Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Verbesserung der Kommunikationsmöglichkeiten in einem Werkzeug zur verteilten Paarprogrammierung (Saros)

Olaf Loga
Matrikelnummer: 4128187
ologa@gmx.de

Betreuer: Karl Beecher und Stephan Salinger
Eingereicht bei: Prof. Dr. Lutz Prechelt

Berlin, 08. April 2010

Zusammenfassung

Die vorliegende Bachelorarbeit hat zum Ziel, die Kommunikationsmöglichkeiten des Werkzeuges *Saros* zur verteilten Paarprogrammierung in Eclipse aufzuzeigen, zum einen durch eine Chat- und zum anderen durch eine Voice-over-IP Funktion (VoIP). Durch die Chatfunktion können die Teilnehmer während einer Saros Sitzung mithilfe von Text-, durch die VoIP Funktion mithilfe von Sprachnachrichten miteinander kommunizieren. Diese Arbeit beschreibt Konzept, Aufbau, Implementierung, Benutzung und mögliche Weiterentwicklungen der für das Saros Plugin entwickelten Erweiterungen.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

Berlin, den 08. April 2010

Olaf Loga

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Aufbau dieser Arbeit 1
1.2 (Verteilte) Paarprogrammierung 1
1.3 Saros 2
1.3.1 Bisherige Kommunikationsmöglichkeiten in Saros . . . 3
1.3.2 Kommunikation in anderen DPP Eclipse Plugins . . . 3
1.4 XMPP 4
1.4.1 Authentifizierung (Teil des RFC 3920) 4
1.4.2 Kontakt-/Freundesliste (Teil des RFC 3921) 5
1.4.3 XEP-0045: Multi-User-Chat 5
1.4.4 XEP-0166: Jingle 5

2 Ziele **7**
2.1 Chat 7
2.2 VoIP 8
2.2.1 Funktionale Anforderungen 8
2.2.2 Nicht funktionale Anforderungen 9

3 Implementierung **10**
3.1 Chat 10
3.1.1 Bisherige Nutzung 10
3.1.2 Bedienbarkeit 10
3.1.3 Sicherheit 13
3.1.4 Erweiterungsmöglichkeiten 14
3.2 VoIP 14
3.2.1 Funktionen 15
3.2.2 Bedienbarkeit 15
3.2.3 Problemfälle bei der Benutzung 17
3.2.4 Wahl eines geeigneten Audio Codecs 19
3.2.5 Aufbau der VoIP Funktion 21
3.2.6 Ablauf einer VoIP Sitzung 25
3.2.7 Saros StreamService 28
3.2.8 Erweiterungsmöglichkeiten 31
3.2.9 Einstellungsmöglichkeiten 31

4 Fazit **34**

A Anwendungsfälle **35**
A.1 Chat 35
A.2 VoIP 36

B Anmerkungen **38**

Abbildungsverzeichnis

1 Chat View 10
2 Einstellungsseite der Chat Funktion 11
3 Multi-User-Chat im SIP Communicator mit Usern, die sich gerade in einer Saros Sitzung befinden. 12
4 Starten einer neuen VoIP Sitzung 16
5 Einladungsdialog zu einer neuen VoIP Sitzung 16
6 Stoppen einer VoIP Sitzung 17
7 Fehlermeldung über ein fehlendes oder falsch konfiguriertes Wiedergabegerät. 17
8 Warnmeldung über fehlendes oder falsch konfiguriertes Aufnahmegeräte 18
9 UML Diagramm der VoIP Funktion 24
10 Erfolgreiches Aufbauen einer VoIP Sitzung als Sequenzdiagramm 26
11 Erfolgreiches Beenden einer VoIP Sitzung als Sequenzdiagramm 27
12 Zustandsdiagramm einer VoIP Session 27
13 Einstellungsmöglichkeiten der VoIP Funktion 32

1 Einleitung

Die vorliegende Bachelorarbeit befasst sich mit der Verbesserung der Kommunikationsmöglichkeiten in einem Eclipse-Plugin zur verteilten Paarprogrammierung *(Saros)*, das an der Freien Universität Berlin anfänglich in der Diplomarbeit von Riad Djemili (Dje06) entwickelt wurde. Diese wird seitdem in mehreren wissenschaftlichen Arbeiten (Gus07) (Rie08) (Jac09) (Rin09) (Szü10) (Zil09) (Doh09) (Sta10) (Sót09) (Lau10) (Kiw10) innerhalb der AG Software Engineering ausgebaut und weiterentwickelt.

1.1 Aufbau dieser Arbeit

Im ersten Teil dieser Arbeit werde ich das Konzept der (verteilten) Paarprogrammierung in Verbindung mit Saros beschreiben.

Anschließend werde ich auf die Ziele der Chat- und VoIP Funktion und deren Anforderungen präziser eingehen, sowie deren Konzept und Implementierung im dritten Teil genauer ausarbeiten und einen Ausblick über mögliche Erweiterungen dieser Funktionen geben.

Im letzten Teil werde ich noch Anwendungsfälle der einzelnen Funktionen vorstellen.

1.2 (Verteilte) Paarprogrammierung

Unter Paarprogrammierung versteht man eine spezielle Arbeitstechnik der agilen Softwareentwicklung. Es geht dabei darum, dass während des Erstellens von Quellcodes zwei Programmierer an einem Arbeitsplatz zusammenarbeiten. Einer dieser Programmierer schreibt den Quelltext (*Driver*), während der andere eine überwachende Rolle einnimmt (*Observer*). Er denkt über das gerade zu lösende Problem nach, überprüft den soeben geschriebenen Code (z. B. nach Syntax- oder Semantikfehlern) und die Erfüllung der gegebenen Anforderungen an die Implementierung. Dabei spricht er auftretende Probleme sofort an und beide versuchen diese Probleme in einem Gespräch zu lösen. Während der Paarprogrammierung sollten sowohl die Rollen, als auch die Paare nach bestimmten Zeiten gewechselt werden. Dies ist vorteilhaft, da der neue Partner durch *learning-by-doing* schnell in den ihm eventuell unbekannten Quellcode eingearbeitet, aber auch dass die Software Qualität durch die Kontrollfunktion gesteigert werden kann. Die Hauptvorteile sind ein qualitativ höherwertiges Endprodukt, sowie eine höhere Produktivität des Entwicklerpaares. Inwieweit dieser Vorteil jedoch genutzt werden kann hängt z. B. von der Einarbeitungszeit des Teams oder von der Persönlichkeit der Teilnehmer ab (WKCJ) (WK02) (CW). Der größte Nachteil ist, dass zwei Programmierer zur gleichen Zeit am selben Problem eines Projektes arbeiten und sich dadurch die Arbeitskosten erhöhen.

Unter verteilter Paarprogrammierung versteht man die Paarprogrammierung im räumlich getrennten Umfeld, wie es z. B. von *Saros* ermöglicht wird. Die Schwierigkeit der Umsetzung der verteilten Paarprogrammierung besteht darin, die gleichen oder zumindest ähnliche Arbeitsbedingungen wie bei der klassischen Paarprogrammierung zu schaffen. Dazu gehört z. B. dass beide Programmierer gleichzeitig den selben Quelltext sehen und verändern können. Ebenso können beide miteinander kommunizieren und über Probleme diskutieren. Als Variante zur klassischen Paarprogrammierung existiert das *Side-by-Side Programming, SbS* wobei mehrere Programmierer gleichzeitig den Quelltext verändern und mehrere *Observer* einem Programmierer folgen können. Dies hat den wesentlichen Vorteil, dass man schnell mehrere neue Programmierer an einen für sie unbekannten Code heranführen kann, indem man ihnen in einer Sitzung live die wesentlichen Stellen des Quelltextes zeigen kann.

1.3 Saros

Saros ist ein Forschungsprojekt der Software Engineering Arbeitsgruppe des Instituts für Informatik an der Freien Universität Berlin. Es ist eine Open Source Erweiterung der Eclipse Entwicklungsumgebung und dient zur verteilten, kollaborativen Programmierung (engl. *Distributed Pair Programming - DPP*).

Dabei wird die klassische Paarprogrammierung erweitert, indem es mehrere Schreiber (*Driver*) gibt, die unabhängig voneinander verschiedene Dateien eines Projektes bearbeiten können. Darüber hinaus kann es auch mehrere *(*Observer) geben, die z. B. alle einem *Driver* folgen und somit eine mehrfache Überwachungs- bzw. Lehrfunktion entsteht.

Zur Benutzung von Saros benötigt jeder Teilnehmer die Entwicklungsumgebung Eclipse mit installiertem *Saros* Plugin und einen *Jabber* Account auf einem *XMPP* Server über den die Identifikation eines Benutzers möglich ist. Die Funktionen des *Extensible Messaging and Presence Protocol (XMPP)* werden in Kapitel 1.4 genauer behandelt. Der *Jabber* Account ist hauptsächlich für die Identifikation während und vor einer Saros Sitzung wichtig. Jeder Benutzer von *Saros* besitzt eine eigene Freundesliste, in die er neue Benutzer, die über ihre *JabberID* identifiziert werden, hinzufügen oder löschen kann. Dabei stehen diese Benutzer mit ihrer *JabberID* in der jeweiligen Freundesliste. Den *JabberIDs* können jedoch manuell eigene Namen zugewiesen werden, damit diese besser den Benutzern zugeordnet werden können. Eine neue Saros Sitzung wird dabei immer in Verbindung mit einem Projekt aus dem Eclipse Workspace eines bestimmten Benutzers gestartet. Der Benutzer dessen Projekt benutzt (geteilt, geshared) wird, ist im weiteren Verlauf dieser Sitzung der *Host*. Das ausgewählte Projekt wird nun an alle eingeladenen Benutzer übertragen, falls diese die Einladung zu einer neuen

Saros Sitzung annehmen. Der *Host* kann nun den anderen Benutzern Rollen zuweisen (*exclusive Driver, Driver, Observer, Observer im Follow-Mode*).

Die Rolle des *exclusive Driver* bedeutet, dass dieser Benutzer alleinige Schreibrechte besitzt. Alle anderen Benutzer sind währenddessen *Observer* oder *Observer im Follow-Mode* und können die Arbeit nur beobachten.

Der *Follow-Mode* bedeutet, dass der Benutzer den Arbeiten des *Drivers* genau folgen kann, d. h. er folgt automatisch dem Cursor des *Hosts*. Jeder *Observer* kann selbst entscheiden, ob er im *Follow-Mode* sein möchte oder nicht, also ob er den Arbeiten des *Drivers* genau folgen oder ob er sich im aktuellen Projekt frei bewegen möchte.

Weiterhin ist es Möglich, dass der *Host* mehreren Benutzern die *Driver* Rolle zuteilt. Dies bedeutet, dass mehrere Benutzer Schreibrechte besitzen und auch unabhängig voneinander verschiedene Dateien des aktuellen Projektes bearbeiten können. Alle anderen Benutzer, die zu diesem Zeitpunkt keine *Driver* Rolle zugeteilt bekommen haben sind weiterhin *Observer* und können nun frei entscheiden, ob sie einem bestimmten *Driver* folgen oder sich frei im Projekt bewegen möchten. Der *Host* besitzt zu Beginn jeder *Saros* Sitzung die alleinigen *Driver* Rechte, die er jedoch zu jeder Zeit an andere Benutzer abgeben kann. D.h. der *Host* ist nicht zwangsweise während einer *Saros* Sitzung jederzeit *Driver*, sondern kann auch andere Rollen annehmen. Er ist allerdings der Einzige, der *Driver* Rollen vergeben kann, was einem normalen *Driver* nicht möglich ist.

Saros unterstützt neben Java, die am häufigsten verwendeten Eclipse Erweiterung für diverse Programmiersprachen (z. B. C, C++, PHP oder Python) (Kiw10).

1.3.1 Bisherige Kommunikationsmöglichkeiten in Saros

In Saros existiert bisher keine Funktion zur Kommunikation innerhalb des Plugins. Zur Zeit besteht nur die Möglichkeit einen *Skype* Account zu hinterlegen, um schnell und einfach ein externes Skype Gespräch mit anderen Saros Benutzern aufbauen zu können. Die Benutzer müssen also für jegliche Art von Kommunikation (Text und Sprache) externe Applikationen benutzen (*Skype, Mumble, ICQ, etc.*). Dies hat den großen Nachteil, dass *Saros* bisher einen wesentlichen Aspekt der klassischen Paarprogrammierung vernachlässigt. Eine schnelle Diskussion über eventuelle Probleme während der Programmierung kann somit über *Saros* nicht zustande kommen.

1.3.2 Kommunikation in anderen DPP Eclipse Plugins

Im Folgenden werde ich die Kommunikationsmöglichkeiten in *Saros* mit anderen konkurrierenden Eclipse Plugins zur verteilten Paarprogrammierung

vergleichen. Dazu vergleiche ich *Saros* mit *XPairtise* (LS), *XecliP* (Glo) und *PEP* (PMS).

