Freie Universität Berlin

Diplomarbeit zu Erlangung des akademischen Grades
Diplom-Informatiker

Stabilitäts- und Testbarkeitsverbesserungen der Netzwerkschicht in Saros

Björn Gustavs

Matrikelnummer: 3303120
uni@gustavs.de

Berlin, 1. August 2011

Fachbereich Mathematik und Informatik
Institut für Informatik
Arbeitsgruppe Software Engineering

Eingereicht bei Prof. Dr. Lutz Prechelt
Betreuer: Dr. Karl Beecher

Selbstständigkeitserklärung:

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Berlin, den 1. August 2011

Björn Gustavs

Zusammenfassung

Diese Arbeit beschreibt Verbesserungen der Stabilität und der Testbarkeit der Netzwerkschicht von Saros, einer Erweiterung für die Entwicklungsoberfläche Eclipse zur kollaborativen Programmierung mit verteilten Teilnehmern.

Hierzu wird zu Beginn ein möglicher Einsatz des Eclipse Communication Frameworks evaluiert. Anschließend werden aktuelle Kommunikationsprobleme zusammengetragen, analysiert und korrigiert sowie die Einführung und Integration von *Universal Plug and Play* zur Verbesserung des Verbindungaufbaues vorgestellt.

Abschließend wird die Testbarkeit der Netzwerkschicht von Saros untersucht, bewertet und Verbesserungspotential aufgezeigt.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Saros 2

1.2 Paarprogrammierung 3

1.3 Netzwerkschicht von Saros 7

2 Das Eclipse Communication Framework 13

2.1 ECF API 13

2.2 Provider 16

2.3 Der XMPP-Provider 17

2.4 Experimenteller Prototyp 20

2.5 Fazit zum ECF 23

3 Probleme in der Netzwerkschicht von Saros 24

3.1 Manuelles Testen 24

3.2 Identifizierte Versagensfälle und Lösungsbeschreibungen 28

4 Integration von Universal Plug and Play 38

4.1 Einführung UPnP 38

4.2 Ablauf von UPnP auf Netzwerkebene 39

4.3 UPnP für Saros 43

4.4 Integration von UPnP in Saros 46

4.5 Fazit 48

5 Testbarkeit der Netzwerkschicht 51

5.1 Einführung Testbarkeit 51

5.2 Testbarkeit im Entwicklungsprozess 54

5.3 Testbarkeit der Netzwerkschicht von Saros 56

5.4 SarosNet für bessere Testbarkeit 62

6 Arbeiten im Saros-Team 63

7 Ausblick 64

8 Fazit 65

9 Literaturverzeichnis 66

10 Anhang 69

10.1 Beispiel einer detaillierten Beschreibung eines UPnP Gerätes 69

10.2 E-Mailverkehr mit Projektleitern von weupnp bzgl. Code Contribution 71

10.3 Umfragebogen (Saros User Feedback Survey) 75

Abbildungsverzeichnis:

Abbildung 1: Eine Saros-Instanz während einer Sitzung DPP. Zu sehen sind drei Teilnehmer in der Sitzung (1), die Kontaktliste (2), der Chat (3), der aktuell betrachtete Quellcode mit Awarenessinformationen zu Änderungen und Markierungen (4) sowie der Sichtbereich anderer Teilnehmer (5) 3

Abbildung 2: Ablauf und Datenfluss der Erstellung eines S5B (nach [XEP65]) 8

Abbildung 3: Darstellung wichtigster Netzwerkklassen und deren Zusammenhänge 10

Abbildung 4: Übersicht der ECF APIs mit unterstützenden Providern 16

Abbildung 5: GUI des Prototyps, Integration in Eclipse mittels Workspace View (1), Icon zum Starten des Tests (2) und Ausgabe des Ereignisprotokolls (3) 20

Abbildung 6: Auswahl des Testaccounts im Prototyp 21

Abbildung 7: Warten auf andere Testclients im Prototyp 21

Abbildung 8: Wireshark protokolliert TCP-Retransmissions bei aktiviertem Paketverlust 26

Abbildung 9: Anteile von Umfragebögen je Rubrik über 61 Teilnehmer (September 2009 – März 2011) 27

Abbildung 10: Smack Debug Fenster zur Paketverfolgung des Senders 30

Abbildung 11: Smack Debug Fenster zur Paketverfolgung des Empfängers 30

Abbildung 12: Warnung bei langsamer Datenverbindung 31

Abbildung 13: neue Warnung bei langsamer Datenverbindung (Blinken angedeutet) 31

Abbildung 14: UPnP Device Stack (nach UPnP Device Architecture v1.1) 38

Abbildung 15: Netzwerkszenario für Clients in LAN mit NAT Gateway 42

Abbildung 16: GUI der Testanwendung von weupnp 45

Abbildung 17: Konfigurationselemente für UPnP 47

Abbildung 18: Bytestream Verbindungstest meldet IBB Verbindung 49

Abbildung 19: Bytestream Verbindungstest meldet S5B-Verbindung 50

Abbildung 20: Verfolgbarkeit der Testbarkeitsanforderungen [Jun06] 55

Abbildung 21: Auswertung von Softwaremetriken mittels "Metrics2" 59

Abbildung 22: STAN weist zyklische Abhängigkeiten zwischen Paketen nach 60

1 Einleitung

Diese Diplomarbeit befasst sich mit der Netzwerkschicht von Saros und stellt zum Verständnis der Betrachtungen in Kapitel 1 sowohl Saros, seine zugrundeliegenden Konzepte der Paarprogrammierung und dessen Variationen sowie den Aufbau der Netzwerkschicht in Saros vor.

Die im Thema dieser Arbeit genannte Stabilität der Netzwerkschicht kann mit deren Verlässlichkeit im Einsatz beschrieben werden. Die Motivation für deren Verbesserung wird im Kapitel 1.3.1 erläutert.

In Kapitel 2 wird der Einsatz eines Kommunikationsframeworks als Option zu einer höheren Stabilität evaluiert. Anschließend werden in Kapitel 3 aktuelle Probleme der Netzwerkschicht von Saros gesammelt, analysiert und korrigiert, in Kapitel 4 die Einführung von *Universal Plug and Play* beschrieben und abschließend in Kapitel 5 die Testbarkeit der Netzwerkschicht behandelt. Die Untersuchung der Testbarkeit beginnt – zum besseren Verständnis dieses Themas – mit einer Einführung in dessen Grundlagen und mit der Motivation dieser Betrachtung in Saros.

Typografische Konvention

Zur besseren Lesbarkeit dieser Diplomarbeit nutze ich folgende typografische Konventionen:

`Maschinenschrift`
für Markierung von Code, Objekten, Methoden und Programmausgaben

Kursivschrift
für Abkürzungen, Fachwörter, nicht sinnvoll übersetzbare Wörter

1.1 Saros

Saros ist ein im Rahmen einer Diplomarbeit 2006 von Riad Djemili [Dje06] entwickeltes Plugin für die Entwicklungsumgebung Eclipse. Es erlaubt zwei räumlich getrennten Beteiligten, Paarprogrammierung zu betreiben, also gleichzeitig und gemeinsam an einem Projekt in Eclipse zu arbeiten. Saros wurde von einer Reihe von Studenten der Freien Universität Berlin im Rahmen von Diplom-, Bachelor- und Masterarbeiten weiterentwickelt und um neue Funktionen erweitert.

Anfangs war es nur einem Sitzungsteilnehmer – dem *Driver* – möglich, Änderungen durchzuführen und ein zweiter Sitzungsteilnehmer – der *Observer* – beobachtete den Driver und korrigierte ihn wenn nötig. Während einer gemeinsamen Sitzung mit Saros werden Aktionen und Änderungen, die von einem Teilnehmer am Projekt durchgeführt werden (Datei öffnen, Sichtbereich ändern, Texteingaben, Dateioperationen, etc.), an alle anderen Teilnehmer der Sitzung übermittelt und dort repliziert. Diese Informationseinheiten werden als Aktivitäten gekapselt. Das Ziel ist, dass sowohl alle Teilnehmer zu jedem Zeitpunkt mit einer identischen Version des Projektes zusammenarbeiten als auch jeder Teilnehmer die aktuellen Aktivitätsbereiche (Awarenessinformationen) aller anderen erkennen kann.

Saros entwickelte sich von der reinen verteilten Paarprogrammierung weiter. In meiner Studienarbeit erweiterte ich Saros unter anderem um die Mehrere-*Observer*-Funktionalität [Gus07] und Oliver Rieger in seiner Diplomarbeit um die Mehrere-*Driver*-Funktionalität [Rie08]. Die ursprüngliche Einschränkung einer reinen Paarprogrammierung wurde somit aufgehoben. Heute können prinzipiell beliebig viele Teilnehmer einer Saros-Sitzung beitreten und auch Änderungen durchführen.

Diese Änderung zwingt die Beteiligten zwar nicht mehr in die positiven Effekte der klassischen Paarprogrammierung (Kapitel 1.2), jedoch kann einem Teilnehmer oder mehreren Teilnehmern durch die Entziehung von Schreib-Rechten, die Rolle des *Observers* auferlegt werden.

In Abbildung 1 wird eine laufende Saros-Sitzung in einer Eclipse-Instanz dargestellt. Neben der, dem Entwickler gewohnten Programmierumgebung, werden Informationen zu anderen Teilnehmern und der Chat in einem Saros-eigenen Anzeigebereich (*View*) ausgegeben. Der Quellcode-Editor enthält zusätzliche Angaben zu Tätigkeiten anderer Sitzungsteilnehmer (Awarenessinformationen). Dadurch werden Änderungen und Markierungen, die andere Teilnehmer durchgeführt haben, mit einer ihnen zugewiesenen Farbe hinterlegt. Auch der Sichtbereich eines anderen Teilnehmers im gleichen angezeigten Dokument wird mittels einer farblichen Markierung am Rande des Editors dargestellt. Weiterhin werden jene Dateien im *Package-Explorer* gesondert markiert, welche von anderen Teilnehmern betrachtet werden.

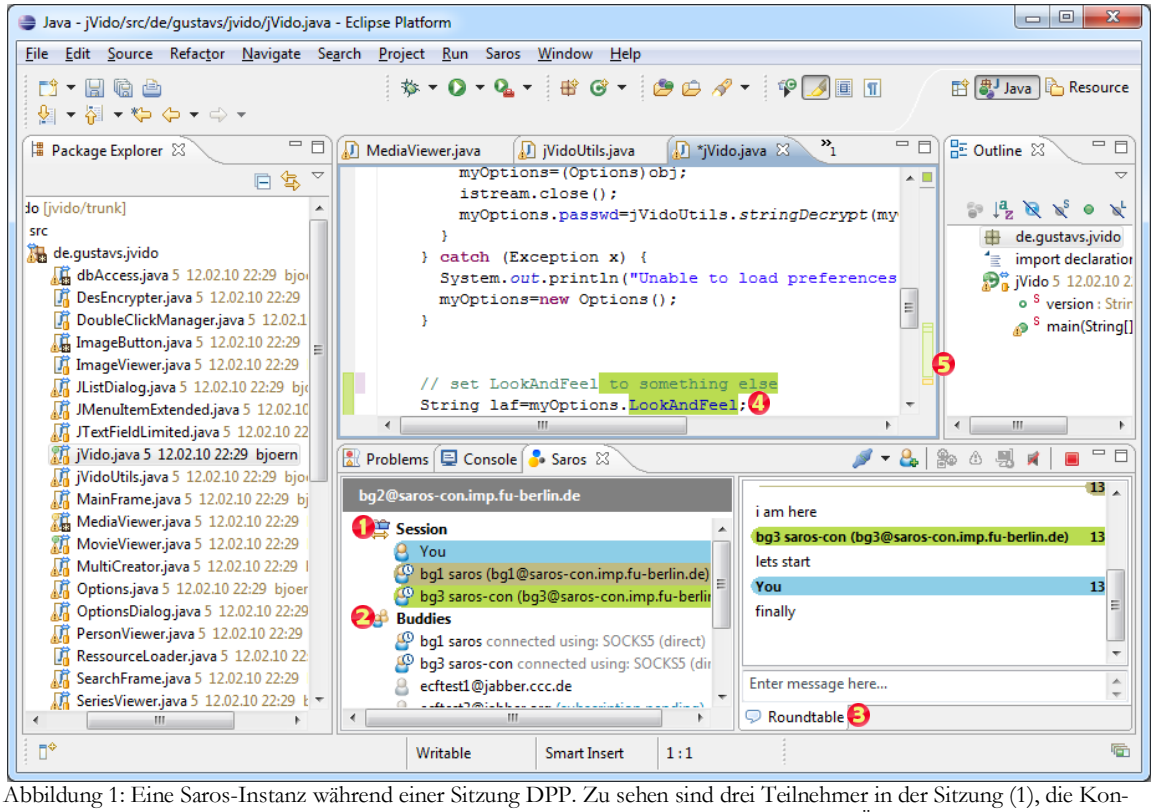

Abbildung 1: Eine Saros-Instanz während einer Sitzung DPP. Zu sehen sind drei Teilnehmer in der Sitzung (1), die Kontaktliste (2), der Chat (3), der aktuell betrachtete Quellcode mit Awarenessinformationen zu Änderungen und Markierungen (4) sowie der Sichtbereich anderer Teilnehmer (5)

1.2 Paarprogrammierung

Unter Paarprogrammierung (englisch „*Pair Programming*“) versteht man eine Arbeitsweise, die insbesondere bei der agilen Softwareentwicklung Verwendung findet. Sie ist eine der primären Praktiken im *Extreme Programming* (XP). [Bec04]

Bei der Paarprogrammierung arbeiten zwei Programmierer gemeinsam und nebeneinander an einem Computer und an einer Aufgabe. Dabei nimmt einer von ihnen die Rolle des *Drivers* ein, welcher den Computer bedient und Code schreibt. Der Zweite übernimmt die Rolle des *Observers* (auch Navigator genannt) und überwacht die Tätigkeiten des *Drivers*. Er führt damit einen kontinuierlichen Design- und Code-Review durch, überdenkt die Herangehensweise und spricht Probleme an. Die Verteilung der Rollen ist nicht fest vergeben, sondern kann und sollte von Zeit zu Zeit wechseln.

Im Dialog und durch Teamarbeit können so schon in der Entwicklungsphase Fehler des Entwicklers bemerkt und korrigiert werden. Dadurch kann schon frühzeitig eine höhere Softwarequalität erreicht werden und nicht erst wenn in einer späteren Entwicklungsphase Tests oder Reviews durchgeführt werden.

Der Nutzen von Paarprogrammierung wurde in einer Reihe von Arbeiten untersucht und bestätigt (z.B. Canfora, Cimitile et al., 2005; Macias 2002; Noll,Atkinson 2003; Tessem 2003) [Bry06].

In einem von Williams, Kessler, Cunningham und Jeffries veröffentlichten Artikel [Wil001] wird ein Experiment einer Software-Engineering-Veranstaltung ausgewertet. An dieser nahmen 41 Studenten teil, von denen 28 in Paaren und 13 individuell arbeiteten. Nach Abschluss der Entwicklungsaufgabe wurden die Programme vorgegebenen Tests unterzogen. Hier liefen signifikant mehr Tests der Paarprogrammierer erfolgreich durch. Im Schnitt lagen diese Tests 15% über denen der Einzelprogrammierer.

	Einzelprogrammierer	Paarprogrammierer
Programm 1	73,4%	86,4%
Programm 2	78,1%	88,6%
Programm 3	70,4%	87,1%
Programm 4	78,1%	94,4%

Tabelle 1: Prozentsätze erfolgreicher Tests (Williams, Kessler, et al.)

Ein weiteres eindrucksvolles Resultat beschreibt [McD03] in der Auswertung einer Untersuchung zur Paarprogrammierung im Rahmen eines Programmierkurses an der *University of California-Santa Cruz* (2000/2001) mit 555 Studenten. Die Untersuchung konzentrierte sich nicht nur auf die Ergebnisse der Paarprogrammierung selber, sondern auch auf Folgeleistungen.

In der Auswertung durch [McD03] wurden folgende Erkenntnisse belegt:

1. mehr Paarprogrammierer als Einzelprogrammierer erreichten die Abschlussprüfung (90,8% : 80,4%)
2. mehr Paarprogrammierer bestanden die Abschlussprüfung (72,3% : 62,8%)
3. die Programmqualität lag bei Paarprogrammierern prozentual höher (86,6% : 68,1%)
4. Einzelprogrammierer schlossen bei der Abschlussprüfung ebenso gut wie Paarprogrammierer ab
5. mehr Paarprogrammierer gaben an, Vertrauen in der eigene Lösung zu haben (89,4% : 71,2%)
6. mehr Paarprogrammierer gaben an, Spaß bei der Lösung der Aufgaben zu haben (5,14 : 4,69 ; Skala: 1=kein Spaß, 7=viel Spaß)
7. mehr Paarprogrammierer haben einen Nachfolgekurs belegt (76,7% : 62,2%)
8. von Studenten mit dem Ziel Informatik zu studieren, haben nach einem Jahr mehr der Paarprogrammierer dies angemeldet („declared a major in CS“) (70,8% : 42,2%)

Diese positiven Auswirkungen der Paarprogrammierung können auf eine Reihe von Gründen zurückgeführt werden. L.A. Williams stellt in [Wil00] folgende positive Effekte vor:

Pair-Pressure

In Paaren wird ungewollt ein positiver Druck auf den Gegenüber ausgeübt. Teilnehmer gaben auch an, den anderen nicht hängen lassen zu wollen, sondern die Aufgabe innerhalb der Paarprogrammierung zu lösen. Der Partner hindert weiterhin daran, sich selbst mit E-Mails oder Surfen abzulenken, wodurch ein konzentrierteres Arbeiten entsteht.

Pair Thinking

Im Paar können bei Problemstellungen mehr Ideen und Alternativen betrachtet und Kombinationslösungen erreicht werden. Die Beteiligten haben unterschiedliche Erfahrungen, Informationsquellen und Sichtweisen auf das Problem, was mehr Lösungswege ermöglicht. (vgl. [Flo91])

Pair Relaying

Bei ausgeprägter Zusammenarbeit im Paar können Probleme gelöst werden, welche jeder Einzelne alleine nicht hätte lösen können. Williams stellt hierzu fest: „together they can evolve solutions to unruly or seemingly impossible problems“ [Wil00].

Pair Reviews

Eine Aufgabe des *Observers* ist es, den aktuell entwickelten Programmcode zu prüfen. Damit wird ein andauernder Codereview realisiert, wodurch Fehler des *Drivers* sofort auffallen können und nicht erst in einer späteren Phase der Entwicklung, in welcher bei komplexen Projekten die Korrektur von Defekten üblicherweise teurer ist. (vgl. [McC98])

Debugging by Explaining

Jemandem ein Problem erklären zu müssen kann helfen, selber eine Lösung zu finden.

> „Another effective technique is to explain your code to someone else. This will often cause you to explain the bug to yourself. Sometimes it takes no more than a few sentences, followed by an embarassed ‘Never mind; I see what's wrong. Sorry to bother you.’ This works remarkably well; you can even use non-programmers as listeners. One university computer center kept a teddy bear near the help desk. Students with mysterious bugs were required to explain them to the teddy bear before they could speak to a human counselor.” [Ker99]

Pair Learning

Durch die Kommunikation und Beobachtung beider Beteiligter der Paarprogrammierung entsteht ein andauernder Wissenstransfer zwischen ihnen. Dies bezieht sich nicht nur auf Programmiertechniken und -kniffe, sondern auch auf den Umgang mit Programmen und Werkzeugen sowie Prozessabläufen. (vgl. [Wil00])

Team Building

Programmierteams, welche Paarprogrammierung durchführten, berichten von verbesserter Teamarbeit währenddessen, als auch darüber hinaus. (vgl. [Wil00])

1.2.1 Distributed Pair/Party Programming

Wird die Einschränkung aufgehoben, dass beide Teilnehmer der Paarprogrammierung am gleichen Rechner sitzen müssen, sondern können sie sich auch an verschiedenen Orten aufhalten, so spricht man von der verteilten Paarprogrammierung (*Distributed Pair Programming*, *DPP*).

Als Vorteile dessen ergeben sich die jeweilige Unabhängigkeit der Beteiligten, wobei jeder seinen ihm angepassten Arbeitsplatz nutzt und es nicht nötig ist, dass sich beide Teilnehmer am gleichen Ort aufhalten. Weiterhin ist es möglich, parallel Aufgaben zu erledigen. Während der *Driver* programmiert, kann der *Observer* beispielsweise online nach benötigten Informationen suchen. Somit wird ein höherer Grad der Parallelität erreicht. (vgl. [Sto03])

Aber es ergeben sich auch neue Herausforderungen, die durch das *DPP*-Werkzeug ausgeglichen werden sollten, um die positiven Effekte der Paarprogrammierung aufrecht zu erhalten. Der direkte Kontakt für die Kommunikation geht verloren und sollte nachgebildet werden. Die wohl wichtigste Kommunikationsform Sprache lässt sich durch Telefonie oder eine *Voice-over-IP* Komponente nachrüsten. Das Zeigen auf einen referenzierten Bildbereich sollte ebenfalls möglich sein („Wo meinst du?“) und ein Whiteboard ist hilfreich, um gezeichnete Veranschaulichungen zu ermöglichen („Hier, ich male dir das mal auf…“). Das Fehlen von Mimik und Körpersprache kann zum Teil durch den Einsatz von Webcams ausgeglichen werden, wenn ausreichend Bandbreite zur Verfügung steht. (vgl. [Sto03])

Dürfen an einer Programmiersitzung mehr als nur zwei Teilnehmer mitwirken, wird dies als *Distributed-Party Programming* – also das Programmieren mit verteilten Beteiligten – bezeichnet.

Beim jetzigen Stand handelt es sich bei Saros – durch die eingangs genannten Funktionserweiterungen – um ein Werkzeug, das auch diese Form der Zusammenarbeit unterstützt.

1.3 Netzwerkschicht von Saros

Für jeglichen Datenaustausch – zum Aufbau einer Sitzung, zum Austausch von Aktivitäten, als auch zur Dateiübertragung – wird das *Extensible Messaging and Presence Protocol* (*XMPP*) unter Verwendung der Bibliothek *Smack* verwendet. *Smack* wird von *Jive Software* entwickelt und veröffentlich in unregelmäßigen Abständen neue Versionen (Version 3.2.1, Stand Juli 2011).

XMPP ging aus dem *Jabber* Protokoll hervor und ist ein 2004 von der *Internet Engineering Task Force* (*IETF*) veröffentlichter Standard (*RFC* 6120, 6121, 6122, 3922, 3923) für den Nachrichtenaustausch und nutzt dazu den XML Standard. Das Protokoll wird durch *XMPP Extension Protocols* (XEPs) erweitert, welche von der *XMPP Standards Foundation* (*XSF*) bereitgestellt werden. Teilnehmer im *XMPP*-Netzwerk werden durch ihre *JID* (*Jabber* ID) identifiziert und adressiert. Die *JID* kann eine Ressource enthalten, welche die adressierte Anwendung bestimmt. Ohne die Angabe der Ressource kann bei gleichzeitiger Verwendung mehrerer *XMPP*-Anwendungen nicht bestimmt werden, welche Anwendung eine eintreffende Nachricht erhält. (vgl. [Szü10])

XMPP beschreibt drei Arten von *XML Stanzas* (Nachrichtentypen) zum Austausch von Informationen:

- *Message Stanza*: einzelne, unidirektionale Nachrichten an Kontakte
- *IQ Stanza*: Anfrage-Antwort Vorgänge
- *Presence Stanza*: Sie übermitteln Netzwerkstatus von *XMPP*-Entitäten (online, offline, Verfügbarkeitsinformation) und dienen beim Abonnieren und Verwalten von Präsenzinformationen anderer Kontakte.

Die zwischen Saros-Instanzen zu übertragenen Informationen (Aktivitäten, Einladung, Awarenessinformationen etc.) werden als *Message Stanza* über das *XMPP*-Netzwerk an die anderen Sitzungsteilnehmer gesendet. Größere Datenmengen, z.B. mehrere gebündelte Aktivitäten oder Dateiübertragungen, werden über einen *Bytestream* versendet. Hierbei handelt es sich um Protokolle, welche als *XMPP*-Erweiterung definiert und in Smack implementiert wurden. Es gibt zwei Arten von *Bytestreams*:

Der ***Socks5 Bytestream*** (*S5B*) ist ein in der *XMPP*-Erweiterung [XEP65] definiertes Protokoll, welches einen bidirektionalen *Bytestream* zwischen zwei *XMPP*-Entitäten beschreibt. *Smack* implementiert diese Protokollerweiterung über *TCP*[1] und erlaubt einen schnellen Kanal für den Austausch von Daten. Die Verbindung kann direkt zwischen den beiden Nutzern (*S5B direct*) oder über einen Vermittler (*S5B mediated*) bestehen. Wegen der Namensähnlichkeit ist darauf hinzuweisen, dass der *Socks5 Bytestream* zwar das *Socks5* Protokoll (RFC 1928) nutzt, aber aufgrund von Anpassungen für das *XMPP*-Netzwerk nicht dazu kompatibel ist.

Zum Aufbau eines direkten *S5B* sendet der *Requester* (Initiator des Verbindungsaufbaues) *IQ Stanzas* an das Ziel (*Target*, Ziel des Verbindungsaufbaues des *Requesters*) mit Informationen

[1] Transmission Control Protocol (TCP), Transportprotokoll, RFC 793, RFC 1323

zum Aushandeln des Verbindungsaufbaues. Teil dessen sind die *IP*-Adressen oder Domain-Namen des *Requesters* (oder des Vermittlers), unter denen er adressierbar ist.

Das *Target* versucht nun, eine *TCP*-Verbindung mit einer der übermittelten Adressen des *Requestors* aufzubauen. Über eine aufgebaute *TCP*-Verbindung wird anschließend der *Bytestream* realisiert. Dieser Ablauf wird in Abbildung 2 dargestellt.

