Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Analyse und Verbesserung der Architektur eines nebenläufigen und verteilten Softwaresystems

Patrick Schlott
Matrikelnummer: 4518872
patrick.schlott@fu-berlin.de

Betreuer: Franz Zieris
Gutachter: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr. Adrian Paschke

Berlin, 1. Oktober 2013

Zusammenfassung

In dieser Arbeit geht es um die Analyse und Verbesserung der Architektur eines bestehenden nebenläufigen und verteilten Softwaresystems. Die Analyse wurde im Softwareprojekt Saros durchgeführt. Saros wird schon seit mehreren Jahren unter hoher Personalfluktuation entwickelt. Dabei wurde viel zu oft vergessen, wichtige Architekturentscheidungen für nachfolgende Entwickler festzuhalten. Dieser Entwicklungsprozess führte dazu, dass sowohl Dokumentation, als auch die Architektur selbst im Laufe der Zeit langsam erodierten.

Im Zuge dieser Arbeit musste ich einen Ansatz finden um unter diesen Umständen die Architektur von Saros analysieren zu können. Bei dieser Analyse konnte ich eine Reihe von Problemen und Architekturbrüchen aufdecken und viele davon auch lösen.

Anschließend habe ich das angesammelte Wissen verwendet um eine Architekturbeschreibung zu erstellen, die Entwicklern in Zukunft dabei helfen sollen die Architektur von Saros besser verstehen zu und in Folge dessen den Einstieg in das Projekt zu erleichtern.

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.

1. Oktober 2013

Patrick Schlott

Inhaltsverzeichnis

1 Einführung **1**
1.1 Saros . . . 1
1.2 Zielstellung dieser Arbeit . . . 3
1.3 Aufbau dieser Arbeit . . . 3

2 Begrifflichkeiten **4**
2.1 Nebenläufigkeit . . . 4
2.1.1 Verteilung . . . 4
2.1.2 Lokale Nebenläufigkeit . . . 5
2.1.3 Produzent-Konsument-Muster . . . 6
2.1.4 Nebenläufigkeit als Eigenschaft des Gesamtsystems . . . 7
2.2 The 4+1 View Model of Architecture . . . 7
2.2.1 Anwendung der Sichten nach Kruchten . . . 9
2.3 JTourBus . . . 9

3 Vorgehen **10**
3.1 Nebenläufigkeit im Saros-System . . . 10
3.2 Vorüberlegung: mögliche Lösungsansätze . . . 10
3.2.1 Ansatz I: Vollständige Systemanalyse . . . 11
3.2.2 Ansatz II: Fehleranalyse . . . 12
3.2.3 Ansatz III: Top-Down Analyse . . . 13
3.3 Vorgehen zur Durchführung einer Top-Down-Analyse . . . 13
3.3.1 Zu erwartende Probleme . . . 14
3.4 Die erste Version der Vorgehensweise . . . 15
3.4.1 Schritt 1: Kernszenario auswählen . . . 15
3.4.2 Schritt 2: Komponenten und Abläufe herausarbeiten . . . 15
3.4.3 Schritt 3: Szenarien verallgemeinern . . . 16
3.4.4 Schritt 4: Nebenläufigkeitseigenschaften analysieren . . . 16
3.4.5 Schritt 5: Auswertung . . . 16
3.4.6 Schritt 6: Behebung gefundener Probleme . . . 16
3.4.7 Schritt 0: Die erste Version der Architektur . . . 16

4 Analyse des Systems **18**
4.1 Schritt 0: Die erste Version der Architektur . . . 18
4.1.1 Informationssammlung . . . 18
4.1.2 Erstellen des ersten *Logical Views* . . . 19
4.2 Das erste Kernszenario . . . 21
4.2.1 Schritt 1: Kernszenario auswählen . . . 22
4.2.2 Schritt 2: Komponenten und Abläufe herausarbeiten . . . 22
4.2.3 Schritt 3: Szenarien verallgemeinern . . . 24
4.2.4 Schritt 4: Nebenläufigkeitseigenschaften analysieren . . . 28
4.2.5 Schritt 5: Auswertung . . . 31
4.3 Kernszenario 2 . . . 35
4.3.1 Schritt 1: Kernszenario auswählen . . . 35
4.3.2 Schritt 2: Komponenten und Abläufe herausarbeiten . . . 36

4.3.3 Schritt 3: Szenarien verallgemeinern 36
4.3.4 Schritt 4: Nebenläufigkeitseigenschaften analysieren . . . 37
4.3.5 Verlassen einer Sitzung 39
4.3.6 Schritt 5: Auswertung 39
4.4 Physische Trennung von Client und Server beim Host 43

5 Implementierung 45
5.1 Umstellung auf eine Client-Server-Architektur für Aktivitäten . . 45
5.1.1 Umsetzung der Client-Server-Architektur 45
5.1.2 Sonderbehandlung von Read-Only Nutzern 48
5.1.3 Problem ITargetedWrapper 49
5.2 Architekturbrüche Aktivitätsverarbeitung 51
5.3 Watchdog 51
5.4 Einladungsprozess 53
5.4.1 Zeitpunkt für das Senden von Aktivitäten definieren . . . 53
5.4.2 Nur der Host darf Projekte hinzufügen 56
5.4.3 Inkonsistenzen nach der *Project Negotiation* 57
5.5 Zusammenfassung und aktueller Stand 58

6 Dokumentation 63
6.1 Architektur Dokumentation 63
6.2 Bewertung der Dokumentation 64
6.3 Dokumentationsverschleiß 65

7 Abschlussbetrachtung 67
7.1 Aktueller Stand und gesetzte Ziele 67
7.2 Ausblick 68

A Architecture Documentation 69
A.1 Components running Session 69
A.1.1 Session Management 71
A.1.2 Activity Sending 73
A.2 Invitation Process 75
A.2.1 Session Negotiation 77
A.2.2 Project Negotiation 78
A.2.3 User States 79

Abbildungsverzeichnis

1 The 4+1 View Model of Architecture 8
2 Alte Komponentenübersicht 19
3 Erste Version der Komponeteneinteilung. 20
4 Aktivitätsdiagramm: Anlegen einer Datei 23
5 Komponenten laufende Sitzung 25
6 Komponente Session Management 27
7 Threads während der Verarbeitung ausgehender Aktivitäten . . . 29
8 Threads während der Verarbeitung eingehender Aktivitäten . . . 30
9 Komponenten während des Einladungsprozesses 37
10 Client-Server-Aufteilung des Systems 46
11 Versenden von Aktivitäten ohne Client-Server-Architektur 47
12 Versenden von Aktivitäten mit Client-Server-Architektur 48
13 Nutzerstati 54

1 Einführung

Die Architektur eines größeren Softwaresystems zu verstehen ist eine der Grundvoraussetzungen dafür, effektiv an der Entwicklung dieses Systems teilnehmen zu können.
Entwickler, die neu in ein Projekt kommen, brauchen einen Überblick darüber, wie das System funktioniert, um zu wissen, welche Änderungen sie durchführen können ohne die Architektur zu verletzen. Zur Vermittlung dieses Wissens können Architekturbeschreibungen dienen. Wenn es eine solche Beschreibungen nicht in angemessener Form gibt, kann im Normalfall auf erfahrene Entwickler zurückgegriffen werden, die ihr Wissen an die neuen Entwickler weitergegeben. Dies wird allerdings Problematisch, wenn es im Projekt dazu noch eine hohe Personalfluktuation gibt. In diesem Fall kann es leicht geschehen, dass es nach einer Weile immer weniger Entwickler gibt, die das System verstehen und neue Entwickler keinen Ansprechpartner haben, der ihnen beim Verstehen des Systems helfen könnte. In diesem Fall haben die neun Entwickler keine Anhaltspunkte darüber, welche Architekturentscheidungen von vorangegangenen Entwicklern getroffen wurden. In Folge dessen beginnt die Architektur langsam zu erodieren, da sie von unerfahrenen Entwicklern aufgrund möglicherweise mangelnder Architekturkenntnisse unabsichtlich verletzt werden kann.
Noch schlimmer wird dieser Punkt, wenn das Softwaresystem eine Reihe von nebenläufigen Prozessen besitzt und über mehrere Netzwerkknoten verteilt ist. Ein solches System ohne eine Architekturbeschreibung zu verstehen kann eine sehr schwere Aufgabe sein, wodurch die Erosion des System noch schneller voranschreitet.
In dieser Arbeit geht es um die Analyse und Verbesserung der Architektur eines nebenläufigen und verteilten Softwaresystems. Das Projekt, in dessen Rahmen diese Arbeit angefertigt wurde war das Softwaresystem Saros, indem sowohl lokale Nebenläufigkeit als auch Verteilung über die einzelnen Netzwerkknoten eine wichtige Rolle spielen.
In diesem Kapitel sollen die Grundlagen für den weiteren Verlauf dieser Arbeit gelegt werden. Zu diesem Zweck stelle ich zunächst das Saros-Projekt in Abschnitt 1.1 vor. In Abschnitt 1.2 werde ich dann die genaue Zielsetzung dieser Arbeit definieren. Im Abschluss dieses Kapitels werde ich den Aufbau dieser Arbeit beschreiben (Abschnitt 1.3) um dem Leser den Einstieg zu erleichtern.

1.1 Saros

Saros [Sar] ist das Softwareprojekt in dessen Umfeld diese Arbeit angefertigt wurde.
Es handelt sich dabei um eine quelloffene Erweiterung für die Eclipse [Ecl] Entwicklungsumgebung. Die Erweiterung ermöglicht verteilte kollaborative Entwicklung an Softwareprojekten. Das bedeutet, dass mehrere Personen gleichzeitig an einem Softwareprojekt arbeiten können ohne dabei am gleichen Ort sein zu müssen.
Zu diesem Zweck können in Saros Sitzungen mit mehreren Teilnehmern gestartet werden. Innerhalb dieser Sitzung können Projekte und Dateien geteilt

werden, woraufhin sämtliche Teilnehmer Zugriff auf diese Ressourcen erhalten und sie editieren können. Saros sorgt dafür, dass sich alle diese geteilten Dateien bei allen Teilnehmern in einem konsistenten Zustand befinden. Konsistent bedeutet hierbei, dass der Inhalt aller geteilten Dateien gleich ist oder zumindest ohne weitere äußere Einwirkungen gleich wird. Um dies zu erreichen, führt jeder Teilnehmer eine lokale Kopie sämtlicher geteilter Dateien. Wenn ein Teilnehmer seine lokale Kopie modifiziert, werden die Änderungen an sämtliche Teilnehmer übertragen und auf deren lokale Kopie angewendet. Auf diese Weise werden die geteilten Dateien bei allen Teilnehmern synchron gehalten und jeder Teilnehmer sieht die Änderungen, die von anderen Teilnehmern durchgeführt wurden.
Um die kollaborative Entwicklung zu ermöglichen, muss daher jeder Knoten (Rechner eines Sitzungsteilnehmers) sowohl Eingaben des Nutzers, als auch eingehende Nachrichten von anderen Nutzern verarbeiten. Diese beiden Ströme müssen synchronisiert werden, um die gemeinsam genutzten Ressourcen in einem konsistenten Zustand zu halten.
Um den Grundgedanken der verteilten kollaborativen Entwicklung zu unterstützen, verfügt Saros darüber hinaus über weitere Funktionen, wie zum Beispiel eine Chatfunktion, oder das Whiteboard, mit dessen Hilfe Notizen und Zeichnungen angefertigt werden können um den anderen Nutzern bestimmte Aspekte besser erklären zu können. Mit Hilfe dieser Funktionen können Entwickler komfortabel zusammenarbeiten und sich zum Beispiel über Implementierungen beraten.
Die Entwicklung von Saros wird hauptsächlich von der Arbeitsgruppe *Software Engineering* des Fachbereichs Informatik der Freien Universität Berlin vorangetrieben. Motivation für die Entwicklung dieses Projekts kam durch die Forschungsschwerpunkte der Arbeitsgruppe, die unter anderem die Untersuchung der Paarprogrammierung umfassen. Ursprünglich wurde Saros daher als Plattform für die Forschungen in diesem Bereich konzeptioniert.
Die Entwicklung wird hauptsächlich im Rahmen von Abschlussarbeiten und Softwareprojekten nach den Prinzipien der agilen Softwareentwicklung durchgeführt. Bei der agile Softwareentwicklung wird, zum Beispiel durch iterative Entwicklung, versucht möglichst effektiv auf sich ändernde Anforderungen reagieren zu können. Einer der Hauptpunkte der agilen Softwareentwicklung ist dabei die These:

> The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. [BBvB+01]

Aufgrund der extrem hohe Personalfluktuation im Projekt fand gerade dieser Austausch von Informationen von Entwickler zu Entwickler nur sehr sporadisch statt. Die Entwicklung läuft nun bereits seit über sieben Jahren und im Laufe der Zeit ist die Architektur immer weiter erodiert und vorhandene Dokumentationen sind inzwischen teilweise dürftig und veraltet.

1.2 Zielstellung dieser Arbeit

Die genaue Zielsetzung war zu Beginn dieser Arbeit noch nicht ganz klar. Die Entwicklung der Zielsetzung kann in Kapitel 3 nachgelesen werden. Die endgültige Zielsetzung in dieser Arbeit bestand aus drei Zielen:

1. **Analyse:** Die Architektur von Saros soll mit Fokus auf Nebenläufigkeitsaspekte analysiert werden.

2. **Verbesserung:** Im Anschluss an diese Analyse sollen einige der gefundenen Probleme gelöst werden um so eine echte Verbesserung der Architektur zu erreichen.

3. **Dokumentation:** Darüber hinaus soll das gesammelte Wissen in geeigneter Weise zusammengetragen und dargestellt werden, um zukünftigen Entwicklern den Einstieg in das Projekt zu erleichtern.

1.3 Aufbau dieser Arbeit

In dieser Arbeit werden ich zunächst in Kapitel 2 die verwendeten Begrifflichkeiten erläutert. Damit sollen die Grundlage für die weitere Arbeit gelegt werden. Anschließend werde ich in Kapitel 3 das Vorgehen beschreiben, das in dieser Arbeit zur Anwendung kam.
Danach werde ich in Kapitel 4 auf die durchgeführte Analyse von Saros eingegangen. Dabei werden auch Probleme beschrieben, die bei dieser Analyse aufgedeckt wurden. (siehe Ziel 1)
In Kapitel 5 gehen ich dann genauer auf die Behebung gefundener Probleme ein. Dabei beschreibe ich, nach welchen Kriterien die zu behebenden Probleme ausgewählt wurden und wie ich sie behoben habe. (siehe Ziel 2)
Nachdem die Analyse und die Umsetzung beschrieben wurden, werde ich im nachfolgenden Kapitel 6 auf die entstandene Architekturbeschreibung eingehen. (Siehe Ziel 3)
Am Ende folgt eine Abschlussbetrachtung dieser Arbeit in Kapitel 7. Dabei werde ich den aktuellen Stand darstellen und aufzeigen ob die gesetzten Ziele umgesetzt wurde.
Im Anhang A findet sich die erstellte Dokumentation.

2 Begrifflichkeiten

In diesem Kapitel werden die grundlegenden Definitionen und Ansätze vorgestellt, die in dieser Arbeit Anwendung fanden. Zunächst wird in Abschnitt 2.1 das Konzept der Nebenläufigkeit definiert und die damit verbundenen Vorteile und Probleme erläutert. Anschließend wird in Abschnitt 2.2 der Ansatz zum Erstellen einer Architektur für Softwaresysteme vorgestellt, der die Grundlage für die Vorgehensweise in dieser Arbeit bildete. Abschließend wird in Abschnitt 2.3 JTourBus vorgestellt. Ein Werkzeug, welches bei der Durchführung der späteren Analyse ein wichtiges Hilfsmittel darstellte.

2.1 Nebenläufigkeit

Nebenläufigkeit ist eine Eigenschaft von Softwaresystem. Ben-Ari definiert ein nebenläufiges System wie folgt:

> A concurrent program is a set of sequential programs that can be executed in parallel[1].[BA06]

Das Konzept der Nebenläufigkeit nimmt im Rahmen dieser Arbeit verschiedene Ausprägungen an. Dabei gibt es zwei Grade der Nebenläufigkeit, die in dieser Arbeit Anwendung finden. *Verteilung* über mehrere Netzwerk-Knoten und *Lokale Nebenläufigkeit* auf einem Knoten.

2.1.1 Verteilung

Verteilung stellt eine Art der Nebenläufigkeit dar, die in verteilten System vorkommt. Tanenbaum und Steen definieren ein verteiltes System wie folgt:

> A distributed system is a collection of independent computers that appears to its users as a single coherent system[2].[TS06]

In Verteilten Systemen kommt es immer zu einer Reihe von nebenläufigen Programmabläufen, da jeder Knoten autonom Aufgaben erledigen kann. Damit sie als zusammenhängendes System funktionieren zu können, müssen die Knoten miteinander kommunizieren, um zum Beispiel Daten zu synchronisieren oder Teilergebnisse von Berechnungen zur Verfügung zu stellen. Typischerweise geschieht dies durch die Verwendung von Nachrichten, die zwischen den Knoten gesendet werden. Eine andere Möglichkeit der Kommunikation wäre zum Beispiel die Verwendung eines gemeinsamer Speichers, wie zum Beispiel einer Datenbank. In dieser Arbeit werde ich mich aber auf Nachrichten als Kommunikationsmittel beschränken.
Häufig werden diese Nachrichten dabei versendet, sobald auf einem Teilsystem eine Operation stattgefunden hat, deren Ergebnis den anderen Knoten mitgeteilt werden muss. Da die einzelnen Teilsysteme unabhängig von einander

[1] Ein nebenläufiges System ist eine Menge von sequentiellen Abläufen, die möglicherweise parallel ausgeführt werden

[2] Ein verteiltes System ist ein Verbund eigenständiger Knoten, die für einen Nutzer als ein einziges, zusammenhängendes System erscheinen.

arbeiten können, weiß ein Empfänger nicht, wann eine neue Nachricht eingeht. Der Empfänger muss also stets bereit sein eine eingehende Nachricht zu empfangen, auch wenn er gerade andere Aufgaben bearbeitet. Häufig werden dafür spezielle Threads verwendet, die Nachrichten in Empfang nehmen und verarbeiten. Näheres dazu in Abschnitt 2.1.2.
Wenn die Reihenfolge, in der Nachrichten bearbeitet werden, wichtig ist, muss darauf geachtet werden, dass diese Reihenfolge nicht durcheinander gerät. Wenn das System aus mehr als zwei Knoten besteht kann eine solche globale Reihenfolge nur dann sichergestellt werden, wenn es einen zentraler Knoten gibt, der für alle Nachrichten als Empfänger dient. Dieser Knoten hat dadurch die Kontrolle über die Reihenfolge der Nachrichten.

2.1.2 Lokale Nebenläufigkeit

Eine weitere Ausprägung der Nebenläufigkeit ist die lokale Nebenläufigkeit. Lokale Nebenläufigkeit liegt dann vor, wenn mehrere Threads auf einem Knoten möglicherweise gleichzeitig ausgeführt werden. Die Verwendung mehrerer Threads auf demselben Knoten hat eine Reihe von Vorteilen. Beispiele dafür sind: [GPB+06]

- **Ausnutzung mehrerer Prozessoren:** Bei einem Knoten mit mehreren Prozessoren können zum Beispiel mehrere Teilrechnungen nebenläufig ausgeführt werden um so schneller zu einem Ergebnis zu kommen als es bei rein sequenzieller Ausführung möglich wäre. Je mehr Prozessoren ein Knoten besitzt, desto mehr Rechenleitung wird verschwendet, wenn wir nur einen dieser Prozessoren nutzen. Bei acht Prozessoren würden so zum Beispiel $\frac{7}{8}$ der Rechenleistung ungenutzt bleiben.

- **Kein Warten auf langsame Prozesse:** Es gibt eine Reihe von Prozessen, die relativ lange dauern. Ein Beispiel dafür sind Eingabe/Ausgabe-Operationen. Ein großer Teil der Ausführungszeit wird dabei einfach nur auf die Antwort des jeweiligen Gerätes gewartet. Diese Wartezeiten können andere Threads nutzen um währenddessen andere Operationen durchzuführen.

- **Kein Blockieren von Nutzeroberflächen:** Programme, die eine Nutzerorfläche besitzen, müssen auch auf Eingaben eines Nutzers reagieren können. Bei rein sequenzieller Abarbeitung muss die Bearbeitung der Nutzereingabe auf dem selben Thread ausgeführt werden, der auch die Eingaben entgegennimmt. Wenn das Programm nun eine langwierige Operation ausführen muss, kann die Oberfläche so lange nicht mehr auf Eingaben reagieren, bis die Operation abgeschlossen ist. Um dieses Problem zu verhindern werden langwierige Operationen häufig in einem anderen Thread ausgeführt. Auf diese Weise kann die Nutzeroberfläche weiterhin auf Anfragen reagieren und der Nutzer merkt nichts davon, dass weitere Operationen im Hintergrund laufen.

Neben diesen Vorteilen gibt es allerdings auch eine Reihe von Risiken, die bei nichtsequenzieller Ausführung von Programmteilen zu beachten sind.

- ***Race Conditions*:** Eine *Race Condition* liegt immer dann vor, wenn die Korrektheit einer Berechnung von der zeitlichen Verzahnung mehrerer Threads abhängt [GPB+06]. Dies führt immer dann zu Problemen, wenn es durch diese Verzahnung dazu kommt, dass Operationen, die eigentlich atomar sein sollten, durch andere Operationen unterbrochen werden. Als atomare Operationen bezeichnet man ein Menge von Teiloperationen, die zusammen eine logische Ausführungseinheit bilden und daher nicht durch andere Operation unterbrochen werden dürfen. Ein Fall einer solchen *Race Condition* sind *check-then-act* Operationen. Als solche werden Operationen bezeichnet, die eine Berechnung auf Grundlage gelesener Daten durchführen [GPB+06]. In der Zeit zwischen Lesen der Daten und Ausführen der entsprechenden Aktion kann ein anderer Thread diese Daten bereits verändert haben. Die erste Aktion wird also auf Grundlage veralteter Daten ausgeführt. Das Erhöhen eines Zählers ist zum Beispiel eine solche Operation.

  Nehmen wir an zwei Threads A und B greifen beide auf einen gemeinsamen Zähler zu. A liest den Wert 2 und erhöht ihn um 1. Bevor er diesen Wert allerdings schreiben kann, ließt B den alten Wert 2 und erhöht ihn auch um 1. Anschließend schreiben beide Threads den neuen Wert. Das Ergebnis ist, dass die Variable am Ende den Wert 3 hat, obwohl sie eigentlich den Wert 4 haben müsste, da sie zweimal erhöht werden sollte.

  Um *Race Conditions* zu unterbinden muss sichergestellt werden, dass atomare Operationen auch als solche ausgeführt werden. Dazu kann man entweder potentiell problematische Operationen auf dem selben Thread ausführen lassen, oder Sperren verwenden um den Zugriff auf gemeinsam genutzte Ressourcen zu synchronisieren. Eine Sperre kann dabei immer nur von einem Thread gehalten werden. Jeder weitere Thread, der versucht sich diese Sperre zu holen muss solange warte, bis die Sperre freigegeben wurde.

- ***Deadlock*:** Die Verwendung von Sperren birgt allerdings ein weiteres Risiko: *Deadlocks*. Ein *Deadlock* liegt dann vor, wenn es eine zirkuläre Abhängigkeit innerhalb des Wartegraphen für Sperren gibt [Lea99]. Das bedeutet zum Beispiel, dass ein *Thread A* die Sperre X hält und versucht Sperre Y zu bekommen. Diese Sperre wird allerdings bereits von Thread B gehalten, der wiederum darauf wartet, das A die Sperre X freigibt. Das Ergebnis ist ein Zustand, indem keiner der beiden Threads weiterarbeiten kann.

2.1.3 Produzent-Konsument-Muster

Eine weitere wichtige Aufgabe von Threads ist es, das System in eine Reihe von Tasks aufzuteilen. Die Ergebnisse dieser Tasks müssen an andere Threads übergeben werden und das mit möglichst wenig Wartezeit. Ein häufig verwendeter Ansatz um dies zu erreichen ist das *Produzent-Konsument-Muster*.
Bei diesem Muster werden Threads in Produzenten und Konsumenten aufgeteilt. Produzenten erzeugen dabei Aufgaben und legen sie zur Verarbeitung in

eine Warteschlange. Konsumenten können sich Aufgaben aus der Warteschlange holen, wenn sie bereit sind neue Aufgaben zu erledigen [GPB+06].
Das Muster trennt also Erzeugung von Aufgaben und deren Bearbeitung voneinander. Auf diese Weise können Abhängigkeiten zwischen Produzenten und Konsumenten aufgelöst werden und Produzenten müssen nicht warten, bis ein Konsument bereit ist die Aufgabe zu übernehmen. Darüber hinaus lässt sich ein solches System gut skalieren, da bei Bedarf weitere Produzenten und Konsumenten hinzugefügt werden.

2.1.4 Nebenläufigkeit als Eigenschaft des Gesamtsystems

Nebenläufigkeit ist immer eine Eigenschaft des gesamten Systems. Im Gegensatz zu einzelnen Funktionen des Systems, gibt es keine isolierte Komponente, die für die Nebenläufigkeit zuständig ist, sondern es gibt mehrere Komponenten, die nebenläufige Prozesse verwenden und/oder mehrere Netzwerkknoten, auf denen unterschiedliche Abläufe stattfinden.
Das Zusammenspiel und die Synchronisierung einer Menge an sequenziellen Programmabläufen in einem Softwaresystem werden im Folgenden als Nebenläufigkeitseigenschaften bezeichnet.
Um solche Nebenläufigkeitseigenschaften des Gesamtsystems effektiv analysieren zu können muss auch ein Überblick über das gesamte System vorliegen.
Um diesen Überblick zu bekommen eignen sich Architekturbeschreibungen und -Diagramme.

2.2 The 4+1 View Model of Architecture

Der *4+1 View Model of Architecture* ist ein Ansatz zur Erstellung von Software-Architekturen, der 1995 von Philippe B. Kruchten, dem damaligen *Director of Process Development* von Rational Software, vorgestellt wurde [Kru95]. Bei diesem Ansatz geht es vor allem um die Aufteilung der Architekturbeschreibung in unterschiedliche Sichten. Jede dieser Sichten legt dabei den Fokus auf klar definierte Aspekt der Entwicklung des Softwaresystems.
Die Idee hinter dieser Aufteilung ist, dass es nicht möglich ist Unmengen an Informationen leserlich in einem Diagramm unterzubringen. Eine solche Vorgehensweise sorgt nur dafür, dass im Endeffekt niemand mehr in der Lage ist, sinnvolle Informationen aus dem Diagramm zu ziehen, da es mit, für die meisten Nutzer unnützen, Informationen überladen ist.
Kruchten definiert dabei vier Sichten. Siehe auch Abbildung 1.
Logical View: Der *Logical View* bildet die Grundlage aller anderen Sichten. Er zeigt die allgemeine Funktionalität, die vom System bereitgestellt wird. Darüber hinaus werden in dieser Sicht die Komponenten des Systems beschrieben, sowie deren Schnittstellen und Aufgaben definiert. Da diese Sicht als Grundlage für die anderen Sichten dient, muss sie erstellt werden, bevor mit der Erstellung anderer Sichten begonnen wird. Der Fokus dieser Sicht liegt auf den statischen Aspekten des Softwaresystems und der äußeren Logik des Systems aus Sicht der Anwender.

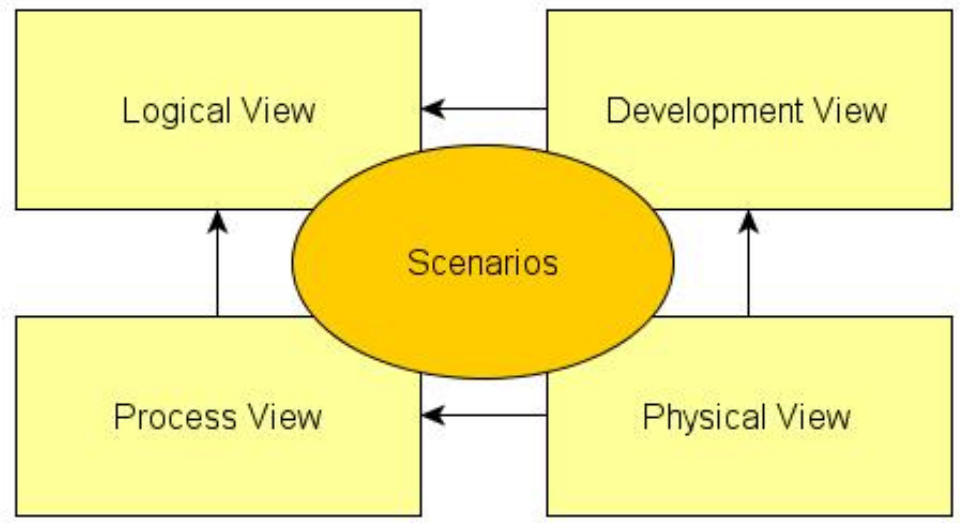


Abbildung 1: Verschiedene Sichten nach Kruchten. Die Pfeile zeigen an, welche Sichten aufeinander aufbauen. Der *Process View* baut zum Beispiel auf dem *Logical View* auf.