	Saros	XPairtise	XecliP	PEP
Chat		X	X	X
VoIP				

Stand: 01.03.2010

Alle in der oben stehenden Tabelle aufgeführten Eclipse Plugins zur verteilten Paarprogrammierung unterscheiden sich in ihrer Chatfunktion nur in kleinen Details. Fast alle Implementierungen benutzen einen Chatraum (*XPairtise* benutzt zwei Chaträume), die während einer Sitzung von den Benutzern betreten werden können. Keines dieser Plugins unterstützt bisher in irgendeiner Form eine VoIP Funktion.

Durch diese Arbeit wird Saros im Vergleich zu den anderen hier aufgeführten Eclipse Plugins die besten Möglichkeiten zur internen Kommunikation besitzen, ohne dass weitere externe Applikationen benötigt werden.

1.4 XMPP

Das *Extensible Messaging and Presence Protocol (XMPP)* ist eine Sammlung von XML-Technologien und dient zur Echtzeitkommunikation. Die Übertragung der Daten kann dabei entweder über einen *XMPP* Server geschehen oder der Server wird nur zum Aufbau der Verbindung benutzt und es existiert eine direkte *Peer-to-Peer* Verbindung zwischen den einzelnen Benutzern. Dies hat den entscheidenden Vorteil, dass die Bandbreite nicht durch den Server, sondern nur durch die Anbindung der einzelnen Benutzer begrenzt ist. Darauf werde ich später im Kapitel 1.4.4 eingehen. *XMPP* wird von mehreren Applikationen und Bibliotheken implementiert und bereitgestellt. Saros benutzt die *Smack API*, eine reine *Java* Bibliothek, die schon mehrere *XMPP* Funktionen implementiert hat. *XMPP* stellt verschiedene Services zur Verfügung und wurde mittlerweile durch einige Erweiterungen verbessert (PSA09). Im weiteren Verlauf werde ich auf die für diese Arbeit wichtigen Services und Erweiterungen eingehen.

1.4.1 Authentifizierung (Teil des RFC 3920)

Zur Benutzung eines *XMPP* Services ist es notwendig, dass jeder Benutzer ein Benutzerkonto (*JabberID*) auf einem *XMPP* Server besitzt. Es ist nicht notwendig, dass alle Benutzer die miteinander kommunizieren wollen auf demselben *XMPP* Server registriert sind, sondern der *XMPP* Server leitet die Nachrichten an Benutzer eines anderen Servers weiter.

Zur Authentifizierung bei einem *XMPP* Server muss der Benutzer zuerst eine TCP Verbindung mit dem Server (z. B. *jabber.org*) aufbauen, auf dem er

sein Benutzerkonto erstellt hat. Anschließend wird ein XML Stream geöffnet über den nun alle Nachrichten ausgetauscht werden können. Dazu sendet der Benutzer dem Server zuerst eine *HELLO-Nachricht* und erhält anschließend eine Liste von Features, die der XML Stream unterstützt. Dies wäre z. B. eine verschlüsselte Übertragung über *SSL (Secure Socket Layer)* und eine Authentifikation über *SASL (Simple Authentication and Security Layer)*, welche beschreibt, wie das Passwort während der Authentifkation verschlüsselt wird. Nun kann der Benutzer sein Kennwort unter Angabe der benutzten Übertragungs- und Authentifizierungsmethode an den Server schicken und ist anschließend erfolgreich authentifiziert. Die Verschlüsselung bei der Übertragung hat allerdings keinen Einfluss auf die Sicherheit des weiteren Datenverkehrs. Dieser läuft über die normale *TCP XMPP Connection*. Es wäre jedoch möglich eine *SSL* Verbindung zu benutzen, falls dies vom *XMPP* Server unterstützt wird (SA04a)(PSA09).

1.4.2 Kontakt-/Freundesliste (Teil des RFC 3921)

Sobald ein Benutzer sich bei einem *XMPP* Server erfolgreich authentifziert hat, erhält dieser die mit seinem Benutzerkonto verknüpfte Kontakt- bzw. Freundesliste (auch *Roster* genannt). In dieser Liste stehen alle Kontakte, die ein Benutzer vorher dieser Liste unter Angabe einer *JabberID* hinzugefügt hat (PSA09)(SA04b). In *Saros* kann ein Benutzer z. B. nun aus dieser Liste auswählen wen er gerne zu einer neuen *Saros* Sitzung einladen oder mit wem er ein neues *VoIP* Telefonat führen möchte. Jeder Benutzer, mit dem man kommunizieren möchte, muss vorher zur persönliche Kontaktliste hinzugefügt werden.

1.4.3 XEP-0045: Multi-User-Chat

Zur Implementierung der Chat Funktion wird die *XMPP* Erweiterung *XEP-0045* benutzt. Diese Erweiterung ermöglicht es den Benutzern Textnachrichten in Chaträumen auszutauschen, vergleichbar zum *Internet Relay Chat (IRC)*. Diese Erweiterung ermöglicht eine umfassende Steuerung eines Chatraumes, wie z. B. das Setzen von Überschriften, das Vergeben von Rechten an verschiedene Benutzer oder das Hinauswerfen oder Bannen von unerwünschten Chatteilnehmern (SA)(PSA09). Für die Implementierung der Chatfunktion werden dabei allerdings nur wenige der Funktionen dieser Erweiterung benutzt (u.a. das Erstellen eines neuen privaten passwortgeschütztem Chatraums).

1.4.4 XEP-0166: Jingle

Die *Jingle XMPP* Erweiterung dient zum einen dem Aufbau einer *Peer-to-Peer (P2P)* Verbindung zwischen zwei Teilnehmern, also einer direkten Verbindung zwischen den Benutzern ohne Umweg über den *XMPP* Server

und zum anderen der Übertragung von Medien (Audio, Video, Dateien) (SALB$^+$) (PSA09). Dies hat den wesentlichen Vorteil, dass keine Bandbreitenbegrenzung durch den Server besteht. Allerdings ist eine solche Verbindung über *Jingle* aufgrund von z. B. einer Firewall nicht immer möglich. Eine *P2P* Verbindung ist besonders für Streams, wie *VoIP* oder *Screensharing*, aber auch für die Übertragung von Dateien besonders hilfreich, da diese Anwendungen eine große Bandbreite benötigen.
In *Saros* wird immer versucht eine *Jingle* Verbindung zu benutzen, da z. B. zu Beginn einer *Saros* Sitzung alle Projektdaten übertragen werden müssen, welche sehr groß werden können. Falls keine *Jingle* Verbindung aufgebaut werden kann benutzt *Saros* eine *IBB (In-Band Bytestreams*; definiert in *XEP-0047*) Verbindung zur Übertragung der Daten. *IBB* ist ein einfacher Bytestream, der die Daten über den *XMPP Server* überträgt. *IBB* ist im Vergleich zu *Jingle* wesentlich langsamer, woraus folgt, dass ohne eine *Jingle* Verbindung das Übertragen von großen Dateien sehr lange dauert und das Benutzen einer *VoIP* oder *Screensharing* Funktion nahezu unmöglich macht.
Jingle bietet schon eine eingebaute Möglichkeit Audiostreams, also eine *VoIP* Sitzung, zu übertragen. Leider hat das Testen diverser *XMPP* Clients wie z. B. *SIP-Communicator* oder *Spark* gezeigt, dass es dort sehr oft zu Problemen beim Verbindungsaufbau kommt. So konnte ich keine *VoIP* Sitzung zwischen zwei Clients starten, trotz gleichem lokalen Netzwerk und richtig konfigurierter Aufnahme- und Wiedergabegeräte. Der eine Client bot die Möglichkeit eine *VoIP* Verbindung zu starten, der andere Client hat dies jedoch ohne Fehlermeldung direkt abgelehnt. Dieser Client hatte keine Möglichkeit eine eigene *VoIP* Sitzung zu starten, da der entsprechende Button nicht vorhanden war. Aufgrund dieser Problematik benutze ich nicht die Funktion der *Jingle* Erweiterung, sondern greife auf den internen *Stream-Service* (Lau10) von *Saros* zurück.

2 Ziele

Das Ziel dieser Arbeit ist es, die Möglichkeit der Kommunikation der klassischen Paarprogrammierung auch in Saros zu implementieren. Wie im vorangegangenen Abschnitt beschrieben, unterstützt Saros zur Zeit noch keinerlei interne Kommunikationsmöglichkeiten. Es ist daher bisher bei der Benutzung von Saros notwendig auf externe Applikationen zur Kommunikation zurückzugreifen. Im Rahmen meiner Bachelorarbeit zeige ich Lösungen zur Implementierung von Sprach- und Textkommunikation in *Saros* auf.

2.1 Chat

Das Ziel der Chatfunktion ist, eine einfache textbasierte Kommunikationsmöglichkeit in Saros zu implementieren. Es sollte also möglich sein, dass alle Sitzungsteilnehmer in einem abgeschlossenen Chatraum miteinander kommunizieren können.

Funktionale Anforderungen

- Alle Benutzer einer Saros Sitzung können in einem separaten *Eclipse View* mit allen Benutzern dieser Sitzung chatten. Dies ist unabhängig von der Rolle des Benutzers.
- Jedem Benutzer steht es frei, wann er den Chatraum betritt oder verlässt.
- Ein Chatraum für alle Sitzungsteilnehmer.
- Der Chatraum soll für andere Benutzer (außerhalb dieser Saros Sitzung) unsichtbar bzw. nicht ohne weitere Kenntnisse über Chatraumnamen und Passwort betretbar sein. Es soll jedoch die Möglichkeit geben, dass auch andere Benutzer (ohne Saros) diesen Chatraum betreten können (durch Kenntnis des Chatraumnamens und des entsprechenden Passwortes).
- Der Name des Chatraums soll vom *Host* einstellbar sein (um diesen auch weitergeben zu können). Andernfalls ist der Name des Chatraums gleich der *SessionID* der aktuellen Sitzung.
- Das Passwort des Chatraums soll auch vom *Host* einstellbar sein.
- Es sollen immer die Einstellungen des *Hosts* benutzt werden, die während des Einladungsprozesses zu Beginn einer neuen *Saros* Sitzung übertragen werden.
- Das bisherige Gespräch soll für jeden neuen Chatteilnehmer nachvollziehbar sein, d. h. jeder neue Teilnehmer erhält die komplette Historie des Chats der aktuellen Sitzung.

- Es soll sichtbar sein, welcher Benutzer welche Nachrichten zu welchem Zeitpunkt geschrieben hat.
- Der Chat soll unabhängig von verwendeten Betriebssystem benutzbar sein.

2.2 VoIP

Das Ziel der VoIP Funktion ist es, dass Benutzer, wie auch während der klassischen Paarprogrammierung, miteinander sprechen können. Dies hat gegenüber der Chatfunktion den Vorteil, dass die Kommunikation wesentlich schneller ist und parallel zur Programmierung erfolgen kann.

2.2.1 Funktionale Anforderungen

- Alle Benutzer einer *Saros* Sitzung können eine neue *VoIP* Sitzung erstellen. Dies soll unabhängig von ihrer aktuellen Rolle möglich sein.
- Der zu einer neuen *VoIP* Sitzung eingeladene Benutzer soll die Möglichkeit besitzen die Einladung anzunehmen oder abzulehnen.
- An einer *VoIP* Sitzung können zwei Personen teilnehmen, wie bei einem normalen Telefongespräch.
- Beide Benutzer können das Gespräch zu jeder Zeit beenden.
- Es ist möglich eine *VoIP* Sitzung ohne Aufnahmegerät zu starten, sodass ein einseitiges Gespräch entstehen kann. Es wird jedoch derjenige Benutzer gewarnt, dessen Aufnahmegerät nicht verfügbar oder nicht richtig konfiguriert ist.
- Das Starten einer *VoIP* Sitzung ohne Wiedergabegerät soll nicht möglich sein.
- Jeder Benutzer soll die Möglichkeit haben die wichtigsten Audio Encoder Einstellungen selbst konfigurieren zu können.
- Jeder Benutzer kann das Aufnahme- und Wiedergabegerät auswählen, welches er gerne benutzen möchte.
- Falls ein Benutzer einen anderen Benutzer zu einem neuen Gespräch einlädt, der schon in einer *VoIP* Sitzung aktiv ist, erhält dieser sofort eine Ablehnung seiner Anfrage.
- Unabhängigkeit von dem verwendetem Betriebssystem.

2.2.2 Nicht funktionale Anforderungen

- Die Verzögerung zwischen Spracheingabe des Senders und Sprachausgabe beim Empfänger sollte nicht mehr als 500 ms betragen, um ein flüssiges Gespräch gewährleisten zu können.
- Die Sprachausgabe soll nicht stottern oder ruckeln.

