Masterarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Entwicklung und Evaluation eines unabhängigen Sitzungsservers für das Saros-Projekt

Denis Washington
Matrikelnummer: 4740587
denis@denisw.de

Betreuer: Franz Zieris
Eingereicht bei: Prof. Lutz Prechelt, Prof. Adrian Paschke

Berlin, 04.01.2016

Zusammenfassung

Saros, ein Werkzeug zur verteilten Echtzeitkollaboration in der Softwareentwicklung, fußt traditionell auf einem *host-basierten Sitzungsmodell*, das die ständige Anwesenheit eines festgelegten Teilnehmers der Sitzung (dem *Host*) erfordert. Diese Arbeit setzt zuvor begonnene Bemühungen fort, einen eigenständigen *Sitzungsserver* für Saros zu entwickeln, der die Rolle des Hosts übernehmen kann und so langlebige Sitzungen ermöglicht, die jeder Teilnehmer beliebig verlassen und auch wiederaufnehmen kann. Außerdem findet eine erste Analyse des entwickelten Servers und des darunterliegenden Saros-Kerns hinsichtlich möglicher Ressourcenlecks statt, die die Langlebigkeit solcher Sitzungen gefährden könnten.

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.

04.01.2016

Denis Washington

Inhaltsverzeichnis

1 Einleitung

Saros ist ein Werkzeug zur verteilten Echtzeitkollaboration zwischen räumlich getrennten Softwareentwicklern. Unter anderem ermöglicht es verteilten Softwareteams, Techniken wie Paar- und Gruppenprogrammierung (*distributed pair programming* bzw. *distributed party programming*) oder interaktive Codedurchsichten durchzuführen, die sonst die gemeinsame Anwesenheit am selben Ort erfordern würden [SBS10]. Bis zu fünf Personen können zu einer Saros-Sitzung zusammenfinden und an geteilten Softwareprojekten arbeiten, von denen jeder Sitzungsteilnehmer automatisch eine lokale Kopie erhält. Führt einer der Teilnehmer eine Änderung am Code oder anderen Dateien des Projekts durch, wird diese unmittelbar an alle anderen Teilnehmer übertragen und in Echtzeit angewendet.

In seiner ursprünglichen und derzeit ausgereiftesten Form ist Saros ein Plugin für die Entwicklungsumgebung Eclipse[1], dessen erste Version durch Riad Djemili im Rahmen seiner Diplomarbeit entstand [Dje06]. Diese Variante wird intern als *Saros/E* bezeichnet. Inzwischen ist auch eine Version des Plugins für IntelliJ IDEA[2] (*Saros/I*) in Entwicklung[3].

2013 begann Nils Bussas mit der Entwicklung eines Sitzungsservers für Saros [Bus14]. Dieser soll die Koordinationsaufgaben des so genannten *Hosts* einer Saros-Sitzung übernehmen können; diese Rolle erfüllt traditionall einer der Sitzungsteilnehmer, was allerdings Probleme mit sich bringt (siehe nächsten Abschnitt). Bussas konnte einen ersten Prototyp des Servers entwickeln, der bereits den Aufbau serverf-basierter Sitzungen, allerdings nicht das Teilen von Projekten darin erlaubte, sodass er sich noch nicht praktisch einsetzen ließ. Außerdem wurde er nicht als eigenständiges Programm, sondern als Erweiterung von Saros/E geschrieben, benötigte also eine vollwertige Instanz der Eclipse-Entwicklungsumgebung zur Ausführung.

Ziel dieser Arbeit war, die von Bussas begonnene Entwicklung fortzuführen und die erste Version eines eigenständigen, praktisch nutzbaren Saros-Sitzungsservers zu entwickeln. Die folgenden Abschnitte erläutern Hintergrund, Motivation, und konkrete Teilgebiete der Arbeit sowie die Struktur dieser Ausarbeitung.

[1] http://eclipse.org
[2] http://www.jetbrains.com/idea/
[3] http://www.saros-project.org/saros-for-intellij

1.1 Hintergrund und Motivation

Sitzungen sind in Saros traditionell **host-basiert**. Sobald ein Nutzer von Saros/E oder Saros/I eine neue Sitzung mit anderen Nutzern aus seiner Kontaktliste initiiert, wird er automatisch zum **Host** dieser Sitzung. Als solcher unterscheidet er sich in mehrerer Hinsicht von den anderen Sitzungsteilnehmern:

- Er darf als einziger Projekte teilen, also zum Teil der Sitzung machen. Außerdem kann nur er weitere Nutzer zur Sitzung einladen.
- Er fungiert als Knotenpunkt für alle Kommunikation innerhalb der Sitzung. Jede relevante Aktion eines Teilnehmers sendet dieser zunächst an den Host, der sie lokal anwendet und wenn nötig (ggf. in angepasster Form) an die anderen Teilnehmer weiterleitet. Dem Host kommt also die Rolle des Sitzungskoordinators zu.
- Sein aktueller lokaler Sitzungszustand gilt immer als der "wahre". Dies macht es unter anderem einfach, fehlerhaft auftretende Inkonsistenzen zwischen den Projektkopien der Teilnehmer aufzulösen (siehe Abschnitt 2.1.5).

Da dem Host eine so zentrale Rolle zukommt, kann seine Sitzung nicht ohne ihn existieren. Verlässt er sie – z.B. weil er Feierabend hat, oder unfreiwillig als Resultat eines Konnektivitätsproblems – so endet die Sitzung zwangsweise auch für alle anderen Teilnehmer, die dann zum Weiterarbeiten eine neue Sitzung mit einem anderen Host aufbauen müssen. Das ist nicht nur unpraktisch, sondern führt auch zum Verlust von Sitzungsinformationen – z.B. die Zuordnungen von Codestellen zu den Teilnehmern, die sie während der Sitzung verfasst haben (in Saros/E als farbliche Markierungen angezeigt).

Ein weiterer, von Bussas beschriebener Nachteil des Ansatzes ist die mangelnde Unterstützung für zeitlich versetzte Projektsynchronisation, da Saros aufgrund seiner Echtzeit-Natur die gleichzeitige Anwesenheit aller Sitzungsteilnehmer erfordert [Bus14, S. 4]. Verlässt eine Nutzerin Alice vorzeitig eine gemeinsame Sitzung mit einem anderen Nutzer Bob – z.B., weil sie früher Feierabend hat – verpasst sie alle Codeänderungen, die Bob danach an den vorher geteilten Projekten durchführt. Ist Bob am nächsten Tag nicht verfügbar, kann Alice ihre Arbeit nicht mit den aktuellen Versionen der Projekte fortsetzen, sofern Bob diese nicht über ein anderes Medium (z.B. E-Mail oder Versionskontrolle) an Alice weitergereicht hat. Eine zeit-

versetzte oder nur teilweise überlappende Zusammenarbeit an einem Projekt ist also mit dem traditionellen Saros-Sitzungsmodell nicht möglich.

1.1.1 Erster Server-Prototyp (Bussas 2014)

Aufgrund der oben genannten Probleme beschäftigte sich Nils Bussas in seiner Bachelorarbeit damit, den ersten Prototyp eines **Sitzungsservers** für Saros zu entwickeln [Bus14]. Ein solcher könnte in einer Sitzung die Rolle des Hosts übernehmen, sodass jeder andere Teilnehmer die Sitzung ohne Konsequenzen verlassen kann. Erlaubt man es zusätzlich, so einer **serverbasierten** bzw. **Serversitzung**[4] später auch wieder beizutreten, wären sehr langlebige Sitzungen möglich, zu denen ein Nutzer nach beliebig langer Abwesenheit zurückkehren könnte. Hätten Alice und Bob also in dem vorher beschriebenen Szenario in einer Serversitzung zusammengearbeitet, wären Bobs Änderungen nach Alices Verlassen der Sitzung immer noch auf den Sitzungsserver synchronisiert worden, der diese dann bei Wiedereintritt in die Sitzung an Alice weitergegeben hätte.

Um den Entwicklungsaufwand zu minimieren, implementierte Bussas seinen Server-Prototypen als Teil des Saros-Eclipse-Plugins. Wird er in den Plugin-Einstellungen aktiviert, akzeptiert er so genannte *Join Session Requests*, mit dem ein Nutzer den Server um eine Einladung zu einer Serversitzung bitten kann (siehe Abschnitt 5.1). Auf diese Weise können mehrere Nutzer einer gemeinsamen Serversitzung beitreten.

Aus Zeitgründen gelang es Bussas jedoch nicht, eine ähnliche Erweiterung des Saros-Protokolls für das Teilen von Projekten zu implementieren. Eine solche ist jedoch notwendig, da ja nur der Host der Sitzung – in diesem Fall also der Server – zum Teilen von Projekten berechtigt ist. Infolge dessen ermöglicht der Bussas-Server-Prototyp nur "leere" Sitzungen ohne geteilte Projekte.

Zwei weitere Probleme ergeben sich aus der Einbettung des Prototyps in Saros/E. Zum einen erschwert die Abhängigkeit von einer interaktiven grafischen Anwendung wie Eclipse die automatisierte Auslieferung, Konfiguration und Wartung des Servers, die man sich im Produktiveinsatz wünschen würde. Zum anderen ist zu erwarten, dass die Notwendigkeit einer vollwertigen Eclipse-Instanz zu einem unnötig hohen Ressourcenverbrauch führt, da der Großteil der Eclipse-Plattform (u.a. die grafische Oberfläche) für einen Saros-Sitzungsserver nicht benötigt wird.

[4] Obwohl auch eine Serversitzung formal über einen Host (nämlich den Server) verfügt, wird der Begriff "host-basiert" in dieser Arbeit nur für Sitzungen mit einem menschlichen Host verwendet.

1.1.2 Der Saros-Kern (u.a. Lasarzik 2015)

Eine Lösung für das letztgenannte Problem bietet das Konzept des **Saros-Kerns** (*Saros Core* im Code). Die Idee hierhinter ist, einen möglichst großen Teil der Saros-Implementation von Abhängigkeiten zu Eclipse bzw. IntelliJ zu befreien und in ein separates Modul zu verschieben, sodass er in verschiedenen Umgebungen wiederverwendet werden kann. In den Plugins muss dann idealerweise nur der für die Entwicklungsumgebung spezifische Code implementiert werden.

Der Saros-Kern entstand als Aufräumarbeit nach der Entwicklung der initialen Version von Saros/I, dem Saros-IntelliJ-Plugin. Diese enthielt Kopien großer Teile von Saros/E, die wo nötig für den IntelliJ-Kontext angepasst wurden. Der resultierende hohe Grad an Codeduplikation erschwerte die parallele Entwicklung der beiden Saros-Varianten. Aus diesem Grund wurde der Kern ins Leben gerufen und immer mehr duplizierter Code dort vereint, unter anderem im Rahmen der Bachelorarbeit von Arndt Lasarzik [Las15].

Durch diese Anstrengungen wurde ein signifikanter Teil der Saros-Kernlogik, unter anderem im Bereich der Netzwerkkommunikation und Projektsynchronisation, wiederverwendbar gemacht. Einige essentielle Komponenten – unter anderem Großteile der Logik für die Sitzungsverwaltung – waren aber immer noch dupliziert. Diese Komponenten hätten also für die Entwicklung einer neuen Saros-Variante – beispielsweise einem eigenständigen Sitzungsserver – erneut kopiert und angepasst werden müssen.

1.2 Zielsetzung

Ziel dieser Arbeit war, auf Basis des Saros-Kerns und der Vorarbeit von Bussas einen eigenständigen, von Eclipse unabhängigen Saros-Sitzungsserver zu entwickeln, der bereits praktisch eingesetzt werden kann. Dieser sollte außerdem bezüglich seiner Eignung für den Langzeitbetrieb unterzogen werden, da dieser Aspekt im Saros-Kontext bisher wenig Beachtung fand.

Insgesamt wurden die folgenden Teilziele festgelegt:

1. Der Entwurf und die Implementation eines Mechanismus, der es den Teilnehmern einer Serversitzung erlaubt, Projekte zu teilen. In Zusammenarbeit mit Ute Neise [Nei] wurde hierfür ein allgemeines Protokoll zum Projektteilen durch Nicht-Host-Teilnehmer (*Non-Host Project Sharing*) entwickelt, das auch in klassischen host-basierten Sitzungen verwendet werden kann.

2. Die *Vorbereitung des Saros-Kerns* für die Entwicklung eines unabhängigen Sitzungsservers. Dies umfasst die Identifikation von essenziellem Nicht-Kern-Code, der für den Sitzungsserver notwendig ist; die Befreiung dieses Codes von Eclipse- bzw. IntelliJ-spezifischen Abhängigkeiten; sowie die Überführung des Codes in den Kern.

3. Die *Implementation des unabhängigen Sitzungsservers* auf Basis des erweiterten Saros-Kerns.

4. Eine *Evaluation des Sitzungsservers* (und des Saros-Kerns) hinsichtlich der Qualität seiner Verwaltung von Systemressourcen wie Hauptspeicher und offenen Dateien. Das Augenmerk lag hierbei auf der Suche nach möglichen Ressourcenlecks, die den Langzeitbetrieb des Servers gefährden könnten.

1.3 Aufbau dieser Arbeit

Im nachfolgenden Kapitel 2 werden zunächst die Grundzüge des Kommunikationsprotokolls und Entwicklungsprozesses von Saros erläutert, auf die in den nachfolgenden Kapiteln Bezug genommen wird. Die nachfolgenden vier Kapitel sind jeweils einem der vier genannten Teilziele dieser Arbeit gewidmet:

- Kapitel 3 stellt den zusammen mit Ute Neise implementierten Mechanismus zum Teilen von Projekten als Nicht-Host vor. Hierzu gehört eine Erläuterung des Saros-Projektaustauschprotokolls (der *Project Negotiation*) sowie ein Einblick in den Entwurfsprozess und die Implementation.

- Kapitel 4 beschreibt Ergänzung des Saros-Kerns für den Sitzungsserver. Es erläutert die eingesetzte Vorgehensweise und illustriert sie anhand einiger der vorgenommenen Refaktorisierungen.

- Kapitel 5 dreht sich um die Implementation des eigentlichen Sitzungsservers auf Basis des Kerns. Es werden die zentralen Entwurfsentscheidungen, einige Aspekte der sowie verbleibende Einschränkungen und Probleme erläutert.

- In Kapitel 6 werden Vorgehen, Beobachtungen und Ergebnisse der Evaluation des Saros-Servers vorgestellt.

Abschließend gibt Kapitel 7 einen Überblick über die Ergebnisse dieser Arbeit und schlägt eine Weiterentwicklung des Saros-Serverkonzepts vor.

2 Grundlagen

Dieses Kapitel führt mehrere zentrale Saros-Begrifflichkeiten ein, die für das Verständnis dieser Arbeit von Bedeutung sind. Hierbei wurden an einigen Stellen Vereinfachungen getroffen, an denen die Details für diese Arbeit nicht interessant sind.

2.1 Das Saros-Kommunikationsprotokoll

Die folgenden Abschnitte bieten einen Überblick über die Art, wie Saros-Instanzen miteinander kommunizieren. Der für diese Arbeit besonders relevante Prozess des Projektaustausches (*Project Negotiaton*) wird detaillierter in Abschnitt 3.1 erläutert.

2.1.1 XMPP

Saros nutzt zur Kommunikation **XMPP**, das **Extensible Messaging and Presence Protocol**[5]. Dieser offene Standard, der ursprünglich unter dem Namen *Jabber* als Instant-Messaging-Protokoll entworfen wurde, ermöglicht den Austausch beliebiger XML-formatierter Nachrichten zwischen zwei oder mehr Teilnehmern. Ähnlich wie E-Mail folgt XMPP einer *förderierten* (*"federated"*) Client-Server-Architektur: es erlaubt Nutzern verschiedener, unabhängig betriebener XMPP-Server, untereinander zu kommunizieren, sofern die Server öffentlich oder untereinander erreichbar sind.

Zur Adressierung von Nachrichten nutzt XMPP – ebenfalls wie E-Mail – eine Kombination aus Nutzernamen und dem Domainnamen zur Identifikation von Accounts (z.B. `denisw@jabber.org`), die als **JID** (*Jabber Identifier*) bezeichnet wird. Zur Unterscheidung mehrerer gleichzeitig verbundener Klienten für denselben Account existiert auch ein erweitertes JID-Format, das einen zusätzlichen *Ressourcennamen* enthält (z.B. `denisw@jabber.org/Saros`).

Neben dem reinen Nachrichtenaustausch bietet XMPP noch weitere Funktionen. Hierzu gehörten unter die Unterstützung für Kontaktlisten (*Roster*) und Onlinestatus-Benachrichtungen (*Presence*), die für die Saros-Kontaktliste und deren Verfügbarkeitsanzeige verwendet werden. Außerdem bietet XMPP ein *"Service Discovery"*-Protokoll, mit dem ein XMPP-Server oder -Klient nach den von ihm unterstützten Funktionen gefragt werden kann; auf diese Weise kann Saros erkennen, ob ein verfügbarer Kontakt auch mit Saros oder aber einem anderen XMPP-Klienten (z.B. einem Chatprogramm) verbunden ist.

[5] http://xmpp.org/

2.1.2 Session und Project Negotiation

Das Saros-Kommunikationsprotokoll gliedert den Aufbau einer vollständigen Sitzung konzeptuell in zwei Phasen, die auf Protokollebene jeweils durch einen speziellen Nachrichtenaustausch (einer **Negotiation**) implementiert sind.

Möchte ein Nutzer (der zukünftige Host) eine Sitzung beginnen, intiiert er zunächst mit allen gewünschten Teilnehmern eine so genannte **Session Negotiation**, deren initiale Nachricht einer Einladung in die Sitzung entspricht (und auf Seite des Empfängers als solche angezeigt wird). Nimmt der Empfänger die Einladung an, gilt der Nutzer als Teilnehmer der Sitzung; ab diesem Zeitpunkt erhält er unter anderem die vom den anderen Teilnehmern generierten *Aktivitäten* (siehe nächsten Abschnitt).

