Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Verbesserung des Session Start im Saros-Projekt

Abschlussarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang Angewandte Informatik

1. Prüfer: Prof. Dr. Stephan Salinger
2. Prüfer: M.Sc. Franz Zieris

Eingereicht von Stefan Moll
13. August 2018

Gewidmet dem Sommer 2018
– 22 Uhr am 8. August bei 29,4 °C (wolkig) –

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Saros [1] ist ein offenes Projekt, das EntwicklerInnen die Möglichkeit bietet, räumlich verteilt und in Echtzeit, kollaborativ an Softwareprojekten zu arbeiten. Dabei liegt ein Fokus auf der aus der agilen Softwareentwicklung stammenden Praktik *Paarprogrammierung*, in der zwei EntwicklerInnen gemeinsam an einem Bildschirm den gleichen Programmcode bearbeiten.

Dafür erweitert Saros die bestehende Entwicklungsumgebung[1] um Funktionen, wie z. B. andere Personen zum gemeinsamen Arbeiten einzuladen, deren Eingaben und Markierungen direkt anzuzeigen sowie dem aktuellen Sichtbereich des Partners auf Dateien zu folgen. Durch den Einsatz von offenen Standards und Software, die es ermöglichen benötigte Infrastruktur selbst zu betreiben bzw. sich einen eigenen Anbieter auszusuchen, können NutzerInnen bzw. Unternehmen frei darüber entscheiden, was mit ihren Daten passiert. Die Unabhängigkeit von Plattformen großer Anbieter sowie die Möglichkeit diese Arbeitsform verteilt auszuführen erfüllt damit eine wichtige Nische. Aus meiner Zeit als angestellter Softwareentwickler ist mir die Situation, in der Probleme am gleichen Bildschirm analysiert wurden, noch gut bekannt. Besonders dann, wenn KollegInnen (z. B. wie damals in Indien) weit entfernt arbeiten, ist Saros eine echte Alternative zum konventionellen Bildschirmteilen.

Bekannt wurde mir *Saros* im Rahmen der Veranstaltung *Komponentenbasierte Entwicklung* von Professor Salinger. Aufgrund meines Interesses an dem Projekt und an Netzwerkthemen begann ich eine eigene Idee für einen serverunabhängigen Einladungsprozess zu überlegen. Im Gespräch mit Herrn Zieris stellte sich jedoch heraus, dass die vorhergehenden Arbeiten am *Instant-Session-Start* noch nicht abgeschlossen wurden, eine Fertigstellung für das Projekt allerdings wünschenswert wäre.

[1] aktuell Primär *Eclipse*, *IntelliJ* in Entwicklung

1.2 Aufgabenstellung

Saros ermöglicht EntwicklerInnen ihren lokalen Programmcode gemeinsam mit anderen zu bearbeiten. Dafür wird zunächst eine Sitzung gestartet und die lokalen Dateien an die eingeladene Person übertragen. Auf diesem Stand werden daraufhin alle Aktivitäten, wie Textänderungen oder Markierungen, fortlaufend synchronisiert.

Dieser Sitzungsstart (*Session-Start*) soll schrittweise verbessert werden.

Dafür ist erwünscht zwei bestehende Einschränkungen teilweise oder, wenn möglich, ganz zu beheben. Zum einen kann die eingeladene Person Dateien erst öffnen und bearbeiten wenn der Sitzungsstart mit der Übertragung des gesamten Projekts abgeschlossen ist. Zum anderen gibt es dabei ein Zeitfenster in dem auch bereits teilnehmende Personen keine Dateiänderungen vornehmen können. Das Ziel besteht daher, dass Dateien möglichst früh bereitstehen, ohne dabei die Arbeit der anderen Personen einzuschränken.

1.3 Vorgehensmodell und Aufbau der Arbeit

Diese Arbeit ist thematisch wie folgt gegliedert. Um die verwendeten Techniken und Systeme zu erklären, wird im *Kapitel 2. Grundlagen* zunächst auf selbige eingegangen, dabei bietet das Glossar kurze Erklärungen wiederkehrender Begriffe. Im Anschluss wird eine Analyse des Projekts, wichtigen Abläufen und bisherigen Abschlussarbeiten auf dem Gebiet des *Instant-Session-Starts* im *Kapitel 3. Analyse* vorgenommen. Dabei werden bisherige Fortschritte sowie Probleme benannt. Anhand dieser Erkenntnisse werden im *Kapitel 4. Definition* die Anforderungen an eine neue Lösung aufgestellt.

Für den Verlauf der Arbeit wurden folgende Meilensteine definiert:

1. Testumgebung und Testcases ausgearbeitet
2. schnelles lesendes Teilnehmen möglich
3. schreibendes Teilnehmen nach Start

Dabei sind Punkt zwei und drei Teil des *Kapitels 5. Entwurf und Implementierung*. Dieses behandelt das iterative Vorgehen und beschreibt den Ablauf der mehreren Implementierungsphasen. Um die Umsetzung frühzeitig zu prüfen und die entstehende Lösung vergleichen zu können wurde der Meilenstein eins definiert. Für diesen wurde bereits frühzeitig eine Testumgebung geschaffen, sowie mehrere Testszenarien erstellt. Auf diese Szenarien wird neben den durchgeführten Tests im *Kapitel 6. Test* eingegangen. Im *Kapitel 7. Ergebnis* wird abschließend diese Arbeit zusammengefasst und ein Ausblick auf weitere Aufgaben gegeben.

2 Grundlagen

In diesem Kapitel werden Begriffe definiert sowie verwendete Systeme und Konzepte beschrieben.

2.1 Glossar

Client	die Saros-Instanz der eingeladenen Person
Consistency Watchdog	ein Saros-Dienst, der fortlaufend Prüfsummen aller geteilten Dateien erstellt und bei Differenzen über einen Konflikt informiert; bietet die Möglichkeit diese aufzulösen indem die Datei des Host neu bezogen wird
Gerrit	die vor GitHub genutzte Review-Plattform des Saros-Projekt
git	das verwendete Versionskontrollsystem im Saros-Projekts
GitHub	Plattform zum Austausch von git-Projekten, seit Juli 2018 die zentrale Plattform für Quellcode, Reviews und Tickets im Saros-Projekt
Hauptentwicklungszweig	der *master*-Entwicklungszweig des Saros-*git*-Projekts
Host	die Saros-Instanz der einladenden Person
Instantübertragung	die neu entstandene Übertragungsart
Pico Container	ein Java-*Framework* um *Dependecy-Injection* umzusetzen; ein Entwurfsmuster um Abhängigkeiten zu Objekten zentral zu verwalten und den Zugriff sowie Lebenslauf der verwalteten Objekte zu steuern
Remote Method Invocation	(RMI) ermöglicht entfernte Methoden-Aufrufe anderer Java-Instanzen
Saros Test Framework	(STF) wird eingesetzt um Saros-Instanzen über RMI zu steuern und automatisierte Oberflächentests auszuführen

Streams	beziehen sich in der gesamten Arbeit auf *Input-/Outputstreams*, nicht zu Verwechseln mit der *Stream-API* seit Java 1.8
Smack	ist die eingesetzte Java-Bibliothek zur Kommunikation mit einem *XMPP*-Server sowie, sofern möglich, direkt zwischen Host und Client
Übertragungsart	die in der Projektaushandlung genutzte Variante zum Austausch von Dateien
Wireshark	Programm zur Aufzeichnung und Analyse von Netzwerkverkehr
XMPP	Abk. für Extensible Messaging and Presence Protocol, ein erweiterbarer offener Standard zur Kommunikation über XML, ermöglicht Ein- sowie Mehrpersonen-Chats, Kontaktlisten, Datenübertragung, Präsenz und weitere Dienste, Kommunikation erfolgt Client-zu-Server-basiert, wobei der Austausch, falls erlaubt, auch zu Konten auf anderen Servern möglich ist
Zstandard	Bibliothek zum verlustfreien Komprimieren

2.2 Saros-Aktivitäten

Aktivitäten sind Nachrichten die in Saros zwischen den Teilnehmern ausgetauscht werden um den Zustand einer Sitzung fortlaufend zu synchronisieren. Dafür existieren unterschiedliche Aktivitätsarten wie die *Text-Selection-Activity* zum Markieren von Text oder *File-Activity* zum Versand von Dateien. Eine Aktivität bildet immer eine einzelne Änderung ab und wird zentral vom Host verwaltet. Dieser empfängt alle Aktivitäten der Clients und verteilt diese falls notwendig weiter.

Eine spezielle Form dieser Aktivitäten ist die *Jupiteraktivität*. Diese wird genutzt um speziell alle Änderungen an einer Datei (als Dokument bezeichnet) zu synchronisieren und wird getrennt in der *Jupiter-Architektur* verarbeitet. Da verschiedene Personen gleichzeitig eine gleiche Datei bearbeiten können, müssen die dabei entstehenden Konflikte aufgelöst werden. Wenn z. B. Person A an Textposition zehn und Person B an Position 100 gleichzeitig jeweils ein Zeichen einfügen, dann muss die Eingabe von B um ein Zeichen verschoben angewendet werden da die Eingabe von A alles Nachfolgende verschiebt. Daher ist es Aufgabe der *Jupiter-Architektur* die Synchronisation aller Änderungen zu übernehmen und Konflikte aufzulösen.

2.3 Docker

Docker ist eine Softwarelösung die das Erstellen, Verbreiten und Starten von Applikationen in eigenen Virtuellen-Maschinen erlaubt [2]. Dabei wird der Ansatz verfolgt Applikation in einem transportablem Format bereitzustellen das alle zur Laufzeit benötigten Abhängigkeiten wie das Betriebssystem, Softwarepakete / Bibliotheken, notwendige Konfigurationen sowie die Applikationen selbst bündelt. Diese werden *Images* genannt und es existieren zentrale Verzeichnisse aus denen bereits existierende Varianten bezogen werden können. So gibt es unter anderem *Images* die einen *XMPP-Server* bereitstellen und damit den sonst hohen Konfigurationsaufwand bei regulärem Installationsweg ersparen. Instanzen dieser Images sind Virtuelle-Maschinen die in *Docker* als *Container* bezeichnet werden. Sie gehören zu den leichtgewichtigen Virtualisierungsvarianten, da sie Teile des ausführenden Betriebssystems nutzen. So startet ein *Container* zwar sein eigenes Betriebssystem, jedoch nur innerhalb eines eigenen Namensbereich im Betriebssystemkern des Wirtsystem. Dieser Namensbereich sorgt dafür das jeder *Container* isoliert von anderen Systemen ist und seinen eigenen Rechtebereich führt.

Docker-Container können zur Kommunikation virtuelle Netzwerkadapter des Wirtsystems erhalten, die wiederum in unterschiedlichen Konfigurationen zusammengefasst werden können. Da diese Netzwerkadapter wie normale Adapter des Wirts manipuliert werden können, kann man somit unterschiedliche Netzwerkszenarien simulieren.

2.4 Netzwerkmanipulation

Das Linux-Anwendungspaket *iproute2* stellt unter den meisten Linux Distributionen das Kommandozeilen Programm *tc*[1] bereit. Dieses ermöglicht es Manipulationen an einzelnen Netzwerkadaptern zu konfigurieren. So ist es mittels eines *Token-Bucket-Filter (tbf)* möglich die Bandbreie des Netzwerkverkehrs zu limitieren. Dafür wird in einem Zeitintervall eine bestimmte Anzahl an *Token* bereitgestellt und jedem passierenden Netzwerkpaket eine größenabhängige Anzahl zugeteilt. Sind keine Token mehr vorhanden müssen diese Pakete warten oder werden verworfen [3]. Mittels der *NetEM*-Option aus *tc* können Netzwerkpakete verzögert, verworfen, beschädigt, dupliziert oder in der Reihenfolge verändert werden [4].

Damit ermöglicht dieses Anwendungspaket unterschiedliche Netzwerksituationen je Adapter zu simulieren, wie sie auch bei der Kommunikation über das Internet oder allgemein fehleranfälliger Netze entstehen können.

[1] Abkürzung für *Traffic Control*

3 Analyse

Die Entwicklung von Saros wurde bereits 2006 im Rahmen der Diplomarbeit von Riad Djemili begonnen und seitdem von über neunzig Personen fortgeführt [5]. Dabei entstand ein Großteil innerhalb von Abschlussarbeiten bzw. der Arbeitsgruppe Software Engineering an der Freien Universität Berlin. Durch den Einsatz von qualitätssichernden Maßnahmen, wie dem verpflichtenden Durchführen von Reviews vor der Integration von Quellcode in den Hauptentwicklungszweig sowie dem Einsatz von automatischen Systemtests die sogar Oberflächen mit einschließen, ist mit Saros ein stabiles System zum verteilten Arbeiten entstanden. Auch die mit anderen Projekten verglichen gute Dokumentation trägt dazu bei, in dem z. B. die Erweiterung *JTourBus*[1] einen Leitfaden durch wichtige Abläufe des Quellcode bietet. Da jedoch viele Abschlussarbeiten wie auch diese auf neue Funktionen ausgerichtet sind, gibt es auch einige Aufgaben, die offen geblieben oder mit der Zeit neu hinzugekommen sind.

So ist einer der auffallenden Punkte das Verwenden von alten Programmversionen wie *Eclipse* in Version 3.7.2 (von Februar 2012[2]) und Java in Version 6 (letztes frei zugängliches Java Development Kit 7 Update 2013[3]). Zudem ist die XMPP-Kommunikation mit aktuellen TLS-Versionen nicht kompatibel und das Aufschieben von *Code-Refactoring* erschwert das Integrieren neuer Funktionen. Dies zeigen zahlreiche alte Quellcodeanmerkungen, wie sie in den von mir bearbeiteten Programmteilen vorkamen, auf.

Um Saros auch auf andere Entwicklungsumgebungen wie aktuell *IntelliJ* zu portieren, wurde begonnen es in mehrere Teilprojekte zu zerlegen. Dies ist jedoch noch nicht abgeschlossen und so gibt es noch einige Probleme, wie zum Beispiel, dass einer Sitzung keine neuen Dateien hinzugefügt werden können. Daher ist sie für diese Arbeit noch nicht relevant.

Saros teilt sich zusammengefasst in folgende Teilprojekte auf:

Kern Dieser umfasst alle zentralen Komponenten zur Netzwerkkommunikation, *XMPP*-Konten, Aktivitätenverwaltung, Dateiprüfungen, Sitzung, Einladungsprozesse und Bibliotheken. Er Umfasst damit alles was nicht grafisch oder anderweitig an die Entwicklungsumgebungen gebunden ist.

Entwicklungsumgebungen Für Eclipse sowie IntelliJ in jeweils eigenen Projekten und bauen auf dem Kern auf. Dort werden alle Inhalte der grafischen Oberflächen

[1] https://www.saros-project.org/jtourbus
[2] https://www.eclipse.org/downloads/packages/eclipse-classic-372/indigosr2
[3] https://blogs.oracle.com/java/java-se-7-update-21-release-and-more

(GUIs) verwaltet.

HTML-GUI Um gleiche Oberflächen für beide Entwicklungsumgebungen einzusetzen, wird eine HTML-Oberfläche entwickelt.

Server Da eine Sitzung momentan immer vom Einladenden verwaltet und bei seinem Verlassen für alle Teilnehmenden beendet wird, entsteht ein Server der permanent laufende Sitzungen ermöglicht.

Whiteboard Eine grafische Oberfläche auf der Personen wie auf einem klassischen Whiteboard zeichnen können. Kommt ausschließlich in der Eclipse-Umgebung zum Einsatz.

Da in dieser Arbeit ein Teil der Einladung sowie Aktivitäten geändert wird, ist der Hauptanteil der Arbeiten am Kern durchzuführen. Einzig für ein neues Feld im grafischen Einstellungsmenü wird eine Anpassung im Eclipse-Teil notwendig.

Des Weiteren ist das Saros-Projekt zum Zeitpunkt dieser Arbeit in einer organisatorischen Umbruchphase, in der ein System-, Prozess- und Personalwechsel stattfindet.

3.1 Einladungsprozess

Mit Saros können NutzerInnen lokale Dateien mit Personen Ihrer *XMPP*-Kontaktliste teilen und in Echtzeit bearbeiten. Dafür wird in einem zweiteiligen Prozess zunächst zu einer Sitzung eingeladen. Dabei wird der Einladende automatisch zum Server, im Folgenden Host genannt und ist die zentrale Instanz zur Steuerung aller Zustände der Sitzung. Zu einer Sitzung können mehrere Personen eingeladen werden, die über einen gemeinsamen Chat sowie über ein Whiteboard zum gemeinsamen Arbeiten verfügen.

