 und 13 Aktivitätsdiagramme zu den beiden Fällen dargestellt.

Im ersten Beispiel wird eine beim SLS eintreffende Änderung vom Jupiter-Server betrachtet. Die Änderung trifft dabei initial beim SLS ein, wobei dieser in einem ersten Schritt prüft, ob die entsprechende Änderung im Jupiter-Server Log eingetragen ist. Wenn die eingetroffene Änderung im Jupiter-Server Log eingetragen ist, handelt es sich um die Bestätigung der Durchführung einer vom SLS initiierten Änderung. Aus diesem Grund wird die Änderung abschließend am JSF angewandt und aus dem Jupiter-Server Log entfernt. Ist die Änderung jedoch nicht im Jupiter-Server Log eingetragen, handelt es sich entweder um eine vom Jupiter-Server initiierte Änderung, oder um eine Änderung, welche zwar vom SLS initiiert wurde, jedoch erneut angewandt werden soll.

Zur weiteren Verarbeitung muss daher im nächsten Schritt geprüft werden, ob das JSF und das VSCF identisch sind. Wenn beide Dateisysteme identisch sind, basiert die Änderung vom Jupiter-Server auf dem gleichen Stand wie der SLS und der Saros-VSCode Klient und kann daher weiterverarbeitet werden. Dabei wird die Änderung am JSF angewandt, in das VSCode Log eingetragen und über LSP an den Saros-VSCode Klienten übermittelt. Dieser wendet die Änderung anschließend über die *Extension API* in VSCode an und dabei entsteht ein neues Änderungsevent, welches als Bestätigungsnachricht zurück an den SLS gesendet wird. Im SLS angekommen, wird die Änderung abschließend noch am VSCF durchgeführt, damit die Datenstände vom VSCF und JSF wieder identisch sind.

Wenn das JSF und das VSCF jedoch nicht identisch sind, basieren die eingetroffenen Änderungen vom Jupiter-Server nicht auf dem gleichen Datenstand wie im SLS. Daher muss initial geprüft werden, ob aktuell noch andere Änderungen im Jupiter-Server Log vorhanden sind, welche noch nicht als durchgeführt vom Jupiter-Server bestätigt wurden. Sind noch weitere Änderungen vorhanden, wird es sich bei den Unterschieden zwischen dem JSF und VSCF um beim SLS eingetroffene Änderungen vom Saros-VSCode Klienten handeln, welche bereits im VSCF, aufgrund der fehlenden Bestätigung durch den Jupiter-Server allerdings noch nicht im JSF angewandt sind. Daher werden alle Einträge aus dem Jupiter-Server Log gelöscht, damit diese nicht verloren gehen und bei Ankunft der Bestätigung wieder angewandt werden. Anschließend wird die Änderung am JSF angewandt, in das VSCode Log eingetragen und abschließend der Datenstand des Saros-VSCode Klienten mit dem des JSF überschrieben. Die bereits vom SLS verarbeiteten Änderungen des Saros-VSCode Klienten sind damit nicht verloren.

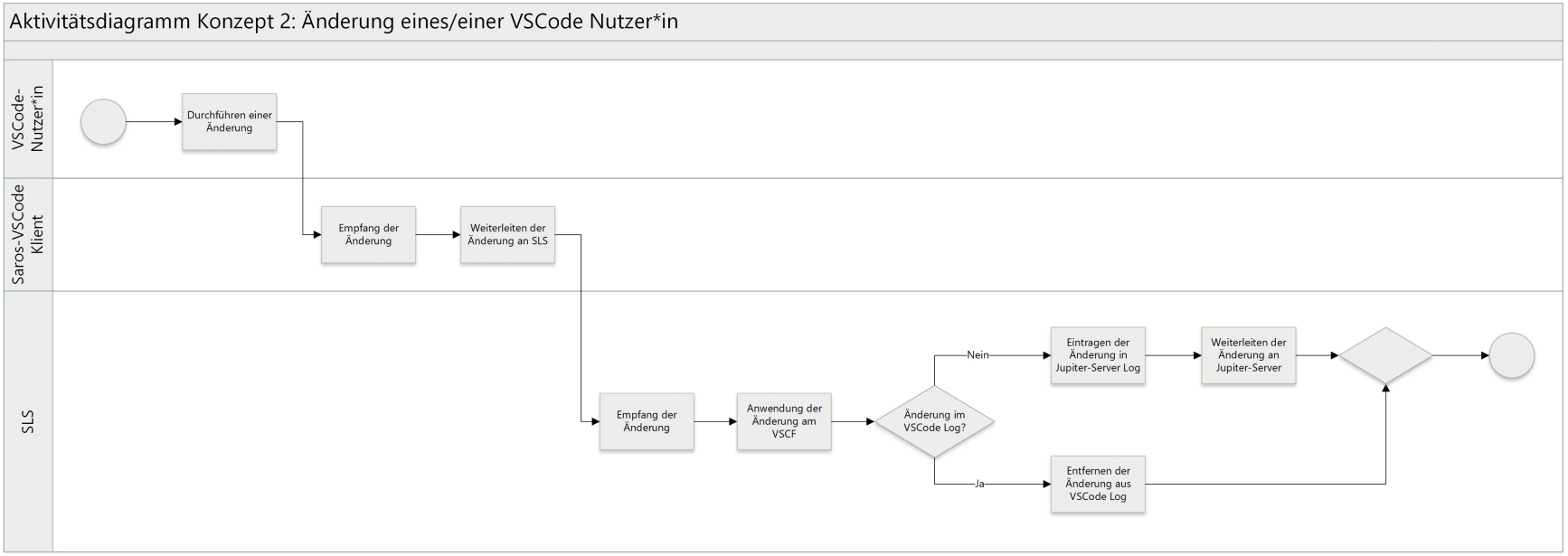

Abbildung 12: Aktivitätsdiagramm Konzept 2: Änderung eines/einer VSCode Nutzer*in

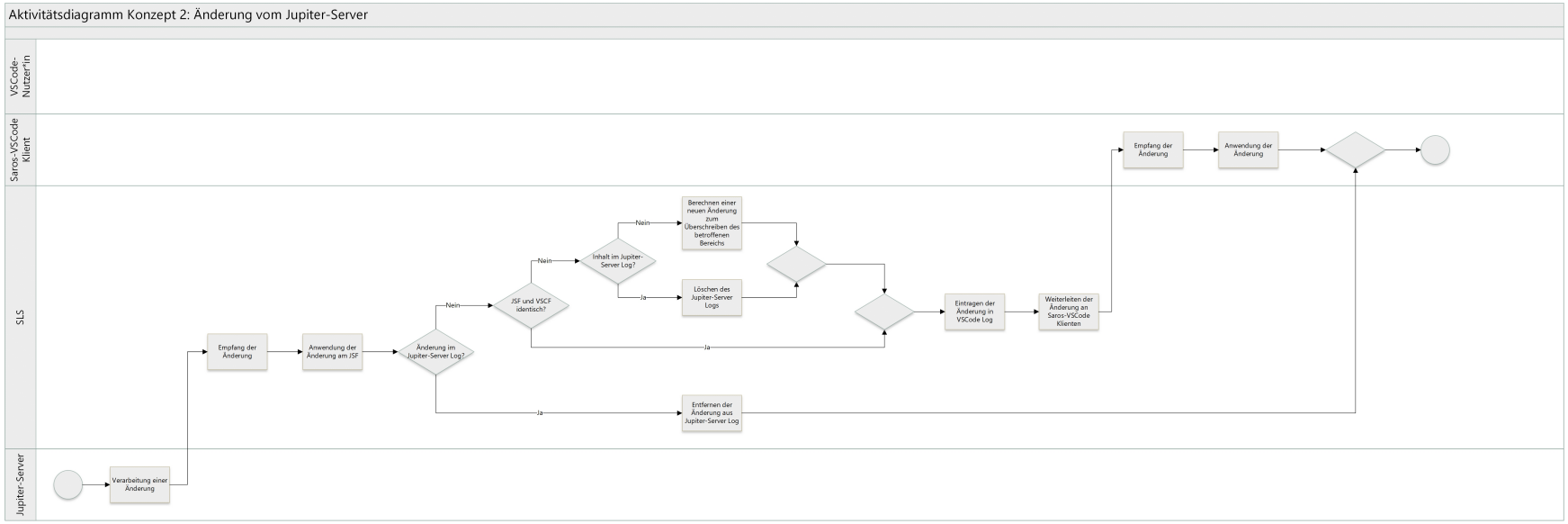

Abbildung 13: Aktivitätsdiagramm Konzept 2: Änderung vom Jupiter-Server

Wenn in seltenen Fällen bei der zweiten Prüfung des Jupiter-Server Logs keine anderen Änderungen eingetragen sind, wird in dem Konzept von einem Fehlerfall ausgegangen, da es sich entweder um eine noch nicht im SLS verarbeitete Änderung des Saros-VSCode Klienten oder um einen unbekannten Grund handelt. In beiden Fällen hat das Konzept zur Erhaltung der Konsistenz nur die Möglichkeit, die beim SLS eingetroffene Änderung im JSF anzuwenden und anschließend den Saros-VSCode Klienten und somit auch das VSCF mit dem Stand des JSF zu überschreiben.

Im zweiten Beispiel wird eine beim SLS eintreffende Änderung vom Saros-VSCode Klienten betrachtet. Der Ablauf in diesem Beispiel ist dabei einfacher, da auf einen Abgleich des VSCF mit dem JSF verzichtet wird. Die Änderung trifft dabei initial beim SLS ein, wobei dieser in einem ersten Schritt prüft, ob die entsprechende Änderung im VSCode Log eingetragen ist. Wenn für die Änderung ein Eintrag im VSCode Log vorhanden ist, handelt es sich um die Bestätigung einer vom SLS initiierten Änderung und sie wird abschließend am VSCF angewandt. Der Eintrag im VSCode Log kann damit entfernt werden und die Verarbeitung ist damit beendet. Wenn für die Änderung jedoch kein Eintrag im VSCode Log vorhanden ist, handelt es sich um eine von VSCode Nutzer*innen initiierte Änderung. Diese muss anschließend am VSCF angewandt werden, bevor die Änderung abschließend in das Jupiter-Server Log eingetragen und an den Jupiter-Server weitergeleitet werden kann.

Anschließend an die Betrachtung der Beispiele von der Architektur mit dem VSCode und dem Jupiter-Server Log, wurde das Konzept inklusive der entsprechenden Saros-Umgebung noch einmal neu analysiert und diskutiert. Dabei wurde auch besonders auf mögliche Probleme bei einer Implementierung geachtet. Parallel dazu, ist das Konzept im Rahmen des Zwischenvortrags zu dieser Arbeit, unter anderem einigen Mitgliedern der Projektgruppe von Saros, präsentiert worden. Sowohl während der Analyse als auch der Vorstellung wurden einige Schwachstellen identifiziert, welche im Folgenden vorgestellt und betrachtet werden.

Als eine erste Schwachstelle stellt sich dabei die unglaubliche Komplexität heraus, welche mit der Zeit durch die Erweiterungen des Konzeptes um neue Komponenten zur Behandlung, der zu diesem Zeitpunkt noch unbekannten Problemfälle, entstanden sind. Dabei ist das erneute Anwenden von Änderungen nach der Entfernung des entsprechenden Eintrages aus dem Log als besonders kritisch zu betrachten und nicht mit dem eigentlichen Konzept von Jupiter und Saros vereinbar. Diese Schwachstelle resultiert dabei aus der konzeptionellen Entscheidung, in dem neuen Konzept einen eigenen Mechanismus zu implementieren, anstatt zu versuchen, die bestehende Implementierung von OT und speziell des dOPT-Algorithmus weiter zu verwenden. Initial war die Idee des Konzeptes dabei, die Konfliktlösung für dem Saros-VSCode Klienten nur in den SLS auszulagern. Durch die ständigen Erweiterungen und Optimierungen entwickelte sich das Konzept dabei zu einem System, in welchem Konflikte zwischen Änderungen so gut wie möglich vermieden werden sollen. Sollten dabei jedoch trotzdem Konflikte zwischen den Änderungen auftreten, wird in diesem Konzept dabei auf das Überschreiben und Rückgängigmachen von Änderungen gesetzt. Zusammenfassend betrachtet ist die Architektur dabei auch nur sehr umständlich mit dem Jupiter-Server kombinierbar, da wichtige Komponenten von operational transformation, wie eine Konfliktlösung auf Basis älterer Datenstände, nicht mehr vorhanden sind. Hinzu kommt weiterhin, dass der in Kapitel 6 beschriebene Problemfall mit die-

sem Konzept nicht vollständig gelöst werden kann. Der entsprechende Anwendungsfall wird zwar durch die Architektur behandelt, allerdings nur, indem die Änderung von Bob durch die Überschreibung des Datenstandes im JSF mit der Änderung von Alice, verloren geht.

Als eine weitere Schwachstelle stellte sich dabei heraus, dass der Jupiter-Server in seiner aktuellen Implementierung standardmäßig keine Bestätigungen nach der Verarbeitung einer Änderung an den Klienten versendet, welcher die Quelle der Änderung ist[17]. Damit fehlt dem Konzept in der aktuellen Version von Saros eine wichtige Kernkomponente, auf der die gesamte Architektur aufbaut. Um die Architektur trotzdem verwenden zu können, müssten Änderungen am Jupiter-Server vorgenommen werden, was wiederum Auswirkungen auf die Implementierungen von Saros für Eclipse und IntelliJ haben kann. Zusätzlich wurde auch während der Vorstellung angemerkt, dass es unter anderem für eine einfachere Wartung des SLS und der Implementierung von Saros für VSCode von Vorteil ist, immer auf bestehende Konzepte aus der Industrie und Forschung zu setzen, anstatt sich einen neuen Mechanismus zu überlegen. Besonders für den Anwendungsfall der Konsistenzerhaltung über mehrere Datenstände und Dateisysteme wird es viele Konzepte geben, welche dabei besser erforscht und eingehender getestet sind.

Aufgrund der Vielzahl an identifizierten Schwachstellen und Problemen wird dieses Konzept nicht verwendet. Hierfür ist auch die Tatsache, dass die genannten Anforderungen nicht erfüllt werden, weil der beschriebene Problemfall nicht gelöst wird, ein entscheidender Faktor. Auch nach einer Abwägung des Nutzens mit dem damit verbundenen Aufwand einer Implementierung und der Wartung, wird das Konzept wieder verworfen. Das Konzept bietet dabei nur den Vorteil einer eingebauten Behandlung von eventuell auftretenden Konfliktfällen, während hierfür ein immenser Implementierungsaufwand benötigt wird und in einigen Fällen weiterhin Änderungen verloren gehen. Trotzdem konnten mithilfe der Entwicklung dieses Konzeptes viele Rand- und Problemfälle identifiziert werden, welche vor der Entwicklung des Konzeptes noch nicht bekannt waren. Die Vielzahl an gesammelten Erfahrungen kann daher genutzt werden, um auf Basis dessen ein besseres Konzept entwickeln zu können, welches gut in die bestehenden Konzepte von Saros und Jupiter integrierbar ist und dabei alle bis jetzt beschriebene Fälle und Beispiele zufriedenstellend behandelt. Um dabei ein Konzept zu finden, welches auf Ansätzen basiert, die bereits durch Forschung und Wirtschaft studiert und getestet sind, werden im nächsten Kapitel noch einmal Alternativen zu OT betrachtet. Dabei soll evaluiert werden, ob andere Ansätze eventuell besser für die Konfliktlösung zwischen dem SLS und dem Saros-VSCode Klienten geeignet sind, als operational transformation.

8 Recherchen zu neuen Konzepten

In einer vernetzten Welt wie der heutigen, mit teilweise Milliarden Nutzer*innen von Applikationen verteilt über die ganze Welt, gibt es viele Anwendungsfälle und somit auch Forschung und Entwicklung für die Konfliktlösung und Erhaltung der Datenkonsistenz zwischen verschiedenen Datenquellen. Um nach den verworfenen

[17] siehe `https://github.com/saros-project/saros/pull/1111#issuecomment-799384566`

Konzepten ein passendes System für die Verbindung zwischen dem SLS und dem Saros-VSCode Klienten zu finden, muss daher eine allgemeine Analyse möglicher Ansätze und Technologien zur Erhaltung der Datenkonsistenz vorgenommen werden. Anschließend soll aufbauend auf diesem Wissen entschieden werden, welcher Ansatz für die Entwicklung eines neuen Konzeptes verwendet werden soll. Damit soll sichergestellt werden, dass die Entwicklung eines neuen Konzeptes neben der Lösung der Problemstellung aus Kapitel 6 und der Rand- und Fehlerfälle aus Kapitel 7 auch Zukunftssicherheit und Wartbarkeit sichergestellt sind. Begonnen wird dabei mit dem Themenbereich der verteilten Datenbanksysteme.