Process View: Der *Process View* beschreibt die Abläufe, die innerhalb des Systems stattfinden. Diese Sicht baut auf dem *Logical View* auf und ordnet den einzelnen Komponenten Abläufe zu. Darüber hinaus werden auch gleichzeitig stattfindende Abläufe definiert. Das beinhaltet sowohl Threads und deren Synchronisierung, als auch Nachrichten, die zwischen den physisch getrennten Teilen eines Systems gesendet werden. Der Fokus dieser Sicht liegt damit auf dem laufenden System und den dabei stattfindenden Programmabläufen.

Development View: Der *Development View* beschreibt die für die Entwicklung eines Systems nötigen Infrastrukturen. Dazu gehört die Aufteilung des Systems in eine Paketstruktur und die Definition von unabhängigen Teilsystemen, zu denen einzelne Entwicklergruppen zugeordnet werden können. Darüber hinaus werden in dieser Sicht auch benötigte Bibliotheken und deren Verwaltung berücksichtigt.

Physical View: Der *Physical View* beschreibt die physische Verteilung des Systems. Dazu gehört die Information auf welchen physischen Knoten welche Teile des Systems laufen werden. Außerdem gehört dazu, wie das System später installiert und vertrieben werden kann, sowie welche Voraussetzungen bei einem Kunden vorliegen müssen.

Scenarios: Neben den vier Sichten sieht Kruchten auch noch den Einsatz von Szenarien vor. Diese sind ein wichtiger Teil des Ansatzes und dienen dazu die Entwicklung der vier Sichten voranzubringen. Szenarien sind Anwendungsfälle, die die Benutzung und die Aufgaben des Systems abbilden. Aus diesen Anwen-

dungsfällen lassen sich dann die einzelnen Sichten entwickeln. Die Szenarien bilden dabei den *+1*-Teil des *4+1 View Model of Architecture*, da sie keine eigene Sicht darstellten aber trotzdem ein wichtiger Bestandteil des Ansatzes sind.

2.2.1 Anwendung der Sichten nach Kruchten

Neben den unterschiedlichen Sichten beschreibt Kruchten auch ein Vorgehen, nach dem eine Architektur erstellt werden kann. Der erste Schritt ist dabei das Aufstellen einer Reihe von Szenarien. Diese Szenarien werden nach bestimmten Kriterien ausgewählt. Diese Auswahlkriterien können dabei zum Beispiel die am häufigsten verwendeten Abläufe sein. Es sind aber auch andere Auswahlkriterien denkbar, wie besonders komplexe oder besonders einfache Abläufe zuerst zu modellieren.
Mit Hilfe dieser erstellten Szenarien wird dann zunächst der *Logical View* erstellt. Dabei werden die Komponenten definiert, die für diese Szenarien wichtig sind. Anschließend werden die anderen Sichten auf Basis dieses *Logical Views* erstellt. Nach dem Erstellen der ersten Version, wird dieser Vorgang wiederholt, indem neue Szenarien mit einbezogen werden. Auf diese Weise lässt sich die Modellierung des Gesamtsystems in kleine Schritte aufteilen.

2.3 JTourBus

JTourBus ein Werkzeug, mit dem Touren im Quellcode angelegt und abgerufen werden können [JTo]. Eine Tour besteht aus mehreren zusammenhängenden Haltestellen. Diese Haltestellen werden direkt im Quellcode mithilfe spezieller Annotationen definiert. JTourBus scannt das Projekt nach diesen Annotationen und zeigt anschließend die definierten Touren an. Anhand dieser Touren kann man sich dann von Haltestelle zu Haltestelle durch den Quellcode navigieren und hat so eine Dokumentation, die den Entwickler direkt am Quellcode entlang leiten kann und so über mehrere Klassen gehen kann.
Auch JTourBus wurde am Fachbereich Informatik der Freien Universität Berlin entwickelt und kommt im Quellcode von Saros zur Anwendung. Bei der Erstellung dieser Arbeit waren die mit JTourBus definierten Touren ein wichtiges Hilfsmittel um das System zu verstehen.

3 Vorgehen

In diesem Kapitel wird das genaue Vorgehen beschrieben, nachdem ich diese Arbeit angefertigt habe und die Aspekte, nach denen ich dieses Vorgehen ausgewählt habe. Außerdem werden vorgestellte Ansätze und deren Umsetzungsmöglichkeiten für das Saros-Projekt diskutiert.
Bevor ich mit der eigentlichen Arbeit beginnen konnte, musste ich zunächst einmal verstehen, wo die Probleme in Saros liegen und einen Ansatz finden, diese Probleme bestmöglich anzugehen.
Dabei wird zunächst beschrieben, wieso eine solche Diskussion überhaupt notwendig war (Abschnitt 3.1) und anschließend werden in den Abschnitt 3.2 - 3.4 die diskutierten Ansätze aufgezeigt und der gewählte Ansatz vorgestellt.

3.1 Nebenläufigkeit im Saros-System

Auch in Saros ist Nebenläufigkeit eine Eigenschaft des gesamten Softwaresystems. Dabei spielen sowohl lokale Nebenläufigkeiten, als auch Verteilung eine wichtige Rolle. Dies wird deutlich, wenn man die Hauptaufgaben einer laufenden Sitzung betrachtet.
Die Grundfunktion in einer laufenden Sitzung ist es, das gleichzeitige Editieren geteilter Ressourcen zu ermöglichen. Ressourcen sind dabei Dateien, die zu einem geteilten Projekt gehören. Dabei können an jedem Knoten (hier bestehend aus einer Saros-Instanz und einem Nutzer) geteilte Ressourcen editiert werden. Diese Aktionen müssen den anderen Saros-Instanzen mitgeteilt werden. Dafür werden Nachrichten verwendet. An jedem Knoten müssen somit sowohl Eingaben des lokalen Nutzers als auch eingehende Nachrichten von anderen Saros-Instanzen verarbeitet werden. Diese beiden Ströme müssen synchronisiert werden um sicherzustellen, dass sich die geteilten Ressourcen zu jedem Zeitpunkt in einem konsistenten Zustand befinden.
Damit die Nutzer nicht durch diese Synchronisation in ihrer Arbeit gestört werden, müssen sowohl für das Empfangen, als auch für das Verarbeiten lokaler Eingaben eine Reihe von Threads benutzt werden. Einerseits um immer bereit zu sein neue Nachrichten zu empfangen und andererseits um zu verhindern, dass die Oberfläche der Saros-Instanz blockiert, während Eingaben verarbeitet und gesendet werden.

3.2 Vorüberlegung: mögliche Lösungsansätze

Ausgangspunkt für diese Arbeit war das Wissen darüber, dass es in Saros eine Reihe von Nebenläufigkeitsproblemen gab.
Es gibt im Saros-Projekt eine Reihe von automatisieren Integrationstests. Eine ganze Reihe dieser STF[3]-Tests schlugen zum Zeitpunkt, als ich mit meiner Masterarbeit begann, fehl.
Diese fehlschlagenden Testfälle habe ich zusammen mit meinem Betreuer analysiert. Dazu habe ich unter anderem eine Tabelle aufgestellt, in der ich die

[3]STF steht für Saros Testing Framework und ist das Framework, das in Saros für automatisierte Nutzertests verwendet wird

fehlschlagenden Testfälle, sowie die dahinter liegenden Versagen festgehalten habe. Bei dieser Analyse stellten wir fest, dass zwar nicht alle, aber doch ein paar dieser Testfälle auf Nebenläufigkeitsproblem zurückzuführen sein könnten. Darüber hinaus gab es noch ein paar Testfälle, die nur von Zeit zu Zeit fehlschlugen. Auch dies deutete auf Nebenläufigkeitsprobleme hin.
Außerdem hat die Masterarbeit von Sebastian Starroske [Sta13] eine Reihe von Nebenläufigkeitsproblemen aufgedeckt.
Um meine Arbeit möglichst effektiv gestalten zu können habe ich zu zunächst über eine Reihe von möglichen Vorgehensweisen nachgedacht wie diese Probleme am besten behoben werden können. Diese Ansätze werden im Folgenden beschreiben.

3.2.1 Ansatz I: Vollständige Systemanalyse

- **Idee:** Zunächst habe ich den naiven Ansatz einer vollständigen Systemanalyse untersucht. Dabei werden sämtliche Klassen in Saros-Quellcode nacheinander auf Threadsicherheit[4] hin untersucht, in der Hoffnung die Gesamtheit aller möglichen Thread-basierten Nebenläufigkeitsprobleme beseitigen zu können. Um zu wissen, welche Klassen gleichzeitigen Zugriffen unterliegen können müssen alle Threads in Saros betrachtet werden.

- **Diskussion:** Um die Ausmaße, die dieser Ansatz annehmen würde, abschätzen zu können habe ich zunächst überprüft, wie viele verschiedene Threadobjekte ich dabei beachten müsste.

  Dabei stellte sich heraus, dass es in Saros eine hohe Anzahl unterschiedlicher Threadobjekten gibt, die untersucht werden müssten. Zum Zeitpunkte dieser Betrachtung waren das über 50 verschiedene Threadobjekte, von denen ein Großteil mit großer Wahrscheinlichkeit auch gleichzeitig mit mehreren anderen Threads laufen konnte. Bei genauerer Betrachtung stellte ich zwar fest, dass viele dieser Threadobjekte kurzlebig sind und wenig Funktionalität ausführen, doch all diese möglichen Verzahnungen zu überprüfen erschien utopisch.

  Aufgrund der Komplexität und Größe von Saros ließ sich der hohe Aufwand einer vollständigen Analyse nicht rechtfertigen. Darüber hinaus spielt bei Nebenläufigkeitsproblemen das zeitliche Zusammenspiel der einzelnen Komponenten eine wichtige Rolle. Das heißt, dass selbst dann Versagen auftreten können wenn alle Klassen für sich genommen korrekt sind. Darüber hinaus zieht dieser Ansatz auch die Verteilungsaspekte eine laufenden Saros-Instanz nicht mit ein.

- **Fazit:** Mir war schnell klar, dass eine vollständige Systemanalyse nicht in Frage kommen würde und ich einen Ansatz verwenden muss, bei dem vor allem die Programmteile analysiert werden, die potentiell viele Defekte aufweisen oder sehr wichtig für das Gesamtsystem sind. Aus diesem Grund

[4] Threadsicherheit bedeutet, dass alle Felder, auf die nebenläufig zugegriffen werden kann, gegen zeitlich ungünstige Zugriffe abgesichert sind.

habe ich bei den nächsten Ansätzen versucht genau diese Programmteile zu finden.

3.2.2 Ansatz II: Fehleranalyse

- **Idee:** Als nächstes habe ich den Ansatz betrachtet, einfach die bekannten Versagen zu untersuchen und zu beheben. Dabei müsste ich zunächst bekannte Versagen aus einer Reihe unterschiedlicher Quellen zusammensuchen und anschließend die zu Grunde liegenden Defekte beheben. Es gibt im Saros-Projekt eine Reihe verschiedener Quellen, aus denen bekannte Versagen extrahiert werden können. Diese sind zum Beispiel:

  1. Fehlgeschlagene Komponenten- und Integrationstests
  2. Bugreports im Bugtracker
  3. Automatisch gesammelte Fehlerberichte

- **Diskussion:** Der große Vorteil an diesem Ansatz wäre, dass hierbei Defekte behoben werden können, die wirkliche Versagen in Saros verursachen. Dem steht gegenüber, dass die Liste der gemeldeten Versagen gefiltert werden müsste, da vermutlich nur ein Teil dieser Meldungen auf Nebenläufigkeitsprobleme zurückzuführen ist. Diese Filterung könnte, je nach verwendeter Quelle, einen erheblichen Zeitaufwand bedeuten. Fehlgeschlagene Komponenten- und Integrationstests sind dabei die einzigen Quellen, bei denen eine solche Analyse in angemessener Zeit denkbar wäre. Die Filterung der automatisch gesammelten Fehlerberichte würde einen extrem hohen Zeitaufwand bedeuten, da diese Fehlerberichte nur in unformatierten Log-Files vorliegen. Diese sinnvoll analysieren zu können und dann auch noch auf Nebenläufigkeitsprobleme zu schließen hielt ich für utopisch.

  Die Analyse der Bugreports könnten zwar nach einer längeren Analyse eine Reihe von Problemen zu Tage fördern, doch einen direkten Mehrwert für weitere Entwickler würde eine solche Analyse nicht bringen, da hier nur bereits bekannte Probleme zusammengefasst werden.

  Außerdem ist auch nicht gesagt, dass die wichtigsten Probleme in den vorgestellten Quellen überhaupt vorkommen. Darüber hinaus kommt noch dazu, dass Defekte gerade im Bereich der Nebenläufigkeit schwer zu beheben sind, wenn man kein tiefgehendes Verständnis des Systems besitzt, da diese Probleme erst durch das Zusammenspiel mehrerer Threads auftreten.

  Ein weiterer wichtiger Punkte ist außerdem, dass auf diese Weise nicht verhindert werden kann, dass der behobene Defekte vom nächsten Entwickler wieder unbeabsichtigt eingebaut wird.

- **Fazit:** Auch dieser Ansatz wies eine Reihe von Problemen auf. Gerade die letzten beiden Punkte (Nachhaltigkeit und Emergenz) stellen dabei große Probleme dar. Die hohe Personalfluktuation im Saros-Projekt macht die

Nachhaltigkeit sogar noch um einiges wichtiger, da neue Entwickler vermutlich wenig Hintergrundwissen über die behobenen Defekte haben, wodurch es leicht passieren kann, dass behobene Defekte wieder auf ähnlich Weise eingefügt werden. Besser wäre ein nachhaltigerer Ansatz.

3.2.3 Ansatz III: Top-Down Analyse

- **Idee:** Bei der *Top-Down*-Analyse wird versucht aus einem höheren Verständnis des Systems heraus auf mögliche Nebenläufigkeitsprobleme zu schließen und anschließend sicher zu stellen, dass diese Probleme nicht auftreten. Dabei kann zum Beispiel von gegebenen Spezifikationen zur Nebenläufigkeit oder Architekturbeschreibungen ausgegangen werden. Diese werden dann hergenommen um deren Einhaltung überprüfen zu können [Sto06].

- **Diskussion:** Der Vorteil an diesem Ansatz wäre, dass Defekte im Bereich der Nebenläufigkeit effektiver behoben werden können, wenn man vorher ein Verständnis des Gesamtsystems erlangt hat.

  Problematisch ist hierbei allerdings, dass es kaum vorhandene Spezifikationen gibt. Diese müssten also zunächst auf Grundlage des erwarteten Verhaltens innerhalb einer Sarossitzung neu erstellt werden, was einen zusätzlichen Zeitaufwand bedeutet.

- **Fazit:** Sich zunächst einen besseren Überblick über die Abläufe in Saros zu verschaffen schien mir eine gute Idee zu sein. Darüber hinaus ist auch das beschriebene Problem eigentlich ein weiterer Punkt, der für diesen Ansatz spricht, wenn man weiterdenkt. Dieser Punkt bedeutet nämlich, dass selbst dieser zusätzliche Zeitaufwand einen Mehrwert für zukünftige Entwickler bringen würde. Die erstellten Spezifikationen sind dann nämlich keine Einwegwerkzeuge, die nur in dieser Arbeit Anwendung finden, sondern sie können nachhaltig dazu dienen neuen Entwicklern in das Saros-Projekt einzuführen und so die Einführung neuer Defekte teilweise verhindern.

Nachdem ich die unterschiedlichen Ansätze untersucht hatte, war ich davon überzeugt, dass eine *Top-Down*-Analyse der beste Weg sei bei dieser Arbeit vorzugehen.

3.3 Vorgehen zur Durchführung einer Top-Down-Analyse

Nachdem ich zum Schluss gekommen war diese Arbeit in Form einer *Top-Down*-Analyse durchzuführen, habe ich eine Literaturrecherche über verschiedene Ansätze zu diesem Thema durchgeführt. Dabei konnte ich eine Reihe von Ansätzen zur Erstellung einer Architekturbeschreibung finden. Viele dieser Ansätze basierten dabei mehr oder weniger auf dem bereits vorgestellten *The 4+1 View Model of Architecture* 2.2, wie beispielsweise das Buch *Software Systems Architecture* [RW05], indem die Idee von Kruchten aufgegriffen und

um einige Sichten und Ratschläge zur praktischen Durchführung erweitert wurde. Kruchten und verwandte Ansätze boten eine gute Beschreibung, wie ein Softwaresystem mit Hilfe von Architekturbeschreibungen erstellt werden kann. Vor allem der *Logical View* und der *Process View* im Allgemeinen, sowie der *Physical View* für bestimmt Aspekte der Verteilung, schienen sehr gut zu meiner Arbeit zu passen. Auch das beschriebene iterative Vorgehen erschien mir sinnvoll.

3.3.1 Zu erwartende Probleme

Es gab allerdings auch eine Reihe von Problemen, die eine direkte Umsetzung dieser Ansätze erschwerte.

- **Saros ist ein bestehendes Softwaresystem:** Das Hauptproblem, am Ansatz von Kruchten und anderen darauf basierenden Ansätzen war, dass sie alle auf die Erstellung einer Architekturbeschreibung für neue Softwaresysteme ausgerichtet waren. Die Beschreibung der einzelnen Schritte ist zwar besonders im [RW05] sehr detailliert, doch beziehen sich die meisten dieser Beschreibungen darauf, wie man am besten eine Architektur für neue Softwaresysteme finden kann.

  Saros ist dagegen ist bestehendes System, welches schon seit mehreren Jahren entwickelt wird und bereits eine implizite Software-Architektur besitzt, auch wenn diese nicht gut dokumentiert ist. Es geht also nicht darum eine neue Architektur zu erstellen, sondern eine vorhandene genauer zu spezifizieren.

- **Wenig explizite Spezifikationen:** Darüber hinaus setzen diese Ansätze eine große und detailliert Anzahl an Spezifikationen voraus und es gibt wenig Hinweise dazu, wie sich diese Ansätze für eine, wie im Saros-Projekt verwendete, *Agile Softwareentwicklung* sinnvoll umsetzen lassen.

  Saros ist ein bereits seit mehreren Jahren von einer großen Zahl unterschiedlicher Entwickler entwickeltes Softwaresystem. Die meisten Entwickler sind dabei nur während der Zeit ihrer Abschlussarbeit für Saros tätig (3-6 Monate) und verlassen das Projekt danach. Häufig bleiben nach Verlassen des Projektes die Abschlussarbeit und der Programmcode selbst als einzige Repräsentation dieses Wissens zurück, sodass implizites Wissen und implizite Spezifikationen der Architektur in Saros verloren gehen.

  Diese Entwicklung sorgte in der Vergangenheit dafür, dass Saros sich in großen Teilen zu einem Altsystem entwickelt hat. Seacord, Lewis and Plakosh definieren ein Altsystem wie folgt:

  > Software systems become legacy systems when they begin to resist modification and evolution[LPS03].

  Im Falle von Saros zeigt sich dieser Punkt dadurch, dass die Dokumentation im Laufe der Zeit immer lückenhafter und veralteter wurde und wenig Ansprechpartner vorhanden sind. Dazu kommt auch, dass sich selbst von

dieser lückenhaften Dokumentation nur ein sehr geringer Teil mit Architekturfragen beschäftigt.

3.4 Die erste Version der Vorgehensweise

Nachdem ich die verschiedenen Ansätze und die damit verbundenen Einschränkungen und Probleme analysiert hatte, habe ich beschlossen, die Grundidee von Kruchten zu verfolgen, diese aber an das Saros-Projekt anzupassen. Aus diesem Grund habe ich zunächst einmal eine vorläufige Vorgehensweise ausgearbeitet. Dieses Vorgehen wollte ich dann zunächst an einem möglichst einfachen Szenario testen um es anschließend anpassen zu können, wenn es nicht die gewünschten Ergebnisse liefern sollte.
Dazu habe ich das Verfahren in eine Reihe von Schritte aufgeteilt. Die Schritte sollen, wie von Kruchten beschrieben, nacheinander für einzelne Kernszenarien ausgeführt werden. Kernszenarien sind Anwendungsfälle, die die grundlegenden Funktionen von Saros darstellen und dienen als Ausgangspunkt für die Systemanalyse und als Anforderungsersatz zur Bestimmung des Soll-Verhaltens.
Die Schritte 1 bis 5 werden dabei mehrfach, für eine Reihe verschiedener Kernszenarien durchlaufen und dienen dem Aufdecken von Problemen. Der anschließende Schritt 6 dient dann der Behebung dieser Probleme.

3.4.1 Schritt 1: Kernszenario auswählen

Im ersten Schritt wird ausgewählt, welches Kernszenario untersucht werden soll. Unterschiedliche Aspekte können dabei die Auswahl beeinflussen, wie zum Beispiel Komplexität des Szenarios oder Wichtigkeit für die Funktionsfähigkeit der Software. Dieser Schritt entspricht der Auswahl von Szenarien bei Kruchten.

3.4.2 Schritt 2: Komponenten und Abläufe herausarbeiten

Als nächstes werden die System-Komponenten herausgearbeitet, die für das gewählte Kernszenario wichtig sind oder durchlaufen werden. Um diese zu finden, sollten Dokumentationen zu Rate gezogen, oder wenn nötig der Verlauf dieses Kernszenarios im Programmcode selbst nachvollzogen werden.
Anschließend sollen die Aufgaben der einzelnen Komponenten klar definiert und Schnittstellen zwischen den Komponenten herausgearbeitet und spezifiziert werden. Dieser Schritt entspricht der Ausarbeitung des *Logical Views*.
Außerdem sollen in diesem Schritt auch die internen Abläufe für dieses Szenario dargestellt werden, um den Komponenten so Aufgaben zuordnen zu können. Dies umfasst auch die Verteilung über die Knoten, sowie die Synchronisation im System verwendeter Threads. Dieser Schritt entspricht dem Erstellen des *Process Views* und des *Physical Views*.
Das Erstellen des *Logical Views* und des *Physical Views* bzw. *Process Views* werden hier als ein Schritt betrachtet. In diesem Schritt geht es um die Erstellung der Sichten für ein konkretes und detailliertes Szenario.
Um die Sichten besser Verstehen zu können sollten in diesem Schritt auch entsprechende Diagramme erstellt werden.

3.4.3 Schritt 3: Szenarien verallgemeinern

Als nächstes folgt ein Schritt, indem geschaut wird, ob sich das analysierte Szenario verallgemeinern lässt. Das bedeutet, dass weitere Anwendungsfälle gesucht werden, deren Bearbeitung ähnlich abläuft. Diese Szenarien werden dann zu eine neuen, übergeordneten Szenario zusammengefasst, welches die konkreten Szenarien auf eine höhere Ebene abstrahiert. Auf diese Weise lässt sich eine bessere Abdeckung an Kernszenarien mit weniger Aufwand erreichen. Außerdem hilft eine solche Verallgemeinerung dabei ein besseres Verständnis des Gesamtsystems zu erhalten.
Die in Schritt 2 erstellten speziellen Diagramme sollten in diesem Schritt auf die höhere Abstraktionsebene angepasst werden.

3.4.4 Schritt 4: Nebenläufigkeitseigenschaften analysieren

Nachdem die Komponenten und Abläufe verstanden und klar dargestellt wurden, kann nun geschaut werden, welche Nebenläufigkeitenseigenschaften für das gewählte Szenario in Betracht gezogen werden müssen. Je nach Szenario muss der Fokus dabei auf anderen Aspekten liegen. So kann es zum Beispiel bei bestimmten Szenarien eine großen Anteil an lokalen Nebenläufigkeiten geben, die es zu beachten gilt, während es bei anderen vor allem um die Verteilungsaspekte geht. Je nach Fokus müssen dabei ein oder mehrere angemessene Darstellungsmöglichkeit gefunden werden. Es sollte allerdings darauf geachtet werden, dass für unterschiedliche Aspekte auch unterschiedliche Darstellungen verwendet werden.

3.4.5 Schritt 5: Auswertung

In diesem Schritt sollen die durchgeführten Analysen und erstellten Diagramme auf Probleme im Softwaresystem hin untersucht werden und bereits bei der Analyse aufgefallene Probleme sollen genau dokumentiert werden. Ein Problem liegt dabei vor, wenn sich Soll- und Istzustand unterscheiden. Dazu müssen eventuell auch Gedanken über das Soll-Verhalten angestellt werden und mit dem gesehenen Ist-Zustand verglichen werden. Nach Abschluss dieses Schrittes wird wieder mit Schritt 1 weitergemacht, wenn noch weitere Kernszenarien analysiert werden sollen.
Bisher erstellte Diagramme können dabei weiterverwendet werden, müssen aber unter Umständen angepasst werden.

3.4.6 Schritt 6: Behebung gefundener Probleme

Der letzte Schritt dient dann der Behebung der gefundenen Probleme. Hier werden die wichtigsten Probleme angegangen um eine Verbesserung des Systems erreichen zu können.

3.4.7 Schritt 0: Die erste Version der Architektur

Um mit dem beschriebenen Vorgehen beginnen zu können musste ich mir erst einmal eine Überblick über das Gesamtsystem verschaffen. Da es sich bei Sa-

ros um ein bereits seit mehreren Jahren entwickeltes System handelt, musste ich dabei nicht, wie bei Kruchten vorgesehen, eine vollkommen neue Architektur entwickeln. Um die bisherige Architektur nicht zu ignorieren wurde vor der Auswahl der ersten Szenarien noch ein *Schritt 0* eingefügt, indem ich versucht habe soviel wie möglich über die bisherige Architektur aus vorhandenen Dokumentationen zu erfahren.

4 Analyse des Systems

In diesem Kapitel beschreibe ich, wie ich die eigentliche Analyse des Systems durchgeführt habe und welche Ergebnisse diese Analyse gebracht hat. Dazu stelle ich zunächst in Abschnitt 4.1 dar, wie ich die vorhandene Dokumentation verwendet habe, um einen ersten Überblick über das System zu bekommen. Anschließend werde ich in den Abschnitten 4.2 - 4.3 auf die einzelnen Kernszenarien und die damit verbundene Analyse eingegangen. Dabei werde ich sowohl Erkenntnisse, als auch Probleme beschrieben, die während dieser Analyse aufgedeckt wurden.
Bei den einzelnen Szenarien geh ich so vor, dass ich diese zunächst analysiere (Schritte 1-4) und erst anschließend die Probleme genau benenne und diskutiere, die bei dieser Analyse aufgefallen sind (Schritt 5).
Die Lösung dieser Probleme folgt dann im nächste Kapitel 5.

4.1 Schritt 0: Die erste Version der Architektur

In diesem Abschnitt beschreibe ich, wie ich die vorhandenen Informationen über die Architektur in Saros gesammelt habe (Abschnitt 4.1.1) um daraus einen ersten Überblick über das System zu erstellen (Abschnitt 4.1.2).

4.1.1 Informationssammlung

Aufgrund meiner, bis zu diesem Zeitpunkt geringen Kenntnisse über den Aufbau von Saros, musste ich zunächst einmal eine Reihe von Quellen sichten, in der Hoffnung, dass sich die benötigten Informationen aus diesen ableiten lassen. Zunächst habe ich die Website des Saros-Projektes nach Informationen über die in Saros verwendete Architektur durchsucht.
Auf der Website gab es einen Abschnitt namens *Technical Documentation*. Dieser Abschnitt soll Entwicklern helfen, einen Einstieg in das Projekt zu erhalten. Unter Anderem gab es dort eine Darstellung der Komponenten, aus denen sich Saros zusammensetzt. Diese Einteilung in Komponenten ist im Rahmen der Diplomarbeit *Verbesserung und Pflege der Dokumentation der DPP-Software Saros* von Meike Johannsen [Joh11], auf Basis eines Artikels über Saros [SOBS10], entstanden und sollte dazu dienen neuen Entwicklern den Einstieg in Saros zu erleichern.
Neben der Website habe ich noch zwei weitere Quellen in der Hoffnung untersucht, aus ihnen weitere Informationen über die Komponenten in Saros ableiten zu könnten. Diese Quellen waren einerseits die Paketstruktur in Saros selbst und andererseits eine Menge an *@Component*-Annotationen, die sich in einigen Klassen finden ließen. Zu Beginn dieser Informationssammlung war noch nicht klar, was die genaue Bedeutung dieser Annotationen war.
Diese Quellen standen teilweise im Widerspruch zueinander, sodass ich zunächst einmal analysieren musste, welche dieser Quellen als Grundlage für meine Architekturübersicht dienen sollte. Dabei bemerkte ich schnell, dass die Paketstruktur und die Einteilung auf der Website viele Gemeinsamkeiten aufwiesen. Der Hauptunterschied lag darin, dass die eigentlichen Komponenten anders benannt

waren als die entsprechenden Pakete. Die Annotationen dagegen unterschieden sich deutlicher von den anderen beiden Quellen. Darüber hinaus war auch nur ein Teil der Klassen annotiert worden, weswegen ich diese Quelle größtenteils ignorierte. Später habe ich herausgefunden, dass diese Annotationen nur für Klassen definiert wurde, die vom Pico-Container erstellt wurden.[5] Die Annotationen wurden für ein Diagramm aller *Singletons* in Saros verwendet, dass allerdings so undurchsichtig ist, dass niemand sinnvolle Informationen aus ihm ziehen konnte.
Darüber hinaus gab es noch eine Diplomarbeit, deren Ziel es war Saros zu modularisieren [Bel11]. Leider brachte diese Arbeit mich wenig weiter, da sie Hauptsächlich auf Möglichkeiten eingeht eine solche Modularisierung technisch umzusetzen und keine tatsächliche Modularisierung vorschlägt.