3 Implementierung

3.1 Chat

Im folgenden Abschnitt werde ich die genaue Umsetzung der Chatfunktion beschreiben. Im ersten Abschnitt werde ich kurz auf die schon bestehenden Funktionen eingehen, die ich weiterentwickelt habe; im weiteren Verlauf auf die Funktionen und die Bedienbarkeit dieser Erweiterung.

3.1.1 Bisherige Nutzung

Die bisherige Chatfunktion benutzt die in der *Smack API* implementierten *XEP-0045: Multi-User-Chat* Erweiterung des *XMPP*. Dabei existiert das Problem, dass jeder Benutzer den Chatraum betreten kann und alle Benutzer von *Saros* sich im gleichen Chatraum befinden. Es ist also möglich mit anderen unbekannten *Saros* Benutzern, in anderen *Saros* Sitzungen zu chatten. Die bisherige Implementierung ist aufgrund dieser Problematik allerdings noch in keinem *Saros* Release enthalten.

3.1.2 Bedienbarkeit

Um den Chat zu benutzen erhält der Benutzer einen separaten *Eclipse View*, der im weiteren Verlauf *Chat View* genannt wird (Abbildung 1). Dieser kann je nach Bedarf ein- oder ausgeblendet werden (*in Eclipse Window* →*Show View* →*Saros* →*Chat View*).

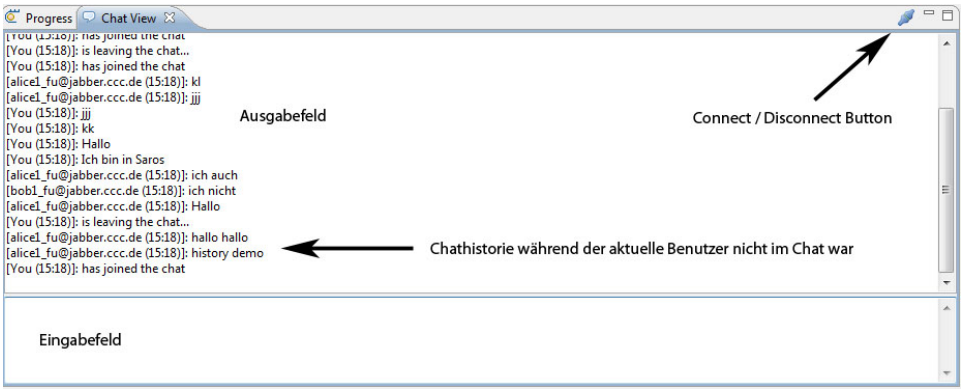

Abbildung 1: Chat View

Jeder Benutzer kann nun in seiner Eclipse Umgebung, nachdem er sich in einer *Saros* Sitzung befindet, in den Chat View wechseln und den Chatraum mithilfe des *Connect Buttons* betreten. Um einen Chatraum betreten zu können, müssen die Einstellungen für die Chatfunktion des *Sitzungs Hosts* gültig sein, da immer nur die Einstellungen des *Hosts* für alle Sitzungsteilnehmer verwendet werden. Der Host muss einen gültigen Chatserver ange-

geben haben (Abbildung 2), sonst ist es keinem Sitzungsteilnehmer möglich den Chatraum zu betreten oder einen neuen zu erzeugen. Als Standardeinstellung benutzt die *Saros* Chat Erweiterung *conference.jabber.ccc.de*.

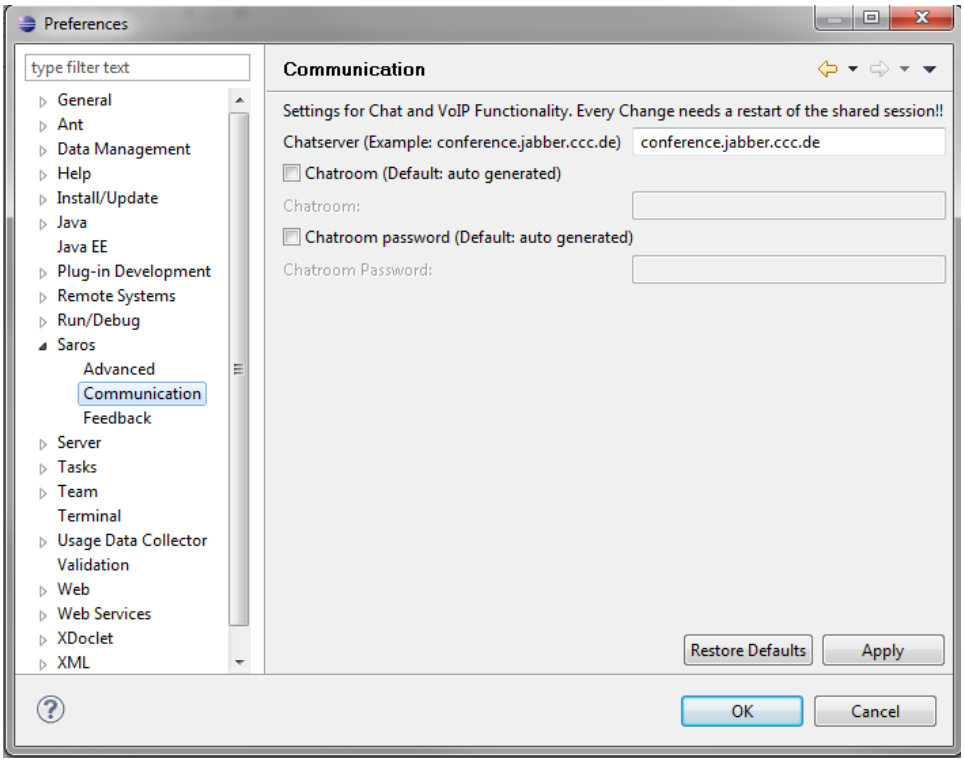

Abbildung 2: Einstellungsseite der Chat Funktion

Außerdem hat der *Host* die Möglichkeit einen benutzerdefinierten Namen für den Chatraum oder ein eigenes Passwort zu vergeben, um z. B. anderen Benutzern außerhalb der aktuellen *Saros* Sitzung diese Daten zu geben, damit diese den Chatraum betreten können. Diese Möglichkeit existiert mithilfe jedes beliebigen *XMPP* Clienten, der die Multi User Chat Funktion (*XEP-0045: Multi-User-Chat*) unterstützt (z. B. *SIP-Communicator*) (Abbildung 3).
Sobald der erste Benutzer einer Sitzung nun in den *Chat View* wechselt und auf den *Connect* Button klickt, wird auf dem Chatserver ein neuer Chatraum erstellt. Ohne weitere Einstellung des *Hosts* ist der Name des Chatraums gleich der SessionID (z. B. *SAROS1930696887*) und ist durch ein pseudo-zufälliges Passwort geschützt (z. B. *-630795124*). Die Generierung des Passwort geschieht durch einen Zufallsgenerator:

```
randomPassword = new Random().nextInt() + "";
```

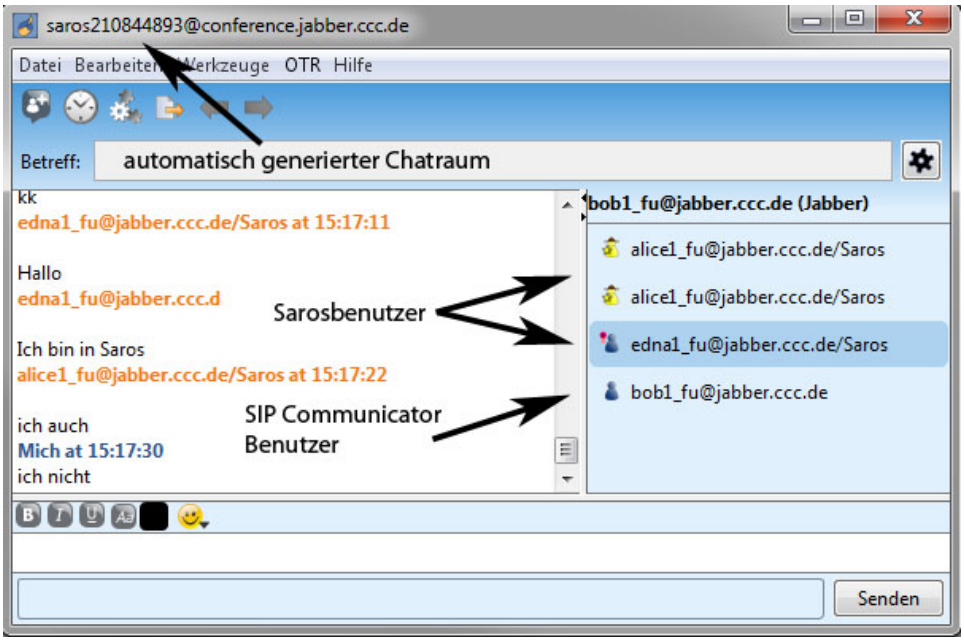

Abbildung 3: Multi-User-Chat im SIP Communicator mit Usern, die sich gerade in einer Saros Sitzung befinden.

Der Chatraum wird mit folgenden Einstellungen gestartet:

```
// Jeder im Chatraum kann Nachrichten absenden, da dieser
    nicht moderiert ist.
submitForm.setAnswer("muc#roomconfig_moderatedroom",
   false);

// Der Chatraum ist kein öffentlicher Raum, sodass dieser
    in keiner Raumliste eines XMPP Clienten aufgelistet
   wird
submitForm.setAnswer("muc#roomconfig_publicroom", false);

// Der Chatraum ist durch ein Passwort geschützt.
submitForm.setAnswer("muc#
   roomconfig_passwordprotectedroom", true);

// Setzen des Passwortes.
submitForm.setAnswer("muc#roomconfig_roomsecret",
                this.comPrefs.password);

// Es können keine Benutzer eingeladen werden, d.h. jeder
    Benutzer muss selbst mithilfe des Namens und des
   Passwortes des Chatraumes den Raum betreten.
submitForm.setAnswer("muc#roomconfig_allowinvites", false
```

```
    );

// Der Chatraum besteht nicht dauerhaft. Sobald alle den
    Raum verlassen, existiert dieser nicht mehr.
submitForm.setAnswer("muc#roomconfig_persistentroom",
    false);
```

Nun können andere Benutzer dieser *Saros* Sitzung diesen Chatraum betreten und erhalten beim Betreten des Chatraums die komplette Historie, um nachvollziehen zu können was bisher in diesem Chatraum diskutiert wurde. Weiterhin wird automatisch eine Nachricht *has joined the Chat* an den Chatraum gesendet, sodass alle Teilnehmer darüber informiert sind, dass ein neuer Benutzer den Chatraum betreten hat. Beim Verlassen des Raumes wird eine *is leaving the chat...* Nachricht gesendet. Sobald der Benutzer erfolgreich den Raum betreten hat kann dieser seine Textnachrichten in das Eingabefeld eintippen und diese durch das Drücken der *Enter Taste* abschicken. Die anderen Benutzer erhalten daraufhin die abgeschickte Nachricht in folgendem Format:

$[Absendername(12:00)] : Nachricht$

Der angezeigte Absendername ist der, den der Empfänger diesem Benutzer zugeordnet hat (in *Saros* ist dies im *Roster View* möglich), andernfalls wird die vollständige *Jabber ID* angezeigt. Falls man selbst der Sender der Nachricht ist, so wird als Absendername *You* angezeigt. Auf den Absendernamen folgt ein Zeitstempel der Form *HH:MM* und anschließend die Nachricht.

Da alle über Saros erstellten Chaträume nicht als *persistentroom* (permanenter Raum) erstellt werden, wird der Raum geschlossen sobald alle Benutzer den Chatraum verlassen haben. Dies passiert entweder manuell durch jeden Benutzer, indem diese den *Disconnect* Button im *Chat View* drücken, oder durch das Beenden der aktuellen Saros Sitzung durch den *Host*. Sobald der Raum geschlossen wurde, wird auch die Historie, also das bisherige Gespräch, auf dem Chatserver gelöscht und sie wird nicht mehr angezeigt, sobald man den Chatraum wieder betritt.

3.1.3 Sicherheit

Die Sicherheit der Chatfunktion basiert auf der Verbindung die in *Saros* zum Austausch der Nachrichten benutzt wird. Da *Saros* eine normale *XMPP* und keine *SSL (Secure Socket Layer) XMPP* Verbindung verwendet, benutzt die Chatfunktion dementsprechend auch keine *SSL* Verbindung zum Austausch der Nachrichten, d. h. es wäre theoretisch möglich dass alle Daten durch unbefugte Dritte mitgelesen werden, da die Verbindung nicht verschlüsselt ist. Eine verschlüsselte Verbindung zwischen zwei Teilnehmern bietet eine *SSL XMPP* Verbindung, die jedoch nicht von jedem *XMPP* Server unterstützt

wird. Da jedoch davon auszugehen ist, dass über diese Chatfunktion keinerlei sensible Daten übertragen werden, die unter allen Umständen vor unbefugten Dritten geschützt werden müssen, sollte die Benutzung der einfachen *XMPP* Verbindung von *Saros* ausreichend sein.