8.1 Verteilte Datenbanksysteme

Im Allgemeinen gibt es für die Replikation von Daten zwei Hauptgründe: die Erhöhung der Verlässlichkeit des Systems und die Erhöhung der Performance [27, Vgl. S. 356], indem ein/eine Nutzer*in beispielsweise bei globalen Anwendungen immer auf eine Datenreplikation in seiner/ihrer Nähe mit niedriger Latenz zugreifen kann. Grundsätzlich unterschieden wird bei der Replikation dabei zwischen verschiedenen Konsistenzmodellen, wobei ein Konsistenzmodell laut Tanenbaum als „ein Vertrag zwischen Prozessen und dem Datenstore“ [27, Zitat. S. 359] definiert wird. Ein Datenstore ist dabei ein Überbegriff für Daten in zum Beispiel verteilten Datenbanken oder verteilten Dateisystemen [27, Vgl. S. 358]. Bei der datenzentrierten Konsistenz wird im Allgemeinen garantiert, dass der Datenstore über die einzelnen Replikas konsistent ist [27, Vgl. S. 375]. Bei der klientenzentrierten Konsistenz wird wiederum garantiert, dass der Zugriff eines einzelnen Klienten auf die Datenbank konsistent ist [27, Vgl. S. 377]. Aufgrund der Architektur von Saros wurde sich bei den Recherchen auf die datenzentrierte Konsistenz beschränkt, welche wiederum in einzelne Konsistenzmodelle unterteilt ist. Aufgrund der Komplexität der Thematik kann im Rahmen dieser Arbeit nicht jedes einzelne Modell erläutert werden[18]. Ein für den Anwendungsfall in dieser Arbeit vielversprechendes Konzept ist eventual consistency, bei dem auch Inkonsistenzen zwischen den Datenstores geduldet werden und davon ausgegangen wird, dass alle Replikas konsistent werden, wenn eine längere Zeit keine zu synchronisierenden Updates durchgeführt werden [27, Vgl. S. 373]. Auf den Saros-VSCode Prototyp bezogen würde dies bedeuten, dass beispielsweise zwischen dem Saros-VSCode Klienten und dem SLS temporäre Inkonsistenzen erlaubt wären, welche sich bei genug Leerlauf wieder auflösen. Auch in Bezug auf die Asynchronität der *Extension API* ist dieser Aspekt interessant. Um ein besseren Überblick erhalten zu können, werden im folgenden daher die verwendeten Technologien zusammengefasst um herauszufinden, welche Möglichkeiten als Konzept für Saros überhaupt in Frage kommen würden.

Traditionelle verteilte Datenbanksysteme nutzten zu Beginn die Eigenschaften der **A**tomizität, **K**onsistenz, **I**solation und **D**auerhaftigkeit (AKID beziehungsweise eher in der englischen Version unter ACID bekannt), welches erstmals 1983 in der Abhandlung „Principles of Transaction-Oriented Database Recovery“ von Theo Härder und Andreas Reuter[19] genannt wurde, um Verlässlichkeit bei verteilten Systemen ga-

[18] für mehr Informationen siehe [27]
[19] siehe [28]

rantieren zu können. Mit höheren Anforderungen an die Skalierbarkeit erkannte Eric A. Brewer im Jahr 2000, dass es in verteilten Systemen unmöglich ist, gleichzeitig die Eigenschaften Konsistenz, Verfügbarkeit und Ausfalltoleranz garantieren zu können, was auch als CAP-Theorem bezeichnet wird [29, Vgl. S. 51]. Als Alternative zu AKID präsentierte Brewer das Designprinzip BASE (**B**asic **A**vailability, **S**oft-state, **E**ventual consistency) [30, Vgl. S. 1], welches typischerweise in NoSQL-Datenbanken zum Einsatz kam. BASE verwirft dabei das pessimistische Zugriffsmodell[20] von AKID und führt zusätzlich erstmals das bereits angesprochene Modell der eventual consistency ein. Im Jahr 2010 stellte Daniel J. Abadi mit der Arbeit „Consistency Tradeoffs in Modern Distributed Database System Design“ dabei fest, dass das bis jetzt verwendete CAP-Theorem nicht mehr zeitgemäß ist. Dabei wird angeführt, dass die Latenz ein wichtiger Faktor in modernen Datenbanksystemen ist, welcher bisher nicht bedacht wurde [31, Vgl. S. 38]. Um das gesamte Potential des neuen Faktors darstellen zu können, wird CAP dabei in PACELC umgeschrieben. In einem Daten replizierendem System muss sich das System bei Feststellung eines Ausfalls (**P**artition tolerance) entweder zwischen Verfügbarkeit (**A**vailability) oder Konsistenz (**C**onsistency) entscheiden. In allen anderen Fällen (**E**lse) muss das System zwischen Latenz (**L**atency) oder Konsistenz (**C**onsistency) wählen [31, Vgl. S. 41]. Dabei ergeben sich aus dem Theorem die Entscheidungen P+A, P+C, E+L und E+C, wobei sich jedes System für ein Modell bei einem Ausfall und ein Modell für die restliche Zeit entscheidet. Die Entscheidung für eine mögliche Einordnung für die Verbindung zwischen dem Saros-VSCode Klienten und dem SLS in das aktuellste Modell ist dabei nicht ganz einfach. Der Hauptgrund hierfür ist, dass die Architektur mit dem SLS als Hauptziel weder eine bessere Performance des Prototyps, noch einer höheren Verfügbarkeit der Daten zu Grunde liegt. Wie bereits erwähnt, wurde die Architektur aus der Intention eines geringeren Implementierungsaufwands und einer möglichen Abdeckung mehrerer IDEs und Editoren gewählt. Dazu kommt, dass weder datenzentrierte noch klientenzentrierte Konsistenzmodelle gut ohne Anpassung in die Saros-VSCode Architektur passen, da aufgrund der Funktionsweise von VSCode nicht nur die Konsistenzerhaltung der Datenstände, sondern auch die konsistente Verarbeitung von Änderungen im Vordergrund steht. Auf Basis der bisherigen Implementationen von Saros und der allgemeinen Arbeitsweise von Jupiter, sowie des Ziels des Prototypen und der Struktur mit dem SLS kann man sagen, dass die Erhaltung der Konsistenz immer das höchste Ziel ist und sich im Zweifel auch immer dafür entschieden wird. Aus diesem Grund kann auch der Saros-VSCode Prototyp beim PACELC Theorem als PC/EC Modell eingestuft werden. Im Fehlerfall sowie im Normalbetrieb würde sich der Saros-VSCode Prototyp somit zu Lasten der Latenz und Verfügbarkeit für die Konsistenz entscheiden.

8.2 CRDTs

Conflict-free Replicated Data Types sind ein neuer Ansatz zur Behandlung der Problematik von Konsistenzerhaltung zwischen verteilten Datenständen. Dabei ist die Idee, Datentypen zu entwickeln, welche nativ kommutativ sind. Hierbei wird weder eine Historie, noch eine Konflikterkennung zur Gewährleistung der Konsistenz benötigt

[20] siehe hierzu pessimistische locks aus Kapitel 5.1

[32, Vgl. S. 104]. Ein einfaches Beispiel für ein CRDT ist das Grow-Only Set (im Folgenden G-Set genannt). Bei diesem Datentyp handelt es sich um eine Menge, bei der nur Elemente hinzugefügt, nicht aber wieder entfernt werden können. Selbst wenn auf zwei unterschiedlichen Klienten Änderungen an diesem Datentyp vorgenommen werden, können die Ergebnisse egal nach welcher Zeitspanne immer konfliktfrei repliziert werden. Hierbei wurde ein einfaches Beispiel vorgestellt, wobei das CRDTs auch komplexere Implementierungen erlauben.

Grundsätzlich gibt es dabei zwei Herangehensweisen, wie CRDTs umgesetzt werden können. Bei State-based CRDTs werden Zustände zwischen den einzelnen Replikas ausgetauscht. Sie werden auch als Convergent Replicated Data Types (im Folgenden CvRDT genannt) bezeichnet. Eine Änderung an einem CvRDT wird dabei zuerst lokal angewandt und anschließend wird der gesamte neue Zustand an die anderen Replikas verteilt und jeweils konfliktfrei zusammengeführt (Merge). Die merge-Operationen müssen dabei immer kommutativ, idempotent und assoziativ sein [33, Vgl. S. 127]. Nach einem Merge muss der neue Zustand dabei größer sein als der alte. Bei Operation-based CRDTs werden dabei Nachrichten mit den durchgeführten Änderungen ähnlich wie bei operational transformation ausgetauscht [34, Vgl.]. Sie werden auch als Commutatice Replicated Data Types (im folgenden CmRDT genannt) bezeichnet. Dabei wird die Änderung erst lokal angewandt und anschließend nur die Änderung an alle Replikas verteilt. Zusätzlich ist es erforderlich zu garantieren, dass die Operationen während der Übertragung nicht verloren gehen und auf jedem Replika auch nur einmal angewandt werden. Dieses Konzept ist dabei meistens datensparender in der Übertragung als die State-based CRDTs und besitzen oft einfachere und kompaktere Status, da Sie sich auf die Garantie der einmaligen Zustellung verlassen und nicht idempotente Operationen nicht unterstützen müssen [33, Vgl. S. 127]. Mit diesen beiden Herangehensweisen wurden von verschiedenen Autor*innen einige CRDTs entworfen. Grundsätzlich können die Datentypen dabei in Sets, Register, Graphen und Counter untergliedert werden, wobei einige Implementierungen in Tabelle 3 erläutert werden. Hierbei ist zu beachten, dass Datentypen wie zum Beispiel Counter grundsätzlich als CmRDT und CvRDT implementiert werden können und in dieser Übersicht nur die gängigsten Implementierungen betrachtet werden.

Grow-only Counter (G-Counter)	Hierbei handelt es sich um einen Zähler, welcher nur erhöht werden kann. Bei der Implementierung als CvRDT wird der globale Gesamtwert des Datentyps aus der Summe aller Einzelwerte berechnet.
Positive-Negative Counter (PN-Counter)	Der PN-Counter ist ein Zähler, bei dem der Wert sowohl vergrößert als auch verkleinert werden kann. In der Implementierung als CmRDT setzt er sich dabei aus zwei G-Counter zusammen, wobei ein Counter die Anzahl der Inkrements und der andere die Anzahl der Dekrements speichert.
Two-Phase Set (2P-Set)	In Anlehnung an den bereits erläuterten PN-Counter handelt es sich beim 2P-Set um einen Datentyp, bei dem Elemente sowohl hinzugefügt als auch wieder entfernt werden können. Als CvRDT ist die Implementierung zum Beispiel durch die Verwendung von 2 G-Sets möglich, wobei ein G-Set die hinzugefügten und das andere die entfernten Elemente speichert. Das Entfernen von Elementen hat dabei immer Vorrang. Sobald also ein Element in dem Set einmal entfernt wird, kann es nicht wieder hinzugefügt werden.
Observed-Remove Set (OR-Set)	Im Gegensatz zum 2P-Set hat beim OR-Set das Hinzufügen immer Vorrang. Als CmRDT implementiert wird davon ausgegangen, dass das Ergebnis konkurrierender Operationen durch eine neue ID erzeugt werden kann. Konstruiert ist es dabei ebenfalls aus zwei G-Sets, welche die Tupel aus Element und ID beinhalten. Beim Löschen eines Elements wird es in das andere G-Set verschoben und beim erneuten Hinzufügen, wird im entsprechenden G-Set ein neues Tupel mit aktuellerer ID hinterlegt. Nach einem Merge hat das Hinzufügen durch die neuere ID somit immer Vorrang.
Last-Writer-Wins Register (LWW-Register)	Bei dem LWW-Register wird die Strategie verwendet, dass mithilfe einer totalen Reihenfolge der Änderungen durch die Verwendung von Zeitstempeln ältere Datenstände immer durch neue überschrieben werden. Dabei können jedoch Daten verloren gehen, wenn zum Beispiel zwei Änderungen nahezu gleichzeitig erzeugt werden.

Tabelle 3: Übersicht CRTD´s [35, S. 17-20]

Als erste Implementierung für CRDTs gilt dabei die WOOT-Datenstruktur[21] mit dem Ziel, bekannte Problemfälle des älteren Konzeptes operational transformation durch einen komplett neuen Ansatz zu lösen [37, Vgl. S. 2]. Dem folgten weitere Revisionen von WOOT, sowie andere CRDTs wie zum Beispiel RGA, Logoot oder LogootSplit. Die Firma TomTom hat für Anwendungsfälle in Ihrem System zusätzlich einen eigenen CRDT entworfen, bei dem Sie die Vorteile verschiedener CRDTs kombinieren. Bei Ihrem Datentyp „OUR-Set“ wird ein Set verwendet, welches aus nur einer Liste mit einem Remove-Flag und ebenfalls einem eindeutigen Identifier an jedem Element besteht und updatebar ist. Für die Konfliktbehebung bei einem Merge von Änderungen an demselben Element wird dabei immer das Element mit der höchsten ID verwendet[22]. Hierbei handelt es sich nicht um alle existierenden CRDTs. Weitere Alternativen sind unter anderem auch Multi-Value Register (MV-Register) oder Last-Write-Wins-element-Set (LWW-element-Set), welche wie viele andere in dieser Arbeit jedoch nicht weiter betrachtet werden können.

Ein für diese Arbeit relevanter Anwendungsfall für CRDTs sind dabei kollaborative Texteditoren, da im Konfliktfall das Verhalten des Algorithmus immer nachvollziehbar bleibt. Ein Dokument kann dabei als eine Menge von Elementen dargestellt werden, wobei ein Element beispielsweise ein Zeichen, eine Zeile oder andere Inhalte repräsentieren kann. Für einige kollaborative Texteditoren gibt es zusätzlich bereits einige Umsetzungen von CRDTs. Hierbei zu erwähnen ist zum Beispiel Teletype für Atom[23], welches mit seinem teletype-crdt Repository[24] die Implementierung auch öffentlich zur Verfügung stellt und an dessen sich bei einer möglichen Implementierung auch orientiert werden kann. Problematisch ist, dass viele Umsetzungen für die Nutzung von CRDTs in kollaborativen Texteditoren nie komplett fertiggestellt oder langfristig unterstützt wurden. So ist Teletype for Atom die weitreichendste Implementierung für CRDTs, welche zum jetzigen Stand der Zeit öffentlich gefunden werden konnte, obwohl sie sich dabei ebenfalls nur in einem Beta-Status befindet und seit zwei Jahren nicht mehr aktualisiert wurde. Nachdem nun ein Überblick über alternative Konzepte und Herangehensweisen der Anwendungsfälle des Saros-VSCode Prototyps gewonnen wurde, kann im folgenden Abschnitt die Verwendung eines bestimmten Konzeptes evaluiert und mit dem Konzept von operational transformation verglichen werden.

8.3 Vergleich der verschiedenen Konzepte

In diesem Kapitel wird die Verwendung der unterschiedlichen Konzepte gegeneinander abgewogen und sich am Ende für die Nutzung einer Technologie bei der weiteren Konzeptionierung eines neuen Konzeptes entschieden. Verteilte Datenbanksysteme sind dabei der erste betrachtete Themenkomplex und bieten eine Fülle an Möglichkeiten, insbesondere zur Erhaltung der Konsistenz zwischen verschiedenen Datenstores. Aufgrund der langen Historie und der auch immer wieder neu erforschten oder an neue Anforderungen angepassten Theoreme und Techniken, sind sie dabei sehr

[21] siehe [36]
[22] siehe [38]
[23] siehe [39]
[24] siehe [40]

ausgereift und arbeiten sehr stabil und konsistent. Problematisch ist jedoch, dass der Fokus der verschiedenen Architekturen entweder auf der Erhöhung der Performance oder der Ausfallsicherheit liegt, was nicht, wie bereits im vorherigen Kapitel erläutert, das übergeordnete Ziel der Architektur mit dem SLS ist. Dazu kommt, dass während der Recherchen kaum detaillierte Konzepte, Übersichten oder Abhandlungen zur Implementierung und eventuellen Schwierigkeiten gefunden werden konnten. Die Entwickler*innen der Datenbanken beschreiben zwar häufig, wie mit Ihrem System die Einrichtung der Replikation, also die Administration der Datenstores, erfolgen soll, die eigentlichen Techniken im Hintergrund werden jedoch nicht weiter spezifiziert. Aufgrund der fehlenden Informationsgrundlage wird der Ansatz der verteilten Datenbanken daher nicht weiter betrachtet.