4.1.2 Erstellen des ersten *Logical Views*

Aus den gesammelten Informationen konnte ich dann eine erste Version des *Logical Views* erstellen. Diese basierte hauptsächlich auf den Komponenten-Informationen, wie sie auf der Website (siehe Abbildung 2) dargestellt wurden, da diese Einteilung auf den ersten Blick als am sinnvollsten erschien. Siehe dazu auch Abbildung 3. Diese erste Version spiegelt meinen Wissensstand, nach dem in Anschnitt 4.1.1 beschriebenen Sammeln von Informationen wieder.

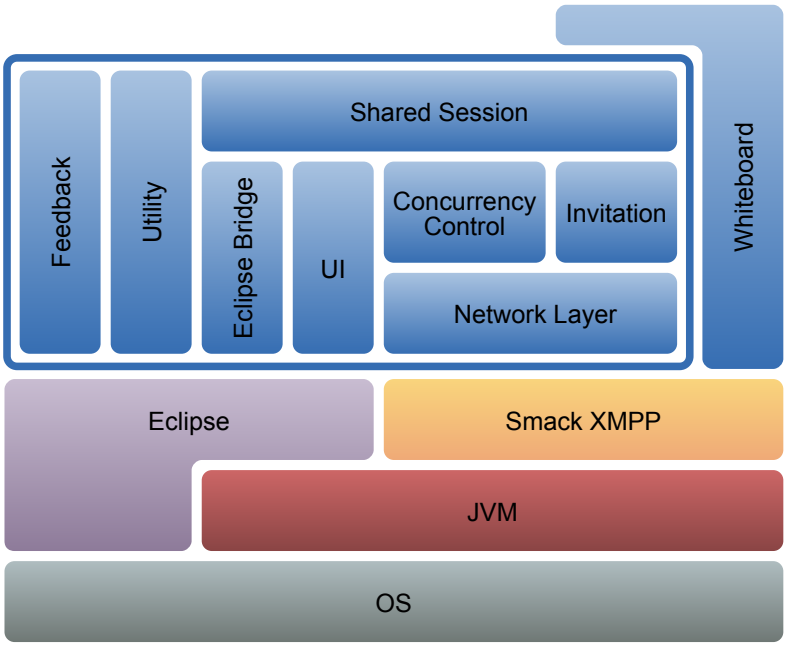


Abbildung 2: Komponentenübersicht, wie sie auf der Website zu finden war. Stand Mai 2013.

[5] Pico-Container ist die Bibliothek, die in Saros zum Verwalten von *Singletons* verwenden wird [pic]. *Singletons* sind Klassen, von denen es zur Laufzeit nur ein Objekt gibt.

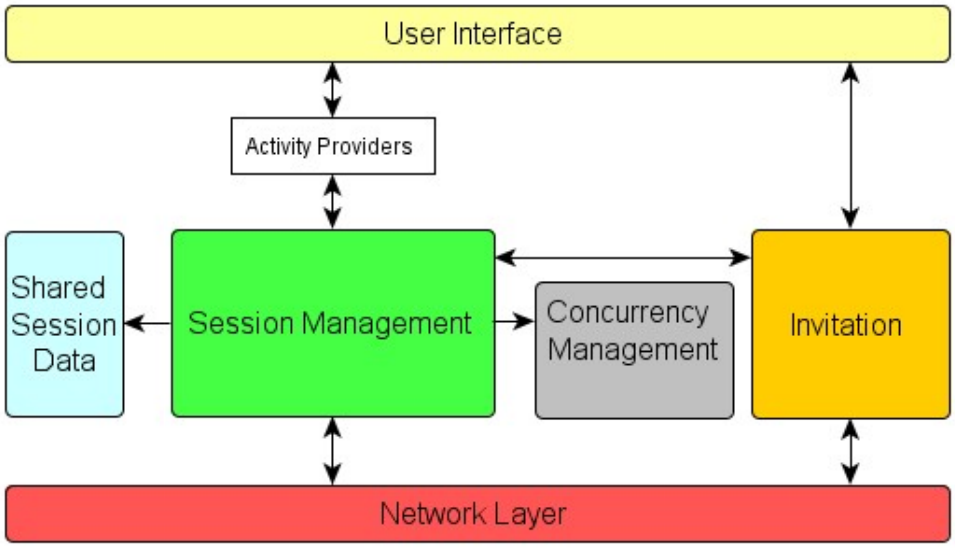


Abbildung 3: Erste Version des Komponenten-Diagramms. Bedeutung Pfeile: Statische Abhängigkeit. Doppelpfeile stellen statische Abhängigkeiten dar, bei denen entweder beide Komponenten von einander abhängig waren oder die Zugriffsrichtung aus der betrachteten Dokumentationen nicht genau hervor gingen. Stand Mai 2013.

Im Gegensatz zur Darstellung auf der Website habe ich mich dazu entschieden einige Sonderkomponenten zu entfernen, die mit dem eigentlichen Kernbetrieb in einer laufenden Sitzung wenig zu tun hatten, wie zum Beispiel die auf der Website zu findende *Feedback* Komponente. Darüber hinaus wurden auch einige Komponenten zusammengefasst und statische Abhängigkeiten zwischen den Komponenten dargestellt, die in der Darstellung auf der Website komplett fehlten. Außerdem habe ich mich auf das Saros-System als solches konzentriert und die unterliegenden Schichten bewusst weggelassen um so ein verständlicheres Bild zu erhalten.

- **Session Management:** Das Kernstück von Saros bildet die Komponente *Session Management*. Diese ist zuständig für die Verarbeitung von Aktivitäten der einzelnen Teilnehmer und leitet Aufrufe an andere Komponenten weiter.

  Aktivitäten repräsentieren Aktionen, die der lokale Nutzer an seiner Saros-Instanz durchgeführt hat. Aktivitäten dienen dazu andere Teilnehmer über diese Aktionen zu informieren. Um die Dateien in einem konsistenten Zustand zu halten, werden in einer laufenden Sitzung Aktivitäten zwischen den einzelnen Teilnehmern hin und her gesendet.

  Die *Session Management* Komponente ist sowohl für die Verarbeitung lokaler Aktivitäten aus der *User Interface* Komponente, als auch für die Verarbeitung eingehender Aktivitäten von anderen Teilnehmern über die

Network Layer Komponente verantwortlich. Dafür greift das *Session Management* auf die *Shared Session Data* Komponente zu, in der die Informationen über geteilte Dateien gespeichert werden.

- **Concurrency Management:** Sowohl eingehende als auch ausgehende Aktivitäten werden vom *Session Management* zunächst durch die *Concurrency Management* geleitet. Diese Komponente ist dafür verantwortlich einen konsistenten Zustand geteilter Dateien zu gewährleisten. Dafür wird der Jupiter-Algorithmus verwendet, der anhand gemachter Änderungen der Zustand der einzelnen Dokumente speichert und in auszuführende Aktionen umwandelt für die einzelnen Teilnehmer umwandelt. Für eine genauere Beschreibung siehe Abschnitt 4.2.3.1.

- **User Interface:** Die Komponente *User Interface* beinhaltet alle sichtbaren Elemente, mit denen der Nutzer interagieren kann und die von Saros bereitgestellt werden, sowie Eclipse selbst. Diese Komponente stellt also Aktionen des Nutzers und die Datenbasis für das Sarossystem dar. Als Verbindungsstück zwischen der *User Interface* und dem *Session Management* dienen die sogenannten *Activity Provider*. Diese sind dafür zuständig Aktionen des lokalen Nutzers in Aktivitäten umzuwandeln und an das *Session Management* weiterzuleiten. Darüber hinaus ist ein *Activity Provider* auch dafür zuständig eingehende Aktivitäten auf die lokale Kopie der geteilter Projekte anzuwenden.

- **Network Layer:** Für die Kommunikation mit den anderen Teilnehmern ist der *Network Layer* zuständig. Dieser ist zum Beispiel dafür verantwortlich Aktivitäten an Teilnehmer zu senden und Aktivitäten von diesen zu empfangen.

- **Invitation:** Die *Invitation* Komponente ist dafür verantwortlich eine Sitzung zu starten, sowie Nutzer und Projekte zu einer laufenden Sitzung hinzuzufügen. Dazu kommuniziert die Komponente sowohl mit der *User Interface* als auch mit der *Network Layer* Komponente.

Dieses erste grobe Komponenten-Diagramm habe ich dann als Grundlage für das weitere Vorgehen verwendet. Dieses Komponenten-Diagramm enthielt noch einige Fehler, die aber im Laufe der Arbeit behoben wurden. Obwohl dieser grobe Überblick noch keine Szenarien oder Sichten berücksichtigte half er mir doch stark bei der weiteren Anfertigung dieser Arbeit.

4.2 Das erste Kernszenario

Nachdem ich das allgemeine Vorgehen definiert und eine erste Version der Architektur in Saros erstellt hatte, konnte ich nun mit der eigentlichen Analyse beginnen. Als erstes habe ich dabei eine Reihe von Kernszenarien aufgestellt, die wichtig für die Funktionalität in einer laufenden Sitzung sind. Als wichtigste Szenarien habe ich dabei die folgenden identifiziert:

1. **Editieren einer geteilten Datei:** Alice und Bob befinden sich in einer Sitzung und haben die Datei *a.txt* geteilt. Alice editiert die Datei indem sie ein *a* eingibt. Die Änderung wird bei Bob angewendet.

2. **Anlegen einer Datei in einem geteilten Projekt:** Alice und Bob befinden sich in einer Sitzung und haben das Projekt *A* geteilt. Alice fügt eine Datei *a.txt* hinzu. Bob erhält die Datei.

3. **Starten einer Sitzung:** Alice startet eine Sitzung mit Bob.

4. **Hinzufügen eines Nutzers zu einer laufenden Sitzung:** Alice und Bob befinden sich in einer Sitzung. Alice lädt Carl zu dieser Sitzung ein.

5. **Hinzufügen von Projekten zu einer Sitzung:** Alice und Bob befinden sich in einer Sitzung. Alice fügt das Projekt *A* zur laufenden Sitzung hinzu. Bob erhält das Projekt.

4.2.1 Schritt 1: Kernszenario auswählen

Nach reiflicher Überlegung fiel meine Wahl auf Szenario 2: **Anlegen einer Datei in einem geteilten Projekt**. Der Grund dafür war, dass es von den in Frage kommenden Kernszenarien als das am wenigsten komplexe erschien. Damit schien es sich am Besten für den ersten Durchlauf zu einigen, da ich anhand dieses Szenarios das Vorgehen testen und weiter verfeinern können würde. Szenario 1 **Editieren einer geteilten Datei** kommt zwar in einer laufenden Sitzung deutlich häufiger vor, allerdings sind hierbei zusätzliche Synchronisationsschritte durch das *Concurrency Management* notwendig, um gleichzeitiges Editieren zu ermöglichen. Ein kurzer Blick in das entsprechende Paket zeigte, dass an dieser Synchronisation mehrere umfangreiche Klassen beteiligt waren. Da ich noch nicht abschätzen konnte, wie kompliziert der dabei verwendete Algorithmus war, entschied ich mich gegen dieses Szenario.

4.2.2 Schritt 2: Komponenten und Abläufe herausarbeiten

Nachdem ich das Szenario ausgewählt hatte, mussten ich die Komponenten herausarbeiteten, die an der Bearbeitung dieses Szenarios beteiligt sind. Dazu habe ich zunächst einmal geschaut, welche Komponenten, gemäß deren Aufgabendefinition aus der ersten Version der Architekturbeschreibung, bei diesem Szenario durchlaufen werden müssten.
Dabei habe ich festgestellt, dass die wichtigste Komponente die *Session Management*-Komponente ist, da diese für die Bearbeitung von Aktivitäten zuständig ist. Darüber hinaus habe ich die Komponente *Invitation* für dieses Szenario ignoriert, da sie in der laufenden Sitzung keine Rolle hat.
Nachdem ich die wichtigsten Komponenten definiert hatte, musste ich nun den Ablauf für dieses Szenario nachvollziehen und diese Überlegungen anschließend anhand des Quellcodes von Saros validieren. Ich wusste zu diesem Zeitpunkt, dass die Verarbeitung dieses Szenarios über Aktivitäten geschehen musste, also habe ich zunächst einmal versucht zu verstehen, wie diese Aktivitäten entstehen. Die Suche führt mich zum *Activity Provider SharedResourcesManager*.

Der *SharedResourcesManager* ist ein *IResourceChangeListener*. *IResourceChangeListener* ist ein Eclipse-Interface, welches die Methode *resourceChanged()* definiert. Diese wird verwendet um über Veränderungen im *Project Browser* von Eclipse informiert zu werden. Der *Project Browser* zeigt alle Projekte im Arbeitsbereich an. Eine implementierende Klasse wird über ihre *resourceChanged()*-Methode von Eclipse informiert, wenn zum Beispiel eine neue Datei angelegt oder gelöscht wird. Dazu muss diese Klasse in Eclipse registriert werden. In diesem konkreten Szenario wird der *SharedResourcesManager* darüber informiert, dass eine neue Datei angelegt wurde. Diese Information kommt an der *resourceChanged()*-Methode an, wo diese Änderung ausgewertet wird. Anschließend wird eine neue Aktivität erstellt und über die *fireActivity()*-Methode, die jeder *Activity Provider* besitzt, an das *Session Management* weitergeleitet.
Beim weiteren Nachvollziehen dieser Verarbeitung half das Tool JTourBus. Zum Zeitpunkt dieser Arbeit gab es sechs Touren in Saros. Zwei davon befassten sich mit der Verarbeitung von Aktivitäten. Auch die Website machte einige Angaben über den Umgang mit Aktivitäten, allerdings waren die Konzepte dort nicht anschaulich genug dargestellt und einige der Informationen zur Aktivitätsverarbeitung waren bereits als veraltet markiert worden.
Mit Hilfe der Informationen aus den JTourBus-Touren konnte ein aktuelles Aktivitätsdiagramm erstellt werden, welches die Bearbeitung dieses Szenarios genauer beschreibt. (siehe dazu auch Abbildung 4). Die Beschreibung dieses Szenarios basiert auf meinem damaligen Stand von Mai 2013 und beinhaltet sowohl das Erstellen der Datei, als auch das Empfangen der Datei auf dem anderen Knoten.

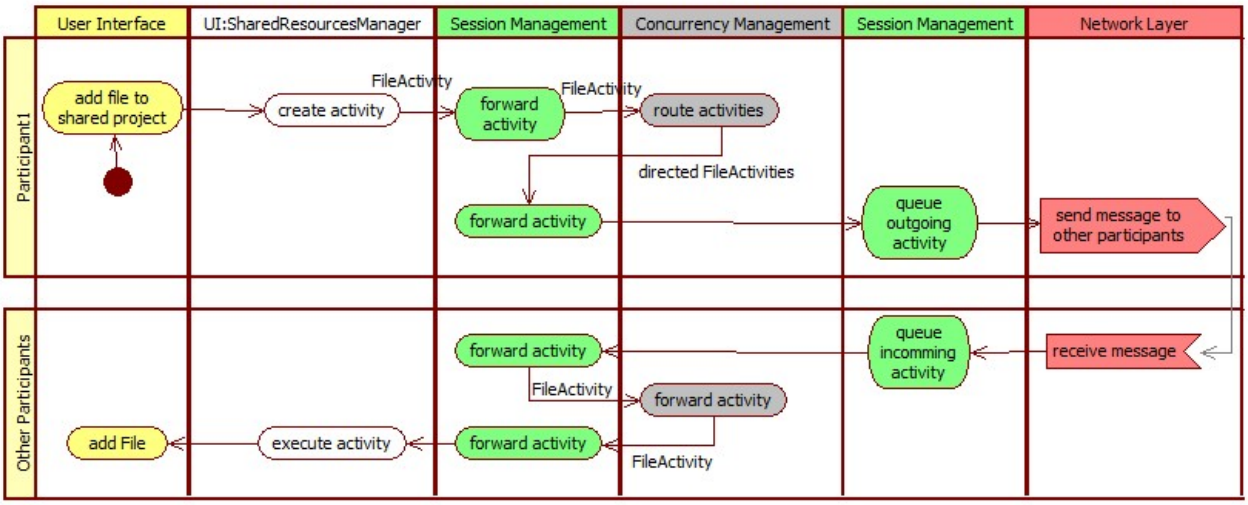


Abbildung 4: Aktivitätsdiagramm **Anlegen einer Datei in einem geteilten Projekt**; Das Präfix UI: steht für die Komponente *User Interface*; Stand: Mai 2013

- **User Interface:** Zunächst fügt der Nutzer eine neue Datei zum geteil-

ten Projekt in Eclipse selbst hinzu. Daraufhin wird der *Activity Provider SharedResourcesManager* aufgerufen.

- **SharedResourcesManager:** Der *SharedResourcesManager* erzeugt eine entsprechende Aktivität. In diesem Fall eine *File*-Aktivität. *File*-Aktivitäten sind Aktivitäten, die Änderungen auf Dateiebene anzeigen. Dies beinhaltet Löschen, Verschieben und eben Anlegen neuer Dateien. Diese *File*-Aktivität wird dann an das *Session Management* weitergeleitet.

- **Session Management (ausgehend):** Das *Session Management* leitet die Aktivität zunächst durch das **Concurrency Management**. Innerhalb dieser Komponente werden der Aktivität die korrekten Empfänger zugeordnet. Im Fall der *File*-Aktivität sind das alle anderen Teilnehmer.

  Anschließend werden die ausgehenden Aktivitäten in einer Warteschlange abgelegt.

- **Network Layer:** Der *Network Layer* holt sich die Aktivität und sendet sie an die jeweiligen Empfänger.

  Auf Seiten des Empfängers kommt die Aktivität an dessen *Network Layer* an, von wo aus sie wieder in einer Warteschlange im *Session Management* abgelegt wird.

- **Session Management (eingehend):** Das Session Management fragt diese Warteschlange periodisch ab und verarbeitet darin befindliche Aktivitäten. Dabei wird die Aktivität auch wieder durch das *Concurrency Management* geschickt, das allerdings in diesem Fall feststellt, dass nichts zu tun ist und die Aktivität einfach zurückgibt.

- **SharedResourcesManager:** Schlussendlich kommt die Aktivität dann am *SharedResourcesManager* des Empfängers an. Dieser verwendet Funktionen der Eclipse-API um eine lokale Kopie der geteilten Datei anzulegen.

4.2.3 Schritt 3: Szenarien verallgemeinern

Nachdem mir der Ablauf für dieses Kernszenario klar war, habe ich überprüft, ob sich dieser Ablauf auch auf weitere Szenarien anwenden lässt. Dabei fiel mir auf, dass an den meisten Stellen im Quellcode keine konkreten *FileActivities* übergeben wurden, sondern meistens nur das Interface *IActivity* verwendet wurde, das alle Aktivitätstypen implementieren. Der Schluss lag also nah, dass die Behandlung von Aktivitäten gut abstrahiert ist und andere Aktivitätentypen ähnlich behandelt werden. Auf diese Weise ließe sich eine große Anzahl von Szenarien auf Varianten des Kernszenarios

6. **Verarbeiten von Aktivitäten:** Alice führt eine beliebige Aktion aus, die eine Aktivität erzeugt. Diese Aktivität wird an Bob gesendet, der sie bei sich anwendet.

zusammenfassen.
Zu diesem Zweck habe ich zunächst einmal eine Liste aller Aktivitätentypen aufgestellt und anschließend geschaut, ob deren Bearbeitung zu einem gemeinsamen Szenario zusammengefasst werden kann.
Bei näherer Betrachtung der verschiedenen Aktivitätentypen fiel zunächst auf, dass alle Typen, bis auf Ausnahmen die später betrachtet werden, dem gleichen Muster folgten. (Siehe dazu auch Abbildung 5).

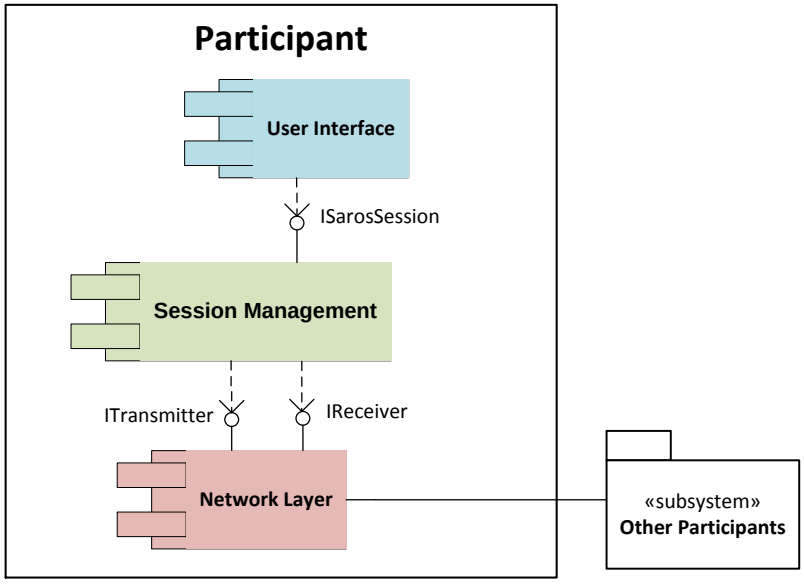


Abbildung 5: Komponenten des Szenarios **Verabeitung von Aktivitäten**

Szenario: Verarbeiten von Aktivitäten (Gesamtbild)

- **User Interface:** Die Erzeugung von Aktivitäten wird zunächst durch Aktionen des Nutzers im *User interface* angestoßen. Dort wird die eigentliche Aktivität von einem *Activity Provider* erzeugt. Diese sind Teil der *User Interface* Komponente und repräsentieren den lokalen Nutzer aus Sicht des Systems. Der *SharedResourcesManager* ist ein solcher *Activity Provider*, der für *File*- und *Folder*-Aktivität zuständig ist. Für andere Aktivitätentypen gibt es dementsprechend auch andere *Activity Provider*.

  Activity Provider müssen sich über das *ISarosSession*-Interface an der *Session Management* Komponente registrieren. Auf diese Wiese können sie erzeugte Aktivitäten weitergeben und werden über eingehende Aktivitäten informiert.

- **Session Management (senden):** Das *Session Management* verarbeitet die Aktivitäten und ordnet ihnen den korrekten Empfänger zu.

Diese Zuordnungen werden dann der *Network Layer*-Komponente über das Interface *ITransmitter* übergeben.(siehe Seite 26 für detailliertere Informationen zur *Session Management* Komponente.

- **Network Layer:** Der *ITransmitter* sendet die Aktivität dann an die anderen Teilnehmer. Beim Empfänger wird das *Session Management* dann vom *IReceiver* darüber informiert, dass eine neue Aktivität eingetroffen ist.

- **Session Management (empfangen):** Das *Session Managment* verarbeitet die Aktivität und gibt sie schlussendlich an das *User Interface* des Empfängers ankommen. Dort werden die Aktivitäten dann vom zuständigen *Activity Provider* auf das Projekt angewendet. Um über eingehende Aktivitäten informiert zu werden registriert das *Session Management* einen *Packet Listener* am *Network Layer*.

Die *Shared Session Data* und die *Concurrency Control* wurden hierbei in der Beschreibung in die Komponente *Session Management* integriert, da sie nur von dieser aufgerufen werden können und sollen.
Das untersuchte Szenario ließ sich also auf das Szenario **Verarbeitung von Aktivitäten** zusammenfassen. Dies schließt auch das zu Beginn genannte Szenario **Editieren einer geteilten Datei** mit ein. Da sich allerdings der Großteil der Verarbeitung in der Komponente *Session Management* abspielt habe ich beschlossen diese nochmal in Unterkomponenten aufzuteilen und die Aufgaben genauer herauszuarbeiten. Das Ergebnis wird auch in Abbildung 6 dargestellt.
Szenario: Verarbeiten von Aktivitäten (Details Session Management)

- **Activitiy Provider/Registry:** Wenn ein *Activity Provider* eine neue Aktivität erzeugt informiert er das *Activity Handling* darüber.

  Wenn eine eingehende Aktivität ausgeführt werden soll, wird die *Registry* darüber informiert. In diesem Fall werden alle registrierten *Activity Provider* über diese Aktivität informiert. Die *Registry* weiß also nur welche *Activity Provider* registriert sind, aber nicht welcher für diese spezielle Aktivität zuständig ist. Ein *Activity Provider* müssen also selbst wissen, ob die Aktivität für ihn bestimmt ist oder nicht.

- **Activity Handling:** Das *Activity Handling* ist die zentrale Komponente für die Verarbeitung von Aktivitäten und ist dafür verantwortlich den Aktivitäten Empfänger zuzuordnen. Die Empfänger werden anhand des Aktivitätstypen und möglicherweise in der Aktivität gesetzter Ziele ermittelt. Diese Aktivitäten werden dann mitsamt den Empfängern an den *Activity Sequencer* weitergegeben.

- **Activity Sequencer:** Der *Activity Sequencer* ist dafür zuständig ausgehende Aktivitäten zu puffern und ihnen eine Sequenznummer zuzuordnen, damit die Empfänger sicherstellen können, dass keine Aktivitäten verloren gegangen ist.

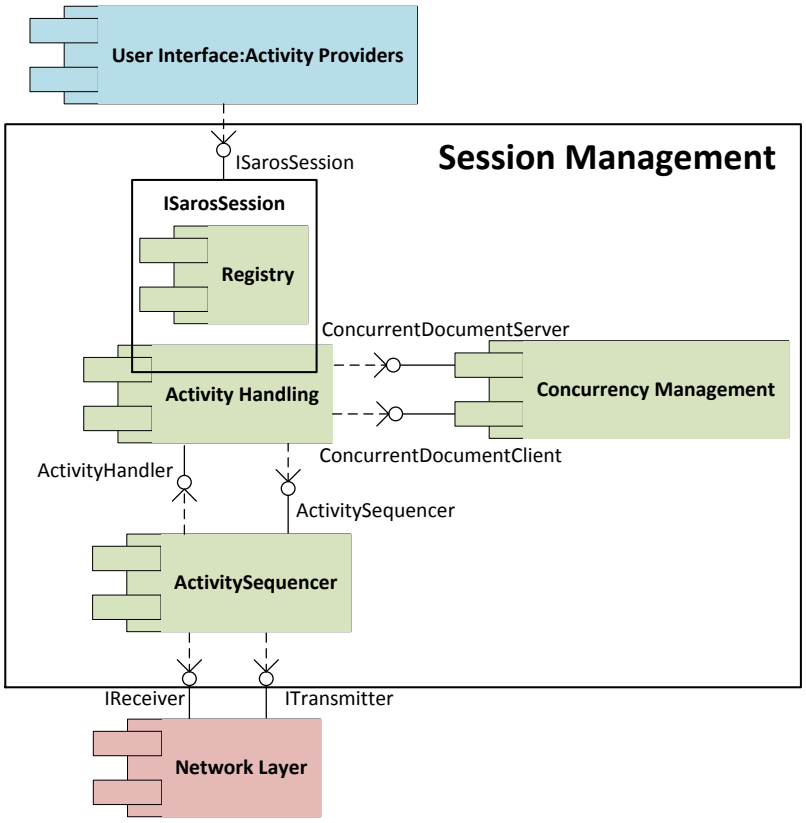


Abbildung 6: Komponente Session Management

Der *Activity Sequencer* ist außerdem dafür verantwortlich ausgehende Aktivitäten für das Senden zu serialisieren und eingehende zu deserialisieren. Für diese Vorgang werden serialisierbare Repräsentationen von Aktivitäten verwendet. Die Aktivitätsdatenobjekt (im Folgenden als ADO bezeichnet).

> Anforderung I: Bei diesen Umwandlungen Aktivität → ADO → Aktivität dürfen die in den Aktivitäten gespeicherten Daten nicht verändert werden oder verloren gehen.

- **Concurrency Management:** Einige Aktivitäten müssen zunächst transformiert werden. "Transformation" bedeutet hierbei, den lokalen an einen globalen Stand anzupassen oder der Aktivität einen Zeitstempel zuzuordnen. Für die "Transformation" dieser Aktivitäten ist das *Concurrency Management* zuständig. Dazu wird der Jupiter-Algorithmus verwendet.