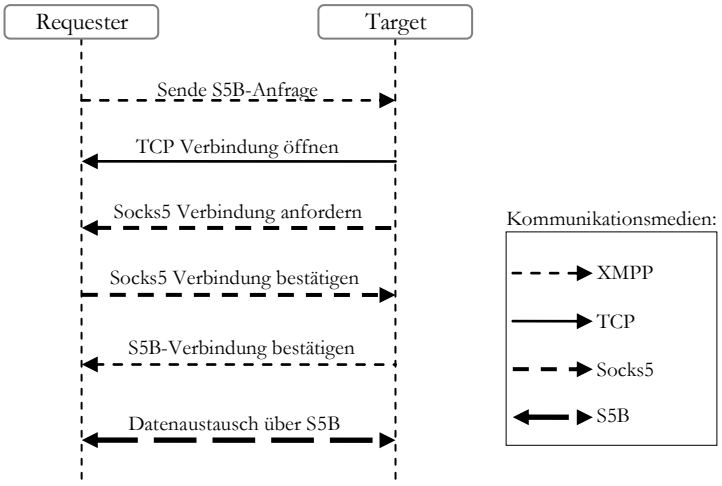

Abbildung 2: Ablauf und Datenfluss der Erstellung eines S5B (nach [XEP65])

Beim *mediated Socks5 Bytestream* kommt ein nicht-lokaler Proxy als Vermittler ins Spiel. *Requester* und *Target* bauen jeweils – ebenfalls wie in Abbildung 2 dargestellt – einen *S5B* zu diesem Vermittler auf, welcher den Datentransport dann jeweils entgegennimmt und weiterleitet.

Saros erweitert den *S5B*-Verbindungsaufbau dahingehend, dass der Empfänger einer Verbindungsanfrage parallel einen zweiten *S5B* zum Sender hin aufbaut (Rollen des *Requesters* und *Targets* werden getauscht). Sollte der Sender keine direkte *S5B*-Verbindung zum Empfänger aufbauen können, kann die zweite Verbindung genutzt werden, sofern diese eine direkte *S5B*-Verbindung ist. Dies hilft in dem Fall, wenn sich der Sender hinter einem NAT-Gateway befindet und der Empfänger keine *TCP* Verbindung zum Sender aufbauen kann, da der Sender nicht direkt erreichbar ist. Sollten durch Restriktionen eines Servers nur unidirektionale Verbindungen aufgebaut worden sein, werden beide Verbindungen von Saros zu einer bidirektionalen Verbindung gebündelt.

Der ***In-Band Bytestream*** (*IBB*) ist ein in der *XMPP*-Erweiterung [XMP47] beschriebenes Protokoll, durch das zwei *XMPP*-Entitäten einen virtuellen *Bytestream* realisieren und Daten übermitteln können. Die Daten werden dazu segmentiert und *Base64*-kodiert in *IQ-Stanzas* über das *XMPP*-Netzwerk versendet. Dadurch ist nur die Kommunikation mit dem *XMPP*-Server nötig und keine ein- oder ausgehende Verbindung zu einem anderen Teilnehmer. Dies hilft in abgeschirmten Netzwerken eine (virtuelle) Datenverbindung zu nutzen. *IBB* ist durch die Zustellung der Nachrichten über XMPP-Server im Vergleich zu *S5B* langsamer und kann laut Standard durch Nachrichtendurchsatzeinschränkungen des Servers noch weiter gedrosselt werden.

Für einen Datenversand versucht Saros zuerst, einen *S5B* aufzubauen und fällt nur dann auf *IBB* zurück, falls dies fehlschlägt.

Den Geschwindigkeitsvorteil von *Socks5 Bytestreams* im Vergleich zu *In-Band Bytestreams* verdeutliche ich im Folgenden durch Geschwindigkeitsmessungen. Dabei wird ein Paket mit Zufallsdaten übertragen. Zur Übertragung wurde die Saros-interne Datenkomprimierung deaktiviert und jeweils der erste Test (dieser beinhaltet den Aufbau des *Bytestreams*) ignoriert, um den reinen Datendurchsatz zu messen.

Übertragungsweg	Durchsatz	Paketgröße
S5B direkt	2873 KiB/s	5MB
IBB über jabber.ccc.de	1,3 KiB/s	512KB
IBB über jabber.org	4 KiB/s	512KB
IBB über saros-con.imp.fu-berlin.de	33,7 KiB/s	512KB

Tabelle 2: Geschwindigkeitsvergleich verschiedener Übertragungswege (jeweils Mittelung über 3 Messungen)

Der Durchsatz über *S5B* ist wesentlich höher als über *IBB*. Aber auch bei *IBB* Übertragungen über verschiedene Server ist ein deutlicher Unterschied zu messen, was durch die Limitierung des Durchsatzes über die *XMPP*-Serversoftware ermöglicht wird.

Die Netzwerkschicht in Saros wird von einer Vielzahl von Klassen realisiert. In der folgenden Abbildung werden die wichtigsten Klassen und ihre Zusammenhänge dargestellt.

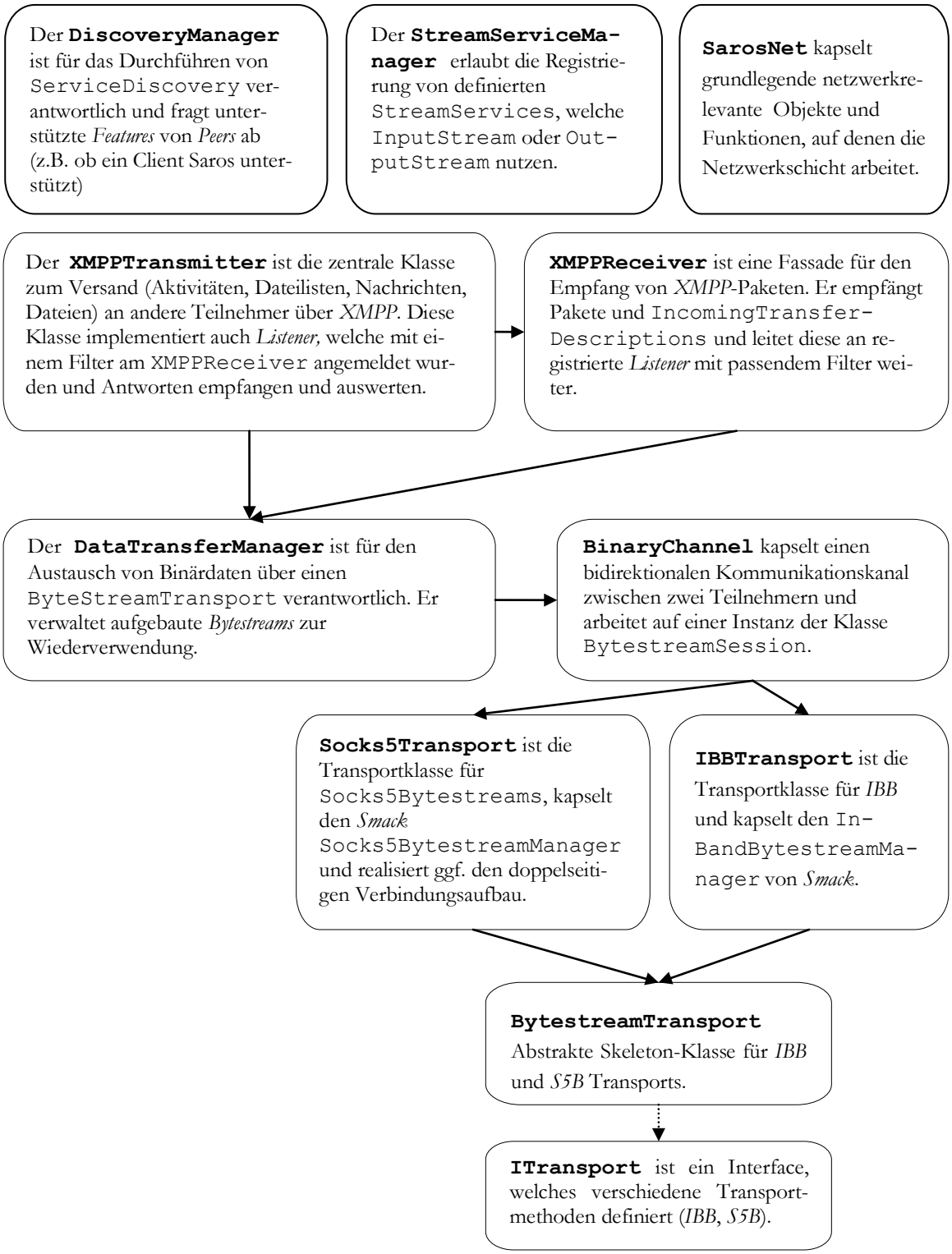

Abbildung 3: Darstellung wichtigster Netzwerkklassen und deren Zusammenhänge

Für Saros-Entwickler ist der `XMPPTransmitter` der Einstiegspunkt zum Versenden von Daten an andere Teilnehmer. Der `XMPPReceiver` dagegen empfängt Daten und reicht diese weiter, wofür sich *Listener* mit einem Filter am `XMPPReceiver` registrieren. Passen eingehende Pakete auf Filter angemeldeter *Listener*, werden die Pakete an diese weitergereicht. Diese Technik folgt dem Entwurfsmuster *Observer*.

Der Versand von größeren Datenblöcken erfolgt nicht mehr durch Nachrichten, sondern durch *Bytestreams*, da Server eine Maximalgröße von Nachrichten definieren. Der `DataTransferManager` organisiert das Erstellen oder die Wiederverwendung von *Bytestreams* und das Senden hierüber.

1.3.1 Motivation für Verbesserungen

Da die Hauptfunktionalität von Saros über die Netzwerkschicht realisiert wird, ist sie eine kritische Komponente. Eine stabile und verlässliche Funktionsweise ist entsprechend grundlegend für die Akzeptanz von Saros beim Anwender. Auch ein kommerzieller Einsatz ist erst dann sinnvoll, wenn Unternehmen auf die Stabilität vertrauen können.

Durch vorherige Diplomarbeiten wurde die Architektur der Netzwerkschicht ([Szü10]) und die verwendete Bibliothek *Smack* bereits verbessert ([Sta10]). Jedoch wird sowohl in Tests bei Endanwendern als auch bei Demonstrationen noch immer auf Verbindungsabbrüche, langsame Verbindungen oder nicht realisierbare Datenverbindungen gestoßen (basierend auf eigenen Erfahrungen, Berichten von Mitarbeitern der Arbeitsgruppe SE und auf Umfragebögen von Endanwendern). Es ist daher nicht der Nachrichtenversand kritisch, sondern die *Bytestream*-Datenverbindung.

Nach der vorhergehenden Einführung in die Struktur der Netzwerkschicht wird anschließend der Einsatz eines Frameworks für die Netzwerkkommunikation evaluiert. Abhängig vom Ergebnis dieser Prüfung wird das weitere Vorgehen entschieden.

1.3.2 Kommunikationsframework als Möglichkeit der Verbesserung

Um eine verlässliche Netzwerkschicht zu erreichen, bestand der Vorschlag, hierfür das Kommunikationsframework von Eclipse zu nutzen. Mit Hilfe des *Eclipse Communication Frameworks* (*ECF*) könnten die Kommunikation und der Datenaustausch zwischen Saros-Instanzen realisiert und die Netzwerkkomponente weitgehend ausgelagert werden. Durch die Wiederverwendung einer externen Lösung können folgende Vorteile entstehen:

- Nutzung einer etablierten Lösung der Netzwerkkommunikation und damit einer Qualitätsverbesserung
- Saros-Entwickler können sich auf ihre Kernkompetenzen konzentrieren, der Verbesserung der *DPP*-Komponenten.

Aber auch mögliche Nachteile dürfen bei der Betrachtung nicht außer Acht gelassen werden:

- Qualität und Eignung des *ECF* sind unbekannt
- geringere Flexibilität, da bei Problemen (beispielsweise bei Versagen innerhalb des *ECF*) auf Lösungen von außerhalb gewartet und vertraut werden muss

Die Eignung des Frameworks für den Einsatz in Saros wird in dem folgenden Kapitel untersucht, indem zum einen der Aufbau des *ECF* analysiert sowie zum anderen dessen Funktionsweise und Integration in einem Prototyp getestet werden. Darauf aufbauend werden Vorteile, Nutzen und Nachteile des Frameworks für Saros diskutiert.

2 Das Eclipse Communication Framework

Das *Eclipse Communication Framework (ECF)* ist ein Framework, um verteilte Server, Anwendungen und Tools zu entwickeln. Es bietet eine modulare Implementierung des *OSGi* 4.2 *Remote Services* Standards mit Unterstützung für *REST*- und *SOAP*-basierte *Remote Services* und asynchronen Nachrichtenaustausch für *Remote Services*. [ECFHP]

Das *ECF* stellt eine Reihe von *APIs* für Kommunikations- und Messaging-Komponenten und zur Interprozesskommunikation für Eclipse-Anwendungen zur Verfügung. Dabei abstrahiert es von konkreten Kommunikationsprotokollen. Diese werden als sogenannte Provider gekapselt zur Verfügung gestellt und können nach Bedarf genutzt werden. (vgl. [JAX09])

2.1 ECF API

Im Folgenden stelle ich (basierend auf [ECFAD]) die *APIs* des *ECF* vor und erläutere kurz ob und inwiefern sie sich für Saros eignen.

ECF Core API

Die *Core API* bietet grundlegende Funktionalitäten und Strukturen, welche von den anderen *APIs* genutzt werden. Sie wäre somit auch für Saros bedeutend.

Zur *Core API* zählen:

- *Distributed Container* (Interface)
- *Distributed Identity Services*
- Factory-basierte Instanziierung von Containern

Die Kernkomponente der *Core API* ist das Interface `IContainer` als „*distributed container*". Alle *ECF Provider* müssen eine Klasse enthalten, die dieses Interface implementiert.

Auf alle folgenden *APIs* wird dann als Adapter zugegriffen, was die modulare Nutzung des *ECF* ermöglicht.

```
IContainer.getAdapter(<interface class>)
```

Datashare API

Die *Datashare API* bietet die Unterstützung von gesicherten, asynchronen Nachrichtenkanälen. Bei den Kanälen kann es sich um Punkt-zu-Punkt Verbindungen oder *Alert*-Dienste (*publish and subscribe*) handeln über welche beliebige Daten (z.B. binärer Datenstrom) gesendet werden können.

Die *Datashare API* bietet einen komfortablen Weg des Datenaustausches. Jedoch ist für die Übertragung von Dateien die *File Transfer API* geeigneter bzw. die *Shared Object API* für das Verteilen von Objekten wie den Aktivitäten von Saros (siehe jeweilige Erklärung der *API*).

Das Nutzen der *Datashare API* würde für beide Bereiche (Dateiübertragung und Aktivitäten-Verteilung) das Implementieren zusätzlicher Steuerungs- und Verwaltungslogik erfordern, was die beiden genannten *APIs* bereits von sich aus bieten. Daher ist die *Datashare API* nicht von Interesse für eine Integration in Saros.

Discovery API

Diese *API* bietet eine Unterstützung für das asynchrone Aufspüren von *Remote Services* in einem Netzwerk.

Saros implementiert all seine notwendige Logik. Weitere externe Services werden nicht benutzt. Das Aufspüren von *Remote Services* ist für Saros somit im dieser Betrachtung nicht relevant.

File Transfer API

Die File *Transfer API* bietet die Unterstützung für das Senden und Empfangen von Dateien.

Saros nutzt *Bytestreams* zum Übertragen von Dateien bzw. binären Daten. Die *File-Transfer API* ist dahingehend zu untersuchen, ob sie hierfür Ersatz bieten kann und somit interessant für die Integration in Saros ist.

Presence API

Die *Presence API* ist der Kern für die Unterstützung von Kontaktlisten (Anzeige und Management), Nachrichtenaustausch und Chaträumen.

Alle Features der *Presence API* sind für Saros relevant, da diese Funktionen die Grundlage der Kommunikation untereinander und der Sitzungsorganisation darstellt.

Remote Services API

Diese *API* bietet die Unterstützung für den Zugriff auf *Remote Services*.

Wie bereits für die *Discovery API* argumentiert, sind externe Dienste und somit die *Remote Services API* für Saros nicht relevant.

Shared Object API

Diese *API* bietet eine protokollunabhängige Unterstützung für das Replizieren von Java-Objekten auf Rechnern innerhalb einer verteilten Gruppe an.

Ein `Shared Object` eignet sich sehr gut dazu, zu übertragende Aktivitäten zu verteilen. Dazu werden Aktivitäten als `Shared Objects` implementiert, welche vom *ECF* dann an alle Teilnehmer einer Sitzung verteilt werden.

Sync API

Diese *API* bietet einen Synchronisationsservice für Anwendungen, welche die Synchronisierung bei Änderung von Dokument realisiert.

Diese *API* klingt zwar interessant für die Synchronisierung bei Änderungen von Projektdateien, ist jedoch für die Netzwerkschicht und damit den Blickwinkel dieser Arbeit nicht von Belang.

Telephony Call API

Diese *API* unterstützt das Starten, Empfangen und Verwalten von *VOIP* Anrufen.

Saros bietet bereits die Möglichkeit, mit Hilfe einer laufenden *Skype*-Instanz von sich heraus Anrufe zu initiieren. Hier wären Vor- und Nachteile einer Nutzung der *ECF-Skype*-Unterstützung abzuwägen. Ein gravierender Nachteil ist die derzeitige Einschränkung auf Win32-Plattformen für den Provider dieser *API*.

2.2 Provider

Wie bereits erwähnt, implementieren die *ECF APIs* keine konkreten Netzwerkprotokolle, sondern sehen Provider zur Umsetzung der Kommunikation auf Netzwerkebene vor. Verschiedene Provider unterstützen unterschiedliche Teilmengen der *ECF APIs*. Abbildung 4 veranschaulicht, welche Provider für welche APIs zur Verfügung stehen.

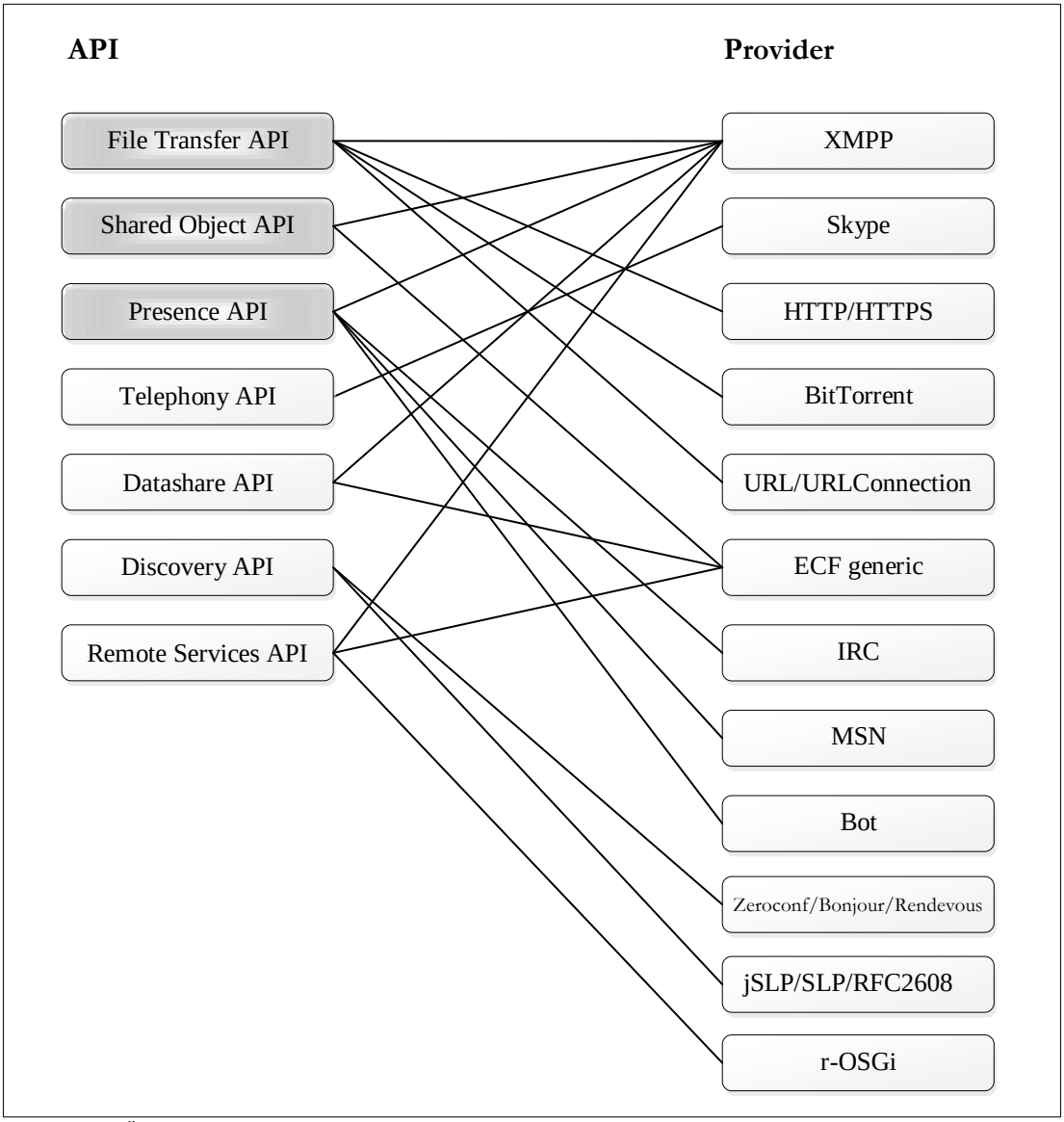

Abbildung 4: Übersicht der ECF APIs mit unterstützenden Providern

In Abbildung 4 wird ersichtlich, dass der Provider *XMPP* bereits alle für Saros interessanten *APIs* (*File Transfer, Shared Object, Presence*) unterstützt und dieser so für eine Nutzung mit *ECF* in Betracht kommen würde.

Daher werde ich im Folgenden die Funktionsweise des *XMPP*-Providers untersuchen. Da dieser die untere Netzwerkschicht des *ECF* darstellt, ist er für die Abschätzung relevant, ob sein Einsatz (über *ECF)* eine Verbesserung gegenüber der gegenwärtigen Nutzung der *Smack* Bibliothek darstellt.

2.3 Der XMPP-Provider

Wie zuvor gezeigt, unterstützt der *XMPP*-Provider die für Saros interessanten *APIs Presence, File Transfer* und *Shared Object*.

Für das Verständnis, wie dieser Provider auf der Netzwerkebene arbeitet, werde ich im Folgenden die Implementierung einiger zentraler Netzwerkfunktionen nachvollziehen. Dies soll anschließend bei der Beurteilung helfen, ob der Einsatz von *ECF* in Verbindung mit dem *XMPP*-Provider Vorteile zur aktuellen Netzwerkschicht von Saros bringen kann.

Als wichtige Eigenschaft des *XMPP*-Providers ist festzustellen, dass dieser – wie die aktuelle Netzwerkschicht von Saros – auf der Bibliothek *Smack* basiert.

2.3.1 Verbindungsaufbau und -abbau

Das Verbindungsmanagement inklusive Verbindungsaufbau und -abbau wird für den *XMPP*-Provider von der Klasse `ECFConnection` übernommen:

```
public class ECFConnection implements ISynchAsynchConnection
```

Die Klasse `ECFConnection` arbeitet hierfür direkt auf einer Instanz der Klasse `XMPPConnection`, welche von *Smack* bereitgestellt wird.

Zum Verbindungsaufbau wird ein Objekt der *Smack*-Klasse `ConnectionConfiguration` konfiguriert (Accountname, Serveradresse, Serverport) und der `connect`-Methode der `XMPPConnection`-Klasse übergeben, welche den Verbindungsaufbau durchführt.

Das Verbindungsmanagement des *XMPP-Providers* arbeitet auf *Smack* und setzt die Funktionalität ohne weiteres Zutun um.

Die folgende Tabelle stellt den Kern der Verbindungslogik der Klasse `ECFConnection` einerseits und von Saros andererseits gegenüber.

ECFConnection	**Saros**

`ConnectionConfiguration config;`
`config = new ConnectionConfiguration(serviceName, serverPort);`
 	
`connection = new XMPPConnection(config);`
 	
`connection.connect();`
 	
`connection.addPacketListener(packetListener, null);`
`connection.addConnectionListener(connectionListener);`
 	
`connection.login(username, (String) data, serverResource);`
	`ConnectionConfiguration connectionConfiguration = getConnectionConfiguration();`
	
	`connection = new XMPPConnection(connectionConfiguration);`
	
	`connection.connect();`
	
	`connection.addConnectionListener(smackConnectionListener);`
	
	`connection.login(username, password, Saros.RESOURCE);`

Tabelle 3: Code-Vergleich des Verbindungsaufbaus (ECF vs. Saros)

Fazit

Der *XMPP*-Provider führt das Verbindungsmanagement auf derselben technischen Ebene wie Saros durch – mittels *Smack* und ohne zusätzliche Funktionalität.

2.3.2 Nachrichtenversand

Der *XMPP*-Provider implementiert die *Presence API* in der Klasse `XMPPContainer`.

```
public class XMPPContainer extends ClientSOContainer implements
   IPresenceService
```

Ein an den `XMPPContainer` gerichteter Aufruf zum Versand einer Nachricht wird über die Klasse `ECFConnection` geleitet, welche wiederum mit der `Chat`-Klasse von *Smack* arbeitet.

Die folgende Tabelle stellt den Kern des Nachrichtenversandes der Klasse `ECFConnection` einerseits und von Saros andererseits gegenüber.

ECFConnection	**Saros**

`Chat localChat = connection.getChatManager().`
`  createChat(receiverName, new MessageListen-`
`er()…);`
 	
`localChat.sendMessage(aMsg);`
	`Chat chat = getChat(jid);`
	`chat.sendMessage(message);`

Tabelle 4: Code-Vergleich des Nachrichtenversendens (ECF vs. Saros)

Fazit

Wieder führt der *XMPP*-Provider die Netzwerkfunktionalität direkt über *Smack* und ohne weitere Funktionalität aus.

2.3.3 Account Management

Die vom *XMPP*-Provider bereitgestellte Klasse `XMPPContainerAccountManager` implementiert aus der *Presence API* das Interface `IAccountManager`. Die gesamte Klasse stützt sich dabei direkt auf die Smack Klasse `AccountManager` und führt alle Funktionalitäten direkt über *Smack* aus.

```
public boolean createAccount(String username, String password, Map attributes)
throws ECFException {
   try {
       AccountManager.createAccount(username, password, attributes);
   } catch (XMPPException e) {
      traceAndThrow("server exception creating account for " +
      username, e);
   }
   return true;

}
```

Listing 1: ECF Code für Erstellung eines XMPP-Accounts

2.3.4 File Transfer

Der *XMPP*-Provider bietet den Austausch von Dateien über die von *SmackX* (*Smack extensions API*) implementierte Klasse `FileTransfer` an. Dieser führt den Versand von Dateien zwischen zwei Teilnehmern transparent über einen *S5B* oder *IBB* durch. Der Empfänger wählt in der Aushandlungsphase den Transporttyp (nach *XEP96*) aus.