Der einzige Schutz den der Chat bietet, ist der Schutz des Chatraums durch ein Passwort, welches entweder durch den Benutzer selbst bestimmt oder durch *Saros* automatisch generiert wird.

3.1.4 Erweiterungsmöglichkeiten

Im Folgenden möchte ich einige, bisher noch nicht implementierte, Erweiterungen der Chatfunktion in *Saros* vorstellen.

- Erstellung mehrerer, paralleler Chaträume pro *Saros* Sitzung bzw. Teilnahme an mehreren Chaträumen gleichzeitig.
- Einladung weiterer Teilnehmer durch den Sitzungs *Host*, auch solcher außerhalb einer *Saros* Sitzung.
- Möglichkeit des *Hosts* Chatteilnehmer aus dem Chatroom zu entfernen
- Funktion des Stummschaltens von Teilnehmer zu Teilnehmer, so dass dieser die Nachrichten des Teilnehmers nicht mehr weiter lesen muss.
- Manuelle Erstellung eines privaten Chats, auch mit Teilnehmern außerhalb von *Saros*.
- Erweiterung des Chats um eine Internet Relay Chat (IRC) Schnittstelle.
- Möglichkeit den kompletten Chat auf dem lokalen Rechner eines Benutzers zu loggen und zu speichern.
- Entwicklung einer Funktion, die es ermöglicht die Einstellungen während einer laufenden Saros Sitzung zu verändern, ohne dass das Neustarten der kompletten Sitzung erforderlich wird.

3.2 VoIP

In diesem Abschnitt werde ich auf die Umsetzung und Bedienbarkeit der *VoIP* Funktion (*VoIP*) in *Saros* eingehen. *VoIP* beschreibt die Möglichkeit ein Telefonat über ein Computernetzwerk (z. B. das Internet) zu führen. Die wesentliche Vorteil an *VoIP* ist, dass keine weiteren Telefonkosten wie bei einem normalen Telefonat entstehen, sondern nur die Internetkosten (oder im lokalen Netzwerk keinerlei Kosten) anfallen. Da *Saros* schon eine Internetverbindung voraussetzt können die Internetkosten in diesem Fall vernachlässigt werden.

Zuerst werde ich auf die Funktionen und auf die Bedienung, im Weiteren Verlauf dann auf die Wahl des richtigen Audio Codecs und die genaue Implementierung eingehen, welche ich an ausgewählten Codesegmenten näher erläutern werde.

3.2.1 Funktionen

Mithilfe der *VoIP* Funktion können sich zwei Saros Benutzer, die sich in der gleichen *Saros* Sitzung befinden, unterhalten. Dies hat den Vorteil, dass man in *Saros* der klassischen Paarprogrammierung wesentlich näher kommt, da es möglich ist schnell und unkompliziert Probleme zu diskutieren und zu lösen ohne den Umweg über einen textbasierten Chat oder über externe Applikationen gehen zu müssen.

Voraussetzung zur Benutzung dieser Funktion ist es, dass beide Benutzer ein Tonwiedergabegerät besitzen und mindestens ein Benutzer ein Aufnahmegerät besitzt (z. B. ein Headset). Andernfalls wäre eine *VoIP* Sitzung sinnlos, da keiner der Teilnehmer etwas sagen kann. Optimal ist es, wenn beide Benutzer ein Aufnahme- und Wiedergabegerät besitzen, um eine beidseitige Kommunikation zu ermöglichen. Jedem Benutzer ist es möglich einen anderen Benutzer der aktuellen *Saros* Sitzung anzurufen, unabhängig von seiner aktuellen Rolle in der Sitzung (*Driver, Observer*). Falls ein Benutzer jemanden einladen möchte, der sich schon in einer *VoIP* Sitzung befindet, so erhält dieser Benutzer eine Ablehnung seiner Einladung. Jedem Benutzer einer *VoIP* Sitzung ist es möglich das Gespräch zu jedem beliebigen Zeitpunkt zu beenden. Beide Benutzer erhalten daraufhin eine Nachricht, die sie über die Beendigung des Telefonates informiert. Derzeit ist die Kommunikation mit Benutzern außerhalb von *Saros* nicht möglich, da die *VoIP* Funktion nur den internen *Saros StreamService* (Kapitel 3.2.7) benutzt.

3.2.2 Bedienbarkeit

Im Verlauf dieses Abschnittes werde ich die Benutzung der *VoIP* Funktion beschreiben, wobei alle nötigen Voraussetzungen erfüllt sind und beide teilnehmenden Benutzer alles richtig konfiguriert haben. Das Ziel ist die erfolgreiche Erstellung und das anschließende Beenden einer *VoIP* Sitzung. Die unterschiedlichen Einstellungen werde ich in einem späteren Abschnitt (3.2.9) beschreiben. Auf eventuelle auftretende Fehler und deren Behandlung wird im nächsten Abschnitt (3.2.3) genauer eingegangen.
Damit eine *VoIP* Sitzung gestartet werden kann, müssen sich mindestens zwei Benutzer in einer *Saros* Sitzung befinden. Nun wechselt einer der beiden Benutzer in den *Session View* des *Saros* Plugins, markiert den Benutzer den er gerne zu einem *VoIP* Telefonat einladen möchte und drückt anschließend den *VoIP Einladungs Button* (Abbildung 4).

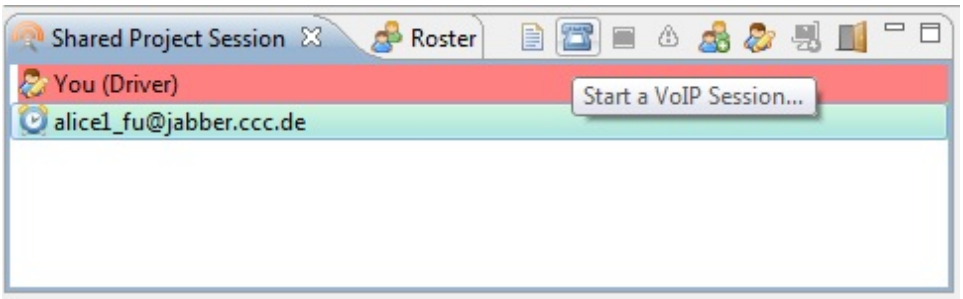

Abbildung 4: Starten einer neuen VoIP Sitzung

Der eingeladene Benutzer erhält nun ein *Ja-Nein* Dialogfenster, um die Einladung entweder anzunehmen oder abzulehnen (Abbildung 5).

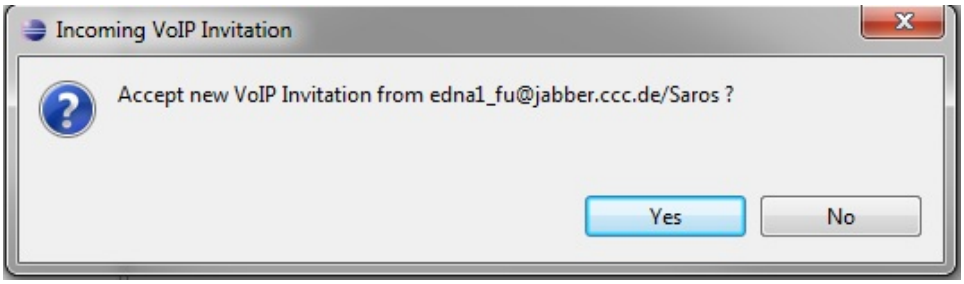

Abbildung 5: Einladungsdialog zu einer neuen VoIP Sitzung

In diesem Beispiel klickt der eingeladene Benutzer auf *Yes*. Die Sitzung wird somit gestartet und beide Benutzer können sich miteinander unterhalten.

Ein Sequenzdiagramm zum erfolgreichen Aufbau einer *VoIP* Sitzung ist in Abbildung 10 zu sehen.

Beide Teilnehmer können das Telefonat zu jedem Zeitpunkt beenden, indem sie in den *Session View* wechseln und den *VoIP Stop* Button anklicken (Abbildung 6).
Beide Gesprächsteilnehmer erhalten daraufhin eine Meldung, dass die *VoIP* Sitzung beendet wurde. Nun ist es ihnen wieder möglich eine neue Sitzung zu starten.

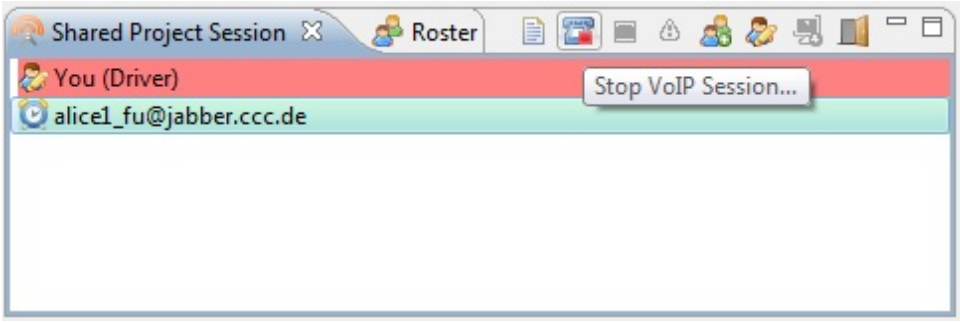

Abbildung 6: Stoppen einer VoIP Sitzung

3.2.3 Problemfälle bei der Benutzung

Im Folgenden werde ich auf Problemfälle und deren Behandlung bei der Benutzung der *VoIP* Funktion eingehen.

- Der einladende Benutzer hat kein für *Saros* verfügbares Wiedergabegerät in den *VoIP* Einstellungen ausgewählt.

  → Es kommt keine neue Sitzung zustande. Der Benutzer erhält eine Fehlermeldung (Abbildung 7), dass sein Wiedergabegerät nicht richtig konfiguriert ist und keine Sitzung gestartet werden kann. Der Benutzer, der eingeladen werden sollte erhält keinerlei Meldung.

- Der eingeladene Benutzer hat kein für *Saros* verfügbares Wiedergabegerät in den *VoIP* Einstellungen ausgewählt.

  → Es kommt keine neue Sitzung zustande. Der einladende Benutzer erhält eine Ablehnung seiner Einladung und der eingeladene Benutzer erhält eine Fehlermeldung (Abbildung 7), dass sein Wiedergabegerät nicht richtig konfiguriert ist und keine neue *VoIP* Sitzung gestartet werden kann.

Abbildung 7: Fehlermeldung über ein fehlendes oder falsch konfiguriertes Wiedergabegerät.

- Der *einladende* Benutzer hat kein für *Saros* verfügbares Aufnahmegerät in den *VoIP* Einstellungen ausgewählt.

  → Es kann eine neue Sitzung gestartet werden. Der einladende Benutzer erhält allerdings eine Warnung (Abbildung 8), dass sein Aufnahmegerät nicht richtig konfiguriert ist, die Sitzung dennoch gestartet werden kann. Er wird außerdem darauf hingewiesen, dass, wenn der andere Benutzer ebenfalls kein Aufnahmegerät konfiguriert hat, die erzeugte Sitzung sinnlos werden kann. Der eingeladene Benutzer erhält wie üblich das *Ja - Nein* Dialogfenster, um die Einladung anzunehmen oder abzulehnen. Er erhält keine Informationen darüber, dass der andere Benutzer kein funktionierendes Aufnahmegerät hat.

- Der *eingeladene* Benutzer hat kein für *Saros* verfügbares Aufnahmegerät in den *VoIP* Einstellungen ausgewählt.

  → Es kann eine neue Sitzung gestartet werden. Der eingeladene Benutzer erhält allerdings vor dem *Ja-Nein* Dialogfenster eine Warnung, dass sein Aufnahmegerät nicht richtig konfiguriert ist, die Sitzung dennoch gestartet werden kann. Er wird außerdem darauf hingewiesen, wenn der andere Benutzer ebenfalls kein Aufnahmegerät konfiguriert hat, die erzeugte Sitzung sinnlos werden kann. Der einladende Benutzer wird nicht über das fehlende Aufnahmegerät des anderen informiert.

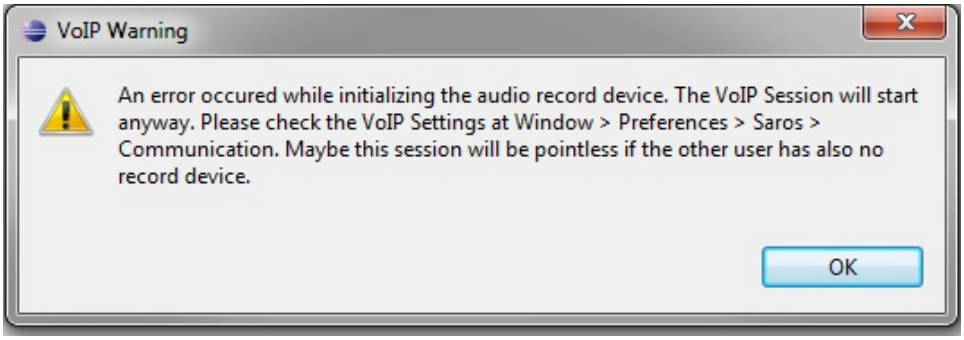

Abbildung 8: Warnmeldung über fehlendes oder falsch konfiguriertes Aufnahmegeräte

- Einer der Teilnehmer einer *VoIP* Sitzung verlässt die *Saros* Sitzung.