4.2.3.1 Der Jupiter-Algorithmus Der Jupiter-Algorithmus besteht aus einer Serverkomponente, sowie einer Clientkomponente für jeden Teilnehmer. Der Server führt für jede in der Sitzung geöffnete Datei und jeden Nutzer eine Jupiter-Ressource. Jede Jupiter-Ressource besitzt einen Zustand in Form eines Jupiter-Zeitstempels. Anhand dieses Zeitstempels kann bestimmt werden, wie viele Änderungen der Nutzer durchgeführt hat (local) und wie viele Änderungen er erhalten hat (remote). Ein Beispiel für einen solchen Zeitstempel wäre [2,5] auf der Serverseite. Dies würde bedeuten, dass der Server einem Client B bisher zwei Jupiter-Aktivitäten geschickt hat und von ihm fünf Jupiter-Aktivitäten zum Verteilen erhalten hat.
Auch jeder Client führen eine Jupiter-Ressource für jede Datei. Diese müssen den selben Zeitstempel besitzen wie auf dem Server, damit die Transformationen korrekt durchgeführt werden können. Wenn eine neue Jupiter-Aktivität für eine Datei generiert wird, wird der *local*-Zeitstempel erhöht und wenn eine Jupiter-Aktivität eingeht der *remote*-Zeitstempel. Auf diese Weise stimmt der *local*-Zeitstempel eines Clients immer mit dem entsprechenden *remote*-Zeitstempel des Servers überein und umgekehrt. Wenn wir unser Beispiel von eben betrachten wäre dieser Zeitstempel also [5,2].
Das *Concurrency Management* wandelt ausgehende *textEdit*-Aktivitäten [6] in Jupiter-Aktivitäten um und eingehende Jupiter-Aktivitäten wieder zurück in *textEdit*-Aktivitäten. Für beide Operationen ist ein konsistenter Zeitstempel Voraussetzung.

4.2.4 Schritt 4: Nebenläufigkeitseigenschaften analysieren

Als nächstes habe ich die Nebenläufigkeitseigenschaften des untersuchten Szenarios analysiert. Diese Analyse habe ich in zwei Teile aufgeteilt. Zunächst habe ich die lokale Nebenläufigkeit für die Verarbeitung von Aktivitäten auf einem Knoten analysiert. Anschließend habe ich dann im zweiten Schritt die Verteilung über alle Knoten betrachtet.

4.2.4.1 Analyse lokaler Nebenläufigkeiten Um die Zusammenhänge der verwendeten *Threads* bei der Verarbeitung von Aktivitäten besser verstehen zu können habe ich das bereits erstellte Aktivitätsdiagramm zu Rate gezogen. Anhand dieses Diagramms habe ich den Verlauf noch einmal nachvollzogen und Stellen notiert, an denen die Ausführung des Programms an einen anderen *Thread* übergeben wird. Siehe dazu auch die Abbildungen 7 für das Senden und 8 für das Empfangen von Aktivitäten.
Eine Reihe verschiedener Threads spielen während der Verarbeitung von Aktivitäten eine Rolle. Zur Synchronisation dieser Threads werden außerdem eine Reihe von Warteschlangen nach den Prinzipien des Produzent-Konsument-Musters (siehe auch Abschnitt 2.1.3) verwendet. Die verschiedenen Threads führen also nicht parallel gleichartige Aktionen aus, sondern es gibt bestimmte Threads mit klar definierten Aufgaben und Übergabepunkte.

[6] Diese repräsentieren eine Eingabe eines Nutzers

Ausgehende Aktivitäten

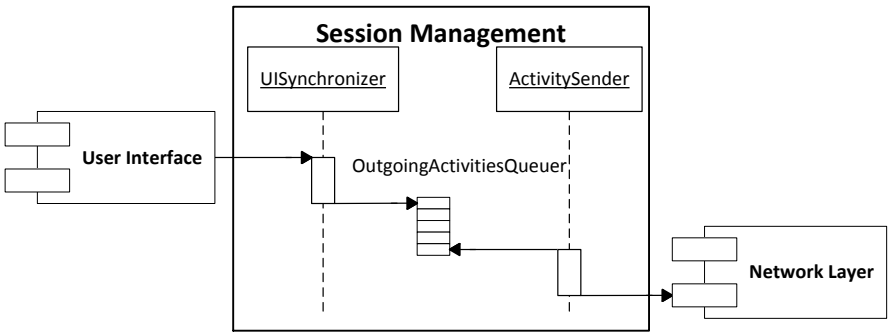


Abbildung 7: Threads und Warteschlangen während der Verarbeitung ausgehender Aktivitäten. Pfeile bedeuten synchrone Aufrufe.

- **Eingangsthread:** Welcher Thread für die Erzeugung einer neuen Aktivität verantwortlich ist lässt sich nicht allgemein beantworten. Die meisten Aktivitäten werden zwar auf dem *Main*-Thread[7] von Eclipse erzeugt, doch es gibt auch andere Stellen, an denen dies zum Beispiel von Eclipse-internen Threads mit speziellen Aufgaben durchgeführt wird. Der Eingangsthread ruft den *UISynchronizer*, synchron auf, bevor Aktivitäten transformiert werden.[8]

- **UISynchronizer:** Die Transformation von Aktivitäten wird auf dem *UISynchronizer*-Thread durchgeführt. Dieser Thread stellt eine Abstraktion des *Main*-Threads dar. Die Verwendung dieses Threads hat einen wichtigen Grund: Mit Hilfe des *UISynchronizers* wird verhindert, dass ungewollte Operationen während der Transformation von Aktivitäten stattfinden. Lokale Änderungen während der Transformation von beispielsweise Jupiter-Aktivitäten würden dazu führen, dass diese Transaktionen auf einen falschen Zustand angewandt werden. Aus diesem Grund soll es während der Transformation keine lokalen Änderungen geben. Da wir Eclipse nicht direkt sperren können, bedienen wir uns hier eines Tricks: Indem die Transformation auf dem *Main*-Thread ausgeführt wird, sorgen wir dafür, dass der lokale Nutzer währenddessen keine Änderungen vornehmen kann, da seine Arbeitsfläche gesperrt ist. (Siehe Abschnitt 2.1.2) Diese Sperre ist so kurz, dass der Nutzer davon nichts mitbekommt, aber reicht aus um Versagen durch falsch eingefügte Zeichen zu vermeiden.

[7] Der *Main*-Thread ist der Thread, der dafür verantwortlich ist Nutzereingaben entgegenzunehmen.

[8] In den meisten Fällen wird hierbei der Thread nicht gewechselt, da auch der *UISynchronizer* eine Abstraktion des *Main*-Threads darstellt.

- **OutgoingActivitiesQueuer:** Nach der Transformation legt der *UISynchronizer* die Aktivität im *OutgoingActivitiesQueuer* ab, um die Arbeitsfläche wieder freizugeben. Dies ist eine Warteschlange, die nach dem Produzent-Konsument-Muster funktioniert (siehe 2.1.3). Produzent ist hierbei der *UISynchronizer* und Konsument der *ActivitySender*.
- **ActivitySender:** Der *ActivitySender-Thread* holt sich periodisch Aktivitäten aus dem *OutgoingActivitiesQueuer* und sendet sie über die *Network Layer* Komponente an die anderen Teilnehmer. Auf diese Art sind Erzeugen und Senden von Aktivitäten voneinander getrennt und müssen nicht auf die Beendigung des anderen Prozesses warten um weiterarbeiten zu können.

Eingehende Aktivitäten

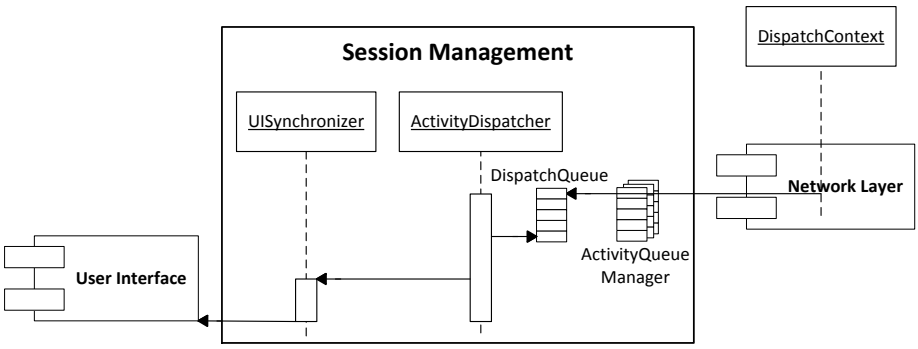


Abbildung 8: Threads und Warteschlangen während der Verarbeitung eingehender Aktivitäten. Pfeile bedeuten synchrone Aufrufe. Stand Juni 2013.

- **DispatchContext:** Der *DispatchContext*-Thread ist für eingehende Aktivitäten verantwortlich. Alle Aktivitäten, die aus dem *Network Layer* kommen, werden zunächst in diesem Thread verarbeitet. Er legt eingehende Aktivitäten in der *dispatchQueue* ab. (Der in der Abbildung dargestellte *ActivityQueueManager* wird später behandelt.)
- **dispatchQueue:** Diese Warteschlange wird verwendet um das Empfangen und Verarbeiten von Aktivitäten zu trennen. Auch hier wird wieder nach dem Produzent-Konsument-Muster vorgegangen. Sie bildet sozusagen das Gegenstück zum *OutgoingActivitiesQueuer* für eingehende Aktivitäten.
- **ActivityDispatcher:** Der *ActivityDispatcher*-Thread ist dafür verantwortlich Aktivitäten aus der *dispatchQueue* zu nehmen und an den *UI-*

Synchronizer zu übergeben. Dieser Thread hat die Aufgabe darauf zu warten, dass der *Main*-Thread die Aktivität bearbeiten kann. Dadurch wird der *DispatchContext* nicht am Empfangen von Aktivitäten gehindert.

- **UISynchronizer:** siehe eingehende Aktivitäten.
- **activityQueueManager:** Schlussendlich gab es noch eine weitere Warteschlange, den *activityQueueManager*. Dieses Objekt verwaltete eine Warteschlange für jeden anderen Teilnehmer. Diese sind dafür verantwortlich die Zeitstempel zu überprüfen. Eingehende Nachrichten wurden in die Warteschlange gelegt, wenn der Zeitstempel nicht stimmte und erst dann wieder abgeholt, wenn die fehlende Nachricht ankam oder eine gewisse Zeit verstrichen war. Diese Warteschlange wurde von einem Thread befüllt und geleert. Die Aufgabe dieser Warteschlange war somit nicht die Trennung von Aufgaben, sondern sie stellte einen Ort dar, an dem Nachrichten abgelegt werden konnten, bis die korrekte Nachricht eingetroffen ist.

4.2.4.2 Verteilungsaspekte Als Nächstes habe ich die Verteilungsaspekten in einer laufenden Sitzung untersucht. Um einen konsistenten Zustand innerhalb einer Sitzung zu gewährleisten, sollte eine globale Reihenfolgen von Aktivitäten hergestellt werden.
Dabei fiel mir zunächst auf, dass es je nach Typ unterschiedliche Wege gab, auf denen Aktivitäten verschickt wurden. Die meisten Aktivitätstypen wurde dabei einfach direkt an alle oder auch einzelne Teilnehmer geschickt. Dieses Verhalten ließ darauf schließen, dass Saros eine Peer-To-Peer (P2P) [9] Netzwerkarchitektur einsetzt.
Bei genauerem Hinsehen fiel mir allerdings auf, dass es eine Reihe von Sonderfällen gab. Bei diesen Sonderfällen wurden die Aktivitäten zunächst an eine zentrale Stelle geschickt und von dort aus an die anderen Teilnehmer weitergeleitet. Das beste Beispiel für dieses Verhalten sind Jupiter-Aktivitäten.
Jupiter-Aktivitäten werden nicht direkt an die einzelnen Knoten gesendet, sondern sie gehen zunächst an den Jupiter-Server. Dieser Server befindet sich auf dem Knoten des Hosts. Der Server transformiert die Jupiter-Aktivität für die einzelnen Teilnehmer und sendet ihnen die richtige Jupiter-Aktivität zu. Durch diesen zentralen Jupiter-Server kann sichergestellt werden, dass die Aktivitäten immer in der richtigen Reihenfolge angewandt werden und die lokalen Kopien der einzelnen Teilnehmer somit konsistent bleiben, auch wenn verschiedene Personen an der gleichen Datei arbeiten (siehe auch Abschnitt 4.2.3.1).

4.2.5 Schritt 5: Auswertung

Nachdem ich das Szenario **Verarbeitung von Aktivitäten** in den Schritte 2 bis 4 genau untersucht hatte musste ich nun in Schritt 5 die Analyse noch

[9] Peer-To-Peer (P2P) bedeutet, dass jeder Netzwerkknoten, hier die einzelnen Teilnehmer, eine direkte Verbindung mit allen anderen Netzwerkknoten hat und seine Nachrichten direkt an diese Knoten senden kann

auswerten. Dabei habe ich Probleme niedergeschrieben und bewertet, die mir während der Analyse aufgefallen sind. Diese ließen sich in die Kategorien

- **Architektur**
- **Lokale Nebenläufigkeit**
- **Verteilung**

einteilen, die mit den Abschnitten der Analyse des Szenarios **Verarbeitung von Aktivitäten** korrespondieren.[10]

4.2.5.1 Architekturverletzungen In den Schritte 2 und 3 habe ich die Komponenten des Systems, sowie deren Aufgaben und Abläufe genau definiert. Das grundlegende Verfahren, das verwendet wurde, schien sinnvoll und die Aufgaben und Grenzen der einzelnen Komponenten ließen sich gut definieren. Allerdings sind mir dabei auch eine Reihe von Probleme aufgefallen, gerade wenn es um die Details der Implementierung ging.
Das Erste, was mir aufgefallen ist, war, dass sämtliche Aktivitäten, egal welchen Typ sie haben, durch die *Concurrency Management* Komponente geleitet wurden. Darüber hinaus wurden auch die Empfänger für Aktivitäten innerhalb dieser Komponente bestimmt. Wenn man die Aufgaben dieser Komponente betrachtet, sollte sie aber nur für die Transformation von Aktivitäten und dem damit verbundenen Einhalten eines konsistenten Zustands der geteilten Dateien, zuständig sein. Für die Verteilung der Aktivitäten ist laut Definition das *Activity Handling* zuständig.

> Problem [A1CM]: Das *Concurrency Management* bestimmt Ziele für Aktivitäten. Dabei wird die Trennung von Aufgaben verletzt, da diese Komponente Aufgaben des *Activitiy Handlings* übernimmt.

Außerdem fiel mir auf, dass es eine Reihen von Sonderbehandlungen im *Concurrency Management* für Nutzer gab, die nur lesenden Zugriff in einer Sitzung hatten. Nutzern, denen die Schreibrechte entzogen wurden, wurden aus dem Jupiter-Algorithmus entfernt und erst wieder hinzugefügt, wenn sie wieder schreiben durften. Dadurch mussten an vielen Stellen im Code Sonderbehandlungen durchgeführt werden.
Der Grund für das Entfernen vom Jupiter-Algorithmus war, dass dadurch Umwandlungen gespart wurden. Da wir allerdings auch problemlos Sitzungen verwalten können, bei denen alle Nutzer Schreibrechte haben, schien dies eine unnötige Verbesserung' zu sein, die in deutlich undurchsichtigerem Code resultierte.

[10] Anmerkung: Probleme werden in der folgenden Form benannt: [<Kategorie><Szenario><Abkürzung>]. Beispiel: Kategorie: Architektur; Szenario: 1; Problem: Concurrency Management; wäre [A1CM]. Probleme, die keinem Szenario zugeordnet werden können, werden mit 3 gekennzeichnet und besitzen keine Kategorie.

> Problem [A1RO]: Sonderbehandlung von Read-Only-Usern resultiert in undurchsichtigem Code.

Das Nächste was mir auffiel war, dass der in Schritt 3 beschriebe Ablauf für Aktivitäten nicht für alle Aktivitätstypen korrekt umgesetzt wurde. So zum Beispiel, bei der *PermissionActivity*, die für das Ändern von Rechten eines Nutzers zuständig ist. Bei der Erzeugung dieser Aktivität wurde kein *Activity Provider* verwendet. Stattdessen wurde die Aktivität direkt im *Activity Handling* erzeugt. Es gab noch weitere Beispiele für Aktivitätstypen, die an der falschen Stelle erzeugt wurden und teilweise sogar einen Großteil des *Activity Handlings* komplett umgingen.

> Problem [A1ArBr]: Es gibt Ausnahmen, bei denen sich der Ablauf der Aktivitäten-Verarbeitung vom allgemeinen Ablauf unterscheidet. Dabei handelt es sich um Architekturbrüche.

Darüber hinaus sollte auch die Tatsache, dass *Activity Provider* sowohl für das Erzeugen, als auch das Verarbeiten von Aktivitäten zuständig sind zumindest angemerkt werden. Ein *Activity Provider* dient somit also sowohl als Produzent, als auch als Konsument.

> Problem [A1ActPr]: *Activity Provider* sind sowohl Produzenten, als auch Konsumenten während der Aktivitätsverarbeitung. Sie erfüllen also eigentlich zu viele, eigentlich gut abgrenzbare Aufgaben.

Außerdem ließen sich zwar die Grenzen der einzelnen Komponenten gut finden, doch die Paketstruktur spiegelte diese Eingrenzung nicht immer gut wieder. So befanden sich manche Klassen in den falschen Paketen. Ein Beispiel dafür ist der *ActivitySequencer*, die Hauptklasse der Komponente *Activity Sequencer*. Diese befand sich im Paket *net*, welche Klassen des *Netzwerk Layers* beinhaltet. Nach meiner Analyse müsste es sich allerdings im Paket für das *Session Management* befinden. Dieses Paket heißt aktuell aus historischen Gründen *project*, was auch nicht dem Namen der Komponente entspricht.

> Problem [A1PaKo]: Es gibt Diskrepanzen zwischen Paketstruktur im Projekt und der Komponenten-Zugehörigkeit von Klassen, sowie der Benennung von Paketen und den Namen von Komponenten. Der *Development View* ist steht also im Widerspruch zum *Logical View*.

Außerdem ist die Einteilung in Komponenten in Saros nicht vollständig umgesetzt worden, sodass es an manchen Stellen auch weitere Abhängigkeiten zwischen den einzelnen Komponenten gibt. Die Klasse *SarosSession* wird zum Beispiel von einer großen Anzahl an Komponenten verwendet, auch wenn diese eigentlich gar keinen Zugriff darauf haben sollten.

Problem [A1Komp]: Die Komponentendarstellung ist idealisiert und entspricht nicht komplett der Realität.

Darüber hinaus konnten gerade im *Session Management* eine Reihe von zirkulären Abhängigkeiten gefunden werden. Diese entstanden, da die einzelnen Unterkomponenten sowohl eingehende, als auch ausgehende Aktivitäten verarbeiten und für eingehende Aktivitäten der Weg *Klasse A* → *Klasse B* verwendet wird und für ausgehende *Klasse B* → *Klasse A*.

Problem [A1Zirk]: Zirkuläre Abhängigkeiten im *Session Management*

4.2.5.2 Lokale Nebenläufigkeit Bei der Untersuchung der lokalen Nebenläufigkeit kam ich zu dem Ergebnis, dass die hier verwendet Threads größtenteils sinnvoll sind und klar definierte Aufgaben haben.
Aufgefallen sind mir dabei allerdings die uneinheitlichen oder schwer auseinanderzuhaltenden Benennungen der einzelnen Threads und Warteschlangen. Beispielsweise heißt die Warteschlange für eingehende Aktivitäten **dispatchQueue:** das Gegenstück dazu für eingehende Aktivitäten aber *OutgoingActivitiesQueuer*.

Problem [L1ThWa]: Threads und Warteschlangen sind schwer auseinanderzuhaltend und nicht einheitlich benannt.

Darüber hinaus gab es noch eine Warteschlange, die merkwürdig erschien: Den *activityQueueManager*. Die Aufgabe dieser Warteschlange war es sicherzustellen, dass keine Nachrichten verloren gehen. Dazu hatte jeder Nutzer eine Warteschlange für jeden anderen Nutzer. Auf diese Weise konnte zwar sichergestellt werden, dass Nachrichten von bestimmten Nutzern nicht verloren gehen, doch eine globale Reihenfolge lässt sich damit nicht herstellen (siehe Problem [V1CS]). Außerdem schien auch die Behandlung von Fehlern unbedacht, denn nach einer Wartezeit wurden nicht erhaltene Aktivitäten einfach ignoriert und es wurde weitergemacht, als wäre nichts passiert, was in inkonsistenten Sitzungen resultieren kann[11].

Problem [L1AQM]: Die Überprüfung und Behandlung von verlorenen Nachrichten durch den *ActivityQueueManager* ist nicht gut durchdacht und hilft nicht dabei eine globale Reihenfolge von Aktivitäten sicherzustellen.

[11] Dieses Problem gehört zwar eigentlich in die Kategorie Verteilung, da der *ActivityQueueManager* aber im Rahmen der Lokalen Nebenläufigkeit untersucht wurde wird er hier aufgeführt.

4.2.5.3 Verteilung Als Letztes habe ich mich mit der Verteilung beschäftigt. Hierbei fiel mir sehr schnell ein grundlegendes Problem auf.

> Problem [V1CS]: Saros verwendet keine einheitliche Netzwerkarchitektur. Es mangelt hierbei an einer Spezifikation und einer entsprechenden Umsetzung.

Saros verwendete eine Mischung aus *P2P*- und *Client-Server*-Architektur. Das Problem an diesem Umstand ist, dass es eine *P2P*-Architektur nicht erlaubt eine globale Reihenfolge von Aktivitäten herzustellen. Das sorgt dafür, dass die Aktivitäts-Reihenfolge durcheinander geraten kann.

> **Beispiel:** Teilnehmer Bob legt eine neue Datei an und editiert sie anschließend sofort. Das Anlegen der Datei erzeugt eine *File*-Aktivität. Diese wird nacheinander an alle anderen Teilnehmer geschickt. Das Editieren verursacht eine *Jupiter*-Aktivität. Diese wird nur an den Host Alice gesendet und von da aus an die anderen Teilnehmer weitergeleitet. Da es keine globale Reihenfolge gibt, kann es dabei sein, dass die Nachrichten in einer falschen Reihenfolge bei Teilnehmer Carl ankommen und somit das Editieren vor dem Erzeugen der Datei ausgeführt wird.

Eine globale Reihenfolge von Nachrichten einzuhalten ist nur dann auf möglich, wenn es eine zentrale Stelle gibt, die das Senden koordiniert. Aus diesem Grund wäre die saubere Lösung eine einheitliche *Client-Server*-Architektur umzusetzen (siehe dazu auch Abschnitt 4.4).

> Anforderung II: Die Kommunikation zwischen den Knoten muss nach dem Client-Server-Prinzip geschehen.

4.3 Kernszenario 2

Bei der Analyse des ersten Kernszenarios konnten ich eine Reihe von Problemen aufdecken. Da mir das verwendete Verfahren außerdem gute Ergebnisse und ein tiefgehendes Verständnis des untersuchten Systems geliefert hat, habe ich mich dazu entschieden auch das zweite Kernszenario nach dem selben Verfahren zu analysieren.

4.3.1 Schritt 1: Kernszenario auswählen

Als erstes habe ich dabei wieder das zu untersuchende Kernszenario ausgewählt. Nachdem ich mich im ersten Kernszenario mit den Abläufe innerhalb einer laufenden Sitzung beschäftigt hatte, wollte ich nun den Einladeprozess genauer betrachten. Das schien mir der nächste logische Schritt zu sein, da der Einladeprozess ein wichtiger Bestandteil einer Saros-Sitzung ist. Darüber hinaus

hatte Sebastian Starroske in seiner Masterarbeit [Sta13] einige Probleme mit dem Einladeprozess aufgedeckt, wodurch eine Untersuchung dieses Vorgangs eine noch größere Priorität bekam.
Als Einstieg habe ich das konkrete Kernszenario **Starten einer Sitzung** gewählt. (siehe Abschnitt 4.2).

4.3.2 Schritt 2: Komponenten und Abläufe herausarbeiten

Der Einladungsprozess in einer Sarossitzung ist in zwei Teilabläufe aufgeteilt.

- Die *Session Negotiation*
- Die *Project Negotiation*

Während der *Session Negotiation* werden neue Nutzer zu einer Sitzung hinzugefügt. Am Ende der *Session Negotiation* ist der Nutzer Teil der Sitzung und kann dadurch Funktionen von Saros benutzen, die unabhängig von den geteilten Ressourcen sind, wie zum Beispiel die den Multiuser-Chat, ein Chatroom, indem sich alle Teilnehmer der Sitzung befinden.
Nach der *Session Negotiation* beginnt die *Project Negotiation.* Dabei werden die in der Sitzung zu teilenden Projekte übertragen. Jeder Teilnehmer verwendet in einer Sitzung eine lokale Kopie der geteilten Ressourcen. Das Übertragen der Projekte ist also notwendig für die eigentliche Arbeit mit Saros.
Bei der Durchführung dieser *Negotiations* gibt es immer eine ausgehende *Negotiation* für den Host und eine eingehende für den Client. Diese beiden kommunizieren über den *Network Layer* miteinander. Zusätzlich gibt es auch noch den *SarosSessionManager*, der die einzelnen *Negotiations* erstellt und den gesamten Einladungsprozess überwacht.
Aus diesen Informationen ist die folgende Abbildung 9 entstanden.
Die einzelnen Komponenten haben hier ähnliche Aufgaben, wie in der zuvor untersuchten laufenden Sitzung, mit dem Unterschied, dass die Komponente *Invitation Management* die Rolle der *Session Management*-Komponente als zentrale Verarbeitungsstelle übernimmt. Die *Session Management*-Komponente wird nur darüber informiert, wenn neue Teilnehmer die Sitzung betreten haben oder Projekte hinzugefügt wurden.

4.3.3 Schritt 3: Szenarien verallgemeinern

Nachdem ich mir klar gemacht hatte, wie die einzelnen Abschnitte des Einladungsprozesses von statten gehen, konnte ich auch die Kernszenarien **Hinzufügen eines Nutzers zu einer laufenden Sitzung** und **Hinzufügen eines Projektes zu einer Sitzung** zum Szenario

7. **Verändern einer Sitzung:** Ein Nutzer oder Projekt wird zur Sitzung hinzugefügt oder entfernt.

zusammenfassen um ein größeres Spektrum abzudecken. Darüber hinaus gehört auch **Verlassen einer Sitzung** zu diesem Szenario dazu. Zunächst habe ich dieses letzte Szenario allerdings außen vor gelassen und mich mit dem eigentlichen Einladeprozess beschäftigt.

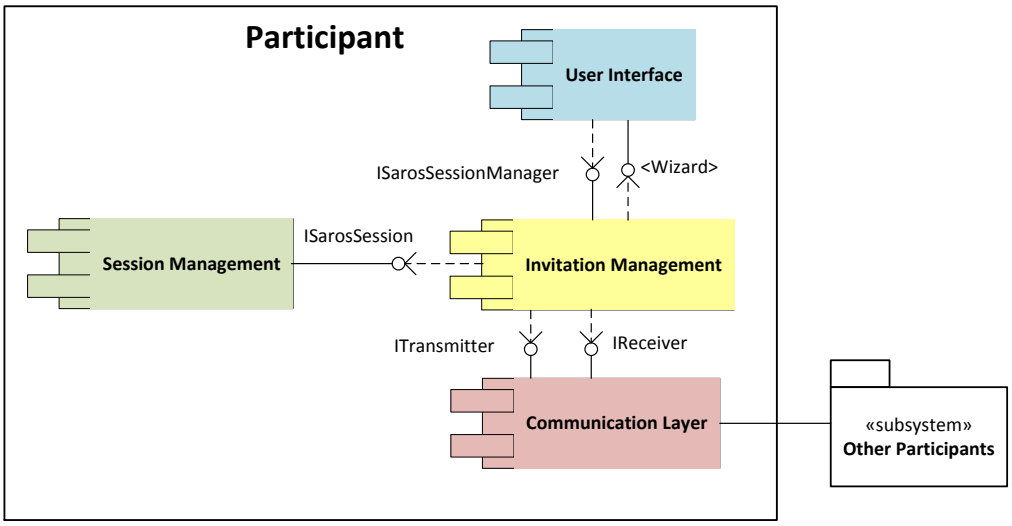


Abbildung 9: Komponenten während des Einladungsprozesses

4.3.4 Schritt 4: Nebenläufigkeitseigenschaften analysieren

Nachdem ich einen Überblick über die Komponenten erlangt haben, die beim Einladungsprozess eine Rolle spielen, habe ich in diesem Schritt die Nebenläufigkeitseigenschaften dieses Einladungsprozesses untersucht. Dazu wurden auch diesmal zunächst lokale Nebenläufigkeit und anschließend die Verteilungsaspekte untersucht.
Dabei stellte sich heraus, dass der Einladeprozess in einem eigenen Thread läuft. Der Host kann mehrere Nutzer gleichzeitig einladen und für jeden wird ein eigener Thread erstellt, sodass es möglich ist mehrere Einladungen an verschiedene Nutzer parallel zu bearbeiten, was sinnvoll erschien.
Bei der Untersuchung der Verteilungsaspekte hingegen gab es eine Besonderheit im Gegensatz zur laufenden Sitzung, die ich beachten musste. Während in einer laufenden Sitzung Aktivitäten verwendet wurden, wurden für die Änderung einer Sitzung größtenteils MESSAGES verwendet. MESSAGES sind genau wie Aktivitäten Nachrichten, die verwendet werden, um die Kommunikation zwischen Teilnehmern zu ermöglichen. Anders als Aktivitäten werden MESSAGES allerdings nicht weiter verarbeitet, sondern gehen direkt vom *Netzwork Layer* an den entsprechenden *Negotiation Process*.
Ein Grund dafür, für die Trennung von MESAGES und Aktivitäten ist, dass die Nachrichten zum Verändern der Sitzung klar von den Nachrichten der eigentlichen Sitzung unterschieden werden können. Außerdem gibt es zum Beispiel während der Einladung noch keine laufende Sitzung, die Aktivitäten verarbeiten kann. Daher habe ich die folgenden Regeln aufgestellt:

Anforderung III:

- **MESSAGES:** MESSAGES werden immer dann verwendet, wenn die Sitzung selbst verändert werden soll. Dies beinhaltet das Hinzufügen von Projekten, sowie das Hinzufügen und Entfernen neuer Nutzer.
- **Aktivitäten:** Aktivitäten sind Nachrichten mit einem speziellen Zweck. Sie werden in einer laufenden Sitzung verwendet und repräsentieren Aktionen, die ein lokaler Nutzer durchgeführt hat. Ein Beispiel dafür ist das Editieren einer geteilten Datei.