Der zweite Teil ist die Freigabe eines oder mehrerer in der Entwicklungsumgebung vorhandenen Projekte innerhalb der Sitzung. Dafür wird eine Liste aller Projektdateien inklusive ihrer Prüfsummen erstellt und an die eingeladene Person, im Folgenden Client genannt, gesendet. Der Client hat daraufhin die Möglichkeit die eingehenden Projekte mit vorhandenen zu synchronisieren oder neue anzulegen. Wird die Synchronisierung mit einem bestehenden Projekt gewählt, werden die Prüfsummen verglichen und eine Liste der Unterschiede an den Host gesendet. Der Host übermittelt daraufhin alle notwendigen Projektdateien.

Archivübertragung

Zur Übermittlung der Projektdateien wird bisher eine ZIP-Archiv-basierte Lösung genutzt. Die Aufgabe dieser Archivdatei ist es den Zustand aller Dateien konsistent abzubilden, damit der Client die exakt gleichen Inhalte wie der Host erhält. Damit ist explizit

gemeint, dass alle Änderungen zu einem bestimmten Zeitpunkt enthalten sind und keine davor getätigten verloren oder danach entstandenen berücksichtigt werden.

Um dies zu gewährleisten, wird während des Erstellvorgangs allen Teilnehmern der Schreibzugriff entzogen und alle Dateien gespeichert. Dadurch wird die Sitzung pausiert und der Zustand nicht mehr verändert sowie alle bisher ungespeicherten Änderungen übernommen. Dies ist wichtig, da die Dateien zur Archiverstellung vom Dateisystem geladen werden.

Die Übertragung eines konsistenten Zustands ist die Basis auf der der Client nun fortlaufend Aktivitäten zur weiteren Synchronisierung erhält. Der Start der Aktivitätenzustellung beginnt während die Sitzung pausiert wurde, damit nachfolgende Zustandsänderungen nicht verloren gehen. Da der Client bereits Aktivitäten erhalten kann, bevor die Archivdatei zugestellt und entpackt wurde, wird eine Warteschlange eingerichtet, die projektbezogen alle Aktivitäten bis zum Erhalt zwischenspeichert und dann ausführt. Somit wird gewährleistet, dass ab dem Pausieren der Sitzung keine Änderungen verloren gehen und der neue Client Teil der Sitzung wird.

Nutzerinteraktion

Wie beschrieben ist der Einladungsprozess zweigeteilt und die teilnehmende Person wird im ersten Schritt zunächst gefragt, ob sie an der Sitzung teilnehmen möchte. Dafür erhält sie einen Dialog, der kurz über den Namen und Größe der Projekte informiert. Erst nach Annahme der Einladung beginnt der Host mit der Erstellung der Dateiliste und Prüfsummenberechnung und übermittelt diese an den Client. Abhängig von der Projektgröße und Übertragungsgeschwindigkeit dauert dieser Vorgang einige Sekunden, bis die eingeladene Person zur Auswahl des Zielprojektes aufgefordert wird. Mit Bestätigung dieses Dialogs beginnt der Host die Übermittlung der Projektdateien, worüber mit Fortschrittsanzeigen informiert wird.

3.2 Arbeiten im Teilgebiet

Die ersten beiden Arbeiten, die sich mit der Implementierung des *Instant-Session-Start* befassten, sind bereits 2015 von David Damm [6] sowie Daniel Theus [7] entstanden. Dieser folgte 2016 die Arbeit von Patrick Fehling [8].

In diesen Arbeiten wurde begonnen eine Möglichkeit zu schaffen neben der Archivübertragung noch weitere Übertragungsarten zu unterstützen und diese abhängig von lokalen Einstellungen auszuwählen bzw. im späteren Versuch während der Projekteinladung auszuhandeln. Die neue Übertragungsvariante sollte den Versand mittels Dateiaktivitäten realisieren, wofür auch Änderungen an der Aktivitätenverarbeitung notwendig wurden. Diese Arbeiten wurden jedoch nicht abgeschlossen und es blieben einige Probleme ungelöst.

Victor Brekenfeld begann zwei Monate vor dem Beginn meines Bearbeitungszeitraums (04.06. – 13.08.2018) damit die Arbeiten wieder aufzugreifen, da er sie im Rahmen seiner Masterarbeit am *Saros-Server* nutzen wollte. Erste Patches entstanden dabei bereits recht früh und wurden zum Review gestellt[4]. Dabei sollte zunächst die Arbeit von Fehling [8] soweit aufgearbeitet werden, dass sie in den aktuellen Hauptentwicklungszweig integriert werden kann. Zum Bearbeitungsbeginn dieser Arbeit war dies jedoch noch nicht abgeschlossen und so wurden im Verlauf Teile dieser Aufgaben von mir übernommen. Dadurch entstanden Verschiebungen, die im *Kapitel 5. Entwurf und Implementierung* beschrieben werden.

Projektübertragung mittels Aktivitäten

Der in den vorhergehenden Arbeiten zum *Instant-Session-Start* verfolgte Ansatz war es die bestehenden Dateiaktivitäten auch für die Projektübertragung zu nutzen. Zwar wird die bisherige Aktivitätenbehandlung bereits lange erfolgreich dazu eingesetzt nach dem Projektaustausch alle Nutzeraktionen zu synchronisieren, jedoch zeigte Patrick Fehling folgende Probleme auf:

1. Empfangene Dateien können während der Projektübertragung nur lesend geöffnet werden.
2. Verwendete Dateiaktivitäten werden auch an alle anderen Clienten versendet. Dies erzeugt zusätzliche vermeidbare Netzwerkauslastung und Verzögerungen und zudem entstehen durch die zusätzlichen Aktivitäten unerwartete Zustände.
3. Aktivitäten, die nach Beginn der Aushandlung entstehen, empfängt der eingeladene erst nach Ende der Projektübertragung.
4. Dateiaktivitäten laden Inhalte immer vollständig in den Arbeitsspeicher. Dies kann je nach Systemkonfiguration zu *Heapspace*-Fehlern führen, wodurch die gesamte Sitzung beendet wird.
5. Aktivitäten werden nicht parallel an TeilnehmerInnen versendet. Dies verlangsamt die Verarbeitung, da auf langsamere Übertragungen gewartet wird.
6. Der Projektaustausch ist nur durch Sitzungsende unterbrechbar.
7. Dateien werden nicht priorisiert. Dadurch werden für das Arbeiten wichtige Dateien unter Umständen zuletzt übertragen.
8. NutzerInnen erhalten nur wenige Informationen zum Fortschritt. [8, S. 12]

Zum Zeitpunkt meiner Analyse bestehen die Probleme eins und drei unverändert. Punkt zwei wurde mit gezielt an Clienten versendbare Dateiaktivitäten gelöst[5]. Der *Heapspace*-

[4] `https://saros-build.imp.fu-berlin.de/gerrit/3498`
[5] `https://saros-build.imp.fu-berlin.de/gerrit/3519`

Fehler aus Punkt vier wurde teilweise versucht zu lösen[6], jedoch nicht erfolgreich. Paralleler Aktivitätenversand aus Punkt fünf ist zwar vor vielen Jahren Implementiert worden, jedoch seit 2013 wegen Fehlern[7] nur experimentell nutzbar und auch die Beseitigung dieser Fehler war nicht erfolgreich[8]. Zu den Punkten sechs bis acht existiert eine nicht Fertiggestellte Implementierung[9], die mittlerweile jedoch teilweise veraltet ist. Die allgemeinen Einschränkungen dieser Übertragungsart werden im folgenden erläutert.

Dateiaktivitäten

Daniel Theus zeigte, dass ein *Eclipse*-Projekt mit vielen kleinen Dateien beim Versand mittels Dateiaktivitäten besonders lange benötigt [7, S. 29f]. In seinen Tests war die von ihm damit implementierte Übertragungsart bei einem gleich großen Projekt, mit 128 kB großen Dateien, ca. 8 mal langsamer, verglichen mit 4 MB Dateien. Dazu war die ZIP-Datei-Variante nur ca. 50 % langsamer.

Um die Gründe zu ermitteln, analysierte ich die Logausgaben zur Übertragung einer ca. 3 kB großen Datei. Dabei zeigte sich, dass Saros zwei Nachrichten versendet: Zum einen ein *TFD*-Paket, welches über das nachfolgende Paket informiert und zum anderen ein *DATA*-Paket, mit dem eigentlichen Dateiinhalt und Beschreibungen. Dem Log kann man entnehmen, dass dieses *DATA*-Paket komprimiert wurde und der Inhalt nur noch halb so groß im Vergleich zur ursprünglichen Datei ist. Die Analyse mit *Wireshark* zeigte, dass für die Übertragung inklusive dem *TFD*-Paket drei Netzwerkpakete entstehen und tatsächlich 1,8 kB versendet werden. Dies setzt sich daraus zusammen, dass das *Data*-Paket zu groß für ein Netzwerkpaket ist und daher geteilt wird. Dadurch entsteht ein Netzwerk-*Overhead* von insgesamt 13 %. Da die lange Übertragungszeit damit noch nicht alleine erklärbar ist, komprimierte ich die betreffende Datei in einem ZIP-Archiv, welches hingegen nur noch 0,8 kB groß und damit halb so groß wie das *DATA*-Paket war.

```
(BinaryChannelConnection.java:315) processing opcode 0xFA [TFD]: id=0, chunks
    =1
(BinaryChannelConnection.java:353) processing opcode 0xFB [DATA]: id=0, DATA
     len=1585 bytes
(NetworkManipulatorImpl.java:99) intercepting incoming packet from:
    alice@example.com/Saros
(DataTransferManager.java:99) received binary XMPP extension:
    TransferDescription [elementName=ados, namespace=de.fu_berlin.inf.dpp,
    recipient=bob@example.com/Saros, sender=alice@example.com/Saros,
    compress=true], size: 1585, RX time: 0 ms [SOCKS5 (D)]
(ActivitySequencer.java:598) rcvd (001) alice@example.com/Saros -> [
    FileActivity [dst:path=SPath [project=P/de.fu_berlin.inf.dpp.core (
    EclipseProjectImpl), projectRelativePath=version.comp], src:path=N/A,
    type=CREATED, encoding=N/A, content=3117 byte(s)]]
```

[6] https://saros-build.imp.fu-berlin.de/gerrit/3085
[7] https://saros-build.imp.fu-berlin.de/gerrit/598
[8] https://saros-build.imp.fu-berlin.de/gerrit/3077
[9] https://saros-build.imp.fu-berlin.de/gerrit/3075

Damit zeigte sich das Dateiaktivitäten ineffizient für den Versand sind und in diesem *Use-Case* mindestens die doppelte Übertragungdauer benötigen. So ist überraschend, dass selbst bei einer 2 Byte großen Datei versucht wird diese zu komprimieren und ein 213 Byte großes *Data*-Paket entsteht.

```
(DataTransferManager.java:99) received binary XMPP extension:
    TransferDescription [elementName=ados, namespace=de.fu_berlin.inf.dpp,
    recipient=bob@example.com/Saros, sender=alice@example.com/Saros,
    compress=true], size: 213, RX time: 0 ms [SOCKS5 (D)]
(ActivitySequencer.java:598) rcvd (001) alice@example.com/Saros -> [
    FileActivity [dst:path=SPath [project=P/de.fu_berlin.inf.dpp.core (
    EclipseProjectImpl), projectRelativePath=a.txt], src:path=N/A, type=
    CREATED, encoding=N/A, content=2 byte(s)]]
```

Ein anderes Problem der Dateiaktivitäten ist das Verwenden von *Byte-Arrays*, was zu den festgestellten *Heapspace*-Fehlern und damit zum Programmabsturz führt. Jede Datei wird bei Verarbeitung als Ganzes in den Arbeitsspeicher geladen und im Netzwerkteil mehrmals kopiert sowie unter Umständen aufgeteilt. Damit wird mindestens doppelt so viel *Heapspace* pro Dateigröße benötigt und das Problem verstärkt wenn viele Dateien gleichzeitig geladen werden. Daher wird eine Logik benötigt, die dafür Sorge trägt, dass nie zu viele Dateien gleichzeitig geladen werden und die mit Dateien die zu groß für den verbleibenden *Heap* sind umgehen kann. Auch das starke Beanspruchen der *Garbage-Collection*, die oft Speicherbereich wieder freigeben muss, belastet das System zusätzlich. Zwar ist das komplette Laden in den Hauptspeicher leichter um Synchronisierungsfehler zu umgehen, da somit ein unveränderlicher Dateiinhalt gespeichert wird, jedoch bleibt es für große Dateien eine unpraktikable Lösung.

Solange man also keine große Umstrukturierung der Aktivitätenverarbeitung vornimmt, ist es nicht möglich eine Lösung zu erstellen, die die Geschwindigkeit der ZIP-Archiv Variante erreichen kann.

3.3 Versionskontrolle und Reviewprozess

Im Projekt wird das Versionskontrollsystem *git* eingesetzt und wie bereits erwähnt werden Änderungen nicht direkt in den Hauptentwicklungszweig übernommen. Dafür ist es notwendig, vorher mindestens ein positives Review für den vorgeschlagenen Patch von anderen Projektpersonen zu erhalten. Dieser Prozess ist ursprünglich über die Plattform *Gerrit* organisiert worden und wurde mit der Umstellung auf *GitHub* auch auf den *GitHub*-spezifischen Review-Prozess umgestellt. Die Umstellung zu *GitHub* erfolgte zu Beginn der siebten Bearbeitungswoche und brachte einige Veränderungen im Prozessablauf. So musste zum einen abgewogen werden, ab wann Änderungen bereits im neuen System gestellt werden sollten und mit dem erschwerten Umgang, bei aufeinander folgenden Patches, umgegangen werden.

4 Definition

In diesem Kapitel werden die Erkenntnisse aus dem vorhergehenden Analyse-Kapitel genutzt, um Anforderungen an eine neue Lösung zu beschreiben. Diese bilden die Grundlage auf der im nächsten Kapitel ein Entwurf erstellt und Fortschritte getestet sowie bewertet werden können.

4.1 Funktionale Anforderungen

FA1 – Frühzeitiges Teilnehmen

Beschreibung	Eingeladene Personen sollen im Rahmen einer Projektaushandlung bereits vor Ende der Gesamtübertragung an einzelnen Dateien arbeiten können.
Nutzen	Die Zeit bis die eingeladene Person arbeiten kann wird verkürzt.
Akzeptanzkriterium	Einzelne Projektdateien sollen, schneller als mit der aktuellen Archivübertragung, von der eingeladenen Person (1) geöffnet und (2) bearbeitet werden.

FA2 – Bestehende Funktionalität erhalten

Beschreibung	Die Verwendung der neuen Übertragungsart muss optional sein.
Nutzen	(1) Die bisherige Archivübertragung ist erprobt und soll, mindestens solange eine neue noch nicht ausgereift ist, bei Fehlern oder wenn sie Nachteile beinhaltet weiterhin bereitstehen. (2) Die *IntelliJ*-Implementierung ist noch in der Entwicklung und unterstützt aktuell kein Hinzufügen neuer Dateien während einer Sitzung. Sie ist somit vorerst auf die Archivübertragung angewiesen.
Akzeptanzkriterium	NutzerInnen können über die Oberfläche bzw. über die allgemeine Konfiguration, die gewünschte Übertragungsart auswählen, der Standard bleibt jedoch die Archivübertragung.
Kommentar	Diese Funktion wurde bereits in der Arbeit von Patrick Fehling vorbereitet.

FA3 – Schreibblockierung aufheben

Beschreibung	Während der ZIP-Datei-Erstellung wird, um Inkonsistenzen zu vermeiden, der Schreibzugriff für alle Personen blockiert.
Nutzen	Das Vermeiden einer Arbeitsunterbrechung erhöht die NutzerInnen-Akzeptanz und wird im *Saros-Server* gewünscht.
Akzeptanzkriterium	Eine Bearbeitungseinschränkung, für (1) bereits teilnehmende oder (2) eingeladene NutzerInnen, findet während des Projektaustauschs nicht statt.

FA 4 – Übertragungsabbruch

Beschreibung	Der Übertragungsvorgang soll vorzeitig beendet werden können.
Nutzen	NutzerInnen sollen Vorgänge die ihnen zu viel Zeit in Anspruch nehmen schnell beenden können.
Akzeptanzkriterium	Für den Fall, dass Saros nicht durch externe Fehler unterbrochen wird, soll der Abbruch durchführbar sein, ohne die Entwicklungsumgebung dafür beenden zu müssen.