  → Die *VoIP* Sitzung wird beendet. Beide Teilnehmer erhalten die Meldung, dass die Sitzung beendet wurde.

- Es besteht keine *Jingle* Verbindung zwischen den Teilnehmern.

→ Es ist davon auszugehen, dass die Audiowiedergabe anfängt zu stottern. Dies kann nur dadurch behoben werden, indem die *VoIP* Sitzung beendet, beide Benutzer ihre *VoIP* Einstellungen anpassen und die Sitzung neu gestartet wird.

- Der einladende Benutzer klickt aus Versehen mehrmals auf den *Start Button*.

  → Es werden mehrere Einladungsprozesse gestartet. Der eingeladene Benutzer erhält allerdings gleichzeitig immer nur ein Einladungsdialog. Lehnt er das erste ab, so erscheint daraufhin ein neues Fenster. Sobald er eine Einladung angenommen hat, werden alle restlichen Einladungen automatisch abgelehnt und der einladende Benutzer erhält auch alle entsprechenden Meldungen, dass Einladungen von ihm abgelehnt wurden.

- Der eingeladene Benutzer reagiert nicht auf die Einladung.

  → Der einladende Benutzer erhält nach einer gewissen Zeit eine Timeout Meldung und der Einladungsprozess wird beendet. Der eingeladene Benutzer kann nach diesem Timeout die Einladung immer noch annehmen oder ablehnen. Das Fenster wird also nicht geschlossen. Falls er jedoch die Einladung nach einem Timeout noch annimmt wird die Session sofort mit einem Hinweisfenster *Session Error* wieder beendet. Es kommt keine neue *VoIP* Sitzung zustande.

- Einer der teilnehmenden Benutzer hat sehr schlechte Qualitätseinstellungen des Audio Encoders gewählt, sodass der andere Benutzer nur minderwertige Audio Daten erhält.

  → Die einzige Möglichkeit dies zu lösen besteht darin, dass der sendende Benutzer seine Qualitätseinstellungen verbessert. Die Wahl schlechter Qualitätseinstellungen muss jedoch möglich sein, um bei einer Verbindung mit wenig Bandbreite (z. B. *IBB*) die Funktion weiter benutzbar zu machen.

3.2.4 Wahl eines geeigneten Audio Codecs

Ein Audio Codec dient zur Komprimierung und Dekomprimierung von Audio Daten; bei wenig Qualitätsverlust soll eine möglichst hohe Komprimierbarkeit erreicht werden, um die benötigte Bandbreite so gering wie möglich zu halten. Im Folgenden werde ich mich nur mit für Sprachübertragung geeignete Audio Codecs beschäftigen, diese vergleichen und die Wahl meines Audio Codecs begründen.

- Die *Global System for Mobile Communication (GSM)* Codecs stammen aus der Sprachübertragung in *GSM* Netzen. GSM wird für die Mobilkommunikation genutzt, also für die Sprach- und Datenübertragung zwischen Mobiltelefonen. Da das *GSM* Netz eine im Vergleich zur heute üblichen Internetanbindung geringe Bandbreite pro Teilnehmer bereitstellt, ist eine besonders starke Kompression notwendig. Im Laufe der Zeit wurde der Codec mehrfach verändert und verbessert, um eine bessere Qualität bei möglichst niedriger Bitrate zu erhalten. *GSM* benutzt je nach verwendetem Codec eine konstante Bitrate von 4.75 - 13 kbps und unterteilt die Sprachdaten in Pakete von 20 ms Länge. Ein *GSM* Kanal hat eine Kapazität von 22.8 kbps. So wird die restlich zur Verfügung stehende Bandbreite durch Bits zur Fehlerkorrektur aufgefüllt, um eine Bandbreite von 22.8 kbps zu erreichen oder der Codec benutzt 11.4 kbps (*Half Rate Codec*), wodurch zwei Gespräche gleichzeitig abgewickelt werden können. (Mes) Je nach *GSM* Codec und Bitrate ist die Fehlerkorrektur unterschiedlich gut. Da jedoch eine möglichst gute Sprachqualität erreicht werden soll, wäre die Wahl des 13 kbps *GSM* Codecs am sinnvollsten. Der *GSM* Codec ist im frei erhältlichen *Java Media Framework (JMF)* enthalten.

- Der *iLBC - internet Low Bitrate Codec* ist ein nur für *VoIP* entwickelter Codec und wird sonst an keiner anderen Stelle verwendet. *iLBC* kodiert die Sprachdaten in konstanter Bitrate in Paketen von 20 ms oder 30 ms. Je nachdem unterscheidet sich die Bitrate von 13.33 kbps (30 ms) oder 15.2 kbps (20 ms). Außerdem ist der *iLBC* während der Übertragung gegenüber Paketverlusten vergleichsweise robust. (iP)

- *G.711* ist ein Codec der sowohl im klassischen Telefonnetz (*ISDN*), als auch im IP Telefonie Bereich eingesetzt wird. Die Sprachdaten werden in 125 μs Pakete unterteilt und auf 8 Bit komprimiert. Daraus ergibt sich eine konstante Bitrate von 64 kpbs, wie sie auch in der Übertragung in einem *ISDN* Netz benutzt wird. Aufgrund verschiedener Steuerungsdaten vergrößert sich die Bitrate in der IP-Telefonie allerdings auf 80 - 128 kbps. (Krö)(Unia) Der *G.711* ist ebenfalls wie der *GSM* Codec im *JMF* enthalten.

- Der *G.729* Codec wird hauptsächlich in der IP-Telefonie eingesetzt. Es existieren unterschiedliche Varianten dieses Codecs, welche auch verschiedene Features unterstützen. So unterstützt eine Variante, der *G.729J*, als einziger eine variable Bitrate. Die vom Codec angebotenen Bitraten variieren zwischen 6.4 kbps, 8 kbps und 11.8 kbps, wobei 8 kbps von jeder *G.729* Variante unterstützt werden. (Unib)

- *Speex* ist ein frei verfügbarer Audio Codec zur Kodierung von Sprachdaten. Speex wird in diversen bekannten *VoIP* Tools benutzt wie z. B.

Asterisk, Mumble oder Teamspeak. Speex hat eine große Auswahl an Bitraten von 2 kbps bis hin zu 44 kbps. Mit *Speex* kann im Gegensatz zu anderen Sprachcodecs der Kompromiss zwischen Qualität und Bitrate durch einen *quality* Parameter (0 - 10, wobei 10 das Beste ist) selbst gesteuert werden. Die Bitrate ist also vom gewählten Qualitätsparameter abhängig. Weiterhin unterstützt *Speex* sowohl konstante als auch variable Bitraten. Dies ist besonders sinnvoll, da so die Bitrate an die Komplexität des Signals angepasst wird und damit Bandbreite gespart werden kann. Außerdem enthält *Speex* eine *Voice Activity Detection (VAD)*, d.h. dass der Codec bestimmte Hintergrundgeräusche oder Stille erkennt und dafür nur bestimmte Parameter speichert, welche während des Decodierens wieder in ein Hintergrundrauschen (sogenanntes *Komfortrauschen*) übersetzt werden. Zusätzlich zu *VAD* existiert eine Funktion namens *Discontinuous Transmission (DTX)*, welche es ermöglicht bei gleichbleibendem Hintergrundrauschen die Übertragung komplett einzustellen, wodurch wieder Bandbreite gespart werden kann. (Val07) Für *Speex* existiert eine eigene Java Bibliothek, die sogenannte *jSpeex*. (PG)

Ich habe mich für den *Speex* Algorithmus entschieden, da dieser zum einen die besten Einstellungsmöglichkeiten bietet, sodass der Benutzer je nach eigener Internetanbindung und *Saros* Verbindung (*IBB* oder *Jingle*) die Qualitätseinstellungen selbst anpassen kann und nicht an festgelegte Einstellungen gebunden ist. Als Standardeinstellungen verwendet die *VoIP* Funktion das Qualitätslevel 8 bei variabler Bitrate und aktiviertem *VAD* und *DTX*. Zum Anderen besitzt *Speex* eine eigene Java Bibliothek und ist vom Betriebssystem unabhängig, was bei dem *JMF* leider nicht der Fall ist, da *JMF* kein Mac Betriebssystem unterstützt. Ein weiterer Vorteil ist, dass der Benutzer keine weiteren Treiber, Codecs oder Applikationen installieren muss um die *VoIP* Funktion zu benutzen. Dies wäre beim *JMF* der Fall, da dieses Framework nicht in der *Java Runtime Environment (JRE)* enthalten ist, sondern zusätzlich installiert werden muss. Diese Vorteile des *Speex* Codecs haben mich dazu gebracht, diesen in meiner *VoIP* Funktion zu benutzen.

3.2.5 Aufbau der VoIP Funktion

In diesem Abschnitt werde ich mithilfe eines UML Diagramms (Abbildung 9) den Aufbau der *VoIP* Funktion beschreiben. Das Kernstück der *VoIP* Funktion ist der *AudioServiceManager*. Dieser steuert das Starten und Beenden der *AudioSenderRunnable* und *AudioReceiverRunnable*. Weiterhin wird die *PreferenceUtils* Klasse benötigt über die der *AudioServiceManager* die installierten Aufnahme- und Wiedergabegeräte erhält. Diese Geräte werden im weiteren Verlauf auch *Mixer* genannt. Außerdem enthält der *AudioServiceManager* das *AudioService* Objekt, auf welches genauer in Abschnitt

3.2.7 eingegangen wird, und einen *StreamSessionListener*, der den *AudioServiceManager* benachrichtigt, sobald eine *Audio Stream Session* beendet wurde.

Die *AudioSenderRunnable* Klasse ist für das Aufnehmen und Encodieren der Audiodaten zuständig. Das Aufnehmen der Audiodaten geschieht mithilfe der *AudioRecorder* Klasse, wobei vor dem Starten des Aufnehmens die *TargetDataLine* mit einem *Mixer* initialisiert werden muss. Dies geschieht durch den Aufruf der *setupSound(Mixer)* Methode. Anschließend wird die *TargetDataLine* geöffnet und es werden ab diesem Zeitpunkt alle Daten die über den entsprechenden *Mixer* gesendet werden, auf diese *Line* geschrieben. Die *TargetDataLine* ist Bestandteil der *javax.sound.sampled* Bibliothek und ist daher in jeder *JRE* ab Version 1.3 enthalten. Die Daten der *TargetDataLine* werden direkt an einen *AudioInputStream* übergeben, der wie ein normaler *InputStream* behandelt werden kann. Dieser *InputStream* kann nun in einen encodierten *InputStream* mithilfe der *Pcm2SpeexAudioInputStream* Klasse in einen encodierten InputStream transformiert werden. Die *Pcm2SpeexAudioInputStream* Klasse ist Bestandteil der *jSpeex* API und enthält den *Encoder*. Das *Encoder* Objekt kann durch die *getEncoder()* Methode erlangt werden, wodurch verschiedene Encoderparameter gesetzt werden können. Das *Pcm2SpeexAudioInputStream* Objekt ist Bestandteil der *AudioSenderRunnable*. Anschließend kann von dem encodierten *AudioInputStream* gelesen werden und die Daten können über einen kleinen Byte Buffer auf den *OutputStream* des *StreamService* geschrieben werden. Hier werden die Daten nun über eine *Jingle* Verbindung oder notfalls über eine *IBB* Verbindung versendet.

Die *AudioReceiverRunnable* Klasse ist für das Decodieren und Wiedergeben der empfangen Audio Daten zuständig. Das Decodieren der Daten geschieht mithilfe des Gegenstücks zur *Pcm2SpeexAudioInputStream* Klasse, der *Speex2PcmAudioInputStream* Klasse. Das *Speex2PcmAudioInputStream* Objekt erhält als Eingabe den vom *StreamService* bereitgestellten *InputStream*, in dem die encodierten Audio Daten stehen. Der *Speex2PcmAudioInputStream* kann wie ein *AudioInputStream* behandelt werden, der nun decodierte Audio Daten enthält und von dem diese gelesen werden können. Dieser Stream wird nun zu Beginn dem *AudioPlayer* übergeben, der die Daten in einem Byte Buffer zwischenspeichert und anschließend auf eine *SourceDataLine*, die zuvor mit einem entsprechenden *Mixer* initialisiert wurde, schreibt. Die *SourceDataLine* ist das Gegenstück zur *TargetDataLine*. Über diese können Audio Daten an einen Wiedergabemixer übergeben werden, der diese nun wiedergibt.