Das Teilen von Projekten mit dem Teilnehmer geschieht anschließend durch eine so genannte **Project Negotiation**. Der Host schickt ihm hierbei zunächst verschiedene Metadaten über die angebotenen Projekte[6] (z.B. die Projektnamen). Der Teilnehmer erhält nun die Möglichkeit, für jedes Projekt einen Zielort für die lokale Kopie zu wählen. Daraufhin schickt er eine Liste der an den Zielorten noch fehlenden oder veralteten Projektdateien zurück an den Host, der diese in einer ZIP-Archivdatei verpackt und an den Teilnehmer sendet. Eine detailliertere Beschreibung der Project Negotiation findet sich in Abschnitt 3.1.

Session und Project Negotiation finden nicht nur während der Erstellung einer neuen Sitzung statt, sondern können auch während einer laufenden Sitzung zum Einsatz kommen:

- Lädt der Host einen weiteren Nutzer zu einer laufenden Sitzung ein, so führt er mit diesem eine Session Negotiation und anschließend eine Project Negotiation mit den zur Zeit geteilten Projekten durch.
- Der Host kann in einer laufenden Sitzung ein weiteres Projekt teilen. Hierfür führt er eine Project Negotiation mit jedem anderen Teilnehmer durch, durch die er die neu hinzugefügten Projekte verteilt.

2.1.3 Aktivitäten

Innerhalb einer Sitzung kommunizieren Saros-Instanzen fast ausschließlich über so genannte **Aktivitäten** (*activities*). Diese Nachrichten nutzen sie

[6] Statt ganzer Projekte können auch nur Teile daraus geteilt werden (*partially-shared projects*). Diese Option hat aber keine Relevanz für diese Arbeit, weshalb hier der Einfachheit halber nur von "Projekten" und nicht "(Teil-)Projekten" gesprochen wird.

hauptsächlich, um sich gegenseitig über relevante Aktionen (z.B. Einfügen von Text in eine geteilte Datei) zu informieren, sodass diese unmittelbar bei jedem Sitzungsteilnehmer ausgeführt werden können. Aktivitäten sind also das Fundament der von Saros gebotenen Echtzeitsynchronisation.

Einige Arten von Aktivitäten werden außerdem für verschiedene Aspekte der Sitzungsverwaltung genutzt, z.B. für die Erkennung von Inkonsistenzen mithilfe des *Concurrency-Watchdog*-Mechanismus (siehe Aschnitt 2.1.5).

Alle Aktivitäten von Nicht-Host-Teilnehmern werden zunächst an den Host geschickt und von diesem dann ggf. weiterverteilt. Der Host hat also die volle Kontrolle über den Aktivitätsfluss und kann diesen beispielsweise zu Konsistenzzwecken ordnen [Sch13, Abs. 5.1].

2.1.4 Der Jupiter-Algorithmus

Da Saros es mehreren Sitzungsteilnehmern erlaubt, dieselbe Datei gleichzeitig zu bearbeiten, besteht die Gefahr von Konflikten, beispielsweise wenn zwei Teilnehmer gleichzeitg an derselben Stelle Text einfügen. Um bei solchen konkurrierenden Änderungen ein deterministisches Ergebnis zu erhalten, integriert das Saros-Kommunikationsprotokoll eine Variante des so genannten **Jupiter-Algorithmus**[7] [NCDL95]. Dieser definiert ein Client-Server-System, bei dem die Klienten jede durchgeführte Bearbeitungsoperation an einen zentralen Server senden. Der Server **transformiert** diese für jeden Klienten jeweils so, dass sie bei Anwendung auf den lokalen Zustand denselben Effekt hat wie bei allen anderen Klienten. Gibt es also beispielsweise zwei Klienten Alice und Bob, die gleichzeitig die folgenden Operationen ausführen:

- *Alice*: Zeichen a an Position x einfügen
- *Bob*: Zeichen b an Position x einfügen

könnte Server beispielsweise folgende transformierten Operationen an Alice und Bob weiterleiten:

- *Alice*: Zeichen b an Position $x + 1$ einfügen
- *Bob*: Zeichen a an Position a einfügen

sodass bei beiden Klienten letztendlich die Zeichenkette ab an Position x eingefügt wird.

[7] Der Algorithmus ist nach dem am Xerox-PARC-Institut entwickelten Kollaborationssystem *Jupiter* benannt, in dessen Kontext der Algorithmus entwickelt und vorgestellt wurde [NCDL95].

In Saros werden Bearbeitungsoperation durch Aktivitäten (siehe vorherigen Abschnitt) des Typs `JupiterActivity` repräsentiert. Die Rolle des Jupiter-Servers übernimmt der Host. Dieser ist als Sitzungsteilnehmer aber auch Klient; er sendet also Operationen an sich selbst und empfängt von dort auch die Operationen der anderen Teilnehmer.

2.1.5 Concurrency Watchdog

Um Inkonsistenzen zwischen den Projektkopien der Sitzungsteilnehmer entgegenzuwirken, die durch verloren gegangene oder fehlerhaft verarbeitete Aktivitäten entstehen können, wurde das Konzept des **Concurrency Watchdog** als Rückfallmechanismus eingeführt. Mit diesem kann erkannt werden, wenn eine geöffnete Datei bei einem Sitzungsteilnehmer nicht mit der Version des Hosts übereinstimmt. Die Nicht-Host-Teilnehmer berechnen dafür regelmäßig eine Prüfsumme für jede offene Datei und schicken diese an den Host. Errechnet der Host eine andere Prüfsumme, weist er den betreffenden Teilnehmer darauf hin, der dann um die aktuelle Version der Datei bitten kann.

2.2 Der Saros-Entwicklungsprozess

An einigen Stellen in dieser Arbeit wird auf Aspekte des Entwicklungsprozesses von Saros Bezug genommen. Diese sind hier kurz erläutert.

Saros ist ein Open-Source-Projekt, dessen Code mit dem verbreiteten Versionskontrollsystem Git[8] verwaltet wird. Saros-Entwicklern ist es jedoch nicht erlaubt, Git-Commits direkt in das Saros-Repository hochzuladen. Stattdessen wird eine Instanz des Code-Review-Systems *Gerrit*[9] verwendet, in dem ein Commit als so genannter **Change** veröffentlicht werden kann. Die anderen Entwickler haben dann die Möglichkeit, den Change zu begutachten, Fragen zu stellen und auf Probleme hinzuweisen. Der Ersteller des Changes kann seinerseits Antwortkommentare verfassen sowie neue Versionen des Changes (**Patch Sets**) hochladen. Hält ein Entwickler einen Change für bereit, kann er ihn befürworten (*+1*) oder ihn für die Integration in den Saros-Hauptzweig freigeben (*+2*). Ist letzteres geschehen, kann der Change per Knopfdruck in den Hauptzweig aufgenommen werden und gilt als integriert (*merged*).

Jeder Change ist mit einer Nummer versehen, mit der er referenziert werden kann (z.B. “Change 2264”). An einigen Stellen dieser Arbeit sind

[8] http://git-scm.com/

[9] https://www.gerritcodereview.com/ (Instanz des Saros-Projekts: http://saros-build.imp.fu-berlin.de/gerrit/)

solche Change-Nummern für erstellte Änderungen erwähnt. Eine volle Liste aller für diese Arbeit erstellten Changes findet sich im Anhang.

3 Projektteilen für Nicht-Hosts

Dieses Kapitel stellt das neu entwickelte "**Non-Host Project Sharing**"-Protokoll vor, das Nicht-Host-Sitzungsteilnehmern das Teilen von Projekten erlaubt und so Serversitzungen mit geteilten Projekten ermöglicht. Es entstand in Zusammenarbeit mit Ute Neise, die sich in ihrer laufenden Studienarbeit [Nei] schon vorher mit der Vervollständigung des Bussas-Server-Prototypen zu beschäftigen begann.

Es folgt zunächst eine detaillierte Beschreibung der bereits in Abschnitt 2.1.2 kurz erwähnten *Project Negotiation*, auf der Non-Host Project Sharing aufbaut. Daraufhin werden Anforderungen, Entwurf und Implementation des neuen Protokolls erklärt und ein Einblick in den gemeinsamen Entwicklungsprozess mit Neise gegeben.

3.1 Die Project Negotiation im Detail

Die **Project Negotiation** führt traditionell der Host der Sitzung mit jeweils einem Nicht-Host-Teilnehmer durch, um Projekte zu teilen. Der Ablauf dieses Nachrichtenaustausches soll im Folgenden genauer beleuchtet werden. Der Einfachheit halber werden dabei die beteiligten Parteien als *Alice* (teilender Host) und *Bob* (empfangender Nicht-Host) bezeichnet. Einen visuellen Überblick über die Project Negotiation bietet Abbildung 1.

Alice beginnt die Project Negotiation mit einem so genannten **Project Negotiation Offer (PNOF)**. Mit dieser Nachricht informiert sie Bob über die angebotenen Projekte. Neben den Projektnamen werden dabei für jedes Projekt die Liste aller geteilten Dateien sowie die MD5-Hashwerte der Dateiinhalte übertragen.

Hat Bob die Zielorte für die Projekte gewählt, ermittelt er zunächst, welche der in der PNOF-Nachricht gelisteten Dateien schon vorliegen (für den Fall, dass es sich bei einem Zielort um eine ältere Kopie desselben Projekts handelt). Hierfür prüft er für jede in der PNOF-Nachricht gelisteten Datei, ob sie lokal existiert und wenn ja, ob ihr errechneter Hashwert mit dem vom Host übertragenen übereinstimmt. Die Teilliste der Dateien, auf die eines von beidem nicht zutrifft (die bei Bob also nicht oder in anderer Version vorliegen), sendet Bob in Form eine Nachricht vom Typ **Project Negotiation Missing Files (PNMF)** an Alice zurück.

Hat Alice die PNMF-Nachricht von Bob erhalten, ist es ihre Aufgabe, die erbetenen Dateien in Form einer ZIP-Archivdatei zu verpacken und an Bob zu senden. Da die Negotiation jedoch nebenläufig zur Aktivitätsverarbeitung und der Benutzeroberfläche des Host-Benutzers ist, ergibt sich

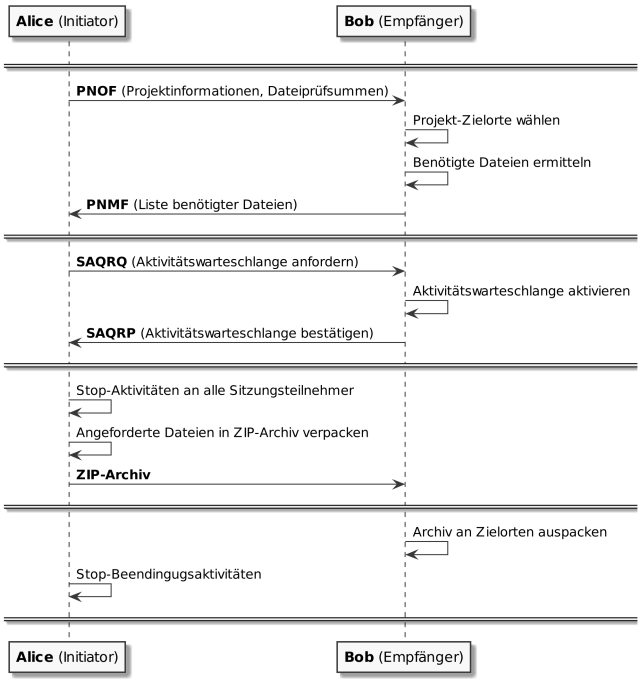

Abbildung 1: Die Project Negotiation als UML-Sequenzdiagramm.

ein Konsistenzproblem: finden Dateiänderungen statt, während das Archiv zusammengestellt wird, besteht die Gefahr, dass ein inkonsistenter Projektzustand festgehalten wird – beispielsweise, wenn eine Refaktorisierung angewandt wird, die sowohl bereits ins Archiv geschriebene als auch noch zu schreibende Dateien betrifft.

Um solche Fälle zu vermeiden, sendet Alice zunächst an jeden Teilnehmer (einschließlich sich selbst) eine **Stop-Aktivität** (`StopActivity`), die dazu auffordert, den Nutzer durch Blockierung der Oberfläche an Dateiänderungen zu hindern. Haben alle Teilnehmer eine Antwort-Aktivität zur Bestätigung gesendet, verschickt Alice zusätzlich einen **Start Activity Queuing Request (SAQRQ)** an Bob. Damit weist sie Bob an, alle zukünftig erhaltenen Aktivitäten nicht sofort zu verarbeiten, sondern sie stattdessen in einer Warteschlange zu sammeln. Auf diese Weise verwirft Bob keine Aktivitäten, die er noch nicht anwenden kann, weil er die Projekte noch nicht empfangen und ausgepackt hat. Bob bestätigt die Aktivierung der Warteschlange mit einer **Start Activity Queuing Response (SAQRP)**, woraufhin Alice schließlich das Archiv mit den Projektdateien erstellt.

Ist das Archiv fertiggestellt, hebt Alice die Blockierung der Benutzeroberflächen auf, indem sie entsprechende Aktivitäten an alle anderen Teil-

nehmer schickt. Daraufhin überträgt Alice das Archiv an Bob. Hat er es vollständig empfangen, entpackt Bob die Dateien an die vorher gewählten Zielorte und arbeitet die Warteschlange mit den Aktivitäten ab, die er während der Übertragung des Archivs erhalten hat. Nun hat er den aktuellen Zustand des Projekts, und die Project Negotiation ist abgeschlossen.

3.2 Anforderungsanalyse

Die Grundlage für den Entwurf des hier vorgestellten Protokolls bildeten die Analysen von Sebastian Starroske und Patrick Schlott [Sta13, Sch13]. Beide untersuchten die in früheren Saros-Versionen vorhandene Möglichkeit, auch als Nicht-Host Projekte zu teilen. In dieser Implementation führte ein Nicht-Host hierfür genauso wie der Host eine Project Negotiation mit jedem anderen Sitzungsteilnehmer durch. Daraus ergaben sich jedoch grundlegende Konsistenz- und Nebenläufigkeitsprobleme, die auf das Fehlen einer zentralen Koordinators für den Verteilungsprozess zurückzuführen waren. Aus diesem Grund entschied sich Schlott dafür, das Projektteilen nur noch dem Host zu erlauben und die Option für Nicht-Hosts zu entfernen.

Jedoch merkt Schlott in seiner Arbeit an, dass das Projektteilen für Nicht-Hosts grundsätzlich implementierbar sei, sofern die Verteilung über den Host und nicht eigenständig durch den Nicht-Host stattfinde, ähnlich wie es bei Aktivitäten der Fall sei [Sch13, Abs. 5.4.2]. Dieser Idee folgend arbeitete Ute Neise an einer Möglichkeit, Projekte in einer Serversitzung zu teilen, indem diese zunächst an den Server gesendet und dann von dort verteilt werden.

Zu Beginn unserer Zusammenarbeit erarbeiteten wir folgende Anforderungen für ein solches Protokoll:

- Der vorhandene Mechanismus der Project Negotiation sollte so weit wie möglich wiederverwendet und Unterschiede minimiert werden. Dies verringert nicht nur den Aufwand für die Implementation, sondern auch die spätere Wartung.
- Die Implementation soll Nicht-Hosts auch in klassischen host-basierten Sitzungen das Teilen von Projekten ermöglichen, um den Wert der Funktion zu erhöhen und server-basierte Sitzungen möglichst wenig speziell zu machen.
- Der Entwurf des Protokolls soll gegenüber dem existierenden host-initiiertem Projektteilungsprozess keine zusätzlichen Nebenläufigkeits- oder Konsistenzprobleme aufweisen.

Zusätzlich formulierten wir die gewünschten Konsistenzgarantien, die das Protokoll bieten sollte, in Form einfacher Benutzungsszenarien. In diesen bezeichnet Alice den Host der Sitzung und Bob den projektteilenden Teilnehmer:

1. Bearbeitet Bob die geteilte Projekte, bevor Alice es angenommen hat, sollte Alice die bereits bearbeiteten Version des Projekts erhalten.
2. Bearbeitet Bob de geteilte Projekt, während er es an Alice sendet, sollten diese Änderungen nicht verloren gehen, d.h. Alice wird über diese Änderungen informiert und kann sie auf die erhaltene Version des Projekts anwenden.
3. Bearbeitet Bob die geteilten Projekte, nachdem Alice sie erhalten, aber bevor sie sie an die anderen Teilnehmer verteilt hat, sollten auch diese Änderungen verteilt werden, sodass allen Teilnehmer nach Abschluss des Prozesses dieselbe Version der Projekte haben.

3.3 Entwicklungsprozess

Die Entwicklung des Non-Host Project Sharing folgte einem iterativen Prozess. Jede Iteration wurde durch ein Treffen mit Neise eingeleitet, in dem wir hauptsächlich Entwurfsideen erarbeiteten und diskutierten. Darauf folgte jeweils eine arbeitsteilige Implementationsphase, in der wir versuchten, diese Ideen im Saros-Code umzusetzen. Dabei versuchten wir oft, gleichzeitig zwei konkurrierende Entwurfsalternativen zu implementieren, um besser deren Vor- und Nachteile abwägen zu können, sodass in einem nachfolgenden Treffen eine fundierte Entwurfsentscheidung getroffen werden konnte.