4.2 Nicht-Funktionale Anforderungen

Zur Steigerung der NutzerInnen-Akzeptanz werden folgende nicht-funktionale Anforderungen definiert:

NFA1 – Übertragungsdauer

Beschreibung	Die Übertragungsdauer der neuen Lösung soll vergleichbar mit der der Archivübertragung sein, damit sich die Gesamtwartezeit nicht unverhältnismäßig verlängert.
Akzeptanzkriterium	Der gesamte Übertragungszeitraum soll nicht länger als die doppelte Zeit der bisherigen Archivübertragung beanspruchen.

NFA2 – Oberflächenreaktionszeit

Beschreibung	Während der Projektübertragung soll die Entwicklungsumgebung bedienbar bleiben.
Akzeptanzkriterium	Im Zeitraum der Projektübertragung reagieren die grafischen Oberflächen mit der üblichen Verzögerung.

NFA3 – Ressourcenverbrauch

Beschreibung	Um die Lösung auch auf speicherschwachen Systemen nutzen zu können, soll der Speicherverbrauch angemessen sein und die Standardspeicherkonfiguration von Eclipse 3.7.2 nicht übersteigen.
Akzeptanz-kriterium	Während des Projektaustausches soll der benötigte Heap-Speicher das eineinhalbfache der bisherigen Übertragungsweise und die 384 MB der Eclipse-Standardkonfiguration nicht übersteigen.

4.3 Qualitätsanforderungen

Folgende nicht-funktionale Anforderungen beschreiben Ansprüche an die Qualität der Umsetzung:

QA1 – Zuverlässigkeit

Beschreibung	(1) Es soll verhindert werden, dass Dateien des Einladenden einen inkonsistenten Zustand annehmen oder Änderungen verloren gehen. (2) Bei Teilnehmern auftretende Inkonsistenzen müssen erkannt werden.

QA2 – Stabilität

Beschreibung	Das System soll auch bei vielen oder großen Dateien in einem stabilen Zustand bleiben.

QA3 – Reviewprozess

Beschreibung	Änderungen müssen den Reviewprozess des Projektes durchlaufen und sollten daher möglichst (1) leicht verständlich sein, (2) Entwicklungskonventionen einhalten und (3) wenige Änderungen innerhalb eines Patches durchführen.

4.4 Abgrenzungskriterien

Da sich die Umsetzung der *IntelliJ-IDEA*-Variante noch in einem frühen Entwicklungsstadium befindet und der *Saros-Server* innerhalb einer Masterarbeit bearbeitet wird, werden diese Programmteile im Rahmen dieser Arbeit nicht detailliert betrachtet. Die entstehende Lösung verfolgt den grundlegenden Ansatz nicht mit Alt-Funktionalität zu brechen und sich auf den Kern bzw. Eclipse-Teil zu fokussieren.

5 Entwurf und Implementierung

Dieses Kapitel beschreibt den Weg zur Umsetzung der Meilensteine *2. schnelles lesendes Teilnehmen möglich* und *3. schreibendes Teilnehmen nach Start*. Basierend auf den Erkenntnissen der Analyse und den Zielen der Definition, werden die aufeinander aufbauenden Entwurfs- und Implementierungsphasen des gewählten iterativen Vorgehens beschrieben. Dabei wird zunächst die Vorplanung und die Erstellung eines Prototypen vorgestellt und aus diesem dann schrittweise die konkrete Umsetzung entwickelt. Die Gründe für dieses Vorgehen beruhen auf der in *QA3 – Reviewprozess* beschriebenen Anforderung, die kurze Entwicklungszyklen begünstigt und darin, dass Erkenntnisse aus dem Review bereits in die nächste Iteration einfließen können.

5.1 Vorplanung

Die Analyse der vorhergehenden Arbeiten [6–8] zeigt, dass es grundlegende Probleme bei der Projektübertragung mittels Dateiaktivitäten gibt (Abschnitt 3.2. Arbeiten im Teilgebiet). Das Erfüllen der funktionalen Anforderungen mittels Dateiaktivitäten ist zwar möglich, die nicht-funktionalen Anforderungen jedoch nur durch eine grundlegende Änderung der Aktivitätenverarbeitung und Netzwerkschichten. Von den notwendigen Verbesserungen könnte zwar auch die Aktivitätenverarbeitung selbst profitieren, dafür müsste jedoch dieser zentrale Teil an vielen Stellen überarbeitet werden. Damit spricht dagegen, dass erprobte Funktionalität, die für ihren ursprünglich konzipierten Anwendungsfall ausreichend ist, an vielen Stellen geändert werden müsste. Dem Risiko dabei die Systemstabilität zu beeinflussen steht die Möglichkeit gegenüber eine weitestgehend in sich geschlossene Lösung zu erarbeiten. Dabei kann die Anforderung *FA2 – Bestehende Funktionalität erhalten* besser berücksichtigt werden und eine Umsetzung ist in einem kleineren und damit für diese Arbeit realistischeren Umfang durchführbar. Als weiteren Vorteil erlaubt eine getrennte Lösung eine Optimierung bezogen auf den jeweiligen Anwendungsfall. So müssen nur benötigte Datenfelder übertragen werden und Entscheidungen, wie z. B. die Wahl der Kompressionsumsetzung, sind unabhängig.

Der vor der Analyse erhoffte Vorteil, durch die Verwendung von Aktivitäten – zur Übertragung –, auch automatisch die Schreibblockierung anderer Clienten nicht mehr zu benötigen, trifft zudem nur eingeschränkt zu. Ein zentrales Problem des *Instant-Session-Start* ist es zu gewährleisten, dass ein Client nur Änderungen in Form von Aktivitäten erhält, die nach dem Laden einer Datei zum Versand entstehen, sowie das ab diesem Zeitpunkt auch alle folgenden übermittelt werden. Andernfalls entstehen inkonsistente Zustände

durch doppelt ausgeführte oder fehlende Änderungen, die im weiteren Verlauf dieses Kapitel beschrieben werden. Der erste Ansatz wäre dabei, dass der Client alle Aktivitäten vor Erhalt der zugehörigen Datei verwirft. Das dies aufgrund der *Jupiter*-Verarbeitung jedoch nicht ausreichend ist, wird im *Abschnitt 5.4. Schreibblockierung aufheben* erörtert. Es ist also auch mit der Übertragung mittels Aktivitäten notwendig, den Beginn der Zustellung regulärer Aktivitäten je Datei zu steuern und damit entfällt der Vorteil diese Verarbeitung nicht ändern zu müssen.

Java-Datenströme

Die Analyse der vorhandenen Archivübertragung zeigt, dass die für *XMPP*-Kommunikation genutzte *Smack*-Bibliothek, neben dem Versand einer angegebenen Datei, auch die Übergabe eines Java-*Input-Stream* (Datenstrom) erlaubt. Dies ist ursprünglich dafür gedacht eine Datei zu versenden, zu welcher die Anwendung bereits einen *Stream* geöffnet hat. Der Aufruf mittels eines Dateiverweises, wie in der Archivübertragung genutzt, vereinfacht diesen Aufruf lediglich, da er Zusatzinformationen wie die Dateigröße selbstständig ermittelt, den Datei-*Stream* öffnet und übergibt. Durch diesen *Stream* ist es mit geringem Aufwand im Verbindungsaufbau möglich, eine neue direkte Datenverbindung zur Gegenseite zu erstellen. Die Bibliothek behandelt diesen Transfer wie einen normalen Dateiversand, der erst endet wenn der *Stream* geschlossen wird. Da dieser Versand unabhängig von der regulären Aktivitätenverarbeitung ist, entfallen damit sonst notwendige Änderungen (siehe *Projektübertragung mittels Aktivitäten* auf Seite 13). So sind Probleme bei der Übertragung großer Dateien in diesem Fall nicht relevant, da die Priorisierung regulärer Aktivitäten entfällt sowie der Versand für bestehende Clienten nicht beeinträchtigt wird. Wie bei der Archivübertragung wird der Datenversand nur für eine bestimmte Gegenseite initiiert.

Aufgrund der einfachen sowie in der Archivübertragung bereits getesteten effizienten Übertragung mittels der *Smack*-Bibliothek wird daher dieser Kommunikationsweg gewählt. Ein aufgebauter *Stream* überträgt beliebige Daten wie Dateiinhalte oder Metadaten byteweise. Dabei zeigte die Netzwerkanalyse mit *Wireshark*, dass beim Versand über eine Direktverbindung Daten unverändert ohne Kompression oder Verschlüsselung übertragen werden.

Im Datei-Fall wird ein *Stream* immer genau für eine Datei verwendet. Alle Bytes die in diesen *Stream* geschrieben werden, werden dabei auch gleichermaßen in eine Ausgabedatei bzw. Ausgabe-*Stream* geschrieben. Metainformationen, wie Dateiname und die zu erwartende Größe der Datei, werden hierfür beim Verbindungsaufbau als Teil des Aushandlungsprotokolls separat übermittelt. Um einen *Stream* auch für mehrere Dateien zu nutzen, muss ein Protokoll definiert werden. Dieses muss erlauben Inhalte und Metainformationen zusammen als Bytefolge in einem Format zu serialisieren, das bei Empfang auch interpretiert werden kann. Dies wird Teil des ersten Prototyps sein.

Aktivitätenverarbeitung während der Übertragung

Die Aufgabe der eingesetzten Archiverstellung ist es, zum Zeitpunkt des Übertragungsbeginns, ein vollständiges Abbild aller Dateien mit allen Änderungen bis zu diesem Zeitpunkt zu erstellen. Dies ist die notwendige Ausgangslage um einen Dateizustand zu synchronisieren, auf dessen Grundlage im Anschluss alle nachfolgenden Aktivitäten fehlerfrei angewendet werden können.

Um einen konsistenten Datenstand während der Erstellung des Abbildes zu gewährleisten, werden zwei Vorkehrungen getroffen: Zum einen wird für den Erstellungszeitraum jede Dateiänderung unterbunden und zum anderen werden alle Dateien gespeichert. Durch das Speichern wird gewährleistet, dass das Dateisystem auch alle ungespeicherten Änderungen des Editors enthält. Dies ist notwendig da alle Inhalte zur Archiverstellung vom Dateisystem geladen werden und der Client sonst keine Kenntnis von Änderungen seit dem vorhergehenden Speichervorgang hat. Das Zwischenspeichern von Aktivitäten die zu diesen Änderungen gehören, um damit auf den Speichervorgang während der Abbilderstellung verzichten zu können, wurde bisher nicht vorgesehen und wird in der Umsetzung nicht betrachtet. Die Möglichkeit die ungespeicherten Inhalte des Editors zu übertragen wäre zwar möglich, dadurch würde der Client jedoch eine im Vergleich zum Dateisystem des Host abweichende Version speichern. Der dabei auftretende Unterschied der Dateiprüfsummen würde wiederum den *Consistency-Watchdog* alamieren.

Der Aktivitätenversand wird bisher immer bezogen auf ein Eclipse-/Entwicklungsprojekt gesteuert. Durch das Starten des Versands an einen neuen Clienten erhält dieser fortlaufend alle Änderungen innerhalb dieses Kontext. Wird nun eine Archivdatei erstellt in der Änderungen nach Beginn der Erstellung enthalten sind, ist es für den Clienten nicht nachvollziehbar welchem Versionsstand eine Datei entspricht. Es ist somit nicht eindeutig möglich zu bestimmen, ob eine erhaltene Aktivität bereits in der empfangenen Datei enthalten ist. Dies führt somit zu Inkonsistenzen und um diese *Race-Condition* zu verhindern wurde die dafür sicherste Möglichkeit gewählt und der Schreibzugriff für alle Clienten unterbunden.

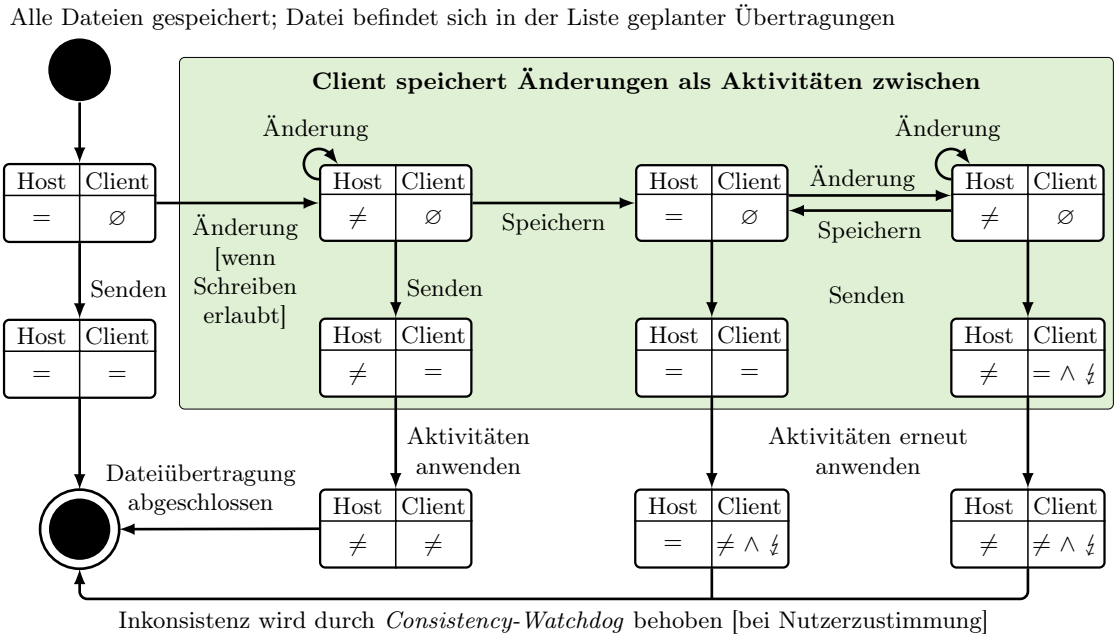


Abbildung 5.1: Zustände und Übergänge einer Datei, im Zeitraum einer Projektübertragung für Host- und Client-Seite. Wenn der Editorinhalt dem Stand des Dateisystems entspricht wird dies mit einem = gekennzeichnet. Eine Ungespeicherte Änderung, die nur dem Editor bekannt ist, mit ≠. Ein ∅ gibt an, ob die Datei noch nicht erhalten wurde und ↯ kennzeichnet eine Inkonsistenz. Inkonsistenzen treten auf, wenn der Host Aktivitäten anwendet und diesen neuen Stand sowie die angewendeten Aktivitäten überträgt. Der Client führt daraufhin – bei Erhalt der Datei – diese Aktivitäten erneut aus. Damit wird der *Consistency-Watchdog* alarmiert, der fortlaufend alle Dateiprüfsummen mit dem Host abgleicht. Dieses Problem entsteht dabei nur, wenn nach dem Beginn ein Speichervorgang ausgelöst wird. Um diesem vorzubeugen findet während der Archiverstellung eine Schreibblockierung für alle Teilnehmer statt.

Um die bisher gewählte Schreibblockierung zu entfernen sind Änderungen an der Aktivitätenverarbeitung notwendig. In der bestehenden Verarbeitung wird vor dem Erstellen der Archivdatei begonnen jede Aktivität an den Client zu versenden damit dieser keine Änderung versäumt. Der Start dieser Zustellung ist bisher nur projektbezogenen möglich. Um auf Dateiänderungen reagieren zu können ist eine zusätzliche dateibasierte Lösung notwendig. Diese soll gewährleisten, dass inkonsistente Zustände nicht entstehen können, indem der Client keine bereits angewandten Aktivitäten erhält. Dieser Ansatz behandelt das Problem hostseitig, wodurch Netzwerkkapazitäten eingespart werden. Da das Entfernen der Schreibblockierung getrennt von der Übertragung bearbeitet werden kann, wird dies Teil eines späteren Prototyps.

5.2 Prototyp

Auf Basis der Vorplanung wurde der erste Prototyp entworfen. Dafür musste zunächst eine neue Implementierung der abstrakten Klassen *AbstractOutgoingProjectNegotiation* und *AbstractIncomingProjectNegotiation* erstellt werden. Diese sind ursprünglich in der Arbeit von Fehling entstanden und wurden von Brekenfeld zur Integration in den Hauptentwicklungszweig aufgegriffen. Die Umsetzung dauerte, einschließlich Abschluss der Reviews, bis Mitte der sechsten von zehn Bearbeitungswochen dieser Arbeit. Da die Änderungen die Grundlage einer neuen Implementierung sind und die Umstellung des Reviewsystems (von *Gerrit* auf *GitHub*) in der siebten Woche stattfanden, verzögerte sich das Review dieser Umsetzung. Somit waren die Änderungen zwar in der fünften Woche bereit, aber blockiert.