Die *PreferenceUtils* Klasse bietet diverse Methoden, um verschiedene Einstellungswerte abzufragen. Für die *VoIP* Funktion habe ich diese Klasse um vier Methoden erweitert, wobei die beiden für den *AudioServiceManager* wichtigen Methoden den aktuell vom Benutzer ausgewählten Aufnahme-

bzw. Wiedergabemixer zurückgeben. Weiterhin wird über die *PreferenceUtils* Klasse abgefragt, welches Encoding Format der Benutzer ausgewählt hat. Damit der Benutzer in den Einstellungen überhaupt einen *Mixer* auswählen kann, der auch über die *getRecordingMixer()* bzw. *getPlaybackMixer()* abgefragt werden kann, müssen zum Start von *Saros* erst alle auf dem aktuellen System verfügbaren *Mixer* abgefragt werden und in Wiedergabe- und Aufnahmemixer aufgeteilt werden. Dies geschieht durch den *MixerManager*.

Der *MixerManager* wird zum Start von *Saros* gestartet. Die *initalizeMixerInfo()* Methode fragt nun alle installierten *Mixer* ab und überprüft, ob der jeweilige *Mixer* eine *TargetDataLine* (→Aufnahmemixer) oder eine *SourceDataLine* (→Wiedergabemixer) unterstützt. Dementsprechend werden die *Mixer* in zwei *HashMaps* (*RecordDevices, PlaybackDevices*) einsortiert. Diese *HashMaps* benutzen als Schlüssel einen String, der den jeweiligen Mixer identifiziert und enthalten als Wert das *Mixer.Info* Objekt des *Mixers*. Der String dient zur Repräsentation des *Mixers* auf der Einstellungsseite der *VoIP* Funktion. Das *Mixer.Info* Objekt ist ein Objekt, das Informationen über den jeweiligen *Mixer* enthält und aus dem später ein *Mixer* Objekt erzeugt werden kann.

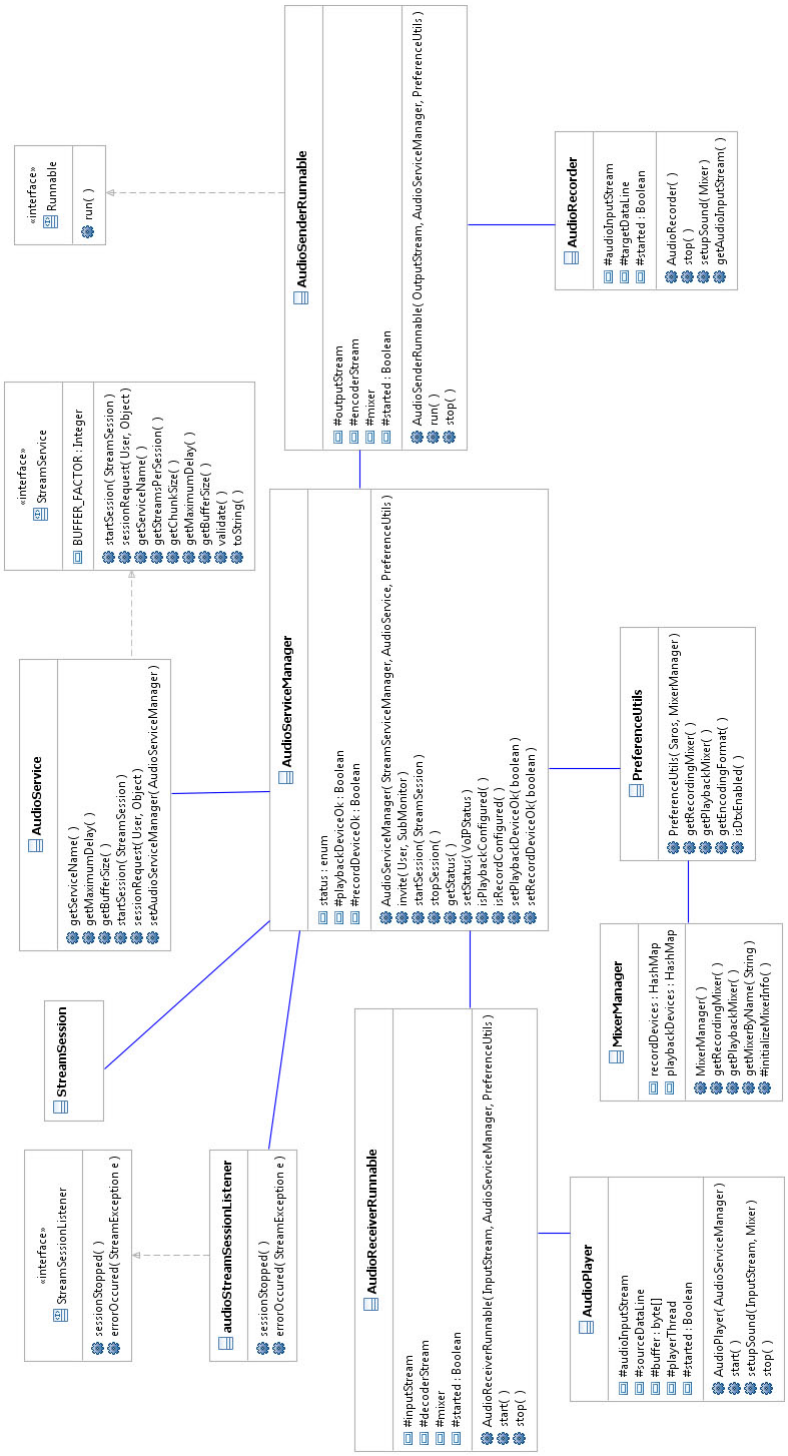

Abbildung 9: UML Diagramm der VoIP Funktion

3.2.6 Ablauf einer VoIP Sitzung

In diesem Abschnitt werde ich auf den Ablauf einer *VoIP* Sitzung mithilfe von diversen Diagrammen genauer eingehen.

Der erfolgreiche Aufbau einer *VoIP* Sitzung ist in dem Sequenzdiagramm in Abbildung 10 zu erkennen. User A und User B sind hierbei Akteure und keine Objekte, wie im Diagramm dargestellt. In diesem Diagramm ist mit dem anfänglichen Aufruf der *invite()* Funktion durch den Einladenden gemeint, dass der Benutzer auf den *Start new VoIP Session...* Button im *Session View* klickt. Der *Listener* für diesen Button ruft dann die *invite()* Funktion des *AudioServiceManagers* auf. Der abschließende *startSession()* Aufruf durch die *AudioServiceManager* Objekte beider Benutzer startet *Rekorder, Encoder, Decoder* und *Player*.
Den erfolgreichen Abbau einer Verbindung erkennt man in dem Sequenzdiagramm in Abbildung 11. Auch hier sind *User A* und *User B* keine Objekte sondern Akteure. *User B* drückt zu Beginn den *Stop VoIP Session* Button im *Session View*. Der entsprechende *Listener* ruft daraufhin die *stopSession()* Funktion des eigenen *AudioServiceManager* Objekts auf. Am Ende des Beendigungsvorgangs erhalten *User A* und *User B* einen Hinweisdialog, welcher durch den *STOP Dialog* Pfeil dargestellt wird.

Ein Zustandsdiagramm, welches die Zustände, die während einer *VoIP* Sitzung auftreten können, zeigt, ist in Abbildung 12 dargestellt.

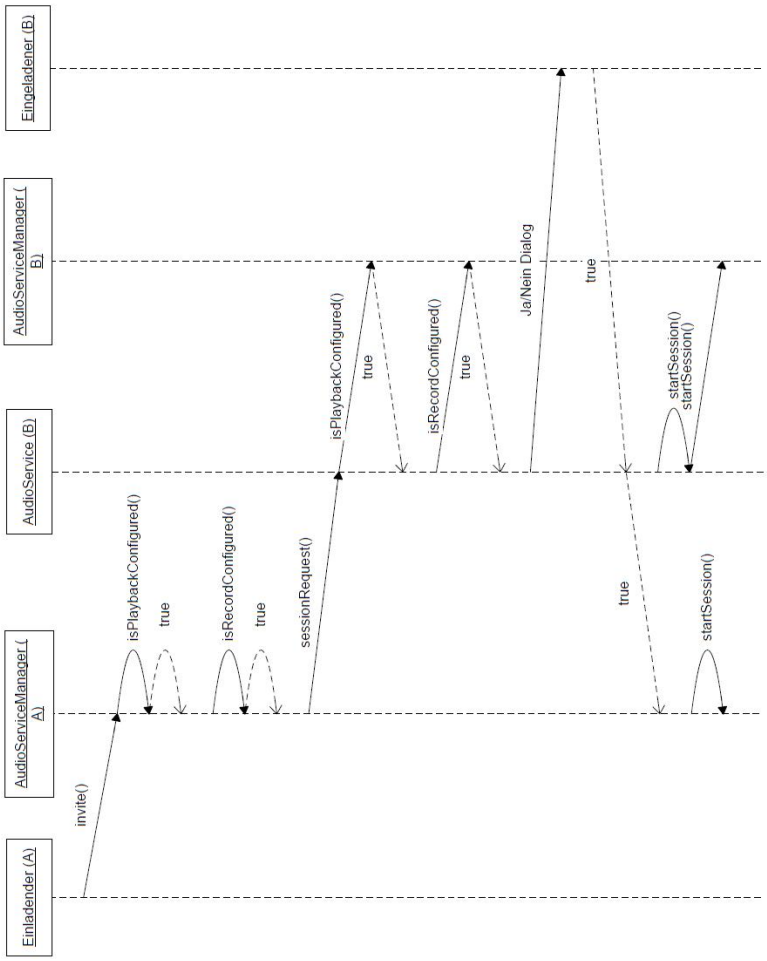

Abbildung 10: Erfolgreiches Aufbauen einer VoIP Sitzung als Sequenzdiagramm

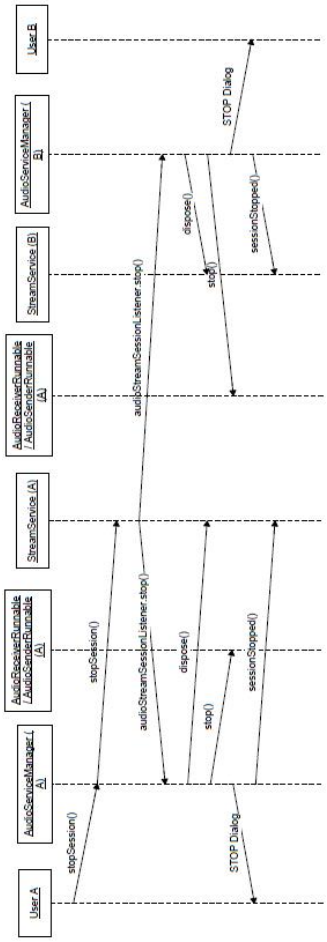

Abbildung 11: Erfolgreiches Beenden einer VoIP Sitzung als Sequenzdiagramm

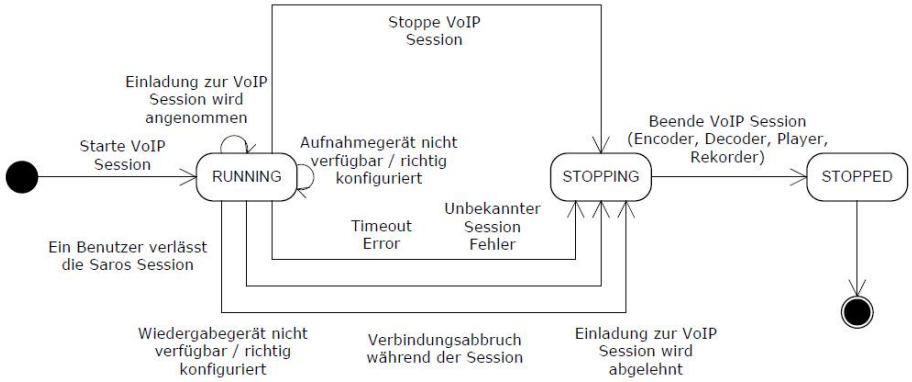

Abbildung 12: Zustandsdiagramm einer VoIP Session

3.2.7 Saros StreamService

Der *StreamService* in *Saros* wurde in einer Bachelorarbeit (Lau10) von Stephan Lau entwickelt. Er bietet die Möglichkeit über die bestehende *Saros* Verbindung (*Jingle* oder *IBB*) einfach Daten auszutauschen. Dies ist nützlich um eine *Screensharing*, *VoIP* oder eine einfache *Dateitransfer* Funktion zu implementieren. Dazu erhält jeder Service eine bestimmte Anzahl von normalen *Input/Outpustreampaaren*, die jeder Service selbst bestimmen kann. Für jeden Service muss dazu eine Klasse implementiert werden, die von *StreamService* erbt.