Die zum *XPPP*-Provider zugehörige Klasse `XMPPOutgoingFileTransferHelper` implementiert das Interface des `ISendFileTransferContainerAdapter`.

Der `XMPPOutgoingFileTransferHelper` ruft die Funktion `XMPPOutgoingFileTransfer.startSend` auf, welche wiederum `OutgoingFileTransfer.sendStream` aufruft. `OutgoingFileTransfer` wird hierbei von *SmackX* implementiert, womit ersichtlich ist, dass die Funktionalität des Dateiversandes vom *ECF* über den *XMPP*-Provider an *SmackX* durchgereicht und dort vom Paket `org.jivesoftware.smackx.filetransfer` implementiert wird.

Fazit

Der Dateiversand über den *XMPP*-Provider wird vollständig durch das `FileTransfer` Paket von *Smack* bereitgestellt. Der *XMPP*-Provider kapselt die Funktionalität des Dateiversandes für die `FileTransfer` *API* des *ECF* und reicht diese an *Smack* weiter.

2.3.5 Fazit aus Provideranalyse

Sowohl die vorgestellten Implementierungen von ausgewählten Kernfunktionalitäten des *XMPP*-Providers als auch weitere von mir bei der Analyse untersuchten Klassen zeigen, dass der *XMPP*-Provider des *ECFs* die bereitgestellten Funktionalitäten ohne weitere Abstraktion oder Sicherungsschicht an die Bibliothek *Smack* weiterreicht. Genau dies ist auch Aufgabe und Intention des *XMPP*-Providers und stellt keine Kritik am Provider dar. Es zeigt jedoch einen mangelnden Mehrwert bei einer Nutzung des *ECFs* gegenüber der direkten Nutzung von *Smack*, wie es Saros gegenwärtig praktiziert.

ECF, bzw. der *XMPP*-Provider, stellt sich als Fassade dar. Eine Fassade ist ein Entwurfsmuster der Softwareentwicklung, welche eine zentrale Schnittstelle für eine zugrundeliegende Menge von Schnittstellen anbietet. In diesem Falle bietet der *XMPP*-Provider eine Schnittstelle für verschiedene Funktionalitäten der *Smack* Bibliothek.

Es bleibt noch anzumerken, dass der *XMPP*-Provider sowohl die *Shared Object API* als auch die *Datashare API* über den Nachrichtenversand implementiert und aller Datenaustausch mit diesen APIs über das *XMPP*-Netzwerk läuft.

> „This uses the ECF shared object API […] The current XMPP-Provider implements this by using the XMPP chatroom. […] Note also that for the XMPP-Provider (xmpp chatrooms) the shared object API is used to *implement* the datashare API […]" [Lew08]

Hierbei muss bedacht werden, dass der Versand von Nachrichten über *XMPP*-Server indirekt und somit verhältnismäßig langsam ist. Die Option, Daten über eine Direktverbindung

an einen Teilnehmer zu senden ist im *XMPP*-Provider nicht vorgesehen und müsste wieder direkt über *Smack* geschehen, wie es Saros bereits tut (*Bytestreams*).

2.4 Experimenteller Prototyp

Prototypen dienen in der Softwareentwicklung zur frühzeitigen Erkennung von Problemen im Entwicklungsprozess, indem schon früh kritische Komponenten des Systems implementiert werden, um Anforderungen zu klären (exploratives Prototyping), die technische Realisierbarkeit zu prüfen (experimentelles Prototyping) oder bereits die Kernfunktionalität des Systems zu realisieren (evolutionäres Prototyping).

Ich stelle im Folgenden einen experimentellen Prototyp vor, welchen ich realisiert habe, um den Einsatz des *ECFs* in einem Eclipse-Plugin zu evaluieren und Details zur Integration von *ECF* sowie dessen interne Funktionsweise praktisch nachzuvollziehen.

2.4.1 Konzeption

Der Prototyp soll als Eclipse-Plugin umgesetzt werden, um im selben Kontext wie Saros zu laufen. Er soll über das *ECF* mit dem *XMPP*-Provider kommunizieren und die *Presence API* nutzen, um Testclients untereinander mit den in dieser *API* angebotenen Mechanismen kommunizieren zu lassen. Die Testclients sollen den Tester über Erfolg oder Misserfolg informieren und nur wenig Benutzerinteraktion verlangen.

2.4.2 Implementierung

Der Prototyp „ECFTester“ wurde als Eclipse-Plugin implementiert. Ein *Workbench View* wird bereitgestellt, welcher in Eclipse ein Ereignisprotokoll sowie eine Icon-Schaltfläche zum Starten des Kommunikationstests bereitstellt.

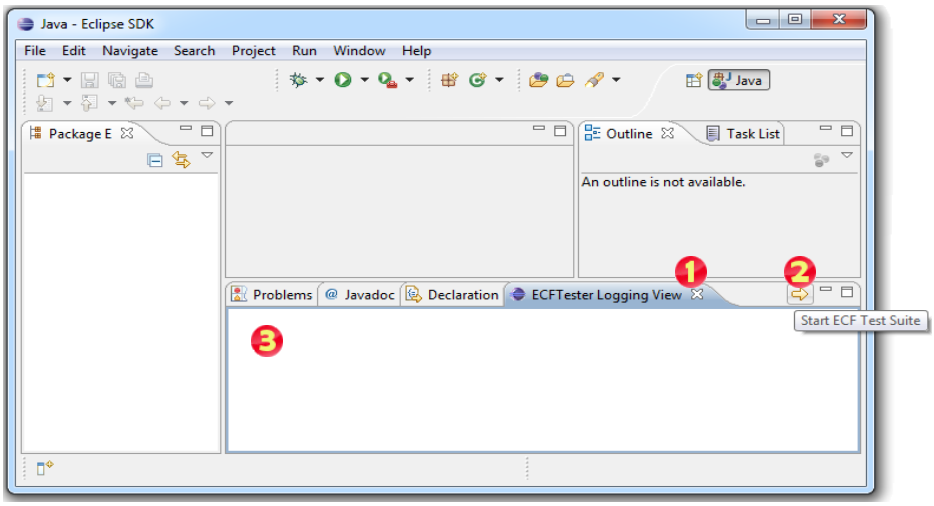

Abbildung 5: GUI des Prototyps, Integration in Eclipse mittels Workspace View (1), Icon zum Starten des Tests (2) und Ausgabe des Ereignisprotokolls (3)

Initialisierung

Der Prototyp ist mit vier vorkonfigurierten *XMPP*-Accounts ausgestattet. Vor dem eigentlichen Testdurchlauf wird ein Assistent zur Konfiguration aufgerufen.

Im ersten Schritt steht die Auswahl eines Testaccounts, mittels welchem sich der Testclient mit dem *XMPP*-Netzwerk verbindet (Abbildung 6). Der Verbindungsaufbau geschieht über das *ECF*.

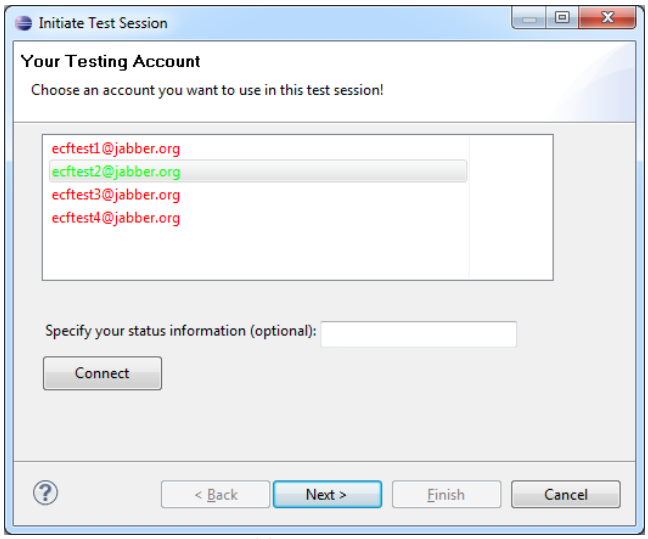

Abbildung 6: Auswahl des Testaccounts im Prototyp

Im zweiten Schritt wird auf weitere Testclients gewartet (Abbildung 7). Dabei werden Präsenzinformationen der Testaccounts in der Kontaktliste ausgewertet, welche von der *Presence API* des *ECF* zur Verfügung gestellt werden.

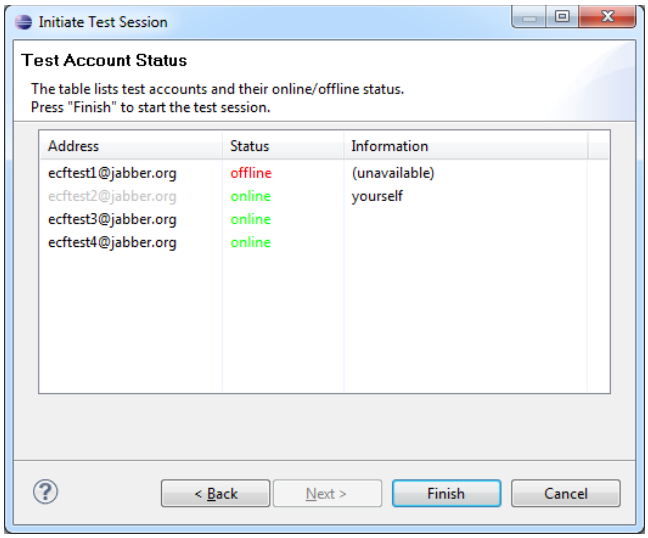

Abbildung 7: Warten auf andere Testclients im Prototyp

Wenn alle Testclients die Initialisierung durch das Abschließen des Assistenten bestätigt haben, wird die Testsuite gestartet.

Bei der Implementierung der *Presence*-Anzeige (Online/Offline Status) fiel wiederum auf, dass die Nutzung des *Rosters* (Kontaktliste) über das *ECF* einerseits und direkt über *Smack* ande-

rerseits, keine nennenswerten Unterschiede beinhaltet. Lediglich verschiedene Klassen- und Methodennamen durch die unterschiedlichen Kapselungen der Funktionalität fallen auf.

Als Beleg folgt ein Vergleich von zwei Codeabschitten, welche die Registrierung eines `RosterListeners` und das Abfragen eines *Presence-Status* der verschiedenen *APIs* gegenüberstellen.

```
container.getRosterManager().addRosterListener(new IRosterListener() {
        @Override
        public void handleRosterUpdate(IRoster roster,
          IRosterItem changedValue) {
                IRosterEntry rosterEntry = (IRosterEntry) changedValue;
                final IPresence presence = rosterEntry.getPresence();
                if (presence.getType().equals(IPresence.Type.UNAVAILABLE))
                ...
        }
    ...
}
```

Listing 2: zusammengefasster Code von ECF für die Registrierung und Benutzung eines RosterListeners

```
public class RosterAdapter implements IRosterListener;
public interface IRosterListener extends RosterListener;

protected IRosterListener rosterListenerBuddys = new RosterAdapter() {
        @Override
        public void presenceChanged(Presence presence) {
            if (presence.isAvailable()) {
                ...
            }
 }
 ...
}
```

Listing 3: zusammengefasster Code von Saros für die Registrierung und Benutzung eines RosterListeners

Synchronisierung

Anschließend durchlaufen alle verbundenen Testclients eine Synchronisierung, bei welcher ein Client als Master festgelegt wird (anhand der niedrigsten Account ID). Dieser generiert eine zufällige TestID und meldet diese allen anderen Clients. Diese bestätigen diese TestID und formen dadurch mit dem Master die Testgruppe.

Diese Synchronisierung ist nötig, da Nachrichten für Kontakte die offline sind auf dem Server zwischengespeichert und bei deren nächster Anmeldung zugestellt werden. Dadurch können Nachrichten einer vorherigen, abgebrochenen Sitzung eintreffen. Durch die Nutzung einer TestID werden solche ungültigen Nachrichten erkannt und aussortiert.

Kommunikationstest

Der Kommunikationstest besteht im Austausch einer Vielzahl von Nachrichten zwischen jedem Paar teilnehmender Testclients. Wenn sowohl das Versenden, als auch das Empfangen jeder Nachricht erfolgreich war und die empfangenen Nachrichten den korrekten Inhalt haben, wird der Test als erfolgreich gewertet und in der Logausgabe vermerkt.

Der Test dient nicht primär dazu, die Qualität des *ECFs* zu testen, sondern mit der Integration des *ECF* in ein eigenes Projekt vertraut zu werden und die Funktionsweise des *XMPP*-Providers nachzuvollziehen.

2.4.3 Fazit aus Prototyp

Auch bei der praktischen Auseinandersetzung mit dem *ECF* durch die Implementierung eines Prototyps wurde deutlich, dass im Vergleich zum direkten Einsatz von *Smack* in Saros, keine technischen Vorteile erreicht werden. Der Implementierungsaufwand für Netzwerkfunktionalitäten sinkt nicht und vor allem liegt dem *ECF* mit dem *XMPP*-Provider dieselbe Technik auf der Netzwerkebene zugrunde – die Bibliothek *Smack*.

Es bleibt anzumerken, dass die Implementierungen von Kontaktliste und Nachrichtenversand mittels *ECF* korrekt funktionieren und auch beim Versand von vielen Nachrichten in kurzer Zeit keine Probleme entstanden.

2.5 Fazit zum ECF

Sowohl bei der Analyse des Aufbaues von *ECF* und des *XMPP*-Providers, als auch während der Implementierung eines Prototyps wurde deutlich, dass der *XMPP*-Provider in seiner Netzwerkkommunikation lediglich auf die von *Smack* implementierte Funktionalität zurückgreift, ohne relevante Zusatz-, Komfort- oder Absicherungsfunktionalitäten hinzuzufügen.

Das Kontaktlistenmanagement und der Nachrichtenaustausch über den *XMPP*-Provider funktionieren wie gezeigt relativ einfach und gut, basieren jedoch sehr eng auf *Smack*.

Die Kontaktliste oder das Versenden von Nachrichten sind bei Saros nicht der sensible Bereich, sondern die *Peer-to-Peer* Datenverbindungen. Das *ECF* bieten über den *XMPP*-Provider keine Möglichkeiten, Daten als *Bytestream* zu versenden. Eine effizientere Art, als über Nachrichten binäre Daten zu senden, steht dem *ECF* mit der *File Transfer API* zur Verfügung, welche die `FileTransfer` Klasse von *Smack* nutzt. Für jeden Transfer wird der *Bytestream* jedoch neu ausgehandelt und aufgebaut. Zwar bietet der *XMPP*-Provider einen Zugriff auf das zugrundeliegende `Connection`-Objekt von *Smack*, über welches direkt mit *Smack* und *Bytestreams* gearbeitet werden könnte, jedoch braucht man dafür nicht das *ECF* einzuführen.

Saros auf *ECF* umzustellen würde (für die Netzwerkkomponente) lediglich die Netzwerkfunktionalität umschichten und doch wieder im selben Maße auf *Smack* basieren wie bisher. Die Nutzung des *ECF* würde eine weitere Zugriffsschicht in Saros erzeugen, während portierte Funktionalitäten (beispielsweise das Accountmanagement) die gleiche Stabilität – weil gleichen Netzwerkcode – aufweisen. Defekte in Smack würden genauso für Implementierungen über *ECF* als auch direkt über Smack gelten.

Daher habe ich in Abstimmung mit meinem Betreuer und mit Kenntnisnahme weiterer Mitarbeiter der Arbeitsgruppe Softwareengineering von der Integration des *ECF* abgesehen und mich im Folgenden direkt mit Kommunikationsproblemen von Saros und *Smack* auseinandergesetzt.

Im folgenden Kapitel werden bestehende Probleme ermittelt, zusammengetragen, vorgestellt und deren Bearbeitung dokumentiert.

3 Probleme in der Netzwerkschicht von Saros

Verschiedene Quellen weisen Probleme in der Netzwerkkommunikation zwischen Teilnehmern einer Sitzung in Saros nach. Im Folgenden werde ich Quellen von Problemberichten aufzeigen und Versagensfälle identifizieren.

Abgrenzung von Begriffen (nach [Pre99])

> Bei einem **Versagen** handelt es sich um ein Verhalten einer Software, welches von der Spezifikation abweicht. Dem liegt ein **Defekt** (bzw. Mangel) im Quellcode der Software zugrunde. Ein Defekt entsteht durch einen **Fehler** im Handeln oder Denken des Programmierers.

Ich habe die folgenden Quellen für existierendes Versagen in Saros herangezogen, um Versagen zu identifizieren und anschließend deren zugrundeliegenden Defekte zu suchen und zu korrigieren:

- manuelles Testen
- Anwenderumfragen
- Fehlerberichte anderer Entwickler (*Bug Reports*)

3.1 Manuelles Testen

Manuelles Testen ist ein dynamisches Verfahren, bei welchem die Software ausgeführt und dabei deren Verhalten auf Übereinstimmung mit Spezifikation und Erwartung geprüft wird.

Es muss bedacht werden, dass durch das Testen zwar Versagen gefunden, aber nicht die Korrektheit einer Software bewiesen werden kann, denn diese vermag nicht getesteten oder auch manuell nicht testbaren Code enthalten, welcher unbekannte Defekte aufweisen kann.

Ich führte zwischen mehreren Saros-Instanzen Tests durch, um nachweisliches Versagen aufzudecken. Bei diesen manuellen Tests handelte es sich um erfahrungsgetriebenes, codebasiertes und realitätsnahes Testen.

3.1.1 Testnetz, Aufbau, Motivation

Vor der Durchführung von Tests stand die Überlegung, wie eine große Abdeckung von Einsatzgebieten von Saros erreicht werden kann.

Das Ausführen von mehreren Saros-Instanzen auf einem oder zwei Rechnern kann nur wenige Szenarien erfassen. Berichte vorheriger Saros-Mitarbeiter nannten Kommunikationsprobleme zwischen Instanzen auf verschiedenen Betriebssystemen. Daher entschloss ich mich dazu, ein Netzwerk von virtuellen Rechnern mit verschiedenen Betriebssystemen zu installieren, welche über ein konfigurierbares Netzwerk kommunizieren.

Somit erstellte und konfigurierte ich verschiedene Betriebssysteme auf Basis der Virtualisierungssoftware *VMWare*:

1. Linux: Debian (Wheezy), als Gateway mit zwei virtuellen Netzwerkkarten
2. Linux: Ubuntu 10.10
3. Linux: Ubuntu 11.4
4. Linux: Suse 11.4
5. Windows XP SP3

Für Testsitzungen kam zusätzlich ein Windows7- und MacOS 10.6-System zum Einsatz.

Ein virtuelles Testsetup mit 3 virtuellen Maschinen unter Linux wurde bereits im Rahmen der Diplomarbeit von Sandor Szücs [Szü10] erstellt. Ich entschied mich jedoch, dies nicht zu nutzen, da ich auch Effekte beim Einsatz von Saros auf bisher unvorbereiteten Systemen erkunden wollte, was sich als wichtig erwiesen hat (siehe Kapitel 3.2.5). Andererseits ziele ich auf mehr Systeme und unterschiedliche Betriebssysteme ab.

Das Debian-System (1) wurde als Gateway eingerichtet. Netzwerkparameter wie Firewalleinstellungen, Paketverlust und Verbindungsverlust können so konfiguriert werden. Bei allen anderen Testsystemen wurde deren Standard-Gateway auf diesen Rechner (1) konfiguriert, wodurch diese das konfigurierbare Netzwerk nutzen. Auf dem Gateway konnte damit die Netzwerkaktivität beobachtet und durch ein Programm zur Analyse des Netzwerkverkehrs (*Wireshark*) der Paketverkehr eingesehen werden.

Tests mit Teilnehmern auf unterschiedlichen Betriebssystemen (Windows, Linux, MacOS) zeigten, dass Saros-Sitzungen erfolgreich gestartet und betrieben werden konnten.

Paketverluste

Für Tests mit Paketverlust konfigurierte ich das Gateway mittels *iptables* auf einen Verlust von 10% der ein- als auch ausgehenden Pakete und bestätigte den Paketverlust mittels *Ping*[2].

```
# for randomly dropping 10% of incoming packets:
iptables -A INPUT -m statistic --mode random --probability 0.1 -j DROP

# and for dropping 10% of outgoing packets:
iptables -A OUTPUT -m statistic --mode random --probability 0.1 -j DROP
```

Listing 4: Konfiguration des Gateways mit iptables für Paketverlust

In Saros-Sitzungen über dieses Netzwerk mit Paketverlust, wurden keine Probleme festgestellt. Dies ist darauf zurückzuführend, dass *Smack* sowohl Nachrichten als auch *Bytestreams* via *TCP* überträgt. Das *Transmission Control Protocol* (*TCP*) bietet einen gesicherten Versand, bemerkt Paketverluste (über Sequenznummern der Pakete) sowie Paketfehler (mittels Checksummen über Pakete) und korrigiert diese Fehler durch erneuten Versand der Pakete. Das Beobachten des Paketverkehrs auf dem Gateway zeigte entsprechend eine Häufung von *TCP-Retransmissions* während einer Saros-Sitzung bei aktiviertem Paketverlust (Abbildung 8).

[2] Diagnosewerkzeug zur Prüfung der Erreichbarkeit eines anderen Rechners, durch Senden von Paketen und Empfangen von Bestätigungen.

Capturing from eth1 - Wireshark

No.	Time	Source	Destination	Protocol	Info
289	20:41:09.252772	192.168.1.4	160.45.111.17	TCP	[TCP segment of a reassembled PDU]
290	20:41:09.252777	192.168.1.4	160.45.111.17	TCP	[TCP segment of a reassembled PDU]
291	20:41:09.252898	160.45.111.17	192.168.1.4	TCP	xmpp-client > ms-sql-s [ACK] Seq=955 Ack=48768 Win=64240 Len=0
292	20:41:09.579884	192.168.1.4	160.45.111.17	TCP	[TCP Retransmission] [TCP segment of a reassembled PDU]
293	20:41:09.580374	160.45.111.17	192.168.1.4	TCP	xmpp-client > ms-sql-s [ACK] Seq=955 Ack=49304 Win=64240 Len=0
294	20:41:09.580962	192.168.1.4	160.45.111.17	TCP	[TCP Retransmission] [TCP segment of a reassembled PDU]
295	20:41:10.236411	192.168.1.4	160.45.111.17	TCP	[TCP Retransmission] [TCP segment of a reassembled PDU]
296	20:41:10.236870	160.45.111.17	192.168.1.4	TCP	xmpp-client > ms-sql-s [ACK] Seq=955 Ack=49531 Win=64240 Len=0
297	20:41:12.679164	fe80::7d6f:74b5:8d7e:7629	ff02::1:2	DHCPv6	Solicit XID: 0x113e90 CID: 0001000114ca4c171cc1defbb9b3
298	20:41:14.127169	192.168.1.4	160.45.111.17	TCP	[TCP segment of a reassembled PDU]
299	20:41:14.130340	160.45.111.17	192.168.1.4	TCP	xmpp-client > ms-sql-s [ACK] Seq=955 Ack=49657 Win=64240 Len=0
300	20:41:14.165743	160.45.111.17	192.168.1.4	TCP	[TCP segment of a reassembled PDU]
301	20:41:14.167001	192.168.1.4	160.45.111.17	TCP	[TCP segment of a reassembled PDU]
302	20:41:14.167498	160.45.111.17	192.168.1.4	TCP	xmpp-client > ms-sql-s [ACK] Seq=1167 Ack=49753 Win=64240 Len=0
303	20:41:14.201466	160.45.111.17	192.168.1.4	TCP	[TCP segment of a reassembled PDU]
304	20:41:14.301715	160.45.111.17	192.168.1.4	TCP	[TCP Retransmission] [TCP segment of a reassembled PDU]

Abbildung 8: Wireshark protokolliert TCP-Retransmissions bei aktiviertem Paketverlust

3.1.2 Anwenderumfragen

Den wissenschaftlichen Wurzeln von Saros ist es zu verdanken, dass Werkzeuge zum Studium von dessen Nutzung implementiert wurden. Neben dem Anfertigen und Übermitteln von Nutzungsstatistiken haben Anwender auch die Möglichkeit, Feedback in Form eines Online-Umfragebogen zu geben. Im Anhang 10.3 ist dieser Umfragebogen aufgeführt.

Nach Sammlung und Analyse aller bisher übermittelten Umfragebögen, habe ich aus den gegebenen Antworten die folgenden 6 Rubriken erstellt und jeden Fragebogen genau einer Rubrik zugeordnet:

- Erfolg:
  Anwender haben ihr Vorhaben erfolgreich umsetzen können.
- kritische Probleme:
  Anwender geben ein nicht näher spezifiziertes Problem an, welches sie vom erfolgreichen Einsatz abhielt.
- Netzwerkprobleme:
  Anwender berichten von schweren Netzwerkproblemen, die den erfolgreichen Einsatz von Saros behinderte.
- wichtiges Feature vermisst:
  Anwender vermissen eine ihnen wichtige Eigenschaft in Saros, berichten aber von keinem Problem.
- langsamer Datentransfer:
  Anwender können Saros erfolgreich einsetzen, kritisieren jedoch einen langsamen Datentransfer bzw. Austausch von Änderungen (Aktivitäten).
- Inkonsistenzen begegnet:
  Anwender berichten von aufgetretenen Projekt-Inkonsistenzen während einer Saros-Sitzung

Das folgende Diagramm zeigt die Anteile aller übermittelten Umfragebögen für diese Rubriken aus dem Zeitraum September 2009 bis März 2011.