Trotz der genannten Einschränkung war es möglich einen Prototyp zu erstellen. Dafür wurden zunächst die Klassen *InstantOutgoingProjectNegotiation* und *InstantIncomingProjectNegotiation* als neue Implementierungen der genannten abstrakten Klassen erstellt. Die Benennungen folgen dabei vorhanden Schemata und nutzen *Instant* als zentralen Begriff um die Zuordnung zu erleichtern. In den Klassen wurde nun zunächst ein zur Archivübertragung ähnlicher Aufbau gewählt, um einen Dateitransfer zu starten, der einen *InputStream* auf Host sowie *OutputStream* auf Client Seite zur Verfügung stellt.

Host-Anteil

Nachdem diese Verbindung aufgebaut werden konnte musste, wie bereits aus der Vorplanung bekannt, ein Protokoll entworfen werden, das mehrere Dateien behandeln kann. Die für den Clienten notwendigen Daten sind Projektzuordnung, Dateiname sowie Dateiinhalt. Da diese Daten eine flexible Länge besitzen, muss dies beim Serialisieren berücksichtigt werden, damit der Empfänger die Daten auch korrekt interpretieren kann. Eine dafür einfache und zuverlässige Methode ist es, zuerst die Länge des darauf folgenden Datenfelds und dann das Datenfeld selbst zu senden. Die Klasse *DataInputStream* bietet die Möglichkeit Java-Datentypen bei Aufruf automatisch in Bytefolgen umzuwandeln und an einen gekapselten Datenstrom zu übergeben. Dabei übernimmt die Klasse das Serialisieren von *Java-Strings* komplett und fügt die Längenangaben selbstständig ein. Daher ist es nur notwendig zum Versand die beiden *Strings*, eine Längenangabe der folgenden Datei sowie die Datei selbst zu übergeben. Dieser Vorgang wird für jede Datei der Übertragungsliste ausgeführt und am Ende mit einem leeren *String* abgeschlossen. Das Schreiben und Lesen der *Streams* ist Teil der Klassen *OutgoingStreamProtocol*, sowie *IncomingStreamProtocol*. In der ersten Ausführung des Prototyps waren mir die *Java-Stream*-Klassen noch nicht im Detail bekannt, daher wurden im Verlauf immer neue Möglichkeiten gefunden um die Umsetzung zu vereinfachen. So serialisierte ich anfangs die Daten mittels der Klasse *ByteBuffer* oder nutzte die Byte Konvertierung der String Klasse, weshalb die Länge noch getrennt behandelt werden musste.

Client-Anteil

Der beim Clienten eingehende *Stream* musste nun noch interpretiert werden und mit diesen Daten die entsprechenden Dateien angelegt. Dabei trat folgende Herausforderung auf: Die Schnittstelle in *Saros-Core* zum Anlegen von Dateien – die im wesentlichen der *Eclipse*-Schnittstelle entspricht – liest einen übergebenen Datenstrom ohne die Möglichkeit einer Begrenzung immer bis zum Ende ein. Zunächst umging ich dieses Problem, indem ich stattdessen die Java-Dateisystem-Schnittstelle nutzte. Diese bietet einen *OutputStream* an und unterstützt damit das *Byte-Array*-weise schreiben von Daten in eine Datei, ohne diese automatisch zu schließen. Damit war es möglich, dass der Prototyp innerhalb einer Schleife zunächst Daten blockweise aus einem *InputStream* las, um diese dann in den *OutputStream* zu schreiben. Durch die Wahl eines kleinen Zwischenspeichers, war die Anwendung nur für den Verarbeitungszeitraum von 8 KiB Blöcken blockiert und konnte vor jeder nächsten Iteration prüfen, ob die Verarbeitung vorzeitig beendet werden soll (Details in *Unterbrechbare Datenströme* auf Seite 28). Um dieses Problem zu lösen, suchte ich eine Möglichkeit in einer der bereits verwendeten Bibliotheken und entdeckte die dafür passende Klasse *BoundedInputStream* aus der *Apache-Commons-IO* Bibliothek. Diese nutzt die Möglichkeit, gemäß dem *Decorator*-Entwurfsmusters[1], einen vorhanden *Stream* zu kapseln und bei Abruf der Daten nur eine vorher bestimmte Menge zurück zu liefern. Zudem kann das automatische Schließen bei *Stream*-Ende – ein Standardverhalten bei *Java-Streams* – von gekapselten *Streams* unterbunden werden, um die Verarbeitung der Übertragung nicht zu beenden. Dieses Verhalten wird Zusammengefasst zum Zerschneiden des *Stream* in die Teile genutzt die zum Schnittstellen Aufruf benötigt werden.

Zusammenfassung des ersten Prototyps

Der Prototyp war in seiner ersten Ausführung noch sehr provisorisch in die bestehenden Abläufe eingebunden, zeigte jedoch, dass die geplante Umsetzung funktioniert. Zudem testete ich bereits, ob das Integrieren der *Zstandard*-Komprimierungsbibliothek funktioniert und die ersten Versuche waren viel versprechend (Seite 32). Des Weiteren entfernte ich probeweise bereits Teile der Schreibblockierung, um ein besseres Verständnis für die kritischen Verarbeitungsabläufe zu erhalten.

Verbesserung des Prototyps

Auf den ersten Prototyp folgend, entstand eine übersichtlichere und zuverlässigere Version, die auch die Behandlung von Ausnahmesituationen und weiteren Funktionen übernahm. Damit die neue Übertragungsart auch aufgerufen wird, ist es notwendig die Aushandlung dieser anzupassen. Die Entscheidung darüber, welche Variante gewählt wird,

[1] [9, S. 175ff]

trifft die bestehende Fabrikklasse *NegotiationFactory*, die vom *SarosSessionManager* verwendet wird. Dieser übergibt dabei den ausgehandelten *TransferType*, dessen Aushandlung später beschrieben wird.

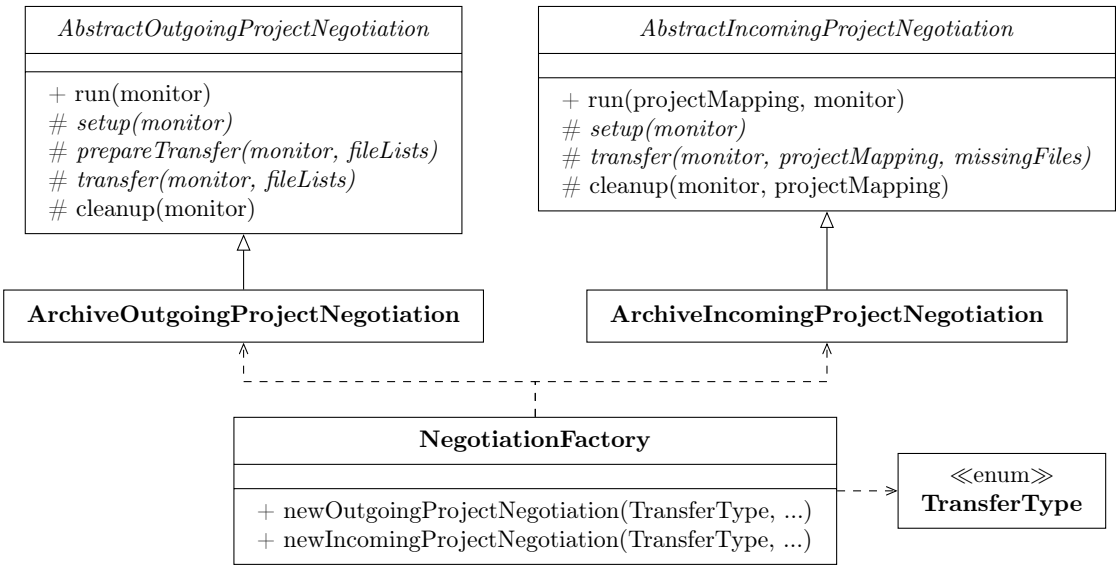


Abbildung 5.2: Bestehende Klassenstruktur zur Instanziierung von Objekten einer konkreten Übertragungsart. Die *NegotiationFactory* wird vom *SarosSessionManager* genutzt um eine Implementierung für den zuvor ausgehandelten *TransferType* zu erhalten. Diese Implementierungen erben von einer Abstrakten Klasse, die in ihrer *run*-Methode die Verarbeitung steuert und dabei alle in den Implementierungen realisierten Methoden aufruft. Diese Struktur beruht auf dem *Gang-of-Four*-Entwurfsmuster *Template* [9, S. 325ff].

Meine Umsetzung fügt sich diesem Prozess, in dem ein neuer *TransferType* erstellt wurde und die genannte Fabrik diesen bei der Auswahl berücksichtigt, um Objekte der Implementierung zur Instantübertragung zurückzugeben.

Priorisierung

Damit die Übertragung auch potentiell gewünschte Dateien zuerst überträgt, wurde ein einfacher Priorisierungsalgorithmus gewählt. Zum einen wird vor der Übertragung die Dateiliste hierarchisch sortiert und Unterordner damit zuletzt übertragen. Zum anderen werden geöffnete Dateien während der Übertragung fortlaufend – sofern noch nicht übertragen – zu erst versendet. Dafür wird ein *Listener* angelegt, der fortan über alle neu geöffneten Dateien aller Teilnehmer informiert wird sowie die aktuelle Liste abfragt. Dabei werden geöffnete Dateien des Hosts priorisiert. Der Host iteriert dann über die sor-

tierte Liste, arbeitet vor dem nächsten Eintrag die Liste der priorisierten ab und prüft, dass jede Datei nur einmal übertragen wird.

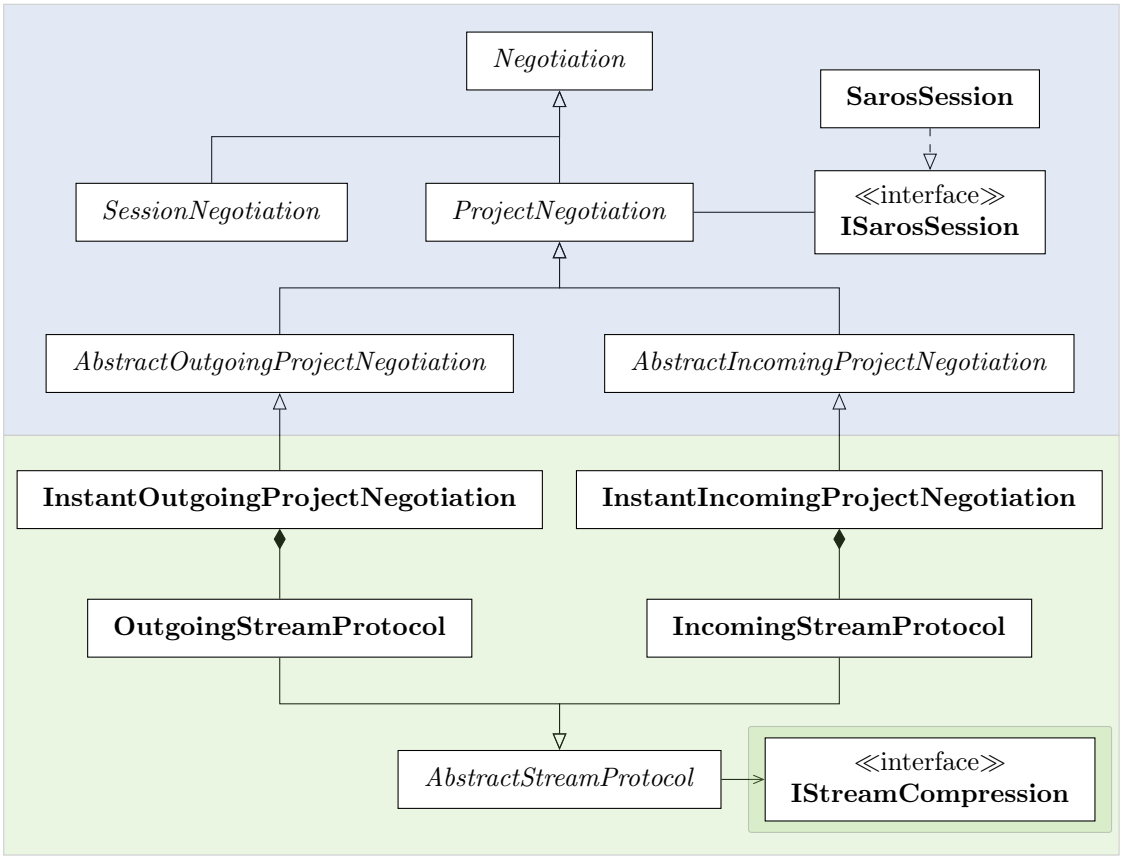


Abbildung 5.3: Ausschnitt der Klassenstruktur zum Übertragungsprozess im *Saros-Core*. Bestehende Klassen werden blau und entstehende Klassen grün hinterlegt. Aufrufe der Klassen werden primär über den *SarosSessionManager* gesteuert, der den Lebenslauf der *SarosSession* und aller Aushandlungen (*Negotiation*-Klassen) steuert. Die Vererbungsstruktur ist bereits sehr komplex und sollte überarbeitet werden, um die tatsächlichen Abläufe besser widerzuspiegeln. Dies ist aber nicht Teil dieser Umsetzung und letzte Änderungen wurden zudem erst parallel zur Prototypenentwicklung durchgeführt. Das Interface *IStreamCompression* wird später genutzt um eine einfach austauschbare Implementierung zu unterstützen.

5.3 Integration des Prototyps

Nachdem der Prototyp ausgereift war wurde er soweit möglich aufgeteilt damit kleinere Patches entstehen. So konnte zunächst mit den Abschnitten begonnen werden, die weitestgehend unabhängig zur parallelen Entwicklung sind.

Aushandlung der Übertragungsart

Damit NutzerInnen die verwendete Übertragungsart beeinflussen können, ist eine neue Einstellungsoption notwendig. Im bestehenden Saros-Einstellungmenü der Eclipse-Erweiterung wurde dafür in der Kategorie *Advanced* die Auswahlbox *Prefer Instant Session Start [experimental]* angelegt und alle Klassen zur Visualisierung und Abfrage des Feldes angepasst. Diese Änderung basierte auf einem Größeren Patch von Fehling[2], aus dem ich notwendige Anpassungen extrahierte und Variablen umbenannte.

Damit das geschaffene Auswahlfeld auch Einfluss auf die Sitzungsaushandlung nimmt, ist es notwendig diese Aushandlung anzupassen. In Saros werden Aushandlungen über den *SessionNegotiationHookManager* verwaltet, der eine Liste mit Verweisen zu allen zu berücksichtigenden *Hooks* sammelt und bei Bedarf bereitstellt. *Hooks* implementieren das Interface *ISessionNegotiationHook* und melden sich während ihrer Erstellung selbstständig im Manager an. Sie haben zur Aufgabe Einstellungen für die Aushandlung zurückzugeben und Antworten des Partners zu verarbeiten. Ein *Hook* ist ein *Singleton*-Objekt, wird also nur einmal erstellt und um dies zu gewährleisten vom *Pico-Container* verwaltet. Dieser instanziiert – unter anderem – alle in der Klasse *CoreContextFactory* konfigurierten Klassen zum Start von Saros.

Wird ein Sitzungsstart ausgeführt, werden diese *Hooks* genutzt um Konfigurationen zu erstellen die zwischen beiden Parteien ausgetauscht werden. Dabei entscheidet der Host am Ende welche Einstellungen final genutzt werden. Die Aushandlung der Übertragungsart findet mittels des *ProjectNegotiationTypeHook* statt, die auf die im vorhergehenden Patch erstellte neue Einstellung zugreift. Am Ende der Aushandlung wird, wenn beide Partner den *Instant-Session-Start* gewählt haben, die Wahl für selbigen Clienten gespeichert und fortan für Projektaushandlungen der Sitzung genutzt. Andernfalls wird die Archivübertragung genutzt. Die Klassen für die Umsetzung waren bereits vorhanden, es musste jedoch ein neuer Wert *Instant* im Enum *TransferType* angelegt und bei allen Abfragen im *ProjectNegotiationTypeHook* sowie in der *NegotiationFactory* berücksichtigt werden.

Damit war die Grundlage geschaffen eine weitere Übertragungsart mit dem Namen *Instant* zu implementieren, ohne die Archivverarbeitung zu beeinflussen und die Anforderung *FA2 – Bestehende Funktionalität erhalten* wurde umgesetzt.

