Diplomarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Saros - Verbesserung des algorithmischen Kerns: Gleichzeitiges Editieren

Norman Warnatsch
Matrikelnummer: 4123733
norman@warnatsch.name

Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr. Elfriede Fehr
Betreuer: Prof. Dr. Lutz Prechelt

Berlin, 27.04.2012

Norman Warnatsch

Zusammenfassung

Saros ist ein Eclipse-Plugin zur Entwicklung eigener Software im Sinne des Distributed Party Programmings. Eine stetige Weiterentwicklung dieses Plugins erfolgt vorwiegend durch Abschlussarbeiten der Studenten der Freien Universität Berlin. Dadurch unterliegt Saros einem permanenten Wandel und einer stetigen Erweiterung.

Eine der größten und fundamentalsten Modifikationen stellte dabei die Umstellung vom Konzept der Distributed Pair Programming zum Distributed Party Programming dar, wobei das gleichzeitige Editieren mit mehreren Nutzern eine wichtige Position darstellt, die es zu überprüfen gilt. Nicht eingeplante Workflows, Deadlocks und Fehlverhalten entstanden vor allem durch Modifikationen des algorithmischen Kerns während der Realisierung einer Studienarbeit oder der neuen Erschaffung eines Features, wie zum Beispiel dem Need-Based Feature.

Auf Grundlage der Konzepte des Debuggings und der Codedurchsicht wird der Ablauf während des Editierens in Saros eingehend analysiert. Dabei folgt eine genauere Untersuchung der einzelnen, verwendeten Konzepte, um die Vor- und Nachteile, so wie ihre Wechselwirkung untereinander erfassen zu können.

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.

Berlin, 27.04.2012

Norman Warnatsch

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Beschreibung des Themas 1
1.2 Ziele der Arbeit 2
1.3 Vorgehensweise 2
1.4 Aufbau der Arbeit 3
1.5 Konventionen 3

2 Begriffsdefinition **4**
2.1 Fehlersuche 4
2.1.1 Debugging 4
2.1.2 Codedurchsicht 5
2.1.3 Technische Unterstützung 6

3 Grundlagen **8**
3.1 Saros 8
3.1.1 Saros Test-Framework 8
3.1.2 Architektur von Saros 9
3.1.3 Verwendung des Jupiter-Algorithmus 10
3.2 Netzwerkschicht von Saros 11
3.2.1 *XMPP*-Nachrichten 11
3.2.2 Übertragung der *XMPP*-Nachrichten 12

4 Verwendung von Debugging und Codedurchsichten **15**
4.1 Feature Need-Based 15
4.1.1 Funktionsweise der bisherigen Version 15
4.1.2 Grund zur Analyse des Features 16
4.1.3 Anwendung der Codedurchsicht 16
4.1.4 Überarbeiten des Features Need-Based 30
4.2 Watchdog 40
4.2.1 Momentaner Aufbau des Watchdogs 40
4.2.2 Erläuterung des Problems 41
4.2.3 Anwendung des Debuggings 41
4.2.4 Erste Analyse der Log-Dateien 42
4.2.5 Auftreten des Dialogs „File Changed! “ 43
4.2.6 Buffer für alle eintreffenden Aktivitäten 44
4.2.7 Stabilisierung des Buffers 44
4.2.8 Falscher Eindruck durch Debugging 45
4.3 Gleichzeitiges Editieren 46
4.3.1 Dokumentation zum Gleichzeitigen Editieren 46
4.3.2 Verarbeitung von Texteingaben 47
4.3.3 Erstellen von entsprechenden Tests 50

4.3.4 Vorgehen zur Fehlersuche 51
4.3.5 Analyse des Jupiter-Algorithmus 52
4.3.6 Erstellen von Aktivitäten 56
4.3.7 Versand der Aktivitäten 58
4.3.8 Versand von Aktivitäten über *XMPP*-Server 60
4.3.9 Empfang der Aktivitäten 61
4.3.10 Verarbeitung der Aktivitäten 64
4.3.11 Analyse der Fehlersuche 67
4.3.12 Grund für den auftretenden Fehler 68
4.3.13 Fehlerbehebung 70

5 Fazit **73**
5.1 Auswertung der Fehlersuche 73
5.2 Auswertung des gleichzeitige Editierens 73

6 Ausblick **75**

Abbildungsverzeichnis **77**

Literatur **78**

1 Einleitung

1.1 Beschreibung des Themas

Saros ist ein Eclipse-Plugin zur Entwicklung von Software nach dem Konzept des Distributed Party Programmings. Durch Abschlussarbeiten von Studenten der Freien Universität Berlin wird Saros permanent weiterentwickelt und um weitere Funktionalitäten ergänzt.

Aktuell wurden im Zeitraum der Diplomarbeit mehrere Projekte zur Modifikation des Whiteboards und der Usability (Verbesserung des Einladungsprozesses) aufgesetzt. Parallel dazu wurden Projekte mit den Titeln "Analyse und Evaluation von Verbesserungsmöglichkeiten der User Experience in Saros" und "Verbesserung der Architektur der DPP-Software Saros durch Einführung einer dokumentierten Modulsicht" abgeschlossen. Die von den Studenten vorgenommenen Aktualisierungen und Weiterentwicklungen besitzen teilweise gravierende Auswirkungen.
Durch neue Features kann und wird der algorithmische Kern modifiziert. Auf Grund der Anzahl geringer und auf den ersten Blick nicht gravierender Änderungen, die sich allerdings aufsummieren, erfolgte im Laufe der Zeit eine Destabilisierung des Kerns.

Ein weiterer Grund für diesen Effekt ist die in 2009 begonnene Umstellung, vom Konzept des Distributed Pair Programmings zum Konzept des Distributed Party Programmings.
Unter Pair Programming wird die gemeinsame Entwicklung an einem Projekt durch zwei Programmierer, zur gleichen Zeit verstanden. Dabei wird ein System verwendet, an dem zwei Entwickler zeitgleich zusammenarbeiten. Der Grundgedanke ist, dass Entwickler A den Code modifiziert, Entwickler B verfolgt und analysiert die Änderungen. Das Distributed Pair Programming bietet den Entwicklern die Möglichkeit, dieses Vorgehen räumlich getrennt voneinander zu bewerkstelligen. Durch den Einsatz von Saros ist es Entwicklern möglich, zeitgleich Änderungen am gleichen Code vorzunehmen.
Die Umstellung auf das Konzept des Distributed Party Programmings stellt dabei die Erweiterung auf eine beliebige Anzahl von Entwicklern dar. Um Inkonsistenzen bei Texteingaben zu verhindern, wurde deshalb der Jupiter-Algorithmus integriert. Die maximale Anzahl an Entwicklern in einer Session ist auf fünf begrenzt.

Zahlreiche Nutzer gaben an, Inkonsistenzen bei dem gleichzeitigen Editieren festzustellen. Daher wird eine Analyse der aktuellen Implementation von Saros vorangetrieben um den Grund für das Fehlverhalten zu extrahieren.

Mit Hilfe der Konzepte des Debuggings und der Codedurchsicht wird eine genaue Betrachtung der Verarbeitung während des Editierens vorgenommen. Auftretende Fehler werden, sofern möglich, behoben. Dabei wird die Wechselwirkung der einzelnen Konzepte aufeinander, sowie ihre Vor- und Nachteile, genauestens analysiert und in dieser Diplomarbeit festgehalten.

1.2 Ziele der Arbeit

Das primäre Ziel dieser Arbeit ist die Analyse, ob die existierende Implementierung für die Realisierung des gleichzeitigen Editierens den Anforderungen entspricht. Dabei soll als sekundäres Ziel eine Analyse der dazu verwendeten Konzepte Debugging und Codedurchsicht, sowie deren gegenseitige Beeinflussung, erstellt werden.

1.3 Vorgehensweise

Um die Probleme der Nutzer und die Instabilitäten des Kerns verstehen zu können, wurden zu Beginn der Diplomarbeit die im Bugtracker des Projektes Saros befindlichen Bugs sortiert und analysiert. Dabei wurde zwischen für diese Ausarbeitung relevanten und den irrelevanten Bugs unterschieden. Gleichzeitig erfolgte eine genaue Unterscheidung in Bezug auf die Aktualität der Bugs, da durch den permanenten Wandel von Saros nicht sichergestellt werden kann, dass erstellte Bugs nachfolgend reproduzierbar sind.

Es folgt die Erstellung standardisierter Tests um das Fehlverhalten zu erzeugen. Danach wurde die Ausführung der Tests auf unterschiedlichen Systemen vorangetrieben. Nachdem das Fehlverhalten erneut erzeugt werden konnte, wurden die Tests schrittweise modifiziert, so dass das erwartete Fehlverhalten weiterhin auftrat, unnötige Tätigkeiten bei dieser Reproduktion jedoch eliminiert werden konnten.

Wurde das gewünschte Fehlverhalten durch standardisierte, automatische Tests erfolgreich reproduziert, werden Konzepte des Debuggings und der Codedurchsicht angewendet, um die Quellen des Fehlverhaltens zu erfassen. Eine nachfolgende Analyse soll ermitteln, inwiefern die einzelnen Konzepte zur Lösung der gegebenen Probleme beitrugen.

Um die Konzepte korrekt analysieren zu können, werden vorher für je ein Konzept eine Problematik in dem Saros-Code genauer betrachtet. Zur Analyse der Codedurchsicht wird das Feature Need-Based untersucht, dem Debugging ist der fehlerhafte Watchdog gewidmet. Für die Problematik des gleichzeitigen Editierens werden beide Konzepte verwendet. Die entdeckten Fehlerquellen werden anschließend genauer analysiert und, wenn möglich, behoben.

1.4 Aufbau der Arbeit

Kapitel 2 erläutert Vorgehensweisen für das Debugging und die Codedurchsicht.

Im Kapitel 3 dieser Arbeit werden die Grundlagen von Saros, sowie deren Architektur und die Kommunikation zwischen mehreren Nutzern über XMPP erläutert.

Kapitel 4 beschäftigt sich mit der Analyse der Konzepte zur Lokalisation der Fehlerquellen und deren Behebung, die das Ziel dieser Arbeit waren, im Anschluss folgt eine Auswertung in Kapitel 5, so wie ein Ausblick in Kapitel 6.

1.5 Konventionen

Um dem Leser einen einfacheren Einblick in diese Diplomarbeit zu gewähren, werden Klassen, Funktionen und Variablen in `Schreibmaschinenschrift` verfasst, die *Kursivschrift* wird für Fachwörter und Abkürzungen verwendet. Der Namensraum aller Pakete, Klassen und Methoden ist `de.fu_berlin.inf.dpp`.

2 Begriffsdefinition

2.1 Fehlersuche

Da das fehlerhafte Verhalten in zahlreichen Bug-Einträgen dokumentiert wurde, bedarf es einer speziellen Fehlersuche, um die Schwachstellen beim gleichzeitigen Editieren zu erfassen und, wenn möglich, zu beheben. Dabei werden zwei Formen der Fehlersuche verwendet und analysiert, inwiefern sie einander beeinflussen.

2.1.1 Debugging

Im Allgemeinen bezeichnet Debugging das Konzept, einen während der Durchführung entdeckten Bug mit Hilfe von Tests zu lokalisieren und danach zu beheben [SCSP07].
Zu Beginn wird bei der Durchführung des Codes ein Fehlverhalten festgestellt. Es folgt der Ansatz, diesen Bug erfolgreich zu reproduzieren und Tests zu erstellen, die mit minimalem Aufwand das Fehlverhalten erneut aufrufen. Diese Tests werden dann zuweilen in automatisierte Tests überführt. Mit sogenannten Debuggern ist es im Folgenden möglich, Daten über den Zustand des Codes während der Durchführung zu erfassen, zu analysieren und mit Hilfe dessen gegebenenfalls die Fehlerquelle zu beheben.
Dabei stellen gerade verteilte Anwendungen, wie Saros selbst, die Ausführung des Debuggings vor Probleme [SCSP07] [BGoetz], da das Arbeiten mit Debuggern ausgeschlossen ist, denn diese können das nicht-sequentielle Verhalten entscheidend beeinflussen.

Delta-Debugging geht von einem konsistenten, funktionierenden Ausgangszustand einer Codebasis aus, die in regelmäßigen und kurzen Intervallen ausführlichen Tests unterzogen werden [DD12][SKleist]. Dadurch soll bei auftretenden Fehlern eine schnelle Extraktion der Quelle, also der dafür verantwortlichen Änderungen, stattfinden. Dies entspricht dem aktuellen Jenkins-Server, der täglich die Sammlung an automatisierten Tests ausführt und die Entwickler über Fehlschläge informiert. Um das Konzept des Delta-Debuggings optimal nutzen zu können, benötigt es einer funktionierenden Codebasis, was bei dem momentanen Projekt nicht der Fall ist. Es ist bekannt, dass der Saros-Code fehlerhaft ist, wodurch das Konzept des Delta-Debuggings nach [DD12] nicht angewandt werden kann. Der Grundgedanke [HHeit, S. 4], Codeabschnitte rekursiv aufzuteilen, wird bei der Fehlersuche im Abschnitt **Gleichzeitiges Editieren** angewendet, in dem der Ablauf zur Verarbeitung von Texteingaben in entsprechende, leichter zu betrachtende Abschnitte aufgeteilt wird.

Debugging mit Hilfe von Breakpoints stellt eine Möglichkeit dar, die Ausführung von Code durch das Setzen von Breakpoints während der Laufzeit zu stoppen und aktuelle Werte und Variablen auszulesen. Dieses Konzept stellt in den meisten Fällen ein nützliches Mittel zur Fehlersuche dar. In diesem Projekt kann es allerdings nicht verwendet werden, da die zu lösenden Bugs nur während der Ausführung mehrerer Instanzen und deren Kommunikation untereinander zufällig auftreten und erst nach Beendigung nachgewiesen werden können. Würden Breakpoints gesetzt, so würde das Erreichen eines Breakpoints eine Instanz von Saros anhalten, nicht jedoch die Anderen. Mögliche Vergleiche am Ende eines Tests würden prinzipiell zur Ungleichheit führen, da die betroffene, angehaltene Instanz die folgende Kommunikation der laufenden Instanzen nicht verarbeiten würde. Wäre es möglich, alle Instanzen bei einem Breakpoint einer Instanz zu stoppen, wäre auch hier eine Fortsetzung nicht möglich, da der Server keinen weiteren Kontakt zu den Usern hätte und diese als ausgeloggt wahrnehmen würde. Damit würde die bestehende Session als beendet betrachtet werden, eine weitere Verarbeitung wäre demnach nicht möglich.

Tracing Debugging stellt ein Konzept bereit, dass den Code um eigene Ausgaben erweitert. Bei korrekter Anwendung lassen sich dadurch Rückschlüsse auf das Verhalten der zu untersuchenden Codebasis schließen. Es existiert die Möglichkeit, automatisierte Tests durchzuführen und Informationen zu extrahieren, ohne das Laufzeitverhalten zu beeinflussen, wie es bei Debugging mit Hilfe von Breakpoints der Fall gewesen wäre.
Ein Nachteil stellt der entsprechende Mehraufwand dar, den diese Variante mit sich führt. Mittels einer gesonderten Betrachtung des Codes, sowie die nützliche Positionierung gewünschter Ausgaben im Code, kann ein Maximum an Informationen gewonnen werden. Dazu ist es unablässig, dass der Softwareablauf bekannt ist. Eine Analyse der Software-Struktur ist daher vor der Nutzung dieses Konzepts unumgänglich. Dennoch ist diese Variante die einzig Zuverlässige, die bei dem Projekt Saros angewendet werden kann, um spezielle Fehlerquellen durch Debugging aufzuspüren.

2.1.2 Codedurchsicht

Eine andere, entscheidende Möglichkeit Fehler aufzuspüren, ist das Konzept der Codedurchsicht[RevT12]. Das Ziel dieser Methodik ist es, durch Analyse des Quellcodes mögliche Fehler zu entdecken. Auch hier bedarf es einer genauen Kenntnis der vorhandenen Softwarestruktur, die es zu untersuchen gilt. Insbesondere sind die im Code unternommenen Abläufe von entscheidendem Interesse. Durch die Kenntnisse können die für diese Arbeit entscheidenden Abläufe extrahiert und die Menge des zu betrachtenden Codes drastisch reduziert werden. Das Prinzip der Codedurchsicht kann dabei in mehrere Arten unterteilt werden.

Informelle Reviews können lediglich als eine Art des "Gegenlesens" verstanden werden [SlideQS12]. Inhaltlich soll es dem technischen Review entsprechen [WikiCR12], sowie Zeit einsparen und ist als nicht formaler Prozess zu betrachten. Dabei erläutert der Autor seine Implementierung in einer Stellungnahme, während diejenigen, die den Code betrachten, Unklarheiten äußern und sich beantworten lassen.

Walkthroughs erfassen vorrangig, wie der Code arbeitet. Grundsätzlich dient diese Variante eher der Verständnis- und weniger der Fehlerfindung [SlideQS12]. Hier werden Probeläufe, sowie Szenarien, diskutiert.

Technische Reviews analysieren gegebene Dokumente, bzw. Code, ob sie mit einer vorhandenen Spezifikation übereinstimmen [WikiCR12]. Durch den permanenten Wandel des bestehenden Projektes und der geringen Spezifikation ist eine solche Durchsicht nicht möglich. Es können lediglich Teilabschnitte, wie der Jupiter-Algorithmus selbst, mit Hilfe dieser Variante untersucht werden. Andernfalls existieren keine zu verwendenden Spezifikationen, die die Verwendung des technischen Reviews rechtfertigen würden. Es erfolgt in dieser Variante ebenso eine Betrachtung, ob Coderichtlinien und Standards eingehalten wurden.

Inspektionen stellen die formalste Variante der Reviews dar und sind nach *IEEE 1028* standardisiert. Es folgt hier eine Überprüfung des Codes auf Mängel. Dazu können im Vergleich zum technischen Review vorher definierte Checklisten verwendet werden, um einen Überblick über die zu untersuchenden Punkte zu haben. Diese Variante wird favorisiert, da sie keinerlei vorhandene Spezifikation der Software erwartet.

2.1.3 Technische Unterstützung

Es blieb die Frage offen, ob technische Unterstützung durch andere Software verwendet werden könnte, um entsprechende Fehler zu entdecken. Darunter zählen statische Analyse Tools [BGoetz], die anhand von *Bug-Pattern* Detektoren [BPattern12] den Code analysieren, wie zum Beispiel FindBugs. Diese Analyse Tools stellen zwar eine Bereicherung zur Fehlersuche dar, sind allerdings auch bei komplexeren Projekten, wie es bei Saros der Fall ist, überfordert. Zahlreiche Durchläufe mit diesem Tool haben gezeigt, dass sie bei der weiteren Betrachtung und Fehlersuche in Saros keinerlei Nutzen bringen. Ihre markierten Fehler stellen sich zuweilen eher als einen unorthodoxen Programmierstil der Saros-Entwickler heraus, nicht aber als realer Fehler.

Ferner wurde bekannt, dass das Tool im Jenkins-Server integriert ist, was die eigene lokale Verwendung unnötig werden lässt. Stattdessen konnte auf die dort ausgegebenen Anmerkungen reagiert werden. Dabei stellte sich heraus, dass die angezeigten Fehler, die nicht Resultat eines unorthodoxen Programmierstil sind, außerhalb des Kerns dieser Diplomarbeit zu finden war, wie zum Beispiel im Bereich *Voice-Over-IP*, weshalb auch die weitere Verwendung ausgeschlossen werden konnte.

3 Grundlagen

3.1 Saros

Das Eclipse-Plugin Saros ermöglicht den Nutzern die Entwicklung im Sinne des Distributed Party Programmings. Die Zielgruppe dieses Projektes erhält durch Saros die Fähigkeit, räumlich getrennt an den scheinbar selben Dokumenten zu arbeiten und die Änderungen anderer Nutzer zeitnah wahrzunehmen.
Es wird dem Entwickler durch die in Saros integrierten Features von *Voice-Over-IP*, sowie einem Chatroom und dem experimentellen *Screensharing* ermöglicht, in Kontakt mit seinen Kollegen zu treten, um eine bessere Koordination bei der Entwicklung zu gewährleisten. Gleichfalls von entscheidender Bedeutung zur Koordination ist das *Whiteboard*, eine Möglichkeit, Skizzen zu erstellen und diese den Anderen zeitgleich präsentieren zu können.

Begonnen hat das Projekt Saros als Erweiterung der Eclipse Plattform zur Realisierung des Konzepts des Distributed Pair Programming, entwickelt in der Diplomarbeit von Riad Djemili im Jahr 2006 [Djemili] und wurde dann durch Sebastian Ziller im Jahre 2009 [Ziller] zum Distributed Party Programming erweitert. Es stellte sich dabei heraus, dass die Eingaben aller Nutzer während des gemeinsamen Editierens einer Datei einer besonderen Verarbeitung benötigen, um einen konsistenten Inhalt für alle Nutzer bereitzustellen.

Die Paarprogrammierung zeichnet sich dadurch aus, dass zwei Nutzer ein Projekt zusammen bearbeiten, aber vor allem auch die Veränderungen des Anderen leicht einsehen und hinterfragen können. Durch diese Methodik der Entwicklung entsteht schon während der Implementierung ein erstes Review, was der frühen Fehlerentdeckung dient und eine höhere Softwarequalität zur Folge hat. Gleichzeitig findet ein Wissensaustausch zwischen diesen beiden Nutzern statt, was für spätere Projekte von Vorteil ist. Der Zeitraum der Zusammenarbeit bei einem Projekt wird als Session bezeichnet. Der User, der die Session beginnt, wird als Host bezeichnet und hat alle Rechte während der Session. Er besitzt gleichfalls die Möglichkeit, anderen Usern Rechte zu entziehen. Das Distributed Party Programming ist die Ausweitung der Paarprogrammierung auf mehr als zwei Nutzer.

3.1.1 Saros Test-Framework

Nachdem das Projekt Saros einen Umfang annahm, der das manuelle Testen von Testfällen erschwerte, wurde das Saros Test-Framework erstellt. Mit Hilfe dieses Test-Frameworks können JUnit-Testfälle entwickelt werden, die den Zugriff eines Users simulieren.

Im Sinne dieses Frameworks werden mehrere Eclipse-Instanzen mit der aktuellen Version von Saros gestartet. Der JUnit-Test für das Saros Test-Framework (*STF*) verwendet die für das Test-Framework vordefinierten User (ALICE, BOB, CARL und DAVE), wobei jeder dieser User auf eine der gestarteten Eclipse-Instanzen, sowie deren Shells und Menüs, zugreifen kann. Die benötigten Zugangsdaten werden vom Tester manuell in einem vordefinierten Format in `configuration.properties` eingetragen.
Durch das Test-Framework lassen sich bis dahin als manuell ausgeführte Tests in *STF*-Tests überführen und automatisieren.

Genauso wie das Projekt Saros befindet sich auch dieses Framework in einem permanenten Wandel. Zur Realisierung mancher Testfälle bedarf es daher noch Modifikationen in der Codebasis des Frameworks.

3.1.2 Architektur von Saros

Die Architektur von Saros zu erfassen ist, bedingt durch die kontinuierliche Erweiterung durch die Studenten, nahezu unmöglich. Jedes neue, durchgeführte Release trägt Veränderungen in sich, die die Struktur Saros betreffen. Obwohl Saros ein Produkt von Informatik-Studenten ist, die im Zuge ihrer Abschlussarbeiten zu dem Code beitrugen, existieren zu dem aktuellen Stand keine Dokumentationen. Diplomarbeiten, die den algorithmischen Kern betrafen, boten zwar die Möglichkeit Grundstrukturen zu erfassen. Nachfolgende Arbeiten veränderten diese beiläufig, um neue Features und ähnliches in das Projekt einbauen zu können. Als Beispiel kann hier das Feature Need-Based betrachtet werden, dessen Einführung den algorithmischen Kern modifizierten, andererseits im gesamten Arbeitsprozess nicht dokumentiert wurde. Weder ist ein definiertes Soll-Verhalten vorhanden, noch sind Dokumentationen über den Aufbau hinterlegt worden. Auf Grund dessen war ein Studium gegebener Dokumentationen nicht möglich. Es musste auf die bestehenden Diplomarbeiten vertraut und dann über ein eigenes Studium eine Visualisierung der Struktur vorgenommen werden. Im Zentrum von Saros steht das gleichzeitige Editieren von Dateien eines Projektes, erweitert wurde Saros mit Features wie zum Beispiel *Voice-Over-IP*, *Multi-User Chat*, *Screensharing* oder das *Whiteboard*. In der Abbildung von Saros [Abbildung 1] wird dieser Kern als *Shared Session* dargestellt.

