Freie Universität Berlin
Fachbereich Mathematik und Informatik
Arbeitsgruppe Software Engineering

Analyse und Erweiterung der Voice over IP Funktionalität in Saros

Bachelorarbeit Florian Pütz
Matrikelnummer: 4121788
fpuetz@inf.fu-berlin.de

Betreuerin: Julia Schenk

Eingereicht bei: Prof. Dr. Lutz Prechelt
Prof. Dr. Elfriede Fehr

Berlin, den 30.07.2010

Eidesstattliche Erklärung

Ich versichere, die Bachelorarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Berlin, den 30.07.2010

Florian Pütz

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Weiterentwicklung des Voice over IP Moduls des Plugins Saros für Eclipse, welches in der Bachelorarbeit "Verbesserung der Kommunikationsmöglichkeiten in Saros" implementiert und in das Projekt Saros eingefügt wurde. Es sollen grundlegende Probleme, die den Programmablauf beeinträchtigen, gelöst und Optimierungsschritte zur Verbesserung der Stabilität und der Effizienz evaluiert werden, um das Modul sowohl jetzt als auch in späteren Entwicklungsschritten möglichst einfach erweitern zu können.

Inhaltsverzeichnis

Einleitung

Paarprogrammierung ist eine Programmiertechnik, bei der zwei Entwickler vor einem Rechner sitzen und gemeinsam am Quellcode arbeiten. Den beiden Teilnehmern wird jeweils eine Rolle zugeordnet: die Rolle des Drivers oder des Observers. Der Driver bearbeitet aktiv den Quellcode und hat die Kontrolle über Maus und Tastatur. Der Observer vollzieht die vom Driver durchgeführten Änderungen nach und stellt gegebenenfalls Verständnisfragen. Dadurch entsteht ein Austausch zwischen dem Driver und dem Observer bezüglich der Änderungen und Entscheidungen. Das gemeinsame Zusammenarbeiten wird in diesem Zusammenhang als Sitzung bezeichnet. Die intensive Kommunikation und Reflexion während einer Sitzung über die getroffenen Entscheidungen ist essenzieller Bestandteil der Paarprogrammierung.

Bei der verteilten Paarprogrammierung arbeiten die Teilnehmer zwar ebenfalls am gleichen Quellcode, befinden sich jedoch an unterschiedlichen Standorten. Dies hat zur Folge, dass keine direkte Kommunikation zwischen den Entwicklern möglich ist.
Deshalb ist es für die Kommunikation der Beteiligten notwendig, auf zusätzliche Anwendungen zurückzugreifen. Saros als Plattform zur verteilten Paarprogrammierung hat den Anspruch, die Kommunikation so gut wie möglich nachzuempfinden, um der Kommunikationssituation während einer nichtverteilten Arbeit möglichst nahe zu kommen.

Vor der Einführung einer in Saros integrierten Voice over IP Funktionalität war es bereits möglich in Saros einzelne Benutzter mittels der externen Anwendung Skype anzurufen, und so über Voice over IP zu kommunizieren. Eine vollständig in Saros integrierte Lösung zur Kommunikation existierte nicht.

Im Rahmen der Bachelorarbeit mit dem Thema „Verbesserung der Kommunikationsmöglichkeiten in Saros“ wurde eine integrierte Möglichkeit zum Aufbau einer Sprachverbindung implementiert. Damit war es zwei Teilnehmern möglich miteinander per Sprache zu kommunizieren. Zusätzlich wurde eine auf XMPP basierte Chatfunktion integriert, die die schriftliche Kommunikation unter allen Teilnehmern einer Sitzung erlaubt.

Ziel meiner Arbeit ist es, zuerst grundlegende Probleme in der bestehenden VoIP-Implementierung zu analysieren und zu beheben, um so eine gute Basis für die weitere Entwicklung des VoIP-Moduls zu schaffen. Dazu werde ich im Analyseteil zunächst auf das Hardcoded-Audioformat-Problem eingehen, welches einen Programmabsturz aufgrund von internen Einstellungen, die zu bestimmten Soundkarten inkompatibel sind, bewirkt. Danach schildere ich ein Problem, das einen der Streams beim Beenden einer VoIP-Sitzung daran hindert, ordnungsgemäß geschlossen zu werden, ohne eine Exception auszulösen. Der dritte Aspekt der Analysetätigkeit widmet sich der Latenz, die während der Nutzung der VoIP-Funktionalität zu beobachten war, und im Vergleich zu anderen VoIP-Anwendungen zu hoch ist.

Im zweiten Teil der Arbeit werde ich einen speziellen Erweiterungsvorschlag, die Einführung einer VoIP-Konferenzfunktion, erörtern und einen Implementierungsansatz im Rahmen eines Prototyps evaluieren. Damit soll die Kommunikation mit mehr als zwei Teilnehmern einer verteilten Sitzung ermöglicht werden; die Sprachkommunikation soll auf diese Weise weiter an das Modell von Sitzungen in der Paarprogrammierung angeglichen werden. Abschließend werde ich ein Fazit über die Arbeit am VoIP-Modul ziehen und einen Ausblick für weitere mögliche Verbesserungen geben.

Überblick über das Saros Plugin und die VoIP-Funktionalität

Das Saros Plugin wurde zu dem Zweck geschaffen, dem Benutzer ein Werkzeug zur Verfügung zu stellen, um eine verteilte Sitzung zur Paarprogrammierung zu betreiben. Der Benutzer kann weitere Benutzer zu seiner Liste der dem Benutzer bekannten Kontakte hinzufügen. Wenn der Benutzer das Kontextmenü eines Projektes öffnet, ist es ihm möglich, das betroffene Projekt mittels der Kontextmenüauswahl „Share Project“ an einen Benutzer aus der Kontaktliste zu senden und darauffolgend eine Sitzung mit ihm zu erstellen. Der Nutzer kann außerdem den Status der Sitzung, in der er sich zurzeit befindet, einsehen. Er kann sehen, wer zurzeit an der Sitzung teilnimmt und welcher Teilnehmer zurzeit welche Rolle bekleidet. Bei diesen Rollen handelt es sich um die Driver- und Observerrollen aus der Technik der Paarprogrammierung. An diese Rollen sind Rechte gebunden, die der jeweilige Inhaber dieser Rolle hat: Der Driver ist in der Lage den Quellcode des gemeinsam bearbeiteten Projektes zu bearbeiten; der Observer ist dazu nicht in der Lage. Eine Sitzung kann in Saros mehrere Driver und mehrere Observer haben. Von einem Driver durchgeführte Änderungen werden an alle Teilnehmer gleichzeitig übertragen, sodass Observer oder andere Driver diese Änderungen mitverfolgen können. In der Liste der Sitzungsteilnehmer ist jedem Teilnehmer eine Farbe zugeordnet; durchgeführte Änderungen werden mit der Farbe des Autors hinterlegt, sodass nachvollzogen werden kann, von wem die betroffene Änderung stammt.

Diese Arbeit beschäftigt sich mit der VoIP-Funktionalität, die in Saros integriert wurde, um VoIP-Kommunikation zwischen zwei Teilnehmern einer Saros Sitzung zu ermöglichen. Im Rahmen der technischen Entwicklungsarbeit benenne ich die Implementierung dieser Funktionalität als das VoIP-Modul. Im Folgenden werden die Klassen des Moduls vorgestellt, weil ich mich in den folgenden Kapiteln auf diese Klassen beziehen werde.

Die beiden Klassen `AudioPlayer` und `AudioReceiverRunnable` modellieren ein Wiedergabegerät zur Wiedergabe empfangener Sprachdaten. Die Klassen `AudioRecorder` und `AudioSenderRunnable` bilden ein Aufnahmegerät zur Aufnahme und zum Versenden von Sprache. Diese vier Klassen bilden das Sound-System, ein System aus zwei Komponenten zur Aufnahme und Wiedergabe von Sprache. Die Komponenten können jeweils getrennt gestartet und beendet werden.
Die Klasse `AudioServiceManager` realisiert den Start und das Beenden einer VoIP-Kommunikationssitzung. Die Komponenten des Sound-Systems werden hier gestartet beziehungsweise beendet.

Zur Übertragung der Sprachpakete greift das System auf den in Saros integrierten Streamservice zu. Dieser Service wurde während der Bachelorarbeit mit dem Thema „Verbesserte Präsenz durch Screensharing für ein Werkzeug zur verteilten Paarprogrammierung“ implementiert. Er ermöglicht den auf Streams basierten Datenaustausch zwischen den Sitzungsteilnehmern. Die Klasse `StreamSession` gibt zu diesem Zweck einen `OutputStream` zum Versenden von Sprachdaten an die Aufnahmekomponente aus, damit die aufgezeichnete Sprache an den Empfänger übertragen werden kann. Die Wiedergabekomponente erhält von der `StreamSession`-Klasse dagegen einen `InputStream`, um empfangene Sprachdaten lesen und anschließend wiedergeben zu

können. Weitere Bestandteile des Moduls sind eine Klasse `MixerManager` zur Implementierung eines Eintrags der verfügbaren Mikrophone und Lautsprecher, sowie weitere Klassen und Interfaces, die für die Anbindung an den Streamservice relevant sind. Dabei handelt es sich zunächst um das Interface `IaudioServiceListener` und die abstrakte Klasse `AbstractAudioServiceListener` zur Registrierung einer Sitzung im Streamservice. Die Klasse `AudioServiceListenerDispatch` vermittelt die Ereignisse des Startens und des Beendens einer VoIP-Sitzung an die registrierten Listener. Die Klasse `AudioService` implementiert die für den Streamservice notwendigen Eigenschaften einer VoIP-Sitzung. Diese Eigenschaften umfassen beispielsweise die Größe eines einzelnen Datensatzes, der übertragen werden soll, und den Servicenamen des VoIP-Moduls; dieser Name wird an den Streamservice zur Identifikation übergeben. Außerdem wird in der Klasse `AudioService` die Einladung eines Teilnehmers zu einer Sitzung implementiert.

Ein Codec dient der Kodierung und der Dekodierung von Audiosignalen. Das Kodieren eines analogen Audiosignals in ein digitales Signal ist notwendig, um das Audiosignal in einer Anwendung digital verarbeiten zu können. Das Dekodieren eines digitalen Audiosignals in ein analoges Signal wird durchgeführt, um das digital kodierte Audiosignal wiedergeben zu können. Zur Kodierung und zur Dekodierung der analogen Sprachsignale wird auf zwei Stream-Klassen aus der JSpeex-API zurückgegriffen: Der `Pcm2SpeexAudioInputStream` wird zur Kodierung der aufgezeichneten Sprache eingesetzt, damit sie anschließend mit dem Streamservice übertragen werden kann; der `Speex2PcmAudioInputStream` wird dagegen für die Dekodierung, die zur anschließenden Wiedergabe notwendig ist, verwendet.