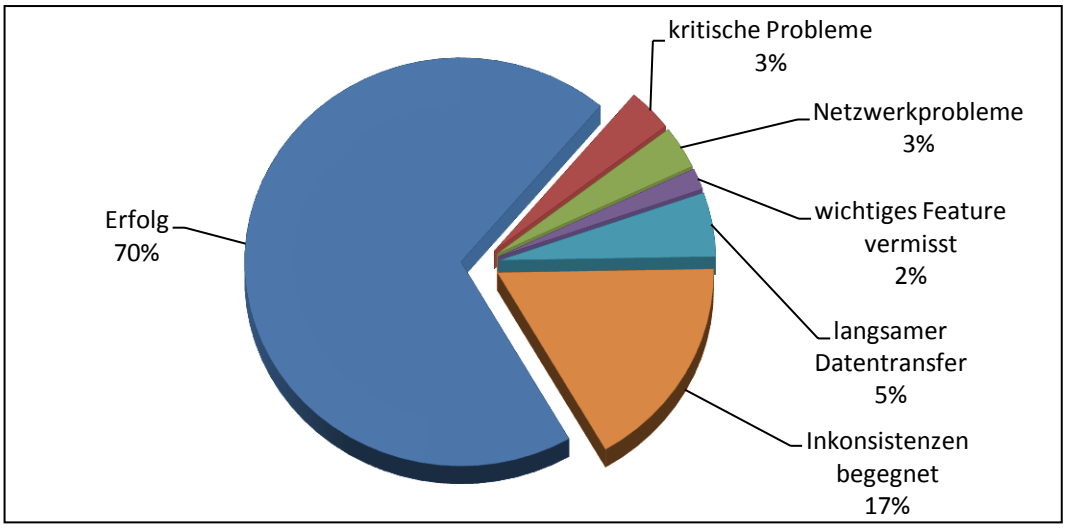

Abbildung 9: Anteile von Umfragebögen je Rubrik über 61 Teilnehmer (September 2009 – März 2011)

Diese Umfrageergebnisse sind nicht repräsentativ für die Gesamtheit der Erfahrungen mit Saros, da zum einen die Auswahl der Stichproben nicht zufällig erfolgte, sondern die Teilnehmer freiwillig und durch Eigenantrieb mitwirkten. Zum anderen ist die Größe der Stichprobe mit 61 Personen in 19 Monaten verschwinden gering. Da Saros in diesem Zeitraum rund 330000 Downloads alleine von der *Sourceforge*-Downloadseite verzeichnete (Saros Installationen über die *Update-Site* nicht eingerechnet) ist von einer deutlich größeren Nutzerzahl auszugehen.

Das Maß der Zufriedenheit der Saros-Nutzer ist aber nicht Ziel dieser Umfrageauswertung, sondern nur ein Nebenprodukt. Im Rahmen dieser Arbeit interessierten vor allem die Meldungen von Problemen, welche die Netzwerkfunktionalität betreffen – zum einen die Häufigkeit dieser Meldungen und zum anderen dokumentierte Details wie Versagensbeschreibungen und Protokolldateien, um Versagensfällen nachgehen zu können. Ich gehe im Rahmen dieser Diplomarbeit nur auf netzwerkrelevante Berichte ein.

Es zeigte sich, dass nur 2 der 61 Teilnehmer sich über Verbindungsprobleme beklagten, wobei jedoch nicht Details zu Verfügung standen, um diese erfolgreich zu untersuchen.

Interessant war weiterhin, dass mehrere Berichte (5%) über langsame Datenübertragungen zu finden waren, die auf einen genutzten *In-Band Bytestream* schließen lassen (vergleiche Geschwindigkeitsvergleich in Kapitel 1.3). Der *IBB* wird, wie in Kapitel 1.3 beschrieben, als Fallbackverfahren eingesetzt, falls eine Verbindung über *Socks5 Bytestream* nicht aufgebaut werden kann. Diese Fälle wurden auch mehrfach während meiner Tätigkeit in der Saros-Arbeitsgruppe an mich herangetragen. Die Verfügbarkeit der *Socks5 Bytestream* Verbindungen zu verbessern, nahm ich mir als einen der zu verbessernden Aspekte in Saros vor.

Bug Reports

Es kam während meiner Zeit im Saros-Team auch dazu, dass Fehler berichtet oder an mich herangetragen wurden, welche ich bearbeitete. Netzwerkrelevante Fälle dessen habe ich in Kapitel 3.2.3, 3.2.4 und 3.2.7 beschrieben.

3.2 Identifizierte Versagensfälle und Lösungsbeschreibungen

Dieses Kapitel beschreibt aufgedecktes Versagen in Saros, die Defektsuche und -behebung.

Die Defektsuche erwies sich des Öfteren als schwierig und langwierig, da mir sowohl die interne Funktionsweise der Netzwerkschicht von Saros als auch der zugrundeliegenden Bibliothek *Smack* durch das *Debuggen* erst vertraut werden mussten. Der bereits etablierte Einsatz von Fehlerprotokollierung half mir oft dabei, Problemen nachzugehen.

3.2.1 Netzwerkproblem beim Einladenden führt zu blockierender Einladung beim Eingeladenen

Aus Erfahrungen von Saros-Entwicklern war bekannt, dass es während des Einladungsprozesses zu Problemen in Form von einer nicht bedienbaren (hängenden) Saro-Instanz kommt, wenn die andere nicht mehr antworten kann. Sei es durch Kommunikationsstörungen oder erzwungene Beendigungen von Instanzen.

Um diese Berichte zu untersuchen, habe ich manuell folgende Abbruchsszenarien während des Einladungsprozesses getestet:

1. Abbruch durch den Einladenden, während Eingeladener die Einladung annehmen soll.
2. Abbruch durch den Eingeladenen, während dieser das neue Projekt konfiguriert (*AddProject*-Assistent).
3. Abbruch während der Datenübertragung der Projektsynchronisation.
    - 3.1 Abbruch durch den Einladenden
    - 3.2 Abbruch durch den Eingeladenen

Bei diesen manuellen Tests zeigte sich, dass es in den Fällen 1 und 2 zu keiner kritischen Situation kam und auch Fall 3.2 ordentlich behandelt wurde.

Im Fall 3.1 kam es jedoch dazu, dass der Eingeladene sein Saros (und Eclipse) nicht mehr bedienen konnte, da unendlich auf Daten gewartet wurde und ein modaler Dialog den Zugriff auf Saros und Eclipse verhinderte.

Bei der Analyse des Netzwerkcodes und dem *Debuggen* dieses Falles offenbarte sich der Grund dieses Verhaltens:

Während des Empfangs einer Dateneinheit (`IncomingTransferObject`), in diesem Falle des Projektarchivs, wartet der Empfänger auf Pakete, welche vom `BinaryChannel` in einer blockierenden Warteschlange verwaltet werden. Durch diese kann ein Aufruf zum

Entnehmen eines Elementes verzögert werden, bis Elemente in der Warteschlange zur Verfügung stehen. Kommen jedoch keine neuen Elemente hinzu und ist die Warteschlange leer, wird unbegrenzt gewartet. Dies passiert in dem Falle, dass vom Sender keine neuen Daten eintreffen, da dieser das Senden ohne Mitteilung einstellte.

```
...
while (true) {
        if (progress.isCanceled()) {
                reject();
                throw new LocalCancellationException();
        }

        BinaryPacket packet;
        try {
                packet = myPackets.take();
                ...
        }
        Spätere Abbruchbedingungen…
}...
```

Listing 5: vereinfachter Codeausschnitt für das Empfangen eines `BinaryChannelTransferObject`

Die Lösung ist hier, nicht mehr mit `BlockingQueue.take()` unendlich auf neue Elemente zu warten, sondern Abbruchbedingungen auch während des Wartens zu testen. Die Korrektur dieses Defektes beinhaltet nun das *Pollen*, also das wiederholte Abfragen nach neuen Paketen, mit einem Timeout. Dadurch kann auf neue Elemente gewartet, aber regelmäßig auch auf einen Abbruch des Vorganges und der Verbindung geprüft werden.

```
...
while (true) {
        if (!this.binaryChannel.isConnected())
                throw new LocalCancellationException(
                "Data connection lost.", CancelOption.NOTIFY_PEER);

        if (progress.isCanceled()) {
                throw new LocalCancellationException(
                "Data reception was manually cancelled.",
                CancelOption.NOTIFY_PEER);
        }

        BinaryPacket packet;
        try {
                packet = myPackets.poll(5, TimeUnit.SECONDS);
                if (packet == null)
                        continue;
        }

        Spätere Abbruchbedingungen…
}...
```

Listing 6: vereinfachter korrigierter Codeausschnitt für das Empfangen eines `BinaryChannelTransferObject`

Nun kann der Abbruch der Datenverbindung erkannt und durch einen Abbruch des lokalen Einladungsprozesses behandelt werden.

Nun wurde zwar der Eingeladene über den Abbruch informiert, jedoch blieb anschließend der Einladungsassistent offen, was der Erwartung eines Anwenders widerspricht. Daher implementierte ich weiterhin, dass sich der Assistent beim Abbruch des Einladungsprozess selbstständig schließt – eine Information zum Abbruch wurde ihm bereits angezeigt.

3.2.2 Abbruch der IBB Verbindung beim Einladungsprozess

Ein schwer eingrenzbares Versagen während des Einladungsprozesses war, dass beim Einsatz des *IBB* zur Projektsynchronisation die Datenverbindung sporadisch abbrechen konnte. Der Einladende erhält von *Smack* die Ausnahme „`No response from server`“. Der Eingeladene erhält dagegen die Meldung, dass die Verbindung abgebrochen wurde, da keine zugehörigen Pakete mehr eintrafen („`IBB [ecftest1@jabber.ccc.de/Saros] Connection was closed by peer. No more packets`“).

Das Nachstellen erwies sich als schwierig, da sich dieses Versagen während der Einladung nicht sicher reproduzieren ließ und somit die beteiligten Faktoren unbekannt waren. Beim Testen mit verschiedenen Servern trat jedoch zutage, dass bei Nutzung des *XMPP*-Servers „jabber.ccc.de“ dieses Versagen leichter zu reproduzieren war, wenn auch nicht hundertprozentig. Ein weiterer Faktor musste also zum Versagen beitragen. Beim weiteren Testen unter verschiedenen Bedingungen fiel weiterhin auf, dass sich nun der Verbindungsabbruch durch das Generieren von Aktivitäten (beispielsweise `EditorActivities` durch Bearbeiten einer geteilten Ressource) auslösen ließ. Werden also parallel zum Datenversand, welcher bei *IBB* mittels *IQ-Stanzas* geschieht, zusätzlich weitere *Stanzas* zur Übertragung generiert, führte dies wiederholbar zum Verbindungsabbruch.

Der Server „jabber.ccc.de“ weist eine hohe Drosselung von Nachrichten auf, was bereits in den Durchsatzmessungen in Kapitel 1.3 verdeutlicht wurde. Bei Auswertung der übertragenen Pakete durch das *Smack*-interne *Debug*-Fenster ließ sich diese Drosselung beobachten.

ecftest1@jabber.ccc.de/Saros | Smack Info

All Packets | Raw Sent Packets | Raw Received Packets | Ad-hoc message | Information

Timestamp			Message	Id	Type	To	From
11:33:43 AM			IQ Received (class=org.jivesoft...	4oSpw-207	result		ecftest2@j...
11:33:43 AM			IQ Sent (class=org.jivesoftware....	4oSpw-208	set	ecftest2@j...	
11:33:46 AM			IQ Received (class=org.jivesoft...	4oSpw-208	result		ecftest2@j...
11:33:46 AM			IQ Sent (class=org.jivesoftware....	4oSpw-209	set	ecftest2@j...	
11:33:49 AM			IQ Sent (class=org.jivesoftware....	4oSpw-210	set	ecftest2@j...	
11:33:49 AM			IQ Received (class=org.jivesoft...	4oSpw-209	result		ecftest2@j...
11:33:52 AM			IQ Received (class=org.jivesoft...	4oSpw-210	result		ecftest2@j...

Abbildung 10: Smack Debug Fenster zur Paketverfolgung des Senders

ecftest2@jabber.ccc.de/Saros | Smack Info

All Packets | Raw Sent Packets | Raw Received Packets | Ad-hoc message | Information

Timestamp			Message	Id	Type	To	From
11:33:46 AM			IQ Received (class=org.jivesoft...	4oSpw-208	set		ecftest1@ja...
11:33:46 AM			IQ Sent (class=org.jivesoftware....	4oSpw-208	result	ecftest1@ja...	
11:33:49 AM			IQ Received (class=org.jivesoft...	4oSpw-209	set		ecftest1@ja...
11:33:49 AM			IQ Sent (class=org.jivesoftware....	4oSpw-209	result	ecftest1@ja...	
11:33:52 AM			IQ Received (class=org.jivesoft...	4oSpw-210	set		ecftest1@ja...
11:33:52 AM			IQ Sent (class=org.jivesoftware....	4oSpw-210	result	ecftest1@ja...	

Abbildung 11: Smack Debug Fenster zur Paketverfolgung des Empfängers

Das Paket „4oSpw-209“ wird um 11:33:46 gesendet und trifft erst 11:33:49 beim Empfänger ein. Pakete müssen vom Empfänger bestätigt werden (Pakettyp: *result*). Diese Bestätigung trifft nun ebenfalls verzögert ein. Werden viele Pakete versendet, schaukelt sich die Verzögerung auf. Übersteigt sie fünf Sekunden, bricht die *IBB*-Verbindung ab.

Diese magischen fünf Sekunden fanden sich in der Standardeinstellung von *Smack* für den *Timeout* der Paketantworten wieder. Der erste Ansatzpunkt um den Verbindungsabbruch zu vermeiden ist das Heraufsetzen des Timeouts via *Smack* mit:

```
SmackConfiguration.setPacketReplyTimeout(timeout);
```

Bei Servern mit hoher Paketdurchsatz-Limitierung kann es in Ausnahmesituationen auch so zu diesen Timeouts kommen, wenn über einen längeren Zeitraum viele `Activity`-Nachrichten die *IBB*-Nachrichten ausbremsen.

Als zweiter Ansatz wirkt weiterhin das Bündeln von Aktivitäten während der Projektinitialisierung dem Timeout entgegen. Der `ActivitySequencer`, welcher für das Sammeln, Sortieren, Bündeln und Versenden von Aktivitäten verantwortlich ist, wurde von mir dahingehend erweitert, dass während der Projektsynchronisierung Aktivitäten stärker gebündelt werden und somit weniger Nachrichten versendet werden müssen.

Beide Ansätze kombiniert, wirken nun den beschriebenen Verbindungsabbrüchen erfolgreich entgegen.

3.2.3 IBB Warnung beim Einladungsprozess

Wie im vorherigen Punkt ersichtlich wurde, sind Datenübertragungen über *IBB* nicht nur langsam, sondern auch anfällig bei serverseitiger Limitierung. Es wird bereits während des Einladungsprozesses im Assistenten ein Hinweis über die verwendete Übertragungsart ausgegeben. Bei *IBB*-Verbindungen wird entsprechend gewarnt, dass der Datenaustausch langsam sein wird.

Während der Tests vor einem Release fiel diese Warnung den Testmanagern nicht auf, obwohl es sich um ein großes Warnsymbol handelte (Abbildung 12). Daher wurde in Absprache mit ihnen nach einer Möglichkeit gesucht, diesen Hinweis besser wahrnehmbar zu machen.

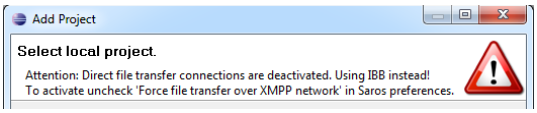

Abbildung 12: Warnung bei langsamer Datenverbindung

Blinkende Reize werden vom Menschen leichter wahrgenommen und lenken die Aufmerksamkeit. Entsprechend schreibt die StVZO § 53a, Absatz 4, 2. Punkt, Warnblinklicht mit einer Frequenz von 1 bis 2 Herz vor.

Aufgrund dessen wurde das Warnsymbol blinkend implementiert und der Warnhinweis insofern konkretisiert, dass der Durchsatz des *IBB* mitgeteilt wird, sofern dieser durch die Übertragung der Dateiliste (als Teil des Einladungsprozesses) über den *Bytestream* bekannt ist (Abbildung 13).

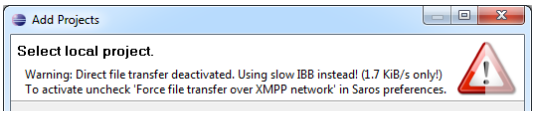

Abbildung 13: neue Warnung bei langsamer Datenverbindung (Blinken angedeutet)

3.2.4 Kontaktumbenennung von Smack unbemerkt

Während meiner Arbeiten im Saros-Team wurde von einem Kommilitonen ein Versagen im Bezug auf das Umbenennen eines Kontaktes (`RosterEntry`) in der Kontaktliste (`Roster`) gemeldet. Der geänderte Name erschien nicht selbstständig in der Kontaktliste, sondern erst nach manueller Aktualisierung (bisheriger *Workaround*) oder durch Neuverbinden mit dem Server.

Smack sieht für das Umbenennen eines `RosterEntry` folgende Prozedur vor:

1. *API*-Aufruf zum Umbenennen eines Kontaktes von der Anwendung (Saros) aus `RosterEntry.setName(newName);`
2. *Smack* baut und sendet ein *IQ Paket* mit zu setzenden Eigenschaften des Kontaktes für die Kontaktliste auf dem *XMPP*-Server
3. *XMPP*-Server antwortet zur Bestätigung mit einem *IQ Paket*, welches den aktualisierten Kontakt enthält
4. *Smack* wertet dieses Paket in der inneren Klasse `Roster.RosterPacketListener` aus:
   a) Anhand der *JID* wird festgestellt, dass dieser Kontakt bereits in unserer lokalen Kontaktliste ist.
   b) Die Prüfung der Eigenschaften dieses Kontaktes erkennt eine geänderte Eigenschaft.
   c) Die Liste wird mit dem empfangenen Eintrag aktualisiert.
   d) Angemeldete *Listener* werden über aktualisierten Eintrag informiert, welche daraufhin z.B. die Anzeige aktualisieren können.

Aufgrund dessen muss auch ohne manuelle Aktualisierung der geänderte Name angezeigt werden können. Es zeigte sich beim *Debuggen*, dass Saros korrekt den neuen Namen setzen lässt, aber keinen Aufruf erhält, die Liste zu aktualisieren. Der entsprechende Defekt befindet sich in *Smack* und liegt darin, dass zum Senden des Änderungspaketes (Schritt 2) das Kontakt-Objekt (`RosterEntry`) bereits mit dem neuen Namen versehen wurde. Dadurch kann in Schritt 4b die Änderung nicht bemerkt werden, wodurch kein Event zur Aktualisierung gesendet wird.

Es folgt der betrachtete Codeausschnitt (original und korrigiert):

RosterEntry.setName (original)	**RosterEntry.setName (korrigiert)**

`this.name = name;`
 	
`RosterPacket packet = new RosterPacket();`
`packet.setType(IQ.Type.SET);`
`packet.addRosterItem(toRosterItem (this));`
`connection.sendPacket(packet);`
	`RosterEntry updatedRosterEntry = new RosterEntry(this.user, name, this.type, this.status, this.roster, this.connection);`
	
	`RosterPacket packet = new RosterPacket();`
	`packet.setType(IQ.Type.SET);`
	`packet.addRosterItem(toRosterItem(updatedRosterEntry));`
	`connection.sendPacket(packet);`

Tabelle 5: Vergleich `RosterEntry.setName` im Original und korrigiert

In meiner Korrektur wird nicht das aktuelle `RosterEntry`-Objekt angepasst und versendet, sondern eine Kopie dessen mit dem neuen Namen. Wenn vom Server die Antwort eintrifft,

enthält sie nun einen `RosterEntry`, welcher sich im Namen vom lokalen `RosterEntry` unterscheidet. Dadurch kann der `Roster` ein Aktualisierungsevent an angemeldete *Listener* senden.

Ein von unserem betrachteten Versagen unabhängiger Defekt fiel mir beim *Debuggen* zusätzlich auf. Er kann in jenen Fällen zu Versagen führen, wenn Anwendungen mit Verweisen oder Kopien von `RosterEntry`-Objekten arbeiten.

In der Implementierung von *Smack* zu Punkt 4 der zuvor genannten Prozedur der Umbenennung wird ein empfangenes `RosterEntry`-Objekt immer zum Aktualisieren in die lokale Liste gesetzt, auch wenn es mit identischem Inhalt schon enthalten ist. In diesem Fall wird jedoch keine Änderungsinformation ausgelöst, obwohl das `RosterEntry`-Objekt durch ein anderes mit gleichem Inhalt ersetzt wurde.

Es folgt der betrachtete Codeausschnitt (original und korrigiert):

Roster.RosterPacketListener. processPacket (original)	**Roster.RosterPacketListener. processPacket (korrigiert)**

`RosterEntry oldEntry =`
`  entries.put(item.getUser(), entry);`
 	
`if (oldEntry == null ||`
`  !oldEntry.equalsDeep(entry)) {`
`  updatedEntries.add(item.getUser());`
`}`
	`RosterEntry currentEntry =`
	`  entries.get(item.getUser());`
	
	`if (currentEntry!=null &&`
	`!currentEntry.equalsDeep(entry)) {`
	`  entries.put(item.getUser(), entry);`
	`  updatedEntries.add(item.getUser());`
	`}`

Tabelle 6: Vergleich `Roster.RosterPacketListener.processPacket` im Original und korrigiert

In meiner Korrektur wird das `RosterEntry`-Objekt nur ersetzt und darüber informiert, wenn dessen Inhalt nicht dem des aktuellen Objektes, der gleichen *JID,* entspricht.

Da die genannten Defekte in *Smack* zu finden und zu beheben sind, habe ich einerseits unsere lokale *Smack*-Bibliothek für Saros angepasst, als auch einen *Bug Report* im dafür vorgesehenen Forum (http://community.igniterealtime.org) der Smack Entwickler verfasst. Dieser wurde von einem Entwickler an einen dazugehörigen Eintrag im offiziellen *Bug Tracker* geknüpft.

3.2.5 Versagen beim Verbindungsaufbau via Socks5, Rückfall auf IBB

Dieses Kapitel behandelt ein Versagen in der Netzwerkschicht von Saros, welches zu korrigieren spürbare Verbesserungen beim Datentransfer mit sich bringen kann: die *Socks5 Bytestream* Verbindung.

Sobald beteiligte Rechner nicht direkt adressierbar sind, beispielsweise keine eigene öffentliche *IP*-Adresse haben und sich nicht im gleichen Subnetz befinden, kann kein *S5B* aufgebaut werden. *Smack* meldet in solchen Fällen eine `XMPPException` mit der Meldung:

> `Could not establish socket with any provided host:` **`item-not-found(404)`**

Wie im *Socks5 Bytestream* Protokoll [XEP65] definiert, übermittelt der *Requester* dem *Target* eine Liste von Internetadressen (*StreamHosts*), an welche dieser *TCP*-Verbindungsanfragen richten soll. Kann mit keinem dieser *StreamHosts* eine *TCP*-Verbindung aufgebaut werden, bricht *Smack* den *S5B*-Verbindungsaufbau mit oben genannter Fehlermeldung ab (in `Socks5BytestreamRequest.accept()`). Kennen beide Beteiligten keine erreichbare Internetadresse des anderen, kann kein *S5B* aufgebaut werden.

Ein lokaler *Socks5-Proxy* (Klasse `Socks5Proxy` von *Smack*) dient jeder Saros-Instanz als *StreamHost*. Dieser wird bei der Initialisierung mit der lokalen Adresse des Rechners konfiguriert, welche über

```
InetAddress.getLocalHost().getHostAddress();
```

ermittelt wird. Diese Adresse wird beim *S5B*-Verbindungsaufbau dem Gegenüber mitgeteilt. Die Annahme, dass hier eine unbrauchbare Adresse ermittelt und übermittelt wird, erwies sich in einigen Testfällen als richtig. Java ermittelt für den obigen Aufruf auf Systemen mit bestimmten Java-Versionen (OpenJava) die *Loopback*-Adresse (127.0.0.1). Wird nun der *Socks5Proxy* mit dieser Adresse initialisiert und als eigene Adresse beworben, werden andere Rechner ihn nicht unter dieser Adresse erreichen können. Hier hilft es, alle korrekten lokalen *IP*-Adressen zu ermitteln und damit den *Socks5Proxy* zu konfigurieren.

Beispiel: Auf einem frisch installierten *Ubuntu* mit installiertem *OpenJDK RE IvedTea6* 1.9.7, ermittelt der Aufruf `InetAddress.getLocalHost()` die *Loopback*-Adresse 127.0.0.1, obwohl ein Netzwerkadapter mit der *IP* 192.168.1.126 konfiguriert wurde und dies ermittelbar ist.

3.2.5.1 Lokale IP-Adressen

Smack bietet eine Funktion für den *Socks5Proxy* an, um diesen mit einer Liste von *IPs* zu konfigurieren (`replaceLocalAddresses(List<String> addresses))`. Daher implementierte ich eine Funktion, welche alle aktiven Netzwerkadapter des Systems ermittelt und deren *IP*-Adressen (*Loopbackadressen* ausgenommen) sammelt. Mit dieser Liste wird nun der *Socks5Proxy* konfiguriert. Dies ist auch für Systeme mit mehreren Netzwerkadaptern wichtig, um über alle Netzwerkadapter erreichbar zu sein.

Die Verbesserung der *Socks5Proxy*-Konfiguration mit lokalen Adressen wirkt sich entsprechend nur auf den Verbindungsaufbau im lokalen Netzwerk aus. Um die Effektivität dieser Änderung zu bestätigen, führte ich eine Reihe von Verbindungstest zwischen Saros-Instanzen auf Rechnern im Institutsnetz durch.

Testkonfiguration:

- Tester1: Win2008, Desktoprechner „Idaban“, FU-LAN
- Tester2: Win7, Eduroam
- Tester3: VM WinXP, über Gateway an Eduroam

Die Ergebnisse der Verbindungstests werden in Tabelle 7 dargestellt.

Durch die Konfiguration des *Socks5Proxy* mit allen ermittelbaren lokalen *IP*-Adressen, verbesserte sich die Erreichbarkeit via *S5B* im lokalen Netz deutlich.

Testpartner, jabber.org	**vorher**	**nachher**
Tester1 → Tester2	*IBB* schlägt fehl	Socks5
Tester1 → Tester3	*IBB* schlägt fehl	Socks5
Tester2 → Tester1	*IBB*	Socks5
Tester3 → Tester1	*IBB*	Socks5
Tester2 → Tester3	*IBB*	Socks5
Tester3 → Tester2	*IBB*	Socks5

Tabelle 7: Verbindungstest Resultate – vor und nach zusätzlicher IP-Bestimmung

3.2.5.2 Öffentliche IP-Adresse

Für Rechner, die keine öffentliche *IP*-Adresse zugewiesen haben (beispielsweise weil sich diese hinter einem Gateway befinden und nur indirekt mit dem Internet verbunden sind) hilft die vorherige Maßnahme nicht, um aus dem Internet heraus adressierbar zu sein. Es ist notwendig, auch die öffentliche *IP* des Rechners zu ermitteln und dem *Socks5Proxy* mitzuteilen, um diese Adresse dem Verbindungsgegenüber mitzuteilen.

Ein etabliertes Verfahren, das hierfür genutzt werden kann, ist *STUN* (*Session Traversal Utilities for NAT*), welches 2008 von der *IETF* im *RFC5389* beschrieben wurde. Dieses *RFC* ersetzt das ältere *RFC3489*, welches wegen mangelnder Verlässlichkeit seiner beworbenen Funktionalitäten abgelöst wurde [RFC5389].