Der Fokus der Betrachtung liegt dabei auf dem Kern des gleichzeitigen Editierens [Abbildung 2], sowie dem *Network Layer*. Dieser befindet sich in dem Abschnitt *Concurrency Control* und dem Teilabschnitt *Session* in *Shared Session*.
Der Kern von Saros konvertiert von Eclipse empfangene Events, ausgelöst durch Texteingaben in *EditorParts* oder durch das Kreieren von Dateien, in Aktivitäten. Als *EditorPart* wird die Ansicht einer geöffneten Datei ver-

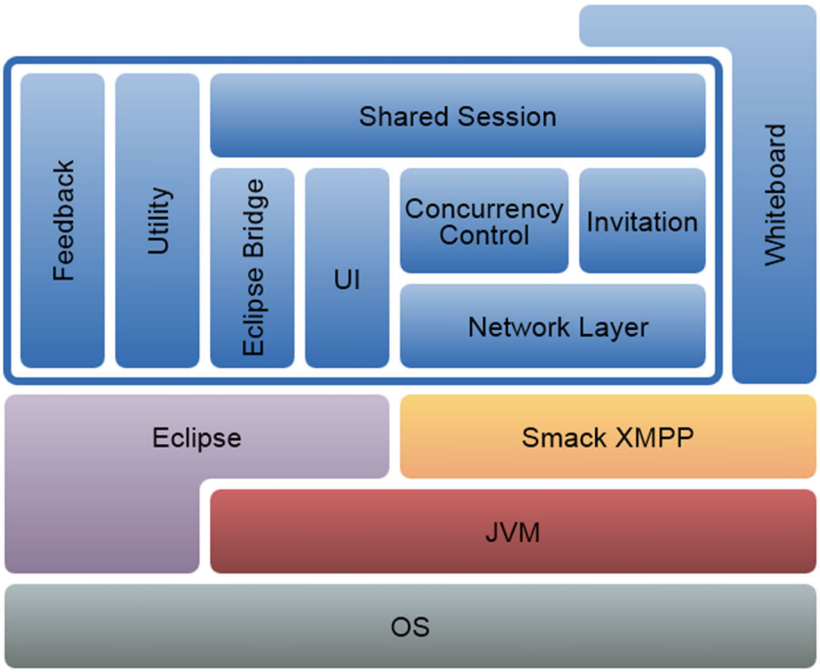

Abbildung 1: Überblick von Saros [STech12]

standen. Diese Aktivitäten werden entsprechend des Jupiter-Algorithmus transformiert und an den Host über *XMPP*-Nachrichten versandt. Der Host transformiert die empfangenen Aktivitäten und verschickt sie an die anderen User einer Session. Der Empfang dieser Aktivitäten erfolgt über einen *Bytestream*, dessen Inhalt in ein schlichtes Paket konvertiert werden muss. Dieses wird in eine XML-Nachricht umgewandelt, die wiederum bis in eine Aktivität konvertiert wird. Die dadurch empfangene Aktivität wird letztlich im entsprechenden *EditorPart*, bzw. in der entsprechenden Ansicht ausgeführt.

3.1.3 Verwendung des Jupiter-Algorithmus

Um die Architektur verstehen zu können, bedarf es zugleich der Kenntnisse über den verwendeten Algorithmus, mit dessen Hilfe jeder User eine konsistente Datenbasis aufweist, ganz gleich, wie viele User sich in der Session befinden.

Das Grundprinzip des Algorithmus sieht eine Sterntopologie mit mehreren Clients vor, in deren Zentrum ein Server steht.
Jeder Client besitzt die gleiche Datenbasis. Editiert ein Client seine Basis, sendet er eine entsprechende Aktivität an den Server, welcher die ankommenden Aktivitäten serialisiert und an die anderen Clients überträgt.

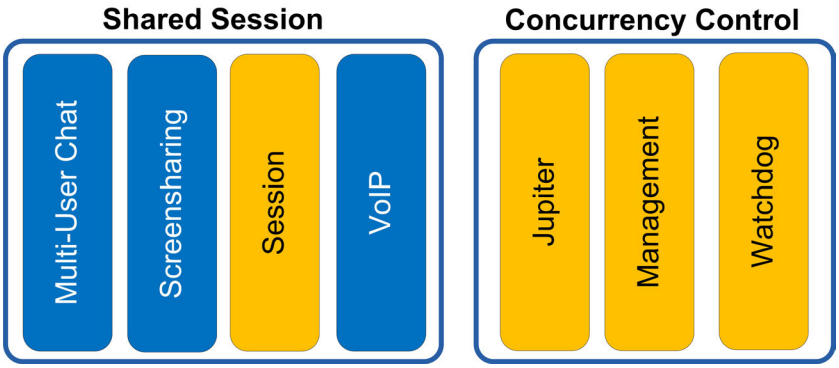

Abbildung 2: Überblick von *Shared Session* [SSession12] und *Concurrency Control*[SConc12]

Diese haben die Aufgabe, die eingehenden Aktivitäten entsprechend ihrer Datenbasis anzupassen [Ziller]. Die Clients unterscheiden demnach nur zwischen Eingaben, die lokal durchgeführt wurden, und jenen, die Andere ausführten.

Entgegen der ersten, möglichen Auffassung, hat der Server, mit dem sich die User verbinden, nichts mit dem Server für den Algorithmus zu tun. Der Host einer jeden Session verwaltet einen Jupiter-Server für die Serialisierung aller Aktivitäten der User in dieser Session und einen Client für seine eigenen Eingaben, die ebenfalls Aktivitäten an den Jupiter-Server senden.

3.2 Netzwerkschicht von Saros

Um Saros vollständig verstehen zu können, benötigt es des Wissens, wie Saros mit anderen Instanzen seiner selbst kommuniziert und in welchem Zusammenhang der Server, mit dem sich die User verbinden müssen, steht. Zur Kommunikation verwendet Saros *XMPP*-Nachrichten und bietet dazu drei Übertragungsformen an, die, in Abhängigkeit von der möglichen Geschwindigkeit der User, gewählt wird.

3.2.1 *XMPP*-Nachrichten

Das *Extensible Messaging and Presence Protocol (XMPP)* wird in Saros mit Hilfe der Bibliothek `Smack` zur Kommunikation verwendet. Dieser von der *Internet Engineering Task Force (IETF)* für den Nachrichtenaustausch veröffentlichte Standard (*RFC 6120, 6121, 6122, 3922, 3923*) entstand aus dem `Jabber` Protokoll. Der Aufbau dieser Nachrichten entspricht dem XML-Standard. Das Protokoll selbst erfährt eine Erweiterung durch die Bereitstellung von *XMPP Extension Protocols* durch die *XMPP Standards Foundation*.

Jeder Teilnehmer in einem *XMPP*-Netzwerk wird durch eine eindeutige *Jabber ID*, auch *JID* genannt, identifiziert [Gustavs]. Gleichzeitig vermag es die *JID*, eine sogenannte *Resource* zu beinhalten, um die Zielanwendung für die entsprechende Nachricht bestimmen zu können. Dies bietet den Vorteil, dass mehrere *XMPP*-Anwendungen gleichzeitig betrieben werden können und die jeweiligen Nachrichten an die korrekte Anwendung überführt werden.
Es werden drei Arten von *XMPP*-Nachrichtentypen, sogenannte *Stanzas*, beschrieben:

- *Message Stanza* ist die Bezeichnung für den Versand einer einzelnen Nachricht an einen Kontakt über den Server.
- *IQ Stanza* beinhaltet Nachrichten die Anfragen an andere Kontakte, mit resultierenden Antworten, darstellen.
- *Presence Stanza* beinhaltet die Nachrichten, welche Statusinformationen zum Netzwerkstatus, beziehungsweise zum Kontakt selbst übertragen.

3.2.2 Übertragung der *XMPP*-Nachrichten

Es existieren drei Arten der Übertragung von *XMPP*-Nachrichten [SMSK], wobei die Varianten *In-Band-Bytestream* und *Sock5 (direkt)* von Saros verwendet werden. Entgegen der allgemeinen Auffassung unterscheidet Saros bei der Wahl der Übertragung nicht zwischen den Daten, die es zu versenden gedenkt. Die Aktivitäten von Texteingaben können ebenso über *Sock5* versendet werden, wie auch ganze Projekte während dem Erstellen einer Session.

Zum besseren Verständnis, werden nachfolgend die Unterschiede zwischen den einzelnen Übertragungsarten näher erläutert.

In-Band-Bytestream [Abbildung 3] ist die Variante, die Daten in *Base64* zu konvertieren und als *Nachrichten* über den *XMPP*-Server zu versenden. Da die Server mitunter eine festgeschriebene Grenze besitzen, kann die Übertragungsrate auf einige wenige KB/s schwinden [Szücs]. Für kurze Nachrichten, wie zum Beispiel einzelne, vom Sender erzeugte Aktivitäten, ist diese Übertragungsform dennoch verwendbar.

Sock5 Bytestream (direct) [Abbildung 4] ist, entgegen des bisherigen Verständnisses, kein *Stream*-Kanal über einen *Sock5*- oder einen *XMPP*-Server, wie es in [Szücs] beschrieben wurde. Es wird zwar eine Anfrage vom Typ *IQ Stanza* von einem Sender an einen Empfänger über den *XMPP*-Server verschickt, jedoch lediglich um die *JID* und die Netzwerkadresse des Senders an den Empfänger zu übertragen [SMSK]. Der Empfänger öffnet

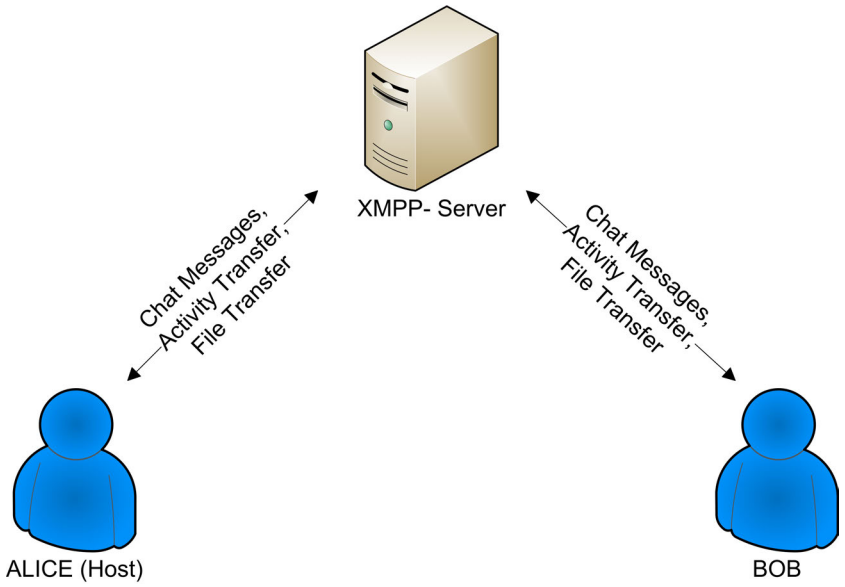

Abbildung 3: *In-Band*-Bytestreams

daraufhin eine TCP-Verbindung und erhält dann die gewünschten Datenpakete.

Sock5 Bytestream (mediated) [Abbildung 5] erfolgt über einen Sender, Empfänger und Host. Es wird davon ausgegangen, dass der Sender die *JID* und die Netzwerkadresse des Hosts kennt. Der Sender schickt eine Nachricht vom Typ *IQ Stanza* mit den Daten vom Host über den *XMPP*-Server zu den Empfänger. Dieser baut eine TCP-Verbindung zum Host auf, welcher eine Bestätigung bei erfolgreichem Verbindungsaufbau an den Empfänger verschickt. Der Empfänger überträgt ein *IQ Stanza* mit der ID des initiierenden *IQ Stanza* über den Server an den Sender. Der Sender baut daraufhin eine TCP-Verbindung zum Host auf. Nachfolgend werden die Datenpakete vom Sender über den Host zum Empfänger weitergeleitet.

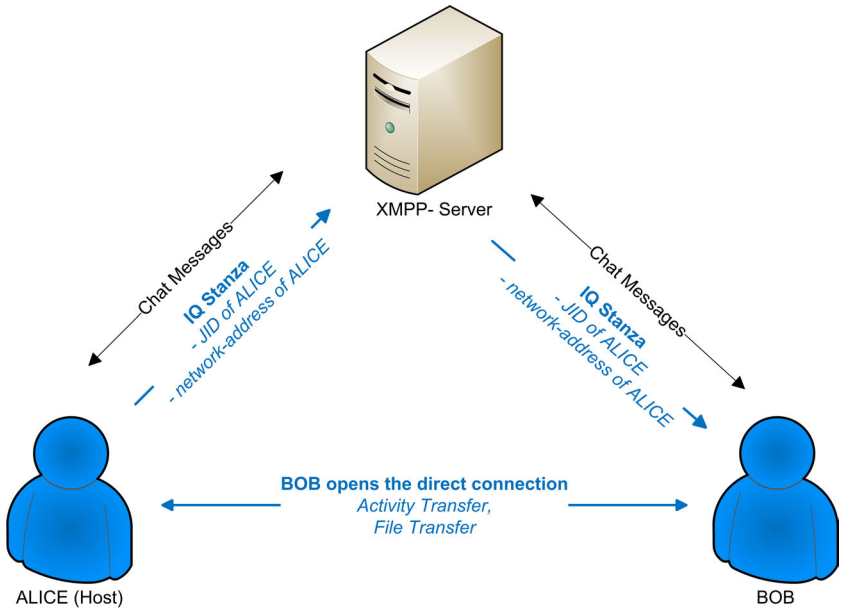

Abbildung 4: *Sock5*-Bytestreams (direct)

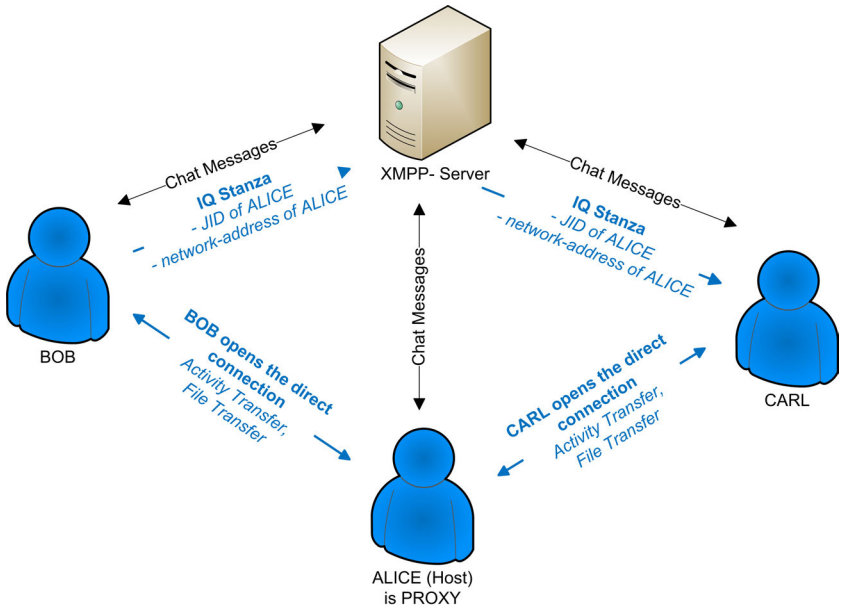

Abbildung 5: *Sock5*-Bytestreams (mediated)

4 Verwendung von Debugging und Codedurchsichten

In diesem Abschnitt wird die Vorgehensweise zur Lösung eines Bugs in dem Saros-Projekt genauer betrachtet. Begonnen wird bei der Überarbeitung des Need-Based-Features, in dem die Codedurchsicht zum Tragen kam, gefolgt von der Fehlersuche zur Stabilisierung des Watchdogs, in dem das Konzept des Debuggings Einfluss fand.

Das Hauptaugenmerk wird auf der Analyse des gleichzeitigen Editierens bei mehr als zwei Usern liegen. Um eine Analyse voranzutreiben, wird eine Kombination aus Debugging und Codedurchsicht verwendet. Von entscheidendem Interesse ist dabei die Wechselwirkung beider Konzepte zur Extraktion von Schwachstellen während des gleichzeitigen Editierens.
Bei der Codedurchsicht erfolgte die Verwendung einer Checkliste, um gerade in den Abschnitten **Feature Need-Based** und **Gleichzeitiges Editieren** entdeckte Fehler zu dokumentieren. Die Basis dieser Checkliste entstammte aus [TTC12] und wurde auf die Bedürfnisse in dieser Arbeit hin reduziert. Um das Verständnis zu erleichtern, wird in den folgenden Abschnitten, in denen die Codedurchsicht angewendet wurde, die Funktionsweise der einzelnen Funktionen und ihrer Zugriffe erläutert, wie es im Sinne des Walkthroughs ist. Wurden Defekte in den jeweiligen Abschnitten erkannt, erfolgt, wenn möglich, eine genaue Angabe darüber, welche Defekte vorlagen.

4.1 Feature Need-Based

Das Feature Need-Based stellt eine sinnvolle Erweiterung des Plugins dar. Ziel war es, bei Bedarf einzelne Dateien getrennt vom Einladungsprozess zu übertragen, um ein vorzeitiges Arbeiten auch während des Prozesses zu ermöglichen.
Gleichzeitig erweiterte es die Funktionalität von Saros dahingehend, dass beim Bearbeiten von noch nicht mit anderen Usern geteilten Dateien die Möglichkeit besteht, diese schnell über einen Dialog zur Session hinzuzufügen.

4.1.1 Funktionsweise der bisherigen Version

Die bisherige Version sah vor, `JupiterActivities`, die bei der Eingabe des Users in einem *EditorPart* kreiert werden, in der Klasse `SarosSession` abzufangen. Die dort getätigten Abfragen entschieden, ob das Feature Need-Based ausgeführt werden sollte.
Der damalige Entwickler vereinte die Fähigkeiten der Dateiübertragung während eines Einladungsprozesses mit der Möglichkeit, eine nicht zur Session gehörende Datei schnell zur Session hinzuzufügen.

4.1.2 Grund zur Analyse des Features

In einer der zahlreichen Tests zur Funktionsweise von Saros stellte sich bald ein Fehlverhalten dieses Features heraus. Das betreffende Szenario sah insgesamt drei User vor, wobei zwei User bereits in einer bestehenden Session waren und ein Weiterer während des Editierens zur Session eingeladen wird. Bei jeder Zeicheneingabe während des Editierens wurde der Host der Session dazu aufgefordert Need-Based zu aktivieren. Der Eingeladene erhielt die Eingaben nicht, weshalb ein inkonsistenter Zustand erzeugt wurde.

Das genaue Szenario:

- User: ALICE, BOB und CARL
- Projekt foo mit der Datei A.txt
- ALICE und BOB arbeiten in einer existierenden Session, welche das Projekt foo enthält, an der Datei A.txt
- Datei A.txt ist bei ALICE und BOB geöffnet
- ALICE lädt CARL zur Session ein, währenddessen arbeitet BOB an der Datei A.txt weiter
- CARL akzeptiert die Einladung und wählt das Zielprojekt aus
- ALICE erhält für jeden Tastenschlag von BOB die Nachfrage, ob Need-Based aktiviert werden sollte.

Dieser Zustand stellte ein Fehlverhalten dar, weshalb eine Fehlersuche zur Beseitigung dieses Szenarios unternommen wurde.

4.1.3 Anwendung der Codedurchsicht

Nach Entdecken dieses Szenarios bedurfte es einer kompletten Codedurchsicht des Features, da eine genaue Fehlersuche angewendet werden musste. Die Wahl fiel auf das Konzept der Codedurchsicht, da dieses Feature erst zu Beginn dieser Ausarbeitung in die Codebasis aufgenommen wurde, die vorgenommenen Änderungen demnach noch als Eintrag im Review-Board [KHeld12] hinterlegt waren. Auf Grund des vorzeitigen Einblicks in das Feature mit Hilfe des Review-Requests war die Verwendung des Konzepts Debugging für die Fehlersuche nicht geeignet. Das Feature, wie es im Review Request hinterlegt war, umfasste 44 Klassen, davon waren neun Klassen Aktivitäten, die um die Pfad-Ausgabe modifiziert wurden. Zwölf Klassen widmeten sich den entsprechenden JUnit- und *STF*-Tests. Eine Erweiterung der betroffenen Methoden um Ausgabeaufrufe war in Vergleich zu dem

Verwenden des Review-Requests und der schnellen Anwendung der Codedurchsicht nicht zu empfehlen.

Durch die weite Verteilung des Need-Based Features werden zuerst zentrale Methoden betrachtet, die den Grundgedanken des Features erkennen lassen. Im Weiteren folgen Methoden, die die Realisierung unterstützen, doch keinen entscheidenden Einfluss in die Logik von Need-Based selbst besitzen.

Zentrum des Need-Based Features

Nach dieser Grundentscheidung folgte die schrittweise Analyse des Features.

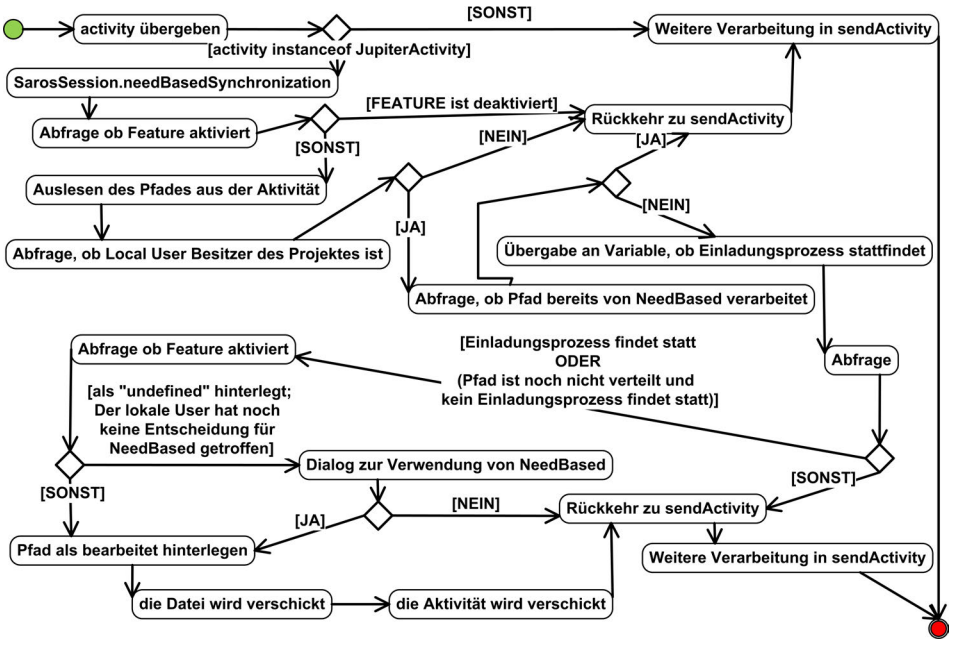

Abbildung 6: Aktivitätsdiagramm des Zentrums vom Need-Based-Feature

Anhand dieses vereinfachten Aktivitätsdiagramms [Abbildung 6] lässt sich erkennen, dass dieses Feature einen komplexeren Umfang hat, als eigentlich erwartet.
Folgende Methoden werden dabei betrachtet:

- `SarosSession.sendActivity`
- `SarosSession.needBasedSynchronization`
- `SarosSession.isInProjectNegotiation`

Begonnen wird mit der Methode `sendActivity` in der Klasse `SarosSession`. Diese Methode ist eine der wichtigsten Schnittstellen. Sie

wird nicht nur ausgeführt, wenn ein lokaler User Aktivitäten erzeugt und an andere Teilnehmer verschickt, sondern auch, wenn Aktivitäten von anderen Usern empfangen und ausgeführt werden, um die Verarbeitung zu bestätigen.
Bei der Analyse dieser Methode mussten folgende Fragen in der Checkliste während der Betrachtung negativ beantworten werden.

- **For each method: Is it no more than about 60 lines long?**
  Die Methode selbst ist ungefähr 60 Zeilen lang, ein Extraktion des Features Need-based wäre demnach angemessen.

- **For each compile module: Is no more than about 600 lines long?**
  Die Klasse selbst ist weiter über 600 Zeilen lang, da `SarosSession` das Zentrum von Saros darstellt. Das Feature selbst ist stark in dieser Klasse verwurzelt, könnte aber eigenständig existieren und die Lesbarkeit erhöhen.

- **Are any modules excessively complex and should be restructured or split into multiple routines?**
  Das Feature Need-Based selbst ist komplex, wie an Abbildung 6 zu erkennen ist, eine Auslagerung wäre daher angebracht.

In Zusammenhang mit dem Need-Based-Feature wird in der Methode überprüft, ob die als Parameter übergebenen Aktivitäten einer Instanz der Klasse `JupiterActivity` entsprechen.
Ist dies der Fall, wird an die zentrale Methode des Need-Based-Features `needBasedSynchronization` in der Klasse `SarosSession` eine in eine `JupiterActivity` konvertierte Aktivität übergeben, sowie der Liste der User, die sie empfangen sollen.