Um die neue Auswahlmöglichkeit auch im *Saros-Test-Framework* automatisiert zu konfigurieren und den *Instant-Session-Start* zu testen, legte ich dafür einen zusätzlichen Patch an. Zum Testen wird dafür, von einer Auswahl an bestehenden Test Klassen geerbt und vor dem Teststart die Übertragungsart gewechselt. Die Vererbung gewährleistet in diesem Fall, dass Änderungen am geerbten Testfall automatisch angewendet werden und kein Code dupliziert wird.

[2] Feature toggle for the ProjectNegotiation strategies – `https://saros-build.imp.fu-berlin.de/gerrit/3067`

Unterbrechbare Datenströme

Um Daten – in Form von *Bytes* – zwischen Abläufen einfach und effizient auszutauschen, stellt die Java-Standardbibliothek die abstrakten Klassen *InputStream* und *OutputStream* – also Ein/Ausgabe-Datenströme – bereit. Eine Verarbeitung die ein Objekt der Klasse *InputStream* entgegen nimmt ruft dessen *read*-Methoden auf, um eine gewünschte Menge an *Bytes* einzulesen. Aufgabe der implementierenden Klasse ist es diese Daten dann fortlaufend bereit zu stellen. Eine klassische Verwendung dieses System ist eine Datei einlesende Klasse, die solange Daten liefert bis die komplette Datei ausgeliefert wurde. Ein großer Vorteil dieser Technik ist, dass konsumierende Methoden – solange sie auf die abstrakte Klasse verweisen – unabhängig von der konkreten Implementierung sind. Nachteilig an diesem Vorgehen ist jedoch, dass in den meisten Fällen der Aufruf einer verarbeitenden Methode blockierend ist. Dies war bereits ein Problem im Prototypen mit der *Saros-Core* Schnittstelle zum Speichern von Dateien. Dabei ist es bis zur Vollständigen Abarbeitung des *InputStream* nicht ohne Umwege möglich eine Verarbeitungen vorzeitig zu beenden, was bei großen Dateien oder langsamen Übertragungen jedoch wünschenswert ist. Die Saros Programmiervorgaben geben für jede länger als 100 Millisekunden dauernde Verarbeitung zudem vor, dass diese auf Abbrüche reagiert.

Damit NutzerInnen über den aktuellen Verarbeitungszustand informiert werden, bietet Saros einen Fortschrittsmonitor an, der über das Interface *IProgressMonitor* gesteuert wird. Dieser besteht aus Informationstexten über den aktuellen Stand, einem Fortschrittsbalken und Schaltflächen um den Dialog in den Hintergrund zu verbergen oder anzuweisen, dass der aktuelle Vorgang beendet werden soll. Eine Verarbeitung hat dann die Aufgabe diesen Wunsch regelmäßig beim Monitor abzufragen und zu berücksichtigen.

Daher überlegte ich, wie dies erreicht werden kann, ohne für jeden verarbeitenden Aufruf einen neuen *Thread*, mit dazu notwendiger Abfrage-Schleife des Status zu erstellen. Eine *Thread*-Erstellung wäre vor allem ineffizient, da dieses Verfahren bei Dateiübertragungen häufig aufgerufen wird und dafür relativ viel Zeit benötigt wird. Im ersten Prototyp umging ich dieses Problem zunächst, indem für den Dateizugriff anstatt der *Saros-Core*-Schnittstelle die Java-Dateisystem-Schnittstelle genutzt wurde. Diese Lösung war jedoch nicht optimal, da bei Nutzung der Java-Schnittstelle die Datei im Anschluss manuell der Entwicklungsumgebung bekannt gemacht und der dafür notwendige Schnittstellenaufruf zunächst implementiert werden muss. Des Weiteren enthielt die Archivübertragung bereits einen alten Aufgabenverweis, da dieses Problem auch dort existiert. Daher suchte ich nach einer wiederverwendbaren und allgemeineren Lösung.

Da ich im Rahmen von anderen aufrufen bereits *InputStreams* ineinander verschachtelte, kam mir die Idee eine Klasse *CancelableInputStream* zu erstellen. Diese ist eine Implementierung der abstrakten Klasse *InputStream* und kapselt diese gemäß dem *Decorator*-Entwurfsmuster, um zusätzliche Funktionalität vor der Ausführung einer Methode zu realisieren [9, S. 175ff]. Dafür wird einem *CancelableInputStream* zur Erstellung ein *InputStream* und *IProgressMonitor* übergeben, die intern gespeichert werden. Wird nun eine *CancelableInputStream* Methode aufgerufen, ruft diese die gleiche Methode des

gekapselten *Stream* auf und prüft bei allen lesenden aufrufen den Monitor auf einen Abbruchwunsch. Ist dies der Fall, wird eine Ausnahme ausgelöst, welche die Abarbeitung damit vorzeitig beendet. Diese Klasse bietet damit eine wiederverwendbare Möglichkeit, jede *InputStream*-Verarbeitung frühzeitig zu beenden und wurde auch in die Archivübertragung eingebunden. Zudem war dieser Patch[3] eine gute Möglichkeit, diese später wiederverwendete Funktionalität bereits unabhängig einzuführen.

Integration in die bestehende Struktur

Der verbleibende Teil des entstandenen Prototyps wurde nun den anderen Änderungen nachfolgend eingebracht. Die abstrakten *Project-Negotiation* Klassen wurden für den Fall konzipiert, dass – zusätzlich zur vorhandenen – eine aktivitätenbasierte Lösung entsteht. Dafür wurden unter anderem die auf der *Smack*-Bibliothek basierenden Codeteile zur Dateiübertragung in die Klassen der Archivübertragung verschoben. Da diese nun in der Instantübertragung benötigt werden, musste zwischen den Möglichkeiten abgewogen werden den gleichen Code zu kopieren, eine weitere Vererbungsebene einzuführen oder den Code in die abstrakten Klassen zu verschieben. Ich entschied mich für die letzte Option, da die abstrakten Klassen damit zwar wieder mehr Funktionalität übernahmen, dies jedoch Codeduplizierung und eine weitere Erhöhung der Komplexität verhindert. Diese Umsetzung war nicht unumstritten und ist aus meiner Sicht ein Kompromiss, da eine komplette Umstrukturierung des Aushandlungsteils und dem Zugriff auf die verwendete Bibliothek zwar am sinnvollsten wären, dafür aber nicht genug Zeit vorhanden war und die Priorität beim Beseitigen der Schreibblockierung liegt.

5.4 Schreibblockierung aufheben

Mit der Ausarbeitung des Prototyps, wurde die notwendige Grundlage geschaffen auf der nun weitere Anforderungen umgesetzt werden. Dabei ist ein wichtiger Punkt das Aufheben der Schreibblockierung für die bereits teilnehmenden Personen (siehe *FA3 – Schreibblockierung aufheben*).

Um dies umzusetzen wurde ein neuer Prototyp begonnen und zunächst die komplette Schreibblockierung entfernt. Aus den Vorüberlegungen (Seite 21) war bereits bekannt, dass die Aktivitätenverarbeitung pro Datei gesteuert werden muss. Dabei sind Host und Client getrennt zu betrachten, da hier unterschiedliche Verarbeitung beeinflusst wird.

[3] Github Pull-Request #125: `https://github.com/saros-project/saros/pull/125`

Client

Der Client muss bei Erhalt einer Aktivität entscheiden, ob diese bereits angewendet werden kann. Dafür existierte bereits die Klasse *ActivityQueuer*, die alle eingehenden Aktivitäten gegen eine Liste von bereits synchronisierten Projekten prüft und falls diese fehlen in eine Warteschlange aufnimmt. Diese Warteschlange wird für die Archivübertragung zu Beginn des Vorgangs aktiviert und nach Abschluss werden alle Aktivitäten sequentiell ausgeführt. Im Quellcode vorhandene *TODO*-Markierungen empfahlen, dass diese Steuerung Teil der *SarosSession* werden sollte, was jedoch zum Vorteil für die folgende Umsetzung nie durchgeführt wurde.

Die wichtigste Methode der *ActivityQueuer*-Warteschlange ist die *process*-Methode, die die eingehenden Aktivitäten der *SarosSession* entgegen nimmt. Damit die Archivübertragung auch weiterhin diese projektbasierte Warteschlange nutzen kann, erstellte ich ein Interface *IActivityQueuer* das einzig die *process*-Methode fordert. Daraufhin benannte ich die *ActivityQueuer* Klasse in *ProjectActivityQueuer* um, damit ihr Anwendungsbereich erkennbar ist und fügte das Interface hinzu. Die *SarosSession* änderte ich nun dahingehend, dass jede Klasse die das Interface implementiert als Warteschlangenimplementierung angemeldet werden kann. Somit war es nun möglich eine neue *ResourceActivityQueuer* Klasse anzulegen, die ebenfalls dieses Interface Implementiert. Diese nimmt eine Liste von Dateien (Ressourcen) entgegen, die noch nicht synchronisiert sind. Die Aufgabe der Instantübertragung war es dann diese Warteschlange vor dem Start der Übertragung einzurichten und nach Erhalt einer Datei darüber zu informieren. In der ersten Umsetzung wurde, zum Test ob die Aktivität angewandt werden kann, nur die Existenz einer Datei geprüft. Beim Synchronisieren mit einem bereits vorhandenen Projekt löst dies das Problem jedoch nur unzureichend. Dann ist es möglich, dass diese Datei bereits vorhanden ist aber im laufe der Übertragung ausgetauscht wird und damit bereits angewandte Aktivitäten verfallen. Durch die Verlagerung der Warteschlangenerstellung in die Aushandlung ist es möglich, dass die sehr unterschiedlichen Konfigurationsmethoden der Warteschlangen nicht Teil des bereits sehr umfangreichen *ISarosSession*-Interface werden. Daher besitzt das Interface auch nur eine Methode.

Meine Annahme, die Projektaushandlung würde wie die Sitzungsaushandlung nicht parallel für einen Clienten ausgeführt, stellte sich im Review jedoch als falsch heraus. Daher muss für diesen Fall noch geklärt werden, ob es sinnvoll ist mehrere parallele Übertragungen – für einen Clienten – zu unterstützen oder ob die Verarbeitung daran angepasst werden muss. Ein Vorteil einer sequentiellen Übertragung wäre, dass Übertragungen nicht um die Bandbreite konkurrieren würden und mit dieser Änderung könnte man wartende Übertragungen leichter zusammenfassen. So wäre gemäß dem Fall eine Person fügt – in einem partiell geteilten Projekt – mehrere Ordner zur Sitzung hinzu, nicht für jeden Vorgang eine eigene Aushandlung notwendig. In diesem Fall müsste die eingeladene Person auch nicht jedem Vorgang einzeln zustimmen.

Host

Da der Client nun – im Falle einer einzigen Übertragung – korrekt auf den Erhalt wartet, muss zudem gewährleistet werden das nur Aktivitäten zugestellt werden, die nicht bereits angewendet wurden (siehe Zustandsdiagramm Abbildung 5.1 auf Seite 22). Dafür wird auf Host-Seite eine Ähnliche Variante gewählt, jedoch geht es dabei um die Erzeugung von Aktivitäten.

Der Host instanziiert für jede – in der Sitzung bearbeitete – Datei einen *JupiterDocumentServer* der für die Synchronisierung einer Datei mit den Clienten zuständig ist. Das hinzufügen zu einem Server wird wiederum vom *JupiterServer* koordiniert, der alle Dateien und dazugehörigen *JupiterDocumentServer* verwaltet. In der bisherigen Verarbeitung wird vor dem Beginn einer Übertragung nun dafür gesorgt, dass der neue Client zu allen aktuellen und entstehenden Instanzen hinzugefügt wird.

Beim Hinzufügen eines Clienten beginnt für diesen auch das Zählen der *JupiterVectorTime*, getrennt für jede Datei. Dieser Zähler besteht aus einem Host- sowie einem Client-Teil und wird als Referenz für jede Änderung erhöht. Wird nun eine Aktivität für einen verwalteten Clienten einfach verworfen, kann der Client die darauf folgenden Aktivitäten nicht verarbeiten, da beide beim gleichen initialen Wert starten und der Client immer auf die Erhöhung um einen Wert prüft.

Daher muss der Zeitpunkt des Hinzufügens eines Clienten zu einem *JupiterDocumentServer* kontrolliert werden. Dafür entwickelte ich folgenden Algorithmus:

1. Das Hinzufügen des Client zu *JupiterDocumentServern* wird für alle zu übertragenen Dateien zunächst unterbunden.
2. Die Projekte der Aushandlung werden der Aktivitäten-Erstellung im *JupiterServer* hinzugefügt, jedoch werden fehlende Dateien nicht in die Verarbeitung für den Clienten aufgenommen.
3. Zum Versandzeitpunkt wird die Datei zunächst gespeichert.
4. Die Datei wird zum Lesen geöffnet. Die Schnittstelle garantiert dabei, dass sich der Dateiinhalt ab Aufruf für den Lesevorgang nicht verändert.
5. Die Datei wird zu bestehenden sowie kommenden *JupiterDocumentServern* hinzugefügt, wodurch ab diesem Zeitpunkt Aktivitäten für den Clienten erstellt werden.
6. Die Datei wird übertragen, der Client fügt Aktivitäten die vor dem Erhalt der Datei entstehen der Warteschlange hinzu.
7. Client hat Datei erhalten und wendet alle wartenden sowie fortan neuen Aktivitäten für diese Datei sequentiell an.

Damit war es nun möglich, das bereits synchronisierte Teilnehmer nicht blockiert werden müssen und fortlaufend Änderungen vornehmen können, ohne einen inkonsistenten Zustand zu verursachen. Somit wurde der zweite Meilenstein *schnelles lesendes Teilnehmen*

möglich erreicht und ein Teil der Anforderung *FA1 – Frühzeitiges Teilnehmen* erfolgreich umgesetzt.

Mit der Umsetzung des dritten Meilenstein *schreibendes Teilnehmen nach Start* wurde vorerst nicht begonnen. Die Klasse *FileReplacementInProgressObservable* wird vom *Consistency-Watchdog*-Dienst genutzt, um auf dem Clienten alle Schreibvorgänge – während eines Dateiaustausches – zu blockieren. Dies wird genutzt um erkannte Inkonsistenzen aufzulösen und während dieses Vorgang keine neuen durch weitere Änderungen zu ermöglichen. Diesen Vorgang zu ändern bedarf einer weiteren Analyse.

5.5 Komprimierung

Ein wichtiger Einflussfaktor auf die Geschwindigkeit des Aushandlungsprozess, vor allem bei kleinen Bandbreiten, ist die Komprimierung. Aus diesem Grund nutzt die Archivübertragung eine ZIP-Datei, die neben der einfachen Bündelung von Dateien auch die Inhalte komprimiert. Die dabei verwendete Komprimierung beruht auf der *ZLIB*-Bibliothek, die bereits 1996 das erste mal im RFC 1950 „standardisiert“ wurde [10]. Die Java-Standardbibliothek bietet diese auch zur Stream-komprimierung an, jedoch existieren mittlerweile effizientere Möglichkeiten.

Eine davon ist die von Facebook entwickelte Bibliothek *Zstandard*[4], die quelloffen und unter freien Lizenzen (BSD, sowie GPLv2) verfügbar ist. *Zstandard* bietet aktuell 22 unterschiedliche Komprimierungsstufen, die je nach Grad eine höhere Kompressionsrate erzielen, jedoch zulasten der Verarbeitungsgeschwindigkeit. P. Skibiński zeigte in seinen Tests, dass die *Zstandard*-Bibliothek neben einer sehr viel höheren Geschwindigkeit auch eine bessere Kompressionsrate liefert [11]. Dies zeigte sich auch in meinen ersten Tests auf Kompressionsstufe 3. So entstand, bei Übertragung des 6 MB großen Eclipseteil von Saros, eine 3,6 MB große Archivdatei. Der komprimierte *Stream* war dabei nur 3,1 MB groß.

Da dieses Ergebnis bereits sehr gut ist, wird in dieser Arbeit kein weiterer Fokus auf die Optimierung gelegt. Damit die Implementierung leicht austauschbar ist, wurde dafür das Interface *IStreamCompression* angelegt, das die Verarbeitung eines ein- und ausgehenden *Streams* fordert. Dieses Interface wird wiederum in der abstrakten Klasse der *Stream*-verarbeitung verwendet und erhält eine Instanz über den im *Saros-Core* verwendeten *Pico-Container*.