STUN wird u.a. von der Bibliothek *jStun* implementiert (noch als *RFC3489*) und ist in *Smack* enthalten, was dessen Nutzung von Saros aus nahe legt. Zur Ermittlung der öffentlichen *IP* eines Rechners, sendet ein *STUN*-Client (z.B. Saros mittels *jSTUN*) eine Anfrage (*Binding Request*) an einen *STUN* Server via *UDP*[3]. Der Server kann aus diesem Datagramm die öffentliche Senderadresse und den Senderport auslesen und sendet ein UDP-Datagramm an den Absender zurück, welches Adresse und Port im Datenbereich des Datagramms überträgt. *STUN* bietet auch die Möglichkeit, den *NAT*-Typ festzustellen, was für Saros in diesem Kontext jedoch nicht relevant ist.

In Saros implementierte ich den Aufruf von *jStun* so, dass nach dem Verbinden zum *XMPP*-Netzwerk die Adressbestimmung durchführt und das Ergebnis der Adressliste des *Socks5Proxy* hinzugefügt wird. Dabei wird eine evtl. vorher gesetzte öffentliche *IP* entfernt. Hier ist ebenfalls zu beachten, dass die *STUN*-Anfrage über alle lokalen Netzwerkadapter gesendet wird. Somit kann Saros seine öffentliche *IP* anderen Instanzen, welche sich nicht im

[3] User Datagram Protocol, ungesichertes Transportprotokoll (RFC 768)

gleichen lokalen Netzwerk befinden, mitteilen und ihnen so den Aufbau einer *TCP*-Verbindung ermöglichen. Eine weitere Hürde kann nun sein, dass das Gateway nicht in der Lage ist, eingehende Verbindungsanfragen an den Rechner, auf dem Saros läuft, weiterzuleiten. Hierfür wurde von mir *Universal Plug and Play* in Saros integriert, was in Kapitel 4 gesondert behandelt wird.

3.2.6 Behandlung von fehlgeschlagenem Versand von Aktivitäten

Sollen Aktivitäten an andere Teilnehmer gesendet werden, werden sie vom `ActivitySequencer` sortiert, zusammengefasst und als `PacketExtension` (*XML*-Repräsentation) der Klasse `XMPPTransmitter` zum Versand übergeben. Kleine Nachrichten (<16KiB) werden als Nachrichten über das *XMPP*-Netzwerk versendet, falls es keinen bestehenden *S5B* oder *IBB* gibt. Ansonsten wird die `PacketExtension` per *Bytestream* versendet. Kommt es in letzterem Fall zu einem Übertragungsfehler und ist die Nachricht nicht kleiner als 16KiB, wurde die Übertragung bisher mit einem Logeintrag fallengelassen.

Durch die (Wieder-)Einführung einer Sendewiederholung im Fehlerfall, werden nun diese Fälle behandelt. Der Versand via *Bytestream* wird nun standardmäßig vier Mal wiederholt, wobei nach dem zweiten Fehlversuch auf den *IBB* gewechselt wird. Dieser ist zwar langsamer, funktioniert aber verlässlicher, wie in Kapitel 1.3 erläutert wurde.

3.2.7 Einladung bricht bei großen Projekten ab

Ein Eintrag im *Bugtracker*[4] von Saros berichtete vom Abbruch einer Einladung, wenn das zu synchronisierende Projekt auffallend groß war. Dieser *Bugreport* wurde mir zugewiesen, da die Abbruchmeldung aus der Netzwerkschicht von Saros stammte.

Der Empfänger der Einladung bricht während der Synchronisierungsphase nach 10 Sekunden mit der Meldung ab: „`localCancel: Collector timeout: no packet received.`"

Diese Meldung lässt sich im `XMPPTransmitter` in der Funktion `receiveArchive` finden. Hier wird der Empfang eines `TransferObjects` mit einem *Timeout* von 10 Sekunden konfiguriert. Solange jedoch der Einladende mit der Erstellung des Archivs beschäftigt ist, sendet er keine Daten und beim Empfänger kann dieses *Timeout* eintreffen.

Als Lösung wartet der Empfänger nun auf den Beginn der Übertragung und beginnt dann erst den Empfang. Während des Wartens auf den Übertragungsbeginn wird auf geordnete Abbrüche geprüft (Teilnehmer bricht manuell Einladungsprozess ab). Während des Empfanges werden weiterhin still abgebrochene Übertragungen mit dem *Timeout* erkannt und behandelt.

[4] unter der URL http://sourceforge.net/tracker/?group_id=167540&atid=843359

3.2.8 Smack Version aktualisieren

Hennig Staib erweiterte und korrigierte verschiedene Bereiche der *Smack* Bibliothek im Rahmen seiner Diplomarbeit [Sta10]. Aufgrund der geringen Aktivität der *Smack* Entwickler war es nicht möglich, seine Patches[5] in einer offiziellen *Smack*-Version zu nutzen. Daher wurde Saros mit einer selbst aktualisierten Version betrieben und ausgeliefert.

Nach mehr als zwei Jahren Pause, veröffentlichte *Jive Software* im Februar 2011 eine neue Betaversion der *XMPP*-Bibliothek *Smack* und im Mai 2011 eine neue Releaseversion (v3.2). Nach Durchsicht des Änderungsprotokolls und nach Rücksprache mit Henning bestätigte sich, dass alle seine Patches in der neuen Version enthalten waren. Dadurch war es mir möglich, Saros auf diese neue Version von *Smack* zu portieren. Es waren nur geringe Anpassungen in Saros nötig, welche durch geänderte Paketstrukturen in *Smack* notwendig wurden.

Auf die Version 3.2.1 konnte bereits im Juli 2011 gewechselt werden. Jeweils sind die Releases durch die Patches zu erweitern, welche Saros-Entwickler vornehmen und nicht in offizielle *Smack*-Versionen eingeflossen sind (siehe Kapitel 3.2.4). Hierfür soll folgender Hinweis zur Aktualisierung helfen:

Im Unterverzeichnis „lib“ des Saros-Projektverzeichnisses findet sich neben den *Smack*-Bibliotheken auch das Archiv des Quellcodes (smack_src_saros.zip) der angepassten, von Saros benutzten *Smack*-Version. Im Archiv wurde ein zusätzliches Unterverzeichnis „local-Patches“ eingeführt, welches jene Patches enthält, die (noch) nicht Bestandteil einer offiziellen *Smack*-Version sind. Diese Patches sollen den Umstieg auf diese neue Versionen erleichtern, falls die neue Version nicht mit der modifizierten Version zusammengeführt (*gemerged*) werden kann.

[5] beschreibt Änderungen zwischen einer Originalversion und einer Folgeversion eines Quellcodes

4 Integration von Universal Plug and Play

Dieses Kapitel beschreibt die Integration von *Universal Plug and Play (UPnP)* in Saros. Es folgt eine Einführung über *UPnP*, die Motivation dies in Saros zu integrieren, Wissenswertes zur Implementierung und abschließend Informationen zur Zusammenarbeit mit dem Open Source Projekt der benutzten *UPnP*-Bibliothek *weupnp*.

4.1 Einführung UPnP

Das *UPnP* definiert eine Menge von Netzwerkprotokollen, welche es Netzwerkgeräten – vor allem auch in Heimnetzwerken – erlauben sollen, sich automatisch und ohne manuelle Hilfe untereinander zu finden und für eine Zusammenarbeit zu konfigurieren. Es ermöglicht weiterhin die Steuerung und den Datenaustausch zwischen *UPnP*-Geräten.

Als „*UPnP Control Points*" werden jene Geräte bezeichnet, welche über *UPnP*-Protokolle andere Geräte steuern. Die Kommunikation dafür ist unabhängig vom Gerätetyp, Betriebssystem oder Gerätehersteller und basiert auf dem Internet Protocol (*IP*) und darauf aufbauenden, standardisierten Technologien *UDP, TCP, HTTP, XML* und *SOAP*. Die Ansteuerung erfolgt nur über die genannten, gerätunabhängigen Protokolle. Für den Datenaustausch werden *SOAP* Nachrichten über *HTTP*-Verbindungen übertragen. *SOAP* ist ein *XML*-basiertes Protokoll zum Austausch von Nachrichten mit Regeln für dessen Nachrichtenformat. Diese Protokollhierarchie und damit realisierte Dienste werden in Abbildung 14 verdeutlicht.

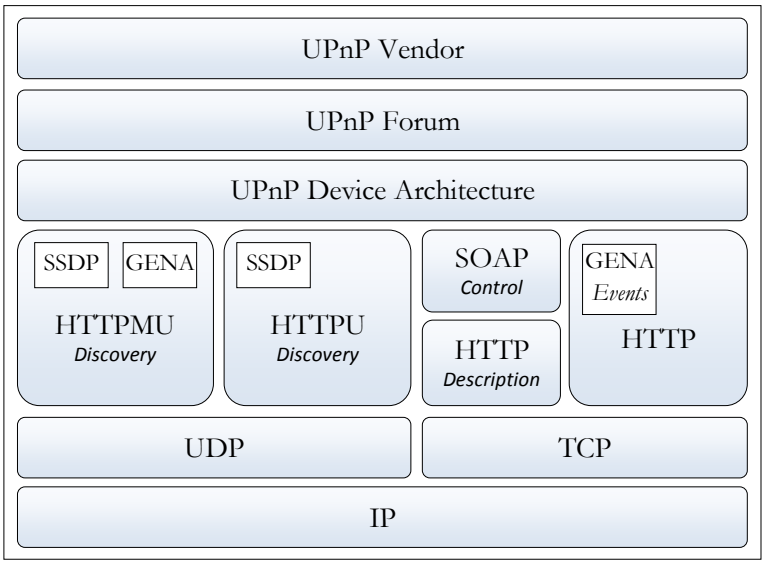

Abbildung 14: UPnP Device Stack (nach UPnP Device Architecture v1.1)

UPnP wurde von Microsoft initiiert und seit 1999 durch das *UPnP* Forum gefördert und geleitet, mit dem Ziel Protokolle und Standards für *UPnP*-Geräte und -technologien zu entwickeln und zu veröffentlichen. Dem UPnP Forum haben sich derzeit (Juli 2011) über 950 Firmen angeschlossen. [UPn11]

Im Jahr 2008 wurde Version 1.1 der *UPnP Device Architecture* Spezifikation veröffentlicht.

4.2 Ablauf von UPnP auf Netzwerkebene

Dieses Kapitel beschreibt die Arbeitsweise von *UPnP*, basierend auf der *UPnP Device Architecture 1.1* Spezifikation. [UPn11]

Adressierung

Das *Internet Protocol* als Grundlage des *UPnP*, setzt für die Netzwerkkommunikation beteiligter Geräte eine konfigurierte *IP*-Adresse voraus. Sofern ein *DHCP* Server verfügbar ist, wird dieser für die Ermittlung einer Adresse benutzt. Andernfalls versorgt sich das *UPnP*-Gerät via *Auto IP* (auch *Zeroconf* genannt) selbstständig mit einer *IP*-Adresse.

Schritt 1: Lokalisierung (Discovery)

Nachdem ein *UPnP*-Gerät einem Netzwerk durch Konfiguration mit einer *IP*-Adresse beigetreten ist, meldet es sich bei den *Control Points* an. Dazu sendet es eine *Notify*-Nachricht als *UDP-Datagramm* via *Multicast* an 239.255.255.250, Port 1900. Tritt ein *Control Point* einem Netzwerk bei, sucht es mittels einer Suchanfrage auf Basis des *Simple Service Discovery Protocol* (*SSDP*) nach Geräten.

In beiden Fällen wird eine *Discovery Message* ausgetauscht, in welcher einige wenige Details zu dem Gerät und den Diensten vermerkt werden, sowie eine Adresse (*URL*) zum Abruf einer genaueren Beschreibung. Bei Geräten, welche mehrere Netzwerkadapter besitzen wird empfohlen, die Nachrichten zur Bekanntmachung und zur Suche über alle Adapter zu senden.

Beispiele:

Suchanfrage eines Control Points nach Geräten des Typs InternetGatewayDevice

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
ST: urn:schemas-upnp-org:device:InternetGatewayDevice:1
MAN: "ssdp:discover"
MX: 2
```

Antwort eines Routers (D-Link DI-524)

```
HTTP/1.1 200 OK
ST:urn:schemas-upnp-org:device:InternetGatewayDevice:1
USN:uuid:001B11A4-0EBE-0000-0000-0000C0A80001::urn:schemas-upnp-
org:device:InternetGatewayDevice:1
Location:http://192.168.0.1:80/desc.xml
Cache-Control:max-age=1800
Server:IGD-HTTP/1.1 UPnP/1.0 UPnP-Device-Host/1.0
Ext:
```

Bekanntmachung eines neuen Gerätes