Conflict-free Replicated Datatypes wiederum sind eine konkurrenzfähige und moderne Alternative zu operational transformation. Sie wurden entwickelt, um bekannte Probleme mit OT-Algorithmen einfach zu lösen und basieren dabei auf einem immer verständlichen und einfach zu implementierenden Konzept. Hinzu kommt, dass bereits viele Datentypen entwickelt und in verschiedensten Arbeiten theoretisch betrachtet und analysiert wurden. Auch für die Verwendung in kollaborativen Editoren gibt es erste Konzepte und sogar Implementierungen, welche im Verlauf dieser Arbeit bereits vorgestellt wurden. Am Ende bietet sie einige beachtliche Vorteile gegenüber operational transformation insbesondere in der Vermeidung beziehungsweise Auflösung von Konflikten. Ein schwerwiegendes Argument gegen die Verwendung von CRDTs in diesem Konzept ist jedoch der Implementierungsaufwand und die Wartbarkeit der Anwendung im Vergleich zum bereits benutzten Konzept mit OT. Auch wenn CRDTs rein technisch durchaus eine gute Wahl gegenüber OT wären, müssten für die Verwendung im SLS aktuell wiederverwendbare Implementierungen des alten Prototyps neu implementiert und der Softwarestack von Saros allgemein um eine neue Technologie erweitert werden.

Da eine neue Technologie die Wartbarkeit der Anwendung Saros im Allgemeinen verkomplizieren würde, operational transformation zusätzlich bereits implementiert und die entsprechenden Funktionen einfach verwendet werden können, wurde sich trotz der positiven Merkmale von CRDTs gegen die Verwendung dieser und für die Nutzung von operational transformation entschieden. Dafür spricht auch, dass zusätzlich für die Verbindung zwischen dem Jupiter-Server und dem SLS OT trotzdem weiterhin verwendet werden muss und das Know-How der Saros-Entwickler*innen im Bereich OT bereits vorhanden ist. Für CRDTs jedoch müsste das Know-How erst neu aufgebaut werden. Die Entwicklung eines darauf basierenden Konzeptes wird dabei im nächsten Kapitel beschrieben.

9 Entwicklung eines neuen Konzeptes

Nach der Analyse aus dem letzten Kapitel, wird im folgenden Abschnitt ein Konzept passend für die Architektur des VSCode Prototypen entwickelt und vorgestellt. Dabei werden die gesammelten Erfahrungen und erkannten Probleme aus den letzten Kapiteln ebenfalls bedacht und in die neue Architektur integriert. Die bereits in Kapitel 7.1 definierten Anforderungen an eine alternative Saros-Architektur haben sich

dabei noch einmal verändert. Gesucht wird nun ein Konzept, welches den folgenden Rahmenbedingungen entspricht:

1. Der in Kapitel 6 anhand des Beispiels erklärte Problemfall muss gelöst werden

2. Der SLS soll erhalten bleiben, um Saros mit vergleichsweise wenig Programmieraufwand auch für andere vergleichbare Editoren oder IDE´s mithilfe des LSP-Protokolls unterstützen zu können

3. Das aktuelle Verhalten von Saros, beziehungsweise die Nutzer*innenerfahrung soll nachvollziehbar bleiben und mit anderen Saros-Implementierungen vergleichbar sein

4. Die im Jupiter-Algorithmus definierten Regeln zur Konfliktbehandlung müssen beachtet werden, damit die Anwendung sich immer wie erwartet verhält

5. Als Kernkomponente einer möglichen Konfliktauflösung soll, der in Saros bereits implementierte OT-Algorithmus verwendet werden

Basierend auf den neuen Anforderungen wurden verschiedene Lösungsansätze betrachtet, wobei die finale Architektur im Folgenden vorgestellt wird. Wie bereits in Kapitel 7 werden dabei anschließend zwei verschiedene Fälle betrachtet: Beim SLS eintreffende Änderungsbenachrichtigungen vom Saros-VSCode Klienten und beim SLS eintreffende Änderungsbenachrichtigungen vom Jupiter-Server.

9.1 Allgemeine Definitionen

Das neue Konzept sieht dabei allgemein vor, den SLS als eine abgewandelte Form des Jupiter-Servers zu gestalten. Um die einzelnen Abläufe hierbei besser darstellen zu können, wird das Konzept des SLS in zwei Aufgabenbereiche unterteilt. Der SLS Jupiter-Server Bereich agiert dabei aus Sicht des Jupiter-Servers als Klient und ist verantwortlich für die Synchronisierung und Kommunikation der Änderungen von und zum SLS. Der SLS VSCode Bereich arbeitet aus Sicht des Saros-VSCode Klienten als sein Jupiter-Server und ist verantwortlich für die Synchronisierung und Kommunikation der Änderungen von und zum Saros-VSCode Klienten über LSP. Vereinfacht dargestellt, ist der SLS ein Jupiter-Server, welcher die Änderungen seiner Klienten, dem Saros-VSCode Klient und dem „richtigen“ Jupiter-Server, entgegennimmt und an die anderen Klienten verteilt. Dabei wird der SLS jedoch immer nur zwischen diesen zwei Parteien vermitteln und synchronisieren.

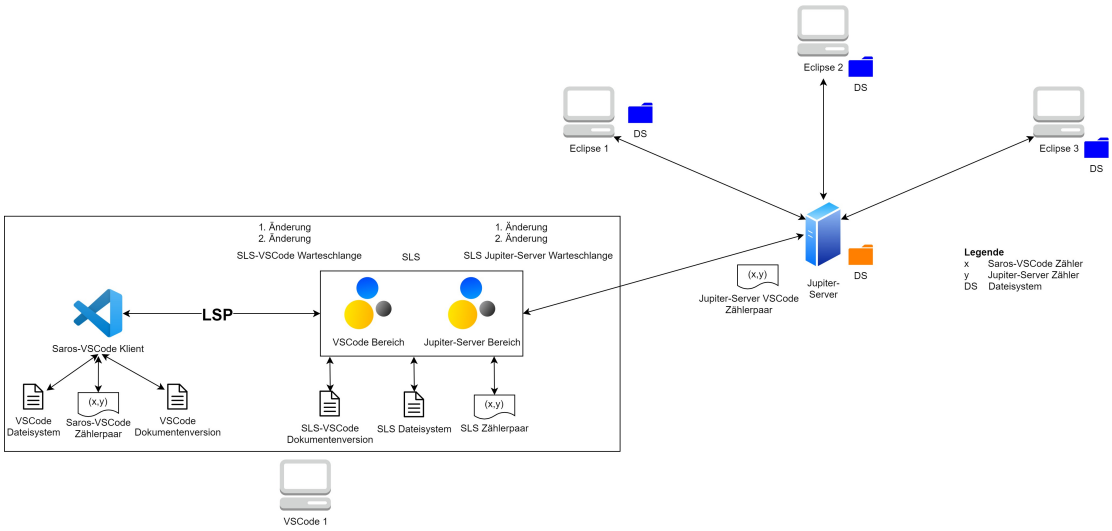

Abbildung 14: Übersicht über die finale Architektur

Die Besonderheit dabei, ist die Synchronisierung der Änderungen zwischen dem SLS VSCode Bereich und dem Saros-VSCode Klient. Im Normalfall ist auf beiden Seiten ein Algorithmus für operational transformation implementiert, welcher sich um eine eventuelle Konfliktauflösung kümmert. In dieser Architektur ist jedoch der SLS VSCode Bereich so aufgebaut, dass er, zusätzlich zu seiner eigenen, auch die Konfliktlösung für den Saros-VSCode Klienten übernimmt. Damit soll die Grundlage geschaffen werden, über die gesamte Saros-Architektur immer die gleiche Technik zur Konfliktlösung verwenden zu können. Zusätzlich soll der immer wieder aufwendig zu implementierendem Teil von Saros, wie in den Anforderungen vorgegeben, nicht in der Erweiterung des angebundenen Editors oder der IDE vorgehalten werden. Weiterhin wird dadurch die Anbindung neuer Editoren über LSP ermöglicht.

Der Hauptgrund dafür, dass in diesem Konzept die Konfliktlösung für den Saros-VSCode Klienten in den SLS VSCode Bereich ausgelagert und zusätzlich die Problematik der Asynchronität der *Extension API* gelöst werden kann, sind zwei neue Komponenten des Saros VSCode Prototyps. Der Saros-VSCode Klient zählt jetzt einerseits die von ihm verarbeiteten Änderungen jeweils von dem/der VSCode Nutzer*in und dem SLS, wie bereits in Kapitel 5.2 beschrieben und speichert andererseits die aktuelle Dokumentenversion von VSCode, auf welche die verarbeitete Änderung im Saros-VSCode Klienten angewandt wurde. Hierfür wurde das Konzept um ein Saros-VSCode Zählerpaar mit einem Saros-VSCode Zähler für die verarbeiteten Änderungen vom Saros-VSCode Klienten und einem Jupiter-Server Zähler für die verarbeiteten Änderungen vom Jupiter-Server erweitert. Der SLS verwaltet ebenfalls ein SLS Zählerpaar, wobei es sich hierbei um eine besondere Implementierung handelt, da es die verarbeiteten Änderungen beider Kommunikationsrichtungen repräsentiert und nicht, wie normalerweise, einer der Zähler die lokal verarbeiteten Änderungen darstellt. Der Zähler vom Jupiter-Server wird im Konzept Jupiter-Server VSCode Zählerpaar genannt, da der Jupiter-Server für jeden Klienten ein eigenes Zählerpaar und somit einen eigenen 2D-Zustandsraum vorhält [16, Vgl. S. 117]. Zusätzlich wird im Saros-VSCode Klienten die VSCode Dokumentenversion eingeführt. In dieser hin-

terlegt der Klient bei der Verarbeitung einer Änderung die vom Event mitgelieferte Dokumentenversion nach der neuen Änderung. VSCode garantiert dabei, dass die Dokumentenversion nach jeder Änderung immer erhöht wird [41, Vgl.]. Im SLS wird dabei ebenfalls die letzte von VSCode bekannte Dokumentenversion in der Variable SLS-VSCode Dokumentenversion hinterlegt. Diese wird anschließend bei jeder Änderung vom SLS an den Saros-VSCode Klienten immer mitgeliefert, um dem Klienten mitzuteilen, auf welcher Dokumentenversion die eventuell transformierten Änderungen beruhen.

Damit kann dem SLS für die Konfliktlösung der Stand der verarbeiteten Änderungen vom Saros-VSCode Klient mitgeteilt werden, wodurch auch erstmals Konflikte zwischen dem Saros-VSCode Klient und dem SLS erkannt und behoben werden können. Durch das Speichern der Dokumentenversion, wird zusätzlich ein Problemfall abgedeckt, bei dem der SLS eine Änderung an den Saros-VSCode Klienten sendet. Während die Änderung versendet wird, ist eine andere Änderung durch den/die VSCode-Nutzer*in durchgeführt worden, welche der SLS zum Zeitpunkt des Versandes seiner Änderung noch nicht kennt. Die Änderung vom SLS wäre ohne vorherige Prüfung der VSCode Dokumentenversion darauf hin auf den neueren Datenstand im Saros-VSCode Klienten angewandt worden, was nun jedoch verhindert wird. Durch die Asynchronität der *Extension API* ist es sogar theoretisch möglich, dass bei der Verarbeitung der Änderung vom SLS im Saros-VSCode Klient das Saros-VSCode Zählerpaar noch nicht aktualisiert ist und der Datenstand im Saros-VSCode Klient sich trotzdem von dem des SLS unterscheidet. Da vor jeder vom SLS initiierten Änderung im Saros-VSCode Klient vorher jedoch einmal die aktuelle VSCode Dokumentenversion abgerufen und mit der mitgelieferten vom SLS abgeglichen wird, kann das Problem nicht auftreten.

Um die bereits in den vorherigen Kapiteln beschriebenen Probleme bei der Verwendung von zwei Datenständen, jeweils für den Jupiter-Server und den Saros-VSCode Klienten im SLS zu vermeiden, wird in diesem Konzept, wie im ursprünglichen Prototyp vorgesehen, nur ein Datenstand im Saros Language Server verwendet. Der Datenstand wird im Konzept dabei durch das SLS Dateisystem repräsentiert. Ein hierfür ausschlaggebender Aspekt ist, dass, wie in der Implementierung des Jupiter-Servers[25], immer nur eine Änderung gleichzeitig verarbeitet wird. Während der Verarbeitung einer Änderung wird der SLS mit seinen Ressourcen, wie dem SLS Zählerpaar und dem SLS Dateisystem, vor Zugriffe durch andere blockiert. Die dabei eventuell auftretenden, kurzzeitigen Verzögerungen in der Verarbeitung und Weiterleitung sind im Verhältnis zur gewonnenen Konsistenz, Zuverlässigkeit bei der Bearbeitung und fehlenden unverständlichen Verhaltensweisen der Anwendung von geringerer Bedeutung und außerdem in Anlehnung an den Jupiter-Server auch in anderen Teilen der Saros-Architektur bereits eingesetzt worden. Dabei basiert das Konzept auf dem Prinzip von eventual consistency, welches im vorherigen Kapitel bereits erläutert wurde.

Damit eintreffende Änderungen während einer aktuellen Blockierung durch eine in der Verarbeitung befindlichen Änderung nicht verloren gehen, wird das Konzept zusätzlich noch um zwei Warteschlangen ergänzt. Die SLS-VSCode Warteschlange

[25] siehe `https://github.com/saros-project/saros/pull/1111#issuecomment-835430892`

speichert dabei die eintreffenden Änderungen vom Saros-VSCode Klienten inklusive der neuen, mitgelieferten VSCode Dokumentenversionen und in der SLS Jupiter-Server Warteschlange werden die eintreffenden Änderungen vom Jupiter-Server hinterlegt. Jede Warteschlange speichert zu der Änderung jeweils das Ankunftsdatum, um anschließend eine Abarbeitung der Änderung in der Reihenfolge gewährleisten zu können, wie sie den SLS erreicht haben. In Abbildung 14 ist das eben erläuterte Konzept dargestellt, wobei die genauen Abläufe im nächsten Kapitel erläutert werden.

9.2 Ablauf einer von dem/der VSCode Nutzer*in initiierten Änderung im Saros-VSCode Klienten

Nachdem im vorherigen Kapitel die Komponenten des neuen Konzeptes erklärt wurden, werden in den folgenden beiden Kapiteln die genauen Abläufe und Funktionsweisen erläutert. Wie bereits in Kapitel 7.2 werden dabei zwei verschiedene Fälle betrachtet, wobei mit dem Fall einer von dem/der VSCode Nutzer*in initiierten Änderung begonnen wird. Anschließend wird der Fall einer vom Jupiter-Server initiierten Änderung betrachtet. Um den folgenden Erklärungen besser folgen zu können, ist zu Beginn der Kapitel 9.2 und 9.3 eine Übersicht der einzelnen Prozessschritte dargestellt[26]. Anschließend an die Übersichten folgen detaillierte Beschreibungen der Schritte inklusive der Hintergründe zu den entsprechenden Entscheidungen. Dabei sind die textuellen Beschreibungen mit den entsprechenden Nummerierungen in der Übersicht der Prozessschritte verknüpft.