Nachdem ich die einzelnen MESSAGES, die während des Einladeprozesses verschickt werden, untersucht hatte, stellte ich fest, dass MESSAGES nur zwischen dem Host und jeweils einem Eingeladenen versendet werden. Der eigentliche Einladungsprozess zeigt also auf den ersten Blick keine Probleme.
Auf den zweiten Blick allerdings fand ich heraus, dass hier noch eine weitere Ebene zu beachten ist. Während ein neuer Nutzer hinzugefügt wird läuft für alle anderen Teilnehmer die eigentliche Sitzung parallel dazu weiter. Das bedeutet, dass weiterhin Aktivitäten an alle Teilnehmer gesendet werden.
Der eigentlich interessante Aspekt am Einladungsprozess ist also die Synchronisation zwischen laufender Sitzung und Einladungsprozess, da hier festgelegt werden muss, welche Aktivitäten der neue Nutzer wann erhalten muss, um eine konsistente Sitzung zu ermöglichen. Hier zeigt sich auch ein Vorteil an der Trennung von MESSAGES und Aktivitäten. Die für den Einladeprozess bestimmten MESSAGES können von den Aktivitäten aus der laufenden Sitzung unterschieden werden.

4.3.4.1 Aktivitäten während des Einladungsprozesses Nach der anfänglichen Analyse habe ich den Fokus nun auf die Synchronisation zwischen laufender Sitzung und Einladungsprozess gelegt, da es hier eine Reihe bekannten Versagen gab.
Zunächst habe ich die *Session Negotiation* untersucht. Dabei habe ich festgestellt, dass Nutzer erst am Ende dieser *Session Negotiation* zur Sitzung hinzugefügt werden und Aktivitäten erhalten. Hier gab es also keine Interaktion zwischen laufender Sitzung und Einladungsprozess. Im Rahmen dieser Analyse habe ich ein Sequenzdiagramm für das Versenden von MESSAGES erstellt. Dieses findet sich in der später erstellten Dokumentation im Anhang wieder. (Kapitel A).
Als nächstes habe ich das Versenden von Aktivitäten während der *Project Negotiation* untersucht. Auch hier wurde zunächst ein Sequenzdiagramm erstellt um nachvollziehen zu können, was während der *Project Negotiation* passiert. Zusätzlich habe ich untersucht, wie Aktivitäten vom neuen Nutzer behandelt werden.
Zunächst einmal stellte ich dabei fest, dass Nutzer, sobald sie die *Session Negotiation* beendet hatten von anderen Sitzungsteilnehmern als vollwertige Mit-

glieder der Sitzung betrachtet wurden. Das heißt, dass sie sämtliche Aktivitäten erhielten, die in der Sitzung versendet wurden. Dies schließt auch Aktivitäten ein, die sich auf Ressourcen beziehen, die der fragliche Nutzer noch gar nicht besitzt, da sie zu Beginn der *Project Negotiation* noch überhaupt nicht übertragen wurden.
Darüber hinaus stellte ich fest, dass es zwei Mechanismen gab die laufende Sitzung mit dem Einladeprozess zu synchronisieren.

- **Blockieren der Sitzung:** Der erste Mechanismus war das Blockieren einer laufenden Sitzung. Projekte, die geteilt werden sollen werden während der *Project Negotiation* zu einem Archiv zusammengepackt und dann verschickt. Sobald der Host damit begonnen hat das Archiv zu packen wurde die gesamte Sitzung blockiert. Dies sollte verhindern, dass die einzupackenden Ressourcen währenddessen verändert werden.

- **Zurückhalten von Aktivitäten:** Der zweite Mechanismus war das Zurückhalten von Aktivitäten auf Seiten des Eingeladenen. Zu Beginn der *Projekt Negotiation* wird der Eingeladene dazu aufgefordert einen Speicherort für die zu teilenden Ressourcen zu definieren. Sobald er diesen Dialog abgeschlossen hat, beginnt er damit Aktivitäten, die sich auf geteilte Ressourcen beziehen (Die sogenannten *Resource*-Aktivitäten[12]) in einer Warteschlange zu speichern. Diese Warteschlange wird erst geleert, sobald der Eingeladene alle Ressourcen erhalten hat. Auf diese Weise kann er die Aktivitäten dann auf die Ressource anwenden.

4.3.5 Verlassen einer Sitzung

Zu guter Letzt habe ich mir auch das Verlassen einer Sitzung angeschaut. Dieser Fall gehört zwar nicht zum Einladungsprozess dazu, verändert aber trotzdem die Sitzung an sich. Wenn ein Nutzer die Sitzung verlässt, wird bei ihm die stopSession()-Methode im *Session Management* aufgerufen.
Diese Methode informiert die anderen Teilnehmer der Sitzung darüber, dass der lokale Nutzer die Sitzung verlassen hat und beendet dann die Sitzung bei sich.

4.3.6 Schritt 5: Auswertung

Nachdem ich die Abläufe während des Einladeprozessen verstanden hatte, konnte ich nun damit beginnen Probleme zu spezifizieren, die es dabei gab. Auch hier habe ich die Probleme in eine Reihe von Kategorien aufgeteilt. Einige dieser Probleme waren bereits vor meiner Arbeit bekannt, allerdings fehlten teilweise die genauen Hintergründe.

4.3.6.1 Architektur In Abschnitt 4.3.4 habe ich eine Trennung von Messages und Aktivitäten definiert. Bei der Betrachtung der *Project Negotiation* fiel auf, dass in diesem Ablauf Aktivitäten verwendet wurden, obwohl es sich

[12] Diese werden als *Resource*-Aktivitäten bezeichnet und sind keine eigenen Aktivitätstypen, sondern stellen ein Interface dar, dass alle Aktivitäten, die sich auf geteilte Ressourcen beziehen implementieren müssen.

eindeutig um einen Vorgang zur Veränderung der Sitzung handelt (Hinzufügen von Projekten).

> Problem [A2MeAk]: Die Einteilung in MESSAGES und Aktivitäten wird verletzt. Damit liegen Spezifikationsverletzungen vor.

Darüber hinaus fand ich heraus, dass LEAVEMESSAGES beim Verlassen einer Sitzung direkt an alle Teilnehmer gesendet werden und nicht zunächst an den Host.

> Problem [A2Leave]: Beim Verlassen einer Sitzung wird eine P2P-Architektur verwendet. Die stellt erneut eine Verletzung der Client-Server-Architektur dar.

Neben dem Architekturbruch hatte die Verwendung einer P2P-Architektur auch einen weiteren Effekt: Da mit dieser Implementierung nicht sichergestellt werden konnte in welcher Reihenfolge die LEAVEMESSAGES bei den Nutzern ankommen, wurde eine feste Wartezeit eingefügt, nachdem die MESSAGE an den Host gesendet wurde. Auf diese Weise sollte sichergestellt werden, dass der Host diese MESSAGE zuerst erhält. Auf eine feste Wartezeit angewiesen zu sein ist immer eine schlechte Idee, in einem verteilten System wie Saros fällt dieser Umstand allerdings noch deutlicher auf. In einem solchen verteilten System kann es immer mal dazu kommen, dass Netzwerke langsam sind oder rechenschwache Knoten an der Bearbeitung teilhaben. Wir können also nie ganz sicherstellen, wie lange eine solche feste Wartezeit sein müsste, um ausreichend zu sein.
Außerdem sorgte die Mischung von P2P und Client-Server offensichtlich auch für Verwirrung bei Programmierern.
In der Entwicklerversion gibt es eine Funktion für den Host, einen Nutzer aus der Sitzung zu entfernen. Diese Funktion funktionierte allerdings nicht, da weitere Teilnehmer nicht darüber informiert wurden, dass ein Nutzer aus der Sitzung entfernt wurde. Der Grund hierfür war, dass der entfernte Nutzer davon ausging, dass der Host die anderen Teilnehmer informiert (Client-Server), während der Host davon ausging, dass der entfernte Nutzer die anderen Teilnehmer informiert (P2P). Das Ergebnis war, dass niemand die anderen Teilnehmer informiert hat.

> Problem [A2Kick]: Das Entfernen von Nutzern funktioniert nicht.

Dieser Punkt war nicht so schlimm, da das Entfernen von Nutzern nur in der Entwicklerversion eingeschaltet war. Er zeigt allerdings, wie unzureichend spezifizierte Architekturentscheidungen zu Versagen führen können.

4.3.6.2 Inkonsistenzen während des Einladeprozesses Das wichtigste Versagen während des Einladeprozesses war, dass es zu Inkonsistenzen während

der Einladung kommen kann, wenn geteilte Dokumente während der Einladung einer dritten Person modifiziert wurden. Dieses Versagen konnte gut reproduziert werden und außerdem gab es einen automatischen GUI-Test, der das Versagen zuverlässig aufgezeigt hatte.

> Problem [I2Inko]: Das Editieren geteilter Ressourcen während der *Project Negotiation* kann zu Inkonsistenzen führen.

Das passiert zum Beispiel bei folgender Ausgangssituation: Wir haben eine Sitzung mit den beiden Teilnehmern Alice und Bob. Alice ist dabei der Host. Alice lädt nun einen weiteren Teilnehmer Carl ein. Während der ganzen Zeit, in der sich Carl im Einladungsprozess befindet editiert Bob eine geteilte Datei. Das Ergebnis ist, dass sich der Dateiinhalt von Carl am Ende der *Project Negotiation* von dem Zustand bei Bob und Alice unterscheidet.
Beim Untersuchen dieses Problems habe ich unter anderem Log-Files zu Rate gezogen. Dabei fiel auf, dass es zu einer Reihe von *Exceptions* kam, wenn Nutzer zwischen dem Ende der *Session Negotiation* und dem Akzeptieren der *Project Negotiation* bestimmte Aktivitäten erhielten. Aus diesen *Exceptions* konnte ich erkennen, dass die Probleme immer beim Verarbeiten von *Ressource*-Aktivitäten entstanden. Genauer, beim Deserialisieren der entsprechenden *ADO*s.
Bei dieser Deserialisierung wird auch der Pfad zur Ressource überprüft. Da die entsprechende Ressource aber noch nicht übertragen worden ist, wird eine *Exception* geworfen und die Aktivität wird verworfen. Das beschriebene Zurückhalten von Aktivitäten (Seite 39) hilft hierbei nicht, da es zu diesem Zeitpunkt noch nicht zur Anwendung kommt.

> Problem [I2AkPN]: Aktivitäten werden während der *Project Negotiation* verworfen.

Dieses Verwerfen von Aktivitäten sorgte außerdem dafür, dass Jupiter-Aktivitäten inkonsistente Jupiter-Zeitstempel bekamen, was ich auch aus *Exceptions* in den Log-Files erkennen konnte. Jupiter-Aktivitäten gehören zu den Aktivitäten, die unter der genannten Umständen verworfen werden. Der Server sendet also immer weiter Jupiter-Aktivitäten und erhöht in Folge dessen seinen *local*-Zeitstempel, während diese Aktivitäten niemals bei Client ankommen und er seinen *remote*-Zeitstempel für diese Ressource daher auch nicht erhöht.
Das Resultat ist, dass sich die Jupiter-Zeitstempel des Servers am Ende der *Project Negotiation* von denen des Clients unterscheiden und der Client Jupiter-Aktivitäten daher nicht mehr anwenden kann, da der Jupiter-Algorithmus Aktivitäten, die einen falschen Zeitstempel besitzen als fehlerhaft betrachtet und ignoriert.
Bei der Betrachtung dieses Versagens fiel mir auf, dass es keine Spezifikation dazu gibt, wann ein die Sitzung betretender Nutzer welche Aktivitätentypen erhalten darf. Es gab nur zwei Zustände: Der Nutzer befindet sich in einer

Sitzung oder er befindet sich nicht in der Sitzung. Damit Aktivitäten allerdings nicht verworfen werden, müssten weitere Zustände definiert sein.

> Problem [I2UseSt]: Nutzerzustände während des Einladungsprozesses sind nicht ausreichend spezifiziert.

Aus den Problemen [I2UseSt] und [I2AkPN] ergab sich noch ein weiteres Versagen. Ein Nutzer Bob, der einer Sitzung beitraten wussten am Ende der *Project Negotiation* nicht, welche Editoren die anderen Teilnehmer offen hatte. Erst wenn der entsprechende Teilnehmer seinen Editor gewechselt hat wurde dies auch bei Bob angezeigt.

> Problem [I2Info]: Statusinformationen werden nach Abschluss der *Project Negotiation* nicht korrekt angezeigt.

Der Grund für dieses Versagen war, dass die Statusinformationen zwar gesendet wurden, dies aber erst geschah, sobald der Nutzer die *Session Negotiation* beendet hatte, wodurch auch diese Aktivitäten verworfen wurden.
Neben den ganzen Versagen, die während des Einladungsprozesses auftreten können gibt es noch ein weiteres Problem, welches sich auf der Ebene der User-Experience befindet. Während der Host das Archiv mit den fehlenden Ressourcen packt wird die Sitzung für alle Teilnehmer blockiert. Bei sehr großen Projekten kann dies bedeuten, dass keiner der Teilnehmer für eine längere Zeit arbeiten kann.

> Problem [I2Block]: Nutzer werden während des Einladeprozesses blockiert. Dadurch werden sie in ihrer Arbeit behindert.

4.3.6.3 Der Watchdog Es gab noch ein weiteres bekanntes Versagen, welches mit dem Entstehen von Inkonsistenzen während des Einladungsprozesses einherging. In einer laufenden Saros-Sitzung wird ein sogenannter *Watchdog* verwendet um inkonsistente Dateien zu finden und diese Inkonsistenzen zu beheben. Dieser *Watchdog*, besteht aus zwei Teilen: Dem *Watchdog*-Server auf Seiten des Host und die *Watchdog*-Clients auf Seiten der anderen Teilnehmer. Der *Watchdog*-Server sendet alle 10 Sekunden *Checksum*-Aktivitäten an alle Teilnehmer einer Sitzung, in denen Informationen über den Inhalt geteilter Dateien enthalten sind, bestehend aus einem Hashcode und dem entsprechende Jupiter-Zeitstempel für diese Jupiter-Ressource. Der Jupiter-Zeitstempel wird dabei benötigt um feststellen zu können, ob sich der Client im selben Zustand wie der Host befindet. Mit diesen Information können *Watchdog*-Clients dann bestimmen, ob die Dateien noch konsistent sind. Wenn nicht, wird dies angezeigt und der Nutzer kann eine Wiederherstellung des korrekten Dateiinhalts anstoßen.

Das Problem hierbei war allerdings, dass die bereits beschriebenen Inkonsistenzen vom *Watchdog* nicht erkannt wurden, obwohl sie eindeutig vorlagen.

> Problem [W2WDog]: Inkonsistenzen können nicht aufgelöst werden.

Dieses Problem wurde bereits mehrfach untersucht, zuletzt von [Sta13]. Als Grund für diese Probleme nannte er inkonsistente Jupiter-Zeitstempel, die dafür sorgen würden, dass der *Watchdog*-Server seine Arbeit einstellt. Leider führt er diese Annahme nicht weiter aus, so dass ich nur spekulieren konnte, dass sie Erklärung dazu war, das der *Watchdog*-Server die Jupiter-Zeitstempel benötigt um den Zustand geöffneter, nicht gespeicherter Dateien überprüfen zu können und er daher aufhört *Checksum*-Aktivitäten zu senden, sobald er feststellt, dass es Inkonsistente Jupiter-Zeitstempel gibt. Da der *Watchdog* die Jupiter-Zeitstempel benötigt um Inkonsistenzen sinnvoll erkennen zu können, erschien mir diese Begründung sinnvoll.
Wie sich später bei der Problembehebung herausstellen sollte, lag das eigentliche Problem allerdings ganz woanders. Mehr dazu im Kapitel 5.3.

4.4 Physische Trennung von Client und Server beim Host

Bei der Betrachtung der aufgezeigten Probleme fiel mir noch ein weiteres Problem auf. Bei unserer jetzigen Client-Server-Architektur fungiert der Host sowohl als Server, als auch als Client.

> Problem [3CSPhy]: Die Client- und eine Server-Komponente beim Host sind nicht physisch getrennt

Zur Zeit ist die Server-Komponente ein Teil des Hosts[13]. Aktivitäten, die an den Host gehen werden zunächst an seinen Server und von da aus intern an den Client-Teil weitergeleitet. Eine andere Möglichkeit der Umsetzung wäre es einen konkreten Saros-Server mit eigener Datenbasis zu haben.
Eine physische Trennung von Client und Server hätte eine Reihe von Vorteilen für die beiden analysierten Szenarien.

1. **Keine Nutzereingaben:** Der Server könnte mit einer eigenen Datenbasis immer genau bestimmten, wann welche Aktivität angewendet werden soll. Zur Zeit muss der Server immer damit rechnen, dass der lokale Nutzer Änderungen an den Dokumenten durchführt. Aus diesem Grund muss der *Main*-Thread von Eclipse zur Synchronisation verwendet werden. Diese Art der Synchronisation könnte entfernt werden.

2. **Verständnis/Modularität:** Eine solche Architektur wäre einfacher zu verstehen, da die Server-Komponenten im Quellcode nicht mit den Client-Funktionalitäten des Hosts verzahnt sind. Zum Beispiel würden viele Abfragen der Form "wenn Host dann..." wegfallen.

[13] Verwendet ursprünglich für Jupiter-Aktivitäten und inzwischen für alle Aktivitäten. Siehe Abschnitt 5.1.1

3. **Dedizierter Server:** Eine eigenständige Server-Komponente hätte außerdem den Vorteil, dass diese Komponente unabhängig von Eclipse-Instanzen laufen könnte. So könnten zum Beispiel Firmen einen dedizierten Server aufsetzen, auf dem eine dauerhafte Sitzung läuft, zu der sich Teilnehmer bei Bedarf verbinden könnten. Darüber hinaus wäre der Server unabhängig von Eclipse, wodurch ein Schritt in Richtung Portierbarkeit von Saros gegangen werden könnte.

4. **Blockierung:** Durch die getrennte Datenbasis wäre es möglich einen nicht blockierenden Einladungsprozess umzusetzen, da die Ressource des Servers nicht von einem Nutzer verändert wird währenddessen. Mit Hilfe dieser eigenen Datenbasis wüsste der Server immer genau, welche Aktivitäten er bereits verarbeitet hat und könnte bestimmte Aktivitäten kurz zurückhalten, solange die entsprechende Ressource gepackt wird.

5 Implementierung

Nachdem ich eine Reihe von Problemen der Saros-Software aufgezeigt hatte, habe ich mich nun an die Behebung der meiner Meinung nach wichtigsten, in einem angemessenen Zeitrahmen erfüllbaren Probleme gemacht. Als wichtigste Probleme habe ich dabei die Probleme [V1CS](Client-Server), [I2Inko] (Inkonsistenzen während des Eiladungsprozesses) und [W2WDog] identifiziert, da die Lösung der Probleme [I2Inko] und [W2WDog] wichtig für die Konsistenz einer Sitzung sind und die Lösung des Problems [V1CS] das System an sich einheitlicher macht und eine wichtige Voraussetzung dafür ist, dass die beiden anderen genannten Probleme sinnvoll behoben werden können.

5.1 Umstellung auf eine Client-Server-Architektur für Aktivitäten

Zugehörige Probleme:
[V1CS] (Client-Server);
[A1CM] (Aktivitätsverteilung im *Concurrency Management*);

Das erste Problem, mit dessen Lösung ich mich im Rahmen dieser Arbeit beschäftigt habe, war die Umstellung der Aktivitätsverarbeitung (senden und empfangen) auf eine einheitliche Client-Server-Architektur. Wie bereits in Kapitel 4 erwähnt, ist der Hauptgrund für diesen Schritt, dass es zum Einen zu Verwirrungen für Entwickler führen kann, wenn mehrere Netzwerkarchitekturen verwendet werden und zum Anderen, dass eine globale Ordnung von Aktivitäten nur mit einer zentralen Verarbeitungsstelle sinnvoll zu realisieren ist.
Bei der Umstellung auf eine Client-Server-Architektur habe ich versucht mich am bereits existierende Senden von Jupiter-Aktivitäten zu orientieren. Dabei sind alle Teilnehmer inklusive des Hosts zunächst einmal Clients. Der Host fungiert allerdings zusätzlich auch als Server. Alle Clients senden ihre Aktivitäten an den Host. Dieser verarbeitet eingehende Aktivitäten als Server und schickt dann die verarbeiteten Aktivitäten an den jeweiligen Client. Wie bereits erwähnt ist auch er selbst Client und leitet die Aktivität daher auch an sich selbst weiter. (siehe Abbildung 10)
Noch besser wäre es gewesen, den Server auch physisch von Host zu trennen (siehe Problem [3CSPhy]). Allerdings wäre der Aufwand für eine solche Umstellung zu hoch gewesen, sodass ich mich für die Erweiterung der vorhandenen Client-Server Implementierung als ersten Schritt in diese Richtung entschieden habe.

5.1.1 Umsetzung der Client-Server-Architektur

Um die Client-Server-Architektur umsetzen zu können, musste ich sowohl die Verteilung bei den Clients, als auch die vorhandene Serverkomponente (verwendet für Jupiter-Aktivitäten) anpassen.
Die Verteilung von Aktivitäten wurde im *Concurrency Mangement* der Clients durchgeführt, der die Aktivität erstellt hat. Abbildung 11 zeigt wie Aktivitäten

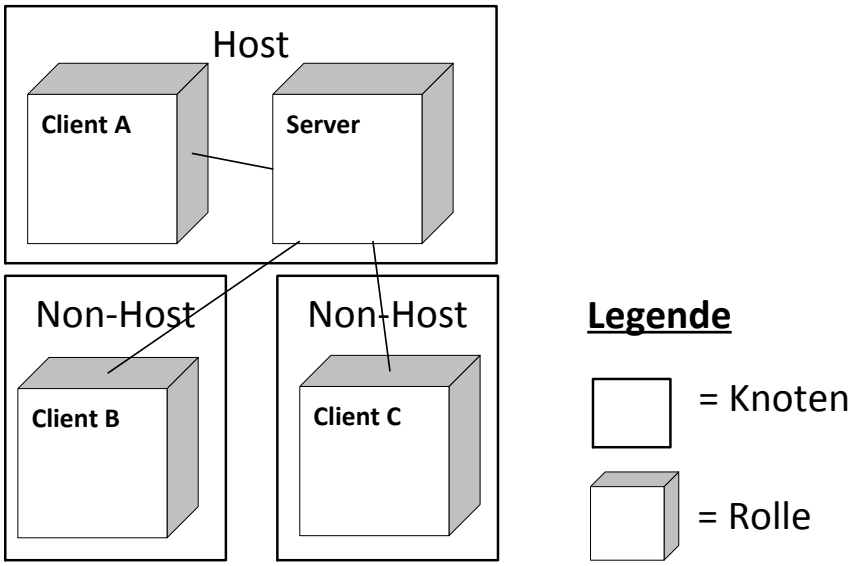


Abbildung 10: Aufteilung des Systems in Clients und Server.

vor der Umsetzung der Client-Server-Architektur verteilt wurden.
Wir gehen davon aus, dass Client B Aktivitäten erzeugt.
Im Falle von Jupiter-Aktivitäten, wurde bereits eine Client-Server-Architektur verwendet. In diesen Fällen wird die Aktivität umgewandelt, mit einem Zeitstempeln versehen und an den Server adressiert. Dieser transformiert die Aktivität für jeden Client individuell und sendet jedem Client eine entsprechende Jupiter-Aktivität. Hier sind das Client C und den lokalen Client A.
Im Falle einer *TargetedActivity* ging die Aktivität direkt an den Empfänger, hier Client C.
Alle anderen Aktivitätstypen wurden direkt an alle Clients gesendet, wobei Aktivitäten für den Host vorher noch über den Server gesendet wurden.
Dieses Verhalten habe ich so angepasst, dass Aktivitäten nun immer zunächst an den Server geschickt werden. Dazu musste ich sowohl die Verarbeitung auf Seiten des Servers, als auch auf Seiten des Clients anpassen. Abbildung 12 zeigt diesen Zustand.
Auch hier gehen wir wieder davon aus, dass Client B Aktivitäten erzeugt. Die Verarbeitung von Jupiter-Aktivitäten musste dabei nicht verändert werden. Alle Aktivitäten, die Client B erzeugt gehen nun nur noch an den Server.
Der Server ist nun für die Verteilung der Aktivitäten zuständig. Jupiter-Aktivitäten werden dabei genauso behandelt wie vorher. Für alle anderen Aktivitäten überprüft der Server jetzt, ob es sich um eine *Targeted*-Aktivität handelt oder nicht.
Targeted-Aktivitäten sind keine eigenständigen Aktivitätstypen. Stattdessen

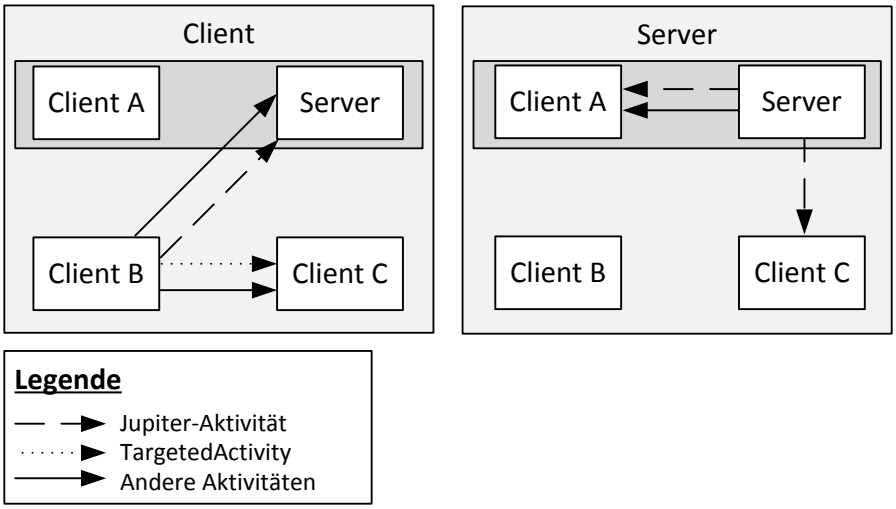


Abbildung 11: Versenden von Aktivitäten ohne Client-Server-Architektur

müssen alle Aktivitätstypen, die das Interfaces die nur an einzelne Teilnehmer gesendet werden sollen das Interface *ITargetedActivity* implementieren. Dieses Interface deklariert einen Empfänger, für den diese Aktivität bestimmt ist.
Alle anderen Aktivitäten gehen an alle Clients, wobei darauf geachtet wird, dass der Erzeuger seine eigene Aktivität nicht zurückbekommt.
Wie an den Abbildungen 11 und 12 zu sehen ist ändert diese Anpassung nichts an der eigentlichen Logik, nach der Aktivitäten verarbeitet werden. Da Aktivitäten auch vorher schon an alle Clients gesendet wurden, sollte es keine Probleme geben, wenn diese nun als Zwischenstation über den Server gehen (Ein Problem trat dabei allerdings doch auf. Siehe Abschnitt 5.1.3). Diese Implementierung hat eine Reihe von Vorteilen.

- Zum Einen sollte die einheitliche Netzwerkarchitektur für ein einfacheres Verständnis sorgen. Nachrichten werden immer an den Server gesendet, egal welchen Typ sie haben. Darüber hinaus ist jetzt klar definiert, dass wir eine Client-Server-Architektur verwenden.

- Zum Anderen gibt es nun eine zentrale Stelle, die erstellte Aktivitäten passieren müssen. So kann eine globale Reihenfolge sichergestellt werden und eventuell auftretende Versagen durch direkt gesendete Aktivitäten können verhindert werden.