Für die *VoIP* Funktion sieht dieser Service folgendermaßen aus (Bemerkungen und Kommentare sind im Code zu finden):

```
public class AudioService extends StreamService {

    private static final Logger log = Logger.getLogger(
   AudioService.class);

    protected AudioServiceManager audioServiceManager;

   //Der AudioServiceManager wird im AudioService
   benötigt, um den aktuellen Status einer VoIP Sitzung
   abzufragen und bei einer Anfrage zu einer neuen VoIP
   Sitzung gegebenenfalls eine neue Sitzung zu starten.

    public AudioService(AudioServiceManager
   audioServiceManager) {
        this.audioServiceManager = audioServiceManager;
    }

   // Die chunkSize beschreibt die minimale Größe an
   Bytes, die pro Paket verschickt werden müssen. Da die
   Größe von Speex Frames 160 Byte beträgt, hat sich
   während des Testens eine chunkSize von vier Speex
   Frames als sinnvoll herausgestellt. Ein Array wird
   zurückgegeben, da für jedes Streampaar entschieden
   werden kann wie groß die chunkSize ist. Die Anzahl der
    Streams wird in dieser Klasse nicht überschrieben, da
    die Standardgröße des StreamService Eins ist und der
   AudioService auch nicht mehr als ein Input-/
   Ouputstreampaar benötigt.

    @Override
    public int[] getChunkSize() {
        return new int[] { 640 };
    }

   // Der serviceName (hier: AudioService) identifiziert
   diesen Service.
```

```
@Override
public String getServiceName() {
    return "AudioService";
}

// getMaximalDelay gibt die maximale Verzögerung, die
ein Stream erreichen darf, zurück. In meinem Fall habe
 ich an dieser Stelle 1000 ms gewählt, um
sicherzustellen, dass die Sitzung nicht ungewollt
beendet wird. Eine Unterhaltung mit 1000 ms ist nahezu
 unmöglich, da jeder Benutzer die gesprochene
Nachricht mit 1 s Verzögerung erhalten würde. Jedoch
kann so jeder Benutzer durch die Benutzung der
Funktion selbst herausfinden, ob die Verzögerung
subjektiv empfunden zu groß ist und er das Gespräch
beenden möchte. Ein Array wird zurückgegeben, da für
jedes Streampaar die Verzögerung einzeln konfiguriert
werden kann.

@Override
public long[] getMaximumDelay() {
    return new long[] { 1000 };
}

// getBufferSize gibt die Buffer Größe für jedes
Streampaar zurück. Diese Größe gibt an, wieviele Bytes
 maximal pro Paket übertragen werden können. Es werden
 also immer min. chunkSize Bytes und max. bufferSize
Bytes pro Paket übertragen. Die durch den
StreamService vorgegebene Größe ist 10 * chunkSize.
Daraus resultiert jedoch bei der VoIP Funktion eine
sehr große Verzögerung von mehr als einer Sekunde,
weswegen die VoIP Funktion Pakete von maximal nur 2 *
chunkSize Bytes zulässt.

@Override
public int[] getBufferSize() {
    return new int[] { getChunkSize()[0] * 2 };
}

// Die startSession Funktion wird automatisch
aufgerufen, sobald sessionRequest true zurückgibt.
Dies passiert sobald der eingeladene Benutzer die
Einladung über einen Ja - Nein Dialog annimmt.

@Override
public void startSession(StreamSession newSession) {
    audioServiceManager.startSession(newSession);
```

```
 }

// Die sessionRequest Funktion wird automatisch im
Client des Eingeladenen aufgerufen, sobald dieser eine
 neue Einladung erhält. Diese gibt entweder true (bei
akzeptierter Einladung) oder false zurück (bei
abgelehnter Einladung). Bei akzeptierter Einladung
wird die startSession Funktion aufgerufen.
// In dieser Methode wird überprüft ob sich der
Benutzer bereits in einer VoIP Sitzung befindet und ob
 sowohl das Wiedergabe- und Aufnahmegerät verfügbar
sind. Dementsprechend wird entweder direkt eine
Ablehnung an den Einladenden gesendet (falls sich der
Benutzer in einer VoIP Sitzung befindet oder das
Wiedergabegerät nicht richtig konfiguriert ist) oder
es wird ein Ja - Nein Dialog angezeigt (eventuell mit
zusätzlicher Warnung, falls der Benutzer kein richtig
konfiguriertes Aufnahmegerät besitzt).

 @Override
 public boolean sessionRequest(User from, Object
initial) {
     log.info(from + "wants to start a VoIP Session");

     // Is user already in a voip session and are the
devices properly
     // configured?
     if (audioServiceManager.getStatus() ==
AudioServiceManager.VoIPStatus.RUNNING)
         return false;

     if (!audioServiceManager.isPlaybackConfigured())
{
         EclipseUtils
             .openErrorMessageDialog(
                 EditorAPI.getShell(),
                 "VoIP: Playback device error",
                 from.getJID()
                     + "wants to start a VoIP session
with you but your playback device is not properly
configured. Please check the VoIP Settings at Window >
 Preferences > Saros > Communication. The VoIP session
 will NOT be started! Send reject packet.");
         return false;
     }

     if (!audioServiceManager.isRecordConfigured()) {
         return EclipseUtils
             .openQuestionMessageDialog(
```

```
                    EditorAPI.getShell(),
                    "Incoming VoIP Invitation",
                    "Accept new VoIP Invitation from "
                        + from.getJID()
                        + " ?\n Warning: Your record
device is not properly configured. Please check the
VoIP Settings at Window > Preferences > Saros >
Communication. The VoIP session can be started anyway.
 Maybe this session will be pointless if the other
user has also no record device.");
     } else {
         return EclipseUtils.openQuestionMessageDialog
(EditorAPI.getShell(),
             "Incoming VoIP Invitation", "Accept new
VoIP Invitation from "
                    + from.getJID() + " ?");
     }

 }

}
```

3.2.8 Erweiterungsmöglichkeiten

Nun werde ich noch nicht implementierte Erweiterungen der *VoIP* Funktion vorstellen.

- Teilnahme von mehr als zwei Benutzern an einer *VoIP* Sitzung.
- Mehrere *VoIP* Räume, sodass mehrere Benutzer in verschiedenen Räumen unabhängig voneinander kommunizieren können.
- Die Encodereinstellungen sollten nicht für jeden Benutzer individuell einstellbar sein, sondern für jeden kompletten Raum gelten und z. B. durch den *Host* bestimmt werden. Alternativ wäre eine automatische Anpassung der Encodereinstellungen an die verfügbare Bandbreite der Verbindungen zwischen den Teilnehmern denkbar.
- Implementierung weiterer Audio Codecs.

3.2.9 Einstellungsmöglichkeiten

In Folgendem Abschnitt werde ich auf die verschiedenen Einstellungsmöglichkeiten (Abbildung 13) und deren Auswirkungen auf die Audioqualität eingehen.

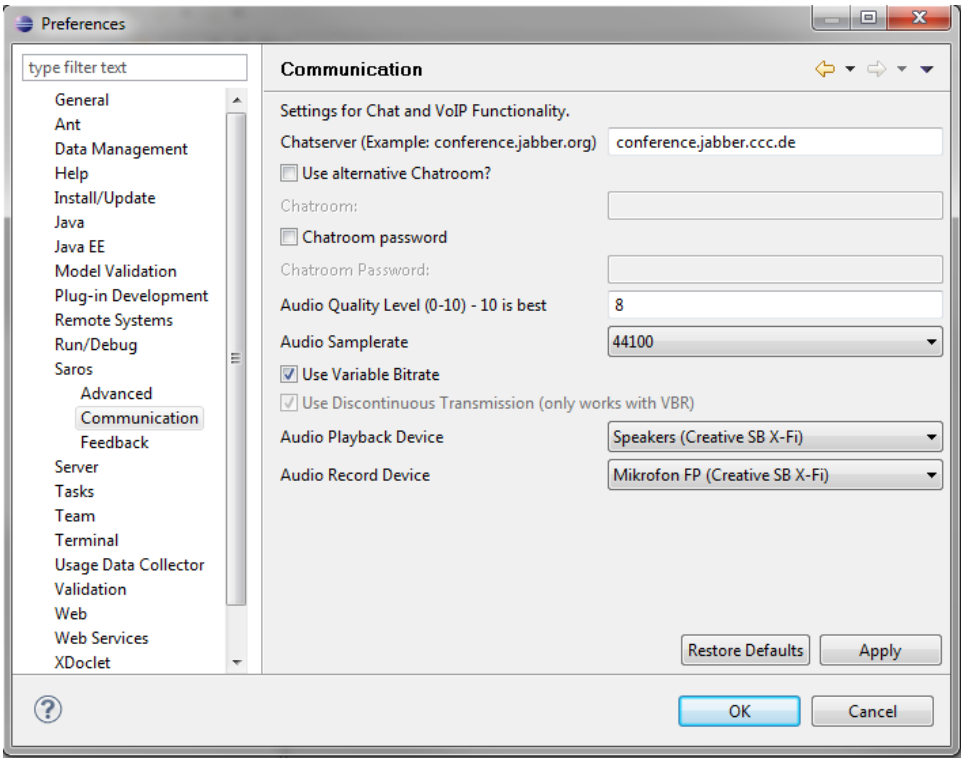

Abbildung 13: Einstellungsmöglichkeiten der VoIP Funktion

Audio Quality Level (0 - 10) - 10 is best Die *Audio Quality Level* Einstellung setzt den Qualitätsparameter des *Speex* Codecs. Dieser Parameter beschreibt den Kompromiss zwischen der Audio Qualität und der Bitrate. D.h. je höher dieser Parameter gesetzt wird, desto besser ist die Qualität und desto höher ist die Bitrate mit der die Audio Daten encodiert werden.

Audio Samplerate Die Samplerate beschreibt die Häufigkeit mit der ein kontinuierliches Signal in ein zeitliches Signal umgewandelt wird. Es beschreibt z. B. wieviele Bytes benötigt werden, um ein bestimmten Zeitabschnitt (hier 20 ms) zu encodieren. Bei 8 Khz werden 160 Byte je Speex Frame benötigt, bei 16 Khz bereits 320 Byte je Frame. Je höher die Samplerate desto weniger Details gehen während des Encodierens verloren. Die Samplerate beeinflusst auch die Bitrate. Wird die Samplerate angehoben, so steigt auch die Bitrate.

Use Variable Bitrate Diese Checkbox gibt dem Benutzer die Möglichkeit anstatt einer konstanten Bitrate eine variable Bitrate (*VBR*) zu benutzen. Dies hat den Vorteil, dass während des Encodierens die Bitrate der Komplexität des Audio Signals angepasst wird.

Use Discontinues Transmission Diese Checkbox ist nur auswählbar, falls der Benutzer einen Haken bei *Use Variable Bitrate* gemacht hat, da die *Discontinues Transmission* Option des *Speex* Codecs nur in Verbindung mit einer *VBR* Encodierung möglich ist. Mithilfe dieser Option wird das Rauschen nicht übertragen und die Bitrate sinkt währenddessen auf bis zu 0 kbps.

Audio Playback Device An dieser Stelle kann der Benutzer das Wiedergabegerät auswählen, dass er gerne für die *VoIP* Funktion benutzen möchte. Die Liste der hier auswählbaren Geräte wird zu Beginn vom *MixerManager* erzeugt.

Audio Record Device An dieser Stelle kann der Benutzer das Aufnahmegerät auswählen, dass er gerne für die *VoIP* Funktion benutzen möchte. Die Liste der hier auswählbaren Geräte wird zu Beginn vom *MixerManager* erzeugt.

4 Fazit

In dieser Arbeit wurde ein Eclipse Plugin zur verteilten Paarprogrammierung (*Saros*) um zwei Kommunikationsmöglichkeiten erweitert. Zum einen wurde eine Chat und zum anderen eine VoIP Funktion implementiert, die es den *Saros* Benutzern ermöglichen soll einfacher und näher an die klassische Paarprogrammierung heranzurücken, indem sie nun mit anderen Teilnehmern text- oder sprachbasiert kommunizieren können. Dies ist ein wesentlicher Bestandteil der klassischen Paarprogrammierung und wurde mit dieser Arbeit der *Saros* Erweiterung hinzugefügt, wobei es noch mehrere, bisher nicht implementierte Erweiterungsmöglichkeiten dieser Funktionen gibt.

Während der Arbeit mit *Saros* ist mir aufgefallen, dass viele verschiedene Studenten mit unterschiedlichen Arbeitsmethoden an diesem Projekt mitgewirkt haben. Dies ist besonders auffällig in der Ausführlichkeit der Dokumentation in- und außerhalb des Quellcodes. Daher fällt es schwer bestimmte Funktionen zu erweitern, da die Einarbeitung in den bestehenden Quellcode mehr Zeit in Anspruch nimmt als sie eigentlich müsste. Auch die Probleme bisheriger Implementierungen sind teilweise unklar oder nicht dokumentiert.