Übersicht zum Ablauf nach einer von dem/der VSCode Nutzer*in initiierten Änderung im Saros-VSCode Klienten

1. Änderung durch den/die VSCode Nutzer*in
2. Asynchrone Informierung des Saros-VSCode Klienten über das Event
3. Abrufen des Saros-VSCode Zählerpaars und der VSCode Dokumentenversion
4. Erhöhen des Saros-VSCode Zählers um 1
5. Senden der Änderung mit Zählerpaar und Dokumentenversion an SLS VSCode Bereich
6. Prüfen, ob aktive Blockierung im SLS
   (a) Wenn Blockierung vorhanden
      i. Hinzufügen der Änderung inklusive Zeitstempel der Ankunftszeit, VSCode Dokumentenversion und Saros-VSCode Zählerpaars zur SLS-VSCode Warteschlange
      -> Weiter mit Schritt 14.

[26] Aufgrund des Umfangs der Abläufe ist es zu empfehlen, während der Erarbeitung der einzelnen Schritte diese zusätzlich zu der Übersicht der Prozessschritte auch parallel in der Darstellung der Architektur in Abbildung 14 und den Aktivitätsdiagrammen in Abbildung 15 und 16 zu verfolgen.

   (b) Wenn keine Blockierung vorhanden
   -> Weiter mit Schritt 7.

7. Blockieren des SLS und seiner Ressourcen

8. Vergleichen des Jupiter-Server Zählers vom Saros-VSCode Zählerpaar mit dem SLS Zählerpaar

   (a) Fall 1.1: Der Jupiter-Server Zähler vom Saros-VSCode Zählerpaar ist gleich dem Jupiter-Server Zähler vom SLS Zählerpaar
   -> Weiter mit Schritt 9.

   (b) Fall 1.2: Der Jupiter-Server Zähler vom Saros-VSCode Zählerpaar ist um mindestens einen Status höher als der Jupiter-Server Zähler vom SLS Zählerpaar

      i. Synchronisation von SLS und Saros-VSCode Klient mit Jupiter-Server
      -> Weiter mit Schritt 12.

   (c) Fall 1.3: Der Jupiter-Server Zähler vom SLS Zählerpaar ist um mindestens einen Status höher als der Jupiter-Server Zähler vom Saros-VSCode Zählerpaar

      i. Konfliktlösung durch Jupiter-Algorithmus
      -> Weiter mit Schritt 9.

9. Vergleichen des Saros-VSCode Zähler vom Saros-VSCode Zählerpaar mit dem SLS Zählerpaar

   (a) Fall 2.1: Der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar ist gleich dem Saros-VSCode Zähler vom SLS Zählerpaar
   -> Weiter mit Schritt 12.

   (b) Fall 2.2: Der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar ist exakt ein Status höher (+1) als der Saros-VSCode Zähler vom SLS Zählerpaar
   -> Weiter mit Schritt 10.

   (c) Fall 2.3: Der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar ist mindestens zwei Status höher als der Saros-VSCode Zähler vom SLS Zählerpaar

      i. Konfliktlösung durch Jupiter-Algorithmus
      -> Weiter mit Schritt 10.

   (d) Fall 2.4: Der Saros-VSCode Zähler vom SLS Zählerpaar ist mindestens einen Status höher als der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar

      i. Synchronisieren von Saros-VSCode Klient mit SLS
      -> Weiter mit Schritt 12.

10. Anwendung der (berechneten) Änderung am SLS Dateisystem und am SLS Zählerpaar und der SLS-VSCode Dokumentenversion

11. Kommunikation der Änderung an Jupiter-Server

12. Beenden der Blockierung des SLS und seiner Ressourcen
13. Überprüfen der Warteschlangen nach wartenden Änderungen und Bearbeitung dieser nach Ankunftsdatum bei Bedarf
14. Ende

Nach einer durchgeführten Änderung durch den/die VSCode Nutzer*in (1.) wird asynchron der Saros-VSCode Klient über das Event informiert und anschließend die Änderungsbenachrichtigung bearbeitet (2.). Dieser ruft das Saros-VSCode Zählerpaar und die aktuelle VSCode Dokumentenversion ab (3.), erhöht den Saros-VSCode Zähler um den Wert 1 (4.)[27] und sendet die Änderung inklusive des neuen Zählerpaars und der abgerufenen VSCode Dokumentenversion mit LSP an den SLS, speziell den SLS VSCode Bereich (5.). Der SLS VSCode Bereich nimmt die Änderung entgegen, prüft, ob aktuell eine Änderung verarbeitet und der SLS somit blockiert ist (6.) und blockiert, wenn der SLS frei ist (6.b), in einem nächsten Schritt die Verarbeitung weiterer Änderungen durch den SLS (7.). Sollte der SLS gerade eine Änderung verarbeiten und somit blockiert sein (6.a), wird die Änderung des Saros-VSCode Klienten mit einem aktuellen Zeitstempel, der VSCode Dokumentenversion und dem Saros-VSCode Zählerpaar in die SLS-VSCode Warteschlange angehangen (6.a.i) und der Prozess endet damit an dieser Stelle (14.).

Für den Zeitraum der Blockierung hat dabei nur der Bereich, welcher die aktuelle Änderung verarbeitet, auch Zugriff auf das SLS Zählerpaar, das SLS Dateisystem und die SLS-VSCode Dokumentenversion. Wenn während der Verarbeitung einer Änderung eine andere beim SLS eintrifft, wird diese ebenfalls, je nach Herkunft, entweder in die SLS Jupiter-Server Warteschlange bei einer Änderung vom Jupiter-Server oder in die SLS-VSCode Warteschlange bei einer Änderung vom Saros-VSCode Klienten nach dem FIFO-Prinzip angehangen. Das zu jeder Änderung mitgespeicherte Datum des Eintreffens beim SLS wird dabei gesichert, damit später die Warteschlangen auch in der richtigen Reihenfolge abgearbeitet werden können. In Anlehnung an den verbesserten dOPT-Algorithmus muss anschließend das mitgelieferte Saros-VSCode Zählerpaar mit dem SLS Zählerpaar verglichen werden. Begonnen wird in diesem Konzept dabei mit dem Vergleich des Jupiter-Server Zählers vom Saros-VSCode Zählerpaar mit dem SLS Zählerpaar (8.). Dabei können folgende mögliche Fälle auftreten:

- Fall 1.1: Der Jupiter-Server Zähler vom Saros-VSCode Zählerpaar ist gleich dem Jupiter-Server Zähler vom SLS Zählerpaar
- Fall 1.2: Der Jupiter-Server Zähler vom Saros-VSCode Zählerpaar ist um mindestens einen Status höher als der Jupiter-Server Zähler vom SLS Zählerpaar
- Fall 1.3: Der Jupiter-Server Zähler vom SLS Zählerpaar ist um mindestens einen Status höher als der Jupiter-Server Zähler vom Saros-VSCode Zählerpaar

Wenn beim Abgleich der Fall 1.1 auftritt (8.a), kann der SLS VSCode Bereich mit der Bearbeitung fortfahren, da die Änderungen vom Saros-VSCode Klienten auf dem

[27] siehe Kapitel 5.2

gleichen Jupiter-Stand wie der SLS basieren und somit keine Konflikte auftreten. Bei Eintreten vom Fall 1.2 (8.b) würde ein Fehler vorliegen, da der Saros-VSCode Klient eine Änderung vom Jupiter-Server verarbeitet hätte, welche der SLS nicht verarbeitet hat. Bei einer robusten Implementierung sollte dieser Fall nie eintreten. Da er jedoch theoretisch möglich ist, wird er hier mit behandelt. Wenn dieser Fall also eintritt, muss der SLS Datenstand einmal mit dem Datenstand des Jupiter-Server und der Datenstand des Saros-VSCode Klienten mit dem des SLS überschrieben werden (8.b.i). In diesem Fall gehen zwar Änderungen des/der VSCode Nutzer*in verloren, jedoch lässt sich das an dieser Stelle nicht mehr verhindern, da der Saros-VSCode Klient und der SLS ansonsten nicht mehr miteinander synchronisiert werden können. Abschließend wird die Blockierung des SLS und seiner Ressourcen aufgehoben (12.) und die Warteschlange nach wartenden Änderungen durchsucht (13.). Fall 1.3 (8.c) würde eintreten, wenn die Änderung des Saros-VSCode Klienten auf einem älteren Jupiter-Server Datenstand basieren würde, als er mittlerweile beim SLS vorliegt. Vereinfacht gesagt, wurde vom SLS bereits eine neue Änderung des Jupiter-Server verarbeitet, welche zum Zeitpunkt der Erstellung der Änderung beim Saros-VSCode Klienten diesem noch nicht bekannt war. In diesem Fall muss erst eine Konfliktlösung mit operational transformation durchgeführt werden (8.c.i), bevor mit dem nächsten Schritt fortgefahren werden kann. Entweder nach Fall 1.1 oder 1.3 muss im weiteren Verlauf, nach der Prüfung des Jupiter-Server Zählers, nun anschließend der Saros-VSCode Zähler bei beiden Zählerpaaren verglichen werden (9.). Dabei können folgende Fälle auftreten:

- Fall 2.1: Der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar ist gleich dem Saros-VSCode Zähler vom SLS Zählerpaar
- Fall 2.2: Der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar ist exakt ein Status höher (+1) als der Saros-VSCode Zähler vom SLS Zählerpaar
- Fall 2.3: Der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar ist mindestens zwei Status höher als der Saros-VSCode Zähler vom SLS Zählerpaar
- Fall 2.4: Der Saros-VSCode Zähler vom SLS Zählerpaar ist mindestens einen Status höher als der Saros-VSCode Zähler vom Saros-VSCode Zählerpaar

Wenn beim Abgleich des Saros-VSCode Zähler Fall 2.1 eintritt (9.a), liegt ein Fehler vor, da dem SLS keine Änderung gemeldet wurde, welche dieser noch nicht kennt und bereits verarbeitet hat. Auch dieser Fall ist nur theoretisch möglich und der Prozess wäre damit nach dem Aufheben der Blockierungen des SLS und seiner Ressourcen (12.) und der anschließenden Prüfung der Warteschlangen (13.) zu Ende (14.). Bei Fall 2.2 (9.b) ist der Saros-VSCode Klient nur eine Änderung weiter als der SLS. Das ist dabei der bestmögliche Fall, da die Änderung ohne Konflikte verarbeitet werden kann. In Fall 2.3 (9.c) wird eine Situation beschrieben, in der dem SLS eine Änderung gemeldet wurde, welche mehr als eine Änderung vom aktuellen Stand des SLS divergiert beziehungsweise dieser eine vorherige Änderung noch nicht verarbeitet hat. Zu diesem Zeitpunkt muss erst wieder eine Konfliktlösung durchgeführt werden (9.c.i), bevor mit dem nächsten Schritt fortgefahren werden kann. Fall 2.4 (9.d) ist erneut ein Szenario, welches praktisch nie auftreten sollte, theoretisch in dieser Ausarbeitung jedoch behandelt wird. Hierbei hätte der SLS bereits mehr Änderungen vom

Saros-VSCode Klienten verarbeitet, als dem Klienten selbst bekannt sind. In diesem Fall sollte der Datenstand des Klienten mit dem des SLS überschrieben werden (9.d.i), damit beide Datenstände wieder synchron sind. Anschließend wird die Blockierung des SLS und seiner Ressourcen beendet (12.) und die Warteschlangen geprüft (13.). Gehen wir in dieser Erklärung von einem optimalen Verlauf, also dem Eintritt des Falls 2.1 oder 2.3 aus, wird im nächsten Schritt des Konzeptes die Änderung des Saros-VSCode Klienten am SLS Dateisystem und somit auch am SLS Zählerpaar angewandt (10.). Außerdem wird die SLS-VSCode Dokumentenversion mit dem neuen Wert vom Saros-VSCode Klienten aktualisiert. Hierbei treten keine Konflikte auf, da alle Ressourcen noch durch die Blockierung vor einem konkurrierenden Zugriff geschützt sind und Konflikte in den Änderungen bereits vorher gelöst wurden. Nach Abschluss dieses Schrittes, wurde die Änderung vom Saros-VSCode Klienten auch erfolgreich beim SLS verarbeitet und kann vom SLS an den Jupiter-Server kommuniziert werden (11.). Anschließend kann die Blockierung des SLS und seiner Ressourcen beendet (12.) und die Warteschlangen nach neuen Änderungen geprüft werden (13.). Neue Änderungen werden dabei in aufsteigender Reihenfolge des Zeitstempels über beide Warteschlangen abgearbeitet und anschließend dort entfernt. Sollte dabei eine Änderung aus der SLS Jupiter-Server Warteschlange exakt denselben Zeitstempel wie eine Änderung aus der SLS-VSCode Warteschlange aufweisen, wird die Änderung des Jupiter-Servers zuerst abgearbeitet. Anschließend ist die Verarbeitung der Änderung beendet (14.).

9.3 Ablauf einer vom Jupiter-Server initiierten Änderung des Saros-VSCode Klienten

Nachdem nun der Ablauf einer Änderung vom Saros-VSCode Klienten betrachtet wurde, wird im Folgenden der umgekehrte Weg, einer Änderung vom Jupiter-Server, erläutert. Wie im vorherigen Kapitel ist dabei zuerst eine Übersicht der einzelnen Prozessschritte dargestellt. Anschließend folgen detaillierte Beschreibungen der einzelnen Schritte inklusive der Hintergründe zu den entsprechenden Entscheidungen.

Übersicht zum Ablauf nach einer vom Jupiter-Server initiierten Änderung

1. Abarbeiten einer Änderung eines anderen Klienten im Jupiter-Server
2. Senden der Änderungsbenachrichtigung an den SLS
3. Ankunft im SLS Jupiter-Server Bereich
4. Prüfen, ob aktive Blockierung im SLS
   (a) Wenn Blockierung vorhanden
      i. Hinzufügen der Änderung inklusive Zeitstempel der Ankunftszeit zur SLS Jupiter-Server Warteschlange
      -> Weiter mit Schritt 16.
   (b) Wenn keine Blockierung vorhanden
   -> Weiter mit Schritt 5.

5. Blockieren des SLS und seiner Ressourcen
6. Vergleichen des Jupiter-Server Zählers vom Jupiter-Server VSCode Zählerpaar mit dem SLS Zählerpaar
   (a) Fall 3.1: Der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar ist gleich dem Jupiter-Server Zähler vom SLS Zählerpaar
   -> Weiter mit Schritt 14.
   (b) Fall 3.2: Der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar ist exakt ein Status höher (+1) als der Jupiter-Server Zähler vom SLS Zählerpaar
   -> Weiter mit Schritt 7.
   (c) Fall 3.3: Der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar mindestens zwei Status höher als der Jupiter-Server Zähler vom SLS Zählerpaar
      i. Konfliktlösung durch Jupiter-Algorithmus
      -> Weiter mit Schritt 7.
   (d) Fall 3.4: Der Jupiter-Server Zähler vom SLS Zählerpaar ist mindestens einen Status höher als der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar
      i. Synchronisieren von Saros-VSCode Klient und SLS mit dem Jupiter-Server
      -> Weiter mit Schritt 14.
7. Vergleichen des Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar mit dem SLS Zählerpaar
   (a) Fall 4.1: Der Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar ist gleich dem Saros-VSCode Zähler vom SLS Zählerpaar
   -> Weiter mit Schritt 8.
   (b) Fall 4.2: Der Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar ist um mindestens einen Status höher als der Saros-VSCode Zähler vom SLS Zählerpaar
      i. Synchronisieren von Saros-VSCode Klient und SLS mit dem Jupiter-Server
      -> Weiter mit Schritt 14.
   (c) Fall 4.3: Der Saros-VSCode Zähler vom SLS Zählerpaar ist um mindestens einen Status höher als der Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar
      i. Konfliktlösung durch Jupiter-Algorithmus
      -> Weiter mit Schritt 8.
8. Anwendung der (berechneten) Änderung am SLS Dateisystem und am SLS Zählerpaar
9. Prüfen der SLS-VSCode Warteschlange durch den SLS VSCode Bereich

   (a) Wenn SLS-VSCode Warteschlange leer
   -> Weiter mit Schritt 10.
   (b) Wenn SLS-VSCode Warteschlange nicht leer
      i. Anwenden aller Änderungen in der SLS-VSCode Warteschlange in richtiger Reihenfolge auf den aktuellen SLS Datenstand
      -> Weiter mit Schritt 10.

10. Weiterleiten der Änderung zur Anwendung an Saros-VSCode Klienten

11. Prüfung der Dokumentenversionen im Saros-VSCode Klienten
   (a) Wenn Version gleich
      i. Anwenden der Änderung
      ii. Senden einer Bestätigung
      -> Weiter mit Schritt 12.
   (b) Wenn Version ungleich
      i. Verwerfen der Änderung
      ii. Senden einer Ablehnung
      -> Weiter mit Schritt 12.