Ein Beispiel für eine solche Entwurfsentscheidung war die Frage, ob für die Projektübertragung Nicht-Host zu Host wiederverwendet sollte oder neuer, darauf zugeschnittener Nachrichtenaustausch entwickelt werden sollte. Ersterer Ansatz entsprach der Anforderung, Wiederverwendung zu maximieren, barg aber die Gefahr, dass die ohnehin komplexe Implementation der Project Negotiation durch möglicherweise benötigte Fallunterscheidungen zusätzlich verkompliziert werden könnte, was Wartbarkeit des Codes so beeinträchtigen würde. Ein separater Negotiation-Typ würde dieses Problem vermeiden, dupliziert aufgrund ihres ähnlichen Zweckes jedoch möglicherweise große Teile der Project Negotiation und deren Implementation. Durch die Entscheidung, beide Ansätze parallel zu verfolgen, konnten wir schnell zu dem Schluss kommen, dass der Nachteil der Duplikation gegenüber dem potentiellen Vorteil eines separaten Nachrichtenaustausches

klar überwog; außerdem fiel die Zahl der nötigen Fallunterscheidungen im Project-Negotiation-Code weit kleiner aus als gedacht. Auf Basis dieser Ergebnisse konnten wir uns zuversichtlich für die Wiederverwendung der Project Negotiation entschieden. Eine ähnlich tiefgehende Betrachtung beider Alternativen durch einen einzelnen Entwickler hätte vermutlich deutlich mehr Zeit benötigt.

Für die Kommunikation von Fortschritten und Erkenntnissen während der Implementationsphasen nutzten wir die Teamplattform *Slack*[10]. Sie erwies sich nicht nur wegen ihrer persistenten Chaträume, sondern auch wegen der Möglichkeit des Dateiuploads (z.B. für Patch-Dateien oder Diagramme) als nützlich. Die hieraus entstandene geteilte Dokumentation des Entwicklungsverlaufs war auch für die Zusammenstellung dieses Kapitels hilfreich. Leider versäumten wir es oft, die Inhalte der persönlichen Treffen festzuhalten und ebenfalls in Slack zu hinterlegen, sodass der Verlauf dort nur lückenhaft dokumentiert ist. Eine diszipliniertere Aufzeichnung wesentlicher Entwurfsfragen und -entscheidungen bei jedem Treffen hätte es sehr viel einfacher gemacht, die Entwicklung im Nachhinein in Erinnerung zu rufen und nachzuvollziehen.

3.4 Entwurf

Der finale Entwurf des Non-Host Project Sharing ist in Abbildung 2 dargestellt. Der Prozess ist in zwei Phasen gegliedert, die jeweils ausschließlich aus unveränderten Project Negotiations bestehen: das Anbieten und Teilen von Projekten mit dem Host (die **Non-Host Project Negotiation**) und die anschließende Verteilung dieser Projekte durch den Host an die anderen Sitzungsteilnehmer (die **Host Project Negotiations**).

Die Non-Host Project Negotiation ist eine normale Project Negotiation und läuft ab wie in Abschnitt 3.1 beschrieben. Der einzige Unterschied betrifft den Zeitpunkt nach Übersenden der Projektmetadaten (d.h., der initialen PNOF-Nachricht) durch den teilenden Nicht-Host. Statt die angebotenen Projekte durch eine PNMF-Antwortnachricht zu akzeptieren, hat der Host hier die Möglichkeit, die Aufnahme der Projekte zur Sitzung abzulehnen, indem er ein Abbruchsignal an den Nicht-Host sendet.[11] Hierdurch

[10] https://slack.com/

[11] Im Prinzip hat ein Nicht-Host genauso die Möglichkeit, eine vom Host stammende Project Negotiation durch ein Abbruchsignal abzulehnen. Jedoch werden die geteilten Projekte in diesem Fall dennoch zur Sitzung hinzugefügt, und der Teilnehmer begibt sich in einen inkonsistenten Zustand. Klickt ein Saros/E-Nutzer die *Cancel*-Schaltfläche im Dialog zur Zielort-Auswahl, wird er sogar darauf hingewiesen, dass ein Abbruch zur Entfernung aus der Sitzung führt; diese Konsequenz war jedoch nicht zu beobachten.

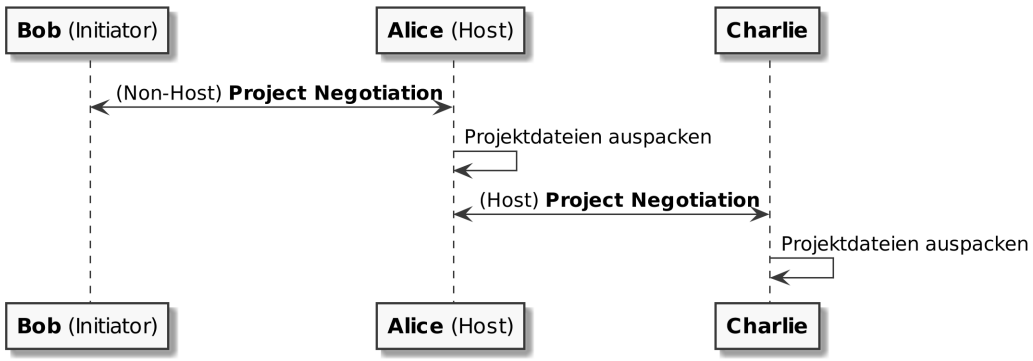

Abbildung 2: Eine schematische Darstellung des Non-Host Project Sharing.

behält der Host die volle Kontrolle darüber, welche Projekte in der Sitzung geteilt werden.[12]

Nimmt Alice die von Bob angebotenen Projekte an, setzen beide die Project Negotiation wie üblich fort. Hiernach gelten die Projekte bereits als Teil der Sitzung und werden bereits zwischen dem Sender und dem Host synchronisiert.

Um die neuen Projekte auch den anderen Sitzungsteilnehmern zur Verfügung zu stellen, beginnt der Host nun parallel mit jedem von ihnen eine Project Negotiation. Der Host geht also so vor, als würde er das Projekt direkt selbst teilen, mit der Ausnahme, dass er den ursprünglichen Sender der Projekte auslässt (dieser besitzt und synchronisiert die Projekte ja bereits). Sind diese *Host Project Negotiations* abgeschlossen, sind die Projekte bei allen Sitzungsteilnehmern synchron, und der Prozess ded Non-Host Project Sharing ist beendet.

Dieser Protokollentwurf erfüllt alle in Abschnitt 3.2 genannten Anforderungen:

- Der vorhandene Mechanismus der Project Negotiation wird unverändert wiederverwendet; es wurden keine zusätzlichen Nachrichten eingeführt. Dieser hohe Grad an Wiederverwendung spiegelt sich auch in der Implementation wider (siehe Abschnitt 3.5).

- Das Protokoll ist gleichermaßen und unverändert für server- und host-basierte Sitzungen verwendbar. Es unterscheidet sich lediglich in der Art der Entscheidung über die Annahme oder Ablehnung von Pro-

[12] Im dem Fall, dass ein Server die Sitzung leitet, muss diese Entscheidung automatisch erfolgen. Der in dieser Arbeit entwickelte Sitzungsserver nimmt alle Projekte bedingungslos an (siehe Abschnitt 5.3.2).

jekten auf der Hostseite (benutzereingabengesteuert bei einem menschlichen Host, automatisiert im Fall eines Sitzungsservers).

- Durch konsequente Wiederverwendung der Project Negotiation erhält das Non-Host Project Sharing dieselben Konsistenzgarantien, was die Überprüfung der Implementation anhand der aufgestellten Benutzungsszenarien bestätigte (siehe 3.6).

3.5 Implementation

Die Implementation des Protokolls erwies sich als überraschend schwierig. Der Code der Project Negotiation war schwer zu verstehen und eng mit den Klassen zur Sitzungsverwaltung verzahnt, sodass die Nachverfolgung der erzeugten Seiteneffekte (z.B. die Registrierung der Projekte als "geteilt") einen beträchtlichen Aufwand darstellte. Hinzu kam, dass keine aktuelle Dokumentation über die Details des Project-Negotiation-Protokolls existierte, sodass diese aus dem schwer zu verfolgenden Code hergeleitet werden musste. Aus diesen Gründen war nicht nur ein intensives Studium des Codes, sondern auch eine Analyse der – glücklicherweise sehr ausführlichen – Logging-Ausgaben von Saros sowie zahlreiche Trial-and-Error-Codeänderungen nötig, um die Funktionsweise des Project-Negotiation-Codes zu ergründen.

Nach Überwinden dieser Einstiegshürden konnte das Non-Host Project Sharing jedoch mit einer sehr kleinen Menge and Codeänderungen implementiert werden. Die nötigen Änderungen beschränken sich auf die Klassen `OutgoingProjectNegotiation` und `IncomingProjectNegotiation`, die den Sender- und Empfängerteil der Project Negotiation implementieren, sowie `SarosSession` und `SarosSessionManager`[13].

Im Wesentlichen wurden folgende Änderungen durchgeführt:

- Im Fall einer Non-Host Project Negotiaion registriert `OutgoingProjectNegotiation` die Projekte erst dann bei `SarosSession` als geteilt, sobald der Host sie angenommen hat. Vorher geschah dies immer im bereit vor Erstellung der `OutgoingProjectNegotiation` in `SarosSessionManager`, da die Möglichkeit der Ablehnung von Projekten nicht vorgesehen war.

- Da `OutgoingProjectNegotiation` nun nicht mehr davon ausgehen kann, dass die Projekte bereits bei `SarosSession` registriert ist, kann

[13] Diese Klassen waren zunächst zwischen dem Saros-Eclipse- und IntelliJ-Plugin dupliziert, konnten jedoch im Laufe der Arbeit in den Saros-Kern migriert werden; siehe Abschnitt 4.

die Klasse nicht mehr von dort die Projektmetadaten beziehen. Diese werden nun stattdessen als ein Instanz der neu eingeführten Datenklasse `ProjectsToShare` übergeben.

- Der `SarosSessionManager` wurde um Code erweitert, der alle als Host erhaltene Projekte an die anderen Teilnehmer der Sitzung verteilt.

Insgesamt wurden 296 Codezeilen hinzugefügt und 65 Zeilen entfernt. Zum Zeitpunkt dieser Arbeit ist der Code noch nicht in den Hauptzweig von Saros integriert, befindet sich jedoch als Gerrit-Patch 2264 in Begutachtung[14].

3.6 Test

Während und nach der Implementation führten wir regelmäßig Tests anhand der in Abschnitt 3.2 genannten Benutzungsszenarien durch. Folgender Testablauf mit drei Saros/E-Nutzern (im Folgenden Alice, Bob und Carl genannt) hat sich aufgrund seiner großen Fallabdeckung bewährt:

1. Zunächst erzeugt Alice ein leeres Projekt und teilt es mit Bob und Carl, um eine Sitzung zu erstellen. Alice wird also der Host der Sitzung.

2. Bob erstellt ein Projekt mit einer großen Datei (z.B. 250 MB), die eine ausreichend große Zeit für das Verpacken und Versenden des Projekts gewährleistet. Außerdem fügt er eine Textdatei mit beliebigem Inhalt zum Projekt hinzu.

3. Nun teilt Bob das neu erstellte Projekt. Bevor Alice es annimmt, fügt Bob ein oder mehrere Zeichen zur Textdatei hinzu.

4. Alice nimmt das Projekt an. Während auf Bobs Seite ein Archiv für das Projekt erstellt wird, versucht Bob, Zeichen zur Textdatei hinzuzufügen. Seine Saros-Instanz sollte ihm melden, dass die Datei zur Zeit schreibgeschützt ist, und die Datei unverändert lassen.

5. In dem Zeitraum, indem das Projekt nach dem Verpacken an Alice übertragen wird, versucht Bob wieder, Zeichen zur Textdatei hinzuzufügen. Dies sollte nun möglich sein.

6. Hat Alice das Projekt erhalten und ausgepackt, sollte auf Seiten von Carl unmittelbar ein Dialog zur Annahme des Projekts erscheinen.

[14] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2264/8

Bevor Carl annimmt, öffnet Alice die Textdatei. Sie sollte exakt denselben Inhalt haben wie im Editor von Bob. Fügt Alice nun ein Zeichen zur Datei hinzu, sollte dieses unmittelbar bei Bob erscheinen; tut Bob dasselbe, sollte die Änderung umgekehrt bei Alice angewendet werden.

7. Carl nimmt das Projekt an. Während Alice das Projekt verpackt, sollte Bob die Textdatei nicht bearbeiten können. Während der Übertragung des Projekts an Carl fügt Bob wie schon zuvor einige Zeichen zur Datei hinzu.

8. Hat Carl das Projekt erhalten, öffnet er die Textdatei, die denselben Inhalt haben sollte wie im Editor von Bob (und Alice). Auch seine Änderungen sollten unmittelbar bei Alice und Bob sichtbar sein.

In einem separaten Test klickt Alice im Projekt-Annahme-Dialog *Cancel*, statt anzunehmen. Bob sollte über den Abbruch informiert und das Projekt bei ihm weiterhin als nicht geteilt angezeigt werden.

Mit der aktuellen Implementation des Non-Host Project Sharings liefern diese Tests alle erwarteten Ergebnisse. Jedoch wurden die Tests ausschließlich manuell ausgeführt, was eine zukünftige Verifizierung erschwert. Bevor die Funktion in den Hauptzweig von Saros einkehrt, wäre es wünschenswert, die beschriebenen Testabläufe mithilfe des Saros-eigenen *STF-Frameworks*[15] zu automatisieren, um zukünftige Regressionen erkennen zu können; aus Zeitgründen konnte dies jedoch nicht im Rahmen dieser Arbeit umgesetzt werden.

3.7 Offene Probleme und Fragen

Obwohl das vorgestellte Protokoll alle gewünschten Anforderungen erfüllt und die Implementation vollständig funktionsfähig ist, bestehen noch Verbesserungspotenzial und Unsicherheit in Bezug auf zukünftige Entwicklungen. Diese werden im Folgenden angesprochen und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

3.7.1 Längere Dauer

Ein Nachteil des Protokolls gegenüber dem direkten Teilen von Projekten durch den Host ist, dass die Verteilung der Projekte unter allen Teilnehmern bis zu doppelt so viel Zeit in Anspruch nimmt. Dies rührt daher,

[15] http://saros-build.imp.fu-berlin.de/stf/STF_Manual.pdf

dass die Non-Host Project Negotiation zwischen dem teilenden Nicht-Host und dem Host vollständig abgeschlossen werden muss, bevor der Host im zweiten Schritt die Projekte an die anderen Teilnehmer verteilen kann. Bei Sitzungen mit mehr als zwei Teilnehmern beträgt die Verteilungsdauer also mindestens der Dauer zweier hintereinander laufender Project Negotiations für die Projekte, was das zweifache Ver- und Entpacken der Projekte und deren zweifache Übertragung über das Netzwerk einschließt. Bei kleinen Projekten, die z.B. hauptsächlich aus Textdateien bestehen, ist dies unproblematisch, doch für größere Projekte kann diese Tatsache einen signifikanten Nachteil gegenüber dem klassischen "Host Project Negotiation" darstellen, bei dem alle nötigen Project Negotiations zur selben Zeit ablaufen.

Ein Ansatz, dieses Problem in Zukunft zu umgehen, ist eine Form des *Projekt-Streamings* über den Host einzuführen. In diesem Modell beginnt der Host bereits Project Negotiations mit den anderen Teilnehmern, sobald er Projekte von einem Nicht-Host angenommen hat. Diese Negotiations führt er fort, bis das Projektarchiv erwartet wird. Beginnt dann die Übertragung des Archivs vom teilenden Nicht-Host, leitet der Host alle erhaltenen Daten unmittelbar an die anderen Teilnehmer weiter, sodass alle Übertragungen zur selben Zeit ablaufen und ungefähr gleichzeitig abgeschlossen werden können.

Projekt-Streaming würde die Gesamtdauer des Verteilungsprozesses signifikant reduzieren, indem es die Verzögerung der Host Project Negotiations gegenüber der Non-Host Project Negotiation nahezu entfernt. Außerdem muss bei diesem Modell das Projektarchiv nur ein einziges Mal auf Seiten des teilenden Nicht-Hosts erstellt werden, da der Host dieses lediglich unverändert weiterleitet. Jedoch würde die Implementation dieses Ansatzes vermutlich größere Umbauten der `IncomingProjectNegotiation`- und `OutgoingProjectNegotiation`-Klassen erfordern, sodass eine vorherige Kosten-Nutzen-Analyse – insbesondere hinsichtlich der Frage, ob die längere Verteilungsdauer für die Saros-Nutzerbasis in der Praxis ein Problem darstellt – angebracht wäre.

3.7.2 Integration mit aktivitätsbasierter Projektübertragung

Parallel zu dieser Arbeit erarbeitete David Damm in seiner Bachelorarbeit [Dam15] einen Entwurf und Prototyp für eine neue Variante der Project Negotiation, in der Projekte nicht durch ein ZIP-Archiv, sondern durch eine Sequenz von Aktivitäten übertragen werden, die jeweils eine Datei des Projekts erstellen. Auf diese Weise soll der Empfänger die bereits übertra-

genen Dateien schon betrachten und bearbeiten können, bevor er das ganze Projekt erhalten hat. Zum Zeitpunkt dieser Arbeit befasst sich Daniel Theus damit, diese Übertragungsform zu evaluieren und als Alternative in die Saros-Codebasis zu integrieren [The].

Die Interaktion zwischen Non-Host Project Sharing und aktivitätsbasierter Projektübertragung ist noch unklar. Ersteres ist zwar prinzipiell nicht an eine bestimmte Übertragungsform gebunden, aufgrund der unterschiedlichen Aktivitätsverarbeitung besteht jedoch die Möglichkeit von bisher noch nicht bedachten Problemen bei der aktivitätsbasierten Übertragung von Nicht-Host zu Host; diese wurde von Damm und Theus nicht getestet.