Die Java Sound-API wird zur Interaktion mit der Hardware benötigt; die Klasse `Mixer` beschreibt die interne Repräsentation des Audiomixers der Soundkarte. Im Sound-System werden zwei Objekte des Typs `Mixer` für die Ansteuerung des Aufnahme- beziehungsweise Wiedergabegerätes eingesetzt. Der Interfacetyp `DataLine` gibt ein den Streams ähnliches Modell für die Interaktion mit Audiogeräten vor; über eine `DataLine` können Audiodaten von einem Aufnahmegerät gelesen werden, um sie aufzuzeichnen, oder an ein Wiedergabegerät geschrieben werden, um sie wiederzugeben. Für diese beiden Aufgaben werden zwei von `DataLine` abgeleitete Interfaces unterschieden: Zur Aufnahme wird eine `TargetDataLine` und zur Wiedergabe wird eine `SourceDataLine` eingesetzt. Sie unterscheiden sich in den spezifizierten Methoden: `TargetDataLine` spezifiziert eine Methode `read()`, und `SourceDataLine` spezifiziert stattdessen eine Methode `write()`. Alle `DataLines` stammen von einem übergeordneten Interface `Line` ab. Dieses übergeordnete Interface spezifiziert die Methoden `open()` und `close()` zum Öffnen und Schließen der `Line`. Außerdem werden Methoden spezifiziert, um Benachrichtigungen über die Änderung des Zustandes eines `Line`-Objektes zu realisieren. Alle `Lines` haben eine statische Variable vom Typ `Line.Info`. Diese Variable speichert beschreibende Informationen über die betroffene `Line`; beispielsweise werden in der `DataLine.Info`-Variablen Informationen über unterstützte Audioformate gespeichert. Der Begriff Audioformat wird im Kapitel 1.1 im Kontext von Java näher erläutert.

`DataLine`-Objekte werden von `Mixer`-Objekten zur Repräsentation eines Datenstroms zu dem Audiogerät, welches durch das jeweilige `Mixer`-Objekt repräsentiert wird, bereitgestellt.

I Analyse der VoIP-Funktionalität

Die in Saros integrierte VoIP-Lösung wurde auf Basis von JSpeex implementiert.
JSpeex ist eine Javaportierung des Open Source Codecs Speex. Die Implementierung sieht eine einfache Kommunikationsstruktur bestehend aus einem Player und einem Recorder vor, die das Gesprochene nebenläufig sendet und empfängt.
Die VoIP-Funktionalität kam mit dem Saros Release 10.5.28 im Mai 2010 hinzu. Daher gibt es noch nicht viele Erfahrungen bezüglich des Einsatzes der VoIP-Funktionalität.
Im Folgenden möchte ich die wesentlichen Probleme des VoIP-Moduls, die ich während meiner Arbeit entdecken konnte, beschreiben. Die Ursache soll analysiert werden, um anschließend entsprechende Lösungen oder Lösungsansätze zu evaluieren.

1. Das Hardcoded-Audioformat-Problem

Am 23.04.2010 ging eine Email in die Mailing Liste „dpp-devel“, die Mailing Liste für Entwickler im Saros Projekt, ein, in der über eine `IllegalArgumentException` während der Inbetriebnahme der VoIP-Funktionalität berichtet wurde. Darin hieß es, das „hartkodierte Audioformat“ sei nicht kompatibel mit der Soundkarte des Autors [1].

1.1 Problemanalyse

Ein `AudioFormat`-Objekt beinhaltet grundlegende für die Hardware relevante Einstellungen darüber, wie Audiodaten vom Mixer, der mit den Angaben aus diesem Objekt initialisiert wurde, verarbeitet werden. Der Mixer ist der Teil der Soundkarte, der analoge Audiosignale an ein Wiedergabegerät ausgibt oder von einem Aufnahmegerät entgegennimmt, um die Signale zur Auswertung in digitaler Form an ein Programm weiterzuleiten, welches in der Lage ist die aufgenommenen Daten auszuwerten. Bei der Erstellung eines `AudioFormat`-Objektes werden Informationen gespeichert, wie die zu verwendende Kodierungstechnik, die Anzahl verwendeter Kanäle, die Samplerate, die Größe eines Samples, die Anzahl und Größe von Frames, und ob die Daten in einem einzelnen Sample in Big- oder LittleEndian Byte Order gespeichert werden sollen.

Die Einstellungen des `AudioFormates` sind fest im Quellcode niedergeschrieben, das heißt wenn diese Einstellungen auf dem betroffenen Computer zu Inkompatibilitätsproblemen führen, dann gibt es keine Ausweichmöglichkeiten, um den Abbruch der Ausführung des VoIP-Moduls zu verhindern. Ich muss also alternative Einstellungen für den Fall eines Scheiterns aufgrund von inkompatiblen festgelegten Einstellungen bereitstellen.

1.2 Lösung

Das `AudioFormat`-Objekt wird zur Konstruktion einer `DataLine.Info` verwendet. Dieses `DataLine.Info`-Objekt wird danach vom `Mixer`-Objekt zum Aufbau der `DataLine` genutzt. Die Lösung des Problems bestand darin, zuerst zu prüfen, ob die Verwendung des mit dem `AudioFormat`-Objekt aufgebauten `DataLine.Info`-Objektes wirklich zu einer Exception führen wird. Um zu testen, ob die auf dem genannten `AudioFormat`-Objekt basierende `DataLine` zu einem Versagen führen wird, verwende ich eine Methode, die ein `DataLine`-Objekt als Parameter erhält. Das übergebene `DataLine`-Objekt wird im Rahmen eines einfachen Tests mit den Methoden `open()` und `start()` gestartet. Auf diese Weise wird überprüft, ob es bei der Verwendung des `Mixer`-Objekts zusammen mit dem übergebenen `DataLine`-Objekt zu Fehlern kommt. Die Methode gibt einen `boolean` zurück, der anzeigt, ob das `DataLine`-Objekt mit dem festgeschriebenen Audioformat vom Mixer unterstützt wird. Ist dies der Fall, dann verwenden wir die aufgebaute `DataLine` unverändert.
Ist dies nicht der Fall, dann wird als nächstes eine Sammlung weiterer `DataLine.Info`-Objekte als Alternativen mit Einstellungen, die sich von den Einstellungen des problematischen `AudioFormats` unterscheiden, erstellt und in einem Array gesammelt. Ist dieses Array leer, dann muss das Programm mit einer Fehlermeldung beendet werden, weil keine alternativ verfügbare Line existiert. Sonst wird iterativ jedes Element des Feldes überprüft, ob es eine gültige Instanziierung vom Typ `Dataline.Info` ist. Zusätzlich wird mithilfe der Methode `lineWorksOk()`, die das zu untersuchende Info-Objekt und das zugehörige Audioformat erhält, getestet, ob das spätere Verwenden dieser Objekte eine Exception auslösen wird. Sie soll einen Sprachdatenaustausch soweit simulieren, dass es eine `Line playLine` und eine `Line recLine` gibt, die in der Lage sind Sprachdaten zu lesen beziehungsweise zu schreiben.
Auf einen tatsächlichen Datenaustausch kann hierbei verzichtet werden, da das eingangs beschriebene Problem bereits vor dem Aufnahmevorgang auftritt.
Wenn dieser Test erfolgreich war, dann wird das aktuelle Objekt für den Schritt der Konstruktion der `TargetDataLine`, die abschließend zum Aufbau der Sprachverbindung herangezogen wird, weiterverwendet.

1.3 Verbleibende Risiken und weiterführende Tests

Dadurch, dass die Lösung die nächstbeste mögliche Line aufgreift, ist es zum jetzigen Zeitpunkt noch unklar, wie sich das `DataLine`-Objekt, welches mit dem als Alternative herausgesuchten `DataLine.Info`-Objekt konstruiert wurde, zur Laufzeit verhält. Zwar ist die Kompatibilität zur Hardware durch den durchgeführten Test während des Initialisierungsvorgangs sichergestellt, aber die Lösung wurde bisher nicht praktisch angewendet. Deshalb bleibt zu testen, wie sich die alternative `DataLine` auf die Soundqualität auswirkt.
Mir stand während meiner Arbeit kein Rechner zur Verfügung, bei dem das Hardcoded-Audioformat-Problem auftritt. Ein Test zur Prüfung der Soundqualität sollte so aussehen, dass zwei Personen zuerst die so bearbeitete Version von Saros ausführen. Als

nächstes müssen sich beide Teilnehmer in einer Saros Sitzung befinden. Einer der Teilnehmer fordert den anderen zu einer VoIP-Konversation auf. Nachdem die Einladungsmitteilung vom Eingeladenen bestätigt wurde, können beide den Initialisierungsprozess in ihrer Konsole derjenigen Instanz von Eclipse, die das Projekt ausführt, beobachten. Damit man sicher sein kann, dass auch wirklich die Lösung getestet wird, habe ich eine Nachricht an den Logger übergeben, der das Testteam über die Konsole darüber informiert, ob der Programmabschnitt, der die Lösung realisiert, aufgerufen wurde. Dies ist für den Fall notwendig, dass die Hardware der Teilnehmer das standardmäßig festgelegte Audioformat akzeptiert, und die alternative Lösung somit gar nicht benötigt wird.

2. Das Streamservice-Problem

Während einer Paarprogrammiersitzung zur Behebung des Hardcoded-Audioformat-Problems konnte ich durch einen praktischen Test meines Entwicklungsstandes beobachten, dass das Beenden einer VoIP-Sitzung zu einer `IOException` mit der Meldung „Outputstream is closed“ führt. Dies ist darauf zurückzuführen, dass der `OutputStream` in der `AudioSenderRunnable`-Klasse, der von der `StreamSession`-Klasse zur Verfügung gestellt wird, bei Terminierung der VoIP-Sitzung nicht ordnungsgemäß geschlossen wird. Ich habe daraufhin alle Methodenaufrufe in der Methode `readLoop()` mit einzelnen `try-catch`-Blöcken beobachtet, und auf diese Weise konnte ich das Problem auf den `OutputStream` eingrenzen. In der Methode `readLoop()` findet die Aufzeichnung und das Versenden der Sprachpakete statt.