12. Warten auf Bestätigung der Durchführung durch den Saros-VSCode Klienten
   (a) Wenn Bestätigung erhalten
      i. Versenden eventuell verarbeiteter Änderungen an Jupiter-Server
      -> Weiter mit Schritt 14.
   (b) Wenn keine Bestätigung bzw. Ablehnung erhalten
   -> Weiter mit Schritt 13.

13. Prüfen der SLS-VSCode Warteschlange durch den SLS VSCode Bereich
   (a) Wenn SLS-VSCode Warteschlange leer
      i. Wenn Timer noch nicht abgewartet
         A. Timer abwarten
         -> Weiter mit Schritt 13.
      ii. Wenn Timer abgewartet
         A. Synchronisieren des Datenstandes vom Saros-VSCode Klienten mit dem aktuellen Datenstand vom SLS
         -> Weiter mit Schritt 14.
   (b) Wenn SLS-VSCode Warteschlange nicht leer
      i. Reset, ob Timer bereits abgewartet wurde
      -> Weiter mit Schritt 9.(b)

14. Beenden der Blockierung des SLS und seiner Ressourcen

15. Überprüfen der Warteschlangen nach wartenden Änderungen und Bearbeitung dieser nach Ankunftsdatum bei Bedarf

16. Ende

Die Ausgangssituation ist, dass der Jupiter-Server eine Änderung eines anderen Klienten verarbeitet hat (1.) und nun eine Änderungsbenachrichtigung an den SLS sendet (2.). Diese Änderung wird dabei vom SLS Jupiter-Server Bereich in Empfang genommen (3.). Bevor mit der Verarbeitung begonnen werden kann, muss zuerst geprüft werden, ob der SLS aktuell durch eine andere Änderung blockiert ist (4.). Wenn dies zutrifft (4.a), wird die Änderung vom Jupiter-Server inklusive eines aktuellen Zeitstempels in die Jupiter-Server Warteschlange angehangen (4.a.i) und der Prozess ist beendet (16.). Wenn der SLS aktuell keine andere Änderung verarbeitet (4.b), blockiert nun der SLS Jupiter-Server Bereich zur Verarbeitung seiner empfangenen Änderung den SLS und seine Ressourcen (5.). Anschließend müssen wieder die Zählerpaare verglichen werden, wobei in diesem Ablauf zu Beginn der Jupiter-Server Zählers vom Jupiter-Server VSCode Zählerpaar mit dem SLS Zählerpaar verglichen wird (6.). Dabei können folgende mögliche Fälle auftreten:

- Fall 3.1: Der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar ist gleich dem Jupiter-Server Zähler vom SLS Zählerpaar
- Fall 3.2: Der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar ist exakt ein Status höher (+1) als der Jupiter-Server Zähler vom SLS Zählerpaar
- Fall 3.3: Der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar mindestens zwei Status höher als der Jupiter-Server Zähler vom SLS Zählerpaar
- Fall 3.4: Der Jupiter-Server Zähler vom SLS Zählerpaar ist mindestens einen Status höher als der Jupiter-Server Zähler vom Jupiter-Server VSCode Zählerpaar

Die verschiedenen Fälle sind dabei vergleichbar mit den Fällen 2.1 – 2.4 aus dem vorherigen Kapitel. Bei Eintritt des Falls 3.1 (6.a) wurde dem SLS keine Änderung vom Jupiter-Server gemeldet, welche der SLS noch nicht kennt oder bereits verarbeitet hat. Aus diesem Grund werden die entsprechenden Blockierungen des SLS entfernt und die Warteschlangen nach neuen Änderungen überprüft. Fall 3.2 (6.b) ist wiederum der Optimalfall, da der Jupiter-Server nur eine Änderung weiter als der SLS ist und diese direkt ohne Konflikte verarbeitet werden kann, da zum Zeitpunkt der Erstellung der Änderung der SLS und der Jupiter-Server auf dem gleichen Datenstand basieren. Bei Fall 3.3 (6.c) wird wieder eine Situation beschrieben, in der dem SLS eine Änderung gemeldet wurde, welche mehr als eine Änderung vom Stand des SLS divergiert beziehungsweise auf einem falschen Datenstand basiert. In diesem Fall muss erst wieder eine Konfliktlösung durchgeführt werden (6.c.i), bevor mit dem nächsten Schritt fortgefahren werden kann. Der letzte Fall 3.4 (6.d) sollte praktisch nicht oder nur sehr selten auftreten und wird theoretisch behandelt, wobei der SLS mehr Änderungen vom Jupiter-Server verarbeitet hätte als der Jupiter-Server selbst. In diesem Fall muss sowohl der SLS als auch der Saros-VSCode Klient mit dem Datenstand des Jupiter-Servers überschrieben werden (6.d.i) und der Prozess endet nach der Entfernung der Blockierungen (14.) und Überprüfung der Warteschlangen (15.).

Nachdem der Jupiter-Server Zähler verglichen wurde, muss im nächsten Schritt der Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar, mit dem des SLS Zählerpaar verglichen werden (7.). Dabei können folgende Fälle auftreten:

- Fall 4.1: Der Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar ist gleich dem Saros-VSCode Zähler vom SLS Zählerpaar
- Fall 4.2: Der Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar ist um mindestens einen Status höher als der Saros-VSCode Zähler vom SLS Zählerpaar
- Fall 4.3: Der Saros-VSCode Zähler vom SLS Zählerpaar ist um mindestens einen Status höher als der Saros-VSCode Zähler vom Jupiter-Server VSCode Zählerpaar

Auch diese Fälle sind wieder mit den Fällen 1.1 – 1.3 vergleichbar. Bei Eintritt des Falls 4.1 (7.a) kann der SLS mit der Bearbeitung fortfahren, da die Änderung vom Jupiter-Server auf dem gleichen Saros-VSCode Datenstand basiert, wie der SLS. Es treten dabei keine Konflikte auf. Fall 4.2 (7.b) sollte in der Praxis wieder nie auftreten, da der Jupiter-Server in der Theorie eine Änderung vom Saros-VSCode Klienten verarbeitet hätte, welche nicht vom SLS verarbeitet wurde. In einem solchen Fall gibt es keine andere sinnvolle Möglichkeit, als den Datenstand des Saros-VSCode Klienten und den des SLS mit dem Datenstand des Jupiter-Servers zu überschreiben (7.b.i). Der Prozess würde anschließend nach dem Aufheben der Blockierungen des SLS (14.) und der Überprüfung der Warteschlangen (15.) enden. Im letzten möglichen Fall 4.3 (7.c), würde die Änderung vom Jupiter-Server auf einem älteren Datenstand des Saros-VSCode Klienten basieren, als er beim SLS mittlerweile ist. Es wurde also vom SLS bereits eine neue Änderung des Saros-VSCode Klienten verarbeitet, welche zum Zeitpunkt der Erstellung der Änderung beim Jupiter-Server diesem noch nicht bekannt war. In einem solchen Fall muss erst eine Konfliktlösung mit operational transformation durchgeführt werden (7.c.i), bevor mit dem nächsten Schritt fortgefahren werden kann.

Anschließend wird die Änderung am SLS Dateisystem und SLS Zählerpaar angewandt (8.). Bis zu diesem Zeitpunkt ist der Ablauf vergleichbar mit einer Änderung vom Saros-VSCode Klienten. Ab dem nächsten Schritt weicht das Konzept jedoch vom anderen Ablauf ab, weshalb für eine verständlichere Erklärung des Konzeptes der SLS in die beiden bereits bekannten Themen SLS VSCode Bereich und SLS Jupiter-Server Bereich aufgeteilt wurde. Zum jetzigen Zeitpunkt ist die Verarbeitung der Änderung im SLS Jupiter-Server Bereich abgeschlossen und der SLS VSCode Bereich ist nun für die Weiterverarbeitung und Aufbereitung der Änderung für den Saros-VSCode Klienten verantwortlich.

Anstatt die Änderung zur Durchführung an den Saros-VSCode Klienten zu übermitteln, wird im nächsten Schritt durch den SLS VSCode Bereich geprüft, ob unverarbeitete Änderungen in der SLS-VSCode Warteschlange vorhanden sind (9.). Wenn die Warteschlange leer ist (9.a), also keine ausstehenden Änderungen vom Saros-VSCode Klienten vorhanden sind, kann mit dem nächsten Schritt fortgefahren werden. Wenn in der Warteschlange jedoch Änderungen vorhanden sind (9.b), müssen diese erst in aufsteigender Reihenfolge des Zeitstempels mit entsprechender Konfliktlösung auf den aktuellen Datenstand des SLS und das SLS Zählerpaar mit der bereits hinzugefügten Änderung vom Jupiter-Server angewandt werden (9.b.i). Dies hat den Hintergrund, dass der Saros-VSCode Klient, speziell VSCode mit seiner *Extension API* durch

seine asynchrone Arbeitsweise bereits während der Verarbeitung auf einem neuen Stand sein kann. Da in diesem Konzept der SLS die Transformation der Änderungen für den Saros-VSCode Klienten übernimmt, müssen somit externe Änderungen immer auf den aktuellen VSCode Datenstand angewandt werden. Dieser kann zu dem Zeitpunkt durch die Änderungen in der SLS-VSCode Warteschlange aktueller sein als dem SLS zum Zeitpunkt der Verarbeitung der Jupiter-Änderung bekannt ist. Dadurch muss der Datenstand des SLS vor Anwendung der Jupiter-Änderung am Saros-VSCode Klienten auf den Datenstand des Saros-VSCode Klienten gebracht werden. Nach Abschluss dieses Schrittes ist das Ziel, dass der SLS auf dem gleichen Datenstand wie der Saros-VSCode Klient ist und zusätzlich die Konfliktauflösung mit der neuen Änderung vom Jupiter-Server für den Saros-VSCode Klienten übernommen hat, nach aktueller Datenlage erreicht.

Ist dieser Schritt abgeschlossen, wird die konfliktfreie Änderung vom Jupiter-Server inklusive der SLS-VSCode Dokumentenversion mit LSP vom SLS an den Saros-VSCode Klienten übermittelt (10.). Da VSCode, wie bereits mehrfach erwähnt, asynchron arbeitet und es somit möglich ist, dass während der Übertragung der Änderung der Nutzer*innen zum Beispiel eine weitere Änderung gemacht hat, können zum Zeitpunkt des Anwendens der Änderung durch den Saros-VSCode Klienten trotzdem Inkonsistenzen entstehen. Aus diesem Grund werden in diesem Konzept, anders als normalerweise bei einer Verbindung zwischen einem Klienten und dem Jupiter-Server, Statusnachrichten eingebaut, welche nach erfolgreicher oder nicht erfolgreicher Verarbeitung vom Saros-VSCode Klienten an den SLS zurückgeschickt werden müssen. Dadurch hat der SLS VSCode Bereich die Möglichkeit, bei Ablehnung oder fehlenden Bestätigung durch den Saros-VSCode Klienten noch einmal zu reagieren, um den Verlust von Änderungen zu vermeiden. Der Saros-VSCode Klient prüft hierfür vor dem Anwenden der Änderung, ob die aktuelle VSCode Dokumentenversion und die mitgelieferte SLS-VSCode Dokumentenversion identisch sind (11.). Wenn sie es sind (11.a), bedeutet das, dass der Saros-VSCode Klient die Änderung auf den gleichen Datenstand anwendet, wie beim SLS während der Verarbeitung und Transformation der Änderung. Daher kann nun die Änderung angewandt (11.a.i) und anschließend eine Bestätigungsnachricht an den SLS geschickt werden (11.a.ii). Wenn die Versionen jedoch ungleich sind (11.b), würde die Änderung vom Saros-VSCode Klienten auf einem anderen Datenstand angewandt werden, wie er zum Zeitpunkt der Verarbeitung dem SLS bekannt war. Da somit eventuelle Konflikte auftreten können, welche der OT-Algorithmus im SLS nicht gelöst hat, wird die Änderung im Klienten verworfen (11.b.i) und eine Ablehnungsbenachrichtigung an den SLS geschickt (11.b.ii).

Während dieser Zeit wartet der SLS auf eine entsprechende Statusnachricht (12.). Wenn der SLS die Bestätigung über eine erfolgreiche Anwendung der Änderung vom Jupiter-Server im Saros-VSCode Klienten erhält (12.a), muss dieser nun die während der Verarbeitung der Änderung vom Jupiter-Server verarbeiteten Änderungen vom Saros-VSCode Klienten sortiert nach dem hinterlegtem Ankunftsdatum an den Jupiter-Server senden (12.a.i). Dabei werden die transformierten Änderungen verschickt, welche bereits auf der neuen Änderung vom Jupiter-Server basieren. Anschließend kann dieser die Blockierung des SLS und seiner Ressourcen beenden (14.) und die Warteschlangen nach neuen Änderungen geprüft werden (15.), wobei diese

in aufsteigender Reihenfolge des Zeitstempels über beide Warteschlangen abgearbeitet und anschließend dort entfernt werden. Sollte dabei eine Änderung aus der SLS Jupiter-Server Warteschlange exakt denselben Zeitstempel wie eine Änderung aus der SLS-VSCode Warteschlange aufweisen, wird die Änderung des Jupiter-Servers wieder zuerst abgearbeitet.

Sollte der SLS jedoch eine Benachrichtigung über eine Ablehnung oder nach einer gewissen Zeit noch keine Statusmeldung erhalten haben (12.b), ist es am wahrscheinlichsten, dass der Saros-VSCode Klient bereits einen neueren Datenstand als der SLS zum Zeitpunkt der Verarbeitung der Änderung vom Jupiter-Server aufweist. In diesem Fall prüft der SLS VSCode Bereich, ob bereits neue Änderungen vom Saros-VSCode Klienten in der SLS-VSCode Warteschlange vorhanden sind (13.). Wenn eine neue Änderung vorhanden ist (13.b), wird wieder an dem Punkt des Konzeptes eingestiegen, wo der SLS VSCode Bereich die neuen Änderungen auf die aktuelle Änderung des Jupiter-Servers anwendet und diese neue und wieder konfliktfreie Änderung anschließend an den Saros-VSCode Klienten mit ausstehender Statusnachricht sendet (9.b). Wenn die SLS-VSCode Warteschlange jedoch noch leer ist (13.a), sieht das Konzept vor, dass der SLS VSCode Bereich noch einmalig einen definierten Zeitrahmen wartet (13.a.i und 13.a.i.A), bis die Prüfung der SLS-VSCode Warteschlange ein letztes Mal durchgeführt wird (13.). Sollte sich hierbei anschließend doch eine Änderung in der Warteschlange befinden (13.b), wird diese wieder angewandt (9.b). In diesem Fall hat das Warten den Fall abgefangen, dass die Statusnachricht eventuell vor der eigentlichen, neuen Änderung vom Saros-VSCode Klienten den SLS erreicht hat und deswegen bei der ersten Prüfung noch nicht in der Warteschlange gefunden wurde. Wie lang genau der SLS dabei warten soll, muss dann bei der eigentlichen Implementierung des Konzeptes geklärt werden. Die Werte bewegen sich dabei aber in einem Rahmen von wenigen Millisekunden, bis maximal wenigen Sekunden. Zusätzlich muss, wenn die Warteschlange gefüllt ist, die Variable welche speichert, ob der Timer bereits einmal abgewartet wurde, zurückgesetzt werden. Wenn in einem Durchgang bereits einmal der Timer abgewartet wurde und anschließend tatsächlich eine Änderung in der SLS-VSCode Warteschlange vorhanden ist, wird diese bekanntlich ebenfalls angewandt. Wenn die neue berechnete Änderung anschließend erneut, aufgrund einer neuen Änderung, vom Saros-VSCode Klienten abgelehnt wird, soll im gleichen Durchgang erneut auf den Timer gewartet werden. Damit hierbei der bereits abgewartete Timer der vorherigen Änderung im gleichen Durchgang kein Problem darstellt, muss er im Fall, dass er bereits erfolgreich verwendet wurde zurückgesetzt werden (13.b.i).