Ein Vorteil der aktivitätsbasierten Übertragung ist, dass sie Implementation von Projekt-Streaming (siehe den vorherigen Abschnitt) stark vereinfachen würde: der Host müsste die vom Nicht-Host geteilten Projekte lediglich über Project Negotiations bei den anderen Teilnehmern bekannt machen, bevor er die erhaltenen Datei-Erstellungsaktivitäten verarbeitet; die vorhandene Logik zur Verteilung von Aktivitäten würde dann dafür sorgen, dass alle Teilnehmer die Projektdateien nahezu zur selben Zeit erhalten.

3.8 Zusammenfassung

Mit dem “Non-Host Project Sharing”-Protokoll erhält Saros einen simplen, konsistenzerhaltenden Projektteilungsmechanismus für Nicht-Hosts in host- und server-basierten Sitzungen. Mit diesem wird der Server-Prototyp von Bussas praktisch nutzbar.

Jedoch besteht immer noch die problematische Abhängigkeit zu Eclipse. Das folgende Kapitel widmet sich den im Rahmen Arbeit getätigten Erweiterungen des Saros-Kerns, der die Implementation eines unabhängig lauffähigen Sitzungsservers ermöglichte.

4 Ergänzung des Saros-Kerns

Dieses Kapitel beschreibt die für diese Arbeit vorgenommenen Codemigrationen in den Saros-Kern, die als Vorbereitung für die Implementation des von Eclipse unabhängigen Sitzungsservers dienten (siehe Kapitel 5). Der Fokus liegt hierbei auf der Erläuterung der befolgten Vorgehensweise sowie der beispielhafte Vorstellung einiger Migrationen.

4.1 Über den Saros-Kern

Der Saros-Kern erfüllt zwei Funktionen. Zum einen ist er eine von den Saros-Varianten gemeinsam genutzte Bibliothek, in der die grundlegende Funktionalität von Saros implementiert ist (z.B. der Großteil des Kommunikationsprotokolls). Zum anderen stellt er Schnittstellen bereit, die zwar bestimmte Funktionalität bereitstellen, jedoch in jeder Saros-Variante getrennt implementiert werden müssen. Ein Beispiel für eine solche Schnittstelle ist `IPreferencestore`, die einen Schlüssel-Wert-Speicher für Saros-Einstellungen zur Verfügung stellt; sie ist in Saros/E als Adapter für die nahezu identische `IPreferencestore`-Schnittstelle der Eclipse-Plattform[16] implementiert, während die Saros/I-Implementation auf den von IntelliJ definierten `PropertiesComponent`-Mechanismus[17] zurückgreift.

Einige Klassen im Kern stellen nicht nur Dienste zur Verfügung, sondern agieren nach ihrer Erstellung auch selbständig, indem sie beispielsweise Hintergrundaufgaben in einem separaten Thread erledigen oder auf den Erhalt bestimmter Aktivitäten (siehe Abschnitt 2.1.3) reagieren. Diese Klassen werden hier – und auch oft innerhalb des Saros-Codes – als **Komponenten** bezeichnet.

4.2 Vorgehensweise

Der bei dieser Arbeit eingesetzte Prozess zur Ergänzung des Saros-bbKerns lässt sich in folgende Schritte gliedern:

1. Die Ermittlung essenzieller Klassen, deren Funktionalität für den zu implementierenden Server benötigt wird, die sich aber noch dupliziert in Saros/E und Saros/I befinden und nicht Teil des Kerns sind

[16] http://help.eclipse.org/mars/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/preference/IPreferenceStore.html

[17] http://www.jetbrains.org/intellij/sdk/docs/basics/persisting_state_of_components.html

2. Für jede ermittelte Klasse, die Erstellung eines *Abhängigkeitsgraphen*, um alle Abhängigkeiten zu Typen der Eclipse- bzw. IntelliJ-Plattform sowie zu anderen Saros-Klassen außerhalb des Kerns zu erkennen
3. Für jede gefundene Abhängigkeit, die Wahl und Umsetzung geeigneter Refaktorisierungen, um die Abhängigkeit aus dem Graphen zu entfernen
4. Nach Entfernung aller problematischen Abhängigkeiten, die Verschiebung der Klasse in den Kern und die Entfernung ihrer Duplikate

Diese Schritte wurden nicht linear durchlaufen, sondern iterativ für jede als essenziell ermittelte Nicht-Kern-Klasse. Diese Art des Vorgehens war besonders deshalb angebracht, da die meisten essenziellen Klassen erst im Laufe der Arbeit mit wachsendem Wissen über die Codebasis ermittelt werden konnten. Beispielsweise fiel erst während der Implementierung des Servers auf, dass dieser aufgrund der fehlenden Nicht-Kern-Komponente `LeaveAndKickHandler` nicht die Nachrichten verarbeitet, mit denen Teilnehmer das Verlassen der Sitzung melden – ein essentieller Teil der Sitzungsverwaltung.

Die folgenden Abschnitte beschreiben die genannten Prozessschritte im Detail.

4.2.1 Essentielle Nicht-Kern-Klassen ermitteln

Den Ausgangspunkt für die Kernergänzung stellte die Identifikation der für den Sitzungsserver essenziellen Nicht-Kern-Klassen dar. “Essenziell” meint hier, dass ihre Funktionalität für die Implementation eines funktionsfähigen Servers unerlässlich ist und folglich dupliziert werden müsste, wenn sie nicht Teil des Kerns wäre. Nicht als essenziell eingestuft wurden Klassen für Zusatzfunktionen, die für den Hauptzweck von Saros unerheblich sind.

Dieser Schritt geschah hauptsächlich durch die Befragung der existierenden Saros-Entwickler. So konnte beispielsweise Stefan Rossbach, der sich bereits mehrere Jahre an der Entwicklung von Saros beteiligt hatte, wertvolle Informationen über den Stand des Saros-Kerns und die dort noch fehlenden Klassen geben. Eine weitere Quelle war die Liste der zwischen Saros/E und Saros/I duplizierten Klassen, die Arndt Lasarzik vor Veröffentlichung seiner Arbeit zum Thema Saros-Kern [Las15] zur Verfügung stellte. Zusätzlichen zu diesen Einblicken kamen Durchsichten der Codebasis, insbesondere des Pakets `de.fu_berlin.inf.dpp.core` in Saros/I, zum Einsatz.

Die zu migrierenden Klassen wurden ausschließlich aus Saros/E gewählt, da diese Variante länger in Entwicklung ist als Saros/I und sich bereits in der Praxis bewährt hat; es ist also zu vermuten, dass der Code ausgereifter und vollständiger ist.

4.2.2 Abhängigkeitsgraphen erstellen

Wurde eine essenzielle Nicht-Kern-Klasse identifiziert, galt es nun, alle direkten und transitiven Abhängigkeiten zu ermitteln, die einer Migration in den Kern im Weg stehen. Diese Abhängigkeiten wurden zu einem **Abhängigkeitsgraphen** geformt, dessen Knoten jeweils einen Saros-Typ außerhalb des Kerns (einen **Nicht-Kern-Typ**) oder einen von Eclipse bereitgestellten Typ (einen **Plattformtyp**) repräsentieren. Die Abhängigkeiten zwischen diesen Typen wurden durch gerichtete, vom abhängenden Typ ausgehende Kanten dargestellt. Nicht in den Graphen aufgenommen wurden Typen des Saros-Kerns, der Java-Standardbibliothek sowie anderer Bibliotheken wie z.B. `log4j`, da diese für den Prozess uninteressant sind. Das gleiche gilt für Abhängigkeiten von Plattformtypen zu anderen Plattformtypen. Ein Beispiel für einen Abhängigkeitsgraphen ist in Abbildung 3 grafisch dargestellt.

Betrachtet wurden statische Abhängigkeiten, also solche, deren Nichterfüllung zu Kompilierungsfehlern führen. In den meisten Fällen deckten sich diese mit den logischen Abhängigkeiten zwischen den Klassen, sodass sie eine gute und leicht überprüfbare Näherung abgaben. In einzelnen Fällen bestanden jedoch auch logische Abhängigkeiten, für die keine entsprechenden statischen Abhängigkeiten bestanden, sodass sie im Abhängigkeitsgraphen fehlten und erst später auffielen. Ein Beispiel hierfür ist die bereits erwähnte `LeaveAndKickHandler`-Komponente: die als essenziell eingestufte Klasse `SarosSession` referenziert diese zwar nicht direkt, erwartet jedoch, dass die Methode `removeUser()` von ihr aufgerufen wird, sobald ein Teilnehmer die aktuelle Sitzung verlässt. Für solche Fälle wäre eine weiterreichende Abhängigkeitsanalyse sinnvoll gewesen.

4.2.3 Problematische Abhängigkeiten entfernen

Nach der Erstellung eines Abhängigkeitsgraphen wurde für jede seiner Kanten entschieden, wie die entsprechende Abhängigkeit aus dem Graphen entfernt werden konnte. Dies geschah meist durch den Einsatz funktionalitätserhaltender Refaktorisierungen [Fow99], die die problematischen Abhängigkeiten in unproblematische (zu einem Typ im Saros-Kern) ver-

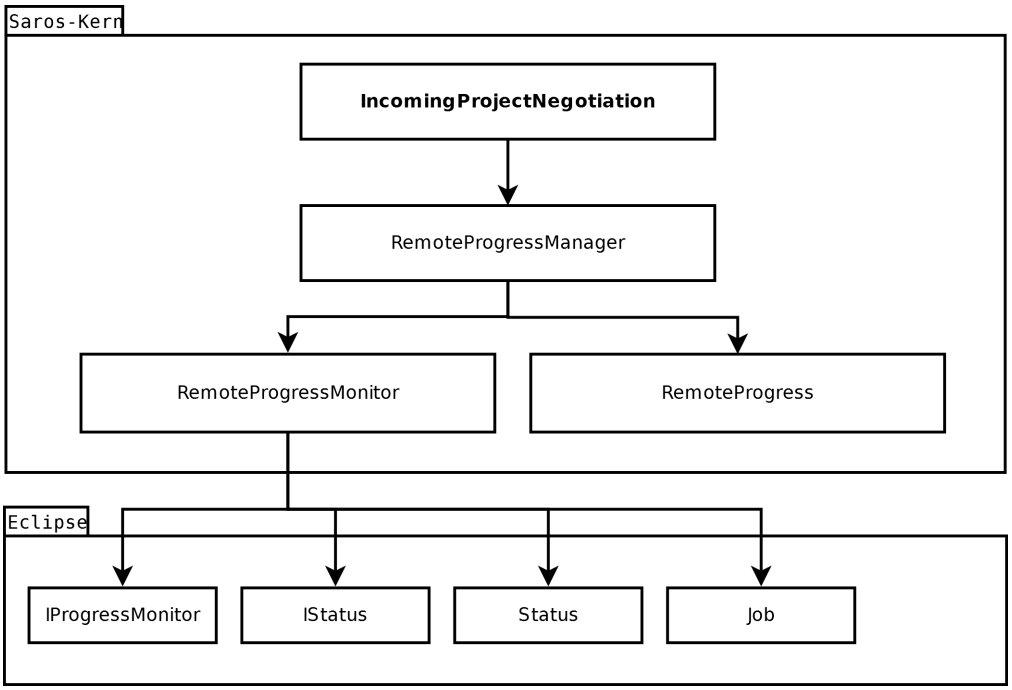

Abbildung 3: Grafische Darstellung des für `IncomingProjectNegotiation` erstellten Abhängigkeitsgraphen. In der Praxis wurden die Graphen informal in Stichpunkten oder – in einfachen Fällen – auch nur ïm Kopf"festgehalten.

wandelten. Dabei wurde inkrementell “von außen nach innen” vorgegangen, also beginnend mit den Abhängigkeiten zu den Typen an den Blättern des Graphen, bis die essenzielle Klasse von allen problematischen Abhängigkeiten befreit war und in den Kern verschoben werden konnte.

Die für diese Arbeit genutzten Refaktorisierungsstrategien lassen sich grob in folgende Klassen einordnen:

1. **Nicht-Kern-Typ in den Kern verschieben:** Nicht-Kern-Typen ohne problematischen Abhängigkeiten (also ohne ausgehende Kante im Abhängigkeitsgraphen) konnten direkt in den Kern verschoben werden. Oft war ebenfalls die Entfernung eines Duplikats in Saros/I nötig, ggf. auch die Verschiebung einer zugehörigen Testklasse.

2. **Plattformtyp-Abhängigkeit durch Kerntyp-Abhängigkeit ersetzen:** Oftmals konnte eine Abhängigkeit zu einem Typ der Eclipse-Plattform leicht durch die Nutzung einer bereits vorhandene Schnittstelle im Saros-Kern ersetzt werden. Ein Beispiel hierfür war eine Nutzung der bereits in Abschnitt 4.1 erwähnten Eclipse-Schnittstelle `IPreferencestore`, deren Portierung zur gleich benannten Schnittstelle des

Kerns lediglich die Änderung einer Importanweisung erforderte. In einigen Fällen war auch die Einführung neuer Methoden, Schnittstellen und Klassen im Kern sinnvoll, um diese Refaktorisierungsstrategie verfolgen zu können (siehe Abschnitt 4.3.1 für ein Beispiel).

3. **Nicht-Kern-Typ-Abhängigkeit durch Kerntyp-Abhängigkeit ersetzen:** In Fällen, in denen ein Nicht-Kern-Typ primär aus plattformspezifischem Code besteht (z.B. Klassen der Benutzeroberfläche), erschien die Ersetzung von Plattformtyp-Abhängigkeiten nicht sinnvoll. In diesen Situationen war es praktikabler, den Typ stattdessen in Saros/E bzw. Saros/I zu belassen und stattdessen die Abhängigkeiten zum Typ selbst zu entfernen. Hierbei konnten dieselben Techniken wie in der vorhergegangenen Refaktorisierungsstrategie verwendet werden. Jedoch ergeben sich hier aus der Möglichkeit, die Typen auf beiden Seiten der Abhängigkeit ändern zu können, zusätzliche Refaktorisierungsoptionen. Beispielsweise kann die Abhängigkeit unter Einsatz des **Observer**-Entwurfsmusters umgekehrt werden (siehe Beispiel in Abschnitt 4.3.3).

4.2.4 Vergleich zur Arbeit von Lasarzik (2015)

Der hier erläuterte Prozess ähnelt dem von Lasarzik beschriebenen, insbesondere in seiner Nutzung von Abhängigkeitsgraphen [Las15, S. 17]. Die beiden Ansätze unterscheiden sich hauptsächlich in der Art, wie die migrierten Klassen gewählt wurden. Während Lasarzik, dessen Arbeit sich vorrangig mit der Verringerung von Duplikation zwischen Saros/E und Saros/I beschäftigte, alle duplizierten Typen gleichermaßen in Betracht zog und lediglich anhand einer Definition von "Relevanz" priorisierte, war es für diese Arbeit notwendig, die Menge der zu migrierenden Nicht-Kern-Klassen problemorientiert zu wählen und möglichst klein zu halten, sodass nur der für das Ziel notwendige Aufwand getrieben werden musste.

4.3 Ergebnisse und Durchführung

Durch die Vorarbeiten von Lasarzik und anderen war die Zahl der im Kern fehlenden Klassen überraschend klein. Folgende Nicht-Kern-Klassen wurden als essenziell identifiziert:

- `OutgoingProjectNegotiation` und `IncomingProjectNegotiation` (Implementation der Project Negotiation, siehe Abschnitte 2.1.2 und 3.1)

- `SarosSession` und `SarosSessionManager` (Sitzungsinitialisierung und -verwaltung)
- `LeaveAndKickHandler`
- `SharedResourcesManager` (u.a. Verarbeitung von Dateisystem-Aktivitäten)

Von diesen Klassen wurden alle außer `SharedResourcesManager` (siehe Abschnitt 4.4) in den Kern migriert. Außerdem konnten zahlreiche ihrer Abhängigkeiten ebenfalls in den Kern verschoben werden. Zusätzlich wurde eine Migration der Klasse `ConcurrencyWatchdogServer` für die Hostseite des Watchdog-Mechanismus (Abschnitt 2.1.5) begonnen; diese war jedoch nicht essenziell für das Funktionieren des Sitzungsservers und wurde nicht beendet.[18]

Im Folgenden werden die eingesetzten Refaktorisierungsstrategien anhand einiger Beispiele illustriert.

4.3.1 Kern statt Plattform: `IRemoteProgressIndicator`

Die Klasse `RemoteProgressManager` war Teil des Abhängigkeitsgraphen von `IncomingProjectNegotiation`. Sie implementiert einen Mechanismus, über den ein Sitzungsteilnehmer einen anderen über den Fortschritt einer laufenden Operation informieren kann. Die Klasse ist dabei sowohl für das Senden von Fortschrittsaktivitäten (*Progress Activities*) als auch für das Lauschen nach solchen Aktivitäten und die Anzeige des berichteten Fortschritts zuständig. Letzteres geschah direkt durch die entsprechenden Schnittstellen der Plattform, sodass die Klasse zwischen Saros/E und Saros/I dupliziert war.

Da der Großteil des Codes von `RemoteProgressManager` nicht Eclipse-spezifisch war, fiel die Entscheidung für die Strategie "*Plattformtyp-Abhängigkeit durch Kerntyp-Abhängigkeit ersetzen*". Erreicht wurde dies durch Kapselung der Fortschrittsanzeige durch eine neue Kern-Schnittstelle `IRemoteProgressIndicator`. Diese wurde dann in Saros/E und Saros/I jeweils mit den entsprechenden Plattform-Schnittstellen implementiert (`EclipseRemoteProgressIndicatorImpl` bzw. `IntelliJRemoteProgressIndicatorImpl`). Da `RemoteProgressManager` neue `IRemoteProgressIndicator`-Instanzen dynamisch erzeugen muss (jede Instanz steht für eine entfernte Operation), wurde mit `IRemoteProgressIndicatorFactory` zusätzlich eine *Abstract Factory* [GHJV95, S. 87] eingeführt und ebenfalls in Sa-

[18] Diese Migration wird zum Zeitpunkt dieser Arbeit von David Sungalia fortgesetzt [Sun].