```
NOTIFY * HTTP/1.1
Cache-Control:max-age=60
Location:http://192.168.0.1:80/desc.xml
NT: urn:schemas-upnp-org:device:InternetGatewayDevice:1
NTS:ssdp:alive
Server: IGD-HTTP/1.1 UPnP/1.0 UPnP-Device-Host/1.0
USN: "uuid:001B11A4-0EBE-0000-0000-0000C0A80001"
```

Schritt 2: Beschreibung (Description)

Um detailliertere Informationen von einem Gerät und dessen Diensten zu erhalten, ruft ein *Control Point* nun die *UPnP*-Beschreibung des Gerätes ab. Dieses Dokument ist in *XML* formuliert und an der in der *Discovery Message* angegebenen *URL* abrufbar.

Zu diesen detaillierten Informationen gehören unter anderem herstellerspezifische Angaben zum Gerät wie Name, Version, Seriennummer und Herstellerhomepage. Weiterhin sind *URLs* für Steuerung, Benachrichtigungen und Präsentation (Konfigurationsoberfläche) enthalten, sowie für jeden Service eine Liste von unterstützten Aktionen und deren Parametern.

Ein Beispiel für eine solche Gerätebeschreibung findet sich in Anhang 10.1, welches die Eigenschaften des im vorigen Beispiel genutzten Routers „D-Link DI-524“ beschreibt.

Schritt 3: Steuerung (Control)

Mit den aus Schritt 2 gewonnenen Informationen zur Steuerung eines Gerätes kann ein *Control Point* nun Aktionen an Dienste des Gerätes senden. Diese Steuernachrichten (*Control Messages*) sind in *XML* formuliert und nutzen das *Simple Object Access Protocol* (*SOAP*). Der Dienst antwortet mit aktionsspezifischen Rückgabewerten.

Schritt 4: Ereignismeldungen (Event Notification)

Ein *Control Point* kann sich bei einem Gerät anmelden, um über Änderungen von Werten verfügbarer Dienste informiert zu werden. Damit wird aufwändiges *Pollen* – wiederholtes Abfragen von Werten – vermieden. Angemeldete *Control Points* werden mittels Ereignismeldungen über Wertänderungen benachrichtigt. Diese Meldungen enthalten die Namen von einer oder mehreren Variablen und deren aktuellen Werten.

Das hierfür verwendete Protokoll ist bekannt als *General Event Notification Architecture* (*GENA*).

Schritt 5: Präsentation (Presentation)

In Schritt 2 wurde eine „*Presentation-URL*“ im gerätebeschreibenden *XML*-Dokument erhalten. Von dieser Adresse kann über einen Webbrowser eine Webseite abgerufen werden. Je nach Hersteller können hier Informationen zum Gerät und zum Status eingesehen oder auch das Gerät konfiguriert und gesteuert werden.

4.2.1 NAT-Netzwerkkonfigurationen

Jeder Rechner benötigt eine eigene *IP*-Adresse, um im Internet kommunizieren zu können. Ist ein Rechner nur in einem lokalen Netz (*LAN*) konfiguriert, besitzt er lediglich eine private *IP*-Adresse (*RFC* 1918), welche zur Identifizierung im Internet nicht genügt. Pakete von diesen Adressen werden im Internet nicht weitergeleitet. Diese Rechner sind auch nicht aus dem Internet heraus adressierbar und somit nicht erreichbar. Das hat den Vorteil, dass die Rechner im *LAN* gegen Angriffe von außen besser abgesichert sind. Aber auch gewünschte Anfragen erreichen sie nicht, wodurch sie ohne weiteres keine Serverdienste bereitstellen können.

Ein Internetgateway kann es Rechnern eines Netzwerkes erlauben, durch dessen Internetverbindung über das Internet zu kommunizieren. Dazu muss das Internetgateway eine öffentliche *IP*-Adresse besitzen. Im Privatbereich sind Kombigeräte aus Modem und Router als Internetgateway verbreitet, wobei das Modem (z.B. DSL-Modem) eine Verbindung mit dem Internet aufbaut. Der Router ermöglicht das Verbinden verschiedener Netzwerke, wobei unterschiedliche Protokolle benutzt werden können, was ein Gateway auszeichnet. Ein DSL-Router sendet beispielsweise *IP*-Pakete aus dem *LAN* über das *PPPoE-Protokoll* [6] zum Internet Provider weiter.

Das *Network Address Translation* (*NAT*)-Verfahren erlaubt es Rechner eines *LANs* über ein Gateway mit dem Internet zu kommunizieren. Dazu manipuliert das *NAT*-Gateway die Pakete, welche aus dem lokalen Netz stammen, deren Zieladressen aber nicht im lokalen Netz liegen. Es ersetzt die Senderadresse eines Paketes durch seine öffentliche Adresse und den Senderport durch einen ihm freien Port. Diese Zuordnung von Quell-*IP*, Ziel-*IP* und neuem Quell-Port legt er in einer *NAT*-Tabelle ab und leitet anschließend das Paket an sein Ziel weiter. Die Antwort auf dieses Paket kann das Gateway anhand des Absenders und Zielports über seine *NAT*-Tabelle dem Zielrechner im LAN wieder zuordnen und durch Ersetzen von Zieladresse und Zielport an diesen weiterleiten. Dieses Verfahren wird *Source NAT* bezeichnet.

[6] PPP over Ethernet, Netzwerkprotokoll, RFC 2516

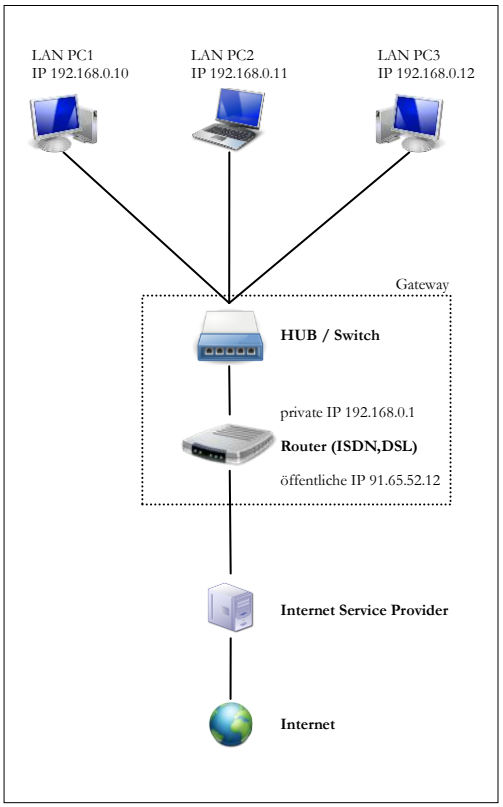

Abbildung 15: Netzwerkszenario für Clients in LAN mit NAT Gateway

Sind mehrere Clients in einem *LAN* organisiert, so sind ihnen private Internetadressen zugewiesen worden (z.B. über *DHCP*) und sie kommunizieren untereinander über diese. Mit dem Internet können sie mit ihnen jedoch nicht arbeiten.

Ein Gateway (z.B. DSL-Router) ist über einen Netzwerkadapter mit dem lokalen Netzwerk verbunden und kann über einen zweiten Netzwerkadapter eine Verbindung mit dem Internet aufbauen (z.B. via *PPPoE*). Haben die Rechner im *LAN* ihren *Default Gateway* auf die private *IP* des Internetgateways konfiguriert, so werden alle Pakete, deren Zieladresse außerhalb der Adressmaske des lokalen Netzwerkes liegen, an diesen gerichtet. Das Gateway leitet diese Pakete mittels *Source NAT* an ihre Empfänger im Internet weiter und reicht Antwortpakete dem Ursprungsrechner im lokalen Netz zurück.

Zurück bei Saros wird damit verständlich, weshalb bei Rechnern in der genannten Netzwerkkonfiguration eingehende *TCP*-Verbindungsanfragen des *Socks5 Bytestream* Verbindungsaufbaues nicht ankommen. Diese Anfrage (*TCP SYN* Paket) trifft am Internetgateway ein (ihm ist die öffentliche *IP* zugeordnet) und dieses hat keinen Anlass, es an einen lokalen Rechner weiterzuleiten.

Die Lösung für dieses Problem ist es, das Gateway so zu konfigurieren, dass bestimmte Pakete immer an einen gewünschten Rechner im *LAN* weitergeleitet werden, was anhand des Zielportes konfiguriert werden kann. Liegt der Zielport eines eintreffenden Paketes in einem gegebenen Intervall, wird dieses an einen bestimmen lokalen Rechner mit einer konfigurierten *IP* mittels *Destination NAT* weitergeleitet. Die Zieladresse (öffentliche *IP* des Gateways) des Paketes wird dabei durch die private *IP* des Zielrechners ersetzt. Eine solche Regel wird *Port Mapping* (Portweiterleitung) genannt. Unterstützt das *Gateway* diese Funktionalität, können Regeln für verschiedenen *IP*-Bereiche und für verschiedene lokale Rechner konfiguriert werden.

Das Erstellen von Portweiterleitungen kann manuell durchgeführt werden oder automatisiert über das *Internet Gateway Device (IGD)-Protokoll* von *UPnP* – sofern das *Gateway UPnP* unterstützt und aktiviert hat. Das *IGD*-Protokoll ist Bestandteil des *UPnP* und beschreibt die automatische Einrichtung von Portweiterleitungen und deren Verwaltung.

4.3 UPnP für Saros

Mit dem Verständnis, wie *NAT*-Gateways den *S5B*-Verbindungsaufbau zwischen Saros-Instanzen behindern können, soll Saros dazu befähigt werden, das Gateway mittels *UPnP/IGD* entsprechend zu konfigurieren. Saros soll am Gateway eine Portweiterleitung für den Port des *Socks5Proxies* (standardmäßig 7777) an die *IP* seines Rechners einrichten. Wollen mehrere Rechner im *LAN* Kontakt mit Rechnern von außerhalb des *LANs* aufnehmen können, müssen sie verschiedene Ports konfigurieren und diese jeweils an sich weiterleiten lassen.

Für die Integration von *UPnP* in Saros stellte sich die Frage, ob sich eine existierende Bibliothek hierfür anbietet oder ob eine eigene Implementierung notwendig ist. Die Wiederverwendung eines Modules kann verschiedene Vorteile mit sich bringen (wie in Kapitel 2 für das *ECF* evaluiert). Eine existierende Lösung spart Ressourcen (in diesem Falle Zeit), kann durch spezialisierte, erfahrene Anbieter eine höhere Qualität erreichen und wird bei aktiver Entwicklung weiter gepflegt.

Bei der Suche nach frei verfügbaren Bibliotheken für *UPnP*-Unterstützung in Java, fanden sich drei Kandidaten:

- **Cling**: aktive Entwicklung, LGPL, ~800KB Bibliothek
- **UPNPLib**: seit 2006 eingestellt, Domain verkauft (Gewinnspiel)
- **weupnp**: aktive Entwicklung, LGPL, ~50KB

Während *Cling* eine umfassende Implementierung von *UPnP* anstrebt, konzentriert sich *weupnp* auf das *UPnP* Protokoll für Portweiterleitungen (*IGD*).

> "*Weupnp* is a lightweight Java library, released under the LGPL licence, designed to implement the UPnP protocol to handle port mappings on Gateway Devices." [weUPnP]

Mit der Bevorzugung von zugeschnittenen und schlanken Lösungen, entschied ich mich dazu, die Bibliothek *weupnp* für Saros zu nutzen.

4.3.1 Die UPnP Bibliothek weupnp

Das Projekt *weupnp* wurde zur Zeit meiner Eignungsprüfung (Stand Mai 2011) aktiv von zwei Hauptentwicklern und zwei weiteren Personen mit Schreibrechten auf das Versionsverwaltungssystem entwickelt und wird auf der Plattform *Google Code* verwaltet und veröffentlicht. Die Projektanfänge lassen sich bis Ende 2008 zurückverfolgen und zeigen seitdem mittlere, konstante Aktivität (durchschnittlich zwei *Commits*[7] im Monat). Die Implementierung wurde laut Projektbeschreibung von der C-Bibliothek *miniupnp* inspiriert und baut somit auch auf dessen Erfahrungen auf.

Erste Tests mit *weupnp* zeigten, dass diese Bibliothek in meinem Heimnetzwerk schnell und verlässlich funktionierte und meinen Router erkannte. Weitere Tests mit anderen

[7] Übertragen von Quellcodeänderungen in ein Versionskontrollsystem

Betriebssystemen und virtuellen Maschinen zeigten jedoch, dass es bei Systemen mit mehreren Netzwerkadaptern dazu kommen kann, dass der Router nicht gefunden wurde. Beim *Debugging* auf diesen Systemen zeigte sich, dass das Senden der *UPnP*-Suchanfrage (*Discovery*) über Netzwerkadaptern geschehen konnte, die nicht mit dem Router in Verbindung standen. Die *UPnP*-Spezifikation empfiehlt jedoch, Suchanfragen über alle relevanten Adapter abzusetzen. (*UPnP-Lokalisierung*)

4.3.2 Code-Beiträge für weupnp

Im Folgenden beschreibe ich Änderungen und Erweiterungen, welche ich für das Projekt *weupnp* realisierte.

4.3.2.1 Multi-Adapter Support

Ich implementierte das Ermitteln der *IP*s aller Netzwerkadapter und das parallele Senden der Suchanfrage über diese. Anschließend konnte auch auf Systemen mit mehreren Netzwerkadaptern der Router via *UPnP* ermittelt werden. Diese Technik half ebenfalls bei der zuvor unzuverlässigen Erkennung der lokalen *IP*, über welche der Router kontaktiert wurde. Ein bisheriger *Workaround* konnte mit der Übernahme der *IP* des Netzwerkadapters ersetzt werden, über welchen die erfolgreiche Abfrage gesendet und Antwort erhalten wurde.

4.3.2.2 Lease Duration

Eine Portweiterleitung kann – sofern das Gateway dies unterstützt – mit einer Gültigkeitsdauer (*Lease Duration*) eingerichtet werden. Auf diese Weise werden temporäre Konfigurationen automatisch rückgängig gemacht, womit diese auch aufgeräumt werden, falls die Anwendung nicht mehr dazu kommt, selbstständig seine eingerichteten Portweiterleitungen aufzuräumen. *Lease Duration* wurde von *weupnp* nicht unterstützt, weshalb ich diese Funktionalität nachgerüstet habe.

4.3.2.3 Testanwendung erweitert und grafische Oberfläche eingeführt

Die Bibliothek *weupnp* enthält eine Klasse für eine Konsolenanwendung, welche eine Gatewaysuche durchführt und bei Erfolg das Erstellen und Entfernen von Portweiterleitungen testet und Statusmeldungen auf der Konsole ausgibt. Nach meinen Korrekturen und Erweiterungen an der Bibliothek, gab ich diese Anwendung mit einer Anleitung zum Starten an Bekannte zum Testen weiter. Es wurde dabei deutlich, wie schwer es für weniger erfahrene Computernutzer ist, diese Konsolenanwendung zu benutzen (Konsole öffnen, Java-Umgebungsvariablen konfiguriert?, Java-Anwendung über Kommandozeile starten, Konsolenausgabe in Zwischenablage kopieren).

Ich implementierte daher eine grafische Benutzeroberfläche mit einer Listenanzeige für die Ausgabe von Informationen des Tests und einem Knopf zum Kopieren der Ausgaben in die Zwischenablage (siehe Abbildung 16).

Weiterhin wurde die Erstellung des Anwendungsarchives (*JAR*-Paket – enthält alle zur Anwendung gehörenden Ressourcen) so konfiguriert, dass die Testanwendungsklasse als *Application entry point* definiert ist. Somit reicht es auf Systemen mit installiertem *Java Runtime*

Enviroment aus, dieses *JAR* auszuführen, um die grafische Testanwendung zu starten. Auf diese Weise kann die Anwendung deutlich einfacher und komfortabler gestartet und benutzt werden.

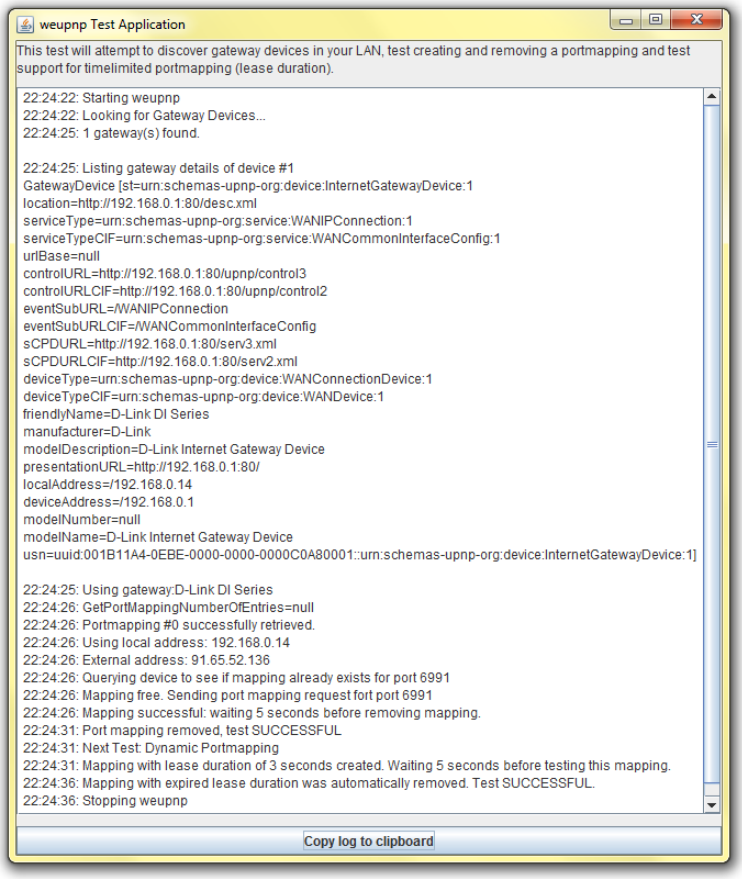

Abbildung 16: GUI der Testanwendung von weupnp

Die in der Testanwendung implementierten Tests habe ich abschließend erweitert für:

- Aktion: `GetPortMappingNumberOfEntries`
- Aktion: `GetGenericPortMappingEntry`
- Unterstützung von *Lease Duration*

Weiterhin wurde die Fehlerauswertung nach *UPnP*-Anfragen erweitert, so dass ein eventueller Fehlercode in der Antwort des kontaktierten Gerätes ausgewertet und als Rückgabewert an den Aufrufer der jeweiligen Methode von *weupnp* zurückgegeben wird.

4.3.3 Code Contribution

Nachdem ich, dank des Open Source Charakters von *weupnp*, die Arbeit und Erfahrung anderer für unsere Zwecke frei nutzen konnte, hatte ich natürlich auch Interesse daran, meine Änderungen wieder in das Projekt einfließen zu lassen. Um dies zu tun, gibt es direkt auf der Homepage des Projektes die Information, wie dies zu tun sei – mit Angabe der E-Mailadressen der Projektverantwortlichen.

Ich erstellte einen Patch und ein fertiges *JAR*-Archiv mit meinen Änderungen und Quellcode, stellte mich vor, beschrieb meine Änderungen und sandte dies per E-Mail an die Projektverantwortlichen. (Der E-Mailverkehr ist in Anhang 0 zu finden.) Bereits einen Tag später erhielt ich Antwort von einem der Projektgründer mit lobendem Dank und der Information, dass sie es genauer testen und begutachten werden. Drei Wochen später meldeten sie sich mit der Auskunft zurück, dass sie die Änderungen gerne einpflegen möchten, jedoch in zwei Teilen.

Ablauf:

1. alle Änderungen übernehmen, welche nicht die *API* verändern

Release durchführen

2. die weiteren Änderungen übernehmen, inklusive Änderungen an der *API*

Da ich mich bereit erklärte, diese Aufteilung der Änderungen zu übernehmen, wurden mir Schreibrechte für das Verwaltungssystem des Projektes eingerichtet. Somit ist es mir möglich, die implementierten Verbesserungen dem Projekt einfließen zu lassen.

4.4 Integration von UPnP in Saros

Nachdem die zugrundeliegende Technik für die *UPnP*-Funktionalität in Saros festgelegt wurde, geht es im Folgenden um die Integration.

Zu Beginn spezifiziere ich folgende Randbedingungen:

1. *UPnP* darf nicht ohne Erlaubnis Portweiterleitungen konfigurieren.
2. *UPnP* soll nicht ohne Erlaubnis regelmäßig aktiv sein.
3. Nach Aktivierung der *UPnP* Unterstützung soll Saros selbstständig die Portweiterleitung einrichten und aktuell halten.
4. Eingerichtete Portweiterleitungen sollen aufgehoben worden sein, wenn Saros beendet oder die Option deaktiviert wurde.

Für die *UPnP* Funktionalität wurde eine neue Klasse `UPnPManager` in Saros eingeführt, welche als Singleton über den `PicoContainer` von Saros verwaltet wird. Der `UPnPManager` kapselt die Verwendung der Bibliothek *weupnp* und stellt Methoden zur Verwaltung und Steuerung der Portweiterleitungen bereit. Alle anderen Klassen von Saros arbeiten somit nicht direkt mit *weupnp*.

Randbedingung 2 beachtend, erfolgt die Suche nach Gateways nicht regelmäßig zum Start von Saros, sondern nur, wenn *UPnP* bereits vom Anwender aktiviert wurde. Sofern nicht bereits eine Gatewaysuche durchgeführt wurde, geschieht dies auch beim Öffnen des *Saros-Configuration-Wizards* bzw. der *Advanced*-Seite des Eclipse-eigenen Konfigurationsmenüs für Saros. Auf diesen beiden Dialogen kann *UPnP* für Saros aktiviert werden (Abbildung 17), wozu die Suche nach einem Gateway notwendig ist, um Inhalt und Zustand der Konfigurationselemente (Checkbox, Auswahlliste) dem Ergebnis der Gatewaysuche anzupassen.

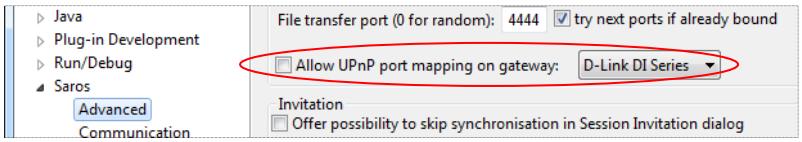

Abbildung 17: Konfigurationselemente für UPnP

Für die Einhaltung der Randbedingung 3, erfolgt das Einrichten oder Aktualisieren der Portweiterleitung bei aktivierter *UPnP*-Funktion sowohl beim Starten des *Socks5Proxys* als auch bei Änderung von dessen Port. Der Port für die Weiterleitung wird direkt vom laufenden *Socks5Proxy* abgefragt.

Zur Konfiguration des *Socks5Proxy* mit der öffentlichen *IP* wird, zusätzlich zu der in Kapitel 3.2.5.2 vorgestellten Technik, auch via *UPnP* die öffentliche *IP* des Gateways abgefragt und dem *Socks5Proxy* übergeben.

Um das in Randbedingung 4 geplante, ordentliche Aufräumen eingerichteter Portweiterleitungen zu realisieren, wurden drei Techniken implementiert.

Zum Ersten das aktive Entfernen einer alten Portweiterleitung beim Erstellen einer neuen, beim Deaktivieren der *UPnP*-Funktion in Saros und beim Beenden von Saros.

Zum Zweiten wird versucht die Portweiterleitung mit einer *Lease Duration* (Gültigkeitsdauer) zu konfigurieren, so dass sich diese nach einer Zeitspanne selbstständig löscht. Ein *Timer* im `UPnPManager` sorgt für die Neukonfiguration bei Ablauf der Gültigkeit. Verweigert das Gateway die Angabe einer *Lease Duration*, wird eine statische Portweiterleitung (ohne Gültigkeitsdauer) konfiguriert.

Zum Dritten speichert Saros sitzungsübergreifend die Daten der zuletzt erstellten Portweiterleitung und löscht diese Daten beim Entfernen dieser Portweiterleitung. Sollte beim Starten von Saros eine solche Konfiguration noch vorgefunden werden, konnte Saros die Portweiterleitung nicht aufheben, was in diesem Fall nachgeholt wird.

Bei der Implementierung war darauf zu achten, dass nach dem Einrichten einer Portweiterleitung, alle inaktiven *IBB*-Verbindungen getrennt werden. Dadurch wird der Neuaufbau eines *Bytestreams* erzwungen.

4.4.1 Intelligente UPnP-Werbung

Als Hilfsangebot von Saros für Anwender, soll es jene Situation erkennen, in denen das Aktivieren von *UPnP* dem Aufbau von *S5B* förderlich sein kann. In diesen Fällen soll der Anwender hierüber informiert werden. Diese Situation liegt dann vor, wenn ein Gateway die eingehenden Verbindungen blockiert. Ein solcher Umstand kann vermutet werden, wenn folgende Bedingungen vorliegen:

- Eine *IBB*-Verbindung wurde aufgrund einer eingehenden Verbindungsanfrage aufgebaut.
- Der *IBB*-Modus wurde nicht per Konfiguration erzwungen.
- Ein Gateway konnte im lokalen Netz ermittelt werden.
- Konnte *STUN* durchgeführt werden, wurde kein freier Netzzugang erkannt (sondern eine *Network Address Translation*).

Liegen diese Bedingungen vor, wird dem Anwender ein Hinweis gegeben, dass ein Gateway erkannt wurde, welches eventuell einen schnellen Datenaustausch verhindert und dass die *UPnP*-Funktion aktiviert werden sollte.

Dieser Test wird nur einmal durchgeführt, um den Anwender nicht zu belästigen.

4.5 Fazit

Mit Hilfe der Bibliothek *weupnp* ist es gelungen, Saros die Fähigkeit zu verleihen, Portweiterleitungen an einem *UPnP*-fähigen Gateway zu konfigurieren.

Befindet sich ein Rechner mit Saros hinter einem Gateway, ist es für Verbindungsanfragen von außen erst nach der Einrichtung einer Portweiterleitung empfänglich. In dieser Netzwerkkonfigurationen war es bisher nur möglich, den langsameren *IBB* zum Datentransfer zu nutzen (vom rückwärtigen Verbindungsaufbau abgesehen). Mit einer Portweiterleitung ist der Rechner und damit Saros nun auch für Verbindungsanfragen aus dem Internet empfänglich und somit in der Lage *S5B* zu nutzen.

Es muss jedoch bedacht werden, dass das *UPnP*-Feature in Saros nicht alle *Bytestream*-Probleme löst, sondern nur in folgenden Fällen:

- das Gateway ist *UPnP*-fähig
- *UPnP* Funktionalität im Gateway ist aktiviert, sowohl die *Discovery*- als auch Portweiterleitungsfunktion – dies kann hersteller- und modellabhängig sein
- *UPnP*-Unterstützung des Gateways ist funktionstüchtig und korrekt – alte Geräte können unausgereift und ein *Firmwareupgrade* notwendig sein (laut Kompatibilitätsliste *miniupnp*)
- eventuell installierte Firewall behindert weder *UPnP*-Datenverkehr noch eingehende Verbindungen

4.5.1 Test im praktischen Einsatz

Beim Akzeptanztest der ersten Saros-Version mit *UPnP*-Unterstützung lautete es in der zusammenfassenden E-Mail (1.7.2011, E. Rosen):

> „It's definitely more fun to use Saros with Socks5 than IBB, thanks to Björn UPnP works fine to overcome NAT"

Um die *UPnP*-Funktionalität in Saros auch selbst im realen Einsatz zu testen, nutzte ich eine Saros-Instanz („SarosFU") im FU-Netz und eine Saros-Instanz („SarosHome") auf meinem Heimrechner, welcher sich in einem *LAN* mit *UPnP*-fähigem *DSL*-Router als Internetgateway befindet. SarosHome läuft mit einem *Socks5Proxy* auf einem noch nicht weitergeleiteten Port. Von der Universität aus steuerte ich meinen Heimrechner fern (mittels *VNC*[8]), konnte so beide Saros-Instanzen bedienen und mit dem *XMPP*-Netzwerk verbinden. Ich nutzte anschließend die „Test data transfer connection" Funktion von Saros, um Testdaten über einen aufzubauenden *Bytestream* zu senden. Ich achtete im *Log* darauf, dass kein rückwärtiger *S5B* aufgebaut und benutzt wurde.

Test 1: SarosHome ohne Portweiterleitung

Ergebnis: Ein *Socks5 Bytestream* **konnte nicht** initiiert werden. Mit Rückfall auf *IBB* wurde der Datenverbindungstest abschließend erfolgreich durchgeführt.

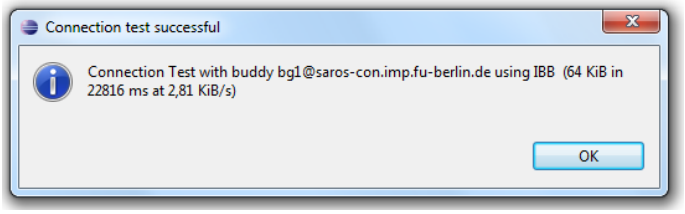

Abbildung 18: Bytestream Verbindungstest meldet IBB Verbindung

[8] VNC (Virtual Network Computing) Fernwartungssoftware, erlaubt die Steuerung von Tastatur und Maus sowie Ansicht der Bildschirmausgabe eines anderen Rechners über ein Netzwerk

Test 2: Ich aktivierte *UPnP* auf SarosHome und führte den Datenverbindungstest ohne Neustart erneut aus.

Ergebnis: Im Log meldete Saros, dass die Portweiterleitung für den konfigurierten *Socks5Proxy*-Port eingerichtet wurde:

```
(UPnPManager.java:251) Port mapping for port 689 added for
192.168.0.11 on D-Link DI Series
```

Der anschließend durchgeführte Übertragungstest **konnte sofort** einen *Socks5 Bytestream* für die Übertragung der Testdaten aufbauen und nutzen (Abbildung 19).

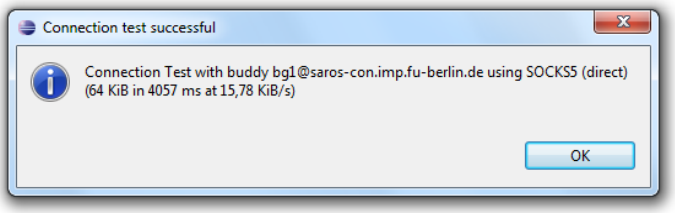

Abbildung 19: Bytestream Verbindungstest meldet S5B-Verbindung

Test 3: Ich startete beiden Instanzen neu, konfigurierte neue Ports (*UPnP* weiterhin aktiv) auf SarosHome und stellte die Verbindungen zum *XMPP*-Netzwerk her. Anschließend führte ich den Datenverbindungstest durch.

Ergebnis: SarosHome konfigurierte die Portweiterleitung und der Datenverbindungstest konnte eine *S5B*-Verbindung aufbauen und nutzen. (wie Abbildung 19)

Zusammenfassung

Durch abschließende Tests konnte ein erfolgreicher Einsatz der *UPnP*-Funktionalität bestätigt werden. Für Anwender mit den im vorherigen Kapitel beschriebenen Netzwerkeinschränkungen und mit verfügbarem *UPnP*-kompatiblen Gateway bringt diese neue Funktionalität eine Verbesserung des Datenverbindungsaufbaues (*S5B* statt *IBB*) und damit eine deutlich höhere Geschwindigkeit beim Übertragen von Daten, woraus schließlich eine flüssige und damit anwenderfreundliche Saros-Sitzung resultiert.

5 Testbarkeit der Netzwerkschicht

Dieses Kapitel stellt zuerst Begriff und Inhalt der Testbarkeit vor, untersucht anschließend die Netzwerkschicht von Saros auf die Eigenschaften der Testbarkeit und gibt Ansätze für Verbesserungen in dieser Hinsicht.

5.1 Einführung Testbarkeit

Testen ist in der Softwareentwicklung ein zentraler Prozess der Qualitätssicherung zum Messen und Verbessern der Qualität von Software. Durch das Testen von Software kann Versagen entdeckt und dadurch zugrundeliegende Defekte korrigiert werden, was die Qualität der Software verbessert. Wie kann nun das Testen verbessert werden? Ein naheliegender Schritt ist es, die Testbarkeit der Software und damit die erreichbare Effizienz von Tests zu verbessern.

Testbarkeit kommt nicht mit einer gefestigten Definition daher, sondern wird von Autor zu Autor mehr oder weniger konkret formuliert. Es folgen drei Definitionen mit steigender Konkretisierung.

Definitionen „Testbarkeit“:
„Das Maß, um das ein System oder eine Komponente die Festlegung von Testkriterien unterstützt und die Performanz von Tests zur Feststellung, ob diese Testkriterien erfüllt wurden.“ [IEE90] „Testbarkeit ist der Grad, zu dem ein Softwareartifakt in einem bestimmten Kontext den Testvorgang ermöglicht.“ [Jun00] Pettichord [Pet02] „…defines testability as visibility and control. Visibility is our ability to observe the states, outputs, resource usage and other side effects of the software under test. Control is our ability to apply inputs to the software under test or place it in specified states.”

Dabei weist Jungmayr [Jun04] darauf hin, dass nicht nur die Testbarkeit von Softwarekomponenten betrachtet werden sollte, sondern auch die der Anforderungen, der Architektur und des Entwurfes. Im Folgenden werde ich mich auf die Testbarkeit von Softwarekomponenten konzentrieren, da diese das Ziel der Untersuchung dieses Abschnittes darstellen.

Nach Pettichord [Pet02] und Jungmayr [Jun04] lässt sich Testbarkeit von Softwarekomponenten anhand verschiedener Eigenschaften bewerten, welche in den folgenden Unterkapiteln aufgeführt und näher erläutert werden.

5.1.1 Beobachtbarkeit (Visibility)

Nur was vom Testgegenstand beobachtet werden kann, kann auch getestet werden. Für den Tester nicht einsehbare Eigenschaften können nicht eingeschätzt und somit nicht bewertet werden.

Die Beobachtbarkeit beschreibt die Möglichkeiten, Ausgaben, interne Zustände und Seiteneffekte der Software beobachten zu können. Weiterhin weist Pettichord [Pet02] darauf hin, dass auch der Zugriff auf Quellcode, Entwurfsdokumente und Änderungsprotokolle zur Beobachtbarkeit gehören (*Understandability*). Nicht nur der Zugriff ist dafür notwendig, sondern auch ein Mitarbeiter des Testteams, welcher mit diesen durchaus technischen Dokumenten umzugehen weiß.

Pettichord [Pet01], [Pet02] nennt eine Reihe von Methoden, um die Beobachtbarkeit zu verbessern:

Mit **Diagnostiken** werden Techniken bezeichnet, welche das Verhalten der Software besser beobachtbar und dadurch auch Versagen identifizierbar zu machen, sobald diese auftreten. Ein verbreitetes Beispiel hierfür sind **Assertionen.** Dies sind Anweisungen, welche Daten daraufhin prüfen, ob Annahmen eines darauf arbeitenden Programmabschnitts über den Inhalt der Daten zutreffen. Werden während des Programmlaufes Annahmen verletzt, wird ein *Error* ausgelöst und eine konfigurierbare Nachricht ausgegeben. Der weitere Programmablauf wird damit – im Gegensatz zu *Exceptions* – nicht gestört. Ungültige Daten können somit erkannt werden, bevor ein Versagen auftritt – auch wenn das Versagen nicht beobachtbar ist.

Event trigger können eingesetzt werden, um Beginn und Ende von Operationen nach außen hin zu signalisieren.

Integritätsprüfungen erlauben es, fehlerhafte Daten zu erkennen, welche dem Anwender sonst verborgen bleiben (Datenbank- und Speicherintegrität).

Weiterhin bieten spezielle **Zugriffsmethoden** die Möglichkeit, Daten oder Zustände abzufragen, welche sonst im Verborgenen bleiben. Diese **Testschnittstellen** können neben der Benutzerschnittstelle einen zusätzlichen Einblick in das System erlauben. Hier sind Sicherheitsaspekte jedoch sehr ernst zu nehmen, um nicht außerhalb des Testszenarios Anwendern oder Angreifern einen unrechtmäßigen Zugriff auf Daten zu ermöglichen.

Als **Monitore** werden Werkzeuge bezeichnet, welche Einblick in die internen Abläufe einer laufenden Anwendung erlauben. Ein Beispiel hierfür sind **Ereignisprotokolle** (*event logs*), welche wichtige Operationen, Ereignisse (z.B. Entscheidungen, Fehler, unerwartete Ereignisse) protokollieren. Hierbei sollten Zeitstempel und die Quelle der Ausgabe ebenfalls protokolliert werden, um den Ablauf von Ereignissen und aufgetretene Probleme nachvollziehbar zu gestalten. Beim Protokollieren ist darauf zu achten, dass zwar alles eventuell Relevante aufgezeichnet wird, dennoch das erzeugte Protokoll noch überschaubar genug sein sollte, um es aufzeichnen und auswerten zu können.

5.1.2 Kontrollierbarkeit (Control)

Unter Kontrollierbarkeit werden die Möglichkeiten der Tester bezeichnet, der zu untersuchenden Software im Test Eingaben zu injizieren und interne Zustände auszulösen. Auf diese Weise können Szenarien getestet werden, welche nur schwierig oder langwidrig vom Anwender auf dem üblichen Weg nachgestellt werden könnten.

Aus dem gleichen Grund sind automatisierte Tests besser anwendbar, da diese einfacher ihre Vorbedingungen aufbauen können und nicht auf Eingabebeschränkungen oder -umwege angewiesen sind.

Exception seeding ermöglicht es dem Tester, auf unterer Ebene (im Kern der Implementierung) Fehler zu simulieren und damit deren Fehlerbehandlung zu testen.

Fault injection erzeugt Ausnahmesituationen (wie z.B. „Out of disk space" oder „Out of Memory"), um die zu testende Software in diesen testen zu können.

Test points erlauben es, im zu untersuchenden System interne Daten anzusehen und auch einzugeben. Damit dienen sie einerseits der Beobachtbarkeit, andererseits auch der Kontrollierbarkeit.

5.1.3 Komplexität

Wird Softwarekomplexität aus dem Blickwinkel des Softwareentwicklers betrachtet – zum Entwerfen von Tests interessiert eben diese Sichtweise – versteht man unter Softwarekomplexität eine Eigenschaft von Software, welche die Schwierigkeit angibt, mit welcher ein Softwareentwickler die Software verstehen und mit ihr arbeiten kann. Als Metriken hierfür bieten sich die zyklomatische Komplexität (McCabe Metrik), die Schachtelungstiefe der Kontrollstrukturen sowie die Anzahl von Schnittstellenparametern an. (vgl. [Wal01])

Die zyklomatische Komplexität (*cyclomatic complexity, CC*) eines betrachteten Moduls gibt die Anzahl unabhängiger Pfade im Programmfluss an, welche durch bedingte Verzweigungen entstehen. Die Idee hinter dieser Metrik ist: Je geringer die Verzweigungen im Ablauf, desto verständlicher ist er. Als Kritik ist jedoch anzumerken, dass der Wert den Sachverhalt der Verständlichkeit vereinfacht. Eine `switch`-Anweisung mit 10 Fällen ist beispielsweise leichter verständlich als ein verschachtelter Aufruf über nur 5 Ebenen. Lorenz und Kidd [Lor94] weisen darauf hin, dass diese Metrik für objektorientierte Software zu kurz greift, da hier die Komplexität auch durch andere Strukturmerkmale beeinflusst wird, als nur durch Verzweigungen. Jedoch gibt die Metrik einen Hinweis auf die Verschachtelung eines Moduls und wird daher (zumindest als Anhaltspunkt) genutzt.

Umfangreiche Schnittstellen erhöhen den Testaufwand, da für gründliches Testen alle Parameterkonstellationen getestet werden müssen.

> „Umfangreiche Schnittstellen mit vielen Parametern, tiefe Klassenhierarchien sowie lange Kaskaden von Methodenaufrufen erschweren die Testfalldefinition, Testvorbereitung und Fehlersuche." [Jun05]

Das Ziel liegt daher darin, hohe Schnittstellen-Komplexität zu vermeiden und nur geringe Vererbungshierarchien zu nutzen.

5.1.4 Abhängigkeiten

Der Grad, zu welchem das betrachtete System innere Abhängigkeiten enthält, hat direkten Einfluss auf die Möglichkeiten, Komponenten isoliert zu testen. Bei hohen Abhängigkeiten können in diesen Bereichen keine kurzen, lokalen Modultests entwickelt werden, da abhängige Module zusätzlich mit einfließen.

Mit einem steigenden Maß an Abhängigkeiten in einem Softwaresystem wird dieses schwerer zu erfassen und zu verstehen, weshalb die zuvor genannte Komplexität auch von dem Grad innerer Abhängigkeiten abhängt.

5.1.5 Trennung der Belange

Werden verschiedene Belange (Angelegenheiten) in einem Modul integriert, erschwert auch dies den Komponententest, da sie durch Kopplungen verschiedene Funktionalitäten beinhalten müssten.

Bei Überlegungen zur Architektur eines Systems sollte immer beachtet werden, thematisch unzusammenhängende Konzepte auch getrennt zu implementieren und voneinander abzugrenzen. Dies ermöglicht die spätere Konzeption von kurzen, lokalen Modultests, welche nicht verschiedene unabhängige Belange testen müssen.

Bei Anwendung des *Model View Presenter* (*MVP*)-Musters können beispielsweise gut modultestbare Architekturen erreicht werden. Da Anwendungslogik und *User Interface* (*UI*)-Logik getrennt sind, können Aspekte der Anwendungslogik in Modultests getestet werden, ohne Anhängigkeiten des *UI* zu begegnen, was automatisierte Tests deutlich erschweren würde.

5.2 Testbarkeit im Entwicklungsprozess

Die Grundpfeiler der Testbarkeit stellen, wie im vorherigen Kapitel vorgestellt, sehr grundlegende Eigenschaften von Softwaresystemen dar. Eine frühzeitige Beachtung von Testbarkeitskriterien ist daher von Vorteil und sollte bereits beim Architekturentwurf durchgeführt werden. Nur so lässt sich gute Testbarkeit bereits in der Architektur verankern. Einem fertigen Produkt nachträglich gute Testbarkeitskriterien nachzurüsten, kann grundlegende Änderungen an der Architektur erfordern und ist somit – wenn denn möglich – aufwändig und teuer. (vgl. [McC98])

Die Zusammenhänge und Intentionen innerhalb des Entwicklungsprozesses von Setzung der Projektziele bis hin zur Implementierung, sind in der folgenden Abbildung von Jungmayr anschaulich dargestellt. [Jun06]

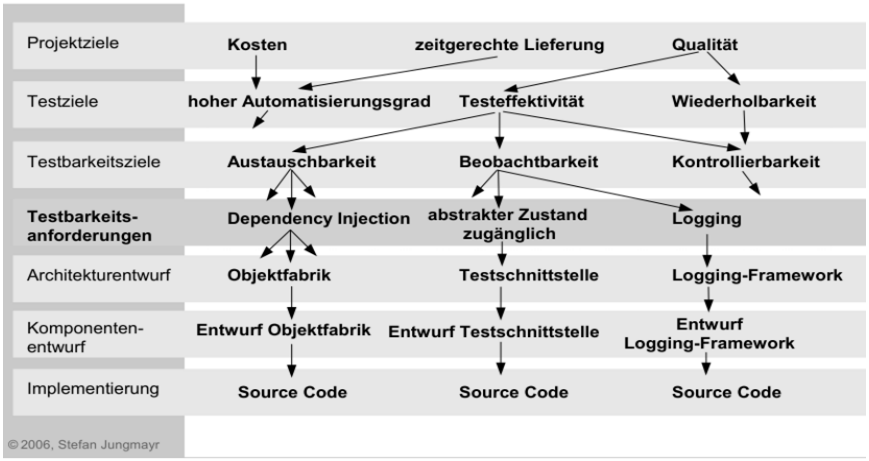

Abbildung 20: Verfolgbarkeit der Testbarkeitsanforderungen [Jun06]

Welche weiteren generellen Aspekte und Methoden sollten während der Softwareentwicklung für eine gute Testbarkeit geprüft oder vorgenommen werden?

5.2.1 Einsatz testgetriebener Entwicklung

Das Prinzip der testgetriebenen Entwicklung schreibt das Verfassen von Tests für eine neue Funktionalität noch vor deren Implementierung vor. Auf diese Weise wird nicht nur eine gute Testabdeckung erreicht, sondern auch eine gute Testbarkeit, da noch vor der Implementierung über Tests und deren Machbarkeit nachgedacht werden muss. Dieses Prinzip sollte – wenn nicht schon von Anfang an vorgeschrieben – eingeführt werden.

5.2.2 Singletons vermeiden

Singletons im zu testenden System sind beim Entwickeln von Test hinderlich, da wegen ihrer Natur der global verfügbaren und einfachen Existenz, immer dieselbe Instanz mit getestet wird und so Testfälle nicht vollständig unabhängig und losgelöst voneinander ablaufen.

Weiterhin kann die Nutzung von Singletons die Kopplung innerhalb der Software erhöhen, da alle Klassen die auf das Singleton zugreifen, von diesem abhängig sind.

5.2.3 Wiederholbarkeit

Die Funktionalität der Software muss eine Wiederholbarkeit garantieren. Identische Tests mit dem gleichen Input müssen das gleiche Ergebnis liefern. Implizite Eingaben (z.B. zeitliche Abhängigkeiten) dürfen nicht die Funktionalität ändern. (vgl. [Jun05])

5.2.4 Testwerkzeuge unterstützen

Testwerkzeuge ermöglichen und fördern das automatisierte und wiederholte Durchführen von Tests. Das langwierige manuelle Ausführen von Test kann damit vermieden werden, was Arbeitskräfte frei stellt, Tests schneller abarbeiten lässt und Testfehler aufgrund menschlichen Versagens ausschließt. Testwerkzeuge können somit effizienteres Testen ermöglichen.

Für den erfolgreichen Einsatz von Testwerkzeugen sind jedoch verschiedene Ansprüche an die zu testende Software zu stellen und benötigen daher möglicherweise zuvor Anpassungen:

- Die Software muss automatisierbar sein und die Zusammenarbeit mit dem Testframework unterstützen. Relevante Faktoren können sein:
  - Programmiersprache
  - Kompatibilität, vorhandene Schnittstellen, Systemarchitektur
  - Umgebung (Betriebssystem, Entwicklungsumgebung)
- Um eine *GUI* testbar zu gestaltet, müssen Ansprüche an dessen Struktur und Beschaffenheit eingehalten werden, Bedienelemente müssen eindeutig identifizierbar sein und Kommandos von außen akzeptieren werden.
- Eine getrennte *UI*- und Businesslogik kann wesentlich sein, falls das Testframework keine *UI*-Tests unterstützt. Des Weiteren ist sie vorteilhaft, um feingranulare Modultests entwickeln zu können.

5.3 Testbarkeit der Netzwerkschicht von Saros

Die Netzwerkschicht von Saros wird nun auf die vorgestellten Faktoren der Testbarkeit hin untersucht. Dies soll sowohl die bereits integrierten Mechanismen zur Testbarkeit als auch noch offenes Potential für dessen Verbesserung aufzeigen. Eine solche von mir durchgeführte Verbesserung werde ich in Kapitel 5.4 beschreiben.

Beobachtbarkeit

5.3.1 Assertionen

Assertionen werden in der Netzwerkschicht nur sehr sporadisch eingesetzt. So finden sich nur 12 explizite `assert`-Anweisungen in 848 Methoden (in 69 Klassen) der Netzwerkschicht[9].
Es kann keine allgemeingültige Aussage getroffen werden, in wie viel Prozent der Methoden Assertionen eingesetzt werden sollen. Werden alle Daten einer Berechnung regulär auf Einhaltung aller Annahmen getestet (*IF-, CASE*-Anweisungen), ist eine Assertion nicht zwingend einzusetzen. Jedoch lassen sich schon bei einer kurzen Inspektion des Quellcodes von Saros Anweisungen finden, welche ungeprüfte Annahmen über die Eingangsparameter oder Klassenvariablen treffen. Beispielsweise nimmt die folgende Methode des `DataTransferManager` an, dass `transferData` und `input` gültige Werte enthält.