[4] `https://github.com/facebook/zstd`

6 Test

In diesem Kapitel werden aus den Anforderungen des *Kapitel 4. Definition*, die Testszenarien abgeleitet und der Aufbau der erstellten Testumgebung, sowie das Testsystem beschrieben. Dann werden die durchgeführten Tests beschrieben, deren Ergebnisse bewertet und überprüft ob die Datenübertragung weiter verbessert werden kann.

6.1 Ableiten der Testfälle aus den Anforderungen

Um mehrere, möglichst anwendungsnahe Szenarien zu testen, werden existierende Projekte in unterschiedlicher Größe zwischen Host und einem Client ausgetauscht. Die Betrachtungen beziehen sich dabei auf neu aufgebaute Sitzungen. Das Einladen einer neuen Person in eine bestehende Sitzung oder das spätere Hinzufügen eines neuen Projekts sind nur leicht abweichende Fälle, da hier der gleiche bestehende Austauschvorgang erfolgt. Zu diesem Prozess sind bereits Testszenarien im *Saros-Test-Framework* vorhanden und wurden zusammen mit den kontinuierlichen Entwicklertests ausgeführt.

Die Tests fokussieren sich neben Funktionstests zum großen Teil auf Geschwindigkeitstests. Das Generieren von Dateien mit zufälligem binärem Inhalt würde dabei den Kompressionseinfluss der Implementierungen vernachlässigen, daher werden die Java-Projekte von Saros für die Tests genutzt.

Kompression wirkt sich vor allem auf *In-Band-Bytestream*-Verbindungen aus, da sie oft über sehr limitierte Serververbindungen genutzt werden. Bei Standard-Konfigurationen von 1000 Bytes pro Sekunde dauert das Übertragen von 3,6 MB eine Stunde, bei 5000 Bytes somit immer noch 12 Minuten und diese Werte werden in der Realität schwer erreicht. In diesen Fällen können eine gute Kompression und geringer *Overhead* somit viel Zeit sparen. Sie sind zusammen mit dem *Overhead* der Übertragung wichtige Einflussfaktoren.

Aufgrund großer Unterschiede in den Verbindungsarten werden, um diese zu verdeutlichen, Ergebnisse gesondert für Socks5 (im Folgenden Direktverbindungen genannt) und Socks5-Proxy Verbindungen (kurz Proxy) aufgeführt. Die Tests für *In-Band-Bytestream*-Verbindungen (kurz IBB) werden nur teilweise ausgeführt, da stundenlange Übertragungen im späteren Einsatz unrealistisch sind. Die Beschreibungen der Tests sind zusammen mit ihrer Ausführung in Abschnitt 6.4. Testfälle und Ergebnisse aufgeführt.

6.2 Aufbau der Testumgebung

Damit Vergleichswerte möglichst realitätsnah und reproduzierbar sind, wird eine Testumgebung benötigt die Umgebungsbedingungen wie Netzwerkgeschwindigkeit und Verzögerungen beeinflussen kann. Um dies in einem kompakten Versuchsaufbau auf einem Laptop durchzuführen, wurde eine Testumgebung basierend auf *Docker* erstellt. Diese bietet viele Vorteile. So steht, mittels eines bestehenden *Docker-Images*[1], bereits schnell ein eigener *XMPP*-Server zur Verfügung und kann beliebig konfiguriert werden. Dies erlaubt mehr Einfluss auf *IBB*-Übertragungsgeschwindigkeiten und das lokale System unterliegt keinen externen Lastschwankungen oder Internetabbrüchen. Des Weiteren kann man *Docker*-Systeme in unterschiedlichen Netzwerken zusammenfassen und einzelne Netzwerkanschlüsse der *Container* mittels Linuxpaket *iproute2*[2] beeinflussen. Dieses bietet unterschiedliche Programme wie *tc* an, mit dem die Bandbreite limitiert und die Latenz einer Verbindung erhöht werden kann. Die Steuerung dieser Umgebung wurde in einem dokumentierten Shell-Skript umgesetzt und zusätzlich auf meinem *GitHub*-Profil zur Verfügung gestellt[3]. Das Skript prüft die Verfügbarkeit aller Abhängigkeiten, erstellt alle notwendigen *Docker-Container* und konfiguriert Netzwerkverbindungen sowie Limitierungen.

Das Limitieren von ausgehenden Netzwerkpaketen ist mit dem *iproute2*-Paket einfacher umzusetzen als eine zusätzliche für Eingehende. Da die Gegenseite jedoch auch im Versand beschränkt ist, ist dies für die Tests ausreichend. Diese Limitierungen wurden mit dem Programm *iperf3* getestet. Dafür wurde auf einem *Container* ein *iperf3*-Server gestartet und mittels des Clienten auf der Gegenseite die zur Verfügung stehende Bandbreite ermittelt. Dafür sendet *iperf3* für einen Zeitraum fortlaufend Daten und ermittelt die erreichte Übertragungsgeschwindigkeit. Dieser Test war sinnvoll, da die Konfiguration dieser Limitierungen sehr fehleranfällig ist. Zur richtigen Konfiguration müssen Zwischenspeichergrößen gewählt werden, die auch eine fortlaufend konstante Drosselung erlauben. Die Latenz wurde mittels *ping*-Befehl überprüft, der die Laufzeit von Netzwerkpaketen ermittelt.

Die genutzten Übertragungswerte der Saros-Container orientieren sich an der Leistung Marktüblicher VDSL-Anschlüsse. Die Bandbreite der *XMPP*-Instanz selbst fällt nicht ins Gewicht und die Verzögerung ist ähnlich zu dem Ergebnis bei Verbindungen mit dem *XMPP*-Server des Saros-Projekt gesetzt. Die *IBB*-Geschwindigkeit des Server ist auf 5000 Bytes pro Sekunde je Client konfiguriert.

Zur Durchführung der Testläufe wird das *Saros-Test-Framework* mit vorher erstellten Testszenarien genutzt, um eine automatische und genauere Messung zu ermöglichen. Damit *RMI*-Verbindungen, die zum Steuern der Saros-Instanzen genutzt werden, nicht

[1] Dockerfile for Ejabberd server – `https://github.com/rroemhild/docker-ejabberd`
[2] networking:iproute2 [Linux Foundation Wiki] – `https://wiki.linuxfoundation.org/networking/iproute2`
[3] `https://github.com/stefaus/saros-docker`

durch die Limitierungen beeinträchtigt werden, kommunizieren diese über ein getrenntes unlimitiertes Netz.

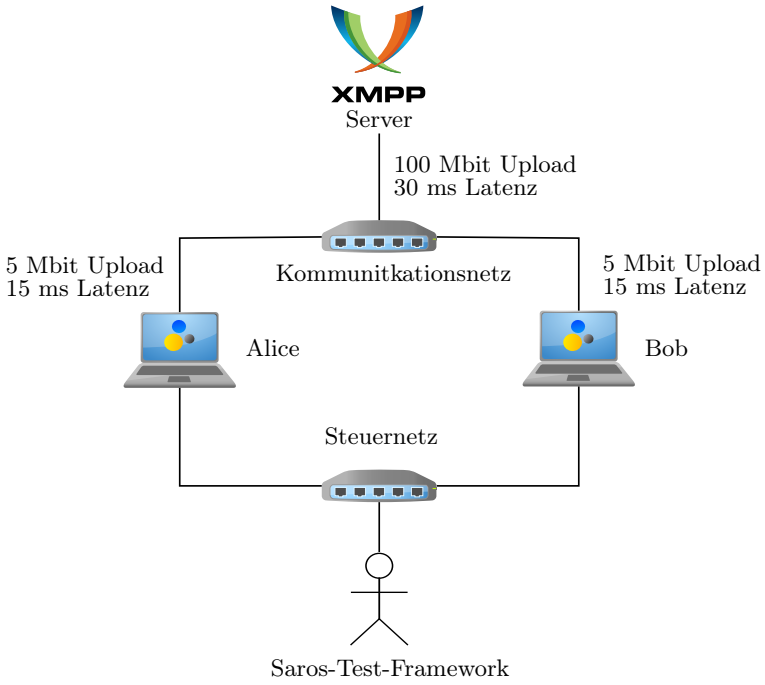


Abbildung 6.1: Dockerbasierte Testumgebung mit genutzten *Containern* und Netzwerken. Neben den zwei abgebildeten Saros-*Containern* können weitere zwei erstellt werden. Dabei wird die Übertragungsgeschwindigkeit jedes *Container* auf 5 Mbit in ausgehender Richtung limitiert und jedes Netzwerkpaket um 15 Millisekunden verzögert. Daraus ergibt sich bei Direktverbindungen zwischen zwei *Containern* eine Gesamtverzögerung von 30 ms. Damit die Steuerung aus dem *Saros-Test-Framework* heraus keinen Beschränkungen unterliegt, ist dieses über ein zweites unlimitiertes Netzwerk angebunden.

6.3 Testsystem

Für die Tests wird ein Laptop mit Intel Core i7-7500U Prozessor, 16 GB Arbeitsspeicher und einer sehr schnellen Solid-State-Disk (SSD) verwendet. Als Betriebssystem kommt *Linux Mint* 18.3 mit Kernel 4.15 zum Einsatz. Alle nicht benötigten Programme sowie Netzwerkverbindungen sind beendet.

Um Schwankungen in der Verarbeitungsgeschwindigkeit zu minimieren, sind alle Stromsparmechanismen des Prozessors deaktiviert und eine zusätzliche Belüftung aufgebaut, damit Testdurchläufe nicht zu stark durch Überhitzung gedrosselt werden. Dies war in ersten Probeläufen der Grund für starke Messausreißer und konnte auch nicht immer verhindert werden. Um eine temperaturbedingte Prozessordrosselung während der Tests zu erkennen, wurden die Sensoren fortlaufend überprüft. Da genug Arbeitsspeicher für alle Dockercontainer und Anwendungen vorhanden ist, ist dadurch kein negativer Einfluss zu erwarten. Das Betriebssystem nutzt den Arbeitsspeicher zum Zwischenspeichern von Dateisystemzugriffen, daher sind Dateizugriffe im ersten Testdurchlauf langsamer als in darauf folgenden, die auf den Gerätezugriff verzichten. Um diesen Einfluss auf die Aussagekraft der Ergebnisse zu minimieren werden Tests mehrfach ausgeführt. Dadurch wird die Zeit für den Gerätezugriff fast irrelevant, da Zwischenspeicher gefüllt werden und dann „warm“ sind. Zudem sind Dateizugriffe durch die verwendete SSD bereits sehr schnell. Dies lässt damit Nachteile der Archivübertragung, die zusätzliche Dateizugriffe zum Speichern der ZIP-Datei nutzt, weniger ins Gewicht fallen.

Durch die Limitierung der Netzwerkgeschwindigkeiten wird auch die Verarbeitung in Saros reglementiert und nicht mehr verarbeitet als aus den Zwischenspeichern entnommen wird. Dies hat einen positiven Effekt auf die Prozessorauslastung, da so während der Übertragungsphasen keine Überlastsituation entsteht und genug Kapazität für alle Container bereit steht. Kurze Lastspitzen wie sie bei der Archiverstellung auftreten konkurrieren nicht übermäßig mit der Verarbeitung in anderen Containern, weil diese währenddessen auf das Ende der Erstellung und den Beginn der Übertragung warten.

6.4 Testfälle und Ergebnisse

6.4.1 Testen funktionaler Anforderungen

Das Erfüllen der funktionalen Anforderungen sowie Testen dieser ist bereits während des Entwicklungsvorgangs Teil der eigenen Entwicklertests. Dafür wurde die aufgebaute Testumgebung genutzt und auf Fehlermeldungen in den *log4j*, sowie Oberflächenausgaben geachtet. Um die Funktionalität auch zukünftig fortlaufend zu testen, wurden automatische Tests im *Saros-Test-Framework* angelegt die mehrere der bestehenden Einladungstests für die neue Übertragungsmethode ausführen.

6.4.2 Testen nicht-funktionaler Anforderungen

Um die aufgestellten nicht-funktionalen Anforderungen zu testen wird die beschriebene Testumgebung genutzt, sowie eigene Tests im *Saros-Test-Framework* definiert, die die Ausführung vereinfachen und automatisiert eine hohe Anzahl von Ausführungen messen können. Alle Angaben bzgl. der Projektgröße beziehen sich auf die Größen der Dateiinhalte, und nicht auf die durch die Speicherblockgröße des Dateisystems beeinflusste Speichergröße, sowie ohne Verwaltungsinformationen wie Dateinamen. Projektdateien aus dem Saros-Projekt basieren auf der gleichen *git*-Version[4] und die Testumgebung ist wie weiter oben beschrieben konfiguriert.

Bei den Tests wurde folgender Effekt festgestellt: Wenn ein Projekt neu in Eclipse importiert wird und das automatische Kompilieren deaktiviert ist, werden Kompilate beim Projektaustausch mit in die Dateiliste aufgenommen und übertragen. Ist dies jedoch aktiviert oder es wird manuell ein Kompiliervorgang gestartet, dann entfallen diese. Da dieses Verhalten als unerwartet eingestuft wird, wurde ein Ticket mit der Nummer 176[5] dafür erstellt. Um Einflüsse auf die Tests zu vermeiden, sind automatische Bauvorgänge in Eclipse deaktiviert und Java-Projekte werden aus einem Verzeichnis ohne Kompilate kopiert.

Zum Testen der QA1 – Zuverlässigkeit, werden die übertragenen Projekte mithilfe des Programm *diff* geprüft. Dabei wird rekursiv die Existenz und Dateiinhalte aller Dateien zwischen Host und Client überprüft, wobei keine Unterschiede auftraten. Zudem wurden in den Tests keine Inkonsistenzmeldungen des Saros-*Consistency-Watchdogs* festgestellt.

[4] commit-id: be075c285681f516f4a9af83d21df92bb8da962c
[5] `https://github.com/saros-project/saros/issues/176`

Übertragungsdauer

Um die NFA1 – Übertragungsdauer zu testen werden folgende Projektstrukturen genutzt:

Test	Info	Dateien	Größe
1	Neues Projekt (Nur Konfigurationsdateien)	3	ca. 1 kB
2	Saros Kern	455	ca. 8,7 MB
3	2 Projekte: Saros Kern + Saros Eclipse	1230	ca. 14,7 MB

Die automatisiert gemessenen Zeiten beginnen ab der Annahme der Projekteinladung auf Client-Seite und Enden mit der ersten in der Entwicklungsumgebung geöffneten Datei, fortan als *Erste Datei* bezeichnet bzw. mit dem vollständigen Ende der Projektübertragung, als *Alle Dateien* bezeichnet.

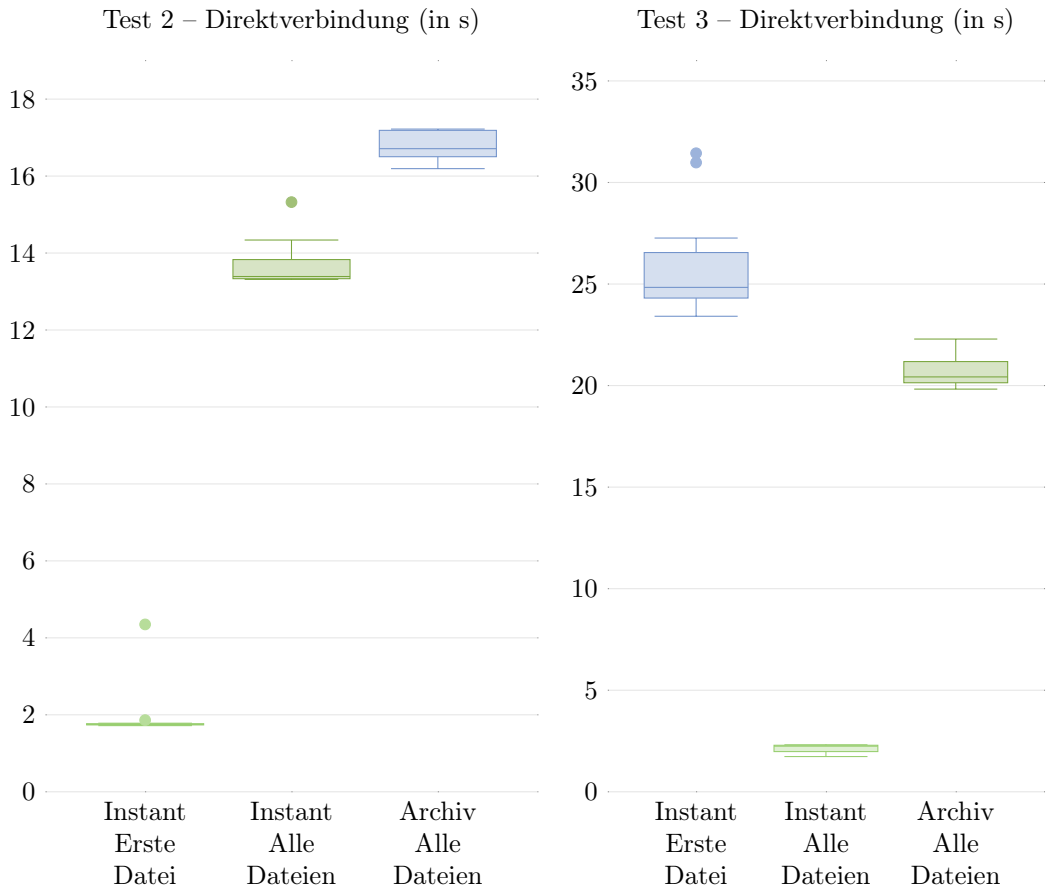


Abbildung 6.2: Quantile der Tests in Reihenfolge ihrer Ausführung. Je Testszenario werden die gleichen Instanzen wiederverwendet und je Art zehn Übertragungen vollständig automatisiert durchgeführt und gemessen.