Durch die übergebene `JupiterActivity` wird klar, dass das Need-Based-Feature nur ausgeführt werden kann, wenn Texteingaben stattfinden, denn diese entsprechen der erforderlichen Klasse.

```
1           if (preferenceUtils.isNeedsBasedSyncEnabled().equals("
                false"))
2               return;
```

Listing 1: SarosSession.needBasedSynchronization

Wenn in den *Saros-Preferences* hinterlegt ist, dass das Need-Based-Feature nicht genutzt werden soll, so wird diese Funktion verlassen. Initialisiert ist diese `Preference` als “undefined“ in `PreferenceConstants`.

```
package de.fu_berlin.inf.dpp.preferences;

import de.fu_berlin.inf.dpp.StartupSaros;

/**
 * Constant definitions for plug-in preferences
 *
 * @author rdjemili
 * @author coezbek
 */
public class PreferenceConstants {
  ...
    public static final String NEEDS_BASED_SYNC = "undefined";

  ...
}
```

Listing 2: PreferenceConstants

Demnach ist bei der ersten Eingabe des lokalen Users gesichert, dass die Methode `needBasedSynchronization` nicht durch diese Abfrage verlassen wird. Es folgt ein Vergleich, ob der lokale User nicht Eigentümer des Projektes ist, gleichfalls wird überprüft, ob die betroffene Datei, in der die Texteingaben stattfinden, bereits von diesem Feature erfolgreich bearbeitet wurde. In beiden Fällen wird die weitere Verarbeitung in dieser Methode abgebrochen und zur Methode `sendActivity` zurückgekehrt.

```
        if (!isOwnedProject(iProject))
            return;

        if (needBasedPathsList.contains(path))
            return;
```

Listing 3: SarosSession.needBasedSynchronization

Dadurch ist sichergestellt, dass nicht mehrfach versucht wird, eine bereits durch dieses Feature versandte Datei erneut zu verschicken, beziehungsweise erneut Abfragen zur Aktivierung des Features zu stellen, wenn diese bereits für die Datei beantwortet wurden.

```
        boolean isProjectTransmitted = isInProjectNegotiation(
            jupiterActivity);

        /*
         * need-based transmission when file is not shared or
             file is in
         * transmission process
         */
        if (isProjectTransmitted
```

```
8                || (!isShared(path.getFile()) && !
                     isProjectTransmitted)) {
9                if (preferenceUtils.isNeedsBasedSyncEnabled().
                     equals("undefined")) {
10                    if (!CollaborationUtils.
                          activateNeedBasedSynchronization(saros))
11                        return;
```

Listing 4: SarosSession.needBasedSynchronization

Im weiteren Verlauf stellt die Methode sicher, ob sich das Projekt der Aktivität in einem Einladungsprozess befindet. Das Ergebnis wird an die Variable `isProjectTransmitted` übergeben. In der existierenden Abfrage wird die Variable verwendet um zu testen, ob ein Einladungsprozess für dieses Projekt gerade stattfindet. Ist dies nicht der Fall, so darf die Datei noch nicht in der Session an andere User verteilt sein, um den Code auszuführen. Es lässt sich daran eines ganz deutlich erkennen: das Übertragen während einer Einladung ist gleichgesetzt mit dem Hinzufügen einer neuen Datei zur Session. Ein Umstand, der Fehler mit sich bringt. Das Versenden einer Datei während der Einladung sollte nicht dem Hinzufügen einer Datei zur Session gleichgesetzt werden. Es sind zwei vollkommen unterschiedliche Features, die unterschiedlich betrachtet werden müssen. In der eigenen Überarbeitung wurde daher ein Aufsplitten der beiden Features in `PreSharing` (Versand während eines Einladungsprozesses) und `QuickAddingAFile` (dem schnellen Hinzufügen einer Datei zu einer bestehenden Session) vorangetrieben.

Der nächste Schritt stellt den Aufruf von `activateNeedBasedSynchronization` in der Klasse `CollaborationUtils` dar. Diese Methode ruft einen Dialog auf, der dem Nutzer die Option gewährt, seine Entscheidung zu speichern. Erst wenn der lokale User diese Option verwendet, wird der Wert in der *Saros-Preference* dauerhaft gesetzt. Ansonsten erhält der User für jede Eingabe in die entsprechende Datei den Dialog.

Im Allgemeinen stellt dies kein Problem dar, wird jede folgende lokale Eingabe doch geblockt, bis die Entscheidung im Dialog getroffen wird. In dem gegebenen Szenario ist der User BOB derjenige, der die Datei editiert.
Durch die Verarbeitung der für den User benötigten Abfragen in der Methode `sendActivity` und der Aktivierung des Need-Based Features mit Hilfe von Objekten der Klasse `JupiterActivity`, reagiert der Host ALICE auf die von BOB getätigten und übermittelten Eingaben. Der Grund dafür findet sich in der Verwendung des Jupiter-Algorithmus, in der der Host einer Session Eingaben anderer User transformiert und an die restlichen User über die Methode `sendActivity` weiterleitet. Damit ist jede empfangene `JupiterActivity` von BOB ein erneuter Auslöser für die Anfrage des Features Need-Based an ALICE.

Entscheidet sich der User bei dem Dialog, Need-Based nicht zu aktivieren, wird die weitere Verarbeitung abgebrochen und der normale Programmablauf in der Methode `sendActivity` wird fortgeführt. Entscheidet er sich für eine Aktivierung, wird der Pfad der entsprechenden Datei in eine Liste überführt, um spätere Versuche des Features zu unterbinden die Datei erneut zu verarbeiten. Anschließend wird die entsprechende Datei an alle User der Session verschickt.

```
1                try {
2                    sendSingleFile(path);
3                    sendActivity(toWhom, jupiterActivity);
4                } catch (FileNotFoundException e) {
5                    log.error("File could not be found, despite
                         existing: " + path,
6                         e);
7                }
```

Listing 5: SarosSession.needBasedSynchronization (Versand der Datei)

Es bleibt weiterhin die Frage offen, inwiefern festgestellt wird, ob ein Projekt Teil des Einladungsprozesses ist. Deshalb erfolgt zwangsläufig eine Analyse der Methode `isInProjectNegotiation`.
Diese Methode verwendet Aktivitäten aus dem Interface `IActivity`, dabei unterscheidet sie weder zwischen Objekte der Klasse `EditorActivity` oder der Klasse `JupiterActivity`, da beide diesem Interface entsprechen.

```
1            if (activity == null)
2                throw new IllegalArgumentException();
3
4            if (!(activity instanceof IResourceActivity))
5                return false;
6
7            SPath path = ((IResourceActivity) activity).getPath();
8            if (path == null)
9                return false;
```

Listing 6: SarosSession.isInProjectNegotiation

Die Methode `isInProjectNegotiation` hinterfragt zu Anfang, ob die übergebene `activity` ungleich `null` ist. Ferner erfolgt die Abfrage, ob die entsprechende Aktivität durch Tätigkeiten basierend auf eine `IResource`, wie zum Beispiel einer Datei oder einem Ordner, erschaffen wurde. Nur in diesen Fällen kann die Aktivität einer Klasse angehören, die das Interface `IResourceActivity` implementiert. Aktivitäten wie zum Beispiel `StopActivity` haben keine direkte `IResource` zum Ziel, können daher also nicht das Interface implementiert haben, demnach gibt die Methode den Wert `false` zurück. Gleichzeitig ist von Bedeutung, ob der extrahierte Pfad

ungleich `null` ist. Dies kann bei Objekten der Klasse `EditorActivity` der Fall sein. Sollte ein *EditorPart* von einer nicht in der Session befindlichen Datei geöffnet werden, so wird eine `EditorActivity` erschaffen, die den Pfad `null` aufweist. Sollte der Pfad nicht verfügbar sein, gibt auch hier die Methode den Wert `false` zurück.

```
1          // determine if we are in a transmission process with
               our project
2          Collection<ProjectNegotiation> projectNegotiations =
               projectNegotiationObservable
3              .getProcesses().values();
4          Set<String> projectIDs = null;
5          for (ProjectNegotiation projectNegotiation :
               projectNegotiations) {
6              projectIDs = projectNegotiation.getProjectNames().
                   keySet();
7          }
8          if (projectIDs != null) {
9              return projectIDs.contains(getProjectID(path.
                   getProject()));
10         }
11         return false;
```

Listing 7: SarosSession.isInProjectNegotiation

Letztlich soll in diesen Zeilen herausgefunden werden, ob das Projekt, welches zu dem vorher extrahierten Pfad der Aktivität gehört, in den momentanen Einladungsprozessen enthalten ist. Dazu werden von `projectNegotiationObservable` alle Einladungen ausgelesen und in einer `FOR`-Schleife durchgegangen. Mit jedem Durchlauf wird das `Set`, welches die `projectIDs` enthalten soll, sofern es welche gibt, mit dem `Set` der aktuellen Einladung überschrieben. Es wird davon ausgegangen, dass nur eine Einladung stattfinden kann. Dies entspricht im Allgemeinen dem bisherigen Verhalten der Software, dennoch ist dieser Programmierstil nicht zu empfehlen. Eine Initialisierung des `Sets` `projectIDs` sollte stattfinden, sowie eine komplette Übergabe aller Projekte einer jeden Einladung aus `projectNegotiationObservable`. Dies ist zu empfehlen, da es späteren Entwicklern das Verständnis zu diesem Code erleichtert, andernfalls wird unnötig Zeit in Anspruch genommen.
Am Ende erfolgt die Abfrage, ob das `Set` Werte enthält und ob das Projekt der Aktivität in diesem `Set` vorhanden ist.

- **Is every variable properly initialized?**
  Wie bereits erwähnt, wird eine Variable in dieser Methode nicht initialisiert. Das erschwert vor allem die Lesbarkeit für den Entwickler.

Zusammenfassung des Zentrums von Need-Based

Einer der Hauptpunkte, die es zu überarbeiten gilt, ist die momentane Integration des Features in die Klasse `SarosSession`. Dabei sollte das Feature eigenständig vorhanden sein, um etwaige Überarbeitungen für die Entwickler zu erleichtern.
Von genauso großem Interesse ist die Verwendung von Aktivitäten der Klasse `JupiterActivity`, da Texteingaben nicht als Auslöser des Features gelten, sondern das Öffnen und Fokussieren einer Datei ausreichen sollten.
Das Feature Need-Based muss zudem in das mögliche Versenden während eines Einladungsprozesses und dem schnellen Hinzufügen einer Datei zur Session zerteilt werden, wodurch die spätere Modifikation erleichtert wird.

Es wird ein besonderer Fokus auf die Umstände gelegt, dass das Need-Based-Feature nur von dem Host genutzt werden kann, was die Notwendigkeit und Nutzbarkeit dieses Features in Frage stellt. Wenn es nicht für andere User zu nutzen ist, warum sollte es dann in Saros integriert sein?
Ein Grund für diese eigenwillige Behandlung könnte ein dokumentierter Bug sein, der beim überlappenden Verteilen von Projekten auftritt. In diesem Fall hat User ALICE eine Session mit BOB eröffnet, ein Projekt ist dabei nur partiell verteilt. BOB fügt daraufhin eine Datei der Session hinzu und auf ALICE's Seite wurde vergessen, dass sie die anderen Dateien mit BOB teilte. Dieser Fehler entstammte der Problematik, dass ALICE nicht mehr als Besitzer eingetragen war, nachdem BOB die Datei hinzufügte. Später wurde dieser Fehler korrigiert.
Ein anderer Grund könnte gewesen sein, dass der Entwickler sich hauptsächlich auf das Versenden von Dateien während eines Einladungsprozesses konzentriert hatte. In diesem Szenario kann nur der User, der die Einladung verschickt, Besitzer sein.

Ebenfalls ist die Modifikation von `isInProjectNegotiation` wichtig, allein um die Lesbarkeit zu erhöhen, auch wenn sie keinen Fehler im eigentlichen Sinne darstellt.

Weitere Abschnitte des Features

In diesem Zusammenhang sind folgende Methoden von Interesse:

- `CollaborationUtils.activateNeedBasedSynchronization`
- `Utils.popUpRememberDecisionDialog`
- `SarosSession.sendSingleFile`
- `FileUtils.syncSingleFile`

- `SharedResourcesManager.exec`
- `CollaborationUtils.needBasedFileHandlingDialog`
- `FileUtils.backupFile`

Die Methode `activateNeedBasedSynchronization` aus der Klasse `CollaborationUtils` ruft lediglich die Methode `popUpRemember-DecisionDialog` der Klasse `Utils` auf und übergibt dabei den entsprechenden Eintrag in der *Saros-Preference* um die Wahl des Users für spätere Zugriffe zu sichern. Dies erfolgt, wenn der User sich entscheidet seine getroffene Wahl für spätere Situationen als Standardwert zu hinterlegen.
Bei `SarosSession.sendSingleFile` ist zu erkennen, dass die Position der Methode in der falschen Klasse stattfand. Die Klasse `SarosSession` sollte keinen Zusammenhang mit dem Versenden von Dateien aufweisen, da dies irreführend ist.

```
1          for (final User recipient : getRemoteUsers()) {
2              final ProgressMonitorDialog dialog = new
                   ProgressMonitorDialog(
3                  EditorAPI.getAWorkbenchWindow().getShell());
4              final SarosSession session = this;
```

Listing 8: SarosSession.sendSingleFile (Versand der Datei)

Für jeden User in der befindlichen Session wird ein neuer `Progress-MonitorDialog` erstellt. Gleichzeitig erfolgt eine Kopie des Objektes, dass die Methode aufruft. Sollte während dieser Übertragung die Session geschlossen werden, kann demnach nicht davon ausgegangen werden, dass alle User den gleichen Datenbestand haben. Angenommen ALICE, BOB und CARL sind in einer Session aktiv. ALICE will die Methode `sendSingleFile` verwenden, der Versand für BOB wurde gerade gestartet und es erfolgt die Vorbereitung für den Versand an CARL. ALICE bricht die Session ab. Das Resultat wäre, dass BOB zwar die Datei bekommen hat, CARL jedoch nicht. Die Möglichkeit zu diesem Szenario wird noch dadurch erhöht, dass dieses Feature ohne Einwirkung des Users arbeitet, sofern der User seine Entscheidung hat speichern lassen.

```
1              Utils.runSafeSWTSync(log, new Runnable() {
2                  public void run() {
3                      try {
4                          dialog.run(true, true, new
                               IRunnableWithProgress() {
5                              public void run(IProgressMonitor
                                   monitor) {
6                                  // synchronize the file and
                                       check if it is
7                                  // correctly transmitted
8                                  FileUtils.syncSingleFile(
                                       recipient, session,
```

```
                                        path, SubMonitor.convert(monitor));

                                    /*
                                     * Notify the session participants of your
                                     * activated editor. This is mandatory to avoid
                                     * inconsistencies with the editor manager and
                                     * by that with the follow mode.
                                     */
                                    editorManager.sendPartActivated();
                                }
                            });
```

Listing 9: SarosSession.sendSingleFile (Versand der Datei)

Um die Übertragung zu bewerkstelligen, muss diese Methode in einem *SWT*-synchronen Thread auf `syncSingleFile` der Klasse `FileUtils` zugreifen, gefolgt von dem Versand, dass der entsprechende *EditorPart* der Datei, sprich die Ansicht, aktiviert wurde. Da es sich um eine größere `FileActivity` handelt, ist die Verwendung des synchronen Threads vorzuziehen.

```
            // Reset jupiter
            if (sarosSession.isHost()) {
                sarosSession.getConcurrentDocumentServer().reset(from.getJID(),
                    path);
            } else {
                sarosSession.getConcurrentDocumentClient().reset(path);
            }
```

Listing 10: FileUtils.syncSingleFile

Es folgt die Abfrage, ob der entsprechende User der Host der Session ist, oder nicht. Ist der User der Host, so wird die Kombination aus dem Pfad der zu versendenden Datei und der *JID* des Senders im `ConcurrentDocumentServer` zurückgesetzt. Andernfalls erfolgt nur ein Zurücksetzen im `ConcurrentDocumentClient`.
Damit wurde der Versand von Dateien im Sinne des Features betrachtet, dennoch muss der Umstand überprüft werden, wie Saros mit der Ankunft einer einzelnen Datei umzugehen gedenkt.

In diesem Zusammenhang tritt die Methode `exec` der Klasse `SharedResourcesManager` in Erscheinung. Dabei interessiert nur jene Methode, die als Parameter eine `FileActivity` erhält, denn nur diese ist für das Feature

von Bedeutung. Die Methode mit dem Parameter der Klasse `IActivity` leitet und konvertiert in Abhängigkeit von der implementierten Klasse des Interfaces `IActivity`, wie zum Beispiel der Klasse `FileActivity`, an die entsprechende Methode mit passendem Parameter weiter.

```
1                boolean needBased = activity.isNeedBased();
```

Listing 11: SharedResourcesManager.exec

Sofern die Aktivität den Zusatz trägt, dass die Datei vom Typ `CREATED` ist, wird aus der `FileActivity` herausgelesen, ob diese im Zuge des Features übertragen wurde. Sollte bejaht worden sein, wird mit der Berechnung der Checksumme der empfangenen Datei begonnen und, sofern der Zielort dieser mit einer Anderen belegt ist, auch die Checksumme der Zielort-Datei. Andernfalls bleibt die Checksumme als `null` initialisiert.

```
1                    Long remoteChecksum = activity.getChecksum();
2                    Long localChecksum = null;
3                    if (file.exists()) {
4                        try {
5                            localChecksum = FileUtils.checksum(file
                                 );
6                        } catch (IOException e1) {
7                            log.debug("Checksum␣could␣not␣be␣
                                 generated.", e1);
8                        }
9                    }
```

Listing 12: SharedResourcesManager.exec

Existiert eine Datei an dem gewünschten Zielort und ist deren Checksumme unterschiedlich zu der Checksumme der empfangenen Datei, so wird ein Dialog für den Empfänger mit Hilfe der Methode `needBasedFileHandling-Dialog` der Klasse `CollaborationUtils` und eines übergebenen boolean-Parameters mit dem Wert `TRUE` aufgerufen. Gibt dieser Dialog den Wert `TRUE` zurück, wird die existierende Datei am Zielort abgespeichert und durch die Methode `backupFile` der Klasse `FileUtils` gesichert.

```
1                        if (CollaborationUtils.
                             needBasedFileHandlingDialog(activity
2                            .getSource().getHumanReadableName(),
                                 file.getName(),
3                            true)) {
4                            // TRUE: backup/rename and than
                                 overwrite
5                            try {
6                                editorManager.saveLazy(path);
7                                FileUtils.backupFile(file, monitor)
                                     ;
8                            } catch (FileNotFoundException e) {
9                                log.error(
```

```
                                    "File could not be found, despite existing: "
                                        + path, e);
                                }
                            }
```

Listing 13: SharedResourcesManager.exec

Sollte allerdings keine Datei am Zielort, oder die Checksummen zwischen der zu empfangenden Datei und der Datei am Zielort identisch sein, so wird die Methode `needBasedFileHandlingDialog` der Klasse `CollaborationUtils` mit einem boolean-Parameter des Wertes `FALSE` aufgerufen.

```
                try {
                    FileUtils.writeFile(
                        new ByteArrayInputStream(activity.getContents()), file,
                        monitor);

                    if (needBased) {
                        if (wasOpenedEditor)
                            editorManager.openEditor(path);

                        sarosSession.getConcurrentDocumentClient().reset(path);
                    }
                } catch (Exception e) {
                    log.error("Could not write file: " + file);
                }
```

Listing 14: SharedResourcesManager.exec

Nach dieser Verarbeitung erfolgt das Schreiben des Inhalts der `FileActivity` an den Zielort. Sofern der Grund für die Aktivität dem Feature geschuldet ist, werden die hinterlegten Daten eines Pfades im `ConcurrentDocumentClient` zurückgesetzt. Damit werden alle Transformationen, die auf der bisherigen Datei des Zielortes stattfanden, aus dem Gedächtnis des Jupiter-Algorithmus entfernt, wodurch verhindert wird, dass der Watchdog durch unterschiedliche Checksummen und Transformationen einen Recovery-Prozess aktivieren will. Es stellt sich dabei die Frage, warum beim Versenden der Datei in der Methode `syncSingleFile` der Klasse `FileUtils` ein Unterschied beim Zurücksetzen einer Datei im Vergleich zu der Methode `exec` in `SharedResourcesManager` stattfindet. Beim Versenden erfolgte eine unterschiedliche Verarbeitung des Zurücksetzens im `ConcurrentDocumentClient` in Abhängigkeit, ob der lokale User der Host der Session ist, oder nicht. Der Empfang hingegen unternahm keine unterschiedliche Betrachtung.

Gleichzeitig bleibt die Frage offen, was die Methode `needBasedFileHandlingDialog` zu realisieren versucht. Es ist bekannt, dass sie den Namen

des Senders, den Dateinamen und einen Boolean-Wert erhält. Bedingt durch den Namen kann ebenfalls von einem Dialog ausgegangen werden. Dagegen spricht das scheinbar unterschiedliche Verhalten, welches je nachdem hervorgerufen wird, ob die Datei am Zielort existiert und eine unterschiedliche Checksumme mit der empfangenen Datei aufweist, oder nicht. Der Aufruf eines Dialogs wird nicht gebraucht, wenn am Zielort keine existierende Datei vorhanden ist.

```
1       public static boolean needBasedFileHandlingDialog(String
            userName,String fileName, boolean showDialog)
2     {
3           String message = MessageFormat.format(
4               Messages.
                    CollaborationUtils_confirm_need_based_file_text
                    , userName,
5               fileName);
6           String messageBalloon = MessageFormat.format(
7               Messages.
8         CollaborationUtils_confirm_need_based_file_balloon_text,
9               userName, fileName);
10          if (showDialog) {
11              return Utils.popUpCustomQuestion(
12                  Messages.
                        CollaborationUtils_confirm_need_based_file,
                         message,
13                  new String[] { "Create␣Backup", "Overwrite" },
                        false);
14          } else {
15              SarosView.showNotification("New␣file␣in␣session!",
                    messageBalloon);
16              return true;
17          }
18      }
```

Listing 15: CollaborationUtils.needBasedFileHandlingDialog

In dieser Methode wird erkannt, dass der übergebene Boolean-Wert `showDialog` dazu verwendet wird, zu entscheiden, ob ein Dialog angezeigt werden soll, oder nicht. Ist ein Dialog nicht erwünscht, so ist `showDialog` auf `FALSE` gesetzt. Dafür erfolgt dann eine Anzeige in Form eines `MessageBalloons`. Damit ist die Funktion irreführend, da, rein nach dem Namen der Methode zu urteilen, von einem Dialog ausgegangen wird.
Letztlich erfolgt die Überprüfung der Methode `backupFile` der Klasse `FileUtils`, aufgerufen in der Methode `exec` von `SharedResourcesManager`. Die Methode wird ausgeführt, wenn der User durch den Dialog von `needBasedFileHandlingDialog` beschlossen hat, die am Zielpfad befindliche Datei abzusichern.

```
1       public static void backupFile(IFile file, SubMonitor
            monitor)
2           throws CoreException, FileNotFoundException {
3           if (!file.exists())
4               throw new FileNotFoundException();
5           IPath newPath = new Path(file.getName().concat("_BACKUP
                "));
6           file.move(newPath, true, monitor);
7       }
```

Listing 16: FileUtils.backupFile

Hier stellt sich insbesondere heraus, dass, sofern der Zielort, zu dem die Datei für das Backup verschoben werden soll, bereits existiert, eine `CoreException` geworfen, aber nicht abgefangen wird. In der Methode `exec` mit dem Parameter `FileActivity`, in der der Backup-Vorgang aufgerufen wird, wird eine `FileNotFoundException` abgefangen, sollte aber eine `CoreException` geworfen werden, so ist sie unbehandelt. Dadurch wird der Inhalt der `FileActivity` bei diesem Szenario nicht an den Zielort geschrieben, die Methode `exec` wird in der Weiterverarbeitung abgebrochen. Das System geht von einer korrekten Übertragung aus, weshalb der Watchdog eine Inkonsistenz feststellen wird, da die Checksumme der lokalen Datei und die Checksumme des anderen Users sich unterscheiden werden.

Zusammenfassung der weiteren Abschnitte

Es wurde festgestellt, dass die Methode `sendSingleFile` beim kurzfristigen Beenden der Session zu Inkonsistenzen führen kann. Dies ist zwar ein nicht gewünschtes Verhalten, könnte aber vernachlässigt werden. Es ist als unwahrscheinlich anzusehen, dass ein User, der eine Datei zur Session hinzufügt, diese zugleich beenden wird. Nur in diesem Szenario wäre das Fehlverhalten zu dokumentieren.
Weitaus entscheidender ist die irreführende Verarbeitung bei `needBasedFileHandlingDialog`. Der Name der Methode lässt auf einen Dialog schließen, was nicht immer der Fall ist, da diese Methode zugleich verwendet wird, um einen User lediglich über den Erhalt einer Datei mit Hilfe eines *Message-Balloons* zu informieren. Nur wenn eine Datei bei dem Empfang überschrieben werden muss, wird ein Dialog angezeigt.
Ein großes Problem stellt das Erstellen eines Backups einer Datei mit Hilfe der Methode `backupFile` dar. Sie fördert das inkonsistente Verhalten und muss überarbeitet werden.

Urteil über die Codedurchsicht bei Need-Based

Aus dem vorherigen Abschnitt wurden folgende Probleme extrahiert

- Bereich Usability
  - Das Feature sollte nicht erst bei einer Texteingabe ausgeführt werden, viel mehr sollte das Öffnen einer Datei ausreichen.
- Bereich Layout und Package
  - Das Feature muss in ein eigenes Package ausgelagert werden, um spätere Modifikationen erheblich zu erleichtern.
  - Die Methode `isInProjectNegotiation` muss von nicht initialisierten Variablen befreit sein.
  - Es muss eine Trennung zwischen dem Versenden während einer Einladung und dem schnellen Hinzufügen einer Datei zur Session erfolgen.
- Anpassung des Codes
  - `sendSingleFile` sollte angepasst werden, um unnötige Kopien zu reduzieren.
  - `needBasedFileHandlingDialog` sollte umbenannt und modifiziert werden.
  - Die Methode `backupFile` bedarf einer Überarbeitung, um ein korrektes Backup einer Datei zu gewährleisten.

Es wurden zahlreiche Problematiken aufgedeckt. Angefangen mit der falschen Positionierung der Methoden, über falsche Bezeichnungen, bis hin zu defekten Codeabschnitten.
Mittels des bekannten Review-Request konnten die Codeabschnitte schneller lokalisiert werden. Die wichtigsten Fehler wurden mit Hilfe der Durchsicht erfasst.