2.1 Problemanalyse

Für die fehlerfreie Terminierung einer VoIP-Sitzung muss sichergestellt werden, dass der Outputstream korrekt geschlossen wird. Das bedeutet, dass er genau dann geschlossen werden muss, wenn die Schleife in der `readLoop()` Methode des `AudiosenderRunnable`-Objektes terminiert, das heißt, wenn die Sitzung beendet wird.
Um herauszufinden, ob das Problem im VoIP-Modul begründet ist, reicht es aus, dass ich das Schließen des `OutputStreams` an genau dieser Stelle erzwinge, und sichergehe, dass keine andere Methode des VoIP-Moduls Einfluss auf den betroffenen Stream hat.
Wenn der Stream trotzdem weiterhin vom VoIP-Modul unabhängig geschlossen wird, dann schließe ich einen Fehler im VoIP-Modul aus. Der nächste Ausgangspunkt zur Lokalisierung des Fehlers wäre die Implementierung des Streamservices, da der betroffene Stream von dort bezogen wurde.

2.2 Lösung

Die Methode `readLoop()` der `AudioSenderRunnable`-Klasse realisiert die Aufnahme während des Betriebs einer VoIP-Sitzung und schließt die für die Aufnahme verwendete Line, den Stream zur Kodierung der Audiodaten und den betroffenen `OutputStream`, wenn die VoIP-Sitzung endet.
Um dafür zu sorgen, dass der `OutputStream` ausschließlich nach Terminierung der Schleife in der Methode `readLoop()` geschlossen wird, habe ich eine Konstruktion aus einem `try-` und einem `finally-`Block vorbereitet. In diesem Block wird das Inputmixerobjekt, welches die interne Repräsentation für das Mikrophon darstellt, der Stream, der für die Kodierung der Sprachdaten verwendet wird, und der betroffene `Outputstream` geschlossen; dieser Schließvorgang soll nach Beenden der Schleife mithilfe des `finally`-Blocks erzwungen werden. Sollte es dabei wiederum zu einer Exception durch den Aufruf der Methode `close()` des Streams kommen, so habe ich nach dem dazugehörigen `catch`-Block einen weiteren `finally`-Block hinzugefügt, in welchem dem `OutputStream`-Objekt `null` zugewiesen wird, um den Stream letztendlich zu zerstören. Dadurch ist sichergestellt, dass der `OutputStream` auch im Fall des Versagens vollständig entfernt wird, damit weitere Exceptions vermieden werden.

2.3 Resultat der Lösung

Der Fehler konnte auf diese Weise nicht behoben werden; der `OutputStream` wird nicht direkt nach der Terminierung der Schleife in der Methode `readLoop()`, sondern noch vorher geschlossen. Dies geschieht ohne einen expliziten Aufruf der Methode `close()`. Deshalb ordne ich den Fehler dem Streamservice zu. Die Auswirkungen des Fehlers sind jedoch nicht kritisch zu betrachten, da weder der Ablauf noch das Beenden einer VoIP-Sitzung gefährdet ist; denn der Fehler tritt erst auf, nachdem ein Teilnehmer den Button zum Beenden der Sitzung angeklickt hat. Ansonsten wird der Vorgang des Beendens korrekt durchgeführt. Weitere Nebeneffekte konnte ich nicht beobachten.

3. Latenzen in der Datenübertragung

Zur Untersuchung des VoIP-Moduls gehörten neben der Durchsicht und der Bearbeitung des Codes auch Tests der Funktionalität. Dazu erstellte ich zunächst eine Sitzung mit einem Projektmitarbeiter. Diese Sitzung verlief fehlerlos und die Übertragung war klar und verständlich. Jedoch fiel uns auf, dass die Übertragung des Signals im Gegensatz zu anderen VoIP-Anwendungen ungefähr zwei Sekunden dauert. Zum Vergleich wurde ein zweites Testszenario aufgebaut, welches aus zwei Programminstanzen auf dem lokalen Computer bestand. Auch in diesem Fall wurde die selbe Verzögerung beobachtet. Im Folgenden werden die kritischen Stellen, die für Verzögerungen anfällig sind, beschrieben und es wird erläutert, inwiefern der beschriebene Programmaspekt die Verzögerung hervorrufen kann.

3.1 Problemanalyse

Zu den Verzögerungen kann es überall dort kommen, wo direkt auf den Sprachdaten gearbeitet wird. In Fall des VoIP-Moduls gibt es dazu drei zentrale Stellen: die Aufnahme, die Übertragung und die Wiedergabe von Sprache. Dabei findet die Aufnahme und die Wiedergabe jeweils im Rahmen der Ausführung des Sound-Systems statt, während die Übertragung der Pakete mithilfe des Streamservices durchgeführt wird.

3.2 Die Implementierung des Sound-Systems

Der Begriff Sound-System bedeutet im Rahmen der Analyse und Erweiterung der VoIP-Funktionalität die Realisierung einer Schnittstelle von der verwendeten Hardware zur Anwendung Saros, die Sprachdaten aufnimmt, überträgt und wiedergibt.
Das Starten und Initialisieren der am Sound-System beteiligten Komponenten wird genau einmal während der Inbetriebnahme der VoIP-Funktionalität ausgeführt. Außerdem wird diese Initialisierung vor einem Datenaustausch durchgeführt. Folglich sind die Latenzen in dem Teilbereich der Anwendung zu suchen, wo die Sprachdaten tatsächlich ausgetauscht werden. Dieser Bereich befindet sich im Falle des Recorders, dem Programmteil, der die Sprachaufnahme ausführt, in der Klasse `AudioSenderRunnable` und im Falle des Players, dem Programmteil, der empfangene Audiodaten abspielt, in der Klasse `AudioPlayer`. Während der Laufzeit der Schleife in der Klasse `AudioSenderRunnable` werden Sprachdaten mit dem `encoderStream`, der zuvor mithilfe der JSpeex-API und der Einstellung des Audioformates vorbereitet wurde, in ein Buffer des Typs Bytearray eingelesen. Anschließend wird der Inhalt des Buffers mittels eines `OutputStreams`, der durch den Streamservice zur Verfügung gestellt wurde, an den empfangenden Teilnehmer gesendet.
Im Rumpf der Schleife in der Klasse `AudioPlayer` dagegen werden zunächst empfangene Audiodaten mit dem `decoderStream`, der mithilfe des Streamservices und der JSpeex-API konstruiert wurde, in ein Bytearray gelesen. Danach wird der Inhalt an den Lautsprecher übergeben und wiedergegeben.
Es wird während der parallelen Bearbeitung der Schleifen also ausschließlich mit den beschriebenen Streams und der Ausgabe der Daten auf den Lautsprecher gearbeitet. Deshalb komme ich zu dem Schluss, dass dort auch das Problem zu suchen ist.

3.3 Die Nutzung des Streamservices

Die `StreamSession`-Klasse stellt Komponenten von Saros, die an das System des Streamservices zur Datenübertragung angebunden wurden, Methoden zur Verfügung, um die benötigten IOStreams auszugeben, und so die Übertragung über das Internet zu realisieren. Die VoIP-Funktionalität greift bereits während des Startvorgangs auf diesen Service zu, um einen `InputStream` für das `AudioReceiverRunnable`- und einen `OutputStream` für das `AudioSenderRunnable`-Objekt zu erhalten, damit die Übertragung der Sprache über das Internet möglich ist. Während des Betriebs einer

VoIP-Sitzung finden auf diesen Streams die nebenläufigen Lese- und Schreibvorgänge unabhängig voneinander statt. Die Klasse `AudioReceiverRunnable` liest empfangene Pakete mit dem zugewiesenen `InputStream` ein, und lässt die gelesenen Audiodaten vom Lautsprecher oder von den Kopfhörern wiedergeben.

Um zu überprüfen, ob die Datenübertragung über das Streamservice-System die Verzögerungen verursacht, kann das Streamservice-System durch ein etabliertes VoIP-Protokoll wie beispielsweise IAX oder SIP ersetzt werden. IAX und SIP werde ich in den Kapiteln 2.6 und 2.7 des zweiten Teils der Arbeit detailliert vorstellen. Dieser Austausch ist möglich, weil der Streamservice eine klare Schnittstelle zum VoIP-Modul hat. So kann das Modul von dieser Schnittstelle gelöst und an das ausgewählte Protokoll angebunden werden. Die übrige Funktionalität des VoIP-Moduls wird beibehalten.
Wenn sich die Latenz nach den durchgeführten Änderungen verbessert, ist davon auszugehen, dass die Verwendung des Streamservice-Systems die Verzögerungen hervorgerufen hat.

3.4 Der Einsatz des Codecs Speex und dessen Javaportierung JSpeex

Im VoIP-Modul wird das Kodieren und Dekodieren von Sprachsignalen mithilfe des Open Source Codec Speex durchgeführt. Für die Nutzung des Codecs steht die API der Javaportierung JSpeex zur Verfügung. Diese API stellt die die Stream-Klassen `Pcm2SpeexAudioInputStream` und `Speex2PcmAudioInputStream` zur Verfügung, die eingelesene Audiodaten kodieren beziehungsweise dekodieren. Während des Betriebs einer VoIP-Sitzung läuft dieser Prozess iterativ ab. Wenn der Vorgang der Kodierung oder der Dekodierung mit JSpeex im Kontext von Saros also Verzögerungen hervorruft, dann wirken sich diese Verzögerungen auf die gesamte VoIP-Kommunikation aus.
Um diesen Fall zu testen, ist es notwendig den Codec auszutauschen. Für Testzwecke kommt beispielsweise der iLBC Codec von ilbcfreeware.org [2] als Ersatz für Speex in Frage. Der iLBC Codec ist unter einer eigenen für nicht-kommerzielle Zwecke freien Lizenz verfügbar. Eine Implementierung in Java ist im SIP-Communicator Projekt (LGPL) vorhanden [3].

II Erweiterung der VoIP-Funktionalität

Die VoIP-Funktionalität unterstützt momentan ausschließlich die Kommunikation zwischen zwei Personen. Sobald sich aber weitere Teilnehmer in einer Sitzung zur verteilten Paarprogrammierung befinden, muss es möglich sein, mit allen Teilnehmern zur selben Zeit sprechen zu können. Um diese Form der VoIP-Kommunikation implementieren und in Saros integrieren zu können, werden im Folgenden Erweiterungsvorschläge vorgestellt und evaluiert, die eine einfache Realisierung der VoIP-Konferenzfunktionalität ermöglichen sollen.

1. Implementierung eigener Strukturen

Zunächst habe ich mich bewusst dazu entschieden, selbst eine Konferenzfunktionalität zu implementieren. Zu Beginn erläutere ich meine Gründe, die mich zu dieser Entscheidung brachten. Danach beschreibe ich meine Entwurfsideen, die ich während der Evaluation dieses Vorgehens erstellt habe. Am Ende erläutere ich, warum ich mich letztendlich gegen dieses Vorgehen entschieden habe, und wie ich weiter vorgegangen bin.