Die Implementierung der *VoIP* Funktion stellte sich zu Beginn aufgrund der Probleme der *Jingle* Erweiterung als schwierig heraus. Die Dokumentation der *Jingle* Erweiterung deutete auf ein einfaches Aufbauen eines Telefonats hin, weswegen ich längere Zeit versucht habe die vorhandenen Probleme zu umgehen. Die zeitgleiche Entwicklung des *StreamServices* durch *Stephan Lau* (Lau10) kam mir daher sehr entgegen, da ich diese Erweiterung nun anstelle von *Jingle* verwenden konnte. Zudem musste ich einen passenden Audiocodec wählen, wobei ich mich letztendlich für den *Speex* Codec entschieden habe. Die Implementierung der *VoIP* Funktion gestaltete sich schwieriger als zunächst angenommen, da mir längere Zeit eine stotternde Audiowiedergabe Probleme bereitete, die ich jedoch durch längeres Ausprobieren der Encodereinstellungen und Optimieren meines eigenen Codes beheben konnte. Da der *StreamService* parallel entwickelt wurde, musste mein *AudioService* mehrfach an einen veränderten *StreamService* angepasst werden; dank einer guten Kommunikation mit *Stephan Lau* bereitete mir diese Anpassung keinerlei größere Probleme.

Durch meine Implementierungen verfügt *Saros* nun über eine integrierte Funktion zur direkten Kommunikation zwischen den Benutzern und bietet damit eine verbesserte Simulation der klassischen Paarprogrammierung. Aufgrund dessen bietet *Saros* gegenüber anderen Eclipse Erweiterungen zur verteilten Paarprogrammierung verbesserte Kommunikationsmöglichkeiten.

A Anwendungsfälle

A.1 Chat

Betreten des Saros Chatraums (C01)

Vorbedingung: Der Benutzer befindet sich in einer Saros Sitzung.

Nachbedingung: Der Benutzer hat den Chatraum betreten und erhält im Chat View folgende Meldung:

[You (18:24)]: has joined the chat

Ablauf:

1. Der Benutzer öffnet den Chat View über Window →Show View →Other →Saros →Chat View
2. Der Benutzer wechselt in den Chat View
3. Der Benutzer klickt auf den Connect Button des Chat Views

Verlassen des Saros Chatraums (C02)

Vorbedingung: Der Benutzer befindet sich in einer Saros Sitzung und hat den Saros Chatraum betreten.

Nachbedingung: Der Benutzer hat den Chatraum verlassen und erhält im Chat View folgende Meldung:

[You (18:29)]: is leaving the chat...

Ablauf:

1. Der Benutzer wechselt in den Chat View
2. Der Benutzer klickt auf den Disconnect Button des Chat Views

Verlassen der Saros Sitzung während der Benutzer im Saros Chatraum ist (C03)

Vorbedingung: Der Benutzer befindet sich in einer Saros Sitzung und hat den Saros Chatraum betreten.

Nachbedingung: Der Benutzer hat die Saros Sitzung und den Chatraum verlassen

Ablauf:

1. Der Benutzer wechselt in den Shared Project Session View
2. Der Benutzer klickt auf den Leave Session Button

Der Host ändert die Einstellungen der Chat Funktion (C04)

Vorbedingung: Der Host hat noch keine Saros Sitzung gestartet.

Nachbedingung: Die neuen Einstellungen werden bei der nächsten Saros Sitzung angewendet.

Ablauf:

1. Der Host ruft die Einstellungsseite über Window →Preferences →Saros →Communication auf.
2. Der Benutzer ändert den Chatserver
3. Der Benutzer klickt auf OK

Alternativen:

2a1. Der Benutzer macht einen Haken bei *Use alternative Chatroom*

2a2. Der Benutzer gibt einen eigenen Namen für den Chatraum ein

2b1. Der Benutzer macht einen Haken bei *Chatroom password*

2b2. Der Benutzer gibt ein eigenes Passwort für den Chatraum ein

A.2 VoIP

Starten einer neuen VoIP Sitzung (V01)

Vorbedingung: Zwei Benutzer (Alice und Bob) befinden sich in einer Saros Sitzung

Nachbedingung: Alice und Bob sind in einer neuen VoIP Sitzung

Ablauf:

1. Alice wechselt in den Shared Project Session View
2. Alice markiert Bob im Shared Project Session View
3. Alice klickt auf den VoIP Start Button
4. Bob erhält eine Einladung zu einer neuen VoIP Sitzung von Alice
5. Bob klickt 'Yes'

Alternativen:

4a. Bob erhält zusätzlich zu der Einladung eine Warnmeldung, dass sein Aufnahmegerät nicht richtig konfiguriert ist

4b. Bob erhält eine Fehlermeldung, dass sein Wiedergabegerät nicht richtig konfiguriert ist

5b. Bob klickt auf 'OK'

4c1. Alice erhält eine Warnung, dass ihr Aufnahmegerät nicht richtig konfiguriert ist.

4c2. Alice klickt auf 'OK'

4c3. Bob erhält eine Einladung zu einer neuen VoIP Sitzung von Alice

5c. Bob klickt auf 'Yes'

4d1. Alice erhält eine Fehlermeldung, dass ihr Wiedergabegerät nicht richtig konfiguriert ist.

4d2. Alice klickt auf 'OK'

Beenden einer VoIP Sitzung (V02)

Vorbedingung: Zwei Benutzer (Alice und Bob) befinden sich in einer VoIP Sitzung

Nachbedingung: Alice und Bob haben die VoIP Sitzung verlassen

Ablauf:

1. Alice oder Bob wechseln in den Share Project Session View
2. Alice oder Bob klicken auf den VoIP Stop Button
3. Alice und Bob erhalten eine Nachricht, dass die VoIP Sitzung beendet wurde
4. Alice und Bob klicken auf OK

Ändern der Einstellung der VoIP Funktion (V03)

Vorbedingung: Der Benutzer (Alice) ist in keiner VoIP Sitzung

Nachbedingung: Die geänderten Einstellungen werden für die nächste VoIP Sitzung übernommen

Ablauf:

1. Alice ruft die Einstellungsseite über Window →Preferences →Saros →Communication auf.
2. Der Benutzer ändert den *Audio Quality Level*
3. Der Benutzer klickt auf 'OK'

Alternativen:

2a. Der Benutzer ändert die *Audio Samplerate*

2b1. Der Benutzer macht einen Haken bei *Use Variable Bitrate*

2b2. Der Benutzer macht einen Haken bei *User Discontinues Transmission*

2c. Der Benutzer ändert das *Audio Playback Device*

2d. Der Benutzer ändert das *Audio Record Device*

Einen Benutzer zu einer VoIP Sitzung einladen, der sich schon in einer aktiven VoIP Sitzung befindet (V04)

Vorbedingung: 2 Benutzer (Alice und Bob) sind in der gleichen Saros Sitzung. Alice befindet sich in einer aktiven VoIP Sitzung. Bob befindet sich in keiner aktiven VoIP Sitzung.

Nachbedingung: Alice befindet sich weiterhin in der aktiven VoIP Sitzung und Bob in keiner VoIP Sitzung

Ablauf:

1. Bob wechselt in den Shared Project Session View
2. Bob makiert Alice
3. Bob klickt auf den VoIP Start Button
4. Bob erhält ein Dialogfeld mit der direkten Ablehnung seiner Einladung
5. Bob klickt auf OK

B Anmerkungen

Zur sprachlichen Vereinfachung und Erhaltung der Übersichtlichkeit verzichte ich auf die gleichzeitige Nennung weiblicher und männlicher Begriffe, sondern verwende ausschließlich die männliche Form ohne das damit irgendeine Diskriminierung verbunden oder beabsichtigt ist.

Die von mir verwendete LaTeXVorlage für die Bachelorarbeit *Verbesserung der Kommunikationsmöglichkeiten in einem Werkzeug zur verteilten Paarprogrammierung (Saros)* habe ich aus der Bachelorarbeit von *Lisa Dohrmann* (Doh09) übernommen.

Literatur

[CW] Alistair Cockburn and Laurie Williams. The Costs and Benefits of Pair Programming. http://collaboration.csc.ncsu.edu/laurie/Papers/XPSardinia.PDF.

[Dje06] R. Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierungverteilter Paarprogrammierung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Djemili06-eclipse-erweiterung-PP.pdf, 08 2006.

[Doh09] Lisa Dohrmann. Erhebung von Benutzerfeedback aus der Nutzung eines Werkzeugs zur verteilten Paarprogrammierung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Dohrmann09-saros-benutzerfeedback.pdf, 07 2009.

[Glo] Rainer Gloede. XecliP - User Guide. http://xeclip.sourceforge.net/pdf/XecliP-UserGuide.pdf.

[Gus07] Björn Gustavs. Weiterentwicklung eines Eclipse-Plug-Ins zur Verteilten Paarprogrammierung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Gustavs07-saros-DPP-II.pdf, 08 2007.

[iP] The iLBCfreeware.org Project. iLBC. http://www.ilbcfreeware.org/.

[Jac09] Christoph Jacob. Weiterentwicklung eines Werkzeuges zur verteilten, kollaborativen Softwareentwicklung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Jacob09-saros-DPP-IV.pdf, 04 2009.

[Kiw10] Alena Kiwitt. Supporting Various Programming Languages and Plug-Ins Within a Tool For Collaborative Software Development. 2010.

[Krö] Tim Kröner. ITU-T G.711. http://www.voip-information.de/itu-t-g711.html.

[Lau10] Stephan Lau. Verbesserte Präsenz durch Screensharing für ein Werkzeug zur verteilten Paarprogrammierung. 2010.

[LS] Stephan Lukosch and Till Schümmer. XPairtise - A Distributed Pair Programming Plug-in For Eclipse. http://xpairtise.sourceforge.net/.

[Mes] Richard Meston. Sorting Through GSM Codecs: A Tutorial. http://www.commsdesign.com/design_corner/OEG20030711S0010.

[PG] Hugues Pisapia and Marc Gimpel. Java Implementation of Speex (jSpeex). http://sourceforge.net/projects/jspeex/.

[PMS] Leandro Pfleger, Giorgio Merize, and Prof. Frank Siqueira. Pep. http://sourceforge.net/projects/pep-pp/.

[PSA09] Remko Troncon Peter Saint-Andre, Kevin Smith. *XMPP: The Definitive Guide*. O'Reilly, 2009.

[Rie08] Oliver Rieger. Weiterentwicklung einer Eclipse Erweiterung zur Realisierung und Protokollierung verteilten Paarprogrammierung im Hinblick auf Kollaboration und Kommunikation. http://www.inf.fu-berlin.de/inst/ag-se/theses/Rieger08-eclipse-erweiterung-PP.pdf, 07 2008.

[Rin09] Marc Rintsch. Agile Weiterentwicklung eines Software-Werkzeuges zur verteilten Paarprogrammierung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Rintsch09-agile-tool-develop.pdf, 06 2009.

[SA] Peter Saint-Andre. XEP-0045: Multi-User Chat. http://xmpp.org/extensions/xep-0045.html.

[SA04a] Peter Saint-Andre. RFC 3920 Extensible Messaging and Presence Protocol (XMPP): Core. http://www.ietf.org/rfc/rfc3920.txt, 10 2004.

[SA04b] Peter Saint-Andre. RFC 3921 Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. http://www.ietf.org/rfc/rfc3921.txt, 10 2004.

[SALB+] Peter Saint-Andre, Scott Ludwig, Joe Beda, Robert McQueen, Sean Egan, and Joe Hildebrand. XEP-0166: Jingle. http://xmpp.org/extensions/xep-0166.html.

[Sót09] Tas Sóti. Einladungsprozess in Saros. http://www.inf.fu-berlin.de/inst/ag-se/theses/Szuecs10-DPPVI.pdf, 10 2009.

[Sta10] Henning Staib. Verbesserung einer XMPP-Bibliothek für den Einsatz in verteilter Paarprogrammierung. 2010.

[Szü10] Sandor Szücs. Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Szuecs10-DPPVI.pdf, 02 2010.

[Unia] International Telecommunication Union. G.711. http://www.itu.int/rec/T-REC-G.711/e.

[Unib] International Telecommunication Union. G.729. http://www.itu.int/rec/T-REC-G.729/e.

[Val07] Jean-Marc Valin. Speex Manual. http://www.speex.org/docs/manual/speex-manual/manual.html, 05 2007.

[WK02] Laurie Williams and Robert Kessler. *Pair Programming Illuminated*. Addison-Wesley Co., 2002.

[WKCJ] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the Case for Pair-Programming. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.5248&rep=rep1&type=pdf.

[Zil09] Sebastian Ziller. Behandlung von Nebenläufigkeitsaspekten in einem Werkzeug zur Verteilten Paarprogrammierung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Ziller09-saros-nebenlaeufigkeit.pdf, 10 2009.