4.1.4 Überarbeiten des Features Need-Based

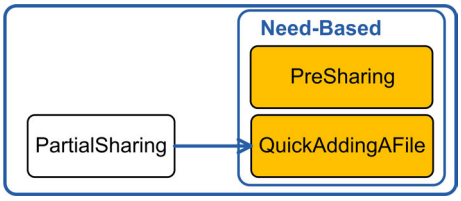

Abbildung 7: Konzept des neuen Need-Based-Feature[MSpier12]

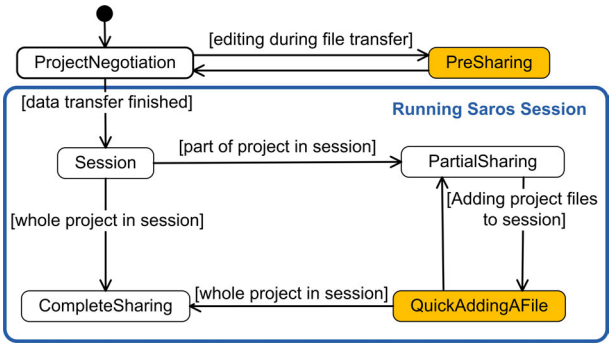

Abbildung 8: Simples Aktivitätsdiagramm des neuen Need-Based-Feature [MSpier12]

Nach Entdecken dieses Szenarios und ersten Einblicken in das Wirken des Features wurde der Entschluss gefasst, dieses auf mehrere Klassen und Methoden verteilte Feature in ein eigenes Package zu überführen und als `de.fu_berlin.inf.dpp.net.needBased` zu bezeichnen. Das Feature bleibt als Need-Based weiterhin bekannt, obwohl eine komplette Aufsplittung in die Features `PreSharing` und `QuickAddingAFile` stattfand, wobei `QuickAddingAFile` nur verwendet werden kann, wenn Partial-Sharing eines Projektes vorliegt [Abbildung 7]. Der Grund für den Erhalt des Namens ist historisch bedingt. Da die Fähigkeiten als Need-Based-Feature im Saros-Team bekannt sind, könnte eine Umbenennung bei der Wiederauffindung entscheidende Probleme mit sich bringen, zum Beispiel wenn ein neuer Entwickler von dem Feature Need-Based hört und annimmt, dass es wegen seiner Defekte nur gelöscht und nicht umgeschrieben wurde. Er könnte versucht sein, das Feature neu aufleben zu lassen und zu spät realisieren, dass eine korrekte Version in der Codebasis bereits vorhanden ist.

Es entstand folgende Aufteilung [Abbildung 9] in dem Package

- `NeedBasedSharingManager` enthält die Methoden, die entscheiden, ob es sich um den Vorgang des `PreSharing`'s oder ob es sich um `QuickAddingAFile` handelt.
- `PreSharing` führt die benötigten Abfragen aus, um sicherzustellen, dass es tatsächlich der Versand während der Einladung durchgeführt werden soll. Gleichzeitig wird der Versand einer Datei dokumentiert, um nicht erneut durch `PreSharing` verarbeitet zu werden. Letztlich startet der Versand der Datei.
- `QuickAddingAFile` überprüft zusätzliche Abfragen, inwiefern der User es wünscht, die Datei zu versenden. Ebenso wie `PreSharing` startet auch hier der Versand.

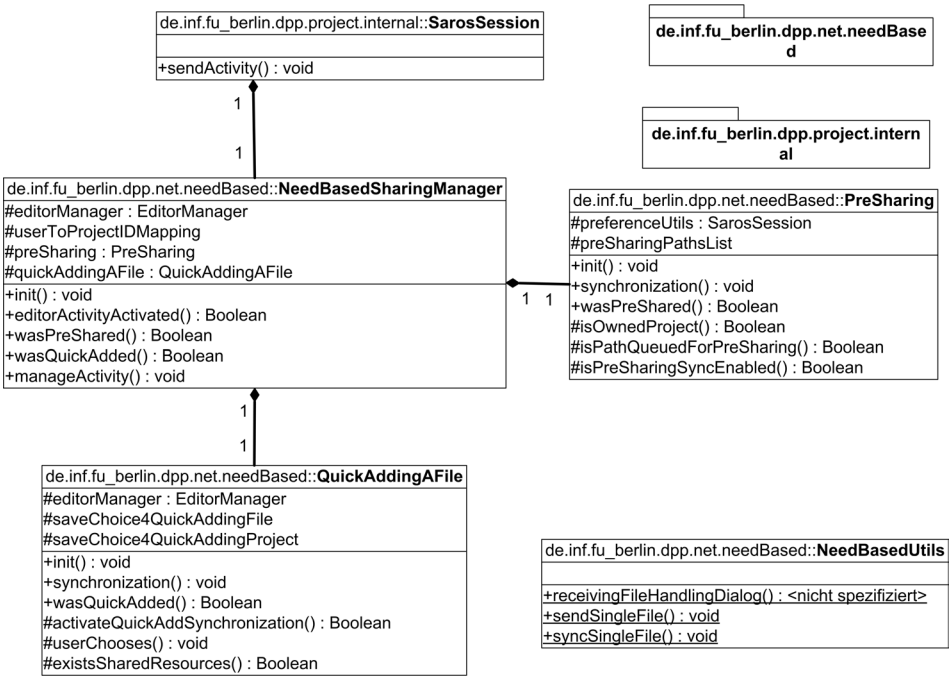

Abbildung 9: Klassendiagramm des neuen Need-Based-Feature

- `NeedBasedUtils` enthält alle Methoden, die Dialog oder Übertragungen bezüglich des Features durchführen.

Die Extraktion erfolgte mittels den im Reviewboard hinterlegten Request für den Patch des Features NeedBased. Da sich spätere Modifikationen durch andere Patches in den betroffenen Bereichen vom NeedBased-Patch in Grenzen hielten, konnte dieses Request zur Lokalisation des Codes verwendet werden. Der daraus resultierende Code wurde im ersten Schritt in eine Klasse `NeedBasedSharingManager` überführt.

NeedBasedSharingManager

Dazu wurde zuerst eine Klasse `NeedBasedSharingManager` erschaffen, die wiederum enthielt eine Methode `manageActivity`. Ihre Aufgabe sollte es sein, die Durchführung des Features zu steuern und an andere, gegebene Klassen zu verteilen [Abbildung 10].
In Abstimmung mit dem Saros-Team wurde der Aufruf von diesem Feature nicht mehr durch `JupiterActivities` realisiert, sondern dadurch, dass der entsprechende *EditorPart* zu der Datei von dem User fokussiert wird. Dazu musste das Kreieren von `EditorActivities` angepasst werden. Wurde bisher eine `EditorActivity` nur mit dem passenden Pfad

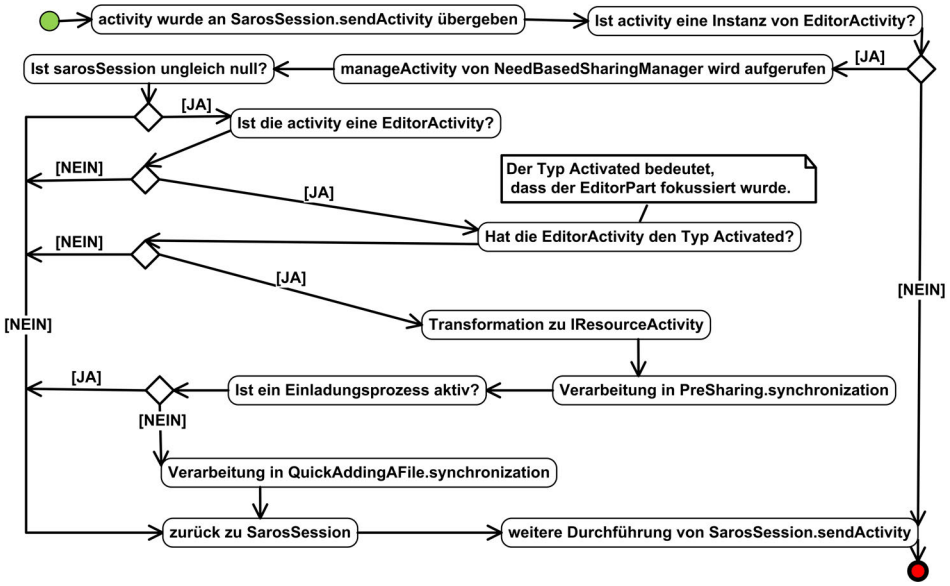

Abbildung 10: Aktivitätsdiagramm von NeedBasedSharingManager

erschaffen, wenn die entsprechende Datei bereits in der Session verteilt ist, sonst nur mit dem Wert `null` initialisiert, so wird nach dem Vorgang der Überarbeitung der Pfad prinzipiell übergeben, ganz gleich, ob die entsprechende Datei verteilt ist, oder nicht.

```
1       public void partActivated(IEditorPart editorPart) {
2
3       ...
4           if (!isSharedEditor(editorPart)
5               || !sarosSession.isShared(this.editorAPI
6                   .getEditorResource(editorPart))
7               || !editorAPI.getEditorResource(editorPart).
                    isAccessible()) {
8               SPath ePath = this.editorAPI.getEditorPath(
                    editorPart);
9               generateEditorActivated(ePath);
10          ...
11          return;
12          }
13      ...
14    }
```

Listing 17: EditorManager.partActivated

Durch die Umstellung des Codes wurde es ermöglicht, `EditorActivities` mit korrektem Pfad wahrzunehmen, auch wenn der betreffende *EditorPart* eine Datei beinhaltet, die noch nicht in der Session verteilt ist. Im Folgenden wurde die Codebasis von dem Feature Need-Based so umgeschrieben, dass sie nur bei Erhalt einer `EditorActivity` reagierte.

```
1       public boolean editorActivityActivated(IActivity activity)
            {
2           if (activity != null)
3               if (activity instanceof EditorActivity)
4                   if (((EditorActivity) activity).getType().
                        equals(
5                       EditorActivity.Type.Activated)) {
6                       return true;
7                   }
8           return false;
9       }
```

Listing 18: NeedBasedSharingManager.editorActivityActivated

Dazu empfahl sich die Entwicklung der Methode `editorActivity-Activated(...)` in der Klasse `NeedBasedSharingManager`, welche einen genauen Abgleich der übergebenen Aktivität durchführte. Sofern es sich um eine `EditorActivity` mit dem Zustand `Activated` handelte, gab die Funktion `TRUE` zurück.

```
1       public void manageActivity(SarosSession sarosSession,
            IActivity activity,
2           List<User> toWhom, boolean isInProjectNegotiation) {
3           if (sarosSession == null)
4               return;
5
6           if (!activity.getSource().equals(sarosSession.
                getLocalUser()))
7               return;
8
9           if (!editorActivityActivated(activity))
10              return;
11
12          preSharing.synchronization(sarosSession, editorManager,
13              isInProjectNegotiation, activity, toWhom);
14
15          if (!isInProjectNegotiation) {
16              quickAddingAFile.synchronization(sarosSession,
                    activity, toWhom);
17          }
```

Listing 19: NeedBasedSharingManager.manageActivity

Die betreffende Abfrage wurde in die Methode `manageActivity` integriert, ebenso wie die Nachfrage, ob der User in einer bestehenden `SarosSession` und ob er Quelle der Aktivität ist. Es folgt die Unterteilung in `PreSharing` und `QuickAddingAFile`. Dabei wird die aktuelle Session, der aktuelle `EditorManager` und der momentane Zustand, ob die Aktivität während eines Einladungsprozesses durchgeführt wird, sowie die entsprechende Aktivität und ihre Ziel-User an die jeweiligen Methoden von `QuickAddingAFile`

und `PreSharing` übergeben. Der Zustand, ob ein Einladungsprozess stattfindet, ist in der Variable `isInProjectNegotiation` hinterlegt. Die Methode `manageActivity` selbst wird in der `SarosSession` in `sendActivity` aufgerufen, sofern es sich bei der Aktivität um eine `EditorActivity` handelt.

PreSharing

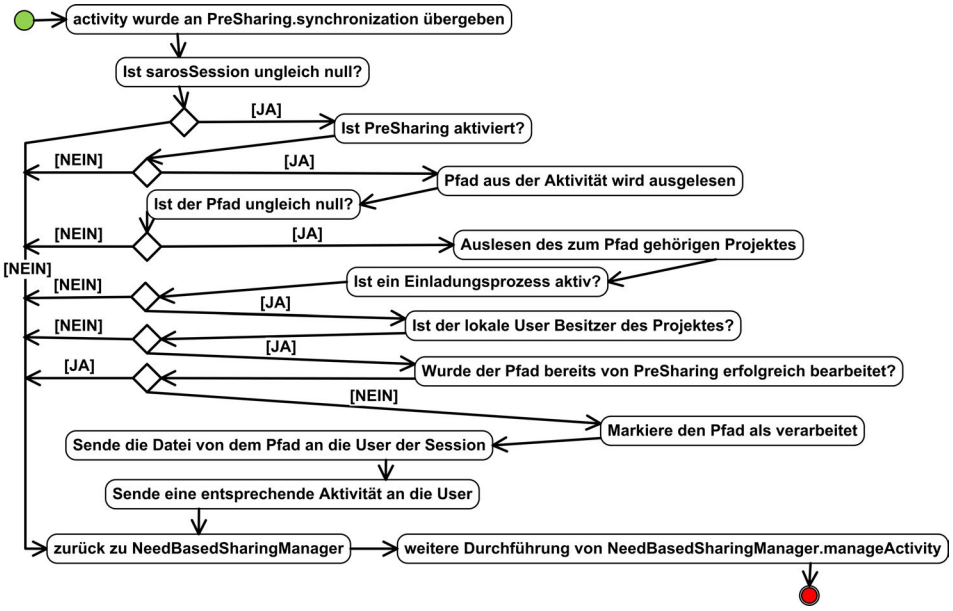

Abbildung 11: Aktivitätsdiagramm von PreSharing

Dieses Teilfeature wurde komplett neugestaltet [Abbildung 11]. Der User selbst wird während dem Öffnen einer Datei nicht gefragt, ob er `PreSharing` einsetzen will. Da es sich um ein experimentelles Feature handelt, ist es standardmäßig in den Preferences von Saros deaktiviert, kann jedoch vom User selbst aktiviert werden. Nur wenn diese Aktivierung stattfindet, kann eine Datei während der Einladung verschickt werden. Dialoge, die das `PreSharing` zu einem späteren Zeitpunkt aktivieren lassen, sind nicht vorgesehen.

Das `PreSharing` widmet sich den benötigten Abfragen, ob diese Funktionalität des Features angewendet werden kann. Die Hauptmethode ist dabei `synchronization`, welche abfragt, ob dieser Teil des Features in den *Preferences* aktiviert ist. Dort ist diese Funktionalität für den User erläutert und zum Aktivieren hinterlegt worden, auch mit genauer Anmerkung, dass es sich hier um ein experimentelles Feature handelt.

```
1           if (sarosSession == null)
2               return;
3
4           if (!isPreSharingSyncEnabled())
5               return;
6           IResourceActivity resActivity = (IResourceActivity)
                activity;
7           final SPath path = resActivity.getPath();
8           if (path == null) {
9               return;
10          }
11          IProject iProject = path.getProject();
12
13          if (!isInProjectNegotiation)
14              return;
15
16          if (!isOwnedProject(sarosSession, iProject))
17              return;
18
19          if (isPathQueuedForPreSharing(path))
20              return;
```

Listing 20: PreSharing.synchronization

In diesem Abschnitt wird durch die Methode `isPreSharingSyncEnabled` herausgefunden, ob der User in den aktuellen *Preferences*, erreichbar durch die Menübar über den Eintrag Saros, dieses Feature aktiviert hat. Wenn nicht, wird das `PreSharing` abgebrochen. Im Folgenden wird der Pfad aus der Aktivität ausgelesen und, sofern einer existiert, auch das Projekt herausgesucht. Sollte ein Einladungsprozess stattfinden, wird fortgefahren. Es wird geschaut, ob der lokale User Besitzer des Projektes ist, andernfalls wird auch hier wieder der Vorgang abgebrochen. Danach wird analysiert, ob diese Datei bereits von dem Feature bearbeitet wurde.

```
1           preSharingPathsList.add(path);
2
3           try {
4               NeedBasedUtils.sendSingleFile(sarosSession,
                    editorManager, path);
5               sarosSession.sendActivity(toWhom, resActivity);
6           } catch (FileNotFoundException e) {
7               log.error("File could not be found, despite
                    existing: " + path, e);
8           }
```

Listing 21: PreSharing.synchronization

Ist dies nicht der Fall, so wird abgespeichert, dass die momentane Datei nun bearbeitet wird. Letztlich wird die Datei mit Hilfe der Klasse `NeedBasedUtils` verschickt.

QuickAddingAFile

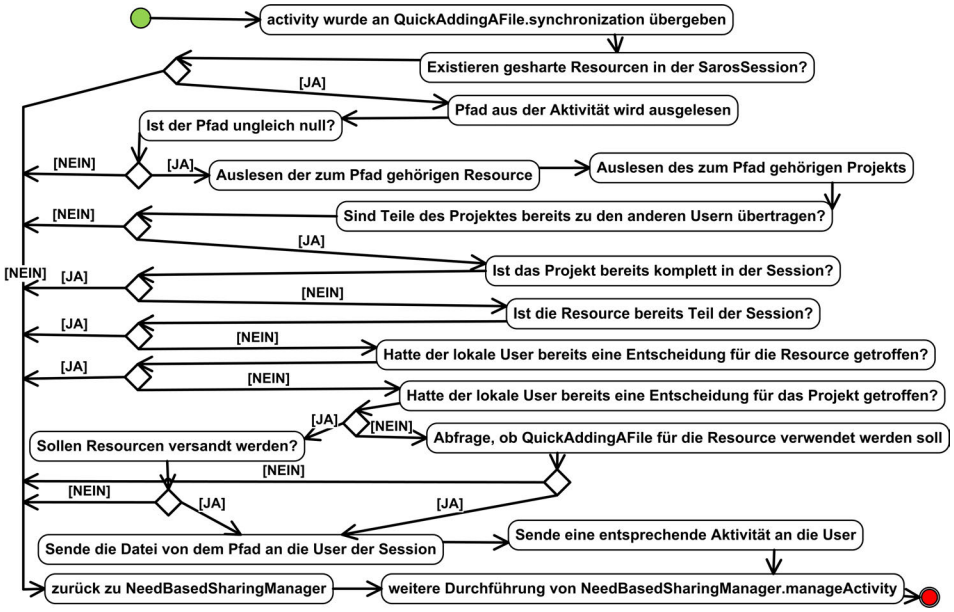

Abbildung 12: Aktivitätsdiagramm von QuickAddingAFile

Das Teilfeature `QuickAddingAFile` [Abbildung 12] wurde komplett überarbeitet. War es zuvor noch mit dem Versenden der Datei während einer Einladung gekoppelt, so ist es in der Handhabung komplett losgelöst von `PreSharing`. Ebenfalls ein wichtiger Aspekt ist, dass nur Dateien schnell hinzugefügt werden können, wenn sie zu dem Teil nicht verteilter Dateien in einem partiell verteilten Projekt gehören.
Das Öffnen einer Datei ruft einen Dialog auf, in dem der User entscheiden kann, ob die Datei einer Session hinzugefügt werden soll. Ferner kann der User über eine Checkbox in dem Dialog entscheiden, ob seine Wahl für alle noch nicht verteilten Dateien in diesem Projekt gelten soll, oder nicht.

```
1          if (!existsSharedResources(sarosSession))
2              return;
3
4          IResourceActivity resActivity = (IResourceActivity)
               activity;
5          SPath path = resActivity.getPath();
6          if (path == null) {
7              return;
8          }
9
10         IResource resource = path.getResource();
11         IProject project = path.getProject();
12
13         if (!sarosSession.isProjectTransmitted(project))
```

```
14                return;
15
16            if (sarosSession.isCompletelyShared(project)) {
17                return;
18            }
19
20            if (sarosSession.isShared(resource))
21                return;
```

Listing 22: QuickAddingAFile.synchronization

In diesem Codeabschnitt wird hinterfragt, ob das verteilte Projekt überhaupt schon übertragen ist. Ist dies der Fall, ist von Interesse, ob es nur partiell verteilt und ob die zu versendende Datei nicht als verteilt eingetragen ist. Sollte dies alles zutreffen, so kann das Teilfeature verwendet werden. Um allerdings herausfinden zu können, ob ein Projekt übertragen wurde, wird die Klasse `SarosProjectMapper` um eine Liste erweitert. Diese speichert bereits erhaltene Projekte ab, wenn sie übertragen worden sind. Deshalb wurde auch die Klassen `OutgoingProjectNegotiation` und `IncomingProjectNegotiation` modifiziert, indem sie eine eigene Liste der momentan zu erhaltenen oder zu versendenden Projekte beinhalten, die, wenn der Prozess abgeschlossen wird, seine Werte an die neue Liste in `SarosProjectMapper` übergeben.

```
1            if (saveChoice4QuickAddingFile.containsKey(resource)) {
2                if ((saveChoice4QuickAddingFile.get(resource) !=
                     DialogChoice.UNKNOWN))
3                    return;
4            }
```

Listing 23: QuickAddingAFile.synchronization

Es muss sichergestellt sein, dass das Teilfeature diese Datei nicht schon verarbeitet hat.

```
1            if (saveChoice4QuickAddingProject.containsKey(resource.
                 getProject())) {
2                if ((saveChoice4QuickAddingProject.get(resource.
                     getProject()) != DialogChoice.UNKNOWN)) {
3                    if (saveChoice4QuickAddingProject.get(resource.
                         getProject()) == DialogChoice.YES) {
4                        try {
5                            NeedBasedUtils.sendSingleFile(
                                 sarosSession,
6                                editorManager, path);
7                            sarosSession.sendActivity(toWhom,
                                 resActivity);
8                        } catch (FileNotFoundException e) {
9                            log.error("File couldn't send", e);
10                       }
11                   }
12                   return;
```

```
13                  }
14              }
```

Listing 24: QuickAddingAFile.synchronization

Gleichzeitig bleibt von Bedeutung, ob für das Projekt der Datei nicht schon eine Entscheidung gespeichert wurde. Ist dies der Fall und sollte das Hinzufügen zur Session gestattet worden sein, so wird das Versenden über die Klasse `NeedBasedUtils` realisiert. Sollte die gespeicherte Entscheidung das Versenden ablehnen, so wird die weitere Ausführung abgebrochen.

```
1           if (activateQuickAddSynchronization(resource)) {
2               try {
3                   NeedBasedUtils
4                       .sendSingleFile(sarosSession, editorManager
                            , path);
5                   sarosSession.sendActivity(toWhom, resActivity);
6                   userChooses(resource, DialogChoice.YES);
7               } catch (FileNotFoundException e) {
8                   log.error("File␣couldn't␣send", e);
9               }
10          } else {
11              userChooses(resource, DialogChoice.NO);
12          }
```

Listing 25: QuickAddingAFile.synchronization

Existiert keine gespeicherte Entscheidung, so wird der User per Dialog befragt, wie er zu entscheiden gedenkt. Bei positiver Wahl, wird die Datei durch die Klasse `NeedBasedUtils` übertragen.

NeedBasedUtils

Diese Klasse beinhaltet mehrere Methoden, um die für das Feature notwendigen Dialoge und den speziellen Versand einer einzelnen Datei zu regulieren. Die Methode `sendSingleFile` erhält als Parameter eine finale SarosSession übergeben. Eine erneute Kopie bei der Bearbeitung eines jeden Users in der Session wurde entfernt.
Die Methode `receivingFileHandlingDialog`, ehemals `needBasedFileHandlingDialog`, erhielt eine komplette Neustrukturierung. Der User hat bei der Anzeige des Dialogs die Möglichkeit mit Hilfe einer Checkbox zu entscheiden, ob er von der bereits vorhandenen Datei am Zielort ein Backup erstellen will, oder nicht. Gleichzeitig wird ihm ermöglicht, den Empfang abzubrechen und damit die Session zu verlassen.

Veränderte Methoden außerhalb des Packages

Damit wurde der Kern des Features merklich stabilisiert, wobei die Methoden, wie zum Beispiel `backupFile` der Klasse `FileUtils`, zusätzlich

angepasst werden mussten. Dies zeichnet sich durch die stabile, erfolgreiche Ausführung der entsprechenden *STF*-Tests, sowie der Beseitigung der auftretenden Dialoge in dem beschriebenen Szenario, aus.
Die Methode wurde dahingehen spezifiziert, dass sie testet, ob der Zielort des Backups schon mit einer anderen Datei belegt ist. Stellt sich dies heraus, wird, wie es auch bei Windows der Fall ist, hinter dem betreffenden Namen der zu sichernden Datei ein in Klammern stehender Index dargestellt. Sollte auch dieser Zielort belegt sein, so wird der Index erhöht.
Die Methode `isInProjectNegotiation` wurde ebenfalls den Anmerkungen entsprechend angepasst. Das Gleiche geschah mit `exec` in der Klasse `SharedResourcesManager`.

4.2 Watchdog

Der Watchdog stellt eine der zentralen Features in dem Plugin Saros dar. Für den Fall, dass Inkonsistenzen bei einem User auftreten, hervorgerufen durch falsche Bearbeitung der entsprechenden Datei, muss der Watchdog in der Lage sein, dem User dies zu vermitteln und ihm die Option gewähren, die Fehler zu beheben, so dass eine weitere Zusammenarbeit ermöglicht wird.

4.2.1 Momentaner Aufbau des Watchdogs

Der Watchdog, der den Recovery-Prozess durchführt, unterteilt sich in fünf Klassen, die `IsInconsistentObservable`, die `ConsistencyAction`, der `ConsistencyWatchdogClient`, der `ConsistencyWatchdogServer` und der `ConsistencyWatchdogHandler`.

`IsInconsistentObservable` Bei festgestellten Inkonsistenzen wird ein `Boolean`-Wert gesetzt, um die `ConsistencyAction` für den betreffenden User aktivieren zu können.

`ConsistencyAction` Die Klasse `ConsistencyAction` startet den Recovery-Prozess und wird durch einen Druck des *Action*-Buttons in der *Saros-Roster-View* betätigt. Dieser Button ist erst für Nicht-Host-User verfügbar, wenn eine Inkonsistenz festgestellt wurde. Der Host selbst erhält diese Möglichkeit nicht, da seine Datenbasis beim Recovery-Prozess als Datenquelle dient und eine Verwendung des Prozesses auf Seiten des Hosts demnach nicht erwartet wird.