1.1 Gründe für eine eigene Entwicklung

Der erste Grund betrifft das Verfahren der Datenübertragung; das VoIP-Modul verwendet zur Datenübertragung den in Saros integrierten Streamservice. Da dieser ausschließlich für das Saros Projekt erstellt wurde, gehe ich zunächst davon aus, dass keine bestehende Realisierung einer Konferenzschaltung existiert, die auf dem Streamservice-System aufbaut. Folglich wäre es notwendig, das VoIP-Modul vom Streamservice zu lösen und gegen ein verbreitetes Übertragungsprotokoll auszutauschen. Als nächstes ist es nötig, eine darauf basierende Implementierung der Konferenzfunktionalität zu suchen und abschließend in Saros zu integrieren. Dies würde den Rahmen der mir zur Verfügung stehenden Zeit sprengen.
Der zweite Grund für eine eigene Entwicklung ist unabhängig von der Verwendung des Streamservices und betrifft die Implementierung des VoIP-Moduls selbst. Dadurch, dass der ursprüngliche Entwickler die VoIP-Funktionalität vollständig selbst aufbaute, bin ich zunächst davon ausgegangen, dass es keine Lösung für die geforderte Konferenzfunktionalität gibt, die nahtlos in die bereits bestehende Lösung integrierbar ist. Es wäre also gegebenenfalls notwendig, eine bereits erhältliche potentielle Lösung soweit anzupassen, dass sie mit der bestehenden Arbeit hinsichtlich der technischen Implementierung vollständig kompatibel ist.

1.2 Beschreibung der ersten Entwurfsideen

Nach meinem ersten Entwurf, der zunächst bis zu einer Struktur zur Teilnehmerverwaltung bearbeitet wurde, werden alle Teilnehmer einer VoIP-Konferenzsitzung erfasst, und jeder Teilnehmer bekommt wie in einer bisherigen VoIP-Sitzung einen `InputStream` und einen `OutputStream` von der Klasse `StreamSession` zugewiesen. Einer der Teilnehmer stellt den Host dar. Im ersten Entwurf ist der Host der Saros Sitzung auch Host der VoIP-Konferenz; später soll sich die Wahl des Hosts danach richten, wer das bessere Verhältnis aus Latenzzeit und Bandbreite hat. Die Latenzzeit muss dabei möglichst niedrig und die Bandbreite möglichst hoch sein. Eine genaue Gewichtung habe ich in diesem Entwicklungsstand nicht ausgearbeitet.
Die Kommunikation verläuft folgendermaßen: Die Sprachdaten aller Teilnehmer werden mit einer Kennung des Absenders versehen und an den Host übertragen. Dort werden die Sprachdaten gesammelt und als nächstes an alle Teilnehmer gesendet. Dabei wird mithilfe der Kennung des ursprünglichen Absenders dafür gesorgt, dass sich ein Teilnehmer nicht selbst hört.

Kernelement meines ersten Entwurfes ist eine Datenstruktur mit dem Namen `AudioMultikomManager`, die auf einer `HashMap` basiert. Dies hat den Vorteil, dass die Suche nach einem Objekt in der `HashMap` auf einer einfachen Zahl als Schlüssel anstatt auf einem komplexen Datentyp durchgeführt werden kann. In dieser Datenstruktur wird für jeden Teilnehmer in einer VoIP-Sitzung ein Objekt des Typs `VoIPParticipant` und eine zugehörige Identifikationsnummer gespeichert. Die Klasse `VoIPParticipant` repräsentiert einen Sitzungsteilnehmer in einer VoIP-Sitzung und enthält neben dem `User`-Objekt, das von Saros für diesen Teilnehmer angelegt wurde, zusätzliche Informationen, die für die VoIP-Sitzung relevant sind; z.B. wird in dieser Klasse gespeichert, ob derjenige der Host der VoIP-Sitzung ist. Außerdem können hier Informationen über Latenzen und Bandbreiten festgehalten werden. Das `User`-Objekt, das von Saros bezogen wurde, soll weitere für die VoIP-Konferenz relevante Informationen zum Abruf verfügbar machen, falls diese Informationen in späteren Entwicklungsschritten von Interesse sind. Die Identifikationsnummer ist nötig, um die Operationen, die in der `AudioMultikomManager`-Klasse definiert wurden, ausführen zu können. Das sind zunächst `add()`, `remove()` und `clear()`. Damit können Nutzer von Saros zur VoIP-Konferenzsitzung hinzugefügt oder entfernt werden. Außerdem kann die VoIP-Sitzung vollständig beendet werden, indem mit dem Aufruf von `clear()` alle Teilnehmer entfernt werden.

Die Kontrolle über den `AudioMultikomManager` sollte der `AudioServiceManager` erhalten, damit dieser die benötigten Streams vom Streamservice anfordern und den Teilnehmern zuweisen kann. Die Sprachdaten der Teilnehmer sollten dann an alle weiteren Teilnehmer, die in der Struktur gespeichert sind, gesendet werden. Ein einzelner Empfänger sollte die Daten aller Teilnehmer gleichzeitig gebündelt erhalten.
Weiter wurde der Entwurf nicht durchgeführt; die Gründe für das Verwerfen des Entwurfs werden im nächsten Abschnitt erläutert.

1.3 Gründe für den Abbruch des Eigenentwurfs und weiteres Vorgehen

Das VoIP-Modul hat in der aktuellen Fassung ein schwerwiegendes Latenzproblem.
Dieses Problem soll dadurch gelöst werden, dass das VoIP-Modul gegen eine etablierte VoIP-Lösung ausgetauscht wird, die dieses Problem nicht hat. Zusätzlich kann der `OutputStream`, der Gegenstand des Streamservice-Problems ist, obsolet werden; dies ist abhängig davon, für welche alternative VoIP-Lösung ich mich entscheide.
Der zweite Grund ist, dass die Nutzung einer bestehenden VoIP-Lösung eine Konferenzfunktionalität bereits enthält, sodass die Konferenzfunktionalität zusätzlich mitgeliefert wird. Im Verlauf der Arbeit werden diese VoIP-Lösungen vorgestellt.

Aus den genannten Gründen schließe ich, dass das Modul durch eine vorhandene Implementierung ganz oder zumindest teilweise ersetzt werden muss. Ein weiterer Vorteil ist, dass die Implementierungen dieser Standards in einigen Fällen staatlich [4] oder durch eine aktive Open Source Entwicklergemeinschaft gefördert und weiterentwickelt werden.

2. Evaluation etablierter VoIP-Standards

In diesem Abschnitt des Erweiterungsteils habe ich fünf populäre VoIP-Standards untersucht. Zuerst habe ich eine Lösung untersucht, die auf dem Eclipse Communication Framework basiert und Jingle als Übertragungsprotokoll verwendet.
Während eines Exkurses habe ich Skype4Java, eine Open Source Java-API für die externe Anwendung Skype, untersucht.
Als nächstes folgt die Spezifikation H.323 der International Telecommunication Union (ITU). Diese Spezifikation beschreibt ein paketorientiertes Protokoll, welches sich für die Übertragung von Daten verschiedener Medientypen, beispielsweise Audio- und Videodaten, eignet. SIP wurde fast zeitgleich mit H.323 herausausgegeben [5]. SIP beschreibt als Spezifikation ein Protokoll, welches speziell auf den Aufbau und den Betrieb von Kommunikationssitzungen ausgerichtet ist. Dabei kann es sich zum Beispiel um VoIP-Konferenzen, Videokonferenzen oder Instant Messaging handeln [6].
Während der Analyse von SIP wird die Spezifikation und dessen Referenzimplementierung JAIN-SIP ausgewertet, und im Rahmen dessen wird das SIP-Communicator Projekt vorgestellt.
Zum Schluss stelle ich das Protokoll IAX vor, welches eigens für den Betrieb einer *Asterisk* Serveranwendung eingeführt wurde. Im Rahmen der Evaluation von IAX stelle ich außerdem den Zweck und die Funktionsweise der Anwendung *Asterisk* vor und werte einen IAX-basierten Prototyp aus.

2.1 Auswertungskriterien der folgenden VoIP-Technologien

Das wichtigste Kriterium für die Auswahl einer VoIP-Technologie ist, ob es zur betrachteten Technologie eine Implementierung in Java gibt, um eine einfache Integration in Saros zu ermöglichen, ohne eine Portierung für Java implementieren zu müssen.
Das nächste wichtige Kriterium ist die Klärung der Lizenzlage zur rechtlichen Absicherung. Saros wurde unter den Bedingungen der Open Source Softwarelizenz General Public License (GPL) in der Version 2.0 herausgegeben. Um potentielle VoIP-Lösungen oder zusätzliche Bibliotheken in das Projekt zur Nutzung einzubinden, muss ich darauf achten, dass die Bedingungen der Lizenz der betroffenen Lösung oder Bibliothek mit den Bedingungen der GPL 2.0 vereinbar sind. Um die Lizenzierung auszuwerten, richte ich mich in den folgenden Evaluationen nach den Richtlinien in [7] und nach der Tabelle in [8].
Weiterhin muss ich klären, ob die auszuwertende Lösung die Konferenzfunktionalität implementiert, denn diese Funktionalität möglichst leicht zu realisieren, ist eine der grundlegenden Zielsetzungen der Arbeit. Das letzte Kriterium besteht darin, zu überprüfen, ob die vorgestellte Lösung das NAT-Problem löst. Das NAT-Problem besteht darin, dass ein NAT-System den Verbindungsaufbau zu einem Rechner, der sich in einem NAT-betriebenen Netzwerk befindet, verhindert. Die Begriffe NAT, NAT-System und das NAT-Problem werden im nächsten Kapitel genauer erklärt. Anders als die anderen Kritierien ist das NAT-Problem aber kein Ausschlusskriterium, weil dieses Problem nicht auftritt, wenn keiner der betroffenen Rechner sich in einem Netzwerk befindet, welches ein NAT-System einsetzt.

Die Auswertungsstruktur sieht vor, dass die zu evaluierende VoIP-Lösung zuerst grundlegend vorgestellt wird, um einen Überblick über den Hintergrund der VoIP-Lösung zu schaffen. Danach werden Vorteile und Nachteile genannt und erläutert. Zum Schluss folgen gegebenenfalls die Gründe, die mich zu einer Entscheidung für oder gegen die vorgestellte VoIP-Lösung geführt haben.