Wenn bei der letzten Prüfung jedoch noch immer keine neue Änderung in der SLS-VSCode Warteschlange vorhanden ist (13.a.ii), muss der Datenstandes vom Saros-VSCode Klienten mit dem aktuellen Datenstand vom SLS, inklusiver der bereits neu verarbeiteten Änderungen, überschrieben werden (13.a.ii.A). Damit würde der/die Nutzer*in des Saros-VSCode Klienten zwar wahrscheinlich eine Änderung verlieren, jedoch sind dann die Datenstände des Klienten, mit dem des SLS wieder synchron und es können neue Änderungen verarbeitet werden. Der Prozess ist an dieser Stelle wieder zu Ende und der SLS kann die Blockierung aufheben (14.) und die Warteschlangen prüfen (15.). Nachdem nun einmal alle Ablaufszenarien vollumfänglich beschrieben und mögliche Konfliktfälle erläutert sind, wird im nächsten Kapitel der

zu Beginn dieser Arbeit beschriebene, erste Problemfall noch einmal im Detail mit der neuen Lösung betrachtet.

9.4 Betrachtung des Problembeispiels

Um abschließend klären zu können, ob mit dem neu entwickelten Konzept die im Verlauf der Arbeit mehrfach dargestellten Problematiken vollständig gelöst sind, wird im folgenden Kapitel das Problembeispiel aus Kapitel 6 noch einmal auf das neue Konzept angewandt. Das Hauptproblem in dem beschriebenen Beispiel lässt sich dabei allgemein so zusammenfassen, dass vom Saros-VSCode Klienten eine Änderung auf Basis eines alten Standes an den SLS versendet wurde, während dieser gerade eine andere Änderung vom Jupiter-Server verarbeitet hatte und somit einen neueren Datenstand als der Saros-VSCode Klient aufwies. Zusätzlich wurde die Änderung vom Jupiter-Server vom SLS an den Saros-VSCode Klienten zeitgleich mit der Änderung vom Saros-VSCode Klienten an den SLS versendet. Durch die fehlende Kontrolle beim Saros-VSCode Klienten resultierten am Ende unterschiedliche Datenstände im Saros-VSCode Klient und dem SLS. Wenn nun das Beispiel erneut betrachtet wird, geht das eben neu vorgestellte Konzept mit diesem Fall wie folgt um.

Gegeben ist ein Szenario mit zwei Nutzer*innen, in der Alice als Host mit Saros für Eclipse und Bob Saros für VSCode, also den Saros-VSCode Klienten verwendet. Zusätzlich können für eine bessere Verständlichkeit des Ablaufes in dem Beispiel die Handlungen in den Aktivitätsdiagrammen in Abbildung 15 und 16 nachvollzogen werden[28]. Beide Nutzer*innen arbeiten dabei am gleichen Dokument mit dem Inhalt *'Hello world'*.

1. Alice beginnt mit der Änderung *' my unique* an Index 5, woraus sich der Inhalt *'Hello my unique world* für Alice ergibt. Diese Änderung wird in eine Jupiter-Aktivität übertragen und über den Jupiter-Server an den SLS von Bob geschickt.

2. Gleichzeitig arbeitet Bob am gleichen Text und fügt ein *'!'* an Index 11 hinzu, woraus sich für Bob *'Hello world!'* ergibt. Diese Änderung wird ebenfalls an den SLS geschickt.

3. Bevor der SLS Bobs Änderung erhält und bearbeitet, empfängt er die Änderung von Alice. In der vorherigen Beschreibung sendet der SLS nun die Änderung von Alice (*' my unique'* an Index 5) den Saros-VSCode Klient, wodurch sich bei Bob *'Hello my unique world!'* ergibt.

An diesem Punkt hätte anschließend die Änderung von Bob den SLS erreicht und wäre einfach angewandt worden, wodurch sich bei Alice die Änderung *'Hello my un!ique world'* ergeben hätte. In dem neuen Konzept wäre es dabei nie zu diesem Verhalten gekommen, da die Änderung bei Bob mit dem Inhalt *'Hello my unique world!'* nie angewandt worden wäre.

Während im ursprünglichen Prototyp alle beim Saros-VSCode eingetroffenen Änderungen vom SLS direkt angewandt worden wären, wird dies durch die Prüfung

[28] Aufgrund ihrer Länge wurden die Aktivitätsdiagramme in dieser Arbeit zur lesbaren Darstellung optimiert. Die originalen Aktivitätsdiagramme sind für eine bessere Lesbarkeit in Print- und Digitalform gestreckt je über eine und zwei Seiten in Anhang A.2, A.3, A.4 und A.5 zu finden

der Dokumentenversionen im Saros-VSCode Klient jetzt verhindert. Die Prüfung der Dokumentenversionen ist dabei der gefundene Mittelweg, um einerseits eine Konfliktlösung zwischen dem Saros-VSCode Klienten und dem SLS zu ermöglichen und andererseits die Berechnung der Konflikte in den SLS auszulagen. Da auch aufgrund der Asynchronität der *Extension API* zum Zeitpunkt des Eintreffens einer Änderung vom SLS beim Saros-VSCode Klient der Datenstand des Klients (in diesem Beispiel Bob), sich durch, im SLS noch unbekannte, Änderungen von Bob von dem Datenstand des SLS und somit auch der Änderung von Alice unterscheiden kann, musste im Saros-VSCode Klient eine weitere Prüfung etabliert werden. Die beste Möglichkeit einer unkomplizierten und verwendbaren Lösung stellte dabei die Prüfung der Dokumentenversionen dar.

Durch die erneute vorherige Prüfung der vom SLS mitgelieferten Dokumentenversion hätte der Saros-VSCode Klient erkannt, dass die empfangene Änderung nicht auf Basis des gleichen Datenstandes wie im Saros-VSCode Klienten berechnet wurde. Der Saros-VSCode Klient hätte darauf hin eine Ablehnung der Änderung an den SLS geschickt, worauf dieser erneut die SLS-VSCode Warteschlange geprüft hätte. Hierbei wäre auch ein weiteres Verhalten aus dem Problembeispiel von Kapitel 6 nicht aufgetreten. Wenn die Änderung von Bob den SLS erreicht, wäre diese nicht einfach angewandt und an Alice gesendet worden. Der SLS ist zum Zeitpunkt der Ankunft von Bobs Änderung noch blockiert, da er noch auf die Bestätigung des Saros-VSCode Klienten der vorherigen Änderung wartet. Die Änderung von Bob wäre somit in die SLS-VSCode Warteschlange geschrieben worden. Wenn dann anschließend die Ablehnungsnachricht vom Saros-VSCode Klienten den SLS erreicht, prüft der SLS sogar noch einmal diese Warteschlange und im Idealfall wird Bobs später eingetroffene Änderung noch angewandt.

Anschließend kann die auf Bobs neuen Datenstand berechnete Änderung von Alice erneut an den Saros-VSCode Klienten gesendet werden, wo sie unter der Bedingung, dass Bob keine weitere Änderung erzeugt hat, auch erfolgreich angewandt wird. Dadurch gehen einerseits selten Änderungen verloren und andererseits können durch die Verwendung von operational transformation die entsprechenden Änderungen auf Basis der identischen Datenstände berechnet werden, wodurch sich sowohl für Bob als auch für Alice zum Ende immer der gleiche Inhalt ergibt.

Durch die erneute Betrachtung des ursprünglichen Problemfalles kann gezeigt werden, dass das in diesem Kapitel vorgestellte Konzept zur Erhaltung der Konsistenz zwischen dem Saros-VSCode Klienten und dem SLS theoretisch funktioniert. Das neu vorgestellte Konzept kann daher als Lösung für die Architektur des SLS verwendet werden.

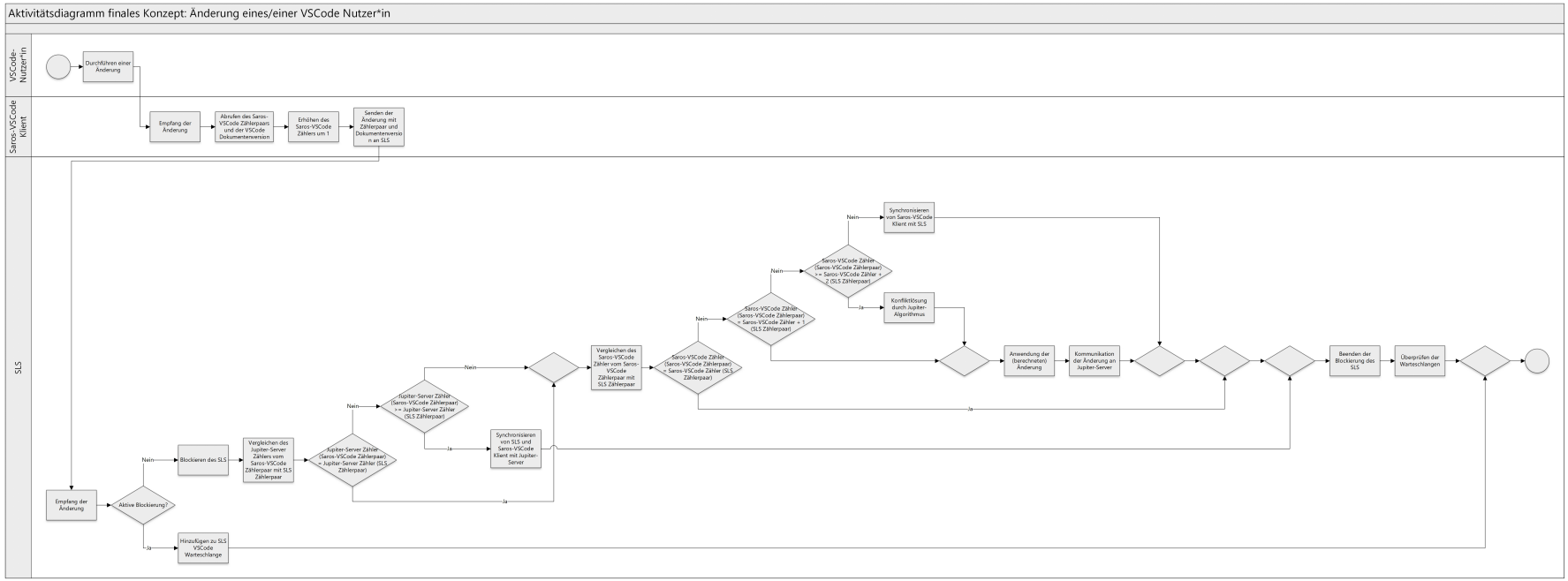

Abbildung 15: Aktivitätsdiagramm finales Konzept: Änderung von VSCode Nutzer*innen

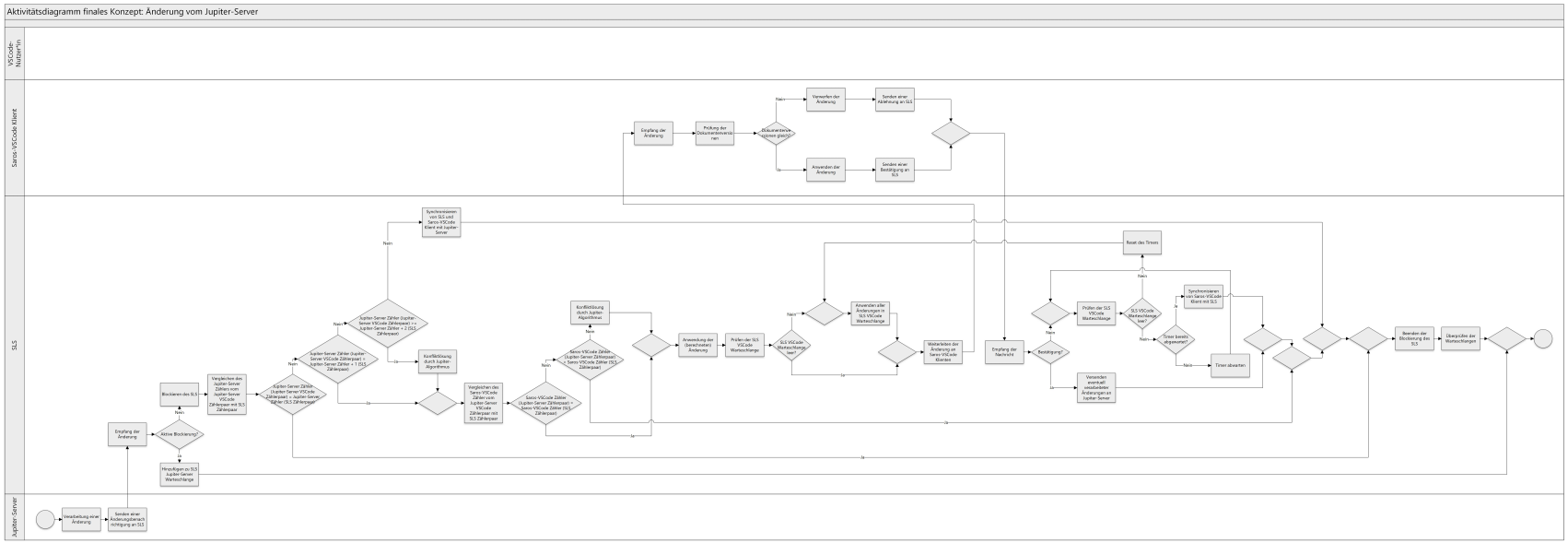

Abbildung 16: Aktivitätsdiagramm finales Konzept: Änderung vom Jupiter-Server

10 Fazit

Ziel dieser Ausarbeitung war es, den bestehenden Prototypen für die Einbindung von Visual Studio Code in die Paarprogrammierungs-Anwendung Saros zu analysieren, evaluieren und anschließend zu erweitern. Um einen besseren Einblick in die entsprechenden Themengebiete zu erhalten, wurde dabei mit einer Einführung in die allgemeine Architektur von Saros und einer Zusammenfassung der grundsätzlichen Ergebnisse und Zielstellungen aus der vorherigen Bachelorarbeit begonnen. Anschließend wurden noch die bereits in der Bachelorarbeit bekannten Probleme betrachtet. Nach Abschluss dieser Einarbeitung war damit einhergehend auch die dieser Ausarbeitung zugrunde liegende Ausgangssituation eingehend erläutert.

Darauf aufbauend konnten erste Analysen des Prototyps vorgenommen werden, in welchen neben einer allgemeinen Funktionsweise, beispielsweise der Implementierung der Nutzer*innenoberfläche, auch konzeptionelle Schwachstellen identifiziert wurden. Aufgrund der beschriebenen Auswirkungen der analysierten Architektur auf die Verhaltensweise des Prototyps, wurde der Fokus dieser Arbeit auf die Lösung der evaluierten Schwachstellen spezifiziert. Für ein besseres Verständnis der Hintergründe von Entscheidungen zu den im weiteren Verlauf der Arbeit entworfenen Konzepten und erkannten Problemen, wurden dabei einführend umfangreich die theoretischen Grundlagen zu operational transformation und Jupiter dargelegt. Basierend auf diesem Wissen konnte anschließend mit einer genauen Erläuterung zu der Problemstellung, sowie den ersten Lösungsansätzen fortgefahren werden.

Das erste entwickelte und analysierte Konzept basierte dabei auf der Idee, erzeugte Änderungen von dem/der Saros-VSCode Nutzer*in direkt wieder im Saros-VSCode Klienten zu entfernen und erst nach einer Verarbeitung durch den SLS erneut anzuwenden. Damit sollte die Erzeugung von Konflikten im Saros-VSCode Klienten verhindert und die Auflösung der Konflikte dem SLS zu überlassen werden. Dieser Lösungsansatz wurde dabei jedoch wieder verworfen, da unter anderem die Reaktionszeiten des implementierten Prototyps nicht zufriedenstellend waren und durch das kurzzeitige Verschwinden der Änderungen die Indexe der Änderungen verfälscht werden konnten. Das zweite entwickelte Konzept sah ebenfalls vor, die Konfliktlösung für den Saros-VSCode Klienten in den SLS auszulagern. Hierbei sollte dies jedoch unter anderem über verschiedene Datenstände im SLS und neu eingeführte Logs realisiert werden. Da allerdings auch mit diesem Konzept die aufgetretenen Probleme aufgrund der Asynchronität der *Extension API* nicht behoben werden konnten, wurde auch dieses Konzept wieder verworfen. Zusätzlich wurde von Teilnehmern der Projektgruppe von Saros angemerkt, dass durch die Verwendung standardisierter Verfahren aus Wirtschaft und Forschung an Stelle von Eigenentwicklungen eine bessere Wartbarkeit und Stabilität des Prototyps realisiert werden kann.