`ConsistencyWatchdogClient` empfängt in Intervallen die Checksummen aller Dateien. Diese werden mit den lokal existierenden Checksummen verglichen. Wird ein Unterschied festgestellt, so wird der Button für die `ConsistencyAction` aktiviert.

Nach Betätigung blockiert der Client alle Eingaben des betreffenden Users. Es folgt eine Anfrage zum Erhalt der aktuellen Dateien vom Host. Die erhaltenen Dateien dienen als Basis um die inkonsistenten Daten zu überschreiben und den Recovery-Prozess zu beenden.

`ConsistencyWatchdogServer` versendet alle 10 Sekunden die aktuellen Checksummen der Dateien, damit die anderen User mögliche Inkonsistenzen erkennen können.

`ConsistencyWatchdogHandler` überträgt die aktuellen Versionen der inkonsistenten Dateien von dem Host an den User, der inkonsistent ist.

4.2.2 Erläuterung des Problems

Der Grund für die zwangsläufige Bearbeitung wird durch den fehlschlagenden Saros-Test `RecoveryWhileTypingTest` deutlich.

Dieser Test verwendet die User ALICE und BOB, wobei ALICE der Host ist. ALICE lädt BOB zu einer Session ein und setzt BOB auf READ-ONLY, danach fügt ALICE permanent in eine Datei Zeichen ein. BOB hat in Eclipse nicht mehr die Möglichkeit, die Datei zu editieren und modifiziert diese außerhalb von Eclipse. ALICE erfährt nichts von diesen Änderungen und es tritt eine Inkonsistenz bei BOB auf. BOB startet den Watchdog, um die Inkonsistenz aufzulösen. Währenddessen versucht ALICE weiterhin diese Datei zu editieren.

ALICE wurde als Host davon in Kenntnis gesetzt und hat die Aufgabe, ihren aktuellen Stand der betroffenen inkonsistenten Dateien an den User zu verschicken.

Der Host setzte alle anderen User und sich selbst auf READ-ONLY und versendete die betreffende Datei an BOB. Gleich nach dem Versenden löst ALICE die Sperre und fing erneut an, die Datei zu editieren.

BOB konnte die empfangene Datei nicht verarbeiten, der alte Zustand der inkonsistenten Datei blieb bestehen.

4.2.3 Anwendung des Debuggings

Bei diesem Fehlverhalten wurde das Konzept des Debuggings verwendet. Der Test `RecoveryWhileTypingTest` war bereits existent und die während dieses Vorgangs entstehenden Log-Dateien boten genug Material, um Ansätze zu finden.

Es wurde daher davon ausgegangen, dass die Verwendung des Debuggings ausreichend wäre, um das Problem zu bewerkstelligen, weshalb auf eine genaue Codedurchsicht verzichtet wurde. Dadurch ist es möglich, einen ersten Eindruck in die Nützlichkeit des Konzeptes zu erhalten.

4.2.4 Erste Analyse der Log-Dateien

Die beim ersten Durchlauf des Tests `RecoveryWhileTypingTest` entstandenen Log-Dateien wurden analysiert. Es stellte sich heraus, dass beim Loggen der Unterschiede zweier Dateien eine `ArrayOutOfBoundException` geworfen wurde.

```
1               // Calculate diff of the two files
2               Diff diff = new Diff(oldContent, newContent);
3               Diff.Change script = diff.diff_2(false);
4
5               // log diff
6               DiffPrint.UnifiedPrint print = new DiffPrint.
                    UnifiedPrint(
7                   oldContent, newContent);
8               Writer writer = new StringWriter();
9               print.setOutput(writer);
10              print.print_script(script);
11
12              String diffAsString = writer.toString();
13              if (diffAsString.trim().length() == 0) {
14                  log.error("No␣inconsistency␣found␣in␣file␣[" +
                        from.getName()
15                      + "]␣" + path);
16              } else {
17                  log.info("Diff␣of␣inconsistency:␣\nPath:␣" +
                        path + "\n"
18                      + diffAsString);
19              }
```

Listing 26: Utils.logDiff

Da dieser Vergleich in späteren Abschnitten nicht mehr verwendet wird, die Berechnung also keinen nennenswerten Nutzen aufweisen konnte, wurde dieser Abschnitt mit einem simplen `try-catch`-Block umgeben. Dadurch wurde die Fehlermeldung abgefangen und die vom Host erhaltene Datei verwendet.

Nach der beschriebenen Anpassung wurde der Test erneut ausgeführt. Daraufhin wurde ein Dialog-Fenster aufgerufen, indem dem inkonsistenten User mitgeteilt wird, dass die betreffende Datei modifiziert wurde. Er hat die Wahlmöglichkeit, die Änderungen zu übernehmen, oder zu verwerfen.

4.2.5 Auftreten des Dialogs „File Changed! “

Der Grund dafür war, dass in dem Augenblick, da die zu überschreibende Datei bei dem User ankommt, noch weitere Aktivitäten durchgeführt werden sollten. Die nahezu zeitgleiche Ausführung unterschiedlicher Aktivitäten auf derselben Datei hat zur Folge, dass Eclipse nicht in der Lage ist, den entsprechenden *EditorPart* in dieser kurzen Zeitspanne korrekt zu aktualisieren, weshalb der Dialog erscheint. Es wurde ein genauerer Blick in die betreffenden Klassen `ConsistencyWatchdogClient` und `ConsistencyWatchdogHandler` geworfen. Dabei wurde festgestellt, dass der inkonsistente User zuerst sich selbst blockiert, damit seine Dateien nicht mehr von ihm editiert werden können, dann sendet er an den Host, dass ein Recovery-Prozess gestartet werden soll. Der Host versendet daraufhin zu allen anderen eine `StopActivity`, blockiert sich selbst und sendet die betroffene Datei in Form einer `FileActivity`, hebt aber gleich danach die Blockade auf.

Nach diesem Aufheben editiert der Host wieder. Die Vermutung, die sich dabei auftat, war, dass die Übertragung und Verarbeitung der `FileActivity` länger braucht, als der Empfang und die Verarbeitung der folgenden Aktivitäten [13].

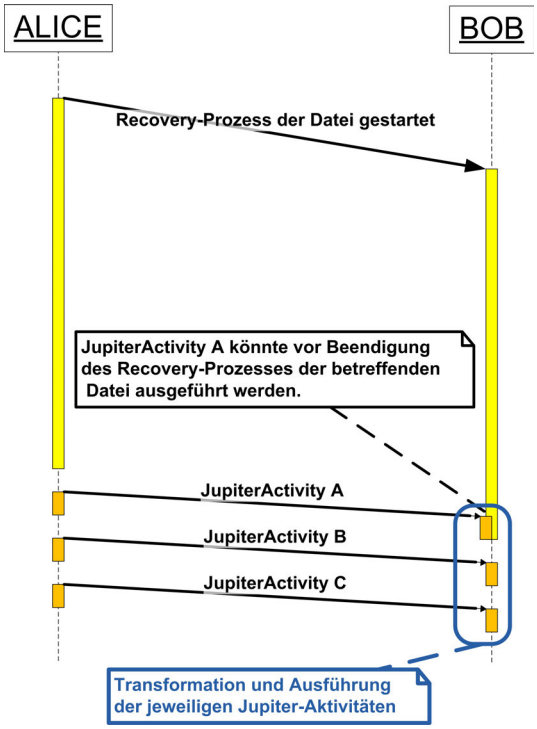

Abbildung 13: Annahme für den Grund des Problems beim Watchdog

Im Zuge dieser Annahme wurde versucht, das Blockieren aller User zu verlängern. Dadurch wurde der Code so abgewandelt, dass der inkonsistente User alle Anderen auf READ-ONLY setzt, während er die Datei empfängt. Dadurch wurde das Auftreten dieses Dialogs verhindert, doch wurde die Dauer des Recovery-Prozesses verlängert.

4.2.6 Buffer für alle eintreffenden Aktivitäten

Die erhöhte Dauer war nicht akzeptabel, daher wurde die Änderung verworfen. Stattdessen wurde ein Buffer bei der Ausführung im `EditorManager` hinzugefügt. Ist der lokale User in einem Recovery-Prozess, werden alle empfangenen Texteingaben mit einem Zeitstempel in den Buffer abgelegt. Am Ende des Recovery-Prozesses erfolgt die Ausführung der Elemente in diesem Buffer. Dabei wurde der Zeitpunkt abgespeichert, an dem der Recovery-Prozess begann, sowie zu jeder Aktivität auch der Zeitpunkt, an dem sie in den Buffer gelegt wurde. War der Recovery-Prozess abgeschlossen, wurden alle Aktivitäten ausgeführt, die einen Zeitstempel nach dem Beginn des Prozesses hatten. Eine Stabilitätssteigerung lies sich dabei feststellen, der Dialog verschwand, doch traten noch immer Inkonsistenzen auf. Es benötigte einer Erweiterung des Tests, um den Grund dafür zu analysieren. Deshalb wurde vor den Textvergleichen die Länge der Texte analysiert.

4.2.7 Stabilisierung des Buffers

Im weiteren Testverlauf stellte sich heraus, dass bei dem vorherigen inkonsistenten User zu wenige Texteingaben ausgeführt werden. Es wurde daher davon ausgegangen, dass die Erfassung der Zeitstempel bestimmter Aktivitäten vor der Erfassung des Zeitstempels für den Recovery-Prozess in den Buffer gelegt und dadurch bei der späteren Ausführung verworfen wurden.
Um dieses Problem zu beheben, wurde die Struktur der Aktivitäten genauer erfasst. Ein Objekt, dem Interface `IActivity` entsprechend, entstammt zuweilen aus einem Objekt mit dem Interface `IActivityDataObject`. Dieses Objekt wiederum wurde in der Klasse `TimedActivityDataObject` aufbewahrt. In dieser Klasse ist ferner noch ein eindeutiger Zeitstempel hinterlegt. Es wurden die für den Watchdog benötigten Aktivitätsobjekte angepasst, um diesen Zeitstempel von der Klasse `TimedActivityDataObject` in entsprechende Objekte des Interfaces `IActivityDataObject` und letztlich in Objekte von `IActivity` zu übergeben.
Dabei war eine Anpassung von drei besonderen Klassen, der `TextEditActivity`, der `TextSelectionActivity` und der `FileActivity`, vonnöten. `TextEditActivity` und `TextSelectionActivity` sind dabei, da sie für die Texteingaben verantwortlich, `FileActivity` ist nötig, da dies die Form ist, über die der Recovery-Prozess Dateien verschickt und die Konsis-

tenz wieder herstellt. Die zugehörigen Klassen mit dem Interface `IActivity-DataObject` wurden ebenfalls modifiziert.

Das Grundgerüst der neuen Verarbeitung war damit gelegt worden, doch benötigte es einer genaueren Analyse, inwiefern die Serialisierbarkeit gegeben war. Dabei stellt sich heraus, dass es vor allem Probleme geben kann, wenn der Recovery-Prozess beendet wird. Gilt der Prozess als beendet, können Aktivitäten ausgeführt werden, bevor der Buffer geleert ist. Dies könnte problematisch werden, wenn die ausgeführten Aktivitäten am Ende der Datei stattfinden. Dadurch könnte versucht werden, Eingaben außerhalb des verfügbaren Bereiches zu schreiben.

Um dieses Problem zu beheben, wurde eine Semaphore eingefügt, die nur einen Thread zulässt. Sie blockiert zwischen dem Deaktivieren des Recovery-Prozesses und dem Leeren des Buffers. Gleichzeitig versucht sie zu blockieren, wenn eine `TextEditActivity` oder eine `TextSelectionActivity` außerhalb des Recovery-Prozesses ausgeführt wird. Dadurch können jene Aktivitäten nur stattfinden, wenn kein Prozess stattfindet.
Durch diese Änderungen kann der Test `RecoveryWhileTypingTest` erfolgreich ausgeführt werden.

4.2.8 Falscher Eindruck durch Debugging

Da die Resultate während des Debuggings diesen Schluss nahelegten, und die Reparaturen zum gewünschten Erfolg führten, wurde davon ausgegangen, dass dies die Fehlerquelle sein müsste. Auch spätere Tests ließen darauf schließen, dass das Problem behoben ist. Dazu wurde ein STF-Test erstellt, in dem drei User interagieren.

Die User ALICE und BOB schreiben, CARL wird auf READ-ONLY gesetzt, editiert aber eine Datei außerhalb von Eclipse. Dadurch entsteht eine Inkonsistenz. CARL versucht diese mit Hilfe des Watchdogs zu lösen, während ALICE und BOB weiter die Datei editieren.

Doch eine Modifikation durch einen anderen Kollegen im Saros-Team zeigte, dass dieser Eindruck täuschen konnte. Der Recovery-Prozess hatte Probleme, eine offene Datei korrekt zu ersetzen, daher führt die Lösung des Kollegen ein vorheriges Schließen der Datei durch.
Die aus dem Debugging-Prozess entstandene Lösung in dieser Diplomarbeit kann als Basis für spätere Varianten verwendet werden, um die Verwendung einer Blockade während des Recovery-Prozesses weiter zu minimieren, bis diese nicht mehr gebraucht wird.

4.3 Gleichzeitiges Editieren

Die Hauptfunktionalität, die in diesem Plugin realisiert wurde, ist das gleichzeitige Editieren ein und derselben Datei eines in einer Session befindlichen Projektes durch die Eingaben mehrerer Nutzer. Um das Auftreten von Inkonsistenzen während dieser Eingaben zu verhindern, wird die Transformation im Sinne des Jupiter-Algorithmus verwendet [Ziller]. Der Versand dieser Transformationen erfolgt über *XMPP*-Nachrichten an die betreffenden User [Gustavs].

4.3.1 Dokumentation zum Gleichzeitigen Editieren

Eine erste Analyse stellte dar, inwiefern Aktivitäten kreiert und verarbeitet werden. Dabei wurde auf die Daten der offiziellen Saros-Webseite zugegriffen [SWeb12]. Es musste erkannt werden, dass die gegebenen Informationen [Abbildung 14] nicht dem aktuellen Stand entsprechen. Wie schon in den Abschnitt **Architektur von Saros** beschrieben, ist Saros in einem permanenten Wandel, dadurch kann auf die Aktualität der Dokumentationen nur selten vertraut werden.
Die Verarbeitung, wie sie zum Zeitpunkt dieser Arbeit war, umfasste zusätzliche Klassen, wie die `SarosSession` selbst, welche eine zentrale Rolle in dem Versenden oder Empfangen von Aktivitäten spielte.

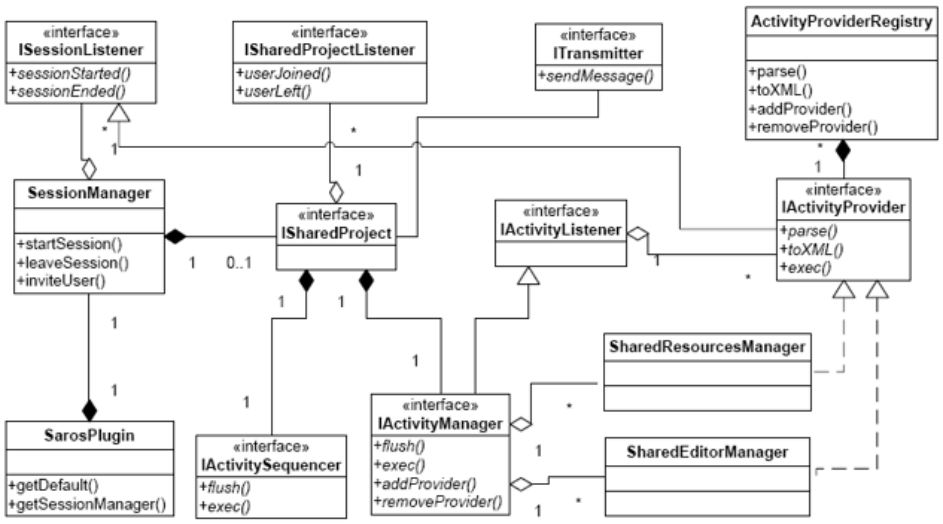

Abbildung 14: Verarbeitung der Aktivitäten (veraltete Version) [SWeb12]

Dasselbe gilt für die Abläufe während des Sendens [Abbildung 15] und Empfangens[Abbildung 16] von Aktivitäten. Auch hier hat sich der Ablauf entscheidend verändert, weshalb eine Verwendung nicht in Betracht kam.

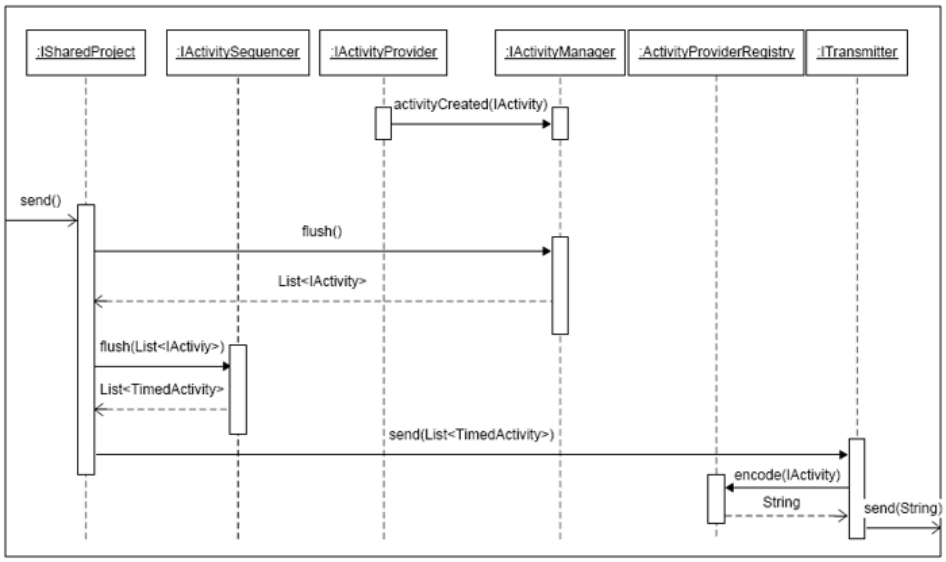

Abbildung 15: Versand von Aktivitäten (veraltete Version)[SWeb12]

Daher war es unumgänglich, den Verarbeitungsablauf neu zu analysieren und zu erfassen.

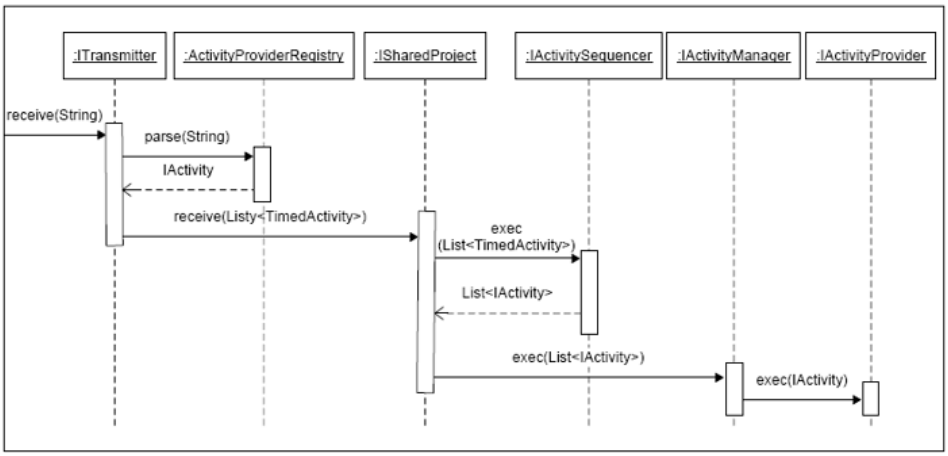

Abbildung 16: Empfang von Aktivitäten (veraltete Version)[SWeb12]

4.3.2 Verarbeitung von Texteingaben

Der Aufbau zur Realisierung des gleichzeitigen Editierens beinhaltet zahlreiche Klassen und lässt sich in vier Kategorien unterteilen, dem Erzeugen, Transformieren, Versenden und Empfangen von Aktivitäten.
Die Kategorien selbst sind klar voneinander zu trennen. Dies vereinfacht die

Fehlersuche, da durch Code-Modifikationen, beziehungsweise Erweiterung um Log-Informationen, in den abgegrenzten Bereichen und der anschließenden Durchführung automatisierter Tests Rückschlüsse auf die Beschaffenheit eben jener gezogen werden können. Bevor eine genaue Spezifizierung der durchgeführten Tests in Betracht kommt, muss eine Betrachtung stattfinden, welche Klassen in den jeweiligen Kategorien zu finden sind, weshalb ein kurzer Überblick von entscheidendem Interesse ist.

Das Erzeugen von Aktivitäten beginnt durch die Klassen `SharedResourcesManager` und `StoppableDocumentListener`, in denen die von Eclipse erzeugten Events abgefangen werden.
Die Klasse `SharedResourcesManager` dient der Verarbeitung aller Events, die nicht während des Editiervorgangs einer Datei entstehen, wie zum Beispiel den Events beim Verschieben oder Kreieren von Dateien. Es folgt die Verarbeitung dieser Events und Weitergabe der gewonnenen Informationen an die Klasse `AbstractActivityProvider`. Die dortige Verarbeitung mündet in der Kreation und dem Versand der passenden Aktivitäten in der Klasse `SarosSession`.
Der `StoppableDocumentListener` verarbeitet alle Events, deren Ursprung bei dem Editieren einer Datei zu finden sind, es folgt eine Weiterleitung an die Klasse `EditorManager` und mündet in den gleichen Funktionen der Klasse `SarosSession`.

Die Transformation dient dazu, allen Usern eine konsistente Datenbasis zu gewährleisten. Um dies zu gewährleisten, wurde der Jupiter-Algorithmus verwendet, in denen ein Client eigene Eingaben an den Server verschickt, der Diese entsprechend transformiert und an die anderen User weiterleitet. Beim Client empfangene Eingaben werden transformiert, um die Konsistenz zu bewahren. Im Falle von Saros enthält jeder User einen eigenen Client und der Host einer Session verwaltet den Jupiter-Server.

Dabei wird in der Klasse `SarosSession` der entsprechende `ConcurrentDocumentServer` beim Ausführen einer Aktivität aufgerufen, sofern der User der Host der Session ist. Diese Klasse lässt die Aktivitäten, sofern es sich um Jupiter-Aktivitäten handelt, von der Klasse `JupiterServer` weiterverarbeiten und übergibt deren Ergebnisse als Liste an die `SarosSession` zurück. Der `JupiterServer` extrahiert den gesuchten `JupiterDocumentServer` für die entsprechende Datei, auf die sich die Aktivität bezieht und übergibt das Ergebnis an den `ConcurrentDocumentServer`. Der `JupiterDocumentServer` hingegen besitzt von jedem User in der Session einen eigenen `Jupiter-Client`. In Abhängigkeit von den jeweiligen Clients lässt er die Aktivität transformieren und übergibt eine Map, in der für jeden User die entsprechend transformierte Aktivität zu finden ist.

Weitaus einfacher ist die Handhabung in Bezug auf die Clients eines jeden Users. Der `ConcurrentDocumentClient` lässt die Aktivität, sofern es sich um eine Jupiter-Aktivität handelt, durch die Klasse `JupiterClient` transformieren. Diese bedarf des Zugriffs auf die Klasse `Jupiter`, um die Transformation zu bewerkstelligen.
Dennoch unterscheidet der Client zwischen eingehenden und ausgehenden Aktivitäten. Ausgehende Aktivitäten werden nicht transformiert, sondern generiert und verschickt, der Server ist für die Transformation verantwortlich. Bei ausgehenden Aktivitäten ist noch von Interesse, ob der Host der Sender ist. Lediglich der Host ist in der Lage, Checksummen-Aktivitäten an die User in der Session zu versenden.

Der Versand beginnt bei der Klasse `SarosSession`, in der die jeweiligen Aktivitäten an den `ActivitySequencer` weitergegeben werden. Dort werden sie in einer Queue aufbewahrt und in regelmäßigen Abständen als Liste an die Klasse `XMPPTransmitter` übergeben. Es folgt eine Umwandlung der Aktivitäten in eine XML-Nachricht und die Wahl der Übertragungsform. Dabei wird unterschieden zwischen einer In-Band-Übertragung, wie *In-Band Bytestreams* (*IBB*), oder, sofern es möglich ist, eine Out-Band-Übertragung an die jeweiligen Nutzer über *Sock5-Bytestream*(*Sock5*). *IBB* ist für kleinere Datenpakete gedacht, da die entsprechenden Server mitunter eine Begrenzung der zu übertragenden Nachrichten aufweisen [Gustavs]. *Sock5* bietet sich durch die direkte Übertragung vor allem im Einladungsprozess zum Versenden der Projekte an, wird auch beim Versenden aller anderen Aktivitäten verwendet.

Der Empfang startet in einem Thread in `BinaryChannelConnection`, dessen einzige Aufgabe der Erhalt eines Bytestreams und die Weiterleitung an den `DataTransferManager` darstellt. Dort werden die erhaltenen Objekte verarbeitet und an den `XMPPReceiver` übergeben, der die Daten des Objektes in eine Paketnachricht verfasst und letztlich in ein `Smack`-Objekt konvertiert. Nachfolgend wird dieses Paket an den `ActivityHandler` versandt. Der Handler liest aus dem Paket die `TimedActivities` aus und lässt alle Aktivitäten einzeln vom `ActivitySequencer` verarbeiten. Dort werden sie einer Liste hinzugefügt und, sofern die Session mit anderen Nutzern bereits gestartet ist, ausgeführt. Andernfalls bleiben die gespeicherten Aktivitäten erhalten, bis sie in der Klasse `SarosSession` ausgeführt werden können. Am Ende werden die Aktivitäten zum Einen dem `EditorManager` übergeben, zum Anderen an die restlichen User verschickt.