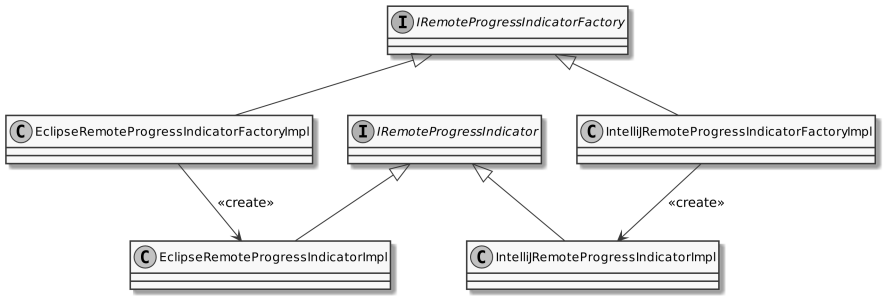

Abbildung 4: Der Zusammenhang zwischen `IRemoteProgressIndicator`, `IRemoteProgressIndicatorFactory` und den Implementierungen im Überblick. Die mit `Eclipse` und `IntelliJ` beginnenden Klassen sind in Saros/E bzw. Saros/I lokalisiert.

ros/E und Saros/I individuell implementiert. Abbildung 4 zeigt die resultierenden Entwurf im Überblick.

Mit diesen Vorbereitungen konnte `RemoteProgressManager` leicht so refaktorisiert werden, dass es nur noch von Kern-Typen abhing und selbst in den Kern verschoben werden konnte (*Nicht-Kern-Typ in den Kern verschieben*).

4.3.2 Kern statt Nicht-Kern: `ISarosSessionContextFactory`

Die Analyse der Abhängigkeiten von `SarosSession` ergab, dass die Klasse für die Initialisierung zahlreicher Komponenten beim Sitzungsstart zuständig war, beispielsweise `StopManager` zum Senden und Verarbeiten und Stop-Aktivitäten (siehe Abschnitt 3.1). Viele dieser Sitzungskomponenten waren leicht in den Kern zu migrieren, andere hatten jedoch tiefgreifende Plattformtyp-Abhängigkeiten und wären nur mit hohem Aufwand zu portieren gewesen.

Glücklicherweise waren die meisten problematischen Sitzungskomponenten optionaler Natur (z.B. Kollektoren für Sitzungsstatistiken), sodass sie für den Sitzungsserver nicht notwendig waren.[19] Da `SarosSession` aber in seinem Konstruktor alle zu erstellenden Komponenten direkt referenzierte, war es nicht ohne weiteres möglich, die Klasse nur mit einem Teil der Sitzungskomponenten in den Kern zu verschieben.

Um dies zu ändern, wurde die Wahl der zu erstellenden Sitzungskomponenten durch eine neue Kern-Schnittstelle namens `ISarosSessionContextFactory` abstrahiert. Diese ist der bereits im Kern definierten `ISarosContextFactory`-Abstraktion nachempfunden, die die Wahl der nicht sit-

[19] Eine wichtige Ausnahme ist `SharedResourcesManager`; siehe Abschnitt 4.4.

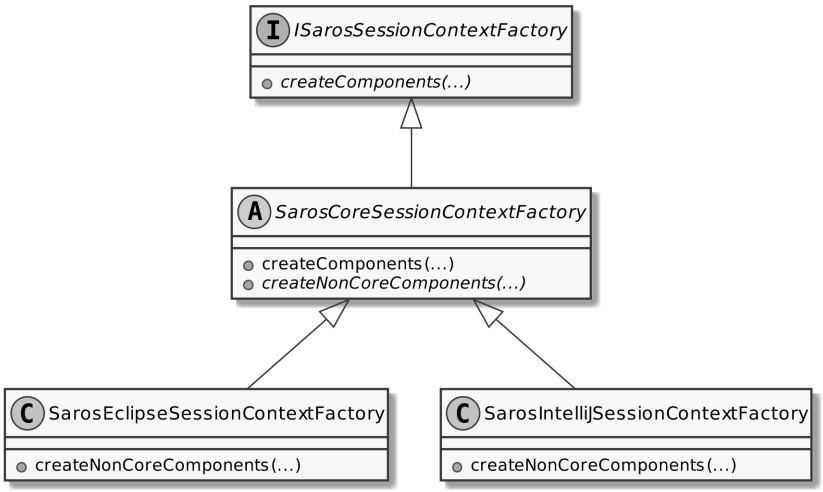

Abbildung 5: Die `ISarosSessionContextFactory`-Typhierarchie.

zungsgebundenen Komponenten (z.B. zum Lauschen auf Einladungsnachrichten) beim Start von Saros abstrahiert. Genau wie `ISarosContextFactory` definiert auch `ISarosSessionContextFactory` eine einzelne Methode `createComponents()`, deren individuelle Implementierung in jeder Saros-Variante es erlaubt, einen jeweils angepassten Satz an Sitzungskomponenten zu wählen. Durch diese Kapselung im Stile des *Strategy*-Entwurfsmusters [GHJV95, S. 315] können auch Nicht-Kern-Komponenten erstellt werden, ohne dass der Verwender der Schnittstelle diese kennen muss.

Abbildung 5 zeigt die `ISarosSessionContextFactory`-Typhierarchie. Neben Implementationen in Saros/E und Saros/I enthält diese auch eine abstrakte Klasse `SarosCoreSessionContextFactory`, in der die Sitzungskomponenten des Kerns aufgelistet werden. Die Unterklassen müssen also nur noch diejenigen Komponenten nennen, die spezifisch für die jeweilige Saros-Variante sind.

Durch die Nutzung der `ISarosSessionContextFactory`-Abstraktion im Konstruktor von `SarosSession` konnten die Abhängigkeiten zu den meisten Sitzungskomponenten, insbesondere den nicht essenziellen, entfernt werden (*Nicht-Kern-Typ-Abhängigkeit durch Kerntyp-Abhängigkeit ersetzen*). Dies ermöglichte es, `SarosSession` in den Kern zu verschieben und dennoch ein Teil der Sitzungskomponenten für eine spätere Migration in Saros/E und Saros/I zu belassen.

4.3.3 Abhängigkeitsumkehrung: `LeaveAndKickHandler`

Ein Problem bei der Migration der Komponente `LeaveAndKickHandler` war, dass diese bei einem unerwartetem Ende der Sitzung – wenn z.B. der Host die Sitzung verlassen hat – eine Methode der Klasse `SarosView` aufrief, um eine passende Nachricht in der Benutzeroberfläche anzuzeigen. Da `SarosView` hochspezifisch für die Oberfläche von Saros/E ist, kam eine Verschiebung in den Kern nicht in Frage. Die Abhängigkeit musste also entfernt werden.

Dies schien nach einem passenden Szenario für die Anwendung des *Observer*-Entwurfsmusters [GHJV95, S. 293]. Statt `SarosView` direkt zu referenzieren, könnte `LeaveAndKickHandler` einen Mechanismus zum Registrieren von Aktionen bieten, die bei einem unerwarteten Sitzungsende ausgeführt werden sollen. `SarosView` könnte dann diesen Mechanismus nutzen, um bei Bedarf die entsprechende Benachrichtigung anzuzeigen. Die Abhängigkeit von `LeaveAndKickHandler` zu `SarosView` würde so also durch eine umgekehrte Abhängigkeit von `SarosView` zu `LeaveAndKickHandler` ersetzt werden, sodass `LeaveAndKickHandler` in den Kern verschoben werden könnte.

Statt diese Refaktorisierung direkt durchzuführen, fiel die Entscheidung jedoch darauf, den vorhandenen Observer-Mechanismus des `SarosSessionManager` wiederzuverwenden. Diese Komponente wird ebenfalls von `LeaveAndKickHandler` benachrichtigt und erlaubte es bereits, auf das Ende einer laufenden Sitzung zu reagieren, indem ein `ISessionLifecycleListener`[20] mit einer Implementation der Methode `sessionEnded()` registriert wird. Es war lediglich nötig, `sessionEnded()` um einen Parameter zu erweitern, der den Grund des Sitzungsendes angibt, sodass `SarosView` die passende Benachrichtigung wählen kann. Die resultierenden Abhängigkeitsänderungen sind in den Abbildungen 6 und 7 dargestellt.

4.4 Nicht abgeschlossene Kernmigrationen

Als einzige essenziell eingestufte Klasse konnte die Sitzungskomponenten `SharedResourcesManager` nicht in den Kern migriert werden. In ihr findet die Verarbeitung von Dateisystem-Aktivitäten statt, mit denen ein Sitzungsteilnehmer signalisiert, dass er z.B. eine Datei erstellt oder einen Ordner verschoben hat. Diese Funktionalität wird eindeutig auch im Sitzungsserver gebraucht. Jedoch greift `SharedResourcesManager` auf eine ho-

[20] Diese Schnittstelle hieß ursprünglich `ISessionListener`, wurde im Laufe dieser Arbeit aber umbenannt.

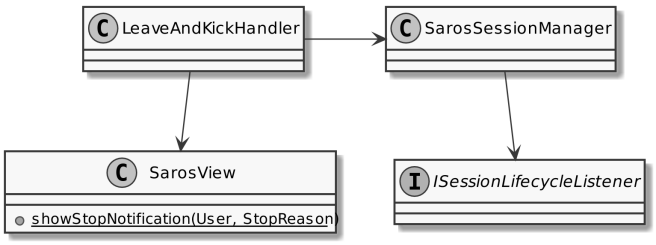

Abbildung 6: UML-Darstellung der problematischen Abhängigkeit von `LeaveAndKickHandler` zu `SarosView`.

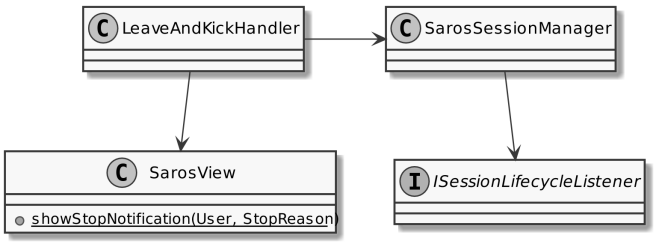

Abbildung 7: Die Abhängigkeiten nach den vorgenommenen Refaktorisierungen. Die problematische Abhängigkeit von `LeaveAndKickHandler` wurde in eine Abhängigkeit von `SarosView` verwandelt.

he Zahl an Eclipse-Schnittstellen zu, von denen einige noch keine Entsprechung im Kern haben, beispielsweise `IResourceChangeListener` zum Erkennen von lokalen Dateisystemänderungen. Da der Migrationsaufwand für den Rahmen dieser Arbeit zu groß gewesen wäre, wurde entschieden, die Bearbeitung von Dateisystem-Aktivitäten vorübergehend im Server getrennt zu implementieren.

Abgesehen davon könnte die Migration weiterer Nicht-Kern-Sitzungskomponenten, insbesondere `ConcurrencyWatchdogServer`, den Funktionsumfang des Sitzungsservers weiter erhöhen.

4.5 Zusammenfassung

Mit den für diese Arbeit durchgeführten Refaktorisierungen sind nun nahezu alle wesentlichen Aspekte der Sitzungsverwaltung und des Saros-Kommunikationsprotokolls im Kern implementiert. Hierdurch wurde nicht nur die Duplikation zwischen Saros/E und Saros/I weiter verringert, sondern auch die Notwendigkeit neuer Duplikation bei der Implementierung des unabhängigen Sitzungsservers minimiert. Dieser ist das Thema des nächsten Kapitels.

5 Implementation des unabhängigen Servers

Dieses Kapitel stellt das primäre Ergebnis dieser Arbeit vor – einen unabhängig von Eclipse oder IntelliJ lauffähigen Sitzungsserver für Saros. Hierfür wird zunächst der von Nils Bussas entwickelte, in Saros/E integrierte Server-Prototyp [Bus14] näher beschrieben, auf dessen Konzept und Code der neue Sitzungsserver aufbaut. Es folgt ein Einblick in den Entwurf und die Implementation des neu entwickelten Servers sowie seiner noch verbleibenden Probleme und Einschränkungen.

5.1 Entwurf und Implementation des Bussas-Server-Prototyps

Die generelle Funktionsweise des Server-Prototyps wurde bereits in Abschnitt 1.1.1 beschrieben. Kurz gesagt erlaubt er Nutzern, einen Server (d.h. einer Saros/E-Instanz, bei der zusätzlich die Server-Funktion aktiviert wurde) um eine Sitzungseinladung zu bitten. Einigt sich eine Gruppe von Nutzern also auf einen Server, kann sie sich so zu einer gemeinsamen Serversitzung zusammenfinden.

Als Grundlage für den Prototyp nahm Bussas die folgenden Erweiterungen am Saros-Protokoll vor [Bus14]:

- Einen zusätzlichen Rückgabewert für Antworten auf XMPP-*"Service Discovery"*-Anfragen (siehe Abschnitt 2.1.1), der Saros-Sitzungsserver als solche identifiziert. Dies erlaubt es, Server in den Saros-Benutzeroberflächen anders zu behandeln, sodass z.B. nur bei ihnen ein Menüpunkt zum Erbitten einer Sitzungseinladung erscheint.
- **Session Join Requests**, über die Nutzer einen Server um die Einladung zu einer Sitzung bitten können. Ein so genanntes `newSession`-**Flag** bestimmt dabei, ob der Server eine neue Sitzung erstellen oder in eine laufende Sitzung einladen soll. Kann der Server die Anfrage nicht erfüllen (siehe nächsten Abschnitt), sendet er eine **Session Join Rejection** zurück.
- **Session Status Requests**, mit denen Informationen zum Sitzungszustand eines Servers abgefragt werden können. Diese umfassen die Beantwortung der Frage, ob der Server zur Zeit eine Sitzung betreibt, sowie ggf. die Zahl der Sitzungsteilnehmer und die Namen der in der Sitzung geteilten Projekte.

Implementiert wurde diese Funktionalität hauptsächlich in Form zweier neuer Saros/E-Komponenten namens `JoinSessionRequestHandler` und

`SessionStatusRequestHandler`, in denen die neuen Nachrichtentypen verarbeitet werden, sowie Datentypen zur Repräsentation dieser Nachrichtentypen (`JoinSessionRequestExtension`, `SessionStatusRequestExtension`, usw.). Letztere sind bereits zu einem späteren Zeitpunkt in den Kern migriert worden, die Komponenten befinden sich jedoch wie ursprünglich in Saros/E.

5.1.1 Schwäche im Entwurf des Join Session Requests

Neben der Abhängigkeit von Eclipse und der sich daraus ergebenden Nachteile (siehe Abschnitt 1.1.1) ist eine weitere Schwäche des Bussas-Server-Prototypen die Gestaltung des Join Session Requests.

Da der Entwurf von Saros nur maximal eine laufende Sitzung pro Saros-Instanz zulässt, kann auch der Server nur eine Sitzung gleichzeitig betreiben. Dies macht den `newSession`-Flag, mit dem der anfragende Nutzer zwischen einer neuen und einer existierenden Serversitzung wählt, problematisch: wird der Server nach einer neuen Sitzung gefragt, während er sich bereits in einer befindet, muss er einen Fehler zurückgeben; das gleiche gilt, wenn der Server um die Einladung in eine laufende Sitzung gebeten wird, obwohl keine existiert. Zu jedem Zeitpunkt führt somit immer nur genau ein Wert des Flags zu einem erfolgreichen Join Session Request.

Diese Tatsache macht den `newSession`-Flag nicht nur redundant, sondern zu einer potentiellen Frustquelle, denn der anfragende Nutzer kann selbst bei vorheriger Abfrage des Serverzustands (per Session Status Request, siehe oben) nicht sicher vorhersagen, ob sich der Server bei Erhalt des Join Session Requests in einer Sitzung befinden wird oder nicht. Es kann beispielsweise sein, dass zwei Entwickler, die sich auf die Zusammenarbeit in einer neuen Serversitzung geeinigt haben, zur selben Zeit versuchen, diese auf dem gewählten Server zu erstellen; in diesem Fall würde einer der Entwickler eine Fehlermeldung erhalten und müsste – vermutlich zu seinem Ärger – in einem zweiten Schritt den Beitritt in die existierende Serversitzung wählen.

Um solche Probleme zu vermeiden, wurde der `newSession`-Flag beim neuen Sitzungsserver entfernt (siehe Abschnitt 5.3).

5.2 Anforderungen an den neuen Sitzungsserver

Das Hauptziel für den in dieser Arbeit entwickelten Sitzungsserver war, dessen praktische Nutzung im Vergleich zum Saros/E-basierten Prototyp

von Bussas deutlich zu erleichtern. Hieraus ließen sich folgende Anforderungen ableiten:

- Der Server sollte, anders als die anderen Saros-Varianten, nicht in eine Entwicklungsumgebung eingebettet sein, sondern als eigenständiger Prozess ohne grafische Oberfläche laufen. Dies reduziert die Anforderungen an die Ausführungsumgebung und erleichtert so die Auslieferung und Wartung des Servers in professionellen IT-Infrastrukturen.
- Der Server muss völlig nicht-interaktiv laufen, um den unbeaufsichtigten Betrieb sowie die vollautomatische Konfiguration und Auslieferung des Servers mit minimalem Aufwand zu ermöglichen. Entscheidungen, die in Saros/E und Saros/I durch Benutzereingaben gesteuert sind (z.B. die Wahl die Speicherorte für geteilte Projekte) müssen also automatisiert getroffen werden.
- Der Server sollte nur ein Minimum an Konfiguration benötigen, sodass er leicht in Betrieb genommen werden kann.

Diese Kriterien dienten als Leitlinien für den Entwurf des Sitzungsservers.