Bevor daher mit der Entwicklung eines neuen Konzeptes basierend auf den Erfahrungen der vorherigen Versuche begonnen werden konnte, wurde eine Analyse aktueller Technologien und Praktiken zum Synchronisieren mehrerer Datenstände vorgenommen. Dabei betrachtet wurde der Bereich der verteilten Datenbanksysteme, sowie Conflict-free Replicated Data Types inklusive verschiedener Umsetzungen und mögliche Konzepte. Basierend auf den Recherchen wurden anschließend die Bereiche mit operational transformation verglichen und sich abschließend für die Verwendung

einer Technologie für ein noch zu konzeptionierende, neue Architektur entschieden. Ausschlaggebend hierfür war die geringere Komplexität in Saros allgemein durch die Verwendung bereits implementierter Technologien und der nicht mehr notwendige Wissensaufbau bei den Entwickler*innen.

Aufbauend auf den Erfahrungen der verworfenen Lösungsansätze und dem gewonnenen Wissen durch die Recherchen, wurde abschließend ein neues Konzept entwickelt. Als Anforderungen hierfür wurde die Lösung des Problemfalls aus Kapitel 6, die weitere Verwendung des SLS, eine gleichbleibende Nutzer*innenerfahrung, eine Konfliktlösung nach Jupiter und die Verwendung von operational transformation identifiziert. Die bestehende Architektur wurde dafür um neue Komponenten, wie das Saros-VSCode und SLS Zählerpaar, die VSCode und SLS-VSCode Dokumentenversionen und die SLS VSCode und SLS Jupiter-Server Warteschlangen, ergänzt. Außerdem wurde sich dafür entschieden, nur einen Datenstand im SLS in Form des SLS Dateisystems zur Vermeidung weiterer Konflikte und Verringerung Komplexität einzusetzen. Die neuen Komponenten verfolgen dabei drei Ziele. Es sollte eine Konfliktlösung zwischen dem Saros-VSCode Klienten ermöglicht werden und die Implementierung der Konfliktlösung für den Saros-VSCode Klienten in den SLS ausgelagert werden. Um auftretende Probleme der asynchronen *Extension API* zu lösen, sollte zuletzt eine einfache und wirksame Prüfung der Datenstände der Änderungen durch die Verwendung der Dokumentenversion in das Konzept eingebaut werden.

Aufbauend auf den neuen Komponenten konnten in Kapitel 9 anschließend das Verhalten des neuen Konzeptes bei Änderung vom Saros-VSCode Klienten beziehungsweise vom Jupiter-Server detailliert mit allen auftretenden Fällen analysiert werden. Zum Ende der Ausarbeitung entstand hierbei ein Konzept, welches alle Vorteile und Erfahrungen der in den Kapiteln beschriebenen Ansätze vereint und die in den vorherigen Kapiteln beschriebenen Probleme löst. Anschließend wurde dies noch an der Anwendung des Problembeispiels aus Kapitel 6 gezeigt. Das entwickelte Konzept konnte dabei sowohl den Problemfall lösen, als auch die restlichen Anforderungen erfüllen. Es wird weiterhin der SLS in dem Konzept verwendet und eine Konfliktlösung mit operational transformation nach den definierten Regeln von Jupiter im SLS realisiert. Dabei ist weiterhin eine eventuelle Anbindung anderer Editoren und IDEs an den SLS möglich. Auch die Erfahrungen der Nutzer*innen bleiben gleich, da sich die Unterschiede im Konzept zu herkömmlichen Saros-Implementierungen vom Verhalten nur in minimal längeren Verarbeitungszeiten von Änderungen widerspiegelt. Damit kann abschließend festgestellt werden, dass die Problemstellung der Arbeit nach eingehender Analyse gelöst wurden und mit dem neuen Konzept Saros in Visual Studio Code integriert werden kann.

10.1 Kritische Betrachtung

Während der Bearbeitung dieser Abhandlung sind einige Arbeitsweisen durchaus kritisch zu betrachten. Einer der Kritikpunkte ist, dass von vornherein eine komplette und detaillierte Ausarbeitung der Probleme und eines funktionieren Konzeptes hätte erfolgen müssen, anstatt mit den ersten Entwürfen von Konzepten noch zu versuchen, schnelle Lösungen zugunsten einer Weiterverfolgung des ursprünglichen Themas dieser Ausarbeitung zu finden. Diese Arbeitsweise hat dabei, auch wenn viele

Probleme evaluiert werden konnten, welche in die finale Konzeption mit eingeflossen sind, zu Beginn der Ausarbeitung viel Zeit gekostet. Ebenfalls ist Zeit verloren gegangen, indem erst nach der Konzeptionierung der ersten Architektur Kontakt mit den Entwickler*innen von Saros direkt aufgenommen wurde. Dabei hätten die betrachteten Lösungsideen schon früher den entsprechenden Mitglieder*innen vorgestellt werden können, um die erst nach der Hälfte der Zeit erkannten Problematiken zum Beispiel mit der Wartbarkeit auch schon früher evaluieren zu können. Aufgrund der durch diese beiden Kritikpunkte fehlenden Zeit, konnte im Rahmen dieser Arbeit keine Implementierung des Konzeptes vorgenommen werden. Diese Tatsache stellt dabei auch den größten Kritikpunkt an der Ausarbeitung dar, da aus diesem Grund die theoretische Beweisführung nicht durch eine praktische Vorführung unterstrichen werden konnte.

10.2 Ausblick

Im Anschluss an diese Arbeit gibt es einige Schwerpunkte, auf Basis derer auch zukünftige Arbeiten aufbauen können. So ist die Implementierung des Konzeptes dabei von größter Importanz für dieses Projekt. Durch eine funktionierende Implementierung kann dabei auch der Weg für die Anbindung weiterer Editoren geebnet werden, wodurch auch Saros im Allgemeinen noch eine viel größere Verbreitung erfahren könnte. Zum jetzigen Zeitpunkt gibt es dabei keine vergleichbare Anwendung, welche viele Editoren und IDEs plattformübergreifend unterstützt. Eine Weiterführung dieses Konzeptes würde somit also auch einen Mehrwert für die Entwicklung des gesamten Projektes und eventuell auch für die vermehrte Nutzung von Paarprogrammierung in der Softwareentwicklung bringen.

Literaturverzeichnis

[1] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the case for pair programming," in *IEEE software*, vol. 17, no. 4, 2000, pp. 19–25.

[2] J. Schenk, L. Prechelt, and S. Salinger, "Distributed-pair programming can work well and is not just distributed pair-programming," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 74–83.

[3] Digital Business Magazin, "Entwicklerkonferenz microsoft build 2015: Höhepunkte der ersten keynote," 2015, accessed: Feb. 22, 2021. [Online]. Available: https://www.digitalbusiness-cloud.de/entwicklerkonferenz-microsoft-build-2015-hoehepunkte-der-ersten-keynote/

[4] I. Richter, "Microsoft build 2015: Highlights der ersten keynote," 2015, accessed: Feb. 22, 2021. [Online]. Available: https://news.microsoft.com/de-de/microsoft-build-2015-highlights-der-ersten-keynote/

[5] software.com, "Ranking the top 5 code editors in 2019," 2019, accessed: Feb. 23, 2021. [Online]. Available: https://www.software.com/src/ranking-the-top-5-code-editors-2019

[6] M. Schäfer, "Implementierung des saros-plugins für visual studio code," Bachelor's Thesis, Freie Universität Berlin, 2020.

[7] Saros, "Saros documentation," accessed: Feb. 28, 2021. [Online]. Available: https://www.saros-project.org/documentation/

[8] M. Corporation, "Extensions capabilities overview," accessed: Feb. 10, 2021. [Online]. Available: https://code.visualstudio.com/api/extension-capabilities/overview#no-dom-access

[9] ——, "Extension host," accessed: Apr. 01, 2021. [Online]. Available: https://code.visualstudio.com/api/advanced-topics/extension-host

[10] ——, "Extension api," accessed: Apr. 01, 2021. [Online]. Available: https://code.visualstudio.com/api

[11] ——, "Language server protocol," accessed: Apr. 08, 2021. [Online]. Available: https://docs.microsoft.com/de-de/visualstudio/extensibility/language-server-protocol?view=vs-2019

[12] ——, "Language server protocol specification - 3.16," accessed: Apr. 08, 2021. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/specification-current/

[13] M. Gala, "The enigma of collaborative editing," accessed: Apr. 15, 2021. [Online]. Available: https://medium.com/@mehulgala77/concurrent-collaborative-editing-d10192e55d2e#:~:text=Event%20passing%20(Operational%20transformation)%3A,from%20position%2010%20etc.).&text=Differential%20sync%3A%20The%20client%20can,server%20version%20and%20local%20copy

[14] C. A. Ellis and S. J. Gibbs, "Concurrency control in groupware systems," in *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, 1989, pp. 399–407.

[15] N. Fraser, "Differential synchronization," in *Proceedings of the 9th ACM symposium on Document engineering*, 2009, pp. 13–20.

[16] 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*, 1995, pp. 111–120.

[17] S. Agarwal, "How real-time collaborative editors work?" accessed: Apr. 24, 2021. [Online]. Available: https://www.srijanagarwal.me/2017/04/collaborative-editing/

[18] 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*, 1998, pp. 59–68.

[19] S. Agarwal, "Operational transformation, the real time collaborative editing algorithm [operations and transformations]," accessed: Apr. 25, 2021. [Online]. Available: https://www.srijanagarwal.me/2017/05/operational-transformation/

[20] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen, "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems," in *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 5, no. 1, 1998, pp. 63–108.

[21] D. Li and R. Li, "Preserving operation effects relation in group editors," in *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, 2004, pp. 457–466.

[22] R. Li and D. Li, "Commutativity-based concurrency control in groupware," in *2005 International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 2005, pp. 10–pp.

[23] 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*, 1996, pp. 288–297.

[24] S. Agarwal, "Analysing different operational transformation algorithms for collaborative editing," accessed: Apr. 25, 2021. [Online]. Available: https://www.srijanagarwal.me/2017/06/analysing-ot/

[25] D. Sun and C. Sun, "Context-based operational transformation in distributed collaborative editing systems," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 10, 2008, pp. 1454–1470.

[26] R. Li and D. Li, "A new operational transformation framework for real-time group editors," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 3, 2007, pp. 307–319.

[27] M. Van Steen and A. S. Tanenbaum, *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017.

[28] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," in *ACM computing surveys (CSUR)*, vol. 15, no. 4, 1983, pp. 287–317.

[29] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," in *Acm Sigact News*, vol. 33, no. 2, 2002, pp. 51–59.

[30] E. Brewer, "Towards robust distributed systems," 01 2000, p. 7.

[31] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," in *Computer*, vol. 45, no. 2, 2012, pp. 37–42.

[32] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating crdts for real-time document editing," in *Proceedings of the 11th ACM symposium on Document engineering*, 2011, pp. 103–112.

[33] C. Baquero, P. S. Almeida, and A. Shoker, "Making operation-based crdts operation-based," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2014, pp. 126–140.

[34] M. Heinrichs, "Verteilte daten ohne mühe: Conflict-free replicated data types," accessed: Mai. 11, 2021. [Online]. Available: https://m.heise.de/developer/artikel/Verteilte-Daten-ohne-Muehe-Conflict-Free-Replicated-Data-Types-3944421.html?seite=all

[35] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," Ph.D. dissertation, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[36] 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*, 2006, pp. 259–268.

[37] D. Sun, C. Sun, A. Ng, and W. Cai, "Real differences between ot and crdt in building co-editing systems and real world applications," *arXiv preprint arXiv:1905.01517*, 2019.

[38] D. Ivanov, "Practical demystification of crdts," accessed: Mar. 24, 2021. [Online]. Available: https://speakerdeck.com/ajantis/practical-demystification-of-crdts?slide=37

[39] "Collaborate in real time in atom," accessed: Mai. 12, 2021. [Online]. Available: https://teletype.atom.io/

[40] "teletype-crdt," accessed: Mai. 12, 2021. [Online]. Available: https://github.com/atom/teletype-crdt

[41] M. Corporation, "Textdocument," accessed: Mai. 14, 2021. [Online]. Available: https://code.visualstudio.com/api/references/vscode-api#TextDocument

A Anhang

A.1 Implementierung des ersten Konzeptes

Der Programmcode für den VSCode-Prototyp aus Kapitel 7.1 ist als Archiv in den eingereichten Dateien unter *Prototyp_Konzept_1.tar* zu finden.

A.2 Finales Aktivitätsdiagramm (Änderung eines/einer VSCode Nutzer*in) - Original

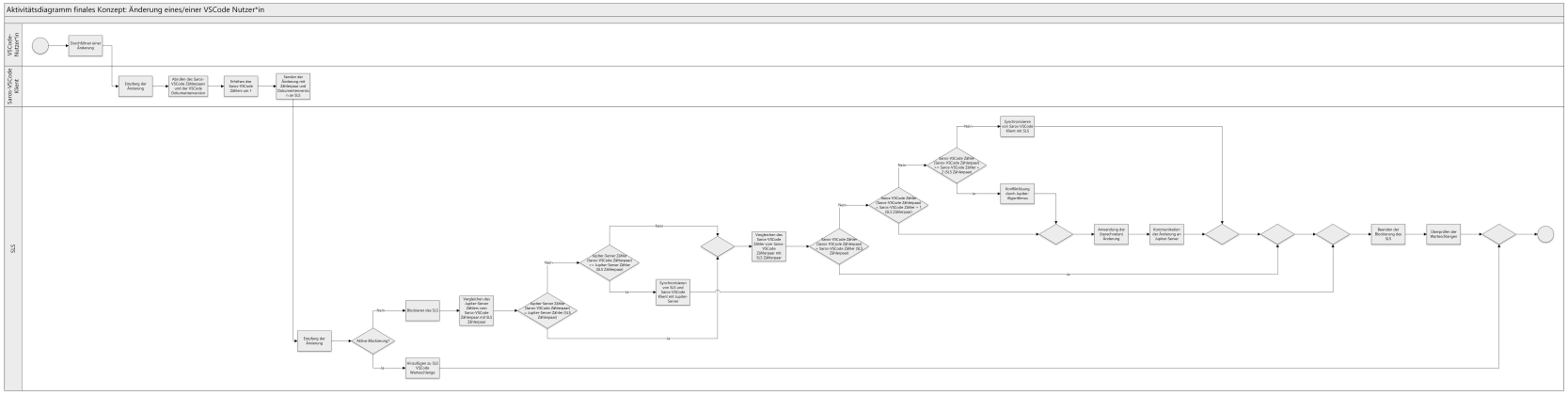

Abbildung 17: Aktivitätsdiagramm finales Konzept: Änderung eines/einer VSCode Nutzer*in

A.3 Finales Aktivitätsdiagramm (Änderung eines/einer VSCode Nutzer*in) über zwei Seiten