Darüber hinaus habe ich die Adressierung der Aktivitäten in eine andere Komponente verlagert. Bisher geschah diese Adressierung im *Concurrency Management*. Diese ist per Definition nur für die Umwandlung von Jupiter-zugehörigen Aktivitäten zuständig (Problem [A1CM]). Daher habe ich die Adressierung von Aktivitäten in das übergeordnete *Activity Handling* verschoben, sodass das

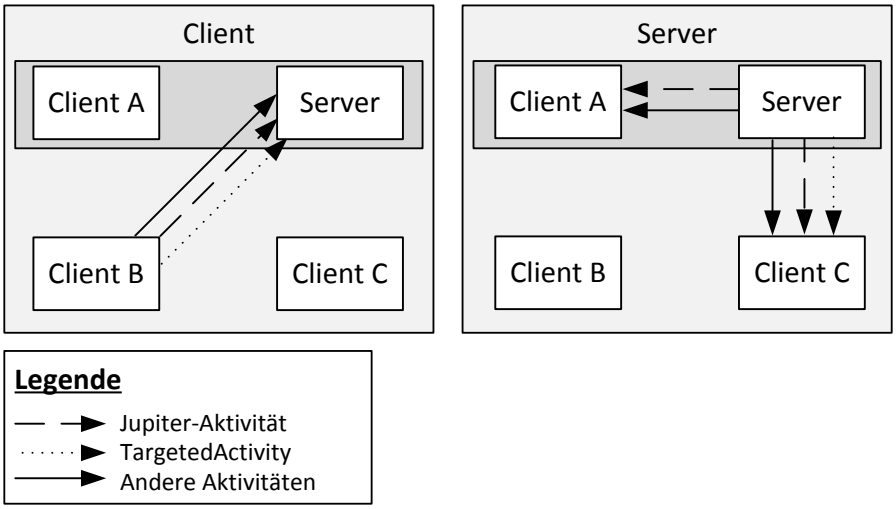


Abbildung 12: Versenden von Aktivitäten mit Client-Server-Architektur

Concurrency Management jetzt wirklich nur noch die Aktivitäten bearbeitet, die auch wirklich umgewandelt werden müssen.

5.1.2 Sonderbehandlung von Read-Only Nutzern

Zugehörige Probleme:
[A1RO] (Sonderbehandlung von Read-Only Nutzern);

Da ich gerade dabei war das *Concurreny Management* umzubauen, habe ich mich auch gleich um die Sonderbehandlung für Nutzer, die nur lesenden Zugriff haben, gekümmert.
Wie bereits im Zuge von der Beschreibung von Problem [A1RO] erwähnt, handelte es sich bei dieser Sonderbehandlung um eine eigentlich unnötige Verbesserung, die in deutlich undurchsichtigerem Code resultierte. Darüber hinaus sorgte die Umstellung auf eine Client-Server Architektur dafür, dass durch die Sonderbehandlung nun eine zusätzliche Umwandlungen nötig war.
Vorher wurde eine *TextEdit*-Aktivität an jeden Nutzer ohne Schreibrechte geschickt und zusätzlich eine Jupiter-Aktivität an den Server. Da die Verteilung nun aber vom Server ausgeht, haben Clients immer nur noch eine Jupiter-Aktivität an den Server gesendet und dieser musste die dann wieder in *TextEdit*-Aktivität für Nutzer ohne Leserechte umwandeln. Dadurch fiel sogar der bereits geringe Vorteil, den die Sonderbehandlung hatte, weg. Auf Grund dieser Überlegung habe ich das *Concurrency Management* so umgebaut, dass alle Nutzer ihre Jupiter-Ressourcen behalten und zwar auch dann wenn sie keine Schreibrechte mehr besitzen. Dabei habe ich das XP-Prinzip "Simplicity" angewandt [Bec99], das besagt, dass immer die einfachste korrekte Lösung im-

plementiert werden soll.
Dazu habe ich einfach den *listener* entfernt, der vorher auf Änderungen der Rechte eines Nutzers reagierte und anschließend alle Sonderbehandlungen entfernt, sodass es nun eine einheitliche Behandlung von *TextEdit*-Aktivitäten gibt. Durch diese Änderung konnte ich sechs Abhängigkeiten entfernen, sowie zwei *listener*. Darüber hinaus konnte ich vier Abfragen auf Schreibrechte entfernen. Außerdem gab es, wie sich später herausstellte, sogar echte Versagen, die mit dieser Sonderbehandlung in Zusammenhang standen (Siehe Abschnitt 5.3.)

5.1.3 Problem ITargetedWrapper

Nach der Umstellung auf eine Client-Server-Architektur fiel mir auf, dass plötzlich eine Reihe von Aktivitäten, die eigentlich nur an einen Teilnehmer gehen sollten, an alle Teilnehmer gesendet wurden.
Vorher wusste ein Client B anscheinend, dass diese Aktivitäten *Targeted* waren und sendete sie direkt an den Empfänger (siehe Abbildung 11 auf Seite 47). Nach der Umstellung sendet er nun alle Aktivitäten (siehe Abbildung 12 auf Seite 48) an den Server. Dieser schien allerdings nicht mehr zu wissen, dass es sich um eine *Targeted*-Aktivität handelte und sendete diese Aktivität daher an alle Teilnehmer. Die Information, dass es sich dabei um eine *Targeted*-Aktivität handelt, schien also bei Senden verloren zu gehen.
Bei der Untersuchung des Problems stellte sich heraus, dass keine der fraglichen Aktivitäten das Interface *ITargetedActivity* implementierte und somit auch keinen Empfänger haben dürfte. Es schien also Sonderfälle zu geben, in dem Aktivitäten, die eigentlich keine *Targeted*-Aktivitäten sind, als solche behandelt wurden.
Eine Suche gründliche Suche nach diesen Sonderfällen förderte den *ITargeted-Wrapper* zu Tage. Dieser wurde verwendet um Aktivitäten in *TargetedActivites* umzuwandeln und ihnen so einen Empfänger zuzuordnen. Eine solche *ITargeted-Wrapper*-Aktivität bestand dabei aus einen Empfänger und der entsprechenden Aktivität.
Bevor eine Aktivität versendet wird, wird sie in ein entsprechendes *ADO* umgewandelt (siehe 27). Bei dieser Umwandlung wurde allerdings die Anforderung I verletzt. Laut dieser dürfen Aktivitäten bei der Umwandlung in *ADO*s und zurück nicht verändert werden. In diesem Fall wurde allerdings nur die enthaltene Aktivität selbst umgewandelt und nicht der *ITargetedWrapper* als Ganzes. Als Ergebnis verschwand der Empfänger, sobald die Aktivität versendet wurde, wodurch der Server nicht mehr wissen, konnte, dass es sich um eine *Targeted*-Aktivität handelt.

> Problem [3Wrap]: Der *ITargetedWrapper* bricht Anforderung I und erschwert die Verteilung von Aktivitäten

Um das Problem zu beheben habe ich zunächst eine gründliche Analyse darüber durchgeführt, welche Aktivitäten mit Hilfe des *ITargetedWrappers* umgewandelt wurden und warum. Dabei stellte ich fest, dass sich die Gründe in drei Kategorien einteilen ließen:

1. **Echte *TargetedActivities:*** Ein paar Aktivitätstypen wurden immer, wenn sie benutzt wurden mit dem *ITargetedWrapper* verpackt. Diese Aktivitäten sind somit echte *Targeted*-Aktivitäten, die allerdings nicht als solche definiert wurden.

2. **Performance:** Es gab auch eine Reihe von Aktivitätentypen, die aus Gründen der Performanz in bestimmten Fällen nur an einzelne Teilnehmer geschickt wurden. Alle Aktivitätstypen in dieser Kategorie waren Statusinformationen, die gesendet wurden um einem Nutzer, der die Sitzung betritt mitzuteilen, wer zum Beispiel welche Editoren offen hat.

3. **Sonderfälle:** In diese Kategorie fallen Aktivitätentypen, die in bestimmten Sonderfällen nur an einen Teilnehmer gehen dürfen. Es gab nur einen Aktivitätstpen, der in diese Kategorie fiel. Während der Wiederherstellung (*Watchdog*) verwendete *File*-Aktivitäten, die den Inhalt der wiederherzustellenden Dateien enthielten.[14]

Nachdem ich diese Analyse durchgeführt hatte, entschied ich mich dafür den *ITargetedWrapper* zu entfernen und die Fälle, in denen der *ITargetedWrapper* verwendet wurde umzubauen. Aktivitätstypen, die in die erste Kategorie fielen, habe ich dabei einfach das Interface *ITargetedActivity* implementieren lassen.
Bei Kategorie 2 habe ich mich dazu entschieden diese Aktivitäten an alle Teilnehmer zu senden. Der Grund dafür ist einfach. Es schadet auch den anderen Teilnehmern nicht aktualisierte Statusinformationen zu erhalten.
Der dritte Fall war ein wenig komplizierter. Da es allerdings nur einen Aktivitätstypen in dieser Kategorie gab entschied ich mich dazu den Sonderfall durch eine eigene Aktivität darzustellen, die *RecoveryFile*-Aktivität. Diese neue Aktivität erbt von der *File*-Aktivität und ist zusätzlich noch *targeted*. Auf diese Weise können normale *File*-Aktivitäten weiterhin wie vorher auch behandelt werden und der Sonderfall kann an einzelne Teilnehmer geschickt werden.
Nachdem ich diese Fälle so umgebaut hatte, dass sie den *ITargetedWrapper* nicht mehr benötigten, konnte ich ihn komplett entfernen. Anschließend funktionierte das Senden von Aktivitäten wieder wie erwartet.

Bei der Behebung dieses Defektes und dem damit einhergehenden Beschäftigen mit *Targeted*-Aktivitäten fiel mir außerdem auf, dass es eigentlich die Aufgabe des Servers sein sollte Empfänger für Aktivitäten festzulegen. Zur Zeit reagiert die Server nur auf bereits gesetzte Ziele der Clients.

Problem [3CliTar]: Clients bestimmten Empfänger von *Targeted*-Aktivitäten

[14] Die Aktivitätsmethapher als "Aktionen des Lokalen Nutzers" wird hier ein wenig aufgeweicht, da der lokale Nutzer die Wiederherstellung nicht direkt steuert. In diesem Fall agiert sozusagen der *Watchdog* als "lokaler Nutzer". Dieser Fall ist aber mit Anforderung III beschrieben Aufteilung von MESSAGES und Aktivitäten konform.

5.2 Architekturbrüche Aktivitätsverarbeitung

Zugehörige Probleme:
[A1ArBr] (Architekturbrüche beim Aktivitätsverarbeitung);
[A2MeAk] (Trennung von MESSAGES und Aktivitäten);

Nachdem ich die Umstellung auf eine Client-Server-Architektur für Aktivitäten abgeschlossen hatte, wollte ich abschließend für dieses Problemgebiet noch einige aufgedeckte Architekturbrüchen beheben, bevor ich mich mit den Problemen im Einladungsprozess beschäftige. Der Hauptgrund dafür war, dass ich die Verarbeitung von Aktivitäten so einheitlich wie möglich gestalten wollte.
Dazu habe ich zunächst für sämtliche Aktivitätstypen, die noch keinen besaßen, einen entsprechenden *Activity Provider* erstellt. Diese Aktivitäten wurden vorher direkt im *Activity Handling* erzeugt und verwendeten die bereits als *deprecated* markierte Methode *abuseActivityCreated()*. Von diesen Aktivitätstypen gab es zwei.
Außerdem habe ich noch geschaut, dass die Trennung von MESSAGES und Aktivitäten eingehalten wird. Ich habe dabei alle MESSAGES und Aktivitätstypen untersucht und festgestellt, dass es einen Fall gab, indem eine Aktivität für die Kommunikation während des Einladeprozesses verwendet wurde. Die Definition von Aktivitäten und MESSAGES besagt allerdings, dass alle Prozesse, die dazu dienen die Sitzung selbst zu verändern mit Hilfe von MESSAGES kommunizieren sollen (siehe Anforderung III auf Seite 38). Um dieser Definition gerecht zu werden, wurden diese Aktivität durch eine entsprechende MESSAGE ersetzt.

5.3 Watchdog

Zugehörige Probleme:
[W2WDog] (Der Watchdog funktioniert nicht);

Das nächste Problemgebiet, mit dessen Lösung ich mich beschäftigen wollte war, die Behebung von Inkonsistenzen während des Einladungsprozesses [I2Inko]. Wie bereits bei der Problembeschreibung erwähnt, hatte Starroske in seiner Masterarbeit [Sta13] einen Zusammenhang zwischen Inkonsistenten Jupiterzeitstempeln und dem nicht mehr funktionierenden *Watchdog* hergestellt. Seine These war, dass der *Watchdog*-Server keine weiteren *Checksum*-Aktivitäten mehr senden würde, sobald er inkonsistente Jupiter-Zeitstempel entdeckt hätte. Aufgrund dieser These bin ich davon ausgegangen, dass inkonsistente Jupiter-Zeitstempel sowohl für das Problem [W2WDog] als auch für das Problem [I2Inko] verantwortlich waren, weswegen ich dachte, dass die Lösung von Problem [I2Inko] auch den *Watchdog* wieder funktionstüchtig machen würde. Die Lösung von Problem [I2Inko] findet sich im nächsten Kapitel 5.4.
Beim Durchsehen von Logfiles zum Nachvollziehen des Problems [I2Inko] fiel mir auf, dass trotz inkonsistenter Jupiter-Zeitstempel *Checksum*-Aktivitäten versendet wurden. Die These von Starroske ließ sich also nicht halten. Aufgrund dieser neuen Erkenntnis beschloss ich mich doch zuerst mit Problem

[W2WDog] zu beschäftigen.

Nachdem ich nun wusste, dass auch bei inkonsistenten Jupiter-Zeitstempeln *Checksum*-Aktivitäten gesendet wurden, sah ich mir diese genauer an und stellte fest, dass das Jupiter-Zeitstempel-Feld dieser Aktivitäten beim Client immer *null* war. Zur Wiederholung: Der *Watchdog* braucht Jupiter-Zeitstempel um feststellen zu können, in welchem Zustand sich geöffnete Dokumente befinden. Der *Watchdog*-Client überprüft für jede eingehende *Checksum*-Aktivität anhand des gesendeten Jupiter-Zeitstempels, ob die Aktivität aktuell ist. Dazu wird der gesendete mit dem eigenen aktuellen Jupiter-Zeitstempel verglichen und nicht mehr aktuelle *Checksum*-Aktivitäten werden vom Client ignoriert.
Bei einem Jupiter-Zeitstempel von *null* wird allerdings nur überprüft ob der lokale Zeitstempel 0 ist. Wenn nicht wird die *Checksum*-Aktivität als nicht mehr aktuell angesehen und ignoriert.
Da der Zeitstempel immer *null* war wurden daher auch alle *Checksum*-Aktivitäten ignoriert, sobald der Jupiter-Zeitstempel mindestes einmal erhöht worden ist, also sobald es schon mindestens eine Änderungen an der Datei gegeben hat. Der *Watchdog* funktionierte also nur, solange niemand die Ressource ändert. Für diese Fälle ist eine Inkonsistenzprüfung allerdings im Grunde nutzlos.

Ich musste also herausfinden an welcher Stelle der Zeitstempel "verschwindet". Nachdem ich das Senden von Aktivitäten allerdings bereits gründlich untersucht hatte, konnte ich den Defekt recht schnell finden. Beim Umwandeln in ein entsprechendes *ADO*, wurde der Zeitstempel schlicht und einfach nicht mit umgewandelt. Das lag daran, dass es zwei Konstruktoren für die Erstellung eines *Checksum*-ADOs gab. Einer davon mit Zeitstempel, der andere ohne.
Vor über drei Jahren wurden alle Aktivitätstypen auf Aktivitäten und serialisierbare *ADO*s aufgeteilt. Davor befand sich der Code zum serialisieren direkt in den Aktivitäten selbst. Für *Checksum-ADOs* wurden dabei die beiden Konstruktoren implementiert. Der Konstruktor ohne Zeitstempel war dabei für Client ohne Leserecht gedacht (Siehe dazu auch Abschnitt 5.1.2).
Leider hatte der damalige Entwickler diesen zweiten Konstruktor für die Umwandlung aller *Checksum*-Aktivität verwendet, was zur Folge hatte, dass der *Watchdog* seit über drei Jahren nicht mehr korrekt funktionierte.

An diesem Beispiel zeigt sich nochmal deutlich, dass die Sonderbehandlung von Nutzern ohne Schreibzugriffen (siehe Abschnitt 5.1.2) und der damit verbundene undurchsichtigere Quellcode auch zum Einbau wirklicher Defekte geführt hat. Daher war die Entscheidung diese Sonderbehandlung zu entfernen im Nachhinein genau die Richtige.
Für die eigentliche Fehlerbehebung musste ich nur den korrekten Konstruktor verwenden. Nach dieser Änderung (Anpassung einer einzigen Zeile) funktionierte der *Watchdog* wieder einwandfrei und Inkonsistenzen, die beim Einladungsprozess entstanden, konnten wieder problemlos aufgelöst werden. Die inkonsistenten Zeitstempel hatten also nichts mit dem eigentlichen Problem zu tun. Stattdessen lag der Defekt in der Umwandlung von Aktivitäten in ADOs.
Die Tatsache, dass dies schon der zweite Defekt war, der durch die Verletzung

der Anforderung I entstanden war, veranlasste den *Product Owner* außerdem dazu Unittests für die Umwandlung von Aktivitäten in ADOs und zurück für alle Aktivitätstypen zu schreiben, damit solche Probleme in Zukunft nicht mehr auftreten.

Bei der Behebung dieses Defekts fand ich darüber hinaus auch eine mögliche Erklärung dafür, wie Starroske zu seiner fehlerhaften Annahme gelangt sein könnte. *Checksum*-Aktivitäten werden nur dann in die entsprechenden Logfiles geschrieben, wenn die Eclipse-Instanzen im Debug-Modus gestartet wurden.
Ich vermute, dass ihm dieser Umstand nicht bewusst war und er sich die Logfiles unterschiedlicher Sitzungen mit unterschiedlichen Logleveln angeschaut hat. In diesem Fall kann es sein, dass er keine *Checksum*-Aktivitäten mehr gefunden hat, weil er die Sitzung mit dem falschen Loglevel gestartet hatte und so davon ausging, dass keine Aktivitäten mehr generiert werden.
Leider kann ich hier nur Vermutungen anstellen, da Starroske in seiner Arbeit nicht beschrieben hat, woran er festgemacht hat, dass der *Watchdog* aufhören würde Aktivitäten zu senden. Ich halte diese Begründung aber für durchaus plausibel.

5.4 Einladungsprozess

Zugehörige Probleme:
[I2Inko] (Inkonsistenzen während des Einladungsprozesses);
[I2AkPN] (Verwerfen von Aktivitäten);
[I2UseSt] (Nutzerstatus nicht ausreichend definiert);
[I2Info] (Statusinformationen werden nicht übertragen);

Nachdem ich nun den *Watchdog* wieder funktionstüchtig gemacht hatte, konnte ich mich weiter mit dem Einladungsprozess beschäftigen.

5.4.1 Zeitpunkt für das Senden von Aktivitäten definieren

Als Erstes habe ich mich mit den offensichtlich zu früh gesendeten Aktivitäten beschäftigt.
Wie bereits in der Analyse erwähnt gab es eine Reihe von Aktivitäten, die zu früh gesendet und aufgrund fehlender Projekte verworfen wurden, was zu einer Reihe von Problemen führte. Die Vermutung war, diese verworfenen Aktivitäten auch für die Inkonsistenzen am Ende des Einladungsprozesses verantwortlich waren.
Um dieses Problem anzugehen, habe ich zunächst eine Reihe von Zuständen definiert, in denen sich ein Nutzer befinden kann. Jeder dieser Zustand ist mit klar definierten Aktionen verbunden und erlaubt dem Nutzer bestimmte Aktivitäten zu empfangen.
Abbildung 13 zeigt die einzelnen Zustände, in denen sich Nutzer befinden können.
Zwischen den Zuständen **outOfSession** und **inSession** hätte man noch den Zustand **inSessionNegotiation** einfügen können, der beschreibt, dass sich ein

Nutzer in der *Session Negotiation* befindet. Dieser Zustand wurde allerdings bewusst weggelassen, da in diesem Zustand die gleichen Einschränkungen gelten, wie im Zustand **outOfSession**

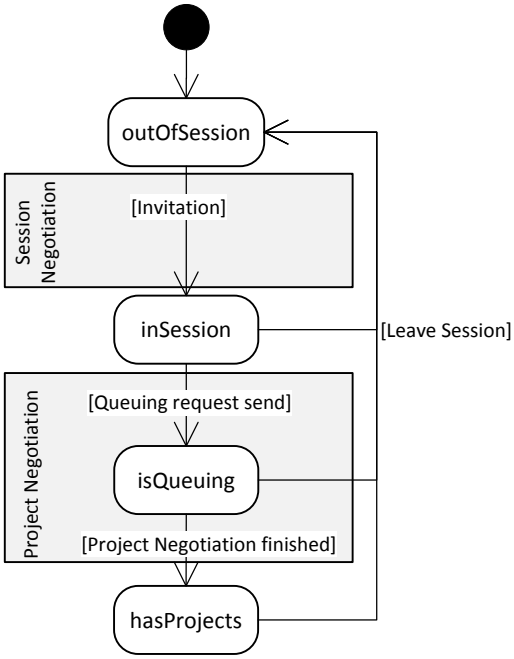


Abbildung 13: Stati in denen sich ein Nutzer während der Einladung befinden kann.

- **outOfSession:** Der Initialzustand eines jeden Nutzers. Er zeigt an, dass der Nutzer nicht Teil der laufenden Sitzung ist und somit auch keine Aktivitäten erhalten darf. Er kann diesen Zustand verlassen, indem er der Sitzung betritt. Sobald ein Nutzer die *Session Negotiation* abgeschlossen hat befindet er sich im Zustand **inSession**

- **inSession:** Dieser Zustand zeigt an, dass der Nutzer der laufenden Sitzung beigetreten ist. In diesem Zustande kann der Nutzer alle Funktionen von Saros nutzen, die nicht an geteilte Ressourcen gebunden sind, wie zum Beispiel den Multiuser-Chat [15].

  Nutzer in diesem Zustand dürfen Aktivitäten erhalten, allerdings nur solche, die sich nicht auf geteilte Ressourcen beziehen, wie zum Beispiel die *ChangeColor*-Aktivität [16].

[15] Ein Chatroom, indem sich alle Mitglieder einer Sitzung befinden.

[16] Diese zeigt an, dass ein Nutzer in der Sitzung seine Farbe geändert hat. Diese Farbe wird zum Beispiel für den Chat verwendet und zeigt an, welche Änderungen der Nutzer in einer Datei durchgeführt hat.

Nutzer verlassen diesen Zustand, sobald sie sich in der *Project Negotiation* befinden und damit begonnen haben eingehende *Resource*-Aktivitäten, in der Warteschlange abzulegen (siehe 39).

- **isQueuing:** Dieser Zustand zeigt an, dass der Nutzer *Resource*-Aktivitäten, in der Warteschlange ablegt. In diesem Zustand darf der Nutzer alle Aktivitäten erhalten, sie aber nicht ausführen. Sobald er die *Project Negotiation* beendet hat geht er in den im Zustand **hasProjects** über.

- **hasProjects:** Dieser Zustand zeigt an, dass der Nutzer ein vollwertiger Teilnehmer ist und sämtliche Aktivitäten empfangen und anwenden kann. In diesem Zustand muss der Nutzer den aktuellen Zustand der Sitzung kennen.

- Jeder Nutzer kann durch Verlassen der Sitzung oder Abbrechen der Einladung in den Zustand **outOfSession** zurückkehren.

Um die Einhaltung dieser Zustände sicherzustellen mussten ich eine Reihe von Änderungen durchführen.
Für den Zustand **outOfSession** gab es nichts zu tun, da Nutzer, die sich nicht in der Sitzung befanden, bereits vorher keine Aktivitäten erhielten.
Vorher gab es allerdings keine klar definierten weiteren Zustände und ein Nutzer ging vom Zustand **outOfSession** direkt in den Zustand **hasProjects** über. Das heißt, dass ein Nutzer sobald er die *Session Negotiation* beendet hatte, alle Aktivitäten erhielt, auch wenn er die entsprechenden Projekte noch nicht besaß.
Die Idee, das Zwischenspeichern von Aktivitäten einfach auf den gesamten Einladungsprozess zu erweitern würde, nicht funktionieren, da diese zurückgehaltenen Aktivitäten vom Host eventuell schon auf die entsprechenden Ressourcen angewandt wurden, wenn er, während der *Project Negotiation* die entsprechenden Ressourcen überträgt. Der Client würden also nicht wissen, welche Aktivitäten der Host bereits angewendet hat und würden einige Aktivitäten zweimal auf die gleiche Ressource anwenden.

Um den Anforderungen des Zustands **inSession** gerecht zu werden, habe ich eine neue Datenstruktur angelegt, die jedem Nutzer eine Liste von Projekten zuordnet, die er bereits besitzt. Neue Projekte werden dieser Liste während der *Project Negotiation* hinzugefügt. Bevor Aktivitäten gesendet werden, überprüft der Server zunächst ob es sich um eine *Resource*-Aktivität handelt, die sich auf eine geteilte Ressource bezieht. Wenn dem so ist überprüft der Server außerdem für jeden Nutzer, ob er Aktivitäten für dieses Projekt bereits empfangen darf und sendet oder verwirft sie dementsprechend.
Die neuen Projekte werden zu dieser Liste hinzugefügt, sobald der Host den Client aufgefordert hat mit dem Zwischenspeichern von Aktivitäten zu beginnen. Dadurch kann der Client nun auch alle Aktivitäten erhalten. Das ist der Zeitpunkt, indem der Client in dem Zustand **isQueuing** übergeht. Sobald das geschieht, wird der neue Nutzer auch zum Jupiter-Server der jeweiligen geteilten Ressourcen hinzugefügt. Wie bereits in der Analyse erwähnt kam es immer

wieder zu Inkonsistenzen, weil Jupiter-Aktivitäten, die im Zustand **inSession** gesendet wurden, einfach verworfen wurden.
Durch die Einführung der beiden Zustände **inSession** und **isQueuing**, sowie die Einhaltung der Anforderungen dieser Zustände konnten ich verhindern, dass Aktivitäten zu früh geschickt werden. Dadurch ließen sich inkonsistente Jupiter-Zeitstempel und vom Client geworfene *Exceptions* für Aktivitäten, die sich auf noch nicht vorhandene Ressourcen beziehen, verhindern.

Zu guter Letzt musste ich noch sicherstellen, dass auch die Anforderung an den letzten Zustand erfüllt werden. In diesem Zustand muss ein Nutzer den aktuellen Zustand der Sitzung kennen. Um das sicherzustellen, schickten bereits vorher alle Nutzer einer Sitzung Informationen darüber, welche Dateien sie bei sich lokal gerade geöffnet haben. Leider war auch hier der Zeitpunkt falsch gesetzt. Das Senden dieser Informationen geschah im Zustand **inSession**, wodurch die entsprechenden Aktivitäten verworfen wurden und der hinzukommende Nutzer im Zustand **hasProjects** keine Informationen darüber hatte, welche Dateien bei den anderen Teilnehmern geöffnet waren. Dieser Zustand hielt solange an, bis der entsprechende Nutzer den aktiven Editor wechselte.
Um auch dieses Problem zu lösen habe ich den Zeitpunkt, zu dem Nutzer ihre Statusinformationen senden, in den Zustand **hasProjects** verschoben. Auf diese Weise weiß der Nutzer nun welche Dateien von welchem Teilnehmer geöffnet wurden, sobald er den Zustand **hasProjects** erreicht hat.

5.4.2 Nur der Host darf Projekte hinzufügen

Schließlich habe ich implementiert, dass neue Projekten nur noch vom Host hinzugefügt werden dürfen. Dies hatte mehrere Gründe. Der Erste war, dass die Einhaltung der vorangegangenen Nutzer-Zustände nur dann sichergestellt werden kann, wenn die *Project Negotiation* vom Host ausgeht. Darüber hinaus gab es noch eine Reihe weiterer Schwierigkeiten, die durch das Hinzufügen von Projekten durch "Non-Hosts" verursacht worden. Wenn Client B zum Beispiel ein Projekt X hinzufügt und Client C dieses sofort annimmt, der Host aber nicht, besitzen B und C bereits das neue Projekt, Änderungen die sie an diesen Vornehmen werden aber nicht übertragen, da der Host für dieses Projekt noch keinen Jupiter-Algorithmus aufgesetzt hat. Wenn nun der Host also verspätet das neue Projekt akzeptiert werden alle Änderungen, die C gemacht hat, inkonsistent, da sie niemals übertragen wurden.
Außerdem widerspricht diese Funktion der angestrebten Client-Server-Architektur, da hier Projekte direkt von Client zu Client versendet wurden, sodass das Deaktivieren einen weiteren Schritt in Richtung Client-Server-Architektur darstellt. Darüber hinaus wurde das Einladen von Nutzern für "Non-Hosts" bereits deaktiviert, sodass diese Änderung nur der nächste logische Schritt war. In Zukunft könnte diese Funktion wieder aktiviert werden, wenn sichergestellt werden kann, dass Clients das Projekt, welches sie zur Sitzung hinzufügen wollen zunächst nur an den Host senden und dieser sich dann um die Verteilung kümmert.

5.4.3 Inkonsistenzen nach der *Project Negotiation*

Zugehörige Probleme:
[I2Inko] (Inkonsistenzen während des Einladungprozesses);

Nachdem ich dafür gesorgt hatte, dass Jupiter-Zeitstempel während der Einladung nicht mehr inkonsistent wurden hatte, ich erwartet, dass auch die Inkonsistenzen in den Dateien selbst nicht mehr auftreten würden. Leider erwies sich diese Annahme als fehlerhaft. Die Jupiter-Zeitstempel schienen nur ein Teil des Problems gewesen zu sein. Der Zeitraum, indem Inkonsistenzen auftreten konnten, konnte zwar durch die Änderung 5.4.1 deutlich verringert worden, doch es gab trotzdem noch einen kurzen Zeitraum, indem es zu Inkonsistenzen auftreten kommen konnte. Dies schien immer dann der Fall zu sein, wenn Aktivitäten während des Einladungsprozesses in der Warteschlange abgelegt wurden, also im Zustand **isQueuing** gesendet wurden.