2.2 Das NAT-Problem

Network Address Translation (NAT) ist ein Verfahren, welches die Abbildung einer öffentlichen IP-Adresse auf mehrere Rechner und Geräte innerhalb eines lokalen Netzwerkes ermöglicht. Den Geräten werden dabei private IP-Adressen zugeteilt, die nur den Komponenten innerhalb dieses Netzwerkes kenntlich gemacht sind. Das Verfahren kann auf Hardwareebene, beispielsweise mit einem NAT-Router, oder auf Softwareebene, beispielsweise mit einer NAT-Firewall, in das Netzwerk eingebunden werden.
Falls eine Kommunikation von einem Gerät außerhalb dieses privaten Netzwerkes mit einem Computer innerhalb dieses Netzes aufgebaut werden soll, muss die Anfrage mithilfe der öffentlichen Adresse durch das eingesetzte NAT-System übersetzt und anhand dieser Übersetzung zum Ziel der Anfrage weitergeleitet werden. Ein Datenaustausch basierend auf diesen Vorgang wird NAT-Traversal genannt [9].
Wenn NAT-Traversal nicht möglich ist, dann wird das Signal von der Anwendung, die den Verbindungsaufbau über das NAT-System versucht, zunächst an das NAT-System

geschickt; aber danach ist es unklar, wohin das NAT-System das Signal weiterleiten muss. Folglich wird das Signal vom NAT-System verworfen, sodass keine Kommunikation zustande kommt.
In den folgenden Evaluationsschritten muss deshalb geklärt werden, inwiefern die betrachteten Lösungen NAT-Traversal spezifizieren und gegebenenfalls implementieren.

2.3 VoIP mit Jingle und dem Eclipse Communication Framework

Die im Folgenden vorgestellte Lösung wurde für die Veranstaltung "Google Summer of Code 2007" entwickelt, und ist eine Instant Messaging Anwendung, die Kommunikation mittels VoIP unterstützt und vollständig in Eclipse integriert ist.

2.3.1 Zweck des Eclipse Communication Frameworks

Das Eclipse Communication Framework (ECF) wurde zu dem Zweck geschaffen, eine zentrale Schnittstelle für den Aufbau von Kommunikationsstrukturen für Anwendungen, die in Eclipse integriert wurden oder auf Eclipse basieren, zu bieten. Dadurch wird eine Umgebung für die Implementierung verschiedener Formen von Kommunikation, beispielsweise Point-to-Point- und Publish-and-Subscribe-Kommunikation, bereitgestellt [10].
Um diesen Zweck zu erfüllen, stellt das ECF eine Bibliothek zur Verfügung, die Spezifikationen und Implementierungen zur Realisierung dieser Kommunikationsstrukturen bietet. In der Implementierung der ECF-Lösung wird auf die Call-API des ECF zurückgegriffen. Diese API beinhaltet Interfaces, die spezifizieren, wie eine konkrete Sitzung zur Telefonie aufgebaut sein muss [11]. Zusätzlich bietet die Call-API Klassen für die Behandlung von Fehlern, die während einer Sitzung auftreten.

2.3.2 Beschreibung und Evaluation der mit ECF realisierten Lösung

Die ECF-basierte VoIP-Lösung nutzt Jinge als Übertragungsprotokoll für die Sprachverbindung. Jingle ist eine Erweiterung des XMPP-Protokolls und spezifizert ein Protokoll für Audio- und Videokommunikationssitzungen. Als Implementierung der Spezifikation Jingle wird in der ECF-VoIP-Lösung die Smack-API verwendet, die eine Implementierung für XMPP und der Erweiterung Jingle zur Verfügung stellt.
Die Instanziierung des Audiosystems wird über die Smack-API an das Java Media Framework delegiert. Das Java Media Framework (JMF) ist eine Java-API, die Funktionalität zur Aufnahme und Wiedergabe von Audio- und Videodaten für Anwendungen in Java bereitstellt [12].
Insgesamt verfügt die ECF-VoIP-Lösung also über ein Sound-System, bereitgestellt durch das JMF, und über ein Übertragungsprotokoll, bereitgestellt durch die Implementierung von Jingle in der Smack-API. Auf Basis dieser Komponenten und der Klassen in der Call-API wurden die Interfaces der Call-API implementiert, sodass eine funktionsfähige VoIP-Lösung entstand.

Ein Nachteil ist aber, dass der Nutzer in Zukunft zusätzlich das Eclipse Communication Framework installieren muss, wenn er auf die VoIP-Funktionalität zurückgreift. Zwar wäre die ECF-VoIP-Lösung selbst in Saros integriert, aber der Nutzer ist trotzdem auf zusätzliche Software angewiesen.
Das Ausschlusskriterium ist, dass Jingle derzeit keine Konferenzschaltung ermöglicht [13].
Das zweite Ausschlusskriterium, welches mich dazu veranlasste, mich gegen die Integration dieser Lösung in Saros zu entscheiden, ist die Lizenzierung; aufgrund des schwach ausgeprägten Copyleftes der Eclipse Public License, die in der vorgestellten Lösung verwendet wird, ist diese nich mit der General Public License, die im Saros Projekt verwendet wird, kompatibel [7].

2.4 Skype4Java

Saros verfügt bereits über eine Möglichkeit zur Kontaktaufnahme per Skype mit einem Kontakt aus der Kontaktliste. Dennoch brachte mich das Gespräch mit einem ehemaligen Mitarbeiter der AG Software Engineering dazu, mich für zwei Arbeitstage mit der Skype-API und deren Implementierung in Java, Skype4Java, auseinanderzusetzen, um einen Einblick in die Möglichkeiten der Skype-API zu erhalten.
Skype4Java ist eine Bibliothek, die zu dem Zweck entwickelt wurde, den Befehlssatz der Skype-API mithilfe von Objekten, Attributen und Methoden in Java zu abstrahieren. Die Skype-API selbst besteht aus einzelnen Befehlen, die an die externe Anwendung Skype gesendet und von dieser ausgeführt werden. Außerdem beinhaltet die Skype-API Antwort- und Fehlernachrichten, die wiederum in der Anwendung, die die API mit Skype4Java aufgerufen hat, verarbeitet werden können. Dies kann zum Beispiel die Bestätigung zu einem angenommenen Telefonat oder eine Nachricht sein, die anzeigt, dass ein Teilnehmer zurzeit nicht verfügbar ist.

Ich entschied mich für einen kurzen Testlauf. Dazu installierte ich Skype, erstellte ein Benutzerkonto für Skype und bereitete ein Testprojekt in Eclipse vor. Als nächstes wurde die Bibliothek in das Projekt integriert. Innerhalb einer Testanwendung konstruierte ich einen einzelnen CALL-Befehl und führte ihn aus, um einen Mitarbeiter des Projekts Saros über die Skype-Anwendung anrufen zu lassen. Mit diesem Test hatte ich Erfolg.

Sofern die vorgestellte Skype-basierte Lösung nicht bereits dadurch obsolet wäre, dass Saros bereits über eine äquivalente Funktionalität verfügt, wäre das Ausschlusskriterium die Lizenz der Bibliothek; die Bibliothek wurde unter der Lizenz Apache License 2.0 veröffentlicht, die mit der in Saros verwendeten General Public License Version 2.0 nicht kompatibel ist [14].

2.5 H.323

H.323 wurde von der International Telecommunication Union (ITU) herausgegeben und beschreibt ein Übertragungsprotokoll zur Realisierung paketorientierter Audio- und Videokommunikation. Diese Spezifikation ist mit weiteren Standards der ITU kompatibel, wodurch eine hohe Erweiterbarkeit einer H.323-basierten Lösung gegeben ist.
Ein Nachteil ist aber, dass NAT-Traversal für Videokommunikation durch die Spezifikation H.460 zwar möglich ist, aber für VoIP-Kommunikation gibt es dazu keine Spezifikation. Außerdem gibt es bislang keine Open Source API in Java. Zwar wurde im Jahr 2000 eine Anfrage zur Spezifikation von H.323 im Rahmen von JAIN eingereicht, diese wurde jedoch im Jahr 2002 zurückgezogen [15]. Die Initiative "Java APIs for Integrated Networks" (JAIN) wird im Kapitel 2.6.1 näher erläutert. IBM bietet eine Java-Umgebung unter dem Namen J323 an. Diese steht aber unter einer proprietären Lizenz.

2.6 SIP

Das Session Initiation Protocol wurde im Jahr 2002 durch die Organisation Internet Engineering Task Force (IETF) spezifiziert [16] und hat den Zweck den Aufbau und den Ablauf von Kommunikationssitzungen zu spezifizieren. Das Protokoll basiert ähnlich wie das Hyptertext Transfer Prototocol auf der Konstruktion und dem Versenden von Klartextnachrichten. Eine Nachricht besteht aus zwei Komponenten: dem Header und dem Body. Im Header werden grundlegende Informationen zur Verbindung selbst gespeichert. Die wichtigsten Informationen sind der Name des auszuführenden Befehls, beispielsweise INVITE, die Adresse des Absenders und des Empfängers in Form einer SIP-URI, die Angabe einer IP-Adresse, einer Portnummer und des zu verwendenden Protokolls für die Antwort auf diese Nachricht, einer Identifikationsnummer für die aufzubauende Verbindung, sowie Format und Länge der Daten im Body der Nachricht [17].
Im Body können mithilfe des Session Description Protocols, ebenfalls eine Spezifikation des IETF [18], weitere Informationen über Medien, die während der beschriebenen Sitzung bei den Teilnehmern zum Einsatz kommen, Verschlüsselungen, eine Benennung der Sitzung und Weiteres hinterlegt werden.
Zur Datenübertragung wird das Real-Time Transport Protocol (RTP) eingesetzt. RTP ist ein Protokoll, welches zum Zweck der Übertragung von medienbasierten Streams,
wie beispielsweise Audio- und Videostreams, geschaffen wurde [19]. RTP wurde ebenfalls von der IETF herausgegeben.

2.6.1 Die Referenzimplementierung JAIN-SIP

Die Initiative "Java APIs for Integrated Networks" (JAIN) ist mit dem Zweck gegründet worden, eine Zusammenfassung von APIs zur Realisierung von Netzwerk- und Telekommunikationsanwendungen in Java zu bieten. Heute beteiligen sich mehr als 80 Firmen aktiv an der Entwicklung der APIs [20].

Das Projekt JAIN-SIP wurde vom National Institute of Standards and Technologies, einer staatlichen Dienststelle der USA für Entwicklungen von Technologien im Bereich der Naturwissenschaften, publiziert und aktiv weitergeführt. Zusätzlich werden neben dem Standard für SIP auch der Standard Session Description Protocoll (SDP) und weitere Spezifikationen des IETF, die mit SIP in Relation stehen, implementiert, um den Funktionsumfang der Referenzimplementierung zu erweitern [21]. Die API steht unter Public Domain, sodass Entwickler diese Implementierung frei nutzen und verändern dürfen. Trotzdem ist die aktive Teilnahme freiwilliger Entwickler am Projekt wie in anderen Open Source Projekten ausdrücklich erwünscht [4].