5.3 Entwurf

Schon früh fiel die Entscheidung, die von Bussas entworfenen Saros-Protokollerweiterungen wiederzuverwenden, da sie einen eleganten Ansatz für die Einführung von Serversitzungen darstellen und bereits über eine funktionierende, von der Saros-Entwicklergemeinde begutachtete Implementation verfügen. Zudem fanden sich bereits Teile des Codes im Saros-Kern, sodass diese direkt verwendet werden konnten. Die in Saros/E verbliebenen Komponenten hatten nur wenige Abhängigkeiten zur Eclipse-Plattform und konnten mittels der in Kapitel 4 beschriebenen Techniken leicht für den neuen Sitzungsserver angepasst werden.

Auf Protokollebene wurde lediglich der Join Session Request modifiziert. Er enthält nun nicht mehr das problematische `newSession`-Flag (siehe Abschnitt 5.1.1). Stattdessen beginnt der erhaltende Server nun automatisch eine neue Sitzung, wenn noch keine existiert, und verschickt sonst eine Einladung für die bestehende Sitzung. Auf diese Weise kann der Sitzungsserver einen Join Session Request immer erfüllen.

5.3.1 Betrieb und Konfiguration

Bei dem neuen Sitzungsserver handelt es sich um ein herkömmliches eigenständiges Java-Programm, das als JAR-Datei verpackt und dann beispielsweise auf der Kommandozeile mit `java -jar` ausgeführt werden kann. Beim Start baut der Server automatisch eine XMPP-Verbindung (siehe Abschnitt 2.1.1) mit dem konfigurierten Account und Passwort auf und wartet anschließend auf eintreffende Join Session Requests. Sobald der erste empfangen wurde, beginnt der Server eine neue Sitzung, die so lange läuft, bis der Server beendet wird (siehe Abschnitt 5.5).

Wie die anderen Saros-Varianten erstellt auch der Sitzungsserver lokale Kopien aller in der Sitzung geteilten Projekte, die er durch die Verarbeitung von Aktivitäten laufend aktualisiert. Der Ordner, in dem die Projekte abgelegt werden sollen, kann konfiguriert werden; wird er nicht angegeben, erstellt der Server beim Start einen temporären Ordner, den er bei Beendigung wieder löscht.

Für den Konfigurationsmechanismus fiel die Wahl auf die Nutzung von *JVM Properties*, die mittels der Option `-D` an den `java`-Befehl übergeben werden können (z.B. `java -Dkey=value ...`). Grund für diese Entscheidung war die einfache Nutzbarkeit und Implementation dieser Konfigurationsform. Außerdem wurden JVM Properties bereits andernorts im Saros-Projekt verwendet (z.B. zum Aktivieren experimenteller Funktionen wie dem Server in Saros/E), sodass ihre Verwendung konsistenter erschien als die Nutzung des ähnlich einfachen und plattformübergreifenden Mechanismus der Umgebungsvariablen. Auf die Nutzung einer Konfigurationsdatei wurde verzichtet, da dies zusätzlichen Implementationsaufwand erfordert hätte und ob der geringen Zahl an Konfigurationsparametern (XMPP-Adresse, XMPP-Passwort, Ordner für Projekte) kein nennenswerter Mehrwert zu erkennen war.

Der Sitzungsserver läuft, wie für Serversoftware üblich, endlos, solange er nicht von außen beendet wird (beispielsweise durch Senden eines `TERM`-Signals in Unix-Systemen) oder abstürzt.

5.3.2 Automatisierung

Um den Server autonom lauffähig zu machen (siehe Anforderungen in Abschnitt 5.2), wurden einige Entscheidungen automatisiert, die in den anderen Saros-Varianten Benutzereingaben erfordern.

- Anfragen, den Account des Servers in die Kontaktliste eines Nutzers aufnehmen zu dürfen, bestätigt der Server automatisch.

- Von Sitzungsteilnehmern angebotene Projekte werden automatisch angenommen und in einem Unterordner des konfigurierten Speicherortes (siehe vorherigen Abschnitt) abgelegt, dessen Name dem Projektnamen entspricht.

Für diese erste Version des Servers wurde bewusst die einfachstmögliche Lösung für jede dieser Entscheidungen gewählt. So konnte die Implementation von Konfigurationsmöglichkeiten für diese Automatismen vermieden werden, die sich in der Praxis möglicherweise als unnötig oder ungenügend herausstellen. Stattdessen ist es sinnvoller, abzuwarten, bis genügend Erfahrungen für die Gestaltung geeigneter Konfigurationsparameter gesammelt wurden.

5.4 Implementation

Der Sitzungsserver baut auf dem für diese Arbeit erweiterten Saros-Kern (siehe Kapitel 4) auf und besteht im wesentlichen aus drei Teilen:

- Notwendige *Implementationen von Kern-Schnittstellen*, die nicht innerhalb des Kern implementiert sind. Hierzu gehört unter anderem die Serverversion der vom Kern bereitgestellten Dateisystem-Abstraktion, die in Abschnitt 5.4.1 näher beschrieben ist. Einige Schnittstellen sind lediglich als funktionslose "Dummies" implementiert, da sie zwar vom Kern erwartet, aber für den Server nicht benötigt werden, wie z.B. im Fall von `IRemoteProgressIndicator` (siehe Abschnitt 4.3.1).
- *Server-spezifische Komponenten*. Bei diesen handelt es sich hauptsächlich um Adaptionen des Server-Codes von Bussas sowie Komponenten zum Verarbeitung von Dateisystem-Aktivitäten, da diese Funktionalität noch nicht in den Kern migriert werden konnte (siehe 4.4).
- Eine *Hauptklasse* zur Initialisierung des Servers.

Der Großteil dieses Codes befindet sich zum Zeitpunkt dieser Arbeit im Gerrit-System (Abschnitt 2.2) als Change 2929[21]. Dieser "Superchange" ist nicht für die direkte Übernahme in den Hauptzweig von Saros gedacht, sondern dient hauptsächlich als einfache Möglichkeit, den Server vor dessen vollständiger Integration auszuprobieren. Aus diesem Grund enthält der Change auch die Änderungen von Change 2264 für Non-Host Project

[21] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2929/

Sharing (Abschnitt 3.5) sowie Change 2062 von Ute Neise[22], der Saros/E um einen Menüeintrag zum Senden von Session Join Requests erweitert.

Für die Zukunft ist geplant, einzelne Bestandteile von Change 2929 in Form kleinerer Changes zu veröffentlichen, die isoliert begutachtet und in den Hauptzweig integriert werden können. Der große Server-Commit kann so zunehmend geschrumpft und schließlich völlig verworfen werden.

Insgesamt besteht der Server zum Zeitpunkt dieser Arbeit aus ca. 1500 Zeilen Produktivcode sowie Unit-Tests mit einem Umfang von ca. 1000 Zeilen.[23]

5.4.1 Umsetzung der Dateisystem-Abstraktion

Der Saros-Kern definiert einen Satz an Schnittstellen, mit denen Projekte und Projektdateien repräsentiert sowie verschiedene Operationen auf diesen (z.B. Datei erstellen oder Ordner verschieben) angeboten werden. Diese Schnittstellen sind in Saros/E und Saros/I mit den entsprechenden Dateisystem-Abstraktionen der Entwicklungsumgebung implementiert und erlauben es dem Kern so, diese Abstraktionen zu nutzen, ohne eine Abhängigkeit zu Eclipse oder IntelliJ zu benötigen.

Die Dateisystem-Schnittstellen des Kerns sind den von Eclipse angebotenen nachempfunden.[24] Sie bilden eine Hierarchie, deren oberste Ebene der **Workspace** (Schnittstelle `IWorkspace`) darstellt. Dieser dient als Behälter für Projekte (`IProject`), die wiederum eine Hierarchie aus Ordnern (`IFolder`) und Dateien (`IFile`) beherbergen. `IProject`, `IFolder` und `IFile` sind alle Subtypen der Schnittstelle `IResource`, die das übergeordnete Konzept der **Ressource** – eines Objektes im Dateisystem – repräsentiert. Außerdem leiten sich `IFolder` und `IProject` vom `IResource`-Subtyp `IContainer` ab, einer Schnittstelle zur Repräsentation von Ressourcen, die ihrerseits Ressourcen enthalten können. Zusätzlich ist noch die Schnittstelle `IPath` für Dateisystem-Pfade definiert.

Der Sitzungsserver implementiert diese Schnittstellen unter direkter Nutzung der Dateisystem-Klassen in der Java-Standardbibliothek (aus den Pa-

[22] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2062/

[23] Diese Zahlen wurden mit dem Tool `sloccount` (http://www.dwheeler.com/sloccount/) anhand des Saros-Repository-Stands von Patch Set 9 des Gerrit-Patches 2929 (http://saros-build.imp.fu-berlin.de/gerrit/#/c/2929/9/) gemessen.

[24] Diese Tatsache stellt ein großes Kompatibilitätsproblem dar, da IntelliJ über ein grundlegend anderes Projektmodell verfügt, das sich nicht nahtlos auf die Konzepte von Eclipse übertragen lässt. Aus diesem Grund können Saros/E und Saros/I nicht problemlos in derselben Sitzung verwendet werden. Der Sitzungsserver wurde nur mit Saros/E getestet.

keten `java.io` und `java.nio.file`) sowie der darauf aufbauenden Hilfsbibliothek *Apache Commons IO*[25]. Der Workspace wird durch einen beliebigen Ordner im Dateisystem dargestellt, der entweder durch die Konfiguration des Servers gegeben oder vom Server selbst erstellt wird (siehe Abschnitt 5.3.1). Die unmittelbaren Unterordner repräsentieren die lokalen Projekte und werden nach den Projektnamen benannt. Die Ressourcen eines Projektes werden ihrer Hierarchie entsprechend in Unterordnern und Dateien innerhalb des Projektordners abgelegt.

Eine Schwierigkeit bei der Implementation der Dateisystem-Abstraktion war, dass für die meisten ihrer Methoden keinerlei erwartetes Verhalten spezifiziert war. Stattdessen wurde auf die Dokumentation der äquivalenten Methoden der Eclipse-Plattform verwiesen, ohne einzuschränken, welche der dort beschriebenen Verhaltensaspekte tatsächlich für die Korrektheit von Saros bedeutend sind. So war es nötig, in allen Randfällen das in der Eclipse-Dokumentation spezifizierte Verhalten zu simulieren, auch wenn der Fall möglicherweise in Saros nie auftreten kann. Eine klare Dokumentation der Dateisystem-Methoden im Kern, jeweils mit expliziter Nennung der speziell für Saros nötigen Semantik, hätte den Aufwand für die Implementation möglicherweise verringern können.

5.4.2 Komponenten aus dem Bussas-Server-Prototyp

Die von Bussas geschriebene `JoinSessionRequestHandler`-Komponente wurde in die Codebasis des Sitzungsservers kopiert und gemäß der in Abschnitt 5.3 beschriebenen Änderung des Join Session Requests angepasst. Zusäztlich wurde die in diesem Kontext überflüssinge Prüfung, ob die Saros-Instanz als Server konfiguriert ist, sowie Abhängigkeiten zu Eclipse und der Nicht-Kern-Klasse `CollaborationUtils` zugunsten existierender Kern-Schnittstellen entfernt.

`SessionStatusRequestHandler` wurde zum Zeitpunkt dieser Arbeit noch nicht portiert.

5.5 Offene Probleme und Einschränkungen

Der entwickelte Sitzungsserver ist bereits funktionsfähig, hat jedoch zum Zeitpunkt dieser Arbeit noch mehrere bekannte Probleme und Beschränkungen:

- Wie der Rest von Saros ist auch der Server auf maximal eine laufende

[25] https://commons.apache.org/proper/commons-io/

Sitzung beschränkt. Er kann also nicht, wie es wünschenswert wäre, mehrere Sitzungen gleichzeitig betreiben.

- Da der Server bei Beginn einer Sitzung zu dessen Host wird und die Sitzung nie verlässt, wird die Sitzung auch nie beendet. Um also einen Server mit einer laufenden Sitzung für die Zusammenarbeit an anderen Projekten als den zuvor dort geteilten zu verwenden, muss zunächst der Sitzung beigetreten, die vorher geteilten Projekte entgegengenommen und diese schließlich aus der Sitzung entfernt werden. Einfacher wäre beispielsweise eine spezielle Nachricht, mit der ein Server angewiesen werden könnte, die aktuell laufende Sitzung zu beenden und die zugehörigen Projektdateien zu löschen.

- Wie erwähnt nutzt der Server die Namen der empfangenen Projekte als Ordnernamen für ihre lokalen Kopien. Teilen zwei Sitzungsteilnehmer jeweils ein Projekt mit demselben Namen, kommt es also zu einem Konflikt auf dem Server. Hier wäre ein Algorithmus zur automatischen Wahl konfliktfreier lokaler Projektnamen (z.B. durch Anhängen einer fortlaufenden Nummer) sinnvoll, wie er bereits in Saros/E und Saros/I zum Einsatz kommt.

- Jede Textänderung, die der Server von einem Sitzungsteilnehmer bei Bearbeitung einer geöffneten Datei erhält, speichert der Server unmittelbar im Dateisystem. Korrekter und effizienter wäre es, die Änderungen nur im Hauptspeicher vorzunehmen und erst dann ins Dateisystem zu schreiben, wenn eine *Speicher-Aktivität* für die betreffende Datei erhalten wird.

- Der Server verfügt nicht über den *Consistency Watchdog* (Abschnitt 2.1.5), sodass eine erhöhte Gefahr für unerkannte Inkonsistenzen in den Projektkopien der Sitzungsteilnehmer besteht. Jedoch ist die Migration des Watchdogs bereits im Zuge der Arbeit von Sungalia im Gange [Sun].

- Die Server-Komponenten sind noch in Saros/E dupliziert und nicht an die Entfernung des `newSession`-Flags im Join Session Request angepasst. Sie sollten in Zukunft zusammen mit der Server-Option aus Saros/E entfernt werden.

Während die letztgenannten Punkte relativ leicht zu behandeln sind, ist die Annahme einer einzelnen laufenden Sitzung so tiefgreifend in die Codebasis von Saros eingebettet, dass die Aufhebung dieser Beschränkung

einen erheblichen Entwurfs- und Implementationsaufwand darstellen würde. Ein alternativer Ansatz wäre die Einführung eines *Mehrprozessmodells*, bei dem für jede gewünschte Serversitzung dynamisch eine neue Instanz des Sitzungsservers gestartet wird; diese Idee wird am Ende dieser Arbeit in Abschnitt 7.3 näher skizziert.

5.6 Zusammenfassung

Das Ziel, einen funktionierenden, unabhängig lauffähigen Sitzungsserver für Saros zu entwickeln, hat diese Arbeit erfüllt, auch wenn bis zum Erreichen der Produktionsreife noch einige Probleme zu beheben sind – sowohl bekannte als vermutlich auch unbekannte. Schon jetzt ist der Server jedoch dazu bereit, in der Praxis getestet zu werden, sodass die daraus gewonnenen Erfahrungen die Prioritäten für die Weiterentwicklung bestimmen können.

Um sich im Produktiveinsatz zu bewähren, muss der neue Server nicht nur generell funktionieren, sondern auch unter lang anhaltender Last verantwortungsvoll mit Systemressourcen umgehen, um Versagen zu vermeiden. Die Untersuchung dieses Kriteriums ist Gegenstand der im nächsten Kapitel vorgestellten Evaluation.

6 Evaluation der Sitzungsserver-Ressourcenverwaltung

Es ist zu erwarten, dass Serversitzungen sehr viel länger laufen werden, als dass bei klassischen host-basierten Sitzungen der Fall ist, denn sie laufen auch nach dem Austritt aller ursprünglichen Teilnehmer weiter und können beliebig oft wieder aufgenommen werden. Aus diesem Grund ist es für den Produktiveinsatz entscheidend, dass sowohl der Sitzungsserver selbst als auch der darunterliegende Saros-Kern zuverlässig mit Systemressourcen wie Hauptspeicher und offenen Dateien umgeht, sodass Ressourcenlecks und die sich daraus ergebenden Probleme im Langzeitbetrieb (z.B. Abstürze durch Hauptspeicherknappheit) nicht auftreten.

Dieses Kapitel stellt einen ersten Versuch zur Evaluation des Sitzungsservers hinsichtlich dieses Kriteriums vor. Ziel war hierbei die Überprüfung des Servers auf das Vorhandensein praxisrelevanter Speicher- und anderer Ressourcenlecks, die eine Gefahr für die dauerhafte Verfügbarkeit des Servers darstellen. Aus Zeitgründen konnten jedoch nur bescheidene Ergebnisse erzielt werden.

6.1 Vorgehen

Für die Suche nach Ressourcenlecks wurden auf zwei verschiedene Analyseverfahren zurückgegriffen:

- **Laufzeitanalyse** des Servers bei der wiederholten Verarbeitung bestimmter Eingaben über einen längeren Zeitraum. Der Fokus lag hier beim Aufspüren von Speicherlecks.
- **Statische Analyse** des Server- und Kern-Codes mithilfe von Analysewerkzeugen, die potenzielle Ressourcenlecks anhand problematischer Codemuster erkennen (z.B. Aufrufe einer `close()`-Methode außerhalb eines `finally`-Blocks).

Diese beiden Techniken erfüllten verschiedene Zwecke. Die statische Analyse ist nur zur Erkennung einfacher Lecks mit explizit freizugebenden Ressourcen (z.B. offenen Dateien) fähig, erlaubt dafür aber die Überprüfung der gesamten Codebasis mit minimalem Aufwand. Bei der Laufzeitanalyse war der Aufwand hingegen sehr viel höher und die Betrachtung auf wenige konkrete Szenarien beschränkt, dafür ist es mit der Untersuchung des Laufzeitverhaltens möglich, sehr viel komplexere Ressourcenlecks und insbesondere auch Speicherlecks zu finden.