Um diese Problem zu untersuchen, habe ich testweise Teilnehmer C eingeladen. Während der Einladung habe ich eine geteilte Datei modifiziert, indem ich an das Ende der Datei eine Reihe von "s" angehängt habe.

Bei der Suche nach diesem Problem habe ich zunächst die dabei entstandenen Logfiles zu Rate gezogen. Dabei stellte ich fest, dass folgende Meldung (in leicht abgeänderten Varianten) immer wieder auftrat:

> Could not apply TextEdit at 28-28 of document with length 27.
> Was supposed to replace ' ' with 's'.

Bei näherer Betrachtung fand ich heraus, dass diese Meldung ab der zweiten *TextEdit*-Aktivität, die auf eine Ressource angewandt werden sollte, auftrat. Daraus konnte ich den Schluss ziehen, dass die erste *TextEdit*-Aktivität nicht korrekt angewendet wurde.
Im Fall der hier betrachteten Logfile-Ausgabe hatte die Ressource beim Anwenden der ersten *TextEdit*-Aktivität 27 Zeichen[17] und an Stelle 27 sollte nun ein "s" eingefügt werden. Die nächste Aktivität, die angewendet werden soll, versucht nun ein "s" an Stelle 28 einzufügen. Dabei entsteht die gesehene Fehlermeldung. Offensichtlich wurde das erste "s" nicht eingefügt, sodass das Dokument immer noch 27 Zeichen hat und somit an Stelle 28 kein neues Zeichen eingefügt werden kann.
Die Frage war also: Wieso wird die erste Aktivität nicht angewendet?

Um diese Frage zu beantworten muss zunächst das Konzept der Eclipse *DocumentProvider* verstanden werden. Eclipse schreibt gemachte Eingaben in einer Datei nicht direkt auf die Festplatte. Anstelle dessen werden *DocumentProvider* verwendet um diese Eingaben zu puffern. Jedem geöffneten Editor ist dabei ein

[17] Das Dateiende wird dabei als eigenes Zeichen betrachtet, sodass Stelle 27 das Dateiende darstellt und ein Einfügen ans Ende der Datei an Stelle 27 geschieht.

solcher *DocumentProvider* zugeordnet. Im Normalfall werden *DocumentProvider* entfernt, sobald der Editor geschlossen wird, da Eclipse beim Schließen die gepufferten Änderungen entweder verwirft oder auf die Festplatte schreibt.
Für eingehende *TextEdit*-Aktivitäten wird der entsprechende *DocumentProvider* verwendet, um die gemachten Änderungen im Editor des lokalen Nutzers anzuzeigen. In einer Sitzung können allerdings auch *TextEdit*-Aktivitäten für Dateien eingehen, die der lokale Nutzer gar nicht geöffnet hat. Im Normalfall würden der entsprechende *DocumentProvider* dabei sofort nach dem Schreiben wieder entfernt werden, da kein entsprechender Editor geöffnet ist. Um dies zu verhindern führt jeder Nutzer in Saros eine Liste mit allen *DocumentProvidern*, die zu einem Editor gehören, der bei mindestens einem Teilnehmer geöffnet sind. Auf diese Weise gibt es immer eine Referenz auf den entsprechenden *DocumentProvider* und dadurch können die Änderungen auch in lokal nicht geöffneten Editoren gepuffert werden.
Zurück zum eigentlichen Problem der nicht angewendeten *TextEdit*-Aktivitäten während des Anwendens von Aktivitäten aus der Warteschlange.
Mit der Hilfe eines anderen Entwicklers (Stefan Rossbach) kam ich dahinter, dass der Defekt darin lag, dass der entsprechende Editor nie aktiviert wurde. Ein *DocumentProvider* wird erst dann angelegt, wenn von mindestens einem Teilnehmer eine entsprechende Aktivität kam, die besagt, dass er den entsprechenden Editor geöffnet hat. Diese Statusinformationen werden erst gesendet, nachdem die Warteschlange bereits geleert wurde. *TextEdit*-Aktivitäten, die vorher angewendet werden haben also noch keinen gespeicherten *DocumentProvider* und werden somit nicht gepuffert.
Um das Problem zu lösen haben wir (Stefan Rossbach und ich) verschiedene Ansätze diskutiert, wie sichergestellt werden kann, dass die Statusinformationen vor den *TextEdit*-Aktivitäten angewendet wurden. Das größte Problem dabei war, dass diese Statusinformationen über Aktivitäten gesendet werden und wir nicht mit Sicherheit sagen können wie lange es dauert, bis alle Aktivitäten angekommen sind, da wir nicht wissen, welcher Teilnehmer wie viele Editoren offen hat. Auf diese Aktivitäten zu warten fiel also als Möglichkeit aus.
Im Endeffekt haben wir uns für eine Lösung entschieden, die sich die Tatsache zu Nutze macht, dass *TextEdit*-Aktivitäten von einem aktiven Editor kommen müssen. Dazu haben wir beim Leeren der Warteschlange vor jede *TextEdit*-Aktivität eine zusätzliche Aktivität eingefügt, die zunächst den entsprechenden Editor aktiviert, wenn dies nicht schon getan wurde. Auf diese Weise werden die Änderungen nicht verworfen, da der entsprechende *DocumentProvider* bereits zur Liste der *DocumentProvider* hinzugefügt wurde.
Diese Änderung, in Verbindung mit dem Verhindern von inkonsistenten Jupiter-Zeitstempeln, sorgte dafür, dass Problem [I2Inko] endlich behoben werden konnte und Inkonsistenzen während des Einladungsprozesses verhindert werden können.

5.5 Zusammenfassung und aktueller Stand

Die folgende Tabelle zeigt einen Überblick über die identifizierten Probleme und den Stand bei Abgabe dieser Arbeit. Eine Reihe von Problemen konnten

von mir behoben werden. Ein paar dieser Probleme wurden auch von anderen Entwicklern behoben, nachdem sie durch meine Analyse darauf aufmerksam gemacht wurden.

ID	**Kurzbeschreibung**
[A1CM]	Aufgaben *Concurrency Management*
Status	Wurde von mir behoben. Das *Concurrency Management* ist nur noch für Jupiter-Spezifische Aktionen zuständig, sodass eine klare Trennung der Aufgaben vorliegt.

ID	**Kurzbeschreibung**
[A1RO]	Sonderbehandlung *Read-Only*-Nutzer
Status	Wurde von mir behoben. Alle Nutzer behalten dauerhaft ihren Jupiter-Ressourcen, egal ob sie Schreibrechte haben oder nicht. Dadurch konnte der Quellcode deutlich aufgeräumt werden.

ID	**Kurzbeschreibung**
[A1ArBr]	Sonderfälle Aktivitätsverarbeitung
Status	Von mir behoben. Ich konnte die Sonderfällen beheben, in denen Aktivitäten direkt im *Session Management* erzeugt wurden oder das *Activity Handling* komplett umgingen.

ID	**Kurzbeschreibung**
[A1ActPr]	*Activity Provider* Konsument und Produzent
Status	Offen. Die *Activity Provider* könnten auf *Activity Provider* und *Activity Consumer* aufgeteilt werden.

ID	**Kurzbeschreibung**
[A1PaKo]	Paketstruktur != Komponenten
Status	Offen. Ein paar Klassen, wie der *ActivitySequencer* wurden allerdings in den richtigen Ordner verschoben. Um das Problem zu lösen sollten zumindest die in der Architekturbeschreibung verwendeten Komponentennamen und die Paketnamen aneinander angepasst werden.

ID	**Kurzbeschreibung**
[A1Komp]	Keine komplette Komponentenaufteilung
Status	Offen. Die in der Architekturbeschreibung beschriebene Komponentenaufteilung könnte als Grundlage für eine korrekte Aufteilung genommen werden.

ID	**Kurzbeschreibung**
[A1Zirk]	Zirkuläre Abhängigkeiten *Session Management*
Status	Offen. Die Abhängigkeit der Komponenten sollte klar definiert werden, dafür können zum Beispiel *listener* verwendet werden.

ID	**Kurzbeschreibung**
[L1ThWa]	Benennung von Threads und Warteschlangen
Status	Offen. Die Threads und Warteschlangen sollten in einem einheitlichen Schema benannt werden.

ID	**Kurzbeschreibung**
[L1AQM]	*ActivityQueueManager*
Status	Wurde von einem anderen Entwickler entfernt. Aktivitäten mit falscher Sequenznummer werden nun bis zu einem Timeout gespeichert und anschließend wird der Nutzer aus der Sitzung entfernt.

ID	**Kurzbeschreibung**
[V1CS]	Client-Server-Architektur
Status	Wurde von mir behoben. Alle Aktivitäten werden nun über einen Zentralen Server geschickt. Dieser ist Teil des Host-Knotens. Dadurch wurde die verwendete Netzwerkarchitektur spezifiziert und eine globale Reihenfolge von Aktivitäten kann hergestellt werden.

ID	**Kurzbeschreibung**
[A2MeAk]	Trennung von MESSAGES und Aktivitäten
Status	Wurde von mir behoben. Alle Nachrichten, die zum Verändern einer Sitzung selbst verwendet werden, werden nun als MESSAGES versendet und Nachrichten, die innerhalb einer Sitzung verwendet werden, werden als Aktivitäten versendet, wodurch die Aufgaben dieser beiden Mechanismen klarer von einander abgegrenzt sind.

ID	**Kurzbeschreibung**
[A2Leave]	P2P für LEAVEMESSAGES
Status	Wurde von einem anderen Entwickler behoben. Der Host sendet jetzt *leaveMessages* und nicht mehr die Clients, wodurch die Client-Server-Architektur weiter vorangetrieben wurde.

ID	**Kurzbeschreibung**
[A2Kick]	Nutzer entfernen funktioniert nicht
Status	Wurde im Rahmen der Behebung von Problem [A2Leave] behoben.

ID	**Kurzbeschreibung**
[I2UseSt]	Spezifikation von Nutzerzuständen
Status	Wurde von mir behoben. Es gibt nun eine Reihe klar definierter Zustände, in denen sich ein Nutzer befinden kann. Mit jedem dieser Zustände sind eine Reihe Eigenschaften verbunden. Die Einhaltung dieser Eigenschaften wurde erfolgreich implementiert.

ID	**Kurzbeschreibung**
[I2Inko]	Inkonsistenzen nach *Project Negotiation*
Status	Wurde von mir behoben. Clients können nun problemlos weiterarbeiten, wenn ein neuer Nutzer hinzukommt, ohne dass Inkonsistenzen auftreten. Einzig das Blocken während der Host die Projekte verpackt verbleibt.

ID	**Kurzbeschreibung**
[I2AkPN]	Verwerfen von Aktivitäten
Status	Wurde im von mir im Rahmen von [I2UseSt] behoben. Während des Einladungsprozesses werden nun keine Aktivitäten mehr gesendet, die der Client nicht verarbeiten kann.

ID	**Kurzbeschreibung**
[I2Info]	Statusinformationen fehlen
Status	Wurde im von mir im Rahmen von [I2UseSt] behoben. Am Ende des Einladungsprozesses weiß der Client, von jedem Teilnehmer, welche Dateien dieser offen hat.

ID	**Kurzbeschreibung**
[I2Block]	Blockieren der Sitzung während der Einladung
Status	Offen. Ich habe mich dazu entschieden erstmal dafür zu sorgen, dass der Einladungsprozess mit der Blockierung korrekt funktioniert und die einzelnen Zustände gut definiert sind.

ID	**Kurzbeschreibung**
[W2WDog]	*Watchdog* funktioniert nicht
Status	Wurde von mir behoben. Der *Watchdog* ist nun in der Lage inkonsistente Zustände von geteilten Ressourcen zu erkennen und diese Inkonsistenzen auf Anfrage zu beheben.

ID	**Kurzbeschreibung**
[3CSPhy]	Physikalische Trennung der Serverkomponente
Status	Offen. Die Trennung sollte so vollzogen werden, dass der Server als eigenständiges Programm ausgelagert werden kann. Dabei sollte vor allem auch die Datenbasis getrennt werden.

ID	**Kurzbeschreibung**
[3Wrap]	*ITargetedWrapper*
Status	Behoben von mir. Der *ITargetedWrapper* wurde entfernt und alle Aktivitäten, die ihn verwendet hatten wurden umgebaut.

ID	**Kurzbeschreibung**
[3CliTar]	Empfänger für TargetedActivities von Clients festgelegt
Status	Offen. Man könnte eine Reihe von *Activity Router* auf dem Server anlegen, die ähnlich den *Activity Providers* für bestimmte Aktivitätstypen wissen, wem diese zugewiesen werden soll.

6 Dokumentation

In diesem Kapitel wird aufgezeigt, wie ich die gesammelten Erkenntnisse zusammengetragen habe, um zukünftigen Entwicklern den Einsteig in das Saros Projekt zu erleichtern. Dabei beschreibe ich zunächst in Abschnitt 6.1 nach welchen Kriterien ich die Dokumentation erstellt habe. Anschließend werden ich eine Evaluation dieser Dokumentation in Abschnitt 6.2 vorstellen und im letzten Abschnitt 6.3 werde ich Möglichkeiten aufzeigen wie diese Dokumentation in Zukunft gepflegt werden könnte. Die konkrete Dokumentation findet sich dann im Anhang (Abschnitt A).

6.1 Architektur Dokumentation

Nachdem ich im Laufe meiner Analyse einen tiefgehenden Einblick in die internen Abläufe von Saros erhalten habe, wollte ich einen Teil dieser Erkenntnisse auch an zukünftige Entwickler weitergeben. Dazu habe ich mir zunächst einmal überlegt, in welcher Form diese Dokumentation angelegt werden soll. Ich habe mich dafür entschieden die Dokumentation in Form einer Architekturbeschreibung auf die Homepage des Saros-Projekts zu übertragen.
Ich hatte bereits eine Architekturbeschreibung für meine Analysearbeit angelegt. Diese Architekturbeschreibung war allerdings viel zu detailliert. Darüber hinaus hatte ich sie nur als Grundlage für meine Analyse und Implementierungen verwendet, sodass ich die vorhandenen Architekturdiagramme noch einmal an den aktuellen Stand anpassen musste.
Bei der eigentlichen Dokumentation habe ich auf folgende Punkte geachtet:

- **Einsteigerfreundlich:** Die Dokumentation soll neuen Entwicklern helfen die verwendete Software-Architektur besser zu verstehen. Dadurch soll der Einstieg erleichtert werden. Darüber hinaus sollen so Architekturbrüche durch neue Entwickler verhindert werden, da Entwickler, die die Architektur verstehen, besser einschätzen können welche Implementierungen Architekturbrüche darstellen.

- **Verteilung:** Da die meisten in dieser Arbeit gefundenen Defekte auf die Kommunikation zwischen verschiedenen Eclipse-Instanzen zurückzuführen waren, wollte ich den Schwerpunkt dieser Dokumentation, neben den Komponentendiagrammen auf Verteilungsaspekte, wie die verwendete Netzwerkarchitektur legen.

- **Detailgrad:** Die Dokumentation muss so detailliert sein, dass die wichtigsten Abläufe und Komponenten beim ersten Lesen verstanden werden, darf aber gleichzeitig nicht so detailliert sein, dass sich Neueinsteiger überfordert fühlen. Gerade an diesem Punkt schien es den vorhandenen Dokumentationen zu mangeln, die entweder zu detailliert waren oder zu abstrakt, wie zum Beispiel die Komponentenübersicht mit im Grunde überflüssigen Komponente, wie Utility[18] und komplett ohne Schnittstellendefinitionen.

[18] Eine "Komponente", in der sich Funktionen befinden, die von allen Komponenten verwendet werden können

- **Aktualität:** Die Dokumentation muss den aktuellen Stand wiedergeben, damit sie einen Hilfe darstellen kann.
- **Konsistenz:** Die verwendeten Architekturdiagramme müssen in sich konsistent sein. Das heißt, dass verwendete Komponenten immer gleich dargestellt werden müssen. Das bezieht sich sowohl auf den Namen, als auch auf die farbliche Darstellung der Komponenten.

Bei der Erstellung der Architektur-Dokumentation habe ich versucht mich an diese Punkte zu halten. Dazu habe ich zunächst eine grobe Überblicksseite über die verwendete Netzwerkarchitektur angelegt und anschließend zwei Unterseiten erstellt, mit den Komponenten während der laufenden Sitzung und während des Einladungsprozesses. Für Entwickler, die weitergehendes Wissen in einem dieser Punkte erlangen möchten, habe ich darüber hinaus noch detailliertere Unterseiten angelegt, in denen die genauen Abläufe dargestellt werden.
Die Architekturbeschreibung wurde auf die Website übertragen. Eine PDF-Version dieser Dokumentation findet sich im Anhang A.
Zur Architekturbeschreibung wollte ich auch noch einen korrespondierende J-TourBus-Tour erstellen. Diese soll den Nutzer anhand der Architekturbeschreibung durch den Quellcode führen, damit die dargestellten Komponenten und Abläufe besser nachvollzogen werden können. Diese Tour ist zu diesem Zeitpunkt noch in Bearbeitung.

6.2 Bewertung der Dokumentation

Eine Bewertung dieser Dokumentation ließ sich unter den gegebenen Umständen nicht so einfach durchführen. Argumentativ kann man sagen, dass die hier erstellte Architekturdokumentation mit dem Quellcode abgestimmt ist und somit einen echten Mehrwert als Dokumentationserweiterung bringt. Darüber hinaus sind die vorhandenen Dokumentationen teilweise veraltet, während die erstellte Architekturbeschreibung aktuell ist.
Eine echte Bewertung der Verbesserung einer Architektur auf Produktebene lässt sich allerdings erst durchführen, wenn es auch eine Architektur gibt, gegen die man die vermeintliche Verbesserung prüfen kann. Eine Architektur kann allerdings erst dann als solche angesehen werden, wenn sie beschrieben wurde und ein Wissen darüber existiert. Eine Architektur, die niemand kennt ist nur eine große Ansammlung von Quellcode. Diese Arbeit legt also in gewisser Weise erst die Grundlage für zukünftige Bewertungen der Architektur auf Produktebene.
Für eine umfassende Bewertung der Architektur auf Prozessebene bräuchte man eine größere Menge an Neueinsteigern in das Projekt, die man befragen und deren Arbeit im Projekt man dahingehend untersuchen könnte, ob durch die Architekturbeschreibung weniger Architekturverletzungen auftreten. Wir haben in Saros allerdings keinen konstanten Strom an Neueinsteigern, sodass eine solche umfassende Untersuchung im Rahmen dieser Arbeit nicht durchführbar war. Dazu kommt außerdem, dass auch hier die Anzahl an Architekturverstößen erst gemessen werden kann, wenn es eine Beschreibung dieser Architektur gibt.
Allerdings habe ich die Dokumentation von den wenigen Neueinsteigern, die es im Projekt zum Zeitpunkt der Fertigstellung dieser Dokumentation gab, be-

werten lassen. Ich habe sie gebeten die Dokumentation durchzugehen und ihre ehrliche Bewertung dazu abzugeben. Hauptsächlich interessierte mich dabei, ob sie der Meinung waren, dass die Dokumentation ihnen dabei geholfen hat das System besser zu verstehen. Leider konnte ich nur eine Bewertung von zwei Personen erhalten. Dies waren einmal ein stellvertretender Student für ein Saros-Softwareprojekt mit drei Teilnehmern und ein Student, der seine Bachelorarbeit im Saros-Projekt anfertigte. Leider waren beide Personen nicht mehr ganz am Anfang der Entwicklung. Die Teilnehmer des Softwareprojekts hatten bereits eine Einführung in das Projekt erhalten und auch schon erste Implementierungen durchgeführt und der andere Student hatte sich auch bereits ein wenig mit Saros beschäftigt.
Die Dokumentation wurde dabei durchaus positiv bewertet. Alle Probanden fanden die Dokumentation nützlich und sagten, dass sie dadurch die Architektur von Saros besser verstanden hätten und der Detailgrad gut gewesen wäre.
Beide Befragte hatten darüber hinaus noch ein paar Anmerkungen dazu, wie die Dokumentation verbessert werden könnte. Neben einigen Detailpunkten war der größte Wunsch, dass Erwähnungen von Klassen und Schnittstellen in der Dokumentation mit einem Link auf die entsprechende Ressource versehen werden.
Diese Anmerkungen habe ich verwendet um die Dokumentation noch einmal in wenigen Punkten zu überarbeiten. Es gibt im Saros-Projekt einen Dienst, der automatisch aus den Javadocs[19] eine vollständige Saros-Dokumentation mit allen Klassen baut. Diese Klassen-Dokumentation habe ich dafür verwendet um dem Wunsch nach Verlinkung nachzukommen.
Leider konnte ich nicht überprüfen ob Entwickler durch das in der Architekturdokumentation vermittelte Wissen weniger Einarbeitungszeit benötigen und weniger Architekturverletzungen begehen als vorher. Für eine solche Analyse bräuchte man detaillierte Daten darüber, wie lange die Einarbeitungszeit und wie hoch die Zahl der Architekturbrüche vor der Erstellung dieser Architekturbeschreibung waren und diese müsste man mit den Daten neuer Entwickler vergleichen.

6.3 Dokumentationsverschleiß

Wichtiger als eine quantitative Bewertung des aktuellen Stands ist allerdings den Verschleiß der Dokumentation zu verhindern. Ein großes Problem einer jeden Architekturbeschreibung oder Dokumentation ist der Verschleiß. Im Laufe der Zeit entfernen sich Dokumentationen ohne Pflege immer weiter vom eigentlichen Projekt. Um diesen Verschleiß zu verlangsamen habe ich versucht die Dokumentation auf Architekturebene zu halten und nicht zu detailliert auf die einzelnen Klassen einzugehen. Auf diese Weise ist die konkrete Dokumentation nicht so sehr von der eigentlichen Implementierung abhängig.
Allerdings wird vermutlich auch diese Architekturdokumentation ohne Pflege im Laufe der Zeit veralten.
Um diesem Verschleiß entgegen zu wirken, könnte man jedes Aktivitäts- und

[19] Annotierte Dokumentationen im Quellcode

Sequenz-Diagramm einer bestimmten JTourBus-Tour zuordnen. Diese Touren könnte man mit Verweisen auf die entsprechenden Diagramme versehen und so die Pflege der Architektur an den entsprechenden Entwickler abgeben. Sobald dieser eine Änderungen macht, der die JTourBus-Tour invalidiert würde, muss er auch das Diagramm anpassen.
Für Komponentendiagramme könnte man sich eine halbautomatisierte Erstellung überlegen. Dazu müsste allerdings zunächst eine komplette Einteilung von Saros in Komponenten vorliegen und diese auch in der Paketstruktur wiederzufinden sein. Wenn eine vollständige Einteilung in Komponenten vorliegen würden, könnte man die Paketbeschreibungen verwenden um die statischen Komponentendiagramme mit Hilfe einer speziellen Beschreibung automatisch erstellen zu lassen.
Dazu wäre es zum Beispiel denkbar einer Komponente die Informationen zu geben in welchen Vorgängen sie vorkommt und mit welchen anderen Komponenten sie über welche Schnittstelle kommunizieren darf.
Für die Komponente *Session Management* könnte man sich folgende Informationen überlegen:

- Color: Green
- Diagramm: laufende Sitzung (siehe Abbildung 5)
  - Zugriff: Network Layer
    - Schnittstelle: ITransmitter
    - Schnittstelle: IReceiver
- Diagramm: Einladungsprozess (siehe Abbildung 9)
- Diagramm: SessionManagement (siehe Abbildung 6)
  - Unterkomponente: Activity Handling
  - Unterkomponente: Activity Sequencer
  - Unterkomponente: Registry
  - Unterkomponente: Concurrency Management

Für eine solche Beschreibung eignen sich XML-Dokumente, die in den jeweiligen Paketen anlegt werden könnten.
Die Aufteilung auf einzelne Diagramme hat dabei einen ganz einfachen Grund. Es gibt im Quellcode bereits eine Reihe von Annotationen, die Komponenten definieren (siehe Seite 19). Auch diese wurden automatisiert erstellt. Allerdings wurden einfach alle diese Annotationen, wie bereits erwähnt, genommen und in ein Diagramm verpackt. Dieses Diagramm war so unübersichtlich, dass niemand sinnvolle Informationen aus ihm ziehen konnte. Am Beispiel dieses Diagramms lässt sich leicht erkennen, wieso diese Informationen auf mehrere Diagramme aufgeteilt werden sollten.

7 Abschlussbetrachtung

In diesem Kapitel werde ich meine abschließenden Gedanken zu dieser Arbeit abgeben. Dazu werde ich zunächst in Abschnitt 7.1 den erreichten Stand mit den gesetzten Zielen vergleichen. Anschließend werde ich in Abschnitt 7.2 offene Punkte beschreiben, die als Anlass für weitere Arbeiten im Saros-Projekt dienen könnten.

7.1 Aktueller Stand und gesetzte Ziele

1. **Analyse:** Die Architektur von Saros soll mit Fokus auf Nebenläufigkeitsaspekte analysiert werden.

2. **Verbesserung:** Im Anschluss an diese Analyse sollen einige der gefundenen Probleme gelöst werden um so eine echte Verbesserung der Architektur zu erreichen.

3. **Dokumentation:** Darüber hinaus soll das gesammelte Wissen in geeigneter Weise zusammengetragen und dargestellt werden, um zukünftigen Entwicklern den Einstieg in das Projekt zu erleichtern.

Im Rahmen dieser Arbeit habe ich eine Architekturanalyse des Softwaresystems Saros durchgeführt. Dazu musste ich mir zunächst ein Vorgehen überlegen. Nach diesem Vorgehen habe ich dann die zwei wichtigen übergeordneten Szenarien **Verarbeiten von Aktivitäten** und **Verändern einer Sitzung** genau analysiert. Ich habe beschlossen es bei der Analyse dieser beiden Szenarien zu belassen. Der Grund dafür war, dass diese beiden Szenarien bereits einen Großteil der Aktionen in einer laufenden Sitzung ausmachen und ich meine Zeitplanung so einteilen musste, dass auch die anderen Zielsetzungen ausreichend Bearbeitungszeit haben. Im Rahmen dieser Analyse konnte ich 22 Probleme und Problemgruppen auf verschiedenen Ebenen aufdecken. Dazu gehörten Architekturverletzungen, sowie Probleme mit der Kommunikation zwischen des einzelnen Teilnehmern einer Sitzung. Das Ziel 1 wurde somit erreicht.

Nachdem ich die Probleme genau beschrieben hatte, habe ich damit begonnen einige davon zu lösen. Ich bin dabei so vorgegangen, dass ich versucht habe die wichtigsten Probleme zuerst zu behandeln um auf diese Weise eine echte Verbesserung der Saros-Software erreichen zu können.
Von den 22 Problemen konnte ich 11 lösen. Bei zwei dieser Problemen habe ich dabei wichtige Spezifikationen definiert:

- Nutzerzustände während des Einladungsprozesses

- Definition und Umsetzung einer einheitlichen Client-Server-Architektur

Außerdem waren zwei dieser gelösten Probleme echte, für die laufende Sitzung kritische Versagen:

- Inkonsistenzen während der Einladung

- Der fehlerhafte Watchdog

Durch die Definition dieser wichtigen Spezifikationen und dem Lösen kritischer Versagen, habe ich eine echte Verbesserung der Architektur und der Stabilität des Systems erreichen können. Ziel 2 wurde somit auch erfüllt.

Schlussendlich habe ich die Architekturbeschreibung so aufbereitet, dass sie neuen Entwicklern dazu dienen kann, das System besser zu verstehen. Dabei habe ich versucht einen angemessenen Anspruch für Neuentwickler sicherzustellen. Dazu habe ich in meiner Architekturbeschreibung zunächst einen allgemeinen Überblick über die Architektur gegeben und anschließend in Unterkapiteln detailliertere Informationen bereitgestellt.
Die Architekturbeschreibung wurde so gut wie möglich von Anfängern bewertet und schien hilfreich zu sein um die Architektur zu verstehen. Auch Ziel 3 habe ich somit erfüllt.

Zusammenfassend kann man sagen, dass ich die gesetzten Ziele erreicht habe und somit, zuversichtlich bin einen entscheidenden Beitrag dazu geleistet zu haben, das Softwaresystem Saros zu verbessern und zukünftigen Entwicklern den Einstieg zu erleichtern.

Sämtliche erstellte Diagramme, sowie sonstige im Rahmen dieser Arbeit erstellte Dateien können im Fachbereichs-SVN eingesehen werden. Dieses ist erreichbar unter `https://svn.imp.fu-berlin.de/agse/students/schlott/`. Im Hauptverzeichnis findet sich eine Datei namens README.txt, die einen Überblick über die dort gespeicherten Dokumente gibt.