```
public void sendData(TransferDescription transferData, byte[] input,
        SubMonitor progress) throws IOException,

        JID recipient = transferData.recipient;
…

…
            long sizeUncompressed = input.length;
```

Listing 7: Auszug aus der Methode sendData, zeigt Annahmen zu Werten

[9] Stand: 19.7.2011

Mit Assertionen kann während des Programmlaufes auf unerwartete Daten oder Zustände aufmerksam gemacht werden, ohne den weiteren Programmfluss zu unterbrechen. Diese Technik einzusetzen ist empfehlenswert und wird in den *Coding Conventions* von Saros auch bereits empfohlen. Dies bei allen existierenden Methoden der Netzwerkschicht nachzuholen, bedarf es eines hohen Aufwandes, da die Einarbeitung in fremden Code und das Prüfen auf Notwendigkeit für jede Funktion erforderlich ist. Es kann sich jedoch auf kritische Bereiche (mit aus Erfahrung erhöhter Versagensrate) und Kernbereiche (häufig ausgeführter Code) konzentriert werden.

5.3.2 Event triggers

Diese Technik wird nicht eingesetzt, jedoch wird mit Einsatz von Ereignisprotokollierung (Kapitel 5.3.5) ein ähnlicher Effekt erzielt – mit der Ausgabe von spezifizierten Meldungen im Protokoll.

5.3.3 Integritätsprüfungen

Die Prüfung einer Datenbankintegrität ist nicht notwendig, da keine Datenbank eingesetzt wird.

Im Gegensatz zu Programmiersprachen wie *C* und *C++,* hindert die für Saros eingesetzte Sprache *Java* den Programmierer an speichermanipulierenden Verfahren. Java verhindert das Überschreiben von Speicher und das Korrumpieren von Daten und fördert so die Daten- und Codeintegrität.

> „The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays. This allows subscript checking to be performed." [Ora11]

5.3.4 Testschnittstellen

Saros bietet in der Netzwerkschicht keine Schnittstellen für das Beobachten (oder das Manipulieren) von Daten, jedoch wäre dies ein wünschenswerter Ansatz für das Verfolgen und Verstehen von internen Abläufen in der Netzwerkschicht von Saros. Beispielsweise ließe sich so der Aufbau von *Bytestreams* (Verfolgung des Protokolls des *Bytestream*-Aufbaues) oder die parallele Nutzung von Netzwerkressourcen verfolgen und Szenarien analysieren.

Smack bietet für das Verfolgen des Paketverkehrs auf dessen Ebene bereits eine Möglichkeit in Form eines Werkzeuges (siehe Abbildung 10 und Abbildung 11). Es stellt ein- und ausgehende Pakete chronologisch dar und ermöglicht auch die Darstellung der Paketinhalte, was mir oft hilfreich war, Grundlagen und Probleme im *XMPP*-Netzwerkverkehr zu verstehen.

5.3.5 Ereignisprotokollierung

Saros nutzt Ereignisprotokollierung (*Logging*) ausführlich, sowohl in der Netzwerkschicht als auch in anderen Bereichen von Saros. In der Netzwerkschicht enthält statistisch gesehen

bald jede dritte Methode eine Log-Ausgabe (etwa 300 Log-Ausgaben, 848 Methoden der Netzwerkschicht).

Markante Aktionen und Ereignisse werden im Protokoll (*Log*) aufgeführt, was das Verfolgen von Vorgängen und die Rückverfolgung von Fehlern erleichtert. Saros bietet dem Entwickler eine einfache Nutzung der Protokollierung und dies soll auch weiterhin ausgiebig eingesetzt werden.

5.3.6 Kontrollierbarkeit

Keiner der Ansätze für eine bessere Kontrollierbarkeit (*Exception seeding, Fault injection*, *Test points)* wird gegenwärtig von Saros genutzt, was das Testen von schwer nachstellbaren Szenarien verhindert. Dabei könnten *Exception seeding* und *Fault injection* für genau diesen Anwendungszweck eingesetzt werden. Der Einsatz dieser Techniken in kritischen Codebereichen (nicht nur der Netzwerkschicht) ist empfehlenswert. Die Erreichbarkeit von Code während der Ausführung kann so erhöht werden und damit die Abdeckung des Codes mittels Tests (*Code Coverage*).

Als kritische Bereiche sind jene Methoden zu sehen, welche direkt oder indirekt mit dem Versand von Nachrichten oder Daten bzw. mit dem Empfangen und Verarbeiten von eingehenden Paketen zu tun haben. Ein Ansatz hierbei ist das Auslösen von *Exceptions* während des Datentransfers und der Einsatz von *Fault injection* zum Abbruch des *Bytestreams*, Trennen der *XMPP*-Verbindung oder der Netzwerkverbindung. Letzteres kann durch das Blockieren aller Pakete von *Smack* simuliert werden.

Generelle Maßnahmen für gute Testbarkeit

5.3.7 Komplexität

Die Komplexität der Netzwerkschicht – aus dem Blickwinkel des Softwareentwicklers – liegt zum einen an dessen **Umfang**. Das Einarbeiten und Verstehen ist bei einer Größe von 9767 *ELOC*[10] in 69 Klassen zeitaufwändig und nicht trivial – ebenso wie das Verstehen des Zusammenspiels der Netzwerkklassen als auch der Funktionsweise von *Smack*. Dieses Wissen geht zum Teil mit den Studenten, die sich eingearbeitet haben, verloren. Für das Entwickeln von Tests existierender Funktionalität, liegt daher jeweils nur ein beschränktes Zeitfenster zur Verfügung.

Die **Kaskadierungen von Methodenaufrufen** hindert zum anderen das Verständnis des Netzwerkcodes. Beispielsweise ist das Versenden von Aktivitäten durch eine Vielzahl von Klassen realisiert, welche Aktivitäten modellieren, erzeugen, zusammenfassen, kodieren und versenden. Möchte man auch auf unterster Ebene der Netzwerkschicht Verständnis sammeln, ist die Auseinandersetzung mit *Smack* und dessen Interna notwendig. Innerhalb

[10] ELOC steht für „estimated lines of code“, eine um Kommentar-, Import- und Leer-Zeilen korrigierte Codezeilenabschätzung, gemessen mittels Analyse-Werkzeug „STAN“, Stand 21.7.2011

der Netzwerkklassen von Saros finden sich Verschachtelungstiefen von bis zu 6 Ebenen. Modularisierung fördert zwar das Strukturieren und damit das Begreifen von komplexen Systemen, jedoch kann sich somit das Durchführen einer Funktionalität leicht über eine Vielzahl von Modulen erstrecken, was die Nachvollziehbarkeit erschwert.

Die maximale **Ableitungstiefe** liegt mit 4 Ebenen im übersichtlichen und nachvollziehbaren Rahmen.

Die durchschnittliche **Anzahl von Methodenparametern** liegt bei etwa 2,5. Dies ist ein sehr übersichtlicher Wert, jedoch sollten jene Methoden überarbeitet werden, welche hier mit 7 oder 9 Parametern herausstechen.

Der durchschnittliche Wert der **zyklomatischen Komplexität** (McCabe Metrik) für die Netzwerkschicht liegt bei 2,02 (Maximum 25). McCabe gab einen Wert von 10 als Obergrenze an, ab welchem eine Komplexitätsreduzierung ratsam ist. Somit ist es empfehlenswert, jene Module von Saros mit einem hohen Wert zu untersuchen und gegebenenfalls zu refaktorieren.

Die in diesem Abschnitt genannten Werte wurden mit Hilfe des Eclipse Plugins „Metrics2“ ermittelt (Abbildung 21). Nach einer Quelltextanalyse bietet es Maßzahlen zu einer Reihe von Softwaremetriken an.

Metrics - src - Depth of Inheritance Tree (avg/max per type)

Metric	Total	Mean	Std. Dev.	Maxim...	Resource causing Maximum
McCabe Cyclomatic Complexity (avg/max per		2,044	2,293	28	/Saros/src/de/fu_berlin/inf/dpp/project/internal/SarosSession.java
Number of Parameters (avg/max per method)		0,99	1,194	16	/Saros/src/de/fu_berlin/inf/dpp/net/internal/XMPPTransmitter.java
Nested Block Depth (avg/max per method)		1,426	0,958	9	/Saros/src/de/fu_berlin/inf/dpp/videosharing/encode/tools/QuantizeFilte...
Afferent Coupling (avg/max per packageFragn		18,604	40,056	253	/Saros/src/de/fu_berlin/inf/dpp
Efferent Coupling (avg/max per packageFragm		4,689	4,991	29	/Saros/src/de/fu_berlin/inf/dpp/net/internal
Instability (avg/max per packageFragment)		0,375	0,254	1	/Saros/src/de/fu_berlin/inf/dpp/propertyTesters
Abstractness (avg/max per packageFragment)		0,163	0,252	1	/Saros/src/de/fu_berlin/inf/dpp/communication/muc/session/events
Normalized Distance (avg/max per packageFra		0,487	0,282	1	/Saros/src/de/fu_berlin/inf/dpp/ui/widgets/explanation/exceptions
Depth of Inheritance Tree (avg/max per type)		1,838	1,417	8	/Saros/src/de/fu_berlin/inf/dpp/ui/widgets/explanation/SimpleExplanatio...
Weighted methods per Class (avg/max per typ	9652	10,796	16,836	172	/Saros/src/de/fu_berlin/inf/dpp/editor/EditorManager.java
Number of Children (avg/max per type)	413	0,462	1,915	32	/Saros/src/de/fu_berlin/inf/dpp/project/AbstractSarosSessionListener.java
Number of Overridden Methods (avg/max per	438	0,49	1,027	11	/Saros/src/de/fu_berlin/inf/dpp/util/UncloseableInputStream.java
Lack of Cohesion of Methods (avg/max per typ		0,232	0,353	1,667	/Saros/src/de/fu_berlin/inf/dpp/ui/preferencePages/FeedbackPreference...
Number of Attributes (avg/max per type)	2074	2,32	3,715	29	/Saros/src/de/fu_berlin/inf/dpp/editor/EditorManager.java
Number of Static Attributes (avg/max per type	679	0,76	3,665	73	/Saros/src/de/fu_berlin/inf/dpp/preferences/PreferenceConstants.java
Number of Methods (avg/max per type)	4363	4,88	6,899	103	/Saros/src/de/fu_berlin/inf/dpp/feedback/SessionStatistic.java
Number of Static Methods (avg/max per type)	359	0,402	2,125	48	/Saros/src/de/fu_berlin/inf/dpp/util/Utils.java

Abbildung 21: Auswertung von Softwaremetriken mittels "Metrics2"

5.3.8 Abhängigkeiten

Zur Einschätzung von Abhängigkeiten in der Netzwerkschicht nutze ich die Metrik *ACD*.

Definitionen:

> Die mittlere *Component Dependency* (*ACD*) [Lak96] ist die mittlere Anzahl von Komponenten, von denen eine jede Komponente direkt oder indirekt abhängt, inklusive sich selber.

Diese Metrik lässt somit auf den Kopplungsgrad innerhalb einer Komponente schließen. Eine Komponente kann je nach Betrachtung ein Modul, Paket oder eine Klasse sein.

Der relative *ACD* (für Anzahl der Komponenten) wurde für die Netzwerkschicht von Saros mittels *STAN*[11] für Klassen von Paketen der Netzwerkklasse ermittelt.

Paket	rel. ACD
net	10,29%
net.UPnP	33,3%
net.business	1,39%
net.internal	27,09%

Tabelle 8: *relative Average Component Dependency* von Paketen der Netzwerkschicht

Die durchschnittliche Komponentenabhängigkeit ist in den Paketen `net` und `net.internal` leicht erhöht, wird aber allgemein als unkritisch eingestuft. Nach Jungmayr [Jun04] gibt es kein festes Limit, um gute von schlechten Architekturen zu unterscheiden. Jedoch gibt er grob einen Wert von 50% als Schwelle an (eine Komponente hängt im Schnitt von jeder zweiten ab), ab welchem von einer schlechten Abhängigkeitsstruktur ausgegangen werden muss.

Diese Metrik sollte daher als Anhaltspunkt gesehen werden, um eine eigene Analyse durchzuführen. So weist das Paket `net.UPnP` zwar einen hohen *ACD*-Wert auf, was nach Betrachtung der Abhängigkeiten mittels *STAN* jedoch aufgrund der Abhängigkeit von den einzigen zwei Klassen zum einzigen Interface so beabsichtigt ist, dies relativiert den Wert.

Eine Strukturanalyse mittels *STAN* zeigt außerdem Zyklen in den Abhängigkeiten schon auf Paketebene (Abbildung 22). Martin kritisiert in [Mar03] solche Abhängigkeiten: "The dependencies between packages must not form cycles." Ein *ACD*-Wert von 82,7% auf Paketebene der Netzwerkschicht trägt diesem Umstand Rechnung. Diese Abhängigkeiten erfordern eine Refaktorierung.

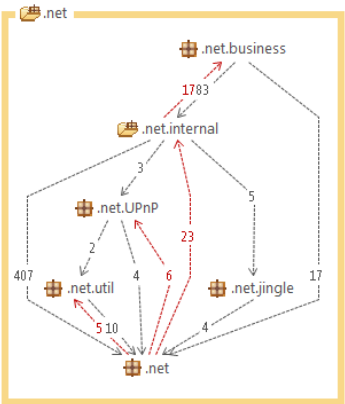

Abbildung 22: STAN weist zyklische Abhängigkeiten zwischen Paketen nach

[11] STAN, **St**rukturanalyse für Java, Eclipse Plugin zur Messung von Softwareeigenschaften: http://stan4j.com

5.3.9 Trennung der Belange

In der Netzwerkschicht sind Abhängigkeiten zu Klassen zu finden, welche nicht zu dieser Schicht gehören, was auf mangelnde Trennung von Belangen schließen lässt. Es sind vereinzelt Kopplungen zur *GUI* zu finden, was insofern hinderlich ist, dass beispielsweise *JUnit*-Tests keine blockierenden Nachrichtenfenster handhaben können. Aber auch Verschmelzungen der Netzwerkschicht von Saros zu dessen Plugin-Klasse sind zu finden, wodurch die Netzwerkfunktionalität an eine komplette Saros-Instanz gebunden ist, was die Entwicklung von Modultests der Netzwerkschicht behindert. Eine Verbesserung hierfür wird in Kapitel 5.4 beschrieben.

5.3.10 Testgetriebene Entwicklung

Die Möglichkeiten zur Entwicklung von Tests wurden und werden durch mehrere Arbeiten in der Arbeitsgruppe Softwareengineering verbessert ([Szü10], Cordes, Chen, Rossbach). Diese Arbeiten zielen darauf ab, Testframeworks bereitzustellen und zu erweitern sowie das Entwickeln von Test zu erleichtern. Durch diese Maßnahmen sollen Entwickler leichter in der Lage sein, testgetriebene Entwicklung einzusetzen, was ihnen aber freigestellt ist. Eine Handlungsanweisung würde hier zwar die Testabdeckung und Testbarkeit verbessern, jedoch ist dies mit den engen zeitlichen Grenzen für diese Arbeiten abzuwägen.

5.3.11 Singletons

Saros nutzt seine Kernklassen (83 Klassen) als *Singletons*, jedoch werden sie in einem *PicoContainer* erzeugt, verwaltet und durch *Dependency Injection* in anderen Klassen verwendet. Dies ermöglicht es, die abhängigen Objekte gegen neue Instanzen oder *Mock*-Objekte auszutauschen.

Ein Weg hierfür ist der Einsatz von *Setter Injection*. Hier werden (beispielsweise beim Vorbereiten von Modultests) *Singleton*-Objekte in einer Klasse mittels einer Set-Funktion mit einer neuen Instanz ersetzt. Ich habe diese Technik bei der Entwicklung der Klasse `SarosNet` (siehe Kapitel 5.4) eingesetzt. Dadurch konnte ich einen *JUnit*-Test verfassen, welcher in der Testvorbereitung neue Instanzen für ansonsten als *Singletons* genutzte Klassen erzeugt und damit arbeitet.

5.3.12 Unterstützung von Testwerkzeugen

Saros nutzt die Testframeworks *JUnit* sowie *STF* (Sandors Test Framework, [Szü10]) und wurde für deren Nutzung angepasst. In einer aktuellen Bachelorarbeit wird gegenwärtig die Codeabdeckung und das Testframework *STF* verbessert. Diese und die in Kapitel 5.3.10 genannten Maßnahmen haben das Ziel, die Testbarkeit von Saros durch den Einsatz von Testwerkzeugen zu verbessern.

5.4 SarosNet für bessere Testbarkeit

Nach der Betrachtung von Verbesserungspotential in der Netzwerkschicht stelle ich eine realisierte Verbesserung vor, welche die Testbarkeit der Netzwerkschicht erhöht.

Zu den Kategorien, Abhängigkeiten und Trennung der Belange gehörte eine testhindernde Eigenschaft der Netzwerkschicht. Die Plugin-Klasse `Saros` beinhaltete Netzwerkkomponenten, was dazu führte, dass das Testen von Netzwerkfunktionalitäten eine Saros-Instanz voraussetzte. Dies stellte eine nicht notwendige und sogar unsinnige Abhängigkeit dar, welche äußerst hinderlich beim Entwickeln von Tests der Netzwerkschicht war. Die Abhängigkeit verletzte das Prinzip *Trennung der Belange*, da eine Plugin-Instanz der Klasse `Saros` nichts mit einer Netzwerkkomponente zu tun hat.

Ich identifizierte in der Klasse Saros somit alle Variablen, Objekte und Methoden, welche zur Netzwerkschicht gehörten (*Smack*-basierter Code, Verbindungsverwaltung, *Bytestream*-Konfiguration etc.) und verschob diese in eine neue Klasse `SarosNet`. Dadurch war auch eine große Zahl anderer Klassen anzupassen, um die neue Klasse zu nutzen. Die Klasse `Saros` erhielt anschließend eine Instanz von `SarosNet`, um darauf Netzwerkfunktionalitäten ausführen zu können.

5.4.1 Fazit

Durch diese durchgängige Trennung von Plugin- und Netzwerkkomponenten wurde eine klarere Abgrenzung zwischen diesen Belangen erreicht. Dies brachte zum einen den Vorteil, dass die Klasse `Saros` nun übersichtlicher (kein Netzwerkcode enthalten), kleiner (auf rund 60% der *LOC* geschrumpft) und somit verständlicher wurde.

Zum anderen wurde die Testbarkeit der Netzwerkkomponente von Saros verbessert. Es können nun feingranulare Modultests für die Netzwerkschicht entwickelt werden, da ein Test eine oder mehrere Instanzen von `SarosNet` erzeugen kann, ohne jeweils auf eine komplette `Saros`-Instanz angewiesen zu sein.

Abschließend entwickelte ich einen ersten *JUnit*-Test, welcher zwei Instanzen von `SarosNet` erzeugt und nutzt, um diese sich mit dem *XMPP*-Netzwerk verbinden zu lassen. Dies wäre zuvor nur mit zwei Saros-Instanzen möglich gewesen.

6 Arbeiten im Saros-Team

Im Rahmen meiner Diplomarbeit arbeitete ich im Saros-Entwicklerteam und nahm somit am Entwicklungsprozess teil, welcher von den Entwicklungstätigkeiten der teilnehmenden Studenten für Abschlussarbeiten sowie von monatlichen Test- und Releasephasen geprägt ist. Planmäßig in der letzten Woche jeden Monats wird eine neue Version von Saros veröffentlicht, was durch die Abarbeitung eines festgelegten Protokolls erfolgt. Es sind dabei fünf Rollen mit vorgeschriebenen Aufgaben vom Saros-Team zu übernehmen: Der Release-Manager (*RM*) und assistierende Release-Manager (*ARM*) bereiten die neue Version für die Testphase vor, führen die Testsuiten aus (*JUnit* & *STF*), testen mit den Testmanagern die versionsgerechte Testsuite und veröffentlichen abschließend die neue Version.

Der Test-Manager (*TM*) und der assistierende Test-Manager (*ATM*) entwerfen eine Testsuite für neue und geänderte Funktionalitäten der neuen Version und führen diese Testsuite aus.

Der Dokumentationsmanager aktualisiert die Liste offener Dokumentationsmängel und weist Einträge daraus Mitarbeitern zu.

Die Besetzung von Rollen und die Beschreibung der durchzuführenden Arbeiten führen zu klaren Verantwortlichkeiten und somit zu einem strukturierten Ablauf des Releasevorganges. Ich hatte jeweils einmal die Rolle des *ATM* und des *TM* inne.

Eine weitere eingesetzte Maßnahme der Qualitätssicherung sind *Code Reviews*. Es wurde vorgeschrieben, dass Änderungen am Quellcode erst nach Durchsicht und Akzeptanz von mindestens zwei anderen Entwicklern in die Versionsverwaltung übertragen werden dürfen. Die Entwickler werden angehalten, diese Reviews wenn möglich als *Peer Reviews* über Saros zu tätigen. Auf diese Weise ergibt sich der Vorteil, Einblicke und Erklärungen vom Entwickler der Änderungen zu erhalten. Auch das Nutzen von Saros als Anwender, anstatt nur als Entwickler, erlaubt einen Perspektivenwechsel, welcher neue Einsichten in die Benutzung und Stabilität von Saros ermöglicht.

Ein weiterer Vorteil, den *Code Reviews* mit sich bringt, ist der dadurch stattfindende Wissenstransfer. Während der Analyse von fremdem Quellcode und durch die Kommunikation mit anderen Entwicklern können neue Techniken, Tricks und Methoden erlernt werden. Aber auch durch Kritik an eigenen Patches können neue Erfahrungen gesammelt werden. Als überwiegend *C*++ praktizierender Entwickler habe ich mir von Saros-Entwicklern mit viel *Java*-Erfahrung neue Programmier- und Dokumentationstechniken aneignen können.

Der Nachteil der Release-Rollen und *Code Reviews* ist der für den Entwickler zusätzliche Zeitaufwand. Diesen muss er in seinem Zeitplan einkalkulieren. Aber wie gezeigt, hat der Aufwand seine Berechtigung und seinen Nutzen – für Saros, für die Entwickler und schlussendlich auch für den Anwender.

7 Ausblick

Die in dieser Arbeit behandelten Probleme in der Netzwerkkommunikation werden – auch wenn mir derzeit keine weiteren offenen Probleme bewusst sind – nicht die letzten sein.

Durch die Vielseitigkeit von Netzwerkkonfigurationen sind unbekannte, problematische Netzwerkszenarien möglich. Somit sind neben eigenen Tests von bekannten und ungewöhnlichen Netzwerkszenarien auch Auswertungen von bisher unbekannten Szenarien wichtig. Dies kann durch Kontaktaufnahme zu Anwendern und durch Analyse von Fehlerprotokollen geschehen. Sowohl ein Ausbau von Analysetechniken (neue hilfreiche *Log*-Ausgaben, z.B. über vorliegendes Netzwerkszenario) kann hier helfen als auch zusätzliche Anregungen im Umfrageformular für den Anwender, um im Problemfall hilfreiche Angaben zu Kommunikationsstörungen oder zum Netzwerk machen zu können.

Weiterhin ist es für die Stabilität der Netzwerkschicht hilfreich, dessen Testbarkeit durch geeignete Techniken wie *Exception Seeding* und *Fault Injection* zu verbessern. Auf diese Weise kann die existierende Fehlerbehandlung analysiert und verbessert werden, was bei Kommunikationsstörungen zu einer robusteren Behandlung (durch Neuversuch oder Fallbacktechniken) von Problemen führen kann. Infolgedessen kann schon Versagen entdeckt werden, ohne dass es im Einsatz beim Anwender auftritt.

8 Fazit

Diese Diplomarbeit behandelt die Netzwerkschicht von Saros und sucht Wege, die Stabilität und damit die Verlässlichkeit dieser Komponente zu erhöhen. Dafür wird eingangs das *Eclipse Communication Framework* als Ersatz für die bisherige Netzwerkschicht evaluiert. Da das *ECF* jedoch auf der gleichen Technologie für die Netzwerkkommunikation basieren würde, wie die bisherige Implementierung in Saros und auch die Integration des *ECF* keine strukturellen Erleichterungen bieten würde, wird von der Nutzung des *ECF* abgesehen und sich im Rahmen dieser Arbeit auf die Stabilität der aktuellen Netzwerkschicht konzentriert. Für diese werden Versagensfälle zusammengetragen und die Suche nach den zugrundeliegenden Defekten, sowie deren Korrektur beschrieben. Dabei habe ich Verbesserungen in der Stabilität des Einladungsprozesses, der Stabilität von *In-Band Bytestreams* sowie vor allem Verbesserungen der Verfügbarkeit von *Socks5 Bytestreams* geschaffen.

Die Probleme beim Aufbau von *Socks5 Bytestreams,* aufgrund unzureichender Kenntnisse einer Saros-Instanz zur Adressierung anderer Instanzen, werden beschrieben und Lösungen in Form von ausgebauter Adressermittlung (sowohl lokaler aus auch globaler Adressen) vorgestellt. Dazu trägt auch die Nutzung von *Universal Plug and Play* bei, dessen Funktionsweise und Integration in Saros beschrieben wird. Die erreichte verbesserte Verfügbarkeit von *Socks5 Bytestreams* resultiert durch die schnellere Datenübertragung in diesen Fällen in einer anwenderfreundlicheren Saros-Sitzung.

Abschließend geht diese Arbeit auf das Thema Testbarkeit ein, beschreibt deren Faktoren (Beobachtbarkeit, Kontrollierbarkeit, Komplexität, Abhängigkeiten, Trennung der Belange) und bewertet diese und weitere Faktoren in Saros mit dem Ziel, Mängel und Verbesserungspotential aufzuzeigen. Hier wird vor allem der Einsatz von *Exception Seeding* und *Fault Injection* für eine Verbesserung der Kontrollierbarkeit der Netzwerkschicht empfohlen, um die Testbarkeit der Netzwerkschicht zu verbessern.

Abschließend zeigt die Einführung der Klasse `SarosNet` meine Umsetzung einer Testbarkeitsverbesserung in Form einer klareren Trennung der Belange bezüglich Plugin und Netzwerk. Dies wiederum erleichtert die Testbarkeit von Netzkomponenten deutlich, da Netzwerktests nicht mehr an eine komplette Saros-Instanz gekoppelt sind.

9 Literaturverzeichnis

[weUPnP] Bahgat, Alessandro and Castagna, Daniele. weupnp Homepage. [Online] [Cited: 07 31, 2011.] http://code.google.com/p/weupnp/.

[Bec04] Beck, Kent. 2004. *Extreme Programming Explained, Second Edition: Embrace change.* s.l. : Addison-Wesley Professional, 2004. 978-0-321-27865-4.

[Bin99] Binder, Robert. 1999. *Testing object-oriented systems: models, patterns, and tools.* s.l. : Addison Wesley, 1999. 978-0201809381.

[Bry06] Bryant, Sallyann, du Boulay, Benedict and Romero, Pablo. 2006. *XP and Pair Programming practices.* University of Sussex, Falmer, UK : IDEAS laboratory, 2006.

[Can07] Canfora, et al. 2006. *Evaluating performances of pair designing in industry.* 2006.

[Dje06] Djemili, Riad. 2006. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteiler Paarprogrammierung. [Online] 8 2006. [Cited: 6 30, 2011.] http://www.inf.fu-berlin.de/inst/ag-se/theses/Djemili06-eclipse-erweiterung-PP.pdf.

[ECFAD] ECF. ECF API Documentation. [Online] http://wiki.eclipse.org/ECF_API_Docs.

[ECFHP] ECF Homepage. [Online] [Cited: 07 27, 2011.] http://www.eclipse.org/ecf/.

[ECF] ECF Providers. [Online] http://wiki.eclipse.org/ECF_Providers.

[Flo91] Flor, N.V. and Hutchins, E.L. 1991. *Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance.* s.l. : Empirical Studies of Programmers: Fourth Workshop, 1991.

[Gus07] Gustavs, Björn. 2007. Weiterentwicklung eines Eclipse-Plug-Ins zur Verteilten Paarprogramierung. [Online] 2007. http://www.inf.fu-berlin.de/inst/ag-se/theses/Gustavs07-saros-DPP-II.pdf.

[IEE90] IEEE. 1990. *IEEE Standard Glossary of Software Engineering Terminology, IEEE 610.12-1990.* 1990. 1–55937467–X.

[RFC5389] IETF. 2008. RFC5389. [Online] 10 2008. [Cited: 07 14, 2011.] http://tools.ietf.org/html/rfc5389.

[JAX09] jaxEnter. 2009. Kommunikationstalent Eclipse: Das ECF-Projekt. [Online] Juli 2009. http://it-republik.de/jaxenter/artikel/Kommunikationstalent-Eclipse-Das-ECF-Projekt-2403.html.

[Jun06] Jungmayr, Stefan. 2006. *Anforderungsgetriebene Entwicklung testbarer Software – ASQF FG Software-Test.* 2006.

[Jun00] —. 2000. Design für Testbarkeit. [Online] 11 26, 2000. [Cited: 07 28, 2011.] http://www.testability.de/Publikationen/STI_jungmayr.pdf.

[Jun04] —. 2004. *Improving testability of object-oriented systems.* s.l. : dissertation.de, 2004. ISBN 3-89825-781-9.

[Jun05] —. 2005. Testbarkeit im Entwicklungsprozess. *Vortrag beim Kolloquium "Testen im System- und Software-Lifecycle".* [Online] 2005. http://www.testbarkeit.de/Publikationen/TAE05_Artikel_jungmayr.pdf.

[Ker99] Kernighan, Biran W. and Pike, Rob. 1999. *The practice of programming.* s.l. : Addison-Wesley, 1999.

[Lak96] Lakos, John. 1996. *Large-Scale C++ Software Design.* s.l. : Addison-Wesley, 1996. 978-0201633627.

[Lew08] Lewis, Scott. 2008. ECF developer mailing list. [Online] 08 22, 2008. [Cited: 07 12, 2011.] http://permalink.gmane.org/gmane.comp.ide.eclipse.ecf.devel/1237.

[Lor94] Lorenz, M. and Kidd, J. 1994. *Object-Oriented Software-Metrics.* s.l. : Pentice Hall, Englewood Cliffs, 1994. 978-0131792920.

[Mar03] Martin, Robert C. 2003. *Agile Software Development.* s.l. : Prentice-Hall, 2003. 9780135974445.

[McC98] McConnell, Steven C. 1998. *Software Project Survival Guide.* s.l. : Microsoft Press, 1998.

[McD03] McDowell, Charlie, et al. 2003. *The Impact of Pair Programming on Student Performance, Perception and Persistence.* s.l. : ICSE 2003, 2003. pp. 602-607. 0-7695-1877-X.

[Ora11] Oracle. The Java Language, An Overview. [Online] [Cited: 07 19, 2011.] http://java.sun.com/docs/overviews/java/java-overview-1.html.

[Pet02] Pettichord, Bret. 2002. *"Design for Testability.* Pacific Northwest Software Quality Conference : s.n., 2002.

[Pet01] —. 2001. *Design for Testability.* 2001. Presentation.

[Pre99] Prechelt, Lutz. 1999. Ausgewählte Themen der Softwareentwicklung. [Online] 1999. [Cited: 07 27, 2011.] http://page.mi.fu-berlin.de/prechelt/swt2/skript.html.

[Rie08] Rieger, Oliver. 2008. Weiterentwicklung einer Eclipse Erweiterung zur Realisierung und Protokollierung verteilten Paarprogrammierung im Hinblick auf Kollaboration und Kommunikation. [Online] 2008. http://www.inf.fu-berlin.de/inst/ag-se/theses/Rieger08-eclipse-erweiterung-PP.pdf.

[Sta10] Staib, Henning. 2010. Verbesserung einer XMPP-Bibliothek für den Einsatz in verteilter Paarprogrammierung. [Online] 06 2010. http://www.inf.fu-berlin.de/inst/ag-se/theses/Staib10-DPPX.pdf.

[Sto03] Stotts, David, et al. 2003. *Virtual teaming: Experiments and experiences with distributed pair programming.* s.l. : in Proc. XP/Agile Universe, 2003.

[Szü10] Szücs, Sandor. 2010. Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung. [Online] 03 2010. http://www.inf.fu-berlin.de/inst/ag-se/theses/Szuecs10-DPPVI.pdf.

[UPn11] UPnP.org. 2011. UPnP Forum. [Online] 2011. [Cited: 07 15, 2011.] http://upnp.org.

[Wal01] Wallmüller, Ernest. 2001. *Software-Qualitätsmanagement in der Praxis.* Wien : s.n., 2001. 3-446-21367-8.

[Wil001] Williams, et al. 2000. *Strengthening the Case for Pair Programming.* s.l. : IEEE 17,4, 2000. pp. 19-25.

[Wil00] Williams, Laurie Ann. 2000. *The Collaborative Software Process.* 2000.

[XMP47] XMPP.org. 2011. XEP-0047: In-Band Bytestreams. [Online] 2011. [Cited: 07 08, 2011.] http://xmpp.org/extensions/xep-0047.html.

[XEP65] —. 2011. XEP-0065: SOCKS5 Bytestreams. [Online] 04 20, 2011. [Cited: 07 07, 2011.] http://xmpp.org/extensions/xep-0065.html.

10 Anhang

10.1 Beispiel einer detaillierten Beschreibung eines UPnP Gerätes

Es folgt das Beschreibungs-XML-Dokument, welches der Router D-Link DI-524 ausgibt.