4.3.3 Erstellen von entsprechenden Tests

Um entsprechende Testcases und Codemodifizierungen für die Fehlersuche erstellen zu können, ist eine zusätzliche Kontrolle über den entsprechenden *XMPP*-Server notwendig. Daher wurde ein *Openfire*-Server [Tabelle 1] eingerichtet. Der Server speichert alle Nachrichten, egal welchen Typs, die er während seines Betriebes empfängt.

	Spezifikation des Servers
Typ	Openfire-Server
Version	3.7.0
Betriebssystem	Windows 7 Professional (64-Bit)
Prozessor	Intel Core i7 Q720
Geschwindigkeit	1,60 GHz
Arbeitsspeicher	8 GB RAM
Netzwerkadapter	WLAN (Intel Centrino Advanced-N 6200 AGN)

Tabelle 1: Der erstellte Openfire-Server

Gleichzeitig wurde die Kontrolle über die Hardware auf Seiten des Clients [Tabelle 2] übernommen. Dadurch ist sichergestellt, dass mögliches Auftreten von inkonsistentem Verhalten nicht auf unterschiedliche Hardware zurückzuführen ist, sondern viel mehr auf unterschiedliche Betriebssysteme.

	Spezifikation des Clients
Betriebssystem 01	Windows XP Media Center Edition (32-Bit)
Betriebssystem 02	Ubuntu 10.04 LTS-Version (32-Bit)
Prozessor	Intel Core 2 Duo
Geschwindigkeit	1,83 GHz
Arbeitsspeicher	4 GB RAM
Netzwerkadapter	WLAN (Intel PRO/Wireless 3945ABG)

Tabelle 2: Der verwendete Client

Den eingetragenen Bugs im Saros-Bug-Tracker entsprechend, wurden vier Grundtests erstellt, die im weiteren Verlauf durch Dauer der Eingabe und Verhältnis der Texteingaben zwischen den Testusern variieren.
Die Grundtests beinhalten die drei Testbuddies ALICE, BOB und CARL. ALICE erstellt eine Session für das Projekt foo mit der Datei readme.txt. Die Anzahl der Eingaben und Löschaktionen ist fest begrenzt, wodurch es mit Hilfe der Eingaben in den Log-Dateien möglich ist, die genaue Anzahl an verschickten und angekommenen Pakete zu erfahren.

Basistests:

1. ALICE, BOB und CARL geben ein Zeichen in der Datei readme.txt ein, navigieren danach in die erste Zeile zum ersten Zeichen.
2. ALICE, BOB und CARL geben eine Zeichenkette in der Datei readme.txt ein, navigieren erst dann in die erste Zeile zum Anfangszeichen.
3. ALICE und BOB geben jeweils ein Zeichen in der Datei readme.txt ein, navigieren danach zum Anfangszeichen der ersten Zeile. CARL markiert einen Abschnitt und löscht ihn, danach navigiert er ebenfalls zum Anfangszeichen der ersten Zeile.
4. ALICE und BOB geben eine Zeichenkette in die Datei readme.txt ein, CARL markiert einen Abschnitt und löscht ihn. Danach navigieren die Buddies zur ersten Zeile und dem ersten Zeichen.

Der Vergleich der Eingaben erfolgte direkt nach dem letzten, eingegebenen Zeichen. Die Bugs, die es zu reproduzieren galt, basierten auf entstehende Inkonsistenzen während des Editierens in einer Zeile. Es war daher anzunehmen, dass die beschriebenen Szenarien einen Anhaltspunkt für die Fehlerquelle liefern würden.

Dabei wurden erst vereinfachte Tests kreiert, bei denen die Dauer der Session im Vordergrund stand. Wurde eine Inkonsistenz bei Einen der beiden Betriebssysteme des Clients festgestellt, so wurde damit begonnen, die Tests in Abhängigkeit von der Anzahl der Eingaben (ein eingegebener Buchstabe entspricht einer Eingabe) durchlaufen zu lassen. Dadurch war nicht mehr die Dauer der Session von Interesse, sondern vielmehr die Eingaben eines jeden Users. Die Grenze dabei wurde bewusst hoch angesetzt, bei 10.000 Eingaben, und dann schrittweise reduziert, bis eine minimale Anzahl an Eingaben gefunden wurde, um die Inkonsistenz heraufzubeschwören. Dies wurde bei 1.000 eingegebenen Zeichen pro User erreicht.

Die Grundtests wurden dahingehend angepasst und fortan zur Durchführung herangezogen, um entsprechende Ausgaben im Sinne des Debuggings in Log-Dateien auszugeben.

4.3.4 Vorgehen zur Fehlersuche

Der Code von Saros, der während des gleichzeitiges Editierens aufgerufen wird, wurde in die Abschnitte **Analyse des Jupiter-Algorithmus**, **Versand von Aktivitäten**, **Empfang von Aktivitäten**, **Verarbeitung der Aktivitäten** und **Versand von Aktivitäten**. Jeder Abschnitt erfuhr eine eigene Analyse mit Hilfe der Konzepte von Debugging und Codedurchsicht.

Sofern die Konzepte sich gegenseitig beeinflussten, wurde dies genauer erläutert.
Während der Verwendung des Debuggings wurden drei eclipse-Instanzen gleichzeitig ausgeführt. Im Folgenden wird von Log-Dateien im Allgemeinen gesprochen, wenn die Log-Daten aller ausgeführten eclipse-Instanzen keine nennenswerten Unterschiede zueinander bieten.

4.3.5 Analyse des Jupiter-Algorithmus

Dieser Abschnitt beinhaltet die Klassen:

- `ConcurrentDocumentServer`
- `JupiterServer`
- `ConcurrentDocumentClient`
- `JupiterClient`
- `Jupiter`

Debugging

Im Zuge der Analyse des Jupiter-Algorithmus wurden die betreffenden JUnit-Tests genauer betrachtet. Es fiel auf, dass die Tests nicht mehrere Eingaben betrachteten, sondern lediglich eine Eingabe pro Client und danach einen Vergleich der Texte der Clients begonnen wurde. Um diesen Umstand zu beheben, wurde Tests kreiert, die mehr als eine Eingabe zu verarbeiten hatten. Auch hier konnte die Zahl der Eingaben variiert werden, letztlich wurde die Anzahl auf 1.000 Eingaben beschränkt. Der Grund hierfür fand sich in der Erkenntnis, dass die modifizierten Grundtests, wie sie im vorherigen Abschnitt beschrieben wurden, bei 1.000 Eingaben pro User Inkonsistenzen erzeugten. Wenn der Fehler bei dem Jupiter-Algorithmus liegen würde, wäre die Inkonsistenz ebenfalls bei dieser Anzahl beim JUnit-Test zu erwarten gewesen, sofern es nur an einem Fehler in diesem Algorithmus liegt. Die Tests sind in der Klasse `DiplomConvergenceTest` des Packages `diplom` zu finden. Das Ergebnis war eindeutig, eine Inkonsistenz konnte nicht erzeugt werden, ganz gleich, welches Betriebssystem verwendet wurde.

Codedurchsicht

Um die Korrektheit des Algorithmus entscheiden zu können, wurde die entsprechende Diplomarbeit [Ziller] zu Rate gezogen, um den Grundgedanken zu erfassen. Im Anschluss folgte eine genaue Betrachtung des Codes. Dabei wurde mit der Client-Seite des Algorithmus begonnen. Es folgte eine genauere Betrachtung der Unterschiede zwischen ausgehenden und eingehenden Aktivitäten.

Es wird mit der Betrachtung der eingehenden Aktivitäten begonnen, anschließend folgt die gesonderte Betrachtung ausgehender Aktivitäten.

Eingehende Aktivitäten werden dabei zwischen Checksummen-Aktivitäten und JupiterAktivitäten unterschieden. Alle anderen auftretenden Aktivitäten werden prinzipiell unverändert der Ergebnis-Liste übergeben. Bei Objekten der Klasse `ChecksumActivity` erfolgt eine Betrachtung auf die Aktualität der empfangenen Aktivität. Die Aktualität ist gegeben, sofern die lokale Vector-Zeit mit der von der empfangenen Remote-Vector-Zeit übereinstimmt. Sollte diese Remote-Vector-Zeit der `ChecksumActivity` größer sein, müssten demnach bestimmte Aktivitäten bei dem User nicht angekommen sein, wodurch die Vector-Zeit kleiner wäre. Sollte einer dieser Umstände eintreten, wird eine `TransformationException` geworfen, die erst im `ConcurrentDocumentClient` abgefangen und in die entsprechende Log-Datei eingefügt wird.

Eine Beeinflussung des Debuggings wurde dadurch in die Wege geleitet, denn sollten Aktivitäten tatsächlich nicht ankommen, was die durchgeführten Tests vermuten lassen, so müssten die entsprechenden Log-Dateien jene Fehler aufweisen. Dies konnte jedoch nicht festgestellt werden. Es könnten drei Gründe für jenes Verhalten existieren:

- Die verschollenen Aktivitäten sind gar nicht während dem Empfang, dem Versand und den jeweiligen Transformationen verschwunden, sondern wurden vielleicht in späteren Schritten nicht korrekt ausgeführt.

- Die `ChecksumActivity` wird nur alle 10 Sekunden verschickt. Es könnte sein, dass die aktuelle Checksummen-Aktivität nicht transferiert wurde, weil die Zeitspanne noch nicht abgelaufen, die Session aber bereits beendet ist. Demnach könnten Aktivitäten in der letzten Zeitspanne zwischen zwei Checksummen-Aktivitäten verschwunden sein. Da die Tests eine unterschiedliche Dauer haben und mitunter von 5 bis 55 Minuten variierten, ist auszuschließen, dass dies der Grund sein kann. Andernfalls wäre Saros in der Lage, herauszufinden, wann die letzten 10 Sekunden einer Session eintreten und dies ist nicht möglich.

- Auch die `ChecksumActivity` kommen nicht korrekt an und verschwinden, genauso wie die Jupiter-Aktivitäten. Dies kann als unwahrscheinlich angesehen werden, da, wie der Abschnitt Watchdog mit dem `RecoveryWhileTypingTest` zeigt, Inkonsistenzen anhand von Objekten der Klasse `ChecksumActivity` erkannt werden.

Rückkehr zur Codedurchsicht. Die Verarbeitung von empfangenen Jupiter-Aktivitäten [Abbildung 17] endet ebenfalls in einer Abfrage, ob die Bedingungen erfüllt sind, unter Verwendung der Methode `checkPreconditions` der Klasse `Jupiter`, bevor die Aktivität transformiert wird. In dieser Methode wird erfasst, ob die empfangene Aktivität vor Veränderungen durch den lokalen User erschaffen wurde, was nur geschehen könnte, wenn der Server des Jupiter-Algorithmus seine Aufgabe zur Serialisierung nicht korrekt durchführte. Gefolgt von der Abfrage, ob die empfangene Aktivität eine Vector-Zeit hat, die größer als die zu erwartende Vector-Zeit ist, was nur eintritt, wenn Aktivitäten nicht ausgeführt wurden. Abschließend wird die Kontrolle gestartet, ob die empfangene lokale Vector-Zeit der Aktivität ungleich der aktuellen Remote-Vector-Zeit des lokalen Users ist. Sollte eine der Kontrollen eintreten, wird eine `TransformationException` mit den genaueren Angaben geworfen, die wiederum in dem `ConcurrencyDocumentClient` abgefangen und in eine Log-Datei überführt wird. Andernfalls tritt eine Transformation der Aktivität und eine Rückgabe des Ergebnisses über den `ConcurrencyDocumentClient` an die `SarosSession` ein.

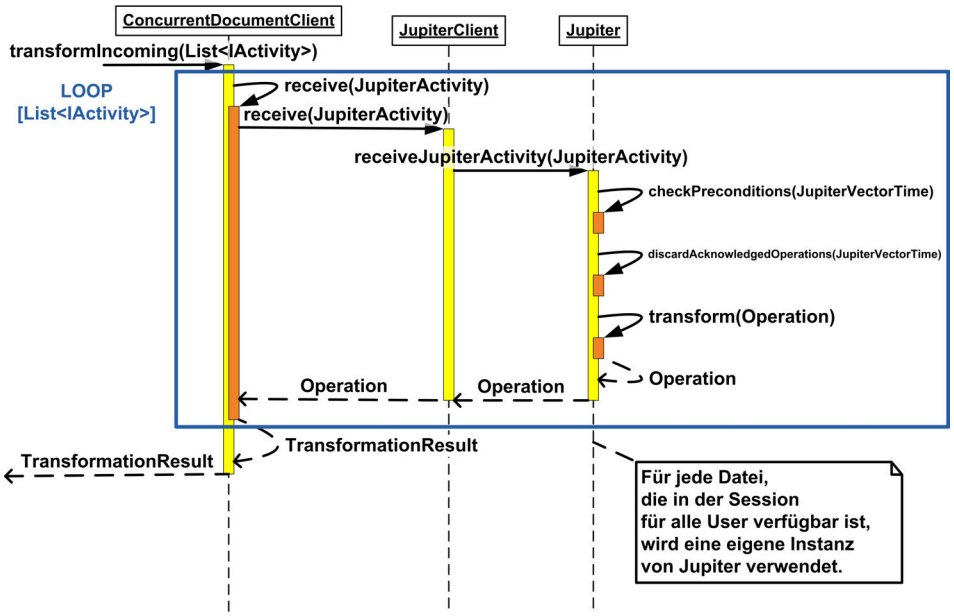

Abbildung 17: Sequenzdiagramm der Transformation von eingehenden Jupiteraktivitäten auf Seiten des Clients

Erneute Beeinflussung des Debuggings. Auch hier wurden Log-Daten nach erwarteten Nachrichten durchsucht, dennoch konnten erneut keine Meldungen gefunden werden. Damit bleiben die zwei möglichen Gründe bestehen. Entweder verschwinden Aktivitäten zu einem späteren Zeitpunkt der Verarbeitung, oder es verschwinden sämtliche Aktivitäten, inklusive der Checksummen-Aktivitäten. Dies wäre bei anderen Tests aufgefallen.

Ausgehende Aktivitäten [Abbildung 18] unterscheiden gleichfalls wie eingehende Aktivitäten zwischen Checksummen- und Jupiter-Aktivitäten. Beim Versenden einer Checksummen-Aktivität ist sichergestellt, dass nur der Host diese Versenden kann. Die Jupiter-Aktivitäten werden generiert, die Aktivitäten werden einer Liste hinzugefügt, um eingehende Aktivitäten korrekt transformieren zu können. Eine Transformation ausgehender Aktivitäten findet nicht statt.

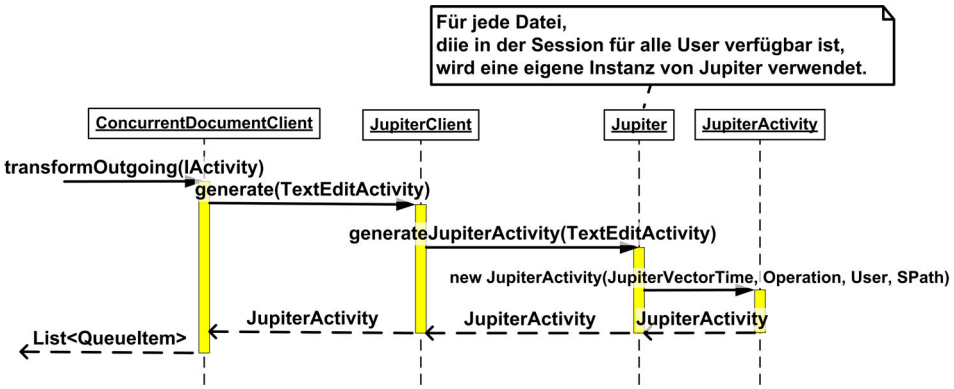

Abbildung 18: Sequenzdiagramm der Transformation von ausgehenden Jupiteraktivitäten auf Seiten des Clients

Die Server-Seite [Abbildung 19] des Jupiter-Algorithmus wird lediglich von dem Host der Session verwaltet. Dazu muss der `ConcurrentDocument-Server` die Aktivitäten transformieren. Der Server beinhaltet für jeden User einen Jupiter-Client. Es folgt eine Transformation für jeden einzelnen Client, wobei alle daraus resultierenden Ergebnisse an den `ConcurrentDocument-Server` zurückgegeben werden. Dieser übergibt das Ergebnis an die `Saros-Session`, um an die User der Session versandt zu werden.

Die Implementierung entspricht der Diplomarbeit [Ziller], ferner sind auch keine Fehler oder Unstimmigkeiten in dem Abschnitt zu entdecken. Auch der Debugging-Prozess ließ keine Fehler erkennen. Es kann daher davon ausgegangen werden, dass das Problem nicht am zentralen Jupiter-Algorithmus zu finden ist.

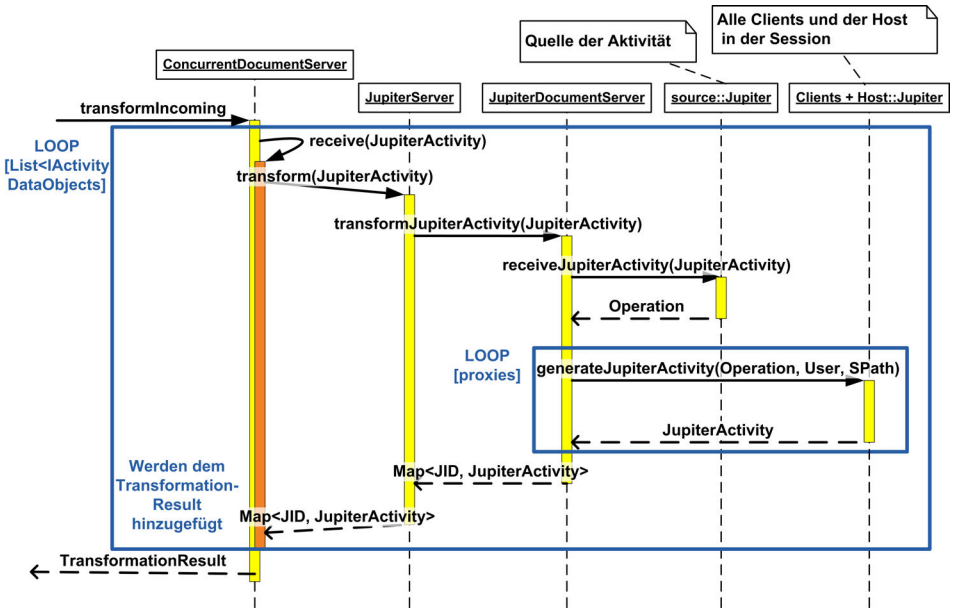

Abbildung 19: Sequenzdiagramm der Transformation von eingehenden Jupiteraktivitäten auf Seiten des Servers

4.3.6 Erstellen von Aktivitäten

Dieser Abschnitt beinhaltet die Klassen [Abbildung 20]:

- `StoppableDocumentListener`
- `EditorManager`
- `SarosSession`

Debugging

Die Ausgaben zeigen, dass die korrekte Anzahl an Eingaben vom `StoppableDocumentListener` erfasst und an den `EditorManager` weitergeleitet werden. Mit der gleichen Anzahl erfolgt einer Übergabe an die `SarosSession`. In den Log-Daten ist hinterlegt, dass zu jedem Zeitpunkt der Tests die Anzahl an Ausgaben konstant war, in Abhängigkeit von den Texteingaben. Dies stimmt auch mit dem normalen Verhalten von Saros überein. Die auftretenden Inkonsistenzen sollten bei mehr als zwei Usern auftreten, bei weniger jedoch nicht. Da das Erstellen von Aktivitäten bei beiden Varianten identisch ist, kann demnach nahe gelegt werden, dass der Fehler hier nicht zu finden ist. Es kann daher davon ausgegangen werden, dass dieser Bereich korrekt verarbeitet wird.

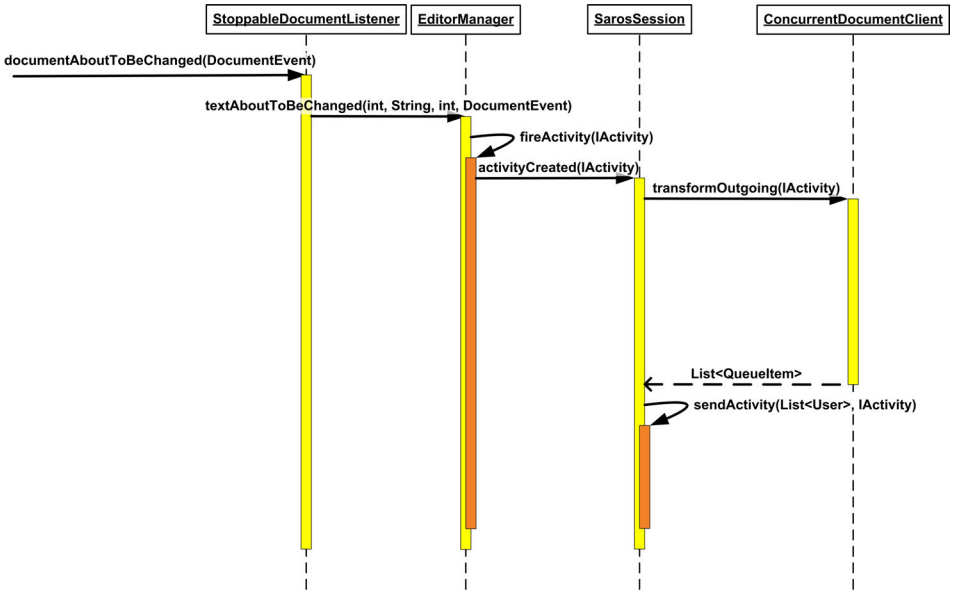

Abbildung 20: Sequenzdiagramm des Erstellens von Jupiteraktivitäten

User	TextEditActivity	JupiterActivity
ALICE(host)	1.000	1.000
BOB	1.000	1.000
CARL	1.000	1.000

Tabelle 3: Aufruf der Methode `activityCreated` in der Klasse `SarosSession`

Anhand der Tabelle [3] wird verdeutlicht, dass die korrekte Anzahl an Objekten von `TextEditActivity` der Methode `activityCreated` übergeben und nach der entsprechenden Transformation ebenfalls die korrekte Anzahl an Objekten der Klasse `JupiterActivity` registriert wurde.

Betriebssystem	ALICE(Host)	BOB	CARL
Windows XP	5.000	1.000	1.000
Ubuntu 10.04 LTS ERFOLG	5.000	1.000	1.000
Ubuntu 10.04 LTS FEHLSCHLAG	5.000	1.000	1.000

Tabelle 4: Aufruf der Methode `sendActivity` in der Klasse `SarosSession`

Jeder User versendet 1.000 Jupiteraktivitäten, eine für jede Texteingabe. ALICE ist der Host der Session und transformiert daher die eingehenden Aktivitäten, um sie an die anderen User zu versenden. Wenn BOB 1.000 Aktivitäten an ALICE verschickt, erfolgt eine Transformation und eine Wei-

terleitung an CARL. Die gleiche Variante erfolgt, wenn CARL seine 1.000 Aktivitäten verschickt. Demnach hat ALICE bereits 2.000 von 5.000 Aktivitäten weitergeleitet. ALICE tätigt ebenfalls 1.000 Texteingaben, daraus entstehen 1.000 Jupiteraktivitäten, die an den lokalen Jupiter-Server über die Methode `sendActivity` weitergeleitet werden. Diese werden transformiert und über die gleiche Methode an die anderen User übertragen. Damit wurden die letzten 3.000 Aufrufe der Methode `sendActivity` von ALICE-Seite aus aufgerufen.

Codedurchsicht

Die Klasse `StoppableDocumentListener` fängt alle Eingaben eines lokalen Users ab, und, sofern die `Boolean`-Variablen `enabled` auf `TRUE` und `pause` auf `FALSE` entsprechend gesetzt sind, werden die Eingaben an den `EditorManager` übergeben.
Die Variable `enabled` wird lediglich beim Ausführen von Texteingaben gesetzt, `pause` hingegen erfährt zu keinem Zeitpunkt einer Veränderung.
Dort erfolgt ein Auslesen des Pfades der Datei von der Eingabe. Um dies zu bewerkstelligen, werden alle offenen *EditorParts* nach dem entsprechenden Dokument durchsucht. Es wurde in dem Code angemerkt, dass dieser Vorgang langsam sein würde, da jeder *EditorPart* abgesucht wird. Dies ist aber zu vernachlässigen, da nicht davon ausgegangen werden kann, dass die Anzahl der *EditorParts* so enorm ist, dass sie eine Wirkung auf die Geschwindigkeit von Saros hätten.
Es werden die entsprechenden Texte des Events ausgelesen und es folgt das Erstellen einer `TextEditActivity`. Über die Methode `fireActivity` wird die Aktivität an die `SarosSession` weitergeleitet. Dort erfolgt eine Transformation im Sinne des Jupiter-Algorithmus und ein anschließender Versand der Aktivitäten mit Hilfe der Methode `sendActivity`.