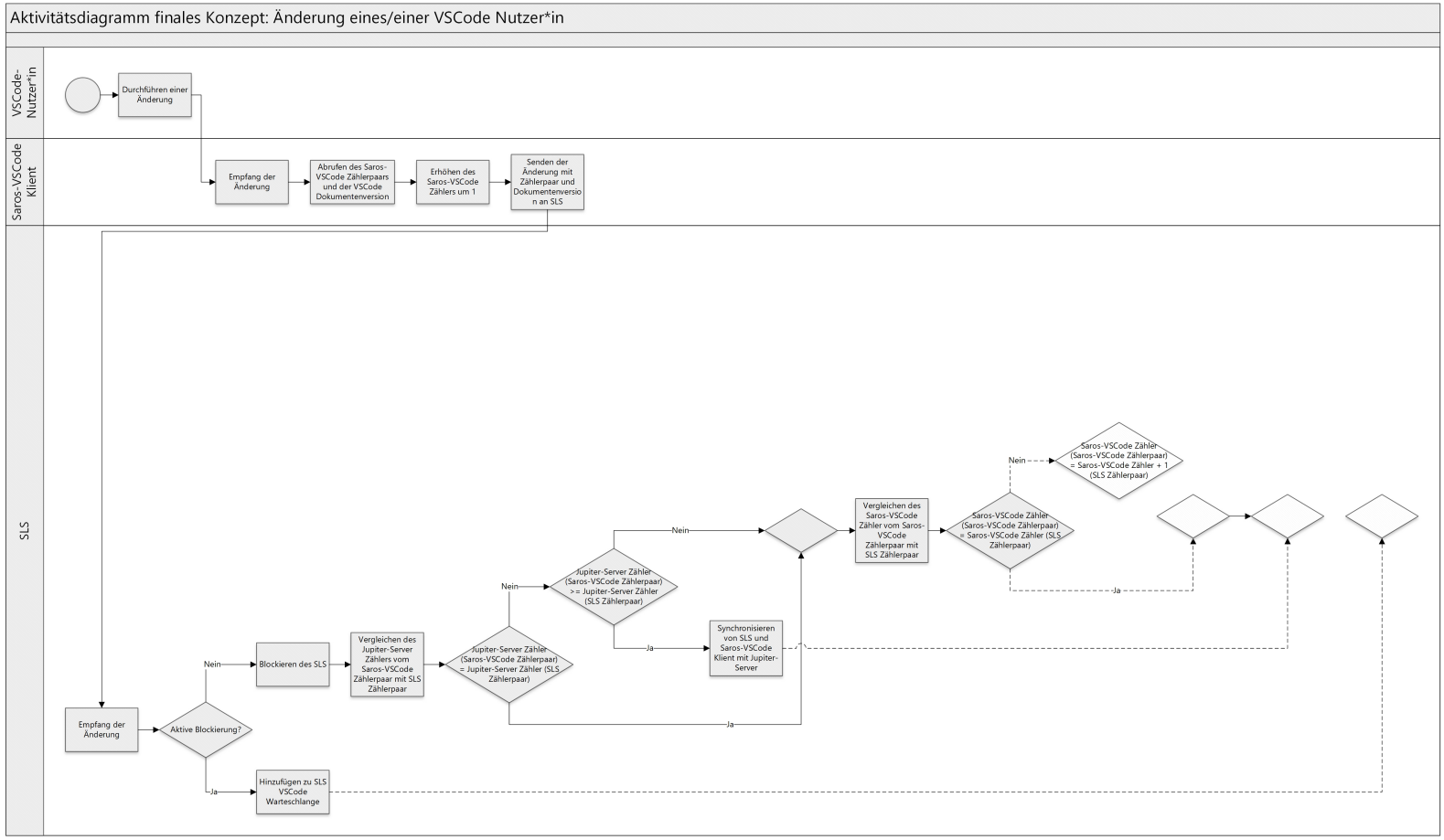

Abbildung 18: Aktivitätsdiagramm finales Konzept: Änderung eines/einer VSCode Nutzer*in Teil 1

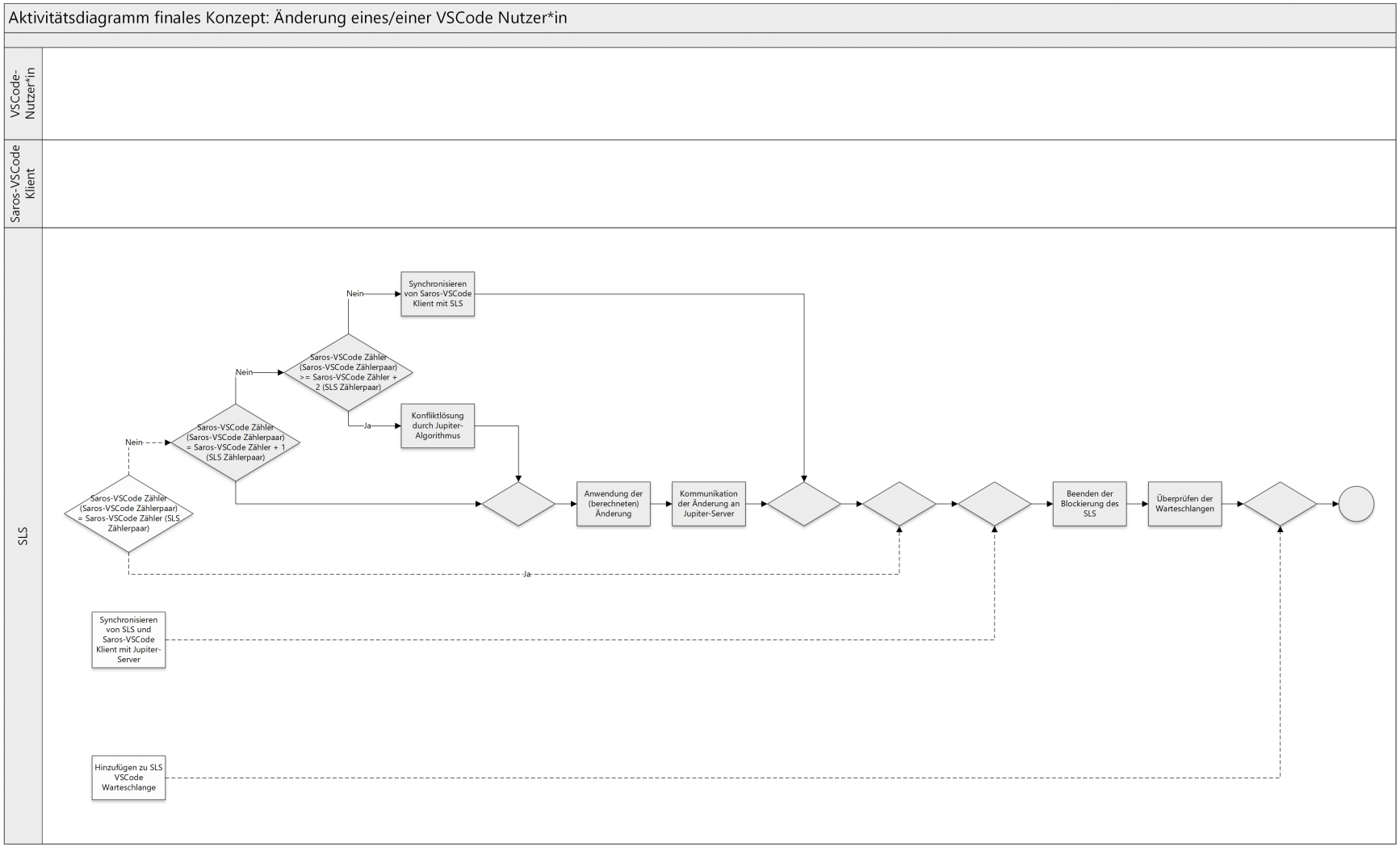

Abbildung 19: Aktivitätsdiagramm finales Konzept: Änderung eines/einer VSCode Nutzer*in Teil 2

A.4 Finales Aktivitätsdiagramm (Änderung vom Jupiter-Server) - Original

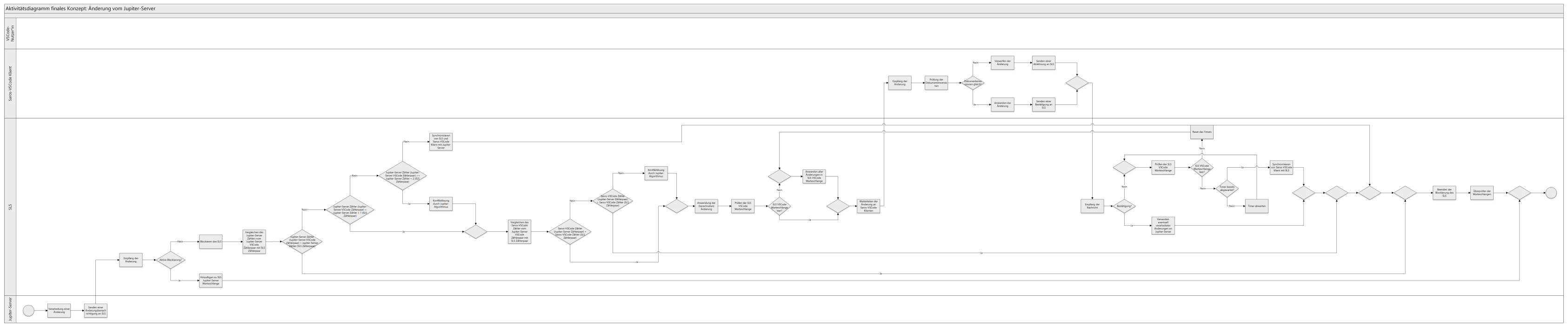

Abbildung 20: Aktivitätsdiagramm finales Konzept: Änderung vom Jupiter-Server

A.5 Finales Aktivitätsdiagramm (Änderung vom Jupiter-Server) über zwei Seiten

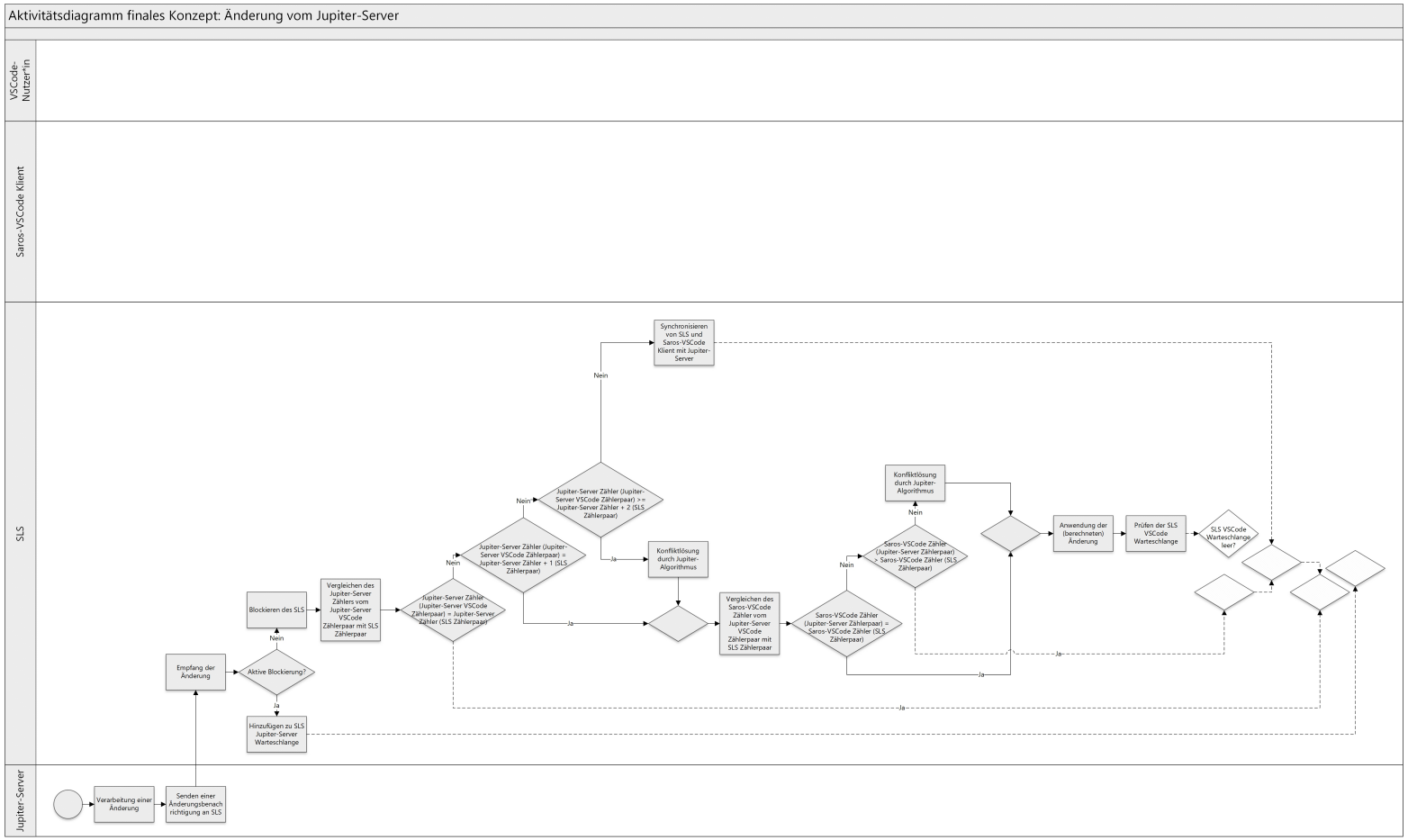

Abbildung 21: Aktivitätsdiagramm finales Konzept: Änderung vom Jupiter-Server Teil 1

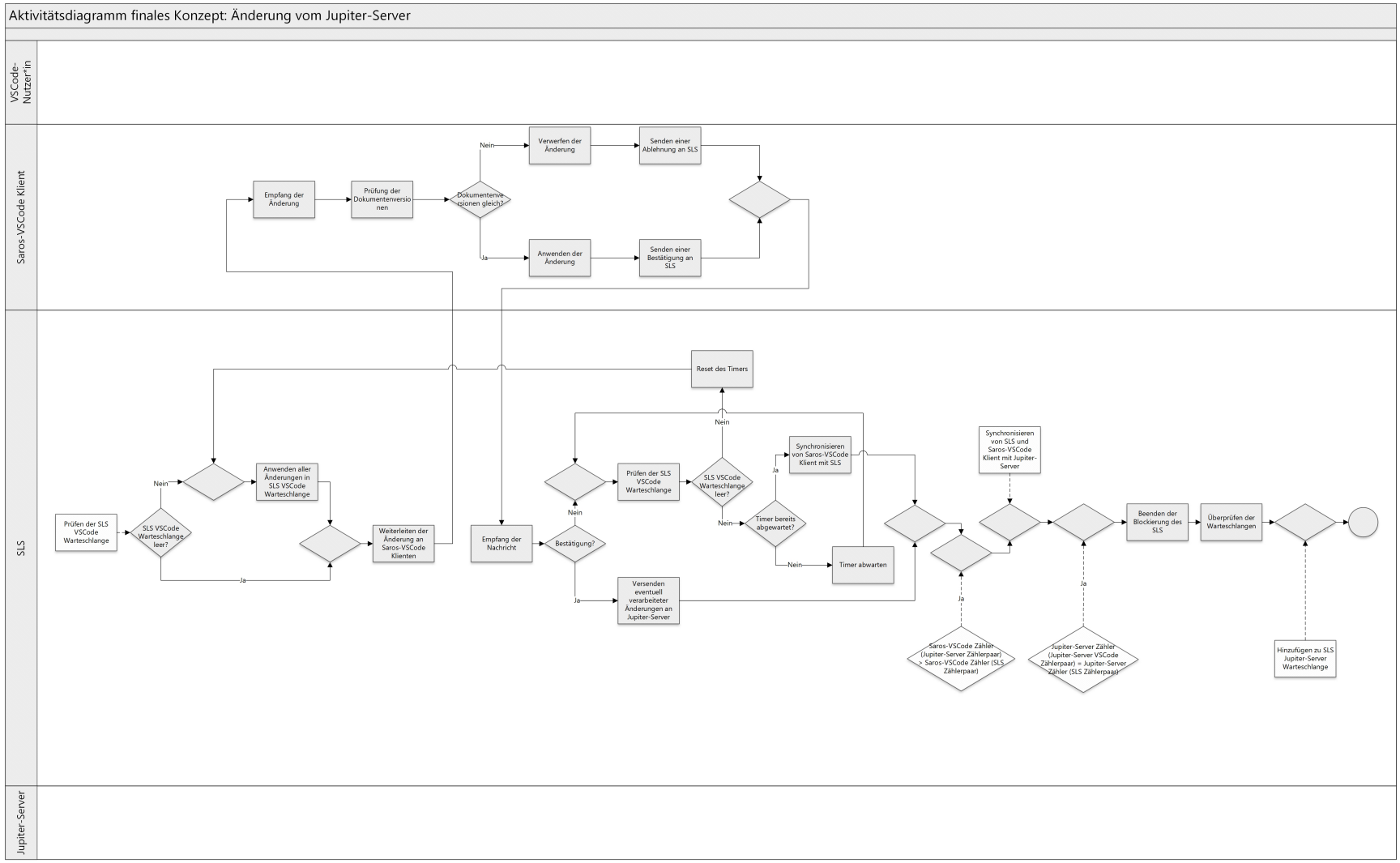

Abbildung 22: Aktivitätsdiagramm finales Konzept: Änderung vom Jupiter-Server Teil 2