2.6.2 Das SIP-Communicator Projekt

Das SIP-Communicator Projekt realisiert einen Instant Messaging Client und VoIP-Client in Java. Eine Instant Messaging Anwendung implementiert die Kommunikation über Sofortnachrichten, einer Ausprägung der synchronen Kommunikation. Dabei sind die Teilnehmer bei einem Instant Messaging Dienst angemeldet; über diesen Service können jetzt Nachrichten versendet werden, die der Empfänger annähernd in Echtzeit empfängt, und die bei ihm angezeigt werden. Die Anwendung SIP-Communicator unterstützt sowohl mehrere Instant Messaging Services, als auch VoIP-Kommunikation mittels SIP. Als VoIP-Provider wird iptel.org verwendet, sodass keine direkte Kommunikation zum Kommunikationspartner, sondern eine über einen externen Service vermittelte Kommunikation aufgebaut wird. Insbesondere wurde die Implementierung von NAT-Traversal an diesen Service delegiert, sodass eine Lösung für das NAT-Problem im Rahmen der Entwicklungsarbeiten nicht nötig ist. Für die Implementierung des SIP-Stacks wird auf JAIN-SIP zurückgegriffen. Ein SIP-Stack ist die Schnittstelle einer Anwendung zu einem Kommunikationsnetzwerk, das SIP als Protokoll verwendet.
Der SIP-Communicator ist unter der Lesser General Public License erhältlich [22].

2.6.3 Vor- und Nachteile von SIP

Ein wesentlicher Vorteil einer Lösung, die auf SIP basiert, sind die zusätzlichen Spezifikationen, die von der IETF herausgegeben und in Verbindung mit einer auf SIP basierenden Anwendung implementiert und genutzt werden können. Es existieren im Rahmen von der Bibliothek JAIN-SIP bereits Implementierungen zusätzlicher Standards. Die JAIN-SIP Bibliothek wird von staatlicher Seite aus aktiv weiterentwickelt und gefördert, sodass eine kontinuierliche Weiterentwicklung gesichert ist. Mit dem SIP-Communicator gibt es eine realisierte VoIP-Lösung, die durch die Nutzung der LGPL legal in Saros integriert werden kann.
Der für mich entscheidende Nachteil, der mich dazu veranlasste, IAX und nicht SIP für meinen Prototyp zu verwenden, ist das Fehlen von einer Spezifikation bezüglich einer Realisierung von NAT-Traversal, obwohl durch den Provider iptel.org auch dies gegeben ist. Eine integrierte Lösung des NAT-Problems existiert aber auch auf diese Weise trotzdem nicht.

Insgesamt halte ich SIP auf Basis von JAIN-SIP als Bibliothek und auf Basis des SIP-Communicators als VoIP-Lösung aus den genannten Gründen für eine gute Lösung für die Verwendung in Saros.

2.7 IAX

IAX wird als Übertragungsprotokoll für die Open Source Anwendung *Asterisk* eingesetzt, die die Funktionen einer Telefonanlage implementiert. Eine Telefonanlage vermittelt Anrufe an den jeweiligen Empfänger und stellt die Verbindung zwischen zwei Kommunikationsteilnehmern her. Als virtuelle Telefonanlage wird *Asterisk* ähnlich eines Servers auf einem externen Rechner ausgeführt und nimmt VoIP-Anrufe zur Vermittlung an den gewünschten Gesprächspartner entgegen. Konfiguriert wird die Anwendung über Konfigurationsdateien. Die wichtigsten Konfigurationsdateien für den Betrieb einer *Asterisk* Telefonanlage sind `iax.conf`, die die relevanten Einstellungen des Übertragungsprotokolls IAX verwaltet, und `extensions.conf`, in der der Dial Plan niedergeschrieben ist. Der Dial Plan legt fest, wie ein eingehender oder ein ausgehender Anruf von der Telefonanlage bearbeitet und weitergeleitet wird [32].
Mit IAX ist es möglich, eine Verbindung zwischen mehreren *Asterisk* Telefonanlagen oder zwischen einer *Asterisk* Telefonanlage und einem Endgerät herzustellen. Bei einem Endgerät kann es sich um ein VoIP-Telefon oder eine Softphone-Anwendung handeln. Eine Softphone-Anwendung ist ein Programm, welches die Funktionalitäten eines Telefons implementiert.

2.7.1 Gründe für die Verwendung von IAX in Saros

Mit NJIAX existiert eine Bibliothek, die die Funktionen des Protokolls IAX implementiert. Sie wurde ursprünglich von Nomasystems bereitgestellt und steht unter der Lesser General Public License in der Version 2.1 zur Verfügung. Diese Lizenz ist kompatibel zu der in Saros verwendeten Lizenz GPL in der Version 2. Eine Dokumentation ist im Quellcode erhältlich [23]. Ein spezielles Ziel der Entwicklung von IAX war, dass eine integrierte Unterstützung für NAT-basierte Netzwerke geboten werden soll. Zu diesem Zweck verwendet eine IAX-basierte Lösung ausschließlich den fest spezifizierten Port 4569. Zusätzlich ist spezifiziert, dass Signale zur Behandlung der Verbindung und Sprachdaten im Gegensatz zu SIP gemeinsam mittels IAX übertragen werden [24]. Unter dem Begriff Signale versteht man spezielle Befehle, die eine VoIP-Sitzung initiieren, steuern oder beenden können. Beispiele dafür sind `NewCall` unter IAX oder `INVITE` unter SIP; diese Befehle initiieren ein neues VoIP-Telefonat. Mit Daten sind die tatsächlichen Sprachdaten gemeint. Die Datenpakete können mittels Port Forwarding des explizit festgelegten Ports an den Zielrechner weitergeleitet werden. Port Forwarding ist ein Verfahren zur Weiterleitung von Datenpaketen an den Zielrechner, die an dem Port des Rechners eingehen, der das Port Forwarding durchführt. Falls ein Router, der NAT verwendet, auch das Port Forwarding Verfahren unterstützt, kann auf diese Weise das NAT-Problem in diesem betroffenen Netzwerk umgangen werden.

Asterisk unterstützt die Funktionalität der VoIP-Konferenzschaltung [25]. Deshalb ist mit einer IAX-basierten Lösung in Saros und der Anwendung *Asterisk* als externe Serveranwendung eine Konferenzschaltung möglich.
Der einzige Nachteil ist, dass der externe Betrieb der Serveranwendung *Asterisk* für die Realisierung einer zur Konferenzschaltung fähigen Lösung notwendig ist. Ich habe mich trotzdem für die Evaluation eines IAX-basierten Prototyps entschieden, weil IAX einen integrierten Lösungsansatz für das NAT-Problem bietet, der durch die Zielsetzung der Entwickler kontinuierlich weiterentwickelt wird.

2.7.2 Implementierung und Auswertung eines Prototyps

Für die Implementierung meines Prototyps habe ich zuerst die notwendigen Voraussetzungen geschaffen, indem ich den Quellcode der Bibliothek NJIAX kompiliert und den resultierenden Bytecode in einem Jar Archiv zusammengefasst habe.
Das nächste Ziel war die Implementierung eines Anrufsignals; es soll möglich sein, mit dem Prototyp einen Anruf durchzuführen. Dazu musste zuerst das Sound-System bereitgestellt werden. NJIAX bietet dafür die zwei abstrakten Klassen `Player` und `Recorder`. Für diese Klassen können je nach verwendetem Audiocodec unterschiedliche Subklassen mit unterschiedlichen Implementierungen erzeugt werden. Für die Implementierung des Sound-Systems in meinem Prototyp habe ich zwei Klassen erstellt, die jeweils von den abstrakten Klassen `Player` beziehungsweise `Recorder` abgeleitet sind. Anschließend habe ich zur Implementierung dieser abgeleiteten Klassen die Funktionalität aus dem VoIP-Modul von Saros übernommen und die übernommene Funktionalität an die Signaturen der Methoden, die in den abstrakten Elternklassen vorgegeben sind, angepasst. Insbesondere habe ich dazu die Funktionalitäten der Klassen `AudioRecorder` und `AudioSenderRunnable` sowie die Funktionalitäten der Klassen `AudioPlayer` und `AudioReceiverRunnable` in der abgeleiteten Player- beziehungsweise Recorderklasse vereint.
Der nächste Schritt ist der Aufbau einer Klasse `Client`, in welcher die Funktionalität, ein Signal zur Initiierung eines Gesprächs an die *Asterisk* Telefonanlage zu senden, realisiert wird; auf diese Weise soll in der Klasse `Client` ein einfaches Anrufsignal implementiert werden. In der Klasse `Client` soll ein Objekt vom Typ `Peer` gespeichert werden. Der Typ `Peer` modelliert einen einzelnen Teilnehmer in einer VoIP-Sitzung und hat einen Benutzernamen, ein Passwort und die IP-Adresse des Host-Computers, auf dem die *Asterisk* Anwendung betrieben wird. Zusätzlich wird ein `PeerListener` übergeben, der Änderungen über den Status des `Peers` verarbeitet. `PeerListener` ist ein Interface; der Listener muss selbst implementiert werden. Im Prototyp ist der Listener so implementiert, dass Änderungen des Status eines Teilnehmers mit einer Konsolenausgabe angezeigt werden. Nach der Konstruktion des `Peers` wird die Verbindung zum Host-Rechner hergestellt.
Damit wurde der `Client` aufgebaut. Der letzte Schritt ist die Implementierung eines Anrufs. Zu diesem Zweck wurde eine Methode `doCall()` definiert, die eine Rufnummer als String kodiert erwartet. In der Methode wird zunächst ein Benutzerkommando `NewCall` konstruiert. Benutzerkommandos sind Signale, die von einem Teilnehmer ge-

neriert und von der *Asterisk* Telefonanlage ausgeführt werden. Dem Kommando `NewCall` werden die anzuwählende Telefonnummer und das `Peer`-Objekt des Teilnehmers, der den Anruf initiiert, als Parameter übergeben. Dann wird ein Objekt vom Typ `Call` vorbereitet. In diesem Objekt wird das `Peer` Objekt und die Rufnummer vom Teilnehmer gespeichert, der das `NewCall` Kommando geschickt hat. Am Ende werden die zu Beginn definierten Audiokomponenten instantiiert und gestartet.

Während der Implementierung des Prototyps hatte ich Probleme, die zum einen die Implementierung des Sound-Systems und zum anderen die Konfiguration der *Asterisk* Telefonanlage betreffen. Es ist noch unklar, wie das Sound-System von der Bibliothek verwendet wird. In der Klasse Call gibt es zwar eine Methode `startRecorder()`, und nach der Dokumentation werden Player und Recorder während des Aufrufs des Konstruktors initialisiert, aber es bleibt unklar, welcher Player und welcher Recorder gemeint ist; in der Bibliothek sind zum einen die abstrakten Komponenten und zum anderen die Komponenten `GSMPlayer` und `GSMRecorder` eingetragen, aber die GSM-Komponenten implementieren keine Funktionalität.
Das zweite Problem ist, dass der *Asterisk*-Host momentan noch Verbindungsversuche zurückweist. Der Grund dafür ist ein Fehler im Dial Plan, der bewirkt, dass eingehende Anrufe von der Telefonanlage zurückgewiesen werden.