4.3.7 Versand der Aktivitäten

Dieser Abschnitt beinhaltet die Klassen [Abbildung 21]:

- `SarosSession`
- `ActivitySequencer`
- `XMPPTransmitter`

Debugging

Das Debugging gibt hier keinen direkten Aufschluss, die Codedurchsicht bleibt ein wichtiger Aspekt zur Analyse. Grund hierfür ist, dass die Methode `sendActivity` aus der Klasse `SarosSession` die Schnittstelle für Aktivitäten jeder Art sind, seien es welche der Klasse `TextEditActivity`

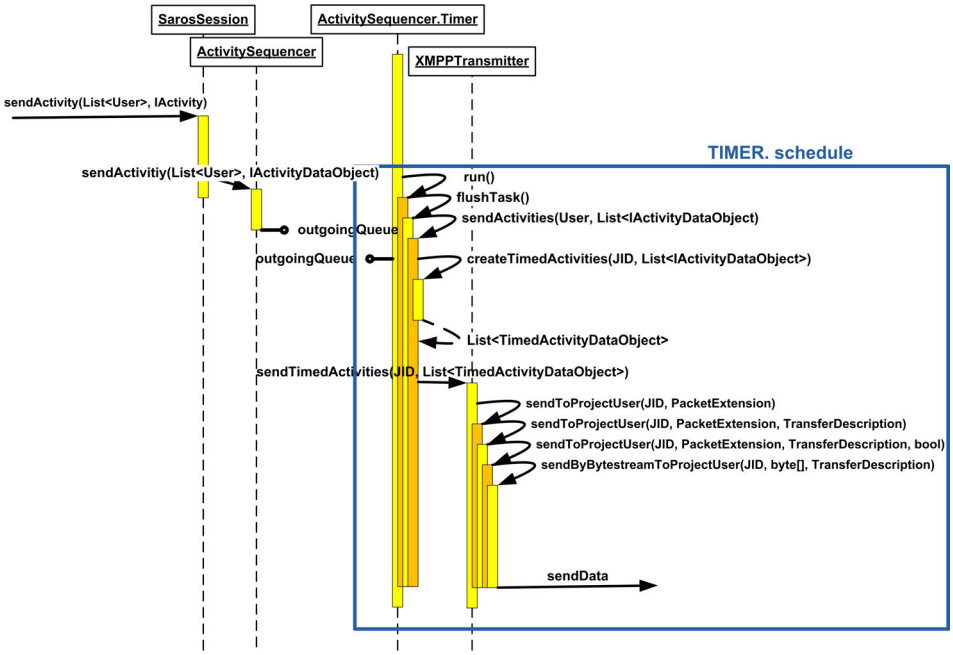

Abbildung 21: Sequenzdiagramm des Versendens von Aktivitäten

oder der Klasse `EditActivity`. Daher erfolgte eine größere Anpassung in den Methoden der Klassen `SarosSession` und `ActivitySequencer`. Dort wurde eine spezielle Ausgabe hinzugefügt, wenn es sich um Objekte der Texteingaben handelte. Es wurde erkannt, dass die Eingaben komplett übergeben wurden und letztlich im `XMPPTransmitter` endeten.

Codedurchsicht

Die Methode `sendActivity` in der Klasse `SarosSession` ist der zentrale Schnittpunkt. Hier werden alle Aktivitäten erneut verarbeitet. Unter anderem ist hier das Zentrum des Need-Based-Features gewesen, dass überarbeitet werden musste. Es folgt noch ein letzter Abgleich, ob es sich um bestimmte Implementierungen des Interfaces `IActivity` handelt. Gleichzeitig wird sichergestellt, dass die Aktivitäten Dokumente betreffen, die in der Session verteilt sind. Letztlich wird, sofern die Bedingungen erfüllt sind, die Aktivität an den `ActivitySequencer` übergeben. Bis auf die Unstimmigkeit mit dem Need-Based-Feature sind keine Schwachstellen zu erkennen.

In `ActivitySequencer` in der Methode `sendActivity` werden Aktivitäten an die `SarosSession` zurückgegeben, wenn der gewünschte Empfänger der lokale User ist, andernfalls wird die Aktivität der `outgoingQueue` hinzugefügt. Diese wird durch einen Timer in stetiger Wiederholung geleert, allerdings mit einem Timeout versehen, so dass ein leichterer Ab-

bruch realisiert werden konnte. In diesem Timer werden die Aktivitäten zu `TimedActivityDataObjects` erweitert und schließlich versandt. Gleichzeitig wird in diesem Timer gesichert, dass mögliche angekommene Aktivitäten, die noch abgespeichert und nicht verarbeitet sind, ausgeführt werden.

Es werden die Aktivitäten gesammelt und dann an den `XMPPTransmitter` übergeben. Diese Klasse dient dazu, die Liste der Aktivitäten in eine *XMPP*-Nachricht zu überführen und anschließend zu Versenden. Sollte die Nachricht nicht entsprechend konvertiert worden und die Nachricht klein genug sein, so wird sie über den *XMPP*-Server verschickt. Andernfalls wird *Sock5* verwendet. Die Verarbeitung des `XMPPTransmitter` entspricht den erforderlichen Maßnahmen.

4.3.8 Versand von Aktivitäten über *XMPP*-Server

Wie im Abschnitt **Netzwerkschicht von Saros** beschrieben, existieren zwei Übertragungsarten (*In-Band (IBB)* und *Sock5(direct)*). Es existiert zusätzlich eine dritte, selten verwendete Variante, die Nachrichten über den Server verschickt, sofern die Nachricht klein genug ist (vorzugsweise weniger als 16378 Zeichen), und welche protokolliert werden können, wenn der Server dafür eingestellt wurde. Diese Variante versendet die Aktivitäten im Sinne einer Chat-Nachricht über den Server.
Die Wahl der Übertragung hängt davon ab, ob eine direkte Verbindung möglich und das zu versendende Paket größer als ein vorgegebenes Limit ist. Sollte dies der Fall sein, so wird *Sock5(direct)* als Übertragung verwendet. Der Nachteil davon ist, dass die verschickten Pakete vom Server nicht protokolliert werden können. Um diese Schwäche zu beseitigen, wurde der Code von Saros dahingehend geändert, dass alle Aktivitäten über den Server verschickt werden um die Protokollierung zu ermöglichen. Sollten auf Grund dieser Änderungen keine Inkonsistenzen bei den Grundtests auftreten, wird der Implementierung, beziehungsweise Verwendung von *Sock5* und *IBB* einer genaueren Betrachtung unterzogen werden.
Dieser Abschnitt beinhaltet die Klassen:

- `XMPPTransmitter`
- `DataTransferManager`

Debugging

Aufgrund des scheinbaren Verlustes von Aktivitäten und der korrekten Funktionsweise des Jupiter-Algorithmus wurde überprüft, wie viele Aktivitäten erfolgreich verschickt und empfangen werden. Dadurch, dass die festgeschriebene Anzahl an Eingaben auf 1.000 lag, konnte eine gute Abschätzung unternommen werden, wie viele Jupiter-Aktivitäten protokolliert werden sollten.

Es stellte sich dabei heraus, dass die Verluste von Aktivitäten nicht der Übertragung geschuldet waren. Die Tests gaben den Verlust von Aktivitäten an, diese wurden aber vollständig vom Server [Tabelle 5] empfangen und weitergeleitet. Das Versenden von Aktivitäten zum Server funktioniert korrekt. Alle zu erwartenden Aktivitäten wurden verschickt, wodurch diese Fehlerquelle von Seiten des Debuggings ausgeschlossen wurde.

Betriebssystem	Testergebnis	Anzahl der JupiterActivity
Windows XP	ERFOLG	12.000
Ubuntu 10.04 LTS	ERFOLG	12.000
Ubuntu 10.04 LTS	FEHLSCHLAG	12.000

Tabelle 5: Vergleich der Daten aus den Log-Daten

Die dokumentierte Anzahl an Aktivitäten ist unabhängig von den Betriebssystemen und dem Erfolg der Tests. Es ist daher davon auszugehen, dass der Grund für die Fehler nicht hier zu finden ist.

Codedurchsicht

Die Codedurchsicht ergab, dass die aufgebauten Verbindungen zwischen den Usern und dem Server prinzipiell TCP-Verbindungen sind, dementsprechend während der Übertragung nichts verloren gehen kann. Auch die Analyse der Methode `sendToProjectUser` in der Klasse `XMPPTransmitter` ergab keine Unstimmigkeiten. Sind die zu versendenden Pakete klein genug, so werden sie als Chat-Nachricht über den *XMPP*-Server versendet, andernfalls werden die *Bytestream*-Varianten angewendet. Es ist sichergestellt, dass bei Fehlschlagen der Verbindungsformen *Sock5(direct)* und *Sock5(mediated)* auf *IBB* umgestellt wird. Sollte auch das fehlschlagen, so wird eine Exception geworfen und es wären entsprechende Nachrichten in den Log-Dateien vorhanden gewesen.

Auch in diesem Abschnitt war keine Unstimmigkeit zu erkennen, es ist vor allem durch das Debugging in diesem Abschnitt anzunehmen, dass der Fehler nicht hier zu finden ist, andernfalls wäre nicht die korrekte Anzahl an Jupiter-Aktivitäten in den Log-Daten des *XMPP*-Servers dokumentiert.

4.3.9 Empfang der Aktivitäten

Dieser Abschnitt beinhaltet die Klassen:

- `BinaryChannel`
- `BinaryChannelConnection`
- `DataTransferManager`

- `XMPPReceiver`
- `ActivitiesHandler`
- `ActivitySequencer`

Debugging

Die Methoden, die während des Empfangs der Aktivitäten verwendet werden, wurden um Ausgaben in die Log-Dateien erweitert, insbesondere auch um spezielle Log-Ausgaben bei dem Abfangen von Exceptions. Die Anzahl der Durchläufe und der verarbeiteten Pakete ließen darauf schließen, dass keine Pakete verloren gehen. Besonders in den Klassen `ActivitiesHandler` und `ActivitySequencer` konnte festgestellt werden, dass die Anzahl der zu erwartenden Aktivitäten genau der Anzahl der verarbeiteten Aktivitäten entspricht. Da auch bei dieser Betrachtung der Log-Dateien, die in den vorherigen Abschnitten beschriebenen möglichen Fehlernachrichten nicht auftraten, kann davon ausgegangen werden, dass es sich hierbei auch nicht um Duplikate von Aktivitäten handelt, da ansonsten die Bedingungen für den Jupiter-Algorithmus verletzt worden wären. Dies hätte Einträge in den Log-Dateien zur Folge gehabt.

Codedurchsicht

Die Klassen `BinaryChannel`, `BinaryChannelConnection` und `DataTransferManager` behandeln den Empfang eines Bytestreams und die Weiterleitung der erhaltenen Pakete. Dabei ruft die Klasse `BinaryChannelConnection` die Methoden der Klasse `BinaryChannel` auf, übernimmt ein `IncomingTransferObject` und übergibt dieses an den `DataTransferManager`. Alle Exceptions, die währenddessen geworfen werden können, werden von `BinaryChannelConnection` korrekt abgefangen. Gleichzeitig wird darauf geachtet, dass mögliche, verwendete Varianten eines `IProgressMonitor` beim Auftreten einer Exception geschlossen werden. Der Empfang des Bytestreams kann als korrekt angesehen werden. Es erfolgte eine genauere Betrachtung bei der Verarbeitung der empfangenen Pakete.

Die Klasse `XMPPReceiver` überführt die über den Bytestream erhaltenen Pakete aus dem `DataTransferManager` an die nächst höher liegende Schicht. Begonnen wird bei der Methode `processIncomingTransferObject`. Dabei wird zuerst in der Methode `processIncomingTransferDescription` abgefragt, ob das erhaltene Paket von einem der hinterlegten Paket-Filter akzeptiert wird, also ob es bereits ein zu verwendendes *XMPP*-Paket ist. Sollte kein Paket-Filter darauf angewendet werden können, wird in der weiteren Methode `processTransferObjectToPacket` das Paket in ein *XMPP*-Paket umgewandelt und dann erneut eine Abfrage startet, ob

betreffendes Paket einem Filter zugeordnet werden kann. Ist dies der Fall, wird es dem entsprechenden Listener übergeben.

In dem Fall des gleichzeitigen Editierens ist dies der `ActivitiesHandler`, in dem mit Hilfe des übergebenen Pakets ein Objekt der Klasse `TimedActivities` ausgelesen wird, welches wiederum eine Liste von `TimedActivityDataObject` liefert. Die weitere Verarbeitung kann nur verhindert werden, in dem die `SessionID` des Objektes von `TimedActivities` nicht der aktuellen Session entspricht. Dies ist nicht möglich, da die entsprechende `SessionID` bei den Einladungen übertragen wird und, sofern die Session nicht verlassen und neu gestartet wird, festgeschrieben bleibt. Existiert momentan eine Session und ist der Sender der Objekte in der Session, dabei reicht es auch, wenn er selbst noch im Einladungsprozess ist, so wird aus den Objekten von `TimedActivityDataObject` die Objekte mit dem Interface `IActivityDataObject` ausgelesen. Jedes einzelne dieser Objekte wird an die Klasse `ActivitySequencer` weitergegeben. Auch in diesem Abschnitt sind keine logischen Fehler zu erkennen.

Der `ActivitySequencer` sammelt in der Methode `exec` alle empfangenen Objekte mit dem Interface `IActivityDataObject` in die `ActivityQueuesManager`. Wenn der Sequencer noch nicht gestartet worden sein sollte, wird eine weitere Verarbeitung verhindert. Wurde er gestartet, so wird die Methode `execQueue` ausgeführt, in der das Objekt von der Klasse `ActivityQueuesManager` schrittweise ausgelesen und als vollständige Liste an die Klasse `SarosSession` übergeben wird.

Die Beeinflussung des Debuggings wird durch den Umstand erzwungen, dass die Elemente von `IActivityDataObject` in ein Objekt der `ActivityQueuesManager` eingefügt wird. Danach erfolgt die Abfrage der Variablen `started`, ob der Sequencer gestartet ist. Sollte der Sequencer zeitweise den Wert der Variablen `started` wechseln, so die Vermutung, könnte dies die Inkonsistenzen erklären, da die Objekte, die sich in der `ActivityQueuesManager` befinden, nicht ausgelesen und damit verarbeitet werden können. Da in der Methode `execQueue` jedes einzelne `IActivityDataObject` ausgelesen wird, wurde in diesem Abschnitt eine Ausgabe platziert. Das Ergebnis zeigte, dass genauso viele Objekte von `IActivityDataObject` in `execQueue` ausgelesen werden, wie in der Methode `exec` des Sequencers hinzugefügt wurde. Daher kann das Problem der Inkonsistenz auch hier nicht entstanden sein können.

4.3.10 Verarbeitung der Aktivitäten

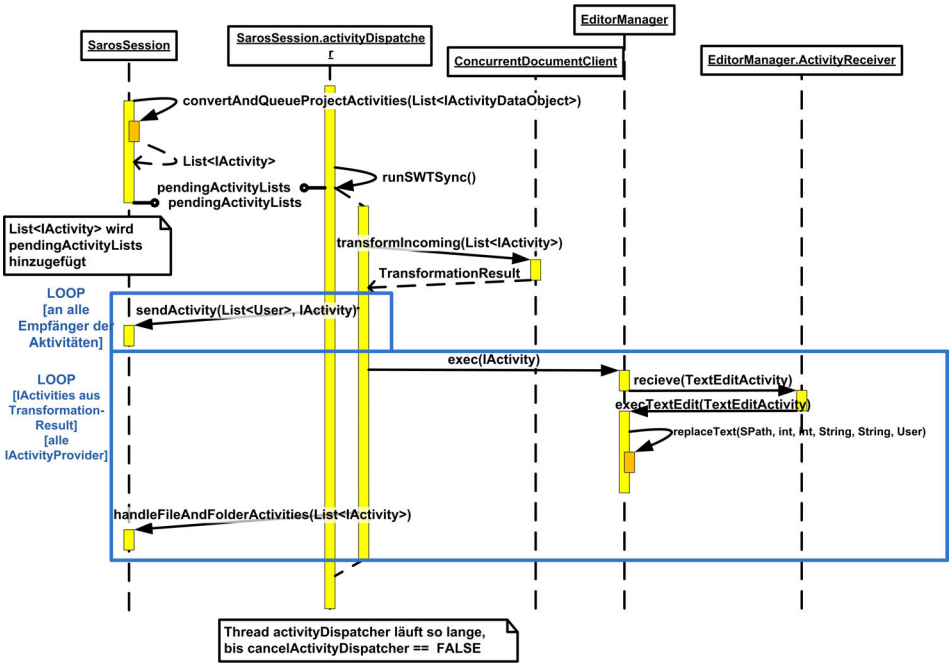

Abbildung 22: Sequenzdiagramm des Verarbeitens von empfangenen Aktivitäten

Dieser Abschnitt beinhaltet die Klassen [Abbildung 22]:

- `SarosSession`
- `EditorManager`

Debugging

Das Debugging ergab in diesem Abschnitt, dass es eine Diskrepanz zwischen der Methode `exec` der Klasse `SarosSession` und dessen `activityDispatcher` gibt [Tabelle 6]. Allein die Ausgabe in der Log-Datei einer eclipse-Instanz zeigte, dass die Häufigkeit, mit der der `activityDispatcher` durchläuft, weitaus geringer ist, als der Durchlauf von `exec`. Die starke Differenz trat bei den Log-Daten der anderen User nicht auf. Gleichzeitig wurde festgestellt, dass die Auslastung der CPU bei diesen Testdurchläufen stark erhöht war. Es ist daher davon auszugehen, dass die hohe Last eine Auswirkung auf dieses Verhalten hat.

Methode	ALICE	BOB	CARL
SarosSession.exec	4648	3528	3478
activityDispatcher.run	4648	2600	3209

Tabelle 6: Diskrepanz zwischen der Ausführung von `SarosSession.exec` und `activityDispatcher.run`

User	Aufruf EditorManager.exec
ALICE	2329
BOB	1693
CARL	2101

Tabelle 7: Anzahl der Aufrufe der Methode `exec` in `EditorManager` durch den `activityDispatcher`

Weitere Unterstützung dieser These wurde in den Log-Daten bei der genaueren Betrachtung der Aufrufe von der Methode `exec` in der Klasse `EditorManager`[Tabelle 7] und der ausgeführten Objekte der Klasse `TextEditActivity` [Tabelle 8] deutlich.

Codedurchsicht

Durch das Ergebnis des Debuggings motiviert, wurde eine genauere Betrachtung der Methode `exec` und dem `activityDispatcher` [Abbildung 23] durchgeführt. Die Methode `exec` konvertiert zuerst alle Objekte von `IActivityDataObject` in `IActivity`. Die Liste der Aktivitäten werden dann in die `pendingActivityLists` überführt. In jedem Durchlauf von dem `activityDispatcher` wird die oberste Liste von `pendingActivityLists` entnommen und einem synchronen SWT-Thread übergeben, der diese Aktivitäten im Sinne des Jupiter-Algorithmus transformiert, an die anderen User verschickt und dann ausführt. Dazu erfolgt eine Übergabe an die Methode `exec` der Klassen des Interfaces `IActivityProvider`, wie auch dem `EditorManager`. Die Methode `exec` von `EditorManager` fragt ab, ob der Sender der Aktivität in der Session vorhanden ist. Ferner wird an den privaten `activityReceiver` der Klasse `AbstractActivityReceiver` die Aktivität übergeben und dort entsprechend ihres

User	Verarbeitete TextEditActivities
ALICE	2000
BOB	1308
CARL	1808

Tabelle 8: Anzahl der Aufrufe im entsprechenden Receiver für `TextEditActivity` in `EditorManager`

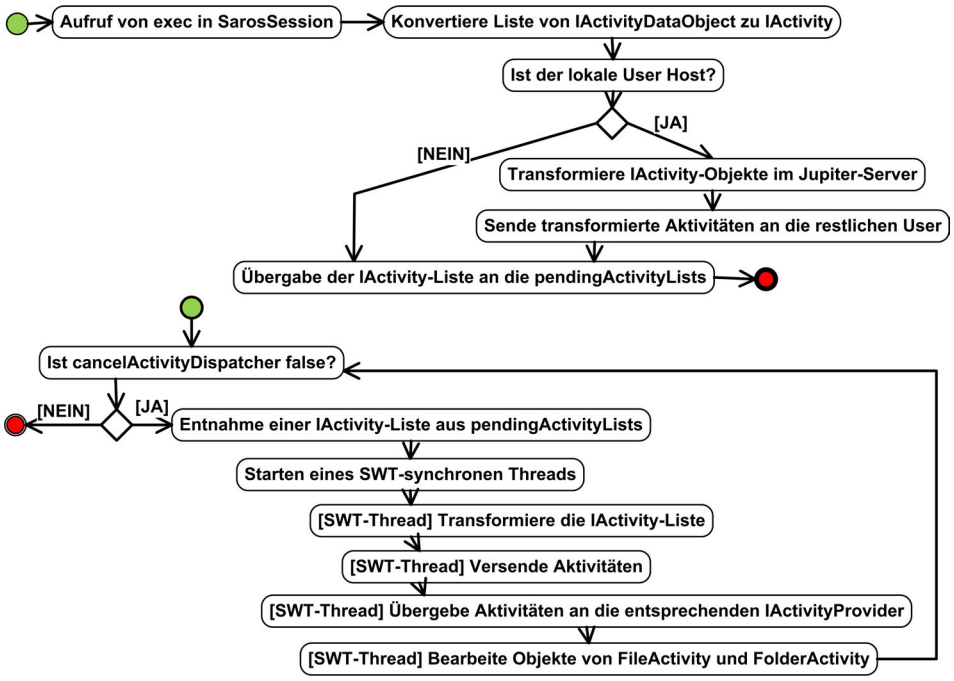

Abbildung 23: Aktivitätsdiagramm des Verarbeitens von empfangenen Aktivitäten

Typs ausgeführt. Bei Objekten der Klasse `TextEditActivity` wird der `StoppableDocumentListener` mit Hilfe der Variable `enabled` während der Ausführung deaktiviert. Der Grund dafür ist, dass beim Ausführen der Aktivität sonst ein Event, wie im Abschnitt **Erstellen einer Aktivität** beschrieben, für diese Eingabe abgefangen und als Eingabe des lokalen Users interpretiert werden würde.

Der `activityDispatcher` selbst ist als *Daemon*-Thread deklariert und wurde bei dem Aufruf des Konstruktors von `SarosSession` gestartet. Es kann daher davon ausgegangen werden, dass der Thread rechtzeitig startete. Auch das Beenden dieses Threads endet erst mit dem Schließen der Session. Ein vorzeitiges Ende oder ein zu später Start von dem `activityDispatcher` kann demnach ausgeschlossen werden. Der Thread wurde nur so lange durchgeführt, bis er abgebrochen wird, nicht aber, bis die `pendingActivityLists` leer ist. Dadurch können noch Aktivitäten auf ihre Ausführung warten, obwohl die Session bereits beendet ist.
Der Grund für die im Debugging festgestellten Probleme findet sich in dem Dasein des `activityDispatchers` als Thread selbst. Da nicht die Java Virtual Machine die Geschwindigkeit eines Threads beeinflussen kann, sondern vielmehr das jeweilige Betriebssystem selbst dies reguliert, kann es geschehen, das bestimmte Threads weitaus langsamer laufen, als Andere. Das

hat zur Folge, dass die Ausführung von Aktivitäten einer Eclipse-Instanz weitaus langsamer vonstatten geht, als bei den Anderen. Ein Ergebnis dessen ist, dass es dem betreffenden User erscheint, als wenn eine Inkonsistenz vorliegt, in Wirklichkeit bedarf es nur noch einer gewissen Zeitspanne, bis die fehlenden Aktivitäten ausgeführt werden.

Die Beeinflussung des Debuggings wurde durch diese Codedurchsicht vorangetrieben. Sollte die Inkonsistenz tatsächlich nur an der zu langsamen Ausführung eines Threads liegen, können die Tests dahingehend geändert werden, dass sie eine längere Zeitspanne nach den Eingaben warten, bevor sie den Inhalt abgleichen. Diese Zeitspanne wurde auf 20 Sekunden gesetzt, ein Zeitraum, der ausreichend sein müsste. Die vorher aufgetretenen Inkonsistenzen wurden damit eliminiert. Damit wird auch erklärt, warum der Watchdog keine Inkonsistenz meldete, denn die Checksummen-Aktivitäten werden auf dem gleichen Pfad wie die Jupiter-Aktivitäten verarbeitet. Werden Jupiter-Aktivitäten demnach verzögert ausgeführt, werden auch Checksummen-Aktivitäten verzögert sein.
Um feststellen zu können, ob der Thread auch ausgeführt wird, wenn die Session bereits beendet ist, wurden die Tests dahingehend geändert, dass nach der letzten Eingabe die Session geschlossen und dann 20 Sekunden gewartet wird, bevor eine Analyse der Eingaben stattfand. Es stellte sich heraus, dass die Tests fehlschlugen. Die Vermutung lag nahe, dass die gespeicherten Aktivitäten nicht mehr ausgeführt wurden.

Ergebnis Die auftretenden Inkonsistenzen lagen an der nicht stattgefundenen Ausführung von Aktivitäten auf Grund eines verlangsamten Prozesses zur Verarbeitung dieser.

4.3.11 Analyse der Fehlersuche

Es war bekannt, das Inkonsistenzen existieren. Dies wurde in zahlreichen Einträgen in dem entsprechendem Bugtracker hinterlegt, ferner auch durch die hier erstellten STF-Tests aufgezeigt. Dabei wurde der komplette Ablauf der Verarbeitung von Texteingaben in dem Projekt Saros in einzelne Abschnitte unterteilt. Jeder Bereich wurde mit Hilfe von Debugging und Codedurchsicht getrennt voneinander betrachtet. Nur wenn sie sich gegenseitig beeinflussten, wurden dies genauer beschrieben.
In diesem Prozess stellte sich heraus, dass bei der Suche nach der entsprechenden Fehlerquelle das Debugging eine entscheidende Rolle spielte, die Codedurchsicht konnte dabei die Vorgehensweise des Debuggings positiv beeinflussen. Ohne das Konzept des Debuggings wäre die Fehlerquelle voraussichtlich übersehen worden.
Die Verwendung der Codedurchsicht kann die Struktur des Codes nahebringen, ist, sofern die entsprechenden Checklisten vorhanden sind, eine nützli-

che Variante, um Strukturfehler zu erkennen. Checklisten verleiten aber zu der Annahme, alle Fehler, die auftreten können, abgehandelt zu haben. Dieser Umstand führt dazu, dass Codeabschnitte als korrekt angesehen werden können, obwohl Fehler vorhanden sind.
Letztlich stellte sich heraus, dass nur durch die Verwendung beider Formen eine korrekte Analyse bewerkstelligt werden konnte.