Ursprünglich war auch eine ergänzende manuelle Codeanalyse geplant, bei der kritische ressourcenverwaltende Komponenten u.a. durch Befragung erfahrener Saros-Entwickler identifiziert und auf das Vorhandensein von Ressourcenlecks untersucht werden sollten. Aus Zeitgründen musste hierauf jedoch verzichtet werden.

6.1.1 Laufzeitanalyse: Lasttests und Heap-Messungen

Für die Laufzeitanalyse wurden insgesamt zwei verschiedene Lasttests entwickelt, die jeweils eine bestimmte Funktionalität des Sitzungsservers nutzen. Diese wurden mithilfe der Saros-internen Testbibliothek *STF* (*Saros Test Framework*) programmiert und so geschrieben, dass ihr "logischer" Speicherbedarf über die Zeit nicht steigt.

- In `ServerEditingStressTest` treten zwei Nutzer der Sitzung des Servers bei und bearbeiten gleichzeitig eine einzelne Datei in einem geteilten Projekt. In jedem Bearbeitungsschritt wird dabei eine zufällig gewählte Anzahl Zeichen durch die gleiche Anzahl neuer Buchstaben ersetzt, sodass die Datei zu jeder Zeit ungefähr lang bleibt.
- Bei `ServerJoinLeaveStressTest` tritt ein einzelner Nutzer wiederholt der Sitzung des Servers zu und verlässt diese wenige Sekunden später wieder. Da der Nutzer immer derselben Sitzung beitritt, sollte auch hier der Speicherbedarf nicht steigen.

Diese beiden Tests wurden über einen Zeitraum von jeweils ca. acht Stunden ausgeführt, damit Langzeiteffekte erkennbar werden.

Während der Testausführung wurde der Sitzungsserver mithilfe der "*Flight Recording*"-Funktion des Werkzeugs *Java Mission Control*[26] überwacht. Dabei wurden unter anderem die Menge des genutzten JVM-Heap-Speichers sowie Statistiken über die Speicherbereiningungsdurchläufe des *Garbage Collectors* aufgenommen.

Anschließend wurde die Aufnahme auf Auffälligkeiten bei der Speichernutzung untersucht. Gemäß der Hinweise von Oracle zur Speicherlecksuche [JMC] wurden dabei nur die Heapzustände nach vollen Durchläufen des Garbage Collectors (**Livesets**) betrachtet, da die Analyse sonst durch nicht mehr referenzierte ("tote") Objekte verfälscht werden würde. Wurde ein stetiges Wachstum von Liveset zu Liveset festgestellt, ließ dies auf ein mögliches Speicherleck schließen.

[26] http://www.oracle.com/technetwork/java/javaseproducts/mission-control/

In diesem Fall kam zusätzlich die "*Heap Dump*"-Funktion des Werkzeugs *VisualVM*[27] zum Einsatz. Mit ihr kann von einem laufenden JVM-Prozess ein Heap-Abbild erstellt und die darin befindlichen Objekte inspiziert werden. Außerdem können zwei Heap-Abbilder verglichen werden, um herauszufinden, welche Objekte in der Zwischenzeit erzeugt oder gelöscht wurden. So war es möglich, auffällige Ansammlungen von Objekten einer bestimmten Klasse zu finden, die Anhaltspunkte für mögliche Quellen eines Speicherlecks geben. Diese wurden dann durch gezielte Begutachtung der Saros-Codebasis untersucht.

6.1.2 Statische Analyse: FindBugs und PMD

Für die statische Analyse kamen die Werkzeuge *FindBugs*[28] 3.0.1 und *PMD*[29] 5.4.1 Einsatz. Beide besitzen eingebaute Überprüfungsregeln, die einfache Arten von Lecks bezüglich Streams, Sockets und ähnlichen manuell freizugebenden Ressourcen automatisch aufspüren können; all diese wurden hier aktiviert. Das gilt auch für die experimentelle Regel `OBL_UNSATISFIED_OBLIGATION` von FindBugs[30].

Die Verwendung von FindBugs erscheint zunächst überflüssig, da es bereits zur automatischen Überprüfung neu eingesandter Gerrit-Changes (siehe Abschnitt 2.2) verwendet wird. Allerdings sind dort die experimentellen Regeln nicht aktiv, sodass die Möglichkeit zuvor nicht entdeckter Probleme bestand.

6.2 Beobachtungen und Ergebnisse

Die folgenden Abschnitte beschreiben den Hergang und die Ergebnisse der durchgeführten Analysen.

6.2.1 Ergebnisse der Laufzeitanalyse

Bei der Ausführung von `ServerJoinLeaveStressTest` wurden die in Abbildung 8 gezeigten Liveset-Größen gemessen. Ein signifikanter Anstieg konnte nicht beobachtet werden. Auch Heap-Abbilder förderten keine Auffälligkeiten zutage. Lediglich ein geringer Anstieg der Größe einer Klasse namens `ArrayNotificationBuffer` war zu erkennen, die jedoch Teil der

[27] https://visualvm.java.net/
[28] http://findbugs.sourceforge.net/
[29] https://pmd.github.io/
[30] http://findbugs.sourceforge.net/bugDescriptions.html#OBL_UNSATISFIED_OBLIGATION

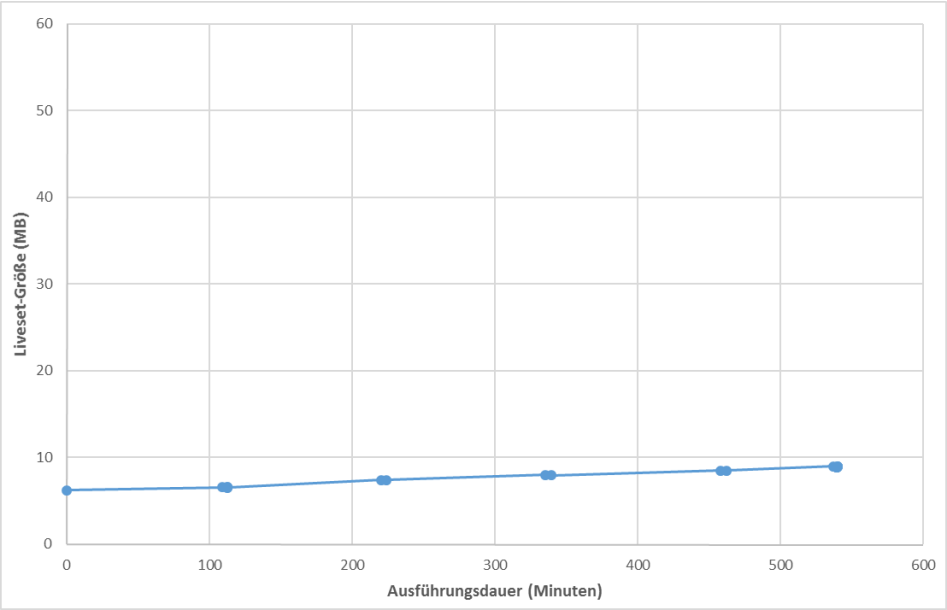

Abbildung 8: Gemessene Liveset-Größen bei `ServerJoinLeaveStressTest`.

von Java Mission Control zur Messung genutzten *Java Management Extensions (JMX)*[31] ist und somit nicht zu Saros gehört. Der Sitzungsserver scheint diese Aufgabe also ohne Speicherlecks bewältigen zu können.

Bei `ServerEditingStressTest` ergab sich hingegen ein anderes Bild (Abbildung 9). Hier ist ein klarer Aufwärtstrend bei der der Größe der Livesets zu beobachten. Heap-Abbilder offenbarten eine hohe und ständig wachsende Menge an Instanzen der Klasse `JupiterActivity` zur Repräsentation von Jupiter-Textbearbeitungsoperationen (siehe Abschnitt 2.1.4). Das erschien merkwürdig, da Aktivitätsobjekte nach ihrer Verarbeitung üblicherweise nicht mehr referenziert werden und daher vom Garbage Collector freigegeben werden müssten.

Die Untersuchung eines Abbilds auf Referenzen zu den Jupiter-Aktivitäten ergab, dass diese in Listen gesammelt wurden, die wiederum innerhalb von Objekten der Klasse `Jupiter` als `ackJupiterActivityList` abgelegt waren. Wie eine Recherche des Jupiter-Algorithmus [NCDL95] ergab, hing dies damit zusammen, dass der Sender einer Textbearbeitungsoperation diese so lange zwischenspeichern muss, bis der Empfänger die Anwendung der Operation bestätigt hat. Andernfalls wäre es bei Erhalt einer Operation unmöglich zu bestimmen, ob – und wenn, wie – diese transformiert werden muss, falls das Gegenüber noch nicht alle an ihn gesendeten

[31] http://www.oracle.com/technetwork/articles/java/javamanagement-140525.html

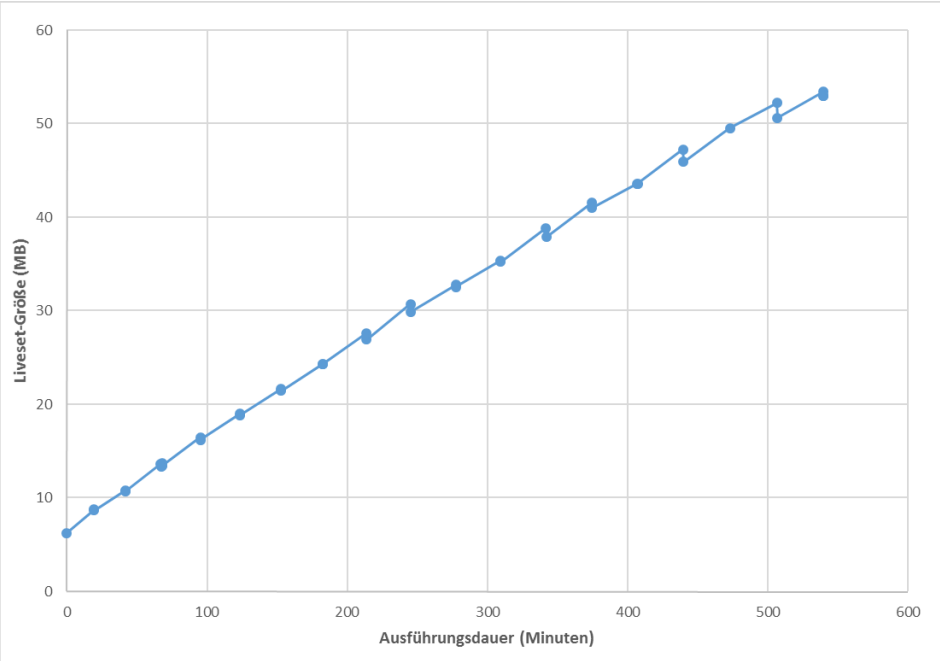

Abbildung 9: Gemessene Liveset-Größen bei `ServerEditingStressTest`.

Operationen angewendet hat. Dieses Wissen ist besonders für den Jupiter-Server entscheidend, da er jede von einem Klienten gemeldete Operation für den Zustand jedes anderen Klienten transformieren können muss, um sie weiteleiten zu können.

In der Saros-Implementation des Jupiter-Algorithmus können Operationen ausschließlich dadurch bestätigt werden, dass der Empfänger seinerseits eine neue Operation in Form einer `JupiterActivity` versendet. Darin gespeichert ist die Zahl der bisher empfangenen und angewendeten Operationen der Gegenseite (**Remote Operation Count**); diese kann so darauf schließen, welche Operationen nicht mehr zwischengespeichert werden müssen. Diese Art der Bestätigung ist ausreichend, wenn alle Sitzungsteilnehmer regelmäßig Bearbeitungsoperationen durchführen; sobald das ein Teilnehmer aber nicht tut, beispielsweise weil er nur als Gutachter an einer Sitzung teilnimmt, werden die an ihn gesendeten Operationen nie bestätigt und der Sender (sprich, der Host in seiner Rolle als Jupiter-Server) kann sie folglich auch nie löschen, sodass sie sich immer weiter ansammeln.

Dieses Problem ist für den Sitzungsserver besonders gravierend, da er es in jeder Serversitzung selbst auslöst: als Host ist er sowohl Jupiter-Server als auch -Klient (siehe Abschnitt 2.1.4), führt durch seine Rolle als passiver

Sitzungsteilnehmer aber keine eigenständigen Operationen durch, sodass die von seinem "Serverteil" an den "Kliententeil" weitergeleiteten Operationen nie bestätigt werden. Im Kontext von Serversitzungen handelt es sich also nicht um einen Randfall, sondern ein inhärentes Speicherleck, dass im Langzeitbetrieb unweigerlich zu einem `OutOfMemoryError` auf Seiten des Sitzungsservers führen würde.

Aus Zeitgründen konnte keine Lösung für die Behebung des Speicherlecks entwickelt werden. Mögliche Ansätze werden in Abschnitt 6.3.1 diskutiert.

Weitere Auffälligkeiten konnten in der Aufnahme von `ServerEditingStressTest` nicht entdeckt werden.

6.2.2 Ergebnisse der statischen Analyse

FindBugs meldete bei der Analyse des Sitzungsservers und Kerns kein Proble, die auf mögliche Ressourcenlecks hinweisen. Die experimentellen Regeln lieferten keine zusätzlichen Ergebnisse gegenüber den stabilen, vom Saros-Projekt genutzten.

Die Analyse mit PMD ergab insgesamt fünf relevate Problemmeldungen. Diese bezogen sich alle auf Codestellen, die eine Instanz der Kern-Schnittstelle `Connection` erwerben, aber nicht schließen (*"Ensure that resources like this Connection object are closed after use"*). An jede dieser Stellen wurde die Methode `getConnection` der Komponente `ConnectionService` aufgerufen, die für die Verwaltung der XMPP-Verbindung (siehe Abschntt 2.1.1) zuständig ist. Da `getConnection` keine neue Verbindung erstellt, sondern lediglich die existierende, von `ConnectionService` verwaltete zurückgibt (wenn vorhanden), kommt dem Aufrufer folglich nicht Aufgabe des Schließens zu. Es handelten sich bei den Meldungen von PMD also um *false positives*.

6.3 Einschränkungen und Ausblick

Die hier vorgestellte Evaluation ist aus Zeitgründen nur oberflächlich geblieben. Zum einen können die genutzten statischen Analysen aufgrund ihrer Natur nur sehr einfache, lokale Arten von Ressourcenlecks aufdecken, die sich anhand von Codemustern und grundlegender Kontrollflussanalyse erkennen lassen; eine gezielten manuelle Codeanalyse hätte vermutlich aussagekräftigere Ergebnisse geliefert. Zum anderen wurde eine gründliche Laufzeitanalyse nur für einen kleinen (wenn auch bedeutenden) Bruchteil der Funktionalität von Saros unternommen. Unter dem Strich wurden

zu wenige Informationen gesammelt, um eine sinnvolle Aussage über die Qualität der Ressourcenverwaltung in der Saros-Codebasis treffen zu können.

Um in Zukunft ein besseren Eindruck mit vertretbarem Aufwand erhalten zu können, wäre die empirische Ermittlung eines **Benutzungsmodells** [WR94] für Saros-Sitzungen sinnvoll, das Aussagen über übliche Teilnehmerhandlungen und Handlungsreihenfolgen zulässt. Auf Basis eines solchen Modells ließe sich dann ein realistischer Lasttest entwickeln, mit dem sich feststellen ließe, ob und wenn welche Ressourcenlecks bei "normaler" Nutzung auftrteten. Träten hier nach Behebung der gefundenen Probleme (deren Praxisrelevanz direkt aus der Beschaffenheit des Tests folgt) keine Lecks mehr auf, wäre dies bereits ein starker Hinweis auf die Korrektheit der Ressourcenverwaltung im Sitzungsserver und Saros-Kern.

6.3.1 Behebung des Jupiter-Speicherlecks

Das Problem der fehlenden Jupiter-Operationsbestätigung durch inaktive Sitzungsteilnehmer (siehe Ergebnisse in Abschnitt 6.2.1) könnte am einfachsten durch die Einführung einer eigenständigen Bestätigungsaktivität gelöst werden. Auf diese Weise könnte ein Teilnehmer, der nach Erhalt einer Jupiter-Operation für eine festgelegte Zeit (z.B. 10 Sekunden) keine eigenen Änderungen vornimmt, dennoch die Anwendung der Operation bestätigen, sodass der Sender diese nicht mehr zwischenzuspeichern braucht. Da die eigenständigen Bestätigungen das existierende Protokoll lediglich ergänzen würden und nur im Fall von Inaktivität versendet werden müssten, ist mit einer nur geringen Erhöhung des Netwerkverkehrs zu rechnen.

Ein Nachteil dieses Ansatzes ist, dass der Host auf die Kooperation der Sitzungsteilnehmer vertrauen müsste, die durch den Verzicht auf Bestätigungen bewusst ein Speicherleck beim Host erzeugen könnten. Dies ist jedoch aus folgenden Gründen unproblematisch:

- Das Saros-Kommunikationsprotokoll verlangt schon in seiner jetzigen Form das Vertrauen der Sitzungsteilnehmer untereinander. Beispielsweise könnte ein Teilnehmer bei jeder gesendeten Jupiter-Aktivität den *Remote Operation Count* (siehe Abschnitt 6.2.1) als 0 angeben, sodass der Host ebenfalls zum unbegrenzten Zwischenspeichern gesendeter Operationen gezwungen ist.

- Im warscheinlichsten Speicherleck-Szenario ist der inaktive Teilnehmer der Sitzungsserver, also der Host selbst. Hier ist kein Vertrauen in einen externen Akteur nötig.

Die vorgestellte Lösung brächte gegenüber dem jetzigen Protokoll also keine zusätzlichen Probleme und wäre für Serversitzungen eine strikte Verbesserung.