```
<root xmlns="urn:schemas-upnp-org:device-1-0">
   <specVersion>
      <major>
         1
      </major>
      <minor>
         0
      </minor>
   </specVersion>
   <device>
      <deviceType>
         urn:schemas-upnp-org:device:InternetGatewayDevice:1
      </deviceType>
      <friendlyName>
         D-Link DI Series
      </friendlyName>
      <manufacturer>
         D-Link
      </manufacturer>
      <manufacturerURL>
         http://www.dlink.com
      </manufacturerURL>
      <modelDescription>
         D-Link Internet Gateway Device
      </modelDescription>
      <modelName>
         D-Link Internet Gateway Device
      </modelName>
      <UDN>
         uuid:001B11A4-0EBE-0000-0000-0000C0A80001
      </UDN>
      <presentationURL>
         http://192.168.0.1:80/
      </presentationURL>
      <serviceList>
         <service>
            <serviceType>
               urn:schemas-upnp-org:service:Layer3Forwarding:1
            </serviceType>
            <serviceId>
               urn:upnp-org:serviceId:L3Forwarding1
            </serviceId>
            <controlURL>
               /upnp/control1
            </controlURL>
            <eventSubURL>
               /Layer3Forwarding
            </eventSubURL>
            <SCPDURL>
               http://192.168.0.1:80/serv1.xml
            </SCPDURL>
         </service>
      </serviceList>
      <deviceList>
         <device>
            <deviceType>
               urn:schemas-upnp-org:device:WANDevice:1
            </deviceType>
            <friendlyName>
               WAN Device
            </friendlyName>
            <manufacturer>
               D-Link
            </manufacturer>
            <manufacturerURL>
               http://www.dlink.com
            </manufacturerURL>
            <modelDescription>
               Residential Gateway
            </modelDescription>
            <modelName>
               Residential Gateway Device
            </modelName>
            <UDN>
               uuid:001B11A4-0EBE-0000-0000-0001C0A80001
```

```
            </UDN>
            <serviceList>
               <service>
                  <serviceType>
                     urn:schemas-upnp-org:service:WANCommonInterfaceConfig:1
                  </serviceType>
                  <serviceId>
                     urn:upnp-org:serviceId:WANCommonInterfaceConfig
                  </serviceId>
                  <controlURL>
                     /upnp/control2
                  </controlURL>
                  <eventSubURL>
                     /WANCommonInterfaceConfig
                  </eventSubURL>
                  <SCPDURL>
                     http://192.168.0.1:80/serv2.xml
                  </SCPDURL>
               </service>
            </serviceList>
            <deviceList>
               <device>
                  <deviceType>
                     urn:schemas-upnp-org:device:WANConnectionDevice:1
                  </deviceType>
                  <friendlyName>
                     WAN Connection Device
                  </friendlyName>
                  <manufacturer>
                     D-Link
                  </manufacturer>
                  <manufacturerURL>
                     http://www.dlink.com
                  </manufacturerURL>
                  <modelDescription>
                     Residential Gateway
                  </modelDescription>
                  <modelName>
                     Residential Gateway Device
                  </modelName>
                  <UDN>
                     uuid:001B11A4-0EBE-0000-0000-0002C0A80001
                  </UDN>
                  <serviceList>
                     <service>
                        <serviceType>
                           urn:schemas-upnp-org:service:WANIPConnection:1
                        </serviceType>
                        <serviceId>
                           urn:upnp-org:serviceId:WANIPConnection
                        </serviceId>
                        <controlURL>
                           /upnp/control3
                        </controlURL>
                        <eventSubURL>
                           /WANIPConnection
                        </eventSubURL>
                        <SCPDURL>
                           http://192.168.0.1:80/serv3.xml
                        </SCPDURL>
                     </service>
                  </serviceList>
               </device>
            </deviceList>
         </device>
      </deviceList>
   </device>
</root>
```

10.2 E-Mailverkehr mit Projektleitern von weupnp bzgl. Code Contribution

Es folgt der E-mailverkehr mit den Projektleitern des Open Source Projektes weupnp. Jede (relevante) E-mail beginnt mit dem Sendedatum, sowie anonymisierten Absender und Empfänger.

23.5.2011, gustavs an weupnp

```
Hallo weupnp developers,

I am student at the Free University of Berlin (Germany) and for a project I am adding
UPnP support.
I chose weupnp to do so and implemented some improvements I want to share with you.


The main things I added:

- support for multi-network interfaces
As soon as there are multiple network interfaces, it is luck and OS/java dependent if the
broadcast can reach the gateway. So now I send it on all suitable interfaces concurrently
and wait for responses.

- try different search types in discovery broadcast if no response to default type.

- refactored add/delete Portmapping
Return errorcode from gateway.
addPortmapping support lease duration.

- get*PortMappingEntry
Returns the PortmappingEntry
Applying more complete details from gateway response.

- for the test class "Main" I added a GUI for logging output. I wanted also "common peo-
ple" test the lib by just executing it.

- and little things...


I tested it with my router on Win7,WinXP,MacOS, several Linux derivatives.
Routers successfully tested with:
- D-Link DIR-300
- FRITZ!Box Fon WLAN 7270 v2


I am sending a diff and an uptodate jar, including source code for you to have a look.

Please let me know if and how I can help to contribute.


Thanks.
Björn
```

24.5.2011, weupnp an gustavs

```
Hello Björn,
what an awesome work you did!
We will try to test your version with all the environments we can reach.

If (as I am sure) that works with our environments we will grant you
commit rights to our repository so that you will be credited for your
commits.

We'll get in touch soon.

Thank you, and kudos to your Uni for letting you to engage in such projects :-)

Cheers,
Alessandro
```

14.6.2011, weupnp an gustavs

```
Hello Björn,
we did some tests and the code looks great.
```

We'd like to commit the code in two phases:
1. first everything that adds functionality and allows us use the present APIs
2. then we'll release a new version of the library (current API + your fixes)
3. and then we'll update the API making the appropriate changes and updates (we'll need to be in touch for that)

We would proceed in this way in order to maximize functionalities for existing users, and make the new APIs available from next version on.
What would you say about that?

Are you interested in making the commits yourself or do you prefer us to handle that?

Thank you

16.6.2011, gustavs an weupnp

Hello Alessandro ,

I'm glad that you like to integrate my patches.

> ...
> We would proceed in this way in order to maximize functionalities for
> existing users, and make the new APIs available from next version on.
> What would you say about that?

I totally agree to the integration in phases. API changes should not be rushed to keep the upgrade for people hassle free.

> Are you interested in making the commits yourself or do you prefer us
> to handle that?
I am glad to do this, to split the changes in two patches.

Thank you =)
Björn Gustavs

19.6.2011, weupnp an gustavs

That'll be great!

First of all, we'll need your Google account so that I can grant to it commit rights on our repository. Which would you like to use?
Then we'll start by committing the first set of changes that don't change the public APIs. Do you have that patch ready?

I wanted to make just a couple of notes:
* we deliberately avoided using e.printStackTrace(); because we don't know anything about the context in which weupnp is running, so I'd comment lines using that, for now.
* for methods that return lists or private fields that are lists, it's better if they return plain java.util.List instead of ArrayList

Please let me know your account so that we can proceed as soon as we're ready

Have a great weekend.
Alessandro

19.6.2011, gustavs an weupnp

Hello again Alessandro,

I prepared a stripped down patch that does not include API changes.
I had to remove some nice things, making me looking forward to commit phase 2. =)

Regards,
Björn

20.6.2011, weupnp an gustavs

That will be great!
Go on and commit it, you should have the appropriate permissions.

We'll make a final review, tag the present version and proceed with phase 2 soon!

Thank you,
Alessandro

21.6.2011, weupnp an gustavs

Hello,
I've seen you commits: thank you!
I spoke with Daniele yesterday and there may be a couple of minor things we would like to ask you to review (concerning coding

conventions& documentation).
Would you be willing to do that if we gave you the details?
Would you prefer if we sent you an email or if we used the code review
tool built into google code?

For example, one possible thing we may want to reconsider is the
option to retain a CLI interface to test weupnp: I was thinking that
if you want to test weupnp on a headless terminal you can't do it
anymore.

What would you say?

Have a great day!

---

x.6.2011, gustavs an weupnp
Concerning the reviews you'd like: of course, you are the project masters and know best the users, usage and expectations. =)

If its no extra work, I'd like the issues per email.

I agree with the CLI option, thats a usage angle I didnt considered. I'd make the CLI happen depending on a passed argument, like
java -jar weupnp.jar nogui
???

Looking forward to hear from you.
Björn

10.7.2011, weupnp an gustavs
Hello,
as promised, here's a brief set of points we'd love to review with you in order to increase readability and maintainability of weupnp.

* equality checks: boolean checks like
```
         if (nameValue.containsKey("NewInternalClient")==false)
```
are less readable than just
```
        if (!nameValue.containsKey("NewInternalClient"))
```
I'd try to stick with the latter throughout the code

* since it is now less unlikely to have multiple valid gateways, I'd add a new method (e.g. getValidGateways()) that returns the complete list of discovered gateways

* we'd restore the CLI interface for running the test. weupnp is targets mostly developers, and the will, in general, familiar with CLI apps. Plus, by having that option, we'd be able to run our test on headless machines as well.
In addition to that, having the GUI would force our lib to depend on swing, even if that is not closeley related to what weupnp is supposed to do.
Because of this (and because it would be easier to maintain), we'd prefer the Main class to use the console for output. What would you say?

Thank you for you great help!

Best,
Alessandro

11.7.2011, gustavs an weupnp

The little NOT operator is easy to miss in my experience, instead of ==false or ==true being more eye catching.
But no problem, to keep the code clean, I will change those conditions to the not operator.

> I'd add a new method (e.g. getValidGateways()) that returns the complete list >of discovered gateways
Agreed, I'll add this method as I did in my first overall patch.

>we'd restore the CLI interface for running the test.
I added this for common users to be able to test their home setup. So I could send those the JAR, asked them to double click and see the log. To collect compatible statistics its nice to have any user be able to perform such test.
To have the CLI working, I could add a parameter check, so advanced users can start the jar with a parameter "-nogui" to have the CLI output.
But that jar still binds to swing.
So if you still prefer the CLI-only anyway, just say so and I will remove the GUI.

Have a nice day.

Regards,
Björn

12.7.2011, weupnp an gustavs

Hi Björn!

I was the one who didn't really want any reference to the swing in an upnp library.
But after reading your answer "I added this for common users to be able to test their home setup", I understand why it can be really helpful to test the library on different systems.

Said so, I still believe that a developer who wants to use a upnp library wouldn't be happy of finding swing dependencies (I wouldn't).
We're going to take a look at Maven to make the compilation conditional to build two jars (with and without gui).

Tank you again.

Daniele

16.7.2011, gustavs an weupnp

Hello,

I committed the changes as proposed.

- boolean conditions check with NOT operator
- removed GUI&Swing (can be reactivated from history, hopefully. I keep the gui for me :)
- added method getAllGateways()

Regards,
Björn

20.7.2011, weupnp an gustavs

Thanks!
That's great news :)

We'll proceed with the new version soon, than.
Will get in touch shortly!

10.3 Umfragebogen (Saros User Feedback Survey)

Firefox

Saros User Feedback Survey

http://saros-build.imp.fu-berlin.de/phpESP/public/survey.php?

Google

Saros User Feedback Survey

1. How did your latest Saros session work out?

2. Which goals did you want to accomplish in your latest Saros session?

3. What didn't go well (if anything) and what happened?

4. Do you have any other comments to share with us (e.g about the interface, specific features or possible improvements)?

5. If you want to subscribe to our low-volume announce list on SourceForge to receive the latest news about Saros, you can enter your email address here. Your address is only used for this purpose and will not be passed on to third parties.

Submit Survey