Nach Abschluss der Implementierungsarbeit kann der Prototyp ein einfaches Anrufsignal generieren und an die *Asterisk* Telefonanlage senden. Während des nächsten Entwicklungsschrittes bleiben drei Dinge zu tun. Der Dial Plan in der Konfigurationsdatei `extensions.conf` der Telefonanlage muss so eingestellt werden, dass es klar ist, wie ein durch den Prototyp generiertes Anrufsignal durchgestellt werden muss. Nachfolgend muss für den Prototyp eine Routine für das Bearbeiten und Akzeptieren von durchgestellten Anrufsignalen implementiert werden. Drittens muss die Funktionalität implementiert werden, über den Prototyp mit Sprache kommunizieren zu können.
Für das Projekt Saros und für mich habe ich die Erkenntnis gewonnen, dass eine IAX-basierte VoIP-Lösung erst dann gut realisierbar ist, wenn die Telefonanlage korrekt konfiguriert ist; das zu erreichen ist das größte Problem bei der Einführung einer VoIP-Lösung auf Basis von IAX. Für die Realisierung der darauffolgenden Schritte bietet die Bibliothek NJIAX Funktionalität an.

Erfahrungen während der Arbeit im Saros Projektteam

Das Saros Projektteam ist in drei Gruppen aufgeteilt. Die Projektmitarbeiter sind studentische Entwickler, die meist im Rahmen einer schriftlichen Arbeit aktiv an der Entwicklung von Saros beteiligt sind. Daneben gibt es Betreuer, die Projektmitarbeiter, die an einer schriftlichen Ausarbeitung arbeiten, begleitend unterstützen. Das Entwicklerteam wird von einem Projektleiter angeführt, der das wöchentliche Meeting leitet und die Aufgaben der Mitarbeiter für die Releases festlegt. Im wöchentlichen Meeting diskutieren der Projektleiter, die Projektmitarbeiter und die Betreuer gemeinsam projektrelevante Themen, wie beispielsweise Änderungen am Entwicklungsprozess im Saros Projekt.
Im Saros Projekt wird nach agilen Ansätzen gearbeitet. Diese agilen Ansätze zeichnen sich durch kurze Planungs- und Entwicklungsphasen aus, die sich wiederholen [36]. Beispielsweise findet einmal im Monat ein Release statt. Im darauffolgenden Meeting werden die Aufgaben für das nächste Release festgelegt, ohne formale und langfristige Pläne auszuarbeiten.
Eine der agilen Praktiken, die im Saros Projekt eingesetzt werden, ist das Daily Standup Meeting. Diese Praktik stammt aus Scrum und sieht vor, dass das Entwicklerteam einmal täglich zusammenkommt. Jeder Mitarbeiter erläutert dann kurz, welche Aufgaben er am Vortag bearbeitete, welche Probleme er dabei hatte, und welche Aufgaben er am Tag des Daily Standup Meetings bearbeiten wird. Der Zweck dieser Meetings ist, dass die Teammitglieder einen besseren Einblick in die Arbeit ihrer Kollegen bekommen. Nach meiner Erfahrung wurde dieser Zweck auch erfüllt; es war interessant zu erfahren, welches Aufgabenfeld vom jeweiligen Entwickler bearbeitet wurde. Leider erhalten Mitarbeiter bezüglich ihrer angesprochenen Probleme selten Hilfe, da nur sie sich im betroffenen Teilbereich auskennen; ich war der einzige, der Verständnis für die Java Sound-API hat, und erhielt daher keine Hilfe bei Fragen über die Funktionalität der Java Sound-API oder Kritik zu meinem Quellcode, der auf diese API oder auf Bestandteile des VoIP-Moduls zurückgreift. Außerdem wurde das Daily Standup Meeting von einem Betreuer beaufsichtigt. Ich konnte beobachten, dass sich einige Projektmitarbeiter unter Druck gesetzt gefühlt haben; sie sprachen ausschließlich mit dem Betreuer, obwohl das Daily Standup Meeting vorsieht, dass die Mitarbeiter ausschließlich untereinander kommunizieren. Ich selbst fühlte mich nicht unter Druck gesetzt, verstehe aber diejenigen, die sich so fühlten; denn die Betreuer griffen oft in das Meeting ein, indem sie mit den Mitarbeitern sprachen, sodass sie Einfluss auf die Mitarbeiter ausübten. Nach der ursprünglichen Idee des Projektleiters und der Betreuer, dieses Daily Standup Meeting einzuführen, sollten die Betreuer aber nicht eingreifen. Dadurch entsteht mein Eindruck, dass ein Mitarbeiter sich unter Druck gesetzt fühlen kann.

Während der letzten Woche eines Monats findet die Releasewoche statt. Ziel ist es, am Freitag dieser Woche eine neue Saros Version zu veröffentlichen. Der Projektleiter verteilt vier Rollen unter den Projektmitarbeitern: Releasemanager, Testmanager, sowie Assistant Releasemanager und Assistant Testmanager. Die Assistenten sind erfahrene Mitarbeiter, die die jeweilige Rolle bereits zuvor ausgeführt hatten, und stehen dem Release- beziehungsweise dem Testmanager unterstützend zur Seite.
Der Releasemanager verschiebt den aktuellen Inhalt des SVN-Trunks in einen Branch. Als nächstes sucht er nach Bugs im Bugtracker, die als kritisch markiert sind, führt die

Testsuite aus, sammelt die Fehlschläge und versendet die Ergebnisse dieser Suche über die Saros Mailing Liste. Am Ende der Releasewoche stellt er den Inhalt des Branches zum Download über die Updateseite oder als Archivdatei zum Entpacken in das Dropin-verzeichnis von Eclipse zur Verfügung und sendet die Änderungen seit dem letzten Release, die in der SVN-Historie enthalten sind, an die Saros Mailing Liste.
Der Testmanager leitet die Durchführung der praktischen Tests. Diese Tests sind in der Testlink Suite, einer Webapplikation zum Schreiben, Verwalten und Lesen von praktischen Testfällen, eingetragen. Ein praktischer Test ist folgendermaßen aufgebaut: Er hat einen Namen, der die zu testende Funktionalität der Software benennt, eine Beschreibung der zu testenden Funktionalität und eine Auflistung einzelner Teilaufgaben, denen Pseudonymen zugeordnet sind.
Die praktischen Tests werden von allen Projektmitarbeitern durchgeführt; dabei werden die Pseudonyme unter den Teilnehmern als Rollen verteilt, damit jeder weiß, wann er welche Teilaufgabe zu bearbeiten hat. Bugs, die zum Fehlschlagen eines Tests geführt haben, werden im Bugtracker notiert. Tests, die durch Änderungen in der Funktionalität obsolet geworden sind, werden vom Testmanager gelöscht.
Nach dem Durchführen der praktischen Tests werden die Bugs, die in der Email des Releasemanagers stehen, und die Bugs, die während der praktischen Tests gefunden wurden, von allen Projektmitarbeitern bearbeitet. Am Ende stellt der Releasemanager die neue Version wie beschrieben zum Download zur Verfügung.

Für eine Releasewoche wurde mir die Rolle des Testmanagers zugeteilt. Dabei habe ich die Erfahrung gemacht, dass die im Saros Projekt praktizierte Form des praktischen Testens sehr fehleranfällig ist; Tests müssen wiederholt werden, wenn ein einzelner Mitarbeiter einen Fehler gemacht hat oder Probleme hat, die nicht mit dem Testfall zusammenhingen. Dadurch nehmen diese Tests viel Zeit in Anspruch. Aber dadurch, dass die Zeit für das Testen auf einen Tag begrenzt ist, muss an dem Tag entweder deutlich länger gearbeitet werden, oder es müssen die verbleibenden Tests weggelassen oder am nächsten Tag nachgeholt werden. Werden die verbleibenden Tests nachgeholt, so fehlt die Zeit zur Bearbeitung der Bugs, da auch dafür nur begrenzte Zeit zur Verfügung steht. Es ist deshalb wünschenswert, dass künftig für eine Reduktion des Zeit- und Personenaufwandes im beschriebenen Prozess der praktischen Tests gesorgt wird.

Fazit und Ausblick

Das Ziel des Analyseteils meiner Arbeit war es, grundlegende Probleme im VoIP-Modul vorzustellen, zu analysieren und zu lösen. Nach diesem Vorgehen habe ich drei spezielle Probleme bearbeitet. Das Hardcoded-Audioformat-Problem bestand darin, dass festgelegte Audioeinstellungen inkompatibel zur Soundkarte eines Mitarbeiters sind. Gelöst wurde das Problem so, dass alternative Einstellungen bereitgestellt wurden, auf die im Fall eines Versagens zurückgegriffen werden kann.
Die Ursache des Streamservice-Problems, das während des Beendens einer VoIP-Sitzung eine Exception verursacht, konnte nicht in der Implementierung des VoIP-Moduls lokalisiert werden; die Ursache befindet sich außerhalb des Moduls. Der nächste Ansatzpunkt wäre diesbezüglich der Streamservice, der den betroffenen Stream bereitstellt.
Zum Problem der Latenzen wurden drei Stellen im Quellcode des VoIP-Moduls vorgestellt, und ich habe erörtert, inwiefern diese Stellen bezüglich des Auftretens von Latenzen als besonders kritisch gelten. Außerdem wurden Testverfahren vorgestellt, um zu testen, ob die Latenzen tatsächlich in der getesteten Stelle begründet sind.

Im zweiten Teil der Arbeit sollte das VoIP-Modul um eine Möglichkeit zur Konferenzschaltung erweitert werden. Dazu habe ich zuerst meine Gründe gegen eine Implementierung der Konferenzfunktionalität auf Basis der bestehenden VoIP-Funktionalität dargelegt. Das Ergebnis war, dass das Modul zurzeit noch Probleme aufweist, die vor einer Erweiterung behoben werden müssen, und dass eine etablierte VoIP-Technologie eine Konferenzfunktion mitliefern kann.
Danach habe ich fünf VoIP-Technologien vorgestellt. Ich habe erörtert, ob es sich um Komplettlösungen oder Protokolle handelt, ob es eine passend lizenzierte Implementierung in Java gibt, und wie diese Technologien das NAT-Problem lösen. Ich kam zu dem Schluss, dass sich SIP und IAX bezüglich der genannten Kriterien eignen; mit dem SIP-Communicator existiert eine Lösung, die aktiv weiterentwickelt wird und in Saros integriert werden kann. Auf Basis von IAX habe ich einen Prototyp aufgebaut, der in der Lage ist mit einer *Asterisk* Telefonanlage zu interagieren.