7.2 Ausblick

Obwohl ich die gesetzten Ziele für mich zufriedenstellend erreichen konnte, gab es noch ein paar wichtige Probleme, deren Lösung mit dem Umfang dieser Arbeit nicht vereinbar waren. Diese Probleme sollten in nachfolgenden Arbeiten im Saros-Projekt bearbeitet werden. Die wichtigsten dieser Probleme sind meiner Meinung nach die Folgenden:

- **Physische Einteilung in Komponenten:** In meiner Arbeit habe ich eine Reihe von Komponenten beschrieben und deren Aufgaben klar definiert, sowie Abhängigkeiten zwischen den verschieden Komponenten beschrieben. Die einzelnen Komponenten sind im Quellcode allerdings nicht komplett gekapselt und die Abhängigkeiten zwischen den Komponenten werden nicht eingehalten. Darüber hinaus stimmt die Benennung von Paketen teilweise nicht mit den definierte Komponenten überein.

- **Physische Trennung von Client und Servers:** Die physische Abtrennung der Serverkomponente hat eine Reihe von Vorteilen. Siehe dazu Abschnitt 4.4 für eine genauere Liste. Hier soll nur gesagt werden, dass die physische Abtrennung des Servers die Synchronisation und das Verständnis an viele Stellen deutlich vereinfachen könnte.

A Architecture Documentation

(Hauptseite)

- **Network Architecture**

  During a *Running Session* the system is composed of multiple Saros-instances that work together in a *Shared Session*. To communicate with other participants and keeps the local copies of shared files consistent Saros uses XMPP-messages. Saros distributes these messages among the participants using a Client-Server-Architecture. The Server is located at the Hosts machine which means, that every participant first sends their messages to the Server and the Server distributes them.

Saros uses two kinds of XMPP-messages for different purposes:

- **MESSAGES:**

  Saros uses *MESSAGES* for changing the session itself. We call this the *Invitation Process*. This includes adding new Projects as well as adding or removing users to a running session. Different *MESSAGES* will be distinguished by their type (The *PackageExtension*).

  See Invitation Process for more informations.

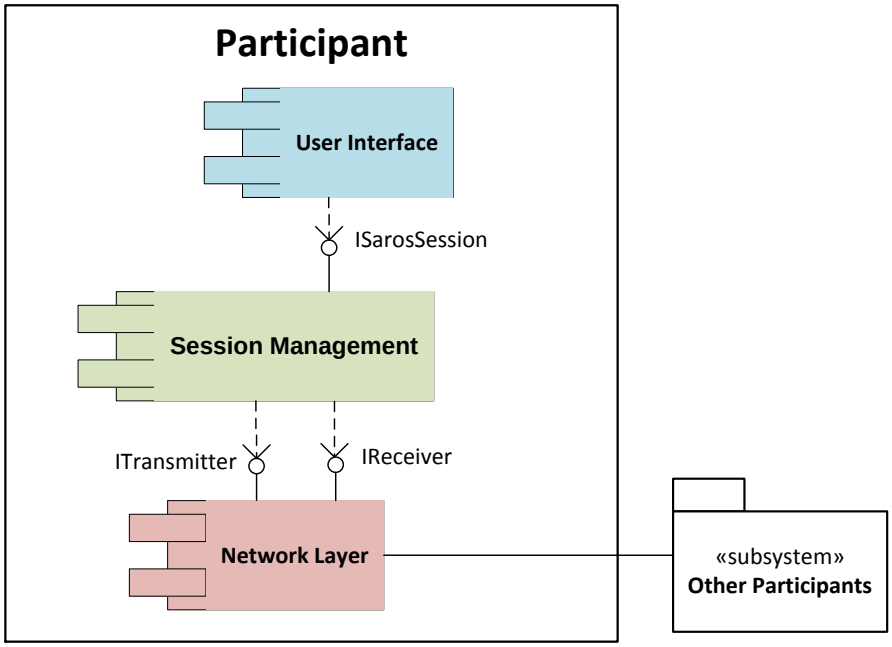
- **Activities:**

  Activities are messages with a special purpose. Activities are used in a running session and represent actions that a local user has performed and remote users need to know about. This includes for example modifying a shared file. Basically everything that doesn't change the session itself is transmitted using Activities.

  To learn more about communication in running session see Components Running Session.

A.1 Components running Session

Saros is composed of a number of functional components. The following figure shows the most important components in a running session.



- **User Interface**

  This component is responsible for managing all the visual elements in Saros, listening to changes in Eclipse and reacting to actions performed by the local user.

- **Session Management**

  This component is responsible for managing a Shared Session and keeping the shared projects in a consistent state across the local copies of all participants. It functions as the core component in a running session and directs communication between all other components.

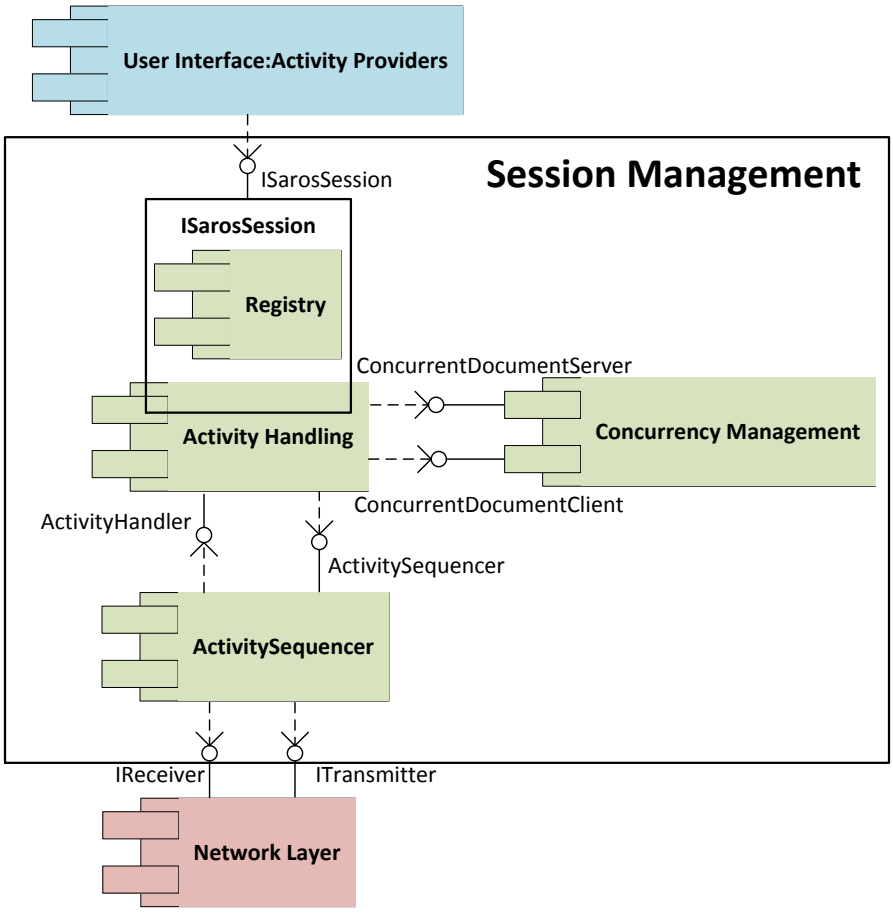
  In general this component takes input from the User Interface, transforms and directs it and afterwards passes the result to the Network Layer. See Session Management for a more detailed view at this component and Activity Sending. The main entrance-point of this component is the ISarosSession-Interface.

- **Network Layer**

  The Network Layer is responsible for communicating with other participants by sending and receiving Activities. The main entrance-points of this component are the Interfaces ITransmitter for sending messages and the IReceiver for receiving them.

A.1.1 Session Management

The Session Management is composed of a number of sub-components as can be seen in the following figure. For more informations about the actual Activity sending see Activity Sending.



- **Activity Providers/Registry**

  The Session Management internally has a Registry that manages registered Activity Providers.

Activity Providers are special part of the User Interface. They are the responsible for transforming actions of the local user into Activities that can then be sent to remote users. They are also responsible for executing remote Activities at the local Saros-instance. Every Activity Provider is responsible for a certain type of Activities. Whenever a registered Activity Provider creates a new Activity the Registry get informed about this and passes it to the Activity Handling.

The Registry also informs all registered Activity Providers whenever an incoming Activity has to be executed locally. Every Activity Provider is himself responsible for knowing which Activities he has to execute and which he has to ignore.

The Registry is implemented as a List in the ISarosSession. Activity Providers therefore have to register at the ISarosSession which than functions as the Registry.

- **Activity Handling**

  This is the main component of the Session Management. It basically functions as a coordinator between the other components. It gets new Activities from the Registry, calls the Concurrency Management if needed, determines receivers for this Activity, and sends the result to the Activity Sequencer or the other way around for incoming Activities.

  The main entrance points of the Activity Handling are the ISarosSession for local Activities and the ActivityHandler for incoming Activities.

- **Concurrency Management**

  The Concurrency Management is responsible for keeping concurrent documents consistent. Saros uses the Jupiter-algorithm to accomplish this. This includes the transformation of some Activities, most importantly the TextEditActivities that represent modifications of shared documents.

  The main entrance points of this component are the ConcurrentDocumentServer for the Server and the ConcurrentDocumentClient for the Clients.

- **Activity Sequencer**

  The Activity Sequencer is responsible for ordering outgoing and incoming Activities to ensure, that no Activities got lost. The Activity Sequencer is registered as a PacketListener at the Network Layer and informs the Activity Handling about incoming Activities as well as forwarding outgoing Activities to the Network Layer.

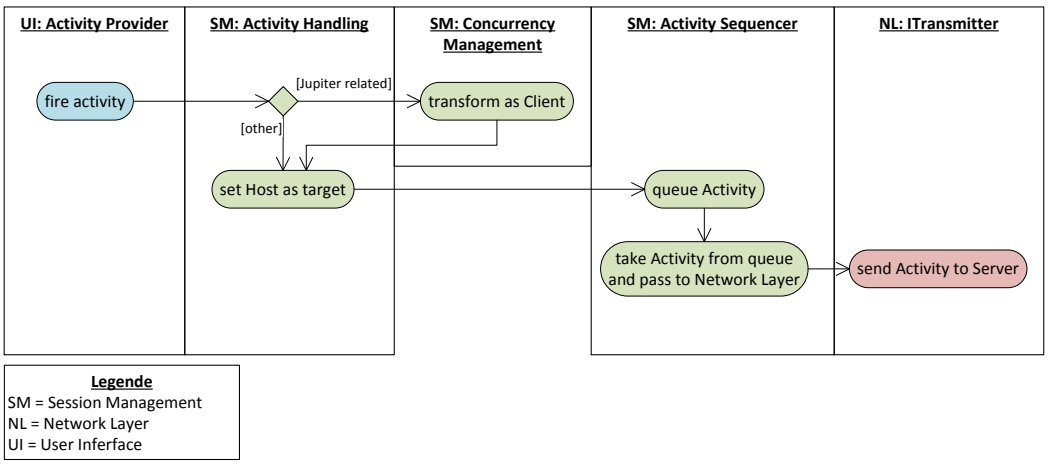
  The main entrance point of this component is the ActivitySequencer.

A.1.2 Activity Sending

Activities are the most important way of communication to keep a Shared Session consistent.

- **Client Sending**

  The following Diagram shows how Activities are sent by clients in Saros. Notice that the Host also acts as a Client for locally created Activities.

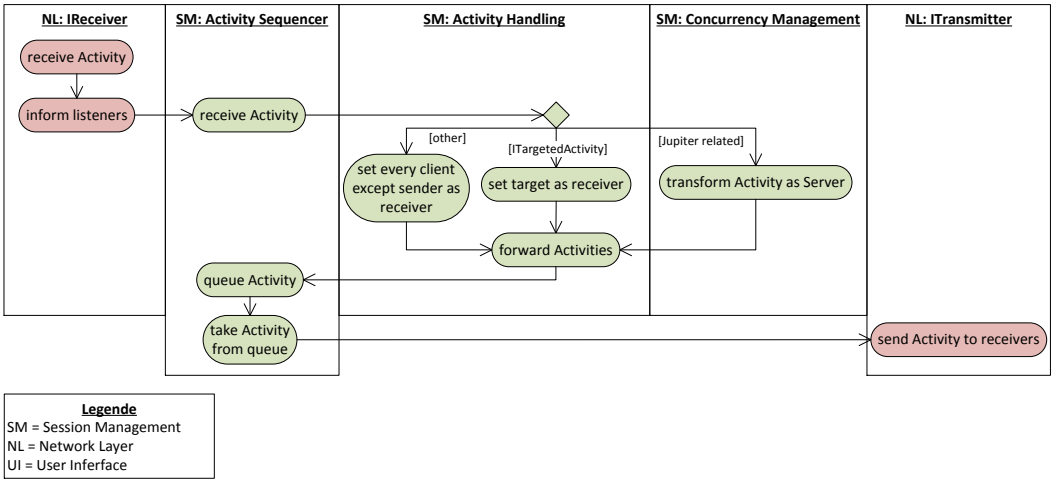


  New Activities are created by the Activity Providers. They take actions of the local user and transform them to Activities. As Activity Providers are registered at the Session Management the Activity Handling gets informed about the new Activity and sets the Host as target for this Activity. If the Activity is Jupiter related it will first be passed to the Concurrency Management where it will be transformed.

  Afterward the Activity will be passed to the Activity Sequencer where it will be queued and also gets a timestamp to ensure a global ordering of Activities. The OutgoingActivityQueue is polled frequently and the Activities will then be passed to the ITransmitter where they are transformed into XMPP-messages and transmitted to the Host.

- **Activity Server**

  All Clients send their Activities to the Server located at the Host. The following diagram shows how the Server handles and directs incoming Activities.



Every participant receives Activities via the IReceiver. The IReceiver allows other components to register packetListeners that will be informed when Messages with a specific PackageExtension arrive. The Activity Sequencer also registers such a listener and will be informed when new Activities arrive. The Activity Sequencer checks that the Activity has the right sequence-number and if so passes it to the Activity Handling.

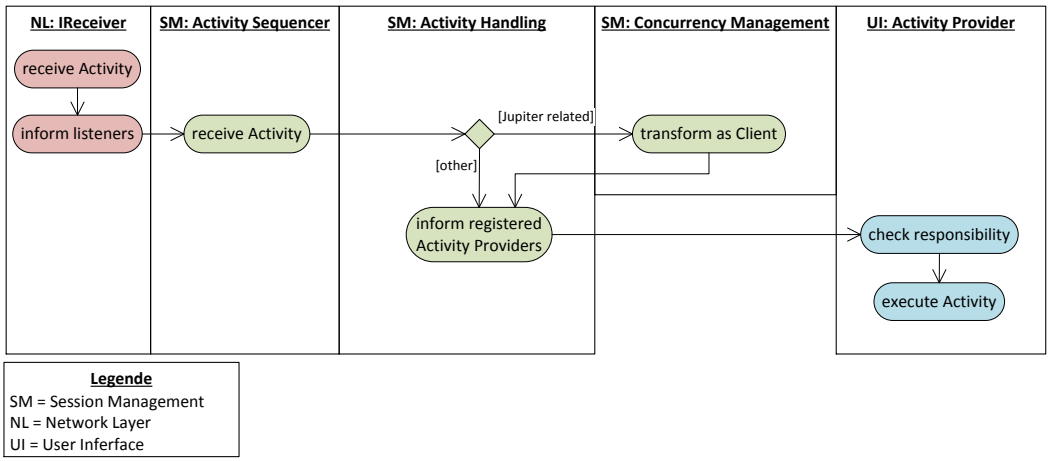
The Activity Handling distinguishes between three different types of Activities:

1. Jupiter related Activity: Activities that are related to the Jupiter-Algorithm will be passed to the Concurrency Management where they will be transformed into an Activity with a different timestamps for each participant. Every participants gets a separate Activity and is set as its receiver.
2. ITargetedActivity: When processing an ITargetedActivity the receiver will be set as the target of the Activity.
3. Others: The receivers of all other Activities are all participants except the creator of the Activity.

After setting the right receivers the Activities will again be send to the ITransmitter, via the Activity Sequencer, and transmitted to the intended receivers.

- **Client Receiving**

  After the Server has distributed the Activities all intended Clients receive them. Again, remember that the Host also acts as a Client despite housing the Server. The following diagram shows how the receiving works.

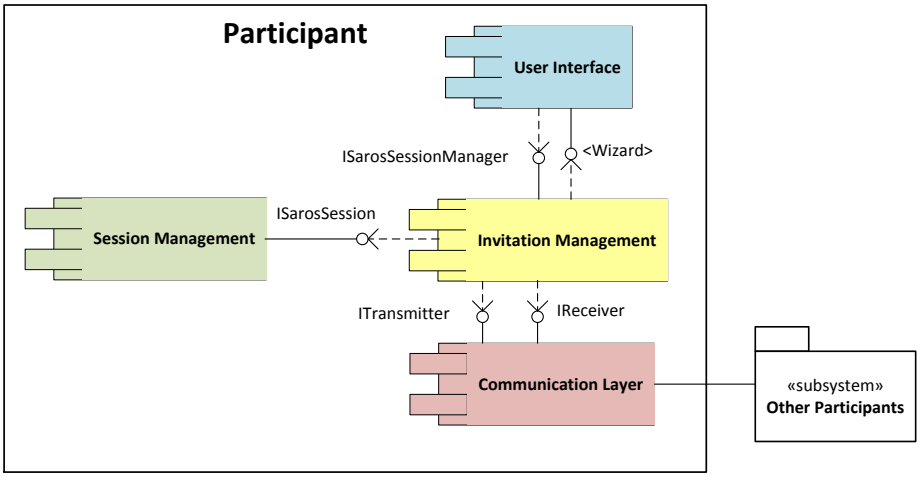


  When a new Activity arrives the Activity Sequencer will be informed about this and passes the Activity to the Activity Handling.

  The Activity Handling passes incoming Activities to the Concurrency Control if needed. For example a JupiterActivity will be transformed back into a TextEditActivity. Afterwards the Activitiy Handling informs all registered Activity Providers about the new Activity. Every Activity Provider checks himself if he is responsible for this type of Activity. If responsible the Activity Provider transforms the Activity back into an action, that can be applied to the local Saros-Instance and executes this action.

A.2 Invitation Process

This Page shows which components take part in the Invitation Process (Modifing a Shared Session itself).



- **User Interface**

  This component is responsible for managing all the visual elements in Saros, listening to changes in Eclipse and reacting to actions performed by the local user. In this context, the User Interface allows the Host to add users or projects and displays the needed invitation-dialogs at both sides.

- **Network Layer**

  This componentis responsible for communicating with other participants by sending and receiving MESSAGES. The main entrance-points of this component are the Interfaces ITransmitter for sending messages and the IReceiver for receiving them.

- **Invitation Management**

  The Invitation Management is responsible for starting a Shared Session or adding new users or projects to a running session. The Invitation Process itself is divided into the two subcomponents Session Negotiation and Project Negotiation. Both are started by the Invitation Management and are only active while the respective Invitation Process is running.

  – Session Negotiation: The Session Negotiation is responsible for adding a new user to the session.

  - Project Negotiation: The Project Negotiation is responsible for sharing projects with the participants and adding new projects to a running session.

A Negotiation is always performed between the Host and a Client to ensure consistency in a running session. Saros supports the invitation of multiple users at once. In this case multiple Negotiation Processes are started concurrently, one for each invitee.

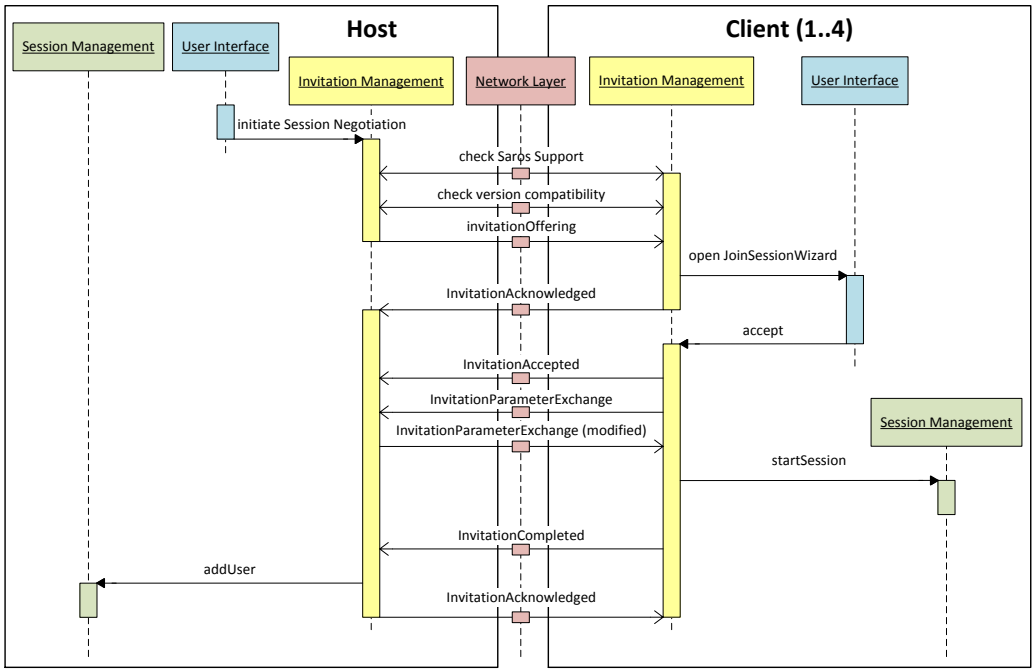
- **Session Management**

  The Session Management is responsible for keeping a running session consistent. For more informations see Session Management. After the Invitation Management has established or changed a session the Session Management is informed about the changes so that new users and projects will be minded by Saros going forward.

  During an Invitation a running session goes on, so that other participants aren't forced to wait until a new user joined for Example. This leads to a couple of rules for joining users that have to be met. See User States for more informations.

A.2.1 Session Negotiation

The following diagram shows how the Session Negotiation works and which MESSAGES are sent between the Host and the invitee.



The Session Negotiation is started, when the Host invites a user to a session. The User Interface registers this action and starts the Negotiation Process. The Invitation Management of the Host first asks the Client if he has a Saros-instance running and if his version is compatible with the version of the Host. Afterwards he sends an invitationOffering to the Client.

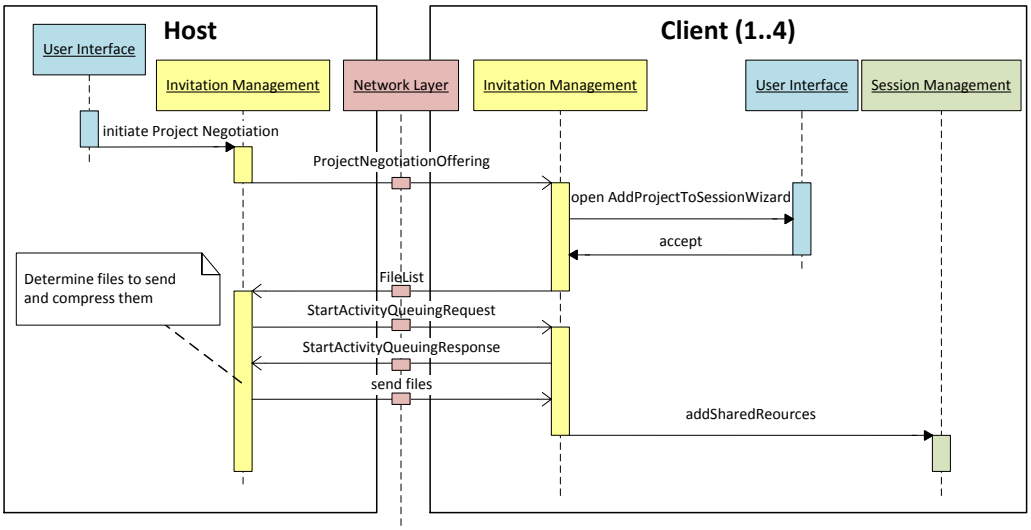
All MESSAGES are send and received by the Network Layer which communicates with the Invitation Management. The Invitation Management informs the User Interface which then opens a wizard, informing the Client about the invitation. At the same time an acknowledgment is sent back to the Host informing him, that the Client has received the invitation.

Once the Client accepted the invitation he first sends an invitationAccepted and afterwards tells the host which session parameters he would prefer. These parameters includes for example the desired color.

The Host evaluates this preferences and sends the actual session parameters back to the Client. With these session parameters the Client then creates a new session and sends the Host a final acknowledgment, to which the Host informs his Session Management that a new user has joined the session.

A.2.2 Project Negotiation

The following diagram shows how the Project Negotiation works and what MESSAGES get sent between the Host and the invitee.



The Project Negotiation starts with the Host sending a ProjectNegotiationOffering to the Client. This Message contains informations that the Client needs, for example which projects will be shared and the ID of the session.

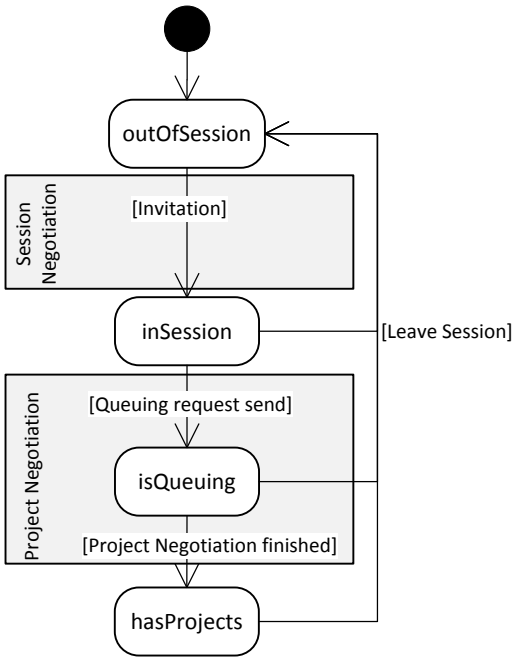
The Invitation Management of the Client then informs the User Interface which opens a wizard, informing the Client about the Project Negotiation and asking him to chose a location for the projects that will be transmitted.

After the Client accepted the Project Negotiation he compares the List of files he got by the ProjectNegotiationOffering with his local files and sends a FileList containing all the files he is missing. The Host than sends a StartActivityQueuingRequest to signal the Client that he should start queuing incoming Activities. After starting queuing the Client acknowledges the queuing request.

The Host then determines which projects he has to send and compresses them into a zip-archive. While the projects get compressed the session is blocked for all users, to ensure that no file gets modified while being compressed. Afterwards the Host sends the archive containing all missing files back to the Client and the Client unpacks them. After the files have been added to the Clients workspace he informs his Session Management about the new files to be managed.

A.2.3 User States

During the Invitation Process every user traverses a few states before he is a full member of the session. The following diagram shows these transitions.



Users that are currently not in a session are in the initial state outOfSession. In this state they don't receive Activities from other users. The state changes when the user gets invited to a session. This is done by the Session Negotiation.

Once the user finished the Session Negotiation he enters the state inSession. This state signals that the user is part of the running session but doesn't have the shared projects yet. While in this state the user can see all other participants and can be seen by them. He is also able to chat with other participants. He can accept some Activities but can't process IResourceActivities as he doesn't have the projects yet. The user leaves this state during a certain point in the following Project Negotiation.

Once the Host has sent the startQueuing-MESSAGE the new user starts to queue any IResourceActivity. The user is now in the state isQueuing. As soon as the user entered this state he receives IResourceActivities from all participants but instead of executing them he just puts them into a queue.

The queued Activities are executed as soon as the Host has sent all missing projects and the new user finished the Project Negotiation.

From that moment on the new user is a full member of the session and has reached the final state hasProjects.

A user can return from every state to the initial state outOfSession by leaving the session or canceling the Session or Project Negotiation.

Literatur

[BA06] Mordechai Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall Europe, 2 edition, 2006.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. The Agile Manifesto. Technical report, The Agile Alliance, 2001.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[Bel11] Wjatscheslaw Belousow. Verbesserung der architektur der dpp-software saros durch einführung einer dokumentierten modulsicht. Master's thesis, Freie Universität Berlin, 2011.

[Ecl] Website. Erreichbar online unter http://www.eclipse.org/, Stand: 15.09.13.

[GPB+06] Brian Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[Joh11] Meike Johannsen. Verbesserung und pflege der dokumentation der dpp-software saros. Master's thesis, Freie Universität Berlin, 2011.

[JTo] Website. Erreichbar online unter http://www.inf.fu-berlin.de/w/SE/JTourBus, Stand: 16.09.13.

[Kru95] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[Lea99] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2 edition, 1999.

[LPS03] Grace A. Lewis, Daniel Plakosh, and Robert C. Seacord. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley, 2003.

[pic] Website. Erreichbar online unter http://picocontainer.codehaus.org/, Stand: 15.09.13.

[RW05] Nick Rozanski and Eoin Woods. *Software Systems Architecture - Working with Stakeholders Using Viewpoints and Perspectives*. Addison-Wesley, 2005.

[Sar] Website. Erreichbar online unter http://www.saros-project.org/, Stand: 15.09.13.

[SOBS10] Stephan Salinger, Christopher Oezbek, Karl Beecher, and Julia Schenk. Saros: an eclipse plug-in for distributed party programming. In *Proceedings of the 2010 ICSE Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '10, pages 48–55, New York, NY, USA, 2010. ACM.

[Sta13] Sebastian Starroske. Improving the reliability of saros using root cause analysis. Master's thesis, Freie Universität Berlin, 2013.

[Sto06] Margaret-Anne Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[TS06] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems. Principles and Paradigms.* Pearson Prentice Hall, 2 edition, 2006.