Alle Tests der Direkt- und Proxyverbindung wurden jeweils 10 mal durchgeführt, was bei drei Testszenarien mit je 20 Durchläufen bereits mehrere Stunden dauerte. Über diesen Zeitraum erwärmte sich das Testsystem zusätzlich durch die hochsommerlichen Temperaturen. Vergleiche zwischen zwei Tagen zeigten Unterschiede von drei bis fünf Sekunden auf und wirkten sich auf beide Übertragungen aus. Um daher die Vergleichbarkeit innerhalb eines Testszenario zu gewährleisten war es notwendig die Tests nacheinander bei gleichen Umgebungsbedingungen auszuführen.

Die Messergebnisse besaßen wie auf den Boxplots zu sehen eine geringe Abweichung, jedoch traten auch wenige Ausreißer auf. Diese waren wie zu erwarten vor allem innerhalb der ersten Durchgänge und sind auf kalte Zwischenspeicher zurückzuführen.

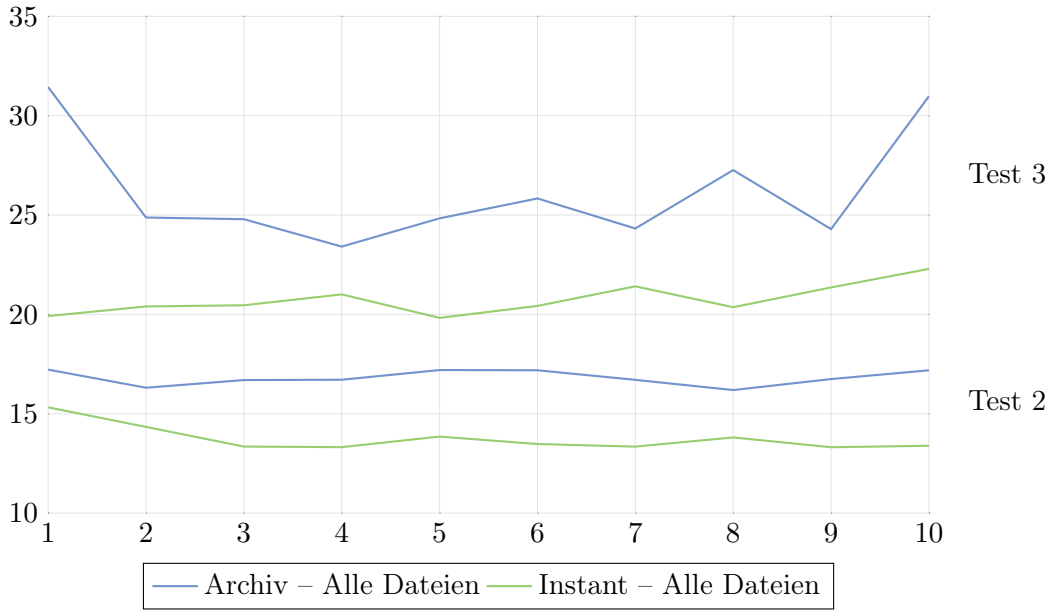
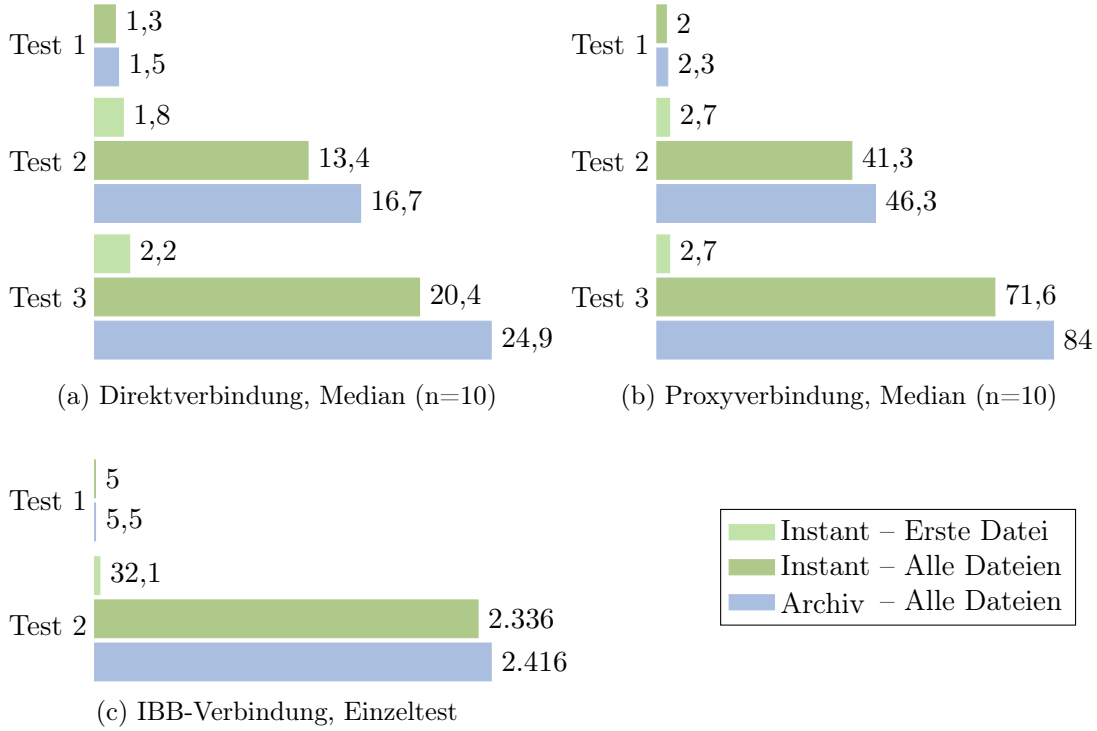


Abbildung 6.3: Verlaufskurven über zehn Durchführungen bei Direktverbindung in Sekunden. In Test 3 wird zuerst die Archivübertragung ausgeführt und zeigt einen schnellen Abfall bereits im zweiten Durchgang. In Test 2 wird die Instantvariante zuerst ausgeführt und ist ab dem dritten Durchgang auf einem niedrigeren Wert. Alle Kurven zeigen einen Verlauf der sich, wie bei Test 3 in der Archivübertragung, mit ein bis zwei steigenden und dann wieder fallenden Werten wiederholt. So steigt ab Durchgang vier die Laufzeit bis Durchgang sechs, fällt bei sieben und steigt wieder. Die Verläufe sind für andere Testdurchführungen ähnlich. Zur besseren Darstellung des Kurvenverlauf beginnt die y-Achse bei zehn Sekunden.

Da die Messdaten nur wenige Ausreißer enthalten, die auf zufällige Ereignisse wie Wärme oder Zwischenspeicher zurückführbar sind, werden diese zur besseren Vergleichbarkeit der Übertragungsvarianten fortan mittels Bildung des Median eliminiert.



(a) Direktverbindung, Median (n=10)

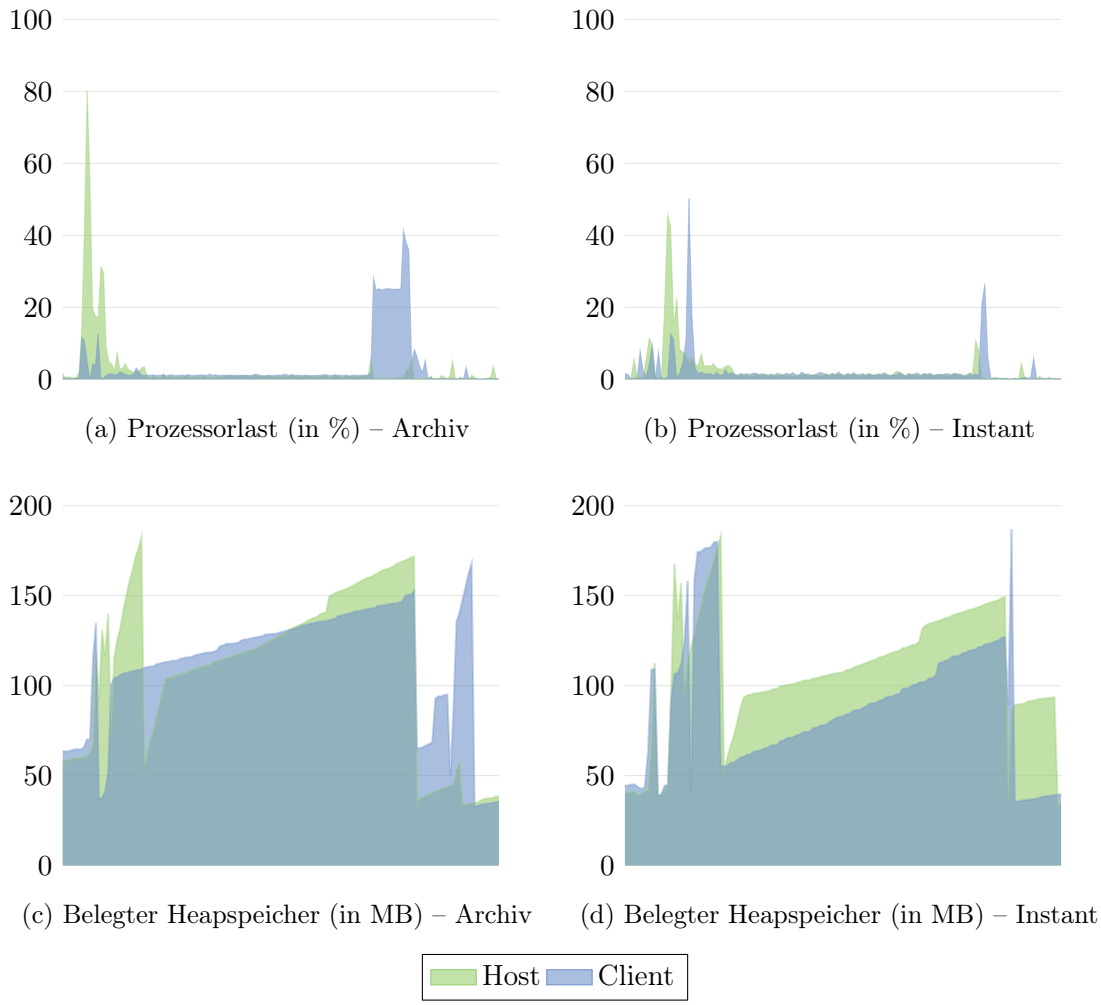
(b) Proxyverbindung, Median (n=10)

(c) IBB-Verbindung, Einzeltest

Abbildung 6.4: Testergebnisse aller Szenarien und Übertragungsarten in Sekunden. Es zeigt sich sich das die erste Datei in allen Messungen innerhalb der ersten Sekunden geöffnet wird und damit weit vor Ende der Gesamtübertragung. Die Instantvariante übermittelt alle Dateien, bei Direkt und Proxyverbindungen, über alle Szenarien betrachtet, im Durchschnitt 14 % früher. IBB-Durchläufe dauern im Vergleich zu Proxyverbindungen ca. 50 mal länger und sind für größere Projekte äußerst ineffizient. Aufgrund der Durchführung als Einzeltest dienen die IBB-Messzeiten lediglich als Richtwert.

Stabilitätstest

Um die Anforderungen *NFA2 – Oberflächenreaktionszeit*, *NFA3 – Ressourcenverbrauch* und *QA2 – Stabilität* bei vielen und großen Dateien zu testen, wurden dem Core-Projekt aus Test 2 zusätzlich 5 mal 25 MB Dateien, sowie eine 100 MB Datei hinzugefügt und die Übertragung wiederum mit der Archivvariante verglichen. Dafür wird das im *Java-Development-Kit* enthaltenen Programm *jconsole* genutzt, das fortlaufend Daten über die gesamte *Java Virtual Machine* (JVM), also inklusive des Aufwands der durch das Ausführen der Entwicklungsumgebung entsteht, sammelt.



(a) Prozessorlast (in %) – Archiv

(b) Prozessorlast (in %) – Instant

(c) Belegter Heapspeicher (in MB) – Archiv

(d) Belegter Heapspeicher (in MB) – Instant

Abbildung 6.5: Ressourcenauslastung der *Java-Virtual-Machine (JVM)* während einer Direktübertragung des zweiten Testszenarios, für beide Übertragungsarten. Die Auslastung des Heap muss zurückhaltend interpretiert werden, da seine Größe durch den unregelmäßig arbeitenden *Garbage-Collector* beeinflusst wird und den tatsächlich benötigten Speicher nicht jederzeit widerspiegeln. In Abbildung (a) sind Auslastungsspitzen beim Archiv Erstellen (Host: kurz nach Start) und Entpacken (Client: zum Ende) zu erkennen. In Abbildung (b) verteilt sich die Auslastung breiter. Detailliertere Zuordnungen wären spekulativ, jedoch sind die Verläufe ausreichend genau um zu zeigen das der Ressourcenverbrauch der Instantvariante das System nicht stärker beansprucht. So übersteigt in beiden Variante die Größe des Heapspeicher nie 200 MB und der Entwicklungsumgebung bleiben selbst bei einem auf maximal 256 MB konfigurierten Heapspeicher noch Kapazitäten offen. Aus mehreren Durchläufen wurden diese beiden als repräsentativ ausgewählt.

Während der Projektübertragung wurden keine Speicherlecks oder vermehrte Aufrufe des *Garbage-Collector* beobachtet. Insgesamt ist keine gesteigerte Auslastung zu erkennen und zum Teil sind leichte Verbesserung festzustellen, wie kürzere Lastspitzen. Während der Tests reagierten die grafischen Oberflächen bei der Übertragung großer Dateien wie gewohnt. Beim Hinzufügen von vielen Dateien zum Projektbaum zeigt Eclipse leichte Verzögerungen im Bereich bis ca. 2 Sekunden in der Darstellung neuer Aktivitäten. Da dieser Effekt die Bedienbarkeit jedoch nicht weiter einschränkt ist die Implementierung praktisch einsetzbar.

Komprimierung

Um den Einfluss des verwendeten *Zstandard*-Kompressors zu prüfen, werden Vergleichstests zur unkomprimierten Übertragung durchgeführt.

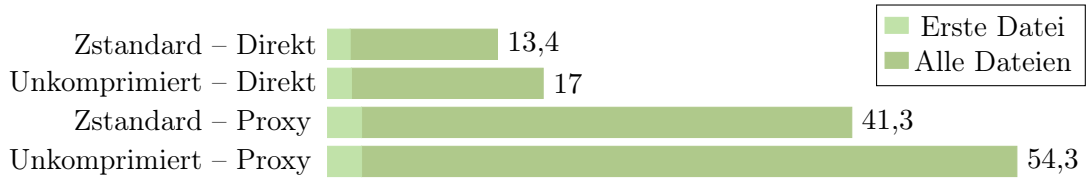


Abbildung 6.6: Ergebnisse des zweiten Testszenario in Sekunden als Median von zehn Ausführungen. Die *Zstandard*-Komprimierung zeigt eine 12 bis 14 prozentige Beschleunigung, die in etwa den Geschwindigkeitsvorteil zur Archivübertragung ausmacht.

Der Vergleich zeigt, wie zu erwarten war, einen deutlichen Vorteil bei Komprimierung. Da die *Zstandard*-Bibliothek noch Optimierungen wie den Einsatz von Wörterbüchern und höheren Komprimierungsstufen zum Verkleinern der Datenmengen erlaubt, sind hier in Zukunft noch weitere Verbesserungen möglich.