4.3.12 Grund für den auftretenden Fehler

Unix- und Windows-Betriebssysteme unterscheiden sich fundamental voneinander, angefangen bei der Philosophie bis zur Realisierung [lInsider12]. Nach dem allgemeinen Verständnis heraus ist Java gerade deshalb so populär, weil es unabhängig von dem verwendeten Betriebssystem genutzt werden kann. Für die Nutzung der Sprache Java ist die *Java Virtual Machine*, die auf jedes Betriebssystem abgestimmt ist, unerlässlich. Dabei ist allerdings zu bedenken, das Java im Sinne des Prinzips der "Sandbox" keinerlei Einfluss auf die Funktionsweise des jeweiligen Betriebssystems hat [Mack99]. Demnach ist es für Java nicht möglich, direkten Einfluss auf die Ausführung von Threads zu erlangen. Prioritäten können zwar gesetzt, müssen von den Betriebssystemen aber nicht beachtet werden. Dieses "Sandbox"-Verhalten ist einer der Punkte, weshalb der Fehler auftrat, denn das Verständnis und die Ausführung von Threads unterscheidet sich zwischen Windows und Unix [TN12CH1][TN12CH3].

„Ein Prozess besteht aus mehreren Threads, von denen jeder eine sog. Zeitscheibe vom Windows NT Kernel zyklisch zugewiesen bekommt, während der er abgearbeitet wird." (Zitat von [Krum01, S. 33])

„System-intern wird in Linux allerdings kein Unterschied zwischen einem Thread und einem Prozess gemacht. Ein Thread ist für Linux ein Prozess, der seinen Kontext ganz oder teilweise mit einem anderen Prozess teilt." (Zitat von [Krum01, S. 62])

Auch wenn Unix und Windows Threads als einen *Light-Weight-Process* definieren, so ist deren Verarbeitung unterschiedlich zueinander. Es bleibt die Frage bestehen, inwiefern das Saros betrifft.
Windows startet eine Instanz von Saros, ausgelagert in einem eigenen Prozess [Krum01, S. 32]. Alle Threads, die in dieser Applikation gestartet werden, bleiben in diesem Prozess und können ihn nicht verlassen [Abbildung 24]. Die Geschwindigkeit eines jeden einzelnen Threads der Instanz von Saros ist demnach abhängig von der Geschwindigkeit des Prozesses für diese Instanz.

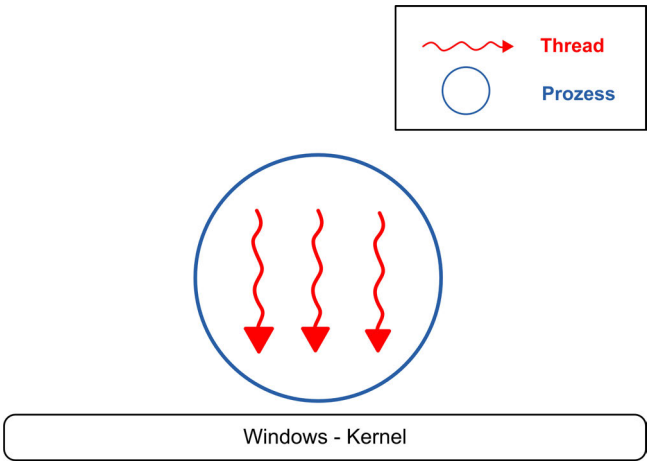

Abbildung 24: Windows-Kernel: Threads in Prozess

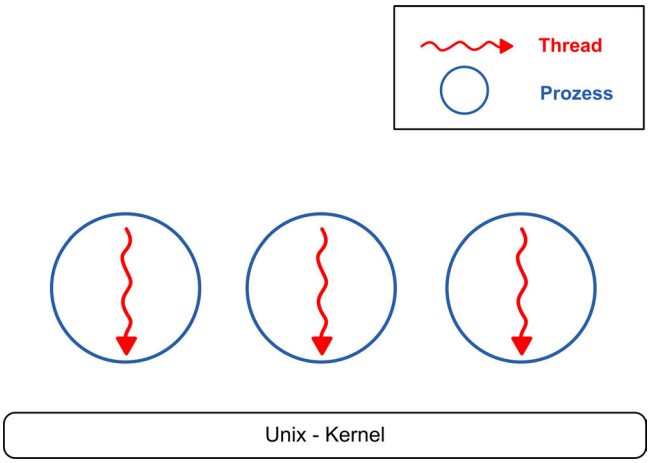

Abbildung 25: UNIX-Kernel: jeder Thread in eigenen Prozess

Unix-Systeme können Threads als eigene Prozesse interpretieren [Abbildung 25], demnach ist nicht sichergestellt, dass die in der Saros-Applikation verwendeten Threads die selbe Geschwindigkeit aufweisen, wie die Applikation selbst. Es existiert auch keine "Zeitscheibe" wie bei Windows, die zyklisch einem Thread zugewiesen wird. Durch dieses vollständige Loslösung dieses Threads von den anderen Threads der Saros-Instanz differenziert die Ausführungsgeschwindigkeit stark von den Anderen. Dadurch entsteht das beobachtete Fehlverhalten.

4.3.13 Fehlerbehebung

Im ersten Schritt wurde versucht, die Dauer des `activityDispatchers` zu verlängern. Dazu erfolgte eine Erweiterung der Bedingung der While-Schleife, die diesen Thread am Leben erhält, einer Erweiterung. Der Thread sollte so lange durchgeführt werden, wie Elemente in der `pendingActivityLists` existieren. Damit wurde die Dauer des Threads erhöht, nicht jedoch die Geschwindigkeit des Programms Saros. Es bestand weiterhin die Problematik, dass der Thread langsamer laufen könnte, als bei den anderen Instanzen.

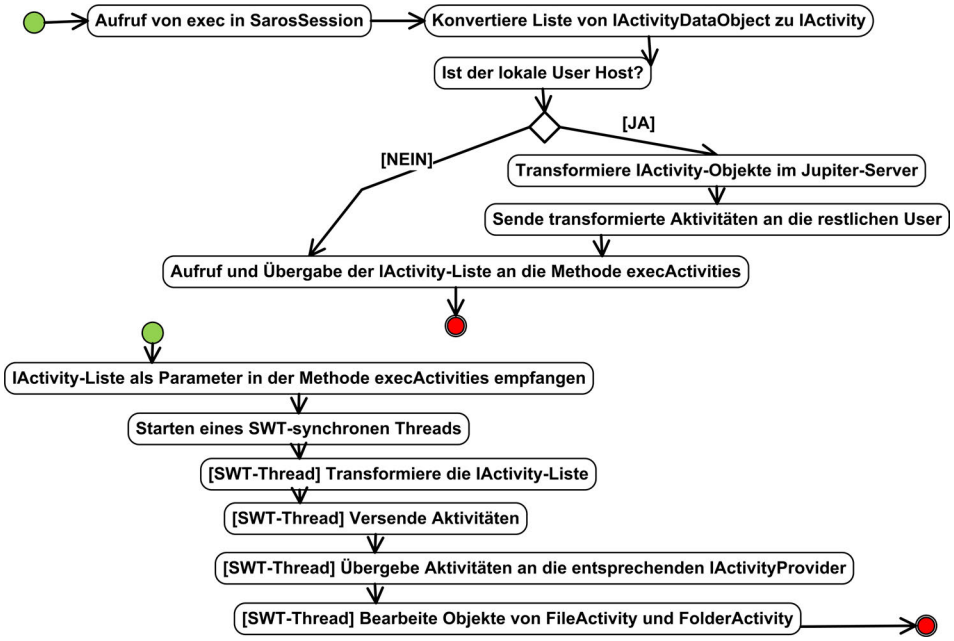

Abbildung 26: Erste Variante zur Behebung des Fehlers

Im nächsten Schritt wurde daher der Thread `activityDispatcher` und die `pendingActivityLists` entfernt. An dessen Stelle kam eine Methode `execActivities` [Abbildung 26], die direkt am Ende von der Methode `exec` der Klasse `SarosSession` ausgeführt wird und die Aktivitäten erhält, die sonst der `pendingActivityLists` hinzugefügt worden wäre. In der Methode `execActivities` wird ein synchroner *SWT*-Thread gestartet, der die übergebenen Aktivitäten transformiert, verschickt und dann ausführt. Der Vorteil ist, dass die Geschwindigkeit der Ausführung von den Instanzen selbst abhängt, dennoch wäre ein asynchroner *SWT-Thread* wünschenswerter, da hier die Prozessor-Leistung besser ausgelastet werden könnte.

In diesem Punkt wurde die Methode `execActivities` dahingehend verändert, dass sie einen asynchronen Thread für die Verarbeitung der Liste mit Aktivitäten startet.
Sofern es sich um normale Jupiter-Aktivitäten handeln würde, wäre dies kein Problem, größere Aktivitäten, wie zum Beispiel welche vom Typ `FileActivity` würden, aufgrund ihres Inhaltes, länger bei der Bearbeitung brauchen, als kleinere. Da es sich um ein asynchrones Verhalten handelt, würden die Jupiter-Aktivitäten mitunter vor den File-Aktivitäten ausgeführt werden.

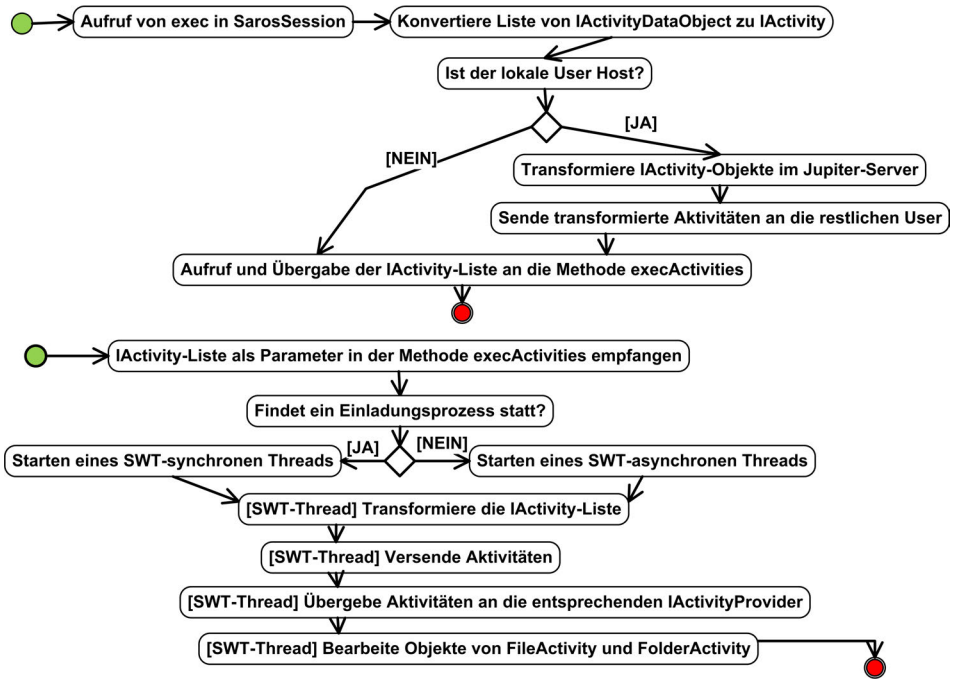

Abbildung 27: Zweite Variante zur Behebung des Fehlers

Es existiert nur ein Augenblick, in dem dies von Bedeutung ist: dem Einladungsprozess. Nur hier existiert das Versenden von File-Aktivitäten. Ansonsten sind nur kleinere Aktivitäten auszuführen. Demnach folgte noch eine letzte Modifikation. Ist der lokale User in einem Einladungsprozess, sprich der `projectNegotiationObservable` enthält Werte, so werden alle Aktivitäten synchron ausgeführt, andernfalls asynchron [Abbildung 27]. Dadurch können die Vorteile des asynchronen Verhaltens genutzt werden, was eine erhebliche Stabilisierung des Editierens bei hoher Belastung der CPU zur Folge hat.

Unabhängig von den Betriebssystemen geben die *STF*-Tests ein konsistentes Verhalten zurück, womit das scheinbar inkonsistente Verhalten eliminiert wurde. Zur Sicherung dieses Verhaltens wurde gleichfalls der `RecoveryWhileTypingTest` durchgeführt, ebenso wie seine Erweiterung mit drei Usern. Auch hier ist sichergestellt, dass die Verarbeitung ein konsistentes Verhalten erzeugt. Es kann demnach von einer erfolgreichen Verarbeitung ausgegangen werden.

5 Fazit

5.1 Auswertung der Fehlersuche

Wie am Beispiele des Watchdogs gezeigt werden konnte, kann das Debugging, wenn es ohne Unterstützung eines zweiten Konzepts verwendet wird, schnell zu falschen Schlussfolgerungen und Ansätzen bei der Lösung des Problems führen. Erst recht, wenn es ein Problem ist, dass auf mehrere Wege gelöst werden kann. Dadurch kann eine umständliche Implementierung vorangetrieben werden, die ein Problem zwar löst, dabei jedoch nicht dem optimalem Weg entspricht und neue Probleme erschaffen könnte.
Dennoch ist das Debugging ein unersetzliches Konzept bei der Fehlersuche und sollte standardmäßig Teil davon sein.

Die Codedurchsicht ist, wie an dem Need-Based-Feature zu sehen war, ein ebenfalls ausschlaggebendes Konzept, hängt jedoch bei verteilten Systemen und nebenläufigem Code stark von dem Fachwissen des Entwicklers ab, der dieses Konzept anwendet. Sind dem Entwickler bestimmte Verhaltensweisen eines Codes oder eines Zielbetriebssystems unbekannt, kann auch die Durchsicht nicht ausreichend sein.
Dies ist an dem Abschnitt des gleichzeitigen Editierens zu erkennen gewesen, in dem die auftretende Inkonsistenz aus der langsameren Verarbeitung eines Threads entsprang. Ohne die gleichzeitige Verwendung des Debuggings wäre der Grund für die Unstimmigkeit beim gleichzeitigen Editieren nicht entdeckt worden. Das Debugging vermochte es, die Fehlerquelle einzugrenzen, die Codedurchsicht hingegen identifizierte die genaue Position.
Anhand dieser gelösten Problematik kann das Ergebnis nur lauten, dass die Verwendung unter gegenseitiger Beeinflussung beider Konzepte eine optimale Variante darstellt, um Quellen von bekanntem Fehlverhalten aufzuspüren.

5.2 Auswertung des gleichzeitige Editierens

Wie sich in den vorherigen Abschnitten gezeigt hatte, konnte die Fehlerquelle beim gleichzeitigen Editieren nur durch die Verwendung der Konzepte des Debuggings und der Codedurchsicht entdeckt werden.
Das Ziel, die Gründe für den Fehler zu lokalisieren, konnte damit erfüllt werden.

Eine andere Möglichkeit, als den `activityDispatcher` abzulösen, ist nicht existent gewesen, da dieser Thread niemals die benötigte Leistung erreichen könnte, ganz gleich, wie er modifiziert wird. Die einzige Variante, dieses Problem zu lösen, bestand darin, die empfangenen Aktivitäten gleich einem eigenen Thread zu übergeben, ohne sie vorher in eine Liste für den `activityDispatcher` zu legen. Dadurch ist sichergestellt, dass alle Aktivitäten prinzipiell gleich verarbeitet werden.

Die Lösung, Aktivitäten während eines Einladungsprozesses synchron zum SWT-Thread, ansonsten asynchron, laufen zu lassen, mag auf den ersten Blick eigenartig anmuten, stellt jedoch sicher, dass die Software auf allen anderen gängigen Systemen stabil läuft, da gerade während des Einladungsprozesses sichergestellt werden muss, dass die ankommenden Aktivitäten synchron laufen.

6 Ausblick

In dieser Arbeit wurde erkannt, dass die Verarbeitung von Aktivitäten stark von der Last des Prozessors abhängig ist. Eine Lösung dessen wurde erstellt, soll jedoch nur den ersten Schritt darstellen. Auf Grund des erkannten Problems muss die Frage gestellt werden, ob eine Reduktion der Features, sowie der Aktivitäten im Allgemeinen, vorangetrieben werden sollte, um die Last auf die CPU zu reduzieren.
Eine Möglichkeit wäre, dem User in den Menüeinstellungen die Fähigkeit zu geben, bestimmte Features deaktivieren zu können. Auch die Möglichkeit, für den Nutzer zweitrangige Aktivitäten, wie die *Viewport*-Anzeige, die Anzeige für einen aktiven *EditorPart* oder die farbliche Hinterlegung des Textes, zu deaktivieren, würde die Last auf die CPU verringern.
Durch die Überprüfung des gleichzeitigen Editierens ist sichergestellt, dass die Überführung von dem Konzept des Distributed Pair Programming zum Distributed Party Programming als erfolgreich angesehen werden kann. Dadurch wird die zu betrachtende Menge des Codes bei zukünftig auftretenden Bugs erheblich reduziert, da die Implementierung des Jupiter-Algorithmus keiner weiteren Betrachtung bedarf, sofern sie nicht modifiziert wird.

Das Need-Based Feature bietet eine stabile Variante, die längere Zeit im Einsatz sein wird. Einer genaueren Betrachtung wäre das `PreSharing` zu unterziehen, da die Übertragung einer Datei parallel zum Einladungsprozesses standardmäßig nicht vorgesehen war, die Wechselwirkungen im laufenden Betrieb noch nicht als sicher betrachtet werden kann.

In der aktuellen Version des Watchdogs werden alle User für die Dauer des Recovery-Prozesses mit einer Schreib-Blockade versehen. Diese Blockade endet erst, wenn der Host die entsprechenden Dateien an den inkonsistenten User versandt hat. Im Abschnitt **Watchdog** wurde die Einführung eines Buffers in den Recovery-Prozess beschrieben. Dadurch wurde ein Ansatz entwickelt, um in zukünftigen Implementierungen die benötigte Zeitspanne einer Schreib-Blockade aller nicht-inkonsistenter User während des Prozesses zu reduzieren. Das Ziel ist es, die User mit einer konsistenten Datenbasis in die Lage zu versetzen, die betroffenen Dateien während des Recovery-Prozesses des inkonsistenten User editieren zu können. Dazu werden alle Aktivitäten der restlichen User in der Session von dem inkonsistenten User in einen Buffer abgelegt, sobald der Recovery-Prozess gestartet wurde. Nach Beendigung soll der Buffer geleert und die enthaltenen Aktivitäten ausgeführt werden.

Abbildungsverzeichnis

1 Überblick von Saros [STech12] 10
2 Überblick von *Shared Session* [SSession12] und *Concurrency Control*[SConc12] 11
3 *In-Band*-Bytestreams 13
4 *Sock5*-Bytestreams (direct) 14
5 *Sock5*-Bytestreams (mediated) 14
6 Aktivitätsdiagramm des Zentrums vom Need-Based-Feature . 17
7 Konzept des neuen Need-Based-Feature[MSpier12] 30
8 Simples Aktivitätsdiagramm des neuen Need-Based-Feature [MSpier12] 31
9 Klassendiagramm des neuen Need-Based-Feature 32
10 Aktivitätsdiagramm von NeedBasedSharingManager 33
11 Aktivitätsdiagramm von PreSharing 35
12 Aktivitätsdiagramm von QuickAddingAFile 37
13 Annahme für den Grund des Problems beim Watchdog 43
14 Verarbeitung der Aktivitäten (veraltete Version) [SWeb12] . . 46
15 Versand von Aktivitäten (veraltete Version)[SWeb12] 47
16 Empfang von Aktivitäten (veraltete Version)[SWeb12] 47
17 Sequenzdiagramm der Transformation von eingehenden Jupiteraktivitäten auf Seiten des Clients 54
18 Sequenzdiagramm der Transformation von ausgehenden Jupiteraktivitäten auf Seiten des Clients 55
19 Sequenzdiagramm der Transformation von eingehenden Jupiteraktivitäten auf Seiten des Servers 56
20 Sequenzdiagramm des Erstellens von Jupiteraktivitäten . . . 57
21 Sequenzdiagramm des Versendens von Aktivitäten 59
22 Sequenzdiagramm des Verarbeitens von empfangenen Aktivitäten 64
23 Aktivitätsdiagramm des Verarbeitens von empfangenen Aktivitäten 66
24 Windows-Kernel: Threads in Prozess 69
25 UNIX-Kernel: jeder Thread in eigenen Prozess 69
26 Erste Variante zur Behebung des Fehlers 70
27 Zweite Variante zur Behebung des Fehlers 71

Literatur

8.1 Saros

[Djemili] R. Djemili, *Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierungverteilter Paarprogrammierung*, Freie Universität Berlin, Inst. für Informatik, Berlin, 2006

[Ziller] S. Ziller, *Behandlung von Nebenläufigkeitsaspekten in einem Werkzeug zur Verteilten Paarprogrammierung*, Freie Universität Berlin, Inst. für Informatik, Berlin, 2009

[Gustavs] B. Gustavs, *Stabilitäts- und Testbarkeitsverbesserungen der Netzwerkschicht in Saros*, Freie Universität Berlin, Inst. für Informatik, Berlin, 2011

[Szücs] Sandor Szücs, *Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung*, Freie Universität Berlin, Inst. für Informatik, Berlin, 2010

[MSpier12] Maria Spiering, *Spezifikation des Need-Based-Features*, Freie Universität Berlin, Inst. für Informatik, Berlin, 2012

[SMSK] Dave Smith, Matthew Miller, Peter Saint-Andre, Justin Karneges, *XEP-0065: SOCKS5 Bytestreams*, http://www.xmpp.org/extensions/xep-0065.html, Stand 30.März 2012

[SWeb12] http://www.saros-project.org/Activities, Stand 30.März 2012

[STech12] http://www.saros-project.org/Technicals, Stand 30.März 2012

[SSession12] http://www.saros-project.org/SharedSession, Stand 30.März 2012

[SConc12] http://www.saros-project.org/Concurrent, Stand 30.März 2012

[KHeld12] Karl Held, *Feature: Need based file synchronization*, http://saros-build.imp.fu-berlin.de/reviews/r/199/, Stand 30.März 2012

8.2 Fehlersuche

[BGoetz] Brian Goetz mit Tim Peierls, Joshua Block, Joseph Bowbeer, David Holmes und Doug Lea, *JAVA - Concurrency in practice*, Kapitel 12, Addison-Wesley, 2009

[SCSP07] Alexander Schill und Thomas Springer, *Verteilte Systeme*, Kapitel 7.4, Springer, 2007

[BPattern12] http://www.ibm.com/developerworks/java/library/j-concurrencybugpatterns/index.html?ca=drs-,
Stand 14.März 2012

8.3 Debugging

[HHeit] Henning Heitkötter, *Erkennung von fehlerauslösenden Eingabedaten mithilfe des Delta-Debugging-Werkzeugs DDinput*,Seminar Qualitätssicherung in der Software-Entwicklung, Westfälische Wilhelms-Universität Münster,2006

[SKleist] Stefan Kleist, *Erkennung von fehlerauslösenden Quelltextänderungen mithilfe des Delta-Debugging- Werkzeugs DDchange*,Seminar Qualitätssicherung in der Software-Entwicklung, Westfälische Wilhelms-Universität Münster,2006

[DD12] http://connection-endpoint.de/wordpress/wp-content/files/deltad.pdf,
Stand 14.März 2012

8.4 Codedurchsicht

[RevT12] http://www.reviewtechnik.de,
Stand 14.März 2012

[SlideQS12] http://www.slideshare.net/creasoft/creasoft-software-qs-review,
Stand 14.März 2012

[Giese05] Jun.-Prof. Dr. Holger Giese, http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Praktika/Softwaretechnikpraktikum/SS05/Vorlesung/SWTPRA05-Vorlesung4.pdf,
Stand 14.März 2012

[WikiCR12] http://de.wikipedia.org/wiki/Review_$Softwaretest$,
Stand 14.März 2012

[TTC12] http://www.ttc.edu.vn/download/materials/Softwareengineering /Implementation/Java%20CheckList.doc,
Stand 14.März 2012

8.5 Betriebssysteme und Java

[Krum01] S. Krumbach, *Synchronisation und Kommunikation von Prozessen und Threads in Java, Windows NT 4.0 und Unix / Linux*, Fachhochschule Köln, Köln, 2001

[Mack99] Holger Mack, *Sicherheit in Java und ActiveX*, Erschienen in: Fox, D.; Horster, P.: *Datenschutz und Datensicherheit - DuD*, Verlag Vieweg,Braunschweig, 1999

[TaWo06] Andrew S.Tanenbaum und Albert S.Woodhull, *Operating Systems - Design and Implementation*, Dritte Edition, Pearson, 2006

[lInsider12] http://www.linuxinsider.com/story/33089.html,
Stand 14.März 2012

[MSDN12] http://msdn.microsoft.com/en-us/library/ms811896.aspx #ucmgch09_topic2,
Stand 14.März 2012

[TN12CH1] http://technet.microsoft.com/en-us/library/bb496993.aspx,
Stand 14.März 2012

[TN12CH3] http://technet.microsoft.com/en-us/library/bb497006.aspx,
Stand 14.März 2012