Zukünftige Entwicklungsschritte hängen davon ab, auf welcher Basis entwickelt wird; entscheidet man sich für die Integration der VoIP-Funktionalität des SIP-Communicators, steht eine Auswahl zusätzlicher Funktionalitäten wie eine Konferenzfunktion zur Verfügung. Zusätzlich können zukünftige Änderungen in Zusammenarbeit mit dem SIP-Communicator Projekt evaluiert und durchgeführt werden. Auf diese Weise steht zusätzliche Unterstützung für die Weiterentwicklung der VoIP-Funktionalität in Saros zur Verfügung.
Das Problem an der Einbindung des SIP-Communicators als VoIP-Lösung ist, dass der Benutzer vom Provider iptel.org abhängig ist; es ist also keine vollständig integrierte Lösung, da die Kommunikation von einem externen Provider vermittelt wird.

Auch mit der Einführung einer IAX-basierten Lösung in Saros stehen Möglichkeiten zur Verfügung, die Konferenzfunktionalität zu realisieren, obgleich diese selbst implementiert werden muss. Aber es existieren auch Open Source Anwendungen, die IAX unterstützen, beispielsweise *YATE* [26].

Das Problem an dieser Open Source Anwendung ist aber, dass sie nicht in Java, sondern in C++ geschrieben ist; ich konnte keine Implementierung einer IAX-basierten VoIP-Anwendung in Java finden. Außerdem ist das Einrichten und das Konfigurieren einer *Asterisk* Telefonanlage ein Problem; die Telefonanlage muss auf einem Rechner als Serveranwendung für die Benutzer von Saros zur Verfügung stehen; dies kostet dem Projekt Ressourcen auf dem betroffenen Rechner. Wenn kein Rechner für den Betrieb der Telefonanlage zur Verfügung steht, dann entstehen weitere Kosten durch die Beschaffung eines zusätzlichen Rechners.

Literaturverzeichnis

[1] E-Mail, von: C.Özbek, am: 23.04.2010, an Mailingliste "dpp-devel",
http://sourceforge.net/mailarchive/message.php?msg_name=op.vb
metpme5x5873%40thimphu.imp.fu-berlin.de , letzter Zugriff: 26.07.2010

[2] Website des iLBC Projektes, von: iLBCfreeware, Global IP Solutions, Stand: 2007,
http://ilbcfreeware.org/ , letzter Zugriff: 26.07.2010

[3] "Package net.java.sip.communicator.impl.neomedia.codec.audio.ilbc", von:
SIP Communicator Projekt, Stand: 2009,
http://bluejimp.com/sip-communicator/api/, letzter Zugriff: 26.07.2010
http://sip-communicator.org/ , letzter Zugriff: 26.07.2010

[4] JAIN-SIP Projektwebsite, von: M. Ranganathan, Phelim O'Doherty, Stand: 2010
https://jain-sip.dev.java.net/ , letzter Zugriff: 26.07.2010

[5] Eintrag "H.323", in: voip-info.org Wiki, Stand: 28.10.2009,
http://www.voip-info.org/wiki/view/H.323 , letzter Zugriff: 26.07.2010

[6] Eintrag "SIP", in: voip-info.org Wiki, Stand: 04.12.2009,
http://www.voip-info.org/wiki/view/SIP , letzter Zugriff: 26.07.2010

[7] "Various Licenses and Comments about Them", von: GNU Project,
Free Software Foundation, Stand: 03.07.2010,
http://www.gnu.org/licenses/license-list.html , letzter Zugriff: 26.07.2010

[8] "Frequently Asked Questions about the GNU Licenses",
von: GNU Project, Free Software Foundation, Stand: 03.07.2010,
http://www.gnu.org/licenses/gpl-faq.html#AllCompatibility ,
letzter Zugriff: 26.07.2010

[9] Eintrag "NAT and VoIP", in: voip-info.org Wiki, Stand: 16.04.2010
http://www.voip-info.org/wiki/view/NAT+and+VOIP , letzter Zugriff: 26.07.2010

[10] Eclipse Communication Framework Dokumentation, von: The Eclipse Foundation,
Stand: 2010, http://www.eclipse.org/ecf/documentation.php ,
letzter Zugriff: 26.07.2010

[11] Eintrag "VoIP via the ECF Call API and the Jingle Protocol",
von: Moritz Post, in: Eclipse Wiki, Stand: 14.10.2007,
http://wiki.eclipse.org/VoIP_via_the_ECF_Call_API_and_the_Jingle_Protocol ,
letzter Zugriff: 26.07.2010

[12] "Java SE Desktop Technologies – Java Media Framework API (JMF)",
von: Oracle, Sun Developer Network (SDN), Stand: 2010,
http://java.sun.com/javase/technologies/desktop/media/jmf/ ,
letzter Zugriff: 26.07.2010

[13] XMPP Spezifikation XEP-0166: Jingle, von: Scott Ludwig et al., Stand: 23.12.2009
http://xmpp.org/extensions/xep-0166.html#intro , letzter Zugriff: 26.07.2010

[14] Skype4Java Projektseite, von: Koji Hisano et al., Stand: 24.07.2010
http://de.sourceforge.jp/projects/skype/ , letzter Zugriff: 26.07.2010

[15] "Java Specification Requests – JSR81: JAIN H323", von: Orit Levin, Stand: 2000
http://jcp.org/en/jsr/detail?id=81 , letzter Zugriff: 26.07.2010

[16] "RFC 3261 – SIP: Session Initiation Protocol", von: J. Rosenberg et al, Internet
Engineering Task Force, Stand: Juni 2002,
http://tools.ietf.org/html/rfc3261 , letzter Zugriff: 26.07.2010

[17] "Header-Format einer SIP Nachricht", von: T. Kröner, Stand: 2010
http://www.voip-information.de/sip/header-format.php , letzter Zugriff: 26.07.2010

[18] "RFC 4566 – SDP: Session Description Protocol", von: M. Handley,
V. Jacobson, C. Perkins, Internet Engineering Task Force, Stand: Juli 2006,
http://tools.ietf.org/html/rfc4566 , letzter Zugriff: 26.07.2010

[19] "RFC 3550 – RTP: A Transport Protocol for Real-Time Applications",
von: H. Schulzrinne et.al., Internet Engineering Task Force, Stand: Juli 2003,
http://tools.ietf.org/html/rfc3550 , letzter Zugriff: 26.07.2010

[20] "JAIN General Q&A", von: Oracle, Sun Developer Network, Stand: 2010,
http://java.sun.com/products/jain/qa.html , letzter Zugriff: 26.07.2010

[21] "JAIN SIP Tutorial – Serving the Developer Community", von: M. Ranganathan,
Phelim O'Doherty, Stand: 18.08.2009,
http://www-x.antd.nist.gov/proj/iptel/tutorial/JAIN-SIP-Tutorialv2.pdf ,
letzter Zugriff: 26.07.2010

[22] SIP-Communicator Website, von: Emil Ivov et.al, Stand: 2010,
http://sip-communicator.org/ , letzter Zugriff: 26.07.2010

[23] njiax Projektwebsite, von: Nomasystems, mrrubinos, mpquique, emiliano.rguez,
Stand: 2007, http://code.google.com/p/njiax/ , letzter Zugriff: 26.07.2010

[24] "IAX IAX2 Architecture: Inter-Asterisk eXchange Protocol",
von: Projekt VoIP Think, Stand: 2010,
http://www.en.voipforo.com/IAX/IAX-architecture.php ,
letzter Zugriff: 26.07.2010

[25] Eintrag "Asterisk", in: voip-info.org Wiki, Stand: 24.07.2010,
http://www.voip-info.org/wiki/view/Asterisk , letzter Zugriff: 26.07.2010

[26] YATE Telephony Engine Website, von: S.C. Null Team Impex SRL,
Stand: 16.07.2010 , http://yate.null.ro/pmwiki/ ,
letzter Zugriff: 26.07.2010

[27] "Speex – Plugins and other Software", von: xiph.org Foundation, Stand: 2006
http://www.speex.org/software/ , letzter Zugriff: 26.07.2010

[28] Bachelorarbeit "Verbesserung der Kommunikationsmöglichkeiten in einem Werkzeug zur verteilten Paarprogrammierung (Saros)", von: O.Loga,
Datum: 08.04.2010,
http://www.inf.fu-berlin.de/inst/ag-se/theses/Loga10-DPPXV.pdf ,
letzter Zugriff: 26.07.2010

[29] Eintrag "Open Source VOIP Software", in: voip-info.org Wiki, Stand: 21.07.2010
http://www.voip-info.org/wiki/view/Open+Source+VOIP+Software ,
letzter Zugriff: 26.07.2010

[30] An Introduction to the JAIN SIP API, von: Emmanuel Proulx, Oracle,
Stand: 17.10.2007,
http://www.oracle.com/technology/pub/articles/dev2arch/2007/10/introduction-jain-sip.html , letzter Zugriff: 26.07.2010

[31] Foliensatz der Vorlesung "Multimediakommunikation - 7. Session Initiation Protocol (SIP)", von: Dr.-Ing. D. Schuster, Stand: 10.07.2010,
http://www.rn.inf.tu-dresden.de/lectures/Multimediakommunikation/MMK-07_SIP.pdf , letzter Zugriff: 26.07.2010

[32] Eintrag "Asterisk config extensions.conf", in: voip-info.org Wiki, Stand: 05.06.2010
http://www.voip-info.org/wiki/view/Asterisk+config+extensions.conf ,
letzter Zugriff: 26.07.2010

[33] Eintrag "Asterisk config iax.conf", in: voip-info.org Wiki, Stand: 27.05.2010
http://www.voip-info.org/wiki/view/Asterisk+config+extensions.conf ,
letzter Zugriff: 26.07.2010

[34] Java Sound-API, von: Sun Microsystems, Oracle, Stand: 25.06.2003,
http://java.sun.com/j2se/1.4.2/docs/api/javax/sound/sampled/package-summary.html , letzter Zugriff: 27.07.2010

[35] NJIAX Javadoc Dokumentation aus dem Quellcode, von Nomasystems, mrrubinos, mpquique, emiliano.rguez, Stand: 2007

[36] "Was ist agile Softwareentwicklung?", von: it-agile GmbH,
http://www.it-agile.de/wasistagilesoftwareentwicklung.html ,
letzter Zugriff: 27.07.2010

[37] JSpeex Dokumentation, Stand: 13.01.2004,
http://jspeex.sourceforge.net/doc/index.html , letzter Zugriff: 28.07.2010