7 Ergebnis

7.1 Zusammenfassung und Bewertung der Ergebnisse

Mit dieser Arbeit wurde ein wesentlicher Schritt zur *Verbesserung des Session-Start* erreicht. Dafür wurde ein neuer Lösungsansatz ausgearbeitet, der die Umsetzung der Aufgabenstellung ermöglicht und die Grundlage für weitere Verbesserungen in diesem Bereich bildet.

Um dies zu erlangen wurde zunächst eine Analyse der vorhergehenden Arbeiten durchgeführt und die Hindernisse die ihre Fortführung erschweren dargestellt. Zudem wurden wichtige Einschränkungen bei der Projektübertragung mittels Dateiaktivitäten benannt.

Daraufhin wurden auf dieser Grundlage sowie Anhand der Aufgabenstellung eine Reihe von Anforderungen erarbeitet die eine vollständige Umsetzung erfüllen sollten. Diese waren wiederum der Ausgangspunkt, um eine neue *Stream*-basierte Übertragungsart zu konzipieren und in einem iterativen Vorgehen mit der Entwicklung eines Prototypen zu beginnen. Dabei zeigte der erste Prototyp bereits das Potential dieses Ansatzes und wurde in eine bereits in Saros integrierte Version weiterentwickelt. Im Rahmen der sich abwechselnden Entwurfs- und Implementierungsphasen wurden die unterschiedlichen Zustände die eine Datei im Zeitraum der Aushandlung annehmen kann dargelegt und die Gründe benannt, die das Einschränken des Schreibzugriffs bisher notwendig machten. Daraufhin wurde ein Lösung geschaffen, Aktivitäten abhängig vom Übertragungsstatus zu versenden bzw. in eine Warteschlange aufzunehmen. Dadurch konnte der Schreibzugriff für bereits teilnehmende Personen aufgehoben werden und es wurden die nächsten notwendigen Schritte aufgezeigt. Des Weiteren wurde die Möglichkeit geschaffen, lange Dateiverarbeitungen zu beenden und auch die Archivverarbeitung damit verbessert.

Damit die in meiner Arbeit entstandene Umsetzung mit der bisherigen sowie mit zukünftigen Entwicklungen verglichen werden kann, wurden zunächst konkrete Testszenarien entworfen. Um diese Szenarien unter realistischen Bedingungen durchzuführen entstand eine wiederverwendbare Testumgebung, die Netzwerkeigenschaften simuliert und das eingesetzte Testframework erweitert, um neben der automatischen Testausführung auch die Ausführungszeiten selbstständig zu messen.

Dabei konnte gezeigt werden, dass die erstellte Lösung den NutzerInnen nicht nur einzelne Dateien früher als mit der bisher genutzten Archivübertragung bereitstellen kann, sondern auch die Gesamtgeschwindigkeit dieser übertrifft. Dadurch müssen sich NutzerInnen bei der Wahl der Übertragungsart nicht zwischen einem frühzeitigen Zustellen von Dateien und dem schnelleren Zugriff auf das Gesamtprojekt entscheiden.

7.2 Fazit

Abschließend bleibt zu sagen, dass die Aufgabenstellung dieser Arbeit nicht grundlos bereits mehrfach gestellt wurde und es sinnvoll war lediglich eine Verbesserung des vorhergehenden Zustands anzustreben. Auch wenn vorher bekannt war, dass die Einarbeitung in das Projekt bereits einen hohen Arbeitsaufwand erfordert, so war der Aufwand den Lebenszyklus einiger Abläufe zu verstehen dennoch höher als erwartet. So war das Verhalten durch viele Abstraktionsschichten und ineinander verzahnte Abläufe zunächst nicht immer offensichtlich.

Zwar hätte ich im Rahmen dieser Arbeit gerne auf ein paar Kompromisse verzichtet und bereits größere Umstrukturierungen vorgenommen, jedoch gelang es mir so eine lang gewünschte Funktionalität umzusetzen. Dies bringt damit nicht nur Vorteile für Saros-NutzerInnen, sondern erleichtert auch die Entwicklung in anderen Teilprojekten wie dem *Saros-Server*.

7.3 Ausblick

Diese Arbeit hat eine Grundlage geschaffen, auf der nun weitere Verbesserungen erfolgen werden. Dabei ist die abschließende Arbeit an der Schreibzugriffseinschränkung für den Client durchzuführen, damit auch eingeladene NutzerInnen sofort Dateien bearbeiten können und zunächst rudimentär gehaltene Abläufe zu erweitern. So wurden bewusst einfache Implementierungen für die Priorisierung von zu übertragenen Dateien gewählt, die Komprimierungsmöglichkeiten nicht ausgeschöpft und eine simple Warteschlange für Aktivitäten erstellt. Diese nun zu erweitern wird weitere Vorteile der neuen Übertragungsart ausschöpfen und die Wartezeiten für NutzerInnen weiter verkürzen.

Es hat sich zudem gezeigt dass das bereits mit vielen Aufgabenkommentaren versehene *Negotiation*-Paket, in welchem die meisten Änderungen vollzogen wurden, einer Grundlegenden Änderung bedarf, die sich auch auf das *Session*-Paket auswirken wird. So spiegelt die Klassenstruktur den Einladungsprozess nur bedingt wider und Teile des Aushandlungsprozesses, wie das Festlegen der Übertragungsart, sind Teil des *Session*-Pakets. Im Weiteren erzeugen Vorgänge wie die Wartezeit zwischen der getrennten Annahme der Sitzungseinladung und Annahme eines Projektes Arbeitsunterbrechungen für die eingeladene Person, die durch das Ermitteln aller benötigten Informationen vor Einladungsbeginn vermieden werden könnten.

Somit bietet Saros noch weitere interessante Aufgaben, an denen ich auch gerne weiterhin mitarbeiten werde.

Literaturverzeichnis

[1] Saros Project. Real-Time Distributed Software Development | Saros. https://www.saros-project.org, . Letzter Aufruf: 24.07.2018.

[2] Docker Inc. Docker overview. https://docs.docker.com/engine/docker-overview/, 2018. Letzter Aufruf: 05.08.2018.

[3] Alexey N. Kuznetsov and Bert Hubert. Linux man Page | tbf - Token Bucket Filter . http://man7.org/linux/man-pages/man8/tc-tbf.8.html, 2001. Letzter Aufruf: 05.08.2018.

[4] Stephen Hemminger, Fabio Ludovici, and Hagen Paul Pfeifer. Linux man Page | tbf - Token Bucket Filter . http://man7.org/linux/man-pages/man8/tc-netem.8.html, 2011. Letzter Aufruf: 05.08.2018.

[5] Saros Project. History | Saros. https://www.saros-project.org/history, . Letzter Aufruf: 24.07.2018.

[6] David Damm. Schnellerer Sitzungsstart in Saros. http://www.inf.fu-berlin.de/w/SE/ThesisSchnellererSitzungsstart, 2015. Letzter Aufruf: 26.07.2018.

[7] Daniel Theus. Entwicklung einer Infrastruktur für wechselbare Projektübertragungsformen und Weiterentwicklung Bestehender in Saros. Bachelorarbeit, Freie Universität Berlin, November 2015.

[8] Patrick Fehling. Entwurf, Implementation und Evaluation des Instant Session Starts für das Open Source IDE-Plugin „Saros“. Bachelorarbeit, Hochschule für Technik und Wirtschaft Berlin, August 2016.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. ISBN 0-201-63361-2.

[10] L. Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. RFC 1950, RFC Editor, May 1996. URL http://www.rfc-editor.org/rfc/rfc1950.txt.

[11] Przemyslaw Skibinski. Vergleich unterschiedlicher Komprimierungsbibliotheken und Konfigurationen – lzbench171_sorted.md. https://github.com/inikep/lzbench/blob/5514862d3ec7e0a6738cdd75d6f1c71d60359819/lzbench171_sorted.md, 2017. Letzter Aufruf: 07.08.2018.

Abbildungsverzeichnis

A Eingebrachte Patches

Folgende Patches wurden auf der Plattform GitHub unter der Adresse `https://github.com/saros-project/saros` im Projekt bereitgestellt und sind mit der dazugehörigen *Pull-Request* Nummer bzw. *git-commit-id* auffindbar.

In den Hauptzweig Integriert

#123 [FEATURE] Toggle Instant Session Start

git-commit-id: 58892e1dd5ff01909b76582cb318a00692cbee8c

```
In preparation for the new feature 'Instant Session Start', this patch
introduces a feature toggle, configurable via the user preference store.

While users can already select the feature in Eclipse, via
Preferences -> Saros -> Advanced, it has no effect.

This Patch is a adapted version of parts of the work from
[Patrick Fehling] and [Victor Brekenfeld].

[Patrick Fehling]: https://saros-build.imp.fu-berlin.de/gerrit/#/c/3067/
[Victor Brekenfeld]: https://saros-build.imp.fu-berlin.de/gerrit/#/c
    /3498/
```

#124 [STF] Test Instant Session Start

git-commit-id: e8f61e2c5ea79e454db77ee025defdb2aeeaeb12

```
This patch enables to prefer Instant Session Start in STF tests and
extends 2 Basic Invitation Tests to run them with Instant Session
Start preferred.

The code is partly adapted from [Patrick Fehling].

[Patrick Fehling]: https://saros-build.imp.fu-berlin.de/gerrit/#/c/3081/
```

#125 [FEATURE][CORE] Cancelable Input Streams

git-commit-id: 93d01bea7ae6cce5401c161cc3b40df0cb7c08c8

```
This patch introduces a class, wrapping Input Streams, to check for user
cancellations at each read access and adds this behavior to the
DecompressArchiveTask.
```

```
It shortens the response time, till user cancelations are proceded,
while decompressing of large files and removes an old FIXME.
```

#126 [FEATURE][CORE] Negotiate Transfer Type Instant

git-commit-id: e83349ba0a40cf4e0a9e804d66075e074f021beb

```
In preparation for the new feature 'Instant Session Start', this patch
introduces a new Transfer Type INSTANT and extends the Project
Negotiation Type Hook to negotiate according to the User Preferences.

While users can already select the feature, it has no effect and Project
Negotiation defaults to ARCHIVE.
```

#127 [INTERNAL] Move Transfer Listener Logic

git-commit-id: 47d74bf4f14414705294603fc8348d7819a7b39b

Nach revert neu mit der Nummer #164 und
git-commit-id: 8de344fe94a6ea2a5d68da766e0c24262e840e28

```
In preparation for the new feature 'Instant Session Start', this patch
move parts of the Transfer Listener logic in Archive type to the
Abstract class and prevent later code duplication.
```

#128 [FEATURE][CORE] Introduce Instant Session Start

git-commit-id: d6ee39254fbadd94d71a76daaff347cf9281ac39

Nach revert neu mit der Nummer #183 und
git-commit-id: 33508cca726560730feeffa8d337536b6da5d12e

```
This patch implements the core functionality of Instant Session Start,
which is optional selectable for Project Negotiation. The main scope of
this feature is to decrease the waiting time for users, till the first
files are provided and they can participate in a session.

Deviant to previous approaches, this one uses a file stream based
solution. Compared to using Activities, this provides increased
performance, because of small overheads and better compresion
possibilities, big files support and is for now, not interfering with
existing activity handling code.

This first Version implements a basic prioritisation, by first sending
important eclipse project files, afterwards allways opened files first
and remaining sorted by path hierarchy.

Following features are excluded, to limit this rather big patch or are
planned for the future:

 - Proper handling of Activities, to remove editor lock
```

```
- Compression (using a libary like zstd)

- Configurable File Prioritisations

- More progress infos (bandwith, remaining file size, time...)

- Performance optimizations, refactoring of ProjectNegotiation

- User interface enhancements

- Priority file sending, while active file transmission
```

Zum Review Eingestellt

#154 [FEATURE][CORE] Replaceable Queue Mechanism

```
This patch extracts a Interface out of the project based ActivityQueuer,
used on client-side, which handle activities in SarosSession, during the
project negotiation.
This moves the queuing responsibility into the transfer implementation
and is the preperation for a resource based queuing implementation.
```

#155 [FEATURE][CORE] Rename ActivityQueuer

```
This patch renames ActivityQueuer to ProjectActivityQueuer, to emphasize
its purpose.

extends #154
```

#156 [FEATURE][CORE] Add ResourceActivityQueuer

```
This patch adds a resource based activity queuer and uses it in Instant
Session start.

extends #155
```

#157 [FEATURE][CORE] Move Projects Addition

```
This patch extracts existing logic to addProjectsToSession() and moves
it call, into the transfer implementation.

extends #156
```

#159 [FEATURE][CORE] No Session Lock for existing User

```
This patch allows to only send Jupiter Activities for Files, a User can
already process. This enables to remove the session lock for existing
session participants.
```

```
The invited User stays in read-only mode, till the end of the
negotiation, but can already view activities for received files.

extends #157
```

Erstellte Tickets

Folgende Tickets wurden während des Bearbeitungszeitraum gestellt um bestehende Fehler oder neue Aufgaben zu dokumentieren:

#137 Move Queuing Responsibility to Project Negotiation

#138 Change Queuing to Resource Based Queuing

#139 Project Files should be configurable in Instant Session Start

#149 Refactoring of Project Negotiation

#158 ResourceActivityQueuer should Discard obsolete Editor Events

#176 Saros does not ignore .class files if Eclipse's Automatic Build is switched off

#179 ResourceActivityQueuer should be more sophisticated

B Testergebnisse

Alle Angaben in Sekunden, Archiv = Archivübertragung komplett, Instant = Instantübertragung komplett, Erste = Erste Datei mit Instantübertragung erhalten, Spalten- und Reihenfolge entspricht der Testausführungsfolge, weitere Erklärungen siehe Seite 38.

Testszenario 1

Direktverbindung		Proxyverbindung		IBB	
Archiv	Instant	Archiv	Instant	Archiv	Instant
1,605	1,505	2,669	2,057	6,083	5,554
1,507	1,505	2,003	2,509	5,005	4,505
1,504	1,078	2,007	2,003	5,505	5,008
1,505	1,005	2,504	2,004	5,004	5,507
1,504	1,012	2,506	1,503	5,005	5,005
1,509	1,511	2,009	2,005	5,512	4,507
1,506	1,005	2,504	1,509	5,506	5,010
2,005	1,004	2,009	2,010	5,507	4,509
2,004	1,534	2,011	1,503	5,505	5,004
1,510	2,004	2,503	2,008	5,504	5,011

Testszenario 2

Direktverbindung

Komprimiert			Unkomprimiert	
Erste	Instant	Archiv	Erste	Instant
4,349	15,323	17,221	1,952	16,641
1,862	14,338	16,310	1,950	16,706
1,747	13,349	16,696	1,891	16,071
1,742	13,320	16,712	1,759	16,864
1,782	13,851	17,200	2,307	17,500
1,755	13,477	17,189	1,807	16,960
1,758	13,346	16,706	1,865	16,978
1,735	13,809	16,193	1,872	17,060
1,719	13,315	16,746	1,802	16,977
1,742	13,390	17,187	2,272	17,368

Proxyverbindung

Komprimiert			Unkomprimiert	
Archiv	Erste	Instant	Erste	Instant
46,940	2,680	41,277	4,879	65,350
46,326	2,657	41,877	3,081	54,450
46,234	2,701	40,857	2,940	54,696
46,352	2,679	41,364	2,728	53,948
46,305	2,707	41,381	2,165	53,746
46,338	2,743	40,969	2,225	53,476
50,664	2,730	40,889	2,215	54,340
46,311	2,745	41,449	2,723	54,031
46,318	2,707	40,840	2,703	54,829
45,862	3,244	41,403	2,183	54,344

IBB

Erste	Instant	Archiv
32,121	2336,204	2416,058

Testszenario 3

Direktverbindung			Proxyverbindung		
Archiv	Erste	Instant	Archiv	Erste	Instant
31,440	2,293	19,916	85,906	2,697	71,823
24,877	2,245	20,400	115,438	2,686	143,813
24,792	2,296	20,460	74,018	2,739	71,067
23,413	2,313	21,007	118,746	2,698	70,289
24,835	1,740	19,826	92,849	2,698	71,334
25,836	1,734	20,427	82,388	2,711	78,400
24,325	2,289	21,412	81,396	2,688	71,294
27,264	2,223	20,362	79,133	3,179	121,296
24,294	2,247	21,357	76,928	2,701	121,626
30,981	2,218	22,291	93,387	2,713	67,894

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur unter Verwendung der angegebene Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Datum, Unterschrift