6.4 Zusammenfassung

Leider ist es auf Basis dieser punktuellen Evaluation nicht möglich zu sagen, ob der Sitzungsserver in punkto Ressourcenverwaltung für den Langzeitbetrieb geeignet ist. Es war aber immerhin möglich, ein praxisrelevantes Speicherleck zu identifizieren und dessen Behebung zu skizzieren. Außerdem konnten zumindest einfache Ressourcenlecks, wie sie von üblichen statischen Analysewerkzeugen erkannt worden wären, ausgeschlossen werden, was vermutlich auf den Begutachtungsprozess des Saros-Projekts zurückzuführen ist. Um stärkere Aussagen treffen zu können, wären aber noch mehrere weitergehende Tests und Analysen notwendig.

Die Evaluation des Sitzungsservers bildete den Abschluss dieser Arbeit. Das nächste Kapitel fasst das Erreichte nochmal zusammen und gibt Vorschläge für zukünftige Arbeiten.

7 Fazit und Ausblick

In diesem letzten Kapitel soll ein abschließender Überblick über das in dieser Arbeit Erreichte sowie Anknüpfpunkte für weitergehende Arbeiten gegeben werden. In diesem Zusammenhang wird außerdem ein Vorschlag für einen Mehr-Prozess-Sitzungsserver skizziert, der eine elegante Lösung für das Problem der Begrenzung auf eine laufende Sitzung pro Saros-Instanz bieten könnte.

7.1 Ergebnisse dieser Arbeit

Die Ziele dieser Arbeit wurden größtenteils erreicht. Insgesamt wurden folgende Ergebnisse erzielt:

- Mit *Non-Host Project Sharing* (siehe Kapitel 3) konnte ein Mechanismus zum Teilen von Projekten in Serversitzungen geschaffen werden, der auf der bereits existierenden Project Negotiation basiert und keine bekannten neuen Konsistenz- oder Nebenläufigkeitsprobleme in das Kommunikationsprotokoll von Saros einführt.
- Durch die Migration zentraler Klassen wurde die Duplikation zwischen Saros/E und Saros/I reduziert sowie die Notwendigkeit neuer Duplikation bei der Implementierung des unabhängigen Sitzungsservers reduziert.
- Es konnte die erste Version eines von Eclipse unabhängigen Servers für Saros-Sitzungen entwickelt werden, dessen einfache Konfigurierbarkeit und nicht-interaktive Arbeitsweise ihn praktikabel für die automatisierte Auslieferung und Wartung in gängigen IT-Infrastrukturen macht.
- Die Evaluation des Sitzungserver förderte einen besonders für Serversitzungen relevantes Speicherleck-Defekt zutage, konnte dem Ziel einer Beurteilung hinsichtlich Eignung für den Langzeitbetrieb jedoch nicht gerecht werden.

7.2 Offene Fragen und Probleme

In dieser Arbeit wurde bereits an mehreren Stellen auf verbleibende Probleme und Möglichkeiten für zukünftige Weiterentwicklungen hingewiesen. Hier eine Zusammenfassung der wesentlichen Punkte:

- Non-Host Project Sharing ist in seiner derzeitigen Form unnötig ineffizient und könnte durch einige Anpassungen an die Geschwindigkeit des host-initiierten Projektteilens angeglichen werden (Abschnitt 3.7.1). Außerdem sind die Kompatibilität des Protokolls mit der von David Damm entwickelten aktivitätsbasierten Project Negotiation noch zu klären (Abschnitt 3.7.2). Zuguterletzt sollte für das Nicht-Host-Projektteilen ein automatisierter Test entwickelt werden (Abschnitt 3.6).
- Der Code für die Verarbeitung von Dateisystem-Aktivitäten ist noch zwischen Saros/E, Saros/I und dem neuen Server dupliziert und sollte in den Kern migriert werden (Abschnitt 4.4).
- Der neue Sitzungsserver befindet sich noch in einem frühen Reifestadium und hat noch mehrere bekannte Probleme (u.a. die fehlende Behandlung von Projektnamenkonflikten), deren Behebung die Eignung des Servers für den Produktiveinsatz erhöhen würden (Abschnitt 5.5).
- Das gefundene Speicherleck ist noch zu beheben (Abschnitt 6.3.1). Durch Erstellung eines realistischen Saros-Benutzungsmodells und eines darauf basierenden Lasttests könnte effektiv nach weiteren praxisrelevanten Ressourcenleck gesucht und die Gesamteignung für den Langzeitbetrieb besser beurteilt werden (Abschnitt 6.3).

Ein Aufgabenbereich, der in dieser Arbeit nicht behandelt wurde, ist die Integration des Serverkonzepts in den Benutzeroberflächen der Saros-Plugins. Zum Zeitpunkt dieser Arbeit existiert hierzu lediglich der in Abschnitt 5.4 genannte Change 2062 von Ute Neise, der die Kontaktliste von Saros/E um einen Kontextmenüeintrag zum Beitreten einer Serversitzung erweitert. Jedoch sind noch weitere Anpassungen nötig; beispielsweise gibt die Kontaktliste zur Zeit keinen optischen Hinweis, ob sich hinter einem Kontakt ein Server oder ein menschliche Nutzer verbirgt.

7.3 Idee: Ein Saros-Superserver

Eine wesentliche Einschränkung des entwickelten Sitzungsserver ist die vom Saros-Kern geerbte Eigenschaft, nur eine einzelne Sitzung zur selben Zeit betreiben zu können (siehe Abschnitt 5.5). Eine einfache Art, dieses Problem zu umgehen, wäre der Start eines separaten Sitzungsservers für jede gewünschte Serversitzung. Dies manuell zu tun, ist jedoch wenig komfortabel.

Ein vielversprechende Ansatz, um diese Idee praktikabel zu machen, ist die Einführung eines Saros-**Superservers**, dessen Konzept dem klassischen UNIX-Dienst `inetd` ähnelt (siehe [TvS02, Abs. 3.4.1]). Statt den Sitzungsserver direkt zu auszuführen, würde ein Administrator stattdessen den Superserver konfigurieren und starten, der daraufhin unter der übergebenen XMPP-Adresse (siehe Abschnitt 2.1.1) auf Anfragen für neue Serversitzungen wartet. Erhält er eine solche, startet er eine neue Instanz des Sitzungsservers und instruiert diesen dabei durch entsprechende Parameter, den anfragenden Nutzer zu einer Sitzung einzuladen. Ab diesem Punkt können der Nutzer und die Sitzungsserver-Instanz dann wie gewohnt direkt miteinander kommunizieren.

Eine solche Mehr-Prozess-Serverarchitektur würde dieselbe Funktionalität und Benutzerfreundlichkeit wie ein einzelner Serverprozess mit Unterstützung für mehrere Sitzungen bieten, ohne einen tiefgreifenden Umbau der Saros-Architektur zu erfordern. Es wären lediglich die Implementation des Superservers und zugehörige Anpassungen am Sitzungsserver nötig. Jedoch wären noch einige Entwurfsfragen zu klären, unter anderem:

- **Wie werden die XMPP-Adressen der dynamisch gestarteten Sitzungsserver gebildet?** Die Nutzung eines separaten XMPP-Accounts für jeden Sitzungsserver ist nicht praktikabel, da XMPP-Server aus Sicherheitsgründen selten die automatisierte Registrierung und Löschung von Accounts erlauben. Eine andere Möglichkeit wäre, die vom Superserver genutzten XMPP-Account wiederzuverwenden und lediglich einen anderen Ressourcennamen (siehe Abschnitt 2.1.1) zu nutzen. Dies würde allerdings auch Anpassungen an einigen Stellen des Saros-Kerns erfordern, die annehmen, dass bei jedem XMPP-Account maximal eine Saros-Instanz auf einmal angemeldet ist, beispielsweise bei `OutgoingSessionNegotiation`[32].

- **Wie werden laufende Serversitzungen identifiziert?** Möchte ein Nutzer einer laufenden Serversitzung beitreten, reicht es in einem Mehr-Sitzungsmodell nicht mehr, nur die Addresse des (Super-)Servers zu kennen. Stattdessen muss auch die konkrete Sitzung benannt werden. Beispielsweise könnte jeder Serversitzung ein Name zugeordnet werden, der entweder vom Superserver generiert oder vom Ersteller der Sitzung gewählt wird. Erhielte der Superserver dann eine Anfra-

[32] https://github.com/saros-project/saros/blob/00eb713f490ec3014fe0046112f13640424de4c3/de.fu_berlin.inf.dpp.core/src/de/fu_berlin/inf/dpp/negotiation/OutgoingSessionNegotiation.java#L220

ge zum Beitritt einer Sitzung, könnte er anhand des übergebenen Namens die entsprechenden Sitzungsserver-Instanz ermitteln und benachrichtigen, damit diese den anfragenden Nutzer einlädt.

Die Beantwortung dieser und anderer Fragen sowie die Implementation dieser Idee wäre ein lohnendes Thema für eine zukünftige Abschlussarbeit.

7.4 Zusammenfassung und Danksagungen

Mit dieser Arbeit konnte das Saros-Projekt dem Ziel eines produktiv einsetzbaren Servers für Saros-Sitzungen deutlich näher gebracht werden. Es ist zu hoffen, dass diese Bemühungen in Zukunft fortgesetzt werden und in einem echten Mehrwert für die Nutzerbasis von Saros resultieren.

An dieser Stelle möchte ich Franz Zieris danken, der sich im Laufe dieser Arbeit konsequent als hervorragender Betreuer und SarosProjektkoordinator erwiesen hat. Außerdem danke ich Prof. Lutz Prechelt und Prof. Adrian Paschke für die Begutachtung dieser Arbeit. Zuguterletzt gilt mein Dank allen Studenten und und anderen Entwicklern, mit denen ich im Saros-Projekt zusammenarbeiten durfte, insbesondere Ute Neise für die Kooperation bei der Entwicklung des Non-Host Project Sharing, sowie Stefan Rossbach, dessen Code-Review-Aktivität und Auskünfte immer wieder wertvoll war.

Literatur

[Bus14] Nils Bussas. Entwicklung eines Server-Prototypen für Saros. Bachelorarbeit, 2014.

[Dam15] David Damm. Schnellerer Sitzungsstart in Saros. Bachelorarbeit, 2015.

[Dje06] Riad Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung. Diplomarbeit, 2006.

[Fow99] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison Wesley, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[JMC] Debug a Memory Leak Using Java Flight Recorder. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks001.html. Oracle Technology Network (Stand 03.01.2016).

[Las15] Arndt Lasarzik. Refaktorisierung des Eclipse-Plugins Saros für die Portierung auf andere IDEs. Bachelorarbeit, 2015.

[NCDL95] David A. Nichols, Pavel Curtis, Micheal Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, pages 111–120, 1995.

[Nei] Ute Neise. Weiterentwicklung des Saros-Servers. Laufende Studienarbeit.

[SBS10] Stephan Salinger, 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*, pages 48–55, 2010.

[Sch13] Patrick Schlott. Analyse und Verbesserung der Architektur eines nebenläufigen und verteilten Softwaresystems. Masterarbeit, 2013.

[Sta13] Sebastian Starroske. Improving the Reliability of Saros Using Root Cause Analysis. Masterarbeit, 2013.

[Sun] David Sungalia. Refaktorisierung und Zusammenführung der Saros-Plugins in das plattformunabhängige Core-Package. https://www.mi.fu-berlin.de/w/SE/ThesisSarosPluginRefactoring. Laufende Bachelorarbeit (Stand 03.01.2016).

[The] Daniel Theus. Entwicklung einer Infrastruktur für wechselbare Projektübertragungsformen und Weiterentwicklung Bestehender in Saros. https://www.mi.fu-berlin.de/w/SE/ThesisOptimierungSitzungsstart. Laufende Bachelorarbeit (Stand 03.01.2016).

[TvS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[WR94] Charles Wohlin and Per Runeson. Certification of software components. *IEEE Transactions on Software Engineering*, 20(6):111–120, June 1994.

8 Liste der eingereichten Codeänderungen

Diese Änderungen sind in der Gerrit"=Instanz des Saros"=Projekts unter http://saros-build.imp.fu-berlin.de/gerrit/ einsehbar.

Integrierte Änderungen

- **2209:** *[INTERNAL] Remove obsolete "project ownership" notion*
- **2235:** *[REFACTOR] Rename CancelableProcess to Negotiation*
- **2249:** *[REFACTOR] Move common methods to Negotiation*
- **2785:** *[INTERNAL] Implement internalWorked() in CoreToEclipseMonitorAdapter*
- **2787:** *[FIX][E] Fix JIDCombo on Eclipse Mars*
- **2788:** *[REFACTOR] Move RemoteProgressManager to core*
- **2799:** *[REFACTOR] Move SarosProjectMapper to core (as SharedProjectMapper)*
- **2803:** *[FIX][E] Fix JIDCombo on Eclipse Mars*
- **2810:** *[REFACTOR][API] Expose getLocally/RemoteOpenEditors() in IEditorManager*
- **2811:** *[REFACTOR] Pull up add/removeSharedEditorListener() to IEditorManager*
- **2812:** *[REFACTOR] Refactor and document editor listener classes*
- **2813:** *[API] Introduce IEditorManager#getContent*
- **2814:** *[DOC] Improve IEditorManager docs*
- **2815:** *[REFACTOR][E] Free ConsistencyWatchdogServer from non-core dependencies*
- **2816:** *[INTERNAL][E] Clean up ConsistencyWatchdogServer*
- **2821:** *[API] Introduce ISarosSessionContextFactory*
- **2829:** *[INTERNAL] Move ActivityHandler to core*
- **2830:** *[INTERNAL] Move ActivityQueuer to core*
- **2831:** *[INTERNAL] Move ActivitySequencer to core*
- **2832:** *[REFACTOR] Move PermissionManager to core*
- **2833:** *[REFACTOR] Move UserInformationHandler to core*
- **2834:** *[INTERNAL] Create already-in-core session components in SarosCoreSessionContextFactory*
- **2835:** *[REFACTOR] Move Client/ServerSessionTimeoutHandler to core*
- **2836:** *[INTERNAL] Remove Eclipse IPreferenceStore dependency from ColorIDSetStorage*
- **2840:** *[REFACTOR][E] Free ConsistencyWatchdogServer from non-core dependencies*
- **2842:** *[REFACTOR] Rename ISarosSessionListener to ISessionLifecycleListener*
- **2843:** *[REFACTOR] Rename ISharedProjectListener to ISessionListener*
- **2844:** *[API] Introduce ISessionListener#userColorChanged*
- **2845:** *[INTERNAL] Remove EditorManager dependency from ChangeColorManager*
- **2846:** *[INTERNAL] Port user color change listeners to use ISessionListener*
- **2847:** *[JUNIT] Introduce MemoryPreferenceStore to core test utilities*
- **2848:** *[REFACTOR] Move ColorIDSetStorage to core*

- **2849:** *[INTERNAL] Move ChangeColorManager to core*
- **2850:** *[API] Add ISessionListener#projectAdded/Removed*
- **2851:** *[INTERNAL] Remove SharedResourcesManager dependency from SarosSession*
- **2854:** *[REFACTOR] Move SarosSession to the core*
- **2863:** *[REFACTOR] Rename ISessionLifecycleListener#projectAdded to #projectResources-Available*
- **2864:** *[REFACTOR] Move IncomingProjectNegotiation to the core*
- **2865:** *[REFACTOR] Move OutgoingProjectNegotiation to the core*
- **2866:** *[REFACTOR] Move INegotiationHandler to the core*
- **2876:** *[S] Create an Eclipse project for the Saros server*
- **2877:** *[INTERNAL][S] Implement IPath for the server*
- **2880:** *[BUILD][S] Add Ant build file for server*
- **2888:** *[API] Add #segment(int) to IPath*
- **2889:** *[API] Pull up #getLocation to IResource*
- **2890:** *[BUILD][S] Add SonarQube properties file for the server*
- **2899:** *[INTERNAL] Move SarosSessionManager to the core*
- **2900:** *[BUILD][I] Explicitly require JavaSE-1.6 execution environment*
- **2902:** *[API] Pass reason to ISessionLifecycleListener#sessionEnded*
- **2904:** *[INTERNAL][S] Implement IResource for the server*
- **2906:** *[JUNIT][S] Introduce FileSystemTestUtils*
- **2920:** *[BUILD][S] Use Ivy dependencies in the server Eclipse project*
- **2932:** *[NOP] Fix Eclipse Ëmpty block"warnings*
- **2933:** *[NOP] Fix all Eclipse Javadoc warnings (except one)*
- **2934:** *[NOP] Add missing @Override annotations*
- **2951:** *[INTERNAL] Remove SarosView dependency from LeaveAndKickHandler*
- **2952:** *[INTERNAL] Move LeaveAndKickHandler to the core*
- **2953:** *[INTERNAL] Turn LeaveAndKickHandler into a session component*
- **2962:** *[BUILD] Add server to the sonarReview config*

Änderungen in Begutachtung

- **2264:** *[FEATURE] Make it possible for non-hosts to add projects*
- **2781:** *Remove "newSession"flag from JoinSessionRequestExtension*
- **2802:** *[FIX][E] Fix JIDCombo on Eclipse Mars*
- **2898:** *Add Vagrant configuration*
- **2907:** *[INTERNAL][S] Implement IFile for the server*
- **2929:** *[S] The huge server patch*
- **2935:** *[NOP] Remove deprecation of User#getJID*
- **2936:** *[API] Remove deprecated IConnectionManager methods*

Änderungen im Entwurfsstadium

- **2912:** *[INTERNAL][S] Implement IContainer and IFolder for the server*
- **2914:** *[INTERNAL][S] Implement IProject for the server*
- **2921:** *[INTERNAL][S] Implement IWorkspace for the server*
- **2940:** *[INTERNAL][S] Implement UISynchronizer for the server*